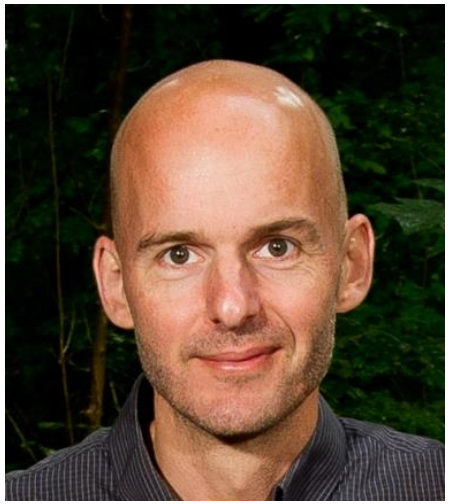
HARDWARE ABSTRACTIONS
- AN INTRODUCTION

WHO'S RESPONSIBLE?

Main Teacher:

- Peter Høgh Mikkelsen (phm@ece.au.dk)
- 11 years in industry (Embedded HW/SW/FPGA)
- 14 years at ECE/AU
- 5123-424



- + TAs

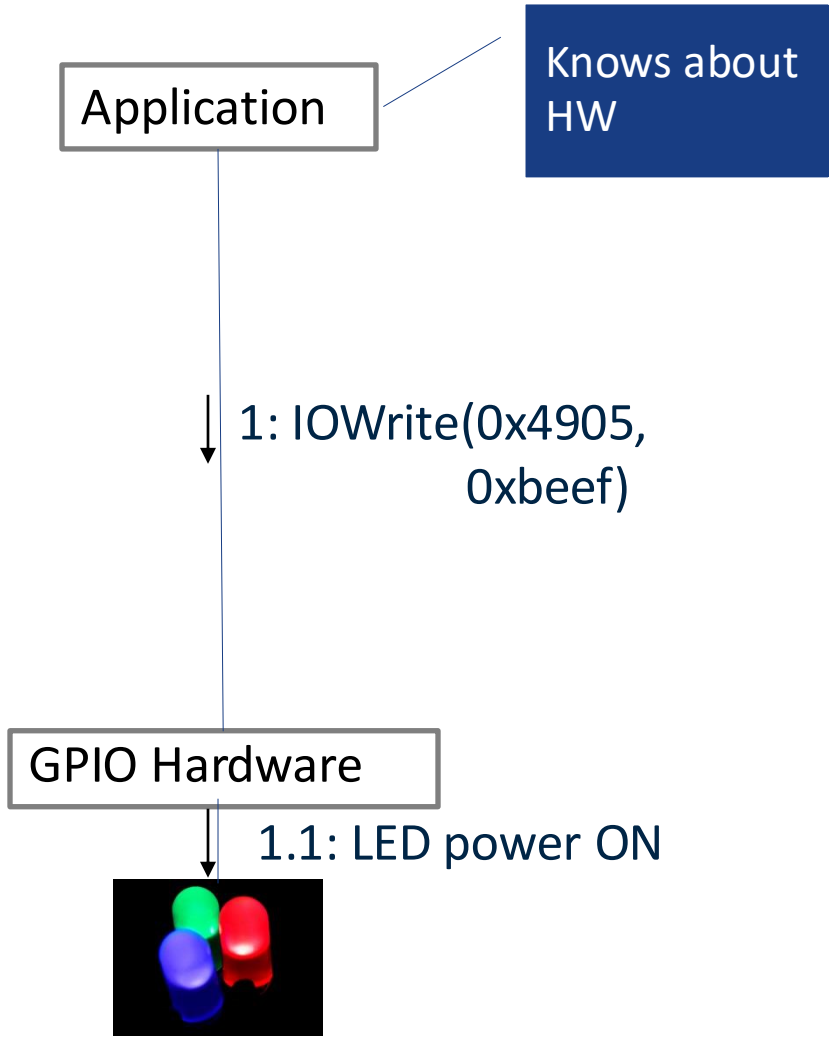
ABSTRACTION?

“A thing is what it is in virtue of its essence."

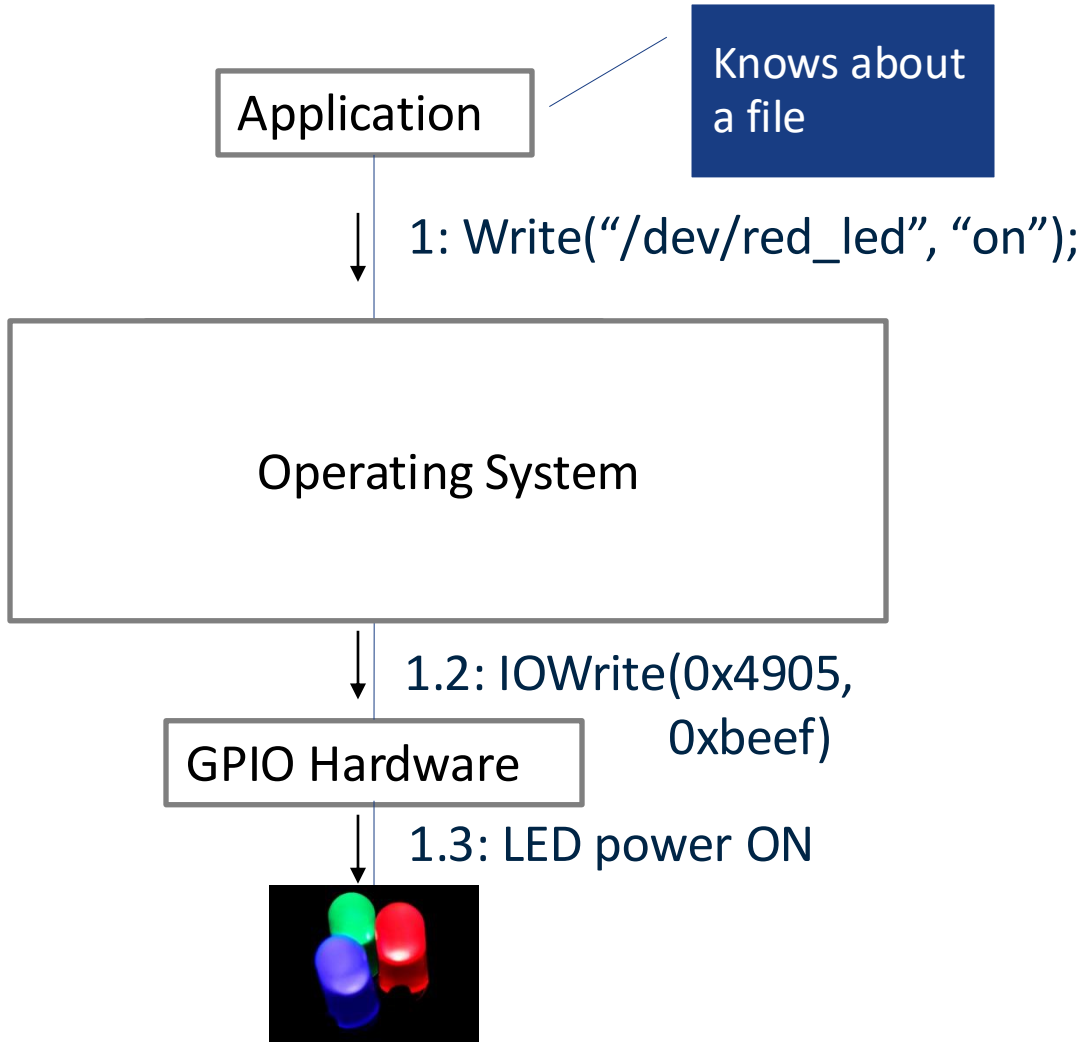
Aristotle, Metaphysics, Book VII

Henri Matisse
The Cat with Red Fish

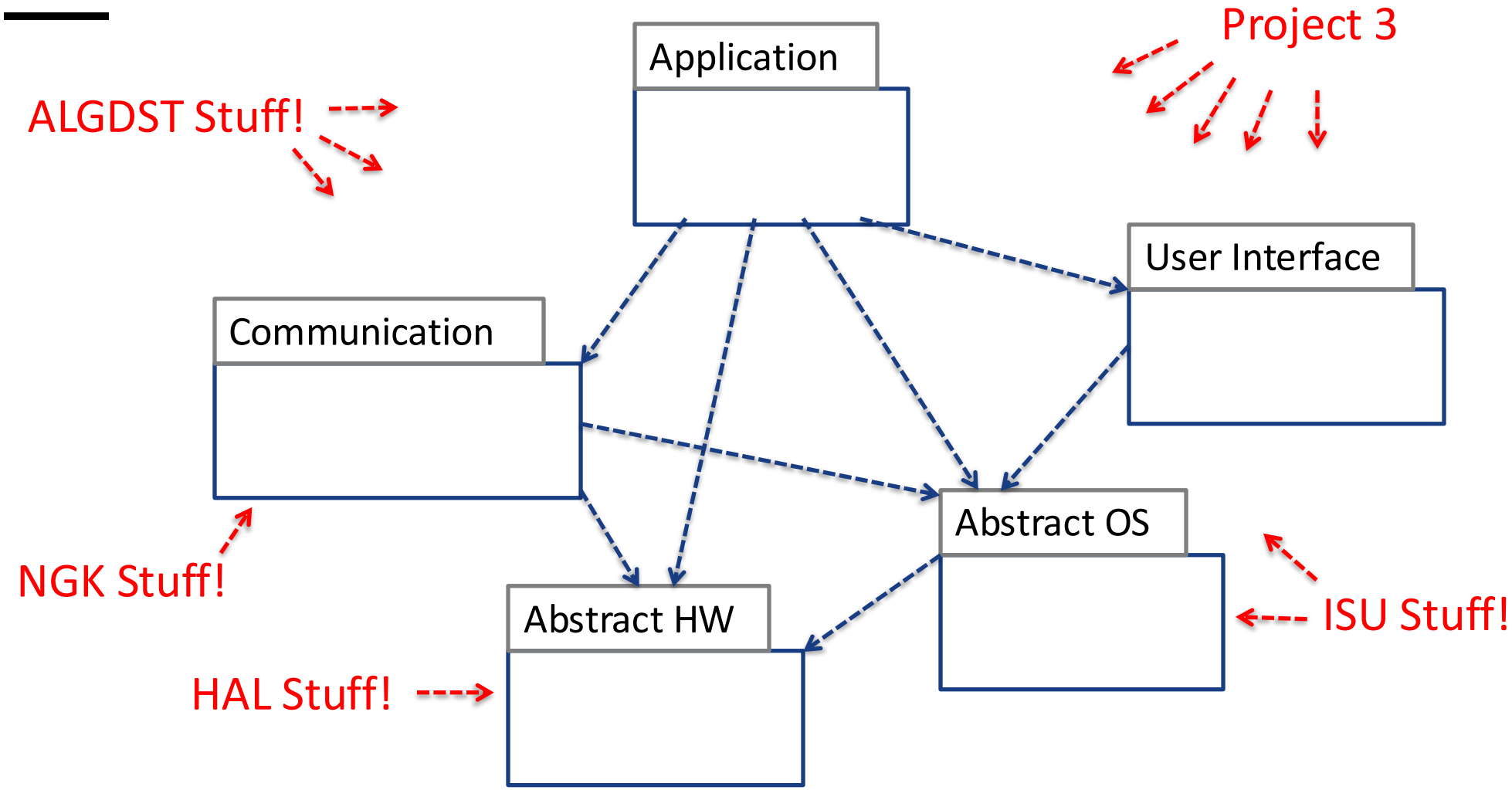
Bare Metal



Abstracted HW

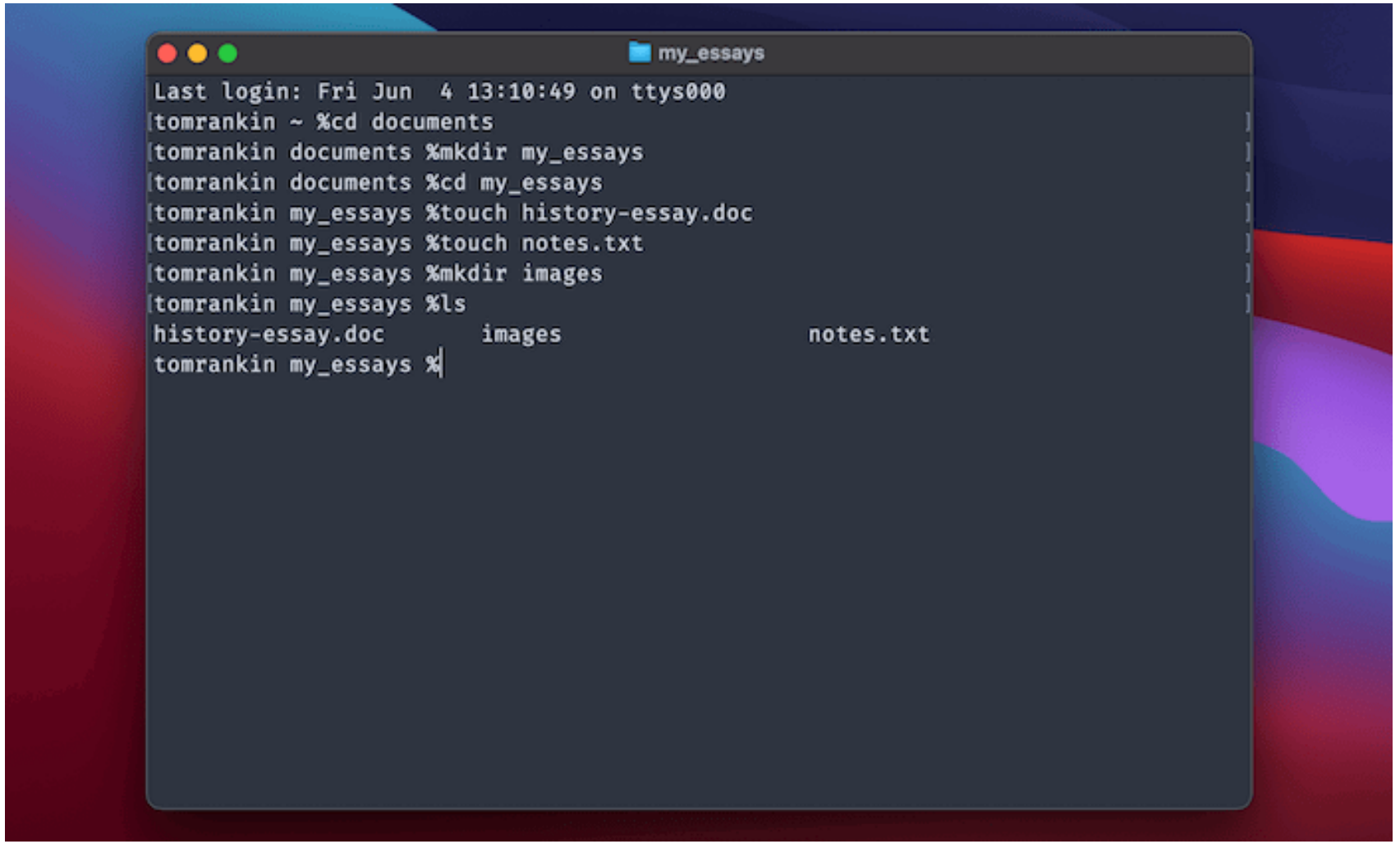


HAL IN SOFTWARE

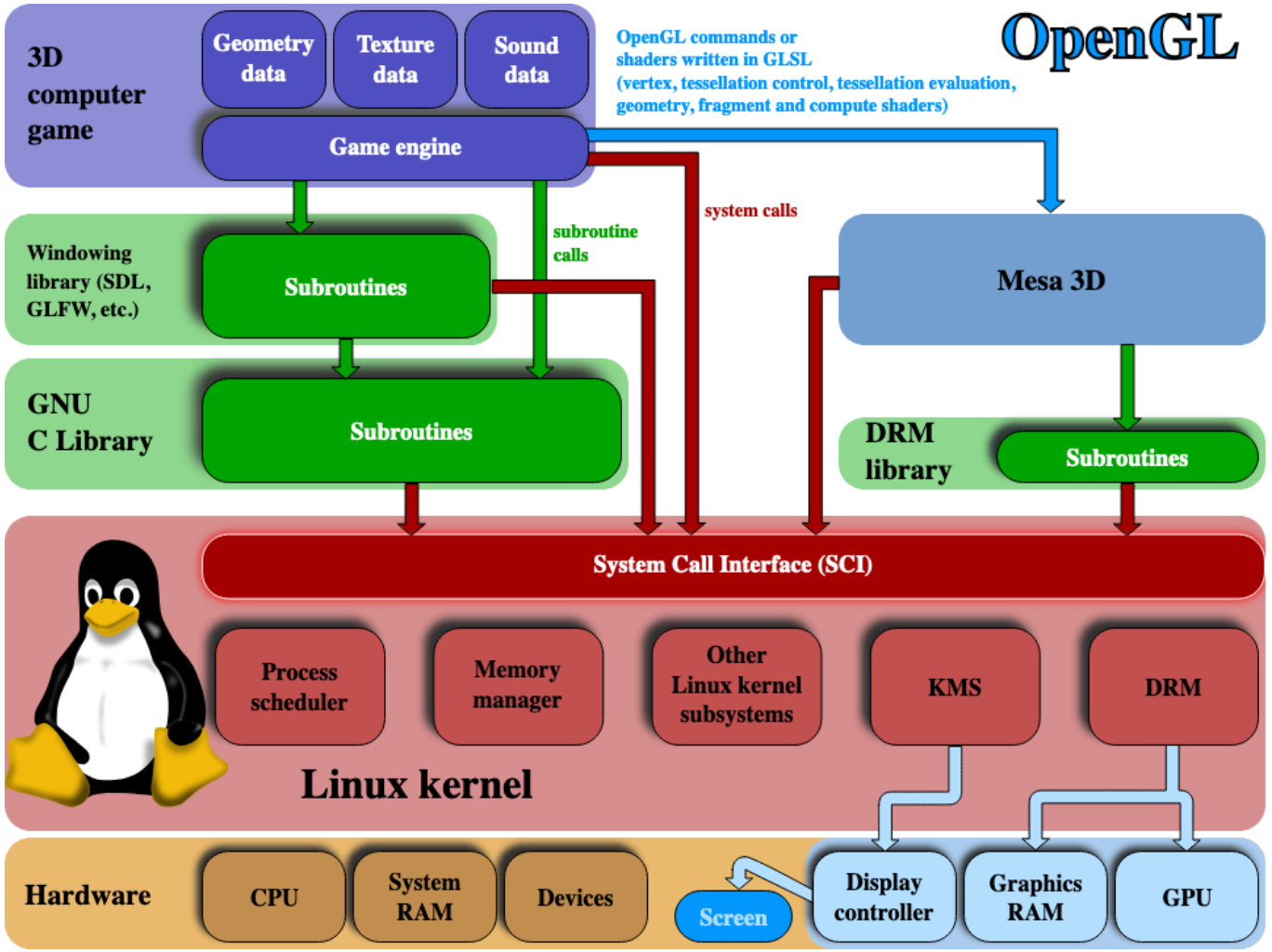


Five-Layer Architecture Pattern

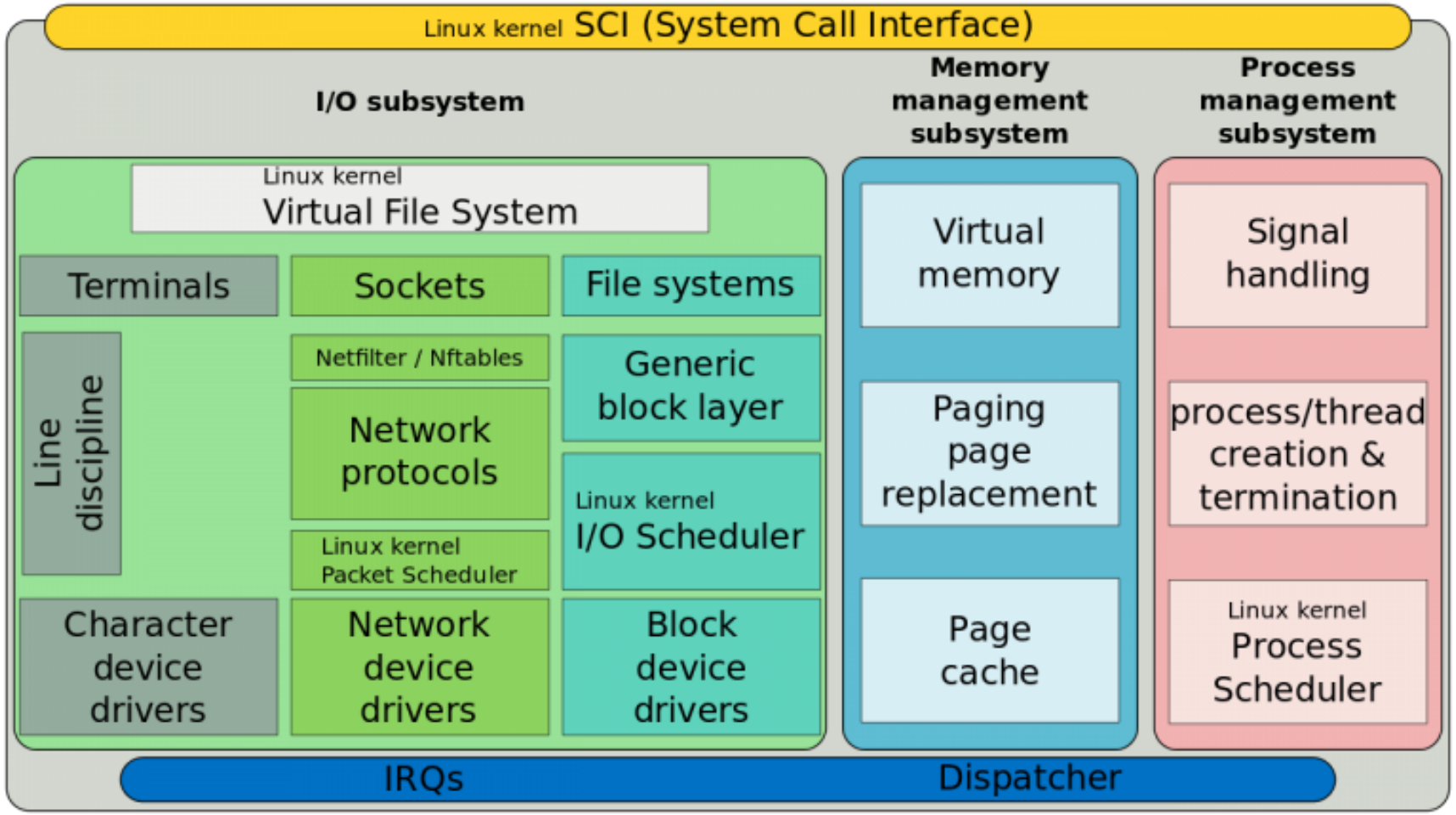
01: INTRODUCTION TO LINUX



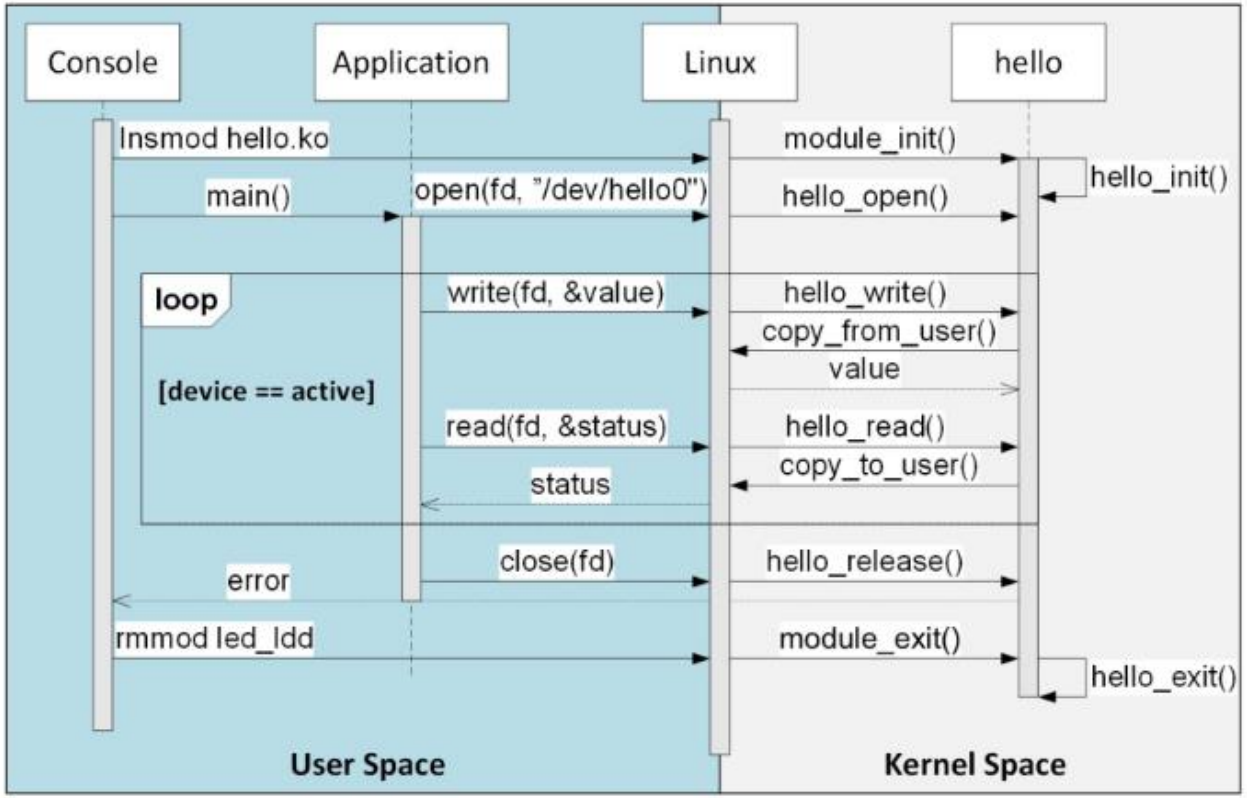
02: KERNEL INTERFACE



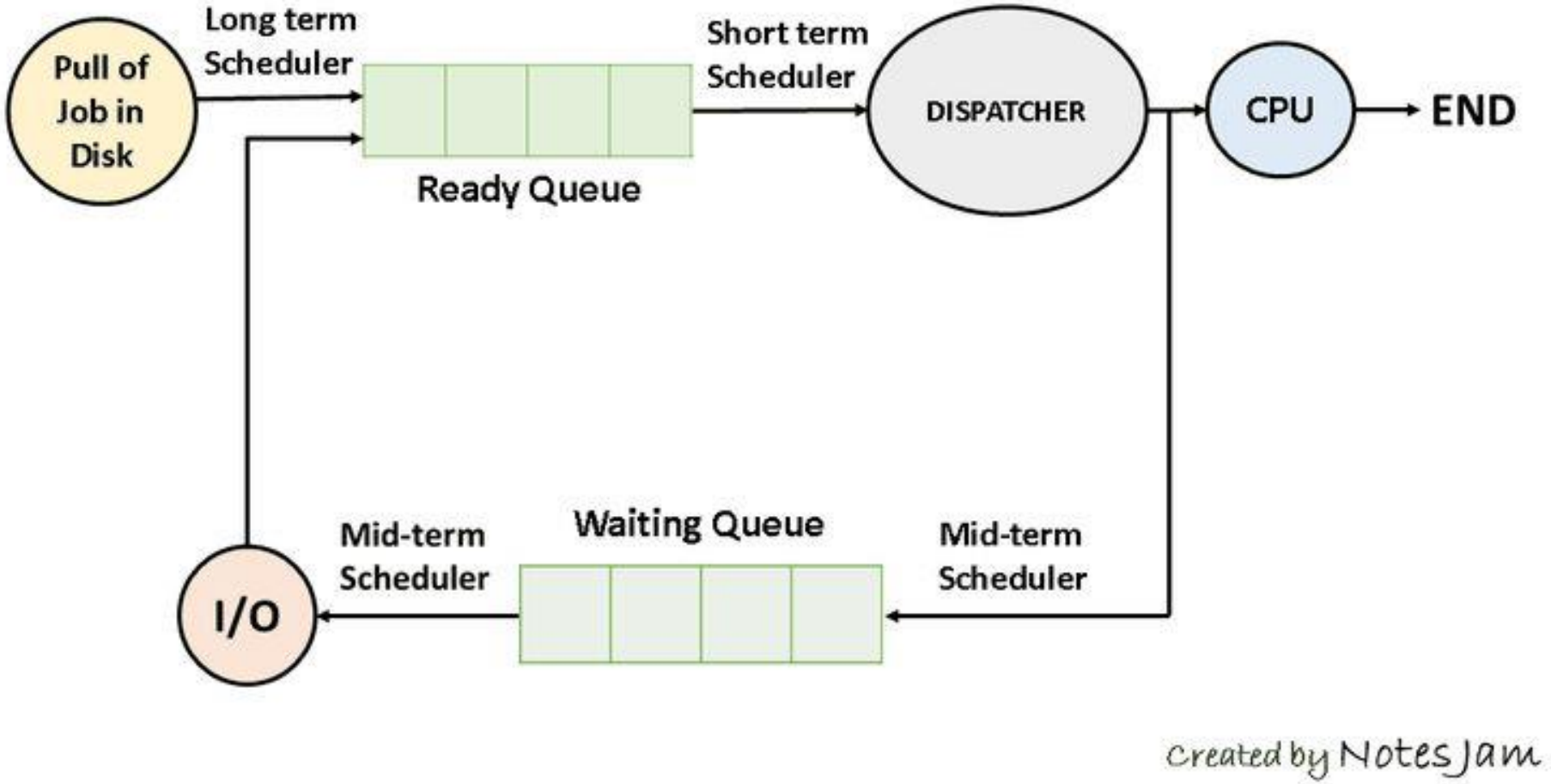
03: LINUX KERNEL AND KERNEL MODULES



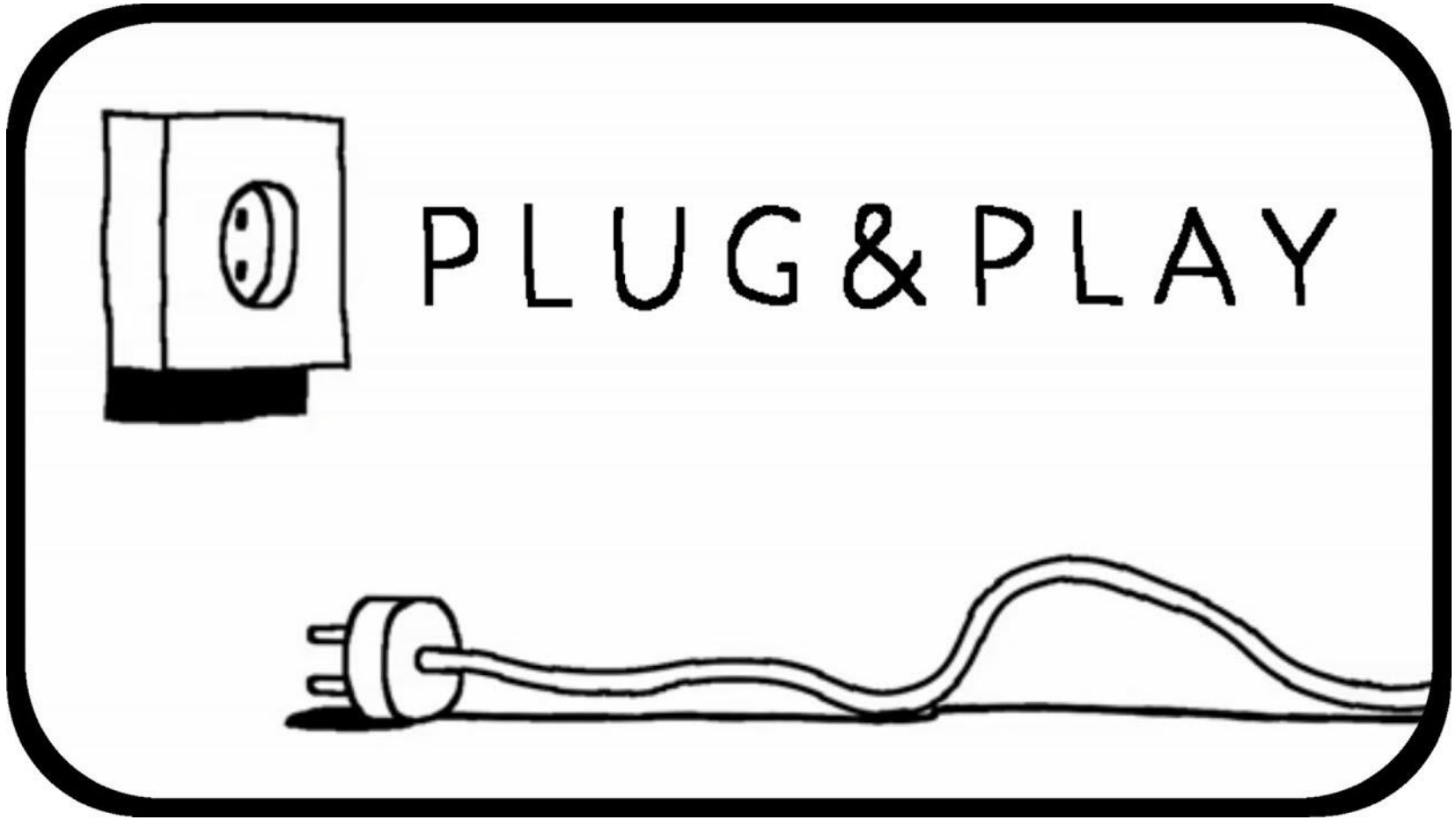
04: CHARACTERS DRIVERS



05: INTERRUPTS AND BLOCKING I/O

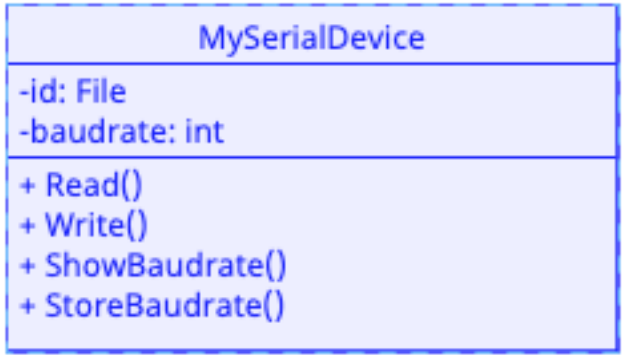


06: LINUX DEVICE MODEL AND TREE

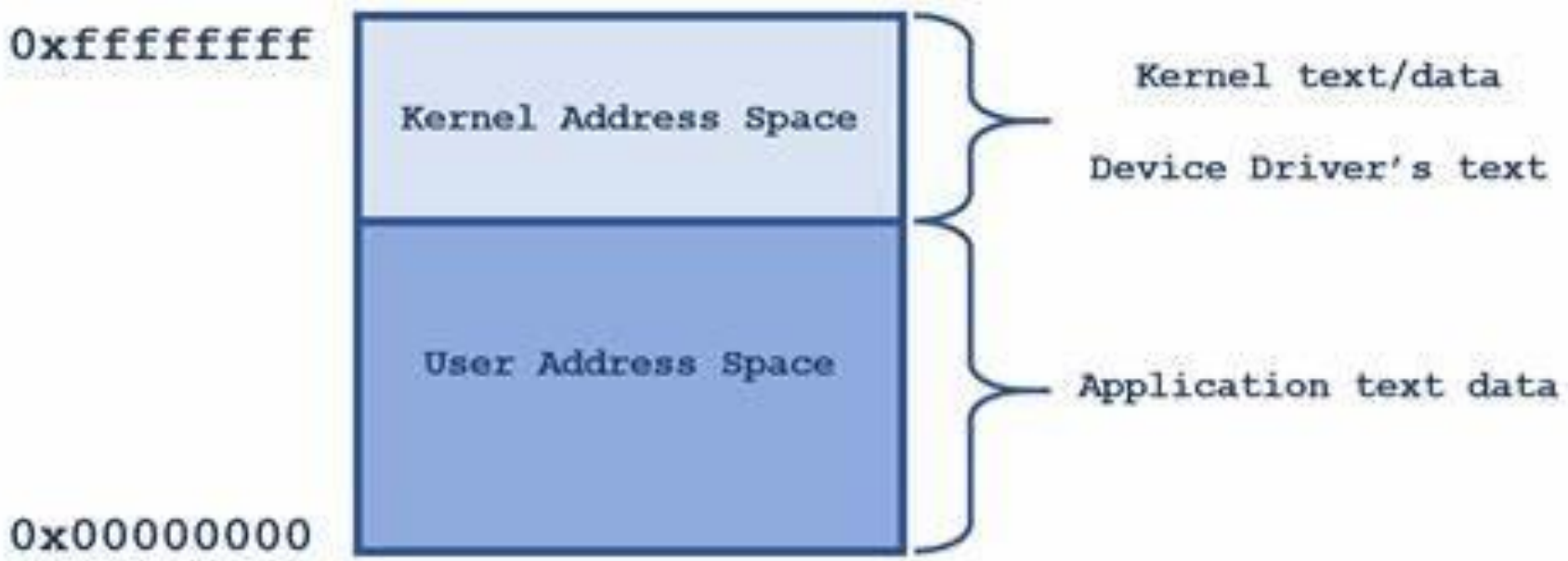


07: LINUX BUS FRAMEWORKS

08: TIMERS, DELAYS (+ ATTRIBUTES)



09: MANAGING MEMORY



EXERCISES

Groups of 2-3 pers.

Feedback Fruits for hand-in and peer reviewing

Approved exercise work + approved review work = approved exercise

Three exercises must be approved to enter exam.

HAND-IN PROCEDURE

- See Lecture Plan on BS for exact dates

SW3HAL_Exercises_E2023 : Schedule

Week	Date	Begin	End	Room	Lecture	Exercise	Hand-in	Review	Commer
35	31.08.2023	12:15	15:50	1593-002	1	1			
36	07.09.2023	12:15	15:50	1593-002	2	2			
37	14.09.2023	12:15	15:50	1593-002	3	3			
38	21.09.2023	12:15	15:50	1593-002	4	4			
39	28.09.2023	12:15	15:50	1593-002		4			
40	05.10.2023	12:15	15:50	1593-002	5	5	4		
41	12.10.2023	12:15	15:50	1593-002		5		4	
42	19.10.2023	Efterårsferie							
43	26.10.2023	12:15	15:50	1593-002	6	6			
44	02.11.2023	12:15	15:50	1593-002		6			
45	09.11.2023	12:15	15:50	1593-002	7	7	6		
46	16.11.2023	12:15	15:50	1593-002		7		6	
47	23.11.2023	12:15	15:50	1593-002	8	8			
48	30.11.2023	12:15	15:50	1593-002		8			
49	07.12.2023	12:15	15:50	1593-002	9	9	7/8		
50	14.12.2023							7/8	Ingen U

Work on ex 4 here

Hand-in ex 4 here

Hand-in ex4 peer review here

- Procedure is described in more detail here: https://gitlab.au.dk/au-ece-sw3hal/exercises

EXAM

2-hour Written Exam:

70% Multiple-Choice

30% Coding / Explain something essay-style

Work with exercises during the course to pass exam!

EXECUTION

At home: Read curriculum, watch videos, and post questions on Discord (Brightspace->Course Links)

In class: Brief Introduction, answer questions and introduce exercise. Work on exercises! (This is where you can get help!!)

On double lectures: Initial follow-up on general questions before exercise work

HOW TO WORK

This course requires an effort to pass!

- Expect and plan for 10 hours/week
- This semester you must study yourself!

- Third semester is hard, seek aid (fellow students / teachers / counselors) if you fall behind.

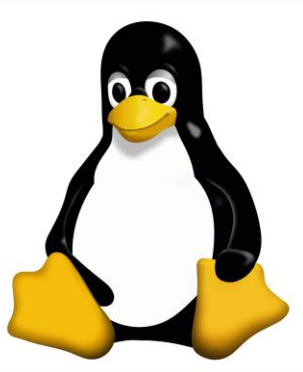
- If you spend HAL lecture hours on other course's work, then you're probably sidetracking. Lecture hours are where you can get (almost immediate) aid and guidance!!

- Teamwork is important, see BS – "Group Work" for details on how to create groups and what to do if things are not working

AARHUS
UNIVERSITY

HARDWARE ABSTRACTIONS
- INTRODUCTION TO LINUX

WHAT IS LINUX



- Linux is an operating system kernel, a “free” Unix clone developed in early 1991 as a hobby project by Linus Thorvalds

- People (typ. Minix/Unix-programmers) around the world soon picked up, developing drivers for different hardware
- Linux was designed from the beginning to be a free OS
  - Free as in “free speech”, not necessarily “free beer”
- Linux gained momentum in academic circles, since it was simple, geeky, and could be changed to fit particular needs

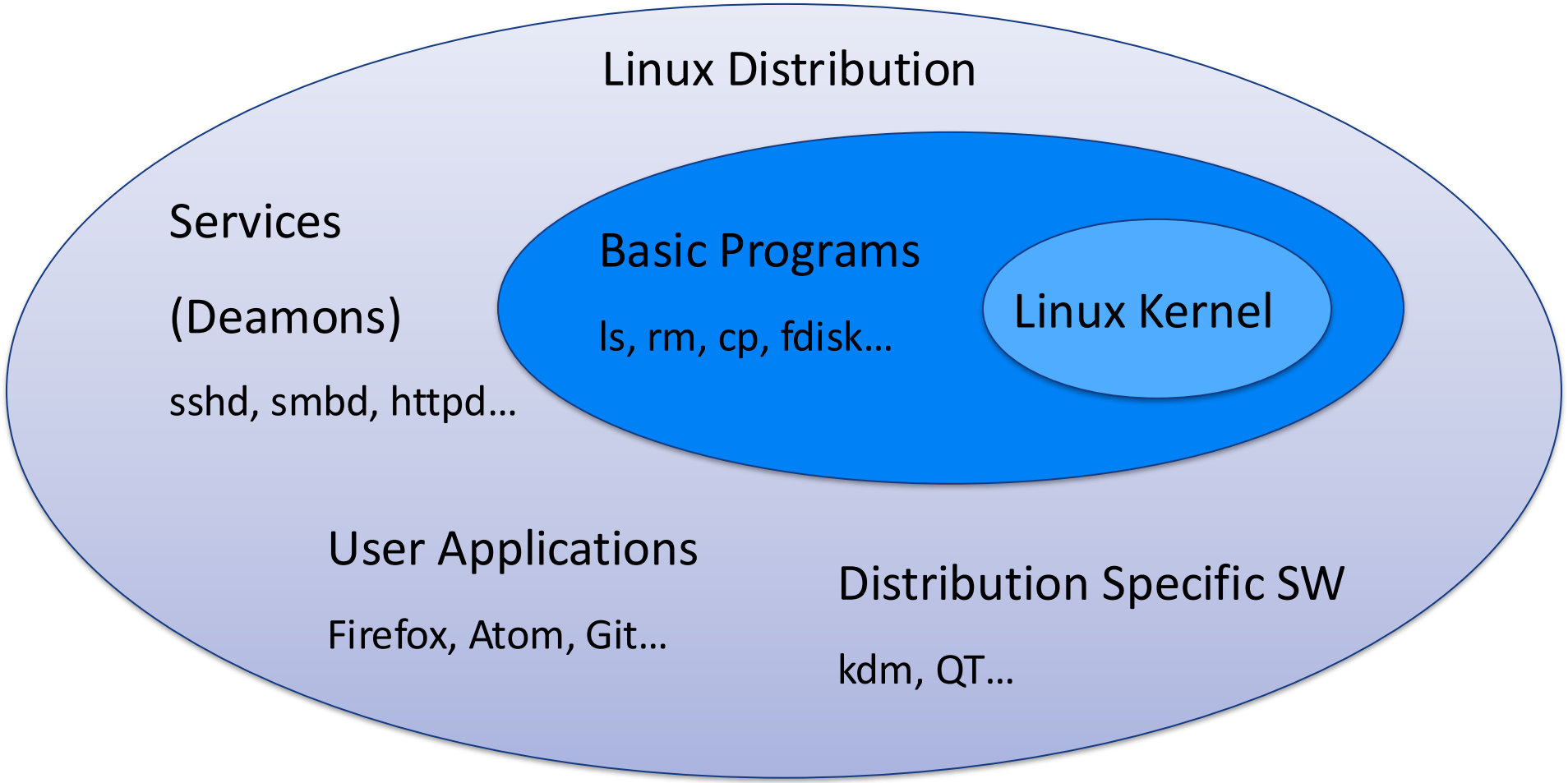
- Today, it is a mature, accessible OS for many platforms
  - 40 mio lines of code (6.14 / 2025), 75.000 commits in 2019, 4189 contributors!

WHAT IS GNU?

- GNU = GNU's Not Unix! (www.gnu.org)

- The GNU project was launched by the Free Software Foundation (FSF) to create a free UNIX-like system (kernel and tools)
  - However, when the tools were ready, the kernel was not
- GNU teamed with Linux to create GNU/Linux, the most common baseline distribution
- The GNU tools are:
  - gcc/g++ - GNU Compiler Collection, incl. GNU C/C++ Compiler
  - gdb - GNU debugger
  - make – GNU make utility
  - bash - A shell / terminal
  - emacs – A text editor

WHAT IS A LINUX DISTRIBUTION?



LINUX DISTRIBUTIONS

- You can obtain the Linux kernel source code (www.kernel.org) compile and run it on your system, then obtain the tools etc. to build a complete distribution

- Build your own Linux distribution
  - https://www.yoctoproject.org/

- There are many different distributions
  - http://distrowatch.com/
  - Find whichever flavor that suits you the best
  - You're left with the choice…

- We will use Lubuntu on the host (PC) and Yocto Linux on the target (Raspberry Pi 0 W)

REASONS TO USE LINUX (AND NOT)

- Pros
  - Open Source (Transparant source code, open to debug/discuss)
  - Mature (Very stable)
  - Supported (Through world-wide net of developers)
  - Portable (Supports a wide number of architectures: x64, ARM, PPC, MIPS…)
  - Scalable (From small-embedded to server parks)
  - Puts you in charge! (You have the power to configure most things)
- Cons
  - Open Source (Added kernel functionality must be publized)
  - Hard to learn and configure
  - Many flavors and choices
  - Puts you in charge! (You may not be capable…)
  - …It may be a bit overwhelming at first!

LINUX IN HAL (AND ISU)

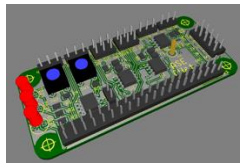
VM-Ware Image

- LUbuntu distribution + tools:
  - Toolchain for bulding Rpi applications + drivers
  - VS Code, Terminal etc.
- User: stud, Password: stud
- Image: ***DeveloperVM.7z***

USB Ethernet/

(UART)

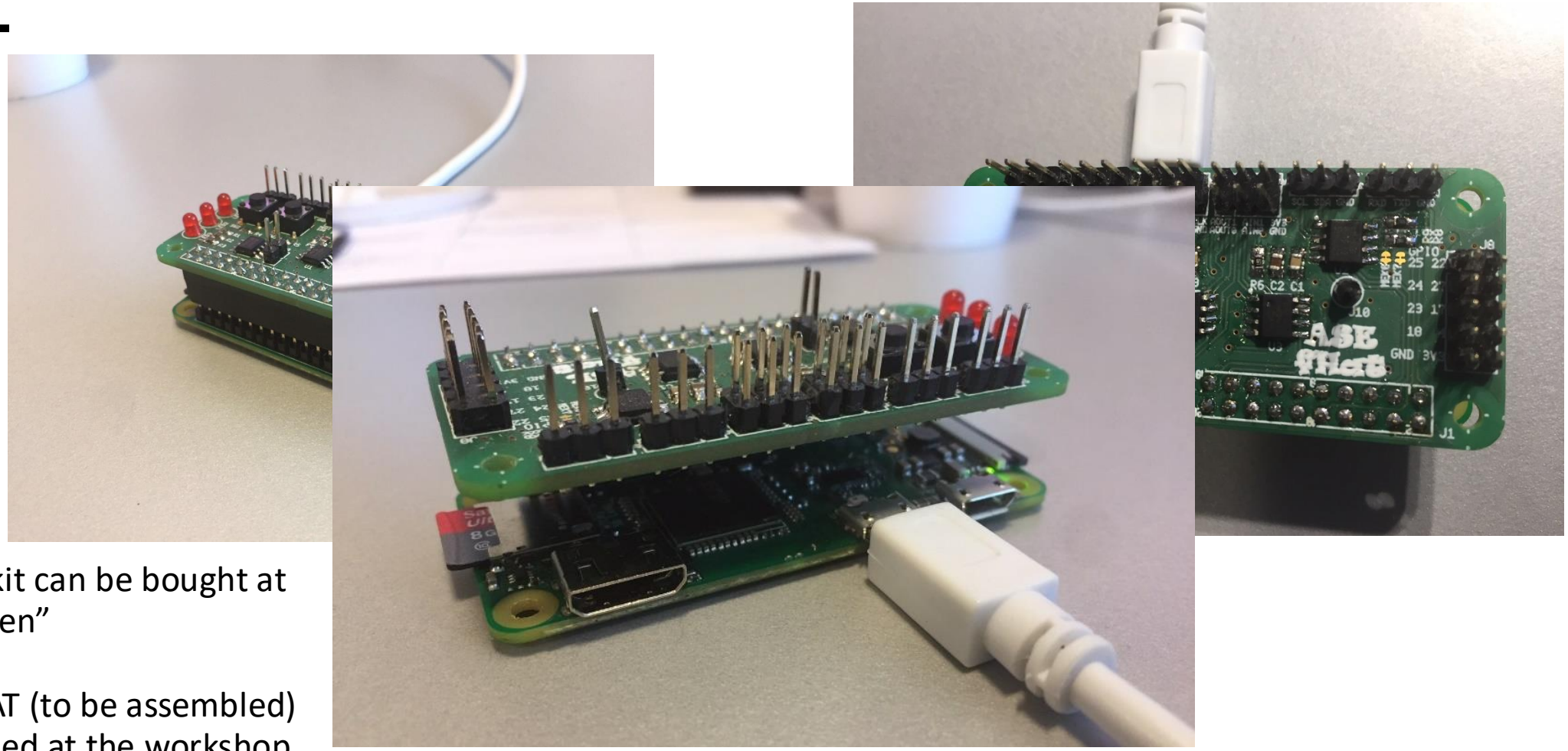
Rpizw + ASE fHAT



- Yocto-based distribution:
  - Shell + all basic tools
  - QT Libraries (To support graphics)
  - C/C++ Libraries
  - No GUI!
- User: root, Password: <none!>
- Image: ***rpizerowifi-image-raspberrypi0-wifi.rpi-sdimg***

Image Location: https://redweb.ece.au.dk/files/ (VPN)

RPI 0 W + ASE FHAT



- RPI0W kit can be bought at “Bogladen”
- The FHAT (to be assembled) is supplied at the workshop in the Shannon building

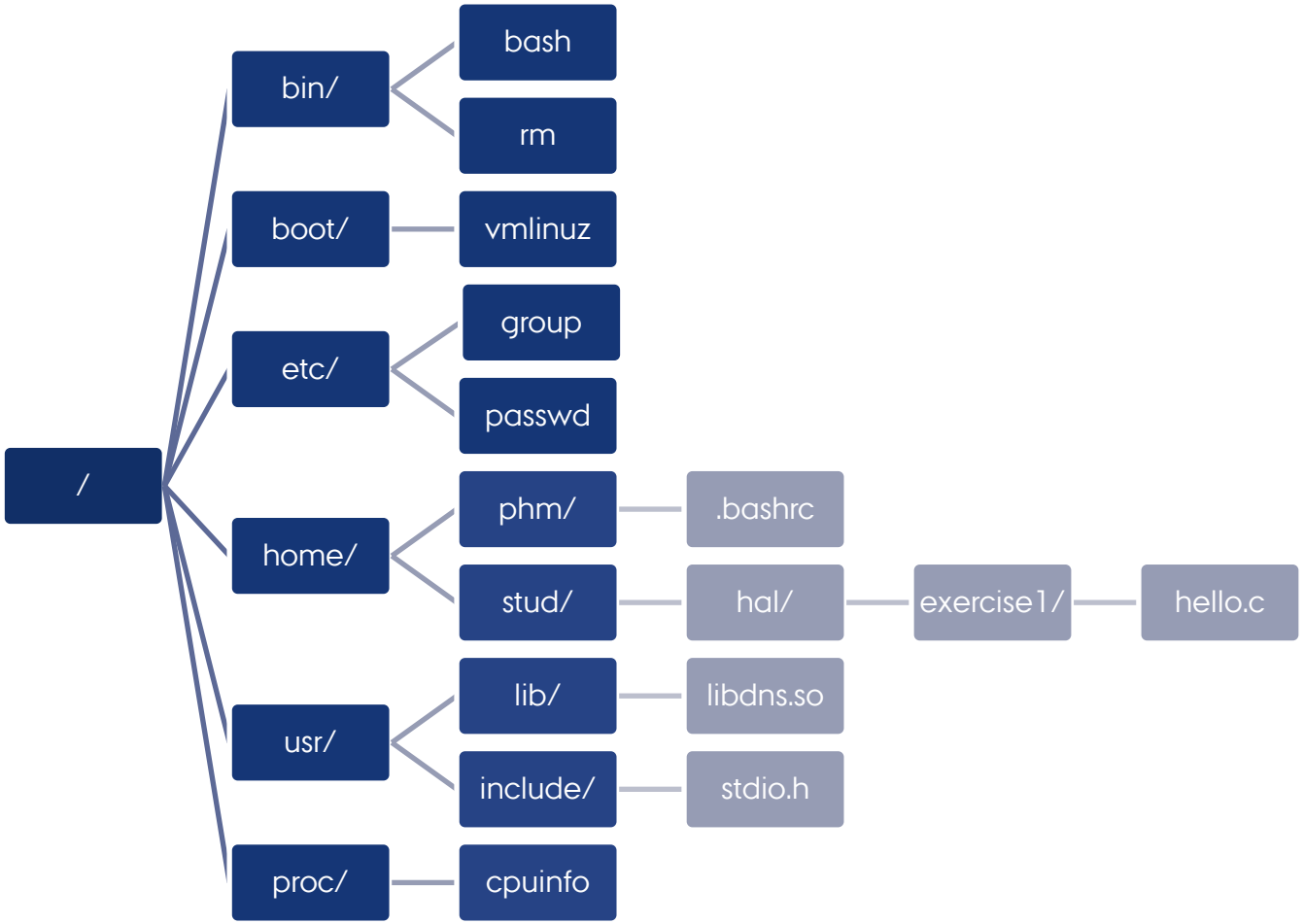
AARHUS
UNIVERSITY

HARDWARE ABSTRACTIONS
- LINUX AND FILES

LINUX AND FILES

- “Everything in Linux is a file. If something is not a file, it’s a process”
- A file can be…
  - Regular files (text or binary) or directories
  - Special files (devices/drivers)
  - Sockets (Network connections)
  - Named pipes (Communication channel between running programs)
  - Process and system information
- The file concept is a common abstraction for interaction with the system (Linux) and it’s devices
  - Storing text in file is done using file I/O
  - Sending a document to a printer is done using file I/O
  - Reading from a network socket is done using file I/O
  - Turning on a LED is done using file I/O

LINUX DIRECTORY STRUCTURE



Executables accesible by user

Boot Images

System Configuration

Home Directories

Data/Libraries accessible by user

System Information

USERS AND PERMISSIONS

- Since everything is files, user permissions and file permissions are tightly coupled in Linux (Unix)

- All files have ownership and permissions:
  - Permissions: Read / Write / Execute
  - Ownership: Owner / Group / Others

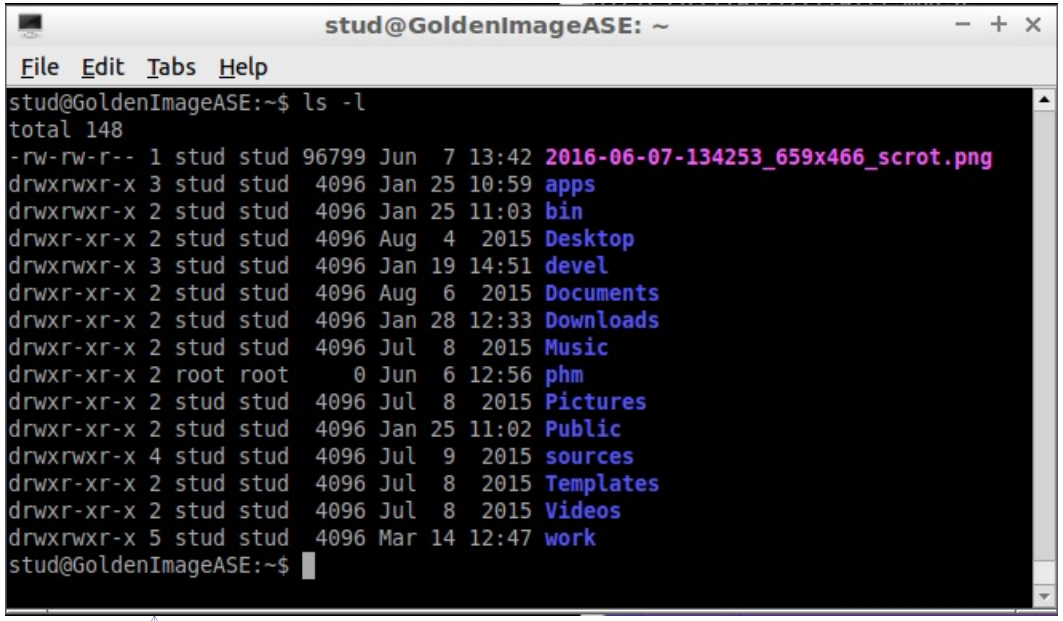
- Users can be created and added to groups, with different access rights.
- Users can also achieve ownership of files and directories

THE SHELL

- The Shell aka Terminal aka Console aka Prompt is the most direct way to use Linux:
  - Input command – Execute – Input next command…

- It allows you to:
  - Run programs
  - Compile programs
  - View file contents
  - File/Folder manipulation
  - Create scripts
  - Combine commands

- Ubuntu uses a Bash shell



user@machine: prompt

BASH COMMANDS (1:3)

- List Files

```
ls [options] [file] // Remember! directories are also files!
ls -l   // Long List including ownership and filetype (folder, link…)
ls -a   // All files including hidden ones (leading . in names)
ls -lt  // Long sorted according to time stamp
```

- Change Directory

```
cd [directory]
cd /usr/bin     // Change to an absolute path
cd ~            // Change to home directory of current user
cd ..           // Change to one directory up
```

- Copy Files

```
cp [from] [to]
cp fileA fileB          // Make a copy of fileA named fileB
cp fileA ~/work/        // Copy fileA to users work subfolder
cp -p fileA fileB       // Preserve timestamp
```

BASH COMMANDS (2:3)

- Search for files:

```
find [base folder] [search pattern]
find /usr/include -name "s*io.h"
```

- Search for text in files:

```
grep [file] [string]
grep /usr/include/stdio.h "putchar"
```

- Find header files and use pipe ("|") to view output in the "less" text viewer (/=search, q=exit):

```
find /usr/include -name "*.h" | less
```

- Look for string in all header files in folder (and subfolders):

```
find /usr/include -name "*.h" | xargs grep "putchar"
```

- ... and view result in viewer:

```
find /usr/include -name "*.h" | xargs grep "putchar" | less
```

BASH COMMANDS (3:3)

Change Ownership:

```
chown [user] [file] //Change ownership of file
chown stud readme.txt
```

Change Permissions:

```
chmod u+w readme.txt // Give user write permissions
chmod 755 my_script.sh // rwx r-x r-x (see octal permissions)
```

Create a script:

```
echo "dir -a" > my_script.sh // Create file containing dir command
chmod u+x my_script.sh // Make script executable
./my_script.sh // Execute script ./ because it is in current folder
```

HELP YOURSELF WITH LINUX

Most questions about shell commands and Linux in general have already been asked on the internet!

Find the information yourself!!

- Google, Stackoverflow etc!!!

- Linux has built-in help:
  - [man]
  - [info]

```
man ls
info find
```

- Commands also have built-in help:

```
ls --help
```

AARHUS
UNIVERSITY

HARDWARE ABSTRACTIONS
- LINUX KERNEL INTERFACE

THIS LECTURE INVESTIGATES...

How does the Operating System provide separation between user-space applications and the OS core itself?

What kind of abstraction is used to provide this separation?

How do applications and OS exchange data?

How do you use it?

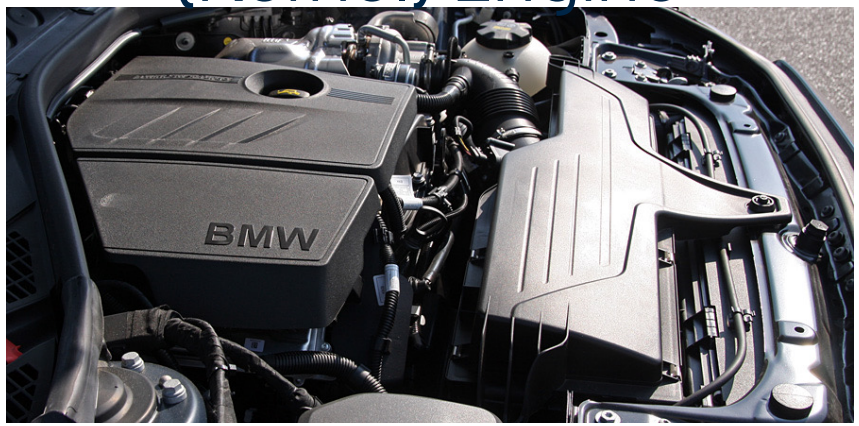
SYSTEM INTERFACE

- Limited interface – I doesn't allow driver to manipulate motor parameters etc.
- Stable interface – across models and vendors
- User commands will (mostly!) be validated

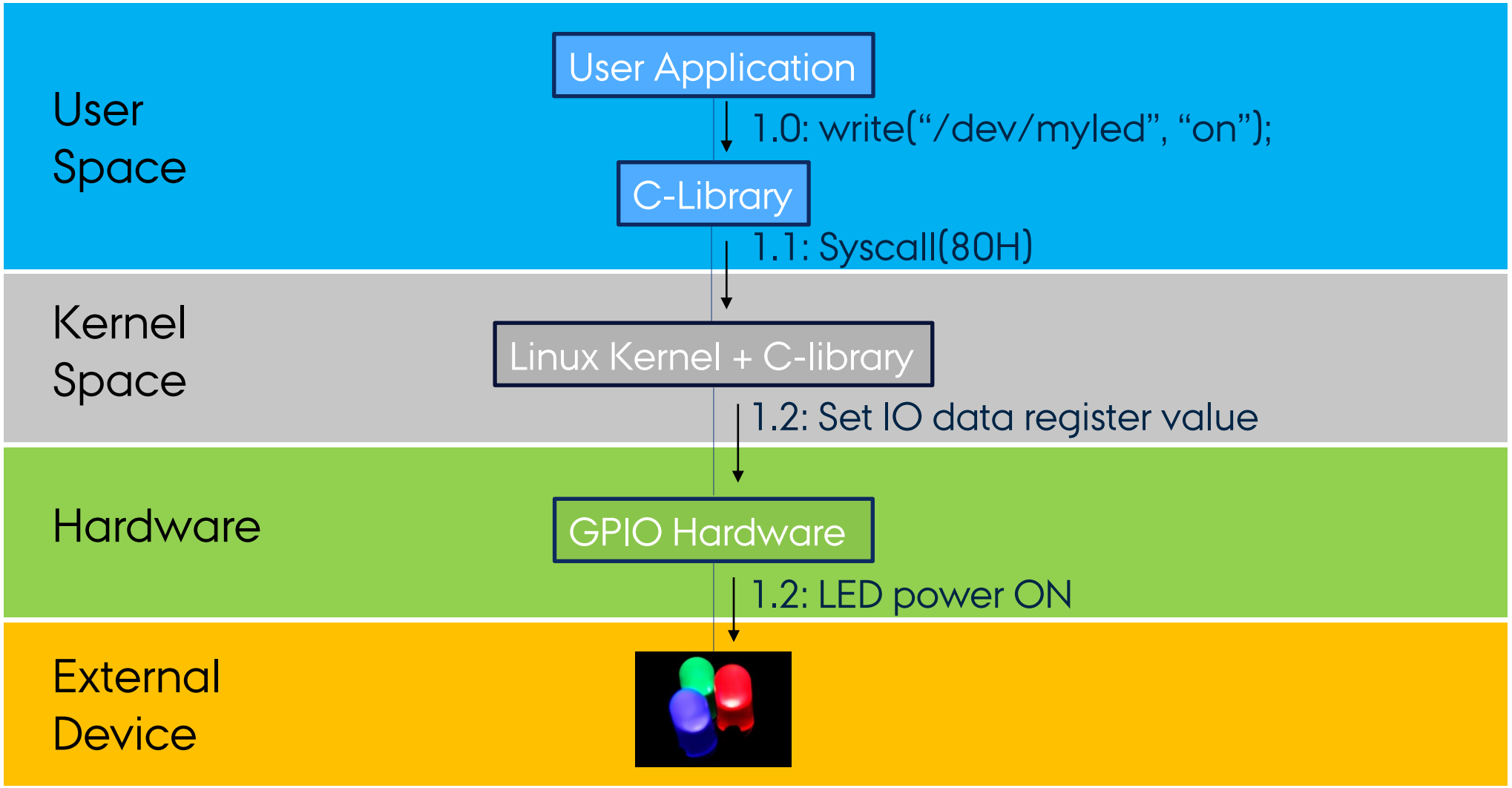
- Priviliged authorization required for access
- Engine parts can be updated with no change in user interface

User Interface

(Kernel) Engine



USER – KERNEL COMMUNICATION EXAMPLE



LINUX KERNEL INTERFACE

- The standard C library is the glue between user (applications) and operating system
  - C-library wraps special calls to the operating system, *system calls*, in functions like open(), read(), exec() and more

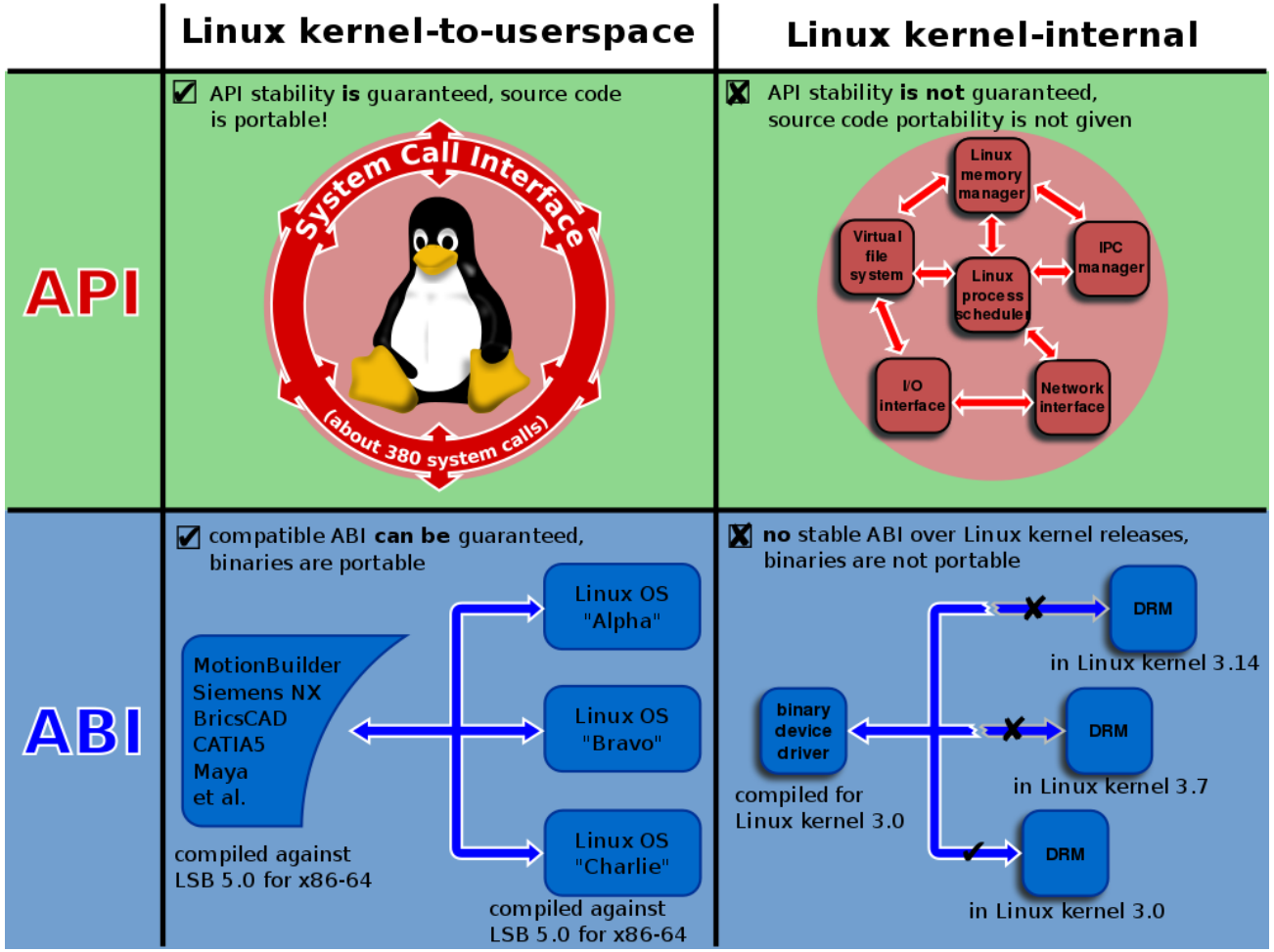
Various layers within Linux, also showing separation between the userland and kernel space

User mode	**User applications**	For example, bash, LibreOffice, Apache OpenOffice, Blender, 0 A.D., Mozilla Firefox, etc.		
	Low-level system components:	**System daemons**: *systemd, runit, logind, networkd, soundd, ...*	**Windowing system**: *X11, Wayland, Mir, SurfaceFlinger (Android)*	**Other libraries:** *GTK+, Qt, EFL, SDL, SFML, FLTK,*
	C standard library	`open()`, `exec()`, `sbrk()`, `socket()`, `fopen()`, `calloc()`, ... (up to 2000 subroutines) *glibc* aims to be POSIX/SUS-compatible, *uClibc* targets embedded systems, *bionic* written for Android, etc.		
Kernel mode	**Linux kernel**	`stat`, `splice`, `dup`, `read`, `open`, `ioctl`, `write`, `mmap`, `close`, `exit`, etc. (about 380 system calls) The Linux kernel System Call Interface (SCI, aims to be POSIX/SUS-compatible)		
		Process scheduling subsystem	IPC subsystem	Memory management subsystem
		Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: *SELinux*, *TOMOYO*, *AppArmor*, *Smack*		
Hardware (CPU, main memory, data storage devices, etc.)				

Source: Wikipedia(Linux)

ROBUST APPLICATION INTERFACE

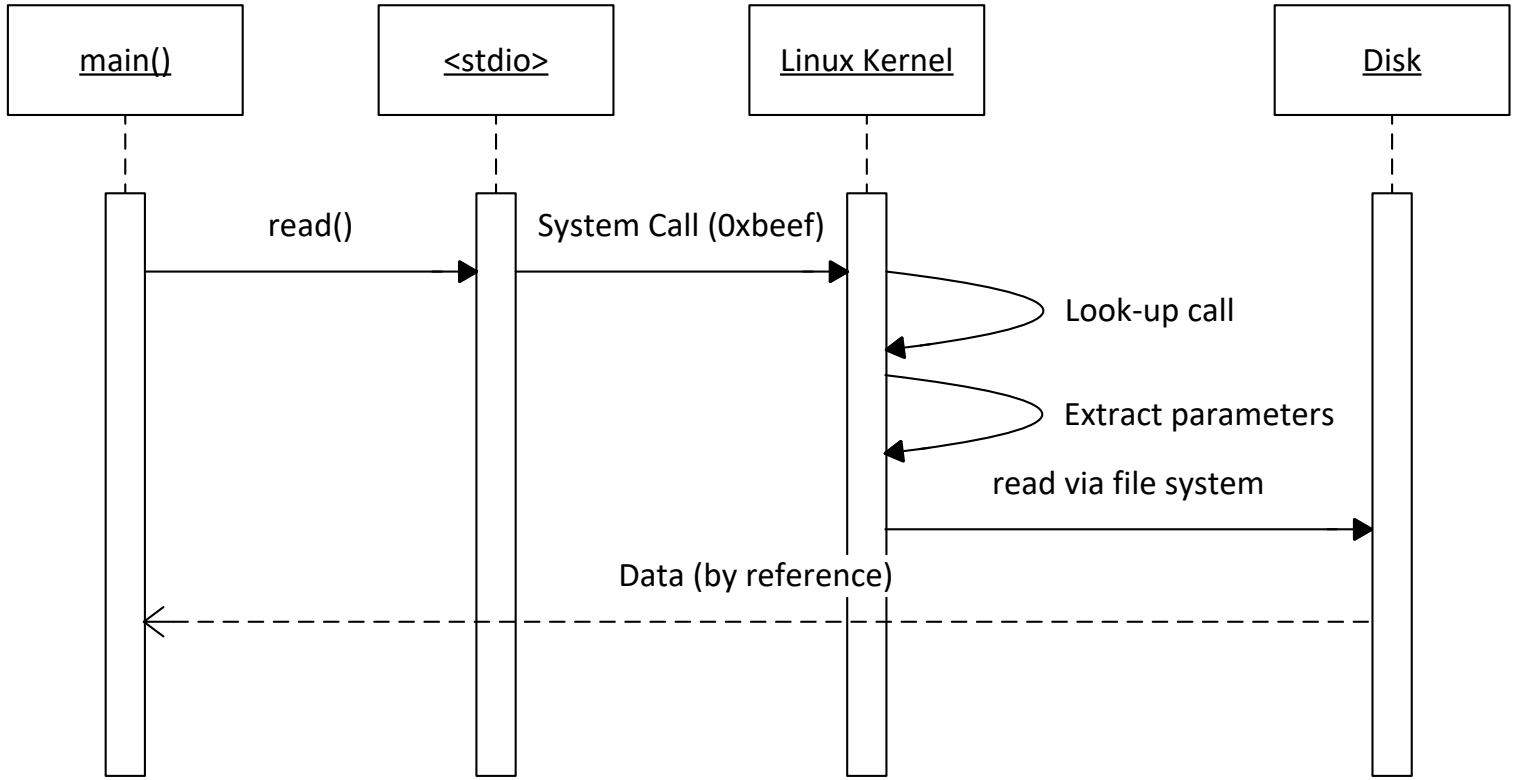
- To users, Linux has stable:
  - Application Programming Interface (API) - Function names and paramters stays the same
  - Application Binary Interface (ABI) – Libraries are binary compatible on similar CPUs
- To Kernel Developers, things are not guaranteed to be stable over different kernel releases and platforms. You will have to test with new kernel releases and platforms



Source: Wikipedia(Kernel_interface)

SYSTEM CALL EXECUTION

- User application, uses normal <stdio> functions to access disk
  - Underlying work is handled by the OS and is transparent to the user



TYPICAL SYSTEM CALLS

Calls to these standard C library functions generates system calls:

- **File Related**:
  - open(), close(), read(), write() // File operations
  - “ioctl()” // Special file operations
  - chmod(), chown() // Access control

  We will focus on these!
- **Process Related** (Process = Program in Execution):
  - exec(), fork(), exit() // Start, spawn and exit process
  - signal(), kill() // Process control
- **Inter process Related** (Communication between programs):
  - pipe() // Pipe for unidirectional data
  - shmget(), shmat() // Shared Memory

- In Unix systems, **files** are used for data exchange between user and OS!!

FILE OPERATIONS

- Four main functions are used:

```
fd = open(pathname, flags, mode);
status = close(fd);
bytes_read = read(fd, buffer, count);
bytes_written = write(fd, buffer, count);
```

- And sometimes a fifth (mainly to support old stuff):

```
status = ioctl(fd, call_id, …);   // For controlling devices
```

- "fd" is a file descriptor, a unique id to the file currently in use.

- Three standard file descriptors exist:
  - 0: stdin // Standard Input, typically input from the terminal
  - 1: stdout // Standard Out, typically output to the terminal (/dev/tty)
  - 2: stderr // Standard Error, typically output to the terminal

OPEN – USER SPACE

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#define BUF_SIZE 1024
int main(int argc, char *argv[])
{
  int fd;

  fd = open("myfile", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  if (fd == -1) {
    printf("Failed to open with err: %s", strerror(errno));
    return -1;
  }

  /* Do Something with the file */
}
```

filename

read+write

create if not available

rd+wr permissions for owner

returned file desctiptor

returns -1 on any error

actual error must be extracted from errno, a global variable

READ – USER SPACE

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#define BUF_SIZE 1024
int main(int argc, char *argv[]) {
  int fd, num_read;
  char buf[128];
  fd = open("myfile", O_RDONLY);

  num_read = read(fd, buf, 16)

  if (num_read == -1){
    printf("Failed to read with err: %s", strerror(errno));
    return num_read; } // Return err
  else if(num_read < 16)
    printf("Read less than requested, only %i bytes", num_read);
  buf[num_read] = 0; // Add zero-termination for string
  printf("Read: %s", buf);
  /* Do Something with buf */ }
```

file descriptor

buffer for result

requested read size

bytes actually read

WRITE – USER SPACE

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#define BUF_SIZE 1024
int main(int argc, char *argv[]) {
 int fd, num_wr;
 char buf[128] = “hello mom”;
 fd = open(”/dev/mydevice", O_WRONLY);

 num_wr = write(fd, buf, 10);

 if (num_wr == -1){
   printf(“Failed to write with err: %s”, strerror(errno));
   return num_wr; } // Return err
 else if(num_wr < strlen(buf))
   printf(“Did not write the whole string, only %i bytes”, num_wr);

 /* Do Something else */
}
```

file descriptor

buffer to be written

requested write size

bytes actually written

CLOSE – USER SPACE

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#define BUF_SIZE 1024
int main(int argc, char *argv[])
{
 int fd, num_wr,
 int status = 0;
 char buf[128] = “hello mom”;
 fd = open("myfile", O_RDONLY);
 num_wr = write(fd, buf, strlen(buf));

 status = close(fd);

 if (status == -1)
   printf(“Failed to close with err: %s”, strerror(errno);

 return status;
}
```

(IOCTL – USER SPACE)

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <termios.h>
#include <string.h>
int main(int argc, char *argv[])
{
 int fd, num_wr,
 int status = 0;
 char buf[128] = “hello mom”;
 fd = open(“/dev/ttyUSB0", O_RDWR);

 status = ioctl(fd, TCSETS, B115200 | CS8);
 if (status == -1)
   printf(“Failed to configure uart with err: %s”, strerror(errno);
 status = close(fd);

 return status;
}
```

Defined in termios.h

bytes actually written

file descriptor

Command ID (config uart)

Cmd Value (115200 8N1)

FILES AND DEVICES

- A ”device” in unix terminology is a piece of hardware:
  - Harddrive, GPIO, Touch screen, Accelerometer, Serial port…
- As everything is files, devices appear as files and you can access them as files!

```
cat /dev/key                    // Readout value of key
echo 1 > /dev/gpio0             // Set gpio0 to ‘1’
```

- You will find devices in:
  - /dev/ - Standard device folder with devices you can read/write:
    - /dev/ttyUSB0 – A USB uart that you can read/write
    - /dev/SDA – Harddrive that you can mount to a folder and use
  - /sys/ - Folder with all devices, also include special files to config devices:
    - /sys/class/leds/input1::capslock/brightness – Caps Lock LED brightness
    - /sys/class/leds/input1::capslock/max_brightness – Configure MAX brightness

GPIO ACCESS FROM USER SPACE

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
/* Usage:                */
/* <set_gpio> <value> */
/*  (argv[0])(argv[1])*/
int main(int argc,
         char *argv[])
{
  int fd, ret;
  if (argc != 2) {
      printf("Error! Usage: set_gpio <value>\n");
      return -1; }

  fd = open("/sys/class/gpio/gpio164/value", O_RDWR);
  /* Write value of cmd line argument to gpio[164] */
  ret = write(fd, argv[1], strlen(argv[1]));
  close(fd);
  return ret; }
```

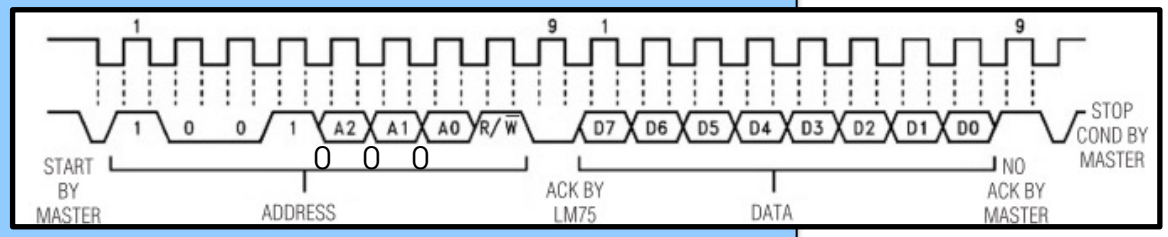
```
/* This example is based on the GPIO Sysfs Interface
(Already installed) More here: https://www.kernel.org/doc/
html/latest/driver-api/gpio/legacy.html#paths-in-sysfs */

/* GPIO must be exported prior to use on the RPI:
   (Export GPIO164 and set direction to output) */
echo 164 > /sys/class/gpio/export
echo "out" > /sys/class/gpio164/direction

# Set GPIO with program compiled to RPI
set_gpio 1
```

I2C ACCESS FROM USER SPACE

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
/* read_lm75 */
int main(int argc,
         char *argv[])
{
  int fd;
  int ret = 0;
  char rddata[4], str[16];
  fd = open("/dev/i2c-2", O_RDWR); // Open i2c-dev on I2C bus 2
       sys command                 i2c device addr
  ret = ioctl(fd, 0x0703, 0x46);   // Set I2C Device Address
  if (ret < 0) {
    printf("Could not find i2c device\n");
    return ret; }
  ret = read(fd, rddata, 1); // Read 1 byte
  printf("First Byte: %i\n", rddata[0]);
  close(fd);
  return ret; }
```

This example uses the generic i2c device driver, **i2c-dev**, installed on the Rpi. See: https://www.kernel.org/doc/html/latest/i2c/dev-interface.html

LM75 I2C Protocol



I2C Device addresses + busses can be found in the ASE fHat schematics on the raspberrypi-zero wiki

AARHUS
UNIVERSITY

LINUX KERNEL INTRODUCTION

THIS LECTURE INVESTIGATES...

What's inside an OS?

How is the source code structured for an OS like Linux?

- What kind of code resides where? –and where is main()?!?

How do **you** extend the OS functionality?

- How do you access functions and subsystems available in the OS (kernel), when working inside the kernel?
- How do you let the user use new functionality?

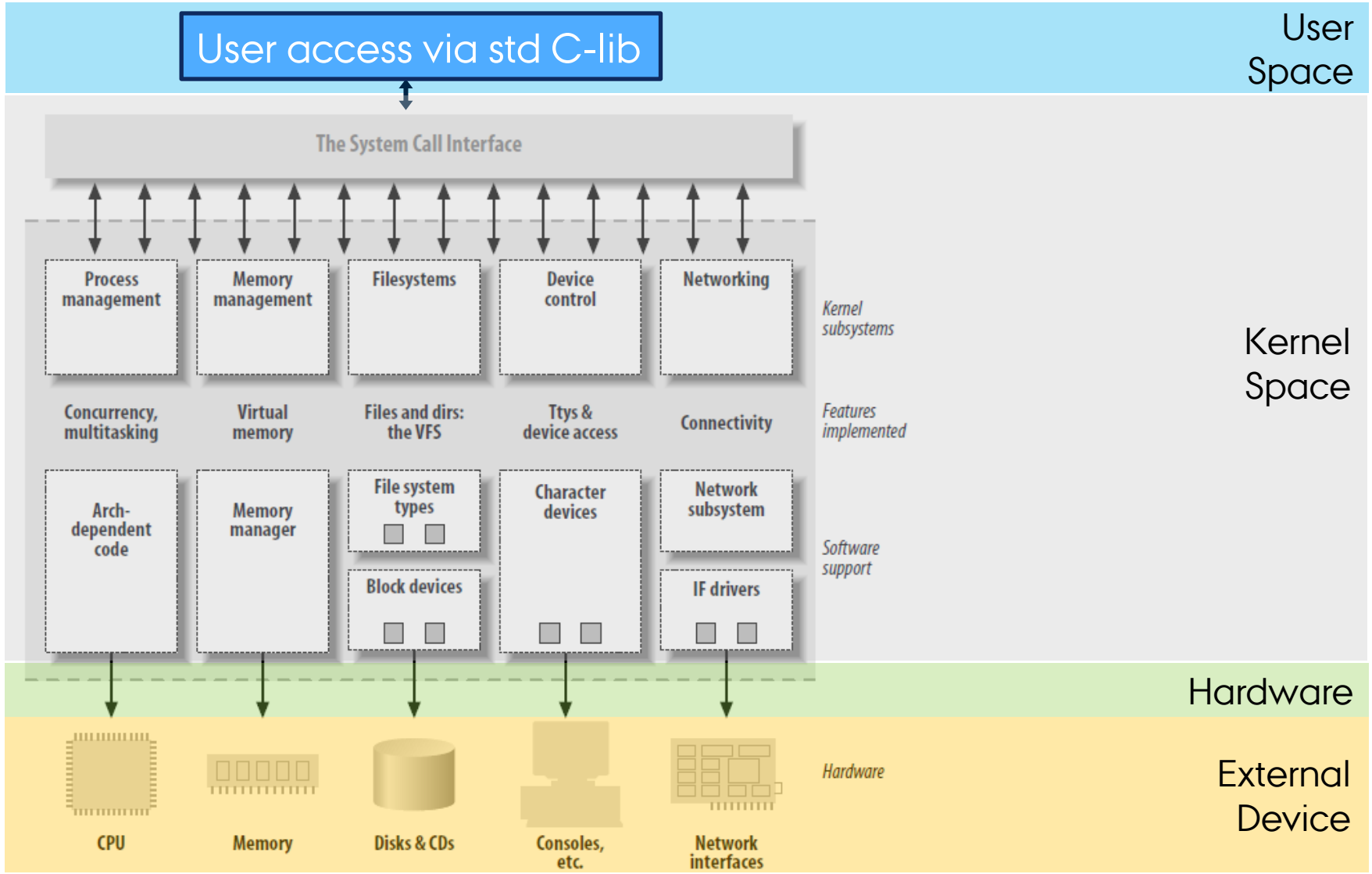
What is a Kernel Module?

- How do you create- and build one?
- How do you add one to a running kernel?

LINUX'S SYSTEM DESIGN

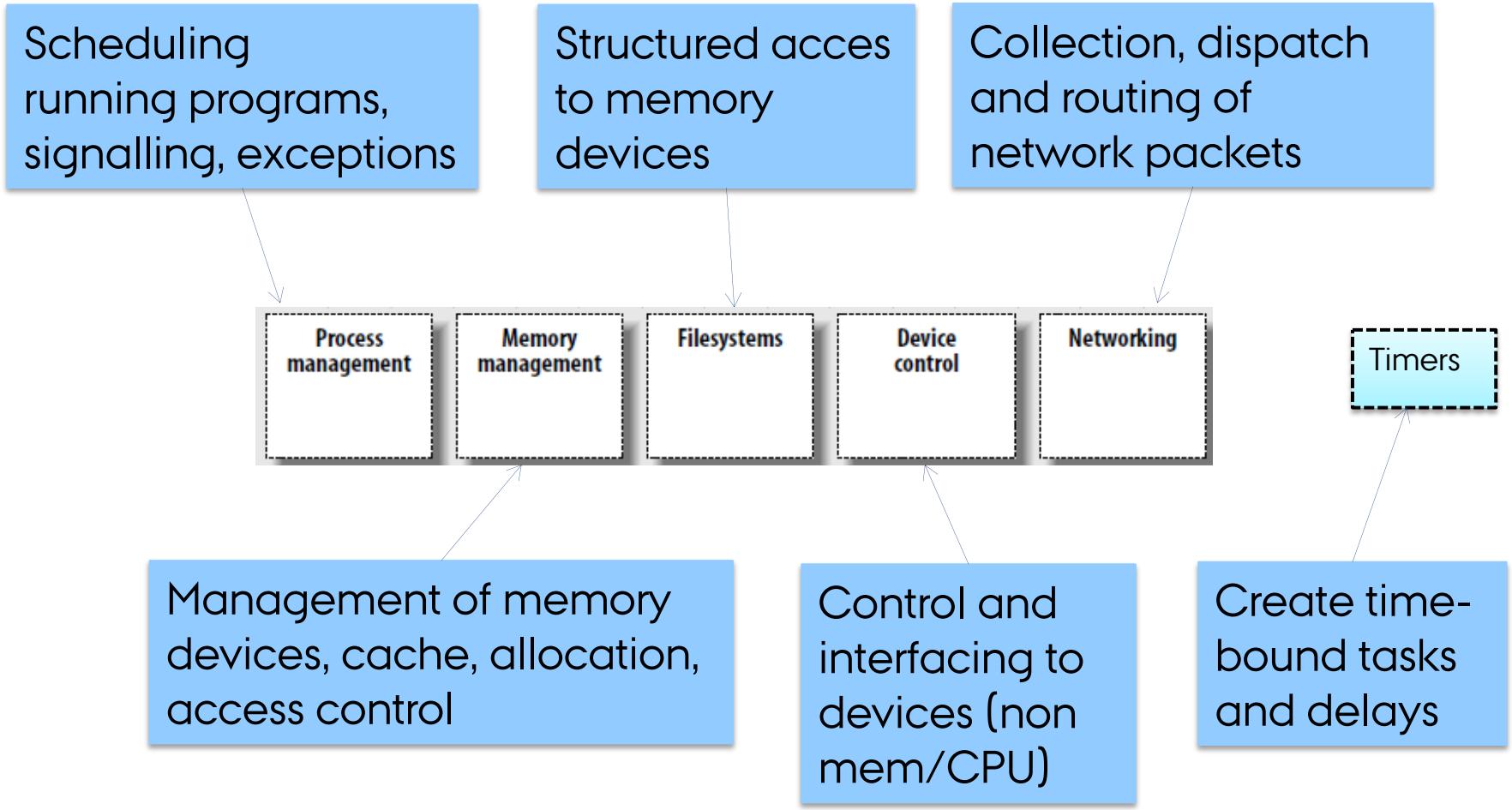
Core OS Components

Blocks adapted to the particular platform



[Corbet et al, Linux Device Drivers 3rd ed]

CORE OS COMPONENTS



These blocks are typical to most operating systems!

LINUX'S ARCHITECTURE

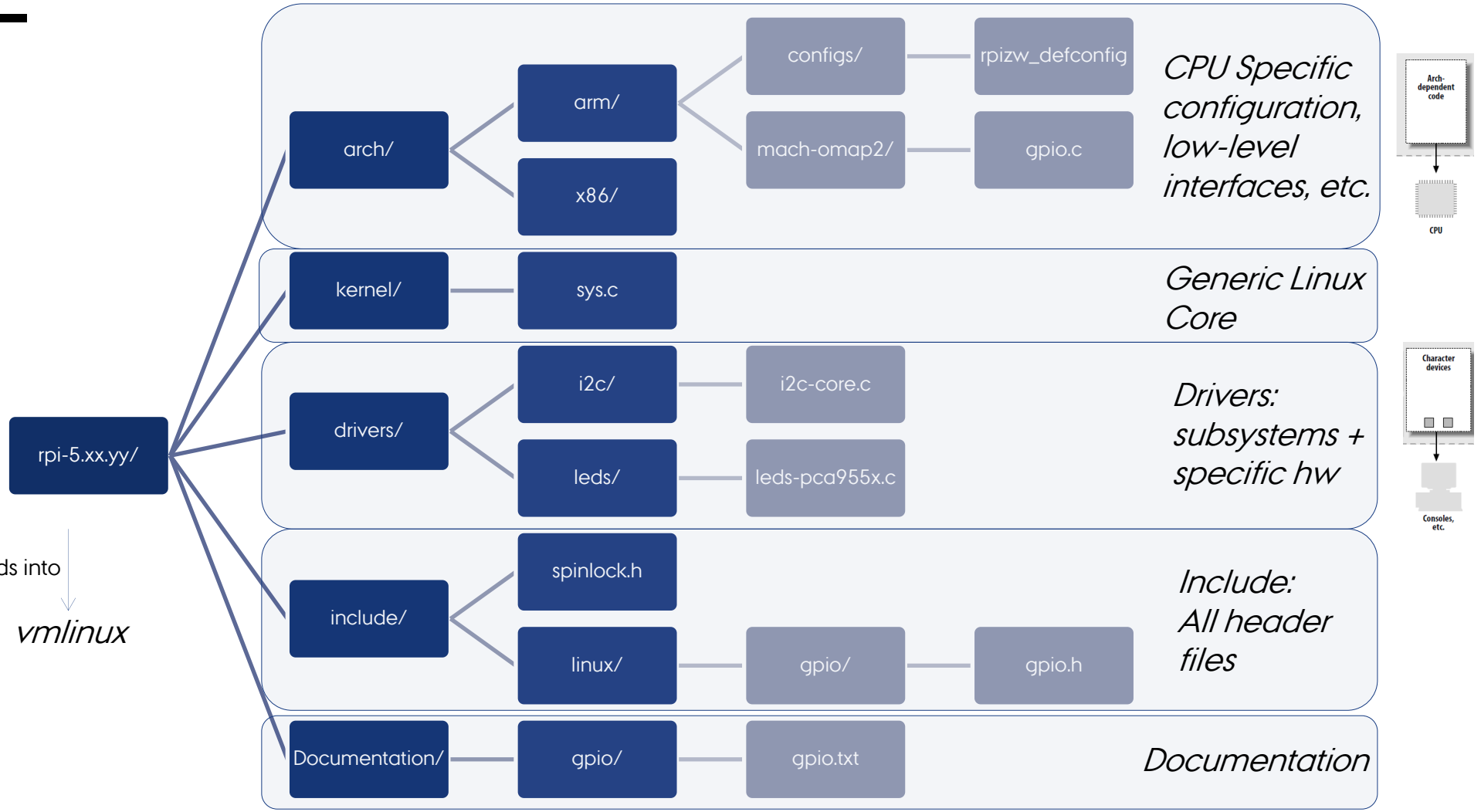
- Linux uses a monolithic kernel
  - All core components is contained within the kernel binary!

- Linux kernel is one binary with no external references!
  - All library functions are contained within!
  - Name: `vmlinux`

- Linux kernel provides build-in libraries and sub-systems to be used in your kernel code
  - std-c like functions, Interrupt-, USB-management, and much more

- The source code is arranged in a tree-like structure, called the *source tree*!

LINUX SOURCE TREE (CODE!)



[/home/stud/sources/rpi-5.xx.yy/]

HOW TO BUILD LINUX!

First, the build system must be configured for the platform:

```
# make rpizw_defconfig
```

Actually, we must tell in which *arch* subfolder *rpizw_defconfig* is and which compiler-toolchain to use:

```
# make ARCH=arm CROSS_COMPILE=arm-poky-linux-gnueabi- rpizw_defconfig
```

When configuration has been prepared, we can build a compressed, self-extracting, Linux image:

```
# make ARCH=arm CROSS_COMPILE=arm-poky-linux-gnueabi- zImage
```

It's basically two steps to build Linux! The output file, `rpi-5.xx.yy/arch/arm/boot/zImage` can be copied to the boot partition of your Raspberry Pi and booted.

```
...3/arch/arm/configs/rpizw_defconfig
#
# Other Architectures
#
CONFIG_ARCH_BCM2835=y
CONFIG_BCM2835_FAST_MEMCPY=y
#
# Processor Type
#
CONFIG_CPU_V6K=y
CONFIG_CPU_THUMB_CAPABLE=y
CONFIG_CPU_32v6=y
CONFIG_CPU_32v6K=y
CONFIG_CPU_ABRT_EV6=y
CONFIG_CPU_PABRT_V6=y
CONFIG_CPU_CACHE_V6=y
CONFIG_CPU_CACHE_VIPT=y
CONFIG_CPU_COPY_V6=y
CONFIG_CPU_TLB_V6=y
CONFIG_CPU_HAS_ASID=y
```

Configuration file:
rpi-5.xx.yy/arch/arm/configs/
Rpizw_defconfig

CREATE YOUR OWN KERNEL CODE

- You can add your own code to Linux to provide support for:
  - New hardware devices
  - New fancy algorithms to be used within the kernel
  - New file systems
  - … mostly up to you!
- Your code will become library code (sort of)
  - No main()!!! – It must be called by someone to execute
  - As it is part of the kernel, user applications can only access it through system calls!!!
  - Functions must be exported to make them available for other kernel modules
  - File system methods are used for data exchange with user applications
- Kernel code have ”full” access to the OS!!!
  - *But it is not the place for applications to run!*

BUILD YOUR OWN KERNEL CODE

Kernel code can be built in two ways:

- Added to the original kernel source tree and configuration (in-tree-compilation)
  - Can be compiled statically with Linux or as a loadable kernel module
  - Must be GPL open-source!
  - Your code is ready for a pull-request to main-line Linux (are you?)

- Can be compiled in your local folder as an out-of-tree loadable kernel module
  - Fast and easy compilation using special Make parameters
  - Easy to test and debug
  - Is not considered 100% GPL and taints the kernel
  - *Is how we'll add our own code in this course*

KERNEL MODULES

- Are like dynamically-linked libraies, loaded when needed
  - Can also be unloaded and reloaded, great for debugging ☺!
- Adds very little overhead, as it uses kernels main memory and is linked directly to the kernels binary code
  - Compared to user programs and micro kernels, that must use special message passing schemes
- Has a well-defined interface to the kernel
  - Constructor/Destructor-like functions to be called during module loading/unloading
- *Method is often used by device drivers*

DEVICE DRIVERS

- Must provide the glue between physical hardware and operating system
  - OS defines the interface between application and driver (and hardware)
  - Applications cannot access HW directly (-and crash the system!!)

- Must inccur only little overhead
  - This is well-supported by kernel modules

- Device Drivers *can be* build as Kernel Modules to add support for new hardware at run-time
  - To provide us with the advantages of kernel modules
  - To avoid bloating the kernel with excessive, possibly unused code
  - The drivers can also be built with the kernel, as any piece of kernel code

OPEN SOURCE COMMUNITY

You are part of a development community!

There are several mailing lists and forums

- lkml.org, linuxquestions.org and several more....

You may copy code ”freely” and adopt

- Refer to source for academic reasons!!!

Use appropriate licensing (Copyleft)

- GPL / LGPL / BSD and more (http://opensource.org/licenses/)

Publish your updates for the benefit of the community

AARHUS
UNIVERSITY

LINUX KERNEL MODULES

MAKING KERNEL MODULES

It's plain-C, but with limitations!

Can only use the functionality present in the kernel:

- No access to the standard C / C++ libraries
- Header files is found in /rpi-5.xx.yy/include/ (or elixir.bootlin.com)
- No floating point operations

Errors in kernel modules can be fatal!

- There is no protection inside the kernel

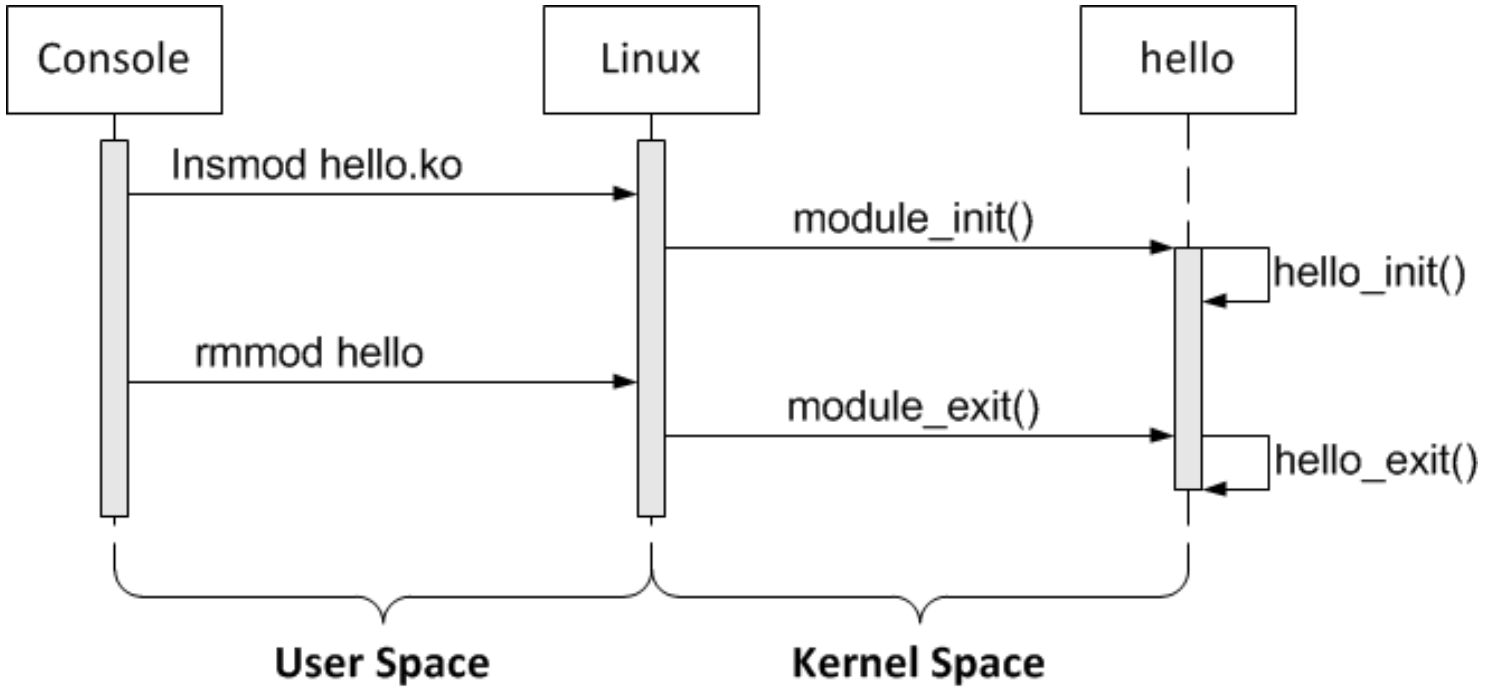
Limited Memory

- The kernel has only a small stack
- Do not allocate a lot of memory
- Use dynamic allocation for larger structures

HELLO_WORLD KERNEL MODULE

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

static int __init hello_init(void)
{
  pr_info("Hello, world\n");
  return 0;
}
static void __exit hello_exit(void)
{
  pr_info("Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_AUTHOR(”Peter HM <phm@ece.au.dk>”);
MODULE_LICENSE("GPL");
```

KERNEL MODULE LIFE-CYCLE



KERNEL MODULE SYMBOLS/LIBS

- **Libraries**
  - <linux/**module.h**> - Contains symbols and functions for dynamically loading modules
- **MODULE_LICENSE**
  - Sets the license of your kernel module (”GPL”, "Proprietary” a.o)
  - If not specified as free (GPL a.o.), the kernel will become “**Tainted**” and the module will fail to link with many kernel symbols and functions
- **MODULE_AUTHOR**
  - States who wrote the module
- **MODULE_DESCRIPTION**
  - Plain text that describes it’s function

COMPILING KERNEL MODULES

- The GNU Make function has an extended syntax for building modules.
  - It runs a Makefile in the kernel source tree!

- To build kernel modules the following must be prepared:
  - Access to the linux kernel source tree for the target platform
  - *~/sources/rpi-5.XX.YY/*
  - Access to the corresponding compiler version
  - *arm-poky-linux-gnueabi-gcc --version* ➔ *9.3.0 etc*

```
Make -C /home/stud/sources/rpi-5.XX.YY M=. modules
```

Kernel source tree

Where to find your code
(. = current folder)

Inserting out-of-tree built modules, result in a tainted kernel. It's political... -and not a problem to our current use.

- You can also create a local Makefile
  - That will call the Makefile in the source tree with the correct parameters!

MAKEFILE FOR KERNEL MODULES

Assumes source file = hello.c

```
# FILE: Makefile (WITH CAPITAL M!!)
obj-m := hello.o
ccflags-y := -std=gnu99 -Wno-declaration-after-statement -Werror

KERNELDIR ?= ~/sources/rpi-5.xx.yy

all default: modules
install: modules_install

modules modules_install help clean:
    $(MAKE) ARCH=arm CROSS_COMPILE=arm-poky-linux-gnueabi- -C
$(KERNELDIR) M=$(shell pwd) $@
```

gcc parameters

Target

Command

<TAB>

Line wraps!!

$@ substitutes target name, ex. ”modules” or ”clean”

MODULE DEPENDENCIES

- **Version dependency**
  - Kernel source tree is used when compiling the module and the module is therefore dependent on kernel's version!
  - A new target kernel version often requires a new compilation!

- **Platform dependency**
  - Build configuration is platform dependent: Changing CPU or platform forces kernel recompilation
  - For general distributable device drivers, many binaries must be provided –or the source code (could be under GPL).
  - Linux works on a variety of processors (x64, ARM, MIPS etc.) and platforms (Beaglebone, Raspberry Pi, PC etc.), which is why you can't just use any supplied binary file.

MODULE LOADING AND UNLOADING

- **insmod**
  - Inserts the module into the kernel and links the module to it. Notifies the kernel what new functionality is now provided.
- **modprobe**
  - Same as insmod, but resolves any modul dependencies and insert the required additional modules. Looks for modules in /lib/modules/
- **rmmod**
  - Removes the module from the kernel. And tells the kernel that the supplied functionality is no longer available
- **lsmod**
  - Lists the modules currently loaded into the kernel

THE INIT FUNCTION

```
static int __init hello_init(void)
{
  pr_info("Hello, world\n");
  return 0;
}
module_init(hello_init);
```

- **Kernel Module's Constructor function**
  - Is used to prepare ressources to be used by the kernel module.
  - I called once, when the module i loaded.
  - The "__init" token hints kernel that this code is used only during initialization, and can be thrown away once the module is loaded, to free up memory.
- **module_init** is a mandatory macro defining the init function (tells the system which function is the Constructor)

THE EXIT FUNCTION

```
static void __exit hello_exit(void)
{
  pr_info("Goodbye, cruel world\n");
}
module_exit(hello_exit);
```

- **Kernel Module's Destructor function**
  - Is used to cleanup ressources used by the kernel module.
  - Is called once, when the module i unloaded.
  - The "__exit" token tells the kernel that this function is only used during module cleanup.
- **module_exit** is a mandatory macro defining the exit function (tells the system which function is the Destructor)

KERNEL LOGKERNEL LOG

- Linux maintains a log of kernel messages
- Messages are tagged with a log level
- Messages with loglevel ≤ System's current loglevel is written to the kernel log
- Current log level can be seen with:

```
$ cat /proc/sys/kernel/printk
4        4        1        7
```

Current Default Minimum Boot value

- Change log level: `$ echo 7 > /proc/sys/kernel/printk`
- To create messages in your code, use `printk(KERN_ALERT "alert\n")` or `pr_alert("alert\n")` style messages
- View kernel log with: `$ dmesg`
- or wait for new input with: `$ dmesg -w` (ctrl-c to exit)

```
pr_debug("Level 7: Debugging Messages\n");
pr_warning("Level 4: Non-Imminent Error\n");
pr_err("Level 3: Non-Critical Error\n");
pr_alert("Level 1: Immidiate user attention required\n");
```


```
[    1.680000] can: raw protocol (rev 20120528)
[    1.690000] can: broadcast manager protocol (rev 20120528 t)
[    1.690000] can: netlink gateway (rev 20130117) max_hops=1
[    1.700000] Key type dns_resolver registered
[    1.710000] cpu: dev_pm_opp_of_cpumask_add_table: couldn't find opp table for cpu:0, -19
[    1.710000] cpu cpu0: dev_pm_opp_get_opp_count: device OPP not found (-19)
[    1.720000] ThumbEE CPU extension supported.
[    1.720000] Registering SWP/SWPB emulation handler
[    1.740000] sunxi-mmc 1c11000.mmc: smc 1 err, cmd 8, RTO !!
[    1.750000] of_cfs_init
[    1.750000] of_cfs_init: OK
[    1.760000] vcc3v0: disabling
[    1.770000] vcc5v0: disabling
[    1.770000] Freeing unused kernel memory: 320K
[    1.780000] sunxi-mmc 1c11000.mmc: smc 1 err, cmd 55, RTO !!
[    1.790000] sunxi-mmc 1c11000.mmc: smc 1 err, cmd 55, RTO !!
[    1.790000] sunxi-mmc 1c11000.mmc: smc 1 err, cmd 55, RTO !!
[    1.800000] sunxi-mmc 1c11000.mmc: smc 1 err, cmd 55, RTO !!
[    1.870000] devpts: called with bogus options
[    2.080000] random: dd: uninitialized urandom read (512 bytes read, 5 bits of entropy available)
[    2.520000] using random self ethernet address
[    2.530000] using random host ethernet address
[    2.590000] using random self ethernet address
[    2.600000] using random host ethernet address
[    3.120000] random: dropbear: uninitialized urandom read (32 bytes read, 7 bits of entropy available)
[    3.170000] random: dnsmasq: uninitialized urandom read (128 bytes read, 7 bits of entropy available)
[    3.180000] random: dnsmasq: uninitialized urandom read (48 bytes read, 7 bits of entropy available)
```

ERROR HANDLING (1)

- During initialization of a module errors may occur:
  - Memory may not be available
  - Resources are unavailable

- Linux does not keep track of what you may have registered.
  - Failing to initialize a module may lead to stray pointers a.o!!!

- **You** must check for errors, and cleanup resources accordingly

- This is a legitimate place to use ”goto” ☺!

ERROR HANDLING (2)

```c
static int __init hello_init(void)
{
  int err;
  /* Allocate resouces */
  err = allocate_it("A");
  if (err) goto fail_A;
  err = allocate_it("B");
  if (err) goto fail_B;
  err = allocate_it("C");
  if (err) goto fail_C;

  return 0; /* success */

  fail_C: deallocate_it("B");
  fail_B: deallocate_it("A");
  fail_A:
  return err; /* propagate the error */
}
```

Functions typically return 0 on success, but NOT always, check!!!

DEALLOCATION OF RESOURCES

Anything allocated/registered in ***init***, must be deallocated/unregistered in ***exit***.

Remember to deallocate in reverse order!!!

```
static int __init hello_init(void)
{
  int err;
  /* Allocate resouces */
  err = allocate_it("A");
  err = allocate_it("B");
  err = allocate_it("C");
  return 0; /* success */
  …
}

static void __exit hello_exit(void)
{
  /* Deallocate resouces */
  deallocate_it("C");
  deallocate_it("B");
  deallocate_it("A");
}
```

AARHUS
UNIVERSITY

LINUX CHARACTER DRIVERS -INTRODUCTION

THIS LECTURE INVESTIGATES...

Which classes of devices are available in Linux

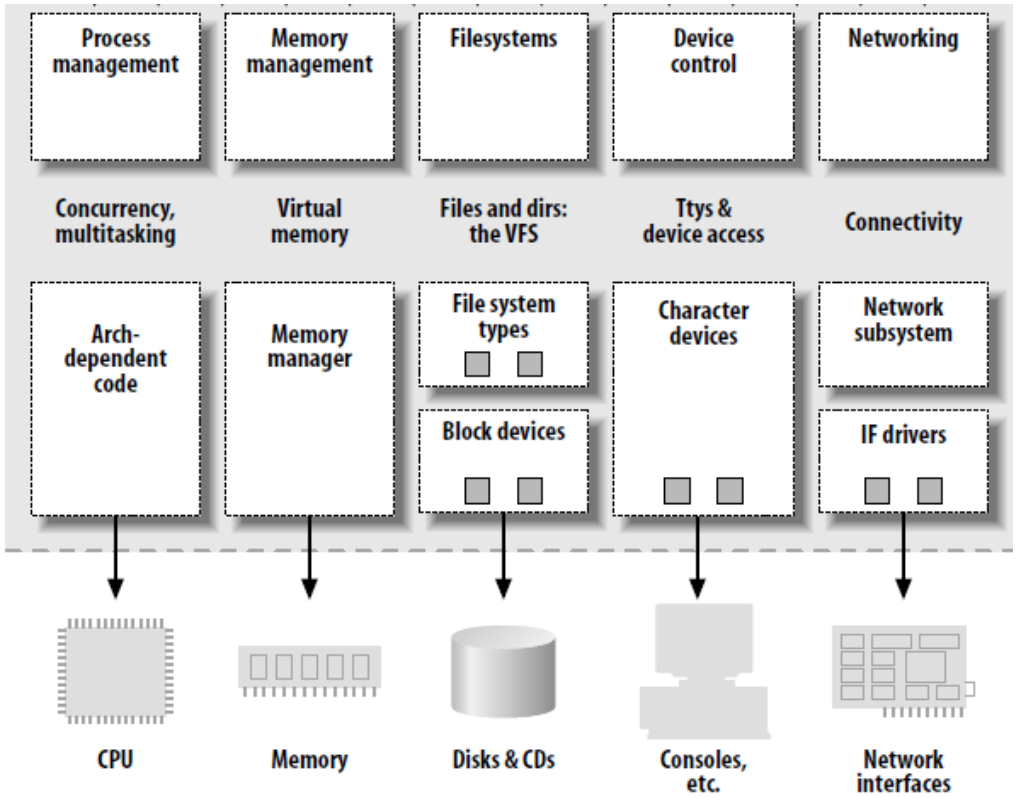
How applications access the correct driver and its functions inside the kernel

What initialization is required in a driver, to make it available to the system

How to write file operation functions to transfer data between hardware devices and user applications

DEVICE CLASSES (1:2)

- **Linux has three devices classes:**
  - Character Devices
  - Block Devices
  - Network Interfaces



- **Character Devices**
  - Device is accessed as a *stream* of data, one character at a time....
  - *Open, close, read, write*
  - Ex: /dev/ttyS0, /dev/input/.. a.o.
  - Device is accessed like an ordinary file, except from moving around the file.
  - To access the next chronological value, another read must be performed.

DEVICE CLASSES (2:2)

- **Block Devices**
  - Block-based interface (Fixed size of data chunks ex 512b)
  - Random-Access to devices
  - Device typically hosts a file system
  - Access through file system (/dev/device)
  - User space interface similar to char devs
  - Kernel space interface completely different than char devs

- **Network Interfaces**
  - Packet based Interface.
  - Hardware or software Interface (loopback)
  - Handles package transmission as dictated by the kernel network sub-system
  - Is NOT mapped to /dev/ttyx

CHARACTER DEVICES

A characters device consists of:

- Kobject: A handle used be the OS to manage drivers
- Owner: A reference to its kernel module (.ko)
- Ops: File operations used for data exchange with user. Open, close, read, write a.o.
- List: A linked-list entry, used by the driver to manage associated devices
- Dev: A device number, used by the file system to find the device (major/minor numbers)
- Count: Number of devices that uses the character driver

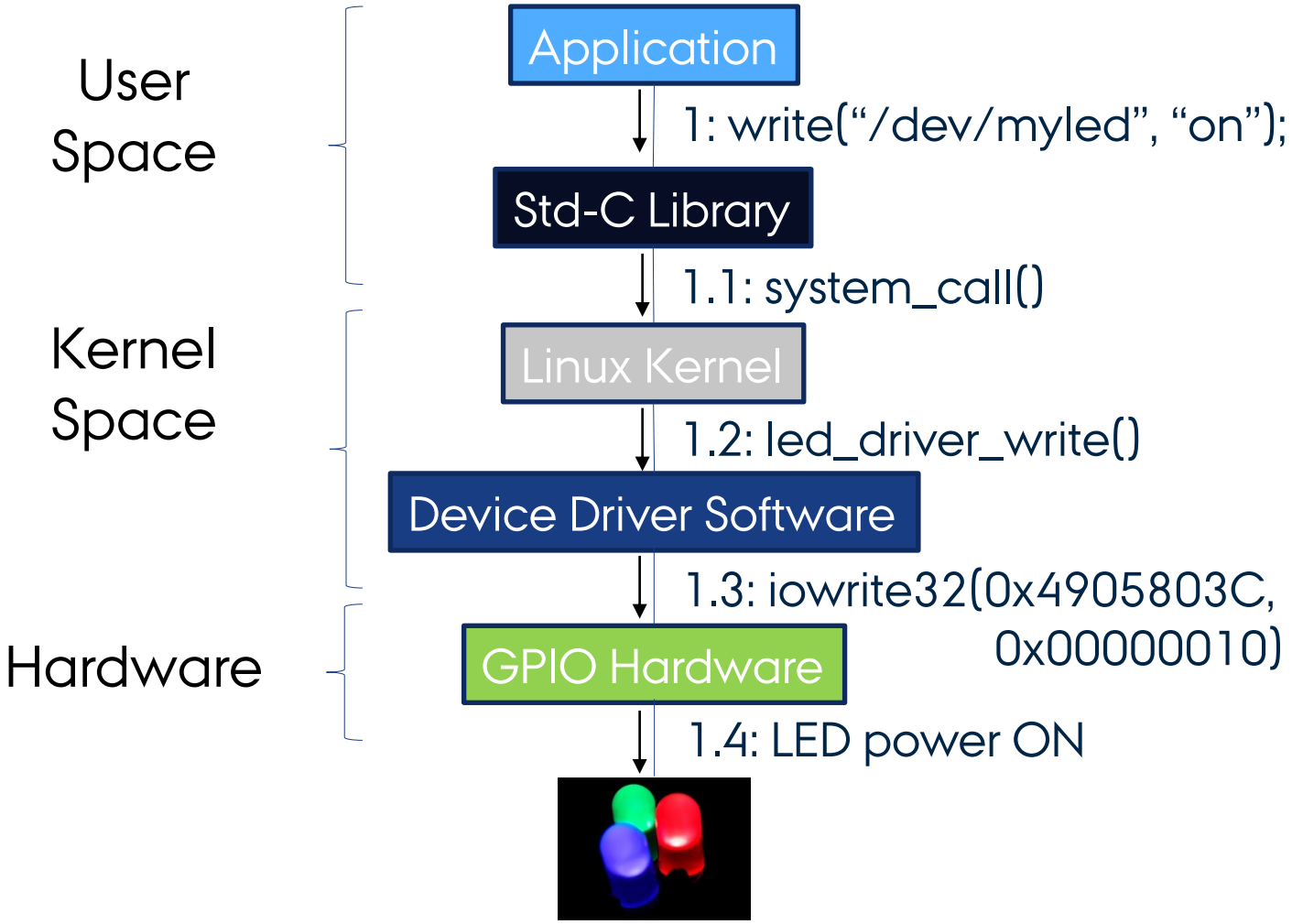
Examples

- Serial ports (RS-232/RS-485), GPIO, A/D Converter, mouse, etc

cdev
kobj: Kobject owner: *module ops: *file_operations list: list_head dev: dev_t count: int
+init() +add() +del()

ASR33 Teletype Origin of the abbreviation tty

LED ACCESS FROM APPLICATION



NODE (FILE) & DRIVER BINDING

- Application knows of a file (ex /dev/led)
- But how does that relate to a specific driver?!!?

- Linux file systems use a structure called **inode** to store metadata about a file:
  - Size, ownership, timestamp, pointers to data blocks… and *root device*

- Root device, **rdev**, tells us on which ex. raw disk that data is stored. In our case, which driver and device must be used!!

- File and driver must be assigned a root **device number**. This number is created from two numbers:
  The major- and minor numbers

Device	dev	inode	Rdev	Type
/dev/sda	6	3	1	Raw Disk
/dev/sda 1	1	7	23	Partition
Hello.txt	23	4556	N/A	File

Example from qnx.com

MAJOR NUMBERS

A unique number to identify the device driver

Says nothing about the functionality

Can be allocated statically in the driver or dynamically during module insertion

Character / Block Devices each have a set of numbers

O**ne major** number per ***driver***

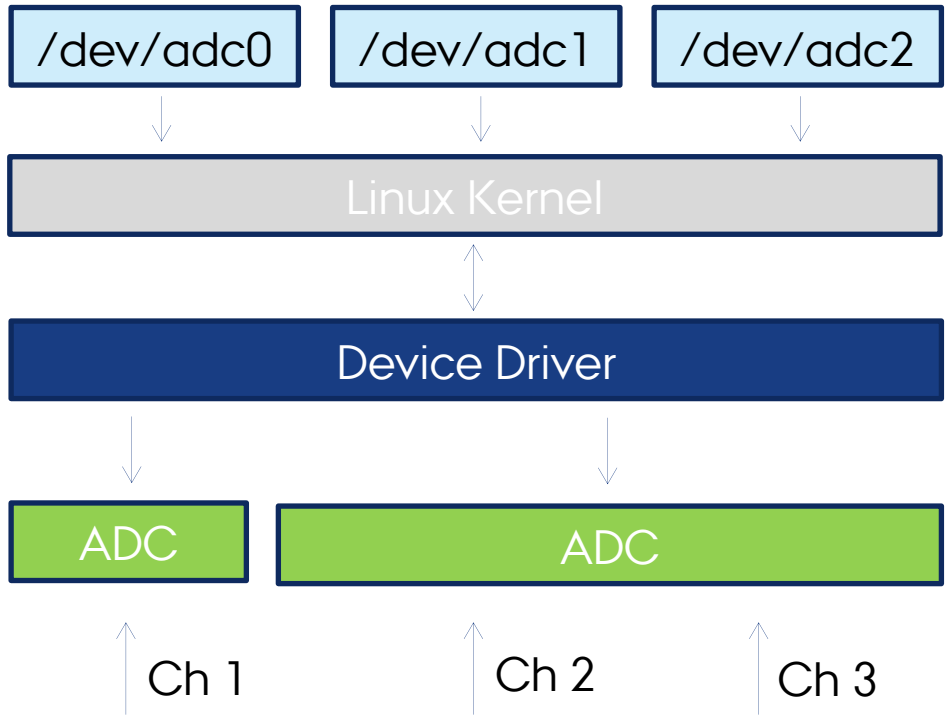
MINOR NUMBERS

Specifies the device that uses the driver

- ***One minor*** *number per* ***device***

Example

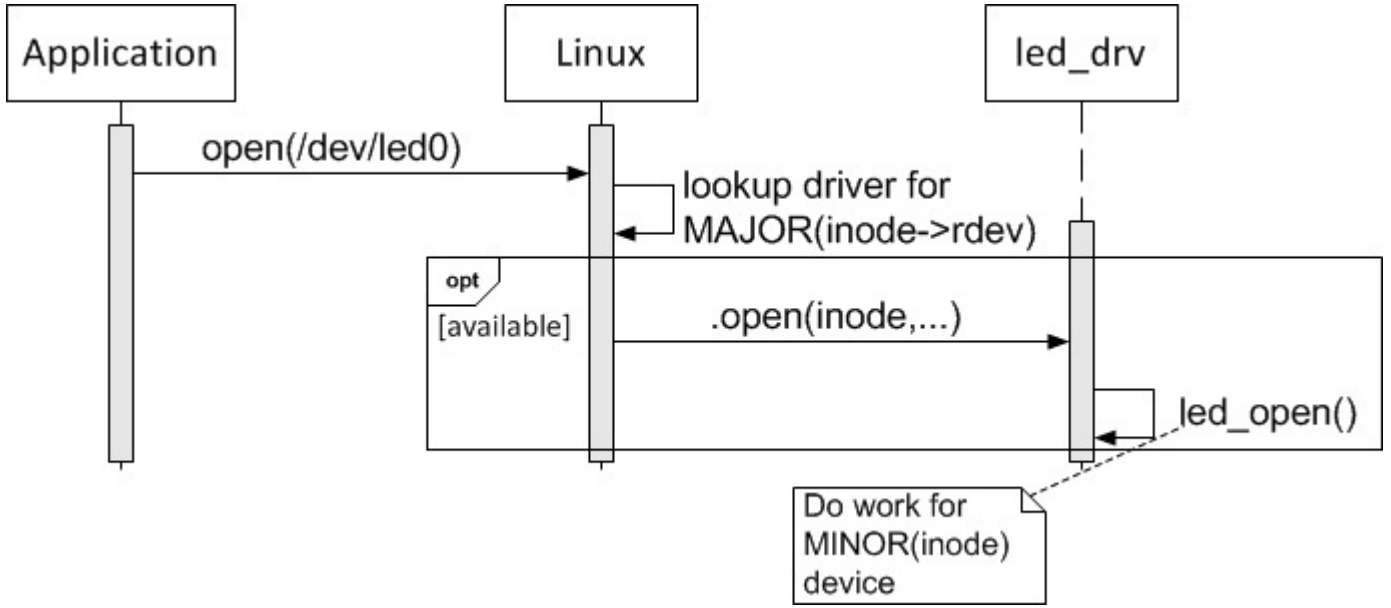
- A/D Converter
  - Major Number: 63
  - Minor Numbers: 0-2
  - One (logical) device per channel



- Kernel knows relation between device file and major/minor

- Driver (you!) knows relation between minor number and physical channels/device

ACCESSING DRIVER WITH MAJOR/MINOR



- The root device number of /dev/led0 is used by Linux to find major number and thereby the corresponding driver
- The minor number can be extracted inside the driver to do specific work for a specific device

AARHUS
UNIVERSITY

LINUX CHARACTER DRIVERS -INITIALIZATION

MODULE INITIALIZATION

The kernel module init function must be updated:

Acquire needed resources

- Memory mapped I/O
- Initialization of hardware
- Acquisition of IRQ

Allocate major/minor number (needed for artificial file)

- Pairing the file in the filesystem with the driver in kernel space
- Can be done statically or dynamically

Initialize and add character driver

- Driver registration - Kernel now aware of driver

```c
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>

static dev_t devno;
static struct class *hello_class;
static struct cdev hello_cdev;
struct file_operations hello_fops;

static int __init hello_init(void)
{
  return 0;
}

static void __exit hello_exit(void)
{

}
module_init(hello_init);
module_exit(hello_exit);
MODULE_AUTHOR("Peter HM <phm@ece.au.dk>");
MODULE_LICENSE("GPL");
```

ALLOCATION OF MAJOR NUMBER

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

const int first_minor = 0;
const int max_devices = 255;
static dev_t devno;

static int __init hello_init(void)
{
  int err=0;

  err = alloc_chrdev_region(&devno, first_minor, max_devices, "hello-driver");
  if(MAJOR(devno) <= 0)
    pr_err("Failed to register chardev\n");

  pr_info("Hello driver got Major %i\n", MAJOR(devno));
  return err;
}
```

Device number is defined as:
devno = major << 20 | minor
(See MKDEV in kdev_t.h)

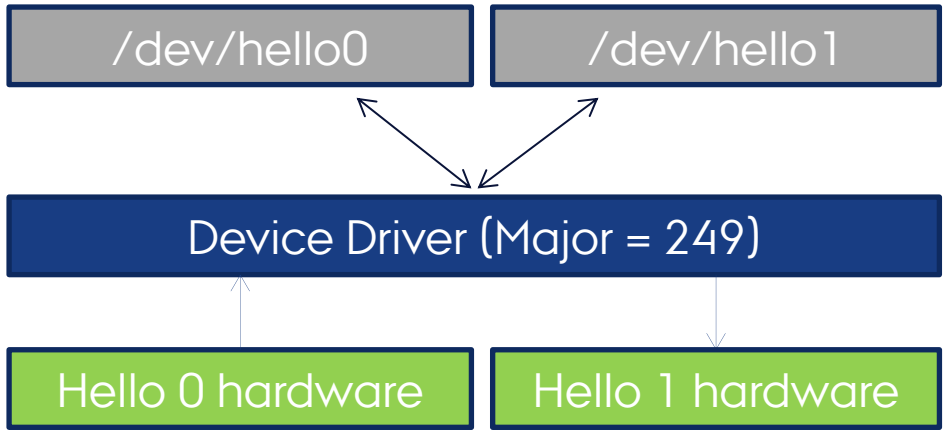
When module is inserted, the allocated major number can also be found with:

```
$ cat /proc/devices
239 hello-driver
```

CREATION OF NODES IN USER SPACE

- Character devices are represented in /dev/ by nodes (files) for data exchange
- The nodes can be created using the command mknod:

```
$ mknod /dev/hello0 c 249 0
$ mknod /dev/hello1 c 249 1
```



- This creates two files: */dev/hello0*, */dev/hello1* using minors *0,1* respectively and both using major *249*
- It can be checked by looking in the /dev folder:

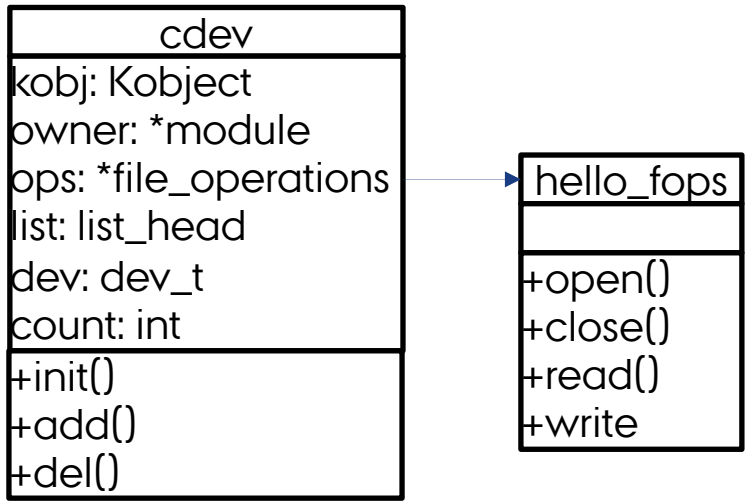
```
$ ls -l /dev/hello*
crw-rw----    1 root   root   249, 0 Jan  1  1970 /dev/hello0
crw-rw----    1 root   root   249, 0 Jan  1  1970 /dev/hello1
```

DRIVER REGISTRATION 1:2

```
…
#include <linux/cdev.h>
static struct class *hello_class;
static dev_t devno;
static struct cdev hello_cdev;
struct file_operations hello_fops;
static int __init hello_init(void)
{
…
  pr_info("Hello driver got Major %i\n", MAJOR(devno));

  hello_class = class_create(THIS_MODULE, "hello-class");
  if (IS_ERR(hello_class)) pr_err("Failed to create class");

  cdev_init(&hello_cdev, &hello_fops);
  err = cdev_add(&hello_cdev, devno, 255);
  if (err) pr_err("Failed to add cdev");

  return err;
}
```

DRIVER REGISTRATION 2:2

- **`class_create`**: creates a device class for our device, required for Linux's device model (later stuff). Class is visible in /sys/class
- **`cdev_init`**: Initializes struct cdev and sets ops (file operations) in cdev
- **`cdev_add`**: Adds cdev to Linux's list of character devices
- Remember to handle errors appropriately by cleaning up!!!
- Btw. **`IS_ERR`** is a preprocessor macro to check if a pointer is vaild



PREPROCESSOR MACROS

The pre-processor performs text manipulation on the source files prior to compilation
Text lines prepended by **#** is handled by the preprocessor

```
#include <stdio.h> /* is replaced by the actual text in the file stdio.h! */

#if 0
  pr_info(”this line will only print when #if 1”);
#endif

#define MIN_TO_SEC(x) ((x) * 60) /* Preprocessor macro (”function”) */

time_sec = MIN_TO_SEC(40);       /* Is replaced by time_sec = 2400 */
```

ERROR HANDLING MACRO

```c
/* Macro to handle Errors */
#define ERRGOTO(label, ...)                    \
  do {                                         \
  pr_err (__VA_ARGS__);                        \
  goto label;                                  \
  } while(0) /* No trailing ; */


static int __init hello_init(void)
{
  int err=0;

  err = alloc_chrdev_region(&devno, first_minor, max_devices, "hello-driver");
  if(MAJOR(devno) <= 0)
    ERRGOTO(err_end, "Failed to allocate dev number\n");
```

This is a Variadic Macro, as i takes a variable number of parameters.
The "..." input is passed directly to pr_err as __VA_ARGS__

The "\" is to break the line, as macros can only be in one line of text

do{} while(0) is a fix to let a macro work as a function. Without it we'd get a ";" too much, resulting in an error:

```c
If(MAJOR(devno) <=0)
  {pr_err("Failed to allocate dev number\n";
  goto err_end;}; /* ONE ; too much */
```

See http://c-faq.com/cpp/multistmt.html

ERROR HANDLING

We must handle cleanup ourselves!

In this particular case ”goto” can be helpful.

Use reverse order deallocating resources

```
static int __init hello_init(void)
{
  int err;
  /* Allocate resouces */
  err = allocate_it("A");
  if (err) goto fail_A;
  err = allocate_it("B");
  if (err) goto fail_B;
  err = allocate_it("C");
  if (err) goto fail_C;

  return 0; /* success */

  fail_C: deallocate_it("B");
  fail_B: deallocate_it("A");
  fail_A:
  return err; /* propagate the error */
}
```

MODULE DE-INITIALIZATION

The exit function must de-allocate resources allocated in init.

Allocation (init)	De-allocation (exit)
alloc_chrdev_region	cdev_del
class_create	NONE
cdev_init	class_destroy
cdev_add	unregister_chrdev_region

De-allocation must be done in reverse order to allocation

```
static void __exit hello_exit(void)
{
  /* Deallocate resouces */
  deallocate_it("C");
  deallocate_it("B");
  deallocate_it("A");
  return;
}
```

AARHUS
UNIVERSITY

LINUX CHARACTER DRIVERS -FILE OPERATIONS

FILE OPERATIONS

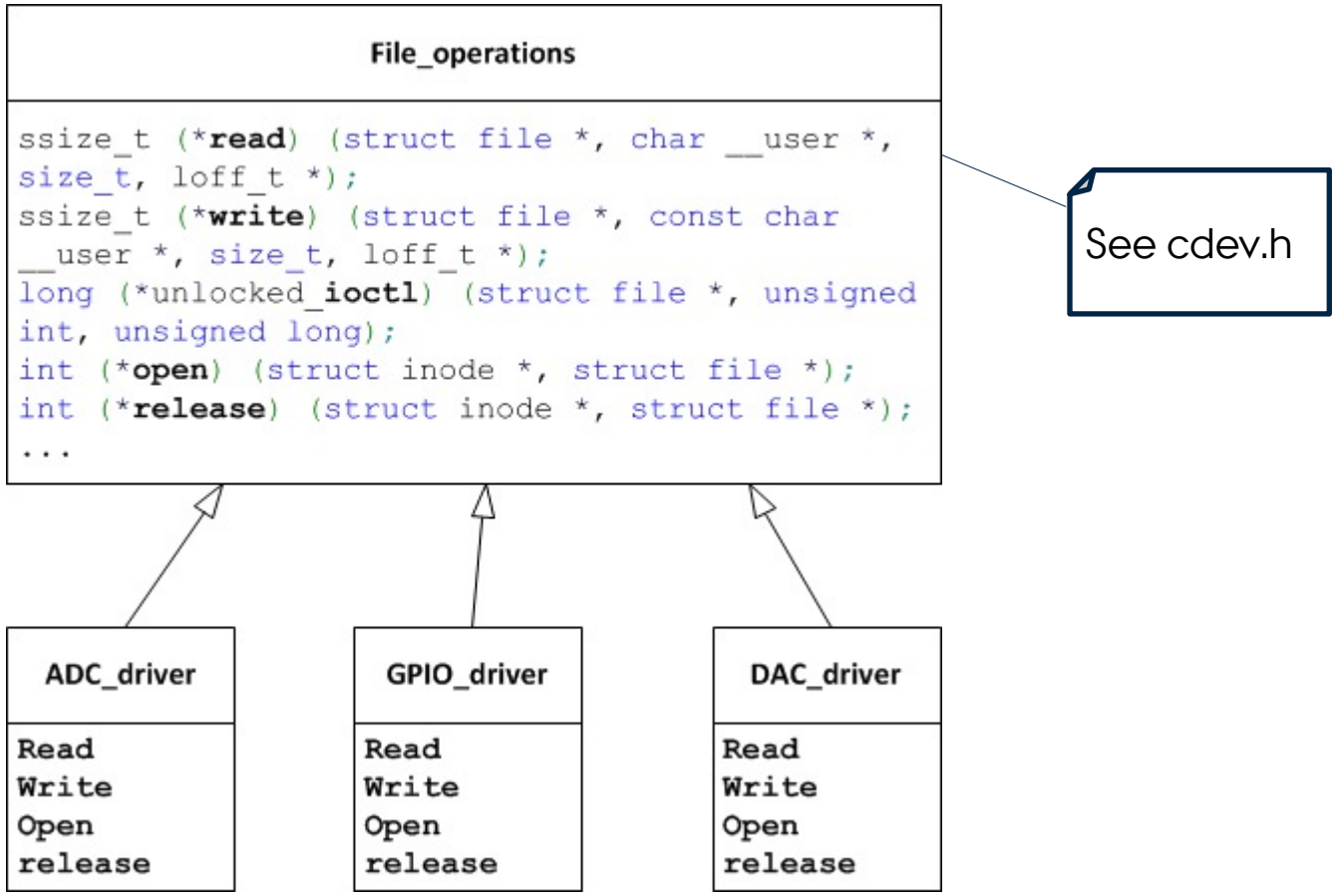
File operations are used to transfer data between device driver and user space application

A Character Device Driver has well-defined file operations associated

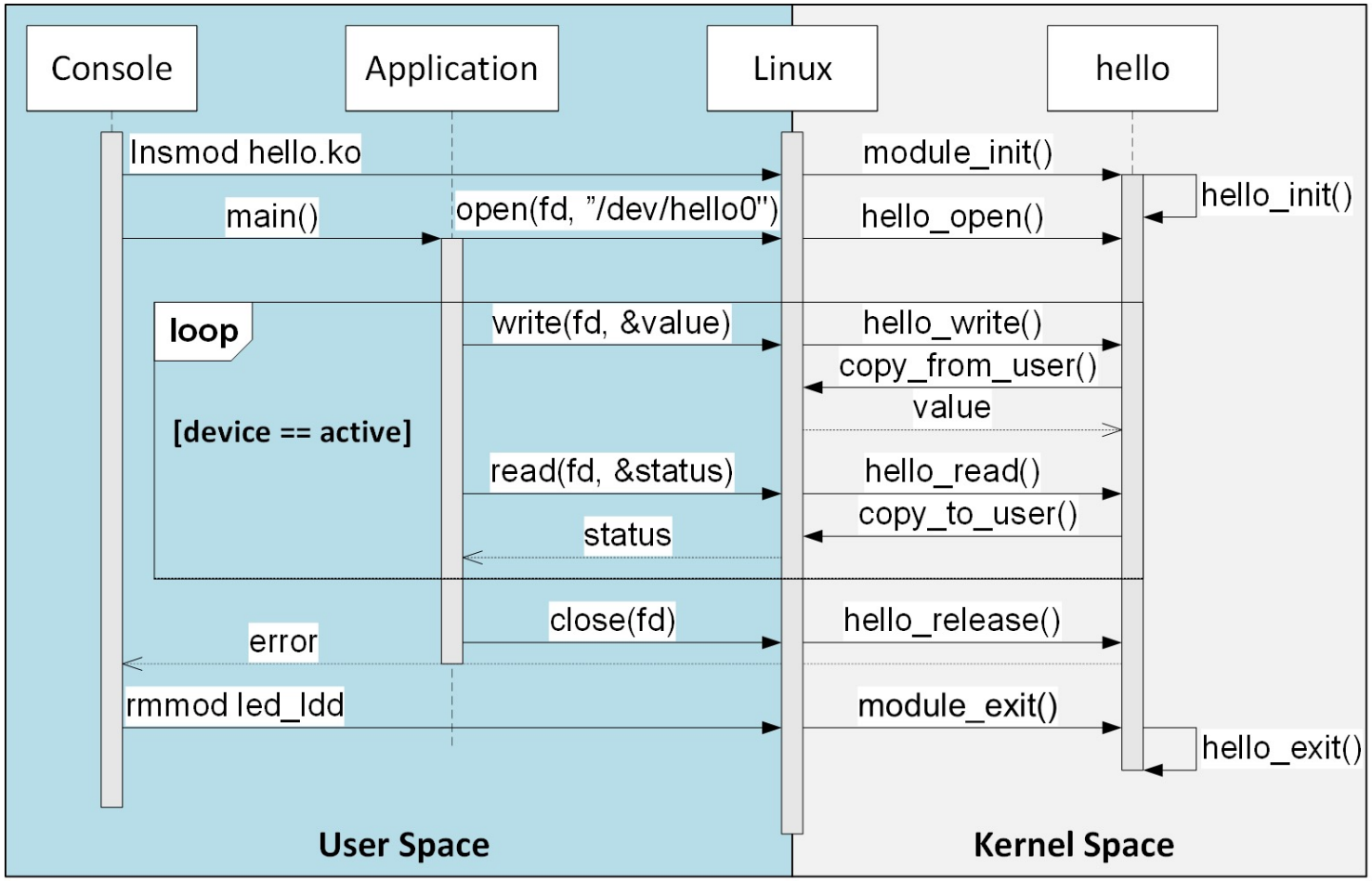
- You must implement what they do!
  - What shall read() do? –and how will shall it exchange data?
  - What shall write() do?

FILE OPERATIONS IN A DRIVER

- We must implement the concrete read/write/open/release operations



DRIVER LIFE CYCLE



CHAR DRIVER STRUCTURE

```
int hello_open(struct inode *inode, struct file *filep) {}

int hello_release(struct inode *inode, struct file *filep) {}

ssize_t hello_write(struct file *filep, const char __user *ubuf,
                    size_t count, loff_t *f_pos) {}

ssize_t hello_read(struct file *filep, char __user *ubuf,
                   size_t count, loff_t *f_pos) {}

struct file_operations hello_fops = {
  .owner   = THIS_MODULE,
  .open    = hello_open,
  .release = hello_release,
  .write   = hello_write,
  .read    = hello_read, };

static int __init hello_init(void) {}

static void __exit hello_exit(void){}

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

OPEN() OPERATION

Called when the device node is opened from user-space application:

```
fp = open(”/dev/hello0”, O_RDWR);)
```

User Space

You must implement the concrete tasks (as required):

- Check for device not ready errors
- Initialize device if opened for the first time
- Dynamically allocate per device memory to be used
  - (Private data can be used to handle concurrency)

```
int hello_open(struct inode *inode, struct file *filep)
{
    pr_info("Opening hello device [major], [minor]: %i, %i\n",
                MAJOR(inode->i_rdev), MINOR(inode->i_rdev));
    /* Example: Allocate dynamic per-device-memory */
    filep->private_data = kzalloc(sizeof(struct its_dev_data), GPF_KERNEL);
    return 0;
}
```

Kernel Space

RELEASE() OPERATION

Called when the device node is closed from user space application:

```
close(fp);
```

User Space

The Release method is in charge of cleaning up

- Relinquishing locks

Things to be done during release are:

- De-allocation of memory allocated during open
- Shut down the device on the last close

```
int hello_release(struct inode *inode, struct file *filep)
{
    pr_info("Closing hello device [major], [minor]: %i, %i\n",
                MAJOR(inode->i_rdev), MINOR(inode->i_rdev));
  /* Example: Deallocate per-device-memory */
    kfree(filep->private_data);
    return 0;
}
```

Kernel Space

FILE READ() OPERATION

Called when the device node is read from the user-spcae application:

```
readlen = read(fp, buffer, req_read_len);
```

User Space

Posix interface defines requirements you must implement in your read function

- You may read (and return) up to ”count” bytes
- You must return the number of bytes read, zero if EOF reached or a negative value on errors
- Also, it is best practice to move the cursor within the file (though we don’t use it)

```
ssize_t hello_read(struct file *filep, char __user *buf,
                   size_t count, loff_t *f_pos) {
  char kbuf[12];
  int len, value;
  value = some_function_to_get_value_from_device();
  len = count < 12 ? count : 12;   /* Truncate to smallest */
  len = snprintf(kbuf, len, ”%i”, value); /* Create string */
  copy_to_user(buf, kbuf, ++len);  /* Copy to user */
  *f_pos += len;                   /* Update cursor in file */
  return len; }                    /* Return length read */
```

Kernel Space

USER- / KERNEL SPACE DATA EXCHANGE

- The kernel- and user space has separate memory locations
- User space has only user access, no access to kernel memory. Trying ➔Page Fault
- User space memory is virtual and may be swappable
- Special functions must be used to copy between memory areas
- Copy from Kernel- to User Space:

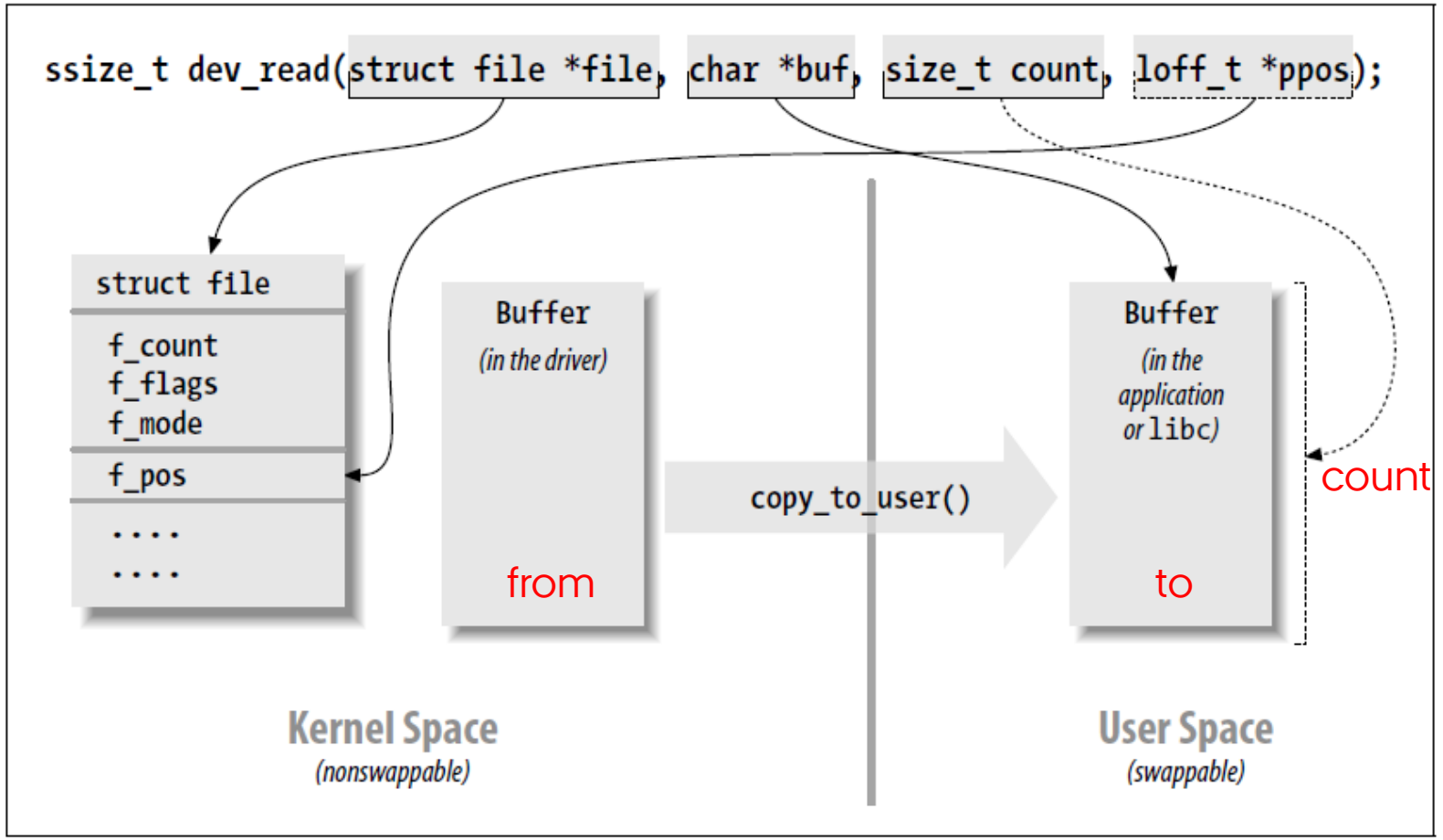
```
unsigned long __must_check copy_to_user(void __user *to, const void *from,
                                        unsigned long count);
```

- Copy from User- to Kernel Space:

```
unsigned long __must_check copy_from_user(void *to, const void __user *from,
                                          unsigned long count);
```

- __user and __must_check are hints to the compiler.
  You must check return value, or you will get a warning!

USER- / KERNEL SPACE DATA EXCHANGE




```
err = copy_to_user(to, from, count);
```

FILE WRITE() OPERATION

Called when the device node is written to from user space application:

```
written_len = write(fp, buffer, req_write_len);
```

- "write" must return the number of bytes written, zero if none written or negative value on errors

```
ssize_t hello_write(struct file *filep, const char __user *ubuf,
                    size_t count, loff_t *f_pos) {
  int len, value, err = 0;
  char kbuf[12];
  len = count < 12 ? count : 12; /* Truncate to smaller buffer size */

  if(copy_from_user(kbuf, ubuf, len) < 0)
    return -EFAULT;

  if (kstrtoint(kbuf, 0, &value)) /* Convert sting to int */
    pr_err("Error converting string to int\n");

  pr_info("Wrote %i from minor %i\n", value, iminor(filep->f_inode));
  *f_pos += len; /* Update cursor in file */
  return len; }  /* return length actually written */
```

DEFAULT FILE OPERATIONS

You don't always have to implement all operations!

- Open / Release has default implementations, if you don't need to do anything special!
- If your driver only supports reading, leave the write operation out. The file-node will only support reading. Writing to it will return an error.

```
ssize_t hello_read(struct file *filep, char __user *ubuf,
                   size_t count, loff_t *f_pos) {}

struct file_operations hello_fops = {
  .owner    = THIS_MODULE,
  .read     = hello_read, };

static int __init hello_init(void) {}

static void __exit hello_exit(void){}

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

AARHUS
UNIVERSITY

AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

PETER HØGH MIKKELSEN
6. MARCH 2018

INTERRUPT MANAGEMENT

INTERRUPTS

Signals until reset by system

ABS Sensor

Fire and forget

Interrupt

Ack

Level-Triggered

Interrupt

Edge-Triggered

System to be notified

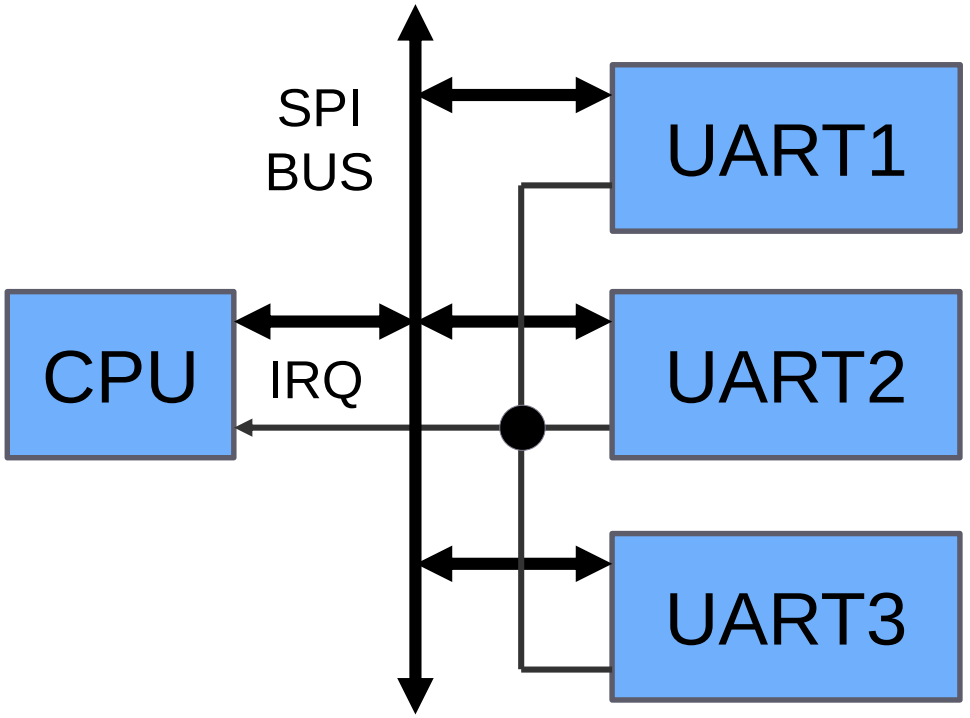
EXTERNAL INTERRUPT TYPES

- Edge-Triggered Interrupts
- The Interrupt is activated by an IRQ signal transition
- ISR is called only one time, hence it does not block
- Depending on architecture, IRQs may be disabled while in ISR
- Good for short (or very long) IRQ signals that doesn't need ACK

- Level-Triggered Interrupts
- The Interrupt is active as long as IRQ is active
- Interrupt Service Routing (ISR) must acknowledge IRQ source, to let it deactivate the IRQ line
- Good for shared IRQ lines

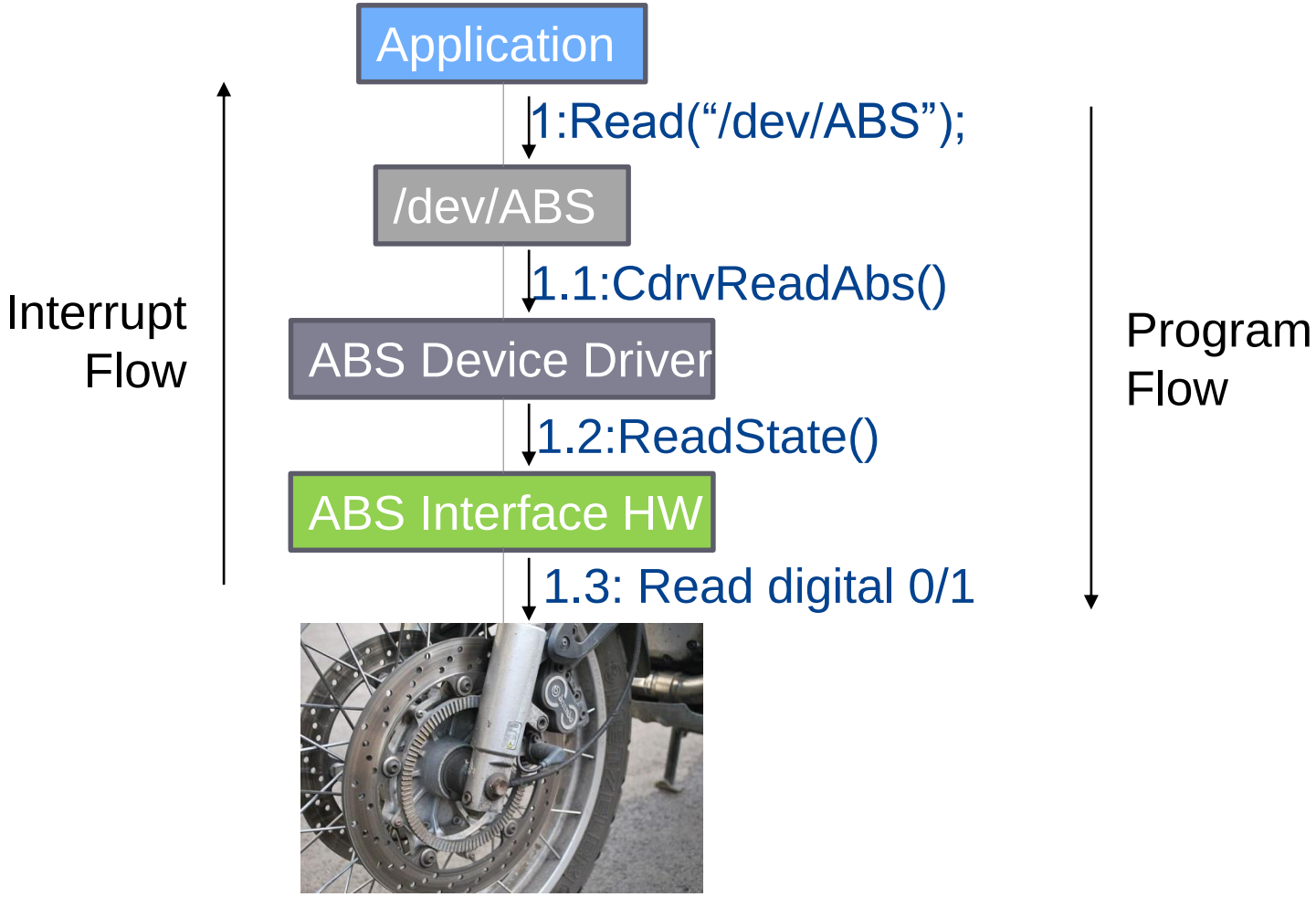
SHARED INTERRUPTS

```
void ISR() {
   ..
   [Who's the IRQ source]
   [Do Something]
   [ACK to IRQ source]
   IRET;
}
```

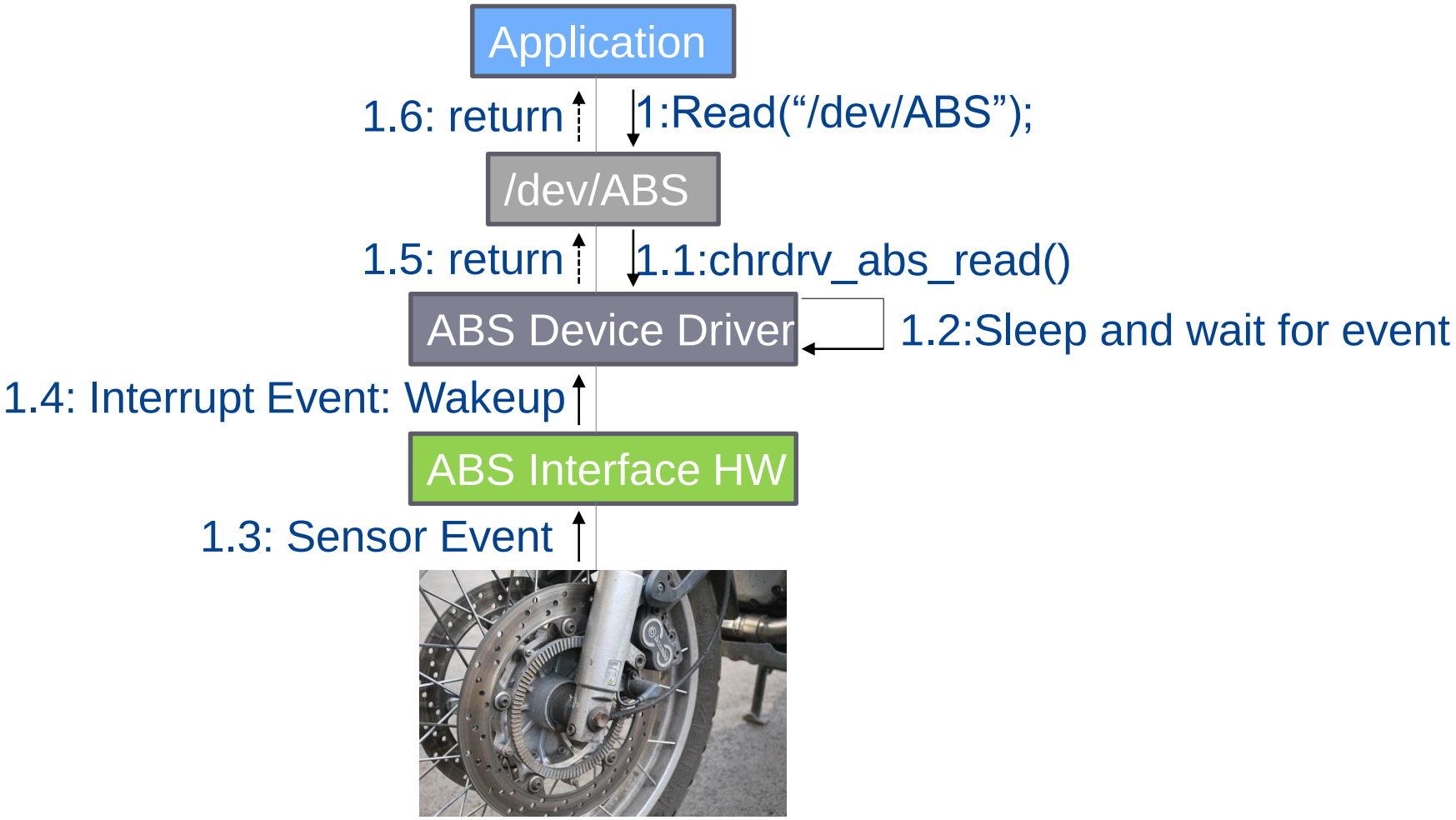


- Interrupt lines are typically sparse and should be preserved
- ISR will continue to be called until all sources has been serviced
- Only possible with Level-Triggered Interrupts
- Clients typically have an IRQ register that can be read by the CPU

PROGRAM FLOW IN LINUX



INTERRUPT FLOW IN LINUX



BLOCKING I/O

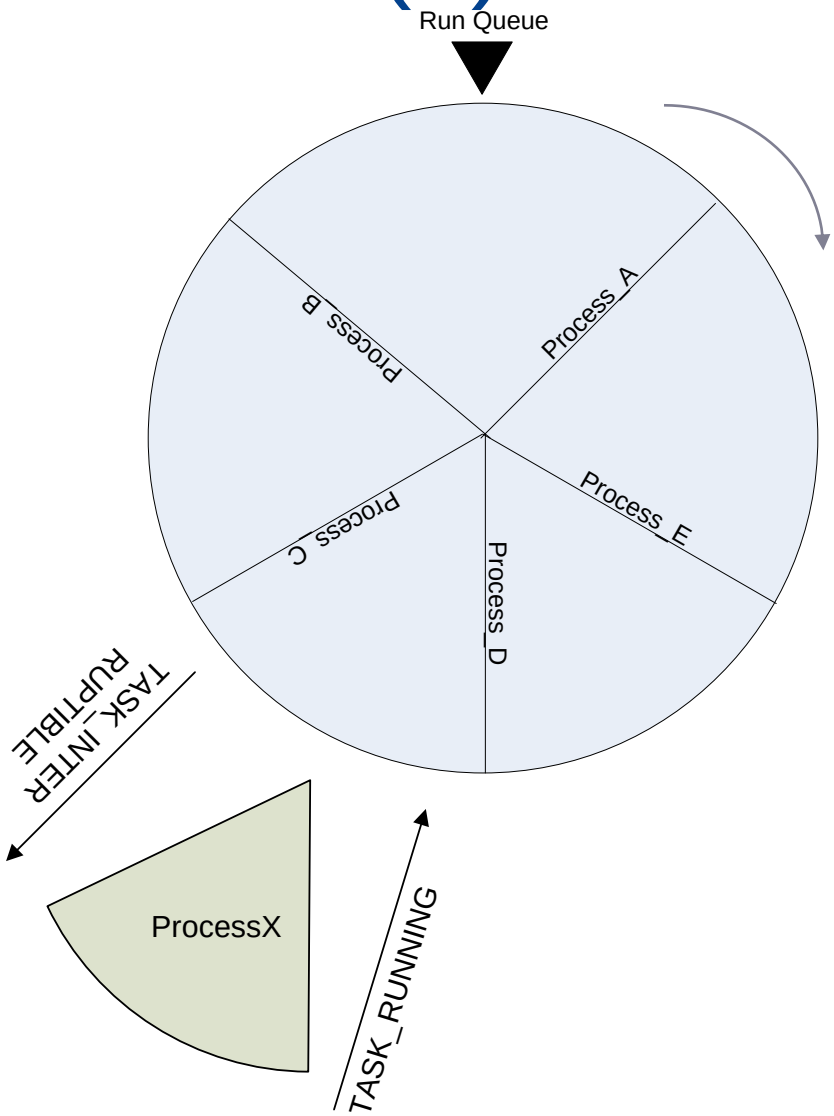
- Linux uses blocking I/O access, to pause applications, while data is becoming ready
- In kernel space, the fops operations (read/write), can put the calling application to sleep (and wake it up) using special functions.
- In user space, the application accessing the device node, and thereby the driver, is put to sleep and later woken up, when data is available as decided in the driver
- Interrupt functions are ONLY available in the kernel!

SLEEPING IN THE KERNEL (1)

- The Linux kernel has a run-queue with active processes
- Each process is devised a time-slice, based on its priority

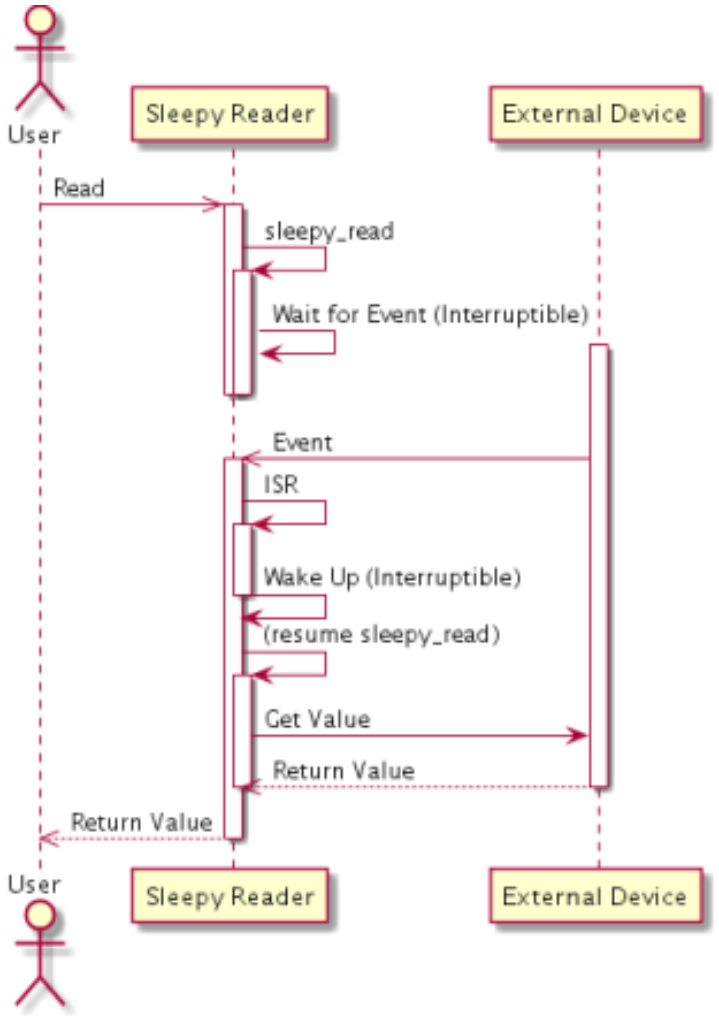
- Active process in the run-queue is marked as TASK_RUNNING
- A sleeping process is taken out of the run-queue and marked as TASK_(UN)INTERRUPTIBLE

- When woken up, the process is put into the run-queue again and executed when decided to by the schduler



SLEEPING IN THE KERNEL (2)

- To implement sleep / wake-up we must use:
- A *wait queue* – to pass events from wake_up to wait_event…
- A wake_up method, to be called when an event occur (ex in an ISR)
- A wait method, that sleeps until woken up by an event passed through the wait queue. This could ex be inside the read method



SLEEPING IN THE KERNEL (3)

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;
ssize_t sleepy_read (struct file *filp, char *buf,
                     size_t count, loff_t *pos){
  ...
  wait_event_interruptible(wq, flag != 0);
  flag = 0;
  ...
  return 0;
} /* EOF */

ssize_t sleepy_write (struct file *filp, const char *buf,
                      size_t count, loff_t *pos){
  ...
  flag = 1;
  wake_up_interruptible(&wq);
  return count;
} /* succeed, to avoid retrial */
```

INTERRUPT HANDLING

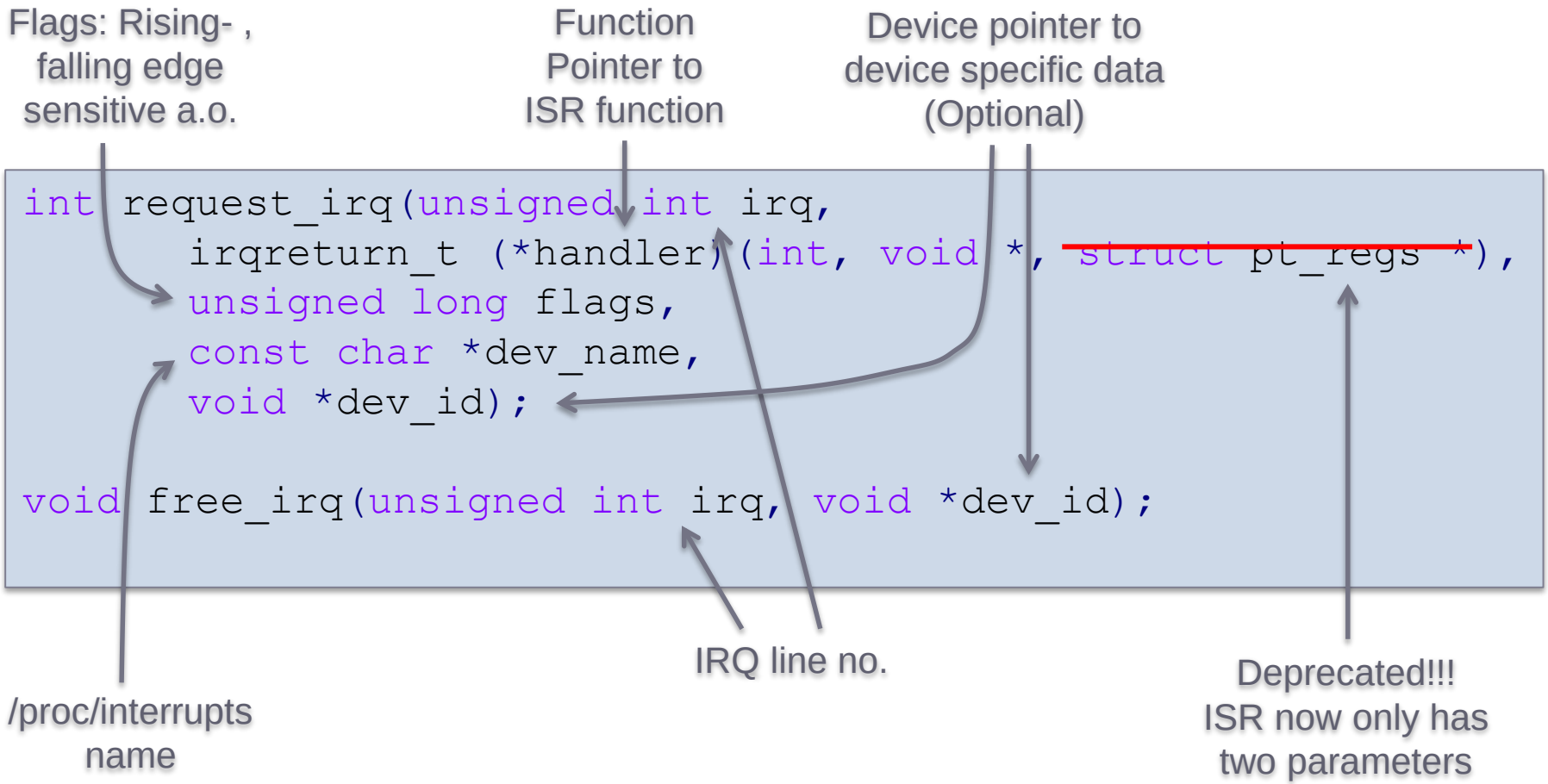
› A module must request an IRQ line before using it and to release it afterwards

› Request the IRQ line in the "**open**" method
› If the device is not opened, we won't listen to the IRQ anyway
› If put in the "init" method, the line will be occupied as soon as the module is inserted in the kernel (OK for debugging)

› Release the IRQ line in the "**release**" method.

› A driver with several minor numbers can share a line
› Special care must be taken with the request/release methods

REQUEST_IRQ / FREE_IRQ



/PROC

```
root@rpi:~# cat /proc/stat
cpu  142 0 530 73528 1269 5 0 0 0
cpu0 142 0 530 73528 1269 5 0 0 0
intr 120154 0 0 0 0 0 0 0 0 0 0 0 577 0 0 0 0 0 ...
0 0 0 385 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 232 0 ...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
ctxt 56055
btime 1233051572
processes 1343
procs_running 2
procs_blocked 0
```

Total Events

IRQ0

IRQ12

/PROC

```
root@rpi:~# cat /proc/interrupts
           CPU0
 11:          0          INTC  prcm
 12:        577          INTC  DMA
 56:        385          INTC  i2c_omap
 74:        232          INTC  serial
 77:          0          INTC  ehci_hcd:usb1
 83:      13231          INTC  mmc0
 95:       7016          INTC  gp timer
186:          0          GPIO  user
187:          0          GPIO  ads7846
369:          0      twl4030  twl4030_keypad
378:          0      twl4030  twl4030_usb
Err:          0
```

IRQ Line | # Events | IRQ Controller | dev_name

DETECTING IRQ NUMBERS

- At hardware level, devices are attached to IRQ lines on an interrupt controller
- In the driver, we need to retrieve the IRQ line number, to be able to install a handler.

- This can be done using several methods:
- The interrupt controller may provide methods for retrieving the number
- The number may be static and queried for using ***platform_get_irq()***.
- HW Interrupt lines are defined in the Device Tree
- ***`irq = gpio_to_irq(gpio_no);`***

- Interrupt probing
- Check for available IRQs
- Generate an interrupt event
- Check for a change in the available IRQs
- Manually or using kernel methods

IRQ DRIVER INIT

```
static int my_drv_init(void) {

  gpio_request(…
  gpio_direction_input(…

  … MKDEV(…
  register_chrdev_region(…

  cdev_init(…
  cdev_add(…

  // Request and ENABLE interrupt
  request_irq(gpio_to_irq(121), my_handler,
       IQRF_TRIGGER_FALLING, “my_irq”, NULL);
```

request_irq could also be placed in my_drv_open()

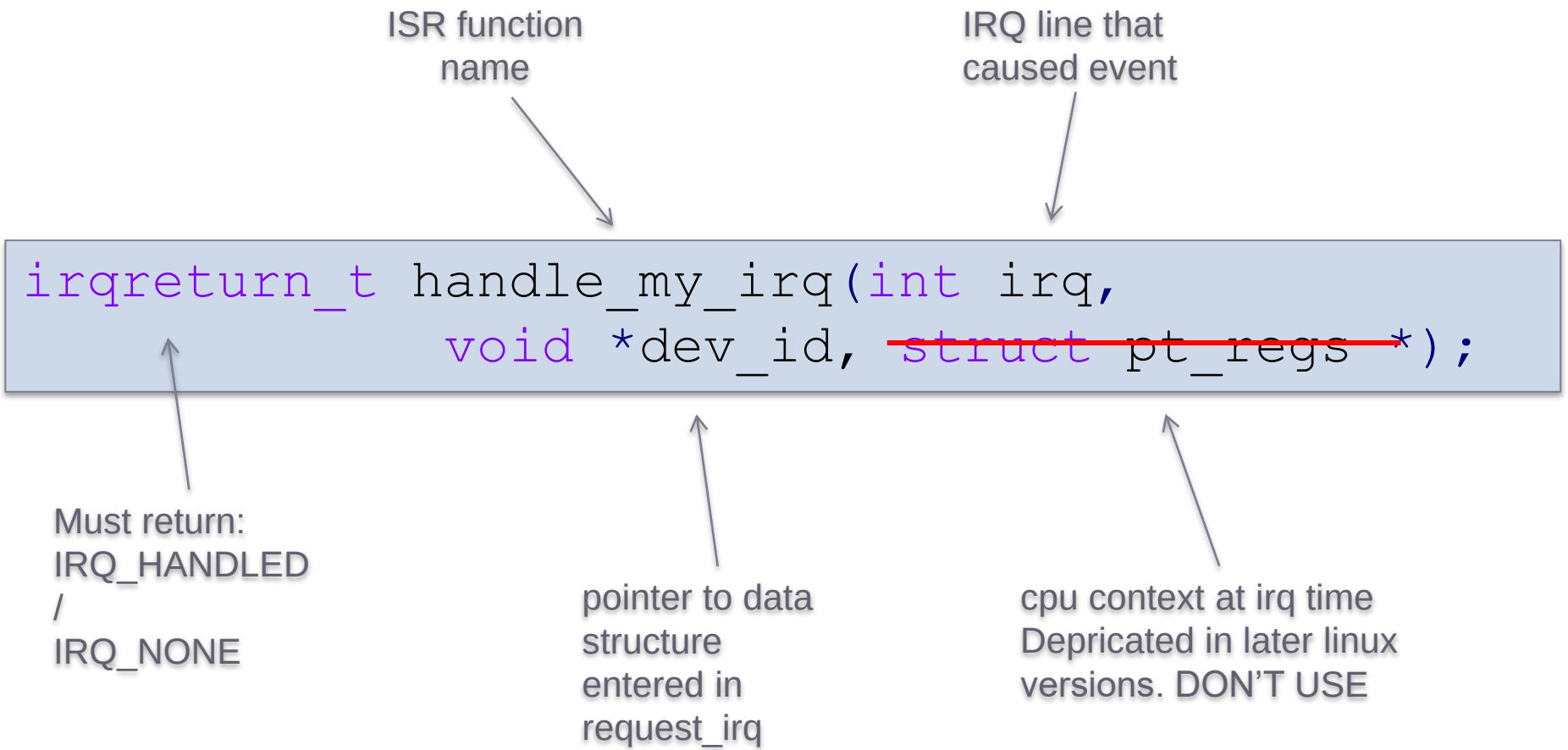
INTERRUPT HANDLER

› The interrupt handler is called upon an interrupt event

› The handler is plain C-code, but:
› The handler must be FAST
› It must not do anything that can sleep
› Do *not* allocate memory (its possible for experts)

› The typical ISR must:
› Wake-up the interrupting device
› Acknowledge the “irq_pending” bit on the device
› Acquire data from the interrupting device
› Wake-up a sleeping function in the driver (ex read)
› Exit with IRQ_HANDLED or IRQ_NONE

INTERRUPT HANDLER



ISR EXAMPLE

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;
irqreturn_t handle_my_irq(int irq, void *dev_id){
  [ack irq pending]
  [readbuf = read value from device (if fast)]
  flag = 1;
  wake_up_interruptible(&wq);
  return IRQ_HANDLED;
}
ssize_t my_read(...){
  wait_event_interruptible(wq, flag == 1);
 [readbuf = read value from device (slow/can block)]
  flag = 0;
  copy_to_user(buf, readbuf, sizeof(readbuf));
  return sizeof(readbuf);
}
```

SHARED INTERRUPTS

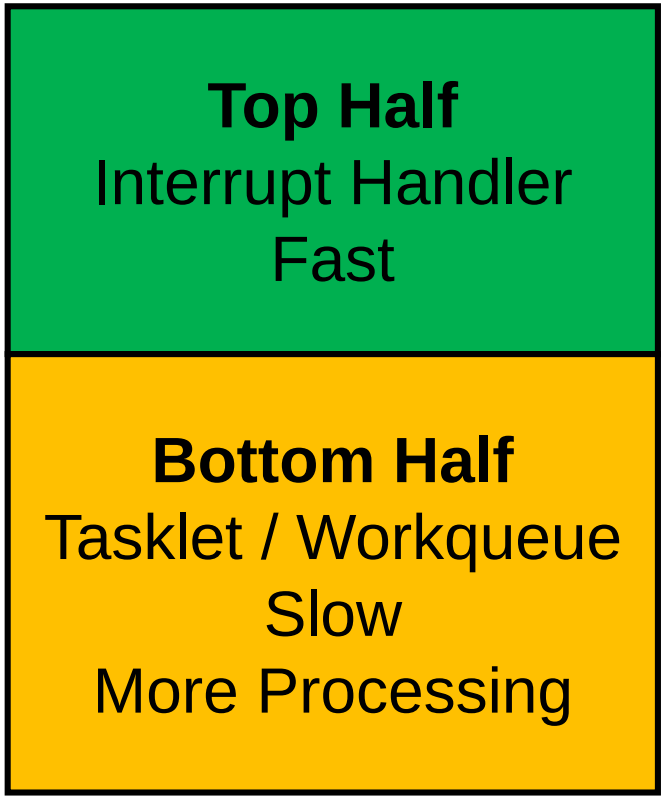
- To minimize the number of IRQ lines used, they may be shared
- request_irq(irq_line, isr, IRQF_TRIGGER_HIGH|**IRQF_SHARED**, “irq_name”, irq_dev_id)
- The ISR should check the IRQ source, and return IRQ_NONE if it wasn’t its device who was the source

```
irqreturn_t my_shared_irq_isr(int irq, void *dev_id) {
  [Get IRQ source]
  [Was IRQ NOT for me?]
     return IRQ_NONE;
  [ack irq pending]
  [readbuf = read value from device(if fast/non-block)]
  flag = 1;
  wake_up_interruptible(wait_queue);
  return IRQ_HANDLED;    }
```

TOP- & BOTTOM HALVES

- Top-Half:
- ISR created with request_irq()
- Ack device irq_pending
- Read device data to buffer
- Schedule tasklet / workqueue / threaded IRQ
- IRQ can be disabled

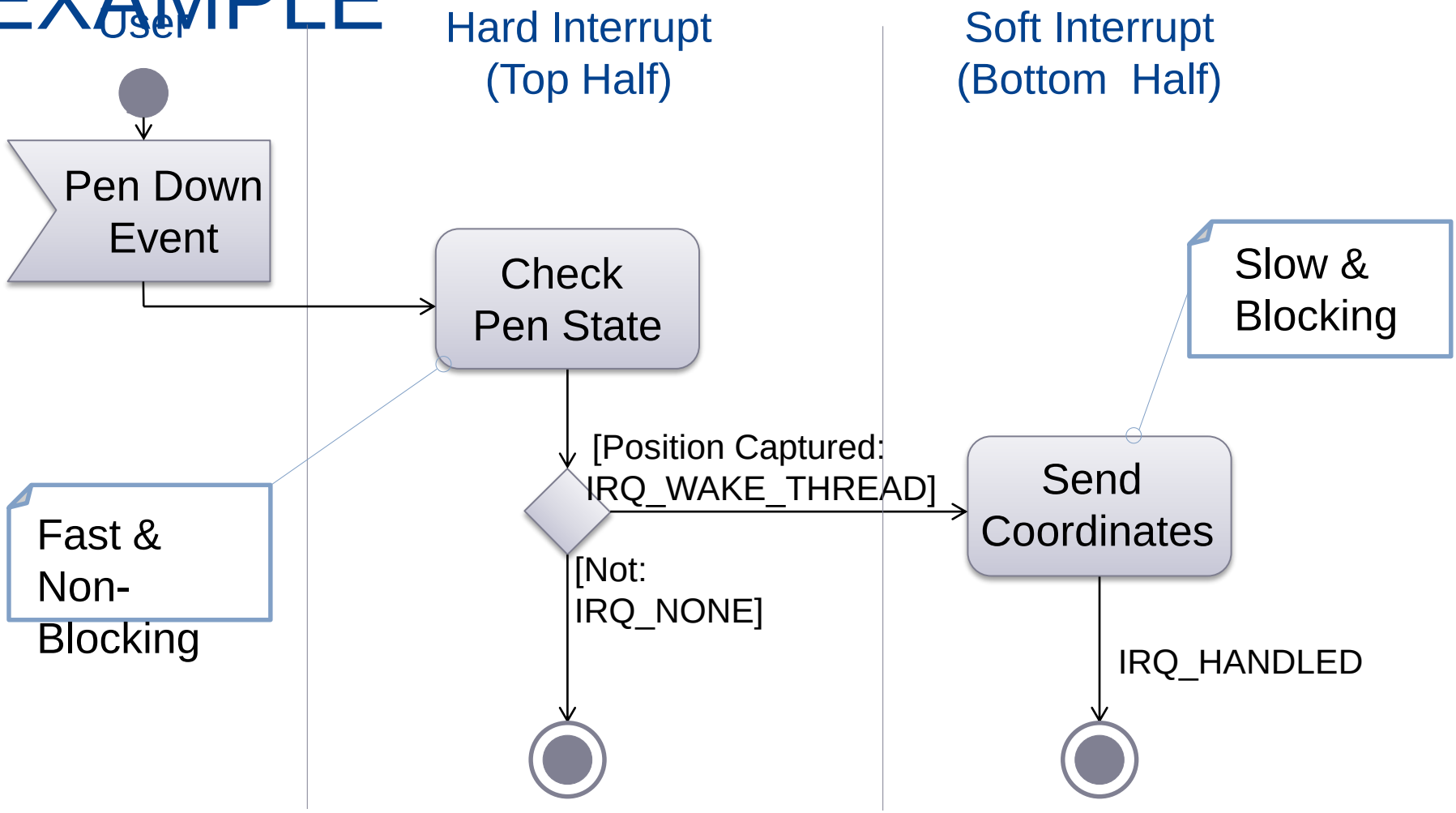
- Bottom-Half
- Scheduled by the kernel
- Handles all heavier processing
- IRQs enabled during processing
- Implemented as tasklet, workqueue or threaded IRQs



THREADED INTERRUPTS

```
irqreturn_t isr_hard (int irq, void *dev_id){
 /* Handle hard IRQ */
 /* Fast time-critical not-blocking stuff */
 return IRQ_WAKE_THREAD;
}

irqreturn_t isr_proc(int irq, void *dev_id){
 /* Handle soft IRQ process */
 /* Stuff that may be deferred in time, blocking etc */
 return IRQ_HANDLED;
}

static int my_cdev_init(…) {
  request_threaded_irq(gpio_to_irq(7), isr_hard,
                       isr_proc, IQRF_TRIGGER_RISING,
                       ”myThreadedIRQ”, NULL);
```

TOUCHPAD THREADED IRQ. EXAMPLE



AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

14. NOVEMBER 2018
PETER HØGH MIKKELSEN

LINUX DEVICE MODEL

DRIVERS SO FAR (IN THIS COURSE)

- Init:
- Allocate specific number of devices

```
static int ldd_init(…){
 err = gpio_request(112);
 err = gpio_request(911);
 register_chrdev_region(MKDEV(64,0),
                        2, ”ldd”);
```

- Read/Write:
- Use minor to switch between devices

```
ssize_t ldd_write(…){
 int minor = iminor(filep->f_inode);
 if (minor == 0)
  val = gpio_get_value(112);
 else if (minor == 1)
  buf = gpio_get_value(911);
```

- Mknod:
- Manual invocation per device

```
# mknod /dev/gpio112 c 64 0
# mknod /dev/gpio911 c 64 1
```

- *Old School Solution*
- *Manual allocation of resources*
- *No Plug-and-Play*

MODERN DRIVER REQUIREMENTS

- Power Management
- Respond to system states (Shutdown, Suspend, Wakeup etc)

- Classification of devices
- Higher device abstraction
- Per class or sub-device configuration and control

- Hot-plugging
- Automated discovery
- Automated driver binding
- Automated device creation in user-space

LINUX DEVICE MODEL

- Power Management
- Call-back methods in driver to handle changes in system state (Suspend, Resume, etc)

- Classification of Devices
- Support for device classes and devices

- Hot-Plugging
- Support for USB, PCI etc. that supports discovery
- Call-back methods in driver to handle hot-plugged devices (Probe, Release)
- Introduction of SysFS, to represent system hardware and drivers in user space and to allow dynamic node creation

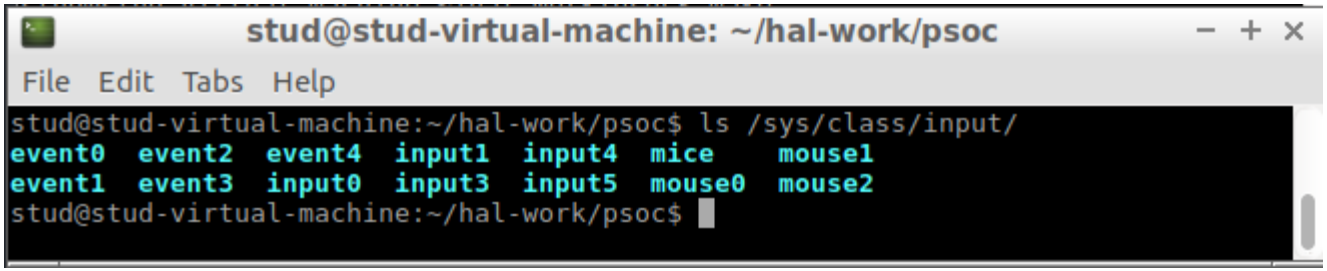
SYSFS

- Virtual file system that visualizes the internal device/driver structure
- Located at /sys/ and created by the kernel at boot time
- Updated when special events occur in underlying drivers, i.e. provides a means to hot-plug hardware

- Reflects the complex structure of composite devices
- `/sys/class/` - Objects organized according to function
- `/sys/bus/` - Objects organized according to bus hierarchy
- `/sys/device/` - Maps directly to the internal kernel device tree
- `/sys/block/` - All block devices present in the system
- ...

DEVICE CLASS

- A class of devices with the same base functionality and properties
- /sys/class/input contain input devices with common properties



- A Class is created by invoking:

Returns devno (major+minor) dynamically allocated by kernel

```
static int my_leds_init(void){
…
  alloc_chrdev_region(&devno, 0, my_num_leds, "my_leds");
  my_led_class = class_create(THIS_MODULE, "my_led_cls");
…
```

- This will create a folder: /sys/class/my_led_cls

DEVICE

- A concrete device of a certain class with possible added functionality and properties
- Example: /sys/class/input/mouse0
- Can have special configuration parameters or properties
- Device of a device class – Little confusing…
- Think: Object of Class

- A *Device* is created by invoking:

Minor no. you wish to associate to device

```
static int my_leds_init(void){
  alloc_chrdev_region(&devno, 0, my_num_leds, "my_leds");
  my_led_class = class_create(THIS_MODULE, "my_led_cls");
…
  my_led_device_101 = device_create(my_led_class, NULL,
       MKDEV(MAJOR(devno), my_minor), &dev_data, "my_led%d", 101);
```

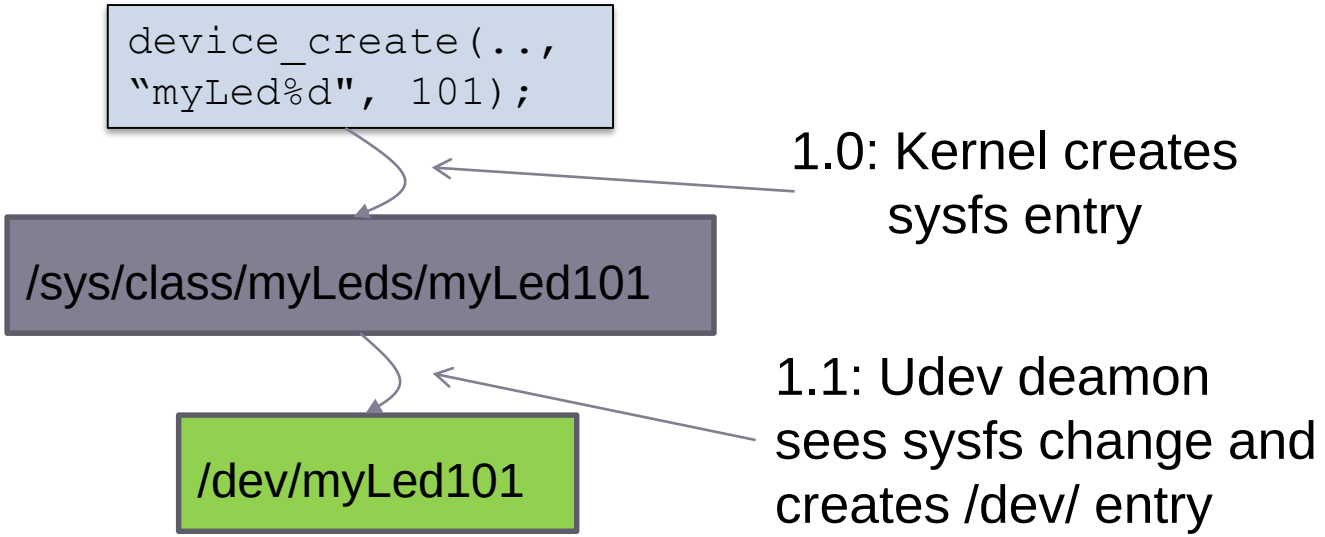
- This creates a file: /sys/class/my_led_cls/my_led101

UDEV (1:2)

- User Level Device Management
- Builds upon *sysfs* information
- Places policy in user space
- Dynamic node creation and naming (/dev/) depending on devices presence
- Uses daemons (services) to monitor /sys/

UDEV (2:2)

- Uses rules in /etc/udev/rules.d/ to create nodes from sysfs entries:
- names + access-rights
- Well-defined names independent of how and when devices are attached
- Try: *udevadm info –a –p /sys/block/sda*



AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

14. NOVEMBER 2018
PETER HØGH MIKKELSEN

LINUX DEVICE MODEL

HOT- & COLD-PLUGGING

- Hot-plugging – Device is added at run-time
- Ex: PCI / PCI-E / USB devices
- Devices are discoverable and provide information about which driver to bind to (plug & play)

- Cold-plugging – Device is present at boot-time
- I2C-, SPI- and devices embedded in SOC have no discovery information, which must be provided explicitly
- Earlier defined in board config files (ex. linux/arch/arm/march-omap2/…)
- Now defined in device-tree
- Device tree is loaded at boot-time

- Device model covers both
- Cold-plugging is handled like hot-plugging at boot-up

PLATFORM DRIVER

- For devices embedded in the SOC, or devices connected directly to the CPU's address/data busses.
- Ex. GPIO, battery chargers, Ethernet controllers
- Devices are typically present at boot-time, and are non-discoverable, so driver information must be provided explicitly during boot

PLATFORM DRIVER

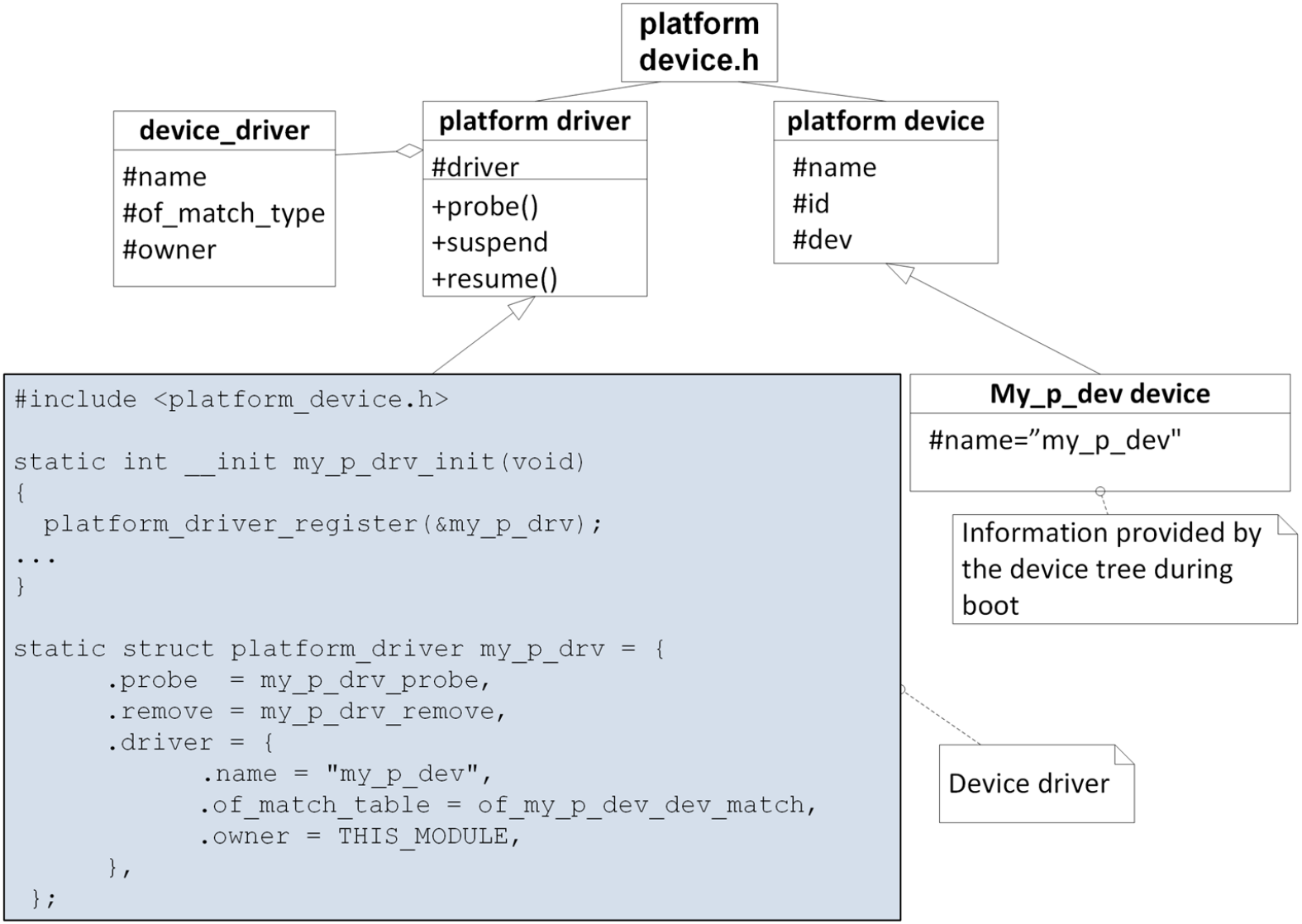
```
//platform_device.h
struct platform_driver {
 int    (*probe)(struct platform_device *dev);
 int    (*remove)(struct platform_device *dev);
 void   (*shutdown)(struct platform_device *dev);
 int    (*suspend)(struct platform_device * dev,
                   pm_message_t mesg);
 int    (*resume)(struct platform_device *dev);
 struct device_driver driver;
};
```

System states

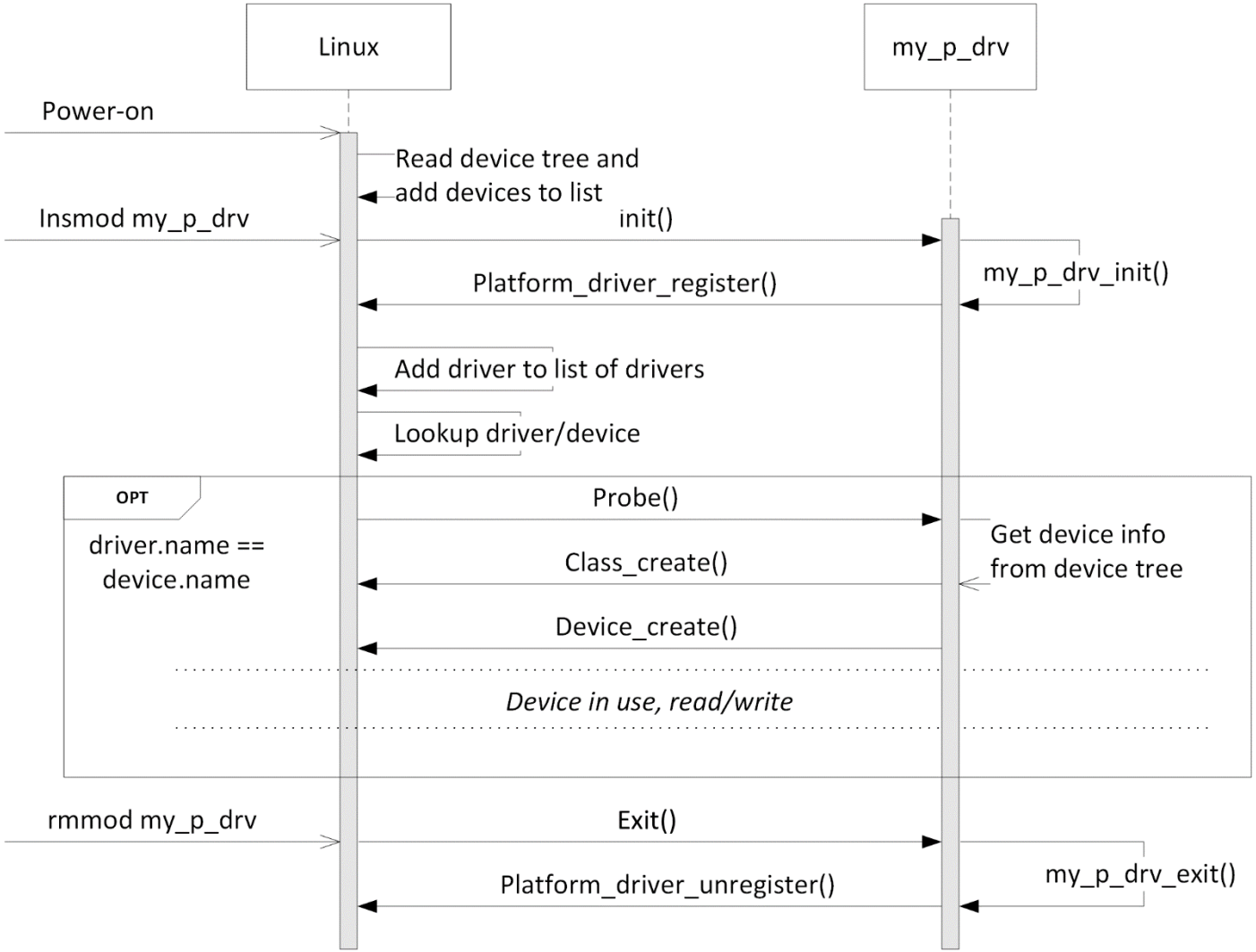
```
//device.h
struct device_driver {
 const char *name;
 struct module *owner;
 struct of_device_id *of_match_table;
 …
};
```

- Identification for old-style device/driver binding
- Identification for device-tree device/driver binding

PLATFORM DEVICE/DRIVER



DEVICE/DRIVER BINDING



DEVICE/DRIVER BINDING

```
//File: my_led.c - Device Driver
static const struct of_device_id my_led_platform_device_match[] =
{
   { .compatible = "ase, my_led",}, {},
};

static struct platform_driver my_led_platform_driver = {
        .probe  = mydevt_probe,
        .remove = mydevt_remove,
        .driver = {
                .name   = "my_led_old_naming",
                .of_match_table = my_led_platform_device_match,
                .owner = THIS_MODULE, }, };
```

Compatible!

```
//File: my_led-overlay.dts - Device Specification
 fragment@0 {
          target-path = "/";
            __overlay__ {
              mydevt: mydevt {
                /* Label to match in driver */
                compatible = "ase, my_led";
```

PLATFORM DRIVER STRUCTURE

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

my_led_probe()
{// Init ressources dynamically and create device(s)}

my_led_remove()
{// Deallocate ressources dynamically and delete device(s)}

static const struct of_device_id my_led_platform_device_match[] =
{
   { .compatible = "my_led",}, {},
};

static struct platform_driver my_led_platform_driver = {
        .probe  = mydevt_probe,
        .remove = mydevt_remove,
        .driver = {
                .name   = "my_led_old_naming",
                .of_match_table = my_led_platform_device_match,
                .owner = THIS_MODULE, }, };
// Continued...
```

PLATFORM DRIVER STRUCTURE

```
// ...Continued
my_led_read(…) {}

my_led_write(…) {}

struct file_operations my_fops = {
    .read  = my_led_read,
    .write = my_led_write, };

my_led_init(void){
 alloc_chrdev_region(); cdev_init(); cdev_add();
 class_create();
 platform_driver_register(&my_led_platform_driver); }

my_led_exit(void){
 platform_driver_unregister(&my_led_platform_driver); }
 class_destroy();
 cdev_del(); unregister_chrdev_region(); }

module_init(my_led_init)
module_exit(my_led_exit)
```

PROBE *-Method called when there is device + driver match*

```
static const u8 gpio = 10;
static const u8 minor = 0;

static int my_p_drv_probe(struct platform_device *pdev){

 printk(KERN_DEBUG "New Platform device: %s\n", pdev->name);

 /* Request resources */
 gpio_request(gpio, "my_p_dev_gpio")

 /* Dynamically add device */
 gpio_device = device_create(my_gpio_class, NULL,
                             MKDEV(major_from_alloc_region, minor),
                             NULL, "mygpio%d", gpio);
return err;}
```

Maps minor 0 to /dev/gpio[10]

```
static int my_p_drv_init(void){
  alloc_chrdev_region(&devno, 0, my_num_gpios, "my_p_drv");
  my_gpio_class = class_create(THIS_MODULE, "my_gpio_cl");
```

REMOVE *-Method called when device or driver is removed*

```
static int my_p_drv_remove(struct platform_device *pdev)
{
  printk (KERN_ALERT "Removing device %s\n", pdev->name);

  /* Remove device created in probe, this must be
   * done for all devices created in probe */
  device_destroy(gpio_class,
                 MKDEV(major_from_alloc_region, minor));

  gpio_free(gpio);
  return 0;
}
```

AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

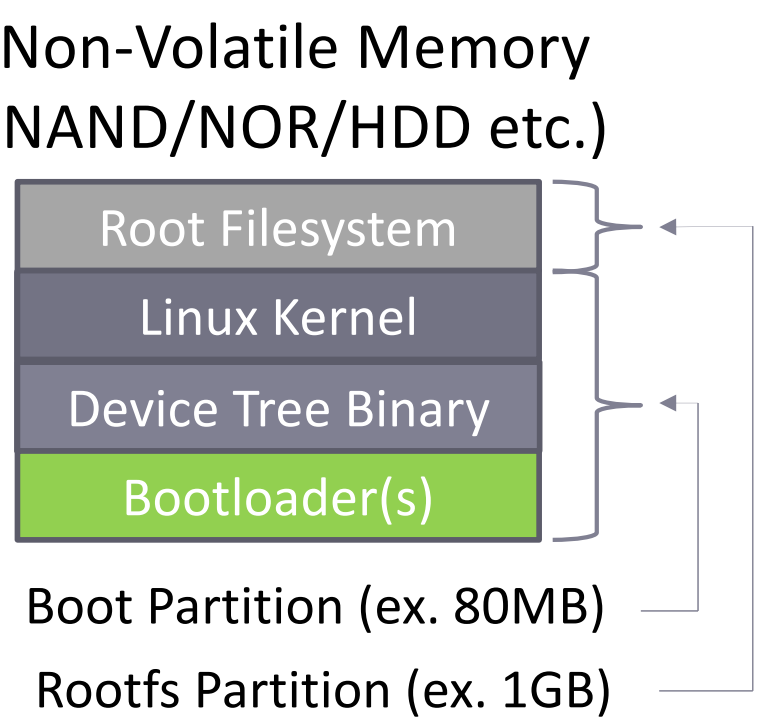
5.NOVEMBER 2021
PETER HØGH MIKKELSEN

DEVICE TREE

LINUX IMAGE

› A Linux image consists of:
- › Bootloader(s) to perform basic initialization and to boot the Linux kernel
- › The Linux kernel binary
- › A root file system with libraries and applications installed

› Recent versions also contains a Device Tree binary (Blob!)

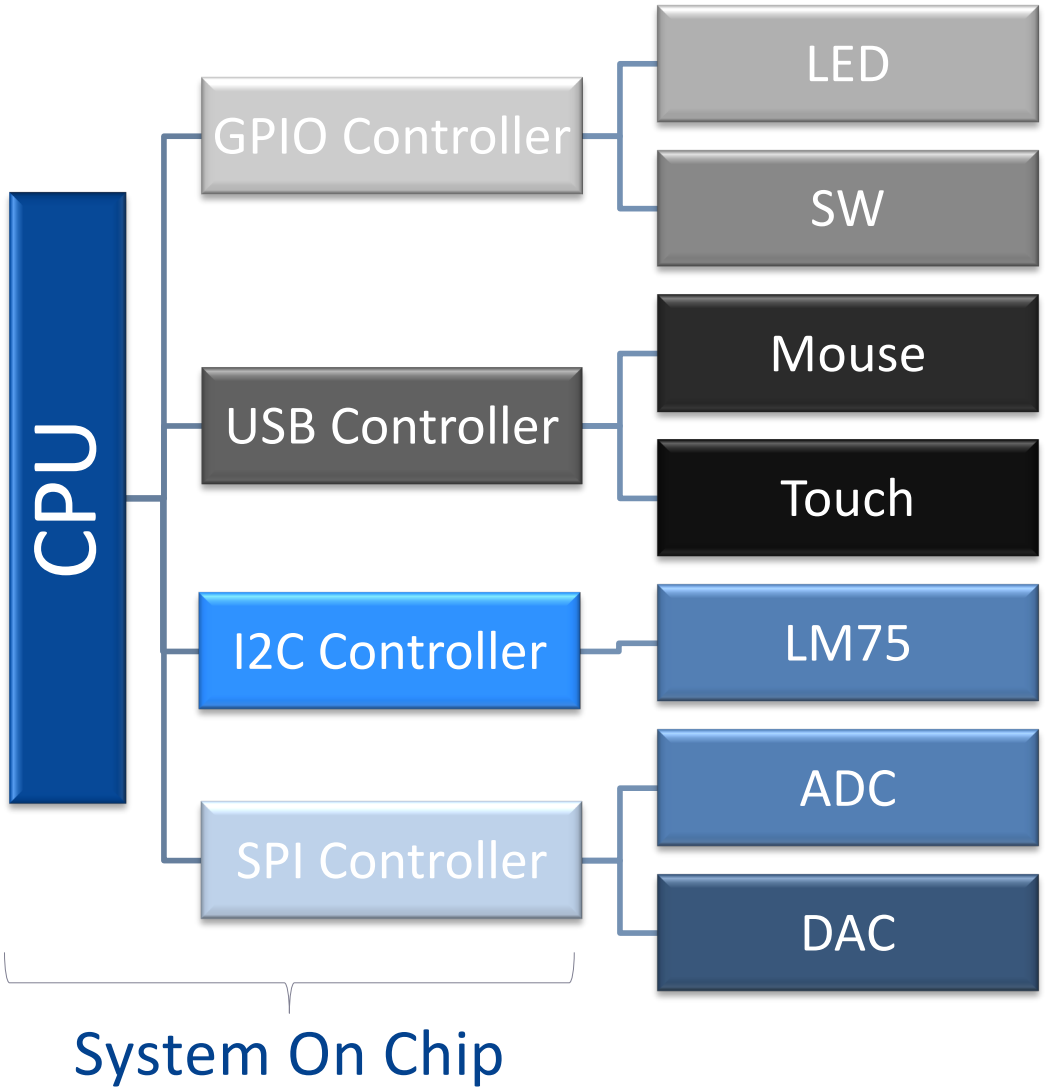


DEVICE TREE

- Describes hardware in the system:
  - Register addresses, interconnect, irq, pins etc.
  - Processor internals and external devices
- Structure resembles a Block Definition Diagram
- Kernel loads DT binary during boot and “hot-plugs” devices found
- Generic standard, defined by devicetree.org
- “Separates” Kernel code and hardware
- Much easier to make new variants and to maintain

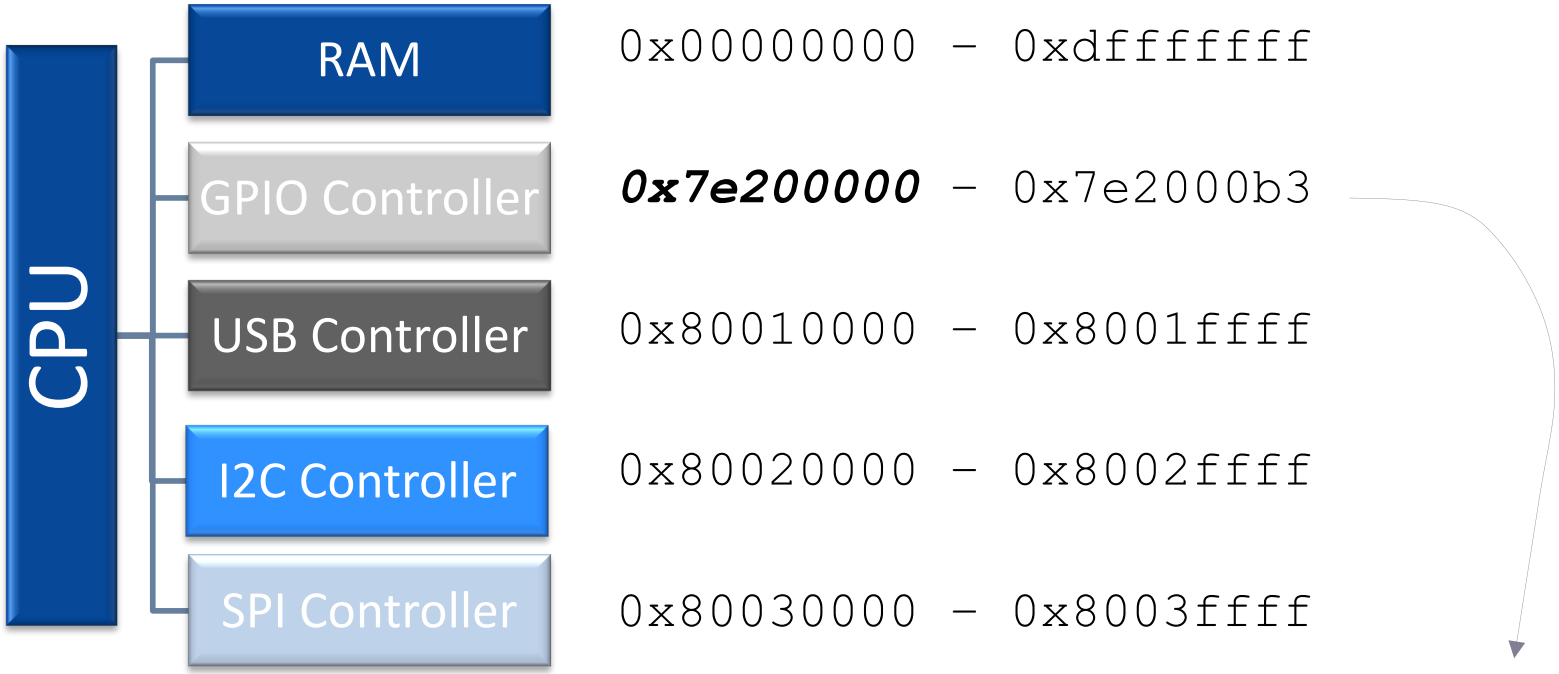
- Old days: HW definition part of kernel
- New variant -> New kernel to build + deploy + maintain ☹
- Already 140+ arm boards in kernel ☹

HARDWARE STRUCTURE



› Each Controller has a memory region assigned by the CPU's memory controller

MEMORY MAP



Register	Address	Data
gpio_datain	***0x7e200000***	0x12345678
gpio_dataout	0x7e200004	0xcafebabe
gpio_dir	0x7e200008	0xdeadbeef

BASIC DEVICE TREE

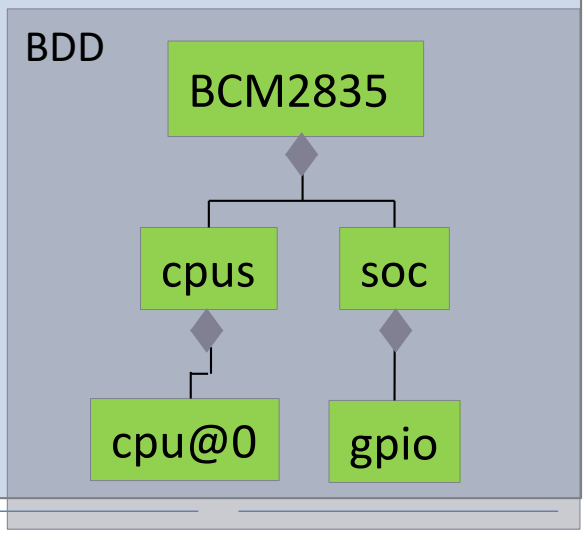
```
/dts-v1/;
/ {
    model = "BCM2835";
    compatible = "brcm,bcm2835";

    chosen {
      bootargs = "earlyprintk console=ttyAMA0"; };

    cpus {
      cpu@0 {
        device_type = "cpu";
        compatible = "arm,arm1176jzf-s";
        reg = <0x0>;
      };
    };

    soc {
      gpio: gpio@7e200000 {
        compatible = "brcm,bcm2835-gpio";
        reg = <0x7e200000 0xb4>;
```

Device name and name of compatible devices

Boot arguments for the Linux kernel

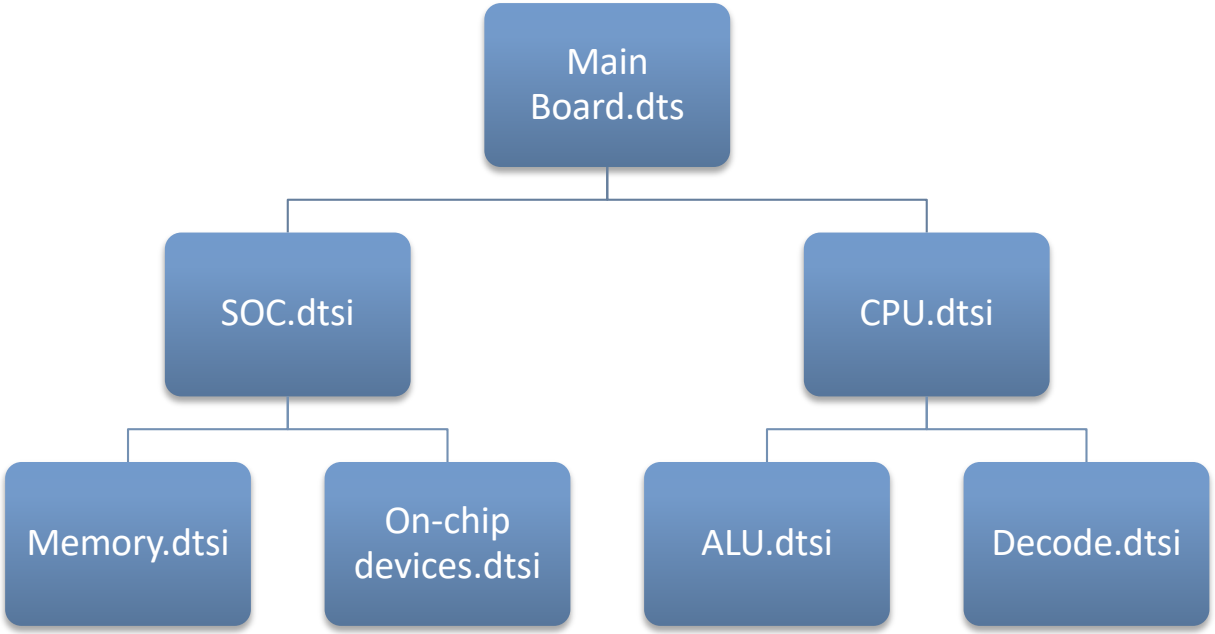
Definition of CPU(s), here one arm1176 cpu

Soc contains all non-cpu devices in the chip, here the gpio controller



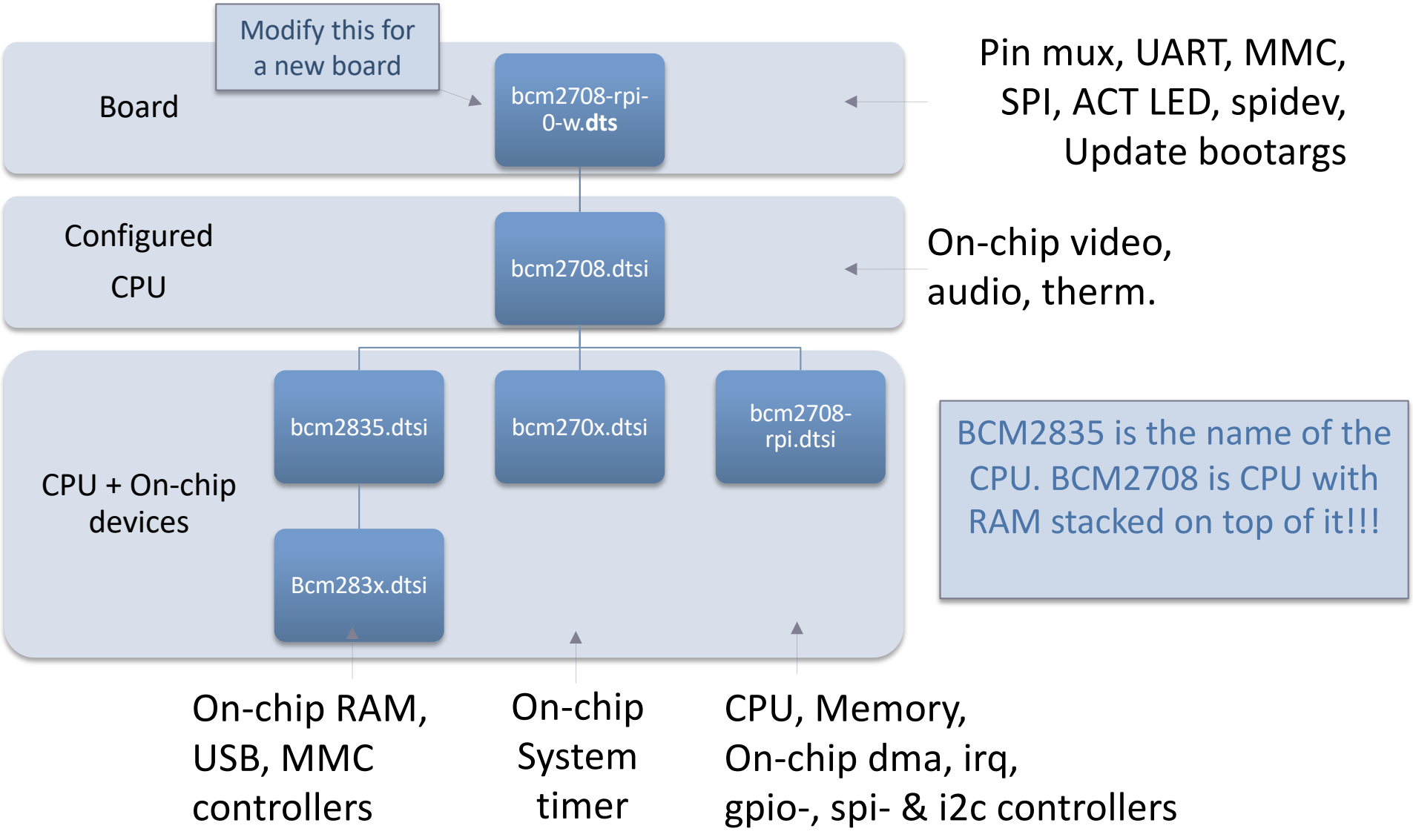
DEVICE TREE FILE STRUCTURE

› Device trees files are structured like a Block Definition Diagram (BDD)



› Higher level files includes and extends (or changes) information given in lower-level files

RPI ZERO W DEVTREE FILES



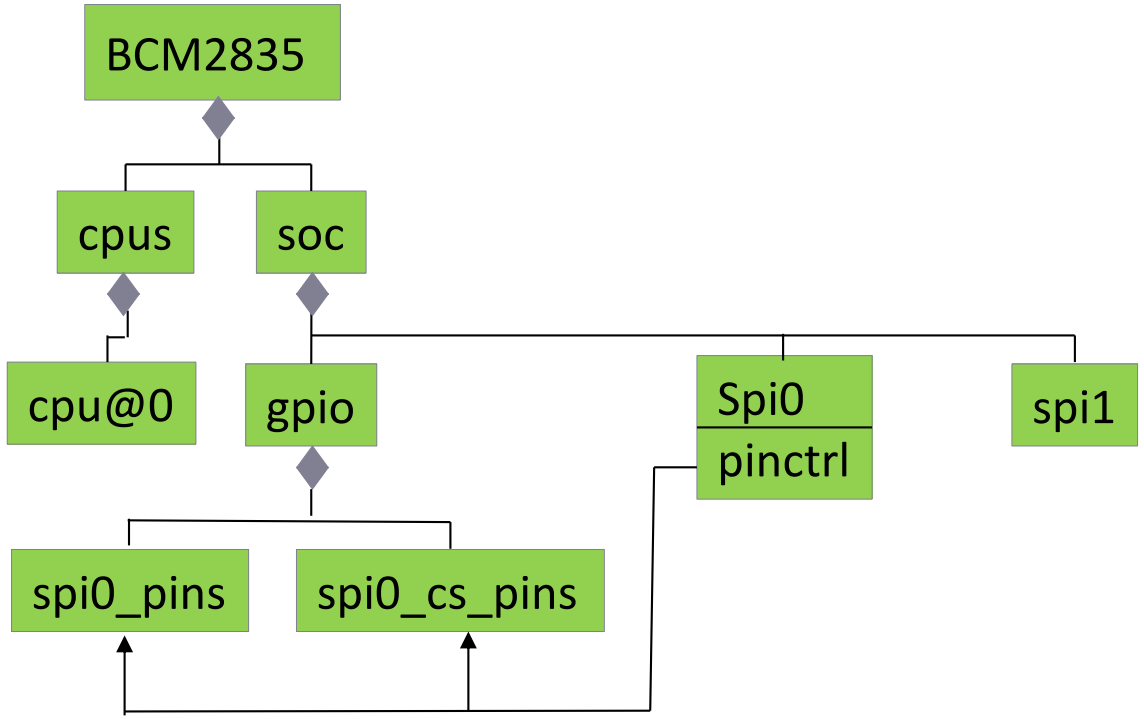
GPIO CONTROLLER SPEC.

```
/* bcm283x.dtsi */
/ {
  soc { /* On-chip devices */
        compatible = "simple-bus"; /* Memory mapped devices */
        #address-cells = <1>;    /* 1x 32-bit address space */
        #size-cells = <1>;         /* Addr-specifier length */

        gpio: gpio@7e200000 {/* Devtree label:Instance name */
          compatible = "brcm,bcm2835-gpio";       /* driver */
          reg = <0x7e200000 0xb4>;               /* addr len */
          interrupts = <2 17>, <2 18>, <2 19>, <2 20>;

          gpio-controller;
          #gpio-cells = <2>;       /* gpio-specifier length */

          interrupt-controller;
          #interrupt-cells = <2>;   /* irq-specifier length */
        };
};
```

GPIO USAGE FOR SPI

```
/* bcm2708-rpi-0-w.dts */

&gpio {                                  /* Append to "gpio" section */
  spi0_pins: spi0_pins {                        /* New sub-component */
    brcm,pins = <9 10 11>;                      /* Pins used by spi0 */
    brcm,function = <4>;          /* Pinmux function alt0 (=spi) */
  };

  spi0_cs_pins: spi0_cs_pins {
    brcm,pins = <8 7>;                      /* CS pins used by spi0 */
    brcm,function = <1>;                       /* pin mode = output */
  };
};

&spi0 { /* Append to "spi0" section
  pinctrl-names = "default";
  pinctrl-0 = <&spi0_pins &spi0_cs_pins>;       /* set pinmux */
  cs-gpios = <&gpio 8 1>, <&gpio 7 1>;    /* set gpio no/dir */
}
```

BDD FOR RASPBERRY PI ZERO W SO FAR



OVERLAYS

- Device tree provides support for add-on boards
  - Overlays, hats, many different names
- Overlay extends/modifies the device tree
  - Full syntax is not supported, you cannot delete nodes fx.
- Overlays are built to stand-alone files .dtbo
  - Can be specified in /boot/config.txt (dtoverlay=mydevt)
  - -or loaded runtime with the *dtoverlay* program
- Overlays are simpler to use
  - Hook into existing sections and add/modify components

GPIO OVERLAY EXAMPLE

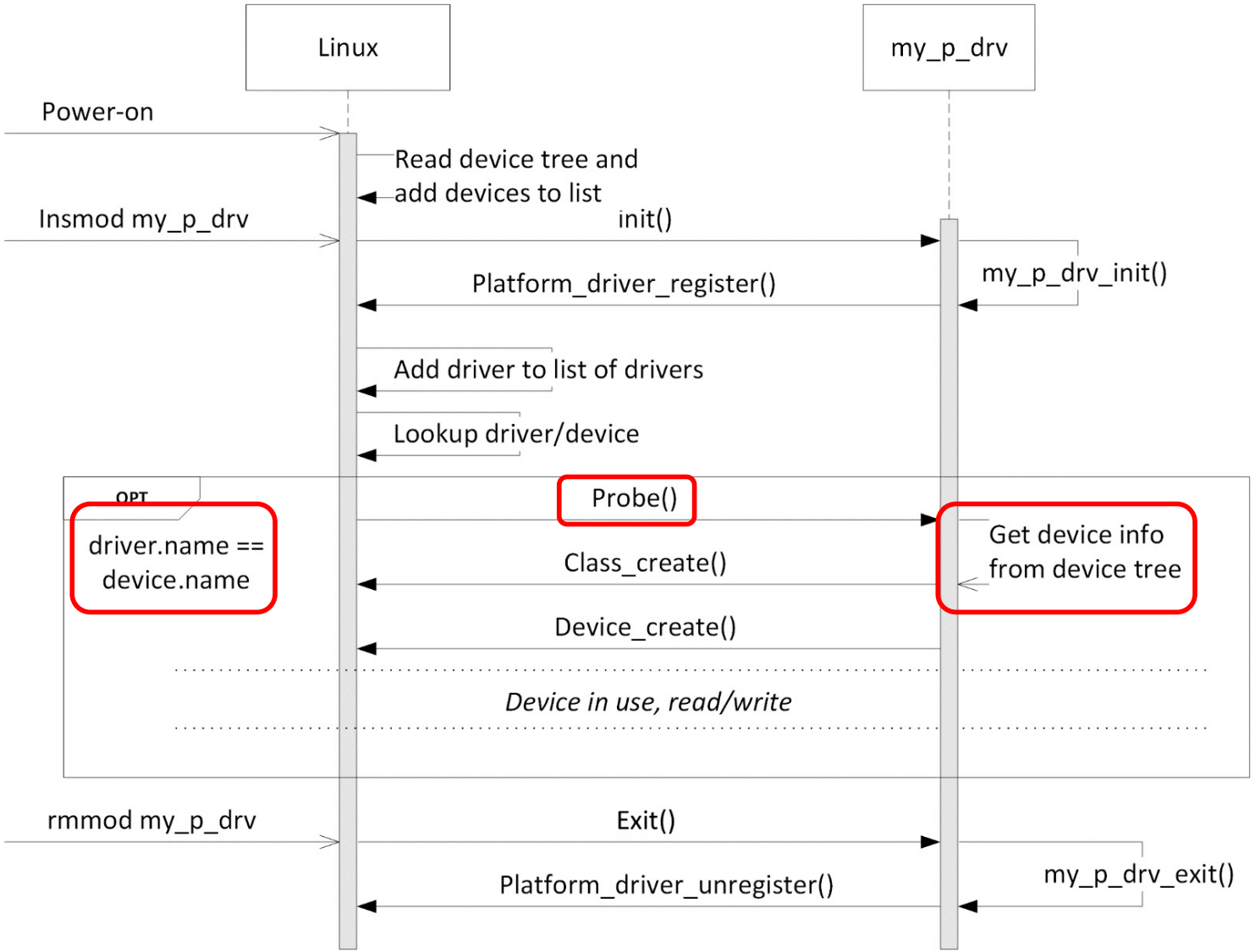
```
/dts-v1/;
/plugin/;
/ {
    compatible = "brcm,bcm2835", "brcm,bcm2708";   /* Rpizw */
    fragment@0 {
    target-path = "/";          /* New device put at base path */
    __overlay__ {
      mydevt: mydevt {          /* devtree label:instance name */
        compatible = "mydevt";  /* Label to match in driver */

        /* Configure gpio module */
        /* <ressource pinno dir> */
        gpios = <&gpio 55 0>, <&gpio 57 1>; /* 55|in 57|out */

        /* Custom Property */
        mydevt-custom = <0x12345678>;        /* 32-bit value */
        status = "okay";                  /* Device is enabled */
      };
    };
};
```

TOOLS / DEBUGGING

- To build a normal dts file, you invoke the dtb compiler:
- dtb –o my.dtb my.dts
- Output is a .dtb file

- To build an overlay, you need access to the dtb file, that you wish to extend
- In Linux, you can use the configured (and built) source tree.
- Dtb can be build in the source tree
- -or it can be built out-of-tree, like we do with kernel modules (See the Makefile provided for today's exercise)
- Output is a .dtb file, but must be renamed .dtbo for all tools to work ◔_◔

DEVICE/DRIVER BINDING



DT USAGE IN DRIVER

› Device Tree parameters are mapped to the specific device type: platform, spi, usb etc.

› The "*of*" set of functions give access to DT parameters

› #include <linux/of_gpio.h>
› #include <linux/of_address.h>
› #include <linux/of_device.h>
› #include <linux/of_irq.h>

› Probe function is called when during device/driver binding – This is where we will retrieve parameters!

READ DT PROPERTY IN DRIVER

```
static int mydevt_probe(struct platform_device *pdev)
{
  u32 gpio_num;

  printk(KERN_ALERT "New device: %s\n", pdev->name);

  /* Retrieve number of GPIOs */
  if ((gpios_len=of_gpio_count(pdev->dev.of_node))<0 ) {
        dev_err(&pdev->dev, "Failed to read of_gpio_count\n");
        return -EINVAL; }

  /* Retrieve GPIO numbers */
  gpio_num = of_get_gpio(pdev->dev.of_node, 0); /* Index 0 */
  if (gpio_num < 0) {
    if (gpios_num != -EPROBE_DEFER)
      dev_err(&pdev->dev,"Failed to parse io %d\n", gpio_num);
    }
}
```

OOD: STRUCTS, STRUCTS AND …

```c
// mydevt_driver.c
#include <of_gpio.h>
#include <platform_device.h>
static int mydevt_probe(struct platform_device *pdev){
  num_gpios = of_gpio_count(pdev->dev.of_node); }
```

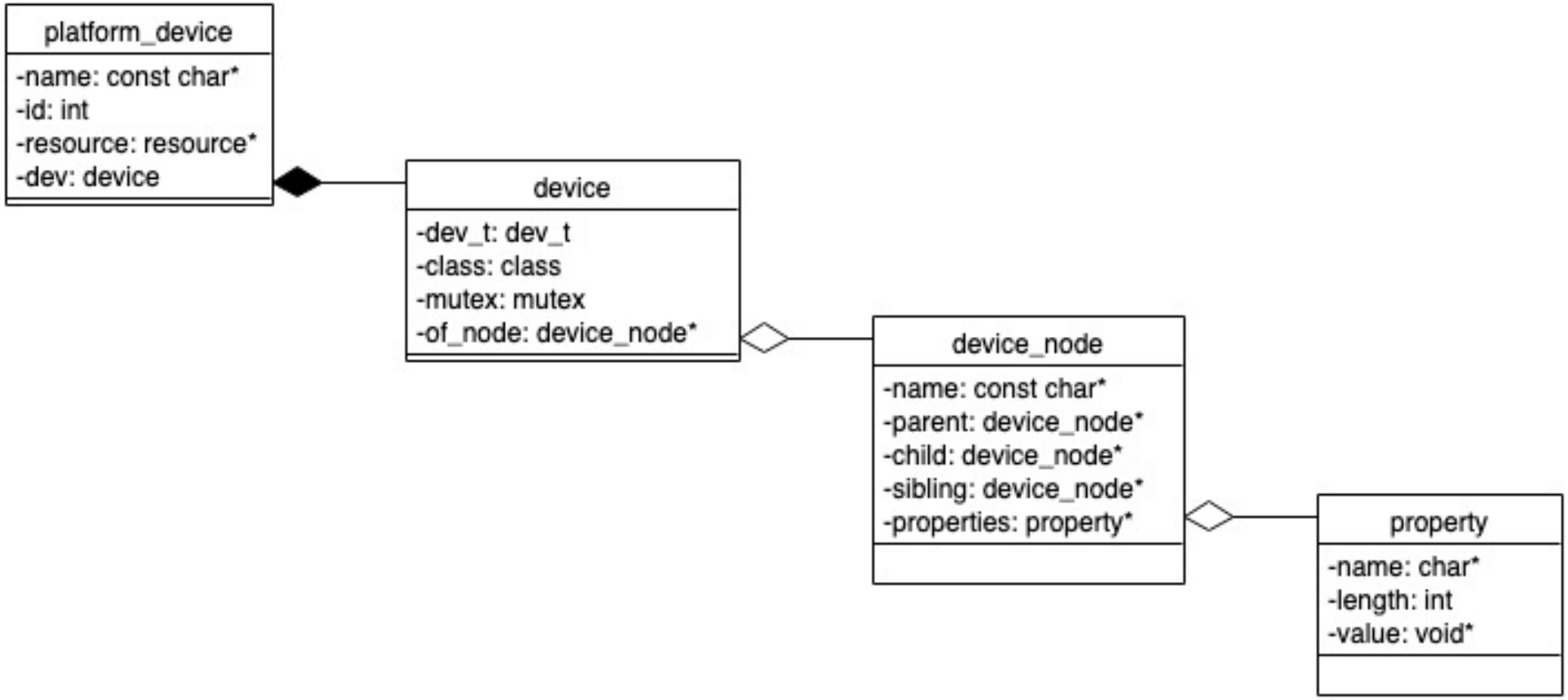
```c
// platform_device.h
#include <device.h>
struct platform_device {
        struct device dev; …
```

dev.of_node to return *device_node** required by of_gpio_count

```c
// device.h
struct device {
  struct device_node *of_node; /* associated DT node */
```

```c
// of_gpio.h
static inline int of_gpio_count(struct device_node *np);
```

OOD: IN UML


platform_device
-name: const char*
-id: int
-resource: resource*
-dev: device
device
-dev_t: dev_t
-class: class
-mutex: mutex
-of_node: device_node*
device_node
-name: const char*
-parent: device_node*
-child: device_node*
-sibling: device_node*
-properties: property*
property
-name: char*
-length: int
-value: void*

GET FLAGS…

```
// file: of_gpio.h

enum of_gpio_flags {
        OF_GPIO_ACTIVE_LOW = 0x1,
        OF_GPIO_SINGLE_ENDED = 0x2,
        OF_GPIO_OPEN_DRAIN = 0x4,
        OF_GPIO_SLEEP_MAY_LOOSE_VALUE = 0x8,

static int of_get_gpio_flags(struct device_node *np,
                             int index,
                             enum of_gpio_flags *flags)
```

of_gpio.h has default mapping, BCRM has different mapping:
0x0 = Input
0x1 = Output

```
// file: plat_drv-overlay.dts (Bcrm GPIO)
…
 gpios = <&gpio 55 0>, <&gpio 57 1>; /* 55|in 57|out */
```

Index 0 Index 1

(sourcetree/Documentation/devicetree/bindings/pinctrl/brcm,bcm2835-gpio.txt)

AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

11. NOVEMBER 2022
PETER HØGH MIKKELSEN

LINUX DEVICE MODEL AND BUSSES

LINUX DEVICE MODEL

› Dynamic node creation from user space
› Udev, Sysfs a.o.

› **Unified abstraction for drivers:**
› **Bus – Device – Driver (Todays subject!!!)**

› Device class- or instance configuration
› Class/Device Attributes

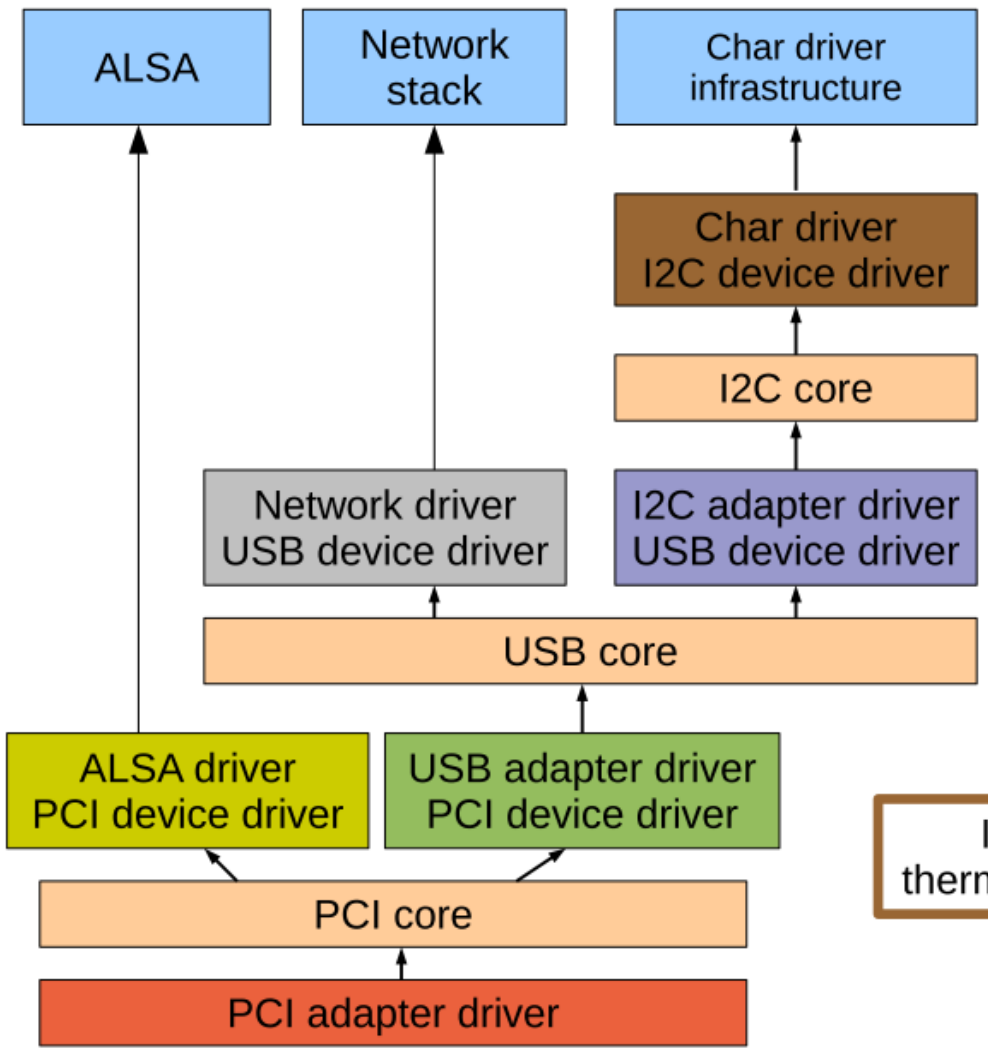
UNIFIED ABSTRACTION

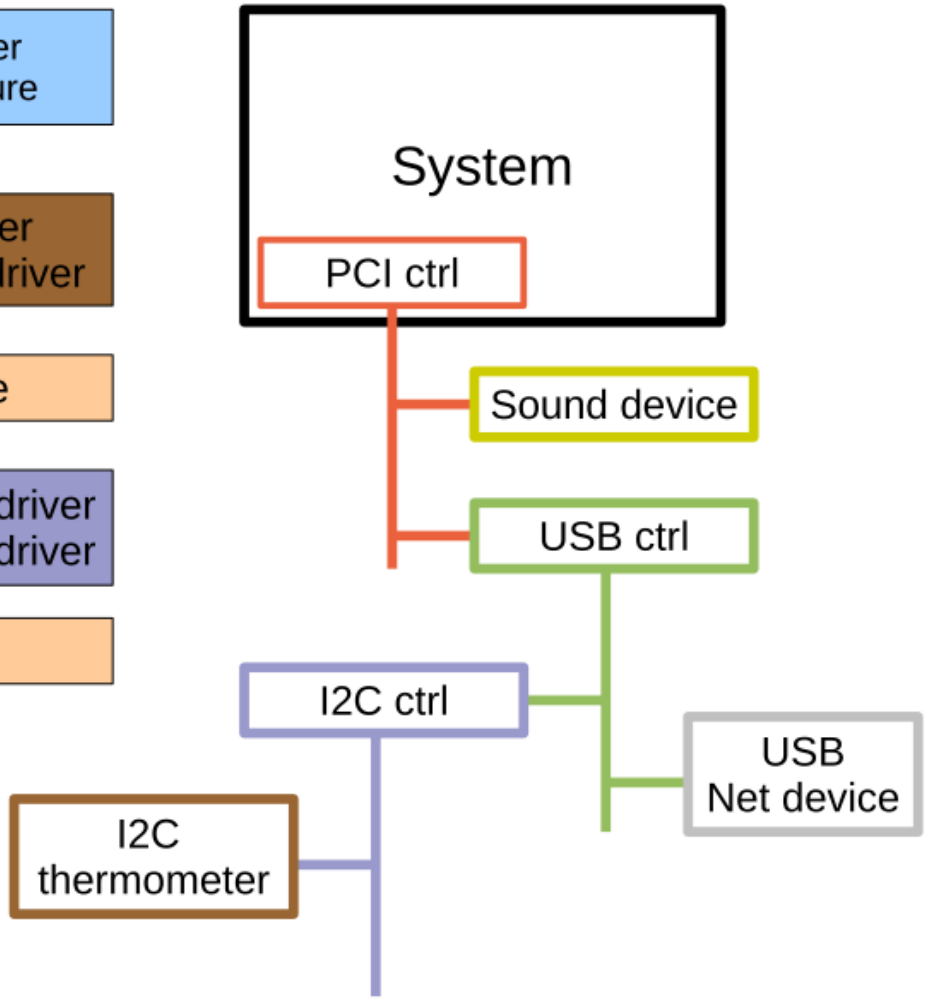
› Basic concepts:
› **Bus** – Channel between two devices
› **Device** – Physical device instance
› **Driver** – Driver that interacts with both

› **Algorithm Drivers** – Core bus drivers (I2C/SPI/.. core)
› /drivers/i2c/algos
› **Adapter Drivers** – Host side drivers
› /drivers/i2c/busses
› **Device Drivers** - Slave side drivers
› Located according to class ex /drivers/input/touchscreen/

(Locations mentioned above are relative to the kernel source directory, ex ~/sources/rpi-4.9/)

DRIVER HIERARCHY

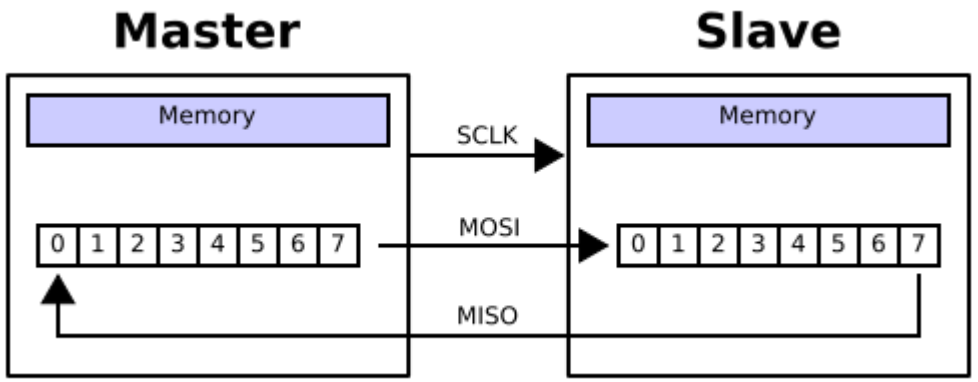




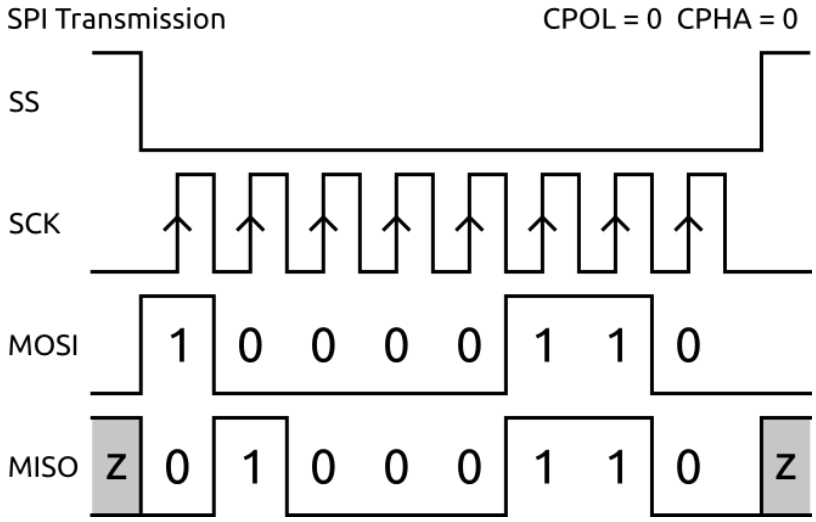
free-electrons.com

EXAMPLE: SPI BUS

- Four-wire bus system
- Typical use:
  - Serial Flash Memory
  - ADC/DAC
  - Touch Screen
  - Radio Controller I/F
- Max 70 Mbits/s

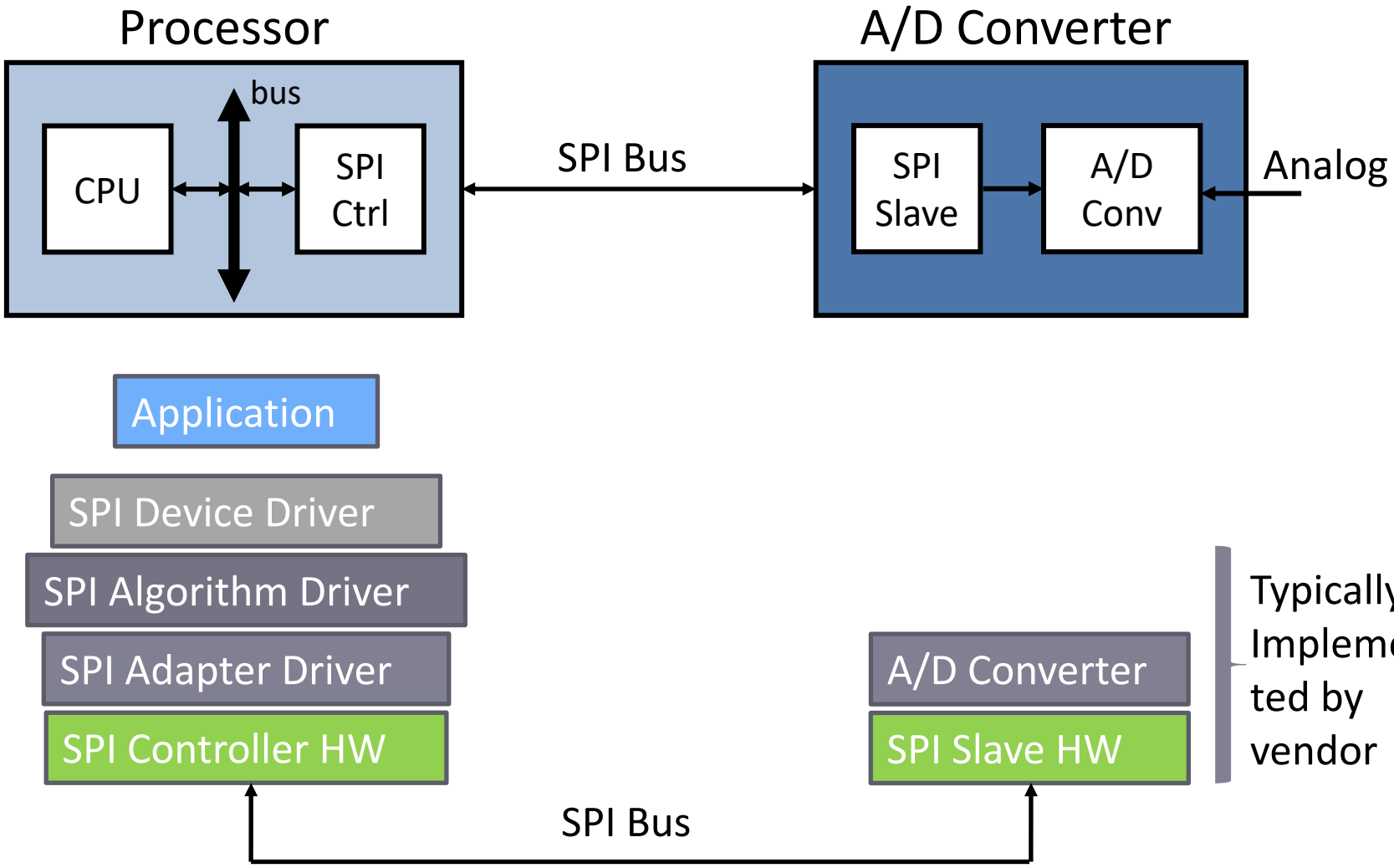


Wikipedia.org



Embeddedmicro.com

SPI ADC EXAMPLE

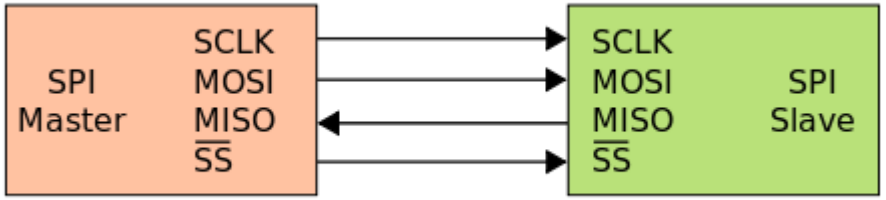


SPI DEVICE DRIVER

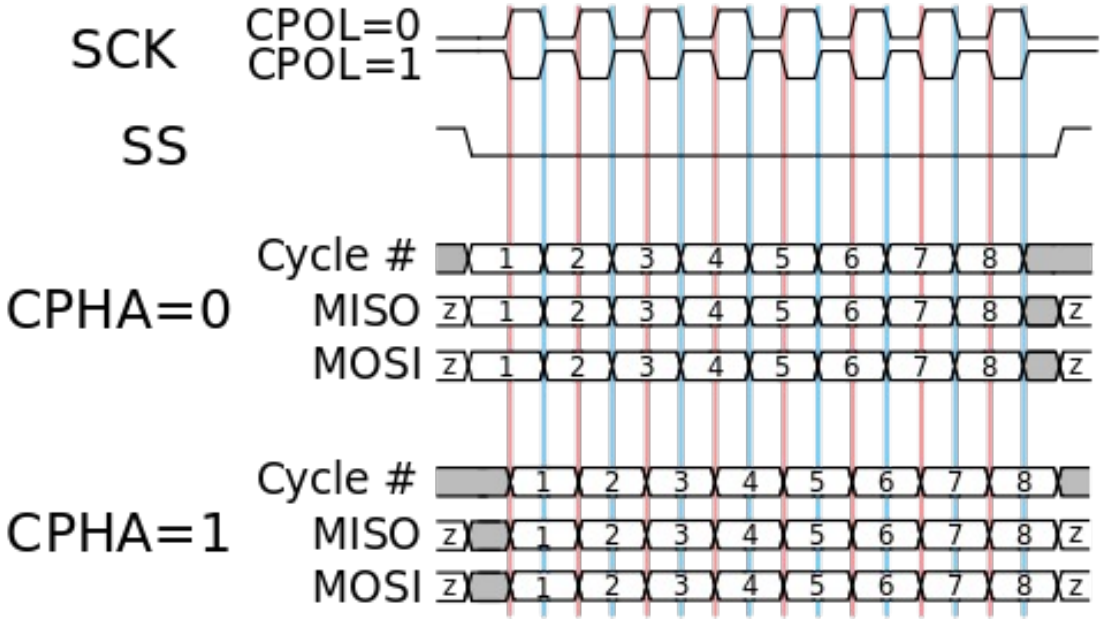
- SPI drivers are specialized platform drivers
- Uses spi.h instead og platform.h
- We must fill in bus type in driver struct
- SPI device information is given in device tree (DT)
- Probe provides a pointer to SPI device found in DT

```
static struct spi_driver spi_drv_spi_driver = {
  .probe = spi_drv_probe,
  .remove = spi_drv_remove,
  .driver = {
    .name = "my_spi_driver",
    .bus = &spi_bus_type,
    .of_match_table = of_spi_drv_device_match,
    .owner = THIS_MODULE, }, };
```

SPI DEVICE CONFIGURATION



wikipedia.org (spi)



wikipedia.org (spi)

› SPI bus number and chip select must be found in schematic
› Frequency and CPHA/CPOL in data sheet

SPI DEVICE TREE

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

/ {
    fragment@0 {
        target = <&spi0>; // SPI Bus 0
        __overlay__ {
            /* needed to avoid dtc warning */
            #address-cells = <1>; // reg has 1 parameter
            #size-cells = <0>;    // reg size = 32-bit
            my_spi_drv:my_spi_drv@0 {
                compatible = "ase, my_spi_drv";
                reg = <0>;       // SPI Chip Select 0
                // spi-cpha; // Comment in to set flag
                // spi-cpol; // Comment in to set flag
                spi-max-frequency = <100000>;
            };
        };
    };
};
```

SPI DRIVER REGISTRATION

```
static const struct of_device_id of_spi_drv_device_match[] = { {
.compatible = "ase, my_spi_drv", }, {}, };


static struct spi_driver spi_drv_spi_driver = {
  .probe = my_spi_drv_probe,
  .remove = my_spi_drv_remove,
  .driver = {
    .name = "my_spi_driver",
    .bus = &spi_bus_type,
    .of_match_table = of_spi_drv_device_match,
    .owner = THIS_MODULE, }, };


int my_spi_drv_init(void) {
  err = alloc_chrdev_region(…);
  cdev_init(…);
  …
  err = spi_register_driver(&spi_drv_spi_driver );
```

PROBE

Aka the method called when there is device + driver match

```
struct myspidev {
  struct spi_device *spi; // Pointer to SPI device
};
static struct myspidev myspidevs[12];
static int nbr_devs = 0;

static int ads7870_spi_probe(struct spi_device *spi){
 int err=0;
 pr_debug("New SPI device: %s at CS: %i\n",
        spi->modalias, spi->chip_select);

 /* Additional SPI configuration */
 spi->bits_per_word = 8;
 err = spi_setup(spi);

 myspidevs[nbr_devs].spi = spi; // Set global ptr to SPI device
 spi_drv_device = device_create(spi_drv_class, NULL,
                        MKDEV(MAJOR(devno), nbr_devs),
                        NULL, "myspidev%d", nbr_devs);
 ++nbr_devs;
return err;}
```

REMOVE

Aka the method called when device or driver is removed

```
static int ads7870_remove(struct spi_device *spi)
{

  printk (KERN_ALERT "Removing SPI device %s at CS: %i\n",
              spi->modalias, spi->chip_select);

  [For each device allocated in Probe]
    device_destroy(spi_drv_class, MKDEV(MAJOR(devno), its_minor));

  return 0;
}
```

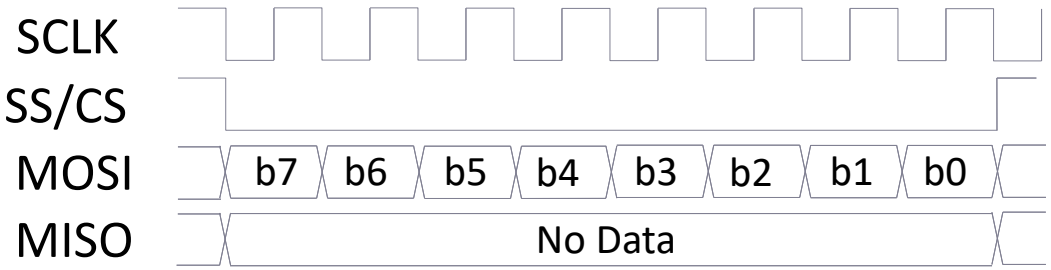
SPI COMMUNICATION

› SPI communication can be half-/full duplex
› SPI communication is specific to each slave device
› SPI controller is very flexible

› An SPI communication sequence is called a "*message*"
› A "*message*" is based on a list of "*transfers*"
› Each transfer can have RX (MISO) and TX (MOSI) data

› When a "message" is built, it can be exchanged with "spi_sync" or "spi_async" –*This will initiate physical communication (Blocking- or non-blocking)*

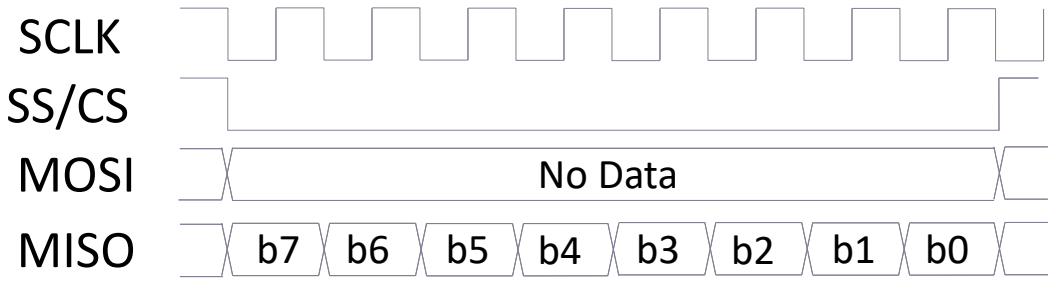
SPI 8-BIT WRITE



transfer

```c
int my_spi_write_byte(struct spi_device *spi, u8 data) {
  struct spi_transfer t[1];  /* Only one transfer */
  struct spi_message m;

  memset(t, 0, sizeof(t)); /* Init Memory */
  spi_message_init(&m);    /* Init Msg */
  m.spi = spi;             /* Use current SPI I/F */

  t[0].tx_buf = &data;     /* Transmit data */
  t[0].rx_buf = NULL;      /* Recieve No data */
  t[0].len = 1;            /* Transfer Size in Bytes */
  spi_message_add_tail(&t[0], &m); /* Add Msg to queue */

  err = spi_sync(m.spi, &m); /* Blocking Transmit */
  return err; }
```

SPI 8-BIT READ




```
int my_spi_read_byte(struct spi_device *spi, u8 *data) {
  struct spi_transfer t[1];  /* Only one transfer */
  struct spi_message m;

  memset(t, 0, sizeof(t)); /* Init Memory */
  spi_message_init(&m);    /* Init Msg */
  m.spi = spi;             /* Use current SPI I/F */

  t[0].tx_buf = NULL;      /* Transmit NO data */
  t[0].rx_buf = data;      /* Recieve data */
  t[0].len = 1;            /* Transfer Size in Bytes */
  spi_message_add_tail(&t[0], &m); /* Add Msg to queue */

  err = spi_sync(m.spi, &m); /* Blocking Transmit */
  return err; }              /* rx is returned by ref in data */
```

ADS7870 8-BIT READ

```
int ads7870_spi_read_reg8(struct spi_device *spi, u8 adr, u8 *data) {
  struct spi_transfer t[2];
  struct spi_message m;
  u8 cmd;

  cmd = (1<<6)|(adr & 0x1f);
  memset(t, 0, sizeof(t));
  spi_message_init(&m);
  m.spi = spi;

  t[0].tx_buf = &cmd;
  t[0].rx_buf = NULL;
  t[0].len = 1;
  spi_message_add_tail(&t[0], &m);

  t[1].tx_buf = NULL;
  t[1].rx_buf = data;
  t[1].len = 1;
  spi_message_add_tail(&t[1], &m);
  spi_sync(m.spi, &m);
  return 0; }
```

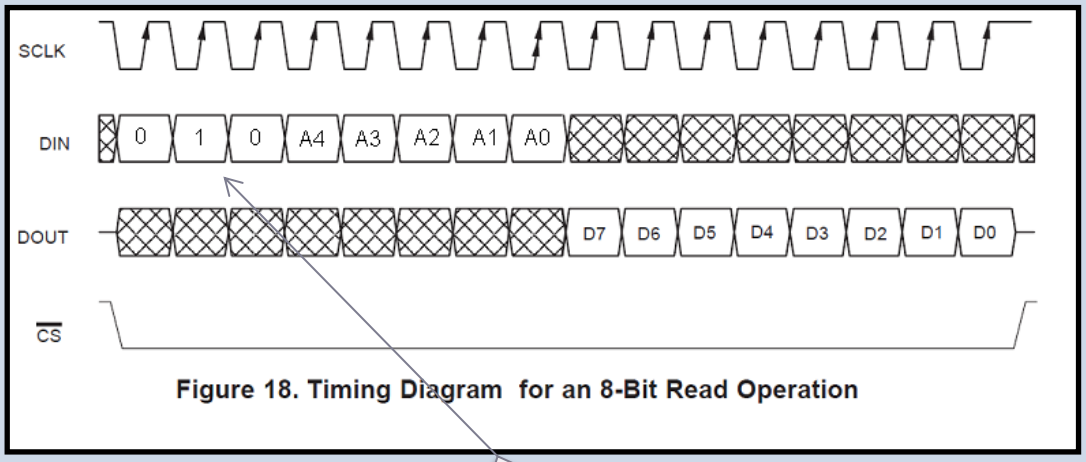


Figure 18. Timing Diagram for an 8-Bit Read Operation

010 to request slave to put data in its MISO buffer (Half duplex)

LINKED LIST FOR DEVICES 1 (OPTIONAL)

```
struct myspidev {
 struct spi_device *spi; // SPI dev ptr
 int ch; //
 dev_t devno; // Device Number (Major+Minor)
 struct list_head device_entry;
};
LIST_HEAD(myspidevs) // Define new list

static int my_spi_probe(struct spi_device *spi){
…
 myspidev *itsdev = kzalloc(sizeof(*myspidev), GFP_KERNEL);
 itsdev->devno = MKDEV(MAJOR(devno), its_minor);
 itsdev->spi = spi;
 INIT_LIST_HEAD(&itsdev->device_entry)
 spi_drv_device = device_create(spi_drv_class, &spi->dev,
                        itsdev->devno, itsdev, "myspidev%d.%d",
                         spi->master->bus_num, spi->chip_select);
 list_add(&itsdev->device_entry, &myspidevs);
return err;
}
```

LINKED LIST FOR DEVICES 2 (OPTIONAL)

```
static int spidev_open(struct inode *inode, struct file *filp) {
 struct myspidev *itsdev;

 // Traverse list to find device with same major/minor (devno)
 list_for_each_entry(itsdev, &myspidevs, device_entry) {
  if (itsdev->devno == inode->i_rdev) {
   // Store device ptr in file ptr's private data
   filp->private_data = itsdev; }
 }
}
```

```
static ssize_t spidev_read(struct file *filp, char __user *buf,
size_t count, loff_t *f_pos) {
 struct myspidev *its_spi_dev;
 uint8 data = 0;
 // Retrieve device from file node's private data
 its_spi_dev = filp->private_data;
 // Read from spi device with appropriate spi_device ptr.
 ads7870_spi_read_reg8(its_spi_dev->spi, its_spi_dev->ch, &data);
```

Check spidev.c in source tree for cleanup

AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

22. NOVEMBER 2019
PETER HØGH MIKKELSEN

LINUX DEVICE MODEL AND ATTRIBUTES

LINUX DEVICE MODEL

- Dynamic node creation from user space
- Udev, Sysfs a.o.

- Unified abstraction for drivers:
- Bus – Device – Driver

- **Device class- or instance configuration**
- **Class/Device Attributes (Todays subject!!!)**

DEVICE DRIVER CONFIGURATION

- Can we make device drivers configurable?
- Can we set the baud rate for a serial port?
- ..then we'll have to change some hardware parameters…
- …but how to pass them to a device driver at run-time?

- Old drivers:
- **IOCTL** system calls with predefined numbers to manipulate settings in drivers
- I2c-dev: `ioctl(fd, 0x0703, 0x48); // Sys call sub-id: 0x0703`

- New drivers:
- **Attributes** (Object oriented terminology)
- Class Attributes (like C++ static member functions)
- Device Attributes (like C++ member functions)
- I2c-dev: `echo lm75 0x48 > /sys/class/i2c-adapter/i2c-1/new_device`

IOCTL

```
ssize_t i2cdev_ioctl(struct inode *Inode, struct file *filep,
                     unsigned int cmd, unsigned long arg)
{
    switch(cmd)
    {
       case 0x0703:
         // Set I2C Address
         i2c_address = arg;
         break;
       default:
          printk(KERN_ALERT "Unknown ioctl-command: %d \n", cmd);
    }
    return 0;
}
```

```
struct file_operations i2cdev_Fops =
{
    .owner    = THIS_MODULE,
    .write    = i2cdev_write,
    .read     = i2cdev_read,
    .unlocked_ioctl = i2cdev_ioctl,
};
```

```
› ioctl(fd, 0x0703, 0x48); // User Space Application
```

ATTRIBUTES

› The Device Model supports attributes for Class or Device configuration:

› Class Attributes (~ C++ static member function) :
› Ex: echo 23 > /sys/class/gpio/export

› Device attributes (~ C++ member function) :
› Ex: echo 1 > /sys/class/gpio/gpiochip0/value

› Attributes have associated setter (store) and getter (show) methods

STORE ATTRIBUTE METHOD

```
static ssize_t my_state_store(struct device *dev,
        struct device_attribute *attr,
        const char *buf, size_t size){

 u8 value;
 /* Retrieve drvdata ptr (IF set in device_create) */
 struct my_dev *d = dev_get_drvdata(dev);

 int err=kstrtou8(buf, 0, &value); // Aut detect base
 if (err<0) {
  printk(KERN_ALERT "Unable to parse string\n");
  return err; }

 printk("Store called with %i\n", value);

 d->value = value; // Set value in drvdata IF used

return size;}
```

```
struct my_dev_t {
  int value;
} my_dev;
```

Copy to-/from user not needed!

See more string conversion functions in kernel.h (kstrto…)

› Return value has function similar to fops read/write

SHOW ATTRIBUTE METHOD

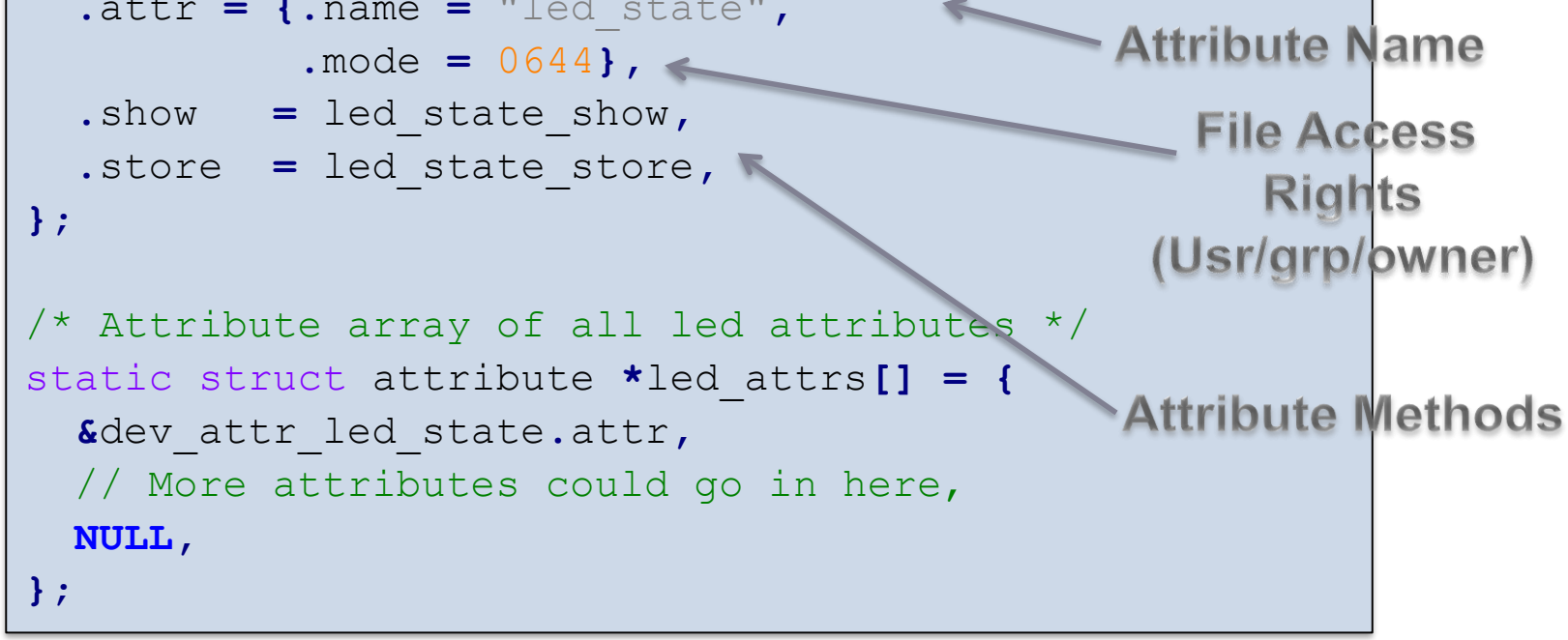
```
static ssize_t my_state_show(struct device *dev,
        struct device_attribute *attr, char *buf){

 /* Retrieve drvdata ptr (set in device_create) */
 struct my_dev *d = dev_get_drvdata(dev);

 int value = 0xdeadbeef;

 int len = sprintf(buf, "%d\n", value);

 return len;
}
```

Copy to-/from user not needed!

› Return value has function similar to fops read/write

ASSIGN ATTRIBUTE TO DEVICE (1:2)

```
/* Linux 4.0 ff. */
/* Struct to describe attribute name */
/* access rights and functions */
static struct device_attribute dev_attr_led_state = {
  .attr = {.name = "led_state",
           .mode = 0644},
  .show   = led_state_show,
  .store  = led_state_store,
};

/* Attribute array of all led attributes */
static struct attribute *led_attrs[] = {
  &dev_attr_led_state.attr,
  // More attributes could go in here,
  NULL,
};
```

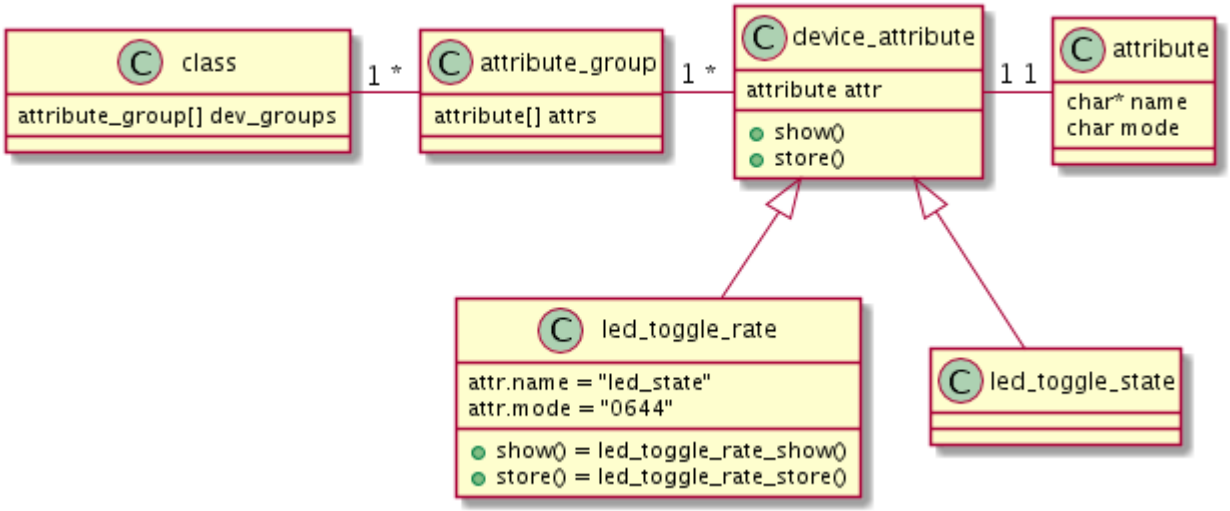


ASSIGN ATTRIBUTE TO DEVICE (2:2)

```
/* Linux 4.0 ff. */
/* led group containing all led attributes */
static const struct attribute_group led_group = {
  /* if .name is present it creates a group */
  /*  folder under device in sys tree */
  //  .name = "led_group",
  .attrs = led_attrs,
};

/* Device can have many groups */
static const struct attribute_group *led_groups[] = {
  &led_group,
  NULL,
};
```

› Can we draw a class diagram of all this..?

ATTRIBUTES – CLASS DIAGRAM



…USING MACROS

```
/* Assuming Names:                  */
/* group name = led                 */
/* attribute name = led_state       */
/* RW/RO/WO = RW (led_state_store() + led_state_show()) */

DEVICE_ATTR_RW(led_state); // Creates dev_attr_led_state

static struct attribute *led_attrs[] = {
    &dev_attr_led_state.attr,
    // More attributes could go in here,
    NULL,
    };

ATTRIBUTE_GROUPS(led); // Creates led_groups, expects led_attrs
```

› Macros creates variables and inits structures
› Be careful about naming!!!

What Macro?!? See https://en.wikipedia.org/wiki/C_preprocessor

DEFAULT DEVICE ATTRIBUTES

› Attributes can be assigned to be default for new devices of a class:

```
ATTRIBUTE_GROUPS(led); // Creates led_groups

static int mygpio_init(void)
{
 alloc_chrdev_region(&devno, 0, my_num_leds, "my_leds");
 my_led_class = class_create(THIS_MODULE, "my_leds");
 my_led_class->dev_groups = led_groups;
…
static int mygpio_probe(struct platform_device *pdev){
 my_gpio_device_101 = device_create(my_led_class, NULL,
    MKDEV(MAJOR(devno), my_minor), &drvdata, "my_gpio%d", 101);
```

All new devices will have attributes

› SysFS files will be created automatically
› Must be done before subsequent calls to device_create()
› Attributes can be accessed directly in file system:

```
# echo on > /sys/class/my_leds/my_led101/led_state
```

AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

22. OCTOBER 2019
PETER HØGH MIKKELSEN

LINUX TIMERS AND DEFERRED WORK

DEFERRING WORK

- The Linux kernel provides timers and means for deferring work
- The concept of Jiffies
- General vs processor specific timers
- Wait functions
- Delay functions (busy waiting)
- Sleeping (sleeping)
- Timers
- Timers with timer callbacks

JIFFIES

› Linux uses a system timer to keep track of time
› Timer resolution ranges 50-1000 ticks / sec

› Jiffies is a 64-bit timer tick counter
› It is reset at power-up
› Flips over after 2^64 timer ticks
› Your responsibility to handle overflow!
› Can be used to measure relative time

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

stamp_1 = jiffies;      /* Current Value */
stamp_2 = stamp_1 + HZ /* +1 second (HZ=jiffies/sec)    */
```

PROCESSOR SPECIFIC TIMERS

- To achieve higher timer resolution you may use fast low-level processor specific (PS) timer functions
  - May rely on CPU clock conuter
  - May rely on a specific CPU timer
  - Is hardware architecture dependent
- Is NOT 100% portable across architectures
- Example: sched_clock()

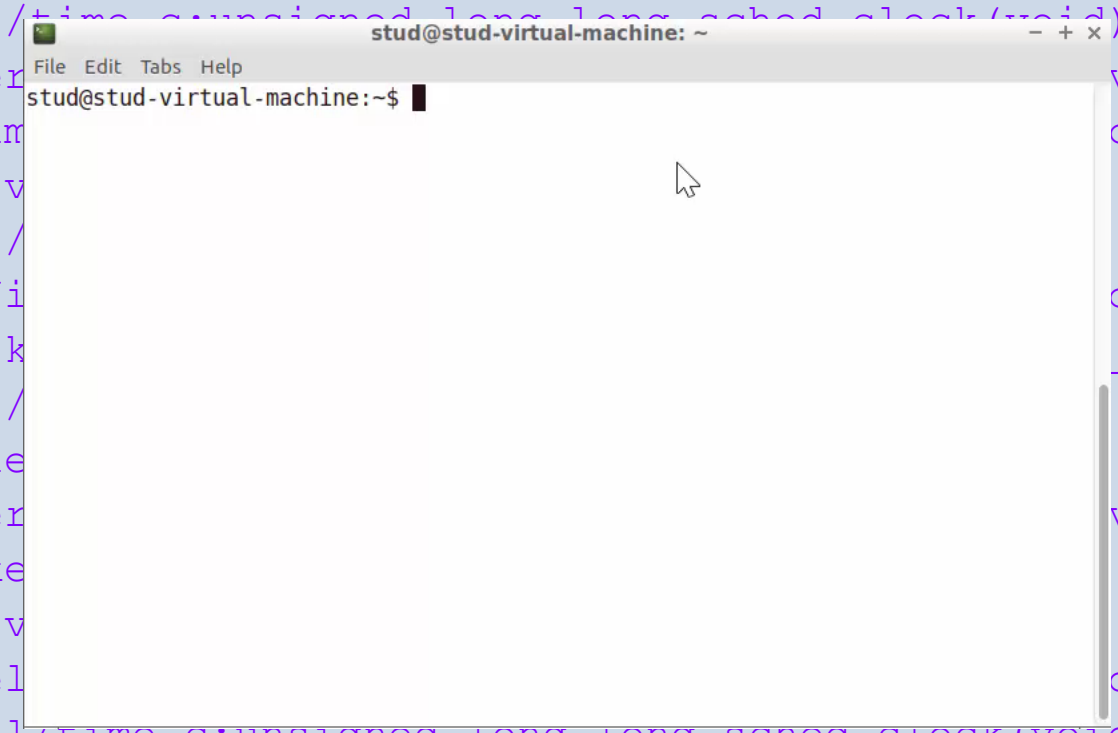
PROCESSOR SPECIFIC TIMING (2)

```
// source/kernel/sched/clock.c
/*
 * Scheduler clock - returns current time in nanosec units.
 * This is default implementation.
 * Architectures and sub-architectures can override this.
 */
unsigned long long __weak sched_clock(void)
{
    return (unsigned long long)(jiffies - INITIAL_JIFFIES)
                    * (NSEC_PER_SEC / HZ);
}
```

- Note the use of default implementations, possibly overridden by architecture specific versions
- What resolution has the default implementation?

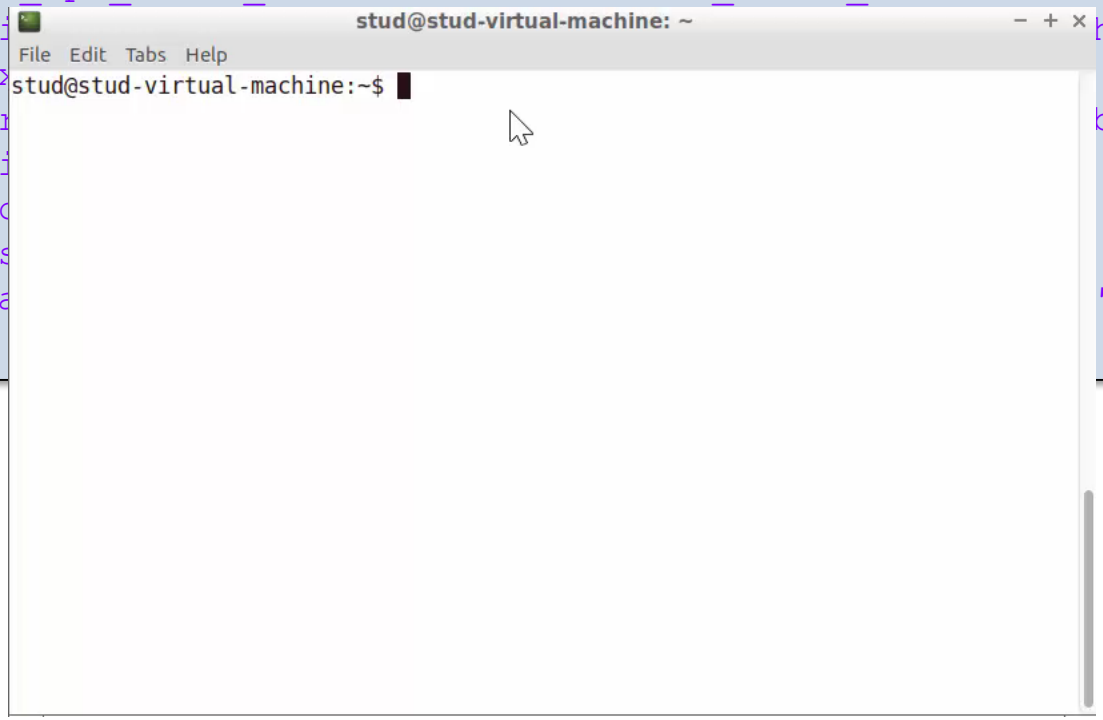
SCHED_CLOCK FOR ARM?

```
stud@stud-virtual-machine:~$ find /sources/rpi-4.9/arch -name
"*.c" | xargs grep " sched_clock(void)"
./frv/kernel/time.c:unsigned long long sched_clock(void)
./powerpc/ker                                          void)
./mips/cavium                                          otrace
sched_clock(v
./c6x/kernel/
./m68k/coldfi                                          ock(void)
./unicore32/k                                          _clock(void)
./x86/kernel/
./sparc/kerne                                          (void)
./mn10300/ker                                          void)
./blackfin/ke
sched_clock(v
./cris/kernel                                          d)
./tile/kernel/time.c:unsigned long long sched_clock(void)
./s390/kernel/time.c:unsigned long long notrace sched_clock(void)
stud@stud-virtual-machine:~/sources/rpi-4.9/arch$
```



SCHED_CLOCK USAGE?

```
stud@stud-virtual-machine:~$ find sources/rpi-4.9/drivers -name "*.c" |
xargs grep "sched_clock()" -ml
./clocksource/dw_apb_timer_of.c:            init_sched_clock();
./clocksource/ti                                                hed_clock(),
./clocksource/px                                                 always be
./clocksource/ar                                                bed and the
./net/irda/pxafi
./perf/arm_pmu.c
./acpi/apei/ghes
stud@stud-virtua                                                "*.c" | wc -l
13277
```



› Only referenced in 3 of 13277 driver .c filers!

CURRENT TIME

› The absolute calendar time can be retrieved by:

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

void do_gettimeofday(struct timeval *tv);
```

› But do you want to do that?

DELAYS (1)

› Busy Waiting

```
for (i=0;i<10000;i++);
```

› How much work do we wish to do while waiting? Is it accomplished?
› Is the delay deterministic?

› Scheduling the process

```
while(time_before(jiffies, jiffies_no_to_timeout_at))
   schedule();
```

› How much work is done during this delay?
› Is the delay deterministic?

DELAYS (2)

› From interrupt handling we recall the function:

```
wait_event_interruptible()
```

› It may include a timeout:

```
wait_event_interruptible_timeout(waitqueue, condition,
                                 timeout_in_jiffies)
```

› How much work is done during this delay?
› Is the delay deterministic?

› If only the delay is needed we may use:

```
#include <linux/sched.h>
signed long schedule_timeout(timeout_in_jiffies)
```

SHORT DELAYS

› Busy Waiting Short Delays

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

› Implemented using low-level software loops
› Precise

› Sleeping Delays

```
#include <linux/delay.h>
void msleep(unsigned int msecs);
void msleep_interruptible(unsigned int msecs);
void ssleep(unsigned int secs);
```

› Puts the calling process to sleep (_int can be woken up by an interrupt)
› Use if you can allow delay to be longer than requested

KERNEL TIMERS

- Use a timer if you need to schedule a process for execution at a later time.

- Kernel Timers are based on software interrupts
- Timer interrupts are asynchronous in nature, and may lead to race conditions

- Implementation resembles that of an interrupt
- Request
- Service Routine

- Timer functions are run in interrupt context!!
- No sleeping! No memory allocation! No user space access!

ONE-SHOT TIMER

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

struct timer_list my_timer;
int my_gpio = 176; // Global variable
int my_delay_ms = 1000;

static void my_timer_callback(struct timer_list *t){
  gpio_set_value(176); // Do something once
}

static int my_probe(struct platform_device *pdev)
{
 timer_setup(&my_timer, my_timer_callback, 0);
 mod_timer(&my_timer, jiffies + msecs_to_jiffies(my_delay_ms));
 …
}
```

PERIODIC TIMER

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

struct timer_list my_timer;
int my_gpio = 176; // Global variable
int my_delay_ms = 1000;

static void my_timer_callback(struct timer_list *t){
 mod_timer(&my_timer, jiffies + msecs_to_jiffies(my_delay_ms));
 gpio_set_value(176);
}

static int my_probe(struct platform_device *pdev)
{
 timer_setup(&my_timer, my_timer_callback, 0);
 mod_timer(&my_timer, jiffies + msecs_to_jiffies(my_delay_ms));
 …
}
```

USING POINTERS AND STRUCTS

```
#include <linux/timer.h>
struct my_dev {
 int gpio;
 int delay;
 struct timer_list ttimer;
}
struct my_dev my_dev_inst;;

static void my_timer_callback(struct timer_list *t){
 struct my_dev *myd = from_timer(myd, t, ttimer);
 mod_timer(&myd->ttimer, jiffies + msecs_to_jiffies(myd->delay));
 gpio_set_value(myd->gpio);
}

static int my_probe(struct platform_device *pdev)
{
 timer_setup(&my_dev_inst.my_timer, my_timer_callback, 0);
 mod_timer(&my_dev_inst.my_timer,
           jiffies + msecs_to_jiffies(my_delay_ms));
 …
}
```

from_timer() uses container_of() to calculate my_dev instance address

"mod_timer" could also be set in an attribute function

CONTAINER_OF

If you have a reference to an object which is part of a struct, container_of(), can return a reference to the containing struct. The address calculation is done at compile-time using macro functions.

my_drv.c

```
struct my_dev {
 int gpio;
 int delay;
 struct timer_list ttimer;
}
static void my_timer_callback(struct timer_list *t){
 struct my_dev *myd = from_timer(myd, t, ttimer);
```

```
//linux/timer.h
#define from_timer(var, callback_timer, timer_fieldname) \
        container_of(callback_timer, typeof(*var), timer_fieldname)
```

```
//In effect…
struct my_dev *myd = container_of(t, struct my_dev *, ttimer)
```

```
//linux/kernel.h
#define container_of(ptr, type, member) ({                  \
 void *__mptr = (void *)(ptr);                              \
 ((type *)(__mptr - offsetof(type, member))); })
```

```
//In effect…
 void *__mptr = (void *)(t);                                \
 struct my_dev *myd = ((struct gt_dev*)(__mptr -
                       ((struct gt_dev*)0)->ttimer)); })
```

CONTAINER_OF…

Example struct & values!

struct my_dev

Type	Name	Address offset
int	gpio	0x0000
int	delay	0x0004
struct timer_list	ttimer	0x0008
int	tbd	0x0024

`struct my_dev *myd` →

`struct timer_list *t` →

Nullptr to calc offset

`(struct my_dev*)0`

`->ttimer`

```
void *__mptr = (void *)(t);
struct gt_dev *myd = ((struct my_dev*)(__mptr - ((size_t)&((struct my_dev*)0)->ttimer)));
```

- result
- cast result
- Given ptr to ttimer
- Mask offset
- Reference of struct my_dev type nullptr pointing (offset) to ttimer (0x08 – 0x00)

EXAMPLE:

*struct my_dev *myd = container_of(0x12340008, struct my_dev *, ttimer)*

=> myd = 0x1234000 (0x12340008 – 0x08)

KERNEL MEMORY MANAGEMENT

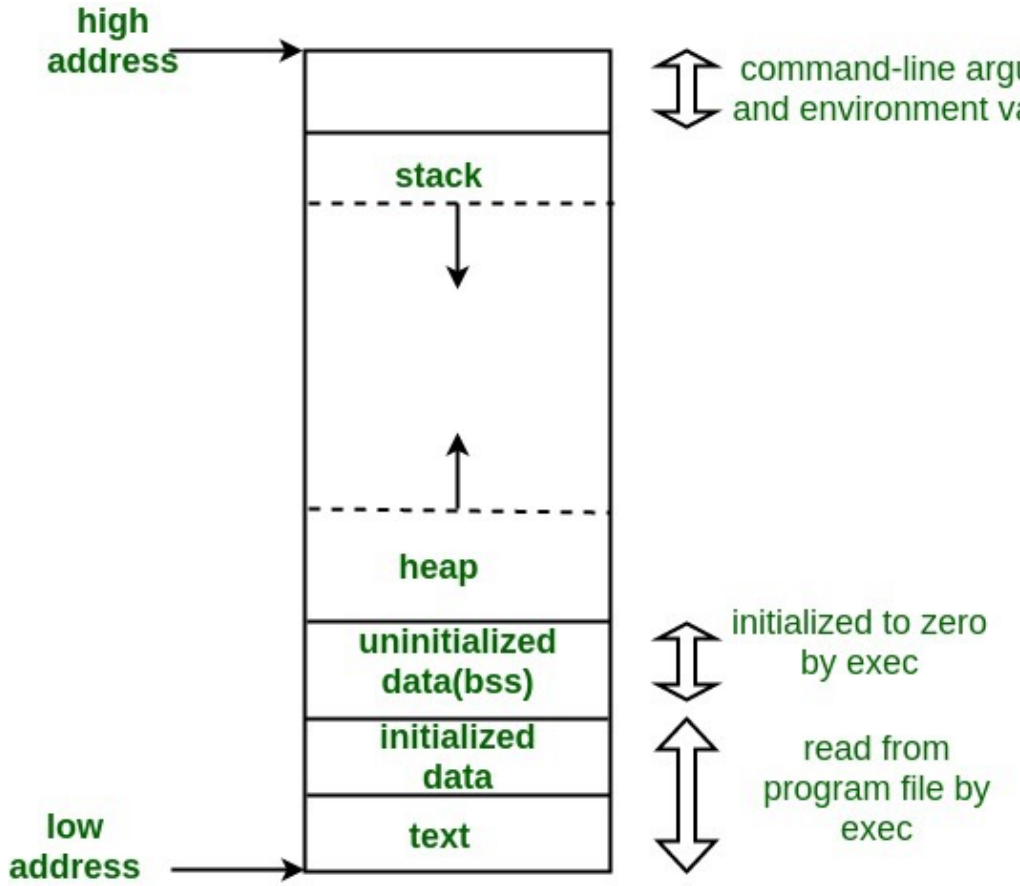
MEMORY LAYOUT OF AN APPLICATION

Bare metal and OS-based programs all use different types of memory:

- Text: program code
- Data: Initialized variables and constants
- BSS: Uninitialized variables
- Heap: Dynamically allocated Memory
- Stack: Stacked Memory

In bare-metal, ex PSoC, with one program, this occupies the whole physical memory

I multitasking OS'es like Linux, we have a one set per application



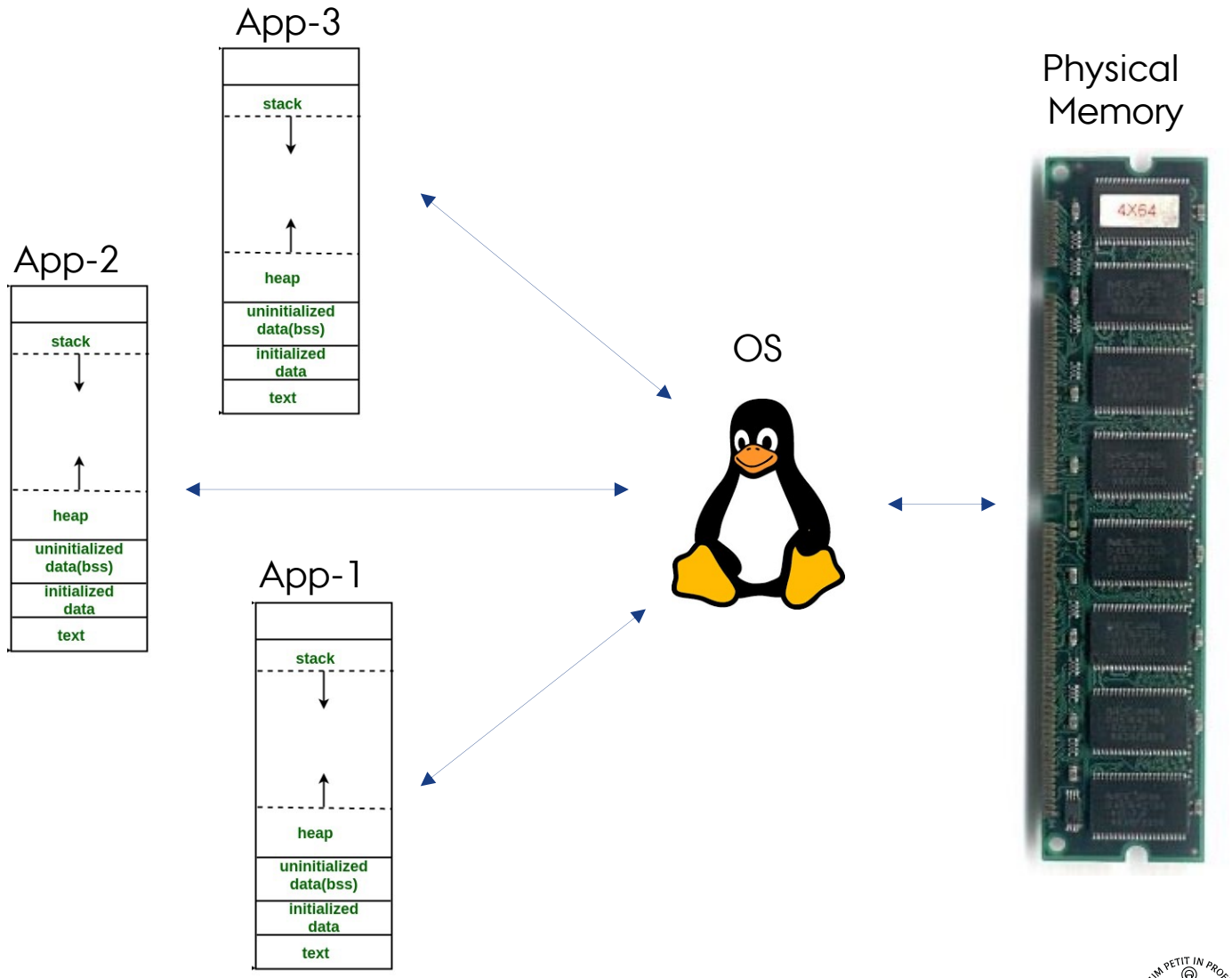
Source: geeksforgeeks.org

MAPPING APPLICATIONS TO MEMORY

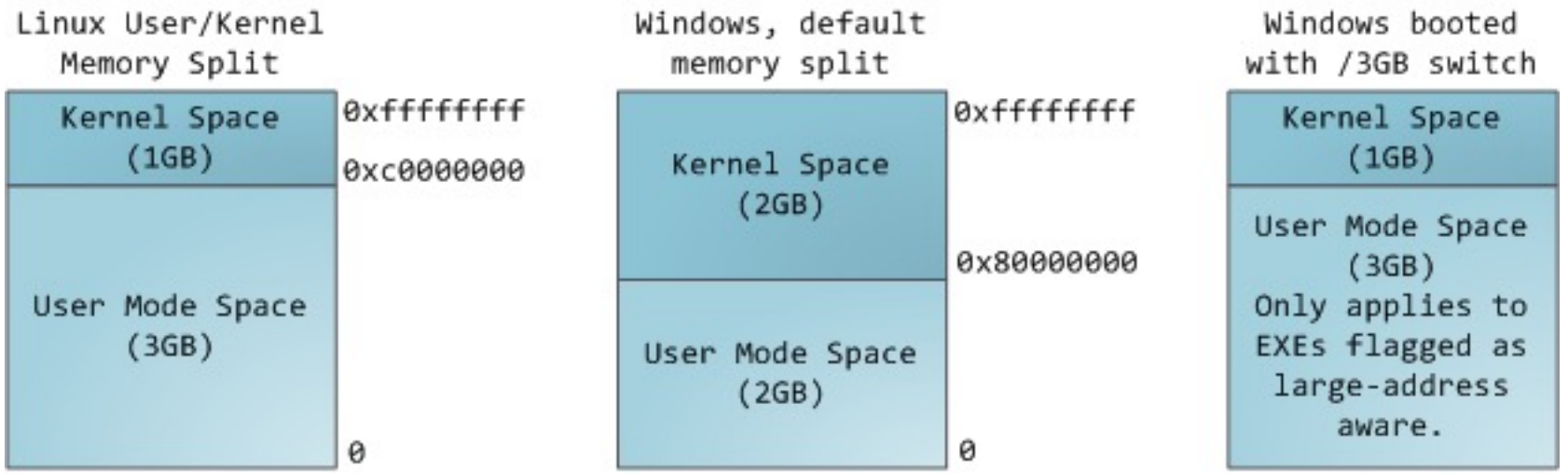
Multitasking system must map memory for all applications into physical memory

It must also be secured that user space applications cannot acces OS memory and other applications memory

How can this be achieved?



USER- / KERMEL MEMORY SPLIT



https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

For security reasons Linux divides memory into two segments:

- Kernel space (priviliged memory)
- User space (unpriviliged memory)

32-bit systems have 2^32 = 4GB adressable memory. 64-bit have 2^64 = 17E6 GB

Kernel space has its own heap/stack when working on behalf of user space process

VIRTUAL MEMORY AREA

Each process has its own sandbox of memory called *Virtual Memory*

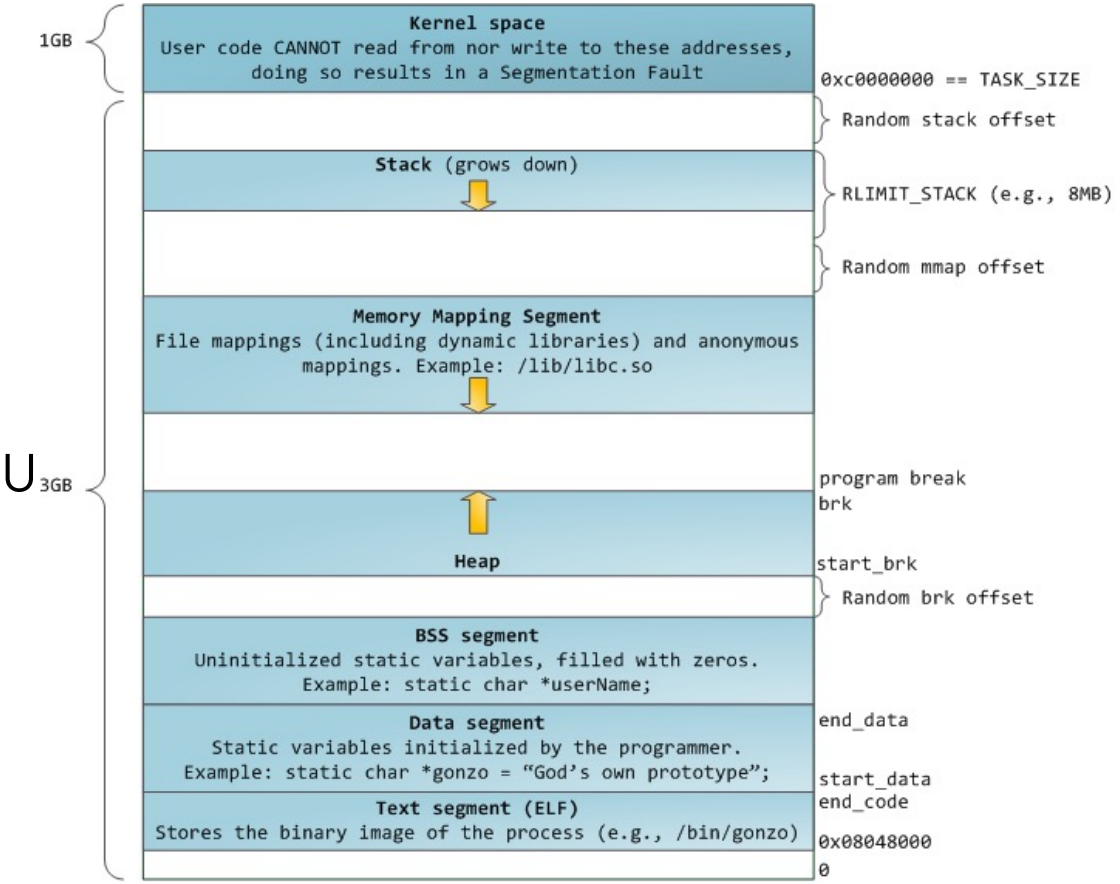
It has a full 4GB (32-bit) memory span and includes heap, stack, code and data segments

The virtual memory is mapped to the physical memory using a hardware device called the MMU

When a program is loaded (and run), Linux allocates the VMAs required.

If more VM is used than is physically available, then it may be swapped to slower memory (hdd)

Process Memory Layout



https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

MMU

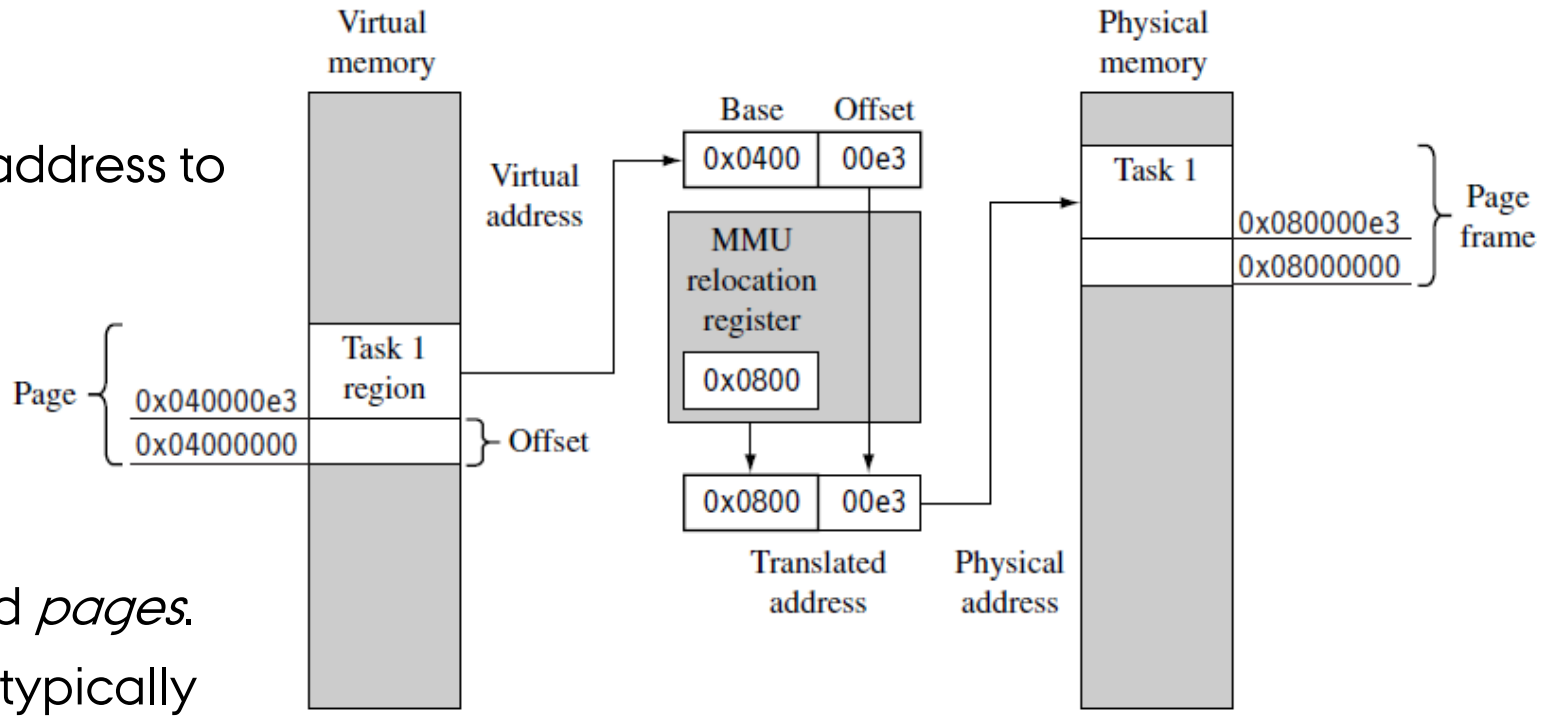
Each VMA has its own internal start address and size

The MMU translates the internal address to a physical address

The translation is typically just an offset

VMA is allocated in chunks called *pages*.

Pages have a fixed size in Linux, typically 4KB



PAGE TABLES

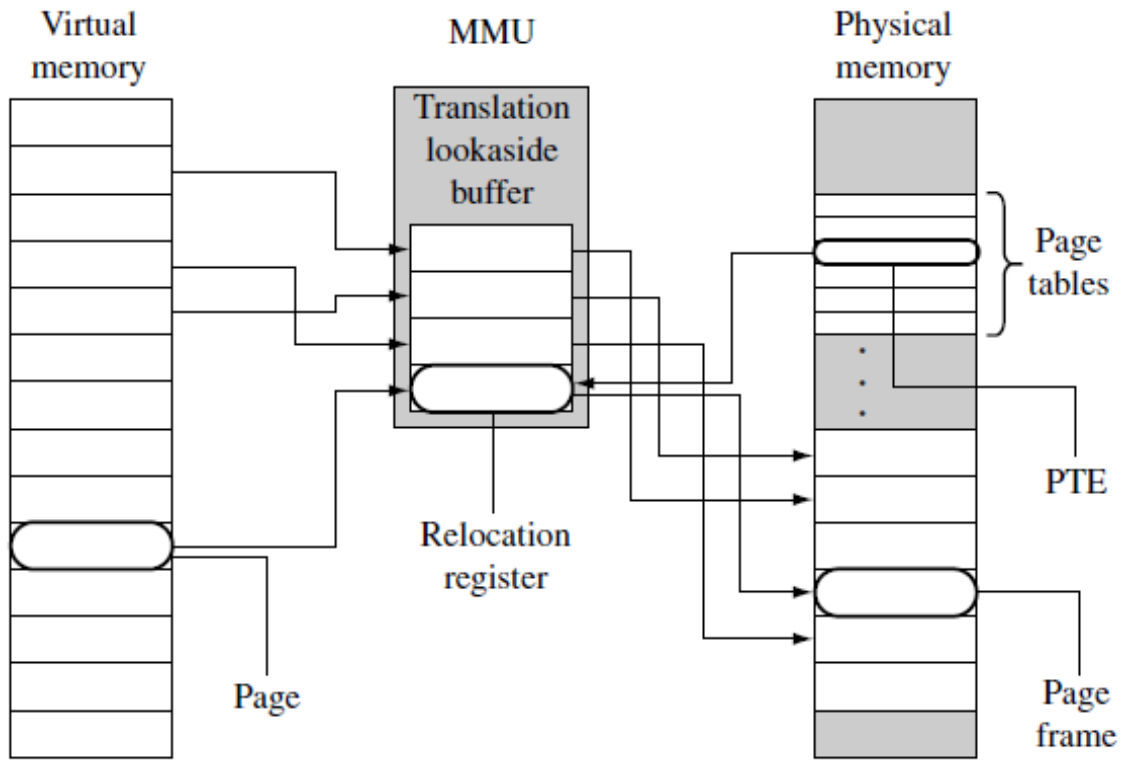
Page tables store information about:

- Physical address + size
- Access permissions
- Cache + Write buffer info

The Translation Lookaside buffer (TLB) is a cached version of the most recently used page tables

Page tables are managed by Linux. Each PTE is related to one proces

Multiple PTEs can point to the same physical address. Why?



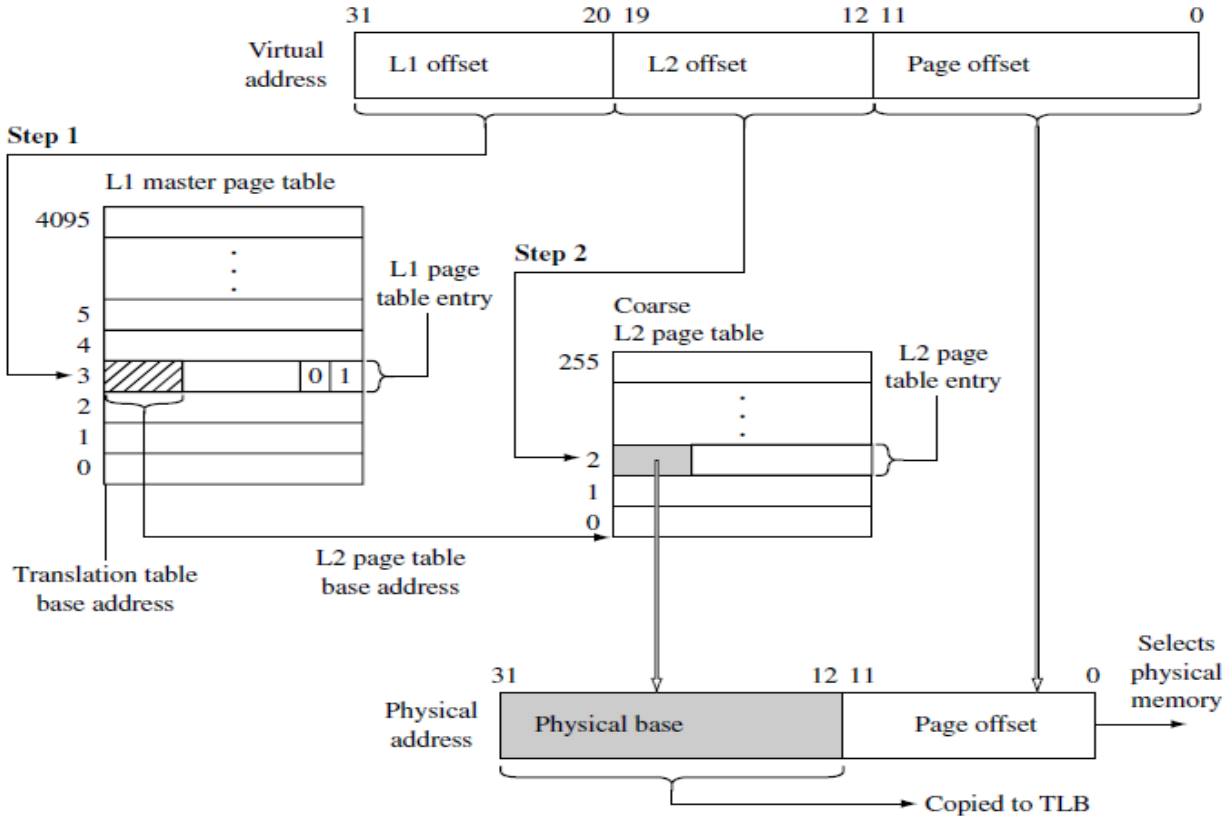
ADDRESS TRANSLATION

Translation is done as follows:

1. Program issues access to a virtual address
2. Upper address bits select PTE in L1 table
3. L1 PTE contains pointer to an L2 table. “L2 offset” selects index in table.
4. L2 PTE contains pointer to physical address

This is handled in hardware as it must happen at CPU speed!

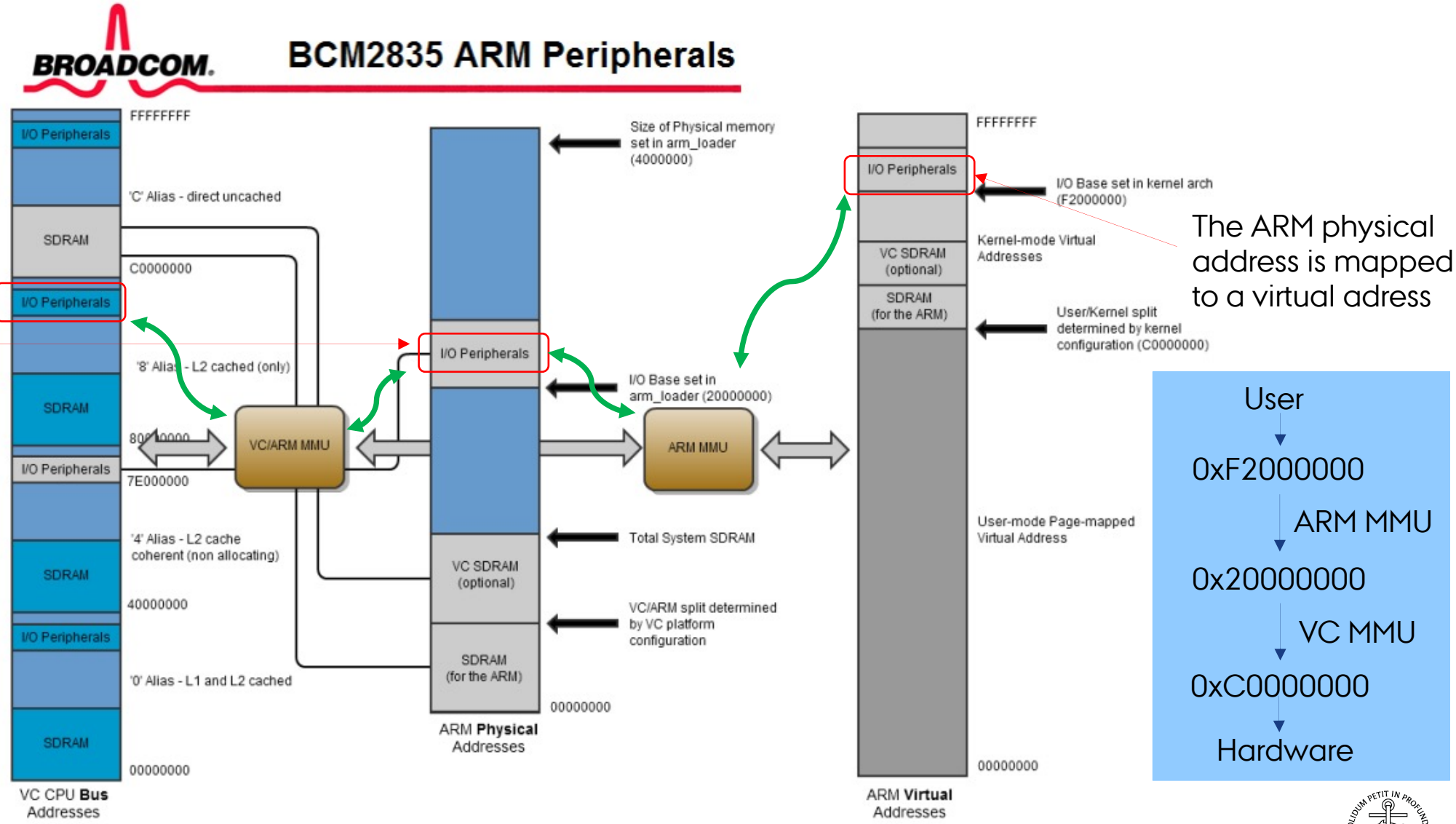
Linux itself is also running in virtual memory, but always with the same offset.



HETEROGENE SYSTEMS

Is physically mapped in the video controller address space

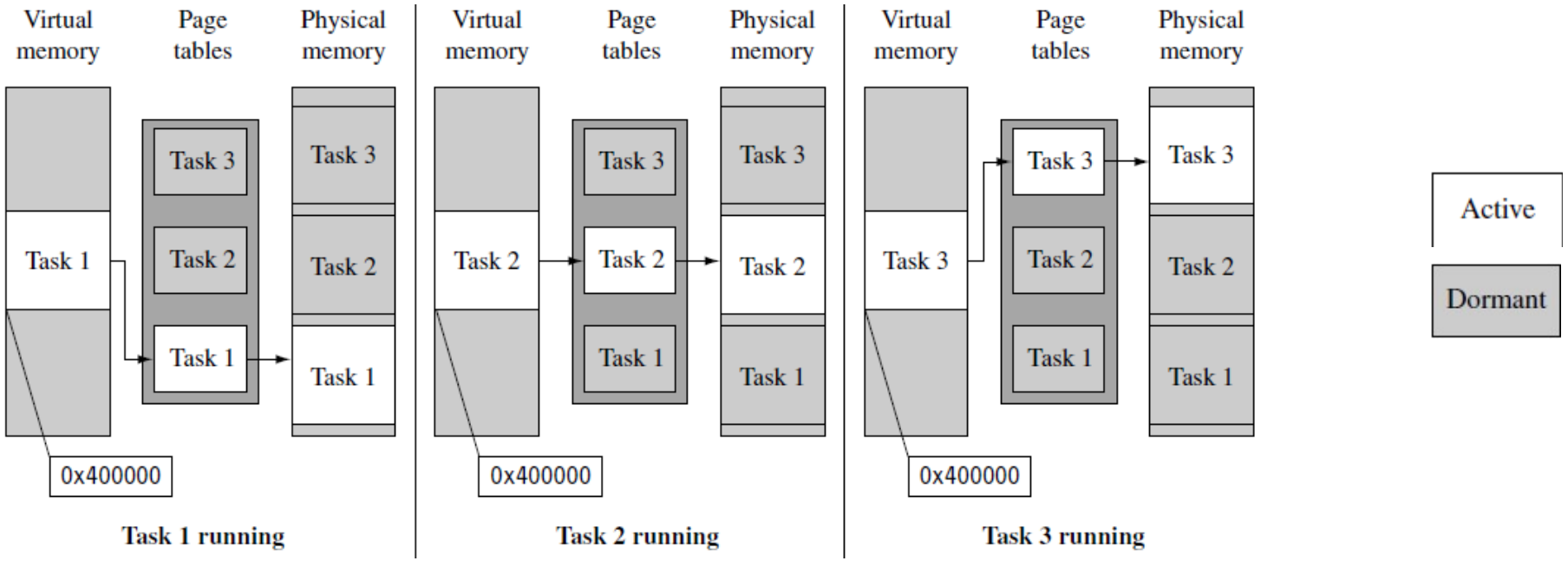
Is mapped to a ”virtual” adress in ARM CPU’s physical memory



The ARM physical address is mapped to a virtual adress

User
↓
0xF2000000
↓ ARM MMU
0x20000000
↓ VC MMU
0xC0000000
↓
Hardware

MULTITASKING



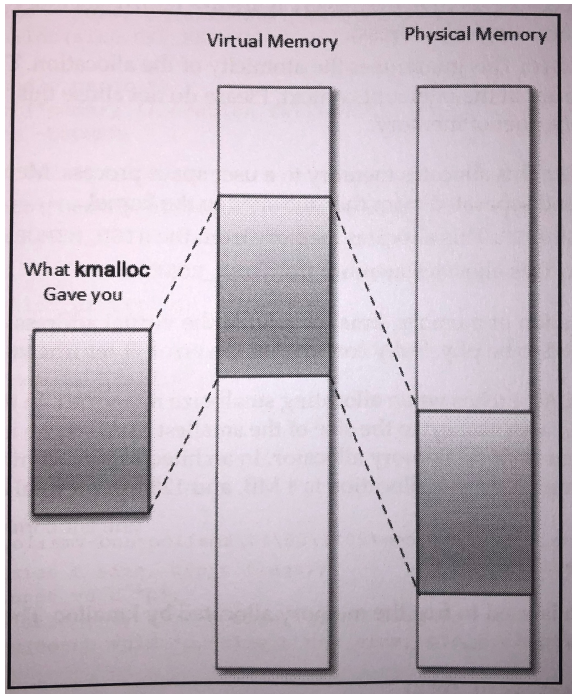
The MMU is build to support multitasking

- The context for the different tasks can all remain in physical memory
- As the OS performs task switching, context is switched by selecting new page table entries
- No data has to be copied during a context switch, which makes it fast!

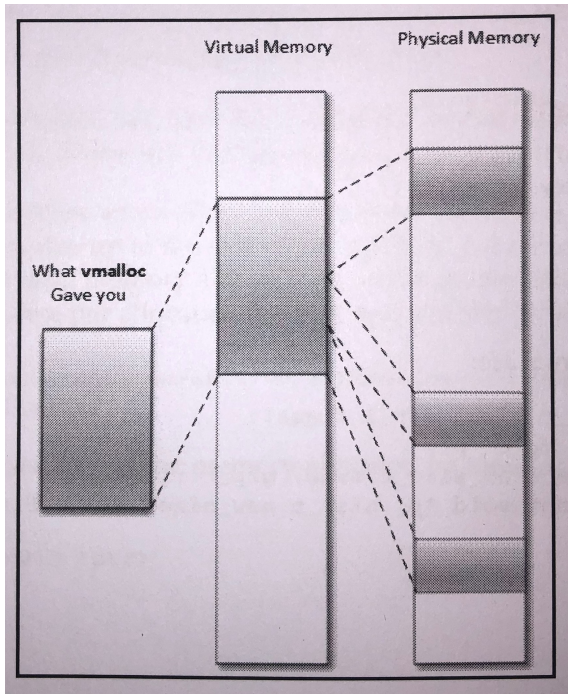
MEMORY ALLOCATION

```
struct gpio_dev {
int gpio;
int dir;
}

int probe(...) {
struct gpio_dev gpio* =
kmalloc(sizeof(my_dev), GFP_KERNEL);
...
}

void remove(...){
kfree(gpio);
}
```

kmalloc



vmalloc



PROCESS MEMORY MAP

```
stud@GoldenImage: cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 658420 /bin/cat
08053000-08054000 r--p 0000a000 08:01 658420 /bin/cat
08054000-08055000 rw-p 0000b000 08:01 658420 /bin/cat
08575000-08596000 rw-p 00000000 00:00 0      [heap]
b73bd000-b75bd000 r--p 00000000 08:01 527475 /usr/lib/locale/locale-archive
b75bd000-b75be000 rw-p 00000000 00:00 0
b75be000-b776b000 r-xp 00000000 08:01 656554 /lib/i386-linux-gnu/libc-2.17.so
b776b000-b776d000 r--p 001ad000 08:01 656554 /lib/i386-linux-gnu/libc-2.17.so
b776d000-b776e000 rw-p 001af000 08:01 656554 /lib/i386-linux-gnu/libc-2.17.so
b776e000-b7771000 rw-p 00000000 00:00 0
b778b000-b778c000 r--p 002c5000 08:01 527475 /usr/lib/locale/locale-archive
b778c000-b778e000 rw-p 00000000 00:00 0
b778e000-b778f000 r-xp 00000000 00:00 0      [vdso]
b778f000-b77af000 r-xp 00000000 08:01 660572 /lib/i386-linux-gnu/ld-2.17.so
b77af000-b77b0000 r--p 0001f000 08:01 660572 /lib/i386-linux-gnu/ld-2.17.so
b77b0000-b77b1000 rw-p 00020000 08:01 660572 /lib/i386-linux-gnu/ld-2.17.so
bfab4000-bfad6000 rw-p 00000000 00:00 0      [stack]
```

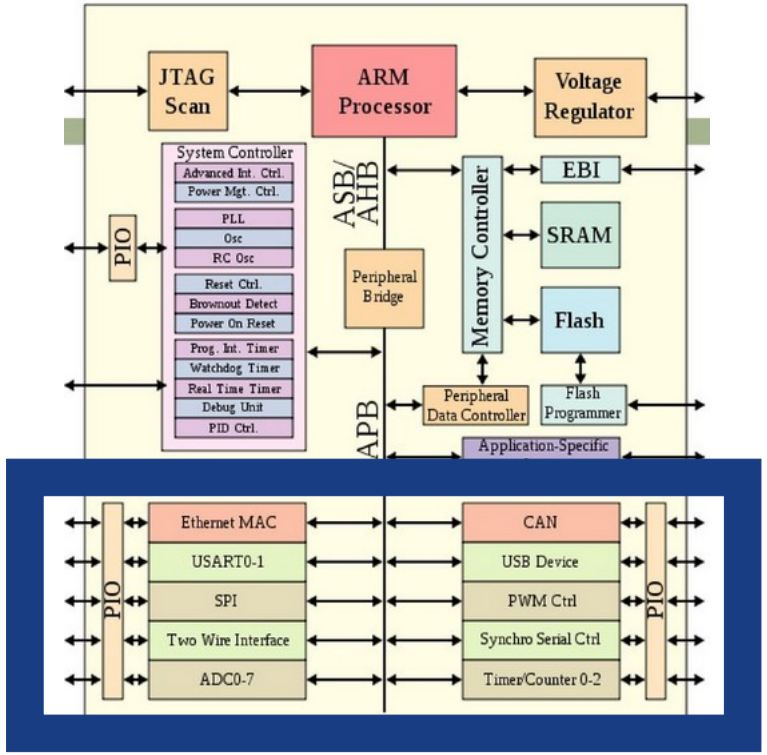
Access Rights

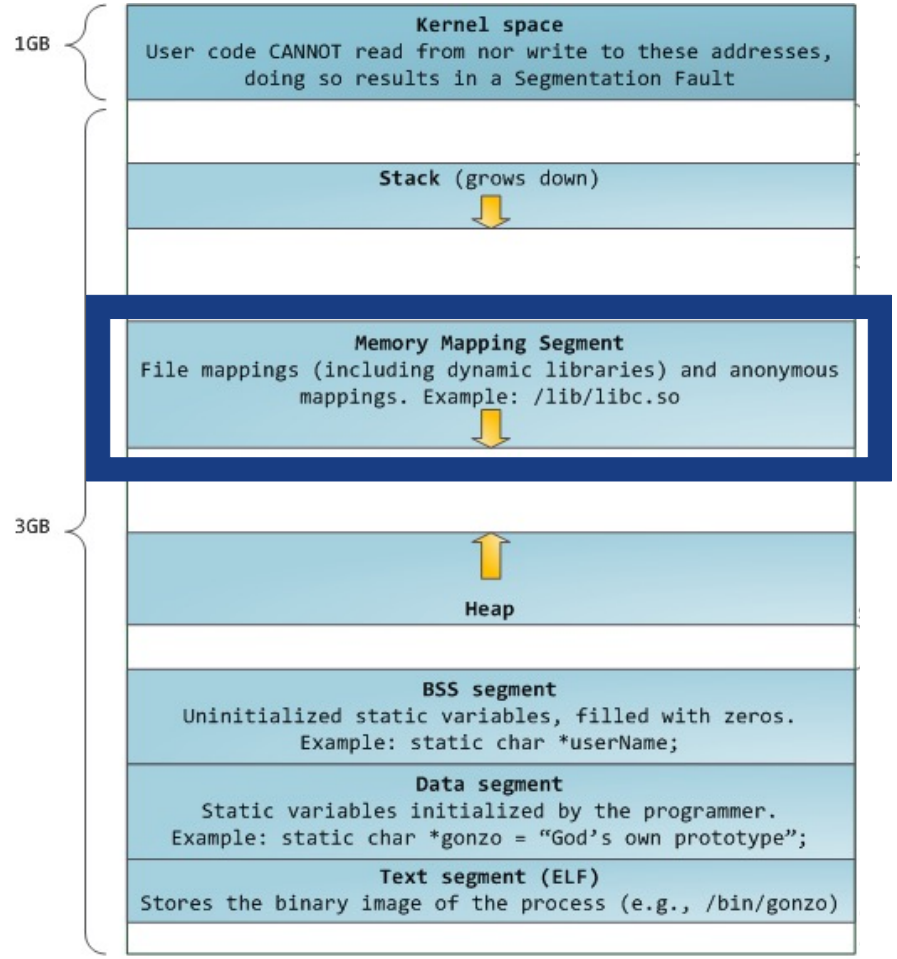
virtual dynamic shared object

I/O MEMORY

Hardware devices, their buffers and control registers are mapped into virtual memory

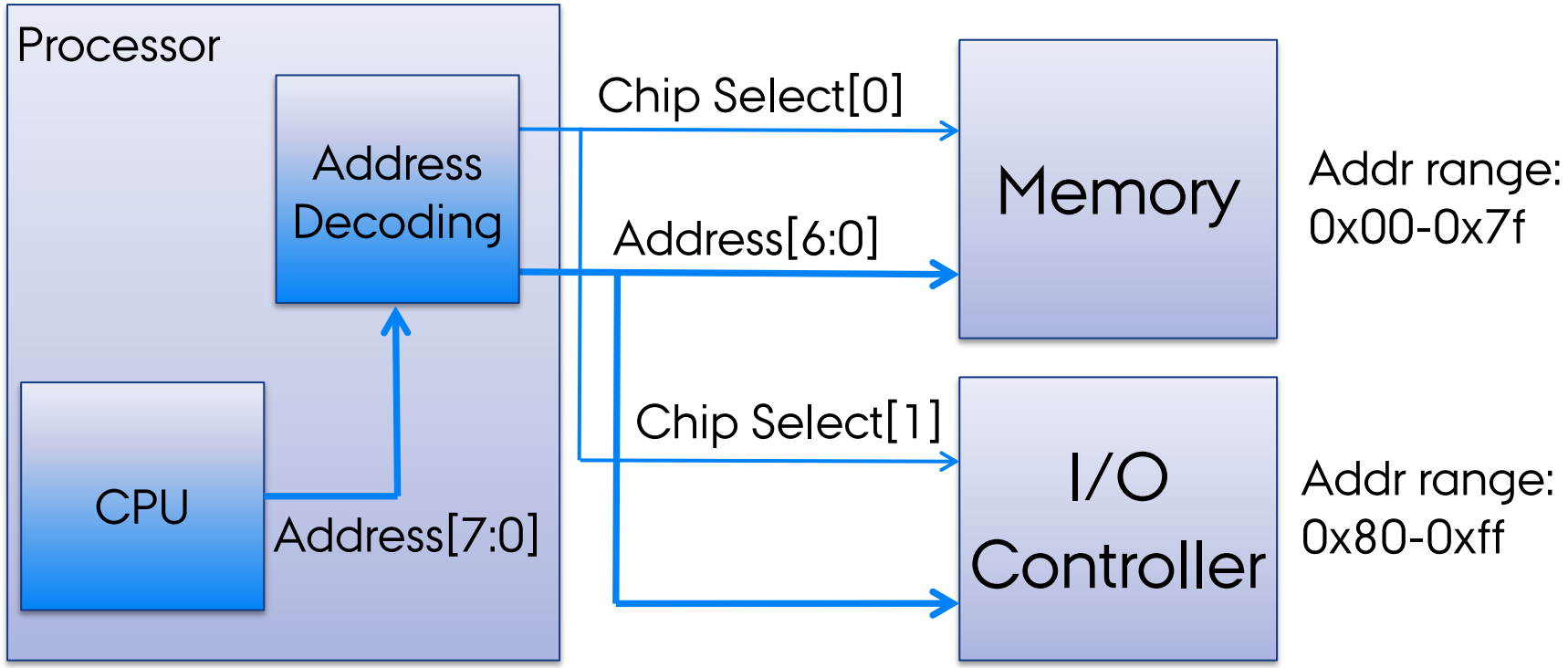
Each hardware block in the processor has its own address range





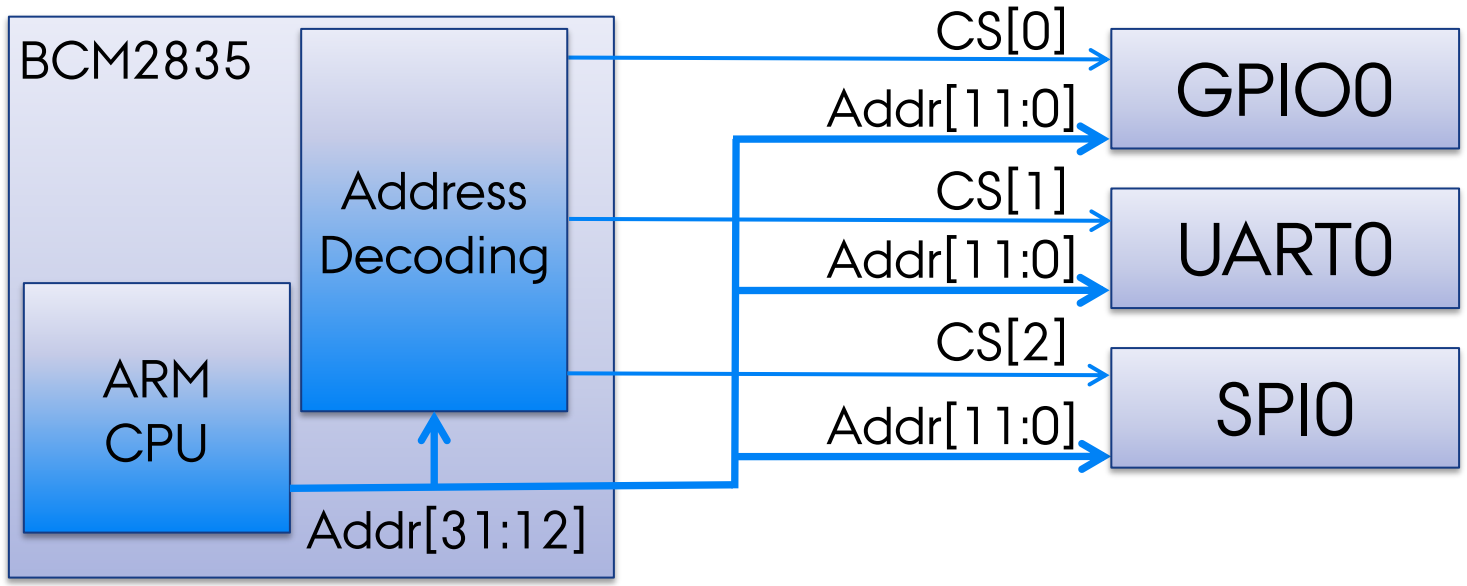
https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

ADDRESS DECODING EXAMPLE




```c
int main(int argc, char *argv[])
{
  u8 * mem_ptr = 0x04; // Pointer to Memory
  u8 * io_ptr = 0xf8;  // Pointer to I/O Controller
}
```

BCM2835 PERIPHERALS



Address	Size	Name	Description
0x7e200000	0x1000	GPIO	GPIOIO
0x7e201000	0x1000	UART0	UART0
0x7e202000	0x1000	ALTMMC	Alternate MMC
0x7e203000	0x1000	PCM	PCM
0x7e204000	0x1000	SPI0	SPI0

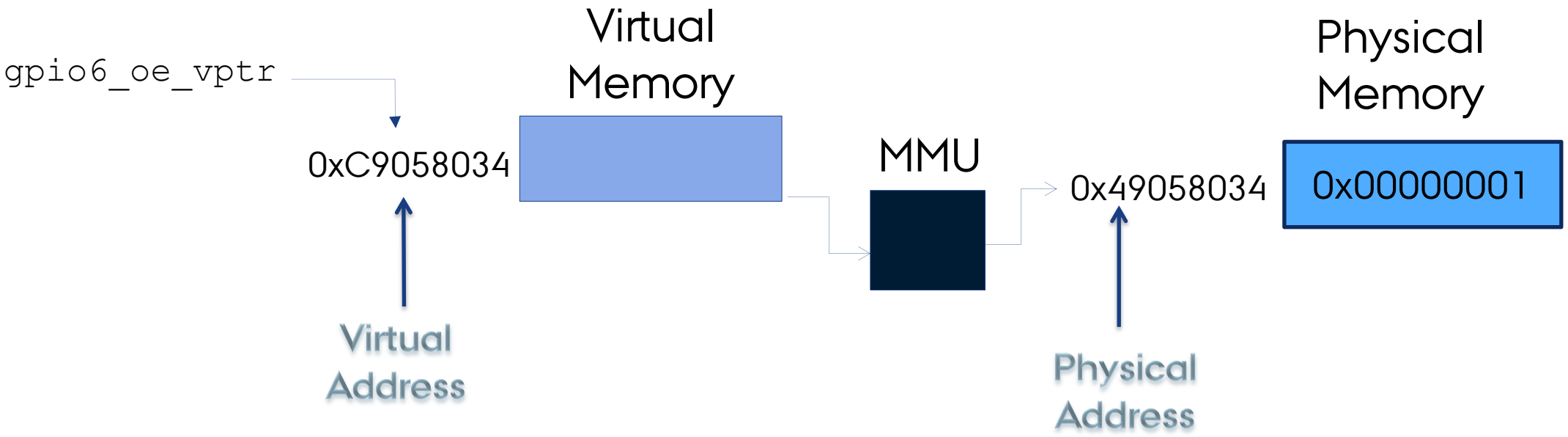
I/O MEMORY ACCESS IN KERNEL SPACE

In kernel code, physical memory can be accessed through the <ioport.h> interface:

```
ssize_t mygpio_write(struct file *filep, const char __user *ubuf,
size_t count, loff_t *f_pos)
…
unsigned long phys_addr = 0x49058030;
unsigned long num_bytes = 16; // 4 x 32-bit
request_mem_region (phys_addr, num_bytes, “gpio_bank6“);
unsigned long gpio6_oe_vptr = ioremap(0x49058034, 4);
unsigned long gpio6_din_vptr = ioremap(0x49058038, 4);
unsigned long gpio6_dout_vptr = ioremap(0x4905803C, 4);

iowrite32(ioread32(gpio6_din_vptr)|(1<<7), gpio6_dout_vptr);
…
iounmap(<addresses previously mapped>);
release_mem_region(phys_addr, num_bytes);
```

I/O MEMORY ACCESS IN KERNEL

Physical Register Addresses are mapped to logical addresses in the kernel




```
gpio6_oe_vptr = ioremap(0x49058034, 4);
```

MEMORY ACCESS IN USER SPACE

Physical Memory can be accessed from User Space through /dev/mem:

```c
/* Adapted from http://www.lartmaker.nl/lartware/port/devmem2.c */
#define MAP_SIZE 4096UL // One memory page
#define MAP_MASK (MAP_SIZE - 1)
int main(void)
{
unsigned int *map_base, *virt_addr;
unsigned int phys_base = 0xbeef0000; // Page aligned
unsigned int offset = 0x00000004; // Byte address offset
int fd = open("/dev/mem", O_RDWR, 0);

map_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
fd, (unsigned int)(phys_base & ~MAP_MASK));
virt_addr = map_base + offset/4; // base is uint (32-bit)
read_result = *virt_addr; // read word (32-bit)
```

MEMORY ACCESS IN USER SPACE

- Map_size = 1 page = 4096 bytes = 0x1000
- Map_mask = 4096-1 = 0x0fff

- Addr = Physical base addr. (0 in example)

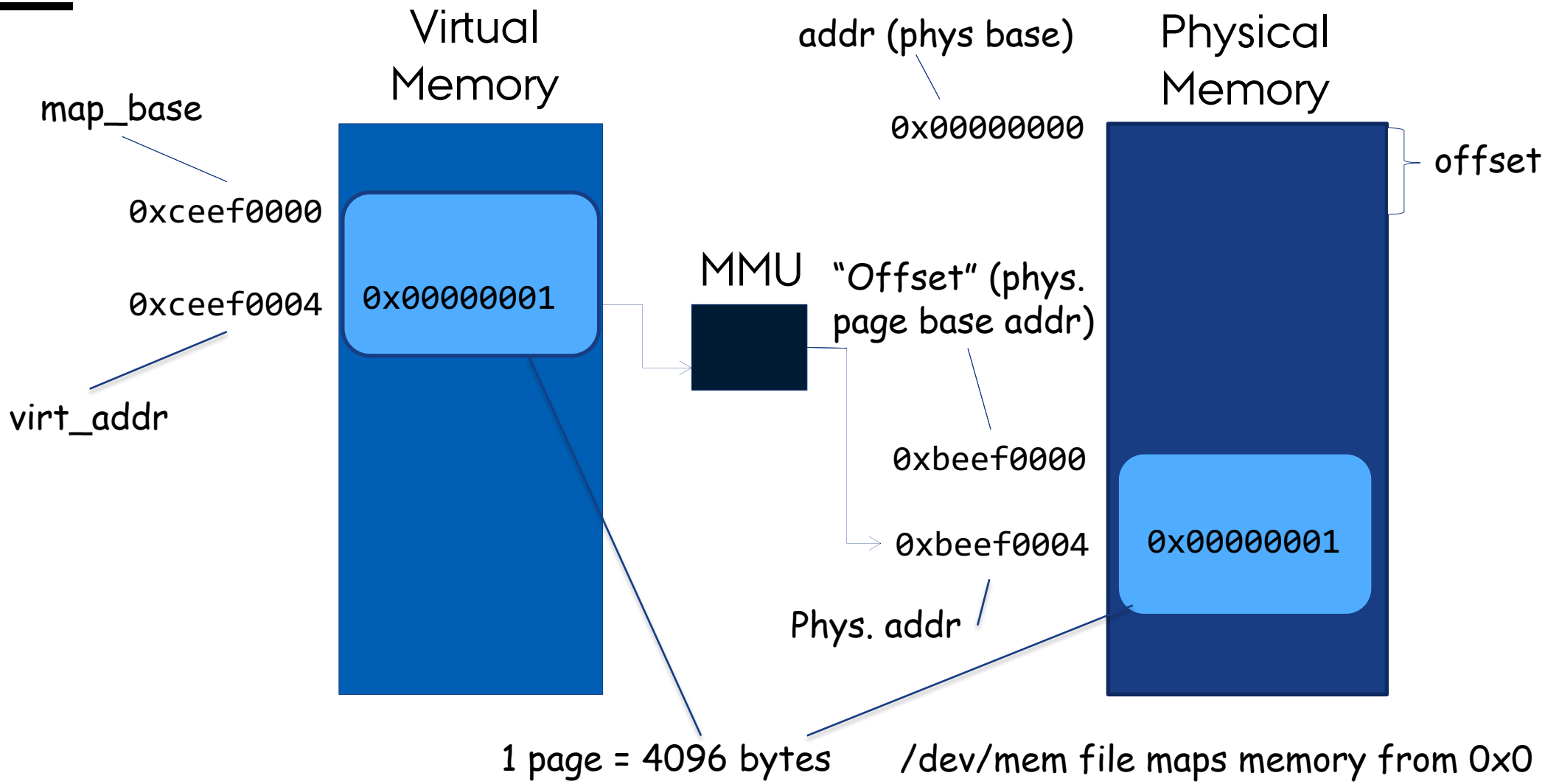
- Offset = page aligned offset from addr (0):
  - Ex: offset = 0xbeef0000 & ~(0x00000fff) = 0xbeef0000

- Map_base = base addr. of logical page, returned by mmap

- Virt_addr = map_base + offset
  - Ex: Virt_addr = 0x**ce**ef0000 + 0x04/4 (32-bit words) = 0x**ce**ef0004

  Note! map_base is 32-bit pointer, so map_base++ = byte addr + 4!

MEMORY ACCESS IN USER SPACE



AARHUS
UNIVERSITY