

arm

Embedded Linux

Aimen Bouchhima
Enicarthage 2020- 2021

Summary

Introduction

Embedded Linux anatomy

Configuration & Build Process of an Embedded Linux System

Linux kernel modules and device drivers

Why Linux-based Embedded Systems?

Open Source (under GNU General Public License v2.0 : GPLv2)

- The full source code is available for learning and adaptation

Engaged community maintaining and improving Linux regularly

- Companies
- Individuals
- Academics
- Hobbyists

Flexible and adaptable: supports many hardware/System-on-Chip (SoC) configurations

- Based on ARM, x86, PowerPC, SPARC, etc.

Proven in many different scenarios (see next slides)

Supported by a very large ecosystem of software

- Bootloader, system programs, networking services, advanced graphic services, etc.

Royalty-free

Linux-based Embedded System Components

Bootloader

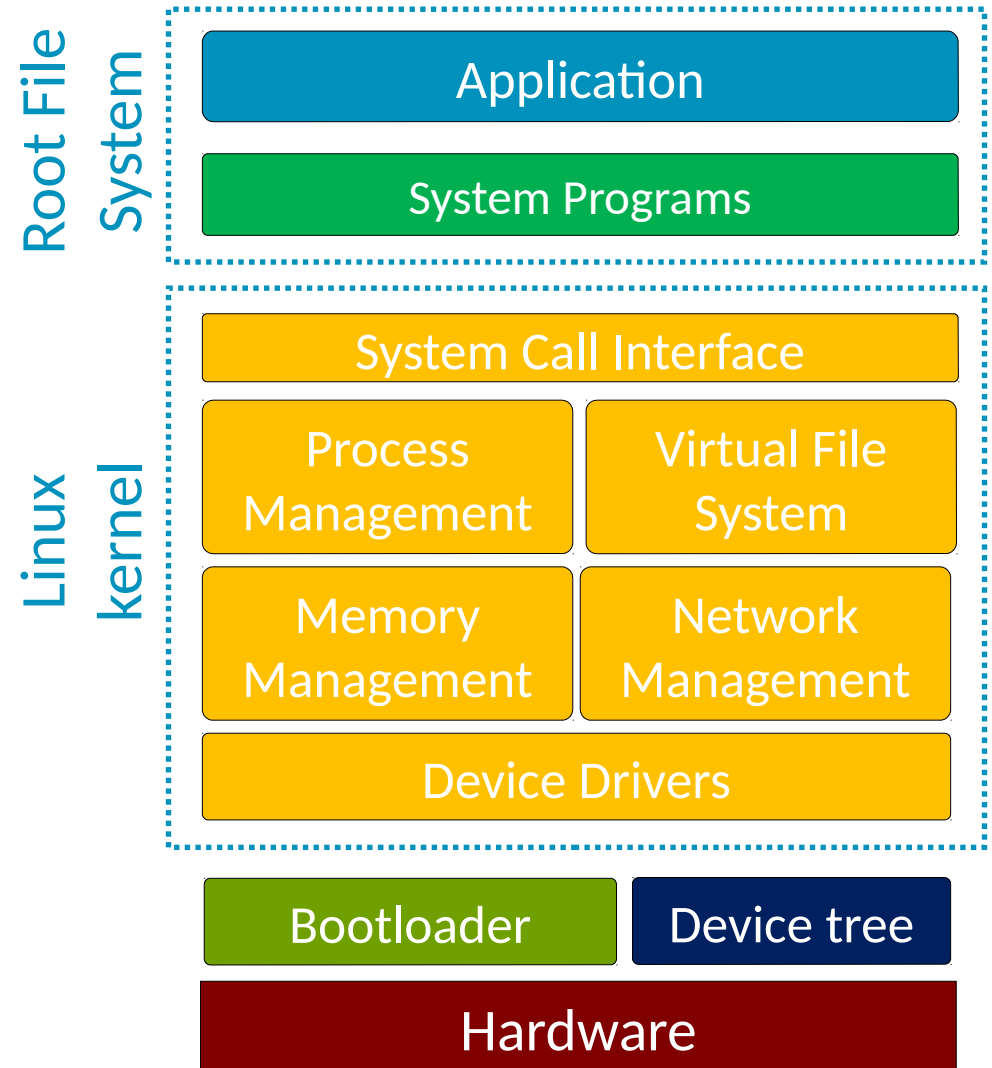
- Software executed at power-up to set-up the hardware to run the operating system

Device tree

- A tree data structure with nodes that describe the physical devices in the hardware needed by the Linux kernel to initialize properly the device drivers

Linux Kernel

- The operating system code providing all the services to manage the hardware resources



Linux-based Embedded System Components

System programs

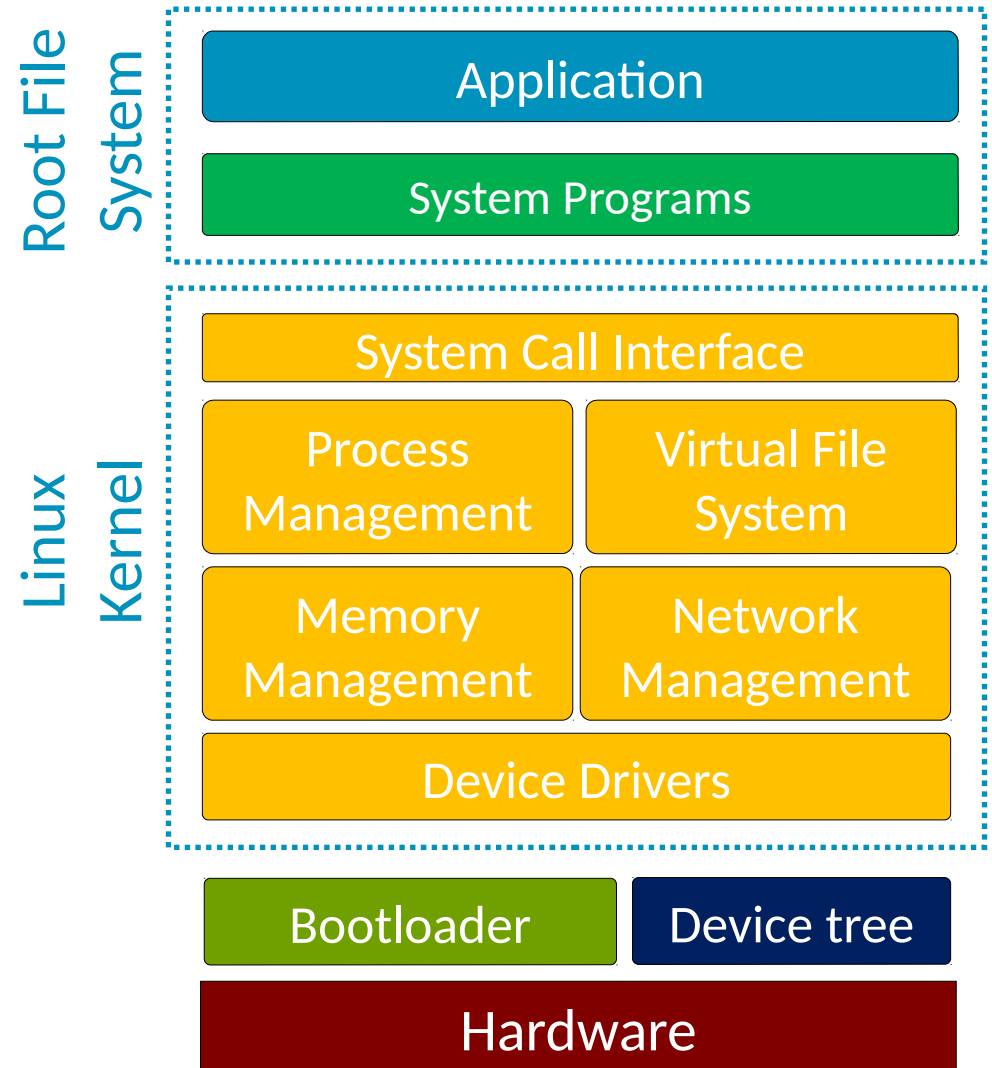
- User-friendly utilities to access operating system services

Application

- Software implementing the functionalities to be delivered to the embedded system user

Root filesystem

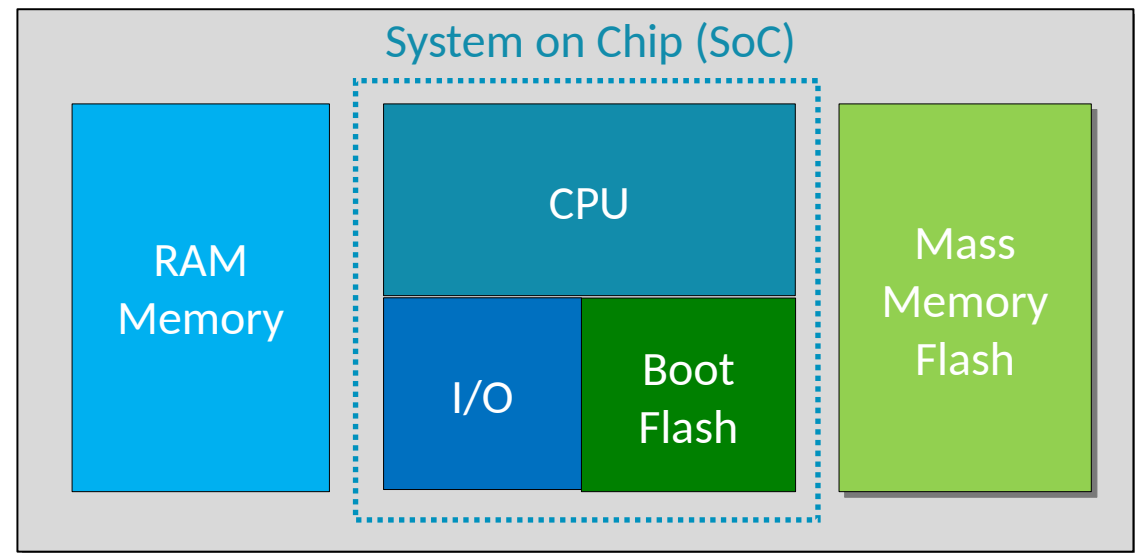
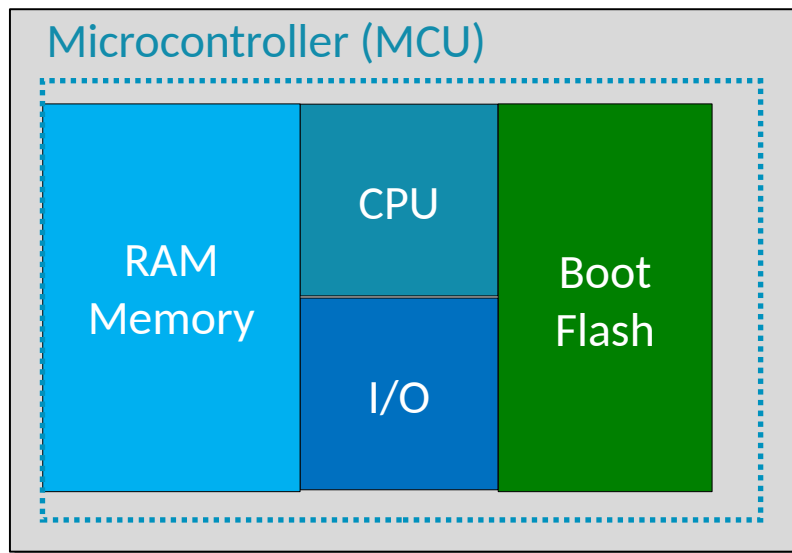
- Container for the Linux Kernel configuration files, the system programs, and the application



Reference Hardware Model Implementations

Multiple implementations of the reference hardware model are possible. For example:

- **Microcontroller-based implementation:** a single device hosts most of the reference model components (e.g. CPU, RAM memory, boot flash).
- **System-on-Chip implementation:** most of the reference model components are discrete components, while the CPU is integrated with some of them (e.g. I/O, boot flash).



CPU Memory Map

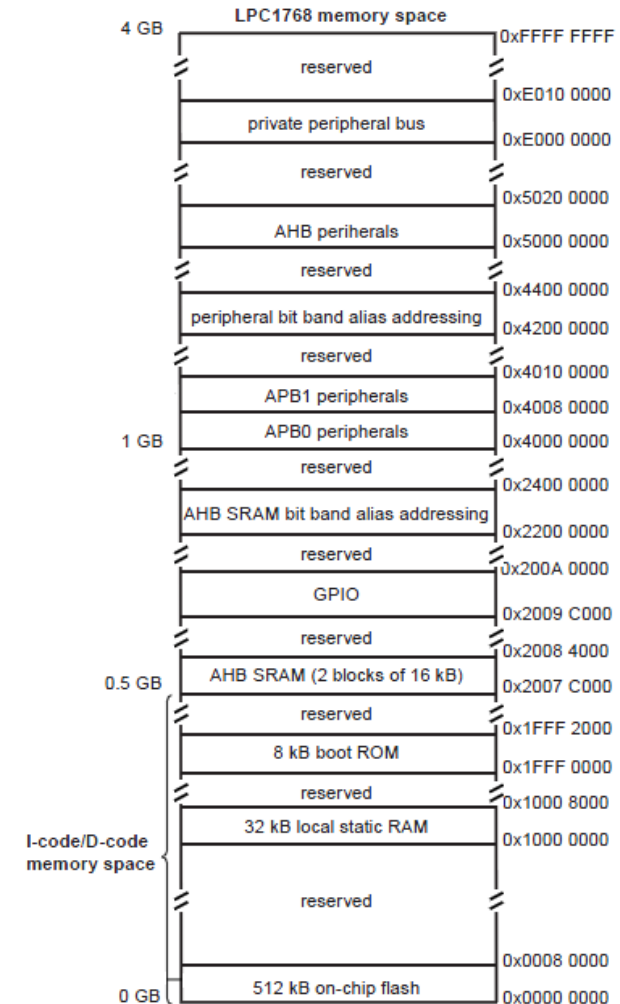
The CPU generates 2^N different addresses ($0 \leq 2^N - 1$).

- N=number of bits of the address bus

Each device (memory, I/O, etc.) is associated with a range of addresses.

The **memory map** describes this association for all the devices.

Memory map of NXP
LPC1768 (Cortex-M3)



Summary

Introduction

Embedded Linux anatomy

- Bootloader

- Kernel

- Root filesystem

- Device tree

Configuration & Build Process of an Embedded Linux System

Linux kernel modules and device drivers

The Role of the Bootloader

When **powered-up**, the processor needs a simple way to get the information it needs.

- Where the software is located
- How to get access to the software location
- Where the stack is located

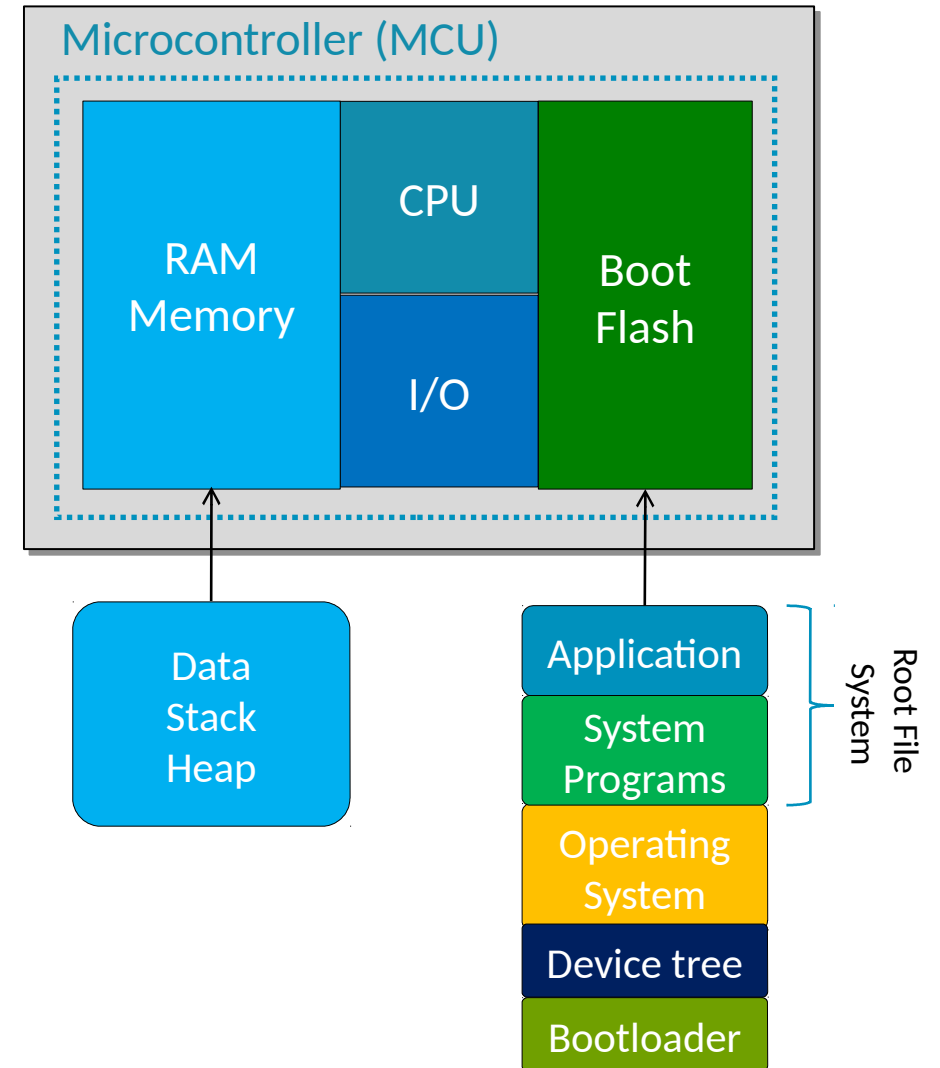
At power-up, the **program counter** is set to a default known value, the **reset vector**.

- The software starting at the reset vector, which will load the **bootloader**, which takes care of providing the processor all the needed information.
- The operations performed depend on the system architecture.

Possible Scenarios

Scenario 1, typical of microcontrollers

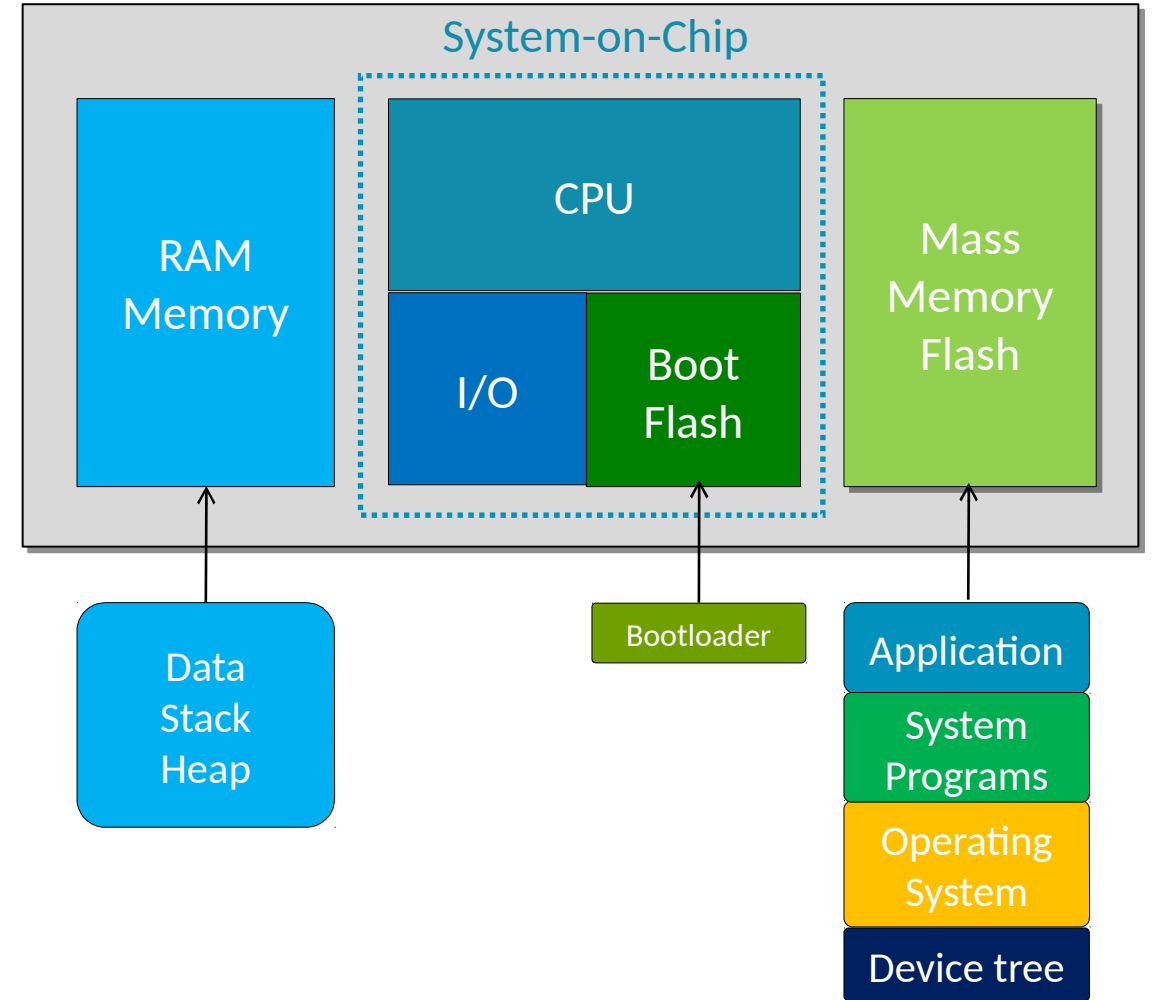
- All the sw (bootloader + device tree + operating system + root filesystem) is stored in persistent storage (boot flash) embedded in the microcontroller.
- All the sw is executed from the persistent storage.
- The CPU reset vector is located in the boot flash.
- The RAM Memory is embedded in the microcontroller and is used for data, stack and heap only.



Possible Scenarios

Scenario 2, typical of System-on-Chip

- The bootloader is stored into the boot flash.
- The CPU reset vector is located in the boot flash.
- The root filesystem, operating systems, and device tree are stored in the mass memory flash and loaded in RAM memory by the bootloader.
- The RAM memory is external to the SoC. It will store the operating system + application software, root filesystem (if configured as RAM disk), data, stack, and heap.

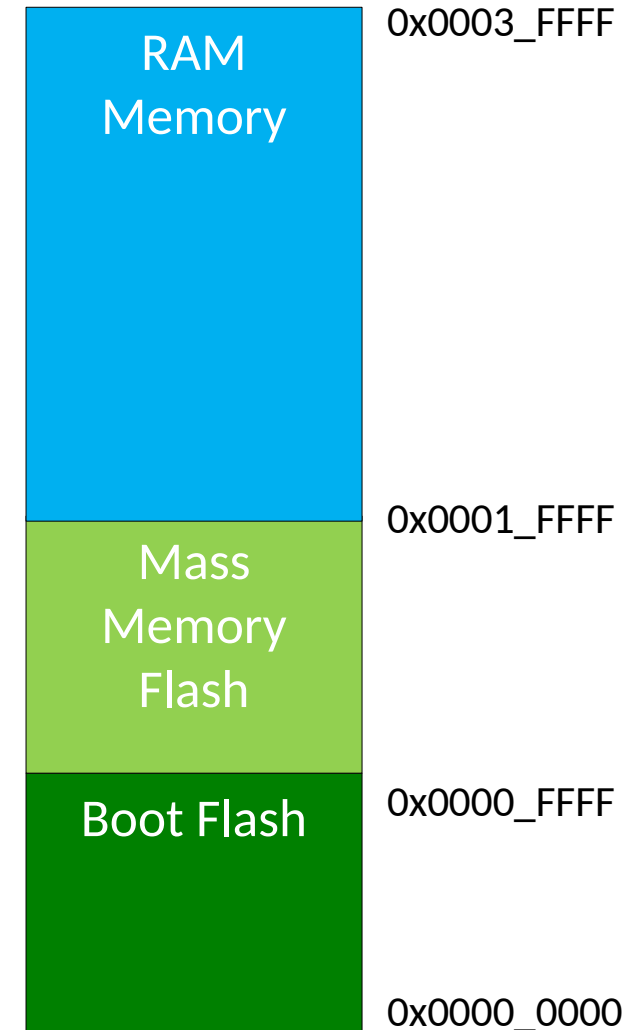


An Example of Bootloader Operations

Let's consider Scenario 2 for a given SoC with the following memory map:

- RAM memory 512 kbytes (arranged as 128 kwords, each 32 bytes long), from 0x0002_0000 to 0x0003_FFFF
- Mass memory flash 256 kbytes (same organization as before), from 0x0001_0000 to 0x0001_FFFF
- Boot flash 256 kbytes (same organization as before), from 0x0000_0000 to 0x0000_FFFF

CPU reset vector



An Example of Bootloader Operations

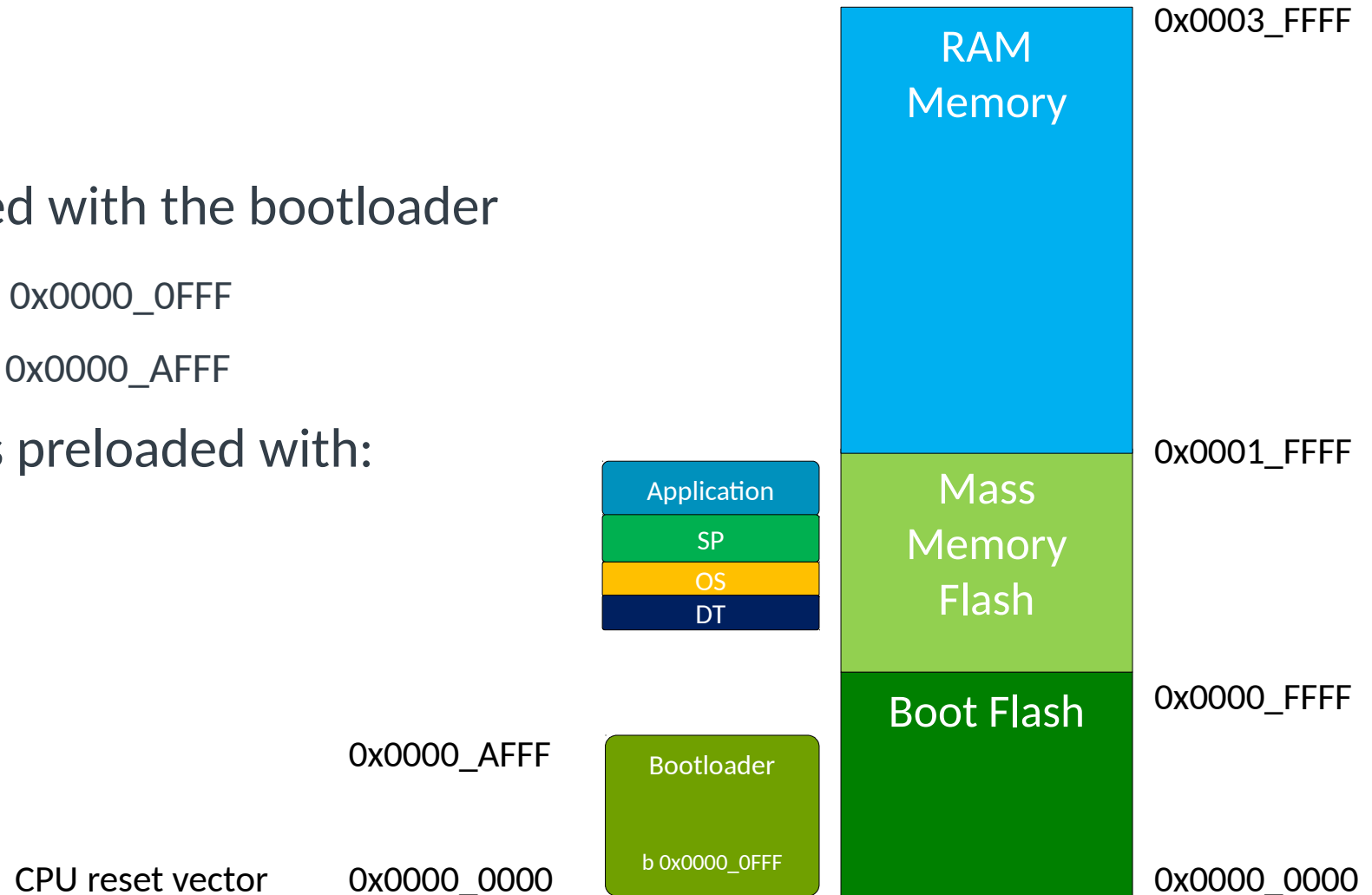
Time = before power up

The boot flash is preloaded with the bootloader

- First bootloader instruction at 0x0000_0FFF
- Last bootloader instruction at 0x0000_AFFF

The mass memory flash is preloaded with:

- Device tree (DT)
- Operating systems (OS)
- System programs (SP)
- Application

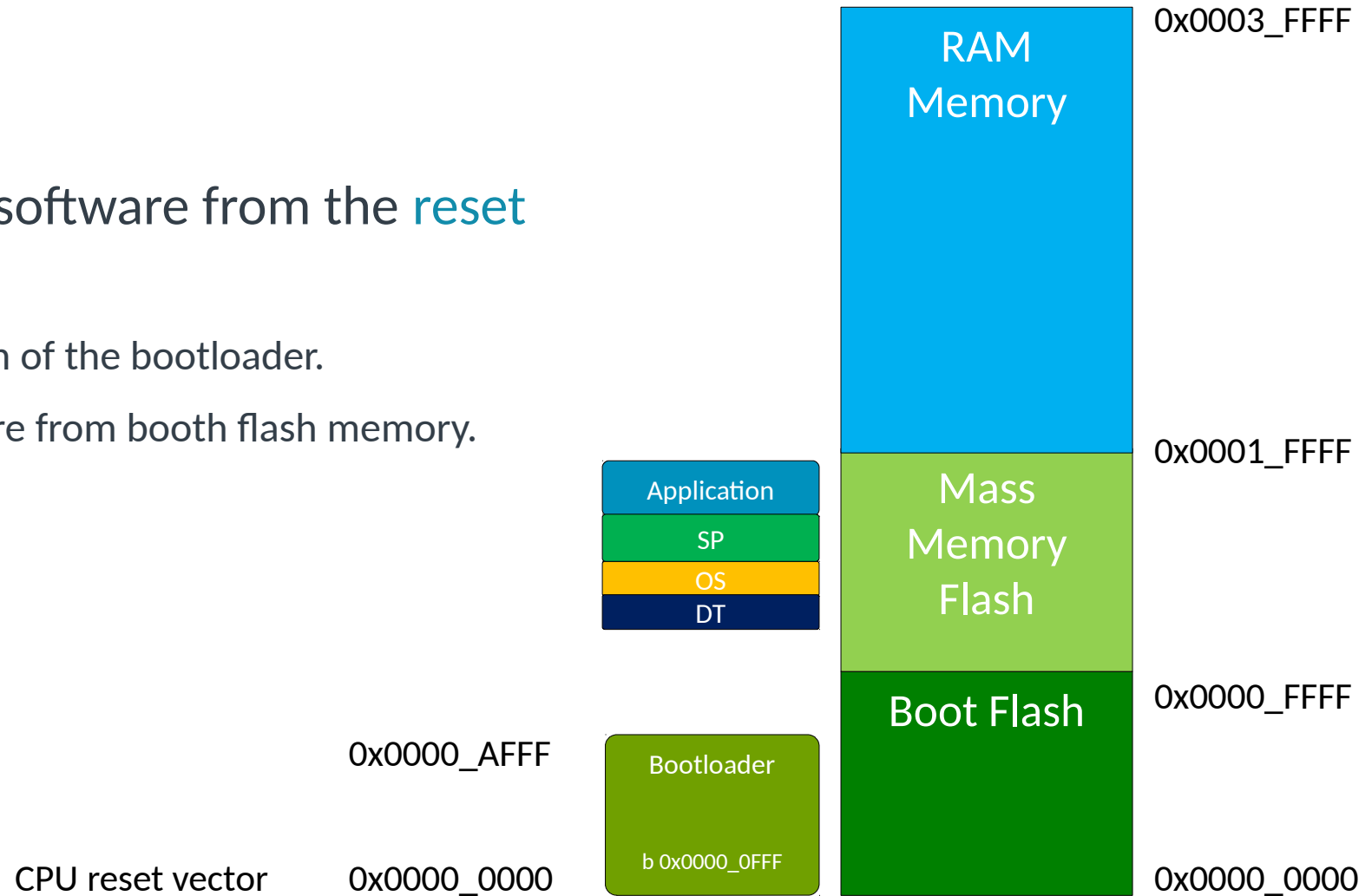


An Example of Bootloader Operations

Time = power up

The CPU starts executing software from the reset vector:

- It jumps to the first instruction of the bootloader.
- It runs the bootloader software from boot flash memory.

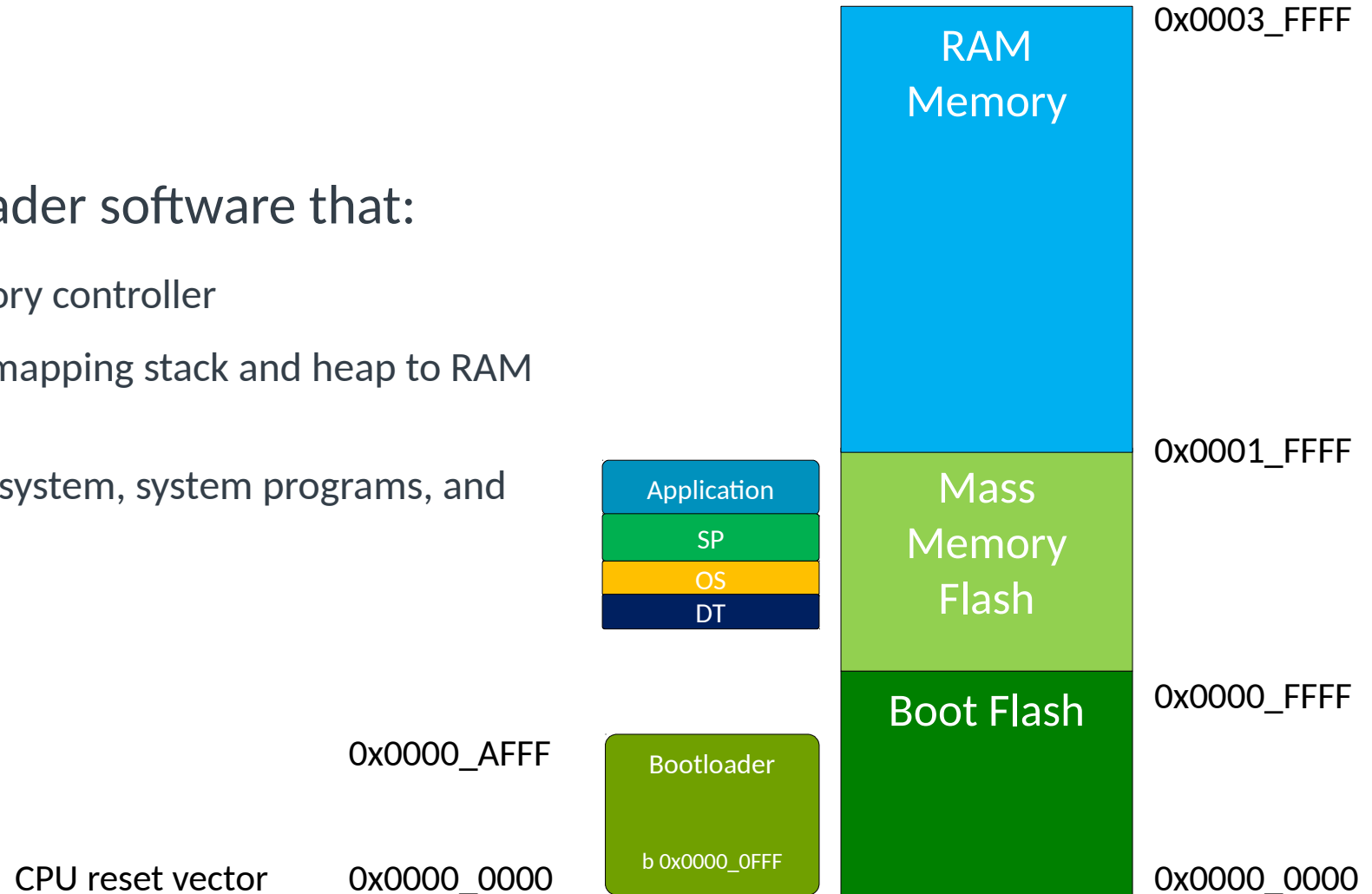


An Example of Bootloader Operations

Time = during bootstrap

The CPU executes bootloader software that:

- Initializes the CPU RAM memory controller
- Sets up the CPU registers for mapping stack and heap to RAM memory
- Copies device tree, operating system, system programs, and applications to RAM memory



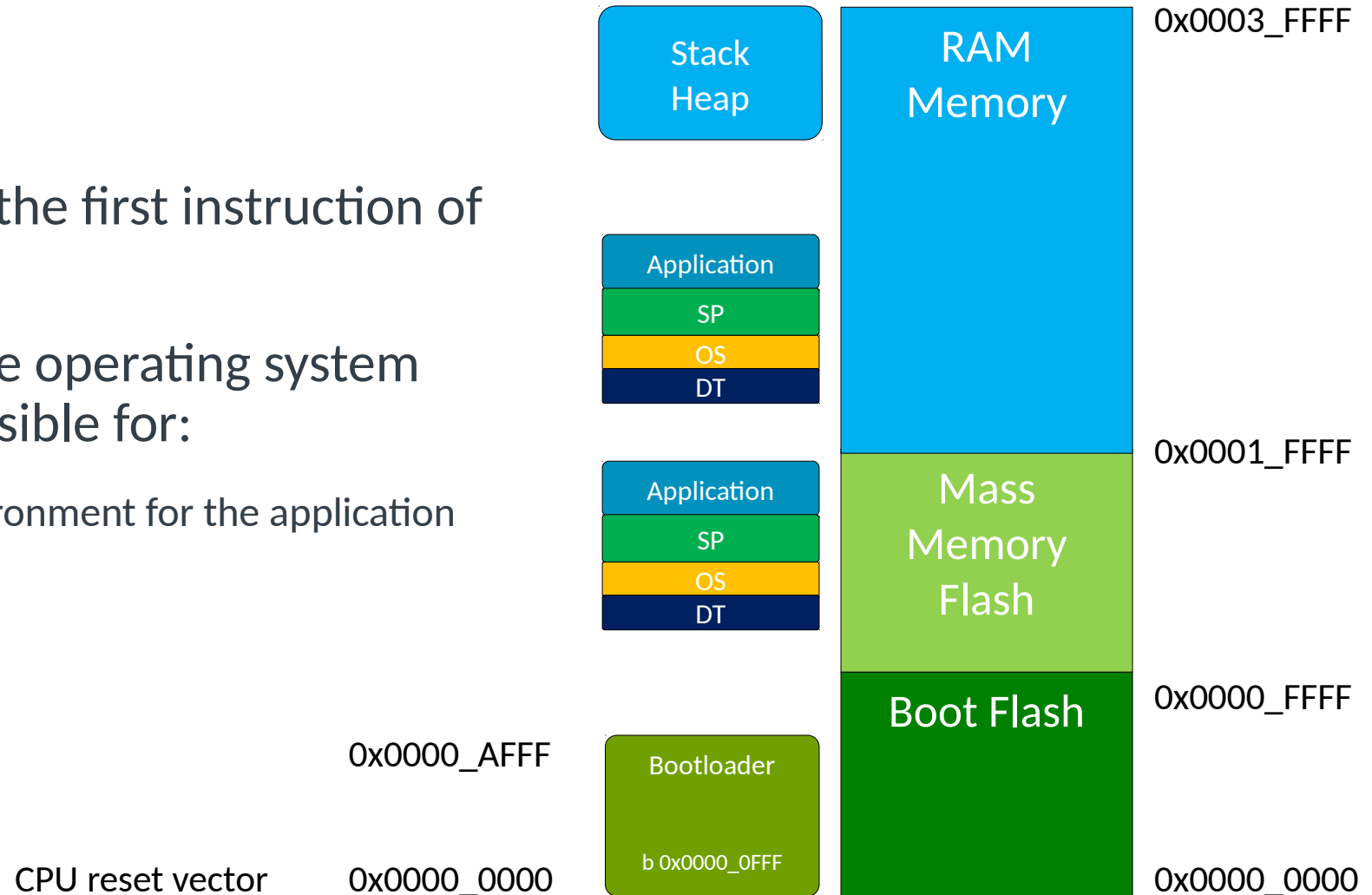
An Example of Bootloader Operations

Time = end of bootstrap

The bootloader jumps to the first instruction of the operating system.

The CPU now executes the operating system software, which is responsible for:

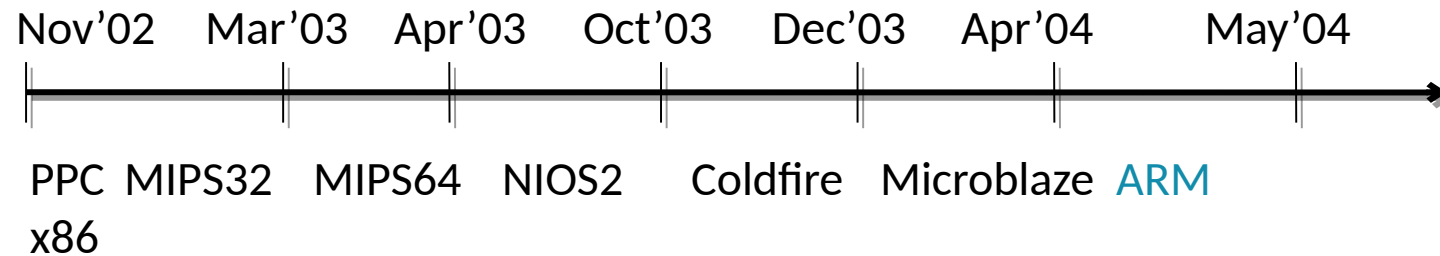
- Setting-up the execution environment for the application
- Starting application execution



The U-Boot Bootloader

Very popular bootloader among embedded system developers

Historic perspective



Today, it is the de-facto standard among embedded systems.

The U-Boot Bootloader

U-Boot architecture is made of two halves:

1st half

- Written mostly in Assembly code
- It runs from the CPU on-chip memory (e.g., on-chip static RAM).
- It initializes the CPU RAM memory controller and relocates itself in off-chip RAM Memory.

2nd half

- Written mostly in C code
- It implements a command-line human-machine interface with scripting capabilities.
- It initializes the minimum set of peripherals to load the device tree Blob, the Linux Kernel, and possibly, the Initial RAM disk to RAM Memory.
- It starts the execution of the Linux Kernel.

The U-Boot Bootloader

Processor-dependent files:

- Specific to the CPU that will run U-Boot (e.g. CPU 1, CPU 2, CPU n)

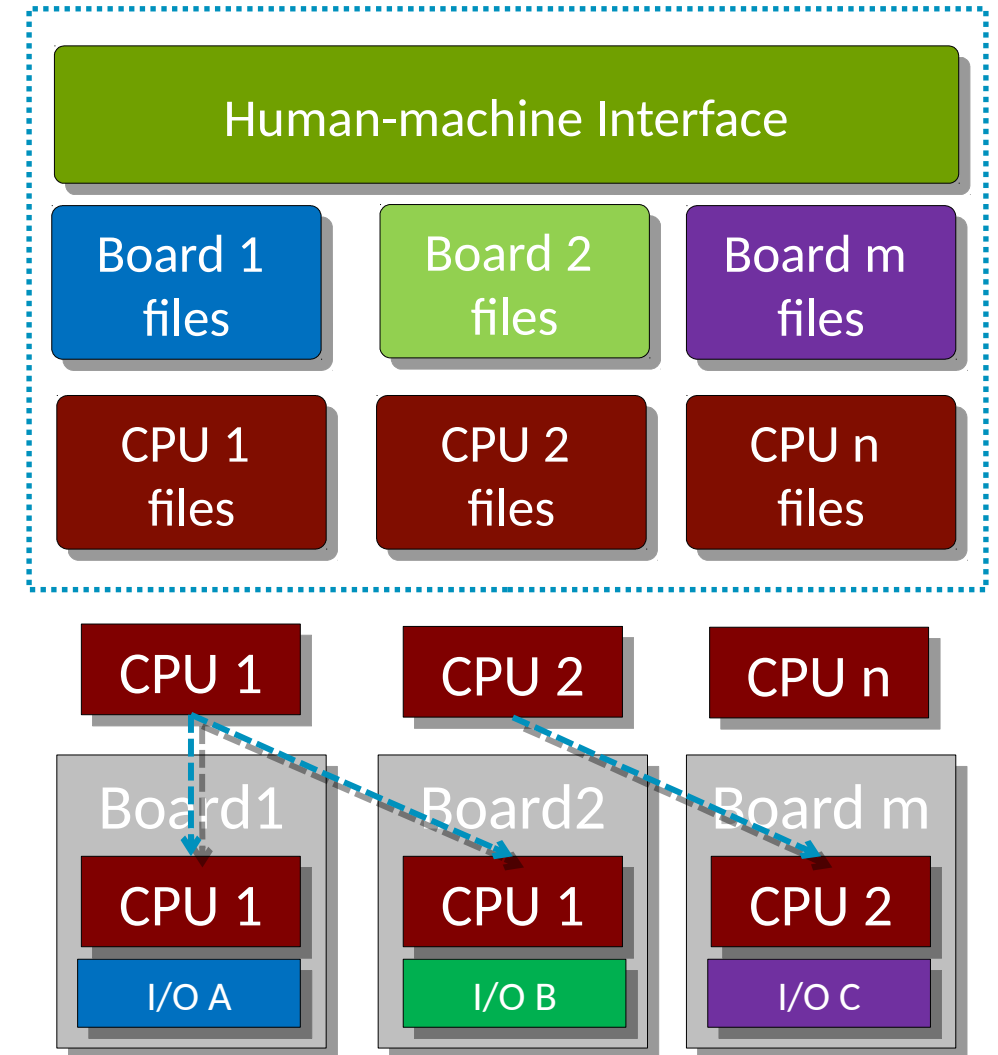
Board-dependent files:

- Specific for the boards hosting the above CPU, which may have different sets of I/O (.e.g, Board 1, hosting CPU 1, and I/O A versus Board 2, hosting CPU 1, and I/O B)

General-purpose files:

- Suitable for all the boards/CPU
- Implement the human-machine interface and the scripting feature of U-Boot

U-Boot source code



Summary

Introduction

Embedded Linux anatomy

- Bootloader

- Linux Kernel

- Root filesystem

- Device tree

Configuration & Build Process of an Embedded Linux System

Linux kernel modules and device drivers

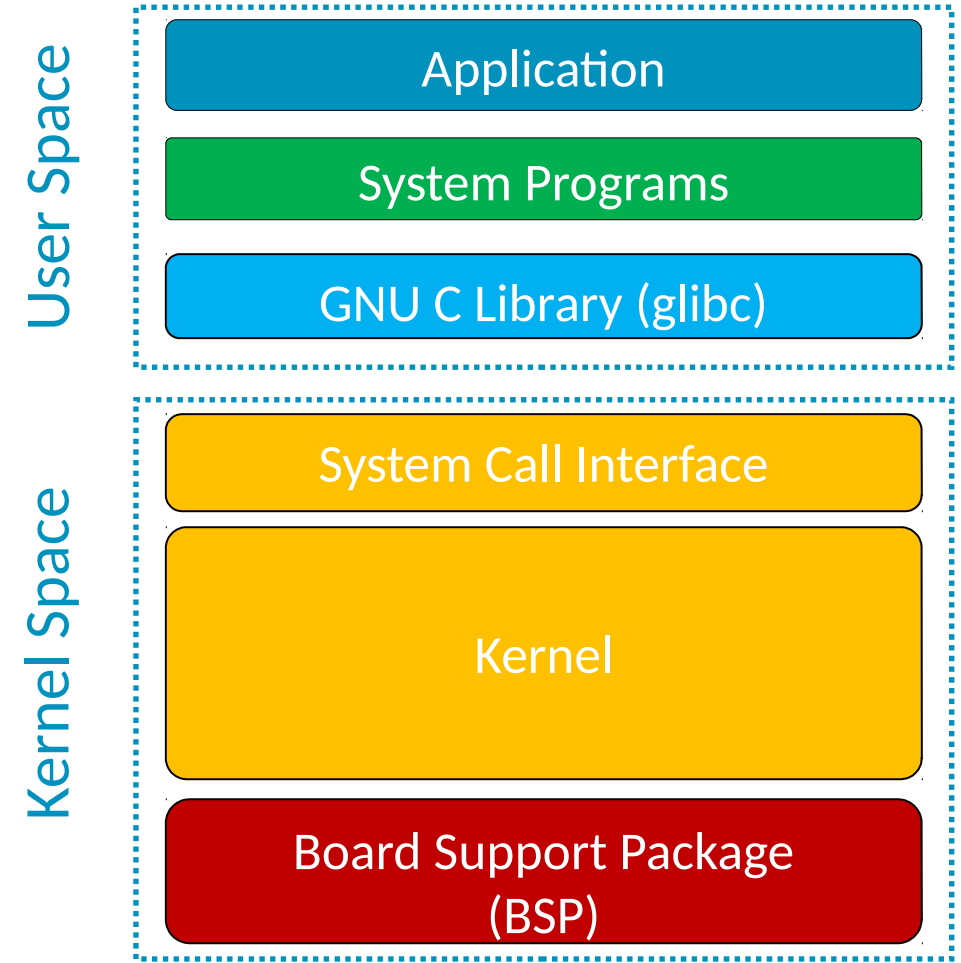
Linux Architecture

Layered architecture based on two levels:

- User space
- Kernel space

User space and kernel space are independent and isolated

User space and kernel space communicate through special purpose functions known as system calls



Linux Architecture

Application

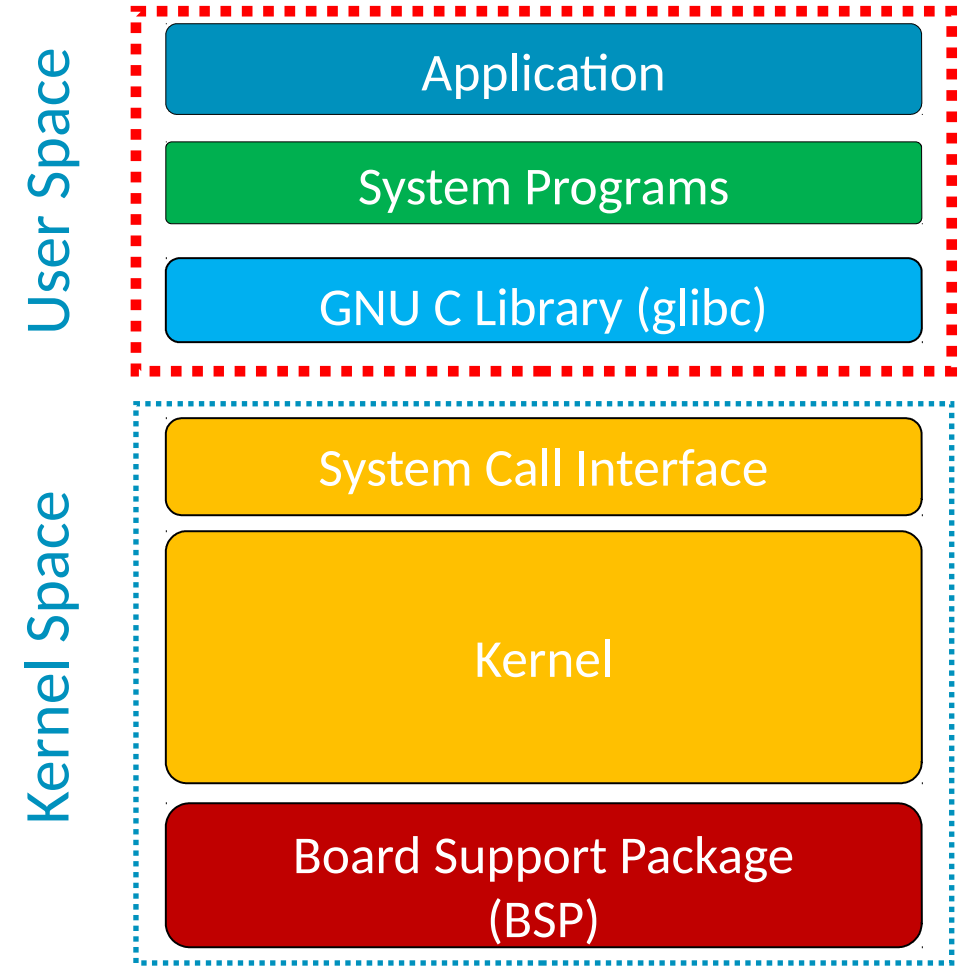
- Software implementing the functionalities to be delivered to the embedded system user

System programs

- User-friendly utilities to access operating system services

GNU C Library (glibc)

- Interface between the User Space and the Kernel Space



Linux Architecture

System call interface

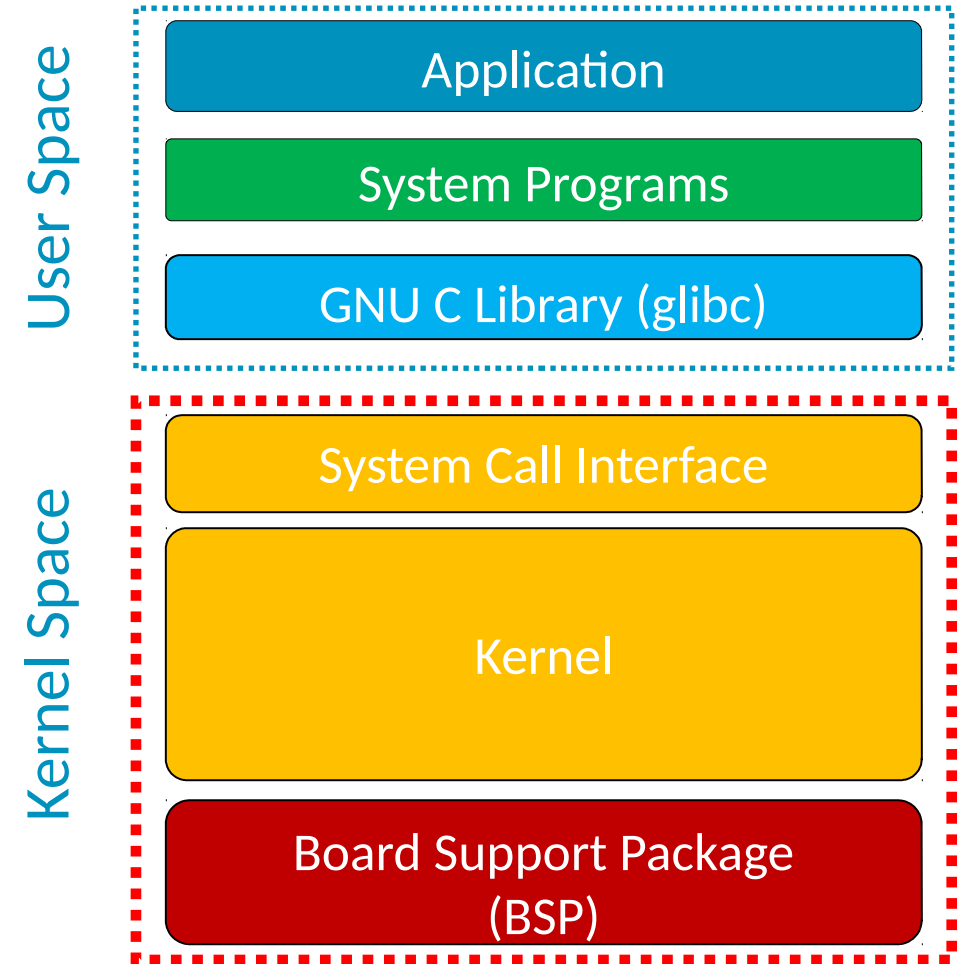
- Entry points to access the services provided by the Kernel (process management, memory management)

Kernel

- Architecture-independent operating system code
- It implements the hardware-agnostic services of the operating system (e.g. the process scheduler).

Board Support Package (BSP)

- Architecture-dependant operating system code
- It implements the hardware specific services of the operating system (e.g. the context switch).



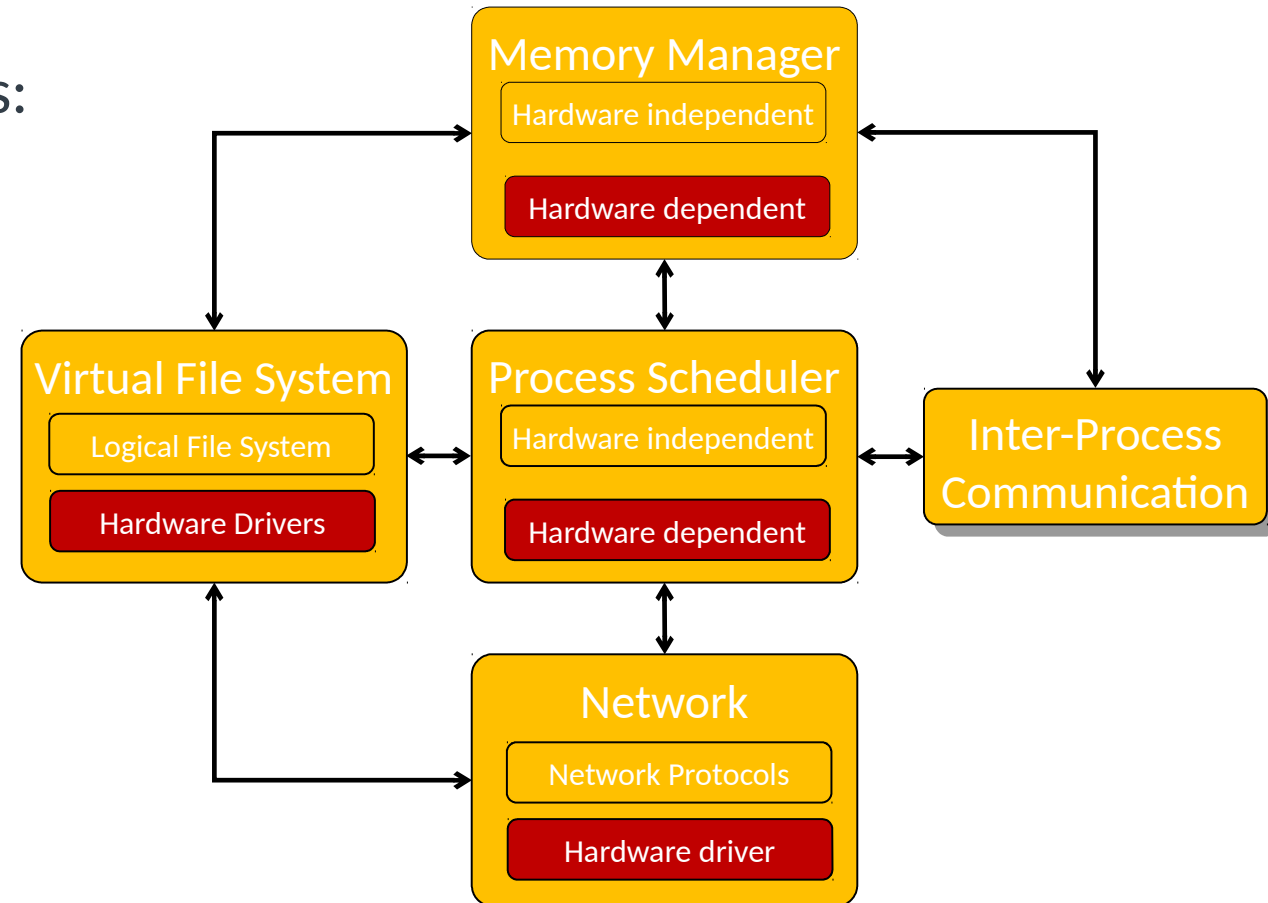
Conceptual View of the Kernel

The Kernel can be divided in five subsystems:

- Process scheduler
- Memory manager
- Virtual file system
- Inter-process communication
- Network

Most of them are composed of:

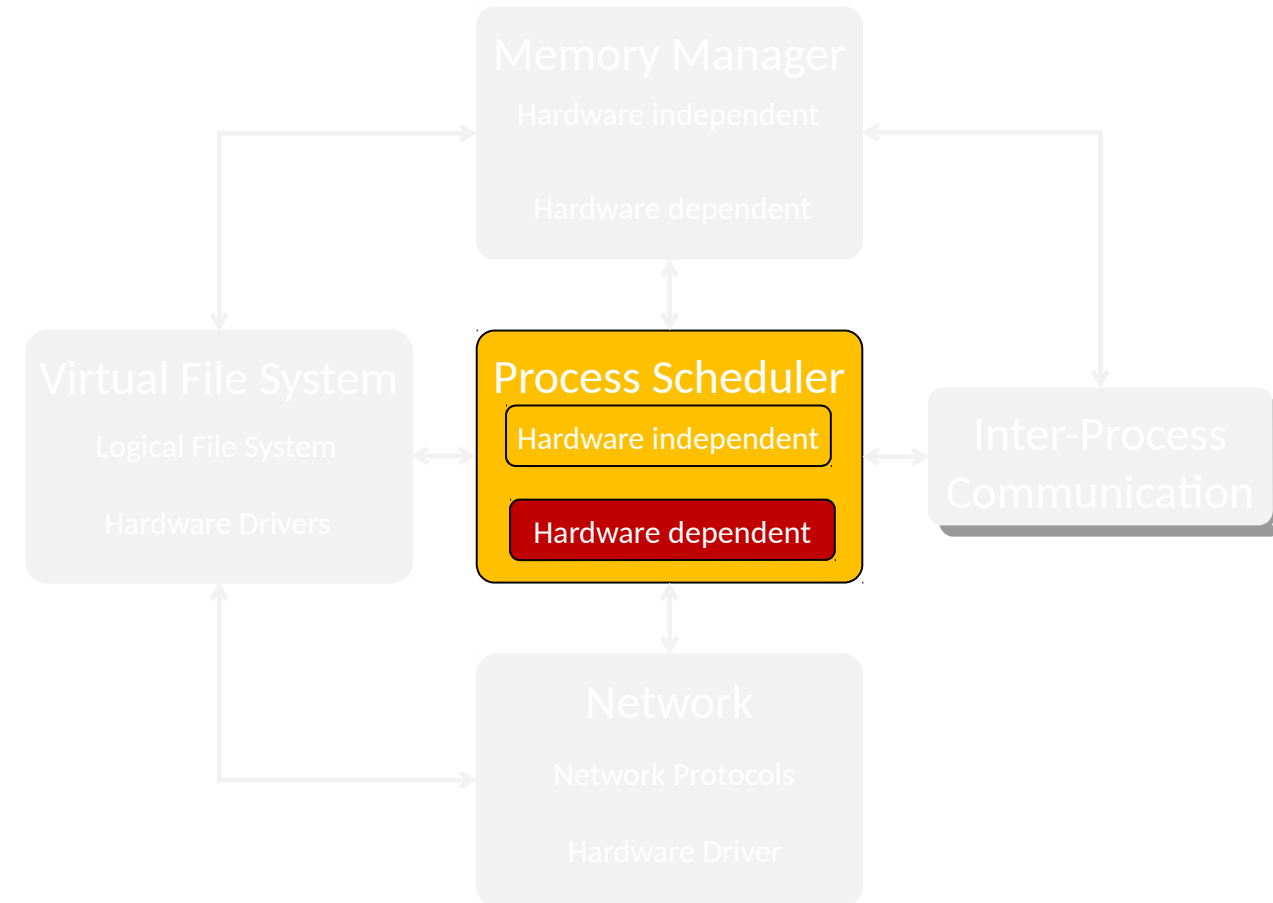
- Hardware-independent code
- Hardware-dependent code



Process Scheduler

Main functions:

- Allows processes to create new copies of themselves
- Implements CPU scheduling policy and context switch
- Receives, interrupts, and routes them to the appropriate Kernel subsystem
- Sends signals to user processes
- Manages the hardware timer
- Cleans up process resources when a processes finishes executing
- Provides support for loadable Kernel modules



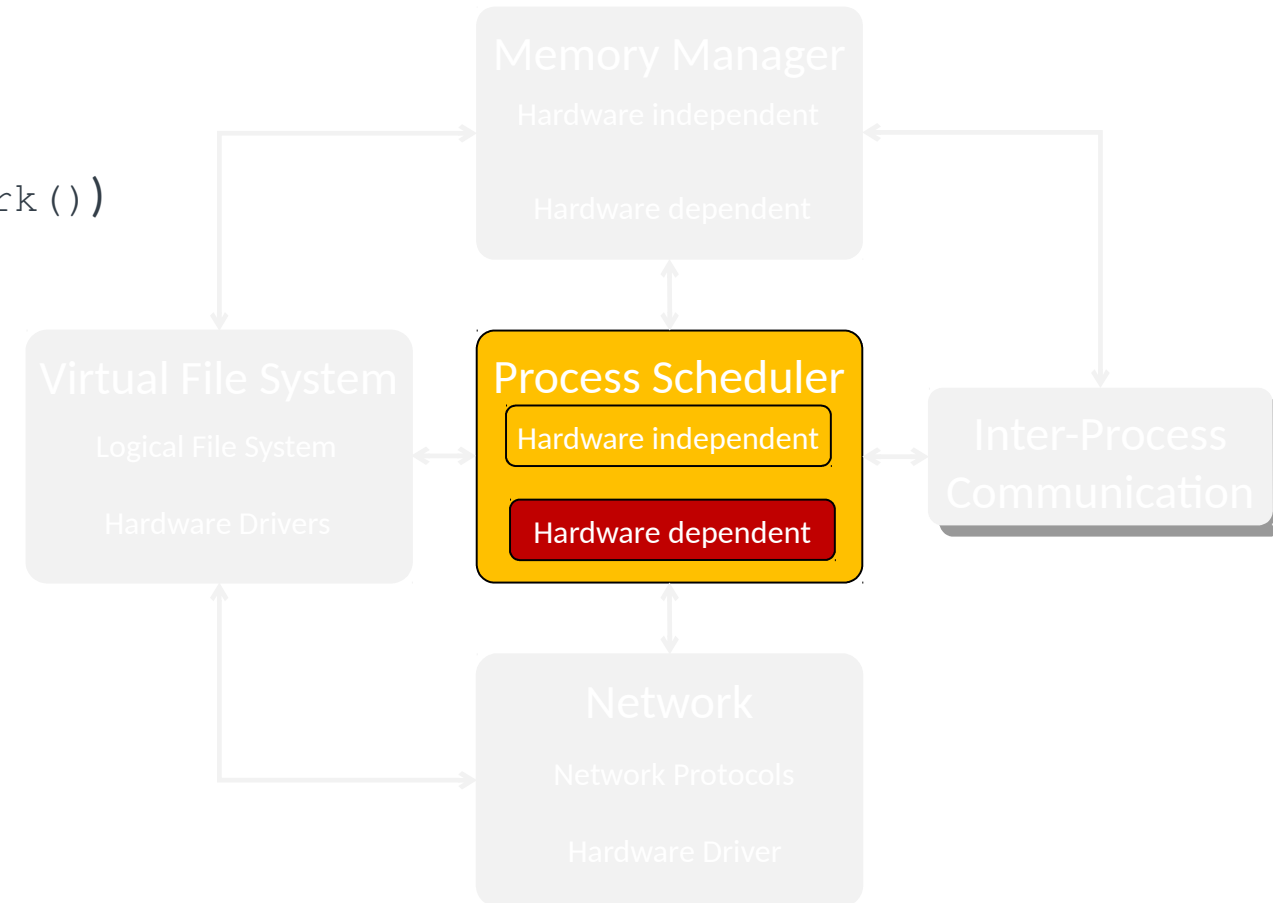
Process Scheduler

External interface:

- System calls interface towards the user space (e.g. `fork()`)
- Intra-Kernel interface towards the kernel space (e.g. `create_module()`)

Scheduler tick:

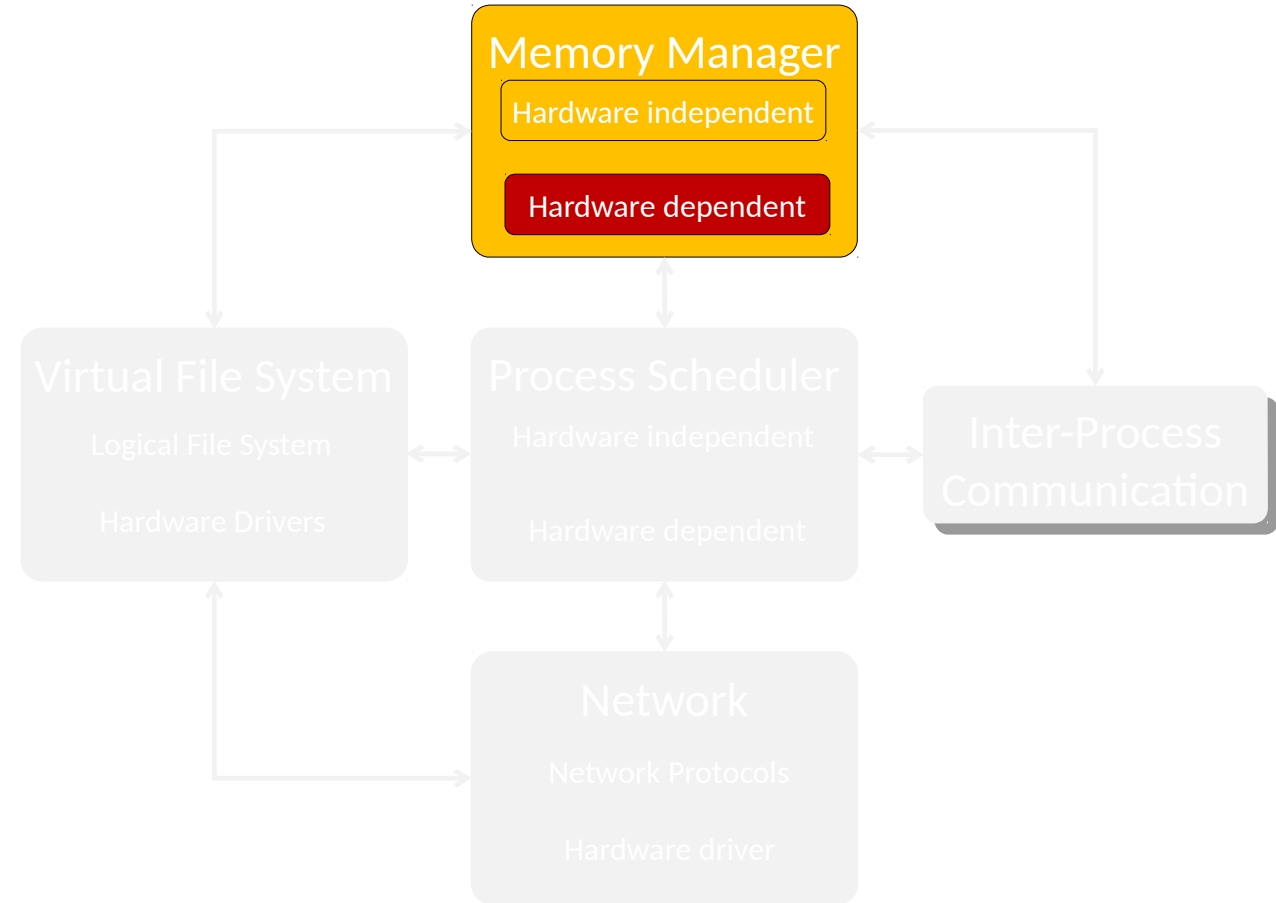
- Directly from system calls (e.g. `sleep()`)
- Indirectly after every system call
- After every slow interrupt



Memory Manager

It is responsible for handling:

- **Large address space:** user processes can reference more RAM memory than what exists physically
- **Protection:** the memory for a process is private and cannot be read or modified by another process; also, the memory manager prevents processes from overwriting code and read-only-data.
- **Memory mapping:** processes can map a file into an area of virtual memory and access the file as memory.



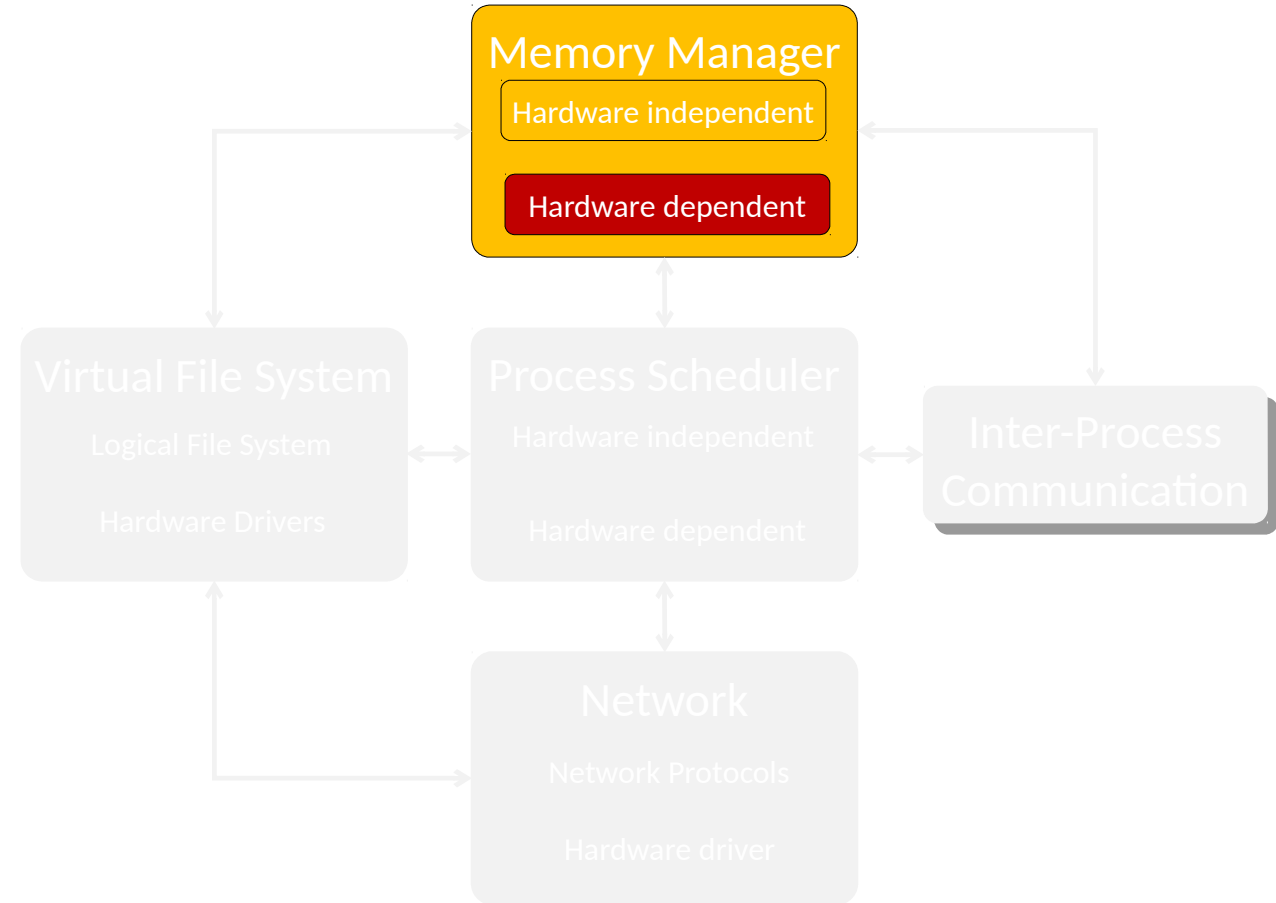
Memory Manager

It uses the Memory Management Unit (MMU) to map virtual addresses to physical addresses.

- It is conventional for a Linux system to have a form of MMU support.

Advantages:

- Processes can be moved among physical memory maintaining the same virtual addresses.
- The same physical memory may be shared among different processes.

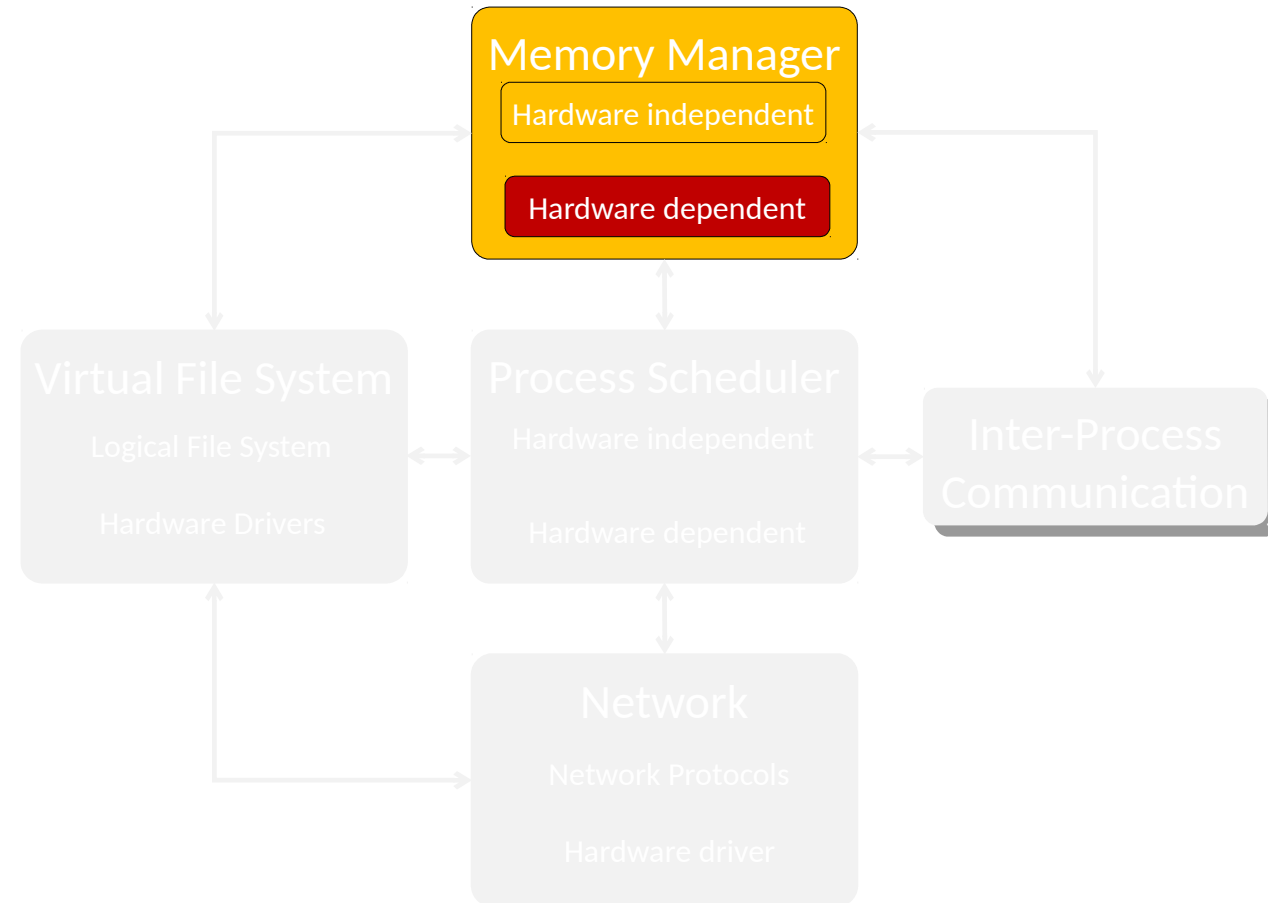


Memory Manager

The MMU detects when a user process accesses a memory address that is not currently mapped to a physical memory location.

The MMU notifies the Linux Kernel the event known as **page fault**.

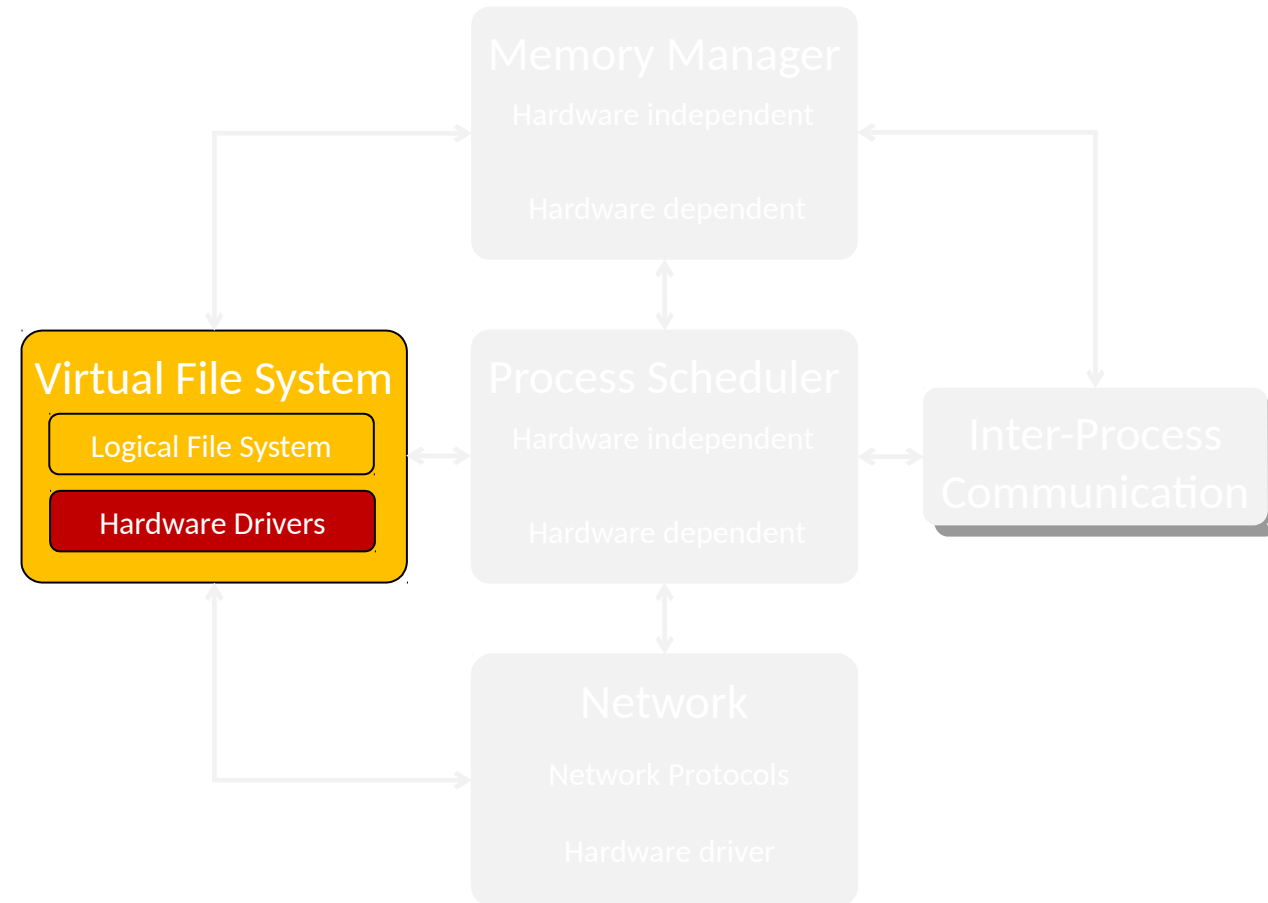
The memory manager subsystem resolves the page fault.



Virtual File System

It is responsible for handling:

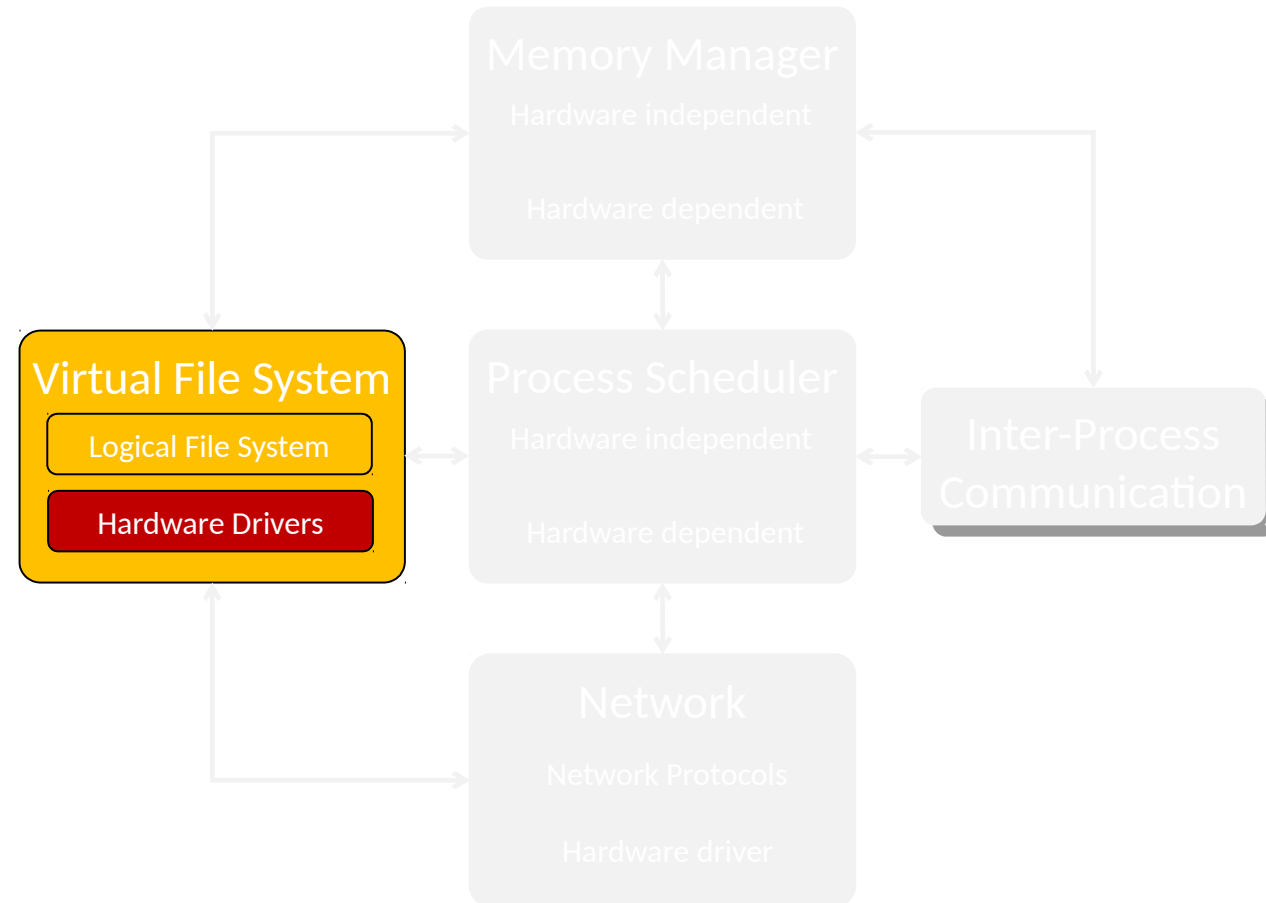
- **Multiple hardware devices:** it provides uniform access to hardware devices.
- **Multiple logical file systems:** it supports many different logical organizations of information on storage media.
- **Multiple executable formats:** it supports different executable file formats (e.g. a.out, ELF).
- **Homogeneity:** it presents a common interface to all of the logical file systems and all hardware devices.



Virtual File System

External interface:

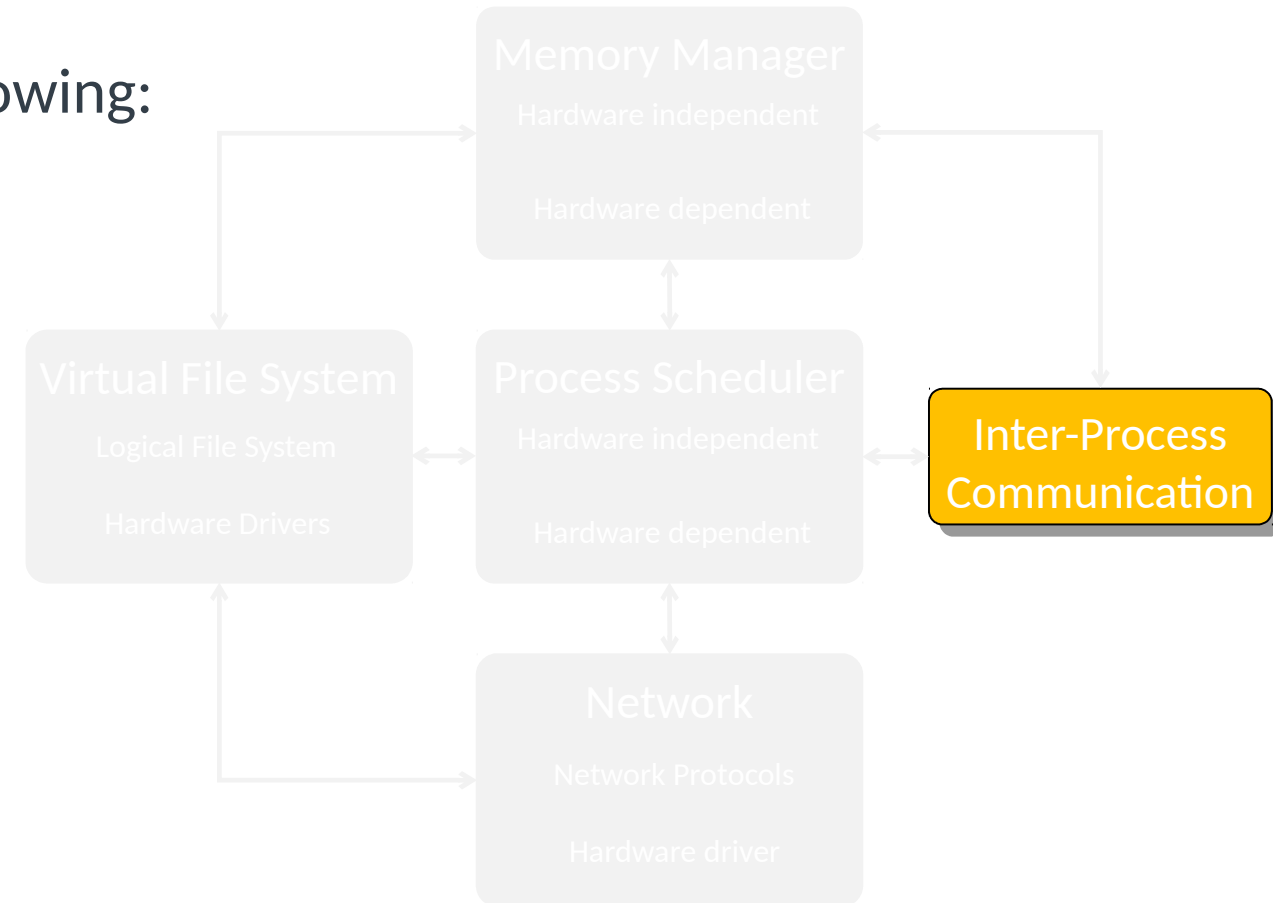
- **System-call interface** based on normal operations on file from the POSIX standard (e.g. open/close/read/write)
- **Intra-kernel interface** based on i-node interface and file interface



Inter-process Communication

It provides mechanisms to processes for allowing:

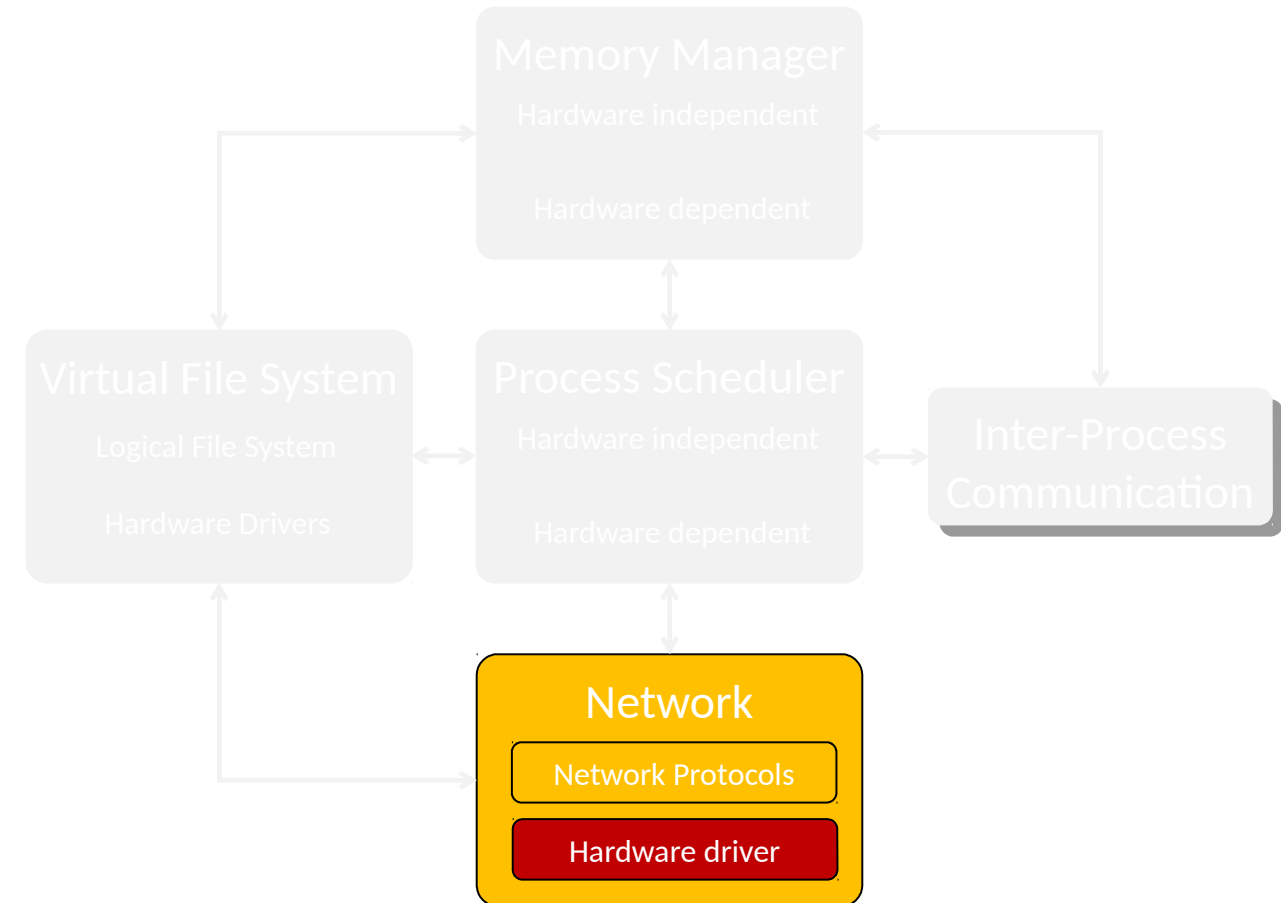
- Resource sharing
- Synchronization
- Data exchange



Network

Provides support for network connectivity

- It implements network protocols (e.g. TCP/IP) through hardware-independent code.
- It implements network card drivers through hardware-specific code.



Summary

Introduction

Embedded Linux anatomy

- Bootloader

- Linux Kernel

- Root filesystem

- Device tree

Configuration & Build Process of an Embedded Linux System

Linux kernel modules and device drivers

Root Filesystem

The Linux Kernel needs a file system, called a **root filesystem**, at startup.

- It contains the configuration file needed to prepare the execution environment for the application (e.g. setting up the Ethernet address).
- It contains the first user-level process (init).

The root filesystem can be:

- A portion of the RAM treated as a file system known as **Initial RAM Disk** (initrd), if the embedded system does not need to store data persistently during its operations
- A **persistent storage in the embedded system**, if the embedded system has to store data persistently during its operations
- A **persistent storage accessed over the network**, if developing a Linux-based embedded system

Typical Layout of the Root Filesystem

```
/          # Disk root
/bin       # Repository for binary files
/lib       # Repository for library files
/dev       # Repository for device files
    console c 5 1    # Console device file
    null c 1 3    # Null device file
    zero c 1 5    # All-zero device file
    tty c 5 0    # Serial console device file
    tty0 c 4 0    # Serial terminal device file
    tty1 c 4 1    #
    tty2 c 4 2    #
    tty3 c 4 3    #
    tty4 c 4 4    #
    tty5 c 4 5    #
/etc       # Repository for config files
    inittab      # The inittab
    /init.d      # Repository for init config files
        rcS      # The script run at sysinit
/proc      # The /proc file system
/sbin      # Repository for accessory binary files
/tmp       # Repository for temporary files
/var       # Repository for optional config files
/usr       # Repository for user files
/sys       # Repository for system service files
/media     # Mount point for removable storage
```

Summary

Introduction

Embedded Linux anatomy

- Bootloader

- Linux Kernel

- Root filesystem

- Device tree

Configuration & Build Process of an Embedded Linux System

Linux kernel modules and device drivers

Device Trees

To manage hardware resources, the Kernel must know which resources are available in the embedded system (i.e. the [hardware description](#): I/O devices, memory, etc).

There are two ways to provide this information to the Kernel:

- [Hardcode](#) it into the Kernel binary code. Each modification to the hardware definition requires recompiling the source code.
- Provide it to the Kernel when the bootloader uses a binary file, the [device tree blob](#).

A device tree blob (DTB) file is produced from a device tree source (DTS).

- A hardware definition can be changed more easily as only DTS recompilation is needed.
- Kernel recompilation is not needed upon changes to the hardware definition. This is a big time saver.

Device Trees

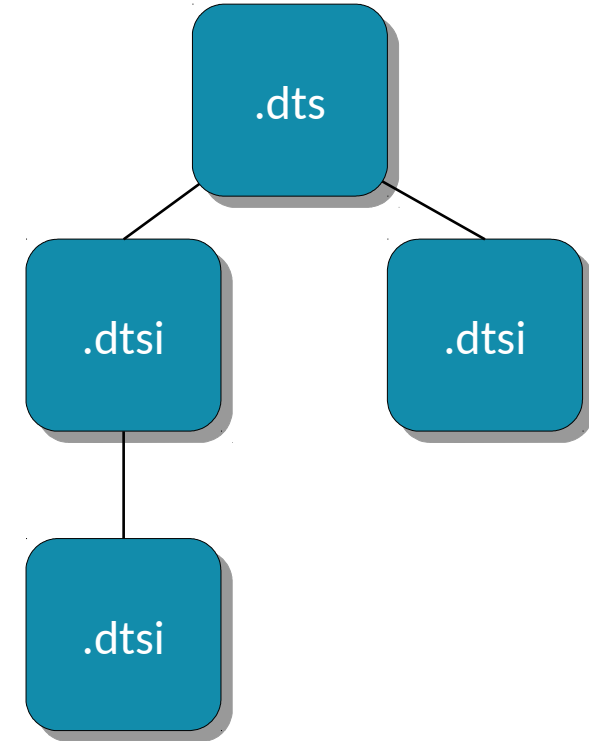
In Arm architecture, all device tree source files are now located in either `arch/arm/boot/dts` or `arch/arm64/boot/dts`.

- `.dts` files for board-level definitions
- `.dtsi` files for included files

A tool, the device tree compiler, compiles the source into a binary form: the **device tree blob** (DTB).

- The DTB is loaded by the bootloader and parsed by the kernel at boot time.

Device tree files are not monolithic. They can be split in several files, including each other.



Device Tree Syntax

```
/ {  
  node0: node@0 {  
    a-string-property = "A string";  
    a-string-list-property = "string ", "string 2";  
    a-byte-data-property = [0x12 0x23 0x45 0x56];  
  
    child-node@0 {  
      a-reference-to-something = <&node1>;  
    };  
  
    child-node@1 {  
    };  
  };  
  node1: node@1 {  
    an-empty-property;  
    a-cell-property = <1 2 3 4>;  
  
    child-node@0 {  
    };  
  };  
};
```

Node label

Node name

Property value as string

Property name

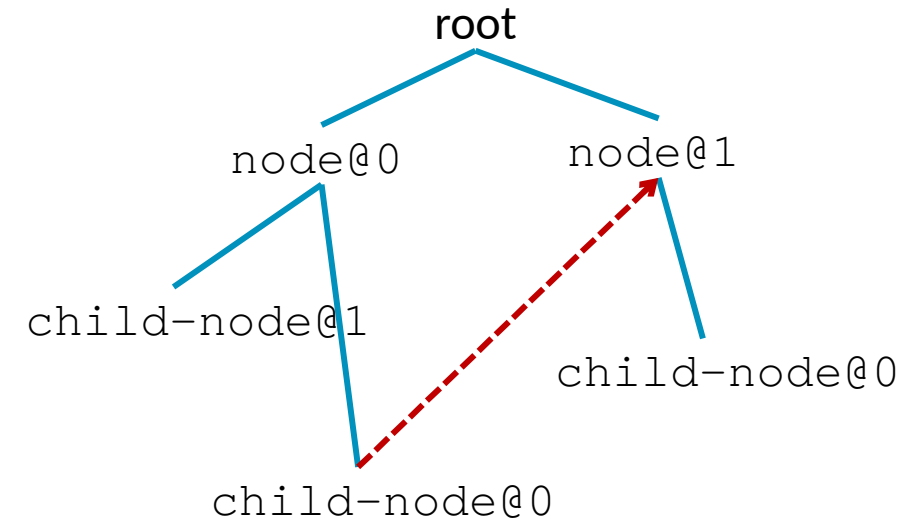
Property value as sequence of bytes

Reference to another node

Address of the node

Property value as sequence of 32-bit integers

Representation of the device tree



Device Tree Content

Under the root of the device tree, we can find:

A `cpus` node, which sub nodes describe each CPU in the system

A `memory` node, which defines the location and size of the RAM

A `chosen` node, which is used to pass parameters to kernel (the kernel command line) at boot time

An `aliases` node, to define shortcuts to certain nodes

One or more nodes defining the buses in the SoC

One or more nodes defining on-board devices

```
/ {  
    alias {};  
  
    cpus {};  
  
    apb@80000000 {  
        apbh@80000000 {  
            /* some devices */  
        };  
        apbx@80040000 {  
            /* some devices */  
        };  
    };  
  
    chosen {  
        bootargs = "root=/dev/nfs";  
    };  
};
```

Device Tree Addressing

The following properties are used:

- `reg = <address1 length1 [...] >`, which lists the address sets (each defined as starting address, length) assigned to the node
- `#address-cells = <num of addresses>`, which states the number of address sets for the node
- `#size-cells=<num of size cells>`, which states the number of size for each set

Note:

- Every node in the tree that represents a device is required to have the `compatible` property.
- `compatible` is the key Linux uses to decide which device driver to bind to a device.

```
/ {  
    cpus {  
        #address-cells = <1>;  
        #size-cells = <0>;  
        cpu@0 {  
            compatible = "arm,cortex-a9";  
            reg = <0>;  
        };  
        cpu@1 {  
            compatible = "arm,cortex-a9";  
            reg = <1>;  
        };  
    };  
};
```

Device Tree Addressing

CPU addressing

- Each CPU is associated with a unique ID only.
- #size-cells=<0>, always

Memory mapped devices

- Typically defined by one 32-bit based address, and one 32-bit length
- #address-cells=<1>
- #size-cells=<1>

```
cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    cpu@0 {
        compatible = "arm,cortex-a9";
        reg = <0>;
    };
};

/{
    #address-cells = <1>;
    #size-cells = <1>;
    gpio1: gpio@0209c000 {
        compatible = "fsl,imx6sx-gpio",
                    "fsl,imx35-gpio";
        reg = <0x0209c000 0x4000>;
    };
};
```

Device Tree Addressing

External bus with chip select line

- Typically, `address-cells` uses 2 cells for the address value: one for the chip select number and one for the offset from the base of the chip select.
- `#address-cells=<2>`
- The length field remains as a single cell.
- `#size-cells=<1>`
- The mapping between bus addressing and CPU addressing is defined by the `ranges` property.

```
external-bus {  
    #address-cells = <2>  
    #size-cells = <1>;  
  
    ranges = <0 0 0x10100000 0x10000  
            1 0 0x10160000 0x10000>;  
    ethernet@0,0 {  
        compatible = "smc,smc91c111";  
  
        reg = <0 0 0x1000>;  
    };  
    i2c@1,0 {  
        compatible = "acme,a1234-i2c-bus";  
  
        reg = <1 0 0x1000>;  
        #address-cells = <1>  
        #size-cells = <0>;  
    rtc@58 {  
        compatible = "maxim,ds1338";  
        reg = <58>;  
    };  
};
```

Bus address

Corresponding CPU address and range

Summary

Introduction

Embedded Linux anatomy

Configuration & Build Process of an Embedded Linux System

- The workflow

- The build systems

- Yocto

Linux kernel modules and device drivers

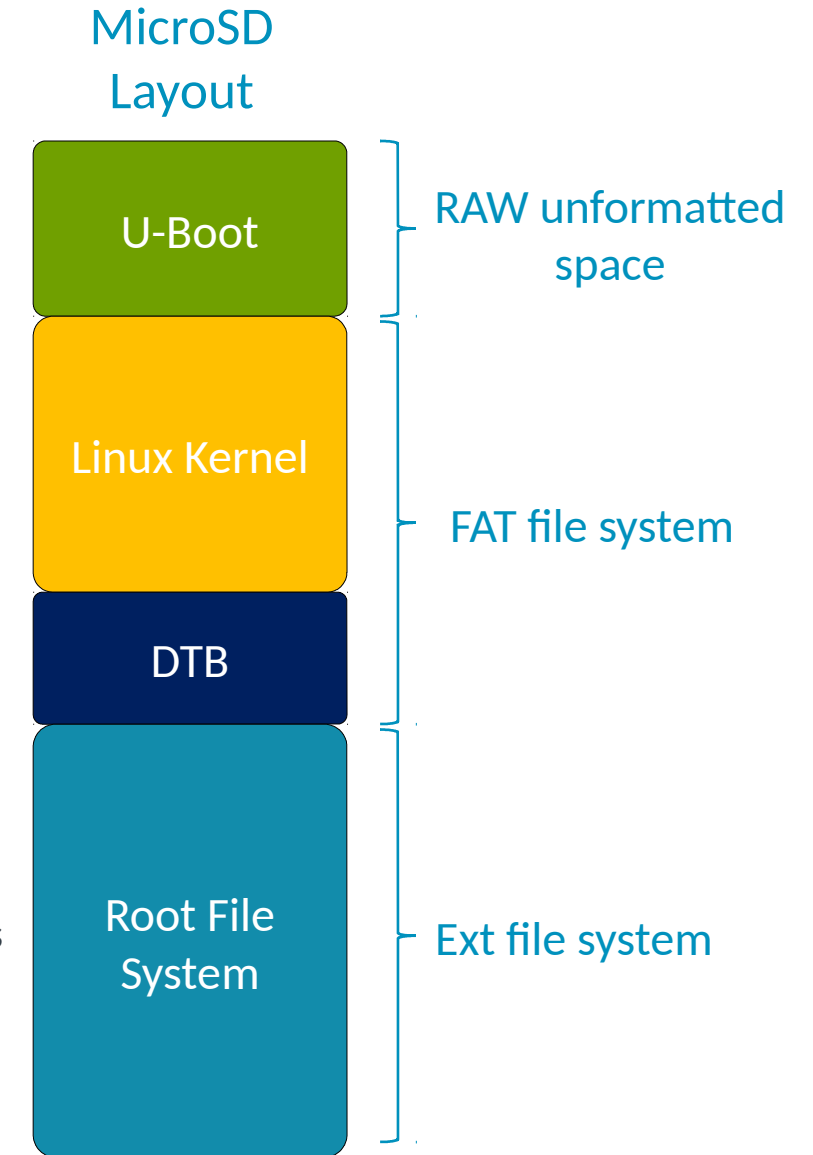
Introduction

An embedded Linux system requires the following components to operate:

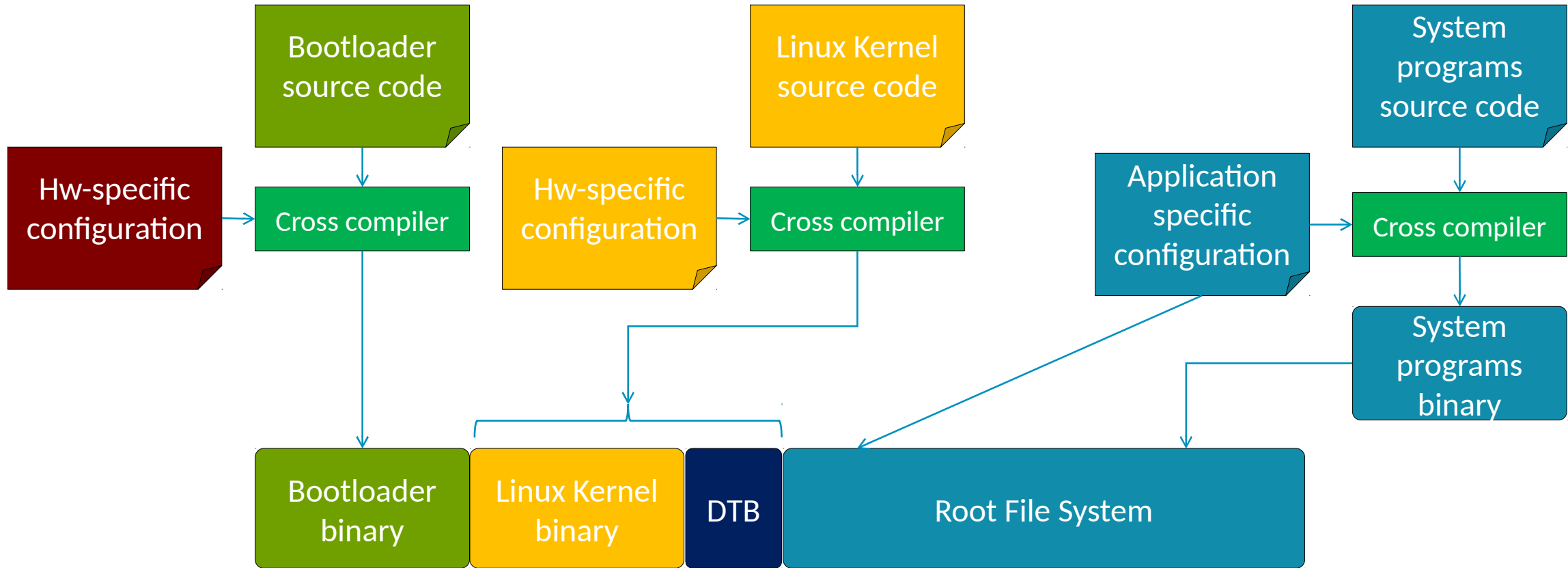
- The bootloader
- The Linux Kernel
- The device tree blob
- The Root File System

All these components shall be:

- Configured for the embedded system hardware platform
- Compiled and linked into an executable format
- Deployed into the embedded system persistent storage for booting and operations



The Workflow



The Workflow

The Bootloader source code shall be procured.

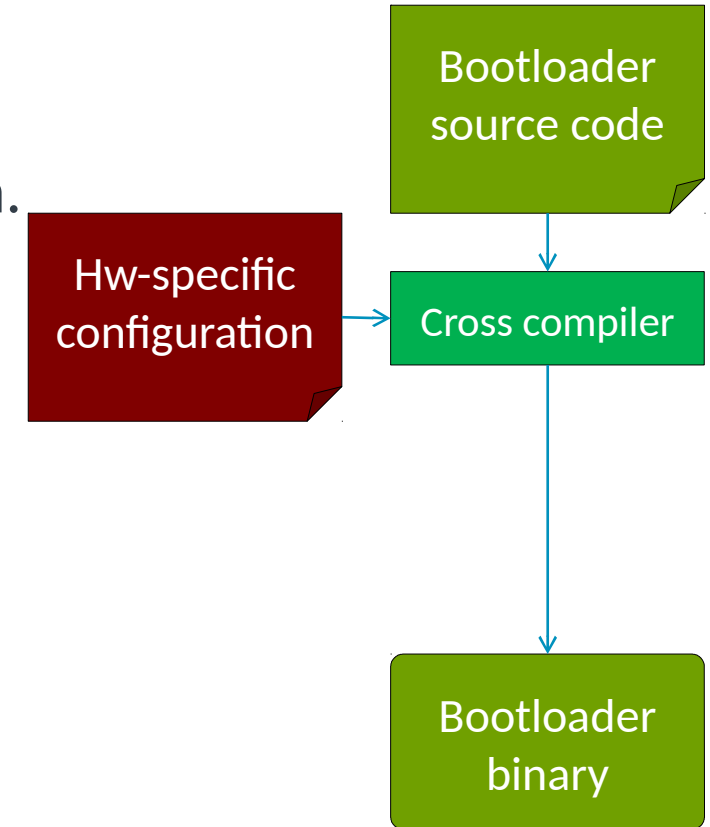
It shall be configured for the specific hw of the embedded system.

- The proper CPU shall be selected.
- The proper board shall be selected.
- Custom configuration may be needed if the hw is non-standard.

It shall be cross-compiled for the CPU of choice obtaining the executable code.

It shall be copied into the boot device.

- First sector of a MicroSD card
- Bootflash device on the embedded system board



The Workflow

The Linux Kernel source code shall be procured.

It shall be configured for the specific hw of the embedded system.

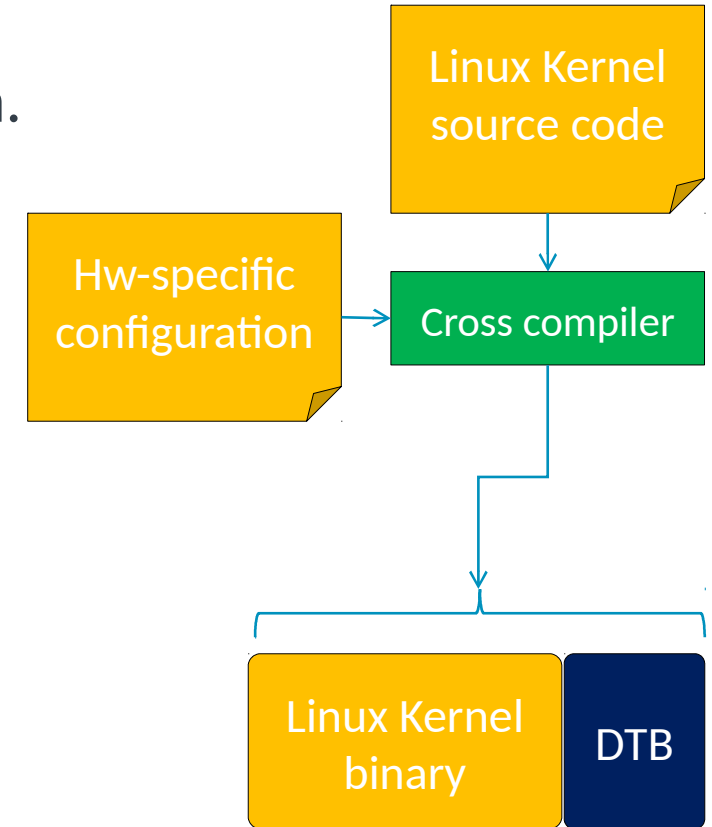
- The proper CPU shall be selected.
- The proper board shall be selected.
- Custom configuration may be needed if the hw is non-standard.

It shall be cross-compiled for obtaining

- Executable Linux kernel
- Executable Linux kernel modules
- Device Tree Blob

The obtained files shall be copied into the boot device.

- Partition on a MicroSD device
- Bootflash device on the embedded system board



The Workflow

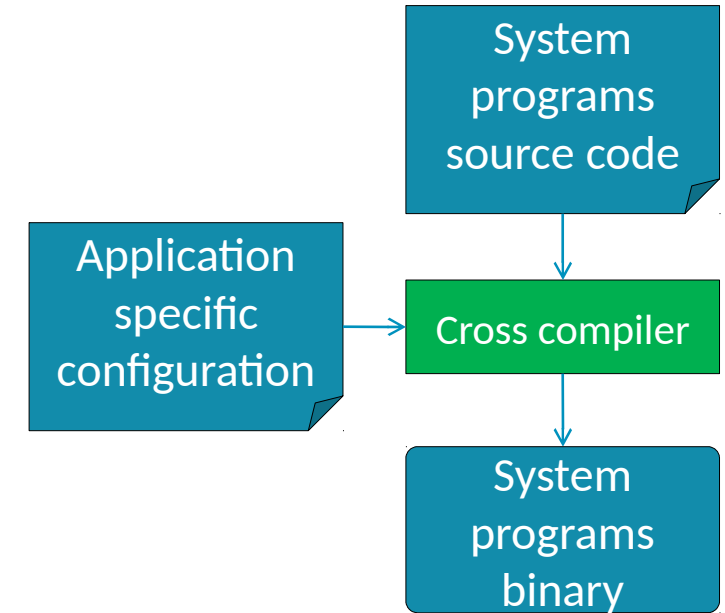
The system programs source code shall be procured.

- Downloaded from Internet
- Received from operating system vendor

They shall be configured for the specific application.

- Only the system programs needed for the application the embedded system is intended for shall be considered.

They shall be cross-compiled obtaining the executable binary.



The Workflow

The root file system shall be prepared.

It typically requires

- To create a file and mount it as a volume on the development host
- To format it using any of the file systems Linux supports (e.g., ext3)
- To create the required directory tree
- To populate it with the needed configuration files
- To populate it with the system program binary

The root file system shall be copied into the embedded system persistent storage.

- Partition on a MicroSD device
- Bootflash device on the embedded system board

```
/                                # Disk root
/bin                            # Repository for binary files
/lib                            # Repository for library files
/dev                            # Repository for device files
    console c 5 1                # Console device file
    null c 1 3                  # Null device file
    zero c 1 5                  # All-zero device file
    tty c 5 0                   # Serial console device file
    tty0 c 4 0                  # Serial terminal device file
    tty1 c 4 1                  #
    tty2 c 4 2                  #
    tty3 c 4 3                  #
    tty4 c 4 4                  #
    tty5 c 4 5                  #
/etc                             # Repository for config files
    inittab                     # The inittab
    /init.d                     # Repository for init config files
        rcS                     # The script run at sysinit
/proc                           # The /proc file system
/sbin                           # Repository for accessory binary files
/tmp                            # Repository for temporary files
/var                            # Repository for optional config files
/usr                            # Repository for user files
/sys                            # Repository for system service files
/media                          # Mount point for removable storage
```

Summary

Introduction

Embedded Linux anatomy

Configuration & Build Process of an Embedded Linux System

- The workflow

- The build systems

- Yocto

Linux kernel modules and device drivers

Build Systems

Building an embedded Linux system is a complex operation.

- Multiple sources shall be configured and compiled.
- Root file system shall be updated at each build through a non-trivial task.
- In cases of multiple hw and multiple hw configurations, manual iteration is needed.

Tools, known as **build systems**, are available to automate such operations.

Build systems takes care of:

- Building the cross compiler for the selected embedded system CPU
- Managing bootloader/kernel/system programs configuration
- Managing bootloader/kernel/system program build
- Preparation of the root file system and boot device image preparation

Build Systems

Several solutions are available.

Hw-vendor custom-built systems

- NXP Linux Target Image Builder (LTIB)

Open-source build systems, among which the most popular are [Yocto](#) and [Buildroot](#)

- Very actively maintained and developed projects
- Widely used in the industry
- Built from scratch from source toolchain, bootloaders, kernel, and root file system

Buildroot vs Yocto: General Aspects

Buildroot

- Focus on simplicity
- Use existing technologies: kconfig, make
- Open community

Yocto

- Provides core recipes and use layers to get support for more packages and more machines
- Custom modifications should stay in a separate layer
- Versatile build system: tries to be as flexible as possible and to handle most use cases
- Open community but governed by the Yocto Project Advisory Board

Buildroot vs Yocto: Configuration

Buildroot reuses kconfig from the Linux kernel

- Entire configuration stored in a single `.config/defconfig`
- Defines all aspects of the system: architecture, kernel version/config, bootloaders, user-space packages, etc.
- Building the same system for different machines to be handled separately

In Yocto the configuration is separated in multiple parts:

- **Distribution** configuration (general configuration, toolchain selection, etc.)
- **Machine** configuration (defines the hw architecture, hw features, BSP)
- **Image** recipe (what system programs should be installed on the target)
- Local configuration (e.g., how many threads to use when compiling, whether to remove build artifacts, etc.)
- Allows to build the same image for different machines or using different distributions or different images for one machine

Buildroot vs Yocto: Purpose

Buildroot is intended for

- Very small root file systems (< 8 MB)
- Simple embedded system (with limited number of system programs)
- Non-dedicated build engineers (e.g., engineers that are not focused only in building embedded Linux)



Yocto is intended for

- Large root file systems
- Large embedded systems
- Support for multiple hw configurations
- Dedicated build engineers



Summary

Introduction

Embedded Linux anatomy

Configuration & Build Process of an Embedded Linux System

- The workflow

- The build systems

- Yocto

Linux kernel modules and device drivers

The Yocto Project

Open-source project hosted by the Linux Foundation

Collaboration of multiple projects that make up the “Yocto Project”

- **Bitbake:** build tool
- **OpenEmbedded core:** software framework used for creating Linux distributions
- **Poky:** a reference distribution of the Yocto Project, containing the OpenEmbedded Build System (BitBake and OpenEmbedded Core) and a set of metadata to start building custom embedded Linux systems
- **Application Development Toolkit:** provides application developer a way to write sw running on the custom-built embedded Linux system without the need for knowing build systems

Support for Arm, PPC, MIPS, and x86

The Yocto Build System

It is composed of multiple **layers** which are containers for the building blocks of the system.

Layers do not contain components source code, only their metadata, called **recipes**.

Recipes define how to build binary outputs called **packages**.

Developer-specific layer

Commercial layer from
operating system vendor

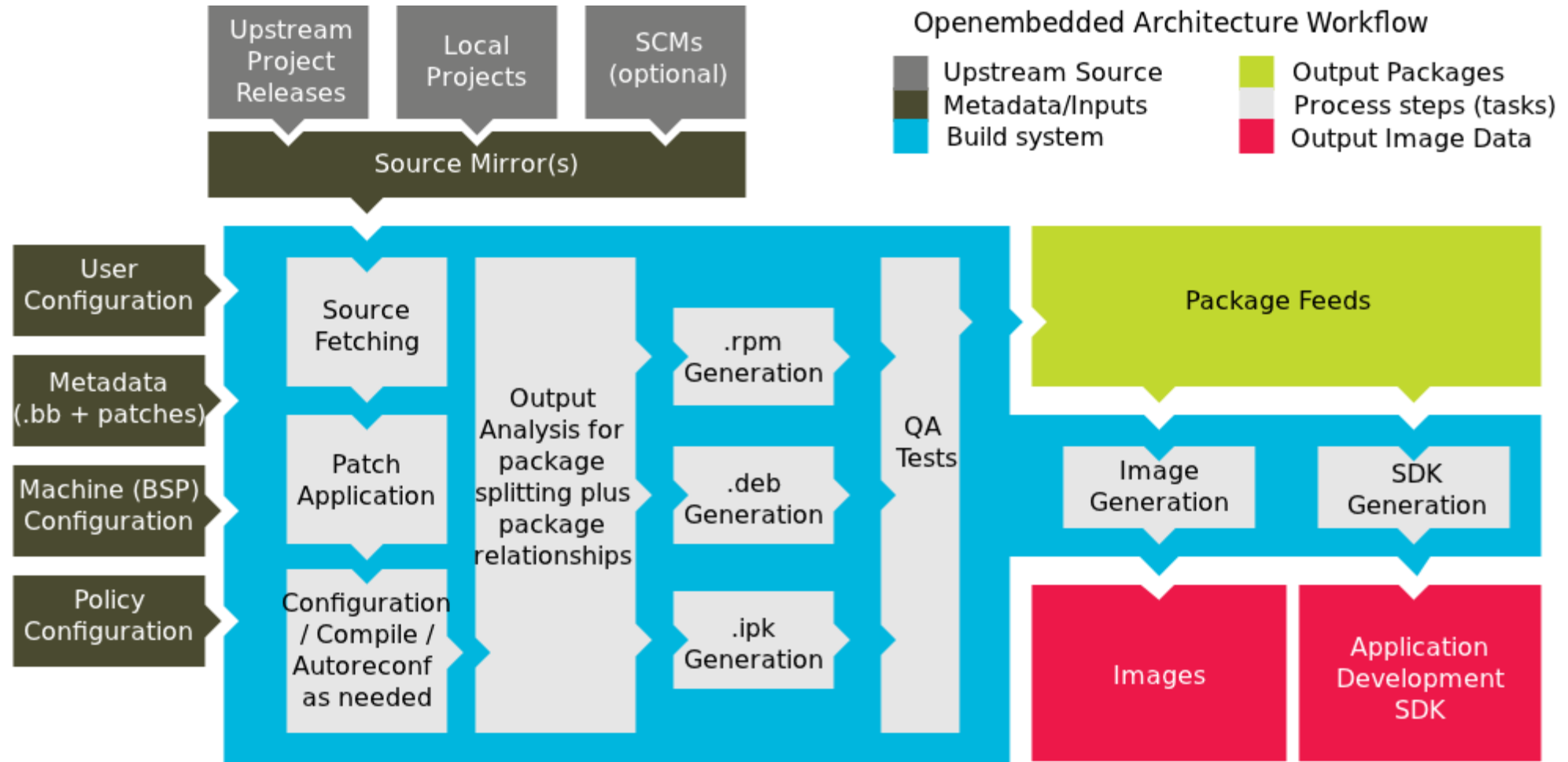
User interface-specific layer

Board Support Package
layer

Yocto-specific layer
metadata (meta-yocto)

OpenEmbedded core
metadata (oe-core)

The Build System Workflow



Source: <http://www.yoctoproject.org/docs/2.1/mega-manual/mega-manual.html>

Summary

Introduction

Embedded Linux anatomy

Configuration & Build Process of an Embedded Linux System

Linux kernel modules and device drivers

Introduction

CPU – I/O interface

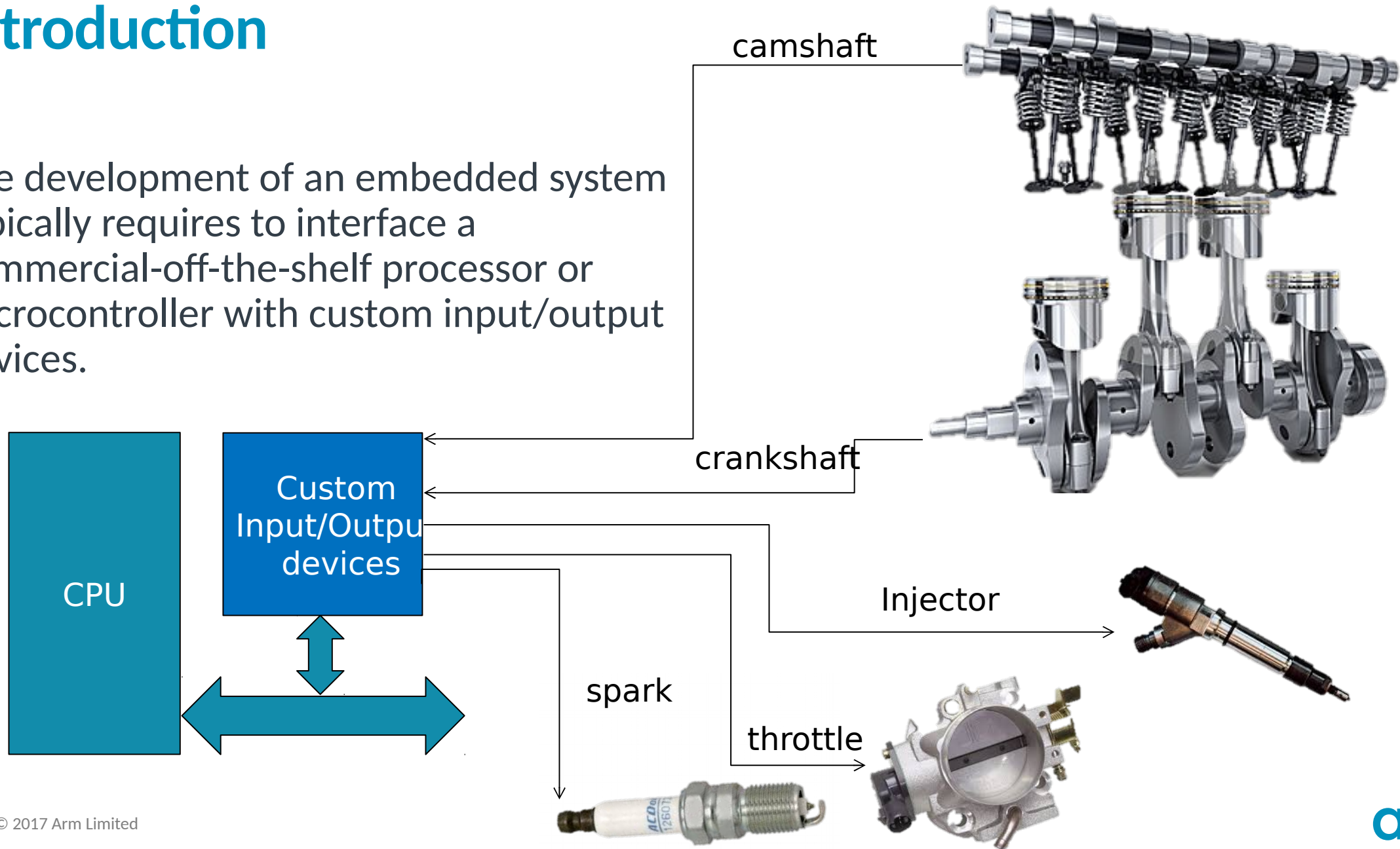
Virtual File System abstraction

Linux Kernel modules

Case study

Introduction

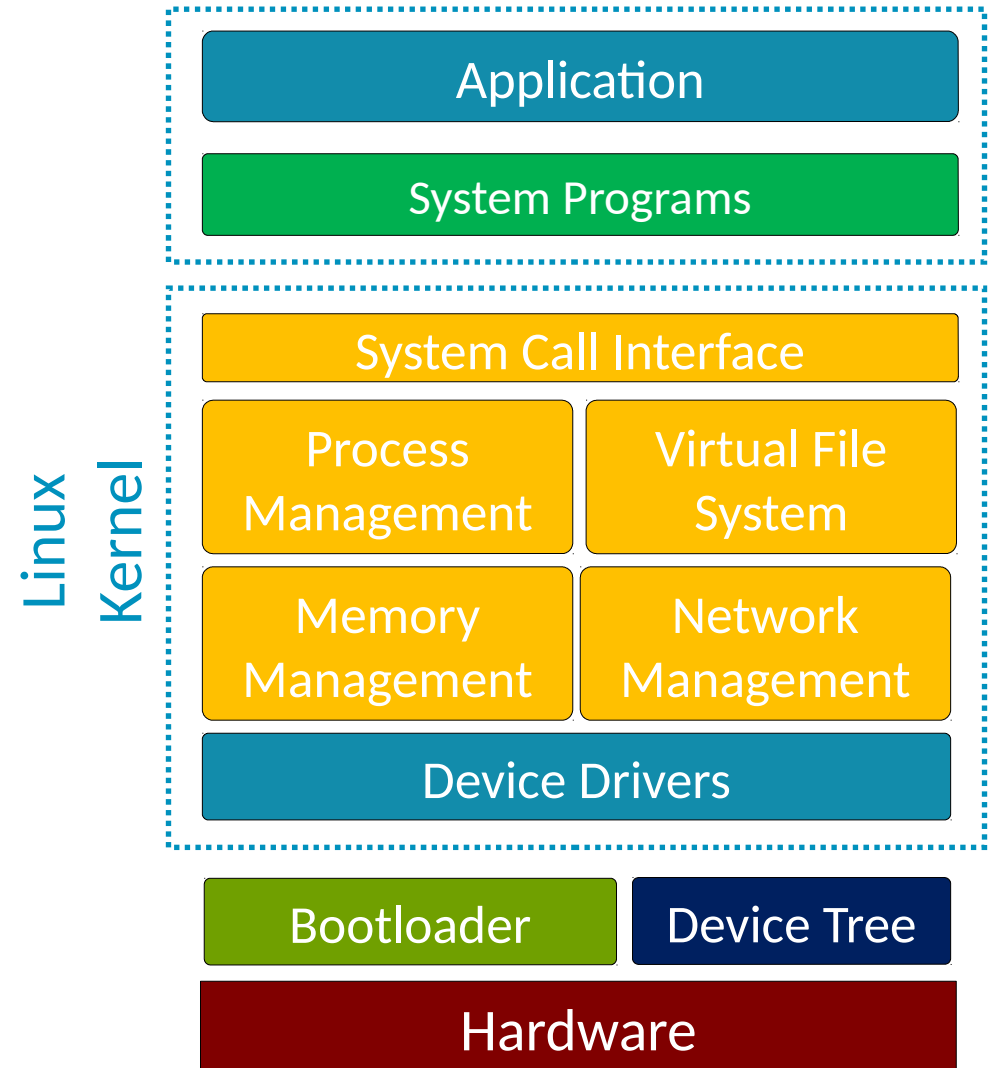
The development of an embedded system typically requires to interface a commercial-off-the-shelf processor or microcontroller with custom input/output devices.



Introduction

From the software point of view interfacing with custom input/output (I/O) devices requires:

- A **user-space application** that reads/writes data from/to a suitable abstraction interface of the hardware
- A **device driver** that translates the operations of the abstraction interface into hardware-specific operations



Summary

Introduction

Embedded Linux anatomy

Configuration & Build Process of an Embedded Linux System

Linux kernel modules and device drivers

Introduction

CPU – I/O interface

Virtual File System abstraction

Linux Kernel modules

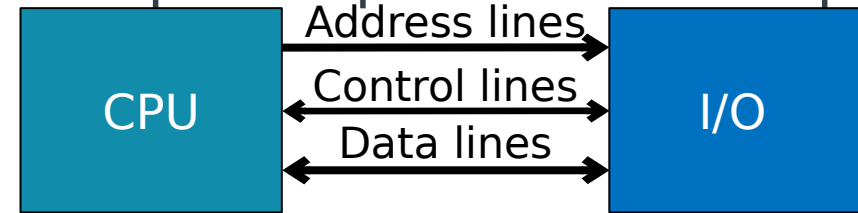
Case study

CPU – I/O Interface

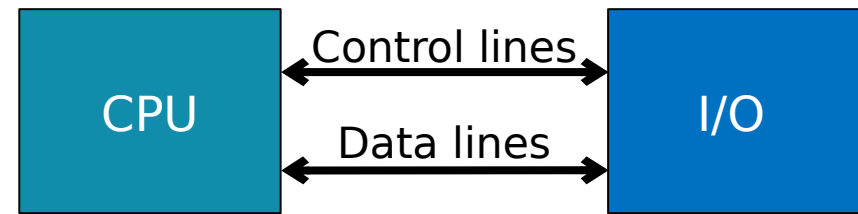
The interconnection between CPU and I/O device can be broadly classified as:

- **Parallel**, where N independent lines connect the CPU with the I/O, and one word (e.g., 8-/16-/32-bits) or blocks of words are transferred at each operation
- **Serial**, where M ($M \ll N$) lines connect the CPU with the I/O, and one bit is transferred at each operation according to a serial communication protocol (such as SPI, I2C, USB)

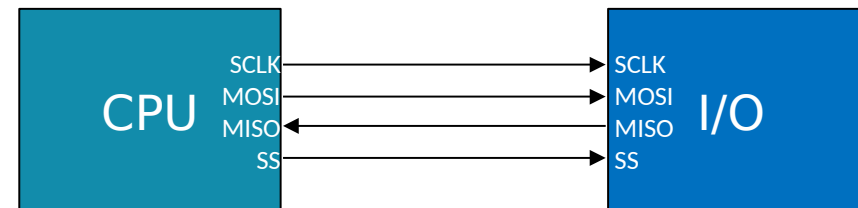
Example of a parallel non-multiplexed bus



Example of a parallel multiplexed bus



Example of a serial bus (SPI)



CPU – I/O Interface

CPU and I/O operate asynchronously:

- CPU runs software.
- I/O performs its own tasks.

When the CPU has to read/write from/to the I/O, a read/write operation is performed.

- Read/write operations are always initiated by the CPU.

How is it possible for the CPU to recognize that an I/O has data ready to be read?

Two techniques are possible:

- **Polling**: The CPU checks the peripheral periodically.
- **Interrupt**: The peripheral asks the CPU's attention when needed.

CPU – I/O Interface with Polling

The software periodically reads the status of each peripheral.

In case the I/O needs to be serviced by the CPU, the corresponding program (**service routine**) is executed.

Advantage

- Simple to implement

Disadvantages

- High latency as I/O are polled serially (time when the I/O is served – time when the peripheral needs service)
- CPU time is wasted when polled devices are not to be serviced.

Example

- Device A provides 1 byte.
- Device B provides 2 bytes.

```
while(1)
{
    if(A_is_ready())
    {
        resA = read_byte_from_A();
    }
    if(B_is_ready())
    {
        resB_1 = read_byte_from_B();
        resB_2 = read_byte_from_B();
    }
    } Device A provides 1 byte.
    Device B provides 2 bytes.
```

CPU – I/O Interface with Interrupt

A dedicated line (interrupt request, IRQ) connects the I/O with the CPU.

In case the I/O needs to be serviced, it asserts the IRQ line.

At the end of each instruction, the CPU checks for the IRQ line; if asserted, it runs the corresponding service routing.

Advantage

- Low latency (time when the peripheral is served – time when the peripheral needs service)

Disadvantage

- Higher hardware complexity

Example

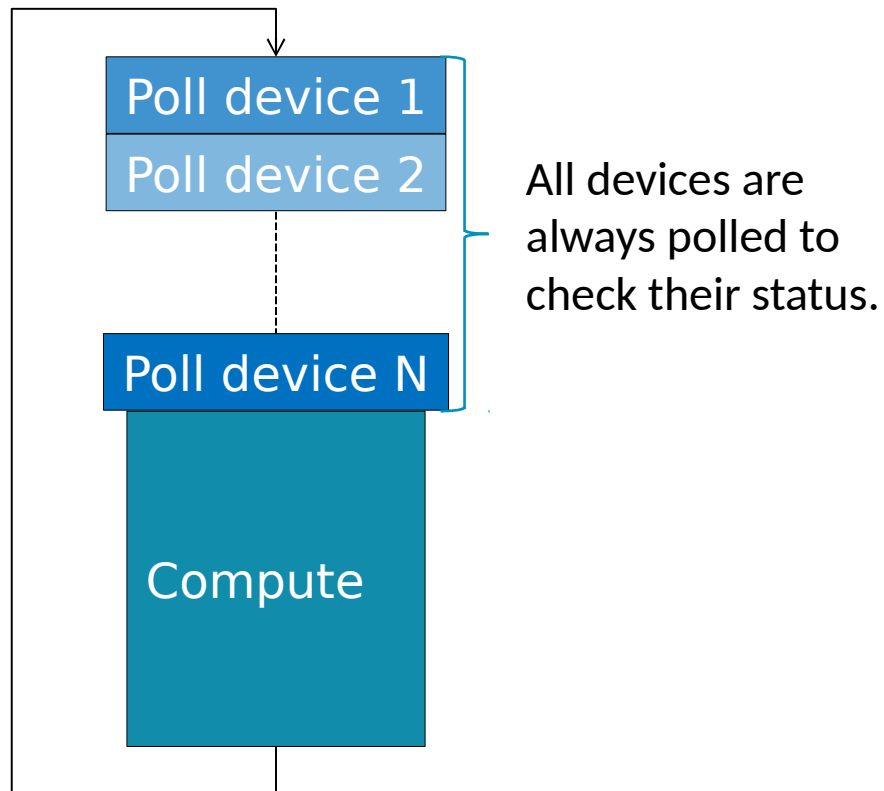
- Device A provides 1 byte (IRQ1)
- Device B provides 2 bytes (IRQ2)

```
IRQ1 ()
{
    resA = read_byte_from_A();
}

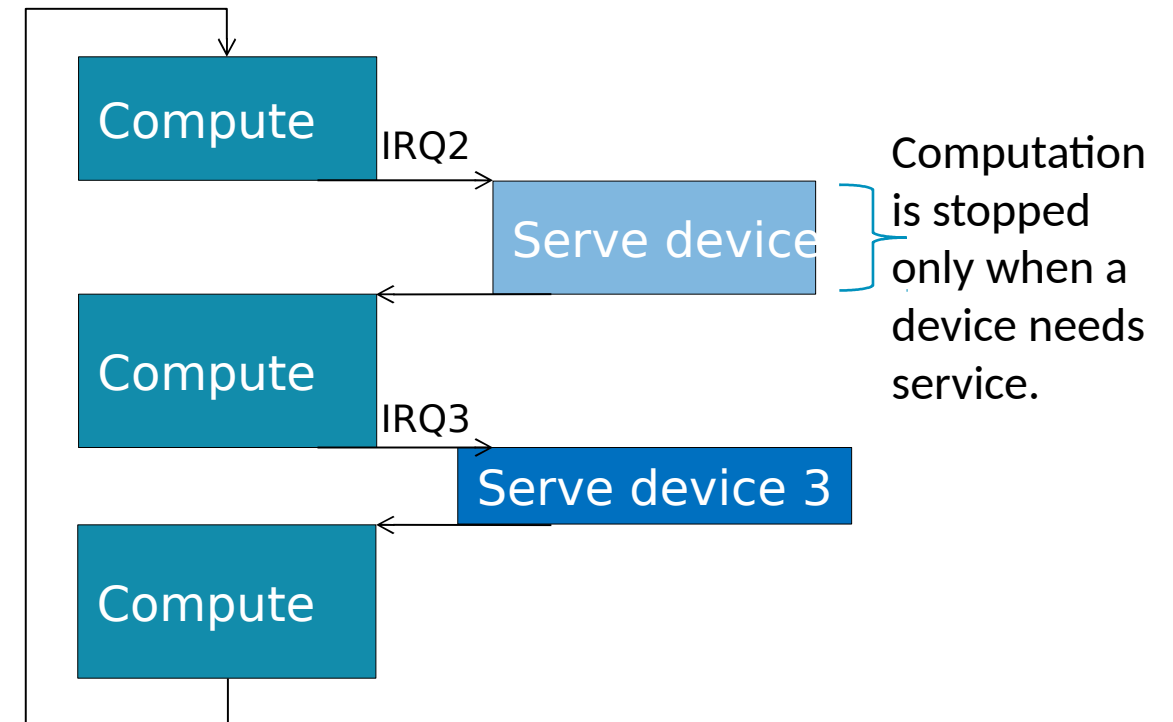
IRQ2 ()
{
    resB_1 = read_byte_from_B();
    resB_2 = read_byte_from_B();
}
```

CPU - I/O Interface

In case of **polling**, the application execution timeline will be the following.



In case of **interrupt**, the application execution timeline will be the following.

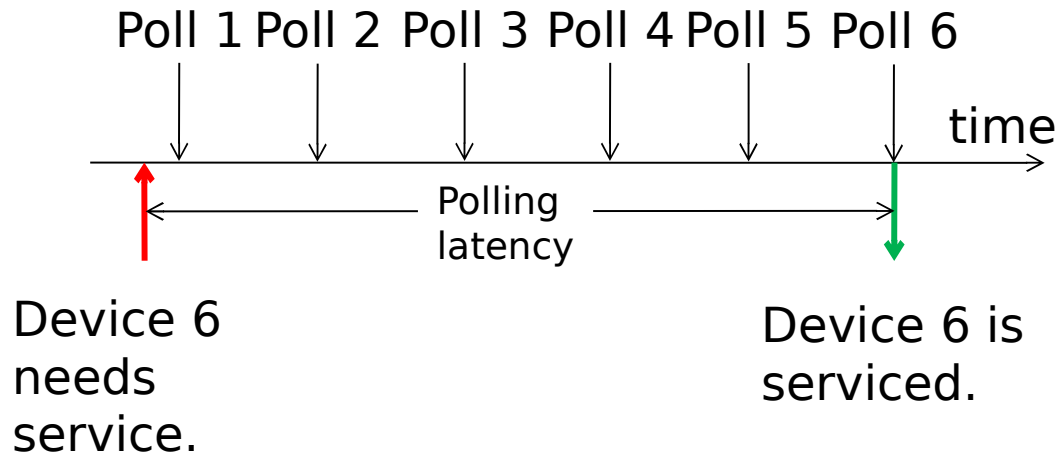


CPU – I/O Interface Latency

In case of **polling**, the latency depends on the order in which the peripherals are polled.

Example

- 10 devices; only device 6 needs service.

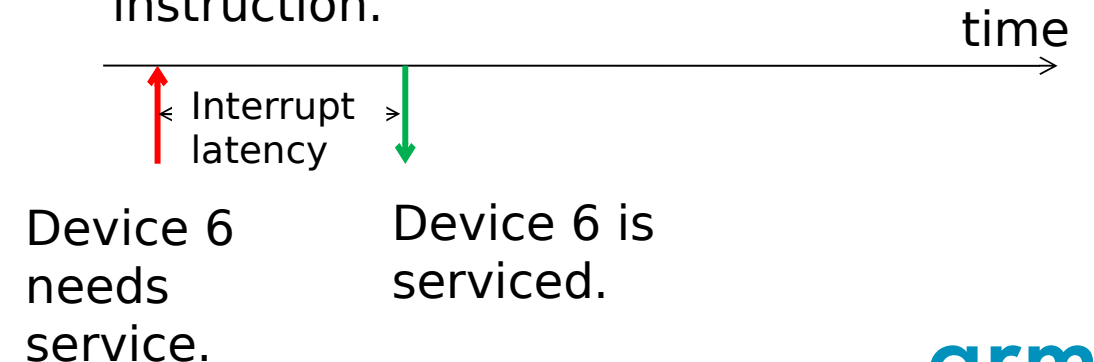


In case of **interrupt**, the latency depends on the number of requests simultaneously asserted, the CPU operating mode, and its architecture.

Example

- 10 devices; only device 6 needs service.

If the CPU is running with interrupts enabled, the latency is equal to one instruction.



CPU – I/O Interface

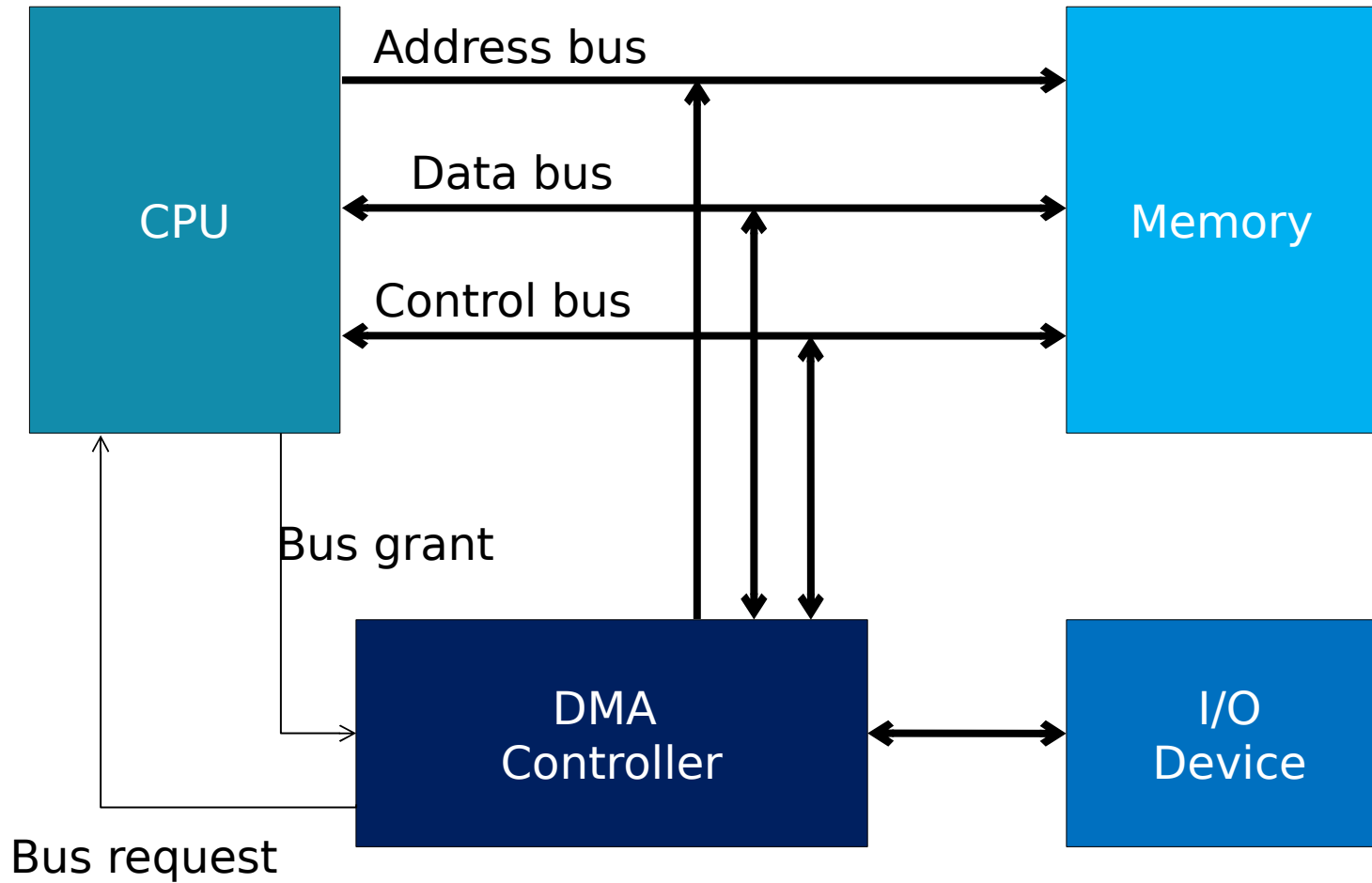
Data transfer can happen in two fashions:

- **Character-based transfer**, where data are transferred one word (e.g., 8/16/32/... bits) at a time
- **Block-based transfer**, where data are transferred as clusters of bytes (e.g, 64/128/... bytes)

The data transfer can be performed by:

- The CPU, which moves data from memory to I/O (or vice-versa) through a program
- The DMA, which releases the CPU from data transfer

Direct Memory Access (DMA) Architecture



Direct Memory Access (DMA) Transfer Modes

Burst

- An entire block of data is transferred in one contiguous sequence.
- The CPU remains inactive for relatively long periods of time (until the whole transfer is completed).

Cycle stealing

- DMA transfers one byte of data and then releases the bus returning control to the CPU.
- It continually issues requests, transferring one byte of data per request, until it has transferred the entire block of data.
- It takes longer to transfer data/the CPU is blocked for less time.

Transparent

- DMA transfers data when the CPU is performing operations that do not use the system buses.

I/O Taxonomy

Output devices which actuate commands via:

- **Analog signals:** output pins carrying voltages or currents
- **Digital level-triggered discrete signals:** output pins forming parallel bus carrying digital voltage levels
- **Digital pulse-width modulated (PWM) discrete signals:** output pins forming parallel bus carrying digital square waveforms with given frequencies and duty cycles
- **Bus-based signals:** output pins implementing serial communication protocols

Inputs devices which acquire sensors status via:

- **Analog inputs:** input pins carrying voltages, which required A/D conversion
- **Digital level-triggered discrete signals:** input pins carrying digital voltage levels
- **Digital pulse-width modulated discrete signals:** input pins forming parallel bus carrying digital information in the form of pulses duration and/or number of pulses
- **Bus-based signals:** input pins implementing serial communication protocols

Typical Operations

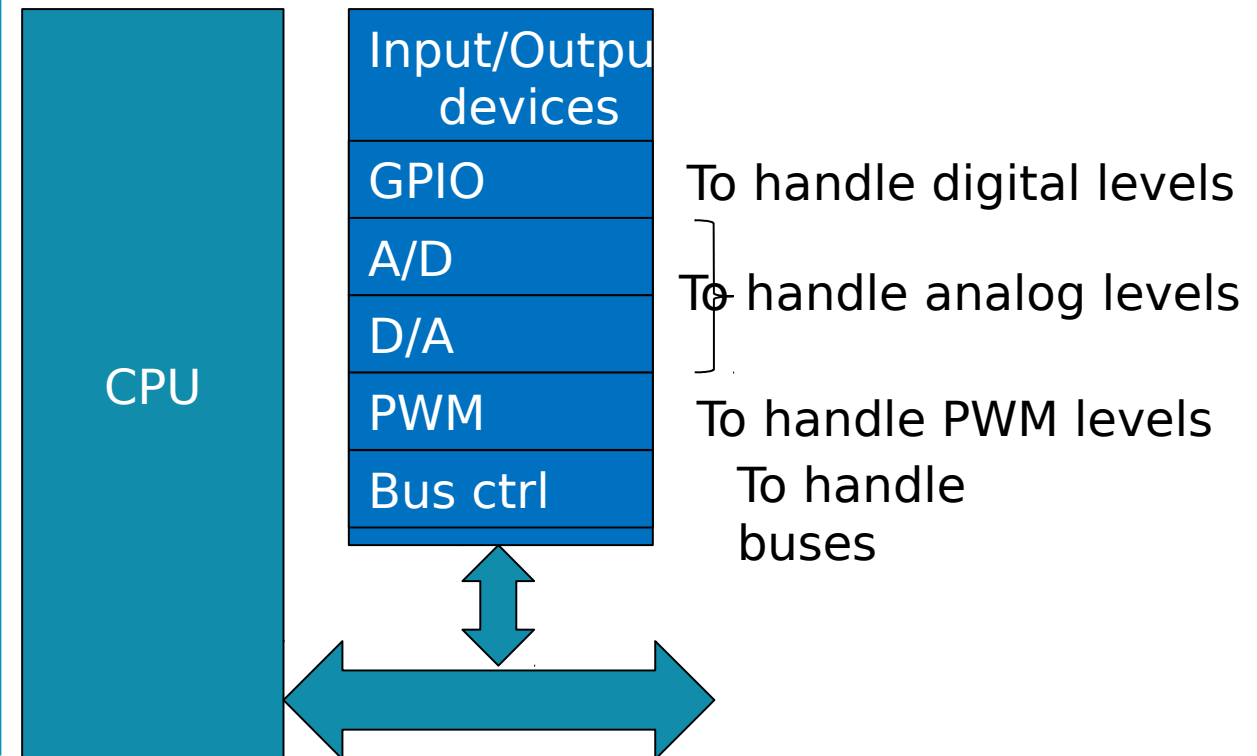
Output devices

- Generate analog levels through D/A
- General digital levels
- Generate PWM signals
- General bus transfer

Input devices

- Acquire an analog value through A/D conversion
- Read digital level
- Measure timing/repetition of digital pulses
- Read bus transfer

CPUs for embedded systems contain dedicated hw for these operations.



Linux Devices

Linux provides an abstraction to make communication with I/O easy.

- Software developer does not need to know every detail of the physical device.
- Portability can be increased by using the same abstraction for different I/O devices.

Linux recognizes three classes of devices:

- **Character devices**, which are devices that can be accessed as stream of words (e.g., 8-/16-/32-/... bits) as in a file; reading word n requires reading all the preceding words from 0 to $n-1$.
- **Block devices**, which are devices that can be accessed only as multiples of one block, where a block is 512 bytes of data or more. Typically, block devices host file systems.
- **Network interfaces**, which are in charge of sending and receiving data packets through the network subsystem of the kernel

Summary

Introduction

Embedded Linux anatomy

Configuration & Build Process of an Embedded Linux System

Linux kernel modules and device drivers

- Introduction

- CPU – I/O interface

- Virtual File System abstraction

- Linux Kernel modules

- Case study

The Virtual File System (VFS) Abstraction

Character/block devices are accessed as files stored in the file system, as each device is associated with a **device file**.

Typical usage:

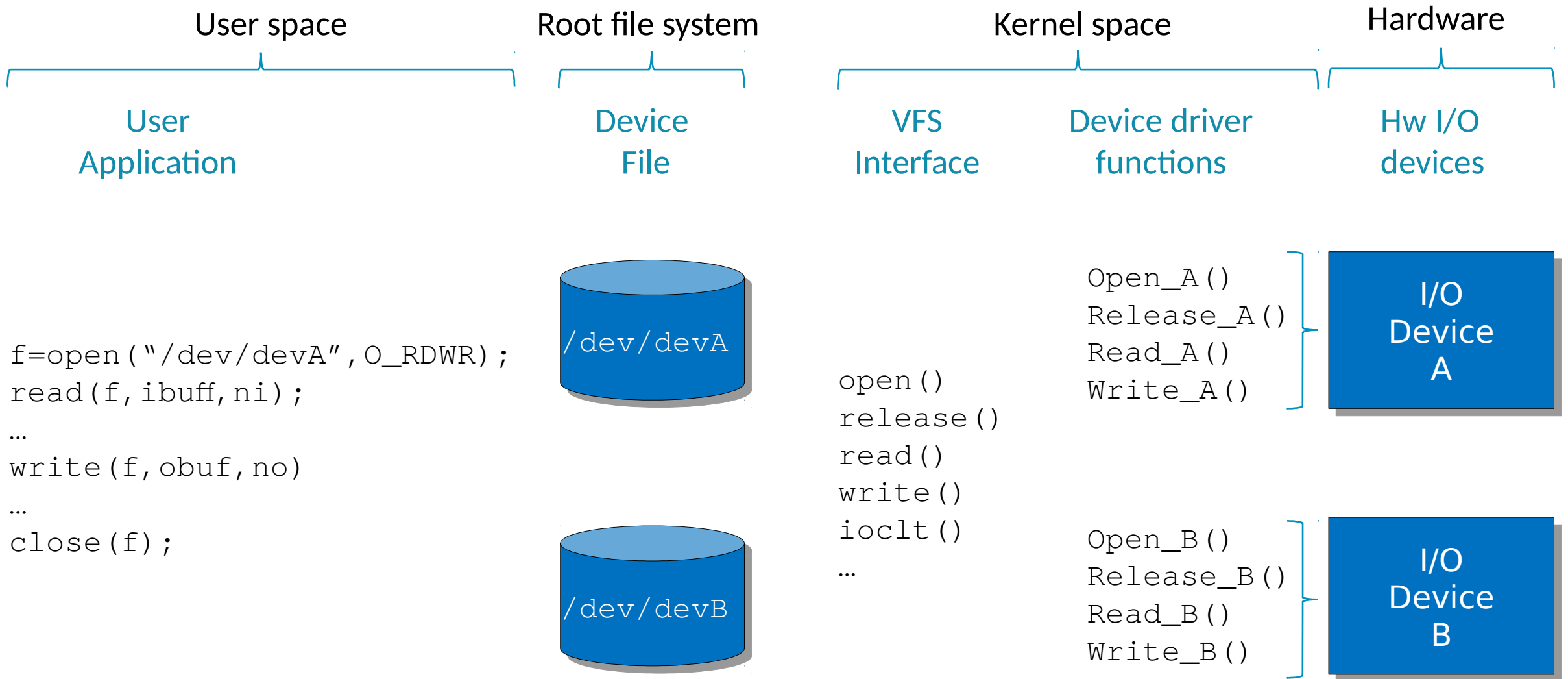
- Open the device file
- Read/Write data from/to device file
- Close the device file

Linux **forwards** the open/read/write/close operations to the I/O device associated to the device file.

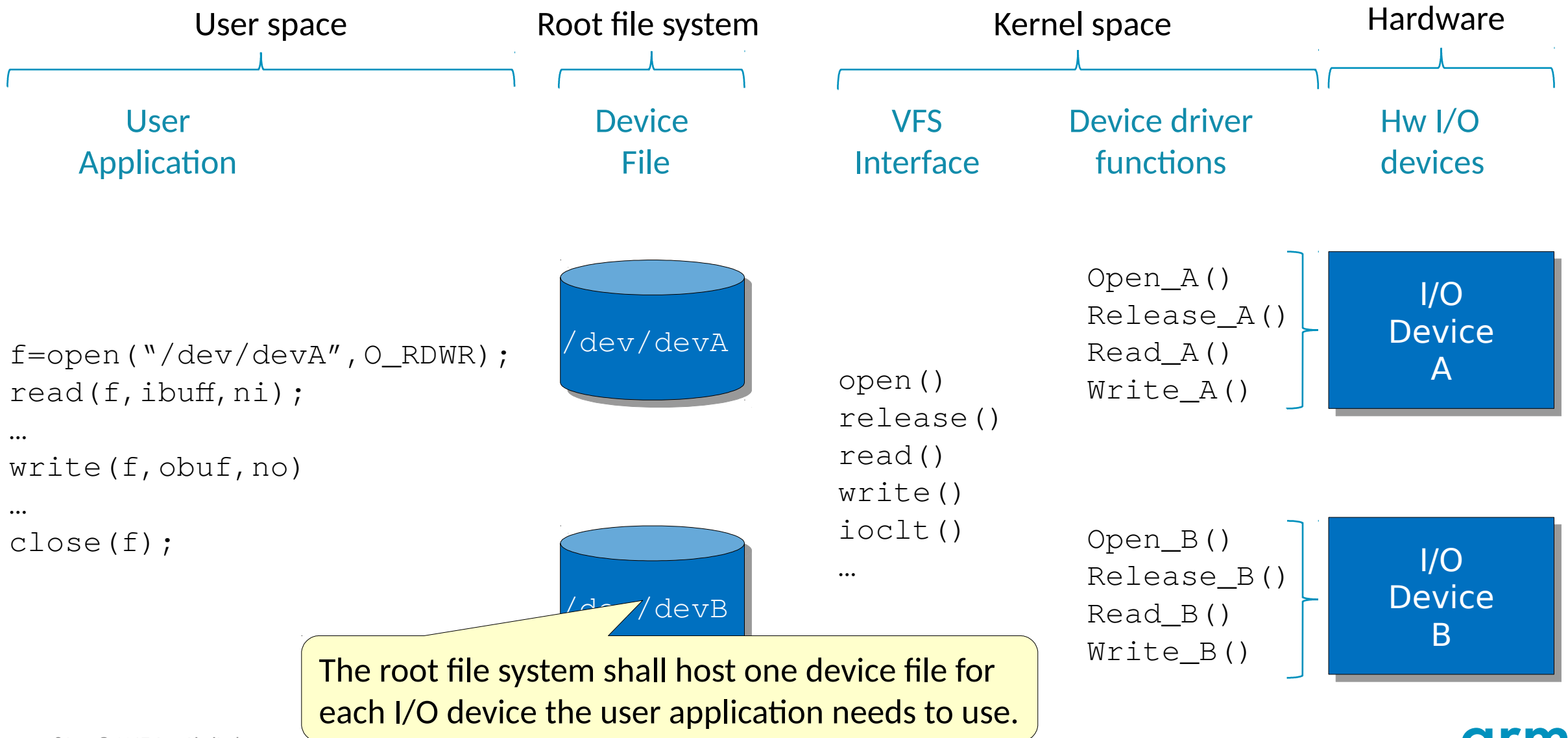
The operations for each I/O device are implemented by a custom piece of software in the Linux kernel: the **device driver**.

In the following, we will consider only character (char) devices.

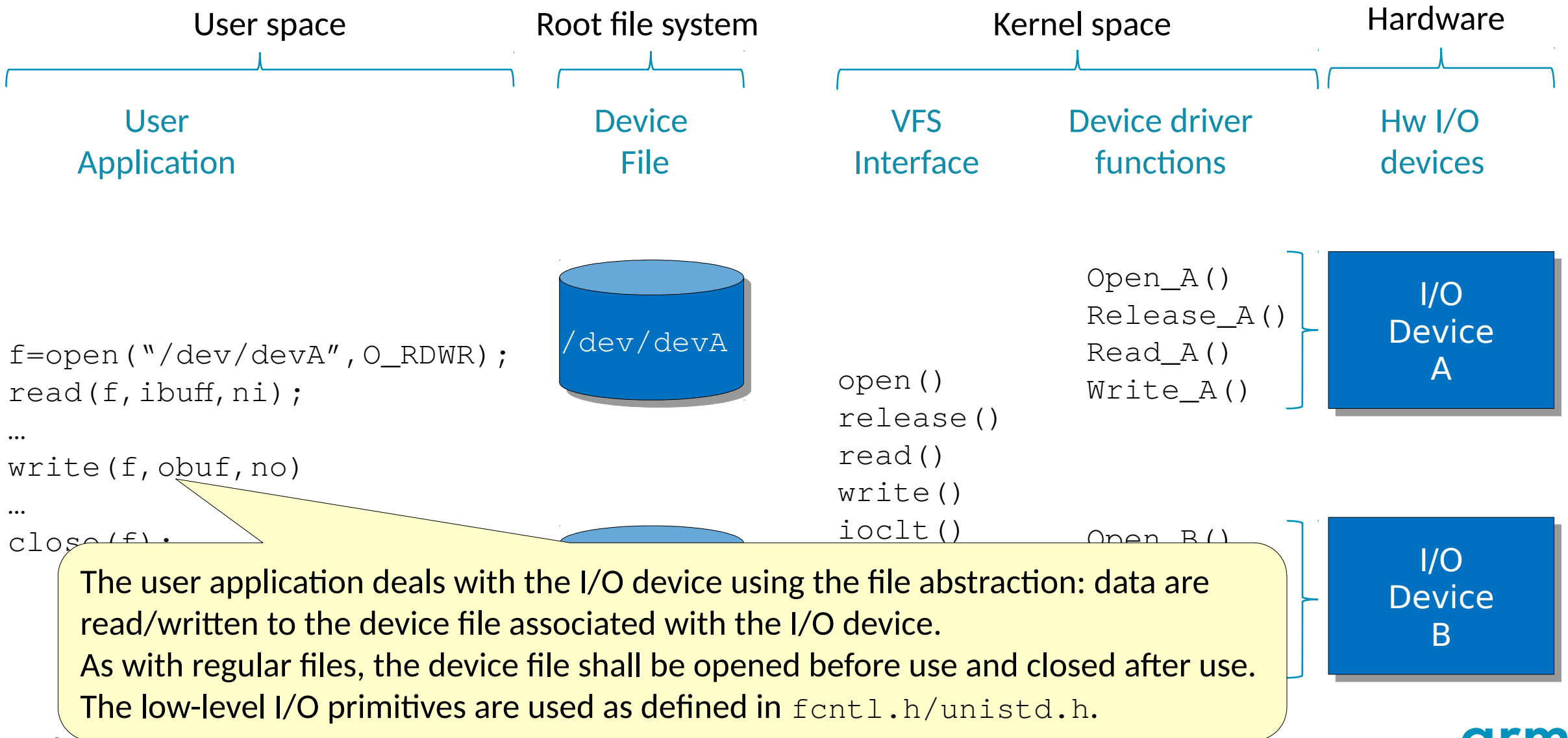
VFS: An Example



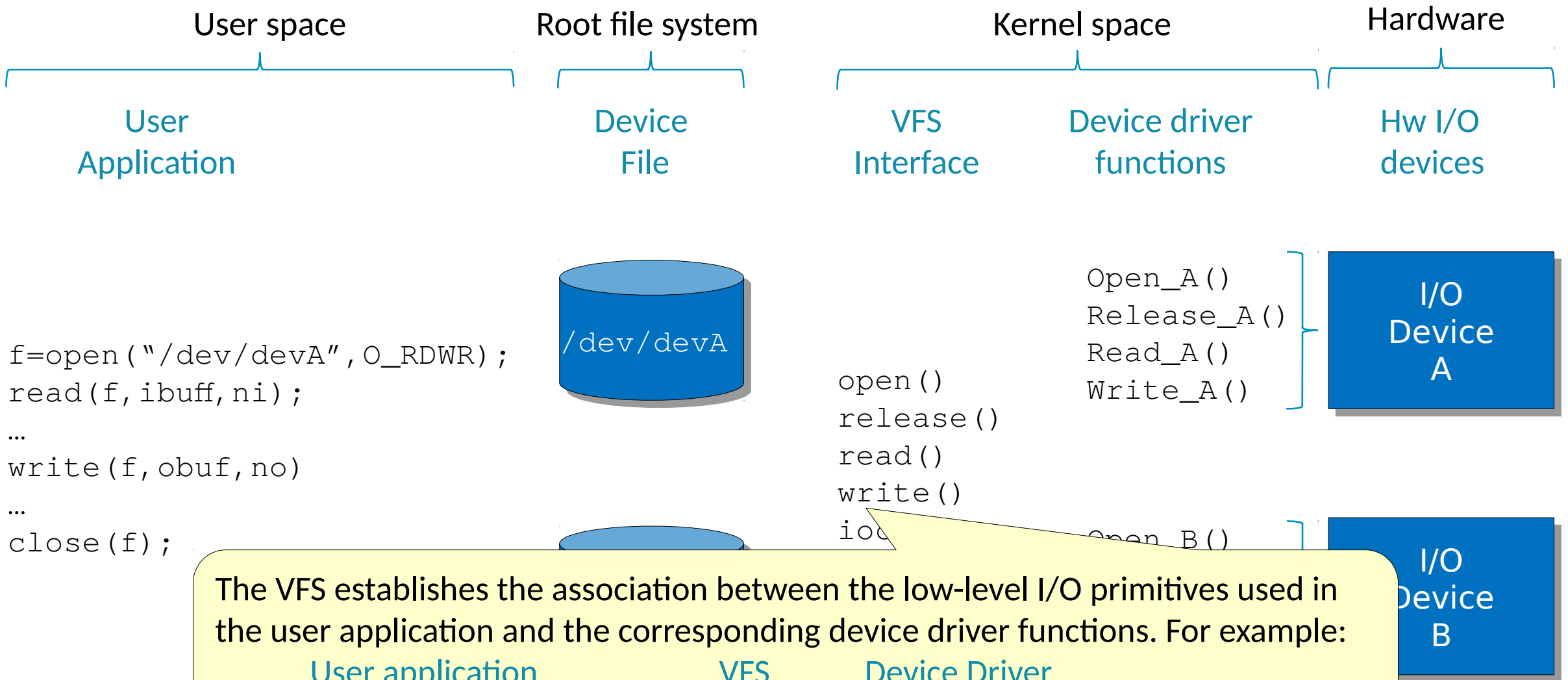
VFS: An Example



VFS: An Example



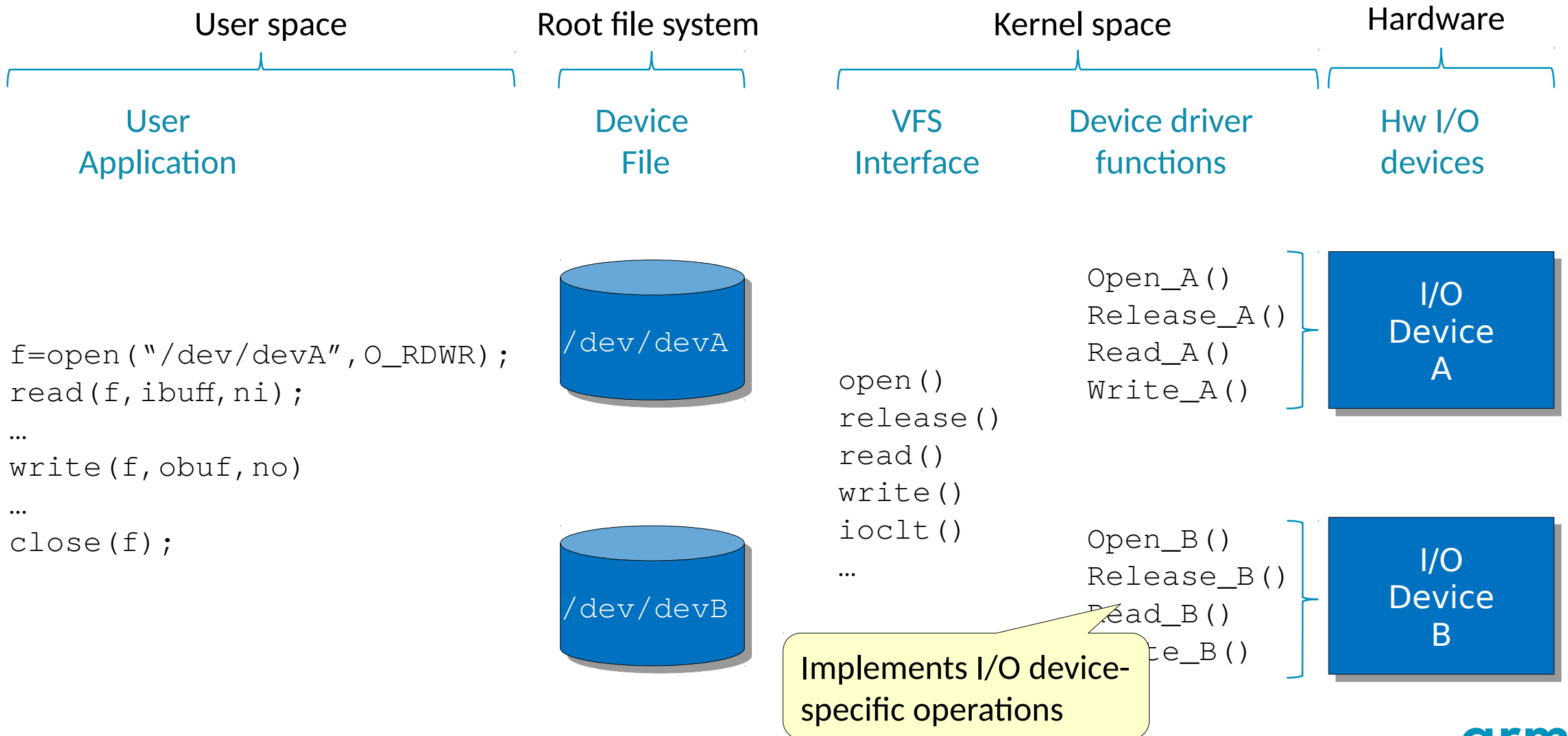
VFS: An Example



The VFS establishes the association between the low-level I/O primitives used in the user application and the corresponding device driver functions. For example:

User application	VFS	Device Driver
<code>f=open("/dev/devA", O_RDWR)</code>	<code>open()</code>	<code>Open_A()</code>

VFS: An Example



VFS Functions: `include/linux/fs.h`

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags) (int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write) (struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read) (struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease) (struct file *, long, struct file_lock **, void **);
    long (*fallocate) (struct file *file, int mode, loff_t offset,
        loff_t len);
    void (*show_fdinfo) (struct seq_file *m, struct file *f);
#ifdef CONFIG_MMU
    unsigned (*mmap_capabilities) (struct file *);
#endif
    ssize_t (*copy_file_range) (struct file *, loff_t, struct file *,
        loff_t, size_t, unsigned int);
    int (*clone_file_range) (struct file *, loff_t, struct file *, loff_t,
        u64);
    ssize_t (*dedupe_file_range) (struct file *, u64, u64, struct file *,
        u64);
};
```

VFS functions: prototypes of the functions Linux sets available for accessing a file. In case of device files, the actions each function performs are defined by the corresponding device driver.

VFS Functions: `include/linux/fs.h`

For character devices the most commonly used VFS functions are the following:

- `ssize_t (*read) (struct file *, char *__user, size_t, loff_t *)`: it reads data from a file.
- `ssize_t (*write) (struct file *, const char *__user, size_t, loff_t *)`: it writes data to a file.
- `int (*ioctl) (struct *inode, struct file *, unsigned int, unsigned long)`: it performs custom operations to the file.
- `int (*open) (struct *inode, struct file *)`: it prepares a file for use.
- `int (*release) (struct inode *, struct file *)`: it indicates the file is no longer in use.

The Device File Concept

The device file is the intermediary through which a user application can exchange data with a device driver.

The device file does not contain any data, while its descriptor contains the relevant information to identify the corresponding driver:

- The **device file type**, which could be either **c = character device**, **b = block device**, or **p = named pipe** (inter-process communication mechanism)
- The **major number**, which is an integer number that identifies univocally a device driver in the Linux kernel
- The **minor number**, which is used to discriminate among multiple instances of I/O devices handled by the same device driver

Summary

Introduction

Embedded Linux anatomy

Configuration & Build Process of an Embedded Linux System

Linux kernel modules and device drivers

Introduction

CPU – I/O interface

Virtual File System abstraction

Linux Kernel modules

Case study

Linux Kernel Modules

A device driver provides an I/O-device specific implementation of the Virtual File System abstraction, and it is located in the kernel space.

A device driver can be:

- Linked with the Linux Kernel and executed at system bootstrap
- [Kernel module](#), which is loaded at runtime through suitable system programs, after the Linux Kernel is booted

In the following slides, we will focus on kernel modules, but the same concepts apply to device drivers linked with the Linux Kernel.

Linux Kernel Modules

System programs

- `mknod`, to create a device file
- `insmod`, to insert the module in the kernel
- `rmmod`, to remove the module from the kernel
- `lsmod`, to list the modules loaded in the kernel

Functions provided by a kernel module:

- **Initialization function**, called upon the execution of the `insmod` system program, takes care of making Linux aware that a new device driver is available
- **Clean-up function**, called upon the execution of `rmmod`, to remove the device driver from the Linux Kernel
- Custom-specific implementations of the VFS abstraction

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Data structure containing the **major number** and the first **minor number** for the module. It identifies univocally the module in the kernel. It shall be used when creating the device file associated with the module.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Data structure used to describe the properties of a character device (see next slide)

Linux Kernel Modules: The cdev Data Structure

```
include/linux/cdev.h
```

```
struct cdev {  
    struct kobject kobj;  
    struct module *owner;  
    const struct file_operations *ops;  
    struct list_head list;  
    dev_t dev;  
    unsigned int count;  
};
```

The relevant fields for the module programmer are:

- `ops`: pointer to the structure defining the association between Virtual Filesystem (VFS) functions and their specific implementations for the module being developed
- `dev`: major number associated with the module
- `count`: minor number associated with the module

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Buffer used for displaying output messages on the Linux console

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Data structure used to associate the VFS functions to their module-specific implementations. In this example, the `read()` VFS function is implemented by the `dummy_read()` function.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

The module initialization function. It is executed as soon as the module enters the Linux kernel and takes care of making the Kernel aware that the new module is available.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

Kernel equivalent of the `printf()`.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It registers a range of character device numbers with the function.

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned
count, const char *name)
```

where:

`dev` is the dynamically-selected major number of the module;

`baseminor` is the first minor number for the module;

`count` is the number of minor number to reserve for the module;

and `name` is the module name.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It prints on the Linux console the major number and the minor number associated with the just-registered character device.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It initializes the character device data structure.

```
void cdev_init( struct cdev *cdev, const struct
file_operations * fops);
```

where:

`cdev` is the structure to initialize and
`fops` is the `file_operations` for the device.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It sets the owner of the module using the `THIS_MODULE` macro.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops =
{
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init
{
    printk(KERN_INFO "Loading dummy module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It adds the character device to the Linux Kernel

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

where:

`p` is the character device structure already initialized;

`dev` is the major number for the device;

and `count` is number of minor numbers for which the device is responsible.

Linux Kernel Modules: The Initialization Function

```
static dev_t dummy_dev;

struct cdev dummy_cdev;

static char buffer[64];

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .read = dummy_read,
};

static int __init dummy_module_init(void)
{
    printk(KERN_INFO "Loading dummy_module\n");
    alloc_chrdev_region(&dummy_dev, 0, 1, "dummy_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, dummy_dev));
    cdev_init(&dummy_cdev, &dummy_fops);
    dummy_cdev.owner = THIS_MODULE;
    cdev_add(&dummy_cdev, dummy_dev, 1);
    return 0;
}
```

It indicates the function terminated correctly.
A non zero value indicates an error occurred.

Linux Kernel Modules: The Clean-up Function

```
static void __exit dummy_module_cleanup(void)
{
    printk(KERN_INFO "Cleaning-up dummy_dev.\n");

    cdev_del(&dummy_cdev);
    unregister_chrdev_region(dummy_dev, 1);
}
```

Linux Kernel Modules: The Clean-up Function

```
static void __exit dummy_module_cleanup(void)
{
    printk(KERN_INFO "Cleaning-up dummy_dev.\n");

    cdev_del(&dummy_cdev);
    unregister_chrdev_region(dummy_dev, 1);
}
```

It removes the character device from the Linux kernel.

It frees the range of major/minor numbers previously registered.

Linux Kernel Modules: Custom VFS Functions

```
ssize_t dummy_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    printk(KERN_INFO "Dummy read (count=%d, offser=%d)\n", (int)count, (int)*f_pos );

    return 1;
}
```

Linux Kernel Modules: Custom VFS Functions

```
ssize_t dummy_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    printk(KERN_INFO "Dummy read: read offser=%d\n", (int)count, (int)*f_pos );

    return 1;
}
```

Implementation of the VFS function to read from a file, where `filp` is the pointer to the data structure describing the opened file; `buf` is the bufer to fill with the data read from the file; `count` is the size of the buffer, and `f_pos` is the current reading position in the file.

Linux Kernel Modules: Custom VFS Functions

```
ssize_t dummy_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    printk(KERN_INFO "Dummy read (count=%d, offser=%d)\n", (int)count, (int)*f_pos );

    return 1;
}
```

It prints a message to show that the read functions has been executed.

It returns the number of byte read from the file.
Returning 0 will block the caller application, which will wait until at least one byte is returned.

Summary

Introduction

Embedded Linux anatomy

Configuration & Build Process of an Embedded Linux System

Linux kernel modules and device drivers

Introduction

CPU – I/O interface

Virtual File System abstraction

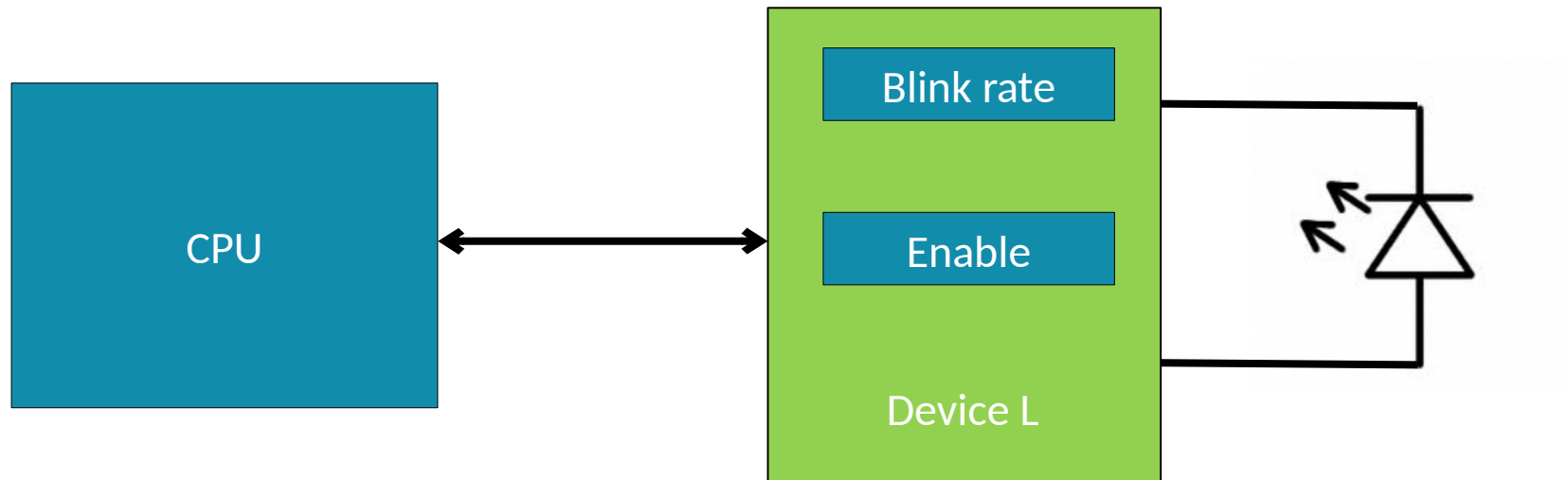
Linux Kernel modules

Case study

The Reference Use Case

To illustrate the concept let's consider this example:

- The custom hardware **device L** is attached to the CPU, and it controls a led.
- When enabled, L turns led on/off according to a user-defined blink rate.
- **Blink rate register**: 32 bits with the blink rate in Hz
- **Enable register**: 1 bit, when set to 0, the device L disabled; when set to 1, the device L enabled.



The CPU/Device Interface

CPU/Device L connection can be either:

- Memory mapped: each register is associated to an address, as in the example below.

Blink rate register	0xf0080000
Enable register	0xf0080004

- Through GPIO, where each register is associated to a set of GPIOs, as in the example below.

Blink rate register	gpio(0-31)
Enable register	gpio(32)

- Serial Communication (SPI, I2C, etc.)

The module-level implementation will be affected by the adopted CPU/Device Interface.

The user-level implementation will abstract these details.

Case study

The module-level point of view

The user-level point of view

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- **Reset:** L is disabled, and the blink rate register is set to zero.
- **Program:** The blink rate register is set to a user-defined value.
- **Enable:** L is enabled.
- **Disable:** L is disabled.
- **Poll rate:** L returns the content of the blink rate register.
- **Poll state:** L returns the content of the enable register.
- **Power-off:** L terminates its operation and starts waiting for the next reset.

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- **Reset:** L is disabled, and the blink rate register is set to zero.
- **Program:** The blink rate register is set to a user-defined value.
- **Enable:** L is enabled.
- **Disable:** L is disabled.
- **Poll rate:** L returns the content of the blink rate register.
- **Poll state:** L returns the content of the enable register.
- **Power-off:** L terminates its operation and starts waiting for the next reset.

Both the blink rate and enable registers are set to zero.
This operation is done once, when starting using the device.

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:


- **Reset:** L is disabled, and the blink rate register is set to zero.
- **Program:** The blink rate register is set to a user-defined value.
- **Enable:** L is enabled.
- **Disable:** L is disabled.
- **Poll rate:** L returns the content of the blink rate register.
- **Poll state:** L returns the content of the enable register.
- **Power-off:** L terminates its operation and starts waiting for the next reset.

The blink rate register is set to a user defined value.
This operation can be done multiple times, during the device usage.

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.


In our example, the device L supports the following functionalities:

- **Reset:** L is disabled, and the blink rate register is set to zero.
 - **Program:** The blink rate register is set to a user-defined value.
 - **Enable:** L is enabled.
 - **Disable:** L is disabled.
 - **Poll rate:** L returns the content of the enable register.
 - **Poll state:** L returns the content of the enable register.
 - **Power-off:** L terminates its operation and starts waiting for the next reset.
- 
- The enable register is set to one.
This operation can be done multiple times, during the device usage.

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- **Reset:** L is disabled, and the blink rate register is set to zero.
 - **Program:** The blink rate register is set to a user-defined value.
 - **Enable:** L is enabled.
 - **Disable:** L is disabled.
 - **Poll rate:** L returns the content of the blink rate register.
 - **Poll state:** L returns the content of the enable register.
 - **Power-off:** L terminates its operation and starts waiting for the next reset.
- 
- The enable register is set to zero.
This operation can be done multiple times, during the device usage.

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- **Reset:** L is disabled, and the blink rate register is set to zero
- **Program:** the blink rate register is set to a user-defined value.
- **Enable:** L is enabled.
- **Disable:** L is disabled.
- **Poll rate:** L returns the content of the blink rate register.
- **Poll state:** L returns the content of the enable register.
- **Power-off:** L terminates its operation.

The blink rate register content is provided to the user level.
This operation can be done multiple times, during the device usage.

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- **Reset:** L is disabled, and the blink rate register is set to zero.
- **Program:** The blink rate register is set to a user-defined value.
- **Enable:** L is enabled.
- **Disable:** L is disabled.
- **Poll rate:** L returns the content of the blink rate register.
- **Poll state:** L returns the content of the enable register.
- **Power-off:** L terminates its operation and starts waiting for the next reset.

The enable register content is provided to the user level.
This operation can be done multiple times, during the device usage.

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- **Reset:** L is disabled, and the blink rate register is set to zero.
- **Program:** The blink rate register is set to a user-defined value.
- **Enable:** L is enabled.
- **Disable:** L is disabled.
- **Poll rate:** L returns the content of the blink rate register.
- **Poll state:** L returns the content of the enable register.
- **Power-off:** L terminates its operation and starts waiting for the next reset.



The enable register content is set to zero.
This operation is done once, after the last usage of the device.

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

The virtual file system functions that are typically used are:

- `open`, which initiates the operations with the device
- `release`, which terminates the operation with the device
- `write`, which sends data coming from the user space to the device
- `read`, which reads from the device and send them to the user space
- `ioctl`, which performs custom operations

The Module Level

For the considered example, the following association between device L functionalities and the VFS is proposed.

Device functionality	Virtual file system function	Notes
Reset	open	open() is used once, to establish the connection with the device.
Program	write	write() is used to send data to the device. The blink rate register shall be selected as target for the write operation using the ioctl() function.
Enable	write	write() is used to send data to the device. The enable register shall be selected as target for the write operation using the ioctl() function.
Disable	write	write() is used to send data to the device. The enable register shall be selected as target for the write operation using the ioctl() function.

The Module Level

For the considered example, the following association between device L functionalities and the VFS is proposed.

Device functionality	Virtual file system function	Notes
Poll rate	read	read() is used to send data to the application. The blink rate register shall be selected as target for the read operation using the ioctl() function.
Poll state	read	read() is used to send data to the application. The enable register shall be selected as target for the read operation using the ioctl() function.
Power-off	release	release() is used to terminate the connection with the device.
None	ioctl	ioctl() is used to select the target for read/write operations.

The Module Level: File Operations

```
static dev_t L_dev;
```

```
struct cdev L_cdev;
```

```
struct file_operations L_fops = {
```

```
    .owner    = THIS_MODULE,  
    .open     = L_open_close,  
    .release  = L_open_close,  
    .write    = L_write,  
    .read     = L_read,  
    .ioctl    = L_ioctl,
```

```
};
```

VFS functions

Device-specific functions

The Module Level: ioctl() Implementation

When `cmd` is set to `BLINK_RATE/ENABLE`, the blink rate/enable register is selected.

```
static ssize_t L_ioctl(struct inode *inode, struct file *filep, unsigned int cmd, unsigned long arg)
{
    switch( cmd )
    {
        case BLINK_RATE:                // global symbol previously defined
            selected_register = BLINK_RATE;    // global variable previously defined
            break;
        case ENABLE:                    // global symbol previously defined
            selected_register = ENABLE;
            break;
    }

    return 0;
}
```

The Module Level: open()/release() Implementation

It disables the device and sets the blink rate to zero. The same operations are valid for open() and release() functions.

```
static int L_open_close(struct inode *inode, struct file *file)
{
    selected_register = ENABLE;           // it selects the target for read/write operation.
    WRITE_DATA_TO_THE_HW( 0 );           // it sends 0 to the enable register.
                                         // it abstracts the low-level CPU/device L interface.

    selected_register = BLINK_RATE;       // it selects the target for read/write operation.
    WRITE_DATA_TO_THE_HW( 0 );           // it sends 0 to the blink register.

    return 0;
}
```

The Module Level: read() Implementation

It reads from the ioctl() selected register and pass the data to the user.

```
static ssize_t L_read(struct file *filp, char *buffer, size_t length, loff_t * offset)
{
    int data;

    READ_DATA_FROM_THE_HW( &data );           // it abstracts the low-level CPU/device L interface.
    copy_to_user(buffer, &data, 4 );          // see next slide

    return 4;                                // it returns the number of bytes read.
}
```


Passing Data to/from the Kernel

Kernel and application are running in two different memory spaces.

Specific functions are needed to move data between them.

```
copy_to_user(void __user *to, const void *from, unsigned long n)
```

Move data from kernel space to user space.

The Module Level: write() Implementation

It writes to the ioctl() selected register the data coming from the user.

```
static ssize_t L_write(struct file *filp, char *buffer, size_t length, loff_t * offset)
{
    WRITE_DATA_TO_THE_HW( buffer );

    return 1;
}
```

The Module Level: Communication with the Device

Hidden in

- `READ_DATA_FROM_THE_HW()`
- `WRITE_DATA_TO_HW()`

Implementation depends on the CPU/Device L connection

Memory mapped example:

- Blink rate register: `0xf0080000`
- Enable register: `0xf0080004`

GPIO example:

- Blink rate register: `GPIO(0-31)` (MSB first)
- Enable register: `GPIO(32)`

Memory Mapped I/O

Memory areas can be used if:

- Available
- Reserved

```
int check_region( unsigned long first, unsigned long n)
```

- It checks whether the desired addresses are available

```
int request_region( unsigned long first, unsigned long n,  
const char *name)
```

- It reserves the desired addresses

```
int release_region( unsigned long first, unsigned long n)
```

- It sets the desired addresses free

Memory Mapped I/O: Initialization

```
static int __init L_module_init(void)
{
    int res;
    alloc_chrdev_region(&L_dev, 0, 1, "L_dev");
    printk(KERN_INFO "%s\n", format_dev_t(buffer, L_dev));
    cdev_init(&L_cdev, &L_fops);
    L_cdev.owner = THIS_MODULE;
    cdev_add(&L_cdev, L_dev, 1);

    r = check_region( ioremap(0xf0080000, 4 ), 8 );
    if(r) {
        printk( KERN_ALERT "Unable to reserve I/O memory\n");
        return -EINVAL;
    }
    request_region(ioremap(0xf0080000, 4), 8, "DevL");
    return 0;
}
```

Translates the physical address of the device (as defined by the memory map) into the corresponding virtual address.

Memory Mapped I/O: Clean-up

```
static void __exit L_module_cleanup(void)
{
    cdev_del(&L_cdev);
    unregister_chrdev_region(L_dev, 1);

    release_region(ioremap(0xf0080000, 4 ), 8);
}
```

Memory Mapped I/O: read

```
int READ_DATA_FROM_THE_HW( int *data )
{
    int    tmp;

    switch( selected_register )
    {
        case BLINK_RATE:
            tmp = inl( ioremap(0xf0080000, 4) );
            break;
        case ENABLE:
            tmp = inl( ioremap(0xf0080000, 4)+4 );
            break;
    }
    *data = tmp;

    return 4;
}
```

Functions for accessing I/O memory:

`inb()`: it reads 8-bit words.

`inw()`: it reads 16-bit words.

`inl()`: it reads 32-bit words.

Memory Mapped I/O: write

```
int WRITE_DATA_TO_THE_HW( char *data )
{
    switch( selected_register )
    {
        case BLINK_RATE:
            outl( (int)*data, ioremap(0xf0080000, 4) );
            break;
        case ENABLE:
            outl( (int)*data, ioremap(0xf0080000, 4)+4 );
            break;
    }

    return 4;
}
```

Functions for accessing I/O memory:

`outb()` : it writes 8-bit words.

`outw()` : it writes 16-bit words.

`outl()` : it writes 32-bit words.

Interrupts

Often, kernel modules have to react to interrupts coming from the hardware.

To handle interrupts, a Kernel module shall:

- Request an interrupt line
- Associate an interrupt handler to an interrupt line
- Implement the interrupt handler

When finished with the device and unregistering the driver, the Kernel module shall free the interrupt line.

Requesting the Interrupt Line

```
int request_irq(  
    unsigned int irq,  
    irqreturn_t (*handler)(),  
    unsigned long flags,  
    const char *dev_name,  
    void *dev_id  
);
```

Interrupt line to manage

Function pointer to the
interrupt handler

Options to be used when associating the
interrupt handler to the interrupt line

Name of the module requesting the interrupt

Pointer to a user-defined structure
containing device-specific data. It can
be NULL.

Freeing the Interrupt Line

```
int free_irq(  
    unsigned int irq,  
    void *dev_id  
);
```

Interrupt line to manage

Pointer to a user-defined structure containing device-specific data. It can be NULL.

The Interrupt Handler

```
static irqreturn_t hlr( int irq, void *dev_id )
{
    /*
     * Do something to handle the interrupt *
     */

    return IRQ_RETVAL(1);
}
```

Interrupt Handling

Interrupt handlers may introduce long latencies.

- Lower-priority interrupts have to wait.
- If interrupts are disabled, everyone has to wait.

Rule of thumb: Keep interrupt handlers as short as possible.

Top-half and Bottom-half

Split interrupt handling in two parts

Top-half

- Manages the interaction with real hw
- Does the minimum amount of work
- Keep the other events pending for the least amount of time

Bottom-half

- Processes the data coming from hw
- It can be interrupted.

Needed Support

The idea

- Create a “process” that waits for incoming data.
- The “process” sleeps until a new data is ready.
- The interrupt handler prepares the new data and dispatches it to the waiting “process”.

Benefits

- The interrupt handler is very short ⇒ low latency
- The “process” can be interrupted ⇒ low latency

Work Queue

General structure in the Linux Kernel

A **work queue** is a list of activities to be executed.

Each activity is defined by

- **Work**: data to be processed
- **Callback**: function to process the work

API to manage work queues

```
struct workqueue_struct *create_workqueue( static char * );
```

```
void destroy_workqueue( struct workqueue_struct * );
```

```
int flush_workqueue( struct workqueue_struct * );
```


Work Queue

The **work** is defined using the `work_struct` structure.

- Typically, the first element of a user-defined structure storing the actual data to be processed by the callback

The **callback** is a generic C function.

API for work management:

```
INIT_WORK( work, func )
```

```
int queue_work( struct workqueue_struct *, struct work_struct * )
```

Case study

The module-level point of view

The user-level point of view

The User Level

At this level, the application invokes the VFS calls to implement the intended behavior.


The mapping between VFS functions and custom hardware functionalities is known and exploited to implement the desired behavior.

For the considered example, the application shall

- Open the connection with the device
- Set the desired blinking rate
- Enable the device
- Adjust the blinking rate (if needed)
- Disable/enable the device (if needed)
- Close the connection with the device


The User Level: Application

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```



Header files for the needed functions
prototypes/data types

```
int main(int argc, char **argv){
    char *app_name = argv[0];
    char *dev_name = "/dev/devL";
    int fd = -1;
    int x, c;
```



Variables needed for the operations of the
application

The User Level: Application

It opens the device file using the `open()` system call.
File is opened as read/write.

```
/*  
 * Open the sample device RD | WR  
 */  
if ((fd = open(dev_name, O_RDWR)) < 0) {  
    fprintf(stderr, "%s: unable to open %s: %s\n", app_name, dev_name, strerror(errno));  
    return( 1 );  
}
```

As a result, the module initializes the device L, whose blink rate register is now zero, and disables it.

The User Level: Application

```
x = ioctl(fd, BLINK_RATE, 0);          // it selects the blink rate register.
                                     // BLINK_RATE is a global symbol defined with the same value
                                     // used in the loadable kernel module.

c = 25;
x = write(fd, &c, 4 );                // it writes 25 in the blink rate register - 4 bytes

x = ioctl(fd, ENABLE, 0);             // it selects the enable register.
c = 1;
x = write(fd, &c, 4 );                // it enables the device.

if (fd >= 0) {                        // it closes the connection with the device.
    close(fd);
}
return( 0 );
}
```