Training Agenda

- Kernel Advance Topics
- Hands-on with Telechip
- Linux Device Driver

C5. Introduction to Android Automotive and Advanced Topics (2/2)

Duration: 40 hours

- 14. Advanced Kernel Topics
- - Kernel Synchronization Techniques
- - Spinlocks, mutexes, semaphores, and atomic variables.
- - Memory Management in Linux Kernel
- - Overview of paging, segmentation, and the slab allocator.
- - Real-Time Linux Considerations
- - Introduction to PREEMPT_RT patches for real-time capabilities.
- Sub-Task: Apply synchronization mechanisms in kernel module development and observe their effects.

Tools

Hardware: Telechips
Software: Linux Kernel Source

Note: Telechips as hardware, offered by Faurecia

C3. Driver Development and Peripheral Interface (1/3)

Duration: 40 hours

- 8. Character, Block, and Network Driver Writing
- - Writing Character, Block, and Network Drivers
- - Differences between character, block, and network drivers.
- - Key functions and structures in writing each type of driver.
- - File Operations in Drivers
- - File operations (`open`, `read`, `write`, `ioctl`).
- - Understanding the VFS (Virtual File System) layer.
- - Understanding the Kernel Driver Model
- - Driver registration and unregistration.
- - Major and minor numbers.
- Project 5: Character Driver Implementation
- - Implement a character driver to control GPIOs on Telechips.

Tools

Hardware: Telechips
Software: Linux Kernel Source, GPIO Tools

Note: Telechips as hardware, offered by Faurecia

C3. Driver Development and Peripheral Interface (2/3)

Duration: 40 hours

- 9. Platform Device Driver Creation and APIs
- - Creating Platform Device Drivers
- - Platform devices and drivers.
- - Registering platform devices.
- - Understanding Platform APIs and Their Usage
- - Overview of platform-specific APIs.
- - Managing power and clock resources for devices.
- Sub-Task: Develop a platform driver for an onboard peripheral like SPI or I2C.

Tools

Hardware: Telechips
Software: Linux Kernel Source, SPI/I2C Tools

Note: Telechips as hardware, offered by Faurecia

C3. Driver Development and Peripheral Interface (3/3)

Duration: 40 hours

- 10. Peripheral Interface (GPIO, SPI, I2C, etc.)
- - Detailed Study of Peripheral Subsystems
- - Overview of common peripherals: GPIO, SPI, I2C, UART, PWM.
- - Implementing and Configuring Peripherals
- - Configuring GPIO pins for input and output.
- - Writing drivers for SPI, I2C communication.
- - Hardware-Software Interfacing
- - Interfacing peripherals with device drivers.
- Project 6: Peripheral Driver Development
- - Write and test drivers for specific peripherals such as SPI and I2C on Telechips.

Tools

Hardware: Telechips
Software: Peripheral Tools, Linux Kernel Source

Note: Telechips as hardware, offered by Faurecia

Kernel Advance Topics

Kernel Synchronization

Overview

- In a multi-threaded or multi-processor environment, multiple threads/processes may access shared kernel resources (variables, data structures, devices) simultaneously. This can lead to race conditions, causing data corruption or system crashes.
- **Critical regions:** Code paths that access and manipulate shared data
- **Race condition:** situation that two threads of execution be simultaneously in the same critical region
- **Dead Locks**
- To avoid such scenario synchronization mechanism used in kernel.
  - Synchronization: ensure that race conditions do not occur and prevent unsafe concurrency
  - Synchronization issues and how to prevent race conditions
  - Ssynchronization methods' interface, behavior, and use

Kernel Synchronization methods

- Atomic Operations
- Spin Locks
- Reader-Writer Spin Locks
- Semaphores
- Reader-Writer Semaphores
- Spin Locks VS Semaphores
- Completion Variables
- BKL: The Big Kernel Lock
- Preemption Disabling
- Ordering and Barriers

Atomic Operations

What It Does

- Performs indivisible read-modify-write operations (cannot be interrupted).
- Ensures a variable is updated atomically (no partial writes).

How It Works

- Uses CPU-level atomic instructions (e.g., x86 LOCK prefix).

NOTE - For Atomic No locking needed—hardware guarantees the atomicity.

How It Fixes Race Conditions

- Prevents partial updates (e.g., counter++ is non-atomic normally).
- Ensures only one thread modifies the variable at a time.

Where to Use It

- Simple counters (atomic_t).
- Flag variables (atomic_read, atomic_set).
- But its Not for complex operations (e.g., linked list updates).

Spin Locks

What It Does

- Locks the CPU in a busy-wait loop until the lock becomes available.
- Used in interrupt context or when sleep is not allowed.

How It Works

- Thread tries to acquire the lock (spin_lock()).
- If locked, it keeps spinning (consuming CPU).
- Once unlocked, it takes the lock and proceeds.

How It Fixes Race Conditions

- Only one thread can hold the lock at a time.
- Ensures exclusive access to shared data.

Where to Use It

- Short critical sections (few instructions).
- Interrupt handlers (cannot sleep).
- Never in long-running code (wastes CPU).

- **NOTE**
  - You will deadlock if you attempt to acquire a spin lock you've already held
  - If use in interrupt handlers then first disable local interrupt requests Otherwise: double acquire deadlock

Semaphore

What It Does

- A counter-based lock that allows multiple threads (up to a limit).

Two types:

- Binary semaphore (like a mutex, but can be released by any thread).
- Counting semaphore (allows N threads).

How It Works

- Thread calls down() (decrements semaphore).
- If count > 0, it proceeds.
- If count = 0, it sleeps until available.
- Thread calls up() (increments semaphore, wakes waiters).

How It Fixes Race Conditions

- Limits concurrent access to a resource.
- Ensures controlled access (e.g., only 5 threads at a time).

Where to Use It

- Resource pooling (e.g., database connections).
- Producer-consumer problems (buffers).
- Not for simple exclusive locks (use mutex instead).

Mutex

What It Does

- A sleeping lock—if the lock is unavailable, the thread sleeps (no CPU waste).
- Ensures only one thread enters the critical section.
- Supports **ownership** (only the locking thread can unlock).

How It Works

- Thread calls mutex_lock().
- If the lock is free, it takes it.
- If locked, the thread sleeps and is woken up later.

How It Fixes Race Conditions

- Only one thread can hold the mutex at a time.
- Other threads wait without spinning, reducing CPU usage.

Where to Use It

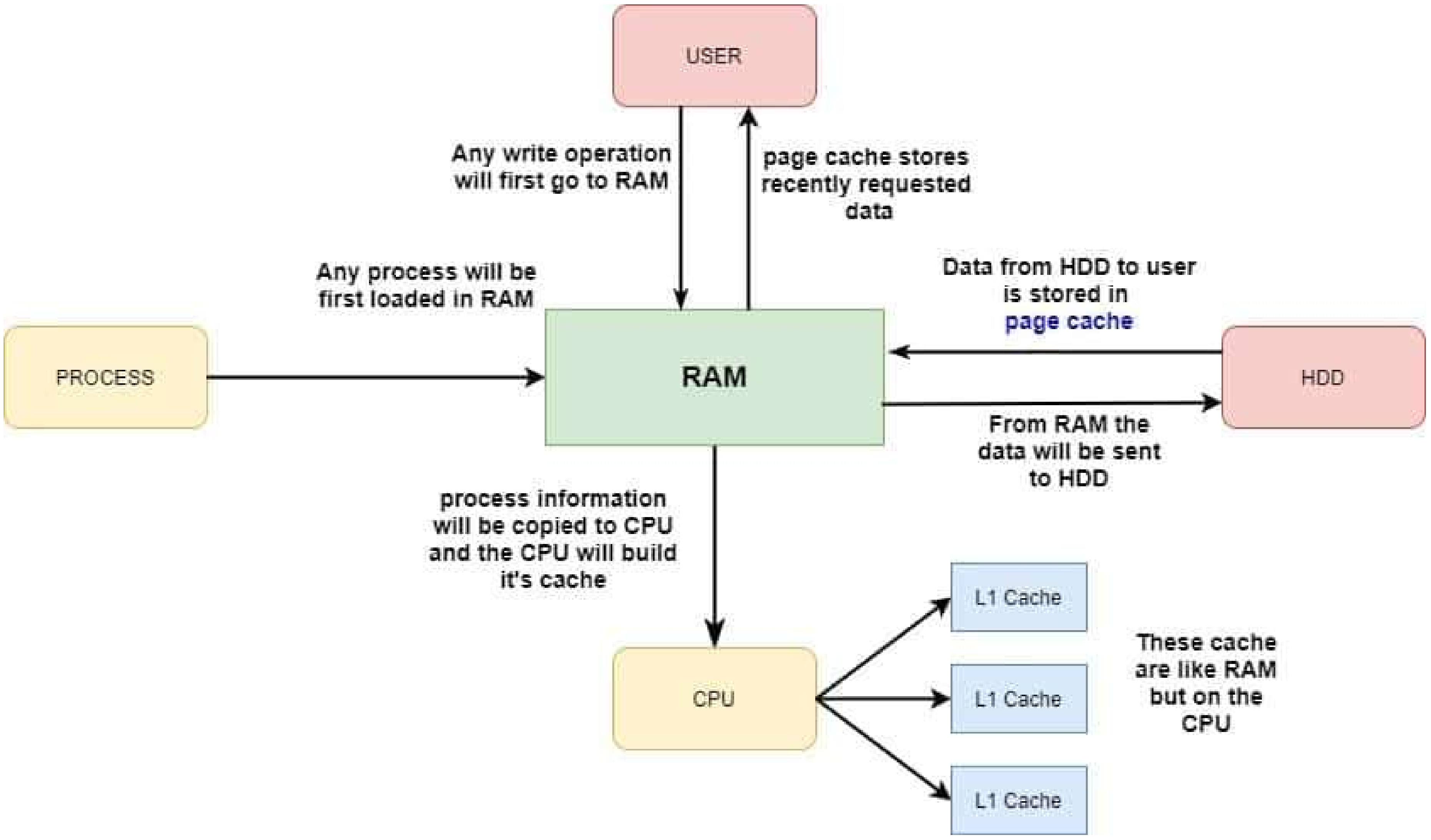
- Long critical sections (file I/O, complex ops).
- User-space & kernel-space synchronization.
- Not in interrupt context (can sleep).

Comparison Table

Feature	Mutex	Semaphore	Spinlock
Type	Binary lock	Counter-based	Busy-wait lock
Threads	1 at a time	N at a time	1 at a time
Blocking?	Sleeps	Sleeps	Spins (no sleep)
Overhead	Medium (context switch)	Medium	Low (CPU burn)
Use Case	Long critical sections	Resource pools	Very short sections
Interrupt-Safe?	No	No	Yes
Ownership	Yes	No	No

Memory Management

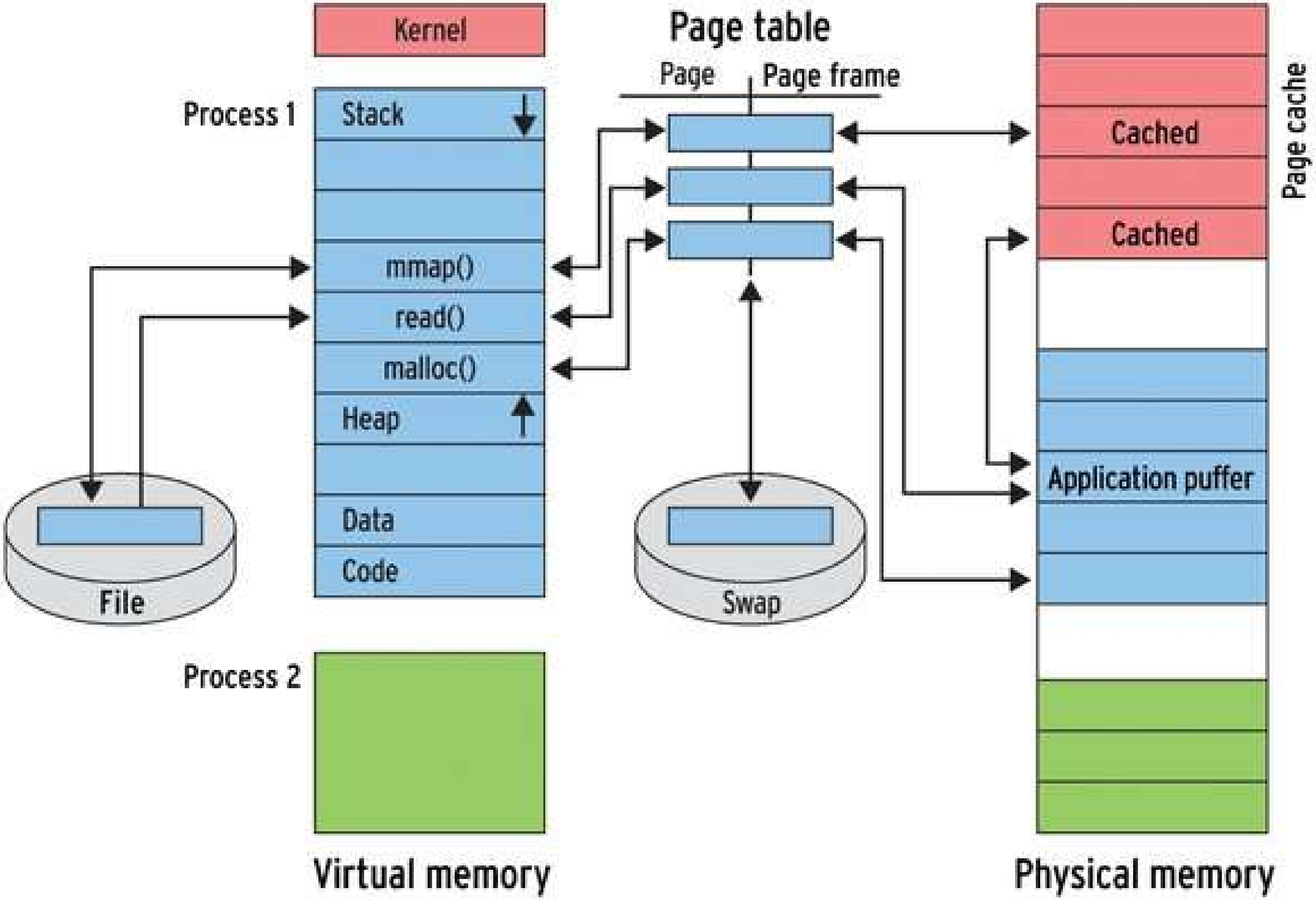
Memory Management Components on Linux


USER
Any write operation will first go to RAM
page cache stores recently requested data
Any process will be first loaded in RAM
Data from HDD to user is stored in page cache
PROCESS
RAM
HDD
From RAM the data will be sent to HDD
process information will be copied to CPU and the CPU will build it's cache
L1 Cache
L1 Cache
L1 Cache
CPU
These cache are like RAM but on the CPU

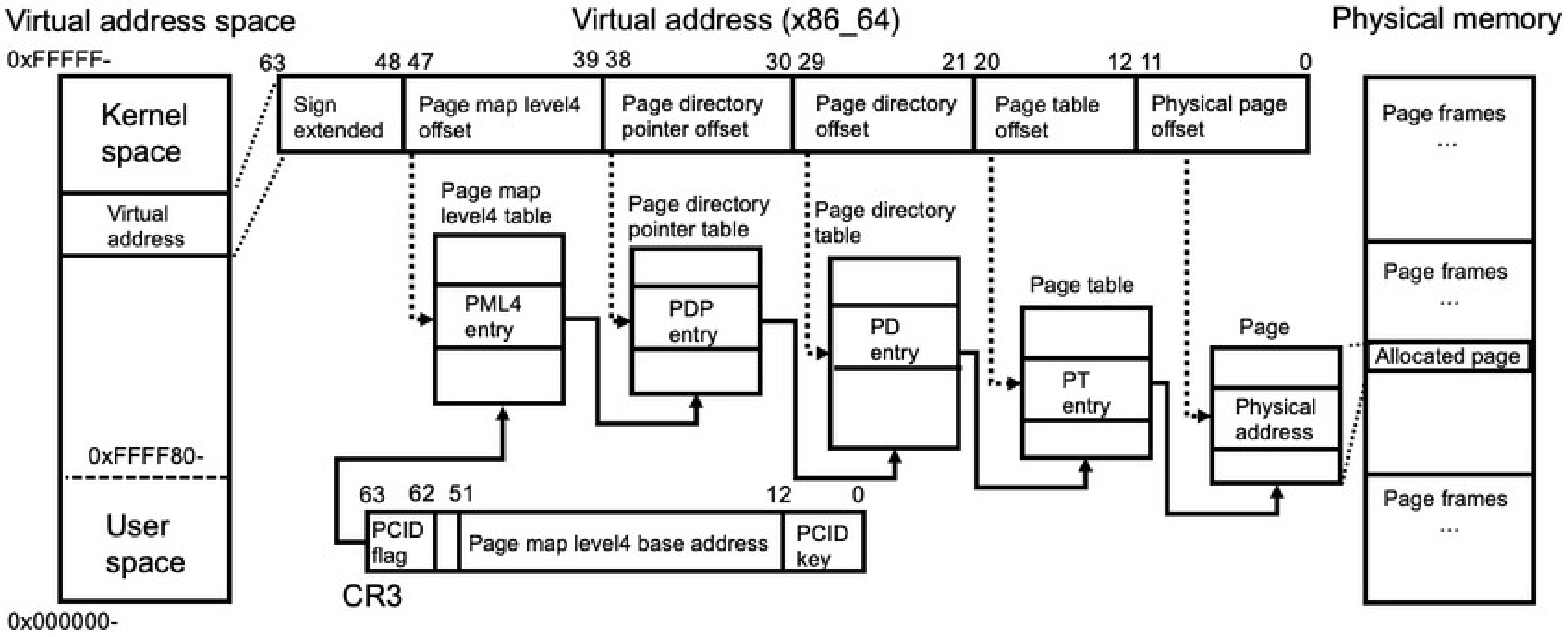
Memory Management Components on Linux

- The **Linux kernel** ensures that virtual memory addresses used by software are mapped to physical memory addresses using **process-specific page tables**
- This **mapping** allows the system to access **memory** contents efficiently
- The **virtual address space** for applications is divided into sections, including code, static data, dynamic data, and the stack
- The operating system also manages **larger page frame** areas for internal caches, such as the **page cache** which stores **special pages** involved in file access to prevent frequent disk access
- **For example**, when the system needs to read data from a disk, it first checks the page cache to see if the data is already stored in memory before accessing the slower disks

Virtual Memory



Page Table


Virtual address space
0xFFFFF-
Kernel space
Virtual address
0xFFFF80-
User space
0x000000-
Virtual address (x86_64)
63
48 47
39 38
30 29
21 20
12 11
0
Sign extended
Page map level4 offset
Page directory pointer offset
Page directory offset
Page table offset
Physical page offset
Page map level4 table
PML4 entry
Page directory pointer table
PDP entry
Page directory table
PD entry
Page table
PT entry
Page
Physical address
63
62
51
12
0
PCID flag
Page map level4 base address
PCID key
CR3
Physical memory
Page frames
...
Page frames
...
Allocated page
Page frames
...

Virtual File System

VFS (Virtual File System)

- The Virtual Filesystem (VFS) is the subsystem of the kernel that implements the filesystem-related interfaces provided to user-space programs.
- All filesystems rely on the VFS to allow them to coexist and interoperate. •
- This enables programs to use standard Unix system calls to read and write to different filesystems on different media.

VFS (Virtual File System)

- The VFS is the glue that enables system calls such as open(), read(), write(),copy() and move() to work regardless of the filesystem or underlying physical medium.
- In older operating systems (think DOS), this would never have worked.
- Modern operating systems abstract access to the filesystems via a virtual interface that such interoperation and generic access is possible.
- New filesystems and new varieties of storage media can find their way into Linux, and programs need not be rewritten or even recompiled.

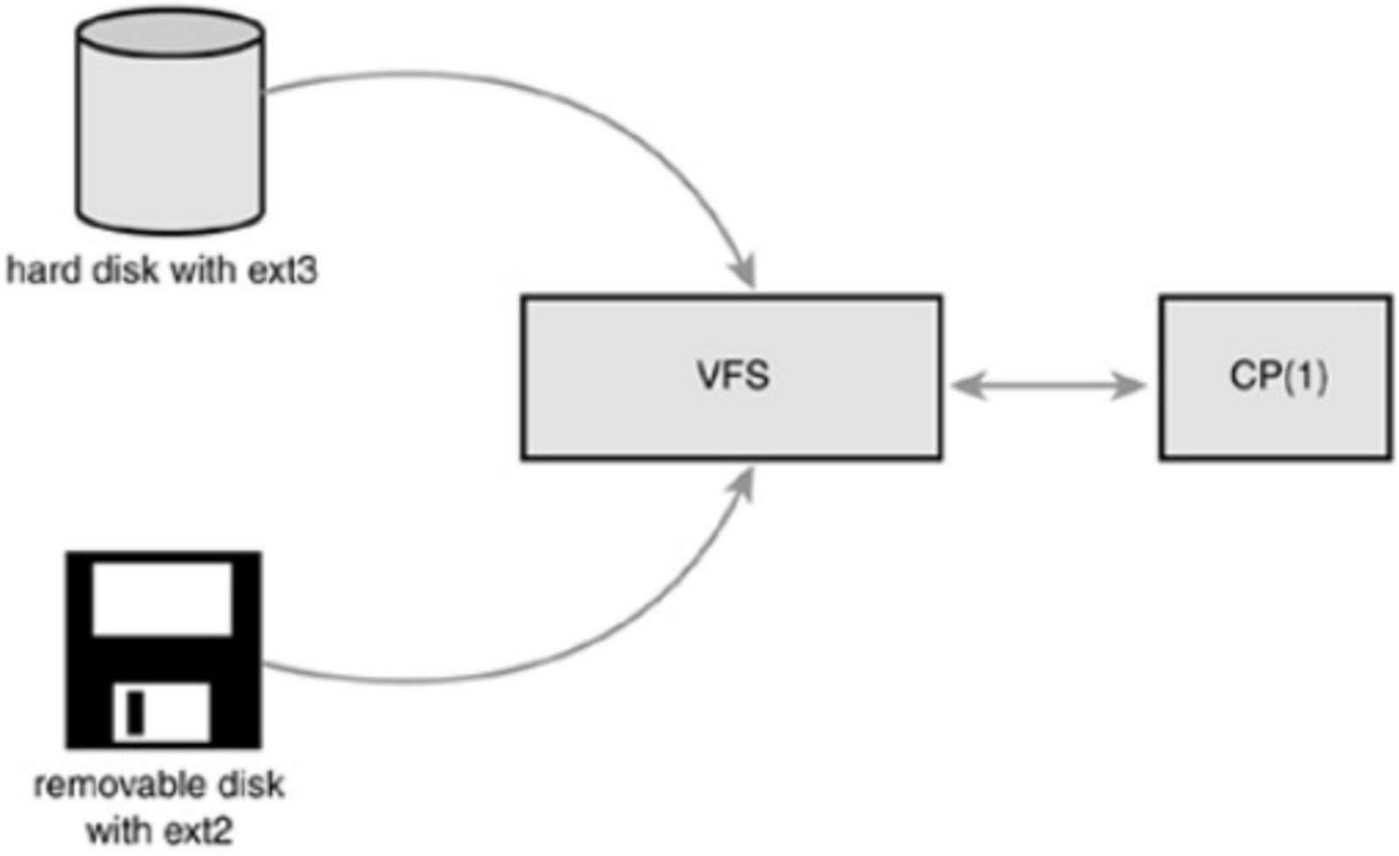
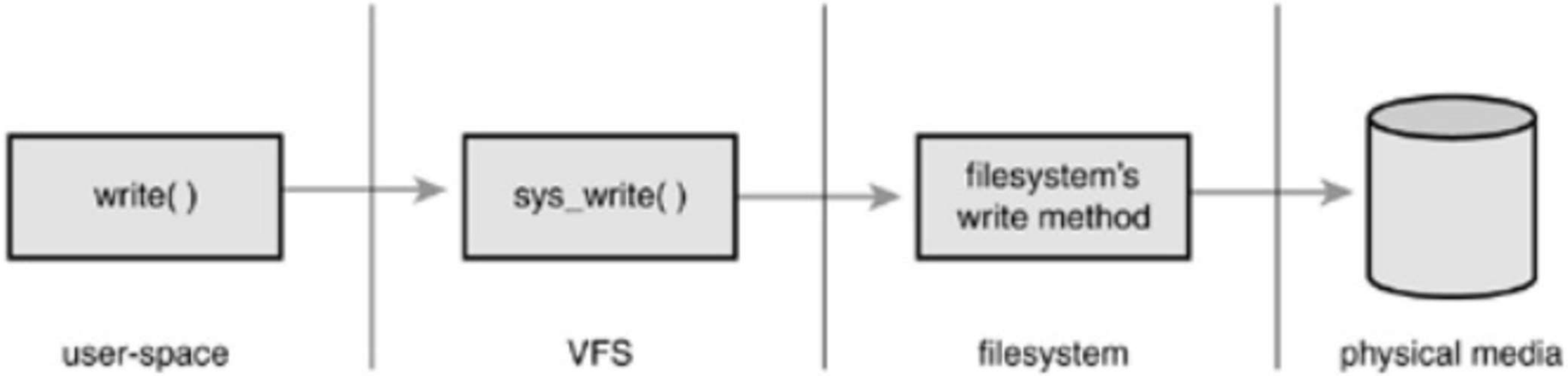


Figure1. The VFS in action: Using the cp(1) utility to move data from a hard disk mounted as ext3 to a removable disk mounted as ext2. Two different filesystems, two different media. One VFS.

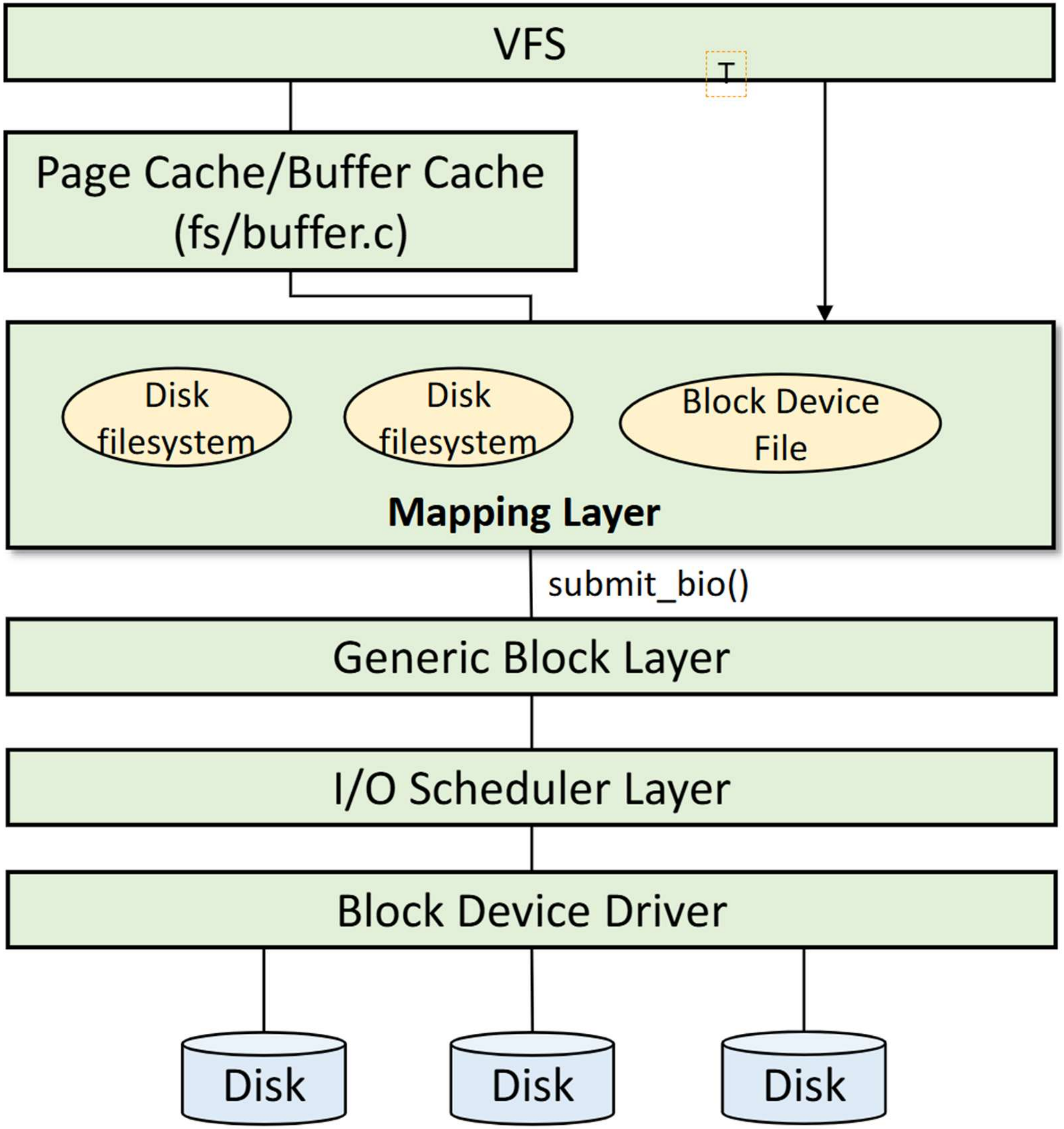
write(f, &buf, len);



For Example =>

Consider a simple user-space program that does **write(f, &buf, len);**

The flow of data from user-space issuing a write() call, through the VFS's generic system call, into the filesystem's specific write method, and finally arriving at the physical media.


VFS
T
Page Cache/Buffer Cache
(fs/buffer.c)
Disk filesystem
Disk filesystem
Block Device File
Mapping Layer
submit_bio()
Generic Block Layer
I/O Scheduler Layer
Block Device Driver
Disk
Disk
Disk

Linux Device Drivers

What is device driver

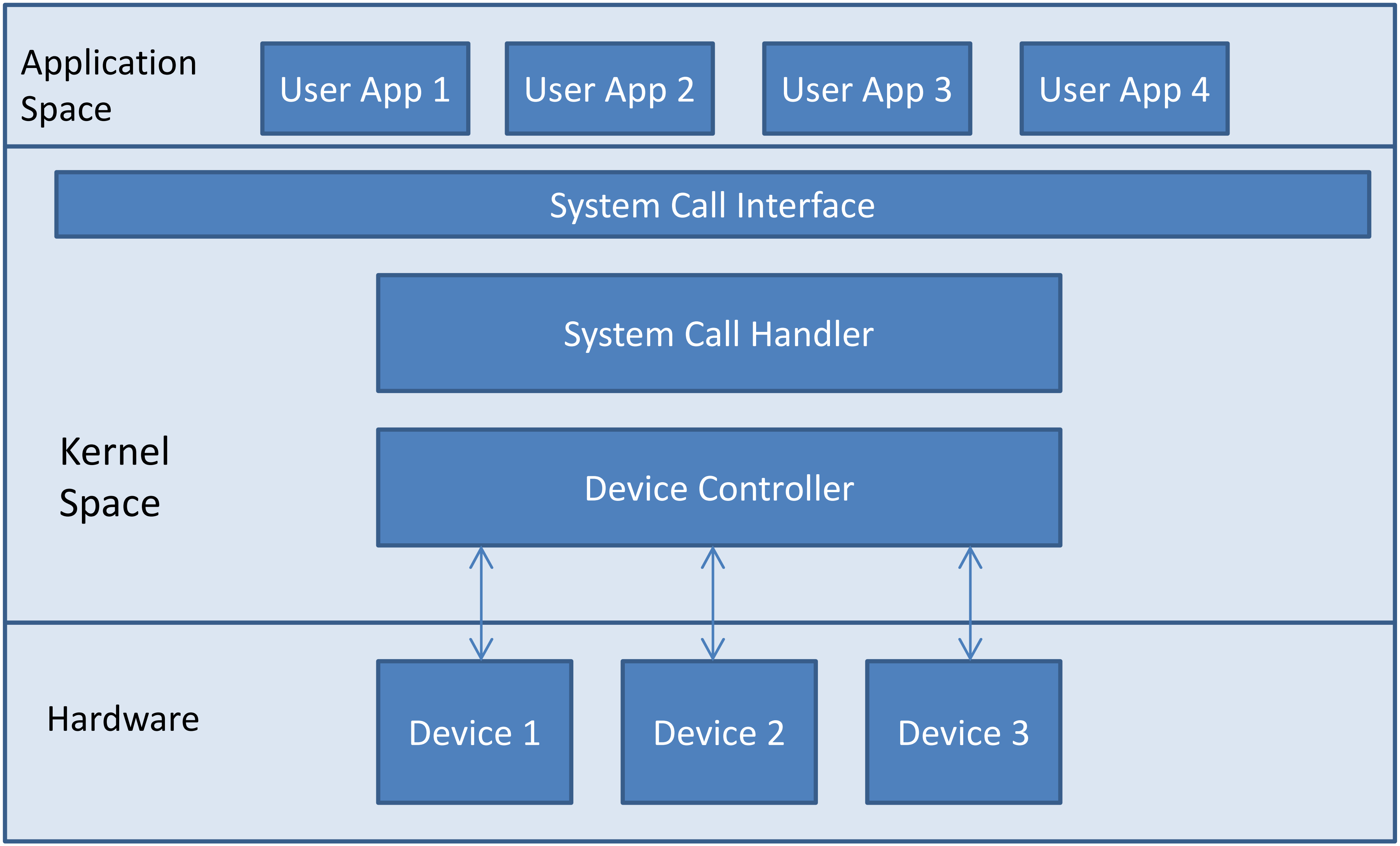
- Software layer between application & hardware
- Used to control the device control & access the data from the device
- Linux kernel must have device driver for each peripheral of the system
- Linux Device Driver
  - Present in built with kernel
  - Loadable as module in run time

Why Drivers Are Needed

Kernel cannot directly control hardware or interact with user programs.

- A driver provides the glue between:
  - User space: Apps, commands (cat, echo, etc.)
  - Kernel space: Hardware, memory, buses

Linux Device Driver Architecture



Types of Device Driver

- Character Driver
  - Can be accessed as a stream of bytes like a file
  - Examples: UART, GPIO etc.
- Block Driver
  - Device that can host a file systems like a disk
  - Handles I/O operation with one or more blocks
  - Examples: HDD, SSD etc.
- Network Driver
  - Device that any network transaction through an interface
  - Interface is in charge of sending & receiving data packets
  - Examples: Ethernet, WiFi etc.

Comparison Table

Feature	Character Drivers	Block Drivers	Network Drivers
Data Unit	Stream of bytes	Fixed-size blocks	Packets
Access Pattern	Sequential	Random	Packet-oriented
Examples	Keyboards, serial ports	Hard drives, SSDs	Ethernet cards, Wi-Fi
Device Files	/dev/ttyS0, /dev/input	/dev/sda, /dev/sdb	No device files (e.g., eth0)
System Calls	read(), write()	read(), write()	socket(), send(), recv()
Buffering	Unbuffered	Buffered	Packet-based
Use Case	Low-latency devices	Storage devices	Network communication

Driver vs Module

Kernel Module
A kernel module is any piece of code that can be loaded into or removed from the Linux kernel at runtime (using insmod/rmmod).

Device Driver

- A driver is a specific type of kernel module that knows how to:
- Communicate with hardware (like I2C, UART, GPIO, etc.)
- Or represent virtual devices (like loopback, /dev/null)
- Register with subsystems like:
  - Character device (/dev/xyz)
  - Block device (/dev/sda)
  - Platform bus / I2C bus / USB stack

“All drivers are modules, but not all modules are drivers.”

Driver vs Module

Feature	Kernel Module	Driver (Kernel Module)
Is it loadable?	Yes	Yes
Talks to hardware?	Not usually	Yes
Creates /dev/ file?	No	Yes (character/block drivers)
Registers with subsystem?	No	Yes
Needed for hardware?	Optional (demo only)	Required
Example	hello.ko	my_gpio_driver.ko

Practical : 3

Major & Minor Numbers

- Device Number represented in 32 bit
  - 12 Bits => Major Number
  - 20 Bits => Minor Number
- Device identification
  - c => character driver
  - b => block driver
- dev_t => device number representation
- MAJOR(dev_t dev);
- MINOR(dev_t dev);
- MKDEV(int major, int minor);

Allocating Device Numbers

- register_chrdev_region
  - dev_t first => beginning device number
  - count => number of contiguous device numbers
  - name => name of the device
  - 0 on success, negative error code on failure
- alloc_chrdev_region
  - dev_t *dev => output parameter on completion
  - firstminor => requested first minor
  - count => number of contiguous device numbers
  - name => name of the device
  - 0 on success, negative error code on failure

Freeing device number

- unregister_chrdev_region
  - dev_t first => beginning of device number range to be freed
  - count => number of contiguous device numbers
  - Returns void
  - The usual place to call would be in module’s cleanup

mknod

- Creates block or character special device files
- mknod [OPTIONS] NAME TYPE [MAJOR MINOR]
  - NAME => special device file name
  - TYPE
    - c, u => creates a character (unbuffered) special file
    - b => creates a block special file
    - p => creates a FIFO
  - MAJOR => major number of device file
  - MINOR => minor number of device file

File Operations

- owner => pointer to module that owns structure
- open
- release
- read
- write
- poll
- ioctl
- mmap
- llseek
- readdir
- flush
- fsync

File Ops - Example

```
struct file_operations scull_fops = {
    .owner = THIS_MODULE,
    .llseek = hello_llseek,
    .read = hello_read,
    .write = hello_write,
    .ioctl = hello_ioctl,
    .open = hello_open,
    .release = hello_release,
};
```

Char Device Registration

- struct cdev
- Allocating dynamically
  - cdev_alloc
- cdev_init
  - cdev => cdev structure pointer
  - fops => file operations structure pointer
- cdev_add
  - cdev => cdev structure pointer
  - dev_t num => first device number
  - count => number of device numbers to be associated

Char device removal

- cdev_del
  - cdev => cdev structure pointer
- cdev structure pointer should not be accessed after passing it to cdev_del

FOPS - open & release

- inode => inode structure pointer
- filp => file structure pointer
- returns 0 or error code based on open

FOPS – read & write

- filp => file structure pointer
- buff => character buffer to be written by read / read by write function
- count => number of bytes
- offp => offset
- Returns 0 or negative error code accordingly

FOPS - ioctl

- inode => inode structure pointer
- filp => file structure pointer
- cmd => ioctl command
- args => ioctl command arguements
- returns 0 or error code

copy_to_user

- to => user pointer where data to be copied
- from => kernel pointer is data source
- count => number of bytes to be copied
- Used during driver read operations

copy_from_user

- to => kernel pointer where data to be copied
- from => user pointer is data source
- count => number of bytes to be copied
- Used during driver write operation

Practical:

- Write Simple Character Driver
- Write Character Driver to control GPIO

Platform Drivers

GPIO Drivers

Linux GPIO-Pinctrl Subsystem

For an IO Port, there are two aspects that need to be configured:

1. The function setting itself,
2. The input and output settings if it is used as a GPIO.

The 1st is controlled by the Pin Controller,
and the 2nd is controlled by the GPIO Controller

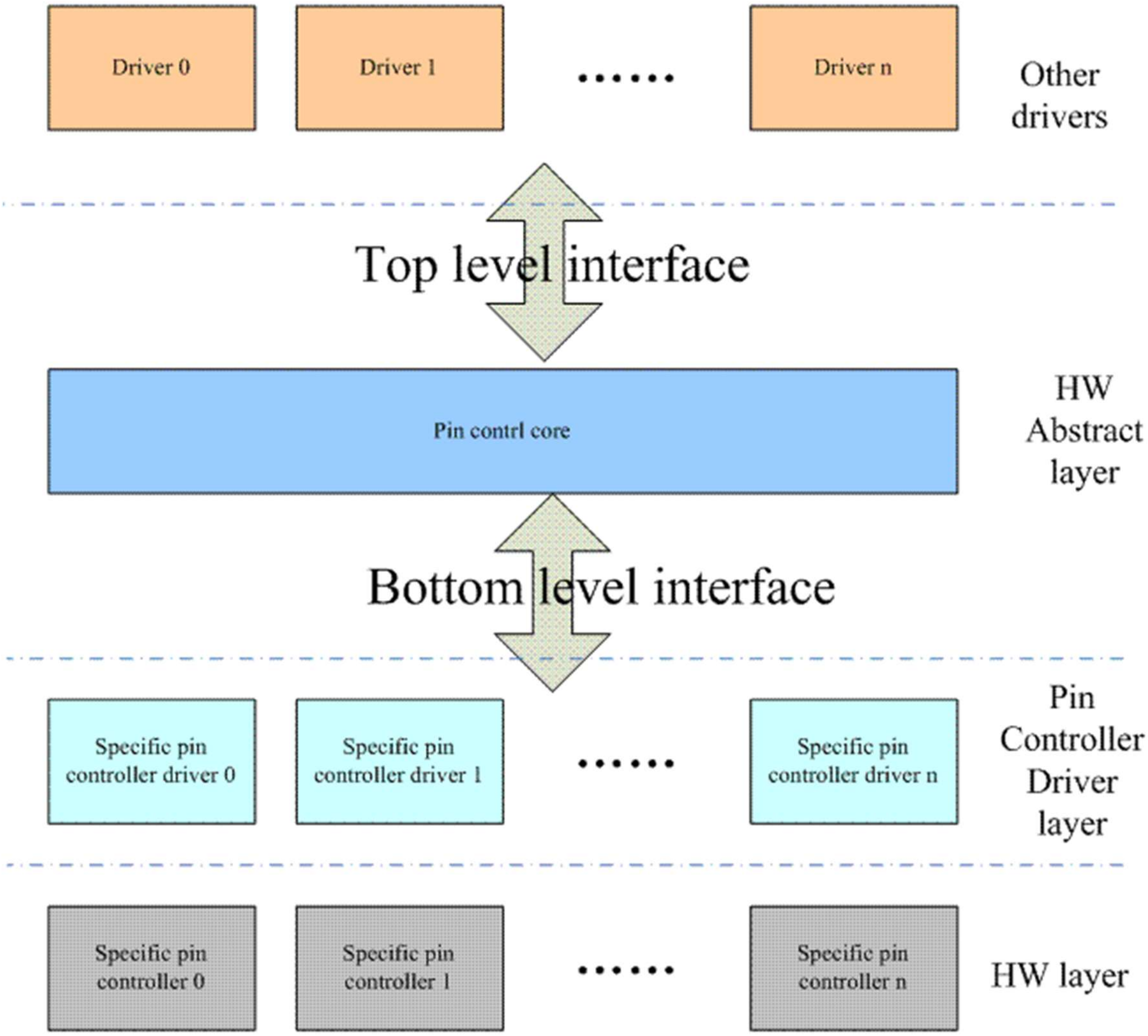
Pin Controller controls include -

- Pin function configuration. For example, whether the I/O pin is a normal GPIO or some special function pin (ADC, DAC, USB, UART, I2C SPI etc.).
- Pin characteristic configuration, such as pull-up/down resistance setting,
- GPIO drive-strength setting, etc.
- Provides abstraction for **pin groupings** and **functional mapping**.

GPIO Controller controls include -

- Configure the direction of GPIO
- If it is output, you can configure high level or low level
- If it is an input, you can get the level status on the GPIO pin
- Interfaces with the Pinctrl subsystem to ensure proper pin configuration.

Linux Pinctrl Subsystem


Driver 0
Driver 1
....
Driver n
Other drivers
Top level interface
Pin contrl core
HW Abstract layer
Bottom level interface
Specific pin controller driver 0
Specific pin controller driver 1
....
Specific pin controller driver n
Pin Controller Driver layer
Specific pin controller 0
Specific pin controller 1
....
Specific pin controller n
HW layer

Linux Pinctrl Subsystem

Top Level Interface (User Space)

- Drivers call APIs like pinctrl_select_state() to configure pins.
- include/linux/pinctrl/consumer.h
- Example: A UART driver (drivers/tty/serial/uart_core.c) requests TX/RX pin states.

HW Abstract Layer

- Manages pin states (active/sleep modes).
- Routes requests to the correct hardware driver.
- Provides abstractions for **pin groups, functions, and mappings**.
- Example : drivers/pinctrl/core.c: Manages pin states, groups, and mappings.
- Example : include/linux/pinctrl/pinctrl.h: Core data structures.
- It Translates generic requests (e.g., "set UART mode") to driver-specific ops

Pin Controller Driver Layer

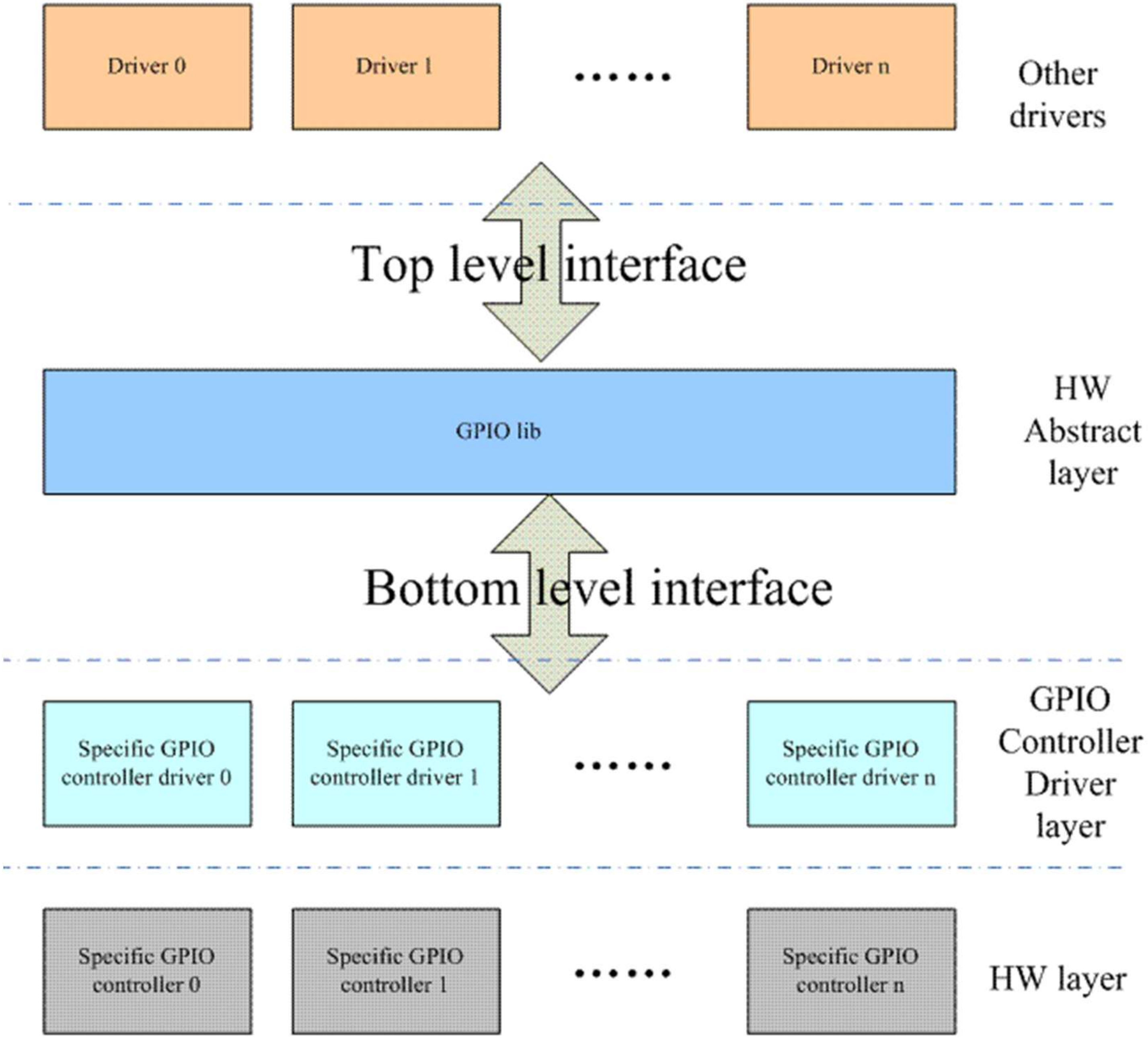
- Vendor-specific drivers (e.g., pinctrl-msm.c for Qualcomm).
- Implements struct pinctrl_ops to handle hardware registers.

HW Layer → Physical Pins

- Actual SoC pins controlled by registers (documented in **Technical Reference Manuals** for each chip).

Demo - Telechip PinCtrl Driver

Linux GPIO Subsystem


Driver 0
Driver 1
....
Driver n
Other drivers
Top level interface
GPIO lib
HW Abstract layer
Bottom level interface
Specific GPIO controller driver 0
Specific GPIO controller driver 1
....
Specific GPIO controller driver n
GPIO Controller Driver layer
Specific GPIO controller 0
Specific GPIO controller 1
....
Specific GPIO controller n
HW layer

Linux GPIO Subsystem

Consumer Drivers (Top Layer)

- Device drivers (e.g., for LEDs, buttons, sensors) request GPIO pins to control hardware.
- Uses **abstracted APIs** to avoid hardware-specific code.
- **Example**: A driver for an LED (drivers/leds/leds-gpio.c) would:
  - Request a GPIO with gpiod_get(dev, "led", GPIOD_OUT_LOW).
  - Toggle it with gpiod_set_value(led_gpio, 1).

GPIO Core (Middle Layer)

- Manages GPIO numbering, mappings, and arbitration.
- Provides the gpio_chip abstraction to bridge consumers and hardware.
- Translates consumer requests (e.g., "GPIO5") to hardware offsets.
- Calls vendor driver’s gpio_chip functions (e.g., .direction_input()).
- **Example** :
  - drivers/gpio/gpiolib.c (Core logic)
  - drivers/gpio/gpiolib-of.c (Device Tree support)

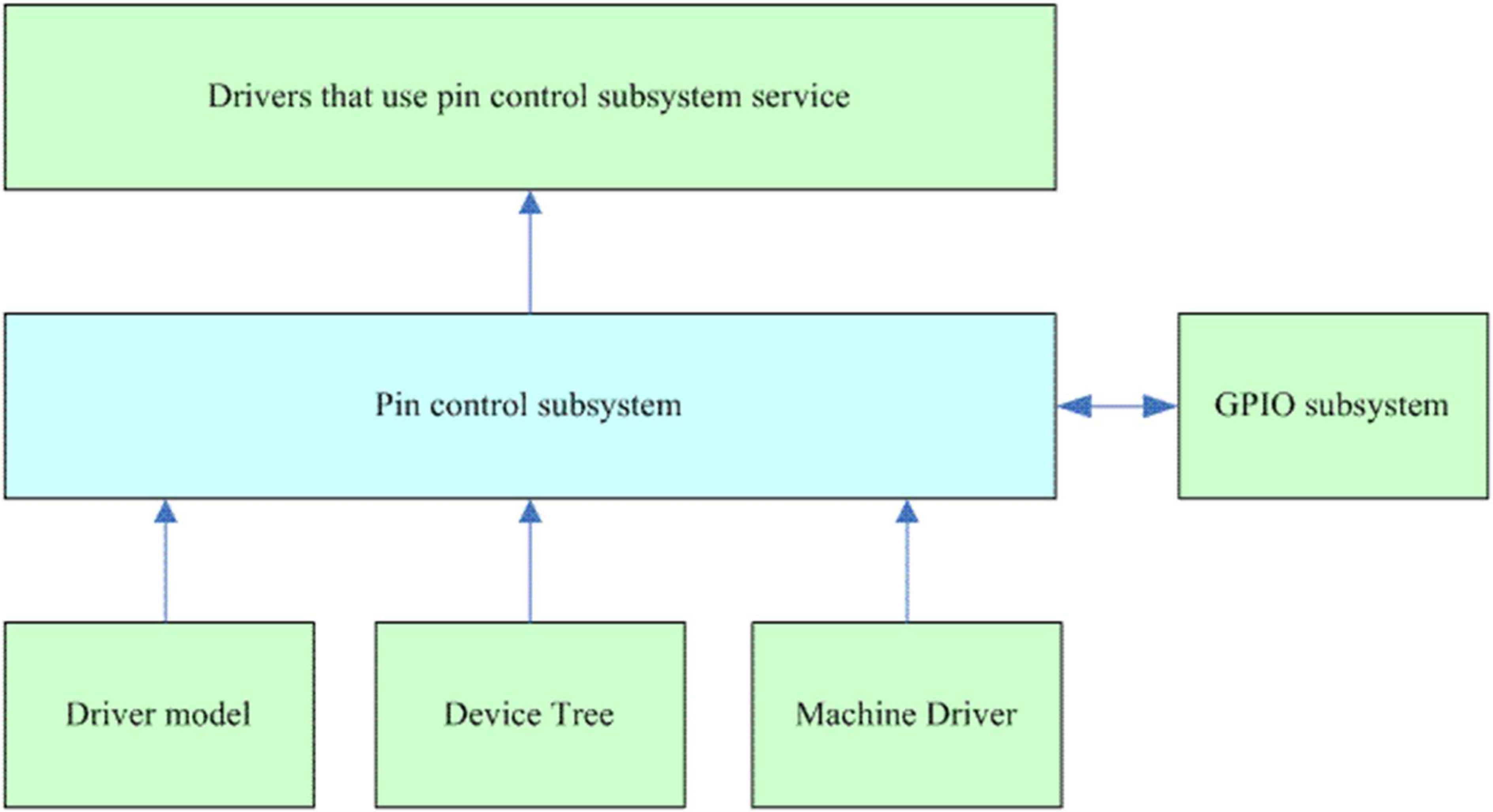
GPIO Controller Drivers (Bottom Layer)

- Implements hardware-specific operations (register read/write), Set pin direction (input/output) Configure pull-up/down resistors.
- Registers with the core via gpiochip_add().
- drivers/gpio/gpio-Telechip.c, drivers/gpio/gpio-mxc.c)

Linux Pinctrl Subsystem

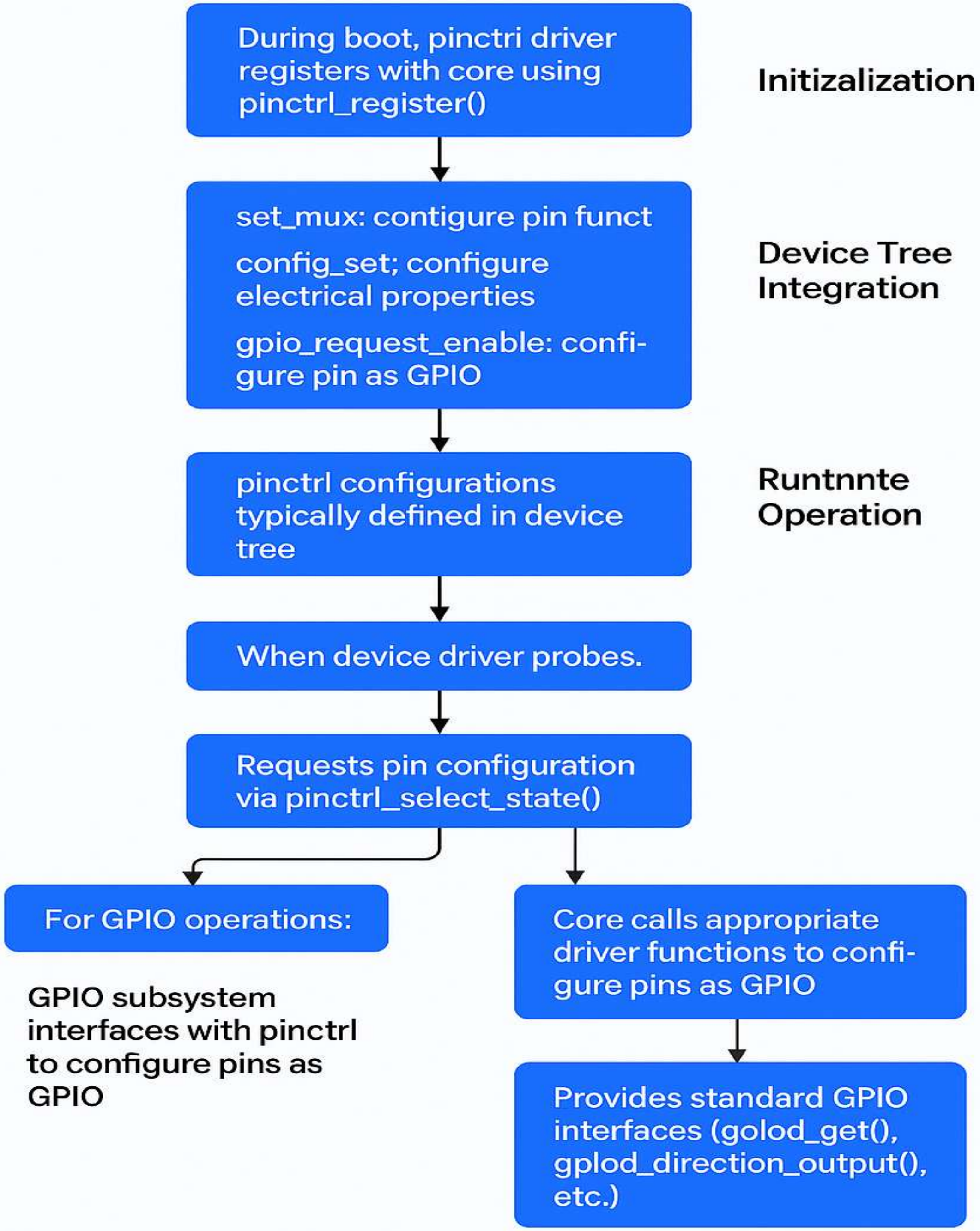
Demo - Telechip GPIO Driver

Linux GPIO + PinControl Overall Subsystem



Linux GPIO & Pinctrl Subsystem

How It Works



During boot, pinctri driver registers with core using pinctrl_register()

Initizalization

set_mux: contigure pin funct

config_set; configure electrical properties

gpio_request_enable: configure pin as GPIO

Device Tree Integration

pinctrl configurations typically defined in device tree

Runtnnte Operation

When device driver probes.

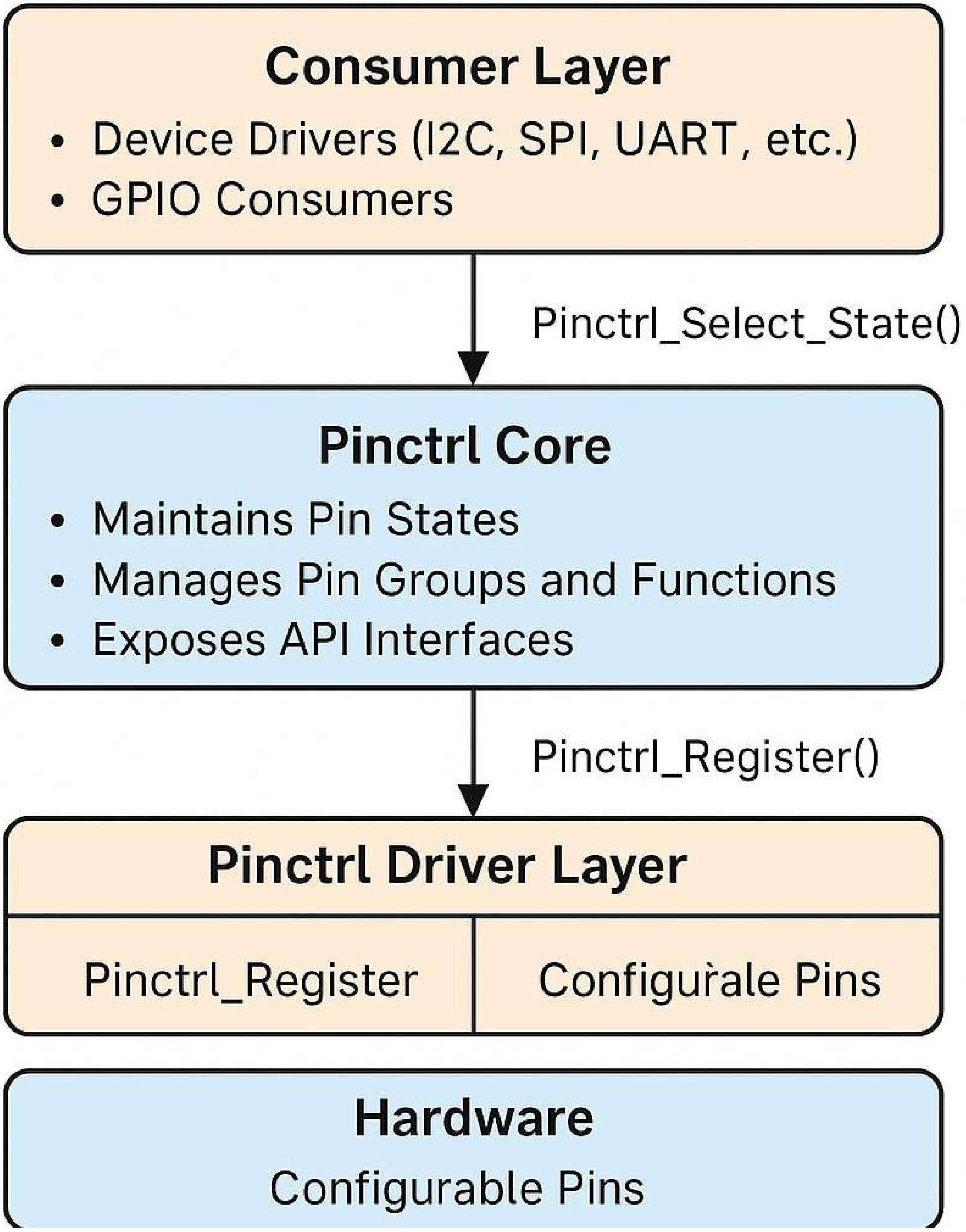
Requests pin configuration via pinctrl_select_state()

For GPIO operations:

GPIO subsystem interfaces with pinctrl to configure pins as GPIO

Core calls appropriate driver functions to configure pins as GPIO

Provides standard GPIO interfaces (golod_get(), gplod_direction_output(), etc.)


Consumer Layer
• Device Drivers (I2C, SPI, UART, etc.)
• GPIO Consumers
Pinctrl_Select_State()
Pinctrl Core
• Maintains Pin States
• Manages Pin Groups and Functions
• Exposes API Interfaces
Pinctrl_Register()
Pinctrl Driver Layer
Pinctrl_Register
Configur̀ale Pins
Hardware
Configurable Pins

How It Works

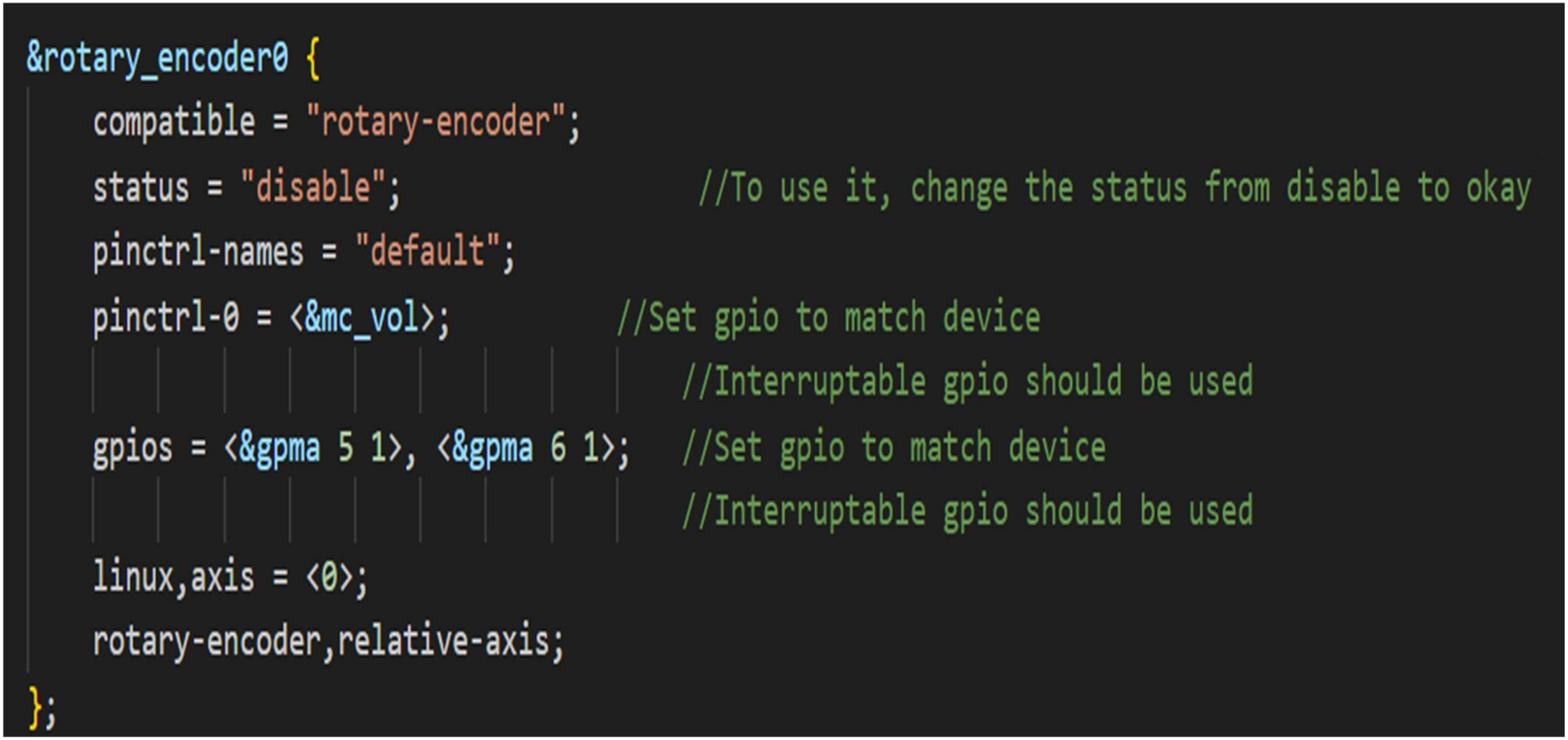
Initialization Phase

1. During boot, pinctrl driver calls pinctrl_register()

2. Provides callbacks:
   - •set_mux() for muxing
   - •config_set() for electrical settings
   - •gpio_request_enable() for GPIO config

How It Works

Device Tree Integration



```
&rotary_encoder0 {
    compatible = "rotary-encoder";
    status = "disable";                   //To use it, change the status from disable to okay
    pinctrl-names = "default";
    pinctrl-0 = <&mc_vol>;            //Set gpio to match device
                                          //Interruptable gpio should be used
    gpios = <&gpma 5 1>, <&gpma 6 1>;   //Set gpio to match device
                                          //Interruptable gpio should be used
    linux,axis = <0>;
    rotary-encoder,relative-axis;
};
```

How It Works

Runtime Operation

- Driver probes
- Calls pinctrl_select_state(dev, "default")
- Pinctrl core forwards this to the hardware driver
- GPIO pins are claimed with gpiod_get(), gpiod_direction_output(), etc.

Telechip GPIO Drivers

Examples

Linux I2C Driver

I2C Features

- Philips Semiconductors (now NXP Semiconductors, Qualcomm) developed a simple
  - Serial
  - 8-bit oriented,
  - bidirectional
  - 2-wire
  - synchronous communication protocol.
- Only 2 lines:
  - SDA (Serial Data Line)
  - SCL (Serial Clock Line)
- I2C protocol derived from System Management BUS (SMBUS – developed by INTEL)

- **I^{2}C is an acronym for the “Inter-IC” bus, a simple bus protocol which is widely used where low data rate communications suffice.**

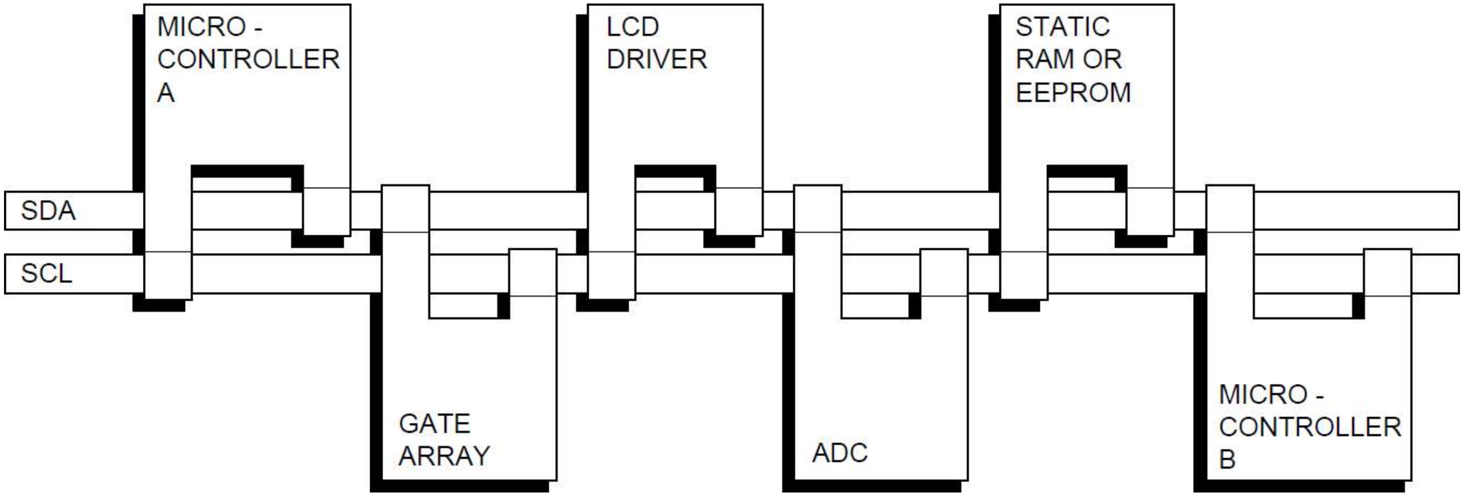
I2C Features

- I2C various Data transfer modes:
  - Standard-mode: 100kbit/s
  - Fast-mode: 400kbit/s
  - Fast-mode Plus (Fm+): 1Mbit/s
  - High-speed mode: 3.4 Mbit/s
  - Ultra Fast-mode: 5 Mbit/s (Unidirectional mode)
- Each device connected to the bus is software addressable by a unique address.
- simple master/slave relationships.
- masters can operate as master-transmitters or as master-receivers

I2C Features

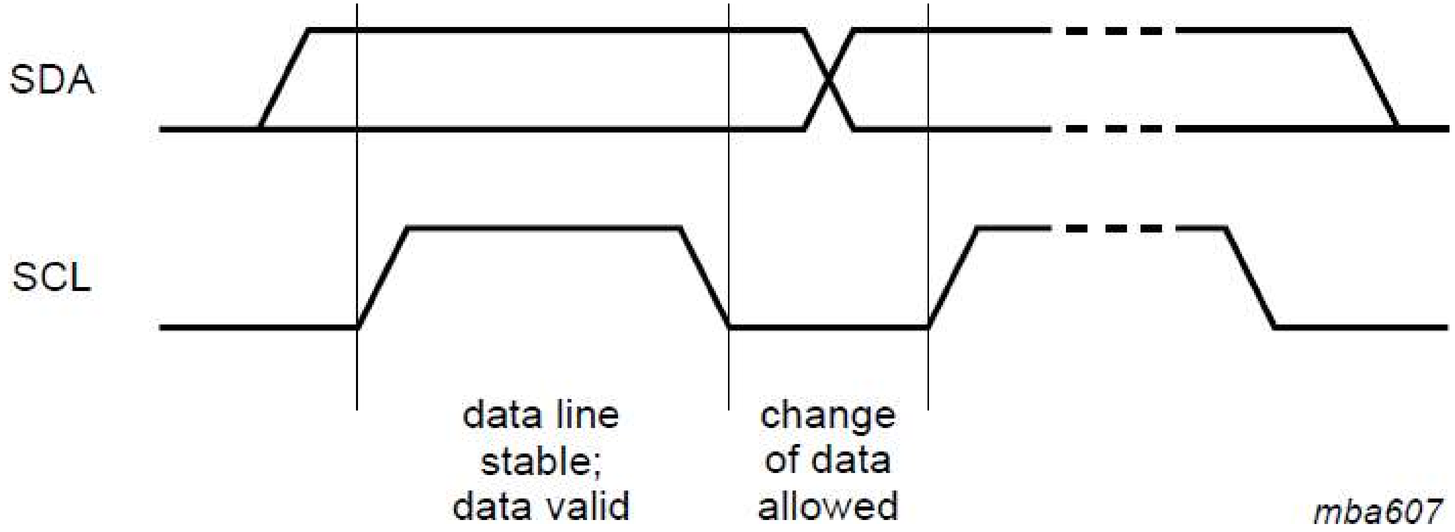
Term	Description
Transmitter	The device which sends data to the bus
Receiver	The device which receives data from the bus
Master	the device which initiates a transfer, generates clock signals and terminates a transfer
Slave	the device addressed by a master
Multi-Master	more than one master can attempt to control the bus at the same time without corrupting the message
Arbitration	procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted

I2C Bus Configuration



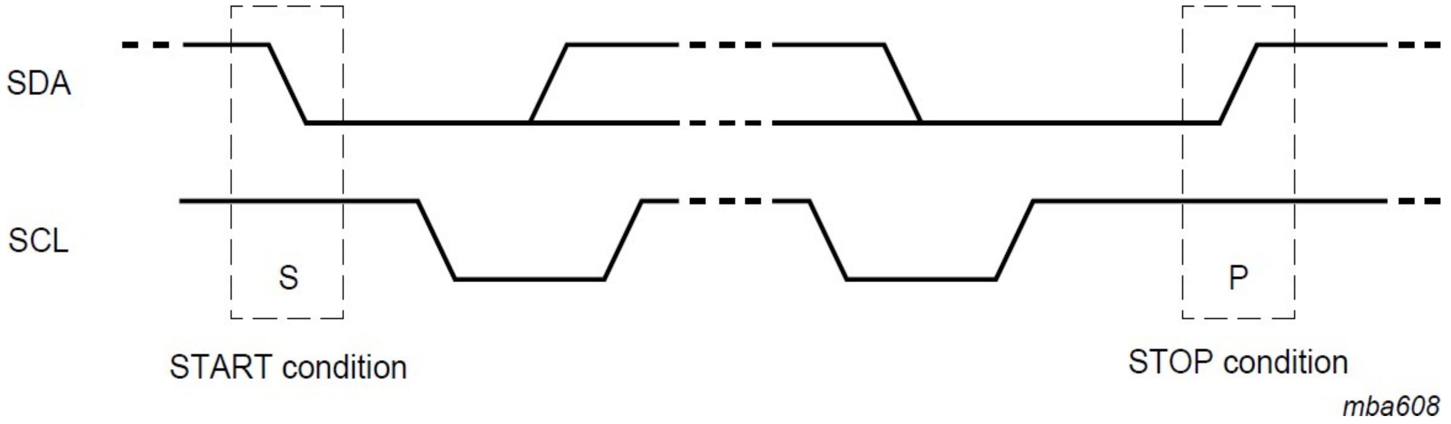
Data Validity

- The data on the SDA line must be stable during the HIGH period of the clock.
- The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW (see Figure).
- One clock pulse is generated for each data bit transferred



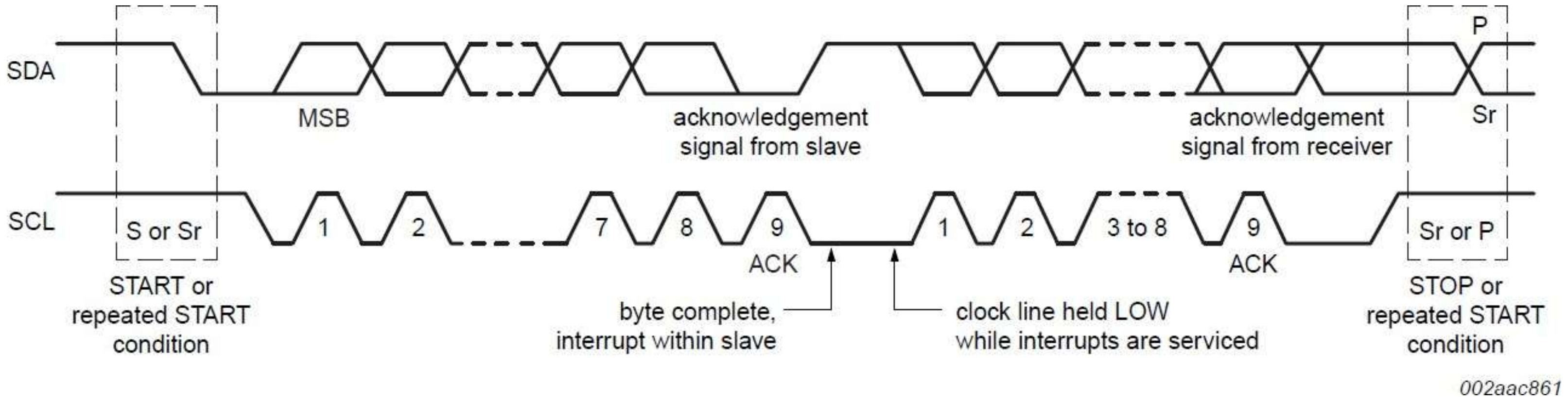
Start and Stop Conditions

- A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition.
- A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.
- START and STOP conditions are always generated by the master.



Byte Format

Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted.
Each byte must be followed by an Acknowledge bit.
Data is transferred with the Most Significant Bit (MSB) first.


SDA
SCL
P
MSB
acknowledgement signal from slave
acknowledgement signal from receiver
Sr
S or Sr
1
2
7
8
9
ACK
1
2
3 to 8
9
ACK
Sr or P
START or repeated START condition
byte complete, interrupt within slave
clock line held LOW while interrupts are serviced
STOP or repeated START condition
002aac861

Acknowledge (ACK) and Not Acknowledge (NACK)

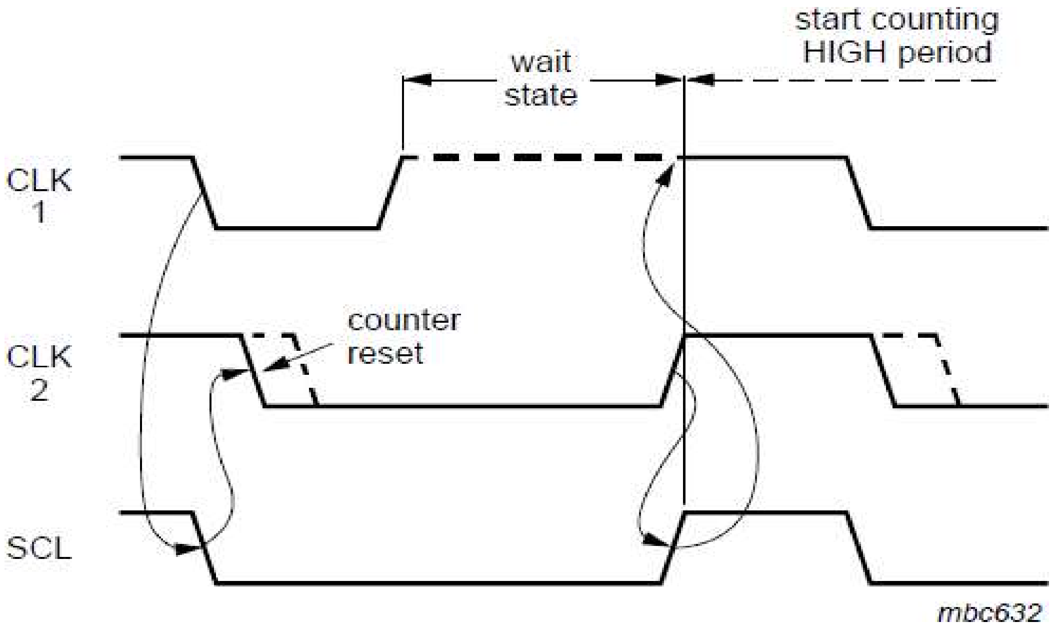
- ACK Defines: the transmitter releases the SDA line during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse.
- NACK Defines: When SDA remains HIGH during this ninth clock pulse, this is defined as the Not Acknowledge signal.

Acknowledge (ACK) and Not Acknowledge (NACK)

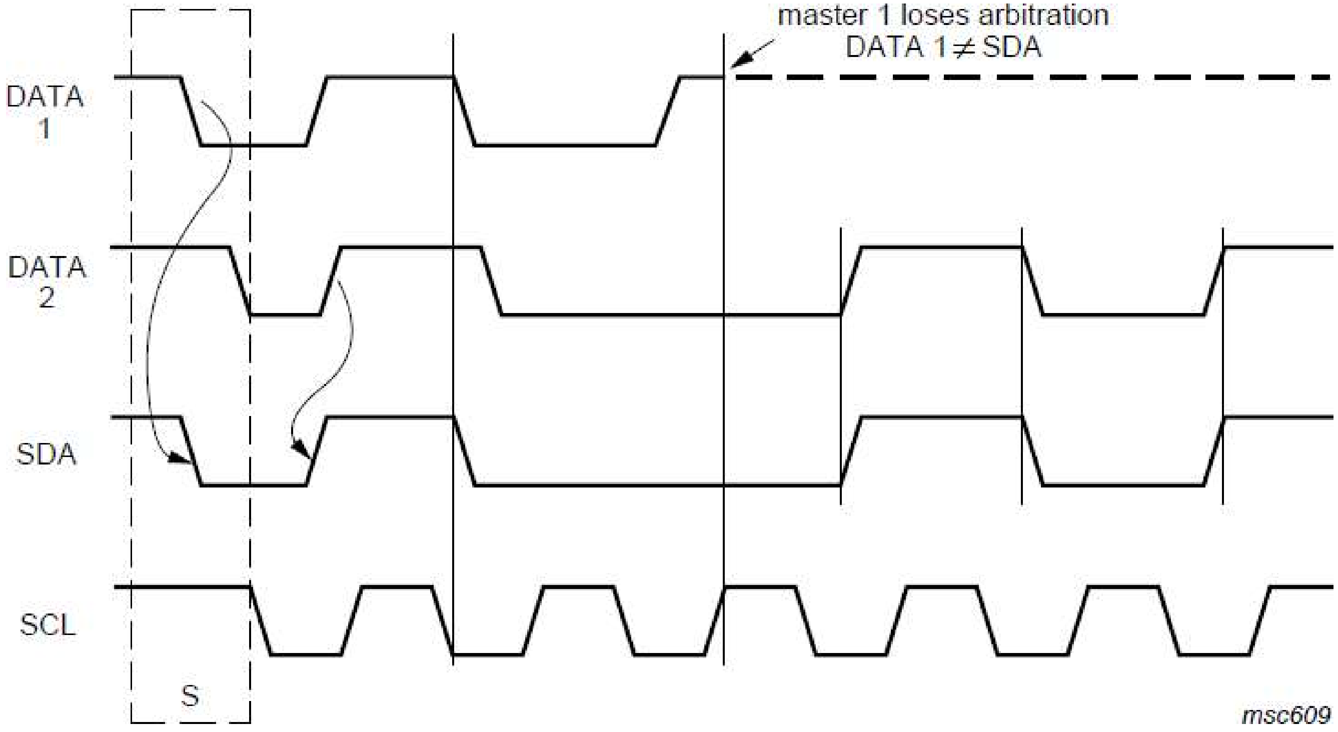
There are five conditions to the generation of a NACK:

1. No receiver is present on the bus with the transmitted address so there is no device to respond with an acknowledge.
2. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master.
3. During the transfer, the receiver gets data or commands that it does not understand.
4. During the transfer, the receiver cannot receive any more data bytes.
5. A master-receiver must signal the end of the transfer to the slave transmitter.

Clock Synchronization



Arbitration



I2C Bit Banging

I2C Bit Banging
It is hard to tell from your schematic but in some instances I2C interfaces are implemented using general purpose I/O port pins and then bit banged in software.

Sometimes the implementer may not operate the I/O pins in this configuration using an open drain methodology and this may play a factor on why an interface without pullup resistors may seem to work.

I2C FAQ

- Q. What happens if I omit the pullup resistors on I2C lines?
- A. There will be no communication on the I2C bus. At all. The MCU will not be able to generate the I2C start condition. The MCU will not be able to transmit the I2C address.

- Q. The lack of pullups is likely to damage any of those two ICs in my board?
- A. Even without the internal pull-ups, a lack of any pull-ups will not damage either IC. The internal build of i2c device SCl and SDA lines are like NPN transistors. They are Open Collectors, essentially current controlled/switched diodes.

- .

I2C FAQ

Why need pull up resistors in I2C?

- Generally you will need to have the pullup resistors for an I2C interface circuit. If the interface is truly a full spec I2C on both ends of the wires then the signal lines without the resistors will never be able to go to the high level.
- They may remain low or go to some intermediate level determined by the leakage current in the parts at each end.
- The reason for this is because true I2C is an open drain bus.
- I2C is a TTL-logic protocol; so your data and clock lines are open-drain. In other words, the I2C hardware can only drive these lines low; they are left floating when not a zero. That's where the pull-up resistors come in.
- Some devices may actually have on-chip pullup resistors in the 20K to 100K ohm range just to hold the interface pins at a high inactive level when the I2C interface on the part is not in use. For simple and short interfaces these pullup resistors may be just enough to provide the current needed to pull the lines high while clocks and/or data is being signaled.
- i2c pull up resistors allows for features like concurrent operation of more than one I2C master (if they are multi-master capable) or stretching (slaves can slow down communication by holding down SCL).

I2C Interrupts

Bus Free interrupt (BF) is generated to inform the Local Host that the I2C bus became free (when a Stop Condition is detected on the bus) and the module can initiate his own I2C transaction.

Start Condition interrupt (STC) is generated after the module being in idle mode have detected (synchronously or asynchronously) a possible Start Condition on the bus (signalized with WakeUp).

Arbitration lost interrupt (AL) is generated when the I2C arbitration procedure is lost.
No-acknowledge interrupt (NACK) is generated when the master I2C does not receive acknowledge from the receiver.

Registers-ready-for-access interrupt (ARDY) is generated by the I2C when the previously programmed address, data, and command have been performed and the status bits have been updated.
This interrupt is used to let the CPU know that the I2C registers are ready for access.

I2C Interrupts

Receive interrupt/status (RRDY) is generated when there is received data ready to be read by the CPU from the I2C_DATA register (see the FIFO Management subsection for a complete description of required conditions for interrupt generation). The CPU can alternatively poll this bit to read the received data from the I2C_DATA register.

Transmit interrupt/status (XRDY) is generated when the CPU needs to put more data in the I2C_DATA register after the transmitted data has been shifted out on the SDA pin (see the FIFO Management subsection for a complete description of required conditions for interrupt generation). The CPU can alternatively poll this bit to write the next transmitted data into the I2C_DATA register.

Receive draining interrupt (RDR) is generated when the transfer length is not a multiple of threshold value, to inform the CPU that it can read the amount of data left to be transferred and to enable the draining mechanism.

Transmit draining interrupt (XDR) is generated when the transfer length is not a multiple of threshold value, to inform the CPU that it can read the amount of data left to be written and to enable the draining mechanism.

How to Program I2C

- Module Configuration Before Enabling the Module
- Initialization Procedure
- Configure Slave Address and DATA Counter Registers
- Initiate a Transfer
- Receive Data
- Transmit Data

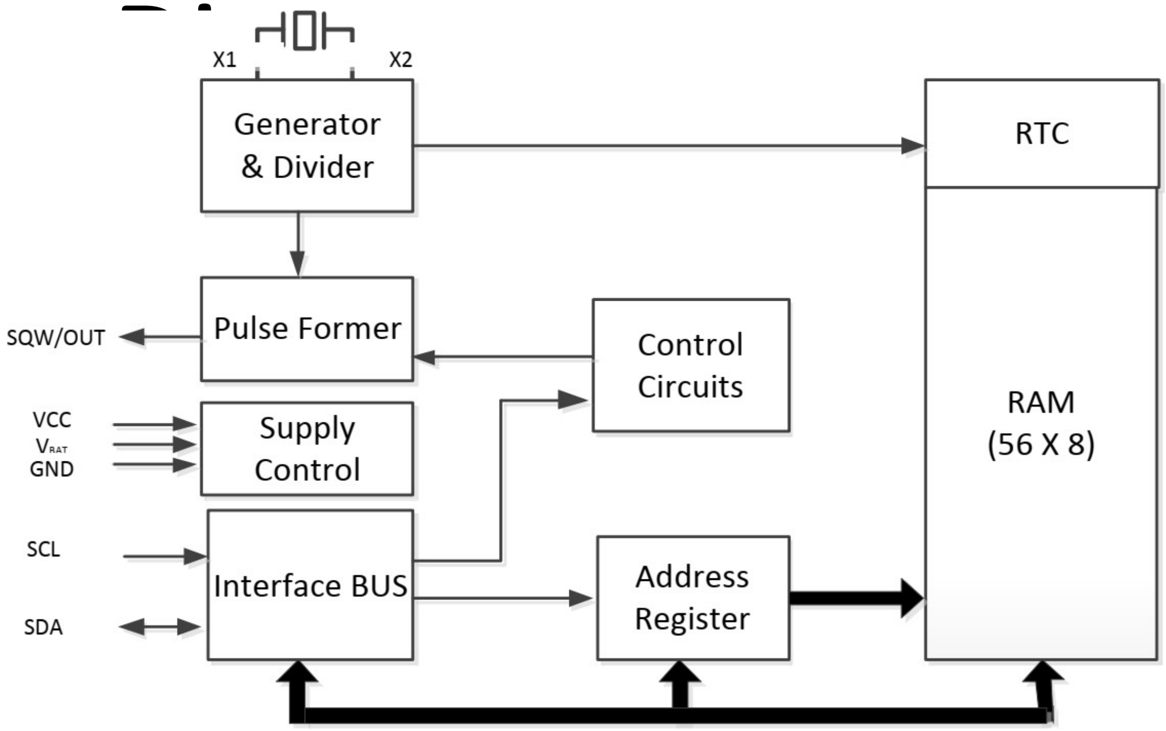
I2C Slave Device

DS1307 RTC

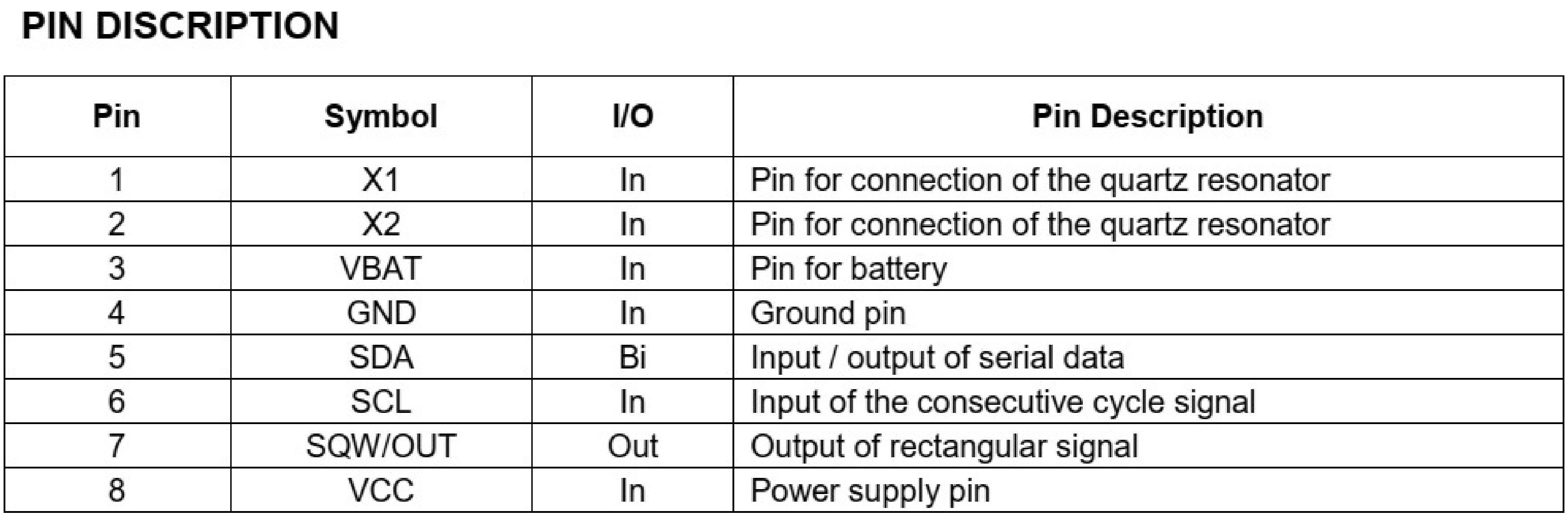
DS1307 Features/Specifications

- Count of seconds, minutes, hours, week days, date, months and years with consideration of the leap years (before 2100)
- 56 bytes of the power self-sufficient RAM for the data storage;
- Two-wire consecutive interface;
- Programmable rectangular output signal;
- Automatic determination of the supply voltage drop and the switching diagram;
- Consumption of less than 500 nA in the back-up supply mode with the operating generator;
- Temperature range of the industrial application: -40degreeCent to – +85degreecent
- Accuracy is better than ±1 minute per month

RTC Functional Block



RTC Pin Description

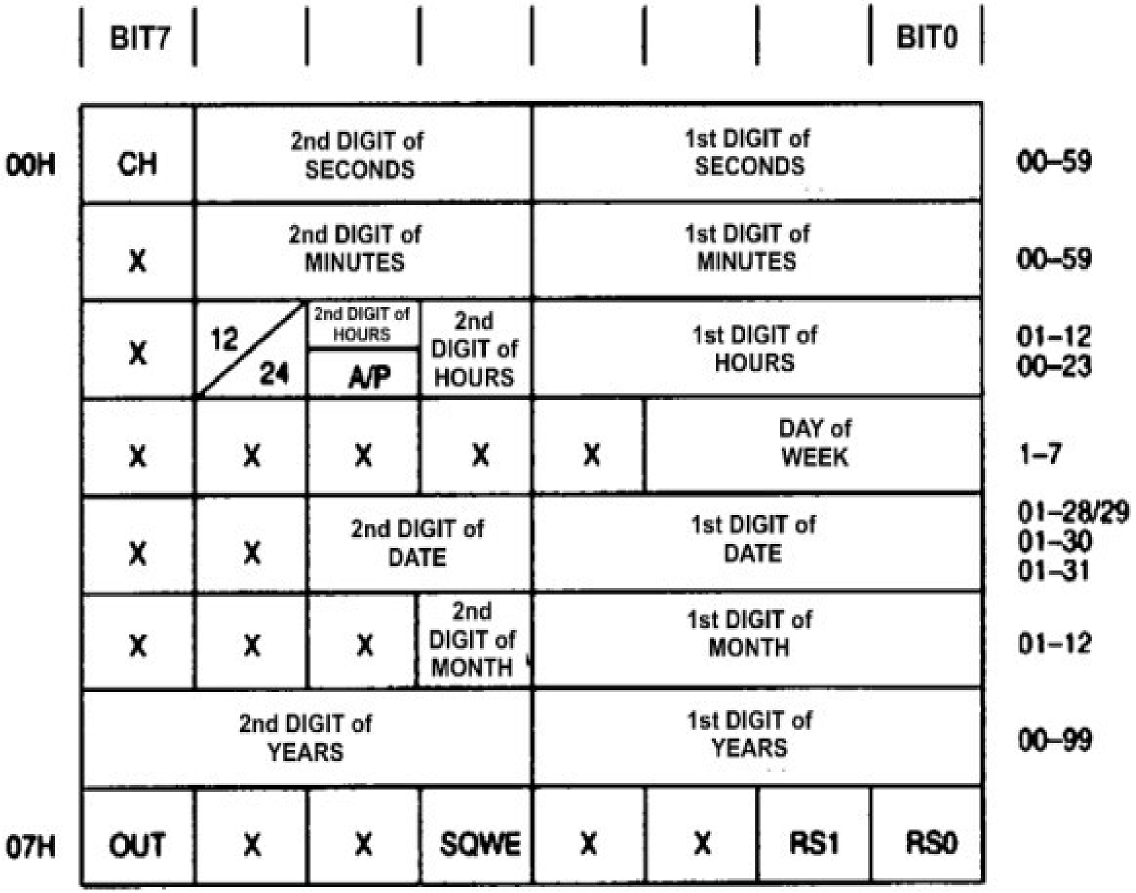


PIN DISCRIPTION

Pin	Symbol	I/O	Pin Description
1	X1	In	Pin for connection of the quartz resonator
2	X2	In	Pin for connection of the quartz resonator
3	VBAT	In	Pin for battery
4	GND	In	Ground pin
5	SDA	Bi	Input / output of serial data
6	SCL	In	Input of the consecutive cycle signal
7	SQW/OUT	Out	Output of rectangular signal
8	VCC	In	Power supply pin

RTC Register Programming Model

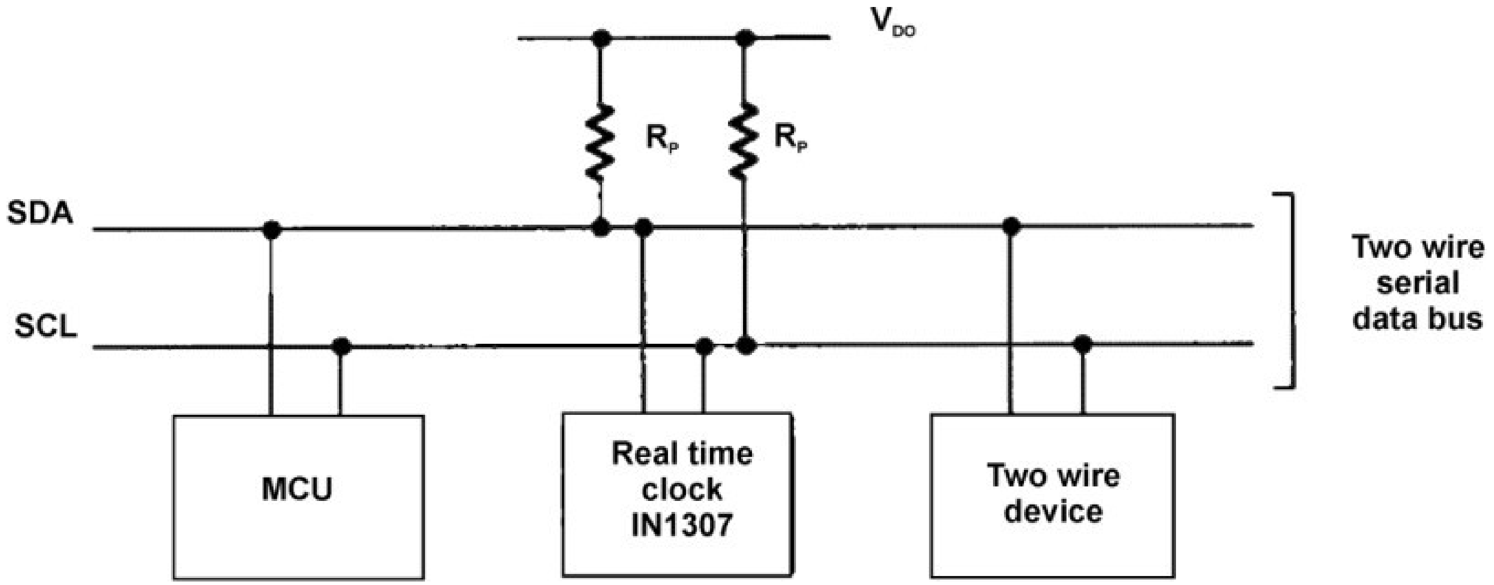
REGISTERS RTC IN1307



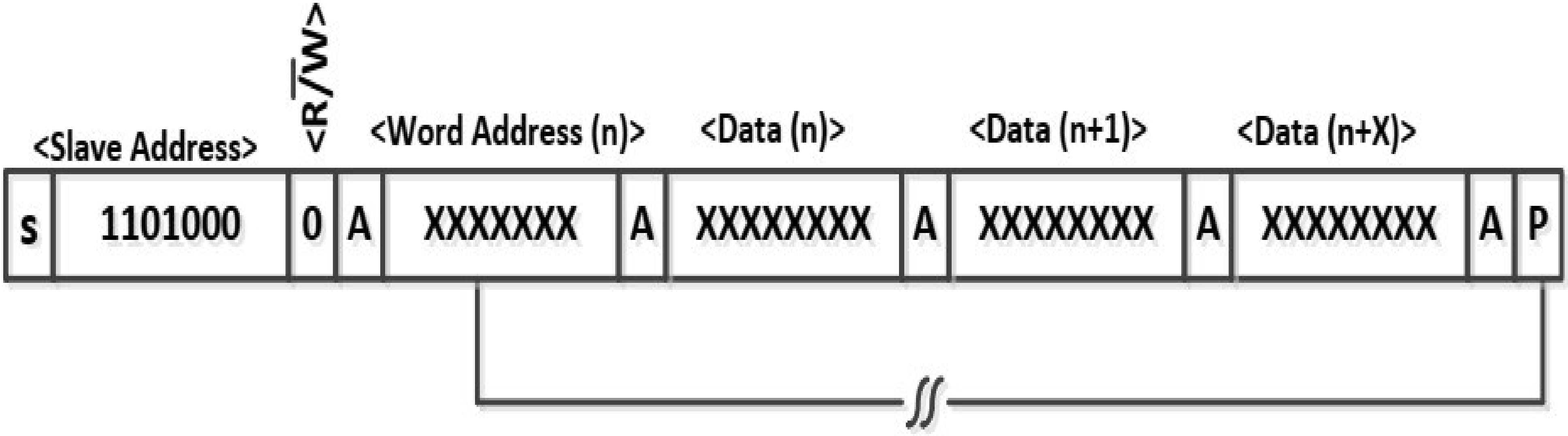
	BIT7							BIT0	
00H	CH	2nd DIGIT of SECONDS			1st DIGIT of SECONDS				00–59
	X	2nd DIGIT of MINUTES			1st DIGIT of MINUTES				00–59
	X	12 / 24	2nd DIGIT of HOURS A/P	2nd DIGIT of HOURS	1st DIGIT of HOURS				01–12 00–23
	X	X	X	X	X	DAY of WEEK			1–7
	X	X	2nd DIGIT of DATE		1st DIGIT of DATE				01–28/29 01–30 01–31
	X	X	X	2nd DIGIT of MONTH	1st DIGIT of MONTH				01–12
	2nd DIGIT of YEARS				1st DIGIT of YEARS				00–99
07H	OUT	X	X	SQWE	X	X	RS1	RS0	

RTC Interface

- I2C Comm. Protocol



RTC Write communication protocol



	<Slave Address>	<$R/\overline{W}$>		<Word Address (n)>		<Data (n)>		<Data (n+1)>		<Data (n+X)>		
S	1101000	0	A	XXXXXXX	A	XXXXXXXX	A	XXXXXXXX	A	XXXXXXXX	A	P

DATA TRANSFERRED (X+1 BYTES + ACKNOWLEDGE)

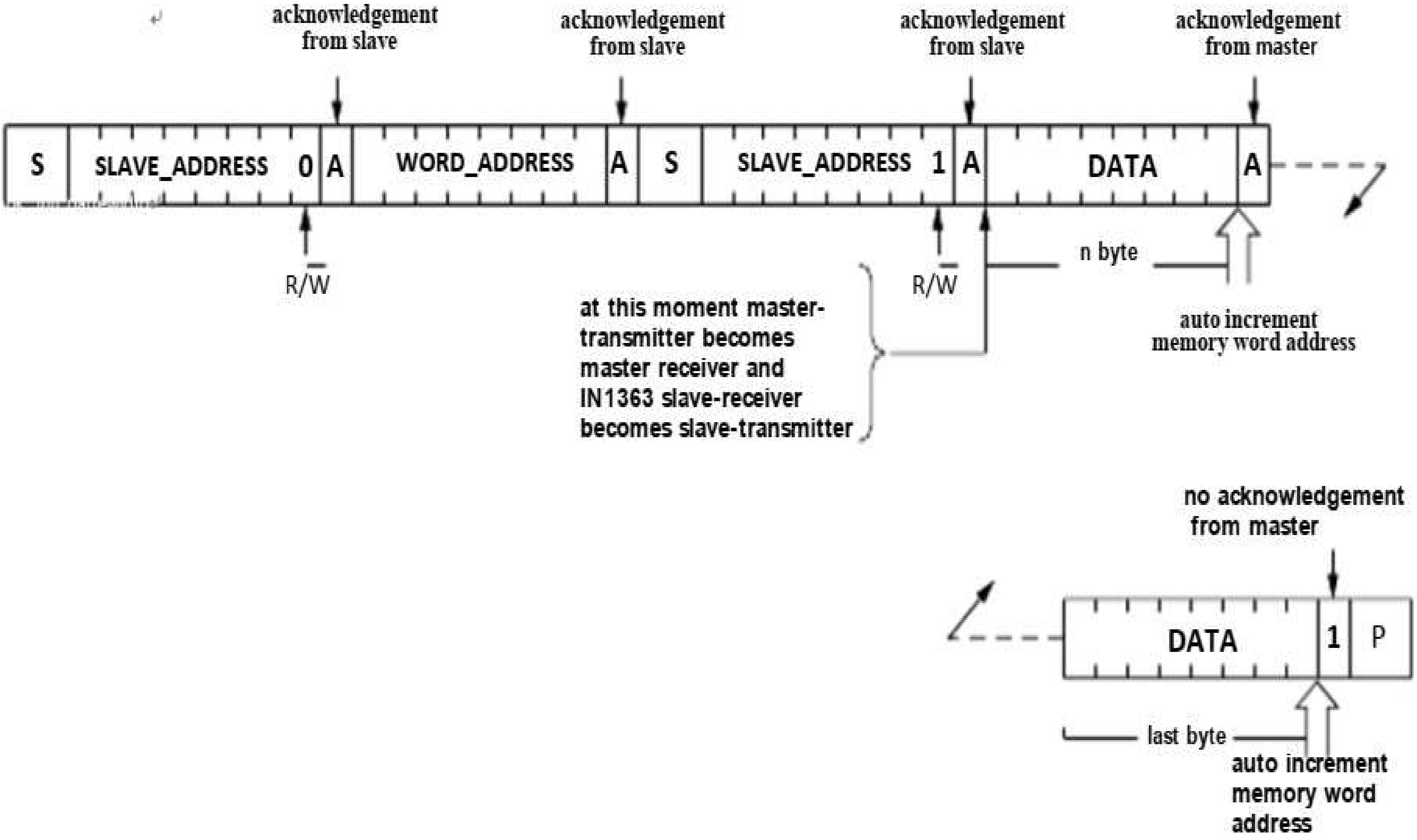
S - START

A - ACKNOWLEDGE

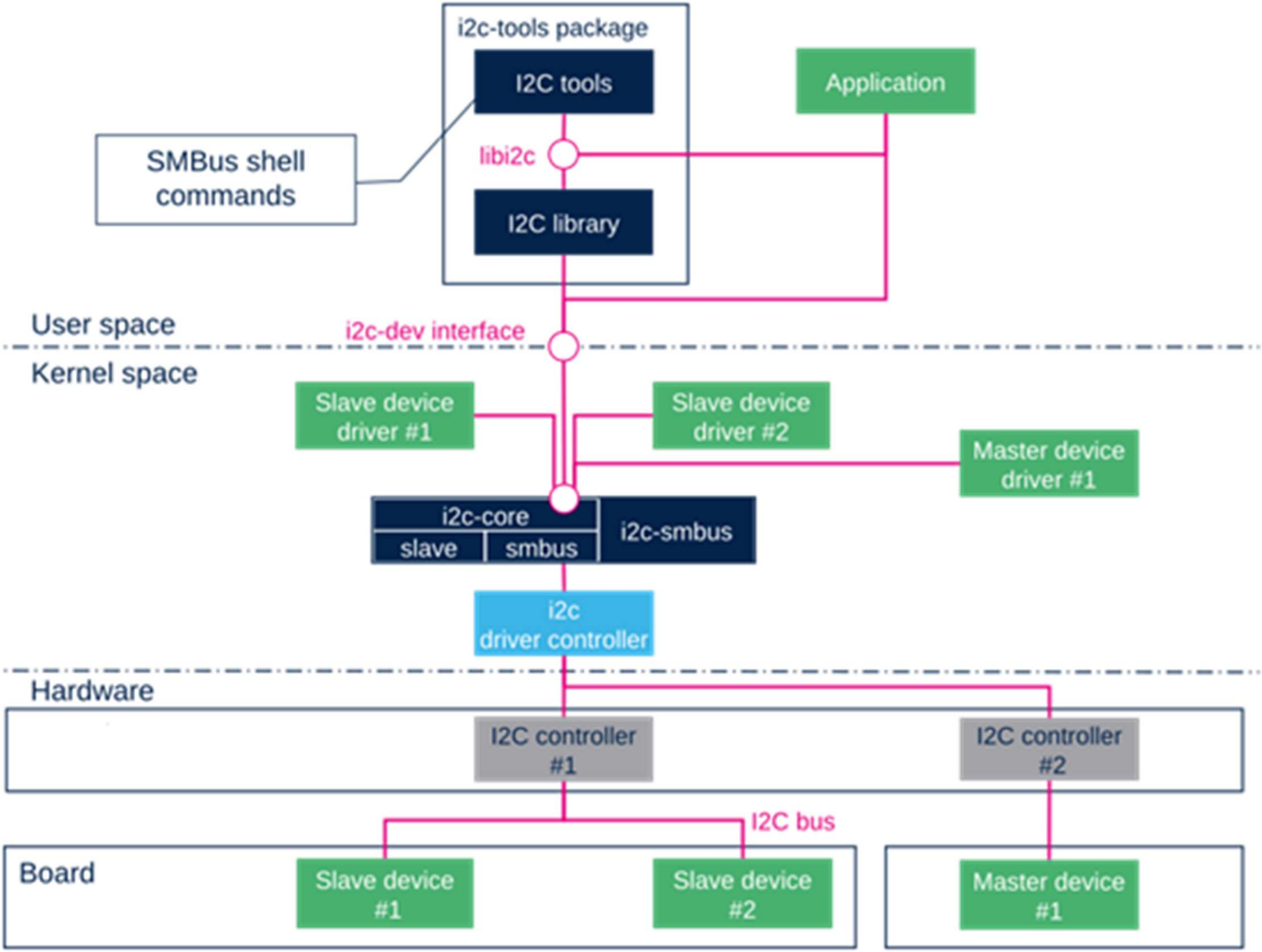
P - STOP

*$R/\overline{W}$ - READ/WRITE OR DIRECTION BIT ADDRESS = D0h

RTC Read communication protocol


acknowledgement from slave
acknowledgement from slave
acknowledgement from slave
acknowledgement from master
S
SLAVE_ADDRESS
0
A
WORD_ADDRESS
A
S
SLAVE_ADDRESS
1
A
DATA
A
R/W̄
R/W̄
n byte
at this moment master-transmitter becomes master receiver and IN1363 slave-receiver becomes slave-transmitter
auto increment memory word address
no acknowledgement from master
DATA
1
P
last byte
auto increment memory word address

Master reads after setting word address (write word address ; read data)


i2c-tools package
I2C tools
Application
SMBus shell commands
libi2c
I2C library
User space
i2c-dev interface
Kernel space
Slave device driver #1
Slave device driver #2
Master device driver #1
i2c-core
slave
smbus
i2c-smbus
i2c driver controller
Hardware
I2C controller #1
I2C controller #2
I2C bus
Board
Slave device #1
Slave device #2
Master device #1

I2C Drivers Architecture

Telechip I2C Drivers

Examples

Thank you