Introduction

This file captures the notes and important concepts extracted from book Linux Device Driver 3e, essentially acting as a quick cheat sheet Targeted for Linux 6.6 LTS — Hybrid C + Rust Reference

---

Linux Device Drivers 3rd Ed. — Modern Ready Reckoner (Part 1)

Covers Chapters 1–5 (Blocking I/O, Timers, Memory, Hardware I/O, Interrupts)

Chapter	Focus
6	Advanced blocking and non-blocking I/O (wait queues, async notifications)
7	Timers and deferred work (bottom halves, tasklets, workqueues)
8	Memory management for drivers (kmalloc, vmalloc, mmap, DMA basics)
9	Hardware I/O and port access
10	Interrupt handling (request_irq, threaded IRQs, shared interrupts)

Chapter 1 — Introduction to Device Drivers

Device drivers are the bridge between user space and hardware. They run in kernel space and expose an interface (usually via /dev/* files) that user applications can call through system calls (open, read, write, ioctl, etc.).

Key Concepts

- **Kernel vs User Space**: kernel has full access to hardware / memory; user space does not.
- **Driver Types:**
  - **Character Drivers** — byte stream devices (UART, I2C, SPI).
  - **Block Drivers** — storage devices (SD, SATA).
  - **Network Drivers** — packet-oriented.
- **Device Node:** created under /dev; represents interface to the driver.
- **Major / Minor Numbers:** kernel identifies driver and device instance.
- **Modular Kernel:** loadable modules (.ko) can be inserted or removed dynamically.

Minimal Example (C)

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

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

static void __exit hello_exit(void) {
    pr_info("Goodbye, kernel world!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dhruv");
MODULE_DESCRIPTION("Minimal example module for LDD3 reckoner");
```

Build and load

```bash
make -C /lib/modules/$(uname -r)/build M=$PWD modules
sudo insmod hello.ko
sudo rmmod hello
dmesg | tail
```

Rust Version

```rust
use kernel::prelude::*;

module! {
    type: HelloModule,
    name: b"hello_rust",
    author: b"Dhruv",
    description: b"Hello World in Rust Kernel",
    license: b"GPL",
}

struct HelloModule;

impl KernelModule for HelloModule {
    fn init() -> Result<Self> {
        pr_info!("Hello from Rust kernel world!\n");
        Ok(HelloModule)
    }
}
```

Learning Notes

- Start every driver journey with a “Hello World” module to verify build → load → unload cycle.
- Understand that each module lives in kernel address space → bugs can crash the system.

- pr_info() replaces printk(KERN_INFO ...) in newer kernels.
- All kernel code must be GPL-compatible if it exports symbols.

Real-World Reference:

Look at `/drivers/char/random.c` for an example of a basic char driver implementation.

Chapter 2 — Building and Loading Modules

Modules are built outside the main kernel tree but linked against its headers using kbuild.

Key Files

```
# Makefile
obj-m := hello.o
```

Run

```
make -C /lib/modules/$(uname -r)/build M=$PWD modules
```

Module Management Commands

Command	Action
`insmod <module.ko>`	Load module manually
`modprobe <module>`	Load module + dependencies
`rmmod <module>`	Remove module
`lsmod`	List loaded modules
`modinfo <module.ko>`	View module metadata

Signing Modules (Linux 6.6)

When Secure Boot is enabled, modules must be signed:

```
scripts/sign-file sha256 key.priv key.x509 hello.ko
```

Learning Notes

- modprobe reads /lib/modules/{ver}/modules.dep to resolve dependencies.
- MODULE_LICENSE("GPL") ensures kernel exports are accessible.

- Keep builds out-of-tree using the M= syntax to avoid polluting the kernel source.

Book Examples

Example Name	Brief Description	GitHub Source
`hello`	Simplest "Hello World" loadable module using `printk`.	martinezjavier/ldd3 – ch2/hello.c
`hello_param`	Demonstrates passing parameters via `insmod` (`module_param`).	ldd3/misc-modules/hellop.c
`export` / `use_export`	Shows how one module exports symbols for another to use.	ldd3/misc-modules/export.c

Real World:

All modern distro kernels use DKMS (Dynamic Kernel Module Support) to rebuild external modules automatically on kernel updates.

Chapter 3 — Character Drivers

Character drivers transfer data as a stream of bytes and are registered with the kernel via `cdev`.

Key Concepts

- **Device Registration:**

```
static dev_t dev;
alloc_chrdev_region(&dev, 0, 1, "simple_char");
```

- **Character Device Structure:**

```
static struct cdev c_dev;
cdev_init(&c_dev, &fops);
cdev_add(&c_dev, dev, 1);
```

- **File Operations:** Each open device file is associated with a set of callbacks (`file_operations`).

Example C Driver

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

```c
#define BUF_SIZE 128
static char device_buffer[BUF_SIZE];
static struct cdev my_cdev;
static dev_t dev;

static ssize_t my_read(struct file *filp, char __user *buf, size_t len,
loff_t *off)
{
    size_t bytes = min(len, (size_t)(BUF_SIZE - *off));
    if (copy_to_user(buf, device_buffer + *off, bytes))
        return -EFAULT;
    *off += bytes;
    return bytes;
}

static ssize_t my_write(struct file *filp, const char __user *buf, size_t
len, loff_t *off)
{
    size_t bytes = min(len, (size_t)(BUF_SIZE - *off));
    if (copy_from_user(device_buffer + *off, buf, bytes))
        return -EFAULT;
    *off += bytes;
    return bytes;
}

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read  = my_read,
    .write = my_write,
};

static int __init my_init(void)
{
    alloc_chrdev_region(&dev, 0, 1, "simple_char");
    cdev_init(&my_cdev, &fops);
    cdev_add(&my_cdev, dev, 1);
    pr_info("Registered char dev %d:%d\n", MAJOR(dev), MINOR(dev));
    return 0;
}

static void __exit my_exit(void)
{
    cdev_del(&my_cdev);
    unregister_chrdev_region(dev, 1);
    pr_info("simple_char removed\n");
}

module_init(my_init);
module_exit(my_exit);
MODULE_LICENSE("GPL");
```

Create the device node:

```
sudo mknod /dev/simple_char c <major> 0
```

Test

```
echo "abc" | sudo tee /dev/simple_char
sudo cat /dev/simple_char
```

Learning Notes

- copy_to_user / copy_from_user handle address space protection.
- Always check user-supplied lengths and pointers.
- Each char driver instance typically corresponds to one minor number.

Book Examples

Example Name	Brief Description	GitHub Source
`scull`	"Simple Character Utility for Loading Localities" — core character driver with `open`, `read`, `write`, etc.	ldd3/scull/
`scullp`, `scullc`, `sculld`, `scullv`	Variants of `scull` showing different memory management models (page-based, chunked, linked, virtual).	ldd3/scull/

Rust Kernel Analogy

```
use kernel::file::FileOperations;

struct MyChar;
impl FileOperations for MyChar {
    kernel::declare_file_operations!(read, write);
    fn read(...) -> Result<usize> { /* similar logic */ }
}
```

Chapter 4 — Debugging Techniques

When developing kernel drivers, bugs can crash the entire system — so you need safe methods to trace behavior, inspect memory, and log intelligently.

Logging and Tracing

Use the kernel's built-in logging functions (preferred over printf)

```
pr_debug("debug: x=%d\n", x);
pr_info("info: device opened\n");
pr_warn("warning: low buffer\n");
pr_err("error: invalid state\n");
```

Enable dynamic debugging for specific modules:

```
echo 'file simple_char.c +p' > /sys/kernel/debug/dynamic_debug/control
```

To disable:

```
echo 'file simple_char.c -p' > /sys/kernel/debug/dynamic_debug/control
```

Using dmesg

All pr_* output goes to the kernel ring buffer, viewable via:

```
dmesg | tail
```

To clear:

```
sudo dmesg -C
```

Kernel Oops and Backtraces

If your driver dereferences an invalid pointer or uses an uninitialized structure, you'll get an Oops message. A sample backtrace:

```
BUG: unable to handle kernel NULL pointer dereference at 00000000
IP: [<ffffffff8123b2e0>] my_read+0x14/0x80 [simple_char]
...
```

Use addr2line to map it back:

```
addr2line -e simple_char.ko 0x14
```

Kernel Probes (kprobes, ftrace, bpf)

For modern kernels, you can dynamically instrument functions.

Example: ftrace

```
echo function > /sys/kernel/debug/tracing/current_tracer
echo my_read > /sys/kernel/debug/tracing/set_ftrace_filter
cat /sys/kernel/debug/tracing/trace
```

Rust Example (for inline tracing)

```
kernel::pr_debug!("MyChar::read() called");
```

Debugging Tools Summary

Tool	Use Case
`dmesg`, `pr_info()`	Basic kernel logs
`dynamic_debug`	Enable/disable runtime debug logs
`ftrace`	Function call tracing
`kgdb`	Step-through debugging
`perf`	Performance profiling
`bpftrace`	Dynamic event tracing

Learning Notes

- Never use printf or std::println!() in kernel space — only kernel-safe logging macros.
- Keep pr_debug lines in code but disable them via config — useful later in production.
- If your driver crashes, check /var/log/kern.log or journalctl -k.
- For hardware I/O debugging, hexdump, strace, and logic analyzers are invaluable.

Book Examples

Example Name	Brief Description	GitHub Source
`faulty`	Module that deliberately crashes to demonstrate kernel debugging.	mharsch/ldd3-samples/faulty
`oops`	Demonstrates triggering oops and inspecting kernel backtraces.	jesstess/ldd3-examples/faulty

Real-World Reference:

Look at `drivers/tty/serial/serial_core.c` — a masterclass in debug logging and safe instrumentation.

Chapter 5 — File Operations Deep Dive

A driver connects with user space primarily through the file_operations structure. This structure defines how the kernel calls your driver on user actions (open, read, write, ioctl, poll, mmap).

Common File Operations

Operation	Description	User API Trigger
`.open`	Initialize or prepare device	`open()`
`.release`	Cleanup	`close()`
`.read`	Transfer data to user	`read()`
`.write`	Receive data from user	`write()`
`.unlocked_ioctl`	Custom control commands	`ioctl()`
`.poll`	Event-based readiness	`select()`, `poll()`
`.mmap`	Map device memory to user space	`mmap()`

Example: ioctl Handling

```
#define IOCTL_CLEAR _IO('a', 1)
#define IOCTL_FILL  _IOW('a', 2, int)

static long my_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    switch (cmd) {
    case IOCTL_CLEAR:
        memset(device_buffer, 0, BUF_SIZE);
        break;
    case IOCTL_FILL: {
        int val;
        if (copy_from_user(&val, (int __user *)arg, sizeof(val)))
            return -EFAULT;
        memset(device_buffer, val, BUF_SIZE);
        break;
    }
    default:
        return -EINVAL;
    }
    return 0;
}

static const struct file_operations fops = {
```

```
    .owner          = THIS_MODULE,
    .read           = my_read,
    .write          = my_write,
    .unlocked_ioctl = my_ioctl,
};
```

Test from user space:

```
int fd = open("/dev/simple_char", O_RDWR);
int val = 0x41;
ioctl(fd, IOCTL_FILL, &val);
```

Example: Non-blocking I/O & Poll

If you want your driver to support select() or poll(), you can use a wait queue.

```
DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;

static ssize_t my_read(...) {
    wait_event_interruptible(wq, flag != 0);
    flag = 0;
    return 0;
}

static __poll_t my_poll(struct file *filp, poll_table *wait) {
    poll_wait(filp, &wq, wait);
    if (flag)
        return EPOLLIN | EPOLLRDNORM;
    return 0;
}
```

From user space:

```
poll() waits until data is ready for reading
```

Learning Notes

- Always validate ioctl command numbers and argument pointers.
- Use `_IOC_DIR`, `_IOC_TYPE`, `_IOC_NR`, `_IOC_SIZE` macros to decode requests.
- Prefer `unlocked_ioctl` over legacy `ioctl` in modern kernels.
- For asynchronous event-driven design, integrate wait queues and poll methods.
- In Rust kernel drivers, these are implemented via traits under `kernel::file`.

Book Examples

Example Name	Brief Description	GitHub Source
`scullconcurrent`	Shows safe concurrent access using semaphores and spinlocks.	ldd3/scull/
`scullpipe`	Blocking I/O driver using wait queues for read/write synchronization.	ldd3/scull/pipe.c

Real-World Reference

- `/drivers/input/evdev.c` — perfect example of `poll`, `read`, and `ioctl`.
- `/drivers/tty/tty_io.c` — full implementation of file ops for terminal devices.

Summary of Part 1

Chapter	Focus	Key Concepts
1	Intro to Drivers	Kernel vs user space, module basics
2	Module Mechanics	Building, inserting, removing
3	Char Drivers	`cdev`, `file_operations`, buffers
4	Debugging	`pr_debug`, ftrace, dynamic_debug
5	File Operations	ioctl, poll, non-blocking I/O

Learning Map

1. Start with "Hello World" kernel module
2. Learn build/load cycle (insmod, modprobe)
3. Write a simple char driver (read, write)
4. Add ioctl & poll
5. Integrate debugging macros

Part 2 — Linux Device Drivers Ready Reckoner

Covers Chapters 6–10 (Blocking I/O, Timers, Memory, Hardware I/O, Interrupts)

Chapter	Focus
6	Advanced blocking and non-blocking I/O (wait queues, async notifications)
7	Timers and deferred work (bottom halves, tasklets, workqueues)
8	Memory management for drivers (kmalloc, vmalloc, mmap, DMA basics)

Chapter	Focus
9	Hardware I/O and port access
10	Interrupt handling (request_irq, threaded IRQs, shared interrupts)

Targeted for Linux 6.6 LTS — Hybrid C + Rust Reference

Chapter 6 — Blocking and Non-blocking I/O

Device drivers often need to suspend a process until a condition is met, e.g., data available for reading.

Wait Queues

Wait queues let a process sleep until a condition becomes true.

```
DECLARE_WAIT_QUEUE_HEAD(my_waitqueue);
static int flag = 0;

static ssize_t my_read(struct file *filp, char __user *buf, size_t len,
loff_t *off)
{
    wait_event_interruptible(my_waitqueue, flag != 0);
    flag = 0;
    return 0;
}
```

- `wait_event_interruptible(queue, condition)` sleeps until `condition` is true.
- Non-blocking behavior: return `-EAGAIN` if `O_NONBLOCK` is set.

Poll and Select

```
static __poll_t my_poll(struct file *filp, poll_table *wait)
{
    poll_wait(filp, &my_waitqueue, wait);
    if (flag)
        return EPOLLIN | EPOLLRDNORM;
    return 0;
}
```

User-space test:

```
fd_set fds;
FD_ZERO(&fds);
FD_SET(fd, &fds);
select(fd+1, &fds, NULL, NULL, NULL);
```

Learning Notes

- Always guard shared variables with proper synchronization (spinlocks/mutexes).
- Non-blocking I/O is critical for GUI apps or network drivers.

Book Examples

Example Name	Brief Description	GitHub Source
`sculluid`	Adds file ownership and user-based access control.	ldd3/scull/sculluid.c
`scull_access`	Demonstrates file permission control and `open` policies.	ldd3/scull/scull_access.c
`scullsingle`	Restricts driver to single/multiple open semantics.	ldd3/scull/

Chapter 7 — Timers and Deferred Work

Kernel allows deferring work to be done later:

- Timers: Schedule function at specific time.
- Tasklets: Run in softirq context, non-blocking, quick execution.
- Workqueues: Kernel thread context, can sleep.

Timer Example

```
#include <linux/timer.h>
static struct timer_list my_timer;

void timer_callback(struct timer_list *t)
{
    pr_info("Timer fired!\n");
}

static int __init my_init(void)
{
    timer_setup(&my_timer, timer_callback, 0);
    mod_timer(&my_timer, jiffies + msecs_to_jiffies(2000));
    return 0;
}
```

Tasklet Example

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

void tasklet_func(unsigned long data) {
    pr_info("Tasklet executed: %lu\n", data);
```

```
}

DECLARE_TASKLET(my_tasklet, tasklet_func, 42);
tasklet_schedule(&my_tasklet);
```

Workqueue Example

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

static void my_work_func(struct work_struct *work) {
    pr_info("Workqueue executed\n");
}
static DECLARE_WORK(my_work, my_work_func);
schedule_work(&my_work);
```

Learning Notes

- **Tasklets:** Cannot sleep, run in interrupt context.
- **Workqueues:** Can sleep, run in process context.
- Timers are suitable for delayed actions; tasklets/workqueues for bottom halves.

Book Examples

Example Name	Brief Description	GitHub Source
`scullm`	Demonstrates mapping kernel memory to user space with `mmap`.	ldd3/scull/mmap.c

---

Chapter 8 — Memory Management

Drivers allocate memory for buffers, device structures, and DMA.

kmalloc / kfree

```
char *buf = kmalloc(128, GFP_KERNEL);
if (!buf) return -ENOMEM;
kfree(buf);
```

- `GFP_KERNEL` may sleep, suitable for process context.
- `GFP_ATOMIC` must be used in interrupt context.

Lookaside Cache (kmem_cache)

```c
struct kmem_cache *my_cache;
my_cache = kmem_cache_create("my_cache", sizeof(struct my_obj), 0,
SLAB_HWCACHE_ALIGN, NULL);
void *obj = kmem_cache_alloc(my_cache, GFP_KERNEL);
kmem_cache_free(my_cache, obj);
```

vmalloc

Allocates contiguous virtual memory, may not be physically contiguous:

```c
void *vbuf = vmalloc(4096);
vfree(vbuf);
```

Per-CPU Variables

```c
DEFINE_PER_CPU(int, my_counter);
this_cpu_inc(my_counter);
```

Learning Notes

- Always free memory in the same context you allocated.
- DMA buffers may need dma_alloc_coherent.
- Avoid vmalloc in high-performance, frequently accessed paths.

Book Examples

Example Name	Brief Description	GitHub Source
`kalloc`	Demonstrates `kmalloc`, `vmalloc`, and slab allocator use.	mharsch/ldd3-samples/memory

---

Chapter 9 — Hardware I/O

Device drivers communicate through I/O ports or memory-mapped I/O.

I/O Port Access

```c
#include <asm/io.h>
outb(0xFF, 0x378); // write byte to parallel port
unsigned char val = inb(0x378); // read byte
```

Check allocation:

```
request_region(0x378, 3, "my_port");
release_region(0x378, 3);
```

Memory-Mapped I/O

```
#include <linux/io.h>
void __iomem *reg;
reg = ioremap(0xFE000000, 0x100);
iowrite32(0x1234, reg);
u32 val = ioread32(reg);
iounmap(reg);
```

Learning Notes

- Use memory barriers (wmb(), rmb()) to prevent reordering.
- Always request/release resources to avoid conflicts.
- For ISA / legacy ports, check /proc/ioports.

Book Examples

Example Name	Brief Description	GitHub Source
`short`	Simulated hardware I/O driver using the parallel port for I/O access and interrupt simulation.	ldd3/short/
`shortprint`	Debug printing version of `short` for port I/O visibility.	ldd3/short/shortprint.c

---

Chapter 10 — Interrupt Handling

Interrupts allow devices to signal the CPU asynchronously.

Basic IRQ Handling

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

static irqreturn_t my_irq_handler(int irq, void *dev_id)
{
    pr_info("IRQ %d triggered\n", irq);
    return IRQ_HANDLED;
}
```

```
request_irq(17, my_irq_handler, IRQF_SHARED, "my_irq", &my_dev);
free_irq(17, &my_dev);
```

- **Top-half:** quick execution, non-blocking.
- **Bottom-half:** deferred work (tasklets, workqueues).

Shared IRQs

- Must specify `IRQF_SHARED`.
- Use unique `dev_id`.
- Return `IRQ_NONE` if the interrupt was not for your device.

Learning Notes

- Avoid sleeping in top-half (use GFP_ATOMIC if allocating memory).
- Use tasklets/workqueues for longer processing.
- Use procfs or sysfs to monitor interrupts (/proc/interrupts).

Book Examples

Example Name	Brief Description	GitHub Source
`shortirq`	Demonstrates interrupt handling and shared IRQs.	ldd3/short/shortirq.c
`shortprintirq`	Interrupt handler with detailed event logging.	ldd3/short/shortprintirq.c

---

Summary of Part 2

Chapter	Focus	Key Points
6	Blocking I/O	Wait queues, poll/select, non-blocking I/O
7	Timers & Deferred Work	Timers, tasklets, workqueues, bottom halves
8	Memory Management	kmalloc, vmalloc, mempools, per-CPU variables
9	Hardware I/O	Ports, memory-mapped I/O, barriers
10	Interrupts	request_irq, top/bottom halves, shared IRQs

Part 3 — Linux Device Drivers Ready Reckoner

Chapters 11–15 (Kernel Types, Data Structures, USB & Serial Drivers, Synchronization)

Chapter	Focus
11	Kernel Data Types & Endianness

Chapter	Focus
12	Linked Lists & Data Structures
13	USB Drivers & Endpoints
14	I2C, SPI, UART Drivers (Character Devices)
15	Concurrency & Synchronization (Spinlocks, Semaphores, RCU)

Chapter 11 — Kernel Data Types & Endianness

Kernel provides explicitly-sized data types for portability.

Standard Types

```
u8  a; // unsigned 8-bit
u16 b; // unsigned 16-bit
u32 c; // unsigned 32-bit
u64 d; // unsigned 64-bit
```

- Defined in `<linux/types.h>`
- `size_t` and `ssize_t` for memory/IO sizes

Endianness Macros

```
#include <asm/byteorder.h>
u32 val = cpu_to_le32(0x12345678);
u32 v2  = le32_to_cpu(val);
```

Pointers and Error Values

- Many kernel functions return ERR_PTR(-errno) instead of NULL for richer error reporting.
- Use IS_ERR() and PTR_ERR() to handle.

Learning Notes

- Always prefer explicitly sized types in drivers for portability.
- Be careful with cross-architecture data transfer (USB, network, storage).
- Avoid assumptions about pointer size — use unsigned long where necessary.

Chapter 12 — Linked Lists & Data Structures

Linux provides doubly-linked circular lists in `<linux/list.h>`:

```
struct my_node {
    int data;
    struct list_head list;
};

LIST_HEAD(my_list);
```

Operations

```
struct my_node *n = kmalloc(sizeof(*n), GFP_KERNEL);
n->data = 42;
list_add(&n->list, &my_list);

struct my_node *tmp;
list_for_each_entry(tmp, &my_list, list) {
    pr_info("Value: %d\n", tmp->data);
}
```

Learning Notes

- List operations are **not thread-safe;** use spinlocks if needed.
- `list_for_each_entry_safe` allows deletion during iteration.

Chapter 13 — USB Drivers & Endpoints

USB devices communicate through endpoints grouped into interfaces.

USB Types

- CONTROL — setup & configuration
- BULK — large data transfer (storage devices)
- INTERRUPT — small, periodic (keyboards)
- ISOCHRONOUS — audio/video streaming

Example USB Skeleton

```
static int my_probe(struct usb_interface *intf, const struct usb_device_id
*id) {
    pr_info("USB device connected\n");
    return 0;
}
```

```
static void my_disconnect(struct usb_interface *intf) {
    pr_info("USB device disconnected\n");
}

static struct usb_driver my_driver = {
    .name = "my_usb",
    .id_table = my_id_table,
    .probe = my_probe,
    .disconnect = my_disconnect,
};

module_usb_driver(my_driver);
```

USB Request Blocks (URBs)

- Allocate with `usb_alloc_urb()`
- Submit with `usb_submit_urb()`
- Callback on completion for async transfers

Learning Notes

- Use `usb_control_msg()` for simple requests.
- URBs are essential for async bulk/interrupt transfers.
- Sysfs exposes USB device info (`/sys/bus/usb/devices`).

Chapter 14 — I2C, SPI, UART Drivers (Character Devices)

These are specialized char drivers interfacing with serial buses.

I2C Example

```
struct i2c_client *client;
i2c_smbus_write_byte_data(client, reg, value);
int val = i2c_smbus_read_byte_data(client, reg);
```

SPI Example

```
struct spi_device *spi;
u8 tx[4] = {0x01,0x02,0x03,0x04};
struct spi_transfer t = {
    .tx_buf = tx,
    .len = sizeof(tx),
};
spi_sync_transfer(spi, &t, 1);
```

UART Example

- Often exposed via `tty_driver`
- Use `struct uart_port` + `struct uart_driver`
- Supports blocking/non-blocking read/write

Learning Notes

- These drivers are char-device-based and fit into `file_operations`.
- Concurrency control is critical: use mutexes or spinlocks.
- I2C/SPI have **master vs slave** roles — always check bus protocol.
- Rust kernel traits (`i2c::I2cClient`, `spi::SpiDevice`) are analogous.

Chapter 15 — Concurrency & Synchronization

Kernel drivers must handle multiple processes and interrupts safely.

Spinlocks

```
spinlock_t lock;
spin_lock(&lock);
// critical section
spin_unlock(&lock);
```

Mutexes

```
struct mutex mtx;
mutex_lock(&mtx);
// safe to sleep
mutex_unlock(&mtx);
```

Read-Copy-Update (RCU)

```
struct my_node *ptr;
rcu_read_lock();
ptr = rcu_dereference(my_head);
rcu_read_unlock();
```

Learning Notes

- **Interrupt context:** spinlocks + GFP_ATOMIC
- **Process context:** mutexes allowed
- Avoid deadlocks by locking order and minimizing critical section.
- RCU allows read-mostly data structures without blocking readers.

Summary of Part 3

Chapter	Focus	Key Points
11	Kernel Types	u8/u16/u32/u64, endianness, error pointers
12	Linked Lists	list_head, safe iteration, synchronization
13	USB	Endpoints, URBs, interfaces, async transfers
14	I2C/SPI/UART	Serial bus drivers as char devices
15	Concurrency	Spinlocks, mutexes, RCU, interrupt safety

Learning Map

1. Use explicit types and handle endianness.
2. Master linked lists and memory-safe iteration.
3. Understand USB endpoint communication and URBs.
4. Serial bus drivers (I2C/SPI/UART) are specialized char drivers.
5. Apply proper locking primitives based on context (spinlock vs mutex).

Part 4 — Linux Device Drivers Ready Reckoner

Chapters 16–20 (DMA, Memory-Mapped Devices, Power Management, Hotplug, Advanced Topics)

Chapter 16 — Direct Memory Access (DMA)

DMA allows **devices to transfer data directly to/from memory** without CPU intervention.

DMA Types

- **Consistent / Coherent DMA:** CPU and device always see same data.
- **Streaming / Normal DMA:** CPU must flush caches.

DMA API Example

Scatter-Gather DMA

- Supports non-contiguous memory
- Often used in network/storage drivers

- Handled via `struct scatterlist` and `dma_map_sg()`

Learning Notes

- DMA-safe memory must be allocated with `dma_alloc_coherent` or `dma_map_single`.
- CPU must not directly access device memory mapped for DMA without proper barriers.
- Watch out for cache coherency on non-coherent architectures.

Chapter 17 — Memory-Mapped Devices (MMIO)

Many devices expose registers in memory space.

Mapping MMIO

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

void __iomem *regs;
regs = ioremap(phys_addr, size);
iowrite32(0x1234, regs);
u32 val = ioread32(regs);
iounmap(regs);
```

Learning Notes

- Always use `ioremap` and `iounmap`.
- Use `wmb()` / `rmb()` to enforce write/read ordering.
- MMIO is preferred over legacy port-mapped I/O on modern platforms.

Chapter 18 — Power Management (PM)

Drivers must respect system sleep, suspend, and resume.

PM Callbacks

```c
static int my_suspend(struct device *dev) { /* save state */ return 0; }
static int my_resume(struct device *dev) { /* restore state */ return 0; }

static const struct dev_pm_ops my_pm_ops = {
    .suspend = my_suspend,
    .resume  = my_resume,
};
```

Runtime PM

- Device can autosuspend when idle
- APIs: `pm_runtime_enable()`, `pm_runtime_get_sync()`, `pm_runtime_put_sync()`

Learning Notes

- Suspend/resume must save device state (registers, buffers).
- Always match runtime PM calls to avoid reference leaks.
- Critical for battery-powered embedded systems.

Chapter 19 — Hotplug & Device Registration

Linux supports dynamic **plug-and-play** for USB, PCI, I2C, SPI.

PCI Driver Example

```
```

Learning Notes

- Always match `.probe` and `.remove` lifecycle.
- Use `devm_*` managed resources to simplify cleanup.
- Hotplug devices require careful interrupt and memory management.

Chapter 20 — Advanced Topics & Kernel Integration

DMA + Interrupts

- Combine **bottom halves** and DMA for efficient transfers.
- Avoid sleeping in interrupt context; schedule workqueues for heavy processing.

Memory Barriers & Atomic Operations

```
```

Kernel Modules Best Practices

- Always free memory and IRQs in `exit` function.
- Avoid busy waiting; prefer wait queues or tasklets.
- Test on multiple architectures if using explicit sizes or endianness.

Learning Notes

- Modern kernel drivers use hybrid `interrupt-driven + DMA` for efficiency.
- Use `devm_*` APIs to reduce boilerplate and resource leaks.
- Keep critical sections minimal; prefer async processing for high-speed devices.

Summary of Part 4

Chapter	Focus	Key Points
16	DMA	dma_alloc_coherent, scatter-gather, CPU/device synchronization
17	MMIO	ioremap, ioread/iowrite, memory barriers
18	Power Management	suspend/resume, runtime PM, autosuspend
19	Hotplug	PCI, USB, device probe/remove, devm resources
20	Advanced Topics	DMA+interrupts, atomic operations, kernel best practices

Learning Map

1. Understand DMA flows and cache coherency.
2. Use MMIO safely with proper ordering.
3. Implement suspend/resume for device power management.
4. Register devices properly for hotplug support.
5. Combine interrupts, DMA, and deferred work efficiently.

Part 5 — LDD3 Mental Map Cheat Sheet

Linux Device Drivers — Kernel Developer Quick Reference Target: Linux 6.6 LTS | Hybrid C/Rust perspective

1. Kernel Data & Types

Concept	C Example	Notes / Rust Analogy
Fixed-size integers	`u8, u16, u32, u64`	Rust: `u8/u16/u32/u64`
Endianness	`cpu_to_le32(val)`	Rust: `to_le()` / `from_le()`
Error pointers	`ERR_PTR(-ENOMEM)`	Rust: `Result<T, Error>`

2. Memory Management

Concept	Example	Notes
`kmalloc`	`kmalloc(128, GFP_KERNEL)`	GFP_ATOMIC in IRQ, GFP_KERNEL in process context

Concept	Example	Notes
`vmalloc`	`vmalloc(4096)`	Virtually contiguous memory
Lookaside cache	`kmem_cache_alloc()`	High-volume object allocation
Per-CPU vars	`DEFINE_PER_CPU(int, counter)`	Each CPU has separate copy
DMA-safe	`dma_alloc_coherent()`	CPU/device coherent memory

3. I/O & Character Devices

Bus/Device	Kernel API	Notes
UART	`tty_driver`, `uart_port`	Blocking/non-blocking read/write
I2C	`i2c_smbus_read/write`	Master/slave roles, char-device style
SPI	`spi_sync_transfer()`	Blocking or async transfers
Generic char	`file_operations`	`.read`, `.write`, `.ioctl`

4. Blocking & Async I/O

Concept	C Example	Notes
Wait queues	`wait_event_interruptible(queue, cond)`	Sleep until condition
Poll/select	`.poll()` with `poll_wait()`	Non-blocking user-space support
Async notification	`fasync_struct`, `kill_fasync()`	Signals user-space process

5. Timers & Deferred Work

Concept	Example	Notes
Kernel timers	`timer_setup()` + `mod_timer()`	Schedule function later
Tasklets	`DECLARE_TASKLET(my_tasklet, func, data)`	Softirq context, cannot sleep
Workqueues	`DECLARE_WORK(work, func)`	Process context, can sleep

6. Interrupts

Concept	Example	Notes
Top-half	`request_irq()` handler	Quick response, no sleep
Bottom-half	Tasklets/Workqueues	Deferred heavy processing

Concept	Example	Notes
Shared IRQ	`IRQF_SHARED` + unique `dev_id`	Check if interrupt is for your device

7. USB Drivers

Concept	Example	Notes
Endpoints	CONTROL/BULK/INT/ISO	Direction: IN/OUT
URB	`usb_alloc_urb()` + `usb_submit_urb()`	Async transfer with callback
Interfaces	`struct usb_interface`	Group endpoints by function

8. PCI & Hotplug

Concept	Example	Notes
PCI driver	`pci_driver`, `.probe`, `.remove`	Devm-managed resources simplify cleanup
Hotplug	USB/PCI dynamic detection	Handle attach/detach cleanly

9. Power Management

Concept	Example	Notes
Suspend/Resume	`.suspend = my_suspend`	Save device state
Runtime PM	`pm_runtime_get_sync()` / `pm_runtime_put_sync()`	Autosuspend idle devices

10. Concurrency & Synchronization

Concept	Example	Notes
Spinlocks	`spin_lock()/spin_unlock()`	IRQ-safe, cannot sleep
Mutex	`mutex_lock()/unlock()`	Sleep allowed, process context
RCU	`rcu_read_lock()/rcu_read_unlock()`	Read-mostly, wait-free reads
Atomic	`atomic_inc()/atomic_dec()`	For counters in IRQ or SMP

11. Advanced DMA + MMIO

Concept	Example	Notes
DMA coherent	`dma_alloc_coherent()`	CPU & device always see same data
MMIO	`ioremap()/iowrite32()/ioread32()`	Use memory barriers
Scatter-gather	`dma_map_sg()`	Non-contiguous memory support

12. Best Practices

- Always `release resources` in `exit` or `.remove`.
- Use `devm_` APIs to simplify cleanup.
- Prefer `async + bottom-half processing` over busy-waiting.
- Keep critical sections minimal.
- Test with `non-blocking and interrupt contexts`.

Quick Mental Map Flow

```
[User-space] <-> [Char device / Bus driver]
                   |
                   v
           [Blocking / Non-blocking I/O]
                   |
        ---------------------------
        |                         |
    [Interrupts]             [Timers / Tasklets / Workqueues]
        |                         |
   [DMA + MMIO]                [Deferred Work]
        |                         |
   [PCI/USB/I2C/SPI/UART]   [Concurrency & Sync]
        |
  [Power Management + Hotplug]
```

Rust Kernel Analogies

- `kmalloc` → `alloc::<T>()` / `Box<T>`
- `ioremap` → `IoMem::map()`
- `tasklet` → `KernelTasklet::schedule()`
- `workqueue` → `WorkQueue::queue()`
- `spinlock` → `SpinLock<T>`
- `per-CPU variable` → `PerCpu<T>`