LINUX DEVICE DRIVERS

Complete Comprehensive Guide

From Kernel Basics to Advanced Driver Development
Including Interrupts, Synchronization, Timers, GPIO, and Real-World Examples

Generated: March 12, 2026

TABLE OF CONTENTS

1. INTRODUCTION TO LINUX

What is Linux?

Linux is a free and open-source operating system kernel based on Unix philosophy. It manages hardware resources and provides essential services to all applications running on the system. The Linux kernel can be customized and recompiled for specific hardware platforms and use cases.

Unix Philosophy

• Do one thing and do it well - Each program focuses on a single task

• Work together - Programs should be composable and interoperable

• Data should be plain text - Easy to read, debug, and process

• Small and modular tools - Complex tasks through composition

• Avoid clutter - Keep designs simple without unnecessary features

• Fail early and clearly - Quick failure with informative error messages

Key Characteristics of Linux

✓ Open Source - Full source code available for study and modification

✓ Portable - Runs on x86, ARM, RISC-V, PowerPC, and many other architectures

✓ Multitasking - Multiple programs execute concurrently

✓ Multi-user - Multiple users access system simultaneously

✓ Secure - User/group permissions and access control mechanisms

✓ Scalable - From embedded systems to supercomputers

2. LINUX ARCHITECTURE

User Space vs Kernel Space

Linux memory is divided into two regions with different privilege levels. User applications cannot directly access kernel memory or hardware. All hardware access must go through system calls.

Aspect	Kernel Space	User Space
Privilege	Ring 0 (Full privileges)	Ring 3 (Limited)
Hardware Access	Direct access	Via system calls
Memory	Unrestricted	Protected/Isolated
Speed	Faster	Slower (context switch)
Examples	Device drivers	Applications

System Calls Bridge

System calls provide the controlled interface between user space and kernel space. When an application needs to perform privileged operations (I/O, memory allocation, etc.), it makes a system call. The kernel then executes the requested operation and returns control to user space.

3. LINUX KERNEL MODULES

What is a Kernel Module?

A kernel module (LKM) is compiled code that can be dynamically inserted into a running kernel. Modules allow extending kernel functionality without recompiling the entire kernel or rebooting the system.

Module Applications

→ Device Driver Support - Add drivers for new hardware

→ Filesystem Support - Add support for new filesystems (ext4, BTRFS, etc.)

→ System Call Extensions - Implement custom system calls

→ Security Modules - Add security features like SELinux

→ Network Protocols - Add new network protocol implementations

→ Performance Monitoring - Add profiling and monitoring tools

Basic Module Structure

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Module description");
MODULE_VERSION("1.0");

static int __init module_init_func(void) {
    printk(KERN_INFO "Module loaded\n");
    return 0;
}

static void __exit module_exit_func(void) {
    printk(KERN_INFO "Module removed\n");
}

module_init(module_init_func);
module_exit(module_exit_func);
```

Module Metadata

MODULE_LICENSE: Specifies license (GPL, Proprietary, etc.). GPL is preferred for kernel modules.

MODULE_AUTHOR: Author information for identification and support contact.

MODULE_DESCRIPTION: Brief description of module functionality.

MODULE_VERSION: Version number for tracking updates and compatibility.

4. MODULE LOADING AND UNLOADING

Loading Modules (insmod)

The insmod command loads a compiled kernel module (.ko file) into the running kernel. The __init function is executed during loading.

```
$ sudo insmod my_module.ko
$ dmesg | tail              # View kernel messages
$ lsmod                      # List loaded modules
$ cat /proc/modules          # View module details
```

Unloading Modules (rmmod)

The rmmod command removes a loaded module from the kernel. The __exit function is executed during unloading.

```
$ sudo rmmod my_module       # Remove by module name
$ modinfo my_module.ko       # View module information
$ modprobe my_module         # Load with dependency resolution
```

Key Directives

__init: Marks initialization code that is freed after loading

__exit: Marks exit code only included if module supports unloading

THIS_MODULE: Reference to current module in kernel

module_init(): Register initialization function with kernel

module_exit(): Register cleanup function with kernel

5. CHARACTER DEVICE DRIVERS FUNDAMENTALS

What is a Character Device?

Character devices handle byte-stream I/O. Data flows sequentially one byte at a time. Examples include serial ports, terminals, keyboards, mice, and sound cards.

Characteristics of Character Devices

• Sequential data access (cannot seek)

• Unbuffered I/O operations

• Real-time responsiveness required

• Cannot be mounted as filesystems

• Often interactive with user input

vs Block Devices

Aspect	Character Device	Block Device
Access	Sequential (byte-by-byte)	Block-based (512B+)
Buffering	Unbuffered	Buffered
Seeking	Not supported	Supported
Example	Serial port, keyboard	Hard disk, USB
Mount	Cannot mount	Can mount as filesystem

6. DEVICE FILES AND /dev DIRECTORY

Understanding Device Files

Device files in /dev provide the interface between user-space applications and device drivers. They are special files, not regular files on disk. Each device file has major and minor numbers.

Examining Device Files

```
$ ls -l /dev/ttyS0
crw-rw---- 1 root dialout 4, 64 Dec 12 /dev/ttyS0
■       ■                    ■  ■
■■ Type ■■ Permissions        ■  ■■ Minor number
           (owner/group)     ■■ Major number

$ ls -l /dev/sda
brw-rw---- 1 root disk 8, 0 Dec 12 /dev/sda
■                             ■  ■
■■ Block device              Major Minor
```

Device File Types

c (Character): Character device (serial port, keyboard)

b (Block): Block device (hard disk, USB drive)

l (Link): Symbolic link

s (Socket): Network socket

p (Pipe): Named pipe (FIFO)

Common /dev Files

→ /dev/null - Discard all data written to it

→ /dev/zero - Provides infinite zeros

→ /dev/random - Random number generator

→ /dev/ttyX - Terminal/console devices

→ /dev/hdX, /dev/sdX - Disk drives

→ /dev/ptyx - Pseudo-terminal devices

→ /dev/mem - Physical memory access

7. MAJOR AND MINOR NUMBERS

Understanding Major Numbers

The major number identifies which device driver will handle the device file. Each driver is assigned a unique major number. When the kernel receives a request for a device file, it uses the major number to find the appropriate driver.

Understanding Minor Numbers

The minor number identifies a specific device instance managed by the driver. For example, a UART driver with major number 4 might use minor numbers 0, 1, 2, 3 for /dev/ttyS0, /dev/ttyS1, /dev/ttyS2, /dev/ttyS3 respectively.

Viewing Assigned Numbers

```
$ cat /proc/devices
Character devices:
  1 mem
  4 tty
 240 my_device

Block devices:
  1 ramdisk
  8 sd

$ ls -l /dev | grep my_device
crw-rw-rw- 1 root root 240, 0 my_device
```

MKDEV and Macros

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

// Create dev_t from major and minor
dev_t dev = MKDEV(240, 0);

// Extract major number
int major = MAJOR(dev);  // Returns 240

// Extract minor number
int minor = MINOR(dev);  // Returns 0
```

8. DYNAMIC AND STATIC DEVICE NUMBER ALLOCATION

Static Allocation

Statically allocate a specific major number for your driver. Advantages: Predictable numbers for scripting. Disadvantages: Risk of conflicts with other drivers.

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

#define MY_MAJOR 240

dev_t dev = MKDEV(MY_MAJOR, 0);

int result = register_chrdev_region(dev, 1, "my_device");
if (result < 0) {
    printk(KERN_ERR "Cannot register major %d\n", MY_MAJOR);
    return -1;
}

printk(KERN_INFO "Major %d allocated\n", MY_MAJOR);
```

Dynamic Allocation

Let the kernel assign an available major number. Advantages: Avoids conflicts, recommended approach. Disadvantages: Number not known until runtime.

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

dev_t dev;

int result = alloc_chrdev_region(&dev, 0, 1, "my_device");
if (result < 0) {
    printk(KERN_ERR "Cannot allocate major number\n");
    return -1;
}

int major = MAJOR(dev);
int minor = MINOR(dev);
printk(KERN_INFO "Allocated Major: %d, Minor: %d\n",
       major, minor);
```

Freeing Device Numbers

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

// For both static and dynamic allocation:
void unregister_chrdev_region(dev_t dev, unsigned int count);

// Example in module exit:
static void __exit my_module_exit(void) {
    // Remove character device
    cdev_del(&my_cdev);

    // Free allocated numbers
    unregister_chrdev_region(dev, 1);

    printk(KERN_INFO "Device numbers freed\n");
}
```

Important: Always free device numbers in module exit. Failing to do so wastes the number space and can cause system issues.

9. CREATING DEVICE NODES (MANUAL AND AUTOMATIC)

Manual Device Node Creation

Use the mknod command to manually create device files. This method is useful for testing before automating with udev.

```
# Create character device node
sudo mknod /dev/my_device c 240 0

# Set permissions
sudo chmod 666 /dev/my_device

# Verify
ls -l /dev/my_device
crw-rw-rw- 1 root root 240, 0 my_device

# Remove device node
sudo rm /dev/my_device
```

Automatic Device Node Creation with udev

Modern Linux systems use udev to automatically create device nodes when drivers load. This eliminates manual node creation.

Using class_create() and device_create()

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

dev_t dev;
static struct class *dev_class;

// In module init:
alloc_chrdev_region(&dev, 0, 1, "my_device");

// Create device class
dev_class = class_create(THIS_MODULE, "my_class");
if (IS_ERR(dev_class)) {
    printk(KERN_ERR "Failed to create class\n");
    return PTR_ERR(dev_class);
}

// Create device automatically
struct device *device = device_create(
    dev_class,      // Device class
    NULL,           // Parent device
    dev,            // Device number
    NULL,           // Driver data
    "my_device"     // Device name
);

if (IS_ERR(device)) {
    class_destroy(dev_class);
    return PTR_ERR(device);
}

// Device appears at /dev/my_device automatically!
```

Cleanup in module exit

```
// In module exit:
device_destroy(dev_class, dev);
class_destroy(dev_class);
cdev_del(&my_cdev);
unregister_chrdev_region(dev, 1);
```

10. CHARACTER DEVICE STRUCTURE (cdev)

Understanding struct cdev

The cdev structure represents a character device in the kernel. It contains pointers to device operations, ownership information, and device numbers.

```
struct cdev {
    struct kobject kobj;           // Kernel object
    struct module *owner;          // Owning module
    const struct file_operations *ops;  // File operations
    struct list_head list;         // Linked list
    dev_t dev;                     // Device number
    unsigned int count;            // Number of devices
};
```

Allocating cdev

Static Allocation:

```
static struct cdev my_cdev;

// In init:
cdev_init(&my_cdev, &my_fops);
my_cdev.owner = THIS_MODULE;

if (cdev_add(&my_cdev, dev, 1) < 0) {
    printk(KERN_ERR "Failed to add cdev\n");
    return -1;
}
```

Dynamic Allocation:

```
struct cdev *my_cdev = cdev_alloc();
if (!my_cdev) {
    printk(KERN_ERR "Cannot allocate cdev\n");
    return -ENOMEM;
}

my_cdev->ops = &my_fops;
my_cdev->owner = THIS_MODULE;

if (cdev_add(my_cdev, dev, 1) < 0) {
    printk(KERN_ERR "Failed to add cdev\n");
    return -1;
}
```

Registering with Kernel (cdev_add)

The cdev_add() function registers the character device with the kernel, linking it to the major number and making it available through /dev.

Removing Device (cdev_del)

The cdev_del() function unregisters the device from the kernel. After this, the device file becomes inaccessible.

11. FILE OPERATIONS STRUCTURE

The file_operations Structure

The file_operations structure defines how a device responds to system calls. It contains function pointers for operations like open, read, write, and release.

```
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 *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    // ... more operations
};
```

Common Operations

Operation	Description
open	Called when user opens device file
release/close	Called when user closes device file
read	Called for read() system call
write	Called for write() system call
seek/llseek	Called for lseek() system call
ioctl/unlocked_ioctl	Called for ioctl() system call
poll/select	Called for select/poll system calls
mmap	Called for memory mapping

Example file_operations

```
static struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .release = my_release,
    .read = my_read,
    .write = my_write,
};

// Implement operations
static int my_open(struct inode *inode, struct file *filp) {
    printk(KERN_INFO "Device opened\n");
    return 0;
}

static int my_release(struct inode *inode, struct file *filp) {
    printk(KERN_INFO "Device closed\n");
    return 0;
}

static ssize_t my_read(struct file *filp, char __user *buf,
                       size_t count, loff_t *f_pos) {
    // Read from device to user buffer
    copy_to_user(buf, kernel_buffer, count);
    return count;
}

static ssize_t my_write(struct file *filp, const char __user *buf,
                        size_t count, loff_t *f_pos) {
    // Write from user buffer to device
    copy_from_user(kernel_buffer, buf, count);
    return count;
}
```

12. IOCTL (INPUT/OUTPUT CONTROL) OPERATIONS

Understanding IOCTL

IOCTL (Input/Output Control) allows user-space programs to send device-specific commands to drivers. While read/write work for data transfer, IOCTL handles control operations like setting parameters, querying status, or configuring hardware.

When to Use IOCTL

• Changing device parameters (baud rate, voltage, etc.)

• Querying device status or capabilities

• Performing device-specific operations

• Enabling/disabling features

• Ejecting removable media

• Screen resolution changes

Defining IOCTL Commands

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

// Command macros
#define IOCTL_SET_VALUE _IOW('a', 1, int)   // Write to driver
#define IOCTL_GET_VALUE _IOR('a', 2, int)   // Read from driver
#define IOCTL_DO_ACTION _IO('a', 3)         // No data transfer

// Components:
// 'a' = magic number (unique identifier)
// 1, 2, 3 = command numbers
// int = data type for _IOW/_IOR
```

IOCTL Command Types

Macro	Direction	Use Case
_IO	None	Command without data
_IOW	Write to kernel	User sends data to driver
_IOR	Read from kernel	Driver sends data to user
_IOWR	Both	Two-way data transfer

Implementing IOCTL Handler

```
static long my_ioctl(struct file *filp, unsigned int cmd,
                    unsigned long arg) {
    switch (cmd) {
    case IOCTL_SET_VALUE:
        // Copy data from user space
        if (copy_from_user(&device_value, (int __user *)arg,
                          sizeof(int))) {
            return -EFAULT;
        }
        printk(KERN_INFO "Value set to: %d\n", device_value);
        break;
    case IOCTL_GET_VALUE:
        // Copy data to user space
        if (copy_to_user((int __user *)arg, &device_value,
                        sizeof(int))) {
            return -EFAULT;
        }
        break;

    default:
        return -EINVAL;  // Invalid command
    }
    return 0;
}

// Add to file_operations
static struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = my_ioctl,
};
```

Using IOCTL from User Space

```
#include <sys/ioctl.h>
#include <fcntl.h>
#include <stdio.h>

int main() {
    int fd = open("/dev/my_device", O_RDWR);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    // Set value using IOCTL
    int value = 42;
    if (ioctl(fd, IOCTL_SET_VALUE, &value) < 0) {
        perror("ioctl set");
        return 1;
    }

    // Get value using IOCTL
    int result;
    if (ioctl(fd, IOCTL_GET_VALUE, &result) < 0) {
        perror("ioctl get");
        return 1;
    }

    printf("Value from device: %d\n", result);
    close(fd);
    return 0;
}
```

13. PROCFS (PROCESS FILE SYSTEM) INTERFACE

Understanding Procfs

Procfs is a virtual filesystem (/proc) that provides a window into kernel data structures. It allows user-space applications to query and modify kernel parameters at runtime without recompilation.

Common /proc Files

• /proc/cpuinfo - CPU information

• /proc/meminfo - Memory usage statistics

• /proc/version - Kernel version

• /proc/modules - Loaded kernel modules

• /proc/devices - Registered devices

• /proc/interrupts - IRQ statistics

• /proc/filesystems - Supported filesystems

Creating Proc Entries

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

// Create a proc directory
struct proc_dir_entry *parent = proc_mkdir("my_driver", NULL);

// Create a proc file with read/write operations
struct proc_dir_entry *entry = proc_create(
    "status",           // File name
    0644,              // Permissions
    parent,            // Parent directory
    &proc_ops          // File operations
);

// Define file operations
static ssize_t proc_read(struct file *filp, char __user *buf,
                         size_t count, loff_t *f_pos) {
    char buffer[256];
    int len = sprintf(buffer, "Driver Status: OK\n");
    copy_to_user(buf, buffer, len);
    return len;
}

static ssize_t proc_write(struct file *filp, const char __user *buf,
                          size_t count, loff_t *f_pos) {
    printk(KERN_INFO "Received %zu bytes from user\n", count);
    return count;
}

static struct file_operations proc_ops = {
    .read = proc_read,
    .write = proc_write,
};
```

Removing Proc Entries

```
// Remove proc entry
remove_proc_entry("status", parent);

// Remove directory
remove_proc_entry("my_driver", NULL);
```

14. SYSFS (SYSTEM FILE SYSTEM) INTERFACE

Understanding Sysfs

Sysfs is a virtual filesystem (/sys) that represents kernel objects and their attributes. It provides a more structured interface than procfs for exposing device and driver information.

Sysfs Structure

/sys/class/ - Classes of devices (net, block, input, etc.)
/sys/devices/ - Physical device hierarchy
/sys/module/ - Loaded kernel modules
/sys/fs/ - Filesystem information
/sys/kernel/ - Kernel parameters and subsystems

Creating Kobjects and Attributes

```
#include <linux/sysfs.h>
#include <linux/kobject.h>

static int device_value = 0;
static struct kobject *my_kobject;

// Show function for attribute
static ssize_t show_value(struct kobject *kobj,
                          struct kobj_attribute *attr,
                          char *buf) {
    return sprintf(buf, "%d\n", device_value);
}

// Store function for attribute
static ssize_t store_value(struct kobject *kobj,
                           struct kobj_attribute *attr,
                           const char *buf,
                           size_t count) {
    sscanf(buf, "%d", &device_value);
    return count;
}

// Define attribute
static struct kobj_attribute value_attr =
    __ATTR(value, 0644, show_value, store_value);

// Create kobject
my_kobject = kobject_create_and_add("my_device",
                                    kernel_kobj);

// Add attribute to kobject
sysfs_create_file(my_kobject, &value_attr.attr);
```

Using Sysfs from User Space

```
# Read sysfs attribute
cat /sys/kernel/my_device/value

# Write sysfs attribute
echo "100" > /sys/kernel/my_device/value
```

```
# Verify change
cat /sys/kernel/my_device/value
```

15. INTERRUPT HANDLING FUNDAMENTALS

What are Interrupts?

An interrupt is a signal to the CPU that something requires immediate attention. Instead of continuously polling devices (wasteful), the CPU can handle other tasks and respond when interrupted.

Polling vs Interrupts

Aspect	Polling	Interrupts
CPU Usage	Continuous (wasteful)	Only when needed
Latency	High (delay until checked)	Low (immediate response)
Power	High	Low
Complexity	Simple	More complex
Best For	Slow devices	Hardware events

Interrupt vs Exception

Interrupts: Asynchronous signals from hardware devices. Can occur at any time during program execution.

Exceptions: Synchronous errors generated by the CPU itself (page faults, division by zero, system calls).

16. INTERRUPT SERVICE ROUTINES (ISRs)

What is an ISR?

An Interrupt Service Routine (ISR) is a function that handles a specific interrupt. When an interrupt occurs, the kernel suspends the current task and executes the appropriate ISR.

ISR Constraints

■ Must be as fast as possible (holds up other interrupts)

■ Cannot call blocking functions (no sleep, no mutex)

■ Cannot access user-space memory directly

■ Must not perform time-consuming operations

■ Cannot perform memory allocation with sleep-capable flags

Basic ISR Structure

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

// Interrupt handler
static irqreturn_t my_irq_handler(int irq, void *dev_id) {
    // Acknowledge interrupt immediately
    disable_device_irq();

    // Quick, time-critical work only
    unsigned int status = read_device_status();

    // Schedule bottom half for deferred work
    tasklet_schedule(&my_tasklet);

    // Acknowledge to hardware
    clear_device_interrupt();
    enable_device_irq();

    return IRQ_HANDLED;  // Indicate successful handling
}
```

Return Values

IRQ_HANDLED: Handler processed the interrupt successfully

IRQ_NONE: Handler didn't process the interrupt (for shared IRQs)

17. TOP HALF AND BOTTOM HALF PROCESSING

The Problem with Long ISRs

If an interrupt handler performs too much work, it blocks other interrupts and impacts system responsiveness. The solution is to split the work into two halves: top half (urgent) and bottom half (deferred).

Top Half (ISR)

The top half runs immediately when an interrupt occurs. It handles time-critical work like acknowledging the interrupt and saving data.

→ Acknowledge hardware interrupt

→ Save device status/data into memory buffer

→ Schedule bottom half for later processing

→ Minimal processing

Bottom Half (Deferred Work)

The bottom half runs later in a less time-critical context. It performs the actual data processing and can block/sleep.

→ Process captured data

→ Perform I/O operations

→ Wake up waiting processes

→ Access blocking functions

Bottom Half Mechanisms

Softirq: Kernel software interrupts for high-priority deferred work

Tasklets: Simple, non-reentrant deferred work mechanism

Work Queues: Deferred work running in kernel threads (can sleep)

Threaded IRQs: ISRs running as kernel threads (preemptible)

Top Half/Bottom Half Example

```
// Top half (ISR)
static irqreturn_t serial_irq_handler(int irq, void *dev_id) {
    unsigned char data = read_uart_data();
    store_in_buffer(data);

    // Schedule bottom half
    tasklet_schedule(&rx_tasklet);

    return IRQ_HANDLED;
}

// Bottom half (Tasklet)
static void serial_rx_tasklet(unsigned long data) {
    struct device *dev = (struct device *)data;

    // Process buffered data
    while (has_buffered_data()) {
        unsigned char byte = get_buffered_byte();
        process_serial_byte(byte);
    }

    // Wake up waiting processes
    wake_up(&serial_wait_queue);
}
```

18. REQUEST AND FREE IRQ OPERATIONS

Registering Interrupts with request_irq()

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

int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev_id);

// Parameters:
// irq - IRQ number to register
// handler - Pointer to ISR function
// flags - Control behavior (IRQF_SHARED, IRQF_TRIGGER_RISING, etc.)
// name - Device name (visible in /proc/interrupts)
// dev_id - Unique identifier for shared IRQs

// Returns:
// 0 on success
// Negative error code on failure
```

Common Flags

Flag	Meaning
IRQF_SHARED	Multiple handlers share same IRQ
IRQF_TRIGGER_RISING	Trigger on rising edge
IRQF_TRIGGER_FALLING	Trigger on falling edge
IRQF_TRIGGER_HIGH	Trigger while line is high
IRQF_TRIGGER_LOW	Trigger while line is low

Example Registration

```
// In module init
static irqreturn_t my_handler(int irq, void *dev_id) {
    printk(KERN_INFO "Interrupt %d received\n", irq);
    return IRQ_HANDLED;
}

static int __init my_module_init(void) {
    int result = request_irq(
        11,                          // IRQ number
        my_handler,                  // Handler function
        IRQF_SHARED | IRQF_TRIGGER_RISING,  // Flags
        "my_device",                 // Name
        (void *)&my_device_data      // Unique ID
    );

    if (result < 0) {
        printk(KERN_ERR "Cannot request IRQ 11\n");
        return -1;
    }
```

```
        printk(KERN_INFO "IRQ 11 registered successfully\n");
        return 0;
    }
```

Freeing Interrupts with free_irq()

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

    // Parameters:
    // irq - IRQ number to free
    // dev_id - Must match the dev_id from request_irq()

    // Example in module exit:
    static void __exit my_module_exit(void) {
        free_irq(11, &my_device_data);  // Must match request_irq
        printk(KERN_INFO "IRQ 11 freed\n");
    }

    // Important: Always call free_irq in module exit!
    // Failing to free causes resource leaks
```

Shared IRQs

When IRQF_SHARED is used, multiple drivers can share the same IRQ. The kernel calls all registered handlers. Each must check if the interrupt is for them and return IRQ_HANDLED or IRQ_NONE.

19. TASKLETS IN LINUX KERNEL

What are Tasklets?

Tasklets are a bottom-half mechanism for deferring work from interrupt handlers. They are simple, run in atomic context, and cannot sleep. A tasklet is guaranteed to run on the same CPU that scheduled it.

Tasklet Characteristics

• Run in atomic context (cannot sleep)

• Non-preemptible (cannot be interrupted)

• Same CPU execution (cache friendly)

• Cannot use mutexes or semaphores

• Can use spinlocks

Static Tasklet Declaration

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

// Tasklet function
void my_tasklet_function(unsigned long data) {
    struct device *dev = (struct device *)data;
    printk(KERN_INFO "Tasklet executing\n");
    // Perform bottom-half work
}

// Static declaration and initialization
static DECLARE_TASKLET(my_tasklet, my_tasklet_function,
                       (unsigned long)&my_device);
```

Dynamic Tasklet Creation

```
struct tasklet_struct *my_tasklet;

// Allocate and initialize
my_tasklet = kmalloc(sizeof(struct tasklet_struct), GFP_KERNEL);
if (!my_tasklet)
    return -ENOMEM;

tasklet_init(my_tasklet, my_tasklet_function,
             (unsigned long)&my_device);
```

Scheduling and Managing Tasklets

```
// Schedule tasklet (normal priority)
tasklet_schedule(&my_tasklet);

// Schedule with high priority
tasklet_hi_schedule(&my_tasklet);

// Disable tasklet
tasklet_disable(&my_tasklet);

// Enable tasklet
tasklet_enable(&my_tasklet);
```

```
// Kill tasklet and wait for completion
tasklet_kill(&my_tasklet);

// Check if tasklet is scheduled
if (tasklet_pending(&my_tasklet))
    printk(KERN_INFO "Tasklet is pending\n");
```

20. WORK QUEUES AND DEFERRED WORK

Work Queues vs Tasklets

Aspect	Tasklets	Work Queues
Context	Atomic (no sleep)	Process context (can sleep)
CPU	Same CPU always	Any CPU
Locks	Spinlock only	Mutex, semaphore, spinlock
Blocking	Cannot block	Can block on locks
Typical Use	Quick processing	Longer operations, I/O

Static Work Queue

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

// Work function
void my_work_function(struct work_struct *work) {
    printk(KERN_INFO "Work executing\n");
    // Can sleep, do blocking I/O, etc.
}

// Static declaration
static DECLARE_WORK(my_work, my_work_function);

// In interrupt handler or anywhere:
schedule_work(&my_work);  // Queue for execution
```

Dynamic Work Allocation

```
struct work_struct *my_work;

// Allocate
my_work = kmalloc(sizeof(struct work_struct), GFP_ATOMIC);
if (!my_work)
    return -ENOMEM;

// Initialize
INIT_WORK(my_work, my_work_function);

// Schedule
schedule_work(my_work);

// Wait for completion
flush_work(my_work);

// Cleanup
cancel_work_sync(my_work);
kfree(my_work);
```

Delayed Work Queues

```
// Delayed work structure
struct delayed_work my_delayed_work;
```

```
// Initialize
INIT_DELAYED_WORK(&my_delayed_work, my_work_function);

// Schedule with delay
schedule_delayed_work(&my_delayed_work, HZ);  // 1 second

// Cancel
cancel_delayed_work_sync(&my_delayed_work);
```

21. WAIT QUEUES AND PROCESS SYNCHRONIZATION

What are Wait Queues?

Wait queues allow processes to sleep until a specific condition becomes true. Instead of busy-waiting (wasting CPU), processes sleep and are woken when the event occurs.

Declaring Wait Queues

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

// Static declaration
static DECLARE_WAIT_QUEUE_HEAD(my_wait_queue);

// Dynamic declaration and init
wait_queue_head_t my_wait_queue;
init_waitqueue_head(&my_wait_queue);
```

Putting Tasks to Sleep

wait_event(): Sleep until condition is true (uninterruptible)

wait_event_interruptible(): Sleep until condition or signal

wait_event_timeout(): Sleep with timeout

wait_event_interruptible_timeout(): Sleep with timeout and signals

Example: Using Wait Queues

```
static DECLARE_WAIT_QUEUE_HEAD(data_wait_queue);
static int data_available = 0;

// In ISR/tasklet: signal that data is ready
void signal_data_ready(void) {
    data_available = 1;
    wake_up(&data_wait_queue);  // Wake one waiter
}

// In user process: wait for data
ssize_t device_read(struct file *filp, char __user *buf,
                    size_t count, loff_t *f_pos) {
    // Sleep until data is available
    wait_event_interruptible(data_wait_queue,
                            data_available == 1);

    // Data is now available
    copy_to_user(buf, data_buffer, count);
    data_available = 0;
    return count;
}

// Wake up all waiters
wake_up_all(&data_wait_queue);
```

22. MUTEX (MUTUAL EXCLUSION LOCKS)

Understanding Mutex

A mutex (mutual exclusion) is a synchronization primitive that ensures only one thread can access a shared resource at a time. When locked, other threads sleep until the mutex is released.

Mutex Characteristics

• Only one owner at a time

• Sleeping threads (not busy-wait)

• Cannot be used in interrupt context

• Can be held for longer durations

• FIFO wake-up order

Declaring and Initializing Mutex

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

// Static declaration
static DEFINE_MUTEX(my_mutex);

// Dynamic declaration and init
struct mutex my_mutex;
mutex_init(&my_mutex);
```

Locking Operations

Function	Behavior
mutex_lock()	Lock (sleep if held)
mutex_trylock()	Try lock (return immediately)
mutex_lock_interruptible()	Lock (can be interrupted)
mutex_unlock()	Release mutex

Mutex Example

```
static DEFINE_MUTEX(device_mutex);
static int shared_value = 0;

// Reading protected data
void read_value(int *result) {
    mutex_lock(&device_mutex);
    *result = shared_value;
    mutex_unlock(&device_mutex);
}

// Writing protected data
void write_value(int new_value) {
    mutex_lock(&device_mutex);
    shared_value = new_value;
    printk(KERN_INFO "Value changed to: %d\n",
           new_value);
    mutex_unlock(&device_mutex);
}

// In file operations
static ssize_t device_write(struct file *filp,
                           const char __user *buf,
                           size_t count,
                           loff_t *f_pos) {
    int value;
    sscanf(buf, "%d", &value);
    write_value(value);
    return count;
}
```

23. SPINLOCKS AND BUSY-WAIT MECHANISMS

Understanding Spinlock

A spinlock is a busy-wait synchronization primitive. When a thread cannot acquire a spinlock, it continuously checks the lock instead of sleeping. Spinlocks are used for short critical sections where sleep overhead is too high.

Spinlock Characteristics

• Busy-waiting (spins on CPU)

• Very short hold time

• Can be used in interrupt context

• Disables preemption

• Low latency for short waits

Declaring Spinlock

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

// Static declaration
static DEFINE_SPINLOCK(my_spinlock);

// Dynamic declaration and init
spinlock_t my_spinlock;
spin_lock_init(&my_spinlock);
```

Spinlock Operations

spin_lock(): Acquire spinlock

spin_unlock(): Release spinlock

spin_trylock(): Try to acquire (non-blocking)

spin_lock_irq(): Lock and disable interrupts

spin_lock_irqsave(): Lock and save interrupt state

Spinlock Context Variants

```
// User context to user context
spin_lock(&my_spinlock);
// Critical section
spin_unlock(&my_spinlock);

// Bottom half to bottom half
spin_lock_bh(&my_spinlock);
// Critical section (soft IRQ disabled)
spin_unlock_bh(&my_spinlock);

// Top half to bottom half (IRQ handler)
spin_lock_irq(&my_spinlock);
// Critical section (interrupts disabled)
spin_unlock_irq(&my_spinlock);

// Safe variant (save/restore state)
unsigned long flags;
```

```
spin_lock_irqsave(&my_spinlock, flags);
// Critical section
spin_unlock_irqrestore(&my_spinlock, flags);
```

24. LINKED LISTS IN LINUX KERNEL

Kernel Linked List Implementation

The Linux kernel provides an efficient doubly-linked list implementation. Unlike user-space lists, the kernel list is intrusive - the list node is embedded in the containing structure.

List Structure

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

// List node structure
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};

// Embed in your structure
struct my_device {
    int id;
    char name[32];
    struct list_head list;  // List node
};
```

List Operations

Operation	Purpose
LIST_HEAD(name)	Declare and initialize list head
list_add(new, head)	Add to beginning of list
list_add_tail(new, head)	Add to end of list
list_del(entry)	Remove from list
list_for_each(pos, head)	Iterate through list
list_for_each_entry(pos, head, member)	Iterate with type safety
list_empty(head)	Check if list is empty
list_splice(new, head)	Join two lists

List Example

```
static LIST_HEAD(device_list);
static DEFINE_SPINLOCK(list_lock);

// Add device to list
void add_device(struct my_device *dev) {
    spin_lock(&list_lock);
    list_add_tail(&dev->list, &device_list);
    spin_unlock(&list_lock);
}

// Remove device from list
void remove_device(struct my_device *dev) {
    spin_lock(&list_lock);
    list_del(&dev->list);
    spin_unlock(&list_lock);
}

// Find device by ID
struct my_device *find_device(int id) {
    struct my_device *dev;
    spin_lock(&list_lock);

    list_for_each_entry(dev, &device_list, list) {
        if (dev->id == id) {
            spin_unlock(&list_lock);
            return dev;
        }
    }

    spin_unlock(&list_lock);
    return NULL;  // Not found
}

// Iterate and print all devices
void print_all_devices(void) {
    struct my_device *dev;
    spin_lock(&list_lock);

    list_for_each_entry(dev, &device_list, list) {
        printk(KERN_INFO "Device: %s (ID: %d)\n",
               dev->name, dev->id);
    }

    spin_unlock(&list_lock);
}
```

25. KERNEL THREADS CREATION AND MANAGEMENT

What are Kernel Threads?

Kernel threads are lightweight processes running in kernel space. They have no user-space memory context and are used for background kernel tasks.

Kernel Thread Characteristics

• Run entirely in kernel space (no user context)

• Preemptible (can be interrupted)

• Can use all kernel services

• Can sleep and block

• Listed in ps output

Creating Kernel Threads

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

// Thread function
static int thread_function(void *data) {
    struct my_data *data = (struct my_data *)data;

    // Thread loop
    while (!kthread_should_stop()) {
        printk(KERN_INFO "Thread running\n");
        msleep(1000);  // Sleep 1 second
    }

    return 0;  // Exit code
}

// Create thread
static struct task_struct *my_thread;

int start_thread(struct my_data *data) {
    my_thread = kthread_create(thread_function,
                               data,
                               "my_thread_%d", 0);
    if (IS_ERR(my_thread))
        return PTR_ERR(my_thread);

    // Wake up the thread
    wake_up_process(my_thread);
    return 0;
}
```

Stopping Kernel Threads

```
// In module exit
int stop_thread(void) {
    if (my_thread) {
        kthread_stop(my_thread);
        my_thread = NULL;
    }
    return 0;
}

// Thread-safe stop check
if (kthread_should_stop())
    break;  // Exit thread loop
```

Binding Threads to CPUs

```
// Bind thread to specific CPU
kthread_bind(my_thread, 0);  // CPU 0

// Run on any CPU (default)
// No call needed
```

26. KERNEL TIMERS (STANDARD)

Introduction to Kernel Timers

Kernel timers allow scheduling a function to execute after a specified time. They are based on jiffies (system ticks) and provide millisecond granularity.

Timer Structure

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

struct timer_list {
    struct list_head entry;
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
};
```

Setting Up Timers

```
// Static declaration
static struct timer_list my_timer;

// Timer callback
void timer_callback(unsigned long data) {
    struct my_device *dev = (struct my_device *)data;
    printk(KERN_INFO "Timer expired\n");

    // Reschedule if needed
    mod_timer(&my_timer, jiffies + HZ);  // 1 second
}

// Initialize timer
void init_timer_func(struct my_device *dev) {
    timer_setup(&my_timer, timer_callback, 0);
    // Old way (deprecated):
    // my_timer.function = timer_callback;
    // my_timer.data = (unsigned long)dev;
    // init_timer(&my_timer);
}
```

Timer Operations

```
// Start or modify timer
mod_timer(&my_timer, jiffies + HZ);  // 1 second from now

// Delete timer
del_timer(&my_timer);

// Delete and wait for pending handlers
del_timer_sync(&my_timer);

// Check if timer is pending
if (timer_pending(&my_timer))
    printk(KERN_INFO "Timer is pending\n");

// Useful constants
```

```
HZ          // Ticks per second (typically 100-1000)
msecs_to_jiffies(ms)  // Convert milliseconds
jiffies_to_msecs(j)   // Convert jiffies
```

Timer Example

```
static struct timer_list heartbeat_timer;
static int heartbeat_count = 0;

void heartbeat_callback(unsigned long data) {
    heartbeat_count++;
    printk(KERN_INFO "Heartbeat #%d\n", heartbeat_count);

    // Reschedule every 5 seconds
    mod_timer(&heartbeat_timer,
              jiffies + (5 * HZ));
}

static int __init driver_init(void) {
    timer_setup(&heartbeat_timer,
                heartbeat_callback, 0);

    // Start timer (execute in 1 second)
    mod_timer(&heartbeat_timer, jiffies + HZ);
    return 0;
}

static void __exit driver_exit(void) {
    del_timer_sync(&heartbeat_timer);
}
```

27. HIGH-RESOLUTION TIMERS (HRT)

When Standard Timers Aren't Enough

Standard kernel timers are limited to jiffies granularity (milliseconds). Applications requiring nanosecond precision need High-Resolution Timers.

HRT Advantages

✓ Nanosecond resolution

✓ 64-bit timestamps

✓ Red-black tree based implementation

✓ Better for multimedia and real-time applications

High-Resolution Timer API

```
#include <linux/hrtimer.h>
#include <linux/ktime.h>

// HRT function
enum hrtimer_restart hrt_callback(struct hrtimer *timer) {
    printk(KERN_INFO "HRT fired\n");

    // Reschedule every 100ms
    hrtimer_forward_now(timer,
                        ktime_set(0, 100000000));

    return HRTIMER_RESTART;
}

// Initialize
static struct hrtimer my_hrtimer;

void setup_hrt(void) {
    hrtimer_init(&my_hrtimer,
                 CLOCK_MONOTONIC,
                 HRTIMER_MODE_REL);

    my_hrtimer.function = hrt_callback;

    // Start timer (1 second from now)
    hrtimer_start(&my_hrtimer,
                  ktime_set(1, 0),
                  HRTIMER_MODE_REL);
}

// Cleanup
void cleanup_hrt(void) {
    hrtimer_cancel(&my_hrtimer);
}

// Time creation
ktime_t t = ktime_set(secs, nanosecs);
```

28. GPIO (GENERAL PURPOSE INPUT/OUTPUT)

GPIO Basics

GPIO pins provide basic hardware interface for reading digital inputs and driving digital outputs. The Linux kernel provides a unified GPIO interface across different hardware platforms.

GPIO API

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

#define GPIO_PIN 17

// Validate GPIO number
if (!gpio_is_valid(GPIO_PIN)) {
    printk(KERN_ERR "Invalid GPIO\n");
    return -EINVAL;
}

// Request GPIO
if (gpio_request(GPIO_PIN, "my_gpio")) {
    printk(KERN_ERR "Cannot request GPIO\n");
    return -EINVAL;
}

// Set as output with initial value
gpio_direction_output(GPIO_PIN, 0);

// Set GPIO value
gpio_set_value(GPIO_PIN, 1);  // High
msleep(100);
gpio_set_value(GPIO_PIN, 0);  // Low

// Set as input
gpio_direction_input(GPIO_PIN);

// Read GPIO value
int value = gpio_get_value(GPIO_PIN);

// Get GPIO as interrupt
int irq = gpio_to_irq(GPIO_PIN);
request_irq(irq, gpio_handler,
            IRQF_TRIGGER_RISING,
            "gpio_int", NULL);

// Export to userspace
gpio_export(GPIO_PIN, 1);  // Allow direction change

// Cleanup
gpio_unexport(GPIO_PIN);
free_irq(irq, NULL);
gpio_free(GPIO_PIN);
```

GPIO from User Space

```
# Export GPIO to user space
echo 17 > /sys/class/gpio/export
```

```
# Set direction to output
echo "out" > /sys/class/gpio/gpio17/direction

# Set GPIO high
echo 1 > /sys/class/gpio/gpio17/value

# Set GPIO low
echo 0 > /sys/class/gpio/gpio17/value

# Set direction to input
echo "in" > /sys/class/gpio/gpio17/direction

# Read GPIO value
cat /sys/class/gpio/gpio17/value

# Unexport GPIO
echo 17 > /sys/class/gpio/unexport
```

CONCLUSION AND BEST PRACTICES

Development Guidelines

Error Handling: Always check return values. Device drivers must be robust and handle all error conditions.

Resource Management: Properly allocate and deallocate all resources (IRQs, memory, GPIO, etc.).

Synchronization: Use appropriate locking mechanisms (mutex for long waits, spinlock for short sections).

Code Quality: Follow kernel coding standards. Use checkpatch.pl to verify code style.

Documentation: Document IOCTL commands, parameters, and hardware requirements thoroughly.

Testing: Test under various conditions: normal operation, error scenarios, edge cases.

Version Compatibility: Maintain compatibility with older kernel versions when possible.

Performance: Profile and optimize critical paths. Use perf and other tools for analysis.

Module Parameters: Support configurable parameters using module_param for flexibility.

Debugging: Use printk() or pr_*() macros liberally during development, remove before production.

Further Resources

→ Linux Kernel Documentation: https://www.kernel.org/doc/

→ Kernel Source: https://github.com/torvalds/linux

→ Linux Device Drivers (Book): O'Reilly

→ Kernel Mailing List: linux-kernel@vger.kernel.org

→ Device Tree Documentation: kernel.org/devicetree

Final Note: Linux device driver development is a rewarding skill that requires deep understanding of kernel internals, hardware interfaces, and synchronization concepts. The journey from learning to mastery involves continuous practice, reading kernel source code, and contributing to the Linux community. Start with simple drivers and gradually tackle more complex projects.