Linux Kernel Modules, Character Device Drivers, and GPIO

NXP Linux Kernel Summer School

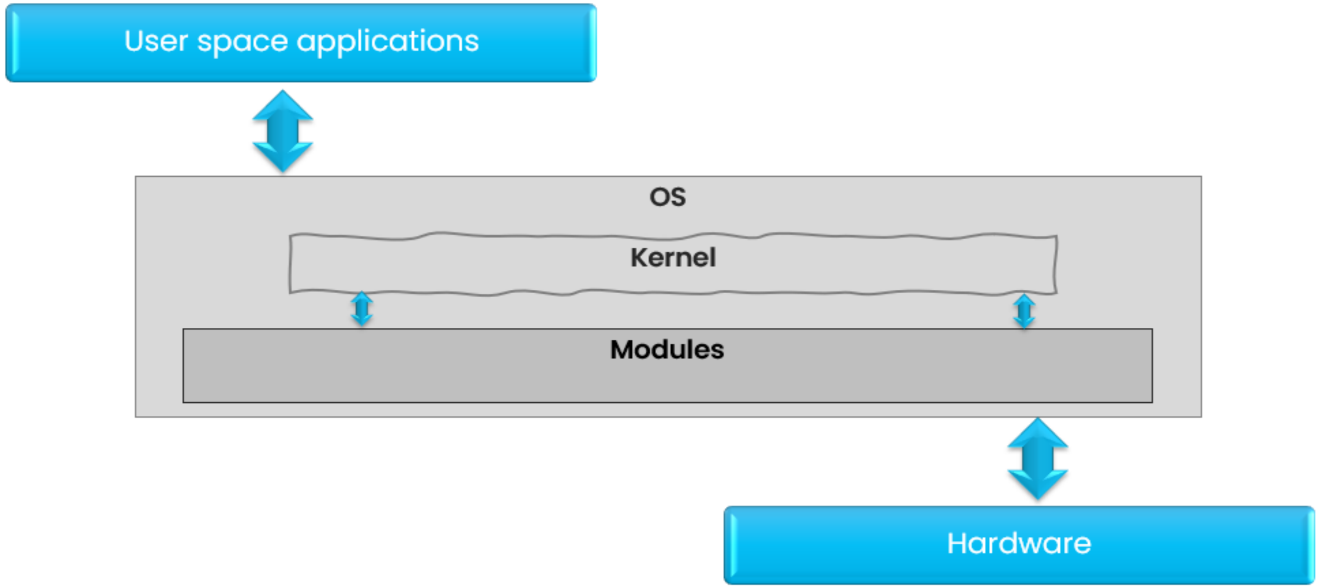
July 2025

Agenda

- Linux Kernel Modules
- Kernel space vs User Space
- Character Device Drivers
- IOCTL
- GPIO
- Timer

What is a Kernel Module?

- Code that executes as part of the Linux kernel
  - Pieces of code that can be loaded and unloaded into the kernel upon demand
- Extends the capabilities and sometimes might modify the behavior of the kernel
- Without modules, one would have to build monolithic kernels and add new functionality directly into the kernel image
- Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality



Anatomy of a Kernel Module

- Several typical components:
  - MODULE_AUTHOR("Jane Doe")
  - MODULE_LICENSE("GPL")
    - The license **must** be an open-source license (GPL,BSD, etc.) or you will "taint" your kernel

- `int init_module(void)`
  - Called when the kernel loads the module
  - Initialize all stuff here
  - Return 0 if all went well, negative if something wrong

- `void cleanup_module(void)`
  - Called when the kernel unloads the module
  - Free all resources here

Kernel Module utilities

- lsmod - Show all loaded modules
- insmod - Insert a Module (excludes dependencies)

```
$sudo insmod <module_name>
```

- modprobe - Load the kernel module plus any module dependencies

```
$sudo modprobe <module_name>
```

- modinfo - Show information about a module

```
$modinfo <module_name>
```

- depmod - Build module dependency database

```
$/lib/modules/$(uname -r)/modules.dep
```

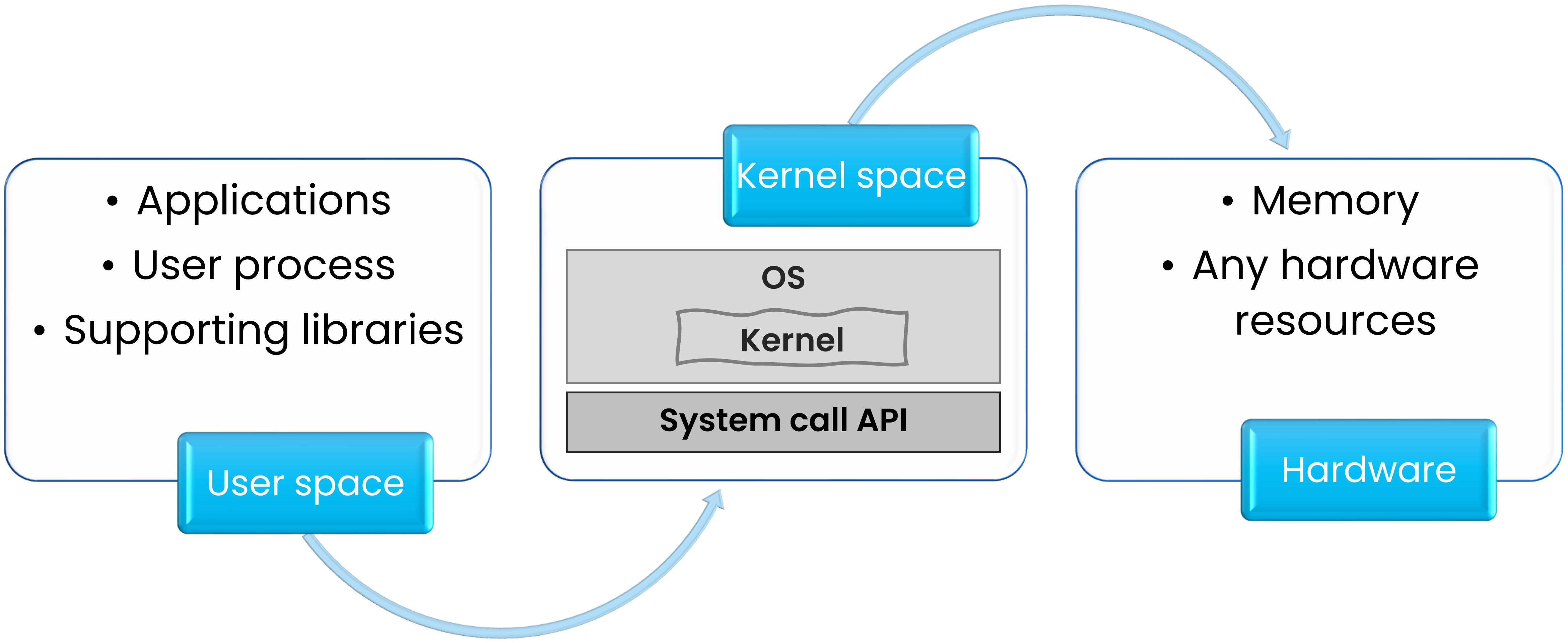
- rmmod - Remove a module

```
$rmmod <module_name>
```

- Show the log

`$dmesg` or `$cat /var/log/syslog`

Kernel space vs User space



Kernel space vs User space

Feature / Aspect	Kernel space	User space
Purpose	Runs the operating system core, drivers, and services	Runs applications and user programs
Access to Hardware	Full access to hardware and memory	No direct access — must go through syscalls or drivers
Example Code	Kernel modules, drivers, system calls	Shells, editors, browsers, user-space apps
Memory Isolation	Can access and modify any memory (requires caution)	Isolated from kernel for safety
Communication with Kernel	Provides syscalls, reads/writes data from user space	Through syscalls, ioctl(), file ops

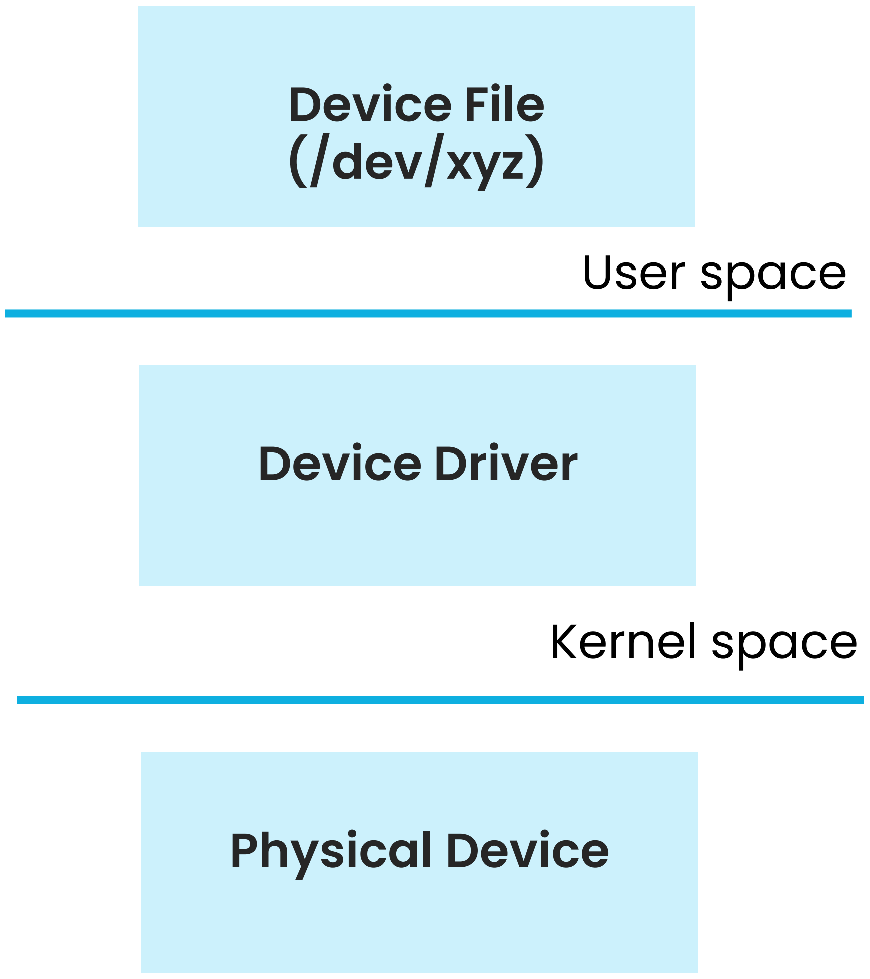
Kernel space vs User space - cont

Feature / Aspect	Kernel space	User space
Crash Impact	Crashes can bring down the entire system (kernel panic)	Crashes affect only the application
Data Sharing	Must use interfaces like copy_from_user()	Must use interfaces like copy_to_user()
Execution Privileges	Runs in ring 0 (highest privilege level)	Runs in ring 3 (lower privilege)
Development Tools	Kernel headers, printk(), dmesg, debugging tools	GCC, GDB, strace, valgrind
Programming API	Kernel API (linux/module.h, linux/fs.h, etc.)	Standard C libraries (libc, stdio.h)

Device drivers vs device files

- Every device is represented by a file in `/dev/`
- Device Driver: Kernel Module that controls a device
- Device File:
  - Interface for the Device Driver to
    - read from or write to a physical device
  - Also known as Device Nodes
  - Created with `mknod` system call

```
mknod [name] <c/b> <major> <minor>
```



Device files

- Character device
  - Stream of data one character at a time
  - No restriction on number of bytes
- Block device
  - Random access to block of data
  - Can buffer and schedule the requests

```
$ ls -l /dev/
...
crw-------    1 root root     10,  60 Dec 15  2023 cpu_dma_latency
crw-------    1 root root     10, 203 Dec 15  2023 cuse
brw-rw----    1 root disk    253,   0 Dec 15  2023 dm-0
brw-rw----    1 root disk    253,   1 Dec 15  2023 dm-1
```

- Internally the kernel identifies each device by a triplet of information
  - Type – character or block
  - Major number – typically the category of devices
  - Minor number – typically the identifier of the device

```
crw-rw-rw- 1 root   root    1, 3   Feb 23 1999  null
crw------- 1 root   root   10, 1   Feb 23 1999  psaux
crw------- 1 rubini tty     4, 1   Aug 16 22:22 tty1
crw-rw-rw- 1 root   dialout 4, 64  Jun 30 11:19 ttyS0
crw-rw-rw- 1 root   dialout 4, 65  Aug 16 00:00 ttyS1
crw------- 1 root   sys     7, 1   Feb 23 1999  vcs1
crw------- 1 root   sys     7, 129 Feb 23 1999  vcsa1
crw-rw-rw- 1 root   root    1, 5   Feb 23 1999  zero
```

Character device driver

- Handles byte-stream-based devices
- Implements file operations (open, read, write, release)
- Appears as a file in /dev/
- Interacts with user space via syscalls

Character device driver implementation

Device Driver -> Kernel Module

Device File -> /dev/<filename>

Register the device ->
register_chrdev / alloc_chrdev_region
mknod
struct file_operations

Implement file operations ->
open/release/read/write/

Unregister the device ->
unregister_chrdev / unregister_chrdev_region

Creation of Character Device Driver: manual vs automatic

Feature / Aspect	Manual (Static)	Automatic (Dynamic)
Device Number Allocation	Hardcoded major number via register_chrdev()	Dynamically allocated via alloc_chrdev_region()
/dev Node Creation	Must use mknod manually in user space	Automatically created with class_create() and device_create()
Device Cleanup	Simple unregister_chrdev()	Requires device_destroy(), class_destroy(), and unregister_chrdev_region()
Ease of Use	Simpler for learning or quick testing	More complex setup, but cleaner and scalable
Risk of Conflict	Possible conflict if major number is already in use	No conflict — kernel assigns a free major number
Portability	Less portable (hardcoded major may not work elsewhere)	More portable and future-proof
Recommended?	For simple labs or quick demos	For real-world drivers

Reading / Writing from character device

```
$ cat /dev/my_chardev
```

- calls `read()` function

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

```
$ echo "hello" > /dev/my_chardev
```

- calls `write()` function

- `unsigned long copy_from_user (void * to, const void __user * from, unsigned long n);`

Comparison: copy_to/from_user vs simple_read/write_from/to_buffer

Feature / Aspect	copy_to/from_user	simple_read/write_from/to_buffer
Purpose	Manual memory copying between user/kernel space	Simplified buffer management for common read/write cases
Level of Abstraction	Low-level, full control	High-level, abstracted utility functions
Typical Usage	When handling custom logic, complex structures	When using a static or predefined buffer
Handles file offset	No — you must manage it yourself	Yes — automatically managed
Buffer Management	Manual (you allocate, copy, validate)	Automatically handles length and offset constraints
Error-prone?	Yes — must check return values, handle edge cases	Less error-prone — includes validation logic

IOCTL

- Input/Output Control
- Used to send custom commands to a device driver via a file descriptor
- Allows communication beyond `read()` and `write()`
- `_IO, _IOR, _IOW, _IOWR`
  - These macros define the type of data exchange between user space and kernel
- Driver-side: `.unlocked_ioctl`
  - Called when user space invokes `ioctl(fd, cmd, arg)`
  - Use `copy_from_user()` and `copy_to_user()` for data exchange

Introduction to GPIO

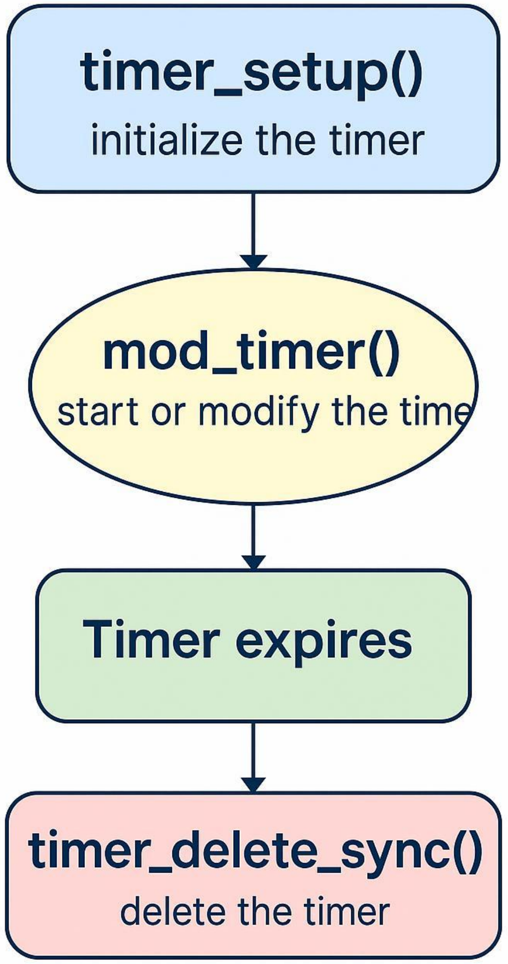
- GPIO stands for General-Purpose Input/Output
- It is a digital pin on a microcontroller or processor
- Can be set to:
  - Input: Read signals (e.g., button press)
  - Output: Send signals (e.g., turn on LED)
- Common Uses
  - Detecting button presses
  - Controlling LEDs, buzzers, or relays
  - Talking to sensors or modules
- `libgpiod -` modern GPIO Interface in Linux
  - C library and CLI toolset for controlling GPIO lines on Linux via the character device interface (/dev/gpiochipN).

Basic libgpiod API Functions

Function	Description	Typical Use
gpiod_get ()	Obtain a GPIO for a given GPIO function	Start GPIO access session, set pin as input/output
gpiod_put()	Dispose of a GPIO descriptor	End GPIO access session
gpiod_set_value()	Assign a gpio's value	Turn on/off an LED, trigger signal
gpiod_get_value()	Return a gpio's value	Read button state or sensor signal

Linux Kernel Timers - introduction

- Allow deferred execution of code in a future context
- Widely used in drivers, networking, and other kernel subsystems for managing time-based events



Summary of Exercises

- Basic char driver
- String reversal in the kernel
- GPIO control using write()/read()
- IOCTL-based control
- Blinking an LED with a timer

Questions?

- Have Fun!

Brighter
Together

nxp.com