EEEN301 Embedded systems

Lecture 17 – Device drivers

Device Drivers, User Space I/O, and Loadable Kernel Modules

Zynq
Vivado 2015.4 and PetaLinux 2015.4

Objectives

- **After completing this module, you will be able to:**
  - Explain the concepts of the Linux device driver model
  - Identify the role and usage of loadable kernel modules
  - Understand the two approaches to userspace drivers
    - `/dev/mem`
    - UIO framework

Outline

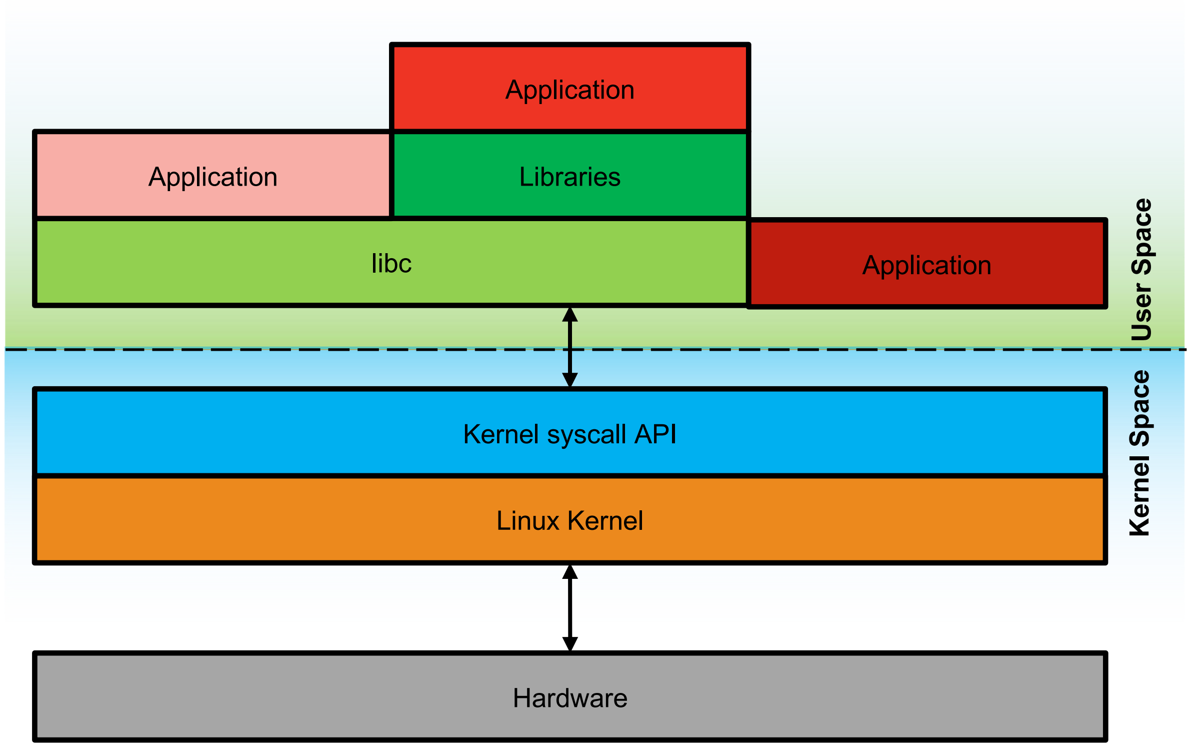
- ***Linux Device Driver Overview***
- **Loadable Modules**
  - Concepts
  - Considerations
- **User Space I/O**
  - Concepts
  - Direct Access to `/dev/mem`
  - User Space I/O (UIO) Framework

User Space vs Kernel Space

- User space is virtualized memory
- Kernel deals with absolute memory
- Kernel must be bullet proof, because it can access anything in the system
- If there is an error, system crashes
- Must follow rigid set of rules – "privileged" mode
- How can a user application access a physical address if the kernel either protects or virtualizes that address?

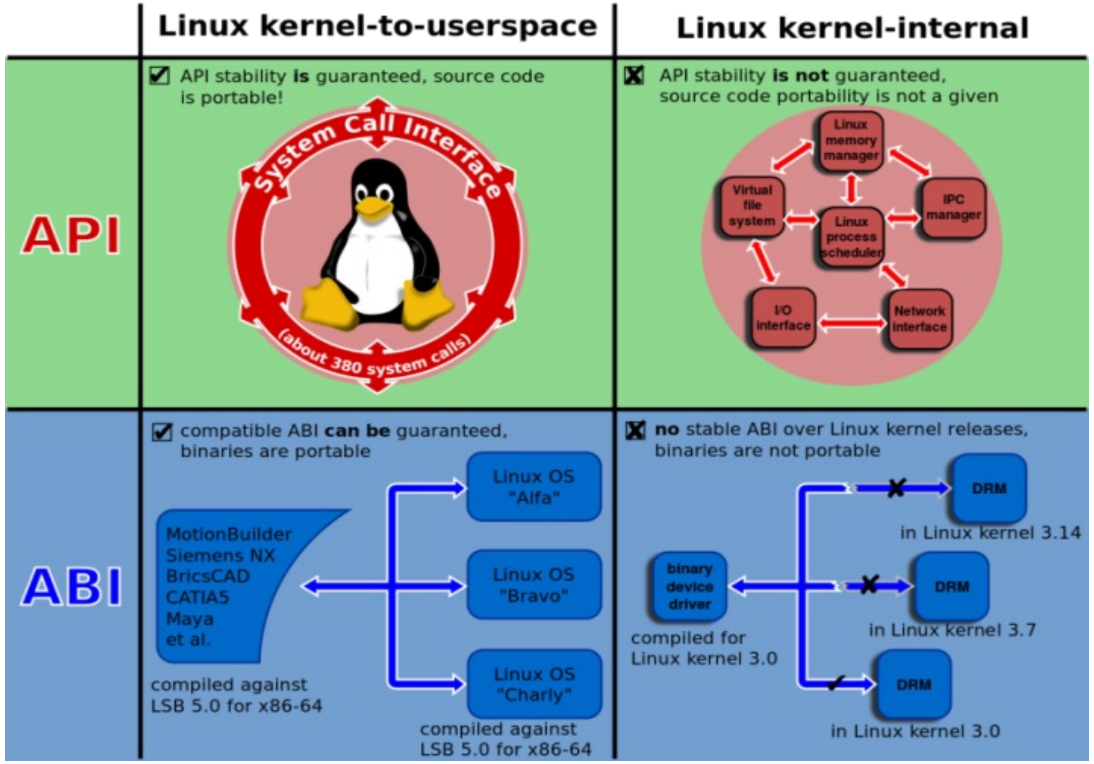
Linux Kernel – Kernel Space vs User Space

- **Kernel Space**
  - Virtual and Physical memory
  - CPU 'Kernel/Supervisor Mode' (ARM Privileged)
- **User Space**
  - Virtual memory only (kernel handles the mapping and page faults)
  - CPU 'User Mode' (ARM Unprivileged)
  - All hardware access via kernel syscall interface



Linux Kernel – Kernel Space vs User Space

- **Drivers**
  - In-built drivers and kernel modules are all run within kernel space
  - Kernel interfaces for drivers in user space
- **ABI/API Compatibility**
  - API = Application Programmers Interface
    - Source code interface, a set of functions for the programmer.
  - ABI = Application Binary Interface
    - Binary code interface, a set of precompiled modules or libraries called by the compiler.
  - Kernel to User API/ABI compatibility is stable
  - Inter-Kernel API/ABI is not stable



☑ API stability **is** guaranteed, source code is portable!

☒ API stability **is not** guaranteed, source code portability is not a given

☑ compatible ABI **can be** guaranteed, binaries are portable

☒ **no** stable ABI over Linux kernel releases, binaries are not portable

The Linux Device Driver Model (1)

- **Linux supports**
  - Thousands of different devices
  - Numerous device categories
    - Network, display, storage
    - user interface
    - sensors/clock sources
    - ...
  - Many bus architectures
    - PCI/PCIe
    - USB
    - SPI/I2C
    - ...
- **Needs a very sophisticated (and complicated) device driver model**

The Linux Device Driver Model (2)

- **At the highest level**
  - Character
    - e.g. keyboard/mouse, parallel port, Bluetooth, console, terminal, sound, video, ...
    - Most custom IP drivers will be of this kind
  - Block
    - Hard/floppy disks, ram disks, CD/DVD
  - Network
    - Ethernet, CAN, Wi-Fi, ...

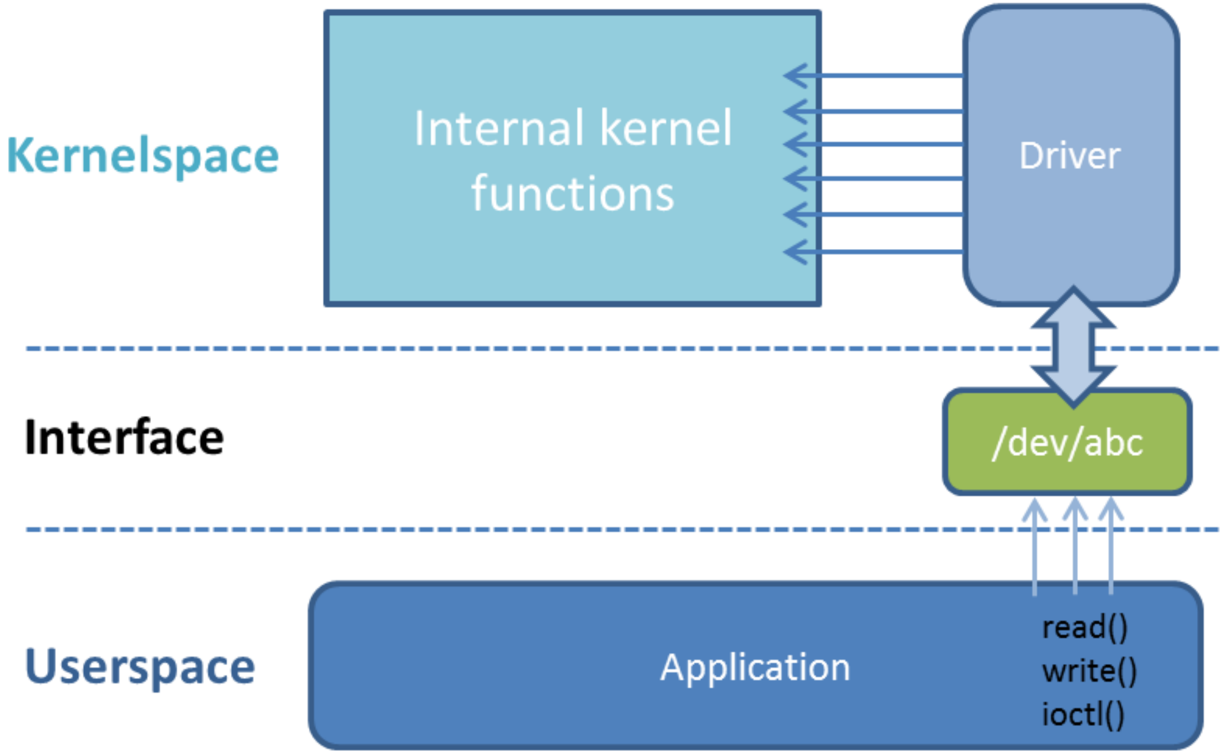
Device Nodes and Numbers

- **Device numbers**
  - Char and block devices identified by a pair of numbers
    - (major,minor)
  - All devices of the same type share a major number
    - `'$ cat /proc/devices'` lists all drivers and devices
- **Device nodes**
  - Symbolic file-system handle to a device
    - /dev/ttyS0 – serial port 0
    - /dev/fb0 – frame buffer 0

Conventional Driver

- This sort of driver uses many internal kernel functions and macros
- Must write an in-kernel driver from scratch
- Debugging the driver will be challenging

when debugging an application



Device Drivers for Custom Hardware

- **Writing custom drivers is a deep topic**
  - Could easily cover over a one-week training
- **Are there any shortcuts?**
  - There are two approaches: `/dev/mem` and user space I/O framework
  - Direct access to device registers via `/dev/mem`
    - Memory map `/dev/mem` into application address space
    - Access device via pointer returned from mmap()
    - Very simple, quick to prototype
    - Limited functionality
      - No IRQ handling
  - UserSpace IO (UIO) framework
    - Generic kernel framework for user space drivers
    - Simple interface, little (or no) custom device driver code at all
    - Can do basic user space IRQ handling

Device Driver Interface

- **Device driver implements standard kernel API**
  - Hooks or entry points for
    - `open/release`
    - `read/write/ioctl/mmap`
    - Interrupts
- **Device driver registration**
  - Initialise a `file_operations` structure with pointers to handler functions
  - Register driver with kernel
- **At run time, kernel automatically calls the driver entry points in response to application behavior**
  - `open/read/write/close/...`
- **For details, see *Linux Device Drivers*, 3rd ed by Corbet, Rubini, Kroah-Hartmann, O'Reilly Press, 2005**

Platform Configuration

- **How do we know what devices are present in the system (and their address/IRQ)?**
  - Some buses are self-describing, e.g. PCI/PCIe/USB
    - OS queries configuration space to find devices
    - Assigns device addresses and IRQs
    - Drivers query this data to access their device
- **System-on-Chip buses are typically static**
- **For ARM Cortex-A9 etc, the device tree (DTS) is used**
- **Device tree enables configuration depending on what is loaded into the system**
  - Standard and custom IP drivers can be loaded

The Device Tree

- **DTS file**
  - Device Tree Source
  - Textual description of system device tree
- **DTB**
  - Device Tree Blob
  - Compiled, binary representation of DTS
- **DTC**
  - Device Tree Compiler
  - Converts DTS to DTB

```
/ {
  cpus {
    ps7_cortexa9_0: cpu@0 {
      compatible = "xlnx,ps7-cortexa9";
      ...
    } ;
    ps7_cortexa9_1: cpu@1 {
      compatible = "xlnx,ps7-cortexa9";
      ...
    } ;
  } ;
  ps7_axi_interconnect_0: amba@0 {
    compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
    ranges ;
    ps7_ddrc_0: ps7-ddrc@f8006000 {
      compatible = "xlnx,zynq-ddrc-1.00";
      reg = < 0xf8000000 0x1000 >;
    }
    ps7_ethernet_0: ps7-ethernet@e000b000 {
      compatible = "xlnx,ps7-ethernet-1.00.a";
      ...
    } ;
    ps7_qspi_0: ps7-qspi@e000d000 {
      compatible = "xlnx,ps7-qspi-1.00.a";
      ...
    } ;
    ps7_gpio_0: ps7-gpio@e000a000 {
      compatible = "xlnx,ps7-gpio-1.00.a";
    } ;
    ps7_usb_0: ps7-usb@e0002000 {
      compatible = "xlnx,ps7-usb-1.00.a";
    } ;
    ...
    ps7_uart_1: serial@e0001000 {
      compatible = "xlnx,ps7-uart-1.00.a", "xlnx,xuartps";
      ...
    } ;
  } ;
} ;
```

Outline

- **Linux Device Driver Overview**
- ***Loadable Modules***
  - Concepts
  - Considerations
- **User Space I/O**
  - Concepts
  - Direct Access to `/dev/mem`
  - User Space I/O (UIO) Framework
- **Summary**

Loadable Kernel Modules

- **Device drivers can be statically or dynamically linked to the kernel**
  - Kernel modules provide dynamic linking capability
  - Driver stored in filesystem as a `.ko` file
  - Loaded into the kernel with `ldmod`
  - Removed with `rmmod`

```
# ldmod mydriver
...
# rmmod mydriver
...
```

Loadable Kernel Modules – Basic Usage

- **Use `lsmod` command to list installed modules**

```
# lsmod
Module                      Size  Used by
mydriver                   30764     1
```

- **“Used by” count shows how many clients**
  - Processes holding open device nodes
  - Internal kernel usages of module
- **Can only `rmmod` when usage count is zero**

Loadable Kernel Modules - Desktop vs Embedded

- **Modules extensively used in desktop systems**
  - Keeps core kernel small while allowing support for many different devices
    - Disk space much cheaper than memory
    - Only load those modules required
- **Still useful in embedded context**
  - Can reduce core kernel boot time
  - Double-cost with memory-based file systems
    - One copy on disk (in memory)
    - One copy in kernel memory
  - Helpful during development phase

Device drivers

- **Device drivers and other Kernel modules do not have a “main”**
- **Instead they have a set of functions.**
- **Two are required to manage the loading and unloading of the module:**
  - module_init(module); Used to initialise the module functionality and to register it. Called during ldmod.
  - module_exit(module); Used to clean things up and de-register the module. Called during rmmod.

- **To interact with the driver, usually 4 or more functions are used, they are mapped via a file operations data structure (fs.h).**
  - dev_open(): Called each time the device is opened from user space.
  - dev_read(): Called when data is sent from the device to user space.
  - dev_write(): Called when data is sent from user space to the device.
  - dev_release(): Called when the device is closed in user space.

- **We will examine this more closely in the lab. For more info:**

http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/

Outline

- **Linux Device Driver Overview**
- **Loadable Modules**
  - Concepts
  - Considerations
- ***User Space I/O***
  - Concepts
  - Direct Access to `/dev/mem`
  - User Space I/O (UIO) Framework
- **Summary**

User Space Device Access

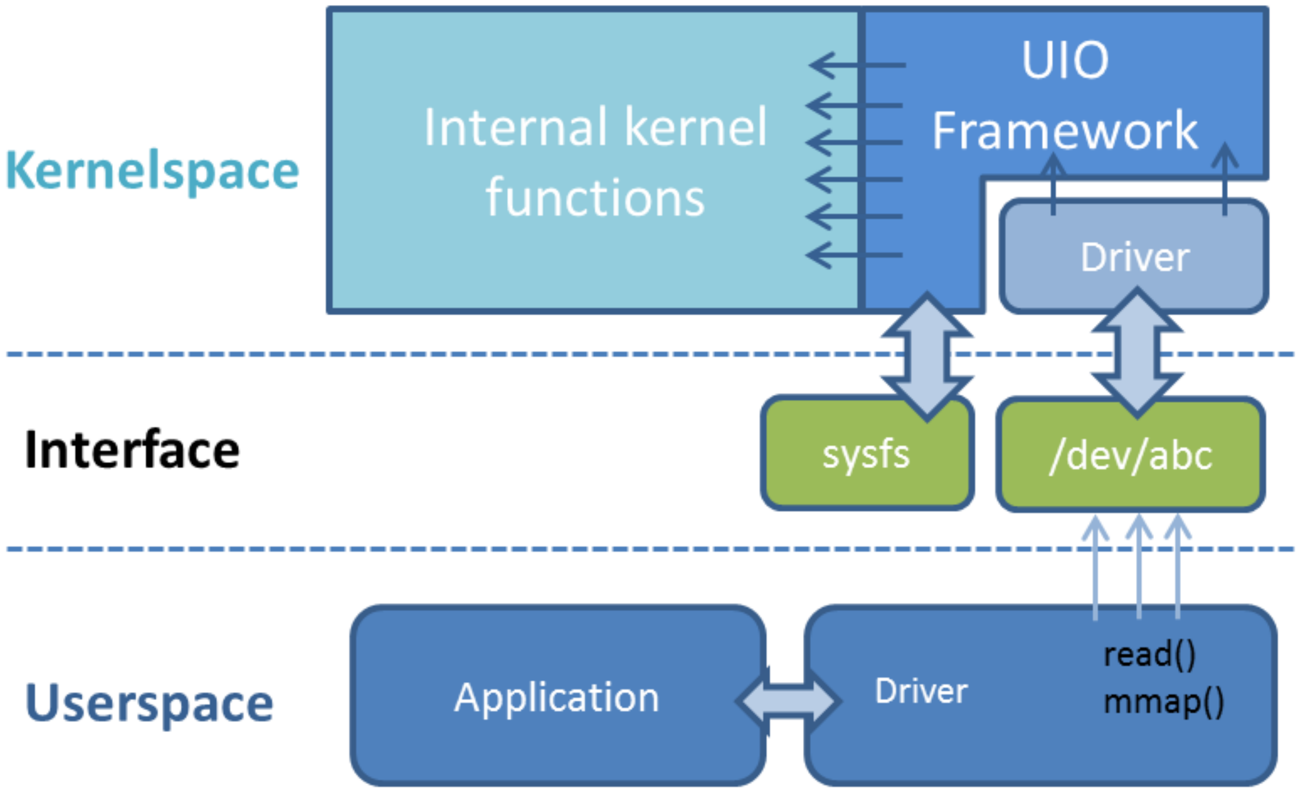
- **Commonly from traditional embedded developers**

"Can't I just access my hardware from user space?"

- **No! Well, yes, but there are rules...**
- **Two approaches considered (may not be supported, or could be slightly different)**
  - Direct access to `/dev/mem`
  - User Space IO (UIO) framework

UIO Driver

- By using `/dev/mem`, Linux is able to map physical device memory to an address accessible from user space
- UIO improves stability by preventing user space from mapping memory that does not belong to the device
- A small kernel driver calls only a few kernel functions
- UIO framework generates a set of directories and attribute files in sysfs Linux kernel memory management



User Space Device Access - `/dev/mem`

- **`/dev/mem`**
  - Userspace interface to system address space
  - Accessed via `mmap()` system call
  - Must be root or have appropriate permissions
  - Quite a blunt tool – must be used carefully
    - Can bypass protections provided by the MMU
    - Possible to corrupt kernel, device or memory of other processes

User Space Device Access - `/dev/mem` Example

Open `/dev/mem`

Memory map

Access via pointer

```c
/*
 * poke utility - for those who remember the good old days!
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
        int fd;
        void *ptr;
        unsigned val;
        unsigned addr, page_addr, page_offset;
        unsigned page_size=sysconf(_SC_PAGESIZE);

        fd=open("/dev/mem",O_RDWR);
        if(fd<1) {
                perror(argv[0]);
                exit(-1);
        }

        if(argc!=3) {
                printf(“Usage: poke <addr> <data>”\n”);
                exit(-1);
        }

        addr=strtoul(argv[1],NULL,0);
        val=strtoul(argv[2],NULL,0);

        page_addr=(addr & ~(page_size-1));
        page_offset=addr-page_addr;

        ptr=mmap(NULL,page_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,page_addr);
        if((int)ptr==-1) {
                perror(argv[0]);
                exit(-1);
        }

        *((unsigned *)(ptr+page_offset))=val;
        return 0;
}
```

User Space Device Access - `/dev/mem` Advantages and Disadvantages

- **Pro**
  - Very simple – no kernel module or code
  - Good for quick prototyping / IP verification
    - peek/poke utilities
  - Portable (in a very basic sense)
- **Con**
  - No interrupt handling possible
  - No protection against simultaneous access
  - Need to know physical address of IP
    - Hard-code?
- **OK for prototyping – not recommended for production**

User Space Device Access - The UIO framework

- **In Linux 2.6.22, the User space IO (UIO) API was introduced**
  - `linux-3.14/drivers/uio`
  - Allows clean, portable implementation of user space device drivers
  - Basic interrupt handling capabilities
- **Very thin kernel-level driver**
  - Register UIO device
  - Trivial interrupt handler
- **All of the real work happens in user space**

UIO - the Application Level

- **Opening the device**
  - Walk through `sysfs` mounted `/sys/class/uio/uioX (remember sys/class/LEDs)`
  - Check virtual file 'name'
  - If it matches

```
fd=open("/dev/uioX",O_RDWR);
```

- **Memory mapping the resources**

```
void *ptr=mmap(NULL, size, PROT_READ|PROT_WRITE,
                        MAP_SHARED, fd, n * PAGE_SIZE);
```

  - n is the mapping number (device specific)
- **`ptr` may now be safely used for direct access to the hardware**

UIO - Interrupt Handling

- **Several options**
  - Issuing a `read()` on the device returns number of interrupts since last read call

```
read(fd, &num_irqs, sizeof(num_irqs));
```

  - Can be blocking or non blocking
    - `O_NONBLOCK` flag in `open()` call
  - `select()` system call on the file descriptor
    - optionally block until an IRQ occurs
  - Actual handling of the interrupt is device dependent

UIO – Kernel Interface (1)

- **By default, even UIO requires a thin kernel-space driver**
  - Register and remap device address map
  - Specify IRQ handler function
  - Register driver with UIO subsystem
- **Bulk of device driver implemented in userspace**

UIO - Pros and Cons

- **Pro**
  - Benefits of `/dev/mem` and `mmap()`
    - Plus IRQ handling
  - No kernel code at all
    - If using OF_GENIRQ extensions
  - No need to recompile and reboot kernel
    - Kernel drivers can easily break the kernel and force a reboot
      - UIO driver errors not usually fatal
    - Open driver development to non-kernel developers
- **Con**
  - Interrupt model is simple but adequate
  - Subject to variable or high latency
  - No support for DMA to/from user space

- **Other**
  - Can avoid some GPL licensing issues
    - Kernel drivers/modules must be GPL licensed
    - No such requirement for user space drivers in UIO

Summary

- **Direct access to hardware through `/dev/mem` is quick and easy but limited**
  - Best for quick prototyping
- **The UIO framework allows you to quickly develop device drivers that can be controlled from user space**
  - Includes interrupt handling
- **The full Linux device driver model is still appropriate and recommended in some circumstances**