The Abdus Salam International Centre for Theoretical Physics

IAEA International Atomic Energy Agency

Joint ICTP-IAEA School on Zynq-7000 SoC and its Applications for Nuclear and Related Instrumentation

Embedded Linux Device Drivers

Fernando Rincón

fernando.rincon@uclm.es

Contents

- Xilinx Device Drivers model
  - Layered approach
- Linux Kernel Space vs User Space
- Linux Device Driver Model
- The Device Tree
- Writing Linux Drivers
  - Full Drivers (out of our scope)
  - /dev/mem approach
  - User Space I/O

C pointer arithmetic under-standing required!!

Xilinx Drivers: The Programmers View

- Provide maximum portability
  - The device drivers are provided as ANSI C source code
- Support FPGA configurability
  - Supports multiple instances of the device without code duplication for each instance, while at the same time managing unique characteristics on a per-instance basis
- Support simple and complex use cases
  - A layered device driver architecture provides both
    - Simple device drivers with minimal memory footprints
    - Full-featured device drivers with larger memory footprints
- Ease of use and maintenance
  - Xilinx uses coding standards and provides well-documented source code for developers

Xilinx Drivers: Layered Architecture

OS-based Apps

Layer 2 : OS Drivers

- portable across applications

Baremetal Apps

Layer 1 : High-level drivers

- full-featured
- portable across operating systems and processors

Layer 0 : Low-level drivers

- for simple use cases

Hardware

Xilinx Device Drivers: Levels 0 & 1

Layer 0

- Low-level device drivers
- Macros & functions to create a small system
- Characteristics:
  - Small memory footprint
  - Little to no error checking
  - Primary device features only
  - No support of device configuration parameters
  - Supports multiple instances of a device with base address input to the API
  - Polled I/O only
  - Blocking function calls

Layer 1

- high-level device drivers
- Macros & functions to utilize all of the features of a device
- Characteristics:
  - Abstract API
  - All device features & configs.
  - Multiple instances of a device
  - Polled and interrupt driven I/O
  - Non-blocking function calls to aid complex applications
  - May have a large memory footprint
  - Buffer interfaces as opposed to byte interfaces

Xilinx Device Drivers: Layers 0 & 1 Example

UARTPs Level 0

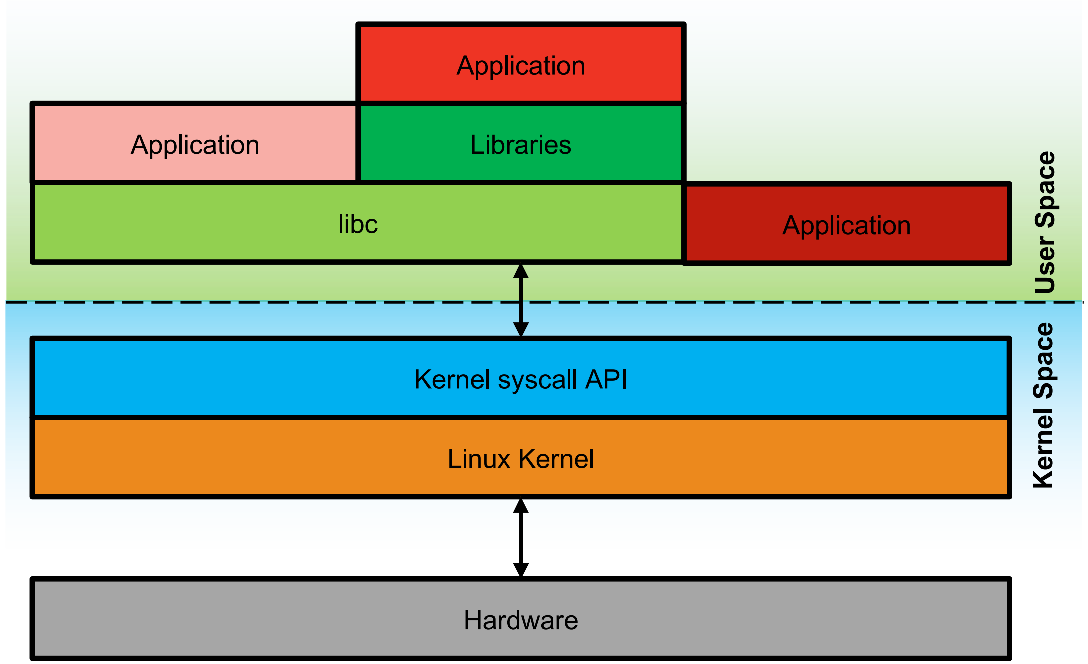
- `XuartPs_SendByte()`
  - Sends one byte using the device.
- `XuartPs_RecvByte()`
  - Receives a byte from the device.

UARTPs Level 1

- `XuartPs_CfgInitialize()`
  - Initializes a specific XUartPs instance such that it is ready to be used
- `XuartPs_Send()`
  - Sends the specified buffer using the device in either polled or interrupt driven mode.
- `XuartPs_Recv()`
  - Receive a specified number of bytes of data from the device and store it into the specified buffer.
- `XuartPs_SetBaudRate()`
  - Sets the baud rate for the device.

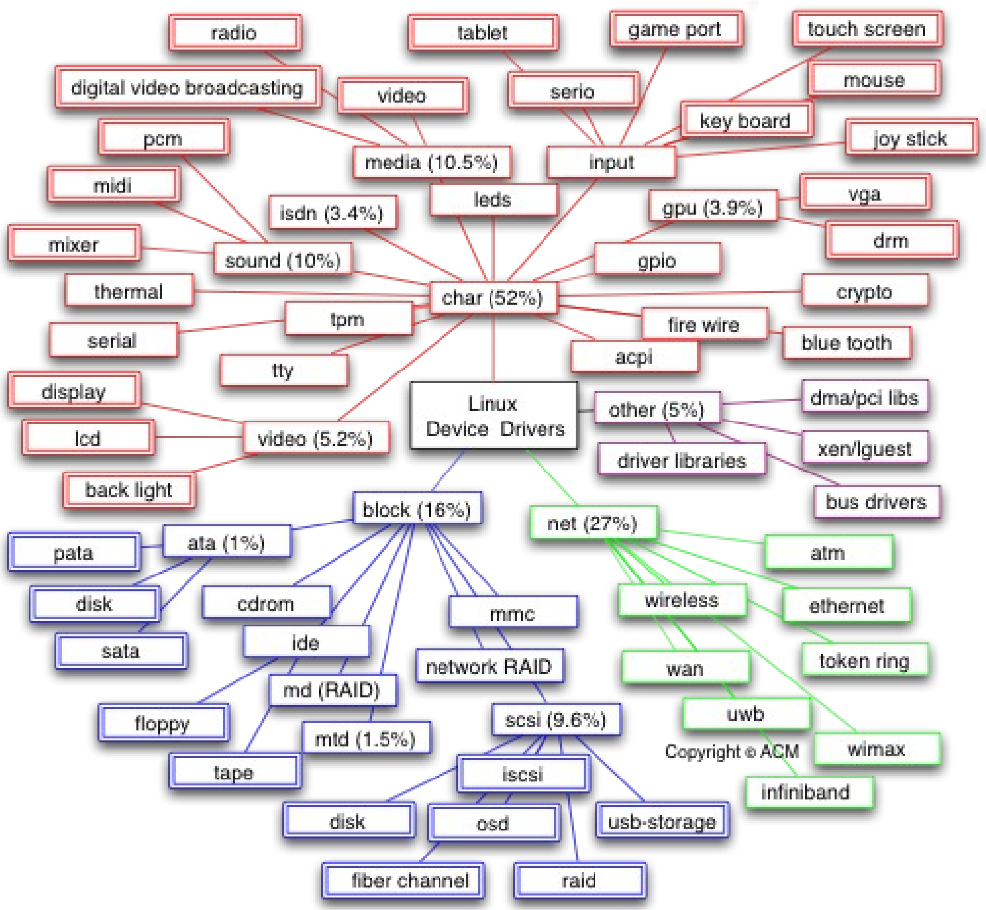
Kernel Space vs User Space

- Kernel Space
  - Virtual and Physical memory
  - CPU ‘Kernel/Supervisor Mode’ (ARM Privileged)
  - Context switching & interrupts
  - System crash in case of error
- 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 Device Driver Model

- Linux supports thousands of very different devices & bus architectures
- Requires a sophisticated a sophisticated model
- Organized around 3 types:
  - Character
    - Keyboard, mouse, console, bluetooth, ...
    - Most IP drivers
  - Block:
    - Mainly for storage
  - Network



Linux device drivers taxonomy
A. Kadav & M. Swift. University of Wisconsin-Madison

Device nodes & numbers

- Everything in linux are files, also devices
- Located at the /dev directory
- Device numbers:
  - Identified by a minor and major number
    - Same type equals same major

minor

major

```
crw-------    1 root     tty        247,   0 Aug 22 10:41 /dev/ttyPS0
```

- Device nodes:
  - Device file created when detected:
    - /dev/ttyACM0
    - /dev/fb0 – frame buffer 0

Platform configuration

- How do you know what devices are present in the system (and their address/IRQ)?
  - Some buses are enumerated; e.g., PCITM/PCIe®/USB technologies
    - OS queries PCI compliant configuration space to find devices
    - Assigns device addresses and IRQs
    - Drivers query this data to access their device
  - Based on auto-discovery
- Traditional system on chip (SoC) buses are typically static
  - Address space defined at synthesis time
  - Cannot be queried at runtime
  - They need a way to provide such information
- For a Zynq® All-programmable SoC, the device tree (DTS) is used
  - Enables configuration depending on what is loaded into the system
  - Standard and custom IP drivers can be loaded

Device Tree: Types of files

DTS file

```
{
    memory {
         device_type = "memory";
         reg = <0x0 0x20000000>;
    };

    cpus {
         cpu@0 {  compatible = "arm,cortex-a9";
                  ...};

         cpu@1 {  compatible = "arm,cortex-a9";
                  ...};
         };

    amba {
         compatible = "simple-bus";
         interrupt-parent = <0x3>;
         ranges;

         ethernet@e000b000 {
                  compatible = "cdns,zynq-gem", "cdns,gem";
                  ...};
         };

    amba_pl {
         #address-cells = <0x1>;
         #size-cells = <0x1>;
         compatible = "simple-bus";
         ranges;

         gpio@41200000 {
         #gpio-cells = <0x2>;
         compatible = "xlnx,xps-gpio-1.00.a";
         ...};
         };
};
```

- Device Tree Source
  - Textual description of the system, buses and peripherals
- Device Tree Blob
  - Binary representation of the DTS
- Device Tree Compiler
  - Converts the DTS into DTB

DTC compiler

DTB file

0010…011
…
0100…100

Device Tree: Properties

- `compatible` identifies bus controller (in case of I2C, SPI, PCI, etc.).
  - special value `compatible = "simple-bus"` => simple memory-mapped bus with no specific handling or driver.
    - Child nodes registered as platform devices
- `#address-cells` =>number of cells (i.e 32 bits values) to form the base address part in the reg property
- `#size-cells` => idem for the size part of the reg property
- `ranges` => address translation between child bus and parent bus.
  - If just `ranges;` => identity translation

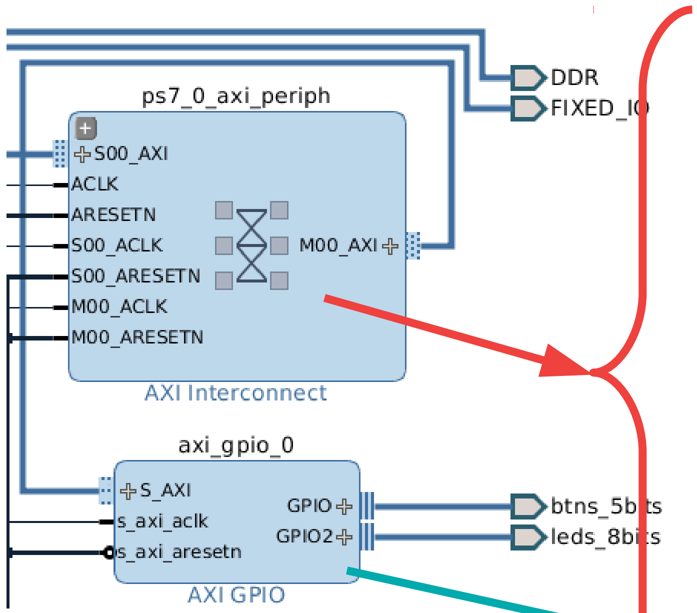
```
amba {
        compatible = "simple-bus";
        #address-cells = <0x1>;
        #size-cells = <0x1>;
        interrupt-parent = <0x3>;
        ranges;

        adc@f8007100 {
                compatible = "xlnx,zynq-xadc-1.00.a";
                reg = <0xf8007100 0x20>;
                interrupts = <0x0 0x7 0x4>;
…
```

Device Tree: Generation

- How is the device tree created?
  - PetaLinux tools includes an automated device tree generator
    - Generates device tree based on hardware design
    - All IP cores and properties exported in DTS
  - Kernel build process compiles DTS $\rightarrow$ DTB and links into kernel image
    - Per-platform DTS files in
      - `<project>/components/plnx_workspace/device-tree-generation`
        - system.dtsi: peripherals in PL
        - ps.dtsi - peripherals in PS
      - `<project>/project-spec/meta-user/recipes-dt/device-tree/files/`
        - /project-spec/meta-user/recipes-dt/device-tree/files/system-top.dts - User overwrite of default parameters and addition of new parameter
      - Changes to the system-top.dts will not be over written by subsequent petalinux-config

Device Tree: Example

AXI GPIO DTS example




```
amba_pl: amba_pl {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    Ranges ;

    axi_gpio_0: gpio@41200000 {
        #gpio-cells = <2>;
        compatible = "xlnx,xps-gpio-1.00.a";
        gpio-controller ;
        reg = <0x41200000 0x10000>;
        xlnx,all-inputs = <0x1>;
        xlnx,all-inputs-2 = <0x0>;
        xlnx,all-outputs = <0x0>;
        xlnx,all-outputs-2 = <0x1>;
        xlnx,dout-default = <0x00000000>;
        xlnx,dout-default-2 = <0x00000000>;
        xlnx,gpio-width = <0x8>;
        xlnx,gpio2-width = <0x8>;
        xlnx,interrupt-present = <0x0>;
        xlnx,is-dual = <0x1>;
        xlnx,tri-default = <0xFFFFFFFF>;
        xlnx,tri-default-2 = <0xFFFFFFFF>;
    };
};
```

compatible device Id./ driver name / version

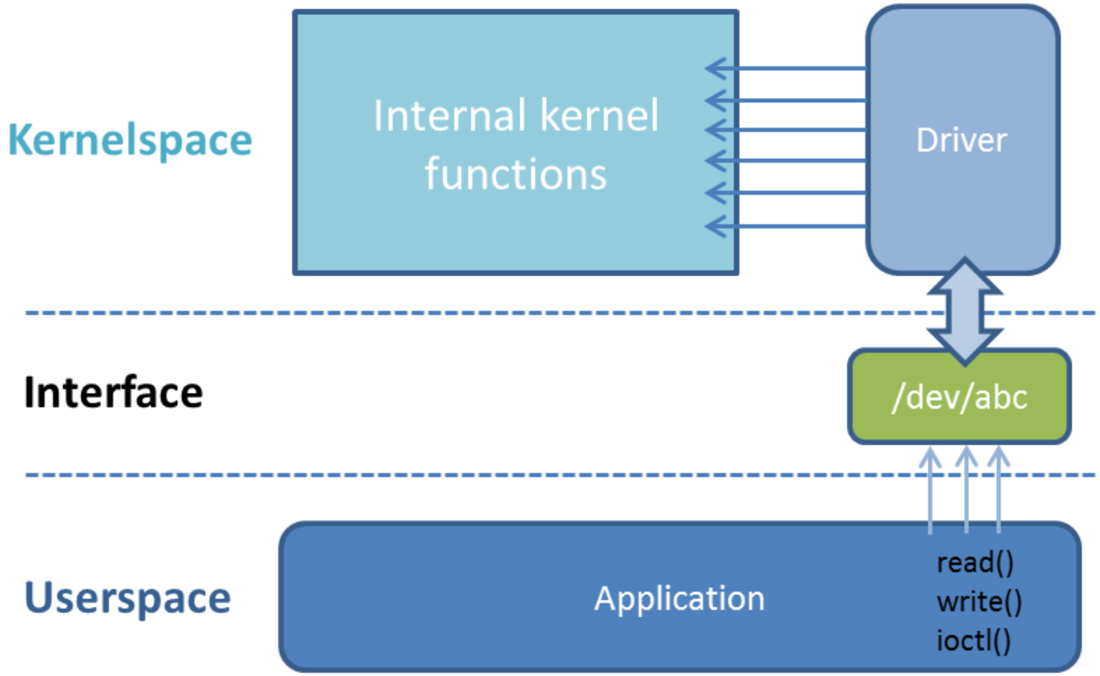
address space

custom parameters

name of device

Writing Drivers

- Developing a kernel driver is a challenging task
  - Use of internal kernel functions and macros
  - Complex to debug
  - Kernel failures lock the device
  - See: *Linux Device Drivers, 3rd ed by Corbet, Rubini, Kroah-Hartmann, O'Reilly Press, 2005*
- Simpler alternatives
  - Direct acces to memory registers
    - /dev/mem device
  - UserSpace IO (IUO)
    - Generic framework for user space drivers



/dev/mem Drivers

- Based on mapping physical device memory into the user space
  - Access through a pointer returned from `mmap()` system call
- The simplest alternative
- But no IRQ handling
- Requires root or appropriate privileges
- Be careful
  - Can bypass protections provided by the MMU
  - Can corrupt kernel, memory or other processes

Simplest Alternative!!

/dev/mem Driver example

Open
/dev/mem

Memory map

Access via
pointer

```
/* 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, 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;
}
```

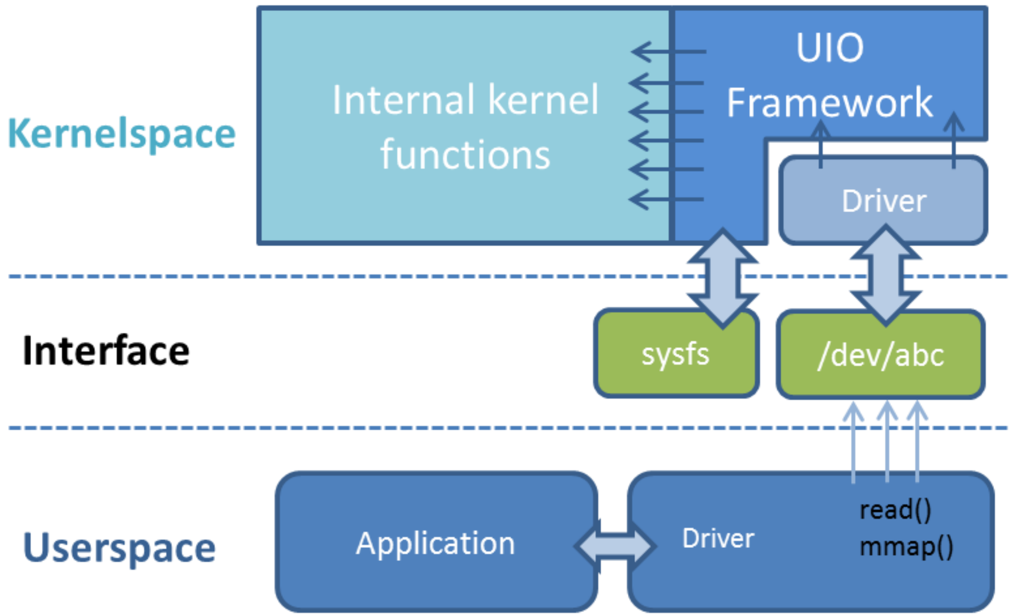
/dev/mem Pros & Cons

- Pros
  - Very simple – no kernel module or code
  - Good for quick prototyping / IP verification
  - peek/poke utilities
  - Portable (in a very basic sense)
- Cons
  - 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

UIO Drivers

- Introduced in Linux 2.6.22
  - Portable implementation of user 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
- generates a set of directories and attribute files in sysfs Linux kernel memory management

Simple but effective Alternative!!



UIO Drivers: SysFS

- Dynamic “virtual” file sytem: sysfs
- Mounted on /sys
- Representation of the device model in the user space
  - Device attributes represented as files
  - Can read/write over the files to set parameters from the user space

```
/sys/class/gpio/gpio7 # echo 1 > value
/sys/class/gpio/gpio7 # cat value
```

Useful for controlling
Devices from shell scripts

```
fprintf(file_led7, “%d”, 1);
fscanf(file_led7, “%d”, &n_ch);
```

The programmatic way

UIO Drivers: Application level

- Opening the device
  - Walk through sysfs mounted `/sys/class/uio/uioX`
  - 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 Drivers: 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 Drivers – Kernel Interface

- 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
  - The rest is implemented in the user space
- The PetaLinux tools includes an extension to UIO
  - Tag device entry in DTS file with compatible=”generic-uio”
  - Enable "Userspace I/O OF driver with generic IRQ handling" kernel option
    - Kconfig > Drivers > UserspaceIO
- No custom kernel code required at all!
- Userspace interface is the same
  - Write 0/1 to the UIO device to disable/enable IRQ
  - Read from the device to wait for IRQ
  - `mmap()` for device register access

UIO Drivers: Example

```
/* open the device */
int fd=open("/dev/uioX",O_RDWR);

/* memory map it */
volatile unsigned int *ptr = \
       mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

/* setup the device registers */
*(ptr+REG_OFFSET) = ...

/* main IRQ/processing loop */
while(1) {
              unsigned tmp = 0x1;
              /* Enable the interrupt */
              write(fd, &tmp,sizeof(tmp));

              /* Wait for an IRQ
              read(fd, &tmp, sizeof(tmp);

              /* Handle the IRQ */
              tmp = *(ptr+REG_OFFSET);
       ...
              *(ptr+REG_OFFSET) = ...
}
```

device name

UIO Drivers: Kernel Interface

- Things to watch for
  - UIO IRQ enable/disable is at kernel IRQ line level
    - Userspace code must
      - Setup device for IRQ generation
      - ACK the IRQ
    - Cannot use shared IRQs in this scheme
  - Only usable for address-mapped devices
    - Cannot support FSL, APU etc

UIO Pros & Cons

- Pros
  - 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
- Cons
  - Interrupt model is simple (but adequate in many cases)
  - Subject to variable or high latency
  - No support for DMA to/from user space