ALL PROGRAMMABLE™

Linux User Space Device Drivers

John Linn

Based on 3.14 Linux kernel

Introduction

- The purpose of this session is to educate about the options for architecting Linux device drivers
- Each method has advantages and disadvantages with the goal to present them accurately
- Hopefully you can have better conversations with customers about options they have which could help get to product quicker
- This training assumes the user has already taken an introduction session into Linux device drivers with emphasis on character and platform device drivers

Device Driver Architectures

- Linux device drivers are typically designed as kernel drivers running in kernel space
- User space I/O is another alternative device driver architecture that has been supported by the Linux kernel since 2.6.24
- People in the Linux kernel community may not always agree on the need to have user space I/O
- Industrial I/O cards have been taking advantage of user space I/O for quite some time
- For some types of devices, creating a Linux kernel driver may be overkill
- Soft IP for FPGAs can have unique requirements that don’t always fit the mold

Legacy User Space Driver Methods (/dev/mem)

- **A character driver referred to as /dev/mem exists in the kernel that will map device memory into user space**
- **With this driver user space applications can access device memory**
  - We started a bad habit of using this a lot at Xilinx
- **Memory access can be disabled in the kernel configuration as this is a big security hole (CONFIG_STRICT_DEVMEM)**
  - Most production kernels for distributions are likely to have it turned off
  - There is a distinction between memory (RAM) and devices which are memory mapped; devices are always allowed
- **Must be root user**
- **A great tool for prototyping or maybe testing new hardware, but is not considered to be an acceptable production solution for a user space device driver**
- **Since it can map any address into user space a buggy user space driver could crash the kernel**

Introduction to UIO

- **The Linux kernel provides a framework for doing user space drivers called UIO**
- **The framework is a character mode kernel driver (in drivers/uio) which runs as a layer under a user space driver**
- **UIO helps to offload some of the work to develop a driver**
- **The "U" in UIO is not for universal**
  - Devices well handled by kernel frameworks should ideally stay in the kernel (if you ask many kernel developers)
  - Networking is one area where semiconductor vendors are doing user space I/O to get improved performance
- **UIO handles simple device drivers really well**
  - Simple driver: Device access and interrupt processing with no need to access kernel frameworks

Kernel Space Driver Characteristics

Advantages

- **Runs in kernel space in the highest privilege mode to allow access to interrupts and hardware resources**
- **There are a lot of kernel services such that kernel space drivers can be designed for complex devices**
- **The kernel provides an API to user space which allows multiple applications to access a kernel space driver simultaneously**
  – Larger and more scalable software systems can be architected
- **Many drivers tend to be kernel space**
  – Asking questions in the open source community is going to be easier
  – Pushing drivers to the open source community is likely easier

Disadvantages

- **System call overhead to access drivers**
  – A switch from user space to kernel space (and back) is required
  – Overhead can be non-deterministic having impact on real time applications
- **Challenging learning curve for developers**
  – The kernel API is different from the application level API such that it takes time to become productive
- **Bugs can be fatal causing a kernel crash**
- **Challenging to debug**
  – Kernel code is highly optimized and there are different debug tools
- **Frequent kernel API changes**
  – Kernel drivers built for one kernel version may not build for another

User Space Device Driver Characteristics

Advantages

- **Less challenging to debug as debug tools are more readily available and common to normal application development**
- **User space services such as floating point are available**
- **Device access is very efficient as there is no system call required**
- **The application API of Linux is very stable**
- **The driver can be written in any language, not just “C”**

Disadvantages

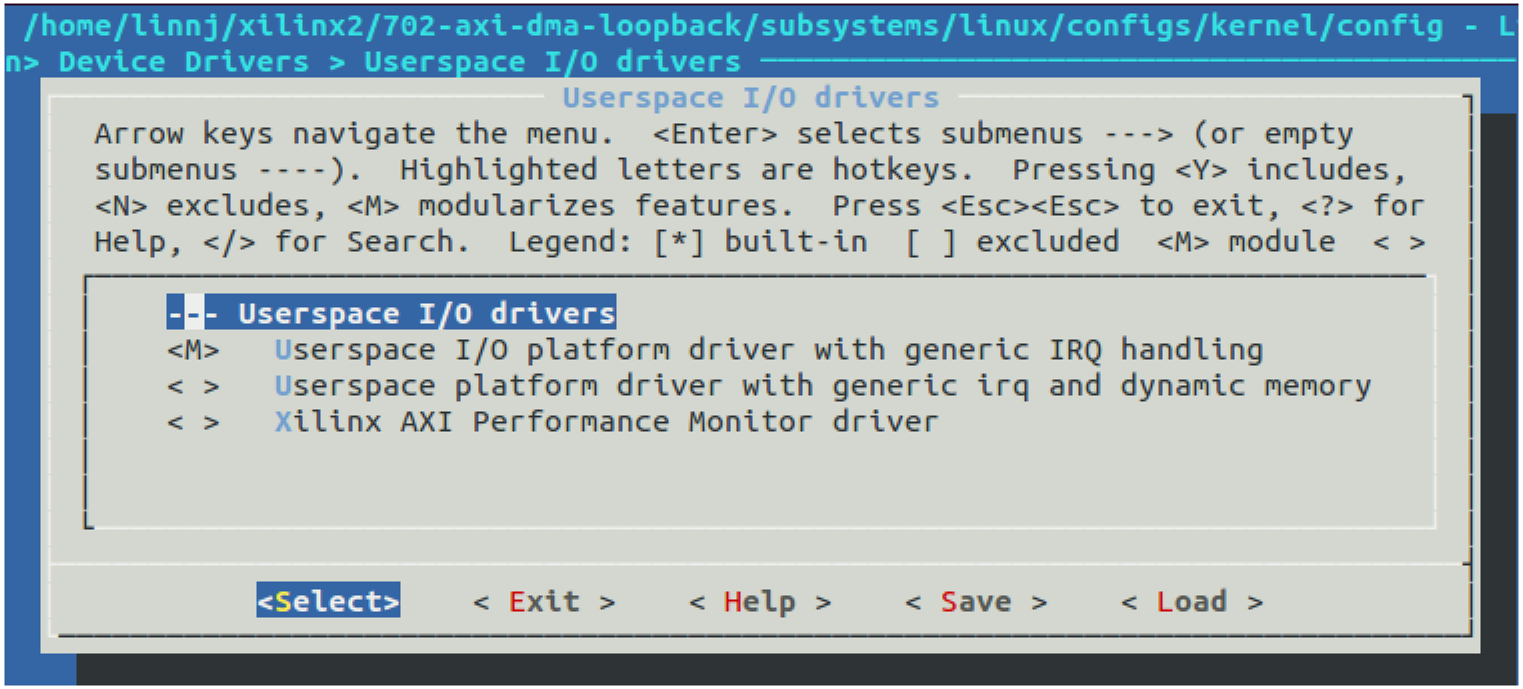
- **No access to the kernel frameworks and services**
  - Contiguous memory allocation, direct cache control, and DMA are not available
  - May have to duplicate kernel code or use a kernel driver to supplement
- **Interrupt handling cannot be done in user space**
  - It must be handled by a kernel driver which notifies user space causing some delay
- **There is no predefined API to allow applications to access the device driver**
  - Concurrency must also be considered if multiple applications access a driver

UIO Framework Features

- **There are two distinct UIO device drivers provided by Linux in drivers/uio**
- **UIO Driver (drivers/uio.c)**
  - For more advanced users as a minimal kernel space driver is required to setup the UIO framework
  - This is the most universal and likely to handle all situations since the kernel space driver can be very custom
  - The majority of work can be accomplished in the user space driver
- **UIO Platform Device Driver (drivers/uio_pdev_irqgen.c)**
  - This driver augments the UIO driver such that no kernel space driver is required
    - It provides the required kernel space driver for uio.c
  - It works with device tree making it easy to use
    - The device tree node for the device needs to use “generic-uio” in it’s compatible property
  - Best starting point since no kernel space code is needed
    - **It is the focus of this presentation**

UIO Driver Kernel Configuration

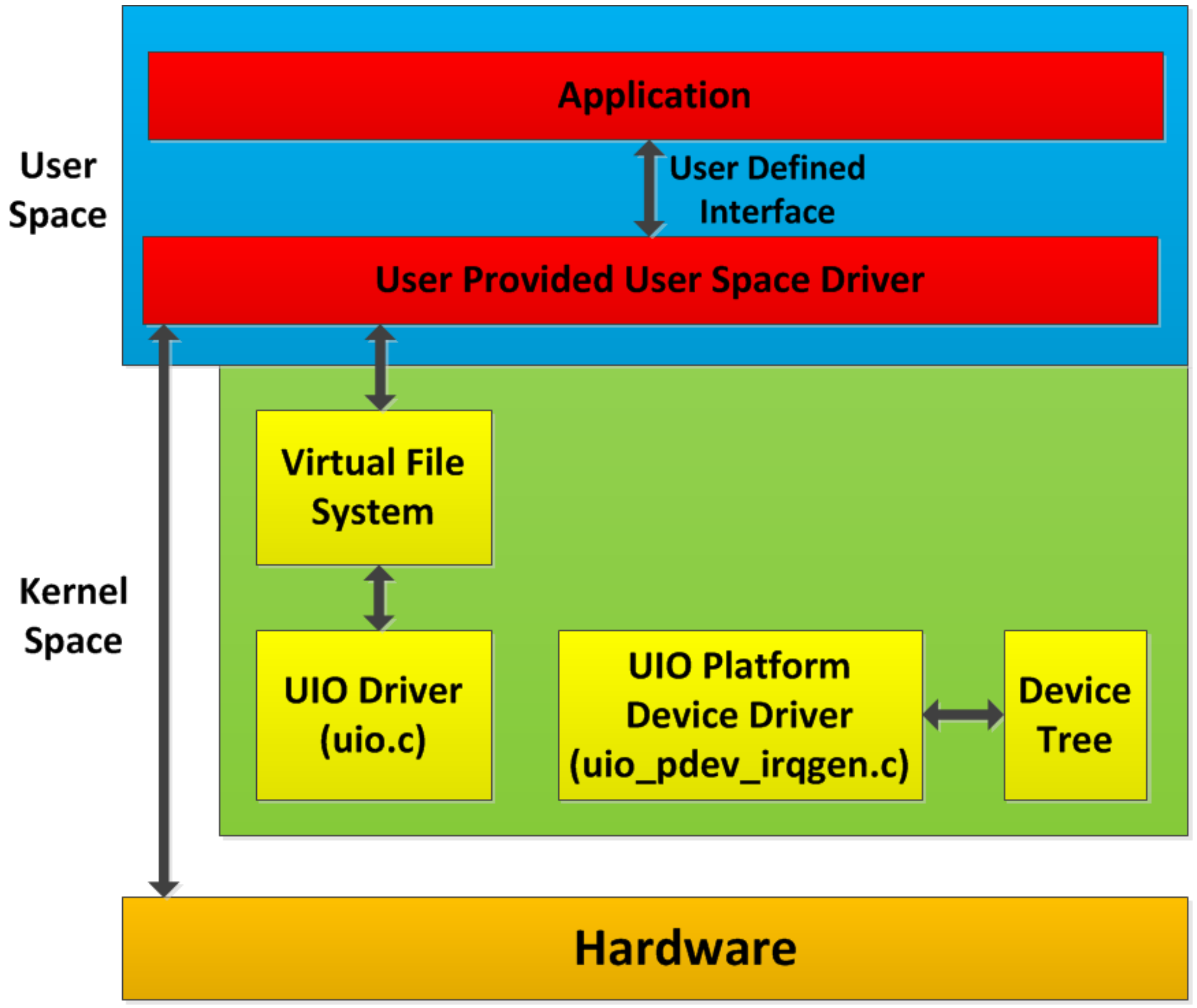
- **UIO drivers must be configured in the Linux kernel**
  - CONFIG_UIO=y
  - CONFIG_UIO_PDRV_GENIRQ=y



```
 /home/linnj/xilinx2/702-axi-dma-loopback/subsystems/linux/configs/kernel/config - L
n> Device Drivers > Userspace I/O drivers ─────────────────────────────────────────
  ┌─────────────────────── Userspace I/O drivers ───────────────────────┐
  │  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  │  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> includes,
  │  <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for
  │  Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >
  │ ┌────────────────────────────────────────────────────────────────────┐
  │ │      --- Userspace I/O drivers
  │ │      <M>   Userspace I/O platform driver with generic IRQ handling
  │ │      < >   Userspace platform driver with generic irq and dynamic memory
  │ │      < >   Xilinx AXI Performance Monitor driver
  │ │
  │ └────────────────────────────────────────────────────────────────────┘
  ├──────────────────────────────────────────────────────────────────────┤
  │         <Select>    < Exit >    < Help >    < Save >    < Load >
  └──────────────────────────────────────────────────────────────────────┘
```

- **The Xilinx kernel is configured to include UIO by default**

UIO Platform Device Driver Details



- The user provides only a user space driver
- The UIO platform device driver configures from the device tree and registers a UIO device
- The user space driver has direct access to the hardware
- The user space driver gets notified of an interrupt by reading the UIO device file descriptor

Kernel UIO API – Sys Filesystem

- **The UIO driver in the kernel creates file attributes in the sys filesystem describing the UIO device**
- **/sys/class/uio is the root directory for all the file attributes**
- **A separate numbered directory structure is created under /sys/class/uio for each UIO device**
  - First UIO device: /sys/class/uio/uio0
  - /sys/class/uio/uio0/name contains the name of the device which correlates to the name in the uio_info structure
  - /sys/class/uio/uio0/maps is a directory that has all the memory ranges for the device
  - Each numbered map directory has attributes to describe the device memory including the address, name, offset and size
    - /sys/class/uio/uio0/maps/map0

User Space Driver Flow

1. **The kernel space UIO device driver(s) must be loaded before the user space driver is started (if using modules)**
2. **The user space application is started and the UIO device file is opened (/dev/uioX where X is 0, 1, 2…)**
   – From user space, the UIO device is a device node in the file system just like any other device
3. **The device memory address information is found from the relevant sysfs directory, only the size is needed**
4. **The device memory is mapped into the process address space by calling the mmap() function of the UIO driver**
5. **The application accesses the device hardware to control the device**
6. **The device memory is unmapped by calling munmap()**
7. **The UIO device file is closed**

User Space Driver Example

No error checking

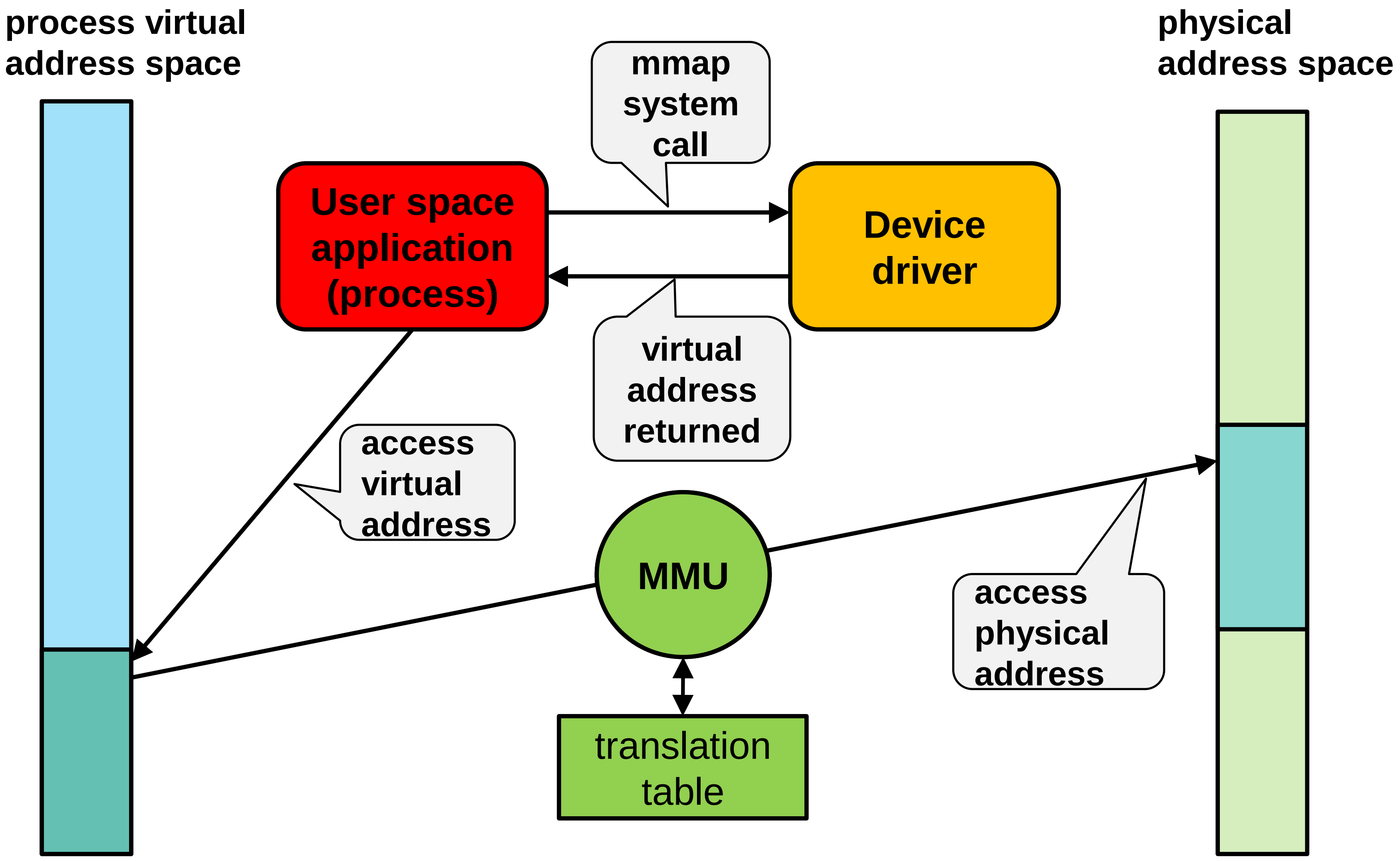
```
#define UIO_SIZE "/sys/class/uio/uio0/maps/map0/size"

int main( int argc, char **argv ) {
    int uio_fd;
    unsigned int uio_size;
    FILE *size_fp;
    void *base_address;

    uio_fd = open( “/dev/uio0”, O_RDWR);

    size_fp = fopen(UIO_SIZE, O_RDONLY);
    fscanf(size_fp, “0x%08X”,  &uio_size);

    base_address =  mmap(NULL, uio_size,
                PROT_READ | PROT_WRITE,
                MAP_SHARED, uio_fd, 0);

    // Access to the hardware can now occur….

    munmap(base_address, uio_size);
}
```

- **Open the UIO device so that it’s ready to use**
- **Get the size of the memory region from the size sysfs file attribute**
- **Map the device registers into the process address space so they are directly accessible**
- **Unmap the device registers to finish**

Mapping Device Memory Details

- **The character device driver framework of Linux provides the ability to map device memory into a user space process address space**
- **A character driver may implement the mmap() function which a user space application can call**
- **The mmap() function creates a new mapping in the virtual address space of the calling process**
  - A virtual address, corresponding to the physical address specified is returned
  - It can also be used to map a file into a memory space such that the contents of the file are accessed by memory reads and writes
- **Whenever the user space program reads or writes in the virtual address range it is accessing the device**
- **This provides improved performance as no system calls are required**

Mapping Device Memory Flow



User Space Application Interrupt Processing

- **Interrupts are never handled directly in user space**
- **The interrupt can be handled by the UIO kernel driver which then relays it on to user space via the UIO device file descriptor**
- **The user space driver that wants to be notified when interrupts occur calls *select()* *or* *read()* on the UIO device file descriptor**
  – The read can be done as blocking or non-blocking mode
- ***read()* returns the number of events (interrupts)**
- **A thread could be used to handle interrupts**
- **Alternatively a user provided kernel driver can handle the interrupt and then communicate data to the user space driver through other mechanisms like shared memory**
  – This may be necessary for devices which have very fast interrupts

User Space Application Interrupt Example

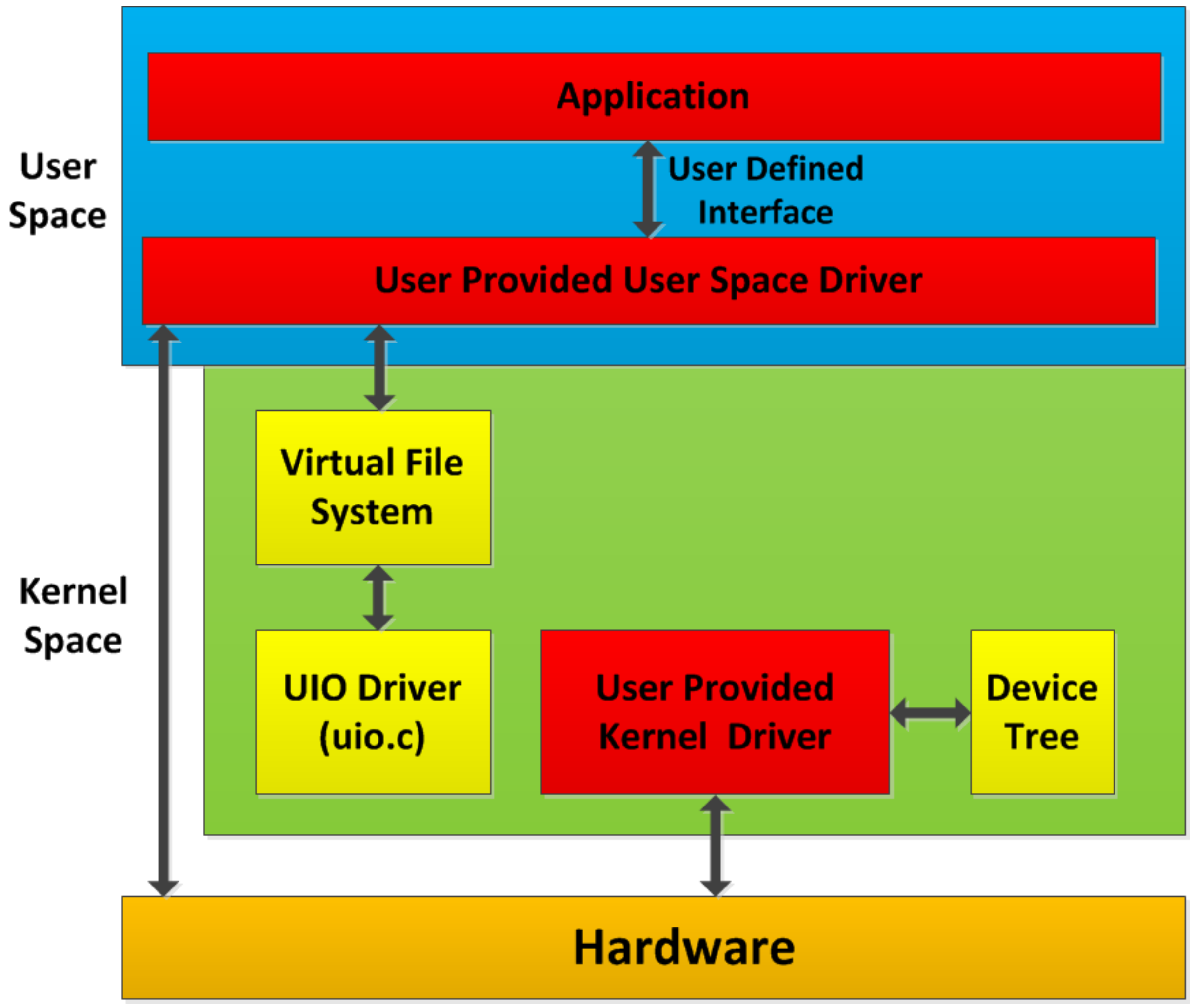
```
int pending = 0;
int reenable = 1;

int uio_fd = open("/dev/uio0", O_RDWR);

read(uio_fd, (void *)&pending, sizeof(int));

// add device specific processing like
// acking the interrupt in the device here

write(uio_fd, (void *)&reenable, sizeof(int));
```

No error checking

- **The UIO device is opened as previously described**
- **Read the UIO device file descriptor to wait for an interrupt, the read blocks by default, a non-blocking read can also be used**
- **The pending variable contains the number of interrupts that have occurred if multiple**
- **Re-enable the interrupt at the interrupt controller level**

Advanced UIO With Both User Space Application and Kernel Space Driver

UIO Driver Details



- **The user provides a kernel driver and a user space driver**
- **The kernel space driver is a platform driver configuring from the device tree and registering a UIO device**
- **The kernel space driver can also provide an interrupt handler in kernel space**
- **The user space driver has direct access to the hardware**

Kernel UIO API - Basics

- The API is small and simple to use
- struct uio_info
  - name: device name
  - version: device driver version
  - irq: interrupt number
  - irq_flags: flags for request_irq()
  - handler: driver irq handler (optional)
  - mem[ ]: memory regions that can be mapped to user space
    - addr: memory address
    - memtype: type of memory region (physical, logical, virtual)

Kernel UIO API - Registration

- **The function *uio_register_device()* connects the driver to the UIO framework**
  - Requires a struct uio_info as an input
  - Typically called from the probe() function of a platform device driver
  - Creates device file */dev/uio#* (#starting from 0) and all associated sysfs file attributes
- **The function uio_unregister_device() disconnects the driver from the UIO framework**
  - Typically called from the cleanup function of a platform device driver
  - Deletes the device file /dev/uio#

Kernel Space Driver Example

No error checking

```
probe() {
    dev = devm_kzalloc(&pdev->dev,
                (sizeof(struct uio_timer_dev)), GFP_KERNEL);
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    dev->regs = devm_ioremap_resource(&pdev->dev, res);
    irq = platform_get_irq(pdev, 0);
    dev->uio_info.name = "uio_timer";
    dev->uio_info.version = 1;
    dev->uio_info.priv = dev;
    dev->uio_info.mem[0].name = "registers";
    dev->uio_info.mem[0].addr = res->start;
    dev->uio_info.mem[0].size = resource_size(res);
    dev->uio_info.mem[0].memtype = UIO_MEM_PHYS;
    dev->uio_info.irq = irq;
    dev->uio_info.handler = uio_irq_handler;
    uio_register_device(&pdev->dev, &dev->info);
}
```

- **Platform device driver initialization in the driver probe() function**
- **Add basic UIO structure initialization**
- **Add the memory region initialization for the UIO**
- **Add the interrupt initialization for the UIO**
- **Register the UIO device with the kernel framework**

UIO Framework Details

- **UIO Driver**
  - The device tree node for the device can use whatever you want in the compatible property as it only has to match what is used in the kernel space driver as with any platform device driver
- **UIO Platform Device Driver**
  - The device tree node for the device needs to use “generic-uio” in it’s compatible property

References

- http://www.celinux.org/elc08_presentations/uio080417celfelc08.pdf
- http://www.osadl.org/fileadmin/dam/interface/docbook/howtos/uio-howto.pdf
- https://www.kernel.org/doc/htmldocs/uio-howto/
- http://lwn.net/Articles/267427/
- Petalinux Device Drivers, User Space IO, and Loadable Kernel Modules https://thesource.gosavo.com/Document/Document.aspx?id=29540098&view=&srlid=28271653&srisprm=False&sritidx=10&srpgidx=0&srpgsz=25
- Device Drivers For Hardware https://thesource.gosavo.com/Document/Document.aspx?id=29540101&view=&srlid=28271653&srisprm=False&sritidx=11&srpgidx=0&srpgsz=25