Userspace drivers in the new Linux kernel

DRIVER SHIFT

New versions of the Linux kernel will support a special userspace driver model, but some technical pitfalls might limit the use of this interesting new feature. **BY EVA-KATHARINA KUNST AND JÜRGEN QUADE**

For years, developers had little success convincing Linus Torvalds of the need for a programming interface for userspace drivers. A userspace driver needs to provide application program interfaces for hardware access, like any other driver; however, the userspace driver operates from the application layer, so it runs in non-privileged mode.

Now Linus has finally given up his resistance. The future kernel 2.6.23 will permit userspace drivers, and it will provide a kernel interface for them [1]. The code originated with Greg Kroah-Hartman and his Industrial IO interface [2].

According to microkernel proponents, simplifying the kernel by moving functions into user space promotes stability. At the same time, removing the driver from the kernel takes it out from under the GPL, which means the vendor doesn't have to reveal the driver's source code. Linux fans have been chatting in various forums about whether this new support for drivers in user space will motivate companies that have previously steered clear of GPLing their software to write binary drivers for their hardware and whether Linux is thus slowly but surely mutating into the kind of microkernel architecture that Linus has always rejected.

As you'll learn in this article, this new userspace driver model still requires some kernel code. The driver needs some kernel space code to link the physical device to the userspace component of the driver (see Figure 1). Additionally, the kernel creates pseudofiles on the sys filesystem, which the userspace part references to learn addresses for access.

In this article, we introduce the new Linux userspace driver model.

Userland to the Fore

The userspace side of the driver, which uses Mmap to bind hardware memory areas into its own address space for hardware access, can't read or write the corresponding registers until it has done so. If the hardware generates interrupts, the kernel part also needs to include an interrupt service routine (ISR).

Userland can make sure it is notified when an interrupt is thrown by blocking and reading from a device file (*/dev/*

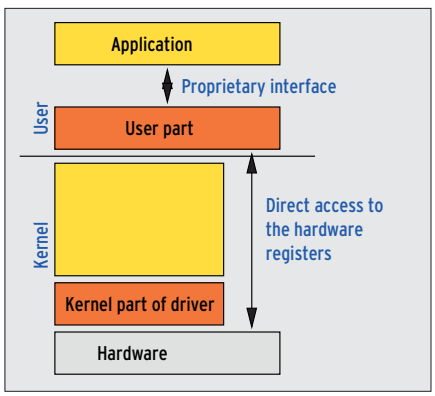

Figure 1: Even a userspace driver needs a kernel component.

uio0) that is created by the kernel. When an interrupt occurs, the process waiting in user space wakes up. The driver components then exchange data via the mapped hardware registers (addresses).

Thus, the userspace driver model just specifies how hardware resources are mapped in an application's memory space and how triggered events (interrupts) are handled – by blocking a read or calling *select()* against a device file. It is left to the driver developer to determine how applications access the device via a userspace driver (see Figure 1), and this is why kernel developers refer to the mechanism as User I/O (UIO).

The Kernel Part Juggles Three Objects

Listing 1 shows the kernel part of a minimal userspace driver without interrupt handling. All told, the code fills three data structures (objects) with content and passes them in to the kernel: The device model, the sys filesystem, or both need a driver (*struct device_driver*) and a device object (e.g., *struct platform_device*); the UIO subsystem needs an info object *uio_info* (Figure 2).

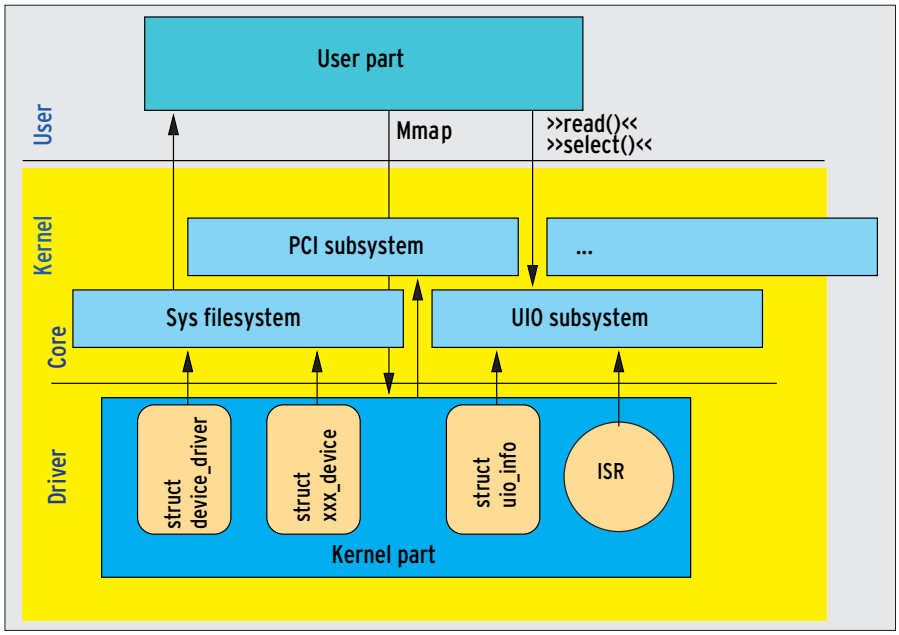

Figure 2: In the kernel part, the driver instantiates three objects and passes them to the kernel core.

If you implement the methods *probe()* and *remove()* for the sys filesystem driver object, you can identify the hardware addresses and enter them in the info object. In this case, you can instantiate a PCI device instead of the platform

Listing 1: kernel_part.c

```
01 #include <linux/module.h>
02 #include <linux/platform_
   device.h>
03 #include <linux/uio_driver.h>
04
05 struct uio_info kpart_info = {
06   .name = "kpart",
07   .version = "0.1",
08   .irq = UIO_IRQ_NONE,
09 };
10
11 static int drv_kpart_
   probe(struct device *dev);
12 static int drv_kpart_
   remove(struct device *dev);
13 static struct device_driver
   uio_dummy_driver = {
14   .name   = "kpart",
15   .bus    = &platform_bus_type,
16   .probe  = drv_kpart_probe,
17   .remove = drv_kpart_remove,
18 };
19
20 static int drv_kpart_
   probe(struct device *dev)
21 {
22   printk("drv_kpart_probe( %p
   )\n", dev );
23   kpart_info.mem[0].addr =
   (unsigned long)kmalloc(512,
   GFP_KERNEL);
24   if( kpart_info.mem[0].
   addr==0 )
25       return -ENOMEM;
26   kpart_info.mem[0].memtype =
   UIO_MEM_LOGICAL;
27   kpart_info.mem[0].size =
   512;
28   if( uio_register_
   device(dev,&kpart_info) )
29       return -ENODEV;
30   return 0;
31 }
32
33 static int drv_kpart_
   remove(struct device *dev)
34 {
35   uio_unregister_
   device(&kpart_info);
36   return 0;
37 }
38
39 static struct platform_device
   *uio_dummy_device;
40 static int __init uio_kpart_
   init(void)
41 {
42   uio_dummy_device = platform_
   device_register_
   simple("kpart", -1,
43
                        NULL, 0);
44   return driver_register(&uio_
   dummy_driver);
45 }
46
47 static void __exit uio_kpart_
   exit(void)
48 {
49   platform_device_
   unregister(uio_dummy_device);
50   driver_unregister(&uio_
   dummy_driver);
51 }
52
53 module_init( uio_kpart_init );
54 module_exit( uio_kpart_exit );
55
56 MODULE_LICENSE("GPL");
```

device. A call to the *uio_register_device()* function finally hands the initialized info object over to the UIO subsystem.

Performance Matters

In the scope of registration, the UIO subsystem checks to see whether the device model contains the *uio* device class. If not, it creates the class. At the same time, the UIO subsystem ensures a major number is assigned to the module. After this, the subsystem reserves a minor number to the driver; udev creates the matching device file, such as */dev/uio0*.

Finally, the routine creates the required attribute files and, if the driver developer implements them, adds the ISR. To write this kernel code, the programmer needs a good working knowledge of the kernel, and especially of device model handling, the sys filesystem, and the implementation of ISRs.

For each driver that uses *uio_register_device()* to register with the UIO subsystem, the subsystem creates a device file with the pattern */dev/uio0*, */dev/uio1*, */dev/uio2*, and so on. Then the user side of the driver must discover which of the listed device files the hardware is hiding behind. To do so, it can use the pseudo-files in the sys filesystem (Figure 3).

The *name* file, which contains what is hopefully the unique driver name, can be read from the path */sys/class/uio/uioX* (where *X* is replaced by *0*, *1*, and so on). The user part finds the address information stored by the kernel part in the relevant directory. The user part then calls *mmap()* to bind the addresses into its own address space. Opening a device file – */dev/uio0*, for example – gives you the file descriptor required for this. If *mmap()* is successful, the user part (see

Listing 2: user_part.c

```
01 #include <stdio.h>
02 #include <fcntl.h>
03 #include <stdlib.h>
04 #include <unistd.h>
05 #include <sys/mman.h>
06
07 #define UIO_DEV "/dev/uio0"
08 #define UIO_ADDR "/sys/class/uio/uio0/maps/map0/addr"
09 #define UIO_SIZE "/sys/class/uio/uio0/maps/map0/size"
10
11 static char uio_addr_buf[16], uio_size_buf[16];
12
13 int main( int argc, char **argv )
14 {
15   int uio_fd, addr_fd, size_fd;
16   int uio_size;
17   void *uio_addr, *access_address;
18
19   addr_fd = open( UIO_ADDR, O_RDONLY );
20   size_fd = open( UIO_SIZE, O_RDONLY );
21   uio_fd  = open( UIO_DEV, O_RDONLY );
22   if( addr_fd<0 || size_fd<0 || uio_fd<0 ) {
23       fprintf(stderr,"Cannot open UIO files...\n");
24       return -1;
25   }
26
27   read( addr_fd, uio_addr_buf, sizeof(uio_addr_buf) );
28   read( size_fd, uio_size_buf, sizeof(uio_size_buf) );
29   uio_addr = (void *)strtoul( uio_addr_buf, NULL, 0 );
30   uio_size = (int)strtol( uio_size_buf, NULL, 0 );
31
32   access_address = mmap(NULL, uio_size,
33   PROT_READ, MAP_SHARED, uio_fd, 0);
34   printf("The HW address %p (length %d) "
35          "can be accessed over"
36          " logical address %p\n", uio_addr,
37          uio_size, access_address);
38
39   // Access to the hardware registers can occur from here on ...
40   // ...
41   return 0;
42 }
```

Pros and Cons of the Userspace Driver Model

Advantages:

- Version change: The user only needs to rebuild the kernel part after making any required modifications.
- Stability: Protects the kernel against buggy drivers.
- Floating point is available.
- Efficient, because processes do not need to be swapped.
- License: The user part of the source code does not need to be published (although this is a controversial subject in the context of the GPL).

Disadvantages:

- Kernel know-how is required for standard drivers, the sys filesystem, IRQ, and PCI.
- Timing is less precise than in kernel space.
- Response to interrupts: Response times are longer than in kernel space (process change).
- Functionality is severely restricted in userland; for example, DMA is not available.
- API: The application can't use a defined interface to access the device.
- Restricted security is sometimes difficult to achieve.

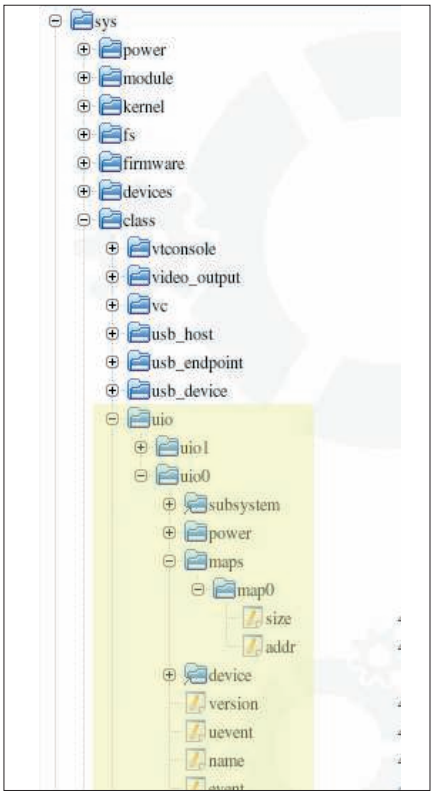

Figure 3: As soon as the kernel part of the userspace driver has been loaded, UIO creates information on the sys filesystem. The user part needs to retrieve and evaluate the information to access the hardware.

sample code in Listing 2) can now access the hardware register.

A routine that needs to be notified when interrupts occur calls *select()* or *read()* in non-blocking mode. Incidentally, *read()* returns the number of events (interrupts) that occur as a 4-byte value. Just to be sure, the mechanism only works if you request exactly 4 bytes from the kernel as an *s32* data type! Access via the *cat* command, for example, which always requests 4096 bytes, will fail.

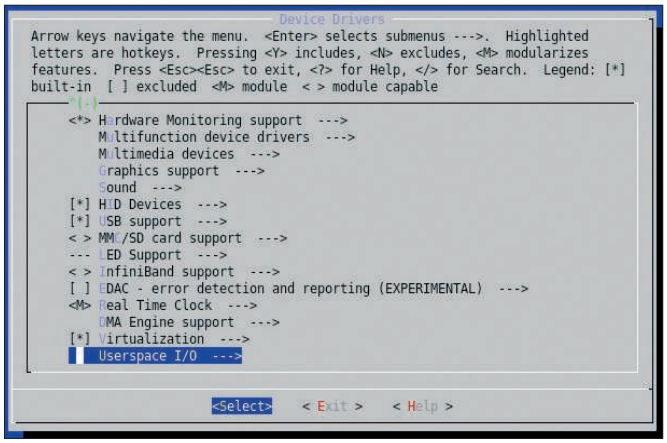

Figure 4: Userspace drivers scheduled for kernel 2.6.23 are initially disabled in the kernel configuration.

Quick and Dirty ISR

The ISR you need to program for the kernel to handle this can be fairly lean. All it has to do is ascertain whether its own hardware has triggered an interrupt and, if so, acknowledge the interrupt on the hardware side and return *IRQ_HANDLED*. If you want to use a kernel-based approach to wake up the sleeping user part, you can call the *uio_event_notify()* function.

Do It Yourself

If you are interested in trying out the new interface, you will need a very recent kernel, for example 2.6.23-rc3. Also, you will need to enable the UIO subsystem below *Device Drivers* | *Userspace I/O* | *Userspace I/O Drivers* in the kernel configuration (see Figure 4). If UIO is built as a module in your kernel, root can load this by running *modprobe uio*.

After this, you can generate the userspace driver – that is, both the kernel module and the userspace part. A makefile is shown in Listing 3. To keep the example simple, it neither binds PCI hardware nor uses any interrupts. For more information on this, please refer to "The Userspace I/O HOWTO" DocBook.

Conclusions

Although Linus Torvalds has now added the UIO subsystem to the current kernel, this change can't be seen as a major paradigm shift with respect to userspace drivers. The new driver model offers a standard structure for developing drivers in userspace, but the 1000 lines of code of the UIO subsystem don't really give programmers more than Mmap and the normal driver interfaces already gave them. Also, the new model provides no mechanisms for binding hardware to the existing kernel subsystems, such as network or block device drivers.

Restricted functionality and the lack of a defined interface between the application and the userspace driver raise barriers against userspace drivers becoming popular outside of embedded applications. ■

Listing 3: Makefile for Kernel and User Part

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m := kernel_part.o
03
04 else
05 KDIR  := /lib/modules/$(shell uname -r)/build
06 PWD   := $(shell pwd)
07
08 default:
09   $(MAKE)   -C $(KDIR)   M=$(PWD) modules
10 endif
11
12 clean:
13   rm -f *.ko *.o user_part
14
15 user_part: user_part.c
16   cc -g -Wall    user_part.c   -o user_part
```

INFO

[1] Linus Torvalds' announcement on kernel 2.6.23-rc1: *http://article.gmane.org/gmane.linux.kernel/559066*

[2] Simple userspace interface for PCI drivers: *http://www.uwsg.indiana.edu/hypermail/linux/kernel/0608.3/1908.html*

[3] Complete listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/86*