Writing Character Device Driver for Linux

Version: 1.0

This Document is intended to be both a tutorial for beginners and a reference guide for advanced Developers.

Contents

```
Right, now that that's over with, let's get into the fun stuff!
```

First few words to this Paper

Some time ago i played around with some Data aquisition cards so i needed more insights to the whole Kernel stuff. Then i found the very good Document written by Robert Baruch (last year) that took me at the Hand and deeper and deeper in the dark Forest of Kernel hacking. Then sometimes later i found the kernel hackers guide at tsx-11 that gave me some details about memory and so on, and i got some more insights. In this time i've been asked more and more by people that are interested in data aquisition and process control. So i got the idea to collect everything related to this stuff and make it availiable for others.

This Document is a *collection* of papers about Linux Drivers i included Robert's tutorial because i can't make it better (i hope Robert won't flame up about this). I added some Parts from Kernel hackers guide (examples). My Part consists of some ideas to the Linux-Hardware interconnection and new concepts that are availiable since v1.0 (Modules etc). My intention was to help both the people that want to learn how a driver can be written and those that want to know special things about drivers.

I hope you will have fun with this.

clausi@chemie.fu-berlin.de

Chapter 1

Introduction

Some words of thanks (Yes i copied this):

... First Robert Baruch

Many thanks to:

Donald J. Becker (becker@metropolis.super.org)
Don Holzworth (donh@gcx1.ssd.csd.harris.com)
Michael Johnson (johnsonm@stolaf.edu)
Karl Heinz Kremer (khk@raster.kodak.com)
Hennus Bergman (csg279@wing.rug.nl)
Jon. Tombs (??)
All the driver writers!

...and of course, Linus "Linux" Torvalds and all the guys who helped develop Linux into a BLOODY KICKIN' O/S!

...and now a word of warning:

Messing about with drivers is messing with the kernel. Drivers are run at the kernel level, and as such are not subject to scheduling. Further, drivers have access to various kernel structures. Before you actually write a driver, be *damned* sure of what you are doing, lest you end up having to re-format your harddrive and re-install Linux!

The information in this Guide is as up-to-date as I could make it. It also has no stamp of approval whatsoever by any of the designers of the kernel. I am not responsible for damage caused to anything as a result of using this Guide.

Chapter 2

General Concepts

2.1 What is an Driver ?

Imagine the following Problem: You buyed a card XXX that does some I/O to periphal devices like printers, plotters, analog devices etc. Normaly you have to write lot of assembler code for programming the registers of this card that is linked to your program code. But every Program that uses this piece of code has to be linked with it. If you change anything of one of your programs you have to rebuild all your programs to be up to date.

The most elegant way to avoid this is to extract the hardware-specific code from your program and make it to an part of your operating system. The specific code then is called via well defined interface routines that look very similar for each problem. With this method the specific code is invisible to the user program.

Remember that the most operating systems uses this concept for their system-calls: The arguments of an call are put on the stack and then a "trap" routine is called. The operating system jumps to an so called "trap-handler" that takes the arguments from the stack and does something with it.

Linux and other *ix-like operating systems make a distinction between system calls and calls that are used for the hardware programming of periphal devices. The routines for this periphal devices are collected in "Driver-Modules" that are visible to the kernel only by a few interface routines. The only problem at this stage is to communicate between the User Program and these routines.

*ix uses the filesystem for this purpose. The Driver looks like a normal File that you can open, close, read from and write to. The kernel sees this operations as special requests and maps it to the appropriate calls in the driver code.

2.2 The Driver and the Filesystem

As mentioned from the user's side of view the driver looks like an ordinary file. If you operate on this file via open, close, read or write requests the kernel looks up the apropriate function in your driver code. But how this path of function references can be found?

All drivers provides a set of routines. Each Driver has an struct char_fops that holds the pointers to this routines. At init time of the driver this struct is hooked into another table where the kernel can find it. The index that is used to dereference this set of routines is called MAJOR number and is unique so that a definite destinction its possible. The special inode file for the driver gets its MAJOR number at creation time via the `mknod` command. The driver code gets its MAJOR by the register_chrdev kernel routine (Your task is to find an unique MAJOR).

Every time a file routine is called on a driver-inode the inode and the file struct is passed to this routine (See. A.2), the kernel dereferences the apropriate routine from the char_fops struct by its MAJOR number and calls it. The inode and the file struct itself are passed to the called fops-routines and can be used to get specific information of the caller-process for Example. The MINOR number, that is also set by the `mknod` command, can be used to configure a special behavior of the driver (e.g rewind or norewind on tapes) or to distinct subdevices (as it is done by the tty drivers).

A list of used MAJORS can be found in $LINUX_SOURCE/include/linux/major.h:

```
/*
 * assignments
 *
 * devices are as follows (same as minix, so we can use the minix fs):
 *
 *      character              block                    comments
```

```
*      --------------------   --------------------   --------------------
*  0 - unnamed                unnamed                minor 0 = true nodev
*  1 - /dev/mem               ramdisk
*  2 -                        floppy
*  3 -                        hd
*  4 - /dev/tty*
*  5 - /dev/tty; /dev/cua*
*  6 - lp
*  7 -                                               UNUSED
*  8 -                        scsi disk
*  9 - scsi tape
* 10 - mice
* 11 -                        scsi cdrom
* 12 - qic02 tape
* 13 -                        xt disk
* 14 - sound card
* 15 -                        cdu31a cdrom
* 16 - sockets
* 17 - af_unix
* 18 - af_inet
* 19 -                                               UNUSED
* 20 -                                               UNUSED
* 21 - scsi generic
* 22 -                        (at2disk)
* 23 -                        mitsumi cdrom
* 24 -                        sony535 cdrom
* 25 -                        matsushita cdrom       minors 0..3
* 26 -
* 27 - qic117 tape
* 28 -
* 29 -
* 30 -
* 31 -
*/
```

The MAJOR numbers go from 0 to MAX_CHRDEV-1. MAX_CHRDEV is defined in linux/fs.h, and is currently set at 32. In general, you want to stay away from devices 0-15 because those are reserved for the "usual" devices.

2.3 The general look of a Driver

The Structure of an driver is (as mentioned) similar for each periphal device.

- You have an init routine that is for initializing your hardware, perhaps getting memory from the kernel and last but not least hooking your driver-routines into the kernel's gearbox[1].
- You have an char_fops struct that is initialized with those routines that you will provide for your driver. This struct is the key to the kernel it is 'registered' by the register_chrdev routine.
- Mostly you have open and release routines that are called whenever you perform a open or close on your special inode. The presence of this routines is not necessary.
- You can have routines for reading and writing data from or to your driver, a ioctl routine that can perform special commands to your driver like config requests or options.
- You have the possibility to readout the kernel environment string to configure your driver via lilo, but on this later on.
- A Interrupt routine can be registered if your hardware support this.

2.4 Compile your Driver into Kernel Code

The most drivers in Linux are linked to the kernel at compile-time. That means if you want to add an driver you have to put your .c and .h files directly somewhere in the kernel source path and rebuild the kernel.

For character device driver this should be done in $LINUX_SOURCE/drivers/char . Edit the Makefile and add

[1]Realize that this routine is called once at boot time, just before the filesystem and your hard disks are initialized so you've no chance to read any config files at startup.

```
ifdef CONFIG_MYDRIVER
OBJS := $(OBJS) my_driver.o
SRCS := $(SRCS) my_driver.c
endif
```

after the OBJS and SRCS definition.

Now your driver will be compiled into the kernel whenever CONFIG_MYDRIVER is defined at build time. To define CONFIG_MYDRIVER in Linux manner do this:

cd to $LINUX_SOURCE/linux and edit the file config.in, add somewhere in the driver section of the file a line like:

```
bool 'My Driver Support' CONFIG_MYDRIVER n
```

To get your driver running you have to hook your init routine into the the kernel so that your init is called at boot time. You have to do this in $LINUX_SOURCE/drivers/char/mem.c . Go to the chr_drv_init routine and add your own init routine:

```
long chr_dev_init(long mem_start, long mem_end)
{
if (register_chrdev(MEM_MAJOR,"mem",&memory_fops))
printk("unable to get major %d for memory devs\n", MEM_MAJOR);
mem_start = tty_init(mem_start);
#ifdef CONFIG_PRINTER
mem_start = lp_init(mem_start);
#endif
#if defined (CONFIG_BUSMOUSE) || defined (CONFIG_82C710_MOUSE) || \
    defined (CONFIG_PSMOUSE) || defined (CONFIG_MS_BUSMOUSE) || \
    defined (CONFIG_ATIXL_BUSMOUSE)
mem_start = mouse_init(mem_start);
#endif
#ifdef CONFIG_SOUND
mem_start = soundcard_init(mem_start);
#endif
#ifdef CONFIG_PCSP
mem_start = pcsp_init(mem_start);
```

```
#endif
#if CONFIG_TAPE_QIC02
mem_start = tape_qic02_init(mem_start);
#endif
/******************--------->>>>>>  here do it
#if CONFIG_MYDRIVER
mem_start = my_driver_init(mem_start);
#endif
******************<<<<<---------   done  */
/*
 *      Rude way to allocate kernel memory buffer for tape device
 */
#ifdef CONFIG_FTAPE
        /* allocate NR_FTAPE_BUFFERS 32Kb buffers at aligned address */
        ftape_big_buffer= (char*) ((mem_start + 0x7fff) & ~0x7fff);
        printk( "ftape: allocated %d buffers alligned at: %p\n",
               NR_FTAPE_BUFFERS, ftape_big_buffer);
        mem_start = (long) ftape_big_buffer + NR_FTAPE_BUFFERS * 0x8000;
#endif
return mem_start;
}
```

Now cd to $LINUX_SOURCE/linux. Type `make config` at $LINUX_SOURCE/linux, you will be asked about your driver support, answer with 'y'. Now rebuild your kernel as usual and install it with lilo.

It is strongly recommended that you make an linux-test entry in lilo so that you can reboot even if one of your drivers fails.

For example add to your /etc/lilo.conf

```
  root = /dev/hda1
  image = /usr/src/linux/zImage
  label = test
```

The other possibility is to make a boot disk via `make disk`

For details refer to the lilo manual.

2.5 Dynamically loaded Drivers

Since Linux Version 0.99.15 Support for Kernel Modules has been added by Jon. Tombs. This Kernel modules can be loaded into the Kernel at runtime. That means loaded and removed at any time after the boot process. The only differences between a loadable Module and a Kernel linked Driver are a special init() routine that is called when the module is loaded into the Kernel and a cleanup routine that is called when the Module is removed.

The typical purpose of this two routines is to register the device and irq or get memory (init_module) and release/free it if the module is removed from memory (cleanup_module).

One disadvantage of this Method is that such Kernel Modules can't allocate a continuous range of kernel Memory that is greater than 4096 bytes.

The programs for loading and removing Modules are present in the modutils package available from tsx-11 or sunsite.

Details on writing such drivers are discussed later.

Chapter 3

Writing a driver

3.1 General

The Next few sections (that come from Robert Baruch's tutorial) will bring you up to understand how a Driver works together with the Kernel and how an user program does something with the driver. Realize – that is not the whole story. Mostly all the Hardware specific stuff is the largest part of work.

The Kernel has no ability to handle with runtime errors as a user program has, so be careful whatever you do. If you do driver development do is slowly and with care. Here are some tips that serves from the worst case:

- Never trust the Users. That means whenever you get data from the User program, check it for validity. Especially check for validity of Pointers.

  ```
  if( data_from_user == NULL )
  printk("Driver: data_from_user invalid ");
  return(-EINVAL);
  else
     if( data_from_user->flags & BUSY ).......
  ```

- Never trust yourself. Nobody is perfect. Test out your Driver step by step.

- Hold the code as clearly as possible. That will prevent own errors. A sample for quick & dirty coding:

```
int driver_does_something_on_hardware(){
outb(0x11,0x330);
outb(0x23,0x333);
return(inb(0x330));
}
```

A reader of this has to guess what is going on with this code. You should doing yourself and others a favour and clear up the code.

```
/* Registers */

#define CARD_XXX_BASE 0x330
#define CARD_XXX_MODE CARD_XXX_BASE
#define CARD_XXX_DATA CARD_XXX_BASE+1
#define CARD_XXX_CMD  CARD_XXX_BASE+2

/* Modes */

#define MODE_1 0x01
#define MODE_2 0x10

/* Commands */

#define COMMAND_1 0x23

int driver_does_something_on_hardware(){
        outb( MODE_1 | MODE_2 ,CARD_XXX_MODE); /* select mode */
        outb( COMMAND_1 ,CARD_XXX_CMD );       /* do command */
        return(inb(CARD_XXX_MODE));            /* read data */
}
```

That means better to spend some more lines of text for clearness.With this example you have the possibility to change the adress-space of your card by changing one Macro (and perhaps one jumper on the card). That shows that the work for the few lines more will be payed with a better maintainance of your code.

- If something is going wrnog with your driver try out to find the error. Do some

  ```
  #ifdef MY_DRIVER_DEBUG
        printk("My Driver: The state of the art ist %d",a_variable);
  #endif
  ```

  statements in your code that will help to find the error and to watch what the driver does.

3.2 First Steps

Now lets do it. First let's have a look of our first Example. A very stupid one, but a very good exercise to get your stuff running.

```
========================================
      File Listing 1: testdata.c
========================================
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/signal.h>
#include <linux/errno.h>

#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>
#include <asm/irq.h>

unsigned long test_init(unsigned long kmem_start)
{
  printk("Test Data Generator installed.\n");
  return kmem_start;
}
```

Now go to $LINUX_SOURCE/drivers/char/mem.c and hook your driver in place as discussed above. Change the Makefile and config.in and rebuild.

At reboot you should see your message.

3.3 A more useful Driver

This example is taken from the *Writing UNIX Device Drivers* book by George Pajari, published by Addison Wesley. It can usually be found in a Barnes and Noble bookstore, or any large bookstore which has a nice section on UNIX. The ISBN is 0-201-52374-4, and it was published in 1992. This book is highly recommended for the device driver writer.

This device driver will actually be read from. You can open and close it (which really won't do much), but the biggest thing it will do is allow you to read from it. This driver won't access any external hardware, and so it is called a "pseudo device driver". That is, it really doesn't drive any device.

Have your Guide handy? OK, now alter your testdata.c file so that it looks like this:

```
========================================
      File Listing 2: testdata.c
========================================
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/signal.h>
#include <linux/errno.h>

#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>
#include <asm/irq.h>

static char test_data[]="Linux is really funky!\n";

static int test_read(struct inode * inode, struct file * file,
                     char * buffer, int count)
{
  int offset;
```

```
  printk("Test Data Generator, reading %d bytes\n",count);
  if (count<=0) return -EINVAL;
  for (offset=0; offset<count; offset++)
    put_fs_byte(test_data[offset % (sizeof(test_data)-1)], buffer+offset);
  return offset;
}

static int test_open(struct inode *inode, struct file *file)
{
  printk("Test Data Generator opened.\n");
  return 0;
}

static void test_release(struct inode *inode, struct file *file)
{
  printk("Test Data Generator released.\n");
}

struct file_operations test_fops = {
NULL, /* test_seek */
test_read, /* test_read */
NULL, /* test_write */
NULL,  /* test_readdir */
NULL,  /* test_select */
NULL,  /* test_ioctl */
NULL, /* test_mmap */
test_open, /* test_open */
test_release /* test_release */
};

unsigned long test_init(unsigned long kmem_start)
{
  printk("Test Data Generator installed.\n");
  if (register_chrdev(21,"test",&test_fops));
    printk("Test Data Generator error: Cannot register to major device 21!\n");
  return kmem_start;
}
```

This Example demonstrates the use of the register_chrdev routine and the use of the char_fops struct that tells the kernel which function to call for which kernel operation.

If a driver has already taken major 21, register_chrdrv will return -EBUSY. Here, all we do is print a message saying that 21 is already taken.

Now, the test_open and test_release functions just print out things to the console. They are really there for debugging purposes, so that you can see when things happen.

The meat of the driver is the test_read function. The first thing it does is print out how many bytes were requested. Then it puts that many bytes into user space. Remember that the driver is executing at the kernel level, and the user space will be differnet from kernel space. We have to do some kind of translation to put the data which is in kernel space into the buffer which is in user space. We use here the put_fs_byte function.

The loop puts the string into the buffer, going back to the beginning of the string if necessary. Once the loop is finished, we just return the actual number of bytes read. The actual number may be different from the requested number. For example, you may be reading from the driver some kind of message which has a fixed size. You may want to code the driver so that if you attempt to read more than the message size, you will get only the message size, and no more. Here, we just give the process however many bytes it wants.

Now, let's get this driver into the kernel. But first what we'll do is create a special file which can be opened, read, and closed. Operations on this special file will activate your driver code.

The special files are normally stored in the /dev directory. Do this:

```
mknod /dev/testdata c 21 0
chmod 0666 /dev/testdata
```

This makes a special character (c) file called testdata, and gives it major 21, minor 0. The chmod makes sure that everyone can read and write the device.

Now recompile the kernel, and reboot. Once again, make sure you fix any warnings or errors in your testdata.c compilation.

Now, go to the /tmp directory (or whereever you want), and write this program:

```
=========================================
      File Listing 3: data.c
=========================================
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

void main(void)
{
  int fd;
  char buff[128];

  fd = open("/dev/testdata",O_RDWR);
  printf("/dev/testdata opened, fd=%d\n",fd);
  if (fd<=0) exit(0);
  printf("sizeof(buff)=%d\n",sizeof(buff));
  printf("Read returns %d\n",read(fd,buff,sizeof(buff)));
  buff[127]=0;
  printf("buff=\n'%s'\n",buff);
  close(fd);
}
```

Compile it using gcc. Run it. If it said "Linux is really funky!" lots of times, pat yourself on the back (or whereever you want) for a job well done. If it didn't, check the output, and see where you went wrong. It could just be that you have a bad or old kernel.

The last line may be partial, since you're only printing out 127 characters.

Experiment 1:

Use mknod to make another special file, this one with minor 1. Call it something like /dev/testdata2. Change the device driver so that in the read call, it finds out which minor is being read from. Use this:

```
  int minor = MINOR(inode->i_rdev);
```

Print out the minor number, and depending on which minor it is, read from a different message string. Test your driver with code similar to data.c.

3.4 Using Memory

You've learned to read, now you're gonna learn to write.

Now that you're reading strings, you may want to write strings and read them back. We'll go through two versions of this – one that uses static memory, and one that dynamically allocates the memory.

Keeping your current driver, all you need to do is add a write function to it, not forgetting to put that write function into the file_operations structure of the driver.

Add this section of code to your driver above the file_operations structure declaration:

```
========================================
  File Listing 4 (partial): testdata.c
========================================
static char test_data[128]="\0";
static int test_data_size=0;

static int test_write(struct inode * inode, struct file * file,
  char * buffer, int count)
{
  printk("Write %d bytes\n",count);
  if (count>127) return -ENOMEM;
  if ((!test_data_size) || (count<=0)) return -EINVAL;
  memcpy_fromfs((void *)test_data, (void *)buffer, (unsigned long)count);
  test_data[127]=0;  /* NUL-terminate the string if necessary */
  test_data_size = count;
  return count;
}
```

Also, alter the test_read function so that instead of using sizeof(test_data) as the size of the test_data string, it uses test_data_size.

In the test_write function, I have decided to prevent the acceptance of strings which are too big to fit (with a NUL-terminator) into the test_data area, rather than just writing only what fits. In this case, if the offered string is too long, I return ENOMEM. The write function in the user's process will return ¡0, and errno will be set to ENOMEM.

Also note that I have used the memcpy_fromfs function, which is real convenient – much more convenient than looping a put_fs_byte.

Compile this driver, and test it by modifying data.c to write some data, then read it back.

Experiment 2:

Re-write the driver so that it can have two different strings for the two minor devices as in experiment 1.

Now that we can write data to the driver, it would be nice if we could dynamically allocate memory to store a string in. We will use kmalloc to do this. (Why is discussed later)

One thing which must be realized with kmalloc – it can only allocate a maximum of one Linux page (4096 bytes). If you want more, you will have to create a linked list.

Change your driver so that instead of listing 4, you have this:

```
=========================================
  File Listing 5 (partial): testdata.c
=========================================
static char *test_data=NULL;
static int test_data_size=0;

static int test_write(struct inode * inode, struct file * file,
  char * buffer, int count)
{
  printk("Write %d bytes\n",count);
  if (count>4095) return -ENOMEM;
  if (test_data!=NULL) kfree_s((void *)test_data, test_data_size);
  test_data_size = 0;
  test_data = (char *)kmalloc((unsigned int)count, GFP_KERNEL);
  if (test_data==NULL) return -ENOMEM;
```

```
  memcpy_fromfs((void *)test_data, (void *)buffer, (unsigned long)count);
  test_data[count]=0;  /* NUL-terminate the string if necessary */
  test_data_size = count;
  return count;
}
```

Here, instead of statically allocating memory for the string, we dynamically allocate it using kmalloc. Note first, that if we had already allocated a string, we free it first by using kfree_s. This is faster than using kfree, because kfree would have to search for the size of the object allocated. Here we know what the size was, so we can use kfree_s. kmalloc vs. malloc is discussed below.

Next, note that we use the GFP_KERNEL priority in the kmalloc. This causes the process to go to sleep if there is no memory available, and the process will wake up again when there is memory to spare. In general, the process will sleep until a page of memory is swapped out to disk.

In the event of catastrophic memory non-availability, kmalloc will return NULL, and we should handle that case. Unfortunately here, we have already freed the previous string – although that could be changed easily by kmallocing, then kfreeing.

The rest of the code reads as in listing 4.

When we get into the section on interrupt handling, we will discuss the use of GFP_ATOMIC as a kmalloc priority.

A brief excursion into kmalloc vs. malloc:

The malloc() call allocates memory in user space, which is fine if that's what you want. Here, we want to have the driver store information so that *any* process can use it, and so we have to allocate memory in the kernel. That means, kmalloc(). Further, there is a maximum of 4096 bytes which can be allocated in any one call of kmalloc. This means that you cannot be guaranteed to get contiguous space of over 4096 bytes. You will have to use a linked list of kmalloced buffers.

Alternatively, you can fool with the init section of the driver, and reserve contiguous space for yourself on init (but then it may as well be statically allocated).

3.5 Process Synchronization

For my next trick, I...fall....a...sleep (SNNXXXX!!)

The thing which really saves multitasking operating systems is that many process sleep when waiting for events to occur. If this were not true, processes would always be burning cycles, and there would really be no big difference between running your processes at the same time, or one after the other.

But when a process sleeps, other processes get to use the CPU. In general, processes sleep when an event they are waiting for has not yet happened. The exception to this is processes which are designed to do work when nothing is happening. For example, you might have a process sitting around using cycles to calculate pi out to a zillion digits. That kind of background process should have its priority set real low so that it isn't executed often when other (presumably more important) processes have work to do.

Since processes sleep when waiting for events, and said events are usually handled by drivers, drivers must cause the processes which called them to sleep if not ready. This is the idea behind the select() call, which will be dealt with in a later chapter.

To illustrate sleeping and waking processes, we will alter our driver from listing 2 by adding a new write function and changing the read function around as follows:

```
========================================
  File Listing 6 (partial): testdata.c
========================================
static char test_data[]="Linux is really funky!\n";
static int wakeups = 0;
static struct wait_queue *wait_queue = NULL;

static int test_write(struct inode * inode, struct file * file,
  char * buffer, int count)
{
  int i;

  printk("Write %d bytes\n",count);
  wake_up_interruptible(&wait_queue);
```

```
  printk("Woke %d processes.\n",wakeups);
  wakeups = 0;
  return count;
}

static int test_read(struct inode * inode, struct file * file,
                     char * buffer, int count)
{
  int offset;

  printk("Test Data Generator, reading %d bytes\n",count);

  printk("Process going to sleep\n");
  wakeups++;
  interruptible_sleep_on(&wait_queue);
  printk("Process has woken up!\n");

  for (offset=0; offset<count; offset++)
    put_fs_byte(test_data[offset % (sizeof(test_data)-1)], buffer+offset);
  return offset;
}
```

Don't forget to put the test_write function in the file_operations struct! But don't compile this driver just yet! Read on...

The operation of this driver is as follows: On a read, put the process to sleep. On a write, wake up all those processes which have gone to sleep in this driver. This will allow the processes to complete the read.

There are two new variables here, wakeups and wait_queue. The wait_queue is a circular queue of processes which are sleeping. It is FIFO, so that the process woken up is the first process which went to sleep.

The kernel handles the queue for us; all we need to do is supply a pointer to the queue and initialize it to NULL (i.e., the queue is empty).

We'll use the wakeups variable to tell us how many processes are taken off the wait_queue (i.e., woken up) – which is the number of processes which have already gone to sleep. So each time a process is slept on, we increment wakeups. When a write request comes in, we wake up wakeups processes and reset wakeups to zero.

Simple, yes? Now we get into the sticky part.

In the Guide, you see that you can choose two ways of sleeping – interruptible or not. Interruptible sleeps can be interrupted (i.e., the process is woken up) by signals (such as SIGUSR) and hardware interrupts. Non-interruptible sleeps can only be interrupted by hardware interrupts. Not even a kill -9 will wake up a non-interruptible process which is sleeping! Suppose you have a signal handler in your process which will react to signal 30 (SIGUSR). That is, you can do kill -30 ¡pid¿. What happens?

When the scheduler gets around to checking the signalled process for runnability, it sees that there is a signal pending. This allows the process to continue to run where it left off, with a twist: when the process leaves kernel mode (the driver call) and enters user mode, the signal handler is called (if there is one). Once the signal handler function exits, one of two things can happen:

- If the original system call exited with -ERESTARTNOINTR, then the process will continue as if it calls the system call again with the same arguments.
- If the original system call did not exit with -ERESTARTNOINTR, but with -ERESTARTNOHAND or -ERESTARTSYS, then the process will continue exitting from the system call with -1, errno -EINTR.
- If the original system call did not exit with -ERSTARTNOINTR, -ERESTARTNOHAND, or -ERESTARTSYS, then the process will continue, exitting from the system call with whatever was returned.

You can see most of this (if you can read mutilated 80386 assembly) in ¡src¿/kernel/sys_call.S and ¡src¿/kernel/signal.c. Although signal handling has been considerably revamped for 0.99pl8, the basic sequence of operations is intact across patch levels. -ERESTARTNOHAND is new in 0.99pl8.

This is important – the driver call should not be completed except for cleanup, since the kernel will return an error for you or redo the system call.

When the process continues to run before calling the signal handler, it picks up where it left off – in the interruptible_sleep_on function. This function takes the process off the wait_queue automatically (which is nice). But then wakeups is not updated (which is not so nice). In that case, when a subsequent write comes in, the number of sleeping processes reported will be wrong!

[pulpit-pounding mode on]

Although for this driver ignoring this is not such a big deal, it is sloppy programming for a driver. Driver code must be so perfect that it operates like a well-oiled machine, with no slip-ups. One error – one bit of code that gets out of sync – and you can at least annoy users and make them throw up their hands in frustration, and at worst panic the kernel and make users throw your code away in frustration! Also, there is nothing worse than spending time debugging an application when the bug is in the driver, or trying to code around a known driver flaw.

[pulpit-pounding mode off]

So how do we solve this out-of-sync problem?

Fact: ignoring interrupts, all processes are atomic when they are in the kernel. That is, unless a process performs an operation which can sleep (like the call to kmalloc we visited above), or a hardware interrupt comes in, the flow of execution goes from entering the kernel to leaving the kernel, with no time taken out to run anything else. This does not mean that the code in user space gets to continue to run. If the process leaves the system call and is not eligible to run, other processes may run and then later on the system call appears to have returned to the process. More on that later.

That fact is good to know. It means that as long as we are sure upon entering the test_write call that wakeups contains the correct number of sleeping processes, test_write will work 100comes in which causes the driver to execute an interrupt handler, we are safe, but here we have no such handler, and so we can ignore that for now. We will deal with interrupts in a later chapter.

So we know that write doesn't really have to be changed. It's really the read that we're concerned about. What we need to do is after we get out of interruptible_sleep_on() we see if we were genuinely woken up through a wakeup call, or if we were signalled. If we were signalled, then we know that the write call wasn't the cause of the wakeup, and so we should really decrement wakeups.

Now for some loose ends. Remember that upon signalling, the kernel only flags the signal for the process, and sets the process to a runnable state. That does not mean that it can run immediately. Another process may get to run first, and that process may very well run the driver's write code, waking up all processes. Of course, we can consider the signalled process to be still asleep when it gets the signal, because it has not yet run its signal handler. So when that other process gets to run the write code, the number of sleeping processes is indeed correct, and wakeups is set to 0.

But now, when the signalled process is run again, the read code will attempt to decrement wakeups, making it -1! The next write will display the wrong number of sleeping processes!

One thing saves us – the fact that we can detect in the read code that the write code was executed, simply because wakeups is 0. Remember that wakeups is incremented before the sleep, so it is guaranteed to be greater than 0 if the write code was not executed before waking up because of a signal.

So if the write code was executed, it really does not make sense to decrement wakeups, so we just say that only if wakeups is non-zero do we decrement.

To implement all this, add this code after the sleep:

```
========================================
  File Listing 7 (partial): testdata.c
========================================
if (current->signal & ~current->blocked) /* signalled? */
{
  printk("Process signalled.\n");
  if (wakeups) wakeups--;
  return -ERESTARTNOINTR; /* Will restart call automagically */
}
```

Now that you've got that straightened out, let's add some more confusion to the mix. Suppose you're in the driver call, doing nice things, and then all of a sudden a nasty timer interrupt (task switch possibility) comes in. What now? Will there be a task switch? No. A RUNNING task in the kernel cannot be switched out, otherwise all hell would break loose. Whew! I'm glad we don't have to pay attention to that!

Well, now that we've gone through all the possible ways signals can make your insides twist, you can code the driver. Remember to put listing 7 into listing 6!

Here's how we're going to test this driver. Several processes will call read (and sleep). When they wake up, they're going to say that they were woken up (as opposed to printing out what they just read – we already know that works). One process will do a write to wake the other processes up. This is the trigger process. Here is the code for the two types of processes:

```
========================================
      File Listing 8: data.c
========================================
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>

/* The reader process */

void signal_handler(int x)
{
  printf("Called signal handler\n");
  signal(SIGUSR1, signal_handler); /* Reset signal handler */
}

void main(void)
{
  int fd;
  char buff[128];
  int rtn;

  signal(SIGUSR1, signal_handler); /* Setup signal handler */

  fd = open("/dev/testdata",O_RDWR);
  printf("/dev/testdata opened, fd=%d\n",fd);
  if (fd<=0) exit(0);

  rtn = read(fd,buff,sizeof(buff));
  printf("Read returns %d\n",rtn);
```

```
  if (rtn<0)
  {
    perror("read");
    exit(1);
  }
  printf("Process woken up!\n");
  close(fd);
}

========================================
      File Listing 9: trigger.c
========================================

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>

/* The writer process */

void main(int argc, char **argv)
{
  int fd;
  char buff[128];
  int rtn;

  fd = open("/dev/testdata",O_RDWR);
  printf("/dev/testdata opened, fd=%d\n",fd);
  if (fd<=0) exit(0);

  if (argc>1)
  {
    kill(atoi(argv[1]),SIGUSR1);
    exit(0);
  }
  rtn = write(fd,buff,sizeof(buff));
  if (rtn<0)
```

```
  {
    perror("write");
    exit(1);
  }
  close(fd);
}
```

Compile these programs using gcc. Now run two or three of the data processes:

```
data &
```

The last thing each of these processes should print is

```
Process going to sleep.
```

because all of these processes are asleep. Now run the trigger program:

```
trigger
```

This should wake up all the other processes, which should say,

```
Process woken up!
```

Had the read function returned an error (like EINTR), they would have said

```
read: <error text>
```

Now, let's test to see if the signal detection and restart mechanism works. Run a single data process in the background via "data &". Remember it's pid. Now, run the trigger process with that pid as an argument:

```
trigger <pid>
```

This will signal ¡pid¿ instead of waking it up via write. The driver should say,

```
Process signalled.
Called signal handler
```

but the process should not wake up, since we restarted the call. Only a write will stop the call.

Experiment 3:

Re-write the driver so that instead of always restarting the call, it returns with EINTR on signal when the read call's count is a special value or values (say anything less than 1000). Test to see if the read call returns EINTR when the trigger program signals the reading process.

3.6 The select call

I want this, that, that...no, THIS, and that. Or, selects!

The select call is one of the most useful calls created for interfacing to drivers. Without it, or a function like it, if you wanted to check a driver for readiness, you would have to poll it regularly. Worse, you would not be able to check multiple drivers for readiness at the same time!

But enough of this. You have select, so rejoice and be happy.

As already implied by the first paragraph, the select system call allows a process to check multiple drivers for readiness. For example, suppose you wanted the process to sit around and wait for one of two file descriptors to be ready for reading. Usually, if a descriptor is not ready for reading and you read it, it will put your process to sleep (or "block"). But you can only read one file descriptor at a time, and here you want to essentially block on _two_ fd's.

In that case, you use the select call. The syntax of select was already explained in the Guide, so let's go about implementing a select function in our driver.

Add the following code to the driver, and put the test_select function in the fops structure:

```
========================================
  File Listing 10 (partial): testdata.c
========================================

static int test_select(struct inode *inode, struct file *file,
                       int sel_type, select_table *wait)
{
  printk("Driver entering select.\n");
  if (sel_type==SEL_IN) /* ready for read? */
  {
    if (wakeups) /* Any process is sleeping in here */
    {
      select_wait(&wait_queue, wait);
      printk("Driver not ready\n");
      return 0;  /* Not ready yet */
    }
    return 1;  /* Ready */
  }
  return 1;  /* Always ready for writes and exceptions */
}
```

==

Here's what this function does. When a process issues a select call with this driver as one of the fd's to select on, the kernel will call test_select with sel_type being SEL_IN. If wakeups is non-zero (that is, processes have read without a process writing) then we will say that the driver is not ready for reading. In this case, select_wait will add the process to the wait_queue and immediately return. The return of 0 indicates that the driver is not ready for the operation.

For any other type of operation (or if there are no processes sleeping in read) we say the driver is ready (return 1).

The only thing that must be remembered is that we are using the same wait_queue structure for processes sleeping in read and processes sleeping in select. This means that writing to the driver will wake up both types of processes. If desired, a different wait_queue could be used, and the appropriate wake up code would have to be written.

Compile this new code into the kernel. We will test this driver by writing a new type of process which will call the select system call. Here is the new process' code:

```
=========================================
        File Listing 11: sel.c
=========================================

#include <stdio.h> /* Doesn't hurt, can only help! */
#include <fcntl.h>
#include <sys/time.h> /* For FD_* and select */

void main(void)
{
  int fd;
  int rtn;
  fd_set read_fds;

  fd = open("/dev/testdata", O_RDWR);
  printf("/dev/testdata opened, fd=%d\n",fd);
  if (fd<=0) exit(0);
  printf("Entering select...\n");
  FD_ZERO(&read_fds);
  FD_SET(fd,&read_fds);
  rtn = select(&read_fds, NULL, NULL, NULL);
  if (rtn<0)
  {
    perror("select");
    exit(0);
  }
  printf("Select returns %d\n",rtn);
}
```

When the kernel is re-loaded, the first test we will perform is to see whether the select call returns immediately given that no processes are sleeping in read. Just run sel – no need to run it in the background. You should see something like:

```
Entering select...
Driver entering select.
Select returns 1
```

This is as it should be – select has determined that one file descriptor is ready for reading.

Our next test is to see whether select sleeps properly. Run this:

```
data &
sel &
trip
```

When sel is run, you should see:

```
Entering select...
Driver entering select.
Read not ready
Driver entering select.
Read not ready
```

The select call in the kernel calls the test_select function again once if the first time the driver is not ready. However, the process is only added to the wait queue once – the first time.

Once the trip program is run, you should see:

```
Process has woken up!
Read returns 1024
Driver entering select.
Select returns 1
```

That is, the data process woke up due to the write, as did the sel process. Note that the test_select function is called once again when the sel process is woken up. This is also a consequence of the kernel design, and is nothing to worry about. Those who are interested in the inner workings of the select call should look in the file ¡src¿/fs/select.c.

A word about signals and select. Since the select call in the driver does not return any error code – just 0 or non-0 – there is no way to decide whether the select call should be restarted or not. Select will return -1, errno EINTR if interrupted by a signal.

Chapter 4

Your Driver and the Hardware

4.1 General

In General the greatest problem for you if you decide to write a device driver is to make the Hardware doing what you want them to do. So the first stage developing a driver should be making familiar with the Hardware. If your card does Port I/O, play around with the Port registers. Some cards like A/D converters uses some tricks to clear FIFO's or select special modes, it depends on the electronically design of the card. Your Task is to get familiar with this tricks.

The second problem is to decide what timing requirements your driver should have. If you want to aquire 20000 samples per second with an A/D card you have to decide how this is done on the card, where the samples go in memory, how the transfer can be done as fast as possible, and so on. For Hardware that produces lot of Data e.g. CCD-Cameras or Framegrabbers you have to get enough Memory where this data can be served.

On the other hand do'nt forget that you have a Multitasking System where other Processes wait for data (to read or to write). Exspecially this People that shoot together selfmade hardware should take to heart this fact. I've seen Stepper Motor cards that needs four Port I/O's per step[1] — very good design to break down your

[1]The four pole motors has been driven by bus-latches with amplifiers at their outputs. The inputs of the latches were connected directly to system bus.

system performance.

All these Problems have an influence on the design of your driver. Fortunally the Linux System has lot of Mechanisms that will help you on this Task.

4.2 Programmed I/O

The simpliest way to do an I/O is the so called Programmed I/O (Polling): You give an request to your hardware and have to wait for response. No problem for DOS-Programmers, you say - you play on some registers to perform the request - you enter a loop that does nothing but looking at some other registers and - if it's ready give your control back. Baah – I say. What does the Program while your Hardware is'nt ready? — Right! .. nothing but burne cycles. Oh sweet conciousness! This time could be used to do many other requests. And what is if your hardware never gets ready ....?

As mentioned in the sections above can this problem be solved with an timeout:

- Program a timer to your expire time with an routine that should be called if the timer expires.
- Use this routine to perform the apropriate operations.
- if it is woken by the timer give control back (dont forget to clear the timer)

Let us have a look how this can be done (this code to make a sound on the PC speaker has been stolen from $LINUX_SOURCE/drivers/char/vt.c):

```
static void

kd_nosound(unsigned long ignored)
{
/* disable counter 2 */
outb(inb_p(0x61)&0xFC, 0x61);
return;
}
```

```
void
kd_mksound(unsigned int count, unsigned int ticks)
{
static struct timer_list sound_timer = { NULL, NULL, 0, 0, kd_nosound };

cli();
del_timer(&sound_timer);
if (count) {
/* enable counter 2 */
outb_p(inb_p(0x61)|3, 0x61);
/* set command for counter 2, 2 byte write */
outb_p(0xB6, 0x43);
/* select desired HZ */
outb_p(count & 0xff, 0x42);
outb((count >> 8) & 0xff, 0x42);

if (ticks) {
sound_timer.expires = ticks;
add_timer(&sound_timer);
}
} else
kd_nosound(0);
sti();
return;
}
```

The other advantage of this concept is the possibility to generate an final timeout if your card is'nt responding.

4.3 Interrupt driven I/O

First, a brief exposition on the Meaning of Interrupts. There are three ways by which a program running in the CPU may be interrupted. The first is the external interrupt. This is caused by an external device (that is, external to the CPU) signalling for attention. These are referred to as "interrupt requests" or "IRQs".

The second method is the exception, which is caused by something internal to the CPU, usually in response to a condition generated by execution of an instruction.

The third method is the software interrupt, which is a deliberately executed interrupt – the INT instruction in assembly. System calls are implemented using software interrupts; when a system call is desired, Linux places the system call number in EAX, and performs an INT 0x80 instruction.

Since drivers usually deal with hardware devices, it is logical that driver interrupts should refer to external interrupts. There are 16 available IRQs – IRQ0 through IRQ15. The following table lists the official uses of the various IRQs:

IRQ	Function
0	timer 0
1	keyboard
2	AT slave 8259 ("cascade")
3	COM2
4	COM1
5	LPT2
6	floppy
7	LPT1
8-12	??????
13	coprocessor error
14,15	??????

Writing drivers which can be interrupted requires care. Be aware that every line you write can be interrupted, and thus cause variable changes to occur. If you really want to protect critical sections from being interrupted, use the cli() and sti() driver calls.

Suppose you wanted to test some kind of funky condition, where success of the condition leads to going to sleep, and being woken up by an interrupt. Consider this code:

```
void driver_interrupt(int unused)
{
  if (!driver_stuff.int_flag) return;  /* Spurious interrupts
                                           are not unheard of */
```

```
  driver_stuff.int_flag=0;
  weird_wacky(); /* Do some weird and wacky stuff
                    here to handle the interrupt */
  disable_ints(); /* Disable the device from issuing interrupts */
  wake_up(&driver_stuff.wait_queue); /* Sets process to TASK_RUNNING */
}

if (conditions_are_ripe())
{
  driver_stuff.int_flag = 1;
  enable_ints(); /* Enable device to interrupt us */
  sleep_on(&driver_stuff.wait_queue); /* Sets process to TASK_UNINTERRUPTIBLE */
}
```

Assume we just leave the conditions_are_ripe code, determining that the conditions are ripe! We have just enabled the device to interrupt the machine. So we are now about to enter the sleep_on code, and what should happen but the pesky device issues an interrupt. Ka-chunk! and we enter the driver_interrupt routine, which does some weird and wacky stuff to handle the interrupt, and then we disable the device's interrupts. Ka-ching! we enter the wake_up function which sets the process up to run again. Boink! we exit the interrupt handler and commence where we left off (just about to enter the sleep_on code). Vooosh! we're now sleeping the process, awaiting an interrupt which will never occur, since the interrupt handler disabled the device from interrupts! What to do?

Use cli() and sti() to protect the critical sections of code:

```
cli();
if (conditions_are_ripe())
{
  driver_stuff.int_flag = 1;
  enable_ints(); /* Enable device to interrupt us */
  sleep_on(&driver_stuff.wait_queue); /* Sets process to TASK_UNINTERRUPTIBLE */
}
else sti();
```

First we clear interrupts. This is not the same as disabling device interrupts! This actually prevents a hardware interrupt from causing the CPU to execute interrupt code. In effect, the interrupt is deferred.

Now we can do our check and perform sleep_on, secure in the knowledge that the interrupt handler cannot be called. The sleep_on (and interruptible_ sleep_on) call has a sti() in it in the right place, so you don't have to worry about calling sti() before sleep_on, and running into a race condition again.

Of course, with any interruptible device driver, you must be careful never to spend too much time in the interrupt routine if you are expecting more than one interrupt, because you may miss your second interrupt.

4.3.1 Timeouts and Interrupts

Suppose you wanted to sleep on an interrupt, but also time out after a period of time. You could always use the add_timer, but that's frowned upon because there are only a limited number of timers available – currently there are 64.

The usual solution is to manually alter the current process's timeout:

```
current->timeout = jiffies + X;
interruptible_sleep_on(&driver_stuff.wait_queue);
```

(Interruptible sleep_on must be used here to allow a timeout to interrupt the sleep). This will cause the scheduler to set the task running again when X jiffies has gone by. Even if the timeout goes off and the process is allowed to continue running, it is probably a good idea to call wake_up_interruptible in case the process needs to be rescheduled.

To find out if it was a timeout which caused the process to wake up, check current-¿timeout. If it is 0, a timeout occurred. Otherwise it should remain what you set it at. If a timeout did not occur, and something else woke the process up, you should set current-¿timeout to 0 to prevent the timeout from continuing.

The disadvantage of this method is that the process can only have one timeout at a time. Over *all* drivers.

4.4 Drivers and signals:

When a process is sleeping in an interruptible state, any signal can wake it up. This is the sequence of events which occurs when a sleeping process receives a signal:

- Set current-¿signal.
- Set the process to a runnable state.
- Execute the rest of the driver call.
- Run the signal handler.
- If the driver call in step 3 returned -ERESTARTNOHAND or -ERESTARTNOINTR, then return from the driver call with EINTR. If the driver call in step 3 returned -ERESTARTSYS, then restart the driver call. Otherwise, just return with whatever was returned from the driver call.

In the driver, you can tell if a sleep has been interrupted by a signal with the following code:

```
if (current->signal & ~current->blocked)
{
  /* Do things based on sleep interrupted by signal */
}
```

4.5 DMA-Transfers

Some Hardware supports an Mechanism to tranfer a Block of Data from System-Memory to a Memory at the card or backwards(Hard and Floppy Disks, Tapes, some A/D cards). This is done by special controllers so that the CPU load is low for this transactions. This is the so called Direct Memory Access (DMA). This sounds wondeful but is a very difficult task in detail expecially for the driver programmer. It requires a complex interaction between interrupts, hardware-controllers and memory management.

BTW, its possible so let's go on. First we have to understand the Principle of operation.

On the initiate Phase of an DMA Operation the CPU gives the DMA-Controller the physical memory adress where the data block is to go and the number of data bytes to transfer, under Linux this is controlled by the two functions set_dma_addr() and set_dma_count(). Once the DMA transfer has been initiated the controller copies the first byte (or word) to the adress specified at the Adress Register. Then it increments the DMA-Adress Register and decrements the Count Register. This is repeated until the count register reached the zero value and the controller causes an Interrupt.

For the DMA Transfer several lines of the system bus are used for the Handshake between The Host-side DMA controller and the Card-side DMA controller while the transfer is done. These lines are grouped in so called DMA-Channels. To use an DMA-Channel in linux you have to tell this to the kernel similar to IRQ use. This is done by the request_dma() function that takes the required DMA-Channel as argument. It should be called in your init function.

The PC has two DMA controllers, the first is connected to channel 0-3 and used for byte transfers (that means byte-by-byte on odd adresses), the other one is used for word (16 bit word-by-word) transfers. At each Word transfer the adress register has to be incremented by 2 and the count register decremented by one (number of words). The PC designers shifted the adress lines of the 2nd controller by one (multiplication by 2). All DMA transfers are limited to the lower 16MB of *physical* memory. Note that addresses loaded into registers must be *physical* addresses, not logical addresses (which may differ if paging is active).

Due to this fact(s) there are some restrictions for DMA-transfers[2]:

- ALL controller registers are 8 bits only, regardless of transfer size
- channel 4 is not used - cascades 1 into 2.
- channels 0-3 are byte - addresses/counts are for physical bytes
- channels 5-7 are word - addresses/counts are for physical words
- transfers must not cross physical 64K (0-3) or 128K (5-7) boundaries

[2]This Rules come from ¡asm/dma.h¿ from Hennus Bergman

- transfer count loaded to registers is 1 less than actual count
- controller 2 offsets are all even (2x offsets for controller 1)
- page registers for 5-7 don't use data bit 0, represent 128K pages
- page registers for 0-3 use bit 0, represent 64K pages

```
(from <asm/dma.h> by Hennus Bergman)

 *  Address mapping for channels 0-3:
 *
 *   A23 ... A16 A15 ... A8  A7 ... A0    (Physical addresses)
 *    |  ...  |   |  ... |   |  ... |
 *    |  ...  |   |  ... |   |  ... |
 *    |  ...  |   |  ... |   |  ... |
 *   P7  ...  P0  A7 ... A0  A7 ... A0
 * |    Page    | Addr MSB | Addr LSB |   (DMA registers)
 *
 *  Address mapping for channels 5-7:
 *
 *   A23 ... A17 A16 A15 ... A9 A8 A7 ... A1 A0    (Physical addresses)
 *    |  ...  |   \   \   ... \  \  \  ... \  \
 *    |  ...  |    \   \   ... \  \  \  ... \  (not used)
 *    |  ...  |     \   \   ... \  \  \  ... \
 *   P7  ...  P1 (0) A7 A6  ... A0 A7 A6 ... A0
 * |      Page      |  Addr MSB   |  Addr LSB  |   (DMA registers)
```

Again, channels 5-7 transfer *physical* words (16 bits), so addresses and counts *must* be word-aligned (the lowest address bit is *ignored* at the hardware level, so odd-byte transfers aren't possible). Transfer count (*not # bytes*) is limited to 64K, represented as actual count - 1 : 64K =¿ 0xFFFF, 1 =¿ 0x0000. Thus, count is always 1 or more, and up to 128K bytes may be transferred on channels 5-7 in one operation.

To do a DMA-Transfer do the following:

- Enable DMA on your card.
  Look to your hardware documentation how this has to be done. Some cards

have registers to enable DMA and IRQ, it depends on your hardware what you have to set.

- Set up Host-side registers

  - First clear the "DMA Pointer Flip Flop". Call clear_dma_ff(unsigned int dmanr) with your DMA-Channel as argument.
  - Set the DMA-mode register to one of DMA_MODE_READ, DMA_MODE_WRITE or DMA_MODE_CASCADE (In this mode the DMA-Controller is used as no-operation slave[3]). set_dma_mode(unsigned int dmanr, char mode) will do this Task for you.
  - Now set DMA-count and DMA adress register but remember the rules above. Use set_dma_addr(unsigned int dmanr, unsigned int a) (If you want to set the Page register only you have the set_dma_page(unsigned int dmanr, char pagenr) routine for this).

    set_dma_count(unsigned int dmanr, unsigned int count) sets the DMA-count register while 'count' represents *bytes* and must be even for channels 5-7.

- start dma-transfer
  Now trigger your hardware to do the transfer. If your card supports DMA-IRQ you should start the transfer and wait for an Interrupt. You could read the residue count to check out your transfer has been completed. To do this get_dma_residue(unsigned int dmanr) should return zero.

Note: There is a very good example for DMA-Programming written by Hennus: look at /drivers/char/tpqic02.c (the QIC-02 Tape driver).

[3]This Mode is used on Network-cards for example, if you intend to use this mode, have a look on /drivers/net/lance.c .

Chapter 5

Special Concepts in Linux

5.1 Accesing Ports in User space

For some operations on the hardware the standard driver calls as read() or write() are too slow. In this case it is better to permit the i/o port access directly to the user process. This has been done for the vgalib graphics library that is used by the X11 server to get maximal performance. For this purpose Linux offers the ioperm() call that permits port access.

```
#include <unistd.h>

int  ioperm(unsigned  long  from,  unsigned  long num, int turn_on);
```

Ioperm sets the port access permission bits for the process for num bytes starting from port address from to the value turn_on. The use of ioperm require root privileges[1]

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

To use ioperm in a user programm it must run suid root.

[1]Only the first 0x3ff I/O ports can be specified in this manner. For more ports, the iopl function must be used. (see manual page)

5.2 Accessing Kernel Memory from User Space

The second Problem in context with video cards is the Memory of the card that have to be acessed from user space. This Memory can be mapped to user space using the mmap() call.

To this a short Example from vgalib:

```
       #include <sys/types.h>
       #include <sys/mman.h>

        [....]

/* open /dev/mem */
if(( mem_fd = open("/dev/mem",O_RDWR))==NULL){
error_and_exit();
}

/* mmap graphics memory */
if(( graph_mem = malloc(GRAPH_SIZE + (PAGE_SIZE-1) ))==NULL){
error_and_exit();
}

/* align to page boundary */
if((unsigned long)graph_mem % PAGE_SIZE)
graph_mem += - ((unsigned long) graph_mem % PAGE_SIZE);

/* now map the memory */
if((graph_mem = (unsigned char *) mmap(
(caddr_t) graph_mem,
GRAPH_SIZE,
PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_FIXED,
mem_fd,
GRAPH_BASE
  )) < (long) 0){
error_and_exit();
}
```

First the /dev/mem device is opened, then enough memory is allocated to do the map on a 4096 byte boundary. GRAPH_SIZE is the size of the Memory of the card beginning at GRAPH_BASE.

Any access to graph_mem now is mapped to screen memory.

For details on the flags please refer to the manual page of mmap.

5.3 Reading the kernel Environment

Linux provides an Method to configure Drivers at boot time without rebuilding the kernel. This configuration options are given by lilo to the kernel as a string. You can give this parameters at boot time at the lilo boot prompt, for example:

```
LILO boot: linux mydriver=0x240,4
```

The kernel passes this options to the driver's bmouse_setup routine. For example:

```
static int my_driver_irq = DEFAULT_IRQ;
static int my_driver_base = DEFAULT_BASE;


void my_driver_setup(char *str, int *ints)
{
if (ints[0] == 2){              /* ints[0] holds the number of arguments */
my_driver_base=ints[1];
my_driver_irq=ints[2];
}
}
```

To bind the option "mydriver=" and the _setup routine together and to make it recognized by the kernel, you have to modify the bootsetups[] array in $LINUX_SOURCE/init/main.c:

```
struct {
char *str;
void (*setup_func)(char *, int *);
```

```
} bootsetups[] = {
{ "reserve=", reserve_setup },
        :
        : /* some option strings */
        :
#ifdef CONFIG_MYDRIVER
{ "mydriver=", my_driver_setup },
#endif
        :
        : /* some other options */
        :
        :
{ 0, 0 }
};
```

Note that the setup routine is called before any init routine. This mechanism is used by several drivers to configure IRQ and base adress.

5.4 Writing Loadable Modules

2.5 This section has partially been stolen from Jon. Tomb's doc file in the modutils package.

A module is a collection of code which can be dynamicly linked into the kernel. Modules can contain things such as device drivers, system calls, and file system types [system calls are actually nolonger permitted].

A module consists of a normal .o file. If you have multiple source files, combine them into a single .o file using "ld -r".

How does the kernel know to use my module?

Your code must contain a routine named "init_module" and a routine named "cleanup_module". These routines are defined as:

```
int init_module(void);
void cleanup\_module(void);
```

"Init_module" will be called when the module is loaded. It should return zero on success and -1 on failure. "Cleanup_module" will be called before the module is deleted. It should undo the operations performed by "init_module". For example, if a module defines a special device type "init_module" should register the device major number and and "cleanup_module" should remove it.

Also note for the module to be installed it must contain a string kernel_version[] that matches the release of the current kernel (i.e. matches uname -r).

Here is a short Example of the parts of an Driver that makes it to an Module:

```
### other includes
#include <linux/module.h>


char kernel_version[] = "1.0"; /* or what ever uname -r says */

### other driver routines

/*
 * Our special open code.
 * MOD_INC_USE_COUNT make sure that the driver memory is not freed
 * while the device is in use.
 */
static int
hw_open( struct inode* ino, struct file* filep)
{
   MOD_INC_USE_COUNT;
   return 0;
}

/*
 * Now decrement the use count.
 */
static void
hw_close( struct inode* ino, struct file* filep)
{
   MOD_DEC_USE_COUNT;
```

```
}

static struct file_operations hw_fops = {
hw_lseek,
hw_read,
NULL,
NULL, /* hw_readdir */
NULL, /* hw_select */
NULL, /* hw_ioctl */
NULL,
hw_open,
hw_close,
NULL /* fsync */
};



/*
 * And now the modules code and kernel interface.
 */
extern int printk( const char* fmt, ...);


int
init_module( void) {
printk( "drv_hello.c:  init_module called\n");
if (register_chrdev(HW_MAJOR, "hw", &hw_fops)) {
  printk("register_chrdev failed: goodbye world :-(\n");
  return -EIO;
} else
  printk( "Hello, world\n");

return 0;
}

void
cleanup_module( void) {
  printk( "drv_hello.c:  cleanup_module called\n");
```

```
  if (MOD_IN_USE)
    printk("hw: device busy, remove delayed\n");


  if (unregister_chrdev(HW_MAJOR, "hw") != 0) {
    printk("cleanup_module failed\n");
  } else {
    printk("cleanup_module succeeded\n");
  }
}
```

Note that there is a new kernel routine unregister_chrdev() that removes the hooked fops functions from the kernel. Be shure to call this routine at cleanup. Do'nt forget to free irq's, dma's and memory pages.

The Macros MOD_INC_USE_COUNT and MOD_DEC_USE_COUNT are used in open/close to prevent the module being freed while it is still running. This is checked by MOD_IN_USE at cleanup.

For loading, unloading and listing modules there are several programs to do this tasks.

"insmod object_file" will install a module into the kernel. The name of the module consists of the name of the object file with the .o and any directory names removed.

"lsmod" will produce a list of modules currently in the kernel, including the module name and size in pages.

"rmmod name" will remove a module. The argument should be the module name, not the object file name.

Appendix A

Reference Guide

A.1 Kernel-callable functions

Note: There is no close for a character device. There is only release. See the file data structure below to find out how to determine the number of processes which have the device open.

init : Initializes the driver on bootup.

```
unsigned long driver_init(unsigned long kmem_start, unsigned long kmem_end)

Arguments: kmem_start -- the start of kernel memory
           kmem_end   -- the end of kernel memory

Returns: The new start of kernel memory.  This will be different from the
         kmem_start argument if you want to allocate memory for the driver.
```

The arguments you use depends on what you want to do. Remember that since you are going to add your init function to kernel/chr_dev/mem.c, you can make your call anything you like, but you have access to the kernel memory start and end.

Generally, the init function initializes the driver and hardware, and displays some message telling of the availability of the driver and hardware. In addition, the register_chrdev function is usually called here.

open : Open a device

```
static int driver_open(struct inode * inode, struct file * file)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device

Returns: 0 on success,
       -errno on error.
```

This function is called whenever a process performs open (or fopen) on the device special file. If there is no open function for the driver, nothing spectacular happens. As long as the /dev file exists, the open will succeed.

read : Read from a device

```
static int driver_read(struct inode * inode, struct file * file,
                       char * buffer, int count)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device
           buffer   -- pointer to the buffer in user space to read into
           count    -- the number of bytes to read

Returns: -errno on error
         >=0 : the number of bytes actually read
```

If there is no read function for the driver, read calls will return EINVAL.

write : Write to a device

```
static int driver_write(struct inode * inode, struct file * file,
                        char * buffer, int count)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device
           buffer   -- pointer to the buffer in user space to write from
           count    -- the number of bytes to write

Returns: -errno on error
         >=0 : the number of bytes actually written
```

If there is no write function for the driver, write calls will return EINVAL.

****

lseek : Change the position offset of the device

```
static int driver_lseek(struct inode * inode, struct file * file,
                        off_t offset, int origin)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device
           offset   -- offset from origin to move to (bytes)
           origin   -- origin to move from :
                       0 = from origin 0 (beginning)
                       1 = from current position
                       2 = from end

Returns: -errno on error
         >=0 : the position after the move
```

See Also: Data Structure 'file'

If there is no lseek function for the driver, the kernel will take the default seek action, which is to alter the file-¿f_pos element. For origins of 2, the default action results in -EINVAL if file-¿f_inode is NULL, or it sets file-¿f_pos to file-¿f_inode-¿i_size + offset otherwise.

ioctl : Various device-dependent services

```
static int driver_ioctl(struct inode *inode, struct file *file,
                        unsigned int cmd, unsigned long arg)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device
           cmd      -- the user-defined command to perform
           arg      -- the user-defined argument.  You may use this
                       as a pointer to user space, since sizeof(long)==
                       sizeof(void *).

Returns: -errno on error
         >=0 : whatever you like! (user-defined)
```

For cmd, FIOCLEX, FIONCLEX, FIONBIO, and FIOASYNC are already defined. See the file linux/fs/ioctl.c, sys_ioctl to find out what they do. If there is no ioctl call for the driver, and the ioctl command performed is not one of the four types listed here, ioctl will return -EINVAL.

select : Performs the select call on the device:

```
static int driver_select(struct inode *inode, struct file *file,
                         int sel_type, select_table * wait)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device
           sel_type -- the select type to perform :
                        SEL_IN (read)
                        SEL_OUT (write)
                        SEL_EX (exception)
           wait     -- see the section "Some Notes" for select.

Returns: 0 if the device is not ready to perform the sel_type operation
         !=0 if it is.
```

See the "Some Notes" section 'way below on information on how to use the select call in drivers. If there is no select call for the driver, select will act as if the driver is ready for the operation.

release : Release device (no process holds it open)

```
static void driver_release(struct inode * inode, struct file * file)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device
```

The release call is activated only when the process closes the device as many times as it has opened it. That is, if the process has opened the device five times, then only when close is called for the fifth time will release be called (that is, provided there were no more calls to open!). If there is no release call for the driver, nothing spectacular happens.

****

readdir : Get the next directory entry

```
static int driver_readdir(struct inode *inode, struct file *file,
                          struct dirent *dirent, int count)

Arguments: inode    -- pointer to the inode structure for this device
           file     -- pointer to the file structure for this device
           dirent   -- pointer to a dirent ("directory entry") structure
           count    -- number of entries to read (currently always 1)

Returns: 0 on success
         -errno on failure.
```

If there is no readdir function for the driver, readdir will return -ENOTDIR. This is really for file systems, but you can probably use it for whatever you like in a non-fs device, as long as you return a dirent structure.

See Also: dirent (data structure)

mmap : Forget this. According to the source (src/linux/mm/mmap.c), for character devices only /dev/[k]mem may be mapped. Besides, I'm not too clear on what it will do.

A.2 Data structures

dirent : Information about files in a directory.

```
#include <linux/dirent.h>

struct dirent {
        long            d_ino;                  /* Inode of file */
        off_t           d_off;
        unsigned short  d_reclen;
        char            d_name[NAME_MAX+1];     /* Name of file */
};
```

file : Information about open files

According to the Hacker's Guide to Linux, this structure is mainly used for writing filesystems, not drivers. However, there is no reason it cannot be used by drivers.

```
#include <linux/fs.h>

struct file {
        mode_t f_mode;
        dev_t f_rdev;                  /* needed for /dev/tty */
         off_t f_pos;                   /* Curr. posn in file */
         unsigned short f_flags;        /* The flags arg passed to open */
          unsigned short f_count;        /* Number of opens on this file */
              unsigned short f_reada;
              struct inode * f_inode;         /* pointer to the inode struct */
              struct file_operations * f_op;  /* pointer to the fops struct */
};
```

file_operations : Tells the kernel which function to call for which kernel function.

```
#include <linux/fs.h>

struct file_operations {
        int  (*lseek) (struct inode *, struct file *, off_t, int);
        int  (*read) (struct inode *, struct file *, char *, int);
         int  (*write) (struct inode *, struct file *, char *, int);
         int  (*readdir) (struct inode *, struct file *, struct dirent *, int);
         int  (*select) (struct inode *, struct file *, int, select_table *);
         int  (*ioctl) (struct inode *, struct file *, unsigned int,
                         unsigned int);
          int  (*mmap) (void);
          int  (*open) (struct inode *, struct file *);
          void (*release) (struct inode *, struct file *);
           int  (*fsync) (struct inode *, struct file *);
};
```

inode : Information about the /dev/xxx file (or inode)

```
#include <linux/fs.h>

struct inode {
        dev_t           i_dev;
        unsigned long   i_ino;  /* Inode number */
        umode_t         i_mode; /* Mode of the file */
        nlink_t         i_nlink;
        uid_t           i_uid;
        gid_t           i_gid;
        dev_t           i_rdev;  /* Device major and minor numbers */
        off_t           i_size;
        time_t          i_atime;
        time_t          i_mtime;
        time_t          i_ctime;
        unsigned long   i_blksize;
        unsigned long   i_blocks;
        struct inode_operations * i_op;
        struct super_block * i_sb;
        struct wait_queue * i_wait;
        struct file_lock * i_flock;
        struct vm_area_struct * i_mmap;
        struct inode * i_next, * i_prev;
        struct inode * i_hash_next, * i_hash_prev;
        struct inode * i_bound_to, * i_bound_by;
        unsigned short i_count;
        unsigned short i_flags;  /* Mount flags (see fs.h) */
        unsigned char i_lock;
        unsigned char i_dirt;
        unsigned char i_pipe;
        unsigned char i_mount;
        unsigned char i_seek;
        unsigned char i_update;
        union {
                struct pipe_inode_info pipe_i;
                struct minix_inode_info minix_i;
                struct ext_inode_info ext_i;
                struct msdos_inode_info msdos_i;
                struct iso_inode_info isofs_i;
                struct nfs_inode_info nfs_i;
```

```
        } u;
};
```

See Also: Driver Calls: MAJOR, MINOR, IS_RDONLY, IS_NOSUID, IS_NODEV, IS_NOEXEC, IS_SYNC

A.3 Driver calls

add_timer : Cause a function to be executed when a given amount of time has passed.

```
#include <linux/sched.h>

void add_timer(long jiffies, void (*fn)(void))

Arguments: jiffies -- The number of jiffies to time out after.
             fn      -- The function in kernel space to run after timeout.
```

Note! This is NOT process-specific! If you are looking for a way to have a process go to sleep and timeout, look for ? Excessive use of this function will cause the kernel to panic if there are too many timeouts active at once.

cli : Macro, Prevent interrupts from occuring

```
#include <asm/system.h>

#define cli() __asm__ __volatile__ ("cli"::)
```

See Also: sti

free_irq : Free a registered interrupt

```
#include <linux/sched.h>
```

```
void free_irq(unsigned int irq)

Arguments: irq -- the interrupt level to free up
```

See Also: request_irq

get_fs_byte, get_fs_word, get_fs_long : Get data from user space

Purpose: Allows a driver to access data in user space (which is in a different segment than the kernel!)

```
#include <asm/segment.h>

inline unsigned char get_fs_byte(const char * addr)
inline unsigned short get_fs_word(const unsigned short *addr)
inline unsigned long get_fs_long(const unsigned long *addr)

Arguments: addr -- the address in user space to get data from

Returns: the value in user space.
```

See Also: memcpy_fromfs, memcpy_tofs, put_fs_byte, put_fs_word, put_fs_long

inb, inb_p : Inputs a byte from a port

```
#include <asm/io.h>

inline unsigned int inb(unsigned short port)
inline unsigned int inb_p(unsigned short port)

Arguments: port -- the port to input a byte from

Returns:  Byte received in the low byte.  High byte unused.
```

See Also: outb, outb_p

IS_RDONLY, IS_NOSUID, IS_NODEV, IS_NOEXEC, IS_SYNC: Macros, check the status of the device on the filesystem

```
#include <linux/fs.h>

#define IS_RDONLY(inode) (((inode)->i_sb) && ((inode)->i_sb->s_flags &
                           MS_RDONLY))
#define IS_NOSUID(inode) ((inode)->i_flags & MS_NOSUID)
#define IS_NODEV(inode) ((inode)->i_flags & MS_NODEV)
#define IS_NOEXEC(inode) ((inode)->i_flags & MS_NOEXEC)
#define IS_SYNC(inode) ((inode)->i_flags & MS_SYNC)
```

kfree, kfree_s : Free memory which has been kmalloced.

```
#include <linux/kernel.h>

#define kfree(x) kfree_s((x), 0)
void kfree_s(void * obj, int size)

Arguments : obj  -- pointer to kernel memory you want to free
            size -- size of block you want to free (0 if you don't know
                    or are lazy -- slows things down)
```

kmalloc : Allocate memory in kernel space

```
#include <linux/kernel.h>

void * kmalloc(unsigned int len, int priority)

Arguments: len      -- the length of the memory to allocate.  Must not be bigger
                       than 4096.
           priority -- GFP_KERNEL or GFP_ATOMIC.  GFP_ATOMIC causes kmalloc
                        to return NULL if the memory could not be found
                       immediately.  GFP_KERNEL is the usual priority.

Returns: NULL on failure, a pointer to kernel space on success.
```

memcpy_fromfs, memcpy_tofs : Copies memory from user(fromfs)/kernel(tofs) space to kernel/user space

```
#include <asm/segment.h>

inline void memcpy_fromfs(void * to, const void * from, unsigned long n)
inline void memcpy_tofs(void * to, const void * from, unsigned long n)

Arguments: to   -- Address to copy data to
           from -- Address to copy data from
           n    -- number of bytes to copy
```

See Also: get_fs_byte, get_fs_word, get_fs_long, put_fs_byte, put_fs_word, put_fs_long

Warning! Get the order of arguments right!

MAJOR, MINOR : Macros, get major/minor device number from inode i_dev entry.

```
#include <linux/fs.h>

#define MAJOR(a) (((unsigned)(a))>>8)
#define MINOR(a) ((a)&0xff)
```

outb, outb_p : Outputs a byte to a port

```
#include <asm/io.h>

inline void outb(char value, unsigned short port)
inline void outb_p(char value, unsigned short port)

Arguments: value -- the byte to write out
           port  -- the port to write it out on
```

See Also: inb, inb_p

printk : Kernel printf

```
#include <linux/kernel.h>

int printk(const char *fmt, ...)

Arguments: fmt -- printf-style format
            ... -- var-arguments, printf-style

Returns: Number of characters printed.
```

put_fs_byte, put_fs_word, put_fs_long : Put data into user space

Purpose: Allows a driver to put a byte, word, or long into user space, which is at a different segment than the kernel.

```
#include <asm/segment.h>

inline void put_fs_byte(char val,char *addr)
inline void put_fs_word(short val,short * addr)
inline void put_fs_long(unsigned long val,unsigned long * addr)

Arguments: addr -- the address in user space to get data from

Returns: the value in user space.
```

See Also: memcpy_fromfs, memcpy_tofs, get_fs_byte, get_fs_word, get_fs_long

Warning! Get the order of arguments right!

register_chrdev : Register a character device with the kernel

```
#include <linux/fs.h>
#include <linux/errno.h>

int register_chrdev(unsigned int major, const char *name,
                            struct file_operations *fops)

Arguments: major -- the major device number to register as
               name  -- the name of the device (currently unused)
                   fops  -- a file_operations structure for the device.

Returns: -EINVAL if major is >= MAX_CHRDEV (defined in fs.h as 32)
          -EBUSY if major device has already been allocated
           0 on success.
```

request_irq : Request to perform a function on an interrupt

```
#include <linux/sched.h>
#include <linux/errno.h>
```

```
int request_irq(unsigned int irq, void (*handler)(int))

Arguments: irq     -- the interrupt to request.
            handler -- the function to handle the interrupt.  The interrupt
                       handler should be of the form void handler(int).
                              Unless you really know what you are doing, don't
                            use the int argument.

Returns: -EINVAL if irq>15 or handler==NULL
          -EBUSY if irq is already allocated.
          0 on success.
```

See Also: free_irq

select_wait : Add a process to the select-wait queue

```
#include <linux/sched.h>

inline void select_wait(struct wait_queue ** wait_address, select_table * p)

Arguments: wait_address -- Address of a wait_queue pointer
            p            -- Address of a select_table
```

Devices which use select should define a struct wait_queue pointer and initialize it to NULL. select_wait adds the current process to a circular list of waits. The pointer to the circular list is wait_address. If p is NULL, select_wait does nothing, otherwise the current process is put to sleep.

See Also: sleep_on, interruptible_sleep_on, wake_up, wake_up_interruptible

sleep_on, interruptible_sleep_on : Put the current process to sleep.

```
#include <linux/sched.h>

void sleep_on(struct wait_queue ** p)
void interruptible_sleep_on(struct wait_queue ** p)

Arguments: q -- Pointer to the driver's wait_queue (see select_wait)
```

sleep_on puts the current process to sleep in an uninterruptible state. That is, signals will not wake the process up. The only thing which will wake a process up in this state is a hardware interrupt (which would call the interrupt handler of the driver) – and even then the interrupt routine needs to call wake_up to put the process in a running state.

interruptible_sleep_on puts the current process to sleep in an interruptible state, which means that not only will hardware interrupts get through, but also signals and process timeouts ("alarms") will cause the process to wake up (and execute interrupt or signal handlers). A call to wake_up_interruptible is necessary to wake up the process and allow it to continue running where it left off.

See Also: select_wait, wake_up, wake_up_interruptible

sti : Macro, Allow interrupts to occur

```
#include <asm/system.h>

#define sti() __asm__ __volatile__ ("sti"::)
```

See Also: cli

sys_getegid, sys_getgid, sys_getpid, sys_getppid, sys_getuid, sys_geteuid : Funky functions which get various information about the current process,

```
#include <linux/sys.h>

int sys_getegid(void)
int sys_getgid(void)
int sys_getpid(void)
int sys_getppid(void)
int sys_getuid(void)
int sys_geteuid(void)

sys_getegid gets the effective gid of the process.
sys_getgid gets the group ID of the process.
sys_getpid gets the process ID of the process.
sys_getppid gets the process ID of the process' parent.
sys_geteuid gets the effective uid of the process.
sys_getuid gets the user ID of the process.
```

wake_up, wake_up_interruptible : Wake up _all_ processes waiting on the wait queue.

```
#include <linux/sched.h>

void wake_up(struct wait_queue **q)
void wake_up_interruptible(struct wait_queue **q)

Arguments: q -- Pointer to the driver's wait_queue (see select_wait)
```

See Also: select_wait, sleep_on, interruptible_sleep_on

When a process issues a select call, it is checking to see if the given devices are ready to perform the given operations. For example, suppose you want a driver to have a command written to it, and to disallow further commands until the current command is complete. Well, in the write call you would block commands if there is already a command operating (for example, waiting for a board to do something). But that would require the process to write over and over again until it succeeds. That just burns cycles.

The select call allows a process to determine the availability of read and write. In the above example, one merely has to select for write on that device's file descriptor (as returned by open), and the process would be put to sleep until the device is ready to be written to.

The kernel will call the driver's driver_select call when the process issues a select call. The arguments to the driver_select call are detailed above. If the wait argument is non-NULL, and there is no error condition caused by the select, driver_select should put the process to sleep, and arrange to be woken up when the device becomes ready (usually through an interrupt).

If, however, the wait argument is NULL, then the driver should quickly see if the device is ready, and return even if it is not. The select_wait function does this already for you (see further).

Putting the process to sleep does not require calling a sleep_on function. It is the select_wait function which is called, with the p argument being equal to the wait argument passed to driver_select.

select_wait is pretty much equivalent to interruptible_sleep_on in that it adds the current process to the wait queue and sleeps the process in an interruptible state. The internals of the differences between select_wait and interruptible_sleep_on are relatively irrelevant here. Suffice it to say that to wake the process up from the select, one needs to perform the wake_up_interruptible call. When that happens, the process is free to run.

However, in the case of interruptible_sleep_on, the process will continue running after the call to interruptible_sleep_on. In the case of select_wait, the process does not. driver_select is called as a "side effect" of the select call, and so completes even when it calls select_wait. It is not select_wait which sleeps the process, but the select call. Nevertheless, it is required to call select_wait to add the process to the wait-queue, since select will not do that.

All one needs to remember for driver_select is:

- Call select_wait if the device is not ready, and return 0.
- Return 1 if the device is ready.

Calling select with a timeout is really no different to the driver than calling it without select. But there is one crucial difference. Remember timing out on interrupts above? Well, interrupt timeouts and select timeouts cannot co-exist. They both use current-¿timeout to wake the process up after a period of time. Remember that!

A.4 Installation notes

Before you sit down and write your first driver, first make sure you understand how to recompile the kernel. Then go ahead and recompile it! Recompilation of the kernel is described in the FAQ. If you can't recompile the kernel, you can't install your driver into the kernel. [Although I hear tell of a package on sunsite which can load and unload drivers while the kernel is running. Until I test out this package, I won't include instructions for it here.]

For character devices, you need to go into the mem.c file in the (source)/linux/kernel/chr_dev directory, to the chr_dev_init function, and add your init function to it. Recompile the kernel, and away you go!

(BTW, would you manually have to do a mknod to make the /dev/xxx entry for your driver? Can you do it in the init function?)

In general, one installs a device special file in /dev manually, by using mknod:

```
mknod /dev/xxx c major minor
```

If you registered your character driver as major device X, then all accesses to /dev/xxx where major==X will call your driver functions.