Journal 3, Creation of device drivers and applications

1 Objectives

To use the Spartan3 PCI Express Starter kit, some device drivers are needed. The objective of this journal is to describe the creation of such drivers for both Windows XP and Linux 2.6 that allows other applications to read from and write to registers on the board.

2 Problem analysis

The Spartan3 PCI Express Starter kit includes a demo application for Windows. The demo application consists of the following:

- A Windows device driver for the board, providing functions for setting and retrieving the state of the leds, and retrieving the state of the buttons. The driver is written using C.
- A small application with a GUI, which allows a user to interact with the board. The application is written in Visual Basic, and accesses the device driver through *drivermanager.dll* (the sourcecode for the .dll is unavailable).

The source code for this, except the interface between the application and the device driver, is available in the "*Tools\Spartan3 PCI Express Starter kit Demo*" folder on CD1.
The Windows driver will need a few modifications, for instance the addition of a few primitive functions for reading and writing single registers, but is, apart from that, useable. For Linux, a driver will need to be written from scratch.
The Windows application is close to useless, as SDU does not use Visual Basic (and so there is no developing environment available), and as C/C++ is the preferred language for the application anyway. So instead a new application will be written from scratch using C/C++. The same goes for the Linux versions. The applications will be created as console applications, to avoid spending too much time with GUIs.
So the following needs to be done or created:

- The existing Windows device driver from the demo needs to be modified, by adding functionality to set and get single registers directly.
- A Windows console application, which can use the device driver to communicate with the Spartan3 board.
- A Linux 2.6 device driver with functionality for setting and getting registers needs to be created.
- A Linux console application that can access the functionality of the device driver and communicate with the Spartan3 board.

3 Device driver basics

A typical workflow for a PCI/PCI Express device driver can be seen on figure 3.a.

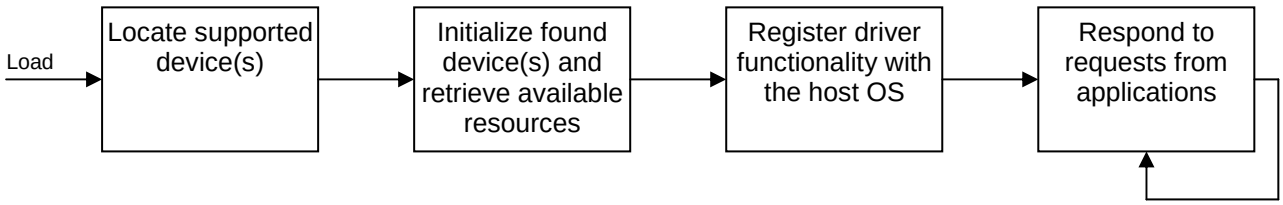

Figure 3.a: A typical flow for a PCI/PCI Express device driver.

A device driver needs a specified entry function that is run when the driver is loaded. This entry function is then responsible for making sure that the necessary initialization is performed.
First, the supported hardware needs to be found. This is usually done by searching for a specific Vendor and Device ID (as specified in the configuration header of the PCI/PCI Express device) amongst the devices on the PCI/PCI Express bus.
If one or more devices are found, these need to be initialized, and their resources (BARs) need to be mapped to the memory range of the host OS.
The third step is to tell the host OS about the functionality this driver provides to other applications, so that these can access the driver. This is typically done through a special structure, in which a function pointer is stored for each kind of request the driver can handle (for instance, open, close, read, write, device control, etc). On Windows this is called the “dispatch table”, while the Linux equivalent is called a “file operations structure” (or fops).
After registering its functionality the driver waits until called upon by an application, and then performs whatever was requested of it.
Drivers can implement functionality for all the major I/O request types. In this particular case however, only one is actually needed. The request type is called *DeviceIOControl* in Windows and *ioctl* in Linux. It sends an I/O control code and a data-pointer from an application to the device driver. The request function in the device driver then typically does a switch-case on the control code, and can, using the data-pointer for both input and output, perform a large range of various functions.

4 FPGA Design

For testing the drivers, the *IOControlDemo* FPGA design is used (see journal 2). This implements two 8 kB memory-type BAR modules. BAR0 is used to access the onboard leds and buttons, and provides four registers mapped as follows:

- *Register 0 (0x00):* Retrieves and sets the state of the single user-led (bit 0)
- *Register 1 (0x04):* Retrieves and sets the state of the eight leds (bit 7-0)
- *Register 6 (0x18):* Retrieves the state of the DIP-switches (bit 3-0)
- *Register 9 (0x24):* Retrieves the state of the user pushbuttons (bit 1-0)

BAR1 just provides an 8 kB blockram with read and write access.

5 Windows driver

The existing Windows driver from the demo comes with most of the functionality that is needed. It consists of a number of files, of which the most important are:

- *s3_1000.c:* Includes the *DriverEntry()* function that is run at load, and the *Control()* function that handles *DeviceIOControl* requests. It is in this function that the new memory read and write functions will need to be added.
- *s3_1000.h:* Describes special data structures used by the device driver.
- *pnp.c:* Contains various functions that are used to initialize the board, and extract and setup its resources so they can be used by the driver.
- *ioctrl.h:* This file holds the definition of the I/O control codes that can be used with *DeviceIOControl*. This file needs to be included by any application that wants to use *DeviceIOControl* requests with the device driver. Entries for the new memory read and write functions will need to be added in here. The unique GUID of the driver, which can later be used to identify it, is also specified.

The driver is found in "*Source\Windows_Device_Drivers_and_Applications\IOControl\Driver*" on CD1.

5.1 Adding new control codes

Two new functions are needed, one that writes a single DWORD register, and one that reads a single DWORD register. First, control-codes for these are created in *ioctrl.h*. This is done using the following form:

```
#define MDS_IOCTL_Device_Function CTL_CODE(DeviceType, Function, Method, Access)
```

The parameters in italics are:

- *Device_Function:* The name of the control code, which should describe the functionality it is assigned to. The following will be used: *READ_REGISTER* and *WRITE_REGISTER.*
- *DeviceType:* Describes the type of device. The Spartan3 board is initialized as a *FILE_DEVICE_UNKNOWN*, so this will also be used here.
- *Function:* A unique integer index, used to distinguish different control codes from each other.
- *Method:* Used to describe the method used for passing data between the application and the device driver. Here, buffered I/O will be used (data is copied between user-space and kernel-space, as opposed to direct I/O where data is transferred directly between user-space memory and device memory – direct I/O can be faster for large transfers, but takes extra code to setup). This is indicated with *METHOD_BUFFERED*.
- *Access:* Used to describe access-rights (read-only / write-only). There is no need to restrict this in any way for the test application, so *FILE_ANY_ACCESS* is used.

A thorough specification on defining control codes can be found at MSDN[1].

[1] "Defining I/O Control Codes" at http://msdn2.microsoft.com/en-us/library/ms795909.aspx

Using these, the two lines that are added to *ioctrl.h* are:

```
#define MDS_IOCTL_READ_REGISTER CTL_CODE(  FILE_DEVICE_UNKNOWN,   \
                                           0x010,                 \
                                           METHOD_BUFFERED,       \
                                           FILE_ANY_ACCESS)

#define MDS_IOCTL_WRITE_REGISTER CTL_CODE( FILE_DEVICE_UNKNOWN,   \
                                           0x011,                 \
                                           METHOD_BUFFERED,       \
                                           FILE_ANY_ACCESS)
```

5.2 Adding new functions

The new functions then need to be added in the *Control()* function in *s3_1000.c*. This function includes a switch-case statement, with a case for all the control codes in *ioctrl.h*. The new functions should simply be added as cases in this statement. The code used for the two functions is:

```
case MDS_IOCTL_READ_REGISTER:
   address = (ULONG) deviceExtension->MemoryStart[pBuffer[0]] + (pBuffer[1]<<2);
   length = 1;
   KdPrint((“\nRead register at: %x\n”, address));
   pBuffer[0] = READ_REGISTER_ULONG((PULONG) address);
   break;

case MDS_IOCTL_WRITE_REGISTER:
   address = (ULONG) deviceExtension->MemoryStart[pBuffer[0]] + (pBuffer[1]<<2);
   length = 0;
   KdPrint((“\nWrite register at: %x\n”, address));
   WRITE_REGISTER_ULONG((PULONG) address, (ULONG) pBuffer[2]);
   break;
```

In the above code, there a couple of important concepts:

- *pBuffer:* A pointer to the I/O buffer of the device driver. Input from the calling application is received from here, and output is written to it also. The number of output bytes to write back to the stack is set using *length*.
- *deviceExtension->MemoryStart[<index>]:* The base address of a specified BAR as seen by the kernel. To obtain the address of a specific register, the index of the BAR is passed through *pBuffer[0]*, and the offset of the register is passed through *pBuffer[1]*. The offset is shifted left two places (as it is a DWORD offset) and added to the base address.
- *KdPrint:* Prints a message that can be read with a kernel debugger. This is not necessary for the functionality, but nonetheless nice to have for debugging purposes.

5.3 Assigning a GUID

To make sure that the driver is only used with applications that are intended for it, a new globally unique identifier (GUID) should be assigned in *ioctrl.h*. The GUID can then be used by applications to obtain a handle to this particular driver. GUIDs can be created using the *GUIDgen*[1] tool.

5.4 Building and installing the driver

To be able to build the driver, it is necessary to install the Windows Driver Kit (WDK). Once installed, it is possible to start various build environments, depending on the target platform of the driver (*Start -> Programs -> Windows Driver Kits*). Each platform has a ‘Checked’ and a ‘Free’ environment, corresponding to debug and release typically found in other build tools. A ‘checked’

[1] GUIDgen can be found in the *Tools* folder on CD1.

build will include all debug output and similar, while these will be stripped from 'free' builds, producing a smaller and faster driver.
Here, the Windows XP x86 Checked build environment is used. The demo device driver comes with a usable make-file. The only thing that is apparently missing is the file *afxres.h*, which can be copied to the driver folder from the *\inc\mfc42* folder in the WDK installation folder.
As the sample driver comes with the necessary makefile, the build process can be started by simply issuing the command *build* in the driver folder from the build environment. The driver file is called *s3_1000.sys*, and is located in the subfolder *\objchk_wxp_x86\i386*.
To install the driver, a driver installation information file (.inf) is needed. The demo application comes with a usable file (*oemsetup.inf*) that just needs to be placed in the same folder as the driver file. The driver can now be installed using the standard Windows driver installation features (*Do not connect to Windows Update -> Install from a list or specific location -> Do not search for driver -> Have disk* and then point at *oemsetup.inf*). For the project, new .inf files have been made for each driver (both for checked and free builds).

5.5 Debugging

Debugging kernel mode drivers is not an easy task, as an error more often than not will lead to a system crash. The best way is to use two systems, one for testing the driver, and another connected to the test system which runs the debugger. This is not always possible though, and usable results can also be obtained with only one machine. Microsoft has created a package called *Debugging Tools for Windows*[1], which can be used both with one or two systems. The package includes the application *WinDbg*, which provides access to the kernel console. To use it locally, Windows needs to be booted with the */debug* parameter, which should be added to the line that boots the used Windows installation in *c:\boot.ini*. An example *boot.ini* could be:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="WinXP" /fastdetect /debug
```

After booting with this parameter, debugging is initiated by starting *WinDbg* and choosing *File -> Kernel Debug -> Local*. This will open the kernel console, which can be refreshed by using the command *!dbgprint*. As the debugger is run locally, many of the more advanced functions are not accessible, but just having access to the kernel console to read messages from the driver is still quite helpful.

6 Windows Application

To use the device driver, a console application will be built. The application and its sourcecode can be found in "*Source\Windows_Device_Drivers_and_Applications\IOControl\Application*" on CD1. The application will need to locate the device driver to use, open it, and be able to send requests to it. To accomplish this, the following functions are used:

- *SetupDiGetClassDevs():* This command is used to retrieve a handle to a device information structure, that holds information for a set of devices matching the parameters. Here, the *ClassGuid* parameter is used to identify the driver, using the same GUID as specified in *s3_1000.h* in the device driver.
  More info: http://msdn2.microsoft.com/en-us/library/ms792959.aspx.
- *SetupDiEnumDeviceInterfaces():* This is used to retrieve and identify a specific, indexed device interface in the aforementioned set. In this case, the first device in the set is retrieved.
  More info: http://msdn2.microsoft.com/en-us/library/ms791242.aspx.

[1] The Debugging package can be found in the *Tools* folder on CD1.

- *SetupDiGetDeviceInterfaceDetail():* Retrieves the details of the selected device interface. This is actually called twice. The first time to retrieve the size of the detail structure, so that a properly-sized buffer can be allocated. The second call then actually retrieves the details, and stores them in the allocated buffer.
  More info here: http://msdn2.microsoft.com/en-us/library/ms792901.aspx.
- *CreateFile():* This function creates a handle to and opens the device. The device to be used is determined from the device-name in the retrieved details-structure.
  More info: http://msdn2.microsoft.com/en-us/library/aa363858.aspx.
- *DeviceIoControl():* Sends a request to the device driver. The handle to the opened device, the control code of the desired function, and various pointers and variables concerning in- and output data are used as parameters.
  More info: http://msdn2.microsoft.com/en-us/library/aa363858.aspx.

The application is now built using MingW, by choosing *Execute -> Compile* from within Dev C++. The *ioctrl.h* file from the driver is included to gain access to the used GUID and the defined control-codes. The application needs to be linked with the setupapi library, which is done by passing the flag *-lsetupapi* to the linker, and making sure that such a file is in the path. In Dev C++ this can be done by choosing *Tools -> Compiler Options* and adding *–lsetupapi* in the *Add these commands to the linker command line* box.

7 Test of Windows device driver and application

The functionality of the driver and application now needs to be tested. This is done with calls to *DeviceIoControl()* using the two new functions to read and write registers. It is chosen to write to register 1, which controls the led-bank, and to read register 9, which holds the state of the user pushbuttons. It is expected that the write will set the leds to a chosen pattern, and that the read will return 0, 1, 2 or 3, depending on whether the pushbuttons are activated or not (the pushbuttons are active-low).
Running the application produces the expected results. Writing the pattern 0x55 (0b01010101) to the led-bank results in every other led being turned on, and the rest off. Reading correctly returns the value 3 when no buttons are pressed, 0 when both are, and 1 or 2 when only one of the buttons is pressed.

8 Linux driver

The Linux device driver needs to be written from scratch. This is done in accordance with the book *Linux Device Drivers*[1] (LDD). The Linux driver will be quite a bit simpler than the Windows driver, as it will only include what is absolutely necessary to initialize the card and read and write registers (as opposed to the Windows driver, which is able to handle numerous additional events such as card-removal and similar).
The driver will be created as a module, and will need three functions, one to handle initialization, one to handle cleanup, and one to handle requests. These functions will be put in the file *s3pcie.c*. Additionally, two control-codes need to be defined, which will be done in *s3pcie.h*.
The driver is found in "*Source\Linux_Device_Drivers_and_Applications\IOControl\Driver*" on CD1.

[1] See the literature list. The book can be freely distributed, and is included in the folder "*Linux Device Drivers, Third Edition*" on CD1.

8.1 Defining control codes

Two control codes need to be defined, one for read and for write. Linux control codes are defined using one of the following forms:

```
#define CODE_NAME _IO(type, number)          //No parameters/return values
#define CODE_NAME _IOR(type, number, size)   //Reads data from driver
#define CODE_NAME _IOW(type, number, size)   //Writes data to driver
#define CODE_NAME _IOWR(type, number, size)  //Bidirectional transfer
```

Type is an 8 bit long so-called “magic” number, which is preferably a unique number assigned to only this device driver. The magic number does not *need* to be unique, but by using different numbers for all device drivers, the risk is lowered that a request by accident is sent to and handled by another driver than the intended receiver. A list of existing and already assigned magic numbers can be found in the file *ioctl-number.txt* in the documentation folder of the Linux installation. Looking through this list, the magic number 0xA5 is chosen.
Number is an index of the control code, and can just be sequential numbers.
The *size* field can be used when a single value needs to be passed between application and driver. This field is not mandatory, but can ease the validity-checking of passed arguments as the control code will then include information about the desired size of the argument. If structures or arrays are required, the field can be ignored.
Using this, the control codes in *s3pcie.h* are defined as:

```
#define S3PCIE_IOC_MAGIC       0xA5

#define S3PCIE_IOREAD_DWORD    _IOWR(S3PCIE_IOC_MAGIC, 1, int)
#define S3PCIE_IOWRITE_DWORD   _IOW(S3PCIE_IOC_MAGIC, 2, int)

#define S3PCIE_IOC_MAXNR       2
```

Notice that the read control code is defined as _IOWR, as the address is passed from the application to the driver, and the data read is then passed back. The *size* field is not used, as pointers will be transferred between application and driver. The last define is used in the driver for a quick check to see if a received control code is valid.

8.2 Includes and defines

The driver needs to include a number of header files:

- *<linux/init.h>:* Provides functionality for specifying which functions to use for initialization and cleanup.
- *<linux/module.h>:* Contains various useful definitions and symbols.
- *<linux/pci.h>:* Provides functionality for accessing PCI (and PCI Express) devices.
- *<linux/cdev.h>:* Contains functions used for registering functionality with the kernel.
- *<asm/uaccess.h>:* Contains functions used for checking the validity of memory locations passed from user-space.
- *“s3pcie.h”:* Contains a specification of the control codes used.

The Linux kernel keeps track of the license under which various modules are distributed. Using a proprietary module will set a “tainted” flag in the kernel. The kernel will continue to work just fine, but in some cases Linux developers will be less likely to help users with tainted kernels, as it is not possible to look through and debug the source code of all the loaded modules.
To set the license for a module, the following macro is used:

```
MODULE_LICENSE(“GPL”);
```

As the driver is not going to be distributed in its current state, the choice of license does not matter much. In this case the GPL license is used for the sake of simplicity.
The next thing to do is to define the devices that this driver will work with in a device table. This is done using the Vendor and Device ids as specified in journal 2:

```
#define VENDOR 0x1597
#define DEVICE_ID 0x0301

static const struct pci_device_id s3pcie_ids[] =
{
   {PCI_DEVICE(VENDOR, DEVICE_ID)},
   {0,},
};
```

As with the Windows driver, information about which functions to call for various requests is needed. This is specified in a file operations structure. As only *ioctl* requests will be used, the following is enough:

```
static int s3pcie_ioctl(  struct inode* inode,
                          struct file *filp,
                          unsigned int cmd,
                          unsigned long arg);

static struct file_operations s3pcie_fops =
{
   .owner = THIS_MODULE,
   .ioctl = s3pcie_ioctl,
};
```

Additionally, variables for storing the handle to the opened device, addresses of the BARs, the used device number, and a kernel structure representing the device need to be declared:

```
static struct pci_dev *dev;
static void* bar_addr[6];
static dev_t first;
static struct cdev* s3pcie_cdev;
```

The last thing to do is to tell the driver about the initialization and cleanup functions, *s3pcie_init()* and *s3pcie_exit()*. This is done with the following two lines, placed at the end of the file:

```
module_init(s3pcie_init);
module_exit(s3pcie_exit);
```

8.3 The initialization function

The function to use for initializing the board when the driver module is loaded is *s3pcie_init()*. It is declared using:

```
static int __init s3pcie_init(void)
```

The __init statement tells the kernel that the function is only used during initialization and can thus be removed to conserve memory once it has completed. The first thing to do in this function is to tell the kernel about the device table defined for this driver:

```
MODULE_DEVICE_TABLE(pci, s3pcie_ids);
```

In Linux, devices are assigned major and minor device numbers. These need to be requested, which is done with the following function:

```
alloc_chrdev_region(&first, 0, 1, “s3pcie”);
```

The first allocated device number is returned in the *first* structure, the first minor number is requested to be *0*, only *1* number is needed, and the device should be called *s3pcie*. With this done, the appropriate device needs to be found and initialized:

```
dev = pci_get_device(VENDOR, DEVICE_ID, NULL);

if(dev)
{
  int err;
  err = pci_enable_device(dev);

  if(!err)
  {
    int n;

    for(int n=0; n<6; n++)
    {
      bar_addr[n]= ioremap(pci_resource_start(dev,0), pci_resource_len(dev,0));
    }

    s3pcie_cdev = cdev_alloc();
    cdev_init(s3pcie_cdev, &s3pcie_fops);
    s3pcie_cdev->owner = THIS_MODULE;

    cdev_add(s3pcie_cdev, first, 1);
  }
}
```

The first call retrieves the first device that matches the specified *VENDOR* and *DEVICE_ID* (in this case there is no reason to support more than one card). If a device was found (*dev* differs from 0), the device is enabled. If everything goes well, the BARs are mapped to kernel memory, and their new addresses are stored in the proper variables. The last four lines allocate and set up a cdev-structure, which registers the drivers functionality with the kernel (notice that the *s3pcie_fops* structure is passed along). After the last call, the driver is “live” and can be called by applications.

8.4 The ioctl function

The *ioctl* function is called whenever an *ioctl* request is received by the driver. Recall that the function is declared like this:

```
static int s3pcie_ioctl(  struct inode* inode,
                          struct file *filp,
                          unsigned int cmd,
                          unsigned long arg);
```

The *inode* and *filp* parameters correspond to the file descriptor opened by the user-space application. The *cmd* parameter holds the index of the control code (*number*) for the request, and the (optional) *arg* parameter can be whatever the user mode application wants to pass to the driver (here it is a pointer to a memory array).

The first thing to do is to check if the control code is valid for this driver:

```
if(_IOC_TYPE(cmd) != S3PCIE_IOC_MAGIC) return –ENOTTY;
if(_IOC_NR(cmd) > S3PCIE_IOC_MAXNR) return –ENOTTY;
```

This checks both if the “magic” number matches that of the driver, and also if the command number is defined. After having validated the command, the actual functionality of the ioctl function follows:

```
switch(cmd)
{
case S3PCIE_IOREAD_DWORD:
   get_user(stack[0], (int __user *)arg);
   get_user(stack[1], (int __user *)(arg+sizeof(int)));
   stack[2] = ioread32(bar_addr[stack[0]] + (stack[1]<<2));
   put_user(stack[2], (int __user *)arg);
   break;
case S3PCIE_IOWRITE_DWORD:
   get_user(stack[0], (int __user *)arg);
   get_user(stack[1], (int __user *)(arg+sizeof(int)));
   get_user(stack[2], (int __user *)(arg+2*sizeof(int)));
   iowrite32(stack[2], bar_addr[stack[0]] + (stack[1]<<2));
   break;
default:
   return –ENOTTY;
}
```

For both cases the index of the target BAR is passed as the first element in the *arg* buffer, and the address as the second element. For *S3PCIE_IOWRITE_DWORD*, the data to write is passed as the third element. The *get_user()* and *put_user()* functions are used to move data between user-space and kernel-space. The *ioread32()* and *iowrite32()* are very similar to the commands used in the Windows driver, and simply read or write a 32 bit register.

8.5 The cleanup function

When the driver is unloaded a few things need to be cleaned up, which is done in *s3pcie_exit()*:

```
cdev_del(s3pcie_cdev);
pci_disable_device(dev);
pci_dev_put(dev);
unregister_chrdev_region(first, 1);
```

First, the functionality of the driver is unregistered from the kernel, to keep applications from calling it. Next, the PCI device is disabled, and the kernel is told that the device is no longer in use. Finally, the assigned device numbers are unregistered.

8.6 Building the driver

To build the driver, a full kernel source tree and a makefile is needed. The source tree is installed as a part of the full Slackware 12 installation, and in the case of building this module, a very simple makefile is enough:

```
obj-m := s3pcie.o
```

This needs to be invoked from the root of the kernel source folder (*/usr/src/linux*), which can be done like this:

```
make –C /usr/src/linux M=`pwd`
```

As this gets a little tedious to write every time the driver needs to be rebuilt, a makefile that handles this is used instead[1]:

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
   obj-m := s3pcie.o

# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else
   KERNELDIR ?= /lib/modules/$(shell uname -r)/build
   PWD := $(shell pwd)
default:
   $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

Using this makefile, it is enough to simply run *make* to build the driver.

8.7 Loading the driver

To load the driver, the *insmod* command is used. However, for applications to be able to access the functionality, a device node in the */dev/* folder needs to be created also. This device is created using the following command:

```
mknod /dev/Node_name c Major Minor
```

Node_name is the name of the node, c specifies a character device, and *Major* and *Minor* is the device number assigned to the device driver by the kernel, when the driver is loaded. The *Minor* number is just set to 0 (as done in the driver), but as the *Major* number is assigned dynamically, it needs to be retrieved from */proc/devices* where all active devices in the system are listed. This is done using the *awk* command to search for the name of the module, and then return the number listed next to it.
All this can be combined into a single script (*load*) that loads the driver, and creates a corresponding device node[2]:

```
#!/bin/sh
module="s3pcie"
device="s3pcie"

/sbin/insmod ./$module.ko $* || exit 1

rm –f /dev/$device

major=$(awk "\$2==\"$module\" {print \$1}" /proc/devices)

mknod /dev/$device c $major 0
```

The *rm* command is used to remove old nodes (should there be any) before creating the new one.

[1] The make file is adapted from LDD, chapter 2, page 24.
[2] The *load* and *unload* scripts are adapted from LDD, chapter 3, page 47.

A similar script (*unload*) is used for unloading the module:

```
#!/bin/sh
module="s3pcie"
device="s3pcie"

/sbin/rmmod ./$module.ko $* || exit 1

rm –f /dev/$device
```

After running the load script, the device and driver are ready for use.

8.8 Debugging

The easiest way of debugging drivers in Linux is to use the *printk()* function to print a message to the system log. This can then be read directly from the Linux console using the *dmesg* console call.

9 Linux application

The application is very simple, as the device is opened with a single call to open:

```
int dev;
dev = open("/dev/s3pcie", NULL);
```

The string passed to open is simply the name of the device node, created by the load script. To communicate with the driver, the *ioctl()* command is used, which has the following syntax from user mode applications:

```
ioctl(int fd, unsigned long cmd, ...);
```

The *fd* argument is the handle returned by open, the *cmd* argument is the control code (as specified in *s3pcie.h*), and any additional arguments can then be passed also. In this case, this will be a pointer to an integer.
The application is build using GCC, with the following command:

```
g++ iocontrol.cpp
```

10 Test of Linux device driver and application

To test the functionality of the device driver and application, it is attempted to write the led bank and read the pushbuttons on the Spartan3 board. This is done by first writing an 8 bit pattern (0x55 is chosen, as it should produce alternating on and off leds) to register 1, and then reading register 9 and printing the value to the console.
The test is successful, as the correct pattern is seen on the leds, and as the result of reading the state of the pushbuttons is correctly 3 when no buttons are pressed, 0 when both are pressed, and either 1 or 2 when only one button is pressed.

11 Conclusion

Drivers and simple applications for both Windows XP and Linux 2.6 have now been created. The functionality is similar on both platforms. Each provides a way of writing or reading a single DWORD register on the board. This can be done from a user mode application through very similar calls, *DeviceIoControl()* on Windows, and *ioctl()* on Linux.
This has been tested, and the functionality works on both platforms.