Diploma Thesis
Implementation of a block device driver for Compactflash disks (Linux x86 architecture)

Kapelonis Kostis A.M.1143
kapelon@csd.uoc.gr
Computer Science Department
University of Crete

September 2, 2003
Herakleion
Crete

Abstract

This document describes the development of a software driver.The driver controls an adapter which can access CompactFlash cards.The adapter is attached to an embedded system (the Dil/NetPC board) featuring a x86 processor.The operating system used is the Linux kernel.The text focuses both on hardware and software.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Dil/NetPC DNP/1486-3V board from SSV Systems (http://www.ssv-embedded.de/) is a powerful, low consumption, x86 platform that can be used as an embedded solution for various projects, although it is clearly developed for networking with Ethernet networks.

It is based on the SC410 processor from AMD (486 compatible), packs 8MB RAM, and features an Ethernet port allowing it to connect to common local area networks.It has also other advanced features like a watchdog timer that can be used for maximum reliability.

Some key features are:

- AMD 32-bit SC410-33 Low Power CPU (486/33MHz)
- 8 MB DRAM Memory
- 2 MB FLASH Memory with Boot Block
- 10BASE-T Ethernet Interface
- COM1 Serial Port with TTL Levels, 16550 compatible
- 20-Bit General Purpose Parallel I/O
- I/O Extension Bus with programmable Interrupts
- 64-pin JEDEC DIL-64 Connector with 2.54mm Centers

1.1 The problem

Development for the Dil/NetPC entails several limitations for programmers accustomed to conventional desktop programming.Apart from the fact that code must be first downloaded to the board , the major shortcoming of the system is the limited flash memory storage.There is no visual output display so debugging must be performed through the serial port,via the network or even with specialized programs (e.g. gdb server/client).

The flash chip on the board is used as a *hard disk* for the system.It contains the bootable image of the OS which is then decompressed into RAM memory in order for the system to function.In the case of Linux, the compressed image[1] accounts for at least 1.7MB[2].This leaves about 300kb on the the flash for programs written by the developer.

This might seem enough space for some small test programs.However,as soon as the programmer starts writing applications that involve saving data on the “disk” this space is inefficient.For example a monitoring program which stores large text logs that follow the actions of the system as time progresses, needs moderate to huge amounts of disk space.

S.S.V. systems sells a high end version of the DNP/1486-3V called the ADNP/1486-3V[3].This board is the enhanced version of the low end model, featuring not only additional hardware components, but also double RAM and FLASH memory chips .This means that the available flash memory is now 4MB and since the OS data are the same there is more room for custom applications.

However this doesn’t solve the space requirements for log keeping.Buying a new board and disposing the old one is not an easy option too.It would be ineffective to buy a new board only for the flash memory, since we though that the other components of the low end board were good enough(the system RAM and the CPU).Another solution should be found and it should be somehow an extension of the DNP/1486-3V.

1.2 The proposed solution

It was decided to attach an external memory storage component on the system that would be used for data and custom programs only.The main operating system would still boot from the on-board FLASH chip.Ideally the space provided, should be easily upgraded allowing for the ever-increasing memory requirements of the future.

Thus,some kind of “adapter” would be attached on the board which would interface with common consumer memory chips (smartmedia,compactflash,memory sticks e.t.c).Therefore,increasing the capacity of the external memory would be as simple as removing the old memory component and replacing it, with one having larger capacity.

In this text we document the development of a device driver used by the Dil/NetPC to interface with such an adapter.The driver is necessary for the operating system to detect and natively use the available space.The programmer would then access this extra space using the filesystem layer of the operating system.

1.3 Intended audience

Readers of this text are expected to be competent in the C language and user-space application programming for the GNU/Linux systems.Some basic knowledge of hardware terms like low/high signal,bus, address/data bits is required.Knowing how to compile and upgrade Linux kernels and modules is essential too.

1.4 Contributions

During development we implemented:

[1]minimal filesystem + essential utilities compressed with gzip -9

[2]including a generic Linux kernel

[3]ADNP is advanced DNP

1. A stable character driver for the Dil/NetPC.Reading it, returns the status of the built-in dip-switch and writing it, changes the status of the on-board LEDS.
2. A stable virtual block driver which controls 64kbs of memory.A filesystem can be created on it and then mounted.It acts essentially as a ram-disk.
3. An unstable block driver which controls Compactflash cards.Data corruption is minimal, but evident though.

1.5 Organization of this report

This document describes the development process in a chronological way.Each chapter is based on the previous one, so they should be read preferably in order.

Chapter 1 is the introduction you are now reading. Chapter 2 is a list of hardware and software terms the reader should be familiar with.Chapter 3 describes the reasons we selected Compactflash cards instead of the other popular FLASH format,Smartmedia cards.Chapter 4 explains in detail the preparation of the Dil/NetPC board, before the actual development can start.This has nothing to do with the driver itself,but it is essential for the development process.

Chapter 5 is theoretical.It provides a high-level view of the Linux kernel and software modules which represent device drivers.Chapter 6 describes some technical details for the compilation of Linux kernel modules.The example module is a character driver for the Dil/NetPC.Chapter 7 introduces the generic Linux kernel interface for writing block device drivers.The example here is a ram-disk.Chapter 8 which consists the gist of the document, describes the actual development of the real driver, which controls Compactflash cards.It ties together the ATA/IDE protocol,the LBA addressing mode, and the chip select concept.The final chapter (9) focuses on testing and troubleshooting for this driver.

Chapter 2

Definitions

The reader of this text is expected to be familiar with the following terms.The terms are referenced many times inside the document so they are considered important for the full understanding of the process.

driver The final code this text describes is commonly called a *driver*.A driver is a special program that enables the operating system to interface with a specific hardware device.

Figure 2.1 shows the components of a system in hierarchical way.Drivers form a layer between the operating system and the devices.They are essentially the “border” between hardware and software.

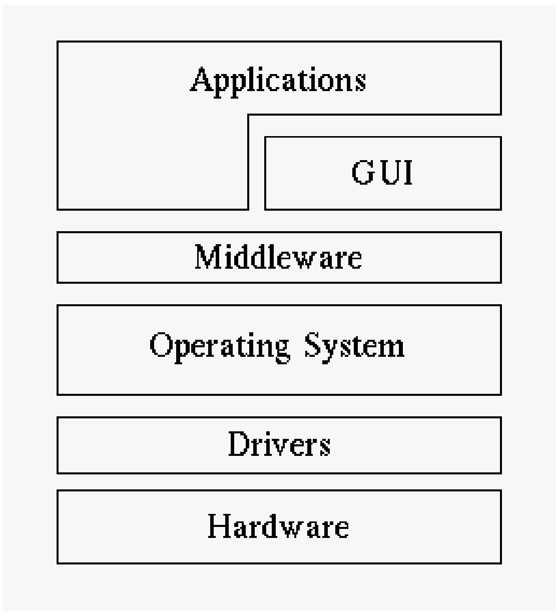

Figure 2.1: The different layers of a system

Most hardware vendors ship their products with drivers, so that customers can take advantage of new devices even if their operating systems didn’t support these devices in the first place.Since this is a custom solution, there is no driver for this purpose.Attaching the adapter onto the Dil/NetPC is the first part of the work,and writing the driver is the second.

smartmedia *Smartmedia* cards are memory storage cards developed by Toshiba.They are thin and small with capacities up to 128MB.They are used primary in cameras and PDAs as storage.They don't feature a micro-controller so low level programming[1] is required for the interface.

The forum responsible for the specifications resides at http://www.ssfdc.or.jp/english/. The original name of Smartmedia card was solid-state floppy disk card, or SSFDC.

compactflash This is another type of memory cards.*Compactflash* technology was invented in 1994 by Sandisk.It became very popular as a method to store data for cameras and PDAs.

Unlike Smartmedia cards,Compactflash cards include a micro-controller which performs many of the low level input/output routines,and so programming for them is easier.They also support the ATA/IDE protocol used for hard disks so they can replace easily a hard disk where reliability is important(they don't contain any movable parts since they use flash memory too).

They also come in greater capacities than Smartmedia.At the time of writing, cards with 512MB or even a GB are becoming popular among consumers.

sector Measurement unit used mainly for hard disks, but also for other devices that transfer data in large chunks.A *sector* is the smallest addressable amount of data.Almost always,a sector holds 512 bytes of data.All data transfered to/from a hard-disk are just a series of sectors eventually.

block A *block* device is a device which stores data in fixed-size chunks of data called blocks.Hard disks are the dominant block devices.Blocks can be any number of even sectors.There are blocks of 1024 bytes(2 sectors),4096 bytes(8 sectors) and so on.Blocks do not have any physical meaning.They are used to group sectors.A critical property of a file-system is the block size used.

C/H/S The physical geometry of a hard-disk or an IDE/ATA compatible device(like a compactflash).Three numbers that stand for cylinders,heads and sectors respectively.A sector is characterized by these 3 numbers uniquely.The capacity of a hard disk is determined by these numbers.This geometry isn't physical on compactflash disks since they emulate the geometry of a hard-disk and use these numbers only for backwards compatibility.

LBA An alternative method to address a sector.*LBA* stands for logical block addressing.All sectors of the device are considered sequential and are assigned a unique number which can be used for identification.The process which converts the C/H/S numbers to LBA is called LBA translation or LBA mapping.

The LBA method was invented in order to overcome some limitations of the C/H/S addressing scheme,and to allow for a unified way to access sectors.

register A *register* is a small set of data/control bits.Writing to registers of a device allows the programmer to transfer data to the device or give it commands,while

[1]dealing with signals

reading from registers results in querying the device or transferring data from it.Common registers have 8,16 or 32 bits width, but larger sizes like 64 bit have already appeared.

signal A *signal* can have many meanings in science.In this context we mean the voltage level of a single pin/bus bit.A signal is either low or high.Asserting a signal means activating it,and depending on the situation it can mean setting it high or low.We can measure a signal by attaching a probe on the cable that carries it or on a pin where it terminates.Laboratory equipment such as oscilloscopes and logic analyzers are used for this purpose or even multi-meters for simple situations.

ATA/IDE *IDE* stands for Integrated Drive Electronics.It is an interface specification used mainly by hard disk drives.It defines a method to communicate with a device that supports it,through a bus/channel.There are several revisions and enhanced versions (like EIDE for Enhanced ide) which specify greater speeds of data transfer and/or power saving options.

ATA which stands for Advanced Technology Attachment is the name the ANSI group uses for the same technology.The forum that maintains the specifications resides at http://www.t13.org/.

A Compactflash disk supports the ATA/IDE protocol.This means that compactflash disks can accept the ATA/IDE commands and respond as if they were hard disks.

Chapter 3

The compactflash solution

Originally, Smartmedia cards would be used as a storage component.After considering the disadvantages of this technology,and comparing it to Compactflash ,we decided to begin development on the latter.

3.1 Smartmedia versus Compactflash

Both Smartmedia and Compactflash cards use FLASH memory.FLASH memory has some important advantages such as high reliability and no loss of data during power-offs.There are no moving parts like hard disks,making them less error-proven and more power friendly.Both cards are also used with cameras and other consumer media devices, which means that have been accepted by vendors and consumers.

Two were the main reasons for selecting Compactflash over Smartmedia.

- Smartmedia cards have a maximum capacity of 128MB.Compactflash cards on the other hand,support much larger capacities.Cards with 1GB capacity have recently appeared.This gives the Dil/NetPC board many capabilities.For example it could be used to store logs 24/7/365,a function which requires vast amounts of storage space.128 MB of space might seem a small limitation today,but tomorrow it could be a major drawback.

- Smartmedia card are “raw” memory cells.There is no intelligence built into them.A programmer wishing to interface with a Smartmedia card,has to work on the lowest level.That is, asserting and de-asserting signals,watching for timing constrains and so on.The programmer is also responsible for managing the bad blocks,a difficult and complicated task.Compactflash cards consist of the flash memory where data is stored but also an on-board smart controller.The controller takes care of the above issues.It manages the memory chips[1] so it can offer the programmer a higher level of functions and methods.

  Additionally, Compactflash cards can emulate the common ATA/IDE protocol which is simple and well documented.The programmer only writes commands in the registers of the controller,and then just waits for the results.The actual work of fetching/storing the data to/from the specified address ranges is done by the controller.

[1]even supports ECC in hardware

3.2 The adapter from elektor

To this purpose, we ordered a CompactFlash adapter from the Elektor electronics magazine(http://www.elektor-electronics.co.uk/).Issue No.316 December 2002 describes the procedure to build and use such an adapter.The adapter is attached on a CompactFlash card and "exports" a 32 ATA/IDE pin-out[2].Sample code is supplied too (can be downloaded for free from the website),although it implements the connection between the adapter and a development 8051 board.We would use it for x86 and the Linux kernel,so the code should be *ported* to this platform.

[2]like the one used for floppy disks

Chapter 4

Preparing the environment

The Dil/NetPC kit from SSV includes the board itself,a power cable,a serial cable for the connection with the host computer and a CD-ROM with software and documentation.There is also printed documentation, mainly selected tutorials from the CD.

One could ask for a crossover network cable since the board features an Ethernet port,which could be used for development with a stand-alone host.Otherwise the board should be connected to the local network of the host computer.In either case a TCP/IP(Ethernet) connection should be available between the board and the host computer.

4.1 Host computer configuration

The whole development process will follow the typical host/target idea.Code will be compiled on a conventional computer called host,and then the compiled binaries/objects will be downloaded to the target platform (in this case the Dil/NetPC) via the serial cable.

The host computer is a modern x86 workstation.It should run one of the many GNU/Linux distributions.Most "development" packages should be installed.The kernel sources should be downloaded from www.kernel.org and then installed and configured, preferably in the home directory of a user.Stable version 2.4.17 should be downloaded.Do not download a newer version since it should match the kernel running on the Dil/NetPC board.

The minicom terminal application should be available and also the lrzsz package which supplies command line tools for zmodem/xmodem/ymodem file transfer.Running minicom with -s command switch loads the configuration parameters screen where the settings of the following section should be entered.

The typical GNU tool chain should also be installed.The gcc compiler and make utility are required.The GNU debugger gdb can also be installed for some user-space test applications.

Warning:Even if your distribution comes with a newer gcc3.x branch this should not be used.Instead the previous gcc stable version (2.95) should be downloaded and installed.The 3.x branch compiles broken kernels if their version is lower than 2.4.21 (which applies in our case).Such kernels and/or modules compile successfully and appear to function properly but after a while they segfault and the system throws a kernel panic.

An editor which syntax highlighting capabilities such as vim or Emacs will be of great assistance.The installation of all development documentation of the distribution (man pages/Howtos/faqs/tutorials) is recommended.

During the earliest stages of development the host computer was running Mandrake 8.2.After some time it was upgraded to Debian stable (Woody) but any modern GNU/Linux distribution should fit.

4.2 Preparing the Dil/NetPC board

The Dil/NetPC board has a version of DOS pre-installed.The first step should be the installation of a network-aware Linux system on it.There is excellent documentation on the SSV CD-ROM that comes bundled with the board, in the form of PDF files[1].The procedure is documented there in details,so we will only outline it here.

4.2.1 Installing Linux on the Dil/NetPC

Data is downloaded onto the flash memory (2MB) of the board through the serial cable.One end should be connected on the board itself and the other on a free serial port of the host computer(COM1/COM2 or /dev/ttyS0 and /dev/ttyS1 under Linux terminology).The connection is then initiated with the use of a *terminal* program.Most GNU/Linux distributions ship Minicom for this purpose.Hyperterminal can also be used on Microsoft Windows computers.The appropriate settings are shown in table 4.1

Setting	Value
Serial device	/dev/ttyS0 (or /dev/ttyS1)
Bps	115200
Databits	8
Parity	None
Stopbit	1
Hardware Flow control	None
Software Flow control	None

Table 4.1: Minicom settings

During power-on,booting messages should now be shown into minicom.The boot procedure stops when it reaches the DOS prompt.To prepare the board for data download type:

```
C:\>md␣linux
C:\>cd␣linux
C:\LINUX\>rb
```

The last command (RB=receive bytes) initiates data transfer using the YMODEM protocol.The board is now waiting for data.Pressing `Ctrl-A` and then `S` in minicom should show a file browser.Navigate to the SSV CD-ROM and select all files that reside in `\Dnpx\dossd\Precfg11\` directory (using the space key).Pressing the enter key should start the actual downloading which might take some time.

[1]opened by Acrobat reader from www.adobe.com

This directory contains four files that form a complete Linux operating system.This is the 11th configuration which comes with kernel 2.4.17

start.bat A DOS batch file.It is a simple script used to boot Linux.It runs loadlin.exe with the needed parameters.

loadlin.exe A small executable which can boot Linux over DOS.It needs a Linux kernel and a Linux filesystem image

zimage This file contains a compiled Linux kernel.It is the result of "make zImage" command inside the kernel source tree.

rimage.gz A compressed (with gzip) filesystem image.This contains all the files apart from the kernel of the operating system.Any changes that need to be permanent should be stored here.

Typing now "start" at the command prompt should boot Linux.The kernel messages(the console) should appear in the minicom window.If everything is OK the familiar login prompt will eventually appear.

One more step is required to make Linux launch automatically on boot.Otherwise each time the board boots, the "start" command should be executed from the *linux* directory.Reboot so that DOS loads again and type the following commands.

```
C:\>type␣autoexec.bat

@ECHO OFF
PROMPT $P$G
PATH C:\


C:\>echo␣"CD␣LINUX">>autoexec.bat
C:\>echo␣"START">>autoexec.bat
C:\>type␣autoexec.bat

@ECHO OFF
PROMPT $P$G
PATH C:\
CD LINUX
START
C:\>
```

Autoexec.bat is a batch file(script) that DOS runs on boot.So as soon as DOS finishes booting it will enter the linux directory and execute the start.bat batch file.

4.2.2 Configuring the network on the Dil/NetPC

During the actual development, code will be simply transfered over FTP on the board.There is no need to download it permanently on the FLASH memory.The Dil/NetPC has TCP/IP networking capabilities via the on-board Ethernet port.The fact that the bandwidth it provides is 10Mbps and that it only supports half-duplex transmission,has little impact on the development process.

Setting up the network on Dil/NetPC might seem as easy as running the *ifconfig* command.This is not the case.Since each time Linux loads from the compressed filesystem image,after next boot any network settings will be lost.To solve this, rimage.gz must be accordingly modified so that network setup will be permanent.

Copy the rimage.gz from the `\Dnpx\dossd\Precfg11\` directory, to the hard disk of the host computer.Then type the following commands on the *host computer.*

```
$gunzip␣rimage.gz
$su
#mkdir␣/mnt/image
#mount␣-t␣minix␣rimage␣/mnt/image
```

Now in the /mnt/image directory there is the filesystem of the Dil/NetPC.Enter the etc/config directory there, (that is /mnt/image/etc/config) and edit at least the *ipaddr* file,the *netmask* and the *broadcast* file which contain the IP,netmask and broadcast address respectively.The above values should be given to you by your network administrator.If you are on a local network with no internet access you can use any of the IP ranges shown in table 4.2

Start	End	prefixed form
10.0.0.0	10.255.255.255	10/8 prefix
172.16.0.0	172.31.255.255	172.16/12 prefix
192.168.0.0	192.168.255.255	192.168/16 prefix

Table 4.2: RFC-1918 assigned private network addresses

When finished, the compressed image should be recreated.

```
#cd␣-
#umount␣/mnt/image
#exit
$gzip␣-9␣rimage
$
```

Finally the resulting *rimage.gz* should be downloaded again on the FLASH memory of the Dil/NetPC.The procedure was described in the previous section.

This concludes the software setup of Dil/NetPC.The next section describes the procedure used to attach the CompactFlash adapter on the Dil/NetPC external bus.

4.3 Hardware setup

The I/O expansion bus of the Dil/NetPC allows for the communication with external devices or memory chips.The bus is 8 bits wide and supports 2 chip-select signals (there is a separate chapter for chip selection).This means that up to two I/O devices can communicate with the Dil/NetPC through the bus[2].

Figure 4.3 shows the bus-related pins that were used.For this design we did not use interrupts,although the Dil/NetPC supports up to 5 interrupts[3].

[2]without the use of additional hardware

[3]there are 5 interrupt pins

Pin(s)	Name	Direction	Description
48	CS1	Output	Chip select signal number 1.
51	IOW	Output	The write signal
50	IOR	Output	The read signal
63-56	D0-D7	Input/Output	The actual data bits
55-52	A0-A3	Output	The address bits

Table 4.3: I/O Dil/NetPC pins

The CompactFlash adapter it connected to the I/O bus of the Dil/NetPC with an IDE cable.We obtained an IDE cable from a PC floppy drive.Contrary to the hard drive IDE cables, this type has only 34 pins (2 rows of 17) instead of 40 pins (2 rows of 20).This presents no problem since only the necessary pins would be used.

One end of the cable is directly attached to the adapter and the other must be soldered on the Dil/NetPC.There is plenty of room on the board for this purpose.

Figure 4.1 shows the IDE connector as viewed from above.The IDE cable should be attached on “pins” soldered on the Dil/NetPC board with this orientation.

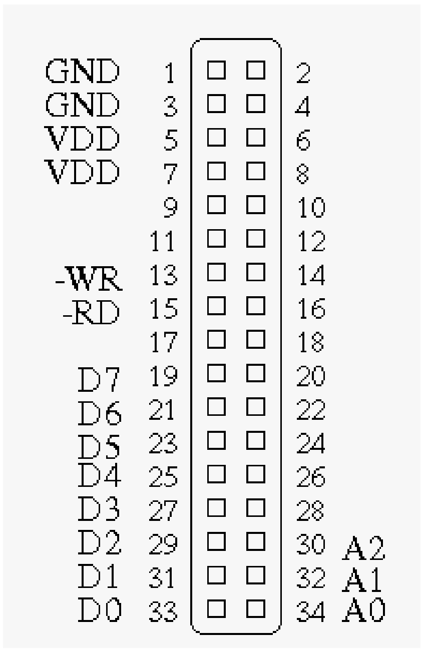

Figure 4.1: IDE pinout (34 pins)

4.3.1 Connecting the Dil/NetPC I/O bus with the IDE cable

Finally, (and this is where the main soldering takes place) the above IDE pins must be connected to the appropriate Dil/NetPC pins.Any loose or wrong connection here will make further development impossible.Figure 4.4 shows which pins of the Dil/NetPC and of the IDE cable should be connected together.Negative names suggest inverted signals.

Dil/NetPC		Directly	IDE cable	
Pin function	Pin number	connected to	Pin number	Pin function
Chip Select 1	48	=	2	-CS1
I/O Channel Ready	49	X		
I/O Read	50	=	15	-RD
I/O Write	51	=	13	-WR
Address bit 3	52	X		
Address bit 2	53	=	30	A2
Address bit 1	54	=	32	A1
Address bit 0	55	=	34	A0
Data bit 7	56	=	19	D7
Data bit 6	57	=	21	D6
Data bit 5	58	=	23	D5
Data bit 4	59	=	25	D4
Data bit 3	60	=	27	D3
Data bit 2	61	=	29	D2
Data bit 1	62	=	31	D1
Data bit 0	63	=	33	D0

Table 4.4: IDE to Dil/NetPC connections

Chapter 5

Conceptual Model

In a perfect world drivers would not exist.Devices would be simply connected to computers and modern operating systems would instantly detect them.While this dream seems to be a reality for some plug-and-play devices,in most cases just plugging the device does nothing.There are many connection methods (serial, parallel, USB1/2, firewire, irda, ATA/IDE, PCMCIA e.t.c),many operating systems and multiple versions/varieties of them, which means that a hardware vendor cannot provide software and support for all possible cases.Instead hardware components include only drivers for popular consumer operating systems,and even then,they must be upgraded from time to time[1].The fact that some hardware vendors don't comply fully with the established standards,or when they comply, they "add" their own proprietary extensions makes drivers a necessity.

In the embedded market, things are no different.Most software produced is part of a custom solution which is developed with reliability and small footprint in mind,but not portability.This is our case too.Although the Dil/NetPC runs Linux, a modern 32-bit operating system with TCP/IP stack , basically it consists of the AMD SC410 CPU, a serial port and an Ethernet port.This forces us to use the raw bus of the CPU itself.The driver we are about to discuss runs on this platform only.

The adapter from the elektor magazine was meant to be used with the 8051 processor.Boards with this kind of processor do not use[2] an operating system.The sample code we had in our hands was very simple.It demonstrated how to read and write a complete sector (512 bytes) from the CompactFlash and nothing more.We needed to adapt and extend this code for the Linux kernel.Thus,the development of the driver focuses on two different areas.The high-level software interface with the Linux kernel which is generic,and the low-level hardware interface to the SC410 bus and the CompactFlash adapter.

Throughout the development process we followed the procedures and practices described in the excellent reference book "Linux device drivers 2nd edition" by Alessandro Rubini and Jonathan Corbet.It is an invaluable source of information since it deals with most technical issues of driver development.What follows, is a brief introduction to Linux modules.The first chapters of the book describes the topic in greater depth and detail.

[1]or when the next version of the OS comes out

[2] there are of course schedulers available

5.1 Linux kernel modules

Drivers for the Linux kernel belong to the so-called "kernel space".This contrasts conventional applications which run in the user space.The kernel space is the part that the kernel itself and it's drivers occupy.Directly below comes the actual hardware the kernel runs on, and directly above comes the user-space where high-level applications reside.User applications can send and receive data to/from kernel-space with *system calls*.System calls are special functions which form the "bridge" between kernel and user space.[3]Figure 5.1 shows the Linux kernel architecture.

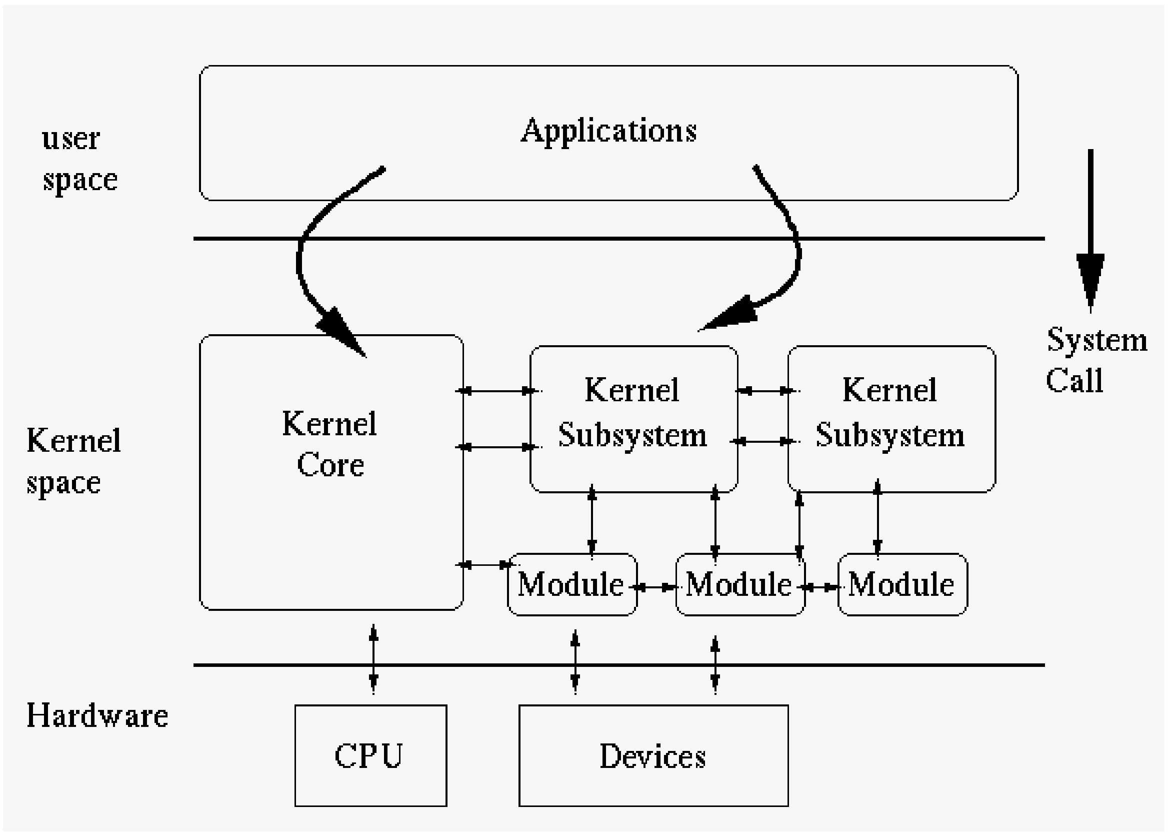

Figure 5.1: Internals of the Linux kernel

Writing a driver for a device means essentially providing a "mapping" between the system calls and the device.What happens when a user application opens the device for reading?What does "read" mean for the device?These questions must be answered by the driver.

Because drivers are parts of the actual running kernel someone would thought that the development cycle would involve the reboot of the system with the new kernel, each time a new version was introduced.This may be the case for the core kernel itself but not for the drivers.The Linux kernel supports loadable modules.These are object files which contain extra code and can be loaded or unloaded during runtime.When loading a module, the kernel gains extra abilities (the functions the module implements).Unloading

[3]Passing from user-space to kernel-space is also triggered by hardware interrupts.

a module when it is no longer needed has the opposite effect.This allows for rapid development cycles with one running kernel.

Loadable modules are conventional object files (.o suffix) which are loaded and unloaded with the *insmod* and *rmmod* command respectively.The *lsmod* command shows current loaded modules.Most Linux drivers are modules,but not all modules are drivers.There are modules which provide pure software functions.Essential and popular modules are included in the kernel distribution files which cover a wide range of hardware devices and software protocols.

There is a well defined programming interface for writing modules.It a set of predefined functions which must be implemented by the programmer.The programmer has a wealth of programming structures and functions to use and extensive debugging capabilities.It is worth noticing however,that the glibc functions the programmer is accustomed with,are not available because modules are part of the kernel space.Every available programming structure is defined in the kernel development headers.For example the common “debugging” function *printf* is not available.The kernel defines it’s own *printk* which redirects it’s output to the system console/logs.

5.2 Types of Linux drivers

Currently there are three types of linux drivers.Drivers for character devices,block devices and network devices.Consequently there are char modules,block modules and network modules.

These names imply the type of device, the driver controls.The devices are categorized into:

char devices Char devices are the most common devices.They are the devices which are accessed in a sequential way.Reading means receiving a stream of characters one after the other, and writing means sending a stream of characters one after the other.Typical char devices are modems,serial ports or terminals.The filesystem representation of a char device is a special character file in the /dev/ directory.Therefore the command

```
echo "ATZ">/dev/ttyS0
```

sends the ATZ string to the modem connected to the first serial port.

block devices Block devices are the devices which transfer data in large chunks.They are the devices which can hold a filesystem and can be accessed with the *mount* command.The dominant block devices are hard disk drives or even CD/DVD drives.They are represented which a special block file in the dev directory.The command

```
dd if=/dev/hda of=boot.sect bs=512 count=1
```

saves the first 512 bytes(the boot sector) of the first hard disk to a file named boot.sect.Block devices are partition-able so there are separate /dev/ files for each partition.Thus */dev/hda1* is the first partition of the first block device, */dev/hdc5* is the fifth partition of the third block device and so on.

net devices Network devices are almost always physical network cards.Naturally, programming for network devices is a bit different from char and block devices.Functions which act when a packet is sent/received via the card,must be implemented.Since there are no explicit read/write system calls for network devices, there is no special /dev/ file for a network driver.The Linux kernel assigns symbolic names to network cards (e.g. eth0,eth1) but these are virtual.Generally, network drivers behave in a special way,unlike the above two categories which are mostly predictable.

The driver we document here is a block driver.

5.3 Hardware access

Any driver of the above categories (which controls a physical hardware device) must at some point communicate with the device it is responsible for.

Accessing the hardware is accomplished by the use of special hardware functions.These functions (also exist in user-space) read or write a single byte to/from a I/O port of the processor address space.The most basic are:

```
unsigned inb(unsigned port);
void outb(unsigned char byte,unsigned port);
```

There are also functions which read/write a word(16bits) instead of a byte or even a long word (32bits) but we didn't use them.Notice that the arguments of *outb()* are in the opposite order than the DOS function with the same name.Much time and effort were wasted until we realized this fact.

The port argument is 16bits in the x86 address space.Usually it is a hex value which maps to a register/memory pointer of the CPU.It can be any value in the range of 0x0-0xFFFF (or 0-65535).

Chapter 6

Programming a module

To familiarize ourselves with module programming, and since the first months of development the hardware wasn't ready (the adapter),we decided to implement a char driver for the Dil/NetPC.

The Dil/NetPC includes 8 Light emitting diodes (or LEDS) which can be used as output and a dip switch which can be used as input of a single byte (8-bit).The idea was that "reading" the device would return the status of the dip-switch(which could be set by just flipping the switches) and "writing" the device would switch on some LEDS according to the input byte.

The full source code of this char driver is included in appendix C.As we found out later,block drivers are more complicated that char drivers.There are however,some common concepts and issues between them.Compiling and running kernel modules can be tricky (not the case with user-space).Our experiences are outlined in the following sections.

6.1 Programming environment

Before we discuss the source code of the module,there are some important programming notes that need to be considered.

6.1.1 Module versions

Linux kernel modules are always compiled against a specific kernel version.Attempting to load a module with different version than the running kernel will make *insmod* abort, with a clear message stating the version mismatch.There are ways to compile modules which can run with multiple kernel versions,or even to force modules to run on a kernel with different version,but these methods are not recommended.

This is why it is imperative that all driver modules should be compiled against kernel 2.4.17, since this is the version Dil/NetPC runs on.Therefore, downloading and unpacking the 2.4.17 kernel source (from ftp.kernel.org or mirrors) is essential for the development process.Contrary to common belief, the sources should be unpacked in some directory other than */usr/src/linux*, since this directory holds the kernel source of the workstation itself.A home directory of a user is the ideal place.Even if the distribution of the workstation is old enough, to claim that it runs 2.4.17, most times the kernel sources it includes are altered and heavily patched[1].So in any case, the pure kernel source should be used.

6.1.2 Compilers and kernel headers

The second issue that needs to be addressed when compiling modules is the compiler.First of all, because the kernel code makes heavy use of GNU extensions,the GNU C compiler (gcc) should be used.A compiler that is only ANSI compliant is not an option.

The real problem however,is the version of gcc.The Linux kernel and gcc are two independent projects which publish new versions as they see fit.There are combinations of kernel and compiler versions,which produce broken kernels or do not compile at all.In our case, gcc version 2.95 was used.Version 2.96 which comes with some versions of the Redhat distributions was *not* tested.And most importantly, gcc versions 3.x are *known* to produce broken kernels prior to 2.4.21.

An important step is pointing the compiler to the kernel source tree that should be used for the compilation process.Thus,when the compiler comes across a line with `#include <linux/fs.h>` for example, it should include the file of the downloaded kernel source tree and not the header file that resides in `/usr/src/linux/include/linux/fs.h` which belongs to the kernel source tree of the workstation.This is accomplished by the *KERNELDIR* directive which defines the location of the kernel headers.It should be passed as an argument to gcc/make or even better, defined in a custom *Makefile*.Appendix C contains a sample Makefile which shows this approach,and also some other required compiler directives for kernel modules.

6.2 Source code description (char driver)

The module is implemented in a single .c file.The code has mainly 4 sections.The first are the functions *init_module()* and *cleanup_module()* which define what happens when the module is loaded and unloaded respectively.These functions are required and together they form the simplest module.Next comes the special structure *file_operations* which defines what are the capabilities of the driver.It contains a pointer to each function the device supports.Functions that have the NULL pointer or are not mentioned at all,are not supported.In our case we only support the "standard" open/release/read/write functions but not poll,seek e.t.c.The third part are the implementations of the above functions.Notice that the prototype of each function is strictly defined by the kernel interface.Finally there are the hardware functions *get_switches()* and *light_led()* which are called from "read" and "write".

6.2.1 Major and minor numbers

One common concept between block and char drivers are *major* and *minor* numbers.A major number is a number from 0 to 255 (8-bit) which is unique for every *type* of device.The minor number is an 8-bit number which is unique for every device present of the *same* type.These numbers are separate for block and char devices.For example, the loopback interface has major block number 7,and the mouse/joystick has major char number 13.If a computer had two joysticks/mice, they would have the same major number(13) but a different minor.

[1]Mandrake for example may call the package kernel-2.4.17-mdk

So a single device is characterized by a combination of major and minor number (if it is the only device of this type,the minor number is usually 0).Currently allocated numbers, for both block and char devices, are listed in `/Documentation/devices.txt` file in the source tree of any Linux kernel.Normally, someone should contact the kernel maintainers and inform them that he/she would need a number for a new driver.This prevents clashes between programmers who write drivers independently at different places of the globe.There are however, major numbers specifically chosen for development and experimental purposes.For our design, we used char major number 254.The whole range 240-254 is assigned as “local/experimental use”[2].Therefore, our driver will use 254 (major) and 0 (minor).

The function *register_chrdev()* attempts to register a major number for use by the driver.This function accepts three arguments and is called from *init_module()* which is the entry point of the module.The first argument is the major number requested from the kernel (254 in our case),the second is the name of the driver (as shown from `cat /proc/devices` command) and the third is a pointer to the structure *file_operations* which was described previously.As expected,inside *cleanup_module()*, which is the exit point of the module, the function *unregister_chrdev()* is called.This releases the major number passed as the first argument.

6.2.2 Indexed access to registers

The LEDS and the dip-switch are assigned to “Port A” and “Port B”.The respective registers are `PADR` and `PBDR`[3].These registers are accessed however,with the so-called indexed mode.The SC410 features two global access registers with names `CSC_INDEX` and `CSC_DATA`[4],which are used as intermediate storage.Accessing a specific register,is a two step process.First the hex address of the register must be written to the global index register(CSC_INDEX) and then the actual data value should be written/read to/from the global data register(CSC_DATA).Here is an example:

```
/* Writing hex value temp to register TARGET */
cli();
outb(TARGET, CSC_INDEX);
outb(temp, CSC_DATA);
sti();

/* Reading register SOURCE to variable result*/
cli();
outb(SOURCE, CSC_INDEX);
result=inb(CSC_DATA);
sti();
```

The *cli()* and *sti()* functions disable and enable interrupts.They are essential since the index and data registers are global,and so we must be certain that no other process accesses them apart from us.The Linux kernel provides a better approach for disabling/enabling interrupts.

[2]This is not the only range of this type
[3]with hex addresses 0xA9 and 0xA8
[4]and hex addresses 0x22 and 0x23

The port A/B registers themselves use indexed access.The respective index registers are PAMR(hex 0xA5) and PBMR(hex 0xA4).Switching on a led or reading the dip-switch status is now a four step process.Two to select the port with indexed access and another two steps to read/write to the selected port with indexed access.

6.2.3 Data transfer to/from user-space

In the "read" function of the driver, where the string representing the dip-switch status is being copied to a character array/pointer,there is an important function called *copy_to_user()*.This function replaces *memcpy()* when memory bytes need to be copied from kernel space to user-space.

There is also a function which performs data transfer the opposite way called *copy_from_user()* which again replaces *memcpy()*.The reason these two functions exist is that kernel-space and user-space are so different that one cannot simply "copy" bytes between them.This kind of transfer is cross-space,something that *memcpy()* cannot handle.

6.3 Testing the char driver

The usage of the driver is simple.After compiling the module on the workstation it should be transfered to the Dil/NetPC board with FTP.Then it should be linked with the running kernel with the *insmod* which requires root privileges.Finally a special character file with the selected major/minor numbers must be created which will represent the driver.This is accomplished with the *mknod* command (requires root privileges too).

```
#mknod /dev/myChar c 254 0
```

And here is a simple example

```
#cat /dev/myChar
10000001
#

#echo "3" >/dev/char
#
```

The *cat* command reads the device and shows that all dip-switches are flipped off apart from the first and the last.The *echo* command writes the device.The third LED will be switched on after running the second example,passing 3 as argument.

Chapter 7

Block drivers (software perspective)

The next logical step was the implementation of a block driver.To become fully acquainted with the Linux kernel interface before dealing with the actual hardware,we decided to build a *virtual* block driver.By virtual, we mean a block driver which would manage a memory chunk of the RAM, and not the real hardware device (the CompactFlash itself).With the completion of the driver we could test essential operations,like constructing a filesystem on this memory space and then mounting it.Since the Dil/NetPC has limited RAM we decided that a virtual block disk of 64kb would serve our purposes.

The complete source code is listed in appendix C at the end of this document.The Makefile that was used for the char driver,can be used in this case too, with minimal changes (mainly the name of the .c source file).

7.1 The queue concept

A block driver has almost the same software architecture as the character driver.There is a major difference though.The structure that defines the capabilities of the driver (named `block_device_operations`) is completely different.Apart from the necessary *open()* and *release()* and some block specific system calls (e.g. *check_media_change())* there are no *read()* or *write()* function pointers.

For each block driver (when registered) the Linux kernel initializes a new *request queue*.When a user-space application needs to communicate with the device,instead of routing data I/O directly to the driver as in character drivers, the Linux kernel just adds a new entry to this queue with the meta-data of the request.The block driver asynchronously serves the first request of the queue,discards it, selects the next and so on.The driver source code, neither implements separate functions for input and output, nor are any defined, in the capabilities structure. There is instead only one function, which describes what happens when a request is served.

This approach has some unique advantages.First of all the queue is not a simple first-in-first-out queue.The Linux kernel examines the geometry of each request and places it accordingly in the queue.Requests which are “close enough” are grouped in the queue or at least placed in the best possible sequence.This results in a highly optimized queue which offers great speeds at block devices such as hard disks.The intelligence behind this idea is implemented in core kernel code,so that the driver programmer is free to concentrate on the specific hardware device and not on generic driver code.Newer kernel versions can improve this code without any change in drivers.

Figure 7.1 shows how I/O is performed for both character and block drivers.It demonstrates the different way, data flows between user and kernel space for both types of drivers.

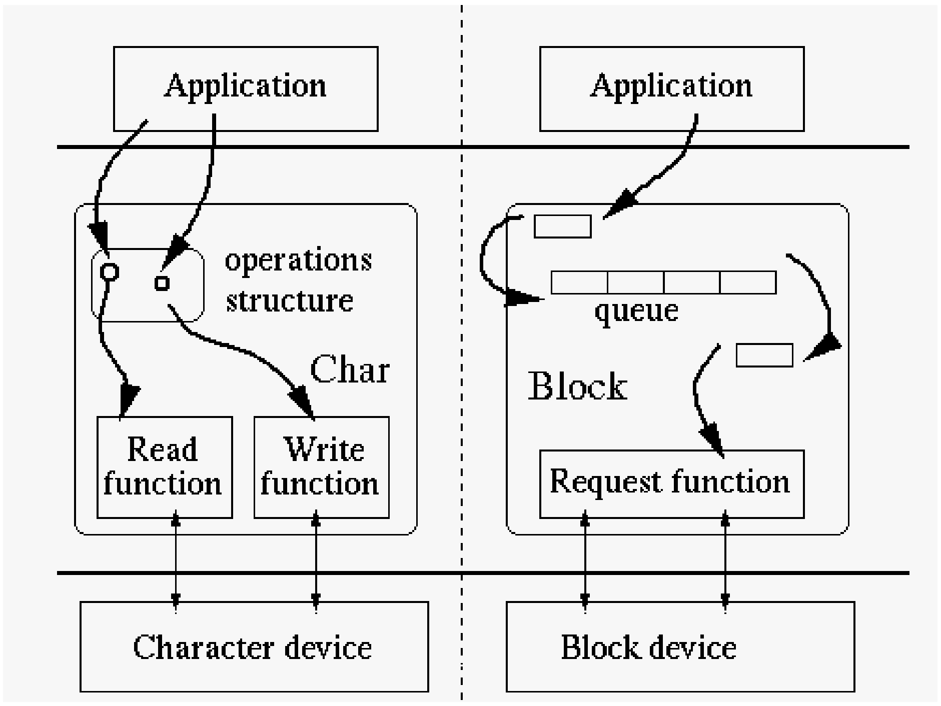

Figure 7.1: Character versus block driver access

The Linux kernel queues are so flexible,that the programmer can either define his/her own queue rules or even use no queue at all and work with the raw requests as they come.It is even possible to use multiple queues in one driver if this is needed.In our case, since this time the “device” is memory and the real hardware described next, is a CompactFlash disk with no moving parts,there was no need for these advanced features.We used the default queue of the Linux kernel.

7.2 Source code description (virtual block driver)

The *init_module()* function of the virtual block driver is similar to the one of the char driver.First the *register_blkdev()* function is called which informs the Linux kernel that this driver implements a block device.The arguments are the same as with the character device registration function.The rest of the code is block specific.The *blk_init_queue()* function deals with the default queue we mentioned above.The first argument indicates that the default queue parameters will be used,while the second argument is a pointer to the function that will serve the requests.

Another responsibility of the *init_module()* is to define the geometry values of the block device.This is accomplished by filling three global arrays with the appropriate information.These two dimensional arrays are indexed by major and minor numbers.The three arrays are `blk_size[][]`,`blksize_size[][]` and `hardsect_size[][]` which define the size of the device in kilobytes,the size of the block in bytes this device uses, and the sector size in bytes (almost always 512).Because our driver implement a new device,memory for the each value must be allocated with the *kmalloc()* function (*malloc()* doesn't exist in kernel space.The example shows how to set the sector size (major is 254):

```
hardsect_size[major]=kmalloc(1 * sizeof(int), GFP_KERNEL);
hardsect_size[major][0]=512; /* 512 Bytes the minimum */
```

An optional setting is how many sectors should the kernel request in advance, when accessing the device.There is an array for all given devices of the same major number `read_ahead[]` and another for each individual device `max_sectors[][]` which define this.Notice that the first array is one-dimensional.These settings affect mainly performance, when it comes to hard disks.In our case, where seek times are zero, they are not essential.

Finally, the memory segment that will hold the actual storage of the driver must be reserved.We use the *vmalloc()* function to allocate 64k of memory to the `data` pointer.We also use the *memset()* function which behaves as in user-space.

The *cleanup_module()* performs in reverse order the above steps.It unregisters the block device,clears the request queue,frees the values of the global arrays, and releases the 64k of memory.

7.2.1 Handling a request

The request function is implemented exactly by the book[1].The whole function is a while loop (it looks infinite but it isn't).The first statement is the `INIT_REQUEST` macro which checks internally the request for validity.The next *printk()* statement prints some vital characteristics of the request.The `CURRENT` macro points to the request which is being served.The actual data transfer that takes places is handled by the function with the same name.It returns the success of the transfer which is passed back to the kernel with the *end_request()* function.

Then the code loops again in order to handle the next request available.If there are no more requests and the queue is empty,the `INIT_REQUEST` macro *returns*,and thus the loop is terminated.

The most important fields of the request structure are the following:

int cmd Shows whether this request is a `READ` or `WRITE` operation.

unsigned long sector Starting sector of the transfer.

unsigned long current_nr_sectors Number of sectors to be transfered.

char *buffer Data buffer where sectors will be read/written from/to.

The last but most important part of the code is the *actual_transfer()* function.The idea behind it, is really simple.The function has available as an argument the current request so it can calculate the size of bytes requested and where to read/write them.First,

[1]Linux device drivers 2nd edition

there is a mandatory check, to see if the request points to data, greater than the capacity of the device.If not,the driver prints a warning and ends the request (with failure).

Then the byte-copying process takes places.We use the *memcpy()* function which is similar with the one from user-space.The arguments depend on the type of the request (read or write).If the operation is "read", we copy from the 64k allocated memory to the buffer of the request.In the opposite case, when we have a "write", we copy from the buffer of the request structure to the memory of the driver.

7.3 Testing the virtual device driver

The best way to test a block driver is to build a file-system on it,mount it and then list/read/write files.We used the `minix` filesystem since the Dil/NetPC has built-in support for it(but not for ext2[2] or for fat[3]).The Minix fs is a simple filesystem which can be created on partitions up to 64MB capacity. After compiling the module on the workstation it should be transfered to the Dil/NetPC board with FTP.Then it should be linked with the running kernel with the *insmod* which requires root privileges.

This time a special *block* file with the selected major/minor numbers must be created which will represent the driver.This is accomplished with the *mknod* command (requires root privileges too).Notice the "b" argument.

```
#mknod /dev/myBlock b 254 0
```

The next step is to build the file-system on the device.This is accomplished by a user-space program called mkfs.minix which is already installed in the Dil/NetPC.

```
#dd if=/dev/zero of=/dev/myBlock bs=1k count=64
#mkfs.minix /dev/myBlock
```

The first command writes zeroes to the device.This isn't necessary since with the *memset()* function we cleared the allocated memory,but nevertheless it is a good tactic before creating a filesystem.Finally we can mount the device:

```
#mkdir /mnt/test
#mount -t minix /dev/myBlock /mnt/test
#cd /mnt/test
#ls
#df
```

The final command (Disk Free) should show that `/dev/myBlock` which is mounted on */mnt/test* is indeed 64KB.There are however 62.5KB free on the device,since one or two kilobytes are reserved by the minix filesystem itself.

[2]the native filesystem of Linux
[3]the native filesystem of DOS/win95/win98

Chapter 8

Block drivers (hardware perspective)

The previous chapter examined the generic programming interface of block drivers.All block drivers for the Linux kernel must conform with it, so that communication between the kernel and the device is standard.This way all user-space applications can use the device without knowing the details of the hardware.A driver is essentially a programming module like any other.It hides an implementation behind a well defined API which is exported to the “outer” world.

Now, the hardware-specific part must be examined.We describe here the low-level communication, between the CPU of the Dil/NetPC and the CompactFlash adapter.The full source code is again available at appendix C.It is similar to the code of the virtual block driver.The same Makefile can be used too.This time however,data transfer is a lot more complicated than a simple *memcpy()* function.

We followed a bottom-up approach for this chapter.First we describe the low-level data transfer as defined by the ATA/IDE protocol,then how to actually control and send commands to the CompactFlash from the Dil/NetPC and finally how to wrap all this a block Linux driver.

8.1 ATA/IDE specifications

CompactFlash cards can operate in three major modes:

1. PC Card Memory mode
2. PC Card I/O mode
3. True IDE mode

We will deal with the third mode which makes CompactFlash cards behave like a hard-disk.With this mode, interfacing with the card is accomplished by the use of the ATA/IDE protocol.Therefore will will only describe the registers,operations and functions of the CompactFlash which are relevant to this mode only.

Since the ATA/IDE protocol is targeted at hard-disks, we used a small subset of it.For example, the protocol defines a “seek” operation which has no point for CompactFlash cards (no movable parts).We are interested only in the basic I/O operations.

8.1.1 Registers

The ATA/IDE protocol defines the following registers.

- Data Register (read/write)
- Alternative Register (read)
- Drive Address Register (only for compatibility)
- Error Register (read)
- Sector Count Register (read/write)
- Sector Number Register (read/write)
- Cylinder Low Register (read/write)
- Cylinder High Register (read/write)
- Drive Head Register (read/write)
- Status Register (read)
- Control Register (write)
- Feature Register (write)
- Command Register (write)

Some registers are overlapped.For example *Command* and *Status* are the same register.Reading from it returns the contents of the Status register,while data written to it are sent to the Command register.

A detailed description of the registers follows.

Data Register

This is a 16 bit wide register.It can be divided into two registers (data high/data low) when 8-bit transfer mode is used.It is the register which holds the actual data which are written/read to/from the CompactFlash Disk.

Alternative Register

Same bits as the Status register.Reading the Status register does clear a pending interrupt while reading the Auxiliary Status register does not.

Drive Address Register

Not used.For compatibility purposes only.

Error Register

In case of an error,this register contains the error code.The host processor should read this register when it detects an error,indicated by bit 0 of the Status Register.

Error Register							
7	6	5	4	3	2	1	0
BBK	UNC	0	IDNF	0	ABRT	0	AMNF

Bit 7(BBK) this bit is set when a Bad Block is detected.

Bit 6(UNC) this bit is set when an Uncorrectable Error is encountered.

Bit 5 this bit is 0.

Bit 4(IDNF) the requested sector ID is in error or cannot be found.

Bit 3 this bit is 0.

Bit 2(Abort) This bit is set if the command has been aborted because of a CompactFlash Storage Card status condition: (Not Ready, Write Fault, etc.) or when an invalid command has been issued.

Bit 1 this bit is 0.

Bit 0(AMNF) This bit is set in case of a general error.

Sector Count Register

Number of sectors to read or write.If zero, the maximum value (256) is assumed.Because we didn't use multiple sector transfers,we always set this register to 1.

Sector Number Register

This register holds the starting sector of the operation.In the case of LBA (described in the next section) it holds bits 0-7.

Cylinder Low Register

This register holds the *low* 8 bits of the cylinder for the next operation.In the case of LBA (described in the next section) it holds bits 8-15.

Cylinder High Register

This register holds the *high* 8 bits of the cylinder for the next operation.In the case of LBA (described in the next section) it holds bits 16-23

Drive Head Register

This register holds the drive/head in the case of C/H/S addressing or bits 24-27 in the case of LBA.

Drive/Head register							
7	6	5	4	3	2	1	0
1	LBA	1	DRV	HS3	HS2	HS1	HS0

Bit 7 this bit is 1.

Bit 6(LBA) 0 to use C/H/S or 1 to select LBA.In the case of LBA this register contains bits 24-27.

Bit 5 this bit is 1.

Bit 4(DRV) The drive number 0 or 1.We used 0

Bit 3(HS3) Bit 3 of the head number in C/H/S mode.Bit 27 in LBA

Bit 2(HS2) Bit 2 of the head number in C/H/S mode.Bit 26 in LBA

Bit 1(HS1) Bit 1 of the head number in C/H/S mode.Bit 25 in LBA

Bit 0(HS0) Bit 0 of the head number in C/H/S mode.Bit 24 in LBA

Status Register

This register returns the status of the CompactFlash card.It is a very important register.

Status register							
7	6	5	4	3	2	1	0
BUSY	RDY	DWF	DSC	DRQ	CORR	0	ERR

Bit 7(BUSY) this bit is 1 while the CompactFlash is performing an operation.No other bits are valid in this case.The command registers are locked too.

Bit 6(RDY) This bit shows whether the CompactFlash is ready to accept the next command.

Bit 5(DWF) If this bit is 1 a write fault has occurred

Bit 4(DSC) This bit shows whether the CompactFlash is ready to accept the next command.(Device seek complete).

Bit 3(DRQ) This bit is 1 when the CompactFlash waits the host to read/write data from/to the Data register.(Data request)

Bit 2(CORR) A correctable error has occurred and corrected too.

Bit 1 This bit is 0.

Bit 0(ERR) An error has occurred.The Error register shows type of error.

Control Register

Writing to this register results in a software reset of the CompactFlash.

Control register							
7	6	5	4	3	2	1	0
X	X	X	X	X	SWRST	-IEN	0

Bit 7 This bit is reserved.

Bit 6 This bit is reserved.

Bit 5 This bit is reserved.

Bit 4 This bit is reserved.

Bit 3 This bit is reserved.

Bit 2(SWRST) While this bit is 1 the CompactFlash performs a software reset.The reset ends when this bit is set again to 0.

Bit 1(-IEN) When this bit is 0 interrupts are enabled.If it is 1 interrupts are disabled.

Bit 0 This bit is 0.

Feature Register

A write-only register that the host can use to select device-specific characteristics.We didn't use it.

Command Register

Another important register.Writing to this register one of the ATA codes,issues the respective command to the CompactFlash Disk.

8.1.2 ATA/IDE Commands

Table B.1 at the end of this document (appendix) lists all ATA/IDE commands.Although there are many commands for various purposes (security,power saving e.t.c) we used the most essential of them which are the following:

Basic ATA/IDE Commands	
Command	hex code
Identify driver	E4h
Read sector(s)	20h/21h
Write sector(s)	30h/31h

These commands are written to the Command Register described in the previous section.We use the "read sector" operation to transfer a single sector (512 bytes) from the CompactFlash card to Dil/NetPC.We use the "write sector" to transfer a single sector from Dil/NetPC to the CompactFlash.The "Identify drive" operation is used to query the CompactFlash for the geometry values and therefore capacity (along with some other informational data).The first two commands accept a single parameter which show the sector of the CompactFlash that will be written/read to/from.It is defined by setting the geometry values (C/H/S or LBA) at the respective registers *before* sending the hex code to the Command Register that starts the operation.

8.2 Communication with the CompactFlash adapter

The CompactFlash adapter from `Elektor` provides an easy way to use a CompactFlash card with IDE/ATA.The adapter is plugged directly on ATA/IDE cable.The ATA/IDE registers are hardwired to predefined addresses.Table 8.1 shows these addresses.Notice that there are two 8-bit data registers instead of one 16-bit wide[1].

Register Name	Base hex address
Data (MSB)	0x0
Alternative	0x6
Drive address	0x7
Data (LSB)	0x8
Error	0x9
Sector count	0xA
Sector number	0xB
Cylinder Low	0xC
Cylinder High	0xD
Drive Head	0xE
Status	0xF
Device Control	same as Alternative
Feature	same as Error
Command	same as Status

Table 8.1: CompactFlash adapter register mappings

To actually communicate with the CompactFlash,the host CPU writes and read from these registers.The sequence of steps required for each operation is examined in detail in the following paragraphs.

8.2.1 Reading a sector

To read a sector first the host CPU asks from the CompactFlash controller to fetch it and copy it to the internal buffer of the card.Then the host CPU reads the two data registers(high and low until all 512 bytes are retrieved.

The procedure involves:

1. Selecting the starting sector by setting the appropriate values at Drive Head,Cylinder low,Cylinder high and Sector registers.

2. Indicating that only one sector should be transfered by writing 1 to Sector Count register.

3. Sending the read command(0x20 or 0x21) to the Command register.

[1]Msb=most significant bits,lsb=least significant bits

4. Waiting until the CompactFlash is ready *and* has fetched the sector.
5. Reading 256 times the two data registers (high and low) until all 512 bytes of the sector are retrieved.

Figure 8.1 shows the flow chart of the procedure.

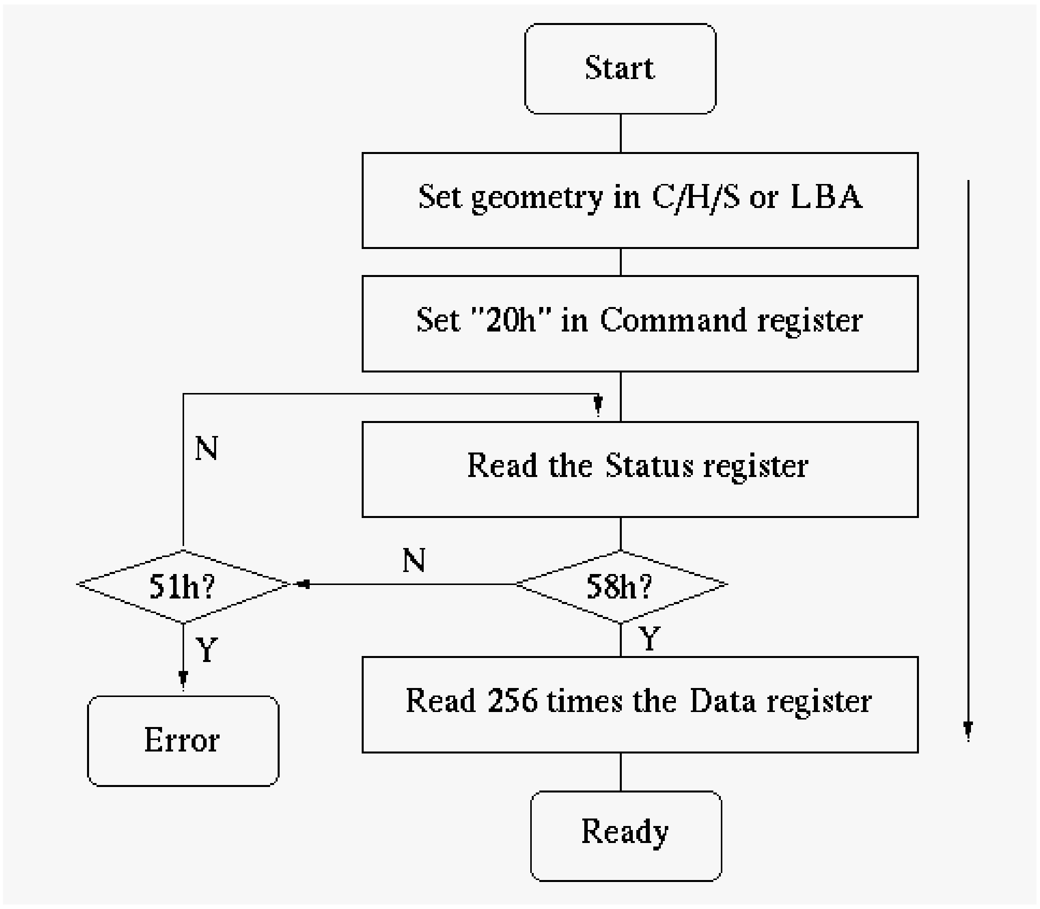
Start
Set geometry in C/H/S or LBA
Set "20h" in Command register
N
Read the Status register
N
51h?
58h?
Y
Y
Error
Read 256 times the Data register
Ready

Figure 8.1: Sector read procedure

8.2.2 Writing a sector

To write a sector first the host CPU copies the data to the internal buffer internal buffer of the card.Then the CompactFlash controller writes the sector to the memory cells and informs the host CPU the success of the operation.

The procedure involves:

1. Selecting the starting sector by setting the appropriate values at Drive Head,Cylinder low,Cylinder high and Sector registers.
2. Indicating that only one sector should be transfered by writing 1 to Sector Count register.
3. Sending the write command(0x30 or 0x31) to the Command register.

4. Waiting until the CompactFlash is ready for the transfer.
5. Writing 256 times the two data registers (high and low) until all 512 bytes of the sector are transfered.
6. Waiting until the CompactFlash has verified that the sector is saved without errors.

Figure 8.2 shows the flow chart of the procedure.

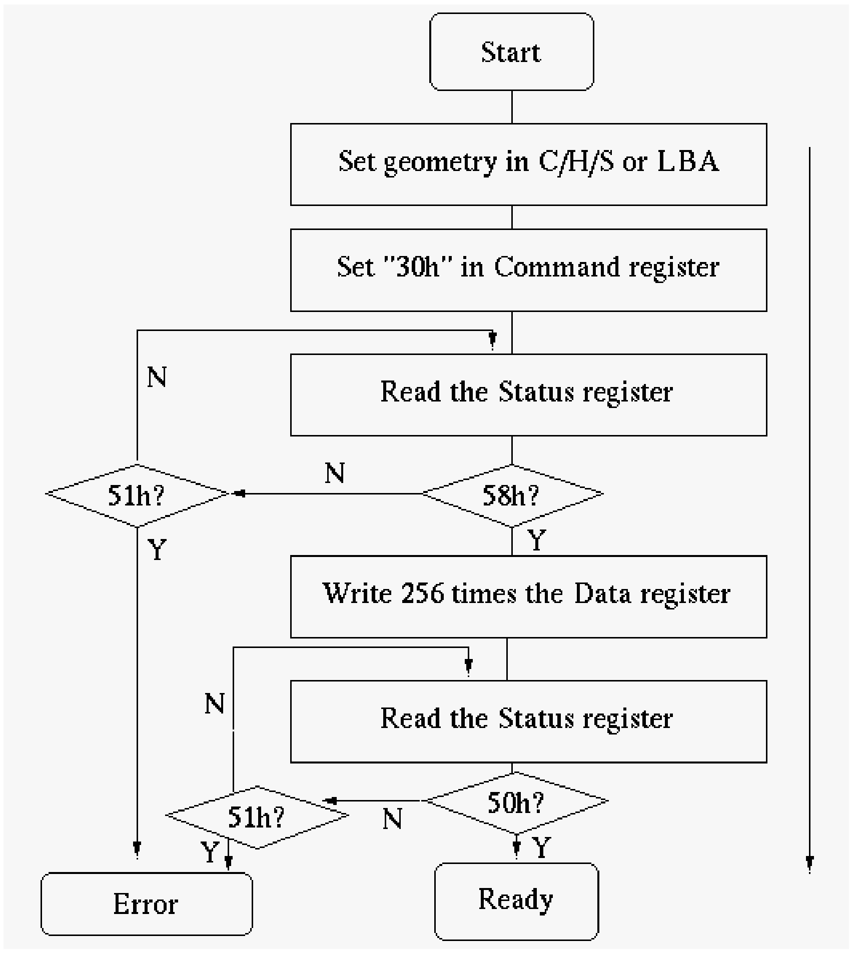
Start
Set geometry in C/H/S or LBA
Set "30h" in Command register
N
Read the Status register
51h?
N
58h?
Y
Y
Write 256 times the Data register
N
Read the Status register
51h?
N
50h?
Y
Y
Error
Ready

Figure 8.2: Sector write procedure

8.2.3 Identifying the CompactFlash

Identifying the CompactFlash card is similar to reading a sector.We request however,a special sector which contains ASCII characters and is hardcoded into the CompactFlash itself,instead of a real sector stored in the memory cells of the card.From this sector we can acquire the vendor of the card,the firmware version,the ID number, and most importantly, the capacity of the card.

The procedure is the same with the read operation,apart from the fact that no sector is selected with Drive head,cylinder low/high and sector registers.Instead, the host CPU sends right away the identify command (0xEC) to the Command register.

From the resulting sector several information fields can be exported.The format of this sector is examined with details in the CompactFlash specifications.Since many fields are vendor-specific we used only the most basic.Table 8.2 shows the fields we are interested in.(Word=2bytes=16bits)

Word Address	Total bytes	Description
0	2	Should be 0x848A for CompactFlash cards
1	2	Number of cylinders
3	2	Number of heads
6	2	Number of sectors per track
21	2	Internal Buffer size in 512 byte increments
7-8	4	Number of total sectors
10-19	20	Serial ID of the Card
23-26	8	Firmware Version
27-46	40	Vendor String

Table 8.2: Identification sector contents

The capacity of the CompactFlash card can be calculated in two ways.The first way is:

$$Totalsectors * 512bytes/sector$$

The second way is:

$$Cylinders * heads * sectors/track * 512bytes/sector$$

These two numbers must match.

8.3 LBA translation

The sample code from the Elektor magazine uses C/H/S addressing mode.With this mode, each sector is characterized by 3 numbers.This is inconvenient since each sector could be defined by only one number.Moreover, there were some technical limitations with this mode.To this purpose, the LBA addressing mode was proposed which considers all sectors of a device members of a single array.Each sector can now be selected by it's index.The LBA address of a sector are 27 bits which uniquely identify it.

Transferring data according to the Linux kernel layer involves only a starting sector and the number of sectors to transfer,so LBA addressing is ideal.The sample code from elektor uses C/H/S addressing so an important issue was to adapt it to use LBA addressing.

```
printf ("Reading track at cyl:%u head:%u sector:%u\n\r",cyl,head,sector);
atrDRHEAD=(0xA0 | head);
atrCYLHIG = cyl>>8;
atrCYLLOW = cyl;
atrSECNR=sector;
atrSECCNT=1;
atrCOMMAND=0x20;
```

This is the original code which reads a sector provided that `cyl`, `head` and `sector` define the geometry of the sector.The `atrXXX` variables are macros which point to the hex addresses of the respective registers.Notice that since this code is for the 8051 processor no *outb()* functions are present,instead simple assignment is used.The first step was to make it compile as a x86 Linux object file.

```
printk ("Reading track at cyl:%u head:%u sector:%u\n",cyl,head,sector);
outb((0xA0 | head),atrDRHEAD);
outb(cyl>>8,atrCYLHIG);
outb(cyl,atrCYLLOW);
outb(sector,atrSECNR);
outb(1,atrSECCNT);
outb(0x20,atrCOMMAND);
```

Then we had to use LBA.The C/H/S code is simple.The sector value is copied directly to the respective register, the low bits of cylinder value are copied to Cylinder Low Register (since *outb()* copies a single byte),and by shifting the value, the high bits are sent to Cylinder high Register.Finally the head value (4 bits-max 8 heads) are copied to Drive head register after an OR operation with 0xA0 (10100000) which selects drive 0 (bit 4) indicating also that this is C/H/S (bit 6).See the previous section for the details of the register.

For the LBA addressing mode the 27 bits must be allocated in these 4 registers.Bits 0-7 belong to the Sector register,bits 8-15 belong to the Cylinder low register,bits 16-23 belong to the Cylinder high register and bits 24-27 belong to the Drive head register.This can be accomplished by shifting operations easily.

Only the low 4 bits of the Drive head register are LBA bits.The high 4 should be 1110 which set LBA mode(bit 6).So if we have a variable with 24-31 LBA bits (28 to 31 are garbage) we must perform an AND operation with 0xF (00001111) to discard the garbage by setting the high 4 bits to zero and keeping the value of the low 4 bits.Then we must perform an OR operation with 0xE0 (11100000) to set LBA mode and leave the other bits intact.

So if variable `sector` holds the LBA address,the final LBA code is:

```
printk ("Reading track at LBA sector:%u\n",sector);
outb((((sector>>24)&0xF)|0xE0),atrDRHEAD);
/* sector >>24 take the [31-24] (we need 27-24) */
/* & 0xF (00001111) discard [31-28] */
/* | 0xE0 (11100000) set LBA */
outb(sector>>16,atrCYLHIG); /* The [23-16] bits */
outb(sector>>8,atrCYLLOW); /* The [15-8] bits */
outb(sector,atrSECNR);  /* The [7-0] bits */
outb(1,atrSECCNT);
outb(0x20,atrCOMMAND);
```

This way we can read and write sectors with only one argument (the LBA address) of the sector instead of three.

8.4 Chip select mapping

To access the external pins of the 8051 the programmers assigns values to predefined hex addressed.With x86 processors this is not so simple.The programmer must "map" the registers of the device to the memory space of the processor.Then,writing and reading to/from this memory range sends and receives commands from the device.

8.4.1 Chip select signal

Chip select is a hardware term.It it a method to share a single bus between multiple devices.The host CPU and the slave devices are all connected to the bus using the same signals (address/data).Then a separate chip select signal for each device is connected to the host CPU.These signals are used by the host CPU to select which device to communicate with.Figure 8.3 shows the configuration.

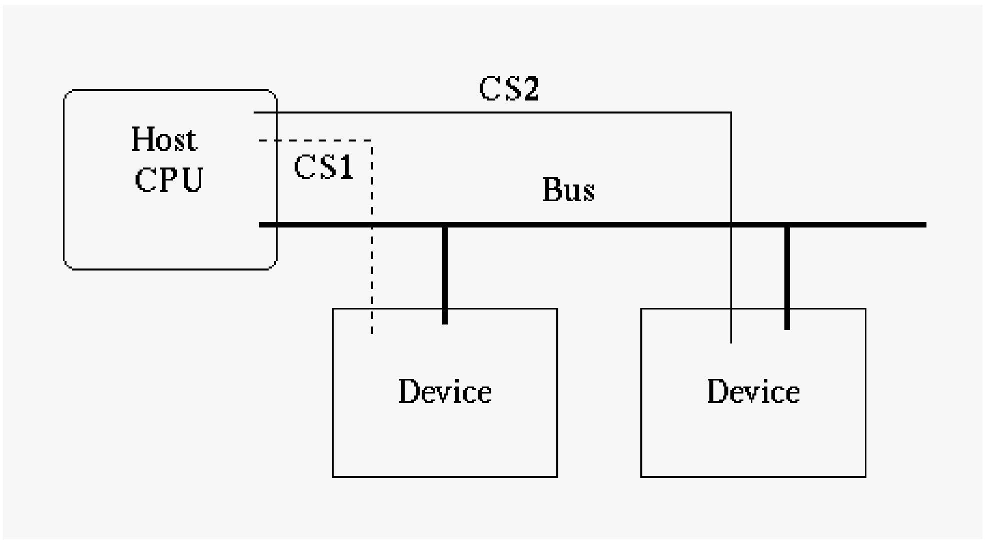

Figure 8.3: Chip select example

The CPU sends commands across the bus and sets the appropriate chip select signal.All devices ignore the bus apart from the one which has the chip select signal set.In our case there is only one device connected to the bus of the Dil/NetPC,but we must still deal which chip select so that when commands are sent to the bus,the chip select signal of the CompactFlash adapter must be set.

8.4.2 I/O mapping concept

The next step is to reserve some memory space from the address range of the processor to map the CompactFlash registers.Table A.3 at the end of this document shows the full address range of DNP/1486-3v.The free segments are shown in table 8.3.(By free we mean unused for both the DNP/1486-3v and the PC/AT standard).

We choose the first available range starting from 0x240.Since we need 16bytes for the registers of the CompactFlash (see table 8.1) we must map 0x240h-0x24Fh.This address range must be mapped to the CompactFlash registers.

Start Range	to	end range
0x240h	-	0x277h
0x280h	-	0x2AFh
0x330h	-	0x33Fh
0x340h	-	0x36F
0x390h	-	0x39F

Table 8.3: Unused address map

The whole idea is that after everything is setup correctly writing the reserved address range doesn't send the data to the actual memory of the CPU,but instead they are directed to the bus of the Dil/NetPC.The chip select signal is activated so the CompactFlash adapter which "listens" to the bus, receives the data and writes them to the internal register it is addressed to.

Here is an example.

```
outb(0x1,0x24B);
```

A simple *outb()* line which sends 0x1 to address 0x24B.This triggers the following:

1. The CPU does not write to the "real" 0x24B address because it detects that 0x240 to 0x24F area is I/O mapped.
2. A byte with value 0x1 is sent to the bus through the 8 bit lines.See table 4.3
3. The address bits of the bus are set to 0xB.The 0x240 address is the base address which is used by the CPU only for chip select mapping.
4. The write signal of the bus is triggered.
5. The chip select signal becomes low.
6. The CompactFlash adapter which is connected to the bus (see table 4.4) notices activity.
7. Since the chip select signal is activated the CompactFlash reads 0x1 from the bus.
8. The adapter notices that address 0xB is internally mapped to the Sector Count Register (see table 8.1).So it writes 0x1 to it
9. According to the ATA/IDE specifications the Sector count is set to 1.

The next subsection describes the registers of the CPU of the Dil/NetPC, that will be used to setup chip select mapping.All registers are accessed with *indexed* mode.First their address is sent to 0x22h and then data is accessed at 0x23h.

8.4.3 AMD SC410 Register set

The chip select pin of the Dil/NetPC what we will use is CS1.The internal pin of the processor is GPIO_CS0.A description for registers of the CPU of the Dil/NetPC that we will use to setup chip select mapping follows:

GPIO Read-Back/Write Register A (0xA6)

This register is the main chip select register which controls which chip select signals are used or not.Since a chip select signal is set low during activation the 0 value shows chip select slots that we will use.

GPIO Read-Back/Write Register A								
Bit	7	6	5	4	3	2	1	0
Default	X	X	X	X	X	X	X	X

Bit 7 GPIO_CS7 Status and Control.

Bit 6 GPIO_CS6 Status and Control.

Bit 5 GPIO_CS5 Status and Control.

Bit 4 GPIO_CS4 Status and Control.

Bit 3 GPIO_CS3 Status and Control.

Bit 2 GPIO_CS2 Status and Control.

Bit 1 GPIO_CS1 Status and Control.

Bit 0 GPIO_CS0 Status and Control.

GPIO_CS Function Select Register A (0xA0)

This register defines whether CPIO_CS0 to CPIO_CS3 are used for input or output

GPIO_CS Function Select Register								
Bit	7	6	5	4	3	2	1	0
Default	0	0	0	0	0	0	0	0

Bit 7 GPIO_CS3 Signal is a Primary Activity and Wake-up (Falling Edge) Enable
0 = Not activity or wake up
1 = Cause activity or wake up

Bit 6 GPIO_CS3 Signal is an Input/Output
0 = Input
1 = Output

Bit 5 GPIO_CS2 Signal is a Primary Activity and Wake-up (Falling Edge) Enable
0 = Not activity or wake up
1 = Cause activity or wake up

Bit 4 GPIO_CS2 Signal is an Input/Output
0 = Input
1 = Output

Bit 3 GPIO_CS1 Signal is a Primary Activity and Wake-up (Falling Edge) Enable
0 = Not activity or wake up
1 = Cause activity or wake up

Bit 2 GPIO_CS1 Signal is an Input/Output
0 = Input
1 = Output

Bit 1 GPIO_CS0 Signal is a Primary Activity and Wake-up (Falling Edge) Enable
0 = Not activity or wake up
1 = Cause activity or wake up

Bit 0 GPIO_CS0 Signal is an Input/Output
0 = Input
1 = Output

GPIO Termination Control Register A (0x3B)

This register controls whether the internal pull-ups for chip selects are enabled or disabled.Since we want our chip select signal to be set high(1) and become low(0) only when activated we must set the appropriate pull-up resistor which will pull the volt level to high.

GPIO Termination Control Register A								
Bit	7	6	5	4	3	2	1	0
Default	1	1	1	1	1	1	1	1

Bit 7 GPIO_CS7 Pull-up Enable/Disable
0 = Disabled
1 = Enabled

Bit 6 GPIO_CS6 Pull-up Enable/Disable
0 = Disabled
1 = Enabled

Bit 5 GPIO_CS5 Pull-up Enable/Disable
0 = Disabled
1 = Enabled

Bit 4 GPIO_CS4 Pull-up Enable/Disable
0 = Disabled
1 = Enabled

Bit 3 GPIO_CS3 Pull-up Enable/Disable
0 = Disabled
1 = Enabled

Bit 2 GPIO_CS2 Pull-up Enable/Disable
0 = Disabled
1 = Enabled

Bit 1 GPIO_CS1 Pull-up Enable/Disable
0 = Disabled
1 = Enabled

Bit 0 GPIO_CS0 Pull-up Enable/Disable
0 = Disabled
1 = Enabled

Suspend Mode Pin State Override Register(0xE5)

This register has many bits with no interest to us.The important bit is 0 which controls the termination of pins.It is the "general" pull-up register.

Suspend Mode Pin State Override Register								
Bit	7	6	5	4	3	2	1	0
Default	0	0	0	X	X	0	0	0

Bit 7 PC Card Socket B Termination Override.

Bit 6 PC Card Socket A Termination Override.

Bit 5 Suspend Mode Termination Override.

Bit 4 Reserved.

Bit 3 Reserved.

Bit 2 ISA Interface Termination Override.

Bit 1 ROM Interface Termination Override.

Bit 0 Pin Termination Latch Command.Writing 1 to this pin results to the proper termination at the pins of registers 3B 3Eh, CAh, EA[6], and F2h.

GP_CSA I/O Address Decode Register(0xB4)

This register defines the 8 low bytes of the address space mapped for CSA.

GP_CSA I/O Address Decode Register								
Bit	7	6	5	4	3	2	1	0
Default	0	0	0	0	0	0	0	0

Bit 7 Chip Select A Address Bit 7.

Bit 6 Chip Select A Address Bit 6.

Bit 5 Chip Select A Address Bit 5.

Bit 4 Chip Select A Address Bit 4.

Bit 3 Chip Select A Address Bit 3.

Bit 2 Chip Select A Address Bit 2.

Bit 1 Chip Select A Address Bit 1.

Bit 0 Chip Select A Address Bit 0.

GP_CSA I/O Address Decode and Mask Register(0xB5)

This register defines the 2 high bytes of the address space mapped for CSA and also the mask that will be used.Details are examined in the next section.

GP_CSA I/O Address Decode and Mask Register								
Bit	7	6	5	4	3	2	1	0
Default	X	X	0	0	0	0	0	0

Bit 7 Reserved.

Bit 6 Reserved.

Bit 5 Mask SA3

Bit 4 Mask SA2

Bit 3 Mask SA1

Bit 2 Mask SA0

Bit 1 Chip Select A Address Bit 9.

Bit 0 Chip Select A Address Bit 8.

GP_CSA/B I/O Command Qualification Register(0xB8)

This register controls the qualification of CSA,meaning whether chip select will set to low only when the address bits of the bus change or when the `read` or `write` signal change too.These are bits 0 and 1.

GP_CSA/B I/O Command Qualification Register								
Bit	7	6	5	4	3	2	1	0
Default	X	0	0	0	X	0	0	0

Bit 7 Reserved.

Bit 6 GP_CSB ISA I/O Cycle Data Bus Width and Timing Selector.

Bit 5 Qualify GP_CSB with IOR/IOW.

Bit 4 Qualify GP_CSB with IOR/IOW.

Bit 3 Reserved.

Bit 2 GP_CSA ISA I/O Cycle Data Bus Width and Timing Selector.

Bit 1-0 Qualify GP_CSA with IOR/IOW.
0 0 = Not additionally qualified by -IOR or by -IOW
0 1 = Additionally qualified by -IOR only
1 0 = Additionally qualified by -IOW only
1 1 = Additionally qualified by either -IOR or -IOW.

GP_CS to GPIO_CS Map Register A(0xB2)

We use this register to assign GP_CSA of the Dil/NetPC to GPIO_CS0 of the CPU.

GP_CS to GPIO_CS Map Register A								
Bit	7	6	5	4	3	2	1	0
Default	1	1	1	1	1	1	1	1

Bit 7-4 Map GP_CSB to One of the GPIO_CS Pins
The number programmed corresponds to the GPIO_CS that is mapped:
0 0 0 0 = GPIO_CS0
0 0 0 1 = GPIO_CS1
. .
1 1 1 0 = GPIO_CS14
1 1 1 1 = Disabled

Bit 3-0 Map GP_CSA to One of the GPIO_CS Pins
The number programmed corresponds to the GPIO_CS that is mapped:
0 0 0 0 = GPIO_CS0
0 0 0 1 = GPIO_CS1
. .
1 1 1 0 = GPIO_CS14
1 1 1 1 = Disabled

8.4.4 Setting chip select mapping in software

The hardware part of the chip select setup was described already.The appropriate pins of the CompactFlash adapter and the Dil/NetPC must be connected together via an ATA/IDE cable.Now we describe how to setup it in software.

The following code informs the the AMD CPU that memory space 0x240 to 0x24F is I/O mapped and that any transfer to/from it should be directed to the bus with the chip select signal (CS1) activated.

```
printk("Setting I/O mapping...");
write_value(0xFF,0xA6);
write_value(read_value(0xA0)|0x01,0xA0);
write_value(read_value(0x3B)|0x01,0x3B);
write_value(read_value(0xE5)|0x01,0xE5);
write_value(0x40,0xB4);
write_value((read_value(0xB5)&0xC0)|0x02,0xB5);
write_value((read_value(0xB8)&0x88),0xB8);
write_value(read_value(0xB2)&0xF0,0xB2);
write_value(0xFE,0xA6);
printk("OK\n");
```

The *write_value()* and *read_value()* functions perform simple I/O in indexed mode which was described in a previous chapter.The *write_value()* function sends it's first argument to the register specified by the second argument.

Here is a line-by-line explanation of the code.

```
write_value(0xFF,0xA6);
```

This line writes 0xFF (11111111) to the GPIO Read-Back/Write Register A disabling all chip selects.

```
write_value(read_value(0xA0)|0x01,0xA0);
```

This line sets the last bit of the GPIO_CS Function Select Register A initializing GPIO_CS0 as output.The OR operations is used to leave the other bits of the register intact.

```
write_value(read_value(0x3B)|0x01,0x3B);
```

This line enables the internal pull-up resistor for GPIO_CS0 only.

```
write_value(read_value(0xE5)|0x01,0xE5);
```

Like interrupts, where apart from the individual bits which enable them, there is an "enable all" bit,setting the last of this register enables the pull-up resistors.Without this line, the previous one doesn't work.

```
write_value(0x40,0xB4);
```

This line defines the low 8 bits of the chip select address.For us it is 0x40 (from the full 0x240).

```
write_value((read_value(0xB5)&0xC0)|0x02,0xB5);
```

This line defines the high 2 bits of the chip select address.For us it is 0x2 (from the full 0x240).The AND operation with 0xC0(1100000) leaves the first 2 bits intact and sets the mask to 0000.We select this mask because we want 16 bytes from 0x240 to 0x24F.If the mask was for example 1000 we would get 0x240 to 0x247(8 bytes) or if it was 1100 we would have 0x240 to 0x243 (4 bytes).

```
write_value((read_value(0xB8)&0x88),0xB8);
```

This line sends 0x88(10001000) to the GP_CSA/B I/O Command Qualification Register showing that we only want chip select to be triggered by the address signal only.(Not by read and/or write signals).

```
write_value(read_value(0xB2)&0xF0,0xB2);
```

This line writes to the GP_CS to GPIO_CS Map Register A the value 0xF0(11110000) mapping GP_CSA to GPIO_CS0 and disabling GP_CSB.

```
write_value(0xFE,0xA6);
```

This line writes 0xFE (11111110) to the GPIO Read-Back/Write Register A enabling only GPIO_CS0.

That is all.Now all *outb()* and *inb()* functions can "talk" to the CompactFlash card using 0x240-0x24F memory range.

8.5 Source code description (Block driver)

The full source code is listed in appendix C along with the appropriate `makefile`.The code is based on the virtual block driver after incorporating everything presented in this chapter.

The *init_module()* function has many responsibilities now.First it checks if 0x240-0x24F is available with the *check_region()*.The first argument is the start of the range requested and the second the number of bytes requested (0x10h =16 bytes).If the range is indeed available it is reserved with the *request_region()*.The third argument is the name that will be shown under `cat /proc/ioports`.

Next it calls the *ata_reset()* function which resets the CompactFlash for 25 microseconds (the minimum time as defined it the ATA/IDE standard).To fill the global arrays with the geometry details of the CompactFlash card, the *ataidentify()* function must be called first to retrieve them.The *cleanup_module()* must now release the memory requested with *release_region()*

The read and write functions transfer a single sector from/to the CompactFlash card to the global `sector_data[]` array.The `sec1[]` and `sec2[]` arrays (each 256 bytes) keep temporarily the low and high bytes of the sector respectively.

The *printerror()* function is only called when an error has occurred.It prints a message to the console explaining the nature of the error.

Finally the we have included a read only proc entry which can be used for debugging purposes.To implement such an entry one has to create it at *init_module()* function and release it at *cleanup_module()*.Then a simple function (*sample_proc_read()*) defines what happens when the proc entry is accessed.The user can run `cat /proc/CompactFlash` and list the identification data of the card and also the contents of last sector transfered.

Chapter 9

Testing the final driver

The first real test of the driver we performed, that would show us if software and hardware were correct,is the retrieval of the identification string of the CompactFlash card.As soon as the module loads, the *ataidentify()* function is called and the output is printed to the console (the minicom terminal in our case).Early tests presented us with the following:

```
Name      : TSHIBA THNCF128 M A
Firmware  : 03 0
Serial    : STCB21M82029B47305C
Cylinders : 978
Heads     : 8
Sectors   : 32
```

The two strange things about this result are the lack of “O” in the TOSHIBA string,and the firmware number.We couldn’t confirm if the serial number is valid since we didn’t know the correct one.The only verifiable part of this result are the geometry values.We looked up the TOSHIBA CompactFlash specification and the values are correct.Based on this fact we considered the test result a success[1] and proceeded to actual data I/O with the CompactFlash card.

9.1 Data corruption

The raw data we used for this test is the source code (text file) of the driver itself.We would write the text to the CompactFlash card then read it back and check for errors.Each test cycle involved the following commands:

```
dd if=/dev/zero of=/dev/cflash bs=512 count=40

dd if=testfile of=/dev/cflash bs=512

dd if=/dev/cflash of=testX bs=512 count=40
```

The first command writes 0s to the first 40 sectors of the CompactFlash card.This process clears the CompactFlash card from the previous test so that each test is exactly

[1]we were proved wrong later

the same.The second command writes `testfile` which is the .c source file to the CompactFlash card.Finally the third command reads back the first 40 sectors of the card to the `testX` file.Then with the *diff* command a comparison between the two file would show if there were any errors.

The first results were disappointing.There was heavy data corruption between the original and output files.Missing bytes,words with wrong order,transposed characters and even sectors with no data (filled with zeroes) were common.Some parts were 100% correct and some others contained incomprehensible text.This lead us to believe that the TOSHIBA string was broken too ("O" was missing).The corruption was so evident that building a filesystem and mounting the CompactFlash device file didn't work at all.

At that point the code that accesses the CompactFlash was simply a rewrite of the Elektor code for 8051.To remedy the situation we tried the following improvements:

- Assuming that the SC410 is faster that the 8051 we inserted several *udelay()* functions after each hardware access inside the code, and also between the consecutive request.The purpose of this was to allow for the hardware to "catch up" with the software.

- We inserted the *barrier()* macro after each hardware access.This macro ensures that any hardware access functions are actually performed when it is encountered in code,and are not optimized for later.Basically when a *barrier()* macro is inserted, any *inb()* and *outb()* functions up to that point are guaranteed to be complete.This is useful when some registers must have the correct values before a specific operation or process starts.

- We inserted *wmb()* and *rmb()* functions which are supersets of the *barrier()* macro and prevent the reordering of hardware writing and reading functions respectively.

- Whenever the code needs to read the value of the Status register,the result was stored to a `volatile` variable.This forced the compiler to actually read again the Status register next time it was needed,instead of assuming that it had not changed and therefore the old value was still valid.Any code that looked like this:

```
printk ("Reading device information...:\n");
while (inb(atrSTATUS) !=0x50)
  {
  }
```

  was changed to this:

```
volatile unsigned char now;
printk ("Reading device information...:\n");
while ((now=inb(atrSTATUS)) !=0x50)
  {
    barrier();
  }
```

The *barrier()* macro here, has double effect.It ensures that the while loop is not empty and thus,the compiler cannot remove it,and secondly it forces the *inb()* function to complete reading the Status register,retrieving the up-to-date value.Notice that the *barrier()* family of macros has a negative impact on performance.

Unfortunately the situation didn't change much.We performed many tests trying to get better results,but with no success.In fact, we observed that things were a lot worse than we thought in the beginning.The system presented Byzantine behavior.The positions of the corrupt bytes were erratic.Each text file had different corruption schemes than the others.Running the same test for a second time without changing anything,sometimes resulted in different output files.Simply loading and unloading the module presented us with up to three identification strings which would show up randomly.

This lead us to believe that something was wrong with the hardware connections,so we decided to search the problem at a lower level.

9.2 Testing the hardware

To test the hardware we resorted to the logic analyzer.We needed to see what was being carried by the actual data bit lines.We inserted the probes between the Dil/NetPC pins and the ATA/IDE cable that connected it with the CompactFlash adapter.Figure 9.1 shows the configuration.

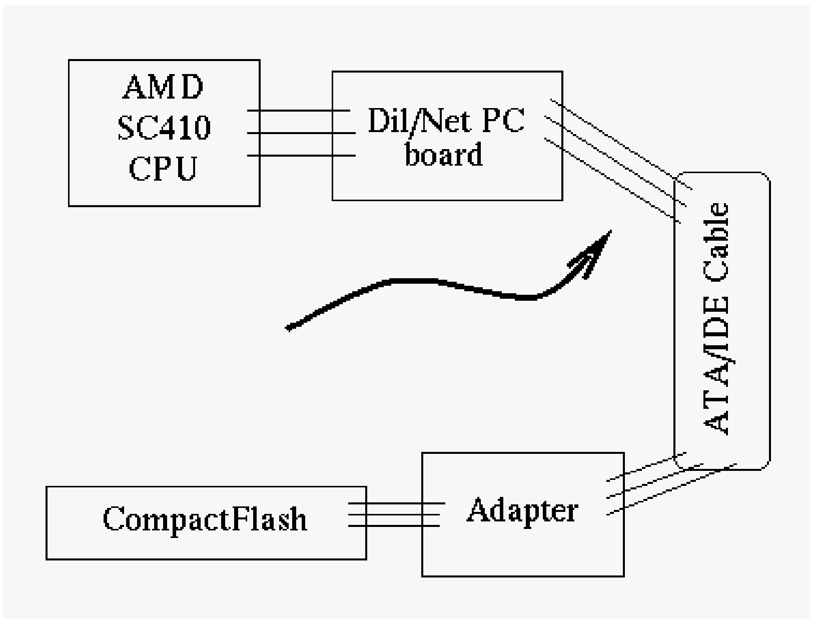

Figure 9.1: Placement of the logic analyzer

We were afraid that the data bytes would never reach the CompactFlash card.We expected the logic analyzer to show us bytes being corrupted during the tranfer process.But this was not the case.We verified that the ATA/IDE commands the code was sending, were actually transfered to the CompactFlash.When the file was written to the card the correct bytes were carried by the data bits.But when the file was retrieved from the CompactFlash, data was already corrupted.The bytes that reached the Dil/NetPC were not correct.

Basically the logic analyzer confirmed that bytes that reached the Dil/NetPC, matched those that the software code was receiving too.The problem was thay they were corrupted in the first place.

The last idea that was proposed was shortening the ATA/IDE cable.We split the cable in two and used one half.This reduced the data corruption a lot(20 bytes corrupted/512 on average).The cable was shortened at about 3cms.Again the results were showing major improvements.(12 bytes corrupted/512 on average).Finally the CompactFlash adapter was soldered directly on the Dil/NetPC board.Displaced bytes were now a very small minority (4 bytes corrupted/512 on average).But this didn't eliminate completely data corruption.The byzantine behavior disappeared though,and now the identification string was stable:

```
Name     : TOSHIBA THNCF128MMA
Firmware : 3.00
Serial   : STCB21M82029B43547C3
```

9.3 Known bugs

We returned back to the software.This time we were testing small files of 512 bytes (a sector).After trial and error we removed the `volatile` variables and also replaced the *inb()* and *outb()* functions with their paused counterparts *inb_p()* and *outb_p()*.

Now all small files of one sector are read back flawlessly.We can even build a filesystem on the CompactFlash and mount it.However after some use,the corrupted bytes build up and all stored files get corrupted.

We stopped development of the driver since we could not export safe conclusions about the cause of the corruption.Although the same text file presented the same corruption schemes for every test run,the position of the corrupted bytes changed for each text file.We could not explain this behavior.For example the text of the BSD software license is read back without errors,while the GPL software license text has small amounts of displaced bytes.Both are simple ASCII files so there is no explanation why the first one works and the other does not.

9.4 Conclusion

The block driver can only be considered experimental and unstable at this point.Use in production environments is not possible yet.Further debugging (software and even hardware) is required.

The fact that shortening the ATA/IDE cable improved the situation,suggests that latency and cable capacitance affect the data transfer.Removing the cable and attaching the adapter directly onto the Dil/NetPC board should eliminate corruption,but this didn't happen.Therefore, there are other factors that we are not aware of yet.

Three areas of the project that should be examined are:

1. Testing of the CompactFlash adapter with an 8051 development board using the original code from elektor.This should show whether there is a hardware fault or not.

2. Explaining why the chip select signal is always activated during a write sector operation.This was detected with the logic analyzer and doesn't happen during the read operation where the chip select signal is only activated when needed.This is a strange behavior and isn't documented anywhere.

3. Finding out why the first sector of a transfer suffers no corruption,but missing bytes can appear at any of the following sectors.All sectors are transfered in a loop,resulting in exactly the same code for each transfer.There is no reason why corruption sometimes appears and other not.

Appendix A

SSV Dil/NetPC DNP/1486-3V specifications

Full specifications are described in the Documentation PDF files of SSV systems which are included in the CD-ROM.We list here the most important for the reader's convenience.

A.1 Full pin-out

Table A.1 shows the first 32 pins (1-32) of the Dil/NetPC.Table A.2 shows the rest 32 pins (33-64).

When holding the Dil/NetPC so that the AMD logo can be read on the CPU,the first pin(1) is on the upper left corner and the last pin (64) is on the upper right one.

A.2 I/O Address map

Table A.3 on page 58 shows the full memory mapping of DNP/1486-3V. This table should be used when selecting free memory ranges for drivers.It is best to select free ranges when they are free both for the Dil/NetPC and the PC/AT Standard.

Pin	Name	Signal	Group	Function
1	PA0	GPIO24	PIO	Parallel I/O, Port A, Bit 0
2	PA1	GPIO25	PIO	Parallel I/O, Port A, Bit 1
3	PA2	GPIO26	PIO	Parallel I/O, Port A, Bit 2
4	PA3	GPIO27	PIO	Parallel I/O, Port A, Bit 3
5	PA4	GPIO28	PIO	Parallel I/O, Port A, Bit 4
6	PA5	GPIO29	PIO	Parallel I/O, Port A, Bit 5
7	PA6	GPIO30	PIO	Parallel I/O, Port A, Bit 6
8	PA7	GPIO31	PIO	Parallel I/O, Port A, Bit 7
9	PB0	GPIO16	PIO	Parallel I/O, Port B, Bit 0
10	PB1	GPIO17	PIO	Parallel I/O, Port B, Bit 1
11	PB2	GPIO18	PIO	Parallel I/O, Port B, Bit 2
12	PB3	GPIO19	PIO	Parallel I/O, Port B, Bit 3
13	PB4	GPIO20	PIO	Parallel I/O, Port B, Bit 4
14	PB5	GPIO21	PIO	Parallel I/O, Port B, Bit 5
15	PB6	GPIO22	PIO	Parallel I/O, Port B, Bit 6
16	PB7	GPIO23	PIO	Parallel I/O, Port B, Bit 7
17	PC0	GPIO12	PIO	Parallel I/O, Port C, Bit 0
18	PC1	GPIO13	PIO	Parallel I/O, Port C, Bit 1
19	PC2	GPIO14	PIO	Parallel I/O, Port C, Bit 2
20	PC3	GPIO15	PIO	Parallel I/O, Port C, Bit 3
21	RXD	COM1_2	SIO	COM1 Serial Port, RXD Pin
22	TXD	COM1_3	SIO	COM1 Serial Port, TXD Pin
23	CTS	COM1_8	SIO	COM1 Serial Port, CTS Pin
24	RTS	COM1_7	SIO	COM1 Serial Port, RTS Pin
25	DCD	COM1_1	SIO	COM1 Serial Port, DCD Pin
26	DSR	COM1_6	SIO	COM1 Serial Port, DSR Pin
27	DTR	COM1_4	SIO	COM1 Serial Port, DTR Pin
28	RI *	COM1_9	SIO C	OM1 Serial Port, RI Pin
29	RESIN	/MR	RESET	RESET Input (Watchdog)
30	TX+	LAN_TX+	LAN	10BASE-T Ethernet, TX+ Pin
31	TX-	LAN_TX-	LAN	10BASE-T Ethernet, TX- Pin
32	GND		—	Ground

Table A.1: Full pin-out of DNP/1486-3V (1-32)

Pin	Name	Signal	Group	Function
33	RX+	LAN_RX+	LAN	10BASE-T Ethernet, RX+ Pin
34	RX-	LAN_RX-	LAN	10BASE-T Ethernet, RX- Pin
35	RESOUT	PWRGD	RESET	RESET Output (Power good from Watchdog)
36	VBAT	VBAT	PSP	SC410 Real Time Clock Battery Input
37	CLKOUT	CLK_IO	PSP	Clock Out (Default 1.8432 MHz)
38	IRTXD	IR_TXD	PSP	SC410 IrDA TXD Pin
39	IRRXD	IR_RXD	PSP	SC410 IrDA RXD Pin
40	INT5	PIRQ7	PSP	Programmable Interrupt Input 5
41	INT4	PIRQ6	PSP	Programmable Interrupt Input 4
42	INT3	PIRQ5	PSP	Programmable Interrupt Input 3
43	INT2	PIRQ4	PSP	Programmable Interrupt Input 2
44	INT1	PIRQ3	PSP	Programmable Interrupt Input 1
45	CS4	GPIO_CS3	PSP	Programmable Chip Select Output 4
46	CS3	GPIO_CS2	PSP	Programmable Chip Select Output 3
47	CS2	GPIO_CS1	PSP	Programmable Chip Select Output 2
48	CS1	GPIO_CS0	PSP	Programmable Chip Select Output 1
49	IOCHRDY	IOCHRDY	PSP	I/O Expansion Bus - I/O Channel Ready
50	IOR	/IOR	PSP	I/O Expansion Bus - I/O Read
51	IOW	/IOW	PSP	I/O Expansion Bus - I/O Write
52	SA3	SA3	PSP	I/O Expansion - Address Bit 3
53	SA2	SA2	PSP	I/O Expansion - Address Bit 2
54	SA1	SA1	PSP	I/O Expansion - Address Bit 1
55	SA0	SA0	PSP	I/O Expansion - Address Bit 0
56	SD7	SD7	PSP	I/O Expansion - Data Bit 7
57	SD6	SD6	PSP	I/O Expansion - Data Bit 6
58	SD5	SD5	PSP	I/O Expansion - Data Bit 5
59	SD4	SD4	PSP	I/O Expansion - Data Bit 4
60	SD3	SD3	PSP	I/O Expansion - Data Bit 3
61	SD2	SD2	PSP	I/O Expansion - Data Bit 2
62	SD1	SD1	PSP	I/O Expansion - Data Bit 1
63	SD0	SD0	PSP	I/O Expansion - Data Bit 0
64	VCC		—	3.3 Volt Power Input

Table A.2: Full pin-out of DNP/1486-3V (33-64)

I/O Address Range	DNP/1486-3V Usage	PC/AT Standard
000h - 00Fh	8237 DMA Controller #1	8237 DMA Controller #1
020h - 021h	8259 Master Interrupt Controller	8259 Master Interrupt Controller
022h - 023h	SC410 CSCIR, CSCDR (Index and Data)	—
040h - 043h	8253 Programmable Timer	8253 Programmable Timer
060h - 06Fh	Port 61h	8042 Keyboard Controller
070h - 07Fh	RTC, NMI Mask Register	RTC, NMI Mask Register
080h - 09Fh	DMA Page Registers	DMA Page Registers
0A0h - 0B1h	8259 Slave Interrupt Controller	8259 Slave Interrupt Controller
0C0h - 0DFh	8237 DMA Controller #2	8237 DMA Controller #2
0F0h - 0F1h	SC410	Math Coprocessor
0F8h - 0FFh	SC410	Math Coprocessor
170h - 177h	Unused	Hard Disk Controller #2
1F0h - 1F8h	Unused	Hard Disk Controller #1
200h - 207h	Unused	Game Port
238h - 23Bh	Unused	Bus Mouse
23Ch - 23Fh	Unused	Alt. Bus Mouse
240h - 277h	Unused	Unused
278h - 27Fh	Unused	Parallel Printer
280h - 2AFh	Unused	Unused
2B0h - 2BFh	Unused	EGA
2C0h - 2CFh	Unused	EGA
2D0h - 2DFh	Unused	EGA
2E0h - 2E7h	Unused	GPIB
2E8h - 2EFh	Unused	Serial Port
2F8h - 2FFh	Unused	Serial Port
300h - 30Fh	On-board LAN Controller	Prototype Card
310h - 31Fh	On-board LAN Controller	Prototype Card
320h - 32Fh	Unused	Hard Disk Controller XT
330h - 33Fh	Unused	Unused
340h - 36Fh	Unused	Unused
370h - 377h	Unused	Floppy Disk Controller #2
378h - 37Fh	Unused	Parallel Printer
380h - 38Fh	Unused	SDLC Adapter
390h - 39Fh	Unused	Unused
3A0h - 3AFh	Unused	SDLC Adapter
3B0h - 3BBh	Unused	MDA Adapter
3BCh - 3BFh	Unused	Parallel Printer
3C0h - 3CFh	Unused	VGA/EGA Adapter
3D0h - 3DFh	Unused	CGA Adapter
3E8h - 3EFh	Unused	Serial Port
3F0h - 3F7h	Unused	Floppy Controller #1
3F8h - 3FFh	Serial	Port COM1 and IrDA Serial Port

Table A.3: Memory map of DNP/1486-3V

Appendix B

ATA/IDE Command Set

This table shows all ATA/IDE commands.

Command	Code
Check Power Mode	E5h/98h
Execute Drive Diagnostic	90h
Erase Sector(s)	C0h
Format track	50h
Identify driver	EC
Idle	E3h/97h
Idle immediate	E1h/95h
Initialize drive parameters	91h
Read buffer	E4h
Read Long sector	22h/23h
Read Multiple	C4h
Read sectors(s)	20h/21h
Read verify sector(s)	40h/41h
Recalibrate	1Xh
Request Sense	03h
Security Disable Password	F6h
Security erase prepare	F3h
Security erase unit	F4h
Security Freeze lock	F5h
Security set password	F1h
Security unlock	F2h
Seek	7Xh
Set Features	EFh
Set Multiple mode	C6h
Set sleep mode	E6h/99/h
Stand by	E2h/96h
Stand by immediate	E0h/94h
Translate sector	87h
Wear level	F5h
Write buffer	E8h
Write long sector	32h/33h
Write multiple	C5h
Write multiple w/o erase	CDh
Write sector(s)	30h/31h
Write sector(s) w/o erase	38h
Write verify	3Ch

Table B.1: CF-ATA Command set

Appendix C

Source code

The source code of the Makefile can be found at
`/Diploma/Final/source-code/Makefile`.

The source code of the Character module can be found at
`/Diploma/Development/code/char/blocktest.c`.

The source code of the virtual block module can be found at
`/Diploma/Development/code/block2/blocktest.c`.

The source code of the real hardware driver can be found at
`/Diploma/Final/source-code/flashtest.c`.

Bibliography

[1] Alessandro Rubini, Jonathan Corbet.*Linux Device Drivers.*O'Reilly and Associates; 2nd edition,June 2001,ISBN 0-59600-008-1.

[2] X3T13 Technical Committee,*AT Attachment-3 Interface (ATA-3 Working draft)*,Revision 7b,27 January 1997

[3] The CompactFlash Association,*CompactFlash Specification* Revision 1.4,July 1999

[4] Toshiba Corporation,*CompactFlash THNCFxxxxMA Series Specification*,TOSHIBA SMALL FORM FACTOR CARD,12 July 2002

[5] SSV embedded systems,*DIL/NetPC DNP/1486-3V Hardware Documentation*,Revision 1.21,8 August 2001

[6] Advanced Micro Devices (AMD),*Elan SC410 Microconroller's Data Sheet*, Publication #21028,Revision B,15 December 1998

[7] Advanced Micro Devices (AMD),*Elan SC410 Microconroller Register Set*, Publication #21032,Revision A,14 January 1997

[8] Advanced Micro Devices (AMD),*Elan SC410 Microconroller's User Manual*, Publication #21030,18 July 1997

[9] Elektor Electronics Magazine,*CompactFlash on IDE Bus*,Issue 309,April 2002,page 12

[10] Elektor Electronics Magazine,*CompactFlash Interface for Microcontroller Systems*,Issue 316,December 2002,page 78

[11] Various authors,*The Linux Documentation project*, http://www.tldp.org