

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



Rapid Data Transfers on COMBO Platform

MASTER'S THESIS

Jiří Slabý

Brno, 2008

Declaration

I hereby declare that I am the sole author of this thesis. All sources and literature used in this thesis are cited and listed properly.

In Brno, May 19, 2008
Jiří Slabý

Advisor: Ing. Pavel Čeleda, Ph.D.

Acknowledgement

I want to thank my thesis advisor Pavel Čeleda. And last but not the least I would like to thank here my parents for their patience too.

Abstract

In this thesis we are concerned in fast transfers from hardware to computer programs and vice versa. Above all we specialize to coherent direct memory access transfers from/to hardware through ring buffers. Also we discuss other transfer methods implemented in other projects and consider using them here. Finally we implement chosen method and test it with all available tools, describe the testing process and compare with already existing solutions.

Keywords

Linux, driver, ring buffer, mmap, DMA, COMBO6x, NetCOPE, Liberouter

Contents

1	Introduction	3
2	Hardware Basics	5
2.1	<i>Devices Interconnection</i>	5
2.1.1	PCI	6
2.2	<i>DMA and Bus Mastering</i>	7
2.2.1	PCI DMA Interface	7
2.2.2	Linux DMA Interface	8
2.3	<i>Memory Management</i>	8
2.3.1	Physical Memory	9
2.3.2	Virtual Memory	9
2.3.3	Memory Map	10
2.4	<i>Character Device</i>	12
2.4.1	Device Nodes	12
2.4.2	udev	12
2.4.3	Character Device Linux Interface	13
2.5	<i>NetCOPE Hardware And szedata2 Design</i>	14
2.5.1	szedata2	14
3	szedata2 Driver	18
3.1	<i>Requirements</i>	18
3.2	<i>Models And Design</i>	19
3.2.1	Use Case Diagram	19
3.2.2	Sequence Diagrams	22
3.3	<i>Code Development</i>	22
3.3.1	Combo6 Code Base	22
3.3.2	Implementation	24
3.3.3	Phase I – Hardware Independent Code	26
3.3.4	Phase I Testing	28
3.3.5	Phase II – Hardware Dependent Code	28
3.3.6	Phase III – Back-porting To Older Kernels	30
3.3.7	Phases II And III Testing	31
3.3.8	Userspace Library	32
3.3.9	Final Code	33

3.4	<i>Measurements And Comparisons</i>	33
3.5	<i>Maintenance</i>	36
4	Conclusion And Future Work	38
4.1	<i>Conclusion</i>	38
4.2	<i>Future Work</i>	38
A	Udev files	41
A.1	<i>udev rules file</i>	41
A.2	<i>csdevname script</i>	41
B	Models	44
B.1	<i>Use Case open_node</i>	45
B.2	<i>Use Case subscribe</i>	45
B.3	<i>Use Case start</i>	45
B.4	<i>Use Case wait_for_data</i>	45
B.5	<i>Use Case get_data</i>	46
B.6	<i>Use Case release_data</i>	46
B.7	<i>Use Case request_space</i>	47
B.8	<i>Use Case put_data</i>	47
B.9	<i>Use Case close_node</i>	48
B.10	<i>Use Case interrupt</i>	48
B.11	<i>Use Case insert_module</i>	48
B.12	<i>Use Case remove_module</i>	49
B.13	<i>Use Case alloc_rings</i>	49
B.14	<i>Use Case dealloc_rings</i>	49
B.15	<i>Sequence Diagrams</i>	49
C	Doxygen documentation	52
C.1	<i>Class Documentation</i>	52
C.2	<i>File Documentation</i>	54
D	Kernel-doc documentation	60
D.1	<i>Userspace interface</i>	60
D.2	<i>Intradrivere interface</i>	62
D.3	<i>Public functions</i>	63
E	CD Contents And Usage	67

Chapter 1

Introduction

To even boot a computer, it is necessary to proceed many transfers inside the hardware among its components. It is a basic function which must work, however there are several methods of how to achieve that. Especially the fast transfers are very important in computer science which results in higher overall throughput of the system itself and also in faster communication with other ones.

As an answer to permanent higher requirements in the networking area, Liberouter project has been established as a project supported by Czech national research and education network CESNET and European project 6NET. The original intention was to develop a router for high traffic networks. After some time, this idea was augmented to other network usages such as network filtering and monitoring. All these functions run on *COMBO6* family PCI cards (see Figure 1.1 for illustration, the photo is taken from official Liberouter web).

This thesis is based on one of many researches that have been already done under Liberouter – *NetCOPE* – which we describe in chapter 2.5. In short, it is a hardware platform trying to achieve universality along with speed and it is fully programmable. Software running on them can do fast transfers if it is instructed to. Drivers are usually used for this purpose. Despite some are ready yet, a new *NetCOPE* software was developed and it needs a new driver.

This thesis describes the evolution process of such driver from its requirements, considering variants to be as fast as possible but still be compliant with hardware specification [9], especially direct memory access (see 2.2) in combination with ring buffers (see 2.5).

After requirements settle down, we model the specification with help of UML and get towards final code implementation in C language (kernel part in GNU flavour, see [13]). As much as possible code parts (all corner cases at best) need testing, we figure out which testing techniques might be applied on drivers and finally comparing with older drivers and card softwares will be done.

Linux In the early stages, we decided to create a driver implementation only for Linux operating system. It is an operating system based on kernel which was established in



Figure 1.1: COMBO6x card

the early nineties of the 20th century by Linus Torvalds. There is over three hundred variants called distributions shipped by either communities or software giants.

Linux kernel in its version 2.6 has no stable application programming interface (API) inside the kernel, but still the API is well-evolved and easy to use, because the development is ridden and checked by a Linux kernel community. We will see consequences of such development method in detailed description of some subsystems later and in compatible layer implementation too.

Major part of Linux kernel is licensed under GPL version 2¹. Chosen license which the implementation will be distributed under is GPL version 2 like most other drivers.

1. License is available at <http://www.gnu.org/licenses/gpl-2.0.html>.

Chapter 2

Hardware Basics

Before we can start writing code, many hardware questions must be answered to know under which circumstances we can achieve concrete goals and what constraints kernel driver implementation has. Understanding hardware-software communication will guide us towards complete implementation.

After each section, we consult what exactly that particular part means for the final code and how to achieve that in Linux kernel.

2.1 Devices Interconnection

The first thing to explain is how data flows from devices to processor and main memory and particularly which types of buses are on the route from a CPU to a device. Overall view may be seen in Figure 2.1.

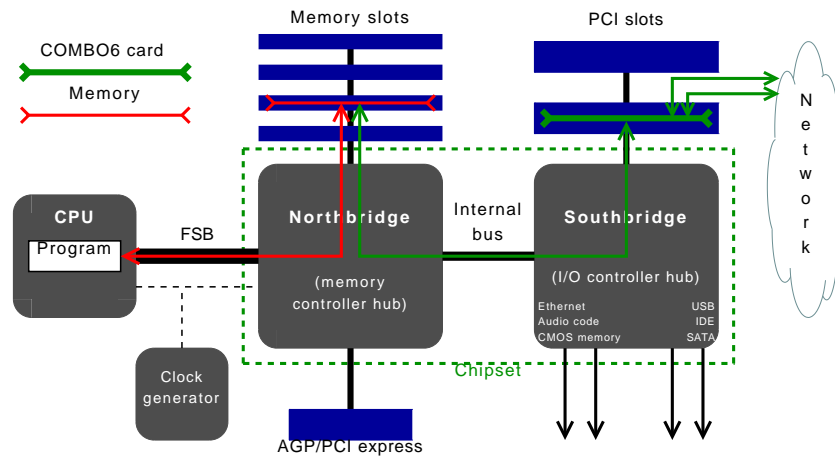


Figure 2.1: Devices interconnection

2.1.1 PCI

We are concerned especially in *peripheral component interconnect* (PCI) devices, because COMBO6 card families are PCI based. Currently we have a legacy PCI (32-bit PCI) COMBO6 card, COMBO6x for PCI-X (64-bit PCI) and COMBO6e based on PCI-Express specification (64-bit packet based PCI). Each such device is physically connected to any PCI bus via PCI slot. PCI bus is attached to *southbridge*, it communicates with *northbridge* over internal bus and it is finally wired with processor via *front-side bus* (FSB) and with memory. Nowadays it is still more common, that all southbridge, northbridge and memory controller are implemented as proper PCI devices directly connected to the main PCI bus.

Older PCI buses have a *control bus* supposed to send controlling signals and synchronize objects which are connected to it and further *address* and *data buses* multiplexed on one address/data bus. A transmission starts by sending address followed by data on the same wires. Legacy PCI address/data bus is only 32-bit wide and hence sending of 64-bit numbers over the bus to older COMBO6 cards is not possible. Unlike data, addresses are normally sent in so-called *single address cycle* (SAC) mode, when 32-bit address is set on the bus and latched by the other side, but it is possible to send even 64-bit address via *double* (sometimes *dual*) *address cycle* (DAC) by negotiating it and repeating SAC twice. The COMBO6 hardware supports SAC nowadays, but we plan to utilize DAC mechanism in the future to address more than 4 GiB of physical memory, see chapter 4.2.

These problems are entirely avoided later in PCI-Express which communicates over a serial physical layer by packets routed by switches and hubs. Each request or reply is a packet which may contain both 32 and 64-bit addresses in the packet header and whatever data in the payload.

Further detailed PCI information is in Legacy PCI, PCI-X[11] and PCI-Express[12] specifications.

Configuration Space Every modern bus supports a mechanism to autoconfigure devices connected to it. Also each PCI device has its configuration space, where all the needed settings are stored or may be set to. This space contains information about who developed the device and which model it is. These data are useful especially in a driver area, where the drivers want to bind only to a specific subset of devices.

In our scope of interest are device, vendor and their subsystem variant IDs contained in this space, because these are mostly merely identifiers designating one concrete device family. It is also helpful in userspace scripts to create persistently named nodes in section 2.4.2.

Linux PCI Support PCI is well supported in Linux since it became a standard. Concrete API is in a scope of the chapter 12 of [8]. In short, a driver's hook is called whenever the PCI subsystem finds a suitable device, initialization is done by functions with a *pci_* prefix, especially device must be enabled and spaces remapped to be usable. Then the device is ready to work. Deinitialization and device switch off is done in driver's remove hook.

Transfers are done by a processor performing standard read and writes over the remapped space, unit by unit. The unit is up to 4 bytes long on 32-bit, on 64-bit up to 8 bytes. These transfers are not too fast and need processor attention.

2.2 DMA and Bus Mastering

Direct memory access known as DMA is a method widely used to speed up transfers in computer engineering. It is based on technique called *bus mastering*. A device which wants to move data from/to memory becomes a bus master and no other device or even processor cannot touch address and data bus. While being a master, it sets address on an address bus and sends or latches data on a data bus. When there are more than one data phases after address phase on multiplexed bus, the transfer is done in so-called *burst mode*.

On older machines, there was merely one device called *DMA controller* (DMAC), which could become a master. Nowadays each PCI device (see also section 2.1.1) may become a master, so that it can access memory directly by its own DMAC¹. As we will see in chapter 2.5, their count is not limited to one per PCI device. An unit which takes care of fair bus accesses by more than one DMAC is called *Arbiter*.

2.2.1 PCI DMA Interface

To become a master on legacy PCI bus, the device asks the Arbiter for a band and after it gets start signal on control bus, it starts sending. First, address is sent, then data part. If there is more data to send, burst mode is used until the Arbiter emits a stop signal. After that signal, the device must finish the transfer of near few bytes in the cache line² and acknowledge stopping. Next device gets start signal from the Arbiter at this moment and starts data transfer. The ancestral device may ask for a band again unless all data have not been sent. This method is implemented in PCI and PCI-X versions of COMBO6 cards.

PCI-Express is a bit different, it asks the other device to get some amount of data in request packet and the device fills reply packet with the data. Sending is similar, the

1. Note that there is no common PCI DMAC, so if the device doesn't have one, it doesn't support DMA either.

2. Its size is configurable through config space.

source device fills the data directly in the request and gets back completion packet. This is obviously implemented in PCI-Express based *COMBO6*.

The transfer is theoretically limited by the other device (including the main memory) boundary³. It must not exceed this limit, the transfer is interrupted by the other side otherwise. Transfers of 1 KiB per one transfer are not uncommon in *COMBO6* case.

2.2.2 Linux DMA Interface

Linux has three groups of DMA APIs. ISA/LPC, PCI and generic. ISA/LPC is used only for old 8237 chips located on old ISA⁴ buses and on today's LPC⁵ buses, neither one is in our scope of interest. See *Documentation/DMA-ISA-LPC.txt* from Linux kernel sources for usage explanation.

PCI interface This interface supports coherent (also called consistent) and non-coherent DMA mappings, both documented in *Documentation/DMA-mapping.txt*. We always must decide which one we should employ.

Non-coherent mappings are often mapped exclusively for a transfer and then unmapped and underlying buffer freed. This is method widely used for socket buffers which are thrown away after successful transmission. We will probably utilize this interface for exactly this purpose out of scope of this thesis, see section 4.2.

The other group are coherent mappings usually mapped at the driver load or the device start and unmapped and freed even after the device stop. Its domain are DMA ring buffers, firmwares used by some devices and executed from main memory meanwhile device operation and so on. This mapping hence perfectly fits our needs and will be implemented in the new module, see section 2.5.

Generic interface It is actually an abstraction of PCI interface. It is not bounded to any PCI device, it may be applied for non-PCI devices, however it obviously doesn't support PCI features completely. API of this type is taken down in *Documentation/DMA-API.txt*.

2.3 Memory Management

This chapter is a short introduction into memory management as used not only in Linux. Memory management in details on Intel processors is described in [5] and also in [4] and chapter 2 of [2].

3. It is 4096 bytes on the main memory with extensions disabled.

4. Industry Standard Architecture

5. Low Pin Count

2.3.1 Physical Memory

In the usual meaning we call main *random access memory* as a physical memory and we usually⁶ have one or more of them plugged in the motherboard. These modules can be addressed by several methods; Linux implements *flat mode*, where each byte of memory is simply addressed by *linear address*. First memory byte has address 0, on 32-bit without any extensions the last possible byte (if there is that much physical memory) has address $2^{32} - 1$.

2.3.2 Virtual Memory

With just physical memory addressing problems arose early. Applications needed large amounts of continuous memory, but it was inconceivable to find such long block after a while system was running. It was caused by process called (*external*) *fragmentation*. Other tasks held memory blocks and there wasn't enough space in between them to return.

Many algorithms were introduced to avoid this behavior, some of them are described in [7] including *buddy system* employed in Linux kernel for in-kernel allocations of larger blocks.

Paging

However, the algorithms only mitigate the risk of fragmentation being hit, we would still have problems with allocations bigger than some constant specific for each allocation mechanism. To really solve the situation with external fragmentation, method named *paging* was introduced. Linear address space is divided into pages of fixed size⁷ that might be mapped to physical memory or disks as swapped pages.

CPU accesses physical memory by virtual addresses with help of *memory management unit* (MMU) which performs transparent translation to physical address. Some modern mainboards contain another unit called IOMMU (MMU for I/O) between a device and the main memory path, so that devices address by logical address too. It allows creating a virtual space even for devices, but it is not widely spread so far so *COMBO6* card developers must not take it for granted.

The paging schema allows us to allocate huge contiguous memory blocks without a need of having contiguous physical memory. In fact there is no need to have as much physical memory as requested. Some memory may be, as written above, backed by a page stored on disk (swapped out) or even have no "physical" page mapped to it thanks to *demand paging*.

6. It's possible not to have this kind of memory by a processor in special computers like *NUMA* (non-uniform memory access).

7. 4096 bytes on x86 with page size extension disabled.

Although 64-bit kernels are able to address up to $2^{47} = 128$ TiB mapped to the bottom of address space on modern processors (the kernel is mapped in the top of the same size thank to 64-bit sign extension, see [6] for reference, with so far unused *memory hole* between them), 32-bit ones are bounded to nowadays real limit of 3 GiB of virtual space per process. The remaining addressable 1 GiB is used as kernelspace. This will be reflected while estimating size of buffers needed for userspace applications along with physical memory constraints for device memory buffers.

Linux memory interface To obtain a buffer smaller than a page is used *kmalloc* function and *kzalloc* returning a zeroed space (buffer filled with zeroes). They are calls of slab allocator (see [4], chapter 8). Freeing is provided by *kfree*.

For larger physically contiguous buffers (consequent pages) *get_free_page* family should be employed – variants for zeroed page(s), for single page, for functions which return *struct page* pointer instead of logical address. Freeing is done by *free_page*. These page allocations might be theoretically used for DMA, its physical addresses could be gained by *page_to_phys* (*struct page*) or *virt_to_phys* (logical address), however this is not clean, because the returned address needn't be device addressable. We should call *dma_alloc_coherent* to get an addressable page. It gets *struct device* as one of parameters where DMA capabilities are written down and so it "knows" which addresses are fine to return. Possibly a wrapper *pci_alloc_consistent* might be used to pass *struct pci_dev* directly. *dma_free_coherent* and *pci_free_consistent* serves as clean-up calls.

vmalloc is utilized for in-kernel virtual contiguous memory allocations. Other wrappers like *vmalloc_32*, *vmalloc_user* and *vmalloc_32_user* exist for allocating buffers addressable by 32-bit addresses and the two latter for buffers intended to be used in userspace. In such allocations are underlying pages marked as *user* and zeroed not to leak any previously stored information, however they are present only in kernels as of 2.6.18. *vfree* frees all these allocations.

For the convenience of COMBO6 being PCI devices, we employ *pci_alloc_consistent* and *pci_free_consistent* for DMA allocations, *vmalloc_32_user* for buffer to be mapped to userspace and obviously *kmalloc* for driver state structures.

2.3.3 Memory Map

As we are able to set (and unset) page present bit in structures MMU traverses and an exception is thrown when the bit is not set while any of the structures is accessed, we can hook a page fault handler to this path. The handler would then install the missing pages.

This can be used for *memory mapping* (MMAP). We register a page fault function by operating system (OS) interface and when an user application maps some space and accesses it, we may set up there any pages we want. According to exact address which

was the root cause of the fault, we compute a number of page which should be in that particular memory place and install it there. Application "sees" this as a transparent process and it accesses the data from the installed page since that moment.

Linux mmap interface Many incompatible interfaces were introduced and removed meanwhile Linux 2.6 development process. First versions support remapping of some sort of memory simply by implementing real fault handler called *nopage*. This function is called for every page fault and must install the requested page (offset is passed in as a parameter) or fail to program be killed. *nopage* hook was changed to *fault* with slightly modified prototype in kernel 2.6.23.

Not to implement *fault* or *nopage* handler for each mapped space, helpers exist. For remapping I/O space, *io_remap_page_range* was used in some variants, because it changed prototype time by time, then *io_remap_pfn_range* was implemented and it is still in use. While for remapping kernel memory obtained by page allocation mechanism (i.e. physically contiguous memory), *remap_pfn_range* may be used, a helper called *remap_vmalloc_range* for virtual memory exists even since kernel 2.6.18 and it remaps exclusively *vmalloc_*user* buffers. We'll employ all these mentioned interfaces, depending on a kernel we compile for, to be backward compatible and to provide complete functionality (and simpleness for later kernels).

Memory Mapped I/O

Memory map is sometimes inappropriately confused with *memory mapped I/O* (MMIO). We are able to access device memory space in kernel by virtual address by MMIO. It is achieved by mapping I/O space into the kernelspace. For example *FPGA* memory space of a *COMBO6* card may be referred by a virtual address after being mapped this way.

MMIO interface Memory mapped I/O is done by invoking *ioremap* or *pci_iomap* for PCI devices. Then, for the former, *read* and *write* are used for device space access. The latter interface is slower, because it calls *ioremap* and all consequent read and writes (functions *ioread*, *iowrite*) are done through calling back *read* and *write* or *in* and *out* depending on what kind is the underlying mapping of. In our case we know, it is mapped memory, so that we use *ioremap/iounmap* and *read* and *write* functions.

We refer to [8], chapter 9 and 15 for further reading about accessing device I/O and memory mapping.

2.4 Character Device

While we are able to pass data from the device to a kernel memory (via DMA) so far, we still need to turn over the data to an userspace application somehow. There are several ways how to achieve that in Linux. Most of them are based on moving data through special files such as sockets, block and character devices.

The basic and simplest idea is to **copy** the data by a processor from the kernel buffer to an user one. It is easy to implement and least error-prone, on the other hand it is one of the slowest known methods and as we need to read high speed data in userspace, it is unusable in our case. Memory map suits in such situation, this eliminates utilization of both sockets and block devices, since neither of them can be mapped, furthermore socket data flow through networking stack. The solely remaining are character devices, introduction to char subsystem internals follows further.

2.4.1 Device Nodes

As an entry point from userspace to kernelspace serve *device nodes*. The existing node in */dev* might be opened and used by userspace application, demanded operations are handled by the driver which has registered it before. It depends on what features are implemented in the driver and what operations are defined as an interface. Handlers for *sdata2* will be discussed in section 3.3 after review of operations done in specification.

If the driver is loaded and the requested node does not exist yet, it must be created to allow access to it. It used to be created manually by userspace program *mknod* or wrapper scripts like *MAKEDEV*, however this was not perfect and mostly causes problems, because of the need of every single node creation and so some dynamic principles described later in this section were introduced.

2.4.2 udev

Userspace device (*udev*) is a dynamic device node creation mechanism with support of running external programs depending on defined configuration rules. It's a successor of *devfs*, a filesystem implemented directly in kernel and mounted on */dev*. Udev mounts only a temporary filesystem (*tmpfs*) on the same directory, but from its name, it does all its work in userspace by listening on a socket for userspace events (*uevents*) from kernel and invoking *mknod*, *modprobe*⁸ and other programs.

For instance the following example shows a node creation in */dev/combo6/* for *COMBO6* device matched by IDs read from PCI configuration space which were sent by kernel in *uevent*.

8. Kernel module insertion program.

```
KERNEL=="combosix*", ATTRS{idVendor}=="18ec" \
  ATTRS{idProduct}=="c058", NAME="combosix/%n"
```

Device node names are random so far – they depend on the kernel PCI cards enumeration exclusively (and it may vary). However we want numbers in names to correspond with each other (e.g. to have *combosix6* and *szedataII6* for one concrete card in PCI slot number 6), so we need to write similar rules to generate a persistent and consistent naming.

Two separate files are needed to achieve that, another one with persistent rules will be created "on the fly". A shell script augmenting both the persistent rules and a rules file for matching the *COMBO6* cards which executes the script are described in detail in Appendix A.

2.4.3 Character Device Linux Interface

The first step to support character devices by a driver is to allocate a region of devices. This was done by *register_chrdev_region* in the past since regions were statically assigned among certain drivers, nowadays is employed a variant for dynamic registration *register_chrdev* which registers 256 nodes immediately or *alloc_chrdev_region* which registers none, but keeps a track of a region being virtually used. These two give a registered region descriptor as a return value and it is then passed to *unregister_chrdev* in cleanup path.

Each character device has its own file operations structure with hooks to be called on each particular event (*open*, *close*, *mmap*...). Entries left blank (except *open* and *close*) means that this concrete function is unsupported. While we pass this structure directly to *register_chrdev*, *alloc_chrdev_region* needs further attention, each incoming hardware device must be added by *cdev_add* with operations as second parameter. *cdev_del* is used for removal.

Although that is everything what is needed from Linux kernel view, this is not enough to support *udev* node creation. Each driver creates a class by *class_create*, frees up by *class_destroy*. Then each device is created by *device_create* and after this call *udev* receives a signal to create a node. The node is deleted after *device_destroy*. Both these functions are created in class namespace, hence the class must be unconditionally created before and freed after *device_** calls. This API evolved even in 2.6 Linux kernel and is probably most unstable one in a scope of this thesis. Many function variants appeared, many changed prototypes and then disappeared during 2.6. Functions like *class_simple_create*, *class_device_create* and others will be mentioned in back-porting chapter 3.3.6.

We choose *alloc_chrdev_region* method, since it is recommended and *device_create* as well. Further character device resources are available in [8], chapter 3 and in [2].

2.5 NetCOPE Hardware And szedata2 Design

As we slightly mentioned in the introduction, all Liberouter projects are based on COMBO6 cards. They contain Xilinx *field-programmable gate arrays* (FPGA) with a PowerPC processor. There was an aspiration to develop an independent platform which would be immutable against fast evolution of networking. This is a goal of *NetCOPE* project, currently built on the top of COMBO6 with network profitable wiring and internal connections. Technical details may be found in Netcope report[9] and, for Liberouter members, also on internal Wikipedia under each link listed on *NetCOPE* specification page.

The project was started in the middle of 2006 and currently we have a functional and tested prototype with 4 network interfaces⁹. Its universal design allows involving it in multiple areas such as network traffic filtering, routing packets, intrusion detection and others – it depends on firmware, so-called *design*, which is loaded into the FPGA.

Schema of *NetCOPE* hardware may be seen in Figure 2.2 (it is sourced from Liberouter web).

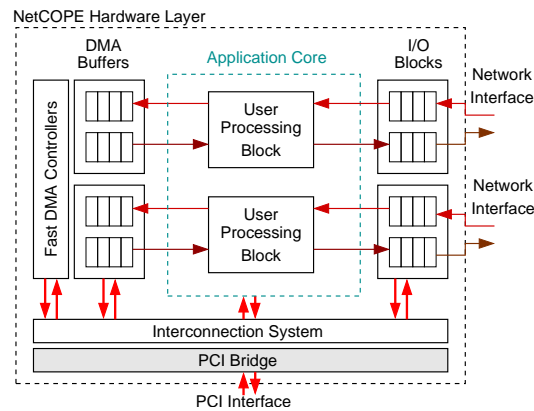


Figure 2.2: Overall schema of *NetCOPE* hardware

2.5.1 szedata2

szedata2 is one of the designs, the abbreviation stands for *straight zero copy data* in the meaning of avoid copying data actively by processor from memory to the application and vice versa. The name was chosen after its predecessor *szedata* version 1.

9. A new hardware with 2 interfaces but with modern bus (PCI-Express) is prepared and tested now.

Design Specification

Both *szedata* versions run in *FPGA* on *NetCOPE* architecture. Version 1 utilizes *PowerPC* as a DMA controller – program running on that processor used to transfer data by *scatter-gather DMA* method (described in [8], chapter 15) over memory block pool. In the first phase, the data transferred at one turn contained only one packet of constant length. While *PowerPC* was working with higher rates in this mode, we found out, that it is a bottleneck of the whole architecture, its control bus was too slow to achieve higher transfer rates.

We tried to improve this behaviour by aggregating more data packets into one transfer block and we redefined data structure too – data are now transferred in a packet of a variable length with modified header in comparison to the older one. We had to redesign also the program running on the card a bit and define the packet structure in the specification.

The packet header includes two sizes: segment size and header size. The former is size of whole segment, i.e. size of the header, padding and data in sum, while the latter is size of optional part of segment containing some control information. These control data are placed directly after the header, optional padding follows and finally the packet contents (payload) with start aligned to 8-byte boundary (thanks to padding). Whole structure is padded at the end again to 8-byte boundary. We refer to this as a *packet of variable length*, its format is shown in Figure 2.3.

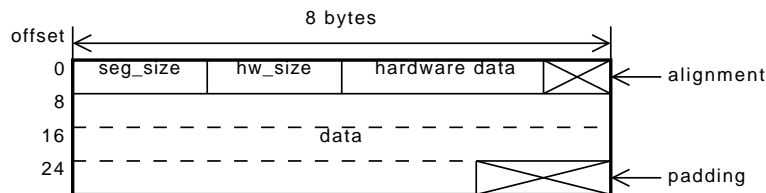


Figure 2.3: *szedata* variable length packet

Despite the improvements, this was still unsuitable due to grown complexity and still the transfers couldn't be performed at estimated (theoretically computed) speeds. We decided to redesign whole design architecture and fork *szedata* project into the version 2. *PowerPC* will be no longer used for DMA transfers, it'll be fully disconnected. Four separate DMA controllers for each interface will be implemented directly in the design instead. By that step we eliminate the impact of slow control bus, but we still have problems with packet transfers, the aggregation is too complex.

We performed some analyses what could be suitable underlying medium for data transfers. Some graphics adapters, several networking (including wireless) and video capture cards use a ring buffer structure with success. We decided to utilize it too.

First, we needed to create a design specification. Let's consider 10 Gbit/s data flow and userspace task latency of 50 ms. We need a buffer of size at least $\frac{10 \cdot 10^3 \cdot 0.05}{8} \doteq 60$ MiB per interface not to throw incoming data away. But as we noted in chapter 2.3, we cannot allocate that much contiguous physical memory. This made us to split the buffer on the page basis and create a mechanism to cope with this.

The idea is to allocate yet another memory for descriptor page with entries pointing to the pages of the interspersed ring buffer. We took in account 64-bit systems, so that the pointers are represented by 8 bytes each. Since we allocate the descriptor memory on a page basis too, it allows us to have only $\frac{4096}{8} = 512$ pointers to ring pages which gives $512 \cdot 4096 = 2$ MiB sized ring buffer. This was not enough, so we introduced two kinds of descriptor entries. One is still a direct pointer to a single ring buffer page, the other one is a pointer to the next descriptor table. To distinguish them, we take to advantage of all page addresses being zeroed in low 12 bits (see [4], chapter 3) by using these few bits for flags. Pointers to the next descriptor table read by the hardware are logically or-ed with 1. We call these pointers *type 1* in Figure 2.4, the ring buffer page pointers are denoted as *type 0*, and for illustration we also connect ring buffer pages with dashed *virtual link*.

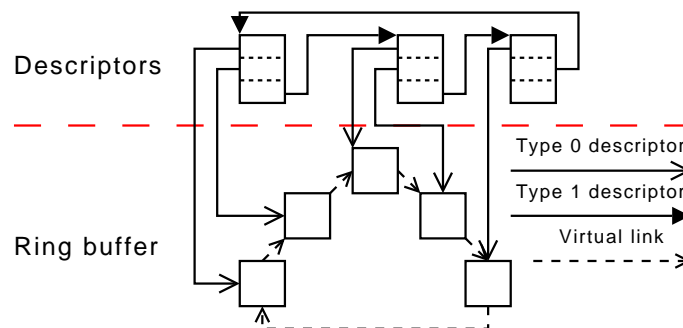


Figure 2.4: szedata2 ring buffer structure

Registers and address space The last thing we need to cover before implementation is how we declare communication channel between the driver and the newly specified design. For the convenience of MMIO, we define design address space as a memory area. Every DMA engine has its own subspace with its controlling registers within this area. Up to 16 engines are supported for future purposes, we use only 2 so far. Each space consist of these register groups:

1. pointer to the first page with descriptors

2. start, end and size registers – ring buffer state pointers, they are current head and tail and size of whole buffer
3. control and status registers – used to instruct the design and check the actual state
4. interrupt registers – timeout, position or both may be used for interrupt generation; the position interrupt is triggered when one of ring buffer registers reaches value stored in position register

As the DMA engine needs to have information about ring buffer before it starts working, register groups number 1. and 2. shall be set prior to sending *start* command to control register 3. Interrupt registers 4. are optional, they may be set and changed on the fly. Concrete steps of hardware initialization and deinitialization will be described in the driver specification in chapter 3.2.

After a successful launch of the engine, we just need to update tail pointers for reception after we process data and head for transmission after we write data to the ring buffer. Any other processing is implemented in the engine and we needn't to care about it.

Chapter 3

szedata2 Driver

This thesis also aims to apply modern techniques in Liberouter development process. We'll use some of agile techniques evolved in the mid-1990s and established in *Agile Manifesto*¹, some basic *unified modeling language* (UML) methods declared as a standard in UML 1.x versions.

Development is scheduled and planned with use of Gantt charts² available in Liberouter *trac* (NetCOPE subsection) system. One task is rigidly dependent on the other so every interested party should finish on schedule otherwise whole project would stall. To make sure we will finish it on time we included fund of time for each task into the Gantt chart.

According to the incremental model[10] we finished the early planning and need to declare requirements. This is performed in chapter 3.1. We then build on it a model in chapter 3.2 and first increment of code itself in further sections also interleaved with testing. Since the requirements and models usually aren't complete (some mistakes and missing features are found) we'll reedit them and will go on with next iteration. These steps will be repeated until we get a complete code.

Finished result must be integrated to the rest of the project into a versioning system and successfully verified to run reliably including performance tests and even after then may be deployed on machines. Also documentation should be provided as a result of development process. This topic is covered in Final code chapter 3.3.9 and further.

3.1 Requirements

We decided to use some methods yet in previous chapters, here we summarize them and add some other requirements we must ensure our implementation will satisfy:

- multiple ring buffer support (per interface and direction)
- correct ring buffers operations (transmit head must not exceed tail)
- ring buffers and their descriptors setup in hardware

1. It is a statement from 2001 establishing principles of agile programming. Freely available at <http://agilemanifesto.org/>.

2. http://en.wikipedia.org/wiki/Gantt_chart

- hardware setup (another initialization, start, stop)
- export/remap data by memory mapping
- some controlling mechanism to communicate with an application
- multiple applications support, multiple access to the data
- high performance
- as highest as possible code correctness (successful testing)
- documentation (the more well³ commented code the better)
- abstraction (an userspace library would be good)
- last but not the least: clean code (fulfil kernel coding style)

The sole thing needs discussion is how and when we should allocate space for a ring buffer. Since the impact to the system might be significant due to large amount of memory requirements, we must consider following two variants. Either we can allocate the space right after the driver loads or alternatively even after first application opens the corresponding character device node.

The former variant assures fast application startup, but we might risk that we'll never use the allocated memory just because no user opens the node. Since the driver is about to be used in environments where the application is expected to be executed if the driver was loaded yet, we proceed with this choice by default. Otherwise we would stall on every application start end exit potentially exposed to start-up failures due to momental memory exhaustion. In spite of all the threats, somebody might want the latter behaviour, so we'll let user to choose by adding an option for this.

3.2 Models And Design

As we mentioned earlier, we use UML for modelling. Many UML tools are available for Linux, we chose one named *Bouml*⁴. It can model most of UML defined diagrams, however we'll use just *use case* and *sequence* diagrams. Complete diagrams and their documentation are in Appendix B, this chapter introduces the way how we got those diagrams as a result.

3.2.1 Use Case Diagram

The starting point is an use case diagram. We create only one use case module which will correspond to our driver and so we name it the same – `szedata2`. Actors choice is

3. Avoid over-commenting, well comments are good, over-commented core is worse than non-commented, see chapter 8 of Documentation/CodingStyle document from kernel sources available from <http://kernel.org/>.

4. Bouml is a free (under the terms of GPL) multiplatform project with support of UML including version 2. See its homepage at <http://bouml.free.fr/> for more info.

simple, we have an *OS*, *hardware* and an *application*. OS just *inserts* and *removes* the driver to and from the kernel and furthermore since there are no asynchronous actions doable by the hardware except optional interrupts, the sole use case regarding hardware is *interrupt*.

Insert and remove takes care of initialization, the module may allocate and also free ring buffers depending on *allocate-on-modprobe* option in these two use cases. HW interrupt use case only wakes up applications waiting for this event.

The application is more complicated actor, it opens (and initializes) the device, process data (and communicate) and finally closes (and stops) the device. All the operations must be atomic due to concurrent access, this is not mentioned neither in the specification nor further, because we take this for granted. Fine grained application use cases description follows.

Open And Initialization

We already know the order of operations that shall be done before using the device from chapter 2.5. We create one use case as a starting point and place following steps in it:

1. *open* system call – initialize buffers if it was the first open (not to break yet started hardware), we use a reference counter to assure only one start of the device
2. *mmap* system call – map buffers
3. *select/subscribe* command – select interfaces which are about to be used
4. *start* command – initialize and start selected hardware (allow exclusively one start per interface throughout all applications)

Even though *open* and *mmap* are different calls from the driver view, application “sees” them as one basic operation, so we leave them in one use case. On the other hand, we find out later that we want to allow the application to select and start interfaces on the fly, so we must place points 3 and 4 in separate use cases which gives three initialization use cases in the end.

Data Processing

The hardware design supports full-duplex communication, so we split processing further in two categories – receive (RX) and transmit (TX) and describe them separately.

RX Since hardware is started and it does RX processing fully standalone, we only check buffer tail pointer for changes in loop. This means calling kernel and getting back area with data (or no data if no change). After finished processing on those data, we free

them up to tell the driver to move the tail. This gives us two separate use cases *get_data* and *release_data*, since the processing is in charge of application.

After we gained access to the data, two receive processing methods raised altogether:

- **load balancing** – several readers exist and the data are divided among them. Every single application gets distinct data and do the same operation on them. It is very similar to *single program multiple data*⁵ in processors theory.
- **multiprocessing** – all readers want all data to do different tasks with them. We can approach this to *multiple program single data*.

We implement only the latter so far, but it is possible that we augment the implementation to the former too, see future work in chapter 4.2.

TX Outbound direction is similar in data processing – application asks for area to write to (gets current transmit head), writes data and issues a command to move the head. It corresponds to the two created use cases *request_space* and *put_data*. It differs in access to the buffer, because all applications shares one TX ring buffer and hence only one may access it at a time not to create holes of unused space⁶. We cope with this by setting up proper prerequisites in the use case descriptions.

Security Note Note that each application can access data of any other writer caused by single shared ring buffer per interface. This is known issue but considered non-risky because of deployment area of the hardware. We expect correct permissions on the device node and so allow access only to authorized programs.

Anyway, we consider implementing multiple TX ring buffers (per application) to avoid any potential security violation. In this case, we decide whether this will be implemented by the design or in the driver. The latter would mean data copying and hence slowdown. More investigation of how much speed impact will this have should be performed but it is not trivial and also out of scope of this thesis.

Close And Deinitialization

After open reference count drops to zero we do steps from the initialization in reverse order. The *close_node* use case consists of a *hardware stop* command, *unmap* and allocated memory cleanup. We might alternatively consider placing the *hardware stop* command in separate use case to allow stopping out of line, anyway we must turn off the hardware on last close if the use case wasn't invoked before.

5. see <http://en.wikipedia.org/wiki/SPMD> for reference

6. There is a plan to extend this behaviour to allow multiple writers too, however it has not been finished yet.

3.2.2 Sequence Diagrams

Sequence diagrams are modeled to show up how events happen during time. The topmost event is executed first and then all events below it towards the bottom one. We create two, depicting RX and TX, they will slightly differ in how application locks and unlocks data.

RX sequence diagram We specify better application code flow here. We establish three lifelines, **application**, **module** and **hardware** which obviously match use case actors.

The work starts by *open* message from **application** to **module**. The diagram copies actions described in use case initialization section above, so it is nonsense to repeat the steps again. We just augment the use case specification further by splitting single use case steps into multiple messages here. For instance *mmap* from use case is here as two messages: *mmap status page* and *mmap RX space*. That is because we try to be as much exact as possible to directly turn these information into the real code in the next chapter.

TX sequence diagram We reflect difference of RX and TX showed before in use case modeling here by not conditioning head move on *put_data* message and we also change all RX occurrences to TX and invoke proper messages, indeed. There is no other difference to consider from model view.

3.3 Code Development

Now we have a complete specification to start implementing. However we needn't to start from scratch, we'll describe usable code base. Then we start with a hardware independent part and complete it with the hardware operating last part. Both these parts will be tested separately and after all an encapsulating library implemented.

3.3.1 Combo6 Code Base

We have evolved base common to all Liberouter projects not to reinvent the wheel for each design. These base drivers are responsible for whole card initialization, an interrupt hook setup, loading firmware/design, controlling *combosix* device node which allows access to all spaces of the card and few other tasks. Since we have two types of COMBO6 cards, two card-specific drivers exist: *combo6* and *combo6x*; they are used by the *combo6core* module consisting of common parts of both. Since this module provides encapsulation, design drivers needn't care about which particular card driver is driving the device, it uses *combo6core* interface to call the proper one, so that the access is fully transparent.

One more convenience is exported info in *sysfs*⁷ and *procfs*⁸ filesystems provided by the central driver. This info may be further used by application to recognize module version or tune some values and parameters.

The design driver should define *combo6_device_ops* structure with name, card type, IDs which it supports and hooks for attach, detach and interrupt and pass it to *combo6_device_register* to register it by the core module. Deregistration is then invoked by *combo6_device_unregister* call. Our structure looks like this C code:

```

1 | static struct combo6_device_ops szedata2x_ops = {
2 |     .owner = THIS_MODULE,
3 |     .name = "szedata2", /* name of this driver */
4 |     .dhw = DHW_COMBO6X, /* combo6x card only */
5 |     .exclusive = 1, /* only one attached driver */
6 |     .intfs = 2, /* limit interfaces */
7 |     .id_lsw = 0x41c10300, /* from id */
8 |     .id_hsw = 0x41c103ff, /* to id */
9 |     .id_text = "NIC_NetCOPE", /* design name */
10 |     .attach = szedata2_hw_attach,
11 |     .detach = szedata2_hw_detach,
12 |     .interrupt = szedata2_hw_interrupt,
13 | };

```

Let's explain values in some entries:

- *owner* is set to not allow used modules removal, core driver adds 1 to reference counter of the module and kernel forbids the module removal
- *intfs* are limited to 2 of total 4, since the testing design supports only 2 interfaces so far, whereas the hardware reports 4 present, so we would end up in initialization of non-existent hardware
- *id_** entries are taken directly from the *szedata2* design, 0x41c1 is *NetCOPE* project, 0x03 is major version (0x02 was *szedata1* design) and finally 0x00 is a minor. As we want to bind this driver to all *szedata2* designs, we grab all its potential minors up to 0xff.

We can see, that the structure also contains a designator of *combo6x* card on the line 4. We must set that entry and that means creating two such structures with the same contents but *dhw* to support both *COMBO6* and *COMBO6x* cards.

7. *sysfs* on Wikipedia: <http://en.wikipedia.org/wiki/Sysfs>

8. *procfs* on Wikipedia: <http://en.wikipedia.org/wiki/Procfs>

3.3.2 Implementation

We used *command* term in the specification many times, but we haven't explained the command passing mechanism. Although *ioctl* system call was uttered as deprecated long time ago, nothing better was introduced so far, its kernel interface has been improved and it is still widely used even in new projects. Hence we'll use it for commands passing too.

We split the implementation into two separate files. One is for non-hardware dependent part (*szedata2.c* and *szedata2.h*) and one with the rest depending on the hardware (*hw.c* and *hw.h*). The former will consist of ring buffer computations, initialization and DMA allocations and also character device node operations (incl. *ioctls* and *mmap*). The latter will contain the structure from the previous section and registration to the *combo6* base and all ring buffer operations done on the hardware (get and set head/tail pointers, ring size and descriptors setting, etc.) and last but not the least invocation of DMA allocation routines from the HW independent file.

We can conveniently use this split for testing and early development, because we hadn't a working design while writing the driver. We'll create dummy *hw.c* with few timers and fake ring operations and develop *szedata2* part as specified and test it by that dummy module. We'll get properly working and tested driver by evoking corner cases by this technique. More of this is in the testing section 3.3.4.

Structures

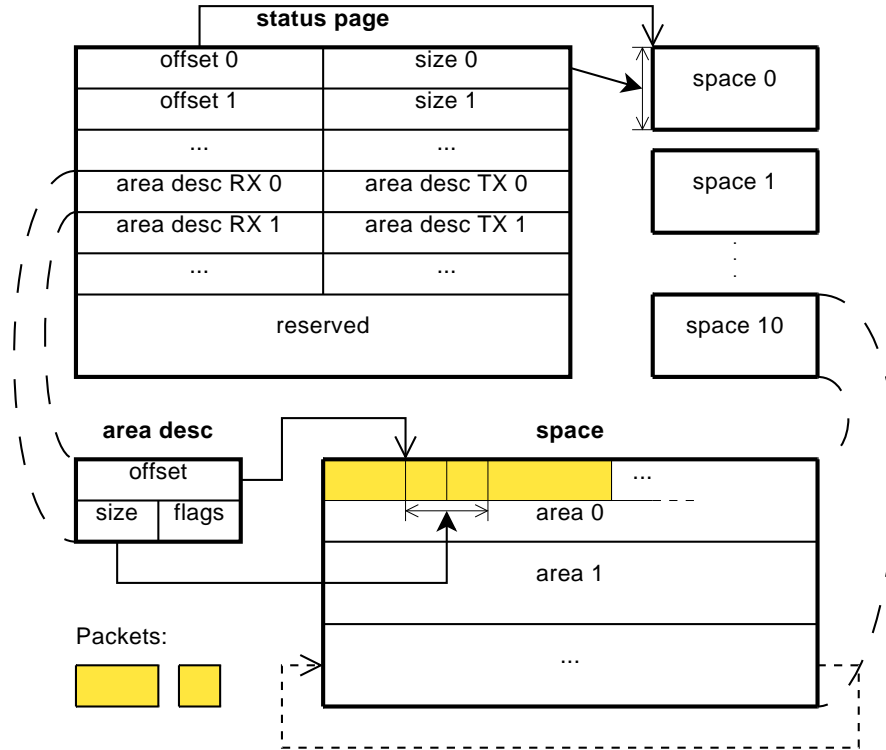
As a first thing let's settle down data structures passed through the system. Many of them raised through the modeling and the module specification, we summarize them here.

Since 64-bit operations are expensive and non-atomic on 32-bit machines, we try to minimize 64-bit computations as much as possible. In fact we need only an *mmap* offset and size to be 64-bit to allow spaces be bigger in sum than 4 GiB on 64-bit or 32-bit PAE⁹ enabled machines. See following structure descriptions and notes about 64-bit needs.

Application-driver interface We decided to pass all read-only information in a status page which is allocated per application on its first open. It is depicted on the top left in Figure 3.1.

First, from the specification, status page shall contain information about spaces to be mapped. According to the manual page of *mmap* and Linux kernel requirements, only two parameters must be known – offset and size. As mentioned earlier, offset is 64-bit value and so the size is. Eleven (indexed by 0 through 10) such pairs might be stored in

9. Physical Address Extension, it extends paging mechanism by one table so that it is possible to address 36-bits of physical address.

Figure 3.1: *mmap* status page structure

the page, however we use only two¹⁰, one for RX space, one for TX. These offsets point to so-called *spaces* (on the right side) which might consist (exclusively ring buffer spaces do) of *areas*. *Area* is directly the ring buffer corresponding to one particular hardware interface in the *COMBO6* hardware case. However we do not call it interface directly, because area-interface one-to-one mapping needn't hold if we recall, that the part which assures the mapping is hardware independent and might be used for any hardware.

The application then remaps all desired spaces indexed by *SZE2_MMIO* constants and may poll for their status in *area descriptors* (other status page entries). It is a structure with 64-bit offset and 32-bit size pointing directly to the area and flags. Flags denote if area/ring buffer wrap occurred and where is the next *area descriptor* for the wrapped data.

There are also sanity checks (magic numbers), version number and set values for e.g. poll in the status page, it is not so important so we let reader to investigate exact

10. Note that it is not space wasting, page size is usually 4 KiB and it is enough space to reserve some entries for future purposes.

descriptor structure. Status page is named as *struct sze2_instance_info* in the source code.

To not poll every area we use commands to get a number of the area ready to be read or written to. Four *ioctl*s corresponding to use cases exist for this purpose. RX lock and unlock are easy to define. We just ask for areas we want (bitmap in one parameter) and unlock all areas without a parameter. The TX direction needs more complicated passing of parameters. Beside the area we are interested in, we need to pass a size too. For locking we define a *struct sze2_tx_lock* containing these two entries, unlocking is the same but the name – *struct sze2_tx_unlock*. All these sizes are sufficient to be 32-bit, since we don't expect somebody to read or write amounts larger than 4 GiB.

The remaining is the area subscription done by *ioctl* too. It contains an array of two members (RX and TX) with info about areas to subscribe and poll threshold (minimal ready-to-process size resuming the application) used for waiting in the driver via *poll* system call. Documentation in Appendix C deals with this interface.

Internal interface We've declared already, that we will split the driver into two parts. Interface between them is straightforward. Interrupt, tail and head pointers passing is needed in one direction, registration and DMA allocation in the other. This interface is described in the documentation contained in Appendix D.

3.3.3 Phase I – Hardware Independent Code

As we don't have fully working design yet, we start with the design independent part. First we write module *init* and *exit* functions. We create a *class* and reserve character device region (with at most *SZEDATA2_MAX_MINORS*¹¹) as described in chapter 2.4.1 and store a number which we got from it to use later in *exit* function when unregistering character device.

We see that we don't allocate any buffers or any other data, so we must export hooks for events when a device comes and leaves, even then we can create and destroy any structures. These operations are yielded to the hardware specific part, because it knows how many areas are needed and how large ring buffers should be allocated.

The registration of incoming device is split into two parts – allocation (*alloc_szedata2*) and registration (*szedata2_register*) itself. After calling the former, callbacks shall be set up and eventually DMA allocations must be done now (or alternatively even on the first open). If we didn't the split, we might end up with kernel panic, because concrete character device is set in the register call and some so far zeroed callback may be invoked already. *szedata2_destroy* is created for cleanup purposes. We must take in account situations when latter register function hasn't been called at all. This is achieved by setting one bit in the register function and test for this bit in the destroy one.

11. currently the value is 8, but it may be altered at compile time, since it is a macro

DMA allocation is performed via PCI interface described in chapter 2.2. Allocation is done per page basis (recall chapter 2.5), so if we consider four interfaces, both RX and TX directions and 50 MiB sized ring buffers, we have $\frac{4 \cdot 2 \cdot 50 \cdot 2^{20}}{4096} \doteq 100000$ pages for 4096 B pages. We need to store physical and virtual addresses for each page (to be able to work with the buffer and also free it later), worst case is on 64-bit machines, where both numbers are 8-bytes, so we further have $100000 \cdot 2 \cdot 8 = 1600000$. This information might be stored in $\frac{1600000}{4096} \doteq 390$ pages. That is a number very close to $2^{MAX_ORDER}^{12}$, so we would end up in failure if we didn't use virtual memory (*vmalloc*) for those descriptors.

Even if we're allocating ring buffer page by page, we might face failures. It is because of how allocations work in the memory management subsystem. For implementation in Linux 2.4 see [4], 2.6 differs slightly, but it is the same in principle. Most of inactive pages is freed even under high pressure, when priority is high. The priority is raised by each failure and the higher the priority is the more pages are tried to free (also the used ones might be swapped out if they belong to userspace). This means we should try to allocate failing pages at least several times to raise the priority with some delay between the tries to let swapper move some pages to the swap device. See *szedata2_alloc_dmaspace* function for the final implementation.

Character Device Operations We passed also *struct file_operations* when registering concrete character device which contains *open*, *release* (close), *mmap*, *poll* and *ioctl* hooks. With use of reference counting we call HW open and close callbacks only once. The rest of *open*, *close* and *poll* (wait for data) is implemented exactly as specified in model by recommended techniques from [8] and later from Linux kernel mailing list.

ioctls need special handling, there are three types of *ioctl* handling¹³. New drivers should use *unlocked_ioctl* and so this implementation. We also implement *compat_ioctl* for 32-bit emulation on 64-bit, however we only call *unlocked_ioctl* from it, because all *ioctl* structure sizes and alignments are compatible. If we didn't implement it, commands wouldn't work, because 32-bit emulation *ioctl* code calls solely this function.

As per specification and requirements in chapter 3.1, commands must be atomic due to multiple application access, hence we use spin locks to avoid race conditions. We use spin locks rather than mutexes, because they are faster and there is not much code in critical sections to be feared of starving caused by long busy waiting loops.

mmap of status page is solved by *remap_vmalloc_range* helper described in chapter 2.3, on the other hand for ring buffer non-continuous memory must be implemented *fault* method.

12. *MAX_ORDER* is a maximum allowed power of 2 to allocate via *get_free_page*, it used to be between 5 and 20 depending on architecture and configuration

13. Explanation is part of virtual filesystem documentation in Linux kernel tree at Documentation/filesystems/vfs.txt

3.3.4 Phase I Testing

After we completed the first increment, we need to do first tests. To check this piece of code we need to implement source which will behave as a fake hardware.

Right on module load code we test forged illegal inputs by fault injection of *NULL* pointers and error states encoded in a pointer (see *ERR_PTR* macro in *linux/err.h*) passed to register functions. Then the first available PCI device (of any type) is used as a parent parameter to allocation function to be able to allocate DMA space. After doing allocation tests, the device is registered along with other more than *SZEDATA2_MAX_MINORS* to test proper error handling. Also another tests as closing used device or destroying opened device are presented in the *szetest.c* file. See *szetest_test_init* function for details.

After the initialization phase, some timers are fired to pretend real hardware work. Ring buffers are filled with random-sized packets with all reachable corner cases such as wraps of packet over page and whole ring buffer, split of packet header over the same boundaries and so on. Functions *szetest_timer* and *szetest_timer1* implement this logic.

To be able to utter the driver works, also reader/writer in user space must be implemented. *szetest2* application is constructed in similar manner like the fake hardware code – to test as much as possible. Some of these userspace tests were ready yet before the implementation itself, because we knew from specification how it should behave and how not. The implementation is further programmed to satisfy all the tests¹⁴.

Tests for all implemented character device hooks are in *szetest2.c*. See for example *test_mmap* which tests correctness of *mmap* by trying to forge invalid values (longer space, wrong offsets etc.). In addition we ought to add tests for commands and reading and writing of "real" data from ring buffer or to it respectively. These checks are implemented in *read_data* and *write_data* in the test application.

3.3.5 Phase II – Hardware Dependent Code

After successful testing of the base, we start a construction of the hardware dependent part. The design was built meanwhile the specification and first part implementation and is fully working now. Note that the design was developed by a Liberouter hardware group out of scope of this thesis, hence we don't mention its development process.

The code we build up here will be a bridge between the currently finished *szedata2* core and the *combo6* base. To be able to cooperate with the hardware behind the base, the driver needs to know constants understood by the hardware. They are described in the hardware specification and are transcribed to a header file named *hw.h*.

Then we start building the source file beginning with *init* and *exit* functions. The decision is so far to create a single module, so both module parts will be linked into one

14. This agile method is known as *test driven development* (TDD), for reference see http://en.wikipedia.org/wiki/Test-driven_development.

loadable object file. It means we can't add another *module_init* and *module_exit* functions in this part and we must solve it by calling those functions from the former code's init and exit functions. Their contents is straightforward from section 3.3.1, we simply take care of *combo6* operations registration and in addition do module parameters alignment.

block_size is aligned to whole page and *block_count* to nearest power of two, because the ring buffer size is a multiple of these two numbers and the design expects buffer size to be a power of two¹⁵.

Even when the attach function from the operations is called with real attached device, we might allocate a *szedata2* device, set callbacks up and register it (all this is *szedata2_hw_attach*). We also allocate a space for descriptor memory information storage just by passing size of this information to the *szedata2* allocation call.

Everything what *open* and *close* callbacks do is allocation and free of ring buffer if it isn't done at init and exit phases (the allocate-on-modprobe option from specification). Common functions *szedata2_hw_alloc/free_dma* are called from both code paths to not copy the code.

The hardware initialization is done even in *start* callback. We use *combo6* core wrappers for 32-bit writes to PCI bus, so that 64-bit descriptor must be send in two rounds, all other values to send are 32-bit. After posting start to control register, the code inside the command waits until status register is either running or idle then the *start* callback finishes and returns. It is possible, that hardware is locked up and it doesn't become ready in a reasonable time, so we must bound the waiting loop by some constant. According to a design specification, 100 μ s should be enough.

Stop callback is implemented in similar manner, except the fact we wait one whole second until DMA transfers get completed. When this timeout expires, we only report failure to logs and go on, because we don't have any chance to tell the kernel, that *close* function (which issues this callback) failed.

Pointer and interrupt callbacks are simple calls of PCI space reads and writes, a problem arises when computing sizes of spaces between read head and stored tail. When tail equals to head it is impossible to distinguish if the buffer is empty or full. Few techniques with less or more overhead exist¹⁶. "Always Keep One Byte Open" method is implemented in the design, head and tail never equals in both cases, if and only if the buffer is empty. This is not a problem for receive pointers in the driver, hardware never sets them the same, it will ever use *size* - 1 at most. But we must take care of transmit size, because we move the head and we mustn't reach the tail. This is done by this code in *szedata2_head_tail_size*:

15. It's formally correct to assume the multiple is a power of two, since page size is a power of two too and $2^a \cdot 2^b = 2^{a+b}$ from algebra.

16. See http://en.wikipedia.org/wiki/Circular_buffer for description of each method.

```
if (head == tail)
    return size - 1;
```

3.3.6 Phase III – Back-porting To Older Kernels

So far the implementation has been run and tested exclusively on later kernels (2.6.24-rc8-mm1¹⁷ and newer) and since we want the driver to be used on many configurations, we have to support all Linux 2.6 kernels in the best case. This is not feasible without a help of a programmer. Several APIs have changed during 2.6 kernel development as we mentioned in *Linux interface* sections earlier.

For the convenience of modularism, we needn't to care of IRQ handler or MMIO mapping changes, since we caught the changes in the *combo6* modules already far before this design was decided to create. We have a file named *kocompat.h* where compatibilities are solved by testing for kernel versions and using proper interfaces. However we must solve *mmap*, *ioctl* and *class* changes for all drivers, because changes in them are too new and unresolved.

While we were porting drivers from one versioning system to the another one in the past, we created two trees with drivers. One is *kernel* with sourcecodes prepared for inclusion to mainline kernel and the another is *build*. This is where we locally build the drivers and where another compatibility stuff lays. It has the same hierarchy and file names except the files contain only *#include* line to "load" the contents of the real file from *kernel* subtree. We'll use this split for compatibility handling, because this issues need to be solved in the *build* tree only.

Mmap After some investigation¹⁸, we found out, that *.fault* handler was introduced in kernel 2.6.23 and previously used *.nopcode* was removed along with that change. Also 2.6.18 was the first kernel, where is *remap_vmalloc_range* available for public use. This gives three variants:

- **before 2.6.18** – we implement *.nopcode* handler for each space
- **between 2.6.18 and 2.6.23** – only non-contiguous spaces need *.nopcode*, i.e. ring buffer; *status_page* is mapped by helper
- **after 2.6.23** – ring buffer is switched to use *.fault*

The code for older variants is obviously located only in the *build* tree in the *szecore2.c* file with function prototypes before *#include* line and implementation after it. Unfortu-

17. A release candidate with Andrew Morton's patch applied.

18. The most powerful tools for such kind of research are *cscope* to see, where the symbol is defined, *git-blame* to figure out when and by what commit was each particular line (and function) changed and *git-show* to see the patch which does the change. Finally *git-describe* on the particular commit gives directly a kernel version the patch was committed after.

nately *.fault* code in *kernel* tree must be placed in conditional section too, but the test are easy to remove right before posting to mainline.

Ioctl *unlocked_ioctl* and *compat_ioctl* were introduced even in 2.6.11 kernel. Before that, merely *ioctl* existed. We behave similarly as for *compat_ioctl*, simply call *unlocked* variant with proper parameters. It is possible thanks to difference being just in function prototype. Again this handler is present only in the *build* tree.

Class With refer back to section 2.4, we already know this is most variable API in 2.6. Let's summarize what happened during 2.6 development in the following points:

- **from 2.6.2 to 2.6.13** – *sysfs* filesystem created and *class_simple* defined in 2.6.2; *class_simple_create* and *class_simple_device_add* are used (with their destroy opposites) to operate with classes and devices
- **from 2.6.13 to 2.6.15** – *class_simple* disappeared with all its functions, *class* structure which was not "simple" enough before is simplified and used instead. *class_create* and *class_device_create* are functions of our interest in these two kernel releases.
- **from 2.6.15 to 2.6.18** – *class_device_create* changed prototype, it may have two parents reference in it (one *device* and one *class_device*).
- **after 2.6.18** – *device_create* should be used instead of *class_device_create*. The older *class_device_create* is going to be removed soon with *class_device* which is considered deprecated nowadays.

We handle this incompatibilities in *kocompat.h* header by testing which kernel release we are building against and providing correct function calls and structure definitions.

3.3.7 Phases II And III Testing

In fact, there isn't much new to test by our testing techniques thanks to test being ready before the driver programming. Only tests showed in phase I must pass again both after phase II completion and now. The tests should show no regressions on every last kernel we have available for phase III tests though. However we might also use other techniques to catch bugs.

Next steps usually lead to a static analysis of C language. Over hundred both commercial and free tools for this purpose exist¹⁹, one of the tools, *sparse*, was designed directly for Linux kernel driver development needs. The checking is included in kernel

19. Non-complete list of checkers is on Wikipedia at http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.

makefile system also, *make C=1* will invoke *sparse* checker ever before each file compilation. The output is printed on standard error like usual compiler warnings and errors. We caught many signedness²⁰, cast and I/O operation bugs thanks to it.

The Linux kernel itself provides many options for development under *Kernel hacking* configuration submenu. Most these configurations slow down kernel operation rapidly, but they still may be handful while seeking for deadlocks or missing lists initialization for example. All these options enabled allowed us to catch some section mismatches (problems when one function may call another from distinct, potentially already freed section and thus kernel crash may occur), unchecked return values of functions which must be checked and also it helped us to prove locking correctness.

Sometimes we hit a design bug, because it is often very hard to emulate high traffic in simulations performed by hardware developers. This bug is in such case forwarded to a concrete VHDL developer via *trac* system. After they solve the issue, we re-test and possibly change code if it is misunderstanding of specification details. We cooperated this way when we discovered a bug in stop hardware path both in the design and the driver for instance.

3.3.8 Userspace Library

Even though there is no library for *szedata v. 1*, we consider to have one for *szedata2*. The main point to have it is to not propagate the kernel interface directly to the user, to simplify the interface (get rid of the status page) and make it abstract.

We create a directory for library in versioning system in a software package, integrate with existing build system and create a file with where the library heart is going to be. The function bodies are exact copy of use cases from the specification, since the use case alone means exactly that it is a separate sequence of commands not being able to split further. If we carry over each use case contents to a library call, we reach the two aims from previous paragraph. Above all, if we create a shared library, we'll get easy to update part (module) of each application.

If we look closer at the use cases, we'll see, that most of the library code may be extracted from *szetest2.c* testing code thanks to its construction – we thought of creating a library from it later, so we wrote the code modularly. We just take away testing stuff and place atomic (from an use case view) commands in one function call if it hasn't been done already. Then this interface is exposed by library header file *libsze2.h*, compiled and linked into *libsze2.so* for further usage.

Testing Since we changed *szetest2* only internally, the tests in it must work even now except the processing is delegated to the library. Its output should not change anyhow

20. <http://en.wikipedia.org/wiki/Signedness>

and it is checked in that manner.

Also the userspace application may be checked by some static code analyser. Even *sparse* may be used for that purpose, but also *lint*²¹ successors such as *splint*²². This helped us to fix some mostly coding style and non-intrusive errors.

3.3.9 Final Code

With a use of an iterative extreme programming technique and test driven development we get to a phase where everything works correctly as far as we can test and prove at best. If we get back to the requirements established in chapter 3.1, we find few so far executory points in it. In fact we only didn't mention we already fulfilled some of them.

Coding style is taken for granted, we took it in consideration while writing the code. *Checkpatch* – one of the kernel scripts checking fulfillment of kernel coding style – passed when checking the code.

Also we didn't forget to document the code for easy reuse or possible changes by other developers. Userspace (library) code documentation is provided through *doxygen*²³, generated text is available in Appendix C. Documentation of kernel code is done via *kernel-doc* facility²⁴, see generated documentation in Appendix D.

Speed (throughput) of the implementation is one of the most interesting areas in this thesis, we'll cover it in the next section. Further deeper testing of final code available in subversion repository is turned over to testers group.

3.4 Measurements And Comparisons

Now when we have all drivers ready for the real operation, we are able to realize some performance tests if the implementation improved the throughput as expected. However, we do only basics measurements in the scope of this thesis, complex tests with reference numbers will be measured by Liberouter testers group, since they have appropriate tools and enough knowledge how to test throughput in a predicative manner.

Also we should keep an eye on a stability while performing the tests, since this sort of traffic may provoke situations which we are not able to evoke in any other way. These measurements may be helpful in finding so far hidden bugs and we found out, that we reveal few in the design and hence we could further improve the stability.

In testing we are interested especially in RX direction, because we have a machine

21. http://en.wikipedia.org/wiki/Lint_programming_tool

22. [http://en.wikipedia.org/wiki/Splint_\(programming_tool\)](http://en.wikipedia.org/wiki/Splint_(programming_tool))

23. A source code documentation generator tool, visit home page with description and documentation at <http://www.stack.nl/~dimitri/doxygen/>.

24. It's a collection of scripts which can generate bunch of formats such as *man*, *pdf*, *html* or whatever can be generated from *sgml* and *DocBook* formats.

which can generate high traffic at the Ethernet and is fully programmable by scripts. *Spirent Communication AX/4000*²⁵ is a broadband test system we use for stress tests. The scripts are available merely for a Tcl shell (*tclsh*), scripts' documentation is available exclusively on internal *trac* system.

We are about to test three distinct packet lengths: 64 (minimal Ethernet frame length according to its RFC²⁶ including header and checksum), 512 and 1518 (maximal frame length) on *szedata* 1 and 2 and compare results with other solutions. Firstly, we try to compare with *Endace DAG* cards intended to be deployed in the same area as our cards with *szedata* design. Secondly, we will do measurement of standard Intel gigabit card working on the kernel networking stack.

Measurements To gather some statistics about incoming data, we must program some utility which would measure time, incoming packets and their volume for *szedata2*. With use of the time, we can count framerate (in packets per second) and also average packet length. Conveniently we might augment *szetest2* application by adding few lines among other yet implemented testing code.

This yields a tool which beside operation with pointers can do some basic measurements and report numbers we may build statistics from. We need also capture traffic with former design, so we augment program for *szedata* version 1, *szetest*, by same changes.

Intel cards need special program for capturing raw *packet capture* (pcap) interface. Conveniently such tool already existed as a one of testers utilities. We need only extend it for per-second counting. Again this may be done in the same manner as previous two cases.

Configurations The *Spirent* is a multiport device. One of its ports was connected to a machine where we performed the tests. The machine is equipped with eight Xeon E5335 cores at 2 GHz and 4 GiB of memory. Both kernel and user environments are 32-bit. A *COMBO6x* card with c610.05.0b *NetCOPE* PCI bridge version is installed in a PCI-X slot. A *szedata2* design from May 12 2008 (minor version 0) is loaded in the FPGA.

The test is performed by sending 10 000 000 packets from the *Spirent* via *tclshell*. The only variable is length. This line starts 64 B sized packets generation on the *Spirent*:

```
sendPackets -bandwidth 100 -port 3 -count 10000000 \  
-IPsrc 192.168.2.1 -length 64
```

25. Visit *Spirent* homepage at <http://www.spirent.com> to see *AX/4000* product sheet.

26. Request For Comments documents are a series of memoranda encompassing new research, innovations, and methodologies applicable to Internet technologies (cited from http://en.wikipedia.org/wiki/Request_for_Comments).

Our changed szetest2 (and for szedata tests also szetest) then gives us results every second:

```
1482421 packets (94875000 B, avg 80 B per packet) per second
```

Note the 80 B per packet average. Every packet is prepended with a header. The header here is 16 bytes. 4 bytes (two 16-bit numbers) are sizes, the resting 12 bytes are medium access control data (previously denoted also as a hardware data). We subtract these values in the test application before adding it to the counters to not obfuscate the statistics.

Intel cards performance was tested on a machine with the same configuration. We just use other Spirent's port. The card itself is an Intel Corporation 82546GB Gigabit Ethernet network adapter connected to a PCI-X slot too.

There is no DAG configuration we tested on. DAG card values are taken from *Endace Capture Performance* document[3] in the following comparison.

Results After we gain the results, we must average the values to have a correct statistics. Computed values are enlisted in the table in Figure 3.2. Their graphic representation then on charts in Figures 3.3 and 3.4. Note the division into two columns. Since we measured both packet sizes and packet count, we might enlist both of them for better comparison. Units are megabits per second (Mbps) and thousand packets per second (kpps).

Packet length (B)	64		512		1518	
	Mbps	kpps	Mbps	kpps	Mbps	kpps
<i>Intel</i>	348	725.0	954	234.9	983	81.2
<i>szedata</i>	96	187.5	770	188.0	987	81.2
<i>szedata2</i>	759	1482.4	961	234.6	987	81.2
<i>DAG</i>	761	1488.1	962	235.0	987	81.2

Figure 3.2: Measurements, table

DAG values are computed as a theoretical limit allowed by Ethernet (see below), since *Endace* claim 100% throughput in their paper.

We see rapid improvement especially when receiving short packets. Performance of *szedata2* in comparison with non-aggregated previous version is several times higher. According to [1], we almost achieved limit of 1 Gbit Ethernet due to frame sizing and overhead of underlying data. Maximum rate for 64 B frames is 1 488 095 (761 Mbps), for 512 B, 234 962 (962 Mbps) and finally for 1518 B it is 81 274 frames per second (986 Mbps).

We can conclude, that we fulfilled performance requirement and so can continue with testing and deployment.

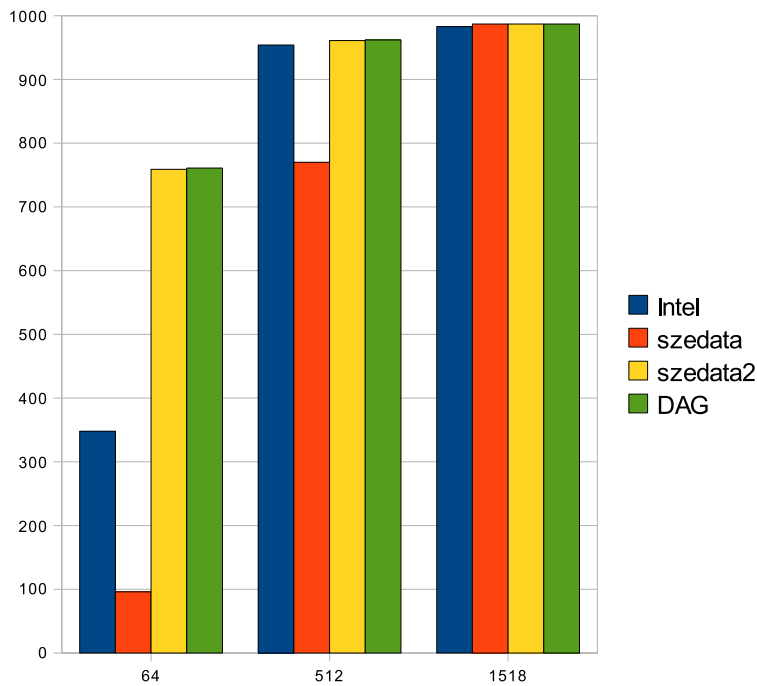


Figure 3.3: Measurements, chart, megabits per second

3.5 Maintenance

Drivers for all designs and card types are deployed on each machine and kernel version after successful tests performed by testers. This must have been done for a first time due to conversion from one versioning system to another and dropping support of kernels older than 2.6. Meanwhile we have been creating this implementation other drivers were moved to the new system with many intrusive changes and it broke compatibility between the two versions from both systems, so they must be tested and distributed at once, now including *szedata2*.

The userspace library is not kernel dependent, so it needn't be necessarily tested on all kernels, however because of testing program dependency on this library, it is tested too. Deployment comprises *strip*²⁷ followed by a copy to a directory shared between Liberouter machines.

Thanks to modularity, further maintenance and updating are easy to realize. If we

²⁷. Binaries are usually freed of unnecessary debugging sections and symbols not needed for running on UNIX systems by *strip* utility (see *man strip*).

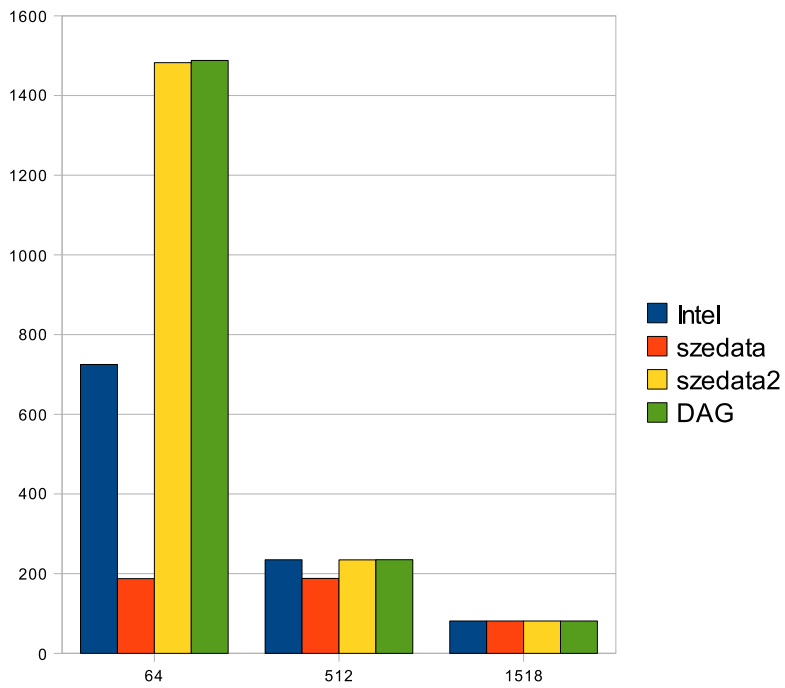


Figure 3.4: Measurements, chart, kilo-packets per second

achieve the state, when all the drivers are from the new versioning system, changing one single module is conceivable by simply removing it and placing in the same directory an updated version either with fixes or some new features. After successful tests indeed.

Chapter 4

Conclusion And Future Work

4.1 Conclusion

This thesis was aimed at data transfers between network and an application in both directions and tried to find out a way how to perform such transfers as fast as possible. As a first step we introduced hardware basics and limitations in fast transfer mechanisms such as DMA on a PCI bus and then walk through possible data structures eligible rapid transmissions and implementation of a ring buffer data structure on *COMBO6* cards.

Also several Linux interfaces for fast operations on the ring buffer were described and we decided to use DMA and memory map which gave us as a result zero CPU time consumed by data copying between hardware and userspace. All this information was instrumental to summarize requirements, model a design and implement the driver with reuse of current *combo6* modules as a base and propose subtle interface between the driver and a library programmed as a part of this thesis too.

We verified functionality during implementation with a help of modern techniques. Each functional increment was tested, some code writing was ridden by tests thanks to requirements and specification being ready before the development itself. After all we were able to do integration testing when we finished the whole module.

Finally, the developed module was compared with other available solutions orienting similarly in network environment. Measurements provided by an external vendor were compared with our solutions and whole package of drivers and library was deployed on Liberouter machines.

4.2 Future Work

Our implementation has been done, however some parts might still be improved. At least four kinds of improvements currently exist, but are out of scope of this thesis. We summarize them in the following points:

- We already introduced idea to support 64-bit PCI addressing in DAC mode in chapter 2.1.1. After hardware being ready to do such transfers, our task would be to add the support to the *combo6* base. Note that *szedata2* driver is 64-bit ready and

so the design registers, what blocks using address higher than 2^{32} is the design as a whole. Its DMA controllers can't do DAC transfers and so PCI core code is told to limit allocations by setting proper DMA mask. Memory management then return pages with physical addresses complying with the address mask.

- It would be worth to have an ability to use the *szedata2* design also in a standard network adapter mode, i.e. with use of kernel networking stack. This needs slight changes of *hw.c* and rewriting whole hardware independent part – a conversion from character device data passing to socket operations.
- Also we know, that we have another option in data processing in the application specific interface. Instead of multiprocessing of incoming data, we would want to balance load among many threads on a multiprocessor machine to do more time-consuming jobs. Remember computation of a ring buffer size, where we considered only 50 ms latencies.
- Further optimizations may be introduced in the area of reading pointer from hardware. Reads and writes to slow (in comparison to the front-side bus) PCI bus may take many hundreds of processor cycles until they finish. To avoid so long delays we might cache pointers for later use for example in *poll*.

Bibliography

- [1] How to test 10 gigabit ethernet performance. Technical report, Spirent Communications, Inc., 2005. <http://www.spirentcom.com/documents/3731.pdf>.
- [2] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, third edition, 2005. ISBN 0-596-00565-2.
- [3] Stephen Donnelly. Dag packet capture performance. Technical report, Endace Limited, 2006. <http://www.endace.com/.../DAGPacketCapturePerformance.pdf>.
- [4] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004. ISBN 0-13-145348-3.
- [5] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 1, Basic Architecture. Intel Corporation, 2008.
- [6] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3A, System Programming Guide, Part 1. Intel Corporation, 2008.
- [7] Donald Ervin Knuth. *The art of computer programming*, volume 1, Fundamental Algorithms. Addison-Wesley Publishing Company, third edition, 1997. ISBN 0-201-89683-4.
- [8] Greg Kroah-Hartman, Jonathan Corbet, and Alessandro Rubini. *Linux Device Drivers*. O'Reilly Media, third edition, 2005. ISBN 0-596-00590-3.
- [9] Tomas Martinek and Martin Kosek. Netcope: Platform for rapid development of network applications. Technical report, Faculty of Information Technology and CESNET z.s.p.o., 2008.
- [10] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, second edition, 2004. ISBN 0-7356-1967-0.
- [11] PCI-SIG®. *PCI Local Bus Specification*. PCI-SIG, 3.0 edition, 2002.
- [12] PCI-SIG®. *PCI Express® Base Specification*. PCI-SIG, 2.0 edition, 2006.
- [13] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*. GNU Press, 4.3.0 edition, 2008. <http://gcc.gnu.org/onlinedocs/gcc.pdf>.

Appendix A

Udev files

Commented file listings of udev rules and a script which is called from those rules follow. This package was developed and tested on udev 120.

A.1 udev rules file

```
1 | KERNEL!="combosix*", GOTO="cesnet_generator_end"  
2 | ACTION!="add", GOTO="cesnet_generator_end"  
3 | NAME=="?*", GOTO="cesnet_generator_end"
```

Since we want to match only additions of combosix devices, rules at line 1 and 2 are needed. To ignore cases where the stored name was set yet, the skip rule at line 3 is there.

```
4 | ENV{CS}="$kernel"  
5 | ENV{MATCHPATH}="$devpath"
```

Set up environment variables used in a shell script later. These variables are filled in directly by *udev* and corresponds to the name suggested by the kernel or a device path (i.e. bus, slot...) respectively.

```
6 | IMPORT{PROGRAM}="csdevname"  
7 |  
8 | ENV{CS_NEW}=="?*", NAME="$env{CS_NEW}"  
9 |  
10 | LABEL="cesnet_generator_end"
```

And finally invoke the script and set up the new name if the script changed it. The last step (line 10) is to define a label where the first 3 rules jump to.

A.2 csdevname script

Shell script **csdevname** called from line 6 above follows:

```
1 | RULES_FILE='/etc/udev/rules.d/70-cesnet.rules'  
2 |
```

```
3 | . /lib/udev/rule_generator.functions
```

This loads udev predefined functions used later. `RULES_FILE` is the name of file, where we store our persistent naming. Variable name is predefined (and used in the script sourced at line 3) and must not be changed.

```
4 | cs_name_taken() {
5 |     local value="$(find_all_rules 'NAME=' $CS)"
6 |     if [ "$value" ]; then return 0; else return 1; fi
7 | }
8 |
9 | find_next_available() {
10 |     raw_find_next_available "$(find_all_rules 'NAME=' "$1")"
11 | }
```

Functions `cs_name_taken` and `find_next_available` are for finding available names for new coming device. Both uses functions defined in the sourced script from line 3, that allows be them as compact as possible.

```
12 | write_rule() {
13 |     local match="$1"
14 |     local name="$2"
15 |     local comment="$3"
16 |     {
17 |         [ "$comment" ] && echo "# $comment"
18 |         echo "SUBSYSTEM==\"combosix\", ACTION==\"add\"$match,
19 |             NAME=\"$name\""
20 |     } >> $RULES_FILE
21 | }
```

`write_rule` takes 3 parameters – under which condition this rule is triggered, static node name and potentially a comment to this rule. The result is written to previously defined `RULES_FILE`.

```
21 | lock_rules_file
22 | choose_rules_file
```

At lines 21 and 22 we lock up the rules file to prevent race conditions (if two writers occurred, it would result in unpredictable behavior) and decide in which file will be the new rules written (since an usual location might be still read-only while booting – `udev` will then move the temporary file to the proper directory after it becomes writable).

```
23 | MATCHPATH=${MATCHPATH%/*/*}
24 | if [ "$MATCHPATH" ]; then
25 |     match="$match, DEVPATH==\"$MATCHPATH/*\""
```

```

26 fi
27
28 if [ -z "$match" ]; then
29     echo "missing valid match" >&2
30     unlock_rules_file
31     exit 1
32 fi

```

Now we define matching rule passed to **write_rule** later (line 46). Sanity check of the value is done here and if it fails we return error.

```

33 if [ "$CS_NAME" ]; then
34     COMMENT="$COMMENT (custom name)"
35     if [ "$CS_NAME" != "$CS" ]; then
36         CS=$CS_NAME;
37         echo "CS_NEW=$CS"
38     fi
39 else
40     base=${CS%%[0-9]*}
41     if cs_name_taken; then
42         CS="$base$(find_next_available "$base[0-9]*")"
43         echo "CS_NEW=$CS"
44     fi
45 fi

```

Here the name, which is returned to rules (**CS_NEW**), is set. Logic is simple, either the one passed by external application is used or a new one is found (first unused).

```

46 write_rule "$match" "$CS" "$COMMENT"
47
48 unlock_rules_file

```

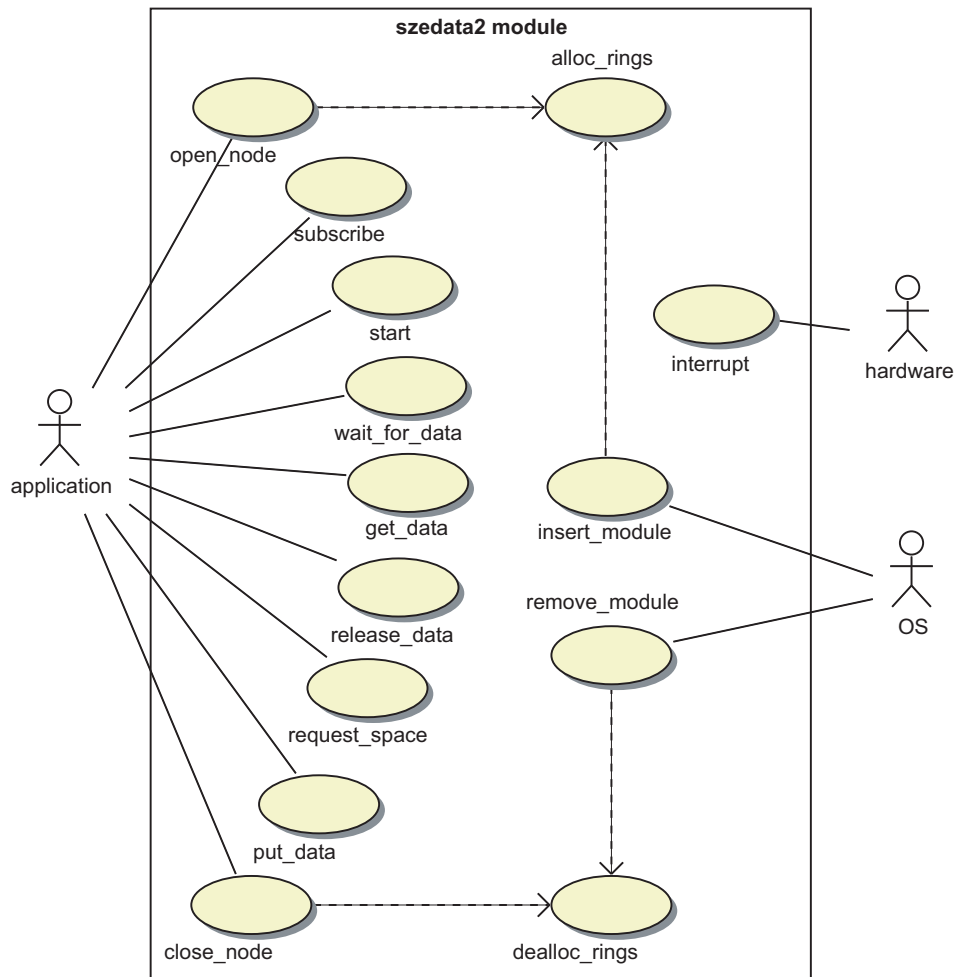
Finally the file with static naming is augmented and unlocked.

Since this moment, if the device come in the system again later, even in other order, system will ever create a node with the same name loaded from the rules file.

Appendix B

Models

Models created using *bouml* are contained in this appendix. First we include the use case diagram with specification of its parts directly after it.



B.1 Use Case open_node

1. the module's device node is opened
2. per application descriptor allocation is done including space for a status structure
3. if (not allocate_on_modprobe and refcount was 0) then call *alloc_rings*
4. node is mmap-ed (offset 0) – status
5. node is mmap-ed with values from previously mapped memory (rx, tx spaces) – note that all buffers are visible as a virtual contiguous memory. The only difference between TX and RX mmio space is read/write protection

B.2 Use Case subscribe**B.2.1 Precondition**

Node must be opened yet and close must not to be in progress

B.2.2 Normal flow

1. subscribe_area ioctl is called with areas bitmap and poll_thresh; both for each direction
2. APP poll_thresh and areas are set to the status page

B.3 Use Case start**B.3.1 Precondition**

Node must be opened yet and close must not to be in progress

B.3.2 Normal flow

1. start ioctl is called
2. each APP rx_tail_ptr and APP rx_head_ptr is set to the current corresponding HW rx_tail_ptr
3. if (refcount was 0) then the device is started – enabling of interrupts, capture start ...

B.4 Use Case wait_for_data**B.4.1 Precondition**

Node must be opened yet and close must not to be in progress

B.4.2 Normal flow

1. poll is invoked
2. while:
 - (a) for all requested rx areas: $\text{difference}(\text{HW rx_head_ptr}, \text{APP rx_tail_ptr}) < \text{APP rx_poll_thresh}$ and
 - (b) for all requested tx areas: $(\text{tx_ring_size} - \text{difference}(\text{HW tx_head_ptr}, \text{HW tx_tail_ptr})) < \text{APP tx_poll_thresh}$
- do: the process is slept and interrupt scheduled
3. return status to the application

B.4.3 Alternative flow

- signal is caught while sleeping → interrupt waiting and return

B.5 Use Case get_data**B.5.1 Precondition**

Node must be opened yet and close must not to be in progress

B.5.2 Normal flow

1. rx_lock_data ioctl is invoked with requested areas bitmap
2. the requested bitmap is logically ANDed with the subscribed bitmap
3. size is computed as difference (APP rx_tail_ptr, HW rx_head_ptr) for each area until size is zero and this area number is stored
4. the size (even 0 is valid) is stored into the status page along with offset obtained from APP rx_tail_ptr
5. offset + size sum is stored into the right APP rx_head_ptr
6. stored area is returned

B.5.3 Alternative flow

- size is zero – invalid area is returned

B.6 Use Case release_data**B.6.1 Precondition**

Node must be opened yet and close must not to be in progress

B.6.2 Normal flow

1. rx_unlock_data ioctl was invoked
2. foreach area do
 - (a) corresponding APP rx_tail_ptr is set to the APP rx_head_ptr value previously stored in get_data
 - (b) HW rx_tail_ptr is updated to the $\min(\text{APP rx_tail_ptrs} > \text{HW rx_tail_ptr})$, if there is no such value, then use $\min(\text{APP rx_tail_ptrs})$ of corresponding area pointers. This is needed to catch pointers actually higher than the others, but still belonging to the slowest readers (head usually wraps around and continues from the entry number 0).

B.7 Use Case request_space**B.7.1 Precondition**

Node must be opened yet and close must not to be in progress. Application must have this area subscribed for tx. TX for requested area is not in progress (only one writer at a time).

B.7.2 Normal flow

1. request_space ioctl is called with size and area parameters
2. size is computed as $\min(\text{max_tx_request_count}, \text{requested size}, \text{tx_ring_buffer_size} - \text{difference}(\text{HW tx_head_ptr}, \text{HW tx_tail ptr}))$
3. the size and HW tx_head_ptr as offset is stored into APP status_page

B.8 Use Case put_data**B.8.1 Precondition**

Node must be opened yet and close must not to be in progress. request_space was called by this application for area in parameter.

B.8.2 Normal flow

1. put_data ioctl is invoked with filled size and area where the data was filled to
2. $\min(\text{size computed in request_space}, \text{supplied size})$ is added to the HW.tx_head_ptr

B.9 Use Case close_node

B.9.1 Precondition

open_node was invoked before this call

B.9.2 Normal flow

1. stop ioctl was called
2. node is unmap-ed
3. module's close function is called
4. if (refcount will be 0) a) the device is stopped (it's waited for dma transfers to be aborted/finished) b) if (allocation on modprobe) then null HW ptrs else call dealloc_rings
5. per application descriptor is deallocated

B.10 Use Case interrupt

B.10.1 Precondition

Hardware was started and expected to run (i.e. refcount > 0)

B.10.2 Normal flow

1. interrupt (timeout or ring buffer pointer reached) is raised – ring buffer is in state we requested
2. wake applications, which are waiting for this info up

B.10.3 Alternative flow

- missed interrupt – set up timer (watchdog) to cope with this

B.11 Use Case insert_module

1. modprobe/inmod the module is invoked
2. if (allocate on modprobe) then call alloc_rings

Note: 'allocate on modprobe' is set to true by default, but might be overridden by module parameter at insertion time

B.12 Use Case remove_module

1. rmmmod the module is called
2. if (allocate on modprobe) then call dealloc_rings

B.13 Use Case alloc_rings

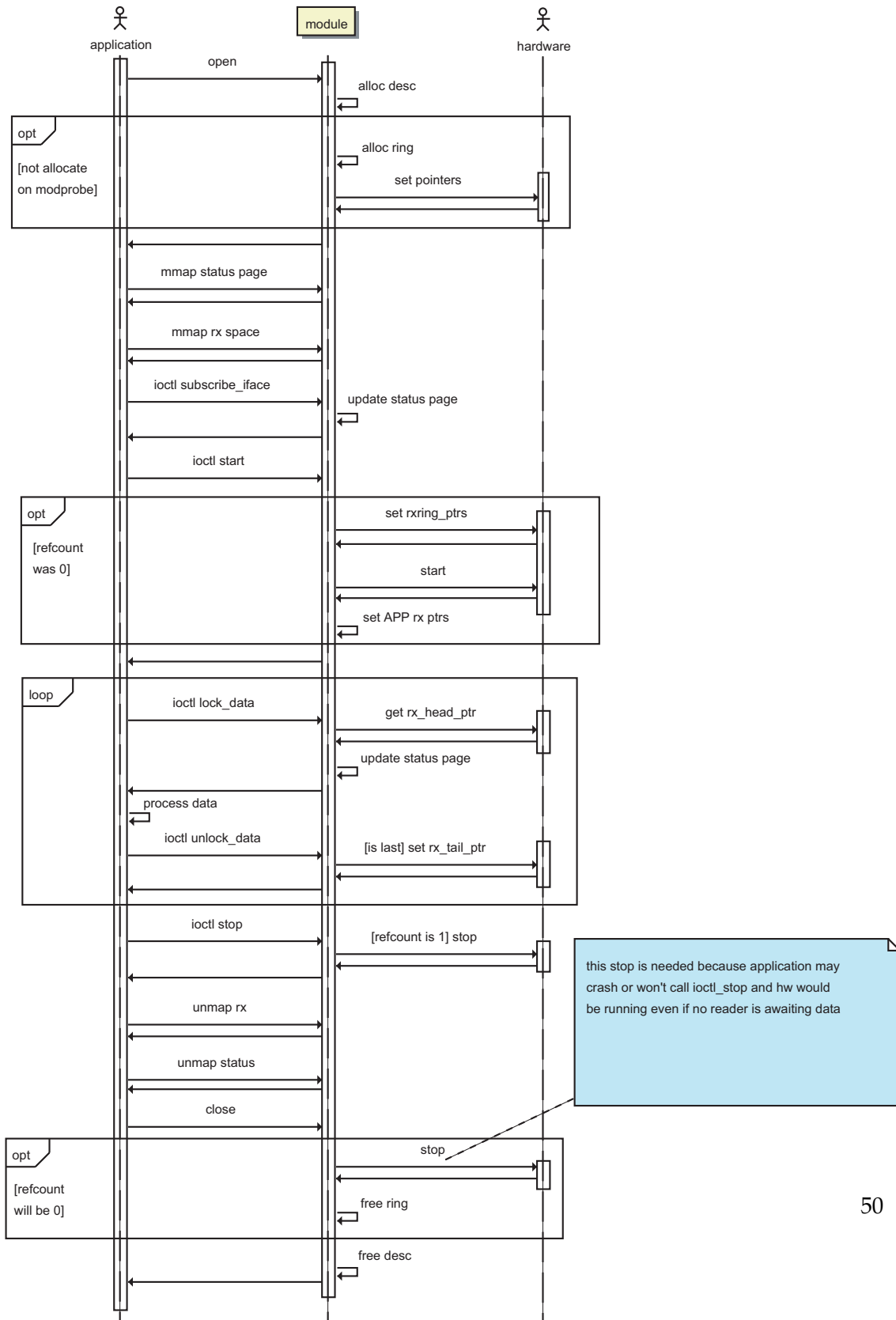
1. alloc_ring is called
2. pages for the ring buffer are allocated, each interface has its own area in the buffer (each interface has its own DMA controller and a ring buffer)
3. physical addresses are put to the hardware page pointers list – the last descriptor points to the first

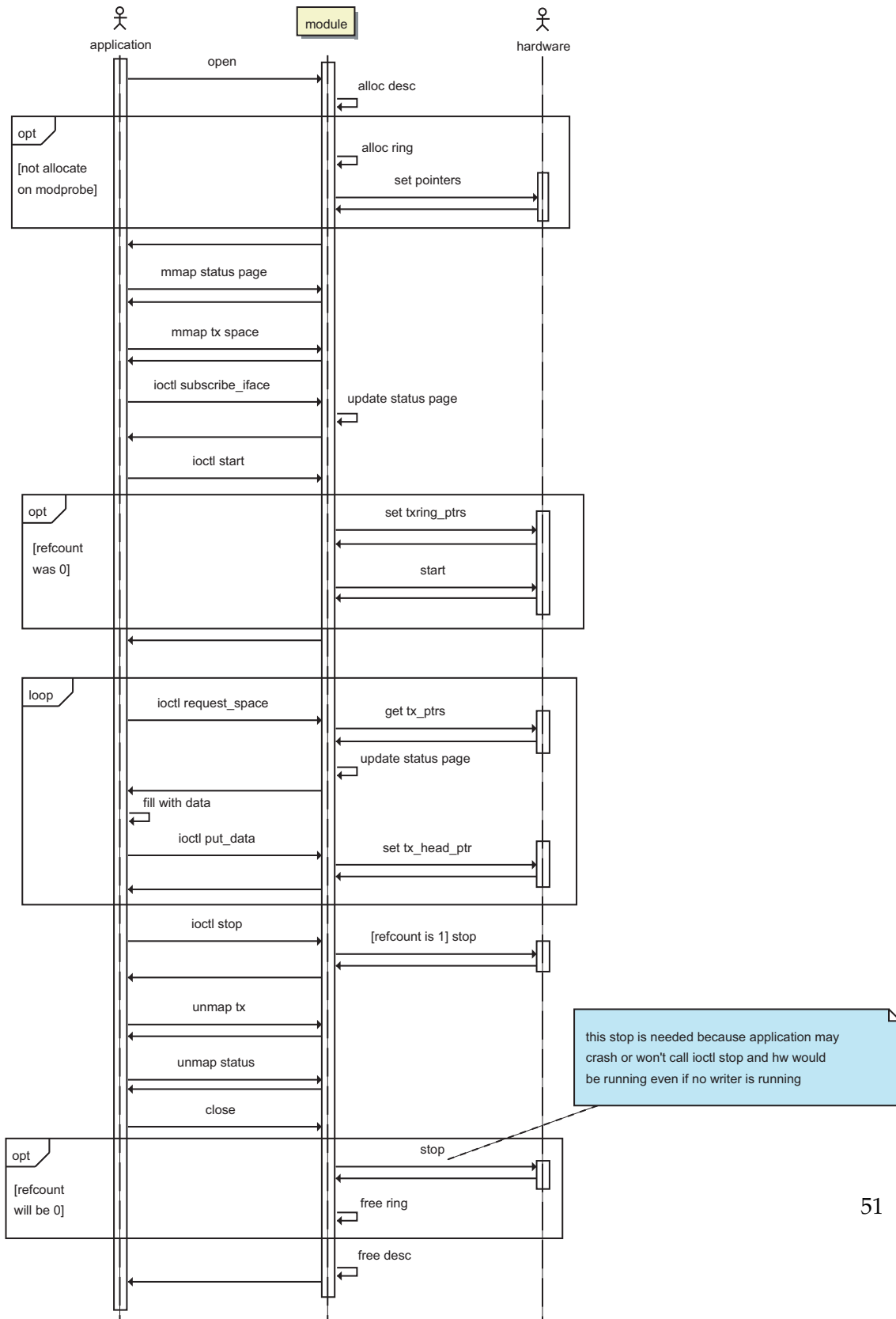
B.14 Use Case dealloc_rings

1. all ring buffer (DMA) mappings and buffers are unmapped and deallocated

B.15 Sequence Diagrams

Now we include sequence diagrams, as the first we depict RX diagram followed by TX one. Remember section 3.2, where we described how we constructed them.





Appendix C

Doxygen documentation

In this appendix we include generated doxygen documentation from the userspace library. All mentioned files are available to see on enclosed CD.

C.1 Class Documentation

C.1.1 `szedata2_packet` Struct Reference

```
#include <libsize2.h>
```

Public Attributes

- `_u16 seg_size`
- `_u16 hw_size`
- `_u8 data [0]`

Detailed Description

`szedata2_packet` (p. 52) - data format of locked area

Data are aligned to 8-byte boundary after header and hw data. Sizes in header are in little endian.

Member Data Documentation

`_u16 szedata2_packet::seg_size`
size of whole packet (header incl.)

`_u16 szedata2_packet::hw_size`
size of hw data (optional)

__u8 szedata2_packet::data[0]

data themselves

The documentation for this struct was generated from the following file:

- software/trunk/libs/libsize2/**libsize2.h**

C.1.2 szedata_lock Struct Reference

#include <libsize2.h>

Public Attributes

- void * **start**
- __u32 **len**
- __u8 **area**
- struct szedata_lock * **next**

Detailed Description

szedata_lock (p. 53) - structure returned from rx/tx lock

Member Data Documentation

void* szedata_lock::start

start pointer of the region

__u32 szedata_lock::len

length of the region

__u8 szedata_lock::area

which area this lock belongs to

Referenced by szedata_tx_lock_data(), and szedata_tx_unlock_data().

struct szedata_lock* szedata_lock::next [read]

next region

The documentation for this struct was generated from the following file:

- software/trunk/libs/libsize2/**libsize2.h**

C.2 File Documentation

C.2.1 software/trunk/libs/libsize2/libsize2.h File Reference

```
#include <poll.h>
#include <linux/types.h>
```

Classes

- struct **szedata_lock**
- struct **szedata2_packet**

Defines

- #define **SZEDATA_POLLRX** POLLIN
- #define **SZEDATA_POLLTX** POLLOUT

Functions

- struct szedata * **szedata_open** (const char *node)
- int **szedata_subscribe** (struct szedata *sze, __u32 *rx, __u32 *tx, __u32 rx_poll, __u32 tx_poll)
- int **szedata_start** (struct szedata *sze)
- unsigned char * **szedata_read_next** (struct szedata *sze, unsigned int *len)
- int **szedata_write_next** (struct szedata *sze, const unsigned char *packet, unsigned int len)
- void **szedata_close** (struct szedata *sze)
- int **szedata_poll** (struct szedata *sze, short *events, int timeout)
- struct szedata_lock * **szedata_rx_lock_data** (struct szedata *sze, __u32 areas)
- int **szedata_rx_unlock_data** (struct szedata *sze, const struct szedata_lock *lock)
- struct szedata_lock * **szedata_tx_lock_data** (struct szedata *sze, __u32 size, __u8 area)
- int **szedata_tx_unlock_data** (struct szedata *sze, const struct szedata_lock *lock, __u32 size)
- int **szedata_fd** (struct szedata *sze)

Variables

- struct szedata2_packet **packed**

Detailed Description

Define Documentation

#define SZEDATA_POLLRX POLLIN

wait for rx

Referenced by `szedata_poll()`.

#define SZEDATA_POLLTX POLLOUT

wait for tx

Referenced by `szedata_poll()`.

Function Documentation

void szedata_close (struct szedata * *sz*)

cleanup everything

Stop hardware if no other is using it. Cleanup userspace (unmap all spaces and free memory).

Parameters:

sz pointer which returned `szedata` open

int szedata_fd (struct szedata * *sz*)

return currently opened filedescriptor (some apps may use their own poll)

Parameters:

sz pointer returned by `szedata_open()` (p. 55)

Returns:

file descriptor on success, -1 on failure (if *sz* == NULL)

Don't use this unless you really know what are you doing. Some crud like pcap needs this.

struct szedata* szedata_open (const char * *node*) [read]

open `szedata` device *node*

Also remaps also all available spaces and does necessary initialization.

Parameters:

node string describing which node to open

Returns:

NULL on failure, otherwise pointer to a structure, which is passed to another functions as primary parameter

int szedata_poll (struct szedata * *szedata*, short * *events*, int *timeout*)

Wait for hardware to be ready

Parameters:

szedata pointer obtained from szedata open

events pointer to events to which wait to and where to return to which of them is ready

timeout how long to wait in ms. Negative means forever.

Returns:

negative on error, zero on timeout, positive when OK

References SZEDATA_POLLRX, and SZEDATA_POLLTX.

unsigned char* szedata_read_next (struct szedata * *szedata*, unsigned int * *len*)

get next packet from szedata

Parameters:

szedata pointer returned by **szedata_open()** (p. 55)

pointer to packet length (filled in func)

Returns:

pointer to returned packet or NULL if no packet arrived or error

struct szedata_lock* szedata_rx_lock_data (struct szedata * *szedata*, __u32 *areas*) [read]

gives back some data, which are ready

If there is no data, NULL is returned. Also if there is internal wrap of ringbuffer (userspace should haven't known about the implementation), it is possible, then 2 structures are returned instead of only one. In that case user should read lock->size from lock->start as usually and then continue with lock->next unless next is NULL. It means ever check lock->next.

Parameters:

szedata pointer to structure from szedata open function

areas bitmap of areas where to data request from

Returns:

NULL on error or no data ready, otherwise pointer to lock structure

int szedata_rx_unlock_data (struct szedata * *sz*, const struct szedata_lock * *lock*)
unlock data which was obtained by rx lock function before

Parameters:

sz pointer to szedata open retval
lock the same as returned by rx lock

Returns:

0 on success, otherwise errno

int szedata_start (struct szedata * *sz*)

start subscribed interfaces

Might be called multiple times, always after *successful* subscribe of some areas.
Otherwise it will fail.

Parameters:

sz pointer returned by sz open function

Returns:

zero on success, otherwise errno

int szedata_subscribe (struct szedata * *sz*, __u32 * *rx*, __u32 * *tx*, __u32 *rx_poll*, __u32 *tx_poll*)

subscribe interfaces to read from or write to

This is usually called after szedata_open to select which interfaces we want to use and how long (for how many packets) wait if there is no data available and poll is invoked. Both per both rx and tx.

Parameters:

sz structure returned by szedata_open
rx bitmap of interfaces to subscribe for rx (filled back)
tx bitmap of interfaces to subscribe for tx (filled back)
rx_poll how many packets should poll block for rx
tx_poll how many packets should poll block for tx

Returns:

0 – success, nonzero – failure

struct szedata_lock* szedata_tx_lock_data (struct szedata * *sz*, *_u32 size*, *_u8 area*)
[read]

lock some space for writing

Driver or library might make the size smaller than requested. Count with this! Also the space might be divided into two logical spaces, check comment in rx lock and fill unless lock->next is NULL.

Parameters:

sz return value of szedata's open function

size how much space is demanded

area in which area are we interested (not bitmap, direct number)

Returns:

NULL on no space was locked, otherwise lock structure

References szedata_lock::area.

int szedata_tx_unlock_data (struct szedata * *sz*, const struct szedata_lock * *lock*, *_u32 size*)

tell the hardware to process written data

Size passed here might be smaller than the requested (or the returned one from tx lock), it means, that the user has written data of only that size in real. This is correct approach and 0 is even correct, it means returned buffer is too small for me.

Parameters:

sz this is what szedata open returns

lock pointer obtained from tx lock

size really written count

Returns:

0 on success, errno on failure

References szedata_lock::area.

int szedata_write_next (struct szedata * *sz*, const unsigned char * *data*, unsigned int *len*)

get next packet from szedata

Parameters:

sz pointer returned by **szedata_open()** (p. 55)

pointer to packet

packet length

Returns:

0 on success, anything else on error

Appendix D

Kernel-doc documentation

D.1 Userspace interface

struct `size2_instance_info`

Name

struct `size2_instance_info` — info mapped at offset 0

Synopsis

```
struct size2_instance_info {
    __u32 magic;
    __u32 reserved1;
    __u64 offsets[SZE2_MMIO_MAX];
    __u64 sizes[SZE2_MMIO_MAX];
    __u32 areas[2];
    __u64 reserved2[5];
    struct size2_adesc adesc[0][2];
};
```

Members

magic 0xde4dAAII (mAjor=0x02, mInor)

reserved1 for future/testing purposes

offsets[SZE2_MMIO_MAX] offset of each mmap space

sizes[SZE2_MMIO_MAX] size of each mmap space

areas[2] subscribed areas (rx and tx)

reserved2[5] reserved for future purposes

adesc[0][2] desc for each area and direction

struct `size2_subscribe_area`

Name

struct `size2_subscribe_area` — `SUBSCRIBE_AREA` ioctl parameter

Synopsis

```
struct size2_subscribe_area {
    __u32 areas[2];
    __u32 poll_thresh[2];
};
```

Members

areas[2] bitmap of areas to be subscribed

poll_thresh[2] minimal size which will wake the application from poll sleep

Description

if areas are set to `~0U`, it means “all available”, corrected value is in mmap space after the ioctl

struct `size2_tx_lock`

Name

struct `size2_tx_lock` — `TXLOCKDATA` ioctl parameter

Synopsis

```
struct size2_tx_lock {
    __u32 size;
    __u8 area;
};
```

Members

size how many bytes should be reserved

area which area to involve

struct `szex2_tx_unlock`

Name

struct `szex2_tx_unlock` — TXUNLOCKDATA ioctl parameter

Synopsis

```
struct szex2_tx_unlock {
    __u32 size;
    __u8 area;
};
```

Members

size how many bytes were really written
area to which area; it must be locked earlier

D.2 Intradriver interface

struct `szedata2_ring`

Name

struct `szedata2_ring` — szedata per ring buffer info

Synopsis

```
struct szedata2_ring {
    unsigned int blks;
    unsigned int areas;
    size_t size;
    struct szedata2_block * ring;
};
```

Members

blks blocks per area
areas areas per space
size block size
ring pointers to ring blocks

D.3 Public functions**szedata2_alloc_dmaspace****Name**

szedata2_alloc_dmaspace — allocate DMAable ring buffers

Synopsis

```
int szedata2_alloc_dmaspace (sd, space, areas, count, size);  
struct szedata2 * sd;  
unsigned int space;  
unsigned int areas;  
unsigned int count;  
size_t size;
```

Arguments

sd szedata2 structure
space RX/TX
areas number of ring buffers to allocate
count number of block count
size block size (must be PAGE_SIZE aligned)

Description

Parent device must be set (on alloc_szedata2) and reference to it is raised by one here.

szedata2_free_dmaspace

Name

szedata2_free_dmaspace — free DMAable space

Synopsis

```
void szedata2_free_dmaspace (sd, space);  
struct szedata2 * sd;  
unsigned int space;
```

Arguments

sd szedata2 structure
space RX/TX

szedata2_get_addr**Name**

szedata2_get_addr — get virtual and physical address of ring buffer

Synopsis

```
size_t szedata2_get_addr (sd, space, area, virt, phys, pos, len);  
struct szedata2 * sd;  
unsigned int space;  
unsigned int area;  
void ** virt;  
dma_addr_t * phys;  
unsigned long pos;  
size_t len;
```

Arguments

sd szedata2 structure
space RX/TX
area ring buffer index
virt pointer to where to store the demanded virtual address
phys pointer to where to store the demanded physical address
pos offset of demanded position
len demanded length

Description

Length of space to the next page boundary is returned (if pos is &-ed with PAGE_MASK, then the returned length should be PAGE_SIZE).

alloc_szedata2**Name**

alloc_szedata2 — allocate new szedata2 structure

Synopsis

```
struct szedata2 * alloc_szedata2 (private_size, parent);  
unsigned int private_size;  
struct device * parent;
```

Arguments

private_size extra size to allocate; pointer to it is set to .private
parent device which this device is bound to

Description

You usually want to call this when new device comes. Then you pass this to register after you allocate some spaces and further initialization.

Parent is needed when you plan to allocate DMA space through this module.

Pointer to new device is returned, or ERR_PTR encoded retval, respectively.

szedata2_register**Name**

szedata2_register — registerr allocate szedata2 device

Synopsis

```
int szedata2_register (sd);  
struct szedata2 * sd;
```

Arguments

sd structure allocated by alloc_szedata2

Description

Call this after you setup (owner, hooks, ...) the structure and, if you plan to allocate DMA now, after you do so (on open alternatively).

Returns 0 on succes, otherwise -errno.

szedata2_destroy**Name**

szedata2_destroy — cleanup szedata2 structure

Synopsis

```
void szedata2_destroy (sd);  
struct szedata2 * sd;
```

Arguments

sd structure to cleanup

Description

It's legal to destroy both registered and unregister structure.

Appendix E

CD Contents And Usage

CD contents is enumerated in these points:

- linux-drivers – drivers including *szedata2*
- software – software branch with *libsze2* library and *szetest2*
- *sze2* – model by *bouml*
- *sze2-generated* – HTML generated model
- *doxygen* – HTML generated *doxygen* documentation
- *kernel-doc* – HTML generated *kernel-doc* documentation
- README – short information about the CD contents
- LICENSE – GPL v. 2 license

All linux drivers can be compiled by a *./svncompile* script and installed by *make install*, both from *linux-drivers* directory. *szedata2* driver is in *linux-drivers/kernel/drivers/szedata2/* and *szetest* dummy module used for testing in the phase I development (see section 3.3) in *linux-drivers/build/drivers/szedata2/*.

Installed drivers may be loaded with standard *modprobe*, e.g. *modprobe szetest2* will load the driver with all its dependences.

When the driver is ready, we can try the functionality with *szetest2* program. It is in the *software/trunk/tools/development/szetest2/* directory, but we first need to compile the library from *software/trunk/libs/libsze2/*. In that place, we run *make* to create a *libsze2.so* object and then we can also run *make* from the *szetest2* directory which will build and link against the library.

Invocation of the test utility must be preceded by setting up proper library path to tell the operating system (the linker) where to find libraries. This is performed by *export LD_LIBRARY_PATH=< . . . >/software/trunk/libs/libsze2/* (see dynamic linker manual page: *man ld*). Then we are able to run the test:

```
$ ./szetest
sub: 1 0
szetest2: poll error
sub1: 0 1
```

It outputs an error, but it is one of our test cases, so this is fine. Two *sub* lines denotes subscription first of RX direction, then TX. When we send some packets, it will output a character every 10 000 received and sent packets and also per second statistics.

The doxygen documentation may be generated in the *doc* subdirectory of library directory by *make*. The kernel one by *scripts/kernel-doc*, see *Documentation/kernel-doc-nano-HOWTO.txt* to see how this can be achieved. Both the script and the documentation are available in the kernel sources (<http://kernel.org/>).