UiO : Department of Informatics
University of Oslo

Discovering and emulating an Intel Gigabit Ethernet device.

Student: Bao Chi Tran Nguyen , Supervisor: Knut Omang

Master's Thesis Spring 2017

Preface

Thanks to Aril Schultzen, Marie Ekeberg, Thomas Plagemann, Morten Rosseland, Linda Sørensen, Esra Boncuk, Torstein Winterseth and David Huynh for their help.

Contents

List of Figures

List of Tables

Abstract

This thesis presents the problem of discovering how the Intel Gigabit Ethernet device, abbreviated *igb*, works and implement it to reach to goal of a *Single-Root I/O virtualization*-capable device. The *igb* is a Gigabit network device card made by Intel used in servers. It is similar to its older card known as the *e1000*. A method is used to discover the *igb* device's design, and compare it with the older *e1000* device to discover useful differences that can be used to create a working implementation of the *igb* device. Many differences and new registers were found, which let us to build an empty framework shell that could load the *igb* driver, but not operate it properly due to lack of handling descriptor queues. Further work could be done to add that feature.

Chapter 1

Introduction

Virtualization is a technology that lets users run multiple virtual machines on a real physical machine. It is marketed to organizations as a replacement of hard physical computers. The technique lets organizations run old existing software on new computers and devices. It makes computers become more high-level and abstracted from the users and the programmers themselves. Tasks that were once done on a single dedicated machine are being moved on to single or multiple virtual machines sharing real hardware resources. This technology is used by companies to combine servers together in a "cloud" environment to reduce maintenance cost. Server providers use it to rent their dedicated servers to multiple customers by combining their hardware thus sharing its usage between virtual machines. This ensures cheap, simple, reliable and secure servers protected by the boundaries of a virtual machine. Keeping the provided servers virtual gives both the customer and the host the ability to simply make an exact copy of the virtual machine to an image file, thus making it easier to migrate its state to other server machines.

Due to the increasing popularity of the technology's use, manufacturers began to design specialized hardware made for this task to increase the performance of running those virtualized computer systems on top of real physical hardware. Both Intel and AMD extended the instruction set of their CPUs to add new instructions that is used by virtualization software to speed up their performance of doing these virtualization tasks. This idea of making specialized hardware for this kind of task was not just reserved for the designers of Cups, but later the designers of I/O-devices took the idea and added virtualization capabilities to their I/O-devices.

Thus a group of computer component vendors called *The PCI Group* ratified a new capability to their bus protocol known as the *Single-Root I/O Virtualization* for use by virtual machines. This new capability lets a physical device virtualize itself, letting them be shown as a single virtual device for each virtual machine, thus making it possible for multiple virtual machines to efficiently share a single physical device. After the capability was ratified, Intel then designed a new network device card called the

82576 that utilize this new ratified capability to increase the performance of handling network traffic made by multiple virtual machines sharing the same network device.

One application that exists that virtualizes computer systems is called *QEMU*. It is a program that emulates a computer system, which means simulating an interconnected CPU, RAM and I/O-devices. The QEMU application can emulate a multitude of computer architectures and peripheral devices connected to the virtual computer system. QEMU can run on many different types of platforms including the x86, PowerPC, MIPS and ARM architectures utilizing their virtualization support provided as an extension by the host computer system. QEMU is implemented by a group of open source developers across the world.

One of QEMU's abilities is to emulate a 82540's series network card, which is known as the *e1000* device in the Linux community. That network device card is used by default when executing QEMU to set up a virtual machine that virtualize a x86 computer system. The virtual *e1000* device gives the virtual machine the ability to communicate with other computers on a network. The supervisor of this thesis made some modifications to this implementation by adding the SR/IOV-capability to it, which gave it the ability to create virtual PCI-Express devices for the virtual machines to utilize for increasing their performance by skipping a layer of indirection. He then renamed the device's name to *igb*.

Since the QEMU PCI maintainer wanted to see a fully working example device, he would not accept his changes[28].

I was given the task to fix this by comparing the differences between the *e1000* device and *igb* device, and make the necessary modifications to the given *igb* device framework to provide a working emulated *igb* network device for the QEMU application.

1.1 Motivation

The reason to emulate this network device is to demonstrate the patches made by my supervisor that adds the SR/IOV-capability to the PCI-Express subsystem of QEMU and be used to help develop a driver for the *igb* device without the need to have the actual physical device available and installed into your workstation.

During the course of software development, a software developer needs to test his programs by executing it on the particular device he is developing for. Sometimes, acquiring this particular device is costly, hard to get or being unavailable making it harder for the developer to run his written program for that particular device he is developing for. This is where virtualization techniques, also known as emulation, solves the problem. They ease the software development process by providing a way to execute programs in a simulated computer system environment. The

environment simulates the needed device, thus removing the need to have it physically available during the software development process.

Hopefully, at the end of the thesis, I will be able to to provide an example of a working *igb* device with the SR/IOV-capability that uses the SR/IOV-emulation patch set[22] provided by my supervisor.

This implementation is intended for developers that develops a driver that utilizes the SR/IOV-capability of this device. Having that tool, they can demonstrate programs that use this capability in their projects without needing a real physical device installed that supports it.

1.2 Problem

The problem of this thesis consist of implementing a device emulator for the Intel Gigabit network device[14], abbreviated as the *igb*, in QEMU. This PCI-Express network card is one of many devices that supports the SR/IOV-capability. One theory explored and proposed by my supervisor is that *igb*'s design is based on the *e1000* device. We believe that the differences might be small enough making it feasible to reuse the *e1000*'s device code found in QEMU to emulate it. After implementing the emulated *igb* device by reusing the currently available *e1000* device code, we could add the SR/IOV-capability to the device emulator code using my supervisor's patches to demonstrate SR/IOV-emulation.

Implementing a working implementation of that adapter card could potentially be made by reusing most (hopefully all) of the source code for the emulated *e1000* network device, and then implement the new extensions that Intel added to the *igb* device to emulate that network device card.

The programming part of this thesis will be to take the given framework and the publicly available specification for the *igb* network device from Intel together with its device driver for Linux, and use them to implement a working emulated *igb* device that supports the SR/IOV-capability for QEMU.

At its current state, the *igb* driver cannot execute through the process of initializing the *igb* device, so discovering the cause that hinders it to do its job is necessary.

The end goal is to make the unmodified *igb* driver for Linux to execute and control the virtual *igb* device in QEMU, and make it transmit and receive packets to and from the network without any noticeable problems.

I was only given the *igb* Linux driver, *igb* device framework and the manuals of the *igb* device. I did not have access to any real *igb* device to inspect and test my ideas on, which means I have to treat the device as a hidden black box. I did have the opportunity to send driver patches to test my assumptions about the device to my supervisor, but due not knowing what to change to pinpoint the bugs and making sure the patches I made

was correct and installable for him, I did not use it. I had no way to verify my patches for correctness, which I had to ensure since he told me that loading those patches onto the computer that has the device installed was time-consuming, and his time was limited.

1.3 Methodology

The method used to develop this device emulator was to follow both the driver and the device's code execution printing its state to the terminal along the way. The new knowledge gained from this process is then used to implement the device emulator code. The official manuals for both devices was consulted to clarify unknown behaviour.

1.4 Outline

Chapter 1 introduces the reader to the world of virtualization and the reasons to do it.

Chapter 2 gives an overview of the computer systems, I/O-devices and software given and describes the details of them.

Chapter 3 describes the given unfinished framework made by my supervisor that pretends to be the *igb* device, but does not behave as it yet.

Chapter 4 describes the goal of this project, which is to end with a working *igb* device that can operate properly with the SR/IOV-capability.

Chapter 5 presents the method I used to discover and implement the *igb* device.

Chapter 6 describes the tools and the environment I am working with to reach the goal. It shows my computer systems that I use to test and implement the *igb* device on.

Chapter 7 shows how I dealt with the problem of not having the physical device i.e thinking of the device as a hidden black box.

Chapter 8 provides the results. Due to the nature of the problem, I ended up with an empty framework device that lets the *igb* driver load, but not operate properly.

Chapter 9 discusses and concludes my journey trying to emulate this device and shares my experiences. It proposes a solution to the problems I was not able to solve.

Chapter 2

Background

This chapter introduces the reader to the concept of virtualization, PC, PCI, PCI-Express, QEMU, Intel's 82576 network device card (*e1000*) and the Intel's Gigabit Ethernet device card (*igb*).

2.1 Virtualization

The concept of virtualization is to provide a computer system that consist of a CPU, memory and the I/O peripheral devices virtually running on real hardware. The CPU expose an instruction set (ISA) for programs, while memory and peripheral devices usually expose a memory map containing registers that can be modified to do the tasks the memory and the peripheral devices were made for. All the I/O devices are connected to the CPU via *bus lines*, which is a set of electrical lines connected to the I/O-pins of the CPU. A good example of a design of a computer architecture widely used today is the *Von Neumann* architecture that follows this design principle.

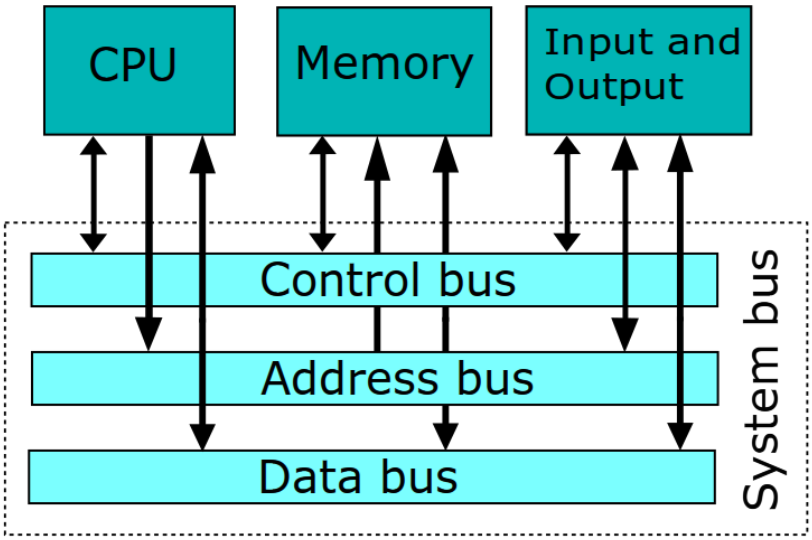

Figure 2.1: The *Von Neumann* architecture that most computer system derive its architectural design from.[33]

What virtualization does is to replicate the functions of the hardware by implementing the interfaces and the functionality made by it in software, thus makes it possible to use the functionalities of the device it is emulating without having it physically installed. By doing that, the cost of owning and maintaining a computer system is greatly reduced. Though, doing this does come with a cost. Running a program that uses a virtual device usually runs a lot slower on many computer systems than running your software on the real hardware itself.

Many cloud hosting providers do use this technology to rent out their dedicated servers to users. This let them share their physical hardware to more users securely than they could without it. *Linode* and *Amazon* are companies that provides this kind of services in the cloud which use this technology to share their computing resources in a secure way to their customers.

These virtual machines are also usually used to run existing software systems, which contains many types of applications running simultaneously on top of an operating system in a virtual machine.

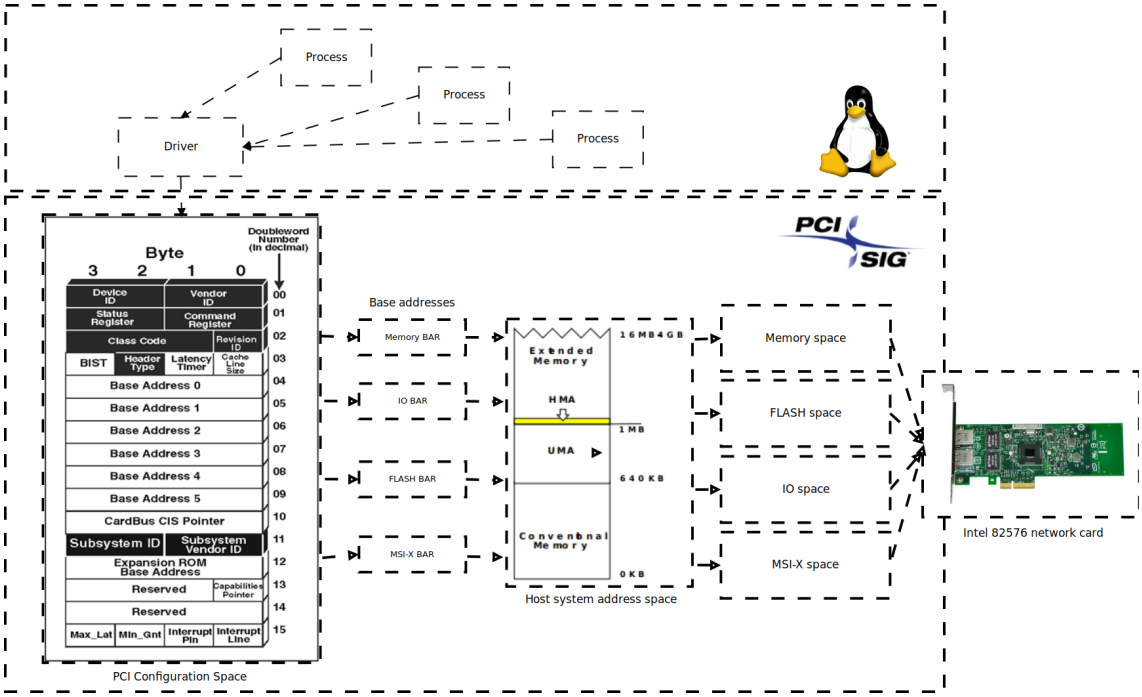

Figure 2.2: Layers of communication

The above figure shows the overall architecture of the components that needs to be emulated. It consist of the *igb* driver communicating with the *igb* device via the *PCI-Express* bus to the *igb* device. The driver runs inside the *Linux* operating system and gets to talk to the processes, while the rest shows the underlying *PCI-Express* subsystem. The block marked as *PCI-SIG* shows roughly how the driver communicates with the *igb* network device card. It shows the operating system's *igb* driver which reads and writes to the PCI-Express configuration space of the *igb* device, maps the *igb* device's register to the system's main address space using the *BAR*-fields and then modify those memory-mapped registers to control the network device.

All these subsystems run in a emulated environment made by the QEMU virtual machine application on top of a host operating system, which is in this case is Linux. The QEMU application emulates all the hardware computer subsystems mentioned above. The PC running this application needs to have certain specific CPU and RAM setup to handle these virtualization tasks, or else the entire QEMU process will run slowly or not execute at all.

2.2 QEMU

Figure 2.3: The official QEMU-logo[6]

QEMU is one of the virtualization software used in the computer industry. It can virtualize several computer machine architectures. The *Personal Computer (PC)*, also known as the *x86*, is one of them. It is also being commonly supported due to being widely used everywhere. QEMU creates the virtual computer environment for the operating system and its applications to execute inside. The hardware is emulated, meaning it is simulated in software, which enables applications and operating system to be run without the need for the real hardware to be installed into your computer. It also enables us to run multiple virtual machines, each having its own virtual environment on a single real host machine.

The operating system that runs inside a virtual machine is known as the *guest OS*. The virtual machine runs in a operating system called the *host OS*.

QEMU recompiles code while the virtual machine is running[3] from the *host machine's* instruction set to the *target machine's* instruction set. It executes as a process on the *host* operating system. The process owns its own memory space like other processes do. Each *virtual CPU* executes as a CPU thread on the host operating system. Those threads are executed asynchronously. The host's kernel scheduler decides when QEMU and its virtual CPUs are preempted, load and executed. Each device gets its own I/O thread[21] for handling the I/O-devices.

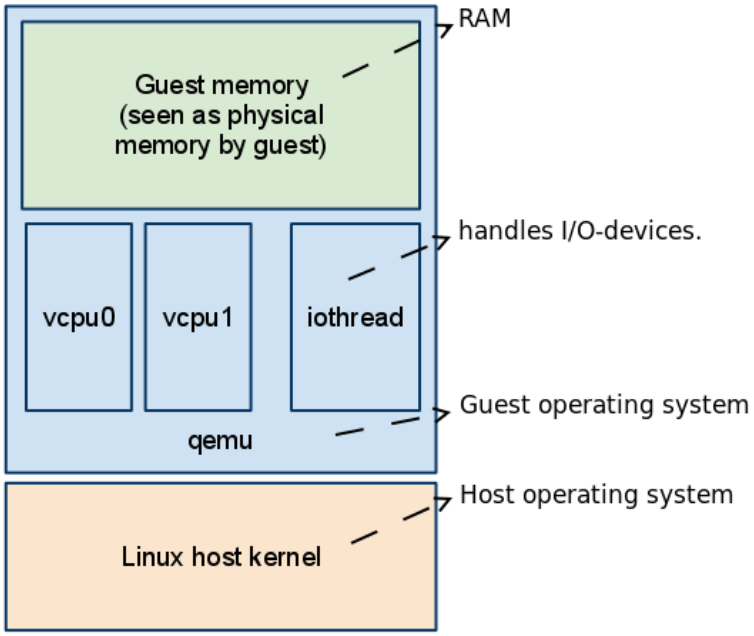

Figure 2.4: QEMU as seen from the *host* operating system.[27]

I/O-devices are implemented by "deriving" from a C structure that represents its device type. For example, QEMU defines a C structure that represent a common part of the PCI device that PCI devices inserts into their C device structure. The new PCI device then implements functions which are registered into the PCI subsystem of QEMU. The subsystem claims an address space in the host operating system, where it registers these functions to be called back, called *callbacks*. When the guest operating system communicates with a device, it reads or writes to that claimed address space, thus calling back one of the registered functions. The function then redirects the job to a virtual device registered into that space instead of executing that job on a real device.

The I/O device's address space are registered into QEMU's memory subsystem using the memory API found and documented in `qemu/memory.h`. A struct named `MemoryRegionOps` contain fields to define the properties of a memory region, e.g memory endianness, and function pointers that points to callbacks that are called when the memory regions are read and written to by a running program, usually the operating system's driver. The C header includes functions to register, initialize and destroy memory regions.

QEMU uses the GNU `autotools` to build the entire source code and documentation. The program generates a *makefile* for the system that compiles and installs the QEMU program.

Executing QEMU is currently a taxing job for a computer to do. Todays processors for the *x86* architecture provides hardware assistance to increase this task's performance. Intel call its virtualization assistance

technology for *VT-x* and AMD calls it *AMD-V*.

This assistance technology can be enabled by passing the *–enable-kvm* parameter to the QEMU executable. The passed parameter enables *KVM* support, which allows QEMU to execute code native on the CPU.

2.3 PC

Figure 2.5: A 80's IBM PC[11] and todays HP Compaq PC[10].

The *PC* is one of the computers that is emulated by QEMU. It is a personal computer designed and commercialized by the company IBM. This computer system has evolved from being a box with two diskette stations, a screen and keyboard to evolve into large towers and flat rack servers. It consist of modular computer components, a design that they hoped would attract third-party vendors to sell new exotic hardware devices for the computer system, which many did. Ironically, some of these vendors later cloned the entire system and released clones that competed against the original IBM PC. The system gradually evolved to become the industry-standard modular computer system composed of third-party components made by different vendors sharing a common interface. The name of the architecture gradually changed to become the *x86* computer architecture. The 64-bit x86 PC is the machine architecture that we will let QEMU emulate for these network devices. The *igb* device will be connected virtually to this virtual computer system.

2.4 Operating system

Inside the computer runs the *operating system (OS)*. The operating system is the program that manages the processes and resources of the computer. It separates the applications running from the devices underneath them in a computer by being the operator between them.

Linux is the operating system that will be used for this thesis due to its open source nature. I do assume it is widely used in servers, which is the type of environment the *igb* device is designed for.

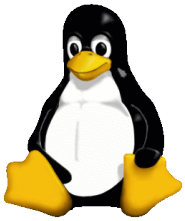

Figure 2.6: It's Tux!

Linux itself is the *kernel* of an operating system. It is the program that maintains multiple programs executing concurrently each sharing the computer system's resources. An application executing in the operating system communicates with the devices by calling functions to control them provided by the drivers. An example of such driver is the Linux's network device driver that provides `ioctl()`-calls for sending and receiving packets over the network. The driver implements those functions by further talking with a hardware network device usually by reading and writing to the device's memory address space.

In Linux, applications gets to talk to other computers by calling the functions in the *BSD socket* API. This API provides structures and functions for applications to send and receive data using either the *TCP* or the *UDP* protocol. The socket API's functions packs the data into Linux's *skb* data structure, seen below, that they pass on to the *network device driver*.

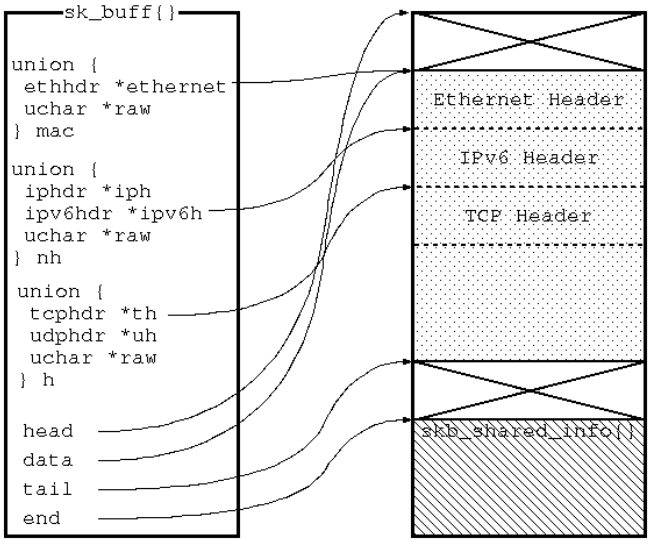

Figure 2.7: SKB data structure.[18]

In my development environment, the operating system executes in a virtual environment made by *QEMU*. The driver inside it communicates with the emulated I/O-devices provided by QEMU, which is implemented in software.

The network cards emulated provides a *PCI configuration space*, where there are *Base Address Registers* that maps different *address spaces* to the host system address space to provide its functions.

The drivers of an operating system communicates with the devices by reading and writing to two different address spaces: *The system address space* for *user*- and *system*-software, and the privileged *I/O address space* made to communicate to the connected I/O peripheral devices.

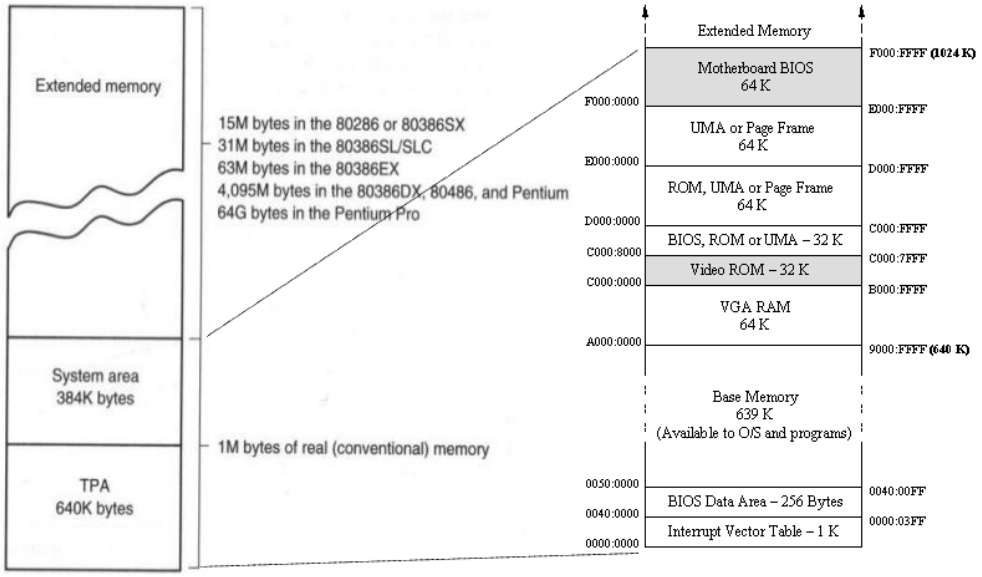

Figure 2.8: Conventional memory.[19][7]

The *system address space* is divided into several other address spaces depending on how the PC was built. When the PC was first released, this address space was divided into a *conventional memory space* for storing data, a *video memory* for displaying data on to the screen and a *read-only memory* for storing the BIOS program. This address space was later extended by IBM and its clones adding new features to the system. Today PC's deploy a myriad of different designs of it.

The *I/O address space* is a separate 16-bit address space that peripheral devices can be installed into, and read and written to using the x86's *in* and *out* assembler instructions. These instructions are usually defined as functions named `inb()` and `outb()` in C libraries. They are also privileged, which means the instructions must be executed while the CPU is set to run in *kernel mode*[32]. They are usually used by the drivers in an operating system.

The I/O peripheral devices connects to the CPU via the bus. The PC had many different kinds of buses during its evolution. The first one was known as the *ISA* bus, which was gradually changed to the *PCI* bus, and then became todays *PCI-Express* bus following accordingly to the computer industry's needs.

The Linux driver for the *igb* device is derived from the Linux driver for the *e1000* device. The engineers decided when implementing it, they copied all of the old code from the older *e1000* device driver and modified it to make the driver for the *igb*, instead of sharing common code between these drivers. Many incompatible changes were made, and both drivers diverged from each other. Both drivers implement Linux's network device API, sometimes called the *New API*, to present itself as a network device to the applications.

The *igb* driver begins by calling `igb_module_init()` which registers the driver into the PCI subsystem of Linux. Then `igb_probe()` is called to probe, initialize and setup the environment for the *igb* device. This function sets it up by settings values in its internal data structures and copying some "ops"-function pointers that it shares with the *e1000* driver. These function pointers points to functions that operate on the internal hardware circuitry on the device. It then checks if the EEPROM and MAC address is valid, and then continues on with its execution. If the checks fails, the initialization-function returns an *error code* that represents a failure to do its job. Then the driver reads the entire EEPROM, calculates its checksum and compares it with the checksum stored in the `CHECKSUM`-register. The resulting sum should be equal to `0xbaba` to pass the checksum test.

Rest of the driver environment consist of packet queues and its interrupt callbacks that are called each time the device transmits and receives a packet from the *igb* device.

A script called *kmake* is executed to recompile the changes to the *igb* driver during development.

```
#!/bin/sh
export KVER=`uname -r`
export KDIR=/lib/modules/$KVER/build

pwd=`pwd`
V=""

if [x$1 = xV=1]; then
    V=$1
    shift
fi
set -x
make -C $KDIR M=$pwd $V $*

exit $?
```

Listing 2.1: A script named *kmake* that compiles the *igb* driver.

Compiling and inserting the driver module is done by invoking `kmake; rmmod igb; insmod igb.ko` in the command shell.

2.5 PCI

Figure 2.9: The PCI-SIG logo.[26]

One of the I/O-buses that a PC can have on its motherboard is the *Peripheral Component Interconnect (PCI)* I/O bus. This is an I/O-bus standardized by a committee of representatives from different companies in the computing industry. This group is called the *PCI-SIG* group (PCI Special Interest Group). This I/O-bus standard is used by a wide variety of computer devices that requires to be connected to a fast low-latency bus to the CPU[26]. It was designed to replace the older ISA bus used by the PC.

Each PCI device in the bus system is represented as a *physical function (PF)*. Each physical function has at least 256 byte mandatory address space allocated for the PCI *configuration space*. It is the device's central address space containing fields used by the host system to discover the device, configure it and control it.

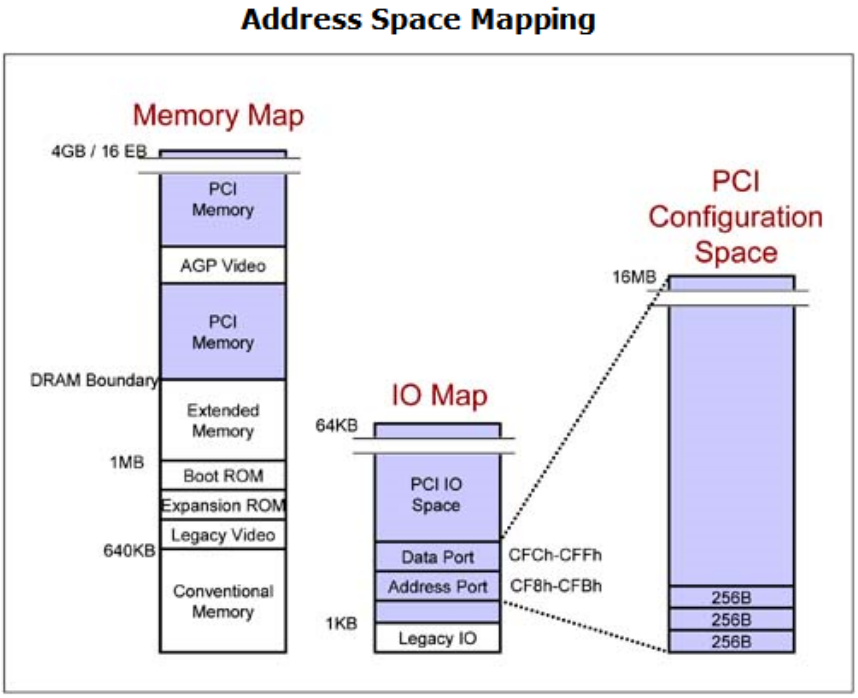

Figure 2.10: The PCI Express's address space mapping[5, page 22]

The first 64 bytes of the address space consist of a standardized header

filled by the vendor, while the remaining 192 bytes are device dependent.

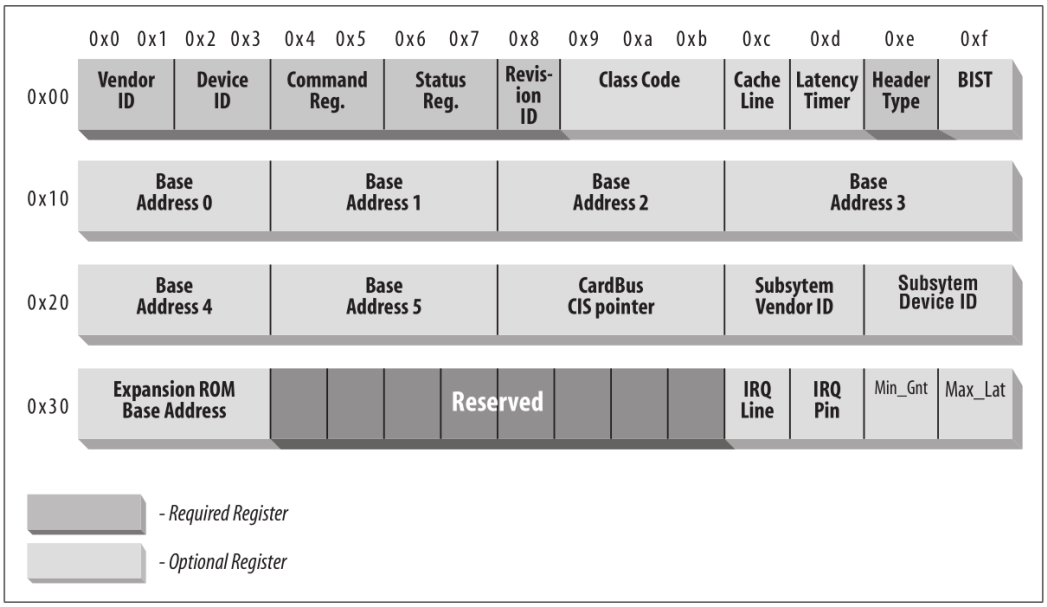

Figure 2.11: The first 64 bytes of the standardized PCI header in the configuration space[8, page 308]

The *dark grey fields* are the required fields that needs to be present, while the *light grey fields* are optional fields. The values in the fields are always *little-endian*[8, page 308] due to its first use on the PC, which read and wrote individual values to memory in *little-endian* order (and still is). The mandatory fields describes the PCI device and configures its behaviour. All identifiers are assigned uniquely by the *PCI-SIG* committee.

This address space also contain the *Base Address Registers*. These fields are used by a program to map the device's I/O-registers into the computer's main system address space. This is done by writing the host system's memory address into the base address register.

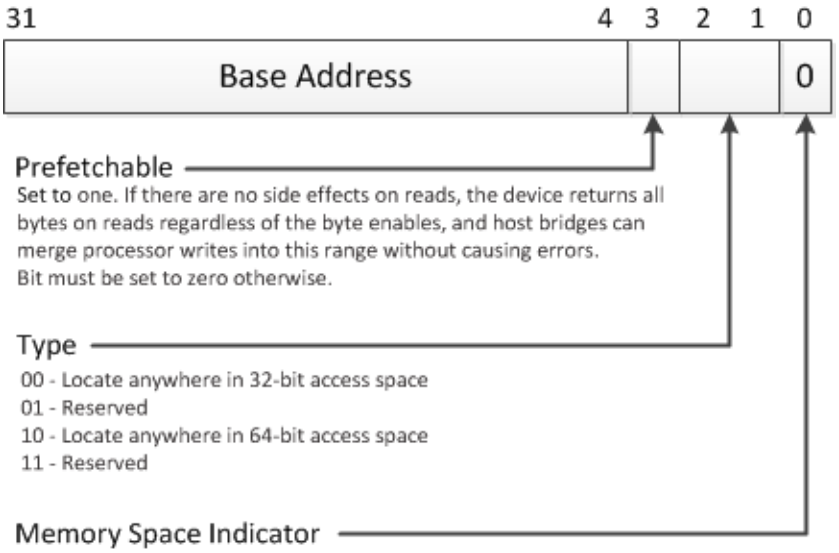

Figure 2.12: The *Base Address Register* format.[20]

The device will then claim that address space in the system's address space and place its registers there. These device registers can then be used by the program to control the device to do its tasks. Each PCI device defines itself what kind of memory space these *BARs* map to.

The PCI devices are accessed by the CPU through the I/O-bus using the I/O-port instructions. By using them, the *PCI's configuration space* is accessed, read and written to by programs. The *configuration space* is located at the I/O-port address ranges `[0xCF8h-0xCFBh]` and `[0xCFCh-0xCFFh]`[29].

Each PCI device is identified by a *bus* number, a *device* number, and a *function* number[8, page 305-306].

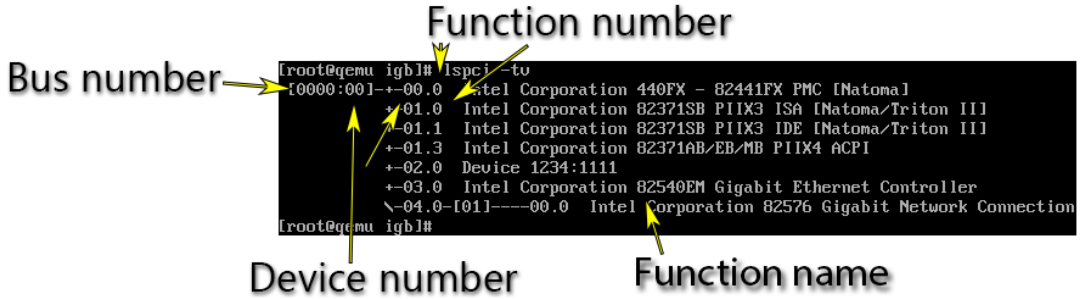

Figure 2.13: The PCI bus with devices connected to it discovered by the *lspci* tool in Linux.

The PCI bus later evolved to become *PCI-X* bus and then todays *PCI-Express* bus due to changing requirements. Both *PCI* and *PCI-X* are now deprecated buses that are not in use anymore by any vendor members of the PCI group. They have both been replaced by the *PCI-Express* bus.

2.6 PCI-Express

The *PCI-Express* bus is a new redesigned I/O-bus for I/O-devices based upon the old PCI bus made by the *PCI group*. It extends the previous buses due to changing requirements in the industry.

From a software perspective, the new PCI-Express bus extends the PCI header from 256 bytes to 4096 bytes for each *PCI function*[5, page 576] in the PCI configuration space. From a hardware perspective on a PC x86 motherboard, the ports to connect the PCI-Express devices has been redesigned. The new PCI-Express port is incompatible with older PCI and PCI-X cards.

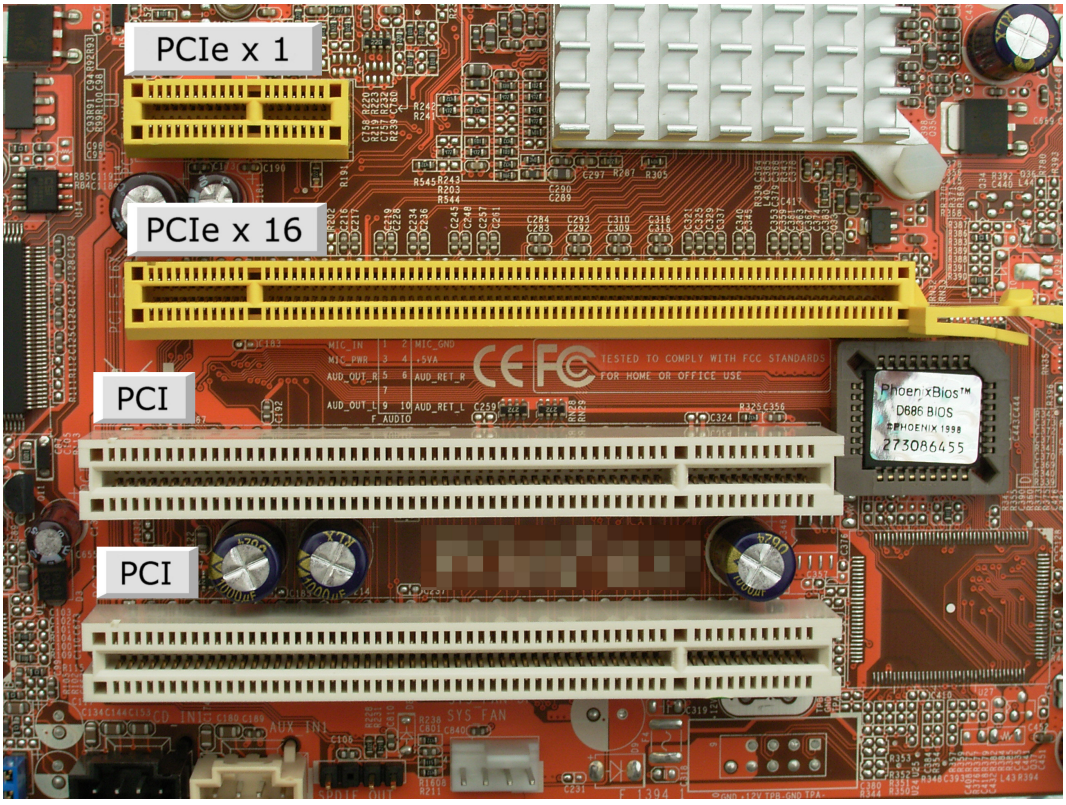

Figure 2.14: Old PCI and new PCI-Express slots.[25]

The PCI-Express slots comes in *x1*, *x8* and *x16* type of ports, where the *x* represents the number of lanes. Each lane consist of *transmit* and *receive* wires that can send and receive a *single* bit per clock cycle[23].

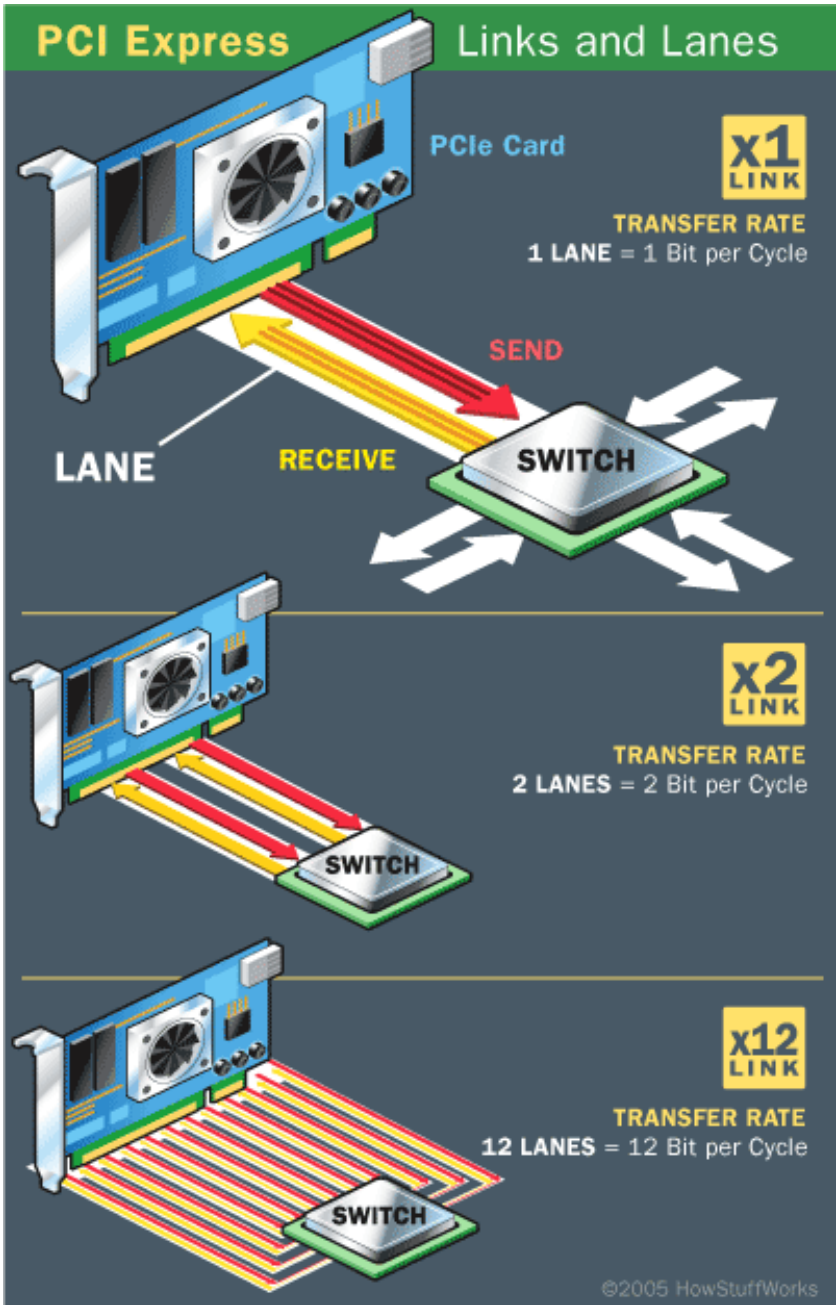

Figure 2.15: PCI-Express lanes[24]

All of these connectors can be connected to all types of PCI-Express slots. That means that PCI-Express cards with a *x1* connector can be connected to a *x16* slot and vice versa. By connecting a *x16* card to an *x1*-port makes the card operate at 1/16th lower off its possible bandwidth. Doing the opposite would not increase its bandwidth.

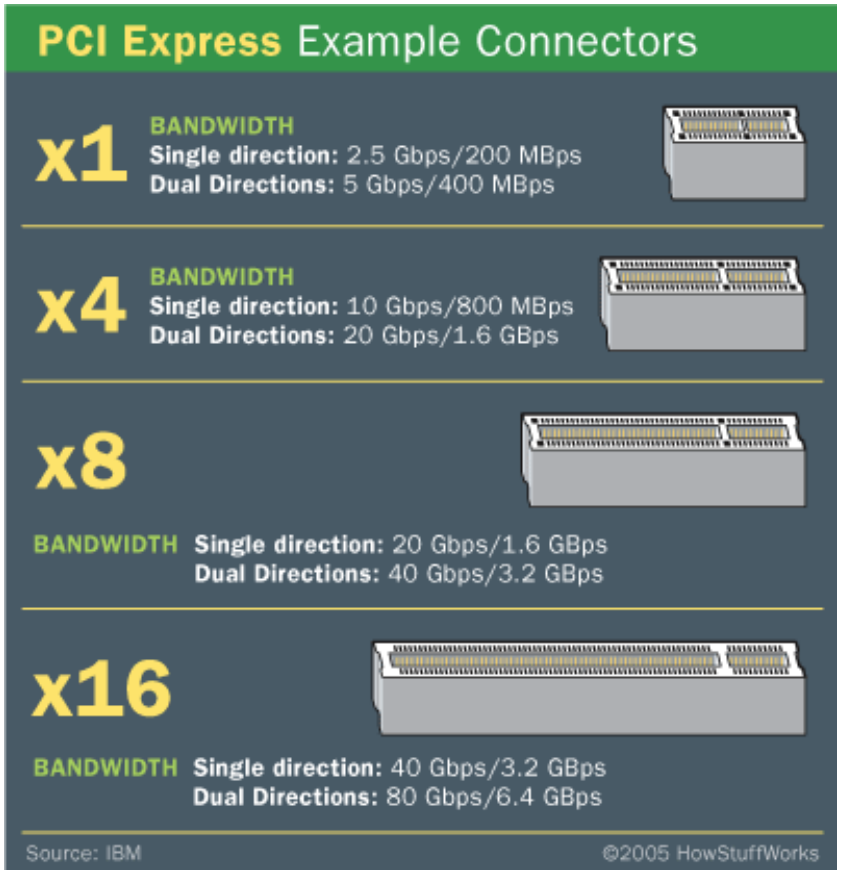

Figure 2.16: Different PCI-Express slots with each own bandwidth.[23]

The PCI-Express configuration space are of 4 kilobytes in size. The first 64 bytes follow the same header conventions as the previous PCI *configuration space* header and retains the same usage. The rest of the header consist of the new extended capabilities the PCI group added for the *PCI-Express* bus.

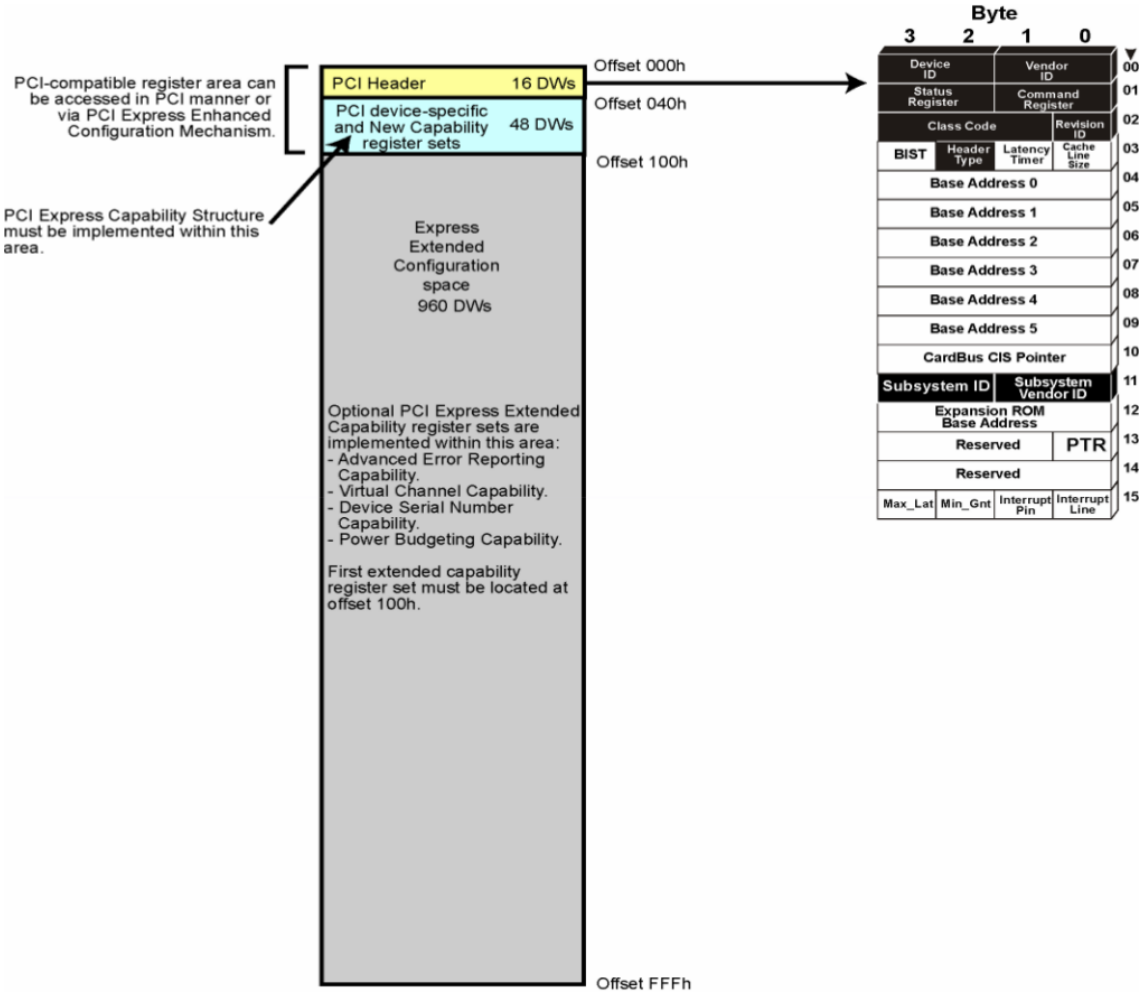

Figure 2.17: PCI-Express configuration space.[4]

2.7 SR/IOV

One of PCI-Express bus extended capabilities that is used by the *igb* device is known as *Single-Root I/O Virtualization (SR-IOV)*. *SR-IOV* gives a single PCI-Express device the ability to create multiple virtual devices seen as *virtual functions (VFs)* on the PCI-Express bus.

The *virtual functions* gets their own configuration space that is derived from the *physical function*'s configuration space. The virtual functions are shown to the host computer system and acts as their own independent PCI-Express device. The allocation of virtual functions is done by modifying the *VF*-registers assigned for controlling it placed in the MMIO address space of the device.[34]

Using this feature, a virtual machine can communicate to its own dedicated virtual device, thus skipping a layer of communication and gain performance. In this case, a Linux driver named *igbvf* is loaded by the

virtual machine to send and receive packets from and to a virtual device made by the *igb* driver. Processing network data this way is more efficient than other alternatives that currently exists.

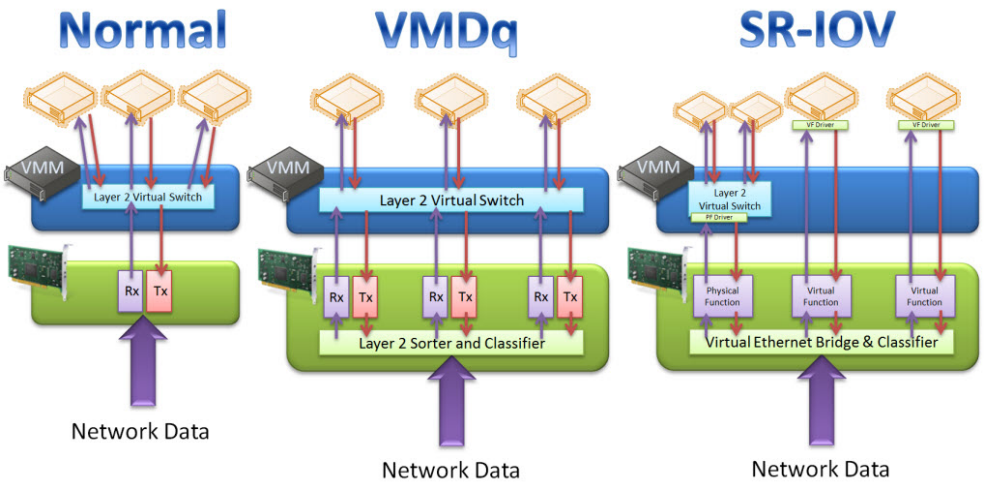

Figure 2.18: A comparison between different solutions of passing data between a driver and a network device.[2]

The above figure shows the VF driver skipping working through the *middle layer*, known as the virtual machine manager (VMM), thus increasing its work efficiency.

2.8 E1000

The Intel's 82540-series network device cards, also known as the *e1000*, are a family of network cards used by desktops and servers. They consist of descriptor queues and interrupt timers. The descriptor queues stores packets in buffers received from the computer network, which is then later copied to the host computer's memory. The interrupt handlers are called for each packet the network card transmits and receives to notify the driver that the job requested is done.

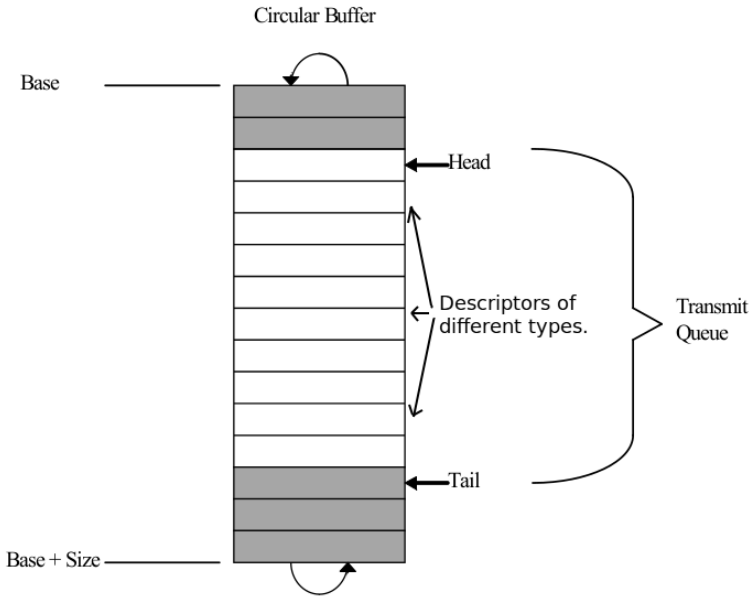

Figure 2.19: Descriptor transmit queue[14]

Sending and receiving packets consist of passing the ownership of packet buffers between the driver and the network device. The data in those buffers are packed into a *descriptor* and inserted into one of the network device's *descriptor queue* with the help of the device's built-in DMA engine that transfers the buffer between the host driver and the device asynchronously.

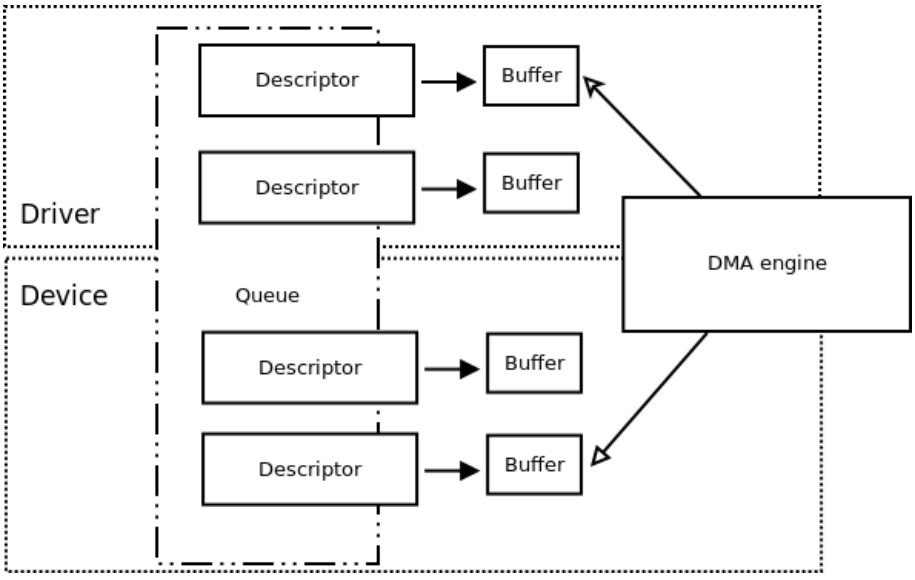

Figure 2.20: DMA engine copying buffers.

The 82540 supports two types of descriptors that are called *legacy descriptor* and *extended descriptor*. Which one is used is defined by

setting the *DEXT*-bit. If it set to 1, the descriptor is *extended*. If it set to 0, the descriptor is *legacy*. All fields seems to be stored in *little-endian* format, which means when reading and writing a field from and to the register and memory, the least-significant byte are placed at the lowest memory address. The rest of the bytes are read and written towards higher memory addresses.

Table 3-8. Transmit Descriptor (TDESC) Layout – Legacy Mode

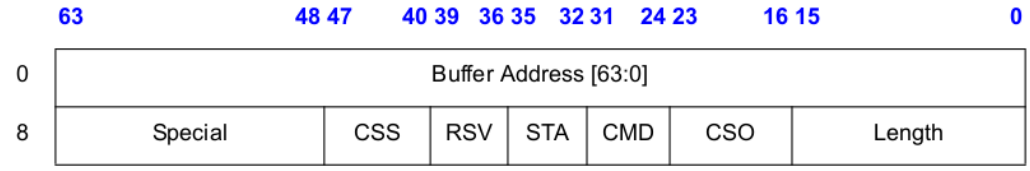

Figure 2.21: Legacy descriptor[31, page 50]

The *legacy descriptor* is the same descriptor that is used by the *e1000* device. The *extended descriptor* is an entirely new descriptor format made to make processing descriptors from the descriptor queues more efficient.

When transmitting data using the extended descriptors, the device formats the descriptors following the *context-*, *data-* and *write-back-* descriptor formatting scheme[14, page 318].

Table 3-7. Transmit Descriptor (TDESC) Layout

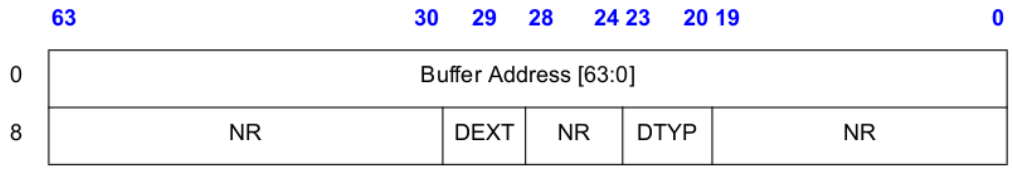

Figure 2.22: Transmit descriptor[31, page 50]

When receiving data, it uses the *read-* and *write-back* format[14, page 300].

The network device provides three types of interrupt mechanisms: *Legacy Interrupts*, *Message-Signaled Interrupts (MSI)* and *MSI-X*. Each packet queue owns a single interrupt handler used to indicate to the driver that something has happened in a descriptor queue. Older devices that only supports *Legacy Interrupts* can only create and handle *a single* queue. Newer devices supports *MSI-X* interrupts, which is used to create and handle *more than one* queue.

The host computer system and the network device transmits and receives packets by sharing a descriptor queue together. The host driver sets up a ring queue where the driver are responsible for handling a part of descriptors in the queue, while the network device are responsible of handling another part of descriptors in the queue.

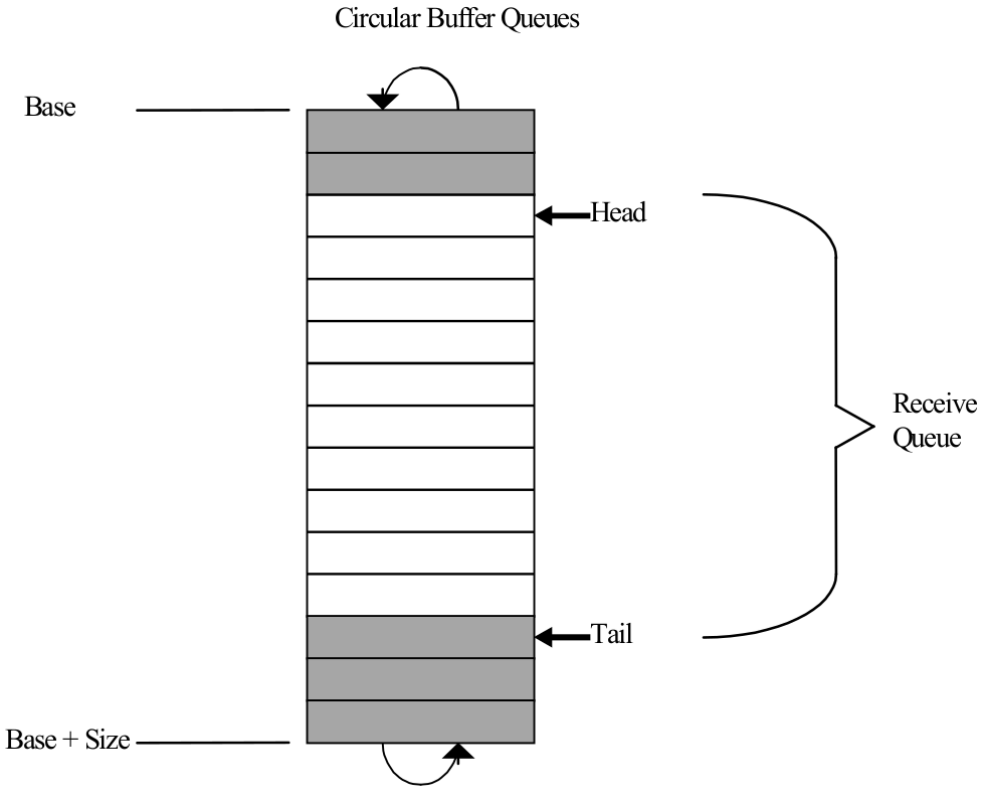

Figure 2.23: Receive queue[14, page 19]

The ring queue contains *descriptors*, shown as a *white rectangle*. The descriptors contains a physical memory address that points to a data buffer containing the packet. The *shaded* areas are owned by the driver, while the *non-shaded* white areas are owned by the hardware device. The driver *appends* and *prepends* descriptors from the ring queue by modifying the *head*- and *tail*-pointers. The device do the same.

The contents of the buffer pointed by a descriptor is copied between the host's system memory and the device's system memory using the device's built-in DMA engine without the need to use the CPU for that task. The driver initiate this copy operation using the DMA engine. The DMA engine can be run asynchronously without the CPU, thus freeing the CPU to do other tasks in the mean time.

A device consist of one or more descriptor queues. Each descriptor queue holds an interrupt handler that interrupts the CPU each time the device is finished working on a descriptor in the queue.

The network devices has an address space called the *Memory-Mapped I/O* which contains several registers to control the device's functions. Some of them control the handling of the descriptor queues. These are accessed by writing a *host memory address* to the *BAR* responsible for these functions. The device will then map the registers to the host's system address space pointed to by the *BARs*. Most hardware functions for this device is accessed through this address space.

The BAR regions consist of four regions:

BAR	Address	Item
0	10h	MMIO
1	14h	Flash
2	18h	I/O

Table 2.1: *e1000*'s Base Address Registers (BARs).[15, page 87]

The main control registers for the device are the `CTRL` and `CTRL_EXT`. The current status of the device is shown in the `STATUS`-register.

2.9 Igb

The Intel 82576, referred to as the *igb*, is a gigabit network device card made for multi-core and virtualized computer systems, usually seen in server environments. The device card supports SR/IOV[14], which allows the device to create multiple unique virtual network devices, seen as *virtual functions*, to be used by each virtual machine independently. The card derives its design from its predecessor Intel *82540* network device card series. The Linux driver for Intel's 82576-series is named *igb*.

The card connects to a *PCI-Express* port. It consist of physical Ethernet or fiber ports to the outside computer network[14]. It increases the number of transmit and receive descriptor queues the previous devices has to 16 descriptor queues[14, page 57].

This device is quite similar to the *e1000* device in its design. Most of its interfaces and registers are quite similar to the *igb* device, but still there are many differences to cover that differentiates this device from the *e1000* device.

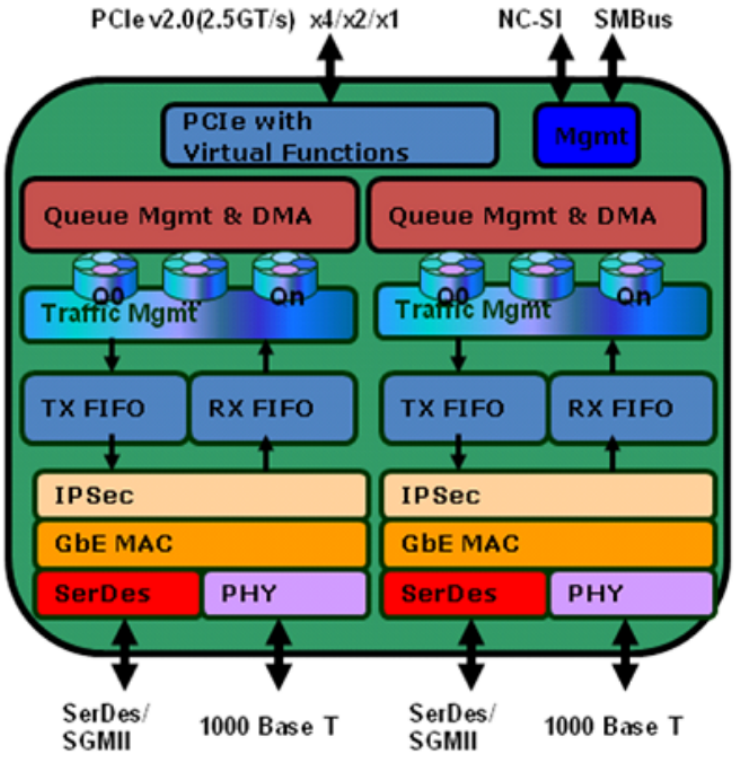

Figure 2.24: An overview of components inside the *igb*.

The 82576 network device (*igb*) supports the old *legacy* descriptors and new extended *advanced* descriptors that is contained in the descriptor queues. The descriptors usually contain information on how to process a packet. The *legacy* descriptor is defined as a *single* descriptor, while the *advanced* descriptor consist of a *context*- and *data*-descriptor to remove the need to duplicate the *same* descriptor for the same type of packets that is sent and received by the *igb* device. When processing these descriptors, some of them is overwritten with a *write-back* descriptor after being processed to indicate to the driver that the descriptor overwritten has been processed by the device.

Table 7-23. Transmit Descriptor (TDESC) Fetch Layout — Legacy Mode

	63 48	47 40	39 36	35 32	31 24	23 16	15 0
0	Buffer Address [63:0]						
8	VLAN	CSS	ExtCMD	STA	CMD	CSO	Length

Table 7-24. Transmit Descriptor (TDESC) Write-Back Layout — Legacy Mode

	63 48	47 40	39 36	35 32	31 24	23 16	15 0
0	Reserved				Reserved		
8	VLAN	CSS	Reserved	STA	CMD	CSO	Length

Figure 2.25: Legacy descriptor[14, page 319]

Table 7-29. Transmit Context Descriptor (TDESC) Layout — (Type = 0010b)

	63 40	39 32	31 16	15 9	8 0
0	Reserved	IPsec SA Index	VLAN	MACLEN	IPLEN

	63 48	47 40	39	38 36	35 30	29	28 24	23 20	19 9	8 0
8	MSS	L4LEN	RSV	IDX	Reserved	DEXT	RSV	DTYP	TUCMD	IPSec ESP_LEN

Figure 2.26: Advanced transmit context descriptor[14, page 322]

7.2.2.3 Advanced Transmit Data Descriptor

Table 7-32. Advanced Tx Descriptor Read Format

0	Address[63:0]								
8	PAYLEN	POPTS	CC	IDX	STA	DCMD	DTYP	MAC	RSV	DTALEN
	63 46	45 40	39	38 36	35 32	31 24	23 20	19 18	17 16	15 0

Table 7-33. Advanced Tx descriptor write-back format

0	RSV		
8	RSV	STA	RSV
	63 36	35 32	31 0

Note: For frames that spans multiple descriptors, all fields **apart** from DCMD.EOP, DCMD.RS, DCMD.DEXT, DTALEN, Address and DTYP are valid only in the first descriptors and are ignored in the subsequent ones.

Figure 2.27: Advanced data descriptor[14, page 325]

The Linux driver sends and receives packets almost the same way as the *e1000* driver. One big difference is that the *igb* driver extends the types of descriptors it supports with a new set of ones called the *"advanced"* descriptors.

To implement an emulated *igb* device, its *interface* to its in-built hardware functions needs to be implemented virtually by QEMU. This interface consist of addressable memory regions consisting of *registers*[14, page 443] mapped by the *Base Address* fields in the *configuration space*. The address space regions for this device is mapped to internal components inside the adapter according to this table[14, page 690]:

BAR	Address	Name
0	0x10	Memory-mapped I/O BAR
1	0x14	Flash BAR
2	0x18	IO BAR
3	0x1C	MSI-X BAR

Table 2.2: *igb*'s Base Address Registers (BARs).[14]

The difference in the `BAR` space compared to the *e1000* device, is that MSI-X interrupts has been added to the BAR space.

This network device provides 16 `RD*` and `TD*` registers in its MMIO address space that are backward compatible with the *e1000* device. They are used to control the 16 descriptor queues provided by the *igb* device.

The PCI configuration space do also contain the SR/IOV-capability to create virtual devices.

One of the BARs maps the *Memory-Mapped I/O* to the host memory. They give the host software access to the physical registers of the device. Most of the registers and its placement of offsets are the same as on the *e1000* device. Some of them have been moved to new offsets, but Intel has taken great care to make *aliases* that remaps these registers to the same offsets as on the *e1000* device. That way, old drivers written for the *e1000* device can be somewhat reused.

The device contains an EEPROM that is *read-only*. The EEPROM stores *product configuration* information[14, page 112]. To read and write from and to it, the driver reads and writes to a single register made for this task. The offset to that register is set at `14h`, which is also at the same offset as on the *e1000* device.

The main control registers for the device is the same as on the *e1000* device called `CTRL` and `CTRL_EXT`. The latter is written to by the driver.

Chapter 3

Framework

The state of the framework given emulates the subsystem described above with the network device just emulated partially. It provides the skeleton code to emulate the *igb* device. It consist of incomplete data structures representing the internals of the *igb* device and incomplete functions that should emulate its behaviour, but currently not all of them does it.

The framework works by providing objects that represents the *igb* device's internal components, which registers callback functions mapped to an address space of the device. The callbacks are called each time the operating system's driver modify the register in that address space. Two functions are defined named `realize` and `destroy` which are called by the PCI subsystem of QEMU to *create* and *destroy* the device when the emulated computer system is turned on and off by the user of it.

The framework gives us the ability to load and almost initialize Linux's *igb* driver.

```
typedef struct IgbState {
    /*< private >*/
    E1000State parent_obj; /* The derived e1000 state. */
    /*< public >*/
    MemoryRegion flash; /* New IGB regions. */
    MemoryRegion msix;
} IgbState;
```

This `struct` represents the state of the *igb* device. It contains a pointer named `parent_obj` that points to the *e1000* device state that the code is derived from. It defines memory regions mapped to callbacks for the `FLASH` and `MSIX` address space.

The functions shown below are the callbacks for the memory regions.

```
static void pci_igb_realize(PCIDevice *d, Error **errp)
```

creates an instance of the *igb* device on the host system and sets up the *igb* device's memory regions for the callbacks. The parameter `PCIDevice *d` points to the object representing the *igb* device. The parameter `Error`

`**errp` can be used to report any error that has occurred by storing error information in it.

```
static void pci_igb_uninit(PCIDevice *d)
```

unregisters the memory regions and removes the device from the host system. The parameter `PCIDevice *d` passed points to the created *igb* PCI-Express device.

```
static uint64_t igb_flash_read(void *opaque, hwaddr addr,
   unsigned size)
```

currently prints out the memory address and how many bytes the driver wants to read, given as parameters when the driver reads the `FLASH` memory on the device, to the standard output. This is to print out the address and size read each time the device's operating system driver reads the device's `FLASH`. This function can be used to discover how the driver interacts with the device's `FLASH` memory. The parameter

```
void *opaque
```

is assumed to be a memory address that points to a *PCI device* `struct` representing the PCI device of the *igb*. The address needs to be casted to a PCI device struct to become accessible by the device emulation code.

```
hwaddr addr
```

is a memory address that points to the first byte the driver wants to read.

```
unsigned size
```

is the number of bytes to read from the memory space pointed by the memory address `addr`. The function returns an `uint64_t`-type value, which is a value read from the `FLASH` as a 64-bit little-endian bytes.

```
static void igb_flash_write(void *opaque, hwaddr addr,
   uint64_t val, unsigned size)
```

currently prints out the memory address and the value the driver wants to write to the `FLASH` memory on the device.

```
void *opaque
```

is an address that points to a PCI device object containing the variables representing the PCI device.

```
hwaddr addr
```

is the memory address passed that points to the first byte in the memory space the driver wants to write to.

```
unsigned size
```

is the number of bytes to read from the memory space pointed by the memory address `addr`.

These functions are callbacks that are called each time the device's operating system driver reads and writes to the related memory region. The `MemoryRegionOps`-object is used to register the *read* and *write* callbacks. The callbacks are assigned into this object:

```
static const MemoryRegionOps igb_flash_ops = {
    .read = igb_flash_read,
    .write = igb_flash_write,
    .endianness = DEVICE_LITTLE_ENDIAN,
    .impl = {
        .min_access_size = 1,
        .max_access_size = 8,
    },
};
```

which is later registered into the host system by the function beneath.

An *igbvf*-device is also made to demonstrate SR/IOV. The *igbvf* creates a PCI virtual function, which derives its PCI configuration space from the *igb*'s physical function space.

```
static const E1000Info igbvf_device = {
    .name      = TYPE_IGBVF,
    .device_id = IGB_82576_VF_DEV_ID,
    .revision  = 0x01,
    .io_bar    = (uint8_t)-1,
    .phy_id2   = I210_I_PHY_ID2,
};

static const TypeInfo igbvf_info = {
    .name          = "igbvf",
    .parent        = TYPE_E1000_BASE,
    .class_data    = (void *) &igbvf_device,
    .class_init    = igbvf_class_init,
    .instance_size = sizeof(IgbState),
};
```

Chapter 4

Goals

The end goal of this project is to implement a network device card that supports the SR/IOV-capability and demonstrate it. This is achieved by filling in the missing implementation code of the functions given and adding new ones to the set of functions. The function should behave as what we think the driver wants that is good enough for the driver to execute properly, even if it is not correct.

Before doing it, an environment for the project needs to be setup, and the code given needs to be compiled and executed. The environment should consist of a PC with Linux as its operating system installed, basic development tools (e.g editor, C compiler and such), QEMU with an image containing Linux installed and the given source code itself. QEMU is compiled and run on a x86 PC running Linux as its operating system.

An implementation of the *igb* device requires at first a initialization function that setups the data structures for representing the device. It should implement the initialization routine that setups the device's `PCI configuration space`, `FLASH` memory, `MMIO` address space, `MSI-X` vectors and the *virtual functions*, which derives its *configuration space* from the *physical function*, for the *igbvf* -driver.

To make Linux boot up with the *igb* driver using the given framework, the callback for the *EERD*-register, used to read the EEPROM, needs to be implemented to make the Linux driver for the *igb* device not crash or halt at init. This is done by implementing the callback function for it that returns the values of the *igb*'s EEPROM to pass the checksum check.

Field	Bit(s)	Initial Value	Description
START	0	0b	Start Read. Writing a 1b to this bit causes the EEPROM to read a (16-bit) word at the address stored in the EE_ADDR field and then storing the result in the EE_DATA field. This bit is self-clearing.
DONE (RO)	1	0b	Read Done. Set to 1b when the EEPROM read completes. Set to 0b when the EEPROM read is not completed. Writes by software are ignored. Reset by setting the START bit.
ADDR	15:2	0x0	Read Address. This field is written by software along with *Start Read* to indicate the word to read.
DATA (RO)	31:16	X	Read Data. Data returned from the EEPROM read.

Figure 4.1: *igb* `EERD` register format.[14]

The above figure shows the format of the `EERD`-register and how the device reads and writes to it. The crucial difference between this interface from the *igb* device and the *e1000* device is that the reserved bits are removed.

Other functions that are needed are fixing the legacy interrupts to ensure that the driver does not restart itself while trying to "up" it to drive a single queue.

Other goals could be to implement the *transmit*- and *receive*-function that is needed to give applications the ability to send and receive packets from the network, `MSI-X` interrupts to additionally add the capability to handle more than one queue and an *igbvf*-device for the *igbvf*-driver that creates a *virtual function* to demonstrate SR/IOV-capability.

The *igb* driver never reads or writes to the FLASH, which means that dummy functions that prints a debug message and returns a dummy value can be implemented to emulate it. The given framework has already done it.

Chapter 5

Methodology and tools

To reach the goal, a way to discover how the device works is necessary. I tried to use the debugging tools called *GDB and DDD* and the IDE called *CLion* to do this, but they all failed to load and parse the code.

I discovered one successful way to find out how the driver communicates with the device, based upon old code from Qumranet[9] and ideas from my supervisor[12], which is to print out the parameters that is passed to the callback functions to the standard output, in my case the terminal emulator on the desktop. I did that by placing calls to the function `printk` in the driver and `printf` in the *igb* device code in QEMU. Doing this made it possible for me to inspect its code path thoroughly, but the downside is having to go through a lengthy process of restarting QEMU each time you make a change to the code. I did get a full function trace of the driver executing on the real device from my supervisor, which could be of use somewhere.

The rest of the job is to figure out a way to implement the correct behaviour of the device using the knowledge gathered from using this method. It proved to be a successful approach to discover how the inner details of the driver works, and thus figure out a way to implement the right behaviour it expects.

In the end, the development process consists of a QEMU virtual machine image with the source code of the *igb* driver in its own window and the source code of the *igb* device emulator itself in another one. Both code were modified, compiled and tested to discover how the device works and implement the necessary changes to the device code to make the driver run properly in the operating system.

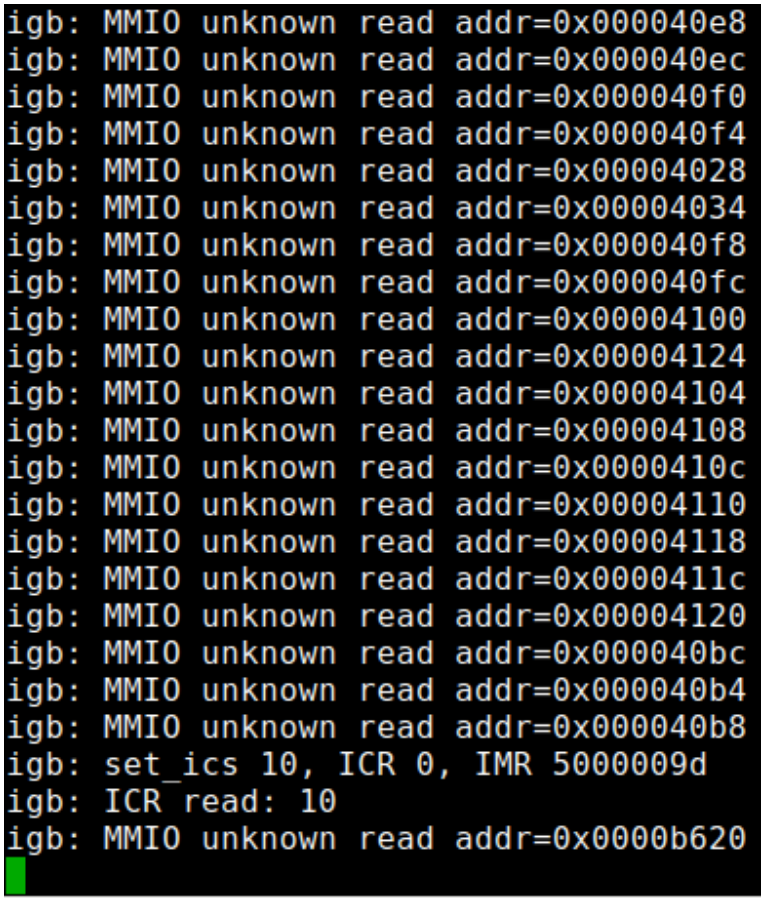

Figure 5.1: Printing out the offsets to the registers that the *igb* driver reads.

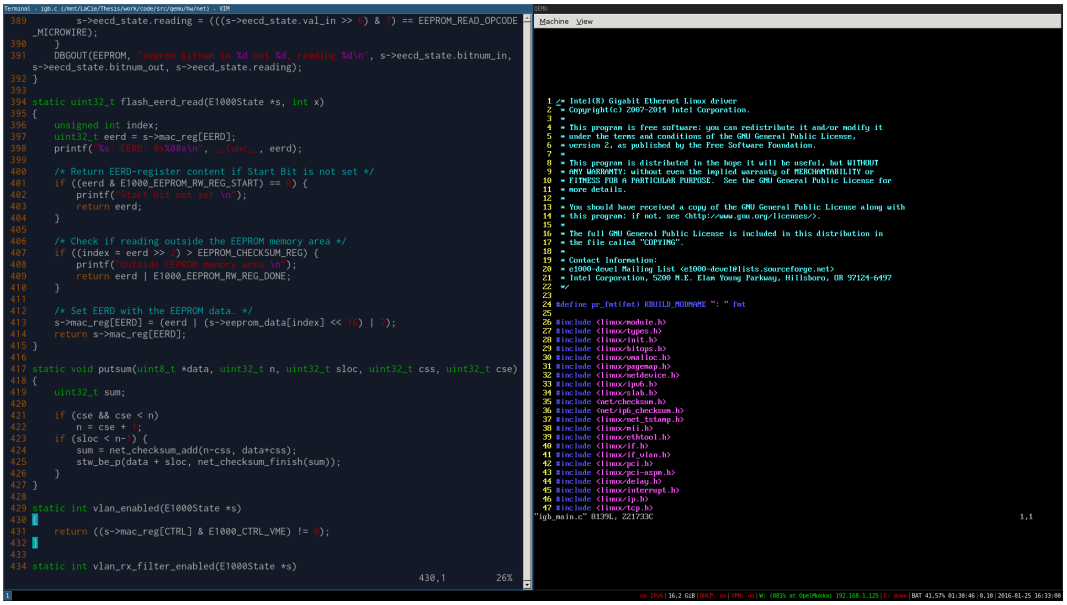

Figure 5.2: Development environment. *igb* device code to the left. *igb* driver code to the right.

Chapter 6

Environment

My development environment consist of KVM-capable PCs running a distribution of Linux that has the framework code installed.

The computer's system specification are

Sony Vaio Pro 13[30]	
Processor:	Intel i7 Haswell laptop core.[16]
Memory:	4 gigabytes of RAM
Operating system:	Arch Linux[1]

A custom-built desktop PC	
Processor:	Intel i5 4670K Haswell[13]
Memory:	16 gigabytes of RAM
Operating system:	Arch Linux[1]

The desktop computer can execute the QEMU properly, while the laptop computer can almost do it due to not having the same CPU performance as the desktop.

Installing QEMU with the SR/IOV-patch onto these computers requires us to step through an installation process that consist of these steps:

1. Configure the build by executing `./configure` with the build-parameters we want.

2. Build the source code by executing `make` or alternatively `make -j4` to compile the source code using 4 CPU threads.

3. Executing `make install` to install the built binary executables and libraries into your system.

4. Install a Linux-distribution with a kernel that contains the *igb* driver, in my case *Arch Linux*, onto a QEMU disk-image.

5. Execute `.\qemu-system-x86_64` and watch it boot the virtual machine.

This build of QEMU with the patch is executed with these parameters to enable the needed features to load the *igb* driver in a performing way:

```
./qemu-system-x86\_64 \\
    -enable-kvm \\ # Enables KVM to increase QEMU's
       execution speed)
    -m 512 \\ # Allocates 512 megabytes of RAM for the
       virtual machine)
    -hda archlinux.img \\ # Loads an image with Arch Linux
       installed on it on to the virtual machine
    -device ioh3420,slot=0,id=pcie\_port.0 \\
    -device igb,mac=DE:AD:BE:EE:04:18,vlan=1,bus=pcie\_port
       .0
```

The first parameter passed `-enable-kvm` enables KVM which lets QEMU use VT-X. This CPU feature increases the execution performance of QEMU when enabled. VT-X can be enabled in the motherboard's BIOS menu before the computer loads and jumps in into the host operating system. Most motherboards have a way of entering the BIOS menu by continuously hitting a certain button on your keyboard during boot.

The second parameter `-m 512` allocates 512 megabytes of RAM for the virtual machine. The third parameter `-hda archlinux.img` tells QEMU to use the hard disk image file `archlinux.img` as the hard disk for the virtual machine. Arch Linux was installed on to a hard disk image formatted in QEMU's disk image format named *qcow*. Arch Linux is a Linux distribution which its goals is to provide a minimalistic Linux operating system environment, which I found to be easy to use to test the driver changes compared to other distributions due to only providing a fast minimal working shell. The Linux-kernel installed is of version *3.18.5-1-ARCH*. The packages installed is of the versions retrieved some fine day during January 2015.

The last two parameters adds the PCI bus and connects the emulated *igb* device into the virtual machine.

Executing this build without any changes to the given framework makes the kernel load the *igb* driver until it reaches a certain point where it verifies the nonvolatile memory (EEPROM) for its consistency, fails the check, dies, and then prints the *"The NVM Checksum Is Not Valid"*-error into the device message log (`dmesg`).

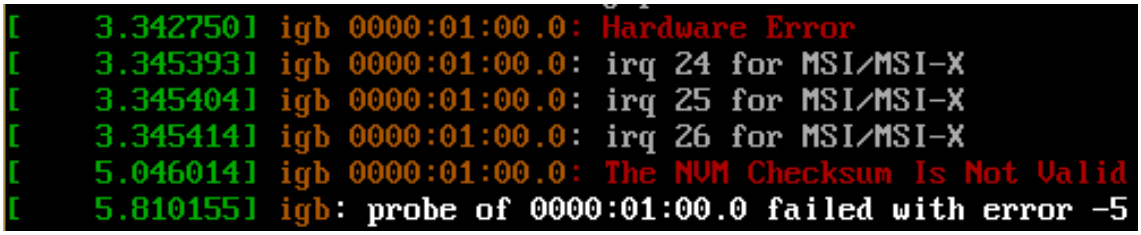

Figure 6.1: The *igb* driver loaded with errors displayed using the given framework.

When you are logged in as *root*, the network device that the *igb* driver inserts itself as can be brought up by invoking `ip link set enp1s0 up` in the command shell, but the *igb* device would not be able to load itself up and neither transmit or receive any packets from the computer network.

Chapter 7

Discovery

I was given the documentation and the driver of the *igb* device to use to discover and learn the design of the *igb* device. Unfortunately, there were certain problems trying to understand the device using this approach. The documentation was scarce, vague and had bits of errors, which made me look into the driver for more information on how the device works. The driver showed me how a program could interact with the device and what it expected the device to do, but it did not show how the internal logic of the device behaved. Overall, I had to treat the *igb* device as a hidden black box.

The documentation seems to be written with the assumption that the reader was well-informed about the *PCI* bus and Intel's network device design evolution throughout its living years. It does describe the address space provided by the device, but it describes vaguely on how it behave. Some small errors were discovered too. None of the drivers behaviour was described in detail in the manuals. The *igb* manual was written as a continuation of the older *e1000* device manual. There were things that were described in the older manuals that could not be found in the newer written manuals.

So to discover how the device behaves, I had to guess my way through it by changing the driver to behave as I assumed how the device works. By changing the driver code, placing marks in the code that warns me which direction the execution of the program is heading and implementing guesses of its behaviour in the device code, I was able to figure out how the *igb* device works by looking at how the driver react to the changes to both the driver and the device code. This led me to discover the descriptor queues and how it relates to the interrupt handlers. Using the knowledge gathered, I implemented the device in a way that makes the driver load itself into the operating system and gets itself up and running.

Both the *e1000* driver and the *igb* driver itself was the main source of knowledge. The driver code shows how both the devices evolved to what it is today by having large amounts of checks that checks for what kind of internal circuits the device connected contains. By looking at these checks,

one could guess how the device is built.

To learn about the device's behaviour, I began looking at the function named

```
static int igb_probe(struct pci_dev *pdev, const struct
   pci_device_id *ent)
```

trying to figure out how the device gets initialized by the driver. It is the first function the kernel calls during init. By placing printouts in the middle of the function, I could identify which code gets executed and what values is written to the MMIO address space of the device. By following its traces leads me into `mac`, `phy` and `nvm` ops functions that setups these internals.

```
s32 igb_read_nvm_eerd(struct e1000_hw *hw, u16 offset, u16
   words, u16 *data)
```

is the first function that I discovered. It reads the EEPROM. Since the device code implements the EEPROM incorrectly, the values it returns is just zero.

Later the driver calls

```
static s32 igb_init_phy_params_82575(struct e1000_hw *hw)
```

to read the PHY via the `MDIC`-register. Since the contents of the PHY is incorrect, the driver exits printing *"Hardware error"* to *dmesg*. By printing out where the driver reads off the `PHY`, I can identify which values that needs to be inserted into the *igb* device code to emulate it. By doing that, the driver registers the appropriate function pointers for the internal circuits it drives.

Chapter 8

Results

After having tried many different solutions to the problem, many of them not pointing in the right direction, this is the resulting framework I ended up with.

Many parts of the given framework were mostly scrapped. Some functions were removed, then merged with *e1000* device code, and modified to work with the newest QEMU master branch (as of December 2016). The changes done was to add a compatibility include header and adding missing C structs and functions to make it compile and run again. Some PHY-values had to be changed to be identical to those values found in the *igb* device described in the manual.

The big hurdle with this task is not having physical access (but indirectly via e-mailing my supervisor) to the real *igb* device, making it difficult to find a working way to implement either the real proper or dummy behaviour of specific parts of the *igb* device to make the driver execute without stopping somewhere. Even if you implement a behaviour the incorrect way, the driver may still execute well, but it will take the wrong turn in its code path, thus doing something else it should not do at all. The reason seems that the engineers who implemented the *igb* driver code did their work by extending the *e1000* driver code without removing the other extra code that cannot be executed on the real *igb* device. So if you implement the callbacks with any "random" dummy behaviour, it may provoke the driver to execute the extra code that was not meant to be executed. You are never really sure you are onto something "correct".

Even if I did not have the real device in front of me, it is possible to make informed guesses on how it should act by looking at the *igb* driver source code and relate it to the TCP/IP networking stack.

The contents of the framework is partly derived from both the existing *e1000* device code and my supervisor's *igb* framework. It both contains the C structures and functions from both of these modules. Many other functions that was copied from the *e1000* device code is commented out, because they would not compile on the newest version of QEMU.

The result of all this trial and error work is an empty framework shell of

the *igb* device that can load Linux's *igb* driver and make it look operational, but cannot send or receive any data yet.

To setup, load and run the *igb* device in QEMU, the development environment should consist of a modern workstation computer since its high performance is needed to compile the code at a workable pace. The QEMU source code is huge and requires a computer with a lot of memory and disk space to work with it. The CPU should be supported by the KVM driver to make executing it much faster, or else you would spend most of your time waiting for the Linux operating system to load each time you reload QEMU after recompiling it or when it crashes. I recommend also to write a script that sets the current working directory to the directory where the *igb* device driver code contains in the command-line shell to jump into it faster.

That setup needs to run the Linux operating system, since the programming environment that contains QEMU only supports that. There are many different distribution of this operating system to choose from, but they should at least contain the standard binary utilities and compilers from the GNU project, which all of them have. The programs from that project are used to build QEMU.

The framework consist of data structures and functions that represents the internal circuits and its behaviour. What you will see is empty and half-working implementations of the `FLASH`, `EEPROM` and `MMIO` address space of the device. Most of the MMIO address space are discovered using the printing technique mentioned before, but I think there are still more to look for.

The EEPROM is emulated by a read callback that is correctly implemented according to the manual. Its reading mechanism consist of basically reading the bits in the EEPROM and combining the read value with the necessary bits that indicates a successful read to bypass the checksum check of the EEPROM in the driver.

Linux separate the DMA engine's and the system's address space, where each space is given unique memory addresses. This mechanism is emulated by QEMU in the form of two functions called `pci_dma_read()` and `pci_dma_write()`.

The driver and the device contains rings and queues for handling 16 transmit and receive queues that is shared between the driver and the device. The driver gets notified of incoming and outgoing packets by interrupts made by the device. The driver writes to the descriptors and maps the packet buffer to it. The DMA transfers that mapped buffer to the device during transmit. There are two kinds of interrupt mechanism that the device supports called "legacy" and MSI-X. An existing implementation to emulate the former already exist in QEMU, while the latter do exist, but are currently not used in this implementation. The legacy interrupt mechanism is still there to be compatible with old driver code that the engineers reused. The older *e1000* device code

provides an implementation for it that can be reused to drive a *single* descriptor queue.

There is a macro called *DBGOUT()* that can be placed inside the functions to print out the value of the parameters passed to the device by the guest operating system's driver.

```
#define DBGOUT(what, fmt, ...) printf("igb:␣" fmt, ##
    __VA_ARGS__)
```

Listing 8.1: `DBGOUT()`

8.1 MMIO

By printing out the memory addresses the driver reads and writes to in the MMIO address space and look them up in the *igb* manual, I discovered new registers that are not emulated yet by the device code. A list of these registers are provided below:

```
#define IGB_RXPBS 0x2404 /* RX PB Size - RW */
#define IGB_RA2 0x54e4

/* Statistics for IGB. */
#define IGB_CBTMPC 0x402c
#define IGB_HGORCL 0x4128
#define IGB_HGORCH 0x412c
#define IGB_HGOTCL 0x4130
#define IGB_HGOTCH 0x4134
#define IGB_LENERRS 0x4138

/* Time sync register descriptions. */
#define IGB_SYSTIML 0xb600
#define IGB_SYSTIMH 0xb604
#define IGB_TIMINCA 0xb608

/* Extended Interrupt Cause. WO */
#define IGB_EICS       0x1520

/* VT Extended Interrupt Cause Set. WO */
//#define IGB_VTEICS     0x1520

/* Interrupt Vector Allocation Registers. R/W */
#define IGB_IVAR       0x1700

/* Split and Replication Receive Control Register queue. R/
    W */
#define IGB_SRRCTL     0x280c
```

```
/* DMA Tx Control. R/W */
#define IGB_DTXCTL      0x3590

/* DMA Tx Switch control. R/W */
#define IGB_DTXSWC      0x3500

/* Receive Long Packet Maximum Length. R/W */
#define IGB_RLPML       0x5004

/* Multiple Receive Queues Command Register. R/W */
#define IGB_MRQC        0x5818

/* Next Generation VMDq Control register R/W */
#define IGB_VT_CTL      0x581c

/* 5-tuple Queue Filter. R/W */
#define IGB_FTQF        0x59e0

/* EType Queue Filter. R/W */
#define IGB_ETQF        0x5cb0

/* VM Offload Register. R/W */
#define IGB_VMOLR       0x5ad0

/* Replication Offload Register. R/W */
#define IGB_RPLOLR      0x5af0

/* Time Sync RX Configuration. R/W */
#define IGB_TSYNCRXCFG 0x5f50

/* TX Time Sync Control Register. R/W */
#define IGB_TSYNCTXCTL 0xb614

/* TX Timestamp Value Low. RO */
#define IGB_TXSTMPL     0xb618

/* TX Timestamp Value High. RO */
#define IGB_TXSTMPH     0xb61c

/* RX Time Sync Control Register. R/W */
#define IGB_TSYNCRXCTL 0xb620

/* RX Timestamp Low. RO */
#define IGB_RXSTMPL     0xb624

/* RX Timestamp High. RO */
#define IGB_RXSTMPH     0xb628
```

```
/* Receive Queue Drop Packet Count. RC */
#define IGB_RQDPC      0xc030
```

Listing 8.2: Memory offsets to new *igb* MMIO address space registers.

These new offsets has been added to the table of callbacks in the device code and a *read-* and *write* stub function is registered to them. The implementation of the discovered registers shared with the *e1000* device are copied from the older *e1000* device code, as seen below:

```
// Register callbacks.
#define REGS(name, ret, params, ...) \
    static ret (*name[])(IGBState *, params) = { \
        __VA_ARGS__ \
    }; \
    enum { n##name = ARRAY_SIZE(name) };

#define rd(x)    [x] = macrd
#define stat(x) [x] = statrd
REGS(rdops, uint32_t, int,
    // MMIO address space.
    rd(E1000_PBA), rd(E1000_RCTL), rd(E1000_TDH), rd(
       E1000_TXDCTL), rd(E1000_WUFC), rd(E1000_TDT), rd(
       E1000_CTRL), rd(E1000_LEDCTL), rd(E1000_MANC), rd(
       E1000_MDIC), rd(E1000_SWSM), rd(E1000_STATUS), rd(
       E1000_TORL), rd(E1000_TOTL), rd(E1000_IMS), rd(
       E1000_TCTL), rd(E1000_RDH), rd(E1000_RDT), rd(
       E1000_VET), rd(E1000_ICS), rd(E1000_TDBAL), rd(
       E1000_TDBAH), rd(E1000_RDBAH), rd(E1000_RDBAL), rd(
       E1000_TDLEN), rd(E1000_RDLEN), rd(E1000_RDTR), rd(
       E1000_RADV), rd(E1000_TADV), rd(E1000_ITR), rd(
       E1000_CTRL_EXT), rd(E1000_FWSM), rd(E1000_SW_FW_SYNC
       ), rd(E1000_GCR),  rd(E1000_IAM),  rd(E1000_WUC), rd
       (E1000_RXCSUM), rd(E1000_MRQC), rd(E1000_RSSRK), rd(
       E1000_EEMNGCTL), rd(IGB_DTXCTL), rd(IGB_RPLOLR), rd(
       IGB_DTXSWC), rd(IGB_RXPBS), rd(IGB_TXSTMPH), rd(
       IGB_RXSTMPL), rd(IGB_RXSTMPH), rd(IGB_RLPML), rd(
       IGB_VT_CTL), rd(IGB_TSYNCRXCTL), rd(IGB_EICS), rd(
       IGB_TSYNCTXCTL), rd(IGB_TSYNCRXCFG), rd(IGB_MRQC),
       rd(IGB_TXSTMPL), rd(IGB_SYSTIML), rd(IGB_SYSTIMH),
       rd(IGB_TIMINCA),

    /* Statistics-registers. They are read-only and resets
       to 0 on read. */
    stat(E1000_ALGNERRC), stat(E1000_SYMERRS),  stat(
       E1000_RXERRC), stat(E1000_SCC), stat(E1000_ECOL),
       stat(E1000_MCC), stat(E1000_LATECOL), stat(
       E1000_COLC), stat(E1000_DC), stat(E1000_TNCRS), stat
```

```
   (E1000_SEC), stat(E1000_CEXTERR), stat(E1000_RLEC),
   stat(E1000_XONRXC), stat(E1000_XONTXC), stat(
   E1000_XOFFRXC), stat(E1000_XOFFTXC), stat(
   E1000_FCRUC), stat(E1000_PRC64), stat(E1000_PRC127),
    stat(E1000_PRC255), stat(E1000_PRC511), stat(
   E1000_PRC1023), stat(E1000_PRC1522), stat(E1000_GPRC
   ), stat(E1000_BPRC), stat(E1000_MPRC), stat(
   E1000_GPTC), stat(E1000_GORCL), stat(E1000_GORCH),
   stat(E1000_GOTCL), stat(E1000_GOTCH), stat(
   E1000_RNBC), stat(E1000_RUC), stat(E1000_RFC), stat(
   E1000_ROC), stat(E1000_RJC), stat(E1000_MGTPRC),
   stat(E1000_MGTPDC), stat(E1000_MGTPTC), stat(
   E1000_TORL), stat(E1000_TORH), stat(E1000_TOTL),
   stat(E1000_TOTH), stat(E1000_TPR), stat(E1000_TPT),
   stat(E1000_PTC64), stat(E1000_PTC127),stat(
   E1000_PTC255), stat(E1000_PTC511), stat(
   E1000_PTC1023), stat(E1000_PTC1522), stat(E1000_MPTC
   ), stat(E1000_BPTC), stat(E1000_TSCTC), stat(
   E1000_TSCTFC), stat(E1000_IAC), stat(E1000_ICRXPTC),
    stat(E1000_ICRXATC), stat(E1000_ICTXPTC), stat(
   E1000_ICTXATC), stat(E1000_ICTXQEC), stat(
   E1000_ICTXQMTC), stat(E1000_ICRXDMTC), stat(
   E1000_ICRXOC), stat(IGB_CBTMPC), stat(IGB_HGORCL),
   stat(IGB_HGORCH), stat(IGB_HGOTCL), stat(IGB_HGOTCH)
   , stat(IGB_LENERRS),

[E1000_TOTH] = rdclr8,    [E1000_TORH] = rdclr8,   [
   E1000_GPRC] = rdclr4,
[E1000_GPTC] = rdclr4,    [E1000_TPR]  = rdclr4,   [
   E1000_TPT]  = rdclr4,
[E1000_ICR]  = icrrd,  [E1000_EECD] = get_eecd, [
   E1000_EERD] = eerd,
[E1000_CRCERRS ... E1000_MPC]          = &macrd, /* bao
   : redefinition of regs? */
[E1000_RA      ... E1000_RA+8*32]      = &macrd,
[IGB_RA2       ... IGB_RA2+8*8]        = &macrd,
[E1000_MTA     ... E1000_MTA+4*128]    = &macrd,
[E1000_VFTA    ... E1000_VFTA+4*128]   = &macrd,
[E1000_RXDCTL  ... E1000_RXDCTL+256*2] = &macrd,
[IGB_SRRCTL    ... IGB_SRRCTL+256*2]   = &macrd,
[E1000_RSSRK   ... E1000_RSSRK+4*10]   = &macrd,
[E1000_RETA    ... E1000_RETA+4*32]    = &macrd,
[IGB_VMOLR     ... IGB_VMOLR+4*8]      = &macrd,
[IGB_IVAR      ... IGB_IVAR+4*8]       = &macrd,
[IGB_RQDPC     ... IGB_RQDPC+0x40*16]  = &macrd,
[IGB_FTQF      ... IGB_FTQF+4*8]       = &macrd,
[IGB_ETQF      ... IGB_ETQF+4*8]       = &macrd,
```

```
)

#define COMMA ,
#define wr(x) [x] = macwr
REGS(wrops, void, int COMMA uint32_t,
    wr(E1000_PBA), wr(E1000_EERD), wr(E1000_SWSM), wr(
       E1000_WUFC), wr(E1000_TDBAL), wr(E1000_TDBAH), wr(
       E1000_TXDCTL), wr(E1000_RDBAH), wr(E1000_RDBAL), wr(
       E1000_LEDCTL), wr(E1000_VET), wr(E1000_CTRL_EXT), wr
       (E1000_SW_FW_SYNC), wr(E1000_GCR), wr(E1000_IAM), wr
       (E1000_WUC), wr(E1000_RXCSUM), wr(E1000_RXDCTL), wr(
       E1000_MRQC), wr(IGB_RLPML), wr(IGB_DTXCTL), wr(
       IGB_RPLOLR), wr(IGB_DTXSWC), wr(IGB_VT_CTL), wr(
       IGB_RXPBS), wr(IGB_TSYNCRXCTL), wr(IGB_EICS), wr(
       IGB_TSYNCTXCTL), wr(IGB_TSYNCRXCFG), wr(IGB_TXSTMPH)
       , wr(IGB_RXSTMPL), wr(IGB_RXSTMPH), wr(IGB_MRQC), wr
       (IGB_TXSTMPL), wr(IGB_SYSTIML), wr(IGB_SYSTIMH), wr(
       IGB_TIMINCA),

    [E1000_TDLEN] = set_dlen,  [E1000_RDLEN] = set_dlen,
             [E1000_TCTL] = set_tctl,
    [E1000_TDT]   = set_tctl,  [E1000_MDIC]  = mdic,
                [E1000_ICS]  = set_ics,
    [E1000_TDH]   = set_16bit, [E1000_RDH]   = set_16bit,
            [E1000_RDT]  = set_rdt,
    [E1000_IMC]   = set_imc,   [E1000_IMS]   = set_ims,
              [E1000_ICR]  = set_icr,
    [E1000_EECD]  = set_eecd,  [E1000_RCTL]  =
       set_rx_control, [E1000_CTRL] = ctrlwr,
    [E1000_RDTR]  = set_16bit, [E1000_RADV]  = set_16bit,
            [E1000_TADV] = set_16bit,
    [E1000_ITR]   = set_16bit,

    [E1000_RA     ... E1000_RA+8*32]      = &macwr,
    [IGB_RA2      ... IGB_RA2+8*8]        = &macwr,
    [E1000_MTA    ... E1000_MTA+4*128]    = &macwr,
    [E1000_VFTA   ... E1000_VFTA+4*128]   = &macwr,
    [E1000_RXDCTL ... E1000_RXDCTL+256*2] = &macwr,
    [IGB_SRRCTL   ... IGB_SRRCTL+256*2]   = &macwr,
    [E1000_RSSRK  ... E1000_RSSRK+4*10]   = &macwr,
    [E1000_RETA   ... E1000_RETA+4*32]    = &macwr,
    [IGB_VMOLR    ... IGB_VMOLR+4*8]      = &macwr,
    [IGB_IVAR     ... IGB_IVAR+4*8]       = &macwr,
    [IGB_RQDPC    ... IGB_RQDPC+0x40*16]  = &macwr,
    [IGB_FTQF     ... IGB_FTQF+4*8]       = &macwr,
    [IGB_ETQF     ... IGB_ETQF+4*8]       = &macwr,
```

```
)
```

Listing 8.3: Register callbacks.

The above code shows all the registers in the MMIO address space that the driver touches. All the `E1000_*` constants are offsets to registers which the *igb* device shares with the *e1000* device. Some modifications are done to the arrays of MMIO registers of the old *e1000* device code[9]. The `RA`-registers are increased to 8*32 bytes and a new `RA2` 8*8 bytes array are added at offset `0x54e0`[14, page 530]. The `MTA`-size was increased. Some *Time Sync* registers was added. The `Random` register is added, but its implementation currently does not generate any random values.

There are placeholder functions registered to some of the MMIO registers shared with the *e1000* device that contains the old *e1000* device code to do the task the *e1000* way, which may break the *igb* driver.

8.2 Device structure

The framework provides a C structure that contains all state of the *igb* device. The field in the struct that stores the register's content in the MMIO address space is increased to be able to store a value for all possible offsets that can be made of a 16-bit string.

```
typedef struct {
    /*< private >*/
    PCIDevice parent_obj;

    /*< public >*/
    NICState *nic;
    NICConf conf;

    // Memory regions.
    MemoryRegion mmio;
    MemoryRegion io;
    MemoryRegion flash;

    uint32_t mac[0xffff]; // Media Access Control (MAC)
       registers.
    uint16_t phy[NPHY];   // Physical device (PHY)
       registers.
    uint16_t eeprom[64];  // EEPROM data.

    uint32_t rxbuf_size;
    uint32_t rxbuf_min_shift;

    struct {
        uint32_t val_in;    // shifted in from guest driver
```

```
        uint16_t bitnum_in;
        uint16_t bitnum_out;
        uint16_t reading;
        uint32_t old_eecd;
    } eecd; // EECD-register.

    /* Timers */
    QEMUTimer *autoneg_timer;
    QEMUTimer *mit_timer;      /* Mitigation timer. */
    bool mit_timer_on;         /* Mitigation timer is
       running. */
    bool mit_irq_level;        /* Tracks interrupt pin
       level. */
    uint32_t mit_ide;          /* Tracks E1000_TXD_CMD_IDE
       bit. */

    uint32_t compat_flags;
} IGBState;
```

Listing 8.4: 82576's descriptors.

8.3 New advanced descriptors

The new descriptors for the *igb* was defined and added to device code.

```
// 82576's descriptors.
union tdesc {
    // Legacy transmit fetch descriptor.
    // Old descriptor provided to be backward compatible.
    struct {
        uint64_t     bufaddr : 64; /* DMA address to buffer
            */
        unsigned int length  : 16; /* Length of buffer. Max
            is 9728 bytes. */
        unsigned int cso     : 8;  /* Checksum offset. */
        unsigned int cmd     : 8;  /* Descriptor command.
           */
        unsigned int sta     : 4;  /* Status. */
        unsigned int extcmd  : 3;  /* Extended descriptor
           command. */
        unsigned int css     : 8;  /* Checksum start field.
            */
        unsigned int vlan    : 16; /* Virtual LAN. */
    } __attribute__((packed)) fetch;

    // Advanced transmit context descriptor.
    // Setup once to avoid a descriptor per packet.
```

```
struct {
    unsigned int iplen          : 9;  /* IP length */
    unsigned int maclen         : 7;  /* MAC length */
    unsigned int vlan           : 16; /* Virtual LAN */
    unsigned int ipsec_sa_i     : 8;  /* IPsec SA index.
        */
    unsigned int rsv1           : 24; /* Reserved. */
    unsigned int ipsec_esp_len : 9;  /* Size of the ESP
        trailer and ESP ICV appended by software. */
    unsigned int tucmd          : 11; /* TCP/UDP command
        field. bit 10-6 reserved. bit 5 encryption. bit
        4 IPSEC_TYPE. bit 3:2 L4T. bit 1 IPV4. SNAP bit
        0. */
    unsigned int dtyp           : 4;  /* Descriptor type
       . 0000b for TCP/IP context transmit descriptor
       type. */
    unsigned int rsv2           : 5;  /* Reserved */
    unsigned int dext           : 1;  /* Descriptor
       extension */
    unsigned int rsv3           : 6;  /* Reserved */
    unsigned int idx            : 3;  /* Index */
    unsigned int rsv4           : 1;  /* Reserved */
    unsigned int l4len          : 8;  /* Layer 4 header
       length? */
    unsigned int mss            : 16; /* Maximum segment
        size. */
} __attribute__((packed)) ctx;

// Advanced transmit data descriptor.
struct {
    uint64_t     bufaddr : 64; /* Address of descriptor
       's data buffer. */
    unsigned int dtalen  : 16; /* Length in bytes of
       data buffer at the address pointed to by this
       specific descriptor. */
    unsigned int rsv1    : 2;  /* Reserved */
    unsigned int mac     : 2;  /* bit 0: ILSec - Apply
       MACSec on packet. 1588 (bit 1) - IEEE 1588
       timestamp packet. */
    unsigned int dtyp    : 4;  /* Type of descriptor.
       0011b */
    unsigned int dcmd    : 8;  /* Descriptor command */
    unsigned int sta     : 4;  /* Status */
    unsigned int idx     : 3;  /* Index */
    unsigned int cc      : 1;  /* Why does the manual
       say it is reserved and set to 0b? */
    unsigned int popts   : 6;  /* Flags */
```

```
        unsigned int paylen   : 18; /* Payload length (
           without header?) */
    } __attribute__((packed)) data;

    // Advanced transmit descriptor write-back format
    // An alternative to the DD-bit. It is written back to
       indicate that a descriptor has been processed.
    struct {
        uint64_t     rsv1 : 64; /* Reserved */
        unsigned int rsv2 : 32; /* Reserved */
         unsigned int sta  : 4; /* Status. Bit 0 is
            Descriptor Done (DD). Bits 1-3 is reserved. */
        unsigned int rsv3 : 28; /* Reserved */
    } __attribute__((packed)) wb;
};
```

Listing 8.5: 82576's descriptors

8.4 Initialize and cleanup.

When the *igb* device is loaded by QEMU, it executes a function that initializes the device. The code that registers this function into the PCI subsystem can be reused from the *e1000* device code, but its implementation needs to be modified to accommodate the differences between the *e1000* device and the *igb* device.

The new implementation consist of implementing code that initialize the extra internals that Intel added to the *igb* device, and modify the existing code that initialize the previously existing internals to follow the new changes.

The *cleanup* function unregisters the device and unallocates claimed resources from the operating system.

8.5 EEPROM

At first, to pass the checksum check at load, we need to implement the EEPROM's behaviour. To implement the EEPROM, one does simply implement the callback for the *EERD*-register according to the format provided in Intel's *igb* manual.

```
static uint32_t eerd(E1000State *s, int x)
{
    unsigned int i;
    uint32_t eerd = s->mac[EERD];
```

```
    /* Return EERD-register content if Start Bit is not set
       . */
    if ((eerd & E1000_EEPROM_RW_REG_START) == 0)
        return eerd;

    /* Check if reading outside the EEPROM memory area. */
    if ((i = eerd >> 2) > EEPROM_CHECKSUM_REG)
        return eerd | E1000_EEPROM_RW_REG_DONE;

    /* Set EERD with the EEPROM data. */
    s->mac[EERD] = (eerd | (s->eeprom[i] << 16) | 2);
    return s->mac[EERD];
}
```

Listing 8.6: EERD-register implementation

The function reads the value written to `EERD` from the `s->mac[EERD]` and checks if the start bit is not set. If it is not set, it returns the current content of this register. Then it checks if the driver tries to read outside the EEPROM space. If true, it returns `EERD` with the *R/W* done-bit set to indicate a read was done. If not, it gathers the value asked from the EEPROM, inserts it into the `EERD` register, sets a bit to indicate a successful read from it, and then returns the value of the `EERD`-register.

Optionally, you could dump the EEPROM from the *igb* device and insert that into the EEPROM data structure to mirror the exact ROM, but the driver does only verify its checksum making it unnecessary to do it.

8.6 PHY (MDIC)

The PHY-registers are read via the `MDIC`-register by the driver. Some PHY identifiers have been changed and are different between the *e1000* device and the *igb* device. Changing them to the correct values defined in the *igb* manual makes the *igb* driver execute through another code path and cause new errors printed out in the *device messages*-log, accessed by executing the command *dmesg* in a terminal.

```
    static const uint16_t phy[] = {
        ...
        [PHY_ID1] = 0x2a8, [PHY_ID2] = 0x391, // p. 654 in
           IGB manual.
        ...
    };
```

Listing 8.7: New default PHY values.

My supervisor discovered that the `Page Select Core Register (register 31)` in PHY was missing that caused a "Hardware Error"

message to show up in *dmesg*. This message was squashed by adding the missing register to the code.

```
static const char phy_regcap[N_PHY_REGS] = {
    ...
    [PHY_AUTONEG_EXP] = PHY_R, [31] = PHY_W,
    ...
};
```

Listing 8.8: `Page Select Core Register`

8.7 Statistics

The *igb* device driver saves multiple counters that counts occurrences of things that happen on the *igb* device in the function `igb_update_stats()`[17][line 5210]. It computes the count by reading the `statistics`-registers and increments its respective variables. The counts on each `statistics`-register is reset to 0 each time it is read.

```
/* Reads a statistics-register in the MMIO address space,
   zeroing it after read. */
static uint32_t mac_statreg(IGBState *s, int index)
{
    const uint32_t ret = mac_readreg(s, index);
    mac_writereg(s, index, 0);
    return ret;
}
```

8.8 Descriptor queues

There are some registers in the MMIO address space that are used to control the transmit and receive process.

The `TCTL` and `RCTL` registers are used to control the transmit and receive process. The `TDBAH`, `TDBAL`, `RDBAH`, `RDBAL` are used to access and modify the descriptor queue ring. The `ICR`-register is written to by the *igb* device to indicate the *cause* of an interrupt that has been asserted. The `CTRL_EXT` is written to by the driver to enable some extra features, like VLAN. I think it can be ignored for now. The descriptor queues in the driver and the device share MSI-X interrupts to indicate that new packets has appeared in the queues.

8.9 Legacy interrupt

One of the challenges I encountered was the driver complaining about *transmit queue 0*-error that is reported by the driver when trying to

"up" it (enable it making it operational). The cause of it was that the `E1000_ICR_INT_ASSERTED`-bit was not set by the *igb* device code each time an interrupt was asserted. That caused the driver to ignore the interrupt by returning the value `IRQ_NONE` each time the interrupt handler was called. The interrupt handler assumes that the bit is set each time there is an interrupt. If it is not set, the interrupt is considered spurious.

The fix to this issue is to make the function that sets the `ICR`-register, called `set_ics()`, set the `E1000_ICR_INT_ASSERTED`-bit each time it is called to indicate to the driver that a real interrupt is asserted by the *igb* device. That eliminates the error the driver complains about.

8.10 VMState

The code contains an object of the C structure `VMState` which I assumes stores the state of the device when the virtual machine is paused. Currently, it stores the current state of all the registers of the `MMIO` and `PHY`. I have not updated this to contain and save all the new registers I have added, which may break QEMU's ability to pause and save the virtual machine's state.

Chapter 9

Conclusion

9.1 Summary

I was given the task to compare the differences of the *e1000* device and the *igb* device, and use the discoveries found to fix the given non-working *igb* device emulator framework, which would later enable us to create operable virtual *igb* devices. A technique was developed to discover the internals of the *igb* device that consist of printing values in the *igb* driver and having callback functions that prints out the parameters the driver is passing to the device. Due to not having the real device available in front of me, I had to discover the device by tracing the driver and implement code based on informed assumptions of how I thought the device works. Since the device was a hidden black box to me, the process of implementing it came down to guesses made from the manuals and my discoveries.

New registers were discovered in the `MMIO` address space and the `PHY`-space, which were added to the device code. The `FLASH` space was discovered to not be used at all by the *igb* driver. Implementing the `EERD`-register made the *igb* driver continue into setting up descriptor rings and creating descriptor queues, which it fails to use. The end result was an empty framework shell with some working implemented functions that is able to load the driver into the operating system. The driver would then not have the ability to transmit and receive packets.

Future work could be to implement emulating those MSI-X descriptor queues which transmits and receives packets using the network API provided by Linux, thus bringing us nearer of reaching the goal of creating a SR/IOV-capable network device in QEMU.

9.2 Open problems

This section is written as a proposal of a possible implementation basing it upon my experiences after spending many months trying to figure out how this device works and implement it by sheer trial and error. It consists of

proposals to build a final working *igb* device implementation, which may be implemented by another student in the future.

9.2.1 MMIO

To further implement the missing MMIO registers where I placed stubs, I recommend tracing the *igb* driver trying to figure out which registers it modifies and what behaviour it expects to continue without encountering on an error. Use the printing method to follow the driver's code flow, then make changes to the *igb* device code to advert its execution to success.

9.2.2 Descriptor queues

These descriptor queues can be emulated by implementing a queue that contains those descriptors following the format described in the previous section, and then adding mechanisms that transforms these new "advanced" descriptors into packets that can be transmitted on the network by the host operating system's network driver. To implement MSI-X, there is an existing API in `qemu/hw/msix.h` that can be used to install a MSI-X interrupt into a descriptor queue provided by QEMU.

QEMU does provide some ready-made API functions to send and receive data over the network using the host operating system that can be used to implement this functionality. These functions can be found in `qemu/net/net.c`. Currently, the *e1000* device code seems to provide an implementation for the old descriptor queue using the old descriptors that transmits the packets using the sockets API (SLIRP) by translating those descriptors into SLIRP packets, which is either a TCP or a UDP packet. There is also a *TAP* API provided by Linux that can be used to transmit those packets to Linux's TAP device accessed in `/dev/net/tap` in the file system. The TAP device accepts Ethernet frames instead of SLIRP packets, which may let you do the task of building the TCP and UDP packets yourself from the descriptors provided by the driver.

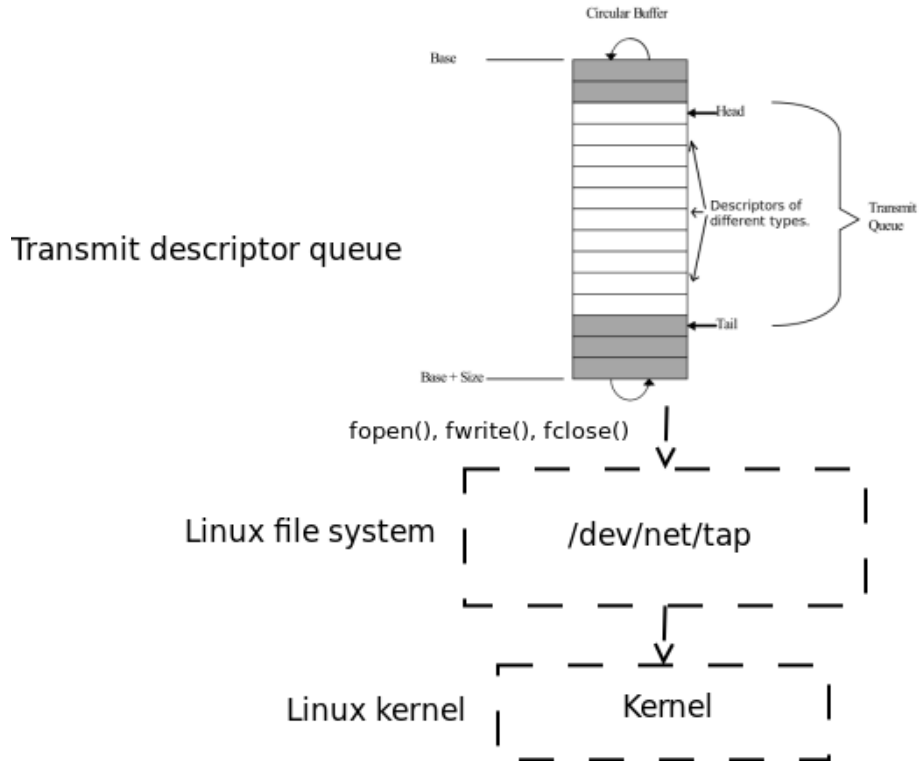

Figure 9.1: A proposal for implementing the descriptor queues using the *TAP* API.

To implement the new form of the descriptor queue, you need to change the old way to transform the new descriptors. The new descriptors are in a new format that needs to be transformed to a format recognizable by Linux's networking API. When that is done, the device needs to overwrite that descriptor it has read with the *write-back* descriptor to indicate to the driver that the device has handled it. Then the device needs to notice the driver by interrupting it either using the legacy interrupt way or the MSI-X interrupt way. When MSI-X is implemented properly, the driver will detect that this capability exists and switch to it automatically.

The placeholder functions taken from the *e1000* device code can be modified to do this task *igb* way.

9.2.3 SR/IOV

When all of the above is done, I guess the *SR/IOV*-capability can be added by building a derived virtual PCI-Express configuration space as my supervisor did in his patches[12]. There are new `VF`-registers[14, page 667] in the `MMIO` address space that needs to be implemented to do that job.

References

[1] archlinux.org. *Arch Linux*.

[2] *Are VMDq and SR-IOV performing the same function*. URL: http://windowsitpro.com/virtualization/q-are-vmdq-and-sr-iov-performing-same-function (visited on Sept. 23, 2016).

[3] Fabrice Bellard. *QEMU, a Fast and Portable Dynamic Translator*. URL: https://www.usenix.org/legacy/events/usenix05/tech/freenix/full_papers/bellard/bellard_html/.

[4] Ravi Budruk. *PCI Express(R) Basics*. 2007.

[5] Ravi Budruk, Don Anderson, and Tom Shanley. *PCI Express System Architecture*. MindShare Inc., 2003.

[6] Benoît Canet. *QEMU-logo*. URL: http://wiki.qemu.org/File:Qemu-logo.png (visited on Apr. 16, 2015).

[7] *Conventional memory*. URL: http://www.petesqbsite.com/sections/tutorials/zines/chronicles/issue3_files/Figure_1-1.gif (visited on Sept. 23, 2016).

[8] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. 3rd ed. O'Reilly, Feb. 2005. ISBN: 0596005903.

[9] *e1000.c of SR/IOV patches v3*. URL: https://github.com/knuto/qemu/blob/sriov_patches_v3/hw/net/e1000.c (visited on Mar. 28, 2017).

[10] *HP Compaq PC*. URL: http://g-ecx.images-amazon.com/images/G/02/uk-electronics/product_content/HP/CompaqQ42011/Get-the-complete-package._V163679618_.jpg (visited on May 12, 2016).

[11] *IBM PC Model 5150*. URL: https://upload.wikimedia.org/wikipedia/commons/f/f1/Ibm_pc_5150.jpg (visited on May 12, 2016).

[12] *igb.c of SR/IOV patches v3*. URL: https://github.com/knuto/qemu/blob/sriov_patches_v3/hw/net/igb.c (visited on Mar. 28, 2017).

[13] Intel. *Intel 4670k*. URL: http://ark.intel.com/products/75048/Intel-Core-i5-4670K-Processor-6M-Cache-up-to-3_80-GHz.

[14] Intel. *Intel(R) 82576EB Gigabit Ethernet Controller Datasheet*.

[15] Intel. *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual*.

[16] *Intel i7 Haswell specifications*. URL: http://ark.intel.com/products/family/75023/4th-Generation-Intel-Core-i7-Processors#@All.

[17] *Linux IGB driver module source code*. URL: http://lxr.free-electrons.com/source/drivers/net/ethernet/intel/igb/igb_main.c (visited on Apr. 4, 2016).

[18] *Linux socket buffers*. URL: http://lxr.free-electrons.com/source/include/linux/skbuff.h (visited on Mar. 20, 2016).

[19] *Memory map of the Personal Computer*. URL: http://image.slidesharecdn.com/conventionalmemory-120903115032-phpapp02/95/conventional-memory-3-728.jpg?cb=1346673135 (visited on Sept. 23, 2016).

[20] *MMIO bar Register format*. URL: http://2we26u4fam7n16rz3a44uhbe1bq2.wpengine.netdna-cdn.com/wp-content/uploads/091613_1237_SystemAddre7.png (visited on Mar. 18, 2016).

[21] *Multithreaded devices*. URL: http://www.linux-kvm.org/images/a/a7/02x04-MultithreadedDevices.pdf (visited on Dec. 15, 2016).

[22] Knut Omang. *Version 3 of a patch for SR/IOV-support in QEMU*.

[23] *PCI Express lanes*. URL: http://computer.howstuffworks.com/pci-express1.htm (visited on Feb. 2, 2016).

[24] *PCI Express lanes*. URL: http://s.hswstatic.com/gif/pci-express-lanes.gif (visited on Nov. 22, 2016).

[25] *PCI ports*. URL: https://upload.wikimedia.org/wikipedia/commons/0/0c/PCI_und_PCIe_Slots.jpg (visited on Feb. 2, 2016).

[26] *PCI-SIG - Home*. URL: https://www.pcisig.com/home/ (visited on June 3, 2015).

[27] *QEMU Internals: Big picture overview*. URL: http://blog.vmsplice.net/2011/03/qemu-internals-big-picture-overview.html (visited on Sept. 22, 2016).

[28] *[Qemu-devel] [PATCH 0/4] pcie: Add support for Single Root I/O Virtualization*. URL: https://lists.nongnu.org/archive/html/qemu-devel/2014-08/msg05110.html (visited on Apr. 7, 2017).

[29] Tom Shandley and Don Anderson. *PCI System Architecture - Third Edition*. MindShare, Inc., 1999.

[30] *Sony VAIO Pro 13 Touch Ultrabook*. URL: http://store.sony.com/vaio-pro-13-touch-ultrabook-zid27-SVP13215PX//cat-27-catid-All-13-Ultrabook-Pro?vva_ColorCode=E2E3DE (visited on May 15, 2015).

[31] *The E1000-driver*. URL: https://downloadcenter.intel.com/download/9180/Network-Adapter-Driver-for-Gigabit-PCI-Based-Network-Connections-for-Linux- (visited on Feb. 12, 2016).

[32] *The Intel 64 and IA-32 Architectures Software Developer's Manuals*. URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (visited on Apr. 11, 2016).

[33] *Von Neumann architecture*. URL: https://en.wikipedia.org/wiki/Von_Neumann_architecture#/media/File:Computer_system_bus.svg (visited on Nov. 30, 2016).

[34] Yu Zhao. *PCI Express I/O Virtualization Howto*. URL: https://www.kernel.org/doc/Documentation/PCI/pci-iov-howto.txt (visited on June 5, 2015).