Slovenská Technická Univerzita v Bratislave
Fakulta Informatiky a Informačných Technológií
Študijný program: Informatika

Jozef Hopko

Ovládač pre operačný systém MINIX 3

Bakalársky projekt

Vedúci projektu: Ing. Matej Košík
Máj, 2007

ANOTÁCIA:

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný odbor: INFORMATIKA
Študijný program: Informatika

Autor: Jozef Hopko
Bakalársky projekt: Ovládač pre operačný systém MINIX3
Vedenie bakalárskeho projektu: Ing. Matej Košík
Máj, 2007

Obsahom tejto práce je analyzovať princíp fungovania ovládačov zariadení v operačnom systéme Minix3 a navrhnúť a implementovať ovládač pre vybrané zariadenie. Zariadením je zvuková karta tvorená ICH4 kontrolérom a AC'97 kodekom používaná v mobilných počítačoch. Na vytvorenie ovládača je potrebné dokonale poznať dané zariadenie ako aj systém pre ktorý ovládač vytvárame. Architektonicky môžme ovládač rozdeliť na hardvérovo nezávislú (vrchnú) a hardvérovo závislú (spodnú) časť. Vrchná časť tvorí univerzálne rozhranie. Komunikuje s operačným systémom ako aj s hardvérovo závislou časťou a je spoločná pre všetky zvukové karty. Spodná časť je závislá na konkrétnom zariadení a pri implementácií sa postupuje podľa špecifikácií výrobcu.

ANNOTATION:

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: INFORMATICS

Author: Jozef Hopko
Bachelor Theses: Device driver for MINIX3
Supervisor: Ing. Matej Košík
2007, May

This thesis analyzes functionality and architecture of device drivers in Minix3 operating system and design and implement selected driver. The target device is multimedia audio controller ICH4 with AC'97 codec used in notebooks. Driver implementation requires knowledge of the target device as well as the host system. The driver can by divided into two layers: hardware-independent (top) and hardware-dependent (bottom) layer. The top layer represents a universal interface. It communicates with operating system as well as with the bottom layer and it is the same for every sound card. The bottom layer is device dependent and the implementation is based on manufacturer specifications.

Prehlásenie

Týmto čestne prehlasujem že som predloženú bakalársku prácu vypracoval samostatne, s použitím uvedenej literatúry a na základe mojich vedomostí.

V Bratislave 11. máj 2007 Jozef Hopko

Contents

1 Introduction

This thesis analyzes the functionality and architecture of device drivers in Minix3 operating system. It also describes design and implemention of the selected driver. The target device is multimedia audio controller ICH4 with AC'97 codec used in notebooks. Driver implementation requires knowledge about the target device as well as the host system. The driver can by architectonically divided into two layers: hardware independent (top) and hardware dependent (bottom) layer. The top layer represents a universal interface. It communicates with operating system as well as with the bottom layer and it is the same for every sound card. The bottom layer is device dependent and the implementation is based on manufacturer specifications.

1.1 Motivation

Contemporary operating systems are essential elements in IT world. Lots of applications rely on these systems – from simple games over medical applications to whole industry. Therefore improving this system is essential for further development. Most of the today's operating system is very complex and not really a good starting point for studying. To understand more complex systems, we primarily need to understand the basic principles. This motivated us to a closer look at an interesting operating system called Minix.

1.2 About Minix3

The first version of Minix was introduced 1987. The author Andy Tanenbaum wrote an open source operating system for education purposes from scratch. Minix stands for mini-UNIX, which means that the system is UNIX compatible and simple enough to understand. The third edition of Minix is designed to be highly reliable, flexible, and secure. Minix3 [4] is microkernel based operating system. It is completely written in the C programming language and the kernel part has fewer than 4000 lines of executable code so it is extremely small.

2 Minix3 Architecture

As mentioned before, Minix3 is microkernel based system. The authors intention was to keep the system "Small is Beautiful" [13, page 17]. It is composed from multiple processes. Each of these processes belongs to one of the four layers, see Figure 1. Processes in the first layer run in the privileged mode. They have unrestricted access to everything. This layer contains only three processes: *kernel task*, *system task* and the *clock task*.

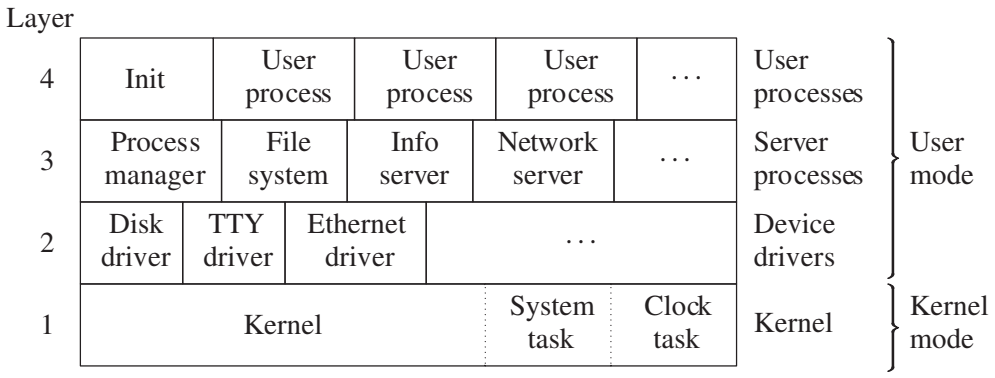

Figure 1: Internal structure of Minix3.

The *kernel task* contains everything what could not be moved elsewhere. Its most prominent services perhaps are IPC primitives (`send`, `receive`, `sendrec` etc). Closely related with IPC primitive is the scheduler of processes that is also part of the kernel. The *system task* implements so called *kernel calls*. These kernel calls are made available to device drivers and server processes (not for normal user processes). These kernel calls usually perform some action that cannot be performed in user mode but is essential for device drivers and servers.

Device drivers run in unprivileged user mode. They use the *system task* for talking to the raw hardware and provide more abstract interface of that particular hardware. Different device driver processes deal with different hardware devices.

Server processes run in unprivileged user mode too. Their services actually form the public interface of the operating system. User processes can use these services via so called *system calls*. The *process manager* implements various system calls related to processes. The *file system* implements various system calls related to dealing with files. It also mediates the communication of user space processes with device drivers, as explained in Section 2.1.

The fact that applications as well as device drivers run in the user mode has several benefits. Those parts that run in user mode are divided into small modules, well insulated from one another. The main goal is that a potential bug in a device driver or an application can not bring down the entire OS. As long as the drivers are not a

fixed part of kernel, the kernel can be kept really small and is easier to maintain and to debug. These features and other aspects greatly enhance the system reliability. In addition to this, the system is transparent and simple enough for a beginner too, so it is a good starting point to the world of operating systems.

2.1 Inter-Process Communication

An important part of a microkernel is a well designed IPC system. Since all programs are well insulated from each other a good data exchange mechanism is needed. Sending of messages between processes solve this problem in Minix3. One process can send a message to another by calling `send()` function, the target process obtains the message by calling `receive()` function. An effective IPC system must have low overhead.

However, when we need to communicate with some device driver, another data exchange scheme is used. As Figure 2 shows that all communication between user process and device driver goes through file system.

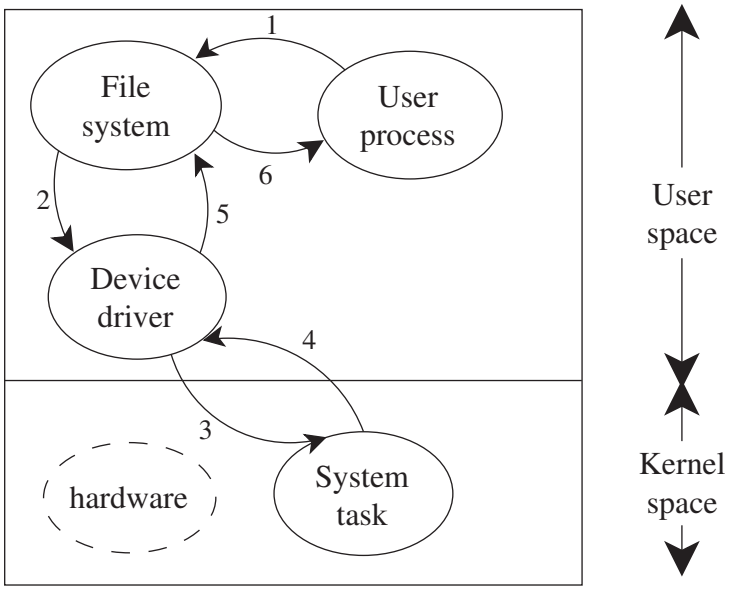

Figure 2: User process to driver communication.

Device drivers in UNIX like systems (what Minix surely is) are represented as files. These special files are created using `mknod` command that also assigns selected major and minor number to this file. File system manages his own process list to ensure correct message passing between user process and device driver. For example imagine a user process that reads a file `/dev/dummy` which has a major number `24`. Process call a `read()` function. This function simply sends a message with requested operation to file system process. File system process compares the major number of accessed file with major numbers of running device drivers and sends a message to corresponding

device driver. After the device driver replied a message or notification is send back to user process.

2.2 Sample driver

The file system provides the necessary abstraction between user and data or user and devices. When we for example read from a file we needn't to bother where the data are coming from. They can be placed on our local disk or received by the network card from a computer that is 1000 kilometers away. This abstraction is very welcomed and makes the programmers life easier (in most cases). To illustrate how device drivers work we take a look at the following driver example with a few explanations.

From the end user application perspective a device is a special file (for example `/dev/mixer`). Special files can be opened by user processes. When user process opens a special file, the file system informs the driver that a user process is trying to open the file. Once the special file is opened, the user process can read from it, or write to it, and on every action, the file system informs the driver. The driver can then reply with data in case of read, or return a status value in case of write.

Here is a simple program that accesses a device:

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

char buffer[100] = "0123456789";

int main(int argc, char** argv)
{
    int fd, count;

    /* Open special file fo reading */
    fd = open("/dev/dummy", O_RDONLY);
    if(fd  < 0) {
        fprintf(stderr, "ERROR: cannot open /dev/dummy.\n");
        return 1;
    }

    /* Read the data */
    count = read(fd , buffer, 100);

    /* Print the data */
    fprintf(stderr, "DEBUG: %s\n", buffer);

    /* Close special file */
    close(fd);
    return 0;
}
```

As you can see it is quite simple. The driver is a little bit more complicated, but in generally it would look something like this:

```
PUBLIC void main(void)
{
  message mess;
```

```
  int result;

  while (TRUE) {
    if (receive(ANY, &mess) != OK) continue;

    switch(mess.m_type) {
    case DEV_OPEN:
    case DEV_CLOSE:
        taskreply(TASK_REPLY, mess.m_source, mess.IO_ENDPT, OK);
        break;
    case DEV_READ:
        do_read(&mess);
        break;
    case DEV_WRITE:
        do_write(&mess);
        break;
    case DEV_STATUS:
        do_status(&mess);
        break;
    default:
        /* Reply to an unexpected message. */
            taskreply(TASK_REPLY, mess.m_source, mess.IO_ENDPT, EINVAL);
        break;
    }
  }
}
```

The main idea is that the driver runs in a loop and waits for messages. After the message is received it is processed and a reply is send back to process that requested specified operation. The example above is simplified and some functions are omitted but it is sufficient for our illustration.

To establish a connection between our driver and special file we need to create the special file. Special files are created with the command `mknod`. For example:

```
mknod /dev/dummy c 24 0
```

This creates a new special file named `/dev/dummy`. The `c` expresses that this device is a character device so it cannot handle random accesses requests. This special file will have major device number `24` and minor device number `0`. If more special files have the same major device number, they we will all be linked to the same driver. The minor number tells the driver which special file is accessed. So the driver can handle multiple special files with the same major number but different minor numbers. The sound driver is a good example of this because it handles `/dev/mixer`, `/dev/audio` and `/dev/rec`.
After the special file is created the executable has to be explicitly loaded as a driver. This can be done with the command service up. For example:

```
service up /sbin/sample_driver -dev /dev/dummy
```

The target operating system is Minix3. The first version (Minix) was developed for operating systems education purposes. In meantime the third version of Minix is designed to be highly reliable, flexible, and secure. Minix3 is microkernel based operating

system. It is completely written in C language and the kernel part has fewer than 4000 lines of executable code so it's extremely small.

The applications as well as device drivers run in the user mode. This approach has several benefits. The parts that run in user mode are divided into small modules, well insulated from one another. The main goal is that a potential bug in a device driver or an application can not bring down the entire OS. As long as the drivers are not a fixed part of kernel, the kernel can be kept really small and is easier to manage and to debug. These features and other aspects greatly enhance the system reliability. In addition to this, the system is transparent and simple enough for a beginner too, so it is a good starting point to the world of operating systems.

3 Hardware

For the driver development and testing was used Intel® Mobile Technology based notebook with ICH4 southbridge[1]. This technology is widely spread. Intel also offers a very good technical and developer resources so there is enough information on the web needed by a driver developer. To reveal what devices we are exactly dealing with, we simply run the `lspci` command under our Linux distribution. In the following output we can see this line:

```
Multimedia audio controller:  Intel Corporation 82801DB/DBL/DBM
(ICH4/ICH4-L/ICH4-M) AC'97 Audio Controller (rev 03)
```

According to product information *Intel® I/O Controller Hub (ICH4)* provides these features:

- High-speed USB 2.0
- AC'97 audio
- High Precision Event Timer
- System manageability bus
- Continual support for six PCI slots
- ATA 100
- LAN connect interface

[1] A chip on a motherboard that controls all onboard devices including the IDE bus and PCI bus

3.1 ICH4 Architecture

To figure out how the devices work, we need a at least basic information about their architecture. Figure 3 shows the whole system architecture but we will focus at the AC'97 Codec and the ICH4 controller. The AC'97 Codec is directly responsible for the sound output and ICH4 controls this codes [9].

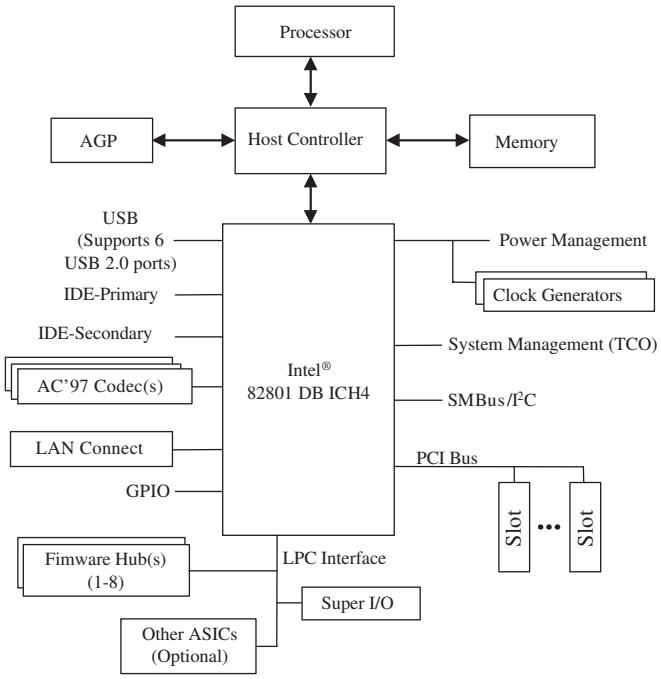

Figure 3: ICH4 architecture overview

Note that the codec is directly connected to the controller. So it is not a classical sound card we can pull from a PCI slot. The audio controller is integrated in the ICH4 controller and acts as PCI device[2]. Therefore we can work with the audio controller as with usual PCI device and need not care whether is it plugged in a PCI slot or not.

3.2 Accessing PCI devices

Every PCI device responds to configuration commands, and it can respond to I/O accesses and/or memory accesses. During the boot time, the BIOS or the OS sets the base address registers (BARs) through configuration space. BARs determine address ranges in I/O or memory space that a device should respond to. Obviously, those

[2]An integrated circuit fitted onto the motherboard itself is called a planar device in the PCI specification [1].

ranges should not be duplicated anywhere else in the I/O space or memory space on the same PCI bus [2].

PCI device has a set of registers that are used for communicating with the CPU. By writing into these registers, we can send data or perform some action. By reading from these registers we can learn what the device's state is. These registers are mapped into computer I/O space using the base addresses [12].

To access concrete device we can enumerate all PCI devices or as shown in Figure 4, we can access the device according to his Bus:Device:Function number. This approach is possible, because the mentioned devices are hardwired into the hardware and they have statically assigned numbers.

Bus:Device:Function	Function Description
Bus 0:Device 30:Function 0	Hub Interface to PCI Bridge
Bus 0:Device 31:Function 0	PCI to LPC Bridge
Bus 0:Device 31:Function 1	IDE Controller
Bus 0:Device 31:Function 3	SMBus Controller
Bus 0:Device 31:Function 5	AC '97 Audio Controller
Bus 0:Device 31:Function 6	AC '97 Modem Controller
Bus 0:Device 29:Function 0	USB UHCI Controller #1
Bus 0:Device 29:Function 1	USB UHCI Controller #2
Bus 0:Device 29:Function 2	USB UHCI Controller #3
Bus 0:Device 29:Function 7	USB 2.0 EHCI Controller
Bus n:Device 8:Function 0	LAN Controller

Figure 4: PCI Device and Functions

PCI devices inserted in PCI slots can not have predefined bus or device number. You can pull out the PCI card from one slot and put it into another. More precisely, you can decide on what bus and slot the device will be, so the software can not rely on predefined numbers. In this case PCI enumeration is used. The driver lists all devices connected to PCI bus, and asks for the Vendor and Product ID. After the device is located, operations can be performed.

However, our Audio Controller has a statically assigned Bus:Device:Function number. Minix3 offers us a couple of functions to access PCI configuration space provided that we know the Bus:Device:Function number. In the following example we will read the Vendor ID, Device ID and Mixer Base Address from Audio Controller configuration space:

```
/* PCII_RREG32_(Bus, Device, Function, Register)
 *   this macro reads 32bit value from PCI configuration space
 */
```

```
#include "../pci/pci_intel.h"

void main()
{
    u16_t vendor_id;
    u16_t device_id;
    u32_t mixer_base;

    vendor_id  = PCII_RREG16_(0, 31, 5, 0x00); /* Bus0:Dev31:Func5:Reg0 */
    device_id  = PCII_RREG16_(0, 31, 5, 0x02);
    mixer_base = PCII_RREG32_(0, 31, 5, 0x10);
}
```

This example simply illustrates the access to PCI configuration space of specified device in Minix3. Description of AC '97 Audio PCI Configuration Space is included in Section 6 (Technical documentation). If you are interested how the PCII_RREG/PCII_WREG macro works please refer to Software Generation of Configuration Transactions and `pci_intel.h` listing also included in Section 6.

3.3 AC'97 Codec

There are many AC'97 compliant codecs from various manufacturers. In our case the codec is marked as CS4201 and manufactured by Cirrus Logic®.

The CS4201 is a mixed-signal serial audio codec with integrated headphone power amplifier compliant with the Intel® Audio Codec '97 Specification, revision 2.1 [6] (referred to as AC '97). It is designed to be paired with a digital controller, typically located on the PCI bus or integrated within the system core logic chip set. The controller is responsible for all communications between the CS4201 and the remainder of the system. The CS4201 contains two distinct functional sections: digital and analog. The digital section includes the AC-link interface, S/PDIF interface, serial data port, GPIO, power management support, and Sample Rate Converters (SRCs). The analog section includes the analog input multiplexer (mux), stereo input mixer, stereo output mixer, mono output mixer, headphone amplifier, stereo Analog-to-Digital Converters (ADCs), stereo Digital-to-Analog Converters (DACs), and their associated volume controls [6].

3.4 AC-Link

„All communication with the CS4201 is established with a 5-wire digital interface to the controller called the AC-link. This interface is shown in Figure 5. All clocking for the serial communication is synchronous to the `BIT_CLK` *signal.* `BIT_CLK` *is generated by the primary audio codec and is used to clock the controller and any secondary audio codecs. Both input and output AC-link audio frames are organized as a sequence of 256 serial bits forming 13 groups referred to as 'slots'. During each audio frame, data is passed bi-directionally between the CS4201 and the controller. The input frame is driven from the CS4201 on the* `SDATA_IN` *line. The output frame is driven from the*

controller on the `SDATA_OUT` *line. The controller is also responsible for issuing reset commands via the* `RESET#` *signal."*

Literaly drawn from [6]

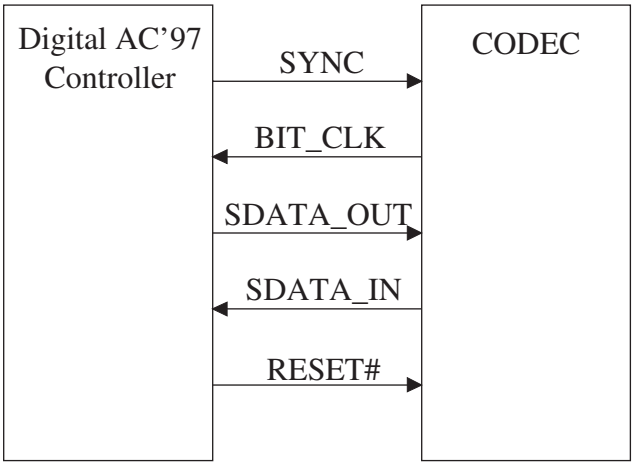

Figure 5: AC-link connections

The previous part describes the AC-link. Not all information are important for us but there is one thing we should be familiar with. The controller and codec communicate trough 5 wires with specified protocol. Data are sent in groups referred to as "slots". This communication takes a while so we have to take care about reading and writing the codec. Imagine this: The controller transmits data to codec and in the same moment we want to read some codec register. The read operation will possibly return wrong value, because it couldn't be performed. Therefore we have to watch the status of the AC-link and commit the read/write operation only if the AC-link bus is ready.

3.5 DMA Engine

To ensure the data exchange between device driver and device, DMA engine is used. ICH4 AC '97 controller provides six 16-bit DMA engines for audio

- PCM in channel
- PCM out channel
- MIC in channel
- MIC 2 in channel
- S/PIDF out channel

3.5.1 Buffer Descriptor List

The Buffer Descriptor list contains up to 32 entries. As shown in Figure 6 each entry contains a pointer to a data buffer, control bits, and the length of the buffer being pointed to. The length represents the number of samples. If we combine this with 16-bit sample size, we get the actual physical length of the buffer. The buffer length is restricted to 65536 samples. "0" in the buffer length indicates no samples to process [7].

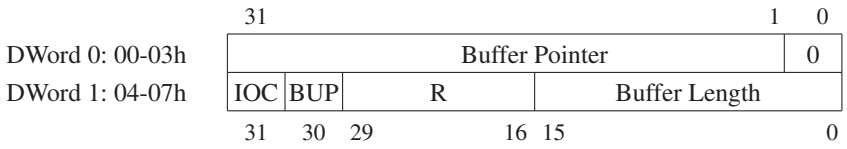

Figure 6: Buffer descriptor list entry.

The maximum length of the buffer descriptor list is is limited by the size of the index registers to 32. Figure 7 shows the organization of the Buffer Descriptor List.

In order to control the actually processed buffer, the count of buffers as well as the actually prefetched index there are three registers to this purpose:

Current Index Value Register (R/W) represent which buffer descriptor is currently being processed.

Last Valid Index Register (R/W) represents the last valid descriptor in the list.

Prefetched Index Value Register (RO) indicates which buffer descriptor in the list has been prefetched.

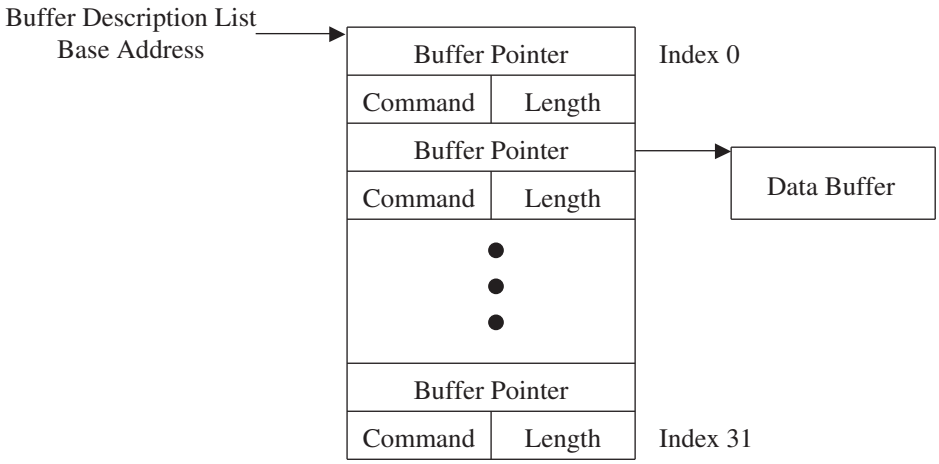

Figure 7: Buffer descriptor list.

3.5.2 DMA Initialization

To initialize a single DMA engine the following steps are described in [7, page 14].

1. Create the buffer descriptor list structure in memory (non-paged poll).
2. Write the Buffer Descriptor List Base Address register with the base address of the buffer descriptor list.
3. Set up the buffer descriptors and their corresponding buffers.
4. Once buffer descriptors have been set in memory, the software writes the Last Valid Index (LVI) register.
5. After the LVI registers have been updated, the software sets the run bit in the control register, in order to execute the descriptor list.

The same process should be repeated for each DMA engine.

4 Implementation

In implementation we have focused on the hardware dependent part and tried to full re-use the existing audio framework [5]. Driver is completely written in C language. To compile the source code we used the `cc` compiler which is a part of the ACK (Amsterdam Compiler Kit).

4.1 Audio framework

In this part we will take a closer to look at the driver design. In order to simplify the driver development process, the driver has been divided into two parts – hardware independent and hardware dependent.

Hardware independent part provides a framework which can be re-used in other drivers. In our case we used Laurens Bronwasser's (and Peter Boonstoppel's) audio framework as a base for our next implementation. We made only minor changes to his source code. The framework has several functions: it takes and processes messages from operating system and manages DMA buffers.

Hardware dependent part implements functionality to functions called by the audio framework. Figure 8 shows the architecture of our driver. System communicates with driver by passing messages to each other using `send()` and `receive()` primitives. The audio framework processes these messages. In case an operation on hardware is required, a call to `drv_*` function in hardware dependent part is made. In hardware dependent part of driver, we access the hardware directly using I/O operations.

4.2 Hardware dependent part

The implementation of hardware depend part is based on information we obtained from datasheets provided by the hardware manufacturer. To control the sound card, we had to master the access to controller as well as to codec (Section 3, Figure 3.). We can access both devices directly using `pci_in, pci_out` functions. However, the codec need special attention and we need some extra code to access these registers correctly. Codec is connected to controller by AC-Link. The communication between these devices takes some extra time, so we can not access codec's registers any time we want. Codec Access Semaphore register (CAS) has to be checked before every read, write operation to codec registers. Now, after we can access device registers, we can fully control this devices. Detailed description of all register and their functionality can be found in datasheets.

Typical sequence of step from initialization to playing sound can be briefly described be like this:

- Detect hardware and obtain base addresses

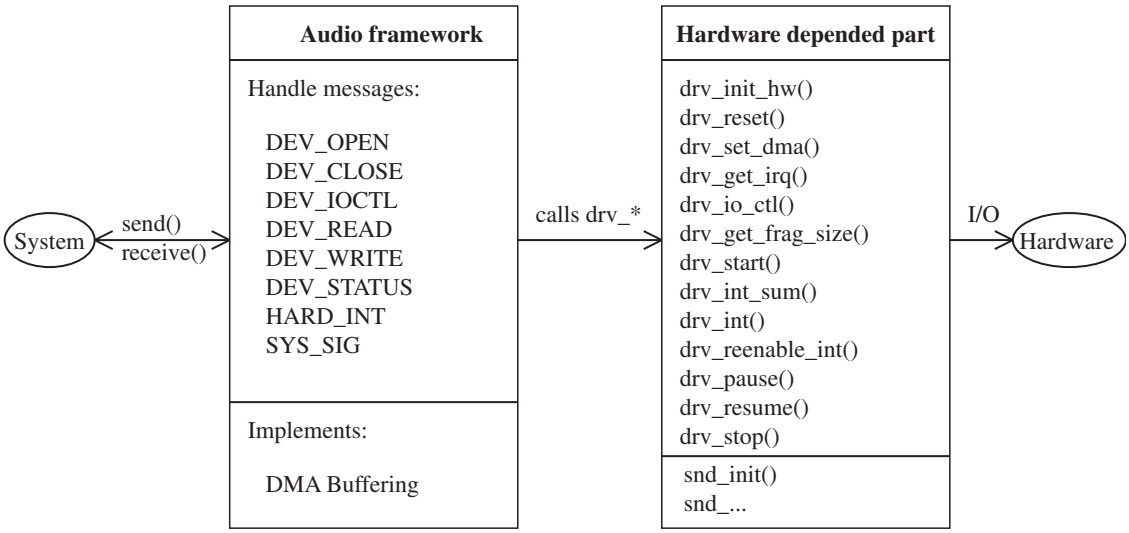

Figure 8: Two layer driver architecture.

- Initialize controller
- Power up codec
- Initialize codec (set volume mixer)
- Set pointers to DMA buffers
- Start bus master operation (start playing)

In the next sections, we will briefly describe some interesting parts of code. It is recomended to read this section together with source codes.

4.3 Device detection

The first thing we need to do is to detect our device, which is done by following function:

```
PRIVATE int detect_hw(void)
{
    u32_t r;
    int devind;
    u16_t v_id, d_id;

    pci_init();
    /* get first device and then search through the list */
    r = pci_first_dev(&devind, &v_id, &d_id);
    while( r > 0 )
    {
        if (v_id == VENDOR_ID && d_id == DEVICE_ID )
```

```
            break;

        r = pci_next_dev(&devind, &v_id, &d_id);
    }
    /* did we find anything? */
    if (v_id != VENDOR_ID || d_id != DEVICE_ID)
        return EIO;

    /* fill dev structure */
    dev.name     = pci_dev_name(v_id, d_id);
    dev.base     = pci_attr_r32(devind, PCI_BAR) & 0xfffffffe;
    dev.irq      = pci_attr_r8(devind, PCI_ILR);
    dev.revision = pci_attr_r8(devind, 0x08);
    dev.d_id     = d_id;
    dev.v_id     = v_id;
    dev.devind   = devind; /* pci device identifier */

    return OK;
}
```

The `detect_hw()` function iterates through all PCI devices and checks the Vendor ID and Device ID. After the device is found, the `dev` structure is filled. In our implementation we made this a little bit more complicated. In order to support multiple chipsets we check for multiple Vendor and Device ID's.

4.4 Accessing registers

As we already mentioned, the codec registers can be accessed directly, but before the CAS register has to be checked.

```
#define MICROS_TO_TICKS(m) (((m)*HZ/1000000)+1)

PRIVATE void micro_delay(unsigned long usec)
{
    tickdelay(MICROS_TO_TICKS(usec));
}

PRIVATE int snd_wait_semaphore()
{
    int count = 100;
    while (count--)
    {
        if ((pci_inb( bus_master_base + ICH_REG_ACC_SEMA ) & ICH_CAS) == 0)
                return OK;
        micro_delay(40);
    }
    return ERR;
}

PRIVATE int snd_read_codec_register(u16_t offs, u16_t *data)
{
    int status;
    /* Wait for semaphore */
    if (snd_wait_semaphore()!=OK)
            return ERR;

    /* Read the data */
    *data = pci_inw(mixer_base+offs);

    /* Check if read was sucessful */
    status = pci_inl(bus_master_base+ICH_REG_GLOB_STA);
```

```
    if (status & ICH_RCS)
    {
        /* clear timeout bit */
        pci_outl(bus_master_base + ICH_REG_GLOB_STA, status);
        return ERR;
    }

    return OK;
}

PRIVATE int snd_write_codec_register(u16_t offs, u16_t data)
{
    int status;
    /* Wait for semaphore */
    if (snd_wait_semaphore()!=OK)
            return ERR;

    /* Write the data */
    pci_outw(mixer_base+offs, data);

    return OK;
}
```

The function `snd_wait_semaphore()` waits until the `CAS` bit in Codec Access Semaphore Register is cleared. After this bit is checked, the function wait 40 microseconds. This delay is recommended by vendor.

4.5 AC'97 initialization

After the hardware is detected a sound chip and codec initialization is required. To access codec and controller registers, we need obtain a base address for these devices

```
/* Read Mixer and Bus Master base address
   from PCI configuration space  */
mixer_base = PCII_RREG32_(0,31,5,0x10) & 0xFFFFFFF8;

bus_master_base = PCII_RREG32_(0,31,5,0x14) & 0xFFFFFFF8;
```

Codec and chip initialization is following

```
/* InitAC7 */
PRIVATE int snd_init_codec()
{
    int status;
    /* Check if the AC link to primary codec is ready */
    dprint("Primary codec ready\n");
    status = snd_primary_codec_ready();
    if (status == OK)
    {
        /* Reset codec */
        dprint("Reset codec\n");
        snd_write_codec_register(AC97_RESET, 0x00);
        /* Power up */
        dprint("Power up codec\n");
        status = snd_power_up_codec();
    }
    return status;
}
/* Chip Initialization */
int snd_chip_init()
{
```

```
    int i;
    for (i=0; i<3; i++)
    {
        /* Disable Interrupts */
        pci_outb(bus_master_base + i*0x10 + 0xB,0); /* 0xB - CR */
        /* Reset Channels */
        pci_outb(bus_master_base + i*0x10 + 0xB,ICH_RESETREGS); /* 0xB - CR */
    }
}
```

We only initialize the primary codec. After the code is initialized we set som nice mixer volume levels.

4.6 DMA Initialization

Before the device can play some sound, DMA buffers needs to be initialized. To simplify access to these buffer, we introduced the following structure:

```
/* Buffer Descriptor Entry */
typedef struct
{
    u32_t   lpBuff;     /* Buffer Pointer */
    u16_t   wLength;    /* Buffer Length */
    u16_t   wPolicy;    /* Policy Bits */
} BDENTRY;
```

This structure represents exactly one entry in the buffer descriptor list (BD list) as described in [7, page 13].

In the following code we initialize the whole buffer descriptor list. However we use only two DMA buffers.

```
/*
 * This function arrange the Buffer Descriptio List. BDENTRY structure
 * represents one entry from BD list. BD list can take up to 32 entries.
 */
int drv_set_dma(u32_t dma, u32_t length, int chan)
{
    u32_t   bd_base = ich_dev[chan].base;
    BDENTRY *bdlist;

    int i;
    int frag_size = length >> 1;

    /* DEBUG */ dprint("drv_set_dma(buff:%X, size:%d, chan:%d) frag_size:%d\n",dma,
        length,chan,  frag_size);

    /* Pause DMA */
    drv_pause(chan);

    /* Reset Bus Master registers */
    reset_bm_regs(bd_base);

    /* Allocate BD List buffer */
    snd_alloc_bdlist(chan);

    /* Update BD List */
    bdlist = ich_dev[chan].lpList;
    for (i=0; i<32; i+=2)
    {
```

```
        bdlist[i+0].lpBuff    = dma;
        bdlist[i+0].wPolicy   = ICH_IOC_ENABLE | ICH_BUP_ENABLE;
        bdlist[i+0].wLength   = (frag_size >> 1);

        bdlist[i+1].lpBuff    = dma + frag_size;
        bdlist[i+1].wPolicy   = ICH_IOC_ENABLE | ICH_BUP_ENABLE;
        bdlist[i+1].wLength   = (frag_size >> 1);
    }

    /* Set DBBAR */
    pci_outl(bus_master_base + bd_base, (u32_t)ich_dev[chan].PhysAddr);
    /*pci_outb(bus_master_base + bd_base + 0x4 , 0); /* ? CIV - R/O  */

    /* Set Last Valid Index */
    pci_outb(bus_master_base + bd_base + ICH_X_LVI , 0);

    /* MIC: Enable */
    pci_outb(bus_master_base + ICH_REG_SDM, 8);

    /* clear interrupts */
    pci_outw(bus_master_base + bd_base + ICH_X_SR, ICH_FIFOE | ICH_BCIS | ICH_LVBCI);

    /*dump_regs();*/
}
```

DMA buffers are allocated by the audio framework. The allocation of buffer descriptor list is handled in the `snd_alloc_bdlist()` function. Pointer to BD list is called Buffer Descriptor Base Address and is 64 kB aligned.

4.7 Future work

As for the further development the volume mixer should be designed and completed. Only drawing of mixer control in "ALSA" style is implemented yet. The mp3 player should be modified and compiled for i586 or MMX to speed up decoding process. Mp3 player process takes ca. 70% of CPU time now. There is also one problem with recording. At the time I don't know where the problem is, but we found out that the interrupt is not received (or captured by the driver) after the bus master operation is started. We also implemented only the PCM out and Mic channel. This approach would possibly avoid some compatibility problem on different chipsets. However channels like PCM2, MIC2 and others should be easily implemented to existing driver.

5 Summary

Many ideas and development are standing behind Minix. The third edition is not just a toy, but should be taken seriously. Perhaps the microkernels are the future of the operating systems. Till then another research and development should be realized. For example the layer model could be refined or IPC mechanism reconsidered. At the current state of art Minix3 can be installed with advantage on embedded devices or specialized computers. Last but definitely not least is the fact that Minix3 provides a good documentation and therefore it is a good learning material for all students.

5.1 Comparison

In this section we will discuss some differences between microkernels and monolithic kernels. We will go through this only briefly because a deep analysis and comparison would offer enough material for whole book. When you are interested why is Linux monolithic and Minix not, then the famous *Linus vs. Tanenbaum Debate*[3] is a recommended reading. Since the operating systems are very complex a lot of decisions and compromises has to be done. There are many problems which have multiple solutions and each solution has his benefits and drawbacks.

As for microkernels the IPC can be a performance problem. As long as the processes are well insulated from each other we can have a perfectly secure and modular system. On the other hand the IPC needs some overhead, so when a lot of processes are running or messages are send wastefully the system will get slower. In this case we trade the security for performance.

In monolithic kernels, the IPC problem is not so serious because the data can be exchanged inside the kernel by simple calls. High cohesion of components, vulnerability and security issues are the price for good performance.

Hardware IPC would be possible solutions for the IPC problem. Future processor with IPC implementation could make these operations in no-time which would greatly enhance the performance of microkernel based systems.

As for monolithic kernels as well as for microkernels safe languages could be used to eliminate the impacts of bad code on the system. Accessing of wrong parts of memory or I/O ports by the application could be avoided already by the compiler.

5.2 Utilization

Minix3 is strongly recommended for academic environment. In combination with study materials [13] Minix3 offers an ideal platform for courses of operating systems. Several open source projects are available for running Minix3. Simulators like Bochs or Qemu allow us to run Minix3 without interference with existing system or hardware. Native

installation of Minix3 is suitable for desktop PC as well as for specialized devices. Network components like routers or embedded applications can be build on Minix3. The operating systems theory can be clearly demonstrated with this system. The possibility of applying theoretical knowledge in practise should be a great motivation for all students.

5.3 Device driver

During the process of writing device driver, we learned a lot about Minix3 and its architecture. The work on hardware dependent part was also very interesting, especially learning the way device works in generally and with DMA. The final driver works well. It was published on the internet to be shared with the Minix3 community.

6 Technical documentation

6.1 Installation and testing

The driver package `intel8x0.tar` includes:

- AC'97 Sound Driver for ICH4
- Simple mp3 player based on mp3lib
- Mixer (not completed yet)
- Playwave and recwave tools from Minix3

Installation steps:

- Unpack file `intel8x0.tar` to `/usr/src/drivers` directory.
- Change directory to `/usr/src/drivers/intel8x0` and compile the driver using `make` command.
- Similarly compile mp3player, playwave and recwave tools.
- Start the driver using `start.sh` script.

To test the functionality we can use the `playwave` program which can be found in `./IBM/` or our simple mp3 player located in `./mp3player` directory. You can play some music using `./mp3play <filename.mp3>` command.

Driver has been successfully tested with Minix version 3.2.1a on ICH4 chipset. It is possible that this driver will work well with other chipsets too – meaning ICH, ICH0, ICH3, ICH5, ICH6. The driver will automatically detect these chipsets, but the functionality hasn't been tested yet.

6.2 Debugging

Device driver consist of following files:

```
intel8x0.tar
├── ac97.h ..... AC97 registers definitions
├── audio_fw.h ..... Audio framework header file
├── audio_fw.c ..... Audio framework implementation
├── intel8x0.h ..... Hardware depended part header file
├── intel8x0.c ..... Hardware depended part implementation
└── ioc_sound.h ..... Sound ioctl() command codes
```

In case we want to modify or debug the driver we would need some extra output. We can enable the debug output as well as for the audio framework as for the hardware dependent part. To display all debug messages we use the `dprint` macro which is defined as following:

```
#define dprint printf
```

As we can see `dprint` is nothing else than a `printf` function. If we don't want to display debug messages, we can change the macro definition to:

```
#define dprint (void)
```

The `dprint` macro definition for audio framework can be found in `audio_fw.h`. As for the hardware dependent part the macro is defined in `intel8x0.c`.
Similarly to `dprint` macro there is an `error` macro. It is defined in `audio_fw.h` as following:

```
#define error printf
```

The main point of all these `dprint`, `error` macro definitions is to logically separate different kind of messages. The source code is also easier to maintain, because we only need to change single line to enable or disable several output messages.
If we experiment with driver, we sometimes need to display values of all device registers. Just for this purpose there is a function `dump_regs()` located in `intel8x0.c`, however by default it is commented out.

References

[1] Internet resource. `http://en.wikipedia.org/wiki/Peripheral_Component_Interconnect`.

[2] Internet resource. `http://www.linux-mips.org/wiki/PCI_Subsystem`.

[3] Internet resource. `http://www.oreilly.com/catalog/opensources/book/appa.html`.

[4] Minix3 homepage. `http://www.minix3.org`.

[5] Laurens Bronwasser. Audio driver in minix, 2005.

[6] Cyrus Logic. *CrystalClear® Audio Codec '97 with Headphone Amplifier CS4201*, February 2001. Prelimitary product information.

[7] Intel. *Intel® 82801AA (ICH) & Intel® 82801AB (ICH0) I/O Controller Hub AC '97*, December 1999.

[8] Intel. *Audio Codec'97 Component Specification*, April 2002. Revision 2.3.

[9] Intel. *Intel® 82801DB I/O Controller Hub 4 (ICH4) Datasheet*, May 2002.

[10] Jaroslav Kysela. Alsa driver for intel ich (i8x0) chipsets. Source code, 2000.

[11] Microsoft. Ac97 wdm sample driver. Source code, 1999.

[12] PCI-SIG. *PCI Local Bus Specification Revision 3.0*, February 3 2004.

[13] A.S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice Hall, 3rd edition, January 04 2006.

A CD Media Contents