FreeBSD and simple char

device driver for real PCI-hardware

The FreeBSD operating system captivates the hearts and minds of it's fans so much, that finds it's way in very diversive industries such as hosting projects and backbone routers. It can run on small embedded devices, as well as on large, multi-core systems.

What you will learn…

- How to program kernel character device driver
- How to use PCI POST card to visualize system's events

What you should know…

- PCI-subsystem of x86-compatible PC
- How to manipulate with sysinstall, make and gcc
- Basic knowledge of how FreeBSD kernel works

And sometimes, can be used in rather unusual state – for instance, to monitor several statuses of PC that is equipped with PCI POST card. We'll show today how it can be achieved.

Into the groove

System engineers usually begin acquainted with new hardware from a bottom line, in other words from low-level, firmware level. It applies either to non x86-architecture, as well as to x86, particularly to IBM PC-compatible machines. The latter ones are known to have a frimware naming like PC BIOS. Climbing from a bottom line, higher to a top, system engineer functionally and logically transforms to a high-level language programmer. Where architectural processes for complex systems take places. Unfortunaly, modern trends in IT shows very clearly the following: young professionals knows very well how to program nice UI, but completely unsure what is hidden under such terms like low-level programming and x86 BIOS calls. It's a pity, because, either low-level procedures, as well as BIOS calls are used every time to start a box with FreeBSD. Let's make a short trip into bootstrapping process for conventional x86-based PC.

First, when power cord is plugged and a power button is pressed, starts CPU. Almost immediately, RAM chips are initialized. Control vector is transferred to the following RAM address: `0xFFFF0`, or `0xFFFF.FFF0`. Both addresses are

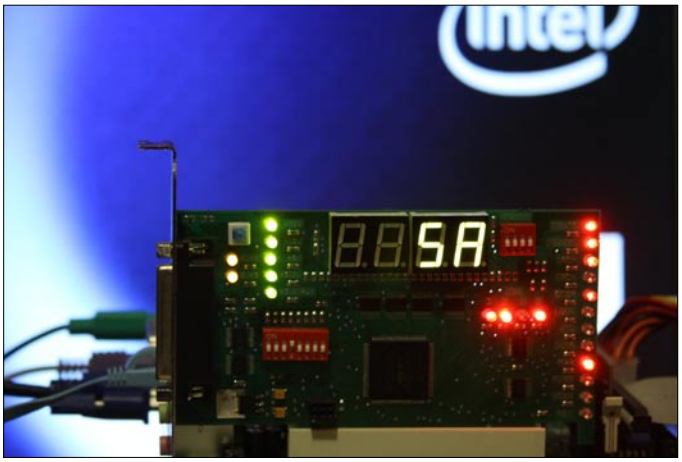

Figure 1. *IC80+ PCI POST card allows to visualize 0x80 / 0x81 ports contents*

```
Booting from Hard Disk...

F1  FreeBSD
F5  Drive 1

F6 PXE
Boot:  F1

BTX loader 1.00  BTX version is 1.02
Consoles: internal video/keyboard
BIOS drive C: is disk0
BIOS 636kB/523244kB available memory

FreeBSD/i386 bootstrap loader, Revision 1.1
(root@almeida.cse.buffalo.edu, Mon Jul 19 01:59:01 UTC 2010)
Loading /boot/defaults/loader.conf
```

Figure 2. *After BIOS initialization here comes BTX loader*

correct, because x86-compatible processor after first initialization must be in so-called *real-mode* state. At that time, memory address space from `0xE0000` to `0xFFFFF` is occupied by FlashROM. Where such procedures like initializations of various PC-components (RS232, LPT, USB) are stored.

In addition to the mentioned physical ports, each PC-system holds a virtual diagnostic port 0x80. PC BIOS (for example, AwardBIOS or PhoenixBIOS) sends into this port specifical values upon start of each init-procedure, so it can be just immediately been scanned by diagnostic equipment (we talk about it later). And finally, PC BIOS tunes then interrupt vectors (0x13, 0x18, 0x19) to have an ability to bootstrap an operating system – from a floppy disk, USB device, or ATA/SATA/SCSI device, in particular the interrupt vectors int

All these procedures are written in assembly-language and therefore it is assumed that resulting binary code is optimal in terms of execution time by CPU.

If all subsystems are functioning properly, PC BIOS will set up int 0x19 vector, so that BTX Loader could start. Which in turn would load whole FreeBSD system. All these things, we've discussed so far, are stored inside firmware (aka PC BIOS for x86 architecture). And if you woke up in the morning, and see your FreeBSD box doesn't want to boot – it could be a hardware error. Unidentified visually by PC BIOS, because it can be burned capacitor between ISA- and PCI-bus, or malfunctioning USB-port, which fails to initialize properly. Here comes diagnostic equipment, like POST (*Power-On Self Test*) card. It's mission is to show you on seven-segment LED display what's wrong. All you need is to have a list of POST-codes by your side, in order to figure out, what's exactly component failed.

IC80 POST-Card from IC Book Labs

I've searched thoroughfully marketplace to find out, what is difference exist between all these POST-cards. There is about a dozen different manufacturers – from eminent grandees, to noname cards, made somewhere in Guangdong province, PRC. Some developers provide a detailed desciption, manuals, and have a tech-support line, others only have look-alike printed circuit board. Since, I pefer to deal with a product, that offers a maximum number of features, I've chosen a POST-card from IC Book Labs [1] company. As it turned out, I made a right decision – more than 10 years of development, support for AMIBIOS, AwardBIOS, PhoenixBIOS, InsydeBIOS. Dual diagnostic LED-display that shows contents of virtual ports 0x80 / 0x81 / 0x84 / 0x1080 / 0x2080. There's also a switch, that allows to go through all POST step-by-step. A really swiss knife.

Though it's price differs from nonamed product by a factor of 10, it provides more features by the same factor. Anyway, let's get started.

Listing 1. *pciconf shows attached devices to system*

```
[root@a-bsd ~]# pciconf -l
hostb0@pci0:0:0:0:      class=0x060000 card=0x464c8086 chip=0x27708086 rev=0x02 hdr=0x00
vgapci0@pci0:0:2:0:     class=0x030000 card=0x464c8086 chip=0x27728086 rev=0x02 hdr=0x00
none0@pci0:0:27:0:      class=0x040300 card=0xd6048086 chip=0x27d88086 rev=0x01 hdr=0x00
pcib1@pci0:0:28:0:      class=0x060400 card=0x00000000 chip=0x27d08086 rev=0x01 hdr=0x01
pcib2@pci0:0:28:2:      class=0x060400 card=0x00000000 chip=0x27d48086 rev=0x01 hdr=0x01
pcib3@pci0:0:28:3:      class=0x060400 card=0x00000000 chip=0x27d68086 rev=0x01 hdr=0x01
uhci0@pci0:0:29:0:      class=0x0c0300 card=0x464c8086 chip=0x27c88086 rev=0x01 hdr=0x00
uhci1@pci0:0:29:1:      class=0x0c0300 card=0x464c8086 chip=0x27c98086 rev=0x01 hdr=0x00
uhci2@pci0:0:29:2:      class=0x0c0300 card=0x464c8086 chip=0x27ca8086 rev=0x01 hdr=0x00
uhci3@pci0:0:29:3:      class=0x0c0300 card=0x464c8086 chip=0x27cb8086 rev=0x01 hdr=0x00
ehci0@pci0:0:29:7:      class=0x0c0320 card=0x464c8086 chip=0x27cc8086 rev=0x01 hdr=0x00
pcib4@pci0:0:30:0:      class=0x060401 card=0x464c8086 chip=0x244e8086 rev=0xe1 hdr=0x01
isab0@pci0:0:31:0:      class=0x060100 card=0x464c8086 chip=0x27b88086 rev=0x01 hdr=0x00
atapci0@pci0:0:31:1:    class=0x01018a card=0x464c8086 chip=0x27df8086 rev=0x01 hdr=0x00
atapci1@pci0:0:31:2:    class=0x01018f card=0x464c8086 chip=0x27c08086 rev=0x01 hdr=0x00
ichsmb0@pci0:0:31:3:    class=0x0c0500 card=0x464c8086 chip=0x27da8086 rev=0x01 hdr=0x00
re0@pci0:1:0:0: class=0x020000 card=0x00018086 chip=0x816810ec rev=0x02 hdr=0x00
none1@pci0:4:0:0:       class=0x118000 card=0x00000000 chip=0x001cb00c rev=0x05 hdr=0x00
```

Boot sequence

I've inserted this IC80 POST card into free PCI-slot and pushed a power button. My mainboard was produced by Intel (D945GCLF2), so I referred to POST-codes that this manufacturer reserved for it's products [2]. Complete startup sequence for all initialization procedures was as follows:

```
22, 23, 25, 28, 34, 12, 58, 50, 51, EB, 58, 92, 90, 94,
                    95, BB, B8,
BA, 5A, 92, 90, 94, BB, BA, EB, BB, BA, 5A, BB, BA, E7,
                    E9, and finally 00
```

- 22 – Reading SPD from memory DIMMs
- 23 – Detecting presence of memory DIMMs
- 25 – Configuring memory
- 28 – Testing memory
- 34 – Loading recovery capsule
- 12 – Starting Application processor initialization
- 58 – Resetting USB bus
- 50 – Enumerating PCI busses
- 51 – Allocating resources to PCI bus
- EB – Calling Legacy Option ROMs
- 58 – Resetting USB bus
- 92 – Detecting presence of keyboard
- 90 – Resetting keyboard
- 94 – Clearing keyboard input buffer
- 95 – Instructing keyboard controller to run Self Test (PS2 only)
- BB – reserved by Intel
- B8 – Resetting removable media
- BA – Detecting presence of a removable media (IDE, CD-ROM detection, etc.)
- 5A – Resetting PATA/SATA bus and all devices
- ...
- E7- Waiting for user input
- E9 – Entering BIOS setup
- 5A – Resetting PATA/SATA bus and all devices
- BA – Detecting presence of a removable media (IDE, CD-ROM detection, etc.)
- 00 – Ready to boot

After last event (ID: 00) boot control is passed to BTX Loader and the usual bootstrapping for FreeBSD begins (see Figure 2).

FreeBSD system is up

After all necessary services enlisted in rc.conf are completed, we can log into FreeBSD and see, how the POST device looks like (in terms of the operating system of course. See Listing 1).

We see all integrated devices, including video- and network-card. And there are 2 strange devices for which there is no loaded driver: `none0@pci0:0:27:0` and `none1@pci0:`

Listing 2. *More advanced information for devices with 'unloaded' driver*

```
[root@a-bsd ~]# pciconf -lv | grep none -A3
none0@pci0:0:27:0:      class=0x040300 card=0xd6048086 chip=0x27d88086 rev=0x01 hdr=0x00
    vendor     = 'Intel Corporation'
    device     = 'IDT High Definition Audio Driver  (BA101897)'
    class      = multimedia
--
none1@pci0:4:0:0:       class=0x118000 card=0x00000000 chip=0x001cb00c rev=0x05 hdr=0x00
    vendor     = 'IC Book Labs'
    device     = 'IC80+PCI POST Diagnostics Card'
    class      = dasp
```

Listing 3. *FreeBSD contains no driver for IC80+PCI POST Diagnostics Card*

```
[root@a-bsd ~]# dmesg | grep pci4
pci4: <ACPI PCI bus> on pcib4
pci4: <dasp> at device 0.0 (no driver attached)
```

Listing 4. *Makefile for sample kernel driver 'hello_world'*

```
KMOD=   hello_world
SRCS=   hello_world.c

.include <bsd.kmod.mk>
```

Listing 5. *Source file for sample kernel driver 'hello_world'*

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/module.h>
#include <sys/sysproto.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
/*
 * The function for implementing the syscall.
 */
static int
hello (struct thread *td, void *arg)
{
        printf ("hello kernel world\n");
        return 0;
}
/*
 * The 'sysent' for the new syscall
 */
static struct sysent hello_sysent = {
        0,                      /* sy_narg */
        hello                   /* sy_call */
};
/*
 * The offset in sysent where the syscall is allocated.
 */
static int offset = NO_SYSCALL;
/*
 * The function called at load/unload.
 */
static int
load (struct module *module, int cmd, void *arg)
{
        int error = 0;
        switch (cmd) {
        case MOD_LOAD :
                printf ("Driver loaded at %d\n", offset); /* logging to syslog */
                uprintf ("Driver loaded at %d\n", offset);  /* logging to terminal */
                break;
        case MOD_UNLOAD :
                printf ("Driver unloaded from %d\n", offset); /* logging to syslog */
                uprintf ("Driver unloaded from %d\n", offset); /* logging to terminal */
                break;
        default :
                error = EOPNOTSUPP;
        }
        return error;
}
SYSCALL_MODULE(hello_world, &offset, &hello_sysent, load, NULL);
```

Listing 6a. *Source file for kernel driver 'ic80'*

```
#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>  /* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  /* defines used in kernel.h */
#include <sys/kernel.h> /* types used in module
                   initialization */
#include <sys/conf.h>   /* cdevsw struct */
#include <sys/uio.h>    /* uio struct */
#include <sys/malloc.h>
#include <sys/types.h>

#define BUFFERSIZE 5

/* Function prototypes */
static d_open_t      ic80_open;
static d_close_t     ic80_close;
static d_read_t      ic80_read;
static d_write_t     ic80_write;

/* Character device entry points */
static struct cdevsw ic80_cdevsw = {
    .d_version = D_VERSION,
    .d_flags = D_PSEUDO | D_NEEDGIANT,
    .d_open = ic80_open,
    .d_close = ic80_close,
    .d_read = ic80_read,
    .d_write = ic80_write,
    .d_name = "ic80",
};

typedef struct s_ic80 {
    char msg[BUFFERSIZE];
    int len;
} t_ic80;

static struct cdev *ic80_dev;
static int count;
static t_ic80 *ic80msg;

MALLOC_DECLARE(M_IC80BUFFER);
MALLOC_DEFINE(M_IC80BUFFER, "ic80buffer", "a buffer
                   for ic80 post card kernel module");

static int

{
    int err = 0;

    switch (what) {
    case MOD_LOAD:
        ic80_dev = make_dev(&ic80_cdevsw, 0, UID_ROOT,
                    GID_WHEEL, 0666, "ic80");
        ic80msg = malloc(sizeof(t_ic80), M_IC80BUFFER,
                    M_WAITOK);
        printf("Driver IC80 loaded\n");
        break;
    case MOD_UNLOAD:
        destroy_dev(ic80_dev);
        free(ic80msg, M_IC80BUFFER);
        printf("Driver IC80 unloaded\n");
        break;
    default:
        err = EOPNOTSUPP;
        break;
    }
    return(err);
}

static int
ic80_open(struct cdev *dev, int oflags, int devtype,
                    struct thread *p)
{
    int err = 0;
    printf("Opening device \"ic80\" ...\n");
    return(err);
}

static int
ic80_close(struct cdev *dev, int fflag, int devtype,
                    struct thread *p)
{
    printf("Closing device \"ic80\" ... \n");
    return(0);
}

/*
 * The read function just takes the buf that was saved
                    via
 * echo_write() and returns it to userland for
                    accessing.
 * uio(9)
 */
```

`4:0:0`. Let's start pciconf in more verbose mode: see Listing 2.

Excellent! The first device for which there is no driver has been loaded – an integrated Intel HD audio device. What about the second one? This is our POST-card. Make sure once again, that no single driver was loaded for it during bootstrap process (see Listing 3).

Actually, the functionality of this card is quite simple – to display on LED0 and LED1 display what has been sent to diagnostic ports 0x80 and 0x81. And I'm quite sure, we're able to write necessary software for it. Well, we're moving to the next section!

Designing simple char-device driver

As an example let's first analyze how to program a very simple kernel driver. It's mission is to write to syslog and terminal a message *Driver loaded*, once it's registered by system. And once the kernel module is unregistered by system, it should write *Driver unloaded*. This will be a truly *Hello, world!*, but with kernel background.

But first, please make sure that the following packages are installed on the system: gcc, make, kernel sources, and share sources. Usually, the first 2 packages are already present in system. You only need to install the latter two. Run sysinstall and install the following packages:

```
sysinstall->Configure->Distributions->src->share
sysinstall->Configure->Distributions->src->sys
```

The good starting point for writing a kernel driver from scratch is to look at: `/usr/src/share/examples/kld/syscall/module/syscall.c`.

And, it would be also worth to look at this page [3] – with explanation about module structure (see Listing 4 and Listing 5).

Start compilation process:

```
# make
```

And now we load module into kernel address space:

Listing 6b. *Source file for kernel driver 'ic80'*

```
static int
ic80_read(struct cdev *dev, struct uio *uio, int
                    ioflag)
{
    int err = 0;
    int amt;

    /*
     * How big is this read operation?  Either as big
                    as the user wants,
     * or as big as the remaining data

    amt = MIN(uio->uio_resid, (ic80msg->len - uio-
                    >uio_offset > 0) ?
         ic80msg->len - uio->uio_offset : 0);
    if ((err = uiomove(ic80msg->msg + uio->uio_offset,
                    amt, uio)) != 0) {
        uprintf("uiomove failed!\n");
    }
    return(err);
}

static int
ic80_write(struct cdev *dev, struct uio *uio, int
                    ioflag)
{
    int err = 0;

    /* Copy the string in from user memory to kernel
                    memory */
    err = copyin(uio->uio_iov->iov_base, ic80msg->msg,
                    4);

    /* Now we need to null terminate, then record the
                    length */
    *(ic80msg->msg + 4) = 0;
    ic80msg->len = 4;

    if (err != 0) { uprintf("Write failed: bad
                    address!\n"); }
    count++;

    outb( 0x80, strtol(ic80msg->msg, 0, 16) );

    return(err);
}

DEV_MODULE(ic80,ic80_loader,NULL);
```

```
# kldload ./hello_world.ko
```

Is it really there?

```
# dmesg | grep Driver
hello_world loaded at 210
```

Okay, the driver functions correctly. You can unload it from memory like that:

```
# kldunload hello_world
```

We got now a very simple driver. Our next step is to add more functionality, i.e. a driver must be able to create character device under `/dev/` tree, and it also must be able to visualize any value of hexadecimal type on the LED0-display of the POST-card.

The latter procedure is done by sending hex value to diagnostic port 0x80. Yes, that's easy (see Listing 6).

One short comment about this source. We create a structure named `ic80_cdevsw`, where we place the functions, that will be called upon access a character device `/dev/ic_80`. For example, upon every close, open, write or read operation.

```
static struct cdevsw ic80_cdevsw = {
 .d_version = D_VERSION,
 .d_flags = D_PSEUDO | D_NEEDGIANT,
 .d_open = ic80_open,
 .d_close = ic80_close,
 .d_read = ic80_read,
 .d_write = ic80_write,
 .d_name = „ic80”,
};
```

Inside `ic80_loader()` function we program the mechanism, how the module should be registered, and unregistered by operating system. Within `ic80_write()` function the value for diagnostic port 0x80 is passed from user-space land to kernel and send directly to port 0x80. Internal buffer `ic80msg` shouldn't be very long, because it should store 5 characters only.

Listing 7. *System load value is pushed to new char device*

```
#!/bin/sh
        while true;
   do
      sleep 1s;
      id='top -d 1 | grep load | awk '{ print $6 }' |
                  head -c 4 | tail -c 2';
      echo $id > /dev/ic80;
   done
```

Driver is ready. Now, when you load it into the kernel address space it will register a new character device `/dev/ic80`, with permissions 0666 and owned by root: wheel. After that you can send any value in `0xXX` format (for example, 0xAB) into `/dev/ic80` and it will be displayed on LED0-screen. I've decided to control system average load, and prepared the following script. Once it is started by user, and once your CPU is heavy loaded, the immediate change of LED0-indicator is guaranteed. My tough and heavy CPU loader job was performed with help of `burnMMX` utility (from `/usr/ports/sysutils/burn` package).

Actually, you can display on LED0-indicator whatever information you need. Values should be in range between 0x00 and 0xFF. For instance, it could be a temperature, either of CPU, or mainboard. Moreover, don't forget about LED1-indicator – you can enhance the kernel driver in order to show twice as more information.

On the 'Net

- *http://en.icbook.com.ua/ [1]*
- *http://www.intel.com/support/motherboards/desktop/sb/CS-025434.htm [2]*
- *http://www.redantigua.com/c-ex-kernel-freebsd-hello.html [3]*

Conclusion

As you can see, designing a character driver is not as difficult as it seems. It can be fun sometimes, especially if you have very unusual hardware and FreeBSD by your side. And you also have a desire to combine them both into one amazing solution. Anyway, I'm sure that every novice FreeBSD user has a potential to create something possible from impossible pieces. Just like any FreeBSD guru, isn't it?

ANTON BORISOV

The very first Anton's experience with UNIX was FreeBSD. It was TWM, wget and Netscape Communicator. Many things have changed greatly since then, but a true simplicity remained unchanged – The Power to Serve. That's why Anton chooses FreeBSD for 'impossible' missions.