# Программирование драйверов устройств

Niek Linnenbank
02.12.2009

## Введение

Это руководство облегчит вам старт в программировании драйверов устройств для MINIX на языке C. Драйвера устройств – это компьютерные программы, которые взаимодействуют с реальными электронными компонентами. Например, компьютер, который вы используете сейчас, читая этот текст, имеет драйвер дисплея. Другим примером является драйвер диска, который читает и пишет не форматированные данные с(на) диск вашего компьютера. Одним словом, компьютерная система не будет пригодна для использования, не имея хороших драйверов.

## Предварительные условия

Разработка драйверов устройств требует навыков программирования и знаний целевых устройств. В MINIX драйверы программируются на языке C. В этом руководстве мы предполагаем, что вы знаете на базовом уровне программирование на C, и что вы имеете для использования работающую MINIX систему, версии не ниже 3.1.5.

## Первые шаги

Следующие секции содержат шаг за шагом инструкции по программированию простых драйверов устройств.

**Пример 1:** Hello, World

Давайте попытаемся создать для начала самый простой драйвер из всех, какие могут быть - Hello World! драйвер устройства. Он имеет только одно назначение: выводит сообщение приветствия на старте, после чего завершается. Прежде создадим каталог в MINIX дереве исходных кодов, для размещения туда кода нашего драйвера. Как пользователь `root` выполните:

```
# cd /usr/src/drivers
# mkdir hello
# cd hello
```

Для сборки драйвера устройства нам понадобится `Makefile`. Вы можете использовать следующий код как отправную точку для вашего `Makefile` при сборке своего драйвера:

***`Makefile:`***

```
#
# Makefile for the hello driver.
#
DRIVER = hello

#
# Directories.
#
u = /usr
i = $u/include
```

```
s = $i/sys
m = $i/minix
b = $i/ibm
d = ..

#
# Build Programs, Flags and Variables.
#
CC      = exec cc
CFLAGS  = -I$i $(CPROFILE)
LDFLAGS = -i -L../libdriver
LIBS    = -ldriver -lsys
OBJ     = hello.o

# build local binary
all build:      $(DRIVER)
$(DRIVER):      $(OBJ)
        $(CC) -o $@ $(LDFLAGS) $(OBJ) $(LIBS)
        install -S 128k $(DRIVER)

# install with other drivers
install:        /usr/sbin/$(DRIVER)
/usr/sbin/$(DRIVER):    $(DRIVER)
        install -o root -cs $? $@

# clean up local files
clean:
        rm -f *.o $(DRIVER)

depend:
        mkdep "$(CC) -E $(CPPFLAGS)" *.c > .depend

# Include generated dependencies.
include .depend
```

После чего создаём файл исходного кода C с программой Hello World! драйвера:

### *hello.c:*

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    printf("Hello, World!\n");
    return EXIT_SUCCESS;
}
```

Теперь мы попытаемся сделать всю сборку корректно:

```
# touch .depend
# make clean
# make
# make install
```

Если вы на этом шаге не получили никаких сообщений об ошибках, попытайтесь выполнить драйвер устройства. Первое, что мы должны при этом сделать, это подредактировать файл `/etc/drivers.conf`. Он содержит список всех драйверов устройств, и их права. Каждый драйвер типично нуждается в доступе только к одному аппаратному устройству, и использует ограниченное число функций, предоставляемых MINIX. Чтобы дать нашему простому пробному драйверу достаточно прав для эксперимента, добавим следующее к

файлу `/etc/drivers.conf`:

***/etc/drivers.conf:***

```
driver hello
{
        system
                UMAP            # 14
                IRQCTL          # 19
                DEVIO           # 21
                SETALARM        # 24
                TIMES           # 25
                GETINFO         # 26
                SAFECOPYFROM    # 31
                SAFECOPYTO      # 32
                SETGRANT        # 34
                PROFBUF         # 38
                SYSCTL
        ;
        ipc
                SYSTEM PM RS LOG TTY DS VM VFS
                pci inet amddev
                ;
    uid 0;
};
```

Старт, остановка и рестарт драйверов устройств должно делаться командой `service(8)`. Для старта нашей программы команда будет иметь следующий вид:

```
# service up /usr/sbin/hello
Hello, World!
RS: restarting /usr/sbin/hello, restarts 0
```

И вот нам сюрприз, оно выводит: Hello, World! Останавливаем драйвер командой:

```
# service down hello
```

Мы получили (при старте) ещё одно сообщение от того, что называется сервером реинкарнаций. В MINIX, как микроядерной системе, драйверы устройств являются отдельными программами, которые посылают и принимают сообщения для взаимодействия с другими компонентами операционной системы. Драйверы устройств, подобно любым другим программам, могут содержать ошибки, и могли бы аварийно завершаться в любой момент времени. Сервер реинкарнаций будет предпринимать попытки рестартовать драйвера устройств, когда они уведомляют о том, что они аварийно остановлены ядром в результате краха, или как в нашем случае, когда они выполняют `exit(2)` неожиданно. Вы можете видеть сервер реинкарнаций в списке процессов как `rs`, когда вы выполняете `ps` команду. Сервер реинкарнаций посылает пульс каждому выполняющемуся драйверу устройства в системе периодически, чтобы проверить что они ещё в состоянии отвечать, и не вошли, другими словами, в бесконечный цикл.

Итак, как мы можем противодействовать этому? Мы позволим нашему драйверу отвечать на пульсы, что позволит RS корректно детектировать, что драйвер всё ещё выполняется. К счастью, как мы увидим это в следующем примере, существует библиотека `libdriver` в MINIX; `libdriver` может осуществлять коммуникации с RS и другими компонентами операционной системы прозрачно, и ещё многое что другое.

**Пример 2**: `/dev/hello`

В этом примере мы расширим функциональность драйвера, и переделаем его реализацию, используя библиотеку `libdriver`. Вместо простого вывода приветствия на старте, мы теперь хотим использовать файл устройства `/dev/hello` для чтения текста Hello World! Мы станем использовать старший ID устройства равный 17 в этом примере. Вы можете всё так же использовать `Makefile` из примера 1. Начнём с того, что создадим `hello.h` заголовочный файл:

### *hello.h:*

```
#ifndef __HELLO_H
#define __HELLO_H

/** The Hello, World! message. */
#define HELLO_MESSAGE "Hello, World!\n"

/** Major ID of /dev/hello. */
#define HELLO_MAJOR 17

#endif /* __HELLO_H */
```

Как вы можете видеть, заголовочный файл содержит препроцессорные макросы для сообщений, и в нём определяется старший ID=17 для устройства `/dev/hello` . Теперь мы готовы наполнить программным кодом файл `hello.c` :

### *hello.c:*

```
#include "../drivers.h"
#include "../libdriver/driver.h"
#include <stdio.h>
#include <stdlib.h>
#include <minix/ds.h>
#include "hello.h"

/*
 * Function prototypes for the hello driver.
 */
_PROTOTYPE( PRIVATE char * hello_name,   (void) );
_PROTOTYPE( PRIVATE int hello_open,      (struct driver *d, message *m) );
_PROTOTYPE( PRIVATE int hello_close,     (struct driver *d, message *m) );
_PROTOTYPE( PRIVATE struct device * hello_prepare, (int device) );
_PROTOTYPE( PRIVATE int hello_transfer,  (int procnr, int opcode,
                                          u64_t position, iovec_t *iov,
                                          unsigned nr_req, int safe)
);
_PROTOTYPE( PRIVATE void hello_geometry, (struct partition *entry) );

/* Entry points to the hello driver. */
PRIVATE struct driver hello_tab =
{
    hello_name,
    hello_open,
    hello_close,
    nop_ioctl,
    hello_prepare,
    hello_transfer,
    nop_cleanup,
    hello_geometry,
    nop_signal,
    nop_alarm,
    nop_cancel,
```

```
    nop_select,
    nop_ioctl,
    do_nop,
};

/** Represents the /dev/hello device. */
PRIVATE struct device hello_device;

PRIVATE char * hello_name(void)
{
    printf("hello_name()\n");
    return "hello";
}

PRIVATE int hello_open(d, m)
    struct driver *d;
    message *m;
{
    printf("hello_open()\n");
    return OK;
}

PRIVATE int hello_close(d, m)
    struct driver *d;
    message *m;
{
    printf("hello_close()\n");
    return OK;
}

PRIVATE struct device * hello_prepare(dev)
    int dev;
{
    hello_device.dv_base.lo = 0;
    hello_device.dv_base.hi = 0;
    hello_device.dv_size.lo = strlen(HELLO_MESSAGE);
    hello_device.dv_size.hi = 0;
    return &hello_device;
}

PRIVATE int hello_transfer(proc_nr, opcode, position, iov, nr_req, safe)
    int proc_nr;
    int opcode;
    u64_t position;
    iovec_t *iov;
    unsigned nr_req;
    int safe;
{
    int bytes, ret;

    printf("hello_transfer()\n");

    bytes = strlen(HELLO_MESSAGE) - position.lo < iov->iov_size ?
            strlen(HELLO_MESSAGE) - position.lo : iov->iov_size;

    if (bytes <= 0)
    {
        return OK;
    }
    switch (opcode)
    {
        case DEV_GATHER_S:
            ret = sys_safecopyto(proc_nr, iov->iov_addr, 0,
                                 (vir_bytes) (HELLO_MESSAGE + position.lo),
```

```
                                bytes, D);
            iov->iov_size -= bytes;
            break;

        default:
            return EINVAL;
    }
    return ret;
}

PRIVATE void hello_geometry(entry)
    struct partition *entry;
{
    printf("hello_geometry()\n");
    entry->cylinders = 0;
    entry->heads     = 0;
    entry->sectors   = 0;
}

PUBLIC int main(int argc, char **argv)
{
    u32_t this_proc;

    printf("%s", HELLO_MESSAGE);

    /* Lookup our task number. */
    if (ds_retrieve_u32("hello", &this_proc) != OK)
    {
        printf("%s: ds_retrieve_u32() failed: %s\n",
                argv[0], strerror(errno));
        return EXIT_FAILURE;
    }
    /* Map major number to our process. */
    if (mapdriver(this_proc, HELLO_MAJOR, STYLE_DEV, TRUE) != OK)
    {
        printf("%s: mapdriver() failed: %s\n",
                argv[0], strerror(errno));
        return EXIT_FAILURE;
    }
    /*
     * Run the main loop.
     */
    driver_task(&hello_tab);
    return OK;
}
```

Давайте попытаемся оценить что делает показанный выше код. Прежде всего, он включает несколько `#include` директив, для требуемых прототипов и функций, используемых программой. Далее декларируется структура `driver`. Эта структура заполняется функциями обратного вызова, которые будут активироваться библиотекой `libdriver` во время исполнения. Здесь определены функции обратного вызова для осуществления операций `open`, `read`, `write` и `close` драйвером устройства. Большинство реальных действий осуществляется в `hello_transfer` функции. Она использует `sys_safecopyto` функцию для копирования `HELLO_MESSAGE` текстовой строки из пространства программы драйвера устройства, в пространство программы пользователя, читающей из устройства `/dev/hello`. Операционная система затем, по требованию драйвера устройства, осуществляет корректное копирование байт между двумя программами. Вектор ввода/вывода `iov` использован функцией `sys_safecopyto` для описания адреса области памяти для загрузки байт, и числа байт, запрошенных пользовательским приложением.

В теле функции `main()` присутствуют вызовы двух специальных функций:

`ds_retrieve_u32()` и `mapdriver()`. Функция `ds_retrieve_u32()` использована для запроса к серверу хранения данных (Data Store Server, DS) номера `endpoint` для нашего драйвера устройства. Номер endpoint используется MINIX для отправки и получения сообщений между процессами. Далее, функция `mapdriver()` используется для отображения старшего ID=17 в `endpoint` драйвера нашего устройства. Это обеспечивает, что любой `read/write` запрос к файлу устройства в `/dev/` с старшим ID 17 будут переадресовываться к процессу нашего драйвера устройства.

В завершение, функция `main()` выполняет `driver_task()` чтобы указать `libdriver` начать обслуживание пользовательских запросов и диспетчирование функций обратного вызова.

Теперь компилируем программу снова, используя те же команды, что и в примере 1, и стартуем драйвер командой `service(8)` :

```
# service up /usr/sbin/hello
Hello, World!
```

Замечательно! Нет более рестартов по инициативе сервера реинкарнаций. Теперь попытаемся увидеть, сможем ли мы считать текст «Hello World!» с файла устройства. Для начала создадим файл символьного устройства `/dev/hello` со старшим ID=17 и младшим ID=0:

```
# mknod /dev/hello c 17 0
```

А затем просто используем команду `cat(1)` для чтения из этого устройства:

```
# cat /dev/hello
hello_open()
hello_transfer()
hello_transfer()
hello_close()
Hello, World!
```

Если вы увидели вывод, подобный тому, что показан выше - ваш драйвер работает!

## Драйвера устройств из реального мира

Пример 3: CMOS  системные часы.

А теперь давайте попытаемся написать драйвер для реальной аппаратной составляющей нашего компьютера, например подобной RTC часам. Это устройство поддерживает системное время, и обладает весьма простым I/O интерфейсом. Вспомните, из нашего введения, что написание драйверов это не только область программирования, но и, в большей мере, область понимания аппаратного обеспечения самого по себе, не так ли? А поэтому, прежде чем переходить к написанию драйвера, внимательно читаем вот эти документы относительно RTC:

http://wiki.osdev.org/CMOS
Чтение из и запись в CMOS.

http://ivs.cs.uni-magdeburg.de/~zbrog/asm/cmos.html
Карта распределения памяти CMOS.

http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC146818.pdf

Описание чипа MC146818 : REAL-TIME CLOCK PLUS RAM (RTC) - опционально, по желанию.

А теперь давайте подредактируем драйвер, написанный в примере 2, так, чтобы мы смогли использовать `/dev/time` для чтения текущего системного времени из RTC. Вы можете всё ещё использовать `Makefile` из предыдущих примеров, если желаете. Первоначально создаём файл с необходимыми определениями:

***time.h:***

```
#ifndef __TIME_H
#define __TIME_H

/* Major ID of /dev/time. */
#define TIME_MAJOR  17

/* Base I/O port of the CMOS. */
#define CMOS_PORT   0x70

/* Data field offsets of the RTC in CMOS. */
#define RTC_SECONDS      0
#define RTC_MINUTES      2
#define RTC_HOURS        4
#define RTC_DAY_OF_WEEK  6
#define RTC_DAY_OF_MONTH 7
#define RTC_MONTH        8
#define RTC_YEAR         9

/* RTC Registers and Flags. */
#define RTC_STATUS_A     10
#define RTC_STATUS_B     11
#define RTC_UIP          0x80
#define RTC_BCD          0x04

#endif /* __TIME_H */
```

А теперь впишем сам код в файл `/usr/src/drivers/time.c`:

***time.c:***

```
#include "../drivers.h"
#include "../libdriver/driver.h"
#include <stdio.h>
#include <stdlib.h>
#include <minix/ds.h>
#include "time.h"

/*
 * Function prototypes for the time driver.
 */
_PROTOTYPE( PRIVATE char * time_name,   (void) );
_PROTOTYPE( PRIVATE int time_open,      (struct driver *d, message *m) );
_PROTOTYPE( PRIVATE int time_close,     (struct driver *d, message *m) );
_PROTOTYPE( PRIVATE struct device * time_prepare, (int device) );
_PROTOTYPE( PRIVATE void time_from_cmos, (char *buffer, int size) );
_PROTOTYPE( PRIVATE int cmos_read_byte, (int offset) );
_PROTOTYPE( PRIVATE unsigned bcd_to_bin, (unsigned value) );
_PROTOTYPE( PRIVATE int time_transfer,  (int procnr, int opcode,
                                         u64_t position, iovec_t *iov,
                                         unsigned nr_req, int safe)
);
```

```
_PROTOTYPE( PRIVATE void time_geometry, (struct partition *entry) );

/* Entry points to the time driver. */
PRIVATE struct driver time_tab =
{
    time_name,
    time_open,
    time_close,
    nop_ioctl,
    time_prepare,
    time_transfer,
    nop_cleanup,
    time_geometry,
    nop_signal,
    nop_alarm,
    nop_cancel,
    nop_select,
    nop_ioctl,
    do_nop,
};

/** Represents the /dev/time device. */
PRIVATE struct device time_device;

PRIVATE char * time_name(void)
{
    printf("time_name()\n");
    return "time";
}

PRIVATE int time_open(d, m)
    struct driver *d;
    message *m;
{
    printf("time_open()\n");
    return OK;
}

PRIVATE int time_close(d, m)
    struct driver *d;
    message *m;
{
    printf("time_close()\n");
    return OK;
}

PRIVATE struct device * time_prepare(dev)
    int dev;
{
    time_device.dv_base.lo = 0;
    time_device.dv_base.hi = 0;
    time_device.dv_size.lo = 0;
    time_device.dv_size.hi = 0;
    return &time_device;
}

PRIVATE int cmos_read_byte(offset)
    int offset;
{
    unsigned long value = 0;
    int r;

    if ((r = sys_outb(CMOS_PORT, offset)) != OK)
    {
```

```
        panic("time", "sys_outb failed", r);
    }
    if ((r = sys_inb(CMOS_PORT + 1, &value)) != OK)
    {
        panic("time", "sys_inb failed", r);
    }
    return value;
}

PRIVATE unsigned bcd_to_bin(value)
    unsigned value;
{
    return (value & 0x0f) + ((value >> 4) * 10);
}

PRIVATE void time_from_cmos(buffer,size)
    char *buffer;
    int size;
{
    int sec, min, hour, day, mon, year;

    /*
     * Wait until the Update In Progress (UIP) flag is clear, meaning
     * that the RTC registers are in a stable state.
     */
    while (!(cmos_read_byte(RTC_STATUS_A) & RTC_UIP))
    {
        ;
    }
    /* Read out the time from RTC fields in the CMOS. */
    sec  = cmos_read_byte(RTC_SECONDS);
    min  = cmos_read_byte(RTC_MINUTES);
    hour = cmos_read_byte(RTC_HOURS);
    day  = cmos_read_byte(RTC_DAY_OF_MONTH);
    mon  = cmos_read_byte(RTC_MONTH);
    year = cmos_read_byte(RTC_YEAR);

    /* Convert from Binary Coded Decimal (BCD), if needed. */
    if (cmos_read_byte(RTC_STATUS_B) & RTC_BCD)
    {
        sec  = bcd_to_bin(sec);
        min  = bcd_to_bin(min);
        hour = bcd_to_bin(hour);
        day  = bcd_to_bin(day);
        mon  = bcd_to_bin(mon);
        year = bcd_to_bin(year);
    }
    /* Convert to a string. */
    snprintf(buffer, size, "%04d-%02x-%02x %02x:%02x:%02x\n",
             year + 2000, mon, day, hour, min, sec);
}

PRIVATE int time_transfer(proc_nr, opcode, position, iov, nr_req, safe)
    int proc_nr;
    int opcode;
    u64_t position;
    iovec_t *iov;
    unsigned nr_req;
    int safe;
{
    int bytes, ret;
    char buffer[1024];

    printf("time_transfer()\n");
```

```
    /* Retrieve system time from CMOS. */
    time_from_cmos(buffer, sizeof(buffer));

    bytes = strlen(buffer) - position.lo < iov->iov_size ?
            strlen(buffer) - position.lo : iov->iov_size;

    if (bytes <= 0)
    {
        return OK;
    }
    switch (opcode)
    {
        case DEV_GATHER_S:
            ret = sys_safecopyto(proc_nr, iov->iov_addr, 0,
                                 (vir_bytes) (buffer + position.lo),
                                  bytes, D);
            iov->iov_size -= bytes;
            break;

        default:
            return EINVAL;
    }
    return ret;
}

PRIVATE void time_geometry(entry)
    struct partition *entry;
{
    printf("time_geometry()\n");
    entry->cylinders = 0;
    entry->heads     = 0;
    entry->sectors   = 0;
}

PUBLIC int main(int argc, char **argv)
{
    u32_t this_proc;

    /* Lookup our task number. */
    if (ds_retrieve_u32("time", &this_proc) != OK)
    {
        printf("%s: ds_retrieve_u32() failed: %s\n",
                argv[0], strerror(errno));
        return EXIT_FAILURE;
    }
    /* Map major number to our process. */
    if (mapdriver(this_proc, TIME_MAJOR, STYLE_DEV, TRUE) != OK)
    {
        printf("%s: mapdriver() failed: %s\n",
                argv[0], strerror(errno));
        return EXIT_FAILURE;
    }
    /*
     * Run the main loop.
     */
    driver_task(&time_tab);
    return OK;
}
```

Теперь нам нужно дать возможность доступа к портам 0x70 и 0x71 CMOS для нового драйвера устройства `time`. Добавим такую запись к файлу `/etc/drivers.conf` :

```
driver time
```

```
{
    io
        0x70:2;
        system
                UMAP            # 14
                IRQCTL          # 19
                DEVIO           # 21
                #SDEVIO         # 22
                SETALARM        # 24
                TIMES           # 25
                GETINFO         # 26
                SAFECOPYFROM    # 31
                SAFECOPYTO      # 32
                SETGRANT        # 34
                PROFBUF         # 38
                SYSCTL
        ;
        ipc
                SYSTEM PM RS LOG TTY DS VM VFS
                pci inet amddev
                ;
    uid 0;
};
```

Для чтения одного байта из CMOS, программе нужно прежде записать смещение, по которому далее будут читаться данные, адрес I/O порта 0x70. Затем вы можете читать данные из I/O порта 0x71. Наш драйвер часов использует этот механизм для чтения соответствующих полей из CMOS для обновления системного времени, с последующей записью его в тот же способ, как мы делали это в примере 2. А теперь давайте сделаем так:

```
# service up /usr/sbin/time
# cat /dev/time
time_open()
time_transfer()
time_transfer()
time_close()
2009-12-02 19:32:46
```

Поздравляю, вы только что написали свой первый реальный драйвер в MINUX3! Вы можете дальше верифицировать то, что ваш драйвер считывает корректное время, используя команду `date(1)`.

**Пример 4:** RS232 порт.
Здесь в оригинале — пустое место.

Перевод: О.Цилюрик , 03.12.2009
Оригинал находится по адресу:
http://wiki.minix3.org/en/DevelopersGuide/DriverProgramming

Дополнительные источники информации (добавлено переводчиком):
[1] http://wiki.minix3.org/en/DataStore - DataStore Server
[2] обсуждение http://groups.google.com/group/minix3/browse_thread/thread/b795489075a4060f
*> There are a pid and an endpoint for each process.*
*> What's the difference?*

*pid is used by unix system calls, say getpid(), kill()...*
*while endpoint is only used in recv()/send() primitives.*

*> How can I get an endpoint from a pid.*

*./include/minix/endpoint.h*
*_ENDPOINT_P() this macro can safe you some time. :-)*