

Санкт-Петербургский политехнический университет Петра Великого
Институт Информационных Технологий и Управления
Кафедра компьютерных систем и программных технологий

Отчёт по практической работе
по предмету «Системное программное обеспечение»

НАПИСАНИЕ ДРАЙВЕРА СЕТЕВОЙ КАРТЫ

Работу выполнил студент гр. 53501/3 _____ Мартынов С. А.

Работу принял преподаватель _____ Душутина Е. В.

Санкт-Петербург
2015

Содержание

Постановка задачи	3
Введение	4
1 Сетевые интерфейсы в Linux	5
1.1 Структура net_device	5
1.2 Организация доступа системы к устройству	7
1.3 Конфигурационное адресное пространство PCI	8
2 Реализация драйвера	11
2.1 Обнаружение и включение устройства	11
2.2 Инициализация устройства	13
2.3 Реализация функций приёма и передачи	19
Заключение	33
Список литературы	34

Постановка задачи

В рамках данной работы необходимо ознакомиться принципами написания драйверов и реализовать драйвер сетевого устройства.

Привести краткую информация о контроллере сетевого устройства и его технические характеристики. Дать описание порядка разработки драйвера и способы взаимодействия ядра с аппаратурой. Для используемых структур представить назначение основных полей. Описать взаимодействие драйвера, находящегося в пространстве ядра, с приложениями уровня пользователя.

Сетевое устройство (сетевая карта) может быть выбрана студентом самостоятельно.

Введение

Драйвер устройства – это низкоуровневая программа, содержащая специфический код для работы с устройством, которая позволяет пользовательским программам (и самой ОС) управлять им стандартным образом.

В современных версиях ядра Linux по умолчанию присутствуют все необходимые драйверы для всех поддерживаемых устройств[1]. Но для старых версий ядра иногда приходится заниматься бэк-портированием драйверов или даже написанием из с нуля, чтобы обеспечить корректную работу железа.

Все устройства можно разделить на:

- **Символьные.** Чтение и запись устройства идет посимвольно. Примеры таких устройств: клавиатура, последовательные порты.
- **Блочные.** Чтение и запись устройства возможны только блоками, обычно по 512 или 1024 байта. Пример - жесткий диск.
- **Сетевые интерфейсы.** Отличаются тем, что не отображаются на файловую систему, т.е. не имеют соответствующих файлов в директории `/dev`, поскольку из-за специфики этих устройств работа с сетевыми устройствами как с файлами неэффективна. Пример - сетевая карта (`eth0`).

В распоряжении имеется относительно старая материнская плата ASUS P5B на чипсете Intel P965, со встроенной сетевой картой на основе Realtek RTL8111B, для которой будет разработан драйвер, работающий в старой версии ядра Linux.

Это довольно популярная платформа i8169, для которой открыта спецификация. Ссылка на неё приводится в списке использованных материалов.

1 Сетевые интерфейсы в Linux

1.1 Структура net_device

В Linux сетевые устройства рассматриваются как интерфейсы в сетевом стеке. Для этого используется структура net_device. Ниже перечисляются некоторые важные поля структуры net_device, которая будет использоваться далее[1].

```
struct net_device
{
    char *name;
    unsigned long base_addr;
    unsigned char addr_len;
    unsigned char dev_addr[MAX_ADDR_LEN];
    unsigned char broadcast[MAX_ADDR_LEN];
    unsigned short hard_header_len;
    unsigned char irq;
    int (*open) (struct net_device *dev);
    int (*stop) (struct net_device *dev);
    int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
    struct net_device_stats* (*get_stats)(struct net_device *dev);
    void *priv;
};
```

В действительности, эта структура имеет значительно больше полей, для разрабатываемого драйвера вполне достаточно перечисленных:

- name – имя устройства. Если первый символ устройства равен null, то register_netdev назначает ему имя "ethn где n – подходящий номер. Например, если в системе уже есть eth0 и eth1, то новое устройство будет поименовано как eth2.
- base_addr – базовый адрес ввода/вывода. Мы обсудим адресацию ввода/вывода далее в настоящей статье.
- addr_len – длина адреса платы (MAC адреса). Для Ethernet-интерфейсов она равна 6.
- dev_addr – адрес платы (Ethernet-адрес или MAC-адрес).
- broadcast – аппаратный адрес широковещательной передачи. Для Ethernet-интерфейсов это FF:FF:FF:FF:FF:FF.

- `hard_header_len` – ("hardware header length") количество восьмеричных символов, которые предваряют передаваемый пакет и идут перед заголовком IP или другой информацией протокола. Для Ethernet-интерфейсов длина `hard_header_len` равна 14.
- `irq` – номер назначенного прерывания.
- `open` – указатель на функцию, которая открывает устройство. Эта функция вызывается всякий раз, когда `ifconfig` активирует устройство (например, "`ifconfig eth0 up`"). Метод `open` должен регистрировать все необходимые системные ресурсы (порты ввода/вывода, IRQ, DMA и т.п.), включать устройство и увеличивать на единицу счетчик использования модуля.
- `stop` – указатель на функцию, которая останавливает интерфейс. Эта функция вызывается всякий раз, когда `ifconfig` деактивирует устройство (например, "`ifconfig eth0 down`"). Метод `stop` освобождает все ресурсы, запрошенные функцией `open`.
- `hard_start_xmit` – с помощью этой функции заданный пакет передается в сеть. Первым аргументом функции является указатель на структуру `sk_buff`. Структура `sk_buff` используется для хранения пакетов в сетевых стеках Linux.
- `get_stats` – эта функция предоставляет статистику интерфейса. В выходных данных команды "`ifconfig eth0`" большая часть полей содержит данные из `get_stats`.
- `priv` – приватные данные драйвера. Это личное поле драйвера и он может использовать его по своему усмотрению. Далее будет показано, как драйвер будет использовать это поле для хранения данных, относящихся к PCI устройствам.

Как уже отмечалось выше, тут представлены не все поля структуры `net_device`. Но важно отметить то, что полей структуры нет никаких ссылок на функцию, принимающую пакеты. Это делается обработчиком прерываний устройства, что также будет рассмотрено далее в этой работе[3].

1.2 Организация доступа системы к устройству

Ввод-вывод с отображением в память (Memory-Mapped I/O)

Наиболее широко используемый способ ввода/вывода – ввод/вывод с отображением в память (memory-mapped I/O)[2]. Т.е. часть адресного пространства CPU интерпретируется не как адреса памяти, а используется для доступа к устройству. В некоторых системах с определенной архитектурой требуется, чтобы устройства имели фиксированные адреса, но в большинстве систем имеется некоторый способ обнаружения устройств. Хорошим примером такой схемы является обход шины PCI. В настоящей статье не рассматривается, как получить такой адрес, но предполагается, что изначально он у вас есть.

Физический адрес является без знаковым числом типа long. Эти адреса не используются напрямую. Вместо этого для того, чтобы получить адрес, который можно было передать в функцию так, как это описано ниже, вам следует вызвать `ioremap`. В ответ Вы получите адрес, который можно использовать для доступа к устройству.

После завершения использования устройства (к примеру, выход из модуля), необходимо вызвать `iounmap` для того, чтобы вернуть ядру адресное пространство. Архитектура большинства систем позволяет выделять новое адресное пространство каждый раз, когда вызывается `ioremap`, и использовать его до тех пор, пока не будет вызван `iounmap`.

Интерфейс доступа к регистрам устройства

Часть интерфейса, наиболее используемая драйверами, это чтение из регистров устройства, отображаемых в память, и запись в них[2]. Linux предоставляет интерфейс для чтения и записи блоков размером 8, 16, 32 или 64 бита. Исторически сложилось так, что они называются доступом к байту (byte), к слову (word), к длинному числу (long) и к двойному слову или четверке слов (quad). Названия функций следующие - `readb`, `readw`, `readl`, `readq`, `writeb`, `writew`, `writel` и `writeq`.

Для некоторых устройств (работающих, например, с буферами кадров) было бы удобнее за один раз передавать блоки, значительно большие чем 8 байтов. Для этих устройств предлагается использовать функции `memcpy_toio`, `memcpy_fromio` и `memset_io`. Не используйте `memset` или `memcpy` для работы с адресами ввода/вывода; они не гарантируют копирование данных в правильном порядке.

Работа функций чтения и записи должна происходить в определенном порядке. Т.е. компилятору не разрешается выполнять переупорядочивание последовательностей ввода-вывода. Если компилятору разрешается оптимизировать порядок, то Вы можете использовать функцию `__readb` и ей подобные с тем, чтобы не требовать строгого сохранения порядка выполнения операций. Пользуйтесь этим с осторожностью. Операция `rmb` блокирует чтение

памяти. Операция `wmb` блокирует запись в память.

Хотя, основные функции синхронны относительно друг друга, устройства, которые установлены в шинах, сами по себе асинхронны. В частности многим авторам драйверов неудобно, что запись в PCI шину осуществляется асинхронно. Они должны выполнить операцию чтения из устройства с тем, чтобы удостовериться, что запись была сделана так, как хотел автор. Эта особенность скрыта от авторов драйверов в API.

Интерфейс доступа к пространству портов

Еще один широко применяемый вариант ввода-вывода, это пространство портов[3]. Это диапазон адресов, отличающихся от адресного пространства обычной памяти. Доступ к этим адресам обычно не столь быстр, поскольку эти адреса отображаются в адреса памяти, к тому же пространство портов потенциально меньше адресного пространства.

В отличие от ввода-вывода с отображением в память, для доступа к пространству портов подготовка не требуется.

Устройства с отображением ввода-вывода

Доступ к этому пространству обеспечивается с помощью набора функций, в которых допускается доступ к 8, 16 и 32 битам, известных как байт (byte), слово (word) и длинное слово (long). Это следующие функции - `inb`, `inw`, `inl`, `outb`, `outw` и `outl`.

Эти функции имеют несколько вариантов. Для некоторых устройств требуется, чтобы доступ к их портам происходил со сниженной скоростью. Эта функциональность обеспечивается при помощи добавления `_r` в конце команды. Имеются также эквиваленты команды `memsru`. Функции `ins` и `out` копируют байты, слова и длинные слова в заданный порт и из него.

1.3 Конфигурационное адресное пространство PCI

Одним из главных усовершенствований шины PCI по сравнению с другими архитектурами ввода-вывода стал её конфигурационный механизм, обладающий конфигурационным адресным пространством, состоящим из 256 байт, которые можно адресовать, зная номер шины PCI, номер устройства и номер функции в устройстве. Первые 64 байта используются стандартным образом, тогда как использование оставшихся байтов зависит от устройства.

На рис.1. показано стандартное конфигурационное адресное пространство PCI. Регистры `DeviceID`, `VendorID`, `Status`, `Command`, `Class Code`, `Revision ID`, `Header Type` являются обязательными для всех PCI-устройств (для многих типов устройств обязательными являются также регистры `Subsystem ID` и `Subsystem Vendor ID`).

Все остальные регистры являются опциональными.

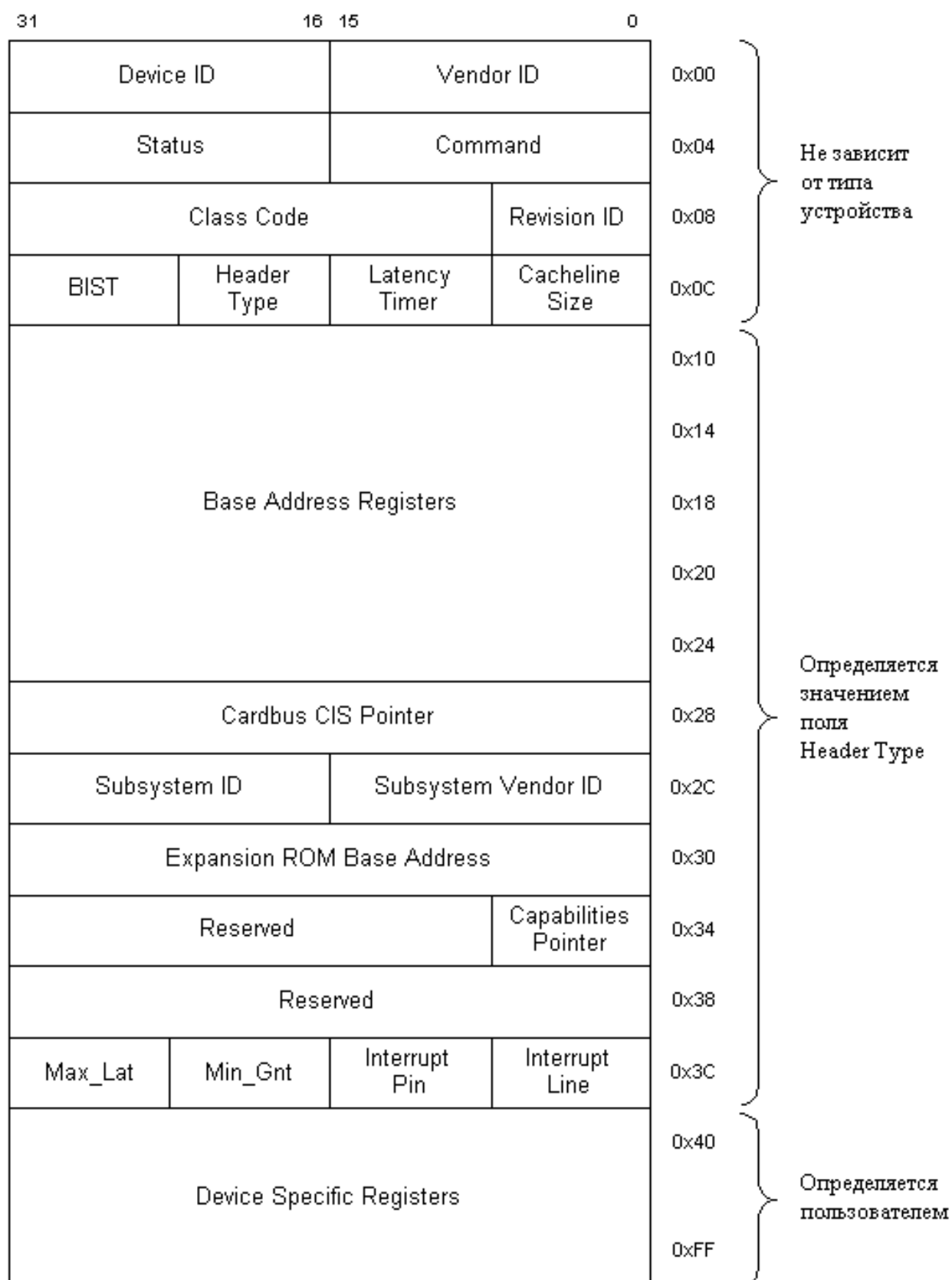


Рис. 1: Конфигурационное адресное пространство

Поля Vendor ID, Device ID и Class Code содержат код фирмы-изготовителя устройства, код устройства и код класса устройства. Классификация устройств и указание кода класса в его конфигурационном пространстве является важной частью спецификации PCI[2].

Код изготовителя, код устройства и код класса применяются в процессе поиска заданного устройства. Если необходимо найти конкретное устройство, то поиск выполняется по кодам устройства и его изготовителя; если необходимо найти все устройства определенного типа, то поиск выполняется по коду класса устройства. После того как устройство найдено, при помощи регистров базовых адресов можно определить выделенные ему области в адресном пространстве памяти и пространстве ввода-вывода (I/O).

Регистры базовых адресов (Base Address Registers) содержат выделенные устройству области в адресном пространстве и пространстве портов I/O. Бит 0 во всех регистрах базовых адресов определяет, куда будет отображен ресурс – на пространство портов I/O или на адресное пространство. Регистр базового адреса, отображаемый на пространство портов, всегда 32-разрядный, бит 0 установлен в 1. Регистр базового адреса, отображаемый на адресное пространство, может быть 32- и 64-разрядным, бит 0 сброшен в 0.

Использование этих регистров будет продемонстрировано в этой работе далее.

2 Реализация драйвера

2.1 Обнаружение и включение устройства

Перед началом работы, необходимо провести поиск и идентификацию устройства. В ядре Linux имеется достаточно богатый набор API для обнаружения устройств, использующих шину PCI (Plug & Play). Функция `pci_find_device` (Листинг 1, строка 20) ищет устройство в списке устройств с заданной сигнатурой или принадлежащие к определённому классу. Если ничего не найдено, возвращается значение `NULL`.

Листинг 1: Обнаружение и включение устройства (`res/r8169-alpha.c`)

```
1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <linux/stddef.h>
4 #include <linux/pci.h>
5
6 static struct pci_dev* probe_for_realtek8169(void)
7 {
8     struct pci_dev *pdev = NULL;
9     /* Ensure we are not working on a non-PCI system */
10    if (!pci_present())
11    {
12        LOG_MSG("<1>pci not present\n");
13        return pdev;
14    }
15
16    #define REALTEK_VENDER_ID 0x10EC
17    #define REALTEK_DEVICE_ID 0X8169
18
19    /* Look for RealTek 8169 NIC */
20    pdev = pci_find_device(REALTEK_VENDER_ID, REALTEK_DEVICE_ID, NULL)
21        ;
22    if (pdev)
23    {
24        /* device found, enable it */
25        if (pci_enable_device(pdev))
26        {
27            LOG_MSG("Could not enable the device\n");
```

```

27     return NULL;
28 }
29 else
30     LOG_MSG("Device enabled\n");
31 }
32 else
33 {
34     LOG_MSG("device not found\n");
35     return pdev;
36 }
37 return pdev;
38 }
39
40 int init_module(void)
41 {
42     struct pci_dev *pdev;
43     pdev = probe_for_realtek8169();
44     if (!pdev)
45         return 0;
46
47     return 0;
48 }

```

Каждый производитель имеет уникальный, назначенный только ему идентификатор ID и назначает уникальный идентификатор ID каждому конкретному виду устройств. Макросы `REALTEK_VENDOR_ID` и `REALTEK_DEVICE_ID` (Листинг 1, строки 16 и 17) определяют эти идентификаторы ID. Найти эти значения можно в "PCI Configuration Space Table" в спецификациях RealTek8169[4].

После того, как устройство обнаружено, то, прежде чем как-то с ним взаимодействовать, его нужно включить.

Функция `probe_for_realtek8169` (Листинг 1, строка 43) выполняет следующие задачи:

- Проверяет, что мы работаем на PCI-совместимой системе (`pci_present` вернет `NULL`, если система не поддерживает PCI).
- Пытается найти устройство RealTek 8169 так, как это показано в таблице 1.
- Включает устройство (с помощью функции `pci_enable_device`), если его найдет.

2.2 Инициализация устройства

На прошлом шаге был разработан драйвер, который позволяет обнаружить и включить сетевое устройство. На данном шаге будет представлен драйвер, который будет способен инициализировать устройство.

Ранее была рассмотрена структура `net_device`, содержащее поле `priv`. Объявим структуру, в которой будут храниться приватные данные устройства (листинг 2), а указатель на эту структуру должен храниться в поле `priv`.

Листинг 2: Структура `rtl8169_private` (`res/r8169-betta.c`)

```
40 struct rtl8169_private
41 {
42     struct pci_dev *pci_dev; /* PCI device */
43     void *mmio_addr; /* memory mapped I/O addr */
44     unsigned long regs_len; /* length of I/O or MMI/O region */
45 };
```

В остальной части функции `init_module` теперь можно определить указатель `net_device` и инициализировать его (листинг 3).

Листинг 3: Инициализация `net_device` (`res/r8169-betta.c`)

```
47 #define DRIVER "rtl8169"
48 static struct net_device *rtl8169_dev;
49
50 static int rtl8169_init(struct pci_dev *pdev, struct net_device **
    dev_out)
51 {
52     struct net_device *dev;
53     struct rtl8169_private *tp;
54
55     /*
56      * alloc_etherdev allocates memory for dev and dev->priv.
57      * dev->priv shall have sizeof(struct rtl8169_private) memory
58      * allocated.
59      */
60     dev = alloc_etherdev(sizeof(struct rtl8169_private));
61     if (!dev)
62     {
```

```

63     LOG_MSG("Could not allocate etherdev\n");
64     return -1;
65 }
66
67 tp = dev->priv;
68 tp->pci_dev = pdev;
69 *dev_out = dev;
70
71 return 0;
72 }
73
74 int init_module(void)
75 {
76     struct pci_dev *pdev;
77     unsigned long mmio_start, mmio_end, mmio_len, mmio_flags;
78     void *ioaddr;
79     struct rtl8169_private *tp;
80     int i;
81
82     pdev = probe_for_realtek8169();
83     if (!pdev)
84         return 0;
85
86     if (rtl8169_init(pdev, &rtl8169_dev))
87     {
88         LOG_MSG("Could not initialize device\n");
89         return 0;
90     }
91
92     tp = rtl8169_dev->priv; /* rtl8169 private information */
93
94     /* get PCI memory mapped I/O space base address from BAR1 */
95     mmio_start = pci_resource_start(pdev, 1);
96     mmio_end = pci_resource_end(pdev, 1);
97     mmio_len = pci_resource_len(pdev, 1);
98     mmio_flags = pci_resource_flags(pdev, 1);
99

```

```

100  /* make sure above region is MMI/O */
101  if (!(mmio_flags & I / ORESOURCE_MEM))
102  {
103      LOG_MSG("region not MMI/O region\n");
104      goto cleanup1;
105  }
106
107  /* get PCI memory space */
108  if (pci_request_regions(pdev, DRIVER))
109  {
110      LOG_MSG("Could not get PCI region\n");
111      goto cleanup1;
112  }
113
114  pci_set_master(pdev);
115
116  /* ioremap MMI/O region */
117  ioaddr = ioremap(mmio_start, mmio_len);
118  if (!ioaddr)
119  {
120      LOG_MSG("Could not ioremap\n");
121      goto cleanup2;
122  }
123
124  rtl8169_dev->base_addr = (long) ioaddr;
125  tp->mmio_addr = ioaddr;
126  tp->regs_len = mmio_len;
127
128  /* UPDATE NET_DEVICE */
129
130  for (i = 0; i < 6; i++)
131  { /* Hardware Address */
132      rtl8169_dev->dev_addr[i] = readb(rtl8169_dev->base_addr + i);
133      rtl8169_dev->broadcast[i] = 0xff;
134  }
135  rtl8169_dev->hard_header_len = 14;
136

```

```

137 memcpy(rtl8169_dev->name, DRIVER, sizeof(DRIVER)); /* Device Name
    */
138 rtl8169_dev->irq = pdev->irq; /* Interrupt Number */
139 rtl8169_dev->open = rtl8169_open;
140 rtl8169_dev->stop = rtl8169_stop;
141 rtl8169_dev->hard_start_xmit = rtl8169_start_xmit;
142 rtl8169_dev->get_stats = rtl8169_get_stats;
143
144 /* register the device */
145 if (register_netdev(rtl8169_dev))
146 {
147     LOG_MSG("Could not register netdevice\n");
148     goto cleanup0;
149 }
150
151 return 0;
152 }

```

Функция `probe_for_realtek8169` (листинг 3, строка 82) была рассмотрена на предыдущем шаге. Функция `rtl8169_init` (листинг 3, строка 50) распределяет память для локального указателя `dev` (листинг 3, строка 60), который должен использовать как `net_device`. Кроме того, эта функция заполняет компоненту `pci_dev` структуры `rtl8169_private` (листинг 3, строка 67) для обнаруженного устройства.

Далее необходимо получить поле `base_addr` для `net_device`. Оно указывает на начало памяти регистров устройства, которые потребуются для реализации ввода-вывода с отображением в память. Чтобы получить базовый адрес для ввода-вывода с отображением в память, используем PCI API такие как `pci_resource_start` (листинг 3, строка 95), `pci_resource_end` (листинг 3, строка 96), `pci_resource_len` (листинг 3, строка 97), `pci_resource_flags` (листинг 3, строка 98). Эти API позволяют читать конфигурационное пространство PCI, не зная о деталях его реализации. Вторым аргументом в этих API – номер BAR. Согласно спецификации RealTek8169, первый BAR (нумеруемый как 0) – I/OAR, тогда как второй BAR (нумеруемый как 1) – MEMAR. Поскольку в этом драйвере используется ввод-вывод с отображением в память, то в качестве второго аргумента передаем 1.

Прежде, чем получить доступ к адресам, возвращаемыми указанными выше API, нужно должны сделать две вещи:

1. Зарезервировать в драйвере указанные выше ресурсы (пространство памяти); это

делается с помощью вызова функции `pci_request_regions` (листинг 3, строка 108).

2. Отобразить адреса ввода-вывода (описанные выше) так, чтобы они использовались при вводе-выводе с отображением в память.

Адрес `ioaddr` назначается компоненте `base_addr` в структуре `net_device` (листинг 3, строка 124), и это то место, откуда можно начинать читать регистры устройства или писать в них.

В оставшейся части кода из листинга 3, выполняется обычная инициализация структуры `net_device`. Стоит отметить, что производится чтение аппаратного адреса из устройства и запись его в `dev_addr`. Если изучить описания регистров в спецификации RealTek8169, то можно видеть, что первые 6 байтов являются аппаратным адресом устройства. Также нужно иметь проинициализированные компоненты указателя на функцию, но без определения какой-либо соответствующую функцию.

Теперь для компиляции модуля без ошибок, добавим пару заглушек (листинг 4).

Листинг 4: Функции-заглушки (`res/r8169-betta.c`)

```
154 static int rtl8169_open(struct net_device *dev)
155 {
156     LOG_MSG("rtl8169_open is called\n");
157     return 0;
158 }
159
160 static int rtl8169_stop(struct net_device *dev)
161 {
162     LOG_MSG("rtl8169_open is called\n");
163     return 0;
164 }
165
166 static int rtl8169_start_xmit(struct sk_buff *skb, struct net_device
    *dev)
167 {
168     LOG_MSG("rtl8169_start_xmit is called\n");
169     return 0;
170 }
171
172 static struct net_device_stats* rtl8169_get_stats(struct net_device
    *dev)
173 {
```

```

174 LOG_MSG("rtl8169_get_stats is called\n");
175 return 0;
176 }

```

В `init_module` пропущена обработка ошибок. В самом простом случае, она может выглядеть так, как показано в листинге 5.

Листинг 5: Обработка ошибок (`res/r8169-betta.c`)

```

178 void cleanup_module(void)
179 {
180     struct rtl8169_private *tp;
181     tp = rtl8169_dev->priv;
182
183     iounmap(tp->mmio_addr);
184     pci_release_regions(tp->pci_dev);
185
186     unregister_netdev(rtl8169_dev);
187     pci_disable_device(tp->pci_dev);
188     return;
189 }

```

На данный момент драйвер может включать устройство и отображать его в пространство пользователя. Можно откомпилировать его код и вставить его в ядро (предполагается, что исходный код ядра - `/usr/src/linux-2.4.22`). Погрузка модуля в ядро (`insmod`) потребует прав суперпользователя.

```

$ gcc -c rtl8169.c -D__KERNEL__ -DMODULE -I /usr/src/linux-2.4.22/include
$ sudo insmod rtl8169.o

```

2.3 Реализация функций приёма и передачи

На предыдущем шаге была показана инициализация устройства, но функции приёма и передачи заменены заглушками. Теперь рассмотрим как осуществлять приём и передачу сетевого трафика.

В RealTek8169 имеется 4 дескриптора передачи, каждый дескриптор имеет фиксированное смещение адреса ввода-вывода[4]. Четыре дескриптора используются циклически. Это означает, что для передачи четырех пакетов драйвер будет использовать в циклическом порядке дескриптор 0, дескриптор 1, дескриптор 2 и дескриптор 3. Для передачи следующего пакета драйвер снова будет использовать дескриптор 0 (при условии, что он свободен). В спецификациях RealTek8169 указывается, что регистры TSAD0, TSAD1, TSAD2 и TSAD3 имеют смещения 0x20, 0x24, 0x28, 0x2C, соответственно. В этих регистрах хранится "начальный адрес дескрипторов передачи т.е. в них хранится стартовый адрес (в памяти) пакетов, которые должны быть переданы. Позже устройство считает содержимое пакетов из этих адресов DMA, переписшет их в свой собственный стек FIFO, а затем выполнит передачу данных в сеть. Таким образом, драйвер должен выделять память прямого доступа (доступ DMA), где будут храниться пакеты, и записывает адрес этой памяти в регистры TSAD.

Приемная часть RTL8169 спроектирована как кольцевой буфер (линейная память, управление которой осуществляется как кольцевой памятью). Всякий раз, когда устройство принимает пакет, содержимое пакета запоминается в память кольцевого буфера и изменяется место, куда будет записываться содержимое следующего пакета (начальный адрес первого пакета + длина первого пакета). Устройство продолжает так запоминать пакеты до тех пор, пока не исчерпается линейная память. В этом случае устройство снова начинает запись с начального адреса линейной памяти, реализуя, таким образом, кольцевой буфер.

Добавим в структуру rtl8169_private поля, в которых будут храниться данные, связанные с передачей пакетов (листинг 6).

Листинг 6: Инициализация net_device (res/r8169.c)

```
93 #define NUM_TX_DESC 4
94 struct rtl8169_private
95 {
96     struct pci_dev *pci_dev; /* PCI device */
97     void *mmio_addr; /* memory mapped I/O addr */
98     unsigned long regs_len; /* length of I/O or MMI/O region */
99     unsigned int tx_flag;
100    unsigned int cur_tx;
```

```

101 unsigned int dirty_tx;
102 unsigned char *tx_buf[NUM_TX_DESC]; /* Tx bounce buffers */
103 unsigned char *tx_bufs; /* Tx bounce buffer region. */
104 dma_addr_t tx_bufs_dma;
105
106 struct net_device_stats stats;
107 unsigned char *rx_ring;
108 dma_addr_t rx_ring_dma;
109 unsigned int cur_rx;
110 };

```

Компонента `tx_flag` (листинг 6, строка 99) должна содержать флаги передачи, уведомляющие устройство о некоторых параметрах, которые будут рассмотрены ниже. Поле `cur_tx` (листинг 6, строка 100) должно хранить текущий дескриптор передачи, а `dirty_tx` (листинг 6, строка 101) указывает на первый из дескрипторов передачи, для которых передача еще не завершена (это означает, что мы можем использовать эти занятые дескрипторы для следующей передачи пакетов до тех пор, пока предыдущий пакет не будет полностью передан). В массиве `tx_buf` (листинг 6, строка 102) хранятся адреса четырех "дескрипторов передачи". Поле `tx_bufs` (листинг 6, строка 103) используется для тех же самых целей. В `tx_buf` и в `tx_bufs` должен храниться именно виртуальный адрес, который может использоваться драйвером, но устройство не может использовать такие адреса. Устройство для доступа нужен физический адрес, который запоминается в поле `tx_bufs_dma` (листинг 6, строка 104).

Компонента `stats` (листинг 6, строка 106) должна хранить статистику с устройства (большая часть статистики, выдаваемой `ifconfig`, берется из этой структуры). Следующая компонента, `rx_ring` (листинг 6, строка 107), является адресом памяти в ядре, где запоминаются принятые пакеты, а `rx_ring_dma` (листинг 6, строка 108) – физический адрес этой памяти. Компонент `cur_rx` (листинг 6, строка 109) используется для отслеживания места, куда будет записываться следующий пакет.

Ниже (листинг 7) приведен список смещений регистров, используемых в исходном коде (более подробную информацию об этих значениях можно получить из спецификаций RealTek8169). Кроме этого, там размер памяти, нужный для кольцевого буфера (листинг 7, строки 6-14), используемого для приёма сообщений.

Листинг 7: Определения регистров (res/r8169.c)

```

6 #define TX_BUF_SIZE 1536 /* should be at least MIU + 14 + 4 */
7 #define TOTAL_TX_BUF_SIZE (TX_BUF_SIZE * NUM_TX_SIZE)
8
9 /* Size of the in-memory receive ring. */
10 #define RX_BUF_LEN_IDX 2 /* 0==8K, 1==16K, 2==32K, 3==64K */
11 #define RX_BUF_LEN (8192 << RX_BUF_LEN_IDX)
12 #define RX_BUF_PAD 16 /* see 11th and 12th bit of RCR:
    0x44 */
13 #define RX_BUF_WRAP_PAD 2048 /* spare padding to handle pkt wrap
    */
14 #define RX_BUF_TOT_LEN (RX_BUF_LEN + RX_BUF_PAD + RX_BUF_WRAP_PAD)
15
16 /* 8169 register offsets */
17 #define TSD0 0x10
18 #define TSAD0 0x20
19 #define RBSTART 0x30
20 #define CR 0x37
21 #define CAPR 0x38
22 #define IMR 0x3c
23 #define ISR 0x3e
24 #define TCR 0x40
25 #define RCR 0x44
26 #define MPC 0x4c
27 #define MULINT 0x5c
28
29 /* TSD register commands */
30 #define TxHostOwns 0x2000
31 #define TxUnderrun 0x4000
32 #define TxStatOK 0x8000
33 #define TxOutOfWindow 0x20000000
34 #define TxAborted 0x40000000
35 #define TxCarrierLost 0x80000000
36
37 /* CR register commands */
38 #define RxBufEmpty 0x01
39 #define CmdTxEnb 0x04

```

```

40 #define CmdRxEnb    0x08
41 #define CmdReset    0x10
42
43 /* ISR Bits */
44 #define RxOK        0x01
45 #define RxErr       0x02
46 #define TxOK        0x04
47 #define TxErr       0x08
48 #define RxOverFlow  0x10
49 #define RxUnderrun  0x20
50 #define RxFIFOOver  0x40

```

Теперь рассмотрим реализацию функций, работающих с устройством для передачи данных (листинг 8). Ранее эти функции были представлены простыми заглушками.

Листинг 8: Определения регистров (res/r8169.c)

```

232 static int rtl8169_open(struct net_device *dev)
233 {
234     int retval;
235     struct rtl8169_private *tp = dev->priv;
236
237     /* get the IRQ
238      * second arg is interrupt handler
239      * third is flags, 0 means no IRQ sharing
240      */
241     retval = request_irq(dev->irq, rtl8169_interrupt, 0, dev->name,
242                          dev);
243     if (retval)
244         return retval;
245
246     /* get memory for Tx buffers
247      * memory must be DMAable
248      */
249     tp->tx_bufs = pci_alloc_consistent(tp->pci_dev, TOTAL_TX_BUF_SIZE,
250                                         &tp->tx_bufs_dma);
251     tp->rx_ring = pci_alloc_consistent(tp->pci_dev, RX_BUF_TOT_LEN,
252                                         &tp->rx_ring_dma);
253

```

```

253     if ((!tp->tx_bufs) || (!tp->rx_ring))
254     {
255         free_irq(dev->irq, dev);
256
257         if (tp->tx_bufs)
258         {
259             pci_free_consistent(tp->pci_dev, TOTAL_TX_BUF_SIZE, tp->
                tx_bufs,
260                 tp->tx_bufs_dma);
261             tp->tx_bufs = NULL;
262         }
263         if (tp->rx_ring)
264         {
265             pci_free_consistent(tp->pci_dev, RX_BUF_TOT_LEN, tp->rx_ring,
                tp->rx_ring_dma);
266             tp->rx_ring = NULL;
267         }
268         return -ENOMEM;
269     }
270
271
272     tp->tx_flag = 0;
273     rtl8169_init_ring(dev);
274     rtl8169_hw_start(dev);
275
276     return 0;
277 }
278
279 static void rtl8169_init_ring(struct net_device *dev)
280 {
281     struct rtl8169_private *tp = dev->priv;
282     int i;
283
284     tp->cur_tx = 0;
285     tp->dirty_tx = 0;
286
287     for (i = 0; i < NUM_TX_DESC; i++)
288         tp->tx_buf[i] = &tp->tx_bufs[i * TX_BUF_SIZE];

```

```

289
290     return;
291 }
292
293 static void rtl8169_hw_start(struct net_device *dev)
294 {
295     struct rtl8169_private *tp = dev->priv;
296     void *ioaddr = tp->mmio_addr;
297     u32 i;
298
299     rtl8169_chip_reset(ioaddr);
300
301     /* Must enable Tx/Rx before setting transfer thresholds! */
302     writeb(CmdTxEnb | CmdRxEnb, ioaddr + CR);
303
304     /* tx config */
305     writel(0x00000600, ioaddr + TCR); /* DMA burst size 1024 */
306
307     /* rx config */
308     writel((((1 << 12) | (7 << 8) | (1 << 7) | (1 << 3) | (1 << 2) | (1
        << 1)),
309         ioaddr + RCR);
310
311     /* init Tx buffer DMA addresses */
312     for (i = 0; i < NUM_TX_DESC; i++)
313     {
314         writel(tp->tx_bufs_dma + (tp->tx_buf[i] - tp->tx_bufs),
315             ioaddr + TSAD0 + (i * 4));
316     }
317
318     /* init RBSTART */
319     writel(tp->rx_ring_dma, ioaddr + RBSTART);
320
321     /* initialize missed packet counter */
322     writel(0, ioaddr + MPC);
323
324     /* no early-rx interrupts */

```



```

325     writew((readw(ioaddr + MULINT) & 0xF000), ioaddr + MULINT);
326
327     /* Enable all known interrupts by setting the interrupt mask. */
328     writew(INT_MASK, ioaddr + IMR);
329
330     netif_start_queue(dev);
331     return;
332 }
333
334 static void rtl8169_chip_reset(void *ioaddr)
335 {
336     int i;
337
338     /* Soft reset the chip. */
339     writeb(CmdReset, ioaddr + CR);
340
341     /* Check that the chip has finished the reset. */
342     for (i = 1000; i > 0; i--)
343     {
344         barrier();
345         if ((readb(ioaddr + CR) & CmdReset) == 0)
346             break;
347         udelay(10);
348     }
349     return;
350 }

```

Функция `rtl8169_open` (листинг 8, строка 232) начинается с запроса IRQ, что делается вызовом API `request_irq` (листинг 8, строка 241). В этой функции регистрируется обработчик прерываний `rtl8169_interrupt`. Эта функция будет вызываться ядром всякий раз, когда устройство генерирует прерывание. Теперь выделим память, куда будут записываться исходящие пакеты, прежде чем они будут переданы в сеть. Вызов API `pci_allocate_consistant` возвращает виртуальный адрес ядра (листинг 8, строка 248). Физический адрес возвращается в третьем аргументе, который потом будет использован драйвером. Таким образом выделяется память для всех четырех дескрипторов. Функция `rtl8169_init_ring` (листинг 8, строка 279) распределяет эту память для четырех дескрипторов.

Теперь рассмотрим функцию `rtl8169_hw_start` (листинг 8, строка 293), которая делает

устройство готовым для передачи (и приёма) пакетов. Сначала перезагружаем устройство для того, чтобы оно было в предсказуемом и известном состоянии (листинг 8, строка 299). Это делается с помощью записи в регистр команд CR (Command Register) значения reset (листинг 8, строка 339), что описано в спецификациях. Дождёмся, пока записанное значение можно будет считать обратно, что означает, что устройство перезагружено. Следующая функция, barrier(), вызывается с тем, чтобы ядро выделило память, требуемую для ввода-выхода, без какой-либо оптимизации (листинг 8, строка 344). Поскольку устройство перезагружено, можно записать в регистр CR значения CmdTxEnb | CmdRxEnb, что означает, что устройство будет передавать и принимать пакеты (листинг 8, строка 302). Затем сконфигурируем регистр TCR (Transmission Configuration Register – конфигурационный регистр передачи) (листинг 8, строка 305). Единственное, что мы установим в регистре TCR, это "Max DMA Burst Size per Tx DMA Burst" (максимальный размер буфера DMA для каждого сохраняемого в DMA пакета). Все остальные значения оставим установленными по умолчанию (подробности в спецификациях). Теперь запишем DMA адрес всех четырех дескрипторов в регистры TSAD (Transmission Start Address Descriptor – дескриптор начального адреса передачи) (листинг 8, строка 315). Остальные строки пока останутся без комментариев. Они потребуются для передачи.

Далее включаем прерывания, сделав для этого запись в регистр IMR (Interrupt Mask Register – регистр маски прерывания) (листинг 8, строка 328). Этот регистр позволит задать, какие прерывания будет генерировать устройство. Наконец, вызываем netif_start_queue (листинг 8, строка 330) с тем, чтобы сообщить ядру, что устройство готово. Осталось лишь реализовать функцию rtl8169_interrupt. Устройство уже готово посылать пакеты, реализация этой функции представлена ниже (листинг 9).

Листинг 9: Функция отправки пакетов (res/r8169.c)

```

352 static int rtl8169_start_xmit(struct sk_buff *skb, struct net_device
    *dev)
353 {
354     struct rtl8169_private *tp = dev->priv;
355     void *ioaddr = tp->mmio_addr;
356     unsigned int entry = tp->cur_tx;
357     unsigned int len = skb->len;
358 #define ETH_MIN_LEN 60 /* minimum Ethernet frame size */
359     if (len < TX_BUF_SIZE)
360     {
361         if (len < ETH_MIN_LEN)
362             memset(tp->tx_buf[entry], 0, ETH_MIN_LEN);

```

```

363     skb_copy_and_csum_dev(skb, tp->tx_buf[entry]);
364     dev_kfree_skb(skb);
365 }
366 else
367 {
368     dev_kfree_skb(skb);
369     return 0;
370 }
371 writel(tp->tx_flag | max(len, (unsigned int) ETH_MIN_LEN),
372        ioaddr + TSD0 + (entry * sizeof(u32)));
373 entry++;
374 tp->cur_tx = entry % NUM_TX_DESC;
375
376 if (tp->cur_tx == tp->dirty_tx)
377 {
378     netif_stop_queue(dev);
379 }
380 return 0;
381 }

```

Функция `rtl8169_start_xmit` (листинг 9, строка 352) крайне проста: сначала она ищет имеющийся дескриптор передачи, а затем проверяет, чтобы размер пакета был, по меньшей мере, 60 байтов (поскольку размер пакетов Ethernet не может быть меньше 60 байтов) (листинг 9, строка 359). Как только это будет сделано, будет вызвана функция `skb_copy_and_csum_dev` (листинг 9, строка 363), которая скопирует содержимое пакетов в имеющуюся память DMA. Следующая функция `writel` (листинг 9, строка 371) информирует устройство о длине пакета. После этого пакеты передаются в сеть. Далее определяем имеющиеся в наличии следующие дескрипторы передачи и, если он будет равен дескриптору, для которого передача еще не завершена (листинг 9, строка 376), то остановим устройство; в противном случае просто выйдем из функции.

Теперь устройство готово отсылать пакеты. Для приёма пакетов, вернёмся к листингу 8. В строке 308 производится конфигурирование сетевой карты для приёма сообщений (подробности есть в спецификации[4]):

- Бит 1 – Принимаются пакеты с проверкой физического адреса (адреса MAC).
- Бит 2 – Принимаются многоадресные пакеты (посылаемые на несколько адресов)
- Бит 3 – Принимаются широковещательные пакеты (посылаемые на все адреса)

- Бит 7 – WRAP. Когда установлен в 1, то RTL8169 будет перемещать оставшуюся часть пакетных данных в память, которая следует непосредственно за концом приемного буфера.
- Биты 8-10 – Максимальный размер буфера DMA для каждого сохраняемого в DMA пакета. Значение 111 соответствует отсутствию ограничений.
- Биты 11-12 – Длина буфера приема. Устанавливаем это значение равным 10, что означает 32K+16 байтов.

Кроме того, стоит отметить конфигурирование регистра RBSTART (листинг 8, строка 319). В этом регистре содержится стартовый адрес приемного буфера. Далее обнуляем регистр MPC (Missed Packet Counter – счетчик ошибочных пакетов) (листинг 8, строка 322) и конфигурируем устройство так, чтобы не генерировались ранние прерывания (листинг 8, строка 325).

Последний важный момент, который нужно обсудить – обработчик прерываний устройства. Этот обработчик прерываний ответственен за прием пакетов, а также за обновление необходимой статистики. Ниже (листинг 10) приведен его исходный код.

Листинг 10: Обработчик прерываний (res/r8169.c)

```

383 static void rtl8169_interrupt(int irq, void *dev_instance, struct
    pt_regs *regs)
384 {
385     struct net_device *dev = (struct net_device*) dev_instance;
386     struct rtl8169_private *tp = dev->priv;
387     void *ioaddr = tp->mmio_addr;
388     unsigned short isr = readw(ioaddr + ISR);
389
390     /* clear all interrupt.
391      * Specs says reading ISR clears all interrupts and writing
392      * has no effect. But this does not seem to be case. I keep on
393      * getting interrupt unless I forcibly clears all interrupt :-(
394      */
395     writew(0xffff, ioaddr + ISR);
396
397     if ((isr & TxOK) || (isr & TxErr))
398     {
399         while ((tp->dirty_tx != tp->cur_tx) || netif_queue_stopped(dev))
400         {

```

```

401     unsigned int txstatus = readl(
402         ioaddr + TSD0 + tp->dirty_tx * sizeof(int));
403
404     if (!(txstatus & (TxStatOK | TxAborted | TxUnderrun)))
405         break; /* yet not transmitted */
406
407     if (txstatus & TxStatOK)
408     {
409         LOG_MSG("Transmit OK interrupt\n");
410         tp->stats.tx_bytes += (txstatus & 0x1fff);
411         tp->stats.tx_packets++;
412     }
413     else
414     {
415         LOG_MSG("Transmit Error interrupt\n");
416         tp->stats.tx_errors++;
417     }
418
419     tp->dirty_tx++;
420     tp->dirty_tx = tp->dirty_tx % NUM_TX_DESC;
421
422     if ((tp->dirty_tx == tp->cur_tx) & netif_queue_stopped(dev))
423     {
424         LOG_MSG("waking up queue\n");
425         netif_wake_queue(dev);
426     }
427 }
428 }
429
430 if (isr & RxErr)
431 {
432     /* TODO: Need detailed analysis of error status */
433     LOG_MSG("receive err interrupt\n");
434     tp->stats.rx_errors++;
435 }
436
437 if (isr & RxOK)

```

```

438 {
439     LOG_MSG("receive interrupt received\n");
440     while ((readb(ioaddr + CR) & RxBufEmpty) == 0)
441     {
442         unsigned int rx_status;
443         unsigned short rx_size;
444         unsigned short pkt_size;
445         struct sk_buff *skb;
446
447         if (tp->cur_rx > RX_BUF_LEN)
448             tp->cur_rx = tp->cur_rx % RX_BUF_LEN;
449
450         /* TODO: need to convert rx_status from little to host endian
451          * XXX: My CPU is little endian only :-)
452          */
453         rx_status = *(unsigned int*) (tp->rx_ring + tp->cur_rx);
454         rx_size = rx_status >> 16;
455
456         /* first two bytes are receive status register
457          * and next two bytes are frame length
458          */
459         pkt_size = rx_size - 4;
460
461         /* hand over packet to system */
462         skb = dev_alloc_skb(pkt_size + 2);
463         if (skb)
464         {
465             skb->dev = dev;
466             skb_reserve(skb, 2); /* 16 byte align the IP fields */
467
468             eth_copy_and_sum(skb, tp->rx_ring + tp->cur_rx + 4, pkt_size
469                             ,
470                             0);
471
472             skb_put(skb, pkt_size);
473             skb->protocol = eth_type_trans(skb, dev);
474             netif_rx(skb);

```

```

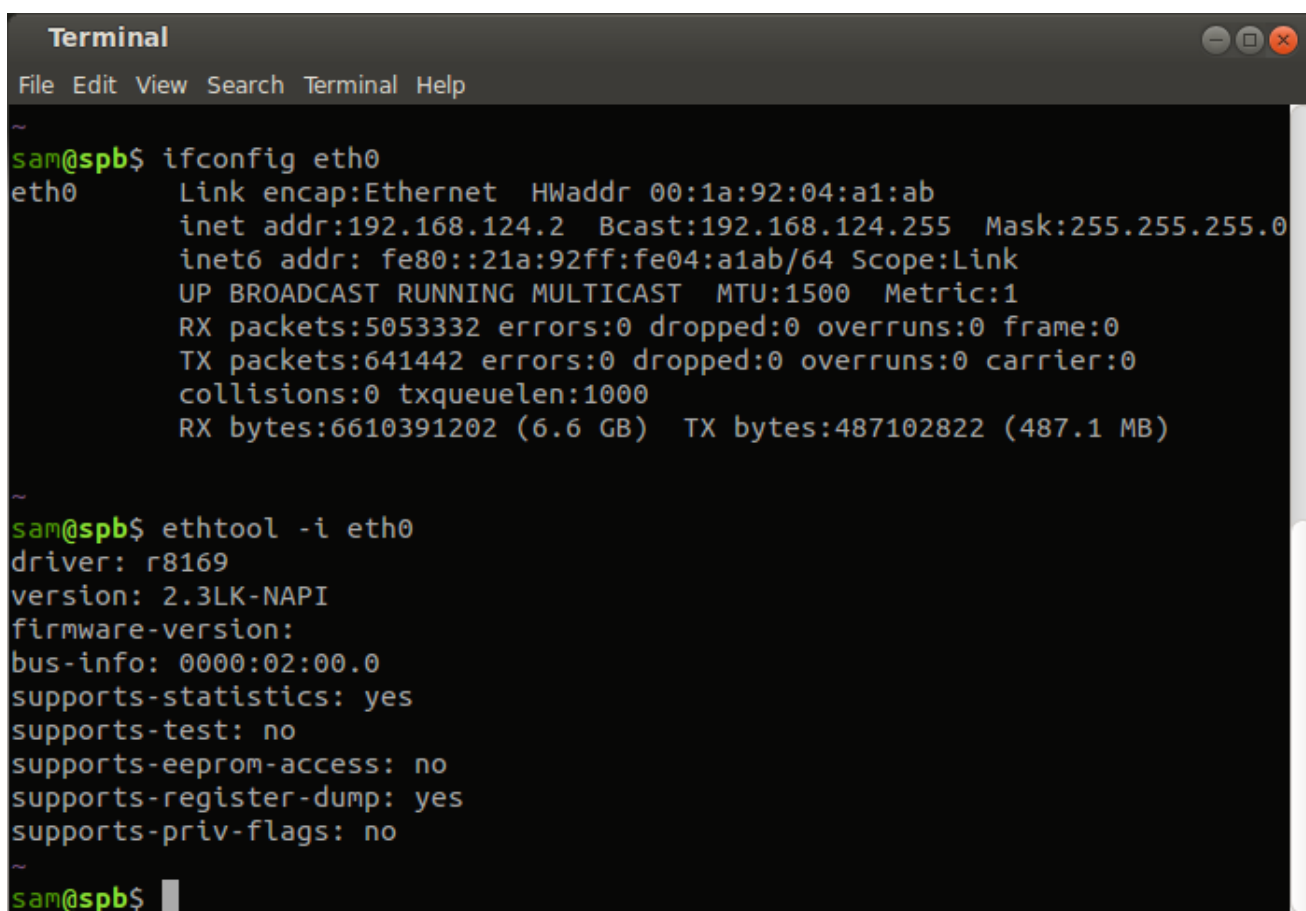
474
475     dev->last_rx = jiffies;
476     tp->stats.rx_bytes += pkt_size;
477     tp->stats.rx_packets++;
478 }
479 else
480 {
481     LOG_MSG("Memory squeeze, dropping packet.\n");
482     tp->stats.rx_dropped++;
483 }
484
485 /* update tp->cur_rx to next writing location */
486 tp->cur_rx = (tp->cur_rx + rx_size + 4 + 3) & ~3;
487
488 /* update CAPR */
489 writew(tp->cur_rx, ioadr + CAPR);
490 }
491 }
492
493 if (isr & CableLen)
494     LOG_MSG("cable length change interrupt\n");
495 if (isr & Timeout)
496     LOG_MSG("time interrupt\n");
497 if (isr & SysErr)
498     LOG_MSG("system err interrupt\n");
499 return;
500 }

```

Как видно из листинга 10, регистр ISR будет считан в переменную `isr` (листинг 10, строка 388). Любое дальнейшее демультиплексирование прерывания – это работа обработчика прерываний. Если принять прерывания `TxOK`, `TxErr` или `RxErr`, то потребуется обновить статистику. Прием прерывания `RxOK` означает, что мы приняли кадр успешно и драйвер обработал его (листинг 10, строка 437). Мы будем читать из приемного буфера до тех пор, пока не прочитаем все данные (эту работу выполняет цикл `while` в строке 440). Сначала проверим, не вышло ли значение `tp->cur_rx` за длину `RX_BUF_LEN` (листинг 10, строка 447). Если это так, мы выставляем значение `wgap`. Первые два байта указывают статус кадра, а следующие два байта указывают длину кадра (длина также включает

первые 4 байта)[4]. Эти значения всегда в обратном порядке и их следует преобразовать. Затем для принятого пакета выделяем место для skb (листинг 10, строка 462), копируем содержимое кадров в skb (листинг 10, строка 468) и ставим skb в очередь для дальнейшей обработки (листинг 10, строка 471). Затем обновляем CAPR (Current Address of Packet Read – текущий адрес чтения пакета) с тем, чтобы устройство RTL8169 знало о месте следующей операции записи. Обратите внимание, что этот обработчик прерываний уже зарегистрирован в функции rtl8169_open.

Таким образом, заглушки заменены полноценными функциями драйвера (Рисунок 2).



```
Terminal
File Edit View Search Terminal Help

~
sam@spb$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:1a:92:04:a1:ab
          inet addr:192.168.124.2  Bcast:192.168.124.255  Mask:255.255.255.0
          inet6 addr: fe80::21a:92ff:fe04:a1ab/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5053332 errors:0 dropped:0 overruns:0 frame:0
          TX packets:641442 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:6610391202 (6.6 GB)  TX bytes:487102822 (487.1 MB)

~
sam@spb$ ethtool -i eth0
driver: r8169
version: 2.3LK-NAPI
firmware-version:
bus-info: 0000:02:00.0
supports-statistics: yes
supports-test: no
supports-eeprom-access: no
supports-register-dump: yes
supports-priv-flags: no

~
sam@spb$
```

Рис. 2: Работающий сетевой интерфейс

Полученный результат во многом совпадает с кодом, находящимся в ядре Linux, но проигрывает только по количеству проверок на ошибки. Кроме того, использует значительно меньше макросов для обеспечения наглядности.

Заключение

В данной работе были рассмотрены теоретические и практические вопросы разработки драйверов для устройств ядра Linux.

Создание драйвера устройств для Linux является достаточно простой задачей так как она сводится к написанию новой функции и определению ее в системе. Тем самым, когда доступно устройство, присущее драйверу, система вызывает вашу функцию.

Однако, необходимо помнить, что драйвер устройства является частью ядра. Это означает, что драйвер запускается на уровне ядра и обладает большими возможностями: записать в любую область памяти, изменить или даже полностью удалить данные, и даже повредить физические устройства.

Спецификация PCI определяет большое количество типов передачи данных и структур. Все алгоритмы реализованы в ядре linux, а программисту драйверов предоставляется удобный и простой интерфейс в виде набора функций, макросов, структур.

Список литературы

- [1] Роберт Лав: «Разработка ядра Linux», Вильямс, 448 стр., 2008, ISBN 5-8459-1085-1, 0-672-32720-1.
- [2] Harvey G. Cragon: «Computer Architecture and Implementation», Cambridge University Press, 238 pages, 2000, ISBN-10: 521651689.
- [3] Rami Rosen: «Linux Kernel Networking: Implementation and Theory», Apress, 650 pages, 2014, ISBN-13: 978-1-4302-6196-4.
- [4] Realtech: RTL8111B, Single-Chip Gigabit LOM Ethernet Controller for PCI Express. Datasheet Rev. 1.4, 02 December 2005, Track ID: JATR-1076-21.