Санкт-Петербургский политехнический университет Петра Великого Институт Информационных Технологий и Управления

Кафедра компьютерных систем и программных технологий

Отчёт по практической работе № 6

по предмету «Проектирование ОС и компонентов»

### Написание драйвера сетевой карты

Работу выполнил студент гр. 63501/3 Мартынов С. А. Работу принял преподаватель Душутина Е. В.

Санкт-Петербург 2016

**Содержание**

[Постановка задачи 3](#_TOC_250011)

[Введение 4](#_TOC_250010)

1. [Сетевые интерфейсы в Linux 5](#_TOC_250009)
   1. [Структура net\_device 5](#_TOC_250008)
   2. [Организация доступа системы к устройству 7](#_TOC_250007)
   3. [Конфигурационное адресное пространство PCI 8](#_TOC_250006)
2. [Реализация драйвера 11](#_TOC_250005)
   1. [Обнаружение и включение устройства 11](#_TOC_250004)
   2. [Инициализация устройства 13](#_TOC_250003)
   3. [Реализация функций приёма и передачи 19](#_TOC_250002)

[Заключение 33](#_TOC_250001)

[Список литературы 34](#_TOC_250000)

# 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 r8169, для которой открыта спецификация. Ссылка на неё приводится в списке использованных материалов.

# Сетевые интерфейсы в Linux

## Структура net\_device

В Linux сетевые устройства рассматриваются как интерфейсы в сетевом стеке. Для этого ис- пользуется структура net\_device. Ниже перечисляются некоторые важные поля структуры net\_device, которая будут 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 вызы- вается 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 вызы- вается 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].

## 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 ис- пользуйте 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. компи- лятору 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 установ- лены 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 обес- печивается при помощи добавления \_p в конце команды. Имеются также эквиваленты команды memcpy. Функции ins и out копируют байты, слова и длинные слова в заданный порт и из него.

## Конфигурационное адресное пространство PCI

Одним из главных усовершенствований шины PCI по сравнению с другими архитектурами ввода-вывода стал её конфигурационный механизм, обладающий конфигурационным адрес- ным 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.

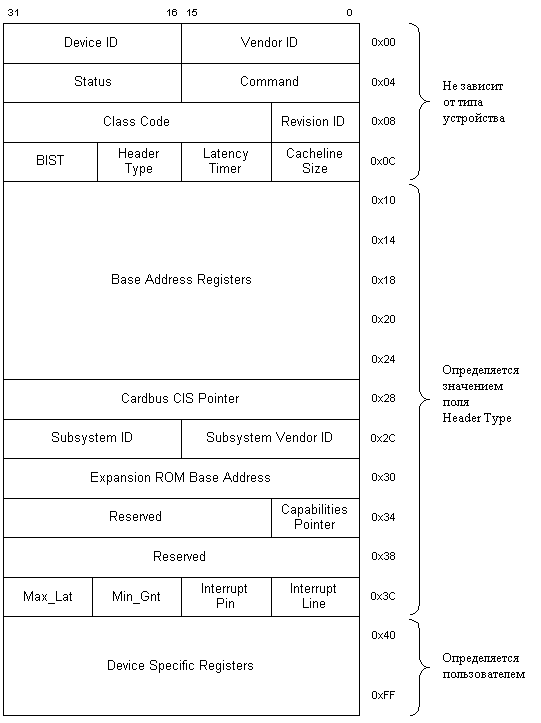


Рис. 1: Конфигурационное адресное пространство

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.

# Реализация драйвера

## 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)

#i nc l ude <l i nux / k e rne l . h>

#i nc l ude <l i nux / module . h>

#i nc l ude <l i nux / s tdde f . h>

#i nc l ude <l i nux / pc i . h>

s t a t i c s t r u c t pci\_dev *\**

{

s t r u c t pci\_dev *\** pdev

/*\** Ensure we are not i f ( ! pci\_ present ( ) )

{

probe\_ for\_ realtek 8169 ( void )

= NULL;

working on a non*−*PCI

system

*\**/

LOG\_MSG( "<1>pc i not pre s e nt \n" ) ;

re turn pdev ;

}

#d e f i n e REALTEK\_VENDER\_ID 0x10EC

#d e f i n e REALTEK\_DEVICE\_ID 0 X8169

/*\** Look f o r RealTek 8169 NIC *\**/

pdev = pci\_ f ind\_ device (REALTEK\_VENDER\_ID,

;

i f ( pdev )

{

/*\** de v i c e found , enable i t *\**/ i f ( pci\_ enable\_ device ( pdev ) )

{

REALTEK\_DEVICE\_ID,

NULL)

LOG\_MSG( " Could not enable the de v i c e \n" ) ;

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

|  |  |  |
| --- | --- | --- |
| 27  28  29  30  31  32  33  34  35 |  | re turn N  }  e l s e  LOG\_MSG(  }  e l s e  {  LOG\_MSG( " d  re turn pde |
| 36 |  | } |
| 37 |  | re turn pdev ; |
| 38 | } |  |
| 39  40  41  42  43  44  45  46  47  48 | i n t init\_modul  {  s t r u c t pci\_d pdev = probe i f ( ! pdev )  re turn 0 ;  re turn 0 ;  } | |

Каждый производитель имеет уникальный, назначенный только ему идентификатор ID и назначает уникальный идентификатор ID каждому конкретному виду устройств. Макросы REALTEK\_VENDER\_ID и REALTEK\_DEVICE\_ID (Листинг 1, строки 16 и 17) опре- деляют эти идентификаторы ID. Найти эти значения можно в "PCI Configuration Space Table"в спецификациях RealTek8169[4].

ULL;

" Device enabled \n" ) ;

e v i c e not found \n" ) ;

v ;

e ( void )

ev *\** pdev ;

\_ for\_ realtek 8169 ( ) ;

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.

## 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)

s t r u c t rtl 8 1 6 9 \_ pri vate

{

s t r u c t pci\_dev *\** pci\_dev ; /*\** PCI de v i c e *\**/

void *\**mmio\_addr ; /*\** memory mapped I /O addr *\**/

unsigned long regs\_ l en ; /*\** l e ng th o f I /O or MMI/O re g i o n *\**/

} ;

40

41

42

43

44

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)

#d e f i n e DRIVER " r t l 8 1 6 9 "

s t a t i c s t r u c t net\_device *\** rtl 8169 \_dev ;

s t a t i c i n t r t l 8 1 6 9 \_ i n i t ( s t r u c t pci\_dev *\** pdev , s t r u c t net\_device *\*\**

dev\_out )

{

s t r u c t net\_device *\** dev ;

s t r u c t rtl 8 1 6 9 \_ pri vate *\** tp ;

/*\**

* al l o c \_ e the rde v a l l o c a t e s memory f o r dev and dev*−*>p r i v .
* dev*−*>p r i v s h a l l have s i z e o f ( s t r u c t rtl 8 1 6 9 \_ pri vate ) memory
* a l l o c a te d .

*\**/

dev = al l o c \_ e the rde v ( s i z e o f ( s t r u c t rtl 8 1 6 9 \_ pri vate ) ) ; i f ( ! dev )

{

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63 LOG\_MSG( " Could not a l l o c a t e etherdev \n" ) ;

64 re turn *−*1;

65 }

66

67 tp = dev*−*>p r i v ;

68 tp*−*>pci\_dev = pdev ;

69 *\** dev\_out = dev ;

70

71 re turn 0 ;

72 }

73

74 i n t init\_module ( void )

75 {

76 s t r u c t pci\_dev *\** pdev ;

77 unsigned long mmio\_start , mmio\_end , mmio\_len , mmio\_flags ;

78 void *\** ioaddr ;

79 s t r u c t rtl 8 1 6 9 \_ pri vate *\** tp ;

80 i n t i ;

81

82 pdev = probe\_ for\_ realtek 8169 ( ) ;

83 i f ( ! pdev )

84 re turn 0 ;

85

86 i f ( r t l 8 1 6 9 \_ i n i t ( pdev , &rtl 8169 \_dev ) )

87 {

88 LOG\_MSG( " Could not i n i t i a l i z e de v i c e \n" ) ;

89 re turn 0 ;

90 }

91

92 tp = rtl 8169 \_dev *−*>p r i v ; /*\** r t l 8 1 6 9 p r i v ate i nf o rmati o n *\**/

93

94 /*\** get PCI memory mapped I /O space base address from BAR1 *\**/

95 mmio\_start = pci\_ resource\_ s tart ( pdev , 1 ) ;

96 mmio\_end = pci\_resource\_end ( pdev , 1 ) ;

97 mmio\_len = pci\_ resource\_ l en ( pdev , 1 ) ;

98 mmio\_flags = pc i \_ re s o urc e \_ f l ag s ( pdev , 1 ) ;

99

|  |  |  |
| --- | --- | --- |
| 100 | /*\** make sure above re g i o n i s MMI/O *\**/ |  |
| 101 | i f ( ! ( mmio\_flags & I / ORESOURCE\_MEM) ) |
| 102 | { |
| 103 | LOG\_MSG( " re g i o n not MMI/O re g i o n \n" ) ; |
| 104 | goto cleanup 1 ; |
| 105 | } |
| 106 |  |
| 107 | /*\** get PCI memory space *\**/ |
| 108 | i f ( pci\_ request\_ regions ( pdev , DRIVER) ) |
| 109 | { |
| 110 | LOG\_MSG( " Could not get PCI re g i o n \n" ) ; |
| 111 | goto cleanup 1 ; |
| 112 | } |
| 113 |  |
| 114 | pci\_set\_master ( pdev ) ; |
| 115 |  |
| 116 | /*\** ioremap MMI/O re g i o n *\**/ |
| 117 | ioaddr = ioremap ( mmio\_start , mmio\_len ) ; |
| 118 | i f ( ! ioaddr ) |
| 119 | { |
| 120 | LOG\_MSG( " Could not ioremap\n" ) ; |
| 121 | goto cleanup 2 ; |
| 122 | } |
| 123 |  |
| 124 | rtl 8169 \_dev *−*>base\_addr = ( long ) ioaddr ; |
| 125 | tp*−*>mmio\_addr = ioaddr ; |
| 126 | tp*−*>r egs\_ l en = mmio\_len ; |
| 127 |  |
| 128 | /*\** UPDATE NET\_DEVICE *\**/ |
| 129 |  |
| 130 | f o r ( i = 0 ; i < 6 ; i ++) |
| 131 | { /*\** Hardware Address *\**/ |
| 132 | rtl 8169 \_dev *−*>dev\_addr [ i ] = readb ( rtl 8169 \_dev *−*>base\_addr | + i ) ; |
| 133  134  135  136 | rtl 8169 \_dev *−*>b roadcast [ i ] = 0 x f f ;  }  rtl 8169 \_dev *−*>hard\_header\_len = 1 4 ; | |

|  |  |  |
| --- | --- | --- |
| 137  138  139  140  141  142  143  144  145  146  147  148 |  | memcpy( rtl  *\**/  rtl 8169 \_d  rtl 8169 \_d  rtl 8169 \_d  rtl 8169 \_d  rtl 8169 \_d  /*\** r e g i s t i f ( re g i s  {  LOG\_MSG  goto cl |
| 149 |  | } |
| 150 |  |  |
| 151 |  | re turn 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.

8169 \_dev *−*>name , DRIVER, s i z e o f (DRIVER) ) ; /*\** Device Name

ev *−*>i r q = pdev*−*>i r q ; /*\** I nte r rupt Number *\**/

ev *−*>open = rtl 8169 \_open ; ev *−*>s top = rtl 8169 \_ stop ;

ev *−*>hard\_start\_xmit = rtl 8169 \_ start\_ xmit ; ev *−*>g et\_ s tats = rtl 8169 \_ get\_ s tats ;

e r the de v i c e *\**/

te r\_ ne tde v ( rtl 8169 \_dev ) )

( " Could not r e g i s t e r ne tde v i c e \n" ) ;

eanup 0 ;

Далее необходимо получить поле base\_addr для net\_device. Оно указывает на начало памя- ти регистров устройства, которые потребуются для реализации ввода-вывода с отображе- нием 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. Второй аргумент в этих API – номер BAR. Согласно специфи- кации 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).

1. 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 | s t a t i c i n t rtl 8169 \_open ( s t r u c t net\_device *\** dev ) |  | |
| 155 | { |
| 156 | LOG\_MSG( " rtl 8169 \_open i s c a l l e d \n" ) ; |
| 157 | re turn 0 ; |
| 158 | } |
| 159 |  |
| 160 | s t a t i c i n t rtl 8169 \_ stop ( s t r u c t net\_device *\** dev ) |
| 161 | { |
| 162 | LOG\_MSG( " rtl 8169 \_open i s c a l l e d \n" ) ; |
| 163 | re turn 0 ; |
| 164 | } |
| 165 |  |
| 166 | s t a t i c i n t rtl 8169 \_ start\_ xmit ( s t r u c t sk\_buff *\** skb , | s t r u c t | net\_device |
|  | *\** dev ) |  |  |
| 167 | { |  |  |
| 168 | LOG\_MSG( " rtl 8169 \_ start\_ xmit i s c a l l e d \n" ) ; |  |  |
| 169  170  171  172  173 | re turn 0 ;  }  s t a t i c s t r u c t net\_ device\_ stats *\** rtl 8169 \_ get\_ s tats ( s t r u c t net\_device  *\** dev )  { | | |

|  |  |  |
| --- | --- | --- |
| 174 |  | LOG\_MSG( " rtl 8169 \_ get\_ s tats i s c a l l e d \n" ) ; |
| 175 |  | re turn 0 ; |
| 176 | } |  |

В init\_module пропущена обработка ошибок. В самом простом случае, она может выглядит так, как показано в листинге 5.

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

void cleanup\_module ( void )

{

s t r u c t rtl 8 1 6 9 \_ pri vate *\** tp ; tp = rtl 8169 \_dev *−*>p r i v ;

iounmap ( tp*−*>mmio\_addr ) ;

pc i \_ re l e as e \_ re g i o ns ( tp*−*>pci\_dev ) ;

unregister\_ netdev ( rtl 8169 \_dev ) ; pci\_ disable\_ device ( tp*−*>pci\_dev ) ; re turn ;

}

178

179

180

181

182

183

184

185

186

187

188

189

На данный момент драйвер может включать устройство и отображать его в пространство пользователя. Можно откомпилировать его код и вставьте 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

## 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 по- рядке 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 "на- чальный 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, перепишет 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, управ- ление которой осуществляется как кольцевой памятью). Всякий раз, когда устройство принимает пакет, содержимое пакета запоминается в память кольцевого буфера и изменя- ется 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)

#d e f i n e NUM\_TX\_DESC 4 s t r u c t rtl 8 1 6 9 \_ pri vate

{

s t r u c t pci\_dev *\** pci\_dev ; /*\** PCI de v i c e *\**/

void *\**mmio\_addr ; /*\** memory mapped I /O addr *\**/

unsigned long regs\_ l en ; /*\** l e ng th o f I /O or MMI/O re g i o n *\**/ unsigned i n t tx\_ f lag ;

unsigned i n t cur\_tx ;

93

94

95

96

97

98

99

100

|  |  |  |
| --- | --- | --- |
| 101 |  | unsigned i n t dirty\_ tx ; |
| 102 |  | unsigned char *\** tx\_buf [NUM\_TX\_DESC] ; /*\** Tx bounce b u f f e r s *\**/ |
| 103 |  | unsigned char *\** tx\_bufs ; /*\** Tx bounce b u f f e r re g i o n . *\**/ |
| 104 |  | dma\_addr\_t tx\_bufs\_dma ; |
| 105 |  |  |
| 106 |  | s t r u c t net\_ device\_ stats s t a t s ; |
| 107 |  | unsigned char *\** rx\_ring ; |
| 108 |  | dma\_addr\_t rx\_ring\_dma ; |
| 109 |  | unsigned i n t cur\_rx ; |
| 110 | } ; |  |

Компонента tx\_flag (листинг 6, строка 99) должна содержать флаги передачи, уведомляю- щие 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 "дескрип- торов 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. Устройству 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  7  8  9  10  11  12  13  14  15  16 | #d e f i n e TX\_BUF\_SIZE 1536  #d e f i n e TOTAL\_TX\_BUF\_SIZE  /*\** S i z e o f the in*−*memory r  #d e f i n e RX\_BUF\_LEN\_IDX 2  #d e f i n e RX\_BUF\_LEN ( 81  #d e f i n e RX\_BUF\_PAD 16  0 x44 *\**/  #d e f i n e RX\_BUF\_WRAP\_PAD 20  *\**/  #d e f i n e RX\_BUF\_TOT\_LEN (R  /*\** 8169 r e g i s t e r o f f s e t s *\** | | |
| 17 | #d e f i n e | TSD0 | 0 x10 |
| 18 | #d e f i n e | TSAD0 | 0 x20 |
| 19 | #d e f i n e | RBSTART | 0 x30 |
| 20 | #d e f i n e | CR | 0 x37 |
| 21 | #d e f i n e | CAPR | 0 x38 |
| 22 | #d e f i n e | IMR | 0 x3c |
| 23 | #d e f i n e | ISR | 0 x3e |
| 24 | #d e f i n e | TCR | 0 x40 |
| 25 | #d e f i n e | RCR | 0 x44 |
| 26 | #d e f i n e | MPC | 0 x4c |
| 27 | #d e f i n e | MULINT | 0 x5c |
| 28 |  |  |  |
| 29  30  31  32  33  34  35  36  37  38  39 | /*\** TSD r e g i s t e r commands *\**  #d e f i n e TxHostOwns 0 x20  #d e f i n e TxUnderrun 0 x40  #d e f i n e TxStatOK 0 x80  #d e f i n e TxOutOfWindow 0 x20  #d e f i n e TxAborted 0 x40  #d e f i n e Tx Carrier Lost 0 x80  /*\** CR r e g i s t e r commands *\**/  #d e f i n e RxBufEmpty 0 x01  #d e f i n e CmdTxEnb 0 x04 | | |

/*\** should be at l e a s t MTU + 14 + 4 *\**/ (TX\_BUF\_SIZE *\** NUM\_TX\_SIZE)

e c e i v e r i ng . *\**/

/*\** 0==8K, 1==16K, 2==32K, 3==64K *\**/ 92 << RX\_BUF\_LEN\_IDX)

/*\** s e e 11 th and 12 th b i t o f RCR:

48 /*\** spare padding to handle pkt wrap

X\_BUF\_LEN + RX\_BUF\_PAD + RX\_BUF\_WRAP\_PAD)

/

/ 00

00

00

000000

000000

000000

|  |  |  |
| --- | --- | --- |
| 40 | #d e f i n e CmdRxEnb | 0 x08 |
| 41 | #d e f i n e CmdReset | 0 x10 |
| 42 |  |  |
| 43 | /*\** ISR Bi ts *\**/ |  |
| 44  45  46  47  48  49  50 | #d e f i n e RxOK 0 x01  #d e f i n e RxErr 0 x02  #d e f i n e TxOK 0 x04  #d e f i n e TxErr 0 x08  #d e f i n e RxOverFlow 0 x10  #d e f i n e RxUnderrun 0 x20  #d e f i n e RxFIFOOver 0 x40 | |

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)

IZE,

|  |  |  |
| --- | --- | --- |
| 232 | s t a t i c i n t rtl 8169 \_open ( s t r u c t net\_device *\** dev ) |  |
| 233 | { |
| 234 | i n t r e tv a l ; |
| 235 | s t r u c t rtl 8 1 6 9 \_ pri vate *\** tp = dev*−*>p r i v ; |
| 236 |  |
| 237 | /*\** get the IRQ |
| 238 | *\** second arg i s i n te r r u p t handler |
| 239 | *\** th i rd i s f l a g s , 0 means no IRQ s hari ng |
| 240 | *\**/ |
| 241 | r e tv a l = request\_ i rq ( dev*−*>i rq , rtl 8 1 6 9 \_ i nte rrupt , 0 , dev ) ; | dev*−*>name , |
| 242 | i f ( r e tv a l ) |  |
| 243 | re turn r e tv a l ; |  |
| 244  245  246  247  248  249  250  251  252 | /*\** get memory f o r Tx b u f f e r s  *\** memory must be DMAable  *\**/  tp*−*>tx\_buf s = pc i \_ a l l o c \_ c o ns i s te nt ( tp*−*>pci\_dev , TOTAL\_TX\_BUF\_S &tp*−*>tx\_bufs\_dma ) ;  tp*−*>rx \_ring = pc i \_ a l l o c \_ c o ns i s te nt ( tp*−*>pci\_dev , RX\_BUF\_TOT\_LEN, &tp*−*>rx\_ring\_dma ) ; | |

|  |  |  |  |
| --- | --- | --- | --- |
| 253 | i f ( ( ! tp*−*>tx\_buf s ) | | ( ! tp*−*>rx \_ring ) ) | |  |
| 254 | { | |
| 255 | f re e \_ i rq ( dev*−*>i rq , dev ) ; | |
| 256 |  | |
| 257 | i f ( tp*−*>tx\_buf s ) | |
| 258 | { | |
| 259 | pc i \_ f re e \_ c o ns i s te nt ( tp*−*>pci\_dev , | | TOTAL\_TX\_BUF\_SIZE, tp*−*> |
|  | tx\_bufs , | |  |
| 260 | tp*−*>tx\_bufs\_dma ) ; | |  |
| 261  262  263  264  265  266 | tp*−*>tx\_buf s = NULL;  }  i f ( tp*−*>rx \_ring )  {  pc i \_ f re e \_ c o ns i s te nt ( tp*−*>pci\_dev , RX\_BUF\_TOT\_LEN, tp*−*>rx\_ tp*−*>rx\_ring\_dma ) ; | | |
| 267 |  | tp*−*>rx \_ring = NULL; | |
| 268 |  | } | |
| 269 |  | re turn *−*ENOMEM; | |
| 270 | } |  | |
| 271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288 | tp*−*>t x\_ f lag = 0 ;  rtl 8 1 6 9 \_ i ni t\_ r i ng ( dev ) ; rtl 8169 \_hw\_start ( dev ) ;  re turn 0 ;  }  s t a t i c void rtl 8 1 6 9 \_ i ni t\_ r i ng ( s t r u c t net\_device *\** dev )  {  s t r u c t rtl 8 1 6 9 \_ pri vate *\** tp = dev*−*>p r i v ; i n t i ;  tp*−*>cur\_tx = 0 ; tp*−*>d i rty\_ tx = 0 ;  f o r ( i = 0 ; i < NUM\_TX\_DESC; i ++)  tp*−*>tx\_buf [ i ] = &tp*−*>tx\_buf s [ i *\** TX\_BUF\_SIZE ] ; | | |

ring ,

|  |  |  |
| --- | --- | --- |
| 289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309 | re turn ;  }  s t a t i c void rtl 8169 \_hw\_start ( s t r u c t net\_device *\** dev )  {  s t r u c t rtl 8 1 6 9 \_ pri vate *\** tp = dev*−*>p r i v ; void *\** ioaddr = tp*−*>mmio\_addr ;  u32 i ;  rtl 8169 \_ chip\_ reset ( ioaddr ) ;  /*\** Must enable Tx/Rx be f o re s e t t i n g t r a n s f e r th re s h o l d s ! writeb (CmdTxEnb | CmdRxEnb, ioaddr + CR) ;  /*\** tx c o n f i g *\**/  w r i t e l ( 0 x00000600 , ioaddr + TCR) ; /*\** DMA burst s i z e 1024  /*\** rx c o n f i g *\**/  w r i t e l ( ( ( 1 << 12 ) | ( 7 << 8 ) | ( 1 << 7 ) | ( 1 << 3 ) | ( 1 <  << 1 ) ) ,  ioaddr + RCR) ; | |
| 310 |  |  |
| 311 | /*\** i n i t Tx b u f f e r DMA addre s s e s *\**/ |  |
| 312 | f o r ( i = 0 ; i < NUM\_TX\_DESC; i ++) |  |
| 313 | { |  |
| 314 | w r i t e l ( tp*−*>tx\_bufs\_dma + ( tp*−*>tx\_buf [ i ] | *−* tp*−*>tx\_buf s ) , |
| 315 | ioaddr + TSAD0 + ( i *\** 4 ) ) ; |  |
| 316 | } |  |
| 317 |  |  |
| 318 | /*\** i n i t RBSTART *\**/ |  |
| 319 | w r i t e l ( tp*−*>rx\_ring\_dma , ioaddr + RBSTART) ; |  |
| 320 |  |  |
| 321 | /*\** i n i t i a l i z e missed packet counter *\**/ |  |
| 322 | w r i t e l ( 0 , ioaddr + MPC) ; |  |
| 323 |  |  |
| 324 | /*\** no early *−*rx i n te r r u p t s *\**/ |  |

*\**/

*\**/

< 2 ) | ( 1

325

writew ( ( readw ( ioaddr + MULINT) & 0 xF000 ) , ioaddr + MULINT) ;

/*\** Enable a l l known i n te r r u p t s by s e t t i n g the i n te r r u p t

writew (INT\_MASK, ioaddr + IMR) ;

mask .

*\**/

netif \_ start\_ queue ( dev ) ;

re turn ;

}

s t a t i c void rtl 8169 \_ chip\_ reset ( void *\** ioaddr )

{

i n t i ;

/*\** S o f t r e s e t the chip . *\**/

writeb ( CmdReset , ioaddr + CR) ;

/*\** Check that the chip has f i n i s h e d the r e s e t . *\**/

f o r ( i = 1000 ; i > 0 ; i *−−*)

{

b a r r i e r ( ) ;

i f ( ( readb ( ioaddr + CR) & CmdReset) == 0 ) break ;

udelay ( 10 ) ;

}

re turn ;

}

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340

341

342

343

344

345

346

347

348

349

350

Функция rtl8169\_open (листинг 8, строка 232) начинается с запроса IRQ, что делается вы- зовом 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 исхо- дящие пакеты, прежде чем они будут переданы в сеть. Вызов API pci\_allocate\_consistant возвращает виртуальный адрес ядра (листинг 8, строка 248). Физический адрес возвраща- ется 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. Сле- дующая функция, barrier(), вызывается с тем, чтобы ядро выделило память, требуемую для ввода-выхода, без какой-либо оптимизации (листинг 8, строка 344). Поскольку устрой- ство 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 – конфигу- рационный 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 потребуется 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)

s t a t i c i n t rtl 8169 \_ start\_ xmit ( s t r u c t sk\_buff *\** skb , s t r u c t net\_device

*\** dev )

{

s t r u c t rtl 8 1 6 9 \_ pri vate *\** tp = dev*−*>p r i v ; void *\** ioaddr = tp*−*>mmio\_addr ;

unsigned i n t entry = tp*−*>cur\_tx ; unsigned i n t l e n = skb*−*>l e n ;

#d e f i n e ETH\_MIN\_LEN 60 /*\** minimum Ethernet frame s i z e *\**/ i f ( l e n < TX\_BUF\_SIZE)

{

i f ( l e n < ETH\_MIN\_LEN)

memset( tp*−*>tx\_buf [ entry ] , 0 , ETH\_MIN\_LEN) ;

352

353

354

355

356

357

358

359

360

361

362

|  |  |  |
| --- | --- | --- |
| 363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378 |  | skb\_cop dev\_kfr  }  e l s e  {  dev\_kfr re turn  }  w r i t e l ( tp  ioadd entry++; tp*−*>cur\_t  i f ( tp*−*>c  {  netif\_st |
| 379 |  | } |
| 380 |  | re turn 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 мень- шей 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 име- ющиеся 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.

y\_and\_csum\_dev ( skb , tp*−*>tx\_buf [ entry ] ) ;

ee\_skb ( skb ) ;

ee\_skb ( skb ) ;

0 ;

*−*>t x\_ f lag | max( len , ( unsigned i n t ) ETH\_MIN\_LEN) ,

r + TSD0 + ( entry *\** s i z e o f ( u32 ) ) ) ;

x = entry % NUM\_TX\_DESC;

ur\_tx == tp*−*>d i rty\_ tx )

op\_queue ( dev ) ;

Теперь устройство готово отсылать пакеты. Для приёма пакетов, вернёмся к листингу

1. 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 премного 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 s t a t i c void rtl 8 1 6 9 \_ i nte rrupt ( i n t i rq , void *\** dev\_instance , s t r u c t pt\_regs *\** re g s )

384 {

385 s t r u c t net\_device *\** dev = ( s t r u c t net\_device *\** ) dev\_instance ;

386 s t r u c t rtl 8 1 6 9 \_ pri vate *\** tp = dev*−*>p r i v ;

387 void *\** ioaddr = tp*−*>mmio\_addr ;

388 unsigned s ho rt i s r = readw ( ioaddr + ISR ) ;

389

390 /*\** c l e a r a l l i n te r ru p t .

391 *\** Specs says reading ISR c l e a r s a l l i n te r r u p t s and w r i t i ng

392 *\** has no e f f e c t . But t h i s does not seem to be case . I keep on

393 *\** g e tti n g i n te r ru p t u n l e s s I f o r c i b l y c l e a r s a l l i n te r r u p t : *−*(

394 *\**/

395 writew ( 0 x f f f f , ioaddr + ISR ) ;

396

397 i f ( ( i s r & TxOK) | | ( i s r & TxErr ) )

398 {

399 while ( ( tp*−*>d i rty\_ tx != tp*−*>cur\_tx ) | | netif\_queue\_stopped ( dev ) )

400 {

nsigned i n t tx s ta tu s = re ad l (

|  |  |  |  |
| --- | --- | --- | --- |
| 401  402  403  404  405  406  407  408  409  410  411  412  413  414  415  416  417  418  419  420  421  422  423  424  425 |  |  | u  i  i  {  }  e  {  }  t t  i  { |
| 426 |  |  | } |
| 427 |  | } |  |
| 428 | } |  |  |
| 429  430  431  432  433  434  435  436  437 | i f  {  }  i f | ( i  /*\** LOG  tp*−*  ( i | |

ioaddr + TSD0 + tp*−*>d i rty\_ tx *\** s i z e o f ( i n t ) ) ;

f ( ! ( tx s ta tu s & ( TxStatOK | TxAborted | TxUnderrun ) ) ) break ; /*\** yet not transmitted *\**/

f ( tx s ta tu s & TxStatOK)

LOG\_MSG( " Transmit OK i n te r ru p t \n" ) ;

tp*−*>s t a t s . tx\_bytes += ( tx s ta tu s & 0 x 1 f f f ) ; tp*−*>s t a t s . tx\_packets++;

l s e

LOG\_MSG( " Transmit Error i n te r ru p t \n" ) ; tp*−*>s t a t s . tx\_ errors++;

p*−*>d i rty\_ tx++;

p*−*>d i rty\_ tx = tp*−*>d i rty\_ tx % NUM\_TX\_DESC;

f ( ( tp*−*>d i rty\_ tx == tp*−*>cur\_tx ) & netif\_queue\_stopped ( dev ) )

LOG\_MSG( " waking up queue \n" ) ; netif\_wake\_queue ( dev ) ;

s r & RxErr )

TODO: Need d e ta i l e d a n a l y s i s o f e r r o r s ta tu s *\**/

\_MSG( " r e c e i v e e r r i n te r ru p t \n" ) ;

>s t a t s . rx\_ errors++;

s r & RxOK)

438 {

439 LOG\_MSG( " r e c e i v e i n te r ru p t re c e i v e d \n" ) ;

440 while ( ( readb ( ioaddr + CR) & RxBufEmpty) == 0 )

441 {

442 unsigned i n t rx\_ status ;

443 unsigned s ho rt rx\_ s i ze ;

444 unsigned s ho rt pkt\_ s i ze ;

445 s t r u c t sk\_buff *\** skb ;

446

447 i f ( 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 l i t t l e to host endian

451 *\** XXX: My CPU i s l i t t l e endian only : *−*)

452 *\**/

453 rx\_ status = *\** ( unsigned i n t *\** ) ( tp*−*>rx \_ring + tp*−*>cur\_rx ) ;

454 rx\_ s i ze = rx\_ status >> 1 6 ;

455

456 /*\** f i r s t two bytes are r e c e i v e s ta tu s r e g i s t e r

457 *\** and next two bytes are frame l e ng th

458 *\**/

459 pkt\_ s i ze = rx\_ s i ze *−* 4 ;

460

461 /*\** hand over packet to system *\**/

462 skb = dev\_alloc\_skb ( pkt\_ s i ze + 2 ) ;

463 i f ( skb )

464 {

465 skb*−*>dev = dev ;

466 skb\_ reserve ( skb , 2 ) ; /*\** 16 byte a l i g n the IP f i e l d s *\**/

467

468 eth\_copy\_and\_sum( skb , tp*−*>rx \_ring + tp*−*>cur\_rx + 4 , pkt\_size

,

469 0 ) ;

470

471 skb\_put ( skb , pkt\_ s i ze ) ;

472 skb*−*>p ro to c o l = eth\_type\_trans ( skb , dev ) ;

473 ne t i f \_ rx ( skb ) ;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 474 |  | | | |
| 475 |  | |  | dev*−*>l ast\_ rx = j i f f i e s ; |
| 476 |  | |  | tp*−*>s t a t s . rx\_bytes += pkt\_ s i ze ; |
| 477 |  | |  | tp*−*>s t a t s . rx\_packets++; |
| 478 |  | |  | } |
| 479 |  | |  | e l s e |
| 480 |  | |  | { |
| 481 |  | |  | LOG\_MSG( "Memory squeeze , dropping packet . \ n" ) ; |
| 482 |  | |  | tp*−*>s t a t s . rx\_dropped++; |
| 483 |  | |  | } |
| 484 |  | |  |  |
| 485 |  | |  | /*\** update tp*−*>cur\_rx to next w r i t i ng l o c a t i o n *\**/ |
| 486 |  | |  | tp*−*>cur\_rx = ( tp*−*>cur\_rx + rx\_ s i ze + 4 + 3 ) & ~ 3 ; |
| 487 |  | |  |  |
| 488 |  | |  | /*\** update CAPR *\**/ |
| 489 |  | |  | writew ( tp*−*>cur\_rx , ioaddr + CAPR) ; |
| 490 |  | | } |  |
| 491  492  493  494  495  496  497 | }  i f ( i s r & CableLen )  LOG\_MSG( " c abl e l e ng th change i n te r ru p t \n" ) ; i f ( i s r & TimeOut)  LOG\_MSG( " time i n te r ru p t \n" ) ; i f ( i s r & Sys Err ) | | | |
| 498 |  | LOG\_MSG( " system e r r i n te r r u p t \n" ) ; | | |
| 499 |  | re turn ; | | |
| 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 wrap. 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).

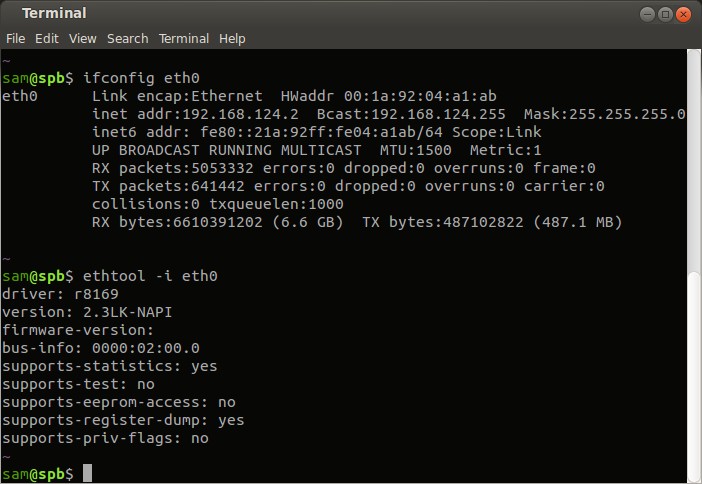


Рис. 2: Работающий сетевой интерфейс

Полученный результат во многом совпадает с кодом, находящимся в ядре Linux, но проиг- рывает 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 определении 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.