РЕАЛИЗАЦИЯ НИЗКОУРОВНЕВОЙ ПОДДЕРЖКИ ШИНЫ PCI В ЯДРЕ ОПЕРАЦИОННОЙ СИСТЕМЫ LINUX

В данной статье на примере решения простой задачи – определения MAC-адреса сетевой карты – рассмотрена реализация низкоуровневой поддержки (low-level support) шины PCI в ядре операционной системы Linux.

ВЛАДИМИР МЕШКОВ

Постановка задачи и исходные данные

Исходные данные – имеется компьютер, функционирующий под управлением ОС Linux, версия ядра 2.4.24. В PCI-слот установлен сетевой адаптер на чипсете RTL8139C (далее – адаптер RTL8139C).

Задача – определить MAC-адрес этого адаптера.

Путей решения этой задачи несколько. Можно воспользоваться командами dmesg или ifconfig:

```
root@bob~/# dmesg | grep eth0
eth0: RealTek RTL8139 at 0xc000, 00:02:44:72:5e:4e, IRQ 11
eth0: Identified 8139 chip type 'RTL-8100B/8139D'

root@bob~/# ifconfig | grep eth0
eth0      Link encap:Ethernet  HWaddr 00:02:44:72:5E:4E
```

Можно написать небольшое приложение следующего вида:

```
/* get_mac.c */
#include <stdio.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <linux/if.h>

int main() {

  int fd;
  struct ifreq ifr;
  unsigned char mac[6];

  fd=socket(AF_INET,SOCK_DGRAM,0);
  memset(&ifr,0,sizeof(struct ifreq));
  memcpy(ifr.ifr_name,"eth0",4);
  ioctl(fd,SIOCGIFHWADDR,&ifr);
  memcpy(mac,(char *)&(ifr.ifr_hwaddr.sa_data), ↵
    sizeof(struct sockaddr));

  printf("%.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
    mac[0],mac[1],mac[2],mac[3],mac[4],mac[5]);

  return 0;
}
```

Можно извлечь MAC-адрес из самого адаптера RTL8139C. Рассмотрим, как это делается.

Согласно спецификации на сетевой адаптер RTL8139C, MAC-адрес занимает первые 6 байт в пространстве портов ввода/вывода (I/O), отведенного адаптеру. Задавая смещение относительно базового порта I/O, можно прочитать все 6 байт MAC-адреса.

Пример функции, выполняющей процедуру чтения MAC-адреса, приведен ниже:

```
void get_mac_addr(u32 base_addr)
{
  u8 mac[6];

/*
 * Последовательно читаем байты из порта base_addr
 * и сохраняем их в массиве mac[]
 */
  for(int i = 0; i < 6; i++)
    mac[i] = inb(base_addr + i);

/*
 * Отображаем результат
 */
  printf("%02X:%02X:%02X:%02X:%02X:%02X\n",
    mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
}
```

Функция get_mac_addr() принимает в качестве параметра адрес порта I/O адаптера RTL8139C и в цикле производит считывание данных MAC-адреса из пространства I/O, выделенного адаптеру. При выходе из цикла в буфере mac[] будет находиться искомый MAC-адрес.

Теперь вся задача сводится к определению значения базового адреса порта I/O адаптера RTL8139C. Как найти этот адрес? Чтобы ответить на этот вопрос, давайте познакомимся поближе с шиной PCI.

Общая характеристика шины PCI

Разработка шины PCI началась весной 1991 года как внутренний проект корпорации Intel (Release 0.1). Специалисты компании поставили перед собой цель разработать недорогое решение, которое бы позволило полностью реализовать возможности нового поколения процессоров 486/Pentium/P6. Особенно подчеркивалось, что разработка проводилась «с нуля», а не была попыткой установки новых «заплат» на существующие решения. В результате шина PCI появилась в июне 1992 года (R1.0). Разработчики Intel отказались от использования шины процессора и ввели еще одну «антресольную» (mez-zanine) шину.

Благодаря такому решению шина получилась, во-первых, процессорно-независимой, а во-вторых, могла работать параллельно с шиной процессора, не обращаясь к ней за запросами и тем самым снижая её загрузку. Стандарт шины был объявлен открытым и передан PCI Special Interest Group (www.pcisig.com), которая продолжила работу по совершенствованию шины. В настоящее время действует спецификация PCI версии 2.3.

Основные возможности шины следующие:

- Синхронный 32- или 64-разрядный обмен данными. При этом для уменьшения числа контактов (и стоимости) используется мультиплексирование, то есть адрес и данные передаются по одним и тем же линиям.
- Поддержка 5V и 3.3V логики. Частота 66 МГц поддерживается только 3.3V логикой.
- Частота работы шины 33 МГц или 66 МГц позволяет обеспечить широкий диапазон пропускных способностей (с использованием пакетного режима):
  - 132 Мб/сек при 32-бит/33 МГц;
  - 264 Мб/сек при 32-бит/66 МГц;
  - 264 Мб/сек при 64-бит/33 МГц;
  - 528 Мб/сек при 64-бит/66 МГц.

  При этом для работы шины на частоте 66 МГц необходимо, чтобы все периферийные устройства работали на этой частоте.
- Полная поддержка multiply bus master (например, несколько контроллеров жестких дисков могут одновременно работать на шине).
- Автоматическое конфигурирование карт расширения при включении питания.
- Спецификация шины позволяет комбинировать до восьми функций на одной карте (например, видео + звук).
- Шина позволяет устанавливать до 4 слотов расширения, однако возможно использование моста PCI-PCI для увеличения количества карт расширения.
- PCI-устройства оборудованы таймером, который используется для определения максимального промежутка времени, в течение которого устройство может занимать шину.

Шина поддерживает метод передачи данных, называемый «linear burst» (метод линейных пакетов). Этот метод предполагает, что пакет информации считывается (или записывается) «одним куском», то есть адрес автоматически увеличивается для следующего байта, при этом увеличивается скорость передачи данных за счет уменьшения числа передаваемых адресов.

На одной шине PCI может присутствовать несколько устройств, каждое из которых имеет свой номер (device number). В системе может присутствовать несколько шин PCI, каждая из которых имеет свой номер (PCI bus number). Шины нумеруются последовательно; шина, подключённая к главному мосту, имеет нулевой номер.

В каждой транзакции (обмене по шине) участвуют два устройства – инициатор (initiator) обмена, он же мастер (master) или ведущее устройство, и целевое (target) устройство, оно же ведомое (slave). Шина PCI все транзакции трактует как пакетные: каждая транзакция начинается фазой адреса, за которой может следовать одна или несколько фаз данных.

Конфигурационное пространство устройства PCI

Согласно спецификации, каждое устройство PCI имеет конфигурационное пространство (configuration space) размером 256 байт, в котором содержится информация о самом устройстве и о ресурсах, занимаемых устройством. Это пространство не приписано ни к пространству памяти, ни к пространству ввода-вывода. Доступ к нему осуществляется по специальным циклам шины Configuration Read и Configuration Write. После аппаратного сброса (или по включении питания) устройства PCI доступны только для операций конфигурационного чтения и записи. В этих операциях устройства выбираются по индивидуальным сигналам IDSEL и сообщают о потребностях в ресурсах и возможных вариантах конфигурирования. После распределения ресурсов, выполняемого программой конфигурирования (во время теста POST), в конфигурационные регистры устройства записываются параметры конфигурирования. Только после этого к устройству становится возможным доступ по командам обращения к памяти и портам ввода-вывода. Для того чтобы всегда можно было найти работоспособную конфигурацию, все ресурсы, занимаемые картой, должны быть перемещаемыми в своих пространствах. Для многофункциональных устройств каждая функция должна иметь свое конфигурационное пространство.

Конфигурационное пространство, формат которого представлен на рис. 1, состоит из трех областей:

- область, не зависимая от устройства (device-independent header region);
- область, определяемая типом устройства (header-type region);
- область, определяемая пользователем (user-defined region).

Подробное описание полей конфигурационного пространства приведено в спецификации [4]. Рассмотрим кратко основные поля.

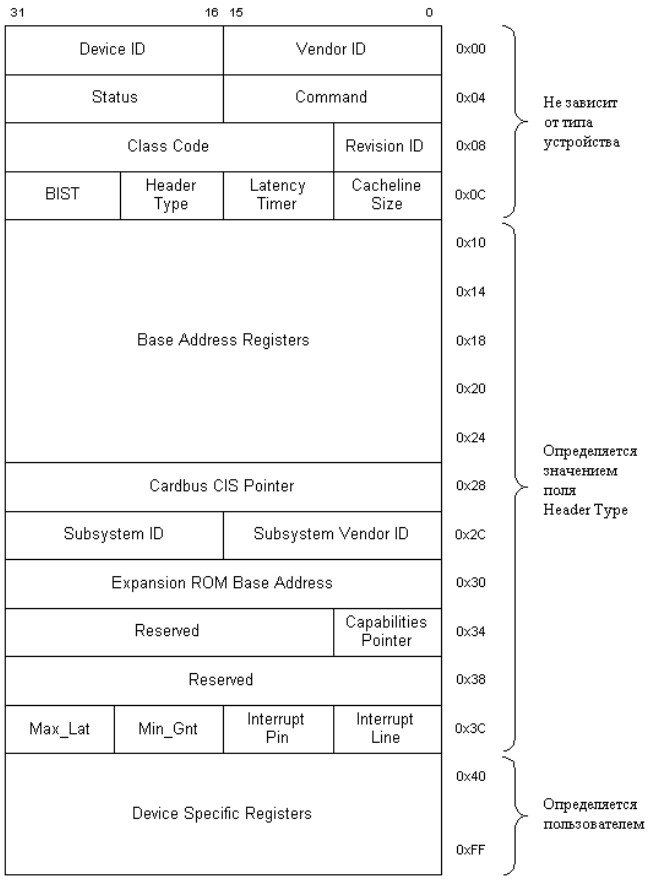

Рисунок 1. Формат конфигурационного пространства PCI

Vendor_ID, Device_ID, Class_Code

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.

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).

Header_Type

Поле Header_Type определяет формат header-type области, а также является ли устройство многофункциональным. Идентификатором многофункционального устройства является бит 7 поля: если бит установлен в 1 – устройство поддерживает несколько функций, если сброшен в 0 – устройство выполняет только функцию.

Биты 0 – 6 определяют собственно формат header-type области: если эти биты обнулены (содержат код 0x00), то формат области header-type соответствует формату, представленому на рис. 1; значение 0x01 идентифицирует устройство как мост PCI-to-PCI, и формат header-type области

описан в спецификации PCI-to-PCI Bridge Architecture Specification; значение 0x02 идентифицирует устройство как мост CardBus. Остальные значения зарезервированы.

Command

Командный регистр (поле Command) содержит средства управления устройством. Назначение отдельных бит этого регистра:

- бит 0 – определяет реакцию устройства на обращение к нему через пространство портов I/O. Если бит сброшен в 0, устройство игнорирует попытки доступа к нему через порты I/O;
- бит 1 – определяет реакцию устройства на обращение к нему через адресное пространство. Если бит установлен в 1, устройство отвечает на обращения к нему через адресное пространство; если бит сброшен в 0, то устройство на попытки доступа к нему не реагирует;
- бит 2 – установленный в 1 бит разрешает устройству работать в режиме Bus Master.

Base Address Registers

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.

Программный доступ к конфигурационному пространству PCI

Configuration Mechanism #1

Поскольку конфигурационное пространство не имеет привязки к какой-либо определенной области адресного пространства, для доступа к нему применяется специальный механизм, названый в спецификации Configuration Mechanism #1. Для работы этого механизма в пространстве портов I/O зарезервированы два 32-разрядных порта, входящих в главный мост: CONFIG_ADDRESS с адресом 0xCF8 и CONFIG_DATA с адресом 0xCFC. Формат CONFIG_ADDRESS представлен на рис. 2.

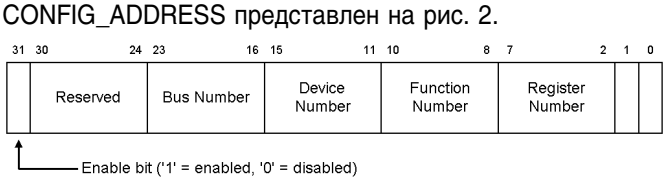

Рисунок 2. Формат регистра CONFIG_ADDRESS

Установленный в 1 бит 31 разрешает обращение к конфигурационному пространству через порт CONFIG_DATA, биты 30 – 24 зарезервированы (read-only), при чтении должны возвращать 0, биты 23 – 26 содержат номер шины, биты 15 – 11 – номер устройства, биты 10 – 8 – номер функции и биты 7 – 2 – номер регистра, к которому выполняется обращение (смещение в конфигурационном пространстве).

Порядок работы Configuration Mechanism #1 следующий – в порт CONFIG_ADDRESS (0xCF8) заносится адрес, соответствующий формату, приведенному на рис. 2; обращением к порту CONFIG_DATA (0xCFC) производится чтение или запись данных в требуемый регистр конфигурационного пространства.

PCI BIOS

Для взаимодействия с устройствами PCI имеются дополнительные функции BIOS, доступные как из реального, так и защищенного режима работы процессора. Эти функции предназначены для работы с конфигурационным пространством и генерации специальных циклов PCI.

Функции PCI BIOS для 16-битного реального режима вызываются через прерывание int 0x1A. Номер функции задается в регистре AX. Признаком нормального выполнения являются значения флага CF = 0 и ноль в регистре AH (AH = 0x00, SUCCESFUL). Если CF = 1, то регистр AH содержит код ошибки:

- 0x81 – неподдерживаемая функция (FUNC_NOT_SUPPORTED);
- 0x83 – неправильный идентификатор производителя (BAD_VENDOR_ID);
- 0x86 – устройство не найдено (DEVICE_NOT_FOUND);
- 0x87 – неправильный номер регистра PCI (BAD_REGISTER_NUMBER), т.е. неправильно задано смещение в конфигурационном пространстве.

Перечислим некоторые функции PCI BIOS (полный перечень содержится в [6]):

- 0xB101 – проверка присутствия PCI BIOS;
- 0xB102 – поиск устройства по коду фирмы-изготовителя;
- 0xB103 – поиск устройства по коду класса;
- 0xB108 – чтение байта конфигурационного пространства устройства PCI;
- 0xB109 – чтение слова конфигурационного пространства устройства PCI;
- 0xB10A – чтение двойного слова конфигурационного пространства устройства PCI.

При чтении информации из конфигурационного пространства в регистры процессора заносятся следующие значения:

- AX – номер функции;
- BH – номер шины, к которой подключено устройство (от 0 до 255);
- BL – номер устройства в старших 5 битах и номер функции в трех младших;
- DI – смещение в конфигурационном пространстве.

После этого следует вызов прерывания int 0x1A, в результате которого в регистрах процессора будут размещены следующие значения:

- ECX – считанная информация (байт/слово/двойное слово);
- AH – код возврата (SUCCESFUL/BAD_REGISTER_NUMBER);
- CF – статус возврата (0 – функция успешно выполнена, 1 – ошибка).

BIOS32

При работе в 32-разрядном защищенном режиме для доступа к функциям PCI BIOS используются средства BIOS32. Процедура проверки наличия BIOS32 предполагает обращение к физическим адресам памяти, поэтому обычно производится из реального режима, до переключения в защищенный. Если BIOS32 поддерживается, то в области памяти BIOS, расположенной в диапазоне 0xE0000 – 0xFFFFF, должна присутствовать специальная 16-байтная структура данных – служебный каталог (BIOS32 Service Directory). Структура каталога приведена в таблице 1.

Таблица 1

Смещение	Размер, байт	Назначение
0x00	4	Сигнатура служебного каталога в коде ASCII: 32
0x04	4	32-разрядный физический адрес точки входа в BIOS32
0x08	1	Номер версии реализации BIOS32 (имеет значение 0x00)
0x09	1	Размер служебного каталога BIOS32 в 16-байтных параграфах
0x0A	1	Контрольная сумма
0x0B	5	Зарезервировано

Процесс поиска служебного каталога BIOS32 заключается в сканировании памяти ПЗУ BIOS: производится поиск сигнатуры «_32_» в диапазоне 0xE0000 – 0xFFFFF по 16-байтным параграфам (начало служебного каталога выровнено на границу 16 байт). После обнаружения сигнатуры производится вычисление и проверка контрольной суммы: если сумма совпадает, то служебный каталог найден.

Для доступа к PCI BIOS в 32-разрядном режиме требуется выполнить дальний вызов через точку входа BIOS32. Перед выполнением вызова в регистры записывается следующая информация:

- в EAX – идентификатор запрашиваемого сервиса, который для PCI BIOS имеет значение «$PCI» (0x49435024);
- в EBX – селектор функции (значение должно быть равно нулю).

После выполнения вызова в регистре AL будет возвращен код результата:

- 0 – операция успешно завершена;
- 0x80 – некорректный идентификатор сервиса;
- 0x81 – недопустимое значение селектора функции.

Если вызов завершен успешно, в регистрах процессора будет размещена следующая информация:

- в EBX – физический адрес базы сервиса BIOS;
- в ECX – размер сегмента сервиса BIOS;
- в EDX – точка входа в сервис BIOS (смещение относительно базы, возвращенной в EBX).

Адрес точки входа в сервис определяется путем сложения физического адреса базы сервиса и смещения относительно базы. Дальнейший порядок обращения к функциям сервиса PCI BIOS не отличается от реального режима, за исключением того, что вместо вызова прерывания int 0x1A необходимо выполнить дальний вызов через точку входа в сервис.

Алгоритм чтения MAC-адреса

Итак, вернемся к решению нашей задачи – определению MAC-адреса сетевого адаптера RTL8139C. Для этого нам был необходим базовый адрес в пространстве I/O, и это значение, как мы уже установили, находится в конфигурационном пространстве устройства. Чтобы извлечь его оттуда, можно воспользоваться сервисом BIOS32 либо работать с устройством напрямую, при помощи Configuration Mechanism #1.

Алгоритм решения задачи при использовании сервиса BIOS32 следующий:

- определяем адрес точки входа в BIOS32. Для этого выполняем поиск служебного каталога BIOS32 в диапазоне 0xE0000 – 0xFFFFF;
- определяем адрес точки в сервис BIOS32 – PCI BIOS. Для этого выполняем дальний вызов через точку входа в BIOS32, задав в регистре EAX идентификатор запрашиваемого сервиса ($PCI);
- используя PCI BIOS, выполняем поиск сетевого адаптера и определяем его координаты – номер шины, номер устройства и номер функции. Поиск производим по коду класса. Код класса сетевого контроллера равен 0x00020000 (см. [4]);
- из конфигурационного пространства сетевого адаптера считываем значение базового адреса порта I/O и, задавая смещение относительно этого адреса, считываем MAC-адрес адаптера RTL8139C.

При использовании Configuration Mechanism #1 всё гораздо проще и сводится к последовательной записи информации в порт CONFIG_ADDRESS и чтении её из порта CONFIG_DATA. Рассмотрим программную реализацию этих алгоритмов.

Программная реализация алгоритма чтения MAC-адреса

Разработаем модуль ядра, который при загрузке будет выполнять поиск сетевого адаптера RTL8139C и считывать его MAC-адрес. Все исходные тексты, рассмотренные в статье, доступны на сайте журнала.

Начнём с описания заголовочных файлов, переменных и информационных структур.

Заголовочных файлов у нас два:

```
#include <linux/module.h>
#include <linux/pci.h>
```

Определим код фирмы-изготовителя адаптера RTL8139C, код типа устройства и код класса устройства.

```
// код фирмы-изготовителя - RealTek
#define VENDOR_ID        0x10EC
// код устройства - сетевая карта RTL8139C
#define DEVICE_ID        0x8139
// код класса сетевого контроллера ([4])
#define CLASS_CODE       0x00020000
```

Функции PCI BIOS, которые мы будем использовать (полный перечень приведен в [6]):

```
// проверка присутствия PCI BIOS в системе
#define PCIBIOS_PCI_BIOS_PRESENT        0xb101
// поиск устройства PCI заданного типа
#define PCIBIOS_FIND_PCI_DEVICE         0xb102
// поиск устройства PCI заданного класса
#define PCIBIOS_FIND_PCI_CLASS_CODE     0xb103
// прочитать байт из конфигурационного пространства
```

```
// устройства PCI
#define PCIBIOS_READ_CONFIG_BYTE       0xb108
// прочитать слово из конфигурационного пространства
// устройства PCI
#define PCIBIOS_READ_CONFIG_WORD       0xb109
// прочитать двойное слово из конфигурационного пространства
// устройства PCI
#define PCIBIOS_READ_CONFIG_DWORD      0xb10a
```

Сигнатура, по которой производится поиск служебного каталога BIOS32 (_32_):

```
#define BIOS32_SIGNATURE (('_' << 0) + ('3' << 8) + ↵
    ('2' << 16) + ('_' << 24))
```

Сигнатура для проверки присутствия PCI BIOS в системе (используется функцией PCIBIOS_PCI_BIOS_PRESENT):

```
#define PCI_SIGNATURE    (('P' << 0) + ('C' << 8) + ↵
    ('I' << 16) + (' ' << 24))
```

Сигнатура, по которой осуществляется поиск сервиса BIOS32:

```
#define PCI_SERVICE      (('$' << 0) + ('P' << 8) + ↵
    ('C' << 16) + ('I' << 24))
```

Определим структуру для хранения информации об устройстве PCI:

```
struct pci_dev_struct {
  // код фирмы-изготовителя и код типа устройства
  u16 vendor_id, device_id;
  // код класса устройства
  u32 class_code;
  // адрес порта I/O
  u32 base_addr;
  // координаты устройства - номер шины, номер устройства
  // на шине и номер функции устройства
  u8 bus, dev, fn;
};
```

Следующие две структуры описывают физические адреса точек входа в BIOS32 и в сервис BIOS32 (PCI BIOS).

Физический адрес точки входа в BIOS32 (в формате селектор:смещение):

```
static struct {
  u32 address;
  u16 segment;
} bios32_indirect = { 0, __KERNEL_CS };
```

Физический адрес точки входа в сервис BIOS32 (PCI BIOS):

```
static struct {
  u32 address;
  u16 segment;
} pci_indirect = { 0, __KERNEL_CS };
```

__KERNEL_CS – селектор сегмента кода, определен в файле include/asm-i386/segment.h:

```
#define __KERNEL_CS 0x10
```

Стандартный служебный каталог BIOS32 имеет следующий вид:

```
union bios32 {
  struct {
    // сигнатура _32_
    u32 signature;
    // 32-х битный физический адрес точки входа в BIOS32
    u32 entry;
    // номер версии, Revision level, 0
    u8 revision;
    // размер служебного каталога в 16-байтных параграфах
    u8 length;
    // контрольная сумма, дополняет все байты до 0
    u8 checksum;
    // зарезервировано, заполняется нулями
    u8 reserved[5];
  } fields;
  char chars[16];
};
```

Рассмотрим функцию, которая производит поиск служебного заголовка BIOS32.

```
int pci_find_bios(void)
{
  union bios32 *check; // служебный каталог BIOS32
  u8 sum;
  int i, length;
```

Сканируем область памяти BIOS в диапазоне адресов 0xe0000 и 0xfffff в поисках сигнатуры «_32_» и служебного каталога BIOS32:

```
  for (check = (union bios32 *) __va(0xe0000);
    check <= (union bios32 *) __va(0xffff0); ++check) {
      if (check->fields.signature != BIOS32_SIGNATURE) continue;
```

Поиск выполняется относительно нижней границы адресного пространства ядра 0xC0000000, на что указывает макрос __va(0xe0000) и __va(0xffff0).

Этот макрос определен в файле include/page.h следующим образом:

```
// нижняя граница адресного пространства ядра
#define __PAGE_OFFSET 0xC0000000
#define PAGE_OFFSET ((unsigned long)__PAGE_OFFSET)
#define __va(x) ((void *)((unsigned long)(x) + PAGE_OFFSET))
```

Если сигнатура найдена – определяем размер служебного каталога BIOS32 в байтах:

```
      length = check->fields.length * 16;
      if (!length) continue;
```

Считываем контрольную сумму и определяем номер версии реализации:

```
      sum = 0;
      for (i = 0; i < length ; ++i)
        sum += check->chars[i];
      if (sum != 0)
        continue;
      if (check->fields.revision != 0) {
        printk("PCI: unsupported BIOS32 revision %d ↵
          at 0x%p\n", check->fields.revision, check);
      continue;
      }
```

Если вышли за пределы диапазона сканирования, то использовать BIOS32 мы не сможем:

```
      if (check->fields.entry >= 0x100000) {
        printk("PCI: BIOS32 entry (0x%p) in high memory, ↵
          cannot use.\n", check);
        return 0;
      } else {
```

Если всё в порядке – вычисляем адрес точки входа в BIOS32 и заполняем структуру bios32_indirect:

```
        unsigned long bios32_entry = check->fields.entry;
        bios32_indirect.address = bios32_entry + ↵
          PAGE_OFFSET; // адрес точки входа в BIOS32
        printk(KERN_INFO "PCI: BIOS32 entry point at ↵
          0x%08x\n", bios32_indirect.address);
      }
      break;
      /* Hopefully more than one BIOS32 cannot happen... */
  }
  return 0;
}
```

Следующая функция, которую мы рассмотрим, определяет адрес точки входа в сервис BIOS32.

```
static u32 bios32_service(u32 service)
{
  u8 return_code; /* %al, код возврата */
  u32 address;    /* %ebx, адрес базы сервиса*/
  u32 length;     /* %ecx, размер сегмента сервиса  */
  u32 entry;      /* %edx, точка входа в сервис */
  u32 flags;
```

Идентификатор сервиса передается в параметрах функции.

Адрес точки входа в сервис BIOS32 определяется путем дальнего вызова через точку входа в BIOS32. Перед выполнением вызова в регистры процессора заносится следующая информация:

- EAX – идентификатор сервиса (в нашем случае это $PCI);
- EBX – селектор функции (должен быть равен 0);
- EDI – адрес точки входа в BIOS32.

После вызова регистры процессора будут содержать следующую информацию:

- AL – код возврата: 0 – запрашиваемый сервис найден, 0x80 – сервис отсутствует (не поддерживается);
- EBX – физический адрес базы сервиса;
- ECX – размер сегмента сервиса;
- EDX – точка входа в сервис BIOS32 (смещение относительно базы, возвращённой в регистре EBX).

Вот как выполняется данный вызов:

```
__save_flags(flags); __cli();
__asm__("lcall (%%edi); cld"
  : "=a" (return_code),
  // физический адрес базы сервиса
  "=b" (address),
  // размер сегмента сервиса
  "=c" (length),
  // точка входа в сервис BIOS32 (смещение относительно
  // базы, возвращенной в EBX)
  "=d" (entry)
  // идентификатор запрашиваемого сервиса ($PCI)
  : "0" (service),
  // селектор функции, должен быть равен нулю
  "1" (0),
  // адрес точки входа в BIOS32
  "D" (&bios32_indirect));
__restore_flags(flags);
```

Конструкция типа

```
__save_flags(flags);
__cli();

/* This code runs with interrupts disabled */

__restore_flags(flags);
```

используется для защиты критичных участков кода от воздействия прерываний.

Проанализируем код возврата:

```
switch (return_code) {
```

Если код возврата равен 0, то сервис PCI BIOS присутствует, и адрес точки входа в него можно определить, сложив значение адреса базы (address) и смещения относительно базы (entry):

```
    case 0:
      // искомый адрес точки входа в сервис BIOS32 (PCI BIOS)
      return address + entry;
```

Если код возврата равен 0x80, запрашиваемый сервис отсутствует:

```
    case 0x80:    /* Not present */
      printk(KERN_WARNING "bios32_service(0x%08x): ↵
          not present\n", service);
      return 0;
    default: /* Shouldn't happen */
      printk(KERN_WARNING "bios32_service(0x%08x): returned ↵
          0x%x -- BIOS bug!\n", service, return_code);
      return 0;
  }
}
```

А теперь рассмотрим порядок обращения к сервису PCI BIOS в защищенном режиме, выполнив проверку присутствия PCI BIOS в системе:

```
int check_pcibios(void)
{
  u32 signature, eax, ebx, ecx;
  u8 status, major_ver, minor_ver, hw_mech;
  u32 flags,
  u32 pcibios_entry;
```

pcibios_entry – это точка входа в сервис, назначение остальных переменных рассмотрим ниже. Ищем точку входа в сервис PCI путём вызова функции bios32_service(). Параметр функции – идентификатор сервиса, сигнатура $PCI. Все адреса отсчитываются относительно нижней границы адресного пространства ядра (PAGE_OFFSET = 0xC0000000).

```
  if ((pcibios_entry = bios32_service(PCI_SERVICE))) {
    pci_indirect.address = pcibios_entry + PAGE_OFFSET;
```

Выполняем обращение к функции проверки присутствия PCI BIOS в системе – 0xB101. Для этого выполняем дальний вызов через точку входа pcibios_entry, предварительно заполнив регистры процессора следующей информацией:

- EAX – запрашиваемая функция сервиса, в данном случае 0xB101;
- EDI – адрес точки входа в сервис.

В результате выполнения вызова в регистрах процессора будет находиться следующая информация:

- EDX – сигнатура запрашиваемого сервиса (PCI в нашем случае);
- AH – признак присутствия сервиса (0 – PCI BIOS присутствует, если в EDX правильная сигнатура, любое другое значение – PCI BIOS отсутствует);
- AL – поддерживаемый аппаратный механизм конфигурирования (см. «Configuration Mechanism #1»);
- BH – номер версии интерфейса;

- BL – подномер версии интерфейса;
- CL – номер последней шины PCI в системе.

Итак, выполняем вызов:

```
__save_flags(flags); __cli();
__asm__(
  "lcall (%%edi); cld\n\t"
  "jc 1f\n\t"
  "xor %%ah, %%ah\n"
  "1:"
  // в EDX возвращается сигнатура "PCI"
  :   "=d" (signature),
  // в AH - признак присутствия, в AL - аппаратный механизм
      "=a" (eax),
  // в BH - номер версии интерфейса PCI, BL - подномер
  // версии интерфейса
      "=b" (ebx),
  // ECX - номер последней шины PCI в системе
      "=c" (ecx)
  // 0xB101 - функция проверки присутствия PCI BIOS
  // в системе
  :   "1" (PCIBIOS_PCI_BIOS_PRESENT),
  // точка входа в сервис BIOS32
      "D" (&pci_indirect)
  :   "memory");
__restore_flags(flags);
```

и обрабатываем полученные результаты:

```
// признак присутствия сервиса с системе
status = (eax >> 8) & 0xff;
// поддерживаемый аппаратный механизм
hw_mech = eax & 0xff;
// номер версии
major_ver = (ebx >> 8) & 0xff;
// номер подверсии
minor_ver = ebx & 0xff;
```

Если сервис присутствует, переменная status будет равна 0. Проверяем это, а заодно и полученную сигнатуру:

```
if (status || signature != PCI_SIGNATURE) {
  printk (KERN_ERR "PCI: BIOS BUG #%x[%08x] ↵
       found\n", status, signature);
  return 0;
}
printk(KERN_INFO "PCI: PCI BIOS revision %x.%02x entry ↵
       at 0x%08x\n", major_ver, minor_ver, pcibios_entry);

return 1;
}
return 0;
}
```

Функция pci_bios_find_device() выполняет поиск устройства заданного типа при помощи PCI BIOS и возвращает его координаты – номер шины, к которой подключено устройство, номер устройства на шине и номер функции устройства:

```
static int pci_bios_find_device(u16 vendor, u16 device_id, ↵
    u16 index, u8 *bus, u8 *dev, u8 *fn)
{
  u16 bx;
  u16 ret;
```

Функция принимает следующие параметры:

- vendor – код фирмы-изготовителя устройства PCI;
- device_id – код типа устройства;
- index – порядковый номер устройства заданного типа. Если устройство одно, то его порядковый номер равен 0.

Параметры bus, dev и fn, соответствующие координатам устройства PCI (номер шины, номер устройства на шине и номер функции), передаются по ссылке и будут изменены на реальные значения.

Для поиска устройства выполняем дальний вызов через точку входа в сервис BIOS32, задав в регистрах процессора соответствующие параметры:

- EAX – запрашиваемая функция сервиса, в данном случае 0xB102.
- ECX – код типа устройства.
- EDX – код фирмы-изготовителя устройства.
- ESI – индекс (порядковый номер) устройства заданного типа.
- В регистр EDI занесем адрес точки входа в сервис.

В результате выполнения вызова в регистрах процессора будет находиться следующая информация:

- BH – номер шины, к которой подключено устройство;
- BL – номер устройства в старших пяти битах и номер функции в трёх младших;
- AH – код возврата (может принимать значения BAD_VENDOR_ID, DEVICE_NOT_FOUND и SUCCESFUL).

Выполняем дальний вызов:

```
__asm__("lcall (%%edi); cld\n\t"
  "jc 1f\n\t"
  "xor %%ah, %%ah\n"
  "1:"
  :     "=b" (bx),
        "=a" (ret)
  :     "1" (PCIBIOS_FIND_PCI_DEVICE),
        "c" (device_id),
        "d" (vendor),
        "S" ((int) index),
  // адрес точки входа в сервис
        "D" (&pci_indirect));
```

Обрабатываем полученный результат:

```
  *bus = (bx >> 8) & 0xff; // номер шины
  *dev = (bx & 0xff) >> 3; // номер устройства на шине
  *fn = bx & 0x3;          // номер функции
  return (int) (ret & 0xff00) >> 8;
}
```

Функция поиска устройства заданного класса pci_bios_find_class() практически не отличается от функции поиска устройства по типу:

```
static int pci_bios_find_class(u32 class_code, u16 index, ↵
    struct pci_dev_struct *pd)
{
  u16 bx;
  u16 ret;
```

В параметрах функции передается указатель на информационную структуру struct pci_dev_struct *pd.

Перед выполнением дальнего вызова в регистры заносятся следующие данные:

- EAX – запрашиваемая функция сервиса – 0xB103.
- ECX – код класса устройства.
- ESI – индекс (порядковый номер) устройства заданного типа.
- В регистр EDI занесем адрес точки входа в сервис.

Результаты выполнения вызова аналогичны предыдущим:

- в регистре BH – номер шины;

- в BL – номер устройства в старших пяти битах и номер функции в трёх младших;
- в AH – код возврата (DEVICE_NOT_FOUND или SUCCESFUL):

```
__asm__("lcall (%%edi); cld\n\t"
  "jc 1f\n\t"
  "xor %%ah, %%ah\n"
  "1:"
  :     "=b" (bx),
        "=a" (ret)
  :     "1" (PCIBIOS_FIND_PCI_CLASS_CODE),
        "c" (class_code),
        "S" ((int) index),
  // адрес точки входа в сервис
        "D" (&pci_indirect));
```

Заносим в структру struct pci_dev_struct *pd координаты устройства:

```
  // номер шины
  pd->bus = (bx >> 8) & 0xff;
  // номер устройства на шине
  pd->dev = (bx & 0xff) >> 3;
  // номер функции
  pd->fn = bx & 0x03;
  return (int) (ret & 0xff00) >> 8;
}
```

После того как устройство заданного типа найдено, необходимо получить данные, находящиеся в его конфигурационном пространстве (см. «Конфигурационное пространство устройства PCI»). Сделаем это при помощи функции pci_bios_read():

```
static int pci_bios_read(int bus, int dev, int fn, int reg, ↵
    int len, u32 *value)
{
  u32 result = 0;
  u32 bx;
```

Параметрами функции являются координаты устройства (bus – номер шины, dev – номер устройства, fn – номер функции), смещение в конфигурационном пространстве (reg) и размер данных для считывания (len, байт/слово/двойное слово). В последний параметр мы поместим считанное из конфигурационного пространства значение, поэтому этот параметр передается по ссылке.

Проверяем правильность переданных параметров:

```
  if (bus > 255 || dev > 31 || fn > 7 || reg > 255)
    return -EINVAL;
```

Для чтения информации из конфигурационного пространства устройства PCI BIOS предоставляет следующие функции [6]:

- 0xB108 – чтение байта;
- 0xB109 – чтение слова;
- 0xB10A – чтение двойного слова.

Эти функции отличаются только размером считываемых данных – байт, слово или двойное слово. Перед вызовом функции в регистры процессора помещается следующая информация:

- EAX – код функции;
- BH – номер шины, к которой подключено устройство;
- BL – номер устройства в старших пяти битах и номер функции в трёх младших битах;
- DI – смещение в конфигурационном пространстве.

После выполнения функции в регистре ECX будут находиться считанные данные, а регистр AH будет содержать код возврата. Подготовим значение для загрузки в регистр BX и прочитаем информацию из конфигурационного пространства устройства, учитывая размер запрашиваемых данных:

```
  bx = ((bus << 8) | (dev << 3) | fn);

  switch (len) {
  case 1: // считываем байт
    __asm__("lcall (%%esi); cld\n\t"
      "jc 1f\n\t"
      "xor %%ah, %%ah\n"
      "1:"
      :   "=c" (*value),
          "=a" (result)
      :   "1" (PCIBIOS_READ_CONFIG_BYTE),
          "b" (bx),
          "D" ((long)reg),
      // точка входа в сервис
          "S" (&pci_indirect));
    break;

  case 2: // считываем слово
    __asm__("lcall (%%esi); cld\n\t"
      "jc 1f\n\t"
      "xor %%ah, %%ah\n"
      "1:"
      :   "=c" (*value),
          "=a" (result)
      :   "1" (PCIBIOS_READ_CONFIG_WORD),
          "b" (bx),
          "D" ((long)reg),
      // точка входа в сервис
          "S" (&pci_indirect));
    break;

  case 4: // считываем двойное слово
    __asm__("lcall (%%esi); cld\n\t"
      "jc 1f\n\t"
      "xor %%ah, %%ah\n"
      "1:"
      :   "=c" (*value),
          "=a" (result)
      :   "1" (PCIBIOS_READ_CONFIG_DWORD),
          "b" (bx),
          "D" ((long)reg),
      // точка входа в сервис
          "S" (&pci_indirect));
    break;
  }
  return (int)((result & 0xff00) >> 8);
}
```

Помимо средств BIOS32 для работы с конфигурационным пространством устройства PCI в защищенном режиме также используется Configuration Mechanism #1. Его порядок работы был рассмотрен в «Configuration Mechanism #1»: в порт CONFIG_ADDRESS (0xCF8) заносится адрес, соответствующий формату, приведенному на рис. 2; обращением к порту CONFIG_DATA (0xCFC) производится чтение или запись данных в требуемый регистр конфигурационного пространства.

Формировать адрес установленного формата будет макрос PCI_CONF1_ADDRESS():

```
#define PCI_CONF1_ADDRESS(bus, dev, fn, reg) | ↵
    (0x80000000 | (bus << 16) | (dev << 11) | ↵
    (fn << 8) | (reg & ~3))
```

Самый старший бит установлен в 1 – это позволит нам получить данные из порта CONFIG_DATA.

После того как адрес сформирован, записываем его в порт CONFIG_ADDRESS.

Функция pci_direct_read() выполняет обращение к устройству PCI при помощи Configuration Mechanism #1:

```
static int pci_direct_read(int bus, int dev, int fn, ↵
    int reg, int len, u32 *value)
{
```

Параметры функции – номер шины bus, номер устройства на шине dev, номер функции fn, смещение в конфигурационном пространстве reg, длина запрашиваемых данных (байт/слово/двойное слово). Результат помещается в параметр value, который передается по ссылке.

Проверяем правильность переданных параметров:

```
  if (bus > 255 || dev > 31 || fn > 7 || reg > 255)
    return -EINVAL
```

Формируем адрес при помощи макроса PCI_CONF1_ADDRESS() и записываем его в порт CONFIG_ADDRESS:

```
  outl(PCI_CONF1_ADDRESS(bus, dev, fn, reg), 0xCF8);
```

Считываем значение из порта CONFIG_DATA:

```
  switch (len) {
  case 1: // считываем байт
    *value = inb(0xCFC + (reg & 3));
    break;
  case 2: // считываем слово
    *value = inw(0xCFC + (reg & 2));
    break;
  case 4: // считываем двойное слово
    *value = inl(0xCFC);
    break;
  }
  return 0;
}
```

Функция pci_direct_find_class() выполняет поиск устройства заданного класса, используя Configuration Mechanism #1, и возвращает его координаты – номер шины, номер устройства на шине и номер функции:

```
int pci_direct_find_class(u32 class_code, ↵
    struct pci_dev_struct *pd)
{
```

Параметры функции – код класса устройства и структура struct pci_dev_struct *pd, в которую необходимо записать координаты устройства.

```
  int bus, dev, fn = 0, idx = 0x08;
  u32 config_dword, code;
```

Переменная idx – это смещение в конфигурационном пространстве устройства, и указывает оно на поле Revision ID, за которым следуют три байта поля Class Code. Для считывания Class Code достаточно считать двойное слово, находящееся по смещению idx, и сдвинуть результат на 8 бит в сторону младших разрядов.

Для поиска устройства по коду класса необходимо просканировать все шины, все устройства на каждой шине и все функции устройства – до тех пор, пока не найдем устройство соответствующего класса (или пока не закончатся шины). С этой целью организуем цикл:

```
  printk(KERN_INFO "Looking for device with class code ↵
    0x%X\n", class_code);
  memset(pd, 0, sizeof(struct pci_dev_struct));
  // сканируем все шины
  for(bus = 0; bus < 256; bus++) {
    // сканируем все устройства на каждой шине
    for(dev = 0; dev < 32; dev++) {
      // сканируем все функции
      for(fn = 0; fn < 8; fn++) {
```

Считываем двойное слово, находящееся по смещению idx, и получаем код класса:

```
        pci_direct_read(bus, dev, fn, idx, 4, &config_dword);
        code = config_dword >> 8;
```

Сравниваем полученный код класса с искомым. При совпадении сохраняем координаты устройства

```
          if(code == class_code) {

            printk(KERN_INFO "OK. Device found.\n");
            printk(KERN_INFO "bus - %d, dev - %d, ↵
                   fn - %d\n", bus, dev, fn);

            pd->bus = bus;
            pd->dev = dev;
            pd->fn = fn;

            return 0;
          }
        }
      }
    }
    return 0x80;
}
```

Все функции, которые мы рассмотрели, будут вызваны во время процедуры инициализации модуля:

```
static int __init pcidev_on(void)
{
  // структура с параметрами PCI-устройства
  struct pci_dev_struct pdev;
  // смещение к данным в конфигурационном пространстве
  // устройства PCI
  int idx = 0;
  // координаты устройства
  u8 bus = 0, dev = 0, fn = 0;
  // командный регистр
  u16 command_reg = 0;
  u32 config_dword = 0;
```

Напомню, что наша задача – прочитать MAC-адрес сетевого адаптера RTL8139C, и для этого нам необходимо получить его базовый адрес в пространстве I/O. Ищем служебный заголовок BIOS32, вычисляем адрес точки входа в BIOS32 и производим проверку присутствия PCI BIOS:

```
  pci_find_bios();
  check_pcibios();
```

Выполняем поиск устройства (сетевого адаптера RTL8139C) по коду типа устройства. В результате мы получим его координаты – номер шины, номер устройства на шине и номер функции:

```
  if(pci_bios_find_device(VENDOR_ID, DEVICE_ID, idx, ↵
    &bus, &dev, &fn) == PCIBIOS_SUCCESSFUL)
    printk(KERN_INFO "Device found by type, bus - %d, ↵
           dev - %d, fn - %d\n", bus, dev, fn);
```

Повторим процедуру, но в этот раз будем искать устройство по коду класса:

```
  memset((void *)&pdev, 0, sizeof(struct pci_dev_struct));
  if(pci_bios_find_class(CLASS_CODE, idx, ↵
    &pdev) == PCIBIOS_SUCCESSFUL)
    printk(KERN_INFO "Device found by class, bus - %d, ↵
           dev - %d, fn - %d\n", pdev.bus, pdev.dev, pdev.fn);
    else {
```

```
        printk(KERN_INFO "Device not founf by class\n");
        return 0;
      }
```

Итак, устройство найдено. Считываем из конфигурационного пространства код фирмы-производителя и заносим это значение в структуру struct pci_dev_struct pdev:

```
/* Read VENDOR ID */
  idx = 0x00;
  if(pci_bios_read(pdev.bus, pdev.dev, pdev.fn, idx, ↵
    2, &config_dword) == PCIBIOS_SUCCESSFUL)
     pdev.vendor_id = (u16)config_dword;
```

То же самое – для кода типа устройства и для кода класса устройства:

```
/* Read DEVICE ID */
  idx = 0x02;
  if(pci_bios_read(pdev.bus, pdev.dev, pdev.fn, idx, ↵
    2, &config_dword) == PCIBIOS_SUCCESSFUL)
     pdev.device_id = (u16)config_dword;

/* Read Class Code */
  idx = 0x08;
  if(pci_bios_read(pdev.bus, pdev.dev, pdev.fn, idx, ↵
    4, &config_dword) == PCIBIOS_SUCCESSFUL)
     pdev.class_code = config_dword >> 8;
```

Считываем значение командного регистра:

```
/* Read Command Register */
  idx = 0x04;
  if(pci_bios_read(pdev.bus, pdev.dev, pdev.fn, idx, ↵
    1, &config_dword) == PCIBIOS_SUCCESSFUL)
     command_reg = config_dword;
```

Считываем значение базового адреса в пространстве I/O. Предварительно проверяем, чтобы бит 0 командного регистра был установлен в единицу. Если это так, то выполняем поиск базового адреса устройства:

```
/* Read Base Address Registers */
  idx = 0x10;
  if(command_reg & 0x01) {
    // сканируем Base Address Registers в поисках адреса
    // порта I/O
    for(; idx < 0x28 ;) {
      if(pci_bios_read(pdev.bus, pdev.dev, pdev.fn, ↵
          idx, 4, &config_dword) == PCIBIOS_SUCCESSFUL) {
        // если нулевой бит равен 1, то адрес порта
        // I/O найден
        if(config_dword & 0x01) {
          config_dword &= ~0x1;
          pdev.base_addr = config_dword;
          break;
        }
        idx += 4;
      }
    }
  } else return 0;
```

Базовый адрес найден. Отобразим информацию об устройстве и прочитаем MAC-адрес адаптера RTL8139C:

```
  display_pcidev_info(&pdev);
  get_mac_addr(pdev.base_addr);
```

Функции display_pcidev_info() и get_mac_addr() выглядят следующим образом:

```
void display_pcidev_info(struct pci_dev_struct *pdev)
{
  printk(KERN_INFO "VENDOR ID - 0x%X\n", pdev->vendor_id);
  printk(KERN_INFO "DEVICE ID - 0x%X\n", pdev->device_id);
  printk(KERN_INFO "CLASS CODE - 0x%X\n", pdev->class_code);
  printk(KERN_INFO "BASE ADDRESS - 0x%X\n", pdev->base_addr);
  return;
}

void get_mac_addr(u32 base_addr)
{
  int i = 0;
  u8 mac[6];

  memset(mac, 0, 6);

/* Get and display MAC address */
  for(; i < 6; i++)
    mac[i] = inb(base_addr + i);

  printk(KERN_INFO "MAC address: ↵
   %02X:%02X:%02X:%02X:%02X:%02X\n", mac[0], mac[1], ↵
   mac[2], mac[3], mac[4], mac[5]);

  return;
}
```

Теперь давайте выполним процедуру чтения MAC-адреса сетевого адаптера RTL8139C, используя Configuration Mechanism #1 для доступа к конфигурационному пространству устройства.

Пытаемся найти устройство по коду класса:

```
/* Direct read PCI */
  printk(KERN_INFO "PCI direct access:\n");
  if(pci_direct_find_class(CLASS_CODE, &pdev) < 0) {
    printk(KERN_INFO "Device not found\n");
    return 0;
  }
```

Считываем код фирмы-производителя, код типа устройства и код класса устройства:

```
/* Read VENDOR_ID */
  idx = 0x00;
  if(pci_direct_read(pdev.bus, pdev.dev, pdev.fn, idx, ↵
    2, &config_dword) == 0)
     printk(KERN_INFO "VENDOR_ID - 0x%X\n", config_dword);

/* Read DEVICE_ID */
  idx = 0x02;
  if(pci_direct_read(pdev.bus, pdev.dev, pdev.fn, idx, ↵
    2, &config_dword) == 0)
     printk(KERN_INFO "DEVICE_ID - 0x%X\n", config_dword);

/* Read Class Code */
  idx = 0x08;
  if(pci_direct_read(pdev.bus, pdev.dev, pdev.fn, idx, ↵
    4, &config_dword) == 0)
     printk(KERN_INFO "CLASS_CODE - 0x%X\n", ↵
            config_dword >> 8);
```

Считываем содержимое командного регистра и значение адреса порта I/O:

```
/* Read Command Register */
  command_reg = 0;
  idx = 0x04;
  if(pci_direct_read(pdev.bus, pdev.dev, pdev.fn, idx, ↵
    2, &config_dword) == 0)
     command_reg = config_dword;

/* Read Base Address Registers */
  idx = 0x10;
  if(command_reg & 0x01) {
    for(; idx < 0x28 ;) {
      if(pci_direct_read(pdev.bus, pdev.dev, pdev.fn, ↵
          idx, 4, &config_dword) == PCIBIOS_SUCCESSFUL) {
          if(config_dword & 0x01) {
             config_dword &= ~0x1;
             pdev.base_addr = config_dword;
             break;
          }
          idx += 4;
      }
    }
```

```
    } else return 0;
```

Считываем значение MAC-адреса сетевого адаптера.

```
    get_mac_addr(config_dword);
    return 0;
}
```

Выгружает модуль из памяти функция pcidev_off:

```
static void __exit pcidev_off(void)
{
  return;
}
```

Инициализация модуля и выгрузка его из памяти выполняется при помощи двух макросов:

```
module_init(pcidev_on);
module_exit(pcidev_off);
```

Исходные тексты модуля доступны на сайте журнала и находятся в файле pcidev.c. При помощи команды make получаем объектный модуль pcidev.o и загружаем его командой insmod:

```
    insmod pcidev.o
```

Вся информация, полученная от устройства, будет собрана в файле /var/log/messages.

```
Feb 11 16:02:59 bob kernel: PCI: BIOS32 entry point at 0xc00fb140
Feb 11 16:02:59 bob kernel: PCI: PCI BIOS revision 2.10 entry at 0x000fb170
Feb 11 16:02:59 bob kernel: Device found by type, bus - 1, dev - 9, fn - 0
Feb 11 16:02:59 bob kernel: Device found by class, bus - 1, dev - 9, fn - 0
Feb 11 16:02:59 bob kernel: VENDOR ID - 0x10EC
Feb 11 16:02:59 bob kernel: DEVICE ID - 0x8139
Feb 11 16:02:59 bob kernel: CLASS CODE - 0x20000
Feb 11 16:02:59 bob kernel: BASE ADDRESS - 0xC000
Feb 11 16:02:59 bob kernel: MAC address: 00:02:44:72:5E:4E
Feb 11 16:02:59 bob kernel: PCI direct access:
Feb 11 16:02:59 bob kernel: Looking for device with class code 0x20000
Feb 11 16:02:59 bob kernel: OK. Device found.
Feb 11 16:02:59 bob kernel: bus - 1, dev - 9, fn - 0
Feb 11 16:02:59 bob kernel: Class Code - 0x20000
Feb 11 16:02:59 bob kernel: VENDOR_ID - 0x10EC
Feb 11 16:02:59 bob kernel: DEVICE_ID - 0x8139
Feb 11 16:02:59 bob kernel: CLASS_CODE - 0x20000
Feb 11 16:02:59 bob kernel: MAC address: 00:02:44:72:5E:4E
```

Пример записи из этого файла сравните с результатами, полученными при помощи команд dmesg и ifconfig (см. «Постановка задачи и исходные данные»).

Заключение

Рассмотренные нами функции являются базовыми в подсистеме низкоуровневой поддержки (low-level support) шины PCI ядра ОС Linux. Все эти функции можно найти в файле arch/i386/kernel/pci-pc.c.

В повседневной практике нет особой необходимости работать напрямую с шиной, для этих целей целесообразно применять функции более высокого уровня, перечень которых приведён в файле Documentation/pci.txt.

Насчет спецификаций и где их брать – спецификация на RTL8139C находится на сайте компании RealTek, www.realtek.com.tw, спецификация PCI 3.0 и перевод на русский язык спецификации PCI 2.0 были найдены на сайте http://dsp.neora.ru. На сайте Intel (www.intel.com) можно взять спецификацию на сетевые карты Intel 8255x – для этого в строке поиска задайте 8255X_OpenSDM (OpenSDM – Open Source Software Developer Manual). Также посетите сайт фирмы Phoenix (www.phoenix.com) – материалы по BIOS.

Список кодов классов и подклассов устройств PCI находится в [4], приложение D.

Литература:

1. Аппаратные средства IBM PC. Энциклопедия, 2-е изд. / М. Гук – СПб.: Питер, 2003. – 923 с.:ил.
2. Программирование на аппаратном уровне: специальный справочник. 2-е изд. / В. Кулаков. – СПб.: Питер, 2003. – 848 с.:ил.
3. Шина PCI (Peripheral Component Interconnect bus). Николай Дорофеев, www.ixbt.com.
4. PCI Local Bus Specification. Revision 3.0. August 12, 2002.
5. Standard BIOS 32-bit Service Directory Proposal, Revision 0.4 May 24, 1993
6. PCI BIOS specification. Revision 2.0. 1993.