Сетевые буферы и управление памятью

Alan Cox
1 октября 1996 г.

Оригинал статьи

Операционная система Linux реализует отраслевой стандарт Berkeley socket API, который создан разработчиками BSD Unix (4.2/4.3/4.4 BSD). В этой статье рассмотрено управление памятью и буферизация для сетевых уровней и драйверов сетевых устройств в ядре Linux, а также объяснены некоторые изменения, появившиеся с течением времени.

Основные концепции

Сетевые уровни являются объектно-ориентированными по своей природе, как и большая часть ядра Linux. Основная структура сетевого кода базируется на реализациях Ross Biro и Orest Zborowski. Основные объекты перечислены ниже.

- **Устройство или интерфейс.** Сетевой интерфейс является программным кодом для приёма и передачи пакетов данных. Обычно интерфейс применяется для физического устройства типа адаптера Ethernet, однако некоторые устройства могут быть программными (например, петлевой интерфейс loopback, служащий для передачи данных самому себе).
- **Протокол.** Каждый протокол представляет собой особый сетевой язык. Некоторые протоколы существуют лишь потому, что производитель выбрал использование фирменных сетевых схем, тогда как другие протоколы разработаны для решения задач общего назначения. В ядре Linux каждый протокол является отдельным модулем кода, который обеспечивает обслуживание уровня сокетов.
- **Сокет** представляет собой соединение, обеспечивающее операции ввода-вывода Unix-файлов, и существует как дескриптор файла в пользовательской программе. В ядре каждый сокет является парой структур, которая представляет интерфейсы с вышележащим уровнем сокета и нижележащим протокольным уровнем.
- **sk_buff.** Все буферы, используемые сетевыми уровнями, представляют собой структуры **sk_buff**. Управление этими буферами обеспечивается низкоуровневыми библиотечными процедурами, доступными всем сетевым системам. Структуры **sk_buff** обеспечивают базовые функции буферизации и управления потоком, которые нужны сетевым протоколам.

Реализация sk_buff

Основной целью функций **sk_buff** является обеспечение согласованного и эффективного метода обработки буферов для всех сетевых уровней и протоколов.

Структура **sk_buff** является управляющей и связана с блоком памяти. В библиотеке **sk_buff** обеспечивается два основных набора функций. Первый набор включает программы управления связными списками **sk_buffs**, а второй – функции для управления связанной со структурой памятью. Буферы хранятся в связных списках, оптимизированных для базовых сетевых операций добавления в конец и удаления из начала. Поскольку основная часть сетевых функций выполняется в процессе прерываний, эти программы написаны для работы с неделимой памятью. Незначительные добавочные издержки снижают затраты, связанные с поиском ошибок.

Мы используем список операций для управления группами пакетов, прибывающих из сети и передаваемых в сеть через физические интерфейсы. Программы управления памятью служат для обработки содержимого пакетов стандартизованными и эффективными способами.

На базовом уровне список буферов поддерживается функциями вида

```
void append_frame(char *buf, int len)
{
  struct sk_buff *skb=alloc_skb(len, GFP_ATOMIC);
  if(skb==NULL)
    my_dropped++;
  else
  {
    skb_put(skb,len);
    memcpy(skb->data,data,len);
    skb_append(&my_list, skb);
  }
}
void process_queue(void)
{
  struct sk_buff *skb;
  while((skb=skb_dequeue(&my_list))!=NULL)
  {
    process_data(skb);
    kfree_skb(skb, FREE_READ);
  }
}
```

Эти два упрощённых фрагмента кода на деле достаточно точно показывают процесс приёма пакета. Функция **append_frame()** похожа на код, вызываемый из прерывания драйвером устройства, принявшего пакет, а функция **process_frame()** похожа на код, вызываемый для подачи данных в протоколы. Если вы посмотрите функции **netif_rx()** и **net_bh()** в файле net/core/dev.c, вы увидите аналогичное управление буферами. Оно более сложно, поскольку подаёт

пакеты в реальные протоколы и управляет потоком данных, но базовые операции остаются теми же. Аналогичная картина наблюдается для буферов, идущих от протокольного кода к пользовательским приложениям.

Пример также показывает использование функции управления данными **skb_put()**. Здесь она служит для резервирования буферного пространства для данных, которые мы хотим передать.

Посмотрим на функцию **append_frame()**. Функция **alloc_skb()** получает буфер размером **len** байтов (Рисунок 1), который состоит из:

- 0 байтов пространства в голове буфера;
- 0 байтов данных;
- **len** байтов пространства в конце буфера.

Функция **skb_put()** (Рисунок 4) увеличивает область данных в памяти вверх за счёт свободного пространства в конце буфера и таким образом резервирует пространство для **memcpy()**. Многие сетевые операции, передающие данные, добавляют пространство в начале кадра всякий раз при выполнении передачи, когда к пакету добавляются заголовки. По этой причине создана функция **skb_push()** (Рисунок 5), которая перемещает начало кадра данных в памяти вниз, если зарезервировано достаточно пространства для завершения этой операции.

Рисунок 1. После вызова alloc_skb.

Рисунок 2. После вызова skb_reserve.

Рисунок 3. Структура sk_buff с данными

Рисунок 4. После вызова skb_put для буфера.

Рисунок 5. После вызова skb_push для предыдущего буфера.

Сразу после выделения буфера все доступное пространство размещается в конце буфера. Функция **skb_reserve()** (Рисунок 2) может быть вызвана до того, как будут добавлены данные. Эта функция позволяет выделить некое пространство в начале буфера. Многие программы передачи начинаются с кода, подобного приведённому ниже

```
skb=alloc_skb(len+headspace, GFP_KERNEL);
skb_reserve(skb, headspace);
skb_put(skb,len);
memcpy_fromfs(skb->data,data,len);
pass_to_m_protocol(skb);
```

В системах типа BSD Unix не нужно заранее знать объем требуемого пространства по причине использования цепочек мелких буферов (mbuf). В Linux используются линейные буферы и упреждающее резервирование места (зачастую несколько байтов в худшем случае), поскольку линейные буферы позволяют выполнять многие операции гораздо быстрее.

Linux поддерживает несколько функций для работы со списками, перечисленных ниже.

- **skb_dequeue()** берет первый буфер из списка. Если список пуст, функция возвращает указатель **NULL**. Эта функция используется для вытягивания буферов из очередей. Буферы добавляются функциями **skb_queue_head()** и **skb_queue_tail()**.
- **skb_queue_head()** помещает буфер в начало списка. Как и все операции со списками, она является неделимой.
- **skb_queue_tail()** помещает буфер в конец списка и является наиболее часто используемой функцией. Почти все очереди обслуживаются одним набором функций размещения данных в очереди и другим набором для извлечения данных из тех же очередей с помощью **skb_dequeue()**.
- **skb_unlink()** удаляет буфер из содержащего его списка. Буфер не освобождается, а просто исключается из списка. Для упрощения некоторых операций не требуется знать в каком списке находится буфер и всегда можно вызвать **skb_unlink()** для буфера, который не находится ни в одном из списков. Эта функция позволяет сетевому коду исключить буфер из обработки даже в тех случаях, когда сетевой протокол на знает, кто в данный момент использует этот буфер. Обеспечивается отдельный механизм блокировки, который не позволяет удалить буфер, используемый в данный момент драйвером устройства.
- Некоторые более сложные протоколы типа TCP сохраняют порядок кадров и меняют порядок, в котором данные были получены. Две функции **skb_insert()** и **skb_append()** позволяют поместить **sk_buff** перед указанным буфером или после него.
- **alloc_skb()** создаёт новый буфер **sk_buff** и инициализирует его. Возвращаемый буфер готов к использованию, но предполагается, что будут заполнены некоторые поля, показывающие, как следует освобождать буфер. Обычно это делается с помощью **skb->free=1**. Буфер можно пометить как неосвобождаемый с помощью **kfree_skb()** (см. ниже).
- **kfree_skb()** освобождает буфер и при установленном **skb->sk** снижает значение счётчика использования памяти сокетом (**sk**). Инкрементирование этих счётчиков зависит от функций сокета и протокольного уровня для предотвращения освобождения сокета с сохраняющимися буферами. Учёт памяти очень важен, поскольку сетевые уровни ядра должны знать размер памяти, связанной с каждым соединением для того, чтобы предотвратить использование слишком большого объёма памяти удалёнными машинами или локальными процессами.
- **skb_clone()** делает копию **sk_buff**, но не копирует область данных, которая должна считаться доступной лишь для чтения.
- Иногда копия данных нужна для редактирования и функция **skb_copy()** обеспечивает такие же возможности, как **skb_clone**, но копирует также данные (что существенно увеличивает издержки).

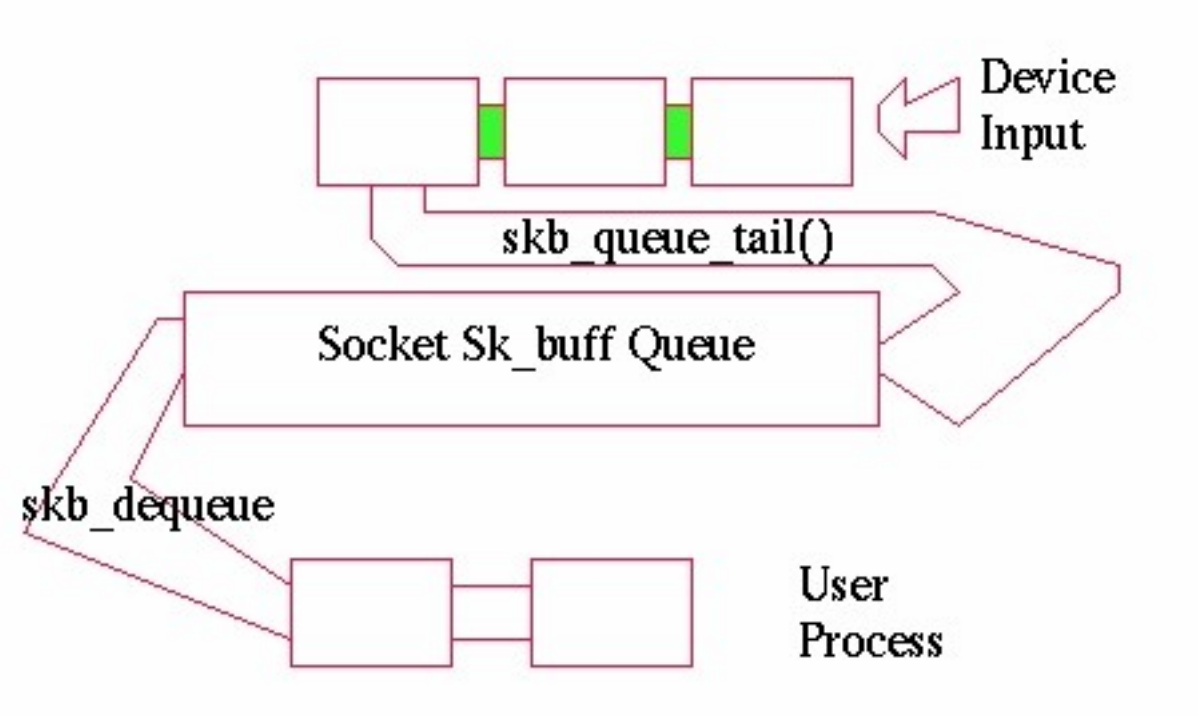

Рисунок 6. Поток пакетов.

Процедуры поддержки вышележащего уровня

Семантика выделения буферов и выстраивания их в очереди для сокетов включает также правила управления потоком данных и передачу всего списка взаимодействий с сигналами и дополнительными параметрами типа обеспечения неблокируемых операций. Созданы две процедуры для упрощения этого в большинстве протоколов.

Функция **sock_queue_rcv_skb()** служит для управления потоком входных данных и обычно используется в виде

```
sk=my_find_socket(whatever);
if(sock_queue_rcv_skb(sk,skb)==-1)
{
    myproto_stats.dropped++;
    kfree_skb(skb,FREE_READ);
    return;
}
```

Эта функция использует счётчики считывания сокетов, чтобы предотвратить попадание в сокет огромного объёма данных. После достижения предела данные отбрасываются. Приложение должно считывать данные достаточно быстро или, как в TCP, протокол должен контролировать поток данных через сеть. TCP действительно предлагает передающей стороне затормозиться, если больше нет возможности помещать данные в очередь.

На передающей стороне **sock_alloc_send_skb()** обеспечивает обработку сигналов, флаг неблокируемой обработки и всю семантику блокировки до тех пор, пока в очереди не появится место, чтобы не было возможности связать всю память с очередями медленного интерфейса. Функции передачи многих протоколов имеют такую процедуру, делающую почти всю работу

```
skb=sock_alloc_send_skb(sk,....)
if(skb==NULL)
    return -err;
skb->sk=sk;
skb_reserve(skb, headroom);
skb_put(skb,len);
memcpy(skb->data, data, len);
protocol_do_something(skb);
```

Большую часть этого мы уже встречали. Очень важна строка **skb->sk=sk**. Функция **sock_alloc_send_skb()** получает память для буфера сокета. Путём установки **skb->sk** мы говорим ядру, что тот, кто выполняет **kfree_skb()** над буфером, должен заручиться памятью для буфера буфера сокета. Таким образом, когда устройство передало буфер и освободило его, пользователь может передавать снова.

Сетевые устройства

Все сетевые устройства Linux используют общий интерфейс, но многие функции, доступные для интерфейса, нужны далеко не всем устройствам. При объектно-ориентированном подходе каждое устройство является объектом с набором методов, которые организованы в структуру. Каждый метод вызывается с указанием устройства в качестве первого аргумента для того, чтобы обойти отсутствие концепций C++ в языке C.

Файл drivers/net/skeleton.c содержит “скелет” драйвера сетевого устройства. Рассмотрите внимательно содержимое этого файла из свежей версии ядра и читайте статью дальше.

Базовая структура

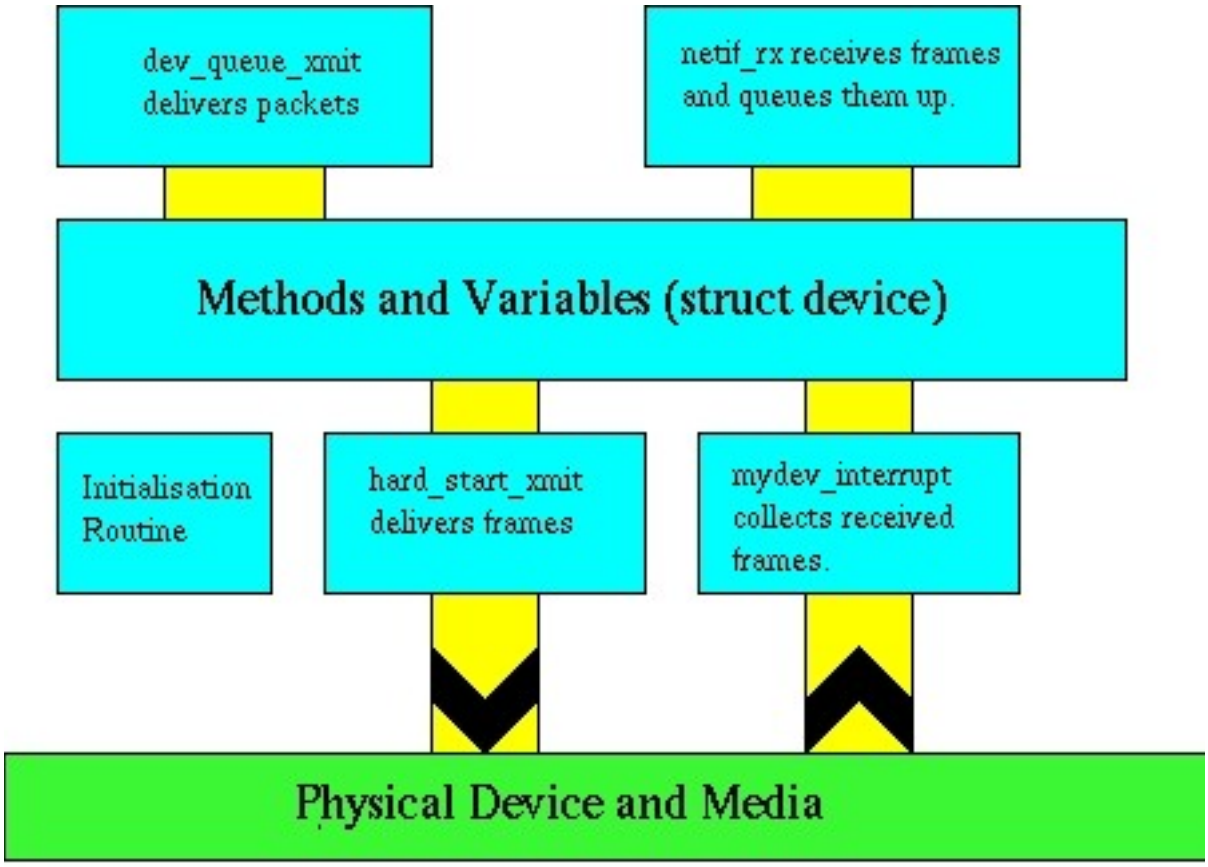

Рисунок 7. Структура сетевого устройства Linux.

Каждое сетевое устройство полностью берет на себя передачу сетевых буферов от протоколов в физическую среду, а также приём и декодирование откликов, генерируемых оборудованием. Входящие кадры помещаются в сетевые буферы, идентифицируемые протоколом, и доставляются функции **netif_rx()**. Эта функция затем передаёт кадры протокольному уровню для дальнейшей обработки.

Каждое устройство обеспечивает набор дополнительных методов для обслуживания остановки, запуска, управления и физической инкапсуляции пакетов. Вся управляющая информация собирается вместе в структурах устройства и служит для управления каждым устройством.

Именование

Все сетевые устройства Linux имеют уникальные имена, никак не связанные с именами, которые устройства могут иметь в файловой системе. Сетевые устройства обычно не представлены в файловой системе, хотя вы можете создать устройство, которое будет привязано к драйверам. Традиционно имя указывает лишь тип устройства, а не его производителя. Многочисленные однотипные устройства нумеруются с использованием суффиксов, начинающихся с 0 (например, устройства Ethernet могут называться eth0, eth1, eth3 и т. д.). Схема именования важна, поскольку она позволяет создавать программы и конфигурации систем в терминах “плат Ethernet”, не задумываясь о конкретном производителе этих плат и смене конфигурации при замене платы.

Ниже перечислены имена, обычно используемые в качестве базовых:

- eth*n* для плат Ethernet;
- tr*n* для адаптеров Token ring;
- sl*n* для устройств SLIP и AX.25 KISS;
- ppp*n* для синхронных и асинхронных устройств PPP;
- plip*n* для модулей PLIP (номер соответствует номеру принтерного порта);
- tunl*n* для туннелей IPIP;
- nr*n* для виртуальных устройств NetROM;
- isdn*n* для устройств ISDN, обслуживаемых isdn4linux[1];
- dummy*n* для Null-устройств;
- lo для петлевого (loopback) устройства.

В некоторых физических уровнях присутствует множество логических интерфейсов для одной среды. Это свойство присуще ATM и Frame Relay, а также multi-drop KISS в среде любительской радиосвязи. В таких случаях требуется драйвер для каждого активного канала. Сетевой код Linux структурирован так, чтобы обеспечить для таких случаев управляемость без избыточного дополнительного кода. Схема регистрации имён позволяет создавать и удалять интерфейсы по мере активизации и отключении каналов. Предложенное соглашение по именованию продолжает обсуждаться, поскольку простая схема типа sl0a, sl0b, sl0c работает для базовых устройств KISS, но не подходит для соединений Frame Relay, где виртуальный канал может переноситься на другой интерфейс.

Регистрация устройства

Каждое устройство создаётся путём заполнения объекта **struct device** и передачи его функции **register_netdev(struct device *)**. Это связывает структуру device с таблицами сетевых устройств ядра. Поскольку переданная структура будет использоваться ядром, её нельзя освободить без дерегистрации устройства с помощью **void unregister_netdev(struct device *)**. Эти вызовы обычно происходят при загрузке системы или загрузке/выгрузке модуля.

Ядро не будет препятствовать созданию множества одноимённых устройств, оно просто “сломается”. Поэтому если драйвер является загружаемым модулем, следует использовать вызов **struct device *dev_get(const char *name)** для проверки того, что имя ещё не используется. При обнаружении конфликта следует использовать **unregister_netdev()** для дерегистрации использующего нужное имя устройства!

Типичный код будет иметь вид

```
int register_my_device(void)
{
  int i=0;
  for(i=0;i<100;i++)
  {
    sprintf(mydevice.name,"mydev%d",i);
    if(dev_get(mydevice.name)==NULL)
    {
      if(register_netdev(&mydevice)!=0)
        return -EIO;
      return 0;
    }
  }
  printk("100 mydevs loaded. Unable to load more.\n");
  return -ENFILE;
}
```

[1]По крайней мере один интерфейс ISDN является имитатором Ethernet – драйвер Sonix PC/Volante ведёт себя во всех аспектах как устройство Ethernet, а не ISDN, поэтому для него используется имя устройства eth. По возможности для новых устройств следует выбирать имена в соответствии со сложившейся практикой. При добавлении совершенно нового типа физического уровня следует ознакомиться с практикой других людей, работающих в похожих проектах, и использовать общую схему именования.

Структура устройства

Вся базовая информация и методы для каждого сетевого устройства хранятся в структуре device. Для создания устройства нужно представить структуру с данными, описанными ниже.

Именование

Во-первых, поле name должно содержать указатель на строку имени устройства в формате, рассмотренном выше. Поле name может также просто содержать 4 символа пробела и в этом случае ядро будет автоматически назначать имя вида eth*n*. Однако этот специальный случай не следует использовать. После Linux 2.0 для этой цели планируется добавить простую функцию поддержки типа **dev_make_name("eth")**.

Параметры шинного интерфейса

Следующий блок параметров используется для поддержки местоположения устройства в адресном пространстве архитектуры. Поле **irq** указывает используемое устройством прерывание (IRQ) и обычно задаётся в процессе загрузки или функцией инициализации. Если прерывание не используется, не известно или не задано, следует указывать значение 0. Прерывание может быть установлено разными способами. Можно использовать функции ядра auto-irq для проверки (зондирования) прерывания или установить прерывание во время загрузки модуля. Сетевые драйверы обычно используют для этого глобальную целочисленную переменную **irq** и пользователь может загрузить модуль с помощью команды вида **insmod mydevice irq=5**. Кроме того, IRQ можно устанавливать динамически с помощью команды **ifconfig**, которая будет обращаться к устройству, как описано ниже.

Поле **base_addr** указывает базовый адрес ввода-вывода (I/O), по которому размещается устройство. Если устройство не использует адрес I/O или работает в системе без поддержки адресации I/O, в этом поле следует указать 0. Если адрес может устанавливаться пользователем, это обычно делается через глобальную переменную **io**. Адрес ввода-вывода для интерфейса можно также установить с помощью команды **ifconfig**.

Определены два диапазона общей памяти для устройств типа ISA-адапетров Ethernet, работающих через общую память. Для текущих целей поля **rmem_start** и **rmem_end** являются устаревшими и для них следует устанавливать значения 0. В полях **mem_start** и **mem_end** следует указывать начальный и конечный адрес блока общей памяти, используемого устройством. Если общая память не используется устройством, в полях следует указать значение 0. Такие устройства позволяют пользователю установить базовый адрес памяти с помощью глобальной переменной **mem**, а затем задать подходящее значение **mem_end**.

Переменная **dma** указывает канал прямого доступа к памяти (DMA), используемый устройством. Linux поддерживает для DMA автоматическое определение (как для прерываний). Если канал DMA не используется или ещё не задан, устанавливается значение 0. Это значение может меняться, поскольку новые системные платы ПК разрешают платам использовать канал 0 шины ISA, а не просто связывать его с обновлением памяти. Если пользователь может устанавливать канал DMA, для этого служит глобальная переменная **dma**.

Важно понимать, что физическая информация используется для контроля и просмотра пользователем (а также внутренними функциями драйвера) и эти области не регистрируются для предотвращения их использования другими. Таким образом, драйвер устройства должен выделить и зарегистрировать I/O, DMA и прерывание, которые он хочет использовать, применяя для этого те же функции, что и любой драйвер устройства (см. свежие статьи Kernel Korner по созданию драйверов устройств в выпусках 23, 24, 25, 26 и 28 или Linux Kernel Hackers' Guide на сайте www.redhat.com:8080/HyperNews/get/khg.html[1], *прим. редактора*).

В поле **if_port** указывается тип физической среды для устройств типа комбинированных плат Ethernet.

Переменные протокольного уровня

Для нормальной работы протоколов сетевого уровня устройство обеспечивает набор флагов и переменных возможностей (capability), который также поддерживается в структуре.

Поле **mtu** задаёт максимальный размер данных (payload) которые могут быть переданы через интерфейс, т. е. наибольший размер пакета без учёта заголовков нижележащего уровня, которые будет добавлять само устройство. Это значение используется протоколами типа IP для выбора подходящего размера пакетов при передаче. Каждый протокол задаёт своё минимальное значение для этого параметра. Устройство не применимо для протокола IPX без поддержки кадров размером не менее 576 байтов. IP требует не менее 72 байтов и не имеет смысла использовать этот протокол при размерах меньше примерно 200 байтов. Вопрос взаимодействия с устройством решается на уровне протокола.

Поле **family** всегда имеет значение **AF_INET** и указывает семейство протоколов, используемых устройством. Linux позволяет устройству одновременно использовать протоколы разных семейств и поддерживает эту информацию лишь для внешнего сходства со стандартным сетевым API систем BSD.

Значение поля типа оборудования **type** берётся из таблицы типов физических сред. Значения, используемые протоколом ARP (см RFC1700[2]), применяются для сред, которые поддерживают ARP, а для других физических уровней заданы свои значения. Новые значения при необходимости добавляются в ядро и пакет **net-tools**, включающий программы типа **ifconfig**, которые служат для декодирования этого поля. Значения, определённые в Linux pre2.0.5, приведены ниже.

Взяты из RFC1700

```
ARPHRD_NETROM       NET/ROM™
ARPHRD_ETHER        Ethernet 10 и 100 Мбит/с
ARPHRD_EETHER       Experimental Ethernet (не используется)
ARPHRD_AX25         AX.25 level 2
```

[1]Приведённая в оригинальной публикации ссылка устарела. Документ доступен здесь. *Прим. перев.*
[2]В соответствии с RFC 3232 этот документ утратил силу. Представленная в нем информация со всеми более поздними дополнениями доступна по ссылке. *Прим. перев.*

```
ARPHRD_PRONET       PROnet token ring (не используется)
ARPHRD_CHAOS        ChaosNET (не используется)
ARPHRD_IEE802       сети 802.2 и в частности token ring
ARPHRD_ARCNET       ARCnet
ARPHRD_DLCI         Frame Relay DLCI
```

Определены в Linux

```
ARPHRD_SLIP         протокол SLIP
ARPHRD_CSLIP        SLIP с компрессией заголовков VJ
ARPHRD_SLIP6        SLIP с 6-битовым кодированием
ARPHRD_CSLIP6       SLIP с 6-битовым кодированием и сжатием заголовков
ARPHRD_ADAPT        интерфейс SLIP в адаптивном режиме
ARPHRD_PPP          интерфейс PPP (синхронный или асинхронный)
ARPHRD_TUNNEL       туннель IPIP
ARPHRD_TUNNEL6      туннель IPv6 over IP
ARPHRD_FRAD         устройство доступа Frame Relay (FRAD)
ARPHRD_SKIP         шифрованный туннель SKIP
ARPHRD_LOOPBACK     петлевой интерфейс (loopback)
ARPHRD_LOCALTLK     Localtalk
ARPHRD_METRICOM     Metricom Radio Network
```

Устройства, помеченные как неиспользуемые, имеют определённый для них тип, но не поддерживаются в net-tools. Ядро Linux обеспечивает дополнительную базовую поддержку для устройств Ethernet и Token Ring.

Поле **pa_addr** служит для указания IP-адреса при активизации устройства. Интерфейсы должны начинать работу со сброшенной переменной адреса. Поле **pa_brdaddr** служит для записи настроенного широковещательного адреса, **pa_dstaddr** указывает адрес другой стороны соединения «точка-точка», **pa_mask** указывает маску сети IP для интерфейса. Все эти поля могут инициализироваться значением 0. Поле **pa_alen** указывает размер адреса (в нашем случае IP) и его следует инициализировать значением 4.

Переменные канального уровня

Переменная **hard_header_len** указывает число байтов, которые нужны устройству в начале переданного ему буфера. Это значение не равно числу байтов физического заголовка, который будет добавляться, хотя обычно используется именно это число. Устройство может использовать это значение для организации заполнения в начале буфера.

В ядрах серии 1.2.x указатель **skb->data** определял начало буфера и такое заполнение не должно было использоваться. Это также означало для устройств с переменным размером заголовков необходимость выделять **max_size+1** байтов и поддерживать байт размера в начале, чтобы знать где начинается реальный заголовок (между заголовком и данными не должно быть пропуска). В Linux 1.3.x это было существенно упрощено. Такой подход гарантирует в начале буфера наличие нужного места. Вы можете использовать **skb_push()** подобающим образом, как было сказано в описании сетевых буферов.

Адреса физической среды (если они есть) помещаются в поля **dev_addr** и **broadcast**, представляющие собой массивы байтов. Адреса, размер которых меньше размера массива, выравниваются по левому краю. Поле **addr_len** служит для указания размера аппаратного адреса. В средах, где аппаратные адреса не используются, в этих полях следует указывать 0. Для остальных сред адрес должна указывать пользовательская программа. Утилита ifconfig позволяет устанавливать аппаратный адрес интерфейса. В этом случае его не требуется задавать изначально, но открытый код должен принимать меры, предотвращающие начало передачи устройством до установки его адреса.

Флаги

Набор флагов служит для указания свойств интерфейса. Некоторые из них служат для обеспечения совместимости и самостоятельной ценности не имеют. Флаги перечислены ниже.

- **IFF_UP** - интерфейс в данный момент активен. В Linux флаги IFF_RUNNING и IFF_UP обычно обрабатываются в паре, а два элемента используются для совместимости. Когда интерфейс не имеет флага IFF_UP, он может быть удалён. В отличие от BSD интерфейс без флага IFF_UP никогда не получает пакетов.
- **IFF_BROADCAST** - интерфейс поддерживает широковещание. Действующий IP-адрес интерфейса хранится в адресах устройства.
- **IFF_DEBUG** - желательна отладка. В настоящее время не используется.
- **IFF_LOOPBACK** - этот флаг может устанавливаться только для петлевого (lo) интерфейса. Установка для других интерфейсов не определена и бессмысленна.
- **IFF_POINTOPOINT** - интерфейс канала «точка-точка» (типа SLIP или PPP). Здесь не поддерживается широковещание. Удалённый адрес «точка-точка» в структуре device допустим. Обычно канал «точка-точка» не имеет маски и широковещательного адреса, но они могут быть разрешены при необходимости.
- **IFF_NOTRAILERS** - «доисторический» флаг совместимости. Не используется.
- **IFF_RUNNING** - см. **IFF_UP**.
- **IFF_NOARP** - интерфейс не делает запросов ARP. Такой интерфейс должен иметь статическую таблицу преобразования адресов или выполнять отображения. Примером такого интерфейса может служить NetROM. Здесь все настраивается вручную, поскольку протокол NetROM не поддерживает запросов ARP.
- **IFF_PROMISC** - по возможности интерфейс будет «слушать» все пакеты в сети. Этот флаг обычно используется для мониторинга, хотя может быть полезен и для организации мостов. Один или два интерфейса типа AX.25 всегда находятся в неразборчивом режиме.

- **IFF_ALLMULTI** - принимать все групповые пакеты. Интерфейс, который не может выполнить такую операцию, но способен принимать все пакеты, при запросе такого режима будет переходить в неразборчивый режим.
- **IFF_MULTICAST** - указывает, что интерфейс поддерживает групповой трафик IP, что отличается от группового трафика на аппаратном уровне. Например, AX.25 поддерживает групповую адресацию IP с помощью аппаратного широковещания. Интерфейсы «точка-точка» типа SLIP обычно поддерживают групповую адресацию IP.

Очередь пакетов

Пакеты выстраиваются на интерфейсе в очередь протокольным кодом ядра. В каждом интерфейсе **buffs[]** представляет собой массив очередей пакетов для каждого уровня приоритета в ядре. Очереди целиком поддерживаются кодом ядра, но должны инициализироваться при активизации самого устройства. Код инициализации имеет вид

```
int ct=0;
while(ct<DEV_NUMBUFFS)
{
    skb_queue_head_init(&dev->buffs[ct]);
    ct++;
}
```

Все остальные поля устанавливаются в 0.

Устройство может выбрать нужный размер очереди путём установки в поле **dev->tx_queue_len** максимального числа пакетов, которые ядро сможет помещать в очередь устройства. Обычно это около 100 для Ethernet и 10 для последовательных линий. Устройство может динамически менять размер очереди, хотя происходить изменение будет с некоторым запаздыванием.

Методы сетевого устройства

Каждое сетевое устройство должно предоставить набор действующих функций (методов) для базовых операций низкого уровня. Оно должно также обеспечивать набор функций поддержки, которые связывают протокольный уровень с требованиями представляемого устройством канального уровня.

Установка

Метод **init** вызывается при инициализации устройств и его регистрации в системе для выполнения требуемых операций проверки на нижнем уровне. Метод возвращает код ошибки при отсутствии устройства, невозможности зарегистрировать области или иных проблемах. Если метод **init** возвратил ошибку, вызов **register_netdev()** возвращает код ошибки, а устройство в системе не создаётся.

Передача кадра

Все устройства должны поддерживать функцию передачи. Возможно существование устройств, не способных передавать. В таких случаях устройство должно поддерживать функцию передачи, которая просто освобождает буфер. Фиктивное устройство (dummy) реализует именно такую функцию передачи.

Функция **dev->hard_start_xmit()** вызывается вызывается для предоставления драйверу указателя на его устройство и сетевого буфера (**sk_buff**) для передачи. Если устройство не способно воспринять буфер, ему следует возвратить код 1 и установить для **dev->tbusy** ненулевое значение. Это действие поставит буфер в очередь для повтора попытки без гарантии такой попытки. Если протокольный уровень решит освободить буфер, отвергнутый драйвером, этот буфер не будет предложен устройству снова. Если устройство знает, что буфер не может быть передан в ближайшее время (например, по причине сильной перегрузки), оно может вызвать функцию **dev_kfree_skb()** для сброса (dump) буфера и вернуть код 0, показывающий, что буфер был обработан.

При наличии места буфер следует обработать. Обработанный буфер уже содержит все заголовки (включая заголовки канального уровня) и нужно лишь загрузить его в устройство для передачи. В дополнение к этому буфер блокируется и это означает, что драйвер устройства становится монопольным владельцем буфера, пока не решит от него отказаться. Содержимое **sk_buff** остаётся доступным только для чтения, за исключением гарантии того, что указатели на следующий и предыдущий буферы свободны, что позволяет использовать примитивы списка **sk_buff** для создания внутренней цепочки буферов.

Когда буфер загружен в устройство или (для некоторых устройств с DMA-управлением) устройство сообщило о завершении передачи, драйвер должен освободить буфер путём вызова **dev_kfree_skb(skb, FREE_WRITE)**. Как только этот вызов сделан, рассматриваемая структура **sk_buff** может исчезнуть в любой момент и драйверу устройства уже не следует использовать её снова.

Заголовки кадров

Необходимо, чтобы протоколы верхних уровней добавляли заголовки нижнего уровня в каждый кадр до его размещения в очереди на передачу. Однако протоколу явно нежелательно заранее знать, как добавить заголовки нижнего уровня ко всем возможным типам кадров. Таким образом, протокольный уровень обращается к драйверу, имея не менее **dev->hard_header_len** свободных байтов в начале буфера. Затем сетевое устройство должно корректно вызвать **skb_push()** и поместить заголовок в пакет с помощью вызова **dev->hard_header()**. Устройства без заголовка канального уровня (например, SLIP) могут использовать взамен этого метода NULL.

Метод вызывается путём предоставления соответствующего буфера, указателя устройства, отождествления протокола, указателей на аппаратные адреса отправителя и получателя, а также размера передаваемого пакета. Поскольку программу можно вызвать до полной сборки протокольных уровней, важно, чтобы метод использовал параметр length, а **не** размер буфера.

Адрес отправителя может иметь значение NULL, что означает использование принятого по умолчанию адреса данного устройства. Значение NULL в поле адреса получателя означает, что адрес неизвестен. Если по причине неизвестности адреса получателя заголовок не может быть завершён, следует выделить место и заполнить все поля, значения которых известны. Функция в таком случае должна возвратить число добавленных байтов со знаком минус. Это свойство в настоящее время используется лишь протоколом IP при обработке пакетов ARP. Если заголовок собран полностью, функция должна возвращать число байтов заголовка, добавленных в начале буфера.

Когда заголовок не может быть заполнен, протокольные уровни попытаются определить (resolve) необходимый адрес. При возникновении такой ситуации вызывается метод **dev->rebuild_header()** с адресом, по которому размещён заголовок, рассматриваемым устройством, IP-адресом получателя и указателем на сетевой буфер. Если устройство способно определить адрес доступными средствами (обычно ARP), функция заполняет физический адрес и возвращает значение 1. Если адрес не может быть указан, функция возвращает 0 и попытка будет повторена, когда у протокольного уровня будут основания предполагать, что преобразование адреса возможно.

Приём

У устройства нет метода приёма, поскольку устройство само вызывает обработку таких событий. Для типового устройства прерывание уведомляет обработчик о готовности полного пакета к приёму. Устройство выделяет буфер подходящего размера с помощью функции **dev_alloc_skb()** и считывает в него байты принятого пакета из оборудования. Затем драйвер устройства анализирует кадр для определения типа пакета. Драйвер устанавливает в **skb->dev** указатель на принявшее кадр устройство. В поле **skb->protocol** указывается протокол, представленный кадром, чтобы кадр мог быть передан нужному протокольному уровню. Указатель на заголовок канального уровня сохраняется в **skb->mac.raw**, а сам заголовок удаляется с помощью функции **skb_pull()**, поскольку протоколу он не нужен. В заключение для изоляции канального и протокольного уровня драйвер устройства должен установить в поле **skb->pkt_type** одно из следующих значений:

- **PACKET_BROADCAST** – широковещание канального уровня;
- **PACKET_MULTICAST** – групповая адресация канального уровня;
- **PACKET_SELF** – кадр для нас;
- **PACKET_OTHERHOST** – кадр для другого хоста.

Последний тип обычно указывается при работе интерфейса в неразборчивом (promiscuous) режиме.

В заключение драйвер вызывает **netif_rx()** для передачи буфера протокольному уровню. Буфер помещается в очередь для обработки сетевыми протоколами после возврата из обработчика прерывания. Отложенная обработка значительно снижает продолжительность интервала запрета прерываний и повышает общую «отзывчивость». После вызова **netif_rx()** буфер перестаёт быть свойством драйвера и больше не может быть изменён или переназначен драйвером.

Управление потоком принятых пакетов применяется протоколами на двух уровнях. Во-первых задаётся максимальное количество данных, которые могут ожидать **netif_rx()** для обработки. Во-вторых, каждый сокет в системе имеет очередь, которая ограничивает количество ожидающих данных. Таким образом, все управление потоком данных выполняется протокольными уровнями. На передающей стороне для устройства используется переменная **dev->tx_queue_len** в качестве ограничителя размера очереди. Размер очереди обычно составляет 100 кадров, что достаточно много, чтобы очередь заполнялась при передаче большого объёма данных через быстрые каналы. На медленных каналах (типа SLIP) очередь обычно рассчитана на 10 кадров, поскольку передача такого количества кадров занимает несколько секунд.

Некоторая «магия» на приёме данных в большинстве устройств, которую следует реализовать по возможности, заключается в резервировании в начале буфера байтов для заголовка IP так, чтобы он начинался на границе слова. В драйверах Ethernet это делается с помощью кода

```
skb=dev_alloc_skb(length+2);
if(skb==NULL)
    return;
skb_reserve(skb,2);
/* 14 байтов заголовка ethernet */
```

для выравнивания заголовков IP по 16-байтовой границе, которая также служит началом строки кэширования и это помогает повысить производительность. В системах SPARC и DEC Alpha эти улучшения очень заметны.

Дополнительная функциональность

Каждое устройство может предоставлять протокольным уровням дополнительные функции и возможности. Отсутствие таких функций не препятствует работе устройства, но может снижать качество обслуживания. Дополнительные операции делятся на две категории - настройка и включение/отключение.

Активация и отключение (Shutdown)

Когда устройство активировано (т. е. установлен флаг **IFF_UP**), вызывается метод **dev->open()**, если устройство поддерживает такой метод. Этот вызов позволяет устройству предпринять любые действия типа включения интерфейса для его использования. Ошибка, возвращаемая этой функцией, заставляет устройство остаться выключенным и пользовательский запрос завершается отказом с ошибкой, возвращённой **dev->open()**

Функцию **dev->open()** можно использовать и с устройствами, загружаемыми в форме модуля. Здесь требуется предотвращение возможности выгрузки модуля, когда устройство используется, поэтому с методом open должен использоваться макрос **MOD_INC_USE_COUNT**.

Метод **dev->close()** вызывается также в тех случаях, когда устройство готово к отключению, и оборудование следует выключать так, чтобы минимизировать нагрузку на машину (например, путём запрета интерфейса или возможности генерировать прерывания). Метод может также использоваться для выгрузки множества модульных устройств после их

отключения. Остальная часть ядра структурирована таким образом, что при закрытии устройства все указатели на него удаляются и устройство можно безопасно исключить из работающей системы. Метод close не может давать отказов.

Настройка и статистика

Набор функций обеспечивает возможность запрашивать и устанавливать параметры работы устройства. Функция **get_stats** при вызове возвращает структуру **enet_statistics** для интерфейса. Эта структура позволяет пользовательским программам типа **ifconfig** видеть загрузку интерфейса и сведения о вызвавших проблемы кадрах.

Функция **dev->set_mac_address()** вызывается всякий раз, когда процесс с правами суперпользователя вызывает операцию ioctl типа **SIOCSIFHWADDR** для изменения физического адреса устройства. Для многих устройств эта функция ничего не значит, а для некоторых просто не поддерживается. В таких случаях для этой функции просто используется указатель **NULL**. Некоторые устройства позволяют менять адрес только в отключённом состоянии. Для таких устройств нужно проверить флаг **IFF_UP** и, если он установлен, следует возвратить **-EBUSY**.

Функция **dev->set_config()** вызывается функцией **SIOCSIFMAP**, когда пользователь вводит команду типа **ifconfig eth0 irq 11**. При этом передаётся структура **ifmap**, содержащая данные ввода-вывода и другие параметры интерфейса. Для большинства интерфейсов эта функция бесполезна и она может возвращать NULL.

Функция **dev->do_ioctl()** вызывается при использовании для интерфейса операции ioctl из диапазона **SIOCDEVPRIVATE** - **SIOCDEVPRIVATE+15**. Все эти операции ioctl принимают структуру **ifreq**, которая копируется в пространство ядра перед вызовом обработчика и копируется обратно по завершении. Для обеспечения максимальной гибкости такие вызовы доступны любому пользователю и при необходимости код проверяет наличие прав суперпользователя. Например, драйвер PLIP использует такие вызовы для установки тайм-аутов параллельного порта, чтобы позволить пользователю точно настроить устройство plip на его машине.

Групповая адресация

Некоторые физические среды типа Ethernet поддерживают групповую адресацию на физическом[1] уровне. Кадр multicast слышен группе хостов сети (не обязательно всем), а не просто передаётся от одного хоста другому.

Возможности плат Ethernet существенно различаются и большинство из них попадает в одну из трёх категорий.

- Без фильтров multicast. Такие платы получают все групповые кадры или не получают их совсем. Это может быть неудобно в сетях с большим объёмом multicast-трафика типа групповых видеоконференций.
- Хэш-фильтры. В плату загружается таблица, задающая маску для нужных групп. Этот метод позволяет отфильтровать часть нежелательного группового трафика.
- Полная фильтрация. Большинство плат, поддерживающих полную фильтрацию, комбинируют этот вариант с двумя предыдущими, поскольку полный фильтр часто ограничен размером 8 или 16 записей.

Возможность программирования поддержки групповой адресации на интерфейсах Ethernet особенно важна. Некоторые протоколы Ethernet (особенно Appletalk и IP multicast) работают на основе групповой адресации Ethernet. К счастью большую часть этой работы выполняет ядро (см. файл net/core/dev_mcast.c).

Ядро включает код, поддерживающий списки физических адресов, которые интерфейсу следует разрешать для групповой адресации. Драйвер устройства может возвращать кадры, которые соответствуют большему количеству групповых адресов, если он не способен поддерживать полной фильтрации.

При изменении списка групповых адресов вызывается функция драйвера устройства **dev->set_multicast_list()**. Драйвер может тогда перезагрузить свои физические таблицы. Обычно это имеет вид

```
if(dev->flags&IFF_PROMISC)
    SetToHearAllPackets();
else if(dev->flags&IFF_ALLMULTI)
    SetToHearAllMulticasts();
else
{
    if(dev->mc_count<16)
    {
        LoadAddressList(dev->mc_list);
        SetToHearList();
    }
    else
        SetToHearAllMulticasts();
}
```

Есть небольшое число плат, которые могут работать только с индивидуальным адресом или в неразборчивом режиме. В этом случае драйвер при запросе групповой адресации переходит в неразборчивый режим и должен сам установить флаг **IFF_PROMISC** в поле **dev->flags**.

Чтобы помочь разработчикам драйверов, список групповых адресов всегда сохраняется действительным. Это упрощает многие драйверы, поскольку сброс при ошибке зачастую приводит к перезагрузке драйвером списков групповых адресов.

Функции поддержки Ethernet

Ethernet является одним из наиболее распространённых типов физических интерфейсов, которые могут обслуживаться. Ядро обеспечивает набор функций общего назначения для поддержки Ethernet.

[1]Точнее было бы сказать «на канальном». *Прим. перев.*

eth_header() является стандартным заголовком Ethernet для процедуры **dev-hard_header** и может применяться драйвером Ethernet. Вместе с функцией **eth_rebuild_header()** это обеспечивает все, что нужно для поиска ARP при заполнении заголовков Ethernet в пакетах IP.

Процедура **eth_type_trans()** ожидает, что будет загружен необработанный пакет Ethernet. Она анализирует заголовки и устанавливает **skb->pkt_type** и **skb→mac,** а также возвращает предлагаемое значение **skb->protocol**. Эта процедура обычно вызывается из обработчика прерываний приёмной части драйвера Ethernet для классификации пакетов.

eth_copy_and_sum() - финальная процедура поддержки Ethernet имеет достаточно сложное внутреннее устройство, но обеспечивает существенное повышение производительности для плат, отображаемых на память. Она обеспечивает копирование данных и подсчёт контрольных сумм при передаче данных из платы в **sk_buff** за один проход. Эта однопроходная операций избавляет от расходов на подсчёт контрольной суммы и существенно повышает пропускную способность IP.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru