Работа допущена к защите

Зав. кафедрой

\_\_\_\_\_\_\_\_\_И. Г. Черноруцкий

“\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_2011 г.

**ВЫПУСКНАЯ РАБОТА БАКАЛАВРА**

Тема: ***Разработка новой функциональности и тестирование беспроводной персональной сети***

**Направление:** 230100.62 - Информатика и вычислительная техника

Выполнил студент гр. 4084/11 \_\_\_\_\_\_\_\_П. В. Васильев

Руководитель \_\_\_\_\_\_\_\_Т. В. Коликова

# Реферат

# Содержание

[Реферат 3](#_Toc296608532)

[Содержание 5](#_Toc296608533)

[Список иллюстраций 8](#_Toc296608534)

[Используемые определения и сокращения 9](#_Toc296608535)

[Введение 10](#_Toc296608536)

[Актуальность темы 10](#_Toc296608537)

[Постановка цели и определение задач 12](#_Toc296608538)

[Краткое содержание работы 13](#_Toc296608539)

[1. Обзор предметной области 13](#_Toc296608540)

[1.1 Анализ существующих реализаций беспроводных персональных сетей 13](#_Toc296608541)

[1.2 Стек BlueZ 14](#_Toc296608542)

[1.2.1 Архитектура BlueZ 15](#_Toc296608543)

[1.2.2 Настройка BlueZ 18](#_Toc296608544)

[1.2.3 Запуск служб 21](#_Toc296608545)

[1.2.4 Настройка и запуск PAN 23](#_Toc296608546)

[1.2.5 Внутреннее устройство PAN 27](#_Toc296608547)

[2. Сбор и анализ требований к предлагаемому решению 29](#_Toc296608548)

[2.1 Анализ и выбор методов реализации PAN профайла для Bluetooth Linux 29](#_Toc296608549)

[2.1.1 Upcall 30](#_Toc296608550)

[2.1.2 Разделяемая память 31](#_Toc296608551)

[2.1.3 Сигналы 31](#_Toc296608552)

[2.1.4 Системные вызовы ядра 32](#_Toc296608553)

[2.1.5 Файловая система 33](#_Toc296608554)

[2.1.6 Управление вводом-выводом 37](#_Toc296608555)

[2.1.7 Сокеты 38](#_Toc296608556)

[3. Особенности реализации и архитектура PAN профайла 40](#_Toc296608557)

[3.1 Использование сокетов Netlink 40](#_Toc296608558)

[3.1.1 Прием и передача дейтаграмм 42](#_Toc296608559)

[3.1.2 Структура адреса сокета Netlink 42](#_Toc296608560)

[3.1.3 Формат сообщения Netlink 43](#_Toc296608561)

[3.1.4 Netlink заголовок 43](#_Toc296608562)

[3.1.4 Макросы Netlink 45](#_Toc296608563)

[3.2 Реализация сетевого драйвера 46](#_Toc296608564)

[3.2.1 Регистрация устройства 47](#_Toc296608565)

[3.2.2 Выгрузка модуля 48](#_Toc296608566)

[3.2.3 Открытие и закрытие 48](#_Toc296608567)

[3.2.4 Передача пакетов 50](#_Toc296608568)

[3.2.5 Контроль одновременной передачи 51](#_Toc296608569)

[3.2.6 Прием пакетов 52](#_Toc296608570)

[3.3 Работа демона пользовательского пространства 53](#_Toc296608571)

[3.3.1 Конечный автомат 55](#_Toc296608572)

[3.3.2 Разбор пакетов 56](#_Toc296608573)

[3.3.3 Буфер пакетов в драйвере 57](#_Toc296608574)

[3.4 Разработка тестового набора модульного тестирования 58](#_Toc296608575)

[3.4.1 Разбор полученных результатов тестирования 59](#_Toc296608576)

[4. Результаты 60](#_Toc296608577)

# Список иллюстраций

[Рис. 1 Применение PAN как точка доступа в Интернет 9](#_Toc296608578)

[Рис. 2 Архитектура BlueZ 16](#_Toc296608579)

[Рис. 3 Вывод ifconfig 23](#_Toc296608580)

[Рис. 4 Архитектура PAN профайла для Bluetooth Linux 40](#_Toc296608581)

[Рис. 5 Структура сообщения Netlink 45](#_Toc296608582)

[Рис. 6 Конечный автомат 55](#_Toc296608583)

[Рис. 7 Структуры пакетов Ethernet и BNEP 56](#_Toc296608584)

[Рис. 8 Передача пакетов в драйвере 57](#_Toc296608585)

# Используемые определения и сокращения

|  |  |
| --- | --- |
| Сокращение | Определение |
| PAN | Personal Area Network |
| BNEP | Bluetooth Network Encapsulation Protocol |
| BCSP | BlueCore Serial Protocol |
| CLI | Command Line Interface |
| ACL | Asynchronous Connectionless Link |
| MTU | Maximum Transmission Unit |
| SCO | Synchronous Connection Oriented |
| IP | Internet Protocol |
| DHCP | Dynamic Host Configuration Protocol |
| L2CAP | Logical Link Control and Adaptation Protocol |
| RFCOMM | Serial port transport protocol over L2CAP |
| HCI | Host Controller Interface |
| PCMCIA | Personal Computer Memory Card International Association |
| SDP | Service Discovery Protocol |
| GNU | GNU's Not UNIX |
| GPL | General Public License |
| PIN | Personal Identification Number |
| MAC | Media Access Control |
| TCP | Transmission Control Protocol |
| API | Application Programming Interface |
| UART | Universal Asynchronous Receiver/Transmitter |
| USB | Universal Serial Bus |
| NAP | Network Access Point |

# Введение

## Актуальность темы

Все больше применений находят беспроводным персональным сетям. Потому что их основным преимуществом является независимость от местоположения. Можно загрузить последние новости через RSS-ленту, выпивая чашку чая в кафе. Или передать файл между двумя компьютерами, не соединяя их кабелем.

Простота реализации и небольшие размеры приемопередатчика позволяют внедрять его во множество компьютерных устройств. Можно соединять между собой разного типа устройства, объединять их в сеть. Например, соединить планшетный ПК с маршрутизатором, который подключен к Интернету, и просматривать Web-страницы лежа на диване. Или соединить два ноутбука по беспроводной персональной сети и играть в сетевые игры.

Небольшой радиус действия беспроводных персональных сетей (примерно 17м) делает их идеальными для развертывания в небольшом помещении, например, дома или в автомобиле. Их можно применять для беспроводного соединения с принтером или беспроводной гарнитурой, для выхода в интернет с коммуникатора или синхронизации данных между телефоном и компьютером. Скорость передачи данных до 2МБит/с.

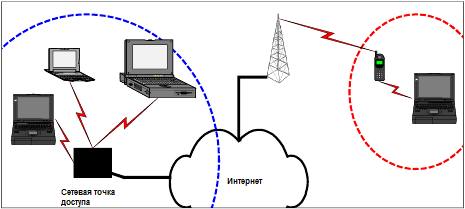


Рис. Применение PAN как точка доступа в Интернет

Еще одним преимуществом является низкое энергопотребление приемопередатчика, что и стало одной из основных причин внедрения беспроводных персональных сетей в телефоны, гарнитуры, планшеты.

Передача данных осуществляется с помощью радиосигналов. Радиосигналы — это электромагнитные волны, которые система связи использует для передачи информации через воздушную среду от одной точки к другой.

Спецификация Bluetooth является одной из самых распространенных реализаций беспроводной персональной сети.

Технология Bluetooth появилась в 1998 г. в результате совместной деятельности компаний Ericsson, IBM, Intel, Nokia и Toshiba. Они работали над созданием решения, обеспечивающего беспроводное взаимодействие компьютерных устройств. Спецификация Bluetooth, которая не считается стандартом, идеальным образом подходит для дешевых малогабаритных устройств с малой потребляемой мощностью и радиоканалами небольшого радиуса действия. Это делает технологию Bluetooth эффективным решением для соединения между собой малогабаритных устройств в небольших рабочих зонах. Именно поэтому Рабочая группа 802.15 выбрала Bluetooth в качестве основы для стандарта 802.15.1. Специальная группа по интересам (Special Interest Group, SIG) Bluetooth опубликовала первую версию спецификации в середине 1999 г. С тех пор она несколько раз дорабатывалась, но технические характеристики остались в основном прежними. Приемопередатчики Bluetooth работают в диапазоне 2,4 ГГц со скоростью до 1 Мбит/с с использованием технологии FHSS. Она предполагает непрерывное скачкообразное изменение частоты во всем отведенном для передачи спектре с частотой 1600 изменений в секунду, что намного быстрее, чем частота изменений, предусмотренная для аналогичной технологии в стандарте 802.11. Устройства Bluetooth с низким энергопотреблением обеспечивают дальность передачи около 9 м (30 футов). Высокомощные Bluetooth-устройства способны работать на расстояниях до 91 м (300 футов), однако такой режим работы применяется редко. Модули Bluetooth имеют относительно небольшие форм-факторы. Типичные габариты 10,2 х 14 х 16 мм, поэтому они могут быть легко встроены в различные пользовательские устройства.

Технология Bluetooth способна обеспечивать автоматическое соединение Bluetooth устройств, находящихся неподалеку одно от другого, но пользователь имеет возможность принять или отклонить возможность соединения с другим пользователем. В случае если есть сомнения в безопасности, соединение можно отклонить. Возможность шифрования также оговорена в спецификации.

**Протоколы и службы**

В Bluetooth используются различные протоколы. Все они образуют стек Bluetooth. Его можно разделить на 4 уровня:

1. Корневые протоколы: Baseband, LMP, L2CAP, SDP
2. Протоколы замены кабеля: RFCOMM
3. Протоколы управления телефонией: TCS-binary, AT-команды
4. Заимствованные протоколы: OBEX, PPP, UDP/TCP/IP и др.

**Профили**

Специальная рабочая группа Bluetooth SIG определила различные модели использования, каждая из которых сопровождается профилем. Профили определяют протоколы и функции, которые поддерживают определенные модели использования. Если устройства от различных производителей соответствуют одному профилю, определенному в спецификации Bluetooth, они смогут взаимодействовать.

Одним из таких профилей является PAN. Он позволяет организовать беспроводную персональную сеть через технологию Bluetooth. Профиль позволяет использовать протокол BNEP в качестве транспорта через Bluetooth-соединение.

## Постановка цели и определение задач

Целью данной работы является реализация простого в использовании и безопасного сервиса для создания персональной сети с удаленным устройством.

Для достижения заданной цели были решении следующие задачи:

1. Анализ существующих реализаций PAN профайла в Linux
2. Анализ и выбор методов реализации PAN профайла для Bluetooth Linux
3. Разработка тестового набора модульного тестирования для проверки работоспособности программы

Также к архитектуре PAN профайла были предъявлены следующие требования:

1. Он должен распространяться под проприетарной лицензией. Это означает, что минимальное количество кода должно находиться в модуле ядра, который обязательно распространяется под открытой лицензией GNU GPL.
2. Версия спецификации PAN 1.0
3. Версия спецификации BNEP 1.0

## Краткое содержание работы

Работа содержит 4 главы.

В первой главе будет проведен анализ существующих реализаций PAN профайла в ОС Linux. Во второй главе будут рассмотрены механизмы реализации PAN профайла для Bluetooth Linux. В третьей главе описаны подробные сведения о реализации и архитектуре разрабатываемого проекта. В 4 главе будут подведены итоги.

# 1. Обзор предметной области

## 1.1 Анализ существующих реализаций беспроводных персональных сетей

Будет производиться анализ PAN в операционных системах на базе ядра GNU/Linux.

В ОС Linux имеются следующие стеки Bluetooth:

1. BlueZ - открытая реализация Bluetooth стека, разработанная Qualcomm. Включен в официальное ядро GNU/Linux.
2. Affix, Bluetooth стек от Nokia Research Center. Разработка прекращена в 2005 году. На тот момент стек поддерживал спицификацию Bluetooth версии 1.1 и 1.2.
3. BlueSoleil - проприетарный стек, разработанный компанией IVT.

Так как стек Affix не разрабатывается уже более 6 лет, нет необходимости рассматривать его архитектуру.

Стек BlueSoleil распространяется под проприетарной лицензией, а значит, провести анализ его архитектуры не представляется возможным.

Таким образом, остается единственный стек, архитектуру которого можно рассмотреть, BlueZ.

## 1.2 Стек BlueZ

BlueZ — канонический стек Bluetooth для Linux. Стек BlueZ поддерживает все основные протоколы и уровни Bluetooth. Был первоначально разработан Qualcomm, и доступен для ядра Linux версии 2.4.6 и выше.

BlueZ имеет модульную архитектуру, ее преимущества:

* Полная модульная реализация
* Безопасность при симметричной многопроцессорной обработке
* Многопоточная обработка данных
* Поддержка нескольких устройств Bluetooth
* Реальная аппаратная абстракция
* Стандартный интерфейс сокетов для всех слоев
* Безопасная поддержка уровней устройств и сервисов

Модули, из которых состоит BlueZ:

* Ядро подсистемы Bluetooth
* L2CAP и SCO на уровне ядра Linux
* RFCOMM, BNEP, CMTP, HIDP в ядре Linux
* HCI UART, PCMCIA, USB и драйвера виртуальных устройств
* Общие Bluetooth и SDP библиотеки и демоны
* Конфигурационные и тестовые утилиты
* Протокол декодирования и аналитические инструменты

BlueZ поддерживает многие архитектуры, которые поддерживает Linux:

* Intel и AMD x86
* AMD64 и EM64T (x86-64)
* SUN SPARC 32/64bit
* PowerPC 32/64bit
* Intel StrongARM и XScale
* процессоры Hitachi/Renesas SH
* Motorola DragonBall

Поддерживается BlueZ во многих Linux-дистрибутивах и в целом совместим с любой системой Linux:

* Debian GNU / Linux
* Ubuntu Linux Ubuntu Linux
* Fedora Core / Red Hat Linux Fedora Core / Red Hat Linux
* OpenSuSE / SuSE Linux OpenSuSE / SuSE Linux
* Mandrake Linux Mandrake Linux
* Gentoo Linux Gentoo Linux

### 1.2.1 Архитектура BlueZ

BlueZ подразделяется на отдельные модули как в ядре, так и в пользовательском пространстве.

На самом нижнем уровне в ядре Linux драйвера отвечают за работу с самими Bluetooth-передатчиками. Этот уровень драйверов поддерживает различные протоколы передачи данных: UART и его различные реализации, например BCSP или H4 (H4 - промышленный стандарт, который требует управления потоком данных, в то время как BCSP - пропиетарный протокол, который работает точно также без управления потоком данных.), PCMCIA, USB. Все драйвера представляют общий API для взаимодействия с верхним уровнем. Такая архитектура дает возможность безопасно работать с драйверами, позволяя исправлять и добавлять новые протоколы для работы с Bluetooth-передатчиками не затрагивая верхние слои, а также дает возможность работать напрямую с Bluetooth-устройствами другим модулям, в обход верхних слоев.

Уровнем выше, так же в пространстве ядра, находится слой, который предоставляет сокетный интерфейс для взаимодействия пространства пользователя и ядра. На этом уровне происходит обработка пакетов для передачи и после приема от драйверов нижнего уровня. Для верхних уровней реализован сокетный интерфейс 4ех типов протоколов: HCI, L2CAP, SCO и RFCOMM. Для подключения к этим сокетам необходимо указать семейство (домен) коммуникации - AF\_BLUETOOTH. Каждый домен выбирает набор протоколов для коммуникации. Домен AF\_BLUETOOTH определяет соответственно 4 протокола: BTPROTO\_HCI, BTPROTO\_L2CAP, BTPROTO\_SCO и BTPROTO\_RFCOMM. Каждый протокол работает по типу SOCK\_STREAM, который обеспечивает надежные, двунаправленные потоки байтов, с поддержкой соединения. Так же реализованы драйвера для протоколов BNEP, CMTP и HID. Для каждого протокола также имеется свой сокетный протокол для работы с верхним уровнем: BTPROTO\_BNEP, BTPROTO\_CMTP и BTPROTO\_HID соответственно. Все протоколы также входят в семейство AF\_BLUETOOTH.

В пространстве пользователя находится демон hcid, который выполняет следующие функции:

1. Выполняет взаимодействие всех профайлов с нижним уровнем.
2. Предоставляет интерфейс по DBus для пользовательских приложений.
3. Управляет поиском устройств.
4. Устанавливает соединения с устройствами.

DBus используется в BlueZ для управления Bluetooth стеком из пользовательского пространства. Есть встроенные в графическое окружения приложения, такие как GNOME Bluetooth для графической среды Gnome, а также kdebluetooth и BlueDevil для KDE. Так же имеется интеграция BlueZ с файловыми менеджерами Nautilus и Konqueror для OBEX профайла. Пакет gnome-vfs-obexftpнеобходим для работы OBEX в графической среде Gnome. Network-Manager, который является менеджером сетевых соединений в Gnome, имеет поддержку управления PAN-соединениями.

Но основным средством управления BlueZ является CLI. Через командную оболочку можно произвести поиск устройств, настройку BlueZ, соединяться с другими устройствами и управлять профайлами. Это является несомненным плюсом BlueZ, так как позволяет использовать стек даже на серверах или встраиваемых устройствах, где применение графического интерфейса недопустимо.

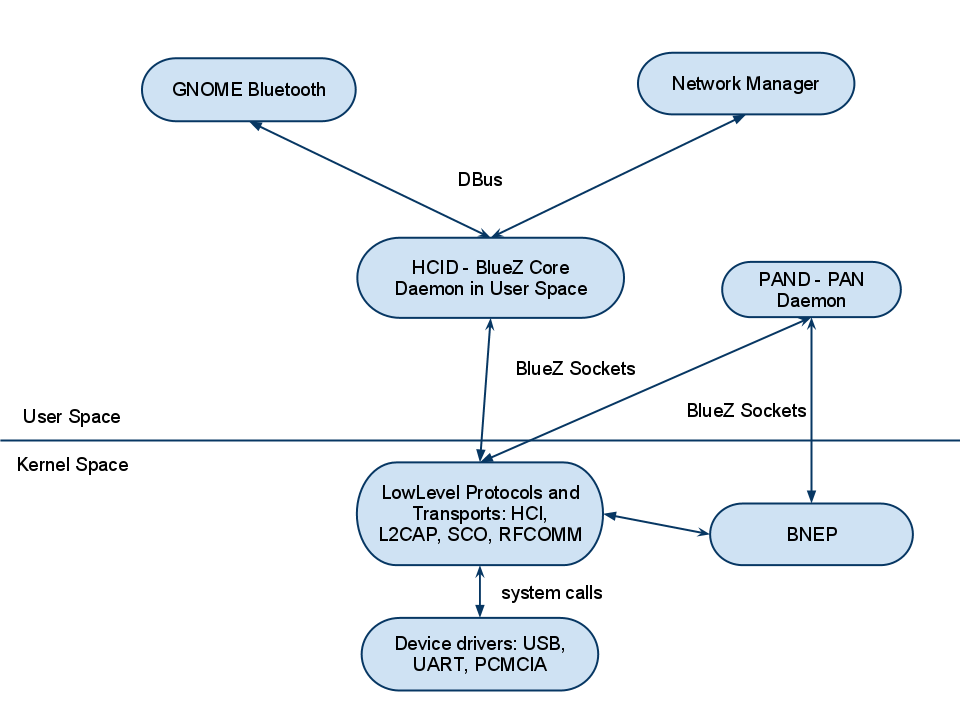


Рис. Архитектура BlueZ

### 1.2.2 Настройка BlueZ

С помощью программы hciconfig можно производить настройку устройств Bluetooth, подключенных к компьютеру. Следующая команда включит Bluetooth и покажет его основные параметры:

$ hciconfig hci0 up  
 hci0: Type: USB  
 BD Address: 00:02:72:28:C8:4A ACL MTU: 192:8 SCO MTU: 64:8  
 UP RUNNING PSCAN ISCAN AUTH ENCRYPT  
 RX bytes:2882377 acl:151624 sco:0 events:212 errors:0  
 TX bytes:2591 acl:184 sco:0 commands:20 errors:0

Из вывода команды hciconfig видно, что устройство Bluetooth подключено к USB-порту, имеет MAC-адрес 00:02:72:28:C8:4A, ACL MTU указывает количество MTU байт - 192 на пакет и размер буфера ACL, равное 8 пакетам, SCO MTU указывает количество MTU байт - 64 на пакет и размер буфера SCO, также равное 8 пакетам. Флаги указывают на то, что устройство включено (UP), работает (RUNNING), сканирование страниц включено (PSCAN), допрашивающее сканирование включено (ISCAN), аутентификация включена (AUTH), шифрование включено (ENCRYPT). Также из вывода можно увидеть статистическую информацию о работе устройства.

Программа hcitool позволяет работать с удаленными устройствами. С помощью нее можно произвести поиск удаленных устройств, получать их имена, качество сигнала, класс устройства, запросить аутентификацию и включить шифрование связи с удаленным устройством:

$ hcitool scan

Scanning ...

88:C6:63:26:61:14 iPod Pavel

Прежде, чем запускать демон hcid, необходимо произвести его настройку. Для этого надо отредактировать конфигурационный файл hcid.conf, находящийся в каталоге /etc/bluetooth:

options {  
 # automatically put new hciN devices up  
 autoinit yes;  
  
 # security mode  
 # none : security disabled  
 # auto : use local pin for incoming connections  
 # user : always ask a pin to the user  
 security auto;  
 pairing multi;  
 passkey “1234”;  
 }  
  
 device {  
 # whatever you want  
 name "PavelPC";  
   
 # 0x100 is fine for a PC, 0x120112 is better for a PDA  
 class 0x100;  
   
 # enable inquiry scan  
 iscan enable;  
  
 # enable page scan  
 pscan enable;  
  
 # default link mode  
 # none : no policy  
 # accept : always accept incoming connections  
 # master : becomes the master on incoming connections  
 # and deny role switch on outgoing connections  
 lm accept;  
   
 # default link policy  
 # none : no policy  
 # rswitch : allow role switch  
 # hold : allow hold mode  
 # sniff : allow sniff mode  
 # park : allow park mode  
 lp rswitch,hold,sniff,park;  
  
 # enable authentication  
 auth enable;  
  
 # enable encryption  
 encrypt enable;  
 }

В категории options, опцией autoinit указывается, что все подключаемые к компьютеру локальные Bluetooth устройства будут автоматически включены. security auto указывает, что будет использоваться заданный PIN для соединения с устройствами. Опция pairing multi дает возможность подключения сразу к нескольким устройствам.

Для создания соединения между двумя Bluetooth устройствами необходимо, чтобы у них был выставлен одинаковый PIN-код. Этот код может быть размером и 8 и 128 бит, но на сотовых телефонах или беспроводной гарнитуре длина этого кода может быть ограничена 4мя цифрами. Опцией passkey указывается PIN код, который будет использоваться для подключения к удаленным устройствам. Когда одно из устройств посылает PIN, добавив к ему случайное число, принимающее устройство сравнивает его с имеющимся и в случае успеха создается ключ связи для защиты данного подключения. Для того, чтобы предотвратить радиоперехват, необходимо использовать аутентификацию и шифрование.

Теперь рассмотрим категорию device. По умолчанию, устройства Bluetooth находятся в резервном режиме и периодически сканируют эфир. С page scan, устройство ожидает, что другие устройства войдут с ним в контакт посредством его кодового имени. Удаленное устройство должно знать это имя. inquiry scan - противоположный процесс; любое устройство может делать вызов, даже если оно не знает надлежащее кодовое имя. Класс позволяет вам искать любой тип устройства, когда вокруг слишком много устройств Bluetooth. Если необходимо скрыть устройство от других людей, то можно отключить сканирование или в hcid.conf, или вручную, после того, как устройство будет включено:  
  
 $ hciconfig hci0 iscan disable

Теперь можно включить BlueZ, выполнив следующую команду:

$ /etc/init.d/bluetooth start

### 1.2.3 Запуск служб

После того, как устройство было запущено и настроено, можно перейти к настройке службы, которая управляет профайлами. Для добавления служб необходимо использовать утилиту sdptool:

# sdptool add NAP  
# sdptool add DUN  
# sdptool add OPUSH  
# sdptool add SP

Также с помошью sdptool можно произвести поиск сервисов на удаленном устройстве:

# sdptool browse 00:A0:96:0A:D1:80  
Browsing 00:A0:96:0A:D1:80 ...  
Service Name: Network Access Point Service  
Service RecHandle: 0x10000  
Service Class ID List:  
 "Network access point" (0x1116)  
Protocol Descriptor List:  
 "L2CAP" (0x0100)  
 PSM: 15  
 "BNEP" (0x000f)  
 Version: 0x0100  
 SEQ16: 10 20 30 40  
Profile Descriptor List:  
 "PAN access point" (0x1116)  
 Version: 0x0100  
  
Service Name: Dial-Up Networking  
Service RecHandle: 0x10001  
Service Class ID List:  
 "Dialup Networking" (0x1103)  
 "Generic Networking" (0x1201)  
Protocol Descriptor List:  
 "L2CAP" (0x0100)  
 "RFCOMM" (0x0003)  
 Channel: 1  
Profile Descriptor List:  
 "Dialup Networking" (0x1103)  
 Version: 0x0100  
  
Service Name: OBEX Object Push  
Service RecHandle: 0x10002  
Service Class ID List:  
 "OBEX Object Push" (0x1105)  
Protocol Descriptor List:  
 "L2CAP" (0x0100)  
 "RFCOMM" (0x0003)  
 Channel: 4  
 "OBEX" (0x0008)  
Profile Descriptor List:  
 "OBEX Object Push" (0x1105)  
 Version: 0x0100  
  
Service Name: Serial Port  
Service Description: COM Port  
Service RecHandle: 0x10003  
Service Class ID List:  
 "Serial Port" (0x1101)  
Protocol Descriptor List:  
 "L2CAP" (0x0100)  
 "RFCOMM" (0x0003)  
 Channel: 1  
Language Base Attr List:  
 code\_ISO639: 0x656e  
 encoding: 0x6a  
 base\_offset: 0x100  
Profile Descriptor List:  
 "Serial Port" (0x1101)  
 Version: 0x0100

### 1.2.4 Настройка и запуск PAN

Демон pand позволяет создавать сетевое соединение через Bluetooth по протоколу BNEP. Для запуска pand необходимо выполнить:

# pand --listen --master --role NAP

Теперь демон pand запущен в режим прослушивания в роли NAP. Любое удаленное устройство Bluetooth, имеющее PAN профайл, может подключиться к этому устройству.

Когда соединение будет установлено, на компьютере появится сетевой интерфейс bnep0. Об этом можно узнать, запустив программу ifconfig.

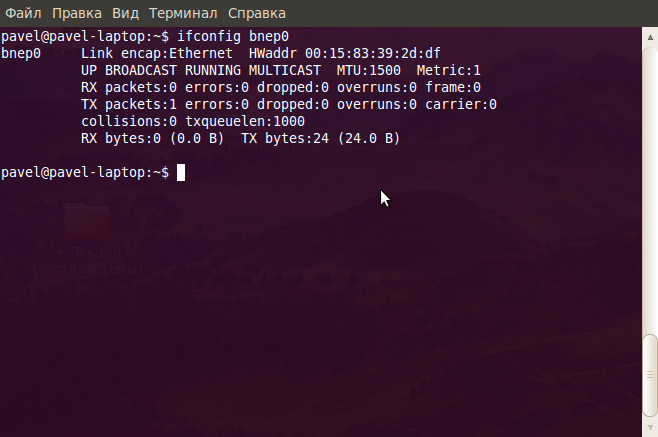


Рис. Вывод ifconfig

Настройку сетевого интерфейса bnep0 можно производить многими способами. Например, можно задать IP адрес вручную через утилиту ifconfig:

ifconfig bnep0 192.168.100.1

Автоматическую настройку IP адреса можно сделать, отредактировав файл interfaces, находящийся в каталоге /etc/network:

auto bnep0

iface bnep0 inet static

address 192.168.100.1

netmask 255.255.255.0

Если на удаленном устройстве установлен DHCP-сервер, то IP адрес будет выдан удаленным устройством автоматически. Если же самим установить DHCP-сервер, то можно будет выдавать IP адрес удаленным устройствам. Вот пример такой настройки DHCP-сервера:

subnet 192.168.100.0 netmask 255.255.255.0 {  
 max-lease-time 3600;  
 option subnet-mask 255.255.255.0;  
 default-lease-time 600;  
 option broadcast-address 192.168.100.255;  
 range 192.168.100.2 192.168.100.254;  
 option domain-name-servers 192.168.1.2;  
 option routers 192.168.1.2;  
}

Для проверки правильности настройки сетевого интерфейса, можно воспользоваться утилитой ping, которая отправляет ICMP запросы на заданный IP адрес. В случае правильной настройки, утилита будет отображать время прохождения сетевого пакета от локального компьютера до удаленного устройства и обратно:

Если же необходимо сделать доступ в интернет или к локальной сети, к которой подключен компьютер, то для этого есть программа iptables, позволяющая производить маршрутизацию пакетов.

Пример. Предположим, что на хосте имеется сетевой интерфейс eth0, с доступом в Интернет. Этот интерфейс имеет IP адрес 192.168.1.2. Тогда, чтобы любой другой сетевой интерфейс на хосте имел доступ в Интернет, нужно выполнить следующие команды:

# echo 1 > /proc/sys/net/ipv4/ip\_forward

# iptables -t nat -A POSTROUTING -o eth0 -j SNAT --to-source 192.168.1.2

Первая команда дает возможность перенаправления сетевых пакетов между интерфейсами. Вторая - делает подмену IP адреса в заголовке IP пакетов, исходящих из интерфейса eth0, жестко устанавливая IP адрес самого сетевого интерфейса.

Дело в том, что если не выполнить последнюю команду, пакеты от других сетевых интерфейсов, например, от bnep0, будут перенаправляться на eth0, а затем в Интернет с IP адресом интерфейса bnep0. Но, вне хоста информации о том, где находится компьютер с этим IP адресом, нет. Это определено сетевыми протоколами маршрутизации.

Когда пакет дойдет до получателя, тот отправит его назад по тому же пути, по которому пришел пакет. Таким образом, пакет дойдет ближайшего к нашему хосту сетевого узла. Однако, этот сетевой узел увидит IP адрес нашего интерфейса bnep0, а не eth0, и не поймет, куда послать пакет, так как не имеет подключенных к себе компьютеров с таким IP адресом.

На самом деле, то, что делает команда iptables, используется при маршрутизации пакетов по сети в маршрутизаторах. Так что в этой команде нет ничего, что могло бы нарушить маршрутизацию пакетов на хосте.

Если сетевой интерфейс eth0 имеет динамический IP адрес, то последнюю команду можно переписать так:

# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE

Различие между SNAT и MASQUERADE состоит в том, что SNAT работает быстрее и более эффективно, однако, если сетевой интерфейс не имеете постоянного IP адреса, то надо использовать действие MASQUERADE, которое предоставляет более простой способ трансляции адресов, поскольку оно автоматически определяет IP адрес, присвоенный заданному интерфейсу. Однако, по сравнению с SNAT это действие требует несколько больших вычислительных ресурсов, хотя и не значительно.

Есть другой способ настройки доступа в Интернет, который применяется для маршрутизации пакетов на 2ом уровне модели OSI и называется сетевой мост. Этот способ предлагается при настройки pand самими разработчиками BlueZ. Пакет, который предоставляет такую возможность в Linux, называется bridge-utils. Пример: пусть имеем ситуацию из примера выше. Тогда, для настройки сетевого моста между bnep0 и eth0, нужно выполнить следующие команды:

# brctl addbr pan0

# brctl addif pan0 eth0

# brctl addif pan0 bnep0

# ifconfig pan0 up

Первой командой создается сетевой мост pan0. Следующие две команды добавляют в сетевой мост pan0 интерфейсы eth0 и bnep0. Последняя команда запускает сетевой мост.

### 1.2.5 Внутреннее устройство PAN

PAN состоит из двух компонент: pand - демон, работающий из пользовательского пространства и модуль ядра - bnep.

Работа демона pand минимальна. Это подключение и отключение от удаленного устройства, настройка шифрования и аутентификации, просмотр активных подключений. В основном он лишь предоставляет интерфейс взаимодействия пользователя с модулями ядра BlueZ. Для “общения” с ядром используется два протокола из домена AF\_BLUETOOTH: BRPOTO\_L2CAP и BTPROTO\_BNEP. Первый протокол используется для создания соединения по L2CAP. BTPROTO\_BNEP является управляющим соединением для создания/удаления сетевого интерфейса и конфигурации протокола BNEP между устройствами. Как только соединение по L2CAP установлено, по протоколу BRPOTO\_BNEP передается ioctl о том, что надо провести конфигурацию протокола BNEP, и создается сетевой интерфейс. Всего таких ioctl четыре: добавить соединение, удалить соединение, получить список соединений, получить информацию о соединении.

Таким образом, вся реализация протокола BNEP находится в модуле ядра. Здесь можно выделить как плюсы, так и минусы. Плюсы:

1. Скорость обработки пакетов. Так как вся реализация протокола BNEP находится в ядре, то это дает прирост скорости в обработке пакетов, потому что код, находящийся в ядре работает быстрее, чем приложения из пользовательского пространства.
2. Нет передачи пакетов из модуля ядра в пространство пользователя. Демон pand лишь управляет подключениями, но не обрабатывает пакеты. А значит не нужно реализовывать дополнительные буферы хранения пакетов.
3. Передача пакетов между сетевым интерфейсом и драйвером Bluetooth происходит в ядре. Если бы обработка пакетов была не в ядре, а в пространстве пользователя, то необходимо было бы использовать дополнительный канал коммуникации для передачи пакета обратно в ядро.

Минусы:

1. Проблема отладки. Любой неправильный код в ядре может приводить к его смерти, т.е. Kernel Panic. Таким образом, отладка требует дополнительных ресурсов. Можно пересобрать ядро с поддержкой отладочных модулей. Или настроить удаленную отладку. Но все это может не спасти при серьезных ошибках, так как отловить момент, при котором ядро сломалось, будет очень затруднительно.
2. Увеличение количества кода в ядре. Это может привести к нестабильности самого ядра. Любая ошибка может вызвать Kernel Panic, привести к утечке памяти или внести уязвимость в ядро. А отладить такое, как уже было сказано выше - очень сложно.
3. Дополнительная нагрузка на ядро. Такая нагрузка может привести, например, к уменьшению производительности системы, если модуль будет отбирать большое количество ресурсов ядра.

Эти минусы довольно существенны. И от качества кода, написанного для BlueZ, очень зависит стабильность ядра и операционной системы. Но ведь из плюсов видно, что они и породили эти минусы.

Если вынести всю обработку пакетов из ядра в пространство пользователя, а ядро использовать лишь для взаимодействия с драйвером Bluetooth и сетевым интерфейсом, то минусы можно преодолеть. Тогда можно будет снизить риск возникновения ошибок в ядре к минимуму. Очевидно, что тогда произойдет потеря в производительности, но насколько сильно это скажется на самой передачи данных между беспроводными устройствами - не известно. Возможно, эта потеря производительности будет незаметна, зато при этом повысится безопасность операционной системы.

# 2. Сбор и анализ требований к предлагаемому решению

## 2.1 Анализ и выбор методов реализации PAN профайла для Bluetooth Linux

Прежде чем выбирать метод реализации PAN профайла, были поставлены следующие ограничения:

1. Код не должен быть открытым. Любой модуль ядра распространяется под лицензией GNU GPL 2. Однако разрабатываемый модуль будет проприетарным.
2. Стек Bluetooth располагается в пространстве пользователя. Это значит, что передача сетевых пакетов будет осуществляться от демона пользовательского пространства в ядро, а не полностью в ядре, как это сделано в BlueZ.

Первое ограничение означает, что обработка пакетов должна происходить в демоне пространства пользователя, так как его можно будет распространять под проприетарной лицензией. Так как сетевой интерфейс все равно необходимо будет реализовать в виде драйвера, то работа этого драйвера должна заключаться только в передаче пакетов в сетевой стек Linux, а также в приеме пакетов и отправке их демону.

Второе ограничение говорит о том, что пакеты от стека демон будет получать из пространства пользователя. Это означает, что не надо будет реализовывать интерфейс взаимодействия между драйвером Bluetooth и демоном.

Для работы со стеком уже имеется API в виде статической библиотеки. Этот API работает со стеком через TCP сокеты. Демон же просто использует функции, предоставляемые этим API, ничего не подозревая о том, как они устроены.

Главным вопросом является метод реализации передачи пакетов от демона в сетевой драйвер, т.е. взаимодействие между Kernel Space и User Space. Таких методов в ОС Linux 7:

1. Файловая система.
2. Сокеты.
3. Управление вводом-выводом (I/O Control).
4. Системные вызовы ядра.
5. Сигналы.
6. Upcall.
7. Разделяемая память.

### 2.1.1 Upcall

Upcall-функциональность ядра GNU/Linux позволяет модулю вызывать программы из пространства пользователя, передавая параметры командной строки и переменные окружения. Делается с помощью вызова следующей функции:

int call\_usermodehelper(char \*path, char \*\*argv, char \*\*envp, enum umh\_wait wait)

* path: путь до пользовательской программы
* argv: передаваемые аргументы командной строки
* envp: передаваемые переменные окружения
* umh\_wait: перечесляемый тип задает, что должен делать модуль после вызова этой функции:
* UMH\_NO\_WAIT: продолжить работу не дожидаясь запуска программы
* UMH\_WAIT\_EXEC: ожидать завершения запуска программы, но не ее завершения
* UMH\_WAIT\_PROC: ожидать завершения работы программы

Как видно, это не тот механизм, которым можно воспользоваться для передачи данных между модулем и демоном.

### 2.1.2 Разделяемая память

Отображение памяти является единственным способом для передачи данных между пользователем и пространством ядра, который не предполагает явного копирования, и это самый быстрый способ для работы с большими объемами данных.

Существует важное различие между обычными функциями read, write и функцией mmap. Во время передачи данных mmap нет “контроля” изменения данных. Это означает, что процесс пользовательского пространства может помещать данные в память, но ядро ​​не будет знать, что новые данные поступили. То же самое происходит и в обратном сценарии: ядро помещает свои данные в разделяемую память, но процесс пользовательского пространства не получает уведомление об этом событии. Это свойство означает, что механизм разделяемой памяти должен быть использован с другими средствами связи, которые передают управляющее сообщение, или разделяемая память должна проверяться через определенные промежутки времени на новые данные. Как read и write, mmap вызов может быть использован с разными файловыми системами и с сокетами.

Безусловно, необходимо какое-то количество памяти, для передачи данных между ядром и пользовательской программой. Для этого можно использовать, например, debugfs. Программа пользовательского пространства использует системные вызовы open, mmap, memcpy и close для работы с разделяемой памятью.

Разделяемая память является интересным решением, так как предоставляет самый быстрый механизм передачи данных между ядром и пользователем, но необходимость в управляющем канале сводит на нет все преимущество быстрой передачи данных. Таким образом, этот механизм взаимодействия между Kernel Space и User Space тоже не подходит.

### 2.1.3 Сигналы

Этот подход несколько отличается от других, так как только ядро может посылать сигнал в пользовательское пространство, но не наоборот. Кроме того, объем данных для передачи, весьма ограничен.

Есть два типа сигнала API, в пространстве пользователя: "нормальные" сигналы, которые не имеют каких-либо данных, и сигналы "реального времени", которые несут в себе 32 бита данных. Основное различие между ними в том, что сигналы реального времени передаются по очереди, в то время как нормальные сигналы - нет. Это означает, что если более чем один нормальный сигнал поступает в процесс прежде, чем он способен обработать его, он получает этот сигнал только один раз, в то время как сигналы реального времени процесс получит все.

Процесс пользовательского пространства регистрирует функцию обработчик сигнала ядра. Это добавляет адрес функции обработчика сигнала к дескриптору процесса. Эта функция запускается на выполнение каждый раз, когда доставляется определенный сигнал.

Фаза отправки сигнала состоит из двух частей:

1. Обновление дескриптора процесса с новым сигналом.
2. Если этот процесс будет перенесен, или если он возвращается из прерывания, он сначала проверяет, есть ли сигнал на рассмотрении. Если да, то первым он запускает обработчик сигнала и только после этого продолжает работу с остальной частью программы.

Этот механизм также не подходит из-за ограниченных возможностей, как передачи самого сигнала, так и объема передаваемых данных.

### 2.1.4 Системные вызовы ядра

Системные вызовы используются, когда программа пользовательского пространства хочет использовать какие-то данные или какие-либо сервисы, предоставляемые ядром Linux. В версии ядра Linux 2.6.23 существовало 324 системных вызова. Все системные вызовы служат функциями, которые полезны для многих прикладных программ (например, операции с файлами, сетевые операции или операции, связанные с процессом). Нет никакого смысла в добавлении очень специфических системных вызовов, которые могут быть использованы только конкретной программы.

Обычно системный вызов вызывается через функцию-обертку, предоставляемую Glibc (например, open(), socket(), getpid()). Внутренне каждый системный вызов идентифицируется уникальным номером.

Когда процесс пользовательского пространства вызывает системный вызов, процессор переключается в режим ядра и выполняет соответствующие функции ядра.

Для того, чтобы действительно перейти от режима пользователя в режим ядра есть некоторые ассемблерные инструкции. Для x86 архитектуры есть две возможности: "Int $ 0x80", или более новый "sysenter". Обе эти возможности являются причиной:

* переключения процессора в режим ядра,
* необходимости сохранения регистров
* некоторых проверок корректности
* запуска системного вызова, соответствующего числу предоставляемого процессом пространства пользователя.

Если выполнение системного вызова завершено, функция system\_call проверяет, существует ли другие задачи, например, реструктуризация или обработка сигнала и, наконец, восстанавливает контекст пользовательского процесса.

Добавление собственного системного вызова является сложным и некорректным вариантом для передачи данных между демоном и модулем, поэтому этот вариант тоже не подходит.

### 2.1.5 Файловая система

Файловые системы procfs, sysfs, configfs и debugfs не являются обязательными для ядра Linux, и могут быть не включены в ядро.

Для обмена данными между пространством пользователя и пространством ядра Linux обеспечивает несколько RAM файловых систем. Эти интерфейсы сами разработаны на основе файлов. Обычно файл представляет одно значение, но оно может также представлять множество значений. Пространство пользователя может получить доступ к этим значениям с помощью стандартных функций read и write. Для большинства файловых систем функции read и write передают результаты в функцию обратного вызова в ядро Linux, которое имеет доступ к соответствующим значениям.

Несмотря на аналогичную функциональность, различные RAM файловые системы предназначены для различных целей. Однако также легко использовать эти файловые системы и для других целей. Такие вопросы как "Какую файловую систему следует использовать?" или "Почему существует необходимость различных файловых систем?" часто возникают в списке рассылок ядра Linux. Аргументы спорные и каждый разработчик, кажется, имеет уникальный мнение.

Преимущество использования функций read и write по сравнению, например, с socket основано на том, что пользователь имеет много инструментов, доступных для передачи данных в пространстве ядра (например, cat, echo) . Эти программы хорошо известны пользователям, и они могут быть использованы в скриптах.

**Procfs**

Procfs – файловая система, расположенная в /proc, является самым известным интерфейсом этого класса. Первоначально она была предназначена для экспорта всех видов обработки информации, таких как текущее состояние процесса или все дескрипторы открытых файлов для пользовательского пространства. Несмотря на свои начальные цели, procfs была использована для множества других целей:

* Предоставлять информацию о работающей системе, такой как информация о процессоре, информация о прерываниях, о доступной памяти или версии ядра.
* Информация об устройствах IDE, SCSI-устройстве и терминалах.
* Информация о сети, такая как ARP таблица, сетевая статистики или список использованных сокетов.

Существует специальный подкаталог /proc/sys. Он позволяет настроить множество параметров в работающей системе. Обычно каждый файл состоит из одного значения. Это значение может соответствовать:

* пределу (например, максимальный размер буфера)
* включению и выключению данной функциональности (например, маршрутизации)
* или представляет некоторые другие переменные ядра

Все каталоги и файлы ниже /proc/sys/ не реализованы интерфейсом procfs. Вместо этого они используют механизм, называемый sysctl.

Следует обратить внимание, что, несмотря на широкое использование procfs, она устарела и должна использоваться только для экспорта информации, связанной с самим процессом.

**Sysfs**

Sysfs была предназначена для представления всей модели устройств, как это представлено в ядре Linux. Она содержит информацию об устройствах, драйверах, шинах и их взаимосвязи. Файловая система Sysfs сильно структурирована и содержит много связей между отдельными каталогами. Что же касается ядра 2.6.23, то оно содержит следующие 9 каталогов верхнего уровня:

* sys/block/ содержит все известные блочные устройства, такие как hda/ ram/ sda/
* sys/bus/ содержит все зарегистрированные шины. Каждый каталог ниже bus/ имеет по умолчанию два подкаталога:
  + device/ для всех устройств, подключенных к этой шине
  + driver/ для всех драйверов, назначенных на эту шину.
* sys/class/ для каждого типа устройств существует подкаталог, например /printer или /sound
* sys/device/ содержит все устройства, известные в ядре, организованные шиной, к которой они подключены
* sys/firmware/ файлы в этом каталоге обрабатывают встроенное программное обеспечение некоторых устройств
* sys/fs файлы для управления файловой системой, в настоящее время используется FUSE - реализацей файловой системы пространства пользователя
* sys/kernel/ содержит каталоги (точки монтирования) для других файловых систем, таких как debugfs, securityfs.
* sys/module/ каждый загруженный модуль ядра представлен в этом ​​каталоге.
* sys/power/ файлы для обработки состояния питания некоторых аппаратных средств

**Configfs**

Configfs есть некоторый аналог Sysfs. Ее можно рассматривать как файловую систему, основанную менеджером объектов ядра. Важное различие между configfs и sysfs том, что в configfs все объекты создаются из пользовательского пространства с помощью вызова mkdir. Ядро отвечает за создание атрибутов (файлов), а затем они могут быть прочитаны и записаны самим пользователем. Если пользователь больше не нуждается в файле, он вызывает команду rmdir, и все будет удалено. Поэтому жизненный цикл объекта configfs полностью контролируется пространством пользователя.

Каждый раз, когда mkdir вызывается, новый "config\_item" создается реализацией ядра. Этот config\_item представляет файлы (атрибуты), отображение и функции обратного вызова хранилища, а также связанное с ним значение. Поэтому каждый mkdir создает новый каталог, а также новые файлы, которые представляют собой новые значения.

Configfs имеет те же ограничения, что и Sysfs: ​​каждый файл должен представлять только одно значение, и оно должно быть меньше, чем PAGE\_SIZE байт.

**Debugfs**

Debugfs является простой в использовании файловой системой, основанной в оперативной памяти, специально предназначенной для отладки. Разработчикам предлагается использовать debugfs вместо procfs для того, чтобы получить некоторую отладочную информацию из их кода ядра. Debugfs довольно гибкая: она обеспечивает возможность установки или получения одного значения с помощью всего одной строки кода, но разработчики также разрешили написать свои функции read/write, и он может использовать seq\_file интерфейс, описанный в procfs разделе.

В итоге получаем: procfs является уже устаревшей файловой системой, если использовать sysfs, то скорость передачи может быть ограничена скоростью записи/чтения жесткого диска, а для использования debugfs в памяти требуется дополнительный управляющий канал. По этому, файловую систему использовать для реализации взаимодействия между ядром и пользователем в PAN профайле нерационально.

### 2.1.6 Управление вводом-выводом

В дополнении к функциям чтения и записи, описанным в разделе Procfs, Sysfs, и подобным интерфейсам ядра, все файловые механизмы дают возможность дополнительных команд управления, которые поддерживают ioctl (Input/Output Control) метод. Механизм ioctl реализован как один вызов в системе, который составляет различные команды в соответствующую функцию пространства ядра. Вызов ioctl имеет три аргумента: файловый (или сокетный) дескриптор, номер, который идентифицирует команду, и данные. Мультиплексирование осуществляется на основе а) дескриптор файла и б) числа команд. Концептуально было бы возможно использовать любое число для новой ioctl команды, но это не рекомендуется, а предлагается использовать общесистемные уникальные номера вместо этого. Это гарантирует, что невозможно выполнить ioctl на неправильном устройстве, приводящем к неожиданному поведению. Точный механизм получения уникального числа команд описан в документации ядра Linux.

Существуют различные типы аргументов для ioctl.

* Команда не требует каких-либо данных.
* Команда записывает некоторые данные в ядро.
* Команда читает некоторые данные из ядра.
* Модуль ядра считывает данные и обменивает их на новые данные.

Тип параметра может быть определен, когда число команды сгенерировано, как описано в документации ядра Linux.

Использование ioctl в качестве передачи данных является хорошим способом, однако накладывает некоторые ограничения. Если использовать ioctl, то необходимо создавать символьное устройство. Таким образом, у модуля будет два видимых интерфейса: сетевое и символьное устройство. Это будет не правильное архитектурное решение.

### 2.1.7 Сокеты

Сокеты позволяют ядру Linux отправлять уведомления приложению пространства пользователя. В отличие от механизмов файловой системы, где ядро может изменить файл, а пользовательская программа не узнает о любом изменении, пока она получает доступ к этому файлу. Сокет базируется на механизме, который позволяет приложениям слушать сокет, и ядро может отправлять им сообщения в любое время. Это приводит к коммуникационному механизму, в котором пространство пользователя и пространство ядра - равноправные партнеры.

Есть разные семейства сокетов, которые могут быть использованы для достижения этой цели:

* AF\_INET: предназначено для работы в сети, но и UDP сокеты могут также использоваться для связи между модулем ядра и пользовательским пространством. Использование сокетов UDP для узла местной связи включает в себя много накладных расходов.
* AF\_PACKET: позволяет пользователю определить все заголовки пакетов.
* AF\_NETLINK (Netlink сокеты): Они специально предназначены для связи между пространством ядра и пользовательского пространства. Существуют различные типы сокетов Netlink в настоящее время реализованных в ядре, все из которых касаются определенного подмножества сетей части ядра Linux.

**UDP сокеты**

UDP сокеты аналогичны UDP сокетам, работающим в пространстве пользователя. С сокетами UDP возможно иметь передачу между модулем ядра на одной системе и приложением пространства пользователя на другой машине.

**Netlink сокеты**

Netlink - специальный IPC, используемый для того, чтобы передать информацию между ядром и процессами пространства пользователя, который обеспечивает полнодуплексный коммуникационный канал между ядром Linux и пространством пользователя. Он использует стандартный API сокета для процессов пользовательского пространства и специальный API ядра для модулей ядра. Сокеты Netlink используют семейство адресов AF\_NETLINK, по сравнению с AF\_INET, используемым сокетами TCP/IP. Сокетный интерфейс для netlink появился в Linux 2.2.

У сокетов Netlink есть следующие преимущества перед другими коммуникационными механизмами:

1. Просто взаимодействовать со стандартным ядром Linux, поскольку только константа (параметр подключения) должна быть добавлена к исходному коду ядра Linux. Нет никакого риска загрязнить ядро или диск в случае его нестабильности, так как сокет может использоваться сразу.
2. Сокеты Netlink являются асинхронными, поскольку они предоставляют очереди, означая, что они не нарушают планирование ядра. Это в отличие от системных вызовов, которые должны быть сразу выполнены.
3. Сокеты Netlink обеспечивают возможность многоадресной передачи.
4. Сокеты Netlink обеспечивают действительно двунаправленный канал передачи: передача сообщений может инициироваться или ядром или приложением пространства пользователя.
5. У них меньше требований к ресурсам (заголовок и обработка) по сравнению со стандартными UDP-сокетами.

Помимо этих преимуществ netlink сокеты имеют два недостатка:

1. Каждый объект, используя netlink сокеты, должен определить свой собственный тип протокола (семейство) в заголовочном файле ядра include/linux/netlink.h, т.е. необходима перекомпиляция ядра прежде, чем это сможет использоваться.
2. Максимальное количество netlink семейств фиксировано к 32. Если все зарегистрируют его собственный протокол, то это число будет исчерпано.

Чтобы устранить эти недостатки, было реализовано «Универсальное семейство Netlink» (Generic Netlink Famaly). Это действует как мультиплексор Netlink в том смысле, что различные приложения могут использовать generic netlink семейство адресов.

Generic Netlink связь - это по существу серия различных каналов передачи, которые мультиплексированы на единственном семействе Netlink. Каналы передачи однозначно определены номерами каналов, которые динамически выделены Generic Netlink контроллером. Ядро или пользователи, которые предоставляют услуги, устанавливают новые каналы передачи, регистрируя их службы в Generic Netlink контроллере. Пользователи этой службы запрашивают контроллер, чтобы увидеть, существует ли служба и определить корректный номер канала.

Каждое Generic Netlink семейство может обеспечить различные "атрибуты" и "команды". Каждая команда имеет свою собственную функцию обратного вызова в модуле ядра и может получить сообщения с различными атрибутами. И команды и атрибуты, "адресуются" идентификатором.

# 3. Особенности реализации и архитектура PAN профайла

## 3.1 Использование сокетов Netlink

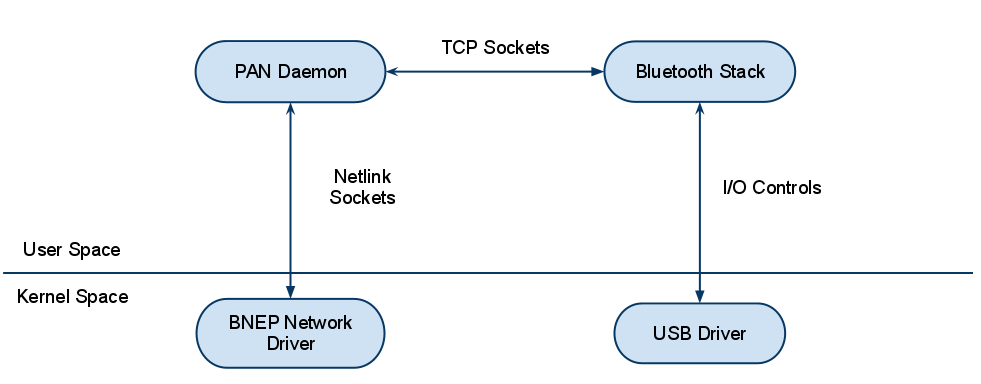


Рис. Архитектура PAN профайла для Bluetooth Linux

Семейство адресов Netlink использует стандартный API сокета BSD в качестве своего интерфейса между программами пространства пользователя и различными компонентами ядра. Создание сокета netlink формируется тем же самым способом как любой другой сетевой сокет через вызов библиотеки сокета:

socket\_fd = socket(AF NETLINK, SOCKET\_RAW, protocol);

Семейство адресов всегда задается как AF\_NETLINK, а его тип всегда задается как SOCKET\_RAW. Единственное изменение в создании сокета netlink - обеспечиваемый протокол. Список доступных протоколов будет продолжать изменяться, как аспекты конфигурации сетевого стека Linux, который добавляет в их конфигурируемость. Ниже представлен список доступных протоколов:

* NETLINK\_ARPD
* NETLINK\_FIREWALL
* NETLINK\_IP6\_FW
* NETLINK\_NFLOG
* NETLINK\_ROUTE
* NETLINK\_ROUTE6
* NETLINK\_TAPBASE
* NETLINK\_TCPDIAG
* NETLINK\_XFRM

Каждый протокол индивидуально описывает свою собственную секцию.

### 3.1.1 Прием и передача дейтаграмм

Сокеты Netlink работают без установления соединения, почти таким же способом, как сокеты UDP. Сообщения могут быть отправлены получателям через открытый сокет netlink с помощью функций sendto или sendmsg. Сообщения могут быть получены функциями recvfrom и recvmsg. Следует отметить, что сообщениями не обмениваются с помощью функций send и recv. Это является причиной того, что сокеты netlink работают без установления соединения. Также как и сокеты UDP, netlink сообщения передаются в дейтаграммах. Нет как таковой никакой гарантии поставки между конечными точками сокета, хотя есть механизмы в заголовке сообщения netlink, которые разработаны, чтобы помочь программисту добавить уровень надежности к протоколу для тех приложений, которые требуют этого.

### 3.1.2 Структура адреса сокета Netlink

Структуру адреса сокета netlink, названную struct sockaddr\_nl, передают ко всем вызовам, которые отправляют или получают сокеты netlink. Эта структура сообщает сетевому стеку ядра места назначения дейтаграмм, и сообщает программе пространства пользователя источника полученных кадров. Структура определена следующим образом:

struct sockaddr\_nl

{

sa\_family\_t nl\_family;

unsigned short nl\_pad;

\_\_u32 nl\_pid;

\_\_u32 nl\_groups;

}

* nl\_family - это поле, определяющее семейство адресов сообщения, отправляемого по сокету. Оно должно всегда устанавливаться в AF\_NETLINK.
* nl\_pad - это поле не используется и всегда должно обнуляться.
* nl\_pid - это поле содержит PID процесса, который должен принять отправляемый кадр или процесса, который передавал получаемый кадр. В это поле можно установить значение PID процесса, который должен получить фрейм, или обнулить для многоадресной адресации или для передачи пакета ядру.
* nl\_groups - это поле используется для того, чтобы отправить многоадресные сообщения по netlink. У каждого netlink семейства протокола есть 32 группы многоадресной передачи, на которые процессы можно передавать и получать сообщения. Чтобы подписаться на определенную группу, необходимо вызвать функцию bind и установить в поле nl\_groups соответствующую битовую маску. Отправка многоадресных пакетов работает таким же образом, устанавливая в поле nl\_groups соответствующий набор значений при вызове sendto или sendmsg. Каждый протокол использует группы многоадресной передачи по-другому, если вообще использует, и их использование определено условно.

Структура sockaddr\_nl приводится к структуре sockaddr и передается соответствующим параметром в функции recv и send.

### 3.1.3 Формат сообщения Netlink

Так же, как у каждого сообщения IP есть стандартный заголовок IP, netlink обменивается сообщениями, имеющими идентичный заголовок. Однако, в отличие от других протоколов, программист обязан создавать этот заголовок для каждого фрейма. Этот заголовок используется, чтобы сохранить метаданные о каждом сообщении netlink и формирует основную инфраструктуру каждого netlink протокола.

### 3.1.4 Netlink заголовок

Netlink заголовок определен следующим образом:

struct nlmsghdr {

\_\_u32 nlmsg len;

\_\_u16 nlmsg type;

\_\_u16 nlmsg ﬂags;

\_\_u32 nlmsg seq;

\_\_u32 nlmsg pid;

}

* nlmsg\_len - каждый заголовок сообщения netlink сопровождается нулем или большим количеством байтов вспомогательных данных. Это 4ехбайтовое поле записывает общую длину данных в сообщении, непосредственно включая заголовок.
* nlmsg\_type - это 2 байтовое поле определяет формат данных, которые следуют за заголовком.
* nlmsg\_ﬂags - это 2 байтовое поле с помощью побитового ИЛИ определяет различные управляющие флаги, которые определяют, как каждое сообщение обрабатывается и интерпретируется.
* nlmsg\_seq - Это 4 байтовое поле является произвольным числом, и используются процессами, которые создают netlink сообщения, чтобы коррелировать запросы с их ответами.
* nlmsg\_pid - Это 4 байтовое поле используется также как nlmsg\_seq. Оно может использоваться для связи сообщений запроса к ответным сообщениям и используется, прежде всего, в сокетах netlink, которые используют функцию многоадресной передачи (допускается, что может быть несколько отправителей и получателей данных). Любое сообщение, которое порождено пользовательским процессом, должно установить этот флаг в значение, возвращенное вызовом функцией getpid. Также надо отметить, что любая программа, получающая netlink сообщения сокета от ядра, обязательно должна проверить, что это поле обнулено.

### 3.1.4 Макросы Netlink

Описывает различные макросы для доступа к сетевым датаграммам и их создание. Буфер, работающий в сетевых сокетах, должен использоваться только с помощью этих макросов.

* NLMSG\_ALIGN - Выравнивает длину сетевого сообщения до большей корректной длины.
* NLMSG\_LENGTH - Рассматривает длину "полезной" части как аргумент и возвращает выравненную длину полям **nlmsg\_len** или *nlmsghdr*.
* NLMSG\_SPACE - Возвращает количество байтов сетевого сообщения.
* NLMSG\_DATA - Возвращает указатель на "полезную" часть сообщения, ассоциированную с *nlmsghdr*.
* NLMSG\_NEXT - Получает следующий заголовок *nlmsghdr* в сообщении, состоящем из нескольких частей. При вызове необходимо убедиться, что в текущем nlmsghdr не установлен флаг NLMSG\_DONE - эта функция не возвращает NULL в конце. Параметр length - это величина, указывающая на оставшуюся длину буфера. Этот макрос уменьшает ее на размер заголовка сообщения.
* NLMSG\_OK - Возвращает значение "истина", если сетевое сообщение не разорвано и готово к обработке.
* NLMSG\_PAYLOAD - Возвращает длину "полезной" части, ассоциированной с *nlmsghdr*.

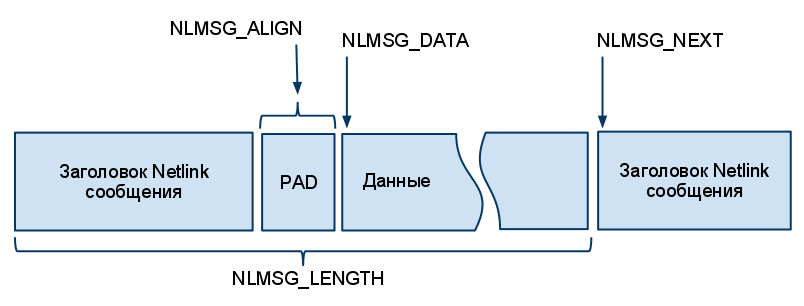


Рис. Структура сообщения Netlink

Вдобавок отмечу, что Netlink Sockets много проще реализуемы, т.к. для приема пакетов для их API необходимо указать callback-функцию, вместо создания дополнительного потока, в случае UDP сокетов. По этому, было решено воспользоваться Netlink сокетами.

## 3.2 Реализация сетевого драйвера

Сетевое устройство регистрирует себя в специальных структурах ядра, для обмена пакетами с внешним миром.

Имеются несколько важных отличий между монтированием дисков и интерфейсом доставки пакетов. Диск представлен как специальный файл в каталоге /dev, а сетевой интерфейс не имеет подобного представления. Стандартные файловые операции (чтение, запись и т.п.) не производятся над сетевым интерфейсом, также не применим принцип Unix «все является файлом». Таким образом, сетевой интерфейс существует в своем пространстве, и экспортирует собственный набор операций.

Сетевое устройство само запрашивает ядро для передачи полученного пакета.

Помимо этого сетевые драйверы должны решать административные задачи: установка адреса, изменение параметров передачи, учет трафика и ведение статистики ошибок. API сетевого драйвера отражает эти требования.

Сетевая подсистема ядра Linux, спроектирована так, чтобы быть независимой от протокола. Это применимо к сетевым протоколам (Internet protocol [IP], IPX или другие) и аппаратному протоколу (Ethernet, token ring и т.д.). При взаимодействии между сетевым драйвером и ядром одновременно может передаваться только 1 пакет. Это позволяет скрывать от драйвера протокол, и от протокола – физическую среду.

Работа сетевого драйвера состоит из нескольких этапов.

### 3.2.1 Регистрация устройства

Сетевой драйвер создает новую структуру данных для каждого нового обнаруженного интерфейса в глобальном списке сетевых устройств.

Каждый интерфейс описывается структурой net\_device, которая определена в файле <linux/netdevice.h>.

Структура net\_device, подобно многим другим структурам ядра, содержит kobject, и, следовательно, счетчик ссылок, и экспортируется через sysfs. Как и другие такие структуры должна создаваться динамически. Для этого ядро предоставляет функцию:

struct net\_device \*alloc\_netdev(int sizeof\_priv, const char \*name, void (\*setup)(struct net\_device \*));

Здесь sizeof\_priv – размер пространства «внутренних данных» драйвера. Эта область размещается вместе со структурой net\_device. Фактически вместо 2-х выделений памяти производится 1 большое, но драйвер делает вид, что не знает об этом. Name – имя интерфейса, каким оно будет в пространстве пользователя. Это имя может иметь printf-стиль – содержать «%d». Ядро заменяет «%d» доступным номером интерфейса. setup – указатель на функцию инициализации, которая используется для настройки структуры net\_device.

После инициализации структуры net\_device, необходимо передать ее в функцию register\_netdev:

int register\_netdev (struct net\_device \*dev);

Так как BNEP протокол и Ethernet поддерживают одни и те же протоколы, то драйвер имеет те же настройки, что и стандартный ethernet-драйвер. По этому, можно воспользоваться функцией ether\_setup, которая задает стандартные настройки для Ethernet устройств.

void ether\_setup (struct net\_device \*dev);

Еще одно важно поле структуры net\_device - priv. Ее назначение - хранение внутренних данных драйвера. Под priv выделяется память одновременно с выделением под net\_device. Прямой доступ к полю priv не одобряется, это сделано в целях повышения производительности и гибкости. Для доступа к этому полю драйвер должен использовать функцию netdev\_priv:

void \*netdev\_priv (struct net\_device \*dev);

### 3.2.2 Выгрузка модуля

При выгрузке модуля ничего особенного не происходит. Функция выгрузки отменяет регистрацию интерфейса, выполняет внутренние операции по выгрузке, и освобождает структуру net\_device.

unregister\_netdev (struct net\_device \*dev);

free\_netdev (struct net\_device \*dev);

Вызов unregister\_netdev удаляет интерфейс из системы. free\_netdev освобождает выделенную память. Если ссылки на эту структуру существуют где-нибудь еще, они могут продолжать использоваться, и драйвер не должен об этом беспокоиться. После отмены регистрации, ядро не станет вызывать методы интерфейса.

### 3.2.3 Открытие и закрытие

До того как интерфейс сможет доставлять пакеты, ядро должно произвести открытие драйвера, после чего произойдет назначение адреса. Ядро открывает или закрывает интерфейс в ответ на команду ifconfig.

Когда ifconfig используется для назначения адреса интерфейсу, он выполняет два действия.

Назначает адрес вызовом ioctrl(SIOCSIFADDR) (Socket I/O Control Set Interface Address). Затем устанавливает бит IFF\_UP в dev->flag, через ioctrl(SIOCSIFFLAGS) (Socket I/O Control Set Interface Flags), активизируя интерфейс.

Работа функция ioctrl(SIOCSIFADDR) мало затрагивает драйвер. Драйвер не выполняет никаких действий, задача не зависит от устройства, и выполняется ядром. Следующая команда (ioctrl(SIOCSIFFLAGS)) приводит к вызову метода устройства open.

Аналогично происходит и выключение интерфейса. ifconfig использует ioctl(SIOCSIFFLAGS) для очистки флага IFF\_UP и вызова метода stop.

Оба метода возвращают 0 в случае успешной работы, и отрицательное значение, если возникла ошибка.

Эти управляющие сообщения были использованы PAN демоном для включения и выключения сетевого интерфейса, а также для передачи драйверу MAC-адреса Bluetooth устройства.

В функции оpen, драйвер должен выполнить еще некоторые действия.

Аппаратный (MAC) адрес должен копироваться из устройства в поле dev->dev\_addr, до того как интерфейс свяжется с внешним миром. Метод также стартует очередь передачи, что позволяет посылать пакеты. Для этого ядро предоставляет функцию:

void netif\_start\_queue(struct net\_device \*dev);

Метод stop выполняет действия обратные оpen. Функцию, реализующую stop, часто называют close или release. Функция:

void netif\_stop\_queue(struct net\_device \*dev);

противоположна netif\_start\_queue. Функция вызывается при закрытии интерфейса (в методе stop). Так же может использоваться временная приостановка передачи, как показано в следующем подразделе.

### 3.2.4 Передача пакетов

Передача – действие, направленное на пересылку пакета по сети. Каждый раз, когда ядру необходимо переслать пакет данных, оно вызывает метод драйвера hard\_start\_transmit, что помещает данные в выходную очередь. Каждый пакет, обработанный ядром, помещается в структуру буфера сокета (sk\_buff), которая определена в <linux/skbuff.h>. Название структуры происходит из абстракции сетевых соединений Unix – сокетов. Хотя интерфейс не имеет ничего общего с сокетами, каждый пакет являлся частью сокетов на более высоких уровнях сети, и буфер ввода/вывода является списком структур sk\_buff. Структура sk\_buff используется для хранения сетевых данных во всех сетевых подсистемах Linux, но буфер сокетов находится далеко от интерфейса. Буфер сокета – сложная структура, и ядро предлагает функции для работы с ним.

Буфер сокета переданный в hard\_start\_xmit, содержит физический пакет, для передающей среды, укомплектованный заголовками передающего уровня. Интерфейс не должен изменять данные при передаче. skb->data – указатель на передающийся пакет, skb->len – длина в октетах.

Функция передачи только выполняет некоторые проверки пакета и передает данные, через связанную с аппаратурой функцию. Однако, нужно проявить осторожность, т.к. длинна пакета может оказаться меньше, чем поддерживает аппаратная среда. При этом во многих сетевых драйверах Linux (и в других операционных системах) обнаруживается утечка данных. Чем создавать защиту, лучше скопировать короткие пакеты в отдельный массив, который можно дополнить нулями до требуемого размера.

В случае успешной работы, hard\_start\_xmit должен возвращать 0. Это делает драйвер ответственным за пересылку пакета, необходимо сделайть это наилучшим образом для гарантирования успешной пересылки, в конце можно освободить skb. Возврат не нулевого значения показывает, что пакет не передался, позже ядро повторит попытку. В этом случае драйвер должен остановить очередь, пока проблема не будет устранена.

### 3.2.5 Контроль одновременной передачи

Функция hard\_start\_xmit защищена от одновременных вызовов spinlock-ом (xmit\_lock) в структуре net\_device. Как только функция закончит работу, она может быть вызвана вновь. Функция заканчивает работу, когда даст все указания аппаратуре для передачи пакета, но аппаратная передача, возможно, не будет завершена.

Реальный аппаратный интерфейс работает по-другому. Он передает пакеты асинхронно, и имеет ограниченное количество памяти для хранения отправляемых пакетов. Когда память (которая может быть рассчитана на 1 пакет) – израсходована, драйвер сообщает сетевой системе, что не следует производить других отправок, до тех пор, пока аппаратура не будет готова принять новые данные.

Это можно сообщить вызовом netif\_stop\_queue, функция останавливает очередь. Каждая остановка очереди должна сопровождаться последующим возобновлением передачи. Делается это с помощью функции:

void netif\_wake\_queue(struct net\_device \*dev);

Эта функция похожа на netif\_start\_queue, за исключением того, что она снова заставляет сетевую систему начать передачу пакетов.

Современные устройства имеют внутреннюю очередь, рассчитанную на несколько передаваемых пакетов, это улучшает производительность сети. Сетевой драйвер для этих устройств должен поддерживать множественное ожидание передачи в любое время, вне зависимости от того заполнилась ли полностью или нет память устройства. Всякий раз, когда память устройства заполняется, и нет места для новых пакетов, драйвер должен остановить очередь до тех пор, пока место не появиться.

Если необходимо запретить передачу пакетов отовсюду, где используется функция hard\_start\_xmit (возможно, в ответ на запрос реконфигурации), тогда следует использовать функцию:

void netif\_tx\_disable(struct net\_device \*dev);

Эта функция работает аналогично netif\_stop\_queue, но еще гарантирует то, что когда она вернет управление, метод hard\_start\_xmit не будет запущен на другом CPU. Передача возобновляется как обычно, вызовом netif\_wake\_queue.

### 3.2.6 Прием пакетов

Прием данных из сети сложнее, чем их передача, потому что sk\_buff должен создаваться и обрабатываться верхними уровнями атомарно.

В функции приема пакетов на первом шаге выделяется буфер под пакет. Следует обратить внимание, что функции выделения буфера dev\_alloc\_skb, необходимо передать его длину. dev\_alloc\_skb вызывает kmalloc атомарно, таким образом, это может быть безопасно использовано в обработчике прерывания. Ядро предлагает другие интерфейсы для выделения буферов сокета, но они не приведены здесь.

Значение, возвращаемое dev\_alloc\_skb, необходимо проверить.

После получения skb, пакет данных копируется в буфер, вызовом memcpy. Функция skb\_put, обновляет указатель конца данных в буфере, и возвращает указатель на созданную область памяти.

Прежде чем принять пакет, сетевой уровень должен получить дополнительную информацию. Должны быть сформированы поля dev и protocol. Код поддержки Ethernet экспортирует вспомогательную функцию eth\_type\_trans, которая находит подходящее значение для поля protocol. Затем надо указать, как вычисляется контрольная сумма для пакета, и вычисляется ли вообще. Возможные значения:

CHECKSUM\_HW

Устройство вычисляет контрольную сумму аппаратно. Пример аппаратного вычисления контрольной суммы – интерфейс SPARC HME.

CHECKSUM\_NONE

Контрольная сумма не проверяется. Задача должна выполняться системным программным обеспечением. Это значение используется по умолчанию, для вновь выделенного буфера.

CHECKSUM\_UNNECESSARY

Не производится никаких вычислений контрольной суммы.

На последнем шаге приема пакета выполняется функция netif\_rx, которая передает буфер сокета верхним уровням. netif\_rx возвращает целое значение, NET\_RX\_SUCCESS (0) если пакет успешно принят, и другое значение в случае возникновения проблем. Имеются три значения (NET\_RX\_CN\_LOW, NET\_RX\_CN\_MOD и NET\_RX\_CN\_HIGH) которые показывают уровень перегрузки сетевой подсистемы; NET\_RX\_DROP - количество пакетов было минимальным. Драйвер может использовать эти значения для прекращения подачи пакетов в ядро, когда возникнет перегрузка, но на практике многие драйверы игнорируют значение возвращаемое netif\_rx.

## 3.3 Работа демона пользовательского пространства

В работу драйвера входит управление подключением к удаленному устройству, обработка BNEP пакетов, прием и передача пакетов драйверу, а также управление драйвером. Демон запущен всегда, пока запущен стек Bluetooth, независимо о того, есть ли подключения или нет. Однако возникали следующие вопросы:

1. Когда загружать и выгружать драйвер.
2. Когда включать и выключать сетевой интерфейс.

Эти вопросы играют важную роль в настройке сети. В отличие от стека BlueZ, где всю настройку сетевого BNEP интерфейса проводит пользователь и, следовательно, ему приходится придумывать сложные скрипты для настройки сети, в рассматриваемой реализации необходимо было эту часть максимально упростить. Сложность здесь добавляет то, что если пользователь захочет использовать DHCP сервер для автоматической выдачи IP адресов удаленным Bluetooth устройствам, сетевой интерфейс должен быть включен до запуска DHCP сервера. Это объясняется тем, что если включить сетевой интерфейс после запуска DHCP сервера, то сервер попросту не станет выдавать IP адреса удаленным устройствам, так как при его запуске, сетевого интерфейса просто не существовало. Перезапуск DHCP сервера может привести к повторной выдаче IP адресов другим участникам сети, если сервер используется и для других сетевых интерфейсов. В итоге, было решено, что сетевой интерфейс должен быть включен всегда, пока включен стек Bluetooth.

Однако это добавляет еще одну проблему. Как только сетевой интерфейс включается, операционная система автоматически начинает передавать пакеты по нему. Это делается, например, для того, чтобы узнать, кто находится на другой стороне. Операционная система попросту посылается ARP запросы сетевому интерфейсу и ждет ответа. Если ответа не приходит, то ничего не происходит, но этот интерфейс больше не используется для передачи трафика. Однако, больше система таких запросов не посылает (конечно, если не перезапустить интерфейс, что, как уже было выяснено ранее, невозможно). Запретить системе передачу пакетов можно с помощью функций:

void netif\_carrier\_off(struct net\_device \*dev);

void netif\_carrier\_on(struct net\_device \*dev);

в сетевом драйвере. Это равносильно тому, если просто отсоединить или подсоединить кабель к сетевой карте. Так как модуль не может знать о подключении к удаленному Bluetooth устройству, эти функции вызываются на определенный тип сообщений, передаваемых по Netlink сокетам.

Таким образом, получается, что модуль загружен всегда, и сетевой интерфейс включен также всегда. А передача пакетов включается только после соединения с удаленным устройством.

Загрузка модуля выполняется с помощью вызова программы modprobe через системный вызов system(). Выгрузка выполняется аналогично, только с помощью программы rmmod.

После того, как модуль загружен, происходит подключение к нему по сокету.

Для передачи MAC-адреса и включения и выключения сетевого интерфейса используются системные Input/Output Controls:

* сообщение SIOCSIFHWADDR устанавливает MAC-адрес сетевого интерфейса. Вместе с сообщением передается сам адрес. При этом сетевой интерфейс должен быть выключен, то есть не содержать флаг IFF\_UP.
* сообщение SIOCSIFFLAGS устанавливает определенные флаги сетевого интерфейса. Для включения интерфейса необходимо установить флаг IFF\_UP, для отключения этот флаг необходимо снять.
* сообщение SIOCGIFFLAGS получает набор флагов, установленных в сетевом интерфейсе.

Для передачи пакетов по Netlink сокетам в драйвере был зарегистрирован собственный id протокола.

### 3.3.1 Конечный автомат

Для управления подключением в демоне используется конечный автомат, состоящих из 5 состояний:

1. Регистрация профайла в стеке Bluetooth
2. Ожидание
3. Подключение к удаленному устройству
4. Подключен
5. Отключение

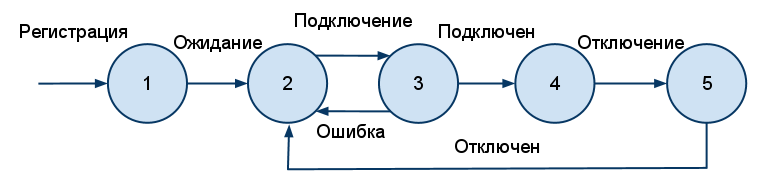


Рис. Конечный автомат

При запуске демона начинается регистрация профайла в стеке Bluetooth. После чего демон переходит в режим ожидания.

В режиме ожидания демон может принимать входящие подключения и выполнять исходящие подключения. Так же в этом режиме запрашивается MAC-адрес Bluetooth устройства и передается в сетевой драйвер.

При исходящем или входящем подключении, демон переходит в третье состояние, в котором осуществляется конфигурация по протоколу L2CAP и BNEP. В случае не успешной конфигурации, демон переходит в режим ожидания. В случае успешной конфигурации - в состояние «подключен».

В этом состоянии демон включает прием пакетов у драйвера, принимает пакеты от драйвера и от стека Bluetooth, обрабатывает их в соответствии со спецификацией протокола BNEP и передает получателю.

Когда удаленное устройство захочет отключиться, в демон придет соответствующее сообщение, которое переведет его в режим отключения. При этом будет произведено отключение передачи пакетов у сетевого драйвера. То же произойдет, если пользователь захочет отключиться от удаленного устройства. После успешного или не успешного отключения, демон перейдет в режим ожидания.

В случае получения сигнала SIG\_TERM от операционной системы или получения соответствующего уведомления от стека об отключении профайла демон произведет удаление регистрации профайла и завершит операции с драйвером.

### 3.3.2 Разбор пакетов

Когда приходят данные от стека, они содержат BNEP заголовок и данные, которые необходимо передать в сетевой драйвер. BNEP заголовок обрабатывается. Из него достается MAC адрес получателя и тип Ethernet пакета. После чего создается Ethernet заголовок, в который записываются MAC адреса получателя и отправителя (Bluetooth устройства) и тип Ethernet пакета. Этот заголовок дополняется данными, полученными вместе с BNEP пакетом, и отправляется в драйвер. Драйвер получает пакет с данными, создает буфер сокета, кладет полученный пакет в созданный буфер и отправляет в сетевой стек ядра.

Когда приходят данные в драйвер, он копирует их в специальную структуру struct iovec и отправляет по Netlink сокетам в демон. Демон получает Ethernet пакет, вытаскивает из Ethernet заголовка MAC адреса отправителя и получателя, формирует BNEP пакет, укладывая в него полученные MAC адреса, тип и дополняет данными, полученными в пакете. После чего отправляет пакет в стек Bluetooth.

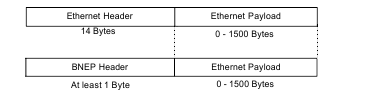


Рис. Структуры пакетов Ethernet и BNEP

### 3.3.3 Буфер пакетов в драйвере

Для приема данных по Netlink сокетам не нужно реализовывать буфер - он уже реализован в ядре и используется автоматически. По этому, беспокоится о потере принятых данных и о задержке при их обработки нет необходимости.

Однако нет гарантии, что при передаче по netlink сокетам, данные не будут потеряны или возникнет задержка при их обработке, что приведет к снижению скорости передачи данных. По этому, пакеты, полученные драйвером от сетевого стека Linux, должны храниться в буфере. Для этого у Netlink сокетов уже имеется специальная очередь для передачи, куда кладутся пакеты. Делается это сразу в функции приема пакетов от сетевого стека Linux. Для передачи пакетов в демон запущен поток, который проверяет очередь на наличие данных, и если таковые имеются, передает их демону.

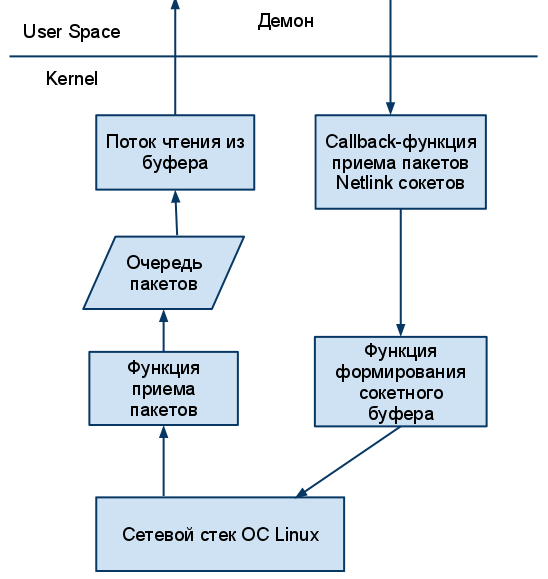


Рис. Передача пакетов в драйвере

## 3.4 Разработка тестового набора модульного тестирования

Модульные тесты позволяют проверить отдельные участки программы, давая возможность проводить рефакторинг кода и проверять, не привело ли изменение кода к его нестабильности. В данном случае, модульные тесты помогают проверять стабильность работы отдельных компонент.

Были разработаны тестовые наборы, позволяющие проверять работу следующих компонент:

1. Разбор сетевых и BNEP пакетов
2. Работу конечного автомата
3. Буферы пакетов в драйвере

Для проверки разбора BNEP пакетов был разработан модуль, генерирующий различные варианты BNEP пакетов. Сгенерированные пакеты передаются в функцию разбора пакета, после чего разобранный пакет направлялся в модуль для последующей проверки.

Также модуль может генерировать пакеты неправильной длины для проверки функции разбора пакетов на ошибки сегментирования и переполнения.

Проверка работы конечного автомата заключалась в проверке правильности перехода его в соответствующее состояние.

Чтобы проверить буфер пакетов драйвера, была разработана функция, генерирующая большое количество пакетов разных размеров и передающая их в буфер с различным временным интервалом.

### 3.4.1 Разбор полученных результатов тестирования

В ходе проведенного тестирования была найдена ошибка, которая проявлялась при передаче по сети пакетов длинной данных без заголовка в 1500 байт. При получении такого пакета от драйвера демон вызывал ошибку сегментации. Проблема оказалась в неправильном использовании макросов Netlink сокетов, которые применяются для определения длины пакетов, и массиве, в который копировались полученные данные. Длина массива, который формировался на стеке, была фиксированной, а макрос определял длину данных больше, чем выделено в массиве. В итоге копирование в такой массив приводило к ошибке сегментирования.

# Результаты

В ходе работы был проведен анализ существующих реализаций PAN профайла в операционной системе Linux. Из 3ех разрабатываемых стеков Bluetooth, архитектуру только одного удалось рассмотреть - BlueZ. При этом были выявлены существенные недостатки его реализации. Они заключались как в архитектуре самого профайла, так и в его неудобстве использования пользователем.

Были рассмотрены механизмы реализации PAN профайла для Bluetooth Linux. Проведен анализ имеющихся механизмов, выявлены неудобства применения некоторых в реализации профайла. В итоге был выбран механизм, наиболее подходящий для реализации, при использовании которого удалось решить одну из проблем, выявленных в реализации стека BlueZ – сохранение стабильности и безопасности ядра Linux.

В ходе разработки демона пользовательского пространства удалось решить еще одну проблему, выявленную у BlueZ. Проблема была связанна с логикой работы сетевого интерфейса в операционной системе. Теперь нет необходимости придумывать сложные скрипты для подключения BNEP драйвера к сети и выдачи IP адресов удаленным устройствам.

В реализации также использовался механизм очереди пакетов, разработанный в ядре, для уменьшения задержек при передаче пакетов от системы в демон.

Разработанный конечный автомат точно определяет текущее состояние профайла, исключая возможные сбои в его работе.

Основной код находится в демоне, что позволяет распространять его под любой лицензией. Функции драйвера минимальны и выполняют лишь транспортировку пакетов между сетевым стеком Linux и демоном. Его код не несет никакой конфиденциальной информации и может свободно распространяться под лицензией GNU GPL.

Было проведено модульное тестирование профайла, в котором проверялись 3 части программы. Была проверена работа конечного автомата, обработчика BNEP пакетов и очереди пакетов. Была выявлена существенная ошибка, при передаче максимального пакета в демон. Эта ошибка была исправлена.

В итоге был получен программный продукт, который был протестирован с помощью модульного тестирования. В котором использовался безопасный механизм межпространственного взаимодействия. И продукт, в котором были решены проблемы, выявленные у другой реализации PAN профайла.