Санкт-Петербургский Государственный Политехнический
Университет
Факультет Технической Кибернетики
Кафедра Информационных и Управляющих Систем

Работа допущена к защите

Зав.кафедрой
_____И.Г.Черноруцкий
“____”_____2011 г.

# ВЫПУСКНАЯ РАБОТА БАКАЛАВРА

Тема: ***Разработка и перенос окружения беспроводного аудио-сервиса***

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

Выполнил студент гр. 4084/2 _____А. А. Петров
Руководитель _____Т. В. Коликова

Санкт-Петербург
2011

# Реферат

С. 57. Рис. 9.

## Разработка и перенос окружения беспроводного аудио-сервиса

Бакалаварская работа посвящана описанию процесса переноса Bluetooth стека на операционную систему семейства GNU/Linux на примере профайла HFP.

Рассмотрены основные сложности связанные с адаптацией аудиосервиса к работе с множеством разрозренных аудиосистем GNU/Linux. Рассматриваются возможные решения задач конфигурации аудиосистем и достижения максимального числа поддерживаемых приложений.

Отдельно рассматривается выбор окружения и вопросы контроля правильности потоков данных.

# Содержание

# Список иллюстраций

## Используемые определения и сокращения

| Сокращение | Определение |
|---|---|
| HFP | Hands free profile |
| RFCOMM | RFCOMM |
| AG | Audio Gateway |
| HF | Hands-free |
| SPP | Serial Port Profile |
| SCO | Synchronous Connection Oriented link |
| A2DP | Advanced Audio Distribution Profile |
| AVRCP | Audio/Video Remote Control Profile |
| GNU | GNU not Unix |
| HSP | Headset profile |
| ALSA | Advanced Linux Sound Architecture |
| IPC | Inter-process communication |
| API | application programming interfaces |
| GUI | Graphical user interface |
| HCI | Host Controller Interface |
| PCM | Pulse Code Modulation |

# Введение

Начиная с 20-х годов XX века технологический процесс невозможно представить без беспроводных технологий. В 2002 явился новой эхой беспроводных технологий - появление первой части стандарта беспроводной технологии Bluetooth и начало победоносного шествования новой технологии по всему миру. Прошло всего лишь несколько лет и на прилавках практически не осталось телефонов, в которых бы не было чипа bluetooth.

Одним из самых популярных bluetooth профайлов является HFP - профайл для работы с беспроводными гарнитурами. Водитель ли вы автомобиля или просто любите высокое удобство - этот профайл был создан для вас.

Существует множество различных реализаций стеков, поддерживающих HFP профайл - они могут быть предназначены для разных операционных систем и разных платформ.

Конфигурация устройств происходит посредством использования профайла SPP - протокола серийного порта ( отдельный профайл бывает представлен в стеках как виртуальный серийный порт, кроме этого однако используется множеством других профайлов) который в свою очередь работает поверх транс портного протокола bluetooth - RFCOMM.

В конфигурации условно принимают участие два устройства - наше и удаленное устройство, как правило hands-free гарнитура или наушники (впрочем это может быть и десктопное устройство). Условно

устройства обозначают AG (audio gateway - сторона bluetooth стека) и HF (Hands-free - сторона устройства ). Так как мы говорим о десктопном приложении, то наше приложение AG.
При запуске стека профайл регистрируется, выполняет инициализацию, при начале соединения устройства устанавливают SPP соединение через который начинают обмен сообщениями, цель которого выявляение поддерживаемых функции, установка громкости, битрейта и других параметров. Если конфигурация проходит успешно, то устройства устанавливают SCO-соединение (далее будем употреблять синхронное соединение), о синхронном соединении речь пойдет далее.

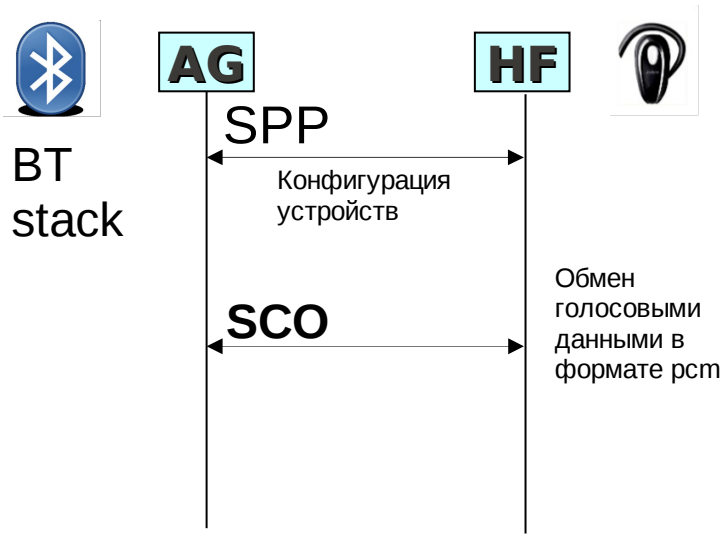


*Рисунок 1. Установка HFP-соединения*

На этом процесс соединения завершается и начинается обмен данными в виде SCO-пакетов, в которых заключены звуковые данные.
В отличии от A2DP в HFP звук передается одноканальный (то есть моно), к нему применяются сложные алгоритмы сжатия, предназначенные специально для передачи голоса.
Все эти замечательные качества HFP делают его одним из самых коммерчески востребованных профайлов для большинства целевых платформ.

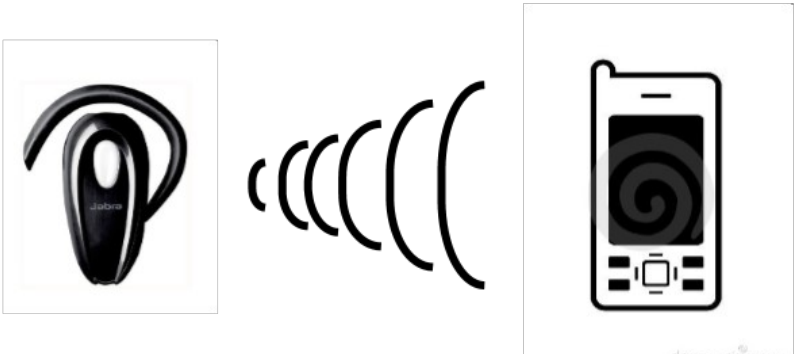

*Рисунок 2. Первоначальная идея работы HFP*

# Глава I. Обзор предметной области

Потребностями в этом профайле обладают как десктопные системы (в основном для возможности использования программ, осуществляющих голосовые звонки), так и встраиваемые системы (рации, другие миниатюрные устройства).

Так как HFP больше всего работает со звуком (присутствует также момент конфигурации устройств, происходящий по протоколу SPP, также возможна параллельная работа с A2DP, использующим асинхронные соединения, AVRCP - занимающимся передачей команд итп) огромную и даже решающую роль при портировании стека играют особенности аудиосистемы целевой платформы.

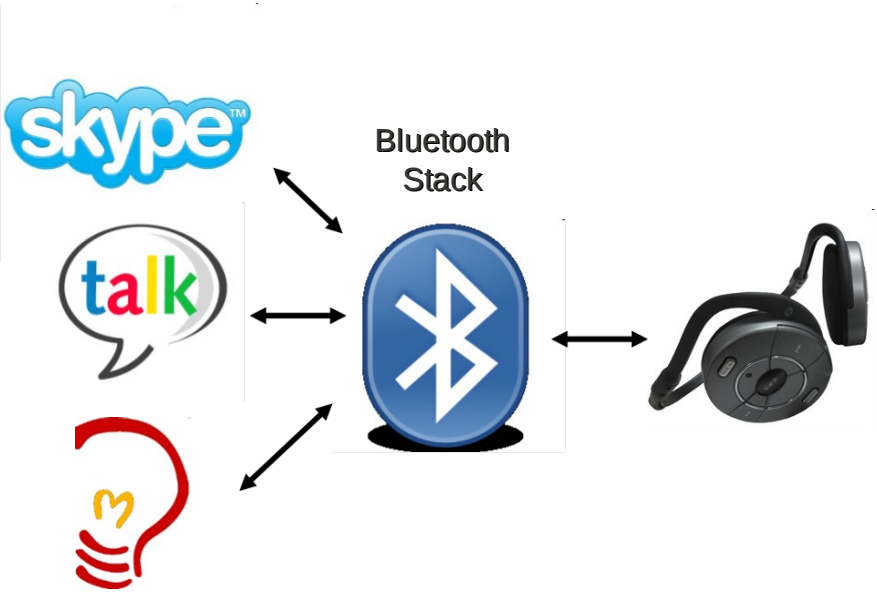


*Рисунок 3. Использование HFP на десктопном PC. Возможность работы с программами, осуществляющими голосовые звонки.*

Целью этой работы является рассмотрение вопроса адаптации стека к работе в операционной системе семейства GNU/Linux - Ubuntu. Адаптация стека в GNU/Linux осложнена множеством факторов. В первую очередь это связано с отсутствием стандартного api и стандартных средств работы со звуком: в GNU/Linux существуют свои аналоги DirectSound, DirectShow, однако они не являются стандартными и могут различаться как в зависимости от дистрибутива, так и от версии к версии, кроме того часто единственный способ для пользователя заставить работать новую аудиосистемы в изменившихся

условиях - вручную править конфигурационные файлы нескольких взаимодействующих с друг другом систем.
Если рассмотреть существующие bluetooth стеки для Linux - единственным действительно сыскавшим популярность стеком является стек BlueZ (Также существует менее популярный проприетарный стек BlueSoliel). Естественно он поддерживает профили HFP/HSP однако его недостатком является ограниченные возможности по конфигурации аудиосистем и отсутствие единого целостного интерфейса, полностью покрывающего потребности пользователя.
Сочетание поддержки широкого спектра аудиосистем и удобного пользовательского интерфейса есть основная цель задача портирования bluetooth стека под операционную систему семейств GNU/Linux.

Для нашего приложения принципиальна наличие возможности получать из аудиосистемы аудиоданные и также отдавать полученные с удаленного устройства данные. Вариантов как это можно сделать огромное множество (значительно больше чем в ОС Microsoft Windows, однако как мы увидем далее, при детальном рассмотрении - это оказывается скорее недостатком, чем достоинством). Рассмотрим их на примере реализации.

1)Phonon — мультимедийный фреймворк, используемый в KDE4, предоставляет удобный API для мультемидийных приложений. Входит в состав Qt, что является огромным плюсом (так как окружение

активно использует Qt), также крайне прост в использовании, так как предоставляет простые и удобные интерфейсы. Например так просто выглядит в Phonon проигрывание аудиофайла:

```
media = new MediaObject(this);
connect(media, SIGNAL(finished()), SLOT(slotFinished()));
media->setCurrentSource("/home/username/music/filename.ogg");
media->play();
```

Недостатками является:
- ограниченная поддержка различными дистрибутивами ( так как Phonon не является самостоятельной аудиосистемой и использует в работе в качестве бэкенда gstreamer или VLC )
- ограниченные возможности использования фильтров
- ограниченные возможности по управлению аудиосистемой на уровне приложения

2) GStreamer - широко используемая аудиосистема, фактически аудио и видео сервер, его рассмотрим далее отдельно.
API gstreamer’a представлено на си, это не является проблемой, так как существует множество bind’ов для других языков - есть адаптированное api и для c++.
Несмотря на большую распространенность чем Phonon так же не является единственно используемой аудиосистемой и относительно сложнее в использовании. Однако обладает намного более широкими возможностями.

3) PulseAudio - ситуация практически аналогична Gstreamer’у

4) ALSA - Advanced Linux System Audio - практически повсеместно присутсвует на десктопах, в последнее время в качестве бэкендов более продвинутых аудиосистем. Является более сложной в использовании и обладает рядом ограничений, которые обходятся путем сложной конфигурации.

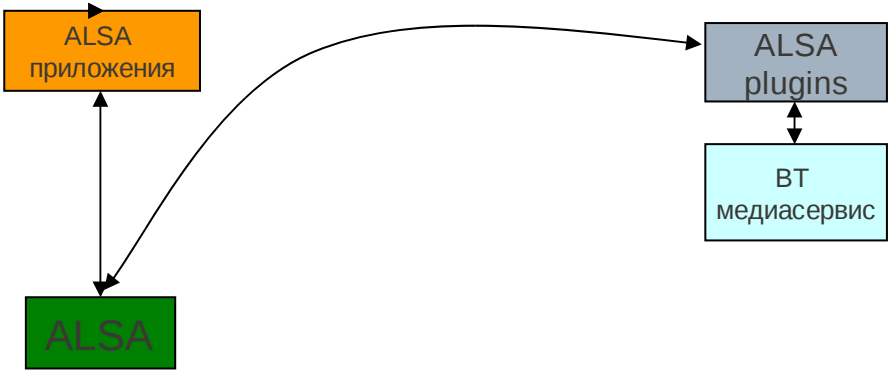


*Рисунок 4. Схема прямого соединия с ALSA*

Видим, что ни один из вариантов не является идеальным, более менее из всего списка выделяется ALSA, с известными ограничениями.

Отдельный вопрос связан с проверкой правильности pcm данных. Задача верификации потока данных сложна, что сложно определить программно правильность прохождения pcm данных на пути от стека до чипсета.

Дело в том что, если проверить правильность потока управления возможно путем написания модульных тестов (unit tests) и проверкой соответствии их реакции на заданный сценарий спецификацией, то с потоком данных сложней.

Попытка верифицировать канал по данным взятых в точках внутри нашей системы проблемно, так как поток управления влияет на поток данных и сигнал с прохождением пути до чипсета может менятся, что будет совершенно справедливо.
Верификация данных между аудиосистемой и удаленным устройством (например посредством использования радиоканальных снифферов специального назначения, либо использованием на стороне удаленного устройства стека, анализирующего входной sco-поток) осложнена использованием шифрования данных в радиоэфире и использования алгоритмов сжатия голоса, которые могут приводить к некоторым искажениям сигнала.

Видно, что автоматизированная верификация довольно сложна, поэтому рассмотрим в дальнейшем методы логирования аудиопотоков и использования его результатов для отладки.

### Межпроцессное взаимодействие

Так как отдельные компоненты системы присутствуют в виде отдельных процессов, то необходимо определить методы их межпроцессного взаимодействия.

**D-bus**

gui и медиасервис существуют в разных процессах, поэтому когда мы говорим об их взаимодействии - мы говорим об Inter-Process Communication - IPC, то есть межпроцессном взаимодействии. Методов межпроцессных взаимодействий огромное множество, начиная с самых простых:

- файлы
- семафоры
- пайпы
- сокеты
- сигналы
- разделяемая память

и заканчивая сложными системами взаимодействия, таких как COM, CORBA и D-Bus.

Что дают более сложные системы ipc? Они дают абстракцию от текущего адресного пространства, может быть даже от копьютера пользователя позволяя например вызывать методы объекта физически находящегося в адресном пространстве другого комьютера в Сети.

# Глава II. Выбор средств реализации и архитектура

В этой работе будет рассмотрено более универсальное решение, основывающееся на взаимодействии с ALSA, используемой в качестве бэкендна звукового сервера (например PulseAudio), который также является фротнэндом для других аудиосистем (GStreamer, ESD).

Такое решение является идеальным в плане охвата приложений, которые сможет использовать пользователь для работы с HFP, но с другой стороны усложняет работу и делает систему менее устойчивой к ошибкам.

Также отдельным вопросом является выбор средств межпроцесного взаимодействия частей нашей системы и методов отладочного контроля ее правильности работы.

Рассмотрим как выглядит bluetooth стек и его окружения, концентрируя внимание на профиле HFP.

Основным звеном работы будет собственно сам bluetooth стек - он ответственен за поиск устройств, операции сопряжения, установления соединения SPP,  конфигурации двух устройств ( HF и AG ), установку SCO-соединения, позже через него будет проходить поток pcm-данных. Однако в нашем случае стек будет черным ящиком, будем считать что нам известны места при-стыковки к нему ( в виде API стека, используемых внешних вызовов, архитектурно-зависимых реализаций и.т.п. ). Следующим элементом являет usb драйвер,

который взаимодействуя с usb ядром Linux будет обмениваться hci командами с донглом bluetooth.

Медиасервис выполняет роль управляющего соединением с удаленным устройством.

Плагины связывают аудиосистему с медиасервисом.

Рассмотрим основные компоненты нашего программного комплекса.

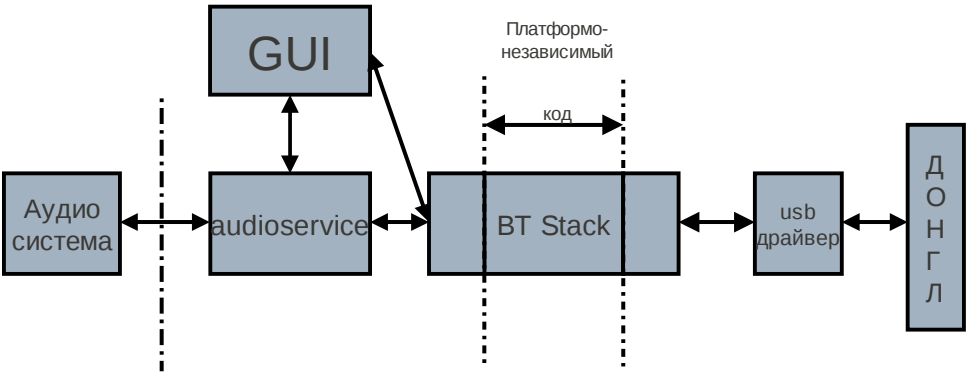


*Рисунок 5. Схема передачи голосовых данных*

Попытаемся представить наш сервис как конечный автомат. Естественно в некотором приближении любая программа является конечным автоматом (пускай количество состояний огромно), мы же будем рассматривать состояния с точки зрения высокоуровневой логики, которая должна определять ключевые моменты работы и состояния программы.

Во-первых определим основных актеров нашей системы

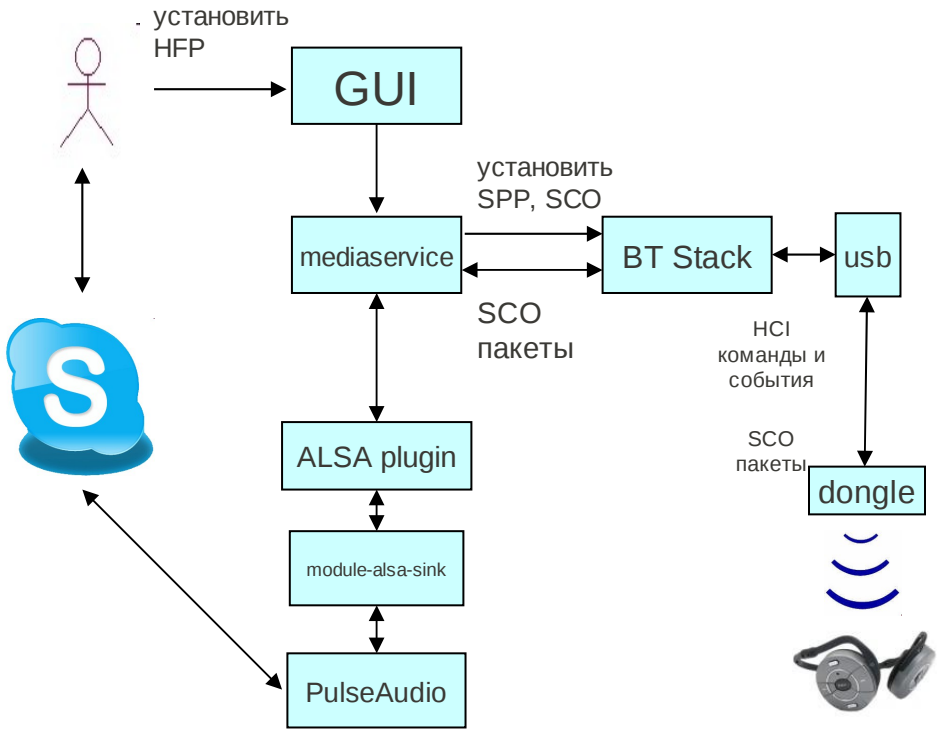


*Рисунок 6 Примерная схема установления HFP соединения*

1) GUI - некоторый комплекс компонент, используемых для взаимодействияс пользователем. Фактически высокоуровневая надстройка, через которую;

- пользователь получает mac-адрес устройства (используемый для установление внутренних соединений)

- пользователь определяет профиль соединения (a2dp, hfp/hsp)

Называем комплексом компонент, так как GUI кроме непосредственной связи с пользователем и графической оболочкой представляемой системой может взаимодействовать со стеком и другими компонентами, обеспечивающих правильную работоспособность системы. Такими компонентами являются например sdp (для определения доступных профайлов), любые иные способы взаимодействия со стеком. Кроме того исторически в gui заложена конфигурация аудиосистемы (а частично на аудиосервис). Таким образом именно GUI возмет часть ответственности за конфигурирование аудио.

Необходимо отметить, что GUI должно знать текущее состояние аудиосервиса, чтобы сигнализировать пользователю о состояние соединения устройства с сервисом (то есть определяет доступные профайлы, после выбора подключения переходит в состояние подключения, в зависимости от удачности подключения переходит в состояние подключено/не подключено, использует дополнительные средства визуализации).

С обратной стороны аудиосервис получает сообщение от gui в котором есть адрес устройства и необходимый профиль и далее сигнализирует сервис об изменении своего состояния, также может получить команду от gui на разрыв соединения Отметим, что если компоненты являются различными процессами, то встает вопрос реализации межпроцессного взаимодействия.

2) аудиосервис - проводит непосредственную конфигурацию устройства, установку соединения посредством api стека, конфигурацию аудиосистемы, передачу аудиопотоков в стек, прием сообщений из стека и изменение своего состояния в соответствии с ними. Также вопрос межпроцессного взаимодействия и способа, как аудиосервис взаимодействует с аудиосистемой.

Так как GUI и аудиосервис обмениваются только управляющими данными, то удобно использовать нечто высокоуровневое для их взаимодействия, например dbus

3) плагины аудиосистемы - по причинам рассмотренным в предыдущей главе были написаны плагины ALSA аудиосистемы, фактически это  shared library, экспортирующая набор функций, через которые аудиосервис будет получать данные от аудиосистемы, и наоборот.

Так как плагин будучи загруженным аудиосистемой будет передавать большое количество данных то нам потребуется нечто, обладающее большой пропускной способностью. Фактически единственно удачное решение - использование сокетов.
Так как сокеты нам нужны в пределах работы ПК удобнее использовать link сокеты.
Так как hfp работает с двумя аудиопотоками, то требуется два плагина, однако так как аудиосистема передает при загрузке плагина тип подключения, то можно реализовать два плагина в одном

Все что делает плагин - проводит необходимые параметры аудио и пытается
присоединиться по сокету к аудиосервису.
Необходимо понимать роль плагина - она огромна, так как без него конечная
аудиосистема не получит данных с hands free и никаких данных не отдаст, но с другой стороны плагин довольно подневолен, так как не может контролировать свою загрузку и даже не может контролировать количество одновременно загруженных плагинов. Высокоуровневая аудиосистема (плагины загружаются alsa аудиосистемой, в свою очердь она может использоваться для смешания потоков pulseaudio, может
использоваться gstreamer'ом, phonon'ом итп) может управлять плагины могут
загружаться аудиосистемой вследствии использования другими приложениями их api, например skype, gtalk, браузеры итд), поэтому вся логика и смена состояний должна быть вынесена подальше от плагинов
Важным моментом является добавление в плагины возможности логирования потоков.

4) Стек - определяет api, организует соединение с устройством, установку соединения и множество других сложных операций, для нас фактически является черным ящиком, поэтому нас интересуют только подстыковки к стеку : с одной стороны посредством api, с другой стороны к драйверуестественно в подстыковках реализовано также

логирование аудиопотоков, чтобы проверить что попадает на вход стека, и что получается на его выходе.

5) usb драйвер - взаимодействует с usb core путем конфигурации и обмена urb, и их разбором. Нас интересует здесь в первую очередь sco поток. Для получения максимального быстродействия в драйверах используются исключительно асинхронные операции, именно они позволяют получить максимальную пропускную способность, но одновременно приводят к сложностям в использовании. Асинхронная природа получения отправки urb приводит к необходимости использования дополнительных механизмов сихнхронизации.

В приведенной системе необходим выбор средств межпроцессного взаимодействия. Был выбран dbus для потока управления и сокеты для потока данных. Выбор обусловлен гибкостью первой системы и высокой пропускной способностью вторых.

# Глава III. Особенности реализации

**Конфигурация аудиосистемы**

В ОС GNU/Linux существует множество различных аудиосистем, обладающих своими достоинствами и недостатками. Такое разнообразие в свою очередь доставляет множество неудобств,

сравнивая с ос windows, где существуют безусловно различные интерфейсы для работы с аудио, но они тем не менее являются взаимозаменяющими. Грубо говоря являют собой идею обратной совместимости.
В GNU/Linux все не так - разнообразие аудиосистем не является результатом одной, а является результатом эволюции множества различных программных продуктов, совместимость между которыми является продуктом отдельных усилий.
Основная наша проблема - это то что мы не знаем api какой аудиосистемы будут использовать пользовательские приложение для проигрывания и записи звука.
Рассмотрим совместимость нашего аудиосервиса с тремя основными аудиосистемами (pulseaudio, ALSA, gstreamer), остальные аудиосистемы(phonon, oss, jack, ...) оставим на совести пользователей - при желании работу с ними можно настроить через поддерживаемые нами аудиосистемы.

**ALSA**

Advanced Linux Sound Architecutre - **продвинутая звуковая архитектура Linux** (англ. *Advanced Linux Sound Architecture*, **ALSA**) — пришла на смену более старой классической аудиосистеме OSS. Аудиосистема ALSA тесно связано с ядром Linux, так как на нижнем уровне представляет для работы с аудиоаппаратурой драйвера, которые являются модулями ядра, ALSA же является уровнем абстракции, связывающим уровень модулей ядра с уровнем

высокоуровневых приложений, работает с нихкой и стабильной задержкой, выпускается под лицензией GNU GPL.

Преимущество ALSA в широком использовании среди приложений, также полная приемственность и поддержка приложений, работающих с OSS (впрочем pulseaudio тоже может эмулировать oss, без посредничества alsa). Недостатки в ограниченных возможностях по смешиванию потоков и отсутсвия возможности взаимодействию с другими аудиосистемами (ALSA может взаимодействовать с другими только не зная об этом)

Для нас практический интерес представляют alsa плагины - shared library, лежащие в папке /usr/lib/alsa-lib и указанные в файлах конфигурации либо другим способом. Наш плагин будет называться libasound_module_pcm_btmhfp.so, где pcm означает, чт модуль работает с модулированным звуком (возможны модули другого назначения, например ctl - контроля громкости), btmhfp - название виртуального устройства.

Для того чтобы наш модуль мог успешно работать с ALSA и выполнять свои задачи он должен соответствать некоторой спецификации, определенной для всех плагинов.

SND_PCM_PLUGIN_DEFINE_FUNC (btmhfp) - точка входа, макрос определяет параметры

/

**********

**********

* Plugin entry point

*

* Parameters are hidden in ALSA defined macros:

* snd_pcm_t **pcmp -

* const char *name -

* snd_pcm_config_t *root -

* snd_pcm_config_t *conf -

* snd_pcm_stream_t stream -

* int mode -

**********

**********/

где pcmp - дескриптор pcm потока, его мы должны вернуть после создания плагина

name - имя плагина - передаетя аудиосистемой

root и conf - вспомогательные поля

stream - тип устройства, которым должен быть плагин (SND_PCM_STREAM_PLAYBACK для sink и

SND_PCM_STREAM_PLAYBACK для source), иными словами определяет направелние потока
mode определяет параметры pcm

Рассмотрим основные особенности плагина на примере playback.
Вызов SND_PCM_PLUGIN_DEFINE_FUNC (btmhfp) , преобразуется препроцессором в
int SND_PCM_PLUGIN_ENTRY(plugin) (snd_pcm_t **pcmp, const char *name,\
snd_config_t *root, snd_config_t *conf, \
snd_pcm_stream_t stream, int mode)
где SND_PCM_PLUGIN_ENTRY(plugin) преобразуется в _snd_pcm_##name##_open (то есть в нашем случае ALSA будет вызывать _snd_pcm_btmhfp_open)
В _snd_pcm_btmhfp_open мы должны создать плагин, в котором оставить callback'и, через которые ALSA сможет работать с нашим устройством.
Создается плагин вызовом snd_pcm_ioplug_create (&d->io, name, stream, mode). Контекст нашего плагина, который нам может понадобится в callback'е можно получить по указателю d->io.private_data.
Callback'и ALSA получит из поля d->io.callback - это указатель на структуру вида:

static snd_pcm_ioplug_callback_t btmhfp_playback_callback =

```
{
    .start          = btmhfp_sink_start,
    .stop           = btmhfp_sink_stop,
    .pointer        = btmhfp_pointer,
    .close          = btmhfp_close,
    .hw_params      = btmhfp_hw_params,
    .prepare        = btmhfp_prepare,
    .transfer       = btmhfp_sink_transfer,
    .poll_descriptors_count    = btmhfp_sink_pdc,
    .poll_descriptors    = btmhfp_sink_pd,
    .poll_revents       = btmhfp_sink_pr,
    .delay          = btmhfp_delay,
};
```

(инициализация структуры подобным образом есть расширение языка C - GNU C)

Где:

start - плагин уже загружен - теперь начинается проигрывание, для нас это означает, что мы должны запускать механизм передачи данных в медиасервис

stop - аудиосистема закончила проигрывание

pointer - возвращает текущую позицию потока

close - плагин выгружается, мы должны отключиться от медиасервиса

hw_params - возвращает параметры виртуального устройства

prepare - вызывается перед началом проигрывания

transfer - выполняет передачу данных из аудиосистемы, впрочем для повышения производительности можно не сразу отправлять данные в медиасервис
delay - возвращает сколько плагин хочет ждать до следующей передачи
Все остальные функции ведут учет количества дескрипторов

Таким образом был создан плагин. Так как плагин выполняется в пользовательском режиме его можно отлжаивать с помощью gdb и вести логирование в файловой системе.

**pulseaudio**

PulseAudio является кроссплатформенным звуковым сервером, стал заменой более старых звуковых сервером наподобие ESD. PulseAudio может работать на gnu/linux, solaris, freebsd и даже windows, распространяется под лицензией GNU GPL, а библиотеки в основном под GNU LGPL.

PulseAudio со временем стала очень попуярной благодаря наличию множеств преимуществ перед другими аудиосистемами в то время. В частности pulseaudio обладает возможностью раздельной установки громкости для разных программ, возможность смешивания аудиопотоков, в том числе с разных звуковых карт и с разными частотными характеристиками. Кроме того поддерживается большим

количеством прилоений и обладает демоном, через который можно производить конфигурацию.

Рассмотрим строение pulseaudio
Pulseaudio может выступать фронтендом практически для любой аудио-системы: так Mplayer, xine, alsa, aRTS, ESD, gnome приложения могут работать через pulseaudio.

Жизненный цикл pulseaudio

Pulseaudio фактически является прослойкой звуковой абстракции между приложениями и alsa. Сам по себе pulseaudio является демоном, запускаемым обычно из под учетной записи пользователя

alexxx 10155 11.8 0.4 160768 4468 ? S<sl 19:22 0:10 /usr/bin/pulseaudio --start --log-target=syslog

Существуют различные способы конфигурации системы для того чтобы пользовательские приложения выполняли роутинг в pulseaudio
Для этого существует gui приложение pavucontrol, которое осуществляет простейшую конфигурацию pulseaudio, также стандартным способ конфигурации является изменение конфигурационных файлов.
В pulseaudio используются те же понятия sink и source
sink - проигрывание

source - запись

Sink-Input. Приложение, поддерживающее PulseAudio, направляет свой вывод в один из объявленных Sink'ов и становится «входом Sink'а»: Sink Input.
Source-Output. - приложение само собирается принимать звук из некоторого Source - само становится выходом Source, то есть Source-Output.

Таким образом плеер создает sink-input, приложение записывающее звук - source-output, а GTalk - sink-input и source-output одновременно.

Рассмотрим еще некоторые важные для нас понятия
1) module - некоторая библиотека, загружаемая pulseaudio, для расширения его возможностей. Нас больше всего будет интересовать модуль module-alsa-sink.
2) client - некто, подключившийся к PulseAudio. Клиент создает потоки Sink-Input и Source-Output, выполняет управление демоном PulseAudio.

Многие медиа-приложения не умеют работать с pulseaudio напрямую, поэтому наша задача - их научить.

Для этого необходимо сконфигурировать alsa, чтобы она выполняла роутинг аудипотоков в pulse

```
pcm.btmhfp
{
   type btmhfp
}

pcm.!default
{
  type pulse
}
```

Первая часть создает виртуальное устройство btmhfp (bluetooth hands-free) и связывает ее ввод вывод с pcm. Вторая часть определяет его по-умолчанию связанным с pulseaudio, если ее не указать, то фактически пользователю будет необходимо лично выбрать pulse в качестве устройства ввода-вывода

Кроме этого надо сконфигурировать pulseaudio так чтобы он sink и source alsa-приложений, перенапревленных через .asoundrc, а также приложений, другим образом синхронизируемых с pulseauido, попали в наши alsa-plugin'ы.

Для этого необходимо в default.pa, конфигурационный файл pulseaudio, добавить строки:

```
load-module module-alsa-sink device=btmhfp sink_name=btmhfp-output
```

```
load-module module-alsa-source device=btmhfp source_name=btmhfp-input
set-default-sink btmhfp-output
set-default-source btmhfp-input
```

Первые две строки загружают модуль pulseaudio module-alsa-sink с параметрами btmhfp и btmhfp-output. Благодаря этому модулю pulseaudio будет выступать в качестве посредника между приложением использующую alsa интерфейсы и нашим модулем, загруженным в режиме проигрывания.
Вторая строка делает то же самое для записи.
Третья и четвертая строка устанавливают btmhfp-output и btmhfp-input стандартными устройствами ввода-вывода.

Существуют другие способы конфигурации pulseaudio - например встроенная утилита pactl.

**GStreamer**

GStreamer - кроссплатформененый фреймворк, написан на си, базируется на системе типов GObject. Используется множеством мультемидийных приложений такими как медиаплееры, звуковые редакторы, потоковые серверы. Первоначально был спроектирован так, чтобы поддерживать кроссплатформенность. Благодаря этому работает как на POSIX платформах, так и на Microsoft Windows, Symbian, Android и других. Кроме кроссплатформенности

предоставляет биндинги для других языков программирования таких как C++, Perl, Ruby, Python, C#. Является свободным программным обеспечением и распространяется под лицензией GNU LGPL.

Основным пользователем технологии GStreamer является среда рабочего стола GNOME, таким образом gstreamer интегрирован в GNOME начиная с версии 2.2. Кроме того используется в приложениях не связанных с GNOME, например в медиаплеере Chameleo, Songbird, также может быть бекэндом аудиосистемы Phonon. Кроме того является частью операционной системы Maemo, на которой базируются линейка смартфонов Nokia.

В целом GStreamer состоит из базового ядра, набора утилит и подключаемых модулей. Рассмотрим некоторые понятия.

Элемент - компонент GStreamer. Из элементов GStreamer составляются цепочки, места стыков элементов называются коннекторами (pads). Через коннекторы можно переправлять данные, коннекторы могут быть двух типов - аналогично pulseaudio - sink pad и source pad. Каждый элемент может иметь несколько коннекторов разных типов, в частности можно сказать что модель в некотором роде была заимствована у DirectShow, который появился на несколько лет ранее.

Популярный медиаплеер Totem использует GStreamer, поэтому нас в первую очередь интересует как сделать так, чтобы GStreamer работал с нашими alsa плагинами. Для этих целей можно использовать элемент

AlsaSink/AlsaSource, но можно поступить проще и воспользоваться системной конфигурацией gstreamer’a.

По традиции изменение конфигурации приложений GNOME окружения происходит через использование системной утилиит семейства gconf.

GConf - это система, используемая в GNOME как замена прямого использования конфигурационных файлов, присутствует как некий аналог реестра Microsoft Windows. База данных gconf использует систему каталогов, располагающуюся в ~/.gconf. Метаданные хранятся в xml формате. Изменения и контроль изменений ведется демоном gconfd/

Получить доступ к GConf можно с использованиемC, C++, Python, Perl, Java и другие языки.

Кроме внесения изменений посредством использования биндингов (которые работают  непосредственно через api) можно использовать системные утилиты, в частности gconftool-2.

gconftool-2 - консольное приложение (в отличии от графического gconf-editor)

gconftool-2 --type boolean --set /apps/nautilus/desktop/volumes_visible true

Здесь --type boolean - тип устанавливаемого значения, --set /apps/nautilus/desktop/volumes_visible - путь до значения, true - выставляемое значение.

В /system/gstreamer/0.10/default находятся настройки gstreamer’a. musicaudiosink отвечает за бэкэнд аудиосистемы. Стандартный вариант - autoaudiosink. Для нас конечно интересен бэкенд через наши alsa плагины. Для этого надо указать строку вида:

gconftool-2 --type string --set /system/gstreamer/0.10/default/musicaudiosink alsasink device=btmhfp

После отключения медиасервиса нужно будет указать

gconftool-2 --type string --set /system/gstreamer/0.10/default/musicaudiosink autoaudiosink

Существуют другие варианты конфигурации - наиболее интересным вариантом конфигурации gstreamer’a является синхронизация его работы с другим аудиосервером - pulseaudio. Этого можно достичь использованием модуля pulsesink:

gconftool-2 --type string --set /system/gstreamer/0.10/default/audiosink pulsesink device=btmhfp

gconftool-2 --type string --set
/system/gstreamer/0.10/default/musicaudiosink autoaudiosink

**Автоматическая конфигурация**

Нельзя придумать ничего хуже ситуации когда пользователь вынужден самостоятельно править конфигурационные файлы с тем чтобы настроить аудиосистему для работы с hands-free.

Мы уже рассмотрели способы как можно сконфигурировать аудиосистемы, теперь вопрос состоит в том как это сделать программно. Следует также учесть возможность ошибок, которые могут возникнуть в процессе работы, после чего пользователь не сможет после отключения аудиосервиса нормально работать с аудиосистемой - такого не должно быть - необходимо предусмотреть варианты перезагрузки, зависания сервиса. Сделать так, чтобы хотя бы после последующего запуска медиасервиса настройки аудиосистемы восстановились.

Предоставим gui право конфигурировать pulseaudio и gstreamer, а медиасервису ALSA. Так как gui на C++ с использованием фреймворка Qt, то можно использовать различные варианты выполнения конфигурации.

1) gconftool-2. Можно использовать system - библиотечную функцию из glibc, однако есть более гибкий вариант - с использованием объект

QProcess. Он позволяет более существенно контролировать процесс выполнения процесса, получать и отправлять ему сигналы, читать и писать в его потоки ввода-вывода.

Рассмотрим момент подключения наших плагинов к gstreamer’у

```
QStringList params;

params.append("-t");
params.append("string");
params.append("-s");
params.append("/system/gstreamer/0.10/default/musicaudiosink");
params.append("alsasink device=" +
    QString(CFG_AUDIOD_ALSA_HFP_PLUGIN));

QProcess::startDetached ("gconftool-2", params);
```

startDetached проверяет был ли системой запущен процесс, после этого процесс выполняется отдельно.

процесс отключения медиасервиса - переключения gstreamer’а на стандартные устройства

```
QStringList params;

params.append("-t");
```

```
        params.append("string");
        params.append("-s");
        params.append("/system/gstreamer/0.10/default/musicaudiosink");
        params.append("autoaudiosink");

        QProcess::startDetached ("gconftool-2", params);
```

Рассмотрим теперь настройку pulseaudio - изменением файла /etc/pulse/default.pa. Выполнить это можно с помощью библиотечных средств c++, но удобнее применить средства Qt.

Откроем default.pa для чтения и записи и инициализируем строки

```
    QFile file("/etc/pulse/default.pa");
    QString s1("load-module module-alsa-sink device=btmhfp
sink_name=btmhfp-output\n");
    QString s2("load-module module-alsa-source device=btmhfp
source_name=btmhfp-input\n");
    QString s3("set-default-sink btmhfp-output\n");
    QString s4("set-default-source btmhfp-input\n");

// добавляем строки в файл

        file.open(QFile::WriteOnly | QFile::Text | QIODevice::Append);
        file.write(s1.toLatin1());
```

```
        file.write(s2.toLatin1());
        file.write(s3.toLatin1());
        file.write(s4.toLatin1());
// теперь перезапускаем pulseaudio, чтобы изменения вступили в силу
QProcess::execute("pulseaudio -k");
```

аналогичным образом конфигурируем pulseaudio для обычной работы

```
        file.open(QFile::ReadWrite | QFile::Text);
        QString s(file.readAll());
        s.remove(s1);
        s.remove(s2);
        s.remove(s3);
        s.remove(s4);
        file.resize(0);
        file.write(s.toLatin1());
        file.close();
        QProcess::execute("pulseaudio -k");
```

Такой способ работы хорош тем, что даже если по ошибке записи о конфигурации были занесены несколько раз - удалены будут все записи.

Следует отметить еще несколько моментов:
1) Всегда необходимо делать резервную копию конфигурационного файла, так чтобы даже в случае сбоя в работе сервиса можно было

программно, либо хотя бы вручную восстановить первоначальное состояние файла.
2) В случае нехватки памяти или ресурсов система может посылать процессам сигналы с тем, чтобы они завершились. Время от времени пользователи мог сами используя системные команды ( kill, pkill итп ) убить наш процесс. Что это означает для нас и для пользователя? Для нас, что мы не можем гарантировать правильную конфигурацию аудиосистемы после завершения нашего процесса. Для пользователя то что с большой вероятностью он больше не сможет ничего прослушать/записать.
В большинстве случаев можно назначить обработчик сигнала, в котором провести конфигурацию системы и спокойно завершиться. Сделать это можно классическим способом:

```
struct sigaction sa;

sa.sa_handler = interceptor;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;

int sa_res = sigaction(SIGTERM, &sa, NULL);
```

Теперь как только кто-либо пошлет нашему приложению сигнал SIGTERM (сигнал об окончании работы) вызовется callback interceptor.

С другой стороны так как наше приложение выполняется под контролем QApplication можно использовать CloseEvent.
Конечно если кто-то решит использовать для завершения медиасервиса SIGKILL - тут уже ничто не поможет кроме правильной обработки при следующем запуске.

Конфигурация ALSA.
Конфигурацию ALSA будем проводить из медиасервиса, так как в отличие от gui его состояние не зависит от состояния конфигурации остальных зависимых аудиосистием.
Конфигурация ALSA происходить путем замены оригинального файла настроек ~/.asoundrc на наш. Добавлять в пользовательский файл свои настройки не стоит, потому как структура asoundrc может быть сложна и запутанна.
Стандартными средствами переименовываем существующий файл в ~/.asoundrc.orig, а на его место кладем наш, после конца работы свой удаляем, а старый переименовываем.
Конечно рассматриваем когда файла конфигурации нету (можно оставить пустой файл) и моменты преждевременной смерти сервиса - правильной обработки сигналов.

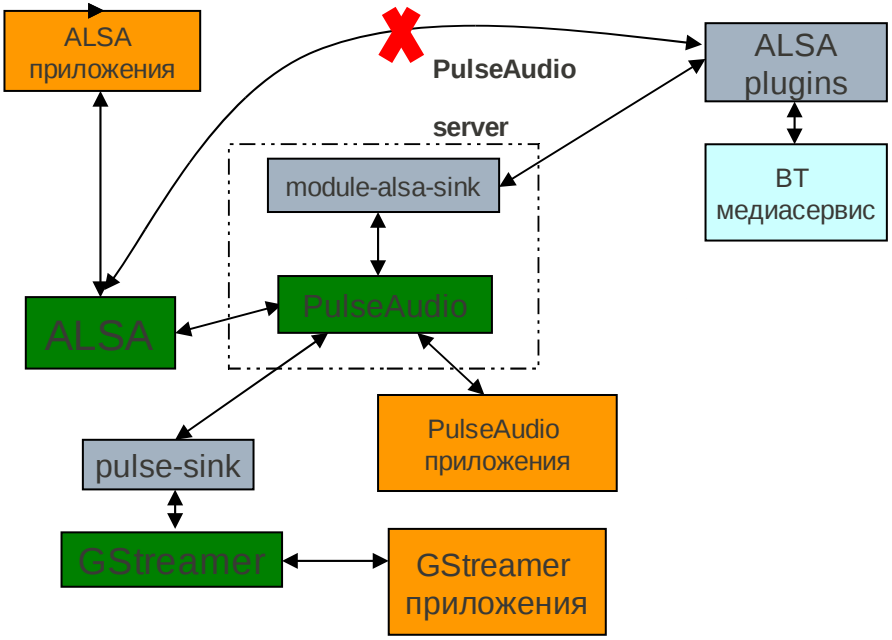


*Рисунок 7. Полученная аудиосистема*

**Верификация потоков.**

Одна из важнейших стадий разработки, одновременно сопровождающая весь процесс разработки и отдладки является верификация аудио-потока. Идея верификации довольно проста - в различных местах следования аудио-потока сохранять в каком либо виде данные. В каком виде? Естесственный способ - сохранение в ФС,

но тем не менее могут быть и другие способы, в зависимости от ситуации.
Настоящие цели верификации очевидны - в условиях сложной системы это
практически единственный способ точно проверить правильность прохождения аудио-потока.

Как уже говорилось - простейший способ - логирование потока в файл. Этот способ доступен в ккомпонентах имеющих доступ к файловой системе, таким образом в драйвере использование логирования невозможно, так как нету прямого доступа к файловой системе.

Рассмотрим некоторые обозначения, которые будет удобно использовать при объяснении логирования.

pcm-данные в процессе пересылки из аудиосистемы в устройство и наоборот проходят большой путь, и правильность передачи зависит от работы множества компонент. Прохождение данных удобно сравнить с понятием потока используемых в математике и физике. Наиболее удобное термины для нашего случая введены в теории автоматического управления.
1) Поток pcm-данных - сигнал. С учетом того что pcm-данные и являются модулированным сигналом - совпадение буквальное
2) Искажения данных - внешние воздействия, являются результатом программных или системных воздействий, могут носить детерминированный или стохастический характер, воздействуют

некотором определенным образом на сигнал - начиная от привнесения помех и заканчивая превращением в белый шум
3) Точки логирования - узлы. В идеале наличие узлов не вносит изменений в систему и в сигнал в частности. В реальности изменения есть, но они должны быть незначительны. Так же и в случае логирования - влияние логов на поведение программного комплекса должно не быть значительным.
4) Устойчивость - устойчивость системы к программным и системным ошибкам
5) Прохождение сигнала направленное, то есть можно выделить сток и исток - sink и source

При желании можно вводит и другие близкие понятия, однако они в целом не так важны для рассматриваемого вопроса.

**Sink** (англ. низина) — «сток», принимающий звуковой поток. Представляет выход звуковой карты: линейный выход, наушники, колонки ноутбука;
**Source** (англ. источник) — источник звука, создающий звуковой поток. Представляет вход звуковой карты: линейный вход, микрофон.

Простота способа с его эффективностью и высокой применимостью определяет его использование в большинстве случаев. Оно позволяет:
- определить случаи прохождения/непрохождения звука путем сравнения размеров дампа в точках стока. Естественно в точках стока

размеры дампов не будут точно совпадать, это связано со следующими факторами

- поток не распространяется мгновенно по системе, кроме того при правильном режиме работы поток может не доходит до конца конвейера

- наличие у файловых поток буферов ввода-вывода (впрочем это можно исправить
использованием функцией fflush — для сброса буфера и setvbuf
- можно установить буфер нулевой длины).

В целом точное совпадение не является важным
- определение точек появления шумов - так как в HFP/HSP в отличии от A2DP на вход hci поступает в не зашифрованном виде, то нету проблемы сконвертировать дамп стока в аудио файл, который можно будет прослушать средствами ОС. Этот метод позволяет определить места появления шумов/искажений звука. Не стоит недооценивать мощь этого метода, так как в случае обнаружение багов, при которых звук подвержен искажениям сложно определить источник проблемы и главное - место в котором появляется искажения. Исследование дампа стока позволяет определить кто повинен в искажениях - модули ALSA, медиасервис, аудио-библиотека, стек или драйвер.
Естественно метод не универсален, так как не способен вылавливать проблемы появляющиеся из-за

- появления взаимных блокировок, проблем с производительностью, других
проблем, приводящих к "зажжевыванию звука", появление задержки потока pcm. Таким образом дапм теряет привязку
к реальному времени

- привязка к ФС.

- данных в потоке, появляющихся в результате взаимодействия с аудиосистемой, нужных для ее функционирования (при ее инициализации, для проверки того что стоки/источники работают), но не несущих смысловой нагрузки при прослушивании ( при запуске любой аудиосистемы она пытается проверить правильность своей конфигурации. В нашем случае pulseaudio находит в конфигурационных файлах команду загрузить module-alsa-sink и в качестве бэкэнда использовать btmhfp. Естественный способ системе понять что происходить - начать отправлять и запрашивать данные из модулей; естесственно эти для пользователя эти данные не имеют никакого смысла, однако по значениям, возвращаемым модулями, система может понять, что конфигурация аудиосистемы была составлена неправильно. В случае pulseaudio для проверки btmhfp-output он отсылает порцию нулевых блоков, а также запрашивает некоторое количество данных (~ нескольких Кб, если рассматривать поток нулей как поток pcm данных, то для устройства вывода это будет тишина, соответственно пользователь не заметит такого тестирования системы) у btmhfp-input. Если за это время модули не начинают правильно отвечать pulseaudio высвечивает пользователю, что аудиосистема настроена неправильно ).

У логирования в файл есть фактическое ограничение - необходимость взаимодействия с файловой системой. Данное ограничение определяющее для работы
с драйвером. В драйвере система логирования была реализована другим способом.
Реализация связана с естественным желанием сделать нечто похожее на
логирование в файл. Этого можно достичь использованием виртуальной файловой системы /proc. Однако надо иметь в виду что операции с ней выглядят значительно иначе, нежели чем работа с обычной ФС, так как содержимое файла получается путем прямого обращения к драйверу с указанием смещения относительно начала файла и желаемого размера буфера. В случае же файловой системой операциями чтения/записи руководят драйвера и службы файловой системы.
Итак, для логирования необходимо выделить внутренний буфер (в статической или динамической памяти),
зарегистрировать файл в /proc и обрабатывать запросы на чтение лога (также можно предоставлять операции записи в файл извне, но нам это незачем).
При таком способое логирования появляются проблемы связанные с размером лога - kmalloc, предназначенная для выделения страниц памяти в режиме ядра, значительно отличается от своего аналога в glibc malloc. Отличия связаны с тем, что kmalloc выделяет память постранично, то есть фактически можно выделить только объем

памяти кратный 4, 6 … кб в зависимости архитектуры. Более серьезное ограничение связано с неспособностью выделять kmalloc’ом большие объемы памяти - как правило верхняя граница порядка 128 Кб. Связано это с отсутсвием виртуальной памяти и работой в реальных адресах, а также работой в контексте прерываний и других специфичных для ядра условиях.

Существуют другие более сложные способы выделения динамической памяти в ядре, но в целом их сложность для такой простой задачи приводит к поиску других вариантов решения задачи. Самым простым является использование статического буфера, связанного со структурой, используемой пользователем для логирования. Это делает невозможным во время работы выбор количества логов и их размеров, впрочем это некритично.

Пример:

```
struct btm_dump
{
   char   btm_dump[LOG_SIZE];
   int    btm_dump_offset;
   struct proc_dir_entry* entry;
   char*  name;
#ifdef MUTEX_BLOCK
   struct mutex dump_mutex;
#else
   spinlock_t dump_spinlock;
```

```
    unsigned long irq_state;
#endif
};
```

рассмотрим этот способ

btm_dump – буффер фиксированного размера, в которые будем складывать даные лога.
btm_dump_offset – смещение внутри буффера.
Entry – структура для работы с файловой системой /proc
name – имя, выбранное пользователем, под которым файл будет зарегистрирован в ФС.
dump_mutex – мьютекс, на случай если мы все-таки решим использовать мьютексы.
dump_spinlock – спинлок, для активного ожидания ресурса.
irq_state – используется вместе со спинлоком. В этой переменной сохранается состояние прерываний до начала синхронизации.

Добавление данных является тривиальной задачей. Более сложной задачей является чтение. Для того чтобы внешний процесс смог прочитать содержимое файла – необходимо зарегистрировать callback для чтения. Примерный вид callback'а:

```
int btm_read_proc (char *page, char **start, off_t off, int count, int *eof,
void *data)
```

```
{
    int len;

    struct btm_dump* dump = data;

    BT_DBG("page %p, start %p, off %ld, count %d, eof %p, data_unused
%p, buffer offset %d frome %d",
                page, start, off, count, eof, data, dump->btm_dump_offset,
LOG_SIZE);

    *start = page;
    *eof = 1;
#ifdef MUTEX_BLOCK
    mutex_lock( &dump->dump_mutex );
#else
    spin_lock_irqsave( &dump->dump_spinlock, dump->irq_state);
#endif
    if( off > dump->btm_dump_offset )
    {
            BT_DBG("off > dump->size");
        len = 0;
    }
    else if( off + count > dump->btm_dump_offset )
    {
        BT_DBG("off + count > dump->size");
```

```
        len = dump->btm_dump_offset - off;
    }
    else
    {
            BT_DBG("else");
        *eof = 0;
        len = count;
    }

    memcpy( *start, &dump->btm_dump[off], len );
#ifdef MUTEX_BLOCK
    mutex_unlock( &dump->dump_mutex );
#else
    spin_unlock_irqrestore( &dump->dump_spinlock, dump->irq_state );
#endif
    BT_DBG("eof - %d, len %d", *eof, len );
    return len;
}
```

Лучше всего процесс чтения демонстрируется при использовании стандартных утилит: cp, less.

```
alexxx@alexxx-laptop:~$ ls -l /proc/hfp_sco_in.raw
-rwxrwxrwx 1 root root 0 2011-06-24 04:28
/proc/hfp_sco_in.raw
```

Размер всегда отображается нулевым, так как у ядра нету сведений о реальной возможностях по считыванию файла

Однако его можно скопировать:

alexxx@alexxx-laptop:~$ cp /proc/hfp_sco_in.raw .
alexxx@alexxx-laptop:~$ ls -l hfp_sco_in.raw
-rwxr-xr-x 1 alexxx alexxx 1000000 2011-06-24 04:32 hfp_sco_in.raw

Видим, что буфер заполнен
btm_read_proc: page e7537000, start ef4aff04, off 0, count 3072, eof ef4aff08, data_unused f97dda40, buffer offset 1000000
btm_read_proc: else
btm_read_proc: eof - 0, len 3072
btm_read_proc: page e7537000, start ef4aff04, off 3072, count 3072, eof ef4aff08, data_unused f97dda40, buffer offset 10000
btm_read_proc: else
btm_read_proc: eof - 0, len 3072
btm_read_proc: page e7537000, start ef4aff04, off 6144, count 3072, eof ef4aff08, data_unused f97dda40, buffer offset 10000
btm_read_proc: else
btm_read_proc: eof - 0, len 3072

Видим, что cp предпочитает зачитывать данные блоками по 3072 байта (3Кб).

Отдельный вопрос в этом случае - синхронизация как на запись, так и на чтения. Для синхронизации в режиме ядра можно исползовать как обычные семафоры и мьютексы, так и более мощное, но потенциально опасное в применении, средство синхронизации - спинлоки. В данном случае использовать мьютексы нельзя, так как это приведет к значительным издержкам, кроме того, так как наш драйвер взаимодействует с usb core задержки мгновенно повлияют на все устройства, пользующимся usb ядром.

Идея работы спинлока - активное ожидание захваченного ресурса - это хорошее решение для нашего случая, так как ресурс захватывается на очень короткое время.

Опасность спинлоков состоит в том, если спинлок не будет разблокирован в короткое время - его активное ожидание захватит всю работу ядра.

Кроме простого прослушивания аудио-потока можно использовать другие способы
сравнения

- сравнение дампов разных стоков - позволяет определить потерю/появление лишних пакетов в разных дампа, такую проверку можно совершить используя diff, kdiff3.

alexxx@alexxx-laptop:~$ kdiff3 /tmp/hfp_sco_in.raw /tmp/hfp_capture_thread.raw

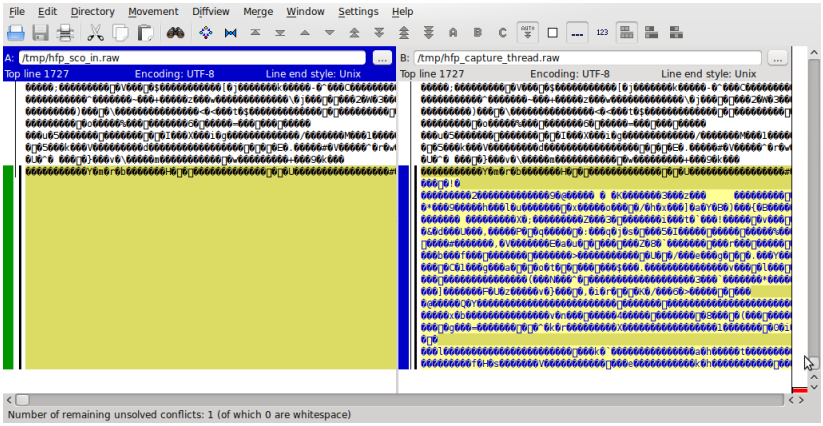


*Рисунок 8. Сравнение двух стоков*

Используя программу сравнили стоки в двух разных точках. Видим, что отличаются только концы.

Следует отметить, что некоторые различия дампов не обязательно означают наличие ошибок (по уже сказанным причинам)

- более точный способ проверки - подмена первоначального стока/источника на некоторую периодическую заданную последовательность
Для проверки playback потока заменяем данные в alsa плагине
memset( buff, 0, sizeof( buff ) );
либо на некую последовательность и сверяем.
Для проверки capture потока подменяем поток с hci в драйвере.

Верифицировать поток можно точно с использованием hex-редактора, либо грубо с использованием аудиоредактора.

Вывод: мы рассмотрели мощный способ отладки работы сервиса, в связи с частотой его применения его код был в конце концов интегрирован в код проекта.
Естественно дамп присутствует только в отладочных сборка при самом высоком уровне трассировки.

bluetooth usb драйвер работает в терминах hci команд - занимается получением и отправкой различного типов пакетов. Ядро linux поддерживает различные способы взаимодействия(получение и отправка данных) с usb устройствами. Центральным понятием является так называемый endpoint - некоторый интерфейс, через который usb драйвер взаимодействует с usb ядром, а то в свою очередь уже с самим устройством.
Такими интерфейсами традиционно являются:

- CONTROL - конфигурации, настройки устройства - не предназначены для непосредственной передачи данных
- INTERRUPT - передача инициируется путем генерации прерывания, предназначена для передачи небольших объемов данных
- BULK - для передачи огромных массивов данных
- ISOCHRONOUS - для передач данных, просходящих с точной переодичностью

По спецификации sco передача пакетов должна просиходить каждые 1 миллисекунду, естественно обмен пакетов с чипсетом происходять по изохронному интерфейсу. Передача происходит посредством передачи очереди urb, каждый из которых содержит очередь фреймов, в которых находятся целые или разделенные между фреймами sco-пакеты. sco пакет содержат заголовок, в котором находится дескриптор, устанавливаемый чипсетом при соединении, и размер пакета. Таким образом пакеты могут быть поделены между urb, фреймами и кроме того еще могут быть и произвольного размера - это делает разбор urb сложным делом.

В процессе разбора (а также благодаря тому, что разные чипсеты различным образом могут располагать пакеты на endpoint) требует конторля.

Рассмотрим на примере capture (Реализация для playback значительно проще, так как ошибится при собственоручном формировании очереди пакетов сложнее). Для мгновенной готовности к передаче потока

драйвер должен постоянно считывать urb из ядра и складывать так, чтобы стек могу получит аудиоданные мгновенно за некоторый, определяемый здравым смыслом и отсутсвием задержек, период.
Один из возможных вариантов - использование двух кольцевых буфферов **
Этот метод несколько проще и наглядней метода, применяемого в bluez - использование механизма сокетных буферов.

Понятия:
usb - Universal Serial Bus — универсальная последовательная шина
urb - USB Request Block - блок, запрашиваемый usb, единица данных обмена между ядром usb и usb устройством
frame - при изохронной передаче urb содержит некоторое количество фреймов, в которых содержаться данные пакетов
mtu - max transfer unit - максимальная длина фрейма/пакета
urb queue - очередь из urb

Стратегия обработки sco пакетов в драйвере:
1) Зарегестрировать usb устройтсво, назначить ему интерфейсы итп
2) Так как у нас будет два типа запросов - IN и OUT - для каждого нужно выделить свою очередь urb (внешнее представление struct urb) для этого используются функции ядра usb_submit_urb(urb, GFP_ATOMIC);
GFP_ATOMIC означает требование на мгновенную передачу в атомарном контексте

3) Так как struct urb всегда выделяется динамически и принадлежит ядру, кроме того так как локальная работа с urb или передача ее в другие структуры/функции может привести к нарушению работы системы подсчета ссылок, то удобно использовать anchor’ы - якоря - пользовательские структуры, создаваемые локально и связываемые ядром с urb. Поэтому чтобы не потерять свои urb - свяжем их с якорями, хранящимися в структуре нашего устройства.
usb_anchor_urb(urb, &bdev->isoc_in_anchor) - связывает urb с якорем bdev->isoc_in_anchor, где bdev - указатель на структуру нашего устройства
4) мы должны заполнить поля urb, чтобы связать его с функциями нашего драйвера

```
   urb->dev      = bdev->udev;
// указатель на структуру нашего устройства
   urb->pipe     = pipe;
// указатель на используемый для этого urb endpoint’a
   urb->context  = context;
// контекст, который позволит получить необходиму информацию о urb при вызове
// callback’a ядром
   urb->complete = btusb_isoc_complete;
// этот callback вызовется когда ядро закончит операции с urb (успешно/неуспешно)
// можно будет узнать из полей
   urb->interval = bdev->isoc_rx_ep->bInterval;
// операция ввода/вывода будет повторяться с интервалом 100мс
```

```
    urb->transfer_flags  = URB_ISO_ASAP;
// изохорный endpoint
    urb->transfer_buffer = context->mem;
// буффер, который будет использован urb для передачи данных
    urb->transfer_buffer_length = context->size;
// его длина, в соответсвии со спецификацией
```

4) После этого можно начинать подписывать urb

```
err = usb_submit_urb(urb, GFP_KERNEL);
```

сразу после этого urb может быть обработан ядром, как только операции обработки закончатся будет вызван callback isoc_complete

5) как только вызван isoc_complete необходимо определить какого типа urb пришел (IN/OUT) из контекста, после этого начать разбор sco пакетов

Рассмотрим случай IN - то есть capture поток.

Сперва необходимо проверить статус urb, если urb без ошибок, то можно начинать разбирать фреймы. Каждый фрейм должен начинаться либо с заголовка, либо с продолжения пакеты, либо он ошибочный.

Заголовок пакета может быть описан структурой

```
struct hci_sco_hdr {
```

```
    __le16    handle;
    __u8    dlen;
} __attribute__ ((packed));
```

2 байта отводится под дескриптор sco-соединения. Этот дескриптор получает стек при установке синхронного соединения. Дескриптор должен совпадать с тем, что получил стек, иначе пакет неправильный. Еще байт отводится под длину пакета - очевидно тело пакета не может быть больше 255 байт. Пакеты могут разрываться между фреймами и даже между urb, в зависимости от чипсета.

Каждая из компонент в каждый момент времени должна пребывать в определенном состоянии - неправильная работа одной единственной компоненты как правило приведет к полной неработоспособности системы - поэтому так важно переход в правильные состояния компонент и их правильный обмен состояниями

**Synchronous Connection Orientied link.**

Синхронное соединение. Данный тип соединения предназначен для передачи голосовых данных. SCO соединение на самом деле есть множество зарезервированных асихронных соединений. Каждое устройство передает голосовые данные в заданные периоды времени. При sco соединении нет повторных передач, одна, но существует режим, поддерживающий исправление ошибок. ?SCO packets may be sent every 1, 2 or 3 timeslots.

Enhanced (расширенный) SCO (eSCO) поддерживает повторную передачу, имеет множество вариантов пакетов, имеет большие интервалы между отдельными ACL, что увеличивает возможности мультипрофайлинговой работы.
Очевидно HFP/HSP использует для передачи и получения данных sco. Не вдаваясь в подробности работы sco, предоставляемого посредством api стека, заметим что данные на hci поступают и приходят в незашифрованном виде (в отличии от a2dp например), то есть raw формате.

**Формат аудиопотока**

Необходимо понимать как кодируется аудиопоток, передаваем устройством в аудиосистему и наоборот.
Рассмотрим вариант playback'a, так как случай capture аналогичный.
На вход аудиосистемы с звуковой карты, tcp/ip, другой аудиосистемы поступает сигнал, подвергшийся импульсно-кодовой модуляции (pcm - pulse code modulation), то есть оцифрованный. В наш alsa-плагин звук попадает в raw формате, в соответствии с параметрами, заданными модулем при загрузке.
Фактически при pcm происходит
формат сэмпла  определяется:
- количеством бит, приходящихся на сэмпл - ими кодируется уровень цифрового сигнала (16 бит)
- количество каналов - для hfp один

- тип кодирования уровня - множество вариантов того, как в заданном количестве бит хранится значения уровня - целые (знаковые, беззнаковые), с плавающей точкой и множество других вариантов. В нашем случае используются знаковые 16-битные уровни
- частота сэмплов - Фактически количество сэмплов, приходящихся на секунду аналогового звука. Вместе с количеством бит на семпл определяет битрейт звука
в нашем случае частота 8000 Гц

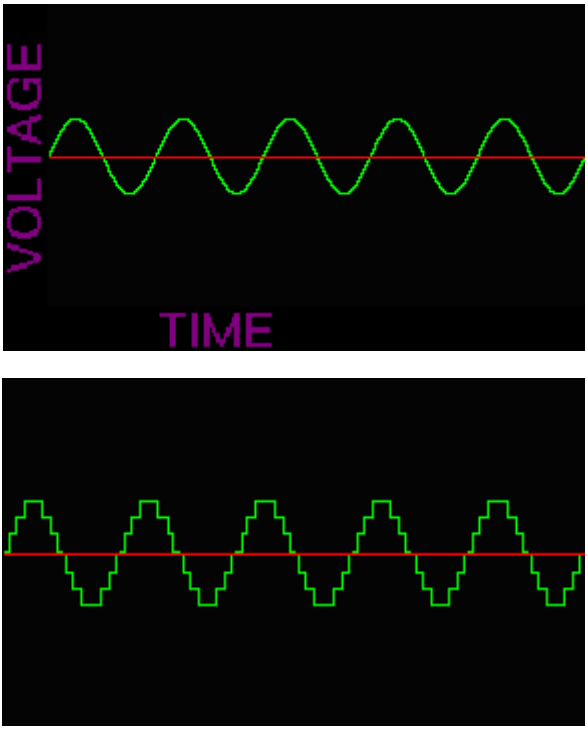


*Рисунок 9 pcm преобразование аналогового сигнала*

Raw формат сам по себе не предназначен для проигрывания, так как проигрывателю будут неизвестны параметры модуляции. Один из простейших медиа форматов является wav - сам по себе является контейнером для pcm.

В начале wav файла находится заголовок такого вида:

после заголовка идут pcm данные.

Таким образом из pcm легко сформировать wav файл либо программным добавлением заголовка, либо с использованием утилит, например утилиты sox

```
#!/bin/bash

if [ $# != 2 ]; then
  echo "bad command line!"
  exit
fi;
echo
"#################################################################
##"
sox -r 8k -e signed -b 16 -c 1 $1 $2
echo
"#################################################################
##"
```

Выходной файл будет в wav формате на 44 байта больше

Сам аудиопоток и представляет из себя набор подобных сэмплов. Огромная сложность состоит в передаче без изменений и задержек через стек до самого драйвера. В том что аудиопоток не модифицируется по пути следования к hci позволяет его без дополнительной раскодировки логировать в файл.
Вопрос логирования аудиопотока рассматривается в отдельной главе.

# Глава IV. Полученные результаты

Большой выбор определяет большие возможности. Но вместе с этим приносит большие сложности – разнообразие аудиосистем становится тяжелой ношой для нашего медиасервиса. Тем не менее с затратой некоторых усилий оказалось возможным конфигурация аудиосистем, так чтобы в целом они приблизились по свойствам к единой монолитной системе. Был продемонстрирован возможный вариант конфигурации и фактические способы автоматической конфигурации, рассмотрены сложности связанные с автоматической конфигурацией такие как возможные нарушения целостности аудиосистемы, проблемы с конфигурацией отдельных компонент.

Также отдельным вопросом для рассмотрения была проблема проверки правильности аудиопотоков. Методы, приведенные в работе, были придуманы и опробованы на практике. Их использование позволило качественно повысить стабильности сервиса.

В результате был реалеизован HFP. Для определения правильности работы протокола были использованы test case'ы, но полноценного тестирования пока что не было произведено. Кроме того в данный момент не поддерживается последний стандарт SCO (в последнем стандарте синхронное соединение активно только в момент передачи данных, в остальные моменты устройства не запрашивают у друг

друга голосых данных – это повышает время работы устройств, работающих от аккумуляторов).

## Список используемых материалов


1. Linux Device Drivers, Third Edition. Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman
2. Linux Kernel Developmen. Jonathan Corbet, LWN.net . Greg KroahOHartman, SuSE Labs / Novell Inc. Amanda McPherson, The Linux Foundatio
3. HANDS-FREE PROFILE 1.5 . Car Working Group
4. Hands-Free Profile . Adopted Version 1.0. Bluetooth SIG Car Working Group
5. Serial Port Profile . BLUETOOTH SPECIFICATION Version 1.1
6. Bluetooth Headset наушники Prolife BT 56
7. Bluetooth Dongle Broadcom 2.1 + EDR
8. Bluetooth Dongle Acrop 2.1 + EDR