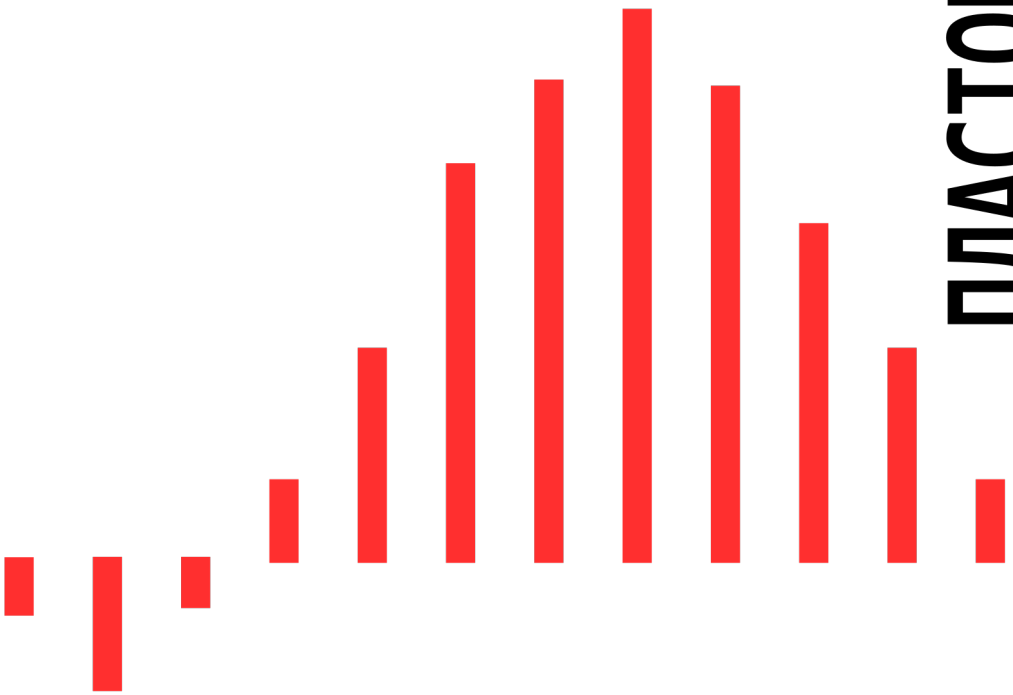
ПЛАСТОВ И.В.



ИЗУЧАЕМ Mediastreamer2

Аннотация

Книга освещает вопросы обработки медиа и стриминга с помощью библиотеки *Mediastreamer2*. Материал книги позволяет изучить *Mediastreamer2* и применить эти знания на практике. Она написана так, что способствует погружению в технологию *Data Flow* читателя не обладающего глубокими знаниями о VoIP. Требуются базовые навыки программирования на Си.

В объемах достаточных для начала работы в области VoIP, книга содержит информацию о правилах построения звуковых графов, работой готовых фильтров, методах разработки фильтров собственного дизайна. Кроме этого, освещены вопросы отладки фильтров и оценки вычислительной нагрузки на процессор. Даны рекомендации и примеры оптимизации нагрузки.

Для широкого круга читателей

Версия книги: 1.2

Лицензия

«Attribution-NonCommercial-ShareAlike»
(«Атрибуция-Некоммерчески-СохранениеУсловий») 4.0
Всемирная (CC BY-NC-SA 4.0)

Без дополнительных ограничений – Вы не вправе применять юридические ограничения или технологические меры, создающие другим юридические препятствия в выполнении чего–либо из того, что разрешено лицензией.

Замечания:

Вы не обязаны действовать согласно условиям лицензии, если конкретная часть материала находится в общественном достоянии или если такое использование вами материала разрешено согласно применимому исключению или ограничению авторских прав.

Вам не даётся никаких гарантий. Лицензия может не включать все разрешения, необходимые вам для использования произведения (материала) по вашему замыслу. Например, иные права, такие как право на обнародование, неприкосновенность частной жизни или неимущественные права могут ограничить вашу возможность использовать данный материал.

Оглавление

Предисловие

Уважаемый читатель, вы держите в руках книгу посвященную использованию VoIP-движка *Mediastreamer2* в ваших собственных разработках. Изначально, на основе опыта многолетнего практического применения *Mediastreamer2,* автором в сетевых блогах был написан цикл статей. Они легли в основу этой книги.

Написано и сверстано в одно лицо, поэтому наверняка в тексте есть опечатки. Если в процессе чтения вы обнаружите ошибки или какие-то несоответствия, прошу присылать ваши замечания и пожелания на электронный адрес:

plastov.igor@yandex.ru

Книга бесплатная, но если она вам понравилась, то ничто не мешает перевести 50 рублей автору на Яндекс-кошелек 410015536289824.

Одним кликом

» Перевести автору 50 рублей «

Новости по изменениям и исправлениям книги публикуются в специальном сообществе:

Обновления книги "Изучаем Mediastreamer2"

Введение

Библиотека *Mediastreamer2* (далее в тексте, для удобства, вместо слова *Mediastreamer2* будем использовать его русскую нотацию: "медиастример"), предназначена для построения систем обработки/передачи звука и видео основанных на технологии VoIP. Благодаря универсальности и интерфейсу для плагинов, медиастример может быть использован не только для разработки и прототипирования указанных систем, но и для работы с другими данными. Библиотека является свободным программным обеспечением с лицензией GPLv3.

Примеры кода, приведенные в книге, демонстрируют применение медиастримера для обработки звуковых данных, но это не мешает применить полученные знания при построении схем обработки видео.

Главы 1 и 3 рассказывают о том как использовать медиастример и писать приложения на языке Си с использованием готовых фильтров медиастримера.

Глава 4 показывает как разработать свой собственный фильтр (плагин). Для этого будет приведен подробный пример.

В главе 5 будет описан механизм перемещения данных внутри медиастримере, что поможет сделать работу ваших фильтры более эффективной.

Отладке фильтров посвящена глава 6, в которой будут показаны подходы к поиску и устранению утечек памяти.

Последняя глава книги 7 рассказывает о способах адаптации топологии схемы к вычислительным возможностям используемой аппаратной платформы.

Глава 1

Обзор возможностей

Библиотека *Mediastreamer2* это медиа-движок, лежащий в основе популярного open-source проекта VoIP-телефона *Linphone*:

https://www.linphone.org/

Он реализует все функции *Linphone* связанные со звуком и видео. Подробный список возможностей движка можно увидеть на этой странице:

http://linphone.org/technical-corner/mediastreamer2

Список платформ, на которых может быть использован *Mediastreamer2,* можно определить по перечню платформ на которых работает *Linphone,* т.е. *IOS*, *Android*, *macOS*, *Windows desktop*, *Gnu/Linux*. Если говорить об аппаратных платформах, то медиастример неприхотлив и может работать на персональном компьютере, телефоне, Raspbery Pi 3B и т.д.. Изложение будет основано на примерах работы в ОС *Ubuntu* на персональном компьютере.

В ходе повествования будут задействованы минимальные навыки работы в терминале *Linux* и программирования на языке Си.

Подробный список возможностей движка можно увидеть на этой странице:

https://www.linphone.org/technical-corner/mediastreamer2

Исходный код находится здесь:

GitLab

История его создания не совсем ясна, но судя по исходному коду, он ранее использовал библиотеку *Glib*, что как бы намекает нам, на возможное дальнее родство с *GStreamer*. В сравнении с которым, медиастример выглядит более легковесным. Но используется та же идея - обработка данных свободно соединяемыми фильтрами - программными модулями у которых строго описаны интерфейсы.

Существенное различие с *GStreamer* состоит в том, что вся база фильтров медиастримера тщательно протестирована, в то время как на сайте *GStreamer*, есть специальная таблица, в которой для каждого фильтра проставлена характеристика протестирован ли он.

Первая версия *Linphone* появилась в 2001 году, её написали два француза из Гренобля: Simon Morlat, оригинальный автор *Linphone*, и *Jehan Monnier*, которые до этого проработали немало лет в телекомдивизионе *Hewlett Packard*. Получается, что на настоящий момент, медиастример существует и развивается без малого 20 лет.

В основе медиастримера лежит архитектура называемая "*Data flow*" (поток данных). Пример такой архитектуры изображен на рисунке 1.1.

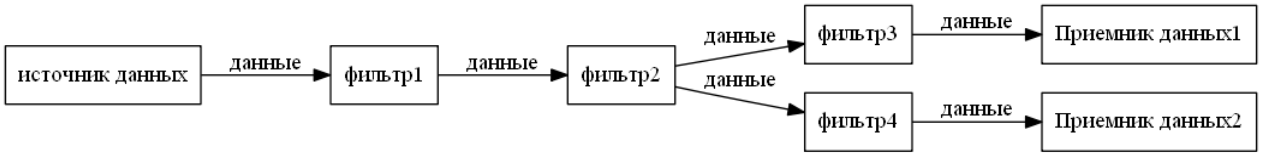


Рис. 1.1 – Архитектура Data flow

В этой архитектуре алгоритм обработки данных задается не программным кодом, а схемой (графом) соединения программных модулей, которые можно выстраивать в любом (почти в любом) порядке. Эти модули называются фильтрами.

Такая подход позволяет реализовать функционал обработки медиа в виде набора фильтров, соединенных в схему обработки и передачи например RTP-трафика VoIP-телефона. При этом разработчику дается возможность абстрагироваться от того, как фильтры работают внутри, тем самым позволяя не тратить время на частности при решении высокоуровневых задач.

Возможность соединять фильтры в произвольные схемы, простая разработка новых фильтров, реализация медиастримера в виде самостоятельной отдельной библиотеки, позволяют использовать его и в других проектах. Причём, проект может быть не обязательно из области VoIP, так как имеется возможность добавлять фильтры сделанные своими руками.

1.1 Перечень готовых фильтров

Поставляемый по умолчанию набор фильтров достаточно богат и, как уже было сказано, может быть расширен фильтрами собственной разработки. В этом разделе мы перечислим готовые фильтры, которые входят в состав медиастримера. После короткого описания фильтра показано название типа, которое используется при создании экземпляров данного фильтра в программном коде.

- Звуковые фильтры
  - Захват и воспроизведение звука:
    - Alsa (Linux): MS_ALSA_WRITE, MS_ALSA_READ
    - Android native sound (libmedia): MS_ANDROID_SOUND_WRITE, MS_ANDROID_SOUND_READ
    - Audio Queue Service (Mac OS X): MS_AQ_WRITE, MS_AQ_READ
    - Audio Unit Service (Mac OS X)
    - Arts (Linux): MS_ARTS_WRITE, MS_ARTS_READ
    - DirectSound (Windows): MS_WINSNDDS_WRITE, MS_WINSNDDS_READ
    - Проигрыватель файлов (raw/wav/pcap файлы) (Linux): MS_FILE_PLAYER
    - Проигрыватель файлов (raw/wav файлы) (Windows): MS_WINSND_READ
    - Запись в файл (wav файлы) (Linux): MS_FILE_REC
    - Запись в файл (wav файлы) (Windows): MS_WINSND_WRITE
    - Mac Audio Unit (Mac OS X)
    - MME (Windows)
    - OSS (Linux): MS_OSS_WRITE, MS_OSS_READ
    - PortAudio (Mac OS X)
    - PulseAudio (Linux): MS_PULSE_WRITE, MS_PULSE_READ
    - Windows Sound (Windows)
  - Фильтры кодирования/декодирования звука:
    - G.711 a-закон: MS_ALAW_DEC, MS_ALAW_ENC
    - G.711 μ-закон: MS_ULAW_DEC, MS_ULAW_ENC
    - G.722: MS_G722_DEC, MS_G722_ENC
    - G.726: MS_G726_32_ENC, MS_G726_24_ENC, MS_G726_16_ENC

* GSM: MS_GSM_DEC, MS_GSM_ENC
* Линейная ИКМ: MS_L16_ENC, MS_L16_DEC
* Speex: MS_SPEEX_ENC, MS_SPEEX_DEC

– Фильтры обработки звука:

* Преобразование канала (моно->стерео, стерео->моно): MS_CHANNEL_ADAPTER
* Конференция: MS_CONF
* Генератор DTMF-сигналов: MS_DTMF_GEN
* Подавление эха (speex): MS_SPEEX_EC
* Эквалайзер: MS_EQUALIZER
* Микшер: MS_MIXER
* Компенсатор потери пакетов (PLC): MS_GENERIC_PLC
* Ресемплер: MS_RESAMPLE
* Детектор тонов: MS_TONE_DETECTOR
* Управление громкостью и измерение уровня сигнала: MS_VOLUME

• Фильтры видео:

– Фильтры захвата и воспроизведения видео:

* Android захват
* Android воспроизведение
* AV Foundation захват (iOS)
* AV Foundation воспроизведение (iOS)
* DirectShow захват (Windows)
* DrawDib воспроизведение (Windows)
* External воспроизведение - Отправка видео на верхний уровень
* GLX воспроизведение (Linux): MS_GLXVIDEO
* Mire - Synthetic moving picture: MS_MIRE
* OpenGL воспроизведение (Mac OS X)
* OpenGL ES2 воспроизведение (Android)
* Quicktime захват (Mac OS X)
* SDL воспроизведение
* Вывод статических изображений: MS_STATIC_IMAGE
* Video For Linux (V4L) захват (Linux): MS_V4L
* Video For Linux 2 (V4L2) захват (Linux): MS_V4L2_CAPTURE
* Video4windows (DirectShow) захват (Windows)

* Video4windows (DirectShow) захват (Windows CE)
* Video For Windows (vfw) захват (Windows)
* XV воспроизведение (Linux)

– Фильтры кодирования/декодирования видео:

* H.263, H.263-1998, MP4V-ES, JPEG, MJPEG, Snow: MS_MJPEG_DEC, MS_H263_ENC, MS_H263_DEC
* H.264 (только декодер): MS_H264_DEC
* Theora: MS_THEORA_ENC, MS_THEORA_DEC
* VP8: MS_VP8_ENC, MS_VP8_DEC

– Фильтры обработки видео:

* JPEG snapshot
* Pixel format converter: MS_PIX_CONV
* Resizer

• Фильтры общего назначения:

– Обмен блоками данных между потоками выполнения: MS_ITC_SOURCE, MS_ITC_SINK

– Перенаправление блоков данных с нескольких входов на один выход: MS_JOIN

– RTP прием/передача: MS_RTP_SEND, MS_RTP_RECV

– Разветвитель, копирующий входные данные на несколько выходов: MS_TEE

– Согласованная нагрузка: MS_VOID_SINK

– Источник тишины: MS_VOID_SOURCE

• Список существующих плагинов:

– Звуковые:

* AMR-NB кодер/декодер
* G.729 кодер/декодер
* iLBC кодер/декодер
* SILK кодер/декодер

– Видео:

* H.264 программный кодер
* H.264 V4L2 кодер/декодер с аппаратным ускорением

Глава 2

Подготовка к работе

В этой главе мы выполним установку медиастримера на компьютер и соберем наше первое приложение на его базе.

Установка *Mediastremer2* на компьютер или виртуальную машину под управлением *Linux Ubuntu* не требует особых навыков. Здесь и далее символом "$" будем обозначать приглашение оболочки *shell* для ввода команд. Т.е. если в листинге вы видите этот символ в начале строки, то значит это строка в которой показаны команды для выполнения в терминале.

Предполагается, что во время действий описанных в этой главе, ваш компьютер имеет доступ к сети Интернет.

2.1 Установка пакета *libmediastremer-dev*

Запускаем терминал и набираем команду:

```
$ sudo apt-get update
```

Будет запрошен пароль на внесение изменений, введите его и менеджер пакетов обновит свои базы. Затем нужно выполнить:

```
$ sudo apt-get install libmediastreamer-dev
```

По этой команде будут автоматически скачаны и установлены необходимые пакеты зависимостей и сама библиотека медиастримера. Общий размер скачанных *deb*-пакетов зависимостей составит примерно 35 МБайт. Подробности об установленном пакете можно узнать командой:

```
$ dpkg -s libmediastreamer-dev
```

Пример ответа:

```
Package: libmediastreamer-dev
Status: install ok installed
Priority: optional
Section: libdevel
Installed-Size: 244
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com
   >
Architecture: amd64
Source: linphone
Version: 3.6.1-2.5
Depends: libmediastreamer-base3 (= 3.6.1-2.5), libortp-dev
Description: Linphone web phone's media library - development files
Linphone is an audio and video internet phone using the SIP protocol
   . It
has a GTK+ and console interface, includes a large variety of audio
   and video
codecs, and provides IM features.
.
This package contains the development libraries for handling media
   operations.
Original-Maintainer: Debian VoIP Team <pkg-voip-maintainers@lists.
   alioth.debian.org>
Homepage: http://www.linphone.org/
```

2.2 Установка инструментов разработки

Устанавливаем компилятор Си и сопутствующие ему инструменты:

```
$ sudo apt-get install gcc
```

Проверяем результат, запросив версию компилятора:

```
$ gcc --version
```

Ответ должен быть примерно таким:

```
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609 Copyright (C)
   2015 Free Software Foundation, Inc. This is free software; see
   the source for copying conditions. There is NO warranty; not even
    for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

2.3 Сборка и запуск пробного приложения

Создаем в *home* папку для наших учебных проектов, назовем её *mstutorial*:

```
$ mkdir ~/mstutorial
```

Для создание пробного приложения воспользуйтесь вашим любимым текстовым редактором и создайте файл Си-программы с именем *mstest.c* со следующим содержанием:

Листинг 2.1: Пробное приложение

```
/* Файл mstest.c Пробное приложение. */

#include "stdio.h"
#include <mediastreamer2/mscommon.h>
int main()
{
  ms_init();
  printf ("Mediastreamer is ready.\n");
}
```

Она выполняет инициализацию медиастримера, печатает приветствие и заканчивает выполнение.

Сохраняем файл и компилируем пробное приложение командой:

```
$ gcc mstest.c -o mstest `pkg-config mediastreamer --libs --cflags
  `
```

Обратите внимание, что строка

```
`pkg-config mediastreamer --libs --cflags`
```

заключена в кавычки, которые находятся на клавиатуре там же где и буква "Ё".

Если файл не содержит ошибок, то после компиляции в каталоге появится файл *mstest*. Запускаем программу:

```
$ ./mstest
```

Результат будет таким:

```
ALSA lib conf.c:4738:(snd_config_expand) Unknown parameters 0
ALSA lib control.c:954:(snd_ctl_open_noupdate) Invalid CTL default:0
ortp-warning-Could not attach mixer to card: Invalid argument
```

```
ALSA lib conf.c:4738:(snd_config_expand) Unknown parameters 0
ALSA lib pcm.c:2266:(snd_pcm_open_noupdate) Unknown PCM default:0
ALSA lib conf.c:4738:(snd_config_expand) Unknown parameters 0
ALSA lib pcm.c:2266:(snd_pcm_open_noupdate) Unknown PCM default:0
ortp-warning-Strange, sound card HDA Intel PCH does not seems to be
   capable of anything, retrying with plughw...
Mediastreamer is ready.
```

В данном листинге мы видим сообщения об ошибках, которые выводит библиотека ALSA, используемая для управления звуковой картой. Сами разработчики медиастримера считают, что на эти сообщения не нужно обращать внимание. Мы в данном случае по неволе согласимся с ними.

Теперь у нас все готово для работы с медиастримером. Мы установили библиотеку медиастримера, инструмент компиляции и с помощью пробного приложения проверили, что инструменты настроены, а медиастример успешно инициализируется.

Глава 3

Примеры использования фильтров

Эта глава будет посвящена нескольким примерам использования фильтров медиастримера. В процессе мы узнаем как организован сигнальный граф, как создавать и уничтожать экземпляры фильтров. Научимся их соединять в схемы обработки данных, обнаружения тональных сигналов, измерения уровня сигнала. Затем мы увидим как передавать сигнал через RTP-поток. С помощью консольной утилиты *TShark* ознакомимся с внутренними полями RTP-пакета. И завершим главу примером дуплексного переговорного устройства.

3.1 Создаем звуковой генератор

В предыдущей главе мы выполнили установку библиотеки медиастримера, инструментов разработки и проверили их функционирование, собрав пробное приложение.

Теперь мы создадим приложение, которое сможет пропиликать на звуковой карте тональный сигнал. Чтобы решить эту задачу нам нужно соединить фильтры в схему звукового генератора, показанную на рисунке 3.1.

Рис. 3.1 – Звуковой генератор

Читаем схему слева направо, именно в этом направлении у нас движется поток данных. Стрелки тоже на это намекают. Прямоугольниками обозначены фильтры, которые обрабатывают блоки данных и выставляют результат на выход. Внутри прямоугольника указана его роль и чуть ниже в круглых скобках тип фильтра. Стрелки, соединяющие прямоугольники, это очереди

данных, по которым блоки данных доставляются от фильтра к фильтру. В общем случае, входов и выходов у фильтра может быть немало.

Начинается все с источника тактов, который задает темп, в котором происходит обсчет данных в фильтрах. По его такту, каждый фильтр обрабатывает все блоки данных которые находятся у него в очереди на входе. И выставляет блоки с результатом на выход — в очередь. Сначала выполняет обсчет самый ближний к источнику тактов фильтр, затем фильтры подключенные к его выходам (выходов может быть много) и так далее. После того, как последний фильтр в цепочке закончит обсчет, выполнение останавливается до того момента, пока не поступит новый такт. Такты, по умолчанию, следуют с интервалом 10 миллисекунд.

Вернемся к нашей схеме. Такты поступают на вход источника тишины, этот фильтр занят тем, что на каждый такт генерирует на своем выходе блок данных содержащих нули. Если рассматривать этот блок как набор звуковых отсчетов, то это ни что иное как тишина. На первый взгляд кажется странным, генерировать блоки данных с тишиной — ведь её нельзя услышать, но они необходимы для работы генератора звукового сигнала. Генератор, использует эти блоки как чистый лист бумаги, записывая в них звуковые отсчеты. В обычном своем состоянии генератор выключен и просто пробрасывает входные блоки на выход. Таким образом, блоки тишины проходят без изменений через всю схему слева направо, попадая в звуковую карту. Которая беззвучно забирает блоки из очереди подключенной к её входу.

Но все меняется, если генератору подать команду на воспроизведение звука, он начинает генерировать звуковые отсчеты и замещает ими отсчеты во входных блоках, выставляя измененные блоки на выход. Звуковая карта, получив их, начинает воспроизводить звук. В листинге 3.1 приведена программа, которая реализует описанную выше схему работы.

Листинг 3.1: Звуковой генератор

```
/* Файл mstest2.c Звуковой генератор. */

#include <mediastreamer2/msfilter.h>
#include <mediastreamer2/msticker.h>
#include <mediastreamer2/dtmfgen.h>
#include <mediastreamer2/mssndcard.h>
int main()
{
    ms_init();

    /* Создаем экземпляры фильтров. */
    MSFilter  *voidsource = ms_filter_new(MS_VOID_SOURCE_ID);
    MSFilter  *dtmfgen = ms_filter_new(MS_DTMF_GEN_ID);
    MSSndCard *card_playback = ms_snd_card_manager_get_default_card(
       ms_snd_card_manager_get());
    MSFilter  *snd_card_write = ms_snd_card_create_writer(
```

```
        card_playback);

    /* Создаем тикер. */
    MSTicker *ticker = ms_ticker_new();

    /* Соединяем фильтры в цепочку. */
    ms_filter_link(voidsource, 0, dtmfgen, 0);
    ms_filter_link(dtmfgen, 0, snd_card_write, 0);

   /* Подключаем источник тактов. */
   ms_ticker_attach(ticker, voidsource);

   /* Включаем звуковой генератор. */
   char key='1';
   ms_filter_call_method(dtmfgen, MS_DTMF_GEN_PLAY, (void*)&key);

   /* Даем, время, чтобы все блоки данных были получены звуковой
      картой.*/
   ms_sleep(2);
}
```

После инициализации медиастримера, выполняется создание трех фильтров: *voidsource*, *dtmfgen*, *snd_card_write*. Создается источник тактового сигнала.

Затем устанавливаются соединения фильтров в соответствии с нашей схемой, причем источник тактов подключается последним, когда схема уже собрана, так как после этого сразу начнется работа схемы. Если подключить источник тактов к незаконченной схеме, то может случится так, что медиастример аварийно завершится, если обнаружит в цепочке хотя бы один фильтр у которого все входы или все выходы "висят в воздухе" (не подключены).

Соединение фильтров выполняется с помощью функции *ms_filter_link()*:

```
int ms_filter_link (MSFilter *f1, int pin1, MSFilter *f2, int pin2)
```

где

f1 указатель на фильтр-источник;

pin1 номер выхода фильтра-источника (обратите внимание, что входы и выходы нумеруются начиная с нуля);

f2 указатель на фильтр-приемник;

pin2 номер входа фильтра-приемника.

Все фильтры соединены и последним подключен источник тактов (далее просто будем называть его тикер). После чего наша звуковая схема запускается в работу, но в динамике компьютера пока ничего не слышно — звуковой генератор выключен и просто пробрасывает через себя входные блоки данных

с тишиной. Чтобы запустить генерацию тонального сигнала, нужно выполнить метод фильтра генератора.

Мы будем включать генерацию двутонального (DTMF) сигнала, соответствующего нажатию на телефоне кнопки "1". Для этого мы с помощью функции
ms_filter_call_method() вызываем метод MS_DTMF_GEN_PLAY, передавая ему в качестве аргумента указатель на символ (переменная *key* в листинге), которому должен соответствовать воспроизводимый сигнал.

Остается скомпилировать программу:

```
$ gcc mstest2.c -o mstest2 `pkg-config mediastreamer --libs --
  cflags`
```

И запустить:

```
$ ./mstest2
```

После запуска программы, вы услышите в динамике компьютера короткий звуковой сигнал состоящий из двух тонов.

Мы написали и запустили в работу нашу первую звуковую схему. Увидели как создавать экземпляры фильтров, как соединять и как вызывать их методы. Порадовавшись первому успеху, нам все же нужно обратить внимание на то, что наша программа перед завершением не освобождает выделенную ей память. В следующем разделе мы уделим внимание корректному завершению наших программ.

3.2 Улучшаем пример тонального генератора

В предыдущем разделе мы написали приложение тонального генератора и с его помощью извлекли звук из динамика компьютера. Теперь мы обратим внимание на то, что наша программа, заканчивая работу, не возвращает память обратно в кучу. Пришло время внести ясность в этом вопрос.

После того, как схема стала нам не нужна, освобождение памяти должно начинаться с остановки конвейера данных. Для этого нужно отключить от схемы источник тактов - тикер. Помогает в этом функция *ms_ticker_detach()*. В нашем случае мы должны отключить тикер от входа фильтра *voidsource*:

```
ms_ticker_detach(ticker, voidsource)
```

Кстати, после остановки конвейера мы можем изменить его схему и снова пустить в работу, опять подключив тикер.

Теперь мы можем его удалить, воспользовавшись функцией *ms_ticker_destroy()*:

```
ms_ticker_destroy(ticker)
```

Конвейер остановлен и мы можем приступить к его разборке на части, разъединяя фильтры. Для этого используется функция *ms_filter_unlink()*:

```
ms_filter_unlink(voidsource, 0, dtmfgen, 0);
ms_filter_unlink(dtmfgen, 0, snd_card_write, 0);
```

назначение аргументов то же самое, что и у функции *ms_filter_link()*.

Удаляем, теперь уже разобщенные, фильтры с помощью *ms_filter_destroy()*:

```
ms_filter_destroy(voidsource);
ms_filter_destroy(dtmfgen);
ms_filter_destroy(snd_card_write);
```

Добавив эти строки в наш пример мы получим корректное, с точки зрения управления памятью, завершение программы.

Как мы видим, правильное завершение программы потребовало от нас добавить примерно столько же строк кода, что было использовано в её начале при сборке схемы, причём в среднем на каждый фильтр пришлось четыре строки кода. Получается, что размер кода программы у нас будет нарастать пропорционально количеству использованных в проекте фильтров. Если говорить о тысяче фильтров в схеме, то к вашему коду добавится четыре тысячи строк рутинных операций по созданию и их уничтожению.

Теперь вы знаете как корректно завершать программу использующую медиастример. В следующих примерах, для компактности изложения, я буду "забывать" это делать. Но вы-то не забудете?

Разработчики медиастримера не предусмотрели программных средств по облегчению манипуляций с фильтрами при сборке/разборке схем. Тем не менее там есть помощник *MSConnectionHelper*, который позволяет быстро вставлять/вынимать фильтр из схемы. Необходимость в нем вы почувствуете, когда количество фильтров в ваших проектах превысит пару десятков.

3.3 Создаем измеритель уровня сигнала

В этом разделе мы соберем схему измерителя уровня сигнала и научимся читать результат измерения из фильтра. Оценим точность измерения. В наборе фильтров, предоставляемых медиастримером есть фильтр, MS_VOLUME, который способен измерять среднеквадратический уровень проходящего через него сигнала, ослаблять сигнал и выполнять массу полезных и неожиданных функций. Но сейчас мы будем использовать его как измеритель.

В качестве источника сигнала будем использовать тональный генератор, сигнал с которого направим на фильтр MS_VOLUME, к выходу которого подключена звуковая карта.

В данном примере фильтр генератора мы будем использовать в несколько другом режиме — он будет генерировать для нас однотональный сигнал, т.е. сигнал содержащий только одно синусоидальное колебание.

Помимо частоты и амплитуды нам понадобится задать длительность сигнала, она должна быть достаточной для того, чтобы через фильтр MS_VOLUME прошло необходимое для измерение количество отсчетов. Для передачи генератору настроек используется структура *MSDtmfGenCustomTone*:

Листинг 3.2: Структура MSDtmfGenCustomTone

```
struct _MSDtmfGenCustomTone{
    char tone_name[8];    /* Текстовое название сигнала из 8 букв.*/
    int duration;         /* Длительность сигнала в миллисекундах.*/
    int frequencies[2];   /* Пара частот из которых должен состоять
       выходной сигнал. */
    float amplitude;      /* Амплитуда тонов, 1.0 соответствует
       уровню 0 дБ от милливатта на нагрузке 600 Ом.*/
    int interval;         /* Пауза в миллисекундах перед началом
       повторного проигрывания сигнала.*/
    int repeat_count;     /* Количество повторов.*/
};
typedef struct _MSDtmfGenCustomTone MSDtmfGenCustomTone;
```

Чтобы запустить генератор в работу, будем использовать его метод MS_DTMF_GEN_PLAY_CUSTOM.

Структурная схема обработки сигнала показана на рисунке 3.2.

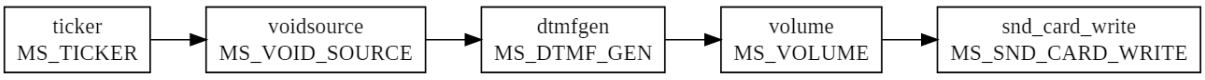


Рис. 3.2 – Измеритель уровня сигнала

Код программы, реализующий эту схему показан в листинге 3.3.

Листинг 3.3: Измеритель уровня сигнала

```
/* Файл mstest3.c Измеритель уровня сигнала. */

#include <mediastreamer2/msfilter.h>
#include <mediastreamer2/msticker.h>
#include <mediastreamer2/dtmfgen.h>
#include <mediastreamer2/mssndcard.h>
#include <mediastreamer2/msvolume.h>
```

```
int main()
{
    ms_init();
    /* Создаем экземпляры фильтров. */
    MSFilter  *voidsource=ms_filter_new(MS_VOID_SOURCE_ID);
    MSFilter  *dtmfgen=ms_filter_new(MS_DTMF_GEN_ID);
    MSFilter  *volume=ms_filter_new(MS_VOLUME_ID);
    MSSndCard *card_playback=ms_snd_card_manager_get_default_card(
       ms_snd_card_manager_get());
    MSFilter  *snd_card_write=ms_snd_card_create_writer(
       card_playback);

    /* Создаем тикер. */
    MSTicker *ticker=ms_ticker_new();

    /* Соединяем фильтры в цепочку. */
    ms_filter_link(voidsource, 0, dtmfgen, 0);
    ms_filter_link(dtmfgen, 0, volume, 0);
    ms_filter_link(volume, 0, snd_card_write, 0);

    /* Подключаем источник тактов. */
    ms_ticker_attach(ticker,voidsource);

    MSDtmfGenCustomTone dtmf_cfg;

   /* Устанавливаем имя нашего сигнала, помня о том, что в массиве
      мы должны
    оставить место для нуля, который обозначает конец строки. */
    strncpy(dtmf_cfg.tone_name, "busy", sizeof(dtmf_cfg.tone_name));
    dtmf_cfg.duration=1000;
    dtmf_cfg.frequencies[0]=440; /* Будем генерировать один тон,
       частоту второго тона установим в 0.*/
    dtmf_cfg.frequencies[1]=0;
    dtmf_cfg.amplitude=1.0; /* Такой амплитуде синуса должен
       соответствовать результат измерения 0.707.*/
    dtmf_cfg.interval=0.;
    dtmf_cfg.repeat_count=0.;

   /* Включаем звуковой генератор. */
   ms_filter_call_method(dtmfgen, MS_DTMF_GEN_PLAY_CUSTOM, (void*)&
      dtmf_cfg);

   /* Даем, время половину секунды, чтобы измеритель накопил данные.
       */
   ms_usleep(500000);

   /* Читаем результат измерения. */
  float level=0;
   ms_filter_call_method(volume, MS_VOLUME_GET_LINEAR,&level);
   printf("Амплитуде␣синуса␣%f␣вольт␣␣соответствует␣
      среднеквадратическое␣значение␣%f␣вольт.\n", dtmf_cfg.amplitude
```

```
        , level);
}
```

Компилируем наш пример, также как мы делали до этого, только используя название файла *mstest3.c* Запускаем на выполнение и получим в консоли результат:

```
Амплитуде синуса 1.000000 вольт соответствует среднеквадратическое
   значение 0.707733 вольт.
```

Как видите, результат измерения совпал до третьего знака после запятой с теоретическим значением равным квадратному корню из двойки поделенному пополам:

```
sqr(2)/2=0,7071067811865475
```

Относительное отклонение результата от истинного значения составило 0.1%. Мы сделали оценку погрешности измерения при максимальном уровне сигнала. Соответственно, при снижении уровня погрешность должна возрастать. Предлагаю вам самостоятельно оценить её для малых уровней сигнала.

3.4 Обнаружитель тонального сигнала

В старые времена, когда не в каждой семье был телевизор, и у половины из них каналы переключались с помощью пассатижей, в обзорах иностранной технической прессы появилась интригующая новость, что один и производителей телевизоров снабдил свои аппараты пультом дистанционного беспроводного управления. Из подробностей было известно, что пульт работает без батареек благодаря использованию необычного подхода — пульт был механический и представлял из себя гибрид музыкального инструмента — металлофона и револьвера. В барабане револьвера находились металлические цилиндры, разные по длине, и когда боёк ударял по одному из них, цилиндр начинал звенеть на своей собственной частоте. Предположительно на ультразвуке. Электроника в телевизоре слышала этот сигнал и определив его частоту выполняла соответствующее действие — переключить канал, изменить громкость, выключить телевизор.

Сегодня мы попробуем сделать реконструкцию этой системы передачи команд, воспользовавшись нашими знаниями медиастримера.

Для имитации пульта, воспользуемся текстом нашего примера тонального генератора. Мы добавим в него управление частотой генератора от нажатий с клавиатуры и приемник с декодером, который будет выводить в консоль

принятые команды. После изменения, генератор должен выдавать тональные сигналы, 6 частот, которыми мы будем кодировать команды увеличения/уменьшения громкости, смены канала, включения/выключения телевизора. Для настройки детектора используется структура MSToneDetectorDef:

Листинг 3.4: Структура MSToneDetectorDef

```
struct _MSToneDetectorDef{
  char tone_name[8];
  int frequency; /* Частота ожидаемого тона. */
  int min_duration; /* Минимальная длительность тона в миллисекундах
     . */
  float min_amplitude; /* Минимальная амплитуда тона, 1.0
     соответствует уровню 0 дБм */
};

typedef struct _MSToneDetectorDef MSToneDetectorDef;
```

Детектору можно передать 10 таких структур, тем самым один детектор можно настроить на обнаружение десяти двутональных сигналов. Но мы с вами будем использовать всего шесть однотональных сигналов. Для загрузки настроек в детектор используется метод MS_TONE_DETECTOR_ADD_SCAN.

Чтобы детектор мог известить нас о том, что на его вход поступил сигнал с искомыми частотными составляющими, мы должны предоставить ему функцию обратного вызова, которую он будет запускать по такому случаю. Настройка обратного вызова делается с помощью функции *ms_filter_set_notify_callback()*. В качестве аргументов она получает указатель на фильтр, указатель на функцию обратного вызова, указатель на данные, которые бы мы хотели передать функции обратного вызова (данные пользователя).

При срабатывании детектора, функция обратного вызова получит данные пользователя, указатель на фильтр детектора, идентификатор события, и структуру MSToneDetectorEvent, описывающую событие:

Листинг 3.5: Структура MSToneDetectorEvent

```
struct _MSToneDetectorEvent{
      char tone_name[8];        /* Имя тона, которое мы ему назначили
          при настройке детектора. */
      uint64_t tone_start_time;    /* Время в миллисекундах, когда
         тон был обнаружен. */
};

typedef struct _MSToneDetectorEvent MSToneDetectorEvent;
```

Схема обработки сигнала изображена на рисунке 3.3, при этом звуковая карта не играет ключевой роли, она оставлена только для того чтобы можно было услышать, что на входе схемы есть сигнал.

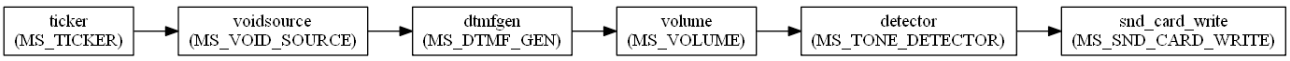


Рис. 3.3 – Обнаружитель тонального сигнала

Ну а теперь сам код программы с комментариями.

Листинг 3.6: Имитатор пульта управления и приемника

```
/* Файл mstest4.c Имитатор пульта управления и приемника. */

#include <mediastreamer2/msfilter.h>
#include <mediastreamer2/msticker.h>
#include <mediastreamer2/dtmfgen.h>
#include <mediastreamer2/mssndcard.h>
#include <mediastreamer2/msvolume.h>
#include <mediastreamer2/mstonedetector.h>

/* Подключаем заголовочный файл с функциями управления событиями
 * медиастримера. */
#include <mediastreamer2/mseventqueue.h>

/* Функция обратного вызова, она будет вызвана фильтром, как только
   он
 * обнаружит совпадение характеристик входного сигнала с заданными.
    */
static void tone_detected_cb(void *data, MSFilter *f, unsigned int
   event_id,
        MSToneDetectorEvent *ev)
{
    printf("_____Принята_команда:_%s\n", ev->
       tone_name);
}

int main()
{
    ms_init();

    /* Создаем экземпляры фильтров. */
    MSFilter  *voidsource = ms_filter_new(MS_VOID_SOURCE_ID);
    MSFilter  *dtmfgen = ms_filter_new(MS_DTMF_GEN_ID);
    MSFilter  *volume = ms_filter_new(MS_VOLUME_ID);
    MSSndCard *card_playback =
        ms_snd_card_manager_get_default_card(ms_snd_card_manager_get
           ());
    MSFilter  *snd_card_write = ms_snd_card_create_writer(
       card_playback);
    MSFilter  *detector = ms_filter_new(MS_TONE_DETECTOR_ID);

    /* Очищаем массив находящийся внутри детектора тонов, он
       описывает
```

```
 * особые приметы разыскиваемых сигналов.*/
ms_filter_call_method(detector, MS_TONE_DETECTOR_CLEAR_SCANS, 0)
    ;

/* Создаем источник тактов - тикер. */
MSTicker *ticker=ms_ticker_new();

/* Соединяем фильтры в цепочку. */
ms_filter_link(voidsource, 0, dtmfgen, 0);
ms_filter_link(dtmfgen, 0, volume, 0);
ms_filter_link(volume, 0, detector, 0);
ms_filter_link(detector, 0, snd_card_write, 0);

/* Подключаем к фильтру функцию обратного вызова. */
ms_filter_set_notify_callback(detector,
        (MSFilterNotifyFunc)tone_detected_cb, NULL);

/* Подключаем источник тактов. */
ms_ticker_attach(ticker,voidsource);

/* Создаем массив, каждый элемент которого описывает
   характеристику
 * одного из тонов, который требуется обнаруживать: Текстовое
    имя
 * данного элемента, частота в герцах, длительность в
    миллисекундах,
 * минимальный уровень относительно 0,775В. */
MSToneDetectorDef  scan[6]=
{
    {"V+",  440, 100, 0.1}, /* Команда "Увеличить громкость". */
    {"V-",  540, 100, 0.1}, /* Команда "Уменьшить громкость". */
    {"C+",  640, 100, 0.1}, /* Команда "Увеличить номер канала".
        */
    {"C-",  740, 100, 0.1}, /* Команда "Уменьшить номер канала".
        */
    {"ON",  840, 100, 0.1}, /* Команда "Включить телевизор". */
    {"OFF", 940, 100, 0.1}  /* Команда "Выключить телевизор". */
};

/* Передаем в детектор тонов приметы сигналов. */
int i;
for (i = 0; i < 6; i++)
{
    ms_filter_call_method(detector, MS_TONE_DETECTOR_ADD_SCAN,
            &scan[i]);
}

/* Настраиваем структуру, управляющую выходным сигналом
   генератора.*/
MSDtmfGenCustomTone dtmf_cfg;
dtmf_cfg.tone_name[0] = 0;
dtmf_cfg.duration = 1000;
```

```
    dtmf_cfg.frequencies[0] = 440;
    /* Будем генерировать один тон, частоту второго тона установим в
        0.*/
    dtmf_cfg.frequencies[1] = 0;
    dtmf_cfg.amplitude = 1.0;
    dtmf_cfg.interval = 0.;
    dtmf_cfg.repeat_count = 0.;

    /* Организуем цикл сканирования нажатых клавиш. Ввод нуля
       завершает
     * цикл и работу программы. */
    char key='9';
    printf("Нажмите клавишу команды, затем ввод.\n"
        "Для завершения программы введите 0.\n");
    while(key != '0')
    {
        key = getchar();
        if ((key >= 49) && (key <= 54))
        {
                printf("Отправлена команда: %c\n", key);
            /* Устанавливаем частоту генератора в соответствии с
             * кодом нажатой клавиши.*/
            dtmf_cfg.frequencies[0] = 440 + 100*(key-49);

            /* Включаем звуковой генератор с обновленной частотой.
               */
            ms_filter_call_method(dtmfgen, MS_DTMF_GEN_PLAY_CUSTOM,
                    (void*)&dtmf_cfg);
        }
        ms_usleep(20000);
    }
}
```

Компилируем и запускаем программу. Если все работает правильно, то после запуска мы должны получить примерно такое поведение программы:

```
$ ./mstest4
ALSA lib conf.c:4738:(snd_config_expand) Unknown parameters 0
ALSA lib control.c:954:(snd_ctl_open_noupdate) Invalid CTL default:0
ortp-warning-Could not attach mixer to card: Invalid argument
ALSA lib conf.c:4738:(snd_config_expand) Unknown parameters 0
ALSA lib pcm.c:2266:(snd_pcm_open_noupdate) Unknown PCM default:0
ALSA lib conf.c:4738:(snd_config_expand) Unknown parameters 0
ALSA lib pcm.c:2266:(snd_pcm_open_noupdate) Unknown PCM default:0
ortp-warning-Strange, sound card Intel 82801AA-ICH does not seems to
    be capable of anything, retrying with plughw...
Нажмите клавишу команды, затем ввод.
Для завершения программы введите 0.
ortp-warning-alsa_set_params: periodsize:256 Using 256
ortp-warning-alsa_set_params: period:8 Using 8
```

Нажимаем любые клавиши от "1" до "6", подтверждая клавишей "*Enter*", должен получаться примерно такой листинг:

```
2
Отправлена команда: 2
                        Принята команда: V–
1
Отправлена команда: 1
                        Принята команда: V+
3
Отправлена команда: 3
                        Принята команда: C+
4
Отправлена команда: 4
                        Принята команда: C–
0
$
```

Мы видим, что тоны команд успешно отправляются и детектор их обнаруживает.

3.5 Передача звукового сигнала через RTP-поток

В этом разделе мы научимся использовать протокол RTP (*RFC 3550 — RTP: A Transport Protocol for Real-Time Applications*) для приема/передачи звукового сигнала по *Ethernet*-сети.

Протокол RTP (*Real Time Protocol)* в переводе означает протокол реального времени, он используется для передачи звука, видео, данных, всего того, что требует передачи в режиме реального времени. В качестве примера возьмем звуковой сигнал. Гибкость протокола такова, что позволяет передавать звуковой сигнал с наперед заданным качеством.

Передача выполняется с помощью UDP-пакетов, что означает что при передаче вполне допускается потеря пакетов. В каждый пакет вкладывается специальный RTP-заголовок и блок данных передаваемого сигнала. В заголовке содержится случайно выбираемый идентификатор источника сигнала, информация о типе передаваемого сигнала, уникальный порядковый номер пакета, для того чтобы пакеты при декодировании могли быть выстроены в правильном порядке, независимо от того, в какой очередности их доставила сеть. Заголовок может содержать дополнительную информацию, так называемое расширение, которое позволяет адаптировать пакет к применению в конкретной прикладной задаче.

Блок данных содержит полезную нагрузку пакета. Внутренняя организация содержимого зависит от типа нагрузки, это могут быть отсчеты монофо-

нического сигнала, стереосигнал, строка видео изображения и т.д.

Тип нагрузки обозначается семибитным числом. Рекомендация RFC3551 (*RTP Profile for Audio and Video Conferenceswith Minimal Control*) устанавливает несколько типов нагрузки в соответствующей таблице приведены описание типов нагрузки и значение кодов, которыми они обозначаются. Часть кодов не имеют жёсткой привязки к какому-либо типу нагрузки - они могут использоваться для обозначения произвольной нагрузки.

Размер блока данных ограничен сверху максимальным размером пакета, который может быть передан в данной сети без сегментирования (параметр MTU). В общем случае это не более 1500 байт. Таким образом, чтобы увеличить количество передаваемых в секунду данных можно до определенного момента увеличивать размер пакета, а затем уже потребуется увеличивать частоту отправки пакетов. В медиастримера это настраиваемый параметр. По умолчанию он равен 50 Гц, т.е. 50 пакетов в секунду. Последовательность передаваемых RTP-пакетов будем называть RTP-потоком.

Чтобы начать передачу данных между источником и приемником, достаточно, чтобы передатчик знал IP-адрес приёмника и номер порта, который тот использует для приема. Т.е. без всяких предварительный процедур источник начинает передавать данные, а приёмник в свою очередь готов немедленно их принять и обработать. По стандарту, номер порта используемый для передачи или приема RTP-потока должен быть четным.

В ситуациях, когда нельзя наперед знать адрес приёмника, используются серверы, на которых приемники оставляют свой адрес, а передатчик может его запросить, сославшись на некое уникальное имя приемника.

В случаях, когда качество канала связи или возможности приёмника неизвестны организуется канал обратной связи, по которому приёмник может информировать передатчик о своих возможностях, о количестве пакетов, которых он не досчитался и т.д. В таком канале используется RTCP-протокол (*RTP Control Protocol*). Формат пакетов передаваемых в этом канале определяется в RFC 3605. По этому каналу передаётся сравнительно немного данных 200..300 байт в секунду, поэтому в целом, его наличие необременительно. Номер порта, на который отправляются RTCP-пакеты должен быть нечетным и на единицу больше номера порта, с которого приходит RTP-поток. В нашем примере мы не будем использовать этот канал, так как возможности приёмника и канала заведомо превышают наши, пока скромные, потребности.

В этом примере схема передачи данных, в отличие от схемы предыдущего, будет разделена на две части: передающий тракт и приемный тракт. Для каждой части мы сделаем свой источник тактов, как показано на рисунке 3.4.

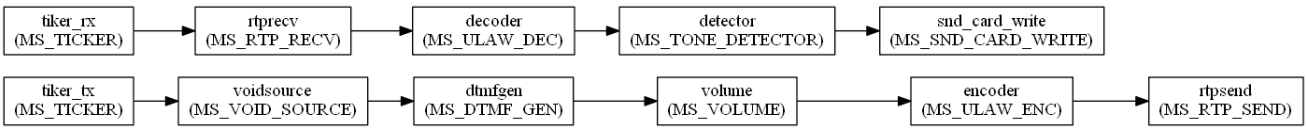


Рис. 3.4 – Использование RTP-потока

Односторонняя связь между ними будет осуществляться с помощью RTP-протокола. В данном примере нам не потребуется внешняя сеть, так как и передатчик и приёмник будут располагаться на одном компьютере — пакеты будут ходить у него внутри.

Для установления RTP-потока в медиастримере используются два фильтра: MS_RTP_SEND и MS_RTP_RECV. Первый выполняет передачу, второй прием RTP-потока. Чтобы эти фильтры начали работать, им нужно передать указатель на объект RTP-сессии, которая может выполнять как преобразование потока блоков данных в поток RTP-пакетов так и выполнять обратное действие. Поскольку внутренний формат данных медиастримера не совпадает с форматом данных RTP-пакета, то перед передачей данных в MS_RTP_SEND нужно использовать конвертер (*encoder*), который преобразует 16-битные отсчеты звукового сигнала в восьмибитные, кодированные по μ-закону (описан в стандарте G.711). На приемной стороне обратное действие выполняет фильтр *decoder*.

В листинге 3.7 приведен текст программы, реализующей схему показанную на рисунке 3.4.

Листинг 3.7: Имитатор пульта управления и приемника с RTP

```
/* Файл mstest6.c Имитатор пульта управления и приемника с RTP. */

#include <mediastreamer2/msfilter.h>
#include <mediastreamer2/msticker.h>
#include <mediastreamer2/dtmfgen.h>
#include <mediastreamer2/mssndcard.h>
#include <mediastreamer2/msvolume.h>
#include <mediastreamer2/mstonedetector.h>
#include <mediastreamer2/msrtp.h>
#include <ortp/rtpsession.h>
#include <ortp/payloadtype.h>

/* Подключаем заголовочный файл с функциями управления событиями
* медиастримера.*/
include <mediastreamer2/mseventqueue.h>

#define PCMU 0

/* Функция обратного вызова, она будет вызвана фильтром, как только
   он
```

```
обнаружит совпадение характеристик входного сигнала с заданными. */
static void tone_detected_cb(void *data, MSFilter *f, unsigned int
   event_id,
MSToneDetectorEvent *ev)
{
  printf("Принята команда: %s\n", ev->tone_name);
}

/*-------------------------------------------------------------------*/
/* Функция регистрации типов полезных нагрузок. */
void register_payloads(void)
{
  /*Регистрируем типы нагрузок в таблице профилей. Позднее, по
     индексу
  взятому из заголовка RTP-пакета из этой таблицы будут извлекаться
  параметры нагрузки, необходимые для декодирования данных пакета.
     */
  rtp_profile_set_payload (&av_profile, PCMU, &payload_type_pcm8000)
     ;
}

/*-------------------------------------------------------------------*/
/* Эта функция создана из функции create_duplex_rtpsession() в
   audiostream.c
медиастримера2. */
static RtpSession *
create_rtpsession (int loc_rtp_port, int loc_rtcp_port,
bool_t ipv6, RtpSessionMode mode)
{
  RtpSession *rtpr;
  rtpr = rtp_session_new ((int) mode);
  rtp_session_set_scheduling_mode (rtpr, 0);
  rtp_session_set_blocking_mode (rtpr, 0);
  rtp_session_enable_adaptive_jitter_compensation (rtpr, TRUE);
  rtp_session_set_symmetric_rtp (rtpr, TRUE);
 rtp_session_set_local_addr (rtpr, ipv6 ? "::" : "0.0.0.0",
    loc_rtp_port,
loc_rtcp_port);
  rtp_session_signal_connect (rtpr, "timestamp_jump",
  (RtpCallback) rtp_session_resync, 0);
  rtp_session_signal_connect (rtpr, "ssrc_changed",
  (RtpCallback) rtp_session_resync, 0);
  rtp_session_set_ssrc_changed_threshold (rtpr, 0);
  rtp_session_set_send_payload_type(rtpr, PCMU);

  /* По умолчанию выключаем RTCP-сессию, так как наш пульт не будет
       использовать её. */
  rtp_session_enable_rtcp (rtpr, FALSE);
  return rtpr;
}

/*-------------------------------------------------------------------*/
```

```
int main()
{
  ms_init();

  /* Создаем экземпляры фильтров. */
  MSFilter *voidsource = ms_filter_new(MS_VOID_SOURCE_ID);
  MSFilter *dtmfgen = ms_filter_new(MS_DTMF_GEN_ID);
  MSFilter *volume = ms_filter_new(MS_VOLUME_ID);
  MSSndCard *card_playback =
  ms_snd_card_manager_get_default_card(ms_snd_card_manager_get());
  MSFilter *snd_card_write = ms_snd_card_create_writer(card_playback
     );
  MSFilter *detector = ms_filter_new(MS_TONE_DETECTOR_ID);

  /* Очищаем массив находящийся внутри детектора тонов, он описывает
   * особые приметы разыскиваемых сигналов.*/
  ms_filter_call_method(detector, MS_TONE_DETECTOR_CLEAR_SCANS, 0);

  /* Подключаем к фильтру функцию обратного вызова. */
  ms_filter_set_notify_callback(detector,
(MSFilterNotifyFunc)tone_detected_cb, NULL);

  /* Создаем массив, каждый элемент которого описывает
     характеристику
   * одного из тонов, который требуется обнаруживать: Текстовое имя
   * данного элемента, частота в герцах, длительность в
      миллисекундах,
   * минимальный уровень относительно 0,775В. */
  MSToneDetectorDef scan[6]=
  {
    {"V+",440, 100, 0.1}, /* Команда "Увеличить громкость". */
    {"V-",540, 100, 0.1}, /* Команда "Уменьшить громкость". */
    {"C+",640, 100, 0.1}, /* Команда "Увеличить номер канала". */
    {"C-",740, 100, 0.1}, /* Команда "Уменьшить номер канала". */
    {"ON",840, 100, 0.1}, /* Команда "Включить телевизор". */
    {"OFF", 940, 100, 0.1}/* Команда "Выключить телевизор". */
  };

  /* Передаем "приметы" сигналов детектор тонов. */
  int i;
  for (i = 0; i < 6; i++)
  {
    ms_filter_call_method(detector, MS_TONE_DETECTOR_ADD_SCAN,
&scan[i]);
  }

  /* Создаем фильтры кодера и декодера */
  MSFilter *encoder = ms_filter_create_encoder("PCMU");
  MSFilter *decoder=ms_filter_create_decoder("PCMU");
  /* Регистрируем типы нагрузки. */
  register_payloads();
```

```
/* Создаем RTP-сессию передатчика. */
RtpSession *tx_rtp_session = create_rtpsession (8010, 8011, FALSE,
        RTP_SESSION_SENDONLY);
rtp_session_set_remote_addr_and_port(tx_rtp_session,"127.0.0.1",
    7010, 7011);
rtp_session_set_send_payload_type(tx_rtp_session, PCMU);
MSFilter *rtpsend = ms_filter_new(MS_RTP_SEND_ID);
ms_filter_call_method(rtpsend, MS_RTP_SEND_SET_SESSION,
    tx_rtp_session);

/* Создаем RTP-сессию приемника. */
MSFilter *rtprecv = ms_filter_new(MS_RTP_RECV_ID);
RtpSession *rx_rtp_session = create_rtpsession (7010, 7011, FALSE,
     RTP_SESSION_RECVONLY);
ms_filter_call_method(rtprecv, MS_RTP_RECV_SET_SESSION,
    rx_rtp_session);

/* Создаем источники тактов - тикеры. */
MSTicker *ticker_tx = ms_ticker_new();
MSTicker *ticker_rx = ms_ticker_new();

/* Соединяем фильтры передатчика. */
ms_filter_link(voidsource, 0, dtmfgen, 0);
ms_filter_link(dtmfgen, 0, volume, 0);
ms_filter_link(volume, 0, encoder, 0);
ms_filter_link(encoder, 0, rtpsend, 0);

/* Соединяем фильтры приёмника. */
ms_filter_link(rtprecv, 0, decoder, 0);
ms_filter_link(decoder, 0, detector, 0);
ms_filter_link(detector, 0, snd_card_write, 0);

/* Подключаем источник тактов. */
ms_ticker_attach(ticker_tx, voidsource);
ms_ticker_attach(ticker_rx, rtprecv);

/* Настраиваем структуру, управляющую выходным сигналом генератора
    . */
MSDtmfGenCustomTone dtmf_cfg;
dtmf_cfg.tone_name[0] = 0;
dtmf_cfg.duration = 1000;
dtmf_cfg.frequencies[0] = 440;
/* Будем генерировать один тон, частоту второго тона установим в
    0. */
dtmf_cfg.frequencies[1] = 0;
dtmf_cfg.amplitude = 1.0;
dtmf_cfg.interval = 0.;
dtmf_cfg.repeat_count = 0.;

/* Организуем цикл сканирования нажатых клавиш. Ввод нуля
    завершает
 * цикл и работу программы. */
```

```
  char key='9';
  printf("Нажмите клавишу команды, затем ввод.\n"
  "Для завершения программы введите 0.\n");
  while(key != '0')
  {
    key = getchar();
    if ((key >= 49) && (key <= 54))
    {
      printf("Отправлена команда: %c\n", key);

      /* Устанавливаем частоту генератора в соответствии с
       * кодом нажатой клавиши. */
      dtmf_cfg.frequencies[0] = 440 + 100*(key-49);

      /* Включаем звуковой генератор с обновленной частотой. */
      ms_filter_call_method(dtmfgen, MS_DTMF_GEN_PLAY_CUSTOM,
      (void*)&dtmf_cfg);
    }

    /* Укладываем тред в спячку на 20мс, чтобы другие треды
     * приложения получили время на работу. */
    ms_usleep(20000);
  }
}
```

Компилируем, запускаем. Внешне, работа программы будет выглядеть как в прошлом примере 3.6, но при этом данные будут передаваться уже через RTP-поток.

3.6 Используем TShark для анализа RTP-пакетов

В этом разделе мы продолжим изучать передачу звукового сигнала с помощью RTP-протокола. Сначала разделим наше тестовое приложение 3.7на два отдельных приложения передатчика и приемника, затем научимся исследовать RTP-поток с помощью анализатора сетевого трафика.

Итак, чтобы нам более ясно было видно какие элементы программы отвечают за передачу RTP, а какие за прием, мы разделяем наш файл *mstest6.c* на две самостоятельные программы передатчика и приемника, общие функции, которые используют и тот и другой мы вынесем в третий файл, который назовем *mstest_common.c* листинг 3.8, он будет подключаться передатчиком и приемником с помощью директивы *include*:

Листинг 3.8: Общие функции для передатчика и приемника

```
/* Файл mstest_common.c RTP Control Protocol */
```

```
#include <mediastreamer2/msfilter.h>
#include <mediastreamer2/msticker.h>
#include <mediastreamer2/msrtp.h>
#include <ortp/rtpsession.h>
#include <ortp/payloadtype.h>

define PCMU 0

/*-------------------------------------------------------------------*/
/* Функция регистрации типов полезных нагрузок. */
void register_payloads(void)
{
 /* Регистрируем типы нагрузок в таблице профилей. Позднее, по
    индексу     взятому
     из заголовка RTP-пакета из этой таблицы будут извлекаться
        параметры
     нагрузки, необходимые для декодирования данных пакета. */
  rtp_profile_set_payload (&av_profile, PCMU, &payload_type_pcm8000)
     ;
}

/*-------------------------------------------------------------------*/
/* Эта функция создана из функции create_duplex_rtpsession() в
   audiostream.c    медиастримера2. */
 static RtpSession *create_rtpsession (int loc_rtp_port, int
    loc_rtcp_port,  bool_t ipv6, RtpSessionMode mode)
{
  RtpSession *rtpr;  rtpr = rtp_session_new ((int) mode);
  rtp_session_set_scheduling_mode (rtpr, 0);
  rtp_session_set_blocking_mode (rtpr, 0);
  rtp_session_enable_adaptive_jitter_compensation (rtpr, TRUE);
  rtp_session_set_symmetric_rtp (rtpr, TRUE);
  rtp_session_set_local_addr (rtpr, ipv6 ? "::" : "0.0.0.0",
     loc_rtp_port,  loc_rtcp_port);
  rtp_session_signal_connect (rtpr, "timestamp_jump",  (RtpCallback)
      rtp_session_resync, 0);
  rtp_session_signal_connect (rtpr, "ssrc_changed",  (RtpCallback)
     rtp_session_resync, 0);
  rtp_session_set_ssrc_changed_threshold (rtpr, 0);
  rtp_session_set_send_payload_type(rtpr, PCMU);

  /* По умолчанию выключаем RTCP-сессию, так как наш пульт не будет
     использовать
  её. */
 rtp_session_enable_rtcp (rtpr, FALSE);
 return rtpr;
}
```

Теперь файл обособленного передатчика *mstest6.c*, листинг 3.9:

Листинг 3.9: Имитатор пульта управления (передатчик)

```
/* Файл mstest6.c Имитатор пульта управления (передатчик). */

#include <mediastreamer2/dtmfgen.h>
#include <mediastreamer2/msrtp.h>
#include "mstest_common.c"

/*-------------------------------------------------------------------*/
int main()
{
  ms_init();

/* Создаем экземпляры фильтров. */
  MSFilter *voidsource = ms_filter_new(MS_VOID_SOURCE_ID);
  MSFilter *dtmfgen = ms_filter_new(MS_DTMF_GEN_ID);

/* Создаем фильтр кодера. */
  MSFilter *encoder = ms_filter_create_encoder("PCMU");

/* Регистрируем типы нагрузки. */
  register_payloads();

/* Создаем RTP-сессию передатчика. */
  RtpSession *tx_rtp_session = create_rtpsession (8010, 8011, FALSE,
      RTP_SESSION_SENDONLY);
 rtp_session_set_remote_addr_and_port(tx_rtp_session,"127.0.0.1",
    7010, 7011);
 rtp_session_set_send_payload_type(tx_rtp_session, PCMU);
 MSFilter *rtpsend = ms_filter_new(MS_RTP_SEND_ID);
 ms_filter_call_method(rtpsend, MS_RTP_SEND_SET_SESSION,
    tx_rtp_session);

/* Создаем источник тактов - тикер. */
 MSTicker *ticker_tx = ms_ticker_new();

/* Соединяем фильтры передатчика. */
 ms_filter_link(voidsource, 0, dtmfgen, 0);
 ms_filter_link(dtmfgen, 0, encoder, 0);
 ms_filter_link(encoder, 0, rtpsend, 0);

/* Подключаем источник тактов. */
  ms_ticker_attach(ticker_tx, voidsource);

/* Настраиваем структуру, управляющую выходным сигналом генератора.
   */
 MSDtmfGenCustomTone dtmf_cfg;
 dtmf_cfg.tone_name[0] = 0;
 dtmf_cfg.duration = 1000;
 dtmf_cfg.frequencies[0] = 440;

/* Будем генерировать один тон, частоту второго тона установим в 0.
   */
```

```
 dtmf_cfg.frequencies[1] = 0;
 dtmf_cfg.amplitude = 1.0;
 dtmf_cfg.interval = 0.;
 dtmf_cfg.repeat_count = 0.;

/* Организуем цикл сканирования нажатых клавиш. Ввод нуля завершает
* цикл и работу программы. */
 char key='9';
 printf("Нажмите клавишу команды, затем ввод.\n"
"Для завершения программы введите 0.\n");
while(key != '0')
{
 key = getchar();
 if ((key >= 49) && (key <= 54))
   {
      printf("Отправлена команда: %c\n", key);
      /* Устанавливаем частоту генератора в соответствии с
       * кодом нажатой клавиши. */
      dtmf_cfg.frequencies[0] = 440 + 100*(key-49);

      /* Включаем звуковой генератор с обновленной частотой. */
      ms_filter_call_method(dtmfgen, MS_DTMF_GEN_PLAY_CUSTOM,       (
          void*)&dtmf_cfg);
   }
   /* Укладываем тред в спячку на 20мс, чтобы другие треды
   * приложения получили время на работу. */
  ms_usleep(20000);
  }
}
```

И наконец, файл приемника *mstest7.c*, листинг 3.10:

Листинг 3.10: Имитатор приемника

```
/* Файл mstest7.c Имитатор приемника. */
include <mediastreamer2/mssndcard.h>
include <mediastreamer2/mstonedetector.h>
include <mediastreamer2/msrtp.h>

/* Подключаем заголовочный файл с функциями управления событиями
   медиастримера.*/
include <mediastreamer2/mseventqueue.h>
/* Подключаем файл общих функций. */
include "mstest_common.c"

/* Функция обратного вызова, она будет вызвана фильтром, как только
   он  обнаружит совпадение характеристик входного сигнала с
   заданными. */
static void tone_detected_cb(void *data, MSFilter *f, unsigned int
   event_id,MSToneDetectorEvent *ev)
{
 printf("Принята команда: %s\n", ev->tone_name);
```

```
}

/*-------------------------------------------------------------------*/
int main()
{
 ms_init();

/* Создаем экземпляры фильтров. */
 MSSndCard *card_playback = ms_snd_card_manager_get_default_card(
    ms_snd_card_manager_get());
 MSFilter *snd_card_write = ms_snd_card_create_writer(card_playback)
    ;
 MSFilter *detector = ms_filter_new(MS_TONE_DETECTOR_ID);

/* Очищаем массив находящийся внутри детектора тонов, он описывает
* особые приметы разыскиваемых сигналов.*/
  ms_filter_call_method(detector, MS_TONE_DETECTOR_CLEAR_SCANS, 0);

/* Подключаем к фильтру функцию обратного вызова. */
ms_filter_set_notify_callback(detector, (MSFilterNotifyFunc)
   tone_detected_cb, NULL);

/* Создаем массив, каждый элемент которого описывает характеристику
* одного из тонов, который требуется обнаруживать:
Текстовое имя
* данного элемента, частота в герцах, длительность в миллисекундах,
* минимальный уровень относительно 0,775В. */
  MSToneDetectorDef scan[6]=
 {
    {"V+",440, 100, 0.1}, /* Команда "Увеличить громкость". */
    {"V-",540, 100, 0.1}, /* Команда "Уменьшить громкость". */
    {"C+",640, 100, 0.1}, /* Команда "Увеличить номер канала". */
    {"C-",740, 100, 0.1}, /* Команда "Уменьшить номер канала". */
    {"ON",840, 100, 0.1}, /* Команда "Включить телевизор". */
    {"OFF", 940, 100, 0.1}/* Команда "Выключить телевизор". */
  };

/* Передаем "приметы" сигналов детектор тонов. */
  int i;
 for (i = 0; i < 6; i++)
 {
   ms_filter_call_method(detector, MS_TONE_DETECTOR_ADD_SCAN,     &
      scan[i]);
 }

/* Создаем фильтр декодера */
  MSFilter *decoder=ms_filter_create_decoder("PCMU");

/* Регистрируем типы нагрузки. */
  register_payloads();

/* Создаем RTP-сессию приемника. */
```

```
  MSFilter *rtprecv = ms_filter_new(MS_RTP_RECV_ID);
  RtpSession *rx_rtp_session = create_rtpsession (7010, 7011, FALSE,
      RTP_SESSION_RECVONLY);
  ms_filter_call_method(rtprecv, MS_RTP_RECV_SET_SESSION,
     rx_rtp_session);

/* Создаем источник тактов - тикер. */
 MSTicker *ticker_rx = ms_ticker_new();

/* Соединяем фильтры приёмника. */
  ms_filter_link(rtprecv, 0, decoder, 0);
  ms_filter_link(decoder, 0, detector, 0);
  ms_filter_link(detector, 0, snd_card_write, 0);

/* Подключаем источник тактов. */
  ms_ticker_attach(ticker_rx, rtprecv);
  char key='9';
  printf( "Для завершения программы введите 0.\n");
  while(key != '0')
 {
    key = getchar();
   /* Укладываем тред в спячку на 20мс, чтобы другие треды     *
      приложения получили время на работу. */
   ms_usleep(20000);
 }
}
```

Компилируем, передатчик и приемник, затем запускаем каждый в своей консоли. Далее должно работать как прежде — только ввод чисел от 1 до 6 мы должны делать в консоли передатчика, а отклик на них должен появляться в консоли приемника. В динамике должны прослушиваться тоны. Если все так, то между приемником и передатчиком у нас установлена связь — происходит непрерывная передача RTP-пакетов от передатчика к приемнику.

Теперь самое время установить анализатор трафика, для этого мы установим консольную версию отличной программы *Wireshark* — она называется *TShark*. Я выбрал *TShark* для дальнейшего изложения для того, чтобы облегчить описание управления программой. С *Wireshark* мне бы потребовалось море скриншотов, которые могут быстро устареть с выпуском новой версии *Wireshark*.

Если вы умеете пользоваться *Wireshark*, то можете использовать его для изучения наших примеров. Но и в этом случае я вам рекомендую освоить *TShark*, так как вам это поможет автоматизировать тестирование ваших VoIP приложений, а также выполнять захват пакетов удаленно.

Устанавливаем *TShark* командой:

```
$ sudo apt-get install tshark
```

Традиционно, проверяем результат установки запрашивая версию программы:

```
$ tshark - -version
```

Если получен адекватный ответ продолжаем далее.

Поскольку наши пакеты ходят пока только внутри компьютера, мы можем попросить *TShark* показывать только такие пакеты. Для этого нужно выбрать захват пакетов с интерфейса loopback (обратная петля), передав *TShark* опцию *-i lo*:

```
$ sudo tshark -i lo
```

В консоль сразу начнут сыпаться сообщения о пакетах отправляемых нашим передатчиком (непрерывно, независимо от того, нажимали мы кнопку на пульте или нет). Возможно, на вашем компьютере есть программы, которые тоже отправляют пакеты по локальной петле, в этом случае мы получим смесь наших и чужих пакетов. Чтобы быть уверенными, что в списке мы видим только пакеты нашего пульта, мы включим фильтрацию пакетов по номеру порта. Нажатием *Ctrl-C* останавливаем анализатор и добавляем опцию фильтрации по номеру порта, который использует наш пульт как порт назначения для своей передачи (8010): *-f "udp port 8010"*. Теперь наша командная строка будет выглядеть так:

```
$ sudo tshark -i lo -f "udp port 8010"
```

В консоли появиться вывод следующего вида (первые 10 строк):

```
 1 0.000000000    127.0.0.1 -> 127.0.0.1    UDP 214 8010 -> 7010 Len=172
 2 0.020059705    127.0.0.1 -> 127.0.0.1    UDP 214 8010 -> 7010 Len=172
 3 0.040044409    127.0.0.1 -> 127.0.0.1    UDP 214 8010 -> 7010 Len=172
 4 0.060057104    127.0.0.1 -> 127.0.0.1    UDP 214 8010 -> 7010 Len=172
 5 0.080082311    127.0.0.1 -> 127.0.0.1    UDP 214 8010 -> 7010 Len=172
 6 0.100597153    127.0.0.1 -> 127.0.0.1    UDP 214 8010 -> 7010 Len=172
 7 0.120122668    127.0.0.1 -> 127.0.0.1    UDP 214 8010 -> 7010 Len=172
 8 0.140204789    127.0.0.1 -> 127.0.0.1    UDP 214 8010 -> 7010 Len=172
 9 0.160719008    127.0.0.1 -> 127.0.0.1    UDP 214 8010 -> 7010 Len=172
10 0.180673685    127.0.0.1 -> 127.0.0.1    UDP 214 8010 -> 7010 Len=172
```

Пока еще это не пакеты, а пронумерованный список событий, где каждая строка это сообщение об очередном пакете, который был замечен на интерфейсе. Поскольку, мы позаботились о фильтрации пакетов, то мы видим в листинге только сообщения о пакетах нашего передатчика. Далее расшифруем эту таблицу по номерам колонок:

- Номер события;
- Время его возникновения;
- IP-адрес источника пакета и IP-адрес приемника пакета;

• Протокол пакета, показан как UDP, поскольку RTP-пакеты передаются как полезная нагрузка внутри UDP-пакетов;

• Размер пакета в байтах, 214;

• Номер порта источника пакета;

• Номер порта приемника пакета;

• Размер полезной нагрузки пакета, отсюда можно сделать вывод, что наш передатчик формирует RTP-пакеты размером 172 байта, который, как утка в сундуке, находится внутри UDP-пакета размером 214 байт.

Теперь пришло время заглянуть внутрь UDP-пакетов, для этого мы запустим *TShark* с дополненным набором ключей:

```
$ sudo tshark -i lo -f "udp port 8010" -P -V -O rtp -o rtp.
   heuristic_rtp:TRUE -x
```

В результате, вывод программы обогатится — к каждому событию добавится расшифровка внутреннего содержимого пакета, который его породил. Чтобы лучше рассмотреть данные вывода, вы можете либо остановить *TShark* нажатием *Ctrl-C*, либо продублировать его вывод в файл, добавив в команде запуска конвейер к программе *tee* с указанием имени выходного файла, *tee* <имя_файла>. Если мы хотим сохранить вывод в файл *log.txt*, команда примет следующий вид:

```
$ sudo tshark -i lo -f "udp port 8010" -P -V -O rtp -o rtp.
   heuristic_rtp:TRUE -x | tee log.txt
```

Теперь посмотрим на то, что мы получили в файле, вот первый пакет из него:

```
1 0.000000000    127.0.0.1 -> 127.0.0.1    RTP 214 PT=ITU-T G.711 PCMU, SSRC=0
   x6B8B4567, Seq=58366, Time=355368720
Frame 1: 214 bytes on wire (1712 bits), 214 bytes captured (1712 bits) on
   interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00
   (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1User Datagram
   Protocol, Src Port: 8010, Dst Port: 7010
Real-Time Transport Protocol    [Stream setup by HEUR RT (frame 1)]
        [Setup frame: 1]
       [Setup Method: HEUR RT]
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
   0... .... = Marker: False
    Payload type: ITU-T G.711 PCMU (0)
    Sequence number: 58366    [Extended sequence number: 58366]
```

```
    Timestamp: 355368720
    Synchronization Source identifier: 0x6b8b4567 (1804289383)
    Payload: ffffffffffffffffffffffffffffffffffffffffffffffffffffffff...

0000   00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00    ..............E.
0010   00 c8 3c 69 40 00 40 11 ff b9 7f 00 00 01 7f 00    ..<i@.@.........
0020   00 01 1f 4a 1b 62 00 b4 fe c7 80 00 e3 fe 15 2e    ...J.b..........
0030   7f 10 6b 8b 45 67 ff ff ff ff ff ff ff ff ff ff    ..k.Eg..........
0040   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
0050   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
0060   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
0070   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
0080   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
0090   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
00a0   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
00b0   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
00c0   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
00d0   ff ff ff ff ff ff
```

Следующий раздел мы посвятим разбору информации, которая содержится в этом листинге и неизбежно поговорим о внутренней структуре RTP-пакета.

3.7 Структура RTP-пакета

В этом разделе мы раскрасим элементы пакета в разные цвета и поговорим об их назначении.

Взглянем на тот же пакет, но уже с подкрашенными полями и с поясняющими надписями, рисунок 3.5.

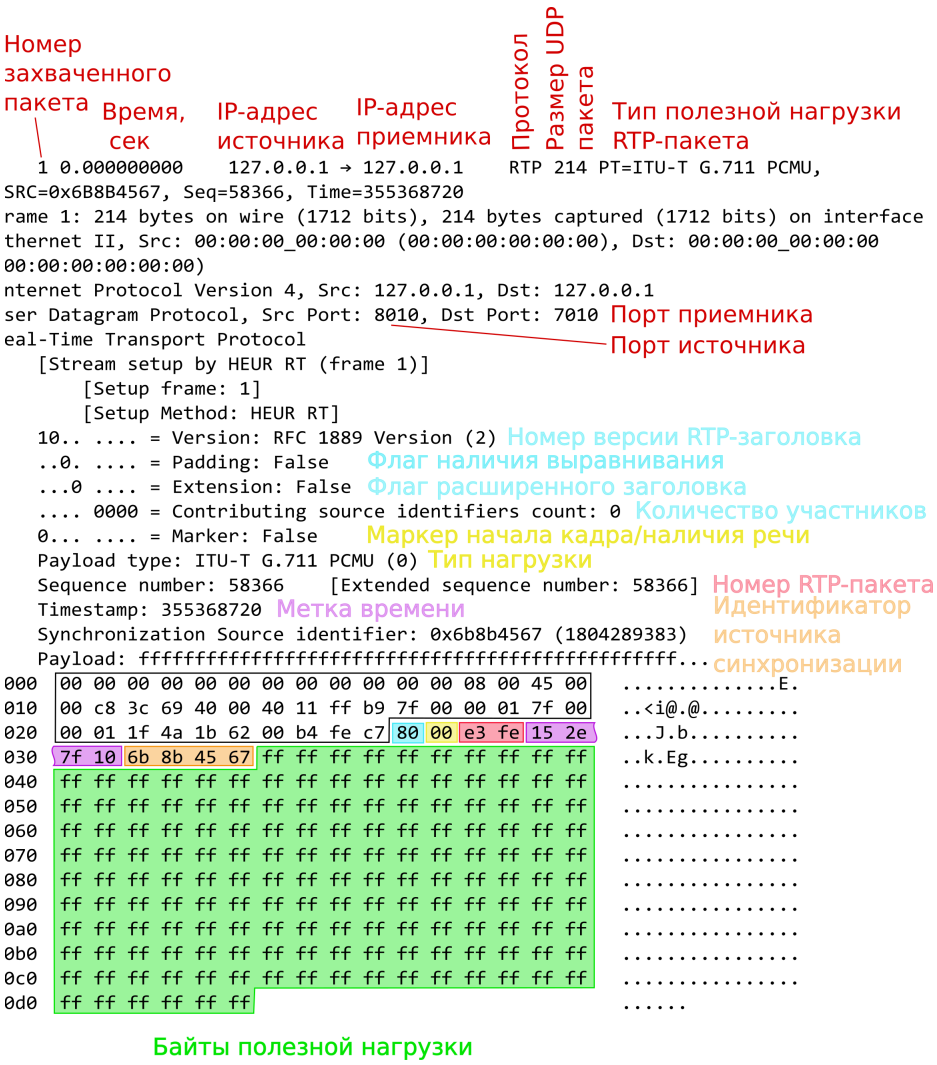


Рис. 3.5 – Поля RTP-пакета

В нижней части листинга подкрашены байты, которые составляют RTP-пакет, а он в свою очередь является полезной нагрузкой UDP-пакета (его заголовок обведен черной линией). Цветными фонами обозначены байты RTP-заголовка, а зеленым цветом выделен блок данных, который содержит полезную нагрузку RTP-пакета. Данные там представлены в шестнадцатиричном формате. В нашем случае это звуковой сигнал сжатый по μ-закону, т.е. один его отсчет имеет размер 1 байт. Поскольку мы использовали семплинг-грейт, установленный по умолчанию (8000 Гц), то при частоте пакетов 50 Гц каждый RTP-пакет должен содержать 160 байт полезной нагрузки. Это мы и увидим, посчитав байты в зеленой области, их должно оказаться 10 строчек.

По стандарту, количество данных в полезной нагрузке должно быть кратно четырем, или иными словами должно содержать целое количество четырехбайтных слов. Если случится так, что ваша полезная нагрузка не будет соответствовать этому правилу, то в конце полезной нагрузки нужно добавить байты с нулевыми значениями и установить бит *Padding* (Дополнение). Этот бит расположен в первом байте RTP-заголовка, он подкрашен бирюзовым

цветом. Обратите внимание, что все байты полезно нагрузки имеют значение 0xFF — так выглядит тишина в формате *μ-law*.

Заголовок RTP-пакета состоит из 12 обязательных байтов, но в двух случаях он может быть длиннее:

• Когда пакет несет звуковой сигнал, полученный смешиванием сигналов от нескольких источников (RTP-потоков), то после первых 12 байт заголовка располагается таблица со списком идентификаторов источников, полезные нагрузки которых были использованы для создания полезной нагрузки этого пакета. При этом в младших четырех битах первого байта заголовка (поле *Contributing source identifiers count*) указывается количество источников. Размер поля составляет 4 бита, соответственно таблица может содержать до 15 идентификаторов источников. Каждый из которых занимает 4 байта. Эта таблица используется при организации конференц-связи;

• Когда заголовок имеет расширение. В этом случае в первом байте заголовка устанавливается бит *X*. В расширенном заголовке, после таблицы участников (если они есть), располагается заголовок расширения размером в одно слово, а вслед за ним слова расширения. Расширение это последовательность слов, которые вы можете использовать для того чтобы передавать дополнительные данные. Стандарт не оговаривает формат этих данных — он может быть любым. Например это могут быть какие-то дополнительные настройки для устройства, которое получает RTP-пакеты. Для некоторых применений, тем не менее разработаны стандарты расширенного заголовка. Так сделано например в стандарте ED-137 для средств связи (*Interoperability Standards for VoIP ATM Components*).

Теперь рассмотрим поля заголовка более детально. Ниже на рисунке 3.6 изображена каноническая картинка со структурой RTP-заголовка, которую я тоже не удержался и раскрасил в те же цвета.

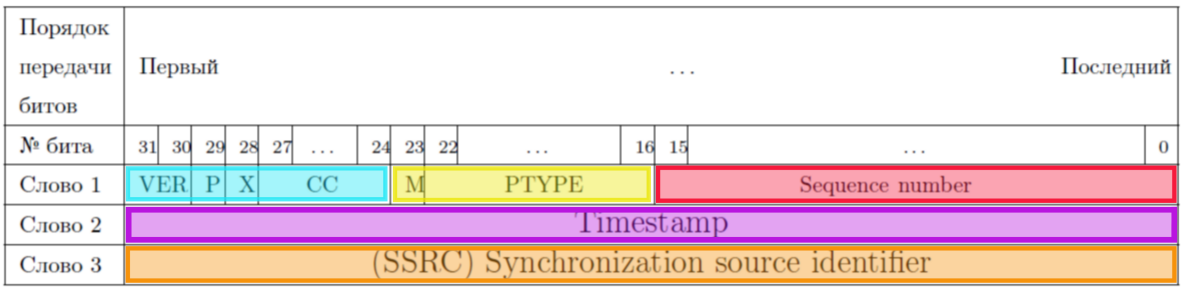


Рис. 3.6 – Заголовок RTP-пакета

VER — номер версии протокола (текущая версия 2);

P — флаг, который устанавливается в случаях, когда RTP-пакет дополняется пустыми байтами на конце;

X — флаг того, что заголовок расширенный;

CC — содержит количество *CSRC*-идентификаторов, следующих за постоянным заголовком (после слов 1..3), на рисунке таблица не показана;

M — маркер начала кадра или наличия речи в канале (если используется детектор пауз в речи). Если приемник не содержит детектор пауз в речи, то этот бит должен быть установлен постоянно;

PTYPE — указывает формат полезной нагрузки;

Sequence number — номер пакета, используется для восстановления порядка воспроизведения пакетов, так как реальная ситуация когда пакеты могут достигнуть приемника не в том порядке в котором их отправили. Начальное значение должно быть случайным, это делается для того, чтобы если применяется шифрование RTP-потока затруднить его взлом. Также это поле позволяет обнаруживать пропуски пакетов;

Timestamp — метка времени. Время измеряется в выборках сигнала, т.е. если пакет содержит 160 выборок, то метка времени следующего пакета будет больше на 160. Начальное значение временной метки должно быть случайным;

SSRC — идентификатор источника пакета, он должен быть уникальным. Его лучше генерировать случайным образом перед запуском RTP-потока.

Если вы будете разрабатывать свой передатчик или приемник RTP-пакетов, вам придется не раз рассматривать ваши пакеты, чтобы чтобы повысить продуктивность этого процесса, я вам рекомендую освоить использование фильтрации пакетов в *TShark*, она позволяет захватывать только те пакеты которые представляют интерес для вас. В условиях, когда в сети работают десятки RTP-устройств это очень ценно. В командной строке *TShark* параметры фильтрации задаются опцией "-f". Мы использовали эту опцию когда хотели захватить пакеты с порта 8010: –f "udp port 8010" Параметры фильтрации по своей сути это набор критериев которым должен соответствовать "отлавливаемый" пакет. Условие может проверять адрес, порт, значение определенного байта в пакете. Условия можно объединять логическими операциями "И", "ИЛИ" и т.п. Очень мощный инструмент.

Если вы хотите просмотреть динамику изменения полей в пакетах, вам потребуется продублировать вывод *TShark* в файл, как это было показано в прошлом разделе, с помощью передачи вывода *TShark* на вход *tee*. Далее открыв лог-файл с помощью *less*, *vim* или другим инструментом, способным

быстро работать с огромными текстовыми файлами и выполнять поиск строк, вы сможете выяснить все нюансы поведения полей пакетов в RTP-потоке.

3.8 Извлечение полезной нагрузки

Если вам потребуется прослушать сигнал передаваемый RTP-потоком, то нужно воспользоваться версией *TShark* с визуальным интерфейсом, т.е. *Wireshark*. Путем несложных манипуляций мышью там можно прослушать, увидеть осциллограмму сигнала. Но при одном условии — если он будет закодирован в формате *μ-law* или *α-low*.

Если полезная нагрузка представлена 16-битными отсчетами, то Wireshark не поможет вам её проиграть. Тогда можно воспользоваться следующим подходом.

С помощью *TShark*, выполняем захват пакетов в файл, при этом указываем фильтр, по которому *TShark* будет выбирать пакеты. В данном случае использован фильтр -f "src port 8010" для получения пакетов отправленных с порта 8010. Опцией «-a» задается время, в течение которого будет выполняться захват. Без этой опции захват можно остановить как обычно, по Ctrl+C.

```
$ sudo tshark -i any -f "udp port 8010" -a duration:60 -w ~/out
```

Затем извлекаем и файла полезную нагрузку: tshark -r ~/out -T fields -e rtp.payload, в потоке символов полезной нагрузки убираем символы новой строки и двоеточия: tr -d '\n',':' . После этого преобразуем отсчеты, представленные в текстовом виде (шестнадцатиричные числа), в бинарный вид: xxd -r -ps.

Одной строкой это делается так:

```
$ tshark -r ~/out -T fields -e rtp.payload | tr -d '\n',':' | xxd -r
    -ps > ~/out.raw
```

Получаем на выходе *raw*-файл *out.raw*, который представляет собой *wav*-файл без заголовка. Чтобы воспроизвести его на динамике компьютера, мы должны явно указать тип, формат данных, количество каналов:

```
$ aplay -t raw --format s16_be --channels 1 ~/out.raw
```

Порядок байтов в отсчетах здесь сетевой, поэтому использован формат big endian (s16_be).

3.9 Дуплексное переговорное устройство

В этом разделе мы сделаем дуплексное переговорное устройство. Схема изображена на рисунке 3.7.

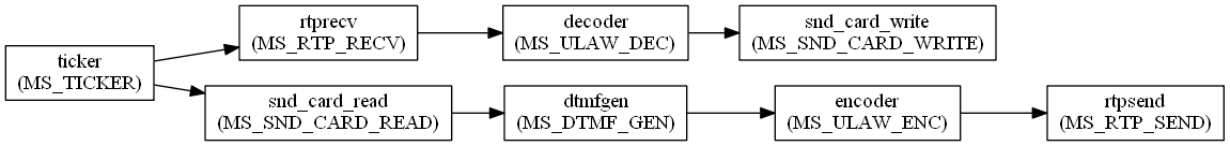


Рис. 3.7 – Дуплексное переговорное устройство

Нижняя цепь фильтров образует передающий тракт, который начинается со звуковой карты. Она выдает отсчеты сигнала с микрофона. По умолчанию это происходит в темпе 8000 отсчетов в секунду. Разрядность данных, которую используют звуковые фильтры медиастримера — 16 бит (следует заметить, что это не принципиально, при желании можно написать фильтры которые будут работать с большей разрядностью). Данные сгруппированы в блоки по 160 отсчетов. Таким образом, каждый блок имеет размер 320 байт. Далее мы подаем данные на вход генератора, который в выключенном состоянии "прозрачен" для данных. Я его добавил на тот случай, когда вам при отладке надоест говорить в микрофон — вы сможете воспользоваться генератором, чтобы "прострелить" тракт тональным сигналом.

После генератора сигнал попадает на кодер, который преобразует наши 16-битные отсчеты по μ-закону (стандарт G.711) в восьмибитные. На выходе кодера мы уже имеем блок данных в два раза меньшего размера. В общем случае, мы можем передавать данные без сжатия, если нам не требуется экономить трафик. Но здесь полезно воспользоваться кодером, так как *Wireshark* может воспроизводить звук из RTP-потока только тогда когда он сжат по μ-закону или a-закону.

После кодера, полегчавшие блоки данных поступают на фильтр *rtpsend*, который положит их в RTP-пакет, установит необходимые флаги и отдаст их медиастримеру для передачи по сети в виде UDP-пакета.

Верхняя цепь фильтров образует приемный тракт, RTP-пакеты, полученные медиастримером из сети, поступают в фильтр *rtprecv*, на выходе которого, они появляются уже в виде блоков данных, каждый соответствует одному принятому пакету. Блок содержит только данные полезной нагрузки, в прошлом разделе на иллюстрации они были показаны зеленым цветом.

Далее блоки поступают на фильтр декодера, который преобразует находящиеся в них однобайтные отсчеты, в линейные, 16-битные. Которые уже могут

обрабатываться фильтрами медиастримера. В нашем случае мы просто отдаем их на звуковую карту, для воспроизведения на динамиках вашей гарнитуры.

Теперь перейдем к программной реализации. Для этого мы объединим файлы приемника и передатчика, которые мы размежевали до этого. Прежде мы использовали фиксированные настройки портов и IP-адресов, но теперь нам нужно, чтобы программа могла использовать те настройки, которые мы укажем при запуске. Для этого бы добавим функционал обработки аргументов командной строки, после чего мы сможем задавать IP-адрес и порт такого же переговорного устройства, с которым мы хотим установить связь.

Сначала добавим в программу структуру *app_vars*, которая будет хранить её настройки:

```
struct _app_vars
{
  int   local_port;                 /* Локальный порт. */
  int   remote_port;                /* Порт переговорного устройства на
     удаленном компьютере. */
  char remote_addr[128];            /* IP-адрес удаленного компьютера.
     */
  MSDtmfGenCustomTone dtmf_cfg; /* Настройки тестового сигнала
     генератора. */
};

typedef struct _app_vars app_vars;
```

В программе она будет объявлена как переменная с именем *vars*. Затем добавим функцию разбора аргументов командной строки:

Листинг 3.11: Функция разбора аргументов командной строки

```
/* Функция преобразования аргументов командной строки в
* настройки программы. */
void  scan_args(int argc, char *argv[], app_vars *v)
{
    char i;
    for (i=0; i<argc; i++)
    {
        if (!strcmp(argv[i], "--help"))
        {
            char *p=argv[0]; p=p + 2;
            printf("  %s walkie talkie\n\n", p);
            printf("--help      List of options.\n");
            printf("--version   Version of application.\n");
            printf("--addr      Remote abonent IP address string.\n"
               );
            printf("--port      Remote abonent port number.\n");
            printf("--lport     Local port number.\n");
            printf("--gen       Generator frequency.\n");
            exit(0);
```

```
        }

        if (!strcmp(argv[i], "--version"))
        {
            printf("0.1\n");
            exit(0);
        }

        if (!strcmp(argv[i], "--addr"))
        {
            strncpy(v->remote_addr, argv[i+1], 16);
            v->remote_addr[16]=0;
            printf("remote addr: %s\n", v->remote_addr);
        }

        if (!strcmp(argv[i], "--port"))
        {
            v->remote_port=atoi(argv[i+1]);
            printf("remote port: %i\n", v->remote_port);
        }

        if (!strcmp(argv[i], "--lport"))
        {
            v->local_port=atoi(argv[i+1]);
            printf("local port : %i\n", v->local_port);
        }

        if (!strcmp(argv[i], "--gen"))
        {
            v -> dtmf_cfg.frequencies[0] = atoi(argv[i+1]);
                printf("gen freq : %i\n", v -> dtmf_cfg.frequencies
                    [0]);
        }
    }
}
```

В результате разбора, аргументы командной строки будут помещены в поля структуры *vars*. Главная функция приложения будет собирать из фильтров передающий и приемный тракты, после подключения тикера управление будет передано бесконечному циклу, который, если частота генератора была задана ненулевой, будет перезапускать тестовый генератор — чтобы он работал без остановки. Эти перезапуски будут нужны генератору из-за его особенности построения, почему-то он не может выдавать сигнал длительностью более 16 секунд. При этом следует заметить, что длительность у него задается 32-битным числом.

Программа целиком показана в листинге 3.12.

Листинг 3.12: Имитатор переговорного устройства

```
/* Файл mstest8.c Имитатор переговорного устройства. */
```

```
#include <mediastreamer2/mssndcard.h>
#include <mediastreamer2/dtmfgen.h>
#include <mediastreamer2/msrtp.h>

/* Подключаем файл общих функций. */
#include "mstest_common.c"

/*-------------------------------------------------------------------*/
struct _app_vars
{
    int  local_port;                /* Локальный порт. */
    int  remote_port;               /* Порт переговорного устройства
       на удаленном компьютере. */
    char remote_addr[128];          /* IP-адрес удаленного компьютера.
        */
    MSDtmfGenCustomTone dtmf_cfg; /* Настройки тестового сигнала
       генератора. */
};

typedef struct _app_vars app_vars;

/*-------------------------------------------------------------------*/
/* Создаем дуплексную RTP-сессию. */
RtpSession* create_duplex_rtp_session(app_vars v)
{
    RtpSession *session = create_rtpsession (v.local_port, v.
       local_port + 1, FALSE, RTP_SESSION_SENDRECV);
    rtp_session_set_remote_addr_and_port(session, v.remote_addr, v.
       remote_port, v.remote_port + 1);
    rtp_session_set_send_payload_type(session, PCMU);
    return session;
}

/*-------------------------------------------------------------------*/
/* Функция преобразования аргументов командной строки в
* настройки программы. */
void  scan_args(int argc, char *argv[], app_vars *v)
{
    char i;
    for (i=0; i<argc; i++)
    {
        if (!strcmp(argv[i], "--help"))
        {
            char *p=argv[0]; p=p + 2;
            printf("  %s walkie talkie\n\n", p);
            printf("--help      List of options.\n");
            printf("--version   Version of application.\n");
            printf("--addr      Remote abonent IP address string.\n"
               );
            printf("--port      Remote abonent port number.\n");
            printf("--lport     Local port number.\n");
```

```
            printf("--gen ------- Generator frequency.\n");
            exit(0);
        }

        if (!strcmp(argv[i], "--version"))
        {
            printf("0.1\n");
            exit(0);
        }

        if (!strcmp(argv[i], "--addr"))
        {
            strncpy(v->remote_addr, argv[i+1], 16);
            v->remote_addr[16]=0;
            printf("remote addr: %s\n", v->remote_addr);
        }

        if (!strcmp(argv[i], "--port"))
        {
            v->remote_port=atoi(argv[i+1]);
            printf("remote port: %i\n", v->remote_port);
        }

        if (!strcmp(argv[i], "--lport"))
        {
            v->local_port=atoi(argv[i+1]);
            printf("local port : %i\n", v->local_port);
        }

        if (!strcmp(argv[i], "--gen"))
        {
            v -> dtmf_cfg.frequencies[0] = atoi(argv[i+1]);
                printf("gen freq : %i\n", v -> dtmf_cfg.frequencies
                    [0]);
        }
    }
}

/*-------------------------------------------------------------------------*/
int main(int argc, char *argv[])
{
    /* Устанавливаем настройки по умолчанию. */
    app_vars vars={5004, 7010, "127.0.0.1", {0}};

    /* Устанавливаем настройки настройки программы в
     * соответствии с аргументами командной строки. */
    scan_args(argc, argv, &vars);

    ms_init();

    /* Создаем экземпляры фильтров передающего тракта. */
    MSSndCard *snd_card =
```

```
    ms_snd_card_manager_get_default_card(ms_snd_card_manager_get
        ());
MSFilter *snd_card_read = ms_snd_card_create_reader(snd_card);
MSFilter *dtmfgen = ms_filter_new(MS_DTMF_GEN_ID);
MSFilter *rtpsend = ms_filter_new(MS_RTP_SEND_ID);

/* Создаем фильтр кодера. */
MSFilter *encoder = ms_filter_create_encoder("PCMU");

/* Регистрируем типы нагрузки. */
register_payloads();

/* Создаем дуплексную RTP-сессию. */
RtpSession* rtp_session= create_duplex_rtp_session(vars);
ms_filter_call_method(rtpsend, MS_RTP_SEND_SET_SESSION,
    rtp_session);

/* Соединяем фильтры передатчика. */
ms_filter_link(snd_card_read, 0, dtmfgen, 0);
ms_filter_link(dtmfgen, 0, encoder, 0);
ms_filter_link(encoder, 0, rtpsend, 0);

/* Создаем фильтры приемного тракта. */
MSFilter *rtprecv = ms_filter_new(MS_RTP_RECV_ID);
ms_filter_call_method(rtprecv, MS_RTP_RECV_SET_SESSION,
    rtp_session);

/* Создаем фильтр декодера, */
MSFilter *decoder=ms_filter_create_decoder("PCMU");

/* Создаем фильтр звуковой карты. */
MSFilter *snd_card_write = ms_snd_card_create_writer(snd_card);

/* Соединяем фильтры приёмного тракта. */
ms_filter_link(rtprecv, 0, decoder, 0);
ms_filter_link(decoder, 0,  snd_card_write, 0);

/* Создаем источник тактов - тикер. */
MSTicker *ticker = ms_ticker_new();

/* Подключаем источник тактов. */
ms_ticker_attach(ticker, snd_card_read);
ms_ticker_attach(ticker, rtprecv);

/* Если настройка частоты генератора отлична от нуля, то
    запускаем генератор. */
if (vars.dtmf_cfg.frequencies[0])
{
    /* Настраиваем структуру, управляющую выходным сигналом
        генератора. */
    vars.dtmf_cfg.duration = 10000;
    vars.dtmf_cfg.amplitude = 1.0;
```

```
    }

    /* Организуем цикл перезапуска генератора. */
    while(TRUE)
    {
        if(vars.dtmf_cfg.frequencies[0])
        {
            /* Включаем звуковой генератор. */
            ms_filter_call_method(dtmfgen, MS_DTMF_GEN_PLAY_CUSTOM,
                    (void*)&vars.dtmf_cfg);
        }
        /* Укладываем тред в спячку на 20мс, чтобы другие треды
         * приложения получили время на работу. */
        ms_usleep(20000);
    }
}
```

Компилируем. Далее программу можно запустить на двух компьютерах. Или на одном, как это я буду делать сейчас. Запускаем *TShark* со следующими аргументами:

```
$ sudo tshark -i lo -f "udp dst port 7010" -P -V -O RTP -o rtp.
   heuristic_rtp:TRUE -x
```

Если поле запуска в консоли появится только сообщение о начале захвата, то это добрый знак -значит наш порт скорее всего не занят другими программами. В другом терминале запускаем экземпляр программы, который будет имитировать "удаленное" переговорное устройство с номером порта 7010:

```
$ ./mstest8 - --port 9010 - --lport 7010
```

Как видно из текста программы, по умолчанию используется IP-адрес 127.0.0.1 (локальная петля).

Еще в одном терминале запускаем второй экземпляр программы, который имитирует локальное устройство. Используем дополнительный аргумент, который разрешает работу встроенного тестового генератора:

```
$ ./mstest8 - --port 7010 - --lport 9010 - --gen 440
```

В этот момент, в консоли с *TShark* должны начать мелькать пакеты передаваемые в сторону "удаленного" устройства, а из динамика компьютера раздастся непрерывный тональный сигнал.

Если все произошло как по писанному, то перезапускаем второй экземпляр программы, но уже без ключа и аргумента "*--gen 440*". Роль генератора теперь будете исполнять вы. После этого можно пошуметь в микрофон, в динамике или наушниках вы должны услышать соответствующий звук. Возможно даже

возникнет акустическое самовозбуждение, убавьте громкость динамика и эффект пропадет.

Если у вы запустили эти программы на двух компьютерах и не запутались в IP-адресах, то вас ждет тот же результат — двусторонняя речевая связь цифрового качества.

Мы рассмотрели в этой главе примеры типовых случаев применения фильтров медиастримера. При этом были использованы простые схемы, где данные перемешаются по одной цепочке фильтров. Следует сказать, что применяя фильтры MS_MIXER, MS_TEE, MS_JOIN можно соединять и разветвлять потоки блоков данных, что дает возможность получить более сложные топологии их обработки.

В следующей главе мы научимся писать свои собственные фильтры — плагины, благодаря этому навыку, вы сможете применить медиастример не только для звука и видео, но и в какой-нибудь иной специфической области.

Глава 4

Разработка фильтров

В этой главе мы научимся писать фильтры и добавим их в проект переговорного устройства.

Плагин — независимо компилируемый программный модуль, динамически подключаемый к медиастримеру и предназначенный для расширения его возможностей. Это позволяет стороннему разработчику, т.е. вам, применять медиастример к решению задач, которых его авторы изначально и не предполагали.

4.1 Общий подход

Чтобы использовать плагин в своей программе, вы с помощью *include* должны подключить заголовочный файл плагина. В теле программы, с помощью функции *ms_filter_register()* выполнить регистрацию нового фильтра. Естественно, ваша программа и исходник плагина должны быть скомпилированы и собраны в одно приложение.

Теперь обратимся к написанию плагина. Все фильтры медиастримера и плагины подчиняются в написании общему канону, что значительно облегчает понимание устройства очередного фильтра, который вы захотели изучить. Поэтому далее, чтобы не размножать сущности, плагины будем называть фильтрами.

4.2 Приступаем к написанию фильтра

Предположим, мы хотим разработать новый фильтр с названием НАШ_ФИЛЬТР (NASH_FILTR). Он будет выполнять элементарную вещь — получать блоки со своего единственного входа и передавать на свои пять выходов. А еще он будет формировать события, в случае если через него

пройдет более пяти блоков с уровнем сигнала ниже заданного порога и если через него пройдет более пяти блоков с уровнем выше порога. Порог будет задаваться с помощью метода фильтра. Второй и третий методы будут разрешать/воспрещать прохождение блоков на выходы.

4.3 Заголовочный файл

В медиастримере, для его взаимодействия с фильтрами, реализован программный интерфейс. Поэтому каждый фильтр должен иметь заголовочный файл, в котором делаются объявления, необходимые для того, чтобы программный интерфейс медиастримера мог правильно взаимодействовать с экземплярами данного фильтра. С этого заголовочного файла и нужно начинать написание фильтра.

В первых строках он должен подключить файл *msfilter.h*, с помощью макроса MS_FILTER_METHOD объявить методы нового фильтра (если они есть), объявить события генерируемые фильтром (если они есть) и объявить экспортируемую структуру типа *MSFilterDesc* с описанием параметров фильтра, листинг 4.1.

Листинг 4.1: Структура MSFilterDesc

```
struct _MSFilterDesc{
    MSFilterId id;     /* Идентификатор типа фильтра заданный в файле
        allfilters.h или нами самими.  */
    const char *name; /* Имя фильтра (латиницей конечно). */
    const char *text; /** Короткий текст, описывающий фильтр. */
    MSFilterCategory category; /* Категория фильтра, описывающая его
        роль. */
    const char *enc_fmt; /* sub-mime используемого формата, должен
       быть указан для категорий фильтров MS_FILTER_ENCODER или
       MS_FILTER_DECODER */
    int ninputs; /* Количество входов. */
    int noutputs; /*Количество выходов.  */
    MSFilterFunc init; /* Функция начальной инициализации фильтра.
       */
    MSFilterFunc preprocess; /* Функция вызываемая однократно перед
       запуском фильтра в работу. */
    MSFilterFunc process; /**< Функция выполняющая основную работу
       фильтра, вызываемая на каждый тик  тикера MSTicker. */
    MSFilterFunc postprocess; /* Функция завершения работы фильтра,
       вызывается однократно после последнего вызова process(),
       перед удалением фильтра. */
    MSFilterFunc uninit; /**< Функция завершения работы фильтра,
       выполняет освобождение памяти, которая была занята при
       создании внутренних структур фильтра. */
    MSFilterMethod *methods; /* Таблица методов фильтра. */
```

```
    unsigned int flags; /* Специальные флаги фильтра  описанные в
        перечислении  MSFilterFlags. */
};

/*
 Структура для описания фильтра.
*/
typedef struct _MSFilterDesc MSFilterDesc;
```

Типы, используемые структурой можно изучить по файлу *msfilter.h*. Заголовочный файл нашего фильтра будет иметь вид показанный в листинге 4.2.

Листинг 4.2: Заголовочный файл фильтра-разветвителя и нойзгейта

```
/* Файл nash_filter.h, Фильтр–разветвитель и нойзгейт. */

#ifndef myfilter_h
#define myfilter_h

/* Подключаем заголовочный файл с перечислением фильтров
   медиастримера. */
#include <mediastreamer2/msticker.h>

/*
    Задаем числовой идентификатор нового типа фильтра.  Это число не
        должно
    совпадать ни с одним из других типов.  В медиастримере  в файле
        allfilters.h
    есть соответствующее перечисление enum MSFilterId. К сожалению,
        непонятно
    как определить максимальное занятое значение, кроме как заглянуть
         в этот
    файл. Но мы возьмем в качестве id для нашего фильтра заведомо
        большее
    значение: 4000.  Будем полагать, что разработчики добавляя новые
        фильтры, не
    скоро доберутся до этого номера.
    */
#define NASH_FILTER_ID 4000

/*
    Определяем методы нашего фильтра. Вторым параметром макроса
        должен
    порядковый номер метода, число от 0.  Третий параметр это тип
        аргумента
    метода, указатель на который будет передаваться методу при вызове
        . У методов
    аргументов может и не быть, как показано ниже.
    */
#define NASH_FILTER_SET_TRESHOLD MS_FILTER_METHOD(NASH_FILTER_ID ,
```

```
    0, float)
#define NASH_FILTER_TUNE_OFF      MS_FILTER_METHOD_NO_ARG(
    NASH_FILTER_ID ,1)
#define NASH_FILTER_TUNE_ON       MS_FILTER_METHOD_NO_ARG(
    NASH_FILTER_ID ,2)

/* Теперь определяем структуру, которая будет передаваться вместе с
    событием. */
struct _NASHFilterEvent
{
    /* Это поле, которое будет выполнять роль флага,
       0 - появились нули, 1 - появился сигнал.*/
    char state;
    /* Время, когда произошло событие. */
    uint64_t time;
};
typedef struct _NASHFilterEvent NASHFilterEvent;

/* Определяем событие для нашего фильтра. */
#define NASH_FILTER_EVENT MS_FILTER_EVENT(MS_RTP_RECV_ID, 0,
    NASHFilterEvent)

/* Определяем экспортируемую переменную, которая будет
    хранить характеристики для данного типа фильтров. */
extern MSFilterDesc nash_filter_desc;

#endif /* myfilter_h */
```

4.4 Исходный файл

Теперь можно заняться исходным файлом. Исходный код фильтра с комментариями показан в листинге 4.3. Здесь реализованы методы, которые мы объявили в заголовочном файле и обязательные функции фильтра. Затем ссылки на методы и функции в определенном порядке помещаются в экспортируемую структуру *nash_filter_desc*. Которая используется медиастримером для "вживления" экземпляров фильтров данного типа в рабочий процесс обработки данных.

Листинг 4.3: Исходный файл фильтра-разветвителя и нойзгейта

```
/* Файл nash_filter.c, Описывает фильтр–разветвитель и нойзгейт. */

#include "nash_filter.h"
#include <math.h>

#define NASH_FILTER_NOUTPUTS 5
```

```
/* Определяем структуру, которая хранит внутреннее состояние фильтра
   . */
typedef struct _nash_filterData
{
    bool_t disable_out;  /* Разрешение передачи блоков на выход. */
    int last_state;    /* Текущее состояние переключателя. */
    char zero_count;       /* Счетчик нулевых блоков. */
    char lag;              /* Количество блоков для принятия решения
       нойзгейтом. */
    char n_count;          /* Счетчик НЕнулевых блоков. */
    float skz_level;       /* Среднеквадратическое значение сигнала
       внутри
блока, при котором фильтр будет пропускать сигнал. Одновременно это
   порог
срабатывания, по которому будет формироваться событие.   */

} nash_filterData;

/*-------------------------------------------------------------------*/
/* Обязательная функция инициализации. */
static void nash_filter_init(MSFilter *f)
{
    nash_filterData *d=ms_new0(nash_filterData, 1);
    d->lag=5;
    f->data=d;
}

/*-------------------------------------------------------------------*/
/* Обязательная функция финализации работы фильтра,
   освобождается память. */
static void nash_filter_uninit(MSFilter *f)
{
    ms_free(f->data);
}

/*-------------------------------------------------------------------*/
/* Определяем образцовый массив с нулями, заведомо
   большего размера чем блок. */
char zero_array[1024]={0};

/* Определяем событие фильтра. */
NASHFilterEvent event;

/*-------------------------------------------------------------------*/
/* Функция отправки события. */
static void send_event(MSFilter *f, int state)
{
    nash_filterData *d =( nash_filterData* ) f->data;
     d->last_state = state;
    /* Устанавливаем время возникновения события,
       от момента первого тика. Время в миллисекундах. */
```

```
    event.time=f -> ticker -> time;
    event.state=state;
    ms_filter_notify(f, NASH_FILTER_EVENT, &event);
}

/*-------------------------------------------------------------------*/
/* Функция вычисляет среднеквадратическое (эффективное) значение
   сигнала внутри
  блока. */
static float calc_skz(nash_filterData *d, int16_t *signal, int
   numsamples)
{
    int i;
    float acc = 0;
    for (i=0; i<numsamples; i++)
    {
        int s=signal[i];
        acc = acc + s * s;
    }
    float skz = (float)sqrt(acc / numsamples);
    return skz;
}

/*-------------------------------------------------------------------*/
/* Обязательная функция основного цикла фильтра,
   вызывается с каждым тиком. */
static void nash_filter_process(MSFilter *f)
{
    nash_filterData *d=(nash_filterData*)f->data;

    /* Указатель на входное сообщение содержащее блок данных. */
    mblk_t *im;
    int i;
    int state;
    /* Вычитываем сообщения из входной очереди
       до полного её опустошения. */
    while((im=ms_queue_get(f->inputs[0]))!=NULL)
    {
        /* Если выходы запрещены, то просто удаляем входное
           сообщение. */
        if ( d -> disable_out)
        {
          freemsg(im);
          continue;
        }

        /* Измеряем уровень сигнала и принимаем решение об отправке
           сигнала. */
        float skz = calc_skz(d, (int16_t*)im->b_rptr, msgdsize(im));
        state = (skz > d->skz_level) ? 1 : 0;
        if (state)
        {
```

```
            d->n_count++;
            d->zero_count = 0;
        }
        else
        {
            d->n_count = 0;
            d->zero_count++;
        }
        if (((d->zero_count > d->lag) || (d->n_count > d->lag))
            && (d->last_state != state)) send_event(f, state);

        /* Приступаем к копированию входного сообщения и раскладке
           по выходам. Но
         * только по тем, к которым подключена нагрузка.
           Оригинальное сообщение
         * уйдет на выход с индексом 0, а его копии попадут на
           остальные
         * выходы. */
        int output_count = 0;
        mblk_t *outm; /* Указатель на сообщение с выходным блоком
           данных. */
        for(i=0; i < f->desc->noutputs; i++)
        {
            if (f->outputs[i]!=NULL)
            {
                if (output_count == 0)
                {
                    outm = im;
                }
                else
                {
                    /* Создаем легкую копию сообщения. */
                    outm = dupmsg(im);
                }
                /* Помещаем копию или оригинал входного сообщения на
                    очередной
                 * выход фильтра. */
                ms_queue_put(f->outputs[i], outm);
                output_count++;
            }
        }
    }
}

/*-----------------------------------------------------------------*/
/* Функция-обработчик вызова метода NASH_FILTER_SET_LAG. */
static int nash_filter_set_treshold(MSFilter *f, void *arg)
{
    nash_filterData *d=(nash_filterData*)f->data;
    d->skz_level=*(float*)arg;
    return 0;
}
```

```
/*-----------------------------------------------------------------------*/
/* Функция-обработчик вызова метода NASH_FILTER_TUNE_OFF. */
static int nash_filter_tune_off(MSFilter *f, void *arg)
{
    nash_filterData *d=(nash_filterData*)f->data;
    d->disable_out=TRUE;
    return 0;
}

/*-----------------------------------------------------------------------*/
/* Функция-обработчик вызова метода NASH_FILTER_TUNE_ON. */
static int nash_filter_tune_on(MSFilter *f, void *arg)
{
    nash_filterData *d=(nash_filterData*)f->data;
    d->disable_out=FALSE;
    return 0;
}

/*-----------------------------------------------------------------------*/
/* Заполняем таблицу методов фильтра, сколько методов
   мы определили в заголовочном файле столько ненулевых
   строк. */
static MSFilterMethod nash_filter_methods[]={
    { NASH_FILTER_SET_TRESHOLD, nash_filter_set_treshold },
    { NASH_FILTER_TUNE_OFF, nash_filter_tune_off },
    { NASH_FILTER_TUNE_ON, nash_filter_tune_on },
    { 0 , NULL } /* Маркер конца таблицы. */
};

/*-----------------------------------------------------------------------*/
/* Описание фильтра для медиастримера. */
MSFilterDesc nash_filter_desc=
{
    NASH_FILTER_ID,
    "NASH_FILTER",
    "A filter with noise gate that reads from input and copy to it's
       five outputs.",
    MS_FILTER_OTHER,
    NULL,
    1,
    NASH_FILTER_NOUTPUTS,
    nash_filter_init,
    NULL,
    nash_filter_process,
    NULL,
    nash_filter_uninit,
    nash_filter_methods,
    0
};

MS_FILTER_DESC_EXPORT(nash_filter_desc)
```

4.5 Применяем новый фильтр

Теперь, не откладывая в долгий ящик, применим наш фильтр в сделанном ранее в 3.9 переговорном устройстве, в котором теперь появится функция регистрации разговора и благодаря нашему фильтру, длительные паузы в речи не будут писаться в файл.

На рисунке 4.1 показана схема видоизмененного переговорного устройства. Наш собственноручный фильтр хотелось изобразить как-то по-особенному ярко. Поэтому вы сразу найдете на схеме наш фильтр.

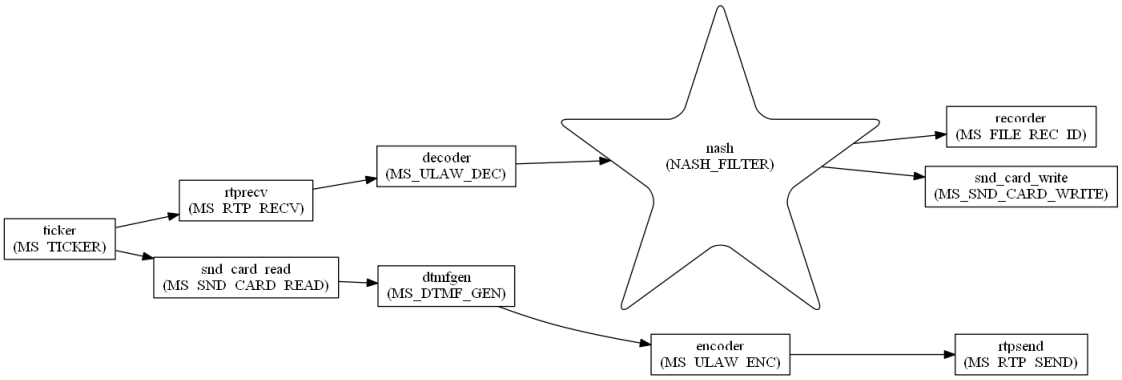


Рис. 4.1 – Наш фильтр в схеме

В схему добавился фильтр-регистратор, который пишет входной сигнал в файл формата *wav*. По замыслу, наш фильтр позволит избавить файл от длительных пауз в речи, тем самым сэкономит место на диске. В начале главы мы описали алгоритм действий фильтра. В основном приложении выполняется обработка событий, которые он формирует. Если событие содержит флаг "0", то основное приложение приостанавливает запись. Как только придет событие с флагом "1" запись возобновляется.

Исходный код программы показан в листинге 4.4. В ней к прежним аргументам командной строки добавились еще два: *--ng*, который задает уровень порога срабатывания фильтра и *--rec*, который запускает запись в файл с именем *record.wav*.

Листинг 4.4: Имитатор переговорного устройства с регистратором и нойзгейтом

```
/* Файл mstest9.c Имитатор переговорного устройства с регистратором
   и
   нойзгейтом. */

#include <mediastreamer2/mssndcard.h>
#include <mediastreamer2/dtmfgen.h>
#include <mediastreamer2/msrtp.h>
#include <mediastreamer2/msfilerec.h>

/* Подключаем наш фильтр. */
#include "nash_filter.h"

/* Подключаем файл общих функций. */
#include "mstest_common.c"

/*-------------------------------------------------------------------*/
struct _app_vars
{
    int  local_port;                /* Локальный порт. */
    int  remote_port;               /* Порт переговорного устройства
       на удаленном компьютере. */
    char remote_addr[128];          /* IP-адрес удаленного компьютера.
        */
    MSDtmfGenCustomTone dtmf_cfg; /* Настройки тестового сигнала
       генератора. */
    MSFilter* recorder;             /* Указатель на фильтр регистратор
       . */
    bool_t file_is_open;            /* Флаг того, что файл для записи
       открыт. */
    /* Порог, при котором прекращается запись принимаемого сигнала в
        файл. */
    float treshold;
    bool_t en_rec;                  /*Включить запись в файл.*/
};

typedef struct _app_vars app_vars;

/*-------------------------------------------------------------------*/
/* Создаем дуплексную RTP-сессию. */
RtpSession* create_duplex_rtp_session(app_vars v)
{
    RtpSession *session = create_rtpsession (v.local_port, v.
       local_port + 1,
            FALSE, RTP_SESSION_SENDRECV);
    rtp_session_set_remote_addr_and_port(session, v.remote_addr, v.
       remote_port,
            v.remote_port + 1);
    rtp_session_set_send_payload_type(session, PCMU);
    return session;
}

/*-------------------------------------------------------------------*/
```

```
/* Функция преобразования аргументов командной строки в
 * настройки программы. */
void  scan_args(int argc, char *argv[], app_vars *v)
{
    char i;
    for (i=0; i<argc; i++)
    {
        if (!strcmp(argv[i], "--help"))
        {
            char *p=argv[0]; p=p + 2;
            printf("  %s walkie talkie\n\n", p);
            printf("--help      List of options.\n");
            printf("--version   Version of application.\n");
            printf("--addr      Remote abonent IP address string.\n"
               );
            printf("--port      Remote abonent port number.\n");
            printf("--lport     Local port number.\n");
            printf("--gen       Generator frequency.\n");
            printf("--ng        Noise gate treshold level from 0. to
               1.0\n");
            printf("--rec       record to file 'record.wav'.\n");
            exit(0);
        }

        if (!strcmp(argv[i], "--version"))
        {
            printf("0.1\n");
            exit(0);
        }

        if (!strcmp(argv[i], "--addr"))
        {
            strncpy(v->remote_addr, argv[i+1], 16);
            v->remote_addr[16]=0;
            printf("remote addr: %s\n", v->remote_addr);
        }

        if (!strcmp(argv[i], "--port"))
        {
            v->remote_port=atoi(argv[i+1]);
            printf("remote port: %i\n", v->remote_port);
        }

        if (!strcmp(argv[i], "--lport"))
        {
            v->local_port=atoi(argv[i+1]);
            printf("local port : %i\n", v->local_port);
        }

        if (!strcmp(argv[i], "--gen"))
        {
            v -> dtmf_cfg.frequencies[0] = atoi(argv[i+1]);
```

```
            printf("gen freq : %i\n", v -> dtmf_cfg.frequencies[0]);
        }

        if (!strcmp(argv[i], "--ng"))
        {
            v -> dtmf_cfg.frequencies[0] = atoi(argv[i+1]);
            printf("noise gate treshold: %f\n", v -> treshold);
        }
         if (!strcmp(argv[i], "--rec"))
        {
            v -> en_rec = TRUE;
            printf("enable recording: %i\n", v -> en_rec);
        }
    }
}

/*-------------------------------------------------------------------*/
/* Функция обратного вызова, она будет вызвана фильтром, как только
   он
 * заметит, что наступила тишина или наоборот тишина сменилась
    звуками. */
static void change_detected_cb(void *data, MSFilter *f, unsigned int
    event_id,
        NASHFilterEvent *ev)
{
    app_vars *vars = (app_vars*) data;

    /* Если запись не была разрешена, то выходим. */
    if (! vars -> en_rec) return;

    if (ev -> state)
    {
        /* Возобновляем запись. */
        if(!vars->file_is_open)
        {
            ms_filter_call_method(vars->recorder, MS_FILE_REC_OPEN,
               "record.wav");
            vars->file_is_open = 1;
        }
        ms_filter_call_method(vars->recorder, MS_FILE_REC_START, 0);
        printf("Recording...\n");
    }
    else
    {
        /* Приостанавливаем запись. */
        ms_filter_call_method(vars->recorder, MS_FILE_REC_STOP, 0);
        printf("Pause...\n");
    }
}

/*-------------------------------------------------------------------*/
int main(int argc, char *argv[])
```

```
{
    /* Устанавливаем настройки по умолчанию. */
    app_vars vars={5004, 7010, "127.0.0.1", {0}, 0, 0, 0.01, 0};

    /* Устанавливаем настройки настройки программы в
     * соответствии с аргументами командной строки. */
    scan_args(argc, argv, &vars);

    ms_init();

    /* Создаем экземпляры фильтров передающего тракта. */
    MSSndCard *snd_card =
        ms_snd_card_manager_get_default_card(ms_snd_card_manager_get
            ());
    MSFilter *snd_card_read = ms_snd_card_create_reader(snd_card);
    MSFilter *dtmfgen = ms_filter_new(MS_DTMF_GEN_ID);
    MSFilter *rtpsend = ms_filter_new(MS_RTP_SEND_ID);

    /* Создаем фильтр кодера. */
    MSFilter *encoder = ms_filter_create_encoder("PCMU");

    /* Регистрируем типы нагрузки. */
    register_payloads();

    /* Создаем дуплексную RTP-сессию. */
    RtpSession* rtp_session = create_duplex_rtp_session(vars);
    ms_filter_call_method(rtpsend, MS_RTP_SEND_SET_SESSION,
       rtp_session);

    /* Соединяем фильтры передатчика. */
    ms_filter_link(snd_card_read, 0, dtmfgen, 0);
    ms_filter_link(dtmfgen, 0, encoder, 0);
    ms_filter_link(encoder, 0, rtpsend, 0);

    /* Создаем фильтры приемного тракта. */
    MSFilter *rtprecv = ms_filter_new(MS_RTP_RECV_ID);
    ms_filter_call_method(rtprecv, MS_RTP_RECV_SET_SESSION,
       rtp_session);

    /* Создаем фильтр декодера. */
    MSFilter *decoder=ms_filter_create_decoder("PCMU");
    //MS_FILE_REC_ID

    /* Регистрируем наш фильтр. */
    ms_filter_register(&nash_filter_desc);
    MSFilter *nash = ms_filter_new(NASH_FILTER_ID);

    /* Создаем фильтр звуковой карты. */
    MSFilter *snd_card_write = ms_snd_card_create_writer(snd_card);

    /* Создаем фильтр регистратора. */
    MSFilter *recorder=ms_filter_new(MS_FILE_REC_ID);
```

```
vars.recorder = recorder;

/* Соединяем фильтры приёмного тракта. */
ms_filter_link(rtprecv, 0, decoder, 0);
ms_filter_link(decoder, 0, nash, 0);
ms_filter_link(nash, 0, snd_card_write, 0);
ms_filter_link(nash, 1, recorder, 0);

/* Подключаем к фильтру функцию обратного вызова, и передаем ей
   в
 * качестве пользовательских данных указатель на структуру с
    настройками
 * программы, в которой среди прочих есть указать на фильтр
 * регистратора. */
ms_filter_set_notify_callback(nash,
        (MSFilterNotifyFunc)change_detected_cb, &vars);
ms_filter_call_method(nash,NASH_FILTER_SET_TRESHOLD, &vars.
   treshold);

/* Создаем источник тактов - тикер. */
MSTicker *ticker = ms_ticker_new();

/* Подключаем источник тактов. */
ms_ticker_attach(ticker, snd_card_read);
ms_ticker_attach(ticker, rtprecv);

/* Если настройка частоты генератора отлична от нуля, то
   запускаем генератор. */
if (vars.dtmf_cfg.frequencies[0])
{
    /* Настраиваем структуру, управляющую выходным сигналом
       генератора. */
    vars.dtmf_cfg.duration = 10000;
    vars.dtmf_cfg.amplitude = 1.0;
}

/* Организуем цикл перезапуска генератора. */
printf("Press␣ENTER␣to␣exit.\n␣");
char c=getchar();
while(c != '\n')
{
    if(vars.dtmf_cfg.frequencies[0])
    {
        /* Включаем звуковой генератор. */
        ms_filter_call_method(dtmfgen, MS_DTMF_GEN_PLAY_CUSTOM,
                (void*)&vars.dtmf_cfg);
    }
    char c=getchar();
    printf("--\n");
}
if (vars.en_rec ) ms_filter_call_method(recorder,
   MS_FILE_REC_CLOSE, 0);
```

```
}
```

Из-за того, что у нас добавились файлы и была использована библиотека *math*, командная строка для компиляции, усложнилась, и выглядит так:

```
$ gcc mstest9.c nash_filter.c -o mstest9 `pkg-config mediastreamer --libs --cflags` -lm
```

После сборки приложения запускаем его на первом компьютере с такими аргументами:

```
$ ./mstest9 --lport 7010 --port 8010 --addr <тут адрес второго компьютера> --rec
```

На втором компьютере запускаем со следующими настройками:

```
$ ./mstest9 --lport 8010 --port 7010 --addr <тут адрес первого компьютера>
```

После этого первый компьютер начнет записывать все, что вы скажете в микрофон второго. При этом в консоли будет написано слово "Recording...". Как только вы замолчите запись будет поставлена на паузу с выводом сообщения "Pause...". Возможно вам придется поэкспериментировать с уровнем порога.

В этой главе мы научились писать фильтры. Как вы могли заметить, в функции *nash_filter_process()* выполняются действия с блоками данных. Поскольку пример учебный, то там для простоты был задействован минимум возможностей медиастримера по манипуляциям с блоками данных.

В следующей главе мы рассмотрим организацию очередей сообщений и функции управления сообщениями в медиастримере. В дальнейшем это поможет вам разрабатывать фильтры с более сложной обработкой информации.

Глава 5

Механизм перемещения данных

Перемещение данных в медиастримере выполняется с помощью очередей, описываемых структурой *queue_t*. По очередям перемещаются вереницы сообщений типа *mblk_t*, которые сами по себе не содержат блоков данных, а только ссылки на предыдущее, следующее сообщение и на блок данных. Кроме этого, хочу подчеркнуть особо, есть еще поле для ссылки на сообщение такого же типа, которое позволяет организовать сообщения в односвязный список. Группу сообщений, объединенных таким списком, будем называть кортеж. Таким образом, любой элемент очереди может оказаться одиночным сообщением *mblk_t*, а может и головой кортежа сообщений *mblk_t*. Каждое сообщение кортежа может иметь свой подопечный блок данных. Зачем нужны кортежи мы обсудим немного позже.

Как было сказано выше, само по себе сообщение не содержит блок данных, вместо этого оно содержит только указатель на область памяти где хранится блок. В этой части общая картина работы медиастримера напоминает склад дверей в мультфильме "Корпорация монстров", где двери (ссылки на данные — т.е. комнаты) с безумной скоростью движутся по подвесным конвейерам, при этом сами комнаты остаются неподвижными.

Теперь, двигаясь по иерархии снизу вверх, рассмотрим детально перечисленные сущности механизма передачи данных в медиастримере.

5.1 Блок данных dblk_t

Блок данных состоит из заголовка и буфера данных. Заголовок описывается структурой dblk_t,

Листинг 5.1: Структура dblk_t

```
typedef struct datab
{
```

```
  unsigned char *db_base; /* Указатель на начало буфер данных. */
  unsigned char *db_lim;  /* Указатель на конец буфер данных. */
  void (*db_freefn)(void*); /* Функция освобождения памяти при
     удалении блока. */
  int db_ref; /* Счетчик ссылок. */
} dblk_t;
```

Поля структуры содержат указатели на начало буфера, конец буфера, функцию удаления буфера данных. Последний элемент в заголовке *db_ref* — счетчик ссылок, если он достигает нуля, это служит сигналом к удалению данного блока из памяти. Если блок данных был создан функцией *datab_alloc()*, то буфер данных будет размешен в памяти сразу после заголовка. Во всех других случаях буфер может располагаться где-то отдельно. В буфере данных будут находиться отсчеты сигнала или другие данные, которые мы хотим обрабатывать фильтрами.

Новый экземпляр блока данных создается с помощью функции:

```
dblk_t *datab_alloc(int size);
```

В качестве входного параметра ей передается размер данных, которые будет хранить блок. Памяти выделяется больше, чтобы в начале выделенной памяти разместить заголовок — структуру *datab*. Но при использовании других функций так происходит не всегда, в некоторых случаях буфер данных может располагаться отдельно от заголовка блока данных. Поля структуры при создании настраиваются так, чтобы её поле *db_base* указывало на начало области данных, а *db_lim* на её конец. Счетчик ссылок *db_ref* устанавливается в единицу. Указатель функции очистки данных устанавливается в ноль.

5.2 Сообщение mblk_t

Как было сказано, элементы очереди имеют тип *mblk_t*, он определен следующим образом:

Листинг 5.2: Структура mblk_t

```
typedef struct msgb
{
  struct msgb *b_prev;    // Указатель на предыдущий элемент списка.
  struct msgb *b_next;    // Указатель на следующий элемент списка.
  struct msgb *b_cont;    // Указатель для подклейки к сообщению
     других сообщений, для создания кортежа сообщений.
  struct datab *b_datap; // Указатель на структуру блока данных.
  unsigned char *b_rptr; // Указатель на начало области данных для
     чтения данных буфера b_datap.
  unsigned char *b_wptr; // Указатель на начало области данных для
     записи данных буфера b_datap.
```

```
  uint32_t reserved1;     // Зарезервированное поле1, медиастример
     помещает туда служебную информацию.
  uint32_t reserved2;     // Зарезервированное поле2, медиастример
     помещает туда служебную информацию.
  #if defined(ORTP_TIMESTAMP)
  struct timeval timestamp;
  #endif
  ortp_recv_addr_t recv_addr;
} mblk_t;
```

Структура *mblk_t* в начале содержит указатели *b_prev*, *b_next*, которые необходимы для организации двусвязного списка (которым является очередь *queue_t*).

Затем идет указатель *b_cont*, который используется только тогда, когда сообщение входит в кортеж. Для последнего сообщения в кортеже это указатель остается нулевым.

Далее мы видим указатель на блок данных *b_datap*, ради которого и существует сообщение. За ним идут указатели, на область внутри буфера данных блока. Поле *b_rptr* указывает место, с которого будут читаться данные из буфера. Поле *b_wptr* указывает место, с которого будут выполняться запись в буфер.

Оставшиеся поля носят служебный характер и не относятся к работе механизма передачи данных.

На рисунке 5.1показано одиночное сообщение с именем *m1* и блоком данных *d1*.

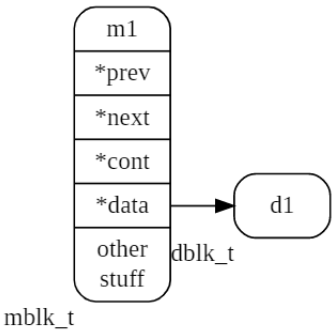


Рис. 5.1 – Сообщение mblk_t

На рисунке 5.2 изображен кортеж из трех сообщений *m1*, *m1_1*, *m1_2*.

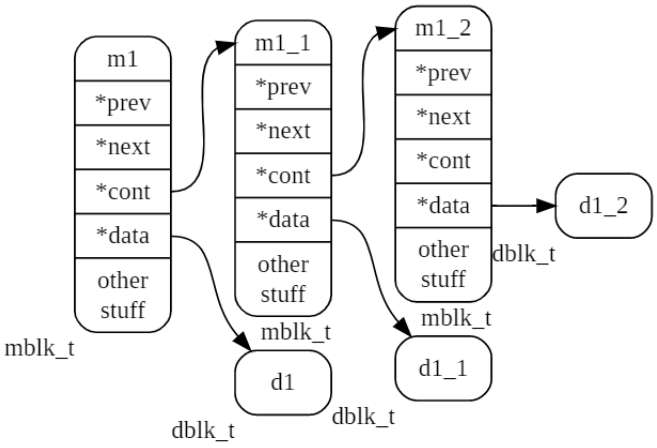


Рис. 5.2 – Кортеж из трех сообщений mblt_t

5.2.1 Функции работы с mblk_t

Новое сообщение *mblk_t* создается функцией *allocb()*:

```
mblk_t *allocb(int size, int pri);
```

она размещает в памяти новое сообщение *mblk_t* с блоком данных указанного размера *size*, второй аргумент — *pri* не используется в рассматриваемой версии библиотеки. На его место нужно подставлять макрос BPRI_MED, (после раскрытия макроса туда будет подставлен ноль). В ходе работы функции будет выделена память под структуру нового сообщения и вызвана функция *mblk_init()*, которая обнулит все поля созданного экземпляра структуры и затем, с помощью упомянутой выше *datab_alloc()*, создаст буфер данных. После чего будет выполнена настройка полей в структуре:

```
mp->b_datap = datab;
mp->b_rptr = mp->b_wptr = datab->db_base;
mp->b_next = mp->b_prev = mp->b_cont = NULL;
```

На выходе получаем новое сообщение с инициализированными полями и пустым буфером данных. Чтобы добавить в сообщение данные, нужно выполнить их копирование в буфер блока данных:

```
memcpy(msg->b_rptr, data, size);
```

где

data указатель на источник данных

size их размер. затем нужно обновить указатель на точку записи, чтобы он снова указывал на начало свободной области в буфере:

```
msg->b_wptr = msg->b_wptr + size
```

Если требуется создать сообщение из уже имеющегося буфера, без копирования, то для этого используется функция *esballoc()*:

```
mblk_t *esballoc(uint8_t *buf, int size, int pri, void (*freefn)(
    void*));
```

Она после создания сообщения и структуры блока данных, настроит его указатели на данные по адресу *buf*. Т.е. в данном случае буфер данных не располагается вслед за полями заголовка блока данных, как это было при создании блока данных функцией *datab_alloc()*. Переданный функции буфер с данными останется там где был, но с помощью указателей будет подстегнут к только что созданному заголовку блока данных, а тот соответственно к сообщению.

К одному сообщению *mblk_t* могут быть последовательно прицеплены несколько блоков данных. Это делается функцией *appendb()*:

```
mblk_t * appendb(mblk_t *mp, const char *data, int size, bool_t pad);
```

mp сообщение к которому будет добавлен еще один блок данных;

data указатель на блок, копия которого будет добавлена в сообщение;

size размер данных;

pad флаг того, что размер выделяемой памяти должен быть выравнен по границе 4 байт (дополнение будет выполнено нулями).

Если в имеющемся буфере данных сообщения достаточно места, то новые данные будут подклеены за уже имеющимися там данными. Если свободного места в буфере данных сообщения окажется меньше чем *size*, то создается новое сообщение, с достаточным размером буфера и данные копируются в его буфер. Это новое сообщение, подцепляется к исходному с помощью указателя *b_cont*. В этом случае сообщение превращается в кортеж.

Если в кортеж требуется добавить еще один блок данных, то нужно использовать функцию *msgappend()*:

```
void msgappend(mblk_t *mp, const char *data, int size, bool_t pad);
```

она отыщет последнее сообщение в кортеже (у него *b_cont* будет нулевым) и вызовет для этого сообщения функцию *appendb()*.

Узнать размер данных в сообщении или кортеже можно с помощью функции *msgdsize()*:

```
int msgdsize(const mblk_t *mp);
```

она пробежится по всем сообщениям кортежа и вернет суммарное количество данных в буферах данных этих сообщений. Для каждого сообщения количество данных вычисляется так:

```
mp->b_wptr - mp->b_rptr
```

Чтобы объединить два кортежа применяется функция *concatb()*:

```
mblk_t *concatb(mblk_t *mp, mblk_t *newm);
```

она присоединяет кортеж *newm* в хвост кортежа *mp* и возвращает указатель на последнее сообщение получившегося кортежа.

При необходимости, кортеж можно превратить в одно сообщение с единым блоком данных, это делается функцией *msgpullup()*:

```
void msgpullup(mblk_t *mp, int len);
```

если аргумент *len* равен -1, то размер отводимого буфера определяется автоматически. Если *len* положительное число, то будет создан буфер этого размера и в него будут скопированы данные сообщений кортежа. Если буфер закончится, то копирование будет на этом прекращено. Первое сообщение кортежа получит буфер нового размера со скопированными данными. Остальные сообщения будут удалены, а память возвращена в кучу.

При удалении структуры *mblk_t* учитывается счетчик ссылок блока данных, если при вызове *freeb()* он оказывается равен нулю, то буфер данных удаляется вместе с экземпляром *mblk_t*, который на него указывает.

Инициализация полей нового сообщения *mblk_init()*:

```
void mblk_init(mblk_t *mp);
```

Добавление в сообщение еще одной порции данных *appendb()*:

```
mblk_t * appendb(mblk_t *mp, const char *data, size_t size, bool_t
    pad);
```

Если новые данные не помещаются в свободное место буфера данных сообщения, то к сообщению прицепляется отдельно созданное сообщение с буфером

нужного размера (в первом сообщении устанавливается указатель на добавленное сообщение) сообщение превращается в кортеж.

Добавление порции данных в кортеж *msgappend()*:

```
void msgappend(mblk_t *mp, const char *data, size_t size, bool_t pad
    );
```

Функция вызывает *appendb()* в цикле.

Объединение двух кортежей в один *concatb()*:

```
mblk_t *concatb(mblk_t *mp, mblk_t *newm);
```

Сообщение *newm* будет присоединено к *mp*.

Создание копии одиночного сообщения *copyb()*:

```
mblk_t *copyb(const mblk_t *mp);
```

Полное копирование кортежа со всеми блоками данных *copymsg()*:

```
mblk_t *copymsg(const mblk_t *mp);
```

Элементы кортежа при этом копируются функцией *copyb()*.

Создание легкой копии сообщения *mblk_t*. При этом блок данных не копируется, а увеличивается счетчик его ссылок *db_ref*:

```
mblk_t *dupb(mblk_t *mp);
```

Создание легкой копии кортежа. Блоки данных не копируются, только увеличиваются их счетчики ссылок *db_ref*:

```
mblk_t *dupmsg(mblk_t* m);
```

Склейка всех сообщений кортежа в одно сообщение *msgpullup()*:

```
void msgpullup(mblk_t *mp,size_t len);
```

Если аргумент *len* равен -1, то размер отводимого буфера будет задан автоматически.

Удаление сообщения, кортежа *freemsg()*:

```
void freemsg(mblk_t *mp);
```

Счетчик ссылок блока данных уменьшается на единицу. Если он при этом достигает нуля, то блок данных тоже удаляется.

Подсчет суммарной объем данных в сообщении или кортеже.

```
size_t msgdsize(const mblk_t *mp);
```

Извлечение сообщения из хвоста очереди:

```
mblk_t *ms_queue_peek_last (q);
```

Копирование содержимого зарезервированных полей одного сообщения в другое сообщение(на самом деле в этих полях находятся флаги, которые используются медиастримером):

```
mblk_meta_copy(const mblk_t *source, mblk *dest);
```

5.3 Очередь queue_t

Очередь сообщений в медиастримере реализована как кольцевой двусвязный список. Каждый элемент списка содержит указатель на блок данных с отсчетами сигнала. Получается, что по очереди перемещаются только указатели на блок данных, в то время как сами данные остаются неподвижными. Структура описывающая очередь *queue_t*, показана ниже:

Листинг 5.3: Структура queue_t

```
typedef struct _queue
{
   mblk_t _q_stopper; /* "Холостой" элемент очереди, не указывает на
       данные, используется только для управления очередью. При
       инициализации очереди (qinit()) его указатели настраиваются
       так, чтобы они указывали на него самого. */
   int q_mcount;          /* Количество элементов в очереди. */
} queue_t;
```

Структура содержит поле — указатель *_q_stopper* типа **mblk_t*, он указывает на первый элемент (сообщение) в очереди. Второе поле структуры это счетчик сообщений, находящихся в очереди. Ниже на рисунке 5.3 показана очередь с именем *q1*, содержащая 4 сообщения *m1*,*m2*, *m3*, *m4*.

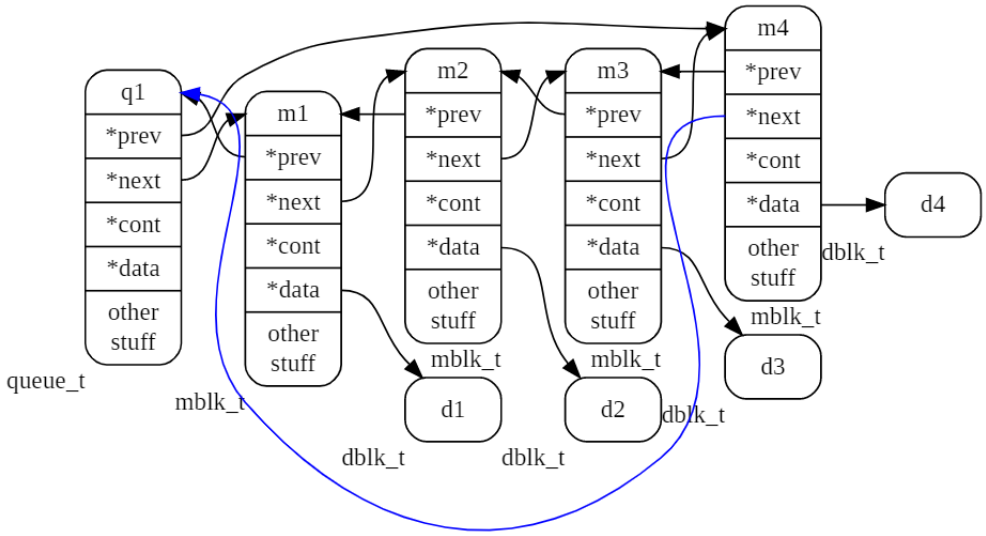


Рис. 5.3 – Очередь из 4х сообщений

На следующем рисунке 5.4 показана очередь с именем *q1*, содержащая 4 сообщения *m1*, *m2*, *m3*, *m4*, при этом *m2* - кортеж из двух сообщений.Сообщение *m2* является головой кортежа, в который водят еще два сообщения *m2_1* и *m2_2*.

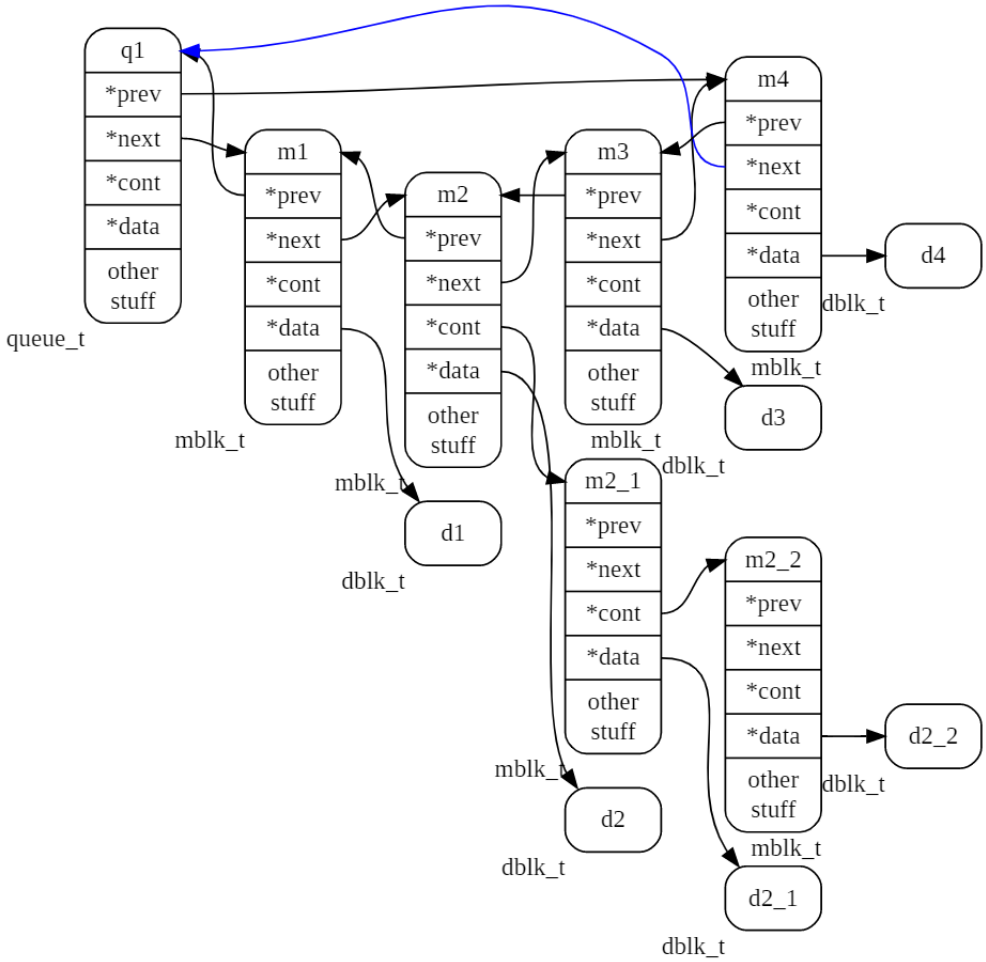


Рис. 5.4 – Очередь из 3х сообщений и кортежа

5.3.1 Функции работы с queue_t

Инициализацию очереди:

```
void qinit(queue_t *q);
```

Поле *_q_stopper* (далее будем называть его "стопор") инициализируется функцией *mblk_init()*, его указатель предыдущего элемента и следующего элемента настраиваются так, чтобы они показывали на него самого. Счетчик элементов очереди обнуляется.

Добавление нового элемента (сообщения):

```
void putq(queue_t *q, mblk_t *m);
```

Новый элемент *m* добавляется в конец списка, указатели элемента настраиваются так, чтобы стопор становился для него следующим элементом, а он для стопора предыдущим. Инкрементируется счетчик элементов очереди.

Извлечение элемента из очереди:

```
mblk_t * getq(queue_t *q);
```

извлекается то сообщение, которое стоит после стопора, счетчик элементов декрементируется. Если в очереди, кроме стопора, элементов нет, то возвращается 0.

Вставка сообщения в очередь:

```
void insq(queue_t *q, mblk_t *emp, mblk_t *mp);
```

Элемент *mp* вставляется перед элементом *emp*. Если *emp* равен 0, то сообщение добавляется в хвост очереди.

Извлечение сообщения из головы очереди:

```
void remq(queue_t *q, mblk_t *mp);
```

Счетчик элементов уменьшается на 1.

Чтение указателя на первый элемент в очереди:

```
mblk_t * peekq(queue_t *q);
```

Удаление всех элементов из очереди с удалением самих элементов:

```
void flushq(queue_t *q, int how);
```

Аргумент *how* не используется. Счетчик элементов очереди устанавливается в ноль.

Макрос чтения указателя на последний элемент очереди:

```
mblk_t * qlast(queue_t *q);
```

При работе с очередями сообщений следует иметь в виду, что при вызове *ms_queue_put(q, m)* с нулевым указателем на сообщение, функция зацикливается. Ваша программа зависнет. Аналогично ведет себя *ms_queue_next(q, m)*.

5.3.2 Соединение фильтров

Описанная выше очередь используются для передачи сообщений от одного фильтра к другому или от одного к нескольким фильтрам. Фильтры и их соединения между собой образуют направленный граф. Вход или выход фильтра будем называть обобщающим словом "пин". Для описания порядка соединений фильтров между собой, в медиастримере используется понятие

"сигнальная точка". Сигнальная точка это структура *MSCPoint*, которая содержит указатель на фильтр и номер одного из его пинов, соответственно она описывает соединение одного из входов или выходов фильтра.

Сигнальная точка графа обработки данных

Листинг 5.4: Структура MSCPoint

```
typedef struct _MSCPoint
{
  struct _MSFilter *filter; /* Указатель на фильтр медиастримера. */
  int pin;                  /* Номер одного из входов или выходов
     фильтра, т.е. пин. */
} MSCPoint;
```

Пины фильтров нумеруются начиная с нуля. Соединение двух пинов очередью сообщений описывается структурой *MSQueue*, которая содержит очередь сообщений и указатели на две сигнальные точки, которые она соединяет:

```
typedef struct _MSQueue
{
  queue_t q;
  MSCPoint prev;
  MSCPoint next;
}MSQueue;
```

Будем называть эту структуру «сигнальный линк». Каждый фильтр медиастримера, содержит таблицу входных и таблицу выходных сигнальных линков (*MSQueue*). Размер таблиц задается при создании фильтра, мы это уже делали с помощью экспортируемой переменной типа *MSFilterDesc*, когда разрабатывали наш собственный фильтр в главе 4. Ниже, в листинге 5.5, показана структура описывающая любой фильтр в медиастримере, *MSFilter*:

Листинг 5.5: Структура MSFilter

```
struct _MSFilter{
    MSFilterDesc *desc;     /* Указатель на дескриптор фильтра. */
    /* Защищенные атрибуты, их нельзя сдвигать или убирать иначе
       будет нарушена работа с плагинами. */
    ms_mutex_t lock;        /* Семафор. */
    MSQueue **inputs;       /* Таблица входных линков. */
    MSQueue **outputs;      /* Таблица выходных линков. */
    struct _MSFactory *factory; /* Указатель на фабрику, которая
       создала данный экземпляр фильтра. */
    void *padding;              /* Не используется, будет
       задействован если добавятся защищенные поля. */
```

```
    void *data;                    /* Указатель на произвольную
       структуру для хранения данных внутреннего состояния фильтра и
        промежуточных вычислений. */
    struct _MSTicker *ticker;     /* Указатель на объект тикера,
       который не должен быть нулевым когда вызывается функция
       process(). */
    /*private attributes, they can be moved and changed at any time
       */
    MSList *notify_callbacks; /* Список обратных вызовов,
       используемых для обработки событий фильтра. */
    uint32_t last_tick;          /* Номер последнего такта, когда
       выполнялся вызов process(). */
    MSFilterStats *stats;        /* Статистика работы фильтра.*/
    int postponed_task; /*Количество отложенных задач. Некоторые
       фильтры могут откладывать обработку данных (вызов process())
       на несколько тактов.*/
    bool_t seen;  /* Флаг, который использует тикер, чтобы помечать
       что этот экземпляр фильтра он уже обслужил на данном такте.*/
};

typedef struct _MSFilter MSFilter;
```

После того, как мы в Си-программе соединили фильтры в соответствии с нашим замыслом (но не подключили тикер), мы тем самым создали направленный граф, узлы которого, это экземпляры структуры *MSFilter*, а ребра это экземпляры сигнальных линков *MSQueue*.

5.4 Закулисная деятельность тикера

Когда я говорил вам, что тикер — это фильтр источник тактов, то была не вся правда о нем. Тикер это объект, который по часам выполняет запуск функций *process()* всех фильтров, к которым он прямо или косвенно подключен. Когда мы в Си-программе подключаем тикер к фильтру графа, мы показываем тикеру граф, которым он с этого момента будет управлять, пока мы его не отключим. После подключения, тикер начинает осматривать вверенный ему на попечение граф, составляя список фильтров в которые в него входят. Чтобы не "сосчитать" один и тот же фильтр дважды он помечает обнаруженные фильтры, устанавливая в них флажок *seen*. Поиск осуществляется по таблицам сигнальных линков, которые есть у каждого фильтра.

Во время свое ознакомительной экскурсии по графу, тикер проверяет есть ли среди фильтров, хотя бы один, который выполняет роль источника блоков данных. Если таковых не находится, то граф признается неправильным и тикер аварийно завершает работу программы.

Если граф оказался "правильным", у каждого найденного фильтра, для его инициализации, вызывается функция *preprocess()*. Как только наступает

момент для очередного такта обработки (по умолчанию каждые 10 миллисекунд), тикер вызывает функцию *process()* для всех найденных ранее фильтров источников, а затем и для остальных фильтров списка. Если фильтр имеет входные линки, то запуск функции *process()* повторяется до тех пор, пока очереди входных линков не опустеют. После этого, тикер переходит к следующему фильтру списка и "прокручивает" его до освобождения входных линков от сообщений. Тикер переходит от фильтра к фильтру пока не закончится список. На этом обработка такта заканчивается.

Теперь мы вернемся к кортежам и поговорим о том, для чего в медиастример была добавлена такая сущность. В общем случае, объем данных, необходимый алгоритму, работающему внутри фильтра не совпадает и не кратен, размеру буферов данных поступающих на вход. Например, мы пишем фильтр, который выполняет быстрое преобразование Фурье, которое по определению может обрабатывать только блоки данных с размером равным степени двойки. Пусть это будет 512 отсчетов. Если данные генерируются телефонным каналом, то буфер данных каждого сообщения на входе будет приносить нам по 160 отсчетов сигнала. Есть соблазн не забирать данные со входа, пока в сигнальном линке не окажется необходимое количество данных. Но в этом случае возникнет коллизия с тикером, который будет безуспешно пытаться прокрутить фильтр до опустошения входного линка. Ранее мы обозначили это правило как третий принцип работы фильтра. В соответствии с этим принципом, функция *process()* фильтра должна забрать все данные из входных очередей.

Кроме этого со входа нельзя будет забрать только 512 отсчетов, так как забирать можно только целыми блоками, т.е. фильтру придется забрать 640 отсчетов и использовав 512 из них и хранить остаток до накопления новой порции данных. Таким образом, наш фильтр, помимо основной своей работы должен обеспечить вспомогательные действия по промежуточному хранению входных данных. Разработчики медиастримера да решения этой общей задачи разработали специальный объект — *MSBufferizer* (буферизатор), который решает эту задачу с помощью кортежей.

5.5 Буферизатор MSBufferizer

Это объект, который может накапливать входные данные внутри фильтра и начнет отдавать их в обработку, как только количество информации окажется достаточным для проворачивания алгоритма фильтра. Пока буферизатор копит данные, фильтр будет работать в холостом режиме, не затрачивая вычислительной мощности процессора. Но как только функция чтения из буферизатора вернет значение отличное от нуля, функция *process()* фильтра начинает забирать из буферизатора и обрабатывать данные порциями нужного размера, до их исчерпания. Невостребованные пока данные остают-

ся в буферизаторе как первый элемент кортежа, к которому прицепляются последующие блоки новых входных данных.

Структура *MSBufferizer*, которая описывает буферизатор показана в листинге 5.6.

Листинг 5.6: Структура MSBufferizer

```
struct _MSBufferizer
{
  queue_t q; /* Очередь сообщений. */
  int size; /* Суммарный размер данных находящихся в буферизаторе в
     данный момент. */
};

typedef struct _MSBufferizer MSBufferizer;
```

5.5.1 Функции работы с MSBufferizer

Создание нового экземпляра буферизатора:

```
MSBufferizer * ms_bufferizer_new(void);
```

Выделяется память, инициализируется в *ms_bufferizer_init()* и возвращается указатель.

Функция инициализации:

```
void ms_bufferizer_init(MSBufferizer *obj);
```

Инициализируется очередь q, поле *size* устанавливается в ноль.

Добавление сообщения:

```
void ms_bufferizer_put(MSBufferizer *obj, mblk_t *m);
```

Сообщение m добавляется в очередь. Вычисленный размер блоков данных прибавляется к *size*.

Перекладка в буферизатор всех сообщений очереди данных линка q:

```
void ms_bufferizer_put_from_queue(MSBufferizer *obj, MSQueue *q);
```

Перенос сообщений из линка q в буферизатор выполняется с помощью функции *ms_bufferizer_put()*.

Чтение из буферизатора:

```
int ms_bufferizer_read(MSBufferizer *obj, uint8_t *data, int datalen
    );
```

Если размер накопленных в буферизаторе данных оказывается меньше запрошенного (*datalen*), то функция возвращает ноль, копирование данных в *data* не выполняется. В противном случае выполняется последовательное копирование данных из кортежей находящихся в буферизаторе. После копирования кортеж удаляется и память освобождается. Копирование заканчивается в тот момент, когда будет скопировано *datalen* байтов. Если место в *data* кончается посреди блока данных источника, то в данном сообщении, он будет сокращен до оставшейся, еще не скопированной части. При следующем вызове копирование продолжится с этого места.

Чтение количества данных, которые доступны данный момент в буферизаторе:

```
int ms_bufferizer_get_avail(MSBufferizer *obj);
```

Возвращает поле *size* буферизатора.

Отбрасывание части данных, находящихся в буферизаторе:

```
void ms_bufferizer_skip_bytes(MSBufferizer *obj, int bytes);
```

Указанное количество байтов данных извлекается и отбрасывается. Отбрасываются самые старые данные.

Удаление всех сообщений находящихся в буферизаторе:

```
void ms_bufferizer_flush(MSBufferizer *obj);
```

Счетчик данных сбрасывается в ноль.

Удаление всех сообщений находящихся в буферизаторе:

```
void ms_bufferizer_uninit(MSBufferizer *obj);
```

Обнуление счетчика не выполняется.

Удаление буферизатора и освобождение памяти:

```
void ms_bufferizer_destroy(MSBufferizer *obj);
```

Примеры использования буферизатора можно найти в исходном коде нескольких фильтров медиастримера. Например в фильтре MS_L16_ENC, который выполняет перестановку байтов в отсчетах из сетевого порядка, в порядок хоста: l16.c

В следующей главе, мы рассмотрим вопрос отладки фильтров.

Глава 6

Отладка крафтовых фильтров

После того, как мы в предыдущей главе рассмотрели механизм перемещения данных, будет логично поговорить о скрывающихся в нем опасности. Одна из особенностей принципа "*data flow*" состоит в том, что выделение памяти из кучи происходит в фильтрах, которые находятся у истоков потока данных, а освобождение памяти с возвращением в кучу делают уже фильтры, расположенные в конце потока. Кроме этого, создание новых данных и их уничтожение может происходить где-то в промежуточных точках. В общем случае, освобождение памяти выполняет не тот фильтр, что создал блок данных.

С точки зрения прозрачного мониторинга за памятью было бы разумно, чтобы фильтр, получая входной блок, после обработки тут же уничтожал его с освобождением памяти, а на выход выставлял бы вновь созданный блок с выходными данными. В этом случае утечка памяти в фильтре легко бы трассировалась — если анализатор обнаружил утечку в фильтре, то значит следующий за ним фильтр не уничтожает входящие блоки надлежащим образом и ошибка в нем. Но с точки зрения поддержания высокой производительности, такой подход к работе с блоками данных, не продуктивен — он приводит к большому количество операций по выделению/освобождению памяти под блоки данных без какого либо полезного выхлопа.

По этой причине фильтры медиастримера, чтобы не замедлять обработку данных, при копировании сообщений используют функции создающие легкие копии (мы рассказывали о них в прошлой главе). Эти функции только создают новый экземпляр заголовка сообщения "пристегивая" к нему блок данных от копируемого "старого" сообщения. В результате, к одному блоку данных оказываются привязанными два заголовка и выполняется инкремент счетчика ссылок в блоке данных. Но выглядеть это будет как два сообщения. Сообщений с таким "обобществленным" блоком данных может быть и больше, так например фильтр MS_TEE порождает сразу десяток таких легких копий,

распределяя их по своим выходам. При правильной работе всех фильтров в цепочке, к концу конвейера этот счетчик ссылок должен достигнуть нуля и будет вызвана функция освобождения памяти: *ms_free()*. Если вызова не происходит, то значит этот кусок памяти уже не вернется в кучу, т.е. он "утечет". Расплатой за использование легких копий служит утрата возможности легко установить (как это было бы в случае использования обычных копий) в каком фильтре графа утекает память.

Поскольку ответственность за поиск утечек памяти в "родных" фильтрах лежит на разработчиках медиастримера, то скорее всего вам не придется их отлаживать. Но вот с вашим крафтовым фильтром — вы сами кузнечик своего счастья и от вашей аккуратности будет зависеть время которое вы проведете в поисках утечек в вашем коде. Чтобы сократить ваше время мытарств с отладкой, мы должны рассмотреть приёмы локализации утечек при разработке фильтров. К тому же, может случиться так, утечка проявит себя только при применении фильтра в реальной системе, где количество "подозреваемых" может оказаться огромным, а время на отладку ограниченным.

Как проявляет себя утечка памяти?

Логично предположить, что в выводе программы *top* будет показываться нарастающий процент памяти, занимаемый вашим приложением.

Внешнее проявление будет состоять в том, что в какой-то момент система станет замедленно реагировать на движение мышки, медленно перерисовывать экран. Возможно также будет расти системный лог, съедая место на жестком диске. При этом ваше приложение начнет вести себя странно, не отвечать на команды, не может открыть файл и т.д.

Чтобы выявить факт возникновения утечки будем использовать анализатор памяти (далее анализатор). Это может быть *Valgrind* или встроенный в компилятор *gcc MemorySanitizer* или что-нибудь иное. Если анализатор покажет, что утечка происходит в одном из фильтров графа, то это означает что пора применить один из способов описанных ниже.

6.1 Метод трех сосен

Как уже было сказано выше, при утечке памяти анализатор укажет на фильтр, который запросил выделение памяти из кучи. Но не сможет сообщить, какой из фильтров "забыл" её вернуть, собственно виновника утечки. Тем самым, анализатор может только подтвердить наши опасения, но не указать на их корень.

Чтобы выяснить расположение в графе "нехорошего" фильтра, можно пойти путем сокращения графа до минимального количества узлов, при

котором анализатор все еще обнаруживает утечку и среди оставшихся «трех сосен» локализовать проблемный фильтр.

Но может случится так, что сокращая число фильтров в графе вы нарушите обычный ход взаимодействия фильтров с другими элементами вашей системы и утечка перестанет проявляться. В этом случае придется работать с полноразмерным графом и использовать подход, который изложен ниже.

6.2 Метод скользящего изолятора

Для простоты изложения для демонстрации воспользуемся графом, который состоит из одной цепочки фильтров. Он изображен на рисунке 6.1.

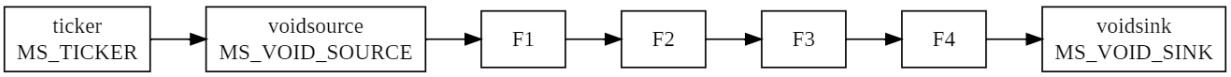


Рис. 6.1 – Типовой граф обработки данных

Обычный граф, в котором наравне с готовыми фильтрами медиастримера применены четыре крафтовых фильтра F1...F4, четырех разных типов, которые вы сделали давно и в их корректности не сомневаетесь. Тем не менее предположим, что в одном или нескольких из них имеется утечка памяти. Запуская нашу программу по надзором анализатора, из его отчета мы узнаем, что некий фильтр запросил некоторое количество памяти и не вернул его в кучу N-ое количество раз. Легко можно догадаться, что будет указано на внутренние функции фильтра типа MS_VOID_SOURCE. У него задача такая — забирать память из кучи. Возвращать её туда должны другие фильтры. Т.е. в данном случае для нас полезной информацией является обнаружение факта утечки.

Чтобы определить, в каком элементе цепочки произошло бездействие приведшее к утечке памяти, предлагается ввести дополнительный фильтр, который просто перекладывает сообщения со входа на выход, но при этом создает не легкую, в нормальную, "тяжелую" копию входного сообщения, полностью удаляя исходное сообщение, поступившее ему на вход. Будем называть такой фильтр изолятором. Полагаем, что поскольку фильтр простой, то утечка в нем исключена. И еще одно положительное свойство — если мы добавим его в любое место нашего графа, то это никак не скажется на работе схемы. Будем изображать фильтр-изолятор в виде круга с двойным контуром.

Включаем изолятор сразу после фильтра *voidsourse*, рисунок 6.2.

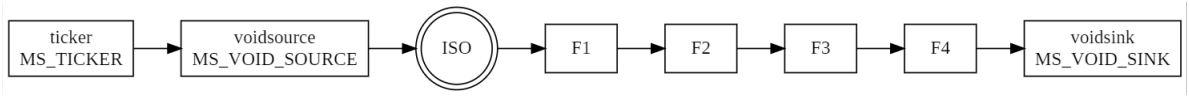


Рис. 6.2 – Схема с изолирующим фильтром

Снова запускаем программу с анализатором, и увидим, что в это раз, анализатор возложит вину на изолятор, а не *voidsource*. Ведь это он теперь создает блоки данных, которые потом теряются неизвестным нерадивым фильтром (или фильтрами). Следующим шагом сдвигаем изолятор по цепочке вправо, на один фильтр и снова запускаем анализ. Так, шаг за шагом двигая изолятор вправо, мы получим ситуацию, когда в в очередном отчете анализатора количество "утекших" блоков памяти уменьшится. Это значит, что на этом шаге изолятор оказался в цепочке сразу после проблемного фильтра. Если "плохой" фильтр был один, то утечка и вовсе пропадет. Таким образом мы локализовали проблемный фильтр (или один из нескольких). "Починив" фильтр, мы можем продолжить двигать изолятор вправо по цепочке до полной победы над утечками.

Реализация фильтра-изолятора

Реализация изолятора выглядит также как обычный фильтр. Заголовочный файл:

Листинг 6.1: Заголовочник изолирующего фильтра

```
/* Файл iso_filter.h  Описание изолирующего фильтра. */

#ifndef iso_filter_h
#define iso_filter_h

/* Задаем идентификатор фильтра. */
#include <mediastreamer2/msfilter.h>

#define MY_ISO_FILTER_ID 1024

extern MSFilterDesc iso_filter_desc;

#endif
```

Сам фильтр:

Листинг 6.2: Исходник изолирующего фильтра

```
/* Файл iso_filter.c  Описание изолирующего фильтра. */

#include "iso_filter.h"
```

```
    static void
iso_init (MSFilter * f)
{
}
    static void
iso_uninit (MSFilter * f)
{
}

    static void
iso_process (MSFilter * f)
{
    mblk_t *im;

    while ((im = ms_queue_get (f->inputs[0])) != NULL)
    {
        ms_queue_put (f->outputs[0], copymsg (im));
        freemsg (im);
    }
}

static MSFilterMethod iso_methods[] = {
    {0, NULL}
};

MSFilterDesc iso_filter_desc = {
    MY_ISO_FILTER_ID,
    "iso_filter",
    "A filter that reads from input and copy to its output.",
    MS_FILTER_OTHER,
    NULL,
    1,
    1,
    iso_init,
    NULL,
    iso_process,
    NULL,
    iso_uninit,
    iso_methods
};

MS_FILTER_DESC_EXPORT (iso_desc)
```

6.3 Метод подмены функций управления памятью

Для более тонких исследований, в медиастримере предусмотрена возможность подмены функций доступа к памяти на ваши собственные, которые

помимо основной работы будут фиксировать "Кто, куда и зачем". Подменяются три функции. Это делается следующим образом:

Листинг 6.3: Подмена функций управления памятью

```
OrtpMemoryFunctions reserv;
OrtpMemoryFunctions my;

reserv.malloc_fun = ortp_malloc;
reserv.realloc_fun = ortp_realloc;
reserv.free_fun = ortp_free;

my.malloc_fun = &my_malloc;
my.realloc_fun = &my_realloc;
my.free_fun = &my_free;

ortp_set_memory_functions(&my);
```

Такая возможность выручает в случаях, когда анализатор замедляет работу фильтров настолько, что нарушается работа системы в которую встроена наша схема. В такой ситуации приходится отказываться от анализатора и использовать подмену функций работы с памятью.

Мы рассмотрели алгоритм действий для простого графа, не содержащего разветвлений. Но этот подход можно применить и для других случаев, конечно с усложнением, но идея останется той же.

В следующей главе, мы рассмотрим вопрос оценки нагрузки на тикер и способы борьбы с чрезмерной вычислительной нагрузкой в медиастримере.

Глава 7

Управление нагрузкой на тикер

Нагрузка на тикер вычисляется как отношение времени проведенного в обработке всех фильтров графа, подключенного к этому тикеру к интервалу между тиками. По умолчанию интервал составляет 10мс, но если вы изменили семплингрейт, то расчет выполняется для установленного вами значения. Вычисление делается по сведениям, накопленным тикером во время работы. Медиастример предоставляет функцию, которая возвращает усредненную по нескольким результатам измерений величину, выраженную в процентах:

```
MS2_PUBLIC float ms_ticker_get_average_load(MSTicker *ticker);
```

Если возвращаемая величина близка к 100% это означает, что данный тикер еле поспевает сделать свою работу до начала очередного тика. Если у нас приложение, которому не требуется работа в реальном времени (например оно просто пишет звук в файл), то для нас не особо важно на какое время был отложен очередной вызов тикера. Но в реалтайм-приложении задержка обработки может повлиять на момент отправки RTP-пакета, что в свою очередь может сказаться на качестве звука или видео. В некоторых случаях влияние задержки отдельных пакетов можно купировать используя буфер пакетов на приемном конце (так называемый джиттер-буфер). При этом ваш звук будет воспроизводиться без дефектов, но с задержкой, пропорциональной длине буфера. Что может быть неприемлемо в случаях, когда звуковой сигнал используется для управления процессами реального времени.

Вам стоит начинать принимать меры уже тогда, когда нагрузка на тикер перешла границу 80%. При такой нагрузке на отдельных тактах тикер начинает запускать обработку с отставанием. Отставание тикера, это время, на которое был отложен очередной запуск тикера. Если отставание тикера превысило некоторую величину то генерируется событие *MSTickerLateEvent*:

Листинг 7.1: Структура MSTickerLateEvent

```
struct _MSTickerLateEvent{
  int lateMs; /* Запаздывание которое было в последний раз, в
     миллисекундах */
  uint64_t time; /* Время возникновения события, в миллисекундах */
int current_late_ms; /* Запаздывание на текущем тике, в
   миллисекундах */
};

typedef struct _MSTickerLateEvent MSTickerLateEvent;
```

По нему которому в консоль выводится сообщение, которое выглядит примерно так:

```
ortp-warning-MSTicker: We are late of 164 miliseconds
```

С помощью функции

```
void ms_ticker_get_last_late_tick(MSTicker *ticker,
   MSTickerLateEvent *ev);
```

можно узнать подробности о последнем таком событии.

7.1 Способы снижения нагрузки

Здесь у нас есть два варианта действий. Первый это изменить приоритет тикера, второй перенести часть работы тикера в другой тред. Рассмотрим эти варианты.

7.1.1 Изменение приоритета

Приоритет тикера имеет три градации, которые определены в перечислении *MSTickerPrio*:

Листинг 7.2: Перечисление MSTickerPrio

```
enum _MSTickerPrio{

MS_TICKER_PRIO_NORMAL, /* Приоритет соответствующий значению по
   умолчанию для данной ОС. */

MS_TICKER_PRIO_HIGH, /* Увеличенный приоритет устанавливается
   nodlinux/MacOS с помощью setpriority() или sched_setschedparams()
    устанавливается политика SCHED_RR. */
```

```
MS_TICKER_PRIO_REALTIME /* Наибольший приоритет, для него под Linux
    используется политика SCHED_FIFO. */
};

typedef enum _MSTickerPrio MSTickerPrio;
```

Чтобы поэкспериментировать с нагрузкой тикера, нам требуется схема, которая во время работы будет наращивать нагрузку и завершать работу когда нагрузка достигнет уровня 99%. В качестве нагрузки будем использовать схему:

```
ticker -> voidsource -> dtmfgen -> voidsink
```

Нагрузка будет увеличиваться добавлением между *dtmfgen* и *voidsink* нового элемента управления уровнем сигнала (тип фильтра MS_VOLUME), с коэффициентом передачи неравным единице, чтобы фильтр не филонил. Она показана на рисунке 7.1.

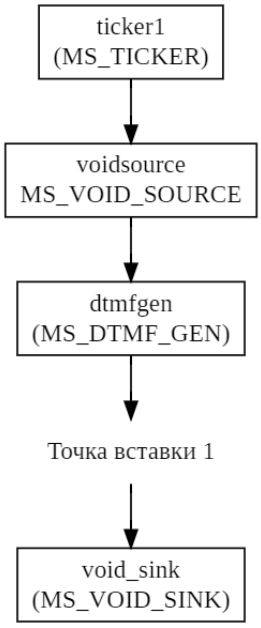


Рис. 7.1 – Нагрузка для тикера

Исходный код приведен в листинге 7.3, он снабжен комментариями, так что разобраться в нем не составит труда:

Листинг 7.3: Переменная вычислительная нагрузка

```
/* Файл mstest13.c Переменная вычислительная нагрузка. */

#include <stdio.h>
#include <signal.h>
```

```
#include <mediastreamer2/msfilter.h>
#include <mediastreamer2/msticker.h>
#include <mediastreamer2/dtmfgen.h>
#include <mediastreamer2/mssndcard.h>
#include <mediastreamer2/msvolume.h>

/*-------------------------------------------------------------------*/
struct _app_vars
{
    int  step;                /* Количество фильтров добавляемых за
       раз. */
    int  limit;               /* Количество фильтров на котором
       закончить работу. */
    int  ticker_priority;     /* Приоритет тикера. */
    char* file_name;          /* Имя выходного файла. */
    FILE *file;
};

typedef struct _app_vars app_vars;
/*-------------------------------------------------------------------*/
/* Функция преобразования аргументов командной строки в
* настройки программы. */
void  scan_args(int argc, char *argv[], app_vars *v)
{
    char i;
    for (i=0; i<argc; i++)
    {
        if (!strcmp(argv[i], "--help"))
        {
            char *p=argv[0]; p=p + 2;
            printf("  %s computational load\n\n", p);
            printf("--help      List of options.\n");
            printf("--version   Version of application.\n");
            printf("--step      Filters count per step.\n");
            printf("--tprio     Ticker priority:\n"
                   "            MS_TICKER_PRIO_NORMAL   0\n"
                   "            MS_TICKER_PRIO_HIGH     1\n"
                   "            MS_TICKER_PRIO_REALTIME 2\n");
            printf("--limit     Filters count limit.\n");
            printf("-o          Output file name.\n");
            exit(0);
        }

        if (!strcmp(argv[i], "--version"))
        {
            printf("0.1\n");
            exit(0);
        }

        if (!strcmp(argv[i], "--step"))
        {
```

```
            v->step = atoi(argv[i+1]);
            printf("step: %i\n", v->step);
        }

        if (!strcmp(argv[i], "--tprio"))
        {
            int prio = atoi(argv[i+1]);
            if ((prio >=MS_TICKER_PRIO_NORMAL) && (prio <=
               MS_TICKER_PRIO_REALTIME))
            {
                v->ticker_priority = atoi(argv[i+1]);
                printf("ticker priority: %i\n", v->ticker_priority);
            }
            else
            {
                printf(" Bad ticker priority: %i\n", prio);
                exit(1);
            }
        }

        if (!strcmp(argv[i], "--limit"))
        {
            v->limit = atoi(argv[i+1]);
            printf("limit: %i\n", v->limit);
        }

        if (!strcmp(argv[i], "-o"))
        {
            v->file_name=argv[i+1];
            printf("file namet: %s\n", v->file_name);
        }
    }
}

/*-------------------------------------------------------------------------*/
/* Структура для хранения настроек программы. */
app_vars vars;

/*-------------------------------------------------------------------------*/
void saveMyData()
{
    // Закрываем файл.
    if (vars.file) fclose(vars.file);
    exit(0);
}

void signalHandler( int signalNumber )
{
    static pthread_once_t semaphore = PTHREAD_ONCE_INIT;
    printf("\nsignal %i received.\n", signalNumber);
    pthread_once( & semaphore, saveMyData );
}
```

```
/*-------------------------------------------------------------------*/
int main(int argc, char *argv[])
{
    /* Устанавливаем настройки по умолчанию. */
    app_vars vars={100, 100500, MS_TICKER_PRIO_NORMAL, 0};

    // Подключаем обработчик Ctrl-C.
    signal( SIGTERM, signalHandler );
    signal( SIGINT,  signalHandler );

    /* Устанавливаем настройки настройки программы в
     * соответствии с аргументами командной строки. */
    scan_args(argc, argv, &vars);

    if (vars.file_name)
    {
        vars.file = fopen(vars.file_name, "w");
    }

    ms_init();
    /* Создаем экземпляры фильтров. */
    MSFilter  *voidsource=ms_filter_new(MS_VOID_SOURCE_ID);
    MSFilter  *dtmfgen=ms_filter_new(MS_DTMF_GEN_ID);

    MSSndCard *card_playback=ms_snd_card_manager_get_default_card(
       ms_snd_card_manager_get());
    MSFilter  *snd_card_write=ms_snd_card_create_writer(
       card_playback);
    MSFilter  *voidsink=ms_filter_new(MS_VOID_SINK_ID);

    MSDtmfGenCustomTone dtmf_cfg;

    /* Устанавливаем имя нашего сигнала, помня о том, что в массиве
       мы должны
     * оставить место для нуля, который обозначает конец строки. */
    strncpy(dtmf_cfg.tone_name, "busy", sizeof(dtmf_cfg.tone_name));
    dtmf_cfg.duration=1000;
    dtmf_cfg.frequencies[0]=440; /* Будем генерировать один тон,
       частоту второго тона установим в 0.*/
    dtmf_cfg.frequencies[1]=0;
    dtmf_cfg.amplitude=1.0; /* Такой амплитуде синуса должен
       соответствовать результат измерения 0.707.*/
    dtmf_cfg.interval=0.;
    dtmf_cfg.repeat_count=0.;

    /* Задаем переменные для хранения результата */
    float load=0.;
    float latency=0.;
    int filter_count=0;

    /* Создаем тикер. */
```

```
MSTicker *ticker=ms_ticker_new();
ms_ticker_set_priority(ticker, vars.ticker_priority);

/* Соединяем фильтры в цепочку. */
ms_filter_link(voidsource, 0, dtmfgen, 0);
ms_filter_link(dtmfgen, 0, voidsink, 0);

MSFilter* previous_filter=dtmfgen;
int gain=1;
int i;

printf("# filters load\n");
if (vars.file)
{
    fprintf(vars.file, "# filters load\n");
}

while ((load <= 99.) && (filter_count < vars.limit))
{
    // Временно отключаем "поглотитель" пакетов от схемы.
    ms_filter_unlink(previous_filter, 0, voidsink, 0);
    MSFilter *volume;
    for (i=0; i<vars.step; i++)
    {
        volume=ms_filter_new(MS_VOLUME_ID);
        ms_filter_call_method(volume, MS_VOLUME_SET_DB_GAIN, &
            gain);
        ms_filter_link(previous_filter, 0, volume, 0);
        previous_filter = volume;
    }
    // Возвращаем "поглотитель" пакетов в схему.
    ms_filter_link(volume, 0, voidsink, 0);

    /* Подключаем источник тактов. */
    ms_ticker_attach(ticker,voidsource);

    /* Включаем звуковой генератор. */
    ms_filter_call_method(dtmfgen, MS_DTMF_GEN_PLAY_CUSTOM, (
        void*)&dtmf_cfg);

    /* Даем, время 100 миллисекунд, чтобы были накоплены данные
        для усреднения. */
    ms_usleep(500000);

    /* Читаем результат измерения. */
    load=ms_ticker_get_average_load(ticker);

    filter_count=filter_count + vars.step;

    /* Отключаем источник тактов. */
    ms_ticker_detach(ticker,voidsource);
```

```
        printf("%i␣␣%f\n", filter_count, load);
        if (vars.file) fprintf(vars.file,"%i␣␣%f\n", filter_count,
           load);
    }
    if (vars.file) fclose(vars.file);
}
```

Сохраняем под именем *mstest13.c* и компилируем командой:

```
$ gcc mstest13.c -o mstest13 `pkg-config mediastreamer --libs --
   cflags`
```

Далее запускаем наш инструмент, чтобы оценить нагрузку тикера работающего с наименьшим приоритетом:

```
$ ./mstest13 --step 100 --limit 40000 --tprio 0 -o log0.txt
```

```
$ ./mstest13 --step 100 --limit 40000 --tprio 1 -o log1.txt
```

```
$ ./mstest13 --step 100 --limit 40000 --tprio 2 -o log2.txt
```

Далее "скармливаем" получившиеся файлы *log0.txt*, *log1.txt*, *log2.txt* великолепной утилите *gnuplot*:

```
$ gnuplot -e "set terminal png; set output 'load.png'; plot 'log0.
   txt' using 1:2 with lines , 'log1.txt' using 1:2 with lines, '
   log2.txt' using 1:2 with lines"
```

В результате работы программы будет создан файл *load.png*, в котором будет отрисован график имеющий вид показанный на 7.2.

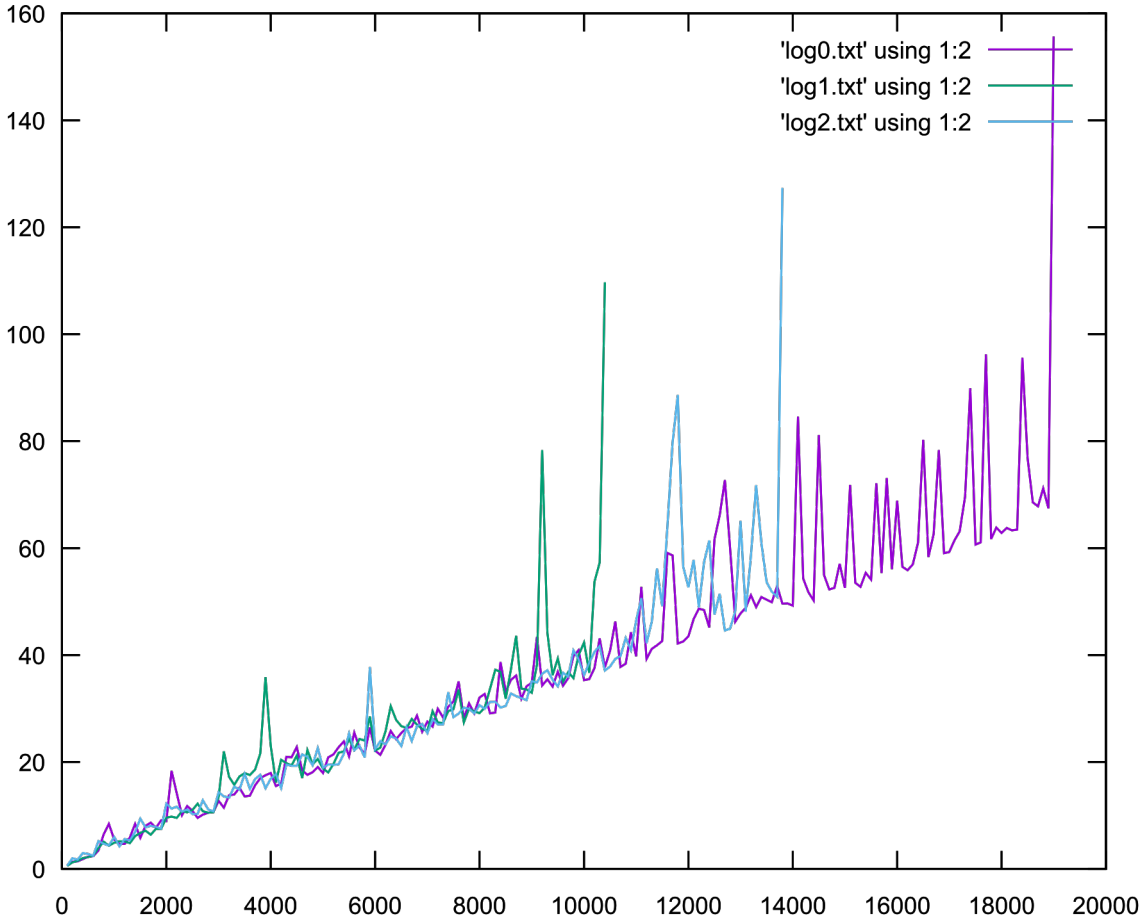


Рис. 7.2 – Нагрузка тикера при разных приоритетах

По вертикали отложена нагрузка тикера в процентах, по горизонтали количество добавленных фильтров нагрузки. На этом графике мы видим, что как и ожидалось, для приоритета 2 (голубая линия), первый заметный выброс наблюдается при подключенных 6000 фильтрах, когда как для приоритетов 0 (фиолетовая) и 1(зеленая) выбросы появляются раньше, при 1000 и 3000 фильтров соответственно.

Для получения усредненных результатов нужно выполнить больше прогонов программы, но уже на этом графике мы видим, что загрузка тикера нарастает линейно, пропорционально количеству фильтров.

7.1.2 Перенос работы в другой тред

Если ваша задача может быть разделена на два и более треда, то тут все просто — создаёте один или больше новых тикеров и подключаете на каждый свой граф фильтров. Если задача не может быть распараллелена, то можно её разделить "поперек", разбив цепочку фильтров на несколько сегментов, каждый из которых будет работать в отдельном треде (т.е. со своим тикером). Далее нужно будет выполнить "сшивку" потоков данных, чтобы

выходные данные первого сегмента попадали на вход следующего. Такой перенос данных между тредами выполняется с помощью двух специальных фильтров MS_ITC_SINK и MS_ITC_SOURCE, они имеют общее название "интертикеры".

Интертикеры

Фильтр MS_ITC_SINK обеспечивает вывод данных из треда он имеет только один вход, выходов у него нет. Фильтр MS_ITC_SOURCE обеспечивает асинхронный ввод данных в тред, он обладает одним выходом, входов не имеет. В терминах медиастримера, эта пара фильтров дает возможность передавать данные между фильтрами работающими от разных тикеров.

Чтобы началась передача данных, эти фильтры нужно соединить, но не так как мы это делали с обычными фильтрами, т.е. функцией *ms_filter_link()*. В данном случае, используется метод MS_ITC_SINK_CONNECT фильтра MS_ITC_SINK:

```
ms_filter_call_method (itc_sink, MS_ITC_SINK_CONNECT, itc_src)
```

Метод связывает два фильтра с помощью асинхронной очереди. Метода для разъединения интертикеров нет.

Пример использования интертикеров

Ниже, на рисунке 7.3, показан пример схемы использования. Тройной стрелкой показана асинхронная очередь между двумя тредами.

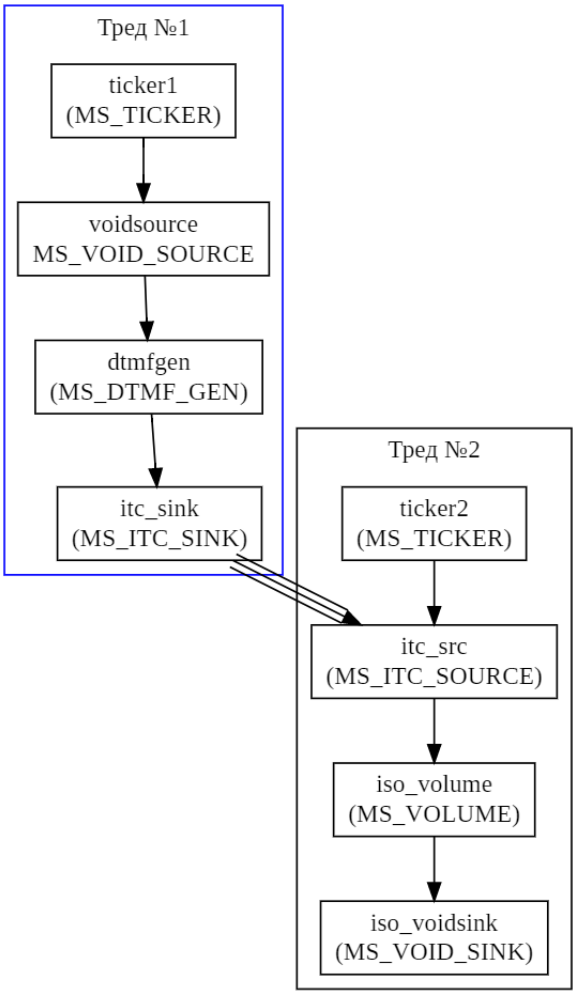


Рис. 7.3 – Применение интертикера

Теперь модифицируем нашу тестовую программу в соответствии с этой схемой так, чтобы в ней как на рисунке работали два треда, связанные интертикерами. В каждом треде будет точка вставки куда будут добавляться новые экземпляров фильтров нагрузки, как показано на рисунке 7.4. Добавляемое количество фильтров будет разделено между тредами ровно попалам.

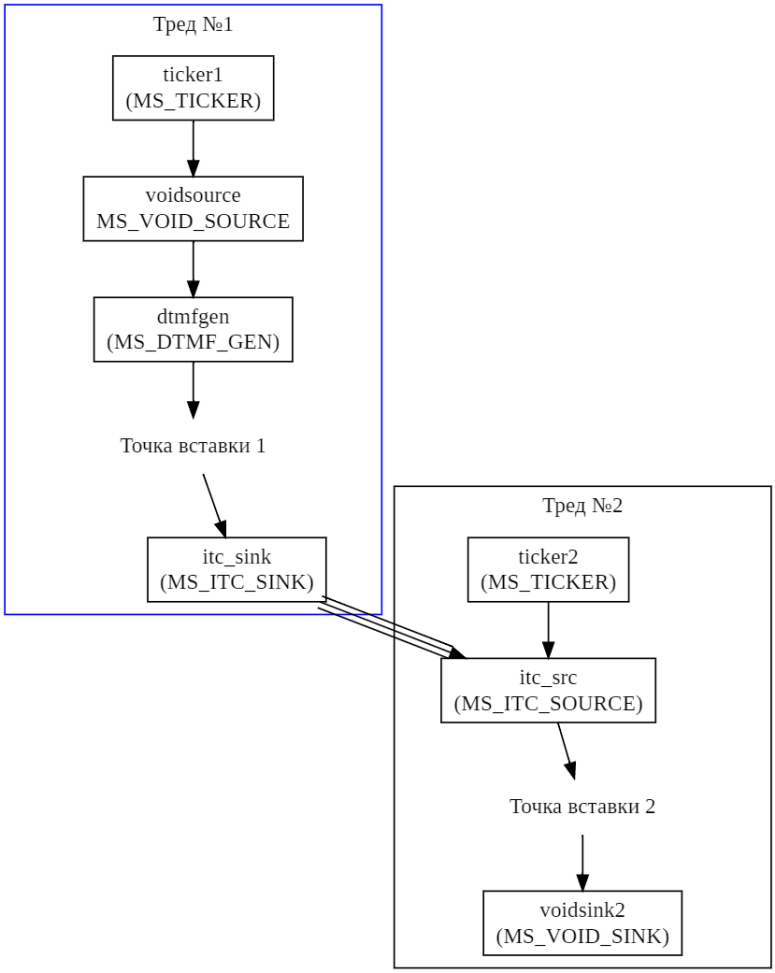


Рис. 7.4 – Разделение нагрузки между двумя тредами

Соответствующий код программы показан в листинге 7.4.

Листинг 7.4: Переменная вычислительная нагрузка с интертикерами

```
/* Файл mstest14.c Переменная вычислительная нагрузка с
   интертикерами. */

#include <stdio.h>
#include <signal.h>

#include <mediastreamer2/msfilter.h>
#include <mediastreamer2/msticker.h>
#include <mediastreamer2/dtmfgen.h>
#include <mediastreamer2/mssndcard.h>
#include <mediastreamer2/msvolume.h>
#include <mediastreamer2/msitc.h>

/*-------------------------------------------------------------*/
struct _app_vars
{
    int   step;                 /* Количество фильтров добавляемых за
       раз. */
```

```
    int  limit;              /* Количество фильтров на котором
       закончить работу. */
    int  ticker_priority;    /* Приоритет тикера. */
    char* file_name;         /* Имя выходного файла. */
    FILE *file;
};

typedef struct _app_vars app_vars;

/*-------------------------------------------------------------------*/
/* Функция преобразования аргументов командной строки в
 * настройки программы. */
void  scan_args(int argc, char *argv[], app_vars *v)
{
    char i;
    for (i=0; i<argc; i++)
    {
        if (!strcmp(argv[i], "--help"))
        {
            char *p=argv[0]; p=p + 2;
            printf("  %s computational load diveded for two threads
               .\n\n", p);
            printf("--help      List of options.\n");
            printf("--version   Version of application.\n");
            printf("--step      Filters count per step.\n");
            printf("--tprio     Ticker priority:\n"
                   "            MS_TICKER_PRIO_NORMAL   0\n"
                   "            MS_TICKER_PRIO_HIGH     1\n"
                   "            MS_TICKER_PRIO_REALTIME 2\n");
            printf("--limit     Filters count limit.\n");
            printf("-o          Output file name.\n");
            exit(0);
        }

        if (!strcmp(argv[i], "--version"))
        {
            printf("0.1\n");
            exit(0);
        }

        if (!strcmp(argv[i], "--step"))
        {
            v->step = atoi(argv[i+1]);
            printf("step: %i\n", v->step);
        }

        if (!strcmp(argv[i], "--tprio"))
        {
            int prio = atoi(argv[i+1]);
            if ((prio >=MS_TICKER_PRIO_NORMAL) && (prio <=
               MS_TICKER_PRIO_REALTIME))
            {
```

```
                v->ticker_priority = atoi(argv[i+1]);
                printf("ticker priority: %i\n", v->ticker_priority);
            }
            else
            {
                printf(" Bad ticker priority: %i\n", prio);
                exit(1);
            }
        }

        if (!strcmp(argv[i], "--limit"))
        {
            v->limit = atoi(argv[i+1]);
            printf("limit: %i\n", v->limit);
        }

        if (!strcmp(argv[i], "-o"))
        {
            v->file_name=argv[i+1];
            printf("file namet: %s\n", v->file_name);
        }
    }
}

/*-------------------------------------------------------------------------*/
/* Структура для хранения настроек программы. */
app_vars vars;

/*-------------------------------------------------------------------------*/
void saveMyData()
{
    // Закрываем файл.
    if (vars.file) fclose(vars.file);
    exit(0);
}

void signalHandler( int signalNumber )
{
    static pthread_once_t semaphore = PTHREAD_ONCE_INIT;
    printf("\nsignal %i received.\n", signalNumber);
    pthread_once( & semaphore, saveMyData );
}

/*-------------------------------------------------------------------------*/
int main(int argc, char *argv[])
{
    /* Устанавливаем настройки по умолчанию. */
    app_vars vars={100, 100500, MS_TICKER_PRIO_NORMAL, 0};

    // Подключаем обработчик Ctrl-C.
    signal( SIGTERM, signalHandler );
    signal( SIGINT,  signalHandler );
```

```
/* Устанавливаем настройки настройки программы в
 * соответствии с аргументами командной строки. */
scan_args(argc, argv, &vars);

if (vars.file_name)
{
    vars.file = fopen(vars.file_name, "w");
}

ms_init();

/* Создаем экземпляры фильтров для первого треда. */
MSFilter  *voidsource = ms_filter_new(MS_VOID_SOURCE_ID);
MSFilter  *dtmfgen    = ms_filter_new(MS_DTMF_GEN_ID);
MSFilter  *itc_sink   = ms_filter_new(MS_ITC_SINK_ID);

MSDtmfGenCustomTone dtmf_cfg;

/* Устанавливаем имя нашего сигнала, помня о том, что в массиве
   мы должны
 * оставить место для нуля, который обозначает конец строки. */
strncpy(dtmf_cfg.tone_name, "busy", sizeof(dtmf_cfg.tone_name));
dtmf_cfg.duration=1000;
dtmf_cfg.frequencies[0]=440; /* Будем генерировать один тон,
   частоту второго тона установим в 0.*/
dtmf_cfg.frequencies[1]=0;
dtmf_cfg.amplitude=1.0; /* Такой амплитуде синуса должен
   соответствовать результат измерения 0.707.*/
dtmf_cfg.interval=0.;
dtmf_cfg.repeat_count=0.;

/* Задаем переменные для хранения результата */
float load=0.;
float latency=0.;
int filter_count=0;

/* Создаем тикер. */
MSTicker *ticker1=ms_ticker_new();
ms_ticker_set_priority(ticker1, vars.ticker_priority);

/* Соединяем фильтры в цепочку. */
ms_filter_link(voidsource, 0, dtmfgen, 0);
ms_filter_link(dtmfgen, 0, itc_sink , 0);

/* Создаем экземпляры фильтров для второго треда. */
MSTicker *ticker2=ms_ticker_new();
ms_ticker_set_priority(ticker2, vars.ticker_priority);
MSFilter *itc_src   = ms_filter_new(MS_ITC_SOURCE_ID);
MSFilter *voidsink2 = ms_filter_new(MS_VOID_SINK_ID);
ms_filter_call_method (itc_sink, MS_ITC_SINK_CONNECT, itc_src);
ms_filter_link(itc_src, 0, voidsink2, 0);
```

```
MSFilter* previous_filter1=dtmfgen;
MSFilter* previous_filter2=itc_src;
int gain=1;
int i;

printf("# filters load\n");
if (vars.file)
{
    fprintf(vars.file, "# filters load\n");
}
while ((load <= 99.) && (filter_count < vars.limit))
{

    // Временно отключаем "поглотители" пакетов от схем.
    ms_filter_unlink(previous_filter1, 0, itc_sink, 0);
    ms_filter_unlink(previous_filter2, 0, voidsink2, 0);
    MSFilter *volume1, *volume2;

    // Делим новые фильтры нагрузки между двумя тредами.
    int new_filters = vars.step>>1;
    for (i=0; i < new_filters; i++)
    {
        volume1=ms_filter_new(MS_VOLUME_ID);
        ms_filter_call_method(volume1, MS_VOLUME_SET_DB_GAIN, &
            gain);
        ms_filter_link(previous_filter1, 0, volume1, 0);
        previous_filter1 = volume1;
    }

    new_filters = vars.step - new_filters;
    for (i=0; i < new_filters; i++)
    {
        volume2=ms_filter_new(MS_VOLUME_ID);
        ms_filter_call_method(volume2, MS_VOLUME_SET_DB_GAIN, &
            gain);
        ms_filter_link(previous_filter2, 0, volume2, 0);
        previous_filter2 = volume2;
    }

    // Возвращаем "поглотители" пакетов в схемы.
    ms_filter_link(volume1, 0, itc_sink, 0);
    ms_filter_link(volume2, 0, voidsink2, 0);

    /* Подключаем источник тактов. */
    ms_ticker_attach(ticker2, itc_src);
    ms_ticker_attach(ticker1, voidsource);

    /* Включаем звуковой генератор. */
    ms_filter_call_method(dtmfgen, MS_DTMF_GEN_PLAY_CUSTOM, (
        void*)&dtmf_cfg);
```

```
            /* Даем, время, чтобы были накоплены данные для усреднения.
               */
            ms_usleep(500000);

            /* Читаем результат измерения. */
            load=ms_ticker_get_average_load(ticker1);

            filter_count=filter_count + vars.step;

            /* Отключаем источник тактов. */
            ms_ticker_detach(ticker1, voidsource);

            printf("%i  %f\n", filter_count, load);
            if (vars.file) fprintf(vars.file,"%i  %f\n", filter_count,
               load);
        }
        if (vars.file) fclose(vars.file);
}
```

Далее компилируем и запускаем нашу программу с тикерами, работающими с наименьшим приоритетом:

```
$ ./mstest14 --step 100 --limit 40000 --tprio 0 -o log4.txt
```

Полученный результат измерений показан на рисунке 7.5

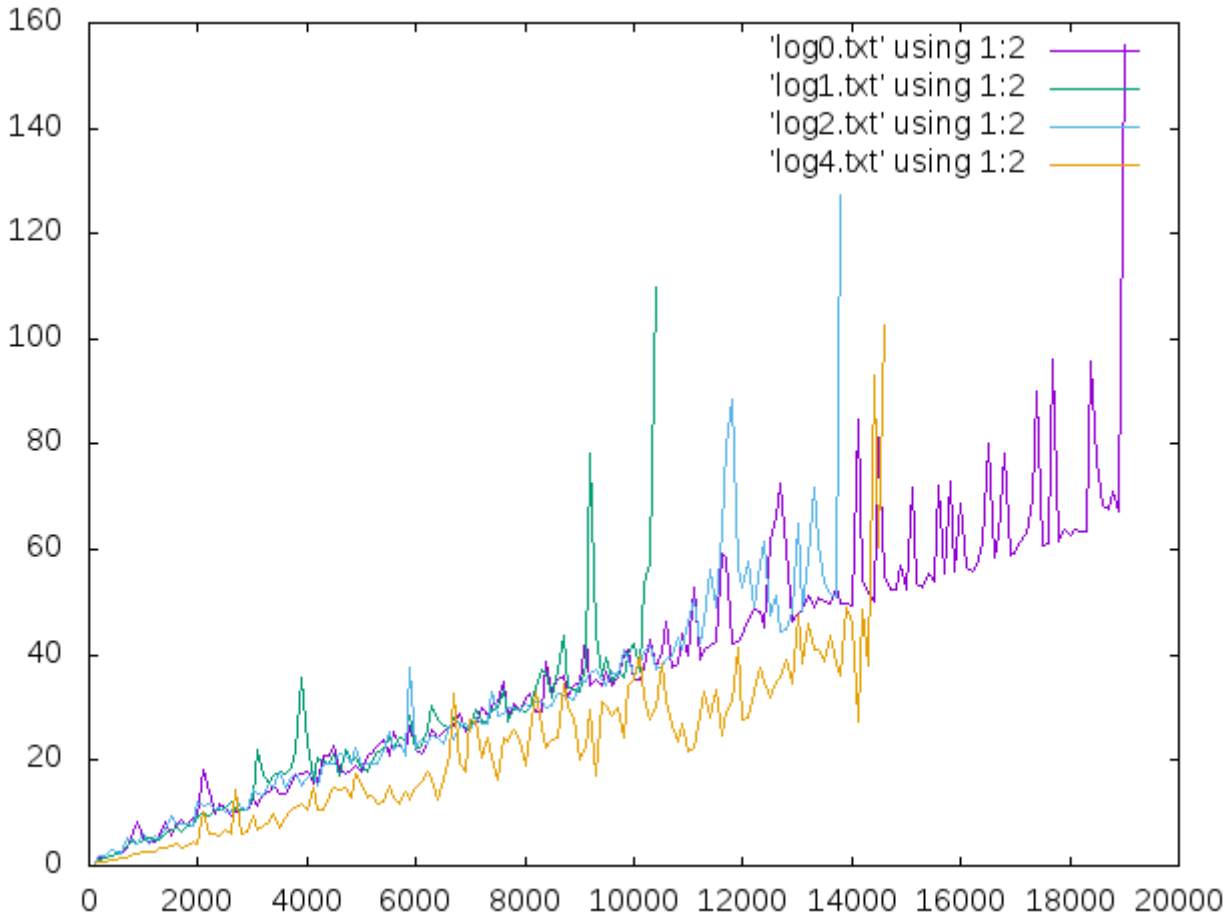


Рис. 7.5 – Нагрузка тикера при использовании дополнительного треда

Для удобства на график добавлены кривые, полученные для первого варианта программы. Оранжевая кривая показывает результат для "двухтредовой" версии программы. Из графика видно, что скорость нарастания загрузки тикера для "двухтредовой" схемы ниже. Нагрузка на второй тикер не показана.

При необходимости можно соединить треды работающие на разных хостах, только при этом вместо интертикеров использовать RTP-сессию (как это мы делали ранее создавая переговорное устройство), здесь также потребуется учесть, что размер RTP-пакетов ограничен сверху величиной MTU.

По результатам этой главы можно сделать вывод, что изменение приоритета тикера не влияет на его производительность, повышение приоритета только снижает вероятность запаздывания тикера. Рецептом к повышению производительности схемы в целом является разделение графа обработки на несколько кластеров со своими индивидуальными тикерами. При этом, платой за снижении нагрузки на тикер является увеличение времени прохождения данных со входа на выход схемы.

Заключение

В этой книге мы осветили лишь часть возможностей и особенностей работы *Mediastreamer2*. Рассматривайте эту книгу как первый шаг к освоению этой технологии.

Предметный указатель

Листинги

Список иллюстраций