Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Ján Veselý

HelenOS Sound Subsystem

Department of Distributed and Dependable Systems

Thesis supervisor: Mgr. Martin Děcký
Study program: Computer Science, Software Systems

2011

I'd like to thank my supervisor, Mgr. Martin Děcký, for his input and guidance during my work on this thesis. I'd also like to thank all HelenOS developers, and especially Jakub Jermář, for their work on making HelenOS the best academic OS platform available. My thanks also go to family and friends who provided moral support.

Contents

Title: HelenOS Sound Subsystem
Author: Ján Veselý
Department: Department of Distributed and Dependable Systems
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail address: martin.decky@d3s.mff.cuni.cz

Abstract: The work examines options for a modern daemon centered audio stack for HelenOS. It studies four different audio architectures; ALSA, OSS, JACK, and PulseAUdio. Each of them implements different approaches to providing general purpose audio support. Champion of every approach is analyzed, its strengths and weaknesses assessed. Based on the results of the analysis, different approaches for HelenOS audio stack are examined and the most promising one implemented. Complete audio stack is implemented, including an audio device driver, and a demonstrator audio application. Direction of future work and improvements is discussed at the end.

Keywords: audio, helenos, sound

Název práce: HelenOS sound subsystem
Autor: Ján Veselý
Katedra (ústav): Katedra distribuovaných a spolehlivých systému
Vedoucí bakalářské práce: Mgr. Martin Děcký
e-mail vedoucího: martin.decky@d3s.mff.cuni.cz

Abstrakt: Práca rozoberá možnosti implementácie moderného audio systému pre HelenOS. Boli vybrané štyri existujúce audio architektúry; ALSA, OSS, JACK a PulseAudio. Každá študovaná architektúra pristupuje k poskytovaniu audio funkcionlity iným spôsobom. Silné a slabé stránky každej implementácie sú analyzované a na základe výsledkov tejto analýzy sú predložené možné riešenia. Najvhodnejšie z týchto riešení je v práci implementované. Implementácia zahŕňa všetky súčasti audio subsystému, od ovládača audio zariadenia, až po jednoduchú audio aplikáciu, ktorá demonštruje možnosti implementovaného prístupu. Na záver sú načrtnuté možnosti ďalšieho rozśírenia práce.

Klíčová slova: audio, helenos, zvuk

Chapter 1

Introduction

1.1 Motivation

There has been a change in the recent years in the way audio stack is designed on modern operating systems. In the past applications were in control of sound cards, whether it was a music player, movie player, or a web browser, each provided its own audio controls and accessed the audio driver directly. This approach has changed, major operating systems started to add an audio routing service, that provides improved user control and better support for multiple audio devices.

This new approach plays into areas that are strong points of micro-kernel design, namely a high performance IPC and address space separation of different tasks. Thus the main motivation is to explore and confirm the suitability of this new approach to micro-kernel multi-service environment.

1.2 Goals and Aims

The main aim of this work is to bring modern audio stack to HelenOS. All levels need to be designed and implemented. Different existing solutions shall be examined and used approaches considered. Both device driver interface and application audio interface shall be designed and implemented.

Functionality shall be demonstrated on Sound Blaster 16 hardware in Qemu virtual environment, or hardware if available. The work shall also include a suitable demonstrator application that uses designed interfaces to playback audio. As a secondary goal, existing USB support shall be improved to include support for isochronous transfers, and a driver for a USB audio devices implemented.

1.3 Existing support in HelenOS

AS of the beginning of this work there is no audio support implemented in the HelenOS operating system. There is, however, an exiting Device Driver Framework and drivers for several PCI connected device are implemented. HelenOS also includes a USB1.1 stack that implements support for control, bulk, and interrupt transfers. There is no support for isochronous USB transfers.

HelenOS memory management enables tasks to request continuous areas of physical memory, but other limitations like alignment or pointer size are not supported. The IPC routines support short messages, bulk data transfers and shared memory areas. There is also no support for high precision timers.

Existing limitations should be either implemented or worked around, based on complexity of such work. They should not impact design decisions in this work.

Chapter 2

Digitized Audio

2.1 Linear PCM

Digital audio may be represented in many forms, one of the simpler ones is Pulse Coded Modulation (PCM). Signal levels are recorded at a given rate and stored. To play the sound the stored values are converted back to signal levels. This direct technique is referred to as Linear PCM, there are other ways to store the same information, such as storing only differences between sample values called Differential PCM. Only Linear PCM will be considered in this work. The are several parameters that define Linear PCM format. [13]

The first one is *sampling rate*, it's easy to guess that this parameter sets number of samples recorded per second. The common values are 8,000 Hz for phone calls, 44,100 Hz used by audio CDs, and 48,000 Hz used by most modern devices. High quality audio storage, like DVD audio, use 96 or 192 kHz [16].

According to the *Nyquist–Shannon sampling theorem* sampling rate double the highest recorded frequency is necessary to completely determine the recorded frequencies. [10]. Given the highest audible frequency for humans is 20kHz [4] it is sufficient to record audio at 40kHz. The most common frequencies of 44.1 kHz and 48 kHz are well above this threshold.

The second parameter is *depth*. Depth sets the range of values in one sample. A more generic way is to set *sample format*. Samples can be recorded as signed or unsigned integers, stored in big or little endian, or even floats. The most common values are 8 bit unsigned, 16 bit signed, 24bit signed, or 32bit float (signed usually -1.0 - 1.0). [13]

The third parameter sets the *number and order of channels*. Unless each channel uses separate data stream we need to know number of channels encoded in one stream as well as their respective positions in the byte stream. Mono audio is used only for voice recording, multimedia use two or more channels. The common values are 3 (2.1), 6 (5.1), and 8 (7.1). This parameter sets not only the total number of channels but also their position in a byte stream. This information is important for 3 dimensional sound as well as to use play correct samples on specialized devices, like sub-woofer for low frequency sounds. Storing all samples in predefined order is the most direct approach. More complicated formats, like DVD audio, store least significant bytes separately. [13] A set of samples for all channels is usually called a *frame*.

Converting from one linear PCM format to another is rather straightforward and usually consists of just converting from one numeric representation to another, i.e. integer to float. It may occasionally include endian change or range shift if only one of the formats is signed. Converting to a format that includes different number of channels is more interesting and leaves room for policy decisions. Should the missing channels be silent? Should they be recreated? Such techniques are used to provide 3D sound illusion out of stereo samples. The most complex part is audio resampling. In few cases where one rate is a divisor of the other, the implementation can be simple arithmetic mean. However, resampling rates that are close to each other requires advanced algorithms that reconstruct the original sound.

Chapter 3

Existing solutions

Audio stack usually consists of these components *audio application*, *audio client library*, *audio daemon*, *audio driver library*, and an *audio driver*. Figure 3.1 displays these layers, including a kernel space boundary a typical means of communication between these layers.

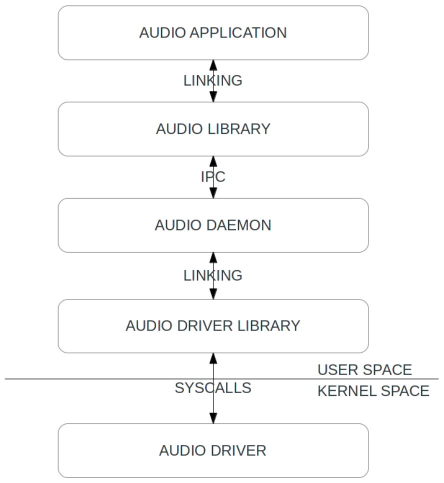


Figure 3.1: Audio stack layers

Not every layer needs to be present in an audio stack implementation and the work done at every layer might be different as well. Four different architectures were examined. From more than a decade old OSS, to relatively recent PulseAudio, they all use different approach to sound playback and recording as well as device enumeration and control. These architectures each represent a category that takes a different approach to providing audio capabilities to applications.

OSS Open Sound System is a traditional sound API for Unix and Unix-like systems. It represents a *"driver knows it all"* category, and defines kernel interfaces for playback/record and device manipulation. OSS provides an application interface and hides hardware limitations in driver implementation. The latest version is 4.1.

ALSA Advanced Linux Sound Architecture is a Linux specific replacement for OSS. It consists of device abstraction level implemented in the Linux kernel and a client userspace library that uses plugin framework to provide additional functionality. ALSA client library is used to convert audio playback API to audio device API. It is similar to audio daemon approach but keeps the 'daemon' part integrated in audio applications in the form of a library. Other implementations in this *"simple drivers, capable library"* category include for example CoreAudio [3], which provides complete OpenAL implementation in its audio stack.

JACK JACK Audio Connection Kit is a multiplatfrom professional sound server. It focuses on low latency, and client synchronization. Among the examined solutions JACK is in its own category. It consists of advanced routing and mixing daemon and a client library. Unlike ALSA or OSS, it does not include device drivers and uses system provided audio backends.

PulseAudio PulseAudio is a relatively recent attempt to improve sound situation in Linux. Previous work in the category of *"generic audio server"* for Linux, like aRts and ESD, failed to gain significant adoption. PulseAudio strives to replace them to the point of providing ESD protocol implementation for ESD enabled applications. It provides a sound routing server, with features like per application volume control and runtime output selection. Although it was designed for Linux systems it is a multiplatform software that can be used on other operating systems, and uses OS specific access to audio hardware.

3.1 Open Sound System (OSS)

Open Sound System is the sound API of choice for many UNIX like systems. It has existed for more than a decade and the current version is 4.1. OSS is not system specific, but persists mainly in the BSD world. OSS implements all of its functionality in kernel space and uses syscalls for communication. Figure 3.2 shows at only the bottom layer of the audio stack is present in an OSS based audio implementation.

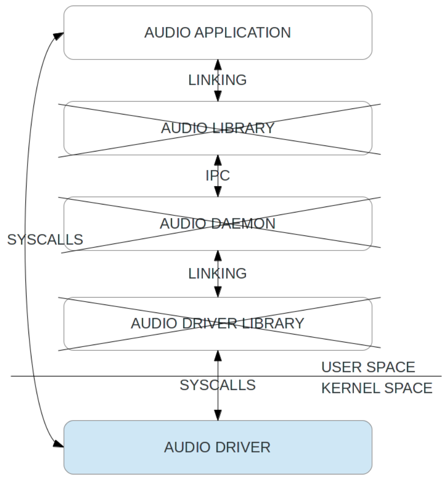


Figure 3.2: OSS audio stack

Implemented in Kernel

The advantage of using sytem call as programmers' level interface has the advantage of not having a client library, that can bring higher level of compatibility. The OSS API is based on audio playback instead of presenting and controlling every hardware feature. Format conversion, resampling, and mixing are all done on the device driver's side. According to the developers it creates almost virtual environment for audio applications. If there are new hardware features added in the future they will be supported without any changes in applications. It is for these reasons that OSS developers consider OSS 4 to be the ultimate audio API.[12, oss api].

On the other hand kernel implementation has many disadvantages. There is absolutely no reason why audio mixing and routing should run with elevated privileges of superuser (or equivalent) mode. It makes kernel code unnecessary complicated and the consequences of bugs in this code are much more severe than in userspace equivalent. The same almost virtual environment, the designers talk about, can be created using library calls instead of syscalls with almost no difference. Stable application level audio API do not need to rely on the syscall interface.

Playback/Record API

Similar to other sound APIs, OSS API can be divided to several categories. There is Device enumeration and configuration API, Audio API (for playback and recording), Mixer API, and MIDI API. OSS Developers claim that OSS API's design goal was aggressive simplicity and ease of use. [12].

OSS audio playback API is similar to other playback APIs. The application needs to open and configure the chosen device and then read or write data. Although OSS supports direct access to device buffer via *mmap* syscall, the developers discourage using it. They claim that "in current workstations use of mmap may make the application 0.01% faster than with normal read/write method. It doesn't have any effect on latencies." [12, mmap]. This makes sense for generic applications and in fact PulseAudio Simple API is similar to the OSS suggested interface. PulseAudio even provides wrapper library that translates OSS calls to PulseAudio interface.

Mixer API

Selecting audio device for playback or capture or setting volume levels are separate syscalls. The mixer interface is rather basic and includes playback/capture devices, selecting active interface and querying available system information.

3.2 Advanced Linux Sound Architecture

Advanced Linux Sound Architecture (ALSA) implements a model that is different to the one used by OSS. ALSA device drivers are relatively simple and provide device abstraction either via a shared device buffer or by implementing read/write pipe interface. The brunt of the work is performed by its audio driver library *alsalib* that uses a complex plugin based interface to provide applications with mixing, conversion a resampling functionality. Advanced Linux Sound Architecture replaced OSS in the Linux kernel. It first appeared in kernel 2.6 and offers a compatibility layer that enables legacy OSS applications to run on top of ALSA.

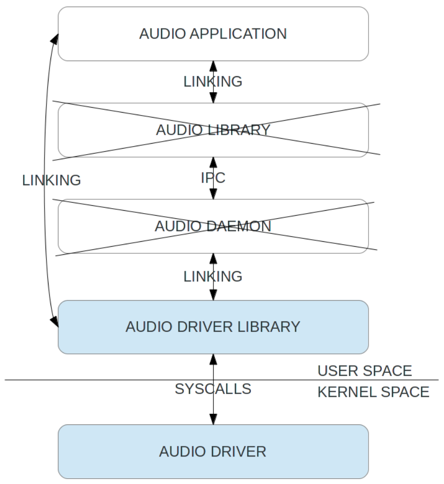


Figure 3.3: ALSA audio stack

Device Abstraction

ALSA represents audio hardware on three levels, *card*, *device*, and *subdevice*. A *card* is a physical hardware connected via a system bus, usually a PCI bus. *Device* is an independent function provided by the card, and a *subdevices* provide inter-dependent functionality. An example might be a card that is able to play one stream with several audio outputs that can't provide independent playback but can be be turned on and off, or have volume changed independently. *Cards*, *devices*, and *subdevices* are represented by numbers. [1] Filenames that represent devices are created using this numbering convention. Thus a file name *pcmC1D3p* represents the fourth device on the second audio card present in the system, similarly *pcmC1D3c* would be a corresponding capture interface.

These devices cannot be accessed directly, ALSA provides various modules to do the job instead. A module with parameters creates an *ALSA device*, that can be used for playback or recording. One of these modules is the *hw* module. It enables pcm playback/record on specified *card* and *device*, There are other plugins that provide format conversion, copying, mixing, introduce pseudo devices, and other functionality.

ALSA audio capabilities are grouped in several sets of API functions. PCM API for pulse code modulation digital playback and recording, Control API for hardware device control, MIDI API for musical instruments or MIDI emulation, and sequencer/timer API for exposing timer functionality present on audio hardware. Figure 3.4 is an ALSA architecture overview by one of the maintainers, Takashi Iwai.

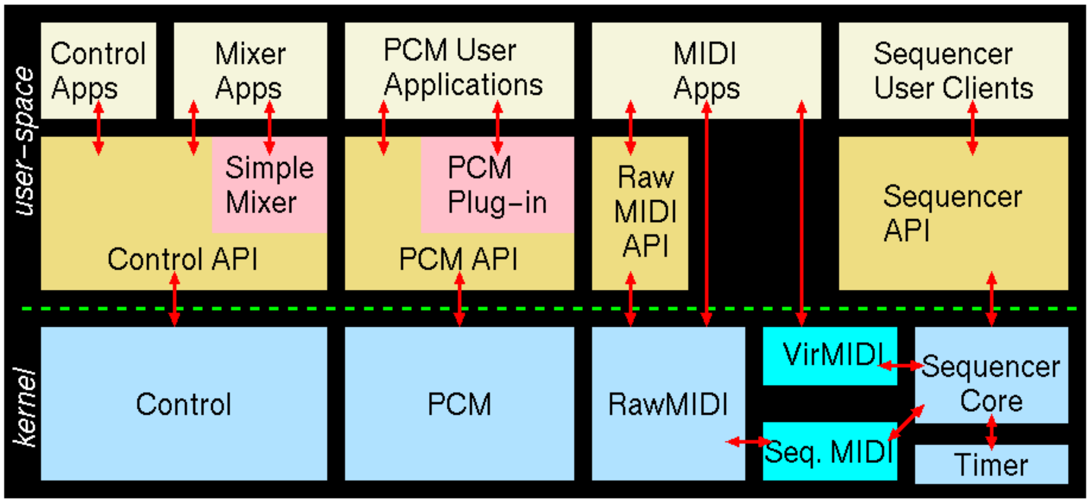


Figure 3.4: ALSA API Overview [19]

PCM Configuration Space

One of the things that ALSA provides is format negotiation. PCM audio data have many parameters that define it. These include sample rate, number and ordering of channels, sample size and format. Samples might be stored as signed/unsigned, integers or floats, using different type width and byte ordering. Moreover, devices may support custom audio formats. ALSA needs to make sure that the data to be played/recorded are in the format requested by the application, and refuse requests or provide conversion if not.

The problem is that devices have not only limited range of every parameter, but some combinations of individually valid parameters don't work. To help with this process ALSA introduced Configuration Spaces. Every parameter is one dimension in the Configuration Space and the set of all valid combinations is a multidimensional object in this space. ALSA API forces the application to setup a configuration that complies with these limitations before passing it to the device.[1]

Class *snd_pcm_hw_params_t* and associated functions are responsible for setting device parameters. Functions like *snd_pcm_hw_params_set_rate_near* help the application choose the best suitable parameters and *snd_pcm_hw_params_test_rate* make sure that those parameters are valid for the underlying device.

While most audio devices support block mode transfer for playback and recording, modern devices have buffers allocated in system memory and thus accessible to applications. ALSA may be setup to use both blocking and non-blocking transfers. The stream status might be polled and data written/read directly, alternatively callbacks to provide/store data are available. Devices that have their buffers in system memory can be accessed by mapping their buffers to the application address space and the data accessed directly.

One way to access memory mapped buffer is to use *snd_pcm_mmap_begin*, this function will provide pointers and offsets to a memory buffer. The buffer is returned to the device by calling *snd_pcm_mmap_commit*, after all data modifications has been made. The other way is to use *snd_pcm_mmap_writei* and co. These wrappers can be used in the same stream oriented way as their non mmap counterparts.

Control API

The control API is again relatively simple. Control settings need to be loaded after opening a device file. The loaded structure includes a list of elements that can be configured, and their respective values, that can be read or set.

3.3 PulseAudio

PulseAudio is an audio server and API used primarily in Linux, but it works on other operating systems like Windows(c) and MacOS(c). Application's sound output or input is processed by the PulseAudio daemon and then forwarded to one or more devices. Sound processing may include resampling, mixing, and volume and channel mapping manipulation. PulseAudio uses audio driver API provided the operating system. It can also provide a virtual device for network streaming. PulseAudio implements the upper layers of audio stack, see Figure 3.5.

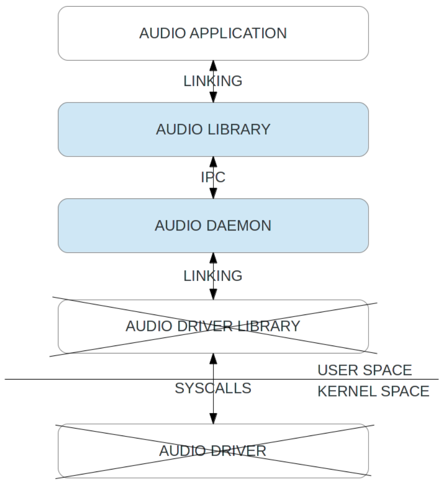


Figure 3.5: PulseAudio stack

PulseAudio provides two audio APIs, a Simple API, and an Asynchronous API. It also provides an API for controlling the daemon. [15]

Simple API

The Simple API provides simplified blocking pipe-like interface for audio playback and recording. This interface is straightforward and easy to use, but a developer is limited to a single pa_simple object, and one connection per application. It is aimed at simple playback/record applications and removes the burden of unnecessary over-configuration by relying on default settings. Adding playback to an application using this interface is a matter of minutes.

Asynchronous API

The Asynchronous API provides advanced stream control, latency control, server events, and a sample cache. As it name suggests, it is an asynchronous interface that uses callbacks and event loop to operate. There are 3 kinds of event loops offered by the PulseAudio client library, pa_main_loop, pa_threaded_main_loop, and pa_glib_main_loop.

pa_main_loop is a standard event loop, it handles server and client events. Once a thread enters the pa_main_loop_run() function it won't exit until a call to pa_main_loop_quit() is made. pa_threaded_main_loop uses a separate thread to process events, and pa_glib_main provides bindings for glib event loop.

At the bottom of the communication hierarchy is the pa_context class. It provides connection to the server, and no more than one instance, per connected server, should be necessary in one application. [15] pa_context class handles all the low level IPC necessary for communication with a PulseAudio server, and it provides some level of control over the PulseAudio daemon instance.

Audio data are transferred using pa_stream objects. Each pa_stream is connected to pa_context that it uses for data transfer. Streams can be used for playback, recording, or sample upload, and are highly configurable. Streams have configurable data format, buffer settings, and required latency, but also stream metadata like media author and title. It is possible to use either polling mode or callbacks for stream data read/write, or even combine the two methods. [15] Uploading samples to sample cache is a functionality that enables clients to upload audio data to the server, so that they are readily available for repeated, low latency, playback.

Control API

Controlling PulseAudio is different from controlling audio devices. The daemon and application connections can be tweaked in a number of ways. Applications can change the size of server side buffers to match their latency need. Stream volume and channel map within frames can be modified, and the server even enables applications to upload audio samples to be readily available without additional data transfer. PulseAudio however limits connecting of application sink and sources to the global ones. [15] Most notably PulseAudio does not take the preferred source or sink as an argument to connection function, but relies on static settings in the *client.conf* file, this can be overridden by an environment variable. [14]

3.4 JACK Audio Connection Kit

JACK Audio Connection Kit is a sound server that aims to reduce latency to the minimum. It is intended for professional use in media content creation. JACK uses kernel realtime extensions to achieve its goal of minimal latency. Like PulseAudio it relies on system provided audio backends to access audio hardware, and implements the two upper layers of audio stack model.

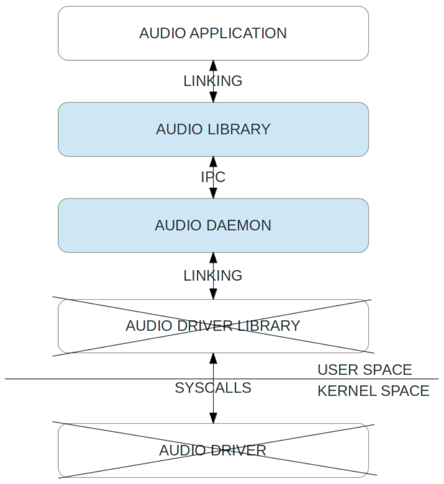


Figure 3.6: JACK audio stack

There are several characteristics that make JACK architecture stand up among audio server implementations.

JACK does not do format conversion This includes change of sample format as well as sampling rate. JACK defaults to 32bit float mono sound representation (any other needs to specified at startup time). This means that applications need to do conversions themselves and can't rely on the audio stack to do it. Conversions required by audio hardware are left to the audio driver library.

It is not possible to send data to JACK daemon JACK events are triggered by the daemon. Thus any application providing data needs to wait for data request. Requests are timed and clients that fail to respond are dropped.

JACK routes individual channels JACK routing works on individual channels instead of multichannel streams. Its support is limited to non-interleaved data format. Applications have to do any interleaving, or deinterleaving, themselves. The only thing JACK provides for clients is mixing.

Creating simple JACK client is simple. The application needs to connect to the server and register a port. The port can be either source or consumer of data. Every event is handled via callbacks. Callbacks exist for data requests, errors, and server side changes. After the necessary callbacks are registered client is activated to signal it's readiness to produce or receive data. Active ports can be connected to other sources/sinks, either by the application, or via external JACK daemon control utility.

JACK's focus is on minimizing latency and on client synchronization. These features are best used for mixing sound from different sources like files, and different musical instruments. The choice of high quality sample format for internal representation and channel based routing works well in such setups. The only case in which format conversion or resampling that needs to be performed is if the audio hardware is not capable of playing/recording audio in the default format.

On the other hand, the lack of conversion and high complexity of channel based routing make JACK unsuitable for general use. Defaulting to 32bit float representation may lead to unnecessary conversions, and popular sources like mp3 files or Internet audio rarely reach quality that would justify using that sample format.

Control API

JACK daemon works on audio ports, and thus most of its settings set or list available ports. It is even possible to use regular expressions to get a port list. Ports can be aliased, tied and of course connected to and disconnected from other ports. Ports can also set their server side buffer size. One interesting feature enables JACK daemon to drop realtime scheduling and work 'faster than realtime'. [8]

Chapter 4

Audio Device Driver Interface

4.1 Requirements

Audio device driver interface shall provide easy access to the common set of features across widely used general purpose audio hardware. It shall allow lightweight implementations of audio drivers that provide basic playback and record functionality. On the other hand the audio playback interface shall allow easy and efficient implementation of applications that use continuous as well as event driven playback.

4.2 Device Capabilities and Driver Design

The basic functionality of every audio device is the ability to playback and/or record audio. Device hardware converts internal, digital representation to external one, usually analog. Some devices can do audio mixing. This means that they can read audio data from multiple locations and produce single output. Number of supported streams is limited and unless the mixing is done after digital to analog conversion this method does not produce higher quality audio and serves mainly as a hardware offload. Modern devices provide separate paths for input and output allowing full duplex audio.

Modern computers are perfectly capable of doing almost all operations on digital audio in software and thus the simplest and most widespread audio devices are a little more than just configurable DACs/ADCs.

The way of implementing audio streams needs to be considered as well. Devices that are connected internally, usually via a PCI or an ISA bus, have access to the system memory and can use system memory buffers. Other devices, mostly those that are connected via USB might implement their own buffers in addition to those accessed by host bus adapters.

The type of information a device provides about its buffers needs to considered as well. Traditional way of using audio buffers was to setup a cyclic buffer and let the device fire an interrupt after every N frames of playback or capture. In this case the information about position in buffer is not available and the driver is only informed when certain points are reached. However, it is beneficial if a device is able to provide position in playback/record buffer at any time. It not only avoids unnecessary interrupts, but also enables lower latency for immediate sound output. On the other hand, precise event timers are required to schedule buffer update actions.

Based on the outlined hardware capabilities HelenOS audio device drivers shall implement one or more audio stream interfaces, based on the capabilities of the controlled device, and register them to the location service. Stream interface has to satisfy these conditions:

- A stream is independent from all other streams, similar to ALSA hardware devices. It means that streams must be able to output different audio format, or be switched between capture/playback without affecting other streams. Note that this condition does not say anything about actual sound output as experienced by the user. Multiple streams may mix the audio and share one output, but mixing has to done on hardware side.
- A stream provides cyclic buffer to be filled with audio data. Size of the buffer is selected by the driver based on the hardware and environment limitations.
- A stream exports set of capabilities it supports. These capabilities include playback, capture, immediate buffer position, and ability to use fragments (interrupt frequency). The granularity of buffer position information, as well as limitations on interrupt frequency are decided by the driver. Stream supports either playback, or capture, or both. Interface providing immediate buffer position should be preferred, using interrupts should be considered if implementing the former approach is not possible or reasonable.
- A stream has to signal events related to the playback/capture

- A stream has to support at least play and stop commands. The play command will start playback/capture using the provided buffer and will continue cycling the buffer until the stop command is issued or an error occurs. The stop command will stop playback/capture immediately, it also resets current buffer position to the beginning of the buffer. If the stream uses events to report produced or consumed fragments it should support stop command that completes the active fragment before taking effect.

The interface for device control is simple, the driver will provide a list of supported properties using an identifier, a human readable name and an integer range that starts at 0. It will allow clients to get and set property values. The driver is responsible for checking that the new value is within the valid range. This interface is similar to USB Audio control mechanism, the only difference is that USB Audio includes resolution property. This is redundant, the range can be easily interpolated to desired values in a utility program without introducing additional values to driver interface. Moreover, USB audio devices silently change settings of invalid values that are within supported range. Dropping the resolution property means that every value in the supported range is valid. This simple and powerful interface allows user to fully control the hardware without introducing complicated schemes to include all possible hardware features.

Almost all interface functions can be implemented using simple message passing IPC. Passing string control identifiers requires data transfers of arbitrary length. The single more demanding part is sharing of stream buffers, these buffers might be often tied to the specific physical address location in the main memory, and/or require special memory attributes, like disabled caching. IPC must be able to share memory areas without moving them from their current location or changing memory attributes.

4.3 Sound Blaster 16 driver

Sound Blaster 16 was introduced in 1992 [17] and is considered ancient by current standards. However, in its time it was so popular that even later PCI devices included sb16 emulation for backwards compatibility.

The original Sound Blaster 16 connects to the computer via ISA bus, although there was a PCI version of SB16 [17] it used different chip and it is not considered in this work. Devices using ISA bus are limited in certain ways, the most important one is that these devices do not have their own DMA controller and instead use the one provided by the host system, this is usually intel 8237 or compatible. SB16 supports 16 bit samples and uses 16 bit DMA transfers. The most common configuration is to use DMA channel 5 for 16 bit DMA transfers and DMA channel 1 for 8 bit transfers.

Sound blaster 16 consists of two internal parts, a mixer and a DSP. The mixer uses CT1745 chip and handles volume levels. These are easily exported via the proposed control interface. SB mixer also includes switches that control which sources are mixed into the output channels. It is possible to mix together audio cd, microphone input, and digitized audio without any software mixing involved. Although this feature might look useful, in reality it isn't. Hardware mixing of specific sources is rarely used, cd player would have to be connected using special audio cable, and the usefulness of microphone amplification is limited. Thus this functionality will not be provided by HelenOS sb16 driver.

The other, and arguably more important, part of Sound Blaster 16 cards is the DSP. DSPv4 that is used on SB16 cards is controlled via a set of commands. These commands start and stop playback or capture, and set audio and DMA transfer parameters. To start a playback or capture the driver needs to setup DMA controller before issuing a command to the DSP. It is possible to query the DMA controller on the current read/write position in the memory buffer. The DSP is capable of issuing interrupts after a playback/capture commands has been completed, even if it was immediately restarted. Thus both buffer position and regular interrupts are available and supported by HelenOS sb16 driver.

Although the Sound Blaster Series Hardware Programming Guide [18] does not mention duplex modes at all, it is possible to configure a SB16 device for full duplex operation. If capture uses 8bit samples and 8bit DMA channel, and playback uses 16bit samples and 16bit DMA channel, or vice-versa, both operations are possible at the same time. This feature is undocumented and probably a side-effect of the DSP design, the driver in Linux kernel supports this mode. HelenOS SB16 driver will not support 8bit transfers at all so this feature will not be available. Given the undocumented nature, more testing on real hardware needs to be done to guarantee consistent behavior.

4.4 USB Audio devices

Most audio devices connect to the host machine via an internal bus like PCI or ISA, or their respective variants. These devices enjoy benefits of internal communication like direct memory access. On the other side, there is an entire class of USB audio devices that rely on USB host controller to provide or receive audio data. Addition of an additional bus to data path adds additional overhead. In the case of USB attached devices it means that audio data need to be prepared one USB frame before data transfer and can be played in the next frame after the transfer. USB uses 1ms frames so this transfer adds additional 2ms to the playback latency time. This is well within the 'lip sync' requirement of 40ms [12, mmap], but drivers should be aware of this.

USB Device Class Definition for Audio Devices does not define a single device or a family of similar devices, it rather presents tools and means based on USB standard that enable implementors to construct audio devices that are recognized by standard USB audio software. Audio class devices use class specific descriptors to inform drivers about inner organization of device's programmable units and convey the position of isochronous endpoints in this organization.

Programmable units are controlled via interface that is similar to the proposed audio control interface, it uses four variables *min,* max, *cur*, and *res* to get and set programmable values. While most of these values would be exported and controlled by users, some need to be adjusted by device drivers. USB Audio class descriptors include pointers to other interfaces that are supposed to control specific settings. USB Audio control interface also includes an optional support for device events via an Interrupt pipe. It may be used to signal changes to the audio device that did not originate from the driver.

USB Audio class mandates that one audio function is organized into an Audio Interface Collection with single control interface and zero or more audio streaming interfaces.[21, page 29]. Audio Class Definition also includes MIDI streaming interface in a separate document, but MIDI playback is out of the scope of this work and it was not considered. After the driver parsed all the descriptors and is aware of the device's structure it needs to configure USB pipes to transport data to/from the device. Isochronous audio pipes support multitude of audio formats that are described in *Universal Serial Bus Device Class Definition for Audio Data Formats.* These formats range from simple PCM to advanced formats like MPEG or AC-3 [20]. Desired format is selected using the audio streaming interface alternate setting.

It can be seen that a generic driver that would be able to control most of the USB Audio devices would be extremely complex. Although it is possible to implement an audio interface that uses playback events on top of the current USB stack, this approach is very suboptimal. USB audio driver would have to provide fake buffer and regularly copy and send data via isochronous USB pipe. A much better way is to export a memory buffer that can be directly accessed by USB host controller hardware and use USB frame count to provide buffer position. The relationship between a USB audio device and a USB Host controller would be very similar to one between SB16 and an ISA DMA controller.

Thus a decision was made to not implement USB audio driver for HelenOS and focus the work on improving the USB stack by removing unnecessary overhead, and adding new features. This will make the future work on USB audio drivers easier, and implementation of the preferred audio interface cleaner.

4.5 Intel High Definition Audio Codecs

One of the most widespread modern general audio cards are implementations of Intel *High Definition Audio*. Intel HDA specification divides audio hardware into two parts. The first part covers device enumeration and host interface, it defines controller's register interface and mandates bus mastering DMA capability for all devices. Audio buffers are defined via a list of buffer descriptors. Each of these descriptors points to a continuous part of an audio buffer. It is possible to setup the device to issue interrupt after every completed part, but Intel HDA devices provide immediate buffer position as well. These controls make it a nice fit for the proposed audio stream interface. Data read from main memory are fed to internal fifo queues and transported via internal time multiplex '*High Definition Audio Link*' to Codecs. [5]

Codecs are described in the other part of *Intel HDA Sepcification*. Similar to USB audio, Codec specification does not define a single device, but provides means and tools for implementors to create their own devices and describe them in standard way. Codecs may include mixers, selectors, amplifiers, digital-analog converters, pin controllers for jack detection, and even GPIO. Multiple codecs can receive data from a single playback stream, but need separate streams for capture. Driver software is supposed to parse Codec's description to understand the codec's internal structure.

To achieve this level of complexity Intel DHA defines a communication protocol over the HDA Link, this protocol not only transports audio data to/from codecs but it also includes commands for enumerating codec capabilities and changing settings. Sending and receiving control data is supported via command ring buffers on the controller side. This approach again makes an excellent fit for the proposed device control interface.

4.6 HelenOS Audio Device support

The level of similarity between Intel HDA and USB Audio is uncanny. Both separate host controllers, providing DMA data source and sink, from actual audio devices. The actual audio devices are not explicitly defined, but rather a set of tools is provided for creating a wide range of devices. Intel HDA even supports Codec hotplug, and the definition of optional 'unsolicited events' is very similar to the way USB audio handles external events. Moreover, Intel HDA mandates all devices to include a wall clock counter, this is roughly equivalent to USB frame counter found on USB host controllers.

Even more surprising is the fact that the ancient Sound Blaster 16 design is similar to the modern audio hardware designs. It includes a digital DMA part in the form of ISA DMA controller, and an output handling analog part that controls the way audio is recorded or reproduced. The fact that SB16 manual includes exact definition of the DSP and analog output parts makes sb16 driver a good candidate for proof of concept implementation of the proposed interfaces, despite the limitations of ISA DMA engine.

Chapter 5

HelenOS Audio Service Design

5.1 Why Use Audio Daemon

Applications traditionally used audio devices directly. An application would claim an audio device and use its capabilities to output or record audio. Device sharing had to be implemented either at the driver level (OSS) or in an audio library (ALSA). This approach worked reasonably well when there was a single dominant audio user, like a music or a video player, although it has its set of problems. Misbehaving application could occupy the audio device indefinitely and effectively prevent all other applications from using it. Not all applications provide volume control, forcing users to manually adjust global volume levels every time an application used audio. Using multiple audio devices was rather cumbersome, most applications would just use the default audio device, and there was no way to change this once the application started.

From the user perspective using an audio daemon solves all these problems. Settings exported by the daemon enable users to select volume levels on per application, or per context basis, and it allows dynamic audio input or output switching. Using an audio daemon, it should be possible to route movie player to HDMI audio output, voip conversation to a bluetooth headset, and music playback to a speaker set, all at the same time. Although this example is a bit extreme and few users require it, there is no technological limit why it should not be possible.

From a software design perspective using audio daemon enables both device drivers and audio applications to remain simple, and leave most of the tasks to the daemon. Applications do not need to consider device capabilities, and use standard audio daemon interface. Drivers only need to export supported formats and sample rates, and any format conversion and/or resampling is taken care of by the daemon.

The major disadvantage of using an audio daemon is the added latency. Applications may require audio output to be synchronized with internal events or other outputs. While postponing video output might be a simple, yet ugly, solution, it can not be done if sounds are played as a response to user generated events. Thus keeping the latency, induced by additional layer, minimal is crucial.

5.2 Daemon Roles

There are two basic roles that an audio daemon should implement. The first one is audio stream routing and transformation, the other is device enumeration and control. These two roles are only lightly connected. Device enumeration is necessary to enable all routes, but device control can be separated into an auxiliary daemon. This separation helps to keep audio routing daemon simple, and separates two parts with very different purpose and functionality. In fact, running auxiliary control daemon is optional on some machines.

The role of the audio routing daemon is to advertise devices to connected applications and route audio data. It is also responsible for doing any conversions, whether mandated by format differences or requested by user settings. The initial implementation shall implement basic linear PCM format conversions, and routing. No resampling or user requested stream manipulation shall be supported at this stage.

The role of the auxiliary audio daemon is to aggregate all audio controls in one place. It shall read audio device control events and respond according to user defined rules. The responses shall include manipulation of the audio routing daemon, and modifications of driver settings. An example of its operation might be reading audio control commands produced by a multimedia keyboard and changing volume levels accordingly. Another example would be software mute of external speakers when audio jack connection is detected on Intel HDA devices.

5.3 Audio Server Design

Every time a new audio device is registered in the location service the audio routing daemon connects to the device and queries its abilities. Based on the exported functionality it creates an internal representation for playback and/or capture. The audio daemon requests access to the device's buffer and prefers to use the buffer position interface if it is available. It then allows audio applications to connect to this device and routes audio data traffic between applications and devices. Several approaches to routing demon architecture and audio interface were considered, their strengths and weaknesses assessed.

Master Audio Routing Daemon

The most direct approach to audio server is to ignore differences between audio devices and audio application. After all both of them can produce and/or consume audio data, and both are required to be simple. Applications and drivers register to the location service as either data providers – *sources* or data consumers – *sinks*. The audio daemon connects sources to sinks and routes audio data and applies necessary conversions. Data that are made available by audio sources are made available via shared buffers and shared buffers are used for data output as well.

In a way it can be said that using this design every audio application needs to provide a virtual audio device in order to play or capture audio data. The major advantage of this approach is reduced complexity of the audio daemon itself. The daemon does not have to implement an internal abstraction layer in order to communicate with drivers and applications, as this abstraction is implemented at the IPC layer. If the complexity is reduced even further, for example by restricting interfaces to single channel and one sample rate and format, the resulting design is similar to the one implemented by JACK audio daemon.

However, this concept has several significant disadvantages. Although the design and implementation of the daemon itself is simple the complexity is moved to the IPC level, especially when it comes to application interface. Applications need to have their own timer source in order to produce or consume data in a steady rate and avoid buffer overruns and underruns. Although the audio daemon has to be able to handle sources and sinks that use different timers, the added complexity on the application side is nontrivial. Moreover, applications may expect the audio interface to provide a timer, and synchronize other work around audio input/output instead of having to account for additional synchronization overhead.

Just In Time Data

JACK daemon uses small or no buffers to minimize latency. Playback data are requested from an audio application just in time to mix them and write to audio device. This technique enables applications to control and modify the data until the very last moment, it limits audio latency and keeps routing implementation simple and good performance.

The main disadvantage of this approach is its timing requirement. The daemon needs to know how much time it needs to mix inputs. It means that the more input are mixed into a single output the less time every application has to provide data without increasing audio latency.

A prototype of an audio daemon that offers this kind of interface was implemented and times necessary to retrieve and mix audio data measured. The resulting data are in the following table 5.1. The experiments was conducted on otherwise idle uniprocessor system running in qemu. Audio clients either read data directly from a file (fread) or copied the data from a prepared memory buffer(memcpy). Tests were repeated with an additional debug output added to the data retrieving callback to trigger more IPC communication and possible rescheduling. Timer precision was set to 1ms. Tests marked with '*' showed significant variation and the recorded values are approximate.

	fread	fread + printf	memcpy	memcpy + printf
Single client	7ms	20ms(*)	<1ms	1-2ms
Two clients	11ms(*)	50ms(*)	1-2ms	2ms

Table 5.1: Mixing times on otherwise idle system

The same test was repeated with an endless loop task running to increase CPU utilization. The results were very similar to the idle system, but the test cases that showed great variation turned even more volatile. Single client times occasionally reached up to 40 ms, a duo of applications could take as long as 90 ms to provide data. These results show high dependence on scheduler behavior. Callback routines have to be optimized with minimal reliance on IPC calls to provide data in order to have good round trip times.

Providing just-in-time data is a simple and elegant solution to minimizing latency. However, its timing requirements make it unsuitable for general purpose audio stack. JACK is a professional audio tool with limited input format and it uses realtime toolkit to provide reliable timing in order to function properly.

Virtual Device Interface

Another way to reuse audio device API in audio daemon is to provide a virtual audio device. There are very few hardware limitations for a virtual device so it can provide large cyclic buffer and a playback or record position. Although, the buffer size is limited only by the OS and its IPC mechanism, the granularity of the position information is dependent on the capabilities of connected audio devices.

Using this approach, every application will have an illusion of one or more audio devices that it can use exclusively. In the HelenOS environment the only difference between using an audio device directly an using the audio daemon would be the target task of the initial connection. This approach offers wide range of possibilities for audio applications. Applications might choose to convert and mix audio data or use multiple buffers and let the audio daemon handle the necessary processing. The availability of playback position enables applications to add data just before they are needed, and the latency is determined by the response time of a standard IPC exchange and the cost of a buffer update on the server side.

On the other hand, a general mapping of N to M cyclic buffers of different sizes and producing or consuming data at slightly different pace creates a synchronization nightmare on the server side. The semantics of the buffer position information needs to be well defined. It can be determined by the position reported by either the slowest or the fastest connected device. Ideally, the semantics would be client configurable. This would add to the complexity of the entire system. Moreover, the limited size of client buffers would cause buffer underrun errors if the position difference between multiple devices reaches high levels.

While this approach looks attractive from the client side, it creates serious issues in audio daemon design. Moreover, it provides more capabilities than are necessary for audio playback and recording. It implements a view that is based on audio hardware implementations instead of audio software requirements. One of the roles of the audio daemon is to bridge this gap. Applications do not need the ability to modify existing audio stream. Starting new streams and stopping those that are no longer desired can handle most of the generic audio requirements, like long term playback or recording, and on demand event response. This observation has lead to the final design discussed in the next section.

Stream pipes

Stream based approach is built on the assumption the it is not necessary to have access to the audio data scheduled for playback. Low overhead in playback start and end is enough to replace this functionality. Moreover, it is beneficial if audio applications do not do any unnecessary audio conversion and mixing themselves. Audio data produced by one application may need to be converted and mixed with data provided by other applications within the audio daemon anyway. Creating separate channel for every audio stream concentrates audio data manipulation in one place, the daemon, and reduces both computational overhead and clients' complexity.

Moreover, if the streams are implemented using pipe interface they represent linear buffers, and avoid the synchronization problems of the shared buffer design. The audio daemon will be able to hold enough backlog audio data to handle stream destruction and device pace differences gracefully. The daemon shall maintain stream buffers of client specified size and use data from these buffers for playback. Applications should use large buffers and open new streams for immediate playback instead of trying to keep a single stream filled with up-to-date data. This approach will enable the daemon to optimize data copying and use as few device events as possible.

It should be noted that although the pipe design treats data transfers differently than the above described shared buffer interface, it does not prevent the use of memory sharing to transfer data. The pipe approach uses one way data transfers and leaves the data transfer technique at the discretion of the HelenOS IPC layer. This keeps the abstraction layers separate and makes network transparency work 'out of the box'.

General use-cases, like continuous music playback and event triggered audio are easily implemented on top of stream pipes. Variable size of server side buffers shall provide enough data for audio processing when necessary and rapid creation of new streams shall handle immediate playback. The recording part of this interface is really simple. Simple reverse of data flow within the audio daemon and using read IPC calls in place of writes shall provide sufficient capabilities for general purpose audio recording. This last considered audio stack architecture is similar to the one implemented by PulseAudio. That is no surprise. PulseAudio is a recent project that builds on the knowledge and experience of previous audio stack implementations. Although it relies on Unix sockets to transfer data instead of using the advanced HelenOS IPC mechanism.

Chapter 6

Implementation

6.1 HelenOS modifications and Sound Blaster 16 driver

It was mentioned in section 1.3 that HelenOS features when it comes to audio device support were rather lacking. There were two major problems, the lack of high precision timers, and no support for additional memory restrictions in DMA memory allocator. The former is a generic problem for the daemon or any other audio software that wants to use buffer position interface. Lack of high precision timers is a known problem on the hardware side and Intel HDA devices implement their own timers. This problem was easily worked around by increasing the kernel tick frequency to 1000 Hz. Although the problem was elevated, it is far from the correct solution. Occasional timing glitches persisted and prevented the buffer position interface from being used in the audio daemon, despite its preference in audio device drivers.

The other problem is specific to Sound Blaster 16, and ISA devices. Most peripheral devices are connected via PCI bus or one of its successors, these devices are able to access system memory directly via a feature called *Bus Mastering*. The devices are limited only by the supported pointer size and their own arbitrary limitations on alignment and page boundaries. ISA devices on the other hand rely on a system provided DMA controller that has its own limitations. The most restricting of these is that the pointer size is limited to 20 bits for 8bit transfers and 24 bits for 16bit transfers. [23] While it is common to run

HelenOS on less than 4 GiB of RAM in order to work around the limitation for PCI devices, running on less than 16 MiB of RAM for 16bit DMA transfers proved tricky.

The final workaround introduced a new memory flag for zones that satisfy the pointer size limitation. Kernel memory initialization divides available physical address space into zones. Not every zone is available for memory allocation, and those that are are treated differently based on the flags set during initialization. There is a *low memory zone* flag for memory that is mapped directly to the kernel space, and *high memory zone* that is not. A new flag called *dma zone* was introduced to mark memory zones that satisfy ISA DMA pointer size limitations. The very first zone of available pages is reserved for DMA allocations only, to restrict the use of this limited resource. Its size on IA-32 HelenOS build is a approximately 1 MiB so it is enough to satisfy the needs HelenOS drivers currently have. Device drivers need to be aware of its limited size, and the routine to release this memory had to be implemented.

After these two problems were solved, implementing a Sound Blaster 16 driver was rather straightforward. The driver takes care to properly check input data in order to stay in the zone of well defined behavior and avoids corner cases. It means that the previously mentioned full duplex hack is not implemented. Conversion from 8bit to 16 bit audio is straightforward and without quality loss so 8bit transfers are not implemented.

6.2 Interface libraries on top of HelenOS IPC

HelenOS IPC mechanism is capable of sending simple messages, transferring data blobs, and establishing shared memory areas. Standard system library provides simple connection abstraction in the form of *sessions* and *exchanges*. *Sessions* maintain relation between two tasks, and *exchanges* provide connections to transfer messages and data. *Exchanges* may share a single connection provided by the session, or start a private connection for every *exchange*. There is no limit on how many messages can be exchanged via an *exchange* during its life.

Every interface used by HelenOS tasks is implemented on top of this abstraction. Different protocols are usually implemented in dedicated libraries, although some opt to split the client part into system libc library. Driver interfaces are generally part of the *libdrv* library.

Three interfaces were designed and implemented in this work, all of them are documented in great detail in Appendix C. The first two, *audio_mixer_iface*, and *audio_pcm_iface*, are implemented as part of the *libdrv* library. *audio_mixer_iface* is the simpler of the two, it allows applications to query audio devices, set volume levels, and mute status. The protocol itself is stateless and the library provides only thin abstraction on top of HelenOS IPC.

The other interface implemented in *libdrv*, *audio_pcm_iface*, provides audio device abstraction over IPC. Each audio device is represented by a ringbuffer, this buffer is shared to an application and controlled using this interface. This interface is also used to query device capabilities. Devices support different audio formats so the application must make sure that data in the shared buffer are in a format that is known and supported by the device. Devices may provide current read/write position or report events like startup, termination, and forward progress. A handler function that is called to process these events is also registered using this interface. The protocol is stateless and provides little more than thin layer on top of IPC that takes care of data transfers and message passing.

The third interface handles communication to and from audio client applications and the audio daemon. It is implemented in its own library called *libhound*, and provides the topmost level of HelenOS audio stack. Application view of the system audio functionality is centered around two classes *hound_context_t*(context) and *hound_stream_t*(stream) and it implies that their equivalents exist on the server side. An overview can be seen on Figure 6.1.

While a great freedom is allowed in server implementation by the library, all implementations must provide *hound_context_id_t* that associates streams to their parent context. Contexts handle management part of client-server communication. They are represented by a HelenOS IPC session, although a context does not maintain an open exchange. Contexts are not expected to cause heavy IPC traffic so starting a new exchange for every command does not add too much overhead. Moreover, sharing an IPC exchange would require additional client side synchronization as almost all commands are implemented by multiple IPC messages. The commands available for contexts include listing available targets (audio devices), connecting to, and disconnecting from a target. Contexts represent a source or sink for data provided or requested by the application. Contexts are connected to audio devices and provide a point of reference within audio server. Contexts are unidirectional, it means that they can either capture or playback audio. Applications that need bidirectional audio have to use more than one context. One thing that contexts don't do is data transfers, each

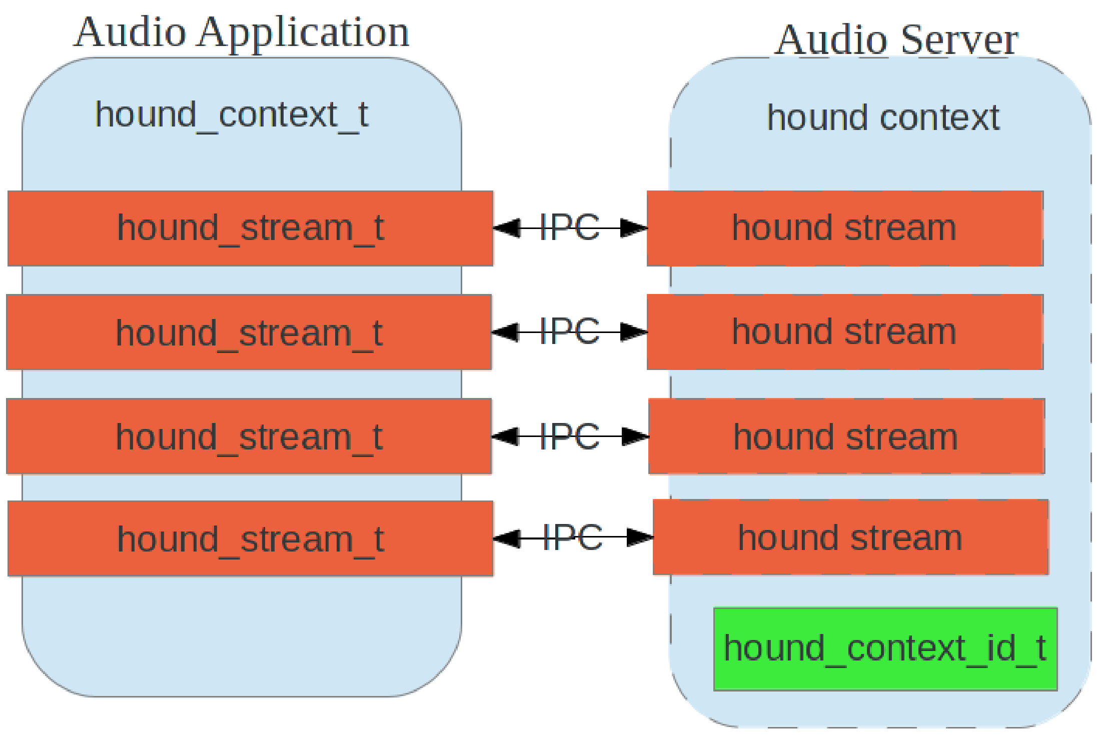


Figure 6.1: libhound IPC

context maintains a list of streams to meet its data transfer needs.

Streams are the audio data transfer entities associated with contexts. Each stream can transfer data in different format. Creating a stream is very lightweight, every open IPC exchange can turn into stream pipe by using STREAM_ENTER command, and exit using STREAM_EXIT. The state transitions are shown in Figure 6.2.

This form of stream creation is very cheap, in fact the libhound client library implements new streams by starting a new IPC exchange over an existing session, and issuing a single STREAM_ENTER command.The stream creation is lightweight and there is no hardwired limit to number of streams, although overuse can cause degraded performance and audio glitches. STREAM_DRAIN is the only command that is supported in the stream state. Issuing this command will block until all data in the stream has been played.

Stream behavior can be tweaked using flags and parameters. The data format parameter was already mentioned, using streams with different formats enables

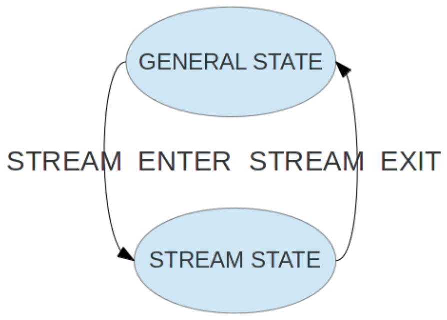


Figure 6.2: Communication states

audio applications to forgo any audio data mixing and conversions and leave this work to the daemon. The other parameter is buffer size. There are no buffers on the client side, other than those implemented by the application itself, but there are buffers on the daemon side. This parameter specifies the size of daemon side stream buffer. Data that were sent to the buffer cannot be modified, but will be removed if the stream is destroyed before they were consumed. Size of the buffer is left for the application programmer to decide but big buffers are recommended for optimal performance. Desire to achieve low latency can tempt the programmer to use smaller buffers but it is generally better solution to just start a new stream when the need arises. Only one flag is currently supported, the DRAIN_ON_EXIT flag. As the name suggests setting this flag makes sure that the data were consumed before the stream is destroyed. This flag is implemented on the client side, *libhound* sends STREAM_DRAIN command before sending STREAM_EXIT.

The client part of *libhound* library includes a convenience features for simple audio needs. The first one is that every context provides a default stream, call the *main stream*. Parameters of this stream are set during context creation, and the stream can be accessed without having to be created explicitly. This feature is ideal for simple audio clients that wish to play or capture audio, like music players or simple voice recorders. The other convenience feature is the availability of immediate playback. The library implicitly creates a new stream with the specified format and buffer size equal to the provided data size, sends the data, drains the stream and destroys it again. This feature is targeted at event triggered playback, like key press sounds.

6.3 Implementing Audio Daemon

At the center of the HelenOS audio stack is its audio daemon, called *hound*. *Hound* communicates with a *location service*, that maintains lists of available service providers. *Hound* registers server name to a naming service and than uses this name to register as an audio provider. The *location service* provides a list of available audio devices, it also sends a notification every time this list is updated. While devices are looked for and added automatically, applications have to connect to the daemon explicitly, this is implemented in the *libhound* library.

The central part of the daemon is a singleton instance of *hound_t* class. This class is very simple, in fact it's just a collection of lists and a mutex to synchronize access. There are separate lists for audio devices, contexts, audio sources, audio sinks, and connections between them. This class also provides listings to the *libhound* interface.

Discovered audio devices are represented by the *audio_device_t* class. This class uses audio pcm interface to access device buffer in system memory and control the device. It maintains IPC connection to the device driver and processes playback or recording related events that originate from the device. Audio device class also includes an instance of audio source and sink. One of them, or both based on device capabilities, are registered with the *hound_t* class. Current implementation relies on event based playback and record interface instead of the more advanced one that provides exact buffer position. Performance without high precision timers was not reliable even if the system timer tick time was reduced to 1ms, and it was heavily dependent on scheduler and other running tasks. Changes necessary to support this interface in the *hound* daemon are discussed later in this chapter.

Connected audio applications are represented by the *hound_ctx_t* class. Unlike audio device, application context can only include sink or source, not both. Contexts are the server side implementation expected by the *libhound* library, and maintain a list of streams used by the connected application. Streams, represented by *hound_ctx_stream_t*, are another class expected by the *libhound* interface. Streams store data sent by the application about to be consumed, or data that were not yet retrieved. Size of the stream buffer is set on creation and determines the stream's latency.

Both contexts and audio devices are abstracted to *audio_sink_t* and *audio_source_t* class. Sink consumes audio data, it might be either a playback device or a recording context. Source, on the other hand, produces audio data, it might be either audio player application or a recording device. This abstraction is necessary because only sources and sinks can be connected by the *connection_t* class. Connection connects single source to a single sink, but a sink or source can be connected more than once. A playback setup looks like the one shown in Figure 6.3. Thick arrows represent data copying and possible mixing and format conversion. Class *audio_pipe_t* implements an audio data FIFO buffer.

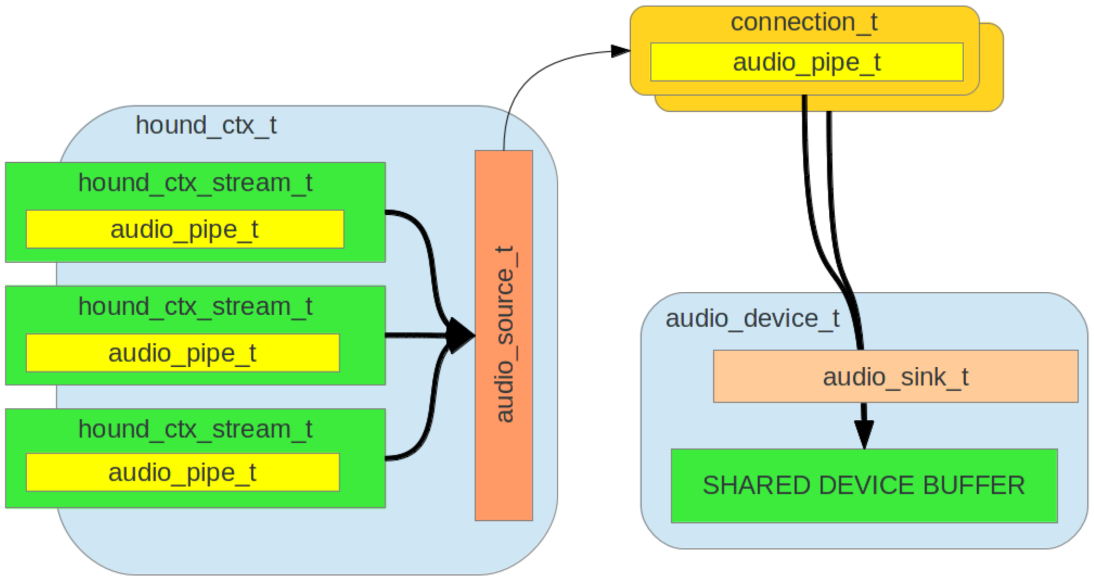


Figure 6.3: Hound playback dataflow

Audio sinks and sources might be either active or passive. The difference is in the way they are used rather than an explicit flag. In general sinks and sources that represent devices are considered active while application contexts are passive. Active sinks pulls data from connections when it needs to fill a buffer, and connections in turn request data from associated sources if their own internal buffer is empty. Active sources will push their data to its connections. Connecting active source to an active sink is possible and creates an audio device loopback. On the other hand connecting passive source to a passive sink will not work unless there another active sink involved. Thus peeking on playback data is possible, but direct copies will not work.

Both audio streams and connections contain an audio data buffer. While this might seem like a redundant and latency worsening design, it is not so. The important thing about audio sources, both passive and active, is that they always push data to all connected connection buffers. This guarantees that no data are lost because of different device pace. It also means that the only data in a playback connection buffer are the data that were already requested for playback by another device.

In general, format of either source or sink should not matter. That will be the case when resampling is fully supported, as mixing routines also perform format conversion. In practice it makes sense to set the sink to a specific format to avoid costly or quality degrading conversions. Moreover, lesser quality audio needs lower data rates and fewer buffer updates are necessary.

When a new connection is created it tries to set the sink format to the match the one of the source. This change will fail if the device is already running, thus the first connected source decides the playback format. The ideal solution would be if devices could handle format change gracefully so that the best format can be selected considering all connections. Or at least make gradual format upgrades possible.

The same problem appears when an application context's format needs to be determined. Context contain streams that use different formats, so a decision has to be made how to select format for the entire context. Current implementation sets context format to a hardwired default, but allows the first stream override it if the context has not been connected yet. This benefits the most common use-case of single stream playback. Ideal solution would have to set the format that allows conversion from all stream with as little quality loss as possible, and it would still have to be supported the connected devices.

Chapter 4.2 says that the buffer position device driver interface is preferable and yet the initial daemon implementation uses event based interface. There are two reasons for this decision. The first one is that using system timer did not produce consistently good performance, and the other one is that the daemon needs to implement event based anyway as a fallback measure for devices that do not support buffer position interface. Thus, it was given a priority.

Extending hound daemon to support buffer position interface will require changes in the way pipe buffers operate. The major advantage of a buffer position interface is that the playback buffer can be filled with data long into the future and it is still possible to update or replace them. It means that pipe buffers will need to maintain backlog of already consumed data in case the buffer needs to be reconstructed following. This backlog needs to be kept at stream buffer level in order to handle both stream removal and disconnect events.

Data structures

The single most widely used data structure in the audio daemon implementation is a linked list. List of streams, list of sinks, list of sources, list of devices. All these lists need easy insertion and removal and the lists are often iterated, though direct access to elements is not required. *audio_pipe_t* is a bit special list. It does not store audio data directly but its elements store offset and a reference counted pointer to data buffer. Thus audio data does not need to be copied for every connection it is pushed into. The data buffer is automatically freed when the last byte has been read by the last connection that used it.

6.4 Available demonstrators

Two demonstrator applications were developed during the course of this work, *wavplay*, and *mixerctl*. Although *wavplay* is just a simple wav file player. It implements several playback options that exercise different audio playback paths. The application implements its own .wav file header parser and uses both libhound and libdrv interfaces for device playback and recording. *Mixerctl* is a trivial implementation of device control utility. It uses audio control interface to display and modify settings of an audio interface. Default device path is hardwired. User manual to both applications is included in Appendix B.

Chapter 7

Conclusion

The primary goal of this work was to design and implement modern audio stack for HelenOS. Despite unexpected problems, the goal was achieved, albeit in a limited form. Hound daemon and its interface are built upon the experience of exiting audio daemon oriented architectures. There were several interesting results uncovered during the course of this work. The lack of high precision timers proved fatal to the use of the more advanced 'buffer position' interface. And despite arguably more advanced, better performance, IPC of the micro-kernel environment the process boundaries are best implemented in the very same places as their monolithic counterparts.

The secondary goal of USB Audio support was not reached. HelenOS USB stack is in the process of radical overhaul that will bring not only support for more sensible implementation of isochronous transfers, but it will be easily extensible to support high speed (USB2) communication and hubs capable of transaction translation. This work can be observed in the HelenOS usb branch available on launchpad.net.

7.1 Future work

While the current implementation provides a solid base for audio support in HelenOS, it is far from feature complete. It can be extended in both depth and breadth. The most visible flaw is the lack of resampling support in the format conversion routines. Resampling is a complex problem that requires specialized algorithms that are out of scope of this work. Especially converting between the most common frequencies of 44.1kHz and 48kHz is difficult to do right.

This problem might be partially worked around by implementing more advanced audio format negotiation in the audio daemon. Setting audio device playback parameters to match user supplied format better might help avoid resampling entirely. Another set of improvements for the audio daemon is on the fly data manipulation like fine grained volume control and channel maps.

The big area for future improvement is the addition of user specified transformations within the Hound daemon. Volume control and channel shuffling, are the first that come to mind but the possibilities are endless.

Then there are the usual improvements, more audio drivers, better audio drivers, more audio applications, better audio applications, more user friendly control utilities and so on.

Bibliography

[1] *A close look at ALSA*, http://www.volkerschatz.com/noise/alsa.html, retrieved, Apr 21, 2012

[2] *A Guide Through The Linux Sound API Jungle*, http://0pointer.de/blog/projects/guide-to-sound-apis.html, retrieved, Apr 8, 2013

[3] *Core Audio Overview: What Is Core Audio?*, https://developer.apple.com/library/mac/#documentation/MusicAudio/Conceptual/CoreAudioOverview/WhatisCoreAudio/WhatisCoreAudio.html, retrieved, Apr 1, 2013

[4] *Hearing range - Wikipedia*, http://en.wikipedia.org/wiki/Hearing_range, retrieved, Apr 1, 2013

[5] *High Definition Audio Specification, Revision 1.0a*

[6] *How does DMA work?*, http://bos.asmhackers.net/docs/dma/docs/dma0.php.htm, retrieved, Nov 14, 2011

[7] *Introduction to sound programming with ALSA — Linux Journal*, http://www.linuxjournal.com/article/6735, retrieved, Apr 21, 2012

[8] *JACK documentation*, http://jackaudio.org/documentation, retrieved, Apr 8, 2013

[9] *Linux ALSA sound notes*, http://www.sabi.co.uk/Notes/linuxSoundALSA.html, retrieved, Apr 21, 2012

[10] *Nyquist-Shannon sampling theorem*, http://www.princeton.edu/˜achaney/tmve/wiki100k/docs/Nyquist%E2 %80%93Shannon_sampling_theorem.html, retrieved, Apr 1, 2013

[11] *OSDev.org ISA DMA*, http://wiki.osdev.org/DMA, retrieved, Nov 14, 2011

[12] *OSS v4.x API reference*, http://manuals.opensound.com/developer/, retrieved, May 5, 2012

[13] *PCM - MultimediaWiki*, http://wiki.multimedia.cx/index.php?title=PCM, retrieved, Feb 16, 2013

[14] *pulse-client.conf(5) Linux man page*

[15] *PulseAudio documentation*, http://freedesktop.org/software/pulseaudio/doxygen/index.html, retrieved, Aug 17, 2012

[16] *Sampling rate - Wikipedia*, http://en.wikipedia.org/wiki/Sampling_rate, retrieved, Feb 16, 2013

[17] *Sound Blaster 16 - Wikipedia*, http://en.wikipedia.org/wiki/Sound_Blaster_16, retrieved Mar 3, 2013

[18] *Sound Blaster Series Hardware Programming Guide*, http://pdos.csail.mit.edu/6.828/2008/readings/hardware/SoundBlaster.pdf, retrieved, Nov 14, 2011

[19] Takashi Iwai *ALSA architecture overview*, http://www.alsa-project.org/˜tiwai/lk2k/archtect.gif, retrieved, Apr 22, 2012

[20] *Universal Serial Bus Device Class Definition for Audio Data Formats, release 1.0*, http://www.usb.org/developers/devclass_docs/frmts10.pdf, retrieved, Mar 3, 2013

[21] *Universal Serial Bus Device Class Definition for Audio Devices, Release 1.0*, http://www.usb.org/developers/devclass_docs/audio10.pdf, retrieved, Mar 3, 2013

[22] *Wave PCM soundfile format*, https://ccrma.stanford.edu/courses/422/projects/WaveFormat/, retrieved, Nov 14, 2011

[23] *Zet project ISA DMA*, http://zet.aluzina.org/index.php/8237_DMA_controller, retrieved, Nov 14, 2011

Appendix A

CD-ROM Content

This thesis includes a CD-ROM medium, on which you will find:

- **HelenOS sources** in the tar archive called *helenos-audio.tgz*.
- **HelenOS bootable CD image** image.iso
- **README** a readme text file, reading it is recommended
- **Qemu wrapper script** run.sh starts qemu-kvm with correct audio device
- **An electronic version of this thesis** , in the file *thesis.pdf*.
- **An electronic version of the programmers manual**, also included in Appendix C, the file is named *documentation.pdf*.

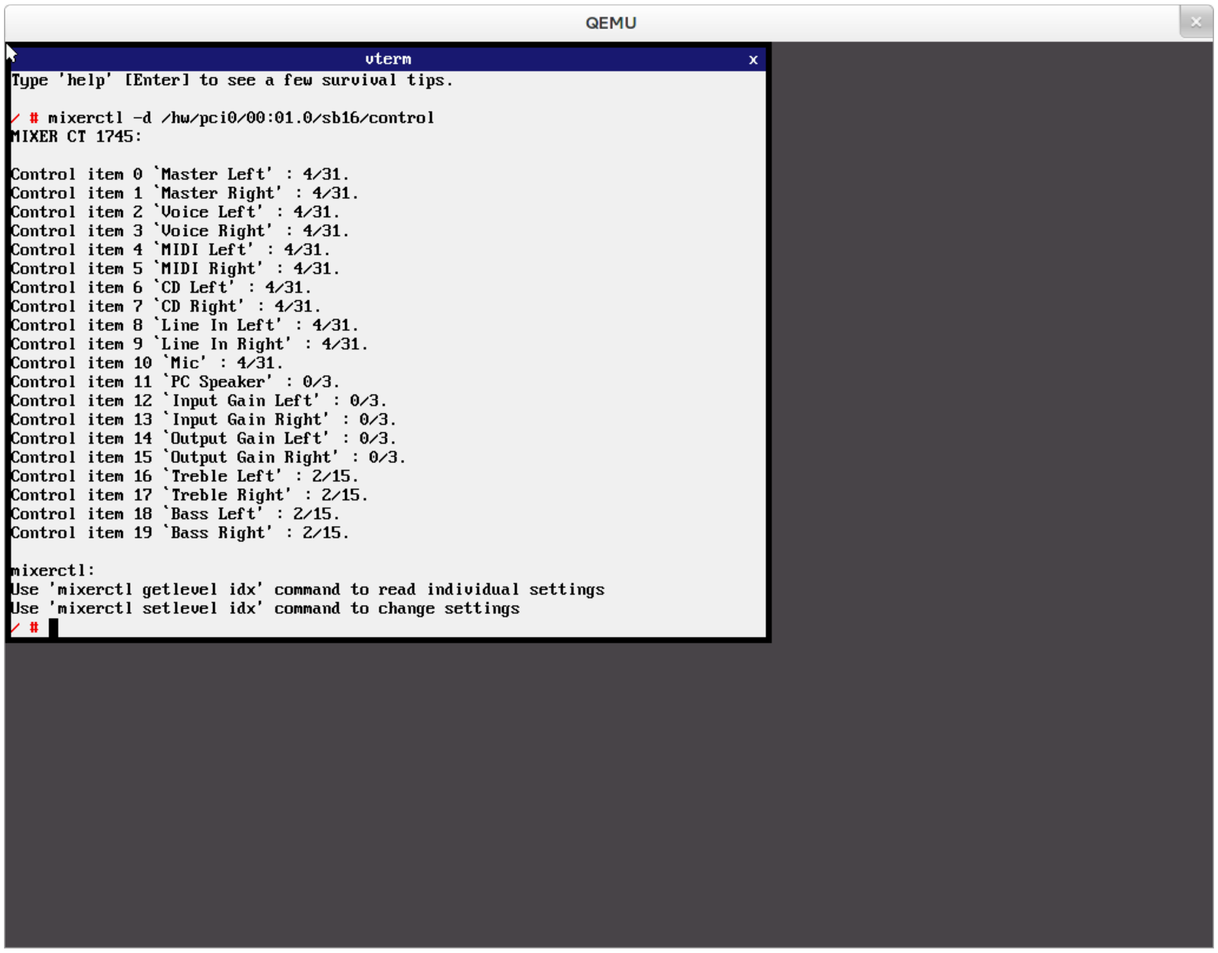
Appendix B

User documentation

mixerctl application

mixerctl is a simple application that controls selected audio hardware. It supports three modes of operation *READ*, *WRITE*, and *LIST*. In all modes the target device can be specified via '-d' option. If no device is specified *mixerctl* tries to connect to **/hw/pci0/00:01.0/sb16/control**. The purpose of this application is to test and audio device control interface.

LIST mode is the default, in this mode the applications lists all reported settings and their current and maximum level. I can be seen on the following picture.

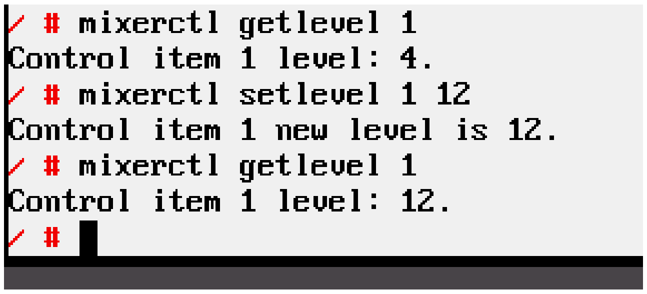


```
Type 'help' [Enter] to see a few survival tips.

/ # mixerctl -d /hw/pci0/00:01.0/sb16/control
MIXER CT 1745:

Control item 0 `Master Left' : 4/31.
Control item 1 `Master Right' : 4/31.
Control item 2 `Voice Left' : 4/31.
Control item 3 `Voice Right' : 4/31.
Control item 4 `MIDI Left' : 4/31.
Control item 5 `MIDI Right' : 4/31.
Control item 6 `CD Left' : 4/31.
Control item 7 `CD Right' : 4/31.
Control item 8 `Line In Left' : 4/31.
Control item 9 `Line In Right' : 4/31.
Control item 10 `Mic' : 4/31.
Control item 11 `PC Speaker' : 0/3.
Control item 12 `Input Gain Left' : 0/3.
Control item 13 `Input Gain Right' : 0/3.
Control item 14 `Output Gain Left' : 0/3.
Control item 15 `Output Gain Right' : 0/3.
Control item 16 `Treble Left' : 2/15.
Control item 17 `Treble Right' : 2/15.
Control item 18 `Bass Left' : 2/15.
Control item 19 `Bass Right' : 2/15.

mixerctl:
Use 'mixerctl getlevel idx' command to read individual settings
Use 'mixerctl setlevel idx' command to change settings
/ #
```

WRITE mode enables user to change a setting level. It is specified by using the *setlevel* command.

READ mode reads a single value. It can be entered using the *getlevel* command.

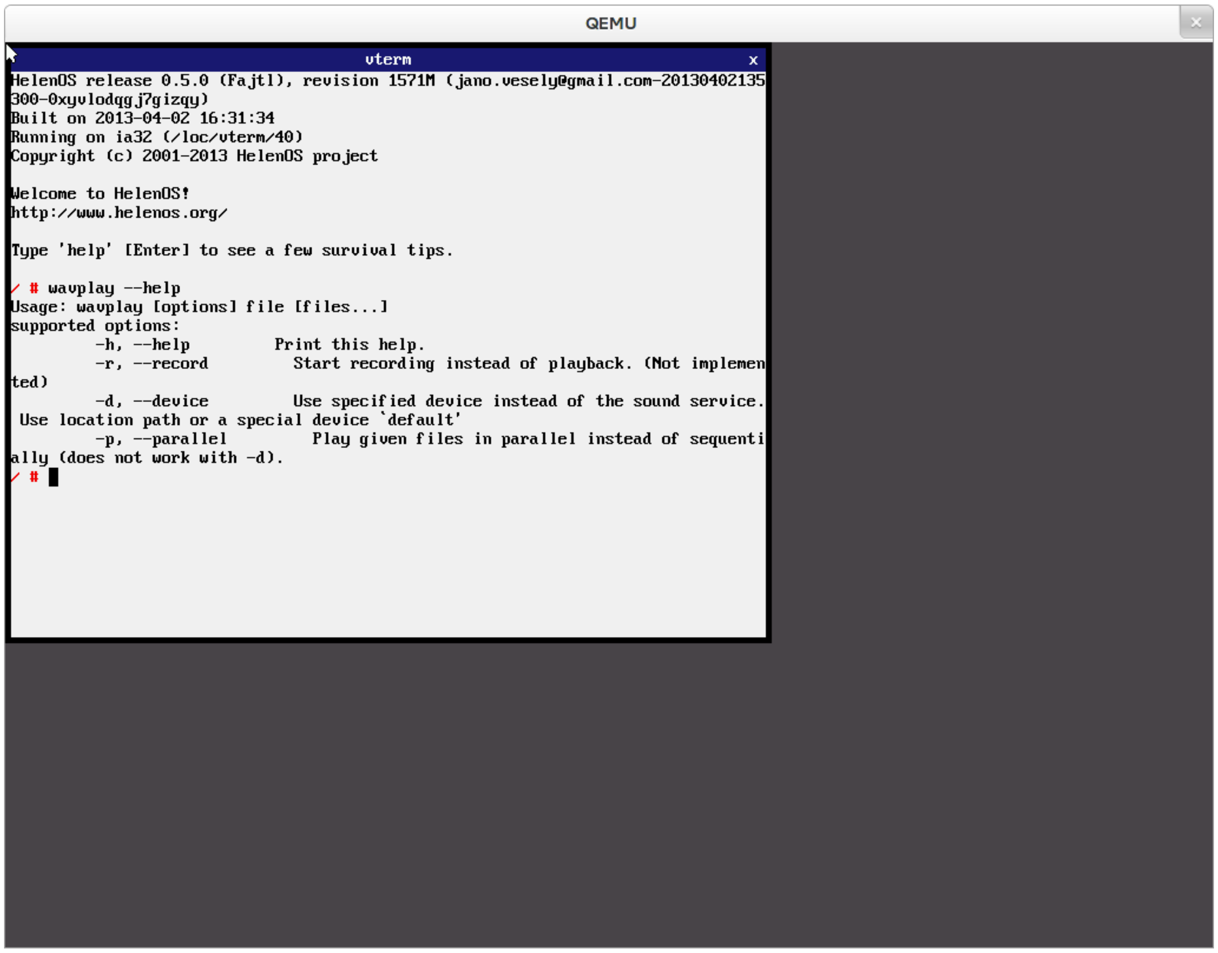
Examples of read and write commands can be seen in the next picture.



```
/ # mixerctl getlevel 1
Control item 1 level: 4.
/ # mixerctl setlevel 1 12
Control item 1 new level is 12.
/ # mixerctl getlevel 1
Control item 1 level: 12.
/ #
```

wavplay application

wavplay is a simple command line audio player and recorder. This application is a bit more complex than *mixerclt* and offers users a usage help. Help is displayed if the application is run with no arguments or if **-h** or –**help** option was specified. It can be seen in the next picture.



```
HelenOS release 0.5.0 (Fajtl), revision 1571M (jano.vesely@gmail.com-20130402135
300-0xyvlodqgj7gizqy)
Built on 2013-04-02 16:31:34
Running on ia32 (/loc/vterm/40)
Copyright (c) 2001-2013 HelenOS project

Welcome to HelenOS!
http://www.helenos.org/

Type 'help' [Enter] to see a few survival tips.

/ # wavplay --help
Usage: wavplay [options] file [files...]
supported options:
         -h, --help         Print this help.
         -r, --record        Start recording instead of playback. (Not implemen
ted)
         -d, --device        Use specified device instead of the sound service.
 Use location path or a special device `default'
         -p, --parallel        Play given files in parallel instead of sequenti
ally (does not work with -d).
/ # 
```

By default *wavplay* tries to play audio files via the hound audio daemon. Multiple files can be specified on the command line and will be played in sequence. There are two options that modify playback behavior:

-d or –**device** switches from standard playback to directly using specified audio device. The device is queried for capabilities and suitable mode is selected. This option recognizes a special ***'default'*** device. Default device is an alias for the first audio playback device registered to location service.

-p or –**parallel** switches to parallel playback. Audio files are played in parallel and different playback paths are exercised in this mode. Unless you have files that mix well together this mode makes little sense for general use.

wavplay also supports audio recording. Recording mode is selected by using **-r** or –**record** option. You can use devices directly in recording mode by using **-d** or –**device** option. In fact you have to, qemu Sound Blaster 16 emulation does not support recording DSP commands, thus the hound based recording was considered low priority and never implemented.

Appendix C

Programming documentation

Implementing audio device drivers (API reference)

HelenOS audio device drivers should be implemented using the Device Driver Framework. There are two audio interfaces *Audio mixer interface* and A*udio pcm interface*. The latter is necessary for the device to be capable of working with the rest of the audio stack. The former is an auxiliary interface that improves user control of his/her system.

Audio mixer interface

Audio device drivers should implement audio mixer interface on one of its functions. Although it is not strictly necessary for correct behavior it is considered good practice to allow user to modify settings that influence device behavior. Moreover, mixer interface is very simple and its implementation will need only few lines of code. IPC protocol used by this interface is implemented in the *libdrv* library and both client side and device side functions are declared in `<audio_mixer_iface.h>`.

Mixer view of the device consists of a list of control items that can be read or set. The number of items in the list is not limited. The DDF function that implements audio mixer interface should be preferably called **control** or **ctl**. There are only four callbacks that the driver needs to implement. These functions are:

- ***get_info(char* name[OUT], unsigned items[OUT])*** This is most certainly the first function any client will call. In its two output parameters it provides simple description of the control interface and a number of control items controlled via this interface. Control items numbered 0 to reported value -1 are considered valid and should respond the functions below. The string returned parameter is only read so it is safe to return static and read-only values.
- ***get_item_info(unsigned item[IN], char* name[OUT], unsigned levels[OUT])*** This function requests information about a control item. The index of the item specified in the first parameter. Note that this parameter does not have to select a valid control item. It is driver's responsibility to behave correctly if the index is incorrect. In the second parameter this function should provide a description of the selected control item. And the third should return number of available settings levels of the selected control item. Valid levels are in range 0 to the reported value - 1. Note that the driver can interpolate the values to set actual level but the user facing range should always be 0 to #levels - 1.
- ***get_item_level(unsigned item[IN], unsigned level[OUT])*** The first parameter in this function selects a control item and the same rules apply as in the previous function. The second parameter provides the current settings level in the range 0 to #levels -1. For one index value the driver has to return valid results for both *get_item_info* and *get_item_level* or neither of the two.
- ***set_item_level(unsigned item[IN], unsigned level[IN])*** The first parameter of this function selects a control item,and the other selects a new setting level. The same rules about item selector apply in this function as well. Moreover, the value of the new setting level has to be checked too. The driver that supports *set_item_level* for one control item has to support *get_item_level* for the same control item. It means there can be no write only items. Note that read only items are allowed.

Sound Blaster 16 driver implementation of this interface can be found in `uspace/drv/audio/sb16/mixer_iface.c`

mixerctl is a simple client implementation of this itnerface

Audio pcm interface

Audio device drivers have to implement this interface on one of its functions in order to work with the other components of the audio stack. However, implementing this interface is not enough the driver also has to register the function in the *pcm-audio* category of the location service. IPC protocol used by this interface is implemented in the *libdrv* library and both client side and device side functions are declared in `<audio_pcm_iface.h>`.

Audio pcm interface covers both playback and recording, and each of them in two modes; buffer position mode and event mode.

Mixer view of the device consists of a list of control items that can be read or set. The number of items in the list is not limited. The DDF function that implements audio mixer interface should be preferably called **control** or **ctl**. There are only four callbacks that the driver needs to implement. These functions are:

- ***get_info_str(char* info[OUT])*** This function should return simple description of the device. The returned string is only ever read so it is safe to return static values.
- ***test_format(pcm_format[IN/OUT])*** This function is used to test whether a linear PCM format is supported by the device. Driver should return **ELIMIT** and modify parameters that are out of device supported bounds.
- ***query_cap(cap[IN])*** Audio driver can report device capabilities using this function. It should return 0 if the capability is not present or not supported, or the actual value of the queried information, ti si usually 1 for boolean options. Currently known capabilities are:
    - **AUDIO_CAP_CAPTURE** – device is capable of capturing audio data from outside source
    - **AUDIO_CAP_PLAYBACK** – device is capable of audio playback
    - **AUDIO_CAP_MAX_BUFFER** – maximum size of device buffer in bytes, has to be page size aligned
    - **AUDIO_CAP_BUFFER_POS** – device can provide accurate information on the current read/write position in the buffer
    - **AUDIO_CAP_INTERRUPT** – device is able to use send events after a fragment has been played/recorded
    - **AUDIO_CAP_INTERRUPT_MIN_FRAMES** – minimum size of fragments
    - **AUDIO_CAP_INTERRUPT_MAX_FRAMES** – maximum size of fragments

  Drivers have to support either event based or buffer position based playback recording. All rivers have to support non zero buffer size and at least one of the CAPTURE, PLAYBACK pair
- ***get_buffer(void **buffer[OUT], size_t size[IN/OUT])*** This is the core function of the interface. The driver should return in the two parameters pointer to and size of the device buffer so that it could be shared. The *size* parameter specifies a requested buffer size. This call should fail if the driver can not provide exactly that size. The value of 0 has a special meaning. It lets the driver decide the size of the buffer. Note that the driver can share its buffer multiple times, but it needs to keep track of how many clients have access to the buffer. The driver must not release a buffer that is shared to a client application. Note that single instance of this interface shall always provide only one buffer. Devices that support multiple buffers and hardware mixing shall report multiple DDF functions that implement this interface.

- ***release_buffer()*** The caller gives up its hold on the buffer. The driver might release it if there are no other users of the buffer.
- ***get_buffer_pos(size_t pos[OUT])*** If the driver reports that it is capable of providing buffer position information, it has to support this call and report the position int the output param.
- ***set_event_session(session[IN])*** *If the driver reports that it can provide regular playback/capture events, it has to support this call. This function sets IPC session that shall be used to report events. Currently know events are:*
    - ***PCM_EVENT_PLAYBACK_STARTED*** *– device has started playback and is active*
    - ***PCM_EVENT_CAPTURE_STARTED*** *– device has started capture and is active*
    - ***PCM_EVENT_FRAMES_PLAYED*** *– one fragment worth of data has been played and another one has started*
    - ***PCM_EVENT_FRAMES_CAPTURED*** *– one fragment worth of data has been captured and another fragment has been started*
    - ***PCM_EVENT_PLAYBACK_TERMINATED*** *– playback has been terminated, read/write pointer is at position 0*
    - ***PCM_EVENT_CAPTURE_TERMINATED*** *– capture terminated, read/write pointer is at position 0*
- ***start_playback(format[IN])*** The driver should start playback using the provided data format. It is not guaranteed that the format has been previously verified by ***test_format*** and this call can be rejected
- ***stop_playback(bool now[IN])*** *The driver should stop playback and reset read/write pointer to position 0. If the device is in event playback mode the parameter specifies whether it should stop now or after finishing the current fragment.*
- ***start_capture(format[IN])*** The driver should start capture using the provided data format. It is not guaranteed that the format has been previously verified by ***test_format*** and this call can be rejected
- ***stop_capture(bool now[IN])*** *The driver should stop capture and reset read/write pointer to position 0. If the device is in event playback mode the parameter specifies whether it should stop now or after finishing the current fragment.*

Sound Blaster 16 driver implementation is in `uspace/drv/audio/sb16/pcm_iface.c`, although the more complex parts are hidden in the DSP driver implementation. *Wavplay* application includes client side use of this interface in file `uspace/app/wavplay/dplay.c`. The implementation includes both buffer position interface and an event based one. Decision based on querying device capabilities is also implemented.

Writing HelenOS audio applications (API reference)

HelenOS audio applications should use the hound daemon for audio playback and capture. It is recommended that they use *libhound* library to communicate with the daemon.

Libhound client interface

libhound provides high level itnerface for implementing audio functionality in HelenOS. The only header that an audio application needs to use is `<hound/client.h>`, although stream flags are declared in `<hound/protocol.h>`. There are two types declared the client header header `hound_context_t` and `hound_stream_t`. Hound context provides means for general client to daemon communication. Creating and destroying connections, and listing connection targets. Data transfers are handled by streams. Every stream is associated with context that created it and therefore does not have to deal with setting up connections. Usually both are types needed in an audio application, although the simplest ones can get all the functionality they need from `hound_context_t`. This is because contexts provide a hidden convenience stream called the main stream. Use of this stream is optional, it will not be created if it's not needed, but may benefit simple use cases.

`hound_context_t` methods:

- ***hound_context_create_playback(char* name[IN], format[IN], size_t bsize[IN])*** This function creates a new context and registers it to the hound daemon. The name parameter is used to identify the context withing the hound daemon, and the remaining parameters are saved and sued by the main stream (see below). If the function fails for any reason, including failure to connect to the daemon it returns NULL. The resulting context can only be used for audio playback.
- ***hound_context_create_capture(char* name[IN], format[IN], size_t bsize[IN])*** This is the capture counterpart of the above function, and the same rules apply.
- ***hound_context_destroy(context[IN])*** This function completely cleans up and destroy existing context. It removes all streams, (invalidating all pointers to them), disconnects all connections and free the occupied memory.
- ***hound_context_set_main_stream_params(new_params[IN], size[IN])*** This function updates the main stream parameters that were set on creation. If the main stream was already operational it is destroyed.
- ***hound_context_get_available_targets(targets[OUT], count[OUT])*** This functions fills returns in its parameters list of available connection targets. For playback context it lists audio sinks, and for capture context it lists audio sources. It lists targets that are not connected currently connected.
- ***hound_context_get_connected_targets(targets[OUT], count[OUT])*** Similar the the one above but lists targets that ARE connected to the context.
- ***hound_context_connect_target(target[IN])*** This function creates connection between the context a the target provided in its parameter. The target should be preferably one previously retrieved using ***hound_context_get_available_targets***. This function accepts a special target named **HOUND_DEFAULT_TARGET** that connects the context to the first available sink or source. Connecting to the first target starts playback/recording immediately.
- ***hound_context_disconnect_target(target[IN])*** This function destroys connection between the context a the target provided in its parameter. The target should be preferably one previously retrieved using ***hound_context_get_connected_targets***. This function accepts a

special target named **HOUND_ALL_TARGETS** that disconnects the context from all currently connected targets.

`hound_stream_t` methods:

- ***hound_stream_create(context[IN], flags[IN], format[IN], size[IN])*** Although technically still a context method, this function servers as a constructor for `hound_stream_t` type. The newly created stream is associated with the context passed in the first parameter. *format* sets the PCM format of transferred data, and *size* sets maximum server side buffer size. If 0 is passed as the maximum buffer size there will be no limit. The *flags* parameter modifies stream behavior and can be handled both client side and server side. Currently only one flag is supported:
  - **HOUND_STREAM_DRAIN_ON_EXIT** – ***hound_stream_drain*** is called destroying the stream to makes ure that all data on the server side has been consumed.
- ***hound_stream_destroy(stream[IN])*** Destroys the streams, this function ends IPC connection and removes the stream from the parent context's list of stream, then free occupied memory.
- ***hound_stream_write(stream[IN], data[IN], size[IN])*** This function sends data to the daemon for playback. Note that trying to write more than maximum buffer size in one batch will fail. Trying to write smaller amounts when the buffer is full will block.
- ***hound_stream_read(stream[IN], data[OUT], size[IN])*** This function reads data from the daemon. Trying to read more than maximum buffer size will fail trying to read when the buffer is empty will block.
- ***hound_stream_drain(stream[IN])*** Block until the server side buffer is empty. Useful for waiting until a playback is complete.

Convenience and helper methods:

- ***hound_write_main_stream(context[IN], data[IN], size[IN])*** This is a wrapper write function for the convenience stream provided by hound context. Rules are the same as in ***hound_stream_write***
- ***hound_read_main_stream(context[IN], data[OUT], size[IN])*** This is a wrapper read function for the convenience stream provided by hound context. Rules are the same as in ***hound_stream_read***
- ***hound_write_replace_main_stream(context[IN], data[IN], size[IN])*** This helper function immediately destroys the main stream a starts a new one with fresh data effectively replacing unconsumed.
- ***hound_write_immediate(context[IN], data[IN], size[IN])*** This helper function creates a new temporary stream with buffer size equal to the provided buffer size, and transfers all data. After that it calls ***hound_stream_drain*** and destroys the temporary context.

Examples

Ex. 1: The first example is a simple audio player very much like the *wavplay* application that has been implemented. Only the relevant audio code included here. Error checking has been removed for clarity. Complete code is in `uspace/app/wavplay/main.c`

```
#include <hound/client.h>

/* assume default audio format, set unlimited daemon buffer */
hound_context_t *ctx = hound_context_create_playback(
    “nice playback context”, AUDIO_FORMAT_DEFAULT, 0);

/* this will start playback with no streams, we don't care */
hound_context_connect_target(ctx, HOUND_TARGET_DEFAULT);

/* now just read and pass data */
static char buffer[BUFFER_SIZE];
size_t data_size = 0;
while (data_size = get_data_from_somewhere(buffer, BUFFER_SIZE)) {
    hound_write_main_stream(ctx, buffer, data_size);
}

/* we are done */
hound_context_destroy(ctx);
```

Ex. 2: The second example uses the ***hound_write_immediate*** helper function to react on key-presses.

```
#include <hound/client.h>

extern uint8_t *beep_sound;
extern size_t beep_size;
extern pcm_format_t beep_format;

/* assume default audio format, set unlimited daemon buffer,
none of it will be used */
hound_context_t *ctx = hound_context_create_playback(
    "nice playback context", AUDIO_FORMAT_DEFAULT, 0);

/* this will start playback with no streams, we don't care,
 * this example is without streams most of the time */
hound_context_connect_target(ctx, HOUND_TARGET_DEFAULT);

while (getchar() != 'x') {
    hound_write_immediate(ctx, beep_format, beep_sound, beep_size);
}

/* we are done */
hound_context_destroy(ctx);
```