CSL
The necessity to make sound backends transparent

Tim Janik <*timj@gtk.org*>
Stefan Westerfeld <*stefan@space.twc.de*>
http://sfk.evilplan.org/csl/csl-paper.ps

Abstract

In this paper, we examine how different approaches to sound input and output handling are currently used in the free software landscape. We discuss kernel interfaces like oss and alsa, sound server approaches like aRtsd, ASD, ESound and NAS and requirements by applications or application suites like GNOME, KDE, WINE and XMMS. We proceed to show how different interoperability problems arise from the very heterogenous landscape, which affect both, developers and end users. Finally, we suggest solving these issues by introducing an abstraction for various sound backends, and present a sample implementation called *CSL* (common sound layer).

1 Many different aproaches to sound I/O

The theme of this paper is to show the benefits of a standardized sound device programming interface, that has the potential to ease the life of many developers and users who are involved with sound processing application. At present, unix applications which require access to sound input and output devices are forced to support a variety of different APIs[1] because there is no standard way to access sound drivers on different unix variants, or even on the same system if user space solutions, such as sound daemons[2] are also meant to be supported. An exemplary survey, as shown in Table 1, counts no less than 18 different sound driver backends included into commonly used sound processing applications. On the one side, there are kernel driver backends for systems such as HP-UX, Sun/Solaris, SGI/Irix, AIX, MacOS and BSD, to name just a few, and on the other side there are backends to support Sound Daemons such as ESound, aRts, ASD, NAS or AF.

1.1 Kernel Interfaces

On modern operating systems, developers usually don't have to deal with hardware devices directly. Instead, the kernel will provide appropriate functionality to access different kinds of soundcards. On linux, there are two commonly used ways to access a soundcard using a kernel driver: *OSS* and *ALSA* as shown in figure 1.

The OSS API is very widely used since it has been around for several years. A big advantage of the API from the developer point of view is, that it practically hasn't changed during this time. OSS drivers are included in the linux kernel, which means that applications can rely on OSS usually being available, and commercial OSS drivers are available for many other unix systems. Our comparision in table 1 shows, that OSS is the defacto standard which most sound applications support.

By comparision, the ALSA API was developed to provide an *Advanced Linux Sound Architecture* for the future. It is not yet finished, and there have been API changes during the development period, to incorporate feedback from the developers who have been using it. ALSA supports advanced audio and MIDI hardware and software much better than OSS. Quite a few applications support ALSA already, and a significant number of linux distributions are shipping with it. The inclusion into the mainstream kernel, which will also make the API a stable and reliable target for developers is expected during the linux 2.5 development cycle, so that ALSA might replace OSS for linux applications in a few years.

1.2 Sound Servers

Many hardware drivers are sufficiently abstracted to support concurrent use by applications, for instance a hard disk may be used by two or more applications

[1]API - Application Programming Interface
[2]With Sound Daemon we generally refer to user space programs which also provide sound I/O facilities as explained in more detail in chapter 1.2.

	AF	AIX	ALSA	aRts	ExtraBSD	BeOS	DirectSound	ESound	HPUX	MacOS	MiNT	NAS	OS/2	OSS	QNX	SGI/IRIX	Sun/Solaris	Win23
amp		X			X	X			X					X		X	X	
ASD														X				
aRts		X	X	[1]	X							X		X		X	X	
ESound		X	X		X				X					X		X	X	
Free Amp			X	X		X	X	X						X	X	X	X	X
Glame			X					X						X		X		
libAO			X	X			X	X		X				X		X	X	
libSDL		X	X	X		X		X		X		X		X		X	X	X
MikMod	X	X	X[2]				X	X	X	X			X	X		X	X	X
mpg123		X	X		X			X	X		X	X	X	X		X	X	X
mpg321			X	X				X						X			X	
SOX			X											X			X	
Timidity			X		X			X	X			X		X			X	X
Wine														X				
XMMS		X	X	X				X						X		X	X	

[1]) Will be possible with CSL
[2]) Plus an ancient UltraSound Driver

Table 1: Sound driver backends in commonly used sound applications

simultaneously and is not blocked by one application opening the device upon startup and closing it at shutdown time. The abstraction supplied here to share hard disk resources is widely known as *file*.

Graphic cards on the other hand are usually abstracted in a different way. Here, the kernel often just provides low level routines to access the cards pixel buffer and to switch graphics or text mode. An extra user space program is then run which interfaces normal application to the card, allowing for resource sharing. The most commonly known example in the unix community is the *Xserver* provided by the X11 windowing system.

Following the basic idea of abstracting access to graphic devices by means of an Xserver, several projects started out aiming at repeating X11s success in the sound domain. For instance, the user might want to listen to music using an Ogg/Vorbis [1] player like ogg123 or XMMS and still hear audio notifications when new email arrives, but most kernel level sound drivers are unable to give access to the sound card to more than one application. This is often perceived as a serious limitation.

Sound servers address this issue by combining sound I/O streams from multiple applications and interfacing with the kernel driver, as shown in figure 2. Thereby, they provide more than one application with sound I/O facilities at the same time, even if the kernel driver does not support this.

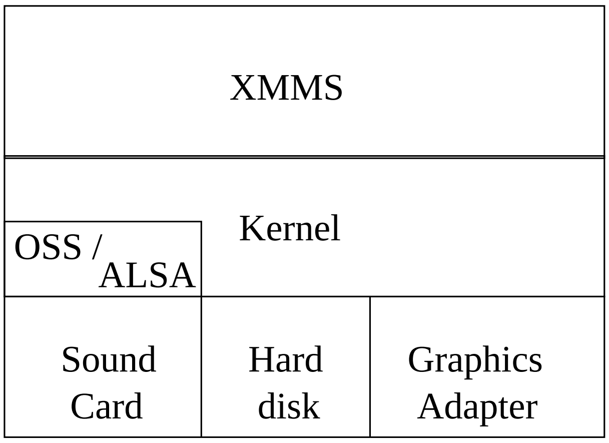

Figure 1: A single application using the kernels sound card driver solely.

Another straightforward idea in the sound server concept is providing access to sound functionality

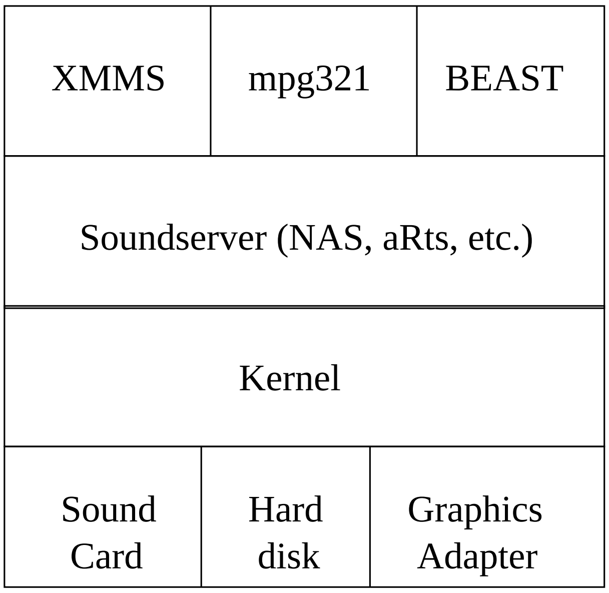

Figure 2: A sound server providing multiple applications with access to a single sound card.

over network connections. When running remote X11 applications, it is desired for sound to be played back on the local machine rather then the remote machine.

Sound servers can also be used to serve a variety of addtional needs. They can make different audio applications interoperable, route audio streams between the applications or process them with additional effects, to name just a few.

1.3 Media Frameworks

Recently, several new projects occoured which are aiming at an even higher level of abstraction for dealing with media streams [3], for instance GStreamer, the Java Media Framework or aRts. They perform operations such as encoding, decoding, framing, processing, transfer and rendering of media contents.

For the scope of this paper, it is enough to recognize that, while media frameworks solve low-level sound I/O issues for the application developer, they themselves suffer from the very problem of low-level sound API diversities which forces them into implementing a variety of different backends.

[3]Media Streams - Data flow of audio, MIDI and video content

2 Sound Applications in the Free Software Landscape

Unfortunately, the current set of abtraction layers as shown in the previous chapter are by large not sufficient, to conveniently solve the problems users and developers are currently facing. In the following, we're going to examine these problems in a detailed manner.

2.1 Developer Requirements

Developers face several challenges during the course of sound application development.

They need to support various sound backends, some of which are platform specific and all of which are implemented using a different API. This significantly increases complexity of many sound applications.

As free software is often written and maintained by a small team of core developers, who usually do not have access to all platforms their software runs on, a lot of these backends tend to be third party contributions, which makes them hard to maintain over a longer period of time.

The need to reimplement various sound backends for every new sound application presents a gross duplication of implementation effort. Projects with a small circle of contributors are especially suffering from this and often can't fully exploit their potential user base due to portability issues remaining unfixed for a long period of time.

Newly occuring sound APIs, such as modernized kernel drivers, or support for a new sound server cannot be planned for in advance, and cause continued maintenance requirement of sound backends in every single sound application.

And last but not least, different sound drivers require different degrees of integrational work, for instance an OSS driver exports file descriptors that have to be polled, while advanced sound servers like aRts may require full main loop and language integration and impose additional library dependancies.

2.2 User Requirements

End users have varying requirements, often situation dependant. For instance, running a media player in a university may mean the user has to use headphones and reroute sound streams from a central server to a local workstation. Or in case the user is playing a game, he probably wants to disable sound servers and let the application use the kernel drivers without further task switches in order to reduce latency.

Another example would be using a composition application of the KDE desktop project like brahms which is based on the aRts media framework and interfacing it to a sound server which is used to serve GNOME programs. Unfortunately it is not currently possible to get KDE and GNOME applications to produce sound side-by-side, as both desktop projects have distinct sound server requirements, namely aRts and ESound, which both lack the needed backends to interface with different sound servers.

To allow users to get these and similar setups to work, applications need to leave the choice up to the user and offer a variety of configurable backends without imposing restrictions which are for example due to platform and/or portability issues. Ideally, sound backend configuration (such as choice of daemon and routing requirements) of the different sound applications used by a user is done in the same standardized way and understood by all applications utilized.

2.3 Application Categories

Applications deal with sound in various ways which can be roughly categorized into:

- **Casual Sound Needs**
  Some applications surve a purpose which is completely unrelated to sound, and only casually need to produce some audio output. A window manager, which produces sound if a window closes is one example.
- **Media Players**
  Media Players are dedicated applications, with the sole purpose of playing a media file, normally either audio or video. Examples are xmms, ogg123 or noatun.
- **Games**
  For games it is often essential to play sounds and music in tight reaction to what the user does. Examples are quake, tuxracer or parsec.
- **Composition or Synthesis**
  There are a lot of applications dedicated to creating, recording or editing music, such as brahms, BEAST or ardour.
- **Infrastructure Applications**
  Some applications like sound servers do not have a purpose of its own, but exist as infrastructure components. Examples are aRts, ASD, ESound and NAS.

3 Designing a common Abstraction

As we have seen in the previous section, the current state of affairs is causing quite a few problems to both, developers and users. Considering the possibilities of how they can be adressed in the most elegant way, we suggest implementing a new unified abstraction for doing sound I/O, which can be used by the majority of the applications discussed. To do this, we'll start by examining which functionality is commonly needed.

3.1 Application Requirements

First of all, there are applications with what we called *Casual Sound Needs*. These usually just need a very simple way to play back a media file, in order to notify the user. For this, a simple API which can play back a sound file (like .wav) on demand should be sufficient.

Next, there are *Media Players*. Looking at the source code of media players we found that most of them come with an extensive collection of backends to support playing back sound on a variety of platforms. Most of them just need to play a stream of audio data. However, there is an interesting requirement for those media players which provide a visualization of the music. This usually needs to be synchronized to the music, which means that the player needs to know which sample is being played right now. Of course, this is also necessary for synchronizing video output with audio output.

We'll find a similar, yet a little different requirement in *Games*. Here, it is important that the reaction of the user is transformed into an appropriate sound effect as soon as possible, to be synchronized with what happens on the screen. Where a media player can use a potentially huge amount of buffering to avoid dropouts during playback, games need precise control over the amount of buffering in order to control latency. If the underlying driver supports it, memory mapping the output might be useful, but this is really just another way of saying that the buffering between the producer (the game) and the consumer (kernel or sound server) must be small, and well-controlled by the application.

Applications concerned with *Composition or Synthesis* might also need exact control over the amount of buffering (latency) and the ability to record and play audio streams. However, depending on how specialized these applications are, they might need to exploit the capabilities of special hardware up to its limits, so a few of these applications will eventually continue to provide support for several seperate driver backends.

As *Infrastructure Applications* usually need to provide whatever other applications need, their requirements include exact buffer control, and playback and recording of streams.

3.2 Considering Alternatives

We think that the optimal way to address these requirements and to solve the problems discussed in the previous sections of this paper is to introduce a new, dependancyless C library which only cares about sound I/O. But lets first look at a few possible alternatives.

First of all, one might argue that the kernel should fulfill all application requirements, ultimately obsoleting the need for sound servers entierly, and making applications interoperable. This comes close to the goals persued by the *ALSA* project. However, as a linux specific solution, this doesn't adress the more general portability problem, and a significant user base remains, who will still want to use sound servers for network transparency or routing between applications.

Another alternative is saying that media frameworks are to perform the required abstraction in the future, so that developers will not need to care about sound I/O details. However, media frameworks themselves need to be implemented somehow, and there the portability problem arises again. There are also applications which don't use media frameworks.

One might also suggest, that a great way of solving the issue is standarizing a protocol for sound servers, similar to the X11 protocol. Then, in the future, all applications just need to use this protocol, and no more platform specific backends need to be written. However, different sound servers currently use very different protocols, and a lot of innovation and discussion is going on here, so a general consensus is lacking.

Finally, one might argue that having a common sound server running all the time or at least standarizing the sound server API instead of the protocol solves the issue completely. However, persuing this path requires the following implications to be taken into account: First, not all systems have sufficient resources to run a sound server, especially on embedded linux systems. Second, the additional performance penalty involved with context switches might not be affordable by low latency applications. Third, implementing cascading sound servers involves additional effort.

3.3 Our Approach

So our conclusion is that a new abstraction layer for sound I/O has to be introduced. It should be a dependancyless C library, which is just a very thin wrapper around the various different sound APIs. It should provide a simple and straight forward interface to the developer and fulfill the requirements of most applications. In a simple setup, this involves playback ability for samples, for most other applications it is sufficient to provide input and output of sound streams with control of the latency involved.

Having one single library supported by every application, the maintainability and extensibility problems outlined in the previous sections should be effectively solved, while innovation in applications, sound servers and operating systems can continue. The library should be structured in a way, so that it is easy to add new backends, which immediately are available to all applications.

Given that we want the library to be usable by all kinds of applications, it should not add percievable overhead or introduce a huge dependancy chain. This is reasonably easy to achive with a thin wrapper layer, which can pull in backends on demand. It also needs to be very portable, because it will have to handle sound I/O in applications with high portability demands. Finally, the license should be liberal enough to allow it to be used in every application, for instance even in closed source games.

4 The CSL Project

As a result to the observations presented so far, the authors started out on an implementation of the proposed common abstraction, called *CSL* - the *Common Sound Layer*.

CSL is a plain C implementation without further dependancies, which provides a common API for handling sound I/O. It is intended to supply application programmers with a convenient way to access sound devices on various platforms, to interface to a variety of sound servers and to configure sound ouptut modalities regardless of the actual backend being used.

4.1 API features

We'll now give a brief survey of the corner aspects of the CSL API:

Initialization, in it's simplest form is performed by calling

```
CslErrorType
 csl_driver_init (
        const char *driver_name,
        CslDriver **driver);
```

which also has a variant for threaded applications `csl_driver_init_mutex()` and a counterpart `csl_driver_shutdown()`.

Sound streams are opened and closed with the following functions:

```
CslErrorType
 csl_pcm_open_output (
        CslDriver       *driver,
        const char      *role,
        unsigned int     rate,
        unsigned int     n_channels,
        CslPcmFormatType format,
        CslPcmStream   **stream_p);
CslErrorType
 csl_pcm_open_input (
        CslDriver       *driver,
        const char      *role,
        unsigned int     rate,
        unsigned int     n_channels,
        CslPcmFormatType format,
        CslPcmStream   **stream_p);
void
 csl_pcm_close (
        CslPcmStream    *stream);
```

These follow for the most part the commonly used OSS API, but also allow for stream role names to be supplied used by sound server backends to identify and if neccessary reroute streams.

Sound input and output facilities are also supported, as well as status inqueries:

```
int
 csl_pcm_read (
        CslPcmStream    *stream,
        unsigned int     n_bytes,
        void            *bytes);
int
 csl_pcm_write (
        CslPcmStream    *stream,
        unsigned int     n_bytes,
        void            *bytes);
CslErrorType
 csl_pcm_get_status (
        CslPcmStream    *stream,
        CslPcmStatus    *status);
```

As an alternative programming model, the user may choose to hook his code up through callbacks which are invoked after select(2)ing on the associated file descriptors and dispatching file descriptor events:

```
unsigned int
 csl_poll_get_fds (
        CslDriver      *driver,
        unsigned int    n_fds,
        CslPollFD      *fds);
void
 csl_poll_handle_fds (
        CslDriver      *driver,
        unsigned int    max_fds,
        CslPollFD      *fds);

typedef void (*CslPcmStreamCallback) (
        void          *user_data,
        CslPcmStream  *stream);
void
 csl_pcm_set_callback (
        CslPcmStream        *stream,
        CslPcmStreamCallback notify,
        void                *user_data,
        CslDestroy           user_data_destroy);

/* convenience variant to select on file descriptors
 * and handle events in one go
 */
int
 csl_poll_wait (
        CslDriver      *driver,
        unsigned int    n_user_fds,
        CslPollFD      *user_fds,
        unsigned int    time_in_ms);
```

References

[1] http://www.xiph.org/ogg/vorbis/

About the authors:

Tim Janik and *Stefan Westerfeld* are both studying Computer Science at the university of Hamburg in Germany and have been involved with the production of music and synthesis software for many years.

Tim Janik is core developer of the BEAST/BSE project (http://beast.gtk.org).

Stefan Westerfeld is core developer of the aRts project (http://www.arts-project.org).

Contents