Sound Systems on Linux: From the Past To the Future

Takashi Iwai <tiwai@suse.de>
SuSE Linux AG, Nuremberg, Germany

Linux 2003 Conference, Edinburgh, Scotland

1 Introduction: The Past

1.1 Life with Audio Cards

The development of audio and sound support on the Linux system has a long history. It has been implemented since the early version of Linux system. Before starting on a journey to the world of Linux sound system, let's take a short glance at the history of PC audio cards.

In the old good days until the middle 1990's, the sound cards equipped on the PC were all ISA cards. Their typical feature was SoundBlaster16 compatible, that is, 16 bit stereo PCM (pulse code modulation) playback and capture[1] . Almost all DOS and Windows games at that time supported the SB16-compatible cards. It had an MPU401 (a serial MIDI) interface shared with a joystick port. It satisfied the demands of both hobby gamers and musicians. The full-duplex capability was not a must.

The advanced feature at that time was the support of MIDI WaveTable and MOD playback on hardware like GUS and SB AWE cards. In the professional area, there were a few audio cards with the multiple channel and digital I/O interfaces. However, it was very rare that such a card works on a Linux system. Little users and manufacturer were interested in this regard.

The situation has varied slowly. The bus was changed from ISA to PCI. And almost all consumer audio cards have been based on AC97 codec[2]. It provides enough functionality for a desktop and a laptop PC including the full-duplex capability. There are variety of AC97 chips. Some of them supports multi-channel playback, and some even with a support of digital I/O. In the professional and so-called "pro-sumer" range, the multi-channel I/O with high quality (24 bit, 96 or 192 kHz) became mandatory. Many such cards support the digital I/O over S/PDIF or AES/EBU, too.

Usually, the evolution follows (or is brought from) its needs. In the case of audio cards, there have been two significant changes between the past and the present situations above: MP3 and DVD. The former makes it possible to distribute the PCM data stream in low bandwidth. As mentioned above, MIDI playback capability was regarded more importantly in the earlier days. Because of the MP3 and other compression technique, the distribution of audio over network takes now usually the PCM data directly instead of rendered audio data like MIDI and MOD. The CPU consumption is no longer a problem thanks to Moore's law. Eventually, the hardware-support of MIDI and MOD playbacks with WaveTable and FM syntheses became less interesting nowadays.

The another, DVD, brought the strong demand of multi-channel (5.1) and digital I/O capabilities. These capabilities are now standard even on an integrated sound chip on the motherboard of a desktop PC.

Another thing to be noted is the recent growth of popularity of USB audio devices. The device are really handy and easy. It's ready to use without opening a box (and no more screwdriver!). Although the USB 1.1 realizes only limited features because of its bandwidth, mostly requested features like multi-channel and/or digital I/O are supported on many devices.

[1] In this text, we call the recording of audio data as *capture* to prevent the confusion of meanings.

[2] Originally, *codec* stands for "compression and decompression". However, It's often used in different manners. Here in the case of audio cards, codec corresponds to the analog-digital-converter (ADC) or digital-analog-converter (DAC) function or the chip for it (like AC97 codec).

1.2 Life with Linux

Now let's back to the topic of our lovely Linux. In general, there are two basic components which build the sound system: the sound device driver and the sound server. The former is the hardware abstraction in the lower level, while the latter gives more high-end capabilities like multiplex access and mixing. In other OS like Windows, the boundary between these two components is not clear. The driver does some heavy jobs like mixing in the kernel, too. On the Linux system, however, these are regarded still separately. In the following section, the sound drivers and the typical sound servers in the past are explained briefly.

OSS

The core part of the sound system is the sound device drivers. On the Linux kernel, the OSS (Open Sound System) drivers have been employed as the standard sound drivers. The OSS drivers — which were originally written by Hannu Savolainen and formerly called as different names like USS/Lite, VoxWare and TASD — has currently two different versions. One is the free software included in the recent standard Linux kernel tree (OSS/Free) and another is the commercial binary-only drivers distributed by 4Front Technologies (OSS/Commercial)[1]. Both drivers have been evolved in different directions, but both of them still have (almost) the API compatibility, based on the earlier version of OSS.

The hardware support on the Linux system has been, unfortunately, relatively poor in comparison with Windows and MacOS. Some drivers were available only by the OSS/Commercial version, especially for the new cards and models. Positively looking, this means that Linux users had at least some support for most of popular audio cards. However, on the other hand, they were not free – in the sense of freedom, too. This situation has remained until recently, because many hardware vendors didn't like to provide the enough technical information to write a fully functional device driver as the open-source.

The OSS provides a simple and easy API[2]. The API was designed for the audio cards at the old ages, mainly 16bit two-channel playbacks and captures. The API follows the standard Unix-style via open/close/read/write system calls. The memory mapping (mmap) is also supported, so that the audio data can be transferred more efficiently. The mixer is represented as (up to 32) playback volumes, an input-gain, and a capture source selector.

The applications are supposed to open and access directly the device files, such as `/dev/dsp` or `/dev/mixer`. The format and buffers are controlled via specific ioctl's. The raw MIDI bytes can be sent or received via a raw MIDI device file. In addition, there is a so-called "sequencer" device, which is used to handle the MIDI events in the higher level with the sequential timing control.

EsounD

Many sound cards support only one PCM stream for each playback and capture, and the driver allows one process to access it exclusively. That is, if you access to a device from two or more applications at the same time, only one application can run and others will be blocked until it quits the operation. For solving this problem, a sound server is introduced. Instead of reading and writing a device file, applications access to a sound server, which accesses to the device exclusively on behalf. For multiple playbacks, a sound server reads multiple PCM streams from several applications and writes a mixed stream to the sound device (mixing function). For the capture direction, on the contrary, a sound server writes the PCM streams from the device to any accessing applications (multiplexing). Also, the samples being played by EsounD can be captured (loopback).

One of the commonly used applications for this purpose is EsounD[3]. EsounD stands for the "Enlightened sound daemon". Lately, GNOME adapted this program as its standard sound server, too.

EsounD provides the fundamental mixing and multiplexing capabilities of playback and capture PCM streams. The PCM streams are read/written through a simple Unix or TCP/IP socket. It is capable also to communicate over network. A simple authentication is implemented, too. The API is defined in libesd library, which is a quite straight-forward implementation in C.

EsounD has many different audio I/O drivers not only for Linux but also for other Unix systems. There are

OSS and ALSA (described later) drivers for Linux, currently. However, only one of them can be built in at the compile time.

aRts

KDE, a big competitor of GNOME, adapted another program as its standard sound server. This server program, aRts[4], was originally written as the "analog realtime synthesizer". Obviously it was inspired by the analog modular synthesizer system which was popular in 1970's. It consists of small basic components, and each of them can be "patched" with cables. aRts uses its own IPC method called MCOP for the patching instead of real cables. MCOP is similar with CORBA but much optimized for the data streaming (in fact, the earlier version of aRts used CORBA). The audio data is transferred over a Unix or TCP/IP socket.

As a sound server, aRts provides more ambitious features than EsounD. In theory, any effects can be inserted by patching modules. Modules are implemented as shared objects which are loaded dynamically onto the server. aRts has also MIDI control capability. As it's based on the CORBA-like communication model, aRts has also good network transparency.

The default language binding of aRts API is C++. Although there is a C wrapper on the C++ binding, the function is limited.

1.3 Always Problems in Life

Hard Life with OSS

As mentioned above, the OSS API is quite simple and easy. In fact, there are many audio applications to support it. However, the "simple and easy" means, in other words, that it doesn't support high-end audio functions. Namely,

1. The non-interleaved format[3] is not supported.
2. Supported sample formats are limited.
3. It forces applications to access the device files directly.
4. The support of digital I/O is quite poor. IEC status bits cannot be handled.
5. The mixer implementation is very limited.

The first two points lead to severe problems with the the progress of audio cards. Although many modern cards are capable to use such formats, the driver still cannot handle them but only the traditional 16 bit 2 channel stereo format. Especially, the non-interleaved format is essential for many professional cards with multichannels.

The handling of sample formats is not as trivial as it appears. Even a linear sample format can have different styles depending on the byte order and the size. For example, 24 bit samples can be packed either in 3 bytes, 4 bytes LSB or 4 bytes MSB. In addition, two different byte-orders for each case. All they express the same value!

The third point above might look not too critical. However, this is a serious problem if any pre- or post-processing is required for the samples before sending or after receiving them. A typical case is that the codec chip supports only a certain sample rate, here 48 kHz is assumed . For listening to a music taken from a CD, the sample rates must be converted from 44.1 kHz. In the OSS, this conversion is always done in the kernel although such a behavior is regarded "evil". This is because applications access to the device file directly, and there is no room between the application and the driver to process this kind of data conversion.

The fourth point is the problem appearing on the modern audio cards, too. The digital I/O interface transfers often non-audio data streams like A52 (so-called AC3) format. Many digital interfaces require the proper set up of the IEC958 (S/PDIF) status bits to indicate several important conditions such as the non-audio bit and the sample rate. This information cannot be passed on the OSS drivers in a consistent way.

The last point comes from the simplicity of mixer abstraction. For example, a matrix mixer, or a state-list element cannot be implemented with the standard mixer API. The only solution for such a non-standard thing is to use a private ioctl.

Pain with Sound Servers

Introducing a sound server solves some of the problems above. For example, the sample rate conversion should

[3] The format where the samples are placed separately in the discrete buffer (or position) for each channel. That is, a group of mono streams is handled as a single multi-channel stream.

be a job of a sound server. However, this doesn't solve everything. There is one important condition. This argument will be valid if (and only if) *all* applications support the same sound server.

Unfortunately the current situation is not ideal. Although more and more applications are born, few of them try to support EsounD or aRts natively. One of the reasons is that there are different platforms and difficult to choose *the only one* solution. And applications can run well even without support of such a sound server when the developer feels the sound server annoying and doesn't use it. This kind of problem would be solved in future once if the desktop system matures, though.

The real problem lying on the sound system in the past is that, again, it was not written for the modern architectures. It works quite well for simply playing a bell over PCM, listening to a music, or recording samples from a microphone. However, if you write a serious high-end audio application which needs to handle the multiple channels with 24 bit samples, there is no way around.

Latency and Synchronization

There are additional things to consider for building a high-end audio system. One of the important factor is latency. The word "latency" may be referred in many different fields. In this text, it means the time delay between the application and the actual I/O. In general, the smaller latency corresponds to a faster response. Thus, getting the smaller latency is extremely important for a real-time audio application.

The latency is introduced in different places. In the case of playback over a conventional sound server, the samples are sent from an application to the analog output through the following path:

1. The digital samples are written on the application buffer.
2. Data are copied to the sound server over socket.
3. Several streams are mixed on the sound server buffer.
4. Mixed data wrote to the device driver.
5. DMA transfer.
6. Through DAC you get analog signals.

The total latency is determined by the buffer size of applications *and* the buffer size of sound server, plus the internal latency of the driver and the chip itself (up to 1 or 2 ms). The problem of the path above is that a certain latency must happen because of the additional buffer on the sound server. Also, the overhead of data copy between the application and the sound server is the another source of latency.

Moreover, the reliability of task scheduling on the Linux is another problem. The response of a process is not deterministic on a multi-process system like Linux. This response can be improved by a real-time (RT) scheduling. However, the RT-scheduling on the standard Linux kernel doesn't perform well under several conditions. For example, the heavy disk access or the graphic access may block the RT-scheduling in the order of 100 ms.

The buffer latency above must be enough large to assure to satisfy this system latency. But how much latency is supposed indeed? In fact, the latency in the real world is also fairly high, but it is not noticed usually. For example, when the listener stands two meter away from the box, there is already latency about 5.9 ms. So, what matters?

The answer is the synchronism. If each stream can be handled independently (e.g. playing a music and a bell at the same time), the acceptable latency should be relatively high. You might not notice a clear difference in the latency of 10 ms or more long time. On the other hand, if data streams must be processed synchronously, or the processed signals may be rerouted, the latency must be enough small.

From this perspective, the mechanism used in the conventional sound servers is not suitable for the synchronized operation. As already noted, it works well for a simple use, but not for the serious high-end audio applications which require the real-time processing and the synchronization of streams.

Connectivity

Another missing feature in the past system is the connectivity. Although the sound servers above already accept multiple access of applications, the connection between applications cannot be changed or configured arbitrarily. Suppose that you run two different applications to output PCM streams and record from a server.

With the past server systems, you cannot switch the connections on the fly.

The same situation can be found in the area of MIDI. The MIDI streams are not handled properly by the sound servers, too. The multiplex access and the dispatching are missing in the conventional system.

And the last missing connectivity is the connection of people. Each project and each audio application was developed independently in most cases, and they were rarely involved with each other.

1.4 Back To The Future

For the better support of audio hardwares, a new sound driver project was started by Jaroslav Kysela and others. The firstly supported card was Gravis UltraSound card only. Lately the project targeted the general support of other cards and was renamed "Advanced Linux Sound Architecture" (ALSA)[5]. The functionality of ALSA has been constantly improved. The range of sound cards supported by ALSA has become always wider including many high-end audio cards. The ALSA drivers were integrated into the 2.5 kernel tree officially as the next standard sound drivers.

The situation around the system latency of Linux kernel gets improved with the recent development. Andrew Morton's low-latency patchset[7] and Robert Love's preemption patchset[8] solved the scheduler-stalling problem significantly. In the 2.5 kernel series, the latter was already included, and the codes were audited to eliminate the long stall in many parts. With the tuned Linux kernel, it's even possible to run the audio system in 1 or 2 ms latency[9].

The change happens also in the community. There have been more and more active communications and discussions over the mailing list. Some hardware vendors have been involved with the development of ALSA, too. The most important results are LADSPA (Linux audio developer simple plugin API) plugins[10] and JACK (Jack Audio Connection Kit)[12]. Both were born and shaped up from the discussion on the Linux audio developer mailing-list[6].

LADSPA is a plugin API for general audio applications. It's very simple and easy to implement in each audio application. There are many plugins for every effect (most of them are written by Steve Harris[11]).

The latter, JACK, is a new sound server for the professional audio applications. JACK was primarily developed by Paul Davis, and employed as the core audio engine of ardour, a harddisk recorder program. It is based on the completely different design concept from the conventional Linux sound servers. It is aimed to glue the different audio applications with the exact synchronization. Together with the ALSA and the low-latency kernel, JACK can run in a very short latency. The number of applications supporting JACK has been growing now.

Finally at this point, we are back to the *present* stage. In the following sections, the implementation and technical issues of ALSA, JACK and other projects related with the sound system are described.

2 ALSA

2.1 Characteristics of ALSA

ALSA was designed to overcome the limitation of existing sound drivers on Linux. In addition to the support of high-end audio cards, the ALSA was constructed with the following basis:

- Separate codes in kernel- and user-spaces
- Common library
- Better management of multiple cards and multiple devices
- Multi-thread-safe design
- Compatibility with OSS

Fig. 1 depicts the basic structure of ALSA system and its data flow. The ALSA system consists of ALSA kernel drivers and ALSA library. Unlike the OSS, ALSA-native applications are supposed to access only via ALSA library, not directly communicating with the kernel drivers. The ALSA kernel drivers offer the access to each hardware component, such as PCM and MIDI, and are implemented to represent the hardware capabilities as much as possible. Meanwhile, the ALSA library complements the lack of function of the cards, and provides the common API for applications. With this system, the compatibility can be easily kept even

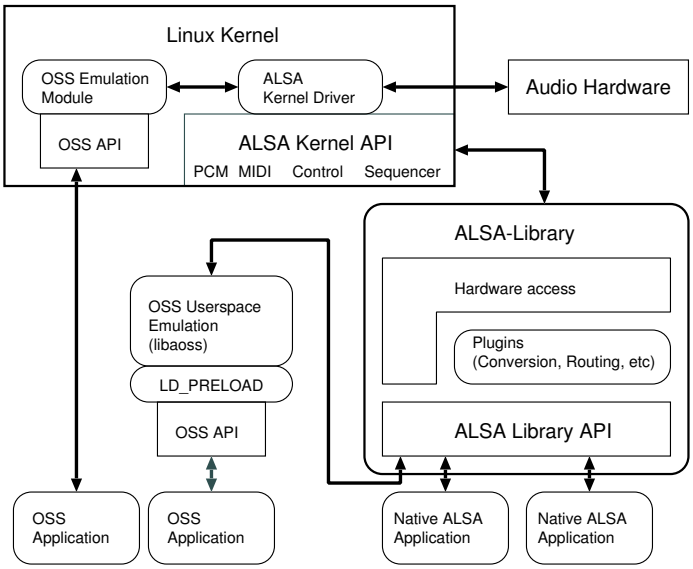

Figure 1: Basic Structure and Flow of ALSA System

if the kernel API is changed, because the ALSA library can absorb the possible internal changes and keep the external API consistent.

The following components are supported by ALSA.

PCM

The PCM is full-duplex as long as the hardware supports. The ALSA PCM has multiple layers in it. Each sound card may have several PCM devices. Each PCM device has two "streams" (directions), playback and capture, and each PCM stream can have more than one PCM "substreams". For example, a hardware supporting multi-playback capability like emu10k1 has multiple substreams. At each open of a PCM device, an empty substream is assigned for use. The substream can be also specified explicitly at its open.

The ALSA supports quite wide range of formats. For example, the linear formats from 8 to 32 bit, 32/64 bit float, non-linear formats like μ-Law and ADPCM. The multi-channel samples can be placed in both the interleaved and the non-interleaved format according to the hardware. The high-end audio cards like RME Hammerfall and HDSP run on the ALSA with the 26 channel non-interleaved mode.

The digital I/O interface is implemented differently according to the hardware. In most cases, an independent PCM device is provided for the IEC958. In other cases, the output type is switched via a control (mixer) switch. Such a difference of implementation is absorbed by the ALSA library's configurator in the user-space level. The IEC958 status bits are accessed usually via a control element as described in the next section.

The PCM streams which are operational synchronously can be linked with each other, so that they can be operated together in the sample-wise accuracy. The linkage of streams is even possible among any streams of any cards. In this case, the accuracy of synchronized operation is not assured, but the application can handle them uniformly.

There is a virtual card (snd-dummy) module which emulates the PCM devices. This works just like `/dev/null` file for playback and `/dev/zero` file for capture. Applications can write PCM data with the given condition to the virtual PCM device but it's never played actually. This function is useful if an application (e.g. a video-game program) requires the audio functionality inevitably but there is no audio device equipped.

Controls

The controls to the card is implemented on the universal control interface. This includes the mixer and card-

specific run-time configurations. The control interface is very flexible to represent the different styles of mixer configuration on different audio cards. In contrast to the simplistic implementation of OSS, the ALSA control interface is implemented to give the all possible functions.

The control elements are managed in a single array (list). Each control element is identified by a name string and an index number. Several different types of data can be handled, such as boolean, integer, enumerated items and byte arrays.

The representation of mixer is dependent on the hardware. Many mixer elements have both an integer element for the volume or the attenuation and a boolean element for the mute switch. The multi-channel volumes can be implemented either in multiple control elements (differently identified by index) or an array in a single control element.

The capture source selection (e.g. line-in, microphone, etc) is one of the difficult cases. If the hardware allows multiple capture sources and mixes them on it, the ALSA also provides the possibility to choose multiple sources. On the other hand, when the hardware has an input MUX, which is an exclusive switch, the ALSA provides usually an enumerated list to let users choose the one.

The digital (IEC958) I/O status bits are stored in the byte array. The applications can access to this control to change the behavior of digital I/O interface, for example, to toggle the no-audio data or consumer/professional data bit. Some bits correlated with the sample rate are changed also by the PCM functions automatically when the PCM sample rate is determined.

MIDI

The raw MIDI byte streams are accessed via a device file. Applications can simply read and write these device files for receiving and sending MIDI byte data. There are several ioctl's for the buffer management and some extra commands. But they are not necessary in most cases.

Sequencer

The ALSA sequencer is a highly abstracted MIDI system. It handles MIDI event packets to deliver to the given destination. It supports the multiplex access. The events can be either scheduled in priority queues for later delivery or dispatched immediately.

When connected to the ALSA sequencer core, each application creates a "client". A client includes one or more "ports". The events are transferred through these ports. In practice, the ALSA sequencer port corresponds to the MIDI port, and 16 MIDI-channels are assigned to each port.

Many applications use the ALSA sequencer as the MIDI dispatching system because of its connectivity capability. Applications can connect to the sequencer system arbitrarily to send or receive MIDI event packets. The connection can be changed dynamically on the fly as a patch-bay.

There is also a virtual MIDI card which converts the MIDI byte stream to the ALSA sequencer event packets, and vice versa. This is useful for the applications which talk only with a raw MIDI device.

Timer

ALSA provides also a generic timer interface. The timer events are informed either via read/poll system call or via asynchronous signal. As default, the system timer and the RTC can be chosen as a global timer source. When the hardware provides more accurate timer sources, the card-specific timer is created, too.

A noteworthy feature is the PCM timer. Each PCM creates a timer which generates the clock at each period[4] boundary, thus is synchronized with the PCM stream completely. This timer is used to control the timing of multiple streams in the dmix plugin, which will be explained later.

Hardware-Dependent Device

The hardware-dependent (hwdep) device is purely driver-specific, and the whole system calls to the device are defined by the driver. Any non-standard features such as the firmware or DSP loading can be implemented on this device.

[4]The *period* is the size at which the hardware generates the interrupt during the DMA transfer of audio data. It's called *fragment* in OSS.

2.2 ALSA Drivers

Basic Design

The ALSA kernel drivers consist of three layers. The low-level layer corresponds to the functions which access to the hardware and is written as callback functions. The middle-level layer is the core part of ALSA drivers including the common routines for each different component (PCM, etc). The top-level layer is the entry point for each card which creates the device entries with the callback table to the corresponding low-level functions. Because of this separation, a driver developer can concentrate on the top and low-level access functions and forget about the whole complicated data flow.

Each card entry and the related hardware components are hold in the common container struct (`snd_device_t`) uniformly. All components are managed in a list assigned to each card, so that all components can be traced from the top-level. This mechanism helps for the consistent destruction and for controlling the disconnection of devices via hotplug. When a device is disconnected, the ALSA driver swaps the file descriptor table for that device in order to prevent the further access to it, and then calls the disconnection callbacks of all assigned components. The destructor will be called eventually from a workqueue which waits until the jobs of all component are finished.

The design of ALSA kernel API is also based on a traditional Unix style. The hardware is accessed via normal open, close, read, write and ioctl system calls. The readv and writev system calls are offered also for the non-interleaved PCM samples to access efficiently in the vector form. The poll system call is implemented in all components, too.

Memory Mapping

The memory mapping is the most efficient method to transfer the data between user-space and kernel-space. It works like a charm. However, it also has some disadvantages:

- Not all hardwares support mmap.
- The buffer size is restricted by the hardware. The possible size is different on each chip.

Thus, when applications use large buffers (for example, an MP3 player), the mmap is not suitable. Rather a simple read/write makes the code cleaner. The mmap gives the best performance for real-time applications with small buffers, which are sensitive to the latency.

Like OSS drivers, ALSA driver provides the mmap capability, too. The application can map the DMA buffer of the driver onto the user-space, so that the data transfer is done simply by writing audio data on the buffer without extra copy.

In addition to this normal mmapped buffer, ALSA maps also the control and the status records onto the user-space. The control record contains the current sample position at which the application is processing (called "application pointer" in ALSA). The status record contains the current sample position at which the DMA is processing (called "hardware pointer") and the current status. With this mapping, the context switching between user and kernel modes can be reduced dramatically, since the application can read and write directly the current state of the driver. In theory, if the audio thread runs in its own accurate timing, it can stay in the user mode and never needs to call any system calls. However, usually an application needs to call `poll` to get synchronized with the real time.

The PCM capture buffer is mapped not in read-only but read-write mode. This condition is required because many applications need to write the data on the capture buffer to mark up the buffer position. Due to this requirement, the playback and the capture streams are divided to different device files.

PCM Configuration

The PCM configuration is one of the most complicated part in the ALSA drivers. Since each hardware has its own limitation, the application needs to negotiate the proper configuration, such as, which type of format, which sample rate, how big the buffer size is or how many periods are allowed. The ALSA has two different configuration types. One is called "hardware parameters" (hw-params), which are the fundamental properties for the PCM stream, such as the sample format, sample rate, number of channels, the buffer size, the period size, etc. Another is called "software parameters" (sw-params), which are optional properties for the precise controls. For example, how the driver be-

haves when underrun/overrun (called "xrun" in ALSA) occurs, or at which timing the DMA starts.

The difficulty of this configuration is that there are different types of limitations depending on the hardware. Some chip supports only certain sample rates, and some supports the rate based on a certain clock only. Or, in a more complicated case, the possible number of channels changes with the defined format and sample rate, etc, etc.

These conditions are defined as "constraints" in the driver. At each time a certain condition is changed, the parameter space is evaluated throughout the defined constraints and reduced to the possible values. Finally, after all conditions are given by the application, the best condition is chosen from the parameter space.

Buffer Management

The DMA buffer is supposed to be physically continuous on many devices. The fragmentation of memory pages is one of the unsolved problems on a system like Linux. When a kernel runs for a long time, the memory pages are fragmented and it becomes difficult to allocate the continuous pages which are enough large for the DMA buffer.

Also, many chips have the limitation of upper memory area for the DMA buffer. ISA cards use the memory under 16MB address. And some PCI chips have 28 or 30 bit limit. The allocation of pages is more difficult in such a case.

For solving this problem, ALSA provides a special module, snd-page-alloc. This module is independent from other ALSA modules and can be loaded in the early boot stage where the pages are not fragmented severely yet. When loaded, it checks the PCI (and other) device entries whether the pre-defined devices are found. If any matching device is found, the module tries to allocate the buffers in advance.

This module also manages the buffers allocated later by the ALSA card modules. The buffers are reserved even after the ALSA card module is unloaded, so that the same buffers can be used for the next reload. This mechanism prevents the buffer exhaust phenomenon when the ALSA modules are loaded/unloaded automatically via kmod.

A few audio chips can use the scatter-gather (SG) buffer. ALSA also supports this type of buffer with help of kernel paging mechanism. The driver and application can access to the buffer linearly through the virtual memory.

2.3 ALSA library

ALSA Plug-ins

The ALSA library is located between the ALSA drivers and the applications, and it works as the entrance to the ALSA system. It provides the consistent ALSA library API for controlling the devices. However, the role of ALSA library is more than that.

As shortly mentioned, ALSA library plays a role to fill the gap between the required function by the application and the supported function by the hardware. This is done by so-called "plugins". The plugin is a dynamically loadable object. There are many different standard plugins provided in the ALSA library. Many of them work for the conversion of data, for example, the sample format, the sample rate, number of channels, interleaved and non-interleaved formats, etc. When application requests a configuration which is not supported by the hardware, ALSA library loads the necessary plugins automatically and does the conversion in run time. Both the hardware-native and the conversion cases are handled transparently. Hence, the application doesn't have to know what is being done in the ALSA.

The plugin works not only for the conversion but also as the user-space driver. For example, applications can access to different systems like JACK or IEEE1394 (FireWire) consistently via ALSA library API. The difference of API is hidden inside the plugin.

An advanced use of plugin is the combination of PCM devices. The several different PCM devices can be combined virtually as one PCM device. For example, you can build a virtual 5.1-channel PCM device combined from three different cards where each of them supports the two channel stereo playback.

ALSA Library API

The ALSA library API is designed to express the ALSA hardware abstraction straightforwardly. Thus, the library API itself doesn't add any new features. The library is in fact a wrapper to the system calls for the direct hardware access. But the application can use the

same API for the access via plugins, too. This is one of the purpose of ALSA library.

Currently the ALSA library API is provided only in C. The syntax of ALSA library API is unique. It's based on the opaque struct style. Most of all structs referred in the ALSA library API are not defined but only declared. Each function takes only the struct pointer as arguments. For example, the typical code to open a PCM device would be like below:

```
int err;
snd_pcm_t *handle;
err = snd_pcm_open(&handle,
   "default", SND_PCM_PLAYBACK,
   0);
```

where the handle struct `snd_pcm_t` is nowhere defined. Another example is the code to retrieve the current status of a PCM stream:

```
snd_pcm_status_t *status;
snd_pcm_status_alloca(&status);
snd_pcm_status(handle, status);
current_state =
   snd_pcm_status_get_state(status);
```

Here the status record struct `snd_pcm_status_t` is again not defined. It is allocated dynamically via `snd_pcm_status_alloca()` function, and the struct field `state` is retrieved by `snd_pcm_status_get_state()` function indirectly.

This strange behavior is chosen for the future extension. By hiding the struct definition inside the library, the binary-compatibility of the API is kept consistently even after the definition of the struct is changed in future. Because of this way of implementation, the functions become fairly redundant. This can be more better and elegantly expressed once if the API is written in C++.

2.4 OSS Compatibility

One of the most important issues in the development of ALSA is the compatibility with OSS. In fact, ALSA can emulate the OSS API quite well.

As found in Fig. 1, there are two routes for the OSS emulation. One is through the kernel OSS-emulation modules, and another is through the OSS-emulation library. In the former route, an add-on kernel module communicates with the OSS applications. The module converts the commands and operates the ALSA core functions. This route is easy to set up and works effectively in most cases.

In another route, the OSS applications run on the top of ALSA library. The ALSA OSS-emulation library works as a wrapper to convert the OSS API to the ALSA API. Since the OSS application accesses the device files directly, the OSS-emulation wrapper needs a hack using `LD_PRELOAD` environment variable. The OSS-emulation library is then pre-loaded for the OSS application so that it replaces the system calls to the sound devices with its own wrapper functions. In this route, the OSS applications can use all functions of ALSA, including the plugins and mmap.

In the former route, some high-end audio cards don't work properly because of the difference of hardware configurations. Although the kernel OSS emulation module has also conversion functions, only the minimal subset is implemented there. In the latter route, on the other hand, the difference can be reduced with the help of plugins in the ALSA library. The application can use the mmap mode even on the hardware with the non-interleaved formats. However, this route requires a dirty hack using `LD_PRELOAD` as its cost, instead.

3 Evolution of Sound Servers

3.1 JACK

Callback Model

The JACK is designed for the high bandwidth data transfer. This means also that it is not intended to run on the network system. It's rather a base for a desktop audio workstations (DAW). The primary goal of JACK is, as already mentioned, to achieve the sound server for the professional audio applications. This implies the following requirements:

- Synchronized operations in a low latency
- Highly flexible connectivity among application

For performing the audio transfer with the exact synchronization, JACK system is based on the callback

model. This is another programming paradigm in comparison with the traditional Unix style model. In this model, each audio processor (called "client" in JACK) is executed passively at the moment when the audio data must be handled on the buffer immediately. For the playback, the callback is invoked when the audio data is to be filled on the buffer. For the capture, the callback is invoked when the audio data is ready to read from the buffer.

On the system based on the traditional style, each applications read or write whenever they want. Thus, the audio streams are handled without synchronization. In the case of callback model, on the contrary, the synchronization of data-flow is assured in the sample-wise accuracy since the timing to call each callback is determined by the server. The same strategy is taken in other modern audio systems like CoreAudio[13] or ASIO[14].

JACK handles only the 32bit float as its audio data unlike the other conventional sound servers. Also, it assumes only the non-interleaved format, i.e. mono streams for multi channels. These differences may make the porting of existing audio applications a bit harder, in addition to the difference of programming style described above.

Basic Structure

Fig. 2 shows the basic flow diagram of the JACK system. There is a JACK server daemon running as a central sound server, and each application accesses to the JACK server as a client. A client has several ports, and the connection between ports are flexible and can be changed on the fly. This concept is quite similar with the ALSA sequencer.

The JACK sound server communicates with the sound engine (e.g. the sound driver) exclusively as well as other sound server systems. The JACK supports several platforms for the sound engine. In addition to the ALSA, PortAudio[15] and Solaris drivers are implemented currently.

JACK has two different types of clients: internal and external clients. The internal client is a kind of plugin. It's a shared object running inside a JACK server. On the other hand, the external client is a thread running independently. This multi-thread communication is the advanced and unique feature of JACK, which is not seen in CoreAudio or ASIO.

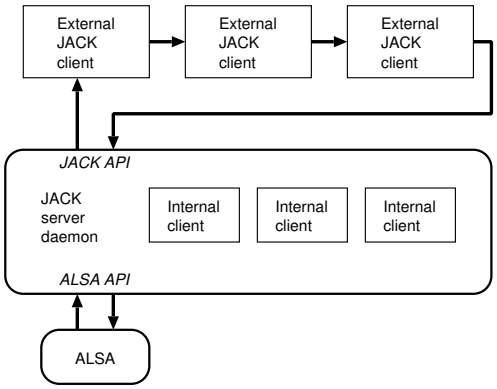

Figure 2: JACK flow diagram

Being executed in a server process, the internal client runs without context switching, and thus it's more efficient than the external client from the perspective of performance. For the development perspective, an external client is easier to develop and debug, because it is in fact a part of the application. Also, it's more tough against the unexpected program failure.

The data transfer in JACK is very efficient. Basically, JACK tries to implement the "zero-copy" data transfer. The JACK server runs in the mmap mode for the data transfer between JACK server and the ALSA. The audio-data buffer between a JACK server and a JACK client is shared via the IPC shared memory. Only the control messages are communicated through FIFO of a normal socket.

In the callback model, each JACK client is executed as an FIFO sequentially. The execution path of clients follows the connection graph. When a client callback is called, it reads the data from the JACK server or another client, processes it, then writes to the next. With this style, the change of the graph connection/disconnection can be handled more easily.

JACK API

JACK provides a highly abstracted API for the audio applications. It's very simple and dedicated to the callback-style programming. Since the low-level configuration for the audio driver is determined in the JACK server level, JACK clients don't have to deal with such a subtle thing. Instead, each client needs to concentrate on the creation, destruction and connection of

ports, and the process of the audio thread itself. A thread is created automatically when an external client starts.

The JACK application developer needs to pay attention to the multi-thread programming. A JACK client is supposed to run always on the multi-thread. The data should be handled with lock-free methods, and the audio thread must not be blocked by others.

Unlike the ASIO or other systems, JACK accepts more flexible hardware configuration. While ASIO uses only two periods (called "double buffer") to get the ideally lowest latency, JACK can handle arbitrary number of periods with the arbitrary size (which are specified by the command line options of JACK daemon). Some audio chips like YMF-PCI prefer more than two periods to run stably because of the chip design. Such a chip is often designed to be driven via timer interrupts instead of DMA-transfer interrupts. Anyway, these difference of hardwares can be hidden inside the JACK server, and applications are free from the hardware restrictions.

3.2 ALSA dmix / dsnoop plugins

The ALSA dmix and dsnoop plugins are one of the advanced features which have been recently added. The letter "d" in dmix and dsnoop stands for "direct". These plugins preform as well as a normal sound server do, that is, sharing the PCM device among different applications for mixing, multiplexing and loopback. However, they are not quite a sound server. More exactly writing, these plugins do not require any server process unlike the conventional systems. In the dmix/dsnoop plugin, the hardware buffer itself is shared by all applications "directly", instead of exclusive control by a server process.

In the conventional server system, there is a central server process, and applications access to it for mixing and multiplexing the stream. The server gathers the data from connected applications, mixes them and then sends to the sound driver. Because of the gather-and-mix strategy, there must be a significant latency. And the RT-scheduling, which is necessary for reducing the latency on the Linux system, may also lead to permission and security problems.

In the dmix/dsnoop plugins, on the other hand, each application writes/reads the shared hardware buffer by itself concurrently. Thus, there is no latency introduced by the mediation of a sound server. Every application can play and capture the audio data with very low latency even without the dedicated RT-scheduling.

For sharing the PCM hardware buffer, the dmix/dsnoop plugin employs simply an mmap method. For the synchronization of the processing pointer (also required during the draining), the ALSA timer interface is used. Since the destination PCM stream is referred as the timer source, the timing to update the period can be accurately informed to applications.

The dsnoop plugin is a loopback/multiplexer for the capture, and applications can simply copy audio data from the shared hardware buffer. The another, dmix plugin, is a dynamic mixer for the playback, and this needs more complex handling like below because of the saturation problem.

The dmix plugin has a common sum-buffer in addition to the hardware buffer. This sum-buffer is shared via IPC shared memory, too. For ease of calculations, the dmix plugin handles only 16 and 24 bit integer samples. The data in the sum-buffer is always 32bit integer.

The basic idea of dmix plugin is that each application adds its own sample to the existing sample value on the shared hardware buffer dynamically. If the resultant sample value overflows the 16/24 bit range, it must be saturated. The sum-buffer is used for this check. The task to compute the addition and the saturation-check becomes complicated because of the possible concurrent access to the buffers. Two atomic operations are required at least: one for the addition and one for the comparison. The typical add-and-saturation code would look like below:

```
sample = *src;
old_sum = *sum;
if (*dst == 0) /* atomic operation */
    sample -= old_sum;
*sum += sample; /* atomic operation */
do {
    old_sum = *sum;
    sample = SATURATION(old_sum);
    *dst = sample;
} while (*sum != old_sum);
```

where `dst` is the shared hardware buffer pointer, `sum` is the sum-buffer pointer, and `src` is the source buffer. In the first atomic operation, the sum-buffer is initialized,

and then the current sample is added on the sum-buffer. The succeeding do-while loop performs the saturation and the substitution to the hardware buffer.

The demerit of the dmix plugin is that this lock-free algorithm is more expensive than the implementation in the server-client model. Also, because of the atomic operations, it's not implemented on all architectures yet.

Another note is that the dmix/dsnoop plugins are not designed for the arbitrary connections. They are provided as an alternative to the simple sound server. Although the CPU consumption of dmix plugin is higher than the others, these plugins are promising for normal consumer purposes, because the ALSA applications can run through these plugins without rewritten at all.

3.3 Other Projects

The sound servers for the network system is still an open question. For the generic sound server belongs to the traditional style like EsounD and aRts, there is a new project, MAS (media application server)[16]. The design of MAS is similar with the conventional sound servers, based on the open/read/write model. It is a modular system like aRts. The advantage of MAS is its tight binding with the X server. The MAS supports several different protocols for the remote data transfer. The protocol can be chosen depending on the network condition. In the worst case, the data can be even tunneled through the X protocol.

Since MAS is still under heavy development, it's still too early to evaluate the performance with other advanced systems. However, this looks like a good solution for the environment with many machines connected over a LAN.

Another interesting system having the similar concept is GStreamer[17]. This is aimed to achieve a universal platform for the whole multimedia including both video and audio. The system is monolithic and each module is implemented as a plugin shared object, as well as many other systems. There are large number of modules already available on GStreamer.

In the area of the broadcasting of audio (and video) streams, a promising project is Helix DNA[18]. This is a collaboration work between the open-source community and the big player like RealNetworks. Its aim is to provide the comprehensive cross-platform for the boradcasting system, including its standard API and the client/server applications. The project is still in the early development stage, and the support of ALSA is not finished yet.

There are so many different things for the single theme — the sound system. And you might have the question: Which platform should we choose? This is, indeed, a very difficult question to answer. The key for the possible solution is to define a portable API. The PortAudio[15] is a good candidate for this. It supports very wide range of OS and sound systems (including both ALSA and OSS) with the same consistent API. PortAudio provides both the traditional and the callback style APIs. Thus, it can be a suitable solution to melt the solid gap between the different architectures like JACK and OSS.

4 The Future: a.k.a. TODO

4.1 Problems of ALSA

Now, let us take a look at the current problems and what will (or should) happen in the future. Regarding to the ALSA, first of all, what should be done is to improve its usability. The current ALSA is full of complexity.

There have been many rumors that the ALSA is so difficult to set up compared with the OSS. It's partly true — there are more configurations (e.g. in /etc/modules.conf) necessary to make the system running properly (although such settings are not always required if one runs only ALSA native applications). In a convenient future, such a problem will disappear when all the major distributors support ALSA primarily.

Besides, there are still more things to consider in this regard. The ALSA is a hardware abstraction. The ALSA covers the hardware capability as much as possible. This policy sometimes leads to the too complex representation. For example, some chips show hundreds of controllable mixer elements, and it's of course too much for end-users. More higher abstraction is needed for the more intuitive usage.

Also, there is no user-friendly editor for the ALSA configuration files, and no enough example configuration files provided. There is no good way to set up the IEC958 status bits, too. This situation tends to prevent the user to try the advanced features, unfortunately.

Finally, the lack of documentation has been a major problem of ALSA project. This is the most important thing to be fixed in the future above all.

4.2 On JACK

JACK is promising and many applications are supporting it in fact. However, it's still not prefect in every case. The most frequently appearing problem is its response and stability.

Even though JACK is designed to cooperate with the ALSA tightly, it happens sometimes that the system takes too long response time than expected. This results in the xrun on the JACK system. In most cases, it's the configuration failure for the certain sound chips. But it also lies on the JACK mechanism, too.

Because the process of audio data is executed through the connection graph, the whole system is blocked once when the execution of a client is stalled by some reason. This phenomenon may be reduced when more JACK clients are written as the internal clients. The internal clients would have no task switching problems like above. Also, the performance of JACK on the SMP system can be more optimized by the dynamic evaluation of execution paths.

As well as the ALSA suffers, a similar problem remains on JACK, too. It's not easy to provide the ideal configuration and the ideal environment for the JACK system. This is not JACK's fault, but rather related to the Linux distribution. Since the RT-scheduling might be involved with the permission and the security problems, it cannot be used so conveniently in the major Linux distributions, which are more sensitive to these problems. This will be hopefully improved in the future, once when JACK is recognized as a standard system for the professional audio applications.

4.3 Future of Audio Devices

The present will become the past when time continues to proceed. Surely, the audio device will be kept improved in future, too. Here, as the final note, the author would like to try to describe some expectations of the future audio devices and the possible problems. The following different directions can be considered depending on the demand.

More Features

The hardware will be equipped with more features such as the hardware encoding of high-compression format like MPEG or Vorbis, or the hardware encoding of the multi-channel formats (A52, DTS). Together with the use of broadband network like ADSL, the broadcast of video/audio streams will be no longer only for professionals.

This will bring us a problem. The handling of encoded formats is not yet realized well on the ALSA. Since the encoding format usually takes the variable bit rate, a change or an extension of the current model would be needed in near future, too.

The structural audio might become an interesting field in the future. This is analogy to the 3D support happening in the recent development of graphic cards. Instead of rendering 3D graphics, the sound effects like 3D spacializing may be implemented on the chip when the surround system becomes more popular.

Higher Quality

The surround system will be more complicated, and more channels will be used such as 6.1 or 7.1 channels. The 6.1 speaker system and the audio cards supporting such an environment came just in the consumer market at this moment.

The more important improvement for the quality is the support of 24 and 32 bit samples. Also more audio cards may support the float format, too. The sample rate will be also based on 96 or 192 kHz as standard.

These increase of audio spec may be dependent on the popularity of the new media, such as upcoming DVD-audio and SACD. As long as people are satisfied with the old-good CD, the drastic change in this regard might not happen, though.

In any cases, this direction of changes wouldn't be a big problem for the existing system. Most of features have been already implemented, and enough tested on the advanced audio cards.

More Convenience

Hotplug devices will be more popular. At this moment, it is fairly difficult to guess whether FireWire or USB2

dominates. But both specs are enough high to replace the existing PCI audio cards. In the future, only the on-board (integrated) audio chip and the hotplug devices might remain in the market.

From the viewpoint of Linux system, this would require the better support of user-space drivers. This implies the reliable kernel scheduling to assure the exact timing for the isochronous transfer on user-space.

References

[1] 4Front Technologies, OSS/Commercial homepage: http://www.opensound.com

[2] 4Front Technologies, OSS programmers guide, v1.1: http://www.opensound.com/pguide/oss.pdf

[3] EsounD project homepage: http://www.tux.org/~ricdude/EsounD.html

[4] aRts project homepage: http://www.arts-project.org

[5] ALSA project homepage: http://www.alsa-project.org

[6] Linux Audio Developers (LAD) mailing-list homepage: http://www.linuxdj.com/audio/lad/

[7] Andrew Morton, low-latency patchset for 2.4 kernels: http://www.zip.com.au/~akpm/linux/schedlat.html

[8] Robert Love, preemptive patchset: http://www.tech9.net/rml/linux/

[9] Resources about low-latency at LAD site: http://www.linuxdj.com/audio/lad/resourceslatency.php3

[10] Richard Furse, Linux Audio Developer's Simple Plugin API (LADSPA) homepage: http://www.ladspa.org/

[11] Steve Harris, SWH Plugins: http://plugin.org.uk/

[12] JACK project homepage: http://jackit.sf.net

[13] Apple Computer Inc., CoreAudio: http://developer.apple.com/audio/coreaudio.html

[14] Steinberg, ASIO developer information area: http://www.steinberg.net/en/ps/support/3rdparty/asio_sdk/

[15] PortAudio homepage: http://www.portaudio.com/

[16] MAS project homepage: http://www.mediaapplicationserver.net

[17] GStreamer project homepage: http://www.gstreamer.net/

[18] Helix Community homepage: https://www.helixcommunity.org/