Boomer:

The New OpenSolaris Audio System

Presented at Kernel Conference Australia 2009

Garrett D'Amore – garrett.damore@sun.com

July 2009

Abstract

This paper describes Boomer, a new audio subsystem for Solaris and OpenSolaris. Boomer provides a new API for audio applications, a new in-kernel mixing framework, a simpler DDI for audio device drivers, new multimedia features, enhancements to audio performance and scalability, and backwards compatibility for existing audio applications.

Table of Contents

Overview....................1
Introduction....................1
Solaris 10 Audio....................1
4Front Technologies and Open Sound System....................1
Open Sound System API....................2
Legacy Sun API....................2
GStreamer....................2
Performance and Scalability....................3
Application Programming Interfaces....................4
Sun audio(7I)....................4
Limitations and Issues....................4
Boomer Solutions....................5
Plans for the API....................5
Open Sound System DSP API....................5
Fragments....................6
Latency Considerations....................6
Confusion with Mixer API....................6
Plans for the API....................6
Enumeration API....................7
Mixer API....................7
Cross Device Security....................8
Free Form ASCII Names and Values....................8
Boomer Architecture....................9
Major Subsystems....................9
Audio Core....................9
Samples, Bytes, and Frames....................9
Client Personality Interface....................10
Character API....................11
STREAMs Module (austr)....................11
Sample Rate and Format Conversion....................12
Mixing Layer 12
Soft Volume Attenuation....................13
Multiple Channel Support....................13
Sun Personality....................13
Open Sound System (OSS) Personality....................15

Audio DDI....16
Audio Control Framework....17

Common AC'97 Module....17

Specific Devices....18
Intel High Definition Audio....18
USB Audio....19
SPARC Devices....19
AC'97 Devices....20
New Devices....20
Sun Ray Ultra-Thin Clients....20

Boomer Device Driver Interface (DDI)....20
Fragments....21
Structures....21
audio_dev_t....21
audio_engine_t....21
audio_engine_ops_t....22
audio_ctrl_t....22
audio_ctrl_desc_t....22
Functions....23
Entry point initialization....23
Audio Device Allocation....23
Audio Device Information....24
Audio Engine Management....24
Audio Engine and Device Registration....24
Data Management....24
Audio Control Support....25
Miscellaneous Functions....25
Entry Points....26
Opening and Closing the Engine....26
Starting and Stopping....26
Engine Positioning....26
Data Format Handling....27
DMA Synchronization....27

Future Directions....27
Sun Ray Audio....28
Virtualized Audio....28
Sun Ray....28

Hot Plug Support....................28
Secured Virtual Environments....................29
Further Driver Improvements....................29
BrandZ Improvements....................29
Dolby Digital Support....................30
Community Involvement....................31

Chapter 1
Overview

Introduction

The Boomer audio subsystem is a new framework for supporting audio devices in Solaris and OpenSolaris. It has been designed to support current generation multimedia audio applications and devices, as well as future devices and applications.

Integrated into OpenSolaris build 115, Boomer includes support for a number of new devices, expanding both device and platform support for audio. It also provides the long-desired ability to support multichannel sound, such as 5.1 and 7.1 surround sound, even on older systems such as the original Sun Ultra 20.

Boomer also supplies the Open Sound System API which has been made popular on Linux and FreeBSD, with the consequence that many additional multimedia applications can now be supported on OpenSolaris much more easily.

Finally, Boomer provides a new Device Driver Interface for audio device drivers, so that writing a new device driver or porting one from another operating system can be done with only a modest amount of effort, which should help to further increase the number of audio devices supported by Solaris.

Solaris 10 Audio

Solaris 10 (and earlier releases, as well as older releases of OpenSolaris) use a second generation audio framework known internally as “SADA” (Sun Audio Device Architecture). Developed in the mid-1990's, this framework (and the supporting API) was designed to support the then-current audio devices and features, without much thought given to future directions for growth. Most of these devices were simple stereo devices, with only a few possible output options. A few of them had support for mixing multiple streams (“multi-stream codecs”), and the framework was designed with this in mind, although no drivers actually took advantage of it. AC'97[1] and the advanced features that came with it, was still not yet part of the landscape.

4Front Technologies and Open Sound System

In order to address the various shortcomings in Solaris audio, one company, 4Front

[1] AC'97 Component Specification, http://download.intel.com/support/motherboards/desktop/sb/ac97_r23.pdf

Technologies[2], developed a replacement audio framework for Solaris (and other UNIX systems) using its own in-house Open Sound System technology. While 4Front's OSS addresses some of the issues, and many people are happy to be able to use it, it was designed with different goals in mind than Boomer – most especially the ability to support many operating systems, even at the expense of supporting various features found on native Solaris.

Sun and 4Front have teamed together to deliver a superior product for OpenSolaris than had previously existed.

Open Sound System API

Originally developed for Linux, the Open Sound System API has found widespread acceptance in the open source community. It is the native API for FreeBSD. It is emulated by a number of other audio frameworks, and supported by nearly all open source multimedia applications. Even though it has been replaced by the ALSA (Advanced Linux Sound Architecture) on Linux, the Linux kernel still provides an emulation layer for OSS applications. By supporting the OSS API, Boomer opens the door to allowing a vast number of these applications to be ported to OpenSolaris – often by doing nothing more than a recompile.

Legacy Sun API

Of course, Sun has a commitment to supporting legacy applications, and provides a compatibility guarantee for applications[3]. Boomer provides full compatibility for the most common audio applications on Solaris, so old binaries of *RealPlayer*[4], *MPlayer*[5], etc. all work just like they always have. (The only exception is for mixer panel applications like gnome-volume-control. Those applications need to be converted to the OSS API.)

GStreamer

GStreamer[6] is the Gnome[7] multimedia framework, and is used by many Gnome applications to access both audio and video features. As part of the Boomer work, Sun has updated the OSSv4 GStreamer plug-in, and is now committed to doing the work to make it a standard part of the GStreamer "good" release. This means that GStreamer applications can immediately benefit from the improvements in Boomer.

---

2 http://www.opensound.com/
3 See the Solaris Application Guarantee, http://www.sun.com/software/solaris/programs/guarantee.xml
4 http://www.real.com
5 http://www.mplayerhq.hu/
6 http://www.gstreamer.net/
7 Gnome is the graphical desktop environment shipped with OpenSolaris; see http://www.gnome.org/

The improvements also include changes in the *gnome-volume-control* application, which allow it to access a much richer set of device features and capabilities, such as shown in Illustration 1.

These improvements have been fed upstream to the GStreamer project itself.

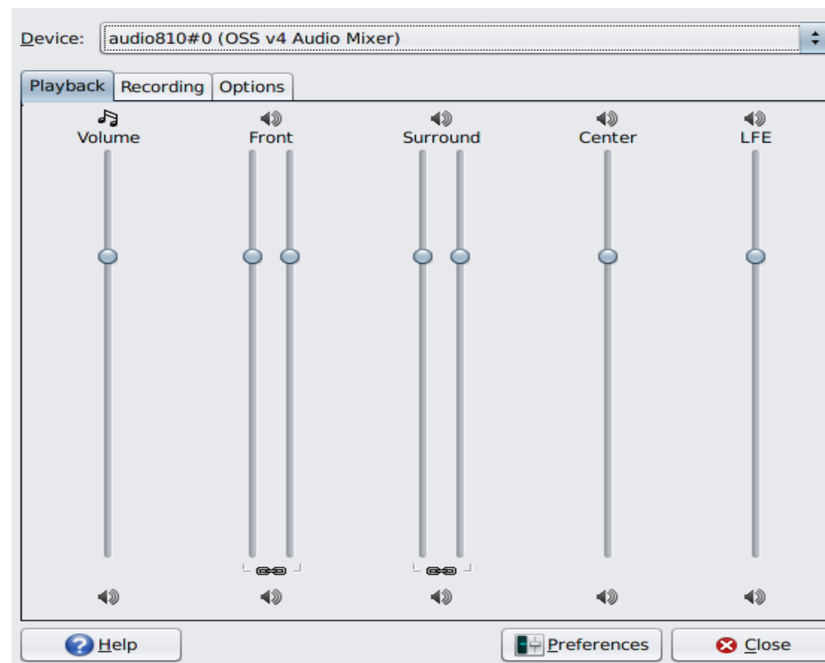

Illustration 1: gnome-volume-control with Boomer

Performance and Scalability

Processing audio can be fairly demanding. One of the most demanding portions commonly encountered is “Sample Rate Conversion” (SRC). The mathematics behind this are complex and well outside the scope of this paper, but the concept is fairly easy to understand. If one has an audio device sampling audio data at a given rate (say 48 kHz, a common rate for audio devices), and then wants to play an audio file that was sampled at a different rate (say 8 kHz), a conversion is required. Historically, Solaris had a converter that could handle all the typical transforms, but did so inefficiently. For Boomer, Sun introduced the GRC3 converter from 4Front Technologies, and further optimized it, giving what is believed to represent the state of the art in software based sample rate conversion.

Additionally, the audio system must be able to support the Sun Ray architecture, in which hundreds of Sun Ray terminals may be connected to a single Sun Ray server. In this case, literally hundreds of audio devices may be present on the system. While this configuration is not yet supported, it is believed that Boomer is capable of scaling to meet this demand.

As part of Sun's in-house testing, Sun performed tests where literally hundreds of audio

applications were playing audio data at the same time, on typical desktop hardware. Ultimately, it was found that memory was a bigger resource bottleneck than the CPU.

Chapter 2
Application Programming Interfaces

There are several interfaces available to applications to use with Boomer for audio services.

Sun **audio**(7I)

This API is the legacy API that applications developed for earlier Solaris releases must use. Sometimes supplemented by a **mixer**(7I) API as well, the interface at first seems simple, but in practice it is quite complicated to use for real-world multimedia applications.

The application is based on STREAMs, and to make full use of its features applications must be prepared to issue STREAMs `ioctl`(2) commands. (In particular, the `I_SETSIG` command is used to setup progress notifications to applications.)

Limitations and Issues

Everything a typical application cares to access for an audio device, including audio format, sample counters, sample rates, volume, and available ports, is configured using the `AUDIO_SETINFO` ioctl. (And these can be read using the `AUDIO_GETINFO` ioctl.) This has several limitations. First, using a fixed structure, `struct audio_info`, limits the extensibility of the interface. For example, only certain kinds of ports and device features can be expressed using this API – it is impossible to express a the notion of jack re-tasking using this API. It is also not possible to express the notion of monitoring several different sources each with different volume levels, yet many devices (nearly all AC'97 devices) can do this.

The second problem comes from applications that incorrectly use this API. A typical bug is to use `AUDIO_GETINFO` to query information, modify it, and send the results back down. The problem here is that there are some tricky semantics surrounding changes to the sample counters, and it is easy for progress tracking to get confused, even when all an application wants to do is change the volume. (To be fair, applications are supposed to initialize all fields not being changed to -1, but again, many applications get this wrong. The API encourages buggy programming.)

The problems become much more complex when one considers the mixer, where multiple applications need to retain their own logical view of the device. Some settings, such as volume levels, might need to be managed on a global basis, while others, such as sample counters, should be uniquely tracked by each application.

To make matters worse, applications are expected to do this through two open file descriptors,

one on /dev/audio (which is where write() and read() for audio transfers take place) and one on /dev/audioctl. There are some tricky rules and undocumented conventions about device sharing, and associating open /dev/audioctl nodes with corresponding open /dev/audio nodes. This was a particularly buggy area of the original Solaris code.

Boomer Solutions

To simplify all this, the entire API implementation was rewritten from scratch. The **mixer**(7I) and **audio_support**(7I) APIs were scrapped (they were not guaranteed to work anyway, were never implemented for Sun Ray, and very few applications actually used them) and implementation was changed so that each device is presented with a single virtual audio device for each physical audio device in the system.

These virtual audio devices also present a virtualized notion of the playback and capture volumes. Changes to the volume are taken to be relative to the master hardware volume, and affect only the application from which they are made.

Additionally, access to physical parameters corresponding to port configuration and similar settings (such as monitor gain) have been removed or stubbed out.

The upshot of this is that desktop volume control applications and mixer panel applications cannot be supported in Boomer using this API. But the resulting code simplification (and corresponding elimination of bugs) justifies this change.

Plans for the API

This legacy API is accessed from C programs by including the file <sys/audio.h>.

While Boomer continues to support this legacy **audio**(7I) API, it is Sun's hope that application authors will steer away from it and use the Open Sound System APIs instead. Sun is not yet marking this API officially Obsolete, because it is still (for now) the only way to access audio features on Sun Ray thin clients. Once Boomer support for Sun Ray thin clients is complete, that barrier will be removed and Sun will officially deprecate this API, even though it will probably continue to ship in Solaris and OpenSolaris for the foreseeable future.

Open Sound System DSP API

The DSP API, intended to be used by applications using /dev/dsp, is the new API introduced with Boomer. While new to Sun, it has been around on other operating systems for quite a long time. This was the original API used for accessing sound card hardware on Linux. It has evolved considerably since those humble beginnings.

The OSS DSP API uses a simple character device driver interface, with separate ioctl(2) calls

for each type of configuration change. The recent version 4.x changes to this API include separate calls to adjust per-application playback volume (SNDCTL_DSP_SETPLAYVOL) and record gain (SNDCTL_DSP_SETRECVOL), which make it easy for an application to adjust volume levels without worrying about the various device details.

The API also supports mmap(2), although Boomer has not yet implemented it. In the future it will be possible to use this mmap API without the restrictions placed upon other implementations.

Fragments

The OSS API is designed around the notion of fragments, and some applications have historically abused this notion. Originally all devices had fixed power-of-two fragment sizes. Modern devices can use any fragment size, or might not guarantee any particular fragment size. This has resulted in various application bugs. Applications that have these bugs have them not just on Solaris but on other operating systems as well.

Latency Considerations

The other problem is that there is no notification to processes to report progress. Some applications use blocking write or read calls to track progress, but even this has trouble when the framework is willing to buffer up more data than the application would like (the additional latency can come as a “surprise” to applications.) In Boomer, two approaches are used. The first is to allow applications to provide Boomer with a “hint” of how much latency they need (and Boomer tries to respect that hint within reason) via the SNDCTL_DSP_SETFRAGMENT call. The second approach is a recommendation to application developers to use non-blocking read or write calls, and only buffer up as much data as they are prepared for. The former approach has so far been more successful than the latter.

Confusion with Mixer API

Yet another issue with the legacy DSP API has been the fact that it is been intermixed with a mixer or device configuration API. While this is fixed with the SNDCTL_DSP_SETPLAYVOL and SNDCTL_DSP_SETRECVOL commands, there are many applications which still make use of legacy calls such as SOUND_MIXER_WRITE_VOLUME. While some of these calls are available in Boomer, their utility is limited and none of them does anything beyond affecting the virtual volume associated with the application. For applications that need to control master levels, or access hardware configuration settings, the mixer API presented below is provided.

Plans for the API

In any case the OSS DSP API is the API that is preferred for most applications going forward. The definitions for this API are provided by including the `<sys/soundcard.h>` header file.

Note that much of the API is documented at http://manuals.opensound.com/ but there are caveats and exclusions. The full list of "approved" commands is documented in the **dsp**(7I) manual page.

Enumeration API

Applications often need to enumerate the audio devices that are present on their system. Boomer supports a /`dev/sndstat` device node which can simply be read (using *cat* or some other tool) to collect information about the devices on the system. Here's an example:

```
gdamore@tabasco{1}$ cat /dev/sndstat
SunOS Audio Framework

Audio Devices:
0: audio810#0 NVIDIA AC'97, CK804 (DUPLEX)
1: audiocmi#0 C-Media PCI Audio, CM8738-033 (DUPLEX)

Mixers:
0: audio810#0 NVIDIA AC'97, CK804
        AC'97 codec: Avance Logic ALC655 (6 channels)
1: audiocmi#0 C-Media PCI Audio, CM8738-033
```

As you can see, in this example there are two audio devices present, one using an **audio810**(7D) part from NVIDIA, and the other is a C-Media add-in card. (This example is taken from a Sun Ultra 20 with a C-Media 8738 add-in device.)

The `sndstat` node also supports an enumeration API via `ioctl` commands, enough for applications to perform programmatic device discovery and enumeration. (This API is supported on `/dev/mixer` as well, which in Boomer is just an alias for the `sndstat` node.) The enumeration typically involves issuing `SNDCTL_SYSINFO`, followed by multiple `SNDCTL_AUDIOINFO` or `SNDCTL_MIXERINFO` calls.

Mixer API

There are times when an application needs to do more than just playing or capturing audio. Typically these are applications which are used to perform device configuration details, or volume management. An example of such an application is *gnome-volume-control*.

For these applications, Boomer provides a version of the OSS v4 mixer API, which can be accessed via the `SNDCTL_MIX_`*`XX`* commands in the `<sys/soundcard.h>` header file.

Cross Device Security

There is a couple of problems with the Mixer API. The first of these is that the original OSS code allowed an application to adjust the settings of any arbitrary audio device by issuing commands against the `/dev/mixerXX` node for any other device (including the generic `/dev/mixer` node.) This creates a security problem where device ownerships are not necessarily honored, and can seriously hamper separation of devices across virtualization boundaries.

In Boomer, the solution is to require applications use two steps when accessing audio devices. The first step is to find the appropriate device (if not already known) by use of the enumeration API (`SNDCTL_AUDIOINFO` and similar) using the generic `/dev/mixer` node (which is not associated with any specific device. This can be used to determine the `/dev` paths associated with a real device.

The second step is to open the actual node for the physical device in question (for example `/dev/sound/audiohd0:mixer`) and perform operations for that device only against its own nodes.

Boomer will prohibit incorrect attempts to access device configuration from a node that is not associated with that device specifically. This closes this security hole.

Free Form ASCII Names and Values

The second problem with the mixer API comes as a result of its chief strength. Control names and values in the OSS mixer API are free form strings.

While this is a wonderful feature for extensibility, since almost any imaginable device setting or adjustment can be supported, it becomes incredibly onerous to application developers who have to contend with unpredictable names. This confounds layout engines, and also presents real challenges in developing localized or accessible applications.

To help applications out, Boomer uses a combination of numeric flags (separate from the ASCII name space) – such as `MIXF_PCMVOL` to indicate a PCM volume control, or `MIXEXT_SCOPE_INPUT` to indicate a control that is intended to be displayed with values associated with data capture – and predefined names backed by macros. These predefined names are not part of stock Open Sound System, but are a value add in Boomer. (Applications can access them by including the `<sys/audio/audio_common.h>` file.) Note that for now the names have no Commitment level, and should not necessarily be relied upon not to change in future versions.

Fortunately, very few applications need be concerned about these details. The primary such application is *gnome-volume-control* itself, which is handled via the GStreamer plug-in.

Chapter 3
Boomer Architecture

Major Subsystems

In Illustration 2, the boxes on the bottom represent different device drivers, the big box in the middle represents the main Boomer core (the **audio**(7D) module), and the top of the diagram represents different components in user space. Also worth mentioning are the **austr**(7D) STREAMs module, and the AC'97 support module; both of these modules are part of the total Boomer system.

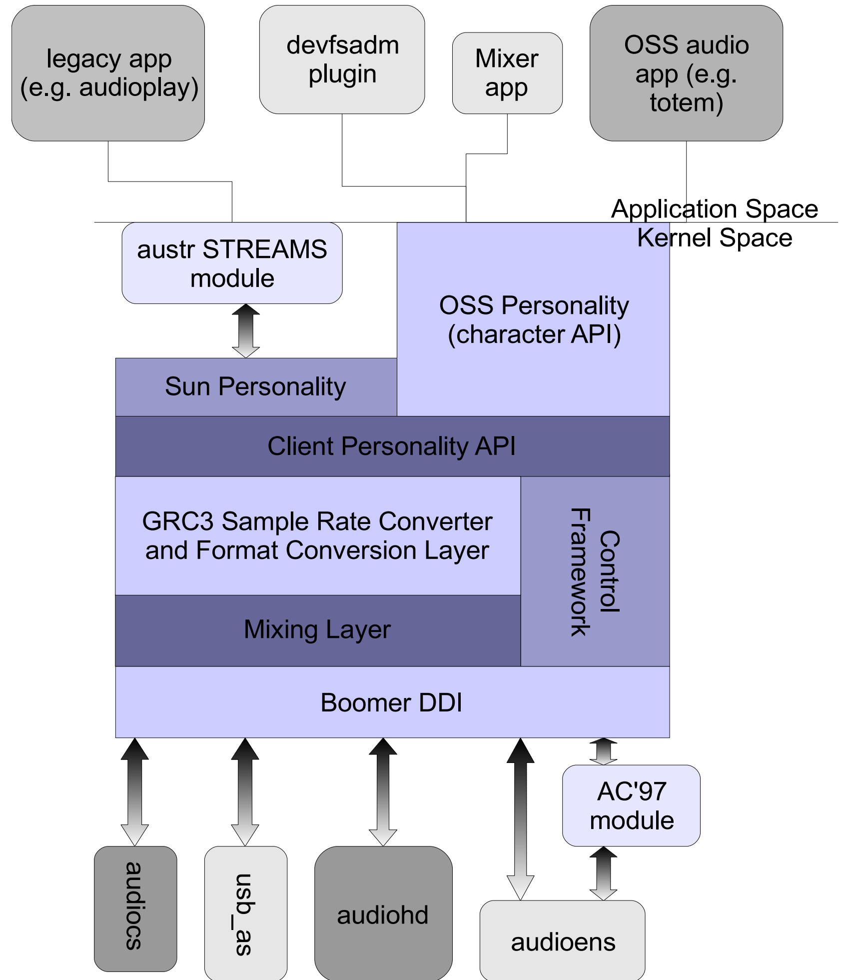

Illustration 2: Boomer components

Audio Core

At the heart of the Boomer subsystem is the audio core. This subsystem, which is implemented in the **audio**(7D) driver, provides all of the internal glue for the framework. It includes the logic to perform software mixing, audio format conversion (including sample rate and channel conversions), the client personality API implementation, and the device driver interfaces for audio drivers. The **audio**(7D) module contains within it several other major subsystems. For many Boomer drivers (**audiopci**(7D), **audiohd**(7D), **audiocs**(7D)), the **audio**(7D) module is the only external dependency (other than the kernel itself!) that is required. (This is typically represented by using the flags "`-dy -Ndrv/audio`" in the linker flags for the audio driver.)

Samples, Bytes, and Frames

The Boomer framework has a number of components that have to keep track of counters. Internally, and in the interfaces exposed to device drivers, the counters are all in terms of "frames" as opposed to "samples" or "bytes" or "words". Its important to understand what is meant by this terminology. A frame represents the entire set of audio data sampled

simultaneously. For example, in a single 8 bit, 8 kHz monophonic stream, there are 8000 frames in each second, each containing 1 sample, of one byte.

Now consider a 16-bit linear 5.1 surround sound at 48 kHz. In this case, each second will have 48,000 frames. Each frame will contain 6 samples, and each sample will contain 2 bytes.

Client Personality Interface

A significant design goal of Boomer is to support multiple different interfaces equivalently. For example, applications using OSS should not behave substantially differently from applications using legacy Sun `audio`(7I). (Note, however, that the legacy interfaces for Solaris are not able to express all of the features possible.) Boomer is also designed to be extensible to support other APIs in the future. For example, some day it may be desirable to support an API developed for MacOS, or the Linux Advanced Linux Sound Architecture (ALSA) API.

In order to facilitate this, Boomer is developed with the notion of multiple "Client Personalities" surrounding a common core. The client personalities communicate with the core using a relatively narrow well-structured internal (Project Private) interface. (This is enforced through the use of a specific header file, "`audio_client.h`", which only provides declarations for objects which the audio personalities need to access. Audio personalities must never `#include` other private audio header files.)

The framework chooses which audio personality a client application is using based on the minor node number that the application opens. (Note that the minor nodes themselves are "cloned", to allow for multiple simultaneous opens.)

A key component of this interface is the use of a "stream-specific" circular buffer. The buffer is used to exchange data between the core and personalities. Personalities are free to provide data in any "convertible" format (specified by use of specific function calls), and the core will consume data from the shared buffer (or for recording, produce it into the shared buffer), performing any necessary conversions as it goes. The buffers consist of head and tail counters, which are maintained as 64-bit non-wrapping integers. The counters represent frames. (Indexes into the buffers may be calculated from these pointers by performing the logical equivalent of a modulo operation against the configured buffer size, although for performance reasons an actual mathematical modulo operation is not performed. Also, there are situations where a reset of the index may occur, without changing the counter, which is another reason modulo operations cannot actually be used.) These indexes are monotonically increasing, and are completely independent of any underlying physical counters or counters used in other streams.

The Boomer core will call into the client personality when a client application calls `open`(2), and will provide to the client application an `audio_client_t` structure corresponding to the actual client application. Each `audio_client_t` also has with it a pair of `audio_stream_t` structures, corresponding the playback and record directions. The audio streams operate independently of each other, and each has its own buffer.

Of course, there is also a master audio device structure (audio_dev_t) for each real device. Client personalities can register client-private data on either the audio_dev_t, or the audio_client_t. There are also ways for one personality to access the private data of another personality. This is useful when two personalities collaborate – such as the personalities used for the OSS mixer and dsp interfaces.

The Boomer core handles minor node management, and the various DDI interfaces for the client personalities, although there are callbacks provided to personalities to trigger on certain DDI events or provide customization. (This is very important for some entry points – the handler for ioctl for example.)

Character API

Note that normally all minor nodes exported by the audio core are character devices associated with the actual device node for the audio device in the device tree. However there is a couple of notable exceptions.

First, the generic /dev/sndstat and (and its /dev/mixer alias) are pseudo nodes not associated with any particular hardware device; these are minor nodes for the **audio**(7D) pseudo device.

Second, the Sun personality exports STREAMs nodes for the /dev/audio and /dev/audioctl interfaces using a single instance of the **austr**(7D) pseudo device. Note however that these pseudo nodes are linked via a layered driver open with underlying internal minor node children (present in /devices but not /dev) of the actual hardware's device node.

In order to simplify implementation of client personalities, Boomer common code implements much of the transport logic for client personalities, including a framework for dealing with ioctl(9E), read(9E), and write(9E) routines.

Note that the common open(9E) handler for Boomer does not enforce exclusive access (F_EXCL) opens. Since each open gets a “virtual” device of its own, there is no need to do such enforcement.

It is worth restating that the Boomer framework creates minor nodes on behalf of the driver, but the minor nodes are attached to the driver's dev_info node in the device tree. This architecture makes it easier for the Solaris DDI to recognize when a device is in use and do proper reference counting.

STREAMs Module (austr)

In order to properly support the STREAMs semantics that are part of the **audio**(7I) API, and support a character device for the core as well, it was necessary to introduce a new STREAMs module, austr(7D). This is a STREAMs pseudo driver, and it exists primarily to provide a node

to hang clone-able STREAMs minor nodes from. As typical with pseudo drivers, there is only one instance in the device tree.

The actual functionality for the austr module is located in the Sun personality in the audio core itself. Thus, the austr module is extremely lightweight, containing only an `_init`(9E) and `_fini`(9E) that call into the audio core.

The book-keeping to ensure that minor nodes are properly tracked to physical drivers is done by having the STREAMs code perform an `ldi_open_by_dev`(9F) on a special internal node created by the framework for each device.

Internally, only the open and close operations make use of the layered driver interface (LDI). For all actual data transfer, simple function calls are used to ensure that a minimum performance penalty is incurred.

Sample Rate and Format Conversion

Boomer has adapted the GRC3 sample rate conversion licensed from 4Front OSS. Boomer has simplified and optimized the sample rate conversion significantly, in an effort to further improve the performance of the code. As a result, the GRC3 converter bundled with Boomer can only deal with 24-bit native-endian signed linear PCM data; but it does so with much greater efficiency than the original more general purpose code.

However, the format conversion layer converts from a client personality's preferred format to this internal native 24-bit format, and also deals with converting the data to the number of channels on the physical device, before passing it to GRC3. This code was heavily modified from OSS. The code can convert from µ-law[8], A-law[9], and the various 8, 16, 24, and 32 bit PCM[10] formats (signed, unsigned, native or reverse endian, etc.) Other formats (such as floating point formats or compressed audio) are not supported natively in the kernel interface, and must be converted in user-land by application code.

Mixing Layer

Once audio data has been converted to 24 bit linear PCM, and has been sample rate converted to a common rate (determined by the audio device itself) the mixing layer mixes the audio streams together.

As part of this process, per-stream and per-channel attenuation adjustments are made to the samples. This allows an application to adjust its own volume levels, without disturbing master

8 See the Wikipedia µ-law algorithm article, http://en.wikipedia.org/wiki/M-law_algorithm
9 See the Wikipedia A-law algorithm article, http://en.wikipedia.org/wiki/A-law_algorithm
10 Pulse-code modulation, see Wikipedia http://en.wikipedia.org/wiki/Pulse-code_modulation

settings for the audio device. Boomer also optionally applies a per-device master soft attenuation, allowing volume control to be supplied even for devices which lack any native hardware support for volume management.

Soft Volume Attenuation

Mixing streams normally results in adding their samples together (using signed arithmetic.) Hence, it is possible for multiple applications to drive the audio output beyond what is representable in the sample format. Therefore, at this point, an extra set of checks are also made to ensure that the process of mixing the audio streams does not cause this to happen. If it should be noticed, a global attenuation is made to the master volume. This attenuation is temporary, and fades away over time (once no longer necessary), using an algorithm licensed from 4Front.

For audio capture, the mixing layer does a simple “fan-out” of the audio data (just copying the samples to each application), and while per-stream attenuation is still applied, there is no additive volume adjustment that is necessary.

Multiple Channel Support

The mixing layer also contains the logic to deal with multiple channels, and channel conversions. The layout of per-channel data in audio buffers is not consistent from one device to another, and Boomer addresses this here. For example, Creative Audigy devices order the data in pairs of interleaved channels (because they use 4 separate stereo DMA engines to provide support for 8 channel audio.) On the other hand, devices from Intel just use a simple interleaving approach. Boomer can elegantly deal with whatever layout or ordering the device presents, in a simple manner that does not add any performance overhead – this is done internally in the mixing core. (This is a significant difference relative to 4Front OSS. The 4Front software uses a separate **remux** driver to merge multiple stereo devices together into one virtual device. This causes extra overhead. However, the 4Front implementation offers support for building multichannel support out of heterogeneous pairs of stereo devices, and also allows the application to directly access a stereo pair component of such a device individually. This last feature is something planned for Boomer, using a somewhat different approach).

Sun Personality

The Sun Personality implements the specific semantics that are documented in **audio**(7I), and represent the legacy interface to Boomer.

There were however some significant problems, with this. Some of the worst of these problems were bugs that were discovered in applications and in the legacy APIs themselves.

The biggest problem is the fact that the API was designed for a single exclusive-use device.

When the second generation mixer was added, some new ioctl commands were added that applications could use to help with sharing the device, but most applications don't make use of them. Two legacy ioctl commands, AUDIO_GETINFO and AUDIO_SETINFO, might need to access either settings associated with hardware, or settings associated with the "application" (in some cases even within the same ioctl), and it is not always possible to tell which the application intended. The second generation API used heuristics to figure this out, based on the order that /dev/audio and /dev/audioctl files were opened, but this heuristic does not always operate properly.

As a result, it is not always easy to predict when an application changes a setting, such as the volume or balance, whether the setting affects the master hardware volume, or a virtual setting (such as a sample count) associated only with the application. Making matters worse, sometimes which setting is affected *changes* during the life of the application.

The Boomer team took a hard look at this, and decided that the best thing to do is to separate applications into two classes. In the first class are applications that just do normal recording and playback, such as *RealPlayer*, *audioplay*, etc. In the second class are applications that are used to manage the master settings, such as *gnome-volume-control*, *mixerctl*, *sdtaudiocontrol*, and *mixer_applet2*.

The stance taken in Boomer is that applications shall be assumed to be in the first class, and therefore are unable to change master settings (including volume or port configuration settings). Their settings are applied only to the local application. To facilitate this, each application gets a virtual device created for it (applications determined by process id), and the settings are tracked for that application as long as it has either a /dev/audio or /dev/audioctl file open. It does not matter in what order these are opened, or whether more than one is open at a time.

The virtual device that the application sees is rather limited – it does not support changing the physical port configuration or monitor gain, and only has monophonic volume control for playback volume and record gain. (These controls will affect the volume of all channels for a given stream.) Put another way, the virtual balance is locked to AUDIO_MID_BALANCE. (The end user can still control the individual physical gains associated with each individual channel using an OSS compliant mixer application.

Applications which fall into the second class are not supported by the legacy **audio**(7I) API in Boomer. Instead, these applications should use the new style OSS mixer API. A GStreamer plug-in is provided so that Gnome mixer panel applications (for example gnome-volume-control) work as expected. Indeed they get the ability to access *additional* functionality, such as control for each of the audio levels in a 7.1 or 5.1 configuration.

The one application not converted which falls into this second class is *sdtaudiocontrol*. However, this application has simply been removed, as approved in LSARC[11] case 2009/074.

[11] LSARC is Sun's Layered Software Architecture Committee. See http://www.opensolaris.org/os/community/arc for more information.

As a consequence of the above changes, the ability to use the *audioplay* and *audiorecord* programs to administer port selection or master settings has been removed. These applications are relegated into the first class of “normal” audio applications. New features will be supplied in the *mixerctl* application to provide alternative ways to administer port configuration and master settings, including volume and monitor gain, from the command line.

Because of the various limitations inherent in the **audio**(7I) API, Sun recommends that new applications use the OSS API.

Open Sound System (OSS) Personality

The Open Sound System (OSS) personality provides the API found on Linux and FreeBSD, and represents a substantial new interface to audio for applications on Solaris.

There are a number of issues with this API as well. There are actually several different sub-APIs used by OSS, and 4Front does not consistently support all of these in different drivers. Additionally, there are parts of the API that are intended for use by applications, and parts that are intended for “internal private use”. The distinction between these is not always clear. There are also parts of the API that are completely irrelevant for Boomer, such as support for wave table synthesizers and MIDI devices.

While Boomer adheres to the primary parts of the API as documented by 4Front at http://manuals.opensound.com/, there are some minor deviations that should not affect properly designed applications.

1. There is no support for MIDI, sequencers, timers, or wave table synthesizers.
2. There is no separate “**vmix**” device. The Boomer core doesn't require separate devices for software mixing, since this functionality is an integral part of the Boomer core. (Solaris clone opens give each open file its own unique minor number, instead.)
3. It is not possible to bypass the software mixing in Boomer, nor is it possible to “directly” access the raw device.
4. Support for certain diagnostic use and optional `ioctl` commands is not present.
5. The actual paths to device names may vary, although applications that correctly use the `/dev/sndstat` or `/dev/mixer` common pseudo devices to discover what devices are present will function properly. Conversely, applications that rely on `/dev/pcm` or `/dev/dsp` might not.
6. Applications that use the older version of the OSS mixer API (such as `SOUND_MIXER_WRITE_VOLUME`) will be assumed to be ordinary audio applications. When they attempt to use this to adjust gain levels, the changes will be made to a virtual setting for the process, and not applied to the physical hardware.

7. The new mixer API is reserved for use by mixer panel type applications. A GStreamer plug-in will be supplied to use this, which will enable the proper operation of the Gnome desktop.
8. Applications making use of the mixer API are required to access the actual mixer device node as reported by the enumeration API. Note that 4Front has agreed to document this requirement as a recommendation in their own API documentation.
9. Predefined names used for both controls and their values are available, so that they can be used by applications which need to support globalization or determine the purpose of a control for other reasons.
10. The ability to change interrupt frequency or latency policies using ioctl commands is severely limited. (Note that the interrupt rate – which governs latencies -- is typically configurable for the device drivers using the `driver.conf`(4) mechanism to configure the interrupt rate, but there should be little need to ever change these from the default (which is normally 175 Hz for most drivers).
11. Not all OSS `ioctl` commands are supported equally. In particular, there are a number of such which are required for legacy applications, but which should probably not be used by new code, since there are often cleaner ways or better ways of achieving the same thing. While Boomer will provide support for many of these, any documentation for them discourages their use[12]. The precise details are supplied in the **dsp**(7I) and **mixer**(7I) man pages.

Audio DDI

As part of Boomer's goal to facilitate porting of device drivers from 4Front OSS and other sources (including Linux and FreeBSD), as well as just generally making it easier to write audio device drivers in general, this project introduces a new kernel Device Driver Interface for audio device drivers. Sun's experience so far is that the average device driver written using the Boomer DDI will be roughly half as many lines of code as a traditional Sun audio driver, and significantly simpler to understand. Sun estimates that it takes an average kernel engineer about a week to port an average driver from earlier releases of Solaris or 4Front OSS to Boomer, after spending some initial time to learn the new DDI. (More senior engineers can accomplish this in significantly less time. Some audio hardware might be significantly more complicated, resulting in a more expensive effort. Two such examples are the HD Audio[13] and the USB audio[14] subsystems.) A simple driver for a typical AC'97 device can be implemented in around 1000 lines of code[15].

---

12 The official way of doing this is to mark the API “Obsolete” in official documentation.
13 See Intel High Definition Audio, http://www.intel.com/design/chipsets/hdaudio.htm
14 Universal Serial Bus Device Class Definition for Audio Devices, http://www.usb.org/developers/devclass_docs/audio10.pdf
15 See for example the code for audiovia823x at http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/Note17

More information about the DDI is provided below.

Audio Control Framework

The audio control framework provides a mechanism for audio drivers to export control objects and operations for use by mixer panel applications and configuration tools. Some other operating systems call their equivalent interfaces "mixer APIs" or similar.

These objects are usually used to control the behavior of hardware mixing devices on audio hardware. (For example, directing which output ports should be active, which input ports should be active, and volume or attenuation settings for individual channels or streams.)

Boomer defines a broad list of "well-known" controls, which are identified by constant string values. These controls are listed in `<sys/audio/audio_common.h>`. Well known controls will have a well understood purpose, such that their behavior is predictable from one driver to the next. These controls also will have specific "types" associated with them, which shall not vary. (For example, the `AUDIO_CTRL_ID_INPUTS` control shall always be an enumeration of ports (`AUDIO_CTRL_TYPE_ENUM`), while the `AUDIO_CTRL_ID_LINEOUT` will normally be a stereo gain setting of `AUDIO_CTRL_TYPE_STEREO`.)

In addition, Boomer allows for device drivers to supply their own "device specific" controls (tunable parameters) using free-form text names. For example, "`ac97_stereo_simulate`" is an AC'97 specific control that determines whether stereo output should be simulated when only a single monaural output is available. Generally it will not be possible for applications to know in advance what device specific controls might be present, and therefore it is not expected for these controls to be readily accessible in GUI applications.

Chapter 4

Common AC'97 Module

It turns out that many of the devices in common use are based around the popular Intel AC'97 specification. This specification defines the interface between audio codecs, and host controllers. However, it says nothing about the host controller interface itself, so there are a large number of variations requiring a corresponding large number of drivers. (This is not unlike the world of Ethernet devices, where the IEEE 802.3[16] standard (clauses 22 and 45) spells out the interface between host controllers and transceivers, but says nothing about how the host controller itself is accessed.)

common/io/audio/drv/audiovia823x/audiovia823x.c

16 The IEEE 802.3 specification is available from http://standards.ieee.org/getieee802/802.3.html

As a result, it is desirable to provide as much of the common AC'97 functionality in a common module, so that device drivers need supply only generic access routines to read and write codec registers, and can leave the bulk of the work in implementing the control API to common shared code.

Device drivers that desire to do this use the common `misc/ac97` module, and access the declarations through the `<sys/audio/ac97.h>` header file. They can still supplement AC'97 with their own controls, or override the default definitions of controls provided by AC'97.

This design also allows for a greater range of devices to be supported, since probably the most common axis for variation in a family of audio devices is the selection of the codecs used on the device. By putting the logic in a common core, Boomer gains the ability to automatically detect different codec behaviors, as well as provide for codec-specific overrides. Additionally, codec-specific behavior can be implemented just once, and all drivers making use of the AC'97 common module can immediately benefit.

Chapter 5
Specific Devices

Intel High Definition Audio

Perhaps the most common audio devices on new motherboards these days are those based upon the Intel High Definition Audio specification. Intel released this new specification for audio devices, to replace AC'97. Unlike AC'97, this is a complete specification for the codecs and the host controller, including a PCI class definition. Thus, there are controllers from multiple parties (including Intel) which also conform to this new specification.

For the foreseeable future, it is expected that parts compatible with this specification will be the audio device of choice integrated on most desktop and laptop hardware.

The specification is very rich, and allows for a mind-boggling set of possible features, including up to 16 channels of 192 kHz 32-bit audio. (This is far in excess of any current needs, or the ability of most human ears to discern.) It also provides for very flexible codec configuration, and up to 15 separate DMA channels for playback and 15 separate DMA channels for record.

Sun believes that this device represents the “high bar” for capabilities of audio subsystems for the foreseeable future. (Although there have been some demonstrations of a system with 24 separate output channels.[17])

[17] See http://en.wikipedia.org/wiki/22.2

The **audiohd**(7) driver has been updated to take advantage of Boomer's additional features. Work on expanding the set of supported device features and codecs for this driver is ongoing.

USB Audio

Boomer provides support for USB audio, but this subsystem is one of the areas that has been changed significantly from the legacy stack.

In the early stages of implementation, Boomer had a separate shim layer which allowed legacy SADA drivers to be used with Boomer. After all of the other drivers were converted, USB still remained. Rather than retain a separate module, this shim layer was merged into the USB audio drivers directly. At this point, there is no longer any support for SADA device drivers in OpenSolaris.

The other significant change is that since Boomer drivers are no longer STREAMs devices, it was necessary to modify the mechanism to set up the plumbing used by the USB audio framework. The configuration of the multiple device nodes is now accomplished using the layered driver interfaces (LDI).

Eventually, the Boomer team would like to see the various separate USB audio modules (**usb_as**, **usb_ac**, and **usb_ah**) combined into a single unified driver, and the code restructured to act more like a natural native Boomer device. The Boomer team also expect to see more devices supported, and additional device features supported by Boomer. Work on USB audio is ongoing.

SPARC Devices

The **audiots**(7D) and **audio1575**(7D) drivers continue to operate, but since the underlying devices are also found on some motherboards, the Boomer team has ported the drivers to the x86 platform. The **audio1575**(7D) driver has also been improved to support 5.1 surround sound if paired with a suitable codec.

The **audiocs**(7D) driver has been updated to Boomer, but otherwise remains unchanged.

AC'97 Devices

The various AC'97 based device drivers have been improved substantially. They all now support multichannel surround, extended Solaris functionality such as suspend/resume and fast reboot support, and in several cases support additional hardware configurations. Furthermore, the formerly closed source **audiovia823x**(7D) and **audioens**(7D) drivers have been re-implemented and are now open source. Additionally, with the changes in the **audio810**(7D) driver, the audio experience in *VirtualBox*[18] guests is significantly improved.

[18] VirtualBox is Sun's virtual machine for x86 processors. See http://www.virtualbox.org/ for more information or to download it for your own use.

New Devices

At integration time, the only new driver introduced with Boomer was the **audiopci**(7D) driver, for legacy Ensoniq 1370 (aka AudioPCI) hardware. Since that time, work has been progressing steadily on new drivers. Already integrated is the **audiocmi**(7D) for C-Media 8738 family devices, and several other drivers are in the qualification stages now. (Forthcoming drivers include support for VIA based devices, and Creative Audigy family devices.)

Sun Ray Ultra-Thin Clients

Support for audio on Sun Ray[19] client hardware is, at this time, unchanged. The Sun Ray software uses a separate emulation of the audio layer which only provides a very limited exclusive access device that supports only the basic **audio**(7I) API.

However, work is in progress on creation of a regular Boomer device driver for Sun Ray devices will offer full mixing capabilities and integrate tightly with the rest of the Boomer framework. This will also bring the OSS API to Sun Ray, allowing the legacy **audio**(7I) to be officially deprecated.

Chapter 6
Boomer Device Driver Interface (DDI)

As stated previously, Boomer introduces a new simple DDI for device drivers to use. While an updated chapter of *Writing Device Drivers*[20] has not yet been written (but hopefully will be soon), it might help to look at a few of the interesting parts of the DDI.

Most audio devices will only need to include `<sys/audio/audio_driver.h>` to pull in all of the definitions that are specific to audio. (The main notable exception is that AC'97 device drivers may also want to include `<sys/audio/ac97.h>`.)

Fragments

One of the units that is discussed in the Boomer DDI is the notion of a “fragment”. A fragment is a chunk of audio data normally processed as part of an interrupt. So, for example, a 48 kHz audio device that interrupts at 175 Hz will have a fragment size of 48000 / 175 = 274 frames. Boomer insists that drivers supply a buffer that contains a whole number of equally sized fragments. This is used only in the buffer sizing. During actual processing, more or less data than a whole fragment can be processed, although drivers should not deviate too much from the fragment size

[19] Sun Ray refers to Sun's thin client family of products. See http://www.sun.com/sunray for more information.
[20] Also known as the WDD, this book is the primary reference for writing device drivers for Solaris and OpenSolaris. It's available via http://docs.sun.com/app/docs/doc/819-3196

because Boomer uses the fragment size as a hint about how much data should be "preloaded" to the device to avoid buffer under-runs.

Structures

Note that most of these structures are opaque to device drivers. This allows the Boomer core to evolve and change without breaking compatibility for device drivers.

`audio_dev_t`

This structure represents a logical audio device. Some physical cards may have more than one of these, but generally there will only be one of these per physical audio device. An audio device may have multiple *engines*, but will always have a single set of controls.

`audio_engine_t`

This structure represents a logical DMA engine. For most simple stereo devices and many multichannel devices, this will correspond to one hardware engine. An audio engine can only operate in one direction (capture or playback) at any given time, and has exactly one configured sample rate, data format, and channel configuration.

A single audio device may have multiple of these engines. Most audio devices actually have two of them, one for playback, and one for capture. (This allows playback and capture to be done simultaneously, in what is referred to as "duplex" mode.)

Some multichannel devices, such as those from Creative Labs, may use multiple physical DMA engines, but represent them to the Boomer framework as a single `audio_engine_t`.

Most of the entry points that a driver provides to Boomer are provided on a per `audio_engine_t` basis, so devices that have engines with different abilities can be represented properly.

`audio_engine_ops_t`

The `audio_engine_ops_t` is the structure supplied by the driver to Boomer as part of registering an audio engine. This structure is not opaque, as its members are provided by the driver. Here it is:

```
struct audio_engine_ops {
      int   audio_engine_version;
#define     AUDIO_ENGINE_VERSION    0

      int   (*audio_engine_open)(void *, int flags,
```

```
                unsigned *fragfr, unsigned *nfrags, caddr_t *buf);
        void    (*audio_engine_close)(void *);

        int     (*audio_engine_start)(void *);
        void    (*audio_engine_stop)(void *);

        uint64_t      (*audio_engine_count)(void *);

        int     (*audio_engine_format)(void *);
        int     (*audio_engine_channels)(void *);
        int     (*audio_engine_rate)(void *);

        void    (*audio_engine_sync)(void *, unsigned);

        size_t        (*audio_engine_qlen)(void *);

        void    (*audio_engine_chinfo)(void *, int chan,
                unsigned *offset, unsigned *incr);
};
```

The descriptions of the entry points is provided below, but note the audio_engine_version field. This allows for version checking, and for new members to be added to the structure without breaking binary compatibility provided that the version number is bumped (and that suitable checks are placed in the framework, of course.)

audio_ctrl_t

The `audio_ctrl_t` structure is used to represent a single modifiable value, or control, for the audio device. Examples controls might be the master output gain, the port to use for audio capture, or the configuration to change the purpose of a jack. Each control has a name, represented in ASCIIZ, a 64-bit value, and a private handle for the driver's own use. The control has entry points into the driver to support reading and writing the value.

audio_ctrl_desc_t

This structure describes a control. Like `audio_engine_ops_t`, it is not opaque to the driver. Here is the structure:

```
struct audio_ctrl_desc {
        const char          *acd_name;        /* Controls Mnemonic */
        uint32_t            acd_type;         /* Entry type */
        uint64_t            acd_flags;        /* Characteristics */
        uint64_t            acd_maxvalue;     /* max value control */
        uint64_t            acd_minvalue;     /* min value control */
        const char          *acd_enum[64];
```

```
};
```

The structure contains a name, in ASCIIZ, a “type” for the control (such as AUDIO_CTRL_TYPE_STEREO), optional flags (such as AUDIO_CTRL_FLAG_READABLE), minimum and maximum values, and an array of strings representing names for possible enumeration values. Potential names can be taken from the AUDIO_CTRL_ID_*XXX* values listed in <sys/audio/audio_common.h>. For the full list of flags and types, see the **audio_engine_ops**(9S) manual page.

If the control is of type AUDIO_CTRL_TYPE_ENUM, then the 64-bit value associated with the control is actually a bit mask. (Only one bit of which can be set unless the AUDIO_CTRL_FLAG_MULTI flag is set.) The ASCIIZ names of the enumerations are stored in the acd_enum field. The acd_maxvalue and acd_minvalue are special fields in this case, which represent the set of bits which can be set (acd_maxvalue) and the set of bits that are modifiable (acd_minvalue).

Functions

The following functions are provided to device drivers.

Entry point initialization

```
void audio_init_ops(struct dev_ops *, const char *name);
void audio_fini_ops(struct dev_ops *);
```

These functions are intended to be used in a driver's _init(9E) or _fini(9E) entry points, and configure the various fields of the dev_ops structure. The name argument is the name of the device driver itself, for example “audiohd”.

Audio Device Allocation

```
audio_dev_t *audio_dev_alloc(dev_info_t *, int);
void audio_dev_free(audio_dev_t *);
```

These entry points are used to allocate and free an audio device. Normally these are used during attach(9E) and detach(9E) processing. They may perform sleeping allocations, and so shouldn't be used in other contexts.

Audio Device Information

```
void audio_dev_set_description(audio_dev_t *, const char *);
void audio_dev_set_version(audio_dev_t *, const char *);
void audio_dev_add_info(audio_dev_t *, const char *);
```

These functions are used to "tag" the audio_dev_t structure with important information. Note that audio_dev_add_info() is cumulative; each call adds another line of text which will be reported for the device in the /dev/sndstat node.

Audio Engine Management

```
audio_engine_t *audio_engine_alloc(audio_engine_ops_t *,
      unsigned flags);
void audio_engine_set_private(audio_engine_t *, void *);
void *audio_engine_get_private(audio_engine_t *);
void audio_engine_free(audio_engine_t *);
```

These routines are used to allocate and free an audio engine. The private state pointer set with audio_engine_set_private() will be passed as the first argument to any of the engine's entry points listed in the audio_engine_ops_t.

The flags argument to audio_engine_alloc() indicates the capabilities of the underlying hardware engine. At the moment, only ENGINE_OUTPUT_CAP and ENGINE_INPUT_CAP are defined, but they may be logically OR'ed together for an engine that can do either playback or capture.

Audio Engine and Device Registration

```
void audio_dev_add_engine(audio_dev_t *, audio_engine_t *);
void audio_dev_remove_engine(audio_dev_t *, audio_engine_t *);
int audio_dev_register(audio_dev_t *);
int audio_dev_unregister(audio_dev_t *);
```

These functions are used to register engines with an audio device, and ultimately to register (or unregister) an audio device with the Boomer framework.

The act of registering the audio device with the framework triggers most of the other actions in Boomer, including creating minor nodes for the device.

Data Management

```
void audio_engine_consume(audio_engine_t *);
void audio_engine_produce(audio_engine_t *);
void audio_engine_reset(audio_engine_t *);
```

The audio_engine_consume() and audio_engine_produce() calls are used to inform the framework that some audio data has either finished playing or been captured (respectively). Normally these are called once per fragment as part of the interrupt processing.

The audio_engine_reset() is used to alert the framework that the device has been reset, and the indexes into the audio buffer that were previously assumed to be valid might not be so any

longer. This can happen in response to fault handling or as part of a device reset during DDI resume handling.

Note that it is important that locks are not held across any of these calls. Also, the data notification calls can be performed at any time, but need to done reasonably regularly (such as from a timer or fragment interrupt).

Audio Control Support

```
audio_ctrl_t *audio_dev_add_control(audio_dev_t *,
      audio_ctrl_desc_t *, audio_ctrl_rd_t, audio_ctrl_wr_t,
      void *);
int audio_dev_add_soft_volume(audio_dev_t *);
void audio_dev_del_control(audio_ctrl_t *);
void audio_dev_update_controls(audio_dev_t *);
int audio_control_read(audio_ctrl_t *, uint64_t *);
int audio_control_write(audio_ctrl_t *, uint64_t);
```

These functions are used to manage the controls for an audio device. Note that each control has read and write accessor functions, and driver-selected state pointer associated with it.

The `audio_dev_add_soft_volume`() function is used to fabricate a control which will provide master volume support by performing software based attenuation of the audio data. It is useful for devices which lack hardware volume support, but should not normally be used otherwise, since it can only provide attenuation and not gain.

The `audio_dev_update_controls`() is provided to support devices with hardware controls which may change asynchronously in a manner not under Boomer's control For example, a device with external volume knobs could call this when it detected that a change had occurred on one of the knobs.

Miscellaneous Functions

```
void audio_dev_warn(audio_dev_t *, const char *, ...);
void audio_dump_bytes(const uint8_t *w, int dcount);
void audio_dump_words(const uint16_t *w, int dcount);
void audio_dump_dwords(const uint32_t *w, int dcount);
```

These are provided as a convenience to the driver. Also, `audio_dev_warn`() can be used in lieu of `cmn_err`(9F), and prefixes the device information to the logged message.

Entry Points

Note that all of the following entry points are members of the `audio_engine_ops_t` structure and take as their first argument the private pointer that was previously established using

audio_engine_set_private(). This is normally a pointer to the driver's private soft state structure.

Opening and Closing the Engine

```
int   (*audio_engine_open)(void *, int flags,
          unsigned *fragfr, unsigned *nfrags, caddr_t *buf);
int   (*audio_engine_close)(void *);
```

The audio_engine_open() entry point opens and initializes the DMA engine and configures any associated hardware (such as sample rate or format conversion logic) for the engine. Upon completion, the routine returns back the size of a fragment in frames, the number of fragments in the audio buffer, and a pointer to the contiguous memory region for the audio buffer itself.

The flags argument indicates the direction of transfer (ENGINE_INPUT or ENGINE_OUTPUT) and may optionally include ENGINE_NDELAY to indicate that the open should not wait if the engine is already in use.

The audio_engine_close() routine correspondingly closes down the audio device. Once audio_engine_close() is performed, the engine must not be interrupting nor performing DMA transfers.

Boomer will access audio data by reading from or writing to the circular memory buffer supplied.

These entry points are only called in user or kernel context. On success they return zero, otherwise they return an error number[21].

Starting and Stopping

```
int   (*audio_engine_start)(void *);
void  (*audio_engine_stop)(void *);
```

These entry points simply start or stop the transfer of audio data. They are optional, but if not supplied, then the value NULL should be used in their place. In that situation audio_engine_open() must actually start the data transfer.

Engine Positioning

```
uint64_t    (*audio_engine_count)(void *);
size_t      (*audio_engine_qlen)(void *);
```

The audio_engine_count() entry point reports the engine's position as a 64-bit increasing and

[21] See the **errno**(5) entry in the system manual for a list of valid error numbers.

non-wrapping value. The starts at 0 when the engine is first opened. The count is supplied in frames. Note that this count does *not* reset when an audio_engine_reset() is performed.

The audio_engine_qlen() entry point reports the depth of a hardware FIFO beyond that visible in the circular buffer. This can allow more accurate positioning.

Data Format Handling

```
int   (*audio_engine_format)(void *);
int   (*audio_engine_channels)(void *);
int   (*audio_engine_rate)(void *);
void  (*audio_engine_chinfo)(void *, int channel,
      unsigned *offset, unsigned *increment);
```

These entry points are used to report the data format of the engine back to the framework after audio_engine_open() is called. The format will be a value such as AUDIO_FORMAT_S16_LE (see <sys/audio/audio_common.h> for other possible values.) The channels will be the total number of channels, such as 6 for a 5.1 surround sound configuration. The rate is the sample rate in frames per second (very often this is simply 48000.)

The audio_engine_chinfo() entry point is a bit more complex, since it is used to report the interleaving of multiple channels. The framework calls this for each channel as reported by audio_engine_channels() -- note that channel numbers start from zero. For a simple interleaving, the offset will be the same as the supplied channel number, and the increment will be total number of channels. However, more complex scenarios are possible, such as grouped pairs of interleaved channels. This entry point is optional; if not supplied than a simple interleaving scheme with each channel presented in order will be assumed.

DMA Synchronization

```
void  (*audio_engine_sync)(void *, unsigned nframes);
```

The audio_engine_sync() entry point synchronizes DMA caches. Normally this just evaluates to a call to ddi_dma_sync(9F). The nframes is the total number of frames to synchronize. The device driver is responsible for knowing the direction and for keeping track of the offset. Most drivers just synchronize the entire buffer for simplicity.

Chapter 7
Future Directions

Boomer is still very much an evolving project, and the Boomer team is working to further improve upon it. Here are some areas under development.

Sun Ray Audio

A significant drive is underway to develop support for Sun Ray thin clients using a Boomer native driver. This will bring new mixing and API functionality to Sun Ray users, and ultimately allow for the legacy **audio**(7I) API to be Obsoleted. Note that at this time Sun Ray is not officially supported on OpenSolaris, but a forthcoming release is expected.

Virtualized Audio

Virtualized audio support is a multifaceted project, where the Boomer team hopes to close a number of gaps in the current architecture.

Sun Ray

The first problem is Sun Ray audio and the OSS API. In the Sun Ray environment, Sun Ray users are likely not to have direct access to the audio hardware on the server (usually the server won't even have any audio hardware!) They do need to be able to access their own hardware.

Historically the Sun Ray software created a pseudo-node in the `/tmp` file system for each DTU, and stored the path name for that node in an environment variable that Sun applications looked at (`AUDIODEV`). However, many open source applications (especially those developed for the OSS API) won't honor such an environment variable, and just assume that they can open `/dev/dsp`.

Hence, `/dev/dsp` (and arguably also `/dev/audio` and the other nodes) must be able to detect that it is being run from within a Sun Ray session, and direct the audio streams appropriately.

This should only be done if the application is being used within a Sun Ray session.

Hot Plug Support

A different situation arises when using hot pluggable devices, such as USB headsets.

Historically, when inserting USB audio hardware, the `/dev/audio` link would change, so that any applications which reopened the audio device would start using the USB device by default. Today, that doesn't work with Boomer.

In many cases, one would like the audio streams of an existing application to be redirected to a USB headset when one is inserted, without having to restart audio applications. (In this case the headset is acting much like inserting an RCA stereo plug on traditional audio hardware.)

However, this isn't always the case. For example, one might install a USB camera with a microphone on it. This device should not be the output device by default, and it may be undesirable to use it for the default input device as well.

In another situation, one can imagine using a USB headset for teleconferencing, but playing MP3 music in the background. In this situation, the configurations should probably not change at all.

Clearly dynamic hot-plug actions are interesting, but need to be configurable by the end user.

The Boomer team hopes to tackle this problem during the second half of 2009 as well. At least a portion of it is required for Sun Ray support.

Secured Virtual Environments

In secured environments, such as an environment with labeled security zones, it is vital that audio device access be strictly controlled. Obviously it would be a fatal breach of security for an untrusted domain be able to capture audio data to listen in to a confidential conversation.

However, it is often the case that the user sitting at the keyboard wants to be able to hear audio output from such environments, such as when watching a video from an unclassified source.

Additionally access to hardware controls has to be strictly controlled to prevent their use for covert communication channels between domains with different security labels.

Right now the solution used for this is rather clumsy, as it requires the end user to grant access to the device to only a single domain at a time, and this often requires applications to be restarted.

A more elegant solution would involve some combination of mixing and the use of a “virtualized” device which could dynamically change based on user preference. (So the ability to record audio data could be moved from one label to another, without restarting the applications).

Conversations about how to achieve this are still under way with the security group.

Further Driver Improvements

The Boomer team intends to continue to drive improvements to the abilities of the existing audio drivers (especially the **audiohd**(7D) and USB audio devices), but is also continuing -- with support from a partnership with 4Front Technologies -- to work on delivery of new drivers for additional hardware. New drivers for a range of hardware from vendors such as Creative, VIA, and Yamaha are already being developed, and more are expected.

Additionally, it is now possible, since all Boomer drivers are open source, for third parties to get involved and contribute either improvements or new hardware support.

BrandZ Improvements

Using Solaris' BrandZ[22] technology, it is possible to run Linux binaries in a branded zone on

[22] BrandZ is a framework that allows applications from other operating systems – such as Linux -- to run natively on Solaris, using Solaris' zones facility to emulate a different host operating system. Learn more at http://www.opensolaris.org/os/community/brandz

Solaris. Linux applications can do many things, including accessing the audio device using the Open Sound System API.

Prior to Boomer, the application support for this was provided by a special driver, **lx_audio**(7D), which emulates the OSS API for Linux applications on top of the Solaris API. This is rather inefficiently done, and ultimately the Linux applications have a limited view of the device. (For example, multichannel audio isn't supported in this configuration.)

It should be a fairly straight-forward matter to rip out most of the **lx_audio**(7D) support and just let the Linux application access the OSS API in Boomer directly. The only conversion support necessary may be to remap the `ioctl` command numbers, since Linux numbers the commands somewhat differently from Solaris.

Dolby Digital[23] Support

Many consumers have expressed an interest in playing compressed format audio (AC3) data. The normal source for such audio data is from DVD media, but it can come from other sources such as video capture cards.

Already in the **audiohd**(7D) driver there is limited support for digital audio transport, but it is limited to uncompressed streams.

There are several ways to support compressed audio streams.

The first is to pass the compressed audio over a digital directly to a digital receiver. To do this, only a transport is required. Its critical that no other data formats can be used at the same time, so this precludes use of a mixer or any volume controls on the host operating system. This is commonly called AC3 pass-thru. While 4Front OSS supports this, Boomer does not. Support for this technology is currently under investigation.

A second option is to include a software decoder. This would allow applications with compressed content to work using existing 5.1 analog configurations and with software remixing, other speaker configurations could also be supported. Fortunately, the most interesting applications already seem to have this capability built in.

A third option, and in the author's opinion perhaps the most interesting, would be to support on the fly compression of regular audio data, and delivery of such compressed streams over a digital link. This would allow a single cable connection between the Solaris host operating system and a home theater system, but still retain the ability to provide full fidelity DVD quality surround sound.

With other compression schemes, such as Dolby Digital TrueHD or DTS Surround there might be other options as well.

---

23 Dolby Digital is the trade name for a compressed audio scheme also known as AC-3, or ATSC standard A/52B. The standard can be downloaded from http://www.atsc.org/standards/a_52b.pdf

This third option is challenging for a number of technical reasons, but if successfully implemented, offers the most flexibility for end-users. Investigation is ongoing.

Chapter 8
Community Involvement

Now that Boomer has integrated, its possible for community members to contribute in meaningful ways – fixes, new drivers, and suggestions are all welcome. The Boomer project has an open source presence on http://www.opensolaris.org/project/opensound/ and there is also a discussion list (opensound-discuss@opensolaris.org) which can be subscribed to from the same site.

There is also a PSARC[24] case 2008/318 covering the initial Boomer integration. A great deal of additional information about Boomer, including draft manual pages for all of the APIs and the DDI, can be found within the case materials[25].

Questions, comments, and feedback can be sent to the discussion list; subscription is not necessary.

It is also worth noting that 4Front Technologies is an active participant in Boomer community.

24 PSARC is the Platform Software Architecture Committee, which governs most of the architectural matters surrounding the core Solaris operating system.

25 http://arc.opensolaris.org/caselog/PSARC/2008/318/

Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA Phone 1-650-960-1300 or 1-800-555-9SUN (9786) Web sun.com

© 2009 Sun Microsystems, Inc. All rights reserved. Sun, Sun Microsystems, the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Information subject to change without notice.