Open Sound System (OSS) version 4.0 from programmer's perspective

White Paper

Author: Hannu Savolainen, 4Front Technologies, hannu@opensound.com

Last modified: 13 Nov 2006

Introduction

Version 4.0 Open Sound System is the result of about 5 years of work. The primary goal has been to guarantee full backward compatibility with applications written for any previous OSS versions and its predecessors such as VoxWare and the original Linux sound (blaster) driver. Thanks to the very generic nature of the OSS API this has been possible. There are just very few changes that may affect some older applications that violate the design guidelines given for the older OSS versions.

However the amount of new functionality added to OSS 4.0 is enormous. For this reason it's recommended that all programmers using the OSS API take a look at the new features described in this white paper.

In this paper OSS 4.0 means the “official” OSS 4.0 implementation released by 4Front Technologies. However due to open nature of OSS there are several independent implementations of the OSS API (for example stock FreeBSD sound driver). They may not have implemented all OSS 4.0 features but this can be handled by using the recommended default policies described in OSS 4.0 Programmer's Guide.

Compatibility

One of the primary design goals of the OSS API has been full backward and forward compatibility. Backward compatibility means that applications using features of any of the previous OSS versions will work unmodified with not just the latest OSS version but also wit any future versions. However this is only possible if the application design has followed the official OSS documentation. Using previous OSS versions in undocumented ways may cause some problems.

Forward compatibility means that applications written for the latest and finest OSS version (such as OSS 4.0) can work also with older versions. However the requirement is that the applications use the recommended default policies if the new functionality is not present. Unimplemented ioctl calls will return errno=EINVAL when the application tries to use an unimplemented function.

Unix/POSIX/Linux device file compatibility

OSS has been a fully compatible implementation of the old and reliable Unix device file model. This makes it very easy to write applications that use OSS because most Unix/Linux programmers are already familiar with this model. There have been some discussion about implementing parts of OSS as a Library. However this was not seen necessary because all lowest level audio API features can be implemented without need of more or less redundant library layer.

Another benefit of the device based model is the weak binding capability provided by it. If there is no audio subsystem installed this doesn't necessarily cause any show stopper problems to the applications. They simply fail when trying to open the sound device file and can disable sound related features. The application itself can continue it's operation which itself is not necessarily related with sound at all. There are no problems caused by missing or incompatible libraries.

Higher level functionalities that would benefit from library level code are not part of the OSS API.

Instead many kind of higher level libraries, such as Jack, can be easily implemented on top of OSS.

Multi-platform availability

OSS 4.0 is currently available for Linux, Solaris, FreeBSD, SCO OpenServer and SCO UnixWare. More ports will be made in the future. Older OSS versions are still available for many less common operating systems.

Porting an OSS application between operating systems will require just recompile. The OSS API has easy to use features to help application designers to handle architecture issues such as endianess without additional work or knowledge.

32/64 bit neutral design

The OSS API has been designed to be fully binary compatible between 32 and 64 bit implementations of the same CPU architecture. This means that OSS installed in a 64 bit operating systems will support also 32 bit applications without additional emulation mechanisms.

Most important changes in OSS 4.0

There are few major implementation changes made to OSS 4.0. They are fully transparent so changes are not necessarily required to existing applications. However future applications will become better if they take care of these changes.

Black box architecture

One of the primary design goals of OSS has been full device abstraction. The idea of this programming model is that applications don't actually access the devices itself. Instead they communicate with a higher level entity which is similar to the networking (TCP/IP) subsystem of most operating systems.

Instead of tweaking bells and whistles of given audio device the application simply tells what it would like to be done. OSS then selects the most optimal strategy to get the actual device to perform the requested operation. For example an application may request OSS to play a 44.1 kHz/16 bit/stereo audio stream. OSS then takes the responsibility to convert this stream to the nearest format supported by the actual device. It may for example perform automatic 44.1 kHz to 48 kHz sample rate conversion if the device doesn't support 44.1 kHz.

However the application can still find out the very deepest features of the device and use them directly.

Core and extended functionality

Another major development since earlier OSS version is partitioning the API features to core functions needed by all applications and more or less optional features that are less frequently useful. This is possible because OSS guarantees sane default values for most parameters the application can set. These defaults are based on 15 years of research and development and in most cases they are better than the values set by typical applications.

This API feature classification is actually just a documentation change.

Device naming

Traditionally all OSS implementations have provided set of “legacy” device files such as /dev/dsp and /dev/dsp0 to /dev/dspN for audio. In OSS 4.0 this device naming scheme has been abandoned. Now devices may have names such as /dev/oss/acme0/pcm0 or /dev/acme0_pcm0. The exact naming depends on the implementation or operating system. For this reason applications should not assume any given naming scheme. However the old pre-OSS4 naming is still supported by automatically generated symbolic links to the right “new style” device.
There are now three kinds (levels) of audio device names:

1. Default devices such as /dev/dsp, /dev/dsp_in, /dev/dsp_ac3 and so on. These devices are intended to be used as the default device until the user changes the application settings. These default device files point to the preferred audio device of its kind. System administrator can change the actual device mapping depending on user requirements. Note that /dev/dsp0 to /dev/dspN are not default devices but legacy ones (see below). Also device files such as /dev/dsp_in2 don't and will not exist. Default audio devices will be described bit later in this document.
2. Actual (new style) devices. After OSS 4.0 the device naming is designed to stay unchanged when new hardware gets added to the system or if the order the devices are initialized changes for any reason. The exact naming scheme depends on the operating system and applications must not make any assumptions about the mechanism. Instead device file names must be handled transparently. Applications can get the actual device name by using the ioctl interface (see below).
3. To stay compatible with earlier OSS versions the old style “legacy” device naming (/dev/dso0, /dev/dsp1, ..., /dev/dspN for audio devices) is still supported. However these device files are now symbolic links to the actual devices. This mapping is managed by the ossdevlinks utility in a way that accidental changes in numbering should be almost impossible. However the system administrator may decide to reset the numbering (using ossdevlinks -r). For this reason there is a risk that applications using legacy devices may get redirected to device that they should not use. New applications should use the new style names (2.) because they are much more reliable than the legacy devices.

Similar naming changes have been made also to the MIDI and mixer devices.

All applications written or modified for OSS 4.0 should use the new style naming. OSS 4.0 API contains a set of new ioctl calls (SNDCTL_SYSINFO, SNDCTL_AUDIOINFO, SNDCTL_MIXERINFO and SNDCTL_MIDIINFO) that return the right device name to use.

Alternatively applications should accept whatever device name the user decides to configure. It is permitted (but not recommended) to check validity of the device by calling stat(2) (NOT lstat) and to check that the device file exists and it's a character device.

Note that earlier OSS versions have created device files such as /dev/dspW* and /dev/audio*. These device files are no longer supported by OSS 4.0 and applications should use the device names mentioned above.

/dev/audio* devices are still used by some proprietary audio device interfaces of various operating systems such as SunOS/Solaris, AIX, HP-UX and Tru64Unix. Applications using such devices must use the right proprietary API instead of OSS.

Fully dynamic major/minor device numbering

In earlier OSS versions all sound devices shared the same major device number. It was fixed in some systems (14 in Linux) and dynamic in some others. In addition the available minor number space (0-255) was allocated to the devices in fixed and very creative way. However it became clear that the old scheme will not work any longer. There are already few large streaming audio servers which have run out of the available minor numbers allocated for audio devices.

In OSS 4.0 there are no fixed major or minor numbers. The operating system will allocate the major number for OSS. In addition each driver module may have separate major device number. Also minor number assignment is dynamic and minor numbers will be given in the order the (audio/MIDI/mixer/etc) devices get created. This order may be different each time OSS is loaded. OSS (the soundon script) will automatically (re)create the device files in /dev with the right major/minor device number.

This new scheme doesn't waste the minor device number space. So it's now possible to have up to 250 audio devices in the same system. In addition the new audio device numbering scheme saves minor numbers allocated for audio. The result is that OSS 4.0 can have practically unlimited number of audio devices installed in the system. With minor modifications in implementation it will be possible to meet any future requirements.

Application designers must not make any assumptions about the major/minor device numbering. Otherwise there may be problems with OSS 4.0.

/dev/sndstat

The old /dev/sndstat device file is still there. It returns the available devices as before. The device numbering has changed to reflect the /dev/dspN to new style device mappings as maintained by the ossdevlinks utility. In this way audio device N reported by /dev/sndstat will still map to the right audio device. However there are several special situations where this mapping fails. For this reason new applications should not try to parse /dev/sndstat to find the available devices. Instead the new device discovery interface (SNDCTL_SYSINFO, SNDCTL_AUDIOINFO, SNDCTL_MIXERINFO and SNDCTL_MIDIINFO) should be used. These ioctl calls will return the stable and reliable /dev/oss/... style device file name for each device.

The new ossinfo command replaces the /dev/sndstat device file. The (few) applications that still use /dev/sndstat to locate audio/mixer devices will work as before. However this method for finding devices must not be used in new applications written for OSS 4.0.

The ossinfo command is different from /dev/sndstat in that the device numbering has no significance.

Ossinfo will report devices in the order they were attached to the system. The actual device file names will be reported by “ossinfo -v”.

/dev/sequencer, /dev/sequencer2, /dev/music

The /dev/sequencer and /dev/music device interface for MIDI and synthesizer devices has been abandoned. This part of the OSS API was redundant with the official MIDI 1.0 specification by MMA. It defined a subset of MIDI 1.0 as a set of C macros that were specific only to OSS. That made maintaining and documenting the API too difficult. In addition application programmers who wanted to use given MIDI 1.0 feature had to figure out how to map this feature to the /dev/sequencer API.

For the above reasons we decided that the old MIDI/sequencer API was a disaster. Instead of trying to push it any further we decided to abandon the whole approach. So the old /dev/sequencer interface has been completely removed from OSS 4.0. Another reason for this was that this interface has not been used by any applications for years.

The replacement will be a new /dev/midi interface that handles the MIDI 1.0 specification fully transparently. This makes it very easy to port MIDI applications written for other architectures to use OSS 4.0. The only OSS specific feature in the new MIDI API is capability to embed timing information to the MIDI data written to or read from the device.

The new MIDI API is not yet available in OSS 4.0. It's scheduled to be included in OSS 4.1 which should be released in the near future.

Legacy mixer interface

One of the most limited parts of the original OSS API was the mixer interface. It was modeled for the first few computer sound cards in the market (Sound Blaster Pro, SB16, Gravis Ultrasound PAS16, etc). Many devices introduced during past 10 years have very little common with the first consumer sound cards and it has not been possible to map their mixer and control functionality to the old legacy mixer API.
For this reason OSS 4.0 has an entirely new mixer device architecture that doesn't make any assumptions about the device. Instead it uses hierarchic meta-data to describe the mixer and/or control panel structure in human understandable format. Mixer applications can use this meta-data to create their user interface in a way that will be fully compatible with any past, current and future sound device.

The old legacy mixer API is still included in OSS 4.0. However it will not be implemented for many devices. In particular the latest sound device architectures such as USB and High Definition Audio (HDA/Azalia) don't provide legacy mixers.

Audio related changes

Audio is the most commonly used feature of computer sound cards (or motherboard sound chips). For this reason majority of applications written for OSS use audio related functions. Examples of audio applications are games, audio and video players, audio recording workstations and streaming media encoders.

Audio functionality of OSS has been mostly unchanged during the history of OSS: Some new ioctl functions have been added over the years. Also some functions designed for certain special purpose have been obsoleted (they are still included in OSS 4.0 but not recommended to be used in modern applications).

Audio device naming

The most notable audio related change in OSS 4.0 is audio device file naming. In the very first OSS version (the original Linux sound driver in 1992) there was just one audio device which was called /dev/dsp. This name is very misleading since actually this device didn't provide any DSP (Digital Signal Processing) features at all. The name was just adopted because the first (Sound Blaster) sound cards called their audio playback and recording devices in this way.

Support for multiple audio devices was introduced in the next version. The second audio device was called /dev/dsp1. Later the number of sound cards supported at the same time was increased and the new devices were named as /dev/dsp2../dev/dspN.

Later the naming was changed so that the actual audio devices were named starting from /dev/dsp0. Meaning of the “old” first device (/dev/dsp) was allocated for the “default” audio device. In this implementation /dev/dsp was a symbolic link to one of the real devices such as /dev/dsp0. The system administrator was able to change the link manually.

In OSS version 3.99.0 the default /dev/dsp device was changed to be some kind of multiplexer device. If the primary audio device appeared to be busy then the application using /dev/dsp was redirected to the next free audio device. The search lists for this functionality were maintained by the ossctl utility.

The flat “legacy” device naming appeared to be inadequate when hot-pluggable devices (such as USB and FireWire) were introduced. The problem caused by such devices was that the /dev/dspN assignment was different depending on which devices were plugged in the system when OSS was started. This made flat device numbering unreliable and unacceptable because one day /dev/dsp7 was connected to card X and next day it was card Y.

The solution was to include the device name as a part of the device file name. In the latest OSS 4.0 versions device files are located under the /dev/oss directory which has a subdirectory for each “card” in the system. For example the first audio device of the first SB Live! Card in the system might be called as /dev/oss/sblive0/pcm0 (the exact naming depends on the operating system and applications must not make any assumptions about the naming). In this way the device names will stay static regardless of the order the cards/devices get added to the system.

The old “legacy” /dev/dspN style device names are still maintained by the ossdevlinks utility that will be run every time OSS is started (it may be necessary to run it if new hot-pluggable get inserted to a live system). The /dev/dspN devices will be symbolic links to the actual devices. Ossdevlinks will remember the earlier numbering. If new devices get added to the system they will get numbers after the last /dev/dspN device previously present in the system. If devices are removed then holes will be left in the numbering (just in case the removed device gets returned back to the configuration). This keeps the legacy device numbering static but the system administrator can reset it by running ossdevlinks -r. The device numbering reported by /dev/sndstat is maintained by ossdevlinks and it will match the /dev/dspN numbering.

Some applications written for earlier OSS versions keep just the /dev/dspN device number stored in their configuration settings. This is no longer recommended. Applications should store the full device file name and let the user to configure any device file name (not limited to /dev/dsp*).

New applications can locate the right device file name by using the devnode field SNDCTL_AUDIOINFO ioctl call.

Default and special purpose audio devices

The most common special purpose audio device file is /dev/dsp. By definition it's the audio device connected to the primary speakers or microphone. Applications that use hard coded audio device file should usually use /dev/dsp. The exact /dev/dsp device assignment can be managed by the system administrator but the exact method used for this depends on the OSS implementation and version being used.

OSS 4.0 defines few other default audio device files dedicated for certain special purposes. This is necessary because the default /dev/dsp device is not always suitable for every possible purpose. The special audio device file names currently defined are:

- /dev/dsp_in is the default recording device. Usually it's identical to /dev/dsp but in some cases the system administrator may want to use different recording device.
- /dev/dsp_out is similar to /dev/dsp_in but used for playback.
- /dev/dsp_mmap is a special audio device files to be used by audio playback applications (such as games) that use the mmap() method for audio output.
- /dev/dsp_multich is the default audio device to be used by applications that do multi channel audio output (4.0, 5.1, 7.1 and so on).
- /dev/dsp_spdifout, /dev/dsp_spdifin, /dev/dsp_ac3 are special device names for applications that do digital (S/PDIF or AES) recording or playback. /dev/dsp_ac3 is intended for applications who like to output AC3 or DTS encoded data to an external AV receiver.

Note that the above device files don't necessarily exist. This may mean that there is no device suitable for that kind of use. However it's also possible that the system is running older OSS version that doesn't know such device. For this reason applications may want to use /dev/dsp as the backup plan if the desired device doesn't exist. Alternative workaround is to ask the system administrator to create such device as a symbolic link to some other device file.

Audio engines and device files

In the earlier OSS versions there was no concept of audio device file or audio engine. Instead of audio “engines” had also a /dev/dspN device file (called as audio device in short). This caused pollution of audio device numbers when there were multiple devices supporting multiple audio engines. This in turn caused problems because the number of minor device numbers available for audio devices was limited. Another drawback was that the available audio device lists shown by applications became longer and longer.

The solution used by OSS 4.0 was breaking the concept of audio device to two parts. Now the system may have any number of audio engines. Some sound cards have just one audio device while “hardware mixing” cards such as SB Live!/Audigy may have up to 32 or more of them. However only one of identical (hardware mixing) audio engines will be visible and have an accessible audio device file in /dev. OSS will automatically select the first available audio engine when an application opens this visible device.

The “ossinfo -a -v” command reports the visible audio device files and the device nodes in /dev that can be used to access the device. An application can use the SNDCTL_AUDIOINFO ioctl call to access this information.

It's possible to get a list of available audio engines by using “ossinfo -e” or SNDCTL_ENGINEINFO. However this information has little or no use.

Audio device discovery

Audio applications have traditionally used freely configurable device names or hard coded /dev/dsp in their configuration settings. Such applications are not affected by the device naming changes of OSS 4.0.

The traditional method for getting list of available audio devices has been parsing /dev/sndstat. However this approach is no longer recommended. It will continue working in the future but minor layout changes made to the format of /dev/sndstat may cause problems in the future. Also the device numbering will become undefined or invalid if the system administrator decides to reset the legacy device numbering by running “ossdevlinks -r”.

The way how new audio applications should get list of the available audio devices is following:

1. Call SNDCTL_SYSINFO to get the numaudios field.
2. Call SNDCTL_AUDIOINFO for each audio device in the system.
3. Let the user to select the device by “clicking” it's name.
4. Store the device file name returned in the devnode field of SNDCTL_AUDIOINFO as the device file name in configuration settings.
5. If running the device selection later use the stored devnode name to locate the currently configured audio device.

The caps field returned by SNDCTL_AUDIOINFO can be used to find out the features supported by the device (such as recording, playback or both).

New virtual mixing architectures

Earlier OSS versions supported virtual mixing using the softoss driver. In this way it was possible to share audio playback devices between multiple applications at the same time. However the old style softsoss virtual mixing devices were visible as separate devices in the device configuration. It was possible to have just one of the audio devices in the system shared in this way.

The new virtual mixer (VMIX) architecture of OSS is significantly improved. In OSS 4.0 there are no virtual mixing devices visible in the system. Instead applications opening the “real” physical device will be automatically redirected to the first available virtual mixer engine attached to it. In this way multiple applications can access the same audio device transparently at the same time.

Exclusive access to audio devices

After introduction of the new transparent virtual mixing concept the audio devices are no longer dedicated exclusively to any given application. In most cases this doesn't cause any problems. However certain applications may want to prevent other applications from using the same device and disturbing their own operation. This is necessary for example when a application outputs specially encoded signals that must in no case be interrupted by random sounds such as system beeps. Another example is DVD players that output AC3 or DTS encoded signals to an external AV receiver.

To gain exclusive access to the audio device applications can open it using O_EXCL flag. This disables redirection to the virtual mixing engines and prevents other applications from disturbing the operation. However the drawback is that any application already running on the device will prevent applications using O_EXCL from opening it. In addition using O_EXCL unnecessarily will prevent other applications from running while there is no real reason to prevent it. For this reason application developers must be very careful when using O_EXCL. It's recommended that this mode is only enabled if the user has asked it in the audio configuration screen or by using some command line switch. Ordinary applications such as games must in no case use O_EXCL.

Automatic format conversions

One of the primary design goals of the OSS API has been ti provide full device abstraction to the applications. The primary method used for this has been fully automatic format and sample rate conversions from the format used by the application to the nearest format supported by the actual device. In this way most applications don't need to check what kind of sampling rates or sample format the device supports. The application simply tells which parameters it likes to use and OSS will take care of the rest.

Unfortunately there are certain (rare) types of applications which don't tolerate this kind of automatic conversions. For example applications using mmap() must disable automatic format conversions. This can be done by calling the SNDCTL_DSP_COOKEDMODE ioctl.

Full duplex audio

There have been two ways to implement full duplex (simultaneous recording and playback). Some devices and/or OSS implementations have used one while the others have used the second. These approaches are:

1. One device file (which is opened only once) is used for both directions (opened with O_RDWR). Applications need to call SNDCTL_DSP_SETDUPLEX to enable full duplex (because few older devices provided limited features when recording and playback were used at the same time).
2. Separate (unidirectional) device files are used for both directions. The recording one is opened with O_RDONLY and the playback device is opened with O_WRONLY. The SNDCTL_DSP_SETDUPLEX call is not used in this case. However the user should (usually) take care that both devices are synchronized to the same crystal oscillator (drift in the rates will cause problems with some applications).

The first approach seems to be more common than the second. Unfortunately this approach doesn't work with all audio devices. The latest hardware architectures (such as USB and Hdaudio/Azalia) have multiple input and output devices and it's not possible to predict which inputs and outputs belong logically together. For this reason the drivers for such architectures will create separate device files for input and output.

Full duplex applications written or modified for OSS 4.0 should use the second approach (or provide it as an alternative even if the first scheme is used by default). In this way it will be guaranteed that application will work with all devices.

The new virtual mixing architecture (vmix) makes it possible to use the first approach also with devices that have separate input and output devices. However this may require manual configuration adjustments by the user.

Loop back audio devices

OSS 4.0 has special audioloop driver that creates number of server/client audio device pairs. The client device behaves like any other audio device but it's not connected to any real audio device. Instead it's an unidirectional pipe that is connected to the application in the server side. The client can use the device for input or output depending on the mode supported by the server. If the server has opened the loop back pipe for recording the client can do playback and vice versa. The client side device can only be opened when the server side is open (otherwise ENXIO will be returned).

The server side of the loop back pipe is slightly different that normal audio devices. Almost any audio application can work as the server. However there are few differences:

- There is no legacy /dev/dspN device for the server side. Instead the application working as the server must open one of the /dev/oss/audioloop0/serverN devices (naming may be different under some operating systems or OSS implementations).
- The client side will be forced to use the same sampling rate and sample format as used by the server side. However automatic format conversions will take care of the other formats.
- The first read or write call made by the server will wait infinitely until some application opens the client side and calls read/write for the first time. After this both sides will work in sync at rate controlled by an interval timer.
- When the client side closes the device the server side application will get errno=ECONNRESET when it calls read/write for the next time. After this the server should close the device, save any recorded data and reopen the device again.

Practically any OSS compatible audio application can be used as the server. However the problem is that some audio players seem to probe the device capabilities and then close and reopen the device for actual playback. This will cause problems with applications that are not prepared to handle this kind of situations.

Another kind of loop back devices are the loop back recording devices provided by the vmix driver. Applications using the vmix loop back devices may record the output “mix” of all applications using the client side output devices. If there are no applications playing audio the loop back input device will return silent data.

Also the vmix loop back devices are hidden from legacy applications (no /dev/dspN device link). So applications using them must be able to open the actual device file such as /dev/oss/vmix0/loop0).

Warnings for memory mapped I/O (mmap)

Memory mapped I/O is a special transfer mode provided by most (but not all) OSS drivers. Applications using this method will bypass practically all features of OSS. For this reason they must be able to use whatever sampling rate, sample format and number of channels the device supports.

A very common mistake made by the developers of mmap applications is assuming that all devices will support given parameters such as 44.1 kHz, 16 bits and stereo. This is fatal error because many devices may be limited to different sampling rate (usually 48 kHz) or sample format (32 bits instead of 16 bits). IF you for any reason decide to use mmap you must ensure that the application will be able to support all possible sampling rate, sample format and number of channels combinations (including the ones to be invented in the future :).

Applications using mmap() should open /dev/dsp_mmap (or some user configurable device) instead of hard coded /dev/dsp. In addition such applications must disable automatic format conversions by calling SNDCTL_DSP_COOKEDMODE. Failure of doing this will cause troubles in most systems.

Most mmap applications seem to be based on some earlier one that is incorrect in many ways. For this reason it's important to contact 4Front technologies (developer@opensound.com) before doing anything.

Warnings for non-blocking audio usage

A common mistake in audio applications is trying to avoid blocking even when there are no reasons to do so. This makes the application very complicated and sensitive to minor differences in behavior between audio devices and OSS versions/implementations.

The most common mistake is opening the audio device with O_NONBLOCK and then forgetting to reset this flag after open. In some OSS implementations the open call may (incorrectly) wait until the device becomes available (the previous application closes it). This is against the OSS specification but unfortunately applications may need to use O_NONBLOCK as a workaround to this bug.

However the O_NONBLOCK flag has a defined meaning in the OSS API. It turns on non-blocking read/write mode. Unfortunately applications that are not prepared to handle strange behavior of non-blocking reads and writes will not work at all. Instead they will produce garbled audio or crash in the beginning. For this reason applications should not use O_NONBLOCK in open or at least they should reset this mode flag by calling fcntl().

Non-blocking I/O (O_NONBLOCK) may be useful in some (very rare) audio applications. The OSS API also provides ioctl calls such as SNDCTL_DSP_GETOSPACE that can be used to implement non-blocking behavior in many different ways. However we have studied number of audio applications and found out that many of them use non-blocking mode in very stupid way. It's common that applications call SNDCTL_DSP_GETOSPACE (or something else) to find out if there is room to write given amount of data to the device. Then they just calculate how long it will take the buffer to drain and then call usleep() wait for that time. This is unnecessary, redundant and very inefficient because OSS will automatically handle the right timing.

Non-blocking operation is only necessary in applications that are busy with some other computations and if blocking in read or write slows down such computations.

Obsoleted functionality

OSS provides full backward compatibility for the documented ioctl calls. However there have been many ioctl calls and other features that have never been documented. Some of them may have existed only in soundcard.h with no driver code that implements them. During development of OSS 4.0 most of these features have been removed from soundcard.h.

In addition there are functions that have been necessary in some very early prototype versions of OSS. Some functions may have been developed for a given application but later replaced by something else. Finally some calls are just unnecessary because OSS can handle such features automatically without any actions by the application.

For this reason OSS application developers should be very careful when using older applications as templates for new ones. All ioctl calls that can be used are documented in the official OSS 4.0 Programmer's Guide. In addition many ioctl calls are now marked as obsolete and a replacement method is given in the documentation. Do not use older OSS documentation or old Linux audio programming related books when writing new applications for OSS.

Warnings for using mixer devices in audio applications

One of the top10 questions made by audio application developers has been which mixer device I should use for /dev/dspN. The question is (and has always been) none. There is no relationship between an audio device and any mixer device in the system. Many sound cards don't have a mixer at all and some others may have several of them. Even the card has a mixer it doesn't necessarily have the functionality expected by the application.

In fact audio applications are not permitted to access any of the /dev/mixer devices. They are reserved to be used by dedicated mixer applications to control system-wide settings. An audio application that uses a mixer device may cause serious side effects to other applications running in the system. In particular this will happen if it's changing settings of a wrong mixer.

The mixer functions audio applications have traditionally tried to use are recording source selection and playback/recording volume control. In OSS 4.0 these functions have been separated from the /dev/mixer interface. There is a set of new audio ioctl calls such as SNDCTL_DSP_SETPLAYVOL that will automatically access the right device. In addition setting changes made by these calls are (usually) limited only to the given audio device. They don't have global side effects that may cause damage to other applications running in the system.

Mixer and control panel features and changes

The old “legacy” mixer interface of OSS was modeled for some early consumer sound cards such as SB16. Such devices have limited set of audio connectors with fixed functionality. Most recent audio device architectures such as USB and HD audio (Azalia) use fully dynamic mixer architecture that is impossible to map to the legacy mixer API. For this reason we have added a new fully dynamic “mixer extension” API to OSS. This API doesn't make any kind of assumptions about the device implementation.

The new OSS mixer architecture is based on meta data that describes the mixer hierarchy. This meta data makes it possible to the mixer applications to create their user interface dynamically based on the features of the device. This approach makes it possible to support any past, current and future sound device without modifications to the mixer/control panel programs.

Most common mixer tasks such as recording /playback volume control and recording source selection now have ioctl functions in the audio API. In this way audio applications can perform these tasks very easily without making assumptions about the features of the hardware being used.

Upcoming MIDI API

MIDI support of earlier OSS versions has been seriously flawed. The /dev/sequencer (/dev/music) architecture was modeled for some early synthesizer chips used in sound cards (OPL2/3 and Gravis Ultrasound). The major drawback was that this API duplicated all features of the official MIDI 1.0 specification (by MMA) with macros that were specific to OSS alone. This made it very difficult to port MIDI applications from other architectures. Also documenting the OSS specific macros appeared to be mission impossible. For this reason the old sequencer architecture was abandoned (this should not cause any problems because this mistaken API has not been used by any applications currently available).

At this moment the new MIDI API is under development and (due to few bugs) it will not be enabled in the first OSS 4.0 versions.

The main idea of the upcoming OSS MIDI API is to let applications to send and receive any MIDI 1.0 compatible messages. There is nothing OSS specific other than the way to embed MIDI timing information. Each packet of MIDI data written to the MIDI port can have a header that specifies the precise time when the MIDI message(s) included in the packed should be processed. For recorded data OSS will include the exact reception time in the header.

An application may use multiple MIDI device (port) at the same time. They may share the same timer device or each of them may have an independent timer. Timers are based on the MIDI timing architecture as defined in the official MIDI 1.0 specification.

System control and information functions

In early OSS versions there were usually just one (or two) audio, MID and mixer device. There was no need to select the device and most applications used just the first (default) one such as /dev/dsp for audio. Later applications have used /dev/sndstat to find out which devices are available in the system.

In OSS 4.0 there is fully featured ioctl interface to find out what kind of devices there are in the system. The new SNDCTL_SYSINFO, SNDCTL_AUDIOINFO(SNDCTL_ENGINEINFO, SNDCTL_MIXERINFO, SNDCTL_MIDIINFO and SNDCTL_CARDINFO ioctl calls make it possible to get detailed information about the devices installed in the system. The ossinfo utility shipped with OSS uses these calls to show details of the system in human readable form.