Open Sound System™

Programmer's Guide

Version 1.0

4Front
Technologies

Table of Contents

Introduction

This manual describes the Open Sound System (OSS) application programming interface. It starts with general background information on OSS devices and programming techniques. It then gets into a detailed description of programming the mixer, audio, MIDI, raw music and the new Virtual Mixer and SoftOSS devices. Also covered are some advanced programming topics and platform specific issues. The manual finishes up with references to further information and a glossary of technical terms used in the manual.

It is assumed that the reader has OSS installed and working and has a basic familiarity with C programming on the platform on which OSS is being used.

OSS is continuously under development, with new features being constantly added. This manual is a work in progress that attempts to document these features. You should periodically check the 4Front Technologies web site to obtain the latest version of the manual. We also welcome comments or corrections to the manual – please send them by e-mail to `support@opensound.com`.

Additional late-breaking information can be found in the `Readme` and other files that come with your copy of the OSS software. For issues related to installing OSS, see the *Open Sound System Installation Guide*.

Background

The Open Sound System (OSS) is a device driver for sound cards and other sound devices under various UNIX and UNIX-compatible operating systems. OSS was derived from the sound driver written for the Linux operating system kernel. The current version now runs on more than a dozen operating system platforms and supports most popular sound cards and sound devices integrated on computer motherboards.

Sound cards normally have several different devices or ports which produce or record sound. There are differences between various cards, but most have the devices described in this section.

The *digitized voice* device (also referred to as a codec, PCM, DSP or ADC/DAC device) is used for recording and playback of digitized sound.

The *mixer* device is used to control various input and output volume levels. The mixer device also handles switching of the input sources from microphone, line-level input and CD input.

The *synthesizer* device is used mainly for playing music. It is also used to generate sound effects in games. The OSS driver currently supports two kinds of synthesizer devices. The first is the Yamaha FM synthesizer chip which is available on most sound cards. There are two models of this FM chip. The Yamaha OPL-2 is a two operator version which was used in early sound cards such as the AdLib and SoundBlaster 1 and 2. It has just 9 simultaneous voices and is not capable of producing

very realistic instrument sounds. The OPL-3 is an improved version that supports 4 operator voices which offer the ability to produce more realistic sounds. The second type of synthesizer devices are the so-called wave table synthesizers. These devices produce sound by playing back pre-recorded instrument samples. This method makes it possible to produce extremely realistic instrument timbres. The Gravis UltraSound (GF1) is an example of a wave table synthesizer.

A MIDI interface is a used to communicate with devices, such as synthesizers, that use the industry standard MIDI protocol. MIDI uses a serial interface running at 31.5 kbps which is similar to (but not compatible with) standard PC serial ports. The MIDI interface is designed to work with on-stage equipment like synthesizers, keyboards, stage props, and lighting controllers. MIDI devices communicate by sending messages through a MIDI cable.

Most sound cards also provide a joystick port and some kind of interface (IDE, SCSI, or proprietary) for a CD-ROM drive. These devices are not controlled by OSS but there are typically separate drivers available.

OSS API Basics

The application programming interface (API) of the OSS driver is defined in the C language header file `<soundcard.h>`[1].

The OSS software ships with a copy of the header file in the `include/sys` subdirectory. You may have older versions of the include file that are included with your operating system (Linux distributions typically include the older OSS/Free driver, for example). It usually causes no harm to use the older header file but you will not be able to use some of the newer features only provided in OSS. Very old versions may also cause compatibility problems. To avoid this, you can either point to the OSS header files when compiling applications (e.g. use the compile option `"-I/usr/lib/oss/include"`) or install the header file in a standard system header file location (e.g. `/usr/include/sys`).

If you get compile errors when building an application, verify that you are using the version of `<soundcard.h>` supplied with OSS.

Device Files Supported by OSS

The OSS driver supports several different types of devices. These are described in the following sections.

---

[1] There is another include file for the Gravis UltraSound card, `<ultrasound.h>`, but normally it should not be required. It is not actually part of the OSS API, but a hardware specific extension to it.

`/dev/mixer`

The mixer device files are used primarily for accessing the built-in mixer circuits of sound cards. A mixer makes it possible to adjust playback and recording levels of various sound sources. This device file is also used for selecting recording sources. Typically a mixer will control the output levels of the digital audio and FM synthesizer and also mix it with the CD input, line level input and microphone input sources.

The OSS driver supports several mixers on the same system. The mixer devices are named `/dev/mixer0`, `/dev/mixer1`, etc. The device file `/dev/mixer` is a symbolic link to one of these device files (usually the first mixer, `/dev/mixer0`).

`/dev/sndstat`

This device file is provided for diagnostic purposes, and unlike all of the other sound devices, produces its output in human readable format. The device prints out information about all of the ports and devices detected by the OSS driver. Running the command "`cat /dev/sndstat`" will display useful information about the driver configuration. It should be noted that the output of `/dev/sndstat` is *not* intended to be machine readable and may change without notice in future versions of OSS.

`/dev/dsp` and /dev/audio

These are the main device files for digitized voice applications. Any data written to this device is played back with the DAC/PCM/DSP device of the sound card. Reading the device returns the audio data recorded from the current input source (the default is the microphone input).

The `/dev/audio` and `/dev/dsp` device files are very similar. The difference is that `/dev/audio` uses logarithmic mu-law encoding by default while `/dev/dsp` uses 8-bit unsigned linear encoding. With mu-law encoding a sample recorded with 12 or 16-bit resolution is represented by one 8-bit byte. Note that the initial sample format is the only difference between these device files. Both devices behave similarly after a program selects a specific sample encoding by calling `ioctl`. The `/dev/audio` device is provided for compatibility with the sound device provided on Sun workstations running SunOS. These device files can be used for applications such as speech synthesis and recognition and voice mail.

Although `/dev/audio` provides minimal compatibility with Sun's API, there is no support for Sun's `ioctl()` interface. OSS under Solaris emulates these calls to some degree to provide compatibility with existing Solaris and SunOS applications, however this emulation is not officially supported by 4Front Technologies.

The OSS driver supports several codec devices on the same system. The audio devices are named `/dev/dsp0`, `/dev/dsp1`, etc. The file `/dev/dsp` is a symbolic link to one of these device files (usually `/dev/dsp0`). A similar naming scheme is used for `/dev/audio` devices.

/dev/sequencer

This device file is intended for electronic music applications. It can also be used for producing sound effects in games. The /dev/sequencer device provides access to any internal synthesizer devices of the sound cards. In addition, this device file can be used for accessing any external music synthesizer devices connected to the MIDI port of the sound card as well as General MIDI daughtercards connected to the WaveBlaster connector of many sound cards. The /dev/sequencer interface permits control of up to 15 synthesizer chips and up to 16 MIDI ports at the same time.

`/dev/music` (formerly /dev/sequencer2)

This device file is very similar to `/dev/sequencer`. The difference is that this interface handles both synthesizer and MIDI devices in the same way. This makes it easier to write device independent applications than it is with `/dev/sequencer`. On the other hand, `/dev/sequencer` permits more precise control of individual notes than `/dev/music`, which is based on MIDI channels.

CAUTION
Unlike the other device files supported by OSS, both `/dev/sequencer` and `/dev/music` accept formatted input. It is not possible, for example, to play music by just sending MIDI files to them.

`/dev/midi`

These are low level interfaces to MIDI bus ports that work much like TTY (character terminal) devices in raw mode. The device files are not intended for real-time use – there is no timing capability so everything written to the device file will be sent to the MIDI port immediately. These devices are suitable for use by applications such as MIDI SysEx and sample librarians.

There device files are named `/dev/midi`00, `/dev/midi`01, etc. (note the two digit device numbering). The device `/dev/midi` is a symbolic link to one of the actual device files (typically `/dev/midi`00).

TIP
Many of the sound device files are numbered from 0 to n. It is possible to find out the proper number by using the command "cat `/dev/sndstat`". The output produced contains a section for each device category. Devices in each category are numbered, with the number corresponding to the number in the device file name. The numbering of devices depends on the order that the devices have been initialized during startup of the driver. This order is not fixed, so don't make any assumptions about device numbers.

/dev/dmfm

This is a raw interface to FM synthesizers. It provides low level register access to the FM sound chip. Devices are named /dev/dmfm0, /dev/dmfm1, etc.

/dev/dmmidi

This is the raw interface to MIDI devices. It provides direct TTY-like access to the MIDI bus for specialized applications. Devices are named /dev/dmmidi0, /dev/dmmidi1, etc.

Device Numbering

The OSS device files share the same major device number. One the Linux platform the major device number is 14; on other operating systems it is usually something else. The minor number assignment is given in table 1 below.

The four least significant bits of the minor number are used to select the device type or class. If there is more than one device in a class the upper 4 bits are used to select the device. For example, the class number of /dev/dsp is 3. The minor number of the first device, /dev/dsp0, is 3 and for the second device, /dev/dsp1, is 19 (16 + 3).

Table 1 - OSS Device Numbers (Linux platform)

Major	Minor	Name
14	0	/dev/mixer0 .. /dev/mixer4
14	1	/dev/sequencer
14	2	/dev/midi00 .. /dev/midi04
14	3	/dev/dsp0 .. /dev/dsp15
14	4	/dev/audio0 .. /dev/audio15
14	5	/dev/dspW0 .. /dev/dspW15
14	6	/dev/sndstat
14	7	/dev/dmfm0
14	8	/dev/music
-	-	/dev/audio (link to /dev/audio0)
-	-	/dev/audioctl (link to /dev/mixer0)
-	-	/dev/dmdsp0 (link to /dev/dsp)
-	-	/dev/dmmidi0 (link to /dev/midi)

-	-	/dev/dsp (link to /dev/dsp0)
-	-	/dev/dspW (link to /dev/dspW0)
-	-	/dev/dspdefault (link to /dev/dsp0)
-	-	/dev/midi (link to /dev/midi00)
-	-	/dev/mixer (link to /dev/mixer0)
-	-	/dev/sequencer2 (link to /dev/music)
-	-	mixer (link to /dev/mixer0)

Programming Guidelines

One of the main goals of the OSS API is full source code portability of applications between systems supporting OSS. This is possible if certain guidelines are followed when designing and programming the audio portion of an application. It is even more important that the rest of your application is written in a portable manner. In practice, most portability problems in the current sound applications written for Linux are in the program modules that perform screen handling. Sound related portability problems are usually just endian problems.

The term portability doesn't just refer to the program's ability to work on different machines running different operating systems. It also includes the ability to work with different sound hardware. This is even more important than operating system portability since differences between the current and future sound devices are likely to be relatively large. OSS makes it possible to write applications which work with all possible sound devices by hiding device specific features behind the API. The API is based on universal physical properties of sound and music rather than hardware specific properties.

This section lists a number of areas to watch for that will help improve the likelihood that OSS applications are portable.

Use API Macros

The macros defined in <soundcard.h> provide good portability since possible future changes to the driver's internals will be handled transparently by the macros. It is possible, for example, to use the /dev/sequencer device by formatting the event messages in the application itself. However it is not guaranteed that this kind of application works in all systems. You should use the macros provided for this purpose in the sound header file.

Device Numbering and Naming

In some cases there might be several sound devices in the same system (e.g. a sound card and on-board audio). In such cases the user may have valid reasons for using different devices with different applications. This is not a major problem with open source applications where the user has the freedom to change device names in source code. However, the situation is different when the source code for the program is not available. In either case it is preferable that the user can specify the devices in the application's preferences or configuration file. The same is true with MIDI and synthesizer numbers used in `/dev/sequencer` and `/dev/music`. Design your application so that it is possible to select the device numbers. In particular, don't hard code your program with device names which have a numeric suffix. For example, it is preferable to program your application to use `/dev/dsp` and not `/dev/dsp0`. While `/dev/dsp` is usually a symbolic link which points to `/dev/dsp0`, the user may have reasons to change audio applications to use `/dev/dsp1` by changing the link. In this case an application that uses `/dev/dsp0` directly will use the incorrect device.

Endian Convention

This is a serious problem with applications using 16-bit audio sampling resolution. Most PC sound cards use little-endian encoding of samples. This means that there are no problems with audio applications on little-endian machines such as Intel x86 and Alpha AXP. In these environments it is possible to represent 16-bit samples as 16-bit integers (signed short). This is also not a problem in big-endian machines which have built-in big-endian codec devices. However, the endian convention is a big problem in mixed endian systems. For example, many RISC systems use big-endian encoding but it is possible to use little-endian ISA or PCI sound cards with them. In this case, using 16-bit integers (signed short) directly will produce just white noise with a faint audio signal mixed in with it. This problem can be solved if the application properly takes care of the endian convention using standard portability techniques.

Don't Use Undefined Default Conditions

For most parameters accepted by the OSS driver there is a defined default value. These defined default values are listed in this manual at the point where the specific features are discussed. However, in some cases the default condition is not fixed but depends on characteristics of the machine and the operating system where the program runs. For example, the timer rate of `/dev/sequencer` is fixed and depends on the system timer frequency parameter (`HZ`). Usually the timer frequency is 100 Hz which gives a timer resolution of 0.01 seconds. However there are systems where the timer frequency is 60 or 1024 Hz. Many programs assume that the tick interval is always 0.01 seconds and will not work on these systems. The proper way to handle this kind of variable condition is to use the method defined for querying the default value.

Don't Try to Open the Same Device Twice

Most device files supported by the OSS driver have been designed to be used exclusively by one application process (`/dev/mixer` is the only exception). It is not possible to re-open a device while the same device is already open by another process. Don't try to overcome this situation by

using `fork`[2] or other tricks. This may work in some situations, but in general the result is undefined.

Avoid Extra Features and Tricks

Think carefully before adding a new feature to your application. A common problem in many programs is that there are lot of unnecessary features[3] which are untested and just cause problems when used. A common example of an extra feature is including a mixer interface in an audio playback application. It is very likely that the feature will be poorly implemented and cause troubles on some systems which are different from the author's. In this case a separate mixer application is probably a more flexible and reliable tested solution.

Don't Use Undocumented Features

There are features that are defined in `<soundcard.h>` but which are not documented here. This features are left undocumented for a reason. Usually they are obsolete features which are no longer supported and will disappear in future driver versions. Some of them are features which have not yet been fully tested and may cause problems on some systems. A third possibility is there are undocumented features which are device dependent and work with only few devices (which are often obsolete). Therefore, avoid the temptation of using features just because they were found when browsing `<soundcard.h>`.

Avoid Invalid Assumptions

There are many common assumptions which make programs non-portable or highly hardware dependent. The following is a list of things that are commonly misunderstood.

Mixer

Not all sound cards have a mixer. This is true with some older sound cards, some sound cards that are not yet fully supported by the OSS driver, and some high end professional ("digital only") devices which are usually connected to an external mixer. Your program will not work with these cards if it requires the availability of a mixer.

Not all mixers have a main volume control. For some reason almost all mixer programs written for the OSS API make this assumption.

The set of available mixer controls is not fixed, but varies between devices. Your application should query the available channels from the driver before attempting to use them (alternatively the

---

[2] Using `fork` is acceptable if only one process actually uses the device. The same is true for multi-threaded programs.

[3] This is a general problem, not one that just applies to sound applications. One of the original design tenets of UNIX was that each program should do exactly one thing well.

application can just selectively ignore some error codes returned by the mixer API but this is a really crude and semantically incorrect method).

Try to avoid automatic use of the main volume mixer control. This control affects the volume of *all* audio sources connected to the mixer. Don't use it for controlling the volume of audio playback since it also affects the volume of an audio CD that may be playing in the background. Your program should use only the PCM channel to control the volume of audio playback.

`/dev/dsp` and `/dev/audio`

The default audio data format is 8 kHz/8-bit unsigned/mono (`/dev/dsp`) or 8 kHz/mu-Law/mono (`/dev/audio`). However, this is not always true. Some devices simply don't support the 8 kHz sampling rate, mono mode or 8-bit/mu-Law data formats. An application which assumes these defaults will produce unexpected results (such as 144 dB noise) with some hardware (such as future 24-bit only sound hardware).

`/dev/sequencer` and `/dev/music`

As mentioned earlier, don't assume that the timer rate of `/dev/sequencer` is 100 Hz (0.01 second). This is not true on all platforms – Linux on Alpha uses a much higher system clock rate, for example.

Set all of the timing parameters of `/dev/music` before using the device. There are no globally valid default values.

Don't assume that there is always at least one MIDI port and/or one synthesizer device. There are sound cards which have just a synthesizer or just a MIDI port.

Don't try to use a MIDI port or synthesizer device before first checking that it exists.

Mixer Programming

Introduction

Most sound cards have some kind of mixer which can be used for controlling volume levels. The OSS API defines a device file, `/dev/mixer`, which can be used to access the mixer functions of the card. It is possible that there is more than one mixer if there are several sound cards installed on the system. The actual mixer device files are `/dev/mixer0`, `/dev/mixer1`, etc. with `/dev/mixer` being just a symbolic link to one of these device files (usually `/dev/mixer0`, but the user has the freedom to assign the link differently).

NOTE

It is possible that no mixers are present on the system. Some sound cards simply don't have any mixer functionality. This is common with some old sound cards. There are also some high end professional sound cards that don't have a mixer. Don't assume that there is a mixer in every system. All systems have `/dev/mixer0` but the `ioctl` calls will fail and set `errno` to `ENXIO` if no mixer is present. Your program should be prepared to handle `ENXIO` returned by any of the `ioctl` calls.

The OSS mixer API is based on channels. A mixer channel is a numbered object which represents a physical control or slider of the mixer. Each of the channels have independently adjustable values which may vary between 0 (off) and 100 (maximum volume). Most of the channels are stereo controls, so it is possible to set values for both stereo channels separately which permits the control of balance. The mixer API contains a few `ioctl` calls for setting and getting the values of these mixer channels.

In addition to volumes, the mixer API also controls the selection of recording sources. With most sound cards it is possible to record simultaneously only from one source, while a few cards (such as the PAS16) allow several recording sources to be active at the same time. After a system reset the microphone input is usually selected as the recording source (but there is no guarantee that this is always true).

NOTE

Changes to the mixer settings will remain active until the system is rebooted or changed again. The driver doesn't change the mixer settings unless instructed to do so by commands.

The third class of mixer `ioctl` calls are functions used for querying the capabilities of the mixer. With these calls it is possible to check which mixer channels are actually present and which can be used as input sources.

> **NOTE**
> The set of available mixer channels is not fixed since different sound cards have different mixers. For this reason it is important to check which channels are available before attempting to use the mixer. It is possible that even the main volume setting is missing. The driver will return -1 and set `errno` to the error code `EINVAL` if a nonexistent mixer channel is assigned.

Mixer channels are bound to pins of the mixer chip. Some mixer chips are used in cards made by several manufacturers. It is possible that some manufacturers have connected the mixer chip in a different way than the others. In this case some mixer channels may have a different meaning than defined below.

It is recommended that mixer functionality is not embedded in programs whose main function is something else (for example, audio). In some sound cards the hardware level mixer implementation may differ significantly from the normal situation. In this case, only a mixer program tailored for that card works properly. Adding mixer functionality to programs may cause unexpected support problems in future.

Types of Mixer Programs

The mixer API of OSS permits writing of generic mixer programs which work with almost any sound card. This is possible only if the program uses the query functions of the API to check the capabilities of the device before trying to use it.

It is also possible to design a mixer so that it works best with a particular sound card. In this way it is easier to design a nice looking GUI which matches the hardware properly. Even in this case it is a good idea to check that the required mixer channels are actually present by using the query `ioctl` functions defined below. In this case you should clearly indicate in the documentation for the program that it requires a particular sound card.

Mixer Channels

The mixer channels have an unique number between 0 and 30. The file `<soundcard.h>` defines some mnemonic names for the channels. Note that these are the current ones, new ones could be added in the future.

The macro `SOUND_MIXER_NRDEVICES` gives the number of channels known when this version of `<soundcard.h>` was written. A program should not try to access channels greater or equal than `SOUND_MIXER_NRDEVICES`.

The channels currently known by the driver are shown in table 2.

Table 2 - Mixer Channels

Macro	Description
SOUND_MIXER_VOLUME	Master output level (headphone/line out volume)
SOUND_MIXER_TREBLE	Treble level of all of the output channels
SOUND_MIXER_BASS	Bass level of all of the output channels
SOUND_MIXER_SYNTH	Volume of the synthesizer input (FM, wavetable). In some cases may be connected to other inputs too.
SOUND_MIXER_PCM	Output level for the audio (Codec, PCM, ADC) device (`/dev/dsp` and `/dev/audio`)
SOUND_MIXER_SPEAKER	Output volume for the PC speaker signals. Works only if the speaker output is connected directly to the sound card. Doesn't affect the built in speaker, just the signal which goes through the sound card. On some sound cards this is actually a generic mono input which may control some other function. For example, in the GUS Max this control adjusts the volume of the microphone signal routed to line out.
SOUND_MIXER_LINE	Volume level for the line in jack
SOUND_MIXER_LINE1 SOUND_MIXER_LINE2 SOUND_MIXER_LINE3	Generic mixer channels which are used in cases when the precise meaning of a physical mixer channel is not known. The actual meaning of these signals is vendor defined. Usually these channels are connected to the synth, line-in and CD inputs of the card but the order of the assignment is not known to the driver.
SOUND_MIXER_MIC	Volume for the signal coming from the microphone jack. In some cases his signal controls only the recording volume from the microphone and on some cards it controls the volume of the microphone signal routed to the output of the card too. In some cards the microphone is not connected to the true microphone input at all but to one of the line level inputs of the mixer chip.
SOUND_MIXER_CD	Volume level for the input signal connected to the CD audio input.
SOUND_MIXER_IMIX	A recording monitor channel on the PAS16 and some other cards. It controls the output (headphone jack) volume of the selected recording sources while recording. This channel only has effect when recording.
SOUND_MIXER_ALTPCM	Volume of the alternate codec device (such as the SoundBlaster emulation of the PAS16 cards).
SOUND_MIXER_RECLEV	Global recording level setting. In the SoundBlaster16 card this controls the input gain, which has just 4 possible levels.

It is important to remember that the exact effect of mixer channels may be slightly different in some sound cards. For this reason, try to avoid too specific descriptions of the mixer channels in documentation of a mixer program.

Querying the Capabilities of the Mixer

The mixer interface of OSS has been designed so that it is possible to compile a mixer program on one system and to use it on another system with different sound hardware. This is possible only if the mixer program follows some guidelines. It has to query for the hardware configuration before taking any other actions with the mixer interface. It does no harm if the program tries to change the volume of a channel without first querying if the channel is valid, since the `ioctl` call will return an error if there is something wrong with the request. However, if a mixer program shows mixer channels that are not valid for the sound hardware, the user may become confused. The `ioctl` calls described in the next section give programs a way to determine the correct sound hardware capabilities.

Using the Mixer Query Interface

All query functions of the mixer API return a bit mask in an integer variable which is passed as an argument to the `ioctl` call. The following code fragment shows the generic method used for the calls described in the following sections.

Listing 1 - Checking for Device Capabilities

```
int mask;
if (ioctl(mixer_fd, SOUND_MIXER_READ_xxxx, &mask) == -1) {
 /* Mixer capability is not available - handle this gracefully ... */
}
```

It is important to note that any `ioctl` call for the mixer API may return -1 and set the `errno` variable to `ENXIO` if no mixer at all is present (it is always possible to open `/dev/mixer0`, even when no mixer is available). The meaning of the bits of the mask are defined in later sections. Testing the bit corresponding to a mixer channel can be done using the expression "`mask & (1 << channel_no)`". The `channel_no` parameter may be one of the `SOUND_MIXER_` macros defined earlier or an integer value between 0 and `SOUND_MIXER_NRDEVICES`. The latter alternative is useful for writing a mixer that dynamically adapts to the capabilities of any card.

Checking Available Mixer Channels

The `ioctl SOUND_MIXER_READ_DEVMASK` returns a bit mask in the variable pointed to by the argument (`mask` in the previous example). To see if a particular mixer channel is supported you need to test if the bit corresponding to the channel number is set. Any attempt to access undefined mixer channels using channel specific `ioctl` calls will return an error (`errno` will be set to `EINVAL`).

Checking Available Recording Devices

The `ioctl` `SOUND_MIXER_READ_RECMASK` returns a bit mask where each bit represents a mixer channel. The channels having their corresponding bit set may be used as a recording channel.

Checking if a Device is Mono or Stereo

Most mixer devices have stereo capability, making it possible to independently set the volumes for both the left and right stereo channels of the mixer channel. However, some devices are mono only and in this case just the left channel volume is used. The `ioctl` call `SOUND_MIXER_READ_STEREODEVS` returns a bit mask where a 1 indicates that the corresponding channel supports stereo. A mixer program should use this information to decide if it should draw sliders for both stereo channels or not. Otherwise, having a stereo control displayed for a mono channel may confuse the user of the application.

Checking General Capabilities of a Mixer

The `ioctl` call `SOUND_MIXER_READ_CAPS` returns a bit mask which describes general capabilities of the mixer. These capabilities are not related to any particular mixer channel. Currently just one mixer capability is defined. The bit `SOUND_CAP_EXCL_INPUT` is set to 1 if only one mixer channel can be selected as a recording source at any one time. If the bit is 0 then it is possible to have several recording devices selected at the same time. In practice, checking this bit is not crucial since the `ioctl` call used for selecting the recording channel handles the two different modes of operation.

Naming of Mixer Channels

The file `<soundcard.h>` defines two sets of printable names for the mixer channels. These names should be used when labelling or naming the mixer channels in application programs. The macro SOUND_DEVICE_LABELS contains a list of printable strings which can be used, for example, to label the sliders for the channels. You could access the names by defining a variable as:

```
const char *labels[] = SOUND_DEVICE_LABELS;
```

For example, `labels[SOUND_MIXER_VOLUME]` contains a textual label ("Vol") for the main volume channel.

The macro `SOUND_DEVICE_NAMES` is similar but it contains names to be used for features such as parsing command options. The names in this macro don't contain blanks or upper case letters.

Meaning of Volume Levels

The OSS driver specifies volume levels using integer values from 0 to 100. The value 0 means

minimum volume (off) and 100 means maximum volume.

Most mixers have anywhere from 3 to 8 bits of accuracy for controlling the volume at the hardware level. The OSS driver scales between the local (0-100) and hardware defined volume. Since this scaling is not exact, the application should be careful when using the volume returned by the `ioctl` calls. If the application writes the volume and then reads it back, the returned volume is usually slightly different (smaller) than the requested one. If the write-read sequence is repeated several times, the volume level slides to zero even if the application makes no changes itself. It is recommended, therefore, that the application reads the volume just during initialization and ignores the volume returned later.

Getting and Setting Volumes

An application program can read and/or write the volume setting of a mixer device by calling the `ioctl` functions `SOUND_MIXER_READ` and `SOUND_MIXER_WRITE`. The mixer channel is given as an argument to the macro. The channel number may be one of the channel identifiers defined in `<soundcard.h>` or an integer between 0 and `SOUND_MIXER_NRDEVICES`. For example, the following call reads the current volume of the microphone input:

```
int vol;
if (ioctl(mixer_fd, SOUND_MIXER_READ(SOUND_MIXER_MIC), &vol) == -1) {
    /* An undefined mixer channel was accessed... */
}
```

The program should check if an error was returned from the `ioctl` call. The driver will return -1 and set `errno` if the mixer channel is not known or if there is no mixer at all.

The volumes for both stereo channels are returned in the same integer variable. The least significant byte gives volume for the left channel and the next 8 bits for the right channel. The upper 16 bits are undefined and should be ignored. For mono devices just the left channel value is valid (the right channel value is set to the left channel value by the driver).

The volume setting can be altered by using the `ioctl` `SOUND_MIXER_WRITE`. It works just like `SOUND_MIXER_READ`, but in addition it alters the actual hardware volume of the channel. Note that this call also returns the new volume in the variable passed as an argument to the `ioctl` call. In some cases the value may be slightly different from the value passed to the call.

NOTE

The `SOUND_MIXER_WRITE` `ioctl` returns the modified volume in the argument used in the call. A temporary variable should be used as the argument, otherwise the volume will slide down on each access.

Selecting the Recording Sources

The OSS driver has two calls for selecting recording sources. In addition, the `SOUND_MIXER_READ_RECMASK` returns the devices which can be used as recording devices.

The `ioctl SOUND_MIXER_READ_RECSRC` returns a bit mask having a bit set for each of the currently active recording sources. The default is currently the microphone input but the application should not assume this.

The `SOUND_MIXER_WRITE_RECSRC` ioctl can be used to alter the recording source selection. If no bits are on, the microphone input will be used.

Some cards, such as the SoundBlaster Pro, only allows one active input source at a time. The driver correctly handles requests for invalid recording source selections and returns a valid setting. A mixer program should always check the recording mask after changing it. It should also update the display if the returned mask is something other than the requested one.

Audio Programming

Introduction

Digital audio is the most common method used to represent sound inside a computer. In this method, sound is stored as a sequence of samples taken from an audio signal at constant time intervals. A sample represents the volume of the signal at the moment when it was measured. In uncompressed digital audio, each sample requires one or more bytes of storage. The number of bytes required depends on the number of channels (mono, stereo) and sample format (8 or 16 bits, mu-Law, etc.). The time interval between samples determines the sampling rate, usually expressed in samples per second or Hertz. Commonly used sampling rates range from 8 kHz (telephone quality) to 48 kHz (DAT tape).

The physical devices used in digital audio are known as an ADC (Analog to Digital Converter) and DAC (Digital to Analog Converter). A device containing both ADC and DAC is commonly known as a codec. The codec device used in SoundBlaster cards is often referred to as a DSP or Digital Signal Processor. Strictly speaking, this term is incorrect since true DSPs are powerful processor chips designed for signal processing applications rather than just a codec.

The sampling parameters affect the quality of the sound which can be reproduced from the recorded signal. The most fundamental parameter is the sampling rate which limits the highest frequency than can be stored. Nyquist's Sampling Theorem states that the highest frequency that can be reproduced from a sampled signal is at most half of the sampling frequency. For example, an 8 kHz sampling rate permits recording of signals in which the highest frequency is less than 4 kHz. Higher frequency signals must be filtered out before feeding them to a DAC.

The encoding format (or sample size) limits the dynamic range of the recorded signal (the difference between the faintest and the loudest signal that can be recorded). Theoretically the maximum dynamic range of a signal is 6 dB for each bit of sample size . This means, for example, that an 8-bit sample size gives dynamic range of 48 dB while 16-bit resolution can achieve 96 dB.

There is a tradeoff with sound quality. The number of bytes required to store an audio sequence depends on the sampling rate, number of channels, and sampling resolution. For example, storing one second of sound with one channel at an 8 kHz sample rate and 8-bit sample size take 8000 bytes of memory. This requires a data rate of 64 kbps, equivalent to one ISDN B channel. The same sound stored as 16-bit 48 kHz stereo samples takes 192 kilobytes. This is a 1.5 Mbps data rate, roughly equivalent to a T1 or ISDN primary rate interface.

Looking at it another way, at the higher data rate 1 megabyte of memory can store just 5.46 seconds of sound. With 8 kHz, 8-bit sampling the same megabyte of memory can hold 131 seconds of sound. It is possible to reduce memory and communication costs by compressing the recorded signal but this is beyond the scope of this document.

OSS provides three kinds of device files for audio programming. The only difference between the devices is the default sample encoding used after opening the device. The `/dev/dsp` device uses 8-bit unsigned encoding while `/dev/dspW` uses 16-bit signed little-endian (Intel) encoding and `/dev/audio` uses logarithmic mu-law encoding. There are no other differences between the devices. All of them work in 8 kHz mono mode after opening them. It is possible to change sample encoding by using `ioctl` calls, after which all of the device files behave in a similar way. However, it is recommended that the device file be selected based on the encoding to be used. This gives the user more flexibility in establishing symbolic links for these devices.

In short, it is possible to record from these devices using the normal `open`, `close`, `read` and `write` system calls. The default parameters of the device files (discussed above) have been selected so that it is possible to record and play back speech and other signals with relatively low quality requirements. It is possible to change many parameters of the devices by calling the `ioctl` functions defined later. All codec devices have the capability to record or playback audio. However, there are devices which don't have recording capability at all. Most audio devices have the capability of working in half duplex mode which means that they can record and play back but not at the same time. Devices having simultaneous recording and playback capability are called full duplex.

The simplest way to record audio data is to use standard UNIX commands such as `cat` and `dd`. For example "`cat /dev/dsp >xyz`" records data from the audio device to a disk file called `xyz` until the command is killed (e.g. with Ctrl-C). The command "`cat xyz >/dev/dsp`" can be used to play back the recorded sound file (note that you may need to change the recording source and level using a mixer program before recording to disk works properly).

Audio devices are always opened exclusively. If another program tries to open the device when it is already open, the driver returns immediately with an error (`EBUSY`).

General Programming Guidelines

It is highly recommended that you carefully read the following notes and also the Programming Guidelines chapter of the Introduction section. These notes are likely to prevent you from making the most common mistakes with the OSS API. At the very least you should read them if you have problems in getting your program to work.

This section lists a number of things that must be taken into account before starting programming digital audio. Many of the features referred to in these notes will be explained in more detail later in this document.

Avoid extra features and tricks. They don't necessarily make your program better but may make it incompatible with future devices or changes to OSS.

Open the device files using `O_RDONLY` or `O_WRONLY` flags whenever it is possible. The driver uses this information when making many optimizing decisions. Use `O_RDWR` only when writing a program which is going to both record and play back digital audio. Even in this case, try to find if

it is possible to close and reopen the device when switching between recording and playback.

Beware of byte order (endian convention) issues with encoding of 16-bit data. This is not a problem when using 8-bit data or normal 16-bit sound cards in little-endian (Intel) machines. However, byte order is likely to cause problems in big-endian machines (68k, PowerPC, SPARC, etc.). You should not blindly try to access 16-bit samples as `signed short`.

The default recording source and recording level are undefined when an audio device is opened. You should inform the user about this and instruct them to use a mixer program to change these settings. It is possible to include mixer features in a program which works with digital audio, but it is not recommended since it is likely to make your program more hardware dependent. Mixers operate differently and in fact may not be present at all.

Explicitly set all parameters your program depends on. There are default values for all parameters but it is possible that some future devices may not support them. For example, the default sampling speed (8 kHz) or sampling resolution (8-bit) may not be supported by some high end professional devices.

Always check if an error code (-1) is returned from a system call such as `ioctl`. This indicates that the driver was not able to execute the request made by your program.

In most cases `ioctl` modifies the value passed in as an argument. It is important to check this value since it indicates the value that was actually accepted by the device. For example, if the program requests a higher sampling rate than is supported by the device, the driver automatically uses the highest possible speed. The value actually used is returned as the new value of the argument. As well, the device may not support all possible sampling rates but in fact just a few of them. In this case the driver uses the supported sampling rate that is closest to the requested one.

Set sampling parameters always so that number of channels (mono/stereo) is set before selecting sampling rate (speed). Failing to do this will make your program incompatible with cards such as the SoundBlaster Pro which supports 44.1 kHz in mono but just 22.05 kHz in stereo. A program which selects 44.1 kHz speed and then sets the device to stereo mode will incorrectly believe that the device is still in 44.1 kHz mode when actually the speed is decreased to 22.05 kHz.

If examining an older program as an example, make sure that it follows these rules and that it actually works. Many old programs were made for early prototype versions of the driver and they are not compatible with later versions (2.0 or later).

Avoid writing programs which work only in 16-bit mode since some audio devices don't support anything other than 8-bit mode. It is relatively easy to write programs so that they are capable of output both in 8 and 16-bit modes. This makes the program usable for other than 16-bit sound card owners. At least you should check that the device supports 16-bit mode before trying to output 16-bit data to it. 16-bit data played in 8-bit mode (and vice versa) just produces a loud annoying noise.

Don't try to use full duplex audio before checking that the device actually supports full duplex mode.

Always read and write full samples. For example, in 16-bit stereo mode each sample is 4 bytes long (two 16-bit sub-samples). In this case the program must always read and write multiples of 4 bytes. Failing to do so will cause lost sync between the program and the device sooner or later. In this case the output and input will be just noise or the left and right channels will be reversed.

Avoid writing programs which keep audio devices open when they are not required. This prevents other programs from using the device. Implement interactive programs so that the device is opened only when user activates recording and/or playback or when the program needs to validate sampling parameters (in this case it should handle `EBUSY` situations intelligently). However, the device can be kept open when it is necessary to prevent other programs from accessing the device.

Always check for and display error codes returned by calls to the driver. This can be done using `perror`, `strerror` or some other standard method which interprets the error code returned in `errno`. Failing to do this makes it very difficult for the end user to diagnose problems with your program.

Simple Audio Programming

For simplicity, recording and playback will be described separately. It is possible to write programs which record and play back audio simultaneously but the techniques for doing this are more complex and so will be covered in a later section.

Declarations for an Audio Program

All programs using the OSS API should include `<soundcard.h>` which is a C language header file containing the definitions for the API. The other header files to be included are `<ioctl.h>`, `<unistd.h>` and `<fcntl.h>`. Other mandatory declarations for an audio application are a file descriptor for the device file and a program buffer which is used to store the audio data during processing by the program. The following is an example of declarations for a simple audio program:

Listing 2 - Definitions for an Audio Program

```
/*
 * Standard includes
 */
#include <ioctl.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/soundcard.h>

/*
 * Mandatory variables.
 */
#define BUF_SIZE 4096
int audio_fd;
unsigned char audio_buffer[BUF_SIZE];
```

In the above, the `BUF_SIZE` macro is used to define size of the buffer allocated for audio data. It is possible to reduce the system call overhead by passing more data in each read and write call. However, shorter buffers give better results when recording. The effect of buffer size will be covered in detail in the section Improving Real Time Performance. Buffer sizes between 1024 and 4096 are good choices for normal use.

Selecting and Opening the Sound Device

An audio device must be opened before it can be used. As mentioned earlier, there are three possible device files which differ only in the default sample encoding format they use (`/dev/dsp` used 8-bit unsigned, `/dev/dspW` uses 16-bit signed little-endian and `/dev/audio` uses mu-law). It is important to open the right device if the program doesn't set the encoding format explicitly.

The device files mentioned above are actually just symbolic links to the actual device files. For example, `/dev/dsp` normally points to `/dev/dsp0`, which is the first audio device detected on the system. The user has the freedom to set the symbolic links to point to other devices if it produces better results. It is good practice to always uses the symbolic link name (e.g. `/dev/dsp`) and not the actual device name (e.g. `/dev/dsp0`). Programs should access the actual device files only if the device name is made easily configurable.

It is recommended that the device file is opened in read only (`O_RDONLY`) or write only (`O_WRONLY`) mode. Read write mode (`O_RDWR`) should be used only when it is necessary to record and play back at the same time (full duplex mode).

The following code fragment can be used to open the selected device defined as DEVICE_NAME). The value of open_mode should be `O_WRONLY`, `O_RDONLY` or `O_RDWR`. Other flags are undefined and must not be used with audio devices.

Listing 3 - Opening a Device File

```
if ((audio_fd = open(DEVICE_NAME, open_mode, 0)) == -1) {
   /* Open of device failed */
   perror(DEVICE_NAME);
   exit(1);
}
```

It is recommended that programs display the error message returned by `open` using standard methods such as `perror` or `strerror`. This information is likely to be very important to the user or support group trying to determine why the device cannot be opened. There is no need to handle the various error messages differently. Only `EBUSY` (Device busy) can be handled by the program by trying to open the device again after some time (although it is not guaranteed that the device will ever become available).

A Simple Recording Application

Writing an application which reads from an audio device is very easy when the recording speed is relatively low, the program doesn't perform time consuming computations, there are no strict real-time response requirements. Solutions to handle exceptions to this case will be presented later in this document. All the program needs to do is to read data from the device and to process or store it in some way. The following code fragment can be used to read data from the device:

Listing 4 - Sound Recording

```
int len;
if ((len = read(audio_fd, audio_buffer, count)) == -1) {
   perror("audio read");
   exit(1);
}
```

In the above example, variable `count` defines the number of bytes the program wants to read from the device. It must be less or equal to the size of `audio_buffer`. In addition, it must always be an integer multiple of the sample size. Using an integer power of 2 (i.e. 4, 8, 16, 32, etc.) is recommended as this works best with the buffering used internally by the driver.

The number of bytes recorded from the device can be used to measure time precisely. The audio data rate (bytes per second) depends on sampling speed, sample size and number of channels. For example, when using 8 kHz 16-bit stereo sampling the data rate is 8000 * 2 * 2 = 32000 bytes/second. This is actually the only way to know when to stop recording. It is important to notice that there is no end of file condition defined for audio devices.

An error returned by `read` usually means that there is a (most likely permanent) hardware error or that the program has tried to do something which is not possible. In general it is not possible to recover from errors by trying again, although closing and reopening the device may help in some cases.

Simple Playback Application

A simple playback program works exactly like a recording program. The only difference is that a playback program calls `write`.

Setting Sampling Parameters

There are three parameters which affect the sound quality (and therefore memory and bandwidth requirements) of sampled audio data. These are:

> sample format (sometimes called as number of bits),
> number of channels (mono or stereo), and
> sampling rate (speed).

NOTE

It is important to always set these parameters in the above order. Setting sampling rate before the number of channels doesn't work with all devices.

It is possible to change sampling parameters only between `open` and the first `read`, `write` or other `ioctl` call made to the device. The effect of changing sampling parameters when the device is active is undefined. The device must be reset using the `ioctl SNDCTL_DSP_RESET` before it can accept new sampling parameters.

Selecting Audio Format

Sample format is an important parameter which affects the quality of audio data. The OSS API supports several different sample formats but most devices support just a few of them. The `<soundcard.h>` header file defines the following sample format identifiers:

Table 3 - Sound Sample Formats

Name	Description
AFMT_QUERY	Not an audio format but an identifier used when querying the current audio format.
AFMT_MU_LAW	Logarithmic mu-law audio encoding.
AFMT_A_LAW	Logarithmic A-law audio encoding.
AFMT_IMA_ADPCM	A 4:1 compressed format where a 16-bit audio sequence is represented using the average of 4 bits per sample. There are several different ADPCM formats and this one is defined by the Interactive Multimedia Association (IMA). The Creative ADPCM format used by the SoundBlaster 16 is not compatible with this one.
AFMT_U8	The standard unsigned 8-bit audio encoding used in PC soundcards.
AFMT_S16_LE	The standard 16-bit signed little-endian (Intel) sample format used in PC soundcards.
AFMT_S16_BE	Big-endian (M68K, PowerPC, SPARC, etc.) variant of the 16-bit signed format.
AFMT_S16_NE	16-bit signed format in machine's native endian convention.
AFMT_S8	Signed 8-bit audio format.

AFMT_S32_LE	Signed little-endian 32-bit format. Used for 24-bit audio data where the data is stored in the 24 most significant bits and the least significant 8 bits are not used (should be set to 0).
AFMT_S32_BE	Signed big-endian 32-bit format. Used for 24-bit audio data where the data is stored in the 24 most significant bits and the least significant 8 bits are not used (should be set to 0).
AFMT_U16_LE	Unsigned little-endian 16-bit format.
AFMT_U16_BE	Unsigned big-endian 16-bit format.
AFMT_MPEG	MPEG MP2/MP3 audio format (currently not supported).

It is important to realize that for most devices just the 8-bit unsigned format (AFMT_U8) is supported at the hardware level (although there are high-end devices which support only 16-bit formats). Other commonly supported formats are AFMT_S16_LE and AFMT_MU_LAW. With many devices AFMT_MU_LAW is emulated using a software based (lookup table) translation between mu-law and 8-bit encoding. This causes poor quality when compared with straight 8 bits.

Applications should check that the sample format they require is supported by the device. Unsupported formats should be handled by converting data to another format (usually AFMT_U8). Alternatively, the program should abort if it cannot do the conversion. Trying to play data in an unsupported format is a fatal error. The result is usually just loud noise which may damage ears, headphones, speakers, amplifiers, concrete walls and other unprotected objects.

The above format identifiers have been selected so that AFMT_U8 is defined as 8 and AFMT_S16_LE is 16. This makes these identifiers compatible with older ioctl calls which were used to select the number of bits. This is valid just for these two formats so format identifiers should not be used as sample sizes in programs.

AFMT_S16_NE is a macro provided for convenience. It is defined to be AFMT_S16_LE or AFMT_S16_BE depending of endian convention of the processor where the program is being run.

The number of bits required to store a sample is:
      4 bits for the IMA ADPCM format,
      8 bits for 8-bit formats, mu-law and A-law,
      16 bits for the 16-bit formats, and
      undefined for the MPEG audio format.

The sample format can be set using the ioctl call SNDCTL_DSP_SETFMT. The following code fragment sets the audio format to AFMT_S16_LE (other formats are similar):

Listing 5 - Setting Sample Format

```
int format;
format = AFMT_S16_LE;
if (ioctl(audio_fd, SNDCTL_DSP_SETFMT, &format) == -1) {
    /* fatal error */
    perror("SNDCTL_DSP_SETFMT");
    exit(1);
}

if (format != AFMT_S16_LE) {

    /* The device doesn't support the requested audio format. The
       program should use another format (for example the one returned
       in "format") or alternatively it must display an error message
       and to abort. */
}
```

The `SNDCTL_DSP_SETFMT ioctl` call simply returns the currently used format if `AFMT_QUERY` is passed as the argument.

It is very important to check that the value returned in the argument after the `ioctl` call matches the requested format. If the device doesn't support this particular format, it rejects the call and returns another format which is supported by the hardware.

A program can check which formats are supported by the device by calling `ioctl SNDCTL_DSP_GETFMTS` as in the listing below:

Listing 6 - Checking for Supported Formats

```
int mask;

if (ioctl(audio_fd, SNDCTL_DSP_GETFMTS, &mask) == -1) {
    /* Handle fatal error ... */
}
if (mask & AFMT_MPEG) {
    /* The device supports MPEG format ... */
}
```

NOTE

`SNDCTL_DSP_GETFMTS` returns only the sample formats that are actually supported by the hardware. It is possible that the driver supports more formats using some kind of software conversion (signed to unsigned, big-endian to little-endian or 8-bits to 16-bits). These emulated formats are not reported by this `ioctl` but `SNDCTL_DSP_SETFMT` accepts them. The software conversions consume a significant amount of CPU time so they should be avoided. Use this feature only if it is not possible to modify the application to produce the supported data format directly.

`AFMT_MU_LAW` is a data format which is supported by all devices. OSS versions prior to 3.6 always reported this format in `SNDCTL_DSP_GETFMTS`. Version 3.6 and later report it only if the device supports mu-law format in hardware. This encoding is meant to be used only with applications and

audio files ported from systems using mu-law encoding (such as SunOS).

Selecting the Number of Channels (Mono/Stereo)

Most modern audio devices support stereo mode. The default mode is mono. An application can select the number of channels calling `ioctl` SNDCTL_DSP_CHANNELS with an argument specifying the number of channels (see listing 7). Some devices support up to 16 channels.

Listing 7 : Setting Number of Channels

```
int channels = 2; /* 1=mono, 2=stereo */
if (ioctl(audio_fd, SNDCTL_DSP_CHANNELS, &channels) == -1) {
   /* Fatal error */
    perror("SNDCTL_DSP_CHANNELS");
    exit(1);
}

if (channels != 2)
{
 /* The device doesn't support stereo mode ... */
}
```

NOTE

Applications must select the number of channels and number of bits before selecting sampling speed. There are devices which have different maximum speeds for mono and stereo modes. The program will behave incorrectly if the number of channels is changed after setting the card to high speed mode. The speed must be selected before the first `read` or `write` call to the device.

An application should check the value returned in the variable pointed by the argument. Many older SoundBlaster 1 and 2 compatible devices don't support stereo. As well, there are high end devices which support only stereo modes.

Selecting Sampling Rate (speed)

Sampling rate is the parameter that determines much of the quality of an audio sample. The OSS API permits selecting any frequency between 1 Hz and 2 GHz. However in practice there are limits set by the audio device being used. The minimum frequency is usually 5 kHz while the maximum frequency varies widely. Some of the oldest sound cards supported at most 22.05 kHz (playback) or 11.025 kHz (recording). The next generation supported 44.1 kHz (mono) or 22.05 kHz (stereo). With modern sound devices the limit is 96 kHz (DVD quality) but there are still few popular cards that support just 44.1 kHz (audio CD quality).

The default sampling rate is 8 kHz. However an application should not depend on the default since there are devices that support only higher sampling rates. The default rate could be as high as 48 kHz with such devices.

Codec devices usually generate the sampling clock by dividing the frequency of a high speed crystal oscillator. In this way it is not possible to generate all possible frequencies in the valid range. For this reason the driver always computes the valid frequency which is closest to the requested one and returns it to the calling program. The application should check the returned frequency and to compare it with the requested one. Differences of few percents should be ignored since they are usually not audible.

With some professional devices the sampling rate may be locked to some external source (S/PDIF, AES/EBU, ADAT, or world clock). In this case the SNDCTL_DSP_SPEED ioctl cannot change the sampling rate, instead the locked rate will be returned. This type of exceptional condition will be explained in the README file for the particular low-level sound driver.

The following code fragment can be used to select the sampling speed:

Listing 8 - Setting Sampling Rate

```
int speed = 11025;
if (ioctl(audio_fd, SNDCTL_DSP_SPEED, &speed)==-1) {
    /* Fatal error */
    perror("SNDCTL_DSP_SPEED");
    exit(Error code);
}
if ( /* returned speed differs significantly from the requested one... */ ) {
    /* The device doesn't support the requested speed... */
}
```

NOTE

Applications must select the number of channels and number of bits before selecting speed. There are devices which have different maximum speeds for mono and stereo modes. The program will behave incorrectly if number of channels is changed after setting the card to high speed mode. Speed must be selected before the first `read` or `write` call to the device.

Other Commonly Used `ioctl` Calls

It is possible to implement most audio processing programs without using any `ioctl` calls other than the three described earlier. This is possible if the application just opens the device, sets parameters, calls `read` or `write` continuously (without noticeable delays or pauses) and finally closes the device. This kind of application can be described as stream or batch application.

There are three additional calls which may be required with slightly more complicated programs. All of them do not require or return an argument (just use an argument of 0).

The `ioctl` SNDCTL_DSP_SYNC can be used when an application wants to wait until the last byte written to the device has been played (it doesn't wait in recording mode). When that occurs, the call resets (stops) the device and returns back to the calling program. Note that this call may take several seconds to execute depending on the amount of data in the buffers. Closing any sound device calls

SNDCTL_DSP_SYNC implicitly.

The ioctl SNDCTL_DSP_RESET stops the device immediately and returns it to a state where it can accept new parameters. It should not be called after opening the device as it may cause unwanted side effects in this situation. The call is only required when recording ir playback needs to be aborted. In general, opening and closing the device is recommended instead of using SNDCTL_DSP_RESET.

The ioctl SNDCTL_DSP_POST is a more lightweight version of SNDCTL_DSP_SYNC. It just tells to the driver that there is likely to be a pause in the output. This makes it possible for the device to handle the pause more intelligently.

NOTE

All of these ioctl calls are likely to cause clicks or unnecessary pauses in the output. You should use them only when they are required.

There are few places where these calls should be used. You should call SNDCTL_DSP_POST when your program is going to pause continuous output of audio data for relatively long time. This kind of situation is, for example, the following:

> after playing a sound effect when a new one is not started immediately (another way is to output silence until next effect starts);

> before the application starts waiting for user input;

> before starting lengthy operation such as loading a large file to memory.

The functions SNDCTL_DSP_RESET or SNDCTL_DSP_SYNC should be called when the application wants to change sampling parameters (speed, number of channels or number of bits).

The application must call SNDCTL_DSP_SYNC or SNDCTL_DSP_RESET before switching between recording and playback modes (or alternatively it should close and reopen the audio device).

You can use SNDCTL_DSP_RESET when playback should be stopped (cancelled) immediately.

Call SNDCTL_DSP_RESET after recording (after the last read from the device) if you are not going to close the device immediately. This stops the device and prevents the driver from displaying an unnecessary error message about recording overrun.

Call SNDCTL_DSP_SYNC when you want to wait until all data has been played.

Interpreting Audio Data

Encoding of audio data depends on the sample format. There are several possible formats, the most common of which are described here.

Mu-law (Logarithmic Encoding)

This is a format that originated from digital telephone technology. Each sample is represented as an 8-bit value which is compressed from the original 16-bit value. Due to logarithmic encoding, the value must be converted to linear format before it is used in computations (two mu-law encoded values cannot simply be added). The actual conversion procedure is beyond the scope of this text. Avoid mu-law if possible and use the 8 or 16-bit linear formats instead.

8-bit Unsigned

This is the normal PC sound card (SoundBlaster) format which is supported by practically all sound hardware. Each sample is stored in an 8-bit byte. The value of 0 represents the minimum level and 255 the maximum. The neutral level is 128 (0x80 in hexadecimal). In practice there is some noise in the silent portions of recorded files, so the byte values may vary between 127 (0x7f) and 129 (0x81).

The C data type to be used is `unsigned char`. To convert from unsigned to signed 8-bit formats, subtract 128 from the value to be converted. Exclusive ORing value with 0x80 does the same (in C use the expression "`value ^= 0x80`").

16-bit Signed

<div style="border:1px solid black">

CAUTION

Care must be taken when working with 16-bit formats. 16-bit data is not portable and depends on the design of both the CPU and audio device. The situation is simple when using a little-endian x86 CPU with a normal soundcard. In this case both the CPU and the soundcard use the same encoding for 16-bit data. However, the same is not true when using 16-bit encoding in a big-endian environment such as SPARC, PowerPC or HP-PA.

</div>

The 16-bit encoding normally used by sound hardware is little-endian (`AFMT_S16_LE`). However there are machines with built-in audio chips which support only big-endian encoding.

When using signed 16-bit data, the C data type best matching this encoding is usually `signed short`. However, this is true only in little-endian machines. In addition, the C standard doesn't define the sizes of particular data types so there is no guarantee that short is 16 bits long in all machines. For this reason, using an array of `signed short` as an audio buffer should be considered a programming error although it is commonly done in audio applications.

The proper way is to use an array of `unsigned char` and to manually assemble/disassemble the

buffer to be passed to the driver. For example:

Listing 9 - Handling 16-bit Data

```
unsigned char devbuf[4096];
int applicbuf[2048];
int i, p=0;

/* Place 2048 16-bit samples into applicbuf[] here */
for (i=0; i<2048; i+=2) {
    /* first send the low byte then the high byte */
    devbuf[p++] = (unsigned char)(applicbuf[i] & 0xff);
    devbuf[p++] = (unsigned char)((applicbuf[i] >> 8) & 0xff);
}
/* Write the data to the device ... */
```

Disassembling the data after input from the file can be performed in similar way (this is left as an exercise for the reader).

The AFMT_S16_NE format can be used when a program wants to encode or decode 16-bit samples locally. It automatically selects the right format for the CPU architecture being compiled for. In this way it's usually possible to simply use signed short format to store the samples.

The AFMT_S32_LE, AFMT_S32_BE, and AFMT_S32_NE formats are a 32-bit signed format which can be used to store audio data of arbitrary precision. Data smaller than 32 bits is stored left justified so that the unused bits are set to all zeroes. For example, 24-but data is store such that the 24 most significant bits are used and the 8 least significant are left as zeroes.

Encoding of Stereo Data

When using stereo data, there are two samples for each time slot. The left channel data is always stored before the right channel data. The samples for both channels are encoded as described previously.

This is extended when more channels are used. For example, with 4 channels the sample values for each channel are send in turn.

Conclusions

The preceding should be all you need to know when implementing basic audio applications. There are many other `ioctl` calls but they are usually not required. However, there are real-time audio applications such as games, voice conferencing systems, sound analysis tools, effect processors and many others. In these applications more advanced techniques are required. These are covered in the later section Advanced Programming Topics. Make sure that you understand all of the basic concepts before proceeding to the advanced section.

MIDI Programming

This section starts with a general introduction to MIDI, then covers programming using both the low-level API and the raw music interface.

What is MIDI?

The acronym MIDI stands for Musical Instrument Digital Interface. *The MIDI 1.0 Detailed Specification* defines both the hardware level interface and the communication protocol used for communication between devices using a MIDI interface. It is primarily a data communication specification but is also used in other ways. This section gives a very cursory overview of MIDI.

The hardware level MIDI interface is an asynchronous serial byte-oriented protocol similar to (but not compatible with) the RS-232 standard. The data transfer rate is 31250 bits per second and devices are connected using MIDI cables which use a 5-pin DIN connector on each end. One cable can carry data on just one direction; a bi-directional connection requires two cables. More than two devices can be connected together by daisy-chaining the devices.

MIDI devices communicate by sending messages through the MIDI cable. Every message starts with a status byte and may have one or more additional data bytes. The status byte has 1 in the most significant bit while the data bytes have 0. This means that the data bytes can take just 128 different values and carry just 7 bits of information.

The upper four bits of a status byte specify the type of status message and the last 4 bits carry the MIDI channel number. Status bytes 0xF0 to 0xFF are reserved for system messages.

There are 16 possible channels in the MIDI cable. Each of them can be assigned to physically separate devices or devices may interpret the messages sent to all channels. Some parameters, such as instrument (program) number, are assigned by channel so each device listening to a particular MIDI channel will play using the same instrument number. The device has the freedom to interpret the instrument number as it wishes.

For example, when a performer presses a key on a music keyboard, a NOTE ON message is transmitted on the MIDI cable. It starts with the status byte 0x9*X* (where the *X* is the channel number). There are two data bytes following the status. The first is the note number, indicating which key the performer pressed. The second specifies the velocity of the key press. The velocity is used to control the volume and some other parameters of the played sound.

It is important to note that no sound is transferred through the MIDI interface, just instructions for how the receiving instrument should be controlled.

A MIDI file contains MIDI messages and other data which can be used by MIDI sequencers and other applications. It is a well-defined interchange format which makes it possible to transfer songs

between virtually any application supporting the format. These files commonly have the extension `mid`.

Unlike some other file formats for storing musical information (e.g. mod), MIDI files don't contain any instrument data. The instruments are defined by including some MIDI program change messages into the files. The playing system has complete freedom to assign the actual instrument timbres for the program numbers.

Low level MIDI Programming

Introduction

There is a separate device file for each installed MIDI interface. The device name contains two decimal digits which specify the interface number. The interface number is shown in the output of `/dev/sndstat`. For example, the device file for the first installed MIDI port is `/dev/midi00`. The name `/dev/midi` is a symbolic link that points to the default MIDI device file, which is usually `/dev/midi00`.

These device files have capabilities similar to the ordinary `/dev/tty` interface. Everything written to the device will be sent to the MIDI port as soon as possible (not necessarily immediately, there could be some earlier written bytes in the queue which delay the transmit). There are no timing features, which makes it difficult to use these devices for sequencer type applications. The intended use of this interface is sending and receiving system exclusive messages. This is required, for example, when making patch editors and librarians for various MIDI synthesizers.

Reading from the device waits until there is at least one byte in the receive buffer. When the first byte is received, the driver will not wait for additional characters. This means that the read usually returns fewer bytes than requested. Since the MIDI transfer rate is fairly high (about 31 Kbaud), several bytes will be received before the reading process finally gets activated and is able to continue execution of the `read` call. On a 50 MHz 486 system, for example, it can receive up to about 60 bytes at a time. On a slower or more heavily loaded system the `read` could return even more data at once.

There are a couple of unnecessary delays in the current implementation, but they seem to be harmless. For example, you can route the incoming MIDI data from one port into another using "`cat /dev/midi00 >/dev/midi01`". There is no noticeable delay between a key press on the keyboard and the sound on the synthesizer connected to `/dev/midi`01.

The `/dev/midi` interface supports the `select` system call, but currently only on the Linux platform.

To use the raw MIDI devices, you will need some knowledge of the MIDI protocol. The official MIDI 1.0 specification is sold by the MIDI Manufacturer's Association (MMA). There are some

books containing the most important parts of the protocol. In addition, various unofficial MIDI specifications are available on the Internet, one of which is **ftp://mitpress.mit.edu/pub/Computer-Music-Journal/Documents/MIDI**.

Changing Parameters

The `SNDCTL_MIDI_PRETIME ioctl` function sets the timeout for which to wait for the first MIDI message byte to be received. It accepts an integer parameter which specifies the time to wait in 100 ms steps. By default the driver waits indefinitely for data.

Raw Music Interface

Background

The raw music interface provides low-level access to the FM synthesizer and the MIDI devices on the soundcard. This interface is provided only in the commercial Open Sound System version (not OSS/Free). The API definitions can be found in the file <dm.h>.

/dev/dmfm0

By providing register level access to the FM synthesizer chip, developers can use the FM synthesizer in applications that are not music or sequencer oriented. Examples are signal generation and acoustic research. By obtaining direct access to the FM chip, you can generate custom FM sounds that cannot be played back via the sequencer. You can control individual parameters and export the resulting sound as a sequencer patch file.

/dev/dmmidi0

On the MIDI side, the raw music API provides simple read and write access to the MIDI port. This enables applications to provide their own MIDI sequencing. Examples of such applications are MIDI lighting controllers and extended MIDI support. In contrast, the /dev/midi device in OSS provides a TTY like interface and does provide intelligent MIDI processing. The raw music MIDI device provides a direct interface to the MIDI device.

Applications That Use the Raw Music Interface

A number of sample applications that use the raw music interface can be downloaded for the OSS web site. They are provided as pre-compiled binaries (with Motif) for the Linux x86 platform.

xfmedit - this is an editor for controlling all the FM synthesizer's registers. This provides OPL-3 2-operator access.

xcmf - This is a player for playing Creative Music Format (cmf) music files, which are a derivative of the standard MIDI except that the patches are contained in the file. This application provides a GUI for selecting the CMF file and tape-recorder like controls.

xmidi - This is a MIDI player which plays back MIDI files in the 4-operator mode of the FM synthesizer. This application allow you to change the MIDI instrument on the fly.

xsynth - This an application which provides a keyboard interface. You can either use the mouse to click on the keys or use the computer keyboard to make the sound. It is a virtual synthesizer which provides access to MIDI channels. It also provides a drum pattern generator. Additionally, it also

allows you to record and play back MIDI files via an FM or wave-table synthesizer.

xmuseq - This is a piano-roll style MIDI editor. It provides complete MIDI editing capabilities such as is found on Windows based MIDI editors such as Windjammer. This is an evolving product. There are plans to provide music notation support.

FM Synthesizer Interface

Introduction

The FM synthesizer uses FM modulation to perform tone generation. The FM synthesizer supports OPL-3 mode (2 or 4 operator stereo mode) or the older OPL-2 mode (AdLib mono mode). FM synthesis interface to the device driver is provided through `ioctls`. The following sections deal with the basic FM synthesis programming techniques.

FM Synthesizer Specifications:

> Yamaha YMF 262 OPL-3 Chip
> AdLib Compatible OPL-2 mode
> OPL-2 mode (AdLib) supports 9 channels of 2-operator FM tones or 6 channels of 2 operator FM tones and 5 percussion instruments
> OPL-3 mode supports 18 channels of 2-operator Stereo FM tones or 6 channels of 4-operator and 6 channels of 2-operator FM tones and 5 percussion instrument channels

FM synthesis uses a modulator cell and a carrier cell. The modulator cell modulates the carrier cell. The FM synthesizer provides 36 cells comprising of 18 modulator cells and 18 carrier cells that result in 18 simultaneous channels being generated. In essence, the 18 channels can generate any 2 operator note.

There are basically 2 modes supported by the FM synthesizer: OPL-2 and OPL-3. OPL-2 mode consists of nine 2-operator note channels with mono output. OPL-3 mode consists of eighteen 2-operator note channels with stereo output or six 4-operator and six 2-operator channels. Both OPL-2 and OPL-3 modes support 5 percussion instruments and when the rhythm mode is selected, the percussion instruments occupy 3 channels (1 channel for bass drum, ½ channel for the other 4 percussion instruments).

Data Structures

The FM synthesizer uses three structures for FM tone generation. The data structure `dm_fm_voice` sets the voice parameters for the FM tone. The parameters do not change for a given type of voice. The second data structure used is `dm_fm_note`. This data structure sets the frequency and octave and sounds the tone when activated for a particular channel. The final data structure is `dm_fm_param`. This data structure controls the rhythm section as well as global parameters for the FM synthesizer.

FM Voice Data Structure

```
struct dm_fm_voice
{
  unsigned char op;          /* 0 for modulator and 1 for carrier */
  unsigned char voice;       /* Channels of 2-op notes */
  unsigned char am;          /* Tremolo or AM modulation effect flag - 1 bit */
  unsigned char vibrato;     /* Vibrato effect flag - 1 bit */
  unsigned char do_sustain;  /* Sustaining sound phase flag -1 1 bit */
  unsigned char kbd_scale;   /* keyboard scaling flag - 1 bit */
  unsigned char harmonic;    /* Harmonic or frequency multiplier - 4 bits */
  unsigned char scale_level; /* Decreasing volume of higher notes - 2 bits */
  unsigned char volume;      /* Volume of output - 6 bits */
  unsigned char attack;      /* Attack phase level of the note - 4 bits */
  unsigned char decay;       /* Decay phase level of the note - 4 bits */
  unsigned char sustain;     /* Sustain phase level of the note - 4 bits */
  unsigned char release;     /* Release phase level of the note - 4 bits */
  unsigned char feedback;    /* Feedback from op 1 or op 2 - 3 bits */
  unsigned char connection;  /* Serial or parallel operator connection-1 bit */
  unsigned char left;        /* Left channel audio output */
  unsigned char right;       /* Right channel audio output */
  unsigned char waveform;    /* Waveform select - 3 bits */
};
```

The fields of the structure are used as follows:

op - holds the type modulator or carrier operator. A value of 0 denotes a modulator cell and 1 denotes a carrier cell.

voice - holds the voice or the channel number. There are 36 op cells that result in 18 channels that can produce a note simultaneously. A value of 0 through 17 specify a channel. In rhythm mode, channels 6, 7 and 8 cannot be used to generate melodic notes.

am - a flag (1-bit) that turns the AM modulation (tremolo) effect on or off. The rate for AM modulation is 3.7 Hz.

vibrato - a flag (1-bit) that turn the vibrato effect on or off. The rate is 6.4 Hz.

do_sustain - a flag (1-bit) that turns the sustained sound on or off. If do_sustain is 1 then the sustaining sound is sound when the note is played. If do_sustain is 0 then the diminishing sound is selected when the note on (see the figure).

Figure 1 - do_sustain bit
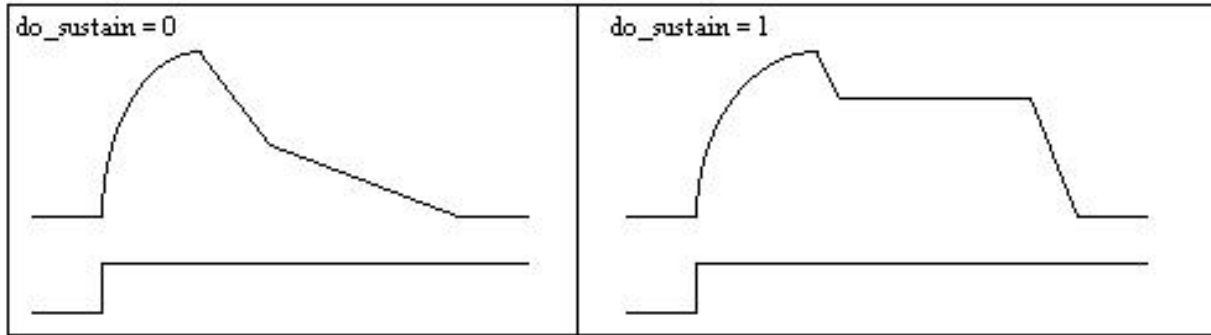
harmonic - a 3 bit field (values 0-7) which represents the harmonic or the multiplication factor that needs to be applied to the frequency(fnum). The following table specifies the multiplication factor with respect to harmonic number.

Table 4 - harmonic values

Harmonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Multiplier	0.5	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

kbd_scale - a flag (1-bit) that turns on the keyboard scale rate. If kbd_scale is 1 then the attack/decay rates become faster as the pitch (fnum + octave) increases.

Table 5 - kbd_scale values

Key Scale	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
KSR = 0	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
KSR = 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

scale_level - is a 2 bit field (values 0-3) which produces a gradual decrease in note output level towards higher pitches (octave+fnum). The following table shows the scale level and the corresponding attenuation.

Table 6 - scale_level values

KSL	0	1	2	3
Attenuation	0 db/Octave	3 dB/Octave	1.5 dB/Octave	6 dB/Octave

volume - a 6 bit field (values 0-63) which represents the total output volume of the op. Maximum attenuation is 47.25 dB. Attenuating the output from a modulator cell will change the frequency spectrum produced by the carrier cell.

attack - a 4-bit field (values 0-15) that sets the attack rate or the rising time of the sound.

decay - a 4-bit field (values 0-15) that sets the decay rate or the diminishing time after the attack.

sustain - a 4-bit field (values 0-15) that sets the sustain level. For continuing sounds the sustain level gives the point of change where the attenuated sounds in the decay mode changes to a sound having a constant level. For diminishing sounds, the sustain level gives the point where the decay mode changes to a release mode.

release - a 4-bit field (values 0-15) that sets the release level. For continuing sounds, the release level defines the rate at which sound disappears after key_off. For diminishing sounds, the release level indicates the attenuation after the sustain level is reached.

feedback - a 3 bit field (values 0-7) which determines the modulation factor for self-feedback. This is applicable only to the modulator cell.

Table 7 - feedback bits

Feedback	0	1	2	3	4	5	6	7
Modulation	0	/16	/8	/4	/2		2	4

connection - a flag (1-bit) that describes the connection of the modulator and carrier. When connection is 0 the carrier is chained to the modulator in a serial connection and produces true FM tones. If connection is 1, the carrier and modulator are connected in parallel to produce two simultaneous tones.

waveform - a 3 bit field (values 0-7) which specifies the shape of the waveform.

left - a flag (1 bit) field which enables (set to 1) the left output channel or disables (set to 0) the left output channel.

right - a flag (1 bit) field which enables (set to 1) the right output channel or disables (set to 0) the right output channel.

FM Note Data Structure

The next data structure consists of parameters such as octave, channel and frequency. These parameters vary compared to the dm_fm_voice characteristics.

```
struct dm_fm_note
{
   unsigned char voice;          /* 18 channels of 2-op notes */
   unsigned char octave;         /* Octave number of the note - 3 bits */
   unsigned int fnum;            /* Frequency of the note - 10 bits */
   unsigned char key_on;         /* Output sound flag - 1 bit */
};
```

voice - holds the voice or the channel number. There are 36 op cells resulting in 18 channels that can produce a note simultaneously. A value of 0 through 17 specify a channel. In rhythm mode,

channels 6, 7, and 8 cannot be used to generate melodic notes.

`octave` - a 3 bit value (values 0-7) which represents the octave number of the note.

`fnum` - a 10 bit value (values 0-1023) which represents the frequency of the note.

`key_on` - a flag (1-bit) that voices the notes (i.e. sound is produced). When key_on is 1, sound is produced. If key_on is 0, no sound is produced. In order to sound a note the key_on should make a 0 to 1 transition. Hence, you need to set it to 0 and then set it to 1 in the FM synthesizer.

FM Parameter Data Structure

The following is a description of the data structure used for the rhythm section and global FM parameters.

```
struct dm_fm_param
{
    unsigned char am_depth;    /* AM modulation depth for AM modulation effect */
    unsigned char vib_depth;   /* Vibrato depth for Vibrato effect */
    unsigned char kbd_split;   /* split keyboard for kbd_scaling */
    unsigned char rhythm;      /* turn on rhythm mode */
    unsigned char bass;        /* bass-occupies channel 7(modulator & carrier) */
    unsigned char snare;       /* snare - occupies modulator of channel 8 */
    unsigned char tomtom;      /* tom-tom - occupies modulator of channel 9 */
    unsigned char cymbal;      /* cymbal - occupies carrier of channel 9 */
    unsigned char hihat;       /* hihat - occupies carrier of channel 8 */
};
```

`am_depth` - a flag (1 bit) field which determines the amplitude modulation (tremolo) depth. The attenuation is 4.8 dB when am_depth=1 and 1 dB when am_depth=0.

`vib_depth` - a flag (1 bit) field which determines the vibrato depth of the op cell. The attenuation factor is 14% when vib_depth=1 and 7% when vib_depth=0.

`kbd_split` - a flag (1 bit) field which determines the split method to select the key scale number. This field is used to select the kbd_scale value. Depending on the pitch of the note (octave plus fnum), a key scale number between 0 and 15 is generated. This key scale number is applied to the attack/decay/sustain/release rates depending whether kbd_scale is 1 or 0. If kbd_split=1 then the key scale number depends on the most significant bit of the frequency. If kbd_split=0 then the key_scale number depends on the 2nd MSB of the frequency.

`rhythm` - a flag (1 bit) field. When rhythm=1, the channels 6,7 and 8 are used to generate percussion instruments such as bass, snare, tomtom, hihat and cymbals. Hence, regular FM notes or operators cannot be played on these channels.

`bass` - a flag (1 bit) field which turns on or off the bass drum percussion instrument. The bass drum requires a modulator and a carrier cell that occupy channel 6. Both operators require note settings

46

(Attack/Decay/Sustain/Release/Octave/Fnum etc) but the key_on field should be set to 0. Only when bass=1 is the bass drum sound produced.

`snare` - a flag (1 bit) field which turns on or off the snare drum percussion instrument. The snare drum requires a modulator cell occupying channel 7. The modulator cell needs to be set with the note settings that simulate a snare drum but the key_on field must be 0. Snare drum sound is produced when snare=1.

`tomtom` - a flag (1 bit) field which turns on of off the tomtom drums. The tomtom drum requires a carrier cell occupying channel 8. The carrier cell needs to be set with note settings that resemble the tomtom drum but the key_on field must be 0. Tomtom drum sound is produced when tomtom=1.

`cymbal` - a flag (1 bit) field which turns on or off the cymbal. The cymbal instrument requires a modulator cell occupying channel 8. The modulator cell must be set with note settings that resemble a cymbal. As with the previous percussion instrument, key_on should be 0. Only when cymbal=1, is the cymbal sound produced.

`hihat` - a flag (1 bit) field which turns on or off the hihat. The hihat instrument requires a carrier cell occupying channel 7. The carrier cell must be set with note settings that resemble a hihat. As with the previous percussion instrument, key_on should be 0. Only when hihat=1, is the hihat sound produced.

FM Synthesizer `ioctl` Functions

In this section we will examine the `ioctl`s that are used to generate FM tones. Before you can issue the `ioctl`s you must first obtain the file descriptor using an open call on the `/dev/sbpfm0` device. In the following examples we will use the file descriptor `fmfd` for the FM synthesizer device.

FM_IOCTL_RESET

This `ioctl` is used to reset the FM synthesizer. After opening the FM device /dev/dmfm0 it is advisable to issue a reset `ioctl`. This `ioctl` takes no parameters and is used as follows:

```
ioctl(fmdev, FM_IOCTL_RESET, NULL);
```

FM_IOCTL_SET_MODE

This `ioctl` is used to set the mode of the FM synthesizer. The `ioctl` takes one parameters which should be set to the value `OPL2` or `OPL3`. `OPL2` sets the FM synthesizer in OPL-2 or AdLib compatible mode. In this mode only 9 channels of 2 op voices with mono output are permitted. In `OPL3` mode, there are 18 channels of 2 op with stereo output or 6 channels of 4 operators and 6 channels of 2 operators with stereo output. The command to set the FM synthesizer in OPL-3 mode is as follows:

```
ioctl(fmfd, FM_IOCTL_SET_MODE, OPL3);
```

FM_IOCTL_SET_VOICE

This `ioctl` sets the voice parameters for the modulator and carrier operators. It accepts one parameter type struct `dm_fm_voice`. The `ioctl` is used as follows:

struct dm_fm_voice voice;
`ioctl`(fmfd, FM_IOCTL_SET_VOICE, &voice);

FM_IOCTL_PLAY_NOTE

This `ioctl` is used to voice a particular FM channel which has been preset with the FM voice characteristics. The parameter is of type struct dm_fm_note. The `ioctl` is used as follows:

```
struct dm_fm_note note;
note.key_on = 0;
ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note);
note.key_on = 1;
ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note);
```

FM_IOCTL_SET_PARAMS

This `ioctl` is used to set global FM parameters as well as control the percussion instruments. The parameter is of type struct `dm_fm_params`. The `ioctl` is used as follows:

```
struct dm_fm_params param;
ioctl(fmfd, FM_IOCTL_SET_PARAMS, &param);
```

FM_IOCTL_SET_OPL

This `ioctl` is used to set the connection type for the 4 op mode. The parameter is a byte defining the connection. If you want the synthesizer in 2-op mode then the `conn_type` = 0x0. If you want the synthesizer in 4-op mode with six 4-op channels then `conn_type` = 0x3F.

```
char conn_type = 0x3f;
ioctl(fmfd, FM_IOCTL_SET_PARAMS, &conn_type);
```

Programming the FM Synthesizer

In this section we will write a simple program to play random notes on the FM synthesizer. The example will demonstrate the capabilities of the device.

Listing 10 - Example FM Synthesizer Program
```
#include <stdio.h>
#include <fcntl.h>
#include <math.h>
```

```
#include <sys/dm.h>

#define VOICES 18
#define RAND(bits) (random() & (1<<(bits)) -1)
main()
{
        int fmfd;
        struct dm_fm_voice modulator, carrier;
        struct dm_fm_note note;
        struct dm_fm_params param;
        int channel_num;

/* First we open the FM device using an open call */
        fmfd = open("/dev/dmfm0", O_WRONLY);
        if (fmfd < 0)
            perror("open");

/* Now we reset the FM synthesizer using the RESET ioctl */
        if (ioctl(fmfd, FM_IOCTL_RESET) == -1)
            perror("reset");

/* Now set the FM synthesizer in OPL3 mode */
        if (ioctl(fmfd, FM_IOCTL_SET_MODE, OPL3) == -1)
            perror("mode");

        while (1) {
/* set global parameters but do not turn on percussion section */
            param.am_depth = RAND(1);
            param.vib_depth = RAND(1);
            param.kbd_split = RAND(1);
            param.rhythm = 0;
            param.bass = 0;
            param.snare = 0;
            param.hihat = 0;
            param.cymbal = 0;
            param.tomtom = 0;
/* send the param structure to the FM synthesizer */
            ioctl(fmfd, FM_IOCTL_SET_PARAMS, &param);

/* Play the note on all channels at the same time */
            for (channel_num = 0; channel_num < VOICES; channel_num++)
            {
/*
 * Now fill in the modulator cell structure using randomly generated
 *  values and masking off the bits. Look at the definition
 * of RAND(bits)
 */
                    modulator.voice = channel_num;
                    modulator.op = 0;
                    modulator.am = RAND(1);
                    modulator.vibrato = RAND(1);
                    modulator.do_sustain = RAND(1);
                    modulator.kbd_scale = RAND(1);
                    modulator.connection = 0;
                    modulator.attack = RAND(4);
                    modulator.decay = RAND(4);
                    modulator.sustain = RAND(4);
                    modulator.release = RAND(4);
                    modulator.volume = RAND(6);
                    modulator.scale_level = RAND(2);
                    modulator.feedback = RAND(3);
                    modulator.waveform = RAND(3);
```

```
                    modulator.left = RAND(1);
                    modulator.right = RAND(1);
/* Send the modulator structure to the FM synth */
                    if (ioctl(fmfd, FM_IOCTL_SET_VOICE, &modulator) == -1)
                        perror("modulator");
/*
 * Now fill in the carrier cell structure using randomly generated
 *  values and masking off the bits. Look at the definition
 * of RAND(bits)
 */
                    carrier.voice = channel_num;
                    carrier.op = 1;
                    carrier.am = RAND(1);
                    carrier.vibrato = RAND(1);
                    carrier.do_sustain = RAND(1);
                    carrier.kbd_scale = RAND(1);
                    carrier.connection = 0;
                    carrier.attack = RAND(4);
                    carrier.decay = RAND(4);
                    carrier.sustain = RAND(4);
                    carrier.release = RAND(4);
                    carrier.harmonic = RAND(4);
                    carrier.volume = RAND(6);
                    carrier.scale_level = RAND(2);
                    carrier.feedback = RAND(3);
                    carrier.waveform = RAND(3);
                    carrier.left = RAND(1);
                    carrier.right = RAND(1);
/* Send the carrier structure to the FM synth */
                    if (ioctl(fmfd, FM_IOCTL_SET_VOICE, &carrier) == -1)
                        perror("carrier");
/*
 * Now fill in the note  structure with random octaves and frequencies.
 * Before sounding the FM tone turn the note off and then key_on the note.
 */
                    note.voice = channel_num;
                    note.octave = RAND(3);
                    note.fnum =  RAND(10);
                    note.key_on = 0;
                    if (ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note) == -1)
                        perror("note");
                    note.key_on = 1;
                    if (ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note) == -1)
                        perror("note");
/* sleep between notes */
                    usleep(100000);
                } /*for loop*/
            } /*while loop */
}
```

Additional Notes on FM Programming

FM synthesis requires many parameter fields to be set and sometimes it is simpler to use patches to simulate various instruments. The Sound Blaster Instrument (SBI) format provides a uniform approach to programming the FM synthesizer. The SBI format only handles sound characteristics. The program has to provide the frequency and octave values. Sounding of the FM tone occurs when the key_on bit is set on a particular voice channel. The following is a description of the SBI file format (Note: the names in parentheses denote the dm_fm_note structure parameter).

Table 8 - SBI File Format

OFFSET (hex)	Description
00-03	File ID - 4 Bit ASCII String "SBI" ending with 0x1A
04 - 23	Instrument Name - Null terminated ASCII string
24	Modulator Sound Characteristics Bit 7: AM Modulation (am) Bit 6: Vibrato (vibrato) Bit 5: Sustaining Sound (do_sustain) Bit 4: Envelop Scaling (kbd_scale) Bits 3-0: Frequency Multiplier (harmonic)
25	Carrier Sound Characteristics Bit 7: AM Modulation (am) Bit 6: Vibrato (vibrato) Bit 5: Sustaining Sound (do_sustain) Bit 4: Envelop Scaling (kbd_scale) Bits 3-0: Frequency Multiplier (harmonic)
26	Modulator Scaling/Output Level Bits 7-6: Level Scaling (scale_level) Bits 5-0: Output Level (volume)
27	Carrier Scaling/Output Level Bits 7-6: Level Scaling (scale_level) Bits 5-0: Output Level (volume)
28	Modulator Attack/Decay Bits 7-4: Attack Rate (attack) Bits 3-0: Decay Rate (decay)
29	Carrier Attack/Decay Bits 7-4: Attack Rate (attack) Bits 3-0: Decay Rate (decay)
2A	Modulator Sustain/Release Bits 7-4: Sustain Level (sustain) Bits 3-0: Release Rate (release)
2B	Carrier Sustain/Release Bits 7-4: Sustain Level (sustain) Bits 3-0: Release Rate (release)
2C	Modulator Wave Select Bits 7-2: All bits clear (0) Bits 1-0: Wave Select (waveform)
2D	Carrier Wave Select Bits 7-2: All bits clear (0) Bits 1-0: Wave Select (waveform)

2E	Feedback/Connection Bits 7-4: All bits clear (0) Bits 3-1: Modulator Feedback (feedback) Bit 0: Connection (connection)
2F-33	Reserved for future use

The above description requires you to declare the following structures:

```
struct dm_fm_voice modulator;
struct dm_fm_voice carrier;
struct dm_fm_note note;
```

Now store the corresponding values from the SBI file into the respective structures. From the above description, the programmer needs to provide are the following parameters:

```
modulator.voice=carrier.voice=x (where x is a channel number between 0 and 17)
modulator.left=carrier.left=x (where x is the left audio output flag 0 or 1)
modulator.right=carrier.right=x (where x is the right audio output flag 0 or 1)
modulator.op = 0      (modulator's op number is 0)
carrier.op = 1        (carrier's op is 1)
note.voice = x        (where x is a channel number from 0-17)
note.fnum =  x        (where x is a frequency number from 0-1023)
note.octave = x       (where x is an octave number from 0-7)
note.key_on = 1       (the note must be keyed off and then keyed on)
```

Programming The FM Synthesizer Using SBI Files

The following code explains the mechanism to read an SBI format file and play the note at frequency 800 in octave number 5.

Listing 11 - Sample Code to Read an SBI File

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include "/sys/dm.h"

#define FALSE 0
#define TRUE 1

struct dm_fm_voice op0, op1;  /* the voice struct to hold the SBI file */
struct dm_fm_note note;       /* the note struct to make the sound */
int fmfd, fd;                 /* fmfd - FM dev handle; fd - SBI file */
char instrument_buf[16];      /* buffer to hold the SBI data from file */

main (argc, argv)
int argc;
char **argv;
{
        fmfd = open("/dev/dmfm0", O_WRONLY);   /* open the FM device */
        ioctl(fmfd,FM_IOCTL_RESET);            /* reset the FM device */
        ioctl(fmfd, FM_IOCTL_SET_MODE, OPL3);  /* set mode to OPL3 */
        set_params();                          /* set global FM params */
        fd = open(argv[1], O_RDONLY);          /* open the SBI file */
```

```c
/* now verify that it is truly an SBI instrument file by reading the
 * header
 */
        if (!verify_sbi(fd)) {
                printf("file is not in SBI format\n");
                exit (0);
        }
        get_instrument(fd);                      /* fill the voice structs */
        play_instrument();                       /* play the sound */
}

/* check for "SBI" + 0x1A (\032) in first four bytes of file */
int verify_sbi(fd)
int fd;
{
    char idbuf[5];      /* get id */
    lseek(fd, 0, SEEK_SET);
    if (read(fd, idbuf, 4) != 4)
        return(FALSE);      /* compare to standard id */
    idbuf[4] = (char)0;
    if (strcmp(idbuf, "SBI\032") != 0)
        return(FALSE);     return(TRUE);
}


get_instrument(fd)
int fd;
{
    lseek(fd, 0x24, SEEK_SET);
    read(fd, instrument_buf, 16);

/* Modulator Characteristics */
        if (instrument_buf[0] & (1<<7))
                op0.vibrato = 1;
        else
                op0.vibrato = 0;
        if (instrument_buf[0] & (1<<6))
                op0.am = 1;
        else
                op0.am = 0;
        if (instrument_buf[0] & (1<<5))
                op0.kbd_scale = 1;
        else
                op0.kbd_scale = 0;
        if (instrument_buf[0] & (1<<4))
                op0.do_sustain = 1;
        else
          op0.do_sustain = 0;
        op0.harmonic = instrument_buf[0] & 0x0F;

/* Carrier Characteristics */
        if (instrument_buf[1] & (1<<7))
                op1.vibrato = 1;
        else
                op1.vibrato = 0;
        if (instrument_buf[1] & (1<<6))
                op1.am = 1;
        else
                op1.am = 0;
        if (instrument_buf[1] & (1<<5))
                op1.kbd_scale = 1;
        else
                op1.kbd_scale = 0;
```

```c
        if (instrument_buf[1] & (1<<4))
                op1.do_sustain = 1;
        else
  op1.do_sustain = 0;
        op1.harmonic = instrument_buf[1] & 0x0F;

/* Modulator Scale/Volume Level */
        op0.scale_level = instrument_buf[2] >>6;
        op0.volume = instrument_buf[2] & 0x3f;

/* Carrier Scale/Volume Level */
        op1.scale_level  = instrument_buf[3] >>6;
        op1.volume = instrument_buf[3] & 0x3f;

/* Modulator Attack/Decay */
        op0.attack = instrument_buf[4] >> 4;
        op0.decay = instrument_buf[4] & 0xF;

/* Carrier Attack/Decay */
        op1.attack = instrument_buf[5] >> 4;
        op1.decay = instrument_buf[5] & 0xF;

/* Modulator Sustain/Release */
        op0.sustain = instrument_buf[6] >> 4;
        op0.release = instrument_buf[6] & 0xF;

/* Carrier Sustain/Release */
        op1.sustain = instrument_buf[7] >> 4;
        op1.release = instrument_buf[7] & 0xF;

/* Modulator Waveform */
        op0.waveform = instrument_buf[8] & 0x03;

/* Carruer Waveform */
        op1.waveform = instrument_buf[9] & 0x03;

/* Modulator Feedback/Connection*/
        op0.connection = instrument_buf[0xA]  & 0x01;
        op1.connection = op0.connection;
        op0.feedback = (instrument_buf[0xA] >> 1) & 0x07;
        op1.feedback = op0.feedback;

/* byte 0xB - 20  Reserved */
}

play_instrument()
{
/*
 * Set the FM channel to channel 0. Fill in the rest of the fields and
 * Issue an ioctl to set the modulator parameters
 */
                op0.op = 0;
                op0.voice = 0;
                op0.left = 1;
                op0.right = 1;
                ioctl(fmfd, FM_IOCTL_SET_VOICE, &op0);
/*
 * Set the FM channel to channel 0. Fill in the rest of the fields and
 * Issue an ioctl to set the carrier parameters
 */
                op1.op = 1;
                op1.voice = 0;
```

54

```
                    op1.left = 1;
                    op1.right = 1;
                    op1.volume = 63;
                    ioctl(fmfd, FM_IOCTL_SET_VOICE, &op1);
/*
 * Fill in the note structure and first key_off the note and then key_on.
 */
                    note.voice = 0;                /* select channel 0 */
                    note.octave = 5;
                    note.fnum = 800;
                    note.key_on = 0;                        /*Key off*/
                    ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note);
                    note.key_on = 1;                        /*Key on*/
                    ioctl(fmfd, FM_IOCTL_PLAY_NOTE, &note);
}

set_params()
{
struct dm_fm_params p;
        p.am_depth = 0;
        p.vib_depth = 0;
        p.kbd_split = 0;
        p.rhythm = 0;
        p.bass = 0;
        p.snare = 0;
        p.tomtom = 0;
        p.cymbal = 0;
        p.hihat = 0;
        ioctl (fmfd, FM_IOCTL_SET_PARAMS, &p);
}
```

FM Synthesizer in 4-Operator Mode

In the 4-op mode the FM synthesizer uses 4 operators consisting of two 2-op channels. From 18 2-op channels, we can get six 4-op channels, with 5 channels for percussion (as described above) and three 2-op channels used for FM voices. The diagram below describes how 18 2-op channels are organized:

Figure 2 - 4-Operator Schematic for Modulator/Carriers
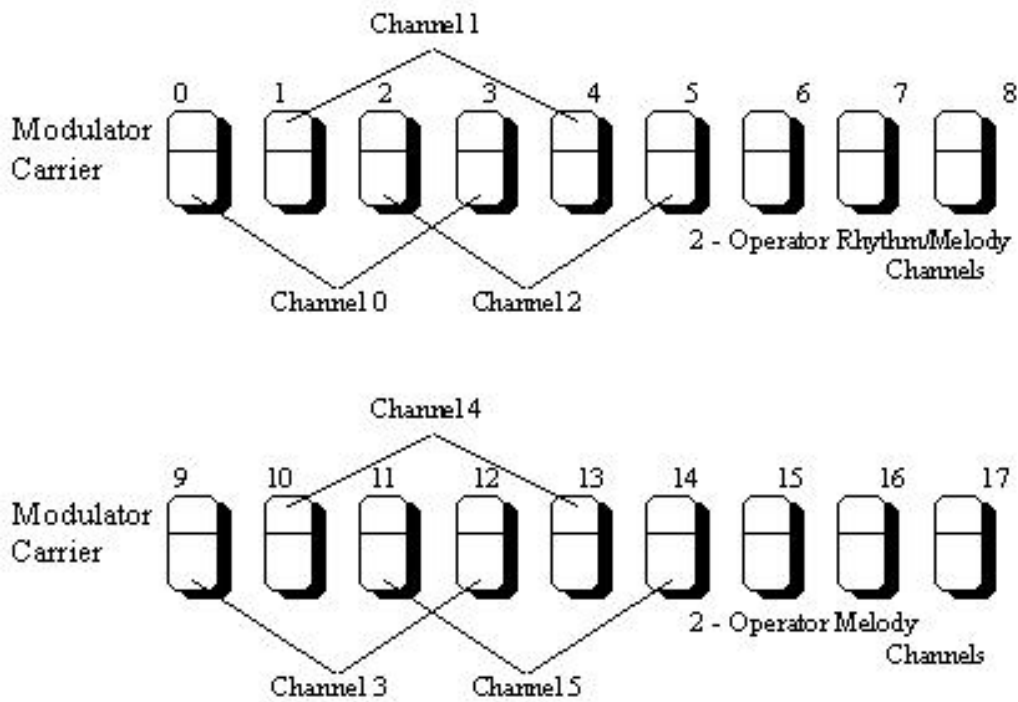
The ioctl FM_IOCTL_SET_OPL is used to set the 4-op connection mask. This ioctl requires a 6 bit mask. If the mask is 0 then all the FM voice channels default to 2-op mode thus yielding 18 channels or 15 voice plus 5 percussion channels. If the mask is set to 0x3F then the FM synthesizer is configured for six 4 op voice channels plus six 2 op voice channels. The six 2 op voice channels can be configured for 5 percussion channels a 3 voice channels or 6 voice channels. In the case of the 4 -op channels the above diagram shows which two op channels go into building the six 4 op channels.

With 4 operators, the following diagram shows how the operators can be connected. The connection bits from the first two operators is designated as C0 and the connection bits from the remaining two are designated as C1.

Figure 3 - Connection Possibilities with 4 operators
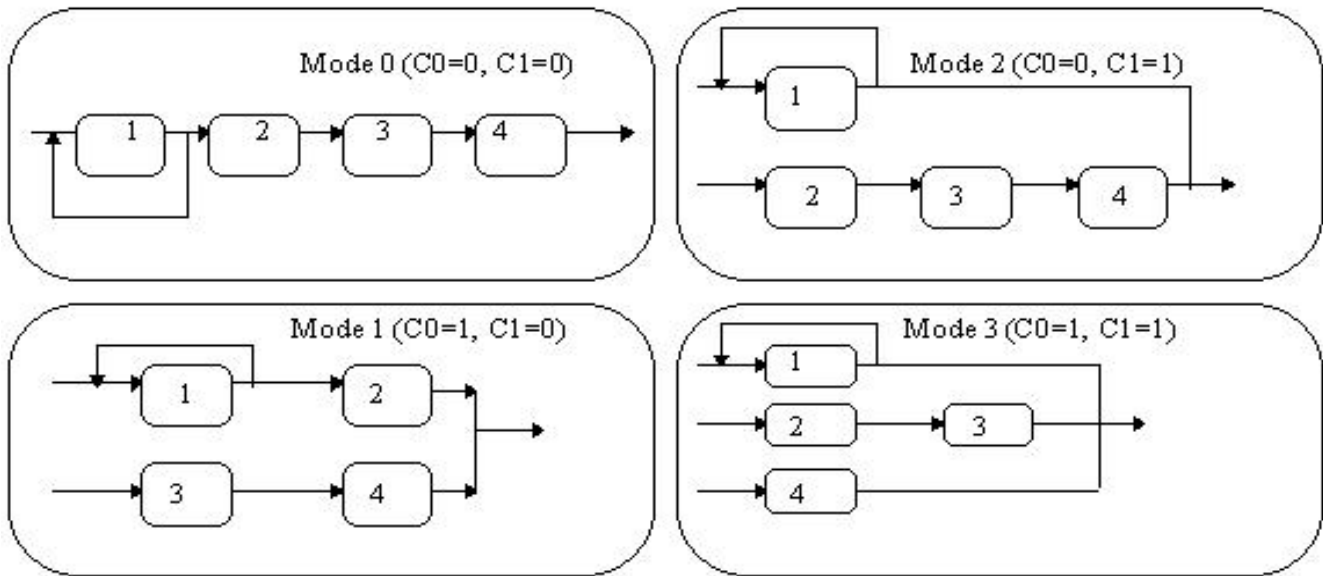
MIDI Interface

Introduction

The MIDI interface is a hi-speed serial interface running at 31,250 baud. There are 8 data bits with 1 start and 1 stop bit. The dmmidi0 device essentially provides simple read and write access to the MIDI device on the sound card.

Table 9 lists the MIDI channel voice messages. These are the most common messages, and are used to control an instrument's voices.

Table 9 - MIDI Channel Voice Messages

Status Byte	Data Bytes	Description
1000cccc (cccc is channel #) 0x80 - 0x8F	0nnnnnnn (Note Number) 0vvvvvvv (Velocity)	Note Off Event. This event is sent when a note is released.
1001cccc (cccc is the channel #) 0x90 - 0x9F	0nnnnnnn (Note Number) 0vvvvvvv (Velocity)	Note On Event. This message is sent when a note is depressed.
1010cccc (cccc is the channel #) 0xA0 - 0xAF	0nnnnnnn (Note Number) 0vvvvvvv (New Velocity)	Polyphonic Key Pressure Event. This message is sent when the velocity of a previously triggered note is changed.
1011cccc (cccc is the channel #) 0xB0 - 0xBF	0ccccccc (Controller) 0vvvvvvv (New Value)	Control Change Event. This message is sent when a controller such as dials and pedals change their value. Certain numbers are reserved for standard controllers.
1100cccc (cccc is the channel #) 0xC0 - 0xCf	0ppppppp (Program Number)	Program Change Event. This event is sent to change the patch or the instrument on a specified channel.
1101cccc (cccc is the channel #) 0xD0 - 0xDF	0ccccccc (Channel Number)	Channel Pressure (After Touch). Use this message to send the single greatest velocity of all notes depressed.
1110cccc (cccc is the channel #) 0xE0 - 0xEf	01111111 (LSB 7 bits) 0mmmmmmm (MSB 7 bits)	Pitch Wheel Change. This message is sent when the pitch bend wheel is changed. The center position has a value of 0x2000. Pitch bend is measured by a 14-bit value.

Table 10 summarizes the MIDI System Exclusive, System Common, and System Real Time Messages. System Exclusive messages are used for transferring data in a manufacturer-dependent manner. System Common messages are directed at all MIDI receivers in a system. System Real Time messages are used for synchronization between clock-based MIDI devices.

Table 10 - MIDI System Messages

Real Time / System	Data Bytes	Description
0xF0 System Exclusive	Variable Length	Uses to send Sequencer Specific Messages. First Data Byte should be the Manufacturer's ID. Message is terminated by 0xF7 (EOX).
0xF1 Undefined		
0xF2 Song Position	14-bit value, LSB first	Song Position used in Karaoke systems.
0xF3 Song Select	1 byte Song Number	Used to select a song in a list of stored songs in a sequencer.
0xF4 Undefined		
0xF5 Undefined		
0xF6 Tune Request	None	Used to request analog synthesizers to tune their oscillators.
0xF7 EOX Terminator	None	Used to terminate a System Exclusive message
0xF8 Timing Clock	None	Use to sync devices such as drum machines. Timing Clock messages are sent at the rate of 24 clocks per quarter note.
0xF9 Undefined	None	
0xFA Start	None	Sent to start a sequencer or external MIDI unit.
0xFB Continue	None	Causes the device to pick up at the next clock mark.
0xFC Stop	None	Sent to stop a sequencer or external MIDI unit.
0xFD Undefined	None	
0xFE Active Sensing	None	Sent every 300ms. It is used to implement a timeout mechanism for a receiver to revert back to its default state. OSS will filter this byte out from the data received from a MIDI port so applications will never see it in incoming data.
0xFF System Reset	None	Initializes the MIDI controller to its power on defaults. Applications should never write this byte; it is reserved for future expansion by OSS.

MIDI Note Specification

Table 11 shows the correspondence between MIDI note pitch numbers and note frequencies. For FM synthesis, the octave is simply the note pitch number divided by 12 (NP/12) and the frequency is note pitch number in the note table modulo 12 (table[NP%12]) where the note table is the table of note names to frequencies described in the section on FM synthesizer programming.

Table 11 - MIDI Note Pitch Numbers and Frequencies

Note Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
76	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

Reading From MIDI Instruments

In order to read input from a MIDI synthesizer, you need to first loop on reading the status byte. Depending on the type of the status byte, additional bytes are read from the MIDI keyboard. Reading data from the MIDI controller is achieved as follows:

1.   Open the SoundBlaster MIDI device using an `open` call. This returns a MIDI device file descriptor.

2.   Read one byte of status info. Use a `read` call on the MIDI device file descriptor to read 1 byte.

3.   Decode the status byte using the table above. If the status byte requires 2 data bytes, then read two bytes of data into different variables (one byte at a time). If the status byte requires 1 data byte then issue a single byte read.

4.   Perform the necessary operations on the data (send it to the FM synthesizer).

5.   Loop on reading the status info byte.

NOTE

Devices can make use of the *running status* feature of the MIDI protocol. Consecutive messages of the same type can omit the status byte. This is a common source of confusion when decoding MIDI messages.

Reading From MIDI Files Using Midilib

To read MIDI codes from MIDI files in format 1 or 0 requires the use of the standard MIDI file library which is supplied along with the software. Please read the documentation on reading MIDI files using the library. The following example (taken from the library example) demonstrates the scheme to read MIDI files and output music. MIDI files basically contain a header with track and SMPTE time code information. The header is also responsible for the tempo of the MIDI tune. Following the header is a track start flag with SMPTE time codes followed by a MIDI opcode described above followed by two or three bytes of data depending on the MIDI opcode. The following scheme describes how to process MIDI files for output to the FM synthesizer or a MIDI synthesizer connected to the MIDI port.

1.   First define your MIDI functions to perform the following basic MIDI codes:

        Key Off
        Key On
        Program Change
        Key Pressure
        Channel Pressure
        Pitch Wheel
        Controller Change (Mf_parameter)
        System Exclusive

These functions are called by the library and can be used to either hook into the FM synthesizer or output directly via a `write` to the MIDI port.

2.   From `main`, you need to initialize all the MIDI library function pointers to point to your specific functions. Otherwise library defaults are taken. In most cases you will have to define the MIDI library function `Mf_getc` to point to a local `getc` type of function which reads a character from the MIDI file and returns it to the calling function in the library. In order to define a `getc` type of routine, you will have to pass a file descriptor which is obtained using an `fopen` system call.

3.   Issue a call to midifile routine which reads the MIDI format 1 files and parses the information.

4.   Close the open file descriptor.

The C-code below describes how to declare local functions. The only functions you need to define for parsing MIDI files are:

```
Mf_error = error;
Mf_header =  txt_header;
Mf_starttrack =  txt_trackstart;
Mf_endtrack =  txt_trackend;
Mf_on =  txt_noteon;
Mf_off =  txt_noteoff;
Mf_pressure =  txt_pressure;
Mf_controller =  txt_parameter;
Mf_pitchbend =  txt_pitchbend;
Mf_program =  txt_program;
Mf_chanpressure =  txt_chanpressure;
Mf_sysex =  txt_sysex;
Mf_metamisc =  txt_metamisc;
Mf_seqnum =  txt_metaseq;
Mf_eot =  txt_metaeot;
Mf_timesig =  txt_timesig;
Mf_smpte =  txt_smpte;
Mf_tempo =  txt_tempo;
Mf_keysig =  txt_keysig;
Mf_sqspecific =  txt_metaspecial;
Mf_text =  txt_metatext;
Mf_arbitrary =  txt_arbitrary;
```

These functions handle the basic MIDI codes described above as well as handle errors, SMPTE and SysEx codes found in mode MIDI Format 1 files. You may wish to ignore them if you need rudimentary MIDI capabilities. The following code uses an array to hold about 100000 MIDI events which are written to the MIDI port of the SoundBlaster. This array is sorted according to the SMPTE time code since MIDI files handle one track at a time.

Listing 12 - Reading a MIDI File

```
/*
 * mftext
 *
 * Convert a MIDI file to verbose text.
 */
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include <fcntl.h>
#include "midifile.h"

static FILE *F;
struct event {
                    char data0;
                    char data1;
                    char data2;
                    int time_stamp;
};

struct event midievent;
int current_event = 0;
```

```c
main(argc,argv)
char **argv;
{
FILE *efopen();
int blastfd;

        if ( argc > 1 )
                F = efopen(argv[1],"r");
        else
                F = stdin;
        blastfd = open("/dev/sbpmidi0", O_WRONLY);
/* Initialize the function pointer in the MIDI file library */
        initfuncs();
        Mf_getc = filegetc;

/* Start processing the MIDI file */
        midifile();
        fclose(F);
/* qsort the events based on the time stamp */
        qsort(midievent, current_event, sizeof(event), compare);

/* write the events out to the MIDI port. Wait for the time on the event */
        for (i=0; i<current_event; i++) {
                waitfor(midi_event[i].time_stamp);
                write(blastfd, midi_event[i].data0, 1);
                write(blastfd, midi_event[i].data1, 1);
/* Write the 3 byte of data only if necessary */
                if (midievent[i].bytes == 3)
                        write(blastfd, midi_event[i].data3, 1);
        }
        exit(0);
}

/* This is the getc routine to read the MIDI file */
filegetc()
{
        return(getc(F));
}

/* This routine opens the MIDI file and flags errors */
FILE *
efopen(name,mode)
char *name;
char *mode;
{
        FILE *f;
        extern int errno;
        extern char *sys_errlist[];
        extern int sys_nerr;
        char *errmess;
        if ( (f=fopen(name,mode)) == NULL ) {
                (void) fprintf(stderr,"*** ERROR *** Cannot open
'%s'!\n",name);
                if ( errno <= sys_nerr )
                        errmess = sys_errlist[errno];
                else
                        errmess = "Unknown error!";
                (void) fprintf(stderr,"************* Reason: %s\n",errmess);
                exit(1);
        }
        return(f);
}
```

```c
/* Compare two events -- used for the qsort c-library function */
int compare(event *one, event *two)
{
        return (one->time_stamp - two->time_stamp)
}

/* suspend the program until the time is up for the event */
waitfor(long currtime)
{
static ulong last_event = 0;
struct timeval tv;
ulong usecs;

        if (last_event == 0) {
                last_event = currtime;
                return;
        }
        usecs = currtime - last_event;
        last_event = currtime;
        tv.tv_sec = usecs/1000000;
        tv.tv_usec = usecs % 1000000;
        select(0,0,0,0, &tv);
}

/* print errors in the MIDI file */
error(s)
char *s;
{
        fprintf(stderr,"Error: %s\n",s);
}

/* Print the header information regarding MIDI format, tracks and tempo */
txt_header(format,ntrks,division)
{
        printf("Header format=%d ntrks=%d
division=%d\n",format,ntrks,division);
}

txt_trackstart()
{
        printf("Track start\n");
}

txt_trackend()
{
        printf("Track end\n");
}

/* Handle key off events - Refer to MIDI opcode chart */
txt_noteon(chan,pitch,vol)
{
        prtime();
        printf("Note on, chan=%d pitch=%d vol=%d\n",chan+1,pitch,vol);
        midievent[current_event].data0 = chan;
        midievent[current_event].data1 = pitch;
        midievent[current_event].data2 = vol;
        midievent[current_event++].bytes = 3;
}

/* Handle key on events - Refer to MIDI opcode chart */
txt_noteoff(chan,pitch,vol)
{
```

```
        prtime();
        printf("Note off, chan=%d pitch=%d vol=%d\n",chan+1,pitch,vol);
        midievent[current_event].data0 = chan;
        midievent[current_event].data1 = pitch;
        midievent[current_event].data2 = vol;
        midievent[current_event++].bytes = 3;
}

/* Handle key pressure event - Refer to MIDI opcode chart */
txt_pressure(chan,pitch,press)
{
        prtime();
        printf("Pressure, chan=%d pitch=%d press=%d\n",chan+1,pitch,press);

}

/* Handle control change events - Refer to MIDI opcode chart */
txt_parameter(chan,control,value)
{
        prtime();
        printf("Parameter, chan=%d c1=%d c2=%d\n",chan+1,control,value);
        midievent[current_event].data0 = chan;
        midievent[current_event].data1 = control;
        midievent[current_event++].data2 = value;
}

/* Handle pitch bend - Refer to MIDI opcode chart */
txt_pitchbend(chan,msb,lsb)
{
        prtime();
        printf("Pitchbend, chan=%d msb=%d lsb=%d\n",chan+1,msb,lsb);
        midievent[current_event].data0 = chan;
        midievent[current_event].data1 = msb;
        midievent[current_event].data2 = lsb;
        midievent[current_event++].bytes = 3;
}

/* Handle program change events - Refer to MIDI opcode chart */
txt_program(chan,program)
{
        prtime();
        printf("Program, chan=%d program=%d\n",chan+1,program);
        midievent[current_event].data0 = chan;
        midievent[current_event].data1 = program;
        midievent[current_event++].bytes = 2;
}

/* Handle channel pressure events - refer to MIDI opcode chart */
txt_chanpressure(chan,press)
{
        prtime();
        printf("Channel pressure, chan=%d pressure=%d\n",chan+1,press);
        midievent[current_event].data0 = chan;
        midievent[current_event].data1 = press;
        midievent[current_event++].bytes = 2;
}

/* Handle sysex events - Refer to MIDI opcode chart */
txt_sysex(leng,mess)
char *mess;
{
        prtime();
```

```
        printf("Sysex, leng=%d\n",leng);
}

/* Unrecognized meta events in the MIDI file - flag warnings */
txt_metamisc(type,leng,mess)
char *mess;
{
        prtime();
        printf("Meta event, unrecognized, type=0x%02x leng=%d\n",type,leng);
}

txt_metaspecial(type,leng,mess)
char *mess;
{
        prtime();
        printf("Meta event, sequencer-specific, type=0x%02x
leng=%d\n",type,leng);
}

txt_metatext(type,leng,mess)
char *mess;
{
        static char *ttype[] = {
                NULL,
                "Text Event",        /* type=0x01 */
                "Copyright Notice",    /* type=0x02 */
                "Sequence/Track Name",
                "Instrument Name",  /* ...        */
                "Lyric",
                "Marker",
                "Cue Point",            /* type=0x07 */
                "Unrecognized"
        };
        int unrecognized = (sizeof(ttype)/sizeof(char *)) - 1;
        register int n, c;
        register char *p = mess;        if ( type < 1 || type > unrecognized
)
                type = unrecognized;
        prtime();
        printf("Meta Text, type=0x%02x (%s)
leng=%d\n",type,ttype[type],leng);
        printf("      Text = <");
        for ( n=0; n<leng; n++ ) {
                c = *p++;
                printf( (isprint(c)||isspace(c)) ? "%c" : "\\0x%02x" , c);
        }
        printf(">\n");
}

txt_metaseq(num)
{
        prtime();
        printf("Meta event, sequence number = %d\n",num);
}

txt_metaeot()
{
        prtime();
        printf("Meta event, end of track\n");
}

txt_keysig(sf,mi)
```

```
{
        prtime();
        printf("Key signature, sharp/flats=%d  minor=%d\n",sf,mi);
}

txt_tempo(tempo)
long tempo;
{
        prtime();
        printf("Tempo, microseconds-per-MIDI-quarter-note=%d\n",tempo);
}

txt_timesig(nn,dd,cc,bb)
{
        int denom = 1;
        while ( dd-- > 0 )
                denom *= 2;
        prtime();
   printf("Time signature=%d/%d  MIDI-clocks/click=%d
32nd-notes/24-MIDI-clocks=%d\n",
                nn,denom,cc,bb);
}

/* Parse the SMPTE time stamp and print the information */
txt_smpte(hr,mn,se,fr,ff)
{
        prtime();
        printf("SMPTE, hour=%d minute=%d second=%d frame=%d fract-frame=%d\n",
                hr,mn,se,fr,ff);
}

txt_arbitrary(leng,mess)
char *mess;
{
        prtime();
        printf("Arbitrary bytes, leng=%d\n",leng);
}

prtime()
{
        printf("Time=%ld  ",Mf_currtime);
}

/* Initialize functions - refer to midifile.h for details */
initfuncs()
{
        Mf_error = error;
        Mf_header =  txt_header;
        Mf_starttrack =  txt_trackstart;
        Mf_endtrack =  txt_trackend;
        Mf_on =  txt_noteon;
        Mf_off =  txt_noteoff;
        Mf_pressure =  txt_pressure;
        Mf_controller =  txt_parameter;
        Mf_pitchbend =  txt_pitchbend;
        Mf_program =  txt_program;
        Mf_chanpressure =  txt_chanpressure;
        Mf_sysex =  txt_sysex;
        Mf_metamisc =  txt_metamisc;
        Mf_seqnum =  txt_metaseq;
        Mf_eot =  txt_metaeot;
        Mf_timesig =  txt_timesig;
```

```
        Mf_smpte =  txt_smpte;
        Mf_tempo =  txt_tempo;
        Mf_keysig =  txt_keysig;
        Mf_sqspecific =  txt_metaspecial;
        Mf_text =  txt_metatext;
        Mf_arbitrary =  txt_arbitrary;
}
```

Music Programming

Introduction

This section describes the programming of different kinds of music and MIDI related applications using OSS. These include full featured MIDI sequencers as well as simpler MIDI playback and recording programs. The MIDI programming interfaces provided by OSS are based on events such as key press and key release. The applications using these interfaces don't produce the audio data sent to the speakers themselves. Instead they control some kind of hardware (synthesizers) which perform the sound generation. For example, a MIDI playback application can send note on and off messages to an external MIDI synthesizer or keyboard which is connected to a MIDI port using a MIDI cable (the MIDI device can be internal to the computer, too).

Another approach is that the application does all this itself and produces a stream of audio samples. These samples are sent directly to /dev/dsp. This kind of approach is used by some well-known MIDI and module players such as Timidity and Tracker. Implementing this kind of application is beyond the scope of this guide.

The basic foundation behind the music programming interfaces of OSS is the MIDI 1.0 specification. While the API provided by OSS may look different from MIDI, there are a lot of similarities. Most events and parameters defined by the OSS API directly follow the MIDI specification. The few differences between OSS API and MIDI are extensions defined by OSS. These extensions make it possible to control built-in synthesizer (wave table) hardware in a way which is not possible or practical with plain MIDI. When applicable, the OSS API follows the General MIDI (GM) and Yamaha XG specifications which further specify how things work.

You should have some degree of understanding of the MIDI and General MIDI specifications (the more the better) before proceeding with this section. The official MIDI specification is available from the MIDI Manufacturers Association (MMA). Their web site (http://www.midi.org) contains some online information. Additional information can be found on various Internet sites as well as from several MIDI related books. Information about the XG MIDI specification is available from Yamaha (http://www.ysba.com).

Midi And Music Programming Interfaces Provided By OSS

Open Sound System provides three different device interfaces for MIDI and music programming. Each of them are intended for a slightly different use.

Fundamentals Of /dev/music

MIDI (music) is a highly real-time process. An experienced listener can pick very minor timing (rhythm) errors from the music being listened to, which makes timing accuracy one of the main goals of the OSS implementation. Unfortunately, general purpose (multiuser and multitasking) computer

systems are not well suited to this kind of tasks. For this reason OSS has been implemented in a way which makes timing precise even in highly loaded systems.

The key idea behind /dev/music and /dev/sequencer interfaces is to make the application and the hardware work asynchronously. This is implemented by separating the application and the playback logic using large buffers. The buffer can hold enough playback data for several seconds. Since the playback process occurs asynchronously in the background, the application can do other processing (graphics updates, for example) without the need to babysit the music playback. The only requirement is that it should write new data before the queue drains completely and causes audible timing errors. In a similar way, input data is queued until the application has time to read it from the buffer.

Queues and Events

The central part of the /dev/music and /dev/sequencer APIs is queuing. There is a queue both for playback and recording. Everything written to the device is first placed at the tail of the playback queue. The application continues it's execution immediately after the data is put on the queue. This happens immediately except in situations where there is not enough space in the queue for all the data. In this case the application blocks until some old data gets played.

It's very important to notice that the playback is not complete when the write call returns. The playback process still continues in the background until all data has been played. This delay depends on timing information included in the playback data and can sometimes be several minutes (even hours or days in some cases). Even after the output buffer has drained, some notes not being explicitly stopped may continue playing (infinitely) until the application writes more data containing the note off command for this note. It's very important to understand this asynchronous behavior of the API. Even when the application tells the playback engine to wait some time (even hours) the associated write may return immediately. The application never waits until the requested time is occurred. After you understand this and have read the MIDI specification you know most the important concepts regarding /dev/music and /dev/sequencer programming.

Similarly, all input data is first appended to the recording queue where it sits until the application reads them off. There is embedded timing information in the data read from the device file which the application should analyze to acquire the actual time of the event.

The data written to or read from the device file is organized as a stream of events. Events are records of 8 or 4 bytes containing a command code and some parameter data. When using /dev/music all events are 8 bytes long. With /dev/sequencer some events are 4 bytes long (mainly for compatibility reasons with older software). Formatting of these events is defined in appendix A. However, applications should never create the event records themselves. Instead they should use the API macros defined later in this chapter.

The playback engine always processes the events in the order they are written to the device. However, there is an ioctl call that can be used to send events immediately (ahead of the queued

data) to the engine. This feature is intended to be used for playing real-time events that occur in parallel to the pregenerated event stream stored in the playback queue.

There are two main types of events. Timing events are commands that control timing of the playback process. They are also included in the recording data before input events (if the time has changed since the previous received event). The playback engine uses these events to delay playback as instructed by the application. The playback engine maintains absolute time since starting the playback (the application can restart the timer whenever it likes). When it encounters a timing event it computes the time when the subsequent event needs to be processed. It then suspends the playback process until the real-time timer gets incremented to this value. After that moment, the playback process continues by executing the next event in the queue (which can sometimes be another timing event).

When an input event is received from one of the devices (usually MIDI ports) the driver writes a time stamp event containing the current real time to the input queue and then appends an event corresponding to the data received from the device. However, the timestamp is written only if it's time is different from the previously received event (to prevent the input queue from filling up unnecessarily in case of sudden input bursts). Finally, the application reads both these events from the queue when it has time to process the input queue. It's possible for the application to merge the newly received input events with the old playback data based on these timestamps.

There is a fundamental difference in timing behavior between /dev/sequencer and /dev/music. The /dev/sequencer device uses fixed timing based on the resolution of the system timer. In most cases the system timer ticks once every 1/100th of second (100 Hz). However in some types of systems this rate is different (such as 1000 Hz). It is the application's responsibility to check the timing rate before using the device. The /dev/music device uses an adjustable timer which supports selecting different tempos and timebases.

The second main type of event is active events. These events are played whenever they reach the head of the playback queue. They are used mainly for sound generating purposes but also for changing various other parameters. These events are instantaneous by definition (they don't consume any time). However, in some cases they may cause some processing delays, for example when a byte is sent to a MIDI port whose hardware level output buffer is full. When no timing events are present in the buffer, the playback engine plays all active events as fast as it can. The same thing also happens when timing events have already been expired when they are written to the device file.

MIDI Ports and Synthesizer Devices

The /dev/music and /dev/sequencer APIs are based on devices. There can be from 0 to N devices in the system at the same time. The API differentiates between these devices by using unique device numbers. It's important to notice that all these devices can be used at the same time. For some reason it looks like most applications using this API use only one device at the same time (which is usually selected using a command line parameter). There are two main types of devices, described in the following sections.

MIDI Ports

MIDI ports are serial communication ports that are present on almost every sound card. Usually they are called MPU401 (UART) devices. There are even dedicated (professional) MIDI only cards that don't have audio capabilities at all. A MIDI port is just a dumb serial port which doesn't have any sound generation capabilities or other intelligence itself. All it does is provide the capability to connect to an external MIDI device using standard MIDI cabling. An external MIDI device can be a full featured MIDI keyboard or a rack mounted tone generator without a keyboard. The MIDI cable interface can also be used to control almost any imaginable device from a MIDI controlled mixer or flame thrower to a washing machine. The MIDI interface is simply used to send and receive bytes of data which control the devices connected to the port. It's possible to have an almost unlimited number of devices on the same MIDI interface by daisy-chaining them or by using external MIDI multiplexing devices. So in practice, a command sent to the MIDI cable may get processed by an unlimited number of devices. Each of them react to the command depending on their internal configuration.

Most sound cards have a so-called wavetable connector on them. This connector can be used to connect a MIDI daughter card. Actually, the wavetable connector is just a branch of the MIDI interface of the parent sound card. Everything written to the MIDI port gets sent both to the wave table daughter card and to the MIDI connection port (usually shared with a joystick port) on the back of the sound-card. Another way to add MIDI devices to a sound card is to solder a MIDI chip on the card itself. In practice this doesn't differ from the daughter card interface in any way.

The common thing between the various ways to implement MIDI devices is that OSS sees just a port which can send and receive MIDI data. In practice it doesn't know anything about the devices connected to the port so it doesn't care about it. It's possible that there are no devices or even a cable connected to the port. In this case playback using this port doesn't generate any sound which may confuse some users.

Another common point between all devices connected to MIDI ports is that they are self contained. The devices contain all the necessary instrument (patch) data. There is no need for the application to worry about so called patch caching when using MIDI ports.

Internal Synthesizers

Synthesizer devices are sound chips (usually based on wave table or FM synthesis) that are always mounted directly on the sound card or system's motherboard. The other main difference is that they provide tighter connection to the OSS driver. OSS has direct control of every hardware level feature of the synth chip while devices connected to a MIDI port can be controlled only by sending MIDI messages to the port. This means that synth devices have usually some capabilities beyond ones provided by plain MIDI (however this will not necessarily be true in the future). The drawback is that both OSS and the application have additional responsibilities which make use of the (old) /dev/sequencer API very tricky with them. For this reason, use of the /dev/sequencer interface is strongly discouraged. The /dev/music API fixes most of these problems but leaves some additional

tasks such as so called patch caching to the application (which will be described later in this chapter).

The currently supported synthesizer chips are the following:

1. Yamaha OPL2/OPL3 FM synthesizer. The OPL2 chip was used in the first widely used sound card (AdLib) in the late 80s. OPL3 is it's successor, originally introduced in the SoundBlaster Pro and still widely used for DOS games compatibility in almost every sound card. FM synthesis provides rich capabilities to produce synthetic sounds. However, it's very difficult to emulate acoustic instrument sounds using it. In addition, the OPL3 chip has a very limited amount of simultaneous voices which makes it practically obsolete. OPL4 is a combined FM and wave table sound chip compatible with OPL3.

2. Gravis Ultrasound (GUS) was the first wave table based sound card on the market. It provides the capability to play up to 32 simultaneous voices by synthesizing them from wave table samples stored on it's on board RAM (up to 8 MB in the latest models but just 512K in the original one). The wave table capability made this card very useful for playing so called module (.MOD, etc) music using 386 and 486 computers of the early 90s. However, major advances in CPU speeds have made this approach very impractical when compared to mixing in software (except when a very large number of voices are used at the same time). The main problem with GUS is it's limited memory capacity which doesn't permit loading the full GM patch set simultaneously. This means that applications supporting GUS must be able to do patch loading/caching. The driver interface originally developed for GUS defines a de facto API which is supported by other wave table device drivers (of OSS) too. This means that programs written for GUS work also with the other ones with some minor modifications.

3. Emu8000 is the wave table chip used on SoundBlaster 32/63/AWE cards. It's very similar with GUS but provides a GM patch set on ROM. This means that patch loading/caching is not necessary (but still possible).

4. SoftOSS is a software based wave table engine by 4Front Technologies. It implements the OSS GUS API by doing the mixing in software. This makes it possible to use any 16 bit sound card (without wave table capabilities) to play with wave table quality instruments. However this mixing process consumes CPU cycles and system RAM which can cause some problems with performance critical applications and/or on underconfigured systems.

In addition to the above, OSS supports some wave table chips which work as MIDI port type devices.

Differences Between Internal Synthesizer and MIDI Port Devices

There is no fundamental difference between these two device types when using the /dev/music interface. The only practical difference is that the internal synth devices need some patch management capabilities from the application. Together with libOSSlib these differences are rather

minimal.

However, the situation is very different with /dev/sequencer. In fact there is nothing common with these device types. There are completely different interfaces for both of these devices. In addition there are some differences between OPL3 and wave table devices with /dev/sequencer which make it difficult to use. For this reason using the /dev/sequencer interface is not recommended.

The /dev/music and /dev/sequencer API acts as a multiplexer which dispatches events to all devices in the system. The application merges the events going to all devices to the same output stream and the playback engine sends them to the destination device. When recording it places input from all input devices to a common input queue where the application picks them (the application should be prepared to handle merged input from multiple devices or to filter the unnecessary data based on the source device number.

All devices known by the driver are numbered using an unique number between 0 and number_of_devices - 1. However, the numbering is slightly different depending on the device file being used. With /dev/sequencer separate numbering is used for internal synthesizer devices and MIDI ports while /dev/music knows only synthesizer devices (MIDI ports are masqueraded as synth devices too). More information about device numbering will be given in the programming section.

Instruments and Patch Caching

The common feature between all MIDI and synthesizer devices is that they produce sound synthetically. Very often they emulate other (acoustic) instruments but many devices can create fully artificial instrument sounds too. Practically all devices are multitimbral which means that they can emulate more than one instrument. Switching between different instruments/programs is done using MIDI program change messages (actually it's equivalent in the OSS API).

Programs are numbered between 0 and 127. The meanings of these program numbers are determined (freely) by the playback device. However in practice all modern devices follow the General MIDI (GM) specification which binds the program numbers to fixed instruments so that, for example, the first instrument is an acoustic piano. It should be noted that in OSS (just like in the MIDI protocol) device numbering starts from 0, however in many tables and books the numbering starts from 1.

OSS assumes that the devices are GM compatible and that the application using the API is GM compatible too. The instrument and program numbers are defined to be GM compatible. However, it's possible for the application to use any other numbering scheme provided that the devices being used support it.

To be able to produce any sound the synthesized device needs some kind of definition for the instrument. The exact implementation depends on the type of the device. For example, with devices using FM synthesis (OPL2/3) the instrument is defined by a set of few parameters (numbers). Devices based on wave table synthesis use prerecorded instrument samples and some additional control information. The information required for one instrument by a particular instrument is called

a patch.

In most cases all the instrument information is stored permanently in the device (for example on ROM chips). In this case the instruments are always there and the playback application doesn't need to care about this. It's usually possible to the application to modify the instruments or even to create new ones but it's beyond the scope of this guide. However there are devices that don't have permanently installed instruments. They just have a limited amount of memory in which the instrument definitions need to be loaded on demand. This process is called patch caching. The OSS API defines a simple mechanism which the application should use to support patch caching devices. The core of this mechanism is OSSlib library which can be linked with the application.

Notes

The main task in playing music using the /dev/music and /dev/sequencer interface is playing notes. For this purpose there are two messages in the MIDI specification. The note on message is used to signal the condition where a key was pressed on the keyboard. The message contains information about the key that was pressed and the velocity it was pressed. When receiving this message the MIDI device behaves just like an analog keyboard instrument (such as piano) by sounding a voice. The pitch of the voice is determined by the key number and the volume is determined by the velocity with which the key was hit. Other characteristics of the voice depend on the instrument that was selected before the note on message.

After a note on message, the sound starts playing on it's own. Depending on the instrument characteristics it may decay immediately or continue playing indefinitely. In any case, each note on message should be followed by a note off message for the same note number. After this message the voice will decay according to the instrument characteristics (it may even already have decayed prior the note off message).

Both the note on and the note off message contain a note number (0 to 127). The note number is simply the number of the key on the keyboard. A value of 60 specifies the middle C.

The OSS API defines events for all MIDI messages including the note on and note off ones.

Voices and Channels

At the lowest level all devices produce sounds using a limited number of operation units called voices. To play a MIDI note the device usually needs one voice but it's possible that it uses more of them (this is called layering). The number of simultaneously voices (degree of polyphony) is limited by the number of voices available on the device. With primitive devices the number of voices can be very low (9 with OPL2 and 18 with OPL3). Most devices support 30 or 32 voices. Some more recent devices support 64 or 128 voices which is the future trend.

When using the /dev/sequencer API the application needs to know how many voices are supported by the particular device. It also needs some kind of mechanism for allocating voice operators for the

notes to be played. The voice number needs to be used as a parameter in all note related events sent to the driver. This task is usually very complicated due to need to handle out of voices situations. For this reason it's recommended to use the /dev/music interface which handles all of this automatically.

The /dev/music API is based channels, just like MIDI. There are 16 possible channels numbered between 0 and 15. It's possible to assign a separate instrument to each channel. Subsequent notes played on this channel will be played using the instrument previously assigned to the channel. Any number of notes can be playing on each channel simultaneously. However, the number of notes actually playing depends on the number of voices supported by the device. When using /dev/music there is no need to do the voice allocation by the application. The application just tells which notes to play on which channels and the device itself takes care of the voice allocation. This makes /dev/music significantly easier to use than /dev/sequencer.

Controlling Other Parameters

The MIDI specification contains some other messages in addition to the basic note on and note off messages. They can be used to alter the characteristics of notes being played and they usually work on a channel basis (i.e. they affect all notes played on a particular channel). Most of these functions are implemented using MIDI control change messages. The OSS API contains an event for all defined MIDI controllers.

Programming /dev/music and /dev/sequencer

In this guide we handle mainly /dev/music programming. The differences between /dev/music and / dev/sequencer interfaces will be described shortly whenever they are encountered in the text.

Initial Steps

This guide is written for OSS version 3.8 or later. There are a few additions made to the OSS API in version 3.8 which mean that certain features will not work with earlier OSS versions (mainly OSSlib). In any case, at least version 3.5 of OSS is required (earlier versions are not supported any more).

For simplicity reasons it's assumed that the OSSlib interface is being used. OSSlib is a library that handles patch caching in an almost transparent way. With OSSlib the application doesn't need to be aware of the details of the particular synthesizer hardware being used.

The file libOSSlib.a (or libOSSlib.so in some operating systems) is distributed as a part of the commercial OSS software. Another way to obtain it is to download snd-util-3.8.tar.gz (or later) from ftp://ftp.opensound.com/ossfree and to compile it locally. However, this is recommended only with OSS/Free. To be able to compile OSSlib you should have OSS 3.8 or later installed on the system. It's also possible to compile OSSlib or applications using it by obtaining the <soundcard.h> file from

the OSS 3.8 distribution but this is not recommended or supported.

To use OSSlib you should use the -DOSSLIB -I/usr/lib/oss/include -L/usr/lib/ oss -lOSSlib options when compiling and linking the application. For example:

cc -DOSSLIB -I/usr/lib/oss/include -L/usr/lib/oss -lOSSlib test.c -o test

It's fairly easy to make the application usable both with and without OSSlib by using an #ifdef OSSLIB directive in the places where there are differences between these cases.

An application using the /dev/sequencer or /dev/music APIs requires some support code to be added in the application. All of this is present in the sample program given later in this chapter. This additional code is required to support buffering used by the SEQ_* macros defined in <soundcard.h>. The following has to be present:

1.   <sys/soundcard.h> must be included in each source file that uses the API.

2.   Define for the buffer being used by the API.

3.   Definition of the seqbuf_dump() routine in case you are not using OSSlib (OSSlib contains this routine).

```
/*
 * Public domain skeleton for a /dev/music compatible OSS application.
 *
 * Use the included Makefile.music to compile this (make -f Makefile.music).
 */

/*
 * Standard includes
 */

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/soundcard.h>

/*
 * This program uses just one output device which is defined by the following
 * macro. However the OSS API permits using any number of devices
 * simultaneously.
 */

#define MY_DEVICE 0 /* 0 is the first available device */

/*
 * The OSS API macros assume that the file descriptor of /dev/music
 * (or /dev/sequencer) is stored in variable called seqfd. It has to be
 * defined in one source file. Other source files in the same application
 * should define it extern.
 */
```

```
int seqfd=-1;

/*
 * A buffer needs to be allocated for buffering the events locally in
 * the program (prior writing them to the device file). The SEQ_DEFINEBUF
 * macro can be used to define the buffer. The argument is the size of the
 * buffer (in bytes). 1024 is a good size (128 events).
 *
 * Note that SEQ_DEFINEBUF() should be used only in one source file in each
 * application. In other source files you should use SEQ_USE_EXTBUF().
 */
#define BUFFSIZE 1024
SEQ_DEFINEBUF(BUFFSIZE);

/*
 * seqbuf_dump() routine is required only when OSSLib is NOT used. It's
 * purpose is to write buffered events to the device file.
 */
#ifndef OSSLIB
/*
 * NOTE! Don't ever define seqbuf_dump() in two source files or when OSSlib
 * is used. It may have unpredictable results.
 */
void seqbuf_dump ()
{
  if (_seqbufptr)
    if (write (seqfd, _seqbuf, _seqbufptr) == -1)
    {
      perror ("write /dev/music");
      exit (-1);
    }
    _seqbufptr = 0;
  }
#endif
```

Opening the Device

The music device file to be used needs to be opened in the beginning of the program. A normal open call can be used for this (fopen or other buffered I/O routines should not be used). Select /dev/music or /dev/sequencer depending on your needs. You need also to open OSSlib by calling OSS_init() in case you use OSSlib.

```
int error, ndevices, tmp;

/*
 * First open the device file (/dev/music in this case but
 * /dev/sequencer will work in the same way). The device is
 * opened with O_WRONLY since we are only going to write. Use
 * O_WRONLY or O_RDWR if you need to use input (too).
 */

if ((seqfd=open("/dev/music", O_WRONLY, 0))==-1)
{
  perror("/dev/music");
  exit(-1);
}

/*
 * Now initialize OSSlib if required.
```

78

```
 */
#ifdef OSSLIB
if ((error=OSS_init(seqfd, BUFFSIZE)) != 0)
{
  fprintf(stderr, "Failed to initialize OSSlib, error %d\n", error);
  exit(-1);
}
#endif
```

After opening the device you should check what devices are available. This can be done using the SNDCTL_SEQ_NRSYNTHS, SNDCTL_SEQ_NRMIDIS, SNDCTL_SYNTH_INFO and SNDCTL_MIDI_INFO ioctl calls which will be covered in detail later.

```
/*
 * Check that the (synth) device to be used is available.
 */

if (ioctl(seqfd, SNDCTL_SEQ_NRSYNTHS, &ndevices)==-1)
{
  perror("SNDCTL_SEQ_NRSYNTHS");
  exit(-1);
}

if (MY_DEVICE >= ndevices)
{
  fprintf(stderr, "Error: The requested playback device doesn't exist\n");
  exit(-1);
}
```

Writing Events

As said earlier, the /dev/music API is event based. In addition to a few ioctl() calls the only way to use this API is by sending events to the device. To make this task easier a macro has been created for each supported MIDI event. These macros are named SEQ_*() and they usually take one or more parameters. For example the SEQ_START_NOTE(device, channel, note, velocity) macro is used to send a MIDI key down message to the given device (synthesizer or MIDI port). This macro itself doesn't write the event directly to the device file. Instead it appends the event after the previous ones in programs local buffer. This local buffer was created using the SEQ_DEFINEBUF(size) macro in the beginning of the program. The events are queued there until SEQ_DUMPBUF() macro is called by the program or the local queue becomes full (in this case SEQ_DUMPBUF will be called automaticly to prevent from overflow). SEQ_DUMPBUF just calls the seqbuf_dump() routine defined by the program or OSSlib depending on the situation.

Due to this buffering the application should call SEQ_DUMPBUF() before it exits or before it suspends writing new events for some reason (waiting for input).

The Minimal /dev/midi Program

By combining the above four code fragments together you have all the necessary initialization code required in a program using /dev/music or /dev/sequencer. All this program does is play a note using

the selected device. This program is also available in the samples.tar.gz package available from ftp://ftp.opensound.com/ossfree.

```
/*
 * Setup timing parameters. The defaults may vary so set them
 * explicitly.
 */
tmp = 96;
if (ioctl(seqfd, SNDCTL_TMR_TIMEBASE, &tmp)==-1)
{
  perror("Set timebase");
  exit(-1);
}

tmp = 60;
if (ioctl(seqfd, SNDCTL_TMR_TEMPO, &tmp)==-1)
{
  perror("Set tempo");
  exit(-1);
}

/*
 * Next use OSSlib to cache the instrument (if required). This is
 * recommended to be done in advance (before SEQ_START_TIMER()) since
 * patch loading from disk to the device can be time consuming. Load
 * only the instruments that are required due to limited memory
 * capacity of certain devices.
 *
 * NOTE! OSSLib loads the instrument automaticly when SEQ_PGM_CHANGE
 * is called. Loading it in advance saves you from possible
 * delays associated with demand loading.
 */
SEQ_LOAD_GMINSTR(MY_DEVICE, 0); /* 0=Acoustic piano */

/*
 * Now we are ready to start playing. The first task is to start
 * the timer. This is mandatory stem since otherwise the timer
 * will never get started. It's extremely important to start the
 * timer just immediately before writing the first event. Doing
 * it too early will cause tempo problems in the beginning.
 */
SEQ_START_TIMER();

/*
 * Select the program/instrument 0 on the MIDI channel 0.
 */
SEQ_PGM_CHANGE(MY_DEVICE, 0, 0);

/*
 * Start the note (60=Middle C) on channel 0. Use 64 as velocity.
 */
SEQ_START_NOTE(MY_DEVICE, 0, 60, 64);

/*
 * Then have relative delay of 96 ticks. The delay is from the
 * previous timing event or from the time when SEQ_START_TIMER()
 * was called.
 */
SEQ_DELTA_TIME(96);

/*
```

```
 * Now stop the note. The sound will not stop immediately. The note
 * just starts decaying and fades off.
 */
SEQ_STOP_NOTE(MY_DEVICE, 0, 60, 64);

/*
 * Have a final delay of 1000 ticks. This gives the last note(s) time
 * to decay naturally. Closing the device without this delay just
 * aborts all voices prematurely.
 */
SEQ_DELTA_TIME(1000);

/*
 * Finally flush all events still in the local buffer (mandatory
 * step before closing the device or prior pausing the application.
 * It's the SEQ_DUMPBUF() call that actually writes the events to the
 * device.
 */
SEQ_DUMPBUF();
close(seqfd);
exit(0);
}
```

The Virtual Mixer

OSS has an optional feature called software mixing which permits simultaneous playback of up to 8 audio streams. This feature is a separately priced option and available only to the customers who have purchased the MIX option. The seven day evaluation license also contains this option.

To use this option you should first enable it by configuring the virtual mixer driver. This can be done by adding one of the "4Front Virtual Mixer" devices using the "Add new card/device" function of soundconf.

NOTE

In earlier versions of OSS the SoftOSS driver also enabled virtual mixer. Since OSS 3.9.1e this is no longer true. Instead of SoftOSS you should configure the "Virtual Mixer" device. This will also enable SoftOSS. Don't configure SoftOSS at the same time as the virtual mixer.

After the software mixing driver is installed the additional 8 audio devices will be shown as below by "cat /dev/sndstat":

```
2: SoftOSS v1.2 CH #0
3: SoftOSS v1.2 CH #1
4: SoftOSS v1.2 CH #2
5: SoftOSS v1.2 CH #3
6: SoftOSS v1.2 CH #4
7: SoftOSS v1.2 CH #5
8: SoftOSS v1.2 CH #6
9: SoftOSS v1.2 CH #7
```

In the above case the first software mixing device is /dev/dsp2 (/dev/audio2) and the last one is /dev/dsp9 (/dev/audio9). You can have up to 8 of these devices active at the same time. Note that the real audio device (usually /dev/dsp0) will not be available while any of the software mixing devices are open. Also, the software mixing devices cannot be used while the actual hardware device (/dev/dsp0) is open.

To test the virtual mixer, get a couple of wav files (eg. sample1.wav and sample2.wav) and type a command such as the following:

```
play -d/dev/dsp3 sample1.wav & play -d/dev/dsp4 sample2.wav &
```

At this point you should hear both the wav files playing simultaneously. Most applications open /dev/dsp or /dev/dsp0 by default. You can make the application use a virtual audio device by simply changing the link /dev/dsp to point to /dev/dsp3 or /dev/dsp4 or any one the other virtual audio devices. However, you cannot use the virtual audio device and the physical audio device (/dev/dsp0 and /dev/dsp1) simultaneously. You will notice that there is degradation of audio quality using the virtual audio device - this is because the driver does sample rate

conversion in software. In future versions of OSS there will be a new `/dev/vdsp` device that will automatically assign an available virtual device to each application using `/dev/vdsp`. This will mean that you won't have to manually assign a virtual audio device to each application.

SoftOSS

Introduction

Until today a special wave table soundcard has been required to play high quality MIDI music. SoftOSS is a kernel module which permits doing the same using any inexpensive 16-bit soundcard together with a sufficiently fast CPU (see the System requirements section).

SoftOSS is 100% compatible with the existing wave table API of OSS which has earlier been used by the Gravis Ultra Sound (GUS) driver. This means that all Linux applications work without modification with SoftOSS.

Technical Background

SoftOSS is a virtual wave table engine that is tightly integrated with the MIDI and audio functionality of OSS. The SoftOSS engine uses CPU cycles to mix pre-recorded audio samples in control of MIDI information coming from any application using `/dev/sequencer` or `/dev/music` (formerly known as `/dev/sequencer2`) device files. The resulting 16-bit stereo audio data stream is then played using an ordinary (16-bit) soundcard (support for 8-bit sound cards will be introduced later). Since the mixing is done inside kernel it doesn't suffer from other processing activity in the system. For this reason it is possible to perform CPU intensive tasks at the same time when using SoftOSS. Sound quality is as good as in a lightly loaded system (other tasks just run slower depending on number of currently active SoftOSS voices/notes).

SoftOSS is fully compatible with the sound sample loading API originally developed for the GUS driver of OSS. This means all applications which support loading samples to GUS will work with SoftOSS without any changes.

The final version of SoftOSS will include a library called OSSlib which permits on demand loading (patch caching) of wave table samples from any programs using the `/dev/music` (`/dev/sequencer`) API of OSS. Together with changes made to `<sys/soundcard.h>`, this library permits adding patch caching to existing applications using `/dev/sequencer` and `/dev/music` with very minimal changes.

The first release of this OSSlib library will permit loading samples from `pat` format (GUS) instrument files and from standard audio files (`au`, `wav`). Later versions will support other patch file formats such as SoundFont (`sf2`). You will need a GUS compatible patch set to run SoftOSS.

A freeware version of this library will be released to permit developing OSS compatible applications with OSS/Free. In addition the freeware library will permit using applications written for OSS to work also with OSS/Free without recompiling.

Specification of the new `/dev/music` API will be released after development of OSSlib is

complete.

Applications of SoftOSS Technology

SoftOSS is mainly designed for playing MIDI music but it's well suited for some other applications too, including:

1. Sound effects in games (not necessarily background music). Since the "mixing" is done at real time priority inside kernel, it's possible to get timing precision and reliability that is not possible with any kind of process based mixing. With SoftOSS sound effects will play perfectly even in slightly under-configured machines. In addition, sound effects programming using SoftOSS and OSSlib is "fire and forget". After an effect is started the application itself doesn't need to worry about it. Of cause, the same is possible with a wave table card too. The best thing is that SoftOSS is perfectly compatible with hardware wave table devices so compatibility with SoftOSS ensures compatibility with wave table cards too.

2. Sound effects in simulators and similar applications. SoftOSS technology permits loading practically unlimited amount (currently there is an artificial limit of 8 MB) into the memory (it's limited just by amount of RAM installed in the system). Starting sounds is easy and it's even possible to change it's volume and panning (3D support is planned in the future). As with games, SoftOSS can later be replaced by a hardware wave table card without any changes to the application. However with a limited number (4 to 8) of simultaneous voices there is no benefit in using an expensive hardware wave table card.

3. Playback of pre-recorded messages, alerts, time signals and similar sounds. Future versions of SoftOSS will even permit triggering this kind of special sounds from many different applications at the same time.

System Requirements

Due to the high processing power requirements of software mixing SoftOSS is targeted to fast machines only. With current (rather non-optimized) version it's possible to play 32 simultaneous voices using 32 kHz sampling frequency using a P120 machine. However even this is better than the sampling frequency that is possible at 32 voices using a popular wave table card (19.2 kHz). With a 100 MHz PowerPC processor (RS6000/AIX) it's possible to get almost 32 simultaneous voices at 44.1 kHz.

For the above reason at least a P120 processor is recommended. However one can use SoftOSS with any 486 class (or above) machine by decreasing the sampling frequency. OSS 3.8 permits selecting a SoftOSS version which matches your CPU while configuring the device.

CPU power requirements of SoftOSS depends on concurrently playing notes (voices). You can use 44.1 kHz sampling frequency with any 486 class CPU as long as number of voices remains low. For

example playing a `mod` file using `gmod` should be possible with any machine.

Even in low-end 486 class machines, SoftOSS gives better MIDI playback quality than the standard FM synth.

Using SoftOSS with a CPU that is too slow is not dangerous. Playback just becomes distorted (it jumps like a broken vinyl record) and the system becomes rather unresponsive. However, the situation returns back to normal after playback is stopped/interrupted or the number of concurrently playing notes decreases below the system dependent limit.

At least 16 MB of RAM is required (32 MB recommended). SoftOSS stores the instrument samples in the system's (physical) RAM. This means that there must be enough spare RAM on the system. The current version of SoftOSS permits loading up to 8 MB of samples which means that using it on machines with less than 16 MB RAM may not produce the desired performance. It is possible to use SoftOSS on systems with less than 16 MB of RAM but care must be taken that too many samples are not loaded. The final 3.8 version of SoftOSS will permit configuring the maximum memory size which makes it safer to use in under-configured machines.

Limitations of SoftOSS

There are a few limitations in using SoftOSS. However, in most cases they are not significant.

SoftOSS allocates the first audio device (`/dev/dsp0`) for itself always when `/dev/sequencer` or `/dev/music` are open. This means that it is not possible to play audio at the same time with MIDI on machines with just one soundcard. `/dev/dsp0` is still accessible when `/dev/music` and `/dev/sequencer` are not open.

SoftOSS uses CPU time which may make it useless in some applications. For example, it is not recommended to use it for playing background music in games. Depending on the degree of polyphony (number of simultaneous notes) it may slow down the game seriously. Note that this is not true with playing sound effects of games, which is a well-suited task for SoftOSS.

Most GUS compatible applications would be very confused if they detect two GUS compatible devices on the same system. However, this is not a problem since GUS (particularly GUS PnP with 8 MB RAM) does everything that SoftOSS does (i.e. you don't need to use SoftOSS if you have a GUS).

Getting SoftOSS

SoftOSS is included in the standard OSS software (currently there is no extra fee).

Getting the Sound Patches

To use SoftOSS you will need to use a GUS compatible Linux application such as `mplay` or `gmod`. You can get them from the OSS Applications page. With `mplay`, you need to copy the public-domain MIDIA instrument files.

Configuring SoftOSS

Configuring SoftOSS is very easy. Run `soundconf` and you will be presented with a menu that looks something like this:

```
Save changes and Exit
Cancel changes and Exit
Add new card/device
Remove a card/device
Verify configuration
Exclude IRQ and DMA numbers
Autodetect soundcards
Security setup
Manual configuration
Install license file
```

If you have not added any sound cards yet (they should be listed above this menu), you should configure it first by activating the "Add new card/device".

To add SoftOSS you just need to activate the "Add new card/device" function and select one of the "4Front Tech. SoftOSS (for *XXX*)" entries where the *XXX* matches (roughly) your CPU. If you can't decide between two or more entries, select the highest one. If it doesn't work (playback jumps), you can start soundconf again to remove this one and to select a lower one.

After adding the SoftOSS engine and a 16-bit soundcard, save the configuration and start OSS using the `soundon` command. Then execute "`cat /dev/sndstat`" and verify that there is at least one audio device and the SoftOSS synth is listed (as shown below).

```
Audio devices:
0: Crystal audio controller (CS4236) (DUPLEX)

Synth devices:
0: SoftOSS
```

Now use the `mplay` program supplied with OSS (default is `/usr/lib/oss/mplay`) to play MIDI files. Type "`mplay # midifile.mid`" (where # is the synth number under the Synth Devices heading in the `/dev/sndstat` output.

Future Plans

The current version of SoftOSS is just a preview release. It doesn't contain all the features which are planned to be included in future versions. The following are some examples:

1. Support for sample distribution formats used for distributing commercial instrument samples and sound effects. SoundFont 2 will be the first format supported.

2. Support for streaming instruments. Streaming instruments permit playing of very large audio files or computer generated sounds together with ordinary (shorter) samples which fit completely in memory. Using streaming instruments will require some form of support by the application but most of it will be handled by OSSlib.

3. Support for 3D voice position and various special effects (this will require faster CPUs than currently available).

Advanced Programming Topics

Introduction

This chapter describes some features of OSS that are very useful or even necessary when used in the right place. However, they don't automatically make your application better if used in situations when they are not necessary. Some of the features to be presented below don't work with all devices (full duplex audio and direct DMA access, among others) or make your application very operating system dependent (e.g. direct DMA access).

It is assumed that you have a full understanding of the features described in the Introduction and Basic Audio sections of this guide. The features described here will work only if the guidelines defined in the basic sections have been followed carefully.

Audio Internals

An application program doesn't normally access the audio hardware directly. All data being recorded or played back is stored in a kernel DMA buffer while the device is accessing it. The application uses normal `read` and `write` calls to transfer data between the kernel buffer and the buffer in the application's data segment.

The audio driver uses an improved version of the so called double buffering method. In the basic double buffering method there are two buffers. One of them is being accessed by the device while the other is being read or written by the application. When the device finishes processing the first buffer, it moves to the other one. This process is repeated as long as the device is in use. This method gives the application time to do some processing at the same time as the device is running. This makes it possible to record and play back without pauses.

The amount of time the application can spend on processing the buffer half depends on the buffer size and the data rate. For example, when a program is recording audio using 8 kHz/8-bit/mono sampling, the data rate is 8 kilobytes/second. If there is 2*4 kilobytes of buffer, it gives the application more than 0.5 seconds of time to store the data to disk and to come back to read from the device. If it spends more than 0.5 seconds, the buffer overruns and the driver has to discard some data. 0.5 seconds is adequate time to store 4K of data to disk. However, things become more complicated when the data rate is increased. For example, with audio CD quality the data rate is 172 kilobytes/second and the available time is just 23 milliseconds. This is about the same as the worst case seek time of normal disk drives, which means that recording is likely to fail. Better results can be achieved by using larger buffers, but it increases latencies related to the buffering.

The method used by the audio driver of OSS could be called multi-buffering. In this method the available buffer space is divided into several equally sized blocks known as fragments. In this way it is possible to increase the available buffer size without increasing the latencies related to the buffering. By default, the driver computes the fragment size so that latencies are about 0.5 seconds

(for output) and about 0.1 seconds (for input) using the current data rate. There is an `ioctl` call for adjusting the fragment size in the case that the application wants to use a different size.

Normal Operation When Writing to the Device

When the program calls write the first time after opening the device, the driver performs the following steps:

1.   Programs the audio hardware to use the sampling parameters (speed, channels and bits) the program has selected.

2.   Computes a suitable size for a buffer fragment (only if the program hasn't requested a specific fragment size explicitly).

3.   Starts filling the first buffer fragment with the data written by the application.

4.   If enough data was written to fill the first fragment completely, the device is started to play it.

5.   Finally, the driver copies rest of the data to the buffer. If all buffer fragments have been used, the application is put to sleep until the first buffer gets played completely.

NOTE

At this point it is possible that the device was not started to play the data. This happens if the application doesn't write enough data to fill one buffer fragment completely. There is no reason to worry about this if the application is going to write more data to the device as soon as it can, or if it closes the device immediately. However (only) if there is going to be a pause of arbitrary length, the application should call the `ioctl SNDCTL_DSP_POST` to activate the playback.

When the application calls `write` a second time, the data is simply stored in the playback buffer and the internal pointers of the driver are updated accordingly. If the application has attempted to write more data than there is currently free space for in the buffer, it will be forced to wait until one fragment gets completely played by the device. This is the normal situation with programs that work properly. They usually write data at least slightly faster than the device plays it. Sooner or later they get the buffer completely filled and the driver forces them to work at the same speed as the device.

A playback underrun situation occurs when the application fails to write more data before the device gets earlier data completely played. This kind of underrun occurs for one of three reasons.

1.   The application needs too much time for processing the data. For example, the program is being run on a slow CPU or there are many other applications using the processor. Also, loading audio data from a floppy disk is likely to fail. It is usually very difficult if not impossible to find a solution to this kind of underrun problem. Possibly only rewriting parts

of the program in assembly language could help.

2.   There are slight variations in the amount of CPU time the application gets. In this way an application which normally works fast enough may randomly run out of time.

3.   The application attempts to work too much in real time. Having less data in the output buffer decreases delays in games and other real time applications. However, the application must take care that it always writes new data before earlier written samples get completely played.

The effect of underrun depends on the audio device. However, in almost every case an audible defect is caused in the playback signal. This may be just a short pause, a click or a repeated section of signal. Repeated underruns may cause very strange effects. For example 100 underruns per second sometimes causes a signal having a frequency of 100 Hz (it could be very difficult to find the reason which causes this effect).

Normal Operation When Reading from the Device

When the program calls read the first time after opening the device, the driver performs the following steps:

1.   Programs the audio hardware to use the sampling parameters (speed, channels and bits) the program has selected.

2.   Computes a suitable size for a buffer fragment (only if the program doesn't have requested specific fragment size explicitly).

3.   Activates the recording process on the device.

4.   Puts the application to sleep until the first fragment of data gets recorded by the device. Note that the application will wait until the whole fragment gets recorded even if it attempted to read just one byte.

5.   After recording of the first fragment is ready, it's contents, up to the amount requested by the application, will be copied to the application's buffer variable.

6.   The read call returns after all bytes requested by the application have been read. If there is more data in the driver's buffer, it is left there.

Subsequent reads work just like the first one except that the device doesn't need to be started again.

A recording overrun situation occurs if the device fills the recording buffer completely. If this happens, the device is stopped and further samples being recorded will be discarded. Possible reasons for recording overruns are very similar to the causes of playback underruns. A very common

situation where playback overrun may occur is recording of high speed audio directly to disk. In Linux this doesn't work except with very fast disk drives (in other environments this should not be a problem).

Buffering - Improving Real-Time Performance

Normally programs don't need to care about the buffering parameters of audio devices. However, most of the features presented in this document have been designed to work with full fragments. For this reason your program may work better if it reads and writes data one buffer fragment at time (please note that this is not normally required).

Determining Buffering Parameters

The driver computes the optimum fragment size automatically depending on sampling parameters (speed, bits and number of channels) and amount of available memory. Application may ask the buffer size by using the following `ioctl` call:

```
int frag_size;
if (ioctl(audio_fd, SNDCTL_DSP_GETBLKSIZE, &frag_size) == -1)
    error();
```

NOTE

This `ioctl` call also computes the fragment size, in case it has not already been done. For this reason you should call it only after setting sampling parameters or setting fragment size explicitly.

The fragment size in bytes is returned in the `frag_size` argument. The application may use this value as the size when allocating (malloc) a buffer for audio data and the count when reading from or writing to the device.

NOTE

Some old audio applications written for Linux check that the returned fragment size is between arbitrary limits (this was necessary with version 0.1 of the driver). New applications should not make this kind of test.

The above call returns the static fragment size. There are two additional calls which return information about the live situation.

```
audio_buf_info info;
ioctl(audio_fd, SNDCTL_DSP_GETISPACE, &info);
ioctl(audio_fd, SNDCTL_DSP_GETOSPACE, &info);
```

The above calls return information about output and input buffering, respectively. The

`audio_buf_info` record contains the following fields:

```
int fragments;
```

Number of full fragments that can be read or written without blocking. Note that this field is reliable only when the application reads/writes full fragments at time.

```
int fragstotal;
```

Total number of fragments allocated for buffering.

```
int fragsize;
```

Size of a fragment in bytes. This is the same value than returned by `ioctl(SNDCTL_DSP_GETBLKSIZE)`.

```
int bytes;
```

Number of bytes that can be read or written immediately without blocking.

These two calls, together with `select`, can be used for writing asynchronous or non-blocking applications. It is important that `SNDCTL_DSP_GETBLKSIZE` be the last `ioctl` call made before the first read or write. This call (as well as read or write) will perform some optimizations which require that the sampling parameters to be used are known. Changing the rate, format, or numbers of channels may cause an error condition with some hardware devices.

It is not recommended to use the SNDCTL_DSP_GETISPACE and SNDCTL_DSP_GETOSPACE functions for obtaining exact synchronization between audio and graphics or other external events. The values returned by these calls are tuned for preventing blocking anf they may in some situations behave unexpectedly. The new SNDCLT_DSP_GETODELAY call should be used for synchronization purposes instead.

Selecting Buffering Parameters

In some cases it may be desirable to select the fragment size explicitly. For example, in real time applications such as games, it is necessary to use relatively short fragments. Otherwise, delays between events on the screen and their associated sound effects become too long. The OSS API contains an `ioctl` call for setting the fragment size and maximum number of fragments:

```
int arg = 0xMMMMSSSS;
if (ioctl(audio_fd, SNDCTL_DSP_SETFRAGMENT, &arg))
    error();
```

The argument to this call is an integer encoded as 0x*MMMMSSSS* (in hex). The 16 least significant bits determine the fragment size. The size is 2^SSSS. For example SSSS=0008 gives fragment size of 256 bytes (2^8). The minimum is 16 bytes (SSSS=4) and the maximum is total_buffer_size/2. Some devices or processor architectures may require larger fragments - in this case the requested fragment size is automatically increased.

The 16 most significant bits (MMMM) determine the maximum number of fragments. By default, the driver computes this based on available buffer space. The minimum value is 2 and the maximum depends on the situation. Set MMMM=0x7fff if you don't want to limit the number of fragments.

The value returned by SNDCTL_DSP_SETFRAGMENT doesn't guarantee that the buffering parameters requested by the application will be used. The parameters to be used will be selected later based on the sampling parameters selected and the capabilities and limitations of the hardware being used. The only way to find out the actual values being used is by calling SNDCTL_DSP_GETBLKSIZE.

Obtaining Buffering Information (pointers)

In some cases it is necessary for an application to know exactly how much data has been played or recorded. The OSS API provides two `ioctl` calls for these purposes. The information returned by these calls is not precise in all cases. Some sound devices use internal buffering which make the returned pointer value very imprecise. In addition, some operating systems don't allow obtaining the value of the actual DMA pointer. Using these calls in applications is likely to make it non-portable between operating systems and makes them incompatible with many popular devices (such as the original Gravis UltraSound). Applications should use `ioctl(SNDCTL_DSP_GETCAPS)` to check device capabilities before using these calls.

```
count_info info;
```

94

```
ioctl(audio_fd, SNDCTL_DSP_GETIPTR, &info);
ioctl(audio_fd, SNDCTL_DSP_GETOPTR, &info);
```

These calls return information about recording and playback pointers, respectively. The count_info structure contains the following fields:

```
int bytes;
```

Number of bytes processed since opening the device. This field divided by the number of bytes/sample can be used as a precise timer. However underruns, overruns and calls to some `ioctl` calls (SNDCTL_DSP_RESET, SNDCTL_DSP_POST and SNDCTL_DSP_SYNC) decrease precision of the value. Also, some operating systems don't permit reading value of the actual DMA pointer so in these cases the value is truncated to previous fragment boundary. The value returned will wrap unpredictably just before every hour or playback or recording has elapsed. For this reason, these calls should not be used for audio and graphics synchronization purposes (unless for less than one hour duration). Use the new SNDCTL_DSP_GETODELAY function instead.

```
int blocks;
```

Number of fragment transitions (hardware interrupts) processed since the previous call to this `ioctl` (the value is reset to 0 after each call). This field is valid only when using direct access to audio buffer.

```
int ptr;
```

This field is the byte offset of the current playback/recording position from the beginning of the audio buffer. This field has little value except when using direct access to an audio buffer.

Non-Blocking Reads and Writes

All audio read and write calls are non-blocking as long as there is enough space/data in the buffer when the application makes the call. The application may use `SNDCTL_DSP_GETOSPACE` and `SNDCTL_DSP_GETISPACE` to check the device's status before making the call. The bytes field tells how many bytes can be read or written without blocking. It is highly recommended to read and write full fragments every time when using select.

Using `select`

The OSS driver supports standard `select` system call. With audio devices, `select` returns 1 in the read or write descriptor bit mask when it is possible to read or write at least one byte without blocking. The application should use `SNDCTL_DSP_GETOSPACE` and `SNDCTL_DSP_GETISPACE` to check the actual situation. Reading and writing full fragments at a time is recommended when `select` is used.

Calling `select` with the `audio_fd` bit set in the readfds parameter has an important side effect. This call starts recording immediately if it has not already started and recording is enabled. (Due to a bug in OSS versions earlier than 3.6 this may not work with all cards.)

Some operating systems (such as Solaris) don't support `select`. In this case the `poll` system call can be used instead.

Checking Device Capabilities

There are some features in the OSS API that don't work with all devices and/or operating systems. For this reason it is important to check that the features are available before trying to use them. The result of using features not supported by the current hardware/operating system combination is undefined.

It is possible to check the availability of certain features by using the `SNDCTL_DSP_GETCAPS` `ioctl` as below:

```
int caps;
ioctl(audio_fd, SNDCTL_DSP_GETCAPS, &caps);
```

This call returns a bit mask defining the available features. The possible bits are:

`DSP_CAP_REVISION` - the 8 least significant bits of the returned bit mask is the version number of this call. In the current version it is 0. This field is reserved for future use.

`DSP_CAP_DUPLEX` - tells if the device has full duplex capability. If this bit is not set, the device supports only half duplex (recording and playback is not possible at the same time).

`DSP_CAP_REALTIME` - tells if the device/operating system supports precise reporting of output pointer position using `SNDCTL_DSP_GETxPTR`. Precise means that accuracy of the reported playback pointer (time) is within a few samples. Without this capability the playback/recording position is reported using precision of one fragment.

`DSP_CAP_BATCH` - indicates that the device has some kind of local storage for recording and/or playback. For this reason the information reported by SNDCTL_DSP_GETxPTR is very inaccurate.

`DSP_CAP_COPROC` - means that there is some kind of programmable processor or DSP chip associated with this device. This bit is currently undefined and reserved for future use.

`DSP_CAP_TRIGGER` - tells that triggering of recording/playback is possible with this device.

`DSP_CAP_MMAP` - tells if it is possible to get direct access to the hardware level recording and/or playback buffer of the device.

Synchronization Issues

In some applications it is necessary to synchronize audio playback/recording with screen updates, MIDI playback, or some other external events. This section describes some ways to implement this kind of feature. When using the features described in this section it is very important to access the device by writing and reading full fragments at time. Using partial fragments is possible but it may introduce problems which are very difficult to handle.

There are several different reasons for using synchronization:

1.   The application should be able to work without blocking in audio reads or writes.

2.   There is a need to keep external events in sync with audio (or to keep audio in sync with external events).

3.   Audio playback and recording needs to be done in sync.

It is also possible to have several of the above goals at the same time.

Avoiding Blocking in Audio Operations

The recommended method for implementing non-blocking reads or writes is to use `select`. Further instructions for using this method have been given above.

Synchronizing External Events With Audio

When there is a need to get audio recording and playback to work in sync with screen updates, it is easier to play the audio at its own speed and to synchronize screen updates with it. To do this, you can use the `SNDCTL_DSP_GETxPTR` calls to obtain the number of bytes that have been processed since opening the device. Then divide the bytes field returned by the call by the number of bytes per sample (for example 4 in 16-bit stereo mode). To get the number of milliseconds since start, you need to multiply the sample count by 1000 and to divide this by the sampling rate.

In this way you can use normal UNIX alarm timers or select to control the interval between screen updates while still being able to obtain exact audio time. Note that any kind of performance problems (playback underruns and recording overruns) disturb audio timing and decrease it's precision.

Recent versions of OSS support the new SNDCTL_DSP_GETODELAY function. It accepts a parameter that points to an integer variable. The call returns the number of unplayed bytes in the kernel buffer (the precision varies between a few samples and one fragment depending on the hardware capabilities). The return value can be used to compute the time before the next sample written to the device will be played. The advantage of this call us that the value will not overflow

or wrap after a period of time, unlike the bytes parameter returned by the SNDCTL_DSP_GETOPTR call.

This synchronization strategy is probably only useful when doing playback. When recording , use approach described in the next section.

Synchronizing Audio With External Events

In games and some other real time applications there is a need to keep sound effects playing at the same time as game events. For example, the sound of an explosion should be played exactly at the time (or slightly later) as the flash on the screen.

The recommended method to be used in this case is to decrease the fragment size and maximum number of fragments used with the device. In most cases this kind of application work best with just 2 or 3 fragments. A suitable fragment size can be determined by dividing the byte rate of audio playback by the number of frames/second to be displayed by the game. It is recommended to avoid too tight timing since otherwise random performance problems may seriously degrade audio output.

Another way to synchronize audio playback with other events is to use direct access to audio device buffer. However, this method is not recommended since it is not possible on all devices and operating systems.

When using the methods described above, there may be a need to start playback and/or recording precisely at the right time. This is possible by using the trigger feature described below.

Synchronizing Recording and Playback Together

In full duplex applications it may be necessary to keep audio playback and recording synchronized together. For example, it may be necessary to play back earlier recorded material at the same time as recording new audio tracks. Note that this kind of application is possible only with devices supporting full duplex operation or by using two separate audio devices together. In the second case it is important that both devices support precisely the sampling rate to be used (otherwise synchronization is not possible). Use the trigger feature when you need this kind of synchronization.

Implementing Real-Time Effect Processors and other Oddities

Here the term "real-time" means an application which records audio data, performs some kind of processing on it, and outputs it immediately without any noticeable delay. Unfortunately, this kind of applications in general is not possible using UNIX-like multitasking operating systems and general purpose computer hardware. There is always some delay between recording a sample and before it is available for processing by the application (the same is true with playback too). In addition, the multitasking overhead (other simultaneously running processes) causes unexpected pauses in operation of the application itself. Normally these kinds of operations are done with

dedicated hardware and system software designed for this kind of use.

It is possible to decrease the delay between input and output by decreasing the fragment size. In theory, the fragment size can be as short as 16 bytes with a fast machine. However, in practice it is difficult to get fragment sizes shorter than 128 to 256 bytes to work. Using direct access to the hardware level audio buffer may provide better results in systems where this feature works.

If you still want to implement this kind of application, you should use short fragments together with select. The shortest fragment size that works depends on the situation and the only way to find it out is making some experiments. And, of course, you should use a device with full duplex capability or two separate devices together.

Usually read/write type real-time full duplex applications require that one fragment of silence data is written to the output device immediately prior to writing the first recorded data to it. This extra data causes some unwanted delay but without it the application (and operating system) has practically no time to do it's own processing. It's important that this silent data is written after the first read is complete, because otherwise playback may start too early.

It should be noted that in general it's not possible to use two or more sound cards in perfect synchronization. Two devices that are not explicitly synchronized together will never work exactly at the same sampling rate. For this reason, there will be some drift between the two sound cards. Eventually, after enough time has elapsed, this will cause problems (the time could be from seconds to hours).

For the above reason, effect processing and multi-track recording may work only when using a single full duplex capable soundcard or a proper multi channel device. There are also devices that may be synchronized together using a special cable which solves this problem.

A similar problem may happen when working with ISDN connections. The ISDN data rate is exactly 8K bytes/sec but not all sound cards are able to work at exactly the 8 kHz rate.

Starting Audio Playback and/or Recording with Precise Timing

The `SNDCTL_DSP_SETTRIGGER` ioctl call has been designed to be used in applications which require starting recording and/or playback with precise timing. Before you use this `ioctl`, you should check that the `DSP_CAP_TRIGGER` feature is supported by the device. Trying to use this `ioctl` with a device not supporting it will give undefined results.

This `ioctl` accepts an integer parameter where two bits are used to enable and disable playback, recording or both. The `PCM_ENABLE_INPUT` it controls recording and `PCM_ENABLE_OUTPUT` controls playback. These its can be used together, provided that the device supports full duplex and the device has been opened for `O_RDWR` access. In other cases the application should use only one of these bits without reopening the device.

The driver maintains these bits for each audio device which supports his feature. Initially, after open, these bits are set to 1 which makes the device work normally.

Before the application can use the trigger `ioctl` to start device operations, the bit to be used should be set to 0. To do this you can use the following code. It is important to note that this can be done only immediately after opening the device (before writing to or reading from it). It is currently not possible to stop or restart a device that has already been active without first reopening the device file.

```
int enable_bits = ~PCM_ENABLE_OUTPUT; /* This disables playback */
ioctl(audiofd, SNDCTL_DSP_SETTRIGGER, &enable_bits);
```

After the above call writes to the device, don't start the actual device operation. The application can fill the audio buffer by outputting data using `write`. Write will return -1 with `errno` set to EAGAIN if the application tries to write when the buffer is full. This permits preloading the buffer with output data in advance. Calling read when `PCM_ENABLE_INPUT` is not set will always return `EAGAIN`.

To actually activate the operation, call `SNDCTL_DSP_TRIGGER` with the appropriate bits set. This will start the enabled operations immediately (provided that there is already data in the output buffer). It is also possible to leave one of the directions disabled while starting another one.

Starting Audio Recording or Playback in Sync with `/dev/sequencer` or `/dev/music`

In some cases it is necessary to synchronize playback of audio sequences with MIDI output (this is possible with recording too). To do this you need to suspend the device before writing to or reading from it. This can be done by calling `ioctl(audiofd, SNDCTL_DSP_SETSYNCRO, 0)`. After this, the device works just like when both the recording and the playback trigger bits (see above) have been set to 0. The difference is that it is not possible to reactivate the device without using features of `/dev/sequencer` or `/dev/music` (`SEQ_PLAYAUDIO` event).

Full Duplex Mode

Full duplex means an audio device has the ability to do input and output in parallel.

Most audio devices are half duplex, which means that they support both recording and playback but can't do them simultaneously due to hardware level limitations (some devices can't do recording at all). In this case it is very difficult to implement applications which do both recording and playback. It is recommended that the device is reopened when switching between recording and playback.

It is possible to get full duplex features by using two separate devices. In the context of OSS this is not called full duplex but simultaneous use of two devices.

Full duplex does not mean that the same device can be used twice. With the current OSS

implementation it is not possible to open a device that is already open. This feature can possibly be implemented in future versions. In this situation you will need to use two separate devices.

Some applications require full duplex operation. It is important that such applications verify that full duplex is possible (using `DSP_CAP_DUPLEX`) before trying to use the device. Otherwise, the behaviour of the application will be unpredictable.

Applications should switch the full duplex feature on immediately after opening the device using `ioctl(audiofd, SNDCTL_DSP_SETDUPLEX, 0)`. This call switches the device to full duplex mode and makes the driver prepared for full duplex access. This must be done before checking the `DSP_CAP_DUPLEX` bit, since otherwise the driver may report that the device doesn't support full duplex.

Using full duplex is simple in theory. The application just:

1.   Opens the device
2.   Turns on full duplex
3.   Sets the fragment size if necessary
4.   Sets the number of channels, sample format, and sampling rate
5.   Starts reading and writing the device

In practice, it is not that simple. The application should be able to handle both input and output correctly without blocking on writes and reads. This almost certainly means that the application must be implemented to use the synchronization methods described earlier.

Accessing the DMA Buffer Directly

In some rare cases it is possible to map audio device's hardware level buffer area into the address space of an application. This method is very operating system dependent and is currently only supported on the Linux platform. In general, this feature should be avoided if possible. It doesn't work with all hardware. Contact 4Front Technologies for assistance if there is no other way to implement your application.

The direct mapping method is possible only with devices that have a hardware level buffer which is directly accessible from the host CPU's address space (for example, a DMA buffer or a shared memory area).

The basic idea is simple. The application uses an operating system dependent method to map the input or the output buffer into it's own virtual address space. In the case of full duplex devices, there are two separate buffers (one for input and one for output). After that, it triggers the desired transfer operation(s). Then, the buffer will be continuously accessed by the hardware until the device is closed. The application can access the buffer area(s) using pointers, but normal read and write calls can no longer be used.

The buffer area is continuously scanned by the hardware. When the pointer reaches the end of the buffer, the pointer is moved back to the beginning. The application can read and write the data using the `SNDCTL_DSP_GETxPTR` calls. The bytes field tells how many bytes the device has processed since beginning. The ptr field gives an offset relative to the beginning of the buffer. This pointer must be aligned to the nearest sample boundary before accessing the buffer using it. The pointer returned by this call is not absolutely precise due to possible delays in executing the `ioctl` call and possible FIFOs inside the hardware device itself. For this reason, the application should assume that the actual pointer is a few samples ahead of the returned value.

When using direct access, the blocks field returned by the `SNDCTL_DSP_GETxPTR` calls has special meaning. The value returned in this field is the number of fragments that have been processed since the previous call to the same `ioctl` (the counter is cleared after the call).

Also, select works in a special way with mapped access. Select returns a bit in the readfds or writefds parameter after each interrupt generated by the device. This happens when the pointer moves from a buffer fragment to another. However, the application should check the actual pointer very carefully. It is possible that the select call returns a relatively long time after the interrupt. It is even possible that another interrupt occurs before the application gets control again.

Note that the playback buffer is never cleaned by the driver. If the application stops updating the buffer, its present contents will be played in a loop again and again. Sufficient play-ahead is recommended, since otherwise the device may play uninitialized (old) samples if there are any performance problems.

No software based sample format conversions are performed by the driver. For this reason the application must use a sample format that is directly supported by the driver.

Platform Specific Issues

In general, all sound/audio programs written for Linux, FreeBSD, SCO or UnixWare use the same OSS API (or its older version called VoxWare). The sound related functionality of these programs is portable between operating systems where OSS is available. However, there are some common portability problems.

Programs that use OS specific libraries or features are not portable or they require some changes before they work.

Some programs include `<soundcard.h>` in a nonstandard way such as `<linux/soundcard.h>` or `<machine/soundcard.h>`. You should instead use `<sys/soundcard.h>`.

Many 16-bit audio programs assume that they are running on a little-endian (x86) machine. This causes problems (e.g. noise) in big-endian RISC machines such as PowerPC, SPARC or HP-PA.

Appendix A - References

The full MIDI definition is found in *The Complete MIDI 1.0 Detailed Specification*, published by the MIDI Manufacturers Association. More information can be found at **http://www.midi.org**.

Information on installing and configuring OSS can be found in the *Open Sound System Installation Guide*.

Listed here are just a few web resources related to sound and multimedia:

http://www.opensound.com

The 4Front Technologies web site has a list of multimedia applications that support OSS, as well as a "killer app" featured each month.

http://www.linuxdoc.org

The Linux Documentation Project has created many HOWTO documents for Linux, including the CD-ROM and Sound HOWTOs. It also includes several freely available books. Many of these are installed on Linux systems in the `/usr/doc` directory.

http://sound.condorow.net

This web site has a comprehensive and up to date list of Linux and UNIX MIDI and sound applications.

http://www.freshmeat.net

This site is a central clearing house for Linux applications, both a large searchable database, as well as announcements of new releases.

Several published books have some coverage of sound support, most notably *Linux Multimedia Guide* published by O'Reilly & Associates. Usenet newsgroups, mailing lists, and local user groups are also a good source of answers to problems related to sound support.

Appendix B - General MIDI patch map

Note that, as per the MIDI spec, program numbers listed here start at one but are zero based in MIDI messages. Some MIDI applications may display the zero-based numbers.

Table 12 - General MIDI Sound Set (all channels except 10)

#	Instrument	#	Instrument	#	Instrument
1	Acoustic Grand Piano	44	Contrabass	87	Lead 7 - Fifths
2	Bright Acoustic Piano	45	Tremolo Strings	88	Lead 8 - Bass+Lead
3	Electric Grand Piano	46	Pizzicato Strings	89	Pad 1 - New Age
4	Honky-Tonk	47	Orchestral Harp	90	Pad 2 - Warm
5	Rhodes Piano	48	Timpani	91	Pad 3 - Polysynth
6	Chorused Piano	49	String Ensemble 1	92	Pad 4 - Choir
7	Harpsicord	50	String Ensemble 2	93	Pad 5 - Bow
8	Clavinet	51	Synth Strings 1	94	Pad 6 - Metallic
9	Celesta	52	Synth Strings 2	95	Pad 7 - Halo
10	Glockenspiel	53	Choir Aahs	96	Pad 8 - Sweep
11	Music Box	54	Voice Oohs	97	FX 1 - Rain
12	Vibraphone	55	Synth Voice	98	FX 2 - Soundtrack
13	Marimba	56	Orchestra Hit	99	FX 3 - Crystal
14	Xylophone	57	Trumpet	100	FX 4 - Atmosphere
15	Tubular Bells	58	Trombone	101	FX 5 - Brightness
16	Dulcimer	59	Tuba	102	FX 6 - Goblins
17	Hammond Organ	60	Muted Trumpet	103	FX 7 - Echoes
18	Percussive Organ	61	French Horn	104	FX 8 - Sci-fi
19	Rock Organ	62	Brass Section	105	Sitar
20	Church Organ	63	Synth Brass 1	106	Banjo
21	Reed Organ	64	Synth Brass 2	107	Shamisen
22	Accordion	65	Soprano Sax	108	Koto

23	Harmonica	66	Alto Sax	109	Kalimba
24	Tango Accordion	67	Tenor Sax	110	Bagpipe
25	Acoustic Guitar (Nylon)	68	Baritone Sax	111	Fiddle
26	Acoustic Guitar (Steel)	69	Oboe	112	Shannai
27	Electric Guitar (Jazz)	70	English Horn	113	Tinkle Bell
28	Electric Guitar (Clean)	71	Bassoon	114	Agogo
29	Electric Guitar (Muted)	72	Clarinet	115	Steel Drum
30	Overdriven Guitar	73	Piccolo	116	Wook Block
31	Distortion Guitar	74	Flute	117	Taiko Drum
32	Guitar Harmonics	75	Recorder	118	Melodic Tom
33	Acoustic Bass	76	Pan Flute	119	Synth Drum
34	Electric Bass (Finger)	77	Blown Bottle	120	Reverse Cymbal
35	Electric Bass (Pick)	78	Shakuhachi	121	Guitar Fret Noise
36	Fretless Bass	79	Whistle	122	Breath Noise
37	Slap Bass 1	80	Ocarina	123	Seashore
38	Slap Bass 2	81	Lead 1 - Square Wave	124	Bird Tweet
39	Synth Bass 1	82	Lead 2 - Saw Tooth	125	Telephone
40	Synth Bass 2	83	Lead 3 - Calliope	126	Helicopter
41	Violin	84	Lead 4 - Chiflead	127	Applause
42	Viola	85	Lead 5 - Charang	128	Gunshot
43	Cello	86	Lead 6 - Voice		

Table 13 - General MIDI Percussion Map (Channel 10)

#	Instrument	#	Instrument	#	Instrument
35	Acoustic Bass Drum	51	Ride Cymbal 1	67	High Agogo
36	Bass Drum 1	52	Chinese Cymbal	68	Agogo Low
37	Side Stick	53	Ride Bell	69	Cabasa

38	Acoustic Snare	54	Tambourine	70	Maracas
39	Hand Clap	55	Splash Cymbal	71	Short Whistle
40	Electric Snare	56	Cowbell	72	Long Whistle
41	Low Floor Tom	57	Crash Cymbal 2	73	Short Guiro
42	Closed Hi Hat	58	Vibraslap	74	Long Guiro
43	High Floor Tom	59	Ride Cymbal 2	75	Claves
44	Pedal Hi Hat	60	Hi Bongo	76	Hi Wood Block
45	Low Tom	61	Low Bongo	77	Low Wood Block
46	Open HiHat	62	Mute Hi Conga	78	Mute Cuica
47	Low-Mid Tom	63	Open High Conga	79	Open Cuica
48	Hi-Mid Tom	64	Low Conga	80	Mute Triangle
49	Crash Cymbal 1	65	High Timbale	81	Open Triangle
50	High Tom	66	Low Timbale		

Appendix C - FM Synthesizer Interface

This section describes the `/dev/sequencer` interface. This interface is now obsolete and has been replaced by the `/dev/music` device.

The `/dev/sequencer` device is used for producing musical type sounds. It can be used to control an FM sound chip (OPL-3 or OPL-4), a wavetable sound card (e.g. GUS), MIDI devices, and other compatible devices (like the SoftOSS software wavetable device). While you can do similar things with the `/dev/music` and `/dev/midi` devices, this one gives you the most control for on-board sound devices.

It is used in a manner vaguely similar to MIDI, you send events to the driver, the events are put in a queue and executed in the background. If you fill the queue, your process will block until the queue is not full. The low "water mark" queue threshold is configurable (the default is half the queue) and settable via the `SNDCTL_SEQ_THRESHOLD ioctl`.

You can avoid blocking by opening in non blocking mode, in which case it will fail with return code `EAGAIN`. If the queue empties, playing stops until more events sent. The queue is quite large (1024 events).

There is a low-level interface, but you will normally make use of the macros defined in `<soundcard.h>` to make programming more convenient (and less likely to break in future). Note that the interface does not run on real-time! You put timing information in the messages and the sounds are played in the background.

The normal way in which the interface is used is the following:

1.   Set buffer size with SEQ_DEFINEBUF()
2.   Define file descriptor, seqfd
3.   Implement buffer writing routine called seqbuf_dump (the example code shown below should work fine, possibly with different error handling)

Listing 13 - Sample Implementation of Buffer Writing Routine

```
void seqbuf_dump ()
{
    if (_seqbufptr)
        if (write (seqfd, _seqbuf, _seqbufptr) == -1) {
            perror ("write /dev/sequencer");
            exit (-1);
        }
    _seqbufptr = 0;
}
```

4.   Open the device (in most cases for write only)
5.   Load instrument patches (if internal sound card)
6.   Set patches for each voice

7.   Start timer (starts when device opened)
8.   Play notes with SEQ_START_NOTE
9.   Timing info with SEQ_DELTA_TIME or SEQ_WAIT_TIME
10.   Stop notes with SEQ_STOP_NOTE
11.   Use other events for various effects
12.   When done call SEQ_DUMPBUF() to flush the buffer

The event commands defined in the header file start with "SEQ_". The sequencer specific `ioctl` functions start with the prefix "SNDCTL_SEQ_".

Patches for the sound devices vary by the type of device. Defined types are FM_PATCH, OPL3_PATCH, WAVE_PATCH, GUS_PATCH, and WAVEFRONT_PATCH. The are often obtained from patch files. The files `/etc/std.o3` and `/etc/drums.o3` are FM patch files for General MIDI. The files `/etc/std.sb` and `/etc/drums.sb` are SBI file format patches. You may also have .sbi patch files for individual voices. Files with extension .pat are patch files for GUS cards. Patches are written to the device using the macros SEQ_WR_PATCH for SEQ_WRPATCH2.

Don't assume a clock rate, you can check it with SNDCTL_SEQ_RATE (it usually 100 ticks per second).

Each voice can only play one sound at a time. If told to play a note any previous one stops. Application needs to handle switching voices if you want polyphony. Like MIDI, channel 10 (9 zero-based) is the percussion channel.

You can get the current time using SNDCTL_SEQ_GETTIME `ioctl`. Given this you can synchronize other things with the playing of the music. It is also possible receive events using SEQ_ECHO_BACK which will be synchronized with the playing.

Note that you can play `/dev/sequencer` independently of `/dev/dsp`. Some programs, e.g. games, use `/dev/dsp` for sound effects and `/dev/sequencer` for music

For an example of using the devices, read the source code for applications such as `playmidi`.

Glossary of Terms

A-law - a logarithmic coding scheme that uses companding to compress 12 bit samples into 8 bits. Used primarily in European digital telephone systems.

ADC - analog to digital converter. A device that converts continuously variable analog values (such as sound pressure measured by a microphone) to discrete digital samples.

ADPCM - Adaptive Delta Pulse Code Modulation. A digital encoding scheme developed by the Interactive Multimedia Association.

API - Application Programming Interface. The set of functions, constants, and variables provided by a software application, library, or device driver.

Bitmask - a bit pattern used to isolate specific bits in a data representation. Often used in conjunction with binary boolean operation such as AND and OR.

Codec - encoder/decoder; a method of coding and decoding data from one format to another. May be implemented in hardware or software.

DAC - digital to analog converter. A device that converts discrete digital samples to continuously variable analog values (such as sound produced by a speaker).

DAT - Digital Audio Tape. A standard for storing digital information on magnetic tape.

DMA - Direct Memory Access. A hardware feature whereby data can be transferred directly between peripheral devices and main memory without the intervention of the CPU.

DSP - Digital Signal Processor. Strictly speaking refers to a processor chip designed for signal processing applications, but often used informally to refer to the ADC and DAC devices on a sound card.

Decibel (dB) - a unit of measurement based on a logarithmic scale. Often used for measuring sound intensity, since the human ear has a logarithmic response.

Digitize - to convert from analog to digital form.

Dynamic Range - the difference between the weakest and loudest values that can be represented. For a perfect analog to digital converter, the dynamic range is approximately 6 dB times the numbers of bits used for sampling.

Endian - The endian convention of a processor refers to the way in which multi-byte numbers are stored in memory. Little-endian systems store the least significant bytes at lower memory address

which big-endian systems use the opposite convention.

FM Synthesis - Frequency Modulation synthesis; a method of sound generation that uses waveform generators and modulators in combination to produce sound.

Full Duplex - in the context of a sound card, refers to the ability to both record and play back simultaneously.

GM - see General MIDI

GUS - Gravis UltraSound; a manufacturer of sound cards.

General MIDI - an extension to the MIDI standard which improves compatibility by defining a minimum set of capabilities and standardized sound sets.

Half Duplex - in the context of a sound card, refers to the limitation that a device can either record or play back, but not both simultaneously.

IMA - Interactive Multimedia Association. A body which defines standards, such as the ADPCM encoding format.

ISDN - Integrated Services Digital Network. A series of ISO standards for voice and data services over digital telecommunications networks.

Ioctl - a system call used to control devices.

Line In/Line Out - a standardized physical and electrical interface for connecting analog audio devices together. Line level differs from microphone level and speaker level.

MIDI - Musical Instrument Digital Interface. A standardized protocol for conveying musical performance information as electronic data.

MMA - MIDI Manufacturers Association. A body which publishes the MIDI standard and promotes the use of MIDI and related technologies.

MOD - MODule file; a music file format that includes both sequencing information as well as sound samples. First popularized on the Amiga computer platform.

MPEG - Moving Pictures Experts Group, a body which sets standards for digital audio and video encoding. Also used informally to refer to the standards produced by the group.

MPU-401 - A de facto standard for a PC MIDI interface developed by Roland Corporation.

MSS - Microsoft Sound System; a (now obsolete) sound card.

Mic - microphone

Mixer - a device used to control sound input and output volume levels and switching of the input sources.

Mu-law - (-law); a logarithmic coding scheme that uses companding to compress 12 bit samples into 8 bits. Used primarily in North American digital telephone systems.

OPL-2 - An FM synthesizer chip developed by Yamaha. It provided 2 operators and 9 voices.

OPL-3 - An FM synthesizer chip developed by Yamaha. It offered improved sound quality of the OPL-2 chip by providing 4 operators and more voices.

OSS - Open Sound System, the multi-platform sound drivers sold by 4Front Technologies.

Operator - a waveform oscillator on an FM synthesizer chip used to produce sound. More operators allow more realistic sounds to be produced.

Overrun - an error condition in which incoming data cannot be read quickly enough, resulting in data loss.

PAS - ProAudio Spectrum, a manufacturer of sound cards.

Patch - In the context of sound generation, the device settings for a sound generator which produce a specific sound (i.e. acoustic piano). The settings are often permanently stored in files, known as patch files.

PCM - Pulse Code Modulation. An encoding scheme for representing audio in digital format.

SB - SoundBlaster. A series of sound cards developed by Creative Labs.

SBI - SoundBlaster Instrument. A file format developed by Creative Labs to define FM synthesizer settings (patches).

SMPTE - Society of Motion Picture Technicians and Engineers. A standards organization. Often used informally to refer to the time code standard developed by the SMPTE.

Sample Rate - that rate at which digital samples are measured or produced. Along with sample size, is one of the fundamental parameters which affects sound quality.

Sample Size - the size, usually expressed in bits, of digitized sound samples. Along with sample rate, is one of the fundamental parameters which affects sound quality.

Sequencer - a device (hardware or software) which controls (sequences) the playing of notes on a music synthesizer.

SoftOSS - an optional feature of OSS which implements wavetable synthesis on non-wavetable sound cards.

SysEx - System Exclusive Message. A class of MIDI system messages which are used to transfer information in a manufacturer dependent format.

Underrun - an error condition in which outgoing data is not available to be sent when required, usually resulting in data loss or noise.

Virtual Mixer - an optional feature of OSS which provides multiple virtual sound devices using only one physical device.

Voice - an independent sound generator.

Wavetable Synthesis - a method of sound generation that uses digital sound samples stored in dedicated memory.

Index