Python OSS Module

A Python interface to the Open Sound System API
July 1997

Timothy Butler

1 Introduction

The **oss** module provides access to the Open Sound System from the Python programming language. The Open Sound System is a UNIX device driver that allows you to record and play digitized audio, control input and output volume levels, select recording sources, and generate synthesizer commands and midi bytes controlled by a low-level sequencer. It runs on FreeBSD, Linux, BSD/OS, Solaris, and other operating systems.

Python is an interpreted, interactive, object-oriented, extensible, and way-cool programming language. An interactive programming environment extended with objects to control your sound card can be a powerful and fun combination.

1.1 Essential References

This manual does not cover all of the details you need to program the Open Sound System. You should consult the following resources to learn about the Open Sound System.

Open Sound System
: http://www.4front-tech.com/oss.html

Open Sound System (OSS) Free
: http://www.4front-tech.com/usslite/

OSS Programmer's Guide
: http://www.4front-tech.com/pguide/index.html

OSS documentation (contains link to valuable "Hackers Guide to VoxWare")
: http://www.4front-tech.com/usslite/docs.html

You can find more information about Python at

Python Home Page
: http://www.python.org

1.2 Obtaining the oss Module

You can download the oss Module from

```
http://www.indra.com/~tim/ossmodule
```

1.3 Installation

The **oss** module consists entirely of a single C language source file, `ossmodule.c`. You must either

1. compile and link the module into the Python interpreter or
2. create and install a shared library in a directory named by your `PYTHONPATH`.

1.3.1 Linking into the Interpreter

1. Copy 'ossmodule.c' into Python's 'Modules' directory.
```
> cp ossmodule.c Python-1.4/Modules
```
2. Edit 'Modules/Setup' and add an entry for the **oss** module.
```
oss ossmodule.c -I/usr/include/machine -I/usr/include/sys
```
   You may need to add or change the C preprocessor's search directories so it can find 'soundcard.h' on your system.

   You can also create a shared library this way if you place the module line in the `*shared*` section of the 'Setup' file. Read the comments in the 'Setup' file for details.
3. Recompile, test, and install the Python interpreter
```
> cd Python-1.4
> make

> ./python
>> import oss

>make bininstall
```

1.3.2 Building a Shared Library

You can use the 'Makefile' provided with the **oss** module distribution to compile a shared library. You will have to edit the 'Makefile' to match your system. Copy the resulting shared library to a directory named by your `PYTHONPATH`.

1.4 Feedback

Please email bug reports and comments concerning the **oss** module to `tim@netbox.com`.

I am not qualified to answer questions about the OSS itself. Part of the reason I wrote the **oss** module was to create an environment where I could experiment with and learn about the OSS.

If you can't get your sound card to work in the first place I can't help; but you might check out the resources listed above. See Section 1.1 [Essential References], page 1.

1.5 Design and Implementation

The C language header file, 'sys/soundcard.h', defines the OSS API. Most of the API consists of macros which define operations and arguments for the `ioctl()` system call.

The **oss** module implements only the bare minimum to use the API including:

Sound device objects.
: Represent devices such as a mixer, sequencer, and digital audio devices. The sound device objects in the **oss** module store a file descriptor and their methods typically involve little more than a call to `ioctl()` with the proper arguments.

Functions that open and return sound device objects.

Module-level variables holding macro "constants".
Bitmasks or device numbers that are ultimately provided as arguments to the `ioctl()` calls. Many module-level variables exist in the **oss** module which correspond to the exact same names of macro constants in '`soundcard.h`'. For example the value of `oss.SOUND_MIXER_NRDEVICES` is the same as `SOUND_MIXER_NRDEVICES` in '`soundcard.h`'.

"Structure" objects.
Hold read-only information about system capabilities.

The API operations, such as `SOUND_MIXER_READ_RECMASK`, typically translate into methods of sound device objects, such as `mixer.read_recmask()`. The correspondence should be clear from the documentation if not from the names.

None of the objects provided by the **oss** module can serve as base classes.

Not every name and data structure provided by '`soundcard.h`' is available through the **oss** module. The **oss** module provides only the interface that is described in the official OSS API documentation. Many parts of '`soundcard.h`' are either unsupported, obsolete, not portable, or undocumented.

The MIDI interface and the /dev/sndstat objects are not provided because these interfaces are simply read and write operations.

The **oss** module was developed using the OSSFree implementation under FreeBSD. Your mileage may vary.

1.6 Future Development

I would like to stick as close as possible to the OSS API as opposed to implementing old or undocumented features. So far, I have provided that part of the API provided implemented by OSSFree. I may add features of the commercial version as an option.

As a separate project I'd like to add a layer on top of the **oss** module (called "possum") that provides graphical interfaces to the sound devices. MIDI protocol support would be handy also.

2 Mixer Programming

2.1 Channel Identifiers

2.1.1 Channel Numbers

Mixer channels are identified by channel numbers stored in these module-level variables. Each channel number variable has a corresponding "MASK" variable has a single bit set shifted left by the channel number.

`SOUND_MIXER_NRDEVICES`
: number of channels known to OSS [0,30]

`SOUND_MIXER_VOLUME`
`SOUND_MASK_VOLUME`
: master output level

`SOUND_MIXER_BASS`
`SOUND_MASK_BASS`
: bass level of all output channels

`SOUND_MIXER_TREBLE`
`SOUND_MASK_TREBLE`
: treble level of all output channels

`SOUND_MIXER_SYNTH`
`SOUND_MASK_SYNTH`
: level of synthesizer (e.g. FM, wave-table)

`SOUND_MIXER_PCM`
`SOUND_MASK_PCM`
: output level for audio device (/dev/dsp, /dev/audio)

`SOUND_MIXER_SPEAKER`
`SOUND_MASK_SPEAKER`
: volume of PC speaker signal routed through card

`SOUND_MIXER_LINE`
`SOUND_MASK_LINE`
: level for line-in jack

`SOUND_MIXER_MIC`
`SOUND_MASK_MIC`
: input level from microphone jack

`SOUND_MIXER_CD`
`SOUND_MASK_CD`
: input level from CD

`SOUND_MIXER_IMIX`
`SOUND_MASK_IMIX`
: output (headphone jack) vol for rec monitor (rec only)

`SOUND_MIXER_ALTPCM`
`SOUND_MASK_ALTPCM`
: volume of alternative CODEC device

`SOUND_MIXER_RECLEV`
`SOUND_MASK_RECLEV`
: global recording level (rec only)

`SOUND_MIXER_IGAIN`
`SOUND_MASK_IGAIN`
: ??

`SOUND_MIXER_OGAIN`
`SOUND_MASK_OGAIN`
: ??

`SOUND_MIXER_LINE1`
`SOUND_MASK_LINE1`
: vendor specific

`SOUND_MIXER_LINE2`
`SOUND_MASK_LINE2`
: vendor specific

`SOUND_MIXER_LINE3`
`SOUND_MASK_LINE3`
: vendor specific

2.1.2 Channel Labels

Two module-level lists, `SOUND_DEVICE_LABELS` and `SOUND_DEVICE_NAMES`, store strings that can be used to name channels. These lists can be indexed by channel number (e.g. `SOUND_DEVICE_LABELS[SOUND_MIXER_VOLUME]`).

`SOUND_DEVICE_LABELS`
: channel names that are suitable for user presentation.

`SOUND_DEVICE_NAMES`
: lowercase fixed-size names without spaces.

2.2 mixer Objects

A mixer object controls the volume levels of input and output channels. A mixer object also selects input sources from the microphone, line, and CD inputs.

Read and write operations on a mixer's file descriptor are undefined and there are no `read()` or `write()` methods on a mixer.

2.2.1 mixer Instantiation

Create an mixer object using the module function `open_mixer()`. The same mixer device may be opened more than once. An open mixer object will be closed before it is deleted.

open_mixer (*filename, flags*) Function

Create, open, and return a new mixer object.

filename defaults to `"/dev/mixer"`.

flags defaults to `FCNTL.O_RDWR`.

2.2.2 mixer Methods

fileno () Method on mixer

Return the file number associated with this mixer.

close () Method on mixer

Free the resources associated with this mixer.

Called on deletion if not called explicitly. Do not call any methods of a closed mixer.

devmask () Method on mixer

Return a bitmask of available mixer channels.

recmask () Method on mixer

Return a bitmask of channels that may be used as recording sources.

stereodevs () Method on mixer

Return a bitmask of stereo channels.

caps () Method on mixer

Return a bitmask describing the general capabilities of a mixer.

Currently, the module-level constant mask `SOUND_CAP_EXCL_INPUT` is the only mixer capability defined.

`SOUND_CAP_EXCL_INPUT`
If this bit is set to 1, then only one mixer channel can be selected at the same time.

read_recsrc () Method on mixer

Return a bitmask of active recording sources.

write_recsrc (*mask*) Method on mixer

Select recording source channels with a bitmask.

If *mask* is zero, then the mic input will be used.

read_channel (*channel_num*) Method on mixer

Return the level (volume) of a channel.

The volume is returned as a tuple, (*left*, *right*), representing the level of both stereo channels. For mono devices, only *left* is valid, (the right channel value is set to the left).

Levels are between 0 (off) and 100 (maximum), inclusive.

write_channel (*channel_num, level*) Method on mixer

Set the level (volume) of a channel.

level may be either a single integer or a tuple (*left*, *right*) with values from 0 (off) to 100 (maximum), inclusive.

If *level* is a single integer, it is applied to just a mono channel or both sides of a stereo channel.

If *level* is a tuple, then each side of a stereo channel will be assigned its own level. Mono-devices use the left value of a tuple.

`write_channel()` returns a tuple that represents the new level(s) of a channel. Because some mixers will quantize levels with only 3 to 8 bits, the new levels may *not* be exactly the same as what was requested. See the OSS developer's guide for details.

3 Sequencer Programming

3.1 sequencer Objects

The sequencer object controls the on-board synthesizer using MIDI-like commands and allows I/O through MIDI ports.

Synthesizers accept commands that resemble MIDI commands but otherwise have nothing to do with the MIDI capabilities of a soundcard. For example, some synthesizer methods are called `note_on()` and `chn_pressure()` but these do not send bytes out the MIDI port. Instead, they only mimic similar effects with the synthesizers.

MIDI I/O support consists entirely of simple commands to read and write individual bytes through specific MIDI ports. MIDI input is not implemented in this version.

Read and write operations on a mixer's file descriptor are undefined and there are no `read()` or `write()` methods on a mixer.

Many methods of a sequencer generate an *event* (synthesizer command or a MIDI byte) that is written immediately to a buffer in the **oss** module. As the buffer fills, events will eventually be passed to the driver's queue. The sequencer `dump_buf()` method flushes the module's buffer to the driver's event queue.

The sequencer driver processes most events in its queue immediately. The sequencer method `wait_time()` intersperses timing events between other events. When the driver processes a timing event in the queue, it waits until that absolute time before continuing. Note that in order for the sequencer to see the timing event it must be flushed to the event queue using `dump_buf()`.

3.1.1 sequencer Instantiation

Create an sequencer object using the module function `open_sequencer()`. Only one sequencer may be open at a time; even within the same application.

An open sequencer object will be closed before it is deleted.

open_sequencer (*filename, flags*): Function

Create, open, and return a new sequencer object.

filename defaults to `"/dev/sequencer"`.

flags defaults to `FCNTL.O_RDWR`.

Use `FCNTL.O_WRONLY` for an output-only program so unnecessary functions like MIDI input aren't initialized. Likewise, use `FCNTL.O_RDONLY` for input-only programs.

3.1.2 sequencer Methods

fileno () Method on sequencer

Return the file number associated with this sequencer.

close () Method on sequencer

Free the resources associated with this sequencer.

Called on deletion if not called explicitly. Do not call any methods of a closed sequencer.

`close()` shuts off all synthesizer sounds immediately, so you might want to add an extra delay before closing this device.

nrsynths () Method on sequencer

Return the number of internal synthesizer devices.

synth_info (*obj*) Method on sequencer

Describe a specific synthesizer.

obj may be either an integer device number or a synth_info object with its `device` member initialized.

If *obj* is an integer then return a new synth_info object describing the device. If *obj* is an existing synth_info then it is filled in with the description and returned.

You must provide a device number that is less than the value returned by the method `nrsynths()`.

See Section 3.2 [synth_info Objects], page 12

nrmidis () Method on sequencer

Return the number of MIDI ports available.

midi_info (*obj*) Method on sequencer

Describe a specific MIDI port.

obj may be either an integer port number or a midi_info object with its `device` member initialized to the desired MIDI port.

If *obj* is an integer then return a new midi_info object describing the port. If *obj* is an existing midi_info then it is filled in with the description and returned.

You must provide a port number that is less than the value returned by the `nrmidis()` method.

See Section 3.3 [midi_info Objects], page 13.

start_timer () Method on sequencer

EVENT Reset the sequencer's timer.

Note that this method generates an *event* that will be processed only when it makes it to the head of the queue.

The timer may not be restarted immediately.

wait_time (*time*) Method on sequencer

EVENT Pause the sequencer until *time*.

time is the absolute time when the driver should continue reading the event queue. Absolute time is measured in units of the kernel timer (typically 100 Hz, 10 ms/tick) from the time the device was opened.

Use the `ctrlrate()` method to determine the actual timer resolution.

ctrlrate () Method on sequencer

Return the timer resolution in ticks per second.

midiout (*midi_dev, byte*) Method on sequencer

EVENT Write a byte to a MIDI port.

midi_dev the MIDI port number.

byte either an integer or a single character that is the byte to output.

set_patch (*dev, voice, patch*) Method on sequencer

EVENT Assign synthesizer instrument number (patch) to a voice.

start_note (*dev, voice, note, vel=64*) Method on sequencer

EVENT Start a voice on a synthesizer device.

note
vel valid MIDI note number and velocity values, [0,127].

Some synthesizers support adjusting the volume of a note (rather than starting a new note) with a *note* value of 255. A *vel* of 255 may specify a volume stored in some synthesizers' internal tables. The use of these values is not portable.

stop_note (*dev, voice, note, vel=64*) Method on sequencer

EVENT Stop the synthesizer note being played.

note the MIDI note number

vel the speed of the key release.

For portability, use the same note number that started the voice.

The voice may have already stopped or decayed. The release time is an instrument-specific parameter so the sound may not stop immediately. Consider adding a delay between the final note and the `close()` method, so the last notes have time to decay before being cut off.

chn_pressure (*dev, voice, pressure*) Method on sequencer

EVENT Change the channel pressure.

`chn_pressure()` mimics the pressure used on some MIDI keyboards. The driver translates pressure into the amount of vibrato in OPL-3 voices. *pressure* may be in the range [0,127].

panning (*dev, voice, pos*) Method on sequencer

EVENT Pan the voice between the left and right voice channels.

pos is an integer [-128, 127] where -128 is far left, 0 is center, and 127 is far right.

The pan position is determined by adding *pos* to the pan value stored in the instrument parameters and lasts until until the note is stopped. The OPL-3 pan value is simply its volume level.

control (*voice, controller, value*) Method on sequencer

EVENT Mimic standard MIDI controllers with the synthesizer.

The change may be made before starting a note and lasts until the note is stopped. Once a note is stopped the controller value reverts to its default.

Midi controllers that may be supported are:

`CTRL_EXPRESSION`
: expression controller of MIDI with values in the range [0,127].

`CTRL_MAIN_VOLUME`
: main volume controller with values in the range [0,100].

Note that these values may have different ranges than their standard MIDI counterparts.

bender_range (*voice, value*) Method on sequencer

Define the magnitude of the lowest and highest pitch-bends.

value is in units of cents (1/100 of a semitone). Although *value* may be set larger, the maximum bend is two octaves (1200 cents or 12 semitones).

See `pitch_bend()`.

pitch_bend (*voice, value*) Method on sequencer

EVENT Alter the pitch of a sequencer voice.

value is in the range [-8192, 8191] representing the pitches of the most extreme low and high bends.

The magnitude of the pitch bend is determined by the method `bender_range()`. If you want *value* to be in units of cents (1/100 semitone) then call `bender_range()` with a value of 8192.

The value can be assigned before or after starting the note and the effect lasts until the note stops.

See `bender_range()`.

expression (*voice, value*) Method on sequencer

EVENT Apply an "expression" MIDI controller to a synthesizer voice.

???

main_volume (*voice, value*) Method on sequencer

EVENT Apply a "main volume" MIDI controller to a synthesizer voice.

???

3.2 synth_info Objects

A synth_info object is a description of a synthesizer device.

3.2.1 synth_info Instantiation

Typically a sequencer's `synth_info()` method creates and returns a synth_info object. You can also create a synth_info (for a sequencer to fill in) with the module-level function `synth_info()`.

synth_info () Function

Return a new, uninitialized synth_info object.

3.2.2 synth_info Members

All of these members are read-only except for `device`.

When a synth_info object is printed, it decodes and displays all of its members. This behavior may change in a future release.

`name` a string containing the name of the device

`device` the device number of the synthesizer This is used only when this synth_info object is passed to a sequencer's `synth_info()` method.

`synth_type`
integer describing the type of synthesizer.

`SYNTH_TYPE_FM`

`SYNTH_TYPE_SAMPLE`

`SYNTH_TYPE_MIDI`

`synth_subtype`
integer which may be used to detect the hardware type.

`nr_voices`
maximum number of voices the device supports in its current mode. If you change the mode, the value will change also.

3.3 midi_info Objects

A midi_info object describes a MIDI port.

3.3.1 midi_info Instantiation

Typically a sequencer's `midi_info()` method creates and returns a midi_info object. You can also create a midi_info (for a sequencer to fill in) with the module-level function `midi_info()`.

midi_info () Function

Return a new, uninitialized midi_info object.

3.3.2 midi_info Members

All of these members are read-only except for `device`.

When a midi_info object is printed, it decodes and displays all of its members. This behavior may change in a future release.

`name` a string containing the name of the device

`device` the midi port number, less than sequencer.nrmidis(). This is used only when this midi_info object is passed to a sequencer's `midi_info()` method.

`dev_type` integer describing the type of sound card where the MIDI port resides.

See Section 5.1 [Sound Card ID Numbers], page 19.

`capabilities`
bit mask describing some MIDI capabilities?? One mask is defined:

`MIDI_CAP_MPU401`
mask

4 Audio Programming

4.1 Audio Formats

Audio formats are represented by the following module-level bit-flag values.

`AFMT_MU_LAW`
: logarithmic mu-Law (8-bit)

`AFMT_A_LAW`
: logarithmic A-Law (8-bit)

`AFMT_IMA_ADPCM`
: ADPCM as defined by the Interactive Multimedia Association (IMA). This is *not* the Creative ADPCM format. Requires an average of 4 bits per sample.

`AFMT_U8`
: unsigned 8-bit

`AFMT_S16_LE`
: signed 16-bit little-endian (Intel)

`AFMT_S16_BE`
: signed 16-bit big-endian (Motorola)

`AFMT_S8`
: unsigned 8-bit

`AFMT_U16_LE`
: unsigned 16-bit little-endian

`AFMT_U16_BE`
: unsigned 16-bit big-endian

`AFMT_MPEG`
: MPEG audio format

4.2 audio Objects

An audio object allows you to play and record digitized audio.

You do so by writing and reading samples to the audio device.

Please be aware that because Python is not as fast as languages that are compiled to machine code, you may have difficulty producing some sounds without output buffer underruns and input overflows. These may manifest themselves as pauses, clicking, humming, or other distortions in the sound.

You may have some luck, however, if you use some of the built-in modules and modules compiled from C. Consider using operations in the `array` and `audioop` modules.

Before an audio object can be read or written, you must perform at least three operations on it:

1. Set its sample format with the `format()` method.
2. Set the number of channels (mono or stereo) with the `stereo()` method.

3. Set the sample rate with the `speed()` method.

These operations must be performed in the order presented above. Please consult the OSS Programmer's Guide for more information.

4.2.1 audio Instantiation

Create an audio object using the module function `open_audio()`.

An open audio object will be closed before it is deleted.

open_audio (*filename, flags*): Function

Create, open, and return a new audio object.

filename defaults to `"/dev/audio"`.

flags defaults to `FCNTL.O_RDWR`.

Use `FCNTL.O_WRONLY` if you intend to only play audio. Likewise, use `FCNTL.O_RDONLY` to only record audio.

4.2.2 audio Methods

fileno () Method on audio

Return the file number associated with this audio object.

close () Method on audio

Free the resources associated with this audio object.

Called on deletion if not called explicitly. Do not call any methods of a closed audio object.

`close()` performs a `sync()` operation.

reset () Method on audio

Stop the audio device.

Once an audio device is reset you may set new format, stereo, and speed parameters.

sync () Method on audio

Wait until the last byte has been played.

Once all of the bytes that were written have been played, `sync()` performs a `reset()` on the audio device.

post () Method on audio

Warn the audio device of a pause in the output.

format (*format_flag*) Method on audio

Select and return the audio format.

See Section 4.1 [Audio Formats], page 15 for valid values of *format_flag*.

You should check the return value to ensure that the format you requested is supported.

To check the current audio format, pass `AFMT_QUERY` as *format* or use the `query_format()` method.

get_formats () Method on audio

Return a mask of all the audio formats supported by this device.

See Section 4.1 [Audio Formats], page 15.

query_format () Method on audio

Return the current audio format.

This is a shortcut for `format(AFMT_QUERY)`. See Section 4.1 [Audio Formats], page 15.

stereo (*is_stereo*) Method on audio

Set the number of channels.

is_stereo: 0=mono, 1=stereo

channels (*num_channels*) Method on audio

Set the number of channels.

This operation is documented in the OSS manuals, but not implemented in my version. -twb

num_channels: 1=mono, 2=stereo

speed (*sampling_rate*) Method on audio

Set the sampling rate.

Sets and returns the sampling rate that was as close as the device could get to the requested *sampling_rate*.

read (*bytes_requested*) Method on audio

Record bytes from the audio device.

Returns a string holding the bytes that were read. This may not be the same length as the number of bytes requested.

Note that the request is in units of *bytes*, not samples. For example, if the device is configured for 16-bit stereo, you should request multiples of four bytes.

write (*samples*) Method on audio

Play a string of samples.

samples is a string holding the bytes that should be written to the audio device.

Returns the actual number of bytes that were written. This could be less than the length of *samples*.

samples should be a length that is a multiple of the the sample size (#channels * format size).

5 Miscellaneous

5.1 Sound Card ID Numbers

These module-level variables hold numbers that identify sound cards.

SNDCARD_ADLIB

SNDCARD_SB

SNDCARD_PAS

SNDCARD_GUS

SNDCARD_MPU401

SNDCARD_SB16

SNDCARD_SB16MIDI

SNDCARD_UART6850

SNDCARD_GUS16

SNDCARD_MSS

SNDCARD_PSS

SNDCARD_SSCAPE

SNDCARD_PSS_MPU

SNDCARD_PSS_MSS

SNDCARD_SSCAPE_MSS

SNDCARD_TRXPRO

SNDCARD_TRXPRO_SB

SNDCARD_TRXPRO_MPU

SNDCARD_AWE32

Variable Index

Function Index

W

Table of Contents