vwsnd - Sound driver for the Silicon Graphics 320 and 540 Visual
Workstations' onboard audio.

At the time of this writing, March 1999, there are two models of
Visual Workstation, the 320 and the 540. This document only describes
those models. Future Visual Workstation models may have different
sound capabilities, and this driver will probably not work on those
boxes.

The Visual Workstation has an Analog Devices AD1843 "SoundComm" audio
codec chip. The AD1843 is accessed through the Cobalt I/O ASIC, also
known as Lithium. This driver programs both chips.

==
QUICK CONFIGURATION

```
        # insmod soundcore
        # insmod vwsnd
```

==
I/O CONNECTIONS

On the Visual Workstation, only three of the AD1843 inputs are hooked
up. The analog line in jacks are connected to the AD1843's AUX1
input. The CD audio lines are connected to the AD1843's AUX2 input.
The microphone jack is connected to the AD1843's MIC input. The mic
jack is mono, but the signal is delivered to both the left and right
MIC inputs. You can record in stereo from the mic input, but you will
get the same signal on both channels (within the limits of A/D
accuracy). Full scale on the Line input is +/- 2.0 V. Full scale on
the MIC input is 20 dB less, or +/- 0.2 V.

The AD1843's LOUT1 outputs are connected to the Line Out jacks. The
AD1843's HPOUT outputs are connected to the speaker/headphone jack.
LOUT2 is not connected. Line out's maximum level is +/- 2.0 V peak to
peak. The speaker/headphone out's maximum is +/- 4.0 V peak to peak.

The AD1843's PCM input channel and one of its output channels (DAC1)
are connected to Lithium. The other output channel (DAC2) is not
connected.

==
CAPABILITIES

The AD1843 has PCM input and output (Pulse Code Modulation, also known
as wavetable). PCM input and output can be mono or stereo in any of
four formats. The formats are 16 bit signed and 8 bit unsigned,
u-Law, and A-Law format. Any sample rate from 4 KHz to 49 KHz is
available, in 1 Hz increments.

The AD1843 includes an analog mixer that can mix all three input
signals (line, mic and CD) into the analog outputs. The mixer has a
separate gain control and mute switch for each input.

There are two outputs, line out and speaker/headphone out. They always produce the same signal, and the speaker always has 3 dB more gain than the line out. The speaker/headphone output can be muted, but this driver does not export that function.

The hardware can sync audio to the video clock, but this driver does not have a way to specify syncing to video.

==
PROGRAMMING

This section explains the API supported by the driver. Also see the Open Sound Programming Guide at http://www.opensound.com/pguide/ . This section assumes familiarity with that document.

The driver has two interfaces, an I/O interface and a mixer interface. There is no MIDI or sequencer capability.

==
PROGRAMMING PCM I/O

The I/O interface is usually accessed as /dev/audio or /dev/dsp. Using the standard Open Sound System (OSS) ioctl calls, the sample rate, number of channels, and sample format may be set within the limitations described above. The driver supports triggering. It also supports getting the input and output pointers with one-sample accuracy.

The SNDCTL_DSP_GETCAP ioctl returns these capabilities.

    DSP_CAP_DUPLEX - driver supports full duplex.

    DSP_CAP_TRIGGER - driver supports triggering.

    DSP_CAP_REALTIME - values returned by SNDCTL_DSP_GETIPTR and SNDCTL_DSP_GETOPTR are accurate to a few samples.

Memory mapping (mmap) is not implemented.

The driver permits subdivided fragment sizes from 64 to 4096 bytes. The number of fragments can be anything from 3 fragments to however many fragments fit into 124 kilobytes. It is up to the user to determine how few/small fragments can be used without introducing glitches with a given workload. Linux is not realtime, so we can't promise anything. (sigh...)

When this driver is switched into or out of mu-Law or A-Law mode on output, it may produce an audible click. This is unavoidable. To prevent clicking, use signed 16-bit mode instead, and convert from mu-Law or A-Law format in software.

==
PROGRAMMING THE MIXER INTERFACE

The mixer interface is usually accessed as /dev/mixer. It is accessed

through ioctls. The mixer allows the application to control gain or
mute several audio signal paths, and also allows selection of the
recording source.

Each of the constants described here can be read using the
MIXER_READ(SOUND_MIXER_xxx) ioctl. Those that are not read-only can
also be written using the MIXER_WRITE(SOUND_MIXER_xxx) ioctl. In most
cases, <sys/soundcard.h> defines constants SOUND_MIXER_READ_xxx and
SOUND_MIXER_WRITE_xxx which work just as well.

SOUND_MIXER_CAPS   Read-only

This is a mask of optional driver capabilities that are implemented.
This driver's only capability is SOUND_CAP_EXCL_INPUT, which means
that only one recording source can be active at a time.

SOUND_MIXER_DEVMASK   Read-only

This is a mask of the sound channels. This driver's channels are PCM,
LINE, MIC, CD, and RECLEV.

SOUND_MIXER_STEREODEVS Read-only

This is a mask of which sound channels are capable of stereo. All
channels are capable of stereo. (But see caveat on MIC input in I/O
CONNECTIONS section above).

SOUND_MIXER_OUTMASK   Read-only

This is a mask of channels that route inputs through to outputs.
Those are LINE, MIC, and CD.

SOUND_MIXER_RECMASK   Read-only

This is a mask of channels that can be recording sources. Those are
PCM, LINE, MIC, CD.

SOUND_MIXER_PCM   Default: 0x5757 (0 dB)

This is the gain control for PCM output. The left and right channel
gain are controlled independently. This gain control has 64 levels,
which range from -82.5 dB to +12.0 dB in 1.5 dB steps. Those 64
levels are mapped onto 100 levels at the ioctl, see below.

SOUND_MIXER_LINE   Default: 0x4a4a (0 dB)

This is the gain control for mixing the Line In source into the
outputs. The left and right channel gain are controlled
independently. This gain control has 32 levels, which range from
-34.5 dB to +12.0 dB in 1.5 dB steps. Those 32 levels are mapped onto
100 levels at the ioctl, see below.

SOUND_MIXER_MIC   Default: 0x4a4a (0 dB)

This is the gain control for mixing the MIC source into the outputs.
The left and right channel gain are controlled independently. This

```
gain control has 32 levels, which range from -34.5 dB to +12.0 dB in
1.5 dB steps.  Those 32 levels are mapped onto 100 levels at the
ioctl, see below.

SOUND_MIXER_CD          Default: 0x4a4a (0 dB)

This is the gain control for mixing the CD audio source into the
outputs.  The left and right channel gain are controlled
independently.  This gain control has 32 levels, which range from
-34.5 dB to +12.0 dB in 1.5 dB steps.  Those 32 levels are mapped onto
100 levels at the ioctl, see below.

SOUND_MIXER_RECLEV       Default: 0 (0 dB)

This is the gain control for PCM input (RECording LEVel).  The left
and right channel gain are controlled independently.  This gain
control has 16 levels, which range from 0 dB to +22.5 dB in 1.5 dB
steps.  Those 16 levels are mapped onto 100 levels at the ioctl, see
below.

SOUND_MIXER_RECSRC       Default: SOUND_MASK_LINE

This is a mask of currently selected PCM input sources (RECording
SouRCes).  Because the AD1843 can only have a single recording source
at a time, only one bit at a time can be set in this mask.  The
allowable values are SOUND_MASK_PCM, SOUND_MASK_LINE, SOUND_MASK_MIC,
or SOUND_MASK_CD.  Selecting SOUND_MASK_PCM sets up internal
resampling which is useful for loopback testing and for hardware
sample rate conversion.  But software sample rate conversion is
probably faster, so I don't know how useful that is.

SOUND_MIXER_OUTSRC      DEFAULT: SOUND_MASK_LINE|SOUND_MASK_MIC|SOUND_MASK_CD

This is a mask of sources that are currently passed through to the
outputs.  Those sources whose bits are not set are muted.

==============================================================================
GAIN CONTROL

There are five gain controls listed above.  Each has 16, 32, or 64
steps.  Each control has 1.5 dB of gain per step.  Each control is
stereo.

The OSS defines the argument to a channel gain ioctl as having two
components, left and right, each of which ranges from 0 to 100.  The
two components are packed into the same word, with the left side gain
in the least significant byte, and the right side gain in the second
least significant byte.  In C, we would say this.

        #include <assert.h>

        ...

                assert(leftgain >= 0 && leftgain <= 100);
                assert(rightgain >= 0 && rightgain <= 100);
                arg = leftgain | rightgain << 8;
```

So each OSS gain control has 101 steps. But the hardware has 16, 32,
or 64 steps. The hardware steps are spread across the 101 OSS steps
nearly evenly. The conversion formulas are like this, given N equals
16, 32, or 64.

```
        int round = N/2 - 1;
        OSS_gain_steps = (hw_gain_steps * 100 + round) / (N - 1);
        hw_gain_steps = (OSS_gain_steps * (N - 1) + round) / 100;
```

Here is a snippet of C code that will return the left and right gain
of any channel in dB. Pass it one of the predefined gain_desc_t
structures to access any of the five channels' gains.

```
        typedef struct gain_desc {
                float min_gain;
                float gain_step;
                int nbits;
                int chan;
        } gain_desc_t;

        const gain_desc_t gain_pcm    = { -82.5, 1.5, 6, SOUND_MIXER_PCM    };
        const gain_desc_t gain_line   = { -34.5, 1.5, 5, SOUND_MIXER_LINE   };
        const gain_desc_t gain_mic    = { -34.5, 1.5, 5, SOUND_MIXER_MIC    };
        const gain_desc_t gain_cd     = { -34.5, 1.5, 5, SOUND_MIXER_CD     };
        const gain_desc_t gain_reclev = {   0.0, 1.5, 4, SOUND_MIXER_RECLEV };

        int get_gain_dB(int fd, const gain_desc_t *gp,
                        float *left, float *right)
        {
                int word;
                int lg, rg;
                int mask = (1 << gp->nbits) - 1;

                if (ioctl(fd, MIXER_READ(gp->chan), &word) != 0)
                        return -1;      /* fail */
                lg = word & 0xFF;
                rg = word >> 8 & 0xFF;
                lg = (lg * mask + mask / 2) / 100;
                rg = (rg * mask + mask / 2) / 100;
                *left = gp->min_gain + gp->gain_step * lg;
                *right = gp->min_gain + gp->gain_step * rg;
                return 0;
        }
```

And here is the corresponding routine to set a channel's gain in dB.

```
        int set_gain_dB(int fd, const gain_desc_t *gp, float left, float right)
        {
                float max_gain =
                        gp->min_gain + (1 << gp->nbits) * gp->gain_step;
                float round = gp->gain_step / 2;
                int mask = (1 << gp->nbits) - 1;
                int word;
                int lg, rg;
                if (left < gp->min_gain || right < gp->min_gain)
                        return EINVAL;
                lg = (left - gp->min_gain + round) / gp->gain_step;
                rg = (right - gp->min_gain + round) / gp->gain_step;
                if (lg >= (1 << gp->nbits) || rg >= (1 << gp->nbits))
                        return EINVAL;
                lg = (100 * lg + mask / 2) / mask;
                rg = (100 * rg + mask / 2) / mask;
                word = lg | rg << 8;

                return ioctl(fd, MIXER_WRITE(gp->chan), &word);
        }
```