Contents

Hacker's Guide to VoxWare 2.4
(second draft)

Hannu Savolainen,
Hannu.Savolainen@helsinki.fi

February 21, 1994

This version is absolutely incomplete and full of errors. It covers just the basic programming. A more complete and correct version should become available some day.

I have not read this text after writing it. I will do it later but at this time my goal is just to write the things down.

Part I

Preface

This document describes sound application programming using the VoxWare driver. Some kind of C and Unix programming skills are required to understand this text but you don't need to be a Guru class programmer. You should know how to open (I mean `open` not `fopen`) and use files. For example if you have ever called `ioctl`, you should have the necessary skills. There are (will be) some text which require use of `select` but you may skip it if you wish.

This document describes the steps required when writing an application which uses a particular device file. There is a separate chapter for each of the device files. Each chapter describes the things required to do when using the particular device. Chapters should be self contained so you may read just the chapters you are interested in. It's recommended that you read the entire chapter before starting programming.

Be careful! Some parts of this document describe features that will be released later (version 3.0??). I have not put warnings about this on every paragraph. Just to the beginning of chapters. The version 3.0 should be released within year -94. Some alpha test versions (2.99.XX) should be available earlier. The schedule and contents of the 3.0 version depends on how much time I'm able to allocate for it.

The reason why I'm writing this document is that I have spent months on developing the VoxWare device driver. Why I have written it? Well, it's a long story. Once upon a time I decided to make a computer animation. Since making computer graphics require more special skills and computer horsepower than I had, I decided to start by making the soundtrack first.

What was required to make the soundtrack using a computer? Answer: Some kind of software and some kind of voice output device. What was available? Answer: The PC speaker. Bah.

Then one day I saw a SoundBlaster card (1.5), looked at it and bought it. What I could do with it? Answer: nothing. There was a funny looking and yet funnier sounding Intelligent Organ application in the discs which came with the card but no programming information. Why? Answer: The programming information was in the Software Development Kit which was sold separately. Back to the shop to buy the SDK. It costed me the same that the card itself but what was possible to do with it? Answer: nothing. Writing an computer animation soundtrack production software under MS-DOS looked like real work. Not for me so I made a decicion to shut down and to go to sleep.

Then after several months I bought an 386 PC and Minix. After several months of fine hacking I had upgraded it to something which looked like Unix (it was Minix-386 with gcc compiler). Then I remembered the soundtrack project and started to study how to used the SB card with it. I decided to write a SB driver for Minix but after couple of weeks it started to look like a real work (which was not for me). I got sound out but there was serious pauses in it.

Then there was a delay of about an year before I finally got the Linux installed to my system (it was May 92). After a week of use I finally decided to write a SB driver for Linux. It too couple of months to write the initial version and couple more to test and debug it. The first version was released on August of 92. It supported just SB and AdLib. Then I decided to add support for SB Pro and then for PAS16. The first true version (1.0) was released on December 92 (was it?) Then the Gravis UltraSound was released. It looked like a perfect soundcard for making soundtracks to computer animations. So back to the keyboard again. Then someone requested an PAS16 driver for ISC so why not to convert the driver to support the ISC or why not to make the driver to support several operating systems at once. Since it was possible to make a driver which supported more than one soundcard types, it should be possible to support several operating systems as well. And this is finally proving to be true.

What's next? I don't know. At least the deadline of the computer animation project has moved to the next century. Even the total soundtrack production system is still unstarted. New exellent soundcards are coming to the market. But what to do next? Answer: Write the bloody documents (this). What then? Try to find sponsors? But how?

I have made a desicion to keep the driver available under a BSD like copyright. I don't want to restrict use of the driver in any ways. I want to keep rights to continue developing it.

In addition I'm trying to keep just one version path available which supports as many soundcards and operating systems as possible. With current version this is possible just if all of the low level soundcard and OS modules are included in the driver package. In future it should be possible to have a VoxWare core package into which the low level interface modules can be installed.

The device file interface of the VoxWare driver is almost OS and hardware independent and easily extensible. I'm trying to improve it so that it can be used as a standard sound interface for Unix. Particularily in the i386 PC field but why not for the other platforms too. It should be possible to port the driver to systems having EISA bus (HP7XX and DEC Alpha PC).

I hope the soundcard and OS manufacturers and other interested parties sponsor this work. The most urgent need is for all kind of documentation about writing device drivers for different operating systems and about the hardware level details of the soundcards. The most difficult problem in writing this driver has been finding programming information for the new soundcards. For example the SB16 was released at the end of 92 and the SDK for it almost a year later. All the necessary programming information was covered by just about 10 pages in the manual. A request to the soundcard manufacturers:

Please, provide the hardware level programming information free of charge as soon as possible after the card is announced. The best way to distribute it is for example a BBS system managed by the manufacturer. Having this kind of document available early will ensure your card gets supported by freeware ans shareware software producers as soon as possible.

I'm still interested to add support for new soundcards. Particularily for the cards which are fully programmable music synthesizers like GUS. However I have no longer possibility to buy new soundcards myself. Therefore I hope that soundcard manufacturers support this project by sending me free cards and developer kits. Particularily for the hot WS synth cards.

I have already started making better driver for the MPU-401 and compatibles (MQX-32M and Super MPU).

I also accept any amounts of monetary sponsoring. It makes it easier to explain my wife why I spend most of my time (and money) to a hobby and don't try to find a productive job[1].

Helsinki, Finland, January 1994

Hannu Savolainen

E-mail: hsavolai@cs.helsinki.fi (or Hannu.Savolainen@helsinki.fi).

Snail Mail: Pallaksentie 4A2, 00970 Helsinki, Finland

FAX: +358 0 395 1968 (not guaranteed to be on all the time)

0.1 Background

VoxWare is a device driver for accessing various PC soundcards under various Unix like operating systems. The current version 2.4 supports just Linux but the driver is being ported to some other environments like SCO, ISC and NetBSD/FreeBSD. Earlier the driver was known as the Linux Sound Driver. Since next version will support more operating systems, it's time to give it a new name. Let's call it VoxWare [2]

The current version supports the most common soundcards such as SoundBlaster (SB) versions 1.0, 1.5, 2.0 and SB Pro, ProAudio Spectrum 16 (PAS16), Advanced Gravis UltraSound (GUS) and Roland MPU-401 Midi card. In addition all cards which are 100% hardware compatible with one of the above

[1] If I find a productive job, I will have money to buy new soundcards but no time to use them or to write the driver. Sad.

[2] If you know the name VoxWare is reserved, please let me know.

are supported. Including ThunderBoard, ATI Stereo F/X and Logitech SoundMan16.

Currently there are lot's of soudcards in the market. Some of the unlisted ones may work with the VoxWare driver while the others don't.

As said the VoxWare driver controls PC soundcards. Most of them have several separate devices which produce or record sound. There are differences between various cards but most of them have the following devices:

1. Digitized voice device (usually called as PCM, DSP or ADC/DAC) is used for recording and playback of digitized voice. When recording the device measures the input level several times per second. The sampled value is represented by a 8 or 16 bit number. The number of samples per second is called *Sample Rate*. Current soundcards support samling rates between 4 kHz and 44.1 kHz.

2. Synthesizer device is used mainly for playing music. The VoxWare driver supports two kind of synthesizer devices. The older one is the Yamaha FM synthesizer chip which is available in most of the soundcards. There are two models of the FM chip. The Yamaha YM3812 is a 2 operator version whic is available in the AdLib, SoundBlaster 1.0 to 2.0 and the first SB Pro models. It has just 9 simultaneous voices and is not capable to produce realistic instrument timbres. The OPL-3 is an improved version of the YM3812. It supports 4 operator voices which produce more realistic sounds.

   The second type of synthesizer devices are so called *Wave Table Synthesizers*. These devices produce sound by playing back prerecorded instrument samples. This method makes it possible to produce extremely realistic instrument timbres. The current VoxWare version support just one soundcard which has a wave table based synthesizer (Gravis UltraSound (GUS)).

   Synthesizer devices don't have recording capability.

3. Midi interface is a device which is used to connect external synthesizers to the computer. Technically the midi interface is similar than the serial ports (RS-323) used in the computers (but not compatible). In hardware level the midi interface is designed to work in stage environments in the middle of megawatt class power amplifiers and lighting equipments.

   The synthesizers and computers communicate together by sending messages via the midi cable.

4. Most of the soundcards have also a joystick port and some kind of interface for a CD-ROM drive. These devices are not controlled by the VoxWare but there are separate drivers available (at least in Linux).

0.2 Getting the VoxWare driver

The latest official version is available as a part of the Linux kernel distribution. You can get it from the nearest anonymous ftp site or BBS carrying Linux. For example thw ftp sites: nic.funet.fi, sunsite.unc.edu and tsx-11.mit.edu have Linux.

The latest ALPHA TEST version is available at
`nic.funet.fi:pub/OS/Linux/ALPHA/sound`
These versions contain some experimental features and/or require more testing. It could even be possible that the version in the Linux kernel distribution is more recent than the last released alpha test version.

The VoxWare driver should work with some other Unix operating systems than Linux. At this time these versions are somehow incomplete. There are working versions for SCO Unix and FreeBSD/NetBSD. Currently these drivers are distributed with the Linux version. Look at the Readme of the driver before installing it to the other OS than Linux. There could be some restrictions in the functionality.

Part II

Programmer's Guide

The programming interface (API) of the VoxWare driver is defined in the include file `sys/soundcard.h`. There is another include file for the Gravis Ultrasound also `sys/ultrasound.h` but normally it should not be required.

The `soundcard.h` is distributed with the driver (sources) and care must be taken that the latest version is always used by the application. In Linux this header file is distributed in the directory linux/include/linux of the kernel sources. You have to ensure that the directory /usr/include/linux is a symbolic link to that directory. It ensures that the /usr/include/linux contains always the latest set of headers.

The file /usr/include/sys/soundcard.h (in Linux) should be a text file containing just an #include for the linux/soundcard.h. Also a symbolic link could work.

In case you have installed a separately distributed (test) version of the driver to Linux, you have to manually copy the soundcard.h from the driver directory to the /usr/include/linux. Otherwise you will not use the latest version when compileing the driver.

In the FreeBSD the soundcard.h is distributed in the directory /usr/include/machine. You should make a sys/soundcard.h similar than in Linux.

On NetBSD, SCO, ISC and others you should copy the soundcard.h from the driver directory to the /usr/include/sys (a symlink is also possible).

If your program (or the driver) doesn't compile, just check that the latest version of the soundcard.h is used. You have to recompile both the driver and the application if there are problems with the versions of the soundcard.h (just to be sure).

The VoxWare driver supports several different types of device files. These types are the following:

1. `/dev/mixer`
   There is just one device of this type (the version 3.0 will have more). It's used mainly for accessing the builtin mixers of some soundcards (currently just SB Pro, SB16, PAS16 and GUS). With mixer it's possible to for example adjust playback and recording levels of various sources. This device file is used also for selecting the recording sources. In the future this device file could be used for some additional tasks such as changing configuration information of the driver. It's not possible to produce sound using this device file. Just the `ioctl` -call is supported.

2. `/dev/sndstat`
   This device file is just for diagnostic purposes. Use `cat /dev/sndstat` to print some usefull information about the driver configuration. This device file has earlier been supported only by the Linux version. Since version 2.3 it has been available in other ones also.

3. /dev/dsp
   This is the main device file for digitized voice applications. Any data written to this device is played with the DAC/PCM/DSP device of the soundcard. Reading this device returns the audio data recorded from the current input source (the default is microphone input). If there is two digitized voice devices installed in the system, the second one can be assessed via the `/dev/dsp1`. For example the PAS16 card has two separate digitized voice channels. The `/dev/dsp16` (`/dev/dsp16_1`) is similar than the `/dev/dsp` but the default sample size is 16 bits/sample.

   This device file can be used for applications such as speech synthesis and recognition and voice mail.

4. /dev/audio
   The `/dev/audio` is similar than the `/dev/dsp`. The difference is that the `/dev/audio` uses logarithmic μ-Law [3] encoding. This device file provides limited compatibility with the `/dev/audio` device of Sun workstations. However the Sun compatible ioctl interface has not been implemented (yet?). If you want to access the full capabilities of the VoxWare driver, the `/dev/dsp` is the recommended device file to use.

5. /dev/sequencer
   This device file is intended for electronic music applications. It can be also used for producing various sound effects in games. The `/dev/sequencer` provides access to any internal synthesizer devices of the soundcards. For example the Yamaha OPL-3 and YM3812 FM synthesizer devices (SB, PAS16 and AdLib) and the wave table synthesizer of the GUS card. In addition this device file can be used for accessing any external music synthesizer devices connected to the Midi -port of the soundcard. More than one synthesizer chips and midi interfaces are supported at the same time.

6. /dev/sequencer2
   Version 3.0 will have an extended version (`/dev/sequencer2` which should be more device independent than the current one. It also supports full capabilities of the finest MIDI cards (for example tape sync).

7. /dev/midi##
   The version 3.0 of the driver will contain a set of device files for the MIDI interfaces. These are much like `tty` devices (raw mode). These device files are intended for 'non-realtime' use. There is no timing capabilities so everything written to the device file will be sent to the MIDI port as soon as possible. There are some ioctl calls for sending commands to the MPU-401 compatible midi adapters. This should make it possible to write

[3] With μ-Law encoding a sample recorded with 12 bit resolution is represented by a 8 bit byte. The VoxWare driver doesn't support 12 bit resolution but converts the μ-Law encoded data to a 8 bit linear format.

Device	Minor	Multi
`mixer`	0	yes
`sequencer`	1	no
`midi`	2	yes
`dsp`	3	yes
`audio`	4	yes
`dsp16`	5	yes
`sndstat`	6	no
unused	7	no
`sequencer2`	8	no

Table 0.1: Minor number assignment of the device files

applications using the MPU-401 directly in the intelligent mode (at least recording works. I'm not sure about playback).

These device files share the same major device number [4]. The minor number assignment is given in the table 0.1. The four least significant bits of the minor number are used to select the device file type or class. If there is more than one devices in this class, the upper 4 bits are used to select the device. For example the class number of the `/dev/dsp` is 3. Then the minor number of `/dev/dsp` is $3 + 16 * 0 = 3$ and the `/dev/dsp1` is $3 + 16 * 1 = 19$.

[4] The major device number is 14 in Linux. On other operating systems it's propably something else.

Chapter 1

Using /dev/mixer

Most of the current soundcards have some kind of mixer which can be used for adjusting volumes of several voice channels. The version 2.4 of the VoxWare driver has mixer support for the SB Pro, SB16, PAS16 and the GUS. There could be mixers in some other cards which work with the driver but usually the mixer part is not supported.

The mixer interface supports several devices or channels. Examples of the channels are microphone, line in, audio input from the CD-ROM drive, built in synthesizer and the digitized voice device. The set of channels available on particular cards differ but the `/dev/mixer` interface provides a way to determine the supported channels.

The mixer interface is used for selecting the recording sources for the `/dev/dsp` and the `/dev/audio`. Most of the soundcards have capability to adjust the recording volumes also. The SB Pro is capable to adjust just the microphone input level and the current version of GUS has just on/off controls for the input sources.

If there are more than one soundcards installed in the system, it's possible that there are more than one mixers. The 'first' one is accessed through `/dev/mixer` and the second through `/dev/mixer1` etc.

The mixer ioctl calls are accessible through every device file of the VoxWare driver. The restriction is that just the first mixer device is accessible in this way. It's for example possible to select the recording source by calling a mixer ioctl on the `/dev/dsp` device. There is no need to open the `/dev/mixer` just for this purpose but remember that this is possible just for the first mixer.

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 itself.

1.1 Mixer channels

A mixer is a device which can be used to adjust reletive volumes between various input sources. The input sources like Mic, Line and CD are called channels. The 'input' means here just that the channels are inputs to the mixer. Output of the mixer is called main volume or just vol.

In fact there are actually two mixers in most soundcards. The first one controls the recording levels and is called input mixer. The secvond is an output mixer and it controls level of the sound actually going out of the card (phones or line out jack). The VoxWare drivers hides this kind of details. There is just one mixer and one level setting for each of the channels. The selection between the different mixers are made automaticly when recording.

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

```
#define SOUND_MIXER_NRDEVICES   12
#define SOUND_MIXER_VOLUME      0
#define SOUND_MIXER_BASS        1
#define SOUND_MIXER_TREBLE      2
#define SOUND_MIXER_SYNTH       3
#define SOUND_MIXER_PCM         4
#define SOUND_MIXER_SPEAKER     5
#define SOUND_MIXER_LINE        6
#define SOUND_MIXER_MIC         7
#define SOUND_MIXER_CD          8
#define SOUND_MIXER_IMIX        9
#define SOUND_MIXER_ALTPCM      10
#define SOUND_MIXER_RECLEV      11
```

The macro `SOUND_MIXER_NRDEVICES` gives the number of channels known when the soundcard.h was written. Any program should not try to access channels greater or equal than `SOUND_MIXER_NRDEVICES`. If more mixer channels are added to the driver, the application program must be recompiled before it's able to use the new channels.

The following channels are currently known by the driver:

- `SOUND_MIXER_VOLUME`
  This is the master output level (headphone/line out volume).
- `SOUND_MIXER_BASS`
  This channel controls the treble level of all of the output channels.
- `SOUND_MIXER_TREBLE`
  This channel controls the bass level of all of the output channels.

- `SOUND_MIXER_SYNTH`
  Volume for the built in synthesizer chip like OPL-3 or GUS. For most cards this controls just the output level. With PAS16 this controls the recording level also since it's possible to record from the FM chip.

- `SOUND_MIXER_PCM`
  Output level for the digitized voice channel.

- `SOUND_MIXER_SPEAKER`
  Output volume for the PC speaker signals. Works only if the speaker output is connected directly to the soundcard. Doesn't affect the built in speaker, just the signal which goes through the soundcard.

- `SOUND_MIXER_LINE`
  Volume level for the line in jack.

- `SOUND_MIXER_MIC`
  Volume for the signal coming from the microphone in jack. This is the only adjustable input volume channel in the SB Pro (the other possible recording sources have fixed input volume).

- `SOUND_MIXER_CD`
  Volume level for signal connected to the CD audio input. Usually this input is just three or more pins in the soundcard.

- `SOUND_MIXER_IMIX`
  Some kind of recording monitor on the PAS16 cards. Controls the output (headphone jack) volume of the selected recording sources while recording. This channel has effect just when recording.

- `SOUND_MIXER_ALTPCM`
  Volume of the alternate digitized voice device (the SB emulation of the PAS16 board).

- `SOUND_MIXER_RECLEV`
  Global recording level setting. In the SB16 card this controls the input gain which has just 4 possible levels.

The soundcard.h defines also two sets of printable names for the channels. These name should be used when labelling or naming the mixer channels by 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:

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

For example labels[SOUND_MIXER_VOLUME] contains a textual label for the main volume channel.

The macro SOUND_DEVICE_NAMES is similar but it contains names to be used for example when parsing command lines etc. The names in this macro don't contain blancs or capital letters.

1.2 Obtaining the current mixer config

The mixer interface of the VoxWare driver has been designed so that it's possible to compile a mixer program on one system and to use it on another system with different hardware setup.

This is possible only if the mixer program follows some guidelines. It has to query for the hardware configuration before making any other actions with the mixer interface. It's not dangerous if the program tries just to change volume of a channel without making the query since the ioctl call returns an error if there is something wrong with the config. But if a mixer programs shows the complete list of channels even if there is no mixer, the user will get confused. The following ioctl calls give the program a way to determine the current setup.

```
int devices;

ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devices);
```

Returns a bitmask in the devices variable. There is one bit position for each of the mixer channels. Existence of a channel can be tested as the following:

```
if (devices & (1 << channel_no))
  {
            /* The channel is supported by the current mixer HW */
  }
```

The soundcard.h contains predefined bitmask macros for each of the mixer channels. For example the SOUND_MIXER_MIC defines the bit which can be used to test if the mic channel is supported.

The VoxWare driver returns some other bitmasks for various purposes:

- SOUND_MIXER_READ_DEVMASK (described above) has a bit on for each supported mixer channel.
- SOUND_MIXER_READ_RECMASK returns a bit on for each mixer channel which can be used as a recording source.
- SOUND_MIXER_READ_STEREODEVS defines the mixer channels which have stereo capability. If the corresponding bit is off, there is no separate left and right channel settings for the channel.

The `SOUND_MIXER_READ_CAPS` returns an integer with bits describing the mixer. Currently just one bit is defined. The bit `SOUND_CAP_EXCL_INPUT` is on if just one channel can be used as a recording source at the same time (SB Pro).

1.3 Getting and setting volumes

An application program can get the volume of a device by calling:

```
int volume;

ioctl(fd, MIXER_READ(channel_no), &volume);
```

The channel_no is one of the channel numbers defined in the soundcard.h. The volume for both stereo channels are returned in the same int variable. The lsb gives volume for the left channel and the next 8 bits for the for the right channel. The upper 16 bits are undefined and should be ingnored.

The volumes for the left and right stereo channel can be extracted as the following:

```
       int left = volume & 0x000000ff;
       int right = (volume & 0x0000ff00) >> 8;
```

Note! Just the left channel value is valid for the mono channels.

The volume setting can be altered by using another ioctl:

```
int volume = (left & 0xff) | ((right & 0xff) << 8);

ioctl(fd, MIXER_WRITE(channel_no), &volume);
```

Just the left stereo channel volume is valid for the mono channels. It's recommended that the left channel value is used for the right channel also.

1.4 Volume levels

The VoxWare driver accepts volumes between 0 and 100. The 0 means off and the 100 means maximum.

Most of the mixers have 3 to 8 bits for the volume and the driver scales between the local and hardware defined volume. Since this scaling is not accurate, the application should be carefull when using the volume returned by the ioctl calls. If the application writes the volume and 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 slides to zero even if the application makes no changes itself. It's recommended that the application reads the volume just during initialization and ignores the volume returned later.

Note! The `MIXER_WRITE` returns the modified volume in the argument after call. A temporary variable should be used as the argument. Otherwise the volume will slide down on each access.

1.5 Selecting the recording sources

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

The `SOUND_MIXER_READ_RECSRC` returns a bitmask having a bit set for each of the currently active recording sources. The default is currently mic in but the application should not assume this. The recording source could have been changed after boot.

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

The SB Pro allows just one active input source at the same time. The driver has a simple expert system to handle invalid recording source selections. A mixer program should always check the recording mask after changing it. It should also update the display if the returned mask is something else than the requested one.

Chapter 2

Using /dev/dsp

The `/dev/dsp` family of device files is designed for recording and playback of digitized voice. Writing an application for these devices is extremely easy. All the program has to do is to open the device, set the recording/playback parameters[1] and finally read or write the data. In practise little bit more is required.

2.1 Some background information

The digitized voice recording process is called sampling. The audio source is connected to a device called *Analog-Digital Converter* (DAC). It measures the input signal and converts the signal level to a 8 or 16 bit number. This process is repeated using constant rate. For example 8000 times per second. This rate is called *Sampling Rate* and is given in units of Hz. When sound is recorded using the default sampling frequency of 8 kHz and the default sample size (resolution) of 8 bits/sample, a constant stream of 8000 bytes is returned for each second.

The sampling rate limits the maximum frequency which is present in the recorded signal to a half of the sampling rate. For example with the default sampling rate of 8 kHz the frequencies above 4 kHz are filtered out by the device.

The sample size limits the dynamic range of the recorded signal (the difference between minimum and maximum recordable signal levels). For example with 8 bits/sample resolution the dynamic range is (2*8)*3 dB = 48 dB and with 16 bits/sample it's 96 dB. These are theoretical limits. Most of the currently available 16 bit soundcards provide less than 85 dB dynamic range (making a perfect DAC is pretty expensive).

In addition to the sampling rate and sampling frequency it's also possible to change the number of channels used when recording the voice. The default is one channels (mono) and the other alternative is two channels (stereo).

[1]Even this is not required if the default values provided by the driver are sucfficient

Playback is the reverse process of recording. Data is just converted back to reproduce the original audio signal. The difference between the original and reproduced signal depends on the sampling rate, sampling resolution and number of channels. With best of the current soundcards it's possible to come close to so called CD quality. The worst case quality (4 kHz/8 bit) is close to a toy telephone. It's also possible to get totally scrambled result if the recording and playback parameters don't match together. For example if a 16 bit recording is played back with just 8 bits, the result is just noise.

The digitized voice devices are not limited just for playback of prerecorded sound. It's possible to produce fully synthetic sound by computing a stream of samples and sending it to the device. The result could be for example music or speech depending on the computation algorithm. Recorded sound can also be analyzed by a computer program. Speech recognition is a good example of such process.

2.2 Your first audio application

The simpliest way to use the `/dev/dsp` devices doesn't require programming at all. You could for example to record your beautyful voice using the following simple procedure:

1. Connect a microphone to the MIC IN jack of the soundcard.
2. Give command `cat /dev/dsp > file`
3. Say: "*Hello, World.*" to the microphone.
4. Wait couple of seconds.
5. Hit ctrl-C.

That's it. Wasn't it easy. Now you can play the file using command `cat file > /dev/dsp`. You should hear someones voice saying: "`Hello, World`". You should recognize the speaker at least if you have talked with yourself on telephone. Since the default sampling frequency (8 kHz) of the `/dev/dsp` is used also in the telephone network, your played back voice should sound similar than in telephone[2].

Try to output the file to the `/dev/audio`. If you still think your voice is beautyful, there must be something wrong in yours ears. Try to answer why it didn't sound good. The answer will be given later.

[2] Writing a telephone speaking trainer application is a good programming exercise which you propably can implement after reading this section.

Now you may ask yourself: "Why should I write a program to use the /dev/dsp if it's so easy to record and playback using cat?" [3]. There are several reasons to have a better recording and playback program:

- The default sampling rate and sampling resolution is not suitable for recording high quality sound. For example you will need better quality if you are recording a library of bird voices. In addition the stereo recording capability is required for some applications.
- There is no EOF condition available in the /dev/dsp. Sound recording using cat continues infinitely or until your disk becomes full. The only way to terminate recording is to kill the cat. With a custom written application it's possible to stop after recording enough data.
- There are unpredictable delays in the recording since the recording is done in fixed size blocks. The reading program waits until the first block is completely recorded before returning a single byte. If the program is killed too early, no bytes are written to the file. This is the reason to the two seconds of delay in the hand recording procedure described earlier. The recorded file will contain up to 2 seconds of silence (or hiss) after the required sound.

If you need just a capability to record to a file or to play from a file, there are several ready applications available. The simpliest one is the srec/splay program of the snd-util package. It simply stores the recorded voice to a disk file without any header information. When the file is played, the correct parameters (rate etc) must be provided by user. There are several better programs available which recognise the commonly used audio file formats such as .VOC or .WAV. The most impressive package is sox (Lsox.tar.gz for Linux). It's capable to convert between any known audio formats.

2.3 Your first own audio application

Now it's time to start programming.

```
/* Sample program for recording from /dev/dsp */

#include <unistd.h>
#include <fcntl.h>

#define DSP_NAME        "/dev/dsp"
```

[3] Exercise: Record this question using the procedure described above, listen the question and answer it. Optionally you could store the question and answer files and write an audible sound FAQ librarian (yet better exercise).

```
#define DSP_MAXCOUNT   10000 /* # of samples to be recorded */

int dspfd;

int main(int argc, char *argv[])
{
   unsigned char buffer[256];
   int i, l;
   int max_count = DSP_MAXCOUNT;
   int next_count;

   if ((dspfd=open(DSP_NAME, O_RDONLY, 0))==-1) die("open failed");

   next_count = sizeof(buffer);

   while (next_count > 0 &&
          (l=read(dspfd, buffer, next_count)) > 0)
   {
       if (write(1, buffer, l)==-1) die("Personal problems.");

       max_count -= l;
       next_count = max_count; /* # of bytes left */
       if (next_count > sizeof(buffer))
           next_count > sizeof(buffer);
   }

   if (l == -1)exit(-1);

   exit(0);
}
```

As you propably noticed, there was nothing special in the above program. The only extra thing the recording program must do is the decision when to stop recording. The device file itself doesn't provide any end of data information[4]. The application could just count the recorded bytes (time) or to wait until the user asks the application to stop.

It's recommended that the audio device is opened using `O_RDONLY` if the device is opened just for reading. It could make some decisions easier to the driver.

If the device file is in use when a program tries to open it, the driver will return EBUSY. There is currently no way to wait until the device gets ready.

The `O_NDELAY/O_NONBLOCK` flack has no effect. The open works always just like with `O_NDELAY`. Subsequent read and write calls ignore this flag.

[4] Of cause a speech recognition program could wait until it recognizes a world "quit".

Making an playback program is even easier. You don't have need to count the bytes to detect the end condition. Remember to open the device using `O_WRONLY`. Writing a program which both records and plays back will be covered later (2.6.1).

2.4 Encoding of the audio data

As said the sampled audio data is encoded as a stream of numbers representing the measured signal levels. When the sampling resolution is 8 bits/sample, each sample will take one byte (char) of space. Simple but not that simple. Just look at the data in your "Hello, World" file and try to find out the numbers representing the minimum, maximum and zero levels of signal.

The 8 bits/sample data is represented in an unsigned char. The zero level is 128 (0x80), the maximum value is 255 (0xff) and the minimum is 0. The fastest way to convert this kind of value to the signed byte (char) format is to XOR the value with 0x80. If you don't believe, just try it.

The 16 bits/sample data is represented using the 16 bit word format of i386 (signed short). Conversion between the signed and unsigned formats can be done by XORing the value with 0x8000.

In the 8 bit stereo mode the sample values of the both channels are interleaved together. The left channel value comes before the right one so that the stream can be represented by string LRLRLRLRLR... where the L denotes a byte for the left channel and the R for the right channel. In the 16 bit stereo mode the data is encoded as LLRRLLRRLLRR....

In 16 bits/sample stereo mode there are 4 bytes for each sample. This means that with 44.1 kHz sampling rate it will take 176 kilobytes to store one seconds of sound. One megabyte of memory can store about 5.6 seconds of sound. Keep this in mind when selecting the recording attributes for a piece of sound.

It's important to keep all the bytes of a sample (instance of time) together. For example in the 16 bits/stereo mode the sample size is 4 bytes. Ensure that the count parameter you pass to the `read` or `write` calls is always an integer multiple of 4. If you make an mistake in this, the result could be just useless noise.

NOTE! Think carefully before changing your application to work with 16 bit data. Converting 8 bit data to 16 bits don't improve the sound quality at all. It just makes the application to use more DMA buffer space.

Look at the least significant bits of your 16 bit audio data. If more than 4 bits are always zero, it could be better to use just 8 bits/sample resolution. Another (better) test is to listen the output data in 8 and 16 bit modes. If there is no audible difference in quality, it's better to use just the 8 bits/sample resolution.

The same applies to the stereo mode also. Don't use the stereo mode if the audio source is monophonic. Stereo mode requires two times more buffer space than the mono mode. Remember also that a 16 bit stereo sample takes four times more space than the 8 bit mono recording.

With the full capabilities of the 16 bit soundcards it takes 176 kb to store just 1 second of audio data. It doesn't look big when compared to the speed of for example SCSI hard disks but it's enough to cause some problems. There are just finite amount of buffers available in the driver for the audio data. In best case there is just 128k of buffers available[5] which means that the driver is able to store less than 1 seconds of data in the buffers. The application process must be able to serve the driver fast enough. In output mode it must provide the next buffer before the old ones are empty. Failure in this will cause a pause or a click in the output signal. It's even more difficult to record long samples using high speed. If the file system is not able to store the data faster than the soundcard provides it, the driver has to keep pauses in the recording. Usually a sync operation takes more than a second which means that the application could be forced to wait until the sync is complete. Since the sync operation is performed once per 30 seconds, it could be hard to record more than 10 to 20 seconds continuously using highs speed mode.

If you plan to record long samples, you could try to record into an unmounted partition (NOT HAVING A FILESYSTEM ON IT). Using a raw disk partition directly bypasses the filesystem code. It should give better performance but I'm not sure if it's enough. The problem is that no length information is recorded automaticly so you will need to use a recording program which is able to do it.

2.5 Changing the sampling parameters

In the previous sections we have studied audio programming with the default sampling parameters of the driver. By default the sound driver works using 8 bits/sample, mono and 8000 Hz. This should be suitable for recording and playing speech but not for computer music applications.

The recording and playback parameters can be changed by making some ioctl calls before calling the `read` or `write` first time. It's important that the sampling parameters are set immediately after opening the device. Changes made after the first `read` or `write` could have strange effects. With some cards the change takes effcect immediately while some others delay the change until the next sync (described some chapters later).

There are just three important `ioctl` calls available in the driver. The firts one sets the sampling resolution. The second selects the number of channels

[5] The GUS card is an exeption to this rule. There is up to 512k of output buffer space available in the card itself (512k in stereo and 256k in mono).

(mono or stereo) and the third sets the sampling rate. It's important to make these calls always in this order[6]

The ioctl calls of the `/dev/dsp` have always the same format:

```
int parm;

parm = <value>;

if (ioctl(fd, CMD, &parm)==-1) error();

if (parm differs significantly from the <value>)
{
  The soundcard is not able to support this value of the
  parameter.
}
```

The ioctl call returns -1 if there is a fatal error. For example if the CMD was not recognised by the driver.

The driver reads the value in the parameter (parm), comparese it with the limits of the hardware, changes the parameter in the device and returns back the actually used value. For example if the application tries to set too high sampling rate, the driver will use the highest available rate and to return it in the parameter. The application has to compare the returned value with the requested value. If they differ too much, the application has to give an error message.

The first step is to set the sampling resolution. The default is 8 bits/sample when the `/dev/dsp` is used and 16 bits/sample with the `/dev/dsp1`. Setting the sampling resolution is an optional step if the application knows the device file it's using. The speed could be set using the following code fragment:

```
int parm, original;

original = parm = <8 or 16>;

if (ioctl(fd, SOUND_PCM_WRITE_BITS, &parm)==-1) error();

if (parm != original)
{
```

[6] The SB Pro card supports just 22050 Hz sampling rate in stereo mode while 44100 Hz is possible in mono mode. If the 44.1 kHz speed is selected before the stereo mode, the application will be told that it's possible. The result is too slow output.

```
  The soundcard is not able to support this sampling
  resolution.
}
```

It's possible to query the current sampling resolution by calling `ioctl(fd, SOUND_PCM_READ_BITS, &parm)`.

The second step is to change the number of channels (mono or stereo). The following code fragment does it for you:

```
int parm, original;

original = parm = <1 or 2>;

if (ioctl(fd, SOUND_PCM_WRITE_CHANNELS, &parm)==-1) error();

if (parm != original)
{
  The soundcard is not able to support the requested
  number of channels (stereo mode).
}
```

It's possible to query the current number of channels by calling `ioctl(fd, SOUND_PCM_READ_CHANNELS, &parm)`.

The last step is setting the sampling rate. The following code fragment does it for you:

```
int parm, original;

original = parm = <requested speed>;

if (ioctl(fd, SOUND_PCM_WRITE_RATE, &parm)==-1) error();

if (parm differs 'significantly' from the original)
{
  The soundcard is not able to support the
  requested sampling rate.
  It could be too low or too high.

  Usually the requested value is rounded to
  the nearest speed supported by the hardware.
```

```
        The driver returns the rounded value in the parm.
    }
```

It's possible to query the current sampling rate by calling `ioctl(fd, SOUND_PCM_READ_RATE, &parm)`.

That was all you need to do. Remember to make all these calls between the `open` and the first `read` or `write`. Make the calls always in the order given above. If you have need to change the parameters later, read the section 2.6.2

The recording source selection is an important step when recording. Most of the soundcards support more than one possible input sources (mic, line input, cd, synthesizer, PC speaker etc.). It's possible to change these parameters by making some mixer ioctl calls. The mixer ioctl interface is availabla trough the `/dev/dsp` also. It's covered in the mixer programming section 1.

2.6 Advanced digitized voice programming

In the previous sections we have covered just making some simple applications. With the VoxWare version 2.4 it's possible to use some more advanced features.

2.6.1 Bidirectional mode

One of the most important ones is simultaneous recording an playback. The VoxWare driver doesn't support the bidirectional mode directly. However there are two ways to implement it. If you have a Pro Audio Spectrum card, there is actually two soundcards in it. You can use the `/dev/dsp` for one direction and the `/dev/dsp1` to the other.

It's possible to change the direction with any of the soundcards. With version 2.2 and later it's quite simple. The direction is switched automaticly when the program calls `write` after `read` or vice versa. It's possible to change the direction as many times as necessary.

There are some limitations in the second method. The first is that the reading program waits until a complete DMA buffer has been recorded. Even if just the first byte is required by the application. The second is that the sound card works just in one direction at the same time. Nothing cannot be recorded while playback is in progress.

2.6.2 Changing sampling paramaters on the fly

With version 2.2 and later it's possible to change the sampling parameters in the middle of recording or playback. It's even possible to change the direction at the same time when changing the parameters. Just call `ioctl(fd, SOUND_PCM_SYNC, 0)` before changing any of the parameters. This call waits until all output buffers have been played and resets the hardware to the state where it's ready to accept new parameters. The drawback is that there will be a short pause in the operation of the device. This may produce a click or pause to the recording or playback.

2.6.3 Realtime features

In the default mode of the `/dev/dsp` driver uses fairly long DMA buffers. This could cause problems for some applications. For example in the recording mode a read waits until there is a full buffer of data available in the device. This happens even if the process wants just to read a very short period of sampled data.

In the playback mode the device waits until there is a full buffer of output data before plays anything. It's difficult to play just a short sample which is shorter than the DMA buffer.

With version 2.2 and later it's possible to decrease the buffer size. The default buffer size is selected by the driver and it depends on the sampling paramaters. The driver tries to select the buffer longest possible buffer size which is shorter than the current data transfer rate (bytes transferred per second). This means that the default buffer can store between 0.5 and 1.0 seconds of data.

The `ioctl(fd, SOUND_PCM_SUBDIVIDE, &div)` is a new call whic requests the driver to use smaller buffers. The value in the parameter (div) should be 1, 2 or 4. After this call the driver will compute the buffer size by dividing the default buffer size by the requested value. This makes the buffer up to 4 times shorter than the default.

Sometimes it's necessary to just stop the recording or playback. The `ioctl(fd, SOUND_PCM_RESET, 0)` does it. All data in the buffers is discarded immediately and the device is put to the similar state than after SOUND_PCM_SYNC. This means that the sampling parameters and/or direction can be changed.

2.7 Some words about DMA buffering

The ADC/DAC devices of the currently supported soundcards use DMA for transferring the samples between the device and the main memory. The driver sets the address of the RAM buffer to the DMA controller and then requests the card to play the buffer. After the transfer has been initiated, the CPU is free to do something usefull. Also the process writing to the device will continue it's execution until the driver runs out of free buffers. When the block has been done, the device raises an hardware interrupt and the driver has opportunity to start the next transfer.

The time required to serve an interrupt is fairly long when compared to the sampling rate. If the driver is not able to start next transfer fast enough, there will be a sharp click in the audio output (one or more samples will be lost while recording). Some soundcards are able to auto initiate the transfer without waiting for response from the driver. With this feature there will be no clicking in the sound. Unfortunately this feature requires that the DMA controller on the PC side must be used in the auto-restart mode. This is not possible with all operating systems (BSD).

The PC bus and the DMA hardware are terribly limited. The DMA buffer must be below 16M in the physical RAM and it must be continuous. In addition both the beginning and end of the buffer must be inside the same DMA page (if the both addresses are divided by the page size, the results must be the same). The DMA page size is 64k for the 8 bit DMA channels (0 to 3) and 128k with the 16 bit ones (5 to 7). This makes it impossible to do the DMA transfer using directly the local buffer of the process. The data must be copied between the process and the DMA buffer which takes some time.

The DMA buffering gives the application some time to process the data. For example it can read more data from a file or write recorded data to a file. High speed recording to disk is sometimes little bit difficult since some filesystems are not able to allocate new blocks fast enough (particalrily the msdos and ext (original) filesystems of Linux).

If the audio data can be processed at least little bit faster than the transfer speed, there will be no pauses in the recording or playback. If the application is too slow (in long runs), it will not be possible to record or playback successfully. In this case it's recommended to use lower speed. Even using less spaceconsuming encoding (8 bits/mono) could help since copying less data takes less time.

Multiprocessing and multiuser environments like Unix have one extra problem which make it difficult to use high speed audio applications. When a process has used it's timeslice, it will put to wait until there are no higher priority processes waiting for the CPU. In worst case the process could have to wait almost a second. This raises two problems. The first one is that the application no longer has enough time to process the data. The other is that even the application needs no time to do processing, there must be enough buffer space to cover the time the process sleeps. In worst case the available space is sufficient just for about 0.18 seconds (SB16 used in the 16 bits/44.1 kHz/stereo mode).

The Gravis UltraSound (GUS) has interesting buffering features in the playback mode. The card has local RAM in the card and the playback data must be transferred to the card's memory before it can be played (there is no such feature for recording). The VoxWare driver allocates 256k of memory per channel so there is extra buffer space for at least 2.9 seconds (in addition to the DMA buffers inside the kernel). This makes the GUS almost immune to the problems caused by occasional short high priority tasks. Of cause the application must be able to produce output faster than the card plays it.

Buffering has some drawbacks when the application doesn't process the data continuously. A recording application must wait until a full buffer has been recorded by the card (even if it requires just couple of the first samples). This introduces some extra delays. Playback is easier but the application has to sync the buffers after it has written a complete fragment of sound. Solutions for these problems have been given in section 2.6.3.

In case the application wants to know the buffer size, there is an ioctl call for it. With the current version of the driver there should be no reason to care about the size.

NOTE! It's not required to call ioctl(SNDCTL_DSP_GETBLKSIZE) and to malloc the buffer using the size returned by the call. This wa required just with some of the earliest versions of the driver. Versions before 0.5 were not able to do buffer repartitioning and the application was required to do it. A good buffer size could be for example 4 or 16 kilobytes.

2.8 Future directions

The current version of the VoxWare driver supports just the 8 and 16 bit uncompressed encodings. Support for the various compression methods supported by the soundcards could be introduced some day.

Another idea is to map the DMA buffer area into the address space of the process. This removes the need to copy data between the DMA buffer and the data segment of the process. The process can use the DMA buffer area directly when processing the data. (I don't know exactly how much time it takes to copy 176 Kb of data per second but it should be pretty much).

Chapter 3

Using /dev/audio

The */dev/audio* device file is just a limited clone of the `/dev/audio` available in the Sparc workstations. The difference between it and the `/dev/dsp` is that this device file uses locarithmic μ-Law encoding of samples. The driver performs the conversion between the μ-Law and the linear 8 bit encoding automaticly while recording or playback.

There is very few use for this device file. It's in the driver just to make it easier to port applications already using the `/dev/audio`. The current implementation doesn't support the ioctl calls available in the true `/dev/audio` device file.

If you are writing an application which works only with the VoxWare driver, don't use the `/dev/audio`. You have nothing to win but everything to lose. When a voice stream is recorded using the `/dev/audio`, the driver converts the data from the native 8 bit format of the soundcard to the μ-Law format. Since the original recording is done just in 8 bits/sample mode, there is no real advantage in using the μ-Law encoding. It just consumes some amount of CPU time to perform the conversion. In playback mode the driver makes the conversion to opposite direction.

Since the μ-Law encoding is logarithmic, the conversion to the μ-Law format and back doesn't (usually) doesn't give the original value as the result. This adds some distortion to the audio signal.

In other words: Forget the `/dev/audio` if you don't have GOOD reasons to use it. Usually it's simple to convert applications using the `/dev/audio` to use the `/dev/dsp`. Aplgorithms producing μ-Law encoded data directly are very rare. Usually the application first computes a linear sample value and then converts it to μ-Law using a conversion table. It should be easy to bypass this conversion.

The situation could change in the future. The SB16 ASP has capability to do the conversion between the 16 bit linear and the μ-Law encodings in hardware. This could make the μ-Law encoding usefull if the driver gets support for the ASP chip. There are several other DSP cards which could have similar capabilities.

Chapter 4

Using /dev/sequencer

The `/dev/sequencer` is a device file which is designed to be used for computer music applications. It controls the synthesizer chips and MIDI ports of the soundcards. The VoxWare driver has capability to control several synthesizers and MIDI ports at the same time. The current version 2.4 supports two different synthesizers (FM and GUS). Both of them can be used at the same time. All of the currently supported soundcards (except the AdLib) have a MIDI port. If you have more than one soundcards, you propable have equal amount of MIDI ports.

With the version 2.4 the `/dev/sequencer` is the only way to use the synthesizers and MIDI ports. In version 3.0 (to be released later) there will be a `/dev/midi*` device for each MIDI port on the system.

4.1 Some background information

The `/dev/sequencer` is some kind of multiplexer which accepts commands or events and distributes them to the synthesizers an MIDI ports. An event is a short record which carries a command and it's parameters. In the original version of the VoxWare driver the events were all 4 bytes long. Since the GUS was introduced there was a need to include a device number to some of the events and the event size was expanded to 8 bytes. The driver is able to detect the record size so it can accept a mixture of 4 and 8 byte events.

Each event perform a simple transaction. For example they select an instrument, start playing a note or stop a note. There is a loose relation between the events and standard MIDI messages. For example the `START_NOTE` event has four parameters which are the synth device number, voice number. note number and volume/velocity. The MIDI standard defines a NOTE ON message which has three parameters: the channel number, note number and velocity. The

VoxWare driver has adopted the note number and velocity parameters from the MIDI standard but the device and voice numbers have different meaning.

The device number selects the synthesizer device to be used to play the note. There is no device number in the MIDI standard. Each device connected to the same MIDI cable will receive all messages transmitted to the cable. Implementation of the device number requires that each synthesizer is connected to the PC with a private MIDI cable.

The voice number is quite similar to a MIDI channel but there are some major differences. When playing a note using an internal synthesizer, the voice number selects the hardware level element which will be used to play the note. Each synthesizer has a limited number of voices. The original Yamaha YM3812 FM synthesizer has just 9 voices. The OPL-3 which is an improved version of the YM3812 supports 6, 12 or 18 voices depending on the mode. The GUS is able to use between 14 and 32 voices (smaller number of voices gives better sound quality). The number of available voices limits the number of simultaneously playable notes since a voice can play just one note at time.

The MIDI channel is just a virtual address. There could be several synthesizers on the same cable that listen the same midi channel. When a NOTE ON message is transmitted, each of them will play it. The other difference between a voice and a MIDI channel is that more than one notes could be playing on the same channel at the same time. The synthesizer devices listening the MIDI cable will handle the simultaneous notes using their own logic. Some devices just stop the previous note when a new NOTE ON is received. Some other are able to play all of them at the same time.

There are 16 channels in the MIDI standard. What happens when a NOTE ON message is sent is not standardized. Each synthesizer device connected to the cable can do whatever it wishes. Usually the synthesizers can be configured to listen just one of the MIDI channels. This means that it's possible to assing a private channel for up to 16 synthesizers. Usually the synthesizers are configured to listen more than one channels. When a NOTE ON message is transmitted on channel 1, it's possible that any number of synthesizers between 0 and N start playing the note. The device sending the message has no way to know what actually happens. Maintainer of the MIDI network has complete freedom to configure his devices as he wishes.

The voice parameter of the internal synthesizers is strightly defined. Exactly one device will start exactly one note. If another note is started while an eralier one is playing, the first one will be shut up. The drawback is that the playing program must be able to keep track of the free voices. It's little bit difficult since the voices could have long decay times. The voice could continue playin even several seconds after it has been requested to stop.

To make controlling of both the internal synthesizers and MIDI ports more difficult, the `/dev/sequencer` interface uses different ways to control them. The internal synthesizer devices accept some high level events such as start note or stop note. The MIDI ports accept just bytes to be transmitted to the port. The application program must construct the MIDI messages itself and then send the bytes to the port. (This will change when the level 2 interface is introduced in version 3.0).

4.2 Programming with the sequencer interface

The API for the `/dev/sequencer` interface is defined in the `sys/soundcard.h`. It must be included by all programs which use the interface.

The API is a set of C preprocessor macros which look like functions. Most of them have one or more parameters. These macros construct the event record and store them to a buffer. When the buffer becomes full, it will be written to the device. There is also a macro which sends the events currently in the buffer.

The API macros were introduced on version 2.0. There are several older programs which encode the events directly. This method works also but is not recommended. The event format is not final yet and some changes/additions are possible in the future. Use the API macros if you want to ensure source code compatibility with the future versions. In addition using them will make your programming easier.

I will not document the raw event format since it will change in version 3.0 (the programs using the old format will continue to work). If you like to batle with the low level details, just look how the API macros are implemented in the soundcard.h. I really don't recommend it.

4.2.1 Using the sequencer API

Before you start to use the API, you have include some standard definitions to your program. First you have to include the `sys/soundcard.h`. Then you have to define the file descriptor and the buffer which is used by the API. The following code fragment does it.

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

int seqfd; /* The file descriptor for the /dev/sequencer */

SEQ_DEFINEBUF(1024):
```

The `SEQ_DEFINEBUF(size)` defines a buffer whic is used by the API. The parameter is the buffer size in bytes (N*1024, N=1,2,3... is a good size). If you want to access the API from several source files, there is no ready solution for it yet. You have to declare the buffer for example on your `main.c`. Then define the same variables as externals on the other source files. The following variables are required by the API:

```
extern unsigned char _seqbuf[];
```

```
extern int _seqbuflen;
extern int _seqbufptr;
```

In addition you have to define a procedure which dumps the buffer to the device file. The following is a sample implementation:

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

The last step is to open the device file. Ensure that the device is opened before calling any procedure which uses it.

```
if ((seqfd=open("/dev/sequencer", O_RDWR, 0))==-1) die();
```

If you are writing an output only program, use `O_WRONLY`. It doesn't enable some unnecessary functions like MIDI input. The same applies to the input only programs.

4.2.2 Determining the current configuration

Since the `/dev/sequencer` is capable to support several devices at the same time, it's important that the application determines the configuration before doing anything usefull. It's not dangerous to output events for nonexistent devices but it's pretty useless.

It could be possible that the open call returns an error. The most common errors are:

- `ENOENT`: The device file is missing. It should be created using mknod(1).
- `ENODEV`: The device file is in the `/dev` directory but the driver for it is not in the kernel. It could also be possible that the major number of the device file is incorrect.
- `ENXIO`: The driver is in the kernel but there is no hardware which supports the particular device file.
- `EBUSY`: The device file is open by some other process, the physical device is already in use or the DMA/IRQ channel of the device is in use.

Some other errors could also be returned if there are some strange problems.

NOTE! The synthesizer and MIDI port numbers are not fixed. The numbers are just assigned in the order the devices were initialized during OS boot. It's bad practise to hard code them.

There are two ioctl calls which return the number of internal synthesizers and MIDI ports.

```
int ndevs;
ioctl(seqfd, SNDCTL_SEQ_NRSYNTHS, &ndevs);
```

This returns the number of internal synthesizer devices currently installed in ndevs. If your program expects a synthesizer device, there is no reason to continue if the call reports 0 devices.

The `SNDCTL_SEQ_NRMIDIS` works similarily but returns the number of MIDI ports on the system.

If the above ioctl calls return an error, it's propable that you are not accessing the right device file. It's also possible that the driver being used is an old version.

Now your program knows that there are some devices to use. There are two additional ioctl calls to determine what the devices actually are. For example if you are writing a program which works just with a particular type of device, you should check that the device is available. Also when you want to give the user a chance to selet the devices, you could use these ioctl calls. The synthesizer and MIDI port numbers can also be obtained by lookin at the printout of `cat /dev/sndstat`.

The following call returns parameters of a synthesizer device.

```
struct synth_info info;

info.device = synth_device_number;
ioctl(seqfd, SNDCTL_SYNTH_INFO, &info)
```

Ensure that you have initialized the field `info.device` before making the call. The device number must be a positive integer which is less than the value returned by `ioctl(SNDCTL_SEQ_NRSYNTHS)`.

The `synth_info` structure has the following fields. There are some additional fields but you should ignore them. If these fields have any importance, it will be defined in the section for the particular synthesizer device.

- `name`: a string containing the name of the device. This can be used if the program shows a list of available devices.
- `device`: This integer should be initialized before calling the ioctl. The value returned by the call could contain some random data.

- `synth_type` and `synth_subtype`. These integers can be used if the program wants to know what kind of device it's using. The `synth_type` could be `SYNTH_TYPE_FM` or `SYNTH_TYPE_SAMPLE` (more types could be introduced later). Each type could have several subtypes and the subtype can be used to detect the hardware type. Look at the definition of `struct synth_info` in the soundcard.h for more info.

- `nr_voices`: Integer which returns the maximum number of voices the device supports in it's current mode. If you change the mode, this value will change also. You should refer the section describing the particular synthesizer device.

Unfortunately the synthesizer devices have quite different capabilities. It's difficult to do anything but just some simple things without knowing the differences between the devices. This thing will hopefully change in the version 3.0 of the VoxWare.

The most important difference between these devices are the way how the instrument definitions are loaded to the driver. The other difference is that the devices have different number of simultaneous voices. There is also a device dependent way to change the number of voices. These things will be covered in separate sections for each type of device. The above structure gives your program a chance to determine the hardware type.

There is also a similar ioctl call for the midi devices.

```
struct midi_info info;

info.device = midi_device_number;
ioctl(seqfd, SNDCTL_MIDI_INFO, &info)
```

Ensure that you have initialized the field `info.device` before making the call. The device number must be a positive integer which is less than the value returned by `ioctl(SNDCTL_SEQ_NRMIDIS)`.

Currently just the `name` and `dev_type` fields of the `midi_info` structure are important. The device type returns the type of the soundcard where the MIDI port sits. It can be used for example when a program tries to locate a WaveBlaster synthesizer card. The WaveBlaster is a daughter card which is installed on a SB16 card. The synthesizer is accessed by sending MIDI messages to the MIDI port of the SB16 card. The MIDI interface of the SB16 has a value `SNDCARD_SB16` in the `dev_type` field. If the program doesn't find a MIDI port of this type, if could assume that the WB is not there [1].

[1] Some other cards have also the WB compatible connector so it's possible that a WaveBlaster sits on another card than SB16. Also the existence of a SB16 MIDI port doesn't guarantee that the WB daughter card is really installed.

4.2.3 General description of events

4.2.4 Event format

The /dev/sequencer device file is used for communication between the sequencer driver and the application program. The data written or read from the device is NOT raw audio data like with /dev/dsp. It's just a stream of events which control the devices. The final audible output is produced by the devices as instructed by the data stream.

The data passed through the /dev/sequencer consists of atoms called events. An event is a record having length of either 4 or 8 bytes. The first byte of the event is used to identify the event size. If the first byte is less than 128 (unsigned char!!!), the event is 4 bytes long. Other events are 8 bytes long. The exeption is the SEQ_FULLSIZE event which has 0xfd in it's first byte. This kind of events have variable length. It's not possible to cut an event in the middle and to continue it in the next write. The SEQ_FULLSIZE event must be the only event written in a write call.

The actual event format is not discussed here since the API macros take care of it. The only exeptions are the few events which could be returned by the driver. These events are described in the section 4.2.9.

The events are usually queued inside the driver and executed later by the timer routine. The driver is able to buffer about 1024 events. The purpose of the buffer is to ensure that the events are available when it comes time to execute them. If the application lets the buffer to become empty, it will propably an error to the rhythm of the tune being played. In worst case it could take more than a second before the application is given time to execute and load more events to the buffer. It should be goal of the application to keep the buffer as full as possible.

It's normal that an event is executed several seconds after the time it has been written to the device file. Sometimes the delay could be even several minutes. The application should be aware about this if it wishes for example to update the display in sync with the music (see 4.3.1).

In addition to the events there are some ioctl calls which are used to communicate with the driver. The main difference between the events and the ioctl calls is that ioctl calls are executed immediately.

4.2.5 Timing

The head of the /dev/sequencer is the event queue. The application writes the events to the tail and the timer routine reads and executes the events at the head. The events are executed as fast as possible until a time stamp event is at the head of the queue. If the time in this event has already elapsed, the playback continues without a delay. Otherwise the playback routine is exited and entered again at the time given in the time stamp event.

It's important to note that the playback process runs in the background. The application has freedom to continue it's own processing independently to the playback process. It will be forced to wait just when it tries to write to full queue or to read form empty queue. (The kernel may find some other reasons to put the process to sleep).

The sequencer device uses an absolute timer with resolution of the kernel timer (100 Hz). The times are given in 1/100th seconds (10ms) since opening the device. If you want to reset the timer afterwards, use the `SEQ_START_TIMER()` macro to send a timer start event to the device. The timer will be restarted after the event gets it's way to the head of the queue. It's important to notice that the timer is not restarted immediately.

The timing system is absolute. This scheme differes from the relative method used by some other systems where the times are time deltas between this and the previous event.

The application writes the timing events by calling the API macro:

```
SEQ_WAIT_TIME(time);
```

The 'time' parameter is a variable or a constant containing the absolute time when the driver should continue reading the event queue. The timer resolution is 10 ms.

NOTE! The above macro just writes the wait event to the local buffer of the applications (like most other macros do). It must be written to the device file using the macro `SEQ_DUMPBUF()`. The application itself (usually) continues execution without any delay after writing a wait event. This kind of timing feature cannot be used to delay execution of the application (see 4.3.1 for more details).

NOTE! It's possible to make delays to the music by using the traditional alarm clock of the Unix environment. It will not give satisfactory results (try it if you want to know why).

Although the timer resolution is 10 ms in all environments the driver runs, it could be possible that there are exeptions. If you want to ensure portability of your application, you have to chack the timer rate after opening the device. The following call:

```
int rate;

rate = 0; /* Important */
ioctl(seqfd, SNDCTL_SEQ_CTRLRATE, &rate)
```

The actual timer rate (number of ticks per second) is returned in the variable rate. Note that the parameter must be initialized to zero before making the call. It could be possible that some later versions allow changing the timer rate (not likely to happen in this way). With a zero argument the call just returns the currently active one. Using this call is little bit like overkilling since the control rate will propably be always 100 Hz. The level 2 interface of the sequencer will have adjustable timer but the implementation will be entirely different and not affect the programs written for the current interface.

4.2.6 Writing to the device

The application program has freedom to write as many events as it wishes on one write call. It just has to be aware about the event sizes and to pack the next event immediately after the previous one. For example if the first event is 4 bytes long, the second one must start at the buffer position 4. Even if it's a 8 bytes long event. The sequencer API macros handle this automaticly. The most difficult thing you have to remember is to call the `SEQ_DUMPBUF()` after writing the last event. This has to be done before the program exits or goes to wait input from the user or the `/dev/sequencer`.

Normally the application continues execution after writing events to the device. The driver will continue with the events and play them in control of the timer. Just when the application tries to write more events than there is available buffer space, it will be put to sleep. In this case it will continue execution after the number of free buffer slots reaches a configurable limit (the default is a half of the total buffer size).

4.2.7 Defining instrument characteristics

- `SEQ_WRPATCH(patch, len)`

- `SEQ_WRPATCH2(patch. len)`

4.2.8 API macros for the output events

There are few API macros which control the voices. The `SEQ_MIDIOUT` is used to send bytes to a MIDI port while the other macros control just the intenal synthesizers. The first parameter of these macros is the device number. For the `SEQ_MIDIOUT` it's the midi port where the byte is to be sent. For the other macros it's the synthesizer number.

The second parameter of other macros than `SEQ_MIDIOUT` is the voice number. A voice is capable to play just one note at the same time. The application has responsibility to allocate and deallocate the voices when required.

The following macros are supported by the current version of the VoxWare driver:

- `SEQ_MIDIOUT(midi_dev, byte)`
  This macro is the only one which controls the MIDI ports (this will change in the version 3.0). The first parameter is the MIDI port number (0 to `SNDCTL_SEQ_NRMIDIS`-1). The second one is the byte to be sent (unsigned char).

- `SEQ_SET_PATCH(dev, voice, patch)`
  Before a voice is started, the instrument number must be defined for it. The instrument number is a number between 0 and 127. See 4.2.7 for more info on instruments.

- `SEQ_START_NOTE(dev, voice, note, vel)`
  This macro starts the voice. The parameters `note` and `vel` are the MIDI note number and velocity values. Valid values for them are between 0 and 127. The note number controls the pitch of the sound. Interpretation of the velocity parameter is device dependent. Usually it controls just the volume but could control some additional characteristics also. Actually the velocity value just gives the speed or force of the key hit. Use velocity of 64 if you don't know better value.

  If the note number of 255 has special meaning. In this case the driver will not start new note but just adjusts volume of the currently playing one. This method is not recommended since it's not guaranteed to work with all soundcards.

  As well the velocity value 255 has special meaning. The driver will use the volume stored in the internal table for the voice. For some of the synthesizer devices there is a way to specify a absolute volume before starting the voice. Not recommended way to use the volumes.

- `SEQ_STOP_NOTE(dev, voice, note, vel)`
  Stops the note being played. This macro has no effect if the voice has already decayed or stopped (or if it has never been started). The `note` parameter is the MIDI note number and the `vel` parameter gives the speed of the key release.

  The current driver version ignores the note number. It's recommended to specify the note number correctly since it could be required by some coundcards in the future. Use the same note number you used when starting the voice.

  Also the velocity value has little or no use with the current driver. Use a value of 64 if you don't know better.

  NOTE! The voice is usually not stopped immediately when the NOTE OFF event is executed. The release time is a instrument specific parameter and could be more that 10 seconds. Keep this in mind when developing the voice allocation algorithm to your application. It's also a good idea to have an extra delay before closing the `/dev/sequencer` since close shuts up the all voices immediately after the event queue is empty.

- `SEQ_CHN_PRESSURE(dev, voice, pressure)`
  Some MIDI keyboards have capability to continuously monitor the pressure which the keys are pressed. Just the most expensive keyboards have capability to monitor the pressures of each key individually. Most other are capable to monitor just the overall pressure.

  The key pressure (also called aftertouch) is used by some synthesizers to enhance the sound. The most common way is to add some vibrato to the sound. The current version of the VoxWare driver is able to turn the vibrato of the OPL-3 voices on or off depending on the pressure. The pressure value is an integer between 0 and 127.

  NOTE that this macro defines the channel pressure. There is no macro (yet) for the pressures of individual keys. Since a voice can play just one note at the same time, there is no difference between these two cases. This could cause some problems when a application is converted to use the level 2 interface (version 3.0).

- `SEQ_PANNING(dev, voice, pos)`
  Both the OPL-3 and GUS are able to pan the voice between the left and right stereo channels. The GUS has 16 different pan positions while the OPL-3 has just three. The `pos` parameter is a integer (signed char) between -128 and 127 (-128=left, 0=center and 127=right). The value in this parameter is added together with the pan value given in the instrument parameters and the sum is used to select the pan position. (For OPL-3 just the instrument level parameter is used).

  The panning value could be set even before starting the note (the recommended way). The panning value remains in effect until the next note off.

  Changing the pan position while a note is playing could produce a clicking sound with some devices (GUS).

- `SEQ_CONTROL(dev, voice, controller, value)`
  The MIDI standard defines some controllers which are used to enhance the way how the notes are played. The current VoxWare version has adopted some of these controllers but the implementation differs little bit from the MIDI standard (see 4.2.8 for details).

  This macro is used to set the value of a controller. The change may be made even before starting the note. The controller is returned back to the default value when the note is stopped.

The `controller` parameter gives the controller number and the `value` gives the value assigned to the controller (see 4.2.8).

Fully MIDI compatible controllers will be introduced in the version 3.0.

- `SEQ_BENDER_RANGE(dev, voice, value)`

- `SEQ_PITCHBEND(dev, voice, value)`
  The pitch bender or pitch wheel is a control device which can be used to alter the pitch of a note continuously up or down. The `SEQ_PITCHBEND` sets the bender value for the voice. The valid bender range is between -8192 and 8191. The value can be assigned before or after starting the note. The bender is returned back to the default value (0) after the note is stopped.

  Interpretation of the bender value can be adjusted with the `SEQ_BENDER_RANGE`. By default the range is 2 semitones (midi notes) up and down. The bender range `value` is given in cents (100th of semitone). The maximum bend is two octaves (1200 cents or 12 semitones). Even a larger value is allowed for the bender range but the driver will not bend more than 2 octaves.

  If you want to bend notes given number of cents up or down, you could use a value of 8192 as the range.

- `SEQ_EXPRESSION(dev, voice, value)`

- `SEQ_MAIN_VOLUME(dev, voice, value)`
  These macros are for the expression and main volume controllers of MIDI. The implementation should follow the MIDI standard but I'm not sure. Currently implemnted just for GUS.

MIDI controllers

The MIDI specification defines a set of controllers. Similar feature is implemented by the VoxWare driver but there are some differences. The most important thing is that the MIDI controllers are channel specific. They affect all notes that play on the channel. The controller values remain in effect until they are explicitly changed by sending a control change message.

The controllers of VoxWare are voice specific and will be reset back to their defaults after the note is stopped. Values must be set every time before starting a note. In addition the numeric ranges differ from the MIDI standard.

The following controllers are known by the VoxWare 2.4:

- `CTRL_EXPRESSION`
  This is the expression controller of MIDI. It accepts values between 0 and 127.

- `CTRL_MAIN_VOLUME`
  This is the MIDI main volume controller. Valid values are between 0 and 100.

4.2.9 Reading from the `/dev/sequencer`

4.3 Advanced topics

4.3.1 Syncronizing the application with the music

4.4 Hardware level differencies

4.4.1 OPL-3 / YM3812 FM synthesizers

4.4.2 Wave table synthesizers

Gravis UltraSound (GUS)

4.5 Proposed level 2 interface specification

Chapter 5

Using /dev/midi

The `/dev/midi##` interface is a feature (to be) introduced in the version 3.0 of the driver (some earlier alpha test versions could have it also). This section is not valid for versions earlier than 3.0.

The VoxWare driver supports several MIDI interfaces at the same time. Most of the soundcards have a MIDI port. In addition the driver supports the Roland MPU-401 midi adapter. Support for MQX-32M by Music Quest and the Super MPU by Roland are under work now.

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 printout produced by `cat /dev/sndstat`. For example the device file for the first installed midi port is `/dev/midi00`.

These device files have similar capabilities than the ordinary `/dev/tty` interface. Everything written to the device will be sent to the midi port as soon as possible (there could be some earlier written bytes in the queue which delay the transmit). There are no timing features which make it difficult to use these devices for sequencer like applications. The intended use for this interface is sending and receiving system exclusive messages. For example this is required when making a patch editors and librarians for various MIDI synthesizers.

Reading from the device waits until there are 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 returns usually less 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 my 486/50 system it can receive up to about 60 bytes at a time. On a slower or heavier loaded system the read could return even more data at once.

There are couple of unnecessary delays in the current implementation, but they seems to be harmless. For example it's possible to route the incoming midi data from one port into another using `cat /dev/midi00 > /dev/midi01`. There is no annoying delays between a key-press on the keyboard and the sound on the synth connected to the `/dev/midi01`.

The `/dev/midi` interface supports `select` but currently just with Linux.

To use the raw MIDI devices, you will need some knowledge of the MIDI protocol. The official MIDI 1.0 specification is available from International MIDI Association. There are some books containing the most important parts of the protocol. In addition unofficial hacks of various MIDI specifications are available using anonymous ftp from ftp.ucsd.edu :midi/doc.

5.1 Changing parameters etc

I don't give exact details at this time since the interface will change before final release. Here are just some hints.

There is possibility to set a timeout which the process waits for the first byte. By default it waits infinitely.

5.2 Intelligent MIDI adapters

The Roland MPU-401 and some other pro-level MIDI cards have various advanced features not available in most soundcards.

For accessing the MPU-401 in it's intelligent mode, there are two ioctl calls. The first turns on the intelligent mode (or rather it resets the card and turns off the UART mode). The second one sends a command to the card. Since some commands have parameters or return some data, there will be a way to handle these situations. Unlike in some other MPU-401 drivers, VoxWare will not contain separate ioctl for each of the commands. There will be just one and the command (with parameters) are passed in the argument. The MPU-401 specific features are available only with cards supporting the intelligent mode. Several soundcards have a MPU-401 compatible interface which supports the UART mode only.

I have used the MPU-401 for recording midi bytes and it works fine. The data received from the card contains all the information returned by the card, including the MIDI data, timing bytes, MPU marks and messages. I have not tried output yet but it should be possible to make it work.

The MPU interface is not there so that everyone can program the MPU-401 itself. The `/dev/sequencer2` interface will support all the features of MPU-401 in way that is portable. Some other MIDI cards have advanced features like tape sync and the sequencer2 interface makes these features accessible in device independent way. The MPU interface is there just for making it easier to port MPU specific applications from other environments. (The main reason why it's there is that it makes it easier to study ins and outs of the MPU-401).

Using the raw MPU interface requires deep knowledge about the MPU-401 internals. There are at least two sources for this information. The first is the MPU-401 Technical Reference Manual and the other is the developers library for the Music Quest MQX-32M.

5.3 What is MIDI

The MIDI is an acronym for Musical Instrument's Digital Interface (or something like it). The MIDI 1.0 Detailed Specification defines both the hardware level interface and the communication protocol used for communication between devices having MIDI interface. It's primarily a data communication specification but also used in many other ways. I don't try to give complete description here. Just the rough idea.

The hardware level MIDI interface is like a RS-323 port but it's not plug compatible with RS-323. The MIDI cable has 5 pin DIN connectors at the ends and the transfer rate is nonstandard (31250 baud???). One cable can carry data just to one direction. Bidirectional connection requires two cables. More than two devices can be connected together by chaining the devices.

The 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 may have just 128 different values and to carry just 7 bits of information.

The first four bits of a status byte (status & 0xF0) specify the type of the status and the last 4 bits carry the midi channel number. Status bytes 0xF0 to 0xFF are reserved for system messages (the last 4 bits contain the message type).

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

For example when the player hits a key on the keyboard, a NOTE ON message is transmitted to the MIDI cable. It starts with a status byte 0x9X where the X is the channel number. There are two data bytes following the status. The first is the note number which tells which key the player has pressed. The second specifies the velocity of the keypress. The velocity is used to control the volume and some other parameters of the played sound.

It's important to notice that no sound is transferred through the MIDI line. Just instructions how the receiving instrument should control itself.

MIDI file is a file containing MIDI messages and some other data which can be used by the MIDI sequencers and other applications. It's a well defined interchange format which makes it possible to transfer songs between virtually any application supporting the format. In the PC workd these files have an extension `.MID`.

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

The .MOD files (ProTracker etc. modules) are quite similar then MIDI files. The difference is that the .MOD files contain the actual instrument sounds in the same file with the performance data of the song. Encoding of the actual song data differ between the .MID and .MOD files but the basic idea is exactly the same. There are some musical features which can be expressed better in a .MOD file than a .MID file or vice versa. The MIDI File Specification doesn't define a way to store the instrument data inside the MIDI files but it gives freedom for application the developers to implement such features.

The difference between the MIDI files and the modules (.MOD) is commonly misunderstood. People think that the modules are better than MIDI files since they sound better when played with the most common soundcards. Most DOS-based MIDI players play these files using a FM synthesizer. In the worst case just a 2 OP one. In contrast the .MOD files are played by dumping prerecorded instrument sounds into a digitized voice devices. The point is that it's possible to make similar player which plays MIDI files. With some modern wave table based synthesizer cards like GUS it's even possible. In other words THE MIDI FILES HAVE NOTHING TO DO WITH THE FM SYNTHESIS.

The advantage of the .MOD format is that the instrument sounds are carried inside the file. Currently there is no standard about how the instruments should be stored inside the MIDI files. One solution to this problem is the General MIDI standard (GM or GMidi). It for example binds the instrument/program numbers to the actual instrument names. The GM standard still gives great degree of freedom to the manufacturers. Nobody has defined how the 'Acoustic Piano' should sound. This is good since future synthesizers could implement it better than today but it makes it difficult to play a file made for a particular GM synth using another one. On the other hand a .MOD file will always sound as good or as bad as with the device used to arrange it.

Chapter 6

The patch manager interface

This interface is currently incomplete. Will be implemented fully sometimes later.

Chapter 7

Internals of the driver

This chapter is extremely incomplete. Most parts of the driver will change significantly in the near future so I don't try to describe the complete driver.

The /dev/dsp (/dev/audio) support is quite stable now so I could try to explain it little bit here. The low level MIDI driver stuff is quite stable also. There will be some additions to the interface so I dont't document it yet.

The file dev_table.h contains the most important data structures. There is a table (two tables in the next version) which contains the IRQ, DMA and I/O address configuration and provides two boot time entry points to the driver (this is the supported_drivers[] array).

The initialization code of the driver calls the probe routine of each configured card. If the probe returns != 0, the attach routine will be called. The probe routine just verifies that the card exists at the address passed in the parameter. The attach routine initializes the card and installs the low level drivers so that the higher level code is able to call it.

The dev_table.h defines also a set of arrays where the low level drivers are installed. The higher level drivers call the low level ones by calling the procedures defined in these tables.

There are 4 types of low level drivers. Each type of driver has different set of service procedures. Most of the types have open and close calls which are called when an application calls open or close for the device file. The low level driver types are:

1. Audio devices

   struct audio_operations * dsp_devs[MAX_DSP_DEV] = NULL; int num_dspdevs = 0;

2. Mixer devices (the version 2.3 supports just one mixer but this will change in the future:

struct mixer_operations * mixer_devs[MAX_MIXER_DEV] = NULL; int num_mixers = 0;

3. Synthesizer devices such as OPL-3 and GUS. Used by /dev/sequencer

   struct synth_operations * synth_devs[MAX_SYNTH_DEV] = NULL; int num_synths = 0;

4. MIDI interfaces/ports

   struct midi_operations * midi_devs[MAX_MIDI_DEV] = NULL; int num_midis = 0;

5. Timer devices

   The version 3.0 will support other timers in addition to the 100 HZ one of the kernel. These timers are used by the level 2 interface of the sequencer driver.

My current development version supports more than one mixer devices at the same time. The midi and synth support has also changed significantly and will change even more in the future.

7.1 Audio driver implementation

The high level driver for the /dev/dsp and /dev/audio device files is in the file audio.c. It contains just the code which handles the open, close, read, write and ioctl calls made by the application. The high level driver uses services of the DMA buffer manager (dmabuf.c). Write call just copies data from the application buffer to the DMA buffer and read does it in opposite direction.

The dmabuf.c does the handling of 'full' DMA buffers in both direction.

The low level audio drivers are called just when the dmabuf.c wants to start input or output of a buffer. When the playback/recording of a DMA buffer is complete, the audio driver calls the DMAbuf_inputintr or DMAbuf_ouputintr routine in the dmabuf.c whic activates the next DMA buffer.

Propably the cleanest example of a low level audio driver is the sb_dsp.c. The other drivers (particularily the GUS one) are not as clear as the sb_dsp since these cards use continuous DMA mode or have local RAM in the card. These features make it difficult to handle any exceptional situations such as partially filled DMA buffers or any kind of 'out of buffers' situations.

I hope this helps. Please contact me if you have any additional questions.

7.2 The driver/OS interface

The VoxWare driver was originally written for Linux. When the driver was ported to other environments, a special interface for this purpose was introduced.

The main idea is that just two files will need to be created when porting the driver to new OS.

- `os.h`
  This file contains a set of C preprocessor macros which look like procedures. The driver uses these macros when it wants to communicate with the operating system or the user process. These macros do tasks like memory allocation, scheduling and moving data to and from the user process. This file also includes the OS specific header files.

- `soundcard.c`
  This file contains the routines which are directly called by the operating system. Including the initialization code and the service calls (open, close, read, write etc). The procedures in this file call the the other parts of this driver (mostly in the sound_switch.c and dev_table.h).

Propably the trickiest part of porting is to get the **`sound_mem_init`** routine in the **`OS/soundcard.c`** to work. It has to be able to allocate up to 128kb RAM blocks which are DMA ready. Not an easy task.