Music Programming

Contents

Introduction

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

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

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

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

MIDI and music programming interfaces provided by OSS

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

[illegible]	[illegible]
[illegible] [illegible]	[illegible]
[illegible]	[illegible]

Fundamentals of /dev/music

MIDI (music) is a deeply real time process. An experienced listener can pick very minor timing (rythm) errors from the music he/she is listening which makes timing accuracy one of the main goals of OSS implementation. Unfortunately general purpose (multuiser and multitasking) computer systems are not well suited to this kind of tasks. For this reason OSS has been implemented in a way which makes timing precise even in highly loaded systems.

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

Queues and events

The central part of /dev/music and /dev/sequencer APIs is queueing. There is a queue both for playback and recording. Everything written to the device is first placed to the tail of the playback queue. The application continues it's execution immediately after the data is put to the queue. This happens immediately except in situations where there is not enough space in the queue for all the data. In this case the application blocks until some old data gets played. It's very important to notice that the playback is not complete when write returns. The playback process still continues in the background until all data has been played. This time depends on timing information included in the playback data and can sometimes be several minutes (even hours or days in some cases). Even after the output buffer has drained some notes not being explicitly stopped may continue playing (infinitely) until the application writes more data containing the note off command for this note.

It's very important to understand this asyncronous behaviour of the API. Even when the application tells the playback engine to wait some time (even hours) the associated write may return immediately. The application never waits until the requested time is occurred. After you understand this and have read the MIDI specification you know most important things about /dev/music and /dev/sequencer programming.

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

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

The playback engine processes the events always in order they are written to the device. However there is an ioctl call that can be used to send events immediately (ahead the queued data) to the engine. This feature is intended to be used to play real time events that occur in parallel to the "pregenerated" event stream stored in the playback queue.

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

When an input event is received from one of the devices (usually MIDI ports) the driver writes a time stamp event containing the current "real" time to the input queue and then appends an event corresponding

the data received from the device. However the timestamp is written only if it's time is different from the previously received event (to prevent the input queue from filling up unnecessarily in case of sudden input bursts). Finally the application reads both these events from the queue when it has time to process the input queue. It's possible to the application to merge the "new" received input events with the "old" playback data based on these timestamps.

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

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

MIDI ports and synthesizer devices

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

There are two main types of devices.

MIDI ports

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

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

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

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

Internal synthesizers

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

There (currently) supported synthesizer chips are the following:

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

2) Gravis Ultrasound (GUS) was the first wave table based soundcard in the market. It provides capability to play up to 32 simultaneous voices by synthesizing them from wave table samples stored on it's on board RAM (up to 8M in latest models but just 512k in the original one). The wave table capability made this card very usefull for playing so called module (.MOD, etc) music using 386 and 486 computers of early 90's. However huge increase in processing speeds of CPUs has made this approach very inpractical when compared to "mixing" in software (except when very large number of voices are used at the same time). The main problem with GUS is it's limited memory capacity which doesn't permit loading the full GM patchset simultaneously. This means that applications supporting GUS must be

able to do patch loading/caching. The driver interface originally developed for GUS defines a de facto API which is supported by other wave table device drivers (of OSS) too. This means that programs written for GUS work also with the other ones with some minor modifications.

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

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

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

The differences between internal synth and MIDI port devices

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

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

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

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

Instruments and patch caching

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

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

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

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

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

Notes

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

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

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

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

Voices and channels

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

When using the /dev/sequencer API the application needs to know how many voices are supported by the particular device. It also needs some kind of mechanism for allocating voice operators for the notes to be played. The voice number needs to be used as a parameter in all note related events sent to the driver. This task is usually very complicated due to need to handle out of voices situations. For this reason it's recommended to use the /dev/music interface which handles all this automaticly.

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

Controlling other parameters

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

Programming with /dev/music and /dev/sequencer

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

Initial steps

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

For simplisity reasons it's assumed that the OSSlib interface is being used. OSSlib is a library that

handles patch caching in almost transparent way. With OSSlib the application doesn't need to be aware about the details of particular synthesizer hardware being used.

libOSSlib.a (or libOSSlib.so in some operating systems) is distributed as a part of the commercial OSS software. Another way to obtain it is to download snd-util-3.8.tar.gz (or later) from ftp://ftp.opensound.com/ossfree and to compile it onsite. However this is recommended only with OSS/Free. To be able to compile OSSlib you should have OSS 3.8 or later installed on the system. It's also possible to compile OSSlib or applications using it by obtaining the soundcard.h file from the OSS 3.8 distribution but this is not recommended/supported.

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

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

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

An application using the /dev/sequencer or /dev/music APIs require some support code to be added in the application. All of it is present in the sample program given in the "The most fundamental /dev/midi program" section. This additional code is required to support buffering used by the SEQ_* macros defined in soundcard.h.. The following has to be present:

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

2) Define the buffer being used by the API.

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

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

/*
 * Standard includes
 */

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

/*
 * This program uses just one output device which is defined by the
following
 * macro. However the OSS API permits using any number of devices
 * simultaneously.
 */
#define MY_DEVICE 0 /* 0 is the first available device */

/*
 * The OSS API macros assume that the file descriptor of /dev/music
 * (or /dev/sequencer) is stored in variable called seqfd. It has to be
 * defined in one source file. Other source files in the same application
 * should define it extern.
 */
int seqfd=-1;

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

/*
 * seqbuf_dump() routine is required only wnen OSSLib is NOT used. It's
 * purpose is to write buffered events to the device file.
 */

#ifndef OSSLIB
/*
 * NOTE! Don't ever define seqbuf_dump() in two source files or when OSSlib
 *       is used. It may have unpredictable results.
 */
void
seqbuf_dump ()
{
  if (_seqbufptr)
    if (write (seqfd, _seqbuf, _seqbufptr) == -1)
      {
```

```
    perror ("write /dev/music");
    exit (-1);
        }
    _seqbufptr = 0;
}
#endif
```

Opening the device

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

```
    int error, ndevices, tmp;

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

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


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

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

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

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

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

Writing events

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

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

The most fundamental /dev/midi program

By combining the above four code fragments together the following one you get all the necessary initialization code required in a program using /dev/music (or /dev/sequencer). All this program does is playing a note using the selected device. This program is also available in the samples.tar.gz package available from ftp://ftp.opensound.com/ossfree.

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

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

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

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

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

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

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

  /*
   * Now stop the note. The sound will not stop immediately. The note
   * just starts decaying and fades off.
```

```
 */
SEQ_STOP_NOTE(MY_DEVICE, 0, 60, 64);

/*
 * Have a final delay of 1000 ticks. This gives the last note(s) time
 * to decay naturally. Closing the device without this delay just
 * aborts all voices prematurely.
 */
SEQ_DELTA_TIME(1000);
/*
        * Finally flush all events still in the local buffer (mandatory
 * step before closing the device or prior pausing the application.
 * It's the SEQ_DUMPBUF() call that actually writes the events to the
        * device.
 */

SEQ_DUMPBUF();
close(seqfd);
exit(0);
}
```