ALSA Howto dev

Willow75

ALSA Howto dev
par Willow75

Table des matières

Chapitre 1. Introduction

Ce HOWTO donne une brève introduction de l'écriture d'applications audio simple avec ALSA.

Le chapitre 2, explique les fonctions les plus fondamentales pour l'audio PCM. Si vous enlevez les explications, vous aurez un programme simple pour la lecture PCM.

Le chapitre 3, traite brièvement des fonctions pour la capture PCM.

Le chapitre 4 vous apprend à écrire un simple client pour le séquenceur d'ALSA. Un programme, basé sur l'exemple seqdemo.c, qui peut recevoir des événements MIDI et qui montre les types d'événement les plus importants.

Le chapitre 5, montre comment le séquenceur MIDI ALSA peut être utilisé pour router des événements d'un port entré MIDI à plusieurs ports sortie, basé sur l'exemple midiroute.c.

Le chapitre 6, combine les entrées PCM playback et MIDI, et détaille le synthétiseur miniFMsynth.c. Ce chapitre présente la lecture audio basé sur le callback, comme proposé par Paul Davis sur la mailing liste linux-audio-dev.

Le chapitre 7, fournit une petit introduction au programmeur de tâches( ) MIDI avec les files d'attente du séquenceur d'ALSA, exemple basées sur le petit arpeggiator miniArp.c.

Il est recommandé de lire également la doc, produite par doxygen, de l'API ALSA pour bien maitriser, ceci ne reste qu'une prise en main.

Compilation d'une application ALSA : utilisez juste

```
-lasound
```

et assurez vous d'avoir écris dans votre programme

```
#include <alsa/asoundlib.h>
```

Chapitre 2. Basic PCM audio

Pour écrire une simple application PCM pour ALSA nous avons besoin d'abord d'un handle pour le dispositif PCM. Nous devons indiquer la direction du stream PCM, qui peut être playback ou capture. Nous devons également fournir quelques informations au sujet de la configuration que nous voulons employer, comme la taille du buffer, le taux d'échantillonnage, le format de données PCM. Ainsi, d'abord nous écrirons :

```
/* Handle for the PCM device */
snd_pcm_t *pcm_handle;

/* Playback stream */
snd_pcm_stream_t stream = SND_PCM_STREAM_PLAYBACK;

/* This structure contains information about    */
/* the hardware and can be used to specify the  */
/* configuration to be used for the PCM stream. */
snd_pcm_hw_params_t *hwparams;
```

Les interfaces ALSA les plus importantes des dispositifs PCM sont "plughw" et "hw". Si vous utilisez l'interface "plughw", vous n'avez pas besoin de vous préocupper des carte sons. Si votre carte son ne supporte pas le taux d'échantillonnage ou le format d'échantillon que vous spécifiez, vos données seront automatiquement converties. Ceci s'applique également au type d'accès et au nombre de canaux.

Avec l'interface "hw", vous devez vérifier que votre matériel supporte les paramètres que vous voulez utilisés.

```
/* Name of the PCM device, like plughw:0,0          */
/* The first number is the number of the soundcard, */
/* the second number is the number of the device.   */
char *pcm_name;
```

Alors on initialise les variables et on assigne une structure hwparams

```
/* Init pcm_name. Of course, later you */
/* will make this configurable ;-)     */
pcm_name = strdup("plughw:0,0");
```

```
/* Allocate the snd_pcm_hw_params_t structure on the stack. */
snd_pcm_hw_params_alloca(&hwparams);
```

Maintenant nous allons ouvrir l'interface PCM

```
/* Open PCM. The last parameter of this function is the mode. */
/* If this is set to 0, the standard mode is used. Possible   */
/* other values are SND_PCM_NONBLOCK and SND_PCM_ASYNC.       */
/* If SND_PCM_NONBLOCK is used, read / write access to the    */
/* PCM device will return immediately. If SND_PCM_ASYNC is    */
/* specified, SIGIO will be emitted whenever a period has     */
/* been completely processed by the soundcard.                */
if (snd_pcm_open(&pcm_handle, pcm_name, stream, 0) < 0) {
  fprintf(stderr, "Error opening PCM device %s\n", pcm_name);
  return(-1);
}
```

Avant de pouvoir écrire des données PCM dans la carte son, nous devons indiquer le type d'accès, le format d'échantillon, le taux d'échantillonnage, le nombre de canaux, le nombre de périodes et la taille de la période.

D'abord, il faut initialiser la structure hwparams avec la configuration totale de l'espace de la carte son.

```
/* Init hwparams with full configuration space */
if (snd_pcm_hw_params_any(pcm_handle, hwparams) < 0) {
  fprintf(stderr, "Can not configure this PCM device.\n");
  return(-1);
}
```

Des informations sur les configurations possibles peuvent être obtenues grâce à un ensemble de fonctions :

```
snd_pcm_hw_params_can_<capability>
snd_pcm_hw_params_is_<property>
snd_pcm_hw_params_get_<parameter>
```

Les paramètres disponibles les plus importants, les types d'accès, la taille du buffer, le nombre de canaux, le format d'échantillon, le taux d'échantillonnage et le nombre de périodes, peuvent être affichés avec l'ensemble de fonctions :

```
snd_pcm_hw_params_test_<parameter>
```

Ces fonctions sont particulièrement importantes si l'interface "hw" est utilisé. L'espace de configuration peut être limité à une certaine configuration grâce à l'ensemble de fonctions :

```
snd_pcm_hw_params_set_<parameter>
```

Pour cet exemple, nous supposons que la carte son peut être configurée pour la lecture 16 bit stéréo échantillonée à 44100 hertz. Donc, il faudra déterminer l'espace de configuration :

```
    int rate = 44100; /* Sample rate */
    int exact_rate;   /* Sample rate returned by */
                      /* snd_pcm_hw_params_set_rate_near */
    int dir;          /* exact_rate == rate --> dir = 0 */
                      /* exact_rate < rate   --> dir = -1 */
                      /* exact_rate > rate  --> dir = 1 */
    int periods = 2;       /* Number of periods */
    int periodsize = 8192; /* Periodsize (bytes) */
```

Le type d'accès indique la manière dont les données multicanales sont stockées dans le buffer. Pour l'accès INTERLEAVED, chaque frame du buffer contient les données consécutives d'échantillon pour les canaux. Pour des données 16 bit stéréo, ceci signifie que le buffer contient des caractères alternatifs de données d'échantillon pour le canal gauche et droit.

Pour l'accès NONINTERLEAVED, chaque période contient d'abord toutes les données d'échantillon pour le premier canal suivi des données d'échantillon pour le deuxième canal et ainsi de suite.

```
    /* Set access type. This can be either    */
    /* SND_PCM_ACCESS_RW_INTERLEAVED or       */
    /* SND_PCM_ACCESS_RW_NONINTERLEAVED.      */
    /* There are also access types for MMAPed */
    /* access, but this is beyond the scope   */
    /* of this introduction.                  */
```

```
if (snd_pcm_hw_params_set_access(pcm_handle, hwparams, SND_PCM_ACCESS_RW_INTERLEAVED) <
  fprintf(stderr, "Error setting access.\n");
  return(-1);
}

/* Set sample format */
if (snd_pcm_hw_params_set_format(pcm_handle, hwparams, SND_PCM_FORMAT_S16_LE) < 0) {
  fprintf(stderr, "Error setting format.\n");
  return(-1);
}

/* Set sample rate. If the exact rate is not supported */
/* by the hardware, use nearest possible rate.         */
exact_rate = snd_pcm_hw_params_set_rate_near(pcm_handle, hwparams, rate, &dir);
if (dir != 0) {
  fprintf(stderr, "The rate %d Hz is not supported by your hardware.\n
                  ==> Using %d Hz instead.\n", rate, exact_rate);
}

/* Set number of channels */
if (snd_pcm_hw_params_set_channels(pcm_handle, hwparams, 2) < 0) {
  fprintf(stderr, "Error setting channels.\n");
  return(-1);
}

/* Set number of periods. Periods used to be called fragments. */
if (snd_pcm_hw_params_set_periods(pcm_handle, hwparams, periods, 0) < 0) {
  fprintf(stderr, "Error setting periods.\n");
  return(-1);
}
```

L’unité de la taille du buffer dépend de la fonction. Parfois elle est donnée en bytes, parfois le nombre de frames doit être indiqué. Une frame réprésente le vecteur de données d’échantillon pour tous les canaux. Pour des données 16 bit stéréo, une frame a une longueur de quatre bytes.

```
/* Set buffer size (in frames). The resulting latency is given by */
/* latency = periodsize * periods / (rate * bytes_per_frame)     */
if (snd_pcm_hw_params_set_buffer_size(pcm_handle, hwparams, (periodsize * periods)>>2) <
  fprintf(stderr, "Error setting buffersize.\n");
  return(-1);
}
```

Si votre matériel ne supporte pas une taille de buffer de 2^n, vous pouvez utiliser la fonction :

```
snd_pcm_hw_params_set_buffer_size_near
```

Ceci fonctionne de la même façon que :

```
snd_pcm_hw_params_set_rate_near
```

Maintenant appliquons la configuration du dispositif PCM indiqué par pcm_handle. Ceci préparera également le dispositif PCM.

```
/* Apply HW parameter settings to */
/* PCM device and prepare device  */
if (snd_pcm_hw_params(pcm_handle, hwparams) < 0) {
  fprintf(stderr, "Error setting HW params.\n");
  return(-1);
}
```

Après que le dispositif PCM soit configuré, nous pouvons commencer à écrire des données PCM. Les premiers accès en écriture lancerons le PCM playback. Pour l'accès en écriture de type interleaved, nous utiliserons la fonction :

```
/* Write num_frames frames from buffer data to    */
/* the PCM device pointed to by pcm_handle.       */
/* Returns the number of frames actually written. */
snd_pcm_sframes_t snd_pcm_writei(pcm_handle, data, num_frames);
```

Pour le type noninterleaved

```
/* Write num_frames frames from buffer data to    */
/* the PCM device pointed to by pcm_handle.       */
/* Returns the number of frames actually written. */
snd_pcm_sframes_t snd_pcm_writen(pcm_handle, data, num_frames);
```

Après l'ouverture de PCM playback, nous devons nous assurer que notre application envoie assez de données au buffer de la carte son. Autrement, un buffer underrun (survient quand la mémoire tampon se vide) se produira. Après qu'un underrun se soit produit, le snd_pcm_prepare devra être utilisé.

```
unsigned char *data;
int pcmreturn, l1, l2;
short s1, s2;
int frames;

data = (unsigned char *)malloc(periodsize);
frames = periodsize >> 2;
for(l1 = 0; l1 < 100; l1++) {
  for(l2 = 0; l2 < num_frames; l2++) {
    s1 = (l2 % 128) * 100 - 5000;
    s2 = (l2 % 256) * 100 - 5000;
    data[4*l2] = (unsigned char)s1;
    data[4*l2+1] = s1 >> 8;
    data[4*l2+2] = (unsigned char)s2;
    data[4*l2+3] = s2 >> 8;
  }
  while ((pcmreturn = snd_pcm_writei(pcm_handle, data, frames)) < 0) {
    snd_pcm_prepare(pcm_handle);
    fprintf(stderr, "------------------------------ Buffer Underrun--------------------
  }
}
```

Pour arrêter la lecture, nous pouvons utiliser :

```
snd_pcm_drop
```

ou

```
snd_pcm_drain
```

La première fonction arrêtera immédiatement la lecture et détruira les frames en attente. La deuxième fonction arretera la lecture après que les frames restante aient été jouées.

```
/* Stop PCM device and drop pending frames */
```

```
snd_pcm_drop(pcm_handle);

/* Stop PCM device after pending frames have been played */
snd_pcm_drain(pcm_handle);
```

Chapitre 3. PCM capture

Il n’est pas possible d’utiliser un pcm_handle pour la lecture et la capture. Ainsi vous devez configurer deux handles si vous voulez accéder au dispositif PCM dans les deux directions. La fonction :

```
snd_pcm_open
```

doit être défini avec le stream réglé en

```
snd_pcm_stream_capture.
```

```
/* Capture stream */
snd_pcm_stream_t stream_capture = SND_PCM_STREAM_CAPTURE;
```

Les autres paramètres sont identique à ceux pour la lecture. Pour le mode de capture interleaved, il faut

```
/* Read num_frames frames from the PCM device  */
/* pointed to by pcm_handle to buffer capdata. */
/* Returns the number of frames actually read. */
snd_pcm_readi(pcm_capture_handle, capdata, num_frames);
```

Pour noninterleaved

```
/* Read num_frames frames from the PCM device  */
/* pointed to by pcm_handle to buffer capdata. */
/* Returns the number of frames actually read. */
snd_pcm_readn(pcm_capture_handle, capdata, num_frames);
```

Comme dans le cas de la lecture, nous devons faire attention que l'application appelle la fonction lecture avant que le buffer capture de la carte son soit complètement rempli. Autrement il y aura un buffer overrun (dépassement de la mémoire tampon).

```
int pcmreturn;

while ((pcmreturn = snd_pcm_readi(pcm_capture_handle, capdata, periodsize>>2)) < 0) {
  snd_pcm_prepare(pcm_capture_handle);
  fprintf(stderr, "------------------------------------ Buffer Overrun ---------------
}
```

Chapitre 4. Ecrire un client pour le sequencer

L’exemple seqdemo.c montre comment créer un simple client pour le séquenceur ALSA. Ce client peut recevoir des données MIDI d’autres clients et il affiche NOTE ON/OFF, 7-bit CONTROLLER and PITCHBENDER events.

```
/* seqdemo.c by Matthias Nagorni */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <alsa/asoundlib.h>

snd_seq_t *open_seq();
void midi_action(snd_seq_t *seq_handle);

snd_seq_t *open_seq() {

  snd_seq_t *seq_handle;
  int portid;

  if (snd_seq_open(&seq_handle, "default", SND_SEQ_OPEN_INPUT, 0) < 0) {
    fprintf(stderr, "Error opening ALSA sequencer.\n");
    exit(1);
  }
  snd_seq_set_client_name(seq_handle, "ALSA Sequencer Demo");
  if ((portid = snd_seq_create_simple_port(seq_handle, "ALSA Sequencer Demo",
            SND_SEQ_PORT_CAP_WRITE|SND_SEQ_PORT_CAP_SUBS_WRITE,
            SND_SEQ_PORT_TYPE_APPLICATION)) < 0) {
    fprintf(stderr, "Error creating sequencer port.\n");
    exit(1);
  }
  return(seq_handle);
}

void midi_action(snd_seq_t *seq_handle) {

  snd_seq_event_t *ev;

  do {
    snd_seq_event_input(seq_handle, &ev);
    switch (ev->type) {
      case SND_SEQ_EVENT_CONTROLLER:
        fprintf(stderr, "Control event on Channel %2d: %5d       \r",
                ev->data.control.channel, ev->data.control.value);
        break;
      case SND_SEQ_EVENT_PITCHBEND:
        fprintf(stderr, "Pitchbender event on Channel %2d: %5d   \r",
                ev->data.control.channel, ev->data.control.value);
```

```
        break;
      case SND_SEQ_EVENT_NOTEON:
        fprintf(stderr, "Note On event on Channel %2d: %5d       \r",
                ev->data.control.channel, ev->data.note.note);
        break;
      case SND_SEQ_EVENT_NOTEOFF:
        fprintf(stderr, "Note Off event on Channel %2d: %5d      \r",
                ev->data.control.channel, ev->data.note.note);
        break;
    }
    snd_seq_free_event(ev);
  } while (snd_seq_event_input_pending(seq_handle, 0) > 0);
}

int main(int argc, char *argv[]) {

  snd_seq_t *seq_handle;
  int npfd;
  struct pollfd *pfd;

  seq_handle = open_seq();
  npfd = snd_seq_poll_descriptors_count(seq_handle, POLLIN);
  pfd = (struct pollfd *)alloca(npfd * sizeof(struct pollfd));
  snd_seq_poll_descriptors(seq_handle, pfd, npfd, POLLIN);
  while (1) {
    if (poll(pfd, npfd, 100000) > 0) {
      midi_action(seq_handle);
    }
  }
}
```

Détaillons un peu ce programme

A)

```
snd_seq_t *open_seq()
```

Cette fonction utilise snd_seq_open pour créer un nouveau client pour le séquenceur ALSA. Puisque nous voulons seulement recevoir des événements MIDI, le paramètre de stream est défini à

```
snd_seq_open_input
```

Le client obtient un nom en utilisant

```
snd_seq_set_client_name
```

Maintenant nous avons besoin d'un port qui peut recevoir des événements MIDI. Nous utilisons donc

```
snd_seq_create_simple_port
```

Nous devons indiquer les possibilités du port pour permettre l'accès en écriture,

```
snd_seq_port_cap_write
```

et pour permettre à d'autres clients de l'utiliser, de souscrire, au port en tant que port d'écriture.

```
snd_seq_port_cap_subs_write
```

Puisque notre application peut lire à partir de port, nous devrions ajouter snd_seq_cap_read, mais pour le moment, ceci peut être omis.

B)

```
void midi_action(snd_seq_t *seq_handle)
```

Cette fonction récupére un événement à partir de l'entrée du buffer du séquenceur ALSA en utlisant

```
snd_seq_event_input
```

L'événement est traité puis libéré avec

```
snd_seq_free_event
```

La boucle est répétée jusqu’à ce que l’entrée du buffer soit vide. Ceci est controlé avec le renvoi de la taille en byte des événements restants dans l’entrée du buffer :

```
snd_seq_input_pending
```

C)

```
int main(int argc, char *argv[])
```

Cette exemple utilise le "polling" pour controler les événements MIDI arrivant.

```
snd_seq_poll_descriptors_count
```

renvoi le nombre de poll descripteurs pour les événements polling entrant(paramètre POLLIN), pour l’instant le nombre est toujours 1.

```
struct pollfd
```

est assigné sur la pile et initialisé en utilisant

```
snd_seq_poll_descriptors
```

L’utilisation de poll renvoie un nombre seulement si un événement MIDI peut être lu à partir du séquenceur ALSA ou après timeout. Vous pouvez lire le manuel de poll si celui ci ne vous est pas familier.

Chapitre 5. Un routeur MIDI

ALSA rend très facile l'implementation de n'importe quel routeur MIDI que vous pouvez imaginer. midiroute.c fournit un exemple simple.

```
/* midiroute.c by Matthias Nagorni */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <alsa/asoundlib.h>

#define MAX_MIDI_PORTS   4

int open_seq(snd_seq_t **seq_handle, int in_ports[], int out_ports[], int num_in, int num_o
void midi_route(snd_seq_t *seq_handle, int out_ports[], int split_point);

/* Open ALSA sequencer wit num_in writeable ports and num_out readable ports. */
/* The sequencer handle and the port IDs are returned.                        */
int open_seq(snd_seq_t **seq_handle, int in_ports[], int out_ports[], int num_in, int num_o

  int l1;
  char portname[64];

  if (snd_seq_open(seq_handle, "default", SND_SEQ_OPEN_DUPLEX, 0) < 0) {
    fprintf(stderr, "Error opening ALSA sequencer.\n");
    return(-1);
  }
  snd_seq_set_client_name(*seq_handle, "MIDI Router");
  for (l1 = 0; l1 < num_in; l1++) {
    sprintf(portname, "MIDI Router IN %d", l1);
    if ((in_ports[l1] = snd_seq_create_simple_port(*seq_handle, portname,
              SND_SEQ_PORT_CAP_WRITE|SND_SEQ_PORT_CAP_SUBS_WRITE,
              SND_SEQ_PORT_TYPE_APPLICATION)) < 0) {
      fprintf(stderr, "Error creating sequencer port.\n");
      return(-1);
    }
  }
  for (l1 = 0; l1 < num_out; l1++) {
    sprintf(portname, "MIDI Router OUT %d", l1);
    if ((out_ports[l1] = snd_seq_create_simple_port(*seq_handle, portname,
              SND_SEQ_PORT_CAP_READ|SND_SEQ_PORT_CAP_SUBS_READ,
              SND_SEQ_PORT_TYPE_APPLICATION)) < 0) {
      fprintf(stderr, "Error creating sequencer port.\n");
      return(-1);
    }
  }
  return(0);
}
```

```
/* Read events from writeable port and route them to readable port 0  */
/* if NOTEON / OFF event with note < split_point. NOTEON / OFF events */
/* with note >= split_point are routed to readable port 1. All other  */
/* events are routed to both readable ports.                          */
void midi_route(snd_seq_t *seq_handle, int out_ports[], int split_point) {

  snd_seq_event_t *ev;

  do {
    snd_seq_event_input(seq_handle, &ev);
    snd_seq_ev_set_subs(ev);
    snd_seq_ev_set_direct(ev);
    if ((ev->type == SND_SEQ_EVENT_NOTEON)||(ev->type == SND_SEQ_EVENT_NOTEOFF)) {
      if (ev->data.note.note < split_point) {
        snd_seq_ev_set_source(ev, out_ports[0]);
      } else {
        snd_seq_ev_set_source(ev, out_ports[1]);
      }
      snd_seq_event_output_direct(seq_handle, ev);
    } else {
      snd_seq_ev_set_source(ev, out_ports[0]);
      snd_seq_event_output_direct(seq_handle, ev);
      snd_seq_ev_set_source(ev, out_ports[1]);
      snd_seq_event_output_direct(seq_handle, ev);
    }
    snd_seq_free_event(ev);
  } while (snd_seq_event_input_pending(seq_handle, 0) > 0);
}

int main(int argc, char *argv[]) {

  snd_seq_t *seq_handle;
  int in_ports[MAX_MIDI_PORTS], out_ports[MAX_MIDI_PORTS];
  int npfd, split_point;
  struct pollfd *pfd;

  if (argc < 2) {
    fprintf(stderr, "\nmidiroute <split_point>\n\n");
    exit(1);
  } else {
    split_point = atoi(argv[1]);
  }
  if (open_seq(&seq_handle, in_ports, out_ports, 1, 2) < 0) {
    fprintf(stderr, "ALSA Error.\n");
    exit(1);
  }
  npfd = snd_seq_poll_descriptors_count(seq_handle, POLLIN);
  pfd = (struct pollfd *)alloca(npfd * sizeof(struct pollfd));
  snd_seq_poll_descriptors(seq_handle, pfd, npfd, POLLIN);
  while (1) {
    if (poll(pfd, npfd, 100000) > 0) {
      midi_route(seq_handle, out_ports, split_point);
```

```
    }
  }
}
```

Le programme crée un port d’entré et deux ports de sortie, il récupére le paramètre split_point. Les événements d’entré de type note qui ont la note < split_point sont routés au premier port de sortie, tous les autres événements de note sont conduits au deuxième port de sortie. Tous les autres événements (les événements de controle, de pitch,...) sont routés simultanément vers les deux ports de sortie.

Naturellement vous pouvez facilement modifier ceci de diverses manières en changeant seulement la condition "if" de la fonction midi_route. Vous pouvez avoir différent instrument joué en fonction de la vélocité, regardons comment cela fonctionne.

A)

```
int open_seq(snd_seq_t **seq_handle, int in_ports[], int out_ports[], int num_in, int n
```

Ici, nous ouvrons le séquenceur ALSA avec

```
snd_seq_open_duplex
```

Puisque nous voulons lire et écrire des événements MIDI, nous avons besoin des ID des ports pour pouvoir distinguer les différents ports de sortie, et nous devons stocker la valeur de retour de snd_seq_create_simple_port, dans cette partie les ports de sortie sont également créés .

Pour que d’autres clients puissent lire à partir de ces sorties, nous devons permettre l’accès read et l’accés read subscription. Il n’est pas nécessaire de permettre l’accès write, bien que l’application elle-même écrive dans le port de sortie. Actuellement, un client peut toujours accéder aux ports sans permission explicite.

```
void midi_route(snd_seq_t *seq_handle, int out_ports[], int pick_channel)
```

Comme dans l’exemple précédent le snd_seq_event_input est utilisé pour lire des événements à partir de l’entrée du buffer. Maintenant nous voudrions router l’événement à un des ports de sortie. Puisqu’il est

plus flexible de permettre arbitrairement aux clients de souscrire aux ports de sortie et de recevoir les événements, nous n’indiquons pas explicitement la destination avec

```
snd_seq_ev_set_dest
```

Au lieu de cela, nous utilisons

```
snd_seq_ev_set_source
```

pour créer l’événement qui provient du port de sortie désiré. La fonction

```
snd_seq_ev_set_subs
```

défini tous les "abonnés" des port de sortie. Nous voulons produire l’événement aussi rapidement que possible, par conséquent, nous utiliserons

```
snd_seq_ev_set_direct
```

pour contourner la file d’attente d’événements, ceci défini la file d’attente pour l’événement snd_seq_queue_direct. Puis nous utilisons snd_seq_event_output_direct pour la sortie unbuffered de l’événement.

En modifiant les arguments de la condition "if", vous pouvez facilement mettre en application une routeur différent, par exemple

```
if (ev->data.control.channel < split_channel)  /*channel splitting*/
if (ev->data.note.velocity < threshold)  velocity dependence*/
if (ev->type == SND_SEQ_EVENT_CONTROLLER)  /*extract controller events*/
```

Chapitre 6. Combining PCM and MIDI: miniFMsynth

Le synthétiseur miniFMsynth.c montre le traitement d'événement MIDI et la lecture PCM.

```
/* miniFMsynth 1.0 by Matthias Nagorni    */
/* This program uses callback-based audio */
/* playback as proposed by Paul Davis on  */
/* the linux-audio-dev mailinglist.       */

#include <stdio.h>
#include <stdlib.h>
#include <alsa/asoundlib.h>
#include <math.h>

#define POLY 10
#define GAIN 5000.0
#define BUFSIZE 512

snd_seq_t *seq_handle;
snd_pcm_t *playback_handle;
short *buf;
double phi[POLY], phi_mod[POLY], pitch, modulation, velocity[POLY], attack, decay, sustain,
int harmonic, subharmonic, transpose, note[POLY], gate[POLY], note_active[POLY];

snd_seq_t *open_seq() {

    snd_seq_t *seq_handle;

    if (snd_seq_open(&seq_handle, "default", SND_SEQ_OPEN_DUPLEX, 0) < 0) {
        fprintf(stderr, "Error opening ALSA sequencer.\n");
        exit(1);
    }
    snd_seq_set_client_name(seq_handle, "miniFMsynth");
    if (snd_seq_create_simple_port(seq_handle, "miniFMsynth",
        SND_SEQ_PORT_CAP_WRITE|SND_SEQ_PORT_CAP_SUBS_WRITE,
        SND_SEQ_PORT_TYPE_APPLICATION) < 0) {
        fprintf(stderr, "Error creating sequencer port.\n");
        exit(1);
    }
    return(seq_handle);
}

snd_pcm_t *open_pcm(char *pcm_name) {

    snd_pcm_t *playback_handle;
    snd_pcm_hw_params_t *hw_params;
    snd_pcm_sw_params_t *sw_params;
```

```
    if (snd_pcm_open (&playback_handle, pcm_name, SND_PCM_STREAM_PLAYBACK, 0) < 0) {
        fprintf (stderr, "cannot open audio device %s\n", pcm_name);
        exit (1);
    }
    snd_pcm_hw_params_alloca(&hw_params);
    snd_pcm_hw_params_any(playback_handle, hw_params);
    snd_pcm_hw_params_set_access(playback_handle, hw_params, SND_PCM_ACCESS_RW_INTERLEAVED)
    snd_pcm_hw_params_set_format(playback_handle, hw_params, SND_PCM_FORMAT_S16_LE);
    snd_pcm_hw_params_set_rate_near(playback_handle, hw_params, 44100, 0);
    snd_pcm_hw_params_set_channels(playback_handle, hw_params, 2);
    snd_pcm_hw_params_set_periods(playback_handle, hw_params, 2, 0);
    snd_pcm_hw_params_set_period_size(playback_handle, hw_params, BUFSIZE, 0);
    snd_pcm_hw_params(playback_handle, hw_params);
    snd_pcm_sw_params_alloca(&sw_params);
    snd_pcm_sw_params_current(playback_handle, sw_params);
    snd_pcm_sw_params_set_avail_min(playback_handle, sw_params, BUFSIZE);
    snd_pcm_sw_params(playback_handle, sw_params);
    return(playback_handle);
}

double envelope(int *note_active, int gate, double *env_level, double t, double attack, dou

    if (gate)  {
        if (t > attack + decay) return(*env_level = sustain);
        if (t > attack) return(*env_level = 1.0 - (1.0 - sustain) * (t - attack) / decay);
        return(*env_level = t / attack);
    } else {
        if (t > release) {
            if (note_active) *note_active = 0;
            return(*env_level = 0);
        }
        return(*env_level * (1.0 - t / release));
    }
}

int midi_callback() {

    snd_seq_event_t *ev;
    int l1;

    do {
        snd_seq_event_input(seq_handle, &ev);
        switch (ev->type) {
            case SND_SEQ_EVENT_PITCHBEND:
                pitch = (double)ev->data.control.value / 8192.0;
                break;
            case SND_SEQ_EVENT_CONTROLLER:
                if (ev->data.control.param == 1) {
                    modulation = (double)ev->data.control.value / 10.0;
                }
                break;
            case SND_SEQ_EVENT_NOTEON:
```

```
                for (l1 = 0; l1 < POLY; l1++) {
                    if (!note_active[l1]) {
                        note[l1] = ev->data.note.note;
                        velocity[l1] = ev->data.note.velocity / 127.0;
                        env_time[l1] = 0;
                        gate[l1] = 1;
                        note_active[l1] = 1;
                        break;
                    }
                }
                break;
            case SND_SEQ_EVENT_NOTEOFF:
                for (l1 = 0; l1 & POLY; l1++) {
                    if (gate[l1] && note_active[l1] && (note[l1] == ev->data.note.note)) {
                        env_time[l1] = 0;
                        gate[l1] = 0;
                    }
                }
                break;
        }
        snd_seq_free_event(ev);
    } while (snd_seq_event_input_pending(seq_handle, 0) > 0);
    return (0);
}

int playback_callback (snd_pcm_sframes_t nframes) {

    int l1, l2;
    double dphi, dphi_mod, f1, f2, f3, freq_note, sound;

    memset(buf, 0, nframes * 4);
    for (l2 = 0; l2 < POLY; l2++) {
        if (note_active[l2]) {
            f1 = 8.176 * exp((double)(transpose+note[l2]-2)*log(2.0)/12.0);
            f2 = 8.176 * exp((double)(transpose+note[l2])*log(2.0)/12.0);
            f3 = 8.176 * exp((double)(transpose+note[l2]+2)*log(2.0)/12.0);
            freq_note = (pitch > 0) ? f2 + (f3-f2)*pitch : f2 + (f2-f1)*pitch;
            dphi = M_PI * freq_note / 22050.0;
            dphi_mod = dphi * (double)harmonic / (double)subharmonic;
            for (l1 = 0; l1 < nframes; l1++) {
                phi[l2] += dphi;
                phi_mod[l2] += dphi_mod;
                if (phi[l2] > 2.0 * M_PI) phi[l2] -= 2.0 * M_PI;
                if (phi_mod[l2] > 2.0 * M_PI) phi_mod[l2] -= 2.0 * M_PI;
                sound = GAIN * envelope(&note_active[l2], gate[l2], &env_level[l2], env_tim
                             * velocity[l2] * sin(phi[l2] + modulation * sin(phi_mod[l2]));
                env_time[l2] += 1.0 / 44100.0;
                buf[2 * l1] += sound;
                buf[2 * l1 + 1] += sound;
            }
        }
    }
    return snd_pcm_writei (playback_handle, buf, nframes);
```

```
}

int main (int argc, char *argv[]) {

    int nfds, seq_nfds, l1;
    struct pollfd *pfds;

    if (argc < 10) {
        fprintf(stderr, "miniFMsynth <device> <FM> <harmonic> <subharmonic> <transpose> <a>
        exit(1);
    }
    modulation = atof(argv[2]);
    harmonic = atoi(argv[3]);
    subharmonic = atoi(argv[4]);
    transpose = atoi(argv[5]);
    attack = atof(argv[6]);
    decay = atof(argv[7]);
    sustain = atof(argv[8]);
    release = atof(argv[9]);
    pitch = 0;
    buf = (short *) malloc (2 * sizeof (short) * BUFSIZE);
    playback_handle = open_pcm(argv[1]);
    seq_handle = open_seq();
    seq_nfds = snd_seq_poll_descriptors_count(seq_handle, POLLIN);
    nfds = snd_pcm_poll_descriptors_count (playback_handle);
    pfds = (struct pollfd *)alloca(sizeof(struct pollfd) * (seq_nfds + nfds));
    snd_seq_poll_descriptors(seq_handle, pfds, seq_nfds, POLLIN);
    snd_pcm_poll_descriptors (playback_handle, pfds+seq_nfds, nfds);
    for (l1 = 0; l1 < POLY; note_active[l1++] = 0);
    while (1) {
        if (poll (pfds, seq_nfds + nfds, 1000) > 0) {
            for (l1 = 0; l1 < seq_nfds; l1++) {
               if (pfds[l1].revents > 0) midi_callback();
            }
            for (l1 = seq_nfds; l1 < seq_nfds + nfds; l1++) {
                if (pfds[l1].revents > 0) {
                    if (playback_callback(BUFSIZE) < BUFSIZE) {
                        fprintf (stderr, "xrun !\n");
                        snd_pcm_prepare(playback_handle);
                    }
                }
            }
        }
    }
    snd_pcm_close (playback_handle);
    snd_seq_close (seq_handle);
    free(buf);
    return (0);
}
```

Il utilise plusieurs paramètres,

```
miniFMsynth <device> <FM> <harmonic> <subharmonic> <transpose> <a> <d> <s> <r>
```

dispositif PCM, force de modulation de fréquence, harmonique de l'oscillateur principal (nombre entier), sous harmonique de l'oscillateur principal (nombre entier), Offset pour les deux oscillateurs (nombre entier), attaque, delay, sustain, release.

Quelque exemple

- Harpsichord, ./miniFMsynth plughw:0 7.8 3 5 24 0.01 0.8 0.0 0.1
- Bell, ./miniFMsynth plughw:0 3.5 7 9 0 0.01 0.2 0.3 1.5
- Oboe ./miniFMsynth plughw:0 0.7 1 3 24 0.05 0.3 0.8 0.2

Le miniFMsynth réagit sur des événements de pitch et de modulation. Puisqu'il n'est pas optimisé pour la performance (vous n'utiliseriez par exemple jamais la fonction "sin" dans un "vrai" programme), vous pourriez devoir diminuer la polyphony dans la source en modifiant #define POLY.

Après avoir lu, et compris, les chapitres précédents, il devrait être facile de comprendre la partie MIDI du programme. Quant à la lecture PCM, miniFMsynth utilise polling, cette technique est plus avancée que la sortie direct décrit dans le chapitre 2. Regardons certains détail

A)

```
snd_pcm_t *open_pcm(char *pcm_name)
```

Pour la plupart des fonctions de snd_pcm_hw_params utilisées ici voir le chapitre 2. Cependant, ici nous n'indiquons pas la taille du buffer, mais nous définissons la taille de la periode à la place.

Puisque nous voulons utiliser le polling pour la sortie PCM, nous devons également définir le paramètre "avail_min" . Un poll dans un fichier PCM descripteur retournera des données seulement si des frames avail_min peuvent être transmis au dispositif.

Nous initialisons snd_pcm_sw_params_t avec la configuration courante appelée snd_pcm_sw_params_current, puis définissons avail_min avec snd_pcm_sw_params_set_avail_min et activons cette configuration en utilisant snd_pcm_sw_params.

B)

```
double envelope(int *note_active, int gate, double *env_level, double t, double attack,
```

Si gate==1, la note est toujours pressé ==> l'enveloppe est dans le secteur attaque, decay, sustain . Si gate==0, l'enveloppe est dans le secteur release. Si t>release, la cellule respective d'oscillateur est libéré par le réglage de note_active à zéro.

C)

```
int midi_callback()
```

Cette partie, en gros, a déjà été vu dans le chapitre 4. Quand un événement NOTEON est reçu, une nouvelle cellule d'oscillateur est initialisée en plaçant la note_active respective à 1. Ceci fonctionne seulement si la polyphony n'est pas excessive, autrement la note est omise.

Si cela ne vous plait pas, vous pouvez le modifier et laissez par exemple la cellule d'oscillateur avec un plus petit env_level qui sera libéré et utilisé pour la nouvelle note. Il y a un détail important au sujet des événements de note : Quelques claviers (par exemple MP 9000 de Kawai) n'envoient pas d'événements NOTEOFF, mais envoient un NOTEON avec la vélocité 0 à la place. Dans ce cas-ci, vous devez ajouter les lignes suivantes

```
if ((ev->type == SND_SEQ_EVENT_NOTEON) && (ev->data.note.velocity == 0))
     ev->type = SND_SEQ_EVENT_NOTEOFF;
```

directement après snd_seq_event_input

D)

```
int playback_callback (snd_pcm_sframes_t nframes
```

Cette partie traite les données PCM et les envoi au dispositif à l'aide de snd_pcm_write. Cette fonction renverra des données immédiatement, puisque nous utilisons seulement le playback_callback, si nous connaisont le nombre nframes qui peuvent être transmis au dispositif PCM. MiniFMsynth écrit toujours des gros morceaux de taille fixe, ainsi nframes équivaut toujours à BUFSIZE.

E)

```
int main (int argc, char *argv[])
```

Ici, nous utilisons le polling pour l'entrée d'événement MIDI et pour la lecture PCM. La fonction poll renverra des données seulement si un événement MIDI peut être lu à partir de l'entrée du buffer du séquenceur ou si au moins des frames avail_min (= =BUFSIZE) peuvent être transmis au dispositif PCM.

Si les données PCM ne sont pas fourni au dispositif de PCM avant que le buffer de la carte son soit vide, un buffer underrun se produit, ceci arrêtera la lecture. Dans ce cas-ci, la valeur de retour du snd_pcm_write sera plus petite que BUFSIZE, nous devrons alors relancer la lecture à l'aide de snd_pcm_prepare.

Chapitre 7. Scheduling MIDI events: miniArp

Dans le chapitre section 5, nous avons utilisé la sortie immédiate des événements MIDI. Cependant, les événements MIDI sont habituellement programmés dans une file d'attente qui est controlée par un temporisateur matériel.

L'arpeggiator miniArp.c fournit un exemple simple de programmeur de tache MIDI

```
/* miniArp.c by Matthias Nagorni */

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <alsa/asoundlib.h>

#define TICKS_PER_QUARTER 128
#define MAX_SEQ_LEN        64

int queue_id, port_in_id, port_out_id, transpose, bpm0, bpm, tempo, swing, sequence[3][MAX_
snd_seq_t *seq_handle;
char seq_filename[1024];
snd_seq_tick_time_t tick;

snd_seq_t *open_seq() {

  snd_seq_t *seq_handle;

  if (snd_seq_open(&seq_handle, "default", SND_SEQ_OPEN_DUPLEX, 0) < 0) {
    fprintf(stderr, "Error opening ALSA sequencer.\n");
    exit(1);
  }
  snd_seq_set_client_name(seq_handle, "miniArp");
  if ((port_out_id = snd_seq_create_simple_port(seq_handle, "miniArp",
            SND_SEQ_PORT_CAP_READ|SND_SEQ_PORT_CAP_SUBS_READ,
            SND_SEQ_PORT_TYPE_APPLICATION)) < 0) {
    fprintf(stderr, "Error creating sequencer port.\n");
  }
  if ((port_in_id = snd_seq_create_simple_port(seq_handle, "miniArp",
            SND_SEQ_PORT_CAP_WRITE|SND_SEQ_PORT_CAP_SUBS_WRITE,
            SND_SEQ_PORT_TYPE_APPLICATION)) < 0) {
    fprintf(stderr, "Error creating sequencer port.\n");
    exit(1);
  }
  return(seq_handle);
}

void set_tempo() {

  snd_seq_queue_tempo_t *queue_tempo;
```

```
  snd_seq_queue_tempo_malloc(&queue_tempo);
  tempo = (int)(6e7 / ((double)bpm * (double)TICKS_PER_QUARTER) * (double)TICKS_PER_QUARTER
  snd_seq_queue_tempo_set_tempo(queue_tempo, tempo);
  snd_seq_queue_tempo_set_ppq(queue_tempo, TICKS_PER_QUARTER);
  snd_seq_set_queue_tempo(seq_handle, queue_id, queue_tempo);
  snd_seq_queue_tempo_free(queue_tempo);
}

snd_seq_tick_time_t get_tick() {

  snd_seq_queue_status_t *status;
  snd_seq_tick_time_t current_tick;

  snd_seq_queue_status_malloc(&status);
  snd_seq_get_queue_status(seq_handle, queue_id, status);
  current_tick = snd_seq_queue_status_get_tick_time(status);
  snd_seq_queue_status_free(status);
  return(current_tick);
}

void init_queue() {

  queue_id = snd_seq_alloc_queue(seq_handle);
  snd_seq_set_client_pool_output(seq_handle, (seq_len<<1) + 4);
}

void clear_queue() {

  snd_seq_remove_events_t *remove_ev;

  snd_seq_remove_events_malloc(&remove_ev);
  snd_seq_remove_events_set_queue(remove_ev, queue_id);
  snd_seq_remove_events_set_condition(remove_ev, SND_SEQ_REMOVE_OUTPUT | SND_SEQ_REMOVE_IGN
  snd_seq_remove_events(seq_handle, remove_ev);
  snd_seq_remove_events_free(remove_ev);
}

void arpeggio() {

  snd_seq_event_t ev;
  int l1;
  double dt;

  for (l1 = 0; l1 < seq_len; l1++) {
    dt = (l1 % 2 == 0) ? (double)swing / 16384.0 : -(double)swing / 16384.0;
    snd_seq_ev_clear(&ev);
    snd_seq_ev_set_note(&ev, 0, sequence[2][l1] + transpose, 127, sequence[1][l1]);
    snd_seq_ev_schedule_tick(&ev, queue_id,  0, tick);
    snd_seq_ev_set_source(&ev, port_out_id);
    snd_seq_ev_set_subs(&ev);
    snd_seq_event_output_direct(seq_handle, &ev);
    tick += (int)((double)sequence[0][l1] * (1.0 + dt));
```

```
  }
  snd_seq_ev_clear(&ev);
  ev.type = SND_SEQ_EVENT_ECHO;
  snd_seq_ev_schedule_tick(&ev, queue_id,  0, tick);
  snd_seq_ev_set_dest(&ev, snd_seq_client_id(seq_handle), port_in_id);
  snd_seq_event_output_direct(seq_handle, &ev);
}

void midi_action() {

  snd_seq_event_t *ev;

  do {
    snd_seq_event_input(seq_handle, &ev);
    switch (ev->type) {
      case SND_SEQ_EVENT_ECHO:
        arpeggio();
        break;
      case SND_SEQ_EVENT_NOTEON:
        clear_queue();
        transpose = ev->data.note.note - 60;
        tick = get_tick();
        arpeggio();
        break;
      case SND_SEQ_EVENT_CONTROLLER:
        if (ev->data.control.param == 1) {
          bpm = (int)((double)bpm0 * (1.0 + (double)ev->data.control.value / 127.0));
          set_tempo();
        }
        break;
      case SND_SEQ_EVENT_PITCHBEND:
        swing = (double)ev->data.control.value;
        break;
    }
    snd_seq_free_event(ev);
  } while (snd_seq_event_input_pending(seq_handle, 0) > 0);
}

void parse_sequence() {

  FILE *f;
  char c;

  if (!(f = fopen(seq_filename, "r"))) {
    fprintf(stderr, "Couldn't open sequence file %s\n", seq_filename);
    exit(1);
  }
  seq_len = 0;
  while((c = fgetc(f))!=EOF) {
    switch (c) {
      case 'c':
        sequence[2][seq_len] = 0; break;
      case 'd':
```

```
        sequence[2][seq_len] = 2; break;
      case 'e':
        sequence[2][seq_len] = 4; break;
      case 'f':
        sequence[2][seq_len] = 5; break;
      case 'g':
        sequence[2][seq_len] = 7; break;
      case 'a':
        sequence[2][seq_len] = 9; break;
      case 'h':
        sequence[2][seq_len] = 11; break;
    }
    c =  fgetc(f);
    if (c == '#') {
      sequence[2][seq_len]++;
      c =  fgetc(f);
    }
    sequence[2][seq_len] += 12 * atoi(&c);
    c =  fgetc(f);
    sequence[1][seq_len] = TICKS_PER_QUARTER / atoi(&c);
    c =  fgetc(f);
    sequence[0][seq_len] = TICKS_PER_QUARTER / atoi(&c);
    seq_len++;
  }
  fclose(f);
}

void sigterm_exit(int sig) {

  clear_queue();
  sleep(2);
  snd_seq_stop_queue(seq_handle, queue_id, NULL);
  snd_seq_free_queue(seq_handle, queue_id);
  exit(0);
}

int main(int argc, char *argv[]) {

  int npfd, l1;
  struct pollfd *pfd;

  if (argc < 3) {
    fprintf(stderr, "\n\nminiArp <beats per minute> <sequence file>\n");
    exit(1);
  }
  bpm0 = atoi(argv[1]);
  bpm = bpm0;
  strcpy(seq_filename, argv[2]);
  parse_sequence();
  seq_handle = open_seq();
  init_queue();
  set_tempo();
  arpeggio();
```

```
  snd_seq_start_queue(seq_handle, queue_id, NULL);
  snd_seq_drain_output(seq_handle);
  npfd = snd_seq_poll_descriptors_count(seq_handle, POLLIN);
  pfd = (struct pollfd *)alloca(npfd * sizeof(struct pollfd));
  snd_seq_poll_descriptors(seq_handle, pfd, npfd, POLLIN);
  transpose = 0;
  swing = 0;
  tick = 0;
  signal(SIGINT, sigterm_exit);
  signal(SIGTERM, sigterm_exit);
  arpeggio();
  while (1) {
    if (poll(pfd, npfd, 100000) > 0) {
      for (l1 = 0; l1 < npfd; l1++) {
        if (pfd[l1].revents > 0) midi_action();
      }
    }
  }
}
```

MiniArp posséde deux paramètres, le tempo (battements par minute) et le nom de fichier de la séquence où la boucle doit être réalisée. Ce fichier de séquence se compose d'une ligne de caractères simples. Chaque note est caractérisée par 4 ou 5 caractères vwxyz, où

- V : correspond au note ('c', 'd', 'e', 'f', 'g', 'a', 'h')
- W : is an optional sharp ('#')
- X : spécifie l'octave
- Y : spécifie la longueur de la note
- Z : spécifie l'intervalle entre 2 notes

Y et Z sont tous les deux représentés comme une fraction d'un quart de note : longueur = 1/4 de 1/y. Une séquence valide serait :

```
c488c#488d488d#488e484c584g584c684e684g584c684e544c584g484e488d#488d488c#488
```

MiniArp traite les événements de pitch et de modulation. N'importe quel événement NOTEON transposera la séquence selon la différence entre la note pressée et le C moyen.

Maintenant, quels sont les secrets de ce programme ? Beaucoup de techniques ont déjà été vues. Les fonctions intéressantes sont init_queue, set_tempo, arpeggio, clear_queue, et get_tick. Un signal handler pour SIGINT et SIGTERM est implementé pour éviter les notes persistantes.

A)

```
void init_queue()
```

Cette partie défini une file d'attente et la taille du buffer. Cette taille est passée au snd_seq_set_client_pool_output comme nombre d'événements. Pour les files d'attente il pourrait être utile de regarder /proc/asound/seq/queues.

B) void set_tempo()

```
void init_queue()
```

La période de programme des événements peut être indiquée en temps réel ou dans les ticks. Si vous voulez changer le tempo, il est pratique de modifier le temps dans les ticks. Ceci changera même le tempo des événements qui ont déja été programmés dans la file d'attente.

Le tempo est transmis au snd_seq_queue_tempo_set_tempo en micro-secondes par tick. La fonction snd_seq_queue_tempo_set_ppq défini les ticks par quart. Habituellement, on veut indiquer le tempo en beats par minute (bpm), par conséquent set_tempo calcule les paramètres corrects de tempo et de ppq en bpm.

C) void arpeggio()

```
void init_queue()
```

Ici, toutes les notes de la séquence sont programmées dans la file d'attente. D'abord, nous devons initialiser la structure d'événement, puis, définir un événement de note de longueur fixe en utilisant le snd_seq_ev_set_note. Cet événement de note est programmé en spécifiant son tick time avec snd_seq_ev_schedule_tick.

La source d'événement est le port de sortie du miniArp et nous utilisons snd_seq_set_subs pour spécifier à ALSA que nous voulons que l'événement soit transmis à tous les "abonnés" de ce port. Pour finir, nous utilisons snd_seq_event_output_direct pour la sortie unbuffered de l'événement de la file d'attente.

Les événements de note fonctionnent comme suit : quand c'est leur tour, ils créent un événement NOTEON et puis le convertissent en événement NOTEOFF, état qui se produit au temps = tempsdépart + la longueur de note.

Puisque nous voulons que la séquence fasse une boucle, nous programmons un événement SND_SEQ_EVENT_ECHO après chaque séquence. Cette fois, la destination est l'application elle-même. Quand l'événement SND_SEQ_EVENT_ECHO est expédié, il active la fonction poll dans main() pour le renvoyer et midi_action est appellé. midi_action, SND_SEQ_EVENT_ECHO auront comme conséquence un appel à arpeggi0 et la boucle est complète.

D) void clear_queue()

```
void init_queue()
```

Quand miniArp reçoit un événement NOTEON, la séquence est transposé. clear_queue est appelé à chaque événement NOTEON pour déclencher une nouvelle séquence. SND_SEQ_REMOVE_OUTPUT | SND_SEQ_REMOVE_IGNORE_OFF sont transmis à snd_seq_remove_events_set_condition pour indiquer à ALSA que seulement les événements de sortie devraient être enlevés de la file d'attente et les événements NOTEOFF ne doivent pas être supprimé.

La dernière condition s'assure que toutes les notes de la séquence reçoivent un événement NOTEOFF après que leur longueur se soit écoulée, même lorsqu'une nouvelle séquence est déclenchée.

E) snd_seq_tick_time_t get_tick(

```
void init_queue()
```

Cette fonction récupére la valeur courante tick à partir d'une structure de file d'attente.

Chapitre 8. Ecrire une application audio graphique

Vous voudriez probablement utiliser l'exemple PCM audio dans une application avec une interface graphique, GUI. Dans ce cas on recommande d'utiliser deux partie, une partie audio et une partie GUI. Les deux partie peuvent alors communiquer par exemple via la mémoire partagée. Cela a aussi l'avantage que vous pouvez augmenter la priorité de la partie audio séparément de la priorité de la partie GUI.

La partie audio pourrait ressembler a ceci

```
while(shared_data->do_audio) {
    process_buffer(buf, bufsize);
    while ((pcm_return = snd_pcm_writei(pcm_handle, buf, bufsize)) < 0) {
        snd_pcm_prepare(pcm_handle);
        fprintf(stderr, "xrun !\n");
    }
}
```

Il n'est pas nécessaire d'utiliser le polling dans ce cas puisque snd_pcm_write bloquerait les frames qui veulent écrire au dispositif..

Chapitre 9. Liens

Documentation resources of the ALSA project (http://www.alsa-project.org/documentation.php3)

The ALSA library API reference (http://www.alsa-project.org/alsa-doc/alsa-lib/)

The most comprehensive site on sound with Linux. (http://linux-sound.org)

Tutorial on writing audio applications by Paul Davis (http://equalarea.com/paul/alsa-audio.html)

Linux MIDI HOWTO by Phil Kerr. This also covers installing and configuring ALSA (http://www.linuxdoc.org/HOWTO/MIDI-HOWTO.html)

Some applications (http://www.suse.de/~mana/)