```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
	"http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" []>

<!-- ****************************************************** -->
<!-- Header  -->
<!-- ****************************************************** -->
<book id="Writing-an-ALSA-Driver">
  <bookinfo>
    <title>Writing an ALSA Driver</title>
    <author>
      <firstname>Takashi</firstname>
      <surname>Iwai</surname>
      <affiliation>
        <address>
          <email>tiwai@suse.de</email>
        </address>
      </affiliation>
     </author>

     <date>Oct 15, 2007</date>
     <edition>0.3.7</edition>

    <abstract>
      <para>
        This document describes how to write an ALSA (Advanced Linux
        Sound Architecture) driver.
      </para>
    </abstract>

    <legalnotice>
    <para>
    Copyright (c) 2002-2005  Takashi Iwai <email>tiwai@suse.de</email>
    </para>

    <para>
    This document is free; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.
    </para>

    <para>
    This document is distributed in the hope that it will be useful,
    but <emphasis>WITHOUT ANY WARRANTY</emphasis>; without even the
    implied warranty of <emphasis>MERCHANTABILITY or FITNESS FOR A
    PARTICULAR PURPOSE</emphasis>. See the GNU General Public License
    for more details.
    </para>

    <para>
    You should have received a copy of the GNU General Public
    License along with this program; if not, write to the Free
    Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,
    MA 02111-1307 USA
    </para>
```

```
    </legalnotice>

  </bookinfo>

<!-- ****************************************************** -->
<!-- Preface  -->
<!-- ****************************************************** -->
  <preface id="preface">
    <title>Preface</title>
    <para>
      This document describes how to write an
      <ulink url="http://www.alsa-project.org/"><citetitle>
      ALSA (Advanced Linux Sound Architecture)</citetitle></ulink>
      driver. The document focuses mainly on PCI soundcards.
      In the case of other device types, the API might
      be different, too. However, at least the ALSA kernel API is
      consistent, and therefore it would be still a bit help for
      writing them.
    </para>

    <para>
    This document targets people who already have enough
    C language skills and have basic linux kernel programming
    knowledge.  This document doesn't explain the general
    topic of linux kernel coding and doesn't cover low-level
    driver implementation details. It only describes
    the standard way to write a PCI sound driver on ALSA.
    </para>

    <para>
      If you are already familiar with the older ALSA ver.0.5.x API, you
    can check the drivers such as <filename>sound/pci/es1938.c</filename> or
    <filename>sound/pci/maestro3.c</filename> which have also almost the same
    code-base in the ALSA 0.5.x tree, so you can compare the differences.
    </para>

    <para>
      This document is still a draft version. Any feedback and
    corrections, please!!
    </para>
  </preface>


<!-- ****************************************************** -->
<!-- File Tree Structure  -->
<!-- ****************************************************** -->
  <chapter id="file-tree">
    <title>File Tree Structure</title>

    <section id="file-tree-general">
      <title>General</title>
      <para>
        The ALSA drivers are provided in two ways.
      </para>

      <para>
```

```
          One is the trees provided as a tarball or via cvs from the
        ALSA's ftp site, and another is the 2.6 (or later) Linux kernel
        tree. To synchronize both, the ALSA driver tree is split into
        two different trees: alsa-kernel and alsa-driver. The former
        contains purely the source code for the Linux 2.6 (or later)
        tree. This tree is designed only for compilation on 2.6 or
        later environment. The latter, alsa-driver, contains many subtle
        files for compiling ALSA drivers outside of the Linux kernel tree,
        wrapper functions for older 2.2 and 2.4 kernels, to adapt the latest
kernel API,
        and additional drivers which are still in development or in
        tests.  The drivers in alsa-driver tree will be moved to
        alsa-kernel (and eventually to the 2.6 kernel tree) when they are
        finished and confirmed to work fine.
        </para>

        <para>
          The file tree structure of ALSA driver is depicted below. Both
          alsa-kernel and alsa-driver have almost the same file
          structure, except for <quote>core</quote> directory. It's
          named as <quote>acore</quote> in alsa-driver tree.

          <example>
            <title>ALSA File Tree Structure</title>
            <literallayout>
          sound
                  /core
                          /oss
                          /seq
                                  /oss
                                  /instr
                  /ioctl32
                  /include
                  /drivers
                          /mpu401
                          /opl3
                  /i2c
                          /l3
                  /synth
                          /emux
                  /pci
                          /(cards)
                  /isa
                          /(cards)
                  /arm
                  /ppc
                  /sparc
                  /usb
                  /pcmcia /(cards)
                  /oss
            </literallayout>
          </example>
        </para>
      </section>

      <section id="file-tree-core-directory">
```

```
<title>core directory</title>
<para>
  This directory contains the middle layer which is the heart
of ALSA drivers. In this directory, the native ALSA modules are
stored. The sub-directories contain different modules and are
dependent upon the kernel config.
</para>

<section id="file-tree-core-directory-oss">
  <title>core/oss</title>

  <para>
    The codes for PCM and mixer OSS emulation modules are stored
  in this directory. The rawmidi OSS emulation is included in
  the ALSA rawmidi code since it's quite small. The sequencer
  code is stored in <filename>core/seq/oss</filename> directory (see
  <link linkend="file-tree-core-directory-seq-oss"><citetitle>
  below</citetitle></link>).
  </para>
</section>

<section id="file-tree-core-directory-ioctl32">
  <title>core/ioctl32</title>

  <para>
    This directory contains the 32bit-ioctl wrappers for 64bit
  architectures such like x86-64, ppc64 and sparc64. For 32bit
  and alpha architectures, these are not compiled.
  </para>
</section>

<section id="file-tree-core-directory-seq">
  <title>core/seq</title>
  <para>
    This directory and its sub-directories are for the ALSA
  sequencer. This directory contains the sequencer core and
  primary sequencer modules such like snd-seq-midi,
  snd-seq-virmidi, etc. They are compiled only when
  <constant>CONFIG_SND_SEQUENCER</constant> is set in the kernel
  config.
  </para>
</section>

<section id="file-tree-core-directory-seq-oss">
  <title>core/seq/oss</title>
  <para>
    This contains the OSS sequencer emulation codes.
  </para>
</section>

<section id="file-tree-core-directory-deq-instr">
  <title>core/seq/instr</title>
  <para>
    This directory contains the modules for the sequencer
  instrument layer.
  </para>
```

```
  </section>
</section>

<section id="file-tree-include-directory">
  <title>include directory</title>
  <para>
    This is the place for the public header files of ALSA drivers,
  which are to be exported to user-space, or included by
  several files at different directories. Basically, the private
  header files should not be placed in this directory, but you may
  still find files there, due to historical reasons :)
  </para>
</section>

<section id="file-tree-drivers-directory">
  <title>drivers directory</title>
  <para>
    This directory contains code shared among different drivers
  on different architectures.  They are hence supposed not to be
  architecture-specific.
  For example, the dummy pcm driver and the serial MIDI
  driver are found in this directory. In the sub-directories,
  there is code for components which are independent from
  bus and cpu architectures.
  </para>

  <section id="file-tree-drivers-directory-mpu401">
    <title>drivers/mpu401</title>
    <para>
      The MPU401 and MPU401-UART modules are stored here.
    </para>
  </section>

  <section id="file-tree-drivers-directory-opl3">
    <title>drivers/opl3 and opl4</title>
    <para>
      The OPL3 and OPL4 FM-synth stuff is found here.
    </para>
  </section>
</section>

<section id="file-tree-i2c-directory">
  <title>i2c directory</title>
  <para>
    This contains the ALSA i2c components.
  </para>

  <para>
    Although there is a standard i2c layer on Linux, ALSA has its
  own i2c code for some cards, because the soundcard needs only a
  simple operation and the standard i2c API is too complicated for
  such a purpose.
  </para>

  <section id="file-tree-i2c-directory-l3">
    <title>i2c/l3</title>
```

```
        <para>
          This is a sub-directory for ARM L3 i2c.
        </para>
      </section>
    </section>

    <section id="file-tree-synth-directory">
        <title>synth directory</title>
        <para>
          This contains the synth middle-level modules.
        </para>

        <para>
          So far, there is only Emu8000/Emu10k1 synth driver under
        the <filename>synth/emux</filename> sub-directory.
        </para>
    </section>

    <section id="file-tree-pci-directory">
      <title>pci directory</title>
      <para>
        This directory and its sub-directories hold the top-level card modules
      for PCI soundcards and the code specific to the PCI BUS.
      </para>

      <para>
        The drivers compiled from a single file are stored directly
      in the pci directory, while the drivers with several source files are
      stored on their own sub-directory (e.g. emu10k1, ice1712).
      </para>
    </section>

    <section id="file-tree-isa-directory">
      <title>isa directory</title>
      <para>
        This directory and its sub-directories hold the top-level card modules
      for ISA soundcards.
      </para>
    </section>

    <section id="file-tree-arm-ppc-sparc-directories">
      <title>arm, ppc, and sparc directories</title>
      <para>
        They are used for top-level card modules which are
      specific to one of these architectures.
      </para>
    </section>

    <section id="file-tree-usb-directory">
      <title>usb directory</title>
      <para>
        This directory contains the USB-audio driver. In the latest version, the
      USB MIDI driver is integrated in the usb-audio driver.
      </para>
    </section>
```

```
    <section id="file-tree-pcmcia-directory">
      <title>pcmcia directory</title>
      <para>
        The PCMCIA, especially PCCard drivers will go here. CardBus
      drivers will be in the pci directory, because their API is identical
      to that of standard PCI cards.
      </para>
    </section>

    <section id="file-tree-oss-directory">
      <title>oss directory</title>
      <para>
        The OSS/Lite source files are stored here in Linux 2.6 (or
      later) tree. In the ALSA driver tarball, this directory is empty,
      of course :)
      </para>
    </section>
  </chapter>


<!-- ****************************************************** -->
<!-- Basic Flow for PCI Drivers  -->
<!-- ****************************************************** -->
  <chapter id="basic-flow">
    <title>Basic Flow for PCI Drivers</title>

    <section id="basic-flow-outline">
      <title>Outline</title>
      <para>
        The minimum flow for PCI soundcards is as follows:

        <itemizedlist>
          <listitem><para>define the PCI ID table (see the section
          <link linkend="pci-resource-entries"><citetitle>PCI Entries
          </citetitle></link>).</para></listitem>
          <listitem><para>create <function>probe()</function>
callback.</para></listitem>
          <listitem><para>create <function>remove()</function>
callback.</para></listitem>
          <listitem><para>create a <structname>pci_driver</structname> structure
          containing the three pointers above.</para></listitem>
          <listitem><para>create an <function>init()</function> function just
calling
          the <function>pci_register_driver()</function> to register the
pci_driver table
          defined above.</para></listitem>
          <listitem><para>create an <function>exit()</function> function to call
          the <function>pci_unregister_driver()</function>
function.</para></listitem>
        </itemizedlist>
      </para>
    </section>

    <section id="basic-flow-example">
      <title>Full Code Example</title>
      <para>
```

```
  The code example is shown below. Some parts are kept
unimplemented at this moment but will be filled in the
next sections. The numbers in the comment lines of the
<function>snd_mychip_probe()</function> function
refer to details explained in the following section.

  <example>
    <title>Basic Flow for PCI Drivers - Example</title>
    <programlisting>
<![CDATA[
  #include <linux/init.h>
  #include <linux/pci.h>
  #include <linux/slab.h>
  #include <sound/core.h>
  #include <sound/initval.h>

  /* module parameters (see "Module Parameters") */
  /* SNDRV_CARDS: maximum number of cards supported by this module */
  static int index[SNDRV_CARDS] = SNDRV_DEFAULT_IDX;
  static char *id[SNDRV_CARDS] = SNDRV_DEFAULT_STR;
  static int enable[SNDRV_CARDS] = SNDRV_DEFAULT_ENABLE_PNP;

  /* definition of the chip-specific record */
  struct mychip {
          struct snd_card *card;
          /* the rest of the implementation will be in section
           * "PCI Resource Management"
           */
  };

  /* chip-specific destructor
   * (see "PCI Resource Management")
   */
  static int snd_mychip_free(struct mychip *chip)
  {
          .... /* will be implemented later... */
  }

  /* component-destructor
   * (see "Management of Cards and Components")
   */
  static int snd_mychip_dev_free(struct snd_device *device)
  {
          return snd_mychip_free(device->device_data);
  }

  /* chip-specific constructor
   * (see "Management of Cards and Components")
   */
  static int __devinit snd_mychip_create(struct snd_card *card,
                                         struct pci_dev *pci,
                                         struct mychip **rchip)
  {
          struct mychip *chip;
          int err;
          static struct snd_device_ops ops = {
```

```
                .dev_free = snd_mychip_dev_free,
        };

        *rchip = NULL;

        /* check PCI availability here
         * (see "PCI Resource Management")
         */
        ....

        /* allocate a chip-specific data with zero filled */
        chip = kzalloc(sizeof(*chip), GFP_KERNEL);
        if (chip == NULL)
                return -ENOMEM;

        chip->card = card;

        /* rest of initialization here; will be implemented
         * later, see "PCI Resource Management"
         */
        ....

        err = snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
        if (err < 0) {
                snd_mychip_free(chip);
                return err;
        }

        snd_card_set_dev(card, &pci->dev);

        *rchip = chip;
        return 0;
}

/* constructor -- see "Constructor" sub-section */
static int __devinit snd_mychip_probe(struct pci_dev *pci,
                             const struct pci_device_id *pci_id)
{
        static int dev;
        struct snd_card *card;
        struct mychip *chip;
        int err;

        /* (1) */
        if (dev >= SNDRV_CARDS)
                return -ENODEV;
        if (!enable[dev]) {
                dev++;
                return -ENOENT;
        }

        /* (2) */
        err = snd_card_create(index[dev], id[dev], THIS_MODULE, 0, &card);
        if (err < 0)
                return err;
```

```
          /* (3) */
          err = snd_mychip_create(card, pci, &chip);
          if (err < 0) {
                  snd_card_free(card);
                  return err;
          }

          /* (4) */
          strcpy(card->driver, "My Chip");
          strcpy(card->shortname, "My Own Chip 123");
          sprintf(card->longname, "%s at 0x%lx irq %i",
                  card->shortname, chip->ioport, chip->irq);

          /* (5) */
          .... /* implemented later */

          /* (6) */
          err = snd_card_register(card);
          if (err < 0) {
                  snd_card_free(card);
                  return err;
          }

          /* (7) */
          pci_set_drvdata(pci, card);
          dev++;
          return 0;
  }

  /* destructor -- see the "Destructor" sub-section */
  static void __devexit snd_mychip_remove(struct pci_dev *pci)
  {
          snd_card_free(pci_get_drvdata(pci));
          pci_set_drvdata(pci, NULL);
  }
]]>
          </programlisting>
        </example>
      </para>
    </section>

    <section id="basic-flow-constructor">
      <title>Constructor</title>
      <para>
        The real constructor of PCI drivers is the <function>probe</function>
callback.
      The <function>probe</function> callback and other component-constructors
which are called
      from the <function>probe</function> callback should be defined with
      the <parameter>__devinit</parameter> prefix. You
      cannot use the <parameter>__init</parameter> prefix for them,
      because any PCI device could be a hotplug device.
      </para>

      <para>
        In the <function>probe</function> callback, the following scheme is
```

```
often used.
      </para>

      <section id="basic-flow-constructor-device-index">
        <title>1) Check and increment the device index.</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  static int dev;
  ....
  if (dev >= SNDRV_CARDS)
          return -ENODEV;
  if (!enable[dev]) {
          dev++;
          return -ENOENT;
  }
]]>
            </programlisting>
          </informalexample>

        where enable[dev] is the module option.
        </para>

        <para>
          Each time the <function>probe</function> callback is called, check the
        availability of the device. If not available, simply increment
        the device index and returns. dev will be incremented also
        later (<link
        linkend="basic-flow-constructor-set-pci"><citetitle>step
        7</citetitle></link>).
        </para>
      </section>

      <section id="basic-flow-constructor-create-card">
        <title>2) Create a card instance</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  struct snd_card *card;
  int err;
  ....
  err = snd_card_create(index[dev], id[dev], THIS_MODULE, 0, &card);
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          The details will be explained in the section
          <link linkend="card-management-card-instance"><citetitle>
          Management of Cards and Components</citetitle></link>.
        </para>
      </section>
```

```
      <section id="basic-flow-constructor-create-main">
        <title>3) Create a main component</title>
        <para>
          In this part, the PCI resources are allocated.

          <informalexample>
            <programlisting>
<![CDATA[
  struct mychip *chip;
  ....
  err = snd_mychip_create(card, pci, &chip);
  if (err < 0) {
          snd_card_free(card);
          return err;
  }
]]>
            </programlisting>
          </informalexample>

          The details will be explained in the section <link
        linkend="pci-resource"><citetitle>PCI Resource
        Management</citetitle></link>.
        </para>
      </section>

      <section id="basic-flow-constructor-main-component">
        <title>4) Set the driver ID and name strings.</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  strcpy(card->driver, "My Chip");
  strcpy(card->shortname, "My Own Chip 123");
  sprintf(card->longname, "%s at 0x%lx irq %i",
          card->shortname, chip->ioport, chip->irq);
]]>
            </programlisting>
          </informalexample>

          The driver field holds the minimal ID string of the
        chip. This is used by alsa-lib's configurator, so keep it
        simple but unique.
          Even the same driver can have different driver IDs to
        distinguish the functionality of each chip type.
        </para>

        <para>
          The shortname field is a string shown as more verbose
        name. The longname field contains the information
        shown in <filename>/proc/asound/cards</filename>.
        </para>
      </section>

      <section id="basic-flow-constructor-create-other">
        <title>5) Create other components, such as mixer, MIDI, etc.</title>
        <para>
```

```
          Here you define the basic components such as
          <link linkend="pcm-interface"><citetitle>PCM</citetitle></link>,
          mixer (e.g. <link
linkend="api-ac97"><citetitle>AC97</citetitle></link>),
          MIDI (e.g. <link
linkend="midi-interface"><citetitle>MPU-401</citetitle></link>),
          and other interfaces.
          Also, if you want a <link linkend="proc-interface"><citetitle>proc
        file</citetitle></link>, define it here, too.
        </para>
      </section>

      <section id="basic-flow-constructor-register-card">
        <title>6) Register the card instance.</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  err = snd_card_register(card);
  if (err < 0) {
          snd_card_free(card);
          return err;
  }
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          Will be explained in the section <link
        linkend="card-management-registration"><citetitle>Management
        of Cards and Components</citetitle></link>, too.
        </para>
      </section>

      <section id="basic-flow-constructor-set-pci">
        <title>7) Set the PCI driver data and return zero.</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
        pci_set_drvdata(pci, card);
        dev++;
        return 0;
]]>
            </programlisting>
          </informalexample>

          In the above, the card record is stored. This pointer is
        used in the remove callback and power-management
        callbacks, too.
        </para>
      </section>
    </section>

    <section id="basic-flow-destructor">
```

```
      <title>Destructor</title>
      <para>
        The destructor, remove callback, simply releases the card
      instance. Then the ALSA middle layer will release all the
      attached components automatically.
      </para>

      <para>
        It would be typically like the following:

        <informalexample>
          <programlisting>
<![CDATA[
  static void __devexit snd_mychip_remove(struct pci_dev *pci)
  {
          snd_card_free(pci_get_drvdata(pci));
          pci_set_drvdata(pci, NULL);
  }
]]>
          </programlisting>
        </informalexample>

        The above code assumes that the card pointer is set to the PCI
        driver data.
      </para>
    </section>

    <section id="basic-flow-header-files">
      <title>Header Files</title>
      <para>
        For the above example, at least the following include files
      are necessary.

        <informalexample>
          <programlisting>
<![CDATA[
  #include <linux/init.h>
  #include <linux/pci.h>
  #include <linux/slab.h>
  #include <sound/core.h>
  #include <sound/initval.h>
]]>
          </programlisting>
        </informalexample>

        where the last one is necessary only when module options are
      defined in the source file.  If the code is split into several
      files, the files without module options don't need them.
      </para>

      <para>
        In addition to these headers, you'll need
      <filename>&lt;linux/interrupt.h&gt;</filename> for interrupt
      handling, and <filename>&lt;asm/io.h&gt;</filename> for I/O
      access. If you use the <function>mdelay()</function> or
      <function>udelay()</function> functions, you'll need to include
```

```
      <filename>&lt;linux/delay.h&gt;</filename> too.
      </para>

      <para>
      The ALSA interfaces like the PCM and control APIs are defined in other
      <filename>&lt;sound/xxx.h&gt;</filename> header files.
      They have to be included after
      <filename>&lt;sound/core.h&gt;</filename>.
      </para>

    </section>
  </chapter>


<!-- ****************************************************** -->
<!-- Management of Cards and Components  -->
<!-- ****************************************************** -->
  <chapter id="card-management">
    <title>Management of Cards and Components</title>

    <section id="card-management-card-instance">
      <title>Card Instance</title>
      <para>
      For each soundcard, a <quote>card</quote> record must be allocated.
      </para>

      <para>
      A card record is the headquarters of the soundcard.  It manages
      the whole list of devices (components) on the soundcard, such as
      PCM, mixers, MIDI, synthesizer, and so on.  Also, the card
      record holds the ID and the name strings of the card, manages
      the root of proc files, and controls the power-management states
      and hotplug disconnections.  The component list on the card
      record is used to manage the correct release of resources at
      destruction.
      </para>

      <para>
        As mentioned above, to create a card instance, call
      <function>snd_card_create()</function>.

        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_card *card;
  int err;
  err = snd_card_create(index, id, module, extra_size, &card);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The function takes five arguments, the card-index number, the
        id string, the module pointer (usually
        <constant>THIS_MODULE</constant>),
```

```
          the size of extra-data space, and the pointer to return the
          card instance.  The extra_size argument is used to
          allocate card-&gt;private_data for the
          chip-specific data.  Note that these data
          are allocated by <function>snd_card_create()</function>.
        </para>
      </section>

      <section id="card-management-component">
        <title>Components</title>
        <para>
          After the card is created, you can attach the components
        (devices) to the card instance. In an ALSA driver, a component is
        represented as a struct <structname>snd_device</structname> object.
        A component can be a PCM instance, a control interface, a raw
        MIDI interface, etc.  Each such instance has one component
        entry.
        </para>

        <para>
          A component can be created via
          <function>snd_device_new()</function> function.

          <informalexample>
            <programlisting>
<![CDATA[
  snd_device_new(card, SNDRV_DEV_XXX, chip, &ops);
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          This takes the card pointer, the device-level
        (<constant>SNDRV_DEV_XXX</constant>), the data pointer, and the
        callback pointers (<parameter>&amp;ops</parameter>). The
        device-level defines the type of components and the order of
        registration and de-registration.  For most components, the
        device-level is already defined.  For a user-defined component,
        you can use <constant>SNDRV_DEV_LOWLEVEL</constant>.
        </para>

        <para>
        This function itself doesn't allocate the data space. The data
        must be allocated manually beforehand, and its pointer is passed
        as the argument. This pointer is used as the
        (<parameter>chip</parameter> identifier in the above example)
        for the instance.
        </para>

        <para>
          Each pre-defined ALSA component such as ac97 and pcm calls
        <function>snd_device_new()</function> inside its
        constructor. The destructor for each component is defined in the
        callback pointers.  Hence, you don't need to take care of
        calling a destructor for such a component.
```

```
      </para>

      <para>
        If you wish to create your own component, you need to
      set the destructor function to the dev_free callback in
      the <parameter>ops</parameter>, so that it can be released
      automatically via <function>snd_card_free()</function>.
      The next example will show an implementation of chip-specific
      data.
      </para>
    </section>

    <section id="card-management-chip-specific">
      <title>Chip-Specific Data</title>
      <para>
      Chip-specific information, e.g. the I/O port address, its
      resource pointer, or the irq number, is stored in the
      chip-specific record.

        <informalexample>
          <programlisting>
<![CDATA[
  struct mychip {
          ....
  };
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        In general, there are two ways of allocating the chip record.
      </para>

      <section id="card-management-chip-specific-snd-card-new">
        <title>1. Allocating via <function>snd_card_create()</function>.</title>
        <para>
          As mentioned above, you can pass the extra-data-length
          to the 4th argument of <function>snd_card_create()</function>, i.e.

          <informalexample>
            <programlisting>
<![CDATA[
  err = snd_card_create(index[dev], id[dev], THIS_MODULE,
                        sizeof(struct mychip), &card);
]]>
            </programlisting>
          </informalexample>

          struct <structname>mychip</structname> is the type of the chip record.
        </para>

        <para>
          In return, the allocated record can be accessed as

          <informalexample>
```

```
            <programlisting>
<![CDATA[
  struct mychip *chip = card->private_data;
]]>
            </programlisting>
          </informalexample>

          With this method, you don't have to allocate twice.
          The record is released together with the card instance.
        </para>
      </section>

      <section id="card-management-chip-specific-allocate-extra">
        <title>2. Allocating an extra device.</title>

        <para>
          After allocating a card instance via
          <function>snd_card_create()</function> (with
          <constant>0</constant> on the 4th arg), call
          <function>kzalloc()</function>.

          <informalexample>
            <programlisting>
<![CDATA[
  struct snd_card *card;
  struct mychip *chip;
  err = snd_card_create(index[dev], id[dev], THIS_MODULE, 0, &card);
  .....
  chip = kzalloc(sizeof(*chip), GFP_KERNEL);
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          The chip record should have the field to hold the card
          pointer at least,

          <informalexample>
            <programlisting>
<![CDATA[
  struct mychip {
          struct snd_card *card;
          ....
  };
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          Then, set the card pointer in the returned chip instance.

          <informalexample>
            <programlisting>
<![CDATA[
```

```
  chip->card = card;
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        Next, initialize the fields, and register this chip
        record as a low-level device with a specified
        <parameter>ops</parameter>,

        <informalexample>
          <programlisting>
<![CDATA[
  static struct snd_device_ops ops = {
          .dev_free =        snd_mychip_dev_free,
  };
  ....
  snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
]]>
          </programlisting>
        </informalexample>

        <function>snd_mychip_dev_free()</function> is the
      device-destructor function, which will call the real
      destructor.
      </para>

      <para>
        <informalexample>
          <programlisting>
<![CDATA[
  static int snd_mychip_dev_free(struct snd_device *device)
  {
          return snd_mychip_free(device->device_data);
  }
]]>
          </programlisting>
        </informalexample>

        where <function>snd_mychip_free()</function> is the real destructor.
      </para>
    </section>
  </section>

  <section id="card-management-registration">
    <title>Registration and Release</title>
    <para>
      After all components are assigned, register the card instance
    by calling <function>snd_card_register()</function>. Access
    to the device files is enabled at this point. That is, before
    <function>snd_card_register()</function> is called, the
    components are safely inaccessible from external side. If this
    call fails, exit the probe function after releasing the card via
    <function>snd_card_free()</function>.
    </para>
```

```
      <para>
        For releasing the card instance, you can call simply
      <function>snd_card_free()</function>. As mentioned earlier, all
      components are released automatically by this call.
      </para>

      <para>
        As further notes, the destructors (both
      <function>snd_mychip_dev_free</function> and
      <function>snd_mychip_free</function>) cannot be defined with
      the <parameter>__devexit</parameter> prefix, because they may be
      called from the constructor, too, at the false path.
      </para>

      <para>
      For a device which allows hotplugging, you can use
      <function>snd_card_free_when_closed</function>.  This one will
      postpone the destruction until all devices are closed.
      </para>

    </section>

  </chapter>


<!-- ****************************************************** -->
<!-- PCI Resource Management  -->
<!-- ****************************************************** -->
  <chapter id="pci-resource">
    <title>PCI Resource Management</title>

    <section id="pci-resource-example">
      <title>Full Code Example</title>
      <para>
        In this section, we'll complete the chip-specific constructor,
      destructor and PCI entries. Example code is shown first,
      below.

        <example>
          <title>PCI Resource Management Example</title>
          <programlisting>
<![CDATA[
  struct mychip {
          struct snd_card *card;
          struct pci_dev *pci;

          unsigned long port;
          int irq;
  };

  static int snd_mychip_free(struct mychip *chip)
  {
          /* disable hardware here if any */
          .... /* (not implemented in this document) */
```

```
        /* release the irq */
        if (chip->irq >= 0)
                free_irq(chip->irq, chip);
        /* release the I/O ports & memory */
        pci_release_regions(chip->pci);
        /* disable the PCI entry */
        pci_disable_device(chip->pci);
        /* release the data */
        kfree(chip);
        return 0;
}

/* chip-specific constructor */
static int __devinit snd_mychip_create(struct snd_card *card,
                                       struct pci_dev *pci,
                                       struct mychip **rchip)
{
        struct mychip *chip;
        int err;
        static struct snd_device_ops ops = {
               .dev_free = snd_mychip_dev_free,
        };

        *rchip = NULL;

        /* initialize the PCI entry */
        err = pci_enable_device(pci);
        if (err < 0)
                return err;
        /* check PCI availability (28bit DMA) */
        if (pci_set_dma_mask(pci, DMA_BIT_MASK(28)) < 0 ||
            pci_set_consistent_dma_mask(pci, DMA_BIT_MASK(28)) < 0) {
                printk(KERN_ERR "error to set 28bit mask DMA\n");
                pci_disable_device(pci);
                return -ENXIO;
        }

        chip = kzalloc(sizeof(*chip), GFP_KERNEL);
        if (chip == NULL) {
                pci_disable_device(pci);
                return -ENOMEM;
        }

        /* initialize the stuff */
        chip->card = card;
        chip->pci = pci;
        chip->irq = -1;

        /* (1) PCI resource allocation */
        err = pci_request_regions(pci, "My Chip");
        if (err < 0) {
                kfree(chip);
                pci_disable_device(pci);
                return err;
        }
        chip->port = pci_resource_start(pci, 0);
```

```
        if (request_irq(pci->irq, snd_mychip_interrupt,
                        IRQF_SHARED, "My Chip", chip)) {
                printk(KERN_ERR "cannot grab irq %d\n", pci->irq);
                snd_mychip_free(chip);
                return -EBUSY;
        }
        chip->irq = pci->irq;

        /* (2) initialization of the chip hardware */
        .... /*   (not implemented in this document) */

        err = snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
        if (err < 0) {
                snd_mychip_free(chip);
                return err;
        }

        snd_card_set_dev(card, &pci->dev);

        *rchip = chip;
        return 0;
}

/* PCI IDs */
static struct pci_device_id snd_mychip_ids[] = {
        { PCI_VENDOR_ID_FOO, PCI_DEVICE_ID_BAR,
          PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0, },
        ....
        { 0, }
};
MODULE_DEVICE_TABLE(pci, snd_mychip_ids);

/* pci_driver definition */
static struct pci_driver driver = {
        .name = "My Own Chip",
        .id_table = snd_mychip_ids,
        .probe = snd_mychip_probe,
        .remove = __devexit_p(snd_mychip_remove),
};

/* module initialization */
static int __init alsa_card_mychip_init(void)
{
        return pci_register_driver(&driver);
}

/* module clean up */
static void __exit alsa_card_mychip_exit(void)
{
        pci_unregister_driver(&driver);
}

module_init(alsa_card_mychip_init)
module_exit(alsa_card_mychip_exit)

EXPORT_NO_SYMBOLS; /* for old kernels only */
```

```
]]>
          </programlisting>
        </example>
      </para>
    </section>

    <section id="pci-resource-some-haftas">
      <title>Some Hafta's</title>
      <para>
        The allocation of PCI resources is done in the
      <function>probe()</function> function, and usually an extra
      <function>xxx_create()</function> function is written for this
      purpose.
      </para>

      <para>
        In the case of PCI devices, you first have to call
      the <function>pci_enable_device()</function> function before
      allocating resources. Also, you need to set the proper PCI DMA
      mask to limit the accessed I/O range. In some cases, you might
      need to call <function>pci_set_master()</function> function,
      too.
      </para>

      <para>
        Suppose the 28bit mask, and the code to be added would be like:

        <informalexample>
          <programlisting>
<![CDATA[
  err = pci_enable_device(pci);
  if (err < 0)
          return err;
  if (pci_set_dma_mask(pci, DMA_BIT_MASK(28)) < 0 ||
      pci_set_consistent_dma_mask(pci, DMA_BIT_MASK(28)) < 0) {
          printk(KERN_ERR "error to set 28bit mask DMA\n");
          pci_disable_device(pci);
          return -ENXIO;
  }

]]>
          </programlisting>
        </informalexample>
      </para>
    </section>

    <section id="pci-resource-resource-allocation">
      <title>Resource Allocation</title>
      <para>
        The allocation of I/O ports and irqs is done via standard kernel
      functions. Unlike ALSA ver.0.5.x., there are no helpers for
      that. And these resources must be released in the destructor
      function (see below). Also, on ALSA 0.9.x, you don't need to
      allocate (pseudo-)DMA for PCI like in ALSA 0.5.x.
      </para>
```

```
      <para>
        Now assume that the PCI device has an I/O port with 8 bytes
        and an interrupt. Then struct <structname>mychip</structname> will have
the
        following fields:

        <informalexample>
          <programlisting>
<![CDATA[
  struct mychip {
          struct snd_card *card;

          unsigned long port;
          int irq;
  };
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        For an I/O port (and also a memory region), you need to have
      the resource pointer for the standard resource management. For
      an irq, you have to keep only the irq number (integer). But you
      need to initialize this number as -1 before actual allocation,
      since irq 0 is valid. The port address and its resource pointer
      can be initialized as null by
      <function>kzalloc()</function> automatically, so you
      don't have to take care of resetting them.
      </para>

      <para>
        The allocation of an I/O port is done like this:

        <informalexample>
          <programlisting>
<![CDATA[
  err = pci_request_regions(pci, "My Chip");
  if (err < 0) {
          kfree(chip);
          pci_disable_device(pci);
          return err;
  }
  chip->port = pci_resource_start(pci, 0);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        <!-- obsolete -->
        It will reserve the I/O port region of 8 bytes of the given
      PCI device. The returned value, chip-&gt;res_port, is allocated
      via <function>kmalloc()</function> by
      <function>request_region()</function>. The pointer must be
      released via <function>kfree()</function>, but there is a
```

```
      problem with this. This issue will be explained later.
      </para>

      <para>
        The allocation of an interrupt source is done like this:

        <informalexample>
          <programlisting>
<![CDATA[
  if (request_irq(pci->irq, snd_mychip_interrupt,
                  IRQF_SHARED, "My Chip", chip)) {
          printk(KERN_ERR "cannot grab irq %d\n", pci->irq);
          snd_mychip_free(chip);
          return -EBUSY;
  }
  chip->irq = pci->irq;
]]>
          </programlisting>
        </informalexample>

        where <function>snd_mychip_interrupt()</function> is the
      interrupt handler defined <link

linkend="pcm-interface-interrupt-handler"><citetitle>later</citetitle></link>.
      Note that chip-&gt;irq should be defined
      only when <function>request_irq()</function> succeeded.
      </para>

      <para>
      On the PCI bus, interrupts can be shared. Thus,
      <constant>IRQF_SHARED</constant> is used as the interrupt flag of
      <function>request_irq()</function>.
      </para>

      <para>
        The last argument of <function>request_irq()</function> is the
      data pointer passed to the interrupt handler. Usually, the
      chip-specific record is used for that, but you can use what you
      like, too.
      </para>

      <para>
        I won't give details about the interrupt handler at this
        point, but at least its appearance can be explained now. The
        interrupt handler looks usually like the following:

        <informalexample>
          <programlisting>
<![CDATA[
  static irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
  {
          struct mychip *chip = dev_id;
          ....
          return IRQ_HANDLED;
  }
]]>
```

```
          </programlisting>
        </informalexample>
      </para>

      <para>
        Now let's write the corresponding destructor for the resources
      above. The role of destructor is simple: disable the hardware
      (if already activated) and release the resources. So far, we
      have no hardware part, so the disabling code is not written here.
      </para>

      <para>
        To release the resources, the <quote>check-and-release</quote>
        method is a safer way. For the interrupt, do like this:

        <informalexample>
          <programlisting>
<![CDATA[
  if (chip->irq >= 0)
          free_irq(chip->irq, chip);
]]>
          </programlisting>
        </informalexample>

        Since the irq number can start from 0, you should initialize
        chip-&gt;irq with a negative value (e.g. -1), so that you can
        check the validity of the irq number as above.
      </para>

      <para>
        When you requested I/O ports or memory regions via
        <function>pci_request_region()</function> or
        <function>pci_request_regions()</function> like in this example,
        release the resource(s) using the corresponding function,
        <function>pci_release_region()</function> or
        <function>pci_release_regions()</function>.

        <informalexample>
          <programlisting>
<![CDATA[
  pci_release_regions(chip->pci);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        When you requested manually via <function>request_region()</function>
        or <function>request_mem_region</function>, you can release it via
        <function>release_resource()</function>.  Suppose that you keep
        the resource pointer returned from <function>request_region()</function>
        in chip-&gt;res_port, the release procedure looks like:

        <informalexample>
          <programlisting>
<![CDATA[
```

```
  release_and_free_resource(chip->res_port);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
      Don't forget to call <function>pci_disable_device()</function>
      before the end.
      </para>

      <para>
        And finally, release the chip-specific record.

        <informalexample>
          <programlisting>
<![CDATA[
  kfree(chip);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
      Again, remember that you cannot
      use the <parameter>__devexit</parameter> prefix for this destructor.
      </para>

      <para>
      We didn't implement the hardware disabling part in the above.
      If you need to do this, please note that the destructor may be
      called even before the initialization of the chip is completed.
      It would be better to have a flag to skip hardware disabling
      if the hardware was not initialized yet.
      </para>

      <para>
      When the chip-data is assigned to the card using
      <function>snd_device_new()</function> with
      <constant>SNDRV_DEV_LOWLELVEL</constant> , its destructor is
      called at the last.  That is, it is assured that all other
      components like PCMs and controls have already been released.
      You don't have to stop PCMs, etc. explicitly, but just
      call low-level hardware stopping.
      </para>

      <para>
        The management of a memory-mapped region is almost as same as
        the management of an I/O port. You'll need three fields like
        the following:

        <informalexample>
          <programlisting>
<![CDATA[
  struct mychip {
          ....
```

```
          unsigned long iobase_phys;
          void __iomem *iobase_virt;
  };
]]>
          </programlisting>
        </informalexample>

        and the allocation would be like below:

        <informalexample>
          <programlisting>
<![CDATA[
  if ((err = pci_request_regions(pci, "My Chip")) < 0) {
          kfree(chip);
          return err;
  }
  chip->iobase_phys = pci_resource_start(pci, 0);
  chip->iobase_virt = ioremap_nocache(chip->iobase_phys,
                                      pci_resource_len(pci, 0));
]]>
          </programlisting>
        </informalexample>

        and the corresponding destructor would be:

        <informalexample>
          <programlisting>
<![CDATA[
  static int snd_mychip_free(struct mychip *chip)
  {
          ....
          if (chip->iobase_virt)
                  iounmap(chip->iobase_virt);
          ....
          pci_release_regions(chip->pci);
          ....
  }
]]>
          </programlisting>
        </informalexample>
      </para>

    </section>

    <section id="pci-resource-device-struct">
      <title>Registration of Device Struct</title>
      <para>
        At some point, typically after calling
<function>snd_device_new()</function>,
        you need to register the struct <structname>device</structname> of the
chip
        you're handling for udev and co.  ALSA provides a macro for
compatibility with
        older kernels.  Simply call like the following:
        <informalexample>
          <programlisting>
```

```
<![CDATA[
  snd_card_set_dev(card, &pci->dev);
]]>
          </programlisting>
        </informalexample>
        so that it stores the PCI's device pointer to the card.  This will be
        referred by ALSA core functions later when the devices are registered.
      </para>
      <para>
        In the case of non-PCI, pass the proper device struct pointer of the BUS
        instead.  (In the case of legacy ISA without PnP, you don't have to do
        anything.)
      </para>
    </section>

    <section id="pci-resource-entries">
      <title>PCI Entries</title>
      <para>
        So far, so good. Let's finish the missing PCI
      stuff. At first, we need a
      <structname>pci_device_id</structname> table for this
      chipset. It's a table of PCI vendor/device ID number, and some
      masks.
      </para>

      <para>
        For example,

        <informalexample>
          <programlisting>
<![CDATA[
  static struct pci_device_id snd_mychip_ids[] = {
          { PCI_VENDOR_ID_FOO, PCI_DEVICE_ID_BAR,
            PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0, },
          ....
          { 0, }
  };
  MODULE_DEVICE_TABLE(pci, snd_mychip_ids);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The first and second fields of
      the <structname>pci_device_id</structname> structure are the vendor and
      device IDs. If you have no reason to filter the matching
      devices, you can leave the remaining fields as above. The last
      field of the <structname>pci_device_id</structname> struct contains
      private data for this entry. You can specify any value here, for
      example, to define specific operations for supported device IDs.
      Such an example is found in the intel8x0 driver.
      </para>

      <para>
        The last entry of this list is the terminator. You must
```

```
      specify this all-zero entry.
      </para>

      <para>
        Then, prepare the <structname>pci_driver</structname> record:

        <informalexample>
          <programlisting>
<![CDATA[
  static struct pci_driver driver = {
          .name = "My Own Chip",
          .id_table = snd_mychip_ids,
          .probe = snd_mychip_probe,
          .remove = __devexit_p(snd_mychip_remove),
  };
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The <structfield>probe</structfield> and
      <structfield>remove</structfield> functions have already
      been defined in the previous sections.
      The <structfield>remove</structfield> function should
      be defined with the
      <function>__devexit_p()</function> macro, so that it's not
      defined for built-in (and non-hot-pluggable) case. The
      <structfield>name</structfield>
      field is the name string of this device. Note that you must not
      use a slash <quote>/</quote> in this string.
      </para>

      <para>
        And at last, the module entries:

        <informalexample>
          <programlisting>
<![CDATA[
  static int __init alsa_card_mychip_init(void)
  {
          return pci_register_driver(&driver);
  }

  static void __exit alsa_card_mychip_exit(void)
  {
          pci_unregister_driver(&driver);
  }

  module_init(alsa_card_mychip_init)
  module_exit(alsa_card_mychip_exit)
]]>
          </programlisting>
        </informalexample>
      </para>
```

```
      <para>
        Note that these module entries are tagged with
      <parameter>__init</parameter> and
      <parameter>__exit</parameter> prefixes, not
      <parameter>__devinit</parameter> nor
      <parameter>__devexit</parameter>.
      </para>

      <para>
        Oh, one thing was forgotten. If you have no exported symbols,
        you need to declare it in 2.2 or 2.4 kernels (it's not necessary in 2.6
kernels).

        <informalexample>
          <programlisting>
<![CDATA[
  EXPORT_NO_SYMBOLS;
]]>
          </programlisting>
        </informalexample>

        That's all!
      </para>
    </section>
  </chapter>


<!-- ****************************************************** -->
<!-- PCM Interface  -->
<!-- ****************************************************** -->
  <chapter id="pcm-interface">
    <title>PCM Interface</title>

    <section id="pcm-interface-general">
      <title>General</title>
      <para>
        The PCM middle layer of ALSA is quite powerful and it is only
      necessary for each driver to implement the low-level functions
      to access its hardware.
      </para>

      <para>
        For accessing to the PCM layer, you need to include
      <filename>&lt;sound/pcm.h&gt;</filename> first. In addition,
      <filename>&lt;sound/pcm_params.h&gt;</filename> might be needed
      if you access to some functions related with hw_param.
      </para>

      <para>
        Each card device can have up to four pcm instances. A pcm
      instance corresponds to a pcm device file. The limitation of
      number of instances comes only from the available bit size of
      the Linux's device numbers. Once when 64bit device number is
      used, we'll have more pcm instances available.
      </para>
```

```
      <para>
        A pcm instance consists of pcm playback and capture streams,
      and each pcm stream consists of one or more pcm substreams. Some
      soundcards support multiple playback functions. For example,
      emu10k1 has a PCM playback of 32 stereo substreams. In this case, at
      each open, a free substream is (usually) automatically chosen
      and opened. Meanwhile, when only one substream exists and it was
      already opened, the successful open will either block
      or error with <constant>EAGAIN</constant> according to the
      file open mode. But you don't have to care about such details in your
      driver. The PCM middle layer will take care of such work.
      </para>
    </section>

    <section id="pcm-interface-example">
      <title>Full Code Example</title>
      <para>
      The example code below does not include any hardware access
      routines but shows only the skeleton, how to build up the PCM
      interfaces.

        <example>
          <title>PCM Example Code</title>
          <programlisting>
<![CDATA[
  #include <sound/pcm.h>
  ....

  /* hardware definition */
  static struct snd_pcm_hardware snd_mychip_playback_hw = {
          .info = (SNDRV_PCM_INFO_MMAP |
                   SNDRV_PCM_INFO_INTERLEAVED |
                   SNDRV_PCM_INFO_BLOCK_TRANSFER |
                   SNDRV_PCM_INFO_MMAP_VALID),
          .formats =          SNDRV_PCM_FMTBIT_S16_LE,
          .rates =            SNDRV_PCM_RATE_8000_48000,
          .rate_min =         8000,
          .rate_max =         48000,
          .channels_min =     2,
          .channels_max =     2,
          .buffer_bytes_max = 32768,
          .period_bytes_min = 4096,
          .period_bytes_max = 32768,
          .periods_min =      1,
          .periods_max =      1024,
  };

  /* hardware definition */
  static struct snd_pcm_hardware snd_mychip_capture_hw = {
          .info = (SNDRV_PCM_INFO_MMAP |
                   SNDRV_PCM_INFO_INTERLEAVED |
                   SNDRV_PCM_INFO_BLOCK_TRANSFER |
                   SNDRV_PCM_INFO_MMAP_VALID),
          .formats =          SNDRV_PCM_FMTBIT_S16_LE,
          .rates =            SNDRV_PCM_RATE_8000_48000,
          .rate_min =         8000,
```

```
        .rate_max =         48000,
        .channels_min =     2,
        .channels_max =     2,
        .buffer_bytes_max = 32768,
        .period_bytes_min = 4096,
        .period_bytes_max = 32768,
        .periods_min =      1,
        .periods_max =      1024,
};

/* open callback */
static int snd_mychip_playback_open(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        struct snd_pcm_runtime *runtime = substream->runtime;

        runtime->hw = snd_mychip_playback_hw;
        /* more hardware-initialization will be done here */
        ....
        return 0;
}

/* close callback */
static int snd_mychip_playback_close(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        /* the hardware-specific codes will be here */
        ....
        return 0;

}

/* open callback */
static int snd_mychip_capture_open(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        struct snd_pcm_runtime *runtime = substream->runtime;

        runtime->hw = snd_mychip_capture_hw;
        /* more hardware-initialization will be done here */
        ....
        return 0;
}

/* close callback */
static int snd_mychip_capture_close(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        /* the hardware-specific codes will be here */
        ....
        return 0;

}

/* hw_params callback */
static int snd_mychip_pcm_hw_params(struct snd_pcm_substream *substream,
```

```
                              struct snd_pcm_hw_params *hw_params)
{
        return snd_pcm_lib_malloc_pages(substream,
                                   params_buffer_bytes(hw_params));
}

/* hw_free callback */
static int snd_mychip_pcm_hw_free(struct snd_pcm_substream *substream)
{
        return snd_pcm_lib_free_pages(substream);
}

/* prepare callback */
static int snd_mychip_pcm_prepare(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        struct snd_pcm_runtime *runtime = substream->runtime;

        /* set up the hardware with the current configuration
         * for example...
         */
        mychip_set_sample_format(chip, runtime->format);
        mychip_set_sample_rate(chip, runtime->rate);
        mychip_set_channels(chip, runtime->channels);
        mychip_set_dma_setup(chip, runtime->dma_addr,
                             chip->buffer_size,
                             chip->period_size);
        return 0;
}

/* trigger callback */
static int snd_mychip_pcm_trigger(struct snd_pcm_substream *substream,
                                  int cmd)
{
        switch (cmd) {
        case SNDRV_PCM_TRIGGER_START:
                /* do something to start the PCM engine */
                ....
                break;
        case SNDRV_PCM_TRIGGER_STOP:
                /* do something to stop the PCM engine */
                ....
                break;
        default:
                return -EINVAL;
        }
}

/* pointer callback */
static snd_pcm_uframes_t
snd_mychip_pcm_pointer(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        unsigned int current_ptr;

        /* get the current hardware pointer */
```

```
        current_ptr = mychip_get_hw_pointer(chip);
        return current_ptr;
}

/* operators */
static struct snd_pcm_ops snd_mychip_playback_ops = {
        .open =        snd_mychip_playback_open,
        .close =       snd_mychip_playback_close,
        .ioctl =       snd_pcm_lib_ioctl,
        .hw_params =   snd_mychip_pcm_hw_params,
        .hw_free =     snd_mychip_pcm_hw_free,
        .prepare =     snd_mychip_pcm_prepare,
        .trigger =     snd_mychip_pcm_trigger,
        .pointer =     snd_mychip_pcm_pointer,
};

/* operators */
static struct snd_pcm_ops snd_mychip_capture_ops = {
        .open =        snd_mychip_capture_open,
        .close =       snd_mychip_capture_close,
        .ioctl =       snd_pcm_lib_ioctl,
        .hw_params =   snd_mychip_pcm_hw_params,
        .hw_free =     snd_mychip_pcm_hw_free,
        .prepare =     snd_mychip_pcm_prepare,
        .trigger =     snd_mychip_pcm_trigger,
        .pointer =     snd_mychip_pcm_pointer,
};

/*
 *  definitions of capture are omitted here...
 */

/* create a pcm device */
static int __devinit snd_mychip_new_pcm(struct mychip *chip)
{
        struct snd_pcm *pcm;
        int err;

        err = snd_pcm_new(chip->card, "My Chip", 0, 1, 1, &pcm);
        if (err < 0)
                return err;
        pcm->private_data = chip;
        strcpy(pcm->name, "My Chip");
        chip->pcm = pcm;
        /* set operators */
        snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK,
                        &snd_mychip_playback_ops);
        snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE,
                        &snd_mychip_capture_ops);
        /* pre-allocation of buffers */
        /* NOTE: this may fail */
        snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
                                              snd_dma_pci_data(chip->pci),
                                              64*1024, 64*1024);
        return 0;
}
```

```
]]>
          </programlisting>
        </example>
      </para>
    </section>

    <section id="pcm-interface-constructor">
      <title>Constructor</title>
      <para>
        A pcm instance is allocated by the <function>snd_pcm_new()</function>
      function. It would be better to create a constructor for pcm,
      namely,

        <informalexample>
          <programlisting>
<![CDATA[
  static int __devinit snd_mychip_new_pcm(struct mychip *chip)
  {
          struct snd_pcm *pcm;
          int err;

          err = snd_pcm_new(chip->card, "My Chip", 0, 1, 1, &pcm);
          if (err < 0)
                  return err;
          pcm->private_data = chip;
          strcpy(pcm->name, "My Chip");
          chip->pcm = pcm;
          ....
          return 0;
  }
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The <function>snd_pcm_new()</function> function takes four
      arguments. The first argument is the card pointer to which this
      pcm is assigned, and the second is the ID string.
      </para>

      <para>
        The third argument (<parameter>index</parameter>, 0 in the
      above) is the index of this new pcm. It begins from zero. If
      you create more than one pcm instances, specify the
      different numbers in this argument. For example,
      <parameter>index</parameter> = 1 for the second PCM device.
      </para>

      <para>
        The fourth and fifth arguments are the number of substreams
      for playback and capture, respectively. Here 1 is used for
      both arguments. When no playback or capture substreams are available,
      pass 0 to the corresponding argument.
      </para>
```

```
      <para>
        If a chip supports multiple playbacks or captures, you can
      specify more numbers, but they must be handled properly in
      open/close, etc. callbacks.  When you need to know which
      substream you are referring to, then it can be obtained from
      struct <structname>snd_pcm_substream</structname> data passed to each
callback
      as follows:

        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_pcm_substream *substream;
  int index = substream->number;
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        After the pcm is created, you need to set operators for each
        pcm stream.

        <informalexample>
          <programlisting>
<![CDATA[
  snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK,
                  &snd_mychip_playback_ops);
  snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE,
                  &snd_mychip_capture_ops);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The operators are defined typically like this:

        <informalexample>
          <programlisting>
<![CDATA[
  static struct snd_pcm_ops snd_mychip_playback_ops = {
          .open =        snd_mychip_pcm_open,
          .close =       snd_mychip_pcm_close,
          .ioctl =       snd_pcm_lib_ioctl,
          .hw_params =   snd_mychip_pcm_hw_params,
          .hw_free =     snd_mychip_pcm_hw_free,
          .prepare =     snd_mychip_pcm_prepare,
          .trigger =     snd_mychip_pcm_trigger,
          .pointer =     snd_mychip_pcm_pointer,
  };
]]>
          </programlisting>
        </informalexample>

        All the callbacks are described in the
```

```
        <link linkend="pcm-interface-operators"><citetitle>
        Operators</citetitle></link> subsection.
      </para>

      <para>
        After setting the operators, you probably will want to
        pre-allocate the buffer. For the pre-allocation, simply call
        the following:

        <informalexample>
          <programlisting>
<![CDATA[
  snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
                                        snd_dma_pci_data(chip->pci),
                                        64*1024, 64*1024);
]]>
          </programlisting>
        </informalexample>

        It will allocate a buffer up to 64kB as default.
      Buffer management details will be described in the later section <link
      linkend="buffer-and-memory"><citetitle>Buffer and Memory
      Management</citetitle></link>.
      </para>

      <para>
        Additionally, you can set some extra information for this pcm
        in pcm-&gt;info_flags.
        The available values are defined as
        <constant>SNDRV_PCM_INFO_XXX</constant> in
        <filename>&lt;sound/asound.h&gt;</filename>, which is used for
        the hardware definition (described later). When your soundchip
        supports only half-duplex, specify like this:

        <informalexample>
          <programlisting>
<![CDATA[
  pcm->info_flags = SNDRV_PCM_INFO_HALF_DUPLEX;
]]>
          </programlisting>
        </informalexample>
      </para>
    </section>

    <section id="pcm-interface-destructor">
      <title>... And the Destructor?</title>
      <para>
        The destructor for a pcm instance is not always
      necessary. Since the pcm device will be released by the middle
      layer code automatically, you don't have to call the destructor
      explicitly.
      </para>

      <para>
        The destructor would be necessary if you created
        special records internally and needed to release them. In such a
```

```
        case, set the destructor function to
        pcm-&gt;private_free:

        <example>
          <title>PCM Instance with a Destructor</title>
          <programlisting>
<![CDATA[
  static void mychip_pcm_free(struct snd_pcm *pcm)
  {
          struct mychip *chip = snd_pcm_chip(pcm);
          /* free your own data */
          kfree(chip->my_private_pcm_data);
          /* do what you like else */
          ....
  }

  static int __devinit snd_mychip_new_pcm(struct mychip *chip)
  {
          struct snd_pcm *pcm;
          ....
          /* allocate your own data */
          chip->my_private_pcm_data = kmalloc(...);
          /* set the destructor */
          pcm->private_data = chip;
          pcm->private_free = mychip_pcm_free;
          ....
  }
]]>
          </programlisting>
        </example>
      </para>
    </section>

    <section id="pcm-interface-runtime">
      <title>Runtime Pointer - The Chest of PCM Information</title>
        <para>
          When the PCM substream is opened, a PCM runtime instance is
        allocated and assigned to the substream. This pointer is
        accessible via <constant>substream-&gt;runtime</constant>.
        This runtime pointer holds most information you need
        to control the PCM: the copy of hw_params and sw_params configurations,
the buffer
        pointers, mmap records, spinlocks, etc.
        </para>

        <para>
        The definition of runtime instance is found in
        <filename>&lt;sound/pcm.h&gt;</filename>.  Here are
       the contents of this file:
          <informalexample>
            <programlisting>
<![CDATA[
struct _snd_pcm_runtime {
        /* -- Status -- */
        struct snd_pcm_substream *trigger_master;
        snd_timestamp_t trigger_tstamp; /* trigger timestamp */
```

```
int overrange;
snd_pcm_uframes_t avail_max;
snd_pcm_uframes_t hw_ptr_base;  /* Position at buffer restart */
snd_pcm_uframes_t hw_ptr_interrupt; /* Position at interrupt time*/

/* -- HW params -- */
snd_pcm_access_t access;        /* access mode */
snd_pcm_format_t format;        /* SNDRV_PCM_FORMAT_* */
snd_pcm_subformat_t subformat;  /* subformat */
unsigned int rate;              /* rate in Hz */
unsigned int channels;          /* channels */
snd_pcm_uframes_t period_size;  /* period size */
unsigned int periods;           /* periods */
snd_pcm_uframes_t buffer_size;  /* buffer size */
unsigned int tick_time;         /* tick time */
snd_pcm_uframes_t min_align;    /* Min alignment for the format */
size_t byte_align;
unsigned int frame_bits;
unsigned int sample_bits;
unsigned int info;
unsigned int rate_num;
unsigned int rate_den;

/* -- SW params -- */
struct timespec tstamp_mode;    /* mmap timestamp is updated */
unsigned int period_step;
unsigned int sleep_min;         /* min ticks to sleep */
snd_pcm_uframes_t start_threshold;
snd_pcm_uframes_t stop_threshold;
snd_pcm_uframes_t silence_threshold; /* Silence filling happens when
                                        noise is nearest than this */
snd_pcm_uframes_t silence_size; /* Silence filling size */
snd_pcm_uframes_t boundary;     /* pointers wrap point */

snd_pcm_uframes_t silenced_start;
snd_pcm_uframes_t silenced_size;

snd_pcm_sync_id_t sync;         /* hardware synchronization ID */

/* -- mmap -- */
volatile struct snd_pcm_mmap_status *status;
volatile struct snd_pcm_mmap_control *control;
atomic_t mmap_count;

/* -- locking / scheduling -- */
spinlock_t lock;
wait_queue_head_t sleep;
struct timer_list tick_timer;
struct fasync_struct *fasync;

/* -- private section -- */
void *private_data;
void (*private_free)(struct snd_pcm_runtime *runtime);

/* -- hardware description -- */
struct snd_pcm_hardware hw;
```

```
        struct snd_pcm_hw_constraints hw_constraints;

        /* -- interrupt callbacks -- */
        void (*transfer_ack_begin)(struct snd_pcm_substream *substream);
        void (*transfer_ack_end)(struct snd_pcm_substream *substream);

        /* -- timer -- */
        unsigned int timer_resolution;  /* timer resolution */

        /* -- DMA -- */
        unsigned char *dma_area;        /* DMA area */
        dma_addr_t dma_addr;            /* physical bus address (not accessible
from main CPU) */
        size_t dma_bytes;               /* size of DMA area */

        struct snd_dma_buffer *dma_buffer_p;    /* allocated buffer */

#if defined(CONFIG_SND_PCM_OSS) || defined(CONFIG_SND_PCM_OSS_MODULE)
        /* -- OSS things -- */
        struct snd_pcm_oss_runtime oss;
#endif
};
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          For the operators (callbacks) of each sound driver, most of
        these records are supposed to be read-only.  Only the PCM
        middle-layer changes / updates them.  The exceptions are
        the hardware description (hw), interrupt callbacks
        (transfer_ack_xxx), DMA buffer information, and the private
        data.  Besides, if you use the standard buffer allocation
        method via <function>snd_pcm_lib_malloc_pages()</function>,
        you don't need to set the DMA buffer information by yourself.
        </para>

        <para>
        In the sections below, important records are explained.
        </para>

        <section id="pcm-interface-runtime-hw">
        <title>Hardware Description</title>
        <para>
          The hardware descriptor (struct
<structname>snd_pcm_hardware</structname>)
        contains the definitions of the fundamental hardware
        configuration.  Above all, you'll need to define this in
        <link linkend="pcm-interface-operators-open-callback"><citetitle>
        the open callback</citetitle></link>.
        Note that the runtime instance holds the copy of the
        descriptor, not the pointer to the existing descriptor.  That
        is, in the open callback, you can modify the copied descriptor
        (<constant>runtime-&gt;hw</constant>) as you need.  For example, if the
maximum
```

```
        number of channels is 1 only on some chip models, you can
        still use the same hardware descriptor and change the
        channels_max later:
          <informalexample>
            <programlisting>
<![CDATA[
          struct snd_pcm_runtime *runtime = substream->runtime;
          ...
          runtime->hw = snd_mychip_playback_hw; /* common definition */
          if (chip->model == VERY_OLD_ONE)
                  runtime->hw.channels_max = 1;
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          Typically, you'll have a hardware descriptor as below:
          <informalexample>
            <programlisting>
<![CDATA[
  static struct snd_pcm_hardware snd_mychip_playback_hw = {
          .info = (SNDRV_PCM_INFO_MMAP |
                   SNDRV_PCM_INFO_INTERLEAVED |
                   SNDRV_PCM_INFO_BLOCK_TRANSFER |
                   SNDRV_PCM_INFO_MMAP_VALID),
          .formats =          SNDRV_PCM_FMTBIT_S16_LE,
          .rates =            SNDRV_PCM_RATE_8000_48000,
          .rate_min =         8000,
          .rate_max =         48000,
          .channels_min =     2,
          .channels_max =     2,
          .buffer_bytes_max = 32768,
          .period_bytes_min = 4096,
          .period_bytes_max = 32768,
          .periods_min =      1,
          .periods_max =      1024,
  };
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
        <itemizedlist>
        <listitem><para>
          The <structfield>info</structfield> field contains the type and
        capabilities of this pcm. The bit flags are defined in
        <filename>&lt;sound/asound.h&gt;</filename> as
        <constant>SNDRV_PCM_INFO_XXX</constant>. Here, at least, you
        have to specify whether the mmap is supported and which
        interleaved format is supported.
        When the is supported, add the
        <constant>SNDRV_PCM_INFO_MMAP</constant> flag here. When the
        hardware supports the interleaved or the non-interleaved
        formats, <constant>SNDRV_PCM_INFO_INTERLEAVED</constant> or
```

<constant>SNDRV_PCM_INFO_NONINTERLEAVED</constant> flag must
be set, respectively. If both are supported, you can set both,
too.
</para>

<para>
  In the above example, <constant>MMAP_VALID</constant> and
<constant>BLOCK_TRANSFER</constant> are specified for the OSS mmap
mode. Usually both are set. Of course,
<constant>MMAP_VALID</constant> is set only if the mmap is
really supported.
</para>

<para>
  The other possible flags are
<constant>SNDRV_PCM_INFO_PAUSE</constant> and
<constant>SNDRV_PCM_INFO_RESUME</constant>. The
<constant>PAUSE</constant> bit means that the pcm supports the
<quote>pause</quote> operation, while the
<constant>RESUME</constant> bit means that the pcm supports
the full <quote>suspend/resume</quote> operation.
If the <constant>PAUSE</constant> flag is set,
the <structfield>trigger</structfield> callback below
must handle the corresponding (pause push/release) commands.
The suspend/resume trigger commands can be defined even without
the <constant>RESUME</constant> flag. See <link
linkend="power-management"><citetitle>
Power Management</citetitle></link> section for details.
</para>

<para>
  When the PCM substreams can be synchronized (typically,
synchronized start/stop of a playback and a capture streams),
you can give <constant>SNDRV_PCM_INFO_SYNC_START</constant>,
too. In this case, you'll need to check the linked-list of
PCM substreams in the trigger callback. This will be
described in the later section.
</para>
</listitem>

<listitem>
<para>
  <structfield>formats</structfield> field contains the bit-flags
of supported formats (<constant>SNDRV_PCM_FMTBIT_XXX</constant>).
If the hardware supports more than one format, give all or'ed
bits. In the example above, the signed 16bit little-endian
format is specified.
</para>
</listitem>

<listitem>
<para>
<structfield>rates</structfield> field contains the bit-flags of
supported rates (<constant>SNDRV_PCM_RATE_XXX</constant>).
When the chip supports continuous rates, pass
<constant>CONTINUOUS</constant> bit additionally.

The pre-defined rate bits are provided only for typical
rates. If your chip supports unconventional rates, you need to add
the <constant>KNOT</constant> bit and set up the hardware
constraint manually (explained later).
</para>
</listitem>

<listitem>
<para>
<structfield>rate_min</structfield> and
<structfield>rate_max</structfield> define the minimum and
maximum sample rate. This should correspond somehow to
<structfield>rates</structfield> bits.
</para>
</listitem>

<listitem>
<para>
<structfield>channel_min</structfield> and
<structfield>channel_max</structfield>
define, as you might already expected, the minimum and maximum
number of channels.
</para>
</listitem>

<listitem>
<para>
<structfield>buffer_bytes_max</structfield> defines the
maximum buffer size in bytes. There is no
<structfield>buffer_bytes_min</structfield> field, since
it can be calculated from the minimum period size and the
minimum number of periods.
Meanwhile, <structfield>period_bytes_min</structfield> and
define the minimum and maximum size of the period in bytes.
<structfield>periods_max</structfield> and
<structfield>periods_min</structfield> define the maximum and
minimum number of periods in the buffer.
</para>

<para>
The <quote>period</quote> is a term that corresponds to
a fragment in the OSS world. The period defines the size at
which a PCM interrupt is generated. This size strongly
depends on the hardware.
Generally, the smaller period size will give you more
interrupts, that is, more controls.
In the case of capture, this size defines the input latency.
On the other hand, the whole buffer size defines the
output latency for the playback direction.
</para>
</listitem>

<listitem>
<para>
There is also a field <structfield>fifo_size</structfield>.
This specifies the size of the hardware FIFO, but currently it

```
        is neither used in the driver nor in the alsa-lib.  So, you
        can ignore this field.
        </para>
        </listitem>
        </itemizedlist>
        </para>
        </section>

        <section id="pcm-interface-runtime-config">
        <title>PCM Configurations</title>
        <para>
        Ok, let's go back again to the PCM runtime records.
        The most frequently referred records in the runtime instance are
        the PCM configurations.
        The PCM configurations are stored in the runtime instance
        after the application sends <type>hw_params</type> data via
        alsa-lib.  There are many fields copied from hw_params and
        sw_params structs.  For example,
        <structfield>format</structfield> holds the format type
        chosen by the application.  This field contains the enum value
        <constant>SNDRV_PCM_FORMAT_XXX</constant>.
        </para>

        <para>
        One thing to be noted is that the configured buffer and period
        sizes are stored in <quote>frames</quote> in the runtime.
        In the ALSA world, 1 frame = channels * samples-size.
        For conversion between frames and bytes, you can use the
        <function>frames_to_bytes()</function> and
          <function>bytes_to_frames()</function> helper functions.
          <informalexample>
            <programlisting>
<![CDATA[
  period_bytes = frames_to_bytes(runtime, runtime->period_size);
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
        Also, many software parameters (sw_params) are
        stored in frames, too.  Please check the type of the field.
        <type>snd_pcm_uframes_t</type> is for the frames as unsigned
        integer while <type>snd_pcm_sframes_t</type> is for the frames
        as signed integer.
        </para>
        </section>

        <section id="pcm-interface-runtime-dma">
        <title>DMA Buffer Information</title>
        <para>
        The DMA buffer is defined by the following four fields,
        <structfield>dma_area</structfield>,
        <structfield>dma_addr</structfield>,
        <structfield>dma_bytes</structfield> and
        <structfield>dma_private</structfield>.
```

```
        The <structfield>dma_area</structfield> holds the buffer
        pointer (the logical address).  You can call
        <function>memcpy</function> from/to
        this pointer.  Meanwhile, <structfield>dma_addr</structfield>
        holds the physical address of the buffer.  This field is
        specified only when the buffer is a linear buffer.
        <structfield>dma_bytes</structfield> holds the size of buffer
        in bytes.  <structfield>dma_private</structfield> is used for
        the ALSA DMA allocator.
        </para>

        <para>
        If you use a standard ALSA function,
        <function>snd_pcm_lib_malloc_pages()</function>, for
        allocating the buffer, these fields are set by the ALSA middle
        layer, and you should <emphasis>not</emphasis> change them by
        yourself.  You can read them but not write them.
        On the other hand, if you want to allocate the buffer by
        yourself, you'll need to manage it in hw_params callback.
        At least, <structfield>dma_bytes</structfield> is mandatory.
        <structfield>dma_area</structfield> is necessary when the
        buffer is mmapped.  If your driver doesn't support mmap, this
        field is not necessary.  <structfield>dma_addr</structfield>
        is also optional.  You can use
        <structfield>dma_private</structfield> as you like, too.
        </para>
        </section>

        <section id="pcm-interface-runtime-status">
        <title>Running Status</title>
        <para>
        The running status can be referred via
<constant>runtime-&gt;status</constant>.
        This is the pointer to the struct
<structname>snd_pcm_mmap_status</structname>
        record.  For example, you can get the current DMA hardware
        pointer via <constant>runtime-&gt;status-&gt;hw_ptr</constant>.
        </para>

        <para>
        The DMA application pointer can be referred via
        <constant>runtime-&gt;control</constant>, which points to the
        struct <structname>snd_pcm_mmap_control</structname> record.
        However, accessing directly to this value is not recommended.
        </para>
        </section>

        <section id="pcm-interface-runtime-private">
        <title>Private Data</title>
        <para>
        You can allocate a record for the substream and store it in
        <constant>runtime-&gt;private_data</constant>.  Usually, this
        is done in
        <link linkend="pcm-interface-operators-open-callback"><citetitle>
        the open callback</citetitle></link>.
        Don't mix this with <constant>pcm-&gt;private_data</constant>.
```

```
        The <constant>pcm-&gt;private_data</constant> usually points to the
        chip instance assigned statically at the creation of PCM, while the
        <constant>runtime-&gt;private_data</constant> points to a dynamic
        data structure created at the PCM open callback.

          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_xxx_open(struct snd_pcm_substream *substream)
  {
          struct my_pcm_data *data;
          ....
          data = kmalloc(sizeof(*data), GFP_KERNEL);
          substream->runtime->private_data = data;
          ....
  }
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          The allocated object must be released in
        <link linkend="pcm-interface-operators-open-callback"><citetitle>
        the close callback</citetitle></link>.
        </para>
        </section>

        <section id="pcm-interface-runtime-intr">
        <title>Interrupt Callbacks</title>
        <para>
        The field <structfield>transfer_ack_begin</structfield> and
        <structfield>transfer_ack_end</structfield> are called at
        the beginning and at the end of
        <function>snd_pcm_period_elapsed()</function>, respectively.
        </para>
        </section>

    </section>

    <section id="pcm-interface-operators">
      <title>Operators</title>
      <para>
        OK, now let me give details about each pcm callback
      (<parameter>ops</parameter>). In general, every callback must
      return 0 if successful, or a negative error number
      such as <constant>-EINVAL</constant>. To choose an appropriate
      error number, it is advised to check what value other parts of
      the kernel return when the same kind of request fails.
      </para>

      <para>
        The callback function takes at least the argument with
        <structname>snd_pcm_substream</structname> pointer. To retrieve
        the chip record from the given substream instance, you can use the
        following macro.
```

```
        <informalexample>
          <programlisting>
<![CDATA[
  int xxx() {
          struct mychip *chip = snd_pcm_substream_chip(substream);
          ....
  }
]]>
          </programlisting>
        </informalexample>

        The macro reads <constant>substream-&gt;private_data</constant>,
        which is a copy of <constant>pcm-&gt;private_data</constant>.
        You can override the former if you need to assign different data
        records per PCM substream.  For example, the cmi8330 driver assigns
        different private_data for playback and capture directions,
        because it uses two different codecs (SB- and AD-compatible) for
        different directions.
      </para>

      <section id="pcm-interface-operators-open-callback">
        <title>open callback</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_xxx_open(struct snd_pcm_substream *substream);
]]>
            </programlisting>
          </informalexample>

          This is called when a pcm substream is opened.
        </para>

        <para>
          At least, here you have to initialize the runtime-&gt;hw
          record. Typically, this is done by like this:

          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_xxx_open(struct snd_pcm_substream *substream)
  {
          struct mychip *chip = snd_pcm_substream_chip(substream);
          struct snd_pcm_runtime *runtime = substream->runtime;

          runtime->hw = snd_mychip_playback_hw;
          return 0;
  }
]]>
            </programlisting>
          </informalexample>

          where <parameter>snd_mychip_playback_hw</parameter> is the
          pre-defined hardware description.
```

```
        </para>

        <para>
        You can allocate a private data in this callback, as described
        in <link linkend="pcm-interface-runtime-private"><citetitle>
        Private Data</citetitle></link> section.
        </para>

        <para>
        If the hardware configuration needs more constraints, set the
        hardware constraints here, too.
        See <link linkend="pcm-interface-constraints"><citetitle>
        Constraints</citetitle></link> for more details.
        </para>
      </section>

      <section id="pcm-interface-operators-close-callback">
        <title>close callback</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_xxx_close(struct snd_pcm_substream *substream);
]]>
            </programlisting>
          </informalexample>

          Obviously, this is called when a pcm substream is closed.
        </para>

        <para>
          Any private instance for a pcm substream allocated in the
          open callback will be released here.

          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_xxx_close(struct snd_pcm_substream *substream)
  {
          ....
          kfree(substream->runtime->private_data);
          ....
  }
]]>
            </programlisting>
          </informalexample>
        </para>
      </section>

      <section id="pcm-interface-operators-ioctl-callback">
        <title>ioctl callback</title>
        <para>
          This is used for any special call to pcm ioctls. But
        usually you can pass a generic ioctl callback,
        <function>snd_pcm_lib_ioctl</function>.
        </para>
```

```
      </section>

      <section id="pcm-interface-operators-hw-params-callback">
        <title>hw_params callback</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_xxx_hw_params(struct snd_pcm_substream *substream,
                               struct snd_pcm_hw_params *hw_params);
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          This is called when the hardware parameter
        (<structfield>hw_params</structfield>) is set
        up by the application,
        that is, once when the buffer size, the period size, the
        format, etc. are defined for the pcm substream.
        </para>

        <para>
          Many hardware setups should be done in this callback,
        including the allocation of buffers.
        </para>

        <para>
          Parameters to be initialized are retrieved by
          <function>params_xxx()</function> macros. To allocate
          buffer, you can call a helper function,

          <informalexample>
            <programlisting>
<![CDATA[
  snd_pcm_lib_malloc_pages(substream, params_buffer_bytes(hw_params));
]]>
            </programlisting>
          </informalexample>

          <function>snd_pcm_lib_malloc_pages()</function> is available
          only when the DMA buffers have been pre-allocated.
          See the section <link
          linkend="buffer-and-memory-buffer-types"><citetitle>
          Buffer Types</citetitle></link> for more details.
        </para>

        <para>
          Note that this and <structfield>prepare</structfield> callbacks
        may be called multiple times per initialization.
        For example, the OSS emulation may
        call these callbacks at each change via its ioctl.
        </para>

        <para>
```

```
          Thus, you need to be careful not to allocate the same buffers
        many times, which will lead to memory leaks!  Calling the
        helper function above many times is OK. It will release the
        previous buffer automatically when it was already allocated.
        </para>

        <para>
          Another note is that this callback is non-atomic
        (schedulable). This is important, because the
        <structfield>trigger</structfield> callback
        is atomic (non-schedulable). That is, mutexes or any
        schedule-related functions are not available in
        <structfield>trigger</structfield> callback.
        Please see the subsection
        <link linkend="pcm-interface-atomicity"><citetitle>
        Atomicity</citetitle></link> for details.
        </para>
      </section>

      <section id="pcm-interface-operators-hw-free-callback">
        <title>hw_free callback</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_xxx_hw_free(struct snd_pcm_substream *substream);
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          This is called to release the resources allocated via
          <structfield>hw_params</structfield>. For example, releasing the
          buffer via
          <function>snd_pcm_lib_malloc_pages()</function> is done by
          calling the following:

          <informalexample>
            <programlisting>
<![CDATA[
  snd_pcm_lib_free_pages(substream);
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          This function is always called before the close callback is called.
          Also, the callback may be called multiple times, too.
          Keep track whether the resource was already released.
        </para>
      </section>

      <section id="pcm-interface-operators-prepare-callback">
       <title>prepare callback</title>
```

```
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_xxx_prepare(struct snd_pcm_substream *substream);
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          This callback is called when the pcm is
        <quote>prepared</quote>. You can set the format type, sample
        rate, etc. here. The difference from
        <structfield>hw_params</structfield> is that the
        <structfield>prepare</structfield> callback will be called each
        time
        <function>snd_pcm_prepare()</function> is called, i.e. when
        recovering after underruns, etc.
        </para>

        <para>
        Note that this callback is now non-atomic.
        You can use schedule-related functions safely in this callback.
        </para>

        <para>
          In this and the following callbacks, you can refer to the
        values via the runtime record,
        substream-&gt;runtime.
        For example, to get the current
        rate, format or channels, access to
        runtime-&gt;rate,
        runtime-&gt;format or
        runtime-&gt;channels, respectively.
        The physical address of the allocated buffer is set to
        runtime-&gt;dma_area.  The buffer and period sizes are
        in runtime-&gt;buffer_size and runtime-&gt;period_size,
        respectively.
        </para>

        <para>
          Be careful that this callback will be called many times at
        each setup, too.
        </para>
      </section>

      <section id="pcm-interface-operators-trigger-callback">
        <title>trigger callback</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_xxx_trigger(struct snd_pcm_substream *substream, int cmd);
]]>
            </programlisting>
```

```
          </informalexample>

          This is called when the pcm is started, stopped or paused.
        </para>

        <para>
          Which action is specified in the second argument,
          <constant>SNDRV_PCM_TRIGGER_XXX</constant> in
          <filename>&lt;sound/pcm.h&gt;</filename>. At least,
          the <constant>START</constant> and <constant>STOP</constant>
          commands must be defined in this callback.

          <informalexample>
            <programlisting>
<![CDATA[
  switch (cmd) {
  case SNDRV_PCM_TRIGGER_START:
          /* do something to start the PCM engine */
          break;
  case SNDRV_PCM_TRIGGER_STOP:
          /* do something to stop the PCM engine */
          break;
  default:
          return -EINVAL;
  }
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          When the pcm supports the pause operation (given in the info
        field of the hardware table), the <constant>PAUSE_PUSE</constant>
        and <constant>PAUSE_RELEASE</constant> commands must be
        handled here, too. The former is the command to pause the pcm,
        and the latter to restart the pcm again.
        </para>

        <para>
          When the pcm supports the suspend/resume operation,
        regardless of full or partial suspend/resume support,
        the <constant>SUSPEND</constant> and <constant>RESUME</constant>
        commands must be handled, too.
        These commands are issued when the power-management status is
        changed.  Obviously, the <constant>SUSPEND</constant> and
        <constant>RESUME</constant> commands
        suspend and resume the pcm substream, and usually, they
        are identical to the <constant>STOP</constant> and
        <constant>START</constant> commands, respectively.
          See the <link linkend="power-management"><citetitle>
        Power Management</citetitle></link> section for details.
        </para>

        <para>
          As mentioned, this callback is atomic.  You cannot call
          functions which may sleep.
```

```
          The trigger callback should be as minimal as possible,
          just really triggering the DMA.  The other stuff should be
          initialized hw_params and prepare callbacks properly
          beforehand.
        </para>
      </section>

      <section id="pcm-interface-operators-pointer-callback">
        <title>pointer callback</title>
        <para>
          <informalexample>
            <programlisting>
<![CDATA[
  static snd_pcm_uframes_t snd_xxx_pointer(struct snd_pcm_substream *substream)
]]>
            </programlisting>
          </informalexample>

          This callback is called when the PCM middle layer inquires
        the current hardware position on the buffer. The position must
        be returned in frames,
        ranging from 0 to buffer_size - 1.
        </para>

        <para>
          This is called usually from the buffer-update routine in the
        pcm middle layer, which is invoked when
        <function>snd_pcm_period_elapsed()</function> is called in the
        interrupt routine. Then the pcm middle layer updates the
        position and calculates the available space, and wakes up the
        sleeping poll threads, etc.
        </para>

        <para>
          This callback is also atomic.
        </para>
      </section>

      <section id="pcm-interface-operators-copy-silence">
        <title>copy and silence callbacks</title>
        <para>
          These callbacks are not mandatory, and can be omitted in
        most cases. These callbacks are used when the hardware buffer
        cannot be in the normal memory space. Some chips have their
        own buffer on the hardware which is not mappable. In such a
        case, you have to transfer the data manually from the memory
        buffer to the hardware buffer. Or, if the buffer is
        non-contiguous on both physical and virtual memory spaces,
        these callbacks must be defined, too.
        </para>

        <para>
          If these two callbacks are defined, copy and set-silence
        operations are done by them. The detailed will be described in
        the later section <link
        linkend="buffer-and-memory"><citetitle>Buffer and Memory
```

```
      Management</citetitle></link>.
      </para>
    </section>

    <section id="pcm-interface-operators-ack">
      <title>ack callback</title>
      <para>
        This callback is also not mandatory. This callback is called
      when the appl_ptr is updated in read or write operations.
      Some drivers like emu10k1-fx and cs46xx need to track the
      current appl_ptr for the internal buffer, and this callback
      is useful only for such a purpose.
      </para>
      <para>
        This callback is atomic.
      </para>
    </section>

    <section id="pcm-interface-operators-page-callback">
      <title>page callback</title>

      <para>
        This callback is optional too. This callback is used
      mainly for non-contiguous buffers. The mmap calls this
      callback to get the page address. Some examples will be
      explained in the later section <link
      linkend="buffer-and-memory"><citetitle>Buffer and Memory
      Management</citetitle></link>, too.
      </para>
    </section>
  </section>

  <section id="pcm-interface-interrupt-handler">
    <title>Interrupt Handler</title>
    <para>
      The rest of pcm stuff is the PCM interrupt handler. The
    role of PCM interrupt handler in the sound driver is to update
    the buffer position and to tell the PCM middle layer when the
    buffer position goes across the prescribed period size. To
    inform this, call the <function>snd_pcm_period_elapsed()</function>
    function.
    </para>

    <para>
      There are several types of sound chips to generate the interrupts.
    </para>

    <section id="pcm-interface-interrupt-handler-boundary">
      <title>Interrupts at the period (fragment) boundary</title>
      <para>
        This is the most frequently found type:  the hardware
      generates an interrupt at each period boundary.
      In this case, you can call
      <function>snd_pcm_period_elapsed()</function> at each
      interrupt.
      </para>
```

```
        <para>
          <function>snd_pcm_period_elapsed()</function> takes the
        substream pointer as its argument. Thus, you need to keep the
        substream pointer accessible from the chip instance. For
        example, define substream field in the chip record to hold the
        current running substream pointer, and set the pointer value
        at open callback (and reset at close callback).
        </para>

        <para>
          If you acquire a spinlock in the interrupt handler, and the
        lock is used in other pcm callbacks, too, then you have to
        release the lock before calling
        <function>snd_pcm_period_elapsed()</function>, because
        <function>snd_pcm_period_elapsed()</function> calls other pcm
        callbacks inside.
        </para>

        <para>
          Typical code would be like:

          <example>
            <title>Interrupt Handler Case #1</title>
            <programlisting>
<![CDATA[
  static irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
  {
          struct mychip *chip = dev_id;
          spin_lock(&chip->lock);
          ....
          if (pcm_irq_invoked(chip)) {
                  /* call updater, unlock before it */
                  spin_unlock(&chip->lock);
                  snd_pcm_period_elapsed(chip->substream);
                  spin_lock(&chip->lock);
                  /* acknowledge the interrupt if necessary */
          }
          ....
          spin_unlock(&chip->lock);
          return IRQ_HANDLED;
  }
]]>
            </programlisting>
          </example>
        </para>
      </section>

      <section id="pcm-interface-interrupt-handler-timer">
        <title>High frequency timer interrupts</title>
        <para>
        This happense when the hardware doesn't generate interrupts
        at the period boundary but issues timer interrupts at a fixed
        timer rate (e.g. es1968 or ymfpci drivers).
        In this case, you need to check the current hardware
        position and accumulate the processed sample length at each
```

```
        interrupt.  When the accumulated size exceeds the period
        size, call
        <function>snd_pcm_period_elapsed()</function> and reset the
        accumulator.
        </para>

        <para>
          Typical code would be like the following.

          <example>
            <title>Interrupt Handler Case #2</title>
            <programlisting>
<![CDATA[
  static irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
  {
          struct mychip *chip = dev_id;
          spin_lock(&chip->lock);
          ....
          if (pcm_irq_invoked(chip)) {
                  unsigned int last_ptr, size;
                  /* get the current hardware pointer (in frames) */
                  last_ptr = get_hw_ptr(chip);
                  /* calculate the processed frames since the
                   * last update
                   */
                  if (last_ptr < chip->last_ptr)
                          size = runtime->buffer_size + last_ptr
                                   - chip->last_ptr;
                  else
                          size = last_ptr - chip->last_ptr;
                  /* remember the last updated point */
                  chip->last_ptr = last_ptr;
                  /* accumulate the size */
                  chip->size += size;
                  /* over the period boundary? */
                  if (chip->size >= runtime->period_size) {
                          /* reset the accumulator */
                          chip->size %= runtime->period_size;
                          /* call updater */
                          spin_unlock(&chip->lock);
                          snd_pcm_period_elapsed(substream);
                          spin_lock(&chip->lock);
                  }
                  /* acknowledge the interrupt if necessary */
          }
          ....
          spin_unlock(&chip->lock);
          return IRQ_HANDLED;
  }
]]>
            </programlisting>
          </example>
        </para>
      </section>

      <section id="pcm-interface-interrupt-handler-both">
```

```
    <title>On calling <function>snd_pcm_period_elapsed()</function></title>
    <para>
      In both cases, even if more than one period are elapsed, you
    don't have to call
    <function>snd_pcm_period_elapsed()</function> many times. Call
    only once. And the pcm layer will check the current hardware
    pointer and update to the latest status.
    </para>
  </section>
</section>

<section id="pcm-interface-atomicity">
  <title>Atomicity</title>
  <para>
  One of the most important (and thus difficult to debug) problems
  in kernel programming are race conditions.
  In the Linux kernel, they are usually avoided via spin-locks, mutexes
  or semaphores.  In general, if a race condition can happen
  in an interrupt handler, it has to be managed atomically, and you
  have to use a spinlock to protect the critical session. If the
  critical section is not in interrupt handler code and
  if taking a relatively long time to execute is acceptable, you
  should use mutexes or semaphores instead.
  </para>

  <para>
  As already seen, some pcm callbacks are atomic and some are
  not.  For example, the <parameter>hw_params</parameter> callback is
  non-atomic, while <parameter>trigger</parameter> callback is
  atomic.  This means, the latter is called already in a spinlock
  held by the PCM middle layer. Please take this atomicity into
  account when you choose a locking scheme in the callbacks.
  </para>

  <para>
  In the atomic callbacks, you cannot use functions which may call
  <function>schedule</function> or go to
  <function>sleep</function>.  Semaphores and mutexes can sleep,
  and hence they cannot be used inside the atomic callbacks
  (e.g. <parameter>trigger</parameter> callback).
  To implement some delay in such a callback, please use
  <function>udelay()</function> or <function>mdelay()</function>.
  </para>

  <para>
  All three atomic callbacks (trigger, pointer, and ack) are
  called with local interrupts disabled.
  </para>

</section>
<section id="pcm-interface-constraints">
  <title>Constraints</title>
  <para>
    If your chip supports unconventional sample rates, or only the
  limited samples, you need to set a constraint for the
  condition.
```

```
      </para>

      <para>
        For example, in order to restrict the sample rates in the some
        supported values, use
        <function>snd_pcm_hw_constraint_list()</function>.
        You need to call this function in the open callback.

        <example>
          <title>Example of Hardware Constraints</title>
          <programlisting>
<![CDATA[
  static unsigned int rates[] =
          {4000, 10000, 22050, 44100};
  static struct snd_pcm_hw_constraint_list constraints_rates = {
          .count = ARRAY_SIZE(rates),
          .list = rates,
          .mask = 0,
  };

  static int snd_mychip_pcm_open(struct snd_pcm_substream *substream)
  {
          int err;
          ....
          err = snd_pcm_hw_constraint_list(substream->runtime, 0,
                                           SNDRV_PCM_HW_PARAM_RATE,
                                           &constraints_rates);
          if (err < 0)
                  return err;
          ....
  }
]]>
          </programlisting>
        </example>
      </para>

      <para>
        There are many different constraints.
        Look at <filename>sound/pcm.h</filename> for a complete list.
        You can even define your own constraint rules.
        For example, let's suppose my_chip can manage a substream of 1 channel
        if and only if the format is S16_LE, otherwise it supports any format
        specified in the <structname>snd_pcm_hardware</structname> structure (or
in any
        other constraint_list). You can build a rule like this:

        <example>
          <title>Example of Hardware Constraints for Channels</title>
          <programlisting>
<![CDATA[
  static int hw_rule_format_by_channels(struct snd_pcm_hw_params *params,
                                        struct snd_pcm_hw_rule *rule)
  {
          struct snd_interval *c = hw_param_interval(params,
                SNDRV_PCM_HW_PARAM_CHANNELS);
          struct snd_mask *f = hw_param_mask(params, SNDRV_PCM_HW_PARAM_FORMAT);
```

```
          struct snd_mask fmt;

          snd_mask_any(&fmt);    /* Init the struct */
          if (c->min < 2) {
                  fmt.bits[0] &= SNDRV_PCM_FMTBIT_S16_LE;
                  return snd_mask_refine(f, &fmt);
          }
          return 0;
  }
]]>
          </programlisting>
        </example>
      </para>

      <para>
        Then you need to call this function to add your rule:

       <informalexample>
         <programlisting>
<![CDATA[
  snd_pcm_hw_rule_add(substream->runtime, 0, SNDRV_PCM_HW_PARAM_CHANNELS,
                      hw_rule_channels_by_format, 0, SNDRV_PCM_HW_PARAM_FORMAT,
                      -1);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The rule function is called when an application sets the number of
        channels. But an application can set the format before the number of
        channels. Thus you also need to define the inverse rule:

       <example>
         <title>Example of Hardware Constraints for Channels</title>
         <programlisting>
<![CDATA[
  static int hw_rule_channels_by_format(struct snd_pcm_hw_params *params,
                                        struct snd_pcm_hw_rule *rule)
  {
          struct snd_interval *c = hw_param_interval(params,
                        SNDRV_PCM_HW_PARAM_CHANNELS);
          struct snd_mask *f = hw_param_mask(params, SNDRV_PCM_HW_PARAM_FORMAT);
          struct snd_interval ch;

          snd_interval_any(&ch);
          if (f->bits[0] == SNDRV_PCM_FMTBIT_S16_LE) {
                  ch.min = ch.max = 1;
                  ch.integer = 1;
                  return snd_interval_refine(c, &ch);
          }
          return 0;
  }
]]>
          </programlisting>
        </example>
```

```
      </para>

      <para>
      ...and in the open callback:
       <informalexample>
         <programlisting>
<![CDATA[
  snd_pcm_hw_rule_add(substream->runtime, 0, SNDRV_PCM_HW_PARAM_FORMAT,
                      hw_rule_format_by_channels, 0,
SNDRV_PCM_HW_PARAM_CHANNELS,
                      -1);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        I won't give more details here, rather I
        would like to say, <quote>Luke, use the source.</quote>
      </para>
    </section>

  </chapter>


<!-- ****************************************************** -->
<!-- Control Interface  -->
<!-- ****************************************************** -->
  <chapter id="control-interface">
    <title>Control Interface</title>

    <section id="control-interface-general">
      <title>General</title>
      <para>
        The control interface is used widely for many switches,
      sliders, etc. which are accessed from user-space. Its most
      important use is the mixer interface. In other words, since ALSA
      0.9.x, all the mixer stuff is implemented on the control kernel API.
      </para>

      <para>
        ALSA has a well-defined AC97 control module. If your chip
      supports only the AC97 and nothing else, you can skip this
      section.
      </para>

      <para>
        The control API is defined in
      <filename>&lt;sound/control.h&gt;</filename>.
      Include this file if you want to add your own controls.
      </para>
    </section>

    <section id="control-interface-definition">
      <title>Definition of Controls</title>
      <para>
```

```
        To create a new control, you need to define the
        following three
      callbacks: <structfield>info</structfield>,
      <structfield>get</structfield> and
      <structfield>put</structfield>. Then, define a
      struct <structname>snd_kcontrol_new</structname> record, such as:

        <example>
          <title>Definition of a Control</title>
          <programlisting>
<![CDATA[
  static struct snd_kcontrol_new my_control __devinitdata = {
          .iface = SNDRV_CTL_ELEM_IFACE_MIXER,
          .name = "PCM Playback Switch",
          .index = 0,
          .access = SNDRV_CTL_ELEM_ACCESS_READWRITE,
          .private_value = 0xffff,
          .info = my_control_info,
          .get = my_control_get,
          .put = my_control_put
  };
]]>
          </programlisting>
        </example>
      </para>

      <para>
        Most likely the control is created via
      <function>snd_ctl_new1()</function>, and in such a case, you can
      add the <parameter>__devinitdata</parameter> prefix to the
      definition as above.
      </para>

      <para>
        The <structfield>iface</structfield> field specifies the control
      type, <constant>SNDRV_CTL_ELEM_IFACE_XXX</constant>, which
      is usually <constant>MIXER</constant>.
      Use <constant>CARD</constant> for global controls that are not
      logically part of the mixer.
      If the control is closely associated with some specific device on
      the sound card, use <constant>HWDEP</constant>,
      <constant>PCM</constant>, <constant>RAWMIDI</constant>,
      <constant>TIMER</constant>, or <constant>SEQUENCER</constant>, and
      specify the device number with the
      <structfield>device</structfield> and
      <structfield>subdevice</structfield> fields.
      </para>

      <para>
        The <structfield>name</structfield> is the name identifier
      string. Since ALSA 0.9.x, the control name is very important,
      because its role is classified from its name. There are
      pre-defined standard control names. The details are described in
      the <link linkend="control-interface-control-names"><citetitle>
      Control Names</citetitle></link> subsection.
      </para>
```

```
  <para>
    The <structfield>index</structfield> field holds the index number
  of this control. If there are several different controls with
  the same name, they can be distinguished by the index
  number. This is the case when
  several codecs exist on the card. If the index is zero, you can
  omit the definition above.
  </para>

  <para>
    The <structfield>access</structfield> field contains the access
  type of this control. Give the combination of bit masks,
  <constant>SNDRV_CTL_ELEM_ACCESS_XXX</constant>, there.
  The details will be explained in
  the <link linkend="control-interface-access-flags"><citetitle>
  Access Flags</citetitle></link> subsection.
  </para>

  <para>
    The <structfield>private_value</structfield> field contains
  an arbitrary long integer value for this record. When using
  the generic <structfield>info</structfield>,
  <structfield>get</structfield> and
  <structfield>put</structfield> callbacks, you can pass a value
  through this field. If several small numbers are necessary, you can
  combine them in bitwise. Or, it's possible to give a pointer
  (casted to unsigned long) of some record to this field, too.
  </para>

  <para>
  The <structfield>tlv</structfield> field can be used to provide
  metadata about the control; see the
  <link linkend="control-interface-tlv">
  <citetitle>Metadata</citetitle></link> subsection.
  </para>

  <para>
    The other three are
    <link linkend="control-interface-callbacks"><citetitle>
    callback functions</citetitle></link>.
  </para>
</section>

<section id="control-interface-control-names">
  <title>Control Names</title>
  <para>
    There are some standards to define the control names. A
  control is usually defined from the three parts as
  <quote>SOURCE DIRECTION FUNCTION</quote>.
  </para>

  <para>
    The first, <constant>SOURCE</constant>, specifies the source
  of the control, and is a string such as <quote>Master</quote>,
  <quote>PCM</quote>, <quote>CD</quote> and
```

```
<quote>Line</quote>. There are many pre-defined sources.
</para>

<para>
  The second, <constant>DIRECTION</constant>, is one of the
following strings according to the direction of the control:
<quote>Playback</quote>, <quote>Capture</quote>, <quote>Bypass
Playback</quote> and <quote>Bypass Capture</quote>. Or, it can
be omitted, meaning both playback and capture directions.
</para>

<para>
  The third, <constant>FUNCTION</constant>, is one of the
following strings according to the function of the control:
<quote>Switch</quote>, <quote>Volume</quote> and
<quote>Route</quote>.
</para>

<para>
  The example of control names are, thus, <quote>Master Capture
Switch</quote> or <quote>PCM Playback Volume</quote>.
</para>

<para>
  There are some exceptions:
</para>

<section id="control-interface-control-names-global">
  <title>Global capture and playback</title>
  <para>
    <quote>Capture Source</quote>, <quote>Capture Switch</quote>
  and <quote>Capture Volume</quote> are used for the global
  capture (input) source, switch and volume. Similarly,
  <quote>Playback Switch</quote> and <quote>Playback
  Volume</quote> are used for the global output gain switch and
  volume.
  </para>
</section>

<section id="control-interface-control-names-tone">
  <title>Tone-controls</title>
  <para>
    tone-control switch and volumes are specified like
  <quote>Tone Control - XXX</quote>, e.g. <quote>Tone Control -
  Switch</quote>, <quote>Tone Control - Bass</quote>,
  <quote>Tone Control - Center</quote>.
  </para>
</section>

<section id="control-interface-control-names-3d">
  <title>3D controls</title>
  <para>
    3D-control switches and volumes are specified like <quote>3D
  Control - XXX</quote>, e.g. <quote>3D Control -
  Switch</quote>, <quote>3D Control - Center</quote>, <quote>3D
  Control - Space</quote>.
```

```
    </para>
  </section>

  <section id="control-interface-control-names-mic">
    <title>Mic boost</title>
    <para>
      Mic-boost switch is set as <quote>Mic Boost</quote> or
    <quote>Mic Boost (6dB)</quote>.
    </para>

    <para>
      More precise information can be found in
    <filename>Documentation/sound/alsa/ControlNames.txt</filename>.
    </para>
  </section>
</section>

<section id="control-interface-access-flags">
  <title>Access Flags</title>

  <para>
  The access flag is the bitmask which specifies the access type
  of the given control.  The default access type is
  <constant>SNDRV_CTL_ELEM_ACCESS_READWRITE</constant>,
  which means both read and write are allowed to this control.
  When the access flag is omitted (i.e. = 0), it is
  considered as <constant>READWRITE</constant> access as default.
  </para>

  <para>
  When the control is read-only, pass
  <constant>SNDRV_CTL_ELEM_ACCESS_READ</constant> instead.
  In this case, you don't have to define
  the <structfield>put</structfield> callback.
  Similarly, when the control is write-only (although it's a rare
  case), you can use the <constant>WRITE</constant> flag instead, and
  you don't need the <structfield>get</structfield> callback.
  </para>

  <para>
  If the control value changes frequently (e.g. the VU meter),
  <constant>VOLATILE</constant> flag should be given.  This means
  that the control may be changed without
  <link linkend="control-interface-change-notification"><citetitle>
  notification</citetitle></link>. Applications should poll such
  a control constantly.
  </para>

  <para>
  When the control is inactive, set
  the <constant>INACTIVE</constant> flag, too.
  There are <constant>LOCK</constant> and
  <constant>OWNER</constant> flags to change the write
  permissions.
  </para>
```

```
    </section>

    <section id="control-interface-callbacks">
      <title>Callbacks</title>

      <section id="control-interface-callbacks-info">
        <title>info callback</title>
        <para>
          The <structfield>info</structfield> callback is used to get
        detailed information on this control. This must store the
        values of the given struct <structname>snd_ctl_elem_info</structname>
        object. For example, for a boolean control with a single
        element:

          <example>
            <title>Example of info callback</title>
            <programlisting>
<![CDATA[
  static int snd_myctl_mono_info(struct snd_kcontrol *kcontrol,
                          struct snd_ctl_elem_info *uinfo)
  {
          uinfo->type = SNDRV_CTL_ELEM_TYPE_BOOLEAN;
          uinfo->count = 1;
          uinfo->value.integer.min = 0;
          uinfo->value.integer.max = 1;
          return 0;
  }
]]>
            </programlisting>
          </example>
        </para>

        <para>
          The <structfield>type</structfield> field specifies the type
        of the control. There are <constant>BOOLEAN</constant>,
        <constant>INTEGER</constant>, <constant>ENUMERATED</constant>,
        <constant>BYTES</constant>, <constant>IEC958</constant> and
        <constant>INTEGER64</constant>. The
        <structfield>count</structfield> field specifies the
        number of elements in this control. For example, a stereo
        volume would have count = 2. The
        <structfield>value</structfield> field is a union, and
        the values stored are depending on the type. The boolean and
        integer types are identical.
        </para>

        <para>
          The enumerated type is a bit different from others.  You'll
          need to set the string for the currently given item index.

          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_myctl_enum_info(struct snd_kcontrol *kcontrol,
                          struct snd_ctl_elem_info *uinfo)
  {
```

```
        static char *texts[4] = {
                "First", "Second", "Third", "Fourth"
        };
        uinfo->type = SNDRV_CTL_ELEM_TYPE_ENUMERATED;
        uinfo->count = 1;
        uinfo->value.enumerated.items = 4;
        if (uinfo->value.enumerated.item > 3)
                uinfo->value.enumerated.item = 3;
        strcpy(uinfo->value.enumerated.name,
               texts[uinfo->value.enumerated.item]);
        return 0;
  }
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        Some common info callbacks are available for your convenience:
      <function>snd_ctl_boolean_mono_info()</function> and
      <function>snd_ctl_boolean_stereo_info()</function>.
      Obviously, the former is an info callback for a mono channel
      boolean item, just like <function>snd_myctl_mono_info</function>
      above, and the latter is for a stereo channel boolean item.
      </para>

    </section>

    <section id="control-interface-callbacks-get">
      <title>get callback</title>

      <para>
        This callback is used to read the current value of the
      control and to return to user-space.
      </para>

      <para>
        For example,

        <example>
          <title>Example of get callback</title>
          <programlisting>
<![CDATA[
  static int snd_myctl_get(struct snd_kcontrol *kcontrol,
                           struct snd_ctl_elem_value *ucontrol)
  {
        struct mychip *chip = snd_kcontrol_chip(kcontrol);
        ucontrol->value.integer.value[0] = get_some_value(chip);
        return 0;
  }
]]>
          </programlisting>
        </example>
      </para>

      <para>
```

```
        The <structfield>value</structfield> field depends on
        the type of control as well as on the info callback.  For example,
        the sb driver uses this field to store the register offset,
        the bit-shift and the bit-mask.  The
        <structfield>private_value</structfield> field is set as follows:
          <informalexample>
            <programlisting>
<![CDATA[
  .private_value = reg | (shift << 16) | (mask << 24)
]]>
            </programlisting>
          </informalexample>
        and is retrieved in callbacks like
          <informalexample>
            <programlisting>
<![CDATA[
  static int snd_sbmixer_get_single(struct snd_kcontrol *kcontrol,
                                    struct snd_ctl_elem_value *ucontrol)
  {
          int reg = kcontrol->private_value & 0xff;
          int shift = (kcontrol->private_value >> 16) & 0xff;
          int mask = (kcontrol->private_value >> 24) & 0xff;
          ....
  }
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
        In the <structfield>get</structfield> callback,
        you have to fill all the elements if the
        control has more than one elements,
        i.e. <structfield>count</structfield> &gt; 1.
        In the example above, we filled only one element
        (<structfield>value.integer.value[0]</structfield>) since it's
        assumed as <structfield>count</structfield> = 1.
        </para>
      </section>

      <section id="control-interface-callbacks-put">
        <title>put callback</title>

        <para>
          This callback is used to write a value from user-space.
        </para>

        <para>
          For example,

          <example>
            <title>Example of put callback</title>
            <programlisting>
<![CDATA[
  static int snd_myctl_put(struct snd_kcontrol *kcontrol,
                           struct snd_ctl_elem_value *ucontrol)
```

```
  {
          struct mychip *chip = snd_kcontrol_chip(kcontrol);
          int changed = 0;
          if (chip->current_value !=
               ucontrol->value.integer.value[0]) {
                  change_current_value(chip,
                              ucontrol->value.integer.value[0]);
                  changed = 1;
          }
          return changed;
  }
]]>
            </programlisting>
          </example>

          As seen above, you have to return 1 if the value is
        changed. If the value is not changed, return 0 instead.
        If any fatal error happens, return a negative error code as
        usual.
        </para>

        <para>
        As in the <structfield>get</structfield> callback,
        when the control has more than one elements,
        all elements must be evaluated in this callback, too.
        </para>
      </section>

      <section id="control-interface-callbacks-all">
        <title>Callbacks are not atomic</title>
        <para>
          All these three callbacks are basically not atomic.
        </para>
      </section>
    </section>

    <section id="control-interface-constructor">
      <title>Constructor</title>
      <para>
        When everything is ready, finally we can create a new
      control. To create a control, there are two functions to be
      called, <function>snd_ctl_new1()</function> and
      <function>snd_ctl_add()</function>.
      </para>

      <para>
        In the simplest way, you can do like this:

        <informalexample>
          <programlisting>
<![CDATA[
  err = snd_ctl_add(card, snd_ctl_new1(&my_control, chip));
  if (err < 0)
          return err;
]]>
          </programlisting>
```

```
        </informalexample>

          where <parameter>my_control</parameter> is the
        struct <structname>snd_kcontrol_new</structname> object defined above, and
chip
        is the object pointer to be passed to
        kcontrol-&gt;private_data
        which can be referred to in callbacks.
        </para>

        <para>
          <function>snd_ctl_new1()</function> allocates a new
        <structname>snd_kcontrol</structname> instance (that's why the definition
        of <parameter>my_control</parameter> can be with
        the <parameter>__devinitdata</parameter>
        prefix), and <function>snd_ctl_add</function> assigns the given
        control component to the card.
        </para>
      </section>

      <section id="control-interface-change-notification">
        <title>Change Notification</title>
        <para>
          If you need to change and update a control in the interrupt
        routine, you can call <function>snd_ctl_notify()</function>. For
        example,

          <informalexample>
            <programlisting>
<![CDATA[
  snd_ctl_notify(card, SNDRV_CTL_EVENT_MASK_VALUE, id_pointer);
]]>
            </programlisting>
          </informalexample>

          This function takes the card pointer, the event-mask, and the
        control id pointer for the notification. The event-mask
        specifies the types of notification, for example, in the above
        example, the change of control values is notified.
        The id pointer is the pointer of struct
<structname>snd_ctl_elem_id</structname>
        to be notified.
        You can find some examples in <filename>es1938.c</filename> or
        <filename>es1968.c</filename> for hardware volume interrupts.
        </para>
      </section>

      <section id="control-interface-tlv">
        <title>Metadata</title>
        <para>
        To provide information about the dB values of a mixer control, use
        on of the <constant>DECLARE_TLV_xxx</constant> macros from
        <filename>&lt;sound/tlv.h&gt;</filename> to define a variable
        containing this information, set the<structfield>tlv.p
        </structfield> field to point to this variable, and include the
        <constant>SNDRV_CTL_ELEM_ACCESS_TLV_READ</constant> flag in the
```

```
      <structfield>access</structfield> field; like this:
      <informalexample>
        <programlisting>
<![CDATA[
  static DECLARE_TLV_DB_SCALE(db_scale_my_control, -4050, 150, 0);

  static struct snd_kcontrol_new my_control __devinitdata = {
          ...
          .access = SNDRV_CTL_ELEM_ACCESS_READWRITE |
                    SNDRV_CTL_ELEM_ACCESS_TLV_READ,
          ...
          .tlv.p = db_scale_my_control,
  };
]]>
        </programlisting>
      </informalexample>
      </para>

      <para>
      The <function>DECLARE_TLV_DB_SCALE</function> macro defines
      information about a mixer control where each step in the control's
      value changes the dB value by a constant dB amount.
      The first parameter is the name of the variable to be defined.
      The second parameter is the minimum value, in units of 0.01 dB.
      The third parameter is the step size, in units of 0.01 dB.
      Set the fourth parameter to 1 if the minimum value actually mutes
      the control.
      </para>

      <para>
      The <function>DECLARE_TLV_DB_LINEAR</function> macro defines
      information about a mixer control where the control's value affects
      the output linearly.
      The first parameter is the name of the variable to be defined.
      The second parameter is the minimum value, in units of 0.01 dB.
      The third parameter is the maximum value, in units of 0.01 dB.
      If the minimum value mutes the control, set the second parameter to
      <constant>TLV_DB_GAIN_MUTE</constant>.
      </para>
    </section>

  </chapter>


<!-- ****************************************************** -->
<!-- API for AC97 Codec  -->
<!-- ****************************************************** -->
  <chapter id="api-ac97">
    <title>API for AC97 Codec</title>

    <section>
      <title>General</title>
      <para>
        The ALSA AC97 codec layer is a well-defined one, and you don't
      have to write much code to control it. Only low-level control
      routines are necessary. The AC97 codec API is defined in
```

```
      <filename>&lt;sound/ac97_codec.h&gt;</filename>.
      </para>
    </section>

    <section id="api-ac97-example">
      <title>Full Code Example</title>
      <para>
          <example>
            <title>Example of AC97 Interface</title>
            <programlisting>
<![CDATA[
  struct mychip {
          ....
          struct snd_ac97 *ac97;
          ....
  };

  static unsigned short snd_mychip_ac97_read(struct snd_ac97 *ac97,
                                             unsigned short reg)
  {
          struct mychip *chip = ac97->private_data;
          ....
          /* read a register value here from the codec */
          return the_register_value;
  }

  static void snd_mychip_ac97_write(struct snd_ac97 *ac97,
                                   unsigned short reg, unsigned short val)
  {
          struct mychip *chip = ac97->private_data;
          ....
          /* write the given register value to the codec */
  }

  static int snd_mychip_ac97(struct mychip *chip)
  {
          struct snd_ac97_bus *bus;
          struct snd_ac97_template ac97;
          int err;
          static struct snd_ac97_bus_ops ops = {
                  .write = snd_mychip_ac97_write,
                  .read = snd_mychip_ac97_read,
          };

          err = snd_ac97_bus(chip->card, 0, &ops, NULL, &bus);
          if (err < 0)
                  return err;
          memset(&ac97, 0, sizeof(ac97));
          ac97.private_data = chip;
          return snd_ac97_mixer(bus, &ac97, &chip->ac97);
  }

]]>
          </programlisting>
        </example>
      </para>
```

```
    </section>

    <section id="api-ac97-constructor">
      <title>Constructor</title>
      <para>
        To create an ac97 instance, first call <function>snd_ac97_bus</function>
      with an <type>ac97_bus_ops_t</type> record with callback functions.

        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_ac97_bus *bus;
  static struct snd_ac97_bus_ops ops = {
        .write = snd_mychip_ac97_write,
        .read = snd_mychip_ac97_read,
  };

  snd_ac97_bus(card, 0, &ops, NULL, &pbus);
]]>
          </programlisting>
        </informalexample>

      The bus record is shared among all belonging ac97 instances.
      </para>

      <para>
      And then call <function>snd_ac97_mixer()</function> with an
      struct <structname>snd_ac97_template</structname>
      record together with the bus pointer created above.

        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_ac97_template ac97;
  int err;

  memset(&ac97, 0, sizeof(ac97));
  ac97.private_data = chip;
  snd_ac97_mixer(bus, &ac97, &chip->ac97);
]]>
          </programlisting>
        </informalexample>

        where chip-&gt;ac97 is a pointer to a newly created
        <type>ac97_t</type> instance.
        In this case, the chip pointer is set as the private data, so that
        the read/write callback functions can refer to this chip instance.
        This instance is not necessarily stored in the chip
        record.  If you need to change the register values from the
        driver, or need the suspend/resume of ac97 codecs, keep this
        pointer to pass to the corresponding functions.
      </para>
    </section>

    <section id="api-ac97-callbacks">
      <title>Callbacks</title>
```

```
      <para>
        The standard callbacks are <structfield>read</structfield> and
      <structfield>write</structfield>. Obviously they
      correspond to the functions for read and write accesses to the
      hardware low-level codes.
      </para>

      <para>
        The <structfield>read</structfield> callback returns the
        register value specified in the argument.

        <informalexample>
          <programlisting>
<![CDATA[
  static unsigned short snd_mychip_ac97_read(struct snd_ac97 *ac97,
                                             unsigned short reg)
  {
          struct mychip *chip = ac97->private_data;
          ....
          return the_register_value;
  }
]]>
          </programlisting>
        </informalexample>

        Here, the chip can be cast from ac97-&gt;private_data.
      </para>

      <para>
        Meanwhile, the <structfield>write</structfield> callback is
        used to set the register value.

        <informalexample>
          <programlisting>
<![CDATA[
  static void snd_mychip_ac97_write(struct snd_ac97 *ac97,
                       unsigned short reg, unsigned short val)
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
      These callbacks are non-atomic like the control API callbacks.
      </para>

      <para>
        There are also other callbacks:
      <structfield>reset</structfield>,
      <structfield>wait</structfield> and
      <structfield>init</structfield>.
      </para>

      <para>
        The <structfield>reset</structfield> callback is used to reset
      the codec. If the chip requires a special kind of reset, you can
```

```
      define this callback.
      </para>

      <para>
        The <structfield>wait</structfield> callback is used to
      add some waiting time in the standard initialization of the codec. If the
      chip requires the extra waiting time, define this callback.
      </para>

      <para>
        The <structfield>init</structfield> callback is used for
      additional initialization of the codec.
      </para>
    </section>

    <section id="api-ac97-updating-registers">
      <title>Updating Registers in The Driver</title>
      <para>
        If you need to access to the codec from the driver, you can
      call the following functions:
      <function>snd_ac97_write()</function>,
      <function>snd_ac97_read()</function>,
      <function>snd_ac97_update()</function> and
      <function>snd_ac97_update_bits()</function>.
      </para>

      <para>
        Both <function>snd_ac97_write()</function> and
        <function>snd_ac97_update()</function> functions are used to
        set a value to the given register
        (<constant>AC97_XXX</constant>). The difference between them is
        that <function>snd_ac97_update()</function> doesn't write a
        value if the given value has been already set, while
        <function>snd_ac97_write()</function> always rewrites the
        value.

        <informalexample>
          <programlisting>
<![CDATA[
  snd_ac97_write(ac97, AC97_MASTER, 0x8080);
  snd_ac97_update(ac97, AC97_MASTER, 0x8080);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        <function>snd_ac97_read()</function> is used to read the value
        of the given register. For example,

        <informalexample>
          <programlisting>
<![CDATA[
  value = snd_ac97_read(ac97, AC97_MASTER);
]]>
          </programlisting>
```

```
        </informalexample>
      </para>

      <para>
        <function>snd_ac97_update_bits()</function> is used to update
        some bits in the given register.

        <informalexample>
          <programlisting>
<![CDATA[
  snd_ac97_update_bits(ac97, reg, mask, value);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        Also, there is a function to change the sample rate (of a
        given register such as
        <constant>AC97_PCM_FRONT_DAC_RATE</constant>) when VRA or
        DRA is supported by the codec:
        <function>snd_ac97_set_rate()</function>.

        <informalexample>
          <programlisting>
<![CDATA[
  snd_ac97_set_rate(ac97, AC97_PCM_FRONT_DAC_RATE, 44100);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The following registers are available to set the rate:
      <constant>AC97_PCM_MIC_ADC_RATE</constant>,
      <constant>AC97_PCM_FRONT_DAC_RATE</constant>,
      <constant>AC97_PCM_LR_ADC_RATE</constant>,
      <constant>AC97_SPDIF</constant>. When
      <constant>AC97_SPDIF</constant> is specified, the register is
      not really changed but the corresponding IEC958 status bits will
      be updated.
      </para>
    </section>

    <section id="api-ac97-clock-adjustment">
      <title>Clock Adjustment</title>
      <para>
        In some chips, the clock of the codec isn't 48000 but using a
      PCI clock (to save a quartz!). In this case, change the field
      bus-&gt;clock to the corresponding
      value. For example, intel8x0
      and es1968 drivers have their own function to read from the clock.
      </para>
    </section>

    <section id="api-ac97-proc-files">
```

```
      <title>Proc Files</title>
      <para>
        The ALSA AC97 interface will create a proc file such as
      <filename>/proc/asound/card0/codec97#0/ac97#0-0</filename> and
      <filename>ac97#0-0+regs</filename>. You can refer to these files to
      see the current status and registers of the codec.
      </para>
    </section>

    <section id="api-ac97-multiple-codecs">
      <title>Multiple Codecs</title>
      <para>
        When there are several codecs on the same card, you need to
      call <function>snd_ac97_mixer()</function> multiple times with
      ac97.num=1 or greater. The <structfield>num</structfield> field
      specifies the codec number.
      </para>

      <para>
        If you set up multiple codecs, you either need to write
      different callbacks for each codec or check
      ac97-&gt;num in the callback routines.
      </para>
    </section>

  </chapter>


<!-- ****************************************************** -->
<!-- MIDI (MPU401-UART) Interface  -->
<!-- ****************************************************** -->
  <chapter id="midi-interface">
    <title>MIDI (MPU401-UART) Interface</title>

    <section id="midi-interface-general">
      <title>General</title>
      <para>
        Many soundcards have built-in MIDI (MPU401-UART)
      interfaces. When the soundcard supports the standard MPU401-UART
      interface, most likely you can use the ALSA MPU401-UART API. The
      MPU401-UART API is defined in
      <filename>&lt;sound/mpu401.h&gt;</filename>.
      </para>

      <para>
        Some soundchips have a similar but slightly different
      implementation of mpu401 stuff. For example, emu10k1 has its own
      mpu401 routines.
      </para>
    </section>

    <section id="midi-interface-constructor">
      <title>Constructor</title>
      <para>
        To create a rawmidi object, call
      <function>snd_mpu401_uart_new()</function>.
```

```
        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_rawmidi *rmidi;
  snd_mpu401_uart_new(card, 0, MPU401_HW_MPU401, port, info_flags,
                      irq, irq_flags, &rmidi);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The first argument is the card pointer, and the second is the
      index of this component. You can create up to 8 rawmidi
      devices.
      </para>

      <para>
        The third argument is the type of the hardware,
      <constant>MPU401_HW_XXX</constant>. If it's not a special one,
      you can use <constant>MPU401_HW_MPU401</constant>.
      </para>

      <para>
        The 4th argument is the I/O port address. Many
      backward-compatible MPU401 have an I/O port such as 0x330. Or, it
      might be a part of its own PCI I/O region. It depends on the
      chip design.
      </para>

      <para>
        The 5th argument is a bitflag for additional information.
        When the I/O port address above is part of the PCI I/O
      region, the MPU401 I/O port might have been already allocated
      (reserved) by the driver itself. In such a case, pass a bit flag
      <constant>MPU401_INFO_INTEGRATED</constant>,
      and the mpu401-uart layer will allocate the I/O ports by itself.
      </para>

        <para>
        When the controller supports only the input or output MIDI stream,
        pass the <constant>MPU401_INFO_INPUT</constant> or
        <constant>MPU401_INFO_OUTPUT</constant> bitflag, respectively.
        Then the rawmidi instance is created as a single stream.
        </para>

        <para>
        <constant>MPU401_INFO_MMIO</constant> bitflag is used to change
        the access method to MMIO (via readb and writeb) instead of
        iob and outb. In this case, you have to pass the iomapped address
        to <function>snd_mpu401_uart_new()</function>.
        </para>

        <para>
        When <constant>MPU401_INFO_TX_IRQ</constant> is set, the output
```

```
        stream isn't checked in the default interrupt handler.  The driver
        needs to call <function>snd_mpu401_uart_interrupt_tx()</function>
        by itself to start processing the output stream in the irq handler.
        </para>

      <para>
        Usually, the port address corresponds to the command port and
        port + 1 corresponds to the data port. If not, you may change
        the <structfield>cport</structfield> field of
        struct <structname>snd_mpu401</structname> manually
        afterward. However, <structname>snd_mpu401</structname> pointer is not
        returned explicitly by
        <function>snd_mpu401_uart_new()</function>. You need to cast
        rmidi-&gt;private_data to
        <structname>snd_mpu401</structname> explicitly,

        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_mpu401 *mpu;
  mpu = rmidi->private_data;
]]>
          </programlisting>
        </informalexample>

        and reset the cport as you like:

        <informalexample>
          <programlisting>
<![CDATA[
  mpu->cport = my_own_control_port;
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The 6th argument specifies the irq number for UART. If the irq
      is already allocated, pass 0 to the 7th argument
      (<parameter>irq_flags</parameter>). Otherwise, pass the flags
      for irq allocation
      (<constant>SA_XXX</constant> bits) to it, and the irq will be
      reserved by the mpu401-uart layer. If the card doesn't generate
      UART interrupts, pass -1 as the irq number. Then a timer
      interrupt will be invoked for polling.
      </para>
    </section>

    <section id="midi-interface-interrupt-handler">
      <title>Interrupt Handler</title>
      <para>
        When the interrupt is allocated in
      <function>snd_mpu401_uart_new()</function>, the private
      interrupt handler is used, hence you don't have anything else to do
      than creating the mpu401 stuff. Otherwise, you have to call
      <function>snd_mpu401_uart_interrupt()</function> explicitly when
```

```
      a UART interrupt is invoked and checked in your own interrupt
      handler.
      </para>

      <para>
        In this case, you need to pass the private_data of the
        returned rawmidi object from
        <function>snd_mpu401_uart_new()</function> as the second
        argument of <function>snd_mpu401_uart_interrupt()</function>.

        <informalexample>
          <programlisting>
<![CDATA[
  snd_mpu401_uart_interrupt(irq, rmidi->private_data, regs);
]]>
          </programlisting>
        </informalexample>
      </para>
    </section>

  </chapter>


<!-- ****************************************************** -->
<!-- RawMIDI Interface  -->
<!-- ****************************************************** -->
  <chapter id="rawmidi-interface">
    <title>RawMIDI Interface</title>

    <section id="rawmidi-interface-overview">
      <title>Overview</title>

      <para>
      The raw MIDI interface is used for hardware MIDI ports that can
      be accessed as a byte stream.  It is not used for synthesizer
      chips that do not directly understand MIDI.
      </para>

      <para>
      ALSA handles file and buffer management.  All you have to do is
      to write some code to move data between the buffer and the
      hardware.
      </para>

      <para>
      The rawmidi API is defined in
      <filename>&lt;sound/rawmidi.h&gt;</filename>.
      </para>
    </section>

    <section id="rawmidi-interface-constructor">
      <title>Constructor</title>

      <para>
      To create a rawmidi device, call the
      <function>snd_rawmidi_new</function> function:
```

```
        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_rawmidi *rmidi;
  err = snd_rawmidi_new(chip->card, "MyMIDI", 0, outs, ins, &rmidi);
  if (err < 0)
          return err;
  rmidi->private_data = chip;
  strcpy(rmidi->name, "My MIDI");
  rmidi->info_flags = SNDRV_RAWMIDI_INFO_OUTPUT |
                      SNDRV_RAWMIDI_INFO_INPUT |
                      SNDRV_RAWMIDI_INFO_DUPLEX;
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
      The first argument is the card pointer, the second argument is
      the ID string.
      </para>

      <para>
      The third argument is the index of this component.  You can
      create up to 8 rawmidi devices.
      </para>

      <para>
      The fourth and fifth arguments are the number of output and
      input substreams, respectively, of this device (a substream is
      the equivalent of a MIDI port).
      </para>

      <para>
      Set the <structfield>info_flags</structfield> field to specify
      the capabilities of the device.
      Set <constant>SNDRV_RAWMIDI_INFO_OUTPUT</constant> if there is
      at least one output port,
      <constant>SNDRV_RAWMIDI_INFO_INPUT</constant> if there is at
      least one input port,
      and <constant>SNDRV_RAWMIDI_INFO_DUPLEX</constant> if the device
      can handle output and input at the same time.
      </para>

      <para>
      After the rawmidi device is created, you need to set the
      operators (callbacks) for each substream.  There are helper
      functions to set the operators for all the substreams of a device:
        <informalexample>
          <programlisting>
<![CDATA[
  snd_rawmidi_set_ops(rmidi, SNDRV_RAWMIDI_STREAM_OUTPUT,
&snd_mymidi_output_ops);
  snd_rawmidi_set_ops(rmidi, SNDRV_RAWMIDI_STREAM_INPUT, &snd_mymidi_input_ops);
]]>
          </programlisting>
```

```
        </informalexample>
      </para>

      <para>
      The operators are usually defined like this:
        <informalexample>
          <programlisting>
<![CDATA[
  static struct snd_rawmidi_ops snd_mymidi_output_ops = {
          .open =    snd_mymidi_output_open,
          .close =   snd_mymidi_output_close,
          .trigger = snd_mymidi_output_trigger,
  };
]]>
          </programlisting>
        </informalexample>
      These callbacks are explained in the <link

linkend="rawmidi-interface-callbacks"><citetitle>Callbacks</citetitle></link>
      section.
      </para>

      <para>
      If there are more than one substream, you should give a
      unique name to each of them:
        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_rawmidi_substream *substream;
  list_for_each_entry(substream,
                      &rmidi->streams[SNDRV_RAWMIDI_STREAM_OUTPUT].substreams,
                      list {
          sprintf(substream->name, "My MIDI Port %d", substream->number + 1);
  }
  /* same for SNDRV_RAWMIDI_STREAM_INPUT */
]]>
          </programlisting>
        </informalexample>
      </para>
    </section>

    <section id="rawmidi-interface-callbacks">
      <title>Callbacks</title>

      <para>
      In all the callbacks, the private data that you've set for the
      rawmidi device can be accessed as
      substream-&gt;rmidi-&gt;private_data.
      <!-- <code> isn't available before DocBook 4.3 -->
      </para>

      <para>
      If there is more than one port, your callbacks can determine the
      port index from the struct snd_rawmidi_substream data passed to each
      callback:
        <informalexample>
```

```
          <programlisting>
<![CDATA[
  struct snd_rawmidi_substream *substream;
  int index = substream->number;
]]>
          </programlisting>
        </informalexample>
      </para>

      <section id="rawmidi-interface-op-open">
      <title><function>open</function> callback</title>

        <informalexample>
          <programlisting>
<![CDATA[
  static int snd_xxx_open(struct snd_rawmidi_substream *substream);
]]>
          </programlisting>
        </informalexample>

        <para>
        This is called when a substream is opened.
        You can initialize the hardware here, but you shouldn't
        start transmitting/receiving data yet.
        </para>
      </section>

      <section id="rawmidi-interface-op-close">
      <title><function>close</function> callback</title>

        <informalexample>
          <programlisting>
<![CDATA[
  static int snd_xxx_close(struct snd_rawmidi_substream *substream);
]]>
          </programlisting>
        </informalexample>

        <para>
        Guess what.
        </para>

        <para>
        The <function>open</function> and <function>close</function>
        callbacks of a rawmidi device are serialized with a mutex,
        and can sleep.
        </para>
      </section>

      <section id="rawmidi-interface-op-trigger-out">
      <title><function>trigger</function> callback for output
      substreams</title>

        <informalexample>
          <programlisting>
<![CDATA[
```

```
  static void snd_xxx_output_trigger(struct snd_rawmidi_substream *substream,
int up);
]]>
          </programlisting>
        </informalexample>

        <para>
        This is called with a nonzero <parameter>up</parameter>
        parameter when there is some data in the substream buffer that
        must be transmitted.
        </para>

        <para>
        To read data from the buffer, call
        <function>snd_rawmidi_transmit_peek</function>.  It will
        return the number of bytes that have been read; this will be
        less than the number of bytes requested when there are no more
        data in the buffer.
        After the data have been transmitted successfully, call
        <function>snd_rawmidi_transmit_ack</function> to remove the
        data from the substream buffer:
          <informalexample>
            <programlisting>
<![CDATA[
  unsigned char data;
  while (snd_rawmidi_transmit_peek(substream, &data, 1) == 1) {
          if (snd_mychip_try_to_transmit(data))
                  snd_rawmidi_transmit_ack(substream, 1);
          else
                  break; /* hardware FIFO full */
  }
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
        If you know beforehand that the hardware will accept data, you
        can use the <function>snd_rawmidi_transmit</function> function
        which reads some data and removes them from the buffer at once:
          <informalexample>
            <programlisting>
<![CDATA[
  while (snd_mychip_transmit_possible()) {
          unsigned char data;
          if (snd_rawmidi_transmit(substream, &data, 1) != 1)
                  break; /* no more data */
          snd_mychip_transmit(data);
  }
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
        If you know beforehand how many bytes you can accept, you can
```

```
        use a buffer size greater than one with the
        <function>snd_rawmidi_transmit*</function> functions.
        </para>

        <para>
        The <function>trigger</function> callback must not sleep.  If
        the hardware FIFO is full before the substream buffer has been
        emptied, you have to continue transmitting data later, either
        in an interrupt handler, or with a timer if the hardware
        doesn't have a MIDI transmit interrupt.
        </para>

        <para>
        The <function>trigger</function> callback is called with a
        zero <parameter>up</parameter> parameter when the transmission
        of data should be aborted.
        </para>
      </section>

      <section id="rawmidi-interface-op-trigger-in">
      <title><function>trigger</function> callback for input
      substreams</title>

        <informalexample>
          <programlisting>
<![CDATA[
  static void snd_xxx_input_trigger(struct snd_rawmidi_substream *substream, int
up);
]]>
          </programlisting>
        </informalexample>

        <para>
        This is called with a nonzero <parameter>up</parameter>
        parameter to enable receiving data, or with a zero
        <parameter>up</parameter> parameter do disable receiving data.
        </para>

        <para>
        The <function>trigger</function> callback must not sleep; the
        actual reading of data from the device is usually done in an
        interrupt handler.
        </para>

        <para>
        When data reception is enabled, your interrupt handler should
        call <function>snd_rawmidi_receive</function> for all received
        data:
          <informalexample>
            <programlisting>
<![CDATA[
  void snd_mychip_midi_interrupt(...)
  {
          while (mychip_midi_available()) {
                  unsigned char data;
                  data = mychip_midi_read();
```

```
                snd_rawmidi_receive(substream, &data, 1);
        }
  }
]]>
          </programlisting>
        </informalexample>
      </para>
    </section>

    <section id="rawmidi-interface-op-drain">
    <title><function>drain</function> callback</title>

      <informalexample>
        <programlisting>
<![CDATA[
  static void snd_xxx_drain(struct snd_rawmidi_substream *substream);
]]>
        </programlisting>
      </informalexample>

      <para>
      This is only used with output substreams.  This function should wait
      until all data read from the substream buffer have been transmitted.
      This ensures that the device can be closed and the driver unloaded
      without losing data.
      </para>

      <para>
      This callback is optional. If you do not set
      <structfield>drain</structfield> in the struct snd_rawmidi_ops
      structure, ALSA will simply wait for 50 milliseconds
      instead.
      </para>
    </section>
  </section>

  </chapter>


<!-- ****************************************************** -->
<!-- Miscellaneous Devices  -->
<!-- ****************************************************** -->
  <chapter id="misc-devices">
    <title>Miscellaneous Devices</title>

    <section id="misc-devices-opl3">
      <title>FM OPL3</title>
      <para>
        The FM OPL3 is still used in many chips (mainly for backward
      compatibility). ALSA has a nice OPL3 FM control layer, too. The
      OPL3 API is defined in
      <filename>&lt;sound/opl3.h&gt;</filename>.
      </para>

      <para>
        FM registers can be directly accessed through the direct-FM API,
```

```
      defined in <filename>&lt;sound/asound_fm.h&gt;</filename>. In
      ALSA native mode, FM registers are accessed through
      the Hardware-Dependant Device direct-FM extension API, whereas in
      OSS compatible mode, FM registers can be accessed with the OSS
      direct-FM compatible API in <filename>/dev/dmfmX</filename> device.
      </para>

      <para>
        To create the OPL3 component, you have two functions to
        call. The first one is a constructor for the <type>opl3_t</type>
        instance.

        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_opl3 *opl3;
  snd_opl3_create(card, lport, rport, OPL3_HW_OPL3_XXX,
                  integrated, &opl3);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The first argument is the card pointer, the second one is the
      left port address, and the third is the right port address. In
      most cases, the right port is placed at the left port + 2.
      </para>

      <para>
        The fourth argument is the hardware type.
      </para>

      <para>
        When the left and right ports have been already allocated by
      the card driver, pass non-zero to the fifth argument
      (<parameter>integrated</parameter>). Otherwise, the opl3 module will
      allocate the specified ports by itself.
      </para>

      <para>
        When the accessing the hardware requires special method
        instead of the standard I/O access, you can create opl3 instance
        separately with <function>snd_opl3_new()</function>.

        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_opl3 *opl3;
  snd_opl3_new(card, OPL3_HW_OPL3_XXX, &opl3);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
```

```
          Then set <structfield>command</structfield>,
          <structfield>private_data</structfield> and
          <structfield>private_free</structfield> for the private
          access function, the private data and the destructor.
          The l_port and r_port are not necessarily set.  Only the
          command must be set properly.  You can retrieve the data
          from the opl3-&gt;private_data field.
        </para>

        <para>
          After creating the opl3 instance via
<function>snd_opl3_new()</function>,
          call <function>snd_opl3_init()</function> to initialize the chip to the
          proper state. Note that <function>snd_opl3_create()</function> always
          calls it internally.
        </para>

        <para>
          If the opl3 instance is created successfully, then create a
          hwdep device for this opl3.

          <informalexample>
            <programlisting>
<![CDATA[
  struct snd_hwdep *opl3hwdep;
  snd_opl3_hwdep_new(opl3, 0, 1, &opl3hwdep);
]]>
            </programlisting>
          </informalexample>
        </para>

        <para>
          The first argument is the <type>opl3_t</type> instance you
        created, and the second is the index number, usually 0.
        </para>

        <para>
          The third argument is the index-offset for the sequencer
        client assigned to the OPL3 port. When there is an MPU401-UART,
        give 1 for here (UART always takes 0).
        </para>
      </section>

      <section id="misc-devices-hardware-dependent">
        <title>Hardware-Dependent Devices</title>
        <para>
          Some chips need user-space access for special
        controls or for loading the micro code. In such a case, you can
        create a hwdep (hardware-dependent) device. The hwdep API is
        defined in <filename>&lt;sound/hwdep.h&gt;</filename>. You can
        find examples in opl3 driver or
        <filename>isa/sb/sb16_csp.c</filename>.
        </para>

        <para>
          The creation of the <type>hwdep</type> instance is done via
```

```
        <function>snd_hwdep_new()</function>.

        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_hwdep *hw;
  snd_hwdep_new(card, "My HWDEP", 0, &hw);
]]>
          </programlisting>
        </informalexample>

        where the third argument is the index number.
      </para>

      <para>
        You can then pass any pointer value to the
        <parameter>private_data</parameter>.
        If you assign a private data, you should define the
        destructor, too. The destructor function is set in
        the <structfield>private_free</structfield> field.

        <informalexample>
          <programlisting>
<![CDATA[
  struct mydata *p = kmalloc(sizeof(*p), GFP_KERNEL);
  hw->private_data = p;
  hw->private_free = mydata_free;
]]>
          </programlisting>
        </informalexample>

        and the implementation of the destructor would be:

        <informalexample>
          <programlisting>
<![CDATA[
  static void mydata_free(struct snd_hwdep *hw)
  {
          struct mydata *p = hw->private_data;
          kfree(p);
  }
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The arbitrary file operations can be defined for this
        instance. The file operators are defined in
        the <parameter>ops</parameter> table. For example, assume that
        this chip needs an ioctl.

        <informalexample>
          <programlisting>
<![CDATA[
  hw->ops.open = mydata_open;
```

```
  hw->ops.ioctl = mydata_ioctl;
  hw->ops.release = mydata_release;
]]>
          </programlisting>
        </informalexample>

        And implement the callback functions as you like.
      </para>
    </section>

    <section id="misc-devices-IEC958">
      <title>IEC958 (S/PDIF)</title>
      <para>
        Usually the controls for IEC958 devices are implemented via
      the control interface. There is a macro to compose a name string for
      IEC958 controls, <function>SNDRV_CTL_NAME_IEC958()</function>
      defined in <filename>&lt;include/asound.h&gt;</filename>.
      </para>

      <para>
        There are some standard controls for IEC958 status bits. These
      controls use the type <type>SNDRV_CTL_ELEM_TYPE_IEC958</type>,
      and the size of element is fixed as 4 bytes array
      (value.iec958.status[x]). For the <structfield>info</structfield>
      callback, you don't specify
      the value field for this type (the count field must be set,
      though).
      </para>

      <para>
        <quote>IEC958 Playback Con Mask</quote> is used to return the
      bit-mask for the IEC958 status bits of consumer mode. Similarly,
      <quote>IEC958 Playback Pro Mask</quote> returns the bitmask for
      professional mode. They are read-only controls, and are defined
      as MIXER controls (iface =
      <constant>SNDRV_CTL_ELEM_IFACE_MIXER</constant>).
      </para>

      <para>
        Meanwhile, <quote>IEC958 Playback Default</quote> control is
      defined for getting and setting the current default IEC958
      bits. Note that this one is usually defined as a PCM control
      (iface = <constant>SNDRV_CTL_ELEM_IFACE_PCM</constant>),
      although in some places it's defined as a MIXER control.
      </para>

      <para>
        In addition, you can define the control switches to
      enable/disable or to set the raw bit mode. The implementation
      will depend on the chip, but the control should be named as
      <quote>IEC958 xxx</quote>, preferably using
      the <function>SNDRV_CTL_NAME_IEC958()</function> macro.
      </para>

      <para>
        You can find several cases, for example,
```

```
      <filename>pci/emu10k1</filename>,
      <filename>pci/ice1712</filename>, or
      <filename>pci/cmipci.c</filename>.
      </para>
    </section>

  </chapter>


<!-- ****************************************************** -->
<!-- Buffer and Memory Management  -->
<!-- ****************************************************** -->
  <chapter id="buffer-and-memory">
    <title>Buffer and Memory Management</title>

    <section id="buffer-and-memory-buffer-types">
      <title>Buffer Types</title>
      <para>
        ALSA provides several different buffer allocation functions
      depending on the bus and the architecture. All these have a
      consistent API. The allocation of physically-contiguous pages is
      done via
      <function>snd_malloc_xxx_pages()</function> function, where xxx
      is the bus type.
      </para>

      <para>
        The allocation of pages with fallback is
      <function>snd_malloc_xxx_pages_fallback()</function>. This
      function tries to allocate the specified pages but if the pages
      are not available, it tries to reduce the page sizes until
      enough space is found.
      </para>

      <para>
      The release the pages, call
      <function>snd_free_xxx_pages()</function> function.
      </para>

      <para>
      Usually, ALSA drivers try to allocate and reserve
       a large contiguous physical space
       at the time the module is loaded for the later use.
       This is called <quote>pre-allocation</quote>.
       As already written, you can call the following function at
       pcm instance construction time (in the case of PCI bus).

        <informalexample>
          <programlisting>
<![CDATA[
  snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
                                        snd_dma_pci_data(pci), size, max);
]]>
          </programlisting>
        </informalexample>
```

```
          where <parameter>size</parameter> is the byte size to be
        pre-allocated and the <parameter>max</parameter> is the maximum
        size to be changed via the <filename>prealloc</filename> proc file.
        The allocator will try to get an area as large as possible
        within the given size.
        </para>

        <para>
        The second argument (type) and the third argument (device pointer)
        are dependent on the bus.
        In the case of the ISA bus, pass <function>snd_dma_isa_data()</function>
        as the third argument with <constant>SNDRV_DMA_TYPE_DEV</constant> type.
        For the continuous buffer unrelated to the bus can be pre-allocated
        with <constant>SNDRV_DMA_TYPE_CONTINUOUS</constant> type and the
        <function>snd_dma_continuous_data(GFP_KERNEL)</function> device pointer,
        where <constant>GFP_KERNEL</constant> is the kernel allocation flag to
        use.
        For the PCI scatter-gather buffers, use
        <constant>SNDRV_DMA_TYPE_DEV_SG</constant> with
        <function>snd_dma_pci_data(pci)</function>
        (see the
            <link
linkend="buffer-and-memory-non-contiguous"><citetitle>Non-Contiguous Buffers
            </citetitle></link> section).
        </para>

        <para>
          Once the buffer is pre-allocated, you can use the
          allocator in the <structfield>hw_params</structfield> callback:

          <informalexample>
            <programlisting>
<![CDATA[
  snd_pcm_lib_malloc_pages(substream, size);
]]>
            </programlisting>
          </informalexample>

          Note that you have to pre-allocate to use this function.
        </para>
      </section>

      <section id="buffer-and-memory-external-hardware">
        <title>External Hardware Buffers</title>
        <para>
          Some chips have their own hardware buffers and the DMA
        transfer from the host memory is not available. In such a case,
        you need to either 1) copy/set the audio data directly to the
        external hardware buffer, or 2) make an intermediate buffer and
        copy/set the data from it to the external hardware buffer in
        interrupts (or in tasklets, preferably).
        </para>

        <para>
          The first case works fine if the external hardware buffer is large
        enough.  This method doesn't need any extra buffers and thus is
```

more effective. You need to define the
<structfield>copy</structfield> and
<structfield>silence</structfield> callbacks for
the data transfer. However, there is a drawback: it cannot
be mmapped. The examples are GUS's GF1 PCM or emu8000's
wavetable PCM.
</para>

<para>
The second case allows for mmap on the buffer, although you have
to handle an interrupt or a tasklet to transfer the data
from the intermediate buffer to the hardware buffer. You can find an
example in the vxpocket driver.
</para>

<para>
Another case is when the chip uses a PCI memory-map
region for the buffer instead of the host memory. In this case,
mmap is available only on certain architectures like the Intel one.
In non-mmap mode, the data cannot be transferred as in the normal
way. Thus you need to define the <structfield>copy</structfield> and
<structfield>silence</structfield> callbacks as well,
as in the cases above. The examples are found in
<filename>rme32.c</filename> and <filename>rme96.c</filename>.
</para>

<para>
The implementation of the <structfield>copy</structfield> and
<structfield>silence</structfield> callbacks depends upon
whether the hardware supports interleaved or non-interleaved
samples. The <structfield>copy</structfield> callback is
defined like below, a bit
differently depending whether the direction is playback or
capture:

<informalexample>
<programlisting>

```
<![CDATA[
  static int playback_copy(struct snd_pcm_substream *substream, int channel,
               snd_pcm_uframes_t pos, void *src, snd_pcm_uframes_t count);
  static int capture_copy(struct snd_pcm_substream *substream, int channel,
               snd_pcm_uframes_t pos, void *dst, snd_pcm_uframes_t count);
]]>
```

</programlisting>
</informalexample>
</para>

<para>
In the case of interleaved samples, the second argument
(<parameter>channel</parameter>) is not used. The third argument
(<parameter>pos</parameter>) points the
current position offset in frames.
</para>

<para>
The meaning of the fourth argument is different between

```
      playback and capture. For playback, it holds the source data
      pointer, and for capture, it's the destination data pointer.
      </para>

      <para>
        The last argument is the number of frames to be copied.
      </para>

      <para>
        What you have to do in this callback is again different
        between playback and capture directions. In the
        playback case, you copy the given amount of data
        (<parameter>count</parameter>) at the specified pointer
        (<parameter>src</parameter>) to the specified offset
        (<parameter>pos</parameter>) on the hardware buffer. When
        coded like memcpy-like way, the copy would be like:

        <informalexample>
          <programlisting>
<![CDATA[
  my_memcpy(my_buffer + frames_to_bytes(runtime, pos), src,
            frames_to_bytes(runtime, count));
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        For the capture direction, you copy the given amount of
        data (<parameter>count</parameter>) at the specified offset
        (<parameter>pos</parameter>) on the hardware buffer to the
        specified pointer (<parameter>dst</parameter>).

        <informalexample>
          <programlisting>
<![CDATA[
  my_memcpy(dst, my_buffer + frames_to_bytes(runtime, pos),
            frames_to_bytes(runtime, count));
]]>
          </programlisting>
        </informalexample>

        Note that both the position and the amount of data are given
      in frames.
      </para>

      <para>
        In the case of non-interleaved samples, the implementation
      will be a bit more complicated.
      </para>

      <para>
        You need to check the channel argument, and if it's -1, copy
      the whole channels. Otherwise, you have to copy only the
      specified channel. Please check
      <filename>isa/gus/gus_pcm.c</filename> as an example.
```

```
      </para>

      <para>
        The <structfield>silence</structfield> callback is also
        implemented in a similar way.

        <informalexample>
          <programlisting>
<![CDATA[
  static int silence(struct snd_pcm_substream *substream, int channel,
                     snd_pcm_uframes_t pos, snd_pcm_uframes_t count);
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        The meanings of arguments are the same as in the
      <structfield>copy</structfield>
      callback, although there is no <parameter>src/dst</parameter>
      argument. In the case of interleaved samples, the channel
      argument has no meaning, as well as on
      <structfield>copy</structfield> callback.
      </para>

      <para>
        The role of <structfield>silence</structfield> callback is to
        set the given amount
        (<parameter>count</parameter>) of silence data at the
        specified offset (<parameter>pos</parameter>) on the hardware
        buffer. Suppose that the data format is signed (that is, the
        silent-data is 0), and the implementation using a memset-like
        function would be like:

        <informalexample>
          <programlisting>
<![CDATA[
  my_memcpy(my_buffer + frames_to_bytes(runtime, pos), 0,
            frames_to_bytes(runtime, count));
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        In the case of non-interleaved samples, again, the
      implementation becomes a bit more complicated. See, for example,
      <filename>isa/gus/gus_pcm.c</filename>.
      </para>
    </section>

    <section id="buffer-and-memory-non-contiguous">
      <title>Non-Contiguous Buffers</title>
      <para>
        If your hardware supports the page table as in emu10k1 or the
      buffer descriptors as in via82xx, you can use the scatter-gather
```

```
      (SG) DMA. ALSA provides an interface for handling SG-buffers.
      The API is provided in <filename>&lt;sound/pcm.h&gt;</filename>.
      </para>

      <para>
        For creating the SG-buffer handler, call
        <function>snd_pcm_lib_preallocate_pages()</function> or
        <function>snd_pcm_lib_preallocate_pages_for_all()</function>
        with <constant>SNDRV_DMA_TYPE_DEV_SG</constant>
        in the PCM constructor like other PCI pre-allocator.
        You need to pass <function>snd_dma_pci_data(pci)</function>,
        where pci is the struct <structname>pci_dev</structname> pointer
        of the chip as well.
        The <type>struct snd_sg_buf</type> instance is created as
        substream-&gt;dma_private. You can cast
        the pointer like:

        <informalexample>
          <programlisting>
<![CDATA[
  struct snd_sg_buf *sgbuf = (struct snd_sg_buf *)substream->dma_private;
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        Then call <function>snd_pcm_lib_malloc_pages()</function>
      in the <structfield>hw_params</structfield> callback
      as well as in the case of normal PCI buffer.
      The SG-buffer handler will allocate the non-contiguous kernel
      pages of the given size and map them onto the virtually contiguous
      memory.  The virtual pointer is addressed in runtime-&gt;dma_area.
      The physical address (runtime-&gt;dma_addr) is set to zero,
      because the buffer is physically non-contiguous.
      The physical address table is set up in sgbuf-&gt;table.
      You can get the physical address at a certain offset via
      <function>snd_pcm_sgbuf_get_addr()</function>.
      </para>

      <para>
        When a SG-handler is used, you need to set
      <function>snd_pcm_sgbuf_ops_page</function> as
      the <structfield>page</structfield> callback.
      (See <link linkend="pcm-interface-operators-page-callback">
      <citetitle>page callback section</citetitle></link>.)
      </para>

      <para>
        To release the data, call
      <function>snd_pcm_lib_free_pages()</function> in the
      <structfield>hw_free</structfield> callback as usual.
      </para>
    </section>

    <section id="buffer-and-memory-vmalloced">
```

```
      <title>Vmalloc'ed Buffers</title>
      <para>
        It's possible to use a buffer allocated via
      <function>vmalloc</function>, for example, for an intermediate
      buffer. Since the allocated pages are not contiguous, you need
      to set the <structfield>page</structfield> callback to obtain
      the physical address at every offset.
      </para>

      <para>
        The implementation of <structfield>page</structfield> callback
        would be like this:

        <informalexample>
          <programlisting>
<![CDATA[
  #include <linux/vmalloc.h>

  /* get the physical page pointer on the given offset */
  static struct page *mychip_page(struct snd_pcm_substream *substream,
                                  unsigned long offset)
  {
          void *pageptr = substream->runtime->dma_area + offset;
          return vmalloc_to_page(pageptr);
  }
]]>
          </programlisting>
        </informalexample>
      </para>
    </section>

  </chapter>


<!-- ****************************************************** -->
<!-- Proc Interface  -->
<!-- ****************************************************** -->
  <chapter id="proc-interface">
    <title>Proc Interface</title>
    <para>
      ALSA provides an easy interface for procfs. The proc files are
      very useful for debugging. I recommend you set up proc files if
      you write a driver and want to get a running status or register
      dumps. The API is found in
      <filename>&lt;sound/info.h&gt;</filename>.
    </para>

    <para>
      To create a proc file, call
      <function>snd_card_proc_new()</function>.

      <informalexample>
        <programlisting>
<![CDATA[
  struct snd_info_entry *entry;
  int err = snd_card_proc_new(card, "my-file", &entry);
```

```
]]>
        </programlisting>
      </informalexample>

      where the second argument specifies the name of the proc file to be
    created. The above example will create a file
    <filename>my-file</filename> under the card directory,
    e.g. <filename>/proc/asound/card0/my-file</filename>.
    </para>

    <para>
    Like other components, the proc entry created via
    <function>snd_card_proc_new()</function> will be registered and
    released automatically in the card registration and release
    functions.
    </para>

    <para>
      When the creation is successful, the function stores a new
    instance in the pointer given in the third argument.
    It is initialized as a text proc file for read only.  To use
    this proc file as a read-only text file as it is, set the read
    callback with a private data via
     <function>snd_info_set_text_ops()</function>.

      <informalexample>
        <programlisting>
<![CDATA[
  snd_info_set_text_ops(entry, chip, my_proc_read);
]]>
        </programlisting>
      </informalexample>

    where the second argument (<parameter>chip</parameter>) is the
    private data to be used in the callbacks. The third parameter
    specifies the read buffer size and the fourth
    (<parameter>my_proc_read</parameter>) is the callback function, which
    is defined like

      <informalexample>
        <programlisting>
<![CDATA[
  static void my_proc_read(struct snd_info_entry *entry,
                           struct snd_info_buffer *buffer);
]]>
        </programlisting>
      </informalexample>

    </para>

    <para>
    In the read callback, use <function>snd_iprintf()</function> for
    output strings, which works just like normal
    <function>printf()</function>.  For example,

      <informalexample>
```

```
        <programlisting>
<![CDATA[
  static void my_proc_read(struct snd_info_entry *entry,
                           struct snd_info_buffer *buffer)
  {
          struct my_chip *chip = entry->private_data;

          snd_iprintf(buffer, "This is my chip!\n");
          snd_iprintf(buffer, "Port = %ld\n", chip->port);
  }
]]>
        </programlisting>
      </informalexample>
    </para>

    <para>
    The file permissions can be changed afterwards.  As default, it's
    set as read only for all users.  If you want to add write
    permission for the user (root as default), do as follows:

      <informalexample>
        <programlisting>
<![CDATA[
 entry->mode = S_IFREG | S_IRUGO | S_IWUSR;
]]>
        </programlisting>
      </informalexample>

    and set the write buffer size and the callback

      <informalexample>
        <programlisting>
<![CDATA[
  entry->c.text.write = my_proc_write;
]]>
        </programlisting>
      </informalexample>
    </para>

    <para>
      For the write callback, you can use
    <function>snd_info_get_line()</function> to get a text line, and
    <function>snd_info_get_str()</function> to retrieve a string from
    the line. Some examples are found in
    <filename>core/oss/mixer_oss.c</filename>, core/oss/and
    <filename>pcm_oss.c</filename>.
    </para>

    <para>
      For a raw-data proc-file, set the attributes as follows:

      <informalexample>
        <programlisting>
<![CDATA[
  static struct snd_info_entry_ops my_file_io_ops = {
          .read = my_file_io_read,
```

```
  };

  entry->content = SNDRV_INFO_CONTENT_DATA;
  entry->private_data = chip;
  entry->c.ops = &my_file_io_ops;
  entry->size = 4096;
  entry->mode = S_IFREG | S_IRUGO;
]]>
        </programlisting>
      </informalexample>

      For the raw data, <structfield>size</structfield> field must be
      set properly.  This specifies the maximum size of the proc file access.
    </para>

    <para>
      The read/write callbacks of raw mode are more direct than the text mode.
      You need to use a low-level I/O functions such as
      <function>copy_from/to_user()</function> to transfer the
      data.

      <informalexample>
        <programlisting>
<![CDATA[
  static ssize_t my_file_io_read(struct snd_info_entry *entry,
                              void *file_private_data,
                              struct file *file,
                              char *buf,
                              size_t count,
                              loff_t pos)
  {
          if (copy_to_user(buf, local_data + pos, count))
                  return -EFAULT;
          return count;
  }
]]>
        </programlisting>
      </informalexample>

      If the size of the info entry has been set up properly,
      <structfield>count</structfield> and <structfield>pos</structfield> are
      guaranteed to fit within 0 and the given size.
      You don't have to check the range in the callbacks unless any
      other condition is required.

    </para>

  </chapter>


<!-- ****************************************************** -->
<!-- Power Management  -->
<!-- ****************************************************** -->
  <chapter id="power-management">
    <title>Power Management</title>
    <para>
```

If the chip is supposed to work with suspend/resume functions, you need to add power-management code to the driver. The additional code for power-management should be <function>ifdef</function>'ed with <constant>CONFIG_PM</constant>.
</para>

<para>
If the driver <emphasis>fully</emphasis> supports suspend/resume that is, the device can be properly resumed to its state when suspend was called, you can set the <constant>SNDRV_PCM_INFO_RESUME</constant> flag in the pcm info field. Usually, this is possible when the registers of the chip can be safely saved and restored to RAM. If this is set, the trigger callback is called with <constant>SNDRV_PCM_TRIGGER_RESUME</constant> after the resume callback completes.
</para>

<para>
Even if the driver doesn't support PM fully but partial suspend/resume is still possible, it's still worthy to implement suspend/resume callbacks. In such a case, applications would reset the status by calling <function>snd_pcm_prepare()</function> and restart the stream appropriately. Hence, you can define suspend/resume callbacks below but don't set <constant>SNDRV_PCM_INFO_RESUME</constant> info flag to the PCM.
</para>

<para>
Note that the trigger with SUSPEND can always be called when <function>snd_pcm_suspend_all</function> is called, regardless of the <constant>SNDRV_PCM_INFO_RESUME</constant> flag. The <constant>RESUME</constant> flag affects only the behavior of <function>snd_pcm_resume()</function>. (Thus, in theory, <constant>SNDRV_PCM_TRIGGER_RESUME</constant> isn't needed to be handled in the trigger callback when no <constant>SNDRV_PCM_INFO_RESUME</constant> flag is set. But, it's better to keep it for compatibility reasons.)
</para>
<para>
In the earlier version of ALSA drivers, a common power-management layer was provided, but it has been removed. The driver needs to define the suspend/resume hooks according to the bus the device is connected to. In the case of PCI drivers, the callbacks look like below:

<informalexample>
<programlisting>

```
<![CDATA[
  #ifdef CONFIG_PM
  static int snd_my_suspend(struct pci_dev *pci, pm_message_t state)
  {
          .... /* do things for suspend */
```

```
          return 0;
  }
  static int snd_my_resume(struct pci_dev *pci)
  {
          .... /* do things for suspend */
          return 0;
  }
  #endif
]]>
        </programlisting>
      </informalexample>
    </para>

    <para>
      The scheme of the real suspend job is as follows.

      <orderedlist>
        <listitem><para>Retrieve the card and the chip data.</para></listitem>
        <listitem><para>Call <function>snd_power_change_state()</function> with
          <constant>SNDRV_CTL_POWER_D3hot</constant> to change the
          power status.</para></listitem>
        <listitem><para>Call <function>snd_pcm_suspend_all()</function> to
suspend the running PCM streams.</para></listitem>
        <listitem><para>If AC97 codecs are used, call
        <function>snd_ac97_suspend()</function> for each
codec.</para></listitem>
        <listitem><para>Save the register values if necessary.</para></listitem>
        <listitem><para>Stop the hardware if necessary.</para></listitem>
        <listitem><para>Disable the PCI device by calling
          <function>pci_disable_device()</function>.  Then, call
          <function>pci_save_state()</function> at last.</para></listitem>
      </orderedlist>
    </para>

    <para>
      A typical code would be like:

      <informalexample>
        <programlisting>
<![CDATA[
  static int mychip_suspend(struct pci_dev *pci, pm_message_t state)
  {
          /* (1) */
          struct snd_card *card = pci_get_drvdata(pci);
          struct mychip *chip = card->private_data;
          /* (2) */
          snd_power_change_state(card, SNDRV_CTL_POWER_D3hot);
          /* (3) */
          snd_pcm_suspend_all(chip->pcm);
          /* (4) */
          snd_ac97_suspend(chip->ac97);
          /* (5) */
          snd_mychip_save_registers(chip);
          /* (6) */
          snd_mychip_stop_hardware(chip);
          /* (7) */
```

```
          pci_disable_device(pci);
          pci_save_state(pci);
          return 0;
  }
]]>
        </programlisting>
      </informalexample>
    </para>

    <para>
    The scheme of the real resume job is as follows.

    <orderedlist>
    <listitem><para>Retrieve the card and the chip data.</para></listitem>
    <listitem><para>Set up PCI. First, call
<function>pci_restore_state()</function>.
        Then enable the pci device again by calling
<function>pci_enable_device()</function>.
        Call <function>pci_set_master()</function> if necessary,
too.</para></listitem>
    <listitem><para>Re-initialize the chip.</para></listitem>
    <listitem><para>Restore the saved registers if necessary.</para></listitem>
    <listitem><para>Resume the mixer, e.g. calling
    <function>snd_ac97_resume()</function>.</para></listitem>
    <listitem><para>Restart the hardware (if any).</para></listitem>
    <listitem><para>Call <function>snd_power_change_state()</function> with
        <constant>SNDRV_CTL_POWER_D0</constant> to notify the
processes.</para></listitem>
    </orderedlist>
    </para>

    <para>
    A typical code would be like:

      <informalexample>
        <programlisting>
<![CDATA[
  static int mychip_resume(struct pci_dev *pci)
  {
          /* (1) */
          struct snd_card *card = pci_get_drvdata(pci);
          struct mychip *chip = card->private_data;
          /* (2) */
          pci_restore_state(pci);
          pci_enable_device(pci);
          pci_set_master(pci);
          /* (3) */
          snd_mychip_reinit_chip(chip);
          /* (4) */
          snd_mychip_restore_registers(chip);
          /* (5) */
          snd_ac97_resume(chip->ac97);
          /* (6) */
          snd_mychip_restart_chip(chip);
          /* (7) */
          snd_power_change_state(card, SNDRV_CTL_POWER_D0);
```

```
          return 0;
  }
]]>
        </programlisting>
      </informalexample>
    </para>

    <para>
        As shown in the above, it's better to save registers after
        suspending the PCM operations via
        <function>snd_pcm_suspend_all()</function> or
        <function>snd_pcm_suspend()</function>.  It means that the PCM
        streams are already stoppped when the register snapshot is
        taken.  But, remember that you don't have to restart the PCM
        stream in the resume callback. It'll be restarted via
        trigger call with <constant>SNDRV_PCM_TRIGGER_RESUME</constant>
        when necessary.
    </para>

    <para>
      OK, we have all callbacks now. Let's set them up. In the
      initialization of the card, make sure that you can get the chip
      data from the card instance, typically via
      <structfield>private_data</structfield> field, in case you
      created the chip data individually.

      <informalexample>
        <programlisting>
<![CDATA[
  static int __devinit snd_mychip_probe(struct pci_dev *pci,
                               const struct pci_device_id *pci_id)
  {
          ....
          struct snd_card *card;
          struct mychip *chip;
          int err;
          ....
          err = snd_card_create(index[dev], id[dev], THIS_MODULE, 0, &card);
          ....
          chip = kzalloc(sizeof(*chip), GFP_KERNEL);
          ....
          card->private_data = chip;
          ....
  }
]]>
        </programlisting>
      </informalexample>

        When you created the chip data with
        <function>snd_card_create()</function>, it's anyway accessible
        via <structfield>private_data</structfield> field.

      <informalexample>
        <programlisting>
<![CDATA[
  static int __devinit snd_mychip_probe(struct pci_dev *pci,
```

```
                                const struct pci_device_id *pci_id)
  {
          ....
          struct snd_card *card;
          struct mychip *chip;
          int err;
          ....
          err = snd_card_create(index[dev], id[dev], THIS_MODULE,
                                sizeof(struct mychip), &card);
          ....
          chip = card->private_data;
          ....
  }
]]>
        </programlisting>
      </informalexample>

    </para>

    <para>
      If you need a space to save the registers, allocate the
        buffer for it here, too, since it would be fatal
    if you cannot allocate a memory in the suspend phase.
    The allocated buffer should be released in the corresponding
    destructor.
    </para>

    <para>
      And next, set suspend/resume callbacks to the pci_driver.

      <informalexample>
        <programlisting>
<![CDATA[
  static struct pci_driver driver = {
          .name = "My Chip",
          .id_table = snd_my_ids,
          .probe = snd_my_probe,
          .remove = __devexit_p(snd_my_remove),
  #ifdef CONFIG_PM
          .suspend = snd_my_suspend,
          .resume = snd_my_resume,
  #endif
  };
]]>
        </programlisting>
      </informalexample>
    </para>

  </chapter>


<!-- ****************************************************** -->
<!-- Module Parameters  -->
<!-- ****************************************************** -->
  <chapter id="module-parameters">
    <title>Module Parameters</title>
```

```
    <para>
      There are standard module options for ALSA. At least, each
      module should have the <parameter>index</parameter>,
      <parameter>id</parameter> and <parameter>enable</parameter>
      options.
    </para>

    <para>
      If the module supports multiple cards (usually up to
      8 = <constant>SNDRV_CARDS</constant> cards), they should be
      arrays. The default initial values are defined already as
      constants for easier programming:

      <informalexample>
        <programlisting>
<![CDATA[
  static int index[SNDRV_CARDS] = SNDRV_DEFAULT_IDX;
  static char *id[SNDRV_CARDS] = SNDRV_DEFAULT_STR;
  static int enable[SNDRV_CARDS] = SNDRV_DEFAULT_ENABLE_PNP;
]]>
        </programlisting>
      </informalexample>
    </para>

    <para>
      If the module supports only a single card, they could be single
    variables, instead.  <parameter>enable</parameter> option is not
    always necessary in this case, but it would be better to have a
    dummy option for compatibility.
    </para>

    <para>
      The module parameters must be declared with the standard
    <function>module_param()()</function>,
    <function>module_param_array()()</function> and
    <function>MODULE_PARM_DESC()</function> macros.
    </para>

    <para>
      The typical coding would be like below:

      <informalexample>
        <programlisting>
<![CDATA[
  #define CARD_NAME "My Chip"

  module_param_array(index, int, NULL, 0444);
  MODULE_PARM_DESC(index, "Index value for " CARD_NAME " soundcard.");
  module_param_array(id, charp, NULL, 0444);
  MODULE_PARM_DESC(id, "ID string for " CARD_NAME " soundcard.");
  module_param_array(enable, bool, NULL, 0444);
  MODULE_PARM_DESC(enable, "Enable " CARD_NAME " soundcard.");
]]>
        </programlisting>
      </informalexample>
    </para>
```

```
    <para>
      Also, don't forget to define the module description, classes,
      license and devices. Especially, the recent modprobe requires to
      define the module license as GPL, etc., otherwise the system is
      shown as <quote>tainted</quote>.

      <informalexample>
        <programlisting>
<![CDATA[
  MODULE_DESCRIPTION("My Chip");
  MODULE_LICENSE("GPL");
  MODULE_SUPPORTED_DEVICE("{{Vendor,My Chip Name}}");
]]>
        </programlisting>
      </informalexample>
    </para>

  </chapter>


<!-- ****************************************************** -->
<!-- How To Put Your Driver  -->
<!-- ****************************************************** -->
  <chapter id="how-to-put-your-driver">
    <title>How To Put Your Driver Into ALSA Tree</title>
        <section>
        <title>General</title>
        <para>
        So far, you've learned how to write the driver codes.
        And you might have a question now: how to put my own
        driver into the ALSA driver tree?
        Here (finally :) the standard procedure is described briefly.
        </para>

        <para>
        Suppose that you create a new PCI driver for the card
        <quote>xyz</quote>.  The card module name would be
        snd-xyz.  The new driver is usually put into the alsa-driver
        tree, <filename>alsa-driver/pci</filename> directory in
        the case of PCI cards.
        Then the driver is evaluated, audited and tested
        by developers and users.  After a certain time, the driver
        will go to the alsa-kernel tree (to the corresponding directory,
        such as <filename>alsa-kernel/pci</filename>) and eventually
        will be integrated into the Linux 2.6 tree (the directory would be
        <filename>linux/sound/pci</filename>).
        </para>

        <para>
        In the following sections, the driver code is supposed
        to be put into alsa-driver tree. The two cases are covered:
        a driver consisting of a single source file and one consisting
        of several source files.
        </para>
        </section>
```

```
        <section>
        <title>Driver with A Single Source File</title>
        <para>
        <orderedlist>
        <listitem>
        <para>
        Modify alsa-driver/pci/Makefile
        </para>

        <para>
        Suppose you have a file xyz.c.  Add the following
        two lines
      <informalexample>
        <programlisting>
<![CDATA[
  snd-xyz-objs := xyz.o
  obj-$(CONFIG_SND_XYZ) += snd-xyz.o
]]>
        </programlisting>
      </informalexample>
        </para>
        </listitem>

        <listitem>
        <para>
        Create the Kconfig entry
        </para>

        <para>
        Add the new entry of Kconfig for your xyz driver.
      <informalexample>
        <programlisting>
<![CDATA[
  config SND_XYZ
          tristate "Foobar XYZ"
          depends on SND
          select SND_PCM
          help
            Say Y here to include support for Foobar XYZ soundcard.

            To compile this driver as a module, choose M here: the module
            will be called snd-xyz.
]]>
        </programlisting>
      </informalexample>

        the line, select SND_PCM, specifies that the driver xyz supports
        PCM.  In addition to SND_PCM, the following components are
        supported for select command:
        SND_RAWMIDI, SND_TIMER, SND_HWDEP, SND_MPU401_UART,
        SND_OPL3_LIB, SND_OPL4_LIB, SND_VX_LIB, SND_AC97_CODEC.
        Add the select command for each supported component.
        </para>

        <para>
```

```
        Note that some selections imply the lowlevel selections.
        For example, PCM includes TIMER, MPU401_UART includes RAWMIDI,
        AC97_CODEC includes PCM, and OPL3_LIB includes HWDEP.
        You don't need to give the lowlevel selections again.
        </para>

        <para>
        For the details of Kconfig script, refer to the kbuild
        documentation.
        </para>

        </listitem>

        <listitem>
        <para>
        Run cvscompile script to re-generate the configure script and
        build the whole stuff again.
        </para>
        </listitem>
        </orderedlist>
        </para>
        </section>

        <section>
        <title>Drivers with Several Source Files</title>
        <para>
        Suppose that the driver snd-xyz have several source files.
        They are located in the new subdirectory,
        pci/xyz.

        <orderedlist>
        <listitem>
        <para>
        Add a new directory (<filename>xyz</filename>) in
        <filename>alsa-driver/pci/Makefile</filename> as below

      <informalexample>
        <programlisting>
<![CDATA[
  obj-$(CONFIG_SND) += xyz/
]]>
        </programlisting>
      </informalexample>
        </para>
        </listitem>

        <listitem>
        <para>
        Under the directory <filename>xyz</filename>, create a Makefile

      <example>
        <title>Sample Makefile for a driver xyz</title>
        <programlisting>
<![CDATA[
  ifndef SND_TOPDIR
  SND_TOPDIR=../..
```

```
  endif

  include $(SND_TOPDIR)/toplevel.config
  include $(SND_TOPDIR)/Makefile.conf

  snd-xyz-objs := xyz.o abc.o def.o

  obj-$(CONFIG_SND_XYZ) += snd-xyz.o

  include $(SND_TOPDIR)/Rules.make
]]>
        </programlisting>
      </example>
        </para>
        </listitem>

        <listitem>
        <para>
        Create the Kconfig entry
        </para>

        <para>
        This procedure is as same as in the last section.
        </para>
        </listitem>

        <listitem>
        <para>
        Run cvscompile script to re-generate the configure script and
        build the whole stuff again.
        </para>
        </listitem>
        </orderedlist>
        </para>
        </section>

  </chapter>

<!-- ****************************************************** -->
<!-- Useful Functions  -->
<!-- ****************************************************** -->
  <chapter id="useful-functions">
    <title>Useful Functions</title>

    <section id="useful-functions-snd-printk">
      <title><function>snd_printk()</function> and friends</title>
      <para>
        ALSA provides a verbose version of the
      <function>printk()</function> function. If a kernel config
      <constant>CONFIG_SND_VERBOSE_PRINTK</constant> is set, this
      function prints the given message together with the file name
      and the line of the caller. The <constant>KERN_XXX</constant>
      prefix is processed as
      well as the original <function>printk()</function> does, so it's
      recommended to add this prefix, e.g.
```

```
        <informalexample>
          <programlisting>
<![CDATA[
  snd_printk(KERN_ERR "Oh my, sorry, it's extremely bad!\n");
]]>
          </programlisting>
        </informalexample>
      </para>

      <para>
        There are also <function>printk()</function>'s for
      debugging. <function>snd_printd()</function> can be used for
      general debugging purposes. If
      <constant>CONFIG_SND_DEBUG</constant> is set, this function is
      compiled, and works just like
      <function>snd_printk()</function>. If the ALSA is compiled
      without the debugging flag, it's ignored.
      </para>

      <para>
        <function>snd_printdd()</function> is compiled in only when
      <constant>CONFIG_SND_DEBUG_VERBOSE</constant> is set. Please note
      that <constant>CONFIG_SND_DEBUG_VERBOSE</constant> is not set as default
      even if you configure the alsa-driver with
      <option>--with-debug=full</option> option. You need to give
      explicitly <option>--with-debug=detect</option> option instead.
      </para>
    </section>

    <section id="useful-functions-snd-bug">
      <title><function>snd_BUG()</function></title>
      <para>
        It shows the <computeroutput>BUG?</computeroutput> message and
      stack trace as well as <function>snd_BUG_ON</function> at the point.
      It's useful to show that a fatal error happens there.
      </para>
      <para>
         When no debug flag is set, this macro is ignored.
      </para>
    </section>

    <section id="useful-functions-snd-bug-on">
      <title><function>snd_BUG_ON()</function></title>
      <para>
        <function>snd_BUG_ON()</function> macro is similar with
        <function>WARN_ON()</function> macro. For example,

        <informalexample>
          <programlisting>
<![CDATA[
  snd_BUG_ON(!pointer);
]]>
          </programlisting>
        </informalexample>

        or it can be used as the condition,
```

```
        <informalexample>
          <programlisting>
<![CDATA[
  if (snd_BUG_ON(non_zero_is_bug))
          return -EINVAL;
]]>
          </programlisting>
        </informalexample>

      </para>

      <para>
        The macro takes an conditional expression to evaluate.
        When <constant>CONFIG_SND_DEBUG</constant>, is set, the
        expression is actually evaluated. If it's non-zero, it shows
        the warning message such as
        <computeroutput>BUG? (xxx)</computeroutput>
        normally followed by stack trace.  It returns the evaluated
        value.
        When no <constant>CONFIG_SND_DEBUG</constant> is set, this
        macro always returns zero.
      </para>

    </section>

  </chapter>


<!-- ****************************************************** -->
<!-- Acknowledgments  -->
<!-- ****************************************************** -->
  <chapter id="acknowledgments">
    <title>Acknowledgments</title>
    <para>
      I would like to thank Phil Kerr for his help for improvement and
      corrections of this document.
    </para>
    <para>
    Kevin Conder reformatted the original plain-text to the
    DocBook format.
    </para>
    <para>
    Giuliano Pochini corrected typos and contributed the example codes
    in the hardware constraints section.
    </para>
  </chapter>
</book>
```