FreeBSD Architecture Handbook

Содержание

Глава 1. Замечания по блокировке

Эта глава поддерживается проектом FreeBSD SMP Next Generation Project. Комментарии и пожелания направляйте в Список рассылки, посвящённый поддержке многопроцессорности (SMP) во FreeBSD.

Этот документ описывает механизм блокировки, используемый в ядре FreeBSD для обеспечения эффективной поддержки нескольких процессоров в ядре. Блокировку можно рассматривать с нескольких точек зрения. Структуры данных могут быть защищены с помощью блокировок mutex или lockmgr(9). Несколько переменных защищены просто в силу атомарности используемых для доступа к ним операций.

1.1. Мьютексы

Мьютекс (mutex) - это просто блокировка, используемая для реализации гарантированной исключительности. В частности, в каждый момент времени мьютексом может владеть только один объект. Если какой-то объект хочет получить мьютекс, который уже кто-то занял, он должен дождаться момента его освобождения. В ядре FreeBSD владельцами мьютексов являются процессы.

Мьютексы могут быть затребованы рекурсивно, но предполагается, что они занимаются на короткое время. В частности, владельцу мьютекса нельзя выдерживать паузу. Если вам нужно выполнить блокировку на время паузы, используйте блокировку через lockmgr(9).

Каждый мьютекс имеет несколько представляющих интерес характеристик:

Имя переменной

Имя переменной struct mtx в исходных текстах ядра.

Логическое имя

Имя мьютекса, назначенное ему через `mtx_init`. Это имя выводится в сообщениях трассировки KTR и диагностических предупреждающих и ошибочных сообщениях и используется для идентификации мьютексов в отладочном коде.

Тип

Тип мьютекса в терминах флагов MTX_*. Значение каждого флага связано с его смыслом так, как это описано в mutex(9).

MTX_DEF

Sleep-мьютекс

MTX_SPIN

Spin-мьютекс

MTX_RECURSE

Этому мьютексу разрешается блокировать рекурсивно.

Защиты

Список структур данных или членов структур данных, которые защищает этот мьютекс. Для членов структур данных имя будет в форме . члена структуры/.

Зависимые функции

Функции, которые можно вызвать, если этот мьютекс занят.

Таблица 1. Список мьютексов

Variable Name	Logical Name	Type	Protectees	Dependent Functions
sched_lock	"sched lock"	MTX_SPIN | MTX_RECURSE	_gmonparam, cnt.v_swtch, cp_time, curpriority, mtx.mtx_blocked, mtx.mtx_contested, proc.p_procq, proc.p_slpq, proc.p_sflag, proc.p_stat, proc.p_estcpu, proc.p_cpticks proc.p_pctcpu, proc.p_wchan, proc.p_wmesg, proc.p_swtime, proc.p_slptime, proc.p_runtime, proc.p_uu, proc.p_su, proc.p_iu, proc.p_uticks, proc.p_sticks, proc.p_iticks, proc.p_oncpu, proc.p_lastcpu, proc.p_rqindex, proc.p_heldmtx, proc.p_blocked, proc.p_mtxname, proc.p_contested, proc.p_priority, proc.p_usrpri, proc.p_nativepri, proc.p_nice, proc.p_rtprio, pscnt, slpque, itqueuebits, itqueues, rtqueuebits, rtqueues, queuebits, queues, idqueuebits, idqueues, switchtime, switchticks	setrunqueue, remrunqueue, mi_switch, chooseproc, schedclock, resetpriority, updatepri, maybe_resched, cpu_switch, cpu_throw, need_resched, resched_wanted, clear_resched, aston, astoff, astpending, calcru, proc_compare
vm86pcb_lock	"vm86pcb lock"	MTX_DEF	vm86pcb	vm86_bioscall
Giant	"Giant"	MTX_DEF | MTX_RECURSE	nearly everything	lots
callout_lock	"callout lock"	MTX_SPIN | MTX_RECURSE	callfree, callwheel, nextsoftcheck, proc.p_itcallout, proc.p_slpcallout, softticks, ticks	

1.2. Разделяемые эксклюзивные блокировки

Эти блокировки обеспечивают базовый тип функциональности - на чтение/запись и могут поддерживаться процессами, находящимся в состоянии ожидания. На текущий момент они реализованы в lockmgr(9).

Таблица 2. Список разделяемых эксклюзивных блокировок

Имя переменной	Защиты
allproc_lock	allproc zombproc pidhashtbl proc.p_list proc .p_hash nextpid
proctree_lock	proc.p_children proc.p_sibling

1.3. Атомарно защищенные переменные

Переменной, защищенной атомарно, является особая переменная, которая не защищается явной блокировкой. Вместо этого для доступа к данным переменных используются специальные атомарные операции, как описано в atomic(9). Лишь несколько переменных используются таким образом, хотя другие примитивы синхронизации, такие как мьютексы, реализованы с атомарно защищенными переменными.

- mtx.mtx_lock

Глава 2. Написание драйверов устройств для FreeBSD

Эту главу написал Murray Stokely <murray@FreeBSD.org> на основе множества источников, включая справочную страницу intro(4), которую создал Jörg Wunsch <joerg@FreeBSD.org>.

2.1. Введение

Эта глава является кратким введением в процесс написания драйверов устройств для FreeBSD. В этом контексте термин устройство используется в основном для вещей, связанных с оборудованием, относящимся к системе, таких, как диски, печатающие устройства или графические дисплеи с клавиатурами. Драйвер устройства является программной компонентой операционной системы, управляющей некоторым устройством. Имеются также так называемые псевдо-устройства, в случае которых драйвер устройства эмулирует поведение устройства программно, без наличия какой-либо соответствующей аппаратуры. Драйверы устройств могут быть вкомпилированы в систему статически или могут загружаться по требованию при помощи механизма динамического компоновщика ядра `kld`.

Большинство устройств в Unix-подобной операционной системе доступны через файлы устройств (device-nodes), иногда также называемые специальными файлами. В иерархии файловой системы эти файлы обычно находятся в каталоге /dev. В версиях FreeBSD, более старых, чем 5.0-RELEASE, в которых поддержка devfs(5) не интегрирована в систему, каждый файл устройства должен создаваться статически и вне зависимости от наличия соответствующего драйвера устройства. Большинство файлов устройств в системе создаются при помощи команды `MAKEDEV`.

Драйверы устройств могут быть условно разделены на две категории; драйверы символьных и сетевых устройств.

2.2. Механизм динамического компоновщика ядра - KLD

Интерфейс kld позволяет системным администраторам динамически добавлять и убирать функциональность из работающей системы. Это позволяет разработчикам драйверов устройств загружать собственные изменения в работающее ядро без постоянных перезагрузок для тестирования изменений.

Для работы с интерфейсом kld используются следующие команды привилегированного режима:

- `kldload` - загружает новый модуль ядра
- `kldunload` - выгружает модуль ядра
- `kldstat` - выводит список загруженных в данный момент модулей

Скелет модуля ядра

```
/*
 * KLD Skeleton
 * Inspired by Andrew Reiter's Daemonnews article
 */

#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>  /* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  /* defines used in kernel.h */
#include <sys/kernel.h> /* types used in module initialization */

/*
 * Load handler that deals with the loading and unloading of a KLD.
 */

static int
skel_loader(struct module *m, int what, void *arg)
{
  int err = 0;

  switch (what) {
  case MOD_LOAD:                /* kldload */
    uprintf("Skeleton KLD loaded.\n");
    break;
  case MOD_UNLOAD:
    uprintf("Skeleton KLD unloaded.\n");
    break;
  default:
    err = EINVAL;
    break;
  }
  return(err);
}

/* Declare this module to the rest of the kernel */

static moduledata_t skel_mod = {
  "skel",
  skel_loader,
  NULL
};

DECLARE_MODULE(skeleton, skel_mod, SI_SUB_KLD, SI_ORDER_ANY);
```

2.2.1. Makefile

Во FreeBSD имеются заготовки для включения в make-файлы, которые вы можете использовать для быстрой компиляции собственных дополнений к ядру.

```
SRCS=skeleton.c
KMOD=skeleton

.include <bsd.kmod.mk>
```

Простой запуск команды `make` с этим make-файлом приведет к созданию файла skeleton.ko, который можно загрузить в вашу систему, набрав:

```
# kldload -v ./skeleton.ko
```

2.3. Обращение к драйверу устройства

Unix дает некоторый общий набор системных вызовов для использования в пользовательских приложениях. Когда пользователь обращается к файлу устройства, высокие уровни ядра перенаправляют эти обращения к соответствующему драйверу устройства. Скрипт `/dev/MAKEDEV` создает большинство файлов устройств в вашей системе, однако если вы ведете разработку своего собственного драйвера, то может появиться необходимость в создании собственных файлов устройств при помощи команды `mknod`.

2.3.1. Создание статических файлов устройств

Для создания файла устройства команде `mknod` требуется указать четыре аргумента. Вы должны указать имя файла устройства, тип устройства, старшее число устройства и младшее число устройства.

2.3.2. Динамические файлы устройств

Файловая система устройств, devfs, предоставляет доступ к пространству имен устройств ядра из глобального пространства имен файловой системы. Это устраняет потенциальную проблемы наличия драйвера без статического файла устройства или файла устройства без установленного драйвера устройства. Devfs все еще находится в разработке, однако она уже достаточно хорошо работает.

2.4. Символьные устройства

Драйвер символьного устройства передает данные непосредственно в или из процесса пользователя. Это самый распространенный тип драйвера устройства и в дереве исходных текстов имеется достаточно простых примеров таких драйверов.

В этом простом примере псевдо-устройство запоминает какие угодно значения, которые вы в него записываете, и затем может выдавать их назад при чтении из этого устройства.

Приведены две версии, одна для FreeBSD 4.X, а другая для FreeBSD 5.X.

Пример 1. Пример драйвера псевдо-устройства Echo для FreeBSD 4.X

```
/*
 * Simple `echo' pseudo-device KLD
 *
 * Murray Stokely
 */

#define MIN(a,b) (((a)  (b)) ? (a) : (b))

#include sys/types.h
#include sys/module.h
#include sys/systm.h  /* uprintf */
#include sys/errno.h
#include sys/param.h  /* defines used in kernel.h */
#include sys/kernel.h /* types used in module initialization */
#include sys/conf.h   /* cdevsw struct */
#include sys/uio.h    /* uio struct */
#include sys/malloc.h

#define BUFFERSIZE 256

/* Function prototypes */
d_open_t    echo_open;
d_close_t   echo_close;
d_read_t    echo_read;
d_write_t   echo_write;

/* Character device entry points */
static struct cdevsw echo_cdevsw = {
    echo_open,
    echo_close,
    echo_read,
    echo_write,
    noioctl,
    nopoll,
    nommap,
    nostrategy,
    "echo",
    33,             /* reserved for lkms - /usr/src/sys/conf/majors */
    nodump,
    nopsize,
    D_TTY,
    -1
};

struct s_echo {
    char msg[BUFFERSIZE];
    int len;
} t_echo;
```

```
/* vars */
static dev_t sdev;
static int len;
static int count;
static t_echo *echomsg;

MALLOC_DECLARE(M_ECHOBUF);
MALLOC_DEFINE(M_ECHOBUF, "echobuffer", "buffer for echo module");

/*
 * This function is called by the kld[un]load(2) system calls to
 * determine what actions to take when a module is loaded or unloaded.
 */

static int
echo_loader(struct module *m, int what, void *arg)
{
    int err = 0;

    switch (what) {
    case MOD_LOAD:                /* kldload */
        sdev = make_dev(echo_cdevsw,
                    0,
                    UID_ROOT,
                    GID_WHEEL,
                    0600,
                    "echo");
        /* kmalloc memory for use by this driver */
            MALLOC(echomsg, t_echo *, sizeof(t_echo), M_ECHOBUF, M_WAITOK);
        printf("Echo device loaded.\n");
        break;
    case MOD_UNLOAD:
        destroy_dev(sdev);
        FREE(echomsg,M_ECHOBUF);
        printf("Echo device unloaded.\n");
        break;
    default:
            err = EINVAL;
            break;
    }
    return(err);
}

int
echo_open(dev_t dev, int oflags, int devtype, struct proc *p)
{
    int err = 0;

    uprintf("Opened device \"echo\" successfully.\n");
    return(err);
```

```
}

int
echo_close(dev_t dev, int fflag, int devtype, struct proc *p)
{
    uprintf("Closing device \"echo.\"\n");
    return(0);
}

/*
 * The read function just takes the buf that was saved via
 * echo_write() and returns it to userland for accessing.
 * uio(9)
 */

int
echo_read(dev_t dev, struct uio *uio, int ioflag)
{
    int err = 0;
    int amt;

  /* How big is this read operation?  Either as big as the user wants,
     or as big as the remaining data */
  amt = MIN(uio->uio_resid, (echomsg->len - uio->uio_offset > 0) ? echomsg->len -
uio->uio_offset : 0);
  if ((err = uiomove(echomsg->msg + uio->uio_offset,amt,uio)) != 0) {
    uprintf("uiomove failed!\n");
  }

  return err;
}

/*
 * echo_write takes in a character string and saves it
 * to buf for later accessing.
 */

int
echo_write(dev_t dev, struct uio *uio, int ioflag)
{
  int err = 0;

  /* Copy the string in from user memory to kernel memory */
  err = copyin(uio->uio_iov->iov_base, echomsg->msg, MIN(uio->uio_iov-
>iov_len,BUFFERSIZE));

  /* Now we need to null terminate */
  *(echomsg->msg + MIN(uio->uio_iov->iov_len,BUFFERSIZE)) = 0;
  /* Record the length */
  echomsg->len = MIN(uio->uio_iov->iov_len,BUFFERSIZE);
```

```
  if (err != 0) {
    uprintf("Write failed: bad address!\n");
  }

  count++;
  return(err);
}

DEV_MODULE(echo,echo_loader,NULL);
```

Пример 2. Пример драйвера псевдо-устройства Echo для FreeBSD 5.X

```
/*
 * Simple `echo' pseudo-device KLD
 *
 * Murray Stokely
 *
 * Converted to 5.X by Sren (Xride) Straarup
 */

#include sys/types.h
#include sys/module.h
#include sys/systm.h  /* uprintf */
#include sys/errno.h
#include sys/param.h  /* defines used in kernel.h */
#include sys/kernel.h /* types used in module initialization */
#include sys/conf.h   /* cdevsw struct */
#include sys/uio.h    /* uio struct */
#include sys/malloc.h

#define BUFFERSIZE 256
#define CDEV_MAJOR      33

/* Function prototypes */
static d_open_t      echo_open;
static d_close_t     echo_close;
static d_read_t      echo_read;
static d_write_t     echo_write;

/* Character device entry points */
static struct cdevsw echo_cdevsw = {
       .d_open = echo_open,
       .d_close = echo_close,
       .d_maj = CDEV_MAJOR,
       .d_name = "echo",
       .d_read = echo_read,
       .d_write = echo_write
};

typedef struct s_echo {
       char msg[BUFFERSIZE];
       int len;
} t_echo;

/* vars */
static dev_t echo_dev;
static int count;
static t_echo *echomsg;

MALLOC_DECLARE(M_ECHOBUF);
```

```
MALLOC_DEFINE(M_ECHOBUF, "echobuffer", "buffer for echo module");

/*
 * This function is called by the kld[un]load(2) system calls to
 * determine what actions to take when a module is loaded or unloaded.
 */

static int
echo_loader(struct module *m, int what, void *arg)
{
       int err = 0;

       switch (what) {
       case MOD_LOAD:                /* kldload */
               echo_dev = make_dev(echo_cdevsw,
                   0,
                   UID_ROOT,
                   GID_WHEEL,
                   0600,
                   "echo");
               /* kmalloc memory for use by this driver */
               MALLOC(echomsg, t_echo *, sizeof(t_echo), M_ECHOBUF, M_WAITOK);
               printf("Echo device loaded.\n");
               break;
       case MOD_UNLOAD:
               destroy_dev(echo_dev);
               FREE(echomsg,M_ECHOBUF);
               printf("Echo device unloaded.\n");
               break;
       default:
               err = EINVAL;
               break;
       }
       return(err);
}

static int
echo_open(dev_t dev, int oflags, int devtype, struct thread *p)
{
       int err = 0;

       uprintf("Opened device \"echo\" successfully.\n");
       return(err);
}

static int
echo_close(dev_t dev, int fflag, int devtype, struct thread *p)
{
       uprintf("Closing device \"echo.\"\n");
       return(0);
}
```

```
/*
 * The read function just takes the buf that was saved via
 * echo_write() and returns it to userland for accessing.
 * uio(9)
 */

static int
echo_read(dev_t dev, struct uio *uio, int ioflag)
{
    int err = 0;
    int amt;

    /*
         * How big is this read operation?  Either as big as the user wants,
         * or as big as the remaining data
         */
    amt = MIN(uio->uio_resid, (echomsg->len - uio->uio_offset > 0) ?
           echomsg->len - uio->uio_offset : 0);
        if ((err = uiomove(echomsg->msg + uio->uio_offset,amt,uio)) != 0) {
        uprintf("uiomove failed!\n");
    }
    return(err);
}

/*
 * echo_write takes in a character string and saves it
 * to buf for later accessing.
 */

static int
echo_write(dev_t dev, struct uio *uio, int ioflag)
{
       int err = 0;

       /* Copy the string in from user memory to kernel memory */
       err = copyin(uio->uio_iov->iov_base, echomsg->msg,
           MIN(uio->uio_iov->iov_len,BUFFERSIZE - 1));

       /* Now we need to null terminate, then record the length */
       *(echomsg->msg + MIN(uio->uio_iov->iov_len,BUFFERSIZE - 1)) = 0;
       echomsg->len = MIN(uio->uio_iov->iov_len,BUFFERSIZE);

       if (err != 0) {
        uprintf("Write failed: bad address!\n");
       }
       count++;
       return(err);
}

DEV_MODULE(echo,echo_loader,NULL);
```

Для установки этого драйвера во FreeBSD 4.X сначала вам нужно создать файл устройства в вашей файловой системе по команде типа следующей:

```
# mknod /dev/echo c 33 0
```

Когда этот драйвер загружен, вы можете выполнять следующие действия:

```
# echo -n "Test Data" > /dev/echo
# cat /dev/echo
Test Data
```

Устройства, обслуживающие реальное оборудование, описываются в следующей главе.

Дополнительные источники информации

- Учебник по программированию механизма динамического компоновщика ядра (KLD) - Daemonnews Октябрь 2000
- Как писать драйверы ядра в парадигме NEWBUS - Daemonnews Июль 2000

2.5. Блочные устройства (которых больше нет)

Другие UNIX®-системы могут поддерживать со вторым типом дисковых устройств, так называемых устройств с блочной организацией. Блочные устройства являются дисковыми устройствами, для которых ядро организует кэширование. Такое кэширование делает блочные устройства практически бесполезными, или по крайней мере ненадёжными. Кэширование изменяет последовательность операций записи, лишая приложение возможности узнать реальное содержимое диска в любой момент времени. Это делает предсказуемое и надежное восстановление данных на диске (файловые системы, базы данных и прочее) после сбоя невозможным. Так как запись может быть отложенной, то нет способа сообщить приложению, при выполнении какой именно операции записи ядро встретилось с ошибкой, что таким образом осложняет проблему целостности данных. По этой причине серьёзные приложения не полагаются на блочные устройства, и, на самом деле практически во всех приложениях, которые работают с диском напрямую, имеется большая проблема выбора устройств с последовательным доступом (или "raw"), которые должны использоваться. Из-за реализации отображения каждого диска (раздела) в два устройства с разными смыслами, которая усложняет соответствующий код ядра, во FreeBSD поддержка дисковых устройств с кэшированием была отброшена в процессе модернизации инфраструктуры I/O-операций с дисками.

2.6. Сетевые драйверы

В случае драйверов сетевых устройств файлы устройств для доступа к ним не используются. Их выбор основан на другом механизме, работающем в ядре, и не использующем вызов open(); об использование сетевых устройств в общем случае рассказано в описании системного вызова socket(2).

Почитайте справочную информацию о вызове ifnet(), устройстве loopback, почитайте драйверы Билла Пола (Bill Paul), и так далее..

Глава 3. Подсистема звука

3.1. Введение

Перевод на русский язык: Виталий Богданов (gad@gad.glazov.net)

В подсистеме звука FreeBSD существует чёткое разделение между частью, поддерживающей общие звуковые возможности и аппаратно зависимой частью. Данная особенность делает более простым добавление поддержки новых устройств.

pcm(4) занимает центральное место в подсистеме звука. Его основными элементами являются:

- Интерфейс системных вызовов (read, write, ioctls) к функциям оцифрованного звука и микшера. Командный набор ioctl совместим с интерфейсом *OSS* или *Voxware*, позволяя тем самым портирование мультимедиа приложений без дополнительной модификации.
- Общий код обработки звуковых данных (преобразования форматов, виртуальные каналы).
- Единый программный интерфейс к аппаратно-зависимым модулям звукового интерфейса.
- Дополнительная поддержка нескольких общих аппаратных интерфейсов (ac97) или разделяемого аппаратно-специфичного кода (например: функции ISA DMA).

Поддержка отдельных звуковых карт осуществляется с помощью аппаратно-специфичных драйверов, обеспечивающих канальные и микшерные интерфейсы, включаемые в общий код.

В этой главе термином мы будем называть центральную, общую часть звукового драйвера, как противопоставление аппаратно-специфичным модулям.

Человек, решающий написать драйвер наверняка захочет использовать в качестве шаблона уже существующий код. Но, если звуковой код хорош и чист, он также в основном лишён комментариев. Этот документ - попытка рассмотрения базового интерфейса и попытка ответить на вопросы, возникшие при адаптировании существующего кода.

Для старта с рабочего примера, вы можете найти шаблон драйвера, оснащенного комментариями на http://people.FreeBSD.org/~cg/template.c

3.2. Файлы

Весь исходный код, на сегодняшний момент (FreeBSD 4.4), содержится в каталоге /usr/src/sys/dev/sound/, за исключением публичных определений интерфейса ioctl, находящихся в /usr/src/sys/sys/soundcard.h

В подкаталоге pcm/ родительского каталога /usr/src/sys/dev/sound/ находится главный код, а в каталогах isa/ и pci/ содержатся драйвера для ISA и PCI карт.

3.3. Обнаружение, подключение, и т.д.

Обнаружение и подключение звуковых драйверов во многом схоже с драйвером любого другого устройства. За дополнительной информацией вы можете обратиться к главам ISA или PCI данного руководства.

Но всё же, звуковые драйвера немного отличаются:

- Они объявляют сами себя, как устройства класса , с частной структурой устройства :

  ```
  static driver_t xxx_driver = {
      "pcm",
      xxx_methods,
      sizeof(struct snddev_info)
  };

  DRIVER_MODULE(snd_xxxpci, pci, xxx_driver, pcm_devclass, 0, 0);
  MODULE_DEPEND(snd_xxxpci, snd_pcm, PCM_MINVER, PCM_PREFVER,PCM_MAXVER);
  ```

  Большинство звуковых драйверов нуждаются в сохранении личной информации, касающейся их устройства. Структура с личными данными обычно выделяется при вызове функции attach. Её адрес передаётся посредством вызовов `pcm_register()` и `mixer_init()`. Позже передаёт назад этот адрес, в качестве параметра в вызовах к интерфейсам звукового драйвера.

- Функция подключения звукового драйвера должна объявлять её микшерный или AC97 интерфейс посредством вызова `mixer_init()`. Для микшерного интерфейса это взамен вернёт вызов `xxxmixer_init()`.
- Функция подключения звукового драйвера передаёт общие настройки каналов посредством вызова `pcm_register(dev, sc, nplay, nrec)`, где `sc` - адрес структуры данных устройства, используемой в дальнейших вызовах от , а `nplay` и `nrec` - количество каналов проигрывания и записи.
- Функция подключения звукового драйвера объявляет каждый из её каналов с помощью вызовов `pcm_addchan()`. Это установит занятость канала в и вызовет взамен вызов `xxxchannel_init()`.
- Функция отключения должна вызывать `pcm_unregister()` перед объявлением её ресурсов свободными.

Существует два метода работы с не PnP устройствами:

- Использование метода `device_identify()` (пример смотрите в: sound/isa/es1888.c). `device_identify()` пытается обнаружить оборудование, использующее известные адреса, и если найдёт поддерживаемое устройство, то создаст новое pcm устройство, которое затем будет передано процессу обнаружения/подключения.
- Использование выборочной конфигурации ядра с соответствующими хинтами для pcm устройств (пример: sound/isa/mss.c).

драйверы должны поддерживать `device_suspend`, `device_resume` и `device_shutdown` функции, для корректного функционирования управления питанием и процесса выгрузки модуля.

3.4. Интерфейсы

Интерфейс между и звуковыми драйверами определён в терминах объектов ядра.

Есть 2 основных интерфейса, которые обычно обеспечивает звуковой драйвер: *канальный* и, либо *микшерный* либо *AC97*.

Интерфейс *AC97* довольно мало использует доступ к ресурсам оборудования (чтение/запись регистров). Данный интерфейс реализован в драйверах для карт с кодеком AC97. В этом случае фактический микшерный интерфейс обеспечивается разделяемым кодом AC97 в .

3.4.1. Канальный интерфейс

3.4.1.1. Общие заметки о параметрах функций

Звуковые драйверы обычно имеют структуру с личными данными для описания их устройства и по одной структуре на каждый поддерживаемый канал проигрывания или записи данных.

Для всех функций канального интерфейса первый параметр - непрозрачный указатель.

Второй параметр это указатель на структуру с данными канала. Исключение: У `channel_init()` это указатель на частную структуру устройства (данная функция возвращает указатель на канал для дальнейшего использования в).

3.4.1.2. Обзор операций передачи данных

Для передачи данных, и звуковые драйвера используют разделяемую область памяти, описанную в .

принадлежит , и звуковые драйверы получают нужные значения с помощью вызовов функций (`sndbuf_getxxx()`).

Область разделяемой памяти имеет размер, определяемый с помощью `sndbuf_getsize()` и разделён на блоки фиксированного размера, определённого в `sndbuf_getblksz()` количества байт.

При проигрывании, общий механизм передачи данных примерно следующий (обратный механизму, используемому при записи):

- В начале, заполняет буфер, затем вызывает функцию звукового драйвера `xxxchannel_trigger()` с параметром PCMTRIG_START.
- Затем звуковой драйвер многократно передаёт всю область памяти (`sndbuf_getbuf()`, `sndbuf_getsize()`) устройству, с количеством байт, определённым в `sndbuf_getblksz()` . Взамен это вызовет `chn_intr()` функцию для каждого переданного блока (это обычно происходит во время прерывания).

- `chn_intr()` копирует новые данные в область, которая была передана устройству (сейчас свободная) и вносит соответствующие изменения в структуру .

3.4.2. channel_init

`xxxchannel_init()` вызывается для инициализации каждого из каналов проигрывания или записи. Вызовы инициируются функцией подключения звукового драйвера. (Подробнее в главе Обнаружение и подключение).

```
          static void *
          xxxchannel_init(kobj_t obj, void *data,
             struct snd_dbuf *b, struct pcm_channel *c, int dir)
          {
              struct xxx_info *sc = data;
              struct xxx_chinfo *ch;
               ...
              return ch;
           }

            b - это адрес канальной
              struct snd_dbuf.  Она должна
          быть инициализирована в функции посредством
          вызова sndbuf_alloc().  Нормальный
          размер буфера для использования - наименьшее кратное
          размера передаваемого блока данных для вашего устройства.

            c - это
              указатель на структуру
          контроля pcm канала.  Это не прозрачный
          объект.  Функция должна хранить его в локальной структуре
          канала, для дальнейшего использования в вызовах к
              pcm (например в:
              chn_intr(c)).

            dir определяет для каких целей
          используется канал
          (PCMDIR_PLAY или
              PCMDIR_REC).

            Функция должна возвращать указатель на личную,
          область, используемую для контроля этого
          канала. Он будет передаваться в качестве параметра в
          других вызовах канального интерфейса.

        channel_setformat

        xxxchannel_setformat() настраивает
      устройство на конкретный канал определённого формата звука.

                    static int
```

```
    xxxchannel_setformat(kobj_t obj, void *data, u_int32_t format)
    {
        struct xxx_chinfo *ch = data;
         ...
        return 0;
     }

      format используется, как
        AFMT_XXX значение
        (soundcard.h).

  channel_setspeed

  xxxchannel_setspeed() устанавливает
оборудование канала на определённую шаблонную скорость и возвращает
возможную корректирующую скорость.

            static int
    xxxchannel_setspeed(kobj_t obj, void *data, u_int32_t speed)
    {
        struct xxx_chinfo *ch = data;
         ...
        return speed;
     }

  channel_setblocksize

  xxxchannel_setblocksize() устанавливает
    размер передаваемого блока между
    pcm и звуковым драйвером, и между
    звуковым драйвером и устройством.  Обычно это будет количество
переданных байт перед прерыванием. Во время трансфера звуковой
драйвер должен должен вызывать
pcm функцию chn_intr() каждый
раз при передаче блока данных такого размера.

  Большинство звуковых драйверов только берут на заметку
размер блока для использования во время передачи данных.

            static int
    xxxchannel_setblocksize(kobj_t obj, void *data, u_int32_t blocksize)
    {
        struct xxx_chinfo *ch = data;
          ...
        return blocksize;
     }

      Функция возвращает возможно согласованный  размер
    блока.  В случае, если размер блока действительно
    изменился должен быть произведён вызов
        sndbuf_resize() для корректирования
```

```
    буфера.

  channel_trigger

  xxxchannel_trigger() вызывается
    pcm для контроля над трансферными
операциями в драйвере.

            static int
    xxxchannel_trigger(kobj_t obj, void *data, int go)
    {
        struct xxx_chinfo *ch = data;
         ...
        return 0;
     }

      go определяет действие для
    текущего вызова.  Возможные значения:

          PCMTRIG_START: драйвер
            должен начать передачу данных из или в канальный
        буфер.  Буфер и его размер могут быть получены через
            вызов sndbuf_getbuf() и
            sndbuf_getsize().

          PCMTRIG_EMLDMAWR /
            PCMTRIG_EMLDMARD: говорит
    драйверу, что входной или выходной буфер возможно
    был обновлён.  Большинство драйверов игнорируют
    эти вызовы.

          PCMTRIG_STOP /
            PCMTRIG_ABORT: драйвер должен
            остановить текущую передачу данных.

  Если драйвер использует ISA DMA,
    sndbuf_isadma() должна вызываться
перед выполнением действий над устройством, она также
позаботится о вещах со стороны DMA чипа.

  channel_getptr

  xxxchannel_getptr() возвращает
    текущее смещение в передаваемом буфере.  Обычно вызывается
в chn_intr(), и так
    pcm узнаёт, где брать данные для
новой передачи.

  channel_free

  xxxchannel_free() вызывается для
```

```
освобождения ресурсов канала.  Например: должна вызываться,
    при выгрузке драйвера, если структуры данных канала
распределялись динамично или, если
sndbuf_alloc() не использовалась
для выделения памяти под буфер.

  channel_getcaps

            struct pcmchan_caps *
    xxxchannel_getcaps(kobj_t obj, void *data)
    {
        return xxx_caps;
     }

      Подпрограмма возвращает указатель на (обычно
    статически-определяемую) структуру
        pcmchan_caps (описанную в
        sound/pcm/channel.h.  Структура содержит
    данные о минимуме и максимуме шаблонных частот и
    воспринимаемых звуковых форматах.  Для примера смотрите
    исходный код любого звукового драйвера.

  Другие функции

  channel_reset(),
    channel_resetdone(), и
    channel_notify() предназначены для
специальных целей и не должны употребляться в драйвере
без обсуждения с авторами ({cg}).

  channel_setdir() is deprecated.

Микшерный интерфейс

  mixer_init

  xxxmixer_init() инициализирует
    оборудование и говорит pcm какие микшерные
    устройства доступны для проигрывания и записи

            static int
    xxxmixer_init(struct snd_mixer *m)
    {
        struct xxx_info   *sc = mix_getdevinfo(m);
        u_int32_t v;

        [Initialize hardware]

        [Set appropriate bits in v for play mixers]
        mix_setdevs(m, v);
        [Set appropriate bits in v for record mixers]
```

```
        mix_setrecdevs(m, v)

        return 0;
    }

      Устанавливает биты в целом значении и вызывает
        mix_setdevs() и
        mix_setrecdevs() чтобы сообщить
        pcm какие устройства существуют.

  Определения битов микшера могут быть найдены в
    soundcard.h
    (SOUND_MASK_XXX значения и
    SOUND_MIXER_XXX битовые сдвиги).

  mixer_set

  xxxmixer_set() устанавливает уровень
    громкости для одного микшерного устройства.

            static int
    xxxmixer_set(struct snd_mixer *m, unsigned dev,
                     unsigned left, unsigned right)
    {
        struct sc_info *sc = mix_getdevinfo(m);
        [set volume level]
        return left | (right  8);
    }

      Устройство определяется, как SOUND_MIXER_XXX
        значение Допустимые значения уровней громкости лежат
        в пределах [0-100].  Равное нулю значение должно выключать звук
        устройства.

      Вероятно уровни оборудования не будут совпадать с
        входной шкалой, и будет происходить некоторое округление, подпрограмма
        будет возвращает точные значения (в промежутке 0-100), как уже
        было сказано.

  mixer_setrecsrc

  xxxmixer_setrecsrc() устанавливает
    исходное записывающее устройство.

            static int
    xxxmixer_setrecsrc(struct snd_mixer *m, u_int32_t src)
    {
        struct xxx_info *sc = mix_getdevinfo(m);

        [look for non zero bit(s) in src, set up hardware]
```

```
            [update src to reflect actual action]
            return src;
          }

           Желаемые записывающие устройства указываются в битовом поле

           Возвращается фактический набор устройств для записи.
             Некоторые драйверы могут устанавливать только одно устройство для
         записи.  Функция должна возвращать -1, в случае возникновения
         ошибки.

       mixer_uninit, mixer_reinit

       xxxmixer_uninit() должна проверить,
         что все звуки выключены (mute), и, если возможно выключить
     оборудование микшера

       xxxmixer_reinit() должна удостовериться,
         что оборудование микшера включено и все установки, неконтролируемые
         mixer_set() или
         mixer_setrecsrc() восстановлены.

     Интерфейс AC97

      AC97

     Поддержка интерфейса AC97 осуществляется
   драйверами с кодеком AC97.  Он поддерживает только три метода:

       xxxac97_init() возвращает
         количество найденных ac97 кодеков.

       ac97_read() и
         ac97_write() читают или записывают
     данные определенного регистра.

     Интерфейс AC97 используется кодом
       AC97 в pcm для выполнения операций
   более высокого уровня.  За примером обращайтесь к
       sound/pci/maestro3.c или к другим
   файлам из каталога sound/pci/.
```