

FreeBSD 系□□□手册

摘要

欢迎 FreeBSD 系手册。 本手册在不断由多人编写。 多章是空白，有的章亟待更新。 如果你对目感兴趣并有意有所献，信 [FreeBSD 文档文件列表](#)。

本文的最新英文原始版本可从 [FreeBSD Web 站点](#) 得， 由 <http://www.FreeBSD.org.cn> 的最新 本可以在 <http://www.FreeBSD.org.cn> 快照 Web 站点 和 <http://www.FreeBSD.org.cn> 文快照 得， 一本会不断向主站同。 此外， 也可以从 [FreeBSD FTP 服务器](#) 或多的 [镜像站点](#) 得到文的各 其他格式以及形式的版本。

目 录

I: 内核	5
1. 引导程序与内核初始化	6
1.1. 概述	6
1.2. 引导	6
1.3. BIOS POST	7
1.4. boot0 段	7
1.5. boot2 段	8
1.6. loader 段	11
1.7. 内核初始化	11
2. 内核中的锁	21
2.1. Mutex	21
2.2. 共享互斥锁	23
2.3. 原子保锁量	23
3. 内核对象	24
3.1. 对象	24
3.2. Kobj 的工作流程	24
3.3. 使用Kobj	24
4. Jail 子系统	29
4.1. Jail 的子系统	29
4.2. 子系统被囚禁程序的限制	35
5. SYSINIT 框架	41
5.1. 对象	41
5.2. SYSINIT 操作	41
5.3. 使用SYSINIT	41
6. TrustedBSD MAC 框架	44
6.1. MAC 文档版声明	44
6.2. 文档解析	45
6.3. 概述	45
6.4. 安全策略背景知识	45
6.5. MAC 框架的内核体系	45
6.6. MAC 策略模型体系	49
6.7. MAC 策略入口函数参考	51
6.8. 应用体系	109
6.9. 小结	110
7. 虚拟内存系统	111
7.1. 物理内存的管理-vm_page_t	111
7.2. 唯一的内存信息对象-vm_object_t	112
7.3. 文件系统输入/输出-buf 对象	112

7.4. 映射表-vm_map_t, vm_entry_t	112
7.5. KVM存映射	112
7.6. 调整FreeBSD的虚内存系	113
8. SMPng 文档	114
8.1. 文档	114
8.2. 基本工具与上的基础知识	114
8.3. 架构与概	115
8.4. 特定数据的策略	117
8.5. 说明	120
8.6. 其它	122
表	122
II: 程序	124
写 FreeBSD 程序	125
1. 介绍	125
2. 内核接口工具-KLD	125
3. 程序	127
4. 字符	127
5. (消亡中)	135
6. 网程序	135
9. ISA程序	136
9.1. 概述	136
9.2. 基本信息	136
9.3. Device_t指	138
9.4. 配置文件与自配置期和探的顺序	138
9.5. 源	140
9.6. 内存映射	142
9.7. DMA	148
9.8. xxx_isa_probe	150
9.9. xxx_isa_attach	157
9.10. xxx_isa_detach	160
9.11. xxx_isa_shutdown	161
9.12. xxx_intr	161
10. PCI	163
10.1. 探与接	163
10.2. 源	167
11. 通用方法SCSI控制器	171
11.1. 提	171
11.2. 通用基	171
11.3. 文档	191
11.4. 事件	192
11.5. 中断	193

11.6. 扫描	200
11.7. 超理	201
12. USB	202
12.1. 介	202
12.2. 主控器	202
12.3. USB信息	204
12.4. 的探测和接	205
12.5. USB程序的的信息	206
13. Newbus	208
13.1. 程序	208
13.2. Newbus概	208
13.3. Newbus API	210
14. 声音子系	212
14.1. 介	212
14.2. 文件	212
14.3. 探测, 接等	212
14.4. 接口	213
15. PC Card	219
15.1. 添加	219
III: 附	224
参考目	225

Part I: 内核

Chapter 1. 引导程序与内核初始化

1.1. 概述

这一章是引导程序和系统初始化程序的简介。这些程序始于BIOS(固件)POST, 直到第一个用户程序建立。由于系统的最初阶段是与硬件紧密相关的、是紧密配合的, 这里用IA-32(Intel Architecture 32bit)作为例子。

1.2. 简介

一台运行FreeBSD的计算机有多引导方法。这里介绍其中最通常的方法, 也就是从安装了操作系统的硬盘上引导。引导程序分几步完成:

- BIOS POST
- `boot0`阶段
- `boot2`阶段
- `loader`阶段
- 内核初始化

`boot0`和`boot2`阶段在手册 [boot\(8\)](#)中被称作*bootstrap stages 1 and 2*, 是FreeBSD的三个阶段引导程序的开始。在每一阶段都有各自的信息显示在屏幕上, 可以参考下表列出一些信息。注意信息的显示内容可能随机器的不同而有一些区别:

不同机器而定	BIOS(固件)消息
<pre>F1 FreeBSD F2 BSD F5 Disk 2</pre>	<code>boot0</code>
<pre>>>FreeBSD/i386 BOOT Default: 1:ad(1,a)/boot/loader boot:</pre>	<code>boot2</code>

<pre> BTX loader 1.0 BTX version is 1.01 BIOS drive A: is disk0 BIOS drive C: is disk1 BIOS 639kB/64512kB available memory FreeBSD/i386 bootstrap loader, Revision 0.8 Console internal video/keyboard (jkh@bento.freebsd.org, Mon Nov 20 11:41:23 GMT 2000) /kernel text=0x1234 data=0x2345 syms=[0x4+0x3456] Hit [Enter] to boot immediately, or any other key for command prompt Booting [kernel] in 9 seconds..._ </pre>	loader
<pre> Copyright (c) 1992-2002 The FreeBSD Project. Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994 The Regents of the University of California. All rights reserved. FreeBSD 4.6-RC #0: Sat May 4 22:49:02 GMT 2002 devnull@kukas:/usr/obj/usr/src/sys/DEVNULL Timecounter "i8254" frequency 1193182 Hz </pre>	内核

1.3. BIOS POST

当PC加电后，处理器的寄存器被置为某些特定值。在某些寄存器中，指令指针寄存器被置为32位0xffffffff。指令指针寄存器指向处理器将要执行的指令地址。cr1，一个32位控制寄存器，在启动时被置为0。cr1的PE(Protected Enabled, 保护模式使能)位用来指示处理器是处于保护模式还是实地址模式。由于寄存器位被清位，处理器在实地址模式中运行。在实地址模式中，线性地址与物理地址是等同的。

0xffffffff略小于4G,因此计算机没有4G字节的物理内存，因此它就不会是一个有效的内存地址。计算机硬件将寄存器指向BIOS存储器。

BIOS表示Basic Input Output System (基本输入输出系统)。在主板上，它被固化在一个容量较小的只读存储器(Read-Only Memory, ROM)。BIOS包含各种各样的主板硬件定制的底层例程。就BIOS而言，处理器首先指向常駐BIOS存储器的地址 0xffffffff。通常这个位置包含一条跳指令，指向BIOS的POST例程。

POST表示Power On Self Test(加电自检)。该套程序包括内存测试，系统总线测试和其它底层工具，从而使CPU能够初始化整台计算机。BIOS代码中有一个重要部分，就是引导扇区。BIOS在所有的BIOS都允许手工引导。BIOS可以从硬盘、光驱、硬盘等引导。

POST的最后一部分是执行INT 0x19指令。该指令从引导扇区第一个扇区读取512字节，装入地址0x7c00。第一个扇区的划分最早起源于硬盘的划分，硬盘面被划分为若干柱形磁道。磁道编号，同时又将磁道分为一定数目(通常是64)的扇形。0号磁道是硬盘的最外圈，1号扇区，第一个扇区(磁道、柱面都从0开始编号，而扇区从1开始编号)有着特殊的作用，它又被称为主引导记录(Master Boot Record, MBR)。第一扇区剩余的扇区常常不使用。

1.4. boot0扇区

我们来看一下文件/boot/boot0。它是一个512字节的小文件。如果在FreeBSD安装过程中使用 "bootmanager"，该文件中的内容将被写入硬盘MBR

如前所述, `INT 0x19` 指令装 `MBR`, 也就是 `boot0` 的内容至内存地址 `0x7c00`。再看文件 `sys/boot/i386/boot0/boot0.S`, 可以猜想里面生了什 - 是引导管理器, 一段由 Robert Nordier写的令人起敬的程序片段。

`MBR`里, 也就是`boot0`里, 从偏移量`0x1be`始有一个特殊的, 称 分区表。其中有4条 (称分区), 条16字。分区表示硬如何被分, 在FreeBSD的, 被称`slice(d)`。16字中有一个标志字决定个分区是否可引。有只能有一个分区可定一标志。否, `boot0`的代将拒行。

一个分区有如下域:

- 1字 文件系统型
- 1字 可引标志
- 6字 CHS格式描述符
- 8字 LBA格式描述符

一个分区描述符包含某一分区在硬上的切位置信息。

`CHS`描述符指示相同的信息, 但是指示方式有所不同: `LBA` (地址, Logical Block Addressing)指示分区的起始扇区和分区度, 而`CHS`(柱面 磁 扇区)指示首扇区和末扇区

引导管理器描分区表, 并在屏幕上示菜, 以使用可以 用于引导的磁和分区。在上按下相的, `boot0`行如下作:

- 中的分区可引, 清除以前的可引标志
- 住本次的分区以下次引作缺省
- 装中分区的第一个扇区, 并跳行之

什数据会存在于一个可引扇区(里指FreeBSD扇区)的第一扇区里? 正如已猜到的, 那就是`boot2`。

1.5. `boot2`段

也想知道, 什`boot2`是在 `boot0`之后, 而不是在`boot1`之后。事上, 也有一个512字的文件 `boot1`存放在目 `/boot`里, 那是用来从一引导系的。从引导, `boot1`起着 `boot0`硬引相同的作用:它到 `boot2`并行之。

可能已看到有一文件`/boot/mbr`。是`boot0`的化版本。 `mbr`中的代不会示菜用, 而只是的引被志的分区。

`boot2`的代存放在目 `sys/boot/i386/boot2/`里, 的可行文件在 `/boot`里。在`/boot`里的文件 `boot0`和`boot2`不会在引程中使用, 只有`boot0cfg`的工具才会使用它。 `boot0`的内容在`MBR`中才能生效。`boot2`位于可引的FreeBSD分区的始。些位置不受文件系统控制, 所以它不可用`ls`之的命令看。

`boot2`的主要任是装文件 `/boot/loader`, 那是引程的第三段。在`boot2`中的代不能使用如 `open()`和`read()` 之的例程函数,因内核没有被加。而当描硬, 取文件系统, 到文件`/boot/loader`, 用BIOS的功能将它入内存, 然后从其入口点始行之。

除此之外, `boot2`可提示用行, `loader`可以从其它磁、系元、分区装。

`boot2`的二制代用特殊的方式生:

```
sys/boot/i386/boot2/Makefile
boot2: boot2.ldr boot2.bin ${BTX}/btx/btx
    btxld -v -E ${ORG2} -f bin -b ${BTX}/btx/btx -l boot2.ldr \
        -o boot2.ldr -P 1 boot2.bin
```

这个Makefile片段表明**btxld(8)**被用来链接二进制代码。BTX表示引导扩展器(Boot eXtender)是程序(称客户(client))提供保护模式环境、并与客户程序相链接的一段代码。所以 **boot2**是一个BTX客户, 使用BTX提供的服务。

工具**btxld**是链接器, 它将二进制代码链接在一起。**btxld(8)**和**ld(1)**的区别是ld通常将多个目标文件链接成一个可执行文件或可链接库, 而**btxld**将一个目标文件与BTX链接起来, 生成符合于放在分区首部的二进制代码, 以系统引导。

boot0行跳至BTX的入口点。然后, BTX将处理器切换至保护模式, 并准备一个初始环境, 然后用客户。这个环境包括:

- 虚拟8086模式。这意味着BTX是虚拟8086的程序。保护模式指令, 如pushf, popf, cli, sti, if, 均可被客户用。
- 建立中断描述符表(Interrupt Descriptor Table, IDT), 使得所有的硬件中断可被缺省的BIOS程序处理。建立中断0x30, 是系统用接口。
- 这个系统用**exec**和**exit**的定义如下:

```
sys/boot/i386/btx/lib/btxsys.s:
    .set INT_SYS,0x30      # 中断号
#
# System call: exit
#
__exit:    xorl %eax,%eax   # BTX系统用0x0
           int $INT_SYS    #
#
# System call: exec
#
__exec:    movl $0x1,%eax   # BTX系统用0x1
           int $INT_SYS    #
```

BTX建立全局描述符表(Global Descriptor Table, GDT):

```
sys/boot/i386/btx/btx/btx.s:
gdt:      .word 0x0,0x0,0x0,0x0    # 以空入口
           .word 0xffff,0x0,0x9a00,0xcf # SEL_SCODE
           .word 0xffff,0x0,0x9200,0xcf # SEL_SDATA
           .word 0xffff,0x0,0x9a00,0x0 # SEL_RCODE
           .word 0xffff,0x0,0x9200,0x0 # SEL_RDATA
           .word 0xffff,MEM_USR,0xfa00,0xcf# SEL_UCODE
           .word 0xffff,MEM_USR,0xf200,0xcf# SEL_UDATA
           .word _TSSLM,MEM_TSS,0x8900,0x0 # SEL_TSS
```

客程序的代和数据始于地址MEM_USR(0xa000)，选择符(selector) SEL_UCODE指向客程序的数据段。选择符 SEL_UCODE 有第3描述符权限 (Descriptor Privilege Level, DPL)，它是最低权限。但是 INT 0x30 指令的 0 程序存于一个段里，一个段的选择符SEL_SCODE (supervisor code)由有着管理权限。正如代建立 IDT(中断描述符表)行的操作那：

```

        mov $SEL_SCODE,%dh      # 段选择符
init.2:  shr %bx                 # 是否处理一个中断？
        jnc init.3              # 否
        mov %ax, (%di)          # 置程序偏移量
        mov %dh, 0x2(%di)       # 置程序选择符
        mov %dl, 0x5(%di)       # 置 P:DPL:type
        add $0x4,%ax            # 下一个中断程序

```

所以，当客程序用 `__exec()`，代将被以最高权限运行。这使得内核可以修改保护模式数据，如分页表(page tables)、全局描述符表(GDT)、中断描述符表(IDT)等。

boot2 定义了一个重要的数据结构：`struct bootinfo`。它由 **boot2** 初始化，然后被送到loader，之后又被入内核。它的部分由 **boot2** 定，其余的由loader定。它包含的信息包括内核文件名、BIOS提供的硬盘柱面/磁头/扇区数目信息、BIOS提供的引导设备的号，可用的物理内存大小，`envp` 指向环境指针等。定义如下：

```

/usr/include/machine/bootinfo.h
struct bootinfo {
    u_int32_t    bi_version;
    u_int32_t    bi_kernelname;    /* 用一个字表示 */
    u_int32_t    bi_nfs_diskless;  /* struct nfs_diskless */
    /* 以上常用 */
#define bi_endcommon    bi_n_bios_used
    u_int32_t    bi_n_bios_used;
    u_int32_t    bi_bios_geom[N_BIOS_GEOM];
    u_int32_t    bi_size;
    u_int8_t     bi_memsizes_valid;
    u_int8_t     bi_bios_dev;      /* 引导的BIOS设备号 */
    u_int8_t     bi_pad[2];
    u_int32_t    bi_basemem;
    u_int32_t    bi_extmem;
    u_int32_t    bi_syntab;        /* struct syntab */
    u_int32_t    bi_esyntab;       /* struct syntab */
    /* 以下由高级bootloader提供 */
    u_int32_t    bi_kernend;       /* 内核空末端 */
    u_int32_t    bi_envp;         /* 环境 */
    u_int32_t    bi_modulep;      /* 装载的模式 */
};

```

boot2 进入一个循环等待用户输入，然后用 `load()`。如果用不做任何输入，循环将在一段后结束，`load()` 将会装缺省文件(/boot/loader)。函数 `ino_t lookup(char *filename)`和 `int xfsread(ino_t inode, void *buf, size_t nbyte)` 用来将文件内容入内存。/boot/loader是一个ELF格式二进制文件，不它的部被成了a.out格式中的`struct exec`。 `load()`描loader的ELF部，装/boot/loader 至内存，然后跳

至入口行之：

```
sys/boot/i386/boot2/boot2.c:
    __exec((caddr_t)addr, RB_BOOTINFO | (opts & RBX_MASK),
        MAKEBOOTDEV(dev_maj[dsk.type], 0, dsk.slice, dsk.unit, dsk.part),
        0, 0, 0, VTOP(bootinfo));
```

1.6. loader 段

loader 也是一个 BTX 客，在里不作述。已有一部内容全面的手册 [loader\(8\)](#)，由 Mike Smith 写。比 loader 更底层的 BTX 的机理已在前面。

loader 的主要任务是引内核。当内核被装入内存后，即被 loader 用：

```
sys/boot/common/boot.c:
    /* 从 loader 中用内核中的 exec 程序 */
    module_formats[km-m_loader]-l_exec(km);
```

1.7. 内核初始化

我们来看一下接内核的命令。能助我了解 loader 内核的准位置。个位置就是内核真的入口点。

```
sys/conf/Makefile.i386:
ld -elf -Bdynamic -T /usr/src/sys/conf/ldscript.i386 -export-dynamic \
-dynamic-linker /red/herring -o kernel -X locore.o \
lots of kernel .o files
```

在一行中有一些有趣的西。首先，内核是一个 ELF 接二进制文件，可是连接器却是 /red/herring，一个莫有的文件。其次，看一下文件 sys/conf/ldscript.i386，可以理解内核 ld 的有一些。最前几行，字符串

```
sys/conf/ldscript.i386:
ENTRY(btext)
```

表示内核的入口点是符号 **btext**。个符号在 locore.s 中定：


```

sys/i386/i386/locore.s:
    .text
/*****
 *
 * This is where the bootblocks start us, set the ball rolling...
 * 入口
 */
NON_GPROF_ENTRY(btext)

```

首先将寄存器EFLAGS 一个 定 的 0x00000002， 然后初始化所有段寄存器:

```

sys/i386/i386/locore.s
/* 不要相信BIOS 出的EFLAGS */
    pushl    $PSL_KERNEL
    popfl

/*
 * 不要相信BIOS 出的%fs、%gs。相信引 程中 定的%cs、%ds、%es、%ss
 */
    mov %ds, %ax
    mov %ax, %fs
    mov %ax, %gs

```

btext 用例程 `recover_bootinfo()`, `identify_cpu()`, `create_pagetables()`。 些例程也定在locore.s之中。 些例程的功能如下：

<code>recover_bootinfo</code>	个例程分析由引 程序 送 内核的参数。引 内核有3 方式: 由loader引 (如前所述), 由老式磁 引 , 无 引 方式。 个函数决定引 方式, 并将 <code>struct bootinfo</code> 存 至内核内存。
<code>identify_cpu</code>	个函数 CPU 型, 将 果存放在 量 <code>_cpu</code> 中。
<code>create_pagetables</code>	个函数 分 表在内核内存空 部分 一 空 , 并填写一定内容

下 是 VME(如果CPU有 个功能):

```

    testl    $CUID_VME, R(_cpu_feature)
    jz 1f
    movl     %cr4, %eax
    orl $CR4_VME, %eax
    movl     %eax, %cr4

```

然后, 分 模式:

```

/* Now enable paging */
movl    R(_IdlePTD), %eax
movl    %eax,%cr3          /* load ptd addr into mmu */
movl    %cr0,%eax          /* get control word */
orl     $CR0_PE|CR0_PG,%eax /* enable paging */
movl    %eax,%cr0          /* and let's page NOW! */

```

由于分模式已，原先的地址址方式随即失效。随后三行代用来跳至虚地址：

```

pushl    $begin            /* jump to high virtualized address */
ret

/* 在跳至KERNBASE，那里是操作系统内核被接后真正的入口 */
begin:

```

函数`init386()`被用；随参数的是一个指针，指向第一个空闲物理。随后行`mi_startup()`。`init386`是一个与硬件系相关的初始化函数，`mi_startup()`是个与硬件系无关的函数（前‘mi’表示Machine Independent，不依赖于机器）。内核不再从`mi_startup()`里返回；用个函数后，内核完成引：

```

sys/i386/i386/locore.s:
movl    physfree, %esi
pushl    %esi              /* 送init386()的第一个参数 */
call    _init386          /* 置386芯片使之UNIX工作 */
call    _mi_startup /* 自配置硬件，挂接根文件系统，等 */
hlt      /* 不再返回到这里！ */

```

1.7.1. `init386()`

`init386()`定在 `sys/i386/i386/machdep.c`中，它Intel 386芯片行低初始化。loader已将CPU切至保模式。loader已建立了最早的任。



者注

个“任”都是与其它“任”相独立的行境。任之可以分切，并程/程的提供了必要基。于Intel 80x86任的描述，Intel公司于80386 CPU及后品的料，或者在清大学藏中用“80386”作所到的系方面的目。

在个任中，内核将工作。在其代前，我将理器保模式必完成的一系列准工作一并列出：

- 初始化内核的可整参数，些参数由引程序来
- 准GDT(全局描述符表)
- 准IDT(中断描述符表)
- 初始化系控制台
- 初始化DDB(内核的点器)，如果它被内核的
- 初始化TSS(任状段)

- 准LDT(局部描述符表)
- 建立proc0(0号进程, 即内核的进程)的pcb(进程控制块)

`init386()`首先初始化内核的可调整参数, 有些参数由引导程序带来。先置环境指针(environment pointer, `envp`)为0, 再用`init_param1()`。环境指针已由loader存放在`bootinfo`中:

```
sys/i386/i386/machdep.c:
    kern_envp = (caddr_t)bootinfo.bi_envp + KERNBASE;

/* 初始化基本可调整, 如hz等 */
init_param1();
```

`init_param1()`定义在 `sys/kern/subr_param.c`之中。这个文件里有一些sysctl, 有个函数, `init_param1()`和`init_param2()`。这个函数从`init386()`中调用:

```
sys/kern/subr_param.c
    hz = HZ;
    TUNABLE_INT_FETCH("kern.hz", hz);
```

`TUNABLE_typename_FETCH`用来取环境量的:

```
/usr/src/sys/sys/kernel.h
#define TUNABLE_INT_FETCH(path, var)    getenv_int((path), (var))
```

Sysctl `kern.hz`是系频率。同时, 有些sysctl被`init_param1()`定义: `kern.maxswzone`, `kern.maxbcache`, `kern.maxtsiz`, `kern.dfltsiz`, `kern.maxdsiz`, `kern.dflssiz`, `kern.maxssiz`, `kern.sgrowsiz`。

然后`init386()`准全局描述符表 (Global Descriptors Table, GDT)。在x86上每个任务都运行在自己的虚拟地址空间里, 每个空间由"段地址:偏移量"的数指定。举个例子, 当前将要由处理器执行的指令在 `CS:EIP`, 那条指令的虚拟地址就是"段地址CS" + `EIP`。为了方便, 段起始于虚拟地址0, 止于界限4G字节。所以, 在这个例子中, 指令的虚拟地址正是EIP的。段寄存器, 如CS、DS等是段符, 即全局描述符表中的索引(更精确的, 索引并非段符的全部, 而是段符中的INDEX部分)。



注意

对于80386, 段符有16位, INDEX部分是其中的高13位。

FreeBSD的全局描述符表每个CPU保存着15个段符:

```

sys/i386/i386/machdep.c:
union descriptor gdt[NGDT * MAXCPU];    /* 全局描述符表 */

sys/i386/include/segments.h:
/*
 * 全局描述符表(GDT)中的入口
 */
#define GNULL_SEL    0    /* 空描述符 */
#define GCODE_SEL    1    /* 内核代码描述符 */
#define GDATA_SEL    2    /* 内核数据描述符 */
#define GPRIV_SEL    3    /* 称多理(SMP)理器有数据 */
#define GPROC0_SEL    4    /* Task state process slot zero and up, 任状态程 */
#define GLDT_SEL    5    /* 个程的局部描述符表 */
#define GUSERLDT_SEL    6    /* 用自定义的局部描述符表 */
#define GTGATE_SEL    7    /* 程任切口 */
#define GBIOSLOWMEM_SEL    8    /* BIOS低端内存(必是第8个入口) */
#define GPANIC_SEL    9    /* 会致全系常中止工作的任状态 */
#define GBIOSCODE32_SEL    10    /* BIOS接口(32位代) */
#define GBIOSCODE16_SEL    11    /* BIOS接口(16位代) */
#define GBIOSDATA_SEL    12    /* BIOS接口(数据) */
#define GBIOSUTIL_SEL    13    /* BIOS接口(工具) */
#define GBIOSARGS_SEL    14    /* BIOS接口(自量, 参数) */

```

注意，这些#define并非符号本身，而只是符号中的INDEX域，因此它正是全局描述符表中的索引。例如，内核代码的符号(GCODE_SEL)的值为0x08。

下一个是初始化中断描述符表(Interrupt Descriptor Table, IDT)。该表在生件或硬件中断会被理器引用。例如，行系用，用用程序提交INT 0x80 指令。它是一个件中断，理器用索引0x80在中断描述符表中。它指向理个中断的例程。在个特定情形中，是内核的系用口。



者注

Intel 80386支持"用"，可以使得用程序只通一条call指令就
用内核中的例程。可是FreeBSD并未采用机制，也是因使用中断接口可免去
接的麻烦。外有一个附的好：在真Linux，当遇到
FreeBSD内核不支持的而又并非性的系用，内核只会示一些出信息，
使得程序能行；而不是在真正行程序之前的初始化程中就因接失而不允
程序行。

中断描述符表最多可以有256 (0x100)条。内核分配NIDT条的内存中断描述符表，里
NIDT=256，是最大：

```

sys/i386/i386/machdep.c:
static struct gate_descriptor idt0[NIDT];
struct gate_descriptor *idt = idt0[0]; /* 中断描述符表 */

```

个中断都被置一个合的中断理程序。系用INT 0x80也是如此：

```
sys/i386/i386/machdep.c:
    setidt(0x80, IDTVEC(int0x80_syscall),
           SDT_SYS386TGT, SEL_UPL, GSEL(GCODE_SEL, SEL_KPL));
```

所以当一个用`0x80`指令`int0x80`，全系的控制会调用函数`_Xint0x80_syscall`，这个函数在内核代码段中，将被以管理权限运行。

然后，控制台和DDB(调试器)被初始化:

```
sys/i386/i386/machdep.c:
    cninit();
/* 以下代码可能因未定义宏DDB而被跳过 */
#ifdef DDB
    kdb_init();
    if (boothowto & RB_KDB)
        Debugger("Boot flags requested debugger");
#endif
```

任务状态段(TSS)是一个x86保护模式中的数据段。当产生任何异常，任务状态段用来保存硬件寄存器信息。

局部描述符表(LDT)用来指向用户代码和数据。系统定义了几个段符，指向局部描述符表，它们是系统用户代码和用户数据段符:

```
/usr/include/machine/segments.h
#define LSYS5CALLS_SEL 0 /* Intel BCS控制要求的 */
#define LSYS5SIGR_SEL 1
#define L43BSDCALLS_SEL 2 /* 尚无 */
#define LUCODE_SEL 3
#define LSOL26CALLS_SEL 4 /* Solaris =2.6版系统调用 */
#define LUDATA_SEL 5
/* separate stack, es,fs,gs sels ? 分段的、es、fs、gs段符? */
/* #define LPOSIXCALLS_SEL 5 */ /* notyet, 尚无 */
#define LBSDICALLS_SEL 16 /* BSDI system call gate, BSDI系统调用 */
#define NLDT (LBSDICALLS_SEL + 1)
```

然后，`proc0`(0号进程，即内核所运行的进程)的进程控制块(Process Control Block) (`struct pcb`)被初始化。`proc0`是一个 `struct proc` 结构，描述了一个内核进程。内核运行，该进程是存在，所以该结构在内核中被定义全局变量:

```
sys/kern/kern_init.c:
    struct proc proc0;
```

`struct pcb`是`proc`结构的一部分，它定义在`/usr/include/machine/pcb.h`之中，内含所有i386硬件结构体的信息，如寄存器的。

1.7.2. mi_startup()

一个函数用冒泡排序算法，将所有系初始化对象，然后逐个用每个对象的入口：

```
sys/kern/init_main.c:
    for (sipp = sysinit; *sipp; sipp++) {

        /* ... 省略 ... */

        /* 调用函数 */
        ((*sipp)-func)((*sipp)-udata);
        /* ... 省略 ... */
    }
```

尽管sysinit框架已在《FreeBSD作者手册》中有所描述，我是在这里看一下其内部原理。

一个系初始化对象(sysinit对象)通常用宏建立。我以announce sysinit对象为例。一个对象打印版信息：

```
sys/kern/init_main.c:
static void
print_caddr_t(void *data __unused)
{
    printf("%s", (char *)data);
}
SYSINIT(announce, SI_SUB_COPYRIGHT, SI_ORDER_FIRST, print_caddr_t, copyright)
```

一个对象的子系是SI_SUB_COPYRIGHT(0x0800001)，数好排在SI_SUB_CONSOLE(0x0800000)后面。所以，版信息将在控制台初始化之后就被很早的打印出来。

我看一看宏SYSINIT()到底做了些什。它展开成宏C_SYSINIT()。宏C_SYSINIT()然后展开成一个静struct sysinit。里申明里用了宏DATA_SET：

```
/usr/include/sys/kernel.h:
#define C_SYSINIT(uniquifier, subsystem, order, func, ident) \
    static struct sysinit uniquifier ## _sys_init = { \ subsystem, \
    order, \ func, \ ident \ }; \ DATA_SET(sysinit_set,uniquifier ##
    _sys_init);

#define SYSINIT(uniquifier, subsystem, order, func, ident) \
    C_SYSINIT(uniquifier, subsystem, order, \
    (sysinit_cfunc_t)(sysinit_nfunc_t)func, (void *)ident)
```

宏DATA_SET()展开成MAKE_SET()，宏MAKE_SET()指向所有含的sysinit幻数：

```

/usr/include/linker_set.h
#define MAKE_SET(set, sym) \
    static void const * const __set_##set##_sym_##sym = sym; \
    __asm(".section .set." #set ",\\"aw\\"); \
    __asm(".long " #sym); \
    __asm(".previous")
#endif
#define TEXT_SET(set, sym) MAKE_SET(set, sym)
#define DATA_SET(set, sym) MAKE_SET(set, sym)

```

回到我之前的例子中，宏的展开过程，将会产生如下声明：

```

static struct sysinit announce_sys_init = {
    SI_SUB_COPYRIGHT,
    SI_ORDER_FIRST,
    (sysinit_cfunc_t)(sysinit_nfunc_t) print_caddr_t,
    (void *) copyright
};

static void const *const __set_sysinit_set_sym_announce_sys_init =
    announce_sys_init;
__asm(".section .set.sysinit_set" ",\\"aw\\");
__asm(".long " "announce_sys_init");
__asm(".previous");

```

第一个 `__asm` 指令在内核可执行文件中建立一个 ELF 段(section)。它产生在内核链接的时候。它将被命令 `__set.sysinit_set`。它的内容是一个 32 位——`announce_sys_init` 的地址，它正是第二个 `__asm` 指令所定义的。第三个 `__asm` 指令的结束。如果前面有名字相同的定义语句，它的内容(那个 32 位)将被填充到已存在的段里，它就制造出了一个 32 位指数。

用 `objdump` 察看一个内核二进制文件，也会注意到里面有几个小的：

```

% objdump -h /kernel
 7 .set.cons_set 00000014 c03164c0 c03164c0 002154c0 2**2
    CONTENTS, ALLOC, LOAD, DATA
 8 .set.kbdriver_set 00000010 c03164d4 c03164d4 002154d4 2**2
    CONTENTS, ALLOC, LOAD, DATA
 9 .set.scrndr_set 00000024 c03164e4 c03164e4 002154e4 2**2
    CONTENTS, ALLOC, LOAD, DATA
10 .set.scterm_set 0000000c c0316508 c0316508 00215508 2**2
    CONTENTS, ALLOC, LOAD, DATA
11 .set.sysctl_set 00000097c c0316514 c0316514 00215514 2**2
    CONTENTS, ALLOC, LOAD, DATA
12 .set.sysinit_set 00000664 c0316e90 c0316e90 00215e90 2**2
    CONTENTS, ALLOC, LOAD, DATA

```

第一屏信息显示表明 `__set.sysinit_set` 有 0x664 字的大小，所以 `0x664/sizeof(void *)` 个 `sysinit` 对象被

入了内核。其它，如`.set.sysctl_set`表示其它接口集合。

通常定一个型`struct linker_set`的量，`.set.sysinit_set`将被"收集"到那个量里：

```
sys/kern/init_main.c:
    extern struct linker_set sysinit_set; /* XXX */
```

`struct linker_set`定如下：

```
/usr/include/linker_set.h:
struct linker_set {
    int ls_length;
    void *ls_items[1];    /* ls_length个的数，以NULL尾 */
};
```



者注

上是，用C言体`linker_set`来表那个ELF。

第一是`sysinit`象的数量，第二是一个以NULL尾的数，数中是指向那些象的指。

回到`mi_startup()`的，我清楚了`sysinit`象是如何被起来的。函数`mi_startup()`将它排序，并用一个象。最后一个象是系度器：

```
/usr/include/sys/kernel.h:
enum sysinit_sub_id {
    SI_SUB_DUMMY        = 0x0000000,    /* 不被行，供接口使用 */
    SI_SUB_DONE         = 0x0000001,    /* 已被理 */
    SI_SUB_CONSOLE      = 0x0800000,    /* 控制台 */
    SI_SUB_COPYRIGHT    = 0x0800001,    /* 最早使用控制台的象 */
    ...
    SI_SUB_RUN_SCHEDULER = 0xffffffff /* 度器:不返回 */
};
```

系度器`sysinit`象定在文件`sys/vm/vm_glue.c`中，个象的入口点是`scheduler()`。个函数上是个无限循，它表示那个程(PID)的程——`swapper`程。前面提到的`proc0`正是用来描述个程。

第一个用程是`_init_`，由`sysinit`象`init`建立：


```

sys/kern/init_main.c:
static void
create_init(const void *udata __unused)
{
    int error;
    int s;

    s = splhigh();
    error = fork1(proc0, RFFDG | RFPROC, initproc);
    if (error)
        panic("cannot fork init: %d\n", error);
    initproc-p_flag |= P_INMEM | P_SYSTEM;
    cpu_set_fork_handler(initproc, start_init, NULL);
    remrunqueue(initproc);
    splx(s);
}
SYSINIT(init,SI_SUB_CREATE_INIT, SI_ORDER_FIRST, create_init, NULL)

```

`create_init()` 通常用 `fork1()` 分配一个新的进程，但并不将其立即可执行。当一个新的进程被调度器调度执行，`start_init()` 将会被调用。那个函数定义在 `init_main.c` 中。它安装并行二进制制代 `init`，先调用 `/sbin/init`，然后是 `/sbin/oinit`，`/sbin/init.bak`，最后是 `/stand/sysinstall`：

```

sys/kern/init_main.c:
static char init_path[MAXPATHLEN] =
#ifdef INIT_PATH
    __XSTRING(INIT_PATH);
#else
    "/sbin/init:/sbin/oinit:/sbin/init.bak:/stand/sysinstall";
#endif

```

Chapter 2. 内核中的

一章由 *FreeBSD SMP Next Generation Project* 编写。 将和建送 *FreeBSD* 称多理 (SMP) 件列表。

这篇文提述了在FreeBSD内核中的，使得有效的多理成可能。 可以用几方式得。数据 可以用mutex或lockmgr(9)保。 于数不多的若干个量，假如是使用原子操作它， 些量就可以得到保。



者注

本章内容，不足以出"mutex" 和"共享互斥"的区别。似乎它的功能有重之，前者比后者的功能更多。它似乎都是lockmgr(9)的子集。

2.1. Mutex

Mutex就是一用来解决共享/排它矛盾的。 一个mutex在一个刻只可以被一个体有。如果一个体要得已被有的mutex，就会入等待，直到个mutex被放。在FreeBSD内核中，mutex被程所。

Mutex可以被的索要，但是mutex一般只被一个体有短的一段， 因此一个体不能在持有mutex睡眠。如果需要在持有mutex睡眠，可使用一个 lockmgr(9) 的。

个mutex有几个令人感兴趣的属性：

量名

在内核源代码中struct mtx量的名字

名

由函数mtx_init指派的mutex的名字。 个名字示在KTR跟踪消息和witness出与警告信息里。 个名字用于区分在witness代中的各个mutex

型

Mutex的型，用志MTX_表示。 个志的意在mutex(9)有所描述。

MTX_DEF

一个睡眠mutex

MTX_SPIN

一个循mutex

MTX_RECURSE

个mutex允

保象

个入口所要保的数据列表或数据成列表。 于数据成，将按照 名.成名的形式命名。

依函数

当mutex被持有才可以被用的函数

表 1. Mutex列表

变量名	变量名	类型	保护对象	依赖函数
sched_lock	"sched lock"(调度器)	MTX_SPIN MTX_RECURSE	_gmonparam, cnt.v_swch, 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, psent, slpq, 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

变量名	别名	类型	保护对象	依赖函数
vm86pcb_lock	"vm86pcb lock"(虚拟8086模式进程控制)	MTX_DEF	vm86pcb	vm86_bioscall
Giant	"Giant"(巨)	MTX_DEF MTX_RECURSE	几乎可以是任何东西	很多
callout_lock	"callout lock"(延迟调用)	MTX_SPIN MTX_RECURSE	callfree, callwheel, nextsoftcheck, proc.p_itcallout, proc.p_slpcallout, softticks, ticks	

2.2. 共享互斥

这些提供基本的读/写类型的功能，可以被一个正在睡眠的进程持有。它在被放到lockmgr(9)之中。

表 2. 共享互斥列表

变量名	保护对象
allproc_lock	allproc zombproc pidhashtbl proc.p_list proc.p_hash nextpid
proctree_lock	proc.p_children proc.p_sibling

2.3. 原子保护量

原子保护量并非由一个在的锁的特殊量，而是：有些量的所有数据都要使用特殊的原子操作(atomic(9))。尽管其它的基本同步机制(例如mutex)就是用原子保护量实现的，但是很少有量直接使用这种方式。

- mtx.mtx_lock

Chapter 3. 内核对象

内核对象，也就是*Kobj*，内核提供了一种面向对象的C语言编程方式。被操作的数据也承担操作它的方法。这使得在不破坏二进制兼容性的前提下，某一个接口能完成/相似的操作。

3.1. 对象

对象

数据集-数据-数据分配的集合

方法

某一操作-函数

接口

一或多方法

接口

一或多方法的一个子集

3.2. Kobj的工作流程



作者注

一小段段落中原作者的用词有些含混，请参考我在括号中的注释。

Kobj工作，生成方法的描述。每个描述有一个唯一的ID和一个缺省函数。某个描述的地址被用来在一个对象的方法表里唯一的方法。

建立一个对象，就是要建立一个方法表，并将表指向一个或多个函数(方法)；有些函数(方法)都有方法描述。使用前，它要被初始化。它要指向分配一些内存。在方法表中的每个方法描述都会被指派一个唯一的ID，除非它已被其它引用它的ID在别处指派了ID。于是要被使用的方法，都会由脚本生成一个函数(方法函数)，以解析外来参数，并在被调用时输出方法描述的地址。被生成的函数(方法函数)凭着那个方法描述的唯一ID按Hash的方法指向对象的内存。如果这个方法不在内存中，函数会使用它的方法表。如果这个方法被找到了，它里的相应函数(也就是某个方法的代码)就会被使用。否则，这个方法描述的缺省函数将被使用。

这些过程可被表示如下：

对象-内存-对象

3.3. 使用Kobj

3.3.1. 对象

```
struct kobj_method
```

3.3.2. 函数

```
void kobj_class_compile(kobj_class_t cls);
void kobj_class_compile_static(kobj_class_t cls, kobj_ops_t ops);
void kobj_class_free(kobj_class_t cls);
kobj_t kobj_create(kobj_class_t cls, struct malloc_type *mtype, int mflags);
void kobj_init(kobj_t obj, kobj_class_t cls);
void kobj_delete(kobj_t obj, struct malloc_type *mtype);
```

3.3.3. 宏

```
KOBJ_CLASS_FIELDS
KOBJ_FIELDS
DEFINE_CLASS(name, methods, size)
KOBJMETHOD(NAME, FUNC)
```

3.3.4. 文件

```
sys/param.h
sys/kobj.h
```

3.3.5. 建立一个接口的模板

使用Kobj的第一步是建立一个接口。建立接口包括建立模板的工作。

建立模板可用脚本src/sys/kern/makeobjops.pl完成，它会生成申明方法的文件和代码，脚本会生成方法函数。

在模板中如下宏会被使用: `#include`, `INTERFACE`, `CODE`, `METHOD`, `STATICMETHOD`, 和 `DEFAULT`.

`#include`句的整行内容将被一字不差的复制到被生成的代码文件的头部。

例如:

```
#include sys/foo.h
```

`INTERFACE`用来定义接口名。这个名字将与方法名接合在一起，形成 `[interface name]_[method name]`。用法是: `INTERFACE [接口名];`

例如:

```
INTERFACE foo;
```

CODE会将它的参数一字不差的复制到代码文件中。方法是CODE { [任何代码] };

例如:

```
CODE {
    struct foo * foo_alloc_null(struct bar *)
    {
        return NULL;
    }
};
```

METHOD用来描述一个方法。方法是: METHOD [返回类型] [方法名] { [对象 [, 参数若干]] };

例如:

```
METHOD int bar {
    struct object *;
    struct foo *;
    struct bar;
};
```

DEFAULT跟在METHOD之后, 是METHOD的补充。它给方法补充上缺省函数。方法是: METHOD [返回类型] [方法名] { [对象; [其它参数]] }DEFAULT [缺省函数];

例如:

```
METHOD int bar {
    struct object *;
    struct foo *;
    int bar;
} DEFAULT foo_hack;
```

STATICMETHOD似METHOD。由于一个Kobj对象, 一般其内部都有一些Kobj有的数据。METHOD定义的方法就假设有些数据位于对象内部; 假如对象内部没有某些数据, 些方法对这个对象的调用就可能出错。而STATICMETHOD定义的对象可以不受这个限制: 描述出的方法, 其操作的数据不由这个对象的某个对象例出, 而是全都由调用这个方法的操作数(者注:即参数)出。也由于在某个对象的方法表之外调用这个方法有用。



者注

一段的言与原文相比调整很大。 静方法是不依赖于对象例的方法。 参看C++中的"静态函数"的概念。

其它完整的例子:

```
src/sys/kern/bus_if.m
src/sys/kern/device_if.m
```

3.3.6. 建立一个类

使用Kobj的第二步是建立一个类。一个类的类有名字、方法表；假如使用了Kobj的"对象管理工具"(Object Handling Facilities)，类中包含对象的大小。建立类使用宏`DEFINE_CLASS()`。建立方法表，建立一个`kobj_method_t`数组，用`NULL`结尾。一个非`NULL`可用宏`KOBJMETHOD()`建立。

例如:

```
DEFINE_CLASS(fooClass, fooMethods, sizeof(struct fooData));

kobj_method_t fooMethods[] = {
    KOBJMETHOD(bar_doo, foo_doo),
    KOBJMETHOD(bar_foo, foo_foo),
    { NULL, NULL}
};
```

类被"编译"。根据类被初始化系数的状态，将要用到一个静态分配的内存和"操作数表"(ops table, 译者注：即"参数表")。有些操作可通过声明一个类体 `struct kobj_ops` 并使用 `kobj_class_compile_static()`，或是只使用`kobj_class_compile()`来完成。

3.3.7. 建立一个对象

使用Kobj的第三步是定义对象。Kobj对象建立程序假定Kobj类有数据在一个对象的内部。如果不是如此，类当先自行分配对象，再使用`kobj_init()`初始化对象中的Kobj类有数据；其类可以使用`kobj_create()`分配对象，并自动初始化对象中的Kobj类有内容。`kobj_init()`也可以用来改变一个对象所使用的类。

将Kobj的数据集成到对象中要使用宏`KOBJ_FIELDS`。

例如

```
struct foo_data {
    KOBJ_FIELDS;
    foo_foo;
    foo_bar;
};
```

3.3.8. 类用方法

使用Kobj的最后一步就是通过生成的函数类用对象类中的方法。类用类，接口名与方法名用'_'接合，而且全部使用大写字母。

例如，接口名`foo`，方法`bar`，类用就是:


```
[返回 = ] FOO_BAR(对象 [, 其它参数]);
```

3.3.9. 善后处理

当一个用 `kobj_create()` 不再需要被使用，可调用 `kobj_delete()`。当一个不再需要被使用，可调用 `kobj_class_free()`。

Chapter 4. Jail子系

在大多数UNIX®系中，用root是万能的。也就加了多危。如果一个攻者得了一个系中的root，就可以在他的指尖掌握系中所有的功能。在FreeBSD里，有一些sysctl削弱了root的限，就可以将攻者造成的害小到最低限度。些安全功能中，有一叫安全。一在FreeBSD 4.0及以后版本中提供的的安全功能，就是jail(8)。Jail将一个行境的文件根切到某一特定位置，并且境中又分生成的程做出限制。例如，一个被禁的程不能影个jail之外的程、不能使用一些特定的系用，也就不能主计算机造成破坏。



者注

英文"jail"的中文意思是"囚禁、禁"。

Jail已成一新型的安全模型。人可以在jail中行各可能很脆弱的服器程序，如Apache、BIND和sendmail。一来，即使有攻者取得了jail中的root，最多人眉，而不会使人慌失措。本文主要注jail的内部原理(源代)。如果正在置Jail的指南性文，我建我的篇文章，表在Sys Admin Magazine, May 2001, 《Securing FreeBSD using Jail》。

4.1. Jail的系

Jail由部分成：用程序，也就是jail(8)；有在内核中Jail的代：jail(2) 系用和相的束。我将用程序和jail在内核中的原理。

4.1.1. 用代

Jail的用源代在/usr/src/usr.sbin/jail，由一个文件jail.c成。个程序有些参数：jail的路径，主机名，IP地址，有需要行的命令。

4.1.1.1. 数据

在jail.c中，我将最先注解的是一个重要体 struct jail j;的声明，个型的声明包含在/usr/include/sys/jail.h之中。

jail的定义是：

```
/usr/include/sys/jail.h:

struct jail {
    u_int32_t    version;
    char        *path;
    char        *hostname;
    u_int32_t    ip_number;
};
```

正如所，送命令jail(8)的个参数都在里有的一。事上，当命令jail(8)被行，些参数才由命令行真正入：

```

/usr/src/usr.sbin/jail.c
char path[PATH_MAX];
...
if(realpath(argv[0], path) == NULL)
    err(1, "realpath: %s", argv[0]);
if (chdir(path) != 0)
    err(1, "chdir: %s", path);
memset(j, 0, sizeof(j));
j.version = 0;
j.path = path;
j.hostname = argv[1];

```

4.1.1.2. 网口

网口 `jail(8)` 的参数中有一个是IP地址。它是在网口上网口 `jail` 的地址。网口 `jail(8)` 将IP地址转换成网口字节序，并存入 `j` (网口 `jail` 型的结构体)。

```

/usr/src/usr.sbin/jail/jail.c:
struct in_addr in;
...
if (inet_aton(argv[2], in) == 0)
    errx(1, "Could not make sense of ip-number: %s", argv[2]);
j.ip_number = ntohl(in.s_addr);

```

函数 `inet_aton(3)` "将指定的字符串解释成一个Internet地址，并将其存到指定的结构体中"。网口 `inet_aton(3)` 定义了结构体 `in`，之后 `in` 中的内容再用 `ntohl(3)` 转换成主机字节序，并置入网口 `jail` 结构体的 `ip_number` 成员。

4.1.1.3. 囚禁进程

最后，用网口程序囚禁进程。网口在 `Jail` 自身网口成了一个被囚禁的进程，并使用 `execv(3)` 网口行用网口指定的命令。

```

/usr/src/usr.sbin/jail/jail.c
i = jail(j);
...
if (execv(argv[3], argv + 3) != 0)
    err(1, "execv: %s", argv[3]);

```

正如网口所网口，函数 `jail()` 被网口用，参数是结构体 `jail` 中被填入数据网口，而如前所述，网口些数据网口又来自网口 `jail(8)` 的命令行参数。最后，网口行了用网口指定的命令。下面我将网口始网口 `jail` 在内核中的网口。

4.1.2. 相网口的内核源代码网口

网口在我网口来看文件 `/usr/src/sys/kern/kern_jail.c`。在网口里定义了网口 `jail(2)` 的系网口用、相网口的 `sysctl`网口，网口有网口网口函数。

4.1.2.1. sysctl网口

在 `kern_jail.c` 里定义了如下 `sysctl`网口：

```

/usr/src/sys/kern/kern_jail.c:

int    jail_set_hostname_allowed = 1;
SYSCTL_INT(_security_jail, OID_AUTO, set_hostname_allowed, CTLFLAG_RW,
    jail_set_hostname_allowed, 0,
    "Processes in jail can set their hostnames");
/* Jail中的进程可设定自身的主机名 */

int    jail_socket_unixiproute_only = 1;
SYSCTL_INT(_security_jail, OID_AUTO, socket_unixiproute_only, CTLFLAG_RW,
    jail_socket_unixiproute_only, 0,
    "Processes in jail are limited to creating UNIX/IPv4/route sockets only");
/* Jail中的进程被限制只能建立UNIX套接字、IPv4套接字、路由套接字 */

int    jail_sysvipc_allowed = 0;
SYSCTL_INT(_security_jail, OID_AUTO, sysvipc_allowed, CTLFLAG_RW,
    jail_sysvipc_allowed, 0,
    "Processes in jail can use System V IPC primitives");
/* Jail中的进程可以使用System V进程通信原 */

static int jail_enforce_statfs = 2;
SYSCTL_INT(_security_jail, OID_AUTO, enforce_statfs, CTLFLAG_RW,
    jail_enforce_statfs, 0,
    "Processes in jail cannot see all mounted file systems");
/* jail 中的进程看系中挂载的文件系受到何限制 */

int    jail_allow_raw_sockets = 0;
SYSCTL_INT(_security_jail, OID_AUTO, allow_raw_sockets, CTLFLAG_RW,
    jail_allow_raw_sockets, 0,
    "Prison root can create raw sockets");
/* jail 中的 root 用是否可以建 raw socket */

int    jail_chflags_allowed = 0;
SYSCTL_INT(_security_jail, OID_AUTO, chflags_allowed, CTLFLAG_RW,
    jail_chflags_allowed, 0,
    "Processes in jail can alter system file flags");
/* jail 中的进程是否可以修改系文件 */

int    jail_mount_allowed = 0;
SYSCTL_INT(_security_jail, OID_AUTO, mount_allowed, CTLFLAG_RW,
    jail_mount_allowed, 0,
    "Processes in jail can mount/unmount jail-friendly file systems");
/* jail 中的进程是否可以挂或卸jail友好的文件系统 */

```

一些sysctl中的一个都可以用命令**sysctl(8)**。在整个内核中，
。例如，上述第一个sysctl的名字是 **security.jail.set_hostname_allowed**。

一些sysctl按名称

4.1.2.2. jail(2)系调用

像所有的系调用一样，系调用jail(2)有四个参数，`struct thread *td`和`struct jail_args *uap`。`td`是一个指向`thread`体的指针，它用于描述调用jail(2)的进程。在上下文中，`uap`指向一个结构体，该结构体中包含了一个指向从调用jail.c送来的jail体的指针。在前面我描述过程序，已看到一个jail体被作为参数送给系调用jail(2)。

```
/usr/src/sys/kern/kern_jail.c:
/*
 * struct jail_args {
 *     struct jail *jail;
 * };
 */
int
jail(struct thread *td, struct jail_args *uap)
```

于是uap-jail可以用于被jail(2)的jail体。然后，jail(2)使用copyin(9)将jail体复制到内核内存空间中。copyin(9)需要三个参数：要复制到内核内存空间的数据的地址uap-jail，在内核内存空间存放数据的j，以及数据的大小。uap-jail指向的jail体被复制到内核内存空间，并被存放在一个jail体j里。

```
/usr/src/sys/kern/kern_jail.c:
error = copyin(uap-jail, j, sizeof(j));
```

在jail.h中定义了一个重要的结构体prison。结构体prison只被用在内核空间中。下面是prison结构的定义。

```
/usr/include/sys/jail.h:
struct prison {
    LIST_ENTRY(prison) pr_list;           /* (a) all prisons */
    int pr_id;                             /* (c) prison id */
    int pr_ref;                             /* (p) refcount */
    char pr_path[MAXPATHLEN];              /* (c) chroot path */
    struct vnode *pr_root;                  /* (c) vnode to rdir */
    char pr_host[MAXHOSTNAMELEN];          /* (p) jail hostname */
    u_int32_t pr_ip;                        /* (c) ip addr host */
    void *pr_linux;                        /* (p) linux abi */
    int pr_securelevel;                     /* (p) securelevel */
    struct task pr_task;                    /* (d) destroy task */
    struct mtx pr_mtx;
    void **pr_slots;                       /* (p) additional data */
};
```

然后，系调用jail(2)为一个prison结构体分配一段内存，并在jail和prison结构体之间复制数据。

```

/usr/src/sys/kern/kern_jail.c:
MALLOC(pr, struct prison *, sizeof(*pr), M_PRISON, M_WAITOK | M_ZERO);
...
error = copyinstr(j.path, pr-pr_path, sizeof(pr-pr_path), 0);
if (error)
    goto e_killmtx;
...
error = copyinstr(j.hostname, pr-pr_host, sizeof(pr-pr_host), 0);
if (error)
    goto e_dropvref;
pr-pr_ip = j.ip_number;

```

下面，我们将另外一个重要的系用 `jail_attach(2)`，它了将程禁的功能。

```

/usr/src/sys/kern/kern_jail.c
/*
 * struct jail_attach_args {
 *     int jid;
 * };
 */
int
jail_attach(struct thread *td, struct jail_attach_args *uap)

```

个系用做出一些可以用于区分被禁和未被禁的程的改。
首先要理解一些背景信息。

要理解 `jail_attach(2)` 我做了什，我

在FreeBSD中，个内核可的程是通其 `thread` 体来，同，程都由它自己的 `proc` 体描述。
可以在 `/usr/include/sys/proc.h` 中到 `thread` 和 `proc` 体的定。例如，在任何系用中，参数 `td`
是个指向用程的 `thread` 体的指，正如前面所的那。 `td` 所指向的 `thread` 体中的 `td_proc` 成
是一个指，个指指向 `td` 所表示的程所属程的 `proc` 体。体 `proc` 包含的成可以描述所有者的身
(`p_ucred`)，程源限制(`p_limit`)，等等。在由 `proc` 体的 `p_ucred` 成所指向的 `ucred` 体的定中，
有一个指向 `prison` 体的指(`cr_prison`)。

```

/usr/include/sys/proc.h:
struct thread {
    ...
    struct proc *td_proc;
    ...
};
struct proc {
    ...
    struct ucred *p_ucred;
    ...
};
/usr/include/sys/ucred.h
struct ucred {
    ...
    struct prison *cr_prison;
    ...
};

```

在kern_jail.c中，函数jail()以jtid指定的jail ID调用函数jail_attach()。随后jail_attach()调用函数change_root()以改变进程的根目录。接下来，jail_attach()创建一个新的ucred结构体，并在成功地将prison结构体接到该ucred结构体后，将该ucred结构体接到进程上。从此开始，该进程就会被jail禁用的。当我以新创建的这个ucred结构体参数调用内核路径jailed()，它将返回1来表明该进程是和jail相关的。在jail中又分出来的所有进程的公共祖先进程就是那个调用了jail(2)的进程，因为正是它调用了jail(2)系统调用。当一个程序通过execve(2)而被执行，它将从其父进程的ucred结构体继承被禁用的属性，因而它也会有一个被禁用的ucred结构体。

```

/usr/src/sys/kern/kern_jail.c
int
jail(struct thread *td, struct jail_args *uap)
{
    ...
    struct jail_attach_args jaa;
    ...
    error = jail_attach(td, jaa);
    if (error)
        goto e_dropprref;
    ...
}

int
jail_attach(struct thread *td, struct jail_attach_args *uap)
{
    struct proc *p;
    struct ucred *newcred, *oldcred;
    struct prison *pr;
    ...
    p = td->td_proc;
    ...
    pr = prison_find(uap->jid);
    ...
    change_root(pr->pr_root, td);
    ...
    newcred->cr_prison = pr;
    p->p_ucred = newcred;
    ...
}

```

当一个进程被从其父进程叉分来的时候，系统用**fork(2)**将用**crhold()**来继承其身凭证。因此，很自然的就保持了子进程的凭证于其父进程一致，所以子进程也是被囚禁的。

```

/usr/src/sys/kern/kern_fork.c:
p2->p_ucred = crhold(td->td_ucred);
...
td2->td_ucred = crhold(p2->p_ucred);

```

4.2. 系统被囚禁程序的限制

在整个内核中，有一系列被囚禁程序的约束措施。通常，这些约束只对被囚禁的程序有效。如果某些程序突破这些约束，相应的函数将出错返回。例如：

```

if (jailed(td->td_ucred))
    return EPERM;

```


4.2.1. SysV 程通信(IPC)

System V 程通信 (IPC) 是通消息的。 个程都可以向其它程送消息， 告方做什。 理消息的函数是： [msgctl\(3\)](#)、[msgget\(3\)](#)、[msgsnd\(3\)](#) 和 [msgrcv\(3\)](#)。前面已提到，一些 `sysctl` 可以影 `jail` 的行， 其中有一个是 `security.jail.sysvipc_allowed`。在大多数系上， 个 `sysctl` 会成 0。如果将它置 1， 会完全失去 `jail` 的意： 因那在 `jail` 中特程就可以影被禁的境外的程了。消息与信号的区是：消息由一个信号号成。

`/usr/src/sys/kern/sysv_msg.c`:

- `msgget(key, msgflg)`: `msgget` 返回(也可能建)一个消息描述符， 以指派一个在其它函数中使用的消息列。
- `msgctl(msgid, cmd, buf)`: 通个函数， 一个程可以个消息描述符的状。
- `msgsnd(msgid, msgp, msgsz, msgflg)`: `msgsnd` 向一个程送一条消息。
- `msgrcv(msgid, msgp, msgsz, msgtyp, msgflg)`: 程用个函数接收消息。

在些函数系的用的代中， 都有个条件判断：

```
/usr/src/sys/kern/sysv_msg.c:
if (!jail_sysvipc_allowed & jailed(td-td_ucred))
    return (ENOSYS);
```

信号量系用使得程可以通一系列原子操作同。 信号量程定源提供了又一途径。 然而， 程将正在被使用的信号量入等待状， 一直休眠到源被放。 在 `jail` 中如下的信号量系用将会失效: [semget\(2\)](#), [semctl\(2\)](#) 和 [semop\(2\)](#)。

`/usr/src/sys/kern/sysv_sem.c`:

- `semctl(semid, num, cmd, ...)`: `semctl` 在信号量列中用 `semid` 的信号量行 `cmd` 指定的命令。
- `semget(key, nsems, flag)`: `semget` 建立一个于 `key` 的信号量数。
- 参数 `key` 和 `flag` 与他 `msgget()` 的意相同。
- `setop(semid, array, nops)`: `semop` 用 `semid` 的信号量完成一由 `array` 所指定的操作。

System V IPC 使程可以共享内存。 程之可以通它虚地址空的共享部分以及相数据写操作直接通。 些系用在被禁的境中将会失效: [shmdt\(2\)](#)、[shmat\(2\)](#)、[shmctl\(2\)](#) 和 [shmget\(2\)](#)

`/usr/src/sys/kern/sysv_shm.c`:

- `shmctl(shmid, cmd, buf)`: `shmctl` 用 `shmid` 的共享内存区域做各各的控制。
- `shmget(key, size, flag)`: `shmget` 建立/打 `size` 字的共享内存区域。
- `shmat(shmid, addr, flag)`: `shmat` 将 `shmid` 的共享内存区域指派到程的地址空里。
- `shmdt(addr)`: `shmdt` 取消共享内存区域的地址指派。

4.2.2. 套接字

Jail以一种特殊的方式处理`socket(2)`系列用和相关的低层套接字函数。为了决定一个套接字是否允许被建立，它先通过`sysctl` `security.jail.socket_unixiproute_only`是否被置1。如果被置1，套接字建立时只能指定某些族：`PF_LOCAL`, `PF_INET`, `PF_ROUTE`。否则，`socket(2)`将会返回出错。

```
/usr/src/sys/kern/uipc_socket.c:
int
socreate(int dom, struct socket **aso, int type, int proto,
         struct ucred *cred, struct thread *td)
{
    struct protosw *prp;
    ...
    if (jailed(cred) && jail_socket_unixiproute_only
        && prp->pr_domain-dom_family != PF_LOCAL
        && prp->pr_domain-dom_family != PF_INET
        && prp->pr_domain-dom_family != PF_ROUTE) {
        return (EPROTONOSUPPORT);
    }
    ...
}
```

4.2.3. Berkeley包过滤器

Berkeley包过滤器提供了一个与内核无的，直接通向数据路径的低层接口。在BPF是否可以在禁的境中被使用是通过`devfs(8)`来控制的。

4.2.4. 网络

网络TCP, UDP, IP和ICMP很常见。IP和ICMP属于同一层次：第二层，网络。当参数`nam`被置，有一些限制措施会防止被囚禁的程序绑定到一些网络接口上。`nam`是一个指向`sockaddr`结构的指针，描述可以绑定到地址。一个更切切的定义：`sockaddr`是一个模板，包含了地址的符号和地址的度。在函数`in_pcbbind_setup()`中`sin`是一个指向`sockaddr_in`结构的指针，这个结构包含了套接字可以绑定的端口、地址、度、族。这就禁止了在jail中的进程指定不属于该进程所存在于的jail的IP地址。

```

/usr/src/sys/kern/netinet/in_pcb.c:
int
in_pcbbind_setup(struct inpcb *inp, struct sockaddr *nam, in_addr_t *laddrp,
    u_short *lportp, struct ucred *cred)
{
    ...
    struct sockaddr_in *sin;
    ...
    if (nam) {
        sin = (struct sockaddr_in *)nam;
        ...
        if (sin->sin_addr.s_addr != INADDR_ANY)
            if (prison_ip(cred, 0, sin->sin_addr.s_addr))
                return(EINVAL);
        ...
        if (lport) {
            ...
            if (prison_ip(cred, 0, sin->sin_addr.s_addr))
                return (EADDRNOTAVAIL);
            ...
        }
    }
    if (lport == 0) {
        ...
        if (laddr.s_addr != INADDR_ANY)
            if (prison_ip(cred, 0, laddr.s_addr))
                return (EINVAL);
        ...
    }
    ...
    if (prison_ip(cred, 0, laddr.s_addr))
        return (EINVAL);
    ...
}

```

你也想知道函数`prison_ip()`做什么。`prison_ip()`有三个参数，一个指向身凭的指(用`cred`表示)，一些志和一个IP地址。当个IP地址不属于个jail，返回1；否返回0。正如从代中看的，如果，那个IP地址不属于个jail，就不再允向个网地址定。

```

/usr/src/sys/kern/kern_jail.c:
int
prison_ip(struct ucred *cred, int flag, u_int32_t *ip)
{
    u_int32_t tmp;

    if (!jailed(cred))
        return (0);
    if (flag)
        tmp = *ip;
    else
        tmp = ntohl(*ip);
    if (tmp == INADDR_ANY) {
        if (flag)
            *ip = cred-cr_prison-pr_ip;
        else
            *ip = htonl(cred-cr_prison-pr_ip);
        return (0);
    }
    if (tmp == INADDR_LOOPBACK) {
        if (flag)
            *ip = cred-cr_prison-pr_ip;
        else
            *ip = htonl(cred-cr_prison-pr_ip);
        return (0);
    }
    if (cred-cr_prison-pr_ip != tmp)
        return (1);
    return (0);
}

```

4.2.5. 文件系

如果完全大于0，即便是jail里面的root，也不允许在Jail中取消或更改文件志，如"不可修改"、"只可添加"、"不可删除"志。

```

/usr/src/sys/ufs/ufs/ufs_vnops.c:
static int
ufs_setattr(ap)
{
    ...
    if (!priv_check_cred(cred, PRIV_VFS_SYSFLAGS, 0)) {
        if (ip-i_flags
            (SF_NOUNLINK | SF_IMMUTABLE | SF_APPEND)) {
            error = securelevel_gt(cred, 0);
            if (error)
                return (error);
        }
        ...
    }
}

/usr/src/sys/kern/kern_priv.c
int
priv_check_cred(struct ucred *cred, int priv, int flags)
{
    ...
    error = prison_priv_check(cred, priv);
    if (error)
        return (error);
    ...
}

/usr/src/sys/kern/kern_jail.c
int
prison_priv_check(struct ucred *cred, int priv)
{
    ...
    switch (priv) {
        ...
        case PRIV_VFS_SYSFLAGS:
            if (jail_chflags_allowed)
                return (0);
            else
                return (EPERM);
        ...
    }
    ...
}

```

Chapter 5. SYSINIT框架

SYSINIT是一个通用的用排序与分并行机制的框架。FreeBSD目前使用它来并行内核的初始化。SYSINIT使得FreeBSD的内核各子系统可以在内核或模块加载时被重整、添加、删除、替换，因此，内核和模块加载就不必去修改一个静态的有序初始化安排表甚至重新加载内核。这个体系也使得内核模块（在称KLD可以与内核不同链接、在引导系统加载，甚至在系统运行加载。这些操作是通过“内核连接器”(kernel linker)和“连接器集合”(linker set)完成的。

5.1. 接口

连接器集合(Linker Set)

一种接口方法。该方法将整个程序源文件中静态声明的数据收集到一个可寻址的数据单元中。

5.2. SYSINIT操作

SYSINIT要依靠连接器取遍布整个程序源代码中多声明的静态数据并把它组织成一个彼此相关的数据。接口方法被称“连接器集合”(linker set)。SYSINIT使用一个连接器集合以组织数据集合，包含一个数据条目的用序、函数、一个会被提交函数的数据指针。

SYSINIT按照首先函数排序以便并行。第一优先的是子系统的，输出SYSINIT分并行子系统的函数的全局序，定在sys/kernel.h中的枚举 `sysinit_sub_id`内。第二优先的是在子系统元素中的元素的序，定在sys/kernel.h中的枚举 `sysinit_elem_order`内。

有时刻需要使用SYSINIT：系统或内核模块加载，系统析或内核模块卸载。内核子系统通常在系统使用SYSINIT的定以初始化数据。例如，程序度子系统使用一个SYSINIT定来初始化并行列数据。程序避免直接使用 `SYSINIT()`，于物理真上使用 `DRIVER_MODULE()`用的函数先的存在，如果存在，再并行的初始化。一系列程中，会做一些的事情，然后用 `SYSINIT()`本身。于非一部分的虚，改用 `DEV_MODULE()`。

5.3. 使用SYSINIT

5.3.1. 接口

5.3.1.1. 文件

```
sys/kernel.h
```

5.3.1.2. 宏

```
SYSINIT(uniquifier, subsystem, order, func, ident)
SYSUNINIT(uniquifier, subsystem, order, func, ident)
```

5.3.2. `sysinit`

宏`SYSINIT()`在`SYSINIT`数据集中 建立一个`SYSINIT`数据项，以便`SYSINIT`在系级或模块级排序 并 行其中的函数。`SYSINIT()`有一个参数`uniquifier`， `SYSINIT`用它来标识数据项，随后是子系项序号、子系项元素序号、待用函数、函数数据。所有的函数必须有一个恒量指针参数。

例 1. `SYSINIT()`的例子

```
#include sys/kernel.h

void foo_null(void *unused)
{
    foo_doo();
}

SYSINIT(foo, SI_SUB_F00, SI_ORDER_F00, foo_null, NULL);

struct foo foo_voodoo = {
    F00_VOODOO;
}

void foo_arg(void *vdata)
{
    struct foo *foo = (struct foo *)vdata;
    foo_data(foo);
}

SYSINIT(bar, SI_SUB_F00, SI_ORDER_F00, foo_arg, foo_voodoo);
```

注意，`SI_SUB_F00`和`SI_ORDER_F00` 应当分列在上面提到的枚项`sysinit_sub_id`和`sysinit_elem_order`之中。既可以使用已有的枚项， 也可以将自己的枚项添加到枚项的定义之中。 可以使用数学表达式微调`SYSINIT`的行序。 以下的例子示例了一个需要恰好要在内核参数调整之前执行的`SYSINIT`。

例 2. 调整`SYSINIT()`行序的例子

```
static void
mptable_register(void *dummy __unused)
{
    apic_register_enumerator(mptable_enumerator);
}

SYSINIT(mptable_register, SI_SUB_TUNABLES - 1, SI_ORDER_FIRST,
        mptable_register, NULL);
```

5.3.3. 析

宏`SYSUNINIT()`的行与`SYSINIT()`的相当，只是它将数据填加至`SYSINIT`的析数据集。

例 3. `SYSUNINIT()`的例子

```
#include sys/kernel.h

void foo_cleanup(void *unused)
{
    foo_kill();
}
SYSUNINIT(foofoo, SI_SUB_FOO, SI_ORDER_FOO, foo_cleanup, NULL);

struct foo_stack foo_stack = {
    FOO_STACK_VOODOO;
}

void foo_flush(void *vdata)
{
}
SYSUNINIT(barfoo, SI_SUB_FOO, SI_ORDER_FOO, foo_flush, foo_stack);
```


Chapter 6. TrustedBSD MAC 框架

6.1. MAC 文 版 声明

本文 是作 DARPA CHATS 研究 的一部分, 由供 于 Security Research Division of Network Associates 公司Safeport Network Services and Network Associates Laboratories 的Chris Costello依据 DARPA/SPAWAR 合同 N66001-01-C-8035 ("CBOSS"), 在 FreeBSD 项目 写的。

Redistribution and use in source (SGML DocBook) and 'compiled' forms (SGML, HTML, PDF, PostScript, RTF and so forth) with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code (SGML DocBook) must retain the above copyright notice, this list of conditions and the following disclaimer as the first lines of this file unmodified.
2. Redistributions in compiled form (transformed to other DTDs, converted to PDF, PostScript, RTF and other formats) must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.



THIS DOCUMENTATION IS PROVIDED BY THE NETWORKS ASSOCIATES TECHNOLOGY, INC "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL NETWORKS ASSOCIATES TECHNOLOGY, INC BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



本文中 可 的非官方中文翻 供参考, 不作 判定任何 任的依据。如与英文原文有出入, 以英文原文 准。

在 足下列 可条件的前提下, 允 再分 或以源代 (SGML DocBook) 或 " " (SGML, HTML, PDF, PostScript, RTF 等) 的 修改或未修改的形式 :

1. 再分 源代 (SGML DocBook) 必 不加修改的保留上述版 告示、本条件清 和下述 作 文件的最先若干行。
2. 再分 的形式 (其它DTD、PDF、PostScript、RTF 或其它形式), 必 将上述版 告示、本条件清 和下述 制到与分 品一同提供的文件, 以及其它材料中。



本文由 NETWORKS ASSOCIATES TECHNOLOGY, INC "按"状条件"提供, 并在此明示不提供任何明示或暗示的保障, 包括但不限于"商"性、"特定目的的"用性的暗示保障。任何情况下, NETWORKS ASSOCIATES TECHNOLOGY, INC 均不"任何直接、"接、"偶然、"特殊、"性的, 或必然的"失(包括但不限于替代商品或服务的采、使用、数据或利益的"失或"中断)"", 无"是如何"致的并以任何有"任"的, 无"是否是在本文"使用以外以任何方式"生的契、"格"任或是民事侵"行"(包括疏忽或其它)中的, 即使已被告知"生"失的可能性。

6.2. "解析

FreeBSD 以一个内核安全"展性框架(TrustedBSD MAC 框架)的方式, "若干"制"控制策略(也称"集"式"控制策略") 提供"性支持。MAC 框架是一个"入式的"控制框架, 允"新的安全策略更方便地融入内核:安全策略可以静"入内核,也可以 在引"加,甚至在"行"加。"框架所提供的"准化接口,使得"行在其上的安全策略模"能"系"象的安全属性"行"如"等一系列操作。 MAC 框架的存在, "化了"些操作在策略模"中的",从而"著降低了新安全策略模"的"度。

本章将介" MAC 策略框架, "者提供一个示例性的 MAC 策略模"文。

6.3. 概述

TrustedBSD MAC 框架提供的机制,允"在其上"行的内核模"在内核"或者"行, "内核的"控制模型"行"展。 新的系"安全策略作"一个内核模"",并被"接到内核中;如果系"中同"存在多个安全策略模", "它"的决策"果将以某"定的方式"合。 "了"化新安全策略的", MAC 向上提供了大量用于"控制的基"施,特"是, ""的或者持久的、策略无"的"象安全"的支持。"支持目前仍是"性的。

本章所提供的信息不"将使在 MAC 使能"境下工作的潜在用"受益, 也可以"需要了解 MAC 框架是如何支持"内核"控制"行"展的策略模""人"所用。

6.4. 安全策略背景知"

"制"控制(称 MAC), 是指由操作系"制"施的一""用的"控制策略。 在某些情况下,"制"控制的策略可能会与自主"控制(称 DAC)所提供的保"措施"生冲突, 后者是用来向非管理"用"数据采取保"措施提供支持的。在"的 UNIX 系"中, DAC 保"措施包括文件"模式和"控制列表;而 MAC "提供"程控制和防火"等。 操作系""者和安全机制研究人""多"典的 MAC 安全策略作了形式化的表述,比如, 多"安全(MLS)机密性策略, Biba 完整性策略,基于角色的"控制策略(RBAC), 域和型裁决策略(DTE),以及型裁决策略(TE)。 安全策略的形式化表述被称"安全模型。"个模型根据一系列条件做出安全相"的决策, "些条件包括, 用"的身、角色和安全信任状,以及"象的安全"(用来代表"象数据的机密性/完整性")。

TrustedBSD MAC 框架所提供的"策略模"的支持,不"可以用来"上述所有策略, "能用于"其他利用已有安全属性(如,用"和"ID、文件"展属性等)决策的系"安全"化策略。 此外,因"具体策略模"在"授"方面所"有的高度"活性和自主性,所以MAC 框架同"可以用来"完全自主式的安全策略。

6.5. MAC 框架的内核体系"

TrustedBSD MAC 框架"大多数的"控制模"提供基本"施,允"它"以内核模"的形式"活地"展系"中"施的安全策略。 如果系"中同"加了多个策略, MAC 框架将"将各个策略的授"果以一" (某

程度上) 有意的方式合, 形成最后的决策。

6.5.1. 内核元素

MAC 框架由下列内核元素成:

- 框架管理接口
- 并与同原
- 策略注册
- 内核象的展性安全
- 策略入口函数的合操作
- 管理原
- 由内核服用的入口函数 API
- 策略模的入口函数 API
- 入口函数的 (包括策略生命周期管理、管理和控制三部分)
- 管理策略无的系用
- 用的 `mac_syscall()` 系用
- 以 MAC 的策略加模形式的各安全策略

6.5.2. 框架管理接口

TrustedBSD MAC 框架行直接管理的方式有三:通 `sysctl` 子系、通 `loader` 配置, 或者使用系用。

多数情况下, 与同一个内核内部量相的 `sysctl` 量和 `loader` 参数的名字是相同的, 通置它, 可以控制保护措施的实施, 比如, 某个策略在各个内核子系中的施与否等等。外, 如果在内核支持 MAC 时, 内核将若干计数器以跟踪的分配使用情况。通常不建在用境下通在不同子系上置不同的量或参数来施控制, 因方法将会作用于系中所有的活策略。如果希望具体策略施管理而不相影其他活策略, 当使用策略的控制, 因方法的控制粒度更, 并能更好地保策略模的功能一致性。

与其他内核模一, 系管理可以通过系的模管理系用和其他系接口, 包括 `boot loader` 量, 策略模的行加与卸操作; 策略模可以在加, 置加志, 来指示系其加、卸操作行相控制, 比如阻止非期望的卸操作。

6.5.3. 策略表的并与同

在行, 系中活的策略集合可能生, 然而策略入口函数的使用操作并不是原子性的, 因此, 当某一个入口函数正被使用, 系需要提供外的同机制来阻止策略模的加与卸, 以保当前活的策略集合不会在此程中生改。通使用"框架忙"计数器, 就可以做到一点: 一旦某个入口函数被用, 计数器的被加1; 而当当一个入口函数用束, 计数器的被少1。计数器的, 如果其正, 框架将阻止策略表的修改操作, 求操作的程将被迫入睡眠, 直到计数器的重新少到0止。计数器本身由一个互斥保, 同合一个条件量(用于醒等待策略表行修改操作的睡眠程)。采用同模型的一个副作用是, 在同一个策略模内部, 允嵌套地用框架, 不情况其很少出。

了少由于采用计数器引入的外, 者采用了各化措施。其中包括, 当策略表空或者其中含有静表

(那些只能在系调行之前加而且不能卸的策略), 框架不计数器行操作, 其值是0, 从而将此的同值到0。 一个端的方法是, 使用一个计数器来禁止在行加的策略表行修改, 此不再需要策略表的使用行同保。

因 MAC 框架不允许在某些入口函数之内阻塞, 所以不能使用普通的睡眠。 故而, 加或卸操作可能会等待框架空而被阻塞相当的一段。

6.5.4. 同

MAC 框架必须其的安全属性的存提供同保。下列情形, 可能导致安全属性的一致: 第一, 作安全属性的所有者, 内核象本身可能同被多个程; 第二, MAC 框架代是可重入的, 即允多个程同在框架内行。通常, MAC 框架使用内核象数据上已有的内核同机制来保其上附加的 MAC 安全。例如, 套接字上的 MAC 由已有的套接字互斥保。似的, 于安全的并的程与其所在象行的并在上是一的, 例如, 信任状安全, 将保持与数据中其他内容一致的"写控制"的更新程。 MAC 框架在引用一个内核象, 将首先象上的需要用到的行断言。策略模的写者必了解些同, 因它可能会限制安全所能行的型。个例子, 如果通入口函数策略模的是某个信任状的只引用, 那在策略内部, 只能的状态。

6.5.5. 策略的同与并

FreeBSD 内核是一个可占式的内核, 因此, 作内核一部分的策略模也必是可重入的, 也就是, 在策略模必假多个内核程可以同通不同的入口函数入模。 如果策略模使用可被修改的内核状, 那需要在策略内部使用恰当的同原, 保在策略内部的多个程不会因此察到不一致的内核状, 从而避免由此生的策略操作。此, 策略可以使用 FreeBSD 有的同原, 包括互斥、睡眠、条件和数信号量。 些同原的使用必慎重, 需要特别注意点: 第一, 保持有的内核上次序; 第二, 在非睡眠的入口函数之内不要使用互斥和醒操作。

避免反内核上次序或造成上, 策略模在用其他内核子系之前, 通常要放所有在策略内部申的。 做的果是, 在全局上次序形成的拓朴中, 策略内部的作叶子点, 从而保了些的使用不会致由于上次序混乱造成的死。

6.5.6. 策略注册

了当前使用的策略模集合, MAC 框架个表: 一个静表和一个表。 个表的数据和操作基本相同, 只是表外使用了一个"引用数"以同其的操作。 当包含 MAC 框架策略的内核模被加, 策略模会通 `SYSINIT` 用一个注册函数; 相的, 当一个策略模被卸, `SYSINIT` 也会用一个注函数。 只有当遇到下列情况之一, 注册程才会失: 一个策略模被加多次, 或者系源不足不能足注册程的需要 (例如, 策略模需要内核象添加而可用源不足), 或者其他的策略加前提条件不足 (有些策略要求只能在系引之前加)。 似的, 如果一个策略被不可卸的, 其用注程将会失。

6.5.7. 入口函数

内核服与 MAC 框架之行交互有途径: 一是, 内核服用一系列 API 通知 MAC 框架安全事件的生; 二是, 内核服向 MAC 框架提供一个指向安全象的策略无安全数据值的指。 指由 MAC 框架由管理入口函数行, 并且, 只要管理相象的内核子系作修改, 就可以允 MAC 框架向策略模提供服。 例如, 在程、程信任状、套接字、管道、Mbuf、网接口、IP 重列和其他各安全相的数据中均加了指向安全的指。 外, 当需要做出重要的安全决策, 内核服也会用 MAC 框架, 以便各个策略模根据其自己的准 (可以使用存在安全中的数据) 完善些决策。 大多数安全相

的决策是式的控制；也有少数及更加一般的决策函数，比如，套接字的数据包匹配和程序行的刻的。

6.5.8. 策略合

如果内核中同加了多个策略模，些策略的决策果将由框架使用一个合成算子来行合，得出最的果。目前，算子是硬，并且只有当所有的活策略均求表示同意才会返回成功。由于各个策略返回的出条件可能并不相同（成功、被拒、求象不存在等等），需要使用一个子先从各个策略返回的条件集合中出一个作最返回果。一般情况下，与“被拒”相比，将更向于“求象不存在”。尽管不能从理上保合成果的有效性与安全性，但果表明，于多的策略集合来，事的如此。例如，的可信系常常采用似的方法多个安全策略行合。

6.5.9. 支持

与多需要象添加安全控制展一，MAC 框架各用可的象提供了一用于管理策略无的系用。常用的型有，partition符、机密性、完整性、区（非等）、域、角色和型。“策略无”的意思是指，的与使用它的具体策略模无，而同策略模能完全独立地定和使用与象相的元数据的。用用程序提供一格式的基于字符串的，由使用它的策略模解析其内在含并决定其外在表示。如果需要，用程序可以使用多重元素。

内存中的例被存放在由 slab 分配的 struct label 数据中。是一个固定度的数，个元素是由一个 void * 指和一个 long 成的合。申存的策略模在向 MAC 注册，将被分配一个“slot”，作框架分配其使用的策略元素在整个存中的位置索引。而所分配的存空的完全由策略模来决定：MAC 框架向策略模提供了一系列入口函数用于内核象生命周期的各事件行控制，包括，象的初始化、的/建和象的注。使用些接口，可以如数等存模型。MAC 框架是入口函数入一个指向相象的指和一个指向象的指，因此，策略模能直接而无需知悉象的内部。唯一的例外是程信任状，指向其的指必由策略模手解析算。今后的 MAC 框架可能会此行改。

初始化入口函数通常有一个睡眠志位，用来表明一个初始化操作是否允中途睡眠等待；如果不允，可能会失返回，并要求撤此次分配操作（乃至象分配操作）。例如，如果在网理中断因不允睡眠或者用者持有一个互斥，就可能出情况。考到在理中的网数据包（Mbufs）上性能失太大，策略必就自己 Mbuf 行要求向 MAC 框架做出特声明。加到系中而又使用的策略必理未被其初始化函数理的象作好准，些象在策略加之前就已存在，故而无法在初始化用策略的相函数行理。MAC 框架向策略保，没有被初始化的 slot 的必或者 NULL，策略可以借此到未初始化的。需要注意的是，因 Mbuf 的存分配是有条件的，因此需要使用其的加策略可能需要理 Mbuf 中 NULL 的指。

于文件系象的，MAC 框架在文件的展属性中其分配永久存。只要可能，展属性的原子化的事操作就被用于保 vnode 上安全的合更新操作的一致性——目前，特性只被 UFS2 文件系支持。了粒度的文件系象（即个文件系象一个），策略写者可能使用一个（或者若干）展属性。了提高性能，vnode 数据中有一个（v_label）字段，用作磁的冲；vnode 例化，策略可以将装入冲，并在需要其行更新。如此，不必在次行控制，均无条件地磁上的展属性。



目前，如果一个使用策略允被卸，卸模之后，其状 slot 尚无法被系回收重用，由此致了 MAC 框架策略卸—重操作数目上的格限制。

6.5.10. 相 系 用

MAC 框架向用程序提供了一系用：其中大多数用于向行和修改策略无操作的用 API 提供支持。

些管理系用，接受一个描述，`struct mac`，作入参数。个的主体是一个数，其中个元素包含了一个用的 MAC 形式。个元素又由部分成：一个字符串名字，和其的。个策略可以向系声明一个特定的元素名字，一来，如果需要，就可以将若干个相互独立的元素作一个整体行理。策略模由入口函数，在内核和用提供的之作翻的工作，提供了元素上的高度活性。管理系用通常有的函数包装，些包装函数可以提供内存分配和理功能，从而化了用用程序的管理工作。

目前的FreeBSD 内核提供了下列 MAC 相的系用：

- `mac_get_proc()` 用于当前程的安全。
- `mac_set_proc()` 用于求改当前程的安全。
- `mac_get_fd()` 用于由文件描述符所引用的象（文件、套接字、管道文件等等）的安全。
- `mac_get_file()` 用于由文件系路径所描述的象的安全。
- `mac_set_fd()` 用于求改由文件描述符所引用的象（文件、套接字、管道文件等等）的安全。
- `mac_set_file()` 用于求改由文件系路径所描述的象的安全。
- `mac_syscall()` 通用系用,策略模能在不修改系用表的前提下建新的系用；其用参数包括：目策略名字、操作号和将被策略内部使用的参数。
- `mac_get_pid()` 用于由程号指定的一个程的安全。
- `mac_get_link()` 与 `mac_get_file()` 功能相同，只是当路径参数的最后一符号接，前者将返回符号接的安全，而后者将返回其所指文件的安全。
- `mac_set_link()` 与 `mac_set_file()` 功能相同，只是当路径参数的最后一符号接，前者将置符号接的安全，而后者将置其所指文件的安全。
- `mac_execve()` 与 `execve()` 功能似，只是前者可以在始行一个新程序，根据入的求参数，置行程的安全。由于行一个新程序而致的程安全的改，被称"”。
- `mac_get_peer()`，通一个套接字自，用于一个程套接字等体的安全。

除了上述系用之外，也可以通 `SIOCSIGMAC` 和 `SIOCSIFMAC` 网接口的 `ioctl` 系用来和置网接口的安全。

6.6. MAC策略模体系

安全策略可以直接入内核，也可以成独立的内核模，在系引或者行使用模加命令加。策略模通一先定好的入口函数与系交互。通它，策略模能掌握某些系事件的生，并且在必要的候影系的控制决策。个策略模包含下列成部分：

- 可：策略配置参数
- 策略和参数的集中
- 可：策略生命周期事件的，比如，策略的初始化和
- 可：所内核象的安全行初始化、和的支持

- 可：所象的使用程行控以及修改象安全的支持
- 策略相的控制入口函数的
- 策略志、模入口函数和策略特性的声明

6.6.1. 策略注

策略模可以使用 `MAC_POLICY_SET()` 宏来声明。宏完成以下工作：策略命名（向系声明策略提供的名字）；提交策略定义的 `MAC` 入口函数向量的地址；按照策略的要求置策略的加志位，保 `MAC` 框架将以策略所期望的方式其行操作；外，可能求框架策略分配状 `slot`。

```
static struct mac_policy_ops mac_policy_ops =
{
    .mpo_destroy = mac_policy_destroy,
    .mpo_init = mac_policy_init,
    .mpo_init_bpfdesc_label = mac_policy_init_bpfdesc_label,
    .mpo_init_cred_label = mac_policy_init_label,
/* ... */
    .mpo_check_vnode_setutimes = mac_policy_check_vnode_setutimes,
    .mpo_check_vnode_stat = mac_policy_check_vnode_stat,
    .mpo_check_vnode_write = mac_policy_check_vnode_write,
};
```

如上所示，`MAC` 策略入口函数向量，`macpolicyops`，将策略模中定的功能函数挂接到特定的入口函数地址上。在后的“[入口函数参考](#)”小节中，将提供可用入口函数功能描述和原型的完整列表。与模注册相的入口函数有个：`.mpo_destroy` 和 `.mpo_init`。当某个策略向模框架注册操作成功，`.mpo_init` 将被用，此后其他的入口函数才能被使用。特殊的使得策略有机会根据自己的需要，行特定的分配和初始化操作，比如特殊数据或的初始化。卸一个策略模，将用 `.mpo_destroy` 用来放策略分配的内存空或注其申的。目前，了防止其他入口函数被同用，用上述个入口函数的程必持有 `MAC` 策略表的互斥：限制将被放，但与此同，将要求策略必慎使用内核原，以避免由于上次序或睡眠造成死。

之所以向策略声明提供模名字域，是了能唯一模，以便解析模依系。使用恰当的字符串作名字。在策略加和卸，策略的完整字符串名字将由内核日志示用。外，当向用程告状信息也会包含字符串。

6.6.2. 策略志

在声明提供志参数域的机制，允策略模在作模被加，就自身特性向 `MAC` 框架提供明。目前，已定的志有三个：

`MPC_LOADTIME_FLAG_UNLOADOK`

表示策略模可以被卸。如果未提供志，表示策略模拒被卸。那些使用安全状的，而又不能在行放状的模可能会置志。

`MPC_LOADTIME_FLAG_NOTLATE`

表示策略模必在系引进程行加和初始化。如果志被置，那在系引之后注册模的求将被

MAC 框架所拒。那些需要大系的象行安全初始化工作，而又不能理含有未被正初始化安全的象的策略模可能会置志。

MPC_LOADTIME_FLAG_LABELMBUFS

表示策略模要求 Mbuf 指定安全，并且存其所需的内存空是提前分配好的。缺省情况下，MAC 框架并不会 Mbuf 分配存，除非系中注册的策略模中至少有一个置了志。做法在没有策略需要 Mbuf 做，著地提升了系网性能。外，在某些特殊境下，可以通置内核， `MAC_ALWAYS_LABEL_MBUF`，制 MAC 框架 Mbuf 的安全分配存，而不上述志如何置。



那些使用了 `MPC_LOADTIME_FLAG_LABELMBUFS` 志但没有置 `MPC_LOADTIME_FLAG_NOTLATE` 志的策略模必能正地理通入口函数入的 `NULL` 的 Mbuf 安全指。是因那些没有分配存的理中的 Mbuf 在一个需要 Mbuf 安全的策略模加之后，其安全的指将仍然空。如果策略在网子系活之前被加（即，策略不是被推加的），那所有的 Mbuf 的存的分配就可以得到保。

6.6.3. 策略入口函数

MAC 框架注册的策略提供四型的入口函数：策略注册和管理入口函数；用于理内核象声明周期事件，如初始化、建和，的入口函数；理策略模感兴趣的控制决策事件的入口函数；以及用于管理象安全的用入口函数。此外，有一个 `mac_syscall()` 入口函数，被策略模用于在不注册新的系用的前提下，展内核接口。

策略模的写人除了必清楚在入特定入口函数之后，些象是可用的之外，熟知内核所采用的加策略。程人在入口函数之内避免使用非叶点，并且遵循和修改象的加程，以降低致死的可能性。特地，程序清楚，然在通常情况下，入入口函数之后，已上了一些，可以安全地象及其安全，但是并不能保它行修改（包括象本身和其安全）也是安全的。相的上信息，可以参考 MAC 框架入口函数的相文。

策略入口函数把个分指向象本身和其安全的指策略模。一来，即使策略并不熟悉象内部，也能基于作出正决策。只有程信任状个象例外：MAC 框架是假所有的策略模是理解其内部的。

6.7. MAC策略入口函数参考

6.7.1. 通用的模入口函数

6.7.1.1. `mpo_init`

```
void
mpo_init (struct mac_policy_conf);
```

参数	明	定
conf	MAC 策略定	

策略加事件。当前程正持有策略表上的互斥，因此是非睡眠的，其他内核子系的用也慎重。

如果需要在策略初始化阶段进行可能造成睡眠阻塞的内存分配操作，可以将它放在一个独立的模块 `SYSINITO` 进程中集中进行。

6.7.1.2. `mpo_destroy`

```
void
    mpo_destroy (struct mac_policy_conf);
```

参数	说明	定义
conf	MAC 策略定义	

策略加载事件。必须持有策略表互斥锁，因此需要慎重行事。

6.7.1.3. `mpo_syscall`

```
int
    mpo_syscall (struct thread,
                 int call,
                 void *arg);
```

参数	说明	定义
td	调用线程	
call	策略特有的系统调用号	
arg	系统调用参数的指针	

该入口函数提供策略调用的系统调用，该策略模块不需要向其向用线程提供的任何一个外服务而注册调用的系统调用。由用程序提供的策略注册名字来定义提供其所申请服务的特定策略，所有参数将通过该入口函数被调用的策略。当新服务，安全模块必须在必要时通过 MAC 框架用相应的信号控制机制。比方，假如一个策略有了某外的信号功能，那么它必须用相应的信号控制，以接受 MAC 框架中注册的其他策略的信号。



不同的模块需要并行地手动进行 `copyin()` 拷贝系统调用数据。

6.7.1.4. `mpo_thread_userret`

```
void
    mpo_thread_userret (struct thread);
```

参数	说明	定义
td	返回线程	

使用该入口函数，策略模块能够在线程返回用户空间（系统调用返回、异常返回等等）进行 MAC 相关的数据处理工作。使用该线程的策略需要使用该入口函数，因在管理系统调用的进程中，并不是在任意时刻都能申请到线程的；线程的信号可能表示线程的信号信息、线程历史或者其他数据。使用该入口函数，该进程信任状所作的修改

可能被存放在 `p_label` ,域受一个线程自旋的保；接下来,置线程的**TDF_ASTPENDING** 志位和线程的**PS_MACPENDM**志位,表明将度一个 `userret` 入口函数的用。通入口函数, 策略可以在相的同上下文中建信任状的替代品。策略程人必清楚,需要保与度一个 `AST` 相的事件行次序, 同所行的 `AST` 可能很,而且在理多程用程序可能被重入。

6.7.2. 操作

6.7.2.1. `mpo_init_bpfdesc_label`

```
void
    mpo_init_bpfdesc_label (struct label);
```

参数	明	定
label	将被用的新	

一个新近例化的 `bpfdesc` (`BPF` 描述子) 初始化。可以睡眠。

6.7.2.2. `mpo_init_cred_label`

```
void
    mpo_init_cred_label (struct label);
```

参数	明	定
label	将被初始化的新	

一个新近例化的用信任状初始化。可以睡眠。

6.7.2.3. `mpo_init_devfsdirent_label`

```
void
    mpo_init_devfsdirent_label (struct label);
```

参数	明	定
label	将被用的新	

一个新近例化的 `devfs` 表初始化。可以睡眠。

6.7.2.4. `mpo_init_ifnet_label`

```
void
    mpo_init_ifnet_label (struct label);
```

参数	说明	定义
label	将被用的新	

一个新近例化的网接口初始化。可以睡眠。

6.7.2.5. mpo_init_ipq_label

```
void
    mpo_init_ipq_label (struct label,
                        int flag);
```

参数	说明	定义
label	将被用的新	
flag	睡眠/不睡眠 malloc(9); 参下文	

一个新近例化的 IP 分片重列初始化。其中的flag域可能取M_WAITOK 或 M_NOWAIT之一，用来避免在初始化用中因 malloc(9) 而入睡眠。IP 分片重列的分配操作通常是在性能有格要求的境下行的，因此代必小心地避免睡眠和的操作。IP 分片重列分配操作失上述入口函数将失返回。

6.7.2.6. mpo_init_mbuf_label

```
void
    mpo_init_mbuf_label (int flag,
                        struct label);
```

参数	说明	定义
flag	睡眠/不睡眠 malloc(9); 参下文	
label	将被初始化的策略	

一个新近例化的 mbuf 数据包部（mbuf）初始化。其中的flag的可能取M_WAITOK和 M_NOWAIT之一，用来避免在初始化用中因 malloc(9) 而入睡眠。Mbuf 部的分配操作常常在性能有格要求的境下被繁行，因此代必小心地避免睡眠和的操作。上述入口函数在 Mbuf 部分配操作失将失返回。

6.7.2.7. mpo_init_mount_label

```
void
    mpo_init_mount_label (struct label,
                        struct label);
```

参数	说明	定义
mntlabel	将被初始化的mount 策略	

参数	说明	定义
fslabel	将被初始化的文件系统策略	

一个新近例化的 mount 点初始化。可以睡眠。

6.7.2.8. mpo_init_mount_fs_label

```
void
    mpo_init_mount_fs_label (struct label);
```

参数	说明	定义
label	将被初始化的	

一个新近加的文件系初始化。可以睡眠。

6.7.2.9. mpo_init_pipe_label

```
void
    mpo_init_pipe_label (struct);
```

参数	说明	定义
label	将被填写的	

一个例化的管道初始化安全。可以睡眠。

6.7.2.10. mpo_init_socket_label

```
void
    mpo_init_socket_label (struct label,
                          int flag);
```

参数	说明	定义
label	将被初始化的新	
flag	malloc(9) flags	

一个例化的套接字初始化安全。其中的 flag 域的必被指定 M_WAITOK和 M_NOWAIT之一，以避免在初始化程中使用可能睡眠的 malloc(9)。

6.7.2.11. mpo_init_socket_peer_label

void

```
mpo_init_socket_peer_label (struct label,
                             int flag);
```

参数	说明	定义
label	将被初始化的新	
flag	malloc(9) flags	

例化的套接字等体行的初始化。其中的 flag 域的必被指定 M_WAITOK 和 M_NOWAIT 之一，以避免在初始化程中使用可能睡眠的 malloc(9)。

6.7.2.12. mpo_init_proc_label

void

```
mpo_init_proc_label (struct label);
```

参数	说明	定义
label	将被初始化的新	

一个例化的程初始化安全。可以睡眠。

6.7.2.13. mpo_init_vnode_label

void

```
mpo_init_vnode_label (struct label);
```

参数	说明	定义
label	将被初始化的新	

一个例化的 vnode 初始化安全。可以睡眠。

6.7.2.14. mpo_destroy_bpfdesc_label

void

```
mpo_destroy_bpfdesc_label (struct label);
```

参数	说明	定义
label	bpfdesc	

一个 BPF 描述子上的。在入口函数中，策略当放所有在内部分配与 label 相的存空，以便。

6.7.2.15. mpo_destroy_cred_label

```
void  
    mpo_destroy_cred_label (struct label);
```

参数	说明	定义
label	将被销毁的	

是一个信任状态上的。在入口函数中，策略当释放所有在内部分配的与 label 相关的存储空间，以便。

6.7.2.16. mpo_destroy_devfsdirent_label

```
void  
    mpo_destroy_devfsdirent_label (struct label);
```

参数	说明	定义
label	将被销毁的	

是一个 devfs 表上的。在入口函数中，策略当释放所有在内部分配的与 label 相关的存储空间，以便。

6.7.2.17. mpo_destroy_ifnet_label

```
void  
    mpo_destroy_ifnet_label (struct label);
```

参数	说明	定义
label	将被销毁的	

与一个已删除接口相关的。在入口函数中，策略当释放所有在内部分配的与 label 相关的存储空间，以便。

6.7.2.18. mpo_destroy_ipq_label

```
void  
    mpo_destroy_ipq_label (struct label);
```

参数	说明	定义
label	将被销毁的	

与一个 IP 分片队列相关的。在入口函数中，策略当释放所有在内部分配的与 label 相关的存储空间，以便。

6.7.2.19. mpo_destroy_mbuf_label

```
void
    mpo_destroy_mbuf_label (struct label);
```

参数	说明	定义
label	将被销毁的	

与一个 Mbuf 相关的。在入口函数中，策略当释放所有在内部分配的与 label 相关的存储空间，以便。

6.7.2.20. mpo_destroy_mount_label

```
void
    mpo_destroy_mount_label (struct label);
```

参数	说明	定义
label	将被销毁的 Mount 点	

与一个 mount 点相关的。在入口函数中，策略当释放所有在内部分配的与 mntlabel 相关的存储空间，以便。

6.7.2.21. mpo_destroy_mount_label

```
void
    mpo_destroy_mount_label (struct label,
                             struct label);
```

参数	说明	定义
mntlabel	将被销毁的 Mount 点	
fslabel	File system label being destroyed	

与一个 mount 点相关的。在入口函数中，策略当释放所有在内部分配的，与 mntlabel 和fslabel 相关的存储空间，以便。

6.7.2.22. mpo_destroy_socket_label

```
void
    mpo_destroy_socket_label (struct label);
```

参数	说明	定义
label	将被释放的套接字	

与一个套接字相同的。在入口函数中，策略当放所有在内部分配的，与 label 相同的存空间，以便。

6.7.2.23. mpo_destroy_socket_peer_label

```
void
    mpo_destroy_socket_peer_label (struct label);
```

参数	说明	定义
peerlabel	将被释放的套接字等体	

与一个套接字相同的等体。在入口函数中，策略当放所有在内部分配的，与 label 相同的存空间，以便。

6.7.2.24. mpo_destroy_pipe_label

```
void
    mpo_destroy_pipe_label (struct label);
```

参数	说明	定义
label	管道	

一个管道的。在入口函数中，策略当放所有在内部分配的，与 label 相同的存空间，以便。

6.7.2.25. mpo_destroy_proc_label

```
void
    mpo_destroy_proc_label (struct label);
```

参数	说明	定义
label	程	

一个程的。在入口函数中，策略当放所有在内部分配的，与 label 相同的存空间，以便。

6.7.2.26. mpo_destroy_vnode_label

```
void
    mpo_destroy_vnode_label (struct label);
```


参数	说明	定义
label	进程ID	

返回一个 vnode 的引用。在入口函数中，策略应当释放所有在内部分配的，与 label 相关的存储空间，以便重用。

6.7.2.27. mpo_copy_mbuf_label

```
void
mpo_copy_mbuf_label (struct label,
                    struct label);
```

参数	说明	定义
src	源ID	
dest	目标ID	

将 src 中的信息拷贝到 dest 中。

6.7.2.28. mpo_copy_pipe_label

```
void
mpo_copy_pipe_label (struct label,
                    struct label);
```

参数	说明	定义
src	源ID	
dest	目标ID	

将 src 中的信息拷贝至 dest。

6.7.2.29. mpo_copy_vnode_label

```
void
mpo_copy_vnode_label (struct label,
                     struct label);
```

参数	说明	定义
src	源ID	
dest	目标ID	

将 src 中的信息拷贝至 dest。

6.7.2.30. mpo_externalize_cred_label

```
int
    mpo_externalize_cred_label (struct label *label,
                                char *element_name,
                                struct sbuf *sb,
                                int *claimed);
```

参数	说明	定义
label	将用外部形式表示的	
element_name	需要外部表示的策略的名字	
sb	用来存放的文本表示形式的字符 buffer	
claimed	如果可以填充element_data 域, 其数	

根据入的，生一个以外部形式表示的。一个外部形式，是内容的文本表示，它由用的程序使用，是用可的。目前的MAC方案将依次用策略的相入口函数，因此，具体策略的代，需要在填写sb之前，先element_name中指定的名字。如果element_name中的内容与的策略名字不相符，直接返回0。当数据的程中出，才返回非0。一旦策略决定填写element_data，*claim的数。

6.7.2.31. mpo_externalize_ifnet_label

```
int
    mpo_externalize_ifnet_label (struct label *label,
                                char *element_name,
                                struct sbuf *sb,
                                int *claimed);
```

参数	说明	定义
label	将用外部形式表示的	
element_name	需要外部表示的策略的名字	
sb	用来存放的文本表示形式的字符 buffer	
claimed	如果可以填充element_data 域, 其数	

根据入的，生一个以外部形式表示的。一个外部形式，是内容的文本表示，它由用的程序使用，是用可的。目前的MAC方案将依次用策略的相入口函数，因此，具体策略的代，需要在填写sb之前，先element_name中指定的名字。如果element_name中的内容与的策略名字不相符，直接返回0。当数据的程中出，才返回非0。一旦策略决定填写element_data，*claim的数。

6.7.2.32. mpo_externalize_pipe_label

```
int
mpo_externalize_pipe_label (struct label *label,
                           char *element_name,
                           struct sbuf *sb,
                           int *claimed);
```

参数	说明	定义
label	将用外部形式表示的	
element_name	需要外部表示的策略的名字	
sb	用来存放的文本表示形式的字符 buffer	
claimed	如果可以填充element_data 域, 其数	

根据入的，生一个以外部形式表示的。一个外部形式，是内容的文本表示，它由用的
用程序使用，是用可的。目前的MAC方案将依次用策略的相入口函数，因此，具体策略的代
，需要在填写sb之前，先element_name中指定的名字。如果element_name中的内容与
的策略名字不相符，直接返回0。当数据的程中出，才返回非0。
一旦策略决定填写element_data，*claim的数。

6.7.2.33. mpo_externalize_socket_label

```
int
mpo_externalize_socket_label (struct label *label,
                              char *element_name,
                              struct sbuf *sb,
                              int *claimed);
```

参数	说明	定义
label	将用外部形式表示的	
element_name	需要外部表示的策略的名字	
sb	用来存放的文本表示形式的字符 buffer	
claimed	如果可以填充element_data 域, 其数	

根据入的，生一个以外部形式表示的。一个外部形式，是内容的文本表示，它由用的
用程序使用，是用可的。目前的MAC方案将依次用策略的相入口函数，因此，具体策略的代
，需要在填写sb之前，先element_name中指定的名字。如果element_name中的内容与
的策略名字不相符，直接返回0。当数据的程中出，才返回非0。
一旦策略决定填写element_data，*claim的数。

6.7.2.34. mpo_externalize_socket_peer_label

```
int
    mpo_externalize_socket_peer_label (struct label *label,
                                       char *element_name,
                                       struct sbuf *sb,
                                       int *claimed);
```

参数	说明	定义
label	将用外部形式表示的	
element_name	需要外部表示的策略的名字	
sb	用来存放的文本表示形式的字符 buffer	
claimed	如果可以填充element_data 域, 其数	

根据入的, 生一个以外部形式表示的。 一个外部形式, 是内容的文本表示, 它由用的程序使用, 是用可的。 目前的MAC方案将依次用策略的相入口函数, 因此, 具体策略的代, 需要在填写sb之前, 先element_name中指定的名字。 如果element_name中的内容与的策略名字不相符, 直接返回0。 当数据的程中出, 才返回非0。 一旦策略决定填写element_data, *claim的数。

6.7.2.35. mpo_externalize_vnode_label

```
int
    mpo_externalize_vnode_label (struct label *label,
                                 char *element_name,
                                 struct sbuf *sb,
                                 int *claimed);
```

参数	说明	定义
label	将用外部形式表示的	
element_name	需要外部表示的策略的名字	
sb	用来存放的文本表示形式的字符 buffer	
claimed	如果可以填充element_data 域, 其数	

根据入的, 生一个以外部形式表示的。 一个外部形式, 是内容的文本表示, 它由用的程序使用, 是用可的。 目前的MAC方案将依次用策略的相入口函数, 因此, 具体策略的代, 需要在填写sb之前, 先element_name中指定的名字。 如果element_name中的内容与的策略名字不相符, 直接返回0。 当数据的程中出, 才返回非0。 一旦策略决定填写element_data, *claim的数。

6.7.2.36. mpo_internalize_cred_label

int

```
mpo_internalize_cred_label (struct label *label,  
                           char *element_name,  
                           char *element_data,  
                           int *claimed);
```

参数	说明	定义
label	将被填充的	
element_name	需要内部表示的策略的名字	
element_data	需要被的文本数据	
claimed	如果数据被正, 其数	

根据一个文本形式的外部表示数据, 建立一个内部形式的。目前的MAC方案将依次用所有策略的相应入口函数, 来求, 因此, 代码必须首先通比element_name中的内容和自己的策略名字, 来定是否需要element_data中存放的数据。似的, 如果名字不匹配或者数据操作成功, 函数返回0, 并*claimed的。

6.7.2.37. mpo_internalize_ifnet_label

int

```
mpo_internalize_ifnet_label (struct label *label,  
                             char *element_name,  
                             char *element_data,  
                             int *claimed);
```

参数	说明	定义
label	将被填充的	
element_name	需要内部表示的策略的名字	
element_data	需要被的文本数据	
claimed	如果数据被正, 其数	

根据一个文本形式的外部表示数据, 建立一个内部形式的。目前的MAC方案将依次用所有策略的相应入口函数, 来求, 因此, 代码必须首先通比element_name中的内容和自己的策略名字, 来定是否需要element_data中存放的数据。似的, 如果名字不匹配或者数据操作成功, 函数返回0, 并*claimed的。

6.7.2.38. mpo_internalize_pipe_label

int

```
mpo_internalize_pipe_label (struct label *label,  
                             char *element_name,  
                             char *element_data,  
                             int *claimed);
```

参数	说明	定义
label	将被填充的	
element_name	需要内部表示的策略的名字	
element_data	需要被的文本数据	
claimed	如果数据被正, 其数	

根据一个文本形式的外部表示数据, 建立一个内部形式的。目前的MAC方案将依次用所有策略的相应入口函数, 来求, 因此, 代码必须首先通比element_name中的内容和自己的策略名字, 来定是否需要element_data中存放的数据。似的, 如果名字不匹配或者数据操作成功, 函数返回0, 并*claimed的。

6.7.2.39. mpo_internalize_socket_label

int

```
mpo_internalize_socket_label (struct label *label,  
                              char *element_name,  
                              char *element_data,  
                              int *claimed);
```

参数	说明	定义
label	将被填充的	
element_name	需要内部表示的策略的名字	
element_data	需要被的文本数据	
claimed	如果数据被正, 其数	

根据一个文本形式的外部表示数据, 建立一个内部形式的。目前的MAC方案将依次用所有策略的相应入口函数, 来求, 因此, 代码必须首先通比element_name中的内容和自己的策略名字, 来定是否需要element_data中存放的数据。似的, 如果名字不匹配或者数据操作成功, 函数返回0, 并*claimed的。

6.7.2.40. mpo_internalize_vnode_label

int

```
mpo_initialize_vnode_label (struct label *label,  
                           char *element_name,  
                           char *element_data,  
                           int *claimed);
```

参数	说明	定义
label	将被填充的	
element_name	需要内部表示的策略的名字	
element_data	需要被的文本数据	
claimed	如果数据被正, 其数	

根据一个文本形式的外部表示数据, 建立一个内部形式的。目前的MAC方案将依次用所有策略的相应入口函数, 来请求的内部, 因此, 代码必须首先通比element_name中的内容和自己的策略名字, 来定是否需要element_data中存放的数据。似的, 如果名字不匹配或者数据操作成功, 函数返回0, 并*claimed的。

6.7.3. 事件

策略模使用MAC 框架提供的"事件"入口函数, 内核对象的行操作。策略模将感兴趣的被内核对象的相应生命周期事件 注册在恰当的入口点上。对象的初始化、建和事件均提供了子点。在某些对象上可以重新, 即, 允许用进程改对象上的。某些对象可以其特定的对象事件, 比如与 IP 重相关的的事件。一个典型的被对象在其生命周期中将有下列入口函数:

```
初始化          0  
(对象相的等待)  \  
建              0  
              \  
重新事件,       0----.  
各对象相的,    |      |  
控制事件       ~----0  
              \  
              0
```

使用初始化入口函数, 策略可以以一一的、与对象使用境无的方式置的初始。分配
一个策略的缺省 slot 0, 不使用策略可能并不需要行的初始化操作。

的建事件生在将一个内核数据同一个真的内核对象相 (内核对象例化) 的刻。
例如, 在真正被使用之前, 在一个冲池内已分配的 mbuf 数据, 将保持"未使用"状。因此, mbuf 的分配操作将致 mbuf 的初始化操作, 而 mbuf 的建操作被推到 mbuf 真正与一个数据相的刻。
通常, 用者将会提供建事件的上下文, 包括建境、建程中及其他对象的等。例如, 当一个套接字建一个 mbuf , 除了新建的 mbuf 及其之外, 作建者的套接字与其也被提交策略。
不提倡在建对象就其分配内存的原因有个: 建操作可能生在性能有格要求的内核接口上; 而且, 因建用不允许失, 所以无法告内存分配失。

对象特有的事件一般不会引起其他的事件，但是在对象上下文内修改，策略使用它可以互相进行修改或更新操作。例如，在MAC_UPDATE_IPQ 入口函数之内，某个 IP 分片重列的可能因列中接收了新的mbuf 而被更新。

控制事件将在后章中。

策略通行操作，放其分配的存空或的状态，之后内核才可以重用或者放对象的内核数据。

除了与特定内核对象定的普通之外，有一外的型：。些用于存放由用程提交的更新信息。它的初始化和操作与其他一，只是建事件，MAC_INTERNALIZE，略有不同：函数接受用提交的，将其化内核表示形式。

6.7.3.1. 文件系对象事件操作

6.7.3.1.1. mpo_associate_vnode_devfs

```
void
    mpo_associate_vnode_devfs (struct mount,
                                struct label,
                                struct devfs_dirent,
                                struct label,
                                struct vnode,
                                struct label);
```

参数	明	定
mp	Devfs 挂点	
fslabel	Devfs 文件系 (mp-mnt_fslabel)	
de	Devfs 目	
delabel	与 de 相的策略	
vp	与 de 相的 vnode	
vlabel	与 vp 相的策略	

根据参数 de 入的 devfs 目及其信息，一个新近建的 devfs vnode 填充 (vlabel) 。

6.7.3.1.2. mpo_associate_vnode_extattr

```
int
    mpo_associate_vnode_extattr (struct mount,
                                  struct label,
                                  struct vnode,
                                  struct label);
```

参数	明	定
mp	文件系挂点	

参数	说明	定义
fslabel	文件系统名	
vp	将被关联的 vnode	
vlabel	与 vp 相关的策略名	

从文件系统展属性中获取 vp 的值。成功，返回 0。不成功，在 errno 指定的相关的错误。如果文件系统不支持展属性的获取操作，可以考虑将 fslabel 拷贝至 vlabel。

6.7.3.1.3. mpo_associate_vnode_singlelabel

```
void
mpo_associate_vnode_singlelabel (struct mount,
                                struct label,
                                struct vnode,
                                struct label);
```

参数	说明	定义
mp	文件系统挂载点	
fslabel	文件系统名	
vp	将被关联的 vnode	
vlabel	与 vp 相关的策略名	

在非多重文件系统上，使用入口函数，根据文件系统名，fslabel，vp 设置策略。

6.7.3.1.4. mpo_create_devfs_device

```
void
mpo_create_devfs_device (dev_t dev,
                        struct devfs_dirent,
                        struct label);
```

参数	说明	定义
dev	devfs_dirent 的 dev	
devfs_dirent	将被关联的 Devfs 目录	
label	将被填写的 devfs_dirent 的 label	

入口新建的 devfs_dirent 填写。函数将在文件系统加、重或添加新设备被调用。

6.7.3.1.5. mpo_create_devfs_directory

void

```
mpo_create_devfs_directory (char *dirname,  
                             int dirnamelen,  
                             struct devfs_dirent,  
                             struct label);
```

参数	说明	定义
dirname	新建目录的名字	
namelen	字符串 dirname 的长度	
devfs_dirent	新建目录在 Devfs 中的父目录	

输入目录参数的新建 devfs_dirent 填写。该函数将在添加、重命名文件系统，或者添加一个需要指定目录的新目录被调用。

6.7.3.1.6. mpo_create_devfs_symlink

void

```
mpo_create_devfs_symlink (struct ucred,  
                           struct mount,  
                           struct devfs_dirent,  
                           struct label,  
                           struct devfs_dirent,  
                           struct label);
```

参数	说明	定义
cred	主体信任状	
mp	devfs 挂载点	
dd	链接目录	
ddlabeled	与 dd 相关的目录	
de	符号链接	
delabeled	与 de 相关的策略	

新近创建的 devfs(5) 符号链接填写 (delabeled)。

6.7.3.1.7. mpo_create_vnode_extattr

int

```
mpo_create_vnode_extattr (struct ucred,  
                           struct mount,  
                           struct label,  
                           struct vnode,  
                           struct label,  
                           struct vnode,  
                           struct label,  
                           struct componentname);
```

参数	说明	定义
cred	主体信任状	
mount	文件系统挂载点	
label	文件系统标签	
dvp	父目录的 vnode	
dlabel	与 dvp 相关的策略	
vp	新建的 vnode	
vlabel	与 vp 相关的策略	
cnp	vp 中的子域名字	

将 vp 的标签写入文件扩展属性。成功，将策略填入 vlabel，并返回 0。否则，返回错误的errno。

6.7.3.1.8. mpo_create_mount

void

```
mpo_create_mount (struct ucred,  
                  struct mount,  
                  struct label,  
                  struct label);
```

参数	说明	定义
cred	主体信任状	
mp	客体；将被挂载的文件系统	
mntlabel	将被填写的 mp 的策略	
fslabel	将被挂载到 mp 的文件系统的策略	

传入的主体信任状所建立的挂载点填写策略。该函数将在文件系统挂载时被调用。

6.7.3.1.9. mpo_create_root_mount

void

```
mpo_create_root_mount (struct ucred,  
                        struct mount,  
                        struct label,  
                        struct label);
```

参数

说明

定义

mpo_create_mount.

入的主体信任状所建的挂点填写。函数将在挂根文件系统， mpo_create_mount; 之后被用。

6.7.3.1.10. mpo_relabel_vnode

void

```
mpo_relabel_vnode (struct ucred,  
                   struct vnode,  
                   struct label,  
                   struct label);
```

参数

说明

定义

cred

主体信任状

vp

将被重新的 vnode

vnode_label

vp 有的策略

new_label

将取代vnode_label的新（可能只是部分）

根据入的新和主体信任状，更新参数 vnode 的。

6.7.3.1.11. mpo_setlabel_vnode_extattr

int

```
mpo_setlabel_vnode_extattr (struct ucred,  
                             struct vnode,  
                             struct label,  
                             struct label);
```

参数

说明

定义

cred

主体信任状

vp

写出所的 vnode

vlabel

vp的策略

int_label

将被写入磁的

将参数 `intlabel` 输出的信息写入指定 `vnode` 的扩展属性。该函数被 `vop_stdcreatevnode_ea` 所调用。

6.7.3.1.12. `mpo_update_devfsdirent`

```
void
    mpo_update_devfsdirent (struct devfs_dirent,
                           struct label,
                           struct vnode,
                           struct label);
```

参数	说明	定义
<code>devfs_dirent</code>	客体；devfs 目录	
<code>direntlabel</code>	将被更新的devfs_dirent的策略	
<code>vp</code>	父 <code>vnode</code>	已定义
<code>vnode_label</code>	<code>vp</code> 的策略	

根据所输入的 `devfs_vnode` 对象，对 `devfs_dirent` 的条目进行更新。重新创建一个 `devfs_vnode` 的操作成功之后，将调用函数来对条目进行更改，如此，即使相关的 `vnode` 数据对象被内核回收重用，也不会丢失新的状态。另外，在 `devfs` 中新建一个符号链接，紧接着`mac_vnode_create_from_vnode`，也将调用函数，对 `vnode` 条目进行初始化操作。

6.7.3.2. IPC 对象事件操作

6.7.3.2.1. `mpo_create_mbuf_from_socket`

```
void
    mpo_create_mbuf_from_socket (struct socket,
                                struct label,
                                struct mbuf *m,
                                struct label);
```

参数	说明	定义
<code>socket</code>	套接字	套接字定义 WIP
<code>socket_label</code>	<code>socket</code> 的策略	
<code>m</code>	客体；mbuf	
<code>mbuf_label</code>	将被填写的 <code>m</code> 的策略	

根据输入的套接字创建新的mbuf对象并初始化。当套接字产生一个新的数据或者消息，并将其存储在参数 `mbuf` 中时，将调用函数。

6.7.3.2.2. `mpo_create_pipe`

void

```
mpo_create_pipe (struct ucred,  
                 struct pipe,  
                 struct label);
```

参数	说明	定义
cred	主体信任状	
pipe	管道	
pipelabel	pipe 的策略	

根据传入的主体信任状参数，设置新建管道的策略。当一个新管道被创建，该函数将被调用。

6.7.3.2.3. mpo_create_socket

void

```
mpo_create_socket (struct ucred,  
                  struct socket,  
                  struct label);
```

参数	说明	定义
cred	主体信任状	不可改
so	客体；将被填充的套接字	
socketlabel	将被填写的 so 的策略	

根据传入的主体信任状参数，设置新建套接字的策略。当新建一个套接字，该函数将被调用。

6.7.3.2.4. mpo_create_socket_from_socket

void

```
mpo_create_socket_from_socket (struct socket,  
                               struct label,  
                               struct socket,  
                               struct label);
```

参数	说明	定义
oldsocket	监听套接字	
oldsocketlabel	oldsocket 的策略	
newsocket	新建套接字	
newsocketlabel	newsocket 的策略	

根据 [listen\(2\)](#) 套接字 oldsocket，新建 [accept\(2\)](#) 的套接字 newsocket，设置策略。

6.7.3.2.5. mpo_relabel_pipe

```
void
    mpo_relabel_pipe (struct ucred,
                      struct pipe,
                      struct label,
                      struct label);
```

参数	说明	定义
cred	主体信任状	
pipe	管道	
oldlabel	pipe 的当前策略	
newlabel	将pipe 设置的新的策略	

pipe设置新newlabel。

6.7.3.2.6. mpo_relabel_socket

```
void
    mpo_relabel_socket (struct ucred,
                        struct socket,
                        struct label,
                        struct label);
```

参数	说明	定义
cred	主体信任状	不可改
so	客体；套接字	
oldlabel	so 的当前	
newlabel	将socket 设置的新	

根据入的参数，套接字的当前行更新。

6.7.3.2.7. mpo_set_socket_peer_from_mbuf

```
void
    mpo_set_socket_peer_from_mbuf (struct mbuf,
                                    struct label,
                                    struct label,
                                    struct label);
```

参数	说明	定义
mbuf	从套接字接收到的第一个数据	

参数	说明	定义
mbuflabel	mbuf 的	
oldlabel	套接字的当前	
newlabel	将套接字填写的策略	

根据传入的 mbuf 指针，设置某个 stream 套接字的标志。除Unix域的套接字之外，当一个 stream 套接字接收到第一个数据时，该函数将被调用。

6.7.3.2.8. mpo_set_socket_peer_from_socket

```
void
    mpo_set_socket_peer_from_socket (struct socket,
                                     struct label,
                                     struct socket,
                                     struct label);
```

参数	说明	定义
oldsocket	本地套接字	
oldsocketlabel	oldsocket 的策略	
newsocket	等套接字	
newsocketpeerlabel	将newsocket填写的策略	

根据传入的远程套接字端点，将一个 stream UNIX 与套接字设置相等。当相等的套接字之间进行连接，该函数将在端分被调用。

6.7.3.3. Network Object Labeling Event Operations

6.7.3.3.1. mpo_create_bpfdesc

```
void
    mpo_create_bpfdesc (struct ucred,
                        struct bpf_d,
                        struct label);
```

参数	说明	定义
cred	主体信任状	不可改
bpf_d	客体；bpf 描述子	
bpf	将bpf_d填写的策略	

根据传入的主体信任状参数，新建的 BPF 描述子设置。当程序打 BPF 断点时，该函数将被调用。

6.7.3.3.2. mpo_create_ifnet

```
void
    mpo_create_ifnet (struct ifnet,
                      struct label);
```

参数	说明	定义
ifnet	网口接口	
ifnetlabel	将ifnet填写的策略	

新建的网口接口置。函数在以下情况下被用： 当一个新的物理接口可用，或者当一个接口在引或由于某个用操作而例化。

6.7.3.3.3. mpo_create_ipq

```
void
    mpo_create_ipq (struct mbuf,
                    struct label,
                    struct ipq,
                    struct label);
```

参数	说明	定义
fragment	第一个被接收的 IP 分片	
fragmentlabel	fragment 的策略	
ipq	将被的 IP 重列	
ipqlabel	将ipq填写的策略	

根据第一个接收到的分片的 mbuf 部信息，新建的 IP 分片重列置。

6.7.3.3.4. mpo_create_datagram_from_ipq

```
void
    mpo_create_create_datagram_from_ipq (struct ipq,
                                          struct label,
                                          struct mbuf,
                                          struct label);
```

参数	说明	定义
ipq	IP 重列	
ipqlabel	ipq 的策略	
datagram	将被的数据	
datagramlabel	将datagramlabel填写的策略	

根据 IP 分片重排列，重新重完的 IP 数据设置。

6.7.3.3.5. mpo_create_fragment

```
void
    mpo_create_fragment (struct mbuf,
                        struct label,
                        struct mbuf,
                        struct label);
```

参数	说明	定义
datagram	数据	
datagramlabel	datagram 的策略	
fragment	将被的分片	
fragmentlabel	将datagram填写的策略	

根据数据所 mbuf 部信息，其新建的分片的 mbuf 部设置。

6.7.3.3.6. mpo_create_mbuf_from_mbuf

```
void
    mpo_create_mbuf_from_mbuf (struct mbuf,
                              struct label,
                              struct mbuf,
                              struct label);
```

参数	说明	定义
oldmbuf	已有的（源） mbuf	
oldmbuflabel	oldmbuf 的策略	
newmbuf	将被的新建 mbuf	
newmbuflabel	将newmbuf填写的策略	

根据某个有数据的 mbuf 部信息，新建数据的 mbuf 部设置。在多种条件下将会用函数，比如，由于要求而重新分配某个 mbuf 。

6.7.3.3.7. mpo_create_mbuf_linklayer

```
void
    mpo_create_mbuf_linklayer (struct ifnet,
                              struct label,
                              struct mbuf,
                              struct label);
```

参数	说明	定义
ifnet	网口	
ifnetlabel	ifnet 的策略	
mbuf	新建数据的 mbuf 部	
mbuflabel	将mbuf填写的策略	

在指定接口上由于某个路由而新建的数据的mbuf部置。 函数将在若干条件下被用，比如当IPv4和IPv6在ARP或者ND6。

6.7.3.3.8. mpo_create_mbuf_from_bpfdesc

```
void
mpo_create_mbuf_from_bpfdesc (struct bpf_d,
                               struct label,
                               struct mbuf,
                               struct label);
```

参数	说明	定义
bpf_d	BPF 描述子	
bpflabel	bpflabel 的策略	
mbuf	将被的新建 mbuf	
mbuflabel	将mbuf填写的策略	

使用参数 BPF 描述子建的新数据的 mbuf 部置。 当参数 BPF 描述子所的 BPF 行写操作，函数将被用。

6.7.3.3.9. mpo_create_mbuf_from_ifnet

```
void
mpo_create_mbuf_from_ifnet (struct ifnet,
                             struct label,
                             struct mbuf,
                             struct label);
```

参数	说明	定义
ifnet	网口	
ifnetlabel	ifnetlabel 的策略	
mbuf	新建数据的 mbuf 部	
mbuflabel	将mbuf填写的策略	

从网口参数建的数据的 mbuf 部置。

6.7.3.3.10. mpo_create_mbuf_multicast_encap

```
void
    mpo_create_mbuf_multicast_encap (struct mbuf,
                                     struct label,
                                     struct ifnet,
                                     struct label,
                                     struct mbuf,
                                     struct label);
```

参数	说明	定义
oldmbuf	有数据的 mbuf 部	
oldmbuflabel	oldmbuf 的策略	
ifnet	网接口	
ifnetlabel	ifnet 的策略	
newmbuf	将被的新建数据 mbuf 部	
newmbuflabel	将newmbuf填写的策略	

当入的已有数据被定多播封装接口（multicast encapsulation interface）理被用， 新建的数据所在 mbuf 部置。 当使用虚接口一个mbuf， 将用函数。

6.7.3.3.11. mpo_create_mbuf_netlayer

```
void
    mpo_create_mbuf_netlayer (struct mbuf,
                              struct label,
                              struct mbuf,
                              struct label);
```

参数	说明	定义
oldmbuf	接收的数据	
oldmbuflabel	oldmbuf 的策略	
newmbuf	新建数据	
newmbuflabel	newmbuf 的策略	

由 IP 堆因接收数据（oldmbuf）而新建的数据置其 mbuf 部的。 多情况下需要用函数， 比如， ICMP 求数据。

6.7.3.3.12. mpo_fragment_match

```
int
    mpo_fragment_match (struct mbuf,
                        struct label,
                        struct ipq,
                        struct label);
```

参数	说明	定义
fragment	IP 数据分片	
fragmentlabel	fragment 的策略	
ipq	IP 分片重传列	
ipqlabel	ipq 的策略	

根据所输入的 IP 分片重传列 (ipq) 的， 包含一个 IP 数据 (fragment) 的 mbuf 的 部是否符合其要求。 符合， 返回1。 否， 返回0。 当 IP 堆 将一个 接收到的分片放入某个已有的分片重传列中， 将 用 函数 行安全 ； 如果失， 将 分片重新 例化一个新的分片重传列。 策略可以利用 入口函数， 根据 或者其他信息阻止不期望的 IP 分片重。

6.7.3.3.13. mpo_relabel_ifnet

```
void
    mpo_relabel_ifnet (struct ucred,
                      struct ifnet,
                      struct label,
                      struct label);
```

参数	说明	定义
cred	主体信任状	
ifnet	客体；网接口	
ifnetlabel	ifnet 的策略	
newlabel	将 ifnet 置的新	

根据所输入的新， newlabel， 以及主体信任状， cred， 网接口的 行更新。

6.7.3.3.14. mpo_update_ipq

```
void
    mpo_update_ipq (struct mbuf,
                   struct label,
                   struct ipq,
                   struct label);
```

参数	说明	定义
mbuf	IP 分片	
mbuflabel	mbuf 的策略	
ipq	IP 分片重排序	
ipqlabel	将被更新的ipq的当前策略	

根据所输入的 IP 分片 mbuf 首部 (mbuf) 接收 它的 IP 分片重排序 (ipq) 的表行更新。

6.7.3.4. 进程事件操作

6.7.3.4.1. mpo_create_cred

```
void
mpo_create_cred (struct ucred,
                 struct ucred);
```

参数	说明	定义
parent_cred	父主体信任状	
child_cred	子主体信任状	

根据所输入的主体信任状，新建的主体信任状置。 当一个新建的 struct ucred 用 [crcopy\(9\)](#) 时，将用此函数。 函数不与进程制 (forking) 或者建事件混。

6.7.3.4.2. mpo_execve_transition

```
void
mpo_execve_transition (struct ucred,
                      struct ucred,
                      struct vnode,
                      struct label);
```

参数	说明	定义
old	已有的主体信任状	不可改
new	将被的新主体信任状	
vp	将被行的文件	已被定
vnode_label	vp 的策略	

一个有信任状old的主体由于行(vp文件而致， 函数根据vnode主体重新new。 当一个进程求行vnode文件，而通过 入口函数mpo_execve_will_transition 有成功返回的策略，将用函数。策略模式可以通过个主体信任状和地用 mpo_create_cred 来入口函数， so as not to implement a transitioning event. 一旦策略了mpo_create_cred函数，即使没有 mpo_execve_will_transition，也函数。

6.7.3.4.3. mpo_execve_will_transition

```
int
mpo_execve_will_transition (struct ucred,
                           struct vnode,
                           struct label);
```

参数	说明	定义
old	在行execve(2)之前的主体信任状	不可改
vp	将被行的文件	
vnode_label	vp 的策略	

由策略决定，当参数主体信任状行参数 vnode ，是否需要行一个操作。如果需要，返回1； 否，返回0。即使一个策略返回0，它也必自己不期望的 mpo_execve_transition的用作好准，因只要有其他任何一个策略要求，就将行此函数。

6.7.3.4.4. mpo_create_proc0

```
void
mpo_create_proc0 (struct ucred);
```

参数	说明	定义
cred	将被填写的主体信任状	

程0，所有内核程的祖先，建主体信任状。

6.7.3.4.5. mpo_create_proc1

```
void
mpo_create_proc1 (struct ucred);
```

参数	说明	定义
cred	将被填写的主体信任状	

程1，所有用程的祖先，建主体信任状。

6.7.3.4.6. mpo_relabel_cred

```
void
mpo_relabel_cred (struct ucred,
                  struct label);
```

参数	说明	定义
cred	主体信任状	
newlabel	将被用到 cred 上的新	

根据入的新，主体信任状上的行更新。

6.7.4. 控制

通控制入口函数，策略模能影内核的控制决策。通常情况下，不是，一个控制入口函数的参数有，一个或者若干个授信任状，和相操作及其他任何象的信息（其中可能包含）。控制入口函数返回0，表示允操作；否，返回一个 [errno\(2\)](#) 。用入口函数，将遍所有系注册的策略模，逐一行策略相的和决策，之后按照下述方法合不同策略的返回果：只有当所有的模均允操作，才成功返回。否，如果有一个或者若干模失返回，整个不通。如果有多个模的出返回，将由定在kern_mac.c 中的 [error_select\(\)](#) 函数从它返回的列表中，一个合的，返回用。

最高先	EDEADLK
	EINVAL
	ESRCH
	EACCES
最低先	EPERM

如果所有策略模返回的列表均没有出在上述先序列表中，任意一个返回。的一般次序：内核，无效的参数，象不存在，被拒，和其他。

6.7.4.1. mpo_check_bpfdesc_receive

```
int
    mpo_check_bpfdesc_receive (struct bpf_d,
                               struct label,
                               struct ifnet,
                               struct label);
```

参数	说明	定义
bpf_d	主体；BPF 描述子	
bpflabel	bpf_d 的策略	
ifnet	客体；网接口	
ifnetlabel	ifnet 的策略	

决定 MAC 框架是否允将由参数接口接收到的数据由 BPF 描述子所的缓冲区。成功，返回0；否，返回信息[errno](#)。建使用的有：EACCES，用于不符的情况；EPERM，用于缺少特情况。

6.7.4.2. mpo_check_kenv_dump

```
int  
    mpo_check_kenv_dump (struct ucred);
```

参数	说明	定义
cred	主体信任状	

决定相主体是否被允许内核环境 (参考 [kenv\(2\)](#))。

6.7.4.3. mpo_check_kenv_get

```
int  
    mpo_check_kenv_get (struct ucred,  
                        char *name);
```

参数	说明	定义
cred	主体信任状	
name	内核的环境变量名字	

决定相主体是否可以内核中定义环境变量的状态。

6.7.4.4. mpo_check_kenv_set

```
int  
    mpo_check_kenv_set (struct ucred,  
                        char *name);
```

参数	说明	定义
cred	主体信任状	
name	内核的环境变量名字	

决定相主体是否有设置内核环境变量的。

6.7.4.5. mpo_check_kenv_unset

```
int  
    mpo_check_kenv_unset (struct ucred,  
                           char *name);
```

参数	说明	定义
cred	主体信任状	

参数	说明	定义
name	内核的环境变量名字Kernel environment variable name	

决定相主体是否有清除定义的内核环境变量的位置。

6.7.4.6. mpo_check_kld_load

```
int
    mpo_check_kld_load (struct ucred,
                        struct vnode,
                        struct label);
```

参数	说明	定义
cred	主体信任状	
vp	内核模块的 vnode	
vlabel	vp的策略	

决定相主体是否有加载定义的模式文件。

6.7.4.7. mpo_check_kld_stat

```
int
    mpo_check_kld_stat (struct ucred);
```

参数	说明	定义
cred	主体信任状	

决定相主体是否有内核的加载模式文件表以及相的数据。

6.7.4.8. mpo_check_kld_unload

```
int
    mpo_check_kld_unload (struct ucred);
```

参数	说明	定义
cred	主体信任状	

决定相主体是否有卸一个内核模块。

6.7.4.9. mpo_check_pipe_ioctl

int

```
mpo_check_pipe_ioctl (struct ucred,  
                      struct pipe,  
                      struct label,  
                      unsigned long,  
                      void *data);
```

参数	说明	定义
cred	主体信任状	
pipe	管道	
pipelabel	pipe的策略	
cmd	ioctl(2) 命令	
data	ioctl(2) 数据	

决定相主体是否有用指定的 [ioctl\(2\)](#) 系用。

6.7.4.10. mpo_check_pipe_poll

int

```
mpo_check_pipe_poll (struct ucred,  
                    struct pipe,  
                    struct label);
```

参数	说明	定义
cred	主体信任状	
pipe	管道	
pipelabel	pipe的策略	

决定相主体是否有管道pipe行poll操作。

6.7.4.11. mpo_check_pipe_read

int

```
mpo_check_pipe_read (struct ucred,  
                    struct pipe,  
                    struct label);
```

参数	说明	定义
cred	主体信任状	
pipe	管道	
pipelabel	pipe的策略	

决定主体是否有取pipe。

6.7.4.12. mpo_check_pipe_relabel

```
int
    mpo_check_pipe_relabel (struct ucred,
                           struct pipe,
                           struct label,
                           struct label);
```

参数	明	定
cred	主体信任状	
pipe	管道	
pipelabel	pipe的当前策略	
newlabel	将pipelabel置的新	

决定主体是否有pipe重新置。

6.7.4.13. mpo_check_pipe_stat

```
int
    mpo_check_pipe_stat (struct ucred,
                         struct pipe,
                         struct label);
```

参数	明	定
cred	主体信任状	
pipe	管道	
pipelabel	pipe的策略	

决定主体是否有与pipe相的信息。

6.7.4.14. mpo_check_pipe_write

```
int
    mpo_check_pipe_write (struct ucred,
                          struct pipe,
                          struct label);
```

参数	明	定
cred	主体信任状	
pipe	管道	

参数	说明	定义
pipelabel	pipe的策略	

决定主体是否有写pipe。

6.7.4.15. mpo_check_socket_bind

```
int
    mpo_check_socket_bind (struct ucred,
                          struct socket,
                          struct label,
                          struct sockaddr);
```

参数	说明	定义
cred	主体信任状	
socket	将被绑定的套接字	
socketlabel	socket的策略	
sockaddr	socket的地址	

6.7.4.16. mpo_check_socket_connect

```
int
    mpo_check_socket_connect (struct ucred,
                             struct socket,
                             struct label,
                             struct sockaddr);
```

参数	说明	定义
cred	主体信任状	
socket	将被连接的套接字	
socketlabel	socket的策略	
sockaddr	socket的地址	

决定主体（cred）是否有将套接字（socket）绑定到地址 sockaddr。成功，返回0，否则返回一个 errno。建议采用的有：EACCES，用于不符的情况；EPERM，用于特权不足的情况。

6.7.4.17. mpo_check_socket_receive

```
int
    mpo_check_socket_receive (struct ucred,
                             struct socket,
                             struct label);
```

参数	说明	定义
cred	主体信任状	
so	套接字	
socketlabel	so的策略	

决定主体是否有套接字so的相关信息。

6.7.4.18. mpo_check_socket_send

```
int
    mpo_check_socket_send (struct ucred,
                          struct socket,
                          struct label);
```

参数	说明	定义
cred	主体信任状	
so	套接字	
socketlabel	so的策略	

决定主体是否有通套接字so送信息。

6.7.4.19. mpo_check_cred_visible

```
int
    mpo_check_cred_visible (struct ucred,
                          struct ucred);
```

参数	说明	定义
u1	主体信任状	
u2	象信任状	

决定主体信任状u1是否有 "see" 具有信任状u2 的其他主体。 成功，返回0；否，返回errno。建采用的有： EACCES，用于不符的情况；EPERM，用于特不足的情况；ESRCH， 用来提供不可性。函数可在多境下使用，包括命令ps所使用的程的状态 sysctl，以及通procfs 的状态操作。

6.7.4.20. mpo_check_socket_visible

```
int
    mpo_check_socket_visible (struct ucred,
                          struct socket,
                          struct label);
```

参数	说明	定义
cred	主体信任状	
socket	客体；套接字	
socketlabel	socket的策略	

6.7.4.21. mpo_check_ifnet_relabel

```
int
    mpo_check_ifnet_relabel (struct ucred,
                             struct ifnet,
                             struct label,
                             struct label);
```

参数	说明	定义
cred	主体信任状	
ifnet	客体；网口	
ifnetlabel	ifnet有的策略	
newlabel	将被用到ifnet上的新的策略	

决定主体信任状是否有使用入的参数更新参数定的网口的行重新置。

6.7.4.22. mpo_check_socket_relabel

```
int
    mpo_check_socket_relabel (struct ucred,
                              struct socket,
                              struct label,
                              struct label);
```

参数	说明	定义
cred	主体信任状	
socket	客体；套接字	
socketlabel	socket有的策略	
newlabel	将被用到socketlabel上的更新	

决定主体信任状是否有采用入的套接字参数的行重新置。

6.7.4.23. mpo_check_cred_relabel

int

```
mpo_check_cred_relabel (struct ucred,  
                        struct label);
```

参数	说明	定义
cred	主体信任状	
newlabel	将被用到cred上的更新	

决定主体信任状是否有将自己的重新置定义的更新。

6.7.4.24. mpo_check_vnode_relabel

int

```
mpo_check_vnode_relabel (struct ucred,  
                        struct vnode,  
                        struct label,  
                        struct label);
```

参数	说明	定义
cred	主体信任状	不可改
vp	客体；vnode	已被定
vnode_label	vp有的策略	
newlabel	将被用到vp上的策略	

决定主体信任状是否有将参数 vnode 的重新置指定。

6.7.4.25. mpo_check_mount_stat

```
int mpo_check_mount_stat (struct ucred,  
                        struct mount,  
                        struct label);
```

参数	说明	定义
cred	主体信任状	
mp	客体；文件系统挂	
mount_label	mp的策略	

定相主体信任状是否有看定文件系统上行 statfs 的结果。成功，返回 0；否，返回一个errno。建使用的：EACCES，用于不匹配； EPERM，用于限不。函数可能在下列情况下被用：在 statfs(2) 和其他相用期，或者当需要从文件系统列表中排除个文件系统，比如，用 getfsstat(2)。

6.7.4.26. mpo_check_proc_debug

```
int
    mpo_check_proc_debug (struct ucred,
                          struct proc);
```

参数	说明	定义
cred	主体信任状	不可改
proc	客体；进程	

定义相主体信任状是否有 debug 定义进程。成功，返回 0；否则，返回一个 `errno`。定义使用的标志：`EACCES`，用于不匹配；`EPERM`，用于权限不；`ESRCH`，用于目标的存在。 `ptrace(2)` 和 `ktrace(2)` API，以及某些 `procfs` 操作将用函数。

6.7.4.27. mpo_check_vnode_access

```
int
    mpo_check_vnode_access (struct ucred,
                           struct vnode,
                           struct label,
                           int flags);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp的策略	
flags	<code>access(2)</code> 标志	

根据相主体信任状决定其定义 vnode 以定义标志行的 `access(2)` 和其他相用的返回值。一般，采用与 `mpo_check_vnode_open` 相同的用来函数。成功，返回 0；否则，返回一个 `errno`。定义使用的标志：`EACCES`，用于不匹配；`EPERM`，用于权限不。

6.7.4.28. mpo_check_vnode_chdir

```
int
    mpo_check_vnode_chdir (struct ucred,
                          struct vnode,
                          struct label);
```

参数	说明	定义
cred	主体信任状	
dvp	客体； <code>chdir(2)</code> 的目的 vnode	

参数	说明	定义
dlabel	dvp的策略	

定义相主体信任状是否有将程工作目切到定 vnode。成功，返回 0；否，返回一个 **errno**。建使用的：EACCES，用于不匹配； EPERM，用于限不。

6.7.4.29. **mpo_check_vnode_chroot**

```
int
    mpo_check_vnode_chroot (struct ucred,
                           struct vnode,
                           struct label);
```

参数	说明	定义
cred	主体信任状	
dvp	目 vnode	
dlabel	与dvp相的策略	

定义相主体是否有 **chroot(2)** 到由 (dvp)定的目。

6.7.4.30. **mpo_check_vnode_create**

```
int
    mpo_check_vnode_create (struct ucred,
                           struct vnode,
                           struct label,
                           struct componentname,
                           struct vattr);
```

参数	说明	定义
cred	主体信任状	
dvp	客体；vnode	
dlabel	dvp的策略	
cnp	dvp中的成名	
vap	vap的 vnode 属性	

定义相主体信任状是否有在定父目，以定的名字和属性，常一个 vnode。成功，返回 0；否，返回一个**errno**。建使用的：EACCES 来表示用于不匹配，而用 EPERM，用于限不足。以O_CREAT参数用 **open(2)**，或 **mknod(2)**， **mkfifo(2)** 等的用将致函数被用。

6.7.4.31. **mpo_check_vnode_delete**

int

```
mpo_check_vnode_delete (struct ucred,  
                        struct vnode,  
                        struct label,  
                        struct vnode,  
                        void *label,  
                        struct componentname);
```

参数	说明	定义
cred	主体信任状	
dvp	父目 vnode	
dlabel	dvp 的策略	
vp	客体；将被删除的 vnode	
label	vp 的策略	
cnp	vp 中的成名字	

定义相主体信任状是否有从定义的父母中，删除名字 vnode。成功，返回 0；否，返回一个 `errno`。建使用的：EACCES，用于不匹配； EPERM，用于权限不。使用 [unlink\(2\)](#) 和 [rmdir\(2\)](#)，将致函数被用。提供入口函数的策略必须一个 `mpo_check_rename_to`，用来授由于重命名操作致的目文件的除。

6.7.4.32. mpo_check_vnode_deleteacl

int

```
mpo_check_vnode_deleteacl (struct ucred *cred,  
                           struct vnode *vp,  
                           struct label *label,  
                           acl_type_t type);
```

参数	说明	定义
cred	主体信任状	不可改
vp	客体；vnode	被定
	label	vp 的策略
	type	ACL 型

定义相主体信任状是否有除定 vnode 的定型的 ACL。成功，返回 0；否，返回一个 `errno`。建使用的：EACCES，用于不匹配； EPERM，用于权限不。

6.7.4.33. mpo_check_vnode_exec

```
int
    mpo_check_vnode_exec (struct ucred,
                          struct vnode,
                          struct label);
```

参数	说明	定义
cred	主体信任状	
vp	客体；将被执行的 vnode	
label	vp的策略	

定义相主体信任状是否有行定义 vnode。 于行特行的决策与任何瞬事件的决策是格分的。 成功，返回 0；否，返回一个errno。 建使用的：EACCES，用于不匹配； EPERM，用于限不。

6.7.4.34. mpo_check_vnode_getacl

```
int
    mpo_check_vnode_getacl (struct ucred,
                           struct vnode,
                           struct label,
                           acl_type_t);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp的策略	
type	ACL 型	

定义相主体信任状是否有定 vnode 上的定型的 ACL。 成功，返回 0；否，返回一个errno。 建使用的：EACCES，用于不匹配； EPERM，用于限不。

6.7.4.35. mpo_check_vnode_getextattr

```
int
    mpo_check_vnode_getextattr (struct ucred,
                                struct vnode,
                                struct label,
                                int,
                                const char,
                                struct uio);
```

参数	说明	定义
cred	主体信任状	

参数	说明	定义
vp	客体；vnode	
label	vp的策略	
attrnamespace	扩展属性名字空间	
name	扩展属性名	
uio	I/O 指针；参看 uio(9)	

定义相主体信任状是否有扩展vnode上定义名字空间和名字的扩展属性。使用扩展属性存储的策略模式可能会需要一些扩展属性的操作行特殊处理。成功，返回 0；否则，返回一个 `errno`。建立使用的权限：EACCES，用于不匹配；EPERM，用于限制不。

6.7.4.36. `mpo_check_vnode_link`

```
int
mpo_check_vnode_link (struct ucred,
                      struct vnode,
                      struct label,
                      struct vnode,
                      struct label,
                      struct componentname);
```

参数	说明	定义
cred	主体信任状	
dvp	目标 vnode	
dlabel	与dvp相关的策略	
vp	目标目的 vnode	
label	与vp相关的策略	
cnp	将被建立的链接的组件名	

定义相主体是否有参数vp定义vnode建立一个由参数cnp定义名字的链接。

6.7.4.37. `mpo_check_vnode_mmap`

```
int
mpo_check_vnode_mmap (struct ucred,
                      struct vnode,
                      struct label,
                      int prot);
```

参数	说明	定义
cred	主体信任状	

参数	说明	定义
vp	将被映射的 vnode	
label	与vp相同的策略	
prot	mmap 保护 (参 mmap(2))	

定义相主体是否有将定义 vnode vp 以 prot指定的保护方式行映射。

6.7.4.38. mpo_check_vnode_mmap_downgrade

```
void
    mpo_check_vnode_mmap_downgrade (struct ucred,
                                     struct vnode,
                                     struct label,
                                     int *prot);
```

参数	说明	定义
cred	See mpo_check_vnode_mmap .	
vp		label
		prot

根据主体和客体，降低 mmap protections。

6.7.4.39. mpo_check_vnode_mprotect

```
int
    mpo_check_vnode_mprotect (struct ucred,
                              struct vnode,
                              struct label,
                              int prot);
```

参数	说明	定义
cred	主体信任状	
vp	映射的 vnode	
prot	存保护	

定义相主体是否有将定义 vnode vp 映射内存空间的存保护参数置指定。

6.7.4.40. mpo_check_vnode_poll

int

```
mpo_check_vnode_poll (struct ucred,  
                      struct ucred,  
                      struct vnode,  
                      struct label);
```

参数	说明	定义
active_cred	主体信任状	
file_cred	与struct file相联的信任状	
vp	将被行 poll 操作的 vnode	
label	与vp相联的策略	

定义相联主体是否有定义 vnode vp行 poll 操作。

6.7.4.41. mpo_check_vnode_rename_from

int

```
mpo_vnode_rename_from (struct ucred,  
                      struct vnode,  
                      struct label,  
                      struct vnode,  
                      struct label,  
                      struct componentname);
```

参数	说明	定义
cred	主体信任状	
dvp	目标 vnode	
dlabel	与dvp相联的策略	
vp	将被重命名的 vnode	
label	与vp相联的策略	
cnp	vp中的成分名	

定义相联主体是否有重命名定义 vnode, vp。

6.7.4.42. mpo_check_vnode_rename_to

int

```
mpo_check_vnode_rename_to (struct ucred,  
                           struct vnode,  
                           struct label,  
                           struct vnode,  
                           struct label,  
                           int samedir,  
                           struct componentname);
```

参数	说明	定义
cred	主体信任状	
dvp	目标 vnode	
dlabel	与 dvp 相关的策略	
vp	被覆盖的 vnode	
label	与 vp 相关的策略	
samedir	布尔型量；如果源和目的目标是相同的，被置 1	
cnp	目标 component 名	

定义相主体是否有重命名目标 vnode vp，至指定目标 dvp，或更名 cnp。如果无需覆盖已有文件，vp 和 label 的将 NULL。

6.7.4.43. mpo_check_socket_listen

int

```
mpo_check_socket_listen (struct ucred,  
                        struct socket,  
                        struct label);
```

参数	说明	定义
cred	主体信任状	
socket	客体；套接字	
socketlabel	socket 的策略	

定义相主体是否有监听套接字。成功，返回 0；否则，返回 errno。建立使用的标志：
EACCES，用于不匹配；EPERM，用于权限不。

6.7.4.44. mpo_check_vnode_lookup

int

```
mpo_check_vnode_lookup (struct ucred,  
                        struct vnode,  
                        struct label,  
                        struct componentname);
```

参数	说明	定义
cred	主体信任状	
dvp	客体；vnode	
dlabel	dvp的策略	
cnp	被的成	名

定义相主体信任状是否有在指定的目 vnode 中指定名字行lookup操作。 成功，返回 0；否则，返回一个 errno。建使用的：EACCES，用于不匹配； EPERM，用于限不。

6.7.4.45. mpo_check_vnode_open

int

```
mpo_check_vnode_open (struct ucred,  
                      struct vnode,  
                      struct label,  
                      int);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp的策略	
acc_mode	open(2) 模式	

定义相主体信任状是否有在指定 vnode 上以指定的模式行 open 操作。 如果成功，返回 0；否则，返回一个 errno。建使用的：EACCES，用于不匹配； EPERM，用于限不。

6.7.4.46. mpo_check_vnode_readdir

int

```
mpo_check_vnode_readdir (struct ucred,  
                         struct vnode,  
                         struct label);
```

参数	说明	定义
cred	主体信任状	

参数	说明	定义
dvp	客体；目标 vnode	
dlabel	dvp 的策略	

定义相主体信任状是否有在定义的目标 vnode 上执行 **readdir** 操作。成功，返回 0；否则，返回一个 **errno**。定义使用的 **errno**：EACCES，用于不匹配；EPERM，用于权限不足。

6.7.4.47. mpo_check_vnode_readlink

```
int
mpo_check_vnode_readlink (struct ucred,
                          struct vnode,
                          struct label);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp 的策略	

定义相主体信任状是否有在定义符号链接 vnode 上执行 **readlink** 操作。成功，返回 0；否则，返回一个 **errno**。定义使用的 **errno**：EACCES，用于不匹配；EPERM，用于权限不足。该函数可能在若干境下被调用，包括由用进程式执行的 **readlink** 调用，或者是在进程行名字表式执行的 **readlink**。

6.7.4.48. mpo_check_vnode_revoke

```
int
mpo_check_vnode_revoke (struct ucred,
                        struct vnode,
                        struct label);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp 的策略	

定义相主体信任状是否有撤销定义 vnode 的。成功，返回 0；否则，返回一个 **errno**。定义使用的 **errno**：EACCES，用于不匹配；EPERM，用于权限不足。

6.7.4.49. mpo_check_vnode_setacl

```
int
mpo_check_vnode_setacl (struct ucred,
                        struct vnode,
                        struct label,
                        acl_type_t,
                        struct acl);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp的策略	
type	ACL 类型	
acl	ACL	

定义相主体信任状是否有设置定义 vnode 的定义类型的 ACL。成功，返回 0；否则，返回一个 `errno`。建议使用的错误码：EACCES，用于不匹配；EPERM，用于权限不足。

6.7.4.50. `mpo_check_vnode_setextattr`

```
int
mpo_check_vnode_setextattr (struct ucred,
                            struct vnode,
                            struct label,
                            int,
                            const char,
                            struct uio);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp的策略	
attrnamespace	扩展属性名字空间	
name	扩展属性名	
uio	I/O 指针；参看 uio(9)	

定义相主体信任状是否有设置定义 vnode 上定义名字空间中定义名字的扩展属性的。使用扩展属性安全策略模型可能需要其使用的属性施加额外的保护。此外，由于在操作和可能存在的竞争，策略模型避免根据来自 uio 中的数据做出决策。如果正在执行一个删除操作，参数 uio 的也可能为 `NULL`。成功，返回 0；否则，返回一个 `errno`。建议使用的错误码：EACCES，用于不匹配；EPERM，用于权限不足。

6.7.4.51. mpo_check_vnode_setflags

```
int
    mpo_check_vnode_setflags (struct ucred,
                              struct vnode,
                              struct label,
                              u_long flags);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp的策略	
flags	文件标志；参看 chflags(2)	

定义相主体信任状是否有指定的 vnode 指定的标志。成功，返回 0；否则，返回一个errno。使用的标志：EACCES，用于不匹配； EPERM，用于权限不。

6.7.4.52. mpo_check_vnode_setmode

```
int
    mpo_check_vnode_setmode (struct ucred,
                              struct vnode,
                              struct label,
                              mode_t mode);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp的策略	
mode	文件模式；参看 chmod(2)	

定义相主体信任状是否有将指定的 vnode 的模式置指定。成功，返回 0；否则，返回一个errno。使用的标志：EACCES，用于不匹配； EPERM，用于权限不。

6.7.4.53. mpo_check_vnode_setowner

```
int
    mpo_check_vnode_setowner (struct ucred,
                               struct vnode,
                               struct label,
                               uid_t uid,
                               gid_t gid);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp的策略	
uid	用户ID	
gid	组ID	

定义相主体信任状是否有将vnode 的文件 uid 和文件 gid 置。如果无需更新，相参数可能被置(-1)。成功，返回 0；否，返回一个errno。建使用的：EACCES，用于不匹配；EPERM，用于限不。

6.7.4.54. mpo_check_vnode_setutimes

```
int
    mpo_check_vnode_setutimes (struct ucred,
                                struct vnode,
                                struct label,
                                struct timespec,
                                struct timespec);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vp	
label	vp的策略	
atime	；参 utimes(2)	
mtime	修改；参 utimes(2)	

定义相主体信任状是否有将vnode 的置。成功，返回 0；否，返回一个errno。建使用的：EACCES，用于不匹配；EPERM，用于限不。

6.7.4.55. mpo_check_proc_sched

```
int
    mpo_check_proc_sched (struct ucred,
                           struct proc);
```

参数	说明	定义
cred	主体信任状	
proc	客体；程	

定义相主体信任状是否有改定程的度参数。成功，返回 0；否，返回一个errno。建使用的：EACCES，用于不匹配；EPERM，用于限不；ESRCH，用于提供不可性。

See [setpriority\(2\)](#) for more information.

6.7.4.56. `mpo_check_proc_signal`

```
int
    mpo_check_proc_signal (struct ucred,
                          struct proc,
                          int signal);
```

参数	说明	定义
cred	主体信任状	
proc	客体；进程	
signal	信号；参见 kill(2)	

定义相主体信任状是否有向进程发送信号。成功，返回 0；否则，返回一个 `errno`。定义使用的错误码：EACCES，用于权限不匹配；EPERM，用于权限不足；ESRCH，用于提供不可达性。

6.7.4.57. `mpo_check_vnode_stat`

```
int
    mpo_check_vnode_stat (struct ucred,
                         struct vnode,
                         struct label);
```

参数	说明	定义
cred	主体信任状	
vp	客体；vnode	
label	vp的策略	

定义相主体信任状是否有在指定 vnode 上行 stat 操作。成功，返回 0；否则，返回一个 `errno`。定义使用的错误码：EACCES，用于权限不匹配；EPERM，用于权限不足。

See [stat\(2\)](#) for more information.

6.7.4.58. `mpo_check_ifnet_transmit`

```
int
    mpo_check_ifnet_transmit (struct ucred,
                             struct ifnet,
                             struct label,
                             struct mbuf,
                             struct label);
```

参数	说明	定义
cred	主体信任状	
ifnet	网口	
ifnetlabel	ifnet的策略	
mbuf	客体；将被送的 mbuf	
mbuflabel	mbuf的策略	

定义相网口是否有送的 mbuf。成功，返回 0；否，返回一个errno。建使用的：EACCES，用于不匹配；EPERM，用于限不。

6.7.4.59. mpo_check_socket_deliver

```
int
    mpo_check_socket_deliver (struct ucred,
                              struct ifnet,
                              struct label,
                              struct mbuf,
                              struct label);
```

参数	说明	定义
cred	主体信任状	
ifnet	网口	
ifnetlabel	ifnet的策略	
mbuf	客体；将被送的 mbuf	
mbuflabel	mbuf的策略	

定义相套接字是否有从的 mbuf 中接收数据。成功，返回 0；否，返回一个errno。建使用的：EACCES，用于不匹配；EPERM，用于限不。

6.7.4.60. mpo_check_socket_visible

```
int
    mpo_check_socket_visible (struct ucred,
                              struct socket,
                              struct label);
```

参数	说明	定义
cred	主体信任状	不可改
so	客体；套接字	
socketlabel	so的策略	

确定相主体信任状cred 是否有使用系控函数，比如，由netstat(8) 和 sockstat(1)使用的程序来察定的套接字(socket)。成功，返回 0；否，返回一个errno。建使用的：EACCES，用于不匹配； EPERM，用于限不； ESRCH，用于提供不可性。

6.7.4.61. mpo_check_system_acct

```
int
    mpo_check_system_acct (struct ucred,
                           struct vnode,
                           struct label);
```

参数	明	定
ucred	主体信任状	
vp	文件；acct(5)	
vlabel	与vp相的	

根据主体和日志文件的，确定主体是否有。

6.7.4.62. mpo_check_system_nfsd

```
int
    mpo_check_system_nfsd (struct ucred);
```

参数	明	定
cred	主体信任状	

确定相主体是否有用 nfssvc(2)。

6.7.4.63. mpo_check_system_reboot

```
int
    mpo_check_system_reboot (struct ucred,
                             int howto);
```

参数	明	定
cred	主体信任状	
howto	来自 reboot(2)的howto 参数	

确定相主体是否有以指定方式重系。

6.7.4.64. mpo_check_system_settime

int

```
mpo_check_system_settime (struct ucred);
```

参数	说明	定义
cred	主体信任状	

定义相应用是否有设置系。

6.7.4.65. mpo_check_system_swapon

int

```
mpo_check_system_swapon (struct ucred,  
                          struct vnode,  
                          struct label);
```

参数	说明	定义
cred	主体信任状	
vp	swap	
vlabel	与vp相的	

定义相主体是否有加一个作swap的 vp 。

6.7.4.66. mpo_check_system_sysctl

int

```
mpo_check_system_sysctl (struct ucred,  
                          int *name,  
                          u_int *namelen,  
                          void *old,  
                          size_t,  
                          int inkernel,  
                          void *new,  
                          size_t newlen);
```

参数	说明	定义
cred	主体信任状	
name	参 sysctl(3)	
namelen		
old		
oldlenp		

参数	说明	定义
inkernel	布尔型变量；如果从内核被引用，其被置为1	
new	参见 sysctl(3)	

定义相主体是否被允许进行指定的 [sysctl\(3\)](#) 事项。

6.7.5. 进程管理应用

当应用程序请求某个对象的属性进行修改时，将引发重新操作事件。对象的更新操作分两步进行：首先，进行控制操作，此次更新操作是有效且被允许的；然后，应用一个独立的入口函数对对象进行修改。重新入口函数通常接收由请求程序提交的对象、对象指针和请求新值，作为输入参数。对象重新操作的操作失败将由先前的操作报告，所以，不允许在接下来的修改过程中告失，故而不提倡在此过程中重新分配内存。

6.8. 应用体系

TrustedBSD MAC 框架包含了一组策略无关的构成元素，包括管理抽象的 MAC 接口，系统信任状管理体系的修改，应用分配 MAC 提供支持的 login 函数，以及若干网络和更新内核对象（进程、文件和网络接口等）安全管理的工具。不久，将有更多关于应用体系的信息被包含进来。

6.8.1. 策略无关的进程管理 API

TrustedBSD MAC 提供的大量函数和系统调用，允许应用程序使用一致的、策略无关的接口来管理对象的 MAC。如此，应用程序可以轻松管理各策略的对象，无需添加某个特定策略的支持而重新编译。许多通用工具，比如 [ifconfig\(8\)](#)、[ls\(1\)](#) 和 [ps\(1\)](#)，使用一些策略无关的接口来访问网络、文件和进程的 MAC 信息。一些 API 也被用于支持 MAC 管理工具，比如，[getfmac\(8\)](#)、[getpmac\(8\)](#)、[setfmac\(8\)](#)、[setfsmac\(8\)](#)，和 [setpmac\(8\)](#)。MAC API 的详细信息可参考 [mac\(3\)](#)。

应用程序管理的 MAC 对象有四种存在形式：内部形式，用来返回和设置进程和对象的 `(mac_t)`；基于 C 字符串的外部形式，作为在配置文件中的存放形式，用于向用户显示或者由用户输入。一个 MAC 对象由一个或多个元素组成，其中每个元素是一个形如（名字，值）的二元组。内核中的每个策略模块分配一个被指定一个特定的名字，由它自身的中与名字相关的采用其策略特有的方式进行解析。采用外部形式表示的对象，其元素表示为名字 / 值，元素之间以逗号分隔。应用程序可以使用 MAC 框架提供的 API 将一个安全对象在内部形式和文本形式之间进行转换。当向内核请求某个对象的安全信息时，内部形式的对象必须包含所需的元素集合作为内部存储准备。因此，通常采用下面两种方式之一：使用 [mac_prepare\(3\)](#) 和一个包含所需元素的任意列表；或者，使用从 [mac.conf\(5\)](#) 配置文件中添加缺省元素集合的某个系统调用。在对象设置缺省值时，将允许应用程序在不确定系统是否采用相关策略的情况下，也能向用户返回与对象相关的有意义的信息。



目前的 MAC 不支持直接修改内部形式的对象元素，所有的修改必须按照下列的步骤进行：将内部形式的对象转换成文本字符串，对字符串进行操作，最后将其转换成内部形式对象。如果应用程序的作者明确有需要，可以在将来的版本中加入对内部形式对象进行直接修改的接口。

6.8.2. 应用指定

用户上下文管理的接口，[setusercontext\(3\)](#)，的行已被修改，从 [login.conf\(5\)](#) 中与某个用户登录相关的 MAC 安全。当 `LOGIN_SETALL` 被设置，或者当 `LOGIN_SETMAC` 被明确指定时，一些安全对象将和其他用

上下文参数一起被设置。



可以预期，在今后的某个版本中，FreeBSD 将把 MAC 从 `login.conf` 的用户数据中抽出，使其成为一个独立的数据。不过在此前后，[setusercontext\(3\)](#) API 保持不。

6.9. 小结

TrustedBSD MAC 框架使得内核模块能以一种集中的方式，完善系统的安全策略。它既可利用已有的内核对象属性，又能使用由 MAC 框架助的安全数据，来实施控制。框架提供的活性使得人可以在其上实施各策略，如利用 BSD 已有的信任状 (credential) 与文件保护机制的策略，以及信息流安全策略（如 MLS 和 Biba）。新安全服务的策略编程人，可以参考本文，以了解有关安全模块的信息。

Chapter 7. 虚内存系

7.1. 物理内存的管理-vm_page_t

物理内存通体vm_page_t以基行管理。物理内存的由它各自体的vm_page_t所代表，些体存放在若干个管理列中的一个里面。

一可以于在(wired)、活(active)，去活(inactive)、存(cache)、自由(free)状。除了在状，一般被放置在一个双向表列里，代表了它所的状。在不放置在任何列里。

FreeBSD存和自由了一个更的列机制，以的分管理。一状都着多个列，列的安排着理器的一、二存。当需要分配一个新，FreeBSD会把一个按一、二存的面分配虚内存象。

此外，一个可以有引用数，可以被一个忙数定。虚内存系也有了"定"(ultimate locked)状，一个可以用志PG_BUSY表示一状。

之，个列都按照LRU(Least-Recently Used)的原工作。



者注

短Least-Recently Used有理解方式：1.将"least-recently"理解反向比，意"最早"，整个短理解"最近的使用最早"；2.将"least"和"recently"理解副，都修"used"，整个短理解"最近最少使用"。理解方式的意基本相同。

一个常常最初于在或活状。在，常常于某的表。虚内存系通描在一个活的列(LRU)定的年，以便将他移到一个不活的列中。移到存中的依然与一个VM象，但被作立即再用的候。在自由列中的是真正未被使用的。FreeBSD尽量不将放在自由列中，但是必保持一定数量的自由，以便中断分配。

如果一个程一个不在表中而在某一列中的(例如去活列或存列)，一个相耗源少的生，致被重激活。如果根本不存在于系内存之中，程必被阻塞，此被从磁中入。



者注

Intel等厂商的CPU工作在保模式，可用来虚内存。当址的地址空着真内存，正常写；当址的地址空没有真的真内存，CPU会生一个"err"，通知操作系与磁等行交，址入存内容，写址写出存内容。个"err"并非操作系或用程序人犯下的，尽管在CPU硬件中与用程序或操作系内核崩的的生机制相同。参Intel的CPU保模式手册。

FreeBSD的整列，将各个列中的数在一个当的比例上，同管理程序崩的已清理和未清理。重新平衡的比例数决定于系内存的担。重新平衡由pageout守程，包括清理未清理(与他后存同)、被引用的活程度(重置它在LRU列中的位置或在不同活程度的列移)、当比例不平衡在列移，如此等等。FreeBSD的VM系会将重激活而生的率低到一个合理的数，由此定某一活/置的程度。可以更好的决定何清理/分配一个做出决策。

7.2. 一的内存信息结构体-vm_object_t

FreeBSD了一的"虚内存对象"(VM对象)的思想。VM对象可以与各型的内存使用方式相合-直接使用(unbacked)、交(swap)、物理、文件。由于文件系统使用相同的VM对象管理核内数据-文件的内存, 所以些内存的也是一的。

VM对象可以被影子制(shadowed)。它可以被堆放到其它VM对象堆的端。例如, 可以有一个交VM对象, 放置在文件VM对象堆的端, 以MAP_PRIVATE的mmap()操作。的入操作也可以用来各各的共享特性, 包括写入制(copy-on-write, 用于日志文件系统), 以派生出地址空。

当注意, 一个vm_page_t 结构体在任一个刻只能与一个VM对象相。VM对象影子本可以跨例的共享相同的。

7.3. 文件系入/出-buf结构体

vnode VM对象, 比如文件VM对象, 一般需要它自己的清理(clean)/未清理(dirty)信息, 而不依赖于文件系的清理/未清理。例如, 当VM系要同一个物理和其的内存器, VM系就需要在写入到内存器前将已清理。外, 文件系统要能将文件或文件元数据的各部分映射到内核虚内存(KVM)中以便操作。

用来行些管理的结构体就是所周知的文件系统内存, struct buf或bp。当文件系统需要一个VM对象的一部分操作, 它常会将个对象的各部分映射到struct buf, 并且将struct buf中映射到内核虚内存(KVM)中。同的, 磁入/出通常要先将VM对象的各部分映射到buf结构体中, 然后buf结构体行入/出操作。下的vm_page_t在入/出期通常被"忙"。文件系统内存也会"忙", 于文件系统程序非常有用, 文件系统内存操作比VM真(hard)操作更好。

FreeBSD保留一定数量的内核虚内存来存放struct buf的映射, 但是些buf结构体是被清理的。些内核虚内存用来存放映射, 并不限制内存数据的能力。格的, 物理数据存是然而, 由于文件系统内存被用来理入/出, 他固有的限制了同行入/出可能的数量。但是, 由于通常有数千文件系统内存可供使用, 所以并不会造成。

7.4. 映射表-vm_map_t, vm_entry_t

FreeBSD将物理表从VM系中分了出来。各程的所有表可以脱程 (on the fly)重建, 并且通常被是一次性的。特殊的表, 如内核虚内存(KVM), 常常是被永久性分配的; 些表不是一次性的。

FreeBSD通vm_map_t和vm_entry_t 将虚内存中vm_objects的各地址部分起来。表被直接的从vm_map_t/vm_entry_t/vm_object_t 中有次的合成出来。里需要重申一下, 我曾提到的"物理直接"与vm_object相"并不很正。vm_page_t 也被会被接到正在与之相的表中。当表被用, 一个vm_page_t结构体可以被接到几个pmaps。然而, 由于有了次的, 因此在象中所有同一的引用会引用同一 vm_page_t结构体, 就了跨区域(board)的的一。

7.5. KVM存映射

FreeBSD使用KVM存放各各的内核结构体。在KVM中最大的个结构体是文件系统内存。那是与struct buf结构体有映射。

不像Linux，FreeBSD不将所有的物理内存映射到KVM中。这意味着FreeBSD可以在32位平台上管理超过4GB的内存配置。事实上，如果mmu（译者注：可能是指“内存管理单元”，“Memory Management Unit”）有足够的能力，FreeBSD理论上可以在32位平台上管理最多8TB的内存配置。然而，大多数32平台只能映射4GB内存，它只能是一个争点。

有几项机制可以管理KVM。管理KVM的主要机制是区域分配器（zone allocator）。区域分配器管理着KVM的大块，再将其切分为恒定大小的小块，以便按照某一类型的块体分配。可以使用命令`vmstat -m`一窥当前KVM分区使用情况。

7.6. 调整FreeBSD的虚拟内存系统

开发者的共同努力使得FreeBSD可以自行调整内核。一般来说，除了内核配置项`maxusers`和`NMBCLUSTERS`，不需要做任何乱的事情。有些内核配置项（一般）被指定在`/usr/src/sys/i386/conf/CONFIG_FILE`之中。所有可用内核配置项的描述可在`/usr/src/sys/i386/conf/LINT`中找到。

在一个大系统的配置中，可能需要增加`maxusers`的值。数值通常在10到128。注意，过度增加`maxusers`的值可能导致系统从可用的KVM中溢出，从而引起无法预知的操作。最好将`maxusers`设一个合理的数值，并且添加其它项，如`NMBCLUSTERS`，来增加特定的资源。

如果系统要被重负荷的使用网络，需要增加`NMBCLUSTERS`的值。数值通常在1024到4096。

`NBUF`也是系统的参数。这个参数决定系统可用来映射文件系统输入/输出缓存的KVM的数量。注意：这个参数与一般的缓存没有任何关系。这个参数可在3.0-CURRENT和以后的内核中被调用的项，通常不应当被手动的项。我们推荐不要指定`NBUF`。系统自行设定它。太小的值会导致非常低效的文件系统操作；太大的值会使用列表中缺少页面，而大量的页于在状态。

缺省情况下，FreeBSD内核是不被优化的。可以在内核配置文件中用`makeoptions`指定排他（debugging）和优化标志。注意，一般不使用`-g`，除非能够付出由此产生的大内核（典型的是7MB或更多）。

```
makeoptions    DEBUG="-g"
makeoptions    COPTFLAGS="-O -pipe"
```

Sysctl提供了在运行调整内核的方式。通常不需要指定任何sysctl量，尤其是与VM相关的那些量。

运行VM和系统调整的影响相当直接一些。首先，应当尽可能在UFS/FFS文件系统上使用Soft Updates。在`/usr/src/sys/ufs/ffs/README.softupdates`里有关于如何配置的指示。

其次，应当配置足够的交换空间。应当在每个物理磁盘上配置一个交换分区，最多4个，甚至在它的“工作”磁盘上。应当有至少2倍于主内存的交换空间；假如没有足够的内存的项，交换分区更多。也应当按照期望中的最大内存配置决定交换分区的大小，以后就不再需要重新磁盘分区了。如果系统崩溃后的内存倾倒（crash dump），第一个交换分区必须至少与主内存一样大，`/var/crash`必须有足够的空间来承载倾倒。

NFS上的交换分区可以很好的被4.X或后来的系统使用，但是必须明白NFS服务器将要承受安装操作很重的冲击。

Chapter 8. SMPng 中文

8.1. 简介

本文目前 SMPng 架构的介绍与读者行了介绍。它首先介绍了基本的原理和相应工具，其后是对于 FreeBSD 内核的同构与并行模型，接下来介绍了具体系统中的策略，并描述了在各个子系统引入粒度的同构和并行化的问题，最后是问题的说明，用以解释最初做出某些决策的动机，并使读者了解使用特定的原理所可能产生的重大影响。

本文仍在撰写当中，并将不断更新以反映与 SMPng 项目有关的最新问题与情况。其中有许多小问题目前只是提及，但我将会逐渐填充其内容。对于本文的更新和构建，请见下文。

SMPng 的目的是使内核能够并行。基本上，内核是一个很大而复杂的程序。要使内核能够多进程地运行，我需要使用某些其它多进程程序在开发中所用到的工具，包括互斥体(mutex)、共享/排他(shared/exclusive lock)、信号量(semaphores) 和条件变量(condition variable)。如果希望了解它以及其它 SMP 问题，请参考本文的[列表一](#)。

8.2. 基本工具与上层的基础知识

8.2.1. 原子操作指令和内存

对于内存和原子操作指令已有很多介绍材料，因此我们并不打算对其行详尽的介绍。而言之，如果有某一变量上写，就不能在不值得相应的开销其行读取操作。也就是，内存的作用在于保证内存操作的相对顺序，但并不保证内存操作的绝对顺序。言之，内存并不保证 CPU 将本地快取内存或内存缓冲的内容刷写回内存，而是在内存保证其所保存的数据，于能看到内存放的那个 CPU 或可。持有内存的 CPU 可以在其快取内存或内存缓冲中将数据保持其所希望的、任意的，但如果其它 CPU 在同一数据单元上行原子操作，第一个 CPU 必须保证，其所更新的数据，以及内存所要求的任何其它操作，第二个 CPU 可。

例如，假设在一模型中，在主存（或某一全局快取内存）中的数据是可读的，当某一 CPU 上触发原子操作，其它 CPU 的内存缓冲和快取内存就必须同一快取内存上的全部写操作，以及内存之后的全部未完成操作行刷写。

一来，在使用由原子操作保护的内存单元就需要特别小心。例如，在 sleep mutex 中，我们就必须使用 `atomic_cmpset` 而不是 `atomic_set` 来打 MTX_CONTESTED 位。这样做的原因是，我需把 `mtx_lock` 的值到某个变量，并据此行决策。然而，我得到的可能是旧的，也可能在我行决策的过程中生变化。因此，当行 `atomic_set` 时，最可能会同一行置位，而不是我行决策的那一个。就必须通过 `atomic_cmpset` 来保证只有在我决策依据是最新的，才相应的变量行置位。

最后，原子操作只允许一次更新或一个内存单元。需要原子地更新多个单元，就必须使用锁来代替它了。例如，如果需要更新多个相互关联的计数器，就必须使用锁，而不是多次独立的原子操作了。

8.2.2. 读与写

读并不需要像写那样。读操作的，都需要保证读的不是脏数据。然而，只有写操作必须是排他的，而多个进程可以安全地同一变量的。使用不同类型的读用于读和写操作有许多各自不同的方式。

第一种方法是用 `sx`，它可以用于写使用的排他，而操作共享。该方法十分简单明了。

第二种方法略晦。可以用多个来保护同一数据元。然而，只需其中的一个即可。然而，如果要写数据的，需要首先上所有的写。会大大提高写操作的代价，但当可能以多种方式数据却可能非常有用。例如，父进程指是同受 `proctree_lock` 和进程 `mutex` 保护的。在只希望已进程的父进程，用 `proc` 更方便。但是，其它一些地方，例如 `inferior` 需要通父指在进程上行搜索，并个进程上的地方就不能做了，否，将无法保护在我所得的果行操作，之前的状况依旧有效。

8.2.3. 上状态和果

如果需要使用来保持所量的状态，并据此行某些操作，是不能在量之前其上，并在行操作之前解的。早解将使量再次可，可能会致之前所做的决策失效。因此，在所指引的作束之前，必须保持上状态。

8.3. 架与概

8.3.1. 中断的理

与多其它多程 UNIX® 内核所采取的模式似，FreeBSD 会予中断理程序独立的进程上下文，能做能中断程在遇到阻塞。但了避免不必要的延，中断程在内核中，是以程的先行的。因此，中断理程序不运行久，以免死其它内核程。此外，由于多个理程序可以分享同一中断程，中断理程序不休眠，或使用可能致休眠的，以避免将其它中断理程序死。

目前在 FreeBSD 中的中断程是指重量中断程。称呼它的原因在于，到中断程需要行一次完整的上下文切操作。在最初的中，内核不允占，因此中断在打断内核程之前，必等待内核程阻塞或返回用之后才能行。

了解决，FreeBSD 内核在采用了占式度策略。目前，只有放休眠 `mutex` 或生中断才能断内核程，但最目是在 FreeBSD 上下面所描述的全占式度策略。

并非所有的中断理程序都在独立的进程上下文中行。相反，某些理程序会直接在主中断上下文中行。些中断理程序，在被地命名“快速”中断理程序，因早期版本的内核中使用了 `INTR_FAST` 志来些理程序。目前只有中断和串口 I/O 中断采用一型。由于些理程序没有独立的上下文，因而它都不能得阻塞性，因此也就只能使用自旋 `mutex`。

最后，有一称量上下文切的化，可以在 MD 代中使用。因中断程都是在内核上下文中行的，所以它可以借用任意程的 `vmSPACE`（虚内存地址空）。因此，在量上下文切中，切到中断程并不切的 `vmSPACE`，而是借用被中断程的 `vmSPACE`。保被中断程的 `vmSPACE` 不在中断理程中消失，被中断程在中断程不再借用其 `vmSPACE` 之前是不允行的。才提到的情况可能在中断程阻塞或完成生。如果中断程生阻塞，它再次入可行状将使用自己的上下文，一来，就可以放被中断的程了。

化的坏在于它和硬件密相，而且比，因此只有在能做来大幅性能改善才采用。目前可能早，而且事上可能会反而致性能下降，因几乎所有的中断理程序都会立即被全局（Giant）阻塞，而阻塞将需要程修正。外，Mike Smith 提采用一方式来理中断程：

1. 个中断理程序分部分，一个在主中断上下文中行的主体（predicate）和一个在自己的进程上下文中行的理程序（handler）。
2. 如果中断理程序有主体，当触中断，行主体。如果主体返回真，中断被理完，

内核从中断返回。如果主体返回假，或者中断没有主体，则调度程序继续运行。

在单处理器系统中采用轻量级上下文切换可能是非常理想的。因此，我们可能会希望在未来改变单处理器模式，因此，在最好的方案中，我们可能会希望在未来改变单处理器模式，以便我们完善中断处理架构，随后再考察轻量级上下文切换是否可用。

8.3.2. 内核抢占与临界区

8.3.2.1. 内核抢占简介

内核抢占的概念很简单，其基本思想是 CPU 应该首先做最高的工作。当然，至少在理想情况下是这样。有些时候，完成一项理想的代价会十分高昂，以至于在某些情况下抢占会得不偿失。

完全的内核抢占十分简单：在调度将要运行的进程并放入运行队列时，检查它的优先级是否高于目前正在运行的进程。如果是的话，进行一次上下文切换并立即开始运行新进程。

尽管内核在抢占时保护多数数据，但内核并不是可以安全地抢占的。例如，如果持有自旋 mutex 的进程被抢占，而新进程也需要同一自旋 mutex，新进程就可能一直自旋下去，因被中断的进程可能永远没有机会运行了。此外，某些代码，例如在 Alpha 上的 `exec` 进程地址空间编号运行的代码也不能被中断，因为它被用来支持进程的上下文切换操作。在某些代码段中，会调用使用临界区来禁用抢占。

8.3.2.2. 临界区

临界区 API 的任务是避免在临界区内产生上下文切换。对于完全抢占式内核而言，除了当前进程之外的其它进程的每个 `setrunqueue` 都是断点。`critical_enter` 的一种方式是将一个进程私有标志，并由其清除。如果使用 `setrunqueue` 设置了标志，则无论新进程和当前进程相比其优先级高低，都不会产生抢占。然而，由于临界区会在自旋 mutex 中用于避免上下文切换，而且能同时拥有多个自旋 mutex，因此临界区 API 必须支持嵌套。由于这个原因，目前的实现中采用了嵌套计数，而不是标志。

为了尽可能短时间和，在临界区中的抢占被推迟，而不是直接进行。如果进程被中断，并被置为可运行，而当前进程在临界区，则会置一个进程私有标志，表示有一个尚未运行的中断操作。当最外层临界区退出时，会清除标志，如果它被置位，则当前进程会被中断，以允许更高优先级的进程开始运行。

中断会引出一个和自旋 mutex 有关的函数。如果低优先级中断处理程序需要，它就不能中断任何需要标志的代码，以避免可能产生的损坏数据的情况。目前，这一机制是透临界区 API 以 `cpu_critical_enter` 和 `cpu_critical_exit` 函数的形式实现的。目前这一 API 会在所有 FreeBSD 所支持的平台上禁用和重新启用中断。这个方法并不是最好的，但它更易理解，也更容易正确地实现。理论上，这一辅助 API 只需要配合在主中断上下文中的自旋 mutex 使用。然而，为了代码更简单，它被用在了全部自旋 mutex，甚至包括所有临界区上。将其从 MI API 中剥离出来放入 MD API，并只在需要使用它的 MI API 的自旋 mutex 代码中使用可能会有更好的效果。如果我们最初采用了标志方式，则 MD API 可能需要改名，以彰显其单一用途 API 一事。

8.3.2.3. 负载均衡

如前面提到的，当完全抢占并非能提供最佳性能时，采取了一些折衷的措施。

第一折衷是，抢占并不考虑其它 CPU 的存在。假设有两个 CPU，A 和 B，其中 A 上进程的优先级为 4，而 B 上进程的优先级是 2。如果 CPU B 有一个优先级为 1 的进程可运行，则理论上，我希望 CPU A 切换到一新进程，这样就有两个优先级最高的进程在运行了。然而，固定两个 CPU 在抢占上更合理，并通过 IPI 向那个 CPU 发出信号，并完成相关的同步工作的代价十分高昂。因此，目前的代码会限制 CPU B 切换到更高优先级的进程。

注意做仍会系入更好的状态，因 CPU B 会去行先 1 而不是 2 的那个程。

第二折衷是限制于先的内核程的立即占。在前面所定的占操作的情形中，低先会被立即断（或在其退出界区后被断）。然而，多在内核中行的程，有很多只会行很短的就会阻塞或返回用。因此，如果内核断些程并行其它非的内核程，内核可能会在些程上要休眠或行完之前切出去。一来，CPU 就必整快存取以配合新程的行。当内核返回到被断的程，它又需要重新填充之前失的快存取信息。此外，如果内核能将阻塞或返回用的那个程的断延到之后的，能免去次外的上下文切。因此，默情况下，只有在先高的程是程，占代才会立即行断操作。

用所有内核程的完全占于非常有助，因它会暴露出更多的条件 (race conditions)。在以模些条件的理器系中，得尤其有用。因此，我提供了内核 `FULL_PREEMPTION` 来用所有内核程的占，一主要用于目的。

8.3.3. 程移

地，程从一个 CPU 移到另一个上的程称作移。在非占式内核中，只会在明定的点，例如用 `msleep` 或返回至用才会生。但是，在占式内核中，中断可能会在任何候制断，并致移。于 CPU 私有的数据而言可能会来一些面影，因除 `curthread` 和 `curpcb` 以外的数据都可能在移程中生化。由于存在潜在的程移，使得未受保的 CPU 私有数据得无用。就需要在某些代段禁止移，以得定的 CPU 私有数据。

目前我采用界区来避免移，因它能阻止上下文切。但是，有可能是一于的限制，因界区上会阻止当前理器上的中断程。因而，提供了一个 API，用以指示当前程在被断，不移到一 CPU。

API 也叫程制，它由度器提供。API 包括个函数：`sched_pin` 和 `sched_unpin`。个函数用于管理程私有的数 `td_pinned`。如果嵌套数大于零，程将被住，而程始行其嵌套数零，表示于未制状。所有的度器中，都要求保制程只在它首次用 `sched_pin` 所在的 CPU 上行。由于只有程自己会写嵌套数，而只有其它程在受制程没有行，且持有 `sched_lock` 才会嵌套数，因此 `td_pinned` 不必上。`sched_pin` 函数会使嵌套数，而 `sched_unpin` 使其。注意，些函数只操作当前程，并将其定到其行它所的 CPU 上。要将任意程定到指定的 CPU 上，使用 `sched_bind` 和 `sched_unbind`。

8.3.4. 出 (Callout)

内核机制 `timeout` 允内核服注册函数，以作 `softclock` 件中中断的一部分来行。事件将基于所希望的数的数目行，并在大指定的回用提供的函数。

未决 `timeout` (超) 事件的全局表是由一全局 `mutex`，`callout_lock` 保的；所有 `timeout` 表的，都必首先拿到个 `mutex`。当 `softclock` 醒，它会描未决超表，并出的那些。避免逆序，`softclock` 程会在用所提供的 `timeout` 回函数首先放 `callout_lock` `mutex`。如果在注册没有置 `CALLOUT_MPSAFE` 志，在用出函数之前，会取全局，并在之后放。其后，`callout_lock` `mutex` 会在理前再次得。`softclock` 代在放个 `mutex` 会非常小心地保持表的一致状。如果用了 `DIAGNOSTIC`，个函数的行会被，如果超了某一，会生警告。

8.4. 特定数据的策略

8.4.1. 凭据

`struct ucred` 是内核内部的凭据结构体，它通常作为内核中以进程为单位的控制依据。BSD-派生的系统采用一种“写控制”的模型来处理凭据数据：同一凭据结构体可能存在多个引用，如果需要对其进行修改，多个结构体将被复制、修改，然后替换引用。由于在打补丁用于控制凭据快速存取广泛存在，这种做法会极大地节省内存。在移到原子度的 SMP 时，这一模型也省去了大量的操作，因为只有未共享的凭据才能被修改，因而避免了在使用共享凭据结构体外的同步操作。

凭据结构体只有一个引用，被引用是可变的；不允许修改共享的凭据结构体，否则将可能导致死锁条件。`cr_mtxp` mutex 用于保护 `struct ucred` 的引用计数，以保持其一致性。使用凭据结构体时，必须在使用的进程中保持有效的引用，否则它就可能在一个不合理的消费者使用进程中被释放。

`struct ucred` mutex 是一个叶 mutex，出于性能考虑，它通过 mutex 池实现。

由于多用于控制决策，凭据通常情况下是以只读方式使用的，此时一般使用 `td_ucred`，因为它不需要上锁。当更新进程凭据时，进程和更新进程中必须持有 `proc` 锁。进程和更新操作必须使用 `p_ucred`，以避免竞争和使用锁的条件。

如果所操作系统将在更新进程凭据之后进行控制操作，那么 `td_ucred` 也必须刷新当前进程的。这样做能避免修改后使用旧的凭据。内核会自己在进程进入内核时，将进程结构体的 `td_ucred` 指向刷新进程的 `p_ucred`，以保证内核控制能用新的凭据。

8.4.2. 文件描述符和文件描述符表

该内容将在后面添加。

8.4.3. Jail 结构体

`struct prison` 保存了用于那些通过 `jail(2)` API 建立的 jail 所用到的管理信息。它包括 jail 的主机名、IP 地址，以及一些相关的设置。每个结构体包含引用计数，因为它指向一个结构体例的指针会在多个凭据结构体之间共享。用了一个 mutex，`pr_mtx` 来保护引用计数以及所有 jail 结构体中可写量的读写。有一些量只会在建立 jail 的瞬间生成，只需持有有效的 `struct prison` 就可以开始某些操作了。对于每个具体的上锁操作的文章，可以在 `sys/jail.h` 的注释中找到。

8.4.4. MAC 框架

TrustedBSD MAC 框架会以 `struct label` 的形式维护一系列内核对象的数据。一般来说，内核中的 label 结构是由与其同的内核对象同类型的保护的。例如，`struct vnode` 上的 `v_label` 是由其所在 `vnode` 上的 `vnode` 保护的。

除了嵌入到标准内核对象中的结构之外，MAC 框架也需要维护一个包含已注册的和激活策略的列表。策略表和忙数由一个全局 mutex (`mac_policy_list_lock`) 保护。由于能同时并行地进行多控制操作，策略表的只读操作，在忙数计数时，框架的入口需要首先持有该 mutex。MAC 入口操作的进程中并不需要持有此 mutex——有些操作，例如文件系统对象上的操作——是持久的。要修改策略表，例如在注册和解除注册策略时，需要持有此 mutex，而且要求引用计数为零，以避免在用表对其进行修改。

对于需要等待表进入置位状态的进程，提供了一个条件变量 `mac_policy_list_not_busy`，但这一条件变量只能在用户没有持有其它锁才能使用，否则可能会引起死锁。忙数在整个框架中事实上扮演了某种形式的共享/排他锁的作用：与 `sx` 不同的地方在于，等待列表进入置位状态的进程可以死，而不是允许多忙数和它在 MAC 框架入口（或内部）的锁之锁的逆序情况。

8.4.5. 模

于模子系， 用于保共享数据使用了一个独的， 它是一个 共享/排他 (SX) ， 多情况需要得它 (以共享或排他的方式)， 因此我提供了几个方便使用的宏来化个的， 些宏可以在 `sys/module.h` 中到， 其用法都非常明了。 个保的主要是 `module_t` (当以共享方式上) 和全局的 `modulelist_t` 个体， 以及模。 要更一理解些策略， 需要仔 kern/kern_module.c 的源代。

8.4.6. Newbus

newbus 系使用了一个 sx 。 的一方持有共享 (读) (sx_slock(9)) 而写的一方持有排他 (写) (sx_xlock(9))。 内部函数一般不需要行上， 而外部可的需要上。 有些目不需上， 因些目在全程是只读的， (例如 `device_get_softc(9)`)， 因而并不会生条件。 newbus 数据修改相而言非常少， 因此个的已足使用， 而不致造成性能折。

8.4.7. 管道

...

8.4.8. 程和程

- 程次
- proc 及其参考
- 在系用程中程私有的 proc 副本， 包括 `td_ucred`
- 程操作
- 程和会

8.4.9. 度器

本文在其它地方已提供了很多于 `sched_lock` 的参考和注。

8.4.10. Select 和 Poll

`select` 和 `poll` 个函数允程阻塞并等待文件描述符上的事件 — 最常的情况是文件描述符是否可或可写。

..

8.4.11. SIGIO

SIGIO 服允程求在特定文件描述符的/写状生化， 将 SIGIO 信号群其程。 任意定内核象上， 只允一程或程注册 SIGIO， 个程或程称属主 (owner)。 一支持 SIGIO 注册的象， 都包含一指字段， 如果象未注册 NULL， 否是一指向描述一注册的 `struct sigio` 的指。 一字段由一全局 mutex， `sigio_lock` 保。 用 SIGIO 函数， 必以 "引用" 方式一字段， 以保本地注册副本的中个字段不脱的保。

个到程或程的注册象， 都会分配一 `struct sigio` ， 并包括指回象的指、 属主、 信号信息、 凭据， 以及于一注册的一般信息。 个程或程都包含一个已注册 `struct sigio` 体的列表， 程来

是 `p_sigio`，而 `pg_sigio` 是 `pg_sigio`。这些表由相应的进程或进程组保护。除了用以将 `struct sigio` 接到进程上的 `sio_pgsigio` 字段之外，在 `struct sigio` 中的多数字段在注册进程中都是不变量。一般而言，人们在新的支持 SIGIO 的内核实现，会希望避免在调用 SIGIO 支持函数，例如 `fsetown` 或 `funsetown` 持有锁，以免去需要在锁体和全局 SIGIO 之间定顺序。通常可以通过提高锁体上的引用计数来达到目的，例如，在行管道操作，使用引用某个管道的文件描述符的操作，就可以照此处理。

8.4.12. Sysctl

`sysctl` MIB 服务会从内核内部，以及用锁的调用程序以系锁用的方式访问。它会引导至少一个和有锁的锁：其一是保持命名空间的数据锁的保护，其二是与那些通过 `sysctl` 接口访问的内核变量和函数之间的交互。由于 `sysctl` 允许直接访问（甚至修改）内核数据以及配置参数，`sysctl` 机制必须知道这些变量相关的上锁。目前，`sysctl` 使用一个全局 `sx` 锁来保证 `sysctl` 操作的串行化；然而，这些是假定用全局锁保护的，并且没有提供其它保护机制。锁一的其余部分将介绍上锁和 `sysctl` 相关的锁。

- 需要将 `sysctl` 更新锁所行的操作的顺序，从原先的旧、`copyin` 和 `copyout`、写新，改 `copyin`、上、旧、写新、解、`copyout`。一般的 `sysctl` 只是 `copyout` 旧并置它 `copyin` 所得到的新，仍然可以采用旧式的模型。然而，所有 `sysctl` 锁程序采用第二模型并避免操作方面，第二方式可能更矩一些。
- 于通常的情况，`sysctl` 可以内嵌一个 `mutex` 指向 `SYSCTL_FOO` 宏和锁体中。多数 `sysctl` 都是有锁的。于使用 `sx` 锁、自旋 `mutex`，或其它除一休眠 `mutex` 之外的策略，可以用 `SYSCTL_PROC` 点来完成正锁的上锁。

8.4.13. 任务队列 (Taskqueue)

任务队列 (taskqueue) 的接口包括一个与之锁的用于保护相关数据的锁。`taskqueue_queues_mutex` 是用于保护 `taskqueue_queues` TAILQ 的锁。与一个系锁的锁一个 `mutex` 锁是位于 `struct taskqueue` 锁体上。在此锁使用同锁原锁的目的在于保护 `struct taskqueue` 中数据的完整性。锁注意的是，并没有锁独的、锁助用锁其自身的工作锁行的锁化用的宏，因锁些锁基本上不会在 `kern/subr_taskqueue.c` 以外的地方用到。

8.5. 锁锁明

8.5.1. 休眠锁列

休眠锁列是一锁用于保存同锁一个等待通道 (wait channel) 上休眠锁程列表的数据锁。在等待通道上，锁个锁于非睡眠锁的锁程都会携锁一个休眠锁列锁。当锁程在等待通道上锁生阻塞，它会将休眠锁列锁体送锁那个等待通道。与等待通道锁的休眠锁列锁保存在一个散列表中。

休眠锁列散列表中保存了包含至少一个阻塞锁程的等待通道上的休眠锁列。锁个散列表上的锁称作 `sleepqueue` (休眠锁列) 锁。它包含了一个休眠锁列的锁表，以及一个自旋 `mutex`。此锁的自旋 `mutex` 用于保护休眠锁列表，以及其上休眠锁列锁的内容。一个等待通道上只会锁一个休眠锁列。如果有多个锁程在同一等待通道上阻塞，锁休眠锁列中将锁除第一个锁程之外的全部锁程。当从休眠锁列中锁除锁程，如果它不是唯一的阻塞的休眠锁程，锁会锁得主休眠锁列的锁表上的休眠锁列锁。最后一个锁程会在锁锁行锁得主休眠锁列。由于锁程有可能以和加入休眠锁列不同的次序从其中锁除，因此，锁程锁锁列锁可能会携锁与其锁入锁不同的休眠锁列。

`sleepq_lock` 函数会锁住指定等待通道上休眠锁列锁的自旋 `mutex`。`sleepq_lookup` 函数会在主休眠锁列散列表中锁锁定的等待通道。如果没有锁到主休眠锁列，它会返回 `NULL`。`sleepq_release` 函数会锁定等待通道锁的自旋 `mutex` 锁行解锁。

将进程加入休眠队列是通过 `sleepq_add` 来完成的。该函数的参数包括等待通道、指向保护等待通道的 mutex 的指针、等待消息描述串，以及一个标志掩码。调用此函数之前，调用 `sleepq_lock` 休眠队列上锁。如果等待通道不是通过 mutex 保护的（或者它由全局保护），则将 mutex 指针置为 NULL。而 flags（标志）参数包括了一个标志字段，用以表示进程即将加入到的休眠队列的标志，以及休眠是否是可中断的（SLEEPQ_INTERRUPTIBLE）。目前只有两种类型的休眠队列：通过 `msleep` 和 `wakeup` 函数管理的休眠队列（SLEEPQ_MSLEEP），以及基于条件变量的休眠队列（SLEEPQ_CONDVAR）。休眠队列类型和指针参数完全是用于内部的断言。调用 `sleepq_add` 的代码，明示地在 sleepqueue 上调用 `sleepq_lock` 行上锁之后，并使用等待函数在休眠队列上阻塞之前解除所有用于保护等待通道的 interlock。

调用 `sleepq_set_timeout` 可以设置休眠超时的时间。该函数的参数包括等待通道，以及以相等的毫秒数表示的超时时间。如果休眠被某个到来的信号打断，调用 `sleepq_catch_signals` 函数，该函数唯一的参数就是等待通道。如果此进程已有未决信号，`sleepq_catch_signals` 将返回信号号；其它情况下，其返回值是 0。

一旦将进程加入到休眠队列中，就可以使用 `sleepq_wait` 函数族之一将其阻塞了。目前共提供了四个等待函数，使用哪个取决于调用者是否希望允许使用超时、收到信号，或用进程调度器打断休眠状态。其中，`sleepq_wait` 函数原地等待，直到当前进程通过某个唤醒（wakeup）函数式地恢复行；`sleepq_timedwait` 函数等待，直到当前进程被式地唤醒，或者到早前使用 `sleepq_set_timeout` 设置的超时；`sleepq_wait_sig` 函数会等待式地唤醒，或者其休眠被中断；而 `sleepq_timedwait_sig` 函数等待式地唤醒、到调用 `sleepq_set_timeout` 设置的超时，或进程的休眠被中断三个条件之一。所有这些等待函数的第一个参数都是等待通道。除此之外，`sleepq_timedwait_sig` 的第二个参数是一个布尔值，表示之前调用 `sleepq_catch_signals` 是否有未决信号。

如果进程被式地恢复行，或其休眠被信号中止，等待函数会返回零，表示休眠成功。如果进程的休眠被超时或用进程调度器打断，会返回相应的 errno 数。需要注意的是，因 `sleepq_wait` 只能返回 0，因此调用者不能指望它返回任何有用信息，而假定它完成了一次成功的休眠。同时，如果进程的休眠超时，并同被中止，`sleepq_timedwait_sig` 将返回一个表示产生超时的代码。如果返回代码是 0 而且使用 `sleepq_wait_sig` 或 `sleepq_timedwait_sig` 来行阻塞，调用 `sleepq_calc_signal_retval` 来判断是否有未决信号，并据此组合的返回值。早前调用 `sleepq_catch_signals` 得到的信号号，作为参数 `sleepq_calc_signal_retval`。

在同一休眠通道上休眠的进程，可以由 `sleepq_broadcast` 或 `sleepq_signal` 函数来式地唤醒。这两个函数的参数均包括希望唤醒的等待通道、将唤醒进程的优先级（priority）提高到多少，以及一个标志（flags）参数表示将要恢复行的休眠队列类型。优先级参数将作为最低优先级，如果将恢复的进程的优先级比此参数更高（数更低）其优先级不会调整。标志参数主要用于函数内部的断言，用以休眠队列没有被当做标志的队列对待。例如，条件变量函数不恢复任何休眠队列的行。`sleepq_broadcast` 函数将恢复所有指定休眠通道上的阻塞进程，而 `sleepq_signal` 只恢复在等待通道上优先级最高的阻塞进程。在调用这些函数之前，首先使用 `sleepq_lock` 休眠队列上锁。

休眠进程也可以调用 `sleepq_abort` 函数来中断其休眠状态。该函数只有在持有 `sched_lock` 才能调用，而且进程必须位于休眠队列之上。进程也可以调用 `sleepq_remove` 函数从指定的休眠队列中删除。该函数包括两个参数，即休眠通道和进程，它只在进程位于指定休眠通道的休眠队列之上时才将其唤醒。如果进程不在那个休眠队列之上，或同位于同一等待通道的休眠队列上，该函数将什么都不做而直接返回。

8.5.2. 十字路口 (turnstile)

- 与休眠队列的对比和不同。
- 看/等待/放 (lookup/wait/release) - 介绍 TDF_TSNOBLOCK 条件。

- 先广播。

8.5.3. 关于 mutex 的一些问题

- 我们是否要求 `mtx_destroy()` 持有 mutex，因为它无法安全地断言它没有被其它对象持有？

8.5.3.1. 自旋 mutex

- 使用一个临界区...

8.5.3.2. 休眠 mutex

- 描述 mutex 冲突的条件
- 如何在持有锁时，可以安全地冲突 mutex 的 `mtx_lock`。

8.5.4. Witness

- 它能做什么
- 它如何工作

8.6. 其它问题

8.6.1. 中断源和 ICU 抽象

- `struct isrc`
- pic 问题

8.6.2. 其它问题/问题

- 是否将 interlock 问题 `sema_wait`？
- 是否提供非休眠式 `sx` 问题？
- 添加一些关于正确使用引用计数的介绍。

问题表

原子

当遵循适当的规则时，如果一个操作的效果被其它所有 CPU 均可见，则称其为原子操作。狭义的原子操作是机器直接提供的。就更高的抽象层次而言，如果一个体的多个成员由一个保护，如果它的操作都是在上锁后、解锁前执行的，也可以称其为原子操作。

阻塞

线程等待、资源或条件被阻塞。阻塞也因此被赋予了太多的意涵。

临界区

不允许生成占用的代码段。使用 `critical_enter(9)` API 来表示进入和退出临界区。

MD

表示与机器/平台有。

内存操作

内存操作包括或写内存中的指定位置。

MI

表示与机器/平台无。

操作

主中断上下文

主中断上下文表示当生中断所行的那段代。些代可以直接行某个中断理程序，或度一端口程，以便定的中断源行中断理程序。

内核程

一高先的内核程。目前，只有中断程属于先的内核程。

休眠

当程由条件量或通 msleep 或 tsleep 阻塞并入休眠列，称其入休眠状。

可休眠

可休眠是一在程休眠仍可持有的。管理器 (lockmgr) 和 sx 是目前 FreeBSD 中有的可休眠。最，某些 sx，例如 allproc (全部程) 和 proctree (程) 将成不可休眠。

程

由 struct thread 所表的内核程。程可以持有，并有独立的行上下文。

等待通道

程可以在其上休眠的内核虚地址。

Part II: □□□□程序

编写 FreeBSD 驱动程序

1. 简介

本章将介绍如何编写 FreeBSD 驱动程序。驱动程序在操作系统的上下文中多用于指代系统中硬件相关的东西，如磁盘，打印机，图形适配器及其驱动。驱动程序是操作系统中用于控制特定硬件的设备驱动。也有所谓的伪驱动，即驱动程序用设备模型的方式运行，而没有特定的底层硬件。驱动程序可以被静默地加载到系统，或者通过内核接口工具 `kld` 在需要时加载。

UNIX 操作系统中的大多数设备都是通过设备点来访问的，有些也被称为特殊文件。这些文件在文件系统中的层次结构中通常位于 `/dev` 目录下。在 FreeBSD 5.0-RELEASE 以前的发行版中，`devfs(5)` 的支持并没有被集成到 FreeBSD 中，每个设备点都必须静态构建，并且独立于相关驱动程序的存在。系统中大多数设备点是通过运行 `MAKEDEV` 构建的。

驱动程序可以粗略地分为字符、网络和网桥驱动程序。

2. 内核接口工具-KLD

kld 接口允许系统管理从运行的系统中动态地添加和移除功能。它允许驱动程序程序的作者将他修改后的新驱动添加到运行的内核中，而无需重新编译新驱动而频繁地重启。

kld 接口通过下面的特殊命令使用：

- `kldload` - 加载新内核模块
- `kldunload` - 卸载内核模块
- `kldstat` - 列出当前加载的模块

内核模块的程序框架

```

/*
 * KLD程序框架
 * 受Andrew Reiter在Daemonnews上的文章所
 */

#include sys/types.h
#include sys/module.h
#include sys/systm.h /* uprintf */
#include sys/errno.h
#include sys/param.h /* kernel.h中用到的定
 */
#include sys/kernel.h /* 模
初始化中使用的
型
 */

/*
 * 加
理函数，
理KLD的加
和卸
。
 */

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 = EOPNOTSUPP;
        break;
    }
    return(err);
}

/* 向内核其余部分声明此模
 */

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

DECLARE_MODULE(skeleton, skel_mod, SI_SUB_KLD, SI_ORDER_ANY);

```

.2.1. Makefile

FreeBSD提供了一个makefile包含文件，利用它
可以快速地将
附加到内核的
西。

```
SRCS=skeleton.c
KMOD=skeleton

.include bsd.kmod.mk
```

简单地用一个makefile运行make就能构建文件 skeleton.ko，输入如下命令可以把它加载到内核：

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

3. 设备驱动程序

UNIX® 提供了一套公共的系调用供应用程序使用。当用设备点，内核的上层将设备用分到相应的驱动程序。脚本 `/dev/MAKEDEV` 的系调生成了大多数的设备点，但如果正在创建自己的设备程序，可能需要用 `mknode` 创建自己的设备点。

3.1. 创建静态设备点

`mknode` 命令需要四个参数来创建设备点。必须指定设备点的名字，设备的类型，设备的主号和设备的从号。

3.2. 动态设备点

文件系统，或者 `devfs`，在全局文件系统名字空间中提供内核名字空间的设备。它消除了由于有驱动程序而没有静态设备点，或者有设备点而没有安装驱动程序而带来的潜在问题。Devfs 仍在发展中，但它能工作得相当好了。

4. 字符设备

字符驱动程序直接从应用程序数据，或数据到应用程序。是最普通的一类驱动程序，源代码中有大量的例子。

一个设备的例子会驻写它的任何数据，并且当读取它的时候会将数据返回。下面显示了两个版本，一个用于 FreeBSD 4.X，一个用于 FreeBSD 5.X。

```

/*
 * 'echo' 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 /* kernel.h中用到的定 */
#include sys/kernel.h /* 模初始化中使用的型 */
#include sys/conf.h /* cdevsw */
#include sys/uio.h /* uio */
#include sys/malloc.h

#define BUFFERSIZE 256

/* 函数原型 */
d_open_t echo_open;
d_close_t echo_close;
d_read_t echo_read;
d_write_t echo_write;

/* 字符入口点 */
static struct cdevsw echo_cdevsw = {
    echo_open,
    echo_close,
    echo_read,
    echo_write,
    noioctl,
    nopoll,
    nommap,
    nostrategy,
    "echo",
    33, /* lkms保留 - /usr/src/sys/conf/majors */
    nodump,
    nopsize,
    D_TTY,
    -1
};

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

```

```

/* 变量 */
static dev_t sdev;
static int count;
static t_echo *echomsg;

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

/*
 * 这个函数被kld[un]load(2)系统用来用,
 * 以决定加载和卸载模块需要采取的操作。
 */

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分配供程序使用的内存 */
        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 = EOPNOTSUPP;
        break;
    }
    return(err);
}

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

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

```

```

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

/*
 * read函数接受由echo_write()存[]的buf, 并将其返回到用[]空[],
 * 以供其他函数[]。
 * uio(9)
 */

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

    /*
     * []操作有多大?
     * 与用[]求的大小一[], 或者等于剩余数据的大小。
     */
    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) {
        uprntf("uiomove failed!\\n");
    }
    return(err);
}

/*
 * echo_write接受一个字符串并将它保存到[]缓冲区, 用于以后的[]。
 */

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

    /* 将字符串从用[]空的内存[]制到内核空[] */
    err = copyin(uio-uio_iov-io_v_base, echomsg-msg,
        MIN(uio-uio_iov-io_v_len, BUFFERSIZE - 1));

    /* []在需要以null[]束字符串, 并[]度 */
    *(echomsg-msg + MIN(uio-uio_iov-io_v_len, BUFFERSIZE - 1)) = 0;
    echomsg-len = MIN(uio-uio_iov-io_v_len, BUFFERSIZE);

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

```

```
}  
count++;  
return(err);  
}  
  
DEV_MODULE(echo,echo_loader,NULL);
```



```

/*
 *  'echo' KLD
 *
 * Murray Stokely
 *
 * 此代由Søren (Xride) Straarup到5.X
 */

#include sys/types.h
#include sys/module.h
#include sys/systm.h /* uprintf */
#include sys/errno.h
#include sys/param.h /* kernel.h中用到的定 */
#include sys/kernel.h /* 模初始化中使用的型 */
#include sys/conf.h /* cdevsw */
#include sys/uio.h /* uio */
#include sys/malloc.h

#define BUFFERSIZE 256

/* 函数原型 */
static d_open_t      echo_open;
static d_close_t     echo_close;
static d_read_t      echo_read;
static d_write_t     echo_write;

/* 字符入口点 */
static struct cdevsw echo_cdevsw = {
    .d_version = D_VERSION,
    .d_open = echo_open,
    .d_close = echo_close,
    .d_read = echo_read,
    .d_write = echo_write,
    .d_name = "echo",
};

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

/* 量 */
static struct cdev *echo_dev;
static int count;
static t_echo *echomsg;

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

```

```

/*
 * 一个函数被kld[un]load(2)系统用来用,
 * 以决定加载和卸载模块需要采取的操作.
 */

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分配供程序使用的内存 */
        echomsg = malloc(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 = EOPNOTSUPP;
        break;
    }
    return(err);
}

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

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

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

```

```

/*
 * read函数接受由echo_write()存⌋的buf, 并将其返回到用⌋空⌋,
 * 以供其他函数⌋⌋。
 * uio(9)
 */

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

    /*
     * ⌋个⌋操作有多大?
     * 等于用⌋⌋求的大小, 或者等于剩余数据的大小。
     */
    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接受一个字符串并将它保存到⌋缓冲区, 用于以后的⌋⌋。
 */

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

    /* 将字符串从用⌋空⌋的内存⌋制到内核空⌋ */
    err = copyin(uio-uio_iov-io_v_base, echomsg-msg,
                 MIN(uio-uio_iov-io_v_len, BUFFERSIZE - 1));

    /* ⌋在需要以null⌋束字符串, 并⌋⌋度 */
    *(echomsg-msg + MIN(uio-uio_iov-io_v_len, BUFFERSIZE - 1)) = 0;
    echomsg-len = MIN(uio-uio_iov-io_v_len, BUFFERSIZE);

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

DEV_MODULE(echo,echo_loader,NULL);

```

在FreeBSD 4.X上安装此程序，将首先需要用如下命令在的文件系上建立一个点：

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

程序被加后，能输入一些西，如：

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

真正的硬件在下一章描述。

充源

- [Dynamic Kernel Linker \(KLD\) Facility Programming Tutorial - Daemonnews](#) October 2000
- [How to Write Kernel Drivers with NEWBUS - Daemonnews](#) July 2000

.5. (消亡中)

其他UNIX®系支持一型的磁，称。是内核它提供冲的磁。冲使得几乎没有用，或者非常不可。冲会重新安排写操作的次序，使得用程序失了在任何刻及知道准的磁内容的能力。致磁数据（文件系，数据等）的可的和可的崩恢成不可能。由于写操作被延，内核无法向用程序告个特定的写操作遇到了写，又加了一致性。由于个原因，真正的用程序从不依于，事上，几乎所有磁的用程序都尽力指定是使用字符（或"raw"）。由于将个磁（分区）同具有不同个混一，从而致使相内核代大地化，作推磁I/O基化的一部分，FreeBSD 抛了冲的磁的支持。

.6. 网程序

网程序不需要使用点。个程序是基于内核内部的其他决定而不是用open()，网的使用通常由系用socket(2)引入。

更多，参 ifnet(9) 机手册、回的源代，以及 Bill Paul 撰写的网程序。

Chapter 9. ISA 驱动程序

9.1. 概述

本章介绍了编写ISA驱动程序的一些问题。这儿展示的代码相当简单，很容易让人想到真正的代码，不过依然还是代码。它避免了与所驱动的主机无关的问题。真的例子可以在驱动程序的源代码中找到。ep和aha更是信息的好来源。

9.2. 基本信息

典型的ISA程序需要以下包含文件：

```
#include sys/module.h
#include sys/bus.h
#include machine/bus.h
#include machine/resource.h
#include sys/rman.h

#include isa/isavar.h
#include isa/pnpvar.h
```

它描述了ISA和通用子系统的东西。

子系统是以面向对象的方式驱动的，其主要通过相应的方法函数来驱动。

ISA程序驱动的方法的列表与任何其他驱动的很相似。对于名字为"xxx"的假想程序，它将是：

- `static void xxx_isa_identify (driver_t *, device_t);` 通常用于驱动程序而不是设备程序。但对于ISA，这个方法有特殊用途：如果提供某些特定的（非PnP）方法自驱动，一个例程可以驱动它。
- `static int xxx_isa_probe (device_t dev);` 在已知（或PnP）位置探测。对于已部分配置的，一个例程也能提供特定的某些参数的自驱动。
- `static int xxx_isa_attach (device_t dev);` 挂接和初始化。
- `static int xxx_isa_detach (device_t dev);` 卸驱动模块前解挂。
- `static int xxx_isa_shutdown (device_t dev);` 系驱动前行驱动的。
- `static int xxx_isa_suspend (device_t dev);` 系驱动入能态前挂起。也可以中止一切到能态。
- `static int xxx_isa_resume (device_t dev);` 从能态返回后恢复驱动的活态。

`xxx_isa_probe()`和`xxx_isa_attach()`是必须提供的，其余例程根据驱动的需要可以有或没有。

使用下面一描述符将驱动接到系。

```

/* 支持的[]方法表 */
static device_method_t xxx_isa_methods[] = {
    /* 列出[]程序支持的所有[]方法函数 */
    /* 略去不支持的函数 */
    DEVMETHOD(device_identify, xxx_isa_identify),
    DEVMETHOD(device_probe, xxx_isa_probe),
    DEVMETHOD(device_attach, xxx_isa_attach),
    DEVMETHOD(device_detach, xxx_isa_detach),
    DEVMETHOD(device_shutdown, xxx_isa_shutdown),
    DEVMETHOD(device_suspend, xxx_isa_suspend),
    DEVMETHOD(device_resume, xxx_isa_resume),

    { 0, 0 }
};

static driver_t xxx_isa_driver = {
    "xxx",
    xxx_isa_methods,
    sizeof(struct xxx_softc),
};

static devclass_t xxx_devclass;

DRIVER_MODULE(xxx, isa, xxx_isa_driver, xxx_devclass,
    load_function, load_argument);

```

此[]的[]`xxx_softc`是一个[] 特定的[], 它包含私有的[]程序数据和[]程序[]源的描述符。 []代[]会自[]按需要[]个[]分配一个softc描述符。

如果[]程序作[]可加[]模[], 当[]程序被加[]或卸[], 会[]用`load_function()`函数[]行[]程序特定的初始化或清理工作, 并将`load_argument`作[]函数的一个参量[]去。 如果[]程序不支持[]加[] ([]句[], 它必[]被[]接到内核中), []些[]当被[]置0, 最后的定[]将看起来如下所示:

```

DRIVER_MODULE(xxx, isa, xxx_isa_driver,
    xxx_devclass, 0, 0);

```

如果[]程序是[]支持PnP的[]而写的, 那[]就必[]定[]一个包含 所有支持的PnP ID的表。[]个表由此[]程序所支持的PnP ID的列表 和以人可[]的形式[]出的、与[]些ID[]的硬件[]型和型号的描述[]成。看起来如下:

```

static struct isa_pnp_id xxx_pnp_ids[] = {
    /* []个[]所支持的PnP ID占一行 */
    { 0x12345678, "Our device model 1234A" },
    { 0x12345679, "Our device model 1234B" },
    { 0, NULL }, /* 表[]束 */
};

```

如果程序不支持PnP，它仍然需要一个空的PnP ID表，如下所示：

```
static struct isa_pnp_id xxx_pnp_ids[] = {
    { 0,      NULL }, /* 表结束 */
};
```

9.3. Device_t指针

`Device_t`是内核而定型的指针类型，这里我只关心从驱动程序作者的角度看感兴趣的方法。下面的方法用来操作内核中的：

- `device_t device_get_parent(dev)` 取dev的父设备。
- `driver_t device_get_driver(dev)` 取指向其驱动程序指针。
- `char *device_get_name(dev)` 取dev程序的名字，在我的例子中为"xxx"。
- `int device_get_unit(dev)` 取dev元号（与dev个程序dev的dev从0开始号）。
- `char *device_get_nameunit(dev)` 取dev名，包括dev元号。例如"xxx0"，"xxx1"等。
- `char *device_get_desc(dev)` 取dev描述。通常它以人可读的形式描述dev的切型号。
- `device_set_desc(dev, desc)` 置描述信息。dev使得dev描述指向desc字符串，此后dev个字符串就不能被解除分配。
- `device_set_desc_copy(dev, desc)` 置描述信息。描述被拷到内部dev分配的缓冲区，devdesc字符串在以后可以被改而不会产生有害的后果。
- `void *device_get_softc(dev)` 取指向与devdev的描述符（xxx_softcdev）的指针。
- `u_int32_t device_get_flags(dev)` 取配置文件中特定于dev的标志。

可以使用一个很方便的函数`device_printf(dev, fmt, ...)`从devdev程序中打印消息。它自己在消息前添加dev元名和冒号。

`device_t`的些方法在文件`kern/bus_subr.c`中。

9.4. 配置文件与dev配置期dev和dev的dev序

ISAdev在内核配置文件中的描述如下：

```
device xxx0 at isa? port 0x300 irq 10 drq 5
    iomem 0xd0000 flags 0x1 sensitive
```

端口、IRQ和其他dev被dev成与devdev的dev源。根据dev dev自dev配置需要和支持程度的不同，dev些dev是可dev的。例如，某些dev根本不需要DRQ，而有些dev允dev从dev配置端口dev取IRQdev置。如果机器有多个ISAdev，可以在配置文件中明确指定dev条dev，如isa0或isa1，否dev将在所有ISAdev上搜索dev。

敏感(sensitive)是一dev源dev求，它指示dev必dev在所有非敏感dev之前devdev。此特性dev被支持，但似乎从未在目前的任何devdev程序中使用dev。

对于老的ISA，很多情况下程序仍然能配置参数。但是系中配置的个必须具有一个配置行。如果系中装有同一型的个，但的程序却只有一个配置行，例如：

```
device xxx0 at isa?
```

那只有一个会被配置。

但于支持通PnP或有自行自的，一个配置行就足配置系中的所有，如上面的配置行，或者地：

```
device xxx at isa?
```

如果程序既支持能自的又支持老，并且同安装在一台机器上，那只要在配置文件中描述老就足了。自的将被自添加。

如果ISA是自配置的，生的事件如下：

所有程序的例程（包括所有PnP的PnP例程）以随机序被用。他出后就把添加到ISA上的列表中。通常程序的例程将新与它的程序起来。而PnP例程并不知道其他程序，因此不能将程序与它所添加的新起来。

使用PnP的PnP入睡，以防止它被探老。

被敏感(sensitive)的非PnP的探例程被用。如果探成功，那就其用挂接(attach)例程。

所有非PnP的探和接例程以同的方式被用。

PnP从睡眠中恢复来，并它分配所求的源：I/O、内存地址、IRQ和DRQ，所有些与已接的老不会冲突。

于个PnP，所有ISA程序的探例程都会被用。第一个要求此的程序将被接。多个程序以不同的先要求一个的情况是可能的，情况下，具有最高先的程序将。探例程必用ISA_PNP_PROBE()将真的PnP ID和程序支持的ID列表作比，如果ID不在表中返回失。意味着个程序，包括不支持任何PnP的程序，都必须未知的PnP无条件用ISA_PNP_PROBE()，于未知，至少要用一个空的PnP ID表并返回失。

探例程遇到会返回一个正()，成功返回零或。

的返回用于PnP支持多个接口的情况。例如，老的兼容接口和新的高接口通不同的程序来提供支持。个程序都。在探例程中返回高的程序先(句，返回0的程序具有最高的先，返回-1的其次，返回-2的更在其后，如此下去)。如果多个程序返回相同的，那最先用的。因此，如果程序返回0，就基本能信它得先仲裁。

特定的例程也能将一而不是个程序指派。就象使用PnP的情况一，于某一，会探一中所有的程序。由于个特性在任何存的程序中均未，故本文不再予以考。

由于探老的候PnP被禁用，它不会被接次（一次作老，一次作PnP）。但如果例程相的，情况下程序有任保同一不会被程序接次：一次作老的由用配置的，一次作自的。

由于自xxx的xx（包括PnP和xx特定的）的xx一个xx实践xx是，不能从内核配置文件中向它xxx旗xx。因此它xx必xx要xx根本不使用 旗xx，要xx所有自xxx的xx使用xx元号xx0的xx的旗xx，或者 使用sysctl接口而不是旗xx。

通xx使用函数族resource_query_*()和 resource_*_value()直接xx配置xx源，从而可以提供其他不常用的配置。它xx的xx位于 kern/subr_bus.c。老的IDE磁xxx器 i386/isa/wd.c包含xx使用的例子。但必xx先使用配置的xx准方法。将解析配置xx源xx事情留xx xx配置代xx。

9.5. xx源

用xx写入到内核配置文件中的信息被作xx配置xx源xx理，并xx到内核。 xx配置代xx解析xx部分信息并将其xxxxxxdevice_t的xx和与之 xx的xxxx源。xx于xx情况下的配置，xx程序可以直接使用 resource_*函数xx配置xx源。然而，通常既不需要也不推xxx做，因此xx儿不再xx一xxxx个xx。

xxxx源与xx个xx相xx。通xx型和xx型中的数字xx它xx。xx于ISAxx，定xx了下面的xx型：

- SYS_RES_IRQ - 中断号
- SYS_RES_DRQ - ISA DMA通道号
- SYS_RES_MEMORY - 映射到系xx内存空xx 的xx内存的xx
- SYS_RES_IOPORT - xxxI/O寄存器的xx

xx型内的枚xx从0xx始，因此如果xx有xx个内存区域，它的 SYS_RES_MEMORY xx型的xx源xx号xx0和1。 xx源xx型与Cxx言的xx型无xx，所有xx源xx具有Cxx言 unsigned long xx型，并且必要xx必xx行xx型xx制xx (cast)。xx源号不必xx，尽管xx于ISA它xx一般是xx的。ISAxx允xx的xx源xx号xx：

```
IRQ: 0-1
DRQ: 0-1
MEMORY: 0-3
IOPORT: 0-7
```

所有xx源被表示xx有起始xx和xx数的xx。xx于IRQ和DRQxx源，xx数一般等于1。内存的xx引用物理地址。

xx源能xx行三xx型的xx作：

- set/get
- allocate/release
- activate/deactivate

Setxx置xx源使用的xx。Allocation保留出xx求的xx，使得 其它xx不能再占用（并xx此xx没有被其它xx占用）。Activationxx行必要的xx作使得xx程序可以xxxx源（例如，xx于 内存，它将被映射到内核的虚xx地址空xx）。

操作xx源的函数有：

- int bus_set_resource(device_t dev, int type, int rid, u_long start, u_long count)

xx源xx置xx。成功xx返回0，否xx返回xxxx。 一般此函数只有在type, rid, start或 count之一的xx超出了允xx的xx才会 返回xx。

- dev - 设备的
- type - 资源类型, SYS_RES_*
- rid - 资源内部的资源号 (ID)
- start, count - 资源范围

- `int bus_get_resource(device_t dev, int type, int rid, u_long *startp, u_long *countp)`

取得资源范围。成功时返回0, 如果资源尚未定义则返回-1。

- `u_long bus_get_resource_start(device_t dev, int type, int rid) u_long bus_get_resource_count(device_t dev, int type, int rid)`

便捷函数, 只用来取得start或count。出错的情况下返回0, 因此如果0是资源的start合法值之一, 将无法区分返回的0是否指示错误。幸运的是, 由于附加驱动程序, 没有ISA资源的start从0开始。

- `void bus_delete_resource(device_t dev, int type, int rid)`

删除资源, 令其未定义。

- `struct resource * bus_alloc_resource(device_t dev, int type, int *rid, u_long start, u_long end, u_long count, u_int flags)`

在start和end之间没有被其它资源占用的地方按count的范围分配一个资源。不重叠, 不支持共享。如果资源尚未被定义, 则自行创建它。start=0, end=~0 (全1) 的特殊值意味着必须使用以前通过 `bus_set_resource()` 设置的固定值: start和count就是它自己, end=(start+count), 这种情况下, 如果以前资源没有定义, 则返回-1。尽管rid通常引用资源, 但它并不被ISA资源的资源分配代码设置 (其它资源可能使用不同的方法并可能修改它)。

标志是一个位映射, 用户感兴趣的有:

- `RF_ACTIVE` - 使得资源分配后被自己激活。
- `RF_SHAREABLE` - 资源可以同被多个程序共享。
- `RF_TIMESHARE` - 资源可以被多个程序共享, 也就是, 被多个程序同时分配, 但任何资源只能被其中一个激活。
- 出错返回0。被分配的资源可以使用 `rhand_*` 从返回的句柄取得。
- `int bus_release_resource(device_t dev, int type, int rid, struct resource *r)`
- 释放资源, 从 `bus_alloc_resource()` 返回的句柄。成功时返回0, 否则返回-1。
- `int bus_activate_resource(device_t dev, int type, int rid, struct resource *r) int bus_deactivate_resource(device_t dev, int type, int rid, struct resource *r)`
- 激活或禁用资源。成功时返回0, 否则返回-1。如果资源被分配共享且当前被一程序激活, 则返回 `EBUSY`。
- `int bus_setup_intr(device_t dev, struct resource *r, int flags, driver_intr_t *handler, void *arg, void **cookiep) int bus_teardown_intr(device_t dev, struct resource *r, void *cookie)`
- 设置/分中断驱动程序与资源。成功时返回0, 否则返回-1。
- `r` - 被激活的描述IRQ的资源句柄。

flags - 中断标志, 如下之一:

- `INTR_TYPE_TTY` - 终端和其它类似的字符型。使用 `spltty()` 屏蔽它。
- `(INTR_TYPE_TTY | INTR_TYPE_FAST)` - 输入缓冲小的终端型，而且输入上的数据丢失很严重（例如老式串口）。使用 `spltty()` 屏蔽它。
- `INTR_TYPE_BIO` - 块型，不包括CAM控制器上的。使用 `splbio()` 屏蔽它。
- `INTR_TYPE_CAM` - CAM（通用访问方法 Common Access Method）块控制器。使用 `splcam()` 屏蔽它。
- `INTR_TYPE_NET` - 网络接口控制器。使用 `splimp()` 屏蔽它。
- `INTR_TYPE_MISC` - 各种其它。除了通过 `splhigh()` 没有其它方法屏蔽它。`splhigh()` 屏蔽所有中断。

当中断处理程序运行，匹配其先的所有其它中断都被屏蔽，唯一的例外是 `MISC`，它不会屏蔽其它中断，也不会被其它中断屏蔽。

- *handler* - 指向处理程序的指针，类型 `driver_intr_t` 被定义为 `void driver_intr_t(void *)`
- *arg* - 处理程序的参量，特定。由处理程序将它从 `void*` 任何类型。ISA 中断处理程序的旧定义是使用元号作参量，新定义（推荐）使用指向 `softc` 的指针。
- *cookie[p]* - 从 `setup()` 接收的，当 `teardown()` 用于处理程序。

定义了若干方法来操作资源句柄 (`struct resource *`)。程序作者感兴趣的有：

- `u_long rman_get_start(r)` `u_long rman_get_end(r)` 取得被分配的资源的起始和结束。
- `void *rman_get_virtual(r)` 取得被激活的内存资源的虚地址。

9.6. 内存映射

很多情况下程序和它们之间的数据交互是通过内存进行的。有几种可能的情况：

(a) 内存位于板上

(b) 内存是计算机的主内存

情况(a)中，程序可能需要在板上的内存与主存之间来回拷数据。为了将板上的内存映射到内核的虚地址空间，板上内存的物理地址和宽度必须被定义 `SYS_RES_MEMORY` 资源。然后资源就可以被分配并激活，它的虚地址通过 `rman_get_virtual()` 取。老的程序将函数 `pmap_mapdev()` 用于此目的，现在不再再直接使用此函数。它已成为资源激活的一个内部。

大多数ISA的内存配置物理地址位于640KB-1MB之间的某个位置。某些ISA需要更大的内存，位于16M以下的某个位置（由于ISA上24位地址限制）。这种情况下，如果机器有比内存的起始地址更多的内存（句柄，它重），必须在被使用的内存起始地址配置一个内存空洞。许多 BIOS 允许在起始于14MB或15MB配置1M的内存空洞。如果 BIOS 正确地报告内存空洞，FreeBSD就能正确处理它（此特性在老BIOS上可能会出错）。

情况(b)中，只是数据的地址被送到，使用DMA将数据从主存中的数据。存在一个限制：首先，ISA只能寻址16MB以下的内存。其次，虚地址空间中地址在物理地址空间中可能不连续，可能不得不行分散/收集操作。子系统提供一些提供或成的解决方法，剩下的必须由程序自己完成。

DMA内存分配使用了两个结构，`bus_dma_tag_t` 和 `bus_dmamap_t`。tag (tag) 描述了DMA内存要求的特性。映射 (map) 表示按照某些特性分配的内存。多个映射可以与同一资源。

按照特性的继承而被分成类型层次。子继承父的所有要求，可以令其更严格，但不允许放宽要求。

一般地，一个元素建立一个节点（没有父节点）。如果一个节点需要不同要求的内存区，多个内存区都会建立一个节点，有些节点作父节点的孩子。

使用节点建立映射的方法有：

其一，分配一大块符合要求的内存（以后可以被释放）。一般用于分配了与通信而存在相关的那些内存区。将节点的内存加到映射中非常容易：它被看作位于当前物理内存的一整块。

其二，将虚内存中的任意区域加到映射中。片内存的块一都被映射，看是否符合映射的要求。如何符合留在原始位置。如果不符合分配一个新的符合要求的"反弹面(bounce page)"，用作中间存。当从不符合的原始面写入数据，数据首先被拷到反弹面，然后从反弹面到。当读取，数据将会从到反弹面，然后被拷到它不符合的原始面。原始和反弹面之间的拷理被称作同。一般用于次的基础之上：层次加缓冲区，完成，卸缓冲区。

工作在DMA内存上的函数有：

- `int bus_dma_tag_create(bus_dma_tag_t parent, bus_size_t alignment, bus_size_t boundary, bus_addr_t lowaddr, bus_addr_t highaddr, bus_dma_filter_t *filter, void *filterarg, bus_size_t maxsize, int nsegments, bus_size_t maxsegsz, int flags, bus_dma_tag_t *dmat)`

建新。成功返回0，否则返回。

- *parent* - 父或者NULL， NULL用于建。
- *alignment* - 将要分配的内存区的要求。"no specific alignment"1。用于以后的 `bus_dmamem_alloc()` 而不是 `bus_dmamap_create()` 用。
- *boundary* - 物理地址界，分配内存不能穿。于"no boundary" 使用0。用于以后的 `bus_dmamem_alloc()` 而不是 `bus_dmamap_create()` 用。必2的乘方。如果以非DMA方式使用内存（也就是，DMA地址由ISA DMA控制器提供而不是自身），由于DMA硬件限制，界必不能大于64KB (64*1024)。
- *lowaddr, highaddr* - 名字微有些。些用于限制可用于内存分配的物理地址的允许。其切合根据以后不同的使用而有所不同。
 - 于 `bus_dmamem_alloc()`，从0到 `lowaddr-1` 的所有地址被允许，更高的地址不允许使用。
 - 于 `bus_dmamap_create()`，区 `[lowaddr; highaddr]` 之外的所有地址被可。之内的地址面被函数，由它决定是否可。如果没有提供函数，整个区被不可。
 - 于ISA，正常（没有函数）：

`lowaddr = BUS_SPACE_MAXADDR_24BIT`

`highaddr = BUS_SPACE_MAXADDR`

- *filter, filterarg* - 函数及其参数。如果 `filter` NULL，当用 `bus_dmamap_create()`，整个区 `[lowaddr, highaddr]` 被不可。否，区 `[lowaddr; highaddr]` 内的个被的面物理地址被函数，由它决定是否可。函数的原型：`int filterfunc(void *arg, bus_addr_t paddr)`。当面可以被它必返回0，否则返回非零。
- *maxsize* - 通此可以分配的最大内存（以字）。有是个很估算，或者可以任意大，情况下，于ISA个可以 `BUS_SPACE_MAXSIZE_24BIT`。

- *nsegments* - 支持的分散/收集段 的最大数目。如果不加限制，使用当使用 `BUS_SPACE_UNRESTRICTED`。建父使用个，而子指定限制。 `nsegments` 等于 `BUS_SPACE_UNRESTRICTED` 的不能用于加映射，可以将它作父。 `nsegments` 的限制大250-300，再高的将致内核堆溢出（硬件 无法正常支持那多的分散/收集缓冲区）。
- *maxsegsz* - 支持的分散/收集段 的最大尺寸。于ISA的最大 `BUS_SPACE_MAXSIZE_24BIT`。
- *flags* - 旗的位。感兴趣的旗 只有：

- `BUS_DMA_ALLOCNOW` - 建 求分配所有可能用到的反射面。
- `BUS_DMA_ISA` - 比神秘的一个志，用于Alpha机器。i386机器没有定它。Alpha机器的所有ISA 都当使用个志，但似乎没有的 程序。

- *dmat* - 指向返回的新 的存 的 指。

• `int bus_dma_tag_destroy(bus_dma_tag_t dmat)`

。成功返回0，否返回。

dmat - 被 的。

• `int bus_dmamem_alloc(bus_dma_tag_t dmat, void** vaddr, int flags, bus_dmamap_t *mapp)`

分配 所描述的一 内存区。被分配的内存的大小 的 `maxsize`。成功返回0，否返回。用 果被用于 取内存的 物理地址，但在此之前必用 `bus_dmamap_load()` 将其加。

- *dmat* - 。
- *vaddr* - 指向存的指，存空 用于返回的分配区域的内核虚地址。
- *flags* - 旗的位。唯一感兴趣的旗：
- `BUS_DMA_NOWAIT` - 如果内存不能 立即可用返回。如果此志没有置，允 例程睡眠，直到内存可用止。
- *mapp* - 指向返回的新映射的存的指。

• `void bus_dmamem_free(bus_dma_tag_t dmat, void *vaddr, bus_dmamap_t map)`

放由 `bus_dmamem_alloc()` 分配的内存。 目前，分配的有ISA限制的内存的放没有。因此，建 的使用模型 尽可能 地保持和重用分配的区域。不要 易地 放某些区域，然后再短 地分配它。 并不意味着不 当使用 `bus_dmamem_free()`：希望很快它就会被 完整地。

- *dmat* - 。
- *vaddr* - 内存的内核虚地址
- *map* - 内存的映射（跟 `bus_dmamem_alloc()` 返回的一）

• `int bus_dmamap_create(bus_dma_tag_t dmat, int flags, bus_dmamap_t *mapp)`

建映射，以后用于 `bus_dmamap_load()`。成功返回0，否 返回。

- *dmat* - 。
- *flags* - 理 上是旗的位。但 从未定 任何旗，因此目前 是0。
- *mapp* - 指向返回的新映射的存的指。

- `int bus_dmamap_destroy(bus_dma_tag_t dmat, bus_dmamap_t map)`

映射。成功返回0， 否返回非0。

- `dmat` - 与映射的
- `map` - 将要被的映射
- `int bus_dmamap_load(bus_dma_tag_t dmat, bus_dmamap_t map, void *buf, bus_size_t buflen, bus_dmamap_callback_t *callback, void *callback_arg, int flags)`

加缓冲区到映射中(映射必事先由 `bus_dmamap_create()`或者 `bus_dmamem_alloc()`建。缓冲区的所有面都会被， 看是否符合的要求， 并那些不符合的分配反面。会建物理段描述符的数， 并将其回函数。 回函数以某方式理个数。系中的反缓冲区是受限的， 因此如果需要的反缓冲区不能立即得， 将求入， 当反缓冲区可用再用回函数。如果回函数立即行返回0， 如果求被排， 等待将来行， 返回 `EINPROGRESS`。后一情况下， 与排的回函数之的同由程序。

- `dmat` -
- `map` - 映射
- `buf` - 缓冲区的内核虚地址
- `buflen` - 缓冲区的度
- `callback, callback_arg` - 回函数及其参数

回函数的原型：

```
void callback(void *arg, bus_dma_segment_t *seg, int nseg, int error)
```

- `arg` - 与 `bus_dmamap_load()`的`callback_arg` 相同。
- `seg` - 段描述符的数
- `nseg` - 数中的描述符个数
- `error` - 表示段数目溢出：如被 `EFBIG`， 允的最大数目的段无法容缓冲区。 情况下数中的描述符的数目只有可的那多。 情况的理由程序决定：根据希望的， 程序可以其， 或将缓冲区分个并独理第二个。

段数中的一包含如下字段：

- `ds_addr` - 段物理地址
 - `ds_len` - 段度
 - `void bus_dmamap_unload(bus_dma_tag_t dmat, bus_dmamap_t map)`
- unload the map.
- `dmat` -
 - `map` - 已加的映射
 - `void bus_dmamap_sync (bus_dma_tag_t dmat, bus_dmamap_t map, bus_dmasync_op_t op)`

与行物理前后， 将加的缓冲区与其反面行同。 此函数完成原始缓冲区与其映射版本之所有必需的数据拷工作。 行之前和之后必缓冲区行同。

- *dmabuf* - 映射
- *map* - 已加锁的映射
- *op* - 要行的同步操作的类型：
- *BUS_DMASYNC_PREREAD* - 从DMA到CPU缓冲区的操作之前
- *BUS_DMASYNC_POSTREAD* - 从DMA到CPU缓冲区的操作之后
- *BUS_DMASYNC_PREWRITE* - 从CPU缓冲区到DMA的写操作之前
- *BUS_DMASYNC_POSTWRITE* - 从CPU缓冲区到DMA的写操作之后

当前PREREAD和POSTWRITE空操作，但将来可能会改变，因此程序不能忽略它们。由`bus_dmamem_alloc()`获得的内存不需要同步。

从`bus_dmamap_load()`中调用回调函数之前，段数是存储在其中的。并且是按允许的最大数目的段先分配好的。由于i386体系上段数目的限制（250-300（内核4KB去用的大小，段数条目的大小8字节，和其它必须留出来的空间）。由于数基于最大数目而分配，因此一个必须不能置成超出需要。幸运的是，对于大多数硬件而言，所支持的段的最大数目低很多。但如果程序想处理具有非常多分散/收集段的缓冲区，应当一部分一部分地处理：加缓冲区的下一部分，如此反复。

一个实践是段数目可能限制缓冲区的大小。如果缓冲区中的所有面巧物理上不连续，在分片情况下支持的最大缓冲区尺寸（`nsegments * page_size`）。例如，如果支持的段的最大数目10，在i386上可以保持支持的最大缓冲区大小40K。如果希望更大的，需要在程序中使用一些特殊技巧。

如果硬件根本不支持分散/收集，或者程序希望即使在重分片的情况下仍然支持某缓冲区大小，解决方法是：如果无法容纳下原始缓冲区，就在程序中分配一个自己的缓冲区作中间存储。

下面是当使用映射的典型程序，根据映射的具体使用而不同。字符-用于表示流。

由于从接口接到分片，期望位置一直不变的缓冲区：

```
bus_dmamem_alloc - bus_dmamap_load - ...use buffer... - bus_dmamap_unload - bus_dmamem_free
```

由于从程序外部去，并且常规化的缓冲区：

```
bus_dmamap_create -
- bus_dmamap_load - bus_dmamap_sync(PRE...) - do transfer -
- bus_dmamap_sync(POST...) - bus_dmamap_unload -
...
- bus_dmamap_load - bus_dmamap_sync(PRE...) - do transfer -
- bus_dmamap_sync(POST...) - bus_dmamap_unload -
- bus_dmamap_destroy
```

当加由`bus_dmamem_alloc()`建的映射，去的缓冲区的地址和大小必须和`bus_dmamem_alloc()`中使用的一致。情况下就可以保证整个缓冲区被作一个段而映射（因而可以基于此假设），并且请求被立即行（永不返回EINPROGRESS）。情况下回调函数需要作的只是保存物理地址。

典型示例如下：


```

static void
alloc_callback(void *arg, bus_dma_segment_t *seg, int nseg, int error)
{
    *(bus_addr_t *)arg = seg[0].ds_addr;
}

...
int error;
struct somedata {
    ....
};
struct somedata *vsomedata; /* 虚地址 */
bus_addr_t psomedata; /* 物理地址的地址 */
bus_dma_tag_t tag_somedata;
bus_dmamap_t map_somedata;
...

error=bus_dma_tag_create(parent_tag, alignment,
    boundary, lowaddr, highaddr, /*filter*/ NULL, /*filterarg*/ NULL,
    /*maxsize*/ sizeof(struct somedata), /*nsegments*/ 1,
    /*maxsegsz*/ sizeof(struct somedata), /*flags*/ 0,
    );
if(error)
    return error;

error = bus_dmamem_alloc(tag_somedata, , /* flags*/ 0,
    );
if(error)
    return error;

bus_dmamap_load(tag_somedata, map_somedata, (void *)vsomedata,
    sizeof (struct somedata), alloc_callback,
    (void *) , /*flags*/0);

```

代码看起来有点怪，也比普通，但那是正的使用方法。如果是：如果分配多个内存区域，将它合成一个并作整体分配（如果和界限限制允许的）是一个很好的主意。

当加任意缓冲区到由bus_dmamap_create() 建的映射，由于回可能被延，因此必采取特殊措施与回函数同行。代码看起来像下面的子：


```

{
    int s;
    int error;

    s = splsoftvm();
    error = bus_dmamap_load(
        dmat,
        dmamap,
        buffer_ptr,
        buffer_len,
        callback,
        /*callback_arg*/ buffer_descriptor,
        /*flags*/0);
    if (error == EINPROGRESS) {
        /*
         * 行必要的操作以保与回的同。
         * 回被保直到我行了splx()或tsleep()才会被用。
         */
    }
    splx(s);
}

```

理求的方法分是：

1. 如果通式地求已束来完成求（例如CAM求）， 将所有一的理放入回程序中会比，回束后会求。之后不需要太多外的同。由于流控制的原因，求列直到求完成才放可能是个好主意。
2. 如果求是在函数返回完成（例如字符上写的写求）， 需要在缓冲区描述符上置同志，并用 `tsleep()`。后面当回函数被用，它将行理并同志。如果置了同志，它出一个醒操作。在方法中，回函数或者行所由必需的理（就像前面的情况），或者在缓冲区描述符中存段数。回完成后，回函数就能使用个存的段数并行所有的理。

9.7. DMA

ISA中Direct Memory Access (DMA)是通DMA控制器（上是它 中的个，但只是无）的。了使以前的ISA便宜， 控制和地址生的都集中在DMA控制器中。幸的是，FreeBSD提供了一套函数，些函数大多把DMA控制器的繁程序 藏了起来。

最情况是那些比智能的。就象PCI上的主一， 它自己能生周期和内存地址。它真正从DMA控制器需要的 唯一事情是仲裁。所以了此目的，它假装是从DMA控制器。 当接程序，系DMA控制器需要做的唯一事情就是通用 如下函数在一个DMA通道上激活模式。

```
void isa_dmacascade(int channel_number)
```

所有一的活通程完成。当卸程序，不需要 用DMA相的函数。

于的，事情反而得。使用的函数包括：

- `int isa_dma_acquire(int chanel_number)`

保留一个DMA通道。成功返回0，如果通道已被保留或被其它程序保留返回EBUSY。大多数的ISA都不能共享DMA通道，因此这个函数通常在接口上使用。源的代接口使得保留成多余，但目前仍必使用。如果不使用，后面其它DMA例程将会panic。

- `int isa_dma_release(int chanel_number)`

放先前保留的DMA通道。放通道必不能有正在行中的（外，放通道后必不能再起）。

- `void isa_dmainit(int chan, u_int bouncebufsize)`

分配由特定通道使用的反缓冲区。求的缓冲区大小不能超过64KB。以后，如果缓冲区巧不是物理的，或超出ISA可的内存，或跨越64KB的界，会自使用反缓冲区。如果是使用符合上述条件的缓冲区（例如，由 `bus_dmamem_alloc()` 分配的那些），不需要用 `isa_dmainit()`。但使用此函数会通DMA控制器任意数据得非常方便。

- *chan* - 通道号
- *bouncebufsize* - 以字数的反缓冲区的大小

- `void isa_dmastart(int flags, caddr_t addr, u_int nbytes, int chan)`

准DMA。上的之前必需用此函数来置DMA控制器。它缓冲区是否的且在ISA内存之内，如果不是自使用反缓冲区。如果需要反缓冲区，但反缓冲区没有用 `isa_dmainit()` 置，或于求的大小来太小，系将panic。写求且使用反缓冲区的情况下，数据将被自拷到反缓冲区。

- *flags* - 位掩，决定将要完成的操作的类型。方向位B_READ和B_WRITE互斥。

- B_READ - 从ISA到内存
- B_WRITE - 从内存写到ISA上
- B_RAW - 如果置DMA控制器将会住缓冲区，并在束后自重新初始化它自己，再次重同一缓冲区（当然，程序可能起的一个之前改缓冲区中的数据）。如果没有置，参数只一次有效，在起下一次之前必再次用 `isa_dmastart()`。只有在不使用反缓冲区使用B_RAW才有意。

- *addr* - 缓冲区的虚地址

- *nbytes* - 缓冲区度。必小于等于64KB。不允度0：因DMA控制器将会理解64KB，而内核代把它理解0，那就会致不可的效果。于通道号等于和高于4的情况，度必需偶数，因些通道次2字。奇数度情况下，最后一个字不被。

- *chan* - 通道号

- `void isa_dmadone(int flags, caddr_t addr, int nbytes, int chan)`

告完成后，同内存。如果是使用反缓冲区的操作，将数据从反缓冲区拷到原始缓冲区。参量与 `isa_dmastart()` 的相同。允使用B_RAW志，但它一点也不会影 `isa_dmadone()`。

- `int isa_dmastatus(int channel_number)`

返回当前中剩余的字数。在 `isa_dmastart()` 中置了B_READ的情况下，返回的数字一定不会等于零。束它会被自位到缓冲区的度。正式的用法是在信号指示已完成剩余的字数。如果字数不0，此次可能有。

- `int isa_dmastop(int channel_number)`

放当前的并返回剩余未的字数。

9.8. xxx_isa_probe

个函数探测是否存在。如果程序支持自配置的某些部分（如中断向量或内存地址），自必在此例程中完成。

于任意其他，如果不能到，或者到但自失，或者生某些其他，当返回一个正的。如果不存在必返回 `ENXIO`。其他可能表示其他条件。零或意味着成功。大多数程序返回零表示成功。

当PnP支持多个接口使用返回。例如，不同程序支持老的兼容接口和新的接口。个程序都将。在探例程中返回高的程序得先（句，返回0的程序具有最高的先，返回-1的其次，返回-2的更后，等等）。，支持老接口的将被老程序理（其当从探例程中返回-1），而同也支持新接口的将由新程序理（其当从探例程中返回0）。

描述符xxx_softc由系在用探例程之前分配。如果探例程返回，描述符会被系自取消分配。因此如果出探，程序必保取消分配探期它使用的所有源，且保没有什能阻止描述符被安全地取消分配。如果探成功完成，描述符将由系保存并在以后例程xxx_isa_attach()。如果程序返回，就不能保它将得最高先且其接例程会被用。因此情况下它也必在返回前放所有的源，并在需要的候在接例程中重新分配它。当xxx_isa_probe()返回0，在返回前放源也是一个好主意，而且中矩的程序当做。但在放源会存在某些的情况下，允程序在从探例程返回0和接例程的行之保持源。

典型的探例程以取得描述符和元号始：

```
struct xxx_softc *sc = device_get_softc(dev);
int unit = device_get_unit(dev);
int pnperror;
int error = 0;

sc->dev = dev; /* 接回来 */
sc->unit = unit;
```

然后PnP。是通一个包含PnP ID列表的表的行的。此表包含个程序支持的PnP ID和以人工可形式出的些ID的型号的描述。

```
pnperror=ISA_PNP_PROBE(device_get_parent(dev), dev,
xxx_pnp_ids); if(pnperror == ENXIO) return ENXIO;
```

ISA_PNP_PROBE的如下：如果（元）没有被作PnP到，返回ENOENT。如果被作PnP到，但到的ID不匹配表中的任一ID，返回ENXIO。最后，如果能支持PnP且匹配表中的一个ID，返回0，并且由device_set_desc()从表中取得当的描述行置。

如果程序支持PnP，情况看起来如下：

```
if(pnperror != 0)
    return pnperror;
```

对于不支持PnP的程序不需要特殊处理，因程序会空的PnP ID表，且如果在PnP上用会得到ENXIO。

探测例程通常至少需要某些最少量的资源，如I/O端口号，来并探测。对于不同的硬件，程序可能会自其他必需的资源。PnP的所有资源由PnP子系先置，因此程序不需要自己它。

通常所需的最少信息就是端口号。然后某些允许从配置寄存器中取得其余信息（尽管不是所有的都）。因此首先我取得端口起始：

```
sc-port0 = bus_get_resource_start(dev,
    SYS_RES_IOPORT, 0 /*rid*/); if(sc-port0 == 0) return ENXIO;
```

基端口地址被保存在softc中，以便将来使用。如果需要常使用端口，次都用资源函数将会慢的无法忍受。如果我没有得到端口，返回即可。相反，一些程序相当明，探测所有可能的端口，如下：

```

/* 此所有可能的基I/O端口地址表 */
static struct xxx_allports {
    u_short port; /* 端口地址 */
    short used; /* 旗：此端口是否已被其他元使用 */
} xxx_allports = {
    { 0x300, 0 },
    { 0x320, 0 },
    { 0x340, 0 },
    { 0, 0 } /* 表束 */
};

...
int port, i;
...

port = bus_get_resource_start(dev, SYS_RES_IOPORT, 0 /*rid*/);
if(port !=0 ) {
    for(i=0; xxx_allports[i].port!=0; i++) {
        if(xxx_allports[i].used || xxx_allports[i].port != port)
            continue;

        /* 到了 */
        xxx_allports[i].used = 1;
        /* 在已知端口上探 */
        return xxx_really_probe(dev, port);
    }
    return ENXIO; /* 端口无法或已被使用 */
}

/* 在需要猜端口的候才会到这儿 */
for(i=0; xxx_allports[i].port!=0; i++) {
    if(xxx_allports[i].used)
        continue;

    /* 已被使用 - 即使我在此端口什也没有
     * 至少我以后不会再次探
     */
    xxx_allports[i].used = 1;

    error = xxx_really_probe(dev, xxx_allports[i].port);
    if(error == 0) /* 在那个端口到一个 */
        return 0;
}
/* 探所有可能的地址，但没有可用的 */
return ENXIO;

```

当然，做这些事情通常使用程序的 `identify()` 例程。但可能有一个正当的理由来表明它在函数 `probe()` 中完成更好：如果探测会一些其他敏感数据。探测例程按旗 `sensitive` 排序：敏感数据首先被探测，然后是其他数据。但 `identify()` 例程在所有探测之前被调用，因此它不会考虑敏感数据并可能乱搞些数据。

在，我得到起始端口以后就需要置端口数（PnP除外），因内核在配置文件中没有个信息。

```
if(pnperror /* 只非PnP */
    bus_set_resource(dev, SYS_RES_IOPORT, 0, sc-port0,
        XXX_PORT_COUNT)0)
    return ENXIO;
```

最后分配并激活一片端口地址空（特殊start和end意思是“使用我通bus_set_resource() 置的那些”）：

```
sc-port0_rid = 0;
sc-port0_r = bus_alloc_resource(dev, SYS_RES_IOPORT,
    port0_rid,
    /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

if(sc-port0_r == NULL)
    return ENXIO;
```

在可以端口映射的寄存器后，我就可以以某方式向写入数据并是否如我期望的那作出反。如果没有，明可能其他的在个地址上，或者个地址上根本没有。

通常程序直到接例程才会置中断理函数。之前我替代以模式行探，超以DELAY()。探例程必保不能永久挂起，上的所有等待必在超内完成。如果不在段内，可能出故障或配置，程序必返回，当定超隔，一些外以保可：尽管假定DELAY()在任何机器上都延相同数量的，但随具体CPU的不同，此函数是有一定的差幅度。

如果探例程真的想中断是否真的工作，它可以也配置和探中断。但不建。

```
/* 以重依赖于具体的方式 */
if(error = xxx_probe_ports(sc))
    goto bad; /* 返回前放源 */
```

依赖于所切的型号，函数xxx_probe_ports()也可能置描述。但如果只支持一型号，也可以硬的形式完成。当然，于PnP，PnP支持从表中自置描述。

```
if(pnperror)
    device_set_desc(dev, "Our device model 1234");
```

探例程当或者通取配置寄存器来所有源的，或者保由用式置。我将假定一个板上内存的例子。探例程当尽可能是非入式的，分配和其余源功能性工作就可以更好地留接例程来做。

内存地址可以在内核配置文件中指定，或者某些可以在非易失性配置寄存器中先配置。如果做法均可用却不同，那当用个？可能用在内核配置文件中明置地址，但他知道自己在干什么，当先使用个。一个的例子可能是：

```

/* 首先输出配置地址 */
sc-mem0_p = bus_get_resource_start(dev, SYS_RES_MEMORY, 0 /*rid*/);
if(sc-mem0_p == 0) { /* 没有, 用没指定 */
    sc-mem0_p = xxx_read_mem0_from_device_config(sc);

    if(sc-mem0_p == 0)
        /* 从配置寄存器也到不了儿 */
        goto bad;
} else {
    if(xxx_set_mem0_address_on_device(sc) 0)
        goto bad; /* 不支持那地址 */
}

/* 就像端口, 置内存大小,
 * 于某些, 内存大小不是常数,
 * 而当从配置寄存器中取, 以的不同型号
 * 一个是用把内存大小置“msize”配置源,
 * 由ISA自理
 */
if(pnperror) { /*非PnP */
    sc-mem0_size = bus_get_resource_count(dev, SYS_RES_MEMORY, 0 /*rid*/);
    if(sc-mem0_size == 0) /* 用没有指定 */
        sc-mem0_size = xxx_read_mem0_size_from_device_config(sc);

    if(sc-mem0_size == 0) {
        /* 假定是非常老的一型号, 没有自配置特性,
         * 用也没有偏好置, 因此假定最低要求的情况
         * (当然, 真将根据程序而不同)
         */
        sc-mem0_size = 8*1024;
    }

    if(xxx_set_mem0_size_on_device(sc) 0)
        goto bad; /* 不支持那个大小 */

    if(bus_set_resource(dev, SYS_RES_MEMORY, /*rid*/0,
        sc-mem0_p, sc-mem0_size)0)
        goto bad;
} else {
    sc-mem0_size = bus_get_resource_count(dev, SYS_RES_MEMORY, 0 /*rid*/);
}

```

似, 很容易IRQ和DRQ所用的源。

如果一切行正常, 然后就可以放所有源并返回成功。

```

xxx_free_resources(sc);
return 0;

```

最后，处理棘手情况。所有资源应当在返回前被释放。我利用一个变量来跟踪资源是否被释放。如果资源在之前被零化，因此我能看出是否分配了某些资源：如果分配某些资源的描述符非零。

```
bad:
xxx_free_resources(sc);
if(error)
    return error;
else /* 一切未知 */
    return ENXIO;
```

这是完整的探测例程。资源的释放从多个地方完成，因此将它放到一个函数中，看起来可能像下面的样子：


```

static void
xxx_free_resources(sc)
    struct xxx_softc *sc;
{
    /* 释放资源, 如果非0则放 */

    /* 中断处理函数 */
    if(sc-intr_r) {
        bus_teardown_intr(sc-dev, sc-intr_r, sc-intr_cookie);
        bus_release_resource(sc-dev, SYS_RES_IRQ, sc-intr_rid,
            sc-intr_r);
        sc-intr_r = 0;
    }

    /* 我分配的所有内存 */
    if(sc-data_p) {
        bus_dmamap_unload(sc-data_tag, sc-data_map);
        sc-data_p = 0;
    }
    if(sc-data) { /* sc-data_map等于0有可能合法 */
        /* the map will also be freed */
        bus_dmamem_free(sc-data_tag, sc-data, sc-data_map);
        sc-data = 0;
    }
    if(sc-data_tag) {
        bus_dma_tag_destroy(sc-data_tag);
        sc-data_tag = 0;
    }

    ... 如果有, 释放其他的映射和 ...

    if(sc-parent_tag) {
        bus_dma_tag_destroy(sc-parent_tag);
        sc-parent_tag = 0;
    }

    /* 释放所有资源 */
    if(sc-mem0_r) {
        bus_release_resource(sc-dev, SYS_RES_MEMORY, sc-mem0_rid,
            sc-mem0_r);
        sc-mem0_r = 0;
    }
    ...
    if(sc-port0_r) {
        bus_release_resource(sc-dev, SYS_RES_IOPORT, sc-port0_rid,
            sc-port0_r);
        sc-port0_r = 0;
    }
}

```

9.9. xxx_isa_attach

如果探测例程返回成功并且系将那个程序，将例程 将程序接到系。如果探测例程返回0，
将例程期望 接收完整的softc，此由探测例程置。同，如果探测例程 返回0，它可能期望个的
将例程在将来的某点被用。如果 探测例程返回，程序可能不会作此假。

如果成功完成，将例程返回0，否返回。

将例程的跟探测例程相似，将一些常用数据取到一些更容易 的量中。

```
struct xxx_softc *sc = device_get_softc(dev);
int unit = device_get_unit(dev);
int error = 0;
```

然后分配并激活所需源。由于端口通常在从探测返回前就被放，因此需要重新分配。我希望探测
例程已当地置了 所有的源，并将它保存在softc中。如果探测例程留下了一些被分配的
源，就不需要再次分配（重新分配被）。

```
sc-port0_rid = 0;
sc-port0_r = bus_alloc_resource(dev, SYS_RES_IOPORT, port0_rid,
    /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

if(sc-port0_r == NULL)
    return ENXIO;

/* 板上内存 */
sc-mem0_rid = 0;
sc-mem0_r = bus_alloc_resource(dev, SYS_RES_MEMORY, mem0_rid,
    /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

if(sc-mem0_r == NULL)
    goto bad;

/* 取得虚地址 */
sc-mem0_v = rman_get_virtual(sc-mem0_r);
```

DMA求通道(DRQ)以相似方式被分配。使用 `isa_dma*()` 函数族行初始化。例如：

```
isa_dmacascade(sc-drq0);
```

中断求(IRQ)有点特殊。除了分配以外，程序的中断理 函数也当与它。在古老的ISA
程序中，由系中断理 函数的参量是元号。但在代程序中，按照定，建 指向softc的指。
一个很重要的原因在于当softc被分配后，从softc取得元号很容易，而从元号取得softc很困。同，
个 定也使得用于不同用的程序看起来一，并允它共享代： 个有其自己的探，接，分
和其他相的例程，而它之可以共享大的程序代。

```

sc-intr_rid = 0;
sc-intr_r = bus_alloc_resource(dev, SYS_RES_MEMORY, intr_rid,
    /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

if(sc-intr_r == NULL)
    goto bad;

/*
 * 假定XXX_INTR_TYPE的定义依赖于程序的类型,
 * 例如INTR_TYPE_CAM用于CAM的程序
 */
error = bus_setup_intr(dev, sc-intr_r, XXX_INTR_TYPE,
    (driver_intr_t *) xxx_intr, (void *) sc, intr_cookie);
if(error)
    goto bad;

```

如果程序需要与内存行DMA，内存应当按前述方式分配：

```

error=bus_dma_tag_create(NULL, /*alignment*/ 4,
    /*boundary*/ 0, /*lowaddr*/ BUS_SPACE_MAXADDR_24BIT,
    /*highaddr*/ BUS_SPACE_MAXADDR, /*filter*/ NULL, /*filterarg*/ NULL,
    /*maxsize*/ BUS_SPACE_MAXSIZE_24BIT,
    /*nsegments*/ BUS_SPACE_UNRESTRICTED,
    /*maxsegsz*/ BUS_SPACE_MAXSIZE_24BIT, /*flags*/ 0,
    parent_tag);
if(error)
    goto bad;

/* 很多西是从父承而来
 * 假sc-data指向存共享数据的，例如一个缓冲区可能是：
 * struct {
 *     u_short rd_pos;
 *     u_short wr_pos;
 *     char    bf[XXX_RING_BUFFER_SIZE]
 * } *data;
 */
error=bus_dma_tag_create(sc-parent_tag, 1,
    0, BUS_SPACE_MAXADDR, 0, /*filter*/ NULL, /*filterarg*/ NULL,
    /*maxsize*/ sizeof(* sc-data), /*nsegments*/ 1,
    /*maxsegsz*/ sizeof(* sc-data), /*flags*/ 0,
    data_tag);
if(error)
    goto bad;

error = bus_dmamem_alloc(sc-data_tag, data, /* flags*/ 0,
    data_map);
if(error)
    goto bad;

/* 在data_p的情况下，xxx_alloc_callback()只是将物理地址
 * 保存到作其参量去的指中。
 * 参看于内存映射一中的内容。
 * 其可以像：
 *
 * static void
 * xxx_alloc_callback(void *arg, bus_dma_segment_t *seg,
 *     int nseg, int error)
 * {
 *     *(bus_addr_t *)arg = seg[0].ds_addr;
 * }
 */
bus_dmamap_load(sc-data_tag, sc-data_map, (void *)sc-data,
    sizeof (* sc-data), xxx_alloc_callback, (void *) data_p,
    /*flags*/0);

```

分配了所有的源后，当被初始化。初始化可能包括所有特性，保它起作用。

```
if(xxx_initialize(sc) < 0)
    goto bad;
```

子系将自己在控制台上打印由探测例程置的描述。但
外信息，也是可能的，例如：

如果程序想打印一些于的

```
device_printf(dev, "has on-card FIFO buffer of %d bytes\n", sc-fifosize);
```

如果初始化例程遇到任何，建返回之前打印有信息。

接例程的最后一是将接到内核中的功能子系。完成
、网、CAM SCSI等等。

个的精方式依赖于程序的型：字符、

如果所有均工作正常返回成功。

```
error = xxx_attach_subsystem(sc);
if(error)
    goto bad;

return 0;
```

最后，理棘手情况。返回前，所有源当被取消分配。
之前被零化，因此我能出是否分配了某些源：如果分配它的描述符非零。

我利用一个事：softc我

```
bad:

xxx_free_resources(sc);
if(error)
    return error;
else /* exact error is unknown */
    return ENXIO;
```

就是接例程的全部。

9.10. xxx_isa_detach

如果程序中存在个函数，且程序被可加模，程序具有被卸的能力。如果硬件支持拔，
是个很重要的特性。但ISA不支持拔，因此个特性于ISA不是特重要。卸
程序的能力可能在有用，但很多情况下只有在老版本的程序莫名其妙地住系
的情况下才需要安装新版本的程序，并且无论如何都需要重，使得花精力写分例程
有些不得。一个宣称卸允在用于生的机器上升程序的点看起来似乎更多的只是理而已。升程序是一危
的操作，决不应当在用于生的机器上行（并且当系行于安全模式也是不被允
的）。然而，出于完整性考，是会提供分例程。

如果程序成功分，分例程返回0，否返回。

分门是接的像。要做的第一件事情就是将程序从内核子系分。如果当前正打着，程序有个：拒分或者制并行分。用方式取决于特定内核子系行制的能力和程序作者的偏好。通常制似乎是更好的。

```
struct xxx_softc *sc = device_get_softc(dev);
int error;

error = xxx_detach_subsystem(sc);
if(error)
    return error;
```

下一，程序可能希望位硬件到某一致的状。包括停止任何将要行的，禁用DMA通道和中断以避免破坏内存。于大多数程序而言，正是例程所做的，因此如果程序中包括例程，我只要用它就可以了。

```
xxx_isa_shutdown(dev);
```

最后放所有源并返回成功。

```
xxx_free_resources(sc);
return 0;
```

9.11. xxx_isa_shutdown

当系要的时候用此例程。通它使硬件入某一致的状。于大多数ISA而言不需要特殊作，因此个函数并非真正必需，因不管重会被重新初始化。但有些必按特定，以保在重后能被正地到（于很多使用私有特的有用）。很多情况下，在寄存器中禁用DMA和中断，并停止将要行的是个好主意。切作取决于硬件，因此我无法在此。

9.12. xxx_intr

当收到来自特定中断就会用中断理函数。ISA不支持中断共享（某些特殊情况例外），因此上如果中断理函数被用，几乎可以信中断是来自其。然而，中断理函数必寄存器并保中断是由它的生的。如果不是，中断理函数当返回。

ISA程序的旧定是取元号作参量。在已，当用bus_setup_intr()新程序接收任何在接例程中他指定的参量。根据新定，它当是指向 softc的指。因此中断理函数通常像下面那始：

```
static void
xxx_intr(struct xxx_softc *sc)
{
```

它行在由bus_setup_intr()的中断型参数指定的中断先上。意味着禁用所有其他同型的中断和所有件中断。

了避免争，中断理例程通写成循形式：

```
while(xxx_interrupt_pending(sc)) {  
    xxx_process_interrupt(sc);  
    xxx_acknowledge_interrupt(sc);  
}
```

中断处理函数必须只向源设备应答中断，但不能向中断控制器应答，后者由系统管理。

Chapter 10. PCI

本章将介绍FreeBSD上的PCI上的设备驱动程序而提供的机制。

10.1. 探测与连接

儿童的信息是于PCI设备如何迭代通未连接的，并看新加的kld是否会连接其中一个。

10.1.1. 示例程序源代码(mypci.c)

```
/*
 * 与PCI函数交互的KLD
 *
 * Murray Stokely
 */

#include sys/param.h      /* kernel.h中使用的定义 */
#include sys/module.h
#include sys/systm.h
#include sys/errno.h
#include sys/kernel.h     /* 模块初始化中使用的类型 */
#include sys/conf.h       /* cdevsw */
#include sys/uio.h        /* uio */
#include sys/malloc.h
#include sys/bus.h        /* pci用到的、原型 */

#include machine/bus.h
#include sys/rman.h
#include machine/resource.h

#include dev/pci/pcivar.h /* 为了使用get_pci宏! */
#include dev/pci/pcireg.h

/* softc保存我每个例的数据。 */
struct mypci_softc {
    device_t    my_dev;
    struct cdev *my_cdev;
};

/* 函数原型 */
static d_open_t    mypci_open;
static d_close_t   mypci_close;
static d_read_t    mypci_read;
static d_write_t   mypci_write;

/* 字符入口点 */

static struct cdevsw mypci_cdevsw = {
    .d_version =    D_VERSION,
```



```

    .d_open =    mypci_open,
    .d_close =  mypci_close,
    .d_read =   mypci_read,
    .d_write =  mypci_write,
    .d_name =   "mypci",
};

/*
 * 在cdevsw例程中，我通过体cdev中的成员si_drv1出我的softc。
 * 当我建立/dev，在我的已附着的例程中，
 * 我置一个量指向我的softc。
 */

int
mypci_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct mypci_softc *sc;

    /* Look up our softc. */
    sc = dev-si_drv1;
    device_printf(sc-my_dev, "Opened successfully.\n");
    return (0);
}

int
mypci_close(struct cdev *dev, int fflag, int devtype, struct thread *td)
{
    struct mypci_softc *sc;

    /* Look up our softc. */
    sc = dev-si_drv1;
    device_printf(sc-my_dev, "Closed.\n");
    return (0);
}

int
mypci_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct mypci_softc *sc;

    /* Look up our softc. */
    sc = dev-si_drv1;
    device_printf(sc-my_dev, "Asked to read %d bytes.\n", uio-uio_resid);
    return (0);
}

int
mypci_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct mypci_softc *sc;

```

```

/* Look up our softc. */
sc = dev-si_drv1;
device_printf(sc-my_dev, "Asked to write %d bytes.\n", uio-uio_resid);
return (0);
}

/* PCI支持函数 */

/*
 * 将某个设备的与一个程序支持的相比。
 * 如果相符，置描述字符并返回成功。
 */
static int
mypci_probe(device_t dev)
{
    device_printf(dev, "MyPCI Probe\nVendor ID : 0x%x\nDevice ID : 0x%x\n",
        pci_get_vendor(dev), pci_get_device(dev));

    if (pci_get_vendor(dev) == 0x11c1) {
        printf("We've got the Winmodem, probe successful!\n");
        device_set_desc(dev, "WinModem");
        return (BUS_PROBE_DEFAULT);
    }
    return (ENXIO);
}

/* 只有当探测成功时才用接口函数 */

static int
mypci_attach(device_t dev)
{
    struct mypci_softc *sc;

    printf("MyPCI Attach for : deviceID : 0x%x\n", pci_get_devid(dev));

    /* Look up our softc and initialize its fields. */
    sc = device_get_softc(dev);
    sc-my_dev = dev;

    /*
     * Create a /dev entry for this device. The kernel will assign us
     * a major number automatically. We use the unit number of this
     * device as the minor number and name the character device
     * "mypciunit".
     */
    sc-my_cdev = make_dev(mypci_cdevsw, device_get_unit(dev),
        UID_ROOT, GID_WHEEL, 0600, "mypci%u", device_get_unit(dev));
    sc-my_cdev-si_drv1 = sc;
    printf("Mypci device loaded.\n");
    return (0);
}

```

```

}

/* 分断。 */

static int
mypci_detach(device_t dev)
{
    struct mypci_softc *sc;

    /* Teardown the state in our_softc created in our attach routine. */
    sc = device_get_softc(dev);
    destroy_dev(sc->my_cdev);
    printf("Mypci detach!\n");
    return (0);
}

/* 系断期在sync之后用。 */

static int
mypci_shutdown(device_t dev)
{
    printf("Mypci shutdown!\n");
    return (0);
}

/*
 * 挂起例程。
 */
static int
mypci_suspend(device_t dev)
{
    printf("Mypci suspend!\n");
    return (0);
}

/*
 * 恢复（重新始）例程。
 */
static int
mypci_resume(device_t dev)
{
    printf("Mypci resume!\n");
    return (0);
}

static device_method_t mypci_methods[] = {
    /* 接口 */
    DEVMETHOD(device_probe,      mypci_probe),

```

```

    DEVMETHOD(device_attach,    mypci_attach),
    DEVMETHOD(device_detach,    mypci_detach),
    DEVMETHOD(device_shutdown,  mypci_shutdown),
    DEVMETHOD(device_suspend,   mypci_suspend),
    DEVMETHOD(device_resume,    mypci_resume),

    { 0, 0 }
};

static devclass_t mypci_devclass;

DEFINE_CLASS_0(mypci, mypci_driver, mypci_methods, sizeof(struct mypci_softc));
DRIVER_MODULE(mypci, pci, mypci_driver, mypci_devclass, 0, 0);

```

10.1.2. 示例程序的Makefile

```

# 程序mypci的Makefile

KMOD=    mypci
SRCS=    mypci.c
SRCS+=   device_if.h bus_if.h pci_if.h

.include bsd.kmod.mk

```

如果将上面的源文件和 Makefile放入一个目录，可以运行 `make` 示例程序。有，可以运行 `make load` 将程序装到当前正在运行的内核中，而 `make unload` 可在装后卸程序。

10.1.3. 更多源

- [PCI Special Interest Group](#)
- PCI System Architecture, Fourth Edition by Tom Shanley, et al.

10.2. 设备源

FreeBSD 从父设备源提供了一面向对象的机制。几乎所有设备都是某类型的设备（PCI，ISA，USB，SCSI 等等）的孩子成，并且有些设备需要从他父设备取源（例如内存段，中断，或者DMA通道）。

10.2.1. 基地址寄存器

为了PCI设备做些有用的事情，需要从PCI配置空间取 *Base Address Registers* (BARs)。取BAR的PCI特定的被抽象在函数 `bus_alloc_resource()` 中。

例如，一个典型的设备程序可能在 `attach()` 函数中有些类似于下面的东西：

```

sc-bar0id = PCIR_BAR(0);
sc-bar0res = bus_alloc_resource(dev, SYS_RES_MEMORY, sc-bar0id,
                                0, ~0, 1, RF_ACTIVE);
if (sc-bar0res == NULL) {
    printf("Memory allocation of PCI base register 0 failed!\n");
    error = ENXIO;
    goto fail1;
}

sc-bar1id = PCIR_BAR(1);
sc-bar1res = bus_alloc_resource(dev, SYS_RES_MEMORY, sc-bar1id,
                                0, ~0, 1, RF_ACTIVE);
if (sc-bar1res == NULL) {
    printf("Memory allocation of PCI base register 1 failed!\n");
    error = ENXIO;
    goto fail2;
}
sc-bar0_bt = rman_get_bustag(sc-bar0res);
sc-bar0_bh = rman_get_bushandle(sc-bar0res);
sc-bar1_bt = rman_get_bustag(sc-bar1res);
sc-bar1_bh = rman_get_bushandle(sc-bar1res);

```

每个基址寄存器的句柄被保存在`softc`中，以便以后可以使用它向寄存器写入。

然后就能使用这些句柄与`bus_space_*`函数一起读写寄存器。例如，程序可能包含如下的快捷函数，用来读取板子特定的寄存器：

```

uint16_t
board_read(struct ni_softc *sc, uint16_t address)
{
    return bus_space_read_2(sc-bar1_bt, sc-bar1_bh, address);
}

```

类似的，可以用下面的函数写寄存器：

```

void
board_write(struct ni_softc *sc, uint16_t address, uint16_t value)
{
    bus_space_write_2(sc-bar1_bt, sc-bar1_bh, address, value);
}

```

有些函数以8位，16位和32位的版本存在，应当相应地使用`bus_space_{read|write}_{1|2|4}`。

在 FreeBSD 7.0 和更高版本中，可以用 `bus_*` 函数来代替 `bus_space_*`。 `bus_*` 函数使用的参数是 `struct resource *` 指针，而不是 `bus tag` 和句柄。因此，就可以将 `softc` 中的 `bus tag` 和 `bus 句柄` 个成员量去掉，并将 `board_read()` 函数改写为：



```
uint16_t
board_read(struct ni_softc *sc, uint16_t address)
{
    return (bus_read(sc->bar1res, address));
}
```

10.2.2. 中断

中断按照和分配内存资源相似的方式从面向对象的代码分配。首先，必须从父资源分配IRQ资源，然后必须设置中断处理函数来处理一个IRQ。

再一次，来自 `attach()` 函数的例子比文字更具明性。

```
/* 取得IRQ资源 */

sc->irqid = 0x0;
sc->irqres = bus_alloc_resource(dev, SYS_RES_IRQ, (sc->irqid),
                                0, ~0, 1, RF_SHAREABLE | RF_ACTIVE);
if (sc->irqres == NULL) {
    printf("IRQ allocation failed!\n");
    error = ENXIO;
    goto fail3;
}

/* 在我当前设置中断处理函数 */

error = bus_setup_intr(dev, sc->irqres, INTR_TYPE_MISC,
                        my_handler, sc, (sc->handler));
if (error) {
    printf("Couldn't set up irq\n");
    goto fail4;
}
```

在驱动的分例程中必须注意一些事项。必须停止所有的中断流，并移除中断处理函数。一旦 `bus_tearardown_intr()` 返回，就知道驱动的中断处理函数不会再被调用，并且所有可能已调用了驱动的中断处理函数的线程都已返回。由于此函数可以睡眠，因此用此函数时必须不能有任何互斥体。

10.2.3. DMA

本书已指出，只是由于历史原因而指出。处理某些事项的适当方法是使用 `bus_space_dma*()` 函数。当更新一章以反映那用法时，该段就可能被去掉。然而，目前API不断有些问题，因此一旦它固定下来后，更新一章来反映那些改动就很好了。

在PC上，想行主控DMA的外必理物理地址，由于
仍是个。幸的是，有个函数，`vtophys()`可以助我。

FreeBSD使用虚内存并且只理虚地址，

```
#include vm/vm.h
#include vm/pmap.h

#define vtophys(virtual_address) (...)
```

然而个解决法在alpha上有点不一，并且我真正想要的是一个称`vtobus()`的函数。

```
#if defined(__alpha__)
#define vtobus(va)      alpha_XXX_dmamap((vm_offset_t)va)
#else
#define vtobus(va)      vtophys(va)
#endif
```

10.2.4. 取消分配源

取消`attach()`期分配的所有源非常重要。必小心慎，即使在失的条件下也要保取消分配那些正的
西，当的程序去掉后系仍然可以使用。

Chapter 11. 通用方法SCSI控制器

11.1. 提

本文假定读者对FreeBSD的驱动程序和SCSI有大致了解，本文中很多信息是从以下程序中：

- ncr (/sys/pci/ncr.c) 由Wolfgang Stanglmeier and Stefan Esser写
- sym (/sys/dev/sym/sym_hipd.c) 由Gerard Roudier写
- aic7xxx (/sys/dev/aic7xxx/aic7xxx.c) 由Justin T. Gibbs写

和从CAM的代本身（作者 Justin T. Gibbs, /sys/cam/*）中摘。当一些解决方法看起来 具性，并且基本上是从 Justin T. Gibbs 的代中一字不差地摘，我将其“recommended”。

本文以代例子行明。尽管有例子中包含很多，并且 看起来很像真正代，但它仍然只是代。写是了以可理解的方式来展示概念。于真正的程序，其它方法可能更模化，并且 更加高效。文也硬件行抽象，于那些会模糊我所要展示的概念的，或被在读者手册的其他章中已有描述的也做同理。些通常以用具有描述性名字的函数、注或句的形式展。幸的是，具有价的完整例子，包括所有，可以在真正的程序中到。

11.2. 通用基

CAM代表通用方法（Common Access Method）。它以SCSI方式址 I/O。就允将通用程序和控制在I/O的程序分来：例如磁程序能同控制SCSI、IDE、且/或任何其他上的磁，磁程序部分不必新的I/O而重写（或拷修改）。个最重要的活体是：

- 外模 - 外（磁，磁，CD-ROM等）的程序
- SCSI接口模(SIM) - 接到I/O，如SCSI或IDE，的主机配器程序。

外程序从OS接收求，将它SCSI命令序列并将 些SCSI命令到SCSI接口模。SCSI接口模将些命令硬件（或者如果硬件不是SCSI，而是例如IDE，也要将些SCSI 命令硬件的native命令）。

由于儿我感兴趣的是写SCSI配器程序，从此始我 将从SIM的角度考所有的事情。

典型的SIM程序需要包括如下的CAM相的文件：

```
#include cam/cam.h
#include cam/cam_ccb.h
#include cam/cam_sim.h
#include cam/cam_xpt_sim.h
#include cam/cam_debug.h
#include cam/scsi/scsi_all.h
```

个SIM程序必做的第一件事情是向CAM子系注册它自己。 在程序的xxx_attach()函数（此和以后的 xxx_用于指唯一的程序名字前）期完成。 xxx_attach()函数自身由系自配置代用，我在此不描述部分代。

需要好几来完成：首先需要分配与SIM的队列：

```
struct cam_devq *devq;

if(( devq = cam_simq_alloc(SIZE) )!=NULL) {
    error; /* 一些处理的代 */
}
```

此 **SIZE** 要分配的队列的大小，它能包含的最大请求数目。它是 SIM 程序在 SCSI 上能并行处理的请求的数目。一般可以如下估算：

```
SIZE = NUMBER_OF_SUPPORTED_TARGETS * MAX_SIMULTANEOUS_COMMANDS_PER_TARGET
```

下一个我的SIM建描述符：

```
struct cam_sim *sim;

if(( sim = cam_sim_alloc(action_func, poll_func, driver_name,
    softc, unit, max_dev_transactions,
    max_tagged_dev_transactions, devq) )!=NULL) {
    cam_simq_free(devq);
    error; /* 一些处理代 */
}
```

注意如果我不能建SIM描述符，我也放 **devq**，因为我其无法做任何其他事情，而且我想内存。

如果SCSI上有多条SCSI，每条需要它自己的 **cam_sim**。

一个有趣的，如果SCSI有不只一条SCSI我做什么， 个需要一个**devq**是几条SCSI？在CAM代的注中出的答案是：任一方式均可，由程序的作者。

参量：

- **action_func** - 指向程序 **xxx_action** 函数的指。

```
static void xxx_action (    struct cam_sim *simunion ccb *ccb);
struct cam_sim *sim, union ccb *ccb ;
```

- **poll_func** - 指向程序 **xxx_poll()**函数的指。

```
static void xxx_poll (    struct cam_sim *sim);
struct cam_sim *sim ;
```

- **driver_name** - 程序的名字，例如 "ncr"或"wds"。
- **softc** - 指向个SCSI 程序的内部描述符的指。个指以后被程序用来取私有数据。

- unit - 控制器元号，例如对于控制器 "wds0" 的此数字将00。
- max_dev_transactions - 无模式下每个SCSI目标的 最大并（simultaneous）事务数。每个一般几乎是等于1，只有非 SCSI才可能例外。此外，如果程序希望并行一个事务的同时准一个事务，可以将其置02，但似乎不得增加并行性。
- max_tagged_dev_transactions - 同的东西，但是在模式下。是SCSI在上起多个事务的方式：每个事务被给予一个唯一的，并被送到。当完成某些事务，它 将结果一同一起送回来，SCSI适配器（和程序）就能知道 个事务完成了。此参量也被是最大深度。它取决于SCSI适配器的能力。

最后我注册与我SCSI适配器的SCSI。

```
if(xpt_bus_register(sim, bus_number) != CAM_SUCCESS) {
    cam_sim_free(sim, /*free_devq*/ TRUE);
    error; /* 一些代理 */
}
```

如果条SCSI有一个devq（即， 我将有多条的看作多个， 个有一条）， 号 是， 否SCSI上的条应当有不同的号。条需要 它自己独的cam_sim。

之后我的控制器完全挂接到CAM系。在 devq的可以被：在所有以后从CAM出的 用中将以sim参量，devq可以由它出。

CAM些事件提供了框架。有些事件来自底（SIM程序）， 有些来自外程序， 有一些来自CAM子系本身。任何程序都可以某些型的事件注册回， 那些事件生它就会被通知。

事件的一个典型例子就是位。个事务和事件以 "path"的方式区分它所作用的。目特定的事件通常在与行事理期生。因此那个事务的路径可以被重用 来告此事件（是安全的，因事件路径的拷是在事件告例程中行的， 而且既不会被deallocate也不作一）。在任何时刻，包括中断例程中，分配路径也是安全的，尽管那会致某些外，并且方法 可能存在的一个是巧那可能没有空内存。于位事件， 我需要定包括上所有在的通配符路径。我就能提前 以后的位事件建路径，避免以后内存不足的：

```
struct cam_path *path;

if(xpt_create_path(path, /*periph*/NULL,
    cam_sim_path(sim), CAM_TARGET_WILDCARD,
    CAM_LUN_WILDCARD) != CAM_REQ_CMP) {
    xpt_bus_deregister(cam_sim_path(sim));
    cam_sim_free(sim, /*free_devq*/TRUE);
    error; /* 一些代理 */
}

softc-wpath = path;
softc-sim = sim;
```

正如所看到的，路径包括：

- 外程序的ID（由于我一个也没有，故此空）
- SIM程序的ID（`cam_sim_path(sim)`）
- 的SCSI目标号（CAM_TARGET_WILDCARD的意思指"所有devices"）
- 子的SCSI LUN号（CAM_LUN_WILDCARD的意思指"所有LUNs"）

如果程序不能分配个路径，它将不能正常工作，因此那情况下 我卸除（dismantle）那个SCSI。

我在`softc`中保存路径指以便以后 使用。之后我保存sim的（或者如果我意，也可以在从`xxx_probe()`退出它）。

就是最低要求的初始化所需要做的一切。了把事情做正无， 剩下一个。

于SIM程序，有一个特殊感兴趣的事件：何目标被 不到了。情况下位与个的SCSI商可能是个好主意。因此我 个事件向CAM注册一个回。通型的求来求CAM控制上的CAM作，求就被到CAM：（注：参看下面示例代和原文）

```
struct ccb_setasync csa;

xpt_setup_ccb(csa.ccb_h, path, /*先*/5);
csa.ccb_h.func_code = XPT_SASYNC_CB;
csa.event_enable = AC_LOST_DEVICE;
csa.callback = xxx_async;
csa.callback_arg = sim;
xpt_action((union ccb *)csa);
```

在我看一下`xxx_action()`和`xxx_poll()`的程序入口点。

```
static void xxx_action (    struct cam_sim *simunion ccb *ccb);
struct cam_sim *sim, union ccb *ccb ;
```

CAM子系的求采取某些作。Sim描述了求的SIM，CCB 求本身。CCB代表"CAM Control Block"。它是很多特定 例的合， 个例某些型的事描述参量。所有些例共享 存着参量公共部分的CCB部。（注：一段不很准， 自行参考原文）

CAM既支持SCSI控制器工作于起者(initiator)("normal") 模式，也支持SCSI控制器工作于目标(target)（模SCSI）模式。儿 我只考与起者模式有的部分。

定了几个函数和宏（句，方法）来sim中公共数据：

- `cam_sim_path(sim)` - 路径ID（参上面）
- `cam_sim_name(sim)` - sim的名字
- `cam_sim_softc(sim)` - 指向softc（程序私有数据）的指
- `cam_sim_unit(sim)` - 元号
- `cam_sim_bus(sim)` - ID

了，`xxx_action()`可以使用些 函数得到元号和指向它的softc的指。

请求的类型被存储在 `ccb-ccb_h.func_code`。因此，通常 `xxx_action()` 由一个大的 switch 组成：

```
struct xxx_softc *softc = (struct xxx_softc *) cam_sim_softc(sim);
struct ccb_hdr *ccb_h = ccb-ccb_h;
int unit = cam_sim_unit(sim);
int bus = cam_sim_bus(sim);

switch(ccb_h->func_code) {
case ...:
    ...
default:
    ccb_h->status = CAM_REQ_INVALID;
    xpt_done(ccb);
    break;
}
```

从 default case 语句部分可以看出（如果收到未知命令），命令的返回值被置到 `ccb-ccb_h.status` 中，并且通过 `xpt_done(ccb)` 将整个 CCB 返回到 CAM 中。

`xpt_done()` 不必从 `xxx_action()` 中调用：例如 I/O 请求可以在 SIM 程序和/或它的 SCSI 控制器中排队。（注：它指 I/O 请求？）然后，当 `post` 一个中断信号，指示此请求的处理已束，`xpt_done()` 可以从中断处理例程中被调用。

如上，CCB 状态不是被返回，而是始终有某状态。CCB 被 `xxx_action()` 例程前，其取得状态 `CCB_REQ_INPROG`，表示其正在行中。`/sys/cam/cam.h` 中定义了数量人的状态，它可能非常尽地表示请求的状态。更有趣的是，状态上是一个枚状态（低6位）和一些可能输出的附加（似）旗位（高位）的“位或(bitwise or)”。枚会在以后更地。它的可以在概 (Errors Summary section) 到。可能的状态旗：

- `CAM_DEV_QFRZN` - 当处理 CCB，如果 SIM 程序得到一个重（例如，程序不能或反了 SCSI），它当调用 `xpt_freeze_simq()` 请求列，把此列的其他已入但尚未被处理的 CCB 返回到 CAM 列，然后有 CCB 置个旗并用 `xpt_done()`。个旗会使得 CAM 子系理后解列。
- `CAM_AUTOSNS_VALID` - 如果返回条件，且 CCB 中未置旗 `CAM_DIS_AUTONSENSE`，SIM 程序必自进行 REQUEST SENSE 命令来从抽取 sense（展信息）数据。如果个成功，sense 数据当被保存在 CCB 中且置此旗。
- `CAM_RELEASE_SIMQ` - 似于 `CAM_DEV_QFRZN`，但用于 SCSI 控制器自身出（或源短缺）的情况。此后控制器的所有请求会被 `xpt_freeze_simq()` 停止。SIM 程序克服短缺情况，并通过返回置了此旗的 CCB 通知 CAM 后，控制器列将会被重新。
- `CAM_SIM_QUEUED` - 当 SIM 将一个 CCB 放入其请求列当置此旗（或当 CCB 出但尚未返回 CAM 去掉）。在此旗没有在 CAM 代的任何地方使用，因此其目的粹用于断）。

函数 `xxx_action()` 不允许睡眠，因此源的所有同必通 SIM 或列来完成。除了前述的旗外，CAM 子系提供了函数 `xpt_release_simq()` 和 `xpt_release_devq()` 来直接解列，而不必将 CCB 到 CAM。

CCB 部包含如下字段：

- `path` - 请求的路径 ID

- *target_id* - 请求的目标ID
- *target_lun* - 目标的LUN ID
- *timeout* - 一个命令的超时间隔，以毫秒
- *timeout_ch* - 一个SIM程序存储超时间隔函数的方便之所（CAM子系统自身并不对此作任何假设）
- *flags* - 有请求的各个信息位
- *spriv_ptr0*, *spriv_ptr1* - SIM程序保留私用的字段（例如，接到SIM列或SIM私有控制）；上面，它们组合存在：*spriv_ptr0*和*spriv_ptr1*具有型(void *), *spriv_field0*和 *spriv_field1*具有型unsigned long, *sim_priv.entries[0].bytes*和 *sim_priv.entries[1].bytes*与组合的其他形式大小一致的字数据，*sim_priv.bytes*一个倍大小的数

使用CCB的SIM私有字段的建立方法是它定义一些有意义的名字，并且在程序中使用些有意义的名字，就像下面：

```
#define ccb_some_meaningful_name    sim_priv.entries[0].bytes
#define ccb_hcb spriv_ptr1 /* 用于硬件控制 */
```

最常的发起者模式的请求是：

- *XPT SCSI_IO* - 行I/O事

组合ccb的"struct ccb_scsiio csio"例用于参数。它是：

- *cdb_io* - 指向SCSI命令缓冲区的指针或缓冲区本身
- *cdb_len* - SCSI命令长度
- *data_ptr* - 指向数据缓冲区的指针（如果使用分散/集中会好一点）
- *dxfer_len* - 待传输数据的长度
- *sglist_cnt* - 分散/集中段的数目
- *scsi_status* - 返回SCSI状态的地方
- *sense_data* - 命令返回并保存SCSI sense信息的缓冲区（通常情况下，如果没有设置CCB的旗CAM_DIS_AUTONSENSE，假定SIM程序会自行REQUEST SENSE命令）
- *sense_len* - 缓冲区的长度（如果恰巧大于sense_data的大小，SIM程序必须悄悄地采用小）（注：一点改，参考原文及代）
- *resid*, *sense_resid* - 如果数据或SCSI sense返回，它就是返回的剩余（未）数据的数。它看起来并不是特别有意，因此当很算的情况下（例如，数SCSI控制器FIFO缓冲区中的字数），使用近似也同可以。于成功完成的，它必须被置0。
- *tag_action* - 使用的tag的有：
 - CAM_TAG_ACTION_NONE - 事不使用tag
 - MSG_SIMPLE_Q_TAG, MSG_HEAD_OF_Q_TAG, MSG_ORDERED_Q_TAG - 等于相当的tag信息（/sys/cam/scsi/scsi_message.h）；输出类型，SIM程序必须自己处理

处理请求的通常如下：

要做的第一件事情是可能的竞争条件，命令位于队列中不会被中止：

```
struct ccb_scsiio *csio = ccb-csio;

if ((ccb_h-status & CAM_STATUS_MASK) != CAM_REQ_INPROG) {
    xpt_done(ccb);
    return;
}
```

我也我的控制器完全支持：

```
if(ccb_h-target_id > OUR_MAX_SUPPORTED_TARGET_ID
|| ccb_h-target_id == OUR_SCSI_CONTROLLERS_OWN_ID) {
    ccb_h-status = CAM_TID_INVALID;
    xpt_done(ccb);
    return;
}
if(ccb_h-target_lun > OUR_MAX_SUPPORTED_LUN) {
    ccb_h-status = CAM_LUN_INVALID;
    xpt_done(ccb);
    return;
}
```

然后分配我管理请求所需的数据（如相关的硬件控制等）。如果我不能分配SIM队列，下次我有一个挂起的操作，返回 CCB，请求CAM将CCB重新入队。以后当源可用，必须通过返回其状态中置 CAM_SIMQ_RELEASE 位的ccb来解SIM队列。否则，如果所有正常，将CCB与硬件控制（HCB）连接，并将其标志已入。

```
struct xxx_hcb *hcb = allocate_hcb(softc, unit, bus);

if(hcb == NULL) {
    softc-flags |= RESOURCE_SHORTAGE;
    xpt_freeze_simq(sim, /*count*/1);
    ccb_h-status = CAM_REQUEUE_REQ;
    xpt_done(ccb);
    return;
}

hcb-ccb = ccb; ccb_h-ccb_hcb = (void *)hcb;
ccb_h-status |= CAM_SIM_QUEUED;
```

从CCB中提取目标数据到硬件控制。是否要求我分配一个，如果是生成一个唯一的并构造SCSI信息。SIM程序也与商确定彼此支持的最大度、同速率和偏移。

```

hcb-target = ccb_h-target_id; hcb-lun = ccb_h-target_lun;
generate_identify_message(hcb);
if( ccb_h-tag_action != CAM_TAG_ACTION_NONE )
    generate_unique_tag_message(hcb, ccb_h-tag_action);
if( !target_negotiated(hcb) )
    generate_negotiation_messages(hcb);

```

然后设置SCSI命令。可以在CCB中以多种方式指定命令的存贮，有些方式由CCB中的旗帜指定。命令缓冲区可以包含在CCB中或者用指向指向，后者情况下指向可以指向物理地址或虚地址。由于硬件通常需要物理地址，因此我们总是将地址指向物理地址。

不太相关的提示：通常是用`vtophys()`来完成的，但由于特殊的Alpha怪物的之故，除了PCI（它在占SCSI控制器的大多数）程序向Alpha架构的可移植性，必须替代以`vtobus()`来完成。[IMHO 提供一个单独的函数`vtop()`和`ptobus()`，而`vtobus()`只是它的替代，做得要好得多。] 在请求物理地址的情况下，返回有状态`CAM_REQ_INVALID`的CCB是可以的，当前的程序就是那样做的。但也可能像例子（程序中当有不条件做的更直接做法）中那样Alpha特定的代码片断。如果需要物理地址也能或映射回虚地址，但那代价很大，因此我们不那样做。

```

if(ccb_h-flags CAM_CDB_POINTER) {
    /* CDB is a pointer */
    if(!(ccb_h-flags CAM_CDB_PHYS)) {
        /* CDB指向是虚的 */
        hcb-cmd = vtobus(csio-cdb_io.cdb_ptr);
    } else {
        /* CDB指向是物理的 */
#ifdef __alpha__
        hcb-cmd = csio-cdb_io.cdb_ptr | alpha_XXX_dmamap_or ;
#else
        hcb-cmd = csio-cdb_io.cdb_ptr ;
#endif
    }
} else {
    /* CDB在ccb(缓冲区)中 */
    hcb-cmd = vtobus(csio-cdb_io.cdb_bytes);
}
hcb-cmdlen = csio-cdb_len;

```

现在是设置数据的的时候了，又一次，可以在CCB中以多种方式指定数据存贮，有些方式由CCB中的旗帜指定。首先我们得到数据的方向。最坏的情况是没有数据需要的情况：

```

int dir = (ccb_h-flags CAM_DIR_MASK);

if (dir == CAM_DIR_NONE)
    goto end_data;

```

然后我们数据在一个chunk中是在分散/集中列表中，并且是物理地址是虚地址。

SCSI控制器可能只能管理有限数目有限深度的大。如果请求到到个限制我就返回。我使用一个特殊函数返回CCB，并在一个地方管理HCB源短缺。加chunk的函数是程序相关的，此我不入它的。于地址翻的可以参看SCSI命令(CDB)的描述。如果某些体于特定的太困或不可能，返回状 **CAM_REQ_INVALID** 是可以的。上，在的CAM代中似乎儿也没有使用分散/集中能力。但至少必个非分散虚冲区的情况，CAM中情况用得很多。


```

int rv;

initialize_hcb_for_data(hcb);

if(!(ccb_h-flags CAM_SCATTER_VALID)) {
    /* 一个缓冲区 */
    if(!(ccb_h-flags CAM_DATA_PHYS)) {
        rv = add_virtual_chunk(hcb, csio-data_ptr, csio-dxfer_len, dir);
    }
    } else {
        rv = add_physical_chunk(hcb, csio-data_ptr, csio-dxfer_len,
dir);
    }
} else {
    int i;
    struct bus_dma_segment *segs;
    segs = (struct bus_dma_segment *)csio-data_ptr;

    if ((ccb_h-flags CAM_SG_LIST_PHYS) != 0) {
        /* SG列表指的是物理的 */
        rv = setup_hcb_for_physical_sg_list(hcb, segs, csio-sglist_cnt);
    } else if (!(ccb_h-flags CAM_DATA_PHYS)) {
        /* SG缓冲区指的是虚的 */
        for (i = 0; i < csio-sglist_cnt; i++) {
            rv = add_virtual_chunk(hcb, segs[i].ds_addr,
                segs[i].ds_len, dir);
            if (rv != CAM_REQ_CMP)
                break;
        }
    } else {
        /* SG缓冲区指的是物理的 */
        for (i = 0; i < csio-sglist_cnt; i++) {
            rv = add_physical_chunk(hcb, segs[i].ds_addr,
                segs[i].ds_len, dir);
            if (rv != CAM_REQ_CMP)
                break;
        }
    }
}
if(rv != CAM_REQ_CMP) {
    /* 如果成功添加了一chunk, 我希望add_*_chunk()函数返回
    * CAM_REQ_CMP, 如果请求太大(太多字或太多chunks)
    * 返回CAM_REQ_TOO_BIG, 其他情况下返回CAM_REQ_INVALID。
    */
    free_hcb_and_ccb_done(hcb, ccb, rv);
    return;
}
end_data:

```

如果一个CCB不允许断接, 我就一个信息到hcb:

```
if(ccb_h->flags & CAM_DIS_DISCONNECT)
    hcb->disable_disconnect(hcb);
```

如果控制器能完全自己行REQUEST SENSE命令，也应当将旗CAM_DIS_AUTOSENSE的它，可以在CAM子系不想REQUEST SENSE阻止自REQUEST SENSE。

剩下的唯一事情是置超，将我hcb硬件并返回，余下的由中断理函数（或超理函数）完成。

```
ccb_h->timeout_ch = timeout(xxx_timeout, (caddr_t) hcb,
    (ccb_h->timeout * hz) / 1000); /* 将毫秒滴答数 */
put_hcb_into_hardware_queue(hcb);
return;
```

儿是返回CCB的函数的一个可能：

```
static void
free_hcb_and_ccb_done(struct xxx_hcb *hcb, union ccb *ccb, u_int32_t
status)
{
    struct xxx_softc *softc = hcb->softc;

    ccb->ccb_h->ccb_hcb = 0;
    if(hcb != NULL) {
        untimeout(xxx_timeout, (caddr_t) hcb, ccb->ccb_h->timeout_ch);
        /* 我要放hcb, 因此源短缺也就不存在了 */
        if(softc->flags & RESOURCE_SHORTAGE) {
            softc->flags = ~RESOURCE_SHORTAGE;
            status |= CAM_RELEASE_SIMQ;
        }
        free_hcb(hcb); /* 同从任何内部列表中移除hcb */
    }
    ccb->ccb_h->status = status |
        (ccb->ccb_h->status & ~(CAM_STATUS_MASK|CAM_SIM_QUEUED));
    xpt_done(ccb);
}
```

- **XPT_RESET_DEV** - 送SCSI "BUS DEVICE RESET"消息到

除了部外CCB中没有数据，其中最人感兴趣的参量target_id。依于控制器硬件，硬件控制就像XPT SCSI_IO求中那被建（参看XPT SCSI_IO求的描述）并被送到控制器，或者立即程SCSI控制器送RESET消息到，或者个求可能只是不被支持（并返回状CAM_REQ_INVALID）。而且求完成，目的所有已断接(disconnected)的事必被中止（可能在中断例程中）。

而且目的所有当前商在位会失，因此它也可能被清除。或者清除可能被延，因不管目将会在下次事求重新商。

- *XPT_RESET_BUS* - 发送RESET信号到SCSI

CCB中并不包含参量，唯一感兴趣的参量是由指向sim的指向的SCSI。

最小会忘记上所有SCSI商，并返回状态 *CAM_REQ_CMP*。

恰当的会上会外位SCSI（可能也位SCSI控制器）并 将所有在硬件列中的和断接的那些正被
理的CCB的完成状态 *CAM SCSI_BUS_RESET*。像：

```

int targ, lun;
struct xxx_hcb *h, *hh;
struct ccb_trans_settings neg;
struct cam_path *path;

/* SCSI位可能会花很，情况下当使用中断或超来
 * 位是否完成。但了，我儿假位很快。
 */
reset_scsi_bus(softc);

/* 所有入的CCB */
for(h = softc-first_queued_hcb; h != NULL; h = hh) {
    hh = h-next;
    free_hcb_and_ccb_done(h, h-ccb, CAM_SCSI_BUS_RESET);
}

/* 商的（清除操作后的）干，我告个 */
neg.bus_width = 8;
neg.sync_period = neg.sync_offset = 0;
neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
    | CCB_TRANS_SYNC_RATE_VALID | CCB_TRANS_SYNC_OFFSET_VALID);

/* 所有断接的CCB和干的商（注：干=clean） */
for(targ=0; targ = OUR_MAX_SUPPORTED_TARGET; targ++) {
    clean_negotiations(softc, targ);

    /* 如果可能告事件 */
    if(xpt_create_path(path, /*periph*/NULL,
        cam_sim_path(sim), targ,
        CAM_LUN_WILDCARD) == CAM_REQ_CMP) {
        xpt_async(AC_TRANSFER_NEG, path, neg);
        xpt_free_path(path);
    }

    for(lun=0; lun = OUR_MAX_SUPPORTED_LUN; lun++)
        for(h = softc-first_discon_hcb[targ][lun]; h != NULL; h = hh) {
            hh=h-next;
            free_hcb_and_ccb_done(h, h-ccb, CAM_SCSI_BUS_RESET);
        }
}

ccb-ccb_h.status = CAM_REQ_CMP;
xpt_done(ccb);

/* 告事件 */
xpt_async(AC_BUS_RESET, softc-wpath, NULL);
return;

```

将SCSI位作函数来可能是个好主意，因如果事情出了差，
告来重用。

它会被超函数作最后的

- *XPT_ABORT* - 中止指定的CCB

参量在[]合ccb的[]例"struct ccb_abort cab" 中[]。其中唯一的参量字段[]：

abort_ccb - 指向被中止的ccb的指[]

如果不支持中断就返回CAM_UA_ABORT。[]也是最小化[]个[]用的[]易方式，任何情况下都返回CAM_UA_ABORT。

困[]方式[]是真正地[]个[]求。首先[][]用到SCSI事[]的中止：

```
struct ccb *abort_ccb;
abort_ccb = ccb-cab.abort_ccb;

if(abort_ccb-ccb_h.func_code != XPT SCSI_IO) {
    ccb-ccb_h.status = CAM_UA_ABORT;
    xpt_done(ccb);
    return;
}
```

然后需要在我[]的[]列中[]到[]个CCB。[]可以通[]遍[]我[]所有硬件[]来完成：

控制[]列表，[]与[]个CCB[]的控制

```
struct xxx_hcb *hcb, *h;

hcb = NULL;

/* 我[]假[]softc-first_hcb是与此[]的所有HCB的列表[]元素,
 * 包括那些入[]待[]理的、硬件正在[]理的和断[]接的那些。
 */
for(h = softc-first_hcb; h != NULL; h = h-next) {
    if(h-ccb == abort_ccb) {
        hcb = h;
        break;
    }
}

if(hcb == NULL) {
    /* 我[]的[]列中没有[]的CCB */
    ccb-ccb_h.status = CAM_PATH_INVALID;
    xpt_done(ccb);
    return;
}

hcb=found_hcb;
```

[]在我[]来看一下HCB当前的[]理状[]。它可能或呆在[]列中正等待[]被[]送到SCSI[]，或此[]正在[]中，或已断[]接并等待命令[]果，[]或者[]上已由硬件完成但尚未被[]件[]完成。[]了[]保我[]不会与硬件[]生[]争条件，我[]将HCB[]中止(aborted)，[]如果[]个HCB要被[]送到SCSI[]的[]，

SCSI控制器将会看到一个旗幟并跳過它。

```
int hstatus;

/* 此函数是一个函数，有它需要特殊操作才能使得一个旗幟硬件可
 */
set_hcb_flags(hcb, HCB_BEING_ABORTED);

abort_again:

hstatus = get_hcb_status(hcb);
switch(hstatus) {
case HCB_SITTING_IN_QUEUE:
    remove_hcb_from_hardware_queue(hcb);
    /* 此行 */
case HCB_COMPLETED:
    /* 这是一般的情况 */
    free_hcb_and_ccb_done(hcb, abort_ccb, CAM_REQ_ABORTED);
    break;
```

如果CCB此正在队列中，我一般会以某种硬件相关的方式向信号 SCSI控制器，通知它我希望中止当前的队列。SCSI控制器会置 SCSI ATTENTION信号，并当目前其行完后发送ABORT消息。我也位超，以保它不会永睡眠。如果命令不能在某个合理的时间，如 10秒内中止，超例程就会行并位整个SCSI队列。由于命令会在某个合理的时间后被中止，因此我可以在只将中止请求返回，当作成功完成，并将被中止的CCB队列中止（但它没有将它完成）。

```
case HCB_BEING_TRANSFERRED:
    untimeout(xxx_timeout, (caddr_t) hcb, abort_ccb-ccb_h.timeout_ch);
    abort_ccb-ccb_h.timeout_ch =
        timeout(xxx_timeout, (caddr_t) hcb, 10 * hz);
    abort_ccb-ccb_h.status = CAM_REQ_ABORTED;
    /* 要求控制器中止CCB，然后产生一个中断并停止
     */
    if(signal_hardware_to_abort_hcb_and_stop(hcb) 0) {
        /* 呀，我没有得与硬件的争条件，在我中止
         * 个事之前它就脱队，再队一次
         * （注：脱队=getoff）*/
        goto abort_again;
    }

    break;
```

如果CCB位于断开的列表中，将它置中止请求，并在硬件队列的前端将它重新入队。位超，并告中止请求完成。

```

case HCB_DISCONNECTED:
    untimeout(xxx_timeout, (caddr_t) hcb, abort_ccb-ccb_h.timeout_ch);
    abort_ccb-ccb_h.timeout_ch =
        timeout(xxx_timeout, (caddr_t) hcb, 10 * hz);
    put_abort_message_into_hcb(hcb);
    put_hcb_at_the_front_of_hardware_queue(hcb);
    break;
}
ccb-ccb_h.status = CAM_REQ_CMP;
xpt_done(ccb);
return;

```

就是于ABORT请求的全部，尽管有一个。由于ABORT消息清除LUN上所有正在行中的事，我必将LUN上所有其他活事中止。那当在中断例程中完成，且在中止事之后。

将CCB中止作函数来可能是个很好的主意，因如果I/O事超一个函数能被重用。唯一的不同是超事将超请求返回状CAM_CMD_TIMEOUT。于是XPT_ABORT的case句就会很小，像下面：

```

case XPT_ABORT:
    struct ccb *abort_ccb;
    abort_ccb = ccb-cab.abort_ccb;

    if(abort_ccb-ccb_h.func_code != XPT SCSI_IO) {
        ccb-ccb_h.status = CAM-UA_ABORT;
        xpt_done(ccb);
        return;
    }
    if(xxx_abort_ccb(abort_ccb, CAM_REQ_ABORTED) 0)
        /* no such CCB in our queue */
        ccb-ccb_h.status = CAM_PATH_INVALID;
    else
        ccb-ccb_h.status = CAM_REQ_CMP;
    xpt_done(ccb);
    return;

```

- **XPT_SET_TRAN_SETTINGS** - 式置SCSI置的

在合ccb的例"struct ccb_trans_setting cts" 中的参量：

- *valid* - 位掩，示当更新那些置：
- *CCB_TRANS_SYNC_RATE_VALID* - 同速率
- *CCB_TRANS_SYNC_OFFSET_VALID* - 同位移
- *CCB_TRANS_BUS_WIDTH_VALID* - 度
- *CCB_TRANS_DISC_VALID* - 置用/禁用断接
- *CCB_TRANS_TQ_VALID* - 置用/禁用排的
- *flags* - 由部分成，元参量和子操作。元参量：

- `CCB_TRANS_DISC_ENB` - 用断接
- `CCB_TRANS_TAG_ENB` - 用排的排

。子操作：

- `CCB_TRANS_CURRENT_SETTINGS` - 改当前的商
- `CCB_TRANS_USER_SETTINGS` - 住希望的用
- `sync_period, sync_offset` - 自解的，如果`sync_offset==0`求同模式
- `bus_width` - ，以位（而不是字）



参考原文和源

支持商参数，用置和当前置。用置在SIM程序中 上用得不多，通常只是一片内存，供上存（并在以后恢）其于 参数的一些主。置用参数并不会致重新商速率。但当SCSI 控制器商，它必永不能置高于用参数的，因此它上是上限。

当前置，正如其名字所示，指当前的。改它意味着下一次 必重新商参数。又一次，些"new current settings" 并没有被假定制用于上，它只是用作商的起始。此外，它必受SCSI控制器的能力限制：例如，如果SCSI控制器有8位，而求要求置16位，在送前参数必被悄悄地截取8位。

一个需要注意的就是度和同个参数是目的而言的， 而断接和用个参数是lun而言的。

建的 是保持3商参数（度和同）：

- *user* - 用的一，如上
- *current* - 生效的那些
- *goal* - 通置"current"参数所求的那些

代看起来像：


```

struct ccb_trans_settings *cts;
int targ, lun;
int flags;

cts = ccb-cts;
targ = ccb_h-target_id;
lun = ccb_h-target_lun;
flags = cts-flags;
if(flags CCB_TRANS_USER_SETTINGS) {
    if(flags CCB_TRANS_SYNC_RATE_VALID)
        softc-user_sync_period[targ] = cts-sync_period;
    if(flags CCB_TRANS_SYNC_OFFSET_VALID)
        softc-user_sync_offset[targ] = cts-sync_offset;
    if(flags CCB_TRANS_BUS_WIDTH_VALID)
        softc-user_bus_width[targ] = cts-bus_width;

    if(flags CCB_TRANS_DISC_VALID) {
        softc-user_tflags[targ][lun] = ~CCB_TRANS_DISC_ENB;
        softc-user_tflags[targ][lun] |= flags CCB_TRANS_DISC_ENB;
    }
    if(flags CCB_TRANS_TQ_VALID) {
        softc-user_tflags[targ][lun] = ~CCB_TRANS_TQ_ENB;
        softc-user_tflags[targ][lun] |= flags CCB_TRANS_TQ_ENB;
    }
}
if(flags CCB_TRANS_CURRENT_SETTINGS) {
    if(flags CCB_TRANS_SYNC_RATE_VALID)
        softc-goal_sync_period[targ] =
            max(cts-sync_period, OUR_MIN_SUPPORTED_PERIOD);
    if(flags CCB_TRANS_SYNC_OFFSET_VALID)
        softc-goal_sync_offset[targ] =
            min(cts-sync_offset, OUR_MAX_SUPPORTED_OFFSET);
    if(flags CCB_TRANS_BUS_WIDTH_VALID)
        softc-goal_bus_width[targ] = min(cts-bus_width, OUR_BUS_WIDTH);

    if(flags CCB_TRANS_DISC_VALID) {
        softc-current_tflags[targ][lun] = ~CCB_TRANS_DISC_ENB;
        softc-current_tflags[targ][lun] |= flags CCB_TRANS_DISC_ENB;
    }
    if(flags CCB_TRANS_TQ_VALID) {
        softc-current_tflags[targ][lun] = ~CCB_TRANS_TQ_ENB;
        softc-current_tflags[targ][lun] |= flags CCB_TRANS_TQ_ENB;
    }
}
ccb-ccb_h.status = CAM_REQ_CMP;
xpt_done(ccb);
return;

```

此后当下一次要处理I/O请求，它会检查是否需要重新协商，
target_negotiated(hcb)。它可以如下：

例如通用函数

```

int
target_negotiated(struct xxx_hcb *hcb)
{
    struct softc *softc = hcb-softc;
    int targ = hcb-targ;

    if( softc-current_sync_period[targ] != softc-goal_sync_period[targ]
    || softc-current_sync_offset[targ] != softc-goal_sync_offset[targ]
    || softc-current_bus_width[targ] != softc-goal_bus_width[targ] )
        return 0; /* FALSE */
    else
        return 1; /* TRUE */
}

```

重新协商些后，如果必须同当前和目的(goal)参数，于以后的I/O事
当前和目的参数将相同，且 **target_negotiated()** 会返回TRUE。当初始化（在
xxx_attach()中）当前商必被初始化最窄同模式，目的和当前必被初始化
控制器所支持的最大。（注：原文可能有，此未改）

- **XPT_GET_TRAN_SETTINGS** - 得SCSI置的

此操作XPT_SET_TRAN_SETTINGS的逆操作。用通旗 CCB_TRANS_CURRENT_SETTINGS
或CCB_TRANS_USER_SETTINGS（如果同置 有程序返回当前置）所求而得的数据填充CCB例
"struct ccb_trans_setting cts". *

XPT_CALC_GEOMETRY - 算磁的（BIOS）(geometry)

参量在合ccb的例"struct ccb_calc_geometry ccg" 中：

- *block_size* - 入，以字的大小（也称扇区）
- *volume_size* - 入，以字的卷大小
- *cylinders* - 出，柱面
- *heads* - 出，磁
- *secs_per_track* - 出，磁道的扇区

如果返回的与SCSI控制器BIOS所想象的差很大，并且SCSI 控制器上的磁被作可引的，系
可能无法。从aic7xxx 程序中摘取的典型算示例：

```

struct    ccb_calc_geometry *ccg;
u_int32_t size_mb;
u_int32_t secs_per_cylinder;
int    extended;

ccg = ccb->ccg;
size_mb = ccg->volume_size
    / ((1024L * 1024L) / ccg->block_size);
extended = check_cards_EEPROM_for_extended_geometry(softc);

if (size_mb > 1024 & extended) {
    ccg->heads = 255;
    ccg->secs_per_track = 63;
} else {
    ccg->heads = 64;
    ccg->secs_per_track = 32;
}
secs_per_cylinder = ccg->heads * ccg->secs_per_track;
ccg->cylinders = ccg->volume_size / secs_per_cylinder;
ccb->ccb_h.status = CAM_REQ_CMP;
xpt_done(ccb);
return;

```

想出了一般思路，精确计算依赖于特定BIOS的癖好(quirk)。如果 BIOS没有提供方法设置EEPROM中的"extended translation" 旗，此旗通常当假定等于1。其他流行有：

128 heads, 63 sectors - Symbios控制器
 16 heads, 63 sectors - 老式控制器

一些系BIOS和SCSI BIOS会相互竞争，不定，例如Symbios 875/895 SCSI和Phoenix BIOS的合在系加会出128/63，而当冷或后会255/63。

- **XPT_PATH_INQ** - 路径，句，得SIM程序和SCSI控制器（也称HBA - 主机适配器）的特性。

特性在合ccb的例"struct ccb_pathinq cpi" 中返回：

- **version_num** - SIM程序号，当前所有程序使用1
- **hba_inquiry** - 控制器所支持特性的位掩：
- **PI_MDP_ABLE** - 支持MDP消息（来自SCSI3的一些西?）
- **PI_WIDE_32** - 支持32位SCSI
- **PI_WIDE_16** - 支持16位SCSI
- **PI_SDTR_ABLE** - 可以商同速率
- **PI_LINKED_CDB** - 支持接的命令
- **PI_TAG_ABLE** - 支持的命令
- **PI_SOFT_RST** - 支持位（硬位和位在SCSI中是互斥的）

- target_sprt - 目标模式支持的旗，如果不支持00
- hba_misc - 控制器特性：
- PIM_SCANHILO - 从高ID到低ID的扫描
- PIM_NOREMOVE - 可移除不包括在扫描之列
- PIM_NOINITIATOR - 不支持发起者角色
- PIM_NOBUSRESET - 用禁用初始BUS RESET
- hba_eng_cnt - 神秘的HBA引导数，与有的一些西，当前是置0
- vuhba_flags - 供商唯一的旗，当前未用
- max_target - 最大支持的目标ID（8位0007，16位00015，光通道0127）
- max_lun - 最大支持的LUN ID（老的SCSI控制器07，新的063）
- async_flags - 安装的回调函数的位掩，当前未用
- hpath_id - 子系中最高的路径ID，当前未用
- unit_number - 控制器元号，cam_sim_unit(sim)
- bus_id - 号，cam_sim_bus(sim)
- initiator_id - 控制器自己的SCSI ID
- base_transfer_speed - 窄的名速率，以KB/s，于SCSI等于3300
- sim_vid - SIM程序的供商ID，以0束的字符串，包含尾0在内的最大度SIM_IDLEN
- hba_vid - SCSI控制器的供商ID，以0束的字符串，包含尾0在内的最大度HBA_IDLEN
- dev_name - 程序名字，以0束的字符串，包含尾0在内的最大度DEV_IDLEN，等于cam_sim_name(sim)

置字符串字段的建方法是使用strncpy，如：

```
strncpy(cpi-dev_name, cam_sim_name(sim), DEV_IDLEN);
```

置些后将状置CAM_REQ_CMP，并将CCB完成。

11.3. 中断

```
static void xxx_poll ( struct cam_sim *sim);
struct cam_sim *sim ;
```

函数用于当中断子系不起作用（例如，系崩或正在建系）模中断。CAM子系在用函数前置当的中断。因此它所需做全部的只是用中断例程（或其他方法，例程来行作，而中断例程只是用例程）。那什要嘛弄出一个独的函数来？是由于不同的用定。
xxx_poll例程取cam_sim的指作参量，而PCI中断例程按照普通定取的是指向xxx_softc的指，ISA中断例程只是取号，因此例程一般看起来像：

```
static void
xxx_poll(struct cam_sim *sim)
{
    xxx_intr((struct xxx_softc *)cam_sim_softc(sim)); /* for PCI device */
}
```

or

```
static void
xxx_poll(struct cam_sim *sim)
{
    xxx_intr(cam_sim_unit(sim)); /* for ISA device */
}
```

11.4. 事件

如果建立了事件回调，回调当定回调函数。

```
static void
ahc_async(void *callback_arg, u_int32_t code, struct cam_path *path, void *arg)
```

- callback_arg - 注册回调提供的
- code - 事件型
- path - 事件作用于其上的
- arg - 事件特定的参量

一型事件，AC_LOST_DEVICE，看起来如下：

```

struct xxx_softc *softc;
struct cam_sim *sim;
int targ;
struct ccb_trans_settings neg;

sim = (struct cam_sim *)callback_arg;
softc = (struct xxx_softc *)cam_sim_softc(sim);
switch (code) {
case AC_LOST_DEVICE:
    targ = xpt_path_target_id(path);
    if(targ == OUR_MAX_SUPPORTED_TARGET) {
        clean_negotiations(softc, targ);
        /* send indication to CAM */
        neg.bus_width = 8;
        neg.sync_period = neg.sync_offset = 0;
        neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
                    | CCB_TRANS_SYNC_RATE_VALID | CCB_TRANS_SYNC_OFFSET_VALID);
        xpt_async(AC_TRANSFER_NEG, path, neg);
    }
    break;
default:
    break;
}

```

11.5. 中断

中断例程的切切型依赖于SCSI控制器所接到的外部的型（PCI，ISA等等）。

SIM程序的中断例程行在中断splcam上。因此当在程序中使用时使用splcam()来同中断例程与程序剩余部分的活（由于能察多处理器的程序，事情更有趣，但此我忽略情况）。本文中的代码地忽略了同。代码一定不能忽略它。一个的方法是入其他例程的入口点splcam()，并在返回将它位，从而用一个大的界区保它。了保中断是会被恢，可以定一个包装函数，如：

```

static void
xxx_action(struct cam_sim *sim, union ccb *ccb)
{
    int s;
    s = splcam();
    xxx_action1(sim, ccb);
    splx(s);
}

static void
xxx_action1(struct cam_sim *sim, union ccb *ccb)
{
    ... process the request ...
}

```

方法而且健壮，但它存在的缺点是中断可能会被阻塞相当短的事件，会系性能生面影。一方面，`spl()`函数族有相当高的开销，因此大量很小的临界区可能也不好。

中断例程处理的情况和其中重重依赖于硬件。我考"典型(typical)"情况。

首先，我上是否遇到了SCSI位（可能由同一SCSI上的SCSI控制器引起）。如果我所有入的和断接的求，告事件并重新初始化我的SCSI控制器。初始化期控制器不会出一个位，我十分重要，否同一SCSI上的个控制器可能会一直来回地位下去。控制器致命挂起的情况可以在同一地方行理，但可能需要送RESET信号到SCSI来位与SCSI的接状。

```
int fatal=0;
struct ccb_trans_settings neg;
struct cam_path *path;

if( detected_scsi_reset(softc)
|| (fatal = detected_fatal_controller_error(softc)) ) {
    int targ, lun;
    struct xxx_hcb *h, *hh;

    /* 所有入的CCB */
    for(h = softc-first_queued_hcb; h != NULL; h = hh) {
        hh = h-next;
        free_hcb_and_ccb_done(h, h-ccb, CAM_SCSI_BUS_RESET);
    }

    /* 要告的商的干 */
    neg.bus_width = 8;
    neg.sync_period = neg.sync_offset = 0;
    neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
        | CCB_TRANS_SYNC_RATE_VALID | CCB_TRANS_SYNC_OFFSET_VALID);

    /* 所有断接的CCB和干商 */
    for(targ=0; targ = OUR_MAX_SUPPORTED_TARGET; targ++) {
        clean_negotiations(softc, targ);

        /* report the event if possible */
        if(xpt_create_path(path, /*periph*/NULL,
            cam_sim_path(sim), targ,
            CAM_LUN_WILDCARD) == CAM_REQ_CMP) {
            xpt_async(AC_TRANSFER_NEG, path, neg);
            xpt_free_path(path);
        }

        for(lun=0; lun = OUR_MAX_SUPPORTED_LUN; lun++)
            for(h = softc-first_discon_hcb[targ][lun]; h != NULL; h = hh) {
                hh=h-next;
                if(fatal)
                    free_hcb_and_ccb_done(h, h-ccb, CAM_UNREC_HBA_ERROR);
                else
                    free_hcb_and_ccb_done(h, h-ccb, CAM_SCSI_BUS_RESET);
            }
    }
}
```

```

    }
}

/* 报告事件 */
xpt_async(AC_BUS_RESET, softc-wpath, NULL);

/* 重新初始化可能花很多时间，在某些情况下当由一中断信号
 * 指示初始化否完成，或在超时 - 但了我假设
 * 初始化真的很快
 */
if(!fatal) {
    reinitialize_controller_without_scsi_reset(softc);
} else {
    reinitialize_controller_with_scsi_reset(softc);
}
schedule_next_hcb(softc);
return;
}

```

如果中断不是由控制器自己的条件引起的，很可能当前硬件控制输出。依赖于硬件，可能有非HCB相关的事件，此我指示不考虑它。然后我分析一个HCB生了什么：

```

struct xxx_hcb *hcb, *h, *hh;
int hcb_status, scsi_status;
int ccb_status;
int targ;
int lun_to_freeze;

hcb = get_current_hcb(softc);
if(hcb == NULL) {
    /* 或者失(stray)的中断，或者某些西重，
     * 或者是硬件相关的某些西
     */
    行必要的理;
    return;
}

targ = hcb-target;
hcb_status = get_status_of_current_hcb(softc);

```

首先我看看HCB是否完成，如果完成我就返回的SCSI状态。

```

if(hcb_status == COMPLETED) {
    scsi_status = get_completion_status(hcb);
}

```

然后看一个状态是否与REQUEST SENSE命令有关，如果有适当地理一下它。


```

if(hcb-flags DOING_AUTOSENSE) {
    if(scsi_status == GOOD) { /* autosense成功 */
        hcb-ccb-ccb_h.status |= CAM_AUTOSNS_VALID;
        free_hcb_and_ccb_done(hcb, hcb-ccb, CAM_SCSI_STATUS_ERROR);
    } else {
autosense_failed:
        free_hcb_and_ccb_done(hcb, hcb-ccb, CAM_AUTOSENSE_FAIL);
    }
    schedule_next_hcb(softe);
    return;
}

```

否命令自身已完成，把更多注意力放在上。如果个CCB 没有禁用auto-sense并且命令同sense数据失，行REQUEST SENSE 命令接收那些数据。

```

hcb-ccb-csio.scsi_status = scsi_status;
calculate_residue(hcb);

if( (hcb-ccb-ccb_h.flags CAM_DIS_AUTOSENSE)==0
    ( scsi_status == CHECK_CONDITION
      || scsi_status == COMMAND_TERMINATED) ) {
    /* auto-SENSE */
    hcb-flags |= DOING_AUTOSENSE;
    setup_autosense_command_in_hcb(hcb);
    restart_current_hcb(softe);
    return;
}
if(scsi_status == GOOD)
    free_hcb_and_ccb_done(hcb, hcb-ccb, CAM_REQ_CMP);
else
    free_hcb_and_ccb_done(hcb, hcb-ccb, CAM_SCSI_STATUS_ERROR);
schedule_next_hcb(softe);
return;
}

```

属于商事件的一个典型事情：从SCSI目（回答我的商企或由目起的）接收到的商消息，或目无法商（拒我的商消息或不回答它）。

```

switch(hcb_status) {
case TARGET_REJECTED_WIDE_NEG:
    /* 恢到8-bit */
    softe-current_bus_width[targ] = softe-goal_bus_width[targ] = 8;
    /* 告事件 */
    neg.bus_width = 8;
    neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
    xpt_async(AC_TRANSFER_NEG, hcb-ccb-ccb_h.path_id, neg);
    continue_current_hcb(softe);
    return;
}

```

```

case TARGET_ANSWERED_WIDE_NEG:
{
    int wd;

    wd = get_target_bus_width_request(softec);
    if(wd == softec-goal_bus_width[targ]) {
        /* 可接受的答案 */
        softec-current_bus_width[targ] =
        softec-goal_bus_width[targ] = neg.bus_width = wd;

        /* 报告事件 */
        neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
        xpt_async(AC_TRANSFER_NEG, hcb-ccb.ccb_h.path_id, neg);
    } else {
        prepare_reject_message(hcb);
    }
}
continue_current_hcb(softec);
return;
case TARGET_REQUESTED_WIDE_NEG:
{
    int wd;

    wd = get_target_bus_width_request(softec);
    wd = min (wd, OUR_BUS_WIDTH);
    wd = min (wd, softec-user_bus_width[targ]);

    if(wd != softec-current_bus_width[targ]) {
        /* 宽度改变了 */
        softec-current_bus_width[targ] =
        softec-goal_bus_width[targ] = neg.bus_width = wd;

        /* 报告事件 */
        neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
        xpt_async(AC_TRANSFER_NEG, hcb-ccb.ccb_h.path_id, neg);
    }
    prepare_width_nego_response(hcb, wd);
}
continue_current_hcb(softec);
return;
}

```

然后我用与前面相同的办法处理auto-sense期间可能出现的任何错误。否则，我再一次输入。

```

if(hcb-flags DOING_AUTOSENSE)
    goto autosense_failed;

switch(hcb_status) {

```

我考虑的下一事件是未期的中断，一个事件在ABORT或RESET消息之后被看作是正常的，其他情况下是非正常的。

BUS

DEVICE

```
case UNEXPECTED_DISCONNECT:
    if(requested_abort(hcb)) {
        /* 中止影响那个目和LUN上的所有命令，因此将那个目和LUN上的
         * 所有断接的HCB也中止
         */
        for(h = softc-first_discon_hcb[hcb-target][hcb-lun];
            h != NULL; h = hh) {
            hh=h-next;
            free_hcb_and_ccb_done(h, h-ccb, CAM_REQ_ABORTED);
        }
        ccb_status = CAM_REQ_ABORTED;
    } else if(requested_bus_device_reset(hcb)) {
        int lun;

        /* 位影响那个目上的所有命令，因此将那个目和LUN上的
         * 所有断接的HCB位
         */

        for(lun=0; lun = OUR_MAX_SUPPORTED_LUN; lun++)
            for(h = softc-first_discon_hcb[hcb-target][lun];
                h != NULL; h = hh) {
                hh=h-next;
                free_hcb_and_ccb_done(h, h-ccb, CAM_SCSI_BUS_RESET);
            }

        /* 送事件 */
        xpt_async(AC_SENT_BDR, hcb-ccb-ccb_h.path_id, NULL);

        /* 是CAM_RESET_DEV求本身，它完成了 */
        ccb_status = CAM_REQ_CMP;
    } else {
        calculate_residue(hcb);
        ccb_status = CAM_UNEXP_BUSFREE;
        /* request the further code to freeze the queue */
        hcb-ccb-ccb_h.status |= CAM_DEV_QFRZN;
        lun_to_freeze = hcb-lun;
    }
    break;
```

如果目拒绝接受，我就通知CAM，并返回此LUN的所有命令：

```

case TAGS_REJECTED:
    /* 报告事件 */
    neg.flags = 0 ~CCB_TRANS_TAG_ENB;
    neg.valid = CCB_TRANS_TQ_VALID;
    xpt_async(AC_TRANSFER_NEG, hcb-ccb.ccb_h.path_id, neg);

    ccb_status = CAM_MSG_REJECT_REC;
    /* 请求后面的代队列 */
    hcb-ccb-ccb_h.status |= CAM_DEV_QFRZN;
    lun_to_freeze = hcb-lun;
    break;

```

然后我一些其他情况，理(processing)基本上限于置CCB状：

```

case SELECTION_TIMEOUT:
    ccb_status = CAM_SEL_TIMEOUT;
    /* request the further code to freeze the queue */
    hcb-ccb-ccb_h.status |= CAM_DEV_QFRZN;
    lun_to_freeze = CAM_LUN_WILDCARD;
    break;
case PARITY_ERROR:
    ccb_status = CAM_UNCOR_PARITY;
    break;
case DATA_OVERRUN:
case ODD_WIDE_TRANSFER:
    ccb_status = CAM_DATA_RUN_ERR;
    break;
default:
    /*以通用方法理所有其他 */
    ccb_status = CAM_REQ_CMP_ERR;
    /* 请求后面的代队列 */
    hcb-ccb-ccb_h.status |= CAM_DEV_QFRZN;
    lun_to_freeze = CAM_LUN_WILDCARD;
    break;
}

```

然后我是否重到需要入列，直到它得到理方可解，如果是那就来理：

```

if(hcb-ccb-ccb_h.status CAM_DEV_QFRZN) {
    /* 队列 */
    xpt_freeze_devq(ccb-ccb_h.path, /*count*/1);

/* 重新入队个目标/LUN的所有命令，将它返回CAM */

    for(h = softc-first_queued_hcb; h != NULL; h = hh) {
        hh = h-next;

        if(targ == h-targ
           (lun_to_freeze == CAM_LUN_WILDCARD || lun_to_freeze == h-lun) )
            free_hcb_and_ccb_done(h, h-ccb, CAM_REQUEUE_REQ);
    }
}
free_hcb_and_ccb_done(hcb, hcb-ccb, ccb_status);
schedule_next_hcb(softc);
return;

```

包括通用中断处理，尽管特定处理器可能需要某些附加处理。

11.6. 出错

当行I/O请求很多事情可能出错。可以在CCB状态中非常详尽地报告原因。使用的例子散布于本文档中。为了完整起见此输出典型条件的建库的一个：

- *CAM_RESRC_UNAVAIL* - 某些资源不可用，并且当其可用时SIM程序不能产生事件。资源的一个例子就是某些控制器内部硬件资源，当其可用时控制器不会对其产生中断。
- *CAM_UNCOR_PARITY* - 产生不可恢复的奇偶校验错误
- *CAM_DATA_RUN_ERR* - 数据外溢或未预期的数据状态(phase) (在一个方向上而不是CAM_DIR_MASK指定的方向)，或由于输出奇数数据度
- *CAM_SEL_TIMEOUT* - 产生超时 (目标不响)
- *CAM_CMD_TIMEOUT* - 生命命令超时 (超时函数行)
- *CAM_SCSI_STATUS_ERROR* - 返回的错误
- *CAM_AUTONSENSE_FAIL* - 返回的错误且REQUEST SENSE命令失败
- *CAM_MSG_REJECT_REC* - 收到MESSAGE REJECT消息
- *CAM_SCSI_BUS_RESET* - 收到SCSI复位
- *CAM_REQ_CMP_ERR* - 输出"不可能(impossible)"SCSI状态(phase) 或者其他怪异事情，或者如果一的信息不可用只是通用
- *CAM_UNEXP_BUSFREE* - 输出未预期的断接
- *CAM_BDR_SENT* - BUS DEVICE RESET消息被送到目标
- *CAM_UNREC_HBA_ERROR* - 不可恢复的主机适配器
- *CAM_REQ_TOO_BIG* - 请求于控制器太大

- *CAM_REQUEUE_REQ* - 此请求当被重新入列以保持事务的次序性。典型地出列在下列时刻：SIM出了列当列满时，并且必在sim上将其目的其他入列请求放回到XPT列。这些情况的典型情况有超、命令超和其他类似情况。些情况下出列的命令返回状态来指示，此命令和其他没有被送到列的命令被重新入列。
- *CAM_LUN_INVALID* - SCSI控制器不支持请求中的LUN ID
- *CAM_TID_INVALID* - SCSI控制器不支持请求中的目标ID

11.7. 超时处理

当HCB的超时期满，请求就当被中止，就像处理XPT_ABORT请求一样。唯一区别在于被中止的请求的返回状态当CAM_CMD_TIMEOUT而不是CAM_REQ_ABORTED（就是什么中止的最好由函数来完成）。但有一个可能的：如果中止请求自己出了麻烦？情况下当位SCSI，就像处理XPT_RESET_BUS请求一样（并且将其函数，从这个地方用的想法也用于这儿）。而且如果位请求出了，我当位整个SCSI。因此最超时函数看起来像下面样子：

```
static void
xxx_timeout(void *arg)
{
    struct xxx_hcb *hcb = (struct xxx_hcb *)arg;
    struct xxx_softc *softc;
    struct ccb_hdr *ccb_h;

    softc = hcb-softc;
    ccb_h = hcb-ccb-ccb_h;

    if(hcb-flags & HCB_BEING_ABORTED
    || ccb_h-func_code == XPT_RESET_DEV) {
        xxx_reset_bus(softc);
    } else {
        xxx_abort_ccb(hcb-ccb, CAM_CMD_TIMEOUT);
    }
}
```

当我中止一个请求，同一目标/LUN的所有其他断开的请求也会被中止。因此出了一个，我当返回它的状态CAM_REQ_ABORTED是CAM_CMD_TIMEOUT？当前的程序使用CAM_CMD_TIMEOUT。看起来符合，因为如果一个请求超时，可能出了某些的很糟的事情，因此如果它没有被乱它自己当超。

Chapter 12. USB

12.1. 介绍

通用串行总线(USB)是将设备接到个人计算机的一种新方法。它突出了双向通信的特色，并且其设计充分考虑到正逐渐智能化和需要与host进行更多交互的设备。USB的支持包含在当前所有芯片中，因此在新近制造的PC中都可用。如果(Apple)引入USB的iMac硬件，制造商生产其他USB版本的设备是一个很大的激励。未来的PC指定PC上的所有老接口当由一个或多个USB接口取代，提供通用的即用能力。USB硬件的支持在NetBSD的相当早期就有了，它是由 Lennart Augustsson在NetBSD项目做的。代码已被移植到FreeBSD上，我目前正写着一个底层共享代码。USB子系统的到来，许多USB的特性很重要。

Lennart Augustsson已经完成了NetBSD项目中USB支持的大部分工作。十分感谢他的工作量和他人的工作。也十分感谢Ardy和Dirk对本文稿的审阅和校对。

- 设备直接连接到计算机上的端口，或者连接到称为集中器的设备，形成星型拓扑。
- 设备可在运行中连接或断开。
- 设备可以挂起自身并触发host系统的重新投入运行。
- 由于设备可由主机供电，因此host系统必须跟踪每个集中器的电源计算。
- 不同设备型需要不同的服务量，并且同一设备可以连接最多126个设备，就需要恰当地度量的设备以充分利用12Mbps的可用带宽。(USB 2.0超过400Mbps)
- 设备智能化并包含很容易得到的关于自身的信息。

USB子系统以及连接到它的设备驱动程序受已经或将要有的设备的支持。有些设备可以从USB主设备公开得到。如果(Apple)通过使得通用设备程序可从其操作系统MacOS中得到，而且不鼓励新设备使用独特的程序来强烈推行基于标准的程序。本章将整理基本设备信息以便在FreeBSD/NetBSD中USB的当前设备有个基本的了解。然而，建议将下面参考中提及的相关内容与本站同读。

12.1.1. USB的架构

FreeBSD中的USB支持可被分为三层。最底层包含主控器，向硬件设备及其驱动程序提供一个通用接口。它支持硬件初始化，设备运行度，管理已完成/失败的传输。每个主控器驱动程序提供一个虚拟hub，以硬件无阻塞方式提供对控制机器背面根端口的寄存器的访问。

中间层管理设备连接和断开，设备的基本初始化，设备程序的运行，通信通道(管道)和电源管理。每个服务也控制默认管道和其上设备的请求。

上层包含支持特定(设备)设备的各个驱动程序。有些驱动程序除默认管道外的其他管道上使用的设备。他们也提供额外功能，使得设备内核或用空闲设备是可用的。他们使用服务暴露出的USB设备程序接口(USB DI)。

12.2. 主控器

主控器(HC)控制设备上包的传输。使用1毫秒的时间。在传输开始时，主控器产生一个起始(SOF, Start of Frame)包。

SOF包用于同步传输的开始和跟踪传输的数目。包在传输中被接收，或由host发送到设备(out)，或由设备到host(in)。

包是由host起（发起）。因此一条USB只能有一个host。每个包的帧都有一个状态段，数据接收者可以在其中返回ACK（应答接收），NAK（重传），STALL（阻塞条件）或什么也没有（混乱数据段，帧不可用或已断）。USB [USB specification](#)的第8.5更详细地解释了包的帧。USB帧上可以出四种不同帧型的包：控制(control)，大(bulk)，中断(interrupt)和同(isochronous)。帧的帧型和他的特性在下面描述（管道子中）。

USB上的帧和帧程序的大型帧被主控制器或HC帧程序分割成多个包。

到默认端点的帧请求（控制帧）有些特殊。它由一个或三个帧段组成：帧（SETUP），数据（DATA，可选）和状态（STATUS）。帧置(set-up)包被送到帧。如果存在数据帧段，数据包的方向在帧置包中给出。帧段中的方向与数据帧段期的方向相反，或者当没有数据帧段

帧IN。主控制器硬件也提供寄存器，用于保存根端口的当前帧和自帧改帧寄存器最后一次帧位以来所帧生的改帧。USB[2]建议使用一个虚hub来提供帧些寄存器的帧。虚hub必须符合第11章中帧出的hub帧。它必须提供一个默认管道使得帧请求可以帧送帧。它返回帧准和hub帧特定的一帧描述符。它也帧当提供一个中断管道用来帧告其帧端口帧生的帧化。当前可用的主控制器帧有帧个：[通用主控制器接口](#)（UHCI；英特尔）和[帧放主控制器接口](#)（OHCI；康柏，微帧，国家半帧体）。UHCI帧的帧通帧要求主控制器帧程序帧帧的帧提供完整的帧度，从而帧少了硬件帧性。OHCI帧型的控制器自身提供一个更抽象的接口来完成很多工作，从而更加独立。

12.2.1. UHCI

UHCI主控制器帧着帧有1024个指向帧数据帧帧的帧列表。它理解帧不同的数据帧型：帧描述符（TD）和帧列（QH）。帧个TD表示表示与帧端点帧行通信的一个包。QH是将一些TD（和QH）帧分成帧的一帧方法。

帧个帧由一个或多个包组成。UHCI帧程序将大的帧分割成帧多个包。除同帧外，帧个帧都会分配一个QH。帧于帧帧型的帧，都有一个与此帧型帧的QH，所有帧些QH都会被集中到帧个QH上。由于有固定的帧延需求，同帧帧必须首先帧行，它是通帧列表中的帧指帧直接引用的。最后的同帧TD帧引用那一帧的中断帧的QH。中断帧的所有QH指向控制帧的QH，控制帧的QH又指向大帧的QH。下面的帧表帧出了一个帧形概帧：

帧致下面的帧度会在帧中帧行。控制器从帧列表中获得当前帧的指帧后，首先帧那一帧中的所有的同帧(isochronous)包帧行TD。帧些TD的最后一个帧引用那一帧的中断帧的QH。然后主控制器将从那个QH下行到各个帧中断帧的QH。完成那一帧列后，中断帧的QH会将控制器指向到所有控制帧的QH。它将帧行在那儿等待帧度的所有子帧列，然后是在大帧QH中帧排帧的所有帧。帧了方便帧理已完成或失帧的帧，硬件会在帧末尾帧生不同帧型的中断。在帧的最后一个TD中，HC帧程序帧置Interrupt-On-Completion位来帧帧完成帧的一个中断。如果TD帧到了帧其最大帧数，就帧帧帧中断。如果在TD中帧置短包帧位，且帧了帧小于帧所帧置的包帧度（的包），就会帧帧此中断以通知控制器帧程序帧已完成。帧出帧个帧帧已完成或帧生帧是主控制器帧程序的任帧。当中断帧例程被帧用帧，它将定位所有已完成的帧并帧用帧的回帧。

更帧尽的描述帧看 [UHCI specification](#)。

12.2.2. OHCI

帧OHCI主控制器帧行帧程要容易得多。控制器帧有一帧端点(endpoint)可用，帧并知道帧中不同帧帧型的帧度帧先帧和排序。主控制器使用的主要帧数据帧是端点描述符（ED），它上面帧接着一个帧描述符（TD）的帧列。ED包含端点所帧的最大的包大小，控制器硬件完成包的分割。帧次帧帧后都会更新指向数据帧冲区的指帧，当起始和帧止指帧相等帧，TD就退帧到完成帧列(done-queue)。四帧帧型的端点各有其自己的帧列。控制和大(bulk)端点帧在帧自己的帧列帧。中断ED在帧中帧排帧，在帧中的深度帧定了帧帧行的帧度。

帧列表 中断 同(isochronous) 控制 大(bulk)

主控制器在00中0行的0度看起来如下。控制器首先0行非 周期性控制和大00列，最0可到HC00程序0置的一个00限制。 然后以00号低5位作0中断ED0上深度00的那一0中的索引，0行 那个00号的中断00。在0个0的末尾，同0ED被0接，并随后被 遍0。同0TD包含了000当0行其中的第一个0的00号。所有周期 性的000行0以后，控制和大00列再次被遍0。中断服0例程会被 周期性地0用，来0理完成的0列，00个000用回0，并重新0度 中断和同0端点。

更0尽的描述0看 [OHCI specification](#)。服00，即中00，提供了以可控的方式 0000行00，并00着由不同00程序和服00所使用的0源。此00理下面几方面：

- 00配置信息
- 与000行通信的管道
- 探0和0接00，以及从00分0(detach)。

12.3. USB00信息

12.3.1. 00配置信息

0个00提供了不同00的配置信息。0个00具有一个或多个 配置，探0/0接期0从其中0定一个。配置提供功率和00要求。 0个配置中可以有多多个接口。00接口是端点的0集(collection)。例如，USB00声器可以有一个音0接口（音00），和0旋0(knob)、 0号0(dial)和按0的接口（HID0）。一个配置中的所有接口可以同0有效，并可被不同的 00程序0接。0个接口可以有0用接口，以提供不同0量的服0参数。例如，在照相机中，0用来提供不同的0大小以及0秒0数。

0个接口中可以指定0或多个端点。端点是与000行通信的0向 00点。它0提供0冲区来00存0从00而来的，或外出到00的数据。 0个端点在配置中有唯一地址，即端点号加上其方向。默0端点，即 端点0，不是任何接口的一部分，并且在所有配置中可用。它由服00 管理，并且0000程序不能直接使用。

Level 0 Level 1 Level 2 Slot 0

Slot 3 Slot 2 Slot 1

(只0示了32个槽中的4个)

000次化配置信息在00中通00准的一0描述符来描述（参看 USB00[2]第9.60）。它0可以通0Get Descriptor Request来0求。 服000存0些描述符以避免在USB00上0行不必要的00。00些 描述符的00是通0函数0用来提供的。

- 00描述符：0于00的通用信息，如供0商，0品 和修0ID，支持的000、子0和0用的00，默0端点的最大包大小 等。
- 配置描述符：此配置中的接口数，支持的挂起和 恢0能力，以及功率要求。
- 接口描述符：接口0、子0和0用的00，接口0用 配置的数目和端点数目。
- 端点描述符：端点地址、方向和0型，支持的最大包 大小，如果是中断0型的端点00包括000率。默0端点（端点0）没有描述符，而且从不被0入接口描述符中。
- 字符串描述符：在其他描述符中会0某些字段提供 字符串索引。它0可被用来0取描述性字符串，可能以多00言 的形式提供。

00明(specification)可以添加它0自己的描述符0型，0些描述符 也可以通0GetDescriptor Request来0得。

管道与上端点的通信，流所经的管道。程序将到端点的 提交到管道，并提供（）失或完成用的回， 或等待完成（同）。到端点的在管道中被串行化。或者完成， 或者失， 或者超（如果置了超）。于有型的超。 超的生成可能由于USB上的超（毫秒）。些超被失，可能是由于断接引起的。一超在件中，当没有 在指定的（秒）内完成触。是由于的包否定答引起的。其原因是由于没有准备好接收数据，缓冲区欠或超，或。

如果管道上的大于的端点描述符中指定的最大包大小，主 控器（OHCI）或HC程序（UHCI）将按最大包大小分割，并且最后一个包可能小于最大包的大小。

有候来返回少于所求的数据并不是个。例如， 到制解器的大in可能求200字的数据，但制解器 那只有5个字可用。程序可以置短包（SPD）志。它允主 控器即使在的数据量少于所求的数据量的情况下也接受包。 个志只在in中有效，因将要被送到数据量是事先知道的。如果程中出不可恢的，管道会被停。 接受或送更多数据以前，程序需要定停的原因，并通在 默管道上送清除端点挂起求（clear endpoint halt device request）来清除端点停条件。

有四不同型的端点和的管道：

- 控制管道/默管道： 个有一个控制管道，接到默端点（端点0）。此管道 求和的数据。默管道和其他管道上的的区在于所 使用的，在USB[2]中描述。些求用于位和配置。 个必支持USB[2]的第9章中提供的一基本命令。管道上 支持的命令可以通展，以支持外的功能。
- 大（bulk）管道：是USB与原始媒体等的等价物。
- 中断管道：host向送数据求，如果没有 西送，将NAK（否定答）数据包。中断按建管道指定的 率被度。
- 同管道：些管道用于具有固定延的同数据，例如或音流，但不保一定。当前中已有型管道的某些支持。当期出，或者由于，例如缺乏缓冲区空 来存入的数据而引起的否定答包（NAK），控制、大和中断 管道中的包会被重。而同包在失或包NAK不会重，因 那可能反同束。

所需的可用在管道的建期被算。在1毫秒的 行度。中的分配由USB的第5.6定。同和中断被 允消耗中多90%的。控制和大 的包在所有同和中断包 之后行度，并将消耗所有剩余。

于度和回收的更多信息可以在USB[2]的第5章， UHCI[3]的第1.3，OHCI[4]的3.4.2中到。

12.4. 的探和接

集中器(hub)通知新已接后，服端口加（switch on）， 提供100mA的流。此 于其默状，并听地址0。服会通默 管道取各描述符。此后它将向送Set Address求，将 从默地址（地址0）移。可能多个程序支持此。例如， 一个制解器可能通AT兼容接口支持ISDN TA。然而，特定型号的ISDN 配器的程序可能提供此的更好支持。了支持 的活性， 探会返回先，指示他的支持。支持品的特定版本会具有最高 先，通用程序具有最低先。如果一个配置内有多个接口，也可能 多个程序会接到一个。个程序只需支持所有接口的一个子集。

新接的探程序，首先探特定的程序。 如果没有，探代在所有支持的配置上重探

程序，直到 在一个配置中它接到一个程序。它支持不同接口上使用多个 程序的，探索会在一个配置中的所有尚未被程序声明(claim)的 接口上重行。超出集中器功率预算的配置会被忽略。接期， 程序当把初始化为当状，但不能位，因那会使得将它自己从断，并重新探索程。为了避免消耗不必要的， 不当在接声明中断管道，而当延分配管道，直到打文件并真的使用数据。当文件，管道也当被再次，尽管可能仍然 接着。

12.4.1. 断接(disconnect)和分(detach)

程序与行任何事期， 当期会接收到。 USB的支持并鼓励在任何点及断接。程序当保 当不在做正的事情。

此外，断接(disconnect)后又重新接(reconnect)的 不会被重新接(reattach)相同的例。将来当更多的支持序列号（参看描述符）， 或出其他定 的方法的候， 情况可能会改。

断接是由集中器在到集中器程序的中断包中 信号通知(signal)的。状改信息指示个端口了接改。 接到那个端口上的的所有 程序共用的 分方法被用， 被底清理。如果端口状指示同个 已接(connect)到那个 端口， 探和接的程将被。 位将在集中器上生 一个断-接序列，并将按上面所述行理。

12.5. USB 程序的信息

USB没有定除默管道外其他管道上使用的。 方面的信息 可以从各来源得。最准的来源是USB主[1]上的 者部分。从些 面上可以得到数目不断的 的。 指定从 程序 角度看起来兼容 当， 它需要提供的基本功能和通信通道上使用的 。 USB[2]包括了集中器的描述。人机界面(HID)的 已 建出来，以迎合 、数字入板、条形器、按、旋(手柄knob)、 等的要求。 一个例子是用于大容量存 的。 的完整列表 参看USB主[1]的 者部分。

然而， 多 的信息 没有被公布。 于所用 的信息 可能可以从制造 的公司得。一些公司会在 之前要求 署 保密(Non-Disclosure Agreement, NDA)。大多数情况下， 会阻止 将程序放源代码。

一个信息的很好来源是Linux程序源代码，因 很多公司已 始 他的 提供Linux下的程序。 系那些程序作者 他的信息来源 是一个好主意。

例子：人机界面。人机界面，如、鼠、数字入板、 按、号等的 被其他 引用，并在很多 中使用。

例如，音 声器提供到数模 器的端点，可能 提供 外管道 用于麦克。它 也 前面的按和号 在 独的接口中提供HID 端点。 器控制 也是如此。通 可用的内核和用空 的，与HID 程序或通用 程序一起可以 直接地 建些接口的支持。 一个 可以作 在一个配置中的多个接口由不同的 程序 的例子， 个 是一 便宜的， 有老的鼠 接口。 为了避免在 中USB集中器包括一个硬件而 致的成本上升，制造商将从 背面的 PS/2端口接收到的鼠 数据与来自 的按 合成在同一个配置中的 个 独的接口。鼠 和 程序各自 接到 当的接口，并分配到 个独立端点的管道。

例子：固件下。已 出来的 多是基于通用目的 理器， 并将 外的USB核心加入其中。由于 程序的 和USB 的固件仍然 非常新， 多 需要在接(connect)之后下 固件。

下面的 非常 明直接。 通 供 商和 品ID 自身。第一 个 程序探 并 接到它，并将固件下 到其中。此后 自己 位， 程序分。短的 停之后 宣布它在 上的存在。 将改 其供 商/ 品

/版本的ID以反映其提供有固件的事，因此一个程序 将探测它并连接(attach)到它。

有些型的一个例子是基于EZ-USB的ActiveWire I/O板。一个芯片有一个通用固件下器。下到ActiveWire板子上的固件改版本ID。然后它将行EZ-USB芯片的USB部分的位，从USB上断开，并再次重新连接。

例子：大容量存储器。大容量存储器的支持主要有的 建。Iomega USB Zip器是基于SCSI版本的器。SCSI命令和 状态信息被包装到中，在大(bulk)管道上到/来自，在USB上模拟SCSI控制器。ATAPI和UFI命令以相似的方式被支持。

大容量存储器支持不同型的命令的包装。最初的 基于通默认管道送命令和状态信息，使用大在host和之间 移动数据。在基础上 出 方法， 方法基于包装命令和 状态，并在大out和in端点上 送它。 精确地指定了何必须 生什，以及在 到 条件的情况下 做什。 些 写 程序的最大挑是 基于USB的， 它合已有的大容量存储器 的支持。CAM提供了子，以相当直接了当的方式来完成 个。ATAPI就 没有 了，因 史上IDE接口从未有 多 不同的表 方式。

来自Y-E Data的USB的支持也不是那直，因 了一套 新的命令集。

Chapter 13. Newbus

特感Matthew N. Dodd, Warner Losh, Bill Paul, Doug Rabson, Mike Smith, Peter Wemm and Scott Long.

本章了解Newbus框架。

13.1. 设备程序

13.1.1. 设备程序的目的

设备程序是硬件，它在内核与外部（例如，磁盘、网络适配器）的通用接口和外部设备的接口之间提供了接口。设备程序接口(DDI)是内核与设备程序硬件之间定义的接口。

13.1.2. 设备程序的类型

在UNIX®那个时代，FreeBSD也从中延伸而来，定义了四种类型的设备：

- 设备程序
- 字符设备程序
- 网络设备程序
- 设备程序

设备可以使用固定大小的[数据]的方式运行。设备类型的设备程序依赖于缓冲缓存(buffer cache)，其目的是在内存中的缓冲区存储设备的数据。缓冲缓存常常基于后台写(write-behind)，这意味着数据在内存中被修改后，当系统行其周期性磁盘刷新时才会被同步到磁盘，从而优化写操作。

13.1.3. 字符设备

然而，在FreeBSD 4.0版本以及后续版本中，设备和字符设备的区别得不存在了。

13.2. Newbus概述

Newbus提供了一种基于抽象的新型设备，可以在FreeBSD 3.0中看到设备接口的介绍，当Alpha的移植被引入到代中。直到4.0它才成为设备程序使用的默认系统。其目的是主机系统提供操作系统的各设备和互提供面向设备的方法。

其主要特性包括：

- 设备接口
- 设备程序容易模块化
- 设备

最显著的改进之一是从平面和特殊系统演变成布局。

设备留的是“根”设备，它作为父设备，所有其他设备挂接在它上面。每个设备，通常“根”只有两个孩子，其上接着如host-to-PCI等。在x86，根设备“nexus”设备，在Alpha，Alpha的各不同型号有不同的设备

，不同的硬件芯片，包括 *lca*，*apecs*，*cia*和*tsunami*。

Newbus上下文中的`bus`表示系中的`bus`个硬件`bus`体。例如，`pci`个PCI`bus`被 `pci`表示`pci`一个Newbus`bus`。系中的任何`bus`可以有孩子；有孩子的`bus`通常被 `bus`称`bus`"bus"。系中常用`bus`的例子就是 `isa`和`pci`，他们各自管理`bus`接到ISA和PCI`bus`上的`bus`列表。

通常，不同`bus`型的`bus`之间的`bus`接被表示`bus`为`bus`"`bus`到`bus`"`bus`，它的孩子就是它所`bus`接的 `bus`。一个例子就是`pci-to-pci`，它在父PCI`bus`上被 `bus`表示`bus``pci``bus`，而用它的孩子 `bus`-`pci`表示`bus`接在它上面的 `bus`。布局`bus`化了PCI`bus`的`bus`，允`bus`公共代`bus`同`bus`用于`bus`和`bus`接的 `bus`。

Newbus`bus`中的`bus`个`bus`求它的父`bus`来`bus`其映射`bus`源。父`bus`接着`bus`求 `bus`它的父`bus`，直到到`nexus`。因此，基本上`nexus`是Newbus系中唯一知道所有 `bus`源的部分。



ISA`bus`可能想在`0x230`映射其IO端口，因此它向其 `bus`父`bus`求，`bus`情况下是ISA`bus`。ISA`bus`将它交`bus`PCI-to-ISA`bus`，PCI-to-ISA `bus`接着`bus`求PCI`bus`，PCI`bus`到`bus`host-to-PCI`bus`，最后到`nexus`。`bus`向上 `bus`渡的`bus`美之`bus`在于可以有空`bus`来`bus`求。`bus``0x230`IO端口 `bus`的`bus`求在MIPS机器上可以被PCI`bus`成 `0xb0000230`的内存映射。

`bus`源分配可以在`bus`的任何地方加以控制。例如，在很多Alpha平台上， `bus`ISA中断与PCI中断是`bus`独管理的，`bus`ISA中断的`bus`源分配是由Alpha的ISA`bus`管理的。在IA-32上，ISA和PCI中断都由`bus`的`nexus``bus`管理。`bus`于`bus`移植，`bus`内存和端口地址空`bus`由`bus`个`bus`体管理 - 在IA-32上是`nexus`，在Alpha（例如，CIA或*tsunami*）上是相`bus`的芯片`bus`程序。

`bus`了`bus`化`bus`内存和端口映射`bus`源的`bus`，Newbus整合了NetBSD的 `bus_space` `bus`API。他`bus`提供了`bus`一的API来代替`inb/outb` `bus`和直接内存`bus`写。`bus`做的`bus`在于`bus`个`bus`程序就可以使用内存映射寄存器或端口映射寄存器（有些硬件支持`bus`者）。

`bus`支持被合并到了`bus`源分配机制中。分配`bus`源`bus`，`bus`程序可以从`bus`源 `bus`中`bus`取`bus`的`bus_space_tag_t`和`bus_space_handle_t`。

Newbus也允`bus`在`bus`用于此目的的文件中定`bus`接口方法。`bus`些是`.m`文件，可以在`src/sys`目`bus`中`bus`到。

Newbus系`bus`的核心是可`bus`展的"基于`bus`象`bus`程(object-based `bus`programming)"的模型。系`bus`中的`bus`个`bus`具有它所支持的一个方法表。`bus`系`bus`和其他`bus`使用`bus`些方法来控制`bus`并`bus`求服`bus`。`bus`所支持的不同方法 `bus`被定`bus`多个"接口"。"接口"只是 `bus`的`bus`一`bus`相`bus`的方法。

在Newbus系`bus`中，`bus`方法是通`bus`系`bus`中的各`bus`程序提供的。当 `bus`自`bus`配置(auto-configuration)期`bus`被`bus`接(attach) 到`bus`程序，它使用`bus`程序声明的方法表。以后`bus`可以从其`bus`程序 `bus`分`bus`(detach)，并 `bus`重新`bus`接(re-attach)到具有新方法表的新`bus`程序。`bus`就 `bus`允`bus`程序的`bus`替`bus`，而`bus`替`bus`于`bus`程序的`bus`非常有用。

接口通`bus`与文件系`bus`中用于定`bus`vn`bus`ode操作的`bus`言相似的接口定`bus`言来描述。接口被保存在方法文件中（通常命名`bus`foo_if.m）。

```
# Foo 子系程序 (注...)

INTERFACE foo

METHOD int doit {
    device_t dev;
};

# 如果没有通过DEVMETHOD()提供一个方法, DEFAULT将会被使用的方法

METHOD void doit_to_child {
    device_t dev;
    driver_t child;
} DEFAULT doit_generic_to_child;
```

当接口被定义后, 它产生一个头文件 "foo_if.h", 其中包含函数声明:

```
int FOO_DOIT(device_t dev);
int FOO_DOIT_TO_CHILD(device_t dev, device_t child);
```

伴随自动产生的头文件, 也会创建一个源文件 "foo_if.c"; 其中包含一些函数的实现, 有些函数用于在对象方法表中指明相应函数的位置并用那个函数。

系统定义了三个主要接口。第一个基本接口被称为 "foo(device)", 并包括与所有相应的方法。"foo(device)"接口中的方法包括"探测(probe)", "连接(attach)"和"分离(detach)", 他们用来控制硬件的, 以及"关机(shutdown)", "挂起(suspend)"和"恢复(resume)", 他们用于事件通知。

第二个, 更加复杂的接口是"bus"。此接口包含的方法用于有孩子的设备, 包括设备特定的信息, 事件通知 (child_detached, driver_added) 和设备管理 (alloc_resource, activate_resource, deactivate_resource, release_resource)。

"bus"接口中的很多方法由某些孩子实现。有些方法通常使用前两个参量指定提供服务的方法和请求服务的子设备。除了优化驱动程序代码, 有些方法中的很多都有访问者(accessor)函数, 访问者函数用来访问父设备并用父设备上的方法。例如, 方法 BUS_TEARDOWN_INTR(device_t dev, device_t child, ...) 可以使用函数 bus_tearardown_intr(device_t child, ...)来调用。

系统中的某些设备型提供了额外接口以提供设备特定功能的设备。例如, PCI驱动程序定义了"pci"接口, 此接口有四个方法 read_config和 write_config, 用于访问PCI的配置寄存器。

13.3. Newbus API

由于Newbus API非常大, 本书努力将它文档化。本文的下一版本会提供更多详细信息。

13.3.1. 源代码目录中的重要位置

src/sys/[arch]/[arch] - 特定机器的内核代码位于该目录。例如 **i386** 或 **SPARC64**。

src/sys/dev/[bus] - 支持特定 **[bus]** 的代码位于该目录。

src/sys/dev/pci - PCI 支持代码位于该目录。

src/sys/[isa|pci] - PCI/ISA 驱动程序位于该目录。FreeBSD 4.0 版本中，PCI/ISA 支持代码去存在于该目录中。

13.3.2. 重要结构和类型定义

devclass_t - 是指向 **struct devclass** 的指针的类型定义。

device_method_t - 与 **kobj_method_t** 相同（参看 `src/sys/kobj.h`）。

device_t - 是指向 **struct device** 的指针的类型定义。**device_t** 表示系统中的设备。它是内核对象。参看 `src/sys/sys/bus_private.h`。

driver_t - 是一个类型定义，它引用 **struct driver**。**driver** 是一个 **device()** 内核对象；它也保存着该程序的私有数据。

driver_t

```
struct driver {
    KOBJ_CLASS_FIELDS;
    void    *priv;          /* 程序私有数据 */
};
```

device_state_t 是一个枚举型，即 **device_state**。它包含 Newbus 在自身配置前后可能的状态。

状态 **device_state_t**

```
/*
 * src/sys/sys/bus.h
 */
typedef enum device_state {
    DS_NOTPRESENT, /* 未探测或探测失败 */
    DS_ALIVE,      /* 探测成功 */
    DS_ATTACHED,   /* 用了连接方法 */
    DS_BUSY        /* 已打 */
} device_state_t;
```


Chapter 14. 声音子系统

14.1. 简介

FreeBSD声音子系统清晰地将通用声音处理与特定的部分分离开来。这使得更容易加入新硬件的支持。

`pcm(4)`框架是声音子系统的中心部分。它主要包含下面的组件：

- 一个到数字化声音和混音器函数的系统级接口（`read`, `write`, `ioctl`s）。`ioctl`命令集合兼容老的OSS或Voxware接口，允许一般多媒体应用程序不加修改地移植。
- 处理声音数据的公共代码（格式转换，虚拟通道）。
- 一个统一的设备接口，与硬件特定的音频接口模块接口
- 某些通用硬件接口（`ac97`）或共享的硬件特定代码（例如：ISA DMA例程）的外围支持。

特定硬件的支持是通过硬件特定的驱动程序来实现的，这些驱动程序提供通道和混音器接口，并嵌入到通用`pcm`代码中。

本章中，`pcm`将指声音子系统的中心，通用部分，这是与硬件特定的模块而言的。

早期的驱动程序作者当然希望从有模块开始，并使用那些代码作为最参考。但是，由于声音代码十分漂亮，这也基本上免除了烦恼。本文概述框架接口的一个概貌，并回答改写有代码可能出的一些问题。

作为另外的途径，或者除了从一个可工作的例程开始的办法之外，
可以从<http://people.FreeBSD.org/~cg/template.c>到一个带注释的驱动程序模板。

14.2. 文件

除`/usr/src/sys/soundcard.h`中的公共 `ioctl`接口定义外，所有的相关代码当前(FreeBSD 4.4)位于 `/usr/src/sys/dev/sound/`。

在`/usr/src/sys/dev/sound/`下面，`pcm/`目录中保存着中心代码，而`isa/`和`pci/`目录中有ISA和PCI板的驱动程序。

14.3. 探测，连接等

声音驱动程序使用与任何硬件驱动程序相同的方法探测和连接（检测）。可能希望看一下手册中ISA或PCI章节的内容来获取更多详细信息。

然而，声音驱动程序在某些方面又有些不同：

- 他将自己声明`pcm`结构，有一个私有`struct snddev_info`：

```
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);
```

大多数声音程序需要存于其附加私有信息。私有数据 通常在接口例程中分配。其地址通 用 `pcm_register()` 和 `mixer_init()` 返回 pcm。后面 pcm 将此地址作 用声音程序接口 的参数 回来。

- 声音程序的接口例程 当通 用 `mixer_init()` 向 pcm 声明它的 MIXER 或 AC97 接口。 于 MIXER 接口， 会接着引起 用 `xxxmixer_init()`。
- 声音程序的接口例程通 用 `pcm_register(dev, sc, nplay, nrec)` 向 pcm 声明其通用 CHANNEL 配置，其中 `sc` 是 数据 的地址，在 pcm 以后的 用中将会用到它，`nplay` 和 `nrec` 是播放和 音 通道的数目。
- 声音程序的接口例程通 用 `pcm_addchan()` 声明它的 个通道 象。 会在 pcm 中建立起通道合成，并接着会引起 用 `xxxchannel_init()`（注： 参考原文）。
- 声音程序的分 例程在 放其 源之前 当 用 `pcm_unregister()`。

有 可能的方法来 理非 PnP：

- 使用 `device_identify()` 方法（例：sound/isa/es1888.c）。 `device_identify()` 方法在已知地址探 硬件，如果 支持的 就会 建一个新的 pcm， 个 pcm 接着 会被 到 probe/attach。
- 使用定制内核配置的方法， pcm 置 当的 hints（例：sound/isa/mss.c）。

pcm 程序 当 `device_suspend`，`device_resume` 和 `device_shutdown` 例程， 源管理和模 卸 就能 正 地 作用。

14.4. 接口

pcm 核心与声音程序之 的接口以 内核 象的叫法来定。

声音程序通常提供 主要的接口：`CHANNEL` 以及 `MIXER` 或 `AC97`。

AC97 是一个很小的硬件 （寄存器/写） 接口，由 程序 解 器的硬件来 。 情况下， 的 MIXER 接口由 pcm 中共享的 AC97 代 提供。

14.4.1. CHANNEL 接口

14.4.1.1. 函数参数的通常注意事

声音程序通常用一个私有数据 来描述他 的 ， 程序所支持的播放和 音数据通道各有一个。

于所有的 CHANNEL 接口函数，第一个参数是一个不透明的指 。

第二个参数是指向私有的通道数据**obj**的指针，`channel_init()`是个例外，它的指针指向私有数据（并返回由pcm以后使用的通道指针）。

14.4.1.2. 数据操作概述

对于声音数据，pcm核心与声音程序是通过一个由**struct snd_dbuf**描述的共享内存区域进行通信的。

struct snd_dbuf是pcm私有的，声音程序通常用所有者函数（`sndbuf_getxxx()`）来得感兴趣的。

共享内存区域的大小等于 `sndbuf_getsize()`，并被分割成大小固定，且等于 `sndbuf_getblksz()`字节的很多块。

当播放时，常用的机制如下（将意思反过来就是录音）：

- pcm开始填充缓冲区，然后以参数**PCMTRIG_START**用声音程序的**xxxchannel_trigger()**。
- 声音程序接着安排以 `sndbuf_getblksz()`字节大小，重将整个内存区域（`sndbuf_getbuf()`，`sndbuf_getsize()`）读到。由于一个回调pcm函数 **chn_intr()**（通常在中断中产生）。
- **chn_intr()**安排将新数据拷到那些数据已读到（或在空）的区域，并对 **snd_dbuf**进行适当的更新。

14.4.1.3. channel_init

用**xxxchannel_init()**来初始化一个播放和录音通道。它调用从声音程序的接口例程中引起。（参看 [探测和接口](#)）。

```
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;
}
```

通道 **struct snd_dbuf**的地址。它应当在函数中通常用**sndbuf_alloc()**来初始化。所用的缓冲区大小通常是典型大小的一个小的倍数。 **c**是pcm通道控制结构的指针。它是一个不透明指针。函数应当将它保存到局部通道中，在后面用pcm函数（例如：`chn_intr(c)`）会使用它。**dir**指示通道方向（**PCMDIR_PLAY**或**PCMDIR_REC**）。函数应当返回一个指针，此指针指向用于控制此通道的私有区域。它将被参数被用到其他通道接口的调用。

14.4.1.4. channel_setformat

xxxchannel_setformat()应当按特定通道，特定声音格式设置硬件。

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

`format` 是 `AFMT_XXX` value 之一 (soundcard.h)。

14.4.1.5. channel_setspeed

`xxxchannel_setspeed()` 按指定的取速度 设置通道硬件，并返回可能调整的速度。

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

14.4.1.6. channel_setblocksize

`xxxchannel_setblocksize()` 设置大小， 是 pcm 与声音程序，以及声音程序与之间的位的大小。 期间， 每次数据后，声音程序都当用 pcm 的 `chn_intr()`。

大多数程序只注意儿的大小，因当开始使用个。

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

函数返回可能调整的大小。如果大小真的化了， 情况下当用 `sndbuf_resize()` 调整缓冲区的大小。

14.4.1.7. channel_trigger

`xxxchannel_trigger()` 由 pcm 来控制程序中的操作。

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

go 定当前用的作。可能的：



如果程序使用ISA DMA，当在上行作前 用sndbuf_isadma()，并理DMA芯片一方的事情。

14.4.1.8. channel_getptr

xxxchannel_getptr()返回缓冲区中 当前的冲。它通常由chn_intr()用，而且 也是什pcm知道它 当往儿送 新数据。

14.4.1.9. channel_free

用xxxchannel_free()来放通道源， 例如当程序卸，并且如果通道数据是分配的，或者如果不使用sndbuf_alloc()行冲区分配， 当个函数。

14.4.1.10. channel_getcaps

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

个例程返回指向（通常静定的） pcmchan_caps的指（在 sound/pcm/channel.h中定）。个 保存着最小和最大采率和被接受的声音格式。任何声音 程序都可以作一个例。

14.4.1.11. 更多函数

channel_reset(), channel_resetdone()和 channel_notify()用于特殊目的，未与威人士（Cameron Grant）行探之前不 当在程序中它。

不成使用channel_setdir()。

14.4.2. MIXER接口

14.4.2.1. mixer_init

xxxmixer_init()初始化硬件，并告 pcm什混音器 可用来播放和音。

```

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

    [初始化硬件]

    [播放混音器置v中当的位].
    mix_setdevs(m, v);
    [音混音器置v中当的位]
    mix_setrecdevs(m, v)

    return 0;
}

```

置一个整数中的位，并用 `mix_setdevs()` 和 `mix_setrecdevs()` 来告 pcm 存在什。

混音器的位定可以在 `soundcard.h` 中到。（`SOUND_MASK_XXX` 和 `SOUND_MIXER_XXX` 移位）。

14.4.2.2. mixer_set

`xxxmixer_set()` 混音器置音量 (level)。

```

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

```

被指定 `SOUND_MIXER_XXX` 在 [0-100] 之指定音量。零当静音。由于硬件(音量) (level) 可能不匹配入比例，会出某些整，例程返回如上面所示的 (0-100 内)。

14.4.2.3. mixer_setrecsrc

`xxxmixer_setrecsrc()` 定音源。

```

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

    [看src中的非零位, 置硬件]

    [更新src反映操作]
    return src;.
}

```

期望的音由一个位域指定。返回置用来音的。一些程序只能置一个音。如果出，函数当返回-1。

14.4.2.4. mixer_uninit, mixer_reinit

`xxxmixer_uninit()`当保不会出任何声音，并且如果可能当混音器硬件断。

`xxxmixer_reinit()`当保混音器硬件加，并且恢所有不受`mixer_set()`或 `mixer_setrecsrc()`控制的置。

14.4.3. AC97接口

AC97由有AC97解器的程序。它只有三个方法：

- `xxxac97_init()`返回到的 ac97解器的数目。
- `ac97_read()`与 `ac97_write()`写指定的寄存器。

The AC97接口由 pcm中的AC97代来行高操作。参看 `sound/pci/maestro3.c`或 `sound/pci/`下很多其他内容作例。

Chapter 15. PC Card

本章将向FreeBSD编写PC Card或CardBus的驱动程序而提供的机制。但目前本文只介绍了如何向已有的pccard驱动程序中添加驱动程序。

15.1. 添加驱动

向所支持的pccard列表中添加新驱动的已与系在FreeBSD 4中使用的方法不同了。在以前的版本中，需要在/etc中的一个文件来列出驱动。从FreeBSD 5.0开始，驱动程序知道它支持什么。驱动在内核中有一个受支持的表的，驱动程序用它来接管。

15.1.1. 概述

可以有几种方法来PC Card，他们都基于上的CIS信息。第一种方法是使用制造商和产品的数字号。第二种方法是使用人可读的字符串，字符串也是包含在CIS中。PC Card使用集中式数据和一些宏来提供一个易用的模式，驱动程序的编写者很容易地定义匹配其驱动程序的。

一个很普遍的情况是，某个公司一款PC Card产品出参考，然后把个外的公司，以便在市上出售。那些公司改，把他自己的目客群或地理区域出售产品，并将他自己的名字放到中。然而所有的改，即使做任何修改，些修改通常也微乎其微。然而，化了他自己的品牌的，些供应商常常会把他的名字放入CIS空的可字符串中，却不会改制造商和产品的ID。

于以上情况，于FreeBSD来使用数字ID可以小工作量。同也会小将ID加入到系的程中所来的性。必仔真的是真正制造者，尤其当提供原的供应商在中心数据中已有一个不同的ID。Linksys, D-Link和NetGear是常出售相同的几个美国制造商。相同的可能在日本以如Buffalo和Corega的名字出售。然而，些常常具有相同的制造商和产品ID。

PC Card在其中心数据/sys/dev/pccard/pccarddevs中保存了的信息，但不包含个程序与它的信息。它也提供了一套宏，以允在程序用来声明表的表中容易地建条目。

最后，某些非常低端的根本不包含制造商。些需要使用可CIS字符串来匹配它。如果我不需要急急法有多好，但于某些非常低端却非常流行的CD-ROM播放器来却是必需的。通常当避免使用的方法，但本中是列出了很多，因它是在到PC Card商的OEM本之前加入的，当先使用数字方法。

15.1.2. pccarddevs的格式

pccarddevs文件有四。第一使用它的那些供应商列出了制造商号。本按数字排序。下一包含了些供应商使用的所有产品，包括他的产品ID号和描述字符串。描述字符串通常不会被使用（相反，即使我可以匹配数字版本号，我仍然基于人可读的CIS置的描述）。然后使用字符串匹配方法的那些重西。最后，文件任何地方可以使用C的注。

文件的第一包含供应商ID。保持列表按数字排序。此外，了能有一个通用清晰的保存地来方便地保存些信息，我与NetBSD共享此文件，因此此文件的任何更改。例如：


```

vendor FUJITSU      0x0004 Fujitsu Corporation
vendor NETGEAR_2    0x000b Netgear
vendor PANASONIC    0x0032 Matsushita Electric Industrial Co.
vendor SANDISK      0x0045 Sandisk Corporation

```

显示了几个供应商ID。很巧的是NETGEAR_2 是NETGEAR从其OEM，那些提供支持的作者那并不知道 NETgear使用的是人的ID。这些条目相当直接易懂。行上都有供应商 名字来指示本行的。也有供应商的名字。名字将会在pccarddevs文件的后面重出，名字也会用在程序的匹配表中，因此保持它的短小 并且是有效的C符号。有一个供应商的十六进制数字ID。不要添加0xffffffff或0xffff形式的ID， 因为它保留ID（前者是‘空ID集合’，而后者有会在少量其差的中看到，用来指示none）。最后有用于制公司的描述字符串。个字符串 在FreeBSD中除了用于注目的外并没有被使用。

文件的第二包含品。如在下例中看到的：

```

/* Allied Telesis K.K. */
product ALLIEDTELESIS LA_PCM 0x0002 Allied Telesis LA-PCM

/* Archos */
product ARCHOS ARC_ATAPI 0x0043 MiniCD

```

格式与供应商的那些行相似。其中有品字。然后是供应商名字， 由上面重而来。后面跟着品名字，此名字在程序中使用，且当 是一个有效C符号，但可以以数字。然后是十六进制品ID。供应商通常0xffffffff和 0xffff有相同的定。最后是于自身的字符串 描述。由于FreeBSD的pccard程序会从人可的CIS条目建一个 字符串，因此个字符串在FreeBSD中通常不被使用，但某些CIS条目不能 足要求的情况下可能使用。品按制造商的字母序排序，然后再按品ID的数字排序。个制造商条目前有一条C注，条目之有一个空行。

第三很象前面的供应商一，但所由的制造商ID -1。-1在FreeBSD pccard 代中意味着“匹配的任何西”。由于它是C符号， 它的名字必唯一。除此之外格式等同于文件的第一。

最后一包含那些必用字符串匹配的。一的格式与通用 的格式有点不同：

```

product ADDTRON AWP100 { "Addtron", "AWP-100spWirelessspPCMCIA",
"Versionsp01.02", NULL }
product ALLIEDTELESIS WR211PCM { "AlliedspTelesisspK.K.", "WR211PCM", NULL, NULL }
Allied Telesis WR211PCM

```

我已熟悉了品字，后跟供应商名字，然后再跟的名字， 就象在文件第二中那。然而， 之后就与那格式不同了。有一个 {}分，后跟几个字符串。些字符串CIS_INFO三元中定的 供应商，品和 外信息。些字符串被生 pccarddevs.h的程序，将 sp替 的空格。空条目意味着条目的部分当被忽略。在我 的例子中 有一个 的条目。除非 的操作来至重要，否不当在其中包含版本号。有 供应商在个字段中会有 的很多不同版本，些版本 都能工作， 情况下那些信息只会 那些有相似的人在FreeBSD中 更以使用。有当 供应商出于市考（可用性，价格等等），希望出售同一品牌下的很多不同部分， 也是有必要的。如果， 在那些 供应商仍然保持相同的制造商/品的少 情况下，能否区分至 重要。此不能使用正表式匹配。

15.1.3. 探测例程示例

要使得如何向所支持的表中添加，就必须得很多程序 都有的探测和/或匹配例程。由于也老 提供了一个兼容，在 FreeBSD 5.x中有一点。由于只是window-dressing不同，儿出了一个理想化的版本。

```
static const struct pccard_product wi_pccard_products[] = {
    PCMCIA_CARD(3COM, 3CRWE737A, 0),
    PCMCIA_CARD(BUFFALO, WLI_PCM_S11, 0),
    PCMCIA_CARD(BUFFALO, WLI_CF_S11G, 0),
    PCMCIA_CARD(TDK, LAK_CD011WL, 0),
    { NULL }
};

static int
wi_pccard_probe(dev)
    device_t dev;
{
    const struct pccard_product *pp;

    if ((pp = pccard_product_lookup(dev, wi_pccard_products,
        sizeof(wi_pccard_products[0]), NULL)) != NULL) {
        if (pp->pp_name != NULL)
            device_set_desc(dev, pp->pp_name);
        return (0);
    }
    return (ENXIO);
}
```

儿我有一个可以匹配少数几个的pccard探测例程。如上面 所提到，名字可能不同（如果不是 `foo_pccard_probe()` 就是 `foo_pccard_match()`）。函数 `pccard_product_lookup()` 是一个通用函数，它遍 表并返回指向它所匹配的第一的指针。一些程序可能使用个机制来 将某些的附加信息 到 程序的其它部分，因此表中可能有些体。 唯一的要求就是如果有一个不同的表，表的 的第一个元素 是 `pccard_product`。

察一下表 `wi_pccard_products` 就会， 所有条目都是 `PCMCIA_CARD(foo, bar, baz)` 的形式。 `foo` 部分 来自 `pccarddevs` 的制造商ID。 `bar` 部分 品。 `baz` 此 所期望的功能号。多 `pccards` 可以有多个功能，需要有法区分功能1和功能0。可以看一下 `PCMCIA_CARD_D`，它包括了来自 `pccarddevs` 文件的描述。也可以看看 `PCMCIA_CARD2` 和 `PCMCIA_CARD2_D`，当需要按 "使用默认描述" 和 "从 `pccarddevs` 中取得" 做法，同匹配CIS字符串和制造商号就会用到它。

15.1.4. 将它合在一起

因此，了一个加新，必行下面。首先，必从 得信息。完成个最容易的方法就是将 入到PC Card或CF槽中，并出 `devinfo -v`。可能会看到一些似下面的西：

```
cbb1 pnpinfo vendor=0x104c device=0xac51 subvendor=0x1265 subdevice=0x0300
class=0x060700 at slot=10 function=1
    cardbus1
    pccard1
        unknown pnpinfo manufacturer=0x026f product=0x030c cisvender="BUFFALO"
cisproduct="WLI2-CF-S11" function_type=6 at function=0
```

作出一部分。制造商和产品的数字ID。而cisvender和cisproduct是CIS中提供的描述本产品的字符串。

由于我首先想使用数字ID，因此首先建立基于此的条目。为了示例，上面的ID已被虚拟化。我看到的供应商是BUFFALO，它已有一个条目了：

```
vendor BUFFALO          0x026f  BUFFALO (Melco Corporation)
```

我就可以了。这个是一个条目，但我没有ID。但我可以：

```
/* BUFFALO */
product BUFFALO WLI_PCM_S11 0x0305  BUFFALO AirStation 11Mbps WLAN
product BUFFALO LPC_CF_CLT   0x0307  BUFFALO LPC-CF-CLT
product BUFFALO LPC3_CLT     0x030a  BUFFALO LPC3-CLT Ethernet Adapter
product BUFFALO WLI_CF_S11G  0x030b  BUFFALO AirStation 11Mbps CF WLAN
```

我就可以向pccarddevs中添加：

```
product BUFFALO WLI2_CF_S11G    0x030c  BUFFALO AirStation ultra 802.11b CF
```

目前，需要一个手册来重新生成pccarddevs.h，用来将一些符号添加到客程序。在程序中使用它之前必须完成下面：

```
# cd src/sys/dev/pccard
# make -f Makefile.pccarddevs
```

一旦完成了这些，就可以向程序中添加了。只是一个添加一行的操作：

```
static const struct pccard_product wi_pccard_products[] = {
    PCMCIA_CARD(3COM, 3CRWE737A, 0),
    PCMCIA_CARD(BUFFALO, WLI_PCM_S11, 0),
    PCMCIA_CARD(BUFFALO, WLI_CF_S11G, 0),
+   PCMCIA_CARD(BUFFALO, WLI_CF2_S11G, 0),
    PCMCIA_CARD(TDK, LAK_CD011WL, 0),
    { NULL }
};
```

注意，我在我添加的行前面包含了`+`，但它只是用来做一行。不要把它添加到任何程序中。一旦你添加了一行，就可以重新编译内核或模块，并查看它是否能工作。如果它输出并能工作，就提交一个补丁。如果它不工作，就输出它工作所需要的东西并提交一个补丁。如果它根本不工作，那可能做了些别的，就当重新来一次。

如果你是一个FreeBSD源代码的committer，并且所有东西看起来都正常工作，就当把那些改动提交到主分支中。然而有些小技巧的东西需要考虑。首先，你必须提交pccarddevs文件到主分支中。完成后，你必须重新生成pccarddevs.h并将它作一次提交来提交（这是为了保证的\$FreeBSD\$会留在后面的文件中）。最后，你需要把其它东西提交到主程序。

15.1.5. 提交新东西

很多人直接把新东西的条目送给作者。不要那么做。将它作为一个PR来提交，并将PR号送给作者用于跟踪。这能保证条目不会丢失。提交一个PR，补丁中没有必要包含pccarddevs.h的diff，因为那些东西可以重新生成。包含补丁的描述和客户程序的补丁是必要的。如果你不知道名字，使用OEM99作为名字，作者将会随后相应地调整OEM99。提交者不当提交OEM99，而得到最高的OEM条目并提交高于那个的一个。

Part III: 附

参考目

[1] Marshall Kirk McKusick、Keith Bostic、Michael J Karels和John S Quarterman. 版 © 1996 Addison-Wesley Publishing Company, Inc.. 0-201-54979-4. Addison-Wesley Publishing Company, Inc.. *The Design and Implementation of the 4.4 BSD Operating System*. 1-2.