FreeBSD 系□□□手册

摘要

□迎□□□ FreeBSD系□□□手册。 □本手册□在不断由□多人_□□□写_。 □多章□□是空白，有的章□亟待更新。如果□□□个□目感□趣并□意有所□献，□□信□ FreeBSD 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□器 或□多的 □像站点 得到□□文□的各□其他格式以及□□形式的版本。

目□

部分 I: 内核

Chapter 1. 引□□程与内核初始化

1.1. 概述

□一章是□引□□程和系□初始化□程的□□。□些□程始于BIOS(固件)POST, 直到第一个用□□程建立。由于系□□□的最初□□是与硬件□□相□的、是□配合的, □里用IA-32(Intel Architecture 32bit)□□作□例子。

1.2. □□

一台□行FreeBSD的□算机有多□引□方法。□里□□其中最通常的方法, 也就是从安装了操作系□的硬□上引□。引□□程分几□完成:

- BIOS POST
- boot0□段
- boot2□段
- loader□段
- 内核初始化

boot0和boot2□段在手册 boot(8)中被称□*bootstrap stages 1 and 2*, 是FreeBSD的3□段引□□程的□始。在□一□段都有各□各□的信息□示在屏幕上, □可以参考下表□□出□些□□。□注意□□的□示内容可能随机器的不同而有一些区□:

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

```
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..._
```

loader

```
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
```

内核

1.3. BIOS POST

当PC加□后，□理器的寄存器被□□某些特定□。在□些寄存器中，　　指令指□寄存器被□□32位□0xffffff0。指令指□寄存器指向□理器将要□行的指令代□。cr1，　　一个32位控制寄存器，在□□□□被□□0。cr1的PE(Protected　Enabled，　保□模式使能)位用来指示□理器是□于保□模式□是□地址模式。　由于□□□□位被清位，□理器在□地址模式中引□。在□地址模式中，□性地址与物理地址是等同的。

□0xffffff0略小于4G,因此□算机没有4G字□物理内存，　　□就不会是一个有效的内存地址。□算机硬件将□个地址□指向BIOS存□□。

BIOS表示*Basic Input Output System*（基本□入□出系□)。在主板上，它被固化在一个相□容量□小的 只□存□器(Read-Only　Memory，　ROM)。BIOS包含各□各□□主板硬件　　定制的底□例程。就□□，□理器首先指向常□BIOS存□器的地址 0xffffff0。通常□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□自□)。 □套程序包括内存□□，系□□□□□和其它底□工具， 从而使得CPU能□初始化整台□算机。□一□段中有一个重要□□，　　就是□定引□□□。□在所有的BIOS都允□手工□□引□□□。　□可以从□□、光□□□器、硬□等□□引□。

POST的最后一□是□行INT　　0x19指令。　　□个指令从引□□□第一个扇区□取512字□，装入地址0x7c00。第一个扇区的□法最早起源于硬□的□□，　　硬□面被分□若干□柱形□道。□□道□号，同□又将□道分□　一定数目(通常是64)的扇形。0号□道是硬□的最外圈，1号扇区，　第一个扇区(□道、柱面都从0□始□号，而扇区从1□始□号) 有着特殊的作用，它又被称□主引□□□(Master Boot Record, MBR)。 第一□剩余的扇区常常不使用。

1.4. boot0□段

□我□看一下文件/boot/boot0。 □是一个□512字□的小文件。如果在FreeBSD安装□程中□□ "bootmanager"，□个文件中的内容将被写入硬□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□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□□中， □被称□slice(d)。16字□中有一个□志字□决定□个分区是否可引□。 有□只能有一个分区可□定□一□志。否□， boot0的代□将拒□□□□行。

一个分区□□有如下域：

- 1字□ 文件系□□型
- 1字□ 可引□□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。 LBA和CHS□□描述符指示相同的信息，但是指示方式有所不同：LBA (□□□□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□管理器□描分区表，并在屏幕上□示菜□，以便用□可以 □□用于引□的磁□和分区。在□□上按下相□的□后，boot0□行如下□作：

- □□□中的分区□可引□，清除以前的可引□□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□系□的。 从□□引□□，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□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□□□，□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.ld -P 1 boot2.bin
```

□个Makefile片断表明btxld(8)被用来□接二□制代□。 BTX表示引□□展器(BooT eXtender)是□程序(称□客□(client) 提供保□模式□境、并与客□程序相□接的一段代□。所以 boot2是一个BTX客□，使用BTX提供的服□。

工具btxld是□接器， 它将□个二□制代□□接在一起。btxld(8)和ld(1) 的区□是ld通常将□个目□文件 □接成一个□□□接□或可□行文件，而btxld □将一个目□文件与BTX□接起来，□生□合于放在分区首部的二□制代□， 以□□系□引□。

boot0□行跳□至BTX的入口点。 然后，BTX将□理器切□至保□模式，并准□一个□□的□境， 然后□用客□。□个□境包括：

- 虚□8086模式。□意味着BTX是虚□8086的□□程序。 □模式指令，如pushf, popf, cli, sti, if，均可被客□□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□理。建立中断0x30，□是系□□用□口。
- □个系□□用exec和 exit的定□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□□用0x0
        int $INT_SYS            #
#
# System call: exec
#
__exec:     movl $0x1,%eax          # BTX系□□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□的代□和数据始于地址MEM_USR(0xa000)，□□符(selector) SEL_UCODE指向客□的数据段。□□符SEL_UCODE □有第3□描述符□限 (Descriptor Privilege Level, DPL)，□是最低□□限。但是 INT 0x30 指令的□理程序存□于□一个段里， □个段的□□符SEL_SCODE (supervisor code)由有着管理□□限。 正如代□建立IDT(中断描述符表)□□行的操作那□:

```
        mov $SEL_SCODE,%dh      # 段□□符
init.2:     shr %bx             # 是否□理□个中断？
        jnc init.3          # 否
        mov %ax,(%di)           # □置□理程序偏移量
        mov %dh,0x2(%di)        # □置□理程序□□符
        mov %dl,0x5(%di)        # □置 P:DPL:type
        add $0x4,%ax            # 下一个中断□理程序
```

所以，当客□□用 __exec()□，代□将被以最高□限□行。 □使得内核可以修改保□模式数据□□，如分□表(page tables)、全局描述符表(GDT)、 中断描述符表(IDT)等。

boot2 定□了一个重要的数据□□： struct bootinfo。□个□□由 boot2 初始化，然后被□送到loader，之后又被□入内核。 □个□□的部分□目由boot2□定，其余的由loader□定。 □个□□中的信息包括内核文件名、BIOS提供的硬□柱面/磁□/扇区数目信息、 BIOS提供的引□□□的□□器□号，可用的物理内存大小，envp 指□(□境指□)等。定□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□□ */
#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□□□的BIOS□元□号 */
    u_int8_t    bi_pad[2];
    u_int32_t   bi_basemem;
    u_int32_t   bi_extmem;
    u_int32_t   bi_symtab;      /* struct symtab * */
    u_int32_t   bi_esymtab;     /* struct symtab * */
        /* 以下□目□高□bootloader提供 */
    u_int32_t   bi_kernend;     /* 内核空□末端 */
    u_int32_t   bi_envp;        /* □境 */
    u_int32_t   bi_modulep;     /* □装□的模□ */
};
```

boot2 □入一个循□等待用□□入，然后□用 load()。如果用□不做任何□入，循□将在一段□□后□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□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□的BTX的机理已□在前面□□□。

loader 的主要任□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□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□制文件， 可是□□□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：

recover_bootinfo	□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□□引□, 无□引□方式。 □个函数决定引□方式，并将□□struct bootinfo 存□至内核内存。
identify_cpu	□个函数□□CPU□型，将□果存放在□量 _cpu中。
create_pagetables	□个函数□分□表在内核内存空□□部分配一□空□，并填写一定内容

下一□是□□VME(如果CPU有□个功能):

```
    testl   $CPUID_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□内核被□接后真正的入口 */
begin:
```

函数init386()被□用；随参数□□的是一个指□，　　指向第一个空□物理□。随后□行mi_startup()。init386是一个与硬件系□相□的初始化函数， mi_startup()是个与硬件系□无□的函数 (前□'mi_'表示Machine Independent，不依□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□，等 */
    hlt     /* 不再返回到□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□程，即内核的□程)的pcb(□程控制□)

init386()首先初始化内核的可□整参数， □些参数由引□程序□来。先□置□境指□(environment pointer, envp)□用， 再□用init_param1()。 envp指□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□整□,如hz等 */
    init_param1();
```

init_param1()定□在 sys/kern/subr_param.c之中。 □个文件里有一些sysctl□，□有□个函数，init_param1()和init_param2()。 □□个函数从init386()中□用:

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

TUNABLE_typename_FETCH用来□取□境□量的□:

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

Sysctlkern.hz是系□□□□率。同□， □些sysctl□被init_param1()□定: kern.maxswzone, kern.maxbcache, kern.maxtsiz, kern.dfldsiz, 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□个任□都□行在自己的虚□地址空□里， □个空□由"段址:偏移量"的数□指定。□个例子，当前将要由□理器□行的指令在 CS:EIP，那□□条指令的□性虚□地址就是"代□段虚□段地址CS" + EIP。 □了□便，段起始于虚□地址0，□止于界限4G字□。所以，在□个例子中， 指令的□性虚□地址正是EIP的□。段寄存器，如CS、DS等是□□符，即全局描述符表中的索引(更精□的□，索引并非□□符的全部， 而是□□符中的INDEX部分)。

□者注

□于80386， □□符有16位，INDEX部分是其中的高13位。

FreeBSD的全局描述符表□□个CPU保存着15个□□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□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□□程 */
#define GLDT_SEL    5   /* □个□程的局部描述符表 */
#define GUSERLDT_SEL    6   /* 用□自定□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□ */
#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，□些#defines并非□□符本身，而只是□□符中的INDEX域，因此它□正是全局描述符表中的索引。例如，内核代□的□□符(GCODE_SEL)的□□0x08。

下一□是初始化中断描述符表(Interrupt Descriptor Table, IDT)。□□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。□是一个□件中断，□理器用索引□0x80在中断描述符表中□□□□。□个□□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□□。□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□口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□□用程序提交INT 0x80指令□，全系□的控制□会□□□函数_Xint0x80_syscall，□个函数在内核代□段中，将被以管理□□限□行。

然后，控制台和DDB(□□器)被初始化:

```
sys/i386/i386/machdep.c:
    cninit();
/* 以下代□可能因□未定□宏DDB而被跳□ */
#ifdef DDB
    kdb_init();
    if (boothowto  RB_KDB)
        Debugger("Boot flags requested debugger");
#endif
```

任□状□段(TSS)是□一个x86保□模式中的数据□□。当□生任□切□□，任□状□段用来□硬件存□任□□□信息。

局部描述符表(LDT)用来指向用□代□和数据。系□定□了几个□□符，指向局部描述符表，它□是系□□用□口和用□代□、用□数据□□符:

```
/usr/include/machine/segments.h
#define LSYS5CALLS_SEL  0   /* Intel BCS□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□□用□口 */
#define LUDATA_SEL  5
/* separate stack, es,fs,gs sels ? 分□的□、es、fs、gs□□符？ */
/* #define  LPOSIXCALLS_SEL 5*/ /* notyet, 尚无 */
#define LBSDICALLS_SEL  16  /* BSDI system call gate, BSDI系□□用□口 */
#define NLDT        (LBSDICALLS_SEL + 1)
```

然后，proc0(0号□程，即内核所□的□程)的□程控制□(Process Control Block) (struct pcb)□□被初始化。proc0是一个 struct proc □□，描述了一个内核□程。内核□行□，□□程□是存在，所以□个□□在内核中被定□□全局□量:

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

□□struct pcb是proc□□的一部分，它定□在/usr/include/machine/pcb.h之中，内含□□i386硬件□□□有的信息，如寄存器的□。

1.7.2. mi_startup()

□个函数用冒泡排序算法，将所有系□初始化□象，然后逐个□用□个□象的入口:

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

        /* ... 省略 ... */

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

尽管sysinit框架已□在《FreeBSD□□者手册》中有所描述， 我□是在□里□□一下其内部原理。

□个系□初始化□象(sysinit□象)通□□用宏建立。 □我□以announce sysinit□象□例。 □个□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)
```

□个□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□成宏C_SYSINIT()。 宏C_SYSINIT()然后展□成一个静□□□ struct sysinit。□□里申明里□用了□一个宏 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□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□的例子中，□□宏的展□□程，将会□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□行文件中建立一个ELF□(section)。 □□生在内核□接的□候。□一□将被命令□.set.sysinit_set。 □一□的内容是一个32位□——announce_sys_init□□的地址，□个□□正是第二个__asm指令所定□的。第三个__asm指令□□□的□束。 如果前面有名字相同的□定□□句，□的内容(那个32位□)将被填加到已存在的□里， □□就□造出了一个32位指□数□。

用objdump察看一个内核二□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.kbddriver_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 0000097c  c0316514  c0316514  00215514  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 12 .set.sysinit_set 00000664  c0316e90  c0316e90  00215e90  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

□一屏信息□示表明□.set.sysinit_set有0x664字□的大小， 所以0x664/sizeof(void *)个sysinit□象被

□□□了内核。 其它□，如.set.sysctl_set表示其它□接器集合。

通□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□接器使用 */
    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    = 0xfffffff /* □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)□0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□□用fork1()　　分配一个新的□程，但并不将其□□□可□行。当□个新□程被□度器□度□行□，start_init()将会被□用。　　那个函数定□在init_main.c中。　　它□□装□并□行二□制代□init，　　先□□/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* □□。 □将□□和建□□送□*FreeBSD* □称多□理 *(SMP)* □件列表.

□篇文□提□□□的□述了在FreeBSD内核中的□，□些□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□。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□有。

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□趣的属性:

□量名

在内核源代□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

□量名	□□名	□型	保□□象	依□函数
sched_lock	"sched lock"(□度器□)	MTX_SPIN | MTX_RECURSE	_gmonparam, cnt.v_swtch, cp_time, curpriority, mtx.mtx_blocked, mtx.mtx_contested, proc.p_procq, proc.p_slpq, proc.p_sflag, proc.p_stat, proc.p_estcpu, proc.p_cpticks proc.p_pctcpu, proc.p_wchan, proc.p_wmesg, proc.p_swtime, proc.p_slptime, proc.p_runtime, proc.p_uu, proc.p_su, proc.p_iu, proc.p_uticks, proc.p_sticks, proc.p_iticks, proc.p_oncpu, proc.p_lastcpu, proc.p_rqindex, proc.p_heldmtx, proc.p_blocked, proc.p_mtxname, proc.p_contested, proc.p_priority, proc.p_usrpri, proc.p_nativepri, proc.p_nice, proc.p_rtprio, pscnt, slpque, itqueuebits, itqueues, rtqueuebits, rtqueues, queuebits, queues, idqueuebits, idqueues, switchtime, switchticks	setrunqueue, remrunqueue, mi_switch, chooseproc, schedclock, resetpriority, updatepri, maybe_resched, cpu_switch, cpu_throw, need_resched, resched_wanted, clear_resched, aston, astoff, astpending, calcru, proc_compare

□量名	□□名	□型	保□□象	依□函数
vm86pcb_lock	"vm86pcb lock"(虚□8086模式□程控制□□)	MTX_DEF	vm86pcb	vm86_bioscall
Giant	"Giant"(巨□)	MTX_DEF | MTX_RECURSE	几乎可以是任何□西	□多
callout_lock	"callout lock"(延□□用□)	MTX_SPIN | MTX_RECURSE	callfree, callwheel, nextsoftcheck, proc.p_itcallout, proc.p_slpcallout, softticks, ticks	

2.2. 共享互斥□

□些□提供基本的□/写□型的功能，可以被一个正在睡眠的□程持有。 □在它□被□一到lockmgr(9)之中。

表 2. 共享互斥□列表

□量名	保□□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□□量

原子保□□量并非由一个□在的□保□的特殊□量，而是： □□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□机制(例如mutex)就是用原子保□□量□□的， 但是很少有□量直接使用□□□理方式。

- mtx.mtx_lock

Chapter 3. 内核□象

内核□象，也就是*Kobj*，□内核提供了一□面向□象　　的C□言□程方式。被操作的数据也承□操作它的方法。□使得在不破坏二□制兼容性的前提下，某一个接口能□□/□相□的操作。

3.1. □□

□象

　数据集合-数据□□-数据分配的集合

方法

　某一□操作-函数

□

　一□或多□方法

接口

　一□或多□方法的一个□准集合

3.2. Kobj的工作流程

□者注

□一小□□段落中原作者的用□有些含混，□参考我在括号中的注□□□。

Kobj工作□，□生方法的描述。□个描述有一个唯一的□□和一个缺省函数。　　某个描述的地址被用来在一个□的方法表里唯一的□□方法。

□建一个□，就是要建立一□方法表，并将□□表□□到一个或多个函数(方法)；　　□些函数(方法)都□有方法描述。使用前，□要被□□。□□□要□□个□分配一些□存。　　在方法表中的□个方法描述都会被指派一个唯一的□□，　　除非已□被其它引用它的□在□□□指派了□□。□于□个将要被使用的方法，　　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□□□□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□□按Hash的方法□□□象的□的□存。　　如果□个方法不在□存中，函数会□□使用□的方法表。如果□个方法被□到了，　　□里的相□函数(也就是某个方法的□□代□)就会被使用。 否□，□个方法描述的缺省函数将被使用。

□些□程可被表示如下：

```
□象-□存-□
```

3.3. 使用Kobj

3.3.1. □□

```
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□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□生申明方法的□文件和代□，脚本□会生成方法□□函数。

在□个模板中如下□□□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 □制到被生成的代□文件的□部。

例如:

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

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

例如:

```
INTERFACE foo;
```

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

例如:

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

□□□METHOD用来描述一个方法。□法是: METHOD [返回□□型] [方法名] { [□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□充。它□□个方法□充上缺省函数。□法是： METHOD [返回□□型] [方法名] { [□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。 □于□个Kobj□象，一般其□部都有一些Kobj□有的数据。 METHOD定□的方法就假□□些□有数据位于□象□部； 假如□象□部没有□些□有数据，□些方法□□个□象的□□就可能出□。而STATICMETHOD定□的□象可以不受□个限制： □□描述出的方法，其操作的数据不由□个□的某个□象□例□出，而是全都由□用□个方法□的操作数(□者注:即参数)□出。 □也□于在某个□的方法表之外□用□个方法有用。

□者注

□一段的□言与原文相比□整很大。 静□方法是不依□于□象□例的方法。 参看C++□中的"静□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□

使用Kobj的第二□是建立一个□。一个□的□有名字、方法表； 假如使用了Kobj的"□象管理工具"(Object Handling Facilities)， □中□包含□象的大小。建立□□使用宏DEFINE_CLASS()。 建立方法表□，□建立一个kobj_method_t数□，用NULL□□尾。 □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}
};
```

□□被"□□"。根据□□被初始化□系□的状□， 将要用到一个静□分配的□存和"操作数表"(ops table， □者注：即"参数表")。□些操作可通□声明一个□□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□象

使用Kobj的第三□是定□□象。Kobj□象建立程序假定Kobj □有数据在一个□象的□部。如果不是如此，□当先自行分配□象， 再使用kobj_init()初始化□象中的Kobj□有数据； 其□可以使用kobj_create()分配□象， 并自□初始化□象中的Kobj□有内容。kobj_init() 也可以用来改□一个□象所使用的□。

将Kobj的数据集成到□象中要使用宏KOBJ_FIELDS。

例如

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

3.3.8. □用方法

使用Kobj的最后一部就是通□生成的函数□用□象□中的方法。 □用□，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，□用就是:

```
[返回□ = ] FOO_BAR(□象 [, 其它参数]);
```

3.3.9. 善后□理

当一个用kobj_create()不再需要被使用□， 可□□个□象□用kobj_delete()。 当一个□不再需要被使用□， 可□□个□□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□□□□境中叉分生成的□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□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□一篇文章，□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□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□

□□jail(8)的参数中有一个是IP地址。□是在网□上□□jail□的地址。 jail(8)将IP地址翻□成网□字□□序，并存入j(jail□型的□□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□□一个Internet地址，并将其□存到指定的□□体中"。inet_aton(3)□定了□□体in， 之后in中的内容再用ntohl(3)□□成主机字□□序， 并置入jail□□体的ip_number成□。

4.1.1.3. 囚禁□程

最后，用□□程序囚禁□程。□在Jail自身□成了一个被囚禁的□程， 并使用execv(3)□行用□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□所□，函数jail()被□用，参数是□□体jail中被填入数据□， 而如前所述，□些数据□又来自jail(8)的命令行参数。 最后，□行了用□指定的命令。下面我将□始□□jail在内核中的□□。

4.1.2. 相□的内核源代□

□在我□来看文件/usr/src/sys/kern/kern_jail.c。 在□里定□了jail(2)的系□□用、相□的sysctl□，□有网□函数。

4.1.2.1. sysctl□

在kern_jail.c里定□了如下sysctl□:

```
/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□程可□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□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□程可以使用System V□程□通□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□程□看系□中挂接的文件系□□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□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□程是否可以挂□或卸□□jail友好的文件系□ */
```

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

4.1.2.2. jail(2)系□□用

像所有的系□□用一□，系□□用jail(2)□有□个参数， struct thread *td和struct jail_args *uap。 td是一个指向thread□□体的指□，□指□用于描述□用jail(2)的□程。 在□个上下文中，uap指向一个□□体，□个□□体中包含了一个指向从用□□ jail.c□送□来的jail□□体的指□。 在前面我□述用□□程序□，□已□看到□一个jail□□体被作□参数□送□系□□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□制到内核内存空□中。 copyin(9)需要三个参数：要□制□内核内存空□的数据的地址 uap-jail，在内核内存空□存放数据的j， 以及数据的大小。uap-jail指向的Jail□□体被□制□内核内存空□， 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□了□一个重要的□□体型prison。 □□体prison只被用在内核空□中。 下面是prison□□体的定□。

```
/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□□用jail(2)□一个prison□□体分配一□内存， 并在jail和prison□□体之□□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_dropvnref;
pr-pr_ip = j.ip_number;
```

下面，我□将□□□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□。 要理解jail_attach(2)□我□做了什□，我□首先要理解一些背景信息。

在FreeBSD中，□个□内核可□的□程是通□其thread□□体来□□的， 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□□上是个指向□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□定的jid □用函数jail_attach()。随后jail_attach()□用函数change_root()以改□ □用□程的根目□。接下来，jail_attach()□建一个新的ucred□□体，并在成功地将prison□□体□接到□个ucred□□体后，将□个ucred□□体□接 到□用□程上。从此□起，□个□用□程就会被□□□被□禁的。 当我□以新□建的□个ucred□□体□参数□用内核路径jailed()□， 它将返回1来□明□个用□身□是和一个jail相□的。 在jail中叉分出来的所有□程的的公共祖先□程就是□个□行了jail(2)的□程， 因□正是它□用了jail(2)系□□用。当一个程序通□execve(2)而被□行□， 它将从其父□程的ucred□□体□承被□禁的属性， 因而它也会□有一个被□禁的ucred□□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□程被从其父□程叉分来的□候，　　系□□用fork(2)将用crhold()来□□其身□凭□。　　□□，很自然的就保持了子□程的身□凭□于其父□程一致，所以子□程也是被□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□□被囚禁程序的限制

在整个内核中，有一系列□被囚禁程序的□束措施。　　通常，□些□束只□被囚禁的程序有效。如果□些程序□□突破□些□束， 相□的函数将出□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□□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□□一个消息描述符的状□。
- 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□□的系□□用的代□中，都有□□一个条件判断：

```
/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□在msgget()的意□相同。
- setop(semid, array, nops): semop□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□id□□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□特殊的方式□理socket(2)系□□用和相□的低□套接字函数。 □了决定一个套接字是否允□被□建，它先□□sysctl□ security.jail.socket_unixiproute_only是否被□置□1。 如果被□□1，套接字建立□将只能指定□些□□族： PF_LOCAL, PF_INET, PF_ROUTE。否□，socket(2)将会返回出□。

```
/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□□器

Berkeley包□□器提供了一个与□□无□的，直接通向数据□路□的低□接口。 □在BPF是否可以在□禁的□境中被使用是通□devfs(8)来控制的。

4.2.4. 网□□□

网□□□TCP, UDP, IP和ICMP很常□。IP和ICMP□于同一□□□次：第二□, 网□□。当参数nam被□置□，有一些限制措施会防止被囚禁的程序□定到一些网□接口上。 nam是一个指向sockaddr□□体的指□， 描述可以□定服□的地址。一个更□切的定□：sockaddr"是一个模板，包含了地址的□□符和地址的□度"。
在函数in_pcbbind_setup()中sin是一个指向sockaddr_in□□体的指□, □个□□体包含了套接字可以□定的端口、地址、□度、□□族。 □就禁止了在jail中的□程指定不属于□个□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  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);
...
}
```

□也□想知道函数prison_ip()做什□。 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□在Jail中取消或更改文件□志，如"不可修改"、"只可添加"、"不可□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□□行机制的框架。 FreeBSD目前使用它来□行内核的□□初始化。SYSINIT使得FreeBSD的内核各子系□可以在内核或模□□□加□□接□被重整、 添加、□除、替□，□□，内核和模□加□□就不必去修改一个静□的有序初始化 安排表甚至重新□□内核。□个体系也使得内核模□ (□在称□*KLD*可以与内核不同□□□、□接、 在引□系□□加□，甚至在系□□行□加□。□些操作是通□ "内核□接器"(kernel linker)和"□接器集合" (linker set)完成的。

5.1. □□

□接器集合**(Linker Set)**

一□□接方法。□□方法将整个程序源文件中静□申明的数据收集到 一个可□近□址的数据□元中。

5.2. SYSINIT操作

SYSINIT要依□□接器□取遍布整个程序源代□多□申明的静□数据 并把它□□成一个彼此相□的数据□。□□□接方法被称□ "□接器集合"(linker set)。 SYSINIT使用□个□接器集合以□□□个数据集合， 包含□个数据条目的□用□序、函数、一个会被提交□□函数的数据指□。

SYSINIT按照□□□先□□□□函数排序以便□行。 第一□□先□的□□是子系□的□□， □出SYSINIT分□□行子系□的函数的全局□序， 定□在sys/kernel.h中的枚□ `sysinit_sub_id`内。第二□□先□□□在子系□中的元素的□序，定□在sys/kernel.h中的枚□ `sysinit_elem_order`内。

有□□□刻需要使用SYSINIT：系□□□或内核模□加□□， 系□析□或内核模□卸□□。内核子系□通常在系□□□□使用SYSINIT 的定□□以初始化数据□□。例如，□程□度子系□使用一个SYSINIT 定□□来初始化□行□列数据□□。□□□□程序□避免直接使用 `SYSINIT()`，□于□□□□上的物理真□□□□使用 `DRIVER_MODULE()`□用的函数先□□□□的存在， 如果存在，再□行□□的初始化。□一系□□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. □□

宏SYSINIT()在SYSINIT□□数据集合中　建立一个SYSINIT数据□，以便SYSINIT在系□□□或模□加□□排序　并□行其中的函数。SYSINIT()有一个参数uniquifier，　SYSINIT用它来□□数据□，随后是子系□□序号、子系□元素□序号、 待□用函数、□□□函数的数据。所有的函数必□有一个恒量指□参数。

例 *1.* SYSINIT()的例子

```
#include sys/kernel.h

void foo_null(void *unused)
{
        foo_doo();
}
SYSINIT(foo, SI_SUB_FOO, SI_ORDER_FOO, foo_null, NULL);

struct foo foo_voodoo = {
        FOO_VOODOO;
}

void foo_arg(void *vdata)
{
        struct foo *foo = (struct foo *)vdata;
        foo_data(foo);
}
SYSINIT(bar, SI_SUB_FOO, SI_ORDER_FOO, foo_arg, foo_voodoo);
```

注意，SI_SUB_FOO和SI_ORDER_FOO　□当分□在上面提到的枚□sysinit_sub_id和sysinit_elem_order之中。既可以使用已有的枚□□，　也可以将自己的枚□□添加到□□个枚□的定□之中。□可以使用数学表□式微□SYSINIT的□行□序。　以下的例子示例了一个需要□好要在内核参数□整的SYSINIT之前□行的SYSINIT。

例 *2.* □整SYSINIT()□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□数据集合。

例 3. SYSUNINIT()的例子

```
#include sys/kernel.h

void foo_cleanup(void *unused)
{
        foo_kill();
}
SYSUNINIT(foobar, 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")，□ FreeBSD □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□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□措施的□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 □□□□，内核将□□若干□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□系□的模□管理系□□用和其他系□接口，包括 boot loader □量，□策略模□□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□化，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"□数器,就可以做到□一点：一旦某个入口函数被□用，□数器的□被□加1；而□当一个入口函数□用□束□，□数器的□被□少1。 □□□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□数器的□重新□少到0□止。 □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□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□□行之前加□而且不能□□卸□的策略）□，框架不□□数器□行操作，其□□是□0，从而将此□的同□□□□到0。 □一个□端的□法是，使用一个□□□□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□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□□其□□□□的安全属性□□的存□□□提供同□保□。下列□□情形，可能□致□安全属性□□的不一致□□：第一，作□安全属性□□的持有者，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□□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□。

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□量和□数信号量。 □□些同□原□的使用必□慎重，需要特□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□□是作□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□□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□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□□的，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□用的策略集合来□，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□□的□□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□法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□□上□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□□□□的性能□失太大，策略必□就自己□　Mbuf　□行□□的要求向　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□0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□□的策略允□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□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□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□□的状□，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□□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□□的系□□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□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□段□行可能造成睡眠阻塞的存□分配操作，可以将它□放在一个□独的模□ SYSINIT() □程中集中□行。

6.7.1.2. mpo_destroy

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

参数	□明	□定
conf	MAC 策略定□	

策略加□事件。必□持有策略□表互斥□，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	□明	□定
td	□用□程	
call	策略特有的系□□用□号	
arg	系□□用参数的指□	

□入口函数提供策略□用的系□□用，□□策略模□不需要□其向用□□程提供的□一个□外服□而注册□用的系□□用。由□用程序提供的策略注册名字来□定提供其所申□服□的特定策略，所有参数将通□□入口函数□□□被□用的策略。 当□□新服□□，安全模□必□在必要□通□ MAC 框架□用相□的□□控制□□机制。 比方□，假如一个策略□□了某□□外的信号功能，那□它□□□用相□的信号□□控制□□，以接受 MAC 框架中注册的其他策略的□□。

不同的模□需要并□地手□□行copyin()拷□系□□用数据。

6.7.1.4. mpo_thread_userret

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

参数	□明	□定
td	返回□程	

使用□入口函数，策略模□能□在□程返回用□空□□（系□□用返回、□常返回等等）□行 MAC 相□的□理工作。使用□□□程□□的策略需要使用□入口函数，因□在□理系□□用的□程中，并不是在任意□刻都能申□到□程□的；□程的□□可能表示□□的□□信息、□程□史□□或者其他数据。□使用□入口函数，□□程信任状所作的修改

可能被存放在 p_label ,□域受一个□程□自旋□的保□；接下来，□置□程□的TDF_ASTPENDING □志位和□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	□明	□定
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	□明	□定
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	□明	□定
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	□明	□定
mntlabel	将被初始化的mount □□策略□□	

参数	□明	□定
fslabel	将被初始化的文件系□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	□明	□定
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	□明	□定
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	□明	□定
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	□明	□定
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	□明	□定
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	□明	□定
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	□明	□定
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	□明	□定
label	将被□□的□□	

□□一个信任状上的□□。在□入口函数中，策略□当□放所有在内部分配的与 label 相□□的存□空□，以便□□□□□。

6.7.2.16. mpo_destroy_devfsdirent_label

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

参数	□明	□定
label	将被□□的□□	

□□一个 devfs 表□上的□□。在□入口函数中，策略□当□放所有在内部分配的与 label 相□□的存□空□，以便□□□□□。

6.7.2.17. mpo_destroy_ifnet_label

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

参数	□明	□定
label	将被□□的□□	

□□与一个已□除接口相□□的□□。在□入口函数中，策略□当□放所有在内部分配的与 label 相□□的存□空□，以便□□□□□。

6.7.2.18. mpo_destroy_ipq_label

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

参数	□明	□定
label	将被□□的□□	

□□与一个 IP 分片□列相□□的□□。在□入口函数中，策略□当□放所有在内部分配的与 label 相□□的存□空□，以便□□□□□。

6.7.2.19. mpo_destroy_mbuf_label

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

参数	□明	□定
label	将被□□的□□	

□□与一个 Mbuf 相□□的□□。在□入口函数中，策略□当□放所有在内部分配的与 label 相□□的存□空□，以便□□□□□。

6.7.2.20. mpo_destroy_mount_label

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

参数	□明	□定
label	将被□□的 Mount 点□□	

□□与一个 mount 点相□□的□□。在□入口函数中，策略□当□放所有在内部分配的与 mntlabel 相□□的存□空□，以便□□□□□。

6.7.2.21. mpo_destroy_mount_label

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

参数	□明	□定
mntlabel	将被□□的 Mount 点□□	
fslabel	File system label being destroyed	

□□与一个 mount 点相□□的□□。在□入口函数中，策略□当□放所有在内部分配的，与 mntlabel 和fslabel 相□□的存□空□，以便□□□□□。

6.7.2.22. mpo_destroy_socket_label

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

参数	□明	□定
label	将被□□的套接字□□	

□□与一个套接字相□□的□□。在□入口函数中，策略□当□放所有在内部分配的，与 label 相□□的存□空□，以便□□□□□。

6.7.2.23. mpo_destroy_socket_peer_label

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

参数	□明	□定
peerlabel	将被□□的套接字□等□体□□	

□□与一个套接字相□□的□等□体□□。在□入口函数中，策略□当□放所有在内部分配的，与 label 相□□的存□空□，以便□□□□□。

6.7.2.24. mpo_destroy_pipe_label

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

参数	□明	□定
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□□的存□空□，以便□□□□□。

6.7.2.25. mpo_destroy_proc_label

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

参数	□明	□定
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□□的存□空□，以便□□□□□。

6.7.2.26. mpo_destroy_vnode_label

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

参数	□明	□定
label	□程□□	

□□一个 vnode 的□□。在□入口函数中，策略□当□放所有在内部分配的，与 label 相□□的存□空□，以便□□□□□。

6.7.2.27. mpo_copy_mbuf_label

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

参数	□明	□定
src	源□□	
dest	目□□□	

将 src 中的□□信息拷□到 dest中。

6.7.2.28. mpo_copy_pipe_label

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

参数	□明	□定
src	源□□	
dest	目□□□	

将 src 中的□□信息拷□至 dest。

6.7.2.29. mpo_copy_vnode_label

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

参数	□明	□定
src	源□□	
dest	目□□□	

将 src 中的□□信息拷□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	□明	□定
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□。

6.7.2.31. mpo_externalize_ifnet_label

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

参数	□明	□定
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□。

6.7.2.32. mpo_externalize_pipe_label

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

参数	□明	□定
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	□明	□定
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	□明	□定
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□。

6.7.2.35. mpo_externalize_vnode_label

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

参数	□明	□定
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□。

6.7.2.36. mpo_internalize_cred_label

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

参数	□明	□定
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，□建一个内部形式的□□□□。　目前的MAC方案将依次□用所有策略的相□入口函数，来□□□□的内部□□□求，　因此，□□代□必□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	□明	□定
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，□建一个内部形式的□□□□。　目前的MAC方案将依次□用所有策略的相□入口函数，来□□□□的内部□□□求，　因此，□□代□必□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	□明	□定
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，□建一个内部形式的□□□□。 目前的MAC方案将依次□用所有策略的相□入口函数，来□□□□的内部□□□求， 因此，□□代□必□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	□明	□定
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，□建一个内部形式的□□□□。 目前的MAC方案将依次□用所有策略的相□入口函数，来□□□□的内部□□□求， 因此，□□代□必□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_internalize_vnode_label (struct label *label,
                                         char *element_name,
                                         char *element_data,
                                         int *claimed);
```

参数	□明	□定
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，□建一个内部形式的□□□□。 目前的MAC方案将依次□用所有策略的相□入口函数，来□□□□的内部□□□求， 因此，□□代□必□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□象的□□□行操作。策略模□将感□趣的被□□内核□象的相□生命周期事件 注册在恰当的入口点上。□象的初始化、□建和□□事件均提供了□子点。在某些□象上□可以□□重新□□，即，允□用□□程改□□象上的□□□。 □某些□象可以□□其特定的□象事件，比如与 IP 重□相□的□□事件。一个典型的被□□□象在其生命周期中将□有下列入口函数：

```
□□初始化                    o
（□象相□的等待）             \
□□□建                       o
                               \
重新□□事件，                    o----.
各□□象相□的，                  |     |
□□控制事件                      ~----o
                                      \
□□□□                                  o
```

使用□□初始化入口函数，策略可以以一□□一的、与□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□□的策略可能并不需要□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□象相□□（内核□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□及的其他□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□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□失□，所以无法□告内存分配失□。

□象特有的事件一般不会引□其他的□□事件，但是在□象上下文□生改□□，策略使用它□可以□相□□□□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□□可能会因□□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□，之后内核才可以重用或者□放□象的内核数据□□。

除了与特定内核□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□□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	□明	□定
fslabel	文件系□□□	
vp	将被□□的 vnode	
vlabel	与 vp 相□□的策略□□	

从文件系□□展属性中□取 vp 的□□。成功，返回 0。 不成功，□在 errno 指定的相□的□□□□。 如果文件系□不支持□展属性的□取操作，□可以考□将 fslabel 拷□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	□明	□定
mp	文件系□挂□点	
fslabel	文件系□□□	
vp	将被□□的 vnode	
vlabel	与 vp 相□□的策略□□	

在非多重□□文件系□上，使用□入口函数，根据文件系□□□，fslabel， □ vp □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	□明	□定
dev	devfs_dirent □□的□□	
devfs_dirent	将被□□的 Devfs 目□□	
label	将被填写的 devfs_dirent □□	

□□入□□新建的 devfs_dirent 填写□□。□函数将在□□文件系□加□、重□或添加新□□□被□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	□明	□定
dirname	新建目□的名字	
namelen	字符串 dirname 的□度	
devfs_dirent	新建目□在 Devfs 中□□的目□□	

□□入目□参数的新建 devfs_dirent 填写□□。□函数将在加□、重□□□文件系□，或者添加一个需要指定目□□□的新□□□被□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	□明	□定
cred	主体信任状	
mp	devfs 挂□点	
dd	□接目□	
ddlabel	与 dd 相□□的□□	
de	符号□接□	
delabel	与 de 相□□的策略□□	

□新近□建的 devfs(5) 符号□接□填写□□（delabel）。

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	□明	□定
cred	主体信任状	
mount	文件系□挂□点	
label	文件系□□□	
dvp	父目□ vnode	
dlabel	与 dvp 相□□的策略□□	
vp	新□建的 vnode	
vlabel	与 vp 相□□的策略□□	
cnp	vp中的子域名字	

将 vp 的□□写入文件□展属性。成功，将□□填入 vlabel， 并返回 0。否□，返回□□的□□□□。

6.7.3.1.8. mpo_create_mount

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

参数	□明	□定
cred	主体信任状	
mp	客体；将被挂□的文件系□	
mntlabel	将被填写的 mp 的策略□□	
fslabel	将被挂□到 mp 的文件系□的策略□□。	

□□入的主体信任状所□建的挂□点填写□□。□函数将在文件系□挂□□被□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	□明	□定
□ 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□□，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	□明	□定
cred	主体信任状	
vp	将被重新□□的 vnode	
vnodelabel	vp □有的策略□□	
newlabel	将取代vnodelabel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	□明	□定
cred	主体信任状	
vp	写出□□所□□的 vnode	
vlabel	vp的策略□□	
intlabel	将被写入磁□的□□	

将参数 intlabel □出的□□信息写入指定 vnode 的□展属性。 □函数被 vop_stdcreatevnode_ea 所□用。

6.7.3.1.12. mpo_update_devfsdirent

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

参数	□明	□定
devfs_dirent	客体；devfs 目□□	
direntlabel	将被更新的devfs_dirent的策略□□	
vp	父 vnode	已□定
vnodelabel	vp的策略□□	

根据所□入的 devfs vnode □□，□ devfs_dirent 的□□□行更新。 重新□□一个 devfs vnode 的操作成功之后，将□用□函数来□□□□的改□，如此，即使相□的 vnode 数据□□被内核回收重用， 也不会□失□□的新状□。□外，在 devfs 中新建一个符号□接□，□接着mac_vnode_create_from_vnode， 也将□用□函数，□ vnode □□□行初始化操作。

6.7.3.2. IPC □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	□明	□定
socket	套接字	套接字□定 WIP
socketlabel	socket 的策略□□	
m	客体；mbuf	
mbuflabel	将被填写的 m 的策略□□	

根据□入的套接字□□□新□建的mbuf□部□置□□。 □当套接字□生一个新的数据□或者消息，并将其存□在参数 mbuf 中□，将□用□函数。

6.7.3.2.2. mpo_create_pipe

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

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

根据□入的主体信任状参数，□置新建管道的□□。□当一个新管道被□建，□函数将被□用。

6.7.3.2.3. mpo_create_socket

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

参数	□明	□定
cred	主体信任状	不可改□
so	客体；将被□□的套接字	
socketlabel	将被填写的 so 的□□	

根据□入的主体信任状参数，□置新建套接字的□□。□当新建一个套接字，□函数将被□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	□明	□定
oldsocket	□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，□置□□。

6.7.3.2.5. mpo_relabel_pipe

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

参数	□明	□定
cred	主体信任状	
pipe	管道	
oldlabel	pipe 的当前策略□□	
newlabel	将□pipe □置的新的策略□□	

□pipe□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	□明	□定
cred	主体信任状	不可改□
so	客体；套接字	
oldlabel	so 的当前□□	
newlabel	将□socket □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	□明	□定
mbuf	从套接字接收到的第一个数据□	

参数	□明	□定
mbuflabel	mbuf 的□□	
oldlabel	套接字的当前□□	
newlabel	将□套接字填写的策略□□	

根据□入的 mbuf □□，□置某个 stream 套接字的□等□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□□，□函数将被□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	□明	□定
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□入的□程套接字端点，□一个 stream UNIX 与套接字□置□等□□。 □当相□的套接字□之□□行□接□，□函数将在□端分□被□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	□明	□定
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□入的主体信任状参数，□新建的 BPF 描述子□置□□。 当□程打□ BPF □□□点□，□函数将被□用。

6.7.3.3.2. mpo_create_ifnet

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

参数	□明	□定
ifnet	网□接口	
ifnetlabel	将□ifnet填写的策略□□	

□新建的网□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	□明	□定
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	□明	□定
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□□列，□□□重□完□的 IP 数据□□置□□。

6.7.3.3.5. mpo_create_fragment

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

参数	□明	□定
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□□的 mbuf □部信息，□其新建的分片的 mbuf □部□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	□明	□定
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□置□□。在□多条件下将会□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	□明	□定
ifnet	网□接口	
ifnetlabel	ifnet 的策略□□	
mbuf	新建数据□的 mbuf □部	
mbuflabel	将□mbuf填写的策略□□	

□在□定接口上由于某个□路□□□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	□明	□定
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	□明	□定
ifnet	网□接口	
ifnetlabel	ifnetlabel 的策略□□	
mbuf	新建数据□的 mbuf □部	
mbuflabel	将□mbuf填写的策略□□	

□从网□接口参数□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	□明	□定
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	□明	□定
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	□明	□定
fragment	IP 数据□分片	
fragmentlabel	fragment 的策略□□	
ipq	IP 分片重□□列	
ipqlabel	ipq 的策略□□	

根据所□入的 IP 分片重□□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□□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□□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	□明	□定
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□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	□明	□定
mbuf	IP 分片	
mbuflabel	mbuf 的策略□□	
ipq	IP 分片重□□列	
ipqlabel	将被更新的ipq的当前策略□□	

根据所□入的 IP 分片 mbuf □部（mbuf）□接收 它的 IP 分片重□□列（ipq）的□□□行更新。

6.7.3.4. □程□□事件操作

6.7.3.4.1. mpo_create_cred

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

参数	□明	□定
parent_cred	父主体信任状	
child_cred	子主体信任状	

根据所□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) □，将□用此函数。 □函数不□与□程□制（forking）或者□建事件混□一□。

6.7.3.4.2. mpo_execve_transition

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

参数	□明	□定
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
vnodelabel	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□程□求□行vnode文件，而通□ 入口函数mpo_execve_will_transition 有成功返回的策略□，将□用□函数。策略模□可以通□□入□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	□明	□定
old	在□行execve(2)之前的主体信任状	不可改□
vp	将被□行的文件	
vnodelabel	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	□明	□定
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	□明	□定
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	□明	□定
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□及的其他任何□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□□□□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□□□□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	□明	□定
bpf_d	主体；BPF 描述子	
bpflabel	bpf_d 的策略□□	
ifnet	客体；网□接□	
ifnetlabel	ifnet 的策略□□	

决定 MAC 框架是否□□允□将由参数接□接收到的数据□□□□由 BPF 描述子所□□的□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□的情况。

6.7.4.2. mpo_check_kenv_dump

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

参数	□明	□定
cred	主体信任状	

决定相□主体是否□□被允□□□内核□境状□（参考 kenv(2)）。

6.7.4.3. mpo_check_kenv_get

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

参数	□明	□定
cred	主体信任状	
name	内核的□境□量名字	

决定相□主体是否可以□□内核中□定□境□量的状□。

6.7.4.4. mpo_check_kenv_set

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

参数	□明	□定
cred	主体信任状	
name	内核的□境□量名字	

决定相□主体是否有□□置□定内核□境□量的□。

6.7.4.5. mpo_check_kenv_unset

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

参数	□明	□定
cred	主体信任状	

参数	□明	□定
name	内核的□境□量名字Kernel environment variable name	

决定相□主体是否有□清除□定的内核□境□量的□置。

6.7.4.6. mpo_check_kld_load

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

参数	□明	□定
cred	主体信任状	
vp	内核模□的 vnode	
vlabel	vp的策略□□	

决定相□主体是否有□加□□定的模□文件。

6.7.4.7. mpo_check_kld_stat

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

参数	□明	□定
cred	主体信任状	

决定相□主体是否有□□□内核的加□模□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	□明	□定
cred	主体信任状	

决定相□主体是否有□卸□一个内核模□。

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	□明	□定
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	□明	□定
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	□明	□定
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	□明	□定
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	□明	□定
cred	主体信任状	
socket	将被□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	□明	□定
cred	主体信任状	
socket	将被□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）□定到地址 sockaddr。成功，返回0，否□返回一个□□□□errno。 建□采用的□□□□有：EACCES，用于□□不符的情况；EPERM，用于特□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	□明	□定
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□信息。

6.7.4.18. mpo_check_socket_send

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

参数	□明	□定
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	□明	□定
u1	主体信任状	
u2	□象信任状	

□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□ sysctl，以及通□procfs 的状□□□操作。

6.7.4.20. mpo_check_socket_visible

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

参数	□明	□定
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	□明	□定
cred	主体信任状	
ifnet	客体；网□接口	
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□□更新参数□□定的网□接口的□□□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	□明	□定
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	□明	□定
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	□明	□定
cred	主体信任状	不可改□
vp	客体；vnode	已被□定
vnodelabel	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	□明	□定
cred	主体信任状	
mp	客体；文件系□挂□	
mountlabel	mp的策略□□	

□定相□主体信任状是否有□□看在□定文件系□上□行 statfs 的□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□列表中□□排除□个文件系□□，比如， □用 getfsstat(2)□。

6.7.4.26. mpo_check_proc_debug

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

参数	□明	□定
cred	主体信任状	不可改□
proc	客体；□程	

□定相□主体信任状是否有□ debug □定□程。 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□□目□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	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	
flags	access(2) □志	

根据相□主体信任状决定其□□定 vnode 以□定□□□志□行的 access(2) 和其他相□□用的返回□。一般，□采用与mpo_check_vnode_open 相同的□□来□□□函数。 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□。

6.7.4.28. mpo_check_vnode_chdir

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

参数	□明	□定
cred	主体信任状	
dvp	客体；chdir(2) 的目的 vnode	

参数	□明	□定
dlabel	dvp的策略□□	

□定相□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	□明	□定
cred	主体信任状	
dvp	目□ vnode	
dlabel	与dvp相□□的策略□□	

□定相□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	□明	□定
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	□明	□定
cred	主体信任状	
dvp	父目□ vnode	
dlabel	dvp的策略□□	
vp	客体；将被□除的 vnode	
label	vp的策略□□	
cnp	vp中的成□名	

□定相□主体信任状是否有□从□定的父目□中，□除□定名字的 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□。 使用 unlink(2) 和 rmdir(2)，将□致□函数被□用。 提供□入口函数的策略□必□□□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	□明	□定
cred	主体信任状	不可改□
vp	客体；vnode	被□定
	label	vp的策略□□
	type	ACL □型

□定相□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□限不□。

6.7.4.33. mpo_check_vnode_exec

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

参数	□明	□定
cred	主体信任状	
vp	客体；将被□行的 vnode	
label	vp的策略□□	

□定相□主体信任状是否有□□行□定 vnode。 □于□行特□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	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	
type	ACL □型	

□定相□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	□明	□定
cred	主体信任状	

参数	□明	□定
vp	客体；vnode	
label	vp的策略□□	
attrnamespace	□展属性名字空□	
name	□展属性名	
uio	I/O □□指□；参□ uio(9)	

□定相□主体信任状是否有□□□□定 vnode 上□定名字空□和名字的□展属性。 使用□展属性□□□□存□的策略模□可能会需要□□些□展属性的操作□行特殊□理。 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□。

6.7.4.36. mpo_check_vnode_link

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

参数	□明	□定
cred	主体信任状	
dvp	目□ vnode	
dlabel	与dvp相□□的策略□□	
vp	□接目的 vnode	
label	与vp相□□的策略□□	
cnp	将被□建的□接□□的成□名	

□定相□主体是否有□□参数vp□定的 vnode □建一个由参数cnp□定名字的□接。

6.7.4.37. mpo_check_vnode_mmap

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

参数	□明	□定
cred	主体信任状	

参数	□明	□定
vp	将被映射的 vnode	
label	与vp相□□的策略□□	
prot	mmap 保□ (参□ mmap(2))	

□定相□主体是否有□将□定 vnode vp 以 prot指定的保□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	□明	□定
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	□明	□定
cred	主体信任状	
vp	映射的 vnode	
prot	存□保□	

□定相□主体是否有□将□定 vnode vp 映射内存空□的存□保□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	□明	□定
active_cred	主体信任状	
file_cred	与struct file相□□的信任状	
vp	将被□行 poll 操作的 vnode	
label	与vp相□□的策略□□	

□定相□主体是否有□□□定 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	□明	□定
cred	主体信任状	
dvp	目□ vnode	
dlabel	与dvp相□□的策略□□	
vp	将被重命名的 vnode	
label	与vp相□□的策略□□	
cnp	vp中的成□名	

□定相□主体是否有□重命名□定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	□明	□定
cred	主体信任状	
dvp	目□ vnode	
dlabel	与dvp相□□的策略□□	
vp	被覆□的 vnode	
label	与vp相□□的策略□□	
samedir	布□型□量；如果源和目的目□是相同的，□被置□1	
cnp	目□component名	

□定相□主体是否有□重命名□定 vnode vp，至指定目□ dvp，或更名□cnp。如果无需覆□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	□明	□定
cred	主体信任状	
socket	客体；套接字	
socketlabel	socket的策略□□	

□定相□主体是否有□□听□定套接字。 成功，□返回0；否□，返回□□□□□errno。 建□使用的□□□□：EACCES，用于□□不匹配； EPERM，用于□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	□明	□定
cred	主体信任状	
dvp	客体；vnode	
dlabel	dvp的策略□□	
cnp	被□□的成□名	

□定相□主体信任状是否有□在□定的目□ vnode 中□□□□定名字□行lookup操作。 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□。

6.7.4.45. mpo_check_vnode_open

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

参数	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	
acc_mode	open(2) □□模式	

□定相□主体信任状是否有□在□定 vnode 上以□定的□□模式□行 open 操作。 如果成功，□返回 0；否□，返回一个□□□□。 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	□明	□定
cred	主体信任状	

参数	□明	□定
dvp	客体；目□ vnode	
dlabel	dvp的策略□□	

□定相□主体信任状是否有□在□定的目□ vnode 上□行 readdir 操作。 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□。

6.7.4.47. mpo_check_vnode_readlink

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

参数	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	

□定相□主体信任状是否有□在□定符号□接 vnode 上□行 readlink 操作。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□。 □函数可能在若干□境下被□用，包括由用□□程□式□行的 readlink □用， 或者是在□程□行名字□□□□式□行的 readlink 。

6.7.4.48. mpo_check_vnode_revoke

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

参数	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	

□定相□主体信任状是否有□撤□□定 vnode 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□。

6.7.4.49. mpo_check_vnode_setacl

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

参数	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	
type	ACL □型	
acl	ACL	

□定相□主体信任状是否有□□置□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□限不□。

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	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	
attrnamespace	□展属性名字空□	
name	□展属性名	
uio	I/O □□指□；参□ uio(9)	

□定相□主体信任状是否有□□置□定 vnode 上□定名字空□中□定名字的□展属性的□。 使用□展属性□□安全□□的策略模□可能需要□其使用的属性□施□外的保□。□外， 由于在□□和□□操作□□可能存在的□争， 策略模□□□避免根据来自uio中的数据做出决策。 如果正在□行一个□除操作，□参数 uio 的□也可能□ NULL。 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□。

6.7.4.51. mpo_check_vnode_setflags

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

参数	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	
flags	文件□志；参□ chflags(2)	

□定相□主体信任状是否有□□□定的 vnode □置□定的□志。 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□。

6.7.4.52. mpo_check_vnode_setmode

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

参数	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	
mode	文件模式；参□ chmod(2)	

□定相□主体信任状是否有□将□定 vnode 的模式□置□□定□。 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□。

6.7.4.53. mpo_check_vnode_setowner

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

参数	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	
uid	用□ID	
gid	□ID	

□定相□主体信任状是否有□将□定 vnode 的文件 uid 和文件 gid □置□□定□。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	□明	□定
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)	

□定相□主体信任状是否有□将□定 vnode 的□□□□□□□置□□定□。 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	□明	□定
cred	主体信任状	
proc	客体；□程	

□定相□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	□明	□定
cred	主体信任状	
proc	客体；□程	
signal	信号；参□ kill(2)	

□定相□主体信任状是否有□向□定□程□送□定信号。 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□。

6.7.4.57. mpo_check_vnode_stat

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

参数	□明	□定
cred	主体信任状	
vp	客体；vnode	
label	vp的策略□□	

□定相□主体信任状是否有□在□定 vnode 上□行 stat 操作。 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□。

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	□明	□定
cred	主体信任状	
ifnet	网□接□	
ifnetlabel	ifnet的策略□□	
mbuf	客体；将被□送的 mbuf	
mbuflabel	mbuf的策略□□	

□定相□网□接□是否有□□送□定的 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	□明	□定
cred	主体信任状	
ifnet	网□接□	
ifnetlabel	ifnet的策略□□	
mbuf	客体；将被□送的 mbuf	
mbuflabel	mbuf的策略□□	

□定相□套接字是否有□从□定的 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	□明	□定
cred	主体信任状	不可改□
so	客体；套接字	
socketlabel	so的策略□□	

□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□□，□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	

□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	

□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	□明	□定
cred	主体信任状	

□定相□用□是否有□□置系□□□。

6.7.4.65. mpo_check_system_swapon

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

参数	□明	□定
cred	主体信任状	
vp	swap□□	
vlabel	与vp相□□的□□	

□定相□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	□明	□定
cred	主体信任状	
name	参□ sysctl(3)	
namelen		
old		
oldlenp		

参数	□明	□定
inkernel	布□型□量；如果从内核被□用，其□被置□1	
new	参□ sysctl(3)	

□定相□主体是否□□被允□□行指定的 sysctl(3) 事□。

6.7.5. □□管理□用

当用□□程□求□某个□象的□□□行修改□，将引□重新□□事件。□□的更新操作分□□□行： 首先，□行□□控制□□，□□此次更新操作是有效且被允□的；然后，□用□一个独立的入口函数□□□□行修改。 重新□□入口函数通常接收由□求□程提交的□象、□象□□指□和□求新□□，作□□入参数。 □象重新□□操作的失□将由先期的□□□□□告，所以，不允□在接下来的□□修改□程中□告失□，故而不提倡在此□程中新分配内存。

6.8. □用□体系□□

TrustedBSD MAC 框架包含了一□策略无□的□成元素，包括管理抽象□□的 MAC 接口□， □系□信任状管理体系的修改， □用□分配 MAC □□提供支持的 login □函数， 以及若干□□□□和更新内核□象(□程、文件和网□接口等)安全□□的工具。 不久，将有更多□于□用□体系□□的□□信息被包含□来。

6.8.1. 策略无□的□□管理 API

TrustedBSD MAC 提供的大量□函数和系□□用，允□□用程序使用一□□一的、策略无□的接口来□理□象的 MAC □□。 如此，□用程序可以□松管理各□策略的□□，无需□□加□某个特定策略的支持而重新□□。□多通用工具，比如 ifconfig(8)，ls(1) 和 ps(1)，使用□些策略无□的接口□□网□□□、文件和□程的□□信息。 □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□□□□可参考 mac(3).

□用程序□理的 MAC □□有□□存在形式：内部形式，用来返回和□置□程和□象的□□（mac_t）； 基于 C 字符串的外部形式，作□□□在配置文件中的存放形式，用于向用□□示或者由用□□入。 □一个 MAC □□由一□□□元素□成，其中□个元素是一个形如（名字，□）的二元□。 内核中的□个策略模□分□被指定一个特定的名字，由它□□□□中与□名字□□的□采用其策略特有的方式□行解析。 采用外部形式表示的□□，其□□元素表示□名字 / □，元素之□以逗号分隔。 □用程序可以使用 MAC 框架提供的 API 将一个安全□□在内部形式和文本形式之□□行□□。 □当向内核□□某个□象的安全□□□，内部形式的□□必□□□所需的元素集合作好内部□□存□准□。 □此，通常采用下面□□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) 配置文件中加□缺省元素集合的某个系□□用。在□象□□□置缺省□□，将允□□用程序在不□定 系□是否采用相□策略的情况下，也能向用□返回与□象相□□的有意□的安全□□。

目前的 MAC □不支持直接修改内部形式的□□元素，所有的修改必□按照下列的□□□行：将内部形式的□□□□成文本字符串，□字符串□行□□，最后将其□□成内部形式□□。如果□用程序的作者□明□□有需要， 可以在将来的版本中加入□内部形式□□□行直接修改的接口。

6.8.2. □用□指定□□

用□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□登□□□相□□的 MAC 安全□□。 当 LOGIN_SETALL 被□置，或者当 LOGIN_SETMAC 被明□指定□，□些安全□□将和其他用

□上下文参数一起被□置。

可以□期，在今后的某个版本中，FreeBSD 将把 MAC □□从 login.conf 的用□□□数据□中抽出，□其□□一个独立的数据□。 不□在此前后，setusercontext(3) API□□保持不□。

6.9. 小□

TrustedBSD MAC 框架使得内核模□能以一□集中的方式，完善系□的安全策略。 它□既可利用□有的内核□象属性，又能使用由 MAC 框架□助□□的安全□□数据，来□施□□控制。 框架提供的□活性使得□□人□可以在其上□□各□策略，如利用 BSD □有的信任状（credential） 与文件保□机制的策略，以及信息流安全策略（如 MLS 和 Biba）。 □□新安全服□的策略□程人□，可以参考本文□，以了解□有安全模□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□□的□□体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□可以有一个引用□数，可以被一个忙□数□定。 虚□内存系□也□□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□□的真□内存□，CPU会□生一个"□□"， 通知操作系□与磁□等□□□行交□，□□址□□入存□内容， 写□址□写出存□内容。□个"□□" 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□的后□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□存信息□□体-vm_object_t

FreeBSD□□了□一的"虚□内存□象"(VM□象)的□□思想。 VM□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□使用相同的VM□象管理核内数据-文件的□存， 所以□些□存的□□也是□一的。

VM□象可以被影□制(shadowed)。 它□可以被堆放到其它□□VM□象堆□的□端。例如，可以有一个交□VM□象， 放置在文件VM□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□)，以派生出地址空□。

□当注意，一个vm_page_t □□体在任一个□刻只能与一个VM□象相□□。 VM□象影□本可以□□跨□例的共享相同的□。

7.3. 文件系□□入/□出-buf□□体

vnode VM□象，比如文件VM□象，一般需要□□它□自己的清理(clean)/ 未清理(dirty)信息，而不依□于文件系□的清理/未清理□□。 例如，当VM系□要同□一个物理□和其□□的□□存□器， VM系□就需要在写入到□□存□器前将□□□□□已清理。 □外，文件系□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□体就是□所周知的文件系□□存， struct buf或bp。 当文件系□需要□一个VM□象的一部分操作□， 它常会将□个□象的□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□象的各部分映射到buf□□体中， 然后□buf□□体□行□入/□出操作。 下□的vm_page_t在□入/□出期□通常被□□□"忙"。 文件系□□存也会"忙"，□□于文件系□□□程序非常有用， □文件系□□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□□体□□是被清理□的。 □些内核虚□内存□用来存放映射， 并不限制□存数据的能力。□格的□，物理数据□存是 vm_page_t的一个功能，不是文件系□□存的功能。 然而，由于文件系□□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□□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□□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□□体，□□就□□了跨区域(board)的□存的□一。

7.5. KVM存□映射

FreeBSD使用KVM存放各□各□的内核□□体。在KVM中最大的□个□体是文件系□□存。 那是与struct buf□体有□的映射。

不像Linux，FreeBSD不将所有的物理内存映射到KVM中。 □意味着FreeBSD可以在32位平台上管理超□4GB的内存配置。事□上， 如果mmu(□者注：可能是指"内存管理□元"，"Memory Management Unit") 有足□的能力，FreeBSD理□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，□只能是一个争□点。

有几□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□，再将大□切分□恒定大小的小□， 以便按照某一□□型的□□体分配。□可以使用命令vmstat -m 一□当前KVM分区使用情况。

7.6. □整FreeBSD的虚□内存系□

□□者的□同努力使得FreeBSD可以自行□□□整内核。一般来□， 除了内核配置□□maxusers和NMBCLUSTERS，□不需要做任何□乱的事情。□些内核□□配置□□(一般)被指定在 /usr/src/sys/i386/conf/CONFIG_FILE之中。所有可用内核配置□□的描述可在 /usr/src/sys/i386/conf/LINT中□到。

在一个大系□的配置中，□可能需要□加maxusers的□。 数□□□通常在10到128。注意，□度□加maxusers 的□可能□致系□从□□可用的KVM中溢出，从而引起无法□知的操作。 最好将maxusers□□一个合理的数□，并且添加其它□□， 如NMBCLUSTERS，来□加特定的□源。

如果□的系□要被重□荷的使用网□，□需要□加 NMBCLUSTERS的□。数□□□通常在1024到4096。

NBUF也是□□的□□系□的参数。 □个参数决定系□可用来映射文件系□□入/□出□存的KVM的数量。 注意：□个参数与□一的□存没有任何□系。□个参数可在3.0-CURRENT 和以后的内核中被□□的□□，通常不□当被手□的□□。 我□推□□不要指定NBUF。 □系□自行□定它。太小的□会□致非常低效的文件系□操作； 太大的□会使用□□列中缺少□面，而大量的□□于在□状□。

缺省情况下，FreeBSD内核□□□是不被□化的。 □可以在内核配置文件中用makeoptions 指定排□(debugging)和□化□志。注意，□一般不□使用-g， 除非□能□□付由此□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□行□□整内核的方式。□通常不需要指定任何sysctl□量， 尤其是与VM相□的那些□量。

□行□VM和系□□整的影□相□直接一些。 首先，□当尽可能在UFS/FFS文件系□上使用Soft Updates。在/usr/src/sys/ufs/ffs/README.softupdates 里有□于如何配置的指示。

其次，□当配置足□多的交□空□。 □□当在□个物理磁□上配置一个交□分区，最多4个， 甚至在□的"工作"磁□上。□□当有至少2倍于主内存的交□空□； 假如□没有足□内存的□，交□分区□□更多。 □也□当按照□期望中的最大内存配置决定交□分区的大小， □□以后就不再需要重新□磁□分区了。 如果□□理系□崩□后的内存□倒(crash dump)， 第一个交□分区必□至少与主内存一□大， /var/crash必□有足□的空□来承装□倒。

NFS上的交□分区可以很好的被4.X或后来的系□使用， 但是□必□明白NFS服□器将要□受□装□操作很□的冲□。

Chapter 8. SMPng □□文□

8.1. □□

□□文□□目前 SMPng 架□的□□与□□□行了介□。 它首先介□了基本的原□和相□工具， 其后是□于 FreeBSD 内核的同□与□行模型， 接下来□□了具体系□中的□策略， 并描述了在各个子系□中引入□粒度的同□和□□并行化的□□， 最后是□□的□□□明， 用以解□最初做出某些□□决策的□机， 并使□者了解使用特定的原□所可能□生的重大影□。

□□文□仍在撰写当中， 并将不断更新以反映与 SMPng □目有□的最新□□与□□的情况。 其中有□多小□目前□只是提□， 但我□会逐□□其充□内容。 □于□□文□的更新和建□， □□□文□□□。

SMPng 的目□是使内核能□并□□行。 基本上， 内核是一个很大而□□的程序。 要□内核能□多□程地□行， 我□需要使用某些其它多□程程序在□□□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□量(condition variable)。 如果希望了解它□以及其它 SMP □□， □参□本文的 □□表 一□。

8.2. 基本工具与上□的基□知□

8.2.1. 原子操作指令和内存□

□于内存□和原子操作指令已□有很多介□材料， 因此□一□并不打算□其□行□尽的介□。 □而言之， 如果有□某一□量上写□， 就不能在不□得相□的□□□其□行□取操作。 也就是□， 内存□的作用在于保□内存操作的相□□序，但并不保□内存操作的□格□序。 □言之， 内存□并不保□ CPU 将本地快取□存或存□□冲的内容刷写回内存，而是在□□放□□保其所保□的数据， □于能看到□□放的那个□的 CPU 或□□可□。 持有内存□的 CPU 可以在其快取□存或存□□冲中将数据保持其所希望的、 任意□的□□， 但如果其它 CPU 在同一数据元上□行原子操作，□第一个 CPU 必□保□， 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□在一□□模型中， □□在主存 (或某一全局快取□存) 中的数据是可□的， 当某一 CPU 上触□原子操作□， 其它 CPU 的存□□冲和快取□存就必□□同一快取□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□的内存□元□就需要特□小心。 例如， 在□□ sleep mutex □， 我□就必□使用 `atomic_cmpset` 而不是 `atomic_set` 来打□ MTX_CONTESTED 位。 □□做的原因是， 我□需要把 `mtx_lock` 的□□到某个□量， 并据此□行决策。 然而， 我□□到的□可能是□□的， 也可能在我□□行决策的□程中□生□化。因此， 当□行 `atomic_set` □， 最□可能会□□一□□行置位， 而不是我□□行决策的那一个。 □就必□通□ `atomic_cmpset` 来保□只有在我□的决策依据是最新的□， 才□相□的□量□行置位。

最后， 原子操作只允□一次更新或□一个内存□元。 需要原子地更新多个□元□， 就必□使用□来代替它了。例如， 如果需要更新□个相互□□的□数器□， 就必□使用□， 而不是□次□独的原子操作了。

8.2.2. □□与写□

□□并不需要像写□那□□。 □□□□型的□， 都需要□保通□它□□□的不是□□的数据。 然而， 只有写操作必□是排他的， 而多个□程□可以安全地□同一□量的□。 使用不同□型的□用于□和写操作有□多各自不同的□□方式。

第一□方法是用 sx □， 它可以用于□□写□使用的排他□， 而□□□作□共享□。 □□方法十分□□明了。

第二□方法□略□晦□。 可以用多个□来保□同一数据元。 □□， 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□方式□□数据□却可能非常有用。 例如， 父□程指□是同□受 `proctree_lock` sx □和□程 mutex 保□的。 在只希望□□已□□程的父□程□， 用 proc □更□方便。 但是， 其它一些地方， 例如 `inferior` □□需要通□父指□在□程□上□行搜索， 并□□个□程上□的地方就不能□□做了， 否□， 将无法保□在□我□所□得的□果□行操作□， 之前□□□的状况依旧有效。

8.2.3. 上□状□和□果

如果□需要使用□来保持所□□□量的状□， 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， 在所做□□引□的□作□束之前， 必□□□保持上□状□。

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□程上下文， □□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□□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□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□□做能□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□而需要□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□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， □□度□行□程式□理程序。

在□一模式中□当地采用□量□上下文切□可能是非常□□的。 因□我□可能会希望在未来改□□一模式， 因此□在最好的方案， □□是□□推□在□量□上下文切□之上的工作， 以便□一□完善中断□理架□， 随后再考察□量□上下文切□是否□用。

8.3.2. 内核□占与□界区

8.3.2.1. 内核□占□介

内核□占的概念很□□， 其基本思想是 CPU □□□行□先□最高的工作。 当然， 至少在理想情况下是□□。 有些□候， □成□一理想的代价会十分高昂， 以至于在□些情况下□占会得不□失。

□□完全的内核□占十分□□： 在□度将要□行的□程并放入□行□列□， □□它的□先□是否高于目前正在□行的□程。 如果是□□的□， □行一次上下文切□并立即□始□行□□程。

尽管□能□在□占□保□多数数据， 但内核并不是可以安全地□□□占的。 例如， 如果持有自旋 mutex 的□程被□占， 而新□程也□□□得同一自旋 mutex， 新□程就可能一直自旋下去， 因□被中断的□程可能永□没有机会□行了。 此外， 某些代□， 例如在 Alpha 上的 `exec` □□程地址空□□号□行□□的代□也不能被□断， 因□它被用来支持□□的上下文切□操作。 在□些代□段中， 会通□使用□界区来□□禁用□占。

8.3.2.2. □界区

□界区 API 的□任是避免在□界区内□生上下文切□。 □于完全□占式内核而言， 除了当前□程之外的其它□程的□个 `setrunqueue` 都是□断点。 `critical_enter` 的一□□□方式是□置一□程私有□□， 并由其□□方清除。 如果□用 `setrunqueue` □□置了□个□志， □无□新□程和当前□程相比其□先□高低， 都不会□生□占。 然而， 由于□界区会在自旋 mutex 中用于避免上下文切□， 而且能□同□□得多个自旋 mutex， 因此□界区 API 必□支持嵌套。 由于□个原因， 目前的□□中采用了嵌套□数， 而不□□是□个的□程□志。

□了尽可能□短□□□□， 在□界区中的□占被推□， 而不是直接□□。 如果□程□被□断， 并被置□可□行， 而当前□程□于□界区， □会□置一□程私有□志， 表示有一个尚未□行的□断操作。 当最外□□界区退出□， 会□□□一□志， 如果它被置位， □当前□程会被□断， 以允□更高□先□的□程□始□行。

中断会引□一个和自旋 mutex 有□的□□。 如果低□中断□理程序需要□， 它就不能中断任何需要□□的代□， 以避免可能□生的□坏数据□□的情况。 目前，□一机制是透□□界区 API 以 `cpu_critical_enter` 和 `cpu_critical_exit` 函数的形式□□的。 目前□一 API 会在所有 FreeBSD 所支持的平台上禁用和重新□用中断。 □□方法并不是最□的， 但它更易理解， 也更容易正□地□□。 理□上， □一□助 API 只需要配合在主中断上下文中的自旋 mutex 使用。 然而， □了□代□更□□□， 它被用在了全部自旋 mutex， 甚至包括所有□界区上。 将其从 MI API 中剥□出来放入 MD API， 并只在需要使用它的 MI API 的自旋 mutex □□中使用可能会有更好的效果。 如果我□最□采用了□□□□方式， □ MD API 可能需要改名， 以彰□其□一□独 API □一事□。

8.3.2.3. □□折衷

如前面提到的， 当完全□占并非□能提供最佳性能□， 采取了一些折衷的措施。

第一□折衷是， □占代□并不考□其它 CPU 的存在。 假□我□有□个 CPU， A 和 B， 其中 A 上□程的□先□□ 4， 而 B 上□程的□先□是 2。 如果 CPU B 令一□先□□ 1 的□程□入可□行状□， □理□上， 我□希望 CPU A 切□至□一新□程， □□就有□个□先□最高的□程在□行了。 然而， □定□个 CPU 在□占□更合□， 并通□ IPI 向那个 CPU □出信号， 并完成相□的同□工作的代价十分高昂。 因此， 目前的代□会□制 CPU B 切□至更高□先□的□程。

□注意□□做仍会□系□□入更好的状□， 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□□整快取□存以配合新□程的□行。 当内核返回到被□断的□程□， 它又需要重新填充之前□失的快取□存信息。 此外， 如果内核能□将□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□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 □件中断的一部分来□行。 事件将基于所希望的□□□□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□□体， 它通常作□内核中以□程□□向的□□控制的依据。 BSD-派生的系□采用一□ "写□□制" 的模型来□理凭据数据： 同一凭据□□体可能存在多个引用， 如果需要□其□行修改， □□个□□体将被□制、 修改， 然后替□□引用。 由于在打□□用于□□□□控制的凭据快取□存广泛存在， □□做法会□大地□省内存。 在□移到□粒度的 SMP □， □一模型也省去了大量的□操作， 因□只有未共享的凭据才能□施修改， 因而避免了在使用共享凭据□□外的同□操作。

凭据□□体只有一个引用□， 被□□是可□的； 不允□改□共享的凭据□□体， 否□将可能□致□生□□条件。 cr_mtxp mutex 用于保□ struct ucred 的引用□数， 以□□其一致性。 使用凭据□□体□， 必□在使用□程中保持有效的引用， 否□它就可能在□个不合理的消□者使用□程中被□放。

struct ucred mutex 是一□叶 mutex， 出于性能考□， 它通□ mutex 池□□。

由于多用于□□控制决策， 凭据通常情况下是以只□方式□□的， 此□一般□使用 td_ucred， 因□它不需要上□。 当更新□程凭据□， □□和更新□程中必□持有 proc □。 □□和更新操作必□使用 p_ucred， 以避免□□□和使用□的□□条件。

如果所□系□□用将在更新□程凭据之后□行□□控制□□， □ td_ucred 也必□刷新□当前□程的□。 □□做能□避免修改后使用□□的凭据。 内核会自□在□程□入内核□， 将□程□□体的 td_ucred 指□刷新□□程的 p_ucred， 以保□内核□□控制能用到新的凭据。

8.4.2. 文件描述符和文件描述符表

□□内容将在□后□加。

8.4.3. Jail □□体

struct prison 保存了用于□□那些通□ jail(2) API □建的 jail 所用到的管理信息。 □包括 jail 的主机名、 IP 地址， 以及一些相□的□置。 □个□□体包含引用□数， 因□指向□一□□体□例的指□会在多□凭据□□之□共享。 用了一个 mutex， pr_mtx 来保□□引用□数以及所有 jail □□体中可□□量的□写□□。 有一些□量只会在□建 jail 的□刻□生□化， 只需持有有效的 struct prison 就可以□始□□些□了。 □于□个□目具体的上□操作的文□， 可以在 sys/jail.h 的注□中□到。

8.4.4. MAC 框架

TrustedBSD MAC 框架会以 struct label 的形式□□一系列内核□象的数据。 一般来□， 内核中的 label (□□) 是由与其□□的内核□象同□的□保□的。 例如， struct vnode 上的 v_label □□是由其所在 vnode 上的 vnode □保□的。

除了嵌入到□准内核□象中的□□之外， MAC 框架也需要□□一□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□。 由于能□同□并行地□行□多□□控制□□， □策略表的只□□□， 在□□忙□数□， 框架的入口□需要首先持有□个 mutex。 MAC 入口操作的□程中并不需要□□□持有此 mutex — 有些操作， 例如文件系□□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□， 需要持有此 mutex， 而且要求引用□数□零， 以避免在用表□□其□行修改。

□于需要等待表□入□置状□的□程， 提供了一个条件□量 mac_policy_list_not_busy， 但□一条件□量只能在□用者没有持有其它□□才能使用， 否□可能会引□□逆序□□。 忙□数在整个框架中事□上□扮演了某□形式的 共享/排他 □的作用： 与 sx □不同的地方在于， 等待列表□入□置状□的□程可以□死， 而不是允□忙□数和其它在 MAC 框架入口 (或内部) 的□之□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 (□) □ (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□□根据需要上□。 有些□目不需上□， 因□□些□目在全程是只□的， (例如 device_get_softc(9))， 因而并不会□生□□条件。 □□ newbus 数据□□的修改相□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_sigiolst， 而□□程□□是 pg_sigiolst。 □些表由相□的□程或□程□□保□。 除了用以将 struct sigio □接到□程□上的 sio_pgsigio 字段之外， 在 struct sigio 中的多数字段在注册□程中都是不□量。 一般而言， □□人□在□□新的支持 SIGIO 的内核□象□， 会希望避免在□用 SIGIO 支持函数， 例如 fsetown 或 funsetown 持有□□体□， 以免去需要在□□体□和全局 SIGIO □之□定□□序。 通常可以通□提高□□体上的引用□数来□到□□的目的， 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□理。

8.4.12. Sysctl

sysctl MIB 服□会从内核内部， 以及用□□的□用程序以系□□用的方式触□。 □会引□至少□个和□有□的□□：其一是□□持命名空□的数据□□的保□， 其二是与那些通□ sysctl 接口□□的内核□量和函数之□的交互。 由于 sysctl 允□直接□出（甚至修改）内核□□数据以及配置参数， sysctl 机制必□知道□些□量相□的上□□□。 目前， sysctl 使用一个全局 sx □来□□□ sysctl 操作的串行化； 然而， □些是假定用全局□保□的，并且没有提供其它保□机制。 □一□的其余部分将□□介□上□和 sysctl 相□□□的□□。

- 需要将 sysctl 更新□所□行的操作的□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 □理程序采用第二□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□到 SYSCTL_FOO 宏和□□体中。 □□多数 sysctl 都是有效的。 □于使用 sx □、 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□的上□。

8.4.13. 任□□列 (Taskqueue)

任□□列 (taskqueue) 的接口包括□个与之□□的用于保□相□数据的□。 taskqueue_queues_mutex 是用于保□ taskqueue_queues TAILQ 的□。 与□个系□□□的□一个 mutex □是位于 struct taskqueue □□体上。 在此□使用同□原□的目的在于保□ struct taskqueue 中数据的完整性。 □注意的是， 并没有□独的、 □助用□□其自身的工作□行□的□化用的宏， 因□□些□基本上不会在 kern/subr_taskqueue.c 以外的地方用到。

8.5. □□□明

8.5.1. 休眠□列

休眠□列是一□用于保存同□一个等待通道 (wait channel) 上休眠□程列表的数据□□。 在等待通道上， □个□于非睡眠状□的□程都会携□一个休眠□列□□。 当□程在等待通道上□生阻塞□， 它会将休眠□列□□体送□那个等待通道。 与等待通道□□的休眠□列□保存在一个散列表中。

休眠□列散列表中保存了包含至少一个阻塞□程的等待通道上的休眠□列。 □个散列表上的□称作 sleepqueue (休眠□列) □。 它包含了一个休眠□列的□表， 以及一个自旋 mutex。 此□的自旋 mutex 用于保□休眠□列表，以及其上休眠□列□□的内容。 一个等待通道上只会□□一个休眠□列。 如果有多个□程在同一等待通道上阻塞，□休眠□列中将□□除第一个□程之外的全部□程。 当从休眠□列中□除□程□， 如果它不是唯一的阻塞的休眠□程，□会□得主休眠□列的空□表上的休眠□列□□。 最后一个□程会在恢□□行□□得主休眠□列。 由于□程有可能以和加入休眠□列不同的次序从其中□除， 因此， □程□□□列□可能会携□与其□入□不同的休眠□列。

sleepq_lock 函数会□住指定等待通道上休眠□列□的自旋 mutex。 sleepq_lookup 函数会在主休眠□列散列表中□□□定的等待通道。 如果没有□到主休眠□列， 它会返回 NULL。 sleepq_release 函数会□□定等待通道所□□的自旋 mutex □行解□。

将□程加入休眠□列是通□ sleepq_add 来完成的。 □个函数的参数包括等待通道、 指向保□等待通道的 mutex 的指□、 等待消息描述串， 以及一个□志掩□。 □用此函数之前， □通□ sleepq_lock □休眠□列□上□。 如果等待通道不是通□ mutex 保□的 (或者它由全局□保□)， □□将 mutex 指□□置□ NULL。 而 flags (□志) 参数□包括了一个□型字段， 用以表示□程即将加入到的休眠□列的□型， 以及休眠是否是可中断的 (SLEEPQ_INTERRUPTIBLE)。 目前只有□□□型的休眠□列： 通□ msleep 和 wakeup 函数管理的□□休眠□列 (SLEEPQ_MSLEEP)， 以及基于条件□量的休眠□列 (SLEEPQ_CONDVAR)。 休眠□列□型和□指□□□个参数完全是用于内部的断言□□。 □用 sleepq_add 的代□， □明示地在□□的 sleepqueue □透□ sleepq_lock □行上□之后， 并使用等待函数在休眠□列上阻塞之前解□所有用于保□等待通道的 interlock。

通□使用 sleepq_set_timeout 可以□休眠□置超□。 □个函数的参数包括等待通道， 以及以相□□□□□数□□位的超□□□。 如果休眠□被某个到来的信号打断， □□□□用 sleepq_catch_signals 函数， □个函数唯一的参数就是等待通道。 如果此□程已□有未决信号， □ sleepq_catch_signals 将返回信号□号； 其它情况下， 其返回□□是 0。

一旦将□程加入到休眠□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□个取决于□用□是否希望允□使用超□、 收到信号， 或用□□□程□度器打断休眠状□。 其中， sleepq_wait 函数□□地等待， 直到当前□程通□某个□醒 (wakeup) 函数□式地恢□□行； sleepq_timedwait 函数□等待， 直到当前□程被□式地□醒， 或者□到早前使用 sleepq_set_timeout □置的超□； sleepq_wait_sig 函数会等待□式地□醒， 或者其休眠被中断； 而 sleepq_timedwait_sig 函数□等待□式地□醒、 □到用 sleepq_set_timeout □置的超□， 或□程的休眠被中断□三□条件之一。 所有□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□□， 表示之前□用 sleepq_catch_signals □是否有□□未决信号。

如果□程被□式地恢□□行， 或其休眠被信号□止， □等待函数会返回零， 表示休眠成功。 如果□程的休眠被超□或用□□□程□度器打断， □会返回相□的 errno 数□。 需要注意的是， 因□ sleepq_wait 只能返回 0， 因此□用者不能指望它返回什□有用信息， 而□假定它完成了一次成功的休眠。 同□， 如果□程的休眠□□超□， 并同□被□止， □ sleepq_timedwait_sig 将返回一个表示□生超□的□□代□。 如果返回□□代□是 0 而且使用 sleepq_wait_sig 或 sleepq_timedwait_sig 来□行阻塞， □□□用 sleepq_calc_signal_retval 来□□是否有未决信号， 并据此□□合□的返回□。 □早前□用 sleepq_catch_signals 得到的信号□号， □作□参数□□ sleepq_calc_signal_retval。

在同一休眠通道上休眠的□程， 可以由 sleepq_broadcast 或 sleepq_signal 函数来□式地□醒。 □□个函数的参数均包括希望□醒的等待通道、 将□醒□程的□先□ (priority) 提高到多少， 以及一个□志 (flags) 参数表示将要恢□□行的休眠□列□型。 □先□参数将作□最低□先□， 如果将恢□的□程的□先□比此参数更高 (数□更低) □其□先□不会□整。 □志参数主要用于函数内部的断言， 用以□□休眠□列没有被当做□□的□型□待。 例如， 条件□量函数不□恢□□□休眠□列的□行。 sleepq_broadcast 函数将恢□所有指定休眠通道上的阻塞□程， 而 sleepq_signal □只恢□在等待通道上□先□最高的阻塞□程。 在□用□些函数之前， □首先使用 sleepq_lock □休眠□列上□。

休眠□程也可以通□□用 sleepq_abort 函数来中断其休眠状□。 □个函数只有在持有 sched_lock □才能□用， 而且□程必□□于休眠□列之上。 □程也可以通□使用 sleepq_remove 函数从指定的休眠□列中□除。 □个函数包括□个参数， 即休眠通道和□程， 它只在□程□于指定休眠通道的休眠□列之上□才将其□醒。 如果□程不在那个休眠□列之上， 或同□□于□一等待通道的休眠□列上， □□个函数将什□都不做而直接返回。

8.5.2. 十字□□ (turnstile)

- 与休眠□列的比□和不同。
- □□/等待/□放 (lookup/wait/release) - 介□ TDF_TSNOBLOCK □□条件。

- □先□□播。

8.5.3. □于 mutex □□的一些□□

- 我□是否□要求 mtx_destroy() 持有 mutex， 因□无法安全地断言它□没有被其它□象持有？

8.5.3.1. 自旋 mutex

- 使用一□界区...

8.5.3.2. 休眠 mutex

- 描述 mutex 冲突□的□□条件
- □何在持有十字□□□□□， 可以安全地□冲突 mutex 的 mtx_lock。

8.5.4. Witness

- 它能做什□
- 它如何工作

8.6. 其它□□

8.6.1. 中断源和 ICU 抽象

- struct isrc
- pic □□

8.6.2. 其它□□/□□

- 是否□将 interlock □□ `sema_wait`？
- 是否□提供非休眠式 sx □？
- □加一些□于正□使用引用□数的介□。

□□表

原子

当遵循□当的□□□□□， 如果一操作的效果□其它所有 CPU 均可□， □称其□原子操作。 狭□的原子操作是机器直接提供的。 就更高的抽象□次而言， 如果□□体的多个成□由一个□保□， □如果□它□的操作都是在上□后、 解□前□行的， 也可以称其□原子操作。

阻塞

□程等待□、 □源或条件□被阻塞。 □一□□也因此被□予了太多的意涵。

□界区

不允□□生□占的代□段。 使用 critical_enter(9) API 来表示□入和退出□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□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。

部分 II: □□□□程序

□写 FreeBSD □□□□程序

.1. □介

本章□要介□了如何□FreeBSD□写□□□□程序。□□□□在 □儿的上下文中多用于指代系□中硬件相□的□西，如磁□，打印机， □形□式器及其□□。□□□□程序是操作系□中用于控制特定□□的 □件□件。也有所□的□□□，即□□□□程序用□件模□□□的行□， 而没有特定的底□硬件。□□□□程序可以被静□地□□□系□，或者 通□□□内核□接工具'kld'在需要□加□。

□UNIX®操作系□中的大多数□□都是通□□□□点来□□的，有□也 被称□特殊文件。□些文件在文件系□的□次□□中通常位于 /dev目□下。在FreeBSD 5.0-RELEASE以前的 □行版中， □devfs(5)的支持□没有被集成到FreeBSD中，□个□□ □点必□要静□□建，并且独立于相□□□□□程序的存在。系□中大 多数□□□点是通□□行MAKEDEV□建的。

□□□□程序可以粗略地分□□□，字符和网□□□□□程序。

.2. □□内核□接工具-KLD

kld接口允□系□管理□从□行的系□中□□地添加和□除功能。 □允□□□□□程序的□写者将他□的新改□加□到□行的内核中， 而不用□了□□新改□而□繁地重□。

kld接口通□下面的特□命令使用：

- kldload - 加□新内核模□
- kldunload - 卸□内核模□
- kldstat - 列□当前加□的模□

内核模□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□□ □附加到内核的□西。

```
SRCS=skeleton.c
KMOD=skeleton

.include bsd.kmod.mk
```

□□地用□个makefile□行make就能□□建文件 skeleton.ko，□入如下命令可以把它加□到内核：

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

.3. □□□□□□程序

UNIX® 提供了一套公共的系□□用供用□的□用程序使用。当用□□□ □□□点□，内核的上□将□些□用分□到相□的□□□□程序。脚本 /dev/MAKEDEV□□的系□生成了大多数的□□□点， 但如果□正在□□□自己的□□程序，可能需要用 mknod□建□自己的□□□点。

.3.1. □建静□□□□点

mknod命令需要四个参数来□建□□□点。□必□指定□□□点的名字，□□的□型，□□的主号□和□□的从号□。

.3.2. □□□□□点

□□文件系□，或者□devfs，在全局文件系□名字空□中提供□ 内核□□名字空□的□□。□消除了由于有□□□□程序而没有静□ □□□点，或者有□□□点而没有安装□□□□程序而□来的潜在□□。 Devfs仍在□展中，但已□能□工作得相当好了。

.4. 字符□□

字符□□□□程序直接从用□□程□□数据，或□□数据到用□□程。 □是最普通的一□□□□□程序，源□□中有大量的□□例子。

□个□□的□□□例子会□住□写□它的任何□，并且当□□取它的□候 会将□些□返回□□。下面□示了□个版本，一个□用于FreeBSD 4.X， 一个□用于FreeBSD 5.X。

例 4. □用于FreeBSD 4.X的回□□□□□□□程序□例

```
/*
 * □□'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;
```

```
/* □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");

/*
 * □个函数被kld[un]load(2)系□□用来□用,
 * 以决定加□和卸□模□□需要采取的□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;

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

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

/*
 * 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;

    /*
     * □个□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) {
        uprintf("uiomove failed!\n");
    }
    return(err);
}

/*
 * echo_write接受一个字符串并将它保存到□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-iov_base, echomsg-msg,
        MIN(uio-uio_iov-iov_len, BUFFERSIZE - 1));

    /* □在需要以null□束字符串，并□□□度 */
    *(echomsg-msg + MIN(uio-uio_iov-iov_len, BUFFERSIZE - 1)) = 0;
    echomsg-len = MIN(uio-uio_iov-iov_len, BUFFERSIZE);

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

```
    }
    count++;
    return(err);
}

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

例 5. □用于FreeBSD 5.X回□□□□□□□程序□例

```
/*
 * □□'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");
```

```
/*
 * □个函数被kld[un]load(2)系□□用来□用,
 * 以决定加□和卸□模□□需要采取的□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;

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

static int
echo_close(struct cdev *dev, int fflag, int devtype, struct thread *p)
{
    uprintf("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□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-iov_base, echomsg-msg,
        MIN(uio-uio_iov-iov_len, BUFFERSIZE - 1));

    /* □在需要以null□束字符串，并□□□度 */
    *(echomsg-msg + MIN(uio-uio_iov-iov_len, BUFFERSIZE - 1)) = 0;
    echomsg-len = MIN(uio-uio_iov-iov_len, BUFFERSIZE);

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

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

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□建一个□点：

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

□□程序被加□后，□□□能□□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□一□□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□□的□个□□混□一□，从而致使 相□内核代□□大地□□化，作□推□磁□I/O基□□□□代化的一部分，FreeBSD 抛□了□□□冲的磁□□□的支持。

.6. 网□□□□□□程序

□□网□□□的□□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□□□□□程序

9.1. 概述

本章介□了□写ISA□□□□程序相□的一些□□。□儿展示的□代□ 相当□□，很容易□人□想到真正的代□，不□□依然□□是□代□。 它避免了与所□□的主□无□的□□。真□的例子可以在□□□□程序的 源代□中□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□的□西。

□□子系□是以面向□象的方式□□的，其主要□□通□相□□的 方法函数来□□。

ISA□□程序□□的□□方法的列表与任何其他□□的很相似。 □于名字□"xxx"的假想□□程序，它□将是：

- `static void xxx_isa_identify (driver_t *, device_t);` 通常用于□□□□程序而不是□□□□程序。 但□于ISA□□，□个方法有特殊用途：如果□□提供某些□□特定的 （非PnP）方法自□□□□□，□个例程可以□□它。
- `static int xxx_isa_probe (device_t dev);` 在已知（或PnP）位置探□□□。□于已□部分配置的 □□，□个例程也能□提供□□特定的□某些参数的自□□□。
- `static int xxx_isa_attach (device_t dev);` 挂接和初始化□□。
- `static int xxx_isa_detach (device_t dev);` 卸□□□□□模□前解挂□□。
- `static int xxx_isa_shutdown (device_t dev);` 系□□□前□行□□的□□。
- `static int xxx_isa_suspend (device_t dev);` 系□□入□能状□前挂起□□。也可以中止 切□到□能状□。
- `static int xxx_isa_resume (device_t dev);` 从□能状□返回后恢□□□的活□状□。

`xxx_isa_probe()`和 `xxx_isa_attach()`是必□提供的，其余例程根据□□的 需要可以有□□地□□。

使用下面一□描述符将□□□□□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□束 */
    };
```

9.3. Device_t指□

Device_t是□□□□□而定□的指□□型， □里我□只□心从□□□□程序□写者的角度看感□趣的方法。下面的方法用来操□□□□□中的□：

- device_t device_get_parent(dev) □取□□的父□□。
- driver_t device_get_driver(dev) □取指向其□□程序□□的指□。
- char *device_get_name(dev) □取□□程序的名字，在我□的 例子中□"xxx"。
- int device_get_unit(dev) □取□元号（与□个□□程序□□的□□从0□始□号）。
- char *device_get_nameunit(dev) □取□□名，包括□元号。 例如"xxx0"， "xxx1" 等。
- char *device_get_desc(dev) □取□□描述。通常它以人可□的 形式描述□□的□切型号。
- device_set_desc(dev, desc)□置描述信息。□使得□□描述指向desc字符串， 此后□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□□分配的 □冲区，□□desc字符串在以后可以被改□而不会□生有害的□果。
- void *device_get_softc(dev) □取指向与□□□□的□□描述符 （xxx_softc□□）的指□。
- u_int32_t device_get_flags(dev) □取配置文件中特定于□□的 □志。

可以使用一个很方便的函数device_printf(dev, fmt, …)从□□□□程序中打印□息。它自□在□息前添加□元名和冒号。

device_t的□些方法在文件kern/bus_subr.c 中□□。

9.4. 配置文件与自□配置期□□□和探□的□序

ISA□□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□被□□成与□□□□的□源□。根据□□ □自□配置需要和支持程度的不同，□些□是可□的。例如，某些□□根本不需要□DRQ，而有些□允□□□从□□配置端口□取 IRQ□置。如果机器有多个ISA□□，可以在配置文件中明□指定□条 □□，如isa0或isa1， 否□将在所有ISA□□上搜索□□。

敏感(sensitive)是一□□源□求，它指示 必□在所有非敏感□□之前探□□□。此特性□被支持，但似乎从未在目前的任何□□程序中使用□。

□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□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□□□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□□□PnP□□□入睡眠，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□□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□□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□用并返回失□。

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□中不再予以 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。

□于自□□□的□□（包括PnP和□□特定的）的□一个□践□□是， 不能从内核配置文件中向它□□□旗□。因此它□必□要□根本不使用 旗□，要□□所有自□□□的□□使用□元号□0的□□的旗□，或者 使用sysctl接口而不是旗□。

通□使用函数族resource_query_*()和 resource_*_value()直接□□配置□源，从而可以提供其他不常用的配置。它□的□□位于 kern/subr_bus.c。老的IDE磁□□□器 i386/isa/wd.c包含□□使用的例子。 但必□□先使用配置的□准方法。将解析配置□源□□事情留□ □□配置代□。

9.5. □源

用□写入到内核配置文件中的信息被作□配置□源□理，并□□到内核。 □□配置代□解析□部分信息并将其□□□□□device_t的□和与之 □□的□□□源。□于□□情况下的配置，□□程序可以直接使用 resource_* 函数□□配置□源。 然而， 通常既不需要也不推□□□做， 因此□儿不再□一□□□□个□□。

□□□源与□个□□相□□。通□□型和□型中的数字□□它□。 □于ISA□□，定□了下面的□型：

- *SYS_RES_IRQ* - 中断号
- *SYS_RES_DRQ* - ISA DMA通道号
- *SYS_RES_MEMORY* - 映射到系□内存空□ 的□□内存的□□
- *SYS_RES_IOPORT* - □□I/O寄存器的□□

□型内的枚□从0□始，因此如果□□有□个内存区域，它的 SYS_RES_MEMORY □型的□源□号□0和1。 □源□型与C□言的□型无□， 所有□源□具有C□言 unsigned long □型，并且必要□必□□行□型□制□□ (cast)。□源号不必□□，尽管□于ISA它□一般是□□的。ISA□□允□的□源□号□：

```
IRQ: 0-1
DRQ: 0-1
MEMORY: 0-3
IOPORT: 0-7
```

所有□源被表示□□有起始□和□数的□□。□于IRQ和DRQ□源， □数一般等于1。内存的□引用物理地址。

□□源能□□行三□□型的□作：

- set/get
- allocate/release
- activate/deactivate

Set□置□源使用的□□。Allocation保留出□求的□□，使得 其它□□不能再占用（并□□此□□没有被其它□□占用）。Activation□行必要的□作使得□□程序可以□□□源（例如，□于 内存，它将被映射到内核的虚□地址空□）。

操作□源的函数有：

- int bus_set_resource(device_t dev, int type, int rid, u_long start, u_long count)

  □□源□置□□。成功□返回0，否□返回□□□。 一般此函数只有在type， rid，start或 count之一的□超出了允□的□□才会 返回□□。

- dev - □□程序的□□
- type - □源□型，SYS_RES_*
- rid - □型内部的□源号（ID）
- start, count - □源□□

- `int bus_get_resource(device_t dev, int type, int rid, u_long *startp, u_long *countp)`

  取得□源□□。成功□返回0，如果□源尚未定□□返回□□□。

- `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start或count。出□的情况下返回0，因此如果0是□源的start合法□之一，将无法区分返回 的0是否指示□□。幸□的是，□于附加□□程序，没有ISA□源的 start□从0□始。

- `void bus_delete_resource(device_t dev, int type, int rid)`

  □除□源，令其未定□。

- `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□没有被其它□□占用的地方按count□的□□分配一个□源。不□，不支持□□。如果□源尚未被□置，□自□□建它。start□0，end□~0（全1）的□□特殊□意味着必□使用以前通□`bus_set_resource()`□置的固定□：start和count就是它□自己，end=(start+count)，□□情况下，如果以前□源没有定□，□返回□□。尽管rid通□引用□□，但它并不被ISA□□的□源分配代□□置（其它□□可能使用不同的 方法并可能修改它）。

旗□是一个位映射，□用者感□趣的有：

- *RF_ACTIVE* - 使得□源分配后 被自□激活。
- *RF_SHAREABLE* - □源可以同□ 被多个□□程序共享。
- *RF_TIMESHARE* - □源可以被多个□□ 程序分□共享，也就是□，被多个□□程序同□分配，但任何 □定□□只能被其中一个激活。

出□返回0。被分配的□可以使用 `rhand_*()`从返回的句柄□得。

- `int bus_release_resource(device_t dev, int type, int rid, struct resource *r)`
- □放□源，r□`bus_alloc_resource()` 返回的句柄。成功□返回0，否□返回□□□。
- `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□源。成功□返回0，否□返回□□□。如果 □源被分□共享且当前被□一□□程序激活，□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)`
- □□/分□中断□理程序与□□。成功□返回0，否□ 返回□□□。
- r - 被激活的描述IRQ的□源句柄。

  flags - 中断□先□，如下之一：

- INTR_TYPE_TTY - □端和其它 □似的字符□型□□。使用 spltty()屏蔽它□。
- (INTR_TYPE_TTY | INTR_TYPE_FAST) - □入□冲□小的□端□型 □□，而且□入上的数据□失很□□（例如老式串口）。 使用spltty()屏蔽它□。
- INTR_TYPE_BIO - □□型□□， 不包括CAM控制器上的。使用 splbio()屏蔽它□。
- INTR_TYPE_CAM - CAM （通用□□ 方法Common Access Method）□□控制器。使用 splcam()屏蔽它□。
- INTR_TYPE_NET - 网□接口 控制器。使用 splimp()屏蔽它□。
- INTR_TYPE_MISC - 各□其它□□。除了通□ splhigh()没有其它方法屏蔽它□。 splhigh()屏蔽所有中断。

当中断□理程序□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□理程序的指□， □型driver_intr_t被定□□void driver_intr_t(void *)
- *arg* - □□□□理程序的参量，□□ 特定□□。由□理程序将它从void*□□□任何□□□型。ISA 中断□理程序的旧□定是使用□元号作□参量，新□定（推□） 使用指向□□softc□□的指□。
- *cookie[p]* - 从 setup()接收的□，当□□□ teardown() □用于□□□理程序。

定□了若干方法来操作□源句柄(struct resource *)。□□□□ 程序□写者感□趣的有：

- u_long rman_get_start(r) u_long rman_get_end(r) 取得被分配的□源□□的起始和□束。
- void *rman_get_virtual(r) 取得 被激活的内存□源的虚地址。

9.6. □□内存映射

很多情况下□□□□程序和□□之□的数据交□是通□内存 □行的。有□□可能的□体：

(a) 内存位于□□□上

(b) 内存□□算机的主内存

情况(a)中，□□程序可能需要在□上的内存与主存之□来回 拷□数据。□了将□上的内存映射到内核的虚地址空□，□上内存的 物理地址和□度必□被定□□SYS_RES_MEMORY□源。 然后□源就可以被分配并激活，它的虚地址通□使用 rman_get_virtual()□取。□老的□□程序 将函数pmap_mapdev()用于此目的，□在 不□当再直接使用此函数。它已成□□源激活的一个内部□□。

大多数ISA□的内存配置□物理地位于640KB-1MB□□之□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）。□□情况下，如果机器有 比□□内存的起始地址更多的内存（□句□□，它□重□），□ 必□在被□□使用的内存起始地址□配置一个内存空洞。□多 BIOS允□在起始于14MB或15MB□配置1M的内存空洞。如果BIOS 正□地□告内存空洞，FreeBSD就能□正□□理它□（此特性 在老BIOS上可能会出□□）。

情况(b)中，只是数据的地址被□送到□□，□□使用DMA□□ □□主存中的数据。存在□个限制：首先，ISA□只能□□16MB以下 的内存。其次，虚地址空□中□□的□面在物理地址空□中可能不 □□，□□可能不得不□行分散/收集操作。□□子系□□□些□□ 提供□成□成的解决□法，剩下的必□由□□程序自己完成。

DMA内存分配使用了□个□□， bus_dma_tag_t 和 bus_dmamap_t。 □□（tag）描述了DMA内存要求的特性。映射（map）表示按照□些 特性分配的内存□。多个映射可以与同一□□□□。

□□按照□特性的□承而被□□成□型□次□□。子□□□承父 □□的所有要求，可以令其更□格，但不允□放□要求。

一般地，□个□□□元□建一个□□□□（没有父□□）。如果 □个□□需要不同要求的内存区，□□□个内存区都会□建一个□□，□些 □□作□父□□的孩子。

使用□□□建映射的方法有□□。

其一，分配一大□符合□□要求的□□内存（以后可以被□放）。 □一般用于分配□了与□□通信而存在相□□□□的那些内存区。 将□□的内存加□到映射中非常容易：它□是被看作位于□当物理 内存□□的一整□。

其二，将虚□内存中的任意区域加□到映射中。□片内存的 □一□都被□□，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□□面(bounce page)"，用作中□存□。当从不符合的原始□面写入数据□，数据首先被拷□到反□□面， 然后从反□□面□□到□□。当□取□，数据将会从□□到反□□面， 然后被拷□到它□不符合的原始□面。原始和反□□面之□的拷□ □理被称作同□。□一般用于□次□□的基□之上：□次□□□ 加□□冲区，完成□□，卸□□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□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□ □□。其□切含□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□，更高的地址不允□ 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□面的物理地址被□□□□□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□返回非零□。
  - *maxsize* - 通□此□□可以分配的 最大内存□（以字□□）。有□□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□□。 nsetments 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□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□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□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□□□。

  - 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□□冲区到映射中(映射必□事先由 `bus_dmamap_create()`或者 `bus_dmamem_alloc()`)□建。□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□□冲区是受限的， 因此如果需要的反□□冲区不能立即□得，□将□求入□，当反□ □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* - □冲区的内核虚地址
  - *buflen* - □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□□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□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□冲区与其反□□面□行同□。 此函数完成原始□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□□□冲区□行同□。

- *dmat* - □□
- *map* - 已加□的映射
- *op* - 要□行的同□操作的□型：
- BUS_DMASYNC_PREREAD - 从□□到 □冲区的□操作之前
- BUS_DMASYNC_POSTREAD - 从□□到 □冲区的□操作之后
- BUS_DMASYNC_PREWRITE - 从□冲区到 □□的写操作之前
- BUS_DMASYNC_POSTWRITE - 从□冲区到 □□的写操作之后

当前PREREAD和POSTWRITE□空操作，但将来可能会改□，因此□□程序 中不能忽略它□。由bus_dmamem_alloc() □得的内存不需要同□。

从bus_dmamap_load()中□用回□函数之前， 段数□是存□在□中的。并且是按□□允□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□，和 其它必□留出来的空□）。由于数□基于最大数目而分配，因此□个□ 必□不能□置成超出□□需要。幸□的是，□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□理具有非常多 分散/收集段的□冲区，□□当一部分一部分地□理：加□□冲区的 一部分，□□到□□，然后加□□冲区的下一部分，如此反□。

□一个□践□□是段数目可能限制□冲区的大小。如果□冲区中的 所有□面□巧物理上不□□，□分片情况下支持的最大□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□保支持的最大□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□□冲区大小，□解决□法是：如果无法容□下原始 □冲区，就在□□程序中分配一个□□的□冲区作□中□存□。

下面是当使用映射□的典型□用□序，根据□映射的具体使用而不同。 字符-用于□示□□流。

□于从□接到分□□□，□期□位置一直不□的□冲区：

bus_dmamem_alloc - bus_dmamap_load - ...use buffer... - - bus_dmamap_unload - bus_dmamem_free

□于从□□程序外部□□□去，并且□常□化的□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□冲区的地址和大小必□和bus_dmamem_alloc()中使用的一□。□□情况下就 可以保□整个□冲区被作□一个段而映射（因而回□可以基于此假□），并且□求被立即□行（永□不会返回EINPROGRESS）。□□情况下回□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□□相□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□看起来有点□，也比□□□，但那是正□的使用方法。□□□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□界限制允□的□）是一个很好的主意。

当加□任意□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□行同□。代□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（例如字符上的写求）， 需要在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□个函数通常在□接□□□□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□□冲区。□求的□冲区大小不能超□ 64KB。以后，如果□□□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□ □冲区。如果□□□是使用符合上述条件的□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□ □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□□□冲区是否□□的且在ISA内存□□之内，如果不是□自□使用反□□冲区。如果需要反□□冲区， 但反□□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□□冲区的情况下，数据将被自□拷□到反□ □冲区。

- flags - 位掩□，决定将要完成的操作的□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□冲区，并在□□□束后 自□重新初始化它自己，再次重□□□同一□冲区（当然，□□ 程序可能□起□□的□一个□□之前改□□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□□冲区□使用B_RAW才有意□。
- addr - □冲区的虚地址
- nbytes - □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□□冲区的□操作， □将数据从反□□冲区拷□到原始□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□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□□□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□新的高□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□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□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;
```

□于不支持PnP的□□程序不需要特殊□理，因□□□程序会□□空的 PnP ID表，且如果在PnP□上□用会得到ENXIO。

探□例程通常至少需要某些最少量的□源，如I/O端口号，来□□并探□□。 □于不同的硬件，□□程序可能会自□□□其他必需的□源。PnP□□的 所有□源由PnP子系□□先□置，因此□□程序不需要自己□□它□。

通常□□□□所需要的最少信息就是端口号。然后某些□□允□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□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□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□□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□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 □明□什□在函数probe()中完成更好：如果 □□探□会□一些其他敏感□□□□。探□例程按旗□ sensitive排序：敏感□□首先被探□，然后是 其他□□。但identify()例程在所有探□之前 被□用，因此它□不会考□敏感□□并可能□乱□些□□。

□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□于具体□□的方式□□ */
        if(error = xxx_probe_ports(sc))
            goto bad; /* 返回前□放□源 */
```

依□于所□□□□的□切型号，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 的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□，□□当□先使用□个。一个□□的例子可能是□□的：

```
        /* 首先□□□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，□理棘手情况。所有□源□当在返回前被□放。我□利用□□一个 事□：softc□□在□□□我□以前被零化，因此我□能□□出是否分配了 某些□源：如果分配□□些□源的描述符非零。

```
        bad:

        xxx_free_resources(sc);
        if(error)
              return error;
        else /* □切□□未知 */
            return ENXIO;
```

□是完整的探□例程。□源的□放从多个地方完成，因此将它□到一个 函数中，看起来可能像下面的□子：

```
static void
           xxx_free_resources(sc)
               struct xxx_softc *sc;
           {
               /* □□□个□源，如果非0□□放 */

               /* 中断□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□□的内存 */
               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，□放其他的映射和□□ ...

               if(sc-parent_tag) {
                   bus_dma_tag_destroy(sc-parent_tag);
                   sc-parent_tag = 0;
               }

               /* □放所有□□□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□例程返回成功并且系□□□□接那个□□程序，□□接例程　□□将□□程序□□□接到系□。如果探□例程返回0　，□□接例程期望　接收完整的□□□□softc，此□□由探□例程□置。同□，如果探□例程　返回0，它可能期望□个□□的□接例程□当在将来的某点被□用。如果 探□例程返回□□，□□□程序可能不会作此假□。

如果成功完成，□接例程返回0，否□返回□□□。

□接例程的□□跟探□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□返回前就　　被□放，因此需要重新分配。我□希望探□例程已□□当地□置了　　所有的□源□□，并将它□保存在□□softc中。如果探□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□□的□用程序看起来□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□依□于□□程序的□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□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□□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□在控制台上打印由探□例程□置的□□描述。但 如果□□程序想打印一些□于□□的□外信息，也是可能的，例如：

```
       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 □个□□的精□方式依□于□□程序的□型：字符□□、□□□、网□ □□、CAM SCSI□□□□等等。

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。 我□利用□□一个事□：□□softc□□□我□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。

```
        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，□是一个很重要的 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□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， 决不不□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□□□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□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 □□□□的□□特□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□□的中断□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□理函数必□只向□□□答中断，但不能向中断控制器□答，后者由 系□□□□理。

Chapter 10. PCI□□

本章将□□FreeBSD□了□PCI□□上的□□□写□□程序而提供的机制。

10.1. 探□与□接

□儿的信息是□于PCI□□代□如何迭代通□未□接的□□，并□看新 加□的kld是否会□接其中一个。

10.1.1. 示例□□程序源代□(mypci.c)

```
/*
 * 与PCI函数□行交互的□□KLD
 *
 * Murray Stokely
 */

#include sys/param.h        /* kernel.h中使用的定□ */
#include sys/module.h
#include sys/systm.h
#include sys/errno.h
#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
#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   /* □了使用get_pci宏! */
#include dev/pci/pcireg.h

/* softc保存我□□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□□体cdev中的成□si_drv1□出我□的softc。
 * 当我□建立/dev□□，在我□的已附着的例程中，
 * 我□□置□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□置的□□与□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□成功□才□用□接函数 */

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□□□。 */

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□□□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□（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□，□可以□行 `make`□□示例□□程序。 □有，□可以□行`make load`将□□程序装□到当前正在□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. □□□源

FreeBSD□从父□□□求□源提供了一□面向□象的机制。几乎所有□□ 都是某□□型的□□（PCI, ISA, USB, SCSI等等）的孩子成□，并且□些□□ 需要从他□的父□□□取□源（例如内存段, 中断□, 或者DMA通道）。

10.2.1. 基地址寄存器

□了□PCI□□做些有用的事情，□需要从PCI配置空□□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□□程序可能在`attach()` 函数中有些□似下面的□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);
```

□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□□写入。

然后就能使用□些句柄与bus_space_*函数一起 □写□□寄存器。例如，□□程序可能包含如下的快捷函数，用来□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);
}
```

□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);
}
```

□些函数以8位，16位和32位的版本存在，□□当相□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□， 而不是 bus tag 和句柄。 □□， □就可以将 softc 中的 bus tag 和 bus 句柄□□个成□□量去掉， 并将 board_read() 函数改写□：

```
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□源相似的方式从面向□象的□□代□分配。首先， 必□从父□□分配IRQ□源，然后必□□置中断□理函数来□理□个IRQ。

再一次，来自□□attach()函数的例子比文字 更具□明性。

```
/* 取得IRQ□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□置中断□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□□的分□例程中必□注意一些□□。□必□停□□□的中断流， 并移除中断□理函数。一旦bus_teardown_intr()返回，□知道□的中断□理函数不会再被□用，并且所有可能已□□行了 □个中断□理函数的□程都已□返回。由于此函数可以睡眠，□用此函数□ □必□不能□有任何互斥体。

10.2.3. DMA

本□已□□，只是由于□史原因而□出。□理□些□□的□当方法是 使用bus_space_dma*()函数。当更新□一□以反映那□用法□，□段就可能被去掉。然而，目前API□不断有些□□，因此一旦 它□固定下来后，更新□一□来反映那些改□就很好了。

在PC上，想□行□□主控DMA的外□□□必□□理物理地址，由于 FreeBSD使用虚□内存并且只□理虚地址，□仍是个□□。幸□的是，有个 函数，vtophys()可以□助我□。

```
#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□者□FreeBSD的□□□□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□了以一□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□□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□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□求□列：

```
    struct cam_devq *devq;

    if(( devq = cam_simq_alloc(SIZE) )==NULL) {
        error; /* 一些□理□□的代□ */
    }
```

此□ SIZE □要分配的□列的大小， 它能包含的最大□求数目。 它是 SIM □□程序在 SCSI □上能□并行□理的□求的数目。一般可以如下估算：

```
SIZE = NUMBER_OF_SUPPORTED_TARGETS * MAX_SIMULTANEOUS_COMMANDS_PER_TARGET
```

下一□□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□□□理代□ */
    }
```

注意如果我□不能□建SIM描述符，我□也□放 devq，因□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□□，□□条□□需要它自己的 cam_sim □□。

一个有趣的□□是，如果SCSI□有不只一条SCSI□□我□□□□做， □个□需要一个devq□□□是□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□于控制器 "wds0"的此数字将□0。
- max_dev_transactions - 无□□模式下□个SCSI目□的 最大并□（simultaneous）事□数。□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□行一个事□的同□准□□ 一个事□，可以将其□置□2，但似乎不□得□加□□□□性。
- max_tagged_dev_transactions - 同□的□西，但是 在□□模式下。□□是SCSI在□□上□起多个事□的方式：□个事□ 被□予一个唯一的□□，并被□送到□□。当□□完成某些事□，它 将□果□同□□一起□送回来，□□SCSI□配器（和□□程序）就能知道 □个事□完成了。此参量也被□□是最大□□深度。它取决于SCSI□配器的能力。

最后我□注册与我□的SCSI□配器□□的SCSI□□。

```
    if(xpt_bus_register(sim, bus_number) != CAM_SUCCESS) {
        cam_sim_free(sim, /*free_devq*/ TRUE);
        error; /* 一些□□□理代□ */
    }
```

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□0，否□SCSI□上的□条□□□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□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□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□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□□在内的通配符路径。□□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□□□理代□ */
    }

    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□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□趣的事件：何□目□□□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□(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□。

□求的□型被存□在 ccb-ccb_h.func_code。因此，通常 xxx_action()由一个大的switch□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□句部分可以看出（如果收到未知命令），命令的返回□ 被□置到 ccb-ccb_h.status中，并且通□ □用xpt_done(ccb)将整个CCB返回到CAM中。

xpt_done()不必从 xxx_action()中□用：例如I/O□求可以在SIM□□程序 和/或它的SCSI控制器中排□。（□注：它指I/O□求？） 然后，当□□□□(post)一个中断信号，指示□此□求的□理已□束□， xpt_done()可以从中断□理例程中被□用。

□□上，CCB状□不是□□被□□□一个返回□，而是始□有某□状□。 CCB被□□□xxx_action()例程前，其取得状□CCB_REQ_INPROG，表示其正在□行中。/sys/cam/cam.h 中定□了数量□人的状□□，它□□□能非常□尽地表示□求的状□。 更有趣的是，状□□□上是一个枚□状□□（低6位）和一些可能出□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□旗□□：

- *CAM_DEV_QFRZN* - 当□理CCB□， 如果SIM□□程序得到一个□重□□（例如，□□程序不能□□□□或□反了SCSI□□），它□当□用xpt_freeze_simq()□□ □求□列，把此□□的其他已入□但尚未被□理的CCB返回到CAM□列， 然后□有□□的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_AUTOSENSE，SIM□□程序必□自□□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□求会被xpt_freeze_simq() 停止。SIM□□程序克服□□短缺情况，并通□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□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□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* - □求的路径ID

- *target_id* - □求的目□□□ID
- *target_lun* - 目□□□的LUN ID
- *timeout* - □个命令的超□□隔，以毫秒□
- *timeout_ch* - 一个□SIM□□ 程序存□超□□理函数的方便之所（CAM子系□自身并不□此作任何假□）
- *flags* - 有□□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，它□作□□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□合的其他形式大小一致的字□数□， sim_priv.bytes□一个□倍大小的数□

使用CCB的SIM私有字段的建□方法是□它□定□一些有意□的名字， 并且在□□程序中使用□些有意□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□起者模式的□求是：

- *XPT_SCSI_IO* - □行I/O事□

  □合ccb的"struct ccb_scsiio csio"□例用于□□参量。 它□是：

  - *cdb_io* - 指向SCSI命令□冲区的指□或□冲区本身
  - *cdb_len* - SCSI命令□度
  - *data_ptr* - 指向数据□冲区的指□（如果使用分散/集中会□□一点）
  - *dxfer_len* - 待□□数据的□度
  - *sglist_cnt* - 分散/集中段的□数
  - *scsi_status* - 返回SCSI状□的地方
  - *sense_data* - 命令返回□□□保存SCSI sense信息的□冲区（□□情况下，如果没有 □置CCB的旗□CAM_DIS_AUTOSENSE，□假定SIM□□程序会自□□行 REQUEST SENSE命令）
  - *sense_len* - □冲区的□度（如果□巧大于sense_data的大小，SIM□□程序必□ 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□有意□， 因此当很□□算的情况下（例如，□数SCSI控制器FIFO□冲区中的字□数），使用近似□也同□可以。□于成功完成的□□，它□必□被□置 □0。
  - *tag_action* - 使用的□□的□□有：
    - CAM_TAG_ACTION_NONE - 事□不使用□□
    - MSG_SIMPLE_Q_TAG, MSG_HEAD_OF_Q_TAG, MSG_ORDERED_Q_TAG - □等于□当的□□信息（□/sys/cam/scsi/scsi_message.h）；□□出□□□型，SIM□□程序 必□自己□□□□

    □理□求的通常□□如下：

要做的第一件事情是□□可能的□争条件，□保命令位于□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
    || cch_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□□理□求所需的数据□□（如□相□的硬件控制□等）。 如果我□不能分配□□□SIM□列，□□下我□有一个挂起的操作，返回 CCB，□求CAM将CCB重新入□。以后当□源可用□，必□通□返回其 状□中□置 CAM_SIMQ_RELEASE 位的ccb来解□SIM□列。否□，如果所有 正常，□将CCB与硬件控制□（HCB）□接，并将其□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□数据到硬件控制□。□□是否要求我□分配一个 □□，如果是□□生一个唯一的□□并□造SCSI□□信息。SIM□□程序 也□□与□□□商□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□置SCSI命令。可以在CCB中以多□有趣的方式指定命令的存□，　　□些方式由CCB中的旗□指定。命令□冲区可以包含在CCB中或者用指□　　指向，后者情况下指□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□□是将地址□□□物理地址。

不太相□的提示：通常□是通□□用vtophys()来完成的，但由于　　特殊的Alpha怪□之□，□了PCI□□（它□□在占SCSI控制器的大多数） □□程序向Alpha架□的可移植性，□□必□替代以 vtobus() 来完成。 [IMHO 提供□个□独的函数 vtop() 和 ptobus()，而 vtobus() 只是它□的 □□□代，□□做要好得多。]　　在□求物理地址的情况下，返回□有状□　　CAM_REQ_INVALID　　的CCB是可以的，当前的□□程序就是那□做的。但也　　可能像□个例子（□□程序中□当有不□条件□□的更直接做法）中那□　□□Alpha特定的代□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□不那□做。

```
    if(ccb_h-flags  CAM_CDB_POINTER) {
        /* CDB is a pointer */
        if(!(ccb_h-flags  CAM_CDB_PHYS)) {
            /* CDB指□是虚□的 */
            hcb-cmd = vtobus(csio-cdb_io.cdb_ptr);
        } else {
            /* CDB指□是物理的 */
#if defined(__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(□冲区)中 */
        hcb-cmd = vtobus(csio-cdb_io.cdb_bytes);
    }
    hcb-cmdlen = csio-cdb_len;
```

□在是□置数据的□候了，又一次，可以在CCB中以多□有趣的方式　　指定数据存□，□些方式由CCB中的旗□指定。首先我□得到数据□□的 方向。最□□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□□□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□理有限数目有限□度的　　大□。如果□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□理HCB□源短缺。□加chunk的函数是　□□程序相□的，此□我□不□入它□的□□□□。□于地址翻□□□　　的□□可以参看SCSI命令(CDB)□理的描述。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)) {
        /* □个□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□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□冲区指□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□冲区指□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□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□个CCB不允□断□□接，我□就□□□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□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□的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□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□将会在下一次事□□□求重新□商。

- *XPT_RESET_BUS* - □送RESET信号到SCSI□□

  CCB中并不□□参量，唯一感□趣的参量是由指向□□sim的指□□□ 的SCSI□□。

  最小□□会忘□□□上所有□□的SCSI□商，并返回状□ CAM_REQ_CMP。

  恰当的□□□□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□□□□ 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□，　　它会被超□函数作□最后的□告来重用。

- *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 控制□列表，□□与□个CCB□□的控制□来完成：

```
    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□个旗□并跳□它。

```
    int hstatus;

    /* 此□□示□一个函数，有□需要特殊□作才能使得□个旗□□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);
        /* □□□行 */
    case HCB_COMPLETED:
        /* □是一□□□的情况 */
        free_hcb_and_ccb_done(hcb, abort_ccb, CAM_REQ_ABORTED);
        break;
```

如果CCB此□正在□□中，我□一般会以某□硬件相□的方式□信号 □SCSI控制器，通知它我□希望中止当前的□□。SCSI控制器会□置 SCSI ATTENTION信号，并当目□□其□行□□后□送ABORT消息。我□也□位 超□，以□保目□不会永□睡眠。如果命令不能在某个合理的□□，如 10秒内中止，超□例程就会□行并□位整个SCSI□□。由于命令会在某个 合理的□□后被中止，因此我□□在可以只将中止□求返回，当作成功完成，并将被中止的CCB□□□中止（但□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□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□□，再□□一次
         *  （□注：脱□＝getoff）*/
            goto abort_again;
        }

        break;
```

如果CCB位于断□□接的列表中，□将它□置□中止□求，并在硬件 □列的前端将它重新入□。□位超□，并□告中止□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□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□　□个函数能□被重用。唯一的不同是超□事□将□超□□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□□□的排□
  - *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□□□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□理I/O□求□，它会□□其是否需要重新□商，例如通□□用函数target_negotiated(hcb)。它可以如下□□：

```
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□商□些□后，□果□必□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;
```

□□出了一般思路，精□□算依□于特定BIOS的癖好(quirk)。如果 BIOS没有提供方法□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□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□□□后会是255/63。

- *XPT_PATH_INQ* - 路径□□， □句□□，□得SIM□□程序和SCSI控制器（也称□HBA - 主机□□□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□模式支持的旗□，如果不支持□□0
- hba_misc - 控制器特性□□：
- PIM_SCANHILO - 从高ID到低ID的□□□描
- PIM_NOREMOVE - 可移除□□不包括在□描之列
- PIM_NOINITIATOR - 不支持□起者角色
- PIM_NOBUSRESET - 用□禁用初始BUS RESET
- hba_eng_cnt - 神秘的HBA引□□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□ID（□8位□□□7， 16位□□□15，光□通道□127）
- max_lun - 最大支持的LUN ID（□□老的SCSI控制器 □7，□新的□63）
- async_flags - 安装的□□□理函数的位掩□，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□尾的字符串， 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. □□

```
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□麻□ 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□，□□当定□回□函数。

```
static void
ahc_async(void *callback_arg, u_int32_t code, struct cam_path *path, void *arg)
```

- callback_arg - 注册回□□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□型事件的□□，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□型依□于SCSI控制器所□接到的外□□□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splcam()来同□中断例程与□□程序剩余部分的活□（□于能察□多□理器的□□程序，事情更要有趣，但 此□我□忽略□□情况）。本文□中的□代□□□地忽略了同□□□。 □□代□一定不能忽略它□。一个□□的□法就是在□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□□是中断可能会被阻塞相□ 很□的事件，□会□系□性能□生□面影□。□一方面，spl()函数族有相当高的□外□□，因此大量 很小的□界区可能也不好。

中断例程□理的情况和其中□□□重依□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□一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);
```

```
            }
        }

        /* □告事件 */
        xpt_async(AC_BUS_RESET, softc-wpath, NULL);

        /* 重新初始化可能花很多□□，□□情况下□当由□一中断□信号
         * 指示初始化否完成，或在超□□□□ - 但□了□□我□假□
         * 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□□的条件引起的，□很可能当前硬件控制□ 出□了□□。依□于硬件，可能有非HCB相□的事件，此□我□指示不考□ 它□。然后我□分析□个HCB□生了什□：

```
    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□的某些□西
         */
        □行必要的□理;
        return;
    }

    targ = hcb-target;
    hcb_status = get_status_of_current_hcb(softc);
```

首先我□□□HCB是否完成，如果完成我□就□□返回的SCSI状□。

```
    if(hcb_status == COMPLETED) {
        scsi_status = get_completion_status(hcb);
```

然后看□个状□是否与REQUEST SENSE命令有□，如果有□□□□ 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(softc);
            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(softc);
            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(softc);
        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□□ */
        softc-current_bus_width[targ] = softc-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(softc);
        return;
```

```
    case TARGET_ANSWERED_WIDE_NEG:
        {
            int wd;

            wd = get_target_bus_width_request(softc);
            if(wd = softc-goal_bus_width[targ]) {
                /* 可接受的回答 */
                softc-current_bus_width[targ] =
                softc-goal_bus_width[targ] = neg.bus_width = wd;

                /* 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(softc);
        return;
    case TARGET_REQUESTED_WIDE_NEG:
        {
            int wd;

            wd = get_target_bus_width_request(softc);
            wd = min (wd, OUR_BUS_WIDTH);
            wd = min (wd, softc-user_bus_width[targ]);

            if(wd != softc-current_bus_width[targ]) {
                /* 度改了 */
                softc-current_bus_width[targ] =
                softc-goal_bus_width[targ] = neg.bus_width = wd;

                /* 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_rsponse(hcb, wd);
        }
        continue_current_hcb(softc);
        return;
    }
```

然后我用与前面相同的法理auto-sense期可能出的任何 。否，我再一次入。

```
    if(hcb-flags  DOING_AUTOSENSE)
        goto autosense_failed;

    switch(hcb_status) {
```

我□考□的下一事件是未□期的□接断□，□个事件在ABORT或 BUS DEVICE RESET消息之后被看作是正常的，其他情况下是非正常的。

```
    case UNEXPECTED_DISCONNECT:
        if(requested_abort(hcb)) {
            /* 中止影□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□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□接受□□，我□就通知CAM，并返回此LUN的所有命令：

```
    case TAGS_REJECTED:
        /* 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;
        /* 求后面的代列 */
        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;
        /* 求后面的代列 */
        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) {
        /* □□□列 */
        xpt_freeze_devq(ccb-ccb_h.path, /*count*/1);

*        /* 重新入□□个目□/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□理，尽管特定□理器可能需要某些附加□理。

11.6. □□□□

当□行I/O□求□很多事情可能出□。可以在CCB状□中非常□尽地 □告□□原因。使用的例子散布于本文□中。□了完整起□此□□出 □典型□□条件的建□□□的一个□□：

- *CAM_RESRC_UNAVAIL* - 某些□源 □□不可用，并且当其□□可用□SIM□□程序不能□生事件。□□□源的一个例子就是某些控制器内部硬件□源，当其可用□控制器不会□其 □生中断。
- *CAM_UNCOR_PARITY* - □生不可恢□的奇偶校□□□
- *CAM_DATA_RUN_ERR* - 数据外溢或未□期的数据状□(phase)（□在□一个方向上而不是CAM_DIR_MASK指定的方向），或□于□□□出□奇数□□□度
- *CAM_SEL_TIMEOUT* - □生□□超□（目□不□□）
- *CAM_CMD_TIMEOUT* - □生命令超□（超□函数□行）
- *CAM_SCSI_STATUS_ERROR* - □□返回的□□
- *CAM_AUTOSENSE_FAIL* - □□返回的□□且REQUEST SENSE命令失□
- *CAM_MSG_REJECT_REC* - 收到MESSAGE REJECT消息
- *CAM_SCSI_BUS_RESET* - 收到SCSI□□□位
- *CAM_REQ_CMP_ERR* - 出□"不可能(impossible)"SCSI状□(phase) 或者其他怪□事情，或者如果□一□的信息不可用□只是通用□□
- *CAM_UNEXP_BUSFREE* - 出□未□期的断□□接
- *CAM_BDR_SENT* - BUS DEVICE RESET消息被□送到目□
- *CAM_UNREC_HBA_ERROR* - 不可恢□的主机□□□配器□□
- *CAM_REQ_TOO_BIG* - □求□于控制器太大

- *CAM_REQUEUE_REQ* - 此□求□当被重新入□以保持事□的次序性。□典型地出□在下列 □刻：SIM□□出了□当□□□列的□□，并且必□在sim□□上将目□的 其他入□□求放回到XPT□列。□些□□的典型情况有□□超□、命令超□和其他□似情况。□些情况下出□□的命令返回状□来指示□□， 此命令和其他□没有被□送到□□的命令被重新入□。
- *CAM_LUN_INVALID* - SCSI控制器不支持□求中的LUN ID
- *CAM_TID_INVALID* - SCSI控制器不支持□求中的目□ID

11.7. 超□□理

当HCB的超□期□□，□求就□当被中止，就像□理XPT_ABORT□求 一□。唯一区□在于被中止的□求的返回状□□当□CAM_CMD_TIMEOUT 而不是CAM_REQ_ABORTED（□就是□什□中止的□□最好由函数来完成）。 但□有一个可能的□□：如果中止□求自己出了麻□□□□？□□情况下 □当□位SCSI□□，就像□理XPT_RESET_BUS□求一□（并且将其□□□ 函数，从□个地方□用的想法也□用于□儿）。而且如果□□□位□求出了 □□，我□□当□位整个SCSI□□。因此最□超□函数看起来像下面□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□求□，同一目□/LUN的所有其他断□□接的□求 也会被中止。因此出□了一个□□，我□□当返回它□的状□ CAM_REQ_ABORTED□是CAM_CMD_TIMEOUT？当前的□□程序使用 CAM_CMD_TIMEOUT。□看起来符合□□，因□如果一个□求超□，□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

通用串行□□(USB)是将□□□接到个人□算机的一□新方法。□□ □□突出了双向通信的特色，并且其□□充分考□到了□□正逐□智能化 和需要与host□行更多交互的□□。□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。□果(Apple)引入□□USB的iMac□硬件 制造商生□他□USB版本的□□是一个很大的激励。未来的PC□□指定 PC上的所有老□接器□当由一个或多个USB□接器取代，提供通用的即□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□NetBSD□目□□的。代□已□被移植到FreeBSD上， 我□目前□□着一个底□共享代□。□USB子系□的□□来□，□多USB的特性很重要。

*Lennart Augustsson*已□完成了*NetBSD*□目中*USB*支持的 大部分□□。十分感□□□工作量□人的工作。也十分感□*Ardy*和*Dirk* □本文稿的□□和校□。

- □□直接□接到□算机上的端口，或者□接到称□ 集中器的□□，形成□型□□□□。
- □□可在□行□□接或断□。
- □□可以挂起自身并触□host系□的重新投入□行。
- 由于□□可由□□供□，因此host□件必□跟踪□个 集中器的□源□算。
- 不同□□□型需要不同的服□□量，并且同一□□ 可以□接最多126个□□，□就需要恰当地□度□□上的□□以充分利用12Mbps的可用□□。（USB 2.0超□400Mbps）
- □□智能化并包含很容易□□到的□于自身的信息。

□USB子系□以及□接到它的□□□□□□程序受已□□或将要□□的 □□的支持。□些□□可以从USB主□公□□得。□果(Apple)通□使得 通用□□□程序可从其操作系□MacOS中□得，而且不鼓励□□□新□□ 使用□独的□□程序来□烈推行基于□准的□□程序。本章□□整理基本 信息以便□FreeBSD/NetBSD中USB□的当前□□有个基本的了解。然而， 建□将下面参考中提及的相□□□与本章同□□□。

12.1.1. USB□的□□

FreeBSD中的USB支持可被分□三□。最底□包含主控器，向硬件 及其□度□施提供一个通用接口。它支持硬件初始化，□□□□行□度， □理已完成/失□的□□。□个主控器□□程序□□一个虚□hub， 以硬件无□方式提供□控制机器背面根端口的寄存器的□□。

中□□□理□□□接和断□，□□的基本初始化，□□程序的□□， 通信通道（管道）和□源管理。□个服□□也控制默□管道和其上□□的 □□□求。

□□包含支持特定（□）□□的各个□□程序。□些□□程序□□ 除默□管道外的其他管道上使用的□□。他□也□□□外功能，使得□□ □内核或用□空□是可□的。他□使用服□□暴露出的USB□□程序接口 (USBDI)。

12.2. 主控器

主控器（HC）控制□□上包的□□。使用1毫秒的□。在□□□始 □，主控器□生一个□□始（SOF, Start of Frame）包。

SOF包用于同□□的□始和跟踪□的数目。包在□中被□□，或由host 到□□（out），或由□□到host（in）。

□□□是由host□起（□□□□）。 因此□条USB□□只能有一个host。□个包的□□都有一个状□□段，数据接收者可以在其中返回ACK（□答接收），NAK（重□），STALL（□□ 条件）或什□也没有（混乱数据□段，□□不可用或已断□）。USB□□ USB specification的第8.5□更□□地解□了包的□□。USB□□ 上可以出□四中不同□型的□□：控制(control)， 大□(bulk)， 中断 (interrupt)和同□(isochronous)。□□的□型和他□的特性在下面 描述（管道’子□中）。

USB□□上的□□和□□□□程序□的大型□□被主控器或HC □□程序分割□多个包。

到默□端点的□□□求（控制□□）有些特殊。它□由□或三个□段 □成：□□（SETUP），数据（DATA，可□）和状□（STATUS）。□置（set-up） 包被□送到□□。如果存在数据□段，数据包的方向在□置包中□出。 状□□段中的方向与数据□段期□的方向相反，或者当没有数据□段□
□IN。主控器硬件也提供寄存器，用于保存根端口的当前状□和自从 状□改□寄存器最后一次□位以来所□生的改□。USB□□[2]建□使用一个 虚□hub来提供□□些寄存器的□□。虚□hub必□符合□□第11章中□出的hub□□□。它必□提供一个默□管道使得□□□求可以□送□它。它返回 □准和hub□特定的一□描述符。它也□当提供一个中断管道用来□告其 端口□生的□化。当前可用的主控器□□有□个： 通用主控器接口（UHCI；英特□）和□放主控器接口（OHCI；康柏，微□，国家半□体）。 UHCI□□的□□通□要求主控器□□程序□□□的□□提供完整的□度， 从而□少了硬件□□性。OHCI□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主控器□□着□有1024个指向□□数据□□的□列表。 它理解□□不同的数据□型：□□描述符（TD）和□列□（QH）。□个 TD表示表示与□□端点□行通信的一个包。QH是将一些TD（和QH）□分 成□的一□方法。

□个□□由一个或多个包□成。UHCI□□程序将大的□□分割成 多个包。除同□□□外，□个□□都会分配一个QH。□于□□□型的 □□，都有一个与此□型□□的QH，所有□些QH都会被集中到□个QH上。 由于有固定的□延需求，同□□□必□首先□行，它是通□□列表中的 指□直接引用的。最后的同□TD□□引用那一□的中断□□的QH。中断 □□的所有QH指向控制□□的QH，控制□□的QH又指向大□□□的QH。 下面的□表□出了一个□形概□：

□□致下面的□度会在□□中□行。控制器从□列表中取得当前□ 的指□后，首先□那一□中的所有的同□(isochronous)包□行TD。 □些TD的最后一个 引用那一□的中断□□的QH。然后主控器将从那个QH下行到各个 中断□□的QH。完成那一□列后，中断□□的QH会将控制器指向到所有控制□□的QH。它将□行在那儿等待□度的所有子□列，然后是在大□QH中 排□的所有□□。□了方便□理已完成或失□的□□，硬件会在□□末尾 □生不同□型的中断。在□□的最后一个TD中，HC□□程序□置Interrupt-On-Completion位来□□□完成□的一个中断。如果TD□到了 其最大□□数，就□□□□中断。如果在TD中□置短包□□位，且□□了 小于所□置的包□度（的包），就会□□此中断以通知控制器□□程序□□ 已完成。□出□个□□已完成或□生□□是主控器□□程序的任□。 当中断服□例程被□用□，它将定位所有已完成的□□并□用它□的回□。

更□尽的描述□看 UHCI specification。

12.2.2. OHCI

□OHCI主控器□行□程要容易得多。控制器假□有一□端点(endpoint)可用， 并知道□中不同□□□型的□度□先□和排序。主控器使用的主要 数据□□是端点描述符（ED），它上面□接着一个□□描述符（TD）的□列。ED包含端点所允□的最大的包大小，控制器硬件完成包的分割。□次□□ 后都会更新指向数据□冲区的指□，当起始和□止指□相等□，TD就退□ 到完成□列(done-queue)。四□□型的端点各有其自己的□列。控制和大□(bulk)端点分□在它□自己的□列排□。中断ED在□中排□，在□中的深度 定□了它□□行的□度。

□列表 中断 同□(isochronous) 控制 大□(bulk)

主控器在□□中□行的□度看起来如下。控制器首先□行非　周期性控制和大□□列，最□可到HC□□程序□置的一个□□限制。　然后以□□号低5位作□中断ED□上深度□0的那一□中的索引，□行　那个□□号的中断□□。在□个□的末尾，同□ED被□接，并随后被　遍□。同□TD包含了□□□当□行其中的第一个□的□□号。所有周期　性的□□□行□以后，控制和大□□列再次被遍□。中断服□例程会被　周期性地□用，来□理完成的□列，□□个□□□用回□，并重新□度 中断和同□端点。

更□尽的描述□看　OHCI　specification。服□□，即中□□，提供了以可控的方式　□□□□行□□，并□□着由不同□□程序和服□□所使用的□源。 此□□理下面几方面：

- □□配置信息
- 与□□□行通信的管道
- 探□和□接□□，以及从□□分□(detach)。

12.3. USB□□信息

12.3.1. □□配置信息

□个□□提供了不同□□的配置信息。□个□□具有一个或多个　配置，探□/□接期□从其中□定一个。配置提供功率和□□要求。　□个配置中可以有多个接口。□□接口是端点的□集(collection)。例如，USB□声器可以有一个音□接口（音□□），和□旋□(knob)、　□号□(dial)和按□的接口（HID□）。一个配置中的所有接口可以同□有效，并可被不同的　□□程序□接。□个接口可以有□用接口，以提供不同□量的服□参数。 例如，在照相机中，□用来提供不同的□大小以及□秒□数。

□个接口中可以指定0或多个端点。端点是与□□□行通信的□向　□□点。它□提供□冲区来□□存□从□□而来的，或外出到□□的数据。　□个端点在配置中有唯一地址，即端点号加上其方向。默□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□□ 管理，并且□□□□程序不能直接使用。

Level 0 Level 1 Level 2 Slot 0

Slot 3 Slot 2 Slot 1

(只□示了32个槽中的4个)

□□□次化配置信息在□□中通□□准的一□描述符来描述（参看　USB□□[2]第9.6□）。它□可以通□Get　Descriptor Request来□求。　服□□□存□些描述符以避免在USB□□上□行不必要的□□。□□些　描述符的□□是通□函数□用来提供的。

- □□描述符：□于□□的通用信息，如供□商，□品　和修□ID，支持的□□□、子□和□用的□□，默□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□能力，以及功率要求。
- 接口描述符：接口□、子□和□用的□□，接口□用 配置的数目和端点数目。
- 端点描述符：端点地址、方向和□型，支持的最大包　大小，如果是中断□型的端点□□包括□□□率。默□端点（端点0） 没有描述符，而且从不被□入接口描述符中。
- 字符串描述符：在其他描述符中会□某些字段提供　字符串索引。它□可被用来□取描述性字符串，可能以多□□言 的形式提供。

□□明(specification)可以添加它□自己的描述符□型，□些描述符 也可以通□GetDescriptor Request来□得。

管道与□□上端点的通信，流□所□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□好接收数据，□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□□的数据量□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□□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□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□□的可用性在管道的□建期□被□算。□□在1毫秒的□内 □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□。可能有多个□□□□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，直到 在一个配置中□接到一个□□程序。□了支持不同接口上使用多个□□ 程序的□□，探□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□算的配置会被忽略。□接期□，□□ 程序□当把□□初始化到□当状□，但不能□位，因□那□会使得□□将 它自己从□□上断□，并重新□□探□□程。□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□一个□□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□□。□些□□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□。

□一个信息的很好来源是Linux□□程序源代□，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。□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□它并□接(attach)到它。

□些□型的□□的一个例子是基于EZ-USB的ActiveWire I/O板。□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□，并再次 重新□接。

例子：大容量存□□□。□大容量存□□□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□信息被包装到□中，在大□(bulk)管道上□□到/来自□□，在USB□ 上模□SCSI控制器。ATAPI和UFI命令以相似的方式被支持。

大容量存□□□支持□□不同□型的□命令□的包装。最初的□□ 基于通□默□管道□送命令和状□信息，使用大□□□在host和□□之□ 移□数据。在□□基□上□□出□一□方法，□□方法基于包装命令和 状□□，并在大□out和in端点上□送它□。□□精□地指定了何□必□ □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□□□ 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□□解□了Newbus□□框架。

13.1. □□□□程序

13.1.1. □□□□程序的目的

□□□□程序是□件□件，它在内核□于外□□□（例如，磁□、网□ □配□）的通用□□和外□□□的□□□□之□提供了接口。□□□□程序接口*(DDI)*是内核与□□□□程序□件 之□定□的接口。

13.1.2. □□□□程序的□型

在UNIX®那个□代，FreeBSD也从中延□而来，定□了四□□型的 □□：

- □□□□□程序
- 字符□□□□程序
- 网□□□□□程序
- □□□□□程序

□□□以使用固定大小的[数据]□的方式□行。 □□□型的□□程序依□所□的 □冲区□存*(buffer cache)*，其目的是在内存中的□用区域□存□□□的数据□。□□□冲区□存常常基于后台写 (write-behind)，□意味着数据在内存中被修改后，当系□□行其周期性 磁□刷新□才会被同□到磁□，从而□化写操作。

13.1.3. 字符□□

然而，在FreeBSD 4.0版本以及后□版本中， □□□和字符□□的区□□得不存在了。

13.2. Newbus概□

Newbus□□了一□基于抽象□的新型□□□□， 可以在FreeBSD 3.0中看到□□□□□□的介□，当□Alpha的移植被□入到代□□中。直到4.0它才成□□□□□程序使用的默□系□。其目的是□主机 系□提供□操作系□的各□□□和□□的互□提供更加 面向□象的方法。

其主要特性包括：

- □□□接
- □□程序容易模□化
- □□□

最□著的改□之一是从平面和特殊系□演□□□□□布局。

□□□留的是"根"□□，它作□ 父□□，所有其他□□挂接在它上面。□于□个□□，通常"根" 只有□个孩子，其上□接着□如*host-to-PCI*□ 等□西。□于x86，□□"根"□□□ *"nexus"*□□，□于Alpha，Alpha的各□ 不同型号有不同的□□□□

，□□不同的硬件芯片□，包括 *lca*，*apecs*， *cia*和*tsunami*。

Newbus上下文中的□□表示系□中的□个硬件□体。例如，□个PCI□□被 表示□一个Newbus□□。系□中的任何□□可以有孩子；有孩子的□□通常被 称□*"bus"*。系□中常用□□的例子就是 ISA和PCI，他□各自管理□接到ISA和PCI□□上的□□列表。

通常，不同□型的□□之□的□接被表示□ "□"□□，它的孩子就是它所□接的 □□。一个例子就是*PCI-to-PCI*□，它在父PCI□□上被 表示□*pcibN*，而用它的孩子 *-pciN*表示□接在它上面的 □□。□□布局□化了PCI□□□的□□，允□公共代□同□用于□□和□接的 □□。

Newbus□□中的□个□□□求它的父□□来□其映射□源。父□□接着□求 它的父□□，直到到□nexus。因此，基本上nexus是Newbus系□中唯一知道所有 □源的部分。

ISA□□可能想在`0x230`映射其IO端口，因此它向其 父□□□求，□□情况下是ISA□□。ISA□□将它交□PCI-to-ISA□，PCI-to-ISA □接着□求PCI□□，PCI□□到□host-to-PCI□，最后到□nexus。□□向上 □渡的□美之□在于可以有空□来□□□求。□`0x230`IO端口 的□求在MIPS机器上可以被PCI□□成 `0xb0000230`□的内存映射。

□源分配可以在□□□的任何地方加以控制。例如，在很多Alpha平台上， ISA中断与PCI中断是□独管理的，□ISA中断的□源分配是由Alpha的ISA□□□□ 管理的。在IA-32上，ISA和PCI中断都由□□的nexus□□管理。□于□□移植， 内存和端口地址空□由□个□体管理 - 在IA-32上是nexus，在Alpha（例如，CIA或tsunami）上是相□的芯片□□□程序。

□了□□化□内存和端口映射□源的□□，Newbus整合了NetBSD的 `bus_space` API。他□提供了□一的API来代替inb/outb 和直接内存□写。□□做的□□在于□个□□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□者）。

□□支持被合并到了□源分配机制中。分配□源□，□□程序可以从□源 中□取□□的`bus_space_tag_t`和`bus_space_handle_t`。

Newbus也允□在□用于此目的的文件中定□接口方法。□些是 .m文件，可以在src/sys 目□□中□到。

Newbus系□的核心是可□展的"基于□象□程(object-based programming)"的模型。系□中的□个□□具有它所支持的一个方法表。 系□和其他□□使用□些方法来控制□□并□求服□。□□所支持的不同方法 被定□□多个"接口"。"接口"只是 □□□□的一□相□的方法。

在Newbus系□中，□□方法是通□系□中的各□□□□□程序提供的。当 自□配置*(auto-configuration)*期□□□被□接(attach) 到□□程序，它使用□□程序声明的方法表。以后□□可以从其□□程序 分□*(detach)*，并 重新□接*(re-attach)*到具有新方法表的新□□程序。□就 允□□□程序的□□替□，而□□替□□于□□程序的□□非常有用。

接口通□与文件系□中用于定□vnode操作的□言相似的接口定□□言来描述。接口被保存在方法文件中（通常命名□foo_if.m）。

例 6. *Newbus*的方法

```
# Foo 子系□/□□程序（注□...）

INTERFACE foo

  METHOD int doit {
      device_t dev;
  };

  # 如果没有通□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□□后，它□生一个□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□□生的□文件，也会□建一个源文件 "foo_if.c"；其中包含一些函数的□□，□些函数用于在□象方法表中□□相□函数的位置并□用那个函数。

系□定□了□个主要接口。第一个基本接口被称□ "□□*(device)*"，并包括与所有□□相□ 的方法。"□□*(device)*"接口中的方法 包括"探□*(probe)*"， "□接*(attach)*"和 "分□*(detach)*"，他□用来控制硬件的□□，以及"□□*(shutdown)*"， "挂起*(suspend)*"和 "恢□*(resume)*"，他□用于□□事件通知。

□一个，更加□□接口是"*bus*"。 此接口包含的方法□用于□有孩子的□□，包括□□□□特定的□□□信息 ，事件通知（`child_detached`，`driver_added`）和□□管理 （`alloc_resource`，`activate_resource`，`deactivate_resource`，`release_resource`）。

"bus"接口中的很多方法□□□□□的某些孩子□行服□。 □些方法通常使用前□个参量指定提供服□的□□和□求服□的子□□。□了 □化□□□□程序代□，□些方法中的很多都有□□者(accessor)函数，□□者 函数用来□□父□□并□用父□□上的方法。例如，方法 `BUS_TEARDOWN_INTR(device_t dev, device_t child, …)` 可以使用函数 `bus_teardown_intr(device_t child, …)`来□用。

系□中的某些□□□型提供了□外接口以提供□□□特定功能的□□。 例如，PCI□□□□程序定□了"pci"接口，此接口有□个方法 `read_config`和 `write_config`，用于□□PCI□□ 的配置寄存器。

13.3. Newbus API

由于Newbus API非常□大，本□努力将它文□化。本文□的下一版本会 □来更多信息。

13.3.1. 源代□目□□中的重要位置

src/sys/[arch]/[arch] - 特定机器□□的 内核代□位于□个目□。例如i386□□或 SPARC64□□。

src/sys/dev/[bus] - 支持特定 [bus]的□□位于□个目□。

src/sys/dev/pci - PCI□□支持代□位于 □个目□。

src/sys/[isa|pci] - PCI/ISA□□□□程序 位于□个目□。FreeBSD4.0版本中，PCI/ISA支持代□ □去存在于□个目□中。

13.3.2. 重要□□和□型定□

devclass_t - □是指向 struct devclass的指□的□型定□。

device_method_t - 与 kobj_method_t相同（参看 src/sys/kobj.h）。

device_t - □是指向 struct device的指□的□型定□。 device_t 表示系□中的□□。它是内核□象。 □□□□参看src/sys/sys/bus_private.h。

driver_t - □是一个□型定□，它引用 struct driver。 driver□□是一□ device(□□)内核□象；它也保存着□□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□□型，即 device_state。它包含Newbus□□在自□配置前后 可能的状□。

□□状□device_state_t

```
/*
 * src/sys/sys/bus.h
 */
typedef enum device_state {
  DS_NOTPRESENT,  /* 未探□或探□失□ */
      DS_ALIVE,       /* 探□成功 */
      DS_ATTACHED,    /* □用了□接方法 */
      DS_BUSY         /* □□已打□ */
} device_state_t;
```

Chapter 14. 声音子系□

14.1. □介

FreeBSD声音子系□清晰地将通用声音□理□□与□□特定的□□分□ □来。□使得更容易加入□新□□的支持。

pcm(4)框架是声音子系□的中心部分。它主要□□下面的□件：

- 一个到数字化声音和混音器函数的系□□用接口（read, write, ioctls）。ioctl命令集合兼容老的*OSS*或*Voxware*接口，允□一般多媒体□用程序 不加修改地移植。
- □理声音数据的公共代□（格式□□，虚□通道）。
- 一个□一的□件接口，与硬件特定的音□接口模□接口
- □某些通用硬件接口（ac97）或共享的硬件特定代□ （例如：ISA DMA例程）的□外支持。

□特定声□的支持是通□硬件特定的□□程序来□□的，□些□□程序 提供通道和混音器接口，□入到通用pcm代□中。

本章中，□□pcm将指声音□□程序的 中心，通用部分，□是□比硬件特定的模□而言的。

□期的□□程序□写者当然希望从□有模□□始，并使用那些代□作□ 最□参考。但是，由于声音代□十分□□漂亮，□也基本上免除了注□。 本文□□□□出框架接口的一个概□，并回答改写□有代□□可能出□的 一些□□。

作□□外的途径，或者□除了从一个可工作的□例□始的□法之外， □可以从http://people.FreeBSD.org/~cg/template.c□到一个注□□的 □□程序模板。

14.2. 文件

除/usr/src/sys/sys/soundcard.h中的公共 ioctl接口定□外，所有的相□代□当前(FreeBSD 4.4)位于/usr/src/sys/dev/sound/。

在/usr/src/sys/dev/sound/下面， pcm/目□中保存着中心代□， 而isa/和pci/目□中有 ISA和PCI板的□□程序。

14.3. 探□，□接等

声音□□程序使用与任何硬件□□程序模□相同的方法探□和□接（□□）。 □可能希望□□一下手册中ISA或PCI章□的内容来□取更多信息。

然而，声音□□程序在某些方面又有些不同：

- 他□将自己声明□pcm□□□，□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□□的附加私有信息。私有数据 □□通常在□接例程中分配。其地址通□□用 pcm_register()和 mixer_init()□□□ 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□□AC97□□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□□的指□，　　channel_init()是个例外，它的指□指向私有　　□□□□（并返回由pcm以后使用的通道指□）。

14.4.1.2. 数据□□操作概□

□于声音数据□□，pcm核心与声音□□ 程序是通□一个由struct snd_dbuf描述的 共享内存区域□行通信的。

struct snd_dbuf是 pcm私有的，声音□□程序通□□用□□者 函数（sndbuf_getxxx()）来□得感□趣的□。

共享内存区域的大小等于　sndbuf_getsize()，并被分割□大小固定，且等于　sndbuf_getblksz()字□的很多□。

当播放□，常□的□□机制如下（将意思反□来就是□音）：

- pcm□始□填充□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□大小□□，重□将　　整个内存区域（sndbuf_getbuf()，sndbuf_getsize()）□□到□□。□于□个 □□□回□pcm函数 chn_intr()（□通常在中断□□□生）。
- chn_intr()安排将新数据拷□到那些 数据已□□到□□（□在空□）的区域，并□ snd_dbuf□□□行□当的更新。

14.4.1.3. channel_init

□用xxxchannel_init()来初始化□个播放　和□音通道。□个□用从声音□□程序的□接例程中□起。（参看　探□和□接一□）。

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

b□通道　　struct　　snd_dbuf的地址。它□当在　　函数中通□□用sndbuf_alloc()来初始化。　　所用的□冲区大小通常是□□'典型'□□大小的一个□小的倍数。　　c□　　pcm通道控制□□的指□。□是个不透明　　指□。函数□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□当返回一个指□，此指□指向用于控制此通道的私有　区域。它将作□参数被□□到□其他通道接口的□用。

14.4.1.4. channel_setformat

xxxchannel_setformat()□当按特定通道，　特定声音格式□置硬件。

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

format AFMT_XXX value之一 （soundcard.h）。

14.4.1.5. channel_setspeed

xxxchannel_setspeed()按指定的取速度 置通道硬件，并返回返回可能整的速度。

```
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()置大小， 是pcm与声音程序，以及声音 程序与之的 位的大小。期，次 大小的 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 始 使用个。

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

函数返回可能整的大小。如果大小真的化了， 情况下当用sndbuf_resize()整 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□□□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□□□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□□FreeBSD□□写PC Card或CardBus□□的□□程序而提供的机制。 但目前本文只□□了如何向□有的pccard□□程序中添加□□程序。

15.1. 添加□□

向所支持的pccard□□列表中添加新□□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□□ /etc中的一个文件来列出□□。从FreeBSD 5.0□始，□□□□程序知道它□支持什□□□。□在内核中有一个受支持□□的表， □□程序用它来□接□□。

15.1.1. 概□

可以有□□方法来□□PC Card，他□都基于□上的 CIS信息。第一□方法是使用制造商和□品的数字□号。 第二□方法是使用人可□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，□□□程序的□写 者很容易地□定匹配其□□程序的□□。

一个很普遍的□□情况是，某个公司□一款PC Card□品□□出参考 □□，然后把□个□□□□□外的公司，以便在市□上出售。那些公司改□ 原□□，把向他□的目□客□群或地理区域出售□品，并将他□自己的名字 放到□中。然而所□的□□有□的改□，即使做□任何修改，□些修改通常 也微乎其微。然而，□了□化他□自己版本的品牌，□些供□商常常会把他□ 公司的名字放入CIS空□的可□字符串中，却不会改□制造商和□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□□□□是□的 真正制造者，尤其当提供原□的供□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□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□□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□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□□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□商列出了制造商号□。本□按数字排序。下一□包含了 □些供□商使用的所有□品，包括他□的□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□的CIS□置□□的描述）。然后□使用字符串匹配方法的 那些□□重□□□□的□西。最后，文件任何地方可以使用C□格的注□。

文件的第一□包含供□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
```

□示了几个供□商ID。很□巧的是NETGEAR_2 □□上是NETGEAR从其□□□的OEM，□那些□提供支持的作者那□并不知道 NETgear使用的是□人的ID。□些条目相当直接易□。□行上都有供□商 □□字来指示本行的□□。也有供□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。□有一个□供□商的十六□制数字ID。不要添加0xffffffff或0xffff形式的ID， 因□它□是保留ID（前者是'空ID集合'，而后者有□会在□量□其差的□中看到，用来指示none）。最后□有□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□在下面例子中看到的:

```
/* Allied Telesis K.K. */
product ALLIEDTELESIS LA_PCM    0x0002 Allied Telesis LA-PCM

/* Archos */
product ARCHOS ARC_ATAPI    0x0043 MiniCD
```

格式与供□商的那些行相似。其中有□品□□字。然后是供□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，但可以以数字□□。然后是□的十六□制□品ID。供□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□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， 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□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□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□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□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□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□例程□例

要□得如何向所支持的□□列表中添加□□，就必□□得很多□□程序 都有的探□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□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□。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□ □□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□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 cisvendor="BUFFALO"
cisproduct="WLI2-CF-S11" function_type=6 at function=0
```

作□□出的一部分。制造商和□品□□品的数字ID。而cisvender和 cisproduct□CIS中提供的描述本□品的字符串。

由于我□首先想□先使用数字□□，因此首先□□□建基于此的条目。 □了示例，上面的□已□被□□虚□化了。我□看到的供□商□BUFFALO， 它已□有一个条目了：

```
vendor BUFFALO          0x026f  BUFFALO (Melco Corporation)
```

□□我□就可以了。□□个□□□一个条目，但我□没有□□。但我□ □□：

```
/* 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□□□来 重新□生pccarddevs.h，用来将□些□□符□□ 到客□□□程序。□在□□程序中使用它□之前必□完成下面□□：

```
# cd src/sys/dev/pccard
# make -f Makefile.pccarddevs
```

一旦完成了□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□只是　　用来□□□一行。不要把它添加到□□□□程序中。一旦□添加了□行，就　　可以重新□□内核或模□，并□着看它是否能□□□□。如果它□□出□□　　并能工作，□提交□丁。如果它不工作，□□出□它工作所需要的□西并　　提交一个□丁。如果它根本不□□□□，那□□可能做□了什□，□当重新 □□□一□。

如果□是一个FreeBSD源代□的committer，并且所有□西看起来都　　正常工作，□□□当把□些改□提交到□中。然而有些小技巧的□西□　　需要考□。首先，□必□提交pccarddevs文件到　　□中。完成后，□必□重新□生pccarddevs.h　　并将它作□□一次提交来提交（□是□了□保正□的$FreeBSD$□□会留在后面的文件中）。最后，□需要把 其它□西提交到□□程序。

15.1.5. 提交新□□

很多人直接把新□□的条目□送□作者。□不要那□做。□将它□作□　　PR来提交，并将PR号□□送□作者用于□□。□□□保条目不会□失。提交　PR□，□丁中没有必要包含pccardevs.h的diff，　因□那些□西可以重新□生。包含□□的描述和客□□□程序的□丁是必要　　的。如果□不知道名字，使用OEM99作□名字，作者将会□□后相□地□整OEM99。提交者不□当提交OEM99，而□□□到最高的OEM条目并提交高于那个 的一个。

部分 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.*