pmon 学习笔记

如果生命有终点

我就陪你到终点

如果生命没终点

我就陪你到永远 ……

版本	修改	日期
V1.0	发布第一个版本	2009-6-1
V1.1	修改 start.S 中北桥初始化部分	2010-7-30

目录

扯

一开始大家都不认识，先胡乱抛出几句话扯扯，熟悉熟悉。就是开场白，也有人说叫序。
pmon 是 cpu 上电后执行的代码，相当于 x86PC 机中的 BIOS，兼有 bootloader 的功能，代码来源于早期 BSD 的内核，到如今已旧貌换新颜，糟蹋得差不多了。
pmon 的二进制代码存放于主板上的一块 512KB 的 flash 芯片上，选择这个容量是因为够用了，龙芯 2F 允许的最大 boot rom 容量是 1MB。
这块 flash 芯片的地址是确定的，虚拟地址 0xbfc00000，物理地址 0x1fc00000。
cpu 上电后，会在第一时刻从虚拟地址为 0xbfc00000 的读取指令执行，这个和 x86 的 cpu 是一样的，差别的只是地址。
地址差别的一方面是地址值，另一方面是地址类型。
x86 有实模式和保护模式的区别，上电初期为实模式，地址就是物理地址。MIPS 没有这两种模式的区别，而且一开始就是虚拟地址。虚拟地址是虚的，所以必定需要映射到物理地址。
下面介绍一下 pmon 文件相关的地址问题。
cpu 眼中的地址是虚拟地址，cpu 取指和取数据的地址是物理地址，经过北桥解释后的地址是总线地址，编译器产生的地址（包括解析了所有引用和重定位的符号后）为程序地址，也就是程序它自己理解的地址。不同的地址概念间需要有映射函数来关联。
mips 的虚拟地址到物理地址的映射比 x86 要复杂一些（也更有用些），既有可供页表之类动态建立映射函数的机制，也有 unmapped cacheed/uncached 这种固定映射关系的转换方式。bios 代码由于其执行时机的限制，开始阶段只能使用 unmapped uncached 这个段。地址范围是 0xa0000000~0xc0000000，也就是 2.5G 到 3G 的 512MB 空间。这个范围的映射函数为物理地址＝程序地址-2.5G。所有的 32 位 mips 架构的 cpu 都遵循这个约定。
pmon 的代码绝大部分是用 C 写的，只有个别的体系结构相关的代码使用了汇编。
好，先到这儿吧。

如果你对 pmon 感兴趣，想了解它的框架，
如果你对 pmon 感兴趣，想了解各个设备在 bios 中是如何工作的，
如果你对 pmon 感兴趣，想找一个理解 pmon 的 N 天搞定法。
以及不满足上述条件的其他人
就不要在这浪费时间了。本笔记只讲述流程。

好，想看流程或者愿意浪费时间的，Let‘s go。

start.S

start.S 在 pmon 中的作用？
核心是把 pmon 的二进制文件复制到内存。并初始化 cache，内存控制器，内存和南桥的部分信号。这个代码执行之后会执行 c 代码，解压在二进制中压缩的 bin 文件，跳到解压后的代码继续执行。

由于一上电的时候，内存和内存控制器都处于不确定的状态,所以 cpu 一开始执行的代码不能在内存中存放，只能把 bios 的代码放在非易失性的介质中（实际都是 nor 型的 flash）。由于在这类介质中的执行速度比较慢，所以尽可能的，要把其中的 bios 代码载入到内存执行，这需要先初始化内存控制器。要初始化内存控制器必须获得内存的 spd 信息，而内存的 spd 信息需要通过 i2c 总线读取,而 i2c 模块在南桥上，访问要在初始化 smbus 之后，要访问南桥又必须先初始化北桥，要访问北桥又需要先初始化 cpu 本身。这是个一环扣一环的过程。start.S 基本就是围绕这个流程展开的。当然在初始化南桥的 SMB 总线以外，顺便把南桥相关的其他一些信号也初始了，这些具体的可能要看电路图了。

start.S 是在 cpu 上电之后立即执行的代码。
为什么？就因为它叫 start.S?
MIPS cpu 约定 cpu 执行的第一条指令位于虚拟地址 0xbfc00000，而 pmon 的二进制代码是以 load -r -f bfc00000 gzrom.bin 这个命令烧入 bfc00000 这个地址开始的 flash，硬件布线会使得虚拟地址 0xbfc00000 映射到这块 flash 上。而 start.S 就占据了 gzrom.bin 的开头部分。
为什么 gzrom.bin 的开头就是 start.S?
是./zloader/Makefine.inc 中的 L29：
mips-elf-ld -T ld.script -e start -o gzrom ${START} zloader.o 决定的。
start.S 文件是用汇编语言写的，关于龙芯汇编语言的内容请参阅 mips 指令手册（龙芯没有这方面的公开手册）。
cpu 上电后，内部的寄存器一些可写的寄存器的值是不可预知的，所以代码先设定 cpu 的部分内部寄存器。在第一条汇编指令前有一行注释如下：
/* NOTE!! Not more that 16 instructions here!!! Right now it's FULL! */
就是说这里已经是 16 条指令了，不能再多了云云。如果你实在想不明白这怎么有 16 条指令，那么反汇编一把，看到的确是 16 条。

```
80010000 <_ftext>:
80010000:   40806000    mtc0    zero,c0_sr
80010004:   40806800    mtc0    zero,c0_cause
80010008:   3c080040    lui t0,0x40
8001000c:   40886000    mtc0    t0,c0_sr
80010010:   3c1d8001    lui sp,0x8001
80010014:   67bdc000    daddiu  sp,sp,-16384
```

```
80010018:    3c1c800f      lui gp,0x800f
8001001c:    679ce0e0      daddiu    gp,gp,-7968
80010020:    04110003      bal 80010030 <uncached>
80010024:    00000000      nop
80010028:    04110183      bal 80010638 <locate>
8001002c:    00000000      nop

80010030 <uncached>:
80010030:    3c01a000      lui at,0xa000
80010034:    03e1f825      or    ra,ra,at
80010038:    03e00008      jr    ra
8001003c:    00000000      nop
```

如果你没想明白，还要问那么为什么不能超过 16 条呢？。。。。。。。。。。
我也不知道了。呵呵。
如果你非要铤而走险，在中间多加些指令非让它超过 16 条，那么哥们你太聪明了，你将发现这只是个玩笑而已。是吧，这帮写代码的有时没正经。

好，言归正传，这个 start.S 共有 2656 行，的确不算太短。

现在我们按照 cpu 的执行顺序对 start.S 代码进行逐一分析。

cpu 执行的第一条指令在 L 213，

```
mtc0      zero, COP_0_STATUS_REG
mtc0      zero, COP_0_CAUSE_REG
```

zero 就是寄存器 0($0)，和下面的 t0($8)之类相似，都由 asm.h 中 incldue 的 refdef.h 定义的，由预处理去解析。
一开始将状态寄存器和原因寄存器清零主要是禁用所有的中断和异常检测，并使当前处于内核模式。之后执行：

```
li    t0, SR_BOOT_EXC_VEC /* Exception to Boostrap Location */
mtc0      t0, COP_0_STATUS_REG
```

以上两句只是设定当前的异常处理模式处于启动模式，启动模式(BEV=1)和正常模式(BEV=0)的区别是异常处理的入口地址不同，具体如下：

BEV 位	例外类型	例外向量地址
	Cold Reset/ NMI	0xFFFFFFFF BFC00000
BEV = 0	TLB Refill (EXL=0)	0xFFFFFFFF 80000000
	XTLB Refill (EXL=0)	0xFFFFFFFF 80000000
	Others	0xFFFFFFFF 80000180
BEV = 1	TLB Refill (EXL=0)	0xFFFFFFFF BFC00200
	XTLB Refill (EXL=0)	0xFFFFFFFF BFC00200
	Others	0xFFFFFFFF BFC00380

此表来自龙芯 2F 的用户手册,更详细的 status 位描述见手册第五章。

```
    la    sp, stack
    la    gp, _gp
```

sp 的赋值就是初始化栈，栈是函数（或过程）调用的基础设施，也是使用 c 语言的前提。

```
    stack= start - 0x4000,
```

可见栈的大小为 16KB，不算小了。

_gp 在 ld.script 中定义，gp 的作用主要是加快数据的访问，这里不予关注。

```
        bal uncached            /* Switch to uncached address space */
        nop
    uncached:
        or    ra, UNCACHED_MEMORY_ADDR
        j     ra
        nop
```

先说说 bal 这条指令，这条指令是一个 pc 相对跳转指令(这一点很重要)，跳转的目的是把 ra 寄存器和 0xa0000000 相或，将 cpu 的执行地址映射到 unmapped uncached 段(对于一般的情形，这是多此一举，想想当前 pc 寄存器的值)。

由于地址问题是一个核心问题。下面解释一下龙芯 2F 地址转化过程。

CPU 获得的地址是虚拟地址，在获得这个地址之后，CPU 的内部逻辑首先判断这个地址落在哪个段，有的段虚拟地址到物理地址的转换公式是固定的，比如在 32 位模式下，kseg0，kseg1 就是直接映射的。无论如何，在获得转换后的地址也就是物理地址后，cpu 并不知道这个地址是发往哪里，是内存，pci，还是内部的 cpu 的 io 地址。这个发送方向也就是地址路由是由地址窗口寄存器决定的。在 2F 中，这些地址寄存器的地址是从 0x3ff0_0000 开始的 48 个寄存器。注意，这个地址是物理地址。

以 32 位的为例（这个比较简单），我们要关心的就只有 3 个地址窗口。

从 http://www.loongson.cn/company/下载的龙芯 2F 用户手册中介绍了这 3 个地址窗口。3 个地址窗口分别是：

cpu 窗口 0，负责物理地址到内存地址的转换。

cpu 窗口 1，负责物理地址到 pci 或者 cpu localio 的转换

pcidma 窗口 0，负责把收到的 DMA 请求进行 pci 总线地址到内存地址的转换。

比如 0xbfc00000 这个虚拟地址，cpu 首先判断这个地址落在 kseg1 段，直接转换且不经过 cache，转换成的物理地址 0x1fc00000。接着地址窗口寄存器就派上用场了。一个一个窗口去判断这个地址是否落在自己管辖的范围内（当然更有可能这个动作是并行的）。假设首先 cpu 窗口 0 去判断，它拿自己的 mask 寄存器去 mask 这个传入的地址，两个 cpu 窗口的 mask 都是 0xffffffff_f0000000。mask 后，0x1fc00000 就成了 0x1000_0000，这个地址是 cpu 地址窗口 1 的基地址，所以它就负责由 cpu 窗口 1 来路由。在传往下一个部件之前这个地址会被 cpu 窗口 1 包装一下，下面的就认识了。这个转换公式我没怎么看懂，当然也可能是作者文档没写清楚。有兴趣的可以看看。以上的地址转换在 cpu 核内，还每到本桥的层次。

下面结合代码解释 pmon 二进制的地址问题。

在链接阶段，使用了 ld.scripts 作为链接的规则。生成的 gzrom 程序地址是从 0xffffffff81000000 开始。ld.script 的片段如下：

```
ENTRY(_start)
SECTIONS
{

  . = 0xffffffff81000000;
  .text :
  {
    _ftext = . ;
    *(.text)
```

因此 gzrom（包括用 objcopy 生成的 gzrom.bin）的程序地址是 0xffffffff81000000 到其后的有限范围之内的。而在 cpu 执行这段代码之初，pc 寄存器的值是 0xffffffffbfc00000，而程序自己认为这条指令的地址是 0xffffffff81000000，在寻址方式为 pc 相对寻址或立即寻址时没有任何问题，在其他寻址方式时就会出现问题。比如在 pmon 拷贝到内存之后，现在要引用地址为 0xffffffff82000000 这个程序地址的内容，指令实质为 ld t0,0(0xffffffff82000000),问题就出来了，0xffffffff82000000 这个程序地址对应的段会使用 cache，而当前 cache 还没有初始化，cache 不可用，唯一的解决方法就是将这个地址转换到 0xffffffffa0000000 ~0xfffffffc0000000 这段空间内，且能映射到同一物理位置的地址，其实只要在原地址的基础上再加一个偏移值即可。L 223 的 lacate 的目的主要就是算出这个偏移值,具体计算如下：

```
locate:
    la    s0,start
    subu      s0,ra,s0
    and s0,0xffff0000
```

这里有个很合理的疑问，为什么链接规则文件 ld.script 中程序地址（start）要从 0xffffffff81000000 开始呢?直接搞成 0xffffffffbfc00000 不更简单吗？

下面分析分析这个选择的原因.从传统的 32 位地址空间的分布看，共有 4 个段：

0 ~0x80000000，

0x80000000~0xa00000000，

0xa0000000~0xc0000000

0xc0000000~0xffffffff,

首先第一个段和第四个段首先被排除，因为这两个段内的程序地址转化需要页表的辅助，现在连内存都不可用，不可能用页表。再要排除的是第三个段，用这个段内的地址，可以免去上面这个偏移值的麻烦，但是处于 rom 执行阶段在整个 bios 的执行过程中只占很少的一部分，99%以上的代码是要被载入内存执行的，就没有这个偏移值了，且在内存执行时，当然要尽可能使用 cache 加快执行速度，第三个段的地址引用不使用 cache。当然要提一句，如果不用 cache，使用第三个段作为程序地址也不是不可能的。

locate 除了计算以上说的那个偏移值到 s0 外，接着重新初始化状态寄存器和原因寄存器，这个代码应该是不必要的。

如果使用cpu自带的串口模块，则初始化这个模块。在显卡没有初始化的情形下，调试最好的方式就是使用串口。initserial代码和普通的串口除了地址，没有区别，具体可参看任何一本接口技术书籍，都有讲解这个NS16550兼容芯片的。
龙芯 cpu 的串口地址 COMMON_COM_BASE_ADDR 为 0xbff003f8,这里要说一句，其实这个 0xbff003f8 就是 0xffffffffbff003f8，因为龙芯 2 号系列 cpu 都是 64 位的，而且 mips 结构的 64 位 cpu 都是 32 位兼容的，32 位的地址都默认被符号扩展成 64 位的，前头的 0xbfc00000 其实和 0xffffffffbfc00000 也是一样的。
L 452~L 512的非定义语句都是作为初始化表，其实就是一些供特殊解释的数据。解释部分在 L 524~L 746。这部分和 cpu 相关性非常大，由于有些对旧 cpu（比如 2E）的支持代码仍然存在，部分代码对于龙芯 2F 是不需要的。
下面是对北桥的 pci 配置空间头部的初始化（大小为 256 字节）。从这个意义上说，北桥也是一个 pci 设备。北桥的 pci 配置空间头部首地址为 0xbfe00000(龙芯 2f 手册 P107 表 11-1 有误，将 pci header 和 registers 的地址互换了)，2F 的 pci 控制器遵循 pci2.3 规范。
接下是北桥代码的初始化，大致有十几行，一直到 EXIT_INIT(0)。
先别接看急着看代码，因为好些代码是非必须的，必须的代码为：

```
BONITO_BIS(BONITO_PCICMD,
        PCI_COMMAND_IO_ENABLE|
        PCI_COMMAND_MEM_ENABLE|
        PCI_COMMAND_MASTER_ENABLE)

BONITO_INIT(BONITO_PCIBASE0, 0)
BONITO_INIT(0x150,0x8000000c)
BONITO_INIT(0x154,0xffffffff)
```

实际就是两个功能，一个是设置 pci 总线操作的一些使能，后面的是设置 bar0。

```
BONITO_INIT(BONITO_PCIBASE0, 0)
```

设置 bar0 的基址为 0，没有什么特殊的。

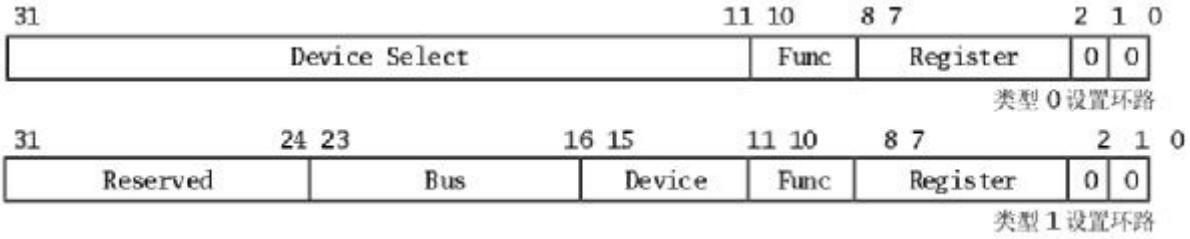
```
BONITO_INIT(0x150,0x8000000c)
BONITO_INIT(0x154,0xffffffff)
```

这两句设置了两个寄存器，名为 PCI_Hit0_Sel_L 和 PCI_Hit0_Sel_H。
2F 手册种中对这个寄存器的作用有介绍：
龙芯 2F 的 PCIX 控制器支持三个 64 位窗口，由{BAR1, BAR0}、{BAR3, BAR2}、{BAR5, BAR4}三对寄存器配置窗口 0、1、2 的基址。而窗口的大小、使能以及其它细节由另外三个对应寄存器 PCI_Hit0_Sel, PCI_Hit1_Sel, PCI_Hit2_Sel控制。

这样设置之后，bar0 就是从 0 开始到 0x8000_0000，64 位地址，可预取。而其它到 bar 由于没有使能，就不起作用。具体寄存器的位定义见手册。
L524~L746 为初始化表（reginit）的解释，代码比较显然。
L752~L 801 为 cpu 串口的初始化（如果使用的话）。

配置空间的访问

福珑盒子和逸珑笔记本都使用 AMD CS5536 的南桥，接着执行 superio_init。
由于下面的代码会涉及配置空间的访问，夹在南桥中间挺不爽的，就在这介绍吧。
访问嘛，肯定就有个地址的概念，所以先简要介绍 PCI 总线配置空间的寻址方式。
配置空间地址有两种格式，如下：



由于迄今为止，在 2F 使用到的 pci 体系结构中都没有超过一个 pci 总线的，所以我们可以以上图中的第一个作为理解设备地址的格式，总线域总为 0。
图中的 device select 的编码都是由连线决定的，比如主板设计出来后，cs5536 南桥的设备号 PCI_IDSEL_CS5536 就定下是 14。
好，看看配置读写的函数的实现。先看读函数 PCICONF_READW。
PCICONF_RWADW 这个宏的定义如下：

```
#define PCICONF_READW(dev, func, reg) \
        li    a0, CFGADDR(dev, func, reg); \
        li    a1, PHYS_TO_UNCACHED(PCI_CFG_SPACE); \
        and a2, a0, 0xffff; \
        or    a1, a2; \
        srl a0, 16; \
        li    a2, BONITO_BASE+BONITO_PCIMAP_CFG; \
        sw    a0, BONITO_PCIMAP_CFG(bonito); \
        lw    zero, BONITO_PCIMAP_CFG(bonito); \
        lw    a0, (a1);
```

上述这个宏中 CFGADDR 定义为：

```
#define    CFGADDR(idsel,function,reg)
                        ((1<<(11+(idsel)))+((function)<<8) +(reg))
```

就是根据上图的方式组合出 32 位的地址，作为要访问的配置空间的地址。
PCI_CFG_SPACE 值为 0x1fe80000，是 pci 访问配置空间的地址区域。
什么意思呢？就是所有对 pci 总线上设备的配置空间的访问都映射到这块地址区域。这块空间的大小为 2KB。这 2KB 大小够吗？看看上面那个图，device 以外的都在地址的低部分，刚好是 11 位，2KB 设计时就是按这个大小来的。
在龙芯 2F 的 pci 体系结构中，对 pci 配置空间的读操作顺序是：

1．把目的地址的高 16 位写入 PCIMAP_CFG 寄存器
2．对 1fe080000 开始的那 2KB 内的空间做读/写操作

3．从目的地址的低 16 位和 bfe80000 相或后的地址读内容

2F 手册 P108 的页尾描述介绍了这个过程。

BONITO_BASE+BONITO_PCIMAP_CFG 是 0x1fe00118，

```
sw    a0, BONITO_PCIMAP_CFG(bonito)
```

这句就是往这个 0xbfe00118 写那个 32 位地址的高 16 位。

```
lw    zero, BONITO_PCIMAP_CFG(bonito);
```

这句紧接这上一句，看得出来，load 个东西到 zero 寄存器就是不关心内容本身，这里只是要对 1fe080000 开始的那 2KB 内的空间做读/写操作而已，再紧接的 lw a0, (a1)是真正意义上的读，a1 寄存器放的是那个 32 地址的低 16 位。

这条指令执行完之后，我们要读的配置寄存器内容就在 a0 寄存器中了。

配置的写类似，就不讲了。

superio_init

superio_init 的代码在 L1966~L2271,这个代码的理解需要结合 AMD cs5536 手册的寄存器说明。

由于在代码中会涉及到对 msr 的访问，这里就先介绍一下，免得到时犯晕。

msr可以认为是外界眼中5536的寄存器的地址，实际上是一种特定格式的packet。用于指定将要访问的目标设备。

有关 msr 的地址先看 Targets/Bonito/include/cs5536.h

有如下定义：

```
#define CS5536_SB_MSR_BASE        (0x00000000)
#define CS5536_GLIU_MSR_BASE      (0x10000000)
#define CS5536_USB_MSR_BASE       (0x40000000)
#define CS5536_IDE_MSR_BASE       (0x60000000)
#define CS5536_DIVIL_MSR_BASE     (0x80000000)
#define CS5536_ACC_MSR_BASE       (0xa0000000)
#define CS5536_GLCP_MSR_BASE      (0xe0000000)
```

在南桥手册的 60 页有如下的映射关系：

```
GLPCI_SB   5100xxxxh
GLIU       5101xxxxh
USB        5120xxxxh
IDE        5130xxxxh
DD         5140xxxxh
ACC        5150xxxxh
GLCP       5170xxxxh
```

也就是有如下对应：

0x0000xxxx 5100xxxxh

USB

0x1000xxxx 5101xxxxh
0x4000xxxx 5120xxxxh
0x6000xxxx 5130xxxxh
0x8000xxxx 5140xxxxh
0xa000xxxx 5150xxxxh
0xe000xxxx 5170xxxxh

由于这个问题的特殊性，我们以 0x5120xxxxh 到 0x4000xxxx 为例，看看 AMD 的 cpu（不是龙芯 cpu）是如何访问南桥 msr 的。
先看到南桥手册 60 页的 Table 4-2 中对地址位的描述。
比如 AMD 的南桥要访问 USB 的 msr，会发送 0x5120xxxx 这种类型的地址，根据 Table 4-2 中的对地址高 9 位的描述：These bits are shifted off to the left and never enter the CS5536 companion device，也就是说，高 9 位的地址是会被丢弃的，不会传到 5536 上。
这个表中对于 bit[22:14]的描述为：These bits are shifted into posi-tions [31:23] by the time they reach the CS5536 companion device. Bits in positions [22:14] are always 0 after shifting，也就是说这 9 位会被 shift 到高 9 位，原 9 位为 0。
现在看下 0x5120xxxx 的变化。
1．丢掉高 9 位后，成了 0x0020xxxx
2．bit[22:13]左移，成了 0x4000xxxx
也就是说，虽然 AMDcpu 发出对南桥 USB 模块访问的地址是 0x5120xxxx，但是这个地址进入南桥时就成了 0x4000xxxx 了。
龙芯 cpu 不会在传递的过程中做任何处理，所以我们要访问南桥的 USB 模块的时候，就是发送 0x4000xxxx 的地址了。其它的地址处理类似。
msr 的介绍就先到这，以下为 superio_init 的代码：

```
LEAF(superio_init)
    // set the id select
    li    v0, 0xbfd00000;
    li    v1, PCI_CFG_BASE;//#define PCI_CFG_BASE        0x02000000
    sw   v1, 0(v0);
```

在 2F 的定义中，0xbfd00000（1fd00000）开始的 1MB 空间相当于传统 PC 的 io 空间，往 bfd00000 这个地址写不知道是干什么。

2:

```
    PCICONF_READW(PCI_IDSEL_CS5536, 0, 0x00);
```

PCICONF_READW 从命名上看就是读取 cs5536 的 0 号设备配置空间的 0 号寄存器。这个配置空间的读操作要传入的参数有三个：dev，func，reg。
PCICONF_READW 执行完之后，要读的内容已经放在 a0 寄存器上了。
事实上，南桥上功能号为 0 的就是南桥本身这个设备，pci 配置空间的第一个 word（mips 中的字都是 32 位的）就是厂商 id 和设备 id，读取后马上比较是否正确。
代码如下，0x208f 和 0x1022 就是设备 id 和厂商 id，代码如下：

```
    li    a1, 0x208f1022;
```

设备ID 厂商ID

```
    beq a0, a1, 55f;
    nop;
    b    2b;
    nop;
55:
    // set the msr enable
    PCICONF_WRITEW(PCI_IDSEL_CS5536, 0, 0xf0, 0x01);
```

cs5536 的配置头部的详细介绍在 AMD 南桥手册 P234，msr enable 见 P236。PCI 配置空间写的宏和读基本一样，只是最后的操作为写操作。手册的解释是

set to 1 to enable access to Model Specific Registers (MSRs)。

```
    // active all the ports
    CS5536_MSR_WRITE((CS5536_GLIU_MSR_BASE | 0x81), 0x0000ffff, 0x0);
```

不管如何，先看看 CS5536_MSR_WRITE 的定义：

```
    #define CS5536_MSR_WRITE(reg, lo, hi) \
        PCICONF_WRITEW(PCI_IDSEL_CS5536, 0, 0xF4, reg); \
        PCICONF_WRITEW(PCI_IDSEL_CS5536, 0, 0xF8, lo); \
        PCICONF_WRITEW(PCI_IDSEL_CS5536, 0, 0xFC, hi);
```

可见，这个宏的第一个参数是寄存器的地址，后两个参数组成一个 64 位数值的高低 32 位。南桥配置空间的 0xF4,0xF8,0xFC，分别是 pci msr 地址，低 32 位，高 32 位。具体解释见南桥手册 P237，P238。

CS5536_GLIU_MSR_BASE 这个常量的定义如下：

```
#define CS5536_SB_MSR_BASE          (0x00000000)
#define CS5536_GLIU_MSR_BASE        (0x10000000)
#define CS5536_ILLEGAL_MSR_BASE     (0x20000000)
#define CS5536_USB_MSR_BASE         (0x40000000)
```

所以这个 msr 的地址实际就是 0x10000081，想当于文档中的地址 0x51010081。参考 AMD P205，看到 0x51010081 是 port active enable reg，这个寄存器的初始值是 0x00000000_0000FFFF，后 16 位是 1 表示使能全部八个端口。这八个端口从 0 到 7 分别是 GLIU，GLPCI_SB，USB，IDE，DD，ACC，保留，GLCP。

跳过 test 代码。

```
    CS5536_MSR_WRITE((CS5536_SB_MSR_BASE | 0x10), 0x00000003,
                                                    0x44000030;
```

这个值是初始值 0x44000030_00000003，后两位的 3 表示 mem/io 写使能，其余的表示设置 flush，read，timeout。具体解释见 AMD 手册 229 页。

```
    CS5536_MSR_WRITE((CS5536_DIVIL_MSR_BASE | 0x0b),
                                SMB_BASE_ADDR, 0xf001);//AMD 357 页
    CS5536_MSR_WRITE((CS5536_DIVIL_MSR_BASE | 0x0c),
                                GPIO_BASE_ADDR, 0xf001 );
    CS5536_MSR_WRITE((CS5536_DIVIL_MSR_BASE | 0x0f),
```

PMS_BASE_ADDR, 0xf001);

上面的几行代码的性质相同，从参数的命名就可以看出来，是设置 io space 的起始地址(也就是说这个起始地址是我们主动写入的，这些地址会在 pci 设备的初始化之后重新分配)。具体解释见 AMD 手册 P357，这些地址的定义如下：

```
#define DIVIL_BASE_ADDR         0xB000
#define SMB_BASE_ADDR           (DIVIL_BASE_ADDR | 0x320)
#define GPIO_BASE_ADDR          (DIVIL_BASE_ADDR | 0x000)
#define MFGPT_BASE_ADDR         (DIVIL_BASE_ADDR | 0x280)
#define PMS_BASE_ADDR           (DIVIL_BASE_ADDR | 0x200)
#define ACPI_BASE_ADDR          (DIVIL_BASE_ADDR | 0x2c0)
```

后面那个 0xf001，0xf 是 io mask，1 表示 enable LBAR。

南桥串口的代码跳过。

接下来是南桥的 ide 部分，自带的中断控制器 8259 和开机键的初始化。都是 GPIO 的设置，说重要吧也重要，不过太多了，跳过。

下面是 SMBus 的初始化，这个是非常重要的。等会探测内存的 spd，就是通过 SMB 总线。所以这个初始化是必须的。具体代码略过。

略过 gpio，flash 和 usb 部分的测试。

以上是南桥相关的初始化，接着看代码。

如果定义了 TEST_CS5536_SMB,则执行 L814~L845，默认不执行。但理解这个代码对于了解 smb（i2c）的访问机制是很有好处的。所以还是委屈各位方便的话看看吧。

start.S 之内存

L855~L871 是些打印语句，也不知道有什么必要。

接下来要执行的是 L929，L929~L940 实际上就是一下几句：

```
    li   t2, 0xbfe00180   //Chip_config0
    ld   a1, 0x0(t2)
   and a1, a1, 0x4ff
```

前 21 位保留，bit9－10 使能 pci 和 cpu 写内存写到内存，bit8 禁用 ddr2 配置

```
    sd   a1, 0x0(t2)
```

bfe00180 对应的是 chip_config0，具体位功能定义见 2F 手册 P116。

L 942　li　msize, 0x0f000000

这句没有作用，在 L979 被重新初始化了，可以去除。

L944 调用 ddr2_config ,初始化内存控制器

```
    bal  ddr2_config
    nop
```

ddr2_config 的代码在 start.S 文件的底部，是 L2402~L2657

```
L2411   move s1, ra     // 保存返回地址
L2413   la    t0, ddr2_reg_data // 取初始化数据的首地址
L2416   addu    t0, t0, s0
            li          t1, 0x1d  //1d 就是 29
            li          t2, CONFIG_BASE
```

这个 ddr2 的初始化表和前头的 reginit 类似，但有一点明显不同，那就是这个地址要加个偏移值（L2416）。原因是 la t0, ddr2_reg_data 这条指令执行后，t0 的值是 0xffffffff8100xxxx，不能直接使用这个地址，原因在前头已经讲述。所以在 L2416 要加上这个偏移值。

2F 手册 P97 页有句话“具体的配置操作是对物理地址 0x000000000FFFFE00 相对应的 29 个 64 位寄存器写入相应的配置参数”，足以解释 L2417 和 L2418 代码的作用。

L2465~L2470

```
    ld      a1, 0x0(t0)
    sd      a1, REG_ADDRESS(t2)  //注意是 sd，8 个字节
    subu    t1, t1, 0x1
    addiu   t0, t0, 0x8
    bne t1, $0, reg_write
    addiu   t2, t2, 0x10            //t2 是地址，每个配置寄存器是 16 个字节
```

以上代码是循环些写完 29 个配置寄存器。

略过 debug 代码，执行到 L2500.

```
L 2500    move      k0 , ra   //不知道这个 ra 值的保存有什么意义。
L 2501 /* read spd to initialize memory controller */
            li a0, 0xa0
            li a1, 3      //Row Address
            bal i2cread
            Nop
```

L2501 的注释已经说明了以下代码的功能是读取 spd 信息初始化内存控制器了。0xa0，0xa1 作为 i2cread 的传入参数。

这里有一个问题，内存上的 spd 的地址是多少？

从代码上看是 0xa0 啦。 网上搜了很久，有网友有这样的帖子：“memory SPD EEPROM 是走 SMBus 总线，SMBus 总线下的设备是通过 SlaveAddress 来标识，一般 Slot 0 上的 memory SPD EEPROM 的 SlaveAddress 为 0x0A0，Slot 1 上的 memory SPD EEPROM 的 SlaveAddress 为 0x0A2，你可以向 SMBus 控制器发送命令来读取 SPD 数据。读 deviceID 要+1, 所以是 0xa1,0xa3,0xa5，如果是 write 的话，就是 0xa0,0xa2,0xa4”。

我看了几个内存的手册，都没有看到关于 spd 的 i2c 地址的约定。但是我曾经看过一个华邦的时钟芯片手册，手册中就明确写了 i2c 的地址。
实际上设备地址的最后一位为 1 表示读操作，为 0 表示写操作。下面看看 i2cread 是如何使用这两个传入参数的，i2cread 的代码在 L2311~L 2394,
开始将 a0 和 a1 转存到 t2 和 t3 上，之后：

```
    /* start condition */
    IO_READ_BYTE(SMB_BASE_ADDR | SMB_CTRL1);
    ori v0, SMB_CTRL1_START;
    IO_WRITE_BYTE(SMB_BASE_ADDR | SMB_CTRL1);
    IO_READ_BYTE(SMB_BASE_ADDR | SMB_STS);
    andi      v0, SMB_STS_BER;
    bnez      v0, i2cerr;
    nop;
    SMBUS_WAIT;
    bnez      a3, i2cerr;
    nop;
```

开头的注释已经说明了这段的功能，就是启动 smbus 工作。

```
/* send slave address */
    move      v0, t2;
    IO_WRITE_BYTE(SMB_BASE_ADDR | SMB_SDA);
    IO_READ_BYTE(SMB_BASE_ADDR | SMB_STS);
    andi      v0, (SMB_STS_BER | SMB_STS_NEGACK);
    bnez      v0, i2cerr;
    nop;
    SMBUS_WAIT;
    bnez      a3, i2cerr;
    nop;
```

这里要注意到 v0 是作为 IO_WRITE_BYTE 的传入参数使用的

```
    /* acknowledge smbus */
    IO_READ_BYTE(SMB_BASE_ADDR | SMB_CTRL1);
    ori v0, (SMB_CTRL1_ACK);
    IO_WRITE_BYTE(SMB_BASE_ADDR | SMB_CTRL1);

    /* send command */
    move      v0, t3;
    IO_WRITE_BYTE(SMB_BASE_ADDR | SMB_SDA);
    IO_READ_BYTE(SMB_BASE_ADDR | SMB_STS);
    andi      v0, (SMB_STS_BER | SMB_STS_NEGACK);
    bnez      v0, i2cerr;
    nop;
    SMBUS_WAIT;
    bnez      a3, i2cerr;
```

```
    nop;
```
发送命令的下一部分，这个是 index，传入参数是 t3，就是开始的 a1。
```
    /* start condition again */
    IO_READ_BYTE(SMB_BASE_ADDR | SMB_CTRL1);
    ori v0, SMB_CTRL1_START;
    IO_WRITE_BYTE(SMB_BASE_ADDR | SMB_CTRL1);
    IO_READ_BYTE(SMB_BASE_ADDR | SMB_STS);
    andi      v0, SMB_STS_BER;
    bnez      v0, i2cerr;
    nop;
    SMBUS_WAIT;
    bnez      a3, i2cerr;
    nop;
```
重新初始化，这是 smbus 的约定。
```
    /* send salve address again */
    move      v0, t2;
    ori v0, 0x01;
    IO_WRITE_BYTE(SMB_BASE_ADDR | SMB_SDA);
    IO_READ_BYTE(SMB_BASE_ADDR | SMB_STS);
    andi      v0, (SMB_STS_BER | SMB_STS_NEGACK);
    bnez      v0, i2cerr;
    nop;
    SMBUS_WAIT;
    bnez      a3, i2cerr;
    nop;
```
上一页提到的，读的话 deviceID 要加一，就是 ori v0, 0x01 起到了加一的作用。
```
    /* stop condition */
    li   v0, SMB_CTRL1_STOP;
    IO_WRITE_BYTE(SMB_BASE_ADDR | SMB_CTRL1);
    SMBUS_WAIT;
    bnez      a3, i2cerr;
    nop;

    /* read data */
    IO_READ_BYTE(SMB_BASE_ADDR | SMB_SDA);
```
IO_READ_BYTE 的定义如下：
```
    #define IO_READ_BYTE(reg) \
        lui v1, 0xbfd0; \
        ori v1, reg;      \
        lbu v0, 0(v1);
```
可见是把读的结果放到 v0 中，这样至此我们就通过 smbus 的协议把要读的结果读到了 v0 寄存器中，随后返回。

```
jr    ra;
nop;
```

具体的 smbus 相关的约定参看 smbus 总线定义。

回到我们调用的入口，就是要将 smbus 总线上（相对）地址为 a0，index 为 3 的内容读到 v0 寄存器中。spd 的相关信息可参看 kingmax 内存的手册 P18~P21,先介绍一下内存控制器需要的几个参数的含义吧。

代码中读取的参数共有 4 个，index 分别是 3,4,5,11h（17）。以 kingmax ddr-667 512MB 内存为例解释读出的各参数的含义。(kingmax 手册 P20)

index 为 3 读出的值为 0Dh，表示 row address 是 13，index 为 4 读出的值为 0Ah，表示 col address 是 10，这两个读出的参数经过简单的转化，写入内存控制器的一个寄存器（程序地址为 0xafffe50），具体见 2F 手册 P99 对这个寄存器的解释。

index 为 5 读出的值为 60h，表示可选的内存 chip 为 1。这个寄存器解释见 2F 手册 P100。

下面是 L2550 ~L 2562 的代码，其中 bne 一句表示读出的 bank 值为 8 的话，就重新设置 affffe10 这个地址的寄存器，具体见 2F 手册 P98 对这个寄存器 bit32 的定义。可见 8bank 的情形可能是比较特殊的。

```
/* Number of banks */
li    a0, 0xa0
li    a1, 0x11 //Bank
bal i2cread
nop
//li    v1, 0x4
//beq v1, v0, 1f
li    v1, 0x8
bne v1, v0, 1f
nop
li    v1, 0x1
li    v0, 0xafffe10
sw    v1, 4(v0)   //注意地址
```

不管如何，内存的配置就这样完成了，下面发个信号告诉内存控制器配置完成。

```
############start##########
li          t2, CONFIG_BASE
la    t0,DDR2_CTL_start_DATA_LO
#la t0,ddr2_start_reg
addu        t0, t0, s0
ld          a1, 0x0(t0)
sd          a1, 0x30(t2)
```

内存控制器的初始化完成后，打印个信息表示一下。代码回到 L949, L951~L962 是内存只读检测。

L 966~L972

```
#if 1
```

```
        TTYDBG    ("\r\ndisable register space of MEMORY\r\n")
            li    t2, 0xbfe00180
            ld    a1, 0x0(t2)
            or    a1, a1,0x100
            sd    a1, 0x0(t2)
#endif
```

由于内存控制器的配置已经完成，L966~L972 代码设置从现在起禁用 DDR2 配置端口，以免意外。具体见 2F 手册 P116。

其实内存的工作还没有做完，前面 ddr2_config 只是做了内存颗粒的行列地址数，bank 数，并没有得到具体的内存大小数据。

L 979

```
/* set the default memory size : 256MB */
li    msize, 0x10000000;
```

注释说设定默认内存大小是 256MB，这样具体有什么作用呢？无论如何都在 L1009 被重新赋值。

L982 开始读取 spd 上的 index 为 31 的内容，还是前面 kingmax DDR2-667 512MB 内存为例。读出值为 80h。

```
        /* read DIMM memory size per side */
        li    a0, 0xa1;
        li    a1, 31;
        bal i2cread;
        nop;
        beqz        v0, .nodimm;
        nop;
        sll a0, v0, 2
        la    a1, bigmem
        addu a1, a1, s0
        addu a1, a1, a0
        lw    tmpsize, (a1)
        li    a0, 0x10
        blt v0, a0, 2f
        nop
        sll tmpsize, v0, 22;                // multiply by 4M
```

tempsize 存储相应的预计内存大小的值，但正如我们的例子，读出的是 80h，左移 22 位，就是 128x4MB=512MB,参考文档中还有 hynix512MB 的内存 spd 表，也可参考之。这段代码之后 tmpsize 为 2^29.，就是 512MB。其中：

```
        la    a1, bigmem
        addu a1, a1, s0
        addu a1, a1, a0
```

这几句话的作用不详。

接着执行到 L999，

```
2:   /* read DIMM number of sides (banks) */
     li    a0, 0xa1;
     li    a1, 5
     bal i2cread
     nop
     bltu       v0, 0x60, .nodimm;
     nop;
     subu       v0, 0x60;
     sll tmpsize, v0;
```

这个内容先前已经探测过了，这个返回的数据如果小于 60h，表示这个内存是 DDR 内存或者内存颗粒不存在。如果返回值是 61h，表示有两边，内存容量加倍。比如 kingmax 文档中所写的 1G 内存的 spd，index31 的数据为 80h，与 512MB 的数值相同，但 index 为 5 的数值 1G 的为 61h，512MB 的为 60h，可见这两个数据就可确定了一条内存的容量。至此，spd 就再没什么用了。

start.S 之 cache

接着执行到 L1033~L1039,功能就是把 bfe00108 的 bit5~bit2 设成 00011b，设定 rom 数据读取延迟。L1041～L1046 设定 io 数据读取延迟，具体见 2F 手册 P114。（为什么不合并这两个操作呢？？？）

L1060~L1069 的代码通过对比 2E 北桥文档和 2F 文档及 bonito.h 文件，基本确定属于 2E 的遗留代码。

ok，内存控制器的初始化算是完毕了。接下来是 cache 的初始化。

由于在上电之后，cpu 的 cache 处于不确定的状态。

cache 初始化后，速度会快很多，下面看看 cache 是怎么初始化的吧。

cp0 的 config 寄存器指明了 cache 的基本情况，config 的位定义如下：

31 18	17	16	15 12	11 9	8 6	5	4	3	2 0
0	1	1	0	IC	DC	IB	DB	0	K0
14	1	1	4	3	3	1	1	1	3

图 5-15 Config 寄存器

IC　表示一级指令 cache

DC 表示一级数据 cache

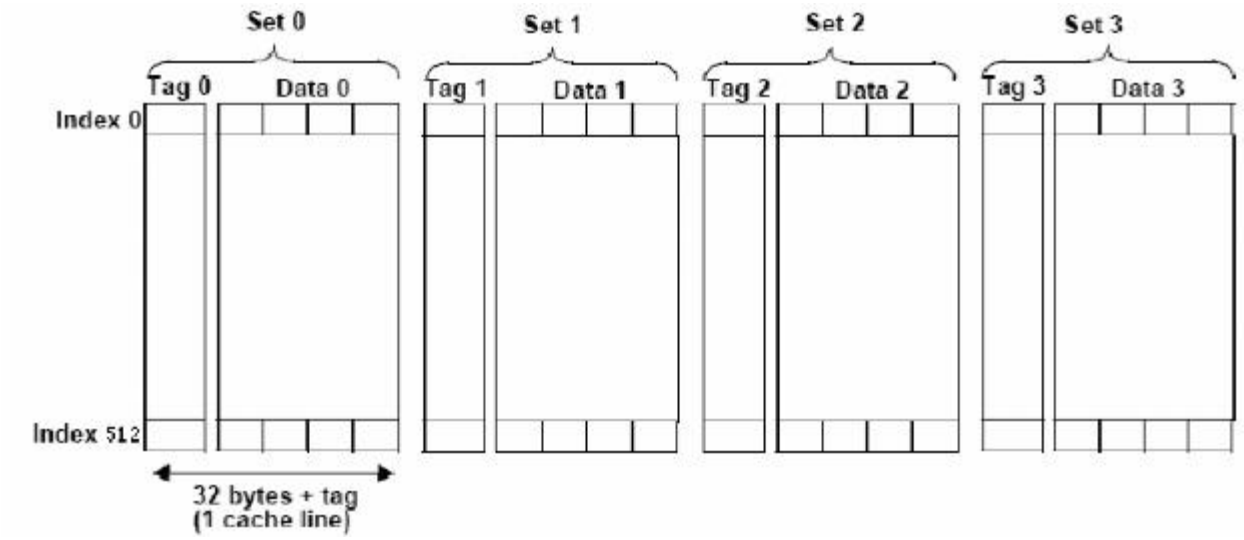
IB　表示一级指令 cache 行大小

DB 表示一级数据 cache 行大小

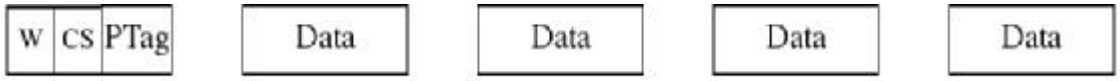
K0 确定 cache 的一致性算法

具体见 2F 手册 P49。
龙芯 2F 采用的是两级 cache 的结构，其中一级 cache 包括 64KB 的指令 cache 和 64KB 的数据 cache，二级 cache 为 512KB（不区分数据与指令）。

龙芯的一级数据和指令 cache，二级 cache 都是 4 路组相连的，cache 的组织有必要说明一下，对理解代码是必须的。
以一级指令 cahce 为例（指令 cache 和数据 cache 的 tag 有不同），示意图如下：



注：这个图是从龙芯手册粘贴的，其中的 index 512 应为 511。
其中每个 set 大小为 16KB，每个 set 内部有 512 个 cache 行，每个 cache 行的大小为数据 32 字节，指令 cache 的 tag 为 29 位，其中 1 位是有效位，物理地址标记 ptag 为 28 位。数据 cache 的组织基本与这相似，由于数据是不像指令是只读的，所以 tag 中多了个状态位，其定义如下：



其中 1 位 W 表示已经写回，2 位 CS（cache state）指示 cache 的状态，如下：

00 ——— 无效
01 ——— 共享
10 ——— 独占
11 ——— 脏

龙芯的 cache 以及 TLB 与 x86 的不同，对系统程序员不是透明的，所以 cache 的组织是需要软件人员处理的，用的武器就是 cache 这条指令。cache 指令的详细说明在 2F 手册 8.2.6。
指令和数据的 cache 都是 28 位的地址位，在龙芯 2F 的体系中，物理地址实际都是 40 位的，tag 中的是高 28 位，低 12 位和虚拟地址相同。

接着看代码到 L1125，对照 2F 的手册，明显感到这个代码与手册有一定差别，很怀疑 L1125~L1129 的必要性。
再到 L1131，接下是算 IC 和 DC 的大小，IC 计算公式为 size＝2^（12＋IC），DC 类似。龙芯 2F 的一级指令和数据 cache 均为 64KB，IC/DC 值为 100b，也就是 4，但看这个代码，意图是报指令 cache 的大小存进 s3，但是计算方法有误。事实上，在 pmon 后面的代码中又对这个 cache 进行了测试，这里的探测没有实际意义。一级指令 cache 行大小的计算公式为 IC 等于 1，则为 32Byte，为 0 则为 16Byte，DB 计算类似。L1153 和 L1154 的计算方式很巧妙。之后的是数据 cache 的计算。
无论如何，现在找到了 cache，要去初始化这些 cache 了，这个工作是通过调用 godson2_cache_init 实现的。代码在 L1693。
L1694 到 L1705 的功能是检测一级指令和数据 cache 的大小到 t5 和 t6 寄存器中，可是这段代码实在是比较烂。不说了，不过从中可看出，这里又检测了一次，故前面的 cache 大小的检测是没有意义的。
看代码到 L1717~L1720，看代码的命名和注释目的是初始化一级数据 cache：

```
1:  slt a3, v0, v1
    beq a3, $0, 1f
    nop
    mtc0    $0, CP0_TAGLO
    cache   Index_Store_Tag_D, 0x0(v0)
    mtc0    $0, CP0_TAGLO
    cache   Index_Store_Tag_D, 0x1(v0)
    mtc0    $0, CP0_TAGLO
    cache   Index_Store_Tag_D, 0x2(v0)
    mtc0    $0, CP0_TAGLO
    cache   Index_Store_Tag_D, 0x3(v0)
    beq $0, $0, 1b
    addiu   v0, v0, 0x20
```

这里有一个虚地址（VA）的概念，比如上面这条指令，当 v0 为 0x80000000 是，这条指令的虚地址为 0x80000000＋0x0，这个虚地址是有讲究的，地址的低两位表示要操作的路（way，4 路组相连的那个’路’），代码中 v0 都是每次加 0x20，所以前面代码中 cache 指令中逗号之后的数字就是路号。之所以要加 0x80000000 是因为这个地址是 uncache 的，否则会有异常产生。
Index_Store_Tag_D 这个命名很明显就是要把 TAGHI 和 TAGLO 中的数据按照一定的方式填充 cache。TAGHI 和 TAGLO 的位定义 2F 手册 5.20。这里 cache 指令目的就是设置这些cache的内容无效。代码中每次cache后都要重新设置TAGLO，不知何故，约定？？？
有一点要提的是 Index_Store_Tag_D 的定义为 0x5（00101b），反汇编中也很明显，但 2F 手册的 P89 中这个数值对于 Index Load Tag Dcache 的操作码，Index Store Tag Dcache 的操作码是 0x9（01001b），查看了 mips64 的 cache 指令编码定义，如果 2F 遵守 mips64 的规范，则 2F 手册的解释应该是对的。代码的作者疏忽了。但居然还能用，奇怪了。

接下来初始化 L2 cache，大小是代码中写死的，2F 手册中也没有讲哪个寄存器是标志二级缓存大小的，只能写死。每路 128KB，也是 4 路。这个 Index_Store_Tag_S 的操作码和 2F 手册是一致的。
接下来的代码将二级缓存和指令 cache 每个 cache 行当前内容置为无效。（为什么没有数据 cache 的无效设置）
至此，godson2_cache_init 这个过程执行完毕。
代码返回后，执行 L1171。L1171 和 L1172 功能不详，可能是 2E 遗留代码。
L1174~L1177 功能是置 config 寄存器 bit1 为 1，bit0 为 0，bit12 为 0。这些代码设置的位可能也是有些问题，比如 bit12 在 2Fcpu 中是写死的，bit0，bit1，bit2 这 3 个位合在一块才能确定 k0 的 cache 一致性算法。这里只写两位，有点不伦不类。估计是前些版本的遗留代码。
好了，cache 就算这么初始化了，看看下面要干些什么。
这里先缕缕思路。我们初始化内存的目的是使用内存，初始化 cache 的目的是用 cache 提高内存使用效率，现在基本设置都差不多了。该准备搬 rom 中的 bios 代码到内存上来了吧。
在搬到内存前可以测测内存的稳定性，方法就是往 0xa0000000 开始的 1M 空间写后读，看前后内容有没有差异，这个地址也不是随便取的。
内存测时候，代码执行到了 L1257，一句“Copy PMON to execute location”告诉我们拷贝 bios 到内存的行动已经万事俱备了。
跳过 debug 部分，到了 L1278。

```
la    a0, start
li    a1, 0xbfc00000
la    a2, _edata
or          a0, 0xa0000000
or          a2, 0xa0000000
subu        t1, a2, a0
srl t1, t1, 2
```

以上的代码功能比较清楚，设置各个寄存器
A0　pmon 的起始程序地址
A1　pmon 的起始物理地址，对应程序地址 a0
A2　pmon 末尾地址_edate
T1　pmon 的实际大小（以字为单位，拷贝的时候用的是 lw 和 sw 指令）
值得一说的是 or...，0xa0000000 这句的作用。此时，cache 仍不能使用，因为 TLB 还没有初始化，所以仍要避免使用 cache 的程序地址的使用，a0 既是 pmon 的起始程序地址，作者又将其作为在内存中的物理地址使用（or 指令之前的那个地址值），这个使用方式是十分正确的。比如假设 start 这个标号的地址是 0x81001234，那么作者希望它搬到内存物理地址为 0x01001234 的位置，
后面的代码有点意思：

```
move        t0, a0
move        t1, a1
move        t2, a2
```

前面算好的 t1 被覆盖掉了，呵呵。

至此，可以这么理解了，t0 表示复制的目的起始地址，t2 是复制的目的结束地址，t1 是复制的源地址。
L1292~L1307 是复制的具体代码，是指就是一个循环，每次复制一个字，标号 1 到 2 之间的代码是用于调试目的的。
之后清零 bss 段所占的空间，之后又是一段 test 代码，跳过。
程序此时执行到了 L1368，看注释可以猜测 ，TLB 的初始化就要开始了。在 TLB 初始化前我们只能使用的程序地址都是在 0xa0000000～0xc0000000（64 表示的话就符号扩展）之间。这个 TLB 的初始化时机是有讲究的，不一定说什么时候最好。但 TLB 是很小的，如果在复制操作之前使能 TLB，是不是有意义？？？可能白白地看着 TLB 被迅速填满又马上被替换。
回到 TLB 初始化的具体代码。
L1373~L1374 是设置 L3 大小，对于 2F 无效，这两句代码当舍去。
接下的是设置 a0,a1,a2 这三个寄存器。
A0 －－－ 内存大小（单位 MB）
A1 －－－ tgt_putchar 函数的在 rom 中地址
A2 －－－ stringserial 函数的在 rom 中地址
至于 a1 和 a2 的地址为什么这样设置，我暂时没有想明白。

```
la    v0, initmips
jalr      v0
nop
```

以上三句，是 cpu 从 rom 取指令到从内存取指令的转变，从此内存执行的时代开始了。jalr v0 的操作是 PC<--v0,不是 bal 那样的 PC＋offset。initmips 的地址是 0x8 开头的，并不使用 TLB，是否使用 cache 要看 config 寄存器的低 3 位，前面说到了，程序只设置了低两位为 1，0，（非常不严谨的做法），底 3 位只有 3 种情况有效，111，011，010，所以就算第三位是 010 了，设置不使用 cache。
到这我们可以明白了，那个 TLB 的注释是挂羊头卖狗肉，根本就没有什么初始化，和程序开始的时候那个 16 条指令的限制一样，都是玩笑。
真正的 TLB 初始化在哪里呢？？？？？？？？？？？？？

从汇编到 c

这个 initmips 在哪里呢？看过一些文档都说是在 tgt_machdep.c 中，我一开始也是这么认为的，但看看前面的 a0,a1,a2 显然都是作为参数列表使用的(去看看 o32 的参数传递规则吧)。那个文件中的 initmips 只有一个参数呀，实际上这个函数的定义在 initmips.c 中，这个文件是在编译过程中产生的。在我写的《pmon 编译过程分析》中写到：“initmips.c 的生成规则对应 compile-output 的 line 710”，看看 compile-output 文件中的 line710，看到了吧
./genrom ../Targets/Bonito/compile/Bonito/pmon > initmips.c

genrom 是一个用 perl 语言写的脚本，我不懂这个语言，但这个脚本很短，基本的意思也很清楚。

initmips 函数定义在 initmips.c 中 L66，当初我是怎么发现这个函数的呢？在当时这个函数的 L72 和 L78 的打印语句还没有被注释掉，我在通过串口（显卡初始化前只能用串口输出）做 pmon 的实验的时候发现中间的“Uncompressing Bios”这一句在不知道什么时候打印出来的，根据大部分文档的解释，直接就跳到 tgt_machdep.c 中，就不可能有这一句的打印。就这样找到了这个 initmips.c 中了。当然通过反汇编看跳转的地址其实一下就看出来了。

总算看 initmips 的具体代码了，也总算是看 c 代码了。

开始几句的地址值是在 perl 脚本中得到的。

L73　　if(!debug||dctrl&1)enable_cache();

这个 debug 的值正常情况下为 0（因为 msize 不为 0），也就是 enable_cache 函数要执行。但是有个问题，这个 dctrl 是什么东西?这个只是通过形式参数传入的，按照 o32 的标准也就是 a0 对应函数的第一个形参（从左往右数，）依此类推。a2 对应的就是第三个参数。这个 initmips 的函数定义是 initmips(unsigned int msize, int dmsize,int dctrl)，也就是 a2 对应的就是 dctrl，我们肯定还记得，调用 initmips 前给 a0,a1,a2 这三个寄存器赋了值，a0 是内存大小（单位 MB），a1 是 tgt_putchar 函数的在 rom 中地址 ，a2 是 stringserial 函数的在 rom 中地址，对照这个函数的定义，第一个参数肯定是对的，当后面的两个就牛头不对马嘴了。我是又看不懂了。

用 minicom 看使用串口的输出是比较简单的。有些想法也可以修改源码，通过串口输出来验证。minicom 的简单使用如下：

```
apt-get install minicom
minicom -s   //设置端口为/dev/ttyS0 即可
插上串口线，
minicom
```

下面就通过 minicom 看看这个 dctrl 的值。在 int debug=(msize==0)这一行之后加上一行 hexserial(dctrl);即可

copy text section done.

Copy PMON to execute location done.

sp=80ffc000bfc010b8

红字加粗的部分就是了，可见的确是 stringserial 函数的在 rom 中地址.所以这个命名和 if(!debug||dctrl&1)enable_cache()这句的条件判断部分就挺怪的了。可能只有作者能解释了这个 dctrl 的作用了。

无论如何，要去执行 enable_cahe 这个函数了。enable_cache 函数在同一个文件中，采用的是内嵌汇编。这段代码就是简单的把 config 寄存器（cp0 的 16 号寄存器）低 3 位设成 011（3），即 cachable。从此 kseg0 段（0x8，0x9 开头的）开始使用 cache 了。

接下来执行

```
while(1)
{
    if(run_unzip(biosdata,0x80010000)>=0)break;
```

}

从名字就可以看出来是在解压。biosdate是一个数组，内容在zloader/pmon.bin.c中。我第一次打开pmon.bin.c文件是差点昏过去。pmon.bin.c文件的生成见compile-output文件：

```
gzip ../Targets/Bonito/compile/Bonito/pmon.bin -c > pmon.bin.gz
./bin2c pmon.bin.gz pmon.bin.c biosdata
```

bin2c也是个perl脚本，就是把pmon.b in.gz的内容转化成biosdata这个数组，否则不方便集成到代码中。

这个解压的代码就不深入了，要知道压缩软件工作方法的可以看看。这个函数的作用就是把biosdata的数组加压到0x8001000开始的地址上去。当然这个biosdata的内容究竟是什么呢？我们是一定要探究的。biosdata来自于pmon.bin.gz，pmon.bin.gz来自于pmon.bin ，那么pmon.bin是哪儿来的呢?由于gzrom.bin是来自于objcopy后的gzrom,这个pmon.bin很可能是来自于名为pmon的elf文件。而且./Targets/Bonito/compile/Bonito/下正好有个叫pmon的elf文件，无疑了。问题来了，这个pmon文件哪儿冒出来的呢！

看compile-output L703

```
mips-elf-ld -EL -N -G 0 -T../../conf/ld.script -e start -S -o pmon ${SYSTEM_OBJ} vers.o
```

看到了吧，其实这个pmon几乎把所有代码都编进去了。包括start.o（虽然个人认为没有必要），不过这里的ld.script不同而已，是代码段地址从0xffffffff80010000开始分配。个人推断，这个pmon用objcopy后的pmon.bin是可以直接使用的，只是没有压缩。有兴趣时试试。

实际上./Targets/Bonito/compile/目录只是一个中转站。回到initmips这个函数。

接下是bss段的初始化，就是清零。

接着调用flush_cache2()，就是把二级缓存都置为无效。如果记性好的话，前面已经做了同样的工作了。

```
godson_2f:
    li      $2, 0x80000000
    addu    $3,$2,512*1024
10:
    cache   3, 0($2)
    cache   3, 1($2)
    cache   3, 2($2)
    cache   3, 3($2)
    addu    $2, 32
    bne     $2,$3, 10b
```

不过前面的那个512*1024改成了128*1024而已，哪个才是对的呢？个人认为是前者。还有2F的prid手册标称是0x6302，与实际不附。唉。

initmips函数最后调用realinitmips这个函数，从名字看出，这个realinitmips才是大家都认为的initmips。genrom脚本中获取了真实的initmips的地址，这个函数还初始化sp寄存器，栈大小为16KB（0x4000），并把msize作为参数传递给那个initmips函数，然后跳转到那个函数。

从现在开始执行的就是那个叫 pmon.bin 的文件的内容了，看反汇编也要看 Targets/Bonito/compile/Bonito/下那个 pmon 文件的反汇编了，从此程序地址都不再用 0x81 开头的了，都用 0x8001 开始的了。

initmips

首先执行的函数是 initmips，在 tgt_machdep.c 文件中。
对于 512MB 的内存，设置 memorysize 为 256，memorysize_high 为 256，这个 memorysize 是 pmon 中能使用的内存大小，pmon 是不需要那么多内存。
接下来执行 tgt_cpufreq()函数。实际上是执行_probe_frequencies 函数探测 cpu 的频率。这个函数也在同一个文件中。

```
_probe_frequencies()
{
        .。。。。。。。
          SBD_DISPLAY ("FREQ", CHKPNT_FREQ);

          md_pipefreq = 300000000;          /* Defaults */
          md_cpufreq   = 66000000;
          。。。。。
```

SBD_DISPLAY 这个函数就是一个打印函数，并在代码中写死只打印 4 个字符，这个限制可以通过在那个函数中写个 for 循环解除，后面的那个参数没有使用。
看到这有两个 freq，一个是 pipefreq，一个是 cpufreq，暂时先不解释，看了代码就清楚了。HAVE_TOD 在../conf/GENERIC_ALL 文件中定义。
接着调用 init_legacy_rtc(),这个函数在 tgt_machdep.c 中定义。
这个函数是对实时钟的操作。PC 中通过 I/O 端口 0x70 和 0x71 来读写 RTC 芯片的寄存器。其中端口 0x70 是 RTC 的寄存器地址索引端口，0x71 是数据端口。这个函数的意图就是读取当前的时间（人理解的格式，年月日时分秒）。
接着执行一个 for 循环

```
/*    Do the next twice for two reasons. First make sure we run from
 *    cache. Second make sure synched on second update. (Pun intended!)
 */
    for(i = 2;    i != 0; i—) {
            cnt = CPU_GetCOUNT();
            timeout = 10000000;
            while(CMOS_READ(DS_REG_CTLA) & DS_CTLA_UIP);

            sec = CMOS_READ(DS_REG_SEC);

            do {
```

```
                timeout--;
                while(CMOS_READ(DS_REG_CTLA) & DS_CTLA_UIP);
                cur = CMOS_READ(DS_REG_SEC);
        } while(timeout != 0 && cur == sec);

        cnt = CPU_GetCOUNT() - cnt;
        if(timeout == 0) {
                break;          /* Get out if clock is not running */
        }
    }
```

假如单看 for 循环内部的话，这个代码的功能就是看 1 秒钟 cpu 的 count 寄存器变化的值。这里先介绍这个 cpu 内部的 count 寄存器。这是个 32 位的一个计数器，每两个 cpu 时钟周期加一次，假设 2F 频率为 800MHz 的话,每 10.74s,即（2^32x2)/ (800x10^6）溢出一次。 CPU_GetCOUNT 函数就是去读取这个寄存器的值而已，代码定义在./pmon/arch/mips/mips.S

```
LEAF(CPU_GetCOUNT)
    mfc0    v0, COP_0_COUNT
    nop
    j   ra
    nop
END(CPU_GetCOUNT)
```

接着看，中间那个 do-while 循环目的是等待一秒，退出条件有两个，要么 timeout 为 0，要么读取的 RTC 到了下一秒。这里有个有趣的问题是：timeout 变成 0 要多久。timeout 的初始值为一千万，timeout 要变成 0，这个循环就要执行一千万次，那么这个循环体每次执行要多少时间呢？可以确定的是这个循环体指令不多，但有两处对 io 接口的访问，这个是比较花时间的。作者的原意是比较明显的，就是正常条件下，一千万次循环时间要远远大于一秒。假如一个循环的时间是 80 个时钟周期，一千万次就是 80x10^7 个时钟周期，以 2F 主频 800MHz 计，就是 1s 钟。可见作者是认为一个循环的时间要远远大于 80 个时钟周期。当然以上的是比较理想的情况，实际上，TLB miss，Cache miss 都加大了这个时间的不确定性，所以外边要套个 for 循环，确保这段代码指令，和变量都在 cache 中了。这是作者开头注释说的第一个原因，他还说了第二个原因，就是 while (timeout != 0 && cur == sec)这个条件为假的时候，实际上就是 cur 开始不等于 sec 的时候，就是 RTC 上刚好更新到下一秒的时候。在第一次进入那个 do-while 循环的时候，那个秒的含义是不确定的，可能是一秒的刚开始，也可能是一秒的中间，也有可能是一秒的末尾了，这样到下一秒到来的时候，这个时间间隔在 0~1s 之间，不能确定是否足够接近一妙，而循环两次就可基本解决这个问题。虽然不可能绝对精确，但是这个时间间隔是极为接近 1s 的。这是作者注释的同步（synched ）的意思吧，我认为。这里好有个问题，就是 count 寄存器是会溢出的，而且前面算了，每 10 秒多一点就溢出一次，看看这个函数的定义：可见这

个返回值是无符号的。

```
u_int     CPU_GetCOUNT __P((void));
```

但是_probe_frequencies()中的 cnt 却是 int 型的。用无符号的话可以处理溢出的问题，但这用 int 也可以用，不解。

接着就按这一秒钟内的 count 寄存器变化的值要算出 cpu 的主频(pipefreq),

```
if (timeout != 0) {
        clk_invalid = 0;
        md_pipefreq = cnt / 10000;
        md_pipefreq *= 20000;
        /* we have no simple way to read multiplier value
         */
        md_cpufreq = 66000000;
}
```

这里的 cpufreq 可能是外部总线的频率吧（66M）。前面算 pipefreq 直接乘以 2 不行吗，为什么这样？不管效率，反正 cpu 的频率算出来了。

至此这个函数调用结束，返回 initmips 函数。

dbginit

这个函数的调用层次很深，这个函数返回后，几乎大部分工作都完成了。这个函数在.../pmon/common/main.c 中，好，下面进入 dbginit 函数。

首先调用__init()，这个__init 就是对__ctors 这个函数的调用的包装，__ctors 函数的的定义如下：

```
static void
__ctors()
{
        void (**p)(void) = __CTOR_LIST__ + 1;

        while (*p)
                (**p++)();
}
```

看得出来，这个函数就是挨个执行初始化列表上的函数。.__CTOR_LIST__是在/Targets/Bonito/conf/ld.script 这个文件中定义的，其中第一个字表示这个构造列表的大小，这个列表的内容相当于 c++中的构造函数。

```
        .ctors :
        {
                __CTOR_LIST__ = .;
                LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
                *(.ctors)
                LONG(0)
```

```
        __CTOR_END__ = .;
    }
    .dtors :
    {
        __DTOR_LIST__ = .;
        LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
        *(.dtors)
        LONG(0)
        __DTOR_END__ = .;
    }
```

ctors 的内容是 c 代码中所有包含 constructor 关键字属性的函数，constructor 是 gcc 中对 c 的扩展，我认为只要函数声明中有__attribute__ ((constructor))，这个函数的地址就会被放在构造列表中，就会在 main 函数开始前被调用。由于这个东西我也不了解，暂时就先按自己的想法理解吧。

标有__attribute__ ((constructor))属性的函数有多少呢？readelf -S pmon 一把。

```
There are 10 section headers, starting at offset 0xd3940:
Section Headers:
  [Nr] Name      Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]           NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text     PROGBITS        80010000 000400 0b4ea8 00  WAX  0   0 512
  [ 2] .data     PROGBITS        800c4ea8 0b52a8 017800 00   WA  0   0  8
  [ 3] data      PROGBITS        800dc800 0ccc00 006c00 00   WA  0   0 1024
  [ 4] .ctors    PROGBITS        800e3400 0d3800 0000f4 00   WA  0   0  8
  [ 5] .dtors    PROGBITS        800e34f4 0d38f4 000008 00   WA  0   0  1
```

看到了吧，这个 ctors 节区大小为 f4,也就是 61 个字，除了首位两个（首是个数，尾是 0 标记），真正的最多是 59 个函数。实际上通过打印一共有 53 个有效的函数。其余是对齐用了???

这 53 个函数从性质上可分为 3 类：命令初始化，文件系统初始化，可执行文件类型初始化。

通过 cscope 的显示明显可以看出，命令初始化的数量最多。这些函数都在../pmon/cmds/的文件中定义，几乎是一个命令一个文件。每个文件的结构基本一致，找两个最熟悉的命令为例分析吧，load，g。

load 命令初始化在 cmds 目录下的的 load.c 文件中。

这个文件的基本是由这么几个部分(基于空间的原因，略作修改)：

```
const Optdesc     cmd_nload_opts[] = {....}     // help 时的输出内容
static int    nload (int argc, char **argv)      {}     //命令的具体执行
Int cmd_nload (int argc, char **argv){}          //
static const Cmd Cmds[] ={}                       //这个命令所在组的信息
static void init_cmd() {}                         //命令的注册
```

g 命令在 cmds 目录下的 cmd_go.c 文件中。

这个文件的由以下几个部分组成：

```
const Optdesc              cmd_g_opts[] = {} //help 时的输出
Int cmd_go (int ac,char ** av){}              //命令的具体执行
static const Cmd DebugCmd[] ={}               //命令所在组的信息
static void init_cmd()={}                     //命令的注册
```

下面就具体分析 load 这个命令初始化的全过程。

首先执行 init_cmd 这个函数，

```
  static void init_cmd __P((void)) __attribute__ ((constructor));
  static void init_cmd()
  {
      cmdlist_expand(Cmds, 1);
  }
```

这个__P 不知是什么原因，没有实际作用。

Cmds 是个 Cmd 结构的数组，Cmd 数组的定义如下：

```
  typedef struct Cmd {
        const char          *name;
        const char          *opts;
        const Optdesc *optdesc;
        const char          *desc;
        int             (*func) __P((int, char *[]));
        int                 minac;
        int                 maxac;
        int                 flag;
  #define CMD_REPEAT 1 /* Command is repeatable */
  #define CMD_HIDE           2 /* Command is hidden */
  #define CMD_ALIAS          4 /* Alias for another command name */
  } Cmd;
```

Load.c 中这个 Cmds 定义如下：

```
   static const Cmd Cmds[] =
   {
        {"Boot and Load"},
        {"load", "[-beastifr][-o offs]",
        cmd_nload_opts,
        "load file",
```

```
        cmd_nload, 1, 16, 0},
        {0, 0}
    };
```

上面的数组是这么理解的。第一项表示这个命令所属的组是”Boot and Load “，第二项是主角，对照 Cmd 的定义，分别表示这个命令是 load，可选的参数有哪些，可选参数的解释，对命令的解释，命令执行的函数，最少参数个数，最多参数个数，这个命令的类型（flag 定义解释得比较清楚）。

现在来看看这个 cmdlist_expand(Cmds,1)的作用。一看比较简单吧，就是把 Cmdlist 的当前未使用的元素地址指向传入的 Cmds 这个数组的地址就行了。由于 Cmdlist 数组的大小为 100，最多只能添加 100 次（当然一个 Cmds 中可以包含多个命令）。 g 这个命令的注册也类似，就是填充 Cmdlist 这个数组。

再看文件系统的初始化。

pmon 基本集成了最常用的文件系统，包括 ext2,fat，iso9660 等。这些文件系统初始化都在../pmon/fs/的下的文件中定义。 下面以我们最熟悉的 ext2 为例介绍文件系统的初始化。

这个 ext2 文件系统实现了 ext2 文件系统功能的一个子集，包含了十几个函数。实现了 ext2 文件系统的基本读写功能。

```
    static DiskFileSystem diskfile = {
        "ext2",
        ext2_open,
        ext2_read,
        ext2_write,
        ext2_lseek,
        is_ext2fs,
        ext2_close,
        NULL,
        ext2_open_dir,
    };
    static void init_diskfs(void) __attribute__ ((constructor));
    static void init_diskfs()
    {
          diskfs_init(&diskfile);
    }
```

diskfs_init($diskfile)函数完成文件系统的注册，传入的参数 diskfile 如前面定义。diskfs_init()函数在 ../pmon/fs/diskfs.c 文件中定义。

```
    int diskfs_init(DiskFileSystem *dfs)
    {
          SLIST_INSERT_HEAD(&DiskFileSystems,dfs,i_next);
          return (0);
    }
```

这个就是调用 SLIST_INSERT_HEAD，其中 DiskFileSystems 这个参数是链表头，这个宏定义如下：

```
#define SLIST_INSERT_HEAD(head, elm, field) do {                    \
        (elm)->field.sle_next = (head)->slh_first;          \
        (head)->slh_first = (elm);                          \
} while (0)
```

就是些链表的常见操作，不具体叙述。

还有一类的初始化是针对文件类型的，分别有 bin,elf,txt 等。下面以最常见的 bin 格式为例介绍初始化,这些文件格式的初始化代码在../pmon/loaders/目录下。

bin 格式的初始化在文件 exec_bin.c 中，这个文件大致有这么三个部分：

```
static long load_bin (int fd, ch ar *buf, int *n, int flags){} //如何 load bin 文件
static ExecType bin_exec = {}                    // bin 文件格式结构体的定义
static void init_exec() {}                       // 注册这种文件格式
```

具体的 bin 文件格式注册如下，

```
static void init_exec()
{
        exec_init(&bin_exec);
}
```

其中 bin_exec 定义为：

```
static ExecType bin_exec =
{
    "bin",                      // 文件格式的类型名
    load_bin,                   // bin 格式文件的 load 执行过程
    EXECFLAGS_NOAUTO, // flag,不自动启动
};
```

以上是三类主要的构造函数，除此之外还有两个函数，分别是

```
void init_controller_list(void) __attribute__((constructor));
static void init_kbd_driver(void) __attribute__((constructor));
```

都在../sys/dev/usb/的目录下，先看第一个。在文件 usb.c 中

```
void init_controller_list(void)
{
    TAILQ_INIT(&host_controller);
}
```

就是初始化 host_controller 而已。

Init_kdb_driver()在 usb_kdb.c 中，定义如下：

```
static void init_kbd_driver(void)
{
    usb_driver_register(&usb_kbd_driver);
}
```

这个 usb_kbd_driver 是个结构体,只有两个成员，分别是 probe 链表信息。

```
struct usb_driver usb_kbd_driver = {
    .probe = usb_kbd_probe,
};
```

至此所有构造函数执行完毕。__init 函数返回，回到了../pmon/common/main.c 的 dbginit 函数中。

envinit 的作用是环境变量的初始化。涉及的东西比较独立，我们在后面专门讲述，这里跳过。
环境变量设置完毕后回到 dgbinit()。接着执行

```
s = getenv("autopower");
if (s == 0 || strcmp(s, "1") != 0) {
        suppress_auto_start();
}
```

如果 autopower 定义了这个变量，返回这个变量的值，否则返回 0。执行 suppress_auto_start()，suppress 有压制 ，制止的意思。可见就是不许自动启动了，这里的自动启动指的是上电后就启动启动进入系统。

```
void suppress_auto_start(void)
{
        /* suppress auto start when power pluged in */
        CMOS_WRITE(0x80, 0xd);
}
static inline void CMOS_WRITE(unsigned char val, unsigned char addr)
{
                linux_outb_p(addr, 0x70);
                linux_outb_p(val, 0x71);
}
```

这个 suppress_auto_start()有点意思，居然往实时钟写个什么东西。

cs5536_devinit

好了，接着执行 tgt_devinit();
这个函数首先执行 cs5536_init()，看名字就知道是对南桥的初始化。
这个函数依次调用以下四个函数。

```
cs5536_vsm_init();
cs5536_i8259_init();
cs5536_ide_init();
cs5536_ohci_init();
```

先看 cs5536_vsm_init()，代码中的那 3 个 PCI_SPECIAL_*都是没有定义的，实际 cs5536_vsm_init()要执行的代码如下：

```
_rdmsr(SB_MSR_REG(SB_ERROR), &hi, &lo);
lo |= SB_MAR_ERR_EN | SB_TAR_ERR_EN | SB_TAS_ERR_EN;
                                        //具体见 AMD 南桥手册 226 页
_wrmsr(SB_MSR_REG(SB_ERROR), hi, lo);
```

```
/* setting the special cycle way */
_rdmsr(DIVIL_MSR_REG(DIVIL_LEG_IO), &hi, &lo);//见 AMD 手册 363 页
lo |= (1 << 28);              //选择关机的信号
_wrmsr(DIVIL_MSR_REG(DIVIL_LEG_IO), hi, lo);

_rdmsr(DIVIL_MSR_REG(DIVIL_BALL_OPTS), &hi, &lo);
lo |= 0x01;        //表示 IDE 引脚接 IDE 控制器，具体见 AMD 手册 365 页
_wrmsr(DIVIL_MSR_REG(DIVIL_BALL_OPTS), hi, lo);
```

这里的 vsm(Virtual Support Module)

这是 pmon 执行过程中第二次正式接触南桥，第一次是在 start.S 中。上面的代码就是三次多寄存器的读写。功能分别是重置错误位，确定关机的信号，表示 ide 引脚都连接 ide 控制器。

接下来是 cs5536_i8259_init();可以看出来是中断处理芯片的初始化。跳过不叙。

接下来是 cs5536_ide_init()，实际执行代码如下：

```
_rdmsr(DIVIL_MSR_REG(DIVIL_BALL_OPTS), &hi, &lo);
lo |= 0x01;//IDE 引脚接 IDE 控制器，具体见 AMD 手册 365 页
_wrmsr(DIVIL_MSR_REG(DIVIL_BALL_OPTS), hi, lo);
```

作者健忘，上面刚做了一次。

```
_rdmsr(SB_MSR_REG(SB_CTRL), &hi, &lo);
lo &= ~( (1<<13) | (1 << 14) );
_wrmsr(SB_MSR_REG(SB_CTRL), hi, lo);
```

这个代码指定主从两个 IDE 控制器的地址，具体见 AMD 手册 229 页。

```
_rdmsr(IDE_MSR_REG(IDE_CFG), &hi, &lo);
lo |= (1 << 1);
_wrmsr(IDE_MSR_REG(IDE_CFG), hi, lo);
```

使能 IDE 控制器，具体见 AMD 手册 P336

```
_rdmsr(IDE_MSR_REG(IDE_DTC), &hi, &lo);
lo &= 0x0;
lo |= 0x98980000;     //PIO-0 mode for 300ns IO-R/W
_wrmsr(IDE_MSR_REG(IDE_DTC), hi, lo);
```

不同模式下（多种 pio 和 dma 模式）的有效脉冲宽度和最小恢复时间的设置，具体见 AMD 手册 P337

```
_rdmsr(IDE_MSR_REG(IDE_CAST), &hi, &lo);
lo &= 0x0;
lo |= 0x990000a0;     //PIO-0 mode
_wrmsr(IDE_MSR_REG(IDE_CAST), hi, lo);
```

特定模式下命令的有效脉冲宽度和最小恢复时间设置，具体见 AMD 手册 P338

```
hi = 0x60000000;//前 3 位是 011，表示针对 IDE 操作
lo = 0x1f0fff8;     //fff8 和 001F0and 后是 1F0，是主 IDE 的 I/O 起始地址
_wrmsr(GLIU_MSR_REG(GLIU_IOD_BM0), hi, lo);
```

具体见 AMD 手册 P219

```
    hi = 0x60000000;
    lo = 0x3f6ffffe;      //主 IDE 还有一个 I/O 地址是 3F6
    _wrmsr(GLIU_MSR_REG(GLIU_IOD_BM1), hi, lo);
```

GLIU_IOD_BM[0,1]具体位定义相同，见 AMD 手册 P219

接下是 ohci 的初始化，代码很短，如下：

```
    hi = 0x40000006;//前 3 位是 010，表示是 USB 设备
    lo = 0x050fffff;//基地址是 0x6050
    _wrmsr(GLIU_MSR_REG(GLIU_P2D_BM1), hi, lo);
```

具体见 AMD 手册 P203

```
    _rdmsr(USB_MSR_REG(USB_OHCI), &hi, &lo);
    hi = (1 << 1) | (0 << 2); // BUS MASTER ENABLE AND MEM ENABLE
    lo = 0x06050000;//基地址是 60500，mem 大小是 256 字节
    _wrmsr(USB_MSR_REG(USB_OHCI), hi, lo);
```

具体见 amd 手册 P266

ohci 也是 usb 主控制器的三大标准之一。关于 ohci 的内容，请参看相关规范。

至此，cs5536_init()执行完毕，函数返回 tgt_devinit()。

跳过与龙芯无关的两个变量的查找。

接下来是 CPU_ConfigCache();

这个函数配置程序中要使用的 cache 相关参数，代码在../pmon/arch/mips/cache.S 中。这个代码的许多工作其实在 start.S 中已经做了，代码看似很长，真正执行的却不多。一开始无非是数据/指令的一级缓存大小和 cache 行大小之类的探测和部分赋值，start.S 中这个工作都做了，只是没有保存这些值而已。

直接转到实际部分，在读 prid 知道是龙芯 2 号之后，跳到 ConfGodson2，可以看到 ConfGodson2 开始的一些代码都被注释了，都是和三级缓存相关的代码，实际上 ConfGodson2 开始的那两句也是没有用的。到了 ConfGodsonL2。设置二级缓存大小后，就到了 ConfResult，就是些变量的 cache 相关设置了。

剩下来 tgt_devinit()还剩下要执行两个函数：_pci_businit(1)，cs5536_pci_fixup()。

_pci_businit

接下来执行_pci_businit(1)。从函数名上看功能是 pci 总线的初始化，实际包括 pci 设备的探询并初始化为一个设备链表，分配相应的资源（io 和 mem 请求）。传入的参数 1 表示要初始化。_pci_businit 这个函数按照枚举的方式扫描总线，并初始化相应的结构，代码在../sys/dev/pci/pciconfi.c 中。

跳过打印相关的语句，下面执行：

```
    init = _pci_hwinit (init, &def_bus_iot, &def_bus_memt);
    pci_roots = init;//正常的返回值为 1，pci_roots 表示 root 的个数
    if (init < 1)
```

```
        return;
```
重点是_pci_hwinit (init, &def_bus_iot, &def_bus_memt)，这个 init 就是传入的参数 1，表示开始初始化，其余两个参数一个是 io 的空间描述，一个是 mem 的空间描述， 都是 tgt_bus_space 的结构体指针，不过在函数中都未使用。

```
struct tgt_bus_space {
        u_int32_t       bus_base;
        u_int32_t       bus_reverse;        /* Reverse bus ops (dummy) */
};
```
bus_reverse 不知道是什么。下面进入_pci_hwinit()。这个的执行代码较长，但基本都是赋值而已。执行语句如下：

```
pci_local_mem_pci_base = PCI_LOCAL_MEM_PCI_BASE;
```
这个值是 0x8000 0000,就是说 pci 的 mem 地址分配从这个地址开始，前面都是用户空间。

```
pd = pmalloc(sizeof(struct pci_device));
pb = pmalloc(sizeof(struct pci_bus));
if(pd == NULL || pb == NULL) {
        printf("pci: can't alloc memory. pci not initialized\n");
        return(-1);
}
```
分配 pci_device 和 pci_bus 这两个结构体。可见分别描述了设备和总线内容。实际上用来描述北桥(或者说是一个 pci 设备的总抽象)和唯一的 pci 总线。

```
pd->pa.pa_flags =    PCI_FLAGS_IO_ENABLED |
                                        PCI_FLAGS_MEM_ENABLED;
pd->pa.pa_iot = pmalloc(sizeof(bus_space_tag_t));//这里不检查返回值?
pd->pa.pa_iot->bus_reverse = 1;
pd->pa.pa_iot->bus_base = BONITO_PCIIO_BASE_VA;
```
这个值为 0xbfd0 0000，所有 io 端口都在 bfd00000 开始的 1MB 空间内。

```
pd->pa.pa_memt = pmalloc(sizeof(bus_space_tag_t));
pd->pa.pa_memt->bus_reverse = 1;
pd->pa.pa_memt->bus_base = 0xb0000000;
pd->pa.pa_dmat = &bus_dmamap_tag;
```
以上代码都是对 pci_device 中的 pa 的成员赋值，pa 是一个 pci_attach_args 结构，描述 pci 设备的一些附加属性。

```
pd->bridge.secbus = pb;
```
表示这个设备所接的桥的下一级总线。pd->bridge 之前没赋值,pmalloc 返回的内容是清 0 的

```
_pci_head = pd;
```
这个_pci_head 全局变量很重要，以后会多次用到。

接下来的代码根据是否按照新的 pci 窗口布局执行不同的代码，旧的布局是 3 个 64MB 的 pci 窗口，新的是 128，32，32。pci 窗口的目的就是把 cpu 送出的地址转化成总线地址。pmon 默认代码中没有定义 NEW_PCI_WINDOW。小技巧：看条件编译的条件是否定义有个简单的方法，加条件编译的部分下一条要报错的代

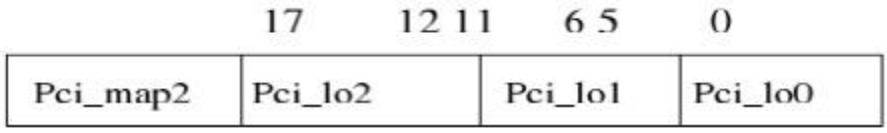
码，如果报了，说明这段代码要被编译，条件为真。
以上是对北桥的概括描述，现在要描述 PCI 总线了。执行的代码如下：

```
/* reserve first window for vga mem */
pb->minpcimemaddr    = PCI_MEM_SPACE_PCI_BASE+0x04000000;
pb->nextpcimemaddr = PCI_MEM_SPACE_PCI_BASE+
                                        BONITO_PCILO_SIZE;
```

注释说前 64M(0x04000000)的 pci 窗口保留给 vga。BONITO_PCILO_SIZE 为 192MB， PCI_MEM_SPACE_PCI_BASE 为 0 ???

```
BONITO_PCIMAP =
    BONITO_PCIMAP_WIN(0, 0x00000000) |
    BONITO_PCIMAP_WIN(1, PCI_MEM_SPACE_PCI_BASE+0x04000000) |
    BONITO_PCIMAP_WIN(2, PCI_MEM_SPACE_PCI_BASE+0x08000000) |
    BONITO_PCIMAP_PCIMAP_2;
```

这个 PCIMAP 寄存器设置了 CPU-->PCI 的地址转换。以下引自《pci 设备初始化.pdf》：为了让 CPU 访问 PCI 空间，需要将 CPU 空间映射到 PCI 空间。在内存空间 256M 上有三个连续的大小均为 64M 的区间，分别称为 PCI_Lo0, PCI_Lo1,PCI_Lo2。这三个区间可以被北桥映射到 PCI 以 64M 对齐的任意位置。映射的关系通过设置 Bonito 的 PCIMAP 寄存器。该寄存器的格式如下图。



PCIMAP 寄存器

pci_lo0,pci_lo1,pci_lo2 分别是上面所说三个区间的高 6 位地址(bit31- 26)，而 Pci_map2 是说明映射到 2G 以上的空间还是 2G 以下的空间。因此上面给 BONITO_PCIMAP 赋值就将 PCI_lo0,PCI_lo1,PCI_lo2 分别映射到了 PCI 空间的从 0,64M,128M 开始的地址。
以上的说法中对于 pci_map2 的描述部分在 2F 中是不适用的，2F 的这个寄存器的高 14 位保留。低 18 位定义相同。

```
pb->minpciioaddr    = PCI_IO_SPACE_BASE+0x000b000;
pb->nextpciioaddr = PCI_IO_SPACE_BASE+ BONITO_PCIIO_SIZE;
传统的 16 位地址的 io 空间，大小是 64KB
pb->pci_mem_base      = BONITO_PCILO_BASE_VA;//0xb000 0000
pb->pci_io_base          = BONITO_PCIIO_BASE_VA;//0xbfd0 0000
```

这两个 base 很重要，前面的 minpciioaddr 和 nextpciioaddr 都要和 base 联系。

```
pb->max_lat = 255;
pb->fast_b2b = 1;
pb->prefetch = 1;
```

```
        pb->bandwidth = 0x4000000;
        pb->ndev = 1;
        _pci_bushead = pb;
```
这个_pci_bushead 全局变量很重要，以后会多次用到。在我们的龙芯 pc 中，尚未采用多 pci 架构，所以 pcibus 都是一个。
```
        _pci_bus[_max_pci_bus++] = pd;

        bus_dmamap_tag._dmamap_offs = 0;
```
下面的 pcibase 是指北桥 PCI 配置头中的 base
```
        BONITO_PCIBASE0 = 0x80000000;
        BONITO_PCIBASE1 = 0x0;
        BONITO_PCIBASE2 = 0x70000000;
        BONITO_PCI_REG(0x40) = 0x80000000;//base 0  的 mask
        BONITO_PCI_REG(0x44) =    0xf0000000;//base 1  的 mask

        /* pci base0/1 can access 256M sdram */
        BONITO_PCIMEMBASECFG = 0;
```

_pci_hwinit 执行完以后返回_pci_businit()，执行如下代码：
```
      struct pci_device *pb;//这个变量命名不合适，应为 pd
        for(i = 0, pb = _pci_head; i < pci_roots; i++, pb = pb->next) {
```
这里的_pci_head 就是刚才在_pci_hwinit 中赋值的。pci_roots 是 1。也就是这个 for 循环的循环体实际上只执行一次。
```
                _pci_scan_dev(pb, i, 0, init);
        }
```
_pci_scan_dev(pb, i, 0, init)中 i 表示总线号(其实就是 0)，0 表示 device 号从 0 开始查找。
_pci_scan_dev()的内容如下：
```
        for(; device < 32; device++) {
                _pci_query_dev (dev, bus, device, initialise);
        }
```
这个代码功能比较清楚，就是以枚举的方式探测相应的设备是否存在，传入参数分别是北桥的设备指针，北桥所在的总线号 0，要探测的设备号 device(0～31)。
_pci_query_dev()代码如下：。
```
static void
_pci_query_dev (struct pci_device *dev, int bus, int device, int initialise)
{
        pcitag_t tag;
        pcireg_t id;
        pcireg_t misc;

        tag = _pci_make_tag(bus, device, 0);
```

在pci设备空间访问的时候，所使用的地址由四部分组成。总线号，设备号，功能号，寄存器号。_pci_make_tag的工作就是把这总线号，设备号，功能号三个数合成一个总线配置空间地址的标识，其中的参数0表示功能号。对于多功能设备，不同功能的总线号和设备号是相同的，且0号功能必定存在，就是说多功能的设备功能号是从0开始的。所以探测0号功能是非常合理的。

```
if (!_pci_canscan (tag))
    return;
```

bus=0且device=0的话_pc_canscan()会返回0,表示是北桥，是特殊的。

```
id = _pci_conf_read(tag, PCI_ID_REG);
```

读PCI配置空间中的厂商ID看看这个设备是否存在。

```
if (id == 0 || id == 0xffffffff) {//表示没有设备
    return;
}
misc = _pci_conf_read(tag, PCI_BHLC_REG);
```

检查是否是多功能设备

```
if (PCI_HDRTYPE_MULTIFN(misc)) {
```

如果是多功能设备，所有功能都加到设备(确切的说是功能列表中去)列表中去。

```
    int function;
    for (function = 0; function < 8; function++) {
        tag = _pci_make_tag(bus, device, function);
        id = _pci_conf_read(tag, PCI_ID_REG);
        if (id == 0 || id == 0xffffffff) {
            continue;
        }
        _pci_query_dev_func (dev, tag, initialise);
```

添加到设备(功能)列表中

```
    }
}
else {
    _pci_query_dev_func (dev, tag, initialise);
```

挂上设备列表并记下空间请求

```
}
```

_pci_query_dev函数就是这些了。我们看看_pci_query_dev_func的实现。

_pci_query_dev_func()的主要工作就是初始化这个探测到的pci设备（功能），加入到pci设备（功能）链表，记录各个设备（功能）的资源请求主要是io空间和mem空间的请求（满足请求不在这个函数中），并相应调整pci总线的特征。

一开始是申请一个pci_device的结构体并初始化它。值得注意的是，这个新设备的父亲是dev，也就是北桥。

```
_pci_device_insert(dev, pd);
```

_pci_device_insert(dev, pd)将pd插入dev->bridge->child中。bus上的所有设备都在这个child列表中，也就是说所有pci设备（功能）都是总线的儿子（不包括北桥）。接下来是进一步初始化设备和因这个设备调整总线的性质。

到了_pci_setupIntRouting(pd);
看名字是设置中断线有关，不懂，不予解释。

```
stat = _pci_conf_read(tag, PCI_COMMAND_STATUS_REG);//读状态寄存器，准备使能
stat &= ~(PCI_COMMAND_MASTER_ENABLE |
          PCI_COMMAND_IO_ENABLE |
          PCI_COMMAND_MEM_ENABLE);
_pci_conf_write(tag, PCI_COMMAND_STATUS_REG, stat);
pd->stat = stat;

/* do all devices support fast back-to-back */
if ((stat & PCI_STATUS_BACKTOBACK_SUPPORT) == 0) {
        pb->fast_b2b = 0;           /* no, sorry */
}
```

backtoback 就是背靠背，就是连续传送，有一个设备不支持，总线的就标志不支持

```
/* do all devices run at 66 MHz */
if ((stat & PCI_STATUS_66MHZ_SUPPORT) == 0) {
        pb->freq66 = 0;             /* no, sorry */
}
```

一个设备不支持 66MHz 就会导致整个总线不能使用 66M 的频率。

```
/* find slowest devsel */
x = stat & PCI_STATUS_DEVSEL_MASK;
if (x > pb->devsel) {
        pb->devsel = x;
}
```

这个见 pci2.3 规范 P200，x＝0 表示快速设备，1 表示中等速度设备，2 表示慢速设备。

```
/* Funny looking code which deals with any 32bit read only cfg... */
bparam = _pci_conf_read(tag, (PCI_MINGNT & ~0x3));
pd->min_gnt = 0xff & (bparam >> ((PCI_MINGNT & 3) * 8));
bparam = _pci_conf_read(tag, (PCI_MAXLAT & ~0x3));
pd->max_lat = 0xff & (bparam >> ((PCI_MAXLAT & 3) * 8));

if (pd->min_gnt != 0 || pd->max_lat != 0) {//为 0 表示没有要求
        /* find largest minimum grant time of all devices */
        if (pd->min_gnt != 0 && pd->min_gnt > pb->min_gnt) {
                pb->min_gnt = pd->min_gnt;
        }
        /* find smallest maximum latency time of all devices */
        if (pd->max_lat != 0 && pd->max_lat < pb->max_lat) {
                pb->max_lat = pd->max_lat;
```

```
        }

        /* subtract our min on-bus time per second from bus bandwidth */
        if (initialise) {
            pb->bandwidth -= pd->min_gnt * 4000000 /
                              (pd->min_gnt + pd->max_lat);
        }
    }
```

上面的和总线的性质有关，pb->bandwidth -= pd->min_gnt * 4000000 / (pd->min_gnt + pd->max_lat);应该是根据设备的要求分配带宽。

```
    bparam = _pci_conf_read(tag, PCI_INTERRUPT_REG);
    bparam &= ~(PCI_INTERRUPT_LINE_MASK <<
                                    PCI_INTERRUPT_LINE_SHIFT);
    bparam |= ((_pci_getIntRouting(pd) & 0xff) <<
                                      PCI_INTERRUPT_LINE_SHIFT);
    _pci_conf_write(tag, PCI_INTERRUPT_REG, bparam);
```

表示设备要求的 interrupe pin 连接到中>断控制器的哪个引脚上

接下来要判断这个设备（功能）的类型，分为三类：pci 桥，IDE 控制器，其它。在 2F 的盒子上，除了北桥外没有 pci 桥，但北桥不调用_pci_query_dev_fun，IDE 的控制器集成在南桥中了。也就是说，所有的设备（功能）都会从最后那条路走。

一开始是一个大循环，

```
    for (reg = PCI_MAPREG_START; reg < PCI_MAPREG_END; reg += 4) {
```

读可能存在的 6 个 base 寄存器，看来要分配 io 空间和 mem 空间了。

```
        old = _pci_conf_read(tag, reg);
        _pci_conf_write(tag, reg, 0xffffffff);
        mask = _pci_conf_read(tag, reg);
        _pci_conf_write(tag, reg, old);
```

上面的代码看起来有些怪，这是读这几个寄存器的特殊之处，具体的解释见 pci2.3 规范的 P208。

如果要申请空间，判断是 io 空间还是 mem 空间，读出的 mack 末位为 1 表示要 i/o 空间，否则是 mem 空间。

如果是 io 空间的申请，则：

```
    mask |= 0xffff0000;
```

io 空间的大小是受限的，mask1 的位数表明申请空间的大小。

```
    pm = pmalloc(sizeof(struct pci_win));
    f(pm == NULL) {
        。。。。。。。
    }
```

现在开始记下空间需求

```
    pm->device = pd;
    pm->reg = reg;    //配置空间的基地址
```

```
pm->flags = PCI_MAPREG_TYPE_IO;
pm->size = -(PCI_MAPREG_IO_ADDR(mask));

_insertsort_window(&pd->parent->bridge.iospace, pm);
```

让这个总线记下这个设备的 io 空间需求。

下面是对于 mem 空间需求的处理：

```
switch (PCI_MAPREG_MEM_TYPE(mask)) {
    case PCI_MAPREG_MEM_TYPE_32BIT:
    case PCI_MAPREG_MEM_TYPE_32BIT_1M://这个是保留的，怎么
            break;
    case PCI_MAPREG_MEM_TYPE_64BIT:
            _pci_conf_write(tag, reg + 4, 0x0);
            skipnext = 1;//两个 32 位当一个，接下的那个 32 位也搞了
            break;
            default:
            _pci_tagprintf(tag, "reserved mapping type 0x%x\n",
                        PCI_MAPREG_MEM_TYPE(mask));
            continue;
    }
```

可见对于 mem 空间的需求和 io 空间有很大的不同。mem 空间是可以支持 64 位地址的，而 io 空间不行，64 位的时候就要占用两个 base-address 寄存器。skipnext 的作用就是标记是不是两个 32 位地址合成 64 位。

```
    if  (!PCI_MAPREG_MEM_PREFETCHABLE(mask)) {
        pb->prefetch = 0;
    }
```

预取。接下来的代码和 io 空间的申请相同。下面来看看_insertsort_window 到底干了什么。

```
pm1 = (struct pci_win *)pm_list;
    while((pm2 = pm1->next)) {
        if(pm->size >= pm2->size) {//空间大的被访问几率高？不见得
            break;
        }
        pm1 = pm2;
    }
    pm1->next = pm;
    pm->next = pm2;
```

就是看按照申请空间的大小把这个 pci_win 结构排序，从大到小排序。

我们已经记录了一个设备（功能）的 io 和 mem 请求，如果它还需要额外的扩展 rom，就再记录并提交申请。具体可见 pci2.3 规范 30h 寄存器的解释。

OK，至此，_pci_query_dev_func 函数执行完毕。也就是对一个检测到的设备（功能）结构体的初始化，插入 pci 设备链表并初步处理了这个设备（功能）提出的需求。

在所有的设备（功能）都执行_pci_query_dev 以后，_pci_businit()要执行的只有_setup_pcibuses(init)了。这个函数功能是分配带宽并处理空间申请。

一开始分配带宽。代码中有一句 def_ltim = pb->bandwidth / pb->ndev，表示平均主义的话每个设备多少带宽，但是我没有找到这个 ndev 初始化后在哪儿被改变了。然后处理空间的请求。

```
for(i = 0, pd = _pci_head; i < pci_roots; i++, pd = pd->next) {
    _pci_setup_windows (pd);
}
```

处理 mem/io 空间请求，并反馈回请求的设备。io 和 mem 空间的请求已经以 pci_win 结构体链表的形式记录下来了，mem 的请求记录在 bridge.memspace 中，io 请求在 bridge.iospace 中。先处理的是 mem 的请求。主要就是

```
pm->address = _pci_allocate_mem (dev, pm->size);
```

实现很显然，就是按顺序切割剩余的空间。

```
address = (dev->bridge.secbus->minpcimemaddr + size - 1)& ~(size - 1);
address1 = address + size;
if (address1 > dev->bridge.secbus->nextpcimemaddr ||
                address1 < dev->bridge.secbus->minpcimemaddr) {
        return(-1);
}
dev->bridge.secbus->minpcimemaddr = address1;
return(address);
```

这个代码的功能应该是很清楚了。pm->address 在得到分配到的空间的基地址后，还要写回那个基地址寄存器。下面的代码就是这个目的。

```
_pci_conf_write(pd->pa.pa_tag, pm->reg, memory);
```

对于 mem 空间的申请就是这样了。 剩下的扩展空间和 io 空间过程相似。

cs5536_pci_fixup

回到 tgt_devinit()，只剩下一个 cs5536_pci_fixup()了。

这个函数实际上执行两个函数 cs5536_pci_otg_fixup() 和 cs5536_lpc_fixup();

先看前一个。代码如下：

```
tag = _pci_make_tag(0, 14, 7);//是 5536 上的 USB controller

/* get the pci memory base address. */
base = _pci_conf_read(tag, 0x10);
base |= 0xb0000000;     //不清楚为什么
```

前头 setup_pcibuses 函数已经完成了空间需求，这里的地址是实际分配到的。

```
cmdsts = _pci_conf_read(tag, 0x04);
_pci_conf_write(tag, 0x04, cmdsts | 0x02);
```

使能 mem space access,接下来的几条指令就要执行写操作

```
        /* set the MUX pin as HOST CONTROLLER */
        val = *(volatile u32 *)(base + 0x04);
        val &= ~(3 << 0);
        val |= (2 << 0);
        *(volatile u32 *)(base + 0x04) = val;

        val = *(volatile u32 *)(base + 0x0c);
        if( (val & 0x03) == 0x02 ){
            printf("cs5536 otg configured for USB HOST CONTROLLER.\n");
        }else if( (val & 0x03) == 0x03 ){
            printf("cs5536 otg configured for USB DEVICE.\n");
        }else{
            printf("cs5536 otg configured for NONE.\n");
        }
    _pci_conf_write(tag, 0x04, cmdsts);
```

这个具体的作用要看南桥关于 usb controller 部分的描述。

接下是 cs5536_lpc_fixup 函数，代码如下：

```
        /* disable the primary irq of IRQ1 and IRQ12 */
        _rdmsr(DIVIL_MSR_REG(PIC_IRQM_PRIM), &hi, &lo);//见 amd 手册 382
        lo &= ~((1 << 1) | (1 << 12));
        _wrmsr(DIVIL_MSR_REG(PIC_IRQM_PRIM), hi, lo);

        /* enable the lpc irq of IRQ1 and IRQ12 to input */
        _rdmsr(DIVIL_MSR_REG(PIC_IRQM_LPC), &hi, &lo);
        lo |= ((1 << 1) | (1 << 12));
        _wrmsr(DIVIL_MSR_REG(PIC_IRQM_LPC), hi, lo);
```

使能键盘和二级 IDE 中断,见 amd 手册 P384。

```
        /* enable the lpc block    */
        _rdmsr(DIVIL_MSR_REG(LPC_SIRQ), &hi, &lo);
        lo |= (1 << 7);
        _wrmsr(DIVIL_MSR_REG(LPC_SIRQ), hi, lo);
```

enable serial interrupt，见 amd 手册 P466。

好了，至此，tgt_devinit()就全部执行完毕了。返回 dbginit()。

Init_net

接下来要执行 init_net (1)，这个函数极度混乱，绝不是初始化网络这么简单。我在第一次看 pmon 代码的时候，这个函数断断续续看了有半个月。

一开始是和 loglevel 相关的，打印相关的，跳过。

接着执行 paraminit();从调用前的注释看是初始化系统相关的全局参数。代码不长，就以下 7 行：

```
hz = HZ;       //HZ 表示 1 秒钟时钟中断的产生次数
tick = 1000000 / HZ;//两次时钟中断中间的 us 数

tz.tz_minuteswest = TIMEZONE;//看来在 pmon 中，这些东西不讲究
tz.tz_dsttime = DST;
#define DST_NONE    0    /* not on dst */
#define DST_USA     1    /* USA style dst */
#define DST_AUST    2    /* Australian style dst */
#define DST_WET     3    /* Western European dst */
#define DST_MET     4    /* Middle European dst */
#define DST_EET     5    /* Eastern European dst */
#define DST_CAN     6    /* Canada */
```

这里的 TIMEZONE 和 DST 在文件头都定义为 0,就是 0 时区，DST_NONE 了。DST 的解释如下（摘自网络）：美国加拿大实行 DST 的时间是 3 月的第二个星期天早晨两点开始到 11 月的第一个星期日的早晨两点。三月第二个个星期日早晨两点所有时钟向前回拨一个小时，到 11 月 DST 截止再拨回来。

```
ncallout = 16 + NPROC; //进程数，pmon 中有无进程概念的必要？
nmbclusters = NMBCLUSTERS;
maxfiles = NDFILE * NPROC;
```

这三个全局变量作用有限，用到的时候再分析，留个印象先。

从 paraminit 返回。执行 vminit();

谈到内存分配，前面我们在多个地方见到了动态分配函数 pmalloc，下面先讲讲这个函数是怎么个流程。pmalloc 代码如下：

```
void * pmalloc(size_t nbytes)
{
    void *p;
    p = malloc(nbytes);
    if(p) {
        bzero(p, nbytes);
    }
    return(p);
}
```

可见这函数有分配和清零两部分。我们只关心的 malloc 函数代码如下蓝色标记。

```
void * malloc(size_t nbytes)
{
    HEADER *p, *q;    /* K&R called q, prevp */
    unsigned nunits;
    nunits = (nbytes + sizeof(HEADER) - 1) / sizeof(HEADER) + 1;
```

这里的 HEADER 就是 header，定义如下：

```
union header {
```

```
        struct {
        union header    *ptr;
        unsigned            size;
        } s;
        ALIGN               x;
    };
```

本质上就是一个指向本联合体的指针，和一块空间的大小。从上一句代码中可以看出，每申请一块空间，要同时要带一个结构头。

```
        if ((q = allocp) == NULL) { /* no free list yet */
            base.s.ptr = allocp = q = &base;
            base.s.size = 0;
        }
```

这里是这样的，allocp 是全局变量，如果没有明确的初始化，全局变量就默认为 0，就是这里的 NULL。上面对应的是第一次执行的时候的情形 ,allocp 的含义是当前可被使用的空间的描述信息。

base 也是一个没有初始化的全局变量。这里算是初始化了一下。

```
        for (p = q->s.ptr; ; q = p, p = p->s.ptr) {
            if (p->s.size >= nunits) {    /* big enough */
                if (p->s.size == nunits)       /* exactly */
                    q->s.ptr = p->s.ptr;
                else {          /* allocate tail end */
                    p->s.size -= nunits;
                    p += p->s.size;
                    p->s.size = nunits;
                }
                allocp = q;
                return ((char *)(p + 1));
            }
```

这个 if 对应的是当前拥有的空间就能满足申请的情形，

```
            if (p == allocp)
                if ((p = morecore (nunits)) == NULL) {
                    return (NULL);
                }
```

这个对应现有空间不足的情形，调用 morecore 函数再申请，分配成功后再在 for 循环内判断是否已经满足要求了。函数代码如下：

```
    mu = NALLOC * ((nu + NALLOC - 1) / NALLOC);
```

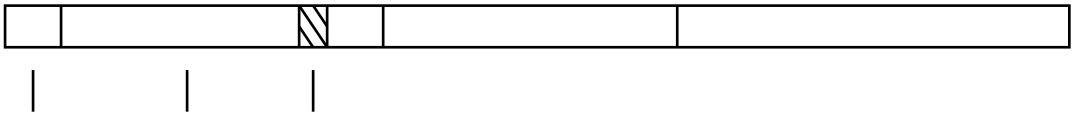
NALLOC 定义为 128，表示每次申请的大小至少为 128 个结构体的大小。

```
    cp = sbrk (mu * sizeof (HEADER));
```

从命名上就看出了要动真格的了。sbrk 代码如下(绿色)：

```
    if (allocp1 + n <= allocbuf + ALLOCSIZE) {//大小如上定义，是 32KB
        allocp1 += n;          //allocp1 表示当前可分配的空间首地址
        return (allocp1 - n);
```

```
    } else
        return (NULL);
```
allocbuf指向一个大数组，为 32KB 大小。sbrk 就是从开始位置切割而已。
```
    if ((int)cp == NULL)
        return (NULL);
    up = (HEADER *) cp;
    up->s.size = mu;
    free ((char *)(up + 1));
    return (allocp);
```
根据现状更新这块分配到的空间头部的记录块。
看完了代码，这个代码就不觉得怪了。在 pmalloc 能使用的 32KB 空间内，每次分配的空间其实包括 header，申请的空间，对齐用的空间三部分。且这块空间至少是 NALLOC 个 header 个结构体大小的空间。是否浪费？不。因为这块空间还会被分割的。具体的可模拟不同情形把代码走一遍。



header　所需空间　对齐用

pmalloc 就到这里了，回到流程上来，看 vminit()函数。就一个 if 语句，如下：
```
    if (!kmem) {                    //这个全局变量 kmem 没有初始化，就是 0
        if (memorysize < VM_KMEM_SIZE * 2) {
            panic ("not enough memory for network");
        }
```
这里的 memorysize 是个外部变量，就是 initmips 中的那个，最大为 256MB，盒子上面内存为 512MB，memorysize 变量就为 256*2^20。VM_KMEM_SIZE 大小为 512KB。
```
        memorysize = (memorysize - VM_KMEM_SIZE) & ~PGOFSET;
```
不足页的不可用,且空出一个 vm,memorysize 单位是字节
```
        if ((u_int32_t)&kmem < (u_int32_t)UNCACHED_MEMORY_ADDR) {
```
这个判断没有理由不成立，还要执行的就一条了,就不管 else 了。
```
            kmem = (u_char *) PHYS_TO_CACHED (memorysize);
```
事实上我认为 kmem 的值是确定的，memorysize 是 256MB－512KB，那么 kmem 就是 0x8ff8 0000(uncached 前是 0x0ff80000)。vminit()至此执行完毕。函数设定的 kmem 变量值指定了待会 kmem_malloc 分配时使用的空间。
接着执行 kmeminit()函数。实际就执行以下几句：
```
    int npg;
    npg = VM_KMEM_SIZE/ NBPG;
```
npg（number of page）计算一个 VM_KMEM_SIZE 的页数，页大小为 4KB。
```
    kmemusage = (struct kmemusage *) kmem_alloc(kernel_map,
                                (vsize_t)(npg * sizeof(struct kmemusage)));
```

我们又看到一个分配函数，kmem_alloc()，有两个传入参数。第一个参数kemel_map,这个变量实际上是没用的，不去理睬。第二个是要申请的空间大小，从这个大小就可猜出，每一个页都有一个控制结构kmemusage，下面看看这个结构体的定义：

```
struct kmemusage {
        short ku_indx;          /* bucket index */
        union {
                u_short freecnt;/* for small allocations, free pieces in page */
                u_short pagecnt;/* for large allocations, pages alloced */
        } ku_un;
};
```

ku_indx 料想就是一个下标，ku_un 从注释中看出是对一块区域的描述。

kmem_alloc(map,size)实际上就是调用 kmem_malloc(map,size,0)，下面看 kmem_malloc 做了点什么。代码如下（蓝色标记）：

```
size = (size + PGOFSET) & ~PGOFSET;
```

每次分配的是 4KB 对齐。

```
if (kmem_offs + size > VM_KMEM_SIZE) { //没有足够空间了，报错返回。
    。。。。。
}
p = (vm_offset_t) &kmem[kmem_offs];
kmem_offs += size;
return p;
```

这个分配的策略很简单，就是从 kmem 开始分配。kmem_offs 表示当前可供分配空间的起始地址。返回值赋值给 kmemusage。

回到 kmeminit()，只剩下一句了。

```
kmem_map = kmem_suballoc(kemel_map, (vaddr_t *)&kmembase,
                         (vaddr_t *)&kmemlimit, (vsize_t)(npg * NBPG), FALSE);
```

kmem_suballoc()代码如下：

```
vm_map_t kmem_suballoc (vm_map_t map, vm_offset_t *base,
                                vm_offset_t *lim, vm_size_t size, canwait)
{
        if (size > VM_KMEM_SIZE)
                panic ("kmem_suballoc");
        *base = (vm_offset_t) kmem;
        *lim = (vm_offset_t)    kmem + VM_KMEM_SIZE;
        return map;
}
```

经过这个调用以后，kmembase 就是 kmem(一般就是 0x8ff80000)，kmemlimit 表示可供分配空间的边界。kmem_alloc 和 kmem_suballoc 到底是什么关系呢？

不管了，至此，kmeminit()执行完毕。

回到 init_net()，接着执行：

```
callout = malloc(sizeof(struct callout) * ncallout, M_TEMP, M_NOWAIT);
```

```
        callfree = callout;
        for (i = 1; i < ncallout; i++) {
                callout[i-1].c_next = &callout[i];
        }
```

先分配 ncallout 个 ncallout 结构体，ncallout 在 paraminit 中已经赋值。为 19。

```
struct callout {
        struct   callout *c_next;              /* next callout in queue */
        void       *c_arg;                      /* function argument */
        void       (*c_func) __P((void *));      /* function to call */
        int c_time;                     /* ticks to the event */
};
```

感觉是一个函数调用的队列。具体的等用到时再说。

这里的 malloc 函数有 3 个参数，和前面的不同。malloc 执行代码如下（蓝色）：

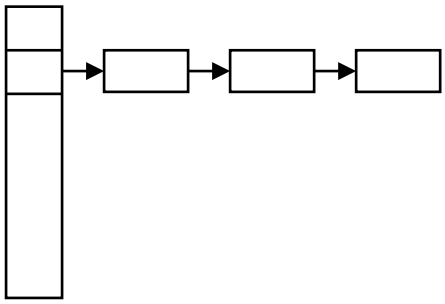
```
    void * malloc(unsigned long size, int type ,int flags)
    {
        。。。。。。

        indx =BUCKETINDX(size);
        kbp = &bucket[indx];
```

这个 malloc 分配的空间大小必须为 2 的多少次幂，返回的 indx 就是这个幂。对应 kmembuckets 数组的下标。kmembucket 结构体定义如下：

```
struct kmembuckets {
        caddr_t kb_next;       /* list of free blocks */
        caddr_t kb_last;       /* last free block */
        long       kb_calls;     /* total calls to allocate this size */
        long       kb_total;     /* total number of blocks allocated */
        long       kb_totalfree;     /* # of free elements in this bucket */
        long       kb_elmpercl;     /* # of elements in this sized allocation */
        long       kb_highwat;       /* high water mark */
        long       kb_couldfree;     /* over high water mark and could free */
};
```

看得出来，这个和内核中的 malloc 的用法几乎是一样的。大致如下图



kmembuckets
数组

```
        s = splimp();
```

这个 spl 是’set priority level’的缩写，在bsd中曾经被用来保护一段代码不被中断，就是产生临界区的功能。一个 bsd 的开发者曾经写道：

when manipulating a disk I/O job queue, you might do:

```
    s = splbio();                   * block "block I/O" (disk controller) interrupts */
    BUFQ_REMOVE(&sc->sc_queue, bp);
    splx(s);                        /* allow "block I/O" interrupts again */
```

pmon 此处的用法其实和这个开发者的例子很相似，if 这段后面就有 splx(s)。顺便提一下，pmon 中有很多来自于 2000 年前后 bsd 内核代码。

```
    if (kbp->kb_next == NULL) {
        kbp->kb_last = NULL;
        if (size > MAXALLOCSAVE)
            allocsize = clmd(round_page(size));
        else
            allocsize = 1 << indx;
        npg = clmd(btoc(allocsize));  //计算这个空间的页数
        va = (caddr_t) kmem_malloc(kmem_map, (vsize_t)ctob(npg),
                                   !(flags & M_NOWAIT));//实际分配空间
        if (va == NULL) {     // 空间不足了
            。。。。。
        }
        kup = btokup(va);//计算这个地址对应的 kmemusage 结构
        kup->ku_indx = indx;
        if (allocsize > MAXALLOCSAVE) {
            。。。
        }

        savedlist = kbp->kb_next;
        kbp->kb_next = cp = va + (npg * NBPG) - allocsize;
        for (;;) {
            freep = (struct freelist *)cp;
            if (cp <= va)
                break;
            cp -= allocsize;
            freep->next = cp;
        }
        freep->next = savedlist;
        if (kbp->kb_last == NULL)
            kbp->kb_last = (caddr_t)freep;
    }
    va = kbp->kb_next;
    kbp->kb_next = ((struct freelist *)va)->next;
```

```
out:
        splx(s);
        return ((void *)va);
}
```

接着执行 startrtclock()，和时钟相关。代码如下：

```
        unsigned long freq;
        freq = tgt_cpufreq () / 4;   /* TB ticker frequency */
```

这个返回外部时钟频率。其实这个名字有点含糊，但 tgt_cpufreq 的注释很清楚，Returns the external clock frequency, usually the bus clock，外部时钟频率。tgt_pipefreq 的注释为 Returns the CPU pipelie clock frequency，cpu 频率。

```
        /* get initial value of real time clock */
        time.tv_sec = tgt_gettime ();
        time.tv_usec = 0;
```

tgt_gettime()会读取时钟芯片上的时间并转化成秒数传回。

```
        clkpertick = freq / hz;
        clkperusec = freq / 1000000;
        _softcompare = get_count() + clkpertick;
        clkenable = 0;
```

这里的 get_count()恒返回 0。_softcompare 就是 clkpertick。以上执行之后，返回 init_net()，接着执行：

```
        mclrefcnt = (char *)malloc(VM_KMEM_SIZE/MCLBYTES,
                                                        M_MBUF, M_NOWAIT);
```

这里的 MCLBYTES 是 2^11，也就是 2KB。

```
        bzero(mclrefcnt, NMBCLUSTERS+CLBYTES/MCLBYTES);
        mb_map = kmem_suballoc(kernel_map, (vm_offset_t *)&mbutl,
                                &maxaddr, NMBCLUSTERS*MCLBYTES, FALSE);
```

接下来执行 mbinit()，代码如下：

```
        s = splimp();
        if (m_clalloc(max(4096 / CLBYTES, 1), M_DONTWAIT) == 0)
                goto bad;
        splx(s);
        return;
bad:
        panic("mbinit");
```

就是调用一个 m_clalloc()，其中的 cl 是 cluster 的缩写。代码如下：

```
        npg = ncl * CLSIZE;
```

从名字看出这个是计算占用多少页。

```
        p = (caddr_t)kmem_malloc(mb_map, ctob(npg), !nowait);
        if (p == NULL) {
```

```
        。。。。。
    }
    ncl = ncl * CLBYTES / MCLBYTES;
    for (i = 0; i < ncl; i++) {
        ((union mcluster *)p)->mcl_next = mclfree;
        mclfree = (union mcluster *)p;
        p += MCLBYTES;
        mbstat.m_clfree++;
    }
    mbstat.m_clusters += ncl;
    return (1);
```

有必要对目前为止所有的分配函数，分配区域和相关结构做一个总结。

Init_net 之 tgt_devconfig

继续看 init_net()，接着执行 tgt_devconfig()，这个递归调用的代码量极大(蓝色)，

```
    _pci_devinit(1);      /* PCI device initialization */
```

这一句作用是初始化 pci 设备配置空间的一些参数，_pci_device 代码就是一个 for 循环，虽然这个 for 的循环体也只执行一遍。代码如下：

```
        struct pci_device *pd;

        for(i = 0, pd = _pci_head; i < pci_roots; i++, pd = pd->next) {
            _pci_setup_devices (pd, initialise);
        }
```

这个_pci_head 我们早见过了。可以理解为北桥，也可以理解为对所有 pci 设备的一个抽象。所有已知的设备（功能）在_pci_head->bridge.child 下。

_pci_setup_devices 函数的代码如下（棕色）：

```
    static void _pci_setup_devices (struct pci_device *parent, int initialise)
    {
        struct pci_device *pd;

        for (pd = parent->bridge.child; pd ; pd = pd->next) {
            struct pci_bus *pb = pd->pcibus;
            pcitag_t tag = pd->pa.pa_tag;
            pcireg_t cmd, misc, class;
```

```
unsigned int ltim;

cmd = _pci_conf_read(tag, PCI_COMMAND_STATUS_REG);
```
读取命令寄存器所在的四个字节。
```
if (initialise) {
    class = _pci_conf_read(tag, PCI_CLASS_REG);
    cmd |= PCI_COMMAND_MASTER_ENABLE
                | PCI_COMMAND_SERR_ENABLE
                | PCI_COMMAND_PARITY_ENABLE;
    cmd |= PCI_COMMAND_IO_ENABLE |
                PCI_COMMAND_MEM_ENABLE;
    if (pb->fast_b2b)
        cmd |= PCI_COMMAND_BACKTOBACK_ENABLE;
    _pci_conf_write(tag, PCI_COMMAND_STATUS_REG, cmd);
```
如果以前总线探测后支持背靠背，则设置每个设备使能这个功能。
```
    ltim = 64;

    misc = _pci_conf_read (tag, PCI_BHLC_REG);
    misc = (misc &
        ~(PCI_LATTIMER_MASK << PCI_LATTIMER_SHIFT))
            | ((ltim & 0xff) << PCI_LATTIMER_SHIFT);
    misc = (misc & ~(PCI_CACHELINE_MASK <<
            PCI_CACHELINE_SHIFT))|
                ((PCI_CACHE_LINE_SIZE & 0xff) <<
                    PCI_CACHELINE_SHIFT);
    _pci_conf_write (tag, PCI_BHLC_REG, misc);

    if(PCI_CLASS(class) == PCI_CLASS_BRIDGE ||
            PCI_SUBCLASS(class) ==
                PCI_SUBCLASS_BRIDGE_PCI ||
                    pd->bridge.child != NULL) {
        _pci_setup_devices (pd, initialise);
```
如果这个设备是个 pci 桥，则递归调用_pci_setup_devices。这个函数实际上并不调用。xiangy 待定
```
    } //if
  } //if
 } // for
}
```
_pci_devinit 就到这里了。

tgt_devconfig之显卡

下面回到tgt_devconfig()执行。现在要执行显卡相关的东西，到现在为止，龙芯pc使用过ATI（2E），SIS(2F 福珑)，SM（2F 逸珑）三款不同的显卡。特别是sis的显卡处理非常复杂，以后可能会解释这块的代码。不过这里我们以SM712的显卡为例走流程。（可以偷点懒不是，^_^），代码如下：

```
    rc = 1;
```

rc反正表示显卡有效，不知道什么的缩写。看注释，下面点灯。

```
    /* light the lcd */
    *((volatile unsigned char *)(0xbfd00000 | HIGH_PORT)) = 0xfe;
    *((volatile unsigned char *)(0xbfd00000 | LOW_PORT)) = 0x01;
    temp = *((volatile unsigned char *)(0xbfd00000 | DATA_PORT));
    /* light the lcd */
    *((volatile unsigned char *)(0xbfd00000 | HIGH_PORT)) = 0xfe;
    *((volatile unsigned char *)(0xbfd00000 | LOW_PORT)) = 0x01;
    *((volatile unsigned char *)(0xbfd00000 | DATA_PORT)) = 0x00;
```

代码中的高端口和低端口组成一个16位的地址，看代码好像是先读出上次的亮度值，再搞成最暗。由于rc为1，接着执行：

```
        fbaddress    =_pci_conf_read(vga_dev->pa.pa_tag,0x10);
        ioaddress    =_pci_conf_read(vga_dev->pa.pa_tag,0x18);
```

在执行_pci_businit()函数的时候会调用_pci_setup_windows，如果是显示设备，就会给vga_dev这个全局变量赋值。这里就是显卡控制器（显卡）。fbaddress和ioaddress表示了显卡想要申请的空间资源。

```
        fbaddress = fbaddress &0xffffff00; //laster 8 bit
        ioaddress = ioaddress &0xfffffff0; //laster 4 bit
        printf("fbaddress 0x%x\tioaddress 0x%x\n",fbaddress, ioaddress);
```

处理对齐需求。

```
        fbaddress |= 0xb0000000;
        ioaddress |= 0xbfd00000;
```

变成我们可分配的虚拟地址。下面开始就要和sm712这个显卡打交道了。

```
        sm712_init((unsigned char *)fbaddress,(unsigned char *)ioaddress);
```

sm712_init代码如下：

```
int    sm712_init(char * fbaddress,char * ioaddress)
{
    u32 smem_size, i;

    smi_init_hw();
```

这个函数很短，就两行代码：

```
    linux_outb(0x18, 0x3c4);
    linux_outb(0x11, 0x3c5);
```

阅读sm712的数据手册，写入0x3c4寄存器的值表示后续寄存器访问的index值，比如这里的linux_outb(0x18, 0x3c4)表是下一句对0x3c5写操作作用于index为0x18的0x3c5寄存器（0x3c5寄存器有上百个index，这个用法很特别）。
0x11表示使用0x0A0000~0xBFFFF的mem空间。32位地址模式，Extemded packed pixel graphics 模式。

```
hw.m_pLFB = SMILFB = fbaddress;
hw.m_pMMIO = SMIRegs = SMILFB + 0x00700000; /* ioaddress */
hw.m_pDPR = hw.m_pLFB + 0x00408000;
hw.m_pVPR = hw.m_pLFB + 0x0040c000;
```

hw这个结构体全局变量是在sm712.h中定义的，用于描述712显卡的属性。上面设置了一些地址。

```
hw.width = SM712_FIX_WIDTH;
hw.height = SM712_FIX_HEIGHT;
hw.bits_per_pixel = SM712_FIX_DEPTH;
hw.hz = SM712_FIX_HZ;
```

设置分辨率为1024x600，16位色，刷新率为60Hz。

```
smi_seqw(0x21,0x00);
```

这个index为21的寄存器基本就是使能。

```
smi_seqw(0x62,0x7A);
```

设置显存与时钟相关的参数并默认使用显存（bit2）。

```
smi_seqw(0x6a,0x0c);
smi_seqw(0x6b,0x02);
```

这两条是合在一块的，用于设置显存频率，计算公式为
14.31818MHz * MNR / MDR
MNR就是CR6a的值，MDR就是CR6b的值。这里我们的显存频率就是66MHz左右。

```
smem_size = 0x00400000;
```

表示显存大小为4MB。

```
for(i = 0; i  < smem_size / 2; i += 4){
    *((volatile unsigned long *)(fbaddress + i)) = 0x00;
}
```

上面的代码表示黑屏，但是循环次数我不理解，

```
*(u32 *)(SMILFB + 4) = 0xAA551133;
if (*(u32 *)(SMILFB + 4) != 0xAA551133)
{
    smem_size = 0x00200000;
    /* Program the MCLK to 130 MHz */
    smi_seqw(0x6a,0x12);
    smi_seqw(0x6b,0x02);
    smi_seqw(0x62,0x3e);
}
```

使用先写后读的方式探测显存是否存在。如果不存在就修改CR62的配置，改为使用内存。

```
    smi_set_timing(&hw);
```

前面的hw赋值总算派上用场了。这个smi_set_timing会匹配设置的这种分辨率，色深，刷新率并使用这个模式下预设的参数配置显卡。细节较多，不叙。

```
    SMI2DBaseAddress = hw.m_pDPR;
    sm712_2d_init();
```

设置2d相关的寄存器。

```
    printf("Silicon Motion, Inc. LynxEM+ Init complete.\n");
    return 0;
}
```

执行完sm712_init后，回到tgt_devconfig，执行：

```
        fb_init(fbaddress, ioaddress);
```

fb_init的代码如下：

```
    int fb_init(unsigned long fbbase, unsigned long iobase)
    {
        pGD = &GD;
        pGD->winSizeX = GetXRes();
        pGD->winSizeY = GetYRes();
        pGD->gdfBytesPP = GetBytesPP();
        pGD->gdfIndex = GetGDFIndex(GetBytesPP());
```

使用显示模式初始化pGD全局变量。gdfIndex为 GDF_16BIT_565RGB

```
        pGD->frameAdrs = 0xb0000000 | fbbase;
```

设置framebuffer起始地址。

```
        _set_font_color();
```

字符显示的配置。

```
        video_fb_address = (void *)VIDEO_FB_ADRS;
```

这里的video_fb_address就是pGD->frameAdes。

```
        memsetl(video_fb_address, CONSOLE_SIZE +
                                (CONSOLE_ROW_SIZE *5), CONSOLE_BG_COL);
```

为什么多清空5行？？？

```
        video_display_bitmap(BIGBMP_START_ADDR, BIGBMP_X,
                                                BIGBMP_Y);
```

这个画图的函数相当长，功能就是把从存放在BIGBMP_START_ADDR这个地址开始的图片内容显示到左上角坐标为（BIGBMP_X,BIGBMP_Y）的位置。

```
        video_console_address = video_fb_address;
                printf("CONSOLE_SIZE %d, CONSOLE_ROW_SIZE %d\n",
    CONSOLE_SIZE, CONSOLE_ROW_SIZE);

        /* Initialize the console */
        console_col = 0;
        console_row = 0;
```

```
        memset(console_buffer, ' ', sizeof console_buffer);

        video_display_bitmap(SMALLBMP0_START_ADDR, SMALLBMP0_X,
                             SMALLBMP0_Y);
        video_display_bitmap(SMALLBMP_START_ADDR_EN_01,
                                 SMALLBMP01_EN_X, SMALLBMP01_EN_Y);
```
显示那个大图片下面的两个带有汉字的小图片。
```
        return 0;
    }
```
回到 tgt_devconfig 中，显卡相关的东西都基本完成了。现在液晶的背光都是关闭的，现在要根据以前保存的亮度重新打开背光。
```
    *((volatile unsigned char *)(0xbfd00000 | HIGH_PORT)) = 0xfe;
    *((volatile unsigned char *)(0xbfd00000 | LOW_PORT)) = 0x01;
    *((volatile unsigned char *)(0xbfd00000 | DATA_PORT)) = temp;
```
好了，和显示相关的基本到这里就完了。咱们接着看 tgt_devconfig 的其它内容。

tgt_devconfig 之 config_init

```
    void config_init()
    {
        TAILQ_INIT(&deferred_config_queue);
        TAILQ_INIT(&alldevs);
        TAILQ_INIT(&allevents);
        TAILQ_INIT(&allcftables);
```
上面初始化了四个链表，开始都为空链。
```
        TAILQ_INSERT_TAIL(&allcftables, &staticcftable, list);
```
这个 staticcftable 内容可不少，一下子都插入到 allcftables 链表中
```
    }
```
先看看看这个 staticcftable 的内容。
```
static struct cftable staticcftable = {
    cfdata
};
```
就是对 cfdata 的一个包装，cfdata 的内容如下：
```
struct  cfdata  cfdata[] = {
/* attachment  driver    unit  state loc     flags parents nm ivstubs starunit1 */
/*   0: mainbus0 at root */
    {&mainbus_ca, &mainbus_cd, 0,  NORM,  loc,    0, pv+1, 0, 0,      0},
/*   1: pcibr0 at mainbus0 */
    {&pcibr_ca,     &pcibr_cd,     0,  NORM,  loc,    0, pv+6, 0, 0,      0},
/*   2: usb* at ohci* */
```

```
        {&usb_ca,       &usb_cd,          0,    STAR,     loc,      0, pv+8, 0, 0,       0},
/*    3: localbus0 at mainbus0 */
        {&localbus_ca, &localbus_cd, 0,    NORM,     loc,       0, pv+6, 0, 0,       0},
/*    4: pci* at pcibr0 bus -1 */
        {&pci_ca,       &pci_cd,          0,    STAR,     loc+1,    0, pv+4, 1, 0,      0},
/*    5: rtl0 at pci* dev -1 function -1 */
        {&rtl_ca,        &rtl_cd,          0,    NORM,    loc+0,    0, pv+0, 3, 0,      0},
/*    6: pciide* at pci* dev -1 function -1 */
        {&pciide_ca,     &pciide_cd,     0,    STAR,    loc+0,      0, pv+0, 3, 0,      0},
/*    7: ohci* at pci* dev -1 function -1 */
        {&ohci_ca,       &ohci_cd,         0,    STAR,    loc+0,      0, pv+0, 3, 0,      0},
/*    8: wd* at pciide* channel -1 drive -1 */
        {&wd_ca,         &wd_cd,           0,    STAR,    loc+0,    0, pv+2, 6, 0,       0},
        {0},
        {0},
        {0},
        {0},
        {0},
        {0},
        {0},
        {0},
        {(struct cfattach *)-1}
};
```

cfdata 的结构体定义如下：

```
struct   cfdata {
        struct    cfattach *cf_attach;       /* config attachment */
        struct    cfdriver *cf_driver;        /* config driver */
        short     cf_unit;              /* unit number */
        short     cf_fstate;          /* finding state (below) */
        int *cf_loc;              /* locators (machine dependent) */
        int cf_flags;             /* flags from config */
        short     *cf_parents;              /* potential parents */
        int cf_locnames;                  /* start of names */
        void      (**cf_ivstubs)          /* config-generated vectors, if any */
                  __P((void));
        short     cf_starunit1;             /* 1st usable unit number by STAR */
};
```

大致介绍一下这个结构体:

cf_attach　包含了设备的匹配，attach，detach 等，

cf_driver　 表示这一类设备的驱动，

cf_unit　　记录这种设备的个数

cf_fstate　 表示搜寻的状态。共有四种状态

#define NORM FSTATE_NOTFOUND 没有发现
#define STAR FSTATE_STAR
#define DNRM FSTATE_DNOTFOUND 不去发现
#define DSTR FSTATE_DSTAR

这些定义的值如下

```
#define FSTATE_NOTFOUND 0 /* has not been found */
#define FSTATE_FOUND 1 /* has been found */
#define FSTATE_STAR 2 /* duplicable */
#define FSTATE_DNOTFOUND 3 //has not been found,and is disabled
#define FSTATE_DSTAR 4 /* duplicable, and is disabled */
```

cf_loc

cf_flags 反正定义的地方全是 0

cf_parents 父亲的 id，是个 short，cf_parents 是 id 数组的下标,等会详述。

cf_locnames 注释说是 start of names，不过我没明白什么意思。

cf_starunit1 对于 cf_flags 为 STAR 的设备，这个参数表示第一个可用的次设备号。

这个 cfdata 数组是通过配置文件../Targers/Bonito2F7inch/conf/Bonito.2F7inch 产生的，配置文件中的部分内容如下。

```
mainbus0 at root
localbus0 at mainbus0
pcibr0 at mainbus0
pci* at pcibr?
rtl0 at pci? dev ? function ?
ohci* at pci? dev ? function ?
usb* at usbbus ?
pciide* at pci ? dev ? function ? flags 0x0000
wd* at pciide? channel ? drive ? flags 0x0000
```

实际上这里已经表明了这个 pci 框架的层次结构。这个框架也可体现 cf_parents 这个成员中。

cfdata 数组中 wd 的 cf_parents 为 pv+2，就是 pv[2]，pv 数组的定义如下：

```
short pv[10] = {
    4, -1, 6, -1, 1, -1, 0, -1, 7, -1,
};
```

pv[2]为 6，这表示它的父亲在 cfdata[6]被描述，cfdata[6]描述的是 pciide 控制器。

pciide 的 cf_parents 为 pv[0]，值为 4，它的父亲就是 cfdata[4]，cfdata[4]描述的是 pci0。

pci 的 cf_parents 为 pv [4]，值为 1，它的父亲就是 cfdata[1]，cfdata[1]描述 pcibr。

pcibr 的 cf_parents 为 pv+6，pv[6]为 0，可见它的父亲是 cfdata[0]，cfdata 描述 mainbus。mainbus 的 cf_parents 为 pv[1]，值为-1，不在这个 cfdata 数组中，表明这个 mainbus 是 root，没有 parents。

tgt_devconfig()之 configure

在 init_net 之前，我们也探测过 pci 总线，不过那时探测的是设备（功能），并不知道这个框架的拓扑结构。而已实际上那些探测的设备并不是我们平常理解的设备，两者的关系比如 IDE 控制器和硬盘，usb 控制器和 u 盘。前面的探测相当于只知道有了哪些控制器。

下面来看看这个 configure，这个可是整个 pci 初始化的核心。

这个函数很简单，实质代码只有如下两句：

```
    if(config_rootfound("mainbus", "mainbus") == 0)
        panic("no mainbus found");
```

也就是调用 config_rootfound 这个函数，参数为两个 mainbus 的字符串。看看：

```
struct device *config_rootfound(char *rootname, void *aux)
{
    void *match;
    if ((match = config_rootsearch((cfmatch_t)NULL, rootname, aux)) != NULL)
        return (config_attach(ROOT, match, aux, (cfprint_t)NULL));
    return (NULL);
}
```

就是两个函数，先看 config_rootsearch((cfmatch_t)NULL, rootname, aux)，再看 config_attach(ROOT, match, aux, (cfprint_t)NULL)。

config_rootsearch 就是传入这个 pci 架构的根的名字，找到根这个设备。我们传入的参数是 NULL 和两个“mainbus”字符串，config_rootsearch 函数代码如下：

```
void *config_rootsearch(cfmatch_t fn, char *rootname, void *aux)
{
    register struct cfdata *cf;
    register short *p;
    struct matchinfo m;

    m.fn = fn;
    m.parent = ROOT;
    m.match = NULL;
    m.aux = aux;
    m.indirect = 0;
    m.pri = 0;
```

这些参数等会都会用到。

```
    for (p = cfroots; *p >= 0; p++) {
```

cfroots 的第二项就是-1，表明这个 pci 架构只有一个 root。所以这个循环只执行一次。 第一项的值为 0，也就是*p 为 0。

```
        cf = &cfdata[*p];
        if (strcmp(cf->cf_driver->cd_name, rootname) == 0)
```

这个就能匹配了，cf就是cfdata[0]。

```
                mapply(&m, cf);
```

这个会改变m.match，config_rootsearch()就是要返回这个match。

```
        }
        return (m.match);
}
```

mapply这个函数后面会多次用到，这里我们就先按照config_rootsearch中调用的上下文，走一下执行流程。函数代码如下：

```
    void mapply(struct matchinfo *m,    struct cfdata *cf)
    {
        register int pri;
        void *match;

        if (m->indirect)
            match = config_make_softc(m->parent, cf);
        else
            match = cf;
```

在config_rootsearch中indirect为0，所以这里match就是cf(cfdata[0])。

```
        if (m->fn != NULL)
            pri = (*m->fn)(m->parent, match, m->aux);
        else {
                if (cf->cf_attach->ca_match == NULL) {
                panic("mapply: no match function for '%s' device",
                    cf->cf_driver->cd_name);
                }
                pri = (*cf->cf_attach->ca_match)(m->parent, match, m->aux);
```

cfdata的match函数就是返回1而已，其余什么都不干。所以pri＝1。

```
        }
        if (pri > m->pri) {
            if (m->indirect && m->match)
                free(m->match, M_DEVBUF);
            m->match = match;
            m->pri = pri;
```

在config_rootsearch中m->pri为0，比pri小。这里把m->match赋值为cfdata[0]的地址，pri赋值为1。这个match就是我们最后要返回的值。

```
        } else {
            if (m->indirect)
                free(match, M_DEVBUF);
        }
    }
```

好了，config_rootsearch找到了叫mainbus的设备，并且返回描述这个设备结构体指针。下面看config_attach(ROOT, match, aux, (cfprint_t)NULL)。这里的ROOT

为 NULL，match 为 cfdata[0]的指针，aux 为字符串”mainbus”，attach 嘛，就是找个依靠，免得一个人孤零零的。

```
struct device *config_attach(struct device *parent, void *match,
                                                    void *aux,   cfprint_t print)
{
    register struct cfdata *cf;
    register struct device *dev;
    register struct cfdriver *cd;
    register struct cfattach *ca;
    struct cftable *t;

    if (parent && parent->dv_cfdata->cf_driver->cd_indirect) {
        dev = match;
        cf = dev->dv_cfdata;
    } else {
        cf = match;
```

没有父亲或者就父亲不给你开后门安排位置，那就自己去天下。

```
        dev = config_make_softc(parent, cf);
```

config_make_softc()主要作用是初始化这个我们所要 attach 的设备，但也顺便给 cf 拉点赞助，更新数据。

```
    }

    cd = cf->cf_driver;
    ca = cf->cf_attach;
    cd->cd_devs[dev->dv_unit] = dev;
```

如果这个 cf 的 config 是第一次被调用，它所扶持的设备一般都是从 0 这个次设备号开始，所以这个 dev->dv_init 为 0。

```
    if (cf->cf_fstate == FSTATE_STAR) {
        if (dev->dv_unit == cf->cf_unit)
            cf->cf_unit++;
```

更新这个类型的设备数目。

```
    } else
        cf->cf_fstate = FSTATE_FOUND;

    TAILQ_INSERT_TAIL(&alldevs, dev, dv_list);
```

找到了一个设备了，加入到 alldevs 中，我们这个 dev 就叫 mainbus0。

```
    device_ref(dev);
```

把这个找到的设备加入到设备列表中，并更新 ref 计数。

```
    if (parent == ROOT)
        printf("%s (root)", dev->dv_xname);
    else {
        printf("%s at %s", dev->dv_xname, parent->dv_xname);
```

```
        if (print)
            (void) (*print)(aux, (char *)0);
    }

    for (cf = t->tab; cf->cf_driver; cf++) {
        if (cf->cf_driver == cd && cf->cf_unit == dev->dv_unit) {
            if (cf->cf_fstate == FSTATE_NOTFOUND)
                cf->cf_fstate = FSTATE_FOUND;
            if (cf->cf_fstate == FSTATE_STAR)
                cf->cf_unit++;
```

根据这个新插入的 dev 看是否有必要改变某些 cf 的属性。

```
        }
    }
    (*ca->ca_attach)(parent, dev, aux);
```

现在用相关的 attach 函数真正的把这个新配置的 dev 依附到他的 parent 上。这里实际调用 mainbus_attach()。这个函数会导致递归调用，这样一层层下去就建立了一个总线框架的拓扑结构，为一个树状结构。

```
    config_process_deferred_children(dev);
    return (dev);
}
```

调用 mainbus_attach 的传入参数是 NULL，描述 manbus0 的 dev，“mainbus”。

```
static void mainbus_attach(struct device *parent, struct device *self, void *aux)
{
    struct mainbus_softc *sc = (struct mainbus_softc *)self;
    struct confargs nca;
```

mainbus_softc 的结构体包含两部分，前面是一个 device 结构体（所以才敢如上赋值），后面是一个 bushook，描述了这个设备挂载的总线的信息，五个成员都在下面被赋值。

```
    sc->sc_bus.bh_dv = (struct device *)sc;
    sc->sc_bus.bh_type = BUS_MAIN;
    sc->sc_bus.bh_intr_establish = NULL;
    sc->sc_bus.bh_intr_disestablish = NULL;
    sc->sc_bus.bh_matchname = mb_matchname;
```

上面表示传入的 dev（self）挂在 mainbus 上，，没有设置中断相关的映射关系。我们在前面看 cfdata 数组定义的时候知道 mainbus 其实有两个下属设备，localbus 和 pcibr，下面就要手动建立联系了。

```
    nca.ca_node = NULL;
    nca.ca_name = "localbus";
    nca.ca_bus = &sc->sc_bus;
    config_found(self, &nca, mbprint);

    nca.ca_node = NULL;
```

```
        nca.ca_name = "pcibr";
        nca.ca_bus = &sc->sc_bus;
        config_found(self, &nca, mbprint);
```

nca 是个 confargs 结构体，在这里起临时变量的作用。在赋值 nca 作为传入参数后，调用config_found()。传入参数还是描述 mainbus0 的那个dev。刚说了，mainbus 有两个儿子，pcibr 和 localbus，这个是静态定义的，代码中写死了。但他们的儿子代码就不可能指定了，要考 config_found 去探测。

下面的两块怀疑是没用的，逻辑上说不通(在只有一个 pci 桥的情况下)，注释掉也没有发现有副作用。

```
        nca.ca_node = NULL;
        nca.ca_name = "pcibr";
        nca.ca_bus = &sc->sc_bus;
        config_found(self, &nca, mbprint);

        nca.ca_node = NULL;
        nca.ca_name = "pcibr";
        nca.ca_bus = &sc->sc_bus;
        config_found(self, &nca, mbprint);
}
```

现在就看 config_found 做了些什么，函数定义如下：

```
#define config_found(d, a, p)      config_found_sm((d), (a), (p), NULL)
```

可见 config_found 就是对 config_found_sm 的包装，函数本质如下：

```
struct device *config_found_sm(struct device *parent, void *aux,
                                        cfprint_t print, cfmatch_t    submatch)
{
        void *match;

        if ((match = config_search(submatch, parent, aux)) != NULL)
                return (config_attach(parent, match, aux, print));
        return (NULL);
}
```

这个函数看着眼熟吧，和 config_rootfound 几乎是一模一样。但 config_rootfound 是点到为止，不往下走，这个函数却会使用递归往下刨。以使用传入参数 aux 代表 pcibr 的调用为例。先看 config_search，这个传入参数为 NULL，代表 mainbus0 的 dev，代表 pcibr 的 nca。

```
void * config_search(cfmatch_t fn, struct device * parent, void * aux)
{
        。。。变量定义。。。

        m.fn = fn;
        m.parent = parent;
        m.match = NULL;
```

```
    m.aux = aux;
    m.indirect = parent && parent->dv_cfdata->cf_driver->cd_indirect;
    m.pri = 0;
```
上面的 indirect 基本上都是 0。
```
    for(t = allcftables.tqh_first; t; t = t->list.tqe_next) {
        for (cf = t->tab; cf->cf_driver; cf++) {
            if (cf->cf_fstate == FSTATE_FOUND)
                continue;
            if (cf->cf_fstate == FSTATE_DNOTFOUND ||
                cf->cf_fstate == FSTATE_DSTAR)
                continue;
            for (p = cf->cf_parents; *p >= 0; p++)
                if (parent->dv_cfdata == &(t->tab)[*p])
                    mapply(&m, cf);
```
这里是我们唯一关心的。对于我们的输入，这里找到的 cf 是 pcibr (cfdata[1])的描述。当然细心的你也许会发现，即使我们的输入的 aux 参数表示的是 pcibr，但是这里的这里的 cf 为 localbus 也是有可能的。因为它的 parents 的确也是 mainbus 呀。关键的东西是 m.pri 和 cf->fstate 参数，在 mapply 中，pri 这个参数有防止重入的功能，体现在 mapply 中的 if(pri > m->pri)这句代码中。这种设计确保所有的儿子都被父亲探测到，不但有顺序，而且不重复。
```
        }
    }
    return (m.match);
}
```
mapply 的代码前面已经看过，对于我们的输入，就是把 m.match 置为 cfdata 数组中描述 pci 的结构体指针而已。返回这个 m.match 后，config_search 函数也返回了，返回值非 NULL 表示 search 到儿子了。

接着执行 config_attach(parent, match, aux, print)。

绕了许久，缘何至此，曾记否？

皆因调 config_rootfound 之 config_attach 之故也。至此，竟成了轮回。

是的，递归了。不过现在还没有看到这个递归的关键环节。都只是例行公事一样通过 attach 插入些设备而已。这里也是是把 pcibr 加入到 alldevs 中罢了。在 config_attach 中有接着执行下一级的 attach，对于 pcibr，就是 pcibrattach。
```
void pcibrattach(struct device * parent,struct device * self, void * aux)
{
    struct pcibr_softc *sc = (struct pcibr_softc *)self;
    struct pcibus_attach_args pba;

    sc->sc_iobus_space = *_pci_bus[sc->sc_dev.dv_unit]->pa.pa_iot;
    sc->sc_membus_space = *_pci_bus[sc->sc_dev.dv_unit]->pa.pa_memt;
    sc->sc_dmatag = *_pci_bus[sc->sc_dev.dv_unit]->pa.pa_dmat;
```

```
        pba.pba_busname = "pci";
        pba.pba_iot = &sc->sc_iobus_space;
        pba.pba_memt = &sc->sc_membus_space;
        pba.pba_dmat = &sc->sc_dmatag;
        pba.pba_pc = NULL;
        pba.pba_bus = sc->sc_dev.dv_unit;
        config_found(self, &pba, pcibrprint);
}
```

就不去看太细节的东西了，很明显有一点，bcibr 明确指定自己的儿子是 pci，并把自己的财产都给了他。接着执行 config_found(self, &pba, pcibrprint)，由于 pcibr 只有一个儿子，返回的自然是 pci 了。接着执行 config_attach，把 pci 插入了。接着递归在 mapply 中调用 pci 的 attach 函数 pciattach。这个 pciattach 和前面的那些 attach 都不太一样，代码比较长。其中的 aux 就是上面的&pba。

```
void pciattach(struct device * parent, struct device *self, void *aux)
{
        struct pcibus_attach_args *pba = aux;
        bus_space_tag_t iot, memt;
        pci_chipset_tag_t pc;
        int bus, device, maxndevs, function, nfunctions;

        iot = pba->pba_iot;
        memt = pba->pba_memt;
        pc = pba->pba_pc;
        bus = pba->pba_bus;
        maxndevs = pci_bus_maxdevs(pc, bus);
```

pci 总线最多能挂 32 个设备，实际上由于硬件特性的原因，一般到不了 20 个。

```
        if (bus == 0)
                pci_isa_bridge_callback = NULL;
```

bus 为 0，callback 为 NULL。下面依顺序探测 pci 设备是否存在（这个工作我们不是第一次做了）

```
        for (device = 0; device < maxndevs; device++) {
                。。。。。。。。

                bhlcr = pci_conf_read(pc, tag, PCI_BHLC_REG);
                    nfunctions = PCI_HDRTYPE_MULTIFN(bhlcr) ? 8 : 1;
```

看这个设备是否为多功能的。

```
                for (function = 0; function < nfunctions; function++) {

                    。。。pa 结构的赋值。。。

                    config_found_sm(self, &pa, pciprint, pcisubmatch);
```

好了，没个功能调用一次config_found_sm，self为pci的cfdata描述，pa为这个功能的描述，pcisubmatch是一个函数，等用到的时候再解释。

```
            }
        }

        if (bus == 0 && pci_isa_bridge_callback != NULL)
            (*pci_isa_bridge_callback)(pci_isa_bridge_callback_arg);
}
```

我们实际上知道pci只有3个儿子，pciide，ohci，rtl。作为功能他们会分别调用config_found_sm，但和前面都不同的是，这里的pcisubmatch非NULL。

```
struct device *config_found_sm(struct device *parent,void * aux,
                                        cfprint_t print, cfmatch_t    submatch)
{
        void *match;

        if ((match = config_search(submatch, parent, aux)) != NULL)
            return (config_attach(parent, match, aux, print));
        if (print)
            printf(msgs[(*print)(aux, parent->dv_xname)]);
        return (NULL);
}
```

我们并不关心print相关的东西，下面我们以pciide的功能识别为例，再来看这个config_found_sm函数。由于传入的submatch为pcisubmatch，在mapply中执行路径就和原来不太一样了。

```
        if (m->fn != NULL)
            pri = (*m->fn)(m->parent, match, m->aux);
        else {
                if (cf->cf_attach->ca_match == NULL) {
                panic("mapply: no match function for '%s' device",
                        cf->cf_driver->cd_name);
                }
            pri = (*cf->cf_attach->ca_match)(m->parent, match, m->aux);
        }
```

我们这里要执行的就是pcisubmatch了。还有这里的传入参数m->aux就是用来描述我们要注册的那3个功能的，aux在之前几乎是去没有去用的。这个pcisubmatch的返回值也是有讲究的，如果传入的match和aux描述的不是一个东西，就返回0，这样pri就是0，这样就保证了mapply返回的match就是我们要它去匹配的那3个设备。如果就是那个match和aux描述的是一个东西。就调用这个设备的match函数。对于pciide，就是调用pciide_match

```
        if (pri > m->pri) {
            if (m->indirect && m->match)
                free(m->match, M_DEVBUF);
```

```
            m->match = match;
            m->pri = pri;
        } else {
            if (m->indirect)
                free(match, M_DEVBUF);
        }
```

除非和 cfdata 中的那三个 pci 的儿子匹配，否则 pri 均为 0，mapply 返回后 m.match 为 NULL。pcisubmatch 的主要功能就是调用那 3 个设备的 match，下面就说说 pciide_match。

这个 match 才叫做 match，函数先判断这个设备是否是 ide 控制器，如果是，调用 pp = pciide_lookup_product(pa->pa_id);这里的 pa ->pa_id 还是我们在 pciattach 中那个循环中探测并保存。pa_id 表示厂商 id。pciide_lookup_product 会匹配那个 ide 控制器的列表，显然我们用的是 CS5536 的南桥的 ide 控制器，厂商 id 就是 AMD，AMD 的 IDE 控制器支持列表如下：

```
const struct pciide_product_desc pciide_amd_products[] =    {
        { PCI_PRODUCT_AMD_PBC756_IDE,      /* AMD 756 */
          0,
          amd756_chip_map
        },
        { PCI_PRODUCT_AMD_CS5536_IDE,      /* AMD CS5536 */
          0,
          amdcs5536_chip_map
        },
};
```

pciide_lookup_product 除了匹配厂商号，还有设备号，这里很清楚的就是 CS5535 嘛。匹配好以后就返回描述 CS5536 的那个结构体描述符。pciide_match 探测好后，返回 1。

这个对于 pci 的儿子 pciide 的 config_search 会返回 pciide 的 cfdata 的描述，之后执行 config_attach，这个函数在插入 pciide 到 alldevs 的设备链表之后会调用 pciide 的 attach 函数，下面看看，很重要哦!!!

```
void pciide_attach(struct device *parent, struct device *self, void *aux)
{
        struct pci_attach_args *pa = aux;

        pcitag_t tag = pa->pa_tag;
        struct pciide_softc *sc = (struct pciide_softc *)self;
        pcireg_t csr;
        char devinfo[256];

        sc->sc_pp = pciide_lookup_product(pa->pa_id);
```

这个函数我们在 pciide 的 match 函数中已经见识过了，就是查找 pmon 支持的 ide 控制器列表，这里会返回描述 CS5536 的 ide 控制器的结构体描述符。内容为：

```
    { PCI_PRODUCT_AMD_CS5536_IDE,     /* AMD CS5536 */
      0,
      amdcs5536_chip_map
    }
```
这个东西等会就要用上了。

```
    sc->sc_pc = pa->pa_pc;
    sc->sc_tag = pa->pa_tag;
    sc->sc_pp->chip_map(sc, pa);
```
这个调用很重要哦。等会就看。
```
    if (sc->sc_dma_ok) {
```
下面看名字貌似是 DMA 相关的，就是一个使能。
```
        csr = pci_conf_read(pc, tag, PCI_COMMAND_STATUS_REG);
        csr |= PCI_COMMAND_MASTER_ENABLE;
        pci_conf_write(pc, tag, PCI_COMMAND_STATUS_REG, csr);
    }
}
```
这个 pciide_attach 的最重要的功能当然是探测有没有（有的话，几个）硬盘挂在硬盘控制器上，并把找到的硬盘以 wd0，wd1 的名字插入 alldevs 的链表中。
代码还是比较繁琐的，我也不大愿意再去看了。阿弥佗佛。指条明路吧。amdcs5536_chip_map->pciide_mapchan->wdcattach->wdcprobe，每个 ide 控制器最多两个 channal，amdcs5536_chip_map 会通过调用 pciide_mapchan->wdcattach 探测这两个 channal 有没有挂载硬盘，真正探测是用过 wdcattach 中的 wdcprobe 函数，这个探测的方法是我们常用的，发命令看有没有回应。如果等了很久还没有收到回应就认为硬盘没有安装。如果硬盘存在，在 wdcattach 中会调用陌生的老朋友 config_found 来插入到 alldevs 中，这样 alldevs 中就有了 wd0 了。

USB

前面简单介绍了一下 wd0 的识别，这里我们看一下 usb。
首先声明，我之前对 USB 的了解是一穷二白，这里特意弄这么一节，纯属自虐。
从 pci 的 attach 开始吧。pciattach 函数会扫描 pci 总线上的所有设备/功能，然后依次调用 config_found_sm。对这个函数，顺着代码看的哥们都快背下了吧。
注意这个 config_found_sm 的调用中 match 函数为 pcisubmatch，不是 NULL，这个对函数执行的流程会造成影响。具体的就是在调用 mapply 的时候，m->fn 非 NULL，这样就会执行 pci 那些儿子的 match，当然也包括 ohci。前面提过的 pci 的儿子有三个，ohci，rtl 和 pciide。如果真的是 ohci 的话，调用 ohci 的 match 会返回 1，赋值给 pri，这样 m->match 就赋值上了，否则 m->match 就返回 NULL。

在 8089 笔记本上，有两个 ohci 控制器，一个是 NEC 的，一个是 5536 内部集成的。所以能从 ohci 的 match 函数返回 1 的就只有两个设备/功能。下面看看 ohci 的 match 函数：

```
if(PCI_CLASS(pa->pa_class) == PCI_CLASS_SERIALBUS &&
    PCI_SUBCLASS(pa->pa_class) == PCI_SUBCLASS_SERIALBUS_USB)
{
        if(((pa->pa_class >>8) & 0xff) == 0x10){
            //printf("Found usb ohci controller\n");
            return 1;
        }
}
```

可见判断的标准很明确的，base class 和 sub class 为 0xc 和 0x3 表明这个设备是串行总线，interface 为 0x10 表明是 ohci，这些都是在 pci 规范的 class code 中定义的，具体可查看 pci2.3 规范的附录。

在 match 上后，最终会在 config_attach 中调用 ohci 的 attach 函数。

ohci_attach 一开始会做些常规的判断并根据硬件连线的情况做些屏蔽处理，比如在 pmon 下 wifi 这个设备（接在 usb 端口）就被屏蔽。接下的代码如下：

```
    if(pci_mem_find(pc, pa->pa_tag, OHCI_PCI_MMBA,
                                &membase, &memsize, &cachable)){
        。。。。。。
    }
```

读取 pci 分配之后这个 ohci 控制器分配到的 mem 地址和大小。具体的分配是在_pci_businit 函数中完成的，这个我们前面已经讲解过了。

```
    if(bus_space_map(memt, membase, memsize, 0, &ohci->sc_sh)){
        。。。。。。
    }
```

从命名上看，和映射有关。bus_space_map 的定义如下：

```
#define bus_space_map(t, addr, size, cacheable, bshp)                \
    ((*(bshp) = (t)->bus_base + (addr)), 0)
```

通过打印，这里的 memt->bus_base 为 0xb0000000，如果你理解龙芯的地址映射过程的话，这种 b 开头的虚拟地址大部分都是映射到 pci 空间的。这里的 size 都是 4KB。

```
    usb_lowlevel_init(ohci);
```

这个函数就不进去看了。其主要内容是 ohci 的结构体成员(hcca，gtd，ohci_dev，control_buf，setup)的空间分配和部分成员(disabled，sleeping，irq，regs，slot_name)的初始化，随后重启 ohci 控制器，最后开启 ohci 控制器。

```
    ohci->hc.uop = &ohci_usb_op;
```

三个 ohci 操作函数指针的初始化。这些函数是会在后面大量调用。3种传输

```
    TAILQ_INSERT_TAIL(&host_controller, &ohci->hc, hc_list);
```

把这个找到的 ohci 控制器加入到 usb 控制器链表中。

```
    ohci->sc_ih = pci_intr_establish(pc, ih, IPL_BIO, hc_interrupt, ohci,
        self->dv_xname);
```

注册中断查询函数。

```
ohci->rdev = usb_alloc_new_device(ohci);
```

在usb_dev数组记录新的ohci设备，并把这个数组项的地址赋值给ohci的成员。usb_alloc_new_device函数非常简单，唯一要注意的是这个新的设备最终可能是有小孩的。

```
/*do the enumeration of  the USB devices attached to the USB HUB(here root hub) ports.*/
usb_new_device(ohci->rdev);
```

从注释就看出来了，这个函数是枚举看是否有连接设备。

对于ohci来说，这个函数是我们最后要看得一个函数了。不幸的是，这个函数非常复杂。

开始的代码是对dev的一些成员的初始化，这个现在也讲不清楚。跳过。

接下来执行一个非常重要的函数，

```
usb_get_descriptor(dev, USB_DT_DEVICE, 0, &dev->descriptor, 8)
```

进入这个函数看看，定义如下：

```
int usb_get_descriptor(struct usb_device *dev, unsigned char type, unsigned char index, void *buf, int size)
{
    int res;
    res = usb_control_msg(dev,        usb_rcvctrlpipe(dev, 0),
                                USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
                                    (type << 8) + index,          0,
                                    buf,    size,    USB_CNTL_TIMEOUT);
    return res;
}
```

代码非常的清楚，先看usb_rcvctrlpipe(dev, 0)这个参数。

从usb_control_msg这个函数的定义中会看到，这个参数叫pipe。好，看这个pipe是如何出来的。参看sys/dev/usb/usb.h中的L314行开始的注释就知道了。

```
max size:     bits 0-1       (00 = 8, 01 = 16, 10 = 32, 11 = 64)
direction:    bit 7          (0 = Host-to-Device [Out], 1 = Device-to-Host [In])
device:       bits 8-14
endpoint:     bits 15-18
Data0/1:      bit 19
speed:        bit 26         (0 = Full, 1 = Low Speed)
pipe type: bits30-31 (00 = isochronous, 01 = interrupt, 10 = control, 11 = bulk)
```

这里的这个pipe和我们说的EP是一对一的关系，这里值是0x80000080，可见解释为控制管道，全速，EP0，设备0，方向输入，maxsize为8。注意ep0是个特殊的ep，必定是用于控制的。

再看usb_control_msg的实现。

首先我们出入的timeout为USB_CNTL_TIMEOUT定义为100，表示100ms内这个get操作要完成。

接着设置setup_packet这个结构体。

```
setup_packet.requesttype = requesttype;
setup_packet.request = request;
setup_packet.value = swap_16(value);
setup_packet.index = swap_16(index);
setup_packet.length = swap_16(size);
```

命名都比较清楚了。requesttype为get descriptor，request为in。value为USB_DT_DEVICE<<8，index为0，size为0，表示8个字节。

这个setup的包我在ohci的规范中没有找到，但是网上有些解释。就不说了。

下面会通过submit_control_msg(dev,pipe,data,size,&setup_packet)去发送这个setup包，要求设备返回设备描述符。

submit_control_msg会原封不动的调用ohci_submit_control_msg。

这个函数很短，首先设置maxsize，由于pipe为0，我们在usb_new_device中定义了EP0的进出包大小都是8字节。这里的maxsize就是8。

之后根据当前要操作的pipe那头是不是root hub调用ohci_submit_rh_msg或者submit_common_msg。由于当前连的是root hub，调用ohci_submit_rh_msg。

参数和ohci_submit_control_msg都是一样的。下面看看这个setup的msg是如何发到root hub上的。注意这时setup_packet的形参叫做cmd了。ohci_submit_rh_msg函数中

在接下来的大switch中，case匹配上RH_GET_DESCRIPTOR，在这个case中的switch中，case匹配0x01，代码会给date_buf赋一个默认的虚拟root hub的内容，并把len置为这个结构的长度。

在这个大switch之后，如果data_buf和目标地址不一致，则这个结构体数据的拷贝(注意长度)。这里stat默认为0，表示正常，如果不正常的话，至于这个虚拟结构的内容在代码中如何解释，用到再说。这个函数返回之后，回到usb_control_msg函数。

回去一看，根本就不判断返回值，可见对这个结果相当的自信，不会有事的。因为和硬件没有接触。

usb_control_msg这个函数最后看dev-<status，这个是在ohci_submit_rh_msg中修改了值的。正常为0，返回读取的长度。

usb_get_descriptor()第四形参

这里的dev->descriptor就是我们传入那个缓冲区。用root_hub_dev_des填充的，这里注意，当前的dev->descriptor就是root_hub_dev_des的地址值。

接着赋值dev->devnum，这个值其实就是root_hub_dev_des中赋值的那个值。

这里首先是1，接着递增，在8089笔记本上就是1和2。

接着执行usb_set_address(dev)。 usb_new_device()中

这个函数直接调用usb_control_msg，这个函数我们见过了，参数不同。看参数。

```
res=usb_control_msg(dev, usb_snddefctrl(dev),
        USB_REQ_SET_ADDRESS, 0,
        (dev->devnum),0,
        NULL,0, USB_CNTL_TIMEOUT);
```

usb_snddefctrl(dev)这个对应的pipe0，这个值为0x80000000，和前面的那个差别就是IN成了OUT。type从名字上看就是设置什么地址而已。传入的data就是dev->devnum，是1。表示要把这个ohci连接的root hub叫做设备1。接着看。
又回到ohci_submit_rh_msg，
这次要做的动作特别简单，就是把dev->rh.devnum赋值为传入的dev->devnum，就是1。仅此。
所以返回应该也没有什么问题。
回到usb_new_device()，接着调用usb_get_descriptor(dev,
USB_DT_DEVICE, 0, &dev->descriptor, sizeof(dev->descriptor))。
这个函数我们前面已经调用过了。不过这次的size参数和上次有不同。到usb_control_msg的层面，pipe的值和前次调用也不同，成了0x80000180，devnum不同了，EP是相同的。

```
usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
                USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
                (type << 8) + index, 0,
                buf, size, USB_CNTL_TIMEOUT);
```

这次((pipe >> 8) & 0x7f) == gohci->rh.devnum，所以ohci_submit_control_msg还是去调用ohci_submit_rh_msg。事实上，只有你在usb口上插了什么东西，才会调用submit_common_msg。因为那时才有除root hub以外的usb device。
这次对于root bug的操作又是get descriptor，实际上这次和上面一次的get_descriptor有任何差别吗？别说大小，其实那个不导致差别。
返回后再次回到usb_new_device()。
注意，由于到现在为止的get descriptor都是直接使用默认的值，所以这里的厂商号，产品号都是0。
接下来是这句，tmpbuf是一个512字节的缓冲区。
usb_get_configuration_no(dev,&tmpbuf[0],0);
这个函数的定义如下：

```
config=(struct usb_config_descriptor *)&buffer[0];
result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno, buffer, 8);
```

这里的cfgno就是0。和前面的usb_get_descriptor最大的不同是，这里的type域是USB_DT_CONFIG，这导致在调用到ohci_submit_rh_msg中case RH_GET_DESCRIPTOR中的case为0x02，按照注释，这次的操作就是获得configuratio descriptor，操作的实现很简单，就是把一个预定义的root_hub_config_des的结构体。
跳过一些错误处理，再执行下面一条，和前面的获得device descriptor一样，这种操作都是每次两遍。

```
result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno, buffer, tmp);
```

实际上，很容易分析，这种两次的操作是没有意义的。但是如果这个get操作是去外部获得的，就不同了。看看ohci的协议，就知道了。第一次会发送命令（8字节），之后从设备响应，这时主设备再去真正获得那个描述符。
这里相当于模拟协议吧，maybe。
获得config之后就是剖析它，执行usb_parse_config(dev,&tmpbuf[0],0)

这个函数非常重要，我们一句句解析。记住我们输入的参数。代码如下：

```
int usb_parse_config(struct usb_device *dev, unsigned char *buffer, int cfgno)
{
    struct usb_descriptor_header *head;
    int index,ifno,epno;
    ifno=-1;
    epno=-1;

    dev->configno=cfgno;
```

这个的值是 0。

```
    head =(struct usb_descriptor_header *)&buffer[0];
```

buffer 对应的就是 root_hub_config_des，在文件 usb-ohci,c 中。

```
    if(head->bDescriptorType!=USB_DT_CONFIG) {
        。。。。。
```

错误处理，这里我们不看

```
    }
    memcpy(&dev->config,buffer,buffer[0]);
```

填充 dev->config.buffer 字段。

```
    dev->config.wTotalLength=swap_16(dev->config.wTotalLength);
    dev->config.no_of_if=0;
```

个别参数的微调。

```
    index=dev->config.bLength;
```

这个 len 是 9。

```
  /* Ok the first entry must be a configuration entry, now process the others */
    head=(struct usb_descriptor_header *)&buffer[index];
```

现在要说的是这个 root_hub_config_des 是由 3 个部分组成的，有每个部分的第一字段标记这部分的大小。前面处理的是第一个部分长度为 9。第二部分是对 interface 的描述，长度为 9，第三部分是对 EP 的描述，长度为 7。总长度在第一部分的 wTotalLength 中记录。如注释所说的，我们要处理其他的部分了。

```
    while(index+1 < dev->config.wTotalLength) {
        switch(head->bDescriptorType) {
            case USB_DT_INTERFACE:
                ifno=dev->config.no_of_if;
                dev->config.no_of_if++;
            memcpy(&dev->config.if_desc[ifno],&buffer[index],buffer[index]);
                dev->config.if_desc[ifno].no_of_ep=0;
```

上面可以看出，如果找到一个 interface，要动作的几个参数就是 config 的 no_of_if, if_desc 数组，其中 if_desc 的 no_of_ep 初始为 0。

```
                break;
            case USB_DT_ENDPOINT:
```

```
                epno=dev->config.if_desc[ifno].no_of_ep;
                dev->config.if_desc[ifno].no_of_ep++;
                memcpy(&dev->config.if_desc[ifno].ep_desc[epno],
                                                &buffer[index],buffer[index]);
                dev->config.if_desc[ifno].ep_desc[epno].wMaxPacketSize

=swap_16(dev->config.if_desc[ifno].ep_desc[epno].wMaxPacketSize);
                break;
```

EP 和 interface 的关系从这里可以一览无余。一个 interface 包含了 n 个 EP。所以上面的复制都是对于 config.if_desc[ifno]的结构体的赋值。

```
            default:
              。。。
        }
```

好，这个函数就解释完了。

再回到 usb_new_device。

接下的 usb_set_maxpacket(dev)不过就是设置收发包的大小。注意这个双重循环实际执行一次，因为当前只有一个 interface 且那个 interface 只有一个 EP。

接下来还是重复昨天的故事，get 以后是 set，这次 set configuratio。

中间的过程都熟悉了，看点关键的（ohci_submit_rh_msg 中）。

```
    case RH_SET_CONFIGURATION:    WR_RH_STAT (0x10000);
```

这个宏的定义如下：

```
#define WR_RH_STAT(x)           writel((x), &gohci->regs->roothub.status)
```

writel 的定义就是赋值而已。

```
#define writel(val, addr) *(volatile u32*)(((u32)(addr)) | 0xa0000000) = (val)
```

这些代码都简单，问题是为什么要写这个 status。一个更让我困惑的问题是，这里的写为什么要或上 0xa0000000？难道这个地址是硬件映射到物理地址空间的？看上去应该是的。打印出这个地址我们会觉得比较眼熟。在 USB 这一节的开始，有个 bus_space_map 的函数，这个函数是处理 usb 的寄存器到 mem 空间的映射，而现在发现这个地址就在那段空间内，所以至少可以确定 gohci ->regs->roothub.status 的地址就是 usb 的寄存器映射过来的。打出 gohci->regs 的地址，就是或上 0xb0000000 的 usb 的 membase 地址。至于这个 status 为 0x10000 是什么？也许是 active 吧。

这个 set configration 顺利返回之后，赋值 dev->toggle 的前两个元素为 0。

回到 usb_new_device，清 dev 的 mf，prod，serial 三个成员。

接着执行 usb_hub_probe(dev,0)，检测这个 ohci 上的接口 0 是不是 root bug，如果是，配置之。

检查比较简单，开始就是些参数的对比。从对比中可以产生这样的理解：

root hub 是一个 interface，

这个 interface 只有一个 EP。

在这个 probe 函数末尾有一个 usb_hub_configure(dev)的调用。

这个用于 hub 这个结构的注册。

这个代码首先占用 hub_dev 数组的一项。之后调用

usb_get_hub_descriptor(dev, buffer, 4)

这个函数会调用老朋友 ohci_summit_control_msg，这次会遇到的 case 不同，是 case RH_GET_DESCRIPTOR | RH_CLASS，注意到 buffer 返回后是作为 usb_hub_descriptor 的结构体指针使用的，在 data_buf 数组赋值的时候就最好对着 usb_hub_descriptor 的定义看，就比较清楚了。

接着又非常传统的再来一遍，接着把读到的（代码写死的那些）赋给 hub_dev 的 desc 成员。

接着调整 desc 中的部分成员，DeviceRemovable 和 PortPowerCtrlMask 的初始化。具体的功能不清楚。

status 那段跳过。接下来的函数 usb_hub_power_on(hub)和供电有关，再接下是每个 port 的简单配置。这些都挺重要的，有时间可以看看。

这个 usb_new_device 总算快到头了，末尾有个老朋友，config_found。很简单，要看看这个 ohci 上有没有连接什么 usb 设备。

配置的时候就确定了，ohci 的儿子就是 usb。

config_found 我们都知道了，主要就是执行一个 match 和 attach，核心是设备的这两个函数。对于 usb 就是 usb_match 和 usb_attach。

usb_attach 主要代码如下：

```
if(usb_storage_probe(dev,0,&usb_stor[usb_max_devs])) {
        memset(&usb_dev_desc[usb_max_devs],0,sizeof(block_dev_desc_t));
        usb_dev_desc[usb_max_devs].target=0xff;
        usb_dev_desc[usb_max_devs].if_type=IF_TYPE_USB;
        usb_dev_desc[usb_max_devs].dev=usb_max_devs;
        usb_dev_desc[usb_max_devs].part_type=PART_TYPE_UNKNOWN;
        usb_dev_desc[usb_max_devs].block_read=&usb_stor_read;
         /* get info and fill it in */
        if(usb_stor_get_info(dev, &usb_stor[usb_max_devs],
                        &usb_dev_desc[usb_max_devs])){
            usb_max_devs++;
            return 1;
        }
    }
```

从代码上看，这个过程是首先探测 usb 存储设备是否存在，如果存在就填充 usb_stor 的一项，之后初始化 usb_dev_desc 的那一项，最后调用 usb_stor_get_info 完成探测。

先看第一个函数 usb_storage_probe，我们现在以没有外接任何 usb 存储设备的例子来看。很快的，在开头的 if 条件上就返回 0 了，因为的 descriptor 的 bDeviceClass 值为 9，不是 0，所以返回 0。整个过程也就结束了。

现在我们以在 NEC 接出来的 usb 口插了一个 U 盘为例走流程。

首先要搞清楚插了 U 盘，ohci 是如何知道的。前面我们跳过了，现在回头。

在 usb_new_device 的末尾有一句 usb_hub_probe，这个函数会判断当前设备是不是 usb 的 hub，如果是，调用 usb_hub_configure，在初始化完一个 hub 结构之后。hub 嘛，就是一个疙瘩上好几个洞。在给每个 port 上电以后。探测每个 port 的状

态。
如果某个 usb 口接了 usb 设备(除了被 mask 掉的)，那么接着就执行

```
usb_get_port_status(dev, i + 1, &portsts)
```

这个会从老路调用 ohci_submit_control_msg()，不同的是匹配下面的 case

```
case RH_GET_STATUS | RH_OTHER | RH_CLASS:
        *(u32 *) data_buf = m32_swap (RD_RH_PORTSTAT); OK (4);
```

其中的 RD_RH_POSTSTAT 会读取 root hub 中那个 port 的状态，并返回。如果有插个什么东西，这个返回值的 portchange 为 1，表示 connection，有挂 usb 设备。其实从这个 status 的获取中我们可以确定，这个寄存器就是映射到 dev->reg 中的。从代码我们可以看到，这个 portchange 的值表征的信息是非常丰富的，具体见代码的不同条件分支。
由于发现了有 usb 挂载，执行 usb_hub_port_connect_change(dev, i)，初始化挂载在 root hub 上的那个设备，代码如下：

一开始重读 status，代码中指出了如何区分低速和高速设备。代码略。

```
/* Clear the connection change status */
usb_clear_port_feature(dev, port + 1,
                       USB_PORT_FEAT_C_CONNECTION);
```

清状态寄存器，以免下次重复探测到。

```
/* Disconnect any existing devices under this port */
```

某些端口的比较特殊，不允许连接。代码略。

```
wait_ms(200);

if (hub_port_reset(dev, port, &portstatus) < 0) {
        。。。
}
```

没细看实现，相当于使能？？？

```
//wait_ms(200);
wait_ms(400);
/* Allocate a new device struct for it */
assert(dev->hc_private!=NULL);
usb=usb_alloc_new_device(dev->hc_private);
usb->slow = (portstatus & USB_PORT_STAT_LOW_SPEED) ? 1 : 0;
dev->children[port] = usb;
usb->parent=dev;
```

这里的初始化比较明显了，注册一个新的 usb 设备，并和 dev 连接上。dev 总算有儿子了。并且 usb 的 hc_private 和其父是共享的。

```
usb->port = port;

if (usb_new_device(usb)) {
        。。。
}
```

}

看到了吧，又递归了。我们就调用 usb_new_device 来处理这个动态插入的家伙。这个 usb_new_device 和前面的那次调用有很大的不同，首先这个设备是接在 root hub 上的，是一个我们所无法预知的设备（U 盘？usb 鼠标？usb 键盘），在代码上，这个设备的一切都要通过真正的探测，而不是像前头有时是走过场，用预定义的数据敷衍了事。所以要重新走一遍 usb_new_device 的流程。

Let's go !

首先看 usb_get_descriptor 这个调用，前面 ochi 调用这个函数的时候，就是给 ohci ->descriptor 赋值为一个预设的结构体指针。看看这次有什么不同。

最大的不同是对 ohci_submit_rh_msg 的调用都变成了 submit_common_msg.。

一看内容，这两个函数完全不同。代码虽长，一行行看吧。如下：

```
    if(pipe != PIPE_INTERRUPT) {
        gohci->transfer_lock ++;
        gohci->hc_control &= ~OHCI_CTRL_PLE;
        writel(gohci->hc_control, &gohci->regs->control);
    }
```

control 中的 PLE 是 periodic list enable 的缩写。在 ohci 中，数据传输的类型可分为四类：interrupt，isochronous，control，bulk。interrupt 和 isochronous 是周期性的。ohci 对于 list 的管理还是比较详细的，这里的 periodic list 就是指 interrupt 和 isochronous 的任务 list 吧。代码中是 disable 的。

```
    if (sohci_submit_job(dev, pipe, buffer, transfer_len, setup, interval) < 0) {
        。。。
    }
```

这个函数和 ohci 的核心机制相关，极为重要。大致意思就是把这个 job 化为一个或者几个 TD，链到 ED 中，等待被执行。

```
#define BULK_TO    500       /* timeout in milliseconds */
    if (usb_pipetype (pipe) == PIPE_BULK)
        timeout = BULK_TO;
    else
        timeout = 2000;g

    timeout *= 40;
```

bulk 的 timeout 值明显要小一些。

```
    while(--timeout > 0) {
        if(!(dev->status & USB_ST_NOT_PROC)){
            break;
        }
        delay(200);
        hc_check_ohci_controller(gohci);
    }
```

这个函数基本功能是查询 done list。

如果 done list 非空，就看这些干完的 TD 是什么，并把它从那个 ED 的 TD 链中 unlink 掉。并在 dl_td_done_list 中修改那个 td 对应的 dev 的 status。上面的 while 就是等这个 dev 的 status 改变。当然在这个 dev 改变的同时（或者没改变时）其它的 dev 的 status 也可能改变，只要在 td done list 有它的事完了。
剩下的所有代码略过。
最后插入的 usb 设备执行 config_attach，基本是调用这个 ohci_common_msg。我们调用 pci 设备树上的儿子 usb 的 match 和 attach 函数去了。
有兴趣的可以接着看 usb_storage.c 中的代码。我走了。

回到 init_net

流水落花春去也，换了人间。
下面的代码不会那么让人头大了。当然我的头本来就挺大的。接着的代码如下：

```
for (pdev = pdevinit; pdev->pdev_attach != NULL; pdev++) {
    if (pdev->pdev_count > 0) {
        (*pdev->pdev_attach)(pdev->pdev_count);
    }
}
```

pdevinit 定义如下：

```
struct pdevinit pdevinit[] = {
    { loopattach, 1 },
    { 0, 0 }
};
```

pdevinit 的结构体定义如下：

```
struct pdevinit {
    void    (*pdev_attach) __P((int));
    int pdev_count;
};
```

可见上面按个 for 循环其实就是调用 loopattach(1)而已。loop 是最简单的网络收发模型，在 pmon 下没有什么意义，不看了。
接下来执行 ifinit()。看得出来，是网络系统的初始化，if 就是 interface 的缩写吧。

```
void  ifinit()
{
    register struct ifnet *ifp;

    for (ifp = ifnet.tqh_first; ifp != 0; ifp = ifp->if_list.tqe_next)
        if (ifp->if_snd.ifq_maxlen == 0)
            ifp->if_snd.ifq_maxlen = ifqmaxlen;
```

这个 ifnet 是个全局变量，功能是描述所有网卡的 ifnet 结构体的链表头。实际上大部分情况下我们都只有一个网卡。在 pmon 下没有无线网卡的驱动，虽然会被探测到，但并不会被加入到设备链表中。
实际上对于 8089 的小本，在 pmon 下唯一的网络设备就是 8169。

```
    if_slowtimo(NULL);
}
```

init_net()总算执行完毕，回到 dbginit()。
执行 histinit()，初始化命令的历史记录，这个历史记录的功能还是很有用的。
这个函数就一句话 ： histno＝1；表示现在历史列表中有一个命令(实际上是 0)。
接着执行 syminit()，实际上就是一行代码：

```
    defsyms ((u_int32_t)sbrk(0) + 1024, memorysize |
                                        (CLIENTPC & 0xc0000000), CLIENTPC);
```

执行的函数代码如下：

```
    defsyms (u_int32_t minaddr,   u_int32_t maxaddr,   u_int32_t pc)
    {
        extern int          start[];
        defsym ("Pmon", (unsigned long) start);
        if (minaddr != ~0)
        defsym ("_ftext", minaddr);
        if (maxaddr != 0)
        defsym ("etext", maxaddr);
        defsym ("start", pc);
    }
```

defsym()函数把这些变量注册在一个 hash 数组中。至于这里的 CLIENTPC，不知道是干什么的。
接下来执行 initial_sr |= tgt_enable (tgt_getmachtype ());
tgt_getmachtype ()就是调用 md_cputype，定义在 arch/mips/mips.S，如下：

```
LEAF(md_cputype)
    mfc0    v0, COP_0_PRID
    j   ra
    nop
END(md_cputype)
```

就是返回 prid 而已。
tgt_enable 也只有一句话：return(SR_COP_1_BIT|SR_FR_32|SR_EXL);
就是设置协处理器 1 有效，可以使用 32 个急存器，重新设置例外处理为启动模式。现在又看不出调用 tgt_getmachtype ()有什么用了。

接下来执行：

```
/* Set up initial console terminal state */
ioctl(STDIN, TCGETA, &consterm);
```

``````````````````````
``````````````
``````````

tgt_machprint()会调用 md_cpuname()，判断打印出 cpu 的类型，其实前面就算出来了，没有使用。无语中。。。。。

tgt_machprint()打印些 cache 的情况。

接下来执行 md_clreg(NULL)，函数就一句：tf = cpuinfotab[whatcpu]，这个 cpuinfotag 数组在调用 dbginit 前初始化 cpuinfotab[0] = &DBGREG;其实 DBGREG 只是空的一段而已。

接着执行 md_setpc(NULL, (int32_t) CLIENTPC)，就是把 cpuinfotab[0]->pc 赋值为 CLINETPC。md_setsp(NULL, tgt_clienttos ())看名字是设置 sp 寄存器，sp 的值是 256MB－64 字节的地方。别忘了栈是向下生长的。

接着调用 DevicesInit();

# DevicesInit

函数功能很清楚，就是挨个查询已经注册的设备,所有已知设备都在 alldevs 这个链表中。如果是 disk 设备，就分配一个 DeviceDisk 结构体，填充名字后插入以 gDivice 为首的链表中。一般都是一个硬盘，名字就是 wd0。DevicesInit()的重头戏是_DevPartOpen(dev_disk, dev_disk->device_name)，代码如下：

```
int cnt = 0;
char path[256];

strcpy(path, dev_name);
if (strncmp(dev_name, "/dev/", 5) != 0)
{
    sprintf(path, "/dev/disk/%s", dev_name);
}
```

对于只有一个硬盘的情形，现在 path 的内容是/dev/disk/wd0。

```
fd = open(path, O_RDONLY | O_NONBLOCK, 0);
```

open()函数定义在../lib/libc/open.c。暂时可以这么理解这个函数：open 就是看看那文件是否存在，如果存在，就分配一个文件 id，这个 id 是一个结构题数组的下标，那个结构体用来保存相关的信息。
```

```
if (fd < 0)
{
    。。。// error when open
}
cnt = dev_part_read(fd, &dev->part);
```
现在开始读取 disk 上的分区，dev->part 传入的目的是存放读取出的分区表
```
if (cnt <= 0)
{
    printf("no partitions\n");
    close(fd);
    return -1;
}
close(fd);
```

这个函数主要调用 dev_part_read(),读取硬盘上的分区表。下面看看代码。
```
int cnt;
DiskPartitionTable* table = NULL;
DiskPartitionTable* p;
int number = 0;

if (ppTable != NULL)
{
    *ppTable = NULL;
}
cnt = read_primary_part_table(fd, 0, &table);
```
这个函数将会读取 fd 代表的 disk 上分区表写到 table 这个地址去。
```
if (cnt <= 0)
{
    return 0;
}
number = cnt;
p = get_extemed_part(table);
```
检测一下有没有扩展分区。
```
if (p != NULL)
{
    /* Found extended partition */
    cnt = dev_logical_read(fd, p);
```
如果存在扩展分区，那么就把他加到分区表中
```
    number += cnt;
```
number 表示分区的个数（主分区+扩展分区）
```
}
```

```
if (ppTable != NULL)
{
    *ppTable = table;
}
```

读取主分区的函数 read_primary_part_table()函数的主题如下：

```
if ((read(fd, leadbuf, SECTOR_SIZE)) != SECTOR_SIZE)
{
    free(leadbuf);
    return 0;
}
```

现在硬盘上的第一个扇区被放到首地址为 leadbuf的缓冲区中，关于这个首扇区，有必要先解释一下：

MBR 中的分区信息在偏移地址 01BEH--01FDH 的 64 个字节，为 4 个分区项内容（分区信息表）。FDISK 对一个磁盘划分的主分区可少于 4 个，但最多不超过 4 个。每个分区表的项目是 16 个字节。

分区表的内容格式如下：

第 0 字节 是否为活动分区，是则为 80H，否则为 00H

第 1 字节 该分区起始磁头号

第 2 字节 该分区起始扇区号（低 6 位）和起始柱面号（高 2 位）

第 3 字节 该分区起始柱面号的低 8 位

第 4 字节 文件系统的类型标志。0x83 表示 ext2/ext3，0x5 表示扩展分区。

第 5 字节 该分区结束磁头号

第 6 字节 该分区结束扇区号（低 6 位）和结束柱面号（高 2 位）

第 7 字节 该分区结束柱面号的低 8 位

第 8~11 字节 相对扇区号，该分区起始的相对逻辑扇区号，高位在后低位在前

第 12~15 字节 该分区所用扇区数，高位在后，低位在前。

以我自己的机子为例，120G 的硬盘。分区情况如下：

```
[root@localhost ~]# fdisk -l
Disk /dev/sda: 120.0 GB, 120034123776 bytes
255 heads, 63 sectors/track, 14593 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
  Device    Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1        3951    31736376   83  Linux
/dev/sda2            3952        7966    32250487+  83  Linux
/dev/sda3            7967       10643    21503002+  83  Linux
/dev/sda4           10644       14593    31728375    5  Extended
/dev/sda5           10644       12644    16072969+  83  Linux
/dev/sda6           12645       14593    15655311   83  Linux
```

前面 3 个是主分区，第四个是扩展分区，包含了两个逻辑分区。

在 os 下读取 mbf 的命令为 dd if=/dev/sda of=part_table bs=1 count=512，在龙芯上把 sda 改为 hda 就行了。

pmon只使用这个首扇区中的第446～510字节之间64字节的内容，使用hexdump看到内容如下：

00001b0 0000 0000 0000 0000 a2e0 624a 0000 0180
00001c0 0001 fe83 ffff 003f 0000 8470 03c8 fe00
00001d0 ffff fe83 ffff 84af 03c8 34ef 03d8 fe00
00001e0 ffff fe83 ffff b99e 07a0 3835 0290 fe00
00001f0 ffff fe05 ffff f1d3 0a30 45ee 03c8 aa55
0000200

排版后我们关心的 4 个主分区内容如下：

1 0 3 2 5 4 7 6 9 8 11 10 13 12 15 14

0180 0001 FE83 FFFF 003F 0000 8470 03C8
FE00 FFFF FE83 FFFF 84AF 03C8 34EF 03D8
FE00 FFFF FE83 FFFF B99E 07A0 3835 0290
FE00 FFFF FE05 FFFF F1D3 0A30 45EE 03C8

好，现在开始解析这个分区表吧。

首先看第 4 个字节，前面 3 个主分区都是 83H，表明都是 ext3 文件系统，第四个主分区是 0x5，表明是扩展分区，至于扩展分区如何处理，等会看了代码就知道了。第 8～11 个字节的内容是分区的起始扇区，比如第一个分区的起始扇区为 0x37，第二个分区的起始扇区为 0x3c884af,所以第一个分区的大小就是 0x3c88470(0x3c884af-0x370)个扇区。我们看到 fdisk -l 的时候的 sda1 描述为：

| Device | Boot | Start | End | Blocks | Id | System |
|---|---|---|---|---|---|---|
| /dev/sda1 | * | 1 | 3951 | 31736376 | 83 | Linux |

大小为 31736376(0x1e44238)个块，在创建文件系统的时候，mkfs.ext3 允许指定块大小，默认为 1KB，也就是两个扇区的大小，所以这个分区就有 63472752 (0x3c88470)个扇区，和分区信息中第 12~15 个字节的表示是一致的。

差不多了，看代码吧。

```
//search the partion table to find the partition with id=0x83 and 0x05
for (cnt = 0, i = 446; i < 510; i += 0x10)
{
    tag = leadbuf[i + 4];
    sec_off = get_logical_part_offset(leadbuf + i);
    size = get_part_size(leadbuf + i);
    if (tag == 0 && sec_off == 0 && size == 0)
    {
        id++;
        continue;
    }

    part = (DiskPartitionTable *)malloc(sizeof(DiskPartitionTable));
    if (part == NULL)
```

```
            {
                continue;
            }
            memset(part, 0, sizeof(DiskPartitionTable));
```
下面开始填充分区的各种信息
```
            part->tag = tag;
            part->id = id;
            part->bootflag = 0;
            part->sec_begin = sec_off;
```
对于扩展分区，sec_begin 这个成员是非常关键的。
```
            part->size = size;
            part->sec_end = part->size + part->sec_begin;
            part->Next = NULL;
            part->logical = NULL;
            part->fs = NULL;

            /* file system */
#ifndef DEBUG
            get_filesystem(fd, part);
#endif
```
把这个探测到的分区插入分区表中。
```
            part_node_insert(table, part);
            cnt++;
            id++;
        }
```
以上是主分区的探测，这个函数返回 dev_part_raad()后，开始探测扩展分区并注册逻辑分区，代码如下：
```
    p = get_extemed_part(table);
```
是否为扩展分区的判断如下，
```
#define IS_EXTENDED(tag) ((tag) == 0x05 || (tag) == 0x0F || (tag) == 0x85)
```
可见不是只有 0x5 才表明是扩展分区哦。
```
    if (p != NULL)
    {
        /* Found extended partition */
        cnt = dev_logical_read(fd, p);
```
遇到了扩展分区，开始读取逻辑分区
```
        number += cnt;
    }
```
dev_logical_read()函数的定义如下：
```
static int dev_logical_read(int fd, DiskPartitionTable* extended)
{
    DiskPartitionTable* table = NULL;
```

```
    DiskPartitionTable* p1;
    __u32 base = 0;
    int cnt;
    int id = 5;

    base = extended->sec_begin;
```
看到了吧，扩展分区的逻辑分区信息在 base 扇区中存放着呢。
```
    p1 = extended;
    while (1)
    {
```
和注册主分区最多只有四个不同不同，扩展分区的逻辑分区还可以是扩展分区，所以只能用 while 了
```
        table = NULL;
        cnt = read_logical_part_table(fd, id, base, &table);
```
这个 read_logical_part_table 和读取主分区表的方法几乎一样，就不进去看了。
```
        if (cnt <= 0)
        {
            return 0;
        }

        /* delete extended part */
        p1 = remove_extended_part(&table);
```
在 pmon 中分区表中删除这个扩展分区，扩展分区和普通分区不用，实际上就是个皮包公司，不行，你有本事 mount 扩展分区试试。
```
#ifndef DEBUG
        get_filesystem(fd, table);
#endif
        part_node_insert(&extended->logical, table);
```
插入这些探测到的新分区。
```
        if (p1 == NULL)
        {
            break;
        }

        base = extended->sec_begin + p1->sec_begin;
        free(p1);
        id++;
    }
    return id - 5 + 1;
}
```
这些函数都返回后，pmon 会把包含了所有分区表信息的描述 disk 的 DeviceDisk 结构体指针加入到 gDivece 链表中。

好了，分区表就这么点事，算是讲完了。

# open 函数

上面我们已经遇到了 open 函数，不过当时没有进入代码，现在我们就以默认启动时要执行的 open(“(wd0,0)/boot/boot.cfg”,O_RDONLY | O_NONBLOCK)为例分析这个函数。

open()函数定义在../lib/libc/open.c

```
int open(const char *filename, int mode)
{
        char    *fname;
        char    *dname;
        int     lu, i;
        int     fnamelen;
```

首先是一些基本参数的处理

```
        fnamelen = strlen(filename);
        fname = (char *)malloc(fnamelen+1);
        memcpy(fname, filename, fnamelen+1);

        for (lu = 0; lu < OPEN_MAX && _file[lu].valid; lu++);
```

至此，lu 表示当前可用的文件描述符 id。

```
        if (lu == OPEN_MAX) {
            ermo = EMFILE;
            free(fname);
            return -1;
        }
```

ok，lu 有效。这个 lu 可是非常重要。

接下来根据 file 的开始符号作不同的处理。比如 al 我们一般设置为/dev/fs/ext2@wd0/boot/vmlinux，就进入第一个 if，tftp 那种情形就进入第二种情形。我们的例子是(wd0,0)开始的，我们跳到那个地方去。

```
        dname = (char *)fname;
        if (strncmp (dname, "/dev/", 5) == 0) {
                dname += 5;
                i = __try_open(fname, mode, dname, lu, 0);
                free(fname);
                return i;
         }
         else if (strpat(dname, "*ftp://*")) {
                i = __try_open(fname, mode, "net", lu, 0);
         }
```

```
        else if (strpat(dname, "file://*")) {
                dname += 6;
                i = __try_open(dname, mode, NULL, lu, FS_FILE);
        }
        else if (*dname == '(') {
                i = __try_open(fname, mode, "fs", lu, 0);
```

可见如果是’(’开头的。这时候调用的实际就是__try_open(“’(wd0,0)/boot/boot.cfg”,只读非阻塞，“fs”，文件描述符，0)。

```
        } else {
                i = __try_open(fname, mode, dname, lu, 0);
        }
        free(fname);
        return i;
}
```

可见 open 函数本身是比较简单的，我们顺着调用路径到看看_try_open()。这个函数调用合适的 open 函数函数并修改 File 结构体。代码如下：

```
static int
__try_open(const char *fname, int mode, char *dname, int lu, int fstype)
{
        FileSystem *fsp;
```

首先要看欲打开文件的文件类型，我们的传入参数为“fs”，不是具体的文件系统，表示打开硬盘。传入的 fstype 为 0，表示 FS_NULL。FileSystems 在 init()中的文件系统注册时初始化为一个链表。

```
        SLIST_FOREACH(fsp, &FileSystems, i_next) {
            if (dname && strbequ(dname, fsp->devname)) {
```

在../pmon/fs/diskfs.c 的注册中我们能看到 diskfs 的 devname 就是 fs，就是它了。

```
                if(fsp->open && (*fsp->open)(lu, fname, mode, 0) != lu)
```

这个 open 是比较真正的 open 了，当然最终的 open 是硬盘驱动的 open，稍后就解析。

```
                    return -1;    /* Error? */
                break;
            }
            else if (fstype != FS_NULL && fsp->fstype == fstype) {
                if(fsp->open && (*fsp->open)(lu, fname, mode, 0) == lu)
                    break;
            }
        }

        if (fsp) {
            /* Opened */
```

如果已经打开有效的文件，那么就占用这个文件资源吧

```
            _file[lu].valid = 1;
```

```
            _file[lu].posn = 0;
            _file[lu].fs = fsp;
            return(lu);
        }
```

open()本身就是这些。我们顺着执行流到 diskfs 的 open 函数去吧。调用参数为 diskfs_open(disk_fs(文件描述符,"(wd0,0)/boot/boot.cfg")，只读非阻塞，0)。

open 函数到现在为止，基本的调用是

```
open()->
    try_open()->
        diskfs_open()->
```

函数调用基本返回后都没有什么动作了，只是些错误检测而已。

diskfs_open()函数代码在../pmon/fs/diskfs.c 中，如下：

```
static int diskfs_open (int fd, const char *fname, int mode, int perms)
{
    DiskFile *dfp;
    char filename[256];

    strcpy(filename, fname);
    dfp = GetDiskFile(fname, filename);
```

这个 GetDiskFile()用来解析这个 fname，分离出 3 个重要内容，稍后讲述

```
    if (dfp == NULL)
    {
        return -1;
    }
    _file[fd].posn = 0;
    _file[fd].data = (void *)dfp;
    if(dfp->dfs->open)
        return ((dfp->dfs->open)(fd, filename, mode, perms));
```

这个就是根据具体的文件系统的 open 函数，去打开文件，我们这是 ext2，dfp->dfs 就是 ext2 文件系统的描述指针，执行的打开函数就是 ext2_open

```
    return -1;
}
```

diskfs_open 主要调用两个函数，一个是 GetDiskFile，另一个是文件系统的 open 函数，先来看 GetDiskFile。

```
static DiskFile* GetDiskFile(const char* fname, char* filename)
{
    DeviceDisk* pdev;
    DiskPartitionTable* part;
    DiskFile *dfp;
    char *dname;
    char *fsname = NULL;
    char *devname = NULL;
```

```
        int i;
        char* p;

        dname = (char *)fname;

        if (strncmp (dname, "/dev/fs/", 8) == 0) {
            。。。。。
        } else if (*dname == '(') {
```

我们进入的就是这个路径。

```
            devname = dname + 1;
            p = strchr(devname, '/');
```

strchr 这个函数的功能是在 devname 字符串中查找’/’，返回第一个’/’的位置。

```
            if (p == NULL)
            {
                strcpy(filename, "");
            }
            else
            {
                strcpy(filename, p + 1);
```

现在 filename 就是”boot/boot.cfg“了

```
            }

            p = strchr(dname, ',');
```

查找逗号，当前 dname 是“wd0,0)”，有逗号的。

```
            if (p == NULL)
            {
                printf("device format is error[%s]\n", dname);
                return NULL;
            }
            *p = '\0';
```

在字符串中把逗号变成’\0’，现在 devname 实际就是字符串”wd0”。

```
            pdev = FindDevice(devname);
```

FindDevice 就是从 gDevice 的链表中查找有没有这个名字的 disk 设备，这个链表是在 DevicesInit 中建立的，一般就只有“wd0”这个设备。代码就不看了。

```
            if (pdev == NULL)
            {
                printf("Couldn't find device [%s]\n", devname);
```

这个错误我们不少见吧，比如你不插硬盘就会见到这条语句。

```
                return NULL;
            }

            fsname = p + 1;
```

```
        p = strchr(fsname, ')');

        if (p == NULL)
        {
            return NULL;
        }
        *p = '\0';
```

现在 fsname 就是字符串”0”

```
        if (isspace(*fsname))
        {
            return NULL;
        }

        part = FindPartitionFromID(pdev->part, atoi(fsname) + 1);
```

FindPartitionFromID()第一个传入参数是那个设备的分区表，后一个参数是第 x 个分区，返回值是那个分区的描述符的指针。代码就不涉及了。

```
        if (part == NULL || part->fs == NULL)
        {
            printf("Don't find partition [%s]\n", devname);
            return NULL;
        }
    } else {
        .。。。
    }
    dfp = (DiskFile *)malloc(sizeof(DiskFile));
    if (dfp == NULL) {
        fprintf(stderr, "Out of space for allocating a DiskFile");
        return NULL;
    }
```

好了，到现在为止，原先的(wd0,0)/boot/boot.cfg 已经被完全拆分了。pdev 现在是存储介质的指针，是个 DeviceDisk 结构体指针，part 是文件所在分区的指针，是个 DiskPartitionTable 结构体指针。这些拆分都被保留在 dfp 的结构体中，如下：

```
    dfp->devname = pdev->device_name;
    dfp->dfs = part->fs;
    dfp->part = part;
```

dfp 是 DiskFile 的结构体指针，描述了一个设备上文件的基本信息。这个结构体最终是 File 结构体的一个 data 参数。

```
    return dfp;
}
```

GetDiskFile 函数返回后回到 disk_open(),执行如下代码：

```
 if (dfp == NULL)
```

```
    {
        return -1;
    }
    _file[fd].posn = 0;
    _file[fd].data = (void *)dfp;
```

好了，现在这个_file[fd]的描述基本都全了，File 结构体的定义如下：

```
typedef struct File {
    short valid;
    unsigned long posn;
    FileSystem *fs;
    void    *data;
} File;
```

其中的 valid 和 fs 是在__try_open()函数中就初始了，现在这个 File 结构虽然填满了，但是够不够呢？如果要读这个文件的内容，到哪儿去读呢？现在通过 data 结构体只能知道那个文件所在硬盘、分区、分区文件系统而已。接下来是文件系统层面的解析。最终我们是要知道那个文件到底在那个分区的哪个地方。由于在龙芯上默认的文件系统都是 ext3 的，ext2 和 ext3 的结构是一样的，下面的文件系统的打开函数就会调用 ext2_read()。

```
    if(dfp->dfs->open)
        return ((dfp->dfs->open)(fd, filename, mode, perms));
    return -1;
```

现在看 diskfs_open 最后一个调用的函数 ext2_open()。文件系统相关的代码在../pmon/fs 下，ext2 的操作函数在../pmon/fs/ext2fs.c。ext2_open()代码如下：

```
int ext2_open(int fd, const char *path, int flags, int mode)
{
    char *str;
    DiskFile* df;

    df = (DiskFile *)_file[fd].data;
    str = path;
```

以前面的打开 boot.cfg 文件为例，这个 path 是字符串”boot/boot.cfg”，(wd0,0)是在 GetDiskFile 中剥离的。

```
    if (*str == '/')
    {
        str++;
    }
    devio_open(fd, df->devname, flags, mode);
```

这个 devio_open 会调用硬件的驱动 wdopen,我们暂时不深入代码，暂时可以理解为看看那个设备是否存在。稍后再解读。反正没判断返回值，不影响执行流。

```
    if (!(ext2_load_linux(fd, str)))
    {
        return fd;
```

```
        }
```
ext2_load_linux()是非常功利的，首先就问是不是可以启动。动机相当不纯呀。马上看看。
```
        return -1;
    }
```
ext2_open()还有两个函数调用没解释，我们先看 ext2_load_linux，再看和硬盘驱动相关的 devio_open()。这里我不准备进到文件系统的代码中去，因为 ext2 文件系统在很多书中都有专门描述，比如《深入理解 Linux 内核》。

ext2_load_linux()函数代码还是在 ext2fs.c 中定义。
```
    static int ext2_load_linux(int fd, const unsigned char *path)
    {
        unsigned int inode;
        char* p;

        if (read_super_block(fd))
            return -1;
```
首先读取超级块。超级块保存了对一个文件系统的许多重要描述。这个读超级块的调用会初始化许多重要的全局变量。
```
        p = path;
        if (path == NULL || *path == '\0')
        {
            p = "/";
        }

        memset(&File_dirent, 0, sizeof(ext2_dirent));
        if ((inode = find_inode(fd, EXT2_ROOT_INO, p, &File_dirent)) == 0)
        {
            return -1;
        }
```
find_inode()函数会返回描述我们要找的那个文件的 inode 编号。并填充 File_dirent 这个结构。
```
        if (File_dirent.file_type == EXT2_FT_DIR)
        {
            printf("%s is directory!\n", p);
            return 0;
        }
```
如果我们要引导的内容是目录，报错。也就是说我们 open 的只能是一个具体的文件，而能是一个目录。
```
        if (ext2_get_inode(fd, inode, File_inode))
        {
            return -1;
```

```
        }
    根据 inode 号，把我们要读的文件的 inode 内容读到 File_inode 中。
        return 0;
    }
```
这里解释一点你可能的疑惑：Fiel_inode 和 File_dirent 都是个全局变量，也就是说打开哪个文件，这些个结构体就表示那个文件的相关内容，所以在 pmon 下不要在一个文件使用的过程中同时使用

如果一切正常，那么返回 0，表示 open 成功。

下面来讲述 devio_open()，这个函数的后两个参数 flags 和 mode 在 pmon 下都没有使用。事实上，pmon 下通过文件系统访问的文件都是只读的。

下面看看 devio_open 到底干了些什么。

```
    int devio_open(int fd, const char *name,int flags,int mode)
    {
        int mj;
        u_int32_t v;
        dev_t dev;
        struct biodev *devstat;
        char strbuf[64], *strp, *p;

        dev = find_device(&name);
```
对于我们一直使用的那个例子，open(wd0,0)/boot/boot.cfg，这里的 name 为“wd0”。这个返回值 dev 是 wd0 的设备号，高 8 位是主设备号，低 8 位是次设备。这个主设备号的来源非常有意思。这个 name 返回后，正常情况下为空。find_device()进一步没有调用函数，代码也很清楚，有兴趣的可以去看看。

```
        if(dev == NULL || *name == '/') {
            ermo = ENOENT;
            return -1;
        }

        mj = dev >> 8;
        devstat = &opendevs[fd];
```
opendevs 是对设备的一些描述，五个成员在下面都被初始化。

```
        devstat->devno = dev;   //设备号
        devstat->offs = 0;
        devstat->base = 0;
        devstat->maptab = 0;
        devstat->end = 0x10000000000;
        if (*name == '@') {
            。。。
        }
        else if (*name != '\0') {
            。。。
```

```
        }
        curproc->p_stat = SRUN;
```
看到这里，不得不说pmon作为一个bios，是个比较委屈的用场，它本来是一个内核，所以有进程的概念，其实作为一个bios，是不必要把事情搞这么复杂的。
```
        ermo = (*devswitch[mj].open)(dev, 0, S_IFCHR, NULL);
```
对于wd0，这里的open就是调用wdopen()。等会再看。
```
        curproc->p_stat = SNOTKERN;

        if(ermo) {
            return -1;
        }

        return(fd);
    }
```
可见devio_open()的函数主要就做两件事，一个是opendevs[fd]的初始化，另一个就是调用主设备对应的open函数，对于硬盘，就是wdopen()。

好，现在就来看wdopen(dev, 0, S_IFCHR,NULL)，代码在../sys/dev/ata/wd.c
```
    int wdopen(dev_t dev, int flag, int fmt, struct proc *p)
    {
        struct wd_softc *wd;
        int unit, part;
        int error;

        unit = WDUNIT(dev);
```
得到次设备号
```
        wd = wdlookup(unit);
```
通过次设备号找到我们要打开的设备，由于是wdlookup，就不用主设备号了。
```
            。。。。。错误处理 。。。。。

        if (wd->sc_dk.dk_openmask != 0) {
            /*
             * If any partition is open, but the disk has been invalidated,
             * disallow further opens.
             */
```
注释说了，这个if成立表示如果这个设备已经被打开了
```
            if ((wd->sc_flags & WDF_LOADED) == 0) {
                error = EIO;
                goto bad3;
            }
        } else {
```
如果这个设备是第一次被打开
```
            if ((wd->sc_flags & WDF_LOADED) == 0) {
```

```
                wd->sc_flags |= WDF_LOADED;
                wd_get_params(wd, AT_WAIT, &wd->sc_params);
```

这个 wd_get_params()会硬盘发出具体的指令，驱动硬盘并获得作出相应的回应。由于比较多细节，且那些结构体的成员较多，有兴趣的可以自行阅读。这里这个命令的含义上获得硬盘的一些常用信息，填充 wd->sc+params 这个结构体。

```
        /* Load the physical device parameters. */
            。。。。。
        part = WDPART(dev);
        /* Insure only one open at a time. */
        switch (fmt) {
        case S_IFCHR:
            wd->sc_dk.dk_copenmask |= (1 << part);
```

标记那个分区被打开了

```
            break;
        case S_IFBLK:
            wd->sc_dk.dk_bopenmask |= (1 << part);
            break;
        }
        wd->sc_dk.dk_openmask =
            wd->sc_dk.dk_copenmask | wd->sc_dk.dk_bopenmask;
        。。。。
        return 0;
    bad:
        。。。。
        return error;
    }
```

好了，wdopen 就算到这里了，由于对于这些细节大部分同志没有了解的必要，就这样吧。当然最重要的原因是：我也不了解，

```
open()->
        try_open()->
                diskfs_open()->
                        GetDiskFile()->
                                ext2_open()->
                                        devio_open()->
                                            find_device()
                                            wdopen()->
                                        ext2_load_linux()->
                                            read_super_block()
                                            find_inode()
                                            ext2_get_inode()
```

# load 内核

龙芯论坛常有人问 pmon 的引导原理，现在我们就看看引导到底是如何实现的。
首先既然引导内核，就首先有个内核所在位置的标识，早期的 pmon 使用 al 这个环境变量表示欲引导的内核，典型的 al 为/dev/fs/ext2@wd0/boot/vmlinux，
现在的 pmon 在保留这个模式的同时，引入了类似 grub 中的 menu.lst 的形式。就是把欲引导的内核和传给内核的参数以一定的格式放在 boot.cfg 的文件中，这个 boot.cfg 文件放在第一硬盘第一分区的根目录下或者 boot 目录下。

先回顾一下执行的流程。在可恶的 init_net 执行完以后，代码会调用一个重要的函数 DevicesInit()用于初始化分区，这个函数我们之前已经分析了。
pmon 的引导相关代码如下(基于 8089 笔记本吧，删除了无关代码)：

```
    switch (get_boot_selection()){
```

get_boot_selection()函数会接收你的按键，根据按下的不同的键进入不同的分支。这个函数中调用了 ioctl()，这个函数非常重要，后面会列出专门一节讲述。

```
        case TAB_KEY:
                ◦◦◦
            break;
        case U_KEY:
                ◦◦◦
            break;
        case NO_KEY:
        case ENTER_KEY:
```

假如你没有任何的按键操作，get_boot_select()就返回 NO_KEY。

```
            if (!load_menu_list())
```

如果没有设置 bootdev 这个环境变量，load_menu_list()按照寻找(wd0,0)/boot.cfg，(wd0,0)/boot/boot.cfg 的顺序寻找启动信息，如果找到合适的可启动的内核，那么就会 run 那个内核，这个函数就一去不复返。这个过程的代码和从 al 读取启动内核位置的本质是一样的，我准备只分析从 al 环境变量读取的情形。

```
                {
                    /* second try autoload env */
                    s = getenv ("al");
```

在没有 boot.cfg 这种启动选择以前，一般都会设置 al 这个环境变量用于选择要启动的内核。同时会设置 karg 这个变量作为 g 的附带参数。

```
                    if (s != 0){
                        autoload (s);
```

好，稍后就以 s 等于/dev/fs/ext2@wd0/boot/vmlinux 为例来介绍这个 autoload。和那个 load_boot_menu 一样，如果不出意外的话，这个函数也是一去不复返的。

```
                    } else {
                      vga_available = 1;
```

```
                printf("[auto load error]you haven't set the kernel path!\n");
            }
        }
        break;
    }
}
```

好，现在看看 autoload()函数。

```
static void autoload(char *s)
{
    char buf[LINESZ] = {0};
    char *pa = NULL;
    char *rd;

    if(s != NULL && strlen(s) != 0) {
        vga_available = 0;
        rd= getenv("rd");
```

这个变量平时很少设置，initrd 在龙芯上也不常用，我只在一次网络安装 debian 系统时使用过这个命令。

```
        if (rd != 0){
            sprintf(buf, "initrd %s", rd);
            do_cmd(buf);
        }

        if (buf[0] == 0) {
            strcpy(buf,"load ");
            strcat(buf,s);
        }
        do_cmd(buf);
```

对于默认的 al 来说，buf 就是字符串 'load /dev/fs/ext2@wd0/boot/vmlinux“。

我们稍后会详细分析这个 do_cmd。这里

```
        if (pa == NULL || pa[0] == '\0')
            pa=getenv("karg");
```

获得 karg 的设置。

```
        strcpy(buf,"g ");
        if(pa != NULL    && strlen(pa) != 0)
            strcat(buf,pa);
        else
            strcat(buf," -S root=/dev/hda1 console=tty");
```

正常的，我们这里的 buf 是“g console=tty root=/dev/hda1 no_auto_cmd”

```
        delay(10000);
        do_cmd (buf);
        vga_available = 1;
```

```
    }
}
```

可见，autoload 实际上就是两个 do_cmd 的函数调用，

do_cmd(“load　/dev/fs/ext2@wd0/boot/vmlinu’)和

do_cmd(“g console=tty root=/dev/hda1 no_auto_cmd’)

简单的说，第一个调用会把那个 vmlinux 文件放到内存上某个位置，g 会执行那个文件。

do_cmd()函数的解析比较繁琐，最终会执行 stat = (CmdTable[i].func) (ac, av)，先看 do_cmd(“load　/dev/fs/ext2@wd0/boot/vmlinu’)

还记得否，init()函数的初始化分为4类，其中的一类就是命令的初始化。CmdTable[i].func 就是执行初始化时注册的那个命令的执行函数。这里就是 cmd_nload 函数。这个函数就是 spawn 的一个包装。spawn 函数初始化了一个“进程”，这些我都不关心，我关心就是 spawn 倒数第二行的的 exit1(sonproc, (*func)(argc, argv) & 0xff),这关系到一个极为关键的地方，spawn 的返回值是什么。在 exit1 之后有句 return(0); /* will not likely happen... */

如果每次都返回 0，那这个返回值就一点价值都没有了，而且正常的你会发现边上的注释。为什么不会执行这个 return 呢？exit1 的作用。

在 exit1 的末尾有一句 longjmp(xx,xx)，可见程序的执行流就被改变了，其实在 spawn 中也有一个 setjmp 来保证 longjmp 返回到哪里，longjmp 和 setjmp 其实是个相反的过程，代码都在../lib/libc/arch/mips/longjmp.S 中。对于这个过程有兴趣的可以细究这个过程，我这里只需要明白，返回值是由执行传给 spawn 的那个命令（第二个参数）返回值决定的就 ok 了。在这里，spawn 的返回值就是执行 nload 命令的返回值。

现在看看这个 nload 到底做了些什么。如果没明白我说什么，就再看一遍代码。nload 这个函数很长，这里只来点关键的。

第一个关键函数，getopt()。比如我们在刷新 pmon 的时候会使用 load -r -f xxx.bin 这样的命令，这个-r 和-f 就是由这个 getopt 解析的。解析之后会设置 flags 变量。

第二个关键函数，open()。

```
    if ((bootfd = open (path, O_RDONLY | O_NONBLOCK)) < 0) {
        perror (path);
        return EXIT_FAILURE;
    }
```

open 函数前面专门讲过，不陌生了吧。

后面我会专门讲解 load pmon 的过程，现在不去管 flash 相关的代码。

这样的话，剩下的代码就不多了，就列在下面：

```
    dl_initialise (offset, flags);
```

这个函数重新初始化了符号表，就是 ls 看到的东西。（试试看，load 前后 ls 有差别没有）。之后清除所有断点。

```
    fprintf (stderr, "Loading file: %s ", path);
    ermo = 0;
    n = 0;
```

```
    if (flags & RFLAG) {
        。。。。
    } else {
        ep = exec (NULL, bootfd, buf, &n, flags);
    }
```

如果没有-r 的选项，执行这个 exec(NULL, bootfd, buf, &n, flags)，记住对于我们 load /dev/fs/ext2@wd0/boot/vmlinux 的例子，这里的 n 和 flags 都是 0。

```
    close (bootfd);
    putc ('\n', stderr);
```

exec 从函数名上看，就是执行这个具体的命令。代码如下：

```
long exec (ExecId id, int fd, char *buf, int *n, int flags)
{
    ExecType *p;
    long ep;
```

可以看出 id 表示这个要 load 的文件的格式，如果为 NULL 表示格式不确定，就调用所有的格式执行函数试试，如果执行顺利，返回 0 表示正确。否则继续尝试下一个类型的文件 loader 函数。这里我们执行的是 elf 格式的内核，所以在执行 elf 的 loader 函数后会返回 ep （entry point），具体的 loader 函数稍后再看。

```
    if (id == NULL) {
        SLIST_FOREACH(p, &ExecTypes, i_next) {
            if (p->flags != EXECFLAGS_NOAUTO) {
                if ((ep = (*p->loader) (fd, buf, n, flags)) != -1) {
                    break;
                }
            }
        }
    } else {
        。。。。。
    }
```

在执行了 elf 的 loader 函数后，exec 函数退出，返回非负数表示 load 成功。我们回到 nload 函数接着看剩下的扫尾工作。

先关闭这个文件 close(bootfd)，接下来是错误处理，不看了，就。剩下就之后下面几句了。我们的 flags 是 0，遇到’!’开头的都执行。

```
    if (!(flags & (FFLAG|YFLAG))) {
        printf ("Entry address is %08x\n", ep);
```

手动 load 过的哥们都见过这句吧。

```
        /* Flush caches if they are enabled */
        if (md_cachestat())
            flush_cache (DCACHE | ICACHE, NULL);
```

刷新一级 cache(数据，指令)

```
        md_setpc(NULL, ep);
        if (!(flags & SFLAG)) {
```

```
                dl_setloadsyms ();
        }
```
至于这两个 set 我都不大理解是做什么用的。
```
    }
```
刚才我们跳过了最重要的 elf 的 loader 函数，现在可以一睹其芳容了。记得我们给的见面礼吗？

ep = (*p->loader) (fd, buf, n, flags))，第一个参数 fd 是文件描述符,buf 是一个大小为 DLREC(550)个字节的缓冲区的首地址，n 是值为 0 的一个局部变量的地址(表示了文件操作的起始字节)，flags 为 0。

Load_elf()函数在../pmon/loaders/exec_elf.c 中，代码较长，我们就跳着讲吧。这个函数和 elf 的格式规范密切相关，如果要阅读这个文件，最好先了解点 elf 的文件格式定义。我推荐滕启明写的《ELF 文件格式分析》，他讲了最基本的概念。在写这个东西的时候，我使用 khexedit 和 hexdump 查看二进制的内容。

首先先读取开头的 DLREC 个字节，判断文件的开头是否是和 elf 文件格式相符。是就打印(elf)，否则出错，返回负数。现在龙芯使用的内核基本都是 64 位的。但是使用的时候一般都使用压缩过的内核，这内核是 32 位的。编译后源码目录的 vmlinux 是 64 位的，通过二进制的对比，他们的 EI_CLASS 位是不同的，我们这里的代码也是通过这个标志位判断当前的内核是 64/32 位的。64 位的内核会调用 load_elf64(fd, buf, n, flags)去执行。这个代码的流程和 load_elf 基本是一样的。中间有个明显的疏忽，在从 load_elf 复制代码后，有个地方的 32 没有改成 64。呵呵。

在确定是 32 位的 elf 文件后，判断程序头部表（programme header table）是否存在。存在并且合理的话就把这个程序头部表读到 phtab 开始的区域中。之后读取节区（section table）表。一个程序头部描述了一个段，一个段中包含了几个节区，大致是这么个关系。在可执行的文件中，我们平时说的代码段，数据段，bss 段都是段，而不是节区。

现在以 8089 逸珑笔记本自带的内核为例看看代码如何执行，我们这里要使用 readelf 这个工具。工欲善其事，必先利其器呀。

```
[root@localhost ~]# readelf -l vmlinux
Elf file type is EXEC (Executable file)
Entry point 0x81000000
There are 2 program headers, starting at offset 52
Program Headers:
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
REGINFO 0x01a000 0x8100a000 0x8100a000 0x00018 0x00018 R   0x4
LOAD   0x010000 0x81000000 0x81000000 0x1d3000 0x1d61d0 RWE 0x10000

 Section to Segment mapping:  节区到段的映射
  Segment Sections...
   00     .reginfo
   01     .text .reginfo .data .sbss .bss
```

可见内核的组织方式是非常简单的，.text .reginfo .data .sbss .bss 这些节区都拼在一块，作为一个可装载的段。
好了，有了这些感性认识，读代码就清楚多了。
Load_elf 接下来的代码就是一个 while 循环，看段类型是 PT_LOAD 就想装载，但代码中也说了，为了兼容旧版本的乱序（地址顺序）链接结果，把代码写的有点不爽了。
最关键的就是 bootread 了。这个直接把要引导的 elf 文件按照它自己的意愿放到内存中，它表达的窗口和是程序头部的 p_vaddr 这里，通过上面的 readelf 看到内核的意愿是放到 0x81000000。
在 pmon 下 load 上面的那个内核的输出为
PMON> load /dev/fs/ext2@wd0/boot/vmlinux
Loading file: /dev/fs/ext2@wd0/boot/vmlinux (elf)
0x81000000/1921024 + 0x811d5000/12752(z) + 91 syms
Entry address is 81000000

其中的第二段以(z)结尾的表示要清零的部分，第三段表示符号表。
具体代码字节看，不说了。
正常情况下返回值是 return (ep->e_entry + dl_offset);，这里的 dl_offset 就是 0，不信打出来看看。
在这个函数返回 ep 后，我们回到 nload 那个函数看看。

```
    if (!(flags & (FFLAG|YFLAG))) {
        printf ("Entry address is %08x\n", ep);
        /* Flush caches if they are enabled */
        if (md_cachestat())
            flush_cache (DCACHE | ICACHE, NULL);
        md_setpc(NULL, ep);
        if (!(flags & SFLAG)) {
            dl_setloadsyms ();
        }
    }
```

当时我们不知道几个 set 是干什么的，现在情况不一样了。setpc 就是在条件允许的情况下，我们要调到的地址，setloadsysms，就是改变 ls 看到的内容。
好了，nload 执行结束，返回成功标记，这样一个 load 过程就算 ok 了。也就是 do_cmd(“load /dev/fs/ext2@wd0/boot/vmlinu’)过程结束了。
load 是好了，还有个 g 呢，回到 autoload 那个函数。假如我们的 karg 是”console=tty root=/dev/hda1 no_auto_cmd’，下面就瞅瞅 do_cmd(“g console=tty root=/dev/hda1 no_auto_cmd’)干什么，好，瞅瞅就瞅瞅。

g 的命令就是执行 cmd_go，在../pmon/cmd/cmd_go.c 中。
由于要引导的内核已经在内存里了，g 要做的就是让 cpu 到那里执行，这需要先处理他撒手而去的一些事宜。

开始就是一些参数检查，把 clientcmd 设置为字符串“g console=tty root =/dev/hda1 no_auto_cmd PMON_VER=xxxx  EC_VER=xxx”。

```
if (!sflag) {
    md_adjstack(NULL, tgt_clienttos ());
}
```

设置新的 sp，位置在内存的高端最后 64 字节

```
clientac = argvize (clientav, clientcmd);
```

解析这个 clientcmd，每项都在 clienttav 这个数组中。这里我们有 6 项，clientac 为 6。

```
initstack (clientac, clientav, 1);
```

这个 initstack 是对栈的一些操作，还是非常重要的，在这个函数中还把所有的 pmon 的环境变量和 g 的参数等放在栈里，用于内核使用。

```
md_registers(1,NULL);
```

打印当前的所有寄存器内容。

```
closelst(2);            /* Init client terminal state */
md_setsr(NULL, initial_sr);
```

设置状态寄存器。

```
tgt_enable (tgt_getmachtype ()); /* set up i/u hardware */
```

设置一些使能。

```
usb_ohci_stop();
rtl8139_stop();
```

关闭一些 usb 设备和网卡

```
if (setjmp (go_return_jump) == 0) {
```

先设置等会我们要执行的寄存器状态

```
    goclient ();
```

goclient()最重要的就是最后调用_go()，把寄存器给填充为我们 setjmp 时的值，关键是 epc 这个寄存器，执行 eret 返回，就返回到 eret 所指的地址去了。关键的在这：

```
LOAD    v0, PC * RSIZE(k0)
LOAD    v1, CAUSE * RSIZE(k0)
MTC0    v0, COP_0_EXC_PC
```

这个 PC 就是我们 load 命令最后 set 的前面那个 md_setpc()存的。这个 goclient()中的_go()之后，pmon 就把控制权交给内核了，阿弥陀佛，再见。

```
}
```

正常的话，下面的都不执行的

```
console_state(1);
printf("\nclient return: %d\n", retvalue);
return 0;
}
```

# Termio

pmon 的代码有很多会涉及到 termio 的函数，这里先介绍一些。
termio 相关的函数代码都在../pmon/fs/termio.c。首先看 termio 在 init()函数初始化时执行的代码。

```
static void init_fs()
{
    devinit ();
```

这个函数非常重要，会初始化描述串口和显卡设备的一些结构体，稍后详细解释。

```
    filefs_init(&termfs);
```

往文件系统链表插入 termio 这个文件系统，以备使用。

```
    _file[0].valid = 1;
    _file[0].fs = &termfs;
    _file[1].valid = 1;
    _file[1].fs = &termfs;
    _file[2].valid = 1;
    _file[2].fs = &termfs;
    _file[3].valid = 1;
    _file[3].fs = &termfs;
    _file[4].valid = 1;
    _file[4].fs = &termfs;
```

不同的下标代表了不同的设备，具体代表的设备如下面所示。

```
    term_open(0, "/dev/tty0", 0, 0); /* stdin */
    term_open(1, "/dev/tty0", 0, 0); /* stdout */
    term_open(2, "/dev/tty0", 0, 0); /* stderr */
    term_open(3, "/dev/tty1", 0, 0); /* kbdin */
    term_open(4, "/dev/tty1", 0, 0); /* vgaout */
}
```

下面简要的介绍 devinit 和 term_open。先看 devinit，定义如下：

```
Int devinit (void)
{
    int i, brate;
    ConfigEntry *q;
    DevEntry *p;
    char dname[10];
    char *s;

    strcpy(dname, "tty_baud");
    for (i = 0; ConfigTable[i].devinfo && i < DEV_MAX; i++) {
```

ConfigTable 数组在 tgt_machdep.c 中定义，对于逸珑 8089，实际上就是：

```
    ConfigEntry ConfigTable[] =
    {
        { (char *)COMMON_COM_BASE_ADDR, 0, ns16550, 256,
                                                    CONS_BAUD, NS16550HZ },
        { (char *)1, 0, fbterm, 256, CONS_BAUD, NS16550HZ },
        { 0 }
    };
```

很明显，第一个表示串口，第二个是显卡，第三个表示结束。

```
        q = &ConfigTable[i];
        p = &DevTable[i];
```

下面要使用 ConfigTable 的内容填充 DevTable 的结构体。

```
        p->txoff = 0;
        p->qsize = q->rxqsize;
        p->sio = q->devinfo;
        p->chan = q->chan;
        p->rxoff = 0;
        p->handler = q->handler;
```

handler 函数比较重要，会在多个场合被调用到。

```
        p->tfunc = 0;
        p->freq = q->freq;
        p->nopen = 0;

        if (p->chan == 0)
            (*p->handler) (OP_INIT, p, NULL, q->rxqsize);

        p->rxq = Qcreate (p->qsize);
        p->txq = Qcreate (p->qsize);
```

这里创建接收和发送队列的缓冲区，大小默认都是 256 个字节。

```
        if (p->rxq == 0 || p->txq == 0)
            return (-1);

        /*
         * program requested baud rate, but fall back to default
         * if there is a problem
         */
/* XXX don't work. env init is not called yet. has to be solved */
        dname[3] = (i < 10) ? i + '0' : i - 10 + 'a';
        if ((s = getenv(dname)) == 0 || (brate = getbaudrate(s)) == 0)
            brate = q->brate;

        if (brate != q->brate) {
            if ((*p->handler)(OP_BAUD, p, NULL, brate)) {
```

```
                brate = q->brate;
                (void)(*p->handler)(OP_BAUD, p, NULL, brate);
            }
        }

        p->t.c_ispeed = brate;
        p->t.c_ospeed = brate;
    }
    return (0);
}
```

这个函数执行完以后，DevTable 的前两个元素就表示串口和显卡了。下面看 term_open，这个函数在 termio 的 init_fs 函数中被连续 5 次调用，分别用于初始化前 5 个文件结构，我们下面以 term_open(4, "/dev/tty1", 0, 0); /* vgaout */为例介绍，函数定义如下：

```
Int term_open(int fd, const char *fname, int mode, int perms)
{
    int c, dev;
    char *dname;
    struct TermDev *devp;
    DevEntry *p;

    dname = (char *)fname;
    if (strncmp (dname, "/dev/", 5) == 0)
        dname += 5;
```

我们传入的是/dev/tty1 或者/dev/tty0，开头都是一样的。

```
    if (strlen (dname) == 4 && strncmp (dname, "tty", 3) == 0) {
        c = dname[3];
        if (c >= 'a' && c <= 'z')
            dev = c - 'a' + 10;
        else if (c >= 'A' && c <= 'Z')
            dev = c - 'A' + 10;
        else if (c >= '0' && c <= '9')
            dev = c - '0';
```

我们的例子中 dev 为 1。

```
        if (dev >= DEV_MAX || DevTable[dev].rxq == 0)
            return  -1;

        devp = (struct TermDev *)malloc(sizeof(struct TermDev));
        if (devp == NULL) {
            errno = ENOMEM;
            return -1;
        }
```

每个文件对应一个 TermDev 结构。前面 5 个文件是特定的。

```
        devp->dev = dev;
        _file[fd].data = (void *)devp;
        p = &DevTable[dev];
        (*p->handler) (OP_OPEN, p, NULL, 0);
```

对于显卡，他的 OP_OPEN 实际上什么也不做。

```
        if (p->nopen++ == 0)
            term_ioctl (fd, SETSANE);
```

如果是第一次打开，那么 reset 一下这个设备的相关设置。

```
    }
    else {
        return -1;
    }

    return fd;
}
```

# printf 和 write

我们最熟悉的函数就是用 printf 在屏幕上打印字符串了。现在就看看吧。
printf 的解析是非常繁琐的。我们就用最简单的情况 printf("hello ,pmon")分析。
printf 定义在 lib/libc/printf.c 中。代码如下：

```
Int printf (const char * fmt, ...)
{
    int                 len;
    va_list     ap;

    va_start(ap, fmt);
    len = vfprintf (stdout, fmt, ap);
    va_end(ap);
```

如果各位大侠看过内核的源码中的可变参数部分，对这几个 va_start 和 va_end 一定不会陌生。我基本上也忘的差不多了，挺不好讲的，不说了，好吧。
我们的例子中，可变参数的参数个数为 1，printf("hello ,pmon")唯一的参数就是这个字符串的首地址。这个 ap 就不管了。我们要打印的就是 fmt 开始的那个字符串的内容。
stdout 的定义为#define stdout    (vga_available?(&_iob[4]):(&_iob[1]))
在显卡初始化以后，vga_available 为 1，stdout 就是_iob[4]
_iob[]的定义如下：

```
FILE    _iob[OPEN_MAX] =
{
```

```
    {0, 1},    // stdin
    {1, 1},    // stdout
    {2, 1},    // stderr
    {3, 1},    // kbdin
    {4, 1},    // vgaout
};
```

第一个参数表示 fd，第二个参数都是 1，表示文件有效。这些个 fd 的 fs 参数在../pmon/fs/termio.c 中定义。均定义为 termfs，马上会用到。

```
    return (len);
}
```

Vfprint 代码如下，代码中的 fp 就是传入的 stdout：

```
Int vfprintf (FILE *fp, const char *fmt, va_list ap)
{
    int    n=0;
    char buf[1024];

    n = vsprintf (buf, fmt, ap);
```

vsprintf()会解析这个 printf 传入的参数，解析后的结果放在以 buf 为首地址的缓冲区中。n 表示解析后的字符串长度。

```
    fputs (buf, fp);
```

fputs 就一句话，write (fp->fd, p, strlen (p));由于 fp->fd 就是 4，这个文件代表 vgaout。

```
    return (n);
}
```

以上的代码实际上就是调用 write (4, p, strlen (p))，对于我们前面的 printf(“hello , pmon”)，这里的 p 就是这个字符串的首地址。

我们曾经分析过 open 的操作，现在就以这个 printf 的调用为例分析 write 操作吧。代码在../lib/libc/write.c 中，如下：

```
int write (int fd, const void *buf, size_t n)
{
    if ((fd < OPEN_MAX) && _file[fd].valid ) {
        if (_file[fd].fs->write)
            return (((_file[fd]).fs->write) (fd, buf, n));
```

如果这个文件（设备）所属的文件系统的 write 函数存在，就执行之。fd 为 4 代表的 vga 的 fs 为 termfs，其 write 函数为 term_write()。

```
        else
            return (-1);
    }
    else {
        return (-1);
    }
}
```

传给 term_write 的参数实际上是 term_write(4,”hello, pmon’，长度)
term_write()代码如下：

```
int term_write (int fd, const void *buf, size_t nchar)
{
    DevEntry        *p;
    struct TermDev *devp;
    char *buf2 = (char *)buf;
    int i, n;

    devp = (struct TermDev *)_file[fd].data;
```

这些 fd 的 data 在 termio.c 中的 init_fs 中的 5 个 term_open 调用的过程中初始化。其中的 dev 成员标志了这个到底是什么设备，0 就是串口，1 就是显卡。也就是说被 write 的设备是显卡。

```
    p = &DevTable[devp->dev];
    n = nchar;

    while (n > 0) {
        /* << LOCK >> */
        while(!tgt_smplock());
```

tgt_smplock()这些用于互斥的操作在 pmon 下实际上是没有什么意义的。并没有对什么资源的申请并占有。

```
        i = Qspace (p->txq);
```

判断发送队列还有多少空间。下面就一个一个发送。

```
        while (i > 2 && n > 0) {
            if ((p->t.c_oflag & ONLCR) && *buf2 == '\n') {
                Qput(p->txq, '\r');
```

查看 termio 的规定，如果设置 oflag 的 OLCUC 位，在发送换行符（'\n'）前先发送回车符（'\r'）。Qput 把要发送的字符保存在缓冲区中。

```
                i--;
            }
            Qput(p->txq, *buf2++);
```

发送传入的那个字符。

```
            n--;
            i--;
        }
```

如果顺利的话，现在要输出的字符都放到发送队列中去了。

```
        tgt_smpunlock();
        /* << UNLOCK >> */

        while (Qused(p->txq)) {
            scandevs();
```

如果有字符要打印，现在要调用 scandevs()。看函数名，就是挨个问，有没有要打印东西的，有就打印了。

```
            }
        }
        return (nchar);
}
```

scandev 这个函数在多个地方被调用，是非常关键的。

这个函数的流程可以分为三个部分，分别是检查输入缓冲区，输出缓冲区，按键。

我们暂时只看后面检查输出的代码，如下：

```
            /* Write queue */
            n = Qused (p->txq);
```

看看有没有要输出的，返回的是要输出的个数。

```
            while (n > 0 && !p->txoff &&
              (*p->handler)(OP_TXRDY, p, NULL, NULL)) {
```

对于逸珑 8089 的显卡，其 handler 函数为 fbterm()，在../pmon/dev/vgacon.c 中，其 OP_TXRDY 恒返回 1。表示随时都准备处理。

```
                char c = Qget(p->txq);
```

取出要打印的字符。

```
                (*p->handler) (OP_TX, p, NULL, c);
```

下面看看 OP_TX 到底干了些什么。

```
        case OP_TX:
            if (vga_available)
                video_putc(data & 0xff);
            break;
```

如果要显示的话，就调用 video_putc()在屏幕上打印那个东西。这个函数没有什么东西了，现在就只看这个函数了吧。

```
                n--;
            }
```

Video_putc 函数代码如下：

```
void    video_putc(const char c)
{
        switch (c) {
        case 13:                /* ignore */
            break;
        case '\n':              /* next line */
            console_newline();
            break;
        case 9:         /* tab 8 */
            CURSOR_OFF console_col |= 0x0008;
            console_col &= ~0x0007;
            if (console_col >= CONSOLE_COLS)
                console_newline();
```

```
            break;
        case 8:         /* backspace */
            console_back();
            break;
        default:            /* draw the char */
```

其它的都不管了，我们只看最常见的情形，显示一个字符。

```
            video_putchar(console_col * VIDEO_FONT_WIDTH,
                          console_row * VIDEO_FONT_HEIGHT, c);
```

在BIOS阶段，显示都是用的字符模式，每个字符的宽度为8个像素，每个字符的高度为16个像素。video_putchar的前面两个参数就是要显示的字符所在位置的左上角的像素点的列位置，行位置。第三个参数是要显示的字符。

```
            console_col++;
            /* check for newline */
            if (console_col >= CONSOLE_COLS)
                console_newline();
```

检测是否到头了。

```
        }
    CURSOR_SET
}
```

就看video_putchar这个函数了，定义如下：

```
void video_putchar(int xx, int yy, unsigned char c)
{
    video_drawchars(xx, yy + VIDEO_LOGO_HEIGHT, &c, 1);
}
```

VIDEO_LOGO_HEIGHT在fb/cfb_console.c中定义为0。这里的xx为水平方向的像素位置，yy为垂直方向的像素位置。c是要显示的符号，1表示显示1次。函数定义如下：

```
static void video_drawchars(int xx, int yy, unsigned char *s, int count)
{
    unsigned char *cdat, *dest, *dest0;
    int rows, offset, c;

    offset = yy * VIDEO_LINE_LEN + xx * VIDEO_PIXEL_SIZE;
```

这里的VIDEO_PIXEL_SIZE为2，表示每个像素占2个字节，VIDEO_LINE_LEN为1024*2，表示一行像素要占用的字节数。我怎么知道这个值是多少呢？printf不行，会递归成死循环的。

小技巧，加如下两句。进入pmon命令行后用m命令看 m -d 81000000即可。

```
    *(long *)(0x81000000) = VIDEO_LINE_LEN;
    *(long *)(0x81000004) = VIDEO_PIXEL_SIZE;
    dest0 = video_fb_address + offset;
```

通过如上的方法，读出这里的video_fb_address值为0xb4000000，可见framebuffer的首地址为0xb4000000。每个屏幕上的点都由framebuffer上的两个字节控制。

0xb4000000 开始的两个字节对应屏幕左上角的像素点。

```
switch (VIDEO_DATA_FORMAT) {
case GDF__8BIT_INDEX:
case GDF__8BIT_332RGB:
```

接下来是一个 switch，VIDEO_DATA_FORMAT 定义如下：

```
#define VIDEO_DATA_FORMAT     (pGD->gdfIndex)
```

pGD 在 fb_init 中初始化，小本上这个是 2，表示色深为 16 位色。同理，同理如果这个为 3，表示 24 位色。代码中设定了如果为 16 位色，就使用 565 的 RGB 编码。这个值的 switch 的代码为：

```
case GDF_16BIT_565RGB:
    while (count--) {
        c = *s;
        cdat = video_fontdata + c * VIDEO_FONT_HEIGHT;
```

我们可以算出来了，一个字符的像素为 16x8(所以字符都挺水蛇腰的)，每个像素占两个字节。cdat 表示我们要显示的点阵的首地址。我们假设这里要显示'A'，到底如何才能看到我们看到的符号呢？

vIdeo_fontdata 是一个点阵数组，字符 A 的点阵如下，特地用红色标记了，差不多吧，去看看 pmon 命令行下的的 A，是不是一样的。就是一样的。

```
0x00,           /* 00000000 */
0x00,           /* 00000000 */
0x10,           /* 00010000 */
0x38,           /* 00111000 */
0x6c,           /* 01101100 */
0xc6,           /* 11000110 */
0xc6,           /* 11000110 */
0xFe,           /* 11111110 */
0xc6,           /* 11000110 */
0xc6,           /* 11000110 */
0xc6,           /* 11000110 */
0xc6,           /* 11000110 */
0x00,           /* 00000000 */
0x00,           /* 00000000 */
0x00,           /* 00000000 */
0x00,           /* 00000000 */
```

现在都明白了，video_fontdata 就是字符的点阵数组了，如何显示呢？

这个字符 A 的点阵为 16 个字节，但实际上在 framebuffer 上我们一个字符占用 256 个字节，或者说点阵中一行是一个字节，但在 framebuffer 上是 16 个字节，这里就有一个索引，每两位索引四个字节。等会就看到了。 1行16个字节

```
or (rows = VIDEO_FONT_HEIGHT, dest = dest0;
        rows—; dest += VIDEO_LINE_LEN) {
```

看这个 for 的循环方式，可以看出是填充一行，一共 16 次，每行差 2048 个字节，这个前面都说过的。

```
                unsigned char bits = *cdat++;
```

bits 现在表示了点阵的第一个字节的内容，代码中的 SHORTSWAP32 没什么用，去掉了，看得就清楚多了。

```
            ((unsigned int *)dest)[0] =
                    (video_font_draw_table16[bits >> 6] & eorx) ^ bgx;
            ((unsigned int *)dest)[1] =
                    (video_font_draw_table16[bits >> 4 & 3] & eorx) ^ bgx;
            ((unsigned int *)dest)[2] =
                    (video_font_draw_table16[bits >> 2 & 3] & eorx) ^ bgx);
            ((unsigned int *)dest)[3] =
                    (video_font_draw_table16[bits & 3] & eorx) ^ bgx;
```

前面刚说了，每两位一个索引，看上面的代码就很明白了，>>6，>>4，>>2，不就是剥离出那两位吗!索引的数组是 video_font_draw_table16，定义如下：

```
static const int video_font_draw_table16[] = {
    0x00000000, 0xffff0000, 0x0000ffff, 0xffffffff
};
```

可见索引对应关系如下：

| | | |
|---|---|---|
| 00 | ———— | 0x00000000 |
| 01 | ———— | 0xFFFF0000 |
| 10 | ———— | 0x0000FFFF |
| 11 | ———— | 0xFFFFFFFF |

这里要注意高低地址的关系，01 表示后面那个像素要点亮，但是后面的像素是在高字节的，所以 01 对应的是 0xFFFF0000。

代码中的 eorx 为 0xa514a514，bgx 为 0。0xa514 这个数有明显特征，就是和它与后，RGB 都只保留了两位数。这就像是对 565 的格式再进行采样，也许很性能相关吧，不去关注了。

```
                }
            dest0 += VIDEO_FONT_WIDTH * VIDEO_PIXEL_SIZE;
            s++;
        }
        break;
```

好了，再细节的东西列位自己看吧。printf 就到这里了。

# 键盘和键盘事件的响应

前面讲了 printf 的输出，下面就介绍它的反方面，那就是键盘输入的响应。

在 dbginit 的过程中，会执行 tgt_devconfig()，对于笔记本，这个函数会调用 kbd_initialize()初始化键盘控制器。8042 键盘控制器有两个端口，状态和控制寄

存器为 64h 端口，60h 为数据寄存器端口。在开始看这个初始化代码前，我们先看看接受输入的函数 kbd_read_data()。

```
static int kbd_read_data(void)
{
    int retval = KBD_NO_DATA; -1
    unsigned char status;

    status = kbd_read_status();
```

8042 键盘控制器文档对寄存器的说明非常清楚。状态寄存器描述如下：

```
|7|6|5|4|3|2|1|0|   8042 Status Register
| | | | | | | +---- output register (60h) has data for system
| | | | | | +----- input register (60h/64h) has data for 8042
| | | | | +------ system flag (set to 0 after power on reset)
| | | | +------- data in input register is command (1) or data (0)
| | | +-------- 1=keyboard enabled, 0=keyboard disabled (via switch)
| | +--------- 1=transmit timeout (data transmit not complete)
| +---------- 1=receive timeout (data transmit not complete)
+----------- 1=even parity rec'd, 0=odd parity rec'd (should be odd)
```

```
    if (status & KBD_STAT_OBF) {
```

如果按下了键但系统没有读取

```
        unsigned char data = kbd_read_input();
```

那么从数据端口读取那个按键的扫描码。

```
        retval = data;
        if (status & (KBD_STAT_GTO | KBD_STAT_PERR))
            retval = KBD_BAD_DATA;
    }
    return retval;
}
```

kbd_initialize 代码较长，这里只讲主干。

```
int kbd_initialize(void)
{
    int status;
    int count;

    status = kb3310_test();
```

这个 kb3310_test 会读取 xram 的第一（0）个字节，如果返回那个值为 0xff，表明出错。

```
    /* Flush the buffer */
    kbd_clear_input();
```

这个函数会检查当前缓冲去中是否有内容，如果有，则抛弃掉。判断是否有内容是用 kbd_read_data()，如果返回非 KBD_NO_DATA，就继续读。

```
kbd_write_command_w(KBD_CCMD_SELF_TEST);
```

向 8042 的控制寄存器发送命令。在 8042 的文档中对命令的解释也很清楚，有兴趣的可以自行阅读。KBD_CCMD_SELF_TEST 是 0xAA，AA 命令的解释为：

AA　　Self Test: diagnostic result placed at port 60h, 55h=OK

```
if (kbd_wait_for_input() != 0x55) {
    printf("Self test cmd failed,ignored!\n");
}
```

很符合文档的规范，返回 0x55 表示正常。下面判断时钟和数据线是否有效

```
kbd_write_command_w(KBD_CCMD_KBD_TEST);
if (kbd_wait_for_input() != 0x00) {
    printf("KBD_TEST cmd failed,ignored!\n");
}
```

下面是正式的键盘接口使能。

```
kbd_write_command_w(KBD_CCMD_KBD_ENABLE);

count = 0;
do {
    kbd_write_output_w(KBD_CMD_RESET);
    status = kbd_wait_for_input();
    if (status == KBD_REPLY_ACK)
        break;
    if (status != KBD_REPLY_RESEND) {
        printf("reset failed\n");
        if (++count > 1)
            break;
    }
} while (1);

if (kbd_wait_for_input() != KBD_REPLY_POR) {
    printf("NO POR, ignored!\n");
}

count = 0;
 do {
     kbd_write_output_w(KBD_CMD_DISABLE);
```

发送这个命令后，键盘会回到上电后的默认状态，并分会一个 ACK。

```
        status = kbd_wait_for_input();
        if (status == KBD_REPLY_ACK)
            break;
        if (status != KBD_REPLY_RESEND) {
            printf("disable failed\n");
            if (++count > 1)
```

```
                    break;
            }
    } while (1);

    kbd_write_command_w(KBD_CCMD_WRITE_MODE);
```
告诉键盘控制器，我的下一次操作是发命令了。
```
    kbd_write_output_w(KBD_MODE_KBD_INT
                    | KBD_MODE_SYS
                    | KBD_MODE_DISABLE_MOUSE | KBD_MODE_KCC);
```
使能中断。下面设置键盘的扫描码
```
        if (!(kbd_write_command_w_and_wait(KBD_CCMD_READ_MODE) &
KBD_MODE_KCC)) {
        。。。IBM powerpc 的特殊处理部分。。。
    }
```
下面使能键盘的输入扫描。前面的命令都不是扫描输入得到的。
```
    if (kbd_write_output_w_and_wait(KBD_CMD_ENABLE) !=
                                                    KBD_REPLY_ACK) {
        return 5;
    }
```
设置扫描周期和延时，下一个命令传送具体的参数。
```
    if (kbd_write_output_w_and_wait(KBD_CMD_SET_RATE) !=
                                                    KBD_REPLY_ACK) {
        return 6;
    }
```
下面的 0 是上面那个命令的参数，表示没有延时，每秒扫描 30 次。
```
    if (kbd_write_output_w_and_wait(0x00) != KBD_REPLY_ACK) {
        return 7;
    }
    return 0;
}
```
好，kbd_initialize 到此为止。usb 键盘和笔记本自带的键盘的处理不同。我这只关注自带键盘（因为这个简单*_*）。

由于在 pmon 中没有中断的功能，所以必须有个时机让 cpu 知道有按键的动作发生。这个时机就是调用 scandevs 函数的时候。

scandevs 这个函数函数可分为三部分，检查输入缓冲区(软件层面)，检查输出缓冲区，检查有没有按键。前面我们看的是第二部分的代码，现在看第三部分。我关心的就一句话，kbd_poll()。
```
void kbd_poll()
{
    if (esc_seq) {
            。。。
    } else {
```

```
        while (kbd_read_status() & KBD_STAT_OBF)
```
这里检查键盘控制器的状态，看是否有未处理的按键。如果有，那么
```
            handle_kbd_event();
```
这个函数会调用 handle_kbd_event，代码如下：
```
static unsigned char handle_kbd_event(void)
{
    unsigned char status = kbd_read_status();
    unsigned int work = 10000;
```
status 表示了当前 8042 键盘控制器的状态，work 表示尝试的次数。避免刚处理了按键很快又有键按下，导致总在这个循环中的问题。其实概率非常非常小啦。
```
    while ((--work > 0) && (status & KBD_STAT_OBF)) {
        unsigned char scancode;
        scancode = kbd_read_input();
```
从 8042 读出来的数据都是第一套的扫描码，前面键盘初始化的时候设置的。
```
        if (!(status & (KBD_STAT_GTO | KBD_STAT_PERR)))
        {
            if (status & KBD_STAT_MOUSE_OBF);
            else
                handle_keyboard_event(scancode);
```
处理键盘的输入，参数是扫描码的值。
```
        }
        status = kbd_read_status();
```
判断现在还有没有未处理的按键，如果有则还在这个循环中。在第一套扫描码中 ，大部分按键的码都是一个字节，部分特殊功能键是 2 或 3 个键码。
```
    }

    return status;
}
```
handle_keyboard_event 的定义如下
```
static inline void handle_keyboard_event(unsigned char scancode)
{
    if (do_acknowledge(scancode))
        handle_scancode(scancode, !(scancode & 0x80));
}
```
跳过处理重复发送键码（比如你一直按下一个键时）的处理，实际就是调用 handle_scancode(scancode, !(scancode & 0x80));这个 scancode 为什么要与上 0x80 呢？因为在第一套扫描码的定义中，按下键和释放这个键的扫描码差 0x80,而且释放键的扫描码都是大于 0x80 的。故可通过此判断是否为释放键。
handle_scancode 代码如下：
```
void handle_scancode(unsigned char scancode, int down)
{
    unsigned char keycode;
```

```
    char up_flag = down ? 0 : 0200;
```
Up_flag 为 0，表示是按下键，否则为释放键。
```
    if (!kbd_translate(scancode, &keycode))
        goto out;
```
对于软件系统来说，处理的不是扫描码，而是键码，所以有个 kbd_translate 函数，这个有兴趣的自己看。
```
    if (1) {
        u_short keysym;
        u_char type;

        ushort *key_map = key_maps[shift_state];
```
这里有有一层软件的抽象，前面的转换成 keywork 是标准的，对于所有的键盘都一样，但我们知道有美式键盘，荷兰语键盘，西班牙键盘等，也就是按了同一个键表达的内容不一样，这个功能是通过 kaymap 的映射实现的。
```
        if (key_map != NULL) {
            keysym = key_map[keycode];
            type = KTYP(keysym);
```
这个 type 和 key_map 相关联，不同类型的键 type 不同，比如字符键，ctrl 键，backspace 键等类型就明显不同。
```
            if (type >= 0xf0) {
                type -= 0xf0;
                if (type == KT_LETTER) {
                    type = KT_LATIN;
```
表示输入是一个字符。
```
                }
                if (*key_handler[type]) {
                    (*key_handler[type]) (keysym & 0xff,
                                          up_flag);
```
根据 type 调用不同的按键处理函数，对于字符键，调用的就是 key_handle[0]，处理很简单，如果是释放键，直接返回，否则给 kbd_code 这个全局变量赋值。
```
                }
            }
        }
    }
    out:
}
```
函数基本上是原路返回，并没有什么动作了。返回到 scandevs 中。

这里有个问题我不理解，返回到 scandevs 后，只有非零 kbd_code 表示按下了一个键并且这个按键没有被处理，也就是执行了 scandevs 后，我的输入不能马上被放到输入缓冲区中，而是在下次 scandevs 时被放到输入缓冲区中。也就是说如果没有下次对 scandevs 的调用，这次的输入就没有作用。因为代码中的确没有在 kbd_poll 后有任何的对输入缓冲区的操作呀！但是，我们明显知道，在 pmon 命

令行下，我们按下一个字符键，就显示一个字符，可见是一旦输入就处理了呀，并没有出现在下次按键后才显示前一个键的状况呀。怎么回事？怎么回事？为什么？为什么？

## 是这样吗？

剪不断，理还乱，别有一番滋味在心头。
的确，世界上有很多事情是我难以精确解释的，不过这个问题不是。
首先做一个实验。
在 pmon 中，最先响应的输入是 get_boot_selection。这个文件的开头部分如下：

```
int get_boot_selection(void)
{
    int flag = 1;
    int c;
    unsigned int cnt;
    unsigned int dly;
    struct termio sav;
    dly = 128;

    ioctl(STDIN, CBREAK, &sav);
    do {
        delay(10000);
        ioctl (STDIN, FIONREAD, &cnt);
```

也许一看就明白，这个 ioctl 调用后返回的 cnt 的值表示了输入的键值。因为如果 cnt 非 0 的话，就认为按下键，就 getchar()。

```
        if(cnt == 0) {
            flag = NO_KEY;
            continue;
        }
        c = getchar();
```

好，get_boot_selection 就到这里。看 ioctl (STDIN, FIONREAD, &cnt)的。如下：

```
        case FIONREAD:
            scandevs ();
            *(int *)argp = Qused (p->rxq);
            break;
```

scandevs 我们都很熟悉了（或者更陌生了）。老生我再谈一下吧：
这个函数分三部分：

1. 输入缓冲区的检查。
2. 输出缓冲区的检查
3. 有没有待处理的硬件（键盘按键，网络包处理等）

我一开始是这样理解这个 ioctl 的，我们在开机后按下了 del 键，但是一直没有被处理，到这个 ioctl (STDIN, FIONREAD, &cnt)执行的时候，会执行第三部分的 kbd_poll ，检查键盘控制器，看有未处理的按键，读出来，scancode 转 keycode，

设置 kbd_code，再往输入缓冲区中写。之后这个 scandevs 执行完毕，马上接着执行*(int *)argp = Qused (p->rxq)，啊，输入缓冲区不空。argp 记录下这个输入缓冲区的内容数量。
这个想法很合理，但代码不是这么写的。问题在于在设置了 kbd_code 后，scandevs 没把键值送到缓冲区就拍屁股走了。所以实际上在执行 Qused (p->rxq)时，我这次的输入并没有加到这个输入缓冲区中（receive queue）。那么这个返回的东西岂不就是 0 吗？好，问题说好了。实验来了。
在代码中添加三句话(红色字体显示)

```
case FIONREAD:
    temp1 = Qused (p->rxq);
     scandevs ();
    temp2 = Qused (p->rxq);
    printf("temp1    %d  :    temp2    %d \n", temp1,    temp2);
     *(int *)argp = Qused (p->rxq);
     break;
```

加入了这个修改的 pmon 执行的效果如何，开机后长按 del 键，结果出来了。
temp1 和 temp2 都是 3。也就是说 scandevs 前后输入缓冲区的内容并没有变化。为什么呢？当然为什么是 3 这个问题更难回答。
其实在启动的时候（包括 kbd 初始化后）我们执行了许多的 printf 函数，有的打到串口，有的打到屏幕上。前面的章节中有个细节，在 printf 执行的过程中会调用 scandevs，不信回去看看，也就是说每个 printf 函数都可能会调用 kbd_poll，从而接收了键盘控制器的输入。所以我有个猜想，如果在 kbd 初始化完成后到 get_boot_selection 之间没有调用 printf(或者其它会调用 scandevs 的函数)，我们在第一次 get_boot_selection 调用 ioctl (STDIN, FIONREAD, &cnt);的结果将是 cnt 为 0，当然 get_boot_selection 是个循环，第二次就会在 scandevs 的输入缓冲区的检查中检查到输入了。cnt 就不为 0 了。
如果我上面的分析没有问题的话，就是当前的输入是下一个 scandevs 才能收到，在 pmon 命令行下即输即显又如何解释呢？说不过去呀？
看代码，不行也行。
Main->getcmd->get_line->getchar->getc->fgetc->read->term_read。
上面的顺序就是 pmon 命令行下接受输入的函数调用路径。这个路径写到 term_read 是因为我们看到了一个可以说明问题的代码了。代码如下：

```
for (i = 0; i < n;) {
        scandevs ();
```

这里的 n 是 1，表示我们要 read 一个字节，这个 for 循环有个特殊的地方发现没有，没有 i++，什么时候 i++呢，在这个函数的末尾。也就是说如果没有读到就一直 scandevs。直到输从入缓冲区中读到一个才 i++，都到这里了，剩下的自己看吧。所以在 pmon 命令行下即输即显是通过反复调用 scandevs，直到得到一个字节的方式实现的。要知道回显是如何实现的可以看 get_line 代码，很明显的。

## 这个流程合理吗？

scandevs 的流程是否合理呢？那三部分的顺序能否掉换呢？也就是第三部分的代码放到第一部分前面。
呵呵。到这里吧。

# Ioctl

代码中调用了多次 ioctl，我们下面就看看这个函数。
ioctl 是一个和 read，open 一样常见的文件操作函数。在 pmon 中，主要用它来和串口，显卡等字符设备交互。

```
int ioctl(int fd, unsigned long op, ...)
{
	void *argp;
	va_list ap;

	va_start(ap, op);
	argp = va_arg(ap, void *);
	va_end(ap);
```

又是对可变参数的处理。在 pmon 代码中，一般都只有三个参数，argp 就是第三个参数。

```
	if ((fd < OPEN_MAX) && _file[fd].valid )
		if (_file[fd].fs->ioctl)
			return (((_file[fd]).fs->ioctl) (fd, op, argp));
	。。。。。。
}
```

对于前 5 个 fd，上面调用的 ioctl 就是调用 term_ioctl，其余的文件一般就是普通文件，ioctl 就是文件系统的 ioctl。这里只看设备文件的 ioctl 实现 term_ioctl。

```
int term_ioctl (int fd, unsigned long op, ...)
{
	DevEntry *p;
	struct termio *at;
	int i;
	void *argp;
	va_list ap;
	struct TermDev		*devp;

	devp = (struct TermDev *)_file[fd].data;
```

data 成员在 term_open 时初始化，devp->dev 标记了这个文件是串口还是显卡。

```
	va_start(ap, op);
	argp = va_arg(ap, void *);
	va_end(ap);
```

```
    if (devp->dev < 0)
        return -1;
    p = &DevTable[devp->dev];
```

fd 为 0,1,2 时，表示为串口，为 3,4 时，表示为显卡。

```
    switch (op) {
        case TCGETA:
            *(struct termio *)argp = p->t;
            break;
```

返回这个设备的 termio 结构体指针。

```
        case TCSETAF:           /* after flush of input queue */
            while (!Qempty (p->rxq))
                Qget (p->rxq);
            (*p->handler) (OP_FLUSH, p, NULL, 1);
            if (p->rxoff) {
                (*p->handler) (OP_RXSTOP, p, NULL, p->rxoff = 0);
                if (p->t.c_iflag & IXOFF)
                    chwrite (p, CNTRL ('Q'));
            }
```

清空接收队列。

```
        case TCSETAW:           /* after write */
            /* no txq, so no delay needed */
            at = (struct termio *)argp;
            if (p->t.c_ispeed != at->c_ispeed) {
                if ((*p->handler) (OP_BAUD, p, NULL, at->c_ispeed))
                    return -1;
            }
            p->t = *at;
            break;
```

用传入的参数给设备的 termio 赋值并改变波特率。

```
        case FIONREAD:
            scandevs ();
            *(int *)argp = Qused (p->rxq);
            break;
```

返回有接收队列中还有多少个键值没有被处理。

```
        case SETINTR:
            p->intr = (struct jmp_buf*) argp;
            Break;
```

设置中断处理函数？

```
        case SETSANE:
            (*p->handler) (OP_RESET, p, NULL, 0);
            setsane(p);
```

```
                break;
```
重新初始化这个设备的控制方式。
```
        case SETNCNE:
                if (argp)
                        *(struct termio *)argp = p->t;
                p->t.c_lflag &= ~(ICANON | ECHO | ECHOE);
                p->t.c_cc[4] = 1;
                break;
```
返回但前的 termio 设置设备并退出标准模式，不回显输入。在标准模式下，输入字符存储在输入队列的缓冲区中，直到读入换行符或是 EOT(^-d)字符后才将所有数据一次输出；在原始模式下，字符一键入就立即输出，没有缓冲区。系统默认的是标准模式。
```
        case CBREAK:
                if (argp)
                        *(struct termio *)argp = p->t;
                p->t.c_lflag &= ~(ICANON | ECHO);
                p->t.c_cc[4] = 1;
                break;
```
和 SETNCNE 的唯一差别是受到删除键允许删除前一个字符。
```
        case GETTERM:
                *(int *)argp = 0;
                if (p->tfunc == 0)
                        return (-1);
                strcpy ((char *)argp, p->tname);
                break;
```
返回当前的 term 的 name。
```
        case SETTERM:
                for (i = 0; TermTable[i].name; i++) {
                        if (!strcmp(argp, TermTable[i].name))
                                break;
                }
```
这个好像没有任何作用。
```
        default:
                break;
    }
    return 0;
}
```
开始阶段在 get_boot_selection 函数中使用 ioctl (STDIN, FIONREAD, &cnt)接收 del 输入，现在很容易理解了，就是读取当前的接收队列中有几个字节的键码，结果放在 cnt 中。

# 环境变量和 flash

pmon 下环境变量可分为三种，使用 set 命令可以看到。一种是代码中写死的，如 pmon 的版本号，一种是计算出来的，比如内存大小，最后一种是写在 flash 上的，即使重刷 pmon 代码也不会改变。比如 al 这种用户自定义的变量。
envvar 是个存储环境变量结构体的数组，大小为 64 项。结构体的定义如下：

```
struct envpair {
        char      *name;
        char      *value;
};
```

至于 64 项环境变量是否够用？没有特殊需要的话，64 项肯定够了，当然非要多于 64 项，可以修改 NVAR 这个常量定义。
envinit 函数定义如下：

```
void envinit ()
{
        int i;

        bzero (envvar, sizeof(envvar));
```

bzeor()当然就是清零。

```
        tgt_mapenv (_setenv);
```

参数_setenv 是一个函数名，会在 tgt_mapenv 中被调用。功能就是注册用户定义的和计算出来的环境变量。

```
        envinited = 1;

        for (i = 0; stdenvtab[i].name; i++) {
            if (!getenv (stdenvtab[i].name)) {
                setenv (stdenvtab[i].name, stdenvtab[i].init);
```

如果这个变量用户没有定义这些变量，那么就是用标准环境变量。

```
            }
        }
}
```

下面进入 tgt_mapenv()。
这个 tgt_mapenv()的代码并不长。功能就是注册用户定义的和计算出来的环境变量。由于自定义的变量是在 flash 中，所以要先处理 flash 相关的问题。

```
        nvram = (char *)(tgt_flashmap()->fl_map_base + FLASH_OFFS);
        printf("nvram %x\n", nvram);
        if(fl_devident((void *)(tgt_flashmap()->fl_map_base), NULL) == 0 ||
                cksum(nvram + NVRAM_OFFS, NVRAM_SIZE, 0) != 0) {
```

先执行这行代码：

```
nvram = (char *)(tgt_flashmap()->fl_map_base + FLASH_OFFS);
```

Tgt_flashmap 的定义很简单，就是返回一个结构体 tgt_fl_mpa_boot8 的指针：

```
struct fl_map * tgt_flashmap()
{
    return tgt_fl_map_boot8;
}
```

Tgt_fl_map_boot8 就在函数上方定义：

```
struct fl_map tgt_fl_map_boot8[]={
    TARGET_FLASH_DEVICES_8
};
```

TARGET_FLASH_DEVICES_8 的定义为：

```
#define TARGET_FLASH_DEVICES_8 \
    { 0xbfc00000, 0x00080000, 1, 1, FL_BUS_8    },      \
    { 0xbc000000, 0x02000000, 1, 1, FL_BUS_8    },      \
    { 0x00000000, 0x00000000 }
```

盒子使用的 flash 是 8 位的，相当于这个 tgt_fl_map_boot8 数组的定义为

```
struct fl_map tgt_fl_map_boot8[]={
     { 0xbfc00000, 0x00080000, 1, 1, FL_BUS_8    },
     { 0xbc000000, 0x02000000, 1, 1, FL_BUS_8    },
     { 0x00000000, 0x00000000 }
};
```

从这个定义中可以看出，有两块 flash，一块就是 pmon，起始地址是 0xbfc00000，大小为 512KB。另一个起始地址为 0xbc000000，大小为 32MB，你可能猜到了，对，就是显存。

fl_map 这个结构体定义如下：

```
struct fl_map {
    u_int32_t fl_map_base;    /* Start of flash area in physical map */
    u_int32_t fl_map_size;    /* Size of flash area in physical map */
    int fl_map_width;         /* Number of bytes to program in one cycle*/
    int fl_map_chips;         /* Number of chips to operate in one cycle */
    int fl_map_bus;           /* Bus width type, se below */
    int fl_map_offset;        /* Flash Offset mapped in memory */
};
#define FL_BUS_8       0x01      /* Byte wide bus */
#define FL_BUS_16      0x02      /* Short wide bus */
#define FL_BUS_32      0x03      /* Word wide bus */
#define FL_BUS_64      0x04      /* Quad wide bus */
#define FL_BUS_8_ON_64    0x05    /* Byte wide on quad wide bus */
#define FL_BUS_16_ON_64 0x06    /*16-bit wide flash on quad wide bus*/
```

回到调用语句

```
nvram = (char *)(tgt_flashmap()->fl_map_base + FLASH_OFFS)
```

FLASH_OFFS 定义为 flash 大小减去 4KB，按这个定义，就是 pmon 所在芯片的最后 4KB 是可以用于存储环境变量(实际上只使用了 512 字节)。nvram 的值就是这块空间的起始地址。
接着执行 fl_devident((void *)(tgt_flashmap()->fl_map_base), NULL) == 0
tgt_flashmap()->fl_map_base 我们知道了，就是 0xbfc00000，fl_devident 一开始执行 map = fl_find_map(base);这 base 就是 0xbfc00000，
这个 fl_find_map 本质就是挨个(也只有两个:->)检查这个传入的 base 在哪儿 map 地址空间里，找到就返回这个 map 的结构体首地址，否则继续，如果没找到一个符合要求的，就返回 NULL 指针。这个地址落在 pmon 的 flash 上，返回那个 map。

```
    if (map != NULL) {
        fl_autoselect(map);
```

识别 flash 厂商和设备 id。

```
        switch (map->fl_map_bus) {
        case FL_BUS_8:
            mfgid = inb(map->fl_map_base);
            chipid = inb(map->fl_map_base + 1);

            if (chipid == mfgid) {    /* intel 16 bit flash mem */
                chipid = inb(map->fl_map_base + 3);
            }
            break;
```

其他 case：

```
        }
        fl_reset(map);

        /* Lookup device type using manufacturer and device id */
        for (dev = &fl_known_dev[0]; dev->fl_name != 0; dev++) {
            if (dev->fl_mfg == mfgid && dev->fl_id == chipid) {
                tgt_flashwrite_disable();
                if (m) {
                    *m = map;
                }
                return (dev);     /* GOT IT! */
            }
        }
        printf("Mfg %2x, Id %2x\n", mfgid, chipid);
    }
```

首先执行的是 fl_autoselect(map)，这个函数的就是执行这些 outb。

```
        switch (map->fl_map_bus) {
          case FL_BUS_8:
    #if NMOD_FLASH_SST
```

```
            outb((map->fl_map_base + SST_CMDOFFS1), 0xAA);
            outb((map->fl_map_base + SST_CMDOFFS2), 0x55);
            outb((map->fl_map_base + SST_CMDOFFS1), FL_AUTOSEL);
    #endif
    #if NMOD_FLASH_AMD
            outb((map->fl_map_base + AMD_CMDOFFS1), 0xAA);
            outb((map->fl_map_base + AMD_CMDOFFS2), 0x55);
            outb((map->fl_map_base + AMD_CMDOFFS1), FL_AUTOSEL);
    #endif
    #if NMOD_FLASH_WINBOND
            outb((map->fl_map_base + WINBOND_CMDOFFS1), 0xAA);
            delay(10);
            outb((map->fl_map_base + WINBOND_CMDOFFS2), 0x55);
            delay(10);
           outb((map->fl_map_base + WINBOND_CMDOFFS1),
                                                    FL_AUTOSEL);
            delay(10);
    #endif
```

事实上，为了支持更多的 flash，代码中的三个 NMOD_FLADH***都定义了。定义在../Targets/Bonito/compile/Bonito/flash.h 中。很幸运的是，这样的重复操作似乎没有造成什么副作用。

盒子的 pmon 芯片使用过 MX 和 SST 两个牌子，下面就以 SST 为例吧，其实不同的芯片编程方式是大同小异的。

看 SST 的命令吧。SST_CMDOFFS1 为 0x5555，SST_CMDOFFS2 为 0x2aaa。

```
        outb((map->fl_map_base + SST_CMDOFFS1), 0xAA);
        outb((map->fl_map_base + SST_CMDOFFS2), 0x55);
        outb((map->fl_map_base + SST_CMDOFFS1), FL_AUTOSEL);
```

看到 SST 手册的 P7，开头就介绍了 6 个软件命令，我们上面三条指令的顺序对于其中的第四条命令 Software ID Entry，一查 FL_AUTOSEL 就是 0x90。在按照这个指令顺序和内容发送后，芯片会返回芯片的 ID 号。fl_autoselect 函数返回后就会读取这个值。

```
        case FL_BUS_8:
            mfgid = inb(map->fl_map_base);
            chipid = inb(map->fl_map_base + 1);

            if (chipid == mfgid) {    /* intel 16 bit flash mem */
                chipid = inb(map->fl_map_base + 3);
            }
            break;
```

这两个 inb 一个返回厂商号（Manufacturer），第二个返回芯片 id，手册说这个地方的读操作地址 A0 为 0 就返回厂商号，为 1 就返回芯片 id,和其它位无关。

接着执行 fl_reset(map);

手册规定在这种状况下，往任一地址写 0xf0 表示芯片信息读取结束。
到现在为止，fl_devident 的功能就是按照芯片的时序读取芯片信息。接着查看 pmon 是否支持这个型号的芯片。

```
for (dev = &fl_known_dev[0]; dev->fl_name != 0; dev++) {
    if (dev->fl_mfg == mfgid && dev->fl_id == chipid) {
        tgt_flashwrite_disable();
        if (m) {
            *m = map;
        }
        return (dev);     /* GOT IT! */
    }
}
```

fl_known_dev 是一个 flash 芯片型号信息的数组，包括了所有明确支持的 flash 芯片。这个 for 循环挨个查找刚才读取的值是否在这个列表中，如果在的话，为传入的二级指针 m 赋值并返回 dev 这个结构的首地址。
好了，从这个函数返回到 tgt_mapenv，执行条件判断代码的下一个条件判断，

```
if(fl_devident((void *)(tgt_flashmap()->fl_map_base), NULL) == 0 ||
        cksum(nvram + NVRAM_OFFS, NVRAM_SIZE, 0) != 0) {
```

cksum 就是把 nvram + NVRAM_OFFS 这个地址开始的 NVRAM_SIZE(这里是 512)个字节求和,这段空间一定和是 0 才正常吗？对，这 512 个字节的第一个半字存放的就是后面 255 个半字的和的相反数，所以和为 0。
这个 if(fl_devident((void *)(tgt_flashmap()->fl_map_base), NULL)语句可绕了不少时间了，概括的说就是看看我们主板的那个 pmon 芯片是不是我们支持的型号，并且是否正常。
这最后 4KB 的内容究竟是什么呢？读读看。

```
!ifconfig rtk0:172.16.2.22
setvga=0
al=dev/fs/ex2@wd0/boot/vmlinux-2.6.27.1
karg=root=/dev/hda1 console=tty
```

以上输出是通过在 cksum 函数中添加这段代码看到的

```
u_int8_t *sp1 = p;
while(i<512){
    if((*sp1)==0)
        printf("\n");
    else
        printf("%c",*sp1);
    if((i%5)==0)      //不能一行打印太多，否则显示异常。
        printf("\n");
    i++;
    sp1++;
}
```

在进入 pmon 的命令提示符后，输入 set，可以看到环境变量的前 4 个就是我们读出来的那 4 个，一模一样。在 pmon 中再设置一个变量并写入 flash 后，再次启动这个读出的可读变量就多了一个，就是我们刚添加的内容，可见这些变量都是用户自定义的变量。
现在看接下来的代码就很清楚了。接下来的代码如下：

```
while((*p++ = *ep++) && (ep <= nvram + NVRAM_SIZE - 1) &&
                                                    i++ < 255) {
    if((*(p - 1) == '=') && (val == NULL)) {
        *(p - 1) = '\0';
        val = p;
    }
}
```

这个对环境变量读取并解析。接着执行：

```
if(ep <= nvram + NVRAM_SIZE - 1 && i < 255) {
    (*func)(env, val);
}
```

在提取一个环境变量名和它的值（如果存在的话）后，调用 (*func)(env, val);这个 func 是传入的参数，回到 tgt_mapenv 函数的调用处，tgt_mapenv (_setenv);传入的参数是_setenv,是一个函数名，现在要调用的就是这个函数。
_setenv 函数在../pmon/common/env.c，函数原型为 static int _setenv (char * name, char *value)，从命名就看出来了，参数表示将要注册的变量名和变量值。
暂时不进入这个函数，先把 tgt_mapenv 这个函数的流程理完。

```
bcopy(&nvram[ETHER_OFFS], hwethadr, 6);
sprintf(env, "%02x:%02x:%02x:%02x:%02x:%02x",
    hwethadr[0], hwethadr[1],
hwethadr[2], hwethadr[3], hwethadr[4], hwethadr[5]);
(*func)("ethaddr", env);
```

hwethadr 是一个长度为 6 的全局字节型数组，这句代码把相应位置的数据读出来作为网卡的 mac 地址。 并写入 env 数组，注册 ethaddr 这个环境变量。看 env 这个命令的输出，除了用户自定义的环境变量，ethaddr 是最前面的。
接着看，注册几个计算出来的环境变量，代码如下：

```
sprintf(env, "%d", memorysize / (1024 * 1024));
(*func)("memsize", env);

sprintf(env, "%d", memorysize_high / (1024 * 1024));
(*func)("highmemsize", env);

sprintf(env, "%d", md_pipefreq);
(*func)("cpuclock", env);
```

```
    sprintf(env, "%d", md_cpufreq);
    (*func)("busclock", env);

    (*func)("systype", SYSTYPE);
```

命名都是很清楚的。

这些执行完之后回到 envinit()，剩下的也不多了。

```
    for (i = 0; stdenvtab[i].name; i++) {
        if (!getenv (stdenvtab[i].name)) {
            setenv (stdenvtab[i].name, stdenvtab[i].init);
        }
    }
```

首先执行 getstdenv (name)，检验我们要注册的变量名和保留标准环境变量名是否冲突。比如 Version 这个名字就是一个保留的名字。

这就不关心命名冲突的问题了，有兴趣的可以自己阅读。

好了，现在进入关键的_setenv 函数吧。下面以在 pmon 下使用“set al /dev/fs/ext2@wd0/boot/vmlinux”这个命令为例解析。set 命令关键就是调用 do_setenv，代码如下：

```
Int do_setenv (char *name, int value, int temp)
{
    if (_setenv (name, value)) {
```

首先就是对_setenv 函数的调用，这个函数稍后解释。

```
        const struct stdenv *sp;
        if ((sp = getstdenv (name)) && striequ (value, sp->init)) {
            /* set to default: remove from non-volatile ram */
            return tgt_unsetenv (name);
```

如果这个变量是重名的，那么就删掉旧的变量。

```
        }
        else if(!temp) {
            /* new value: save in non-volatile ram */
            return tgt_setenv (name, value);
```

tgt_setenv 会把新环境变量写到 flash 中。这个等会再看

```
        }
        else {
            return(1);
        }
    }
    return 0;
}
```

我们只关心_setenv()和 tgt_setenv()两个函数。先看_setenv()。

代码首先会判断这个要注册的变量是否和标准变量（就是代码中写死的那些）重名，我们直接跳过重名的情形。

```
for (ep = envvar; ep < &envvar[NVAR]; ep++) {
    if (!ep->name && !bp)   //如果这个 ep 没有名字，并且 bp 为 NULL
        bp = ep;            // 选择这个 bp 作为新注册的环境变量的指针
    else if (ep->name && striequ (name, ep->name)) //看这个新注册的变
                                                   //量名是否和以前注册的变量名冲突
        break;
}
```

上面这个循环执行以后，如果 bp 为 NULL，说明变量数量已满或者命名冲突，我们不关心非正常的情形。

```
ep = bp;
if (!(ep->name = malloc (strlen (name) + 1)))
    return 0;
strcpy (ep->name, name);
```

分配空间并把 name 和 value 放入。代码够简单吧。

上面的代码只是在内存里注册了环境变量，重启后就没了。tgt_setenv()是往 flash 上写，断电后依然保存。这个函数不短，但逻辑很清楚，就是从 pmon 的最后 4KB 开始的地方读取 512 字节到首地址为 nvramsecbuf 一个内存空间中，在通过合法性和可行性（放不放得下）判断后。再把这个 nvramsecbuf 的内容写回去。我们只看最后关键的两行。

```
cksum(nvrambuf, NVRAM_SIZE, 1);
fl_program(nvram, nvramsecbuf, NVRAM_SECSIZE, TRUE);
```

前面曾经见过 cksum，就是做检验和，第三个参数为 1 表示 set，由于增加了一个变量后校验和就不一定为 0 了，所以要修改第一个半字（16 位），内容为后 255 个半字的和的相反数。这样校验和就又为 0 了。

fl_program 是具体的刷新 flash。第一个参数是这次操作 flash 的开始地址，第二个参数是要写入的字符串，第三个参数是写入的字节数，最后一个参数和打印有关。fl 是 flash 的缩写，函数代码如下：

```
int fl_program(void *fl_base, void *data_base, int data_size, int verbose)
{
    void *base = fl_base;
    int size = data_size;
    char *tmpbuf;

    get_roundup(&base, &size);
```

处理一些地址对齐相关的东西。比如 MX 的 flash 如果一次写一块，块大小必须是 64KB 的倍数，即使我们只想写一个字节。SST 的块是 4KB。

```
    printf("base %x, size %x\n", base, size);
```

这个 base 在现在肯定就是 0xbfc70000， size 就是 0x10000

```
    tmpbuf = (char *)malloc(size);
    if (tmpbuf == 0) {
```

```
            printf("[fl_program] can't malloc");
            return -1;
        }

        memcpy(tmpbuf, base, size);
```
把将要改变到的 flash 上的内容读道 tmpbuf 中，这里是 64KB
```
        memcpy(tmpbuf + (unsigned int)fl_base - (unsigned int)base,
            data_base, data_size);
```
替换修改的部分，现在 tmpbuf 准备好了要写回去的内容。

flash 区别于块设备的一个明显特征就是写操作要先擦除再编程。
```
        if (fl_erase_device(base, size, verbose) == 0 &&
            fl_program_device(base, tmpbuf, size, verbose) == 0){
            return 0;
        }
        return -1;
    }
```
flash 擦除和编程的操作函数都在../pmon/dev/flash.c 中定义。

先看 erase 的操作 fl_erase_device()。这个函数比较长，但有一半都是合法性和可行性检查，我们跳过。

执行到这，base 表示这次操作的首地址，size 表示操作的大小，block 表示操作从第几块开始。
```
    while (size > 0) {
        int boffs = (int)base;
        if (size == map->fl_map_size &&
            dev->fl_cap & (FL_CAP_DE | FL_CAP_A7)) {
```
如果要擦除的地址从 flash 起始地址开始并且这个 flash 是要求要擦全擦的。那么就全部擦出。
```
        } else {
            if ((*dev->functions->erase_sector) (map, dev, boffs) != 0) {
```
这个 erase_sector 函数是这里我们唯一在意的。就是擦除特定的块。
```
                printf("\nError: Failed to enter erase mode\n");
                (*dev->functions->erase_suspend) (map, dev);
                (*dev->functions->reset) (map, dev);
                return (-4);
            }
```
下面的代码用于计算是否和如何擦除下一块。
```
            ..........
        }
        delay(1000);
        for (timeout = 0; ((ok =(*dev->functions->isbusy) (map, dev,
                        0xffffffff, boffs, TRUE)) == 1) &&
                        (timeout < PFLASH_MAX_TIMEOUT); timeout++) {
```

```
            。。。等待直到擦除操作完成。。。。
        }
        delay(1000);
        if (!(timeout < PFLASH_MAX_TIMEOUT)) {
            (*dev->functions->erase_suspend) (map, dev);
        }
        (*dev->functions->reset) (map, dev);
    }

    tgt_flashwrite_disable();
    return (ok);
}
```

代码中调用了很多 dev->functions 的函数，对于 sst 的 flash 函数指针列表如下：

```
struct fl_functions fl_func_sst = { fl_erase_chip_sst,
    fl_erase_sector_sst,
    fl_isbusy_sst,
    fl_reset_sst,
    fl_erase_suspend_sst,
    fl_program_sst
};
```

看擦除一块的函数：

```
int fl_erase_sector_sst(struct fl_map *map, struct fl_device *dev, int offset)
{
    switch (map->fl_map_bus) {
    case FL_BUS_8:
        outb((map->fl_map_base + SST_CMDOFFS1), 0xAA);
        delay(10);
        outb((map->fl_map_base + SST_CMDOFFS2), 0x55);
        delay(10);
        outb((map->fl_map_base + SST_CMDOFFS1), FL_ERASE);
        delay(10);
        outb((map->fl_map_base + SST_CMDOFFS1), 0xAA);
        delay(10);
        outb((map->fl_map_base + SST_CMDOFFS2), 0x55);
        delay(10);
        outb((map->fl_map_base + offset), FL_SECT);
        delay(10);
        break;
    case 其它
        。。。。。。。。
}
```

可见要擦除 flash 的一个块，要发送 6 个字节的命令，最后一个数据表示了要擦除的地址，flash 操作的命令见 SST 手册的 P7。清清楚楚。
写操作是先擦除再编程，编程的命令要简单一些，只发 4 个字节，具体的看代码和芯片手册，都没有什么东西啦。

# GPIO

在 pmon 中和 GPIO 相关的代码并不多，但是 GPIO 的操作对于 pmon 来说是必须的。谁用谁知道。
GPIO 的操作根据 pci 地址是否分配可以分为两个阶段，那么前一阶段的地址是代码中写死的。后一阶段的地址是 pmon 分配函数分配给各个 pci 设备的。
首先看前一阶段的 GPIO 操作。以下是对南桥 GPIO5 的操作代码，在 start.S 中

```
GPIO_HI_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_OUT_EN);
GPIO_LO_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_IN_EN);
GPIO_LO_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_OUT_AUX1_SEL);
GPIO_LO_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_OUT_AUX2_SEL);
GPIO_LO_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_IN_AUX1_SEL);
GPIO_HI_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_PU_EN);
```

从宏名上看就可以猜想功能了，GPIO_HI_BIT 就是把那个位置高。
再看看两个输入参数的定义：

```
#define GPIO_5          5

#define DIVIL_BASE_ADDR         0xB000
#define SMB_BASE_ADDR           (DIVIL_BASE_ADDR | 0x320)
#define GPIO_BASE_ADDR          (DIVIL_BASE_ADDR | 0x000)
```

至于 GPIOL_OUT_EN 这样的寄存器，都是相对于 base 做个偏移而已。
从上面的定义可以看出 GPIO 那一套寄存器的基地址是 0xB000。SMB 那一套寄存器的基地址是 0xB320。
传入参数都清楚了，看看 GPIO_HI_BIT 这个宏都做了些什么。

```
#define GPIO_HI_BIT(bit, reg) \
    lui v1, 0xbfd0; \
    ori v1, reg;        \
    lw    v0, 0(v1);    \
    li    a0, 0x0001; \
    sll a0, bit;        \
    or    v0, a0;         \
    sll a0, 16;         \
    not a0;               \
    and v0, a0;         \
```

```
    sw    v0, 0(v1);
```
这个代码很简单，不妨翻译成 C 代码吧。
```
static void GPIO_HI_BIT(int bit, int reg)
{
    int orig;

    orig = *(unsigned int *)(0xbfd00000 + reg);
    orig = orig   |   (1 << bit);
    orig = orig   & ( ~(1 << (16 + bit)));
    *(unsigned int *)(0xbfd00000 + reg) = orig;
}
```
看到了吧，设置一个位的操作和我们平时的单纯把一个位置为 1 有差别。这些寄存器的布局都如同 P484 的表格一样，都是 32 位的，但是高低 16 位的位定义是相同的。比如要使能位 3 的功能，不但要把位 3 置为 1，而且要把位 19（16＋3）置为 0 方为有效的置为，清某个位刚好相反。

现在再看看这个代码做了什么：
```
    GPIO_HI_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_OUT_EN);
```
输出使能
```
    GPIO_LO_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_IN_EN);
```
禁止输入
```
    GPIO_LO_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_OUT_AUX1_SEL);
    GPIO_LO_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_OUT_AUX2_SEL);
    GPIO_LO_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_IN_AUX1_SEL);
```
上面三句合在一块表示一个意思，就是这个 GPIO 口只使用普通的 IO 输出功能。GPIO 嘛,就是多功能，至于表现成那个功能，就要配置了，GPIOL_OUT_AUX1_SEL，GPIOL_OUT_AUX2_SEL，GPIOL_IN_AUX1_SEL 都是用于配置功能的寄存器。比如在南桥手册 P28 显示 GPIO6 这个引脚就可以配置成 4 种功能，如果要配成 MFGPT0_C1 这个功能，就要在 GPIOL_OUT_AUX1_SEL 这个寄存器的相应位使能。这里把那三个寄存器的描述 GPIO_5 的那位都 disable 掉，就是说不使用那些功能，而是作为一般的输出输入口来用。
```
   GPIO_HI_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_PU_EN);
```
这个是使能一种拉高属性。

上面这个代码如果真的要输出的话，一般还要写一句
```
  GPIO_HI_BIT(GPIO_5, GPIO_BASE_ADDR | GPIOL_OUT_VAL);
```
就是输出的时候设置输出什么电平，上面这句就是输出的时候输出高电平。

我们说了，第一阶段这些寄存器的地址是代码里写死的，这个在本文的第一章有分析，第二阶段的地址是 pci 分配之后的，是多少呢。在 pmon 中提供了一个命令 rdmsr 来获得相应的基地址。

比如我想知道第二阶段 GPIO 这块的基地址，就输入 rdmsr 8000000c。得到这个的基地址是 0xb000。上面为什么查询地址是 8000000c 呢，下面就简单介绍一下。在南桥手册的 P343，我们看到 GPIO 的基地址可以通过 51400000c 这个地址访问到，SMB 的基地址可以通过 51400000b 这个地址访问到，至于 8000_000c 这个

地址是如何转化成 51400000c 的，看 msr 的地址路由吧（手册 P60）。
在 pmon 的代码 pmon/cmds/my_cmd.c 中，有比较完整的 GPIO 第二阶段的操作示例，可以作为参考。

# 你怎么出来了 －－ 图片显示

pmon 基本上是一个黑白的世界。所以能看到些花花绿绿的图片，的确叫人分外眼红呀，截图如下：


按照官方提供的方法，图片显示的操作为把压缩后 8 位色的 bmp 图片烧入 bfc60000 这个地址就行了。
这个图片太土了，好，我们 DIY 一下。
我想把 QQ 的图像放到 pmon 界面的右上角，就像平时在系统中一样，截了个 QQ 的图，如下：


这个图的属性如下：

qq.bmp: PC bitmap data, Windows 3.x format, 237 x 563 x 24

对，是 24 位色的，和官方的不一致。管他，试试先。

用 gzip 压成 qq.bmp.gz，用 load -r -f bfc60000 xxxx/qq.bmp.gz，显示效果如何，缺一节。看样子好像是太长了。因为默认情况下图片不是顶天立地的。所以被截了一块，刚好把我的头像截了一半，好险。不，不是好险，是好惨。

看样子要看代码了。在 fb_init 中，就是下面这句。

video_display_bitmap(BIGBMP_START_ADDR, BIGBMP_X, BIGBMP_Y);

BIGBMP_X 和 BIGBMP_Y 分别是 288，100，可见这个坐标就是图片显示的左上角地址。我们要知道，8089 的分辨率是 1024x600，而 qq 这个图片的高是 563，加上 100 的确超过了 600，被截掉就正常了。既然如此，换个坐标显示不久行了，把显示左上角的坐标改成 700，0 不久行了。哈哈。试试。

修改后烧入显示一塌糊涂，面目全非。会不会是图片太大了。没准，试了个小的，如下：


这个图片的属性如下：

haha.bmp: PC bitmap data, Windows 3.x format, 162 x 195 x 24

试了下这个图片的压缩版，也不能正常的放到右上角。好家伙，bug 吧。没方法了，看代码。

首先看到，这个代码会判断传入的欲显示图片地址存放的是不是 bmp 格式的，如果不是就默认为 bmp 的压缩格式。

看到这，可见 bmp 格式也是支持的。一试上面那个笑脸的图片，烧入 bmp 的话，显示到左上角是没有问题的，可见压缩格式的处理可能是有问题的。可是 QQ 的图片太大了，只能压缩，否则就 ok 了，要个性，就要折腾。看代码。

这个 video_display_bitmap 函数代码看起来长，其实很清楚。

首先我们跳过对压缩格式的处理，先看对 bmp 格式的处理，因为这比较简单，而且压缩的格式解压后，也会执行同样的流程。

```
width = le32_to_cpu(bmp->header.width);
height = le32_to_cpu(bmp->header.height);
bpp = le16_to_cpu(bmp->header.bit_count);
colors = le32_to_cpu(bmp->header.colors_used);
compression = le32_to_cpu(bmp->header.compression);
```

首先读取这个 bmp 图片的几个核心参数，分别是图片的长宽，色深，是否压缩，其中的 colors 我们没有使用。

有兴趣的（比如我）可以使用二进制读取工具看看上面的那两个 bmp 图片头部相关域的值。我用 khexedit 这个工具感觉不错。

这里我们得到的 bpp 是 3，表示是 24 位色。关于 bmp 图片的头部格式，网上一大堆。

```
padded_line = (((width * bpp + 7) / 8) + 3) & ~0x3;
```

由于 bmp 图片的存储有个 4 字节对齐的要求，所以这里计算为了对齐，每行的信息存储要多少个字节来填充（pad）。

```
if ((x + width) > VIDEO_VISIBLE_COLS)
    width = VIDEO_VISIBLE_COLS - x;
if ((y + height) > VIDEO_VISIBLE_ROWS)
    height = VIDEO_VISIBLE_ROWS - y;
```

是否越界的检查，如果超过了显示区域，弃之。

```
bmap = (unsigned char *)bmp + le32_to_cpu(bmp->header.data_offset);
```

很明显了，是像素信息存储的基地址。

```
fb = (unsigned char *)(video_fb_address +
                    ((y + height -1) * VIDEO_COLS * VIDEO_PIXEL_SIZE) +
                   x * VIDEO_PIXEL_SIZE);
```

fb 嘛，就是 framebuffer。这个地址有些怪不是。看看这个 fb 对应的是屏幕上的哪个像素点。就是图片应显示区域的最后一行的左边起始像素。

的确很奇怪呀，一查 bmp 格式的资料，有了。

bmp 的存储设计很怪，一张图片，最先存放的是最后一行的像素信息。然后 4 字节对齐一下，不对齐就填充下。接着存储导数第二行，如此直到第一行。

大家评评理，是不是脑子进水了。还是有什么奇特的原因?

```
switch (le16_to_cpu(bmp->header.bit_count)) {
```

前面说了，是 24 位色的。就 case 24 了。

```
case 8:
        。。。。。。
case 24:
    padded_line -= 3 * width;
```

好，现在的 padded_line 的值就是每行像素信息存放时为对齐而增加的字节数。

```
    ycount = height;
    switch (VIDEO_DATA_FORMAT) {
```

这个在前面讲 printf 那一章就见到过了，pmon 中的 framebuffer 都是 16 位色的。采用的是 565 的 RGB 编码方式。就是 case GDF_16BIT_565RGB 了。

```
        case GDF__8BIT_332RGB:
            。。。。
        case GDF_16BIT_565RGB:
          while (ycount--) {
            xcount = width;
            while (xcount--) {
                FILL_16BIT_565RGB(bmap[2], bmap[1],
                        bmap[0]);
                bmap += 3;
            }
```

每次这个 while 执行完，就显示了一行。FILL_16BIT_565RG 函数负责把 888 模式转化成 565 的模式，实现就是丢些精度。

```
            bmap += padded_line;
```

跳过补齐用的内容，不要显示

```
            fb -= (VIDEO_VISIBLE_COLS +width) * VIDEO_PIXEL_SIZE;
```

一行显示完了，显示上一行。这个是由 bmp 格式的存储方式决定的。

```
        }
```

就是这样一行一行显示，一幅 bmp 图就出来了。

我们接着回头看看压缩的 bmp 是如何处理的。

bmp 图片的明显特征就是头部的前面两个自己是字符 B 和 M 的码。

代码如下:

```
    if( (x == BIGBMP_X) && (y == BIGBMP_Y) ){
```

```
        is_bigbmp = 1;
    }else{
        is_bigbmp = 0;
    }
```

这句话看出，有个 bigbmp 的概念，事实上默认显示的那个图片是由 3 个部分组成的：一个大图片和两个汉字图片。显示时的处理有不同，具体的往下看。

```
    if(is_bigbmp){
            dst_size = 0x80000;
            len = 0xd000;
    }else{
            dst_size = 0x8000;
            len = 0x300;
    }
```

是吧，如果是大的图片，代码认为从 bfc60000 开始的 0xd000 的范围内都是这个图片的内容。否则就是小图片，大小在 0x300 以内。

这里的 dst_size 表示预计的解压后大小的上限。

```
    bg_img_src = malloc(len);
    if (bg_img_src == NULL) {
        。。。
    }
    dst = malloc(dst_size);
    if (dst == NULL) {
        。。。
    }
```

空间申请。

```
    memcpy(bg_img_src, bmp_image, len);
```

把压缩的图片放到 bg_img_src 中。

```
    if( gunzip(dst, dst_size, (unsigned char *)bg_img_src, &len) != 0 ){
        。。。
    }
```

解压那个图片到 dst 那快区域，这个空间是我们刚刚申请的。

```
    bmp = (bmp_image_t *) dst;
```

至此，下面的代码就不知道我们处理的 bmp 图片究竟是不是压缩过的，对于后面的代码处理的就是 bmp 格式了。

好，图片显示的代码介绍完毕。我们的 DIY 行动也很容易了。

关键就是传入的坐标不合，导致就认为是小图片了，这个可以改呀，改后，显示正常。在 pmon 下也可以上 QQ 喽。

想试试嘛，自己动手。