详解嵌入式 linux 启动信息（推荐）

作者：佚名 来源：不详 发布时间：2006-10-27 8:12:31 发布人：zangyl

摘要

我们在这里讨论的是对嵌入式 linux 系统的启动过程的输出信息的注释，通过我们的讨论，大家会对嵌入式 linux 启动过程中出现的、以前感觉熟悉的、但却又似是而非的东西有一个确切的了解，并且能了解到这些输出信息的来龙去脉。

嵌入式 linux 的启动信息是一个很值得我们去好好研究的东西，它能将一幅缩影图呈现在我们面前，来指导我们更加深入地理解 linux 内核。

关键字：linux，嵌入式，启动，bootloader

正文

作为一名嵌入系统开发者，你一定遇到过下面的情景：

在某论坛上看到一篇帖子，上面贴着嵌入式 linux 开发板启动时的有关信息，然后大家在帖子里讨论着这个启动过程中出现的问题，随机举例如下：

```
Linux version 2.4.20-uc0 (root@Local) (gcc version 2.95.3
20010315 (release)(ColdFire patches - 20010318 from http://f/
(uClinux XIP and shared lib patches from http://www.snapgear.com/)) #20 三 6月 1
8 00:58:31 CST 2003
Processor: Samsung S3C4510B revision 6
Architecture: SNDS100
On node 0 totalpages: 4096
zone(0): 0 pages.
zone(1): 4096 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/rom0
Calibrating delay loop... 49.76 BogoMIPS
Memory: 16MB = 16MB total
Memory: 14348KB available (1615K code, 156K data, 40K init)
Dentry cache hash table entries: 2048 (order: 2, 16384 bytes)
Inode cache hash table entries: 1024 (order: 1,
Mount-cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer-cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 4096 (order: 2, 16384 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
Starting kswapd
Samsung S3C4510 Serial driver version 0.9 (2001-12-27) with no serial options en
abled
ttyS00 at 0x3ffd000 (irq = 5) is a S3C4510B
ttyS01 at 0x3ffe000 (irq = 7) is a S3C451
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: BE558-1A5D57 [VIRTUAL BE558-1A5D57] (RO)
RAMDISK driver initialized: 16 RAM disks of 1024K size 1024 blocksize
Samsung S3C4510 Ethernet driver version 0.1 (2002-02-20) <mac@os.nctu.edu.tw>
eth0: 00:40:95:36:35:34
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 1024 bind 1024)
VFS: Mounted root (romfs
Freeing init memory: 40K
```

上面的这些输出信息，也可能包括你自己正在做的嵌入式 linux 开发板的输出信息，其中的每一行，每一个字的含义，你是否深究过，或者说大部分的含义你能确切地知道的？本人想在这里结合本人在实践中一些体会来和广大嵌入式 linux 的开发者一起读懂这些信息。

我们在这里将以一个真实的嵌入式 linux 系统的启动过程为例，来分析这些输出信息。启动信息的原始内容将用标记标出，以区别与注释。

嵌入式 linux 的启动主要分为两个阶段：

① 第一部分 bootloader 启动阶段

② 第二部分 linux 内核初始化和启动阶段

第一节：start_kernel

第二节：用户模式(user_mode)开始，start_kernel 结束

第三节：加载 linux 内核完毕，转入 cpu_idle 进程

第一部分 ： bootloader 启动

```
Boot loader v0.12
NOTE: this boot loader is designed to boot kernels made with the
2.4.xx releases
bootloader for XV
Built at Nov 20 2005 10:12:35
```

Bootloader 头信息，版本，编译时间等，这个因不同的 bootloader 的设计而有所不同，由此你能看出 bootloader 的版本信息，有很多使用的是通用的 bootloader，如 u-boot，redboot 等。

```
Loaded to 0x90060000
```

将 bootloader 加载到内存 ram 中的 0x90060000 处，即将 bootloader 加载到内存的高端地址处。

Linux 内核将被 bootloader 加载到 0x90090000 处。

```
Found boot configuration
```

查找到了启动 boot 的配置信息

```
Booted from parallel flash
```

从 flash 中启动代码，此处的 flash 为并行闪存。Flash 的分类列举如下：

闪存分三类：并行，串行，不可擦除。

①并行 Parallel flash

NOR Flash，Intel 于 1988 年发明．随机读取的速度比较快，随机按字节写，每次可以传输 8Bit。一般适合应用于数据/程序的存贮应用中．NOR 还可以片内执行(execute-in-place)XIP．写入和擦除速度很低。

NAND Flash，1989 年，东芝公司发明．是以块和页为单位来读写的，不能随机访问某个指定的点.因而相对来说读取速度较慢，而擦除和写入的速度则比较快,每次可以传输 16Bit,一般适用在大容量的多媒体应用中，容量大。如：CF，SM.

②串行 Serial Flash 是以字节进行传输的，每次可以传输 1-2Bit.如：MMC,SD,MS 卡．串

行闪存器件体积小，引脚也少，成本相对也更低廉。

③不可擦除 Mask Rom Flash 的特点是一次性录入数据，具有不可更改性，经常运用于游戏和需版权保护文件等的录入。其显著特点是成本低。

注意：任何 flash 器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。NAND 器件执行擦除操作是十分简单的，而 NOR 则要求在进行擦除前先要将目标块内所有的位都写为 0。

从上面的信息，我们可以对 flash 类型特点有个比较明确的了解。

CPU clock rate: 200 MHz

开发板上所使用的 CPU 的主频为 200MHZ.

DRAM size is 128MB (128MB/0MB)

动态内存 ram 大小为 128M。这里我们列举一下内存的类型及工作原理。

根据内存的工作原理可以划分出两种内存：DRAM 和 SRAM

①DRAM 表示动态随机存取存储器。这是一种以电荷形式进行存储的半导体存储器。DRAM 中的每个存储单元由一个晶体管和一个电容器组成。数据存储在电容器中。电容器会由于漏电而导致电荷丢失，因而 DRAM 器件是不稳定的。为了将数据保存在存储器中，DRAM 器件必须有规律地进行刷新。

②SRAM 是静态的，因此只要供电它就会保持一个值。一般而言，SRAM 比 DRAM 要快，这是因为 SRAM 没有刷新周期。每个 SRAM 存储单元由 6 个晶体管组成，而 DRAM 存储单元由一个晶体管和一个电容器组成。相比而言，DRAM 比 SRAM 每个存储单元的成本要高。照此推理，可以断定在给定的固定区域内 DRAM 的密度比 SRAM 的密度要大。

SRAM 常常用于高速缓冲存储器，因为它有更高的速率；而 DRAM 常常用于 PC 中的主存储器，因为其拥有更高的密度。

在嵌入式系统中使用 DRAM 内存的设计比较广泛。

地址辅助说明：

先说明一下内存地址数字情况，主要是为了方便记忆。
可以访问的内存为 4G。
0x40000000 是 1GB 处；0x00040000 是 256K 处，0x00020000 是 128K 处，0x90000000 是 2 GB 多的地方。
1M->0x00100000,
2M->0x00200000,
8M->0x00800000
16M->0x01000000,
32M->0x02000000
256M->0x10000000
64K->0x00010000
4K->0x00001000
这个是个快速记忆的方法，你可以根据地址中 1 的位置和其后 0 的个数来快速知道换算后的地址是在多少兆的地方。比如，1 的后面 5 个 0，代表 1M 的大小，6 个 0，代表 16M，以此类推。

ROMFS found at 0x46040000, Volume name = rom 43f291aa

romfs,只读文件系统所在的地址为：0x46040000 (flash 映射后的第 3 分区)。
卷名为 rom。
romfs 和 rootfs 概念上有所区别。

flash 在内存中的的起始地址为 0x46000000,而 ROMFS 在 flash 分区上的起始位置为 0x00040000，所以 ROMFS 在内存地址中的位置就为 0x46040000。这个细节的部分可以参考 flash 分区时的地方，Creating 3 MTD partitions。

romfs 中包括 kernel 和 app 应用，不包括 bootloader 和 firmware 信息头。romfs 只读文件系统里的内容有很多种分类方法，我们可以将 kernel 和 app 同时放里面，作为根文件系统下的一个文件，也可以在 flash 上另外划分区域来分别存放。

VFS 虚拟文件系统交换器
在 linux 系统中，目前已经开发出多种文件系统，那么如何让这些文件系统能共存在一个系统中呢，从 linux 2.0 开始，引入了虚拟文件系统管理器 VFS 的概念。
Linux 下的文件系统主要可分为三大块：

① 一是上层的文件系统的系统调用，
② 二是虚拟文件系统交换器 VFS(Virtual Filesystem Switch)，
③ 三是挂载到 VFS 中的各实际文件系统，例如 ext2，jffs 等。

VFS 的确切叫法是 Virtual Filesystem Switch 虚拟文件系统交换器，这里的 VFS 中的“S”是

指的 switch，这个需要强调一下的，它很容易被混淆成“system”，如果理解成“system”将是不正确的，请多加注意。
VFS 是具体文件系统 filesystem 的一个管理器。
VFS 是 Linux 内核中的一个软件层，一种软件机制，它也提供了内核中的一个抽象功能，允许不同的文件系统共存，可以称它为 Linux 的文件系统管理者，与它相关的数据结构只存在于物理内存当中。所以在每次系统初始化期间，Linux 都首先要在内存当中构造一棵 VFS 的目录树。VFS 中的各目录其主要用途是用来提供实际文件系统的挂载点。而 rootfs 将是这个目录树的根结点的（root），即 "/"目录，VFS 的结构就是从这个 rootfs 开始的。有了 VFS，那么对文件的操作将使用统一的接口，将来通过文件系统调用对 VFS 发起的文件操作等指令将被 rootfs 文件系统中相应的函数接口所接管。

注意：rootfs 并不是一个具体的文件系统类型，如 jffs。它只是一个理论上的概念。在具体的嵌入系统实例中，可以将某种具体的文件系统设置为根文件系统 rootfs，如我们可以设置 romfs 为根文件系统，也可以设置 jffs 为根文件系统。

这里的 ROMFS 只读文件系统只是一种具体的文件系统类型，也是在嵌入系统中经常使用到的类型。

可见这个内核存储在FLASH中

看完了上面的内容，以后你对出现的类似“kernel Panic:VFS:Unable to mount root fs on 0:00”的含义应该已经了解了。其中“VFS:”就是虚拟文件系统管理器操作时的输出信息了。

```
File linux.bin.gz found
```

linux kernel 内核文件名，它是在只读文件系统 romfs 上的一个组成部分。

```
Unzipping image from 0x4639DE60 to 0x90090000, size = 1316021
```

将 romfs 中的 linux kernel 解压缩到 0x90090000,之后会从这个内存地址启动内核。romfs 为压缩格式文件,使用压缩的只读文件系统，是为了保持制作出来的整个系统所占用的 flash 空间减小。这个内核的大小为 1.3M 左右，这也是目前大多数嵌入系统所使用的方法。

```
Inptr = 0x00000014(20)
Inflating....
```

释放，解压中。。。（变大，充气，膨胀）

```
Outcnt = 0x0030e7c8(3205064)
Final Inptr = 0x001414ad(1316013)
Original CRC = 0xcbd73adb
Computed CRC = 0xcbd73adb
```

做释放后的 CRC 检查

```
Boot kernel at 0x90090000 with ROMFS at 0x46040000
```

kernel 已经被从 romfs 中释放到内存地址 0x90090000 处，可以跳转到此处启动 kernel 了，这里是指定的 kernel 的起始地址

```
Press 'enter' to boot
```

系统等待启动，后面将看到 linux kernel 的启动过程了。

第二部分 ：linux 内核初始化以及启动

第一节：start_kernel

Linux的源代码可以从http://www.kernel.org/得到，或者你可以查看linux代码交叉引用网站：http://lxr.linux.no/ 进行在线的代码查看，这是一个很好的工具网站。

在 start_kernel 中将调用到大量的 init 函数，来完成内核的各种初始化。如：

page_address_init();
sched_init();
page_alloc_init();
init_IRQ();
softirq_init();
console_init();
calibrate_delay();
vfs_caches_init(num_physpages);
rest_init();

具体内容可以参考[http://lxr.linux.no/source/init/main.c]

```
Linux version 2.4.22-uc0 (root@local) (gcc version 2.95.3 20010315 (release)) #33 .?1.. 20 12:09:106
```

上面的代码输出信息，是跟踪 linux 代码分析后得到的，进入 init 目录下的 main.c 的 start_kernel 启动函数.

嵌入式 linux 使用的是 linux 内核版本为 2.4.22

linux source code 代码中 start_kernel 中输出的 linux_banner 信息。这个信息是每个 linux kernel 都会打印一下的信息，如果你没有把这句去掉的话。

```
Found bootloader memory map at 0x10000fc0.
```

bootloader 经过内存映射后的地址为：0x10000fc0, 按上面的地址换算方法，1 后面有 7 个 0，那么虚拟地址 256M 左右处。

```
Processor: ARM pt110 revision 0
```

pT110 是 ARM 微处理器 arm 核的一种，另一种为 pT100。此处为显示 ARM 的类型。

```
On node 0 totalpages: 20480
zone(0): 20480 pages.
zone(0): Set minimum memory threshold to 12288KB
Warning: wrong zone alignment (0x90080000, 0x0000000c, 0x00001000)
zone(1): 0 pages.
zone(2): 0 pages.
```

预留内存大小，在节点 0 上总共 20 页, zone(0) 设置最小内存为 12MB, zone(1)和 zone(2)为 0 页。警告：对齐不正确

```
Kernel command line: root=/dev/mtdblock3
```

Kernel 启动命令设为：/dev/mtdblock3（在后面的说明中会看到 mtdblock3 是指的 flash 上的 romfs 分区。），用来指定根文件系统所在的位置，kernel 会将块设备 mtdblock3 当作文件系统来处理。

也就是说，内核会根据上面的 kernel 命令行，知道只读文件系统 romfs 将是根文件系统 root fs。
start_kernel(void)中输出的上面的这句信息。
这行命令是在 linux 内核启动过程中都会输出的一句。

```
Console: colour dummy device 80x30
```

代码中 console_init()的输出信息, 显示控制台属性：一般使用 VGA text console，标准是 80 X 25 行列的文本控制台，这里是对属性进行了设置。

```
serial_xx: setup_console @ 115
```

串口设置值为 115200，此为波特率输出信息。对串口设置的信息做一个打印的动作，在调试时会非常有用。

```
Calibrating delay loop... 82.94 BogoMIPS
```

Calibrate:校准, 进入时延校准循环。检查 CPU 的 MIPS(每秒百万条指令)，Bogo 是 Bogus(伪)的意思。这里是对 CPU 进行一个实时测试，来得到一个大体的 MIPS 数值
Bogomips,是由 linus Torvalds 写的, 是 Linux 操作系统中衡量计算机处理器运行速度的一种尺度。提供这种度量的程序被称为 BogoMips，当启动计算机时，BogoMips 能显示系统选项是否处于最佳性能。
linux 内核中有一个函数 calibrate_delay（）,它可以计算出 cpu 在一秒钟内执行了多少次一个极短的循环，计算出来的值经过处理后得到 BogoMIPS 值
你可以将计算机的 bogomips 与计算机处理器的 bogomips 进行比较。Torvalds 称这个程序为 BogoMips 来暗示两台计算机间的性能度量是错误的，因为并非所有起作用因素都能被显示出来或被认可。尽管计算机基准中经常用到 MIPS，但环境的变化容易导致度量的错误。Bogomips 能测出一秒钟内某程序运行了多少次。
察看/proc/cpuinfo 文件中的最后一行也能得到这个数值。
上面这个输出，在所有的 linux 系统启动中都会打印出来。

进入内存初始化
mem_init(void), [arch/i386/mm/init.c]

```
Memory: 80MB = 80MB total
Memory: 76592KB available (1724K code, 2565K data, 72K init)
```

当前内存使用情况，将列出总的内存大小, 及分配给内核的内存大小:包括代码部分，数据部分，初始化部分,总共刚好 4M。请留意此处的内核的内存大小的各个值。

进入虚拟文件系统 VFS 初始化
vfs_caches_init()

```
Dentry cache hash table entries: 16384 (order: 5, 131072 bytes)
Inode cache hash table entries: 8192 (order: 4, 65536 bytes)
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer cache hash table entries: 4096 (order: 2, 16384 bytes)
Page-cache hash table entries: 32768 (order: 5, 131072 bytes)
```

名词：

① Dentry：目录数据结构
② Inode：i 节点
③ Mount cache：文件系统加载缓冲
④ buffer cache：内存缓冲区
⑤ Page Cache：页缓冲区

Dentry 目录数据结构(目录入口缓存),提供了一个将路径名转化为特定的 dentry 的一个快的查找机制,Dentry 只存在于 RAM 中；
i 节点(inode)数据结构存放磁盘上的一个文件或目录的信息，i 节点存在于磁盘驱动器上;存在于 RAM 中的 i 节点就是 VFS 的 i 节点，dentry 所包含的指针指向的就是它；
buffer cache 内存缓冲区，类似 kupdated，用来在内存与磁盘间做缓冲处理；
Page Cache 用来加快对磁盘上映像和数据的访问。
在内存中建立各个缓冲 hash 表，为 kernel 对文件系统的访问做准备。
VFS（virtual filesystem switch）虚拟文件切换目录树有用到类似这样的结构表。
上面的输出信息，在一般的 linux 启动过程中都会看到。

POSIX conformance testing by UNIFIX

conformance:顺应, 一致。即 POSIX 适应性检测。UNIFIX 是一家德国的技术公司，Linux 原本要基于 POSIX.1 的, 但是 POSIX 不是免费的, 而且 POSIX.1 证书相当昂贵. 这使得 Linux 基于 POSIX 开发相当困难. Unifix 公司(Braunschweig, 德国) 开发了一个获得了 FIPS 151-2 证书的 Linux 系统. 这种技术用于 Unifix 的发行版 Unifix Linux 2.0 和 Laser moon 的 Linux-FT。
在 2.6 的内核中就将上面的这句输出给拿掉了。

第二节：用户模式(user_mode)开始，start_kernel 结束

PCI: bus0: Fast back to back transfers disabled PCI: Configured XX as a PCI slave with 128MB PCI memory PCI: Each Region size is 16384KB PCI: Reserved memory from 0x10080000 to 0x15080000 for DMA and mapped to 0x12000000

设备的初始化 init()--->do_basic_init()--->pci_init()，初始化 PCI，检测系统的 PCI 设备。

Linux NET4.0 for Linux 2.4 Based upon Swansea University Computer Society NET3.039

英国威尔士，斯旺西大学的 NET3.039, TCP/IP 协议栈
此信息，在 linux 启动过程中都会出现。

Initializing RT netlink socket

对 Socket 的初始化，socket_init()，Netlink 一种路由器管理协议(linux-2.4.22\net\core\Rtnetlink.c，Routing netlink socket interface: protocol independent part。 其中 RT 是 route 路由的意思。这句输出是在 create 产生 rtnetlink 的 socket 套接字时的一个调试输出。)
此信息，在 linux 启动过程中都会出现。

Starting kswapd

启动交换守护进程 kswapd，进程 IO 操作例程 kpiod
kswapd 可以配合 kpiod 运行。进程有时候无事可做，当它运行时也不一定需要把其所有的代码和数据都放在内存中。这就意味着我们可以通过把运行中程序不用的内容切换到交换分

区来更好的是利用内存。大约每隔 1 秒，kswapd 醒来并检查内存情况。如果在硬盘的东西要读入内存，或者内存可用空间不足，kpiod 就会被调用来做移入/移出操作。kswapd 负责检查，kpiod 负责移动。

Journalled Block Device driver loaded

加载日志块设备驱动。

日志块设备是用来对文件系统进行日志记录的一个块设备。日志文件系统是在传统文件系统的基础上，加入文件系统更改的日志记录。

它的设计思想是：跟踪记录文件系统的变化，并将变化内容记录入日志。日志文件系统在磁盘分区中保存有日志记录，写操作首先是对记录文件进行操作，若整个写操作由于某种原因(如系统掉电)而中断，系统重启时，会根据日志记录来恢复中断前的写操作。在日志文件系统中，所有的文件系统的变化都被记录到日志，每隔一定时间，文件系统会将更新后的元数据及文件内容写入磁盘。在对元数据做任何改变以前，文件系统驱动程序会向日志中写入一个条目，这个条目描述了它将要做些什么，然后它修改元数据。

devfs: v1.12c (20020818) Richard Gooch (rgooch@atnf.csiro.au) devfs: boot_options: 0x1

Devfs 模块的输出信息。

设备文件系统 devfs，版本 1.12c，

pty: 256 Unix98 ptys configured

Pty 模块的输出信息，与控制台操作有关的设置。

将通过 devpts 文件系统使用 Unix98 PTYs，（Pseudo-ttys (telnet etc) device 是伪 ttys 设备的缩写。

① TTY(/dev/tty)是 TeleTYpe 的一个老缩写，为用户输入提供不同控制台的设备驱动程序。它的名字来源于实际挂接到 UNIX 系统的、被称为电传打字机(teletype)的终端。在 Linux 下，这些文件提供对虚拟控制台的支持，可以通过按＜Alt-F1＞到＜Alt-F6＞键来访问这些虚拟控制台。这些虚拟控制台提供独立的、同时进行的本地登录对话过程

② ttys(/dev/ttys)是计算机终端的串行接口。/dev/ttyS0 对应 MS-DOS 下的 COM 1。

使用 make dev 脚本 MAKEDEV 来建立 pty 文件。这样系统内核就支持 Unix98 风格的 pty 了。在进行 Telnet 登录时将要用到/dev/pty 设备。 pty 是伪终端设备，在远程登录等需要以终端方式进行连接，但又并非真实终端的应用程序中必须使用这种设备，如 telnet 或 xterm 等程序。Linux 2.2 以后增添了 UNIX98 风格的 Pty 设备，它使用一个新的文件系统(devpts 针对伪终端的文件系统)和一个克隆的设备 cloning device 来实现其功能。

linux-2.4.22\drivers\char\Pty.c, 在 devfs_mk_dir (NULL, "pts", NULL);时会输出上面的信息。

loop: loaded (max 8 devices)

加载返还块设备驱动，最多支持 8 个设备

8139too Fast Ethernet driver 0.9.27 eth0: RealTek RTL8139 at 0x60112000, 00:10:0d:42:a0:03, IRQ 14 eth0: Identified 8139 chip type 'RTL-8100B/8139D'

网卡驱动，基地址为：0x60112000， MAC 地址:00:10:0d:42:a0:03, 中断号：14
RTL8139 的接收路径设计成一个环形缓冲区（一段线性的内存，映射成一个环形内存）。当设备接收到数据时，数据的内容就保存在这个环形缓冲区内并更新下个存储数据的地址（第一个数据包的开始地址＋第一个数据包的长度）。设备会一直保留缓冲区内的数据直到整个缓冲区耗尽。这样，设备会再次重写缓冲区内起始位置的内容，就像一个环那样。
从 2.2 版内核升级到 2.4 版时, RTL-8139 支持模块已不再叫 rtl8139，而叫它 8139too，现在你再看到 8139too 就不会不明白它的来由了吧。

SCSI subsystem driver Revision: 1.00

USB 设备信息，USB 会被当做 SCSI 来处理。

mumk_register_tasklet: (1) tasklet 0x905bf9c0 status @0x9025e974

软中断信息输出。Tasklet 是在 2.4 中才出现，它是为了更好地利用多 CPU。

Probing XX Flash Memory

探测 XX 的闪存（Flash Memory）,"NOR NAND Flash Memory Technology"

Amd/Fujitsu Extended Query Table v1.3 at 0x0040 number of CFI chips: 1

AMD 与富士通合资设立的 Flash 供货商 Spansion。AMD 因获利不佳,已经退出 Flash 市场，后续由 Spansion 合资公司经营.主要生产 NOR 类型的 flash,特点是容量小，速度快。Spansion 商标的 flash，在我们开发中会经常看到。以后大家看到 Spansion 的芯片，就能了解到它和 AMD 还有富士通的来龙去脉了。
Common flash Interface (CFI)是指一个统一的 flash 访问接口，表示这种 flash 是这种接口类型的。

Using buffer write method

使用 flash 写缓冲方式
flash 提供了写 BUFFER 的命令来加快对 flash 上块的操作。对 Flash 擦除和写数据是很慢的。如果用写 BUFFER 的命令会快一点。据手册上说，会快 20 倍。Buffer Size ： 5 bytes 的 buffer 缓冲不是每个块都有，是整个 flash 只有一个 5 bytes 的 buffer，用写 BUFFER 命令对所有的块进行写操作，都要用同一个 buffer,写 Buffer 是主要检查 buffer 是否 available，其实 buffer 起缓冲作用，来提高工作效率。
比如某 flash 有 128 个 128K 字节块。允许用户对任意块进行字节编程和写缓冲器字节编程操作，每字节编程时间为 210μs；若采用写缓冲器字节编程方式，32 字节编程共需 218μs，每字节编程时间仅为 6.8μs。芯片的块擦除时间为 1s，允许在编程或块擦除操作的同时进行悬挂中断去进行读操作，待读操作完成后，写入悬挂恢复命令，再继续编程或块擦除。

Creating 3 MTD partitions on "XX mapped flash": 0x00000000-0x00020000 : "BootLoader" 0x00020000-0x00040000 : "Config" 0x00040000-0x01000000 : "Romfs"

此处为重要信息部分，需要特别留意。
在内存中映射过的 flash,创建三个 MTD 分区：
flash 上的内容将被映射到内存中的对应地址
前 128K 为 BootLoader--->0x00000000-0x00020000
接着的 128K 为系统配置信息 Config 存放的位置--->0x00020000-0x00040000
再后面的 16M - 2X128K 为 romfs 的存放处.--->0x00040000-0x01000000
上面的内容，大家可以根据前面的换算公式得到。

A> 编译的 bootloader 一般大小约 50K 左右；
B> 在此处就知道了配置信息 config 是放在第 2 分区中的；
C> 制作的 romfs 的大小,一般为 8M 或 10M 左右，所以能放得下；

嵌入式 Linux 内核的块设备驱动：
对于 linux 的根文件系统，目前有三种块设备的驱动可以选择，它们分别是：
a） Blkmem 驱动
b） MTD 驱动
c) RAM disk 驱动
Blkmem 驱动是专门为嵌入式 linux 开发的一种块设备驱动，它是嵌入式 linux 系统中最为古老和通用的块设备驱动。它原理相对简单但是配置比较复杂，需要根据你即的 Flash 的分区使用情况来修改代码。当然修改的结果是它可以对一些 NOR 型的 Flash 进行读写操作。不过目前支持的 Flash 类型不够多。如果新加入对一种 Flash 的支持需要作的工作量比较大。
Linux 的 MTD 驱动是标准 Linux 的 Flash 驱动。它支持大量的设备，有足够的功能来定义 Flash 的分区，进行地址映射等等。使用 MTD 你可以在一个系统中使用不同类型的 Flash。它可以将不同的 Flash 组合成一个线性的地址让你来使用。
在标准的 Linux 2.4 内核中 MTD 有一系列的选项，你可以根据个人系统的需要来选择，定制。
另外一种选择就是 RAM disk 驱动。在 PC 上它经常用于没有硬盘的 Linux 的启动过程。它和 Flash 没有直接的关系。不过当 Flash 上启动的是经过压缩的内核时。RAM disk 可以作为根文件系统。
MTD 驱动提供了对 Flash 强大的支持,你通过它甚至可以在 Flash 上运行一个可以读写的真正的文件系统，比如 JFFS2。而 Blkmem 驱动则望尘莫及。

NET4: Linux TCP/IP 1.0 for NET4.0

调用 inet_init [linux-2.4.22\net\ipv4\Af_inet.c]时的输出信息，在启动过程中被 socket.c 调用到。

IP Protocols: ICMP, UDP, TCP, IGMP

列出可以支持的 IP 协议，此处为 kernel 源代码 inet_add_protocol(p);的输出。
在 linux 启动过程中，都会看到这句的输出。

```
IP: routing cache hash table of 512 buckets, 4Kbytes
```

IP 路由代码的输出信息。

ip_rt_init [linux-2.4.22\net\ipv4＼Route.c]，Set the IP module up，路由缓冲 hash 表

```
TCP: Hash tables configured (established 8192 bind 8192)
```

TCP 协议初始化输出信息。tcp_init [linux-2.4.22\net\ipv4\Tcp.c],

```
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
```

UNIX 网络协议信息。

af_unix_init[linux-2.4.22\net\unix\Af_unix.c], 多种连接的一种(IPv4, UNIX domain sockets, IPv6 和 IrDA). SMP 对称多处理器—Symmetrical Multi Processing，这里主要是指 UNIX 的一些网络协议.

上面的关于网络的输出信息是在 linux 启动信息中都会出现的。

加载各种文件系统

```
cramfs: wrong magic
```

会出现“cramfs: wrong magic”别担心这没有什么害处，这个是 kernel 的书写 bug，在 2.6 中有修改之，它是一个警告信息，用来检查 cramfs 的 superblock 超级块的。superblock 也是 VFS 要用到的数据结构。

代码 linux-2.4.22\fs\cramfs\Inode.c：

```
2.4
cramfs_read_super（。。。）
/* Do sanity checks on the superblock */
    if (super.magic != CRAMFS_MAGIC) {
        /* check at 512 byte offset */
        memcpy(&super, cramfs_read(sb, 512, sizeof(super)), sizeof(super));
        if (super.magic != CRAMFS_MAGIC) {
            printk(KERN_ERR "cramfs: wrong magic\n");
            goto out;
        }
    }
```

```
2.6
if (super.magic != CRAMFS_MAGIC) {
                if (!silent)
                    printk(KERN_ERR "cramfs: wrong magic\n");
                goto out;
            }
```

超级块是文件系统的“头部”它包含文件系统的状态、尺寸和空闲磁盘块等信息。如果损坏了一个文件系统的超级块（例如不小心直接将数据写到了文件系统的超级块分区中），那么系统可能会完全不识别该文件系统，这样也就不能安装它了，即使采用 e2fsck 命令也不能处理这个问题。

Cramfs 文件系统：

cramfs 是 Linus Torvalds 本人开发的一个适用于嵌入式系统的小文件系统。由于它是只读的，所以，虽然它采取了 zlib 做压缩，但是它还是可以做到高效的随机读取。 cramfs 不会影响系统读取文件的速度，又是一个高度压缩的文件系统。

我们制作 image 文件之后，我们还要考虑怎样才能在系统运行的时候，把这个 image 文件 mount 上来，成为一个可用的文件系统。由于这个 image 文件不是一个通常意义上的 block 设备，我们必须采用 loopback 设备来完成这一任务，如：

mount -o loop -t cramfs /usr.img /usr

这样，就可以经由 loopback 设备，把 usr.img 这个 cramfs 的 image 文件 mount 到 /usr 目录上去了。内核中需要对 loopback 这个设备的支持。

cramfs 的压缩效率一般都能达到将近 50%。

Cramfs 通过优化索引节点表的尺寸和除去传统文件系统中文件之间的空间浪费来达到节约空间的目的。它还使用了 zlib 压缩，实现优于 2:1 的压缩比例。解压缩过程的系统开销并不是很大，因为 Cramfs 支持指定单块的解压，而并不必解压缩整个文件。

Cramfs 不仅能节省空间，还能避免非正常关机导致的等待 fsck 或手工进行 fsck 的麻烦。它通过只读的方式达到这一目的。

RamDisk 有三种实现方式：

在 Linux 中可以将一部分内存 mount 为分区来使用，通常称之为 RamDisk，分为：Ramdisk, ramfs, tmpfs.

① 第一种就是传统意义上的，可以格式化，然后加载。

这在 Linux 内核 2.0/2.2 就已经支持，其不足之处是大小固定，之后不能改变。

为了能够使用 Ramdisk，我们在编译内核时须将 block device 中的 Ramdisk 支持选上，它下面还有两个选项，一个是设定 Ramdisk 的大小，默认是 4096k；另一个是 initrd 的支持。

如果对 Ramdisk 的支持已经编译进内核，我们就可以使用它了：

首先查看一下可用的 RamDisk，使用 ls /dev/ram*

首先创建一个目录，比如 test，运行 mkdir /mnt/test；

然后对/dev/ram0 创建文件系统，运行 mke2fs /dev/ram0；

最后挂载 /dev/ram0，运行 mount /dev/ram /mnt/test，就可以象对普通硬盘一样对它进行操作了。

② 另两种则是内核 2.4 才支持的，通过 Ramfs 或者 Tmpfs 来实现：

它们不需经过格式化，用起来灵活，其大小随所需要的空间而增加或减少。

Ramfs 顾名思义是内存文件系统，它处于虚拟文件系统（VFS）层，而不像 ramdisk 那样基于虚拟在内存中的其他文件系统(ex2fs)。

因而，它无需格式化，可以创建多个，只要内存足够，在创建时可以指定其最大能使用的内存大小。

如果你的 Linux 已经将 Ramfs 编译进内核，你就可以很容易地使用 Ramfs 了。创建一个目录，加载 Ramfs 到该目录即可：

mkdir /testRam

mount -t ramfs none /testRAM

缺省情况下，Ramfs 被限制最多可使用内存大小的一半。可以通过 maxsize（以 kbyte 为单位）选项来改变。

mount -t ramfs none /testRAM -o maxsize=2000 (创建了一个限定最大使用内存为 2M 的

ramdisk)

③ Tmpfs 是一个虚拟内存文件系统，它不同于传统的用块设备形式来实现的 Ramdisk，也不同于针对物理内存的 Ramfs。

Tmpfs 可以使用物理内存，也可以使用交换分区。在 Linux 内核中，虚拟内存资源由物理内存（RAM）和交换分区组成，这些资源是由内核中的虚拟内存子系统来负责分配和管理。

Tmpfs 向虚拟内存子系统请求页来存储文件，它同 Linux 的其它请求页的部分一样，不知道分配给自己的页是在内存中还是在交换分区中。同 Ramfs 一样，其大小也不是固定的，而是随着所需要的空间而动态的增减。

使用 tmpfs，首先你编译内核时得选择"虚拟内存文件系统支持（Virtual memory filesystem support）" 。

然后就可以加载 tmpfs 文件系统了：

```
# mkdir -p /mnt/tmpfs
# mount tmpfs /mnt/tmpfs -t tmpfs
```

同样可以在加载时指定 tmpfs 文件系统大小的最大限制：

```
# mount tmpfs /mnt/tmpfs -t tmpfs -o size=32m
```

FAT: bogus logical sector size 21072

具体的文件系统 FAT 格式。

虚拟逻辑扇区大小为 20K，linux-2.4.22\fs\fat\Inode.c。

在初始化 MS-DOS 文件系统时，读 MS-DOS 文件系统的 superblock，函数 fat_read_super 中输出的上面的信息。

UMSDOS: msdos_read_super failed, mount aborted.

UMSDOS:一种文件系统，特点容量大但相对而言不大稳定。是 Linux 使用的扩展了的 DOS 文件系统。它在 DOS 文件系统下增加了长文件名、UID/GID、POSIX 权限和特殊文件（设备、命名管道等)功能，而不牺牲对 DOS 的兼容性。允许一个普通的 msdos 文件系统用于 Linux，而且无须为它建立单独的分区，特别适合早期的硬盘空间不足的硬件条件。

VFS: Mounted root (romfs filesystem) readonly

虚拟文件系统 VFS(Virtual Filesystem Switch)的输出信息。

再次强调一下一个概念。VFS 是一种软件机制，也可称它为 Linux 的文件系统管理者，它是用来管理实际文件系统的挂载点,目的是为了能支持多种文件系统。kernel 会先在内存中建立一颗 VFS 目录树，是内存中的一个数据对象，然后在其下挂载 rootfs 文件系统，还可以挂载其他类型的文件系统到某个子目录上。

Mounted devfs on /dev

加载 devfs 设备管理文件系统到 dev 安装点上。

/dev 是我们经常会用到的一个目录。

在 2.4 的 kernel 中才有使用到。每次启动时内核会自动挂载 devfs。

devfs 提供了访问内核设备的命名空间。它并不是建立或更改设备节点，devfs 只是为你的特别文件系统进行维护。一般我们可以手工 mknod 创件设备节点。/dev 目录最初是空的，里面特定的文件是在系统启动时、或是加载模组后驱动程序载入时建立的。当模组和驱动程序卸载时，文件就消失了。

Freeing init memory: 72K

释放 1 号用户进程 init 所占用的内存。

第三节：加载 linux 内核完毕，转入 cpu_idle 进程

系统启动过程中进程情况：

①init 进程

一般来说, 系统在跑完 kernel bootstrapping 内核引导自举后(被装入内存、已经开始运行、已经初始化了所有的设备驱动程序和数据结构等等)， 就去运行 init『万 process 之父』，有了它, 才能开始跑其他的进程,因此，init 进程，它是内核启动的第一个用户级进程，它的进程号总是 1。

你可以用进程查看命令来验证

```
# ps aux
PID Uid VmSize Stat Command
1 0 SW init
2 0 SW [keventd]
3 0 SWN [ksoftirqd_CPU0]
4 0 SW [kswapd]
5 0 SW [bdflush]
6 0 SW [kupdated]
7 0 SW [rbwdg]
9 0 SW [mtdblockd]
10 0 SW [khubd]
80 0 SW [loop0]
```

另外 Linux 有两个 kernel 类的 process 也开始跑了起来，一个是 kflushd/bdflush，另一个是 kswapd；

只有这个 init 是完全属于 user 类的进程, 后两者是 kernel 假借 process 进程之名挂在进程上。

init 有许多很重要的任务，比如象启动 getty（用于用户登录）、实现运行级别、以及处理孤立进程。

init 一开始就去读 /etc/inittab (init 初始化表)，初始化表是按一定格式排列的关于进程运行时的有关信息的。init 程序需要读取/etc/inittab 文件作为其行为指针。这个 inittab 中对于各个 runlevel 运行级别要跑哪些 rc 或 spawn 生出什么有很清楚的设定。

一般, 在 Linux 中初始化脚本在/etc/inittab 文件(或称初始化表)中可以找到关于不同运行级别的描述。inittab 是以行为单位的描述性（非执行性）文本，每一个指令行都是固定格式 inittab 中有 respawn 项，但如果一个命令运行时失败了，为了避免重运行的频率太高，init 将追踪一个命令重运行了多少次，并且如果重运行的频率太高，它将被延时五分钟后再运行。

② kernel 进程

A> 请注意 init 是 1 号进程，其他进程 id 分别是 kflushd/ bdflush, kupdate, kpiod and kswapd。这里有一个要指出的：你会注意到虚拟占用（SIZE）和实际占用（RSS）列都是 0，进程怎么会不使用内存呢？

这些进程就是内核守护进程。大部分内核并不显示在进程列表里。守护进程在 init 之后启动，所以他们和其他进程一样有进程 ID，但是他们的代码和数据都存放在内核占有的内存中。在列表中使用中括号来区别与其他进程。

B> 输入和输出是通过内存中的缓冲来完成的，这让事情变得更快，程序的写入会存放在内存缓冲中，然后再一起写入硬盘。守护进程 kflushd 和 kupdate 管理这些工作。kupdate 间断的工作（每 5 秒）来检查是否有写过的缓冲，如过有，就让 kflushd 把它们写入磁盘。
C> 进程有时候无事可做，当它运行时也不一定需要把其所有的代码和数据都放在内存中。这就意味着我们可以通过把运行中程序不用的内容切换到交换分区来更好的是利用内存。把这些进程数据移入/移出内存通过进程 IO 管理守护进程 kpiod 和交换守护进程 kswapd，大约每隔 1 秒，kswapd 醒来并检查内存情况。如果在硬盘的东西要读入内存，或者内存可用空间不足,kpiod 就会被调用来做移入/移出操作。
D> bdflush - BUF_DIRTY, 将 dirty 缓存写回到磁盘的核心守护进程。 对于有许多脏的缓冲区（包含必须同时写到磁盘的数据的缓冲区）的系统提供了动态的响应。它在系统启动的时候作为一个核心线程启动，它叫自己为“kflushd”，而这是你用 ps 显示系统中的进程的时候你会看得的名字。即定期（5 秒）将脏（dirty）缓冲区的内容写入磁盘，以腾出内存；
E> ksoftirqd_CPUx 是一个死循环，负责处理软中断的。它是用来对软中断队列进行缓冲处理的进程。当发生软中断时，系统并不急于处理，只是将相应的 cpu 的中断状态结构中的 active 的相应的位，置位，并将相应的处理函数挂到相应的队列,然后等待调度时机来临,再来处理.
ksoftirqd_CPUx 是由 cpu_raise_softirq()即 cpu 触发中断，唤醒的内核线程，这涉及到软中断，ksoftirqd 的代码参见 [kernel/softirq.c]
F> keventd，它的任务就是执行 scheduler 调度器队列中的任务，keventd 为它运行的任务提供了可预期的进程上下文。
G> khubd，是用来检测 USB hub 设备的，当 usb 有动态插拔时，将交由此内核进程来处理。在检测到有 hub 事件时会有相应的动作（usb_hub_events()）
H> mtdblockd 是用来对 flash 块设备进行写操作的守护进程。
NAND 类型的 Flash 需要 MTD(Memory Technology Devices 内存技术驱动程序)驱动的支持才能被 linux 所使用。
NAND 的特点是不能在芯片内执行（XIP，eXecute In Place），需要把代码读到系统 RAM 中再执行，传输效率不是最高，最大擦写次数量为一百万次，但写入和擦除的速度很快，擦除单元小，是高数据存储密度的最佳选择。
NAND 需要 I/O 接口，因此使用时需要驱动程序。
I> loop0 是负责处理 loop 块设备的(回环设备)。loopback device 指的就是拿文件来模拟块设备，在我们这里，loop 设备主要用来处理需要 mount 到板上的文件系统,类似 mount /tmp/rootfs /mnt -o loop。.我们的实例有:mount -o loop -t cramfs /xxx.bin /xxx 也就是将 xxx.bin 这个文件 mount 到板上来模拟 cramfs 压缩 ram 文件系统。loop0 进程负责对 loop 设备进行操作。
loopback 设备和其他的块设备的使用方法相同。特别的是，可以在该设备上建立一个文件系统，然后利用 mount 命令把该系统映射到某个目录下以便访问。这种整个建立在一个普通磁盘文件上的文件系统，就是虚拟文件系统 (virtual file system)。

总结：
上面的内容是本人为了在实际开发中更加清楚地了解嵌入式 linux 的启动过程而做的一个总结性的文章。
在对嵌入式 linux 的启动过程做了一个详细注释后，大家会对涉及到嵌入系统的各个概念有了一个更加明确的认识，并能对嵌入系统的软硬件环境的有关设置更加清楚。当你自己动手

结合 linux 源代码来分析时，将会有一个清楚的全局观。
现在，你如果再回头去看文章前面所列出的启动信息例子中的内容，其中 80%的内容，你现在应该能看懂它的来龙去脉了。