Change History

Version	Date	Editor	Comments
0.1	Jan 2014	Bean Huo	Initial Release

Table of Contents

1. Linux 中堆、栈是怎么分的，增长方向

在 Linux 内核中堆和栈是不同的，堆是用户申请时产生，而栈是系统在运行时动态分配的。栈的增长方向是从高地址向低地址，而堆正好相反。同时对于栈也要分用用户态的和内核态的。在内核的栈的，kernel module 代码运行在当前进程的内核态，使用的是当前进程的内核栈，内核栈的大小有限制的，在内核编译时要进行选择，是 4096 还是 8192。

2. 内核的入口

Linux 非压缩内核的入口位于文件/arch/arm/kernel/head-armv.S 中的 stext 段。该段的基地址就是压缩内核解压后的跳转地址。如果系统中加载的内核是非压缩的 Image，那么 bootloader 将内核从 Flash 中拷贝到 RAM 后将直接跳到该地址处，从而启动 Linux 内核。不同体系结构的 Linux 系统的入口文件是不同的，而且因为该文件与具体体系结构有关，所以一般均用汇编语言编写。对基于 ARM 处理的 Linux 系统来说，该文件就是 head-armv.S。该程序通过查找处理器内核类型和处理器类型调用相应的初始化函数，再建立页表，最后跳转到 start_kernel()函数开始内核的初始化工作。检测处理器内核类型是在汇编子函数__lookup_processor_type 中完成的。通过以下代码可实现对它的调用。从 uboot 到内核是直接跳转过去的。

_start()为在 linux 内核中的入口函数，是在 head.s 中

3. uboot 如何可以给 kenel 传一个 wifi 的 mac 地址参数。

详见我的另一篇文章。

4. usb 接到电脑后，设备的增加过程

待续….

5. nand 与 nor flash 区别

NOR flash 带有 SRAM 接口，有足够的地址引脚来寻址，可以很容易地存取其内部的每一个字节。

NAND 器件使用复杂的 I/O 口来串行地存取数据，各个产品或厂商的方法可能各不相同。8 个引脚用来传送控制、地址和数据信息。

Nor flash，有类似于 dram 之类的地址总线，因此可以直接和 CPU 相连，CPU 可以直接通过地址总线对 nor flash 进行访问，而 Nand Flash 没有这类的总线，只有 IO 接口，只能通过 IO 接口发送命令和地址，对 Nand Flash 内部数据进行访问。

相比之下，nor flash 就像是并行访问，Nand Flash 就是串行访问，所以相对来说，前者的速度更快些。

同时 norflash 可以 XIP，也就是片内执行。NAND 是不行的。

NOR 的传输速率很高，但擦除和写入很慢。NAND 正好相反。
NOR 和 NAND 在写之前都要先擦再写，擦也就是把片内的所有位都置 1。

擦除的单位是块，通常一个块为 128KB.

对于写入两者有很大的不同，NOR 可以单个字节写入，而 NAND 必须是以页为单位来写。每个页面通常为 2112 个字节（2048 个数据字节+64 个空闲字节）。

NandFlash 的读写都是面向页的，而擦除则是面向块的，每个块由多个页面构成，如大块 Flash 结构下，每个块由 64 个页面组成，每个页面大小为 2112（2048+64）字节；小块 Flash 结构下，每个块由 32 个页面组成，每个页面大小为 528（512+16）字节，这每个页面中多出来的 64 或者 16 字节通常专门用于 ECC 校验字节。

6. 内核中有几种申请内存的方式，区别是什么

void *kmalloc(size_t size, gfp_t flags)

kmalloc 是内核中最常用的一种内存分配方式，它通过调用 kmem_cache_alloc 函 数来实现。kmalloc 一次最多能申请的内存大小由 include/Linux/Kmalloc_size.h 的 内容来决定，在默认的 2.6.18 内核版本中，kmalloc 一 次最多能申请大小为 131702B 也就是 128KB 字 节的连续物理内存。测试结果表明，如果试图用 kmalloc 函数分配大于 128KB 的内存，编译不能通过。

void *vmalloc(unsigned long size)

前面几种内存分配方式都是物理连续的，能保证较低的平均访问时间。但是在某些场合中，对内存区的请求不是很频繁，较高的内存访问时间也 可以接受，这是就可以分配一段线性连续，物理不连续的地址，带来的好处是一次可以分配较大块的内存。图 3-1 表 示的是 vmalloc 分配的内存使用的地址范围。vmalloc 对 一次能分配的内存大小没有明确限制。出于性能考虑，应谨慎使用 vmalloc 函数。在测试过程中， 最大能一次分配 1GB 的空间。

void * ioremap (unsigned long offset, unsigned long size)

ioremap 是一种更直接的内存“分配”方式，使用时直接指定物理起始地址和需要分配内存的大小，然后将该段 物理地址映射到内核地址空间。ioremap 用到的物理地址空间都是事先确定的，和上面的几种内存 分配方式并不太一样，并不是分配一段新的物理内存。ioremap 多用于设备驱动，可以让 CPU 直接访问外部设备的 IO 空间。ioremap 能映射的内存由原有的物理内存空间决定，所以没有进行测试。

```
static inline void *kzalloc(size_t size, gfp_t flags)
```

这个其实是调用 kmalloc,只是可以把申请的空间再处始化为用户指定的值。

7. __init 与__exit 的作用

__init， __initdata 等属性标志，是要把这种属性的代码放入目标文件的.init.text 节，数据放入.init.data 节——这一过程是通过编译内核时为相关目标平台提供了 xxx.lds 链接脚本来指导 ld 完成的。

对编译成 module 的代码和数据来说，当模块加载时，__init 属性的函数就被执行；
对静态编入内核的代码和数据来说，当内核引导时，do_basic_setup()函数调用 do_initcalls()函数，后者负责所有.init 节函数的执行。

在初始化完成后，用这些关键字标识的函数或数据所占的内存会被释放掉__init 和__exit 标记函数，__initdata 和__exitdata 标记数据。

此宏定义可知标记后的函数与数据其实是放到了特定的（代码或数据）段中。

标记为初始化的函数,表明该函数供在初始化期间使用。

在模块装载之后，模块装载就会将初始化函数扔掉。这样可以将该函数占用的内存释放出来。

__exit 修饰词标记函数只在模块卸载时使用。

如果模块被直接编进内核则该函数就不会被调用。如果内核编译时没有包含该模块,则此标记的函数将被简单地丢弃。

8. linux 中出错定位有哪几种办法

用 printf("%s(%d)-%s: this is main\n",__FILE__,__LINE__,__FUNCTION__);只要包含 stdio.h 就可以

用 gdb printk.

9. linux 内核启动地址 0x30008000 由来

一般情况下都在生成 vmlinux 后，再对内核进行压缩成为 zImage，压缩的目录是 kernel/arch/arm/boot。下载到 flash 中的是压缩后的 zImage 文件，zImage 是由压缩后的 vmlinux 和解压缩程序组成. 查看 2410 的 datasheet ,发现内存映射的基址是 0x3000 0000 ,那么 0x30008000 又是如何来的呢？在内核文档 kernel/Document/arm/Booting 文件中可以看到，在 image 下面用了 32K (0x8000) 的空间存放内核页表，0x30008000 就是 2410 的内核在 RAM 中的启动地址。

10. 内核中申请内存有一些简单的规则：

1，判断申请内存的时候可否睡眠，也就是调用 kmalloc 的时候能否被阻塞。如果在一个中断处理，在中断处理的下半部分，或者有一个锁的时候，就不能被阻塞。如果在一个进程上下文，也没有锁，则一般可以睡眠。

2，如果可以睡眠，指定 GFP_KERNEL。

3，如果不能睡眠，就指定 GFP_ATOMIC。

4，如果需要 DMA 可以访问的内存，比如 ISA 或者有些 PCI 设备，就需要指定 GFP_DMA。

5，需要对 kmalloc 返回的值检查 NULL。

6，为了没有内存泄漏，需要用 kfree()来释放内存

11. export 与 extern

export symbol 就是定义全局变量，全模块都可以使用。

extern 是声明一个在外部已经定义过的全局变量

12. 字节对齐有几种，怎么区分

大端：在这种格式中，字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中

小端：与大端存储格式相反，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节

请写一个 C 函数，若处理器是 Big_endian 的，则返回 0；若是 Little_endian 的，则返回 1
解答：int checkCPU( )

```
{
{
           union w
           {
                  int  a;
                  char b;
           } c;
           c.a = 1;
           return(c.b ==1);
    }
}
```

联合体 union 的存放顺序是所有成员都从低地址开始存放。

```
int  big_endian (void)
     {
       union{
          long l;
          char c[sizeof(long)];
       }u;

       u.l = 1;
       return  (u.c[sizeof(long) - 1] == 1);
     }
```

下面这段代码可以用来测试一下你的编译器是大端模式还是小端模式：

```
short int x;
char x0,x1;
x=0x1122;
x0=((char*)&x)[0];  //低地址单元
x1=((char*)&x)[1];  //高地址单元
```

若 x0=0x11,则是大端；若 x0=0x22,则是小端.

13. C 和 c++中的 extern "C"

通常，在 C 语言的头文件中经常可以看到类似下面这种形式的代码：

```
#ifdef __cplusplus
extern "C" {
#endif
  /**** some declaration or so *****/
  #ifdef __cplusplus
}
#endif
```

这是为了让 CPP 能够与 C 接口而采用的一种语法形式。之所以采用这种方式，是因为两种语言之间的一些差异所导致的。由于 CPP 支持多态性，也就是具有相同函数名的函数可以完成不同的功能，CPP 通常是通过参数区分具体调用的是哪一个函数。在编译的时候，CPP 编译器会将参数类型和函数名连接在一起，于是在程序编译成为目标文件以后，CPP 编译器可以直接根据目标文件中的符号名将多个目标文件连接成一

个目标文件或者可执行文件。但是在 C 语言中，由于完全没有多态性的概念，C 编译器在编译时除了会在函数名前面添加一个下划线之外，什么也不会做（至少很多编译器都是这样干的）。由于这种的原因，当采用 CPP 与 C 混合编程的时候，就可能会出问题。假设在某一个头文件中定义了这样一个函数：

```
int foo(int a, int b);
```

而这个函数的实现位于一个.c 文件中，同时，在.cpp 文件中调用了这个函数。那么，当 CPP 编译器编译这个函数的时候，就有可能会把这个函数名改成_fooii ，这里的 ii 表示函数的第一参数和第二参数都是整型。而 C 编译器却有可能将这个函数名编译成_foo 。也就是说，在 CPP 编译器得到的目标文件中，foo() 函数是由_fooii 符号来引用的，而在 C 编译器生成的目标文件中，foo() 函数是由_foo 指代的。但连接器工作的时候，它可不管上层采用的是什么语言，它只认目标文件中的符号。于是，连接器将会发现在.cpp 中调用了 foo() 函数，但是在其它的目标文件中却找不到_fooii 这个符号，于是提示连接过程出错。extern "C" {} 这种语法形式就是用来解决这个问题的。

14. ARM-linux 启动分几部分，简述流程

15. 如何让一个 IIC 驱动支持多个 device

16. 内核态与用户态进程通信方式有几种

系统调用 \ PROC 文件系统 \ device driver ioctls\netlink\mmap

Linux 系统下用户空间与内核空间数据交换的九种方式，包括内核启动参数、模块参数与 sysfs、sysctl、系统调用、netlink、procfs、seq_file、debugfs 和 relayfs

17. 进程间通信（IPC）机制就是为实现应用与应用之间的数据交换而专门实现，一共有几种，你所用到的是哪几种？

18. USB 驱动设备注册过程

USB 设备都是先注册驱动，当总线发现有新的 USB 设备时，会读取 USB 设备中的 ID，与每个 USB 驱动的中注册的 ID 比较，比对成功自动创建设备并与驱动及总线结构对象相关联

19. uC/OS-II、eCos、Linux 的简要比较

uC/OS-II： 占先式内核，仅支持 bitmap 调度算法，最多支持 65 任务线程，提供比较完善的线程同步服务。开源但非免费，需要开发商业产品的用户，需要购买 license。开发环境，没有限制，对于软件开发来讲仅仅相当于一个函数库。

ecos ：占先式内核，支持 bitmap 调度算法和同优先级分时调度算法，支持 POSIX 标准接口，比较完善的线程同步服务。有自己的功能很强大的 bootloader（redboot：支持在线调试程序，更新程序和内核等），提供很多厂商出的 BSP。network、file system 等各种模块齐全

在 Linux 中，进程的运行时间不可能超过分配给他们的时间片，他们采用的是抢占式多任务处理，所以进程之间的挂起和继续运行无需彼此之间的协作。

20. 硬实时和软实时区别

硬实时与软实时之间最关键的差别在于，软实时只能提供统计意义上的实时。例如，有的应用要求系统在95%的情况下都会确保在规定的时间内完成某个动作，而不一定要求 100%。在许多情况下，这样的“软性”正确率已经可以达到用户期望的水平。比如，用户在操作 DVD 播放机时，只要 98%的情况都能正常播放，用户可能就满意了;而发射卫星、控制核反应堆的应用系统，这些系统的实时性必须达到 100%，是绝对不允许出现意外。在实时操作系统中，系统必须在特定的时间内完成指定的应用，具有较强的“刚性”，而分时操作系统则注重将系统资源平均地分配给各个应用，不太在意各个应用的进度如何，什么时间能够完成。不过，就算是实时操作系统，其“刚性”和“柔性”的程度也有所不同，就好像是系统的“硬度”有所不同，因而有了所谓的“硬实时(hard real-time)”和“软实时 (soft real-time)”。硬实时系统有一个刚性的、不可改变的时间限制，它不允许任何超出时限的错误。超时错误会带来损害甚至导致系统失败、或者导致系统不能实现它的预期目标。软实时系统的时限是一个柔性灵活的，它可以容忍偶然的超时错误。失败造成的后果并不严重，例如在网络中仅仅是轻微地降低了系统的吞吐量。

21. 驱动中申请内存的方法与比较

Kmalloc kmalloc 能够分配的内存块的大小有一个上限. 这个限制随着体系和内核配置选项而变化. 如果你的代码是要完全可移植, 它不能指望可以分配任何大于 128 KB.

kmalloc 最大只能开辟 128k-16, 16 个字节是被页描述符结构占用了。kmalloc 特殊之处在于它分配的内存是物理上连续的,这对于要进行 DMA 的设备十分重要. 而用 vmalloc 分配的内存只是线性地址连续,物理地址不一定连续,不能直接用于 DMA.

进程的 4GB 内存空间被人为的分为两个部分--用户空间与内核空间。用户空间地址分布从 0 到 3GB(PAGE_OFFSET, 在 0x86 中它等于 0xC0000000), 3GB 到 4GB 为内核空间。

内核空间中，从 3G 到 vmalloc_start 这段地址是物理内存映射区域（该区域中包含了内核镜像、物理页框表 mem_map 等等），比如我们使用的 VMware 虚拟系统内存是 160M，那么 3G～3G+160M 这片内存就应该映射物理内存。在物理内存映射区之后，就是 vmalloc 区域。对于 160M 的系统而言，vmalloc_start 位置应在 3G+160M 附近（在物理内存映射区与 vmalloc_start 期间还存在一个 8M 的 gap 来防止跃界），vmalloc_end 的位置接近 4G(最后位置系统会保留一片 128k 大小的区域用于专用页面映射)。 kmalloc 和 get_free_page 申请的内存位于物理内存映射区域，而且在物理上也是连续的，它们与真实的物理地址只有一个固定的偏移，因此存在较简单的转换关系，virt_to_phys()可以实现内核虚拟地址转化为物理地址。

22. uboot 的启动流程？nand flash 如何启动(CPU 支持才行，(CPU 内部有 4K 的 sram)?

像 2410 可以配制成从片外 nandflash 启动，这时因为 2410 一上电，可以把 nand 中的前 4K 的数据搬到 2410 的 4k ram 中去制行。

23. 没有 jtag 和仿真器时，如可定位代码运行到哪里了？

(GPIO 写 LED，或直接读写串口)

24. UBOOT 怎么添加新的命令

25. UBOOT 是如何收发网络包的？（轮询）

26. 主机 PING 不通 UBOOT 是什么原因？

(不支持 ICMP_ECHO_REAUEST)

27. 在 UBOOT 中怎么添加新的启动命令行中的参数？

(_setup(),启动时与.init.setup 段中依次匹配)

28. UBOOT 是怎么把控制权交给 linux 的？

(直接跳转)

29. NAPI 是怎么会事？

30. 什么样的驱动可以做成模块？模块中的 probe 函数是怎么运行的

31. linux 下的设备和驱动分别是怎么匹配的？

(独立遍历和匹配)

32. PCI 设备是怎么初始化的？PCI 的配置空间怎么读写？

(pci_read_config_byte())

33. 如何确定 PCI 设备中某个 BAR 的类型？

(BAR 的最后一位为 1 表示 IO 模式)

34. 如何确定 PCI 设备中某个 BAR 的其大小与长度?

(先写全 1，再读，非零位即长度)

35. percpu 变量是怎么存储的？

(.data.percpu 数据段，数组 core[3]:(a,b,c),再是 core1[3](a,b,c),不是 a[2],b[2],c[2])

36. dts 文件中的 interrupt 是怎么定义的？irq_mapping 是做什么用的？

37. 用 PCI 向网卡读写数据时有什么注意事项？

（如何保证 cache 的一致性？）

38. 内核的上半部和下半部分的根本区别是什么？

（下半底开中断）

39. 下半部分实现机制有几种？主要区别是什么?

(软中断、工作队列、tasklet)

40. kmalloc 和 vmalloc 的区别？

（线性存储空间，物理地址；非线性，虚拟地址）

41. 如何分配 2MB，20MB,200MB 的内核空间？

（vmalloc,get_free_pages,基数）

42. EXPORT_SYMBOL 和 extern 的区别？

（全局内核空间，本链接域）

43. EXPORT_SYMBOLR 的原理，能否导出同一名称的变量？

（模块和内核都有_ksymtab 段，不能同名，因为是全局的）

44. 中断服务程序为什么不能被睡眠？

因为再无法调度回来,

45. 中断服务程序如果确实需要做长时间的任务，怎么办

中断线程化

46. spin_lock 与 semaphore 的区别?

前者死等，后者产生调度。

47. spin_lock 、spin_lock_bh,spin_lock_irq 分别用在什么场合？

48. 为什么 dma 会导致 cache 不一致？

CPU 先访问 cache，hit 后就不访问内存；而设备是直接操作总线地址的

49. 读写时如何避免 cache 不致？

读前使 cache 无效，写前刷新内存。

50. linux 下中断可以嵌套吗？

不同的中断可以自由嵌套，同种中断不能嵌套执行。

51. linux 内核与用户空间的通信机制有哪些？

系统调用、/proc,/sys,ioctl,mmap 共享内存，netlink.

52. 内核的入口函数/地址是怎么确定的？

链接文件 lnk

53. __init,__exit 起什么作用？

初始化完后，退出时释放内存。

54. 如何定位某个回调函数的 caller？

Dump_stack(),__builtin_return_addr()

55. 一个可执行文件有哪些段？

.text,.data,.bss

56. MMU,TLB,PageTable 的关系？

MMU 是执行单元，PageTable 在内存上，TLB 是存在于 L1、L2 上的部分 PageTable

57. CPU 如何管理 MAC 和 PHY 的？

CPU 通过 PCI 管理 MAC，通过 SMI 管理 PHY

58. #define min(x,y)?原理？

编译时的 warning

59. 802.1Q 即 TagVLAN 的以太网帧的格式？

DstMac,SrcMac,Tag,IPType,Payload,FCS

60. 1 个 8GHZ 远比 8 个 1Ghz 有效率，为什么还有多核。

无法作到 8G 的 core

61. tcpdump 和 traceroute 命令的原理？

二层抓包，ICMP 中的 TTL 递增

62. I2C 设备和 PHY 设备的地址是怎么确定的？

硬件设计好的

63. 要用两个不同种类的 I2C 器件，（比如 RTC 和温度传感器），但地址相同，怎么办？

用不同的 I2C 总线或者用 MUX

64. 嵌入式 linux 文件系统有哪些，各咱的区别是什么

RAMDisk 必须把 rootfs 从 flash 中 load 到 ram 中去，它是事读写的。
Cramfs 是一个只读的，他是可以直接在 flash 中运行的。
Linux 中，rootfs 是必不可少的。PC 上主要实现有 ramdisk 和直接挂载 HD（Harddisk，硬盘）上的根文件系统；嵌入式中一般不从 HD 启动，而是从 Flash 启动，最简单的方法是将 rootfs load 到 RAM 的 RAMDisk，稍复杂的就是 直接从 Flash 读取的 Cramfs，更复杂的是在 Flash 上分区，并构建 JFFS2 等文件系统。

* RAMDisk 将制作好的 rootfs 压缩后写入 Flash，启动的时候由 Bootloader load 到 RAM，解压缩，然后挂载到 /。这种方法操作简单，但是在 RAM 中的文件系统不是压缩的，因此需要占用许多嵌入式系统中稀有资源 RAM。

ramdisk 就是用内存空间来模拟出硬盘分区,ramdisk 通常使用磁盘文件系统的压缩存放在 flash 中,在系统初始化时,解压缩到 SDRAM 并挂载根文件系统，在 linux 系统中,ramdisk 有二种,一种就是可以格式化并加载,在 linux 内核 2.0/2.2 就已经支持,其不足之处是大小固定;另一种是 2.4 的内核才支持,通过,ramfs 来实现,他不能被格式化,但用起来方便,其大小随所需要的空间增加或减少,是目前 linux 常用的 ramdisk 技术.

* initrd 是 RAMDisk 的格式，kernel 2.4 之前都是 image-initrd，Kernel 2.5 引入了 cpio-initrd，大大简化了 Linux 的启动过程，附合 Linux 的基本哲学：Keep it simple, stupid(KISS). 不过 cpio-initrd 作为新的格式，还没有经过广泛测试，嵌入式 Linux 中主要采用的还是 image-initrd。

* Cramfs 是 Linus 写的很简单的文件系统，有很好的压缩绿，也可以直接从 Flash 上运行，不须 load 到 RAM 中，因此节约了 RAM。但是 Cramfs 是只读的，对于需要运行时修改的目录（如： /etc, /var, /tmp）多有不便，因此，一般将这些目录做成 ramfs 等可写的 fs。

* SquashFS 是对 Cramfs 的增强。突破了 Cramfs 的一些限制，在 Flash 和 RAM 的使用量方面也具有优势。不过，据开发者介绍，在性能上可能不如 Cramfs。这也是一种新方法，在嵌入式系统采用之前，需要经过更多的测试

65. YAFFS 与 JFFS 比较

YAFFS， Yet Another Flash File System，是一种类似于 JFFS/JFFS2 的专门为 Flash 设计的嵌入式文件系统。与 JFFS 相比，它减少了一些功能，因此速度更快、占用内存更少。 YAFFS 和 JFFS 都提供了写均衡，垃圾收集等底层操作。它们的不同之处在于：

1 ）、 JFFS 是一种日志文件系统，通过日志机制保证文件系统的稳定性。 YAFFS 仅仅借鉴了日志系统的思想，不提供日志机能，所以稳定性不如 JFFS ，但是资源占用少。

2 ）、 JFFS 中使用多级链表管理需要回收的脏块，并且使用系统生成伪随机变量决定要回收的块，通过这种方法能提供较好的写均衡，在 YAFFS 中是从头到尾对块搜索，所以在垃圾收集上 JFFS 的速度慢，但是能延长 NAND 的寿命。

3 ）、 JFFS 支持文件压缩，适合存储容量较小的系统； YAFFS 不支持压缩，更适合存储容量大的系统。

YAFFS 还带有 NAND 芯片驱动，并为嵌入式系统提供了直接访问文件系统的 API ，用户可以不使用 Linux 中的 MTD 和 VFS ，直接对文件进行操作。 NAND Flash 大多采用 MTD+YAFFS 的模式。 MTD （ Memory Technology Devices ，内存技术设备）是对 Flash 操作的接口，提供了一系列的标准函数，将硬件驱动设计和系统程序设计分开。

YAFFS2 是 YAFFS 的升级版，能更好的支持 NAND FLASH 。

JFFS 是由瑞典的 Axis Communications Ab 公司开发的(1999,以 GNU 发布),针对 flash 设备的特性为嵌入式设备开发的.(我边上的兄弟曾想去那里作毕业设计)

JFFS1 和 JFFS2 的设计中都考虑到了 FLASH 的特性特别是满足了上述 3 个条件,包括了垃圾回收,坏块管理等功能. 这两种文件系统属于 LFS(Log-structured File System).这种文件系统的特点是一旦数据出错,容易恢复,但是系统运行是需要占用一定的内存空间,这些空间就是用来存储”log”的.

JFFS 的缺点就是加载时间太长,因为每次加载都需要将 FLASH 上的所有节点(JFFS 的存储单位)到内存,这样也占用了可观的内存空间.除此之外,”circle log”设计使得在对文件数据进行所有的数据都会被重写,这样造成不必要的时间,同时也会减少 FLASH 的寿命.

JFFS2 对 JFFS1 作了些改进,比如所需的内存变少了,垃圾回收机制也优化了.

针对 JFFS1,JFFS2 的缺点,JFFS3 出现了.

YAFFS1 & YAFFS2

“Yet Another Flash File System”作者是新西兰的 Charles Manning 为一家名叫 Alpha one 的公司(http://www.aleph1.co.uk/)设计的,是第一个为 NAND Flash 设计的文件系统.共两个版本 YAFFS1 和 YAFFS2.

YAFFS1 支持 512Bytes/Page 的 NAND Flash;后者 YAFFS2 支持 2kBytes/Page 的 NAND Flash. YAFFS 文件系统也属于 LFS.

跟其他文件系统比较,它具有更好的可移植性,甚至可以使用在没有操作系统的设备上(called “YAFFS/Direct”). YAFFS 采用模块化设计,虽然最初是用在 linux 系统上的,但是也已经移植到其他系统比如 wince.

还有个突出的优点是它在 mount 的时候需要很少的内存.(如果是小页-512byte/page,每 1MByte NAND 大约需要 4KBytes 内存;大页需要大概 1KBytes RAM/1MByte NAND)

JFFS 与 YAFFS 比较,两者各有长处. 一般来说,对于小于 64MBytes 的 NAND Flash,可以选用 JFFS;如果超过 64MBytes,用 YAFFS 比较合适.

http://www.yaffs.net/yaffs-internals

http://www.yaffs.net/yaffs-direct-user-guide

66. mov 与 ldr 区别：

数据从内存到 CPU 之间的移动只能通过 L/S 指令来完成，也就是 ldr/str 指令。
比如想把数据从内存中某处读取到寄存器中，只能使用 ldr
比如：
ldr r0, 0x12345678
就是把 0x12345678 这个地址中的值存放到 r0 中。
而 mov 不能干这个活，mov 只能在寄存器之间移动数据，或者把立即数移动到寄存器中，这个和 x86 这种 CISC 架构的芯片区别最大的地方。
x86 中没有 ldr 这种指令，因为 x86 的 mov 指令可以将数据从内存中移动到寄存器中。

另外还有一个就是 ldr 伪指令，虽然 ldr 伪指令和 ARM 的 ldr 指令很像，但是作用不太一样。ldr 伪指令可以在立即数前加上=，以表示把一个地址写到某寄存器中，比如：
ldr r0, =0x12345678
这样，就把 0x12345678 这个地址写到 r0 中了。所以，ldr 伪指令和 mov 是比较相似的。只不过 mov 指令限制了立即数的长度为 8 位，也就是不能超过 512。而 ldr 伪指令没有这个限制。如果使用 ldr 伪指令时，后面跟的立即数没有超过 8 位，那么在实际汇编的时候该 ldr 伪指令是被转换为 mov 指令的。

ldr 伪指令和 ldr 指令不是一个同东西。

67. linux 系统时钟,Tick、Hz、jiffies 关系

一般由 tick、Hz 表示，Hz 表示每秒要发生时钟中断的次数，一般设为 100,表示 1 秒钟发生 100 次时钟中断，Tick 是 Hz 的倒数，表示每过多少时间分发生一次中断，如 Hz 是 100，则 Tick 为 10Ms.而 jiffies 则是则是在每一次中断中会加 1。
Vxworks 中也有系统时钟，默认值为 60,但我们一般设备为 100,与 linux 相用。