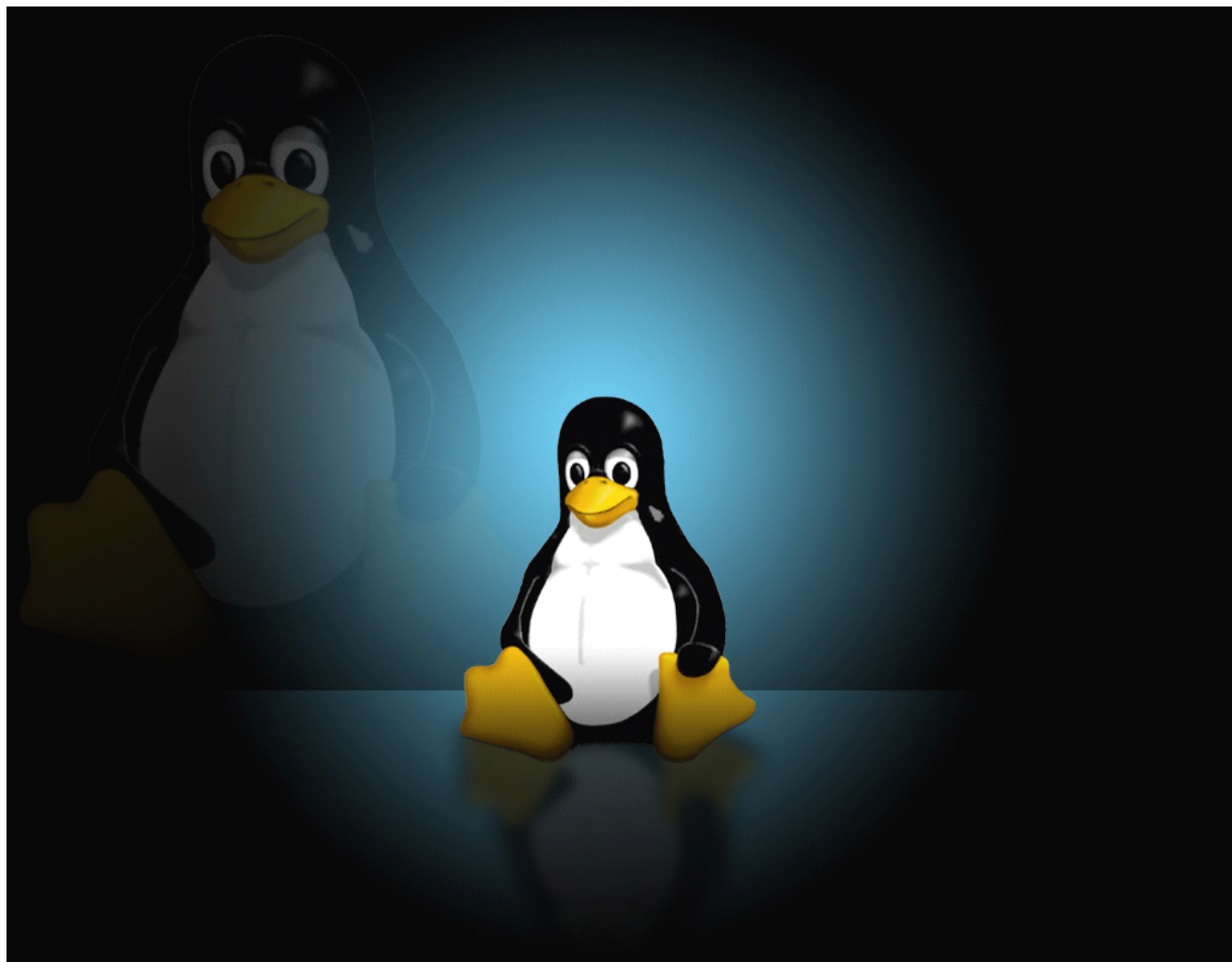


Open Source , Open your life.



<资料整合第二辑>

2010.04.05

大家有好的资料或在网上看到好的文章, 请发往 yuanyangliu258@gmail.com , 我替大家整理
同时也希望有同学愿意加入这样的开源行列, 大家一起学习, 一起讨论, 一起为开源事业奋斗。

有意者邮件联系

申明: 本资料均转载网络, 有删节, 请忽用于商业用途

ARM7 与 ARM9 的区别.....	5
ARM9 与 ARM7 的比较及优化.....	6
ARM9E 处理器的存储器子系统.....	7
性能和效率的提升.....	9
ARM-Linux 根文件系统的制作.....	12
• 目录.....	12
• Shell.....	13
• 库.....	14
• 脚本.....	16
***** 《基本步骤总结》 *****.....	18
ARM 汇编程序中常见的伪指令分析.....	22
ARM 体系各种异常分析.....	28
1.复位异常.....	28
2.IRQ 异常.....	28
3.FIR 异常.....	29
4.未定义指令异常.....	29
5.中止异常.....	29
6.SWI 软件中断异常.....	30
ARM 伪指令讲解.....	30
符号定义 (Symbol Definition) 伪指令.....	30
数据定义 (Data Definition) 伪指令.....	32
汇编控制 (Assembly Control) 伪指令.....	34
其他常用的伪指令.....	36

C 语言最大难点揭秘.....	40
C 语言中文件输入输出函数.....	48
C 语言高效编程的几招.....	55
C 语言宏定义学习.....	59
1 防止一个头文件重复定义.....	59
2 防止由于各种平台和编译器的不同, 而产生的类型字节数.....	62
3 得到指定地址上的一个字节或字.....	62
4 求最大值和最小值.....	63
5 得到一个 field 在结构体(struct)中的偏移量.....	63
6 得到一个结构体中 field 所占用的字节数.....	64
7 按照 LSB 格式把两个字节转化为一个 Word.....	64
8 按照 LSB 格式把一个 Word 转化为两个字节.....	66
Nand Flash 与 Nor Flash.....	67
Nand Flash 读写操作.....	70
The ASCII Chart.....	77
嵌入式 Linux 启动脚本整理.....	80
识别复杂变量的声明.....	85
交叉编译环境搭建.....	88
1. 选定软件版本号.....	88
2. 建立工作目录.....	89
3. 输出和环境变量.....	90
4. 建立编译目录.....	92
建立内核头文件.....	92
建立二进制工具 (binutils)	95

建立初始编译器 (bootstrap gcc)	97
建立 c 库(glibc)	99
建立全套编译器 (full gcc)	101
开发板上移植可执行程序.....	103
1、解包 qtopia-free-1.7.0.tar.gz.....	103
2、在\$QPEDIR 目录下交叉编译:	104
3、建立应用启动器 (.desktop) 文件.....	104
4、建立根文件系统.....	105
内核编译安装过程.....	105
《安装过程》	105
一些升级内核前的备份过程:	107
怎么给内核打补丁?	109
嵌入式经典面试题目录.....	109
预处理器 (Preprocessor)	110
死循环 (Infinite loops)	110
Static.....	112
Const.....	112
Volatile.....	113
位操作 (Bit manipulation)	114
中断 (Interrupts)	115
Typedef.....	116
清理 Linux 老内核.....	118
如何做嵌入式 Linux 操作系统.....	120
移植 U-Boot 到 S3C2440.....	155

1. U-Boot 移植.....	155
1.1 移植准备.....	155
1.1.1 添加开发板的配置选项.....	156
1.1.2 在/board 子目录中建立自己的开发板 mini2440 目录.....	156
1.1.3 在 include/configs/中建立配置头文件.....	157
1.1.4 测试编译能否成功.....	157
1.2 修改 U-Boot 中的文件, 以同时匹配 2440 和 2410.....	157
1.2.1 修改/cpu/arm920t/start.S.....	157
1.2.2 添加 Nand Flash 读取函数.....	164
1.2.3 修改 board/mini2440/lowlevel_init.S 文件.....	167
1.2.4 修改 GPIO 和 PLL 的配置.....	168
1.2.5 修改 u-boot 支持烧写 yaffs 映像文件.....	170
1.2.6 修改 DM9000 驱动.....	175
1.2.7 Nand Flash 驱动.....	175
1.2.8 修改 u-boot-2008.10/cpu/arm920t/s3c24x0/interrupts.c 文件.....	179
1.2.9 修改 cpu/arm920t/s3c24x0/speed.c , 修改根据时钟寄存器来计算时钟的方法。	180
1.2.10 修改 u-boot-2008.10/drivers/usb/usb_ohci.c , 添加对 2440 的支持.....	181
1.2.11 添加 mini2440 的机器 ID.....	182
1.2.12 修改 u-boot-2008.10/include/configs/mini2440.h 头文件.....	182
1.2.13 修改 u-boot-2008.10/lib_arm/board.c 文件.....	186
1.2.14 搜索以下文件, 把支持 S3C2410 的宏定义改成同时支持 S3C2410 和 S3C2440.....	187

ARM7 与 ARM9 的区别

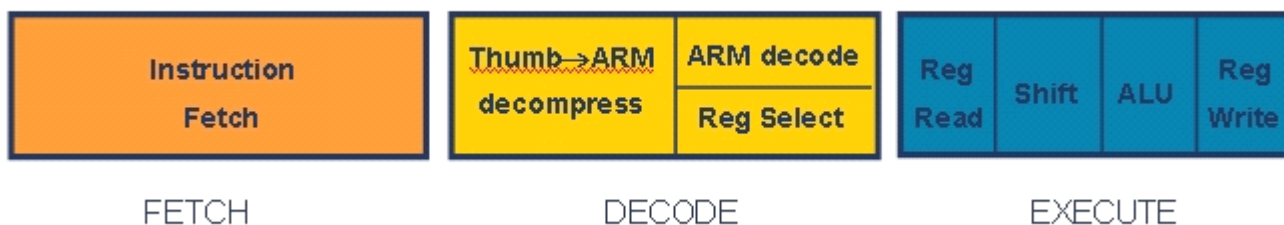
有朋友问我 ARM7 有 ARM9 的区别, 在此进行解答。ARM7 内核是 0.9MIPS/MHz 的三级流水线和冯 • 诺伊曼结构; ARM9 内核是 5 级流水线, 提供 1.1MIPS/MHz 的哈佛结构。arm7 没有 mmu, arm720T 是 MMU 的; arm9 是有 mmu 的, arm940T 只有 Memory protection unit. 不是一个完整的 MMU。ARM7TDMI 提供了非常好的性能 – 功耗比。它包含了 THUMB 指令集快速乘法指令和 ICE 调试技术的内核。ARM9 的时钟频率比 ARM7 更高, 采用哈佛结构区分了数据总线和指令总线。

ARM9 与 ARM7 的比较及优化

目前越来越多的嵌入式系统设计开始采用 ARM9 处理器。理解 ARM9 与 ARM7 的差别, 以及如何针对 ARM9 进行系统优化, 成为了一个令人关注的话题。本文通过对 ARM9 处理器的特点介绍, 介绍针对 ARM9 处理器进行系统优化的一些有效方法。随着更多应用在嵌入式系统中的实现, 嵌入式系统设计向着更高级、更复杂的方向发展。作为 32 位结构体系中的翘楚, ARM 在各种应用领域里得到了极其广泛的应用, 成为目前国内电子设计领域里面的焦点之一(2005 年一项针对国内嵌入式系统开发者的调查表明, 有 63% 的工程师把 ARM 作为 32 位 CPU 的首选)。过去几年里, 绝大部分 ARM 系统都是基于 ARM7 处理器, 最近一年里, 基于 ARM9 处理器的产品越来越多, 研究 ARM9 的特点以及如何优化从 ARM7 到 ARM9 的移植, 成为很多嵌入式系统设计者所关注的热点问题。我们惯称的 ARM9 系列中又存在 ARM9 与 ARM9E 两个系列, 其中 ARM9 属于 ARM v4T 架构, 典型处理器如 ARM9TDMI 和 ARM922T; 而 ARM9E 属于 ARM v5TE 架构, 典型处理器如 ARM926EJ 和 ARM946E。因为后者的芯片数量和应用更为广泛, 所以我们提到 ARM9 的时候更多地是特指 ARM9E 系列处理器(主要就是 ARM926EJ 和 ARM946E 这两款处理器)。下面关于 ARM9 的介绍也是更多地集中于 ARM9E。ARM7 处理器和 ARM9E 处理器的流水线差别 对嵌入式系统设计者来说, 硬件通常是第一考虑的因素。针对处理器来说, 流水线则是硬件差别的最明显标志, 不同的流水线设计会产生一系列硬件差异。让我们来比较一下 ARM7 和 ARM9E 的流水线, 如图 1。可以看到 ARM9E 从 ARM7 的 3 级流水线增加到了 5 级, ARM9E 的流水线中容纳了更多的逻辑操作, 但是每一级的逻辑操作却变得更为简单。比如原来 ARM7 的第三级流水, 需要先内部读取寄存器、然后进行相关的逻辑和算术运算, 接着处理结果回写, 完成的动作非常复杂; 而在 ARM9E 的 5 级流水中, 寄存器读取、逻辑运算、结果回写分散在不同的流水当中, 使得每一级流水处理的动作非常简洁。这就使得处理器的主频可以大幅度地提高。因为每一级流水都对应 CPU 的一个时钟周期, 如果一级流水中的逻辑过于复杂, 使得执行时间居高不下

下, 必然导致所需的时钟周期变长, 造成 CPU 的主频不能提升。所以流水线的拉长, 有利于 CPU 主频的提高。在常用的芯片生产工艺下, ARM7 一般运行在 100MHz 左右, 而 ARM9E 则至少在 200MHz 以上。

ARM7的3级流水线



ARM9E的5级流水线

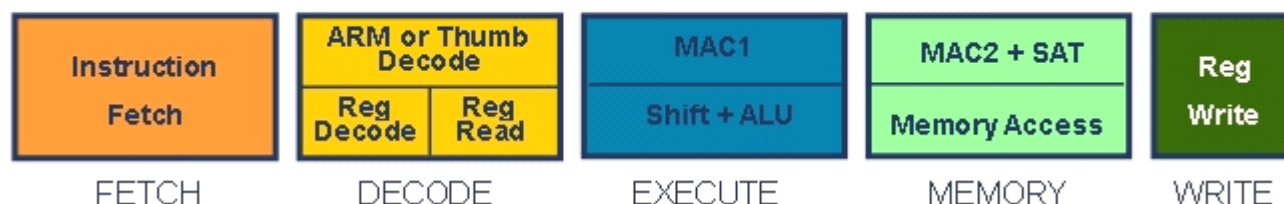


图 1: ARM7 的三级流水线与 ARM9E 的五级流水线。

ARM9E 处理器的存储器子系统

像 ARM926EJ 和 ARM946E 这两个最常见的 ARM9E 处理器中, 都带有一套存储器子系统, 以提高系统性能和支持大型操作系统。如图 2 所示, 一个存储器子系统包含一个 MMU(存储器管理单元)或 MPU(存储器保护单元)、高速缓存(Cache)和写缓冲(Write Buffer); CPU 通过该子系统与系统存储器系统相连。高速缓存和写缓存的引入是基于如下事实, 即处理器速度远远高于存储器访问速度; 如果存储器访问成为系统性能的瓶颈, 则处理器再快也是浪费, 因为处理器需要耗费大量的时间在等待存储器上面。高速缓存正是用来解决这个问题, 它可以存储最近常用的代码和数据, 以最快的速度提供给 CPU 处理(CPU 访问 Cache 不需要等待)。

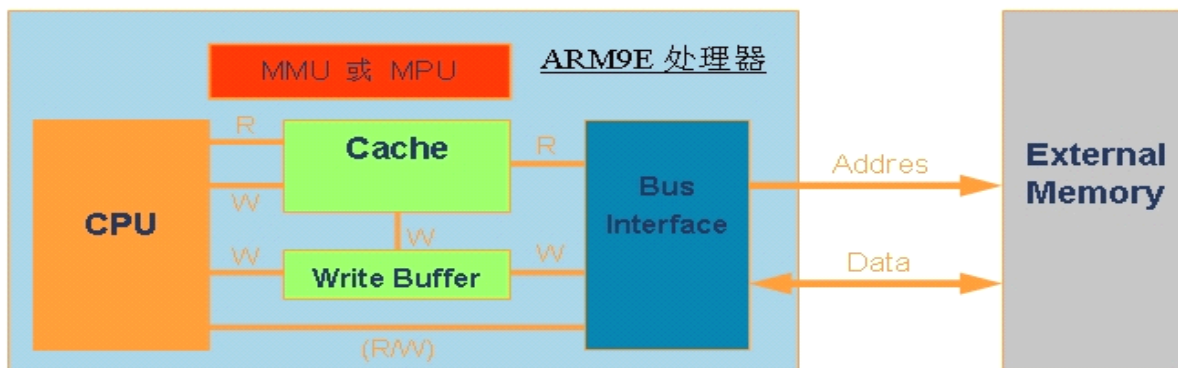


图 2: 复杂处理器内部的存储器子系统。

MMU 则是用来支持存储器管理的硬件单元, 满足现代平台操作系统内存管理的需要; 它主要包括两个功能: 一是支持虚拟/物理地址映射, 二是提供不同存储器地址空间的保护机制。一个简单的例子可以帮助我们理解 MMU 的功能, 如图 3, 在一个操作系统下, 程序开发人员都是在操作系统给定的 API 和编程模型下开发程序; 操作系统通常只开放一个确定的存储器地址空间给用户。这样就带来一个直接的问题, 所有的应用程序都使用了相同的存储器地址空间, 如果这些程序同时启动的话(在现在的多任务系统中这是非常常见的), 就会产生存储器访问冲突。那操作系统是如何来避免这个问题的呢? 操作系统会利用 MMU 硬件单元完成存储器访问虚拟地址到物理地址的转换。所谓虚拟地址就是程序员在程序中使用的逻辑地址, 而物理地址则是真实存储器单元的空间地址。MMU 通过一定的规则, 可以把相同的虚拟地址映射到不同的物理地址上去。这样, 即使有多个使用相同虚拟地址的程序进程启动, 也可以通过 MMU 调度把它们映射到不同的物理地址上去, 不会造成系统错误。

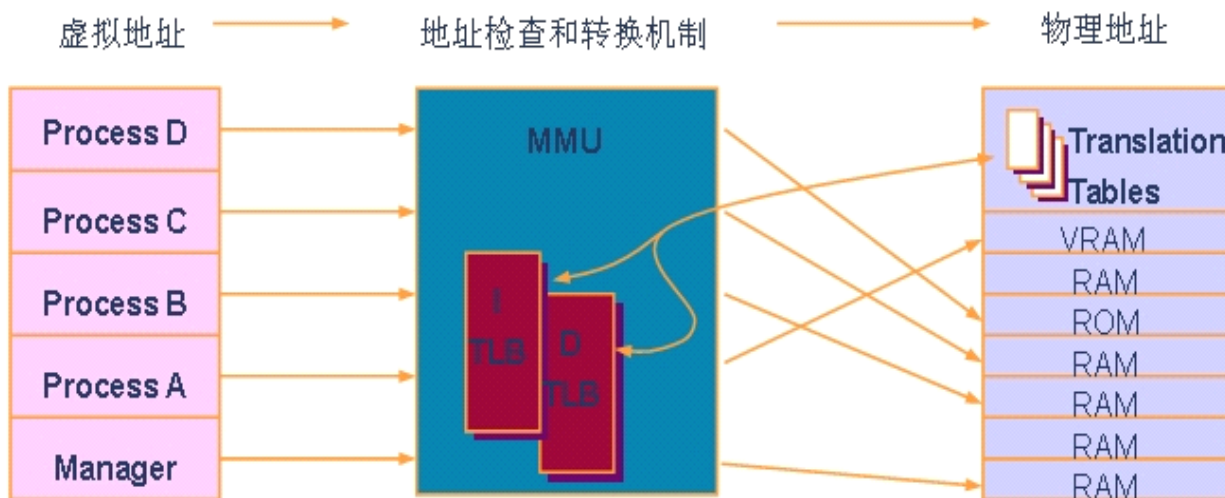


图 3: MMU 的功能和作用。

MMU 处理地址映射功能之外, 还能给不同的地址空间设置不同的访问属性。比如操作系统把自己的内核程序地址空间设置为用户模式下不可访问, 这样的话用户应用程序就无法访问到该空间, 从而保证操作系统内核的安全性。MPU 与 MMU 的区别在于它只有给地址空间设置访问属性的功能而没有地址映射功能。Cache 以及 MMU 等硬件单元的引入, 给系统程序员的编程模型带来了许多全新的变化。

除了需要掌握基本的概念和使用方法之外, 下面几个针对系统优化的点既有趣又重要:

1、系统实时性考虑 因为保存地址映射规则的页表(Page Table)非常庞大, 通常 MMU 中只是存储器了常用的一小段页表内容, 大部分页表内容都存储于主存储器里面; 当调用新的地址映射规则时, MMU 可能需要读取主存储器来更新页表。这在某些情况下会造成系统实时性的丢失。比如当需要执行一段关键的程序代码时, 如果不巧这段代码使用的地址空间不在当前 MMU 的页表处理范围里面, 则 MMU 首先需要更新页表, 然后完成地址映射, 接着才能相应存储器访问; 整个地址译码过程非常长, 给实时性带来非常大的不利影响。所以一般来说带 MMU 和 Cache 的系统在实时性上不如一些简单的处理器; 不过也有一些办法能够帮助提高这些系统的实时效率。一个简单的办法是在需要的时候关闭 MMU 和 Cache, 这样就变成一个简单处理器了, 可以马上提高系统实时性。当然很多情况下这不可行; 在 ARM 的 MMU 和 Cache 设计中, 有一个锁定的功能, 就是说你可以指定某一块页表在 MMU 中不会被更新掉, 某一段代码或数据可以在 Cache 中锁定而不会被刷新掉; 程序员可以利用这个功能来支持那些实时性要求最高的代码, 保证这些代码始终能够得到最快的响应和支持。

2、系统软件优化 在嵌入式系统开发中, 很多系统软件优化的方法都是相同和通用的, 多数情况下这种规则也适用于 ARM9E 架构上。如果你已经是一个 ARM7 的编程高手, 那么恭喜你, 以前你掌握的优化方法完全可以用在新的 ARM9E 平台上, 但是会有一些新的特性需要你加倍注意。最重要的便是 Cache 的作用, Cache 本身并不带来编程模型和接口的变化, 但是如果我们考察 Cache 的行为, 就能够发现对于软件优化, Cache 是有比较大的影响的。

Cache 在物理上就是一块高速 SRAM, ARM9E 的 Cache 组织宽度(cache line)都是 4 个 word(也就是 32 个字节); Cache 的行为受系统控制器控制而不是程序员, 系统控制器会把最近访问存储器地址附近的内容复制到 Cache 中去, 这样, 当 CPU 访问下一个存储器单元的时候(这个访问既可能是取指, 也可能是数据), 可能这个存储器单元的内容已经在 Cache 里了, 所以 CPU 不需要真的到主存储器上去读取内容, 而直接读取 Cache 高速缓存上面的内容就可以了, 从而加快了访问的速度。从 Cache 的工作原理我们可以看到, 其实 Cache 的调度是基于概率的, CPU 要访问的数据既可能在 Cache 中已经存在(Cache hit), 也可能没有存在(Cache miss)。在 Cache miss 的情况下, CPU 访问存储器的速度会比没有 Cache 的情况更坏, 因为 CPU 除了要从存储器访问数据以外, 还需要处理 Cache hit 或 miss 的判断, 以及 Cache 内容的刷新等动作。只有当 Cache hit 带来的好处超过 Cache miss 带来的牺牲的时候, 系统的整体性能才能得到提高, 所以 Cache 的命中率成为一个非常重要的优化指标。根据 Cache 行为的特点, 我们可以直观地得到提高 Cache 命中率的一些方法, 如尽可能把功能相关的代码和数据放置在一起, 减少跳转次数; 跳转经常会引起 Cache miss。保持合适的函数大小, 不要书写太多过小的函数体, 因为线性的程序执行流程是最为 Cache 友好的。循环体最好放置在 4 个 word 对齐的地址, 这样就能保证循环体在 Cache 中是行对齐的, 并且占用最少的 Cache 行数, 使得被多次调用的循环体得到更好的执行效率。

性能和效率的提升

前面介绍了 ARM9E 相比于 ARM7 性能上的提高，这不仅表现在 ARM9E 有更快的主频、更多的硬件特性上面，还体现在某些指令的执行效率上面。执行效率我们可以用 CPU 的时钟周期数(Cycle)来衡量；运行同一段程序，ARM9E 的处理器可以比 ARM7 节省大约 30%左右的时钟周期。效率的提高主要来自于 ARM9E 对于 Load-Store 指令执行效率的增强。我们知道在 RISC 架构的处理器中，程序中大约有 30%的指令是 Load- Store 指令，这些指令的效率对系统效率的贡献是最明显的。

ARM9E 中有两个因素帮助提高 Load-Store 指令的效率： 1)ARM9 内核是哈佛架构，拥有独立的指令和数据总线；相对应，ARM7 内核是指令和數據总线复用的冯·诺依曼架构。 2)ARM9 的 5 级流水线设计把存储器访问和寄存器写回放在不同的流水上面。 两者结合，使得在指令流的执行过程中每个 CPU 时钟周期都可以完成一个 Load 或 Store 指令。下面的表格比较了 ARM7 和 ARM9 处理器之间的 Load -Store 指令。从中可以看出所有的 Store 指令 ARM9 比 ARM7 省 1 个周期，Load 指令可以省 2 个周期(在没有互锁的情况下，编译工具能够通过编译优化消除大多数的互锁可能)。

表 1： ARM7 与 ARM9E 的 Load-Store 指令执行周期数比较。

指令	ARM7		ARM9	
	执行周期数	可能的互锁周期	执行周期数	可能的互锁周期
LDR 读一个 word	3	0	1	0 or 1
LDM 读 n 个 word	n + 2	0	n	0 or 1
STR 写一个 word	2	0	1	0
STM 写 n 个 word	n + 1	0	n	0

综合各种因素，ARM9E 处理器拥有非常强大的性能。但是在实际的系统设计中，设计人员并不总是把处理器性能开到最大，理想情况是把处理器和系统运行频率降低，使得性能刚好能满足应用需求；达到节省功耗和成本的目的。在评估系统能够提供的处理器能力过程中，DMIPS 指标被很多人采用；同时它也被广泛应用于不同处理器间的性能比较。 但是用 DMIPS 来衡量处理器性能存在很大的缺陷。DMIPS 并非字面上每秒百万条指令的意思，它是一个测量 CPU 运行一个叫 Dhrystone 的测试程序时表现出来的相对性能高低的一个单位(很多场合人们也习惯用 MIPS 作为这个性能指标的单位)。因为基于程序的测试容易受到恶意优化的干扰，并且 DMIPS 指标值的发布不受任何机构的监督，所以使用 DMIPS 进行评估时要慎重。例如对 Dhrystone 测试程序进行不同的编译处理，在同一个处理器上运行也可以得出差别很大的结果，如图 4 中是 ARM926EJ 在 32 位 0 等待存储器上运行测试程序的结果。ARM 一直采用比较保守的值作为 CPU 的 DMIPS 标称值，如 ARM926EJ 是 1.1DMPS/MHz。

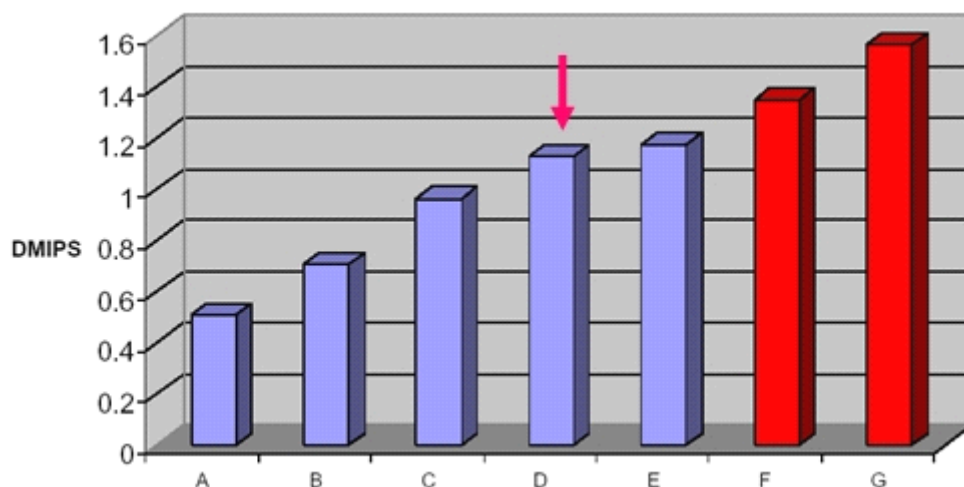


图4: 不同测试条件下 ARM926EJ 处理器的 DMIPS 值。

DMIPS 另外一个缺点是不能测量处理器的数字信号处理能力和 Cache/MMU 子系统的性能。因为 Dhrystone 测试程序不包含 DSP 表达式, 只包含一些整型运算和字符串处理, 并且测试程序偏小, 几乎可以完整地放在 Cache 里面运行而无需与外部存储器进行交互。这样就难以反映处理器在一个真实系统中的真正性能。一种值得鼓励的评估方法是站在系统的角度看问题, 而不仅仅拘泥于 CPU 本身; 而系统性能评估最好的测试向量就是用户应用程序或相近的测试程序, 这是用户所需的最真实的结果。ARM9E 处理器的 DSP 运算能力 伴随应用程序的多样化和复杂化, 诸如多媒体、音视频功能在嵌入式系统里面也是全面开花。这些应用需要相当的 DSP 处理能力; 如果是在传统的 RISC 架构上实现这些算法, 所需的资源(频率和存储器等)会非常不经济。ARM9E 处理器一个非常重要的优势就是拥有轻量级的 DSP 处理能力, 以非常小的成本(CPU 增加功能需要增加硬件)换来了非常实用的 DSP 性能。因为 CPU 的 DSP 能力并不直接反映在像 DMIPS 这样的评测指标中, 同时像以前的 ARM7 处理器中也没有类似的概念; 所以这一点对所有使用 ARM9E 处理器进行开发的人来说, 都是需要注意的一个要点。

表 2: ARM9E 的 DSP 扩展指令表。

指令	操作	用处
SMLAxy{cond}	$16 \times 16 + 32 \rightarrow 32$	带符号 MAC
SMLAWy{cond}	$32 \times 16 + 32 \rightarrow 32$	带符号宽 MAC
SMLALxy{cond}	$16 \times 16 + 64 \rightarrow 64$	带符号长 MAC
SMULxy{cond}	$16 \times 16 \rightarrow 32$	带符号乘
SMULWy{cond}	$16 \times 32 \rightarrow 32$	带符号长乘
QADD Rd, Rm, Rs	SAT (Rm + Rd)	饱和加
QDADD Rd, Rm, Rs	SAT (Rm + SAT(Rs \times 2))	饱和连加
QSUB Rd, Rm, Rs	SAT (Rm - Rd)	饱和减
QDSUB Rd, Rm, Rs	SAT (Rm - SAT(Rs \times 2))	饱和连减
CLZ{cond} Rd, Rm	COUNTZ (Rm)	前导零计算

ARM9E 的 DSP 扩展指令如表2所示, 主要包括三个类型。

1) 单周期的16x16和32x16 MAC 操作, 因为数字信号处理中甚少32位宽的操作数, 在32位寄存器中可以对操作数分段运算显得非常有用。

2) 对原有的算术运算指令增加了饱和处理扩展, 所谓饱和运算, 就是当运算结果大于一个上限或小于一个下限时, 结果就等于上限或是下限; 饱和处理在音频数据和视频像素处理中普遍使用, 现在一条单周期饱和运算指令就能够完成普通 RISC 指令“运算-判断-取值”这一系列操作。

3) 前导零 (CLZ) 运算指令, 提高了归一化和浮点运算以及除法操作的性能。以流行的 MP3 解码程序为例。整个解码过程中前端的三个步骤是运算量最大的, 包括比特流的读入(解包)、霍夫曼译码还有反量化采样(逆变换)。ARM9E 的 DSP 指令正好可以高效地完成这些运算。以44.1 KHz@128 kbps 码率的 MP3 音乐文件为例, ARM7TDMI 需要占用20MHz 以上的资源, 而 ARM926EJ 则只要小于10MHz 的资源本文总结 在从 ARM7 到 ARM9 的平台转变过程中, 有一件事情是非常值得庆幸的, 即 ARM9E 能够完全地向后兼容 ARM7 上的软件; 并且开发人员面对的编程模型和架构基础也保持一致。但是毕竟 ARM9E 中增加了很多新的特性, 为了充分利用这些新的资源, 把系统性能优化好, 需要我们对 ARM9E 做更多深入地了解。

ARM-Linux 根文件系统的制作

来自--[EE 小站](#)

关于根文件系统的制作, 网络上有很多文章, 大多数都只讲到建几个目录, 然后用 Busybox 做个 Shell, 有很多关键的东西没有说。经过很长时间的摸爬滚打, 我终于能够白手起家建立一个根文件系统了。其实我也不懂得原理, 只是告诉大家我的作法, 其中也不免有错误, 欢迎大家指正。

首先介绍根文件系统的组成: 目录、Shell、库、脚本, 一个个来

• 目录

根文件系统要包含这些必须有的目录: /dev、/bin、/usr、/sbin、/lib、/etc、/proc、/sys

/dev 是 devfs (设备文件系统) 或者 udev 的挂在点所在。在使用 devfs 的内核里如果没有/dev, 根本见不到 Shell 启动的信息, 因为内核找不到/dev/console; 在使用 udev 的系统里, 也事先需要在/dev 下建立 console 和 null 这两个节点。关于 devfs 和 udev 的区别, 网上很多文章说。当然如果你的内核已经不支持 devfs 了 (2.6.12以后), 可以使用纯纯的静态节点。也就是用 mknod 人工生成。

/bin、/usr/bin、/usr/sbin、/sbin 是编译 Busybox 这个 Shell 时候就有的, 用于存放二进制可执行文件, 就不多解释了。

/lib 用于存放动态链接库。网上很多文章都说静态编译 Busybox, 可以省去建库的麻烦过程。这样做只能让 Busybox 启动, 我们自己写的, 或者是编译的软件包还是需要动态库的。除非全部静态编译, 你可以试试, 一个 Hello world 就要几百 k。关于库的内容后面仔细说。

/etc 是用来存放初始化脚本和其他配置文件的。关于初始化脚本的内容后面仔细说。

/proc 是用来挂载存放系统信息虚拟文件系统——“proc 文件系统”, “proc 文件系统”在内核里面可以选。如果没有“proc 文件系统”, 很多 Shell 自己的命令就没有办法运行, 比如 ifconfig。“proc 文件系统”不像 devfs 可以自动挂载, 它需要使用初始化脚本挂载。另外, udev 也需要“proc 文件系统”的支持。

/sys 用于挂载“sysfs 文件系统”, “sysfs 文件系统”在内核里面可以选。目前我认为它就是给 udev 提供支持的, 呵呵。“sysfs 文件系统”也需要使用初始化脚本挂载。

另外还可以有/tmp、/mnt、/swp、/var 这样的不是嵌入式系统必须的目录, 在说完 Shell 的制作之后, 我再谈建立目录的事情。

• Shell

Shell 很简单, 就是 Busybox, 上网下载一个来: <http://www.busybox.net/downloads/>。说 Busybox 和 arm-linux-gcc 有兼容性问题, 不过我觉得那是比较低版本的时代问题了, 我用 Busybox 1.8.2和 arm-linux-gcc 3.4.1/3.3.2都可以。解压缩以后找到 Makefile 里面的 ARCH 和 CROSS_COMPILE, 改成:

```
ARCH ?= arm
```

```
CROSS_COMPILE ?= /usr/local/arm/3.4.1/bin/arm-linux-
```


当然 CROSS_COMPILE 由你自己的编译器位置决定, 然后

```
# make menuconfig
```

```
# make
```

```
# make install
```

注意配置的时候把一些 uClinux Only 的东西去掉, 不然会错; 配置的时候还可以修改安装位置, 默认是在 Busybox 下的 “_install”。

之后就可以在 Busybox 生成的 Shell 基础上建立根文件系统了, 我就用命令来说吧, Busybox 在 /home/lxz/busybox, 根文件系统在/home/lxz/rootfs

```
# mkdir /home/lxz/rootfs
```

```
# cd /home/lxz/busybox/_install
```

```
# cp -r ./ /home/lxz/rootfs
```

```
# cd /home/lxz/rootfs
```

```
# mkdir dev
```

```
# mkdir etc
```

```
# mkdir lib
```

```
# mkdir proc
```

```
# mkdir sys
```

```
# mkdir tmp
```

如果不用 devfs, 下面的命令是必须的。必须以 root 用户执行 (用 su 命令可以切换为 root, 切换后用 exit 命令可以返回普通用户) :

```
# cd /home/lxz/rootfs/dev
```

```
# mknod -m 660 console c 5 1
```

```
# mknod -m 660 null c 1 3
```

如果不使用 devfs 没有这两个静态节点, console 的提示根本就看不到, 出现的现象可能是内核提示 Free init memory: XXK 之后, Warning: Unable to find a initial console 之类的, 具体的单词记得不是很准确。我没有试过使用 udev 的时候没有这两个静态节点的情况, 反正放了也不影响把/dev 挂载为 tmpfs。

如果使用 udev, 还需要把 udevd、udevstart、udevadmin 这三个文件放到/sbin 里面 (我使用 udev-117, 网上介绍较多的 udev-100有9个文件要放)。

•库

库可是一件非常麻烦的事情。我建议初学者拷贝买的开发板里面带的文件系统的库, 如果开发板的文件系统是映像, 只需要把映像挂载在某个目录下就可以访问, 假设映像叫做 rootfs.cramfs, 可以这样

```
# mkdir /home/lxz/evb_rootfs
(切换为 root 用户)
# mount -o loop rootfs.cramfs /home/lxz/evb_rootfs
(可以切换为普通用户)
# cd /home/lxz/evb_rootfs/lib
# cp -r ./ /home/lxz/rootfs/lib
```

一般开发板里都会带有很多库, 但是总体积却比较大。可以删掉一些不用的库来减小体积, 但是, 呵呵, 我也不知道那些库具体含有什么函数, 什么情况删什么; 也许以后我会把这部分补上。如果觉得库体积太大, 也可以自己编译 glibc 或者 uclibc, 但是这是非常繁琐的事情——目前我认为库应该和编译器 arm-linux-gcc 一起制作。有个傻瓜式的方案是使用 cross-tool, 下载地址:

<http://www.kegel.com/crosstool/>。虽然 cross-tool 是用来制作交叉编译器的, 但是其过程中生成的 glibc 却可以作为副产品为我们所用。cross-tool 的使用可以看我之前的这篇文章

<http://xianzilu.spaces.live.com/blog/cns!4201FDC93932DDAF!274.entry>。在成功制作了交叉编译器之后, 就可以从 cross-tool 的目录里把 glibc 取出来, 假设 cross-tool 的路径是 /home/lxz/cross-tool, 编译出的编译器叫做 arm-linux-gnu-gcc, gcc 版本 3.4.5, glibc 版本 2.3.6, 想要把 glibc 库拷贝到 /home/lxz/glibc, 下面的操作还是用命令来说明。

```
# cd /home/lxz/cross-tool/build/arm-linux-gnu-gcc/gcc-3.4.5-glibc-2.3.6/build-glibc
# ../glibc-2.3.6/configure --prefix=/home/lxz/glibc
# make install
```

等候安装结束

```
# cd /home/lxz/glibc
# cp -r lib /home/lxz/rootfs
```

这样就把 glibc 的大部队拷贝好了, 但是这样还缺两个库, 我们继续

```
# cd /home/lxz/cross-tool/build/arm-linux-gnu-gcc/gcc-3.4.5-glibc-2.3.6/build-gcc/gcc
# cp libgcc_s.so* /home/lxz/rootfs/lib
```

还缺少一个 libtermcap 库, 这个就稍微有些麻烦。libtermcap-2.0.8-35-armv4l 源码包的下载地址是 <http://www.netwinder.org/mirror/pub/netwinder/SRPMs/nw/9/libtermcap-2.0.8-35.src.rpm>, 你也可以

在这里 <http://www.netwinder.org/allsrpms.html> 找到其他版本的。假设 libtermcap-2.0.8-35.src.rpm 下载到了/home/lxz/libtermcap, 下面继续用命令说明。

```
# cd /home/lxz/libtermcap

# rpm2cpio libtermcap-2.0.8-35.src.rpm | cpio -ivd
# tar xvjf termcap-2.0.8.tar.bz2
```

接下来要打13个补丁, 很汗啊, 请一定按照下面的顺序来打补丁

```
# patch -p0 -i termcap-2.0.8-shared.patch
# patch -p0 -i termcap-2.0.8-setuid.patch
# patch -p0 -i termcap-2.0.8-instnroot.patch
# patch -p0 -i termcap-2.0.8-compat21.patch
# patch -p0 -i termcap-2.0.8-xref.patch
# patch -p0 -i termcap-2.0.8-fix-tc.patch
# patch -p0 -i termcap-2.0.8-ignore-p.patch
# patch -p0 -i termcap-buffer.patch
# patch -p0 -i termcap-2.0.8-bufsize.patch
# patch -p0 -i termcap-2.0.8-colon.patch
# patch -p0 -i libtermcap-aaargh.patch
# patch -p0 -i termcap-2.0.8-glibc22.patch
# patch -p0 -i libtermcap-2.0.8-ia64.patch
```

然后到/home/lxz/libtermcap/termcap-2.0.8里, 找到 Makefile, 修改其中的 CC 和 AR,

```
CC = /usr/local/arm/3.4.1/bin/arm-linux-gcc
```

```
AR = /usr/local/arm/3.4.1/bin/arm-linux-ar
```

当然, 你的编译器在哪里就改成相应的内容。如果嫌麻烦, 可以从本站资料页面下载我已经打好补丁, 修改好 Makefile 的包, 地址 <http://cid-4201fdc93932ddaf.skydrive.live.com/self.aspx/EE%e5%b0%8f%e7%ab%99%e7%90%90%e7%a2%8e%e6%96%87%e4%bb%b6/termcap-2.0.8.tar.bz2>。需要注意的是, 这个包里 CC = arm-linux-gcc、AR = arm-linux-ar, 请设置好缺省路径。然后就可以编译了:

```
# cd /home/lxz/libtermcap/termcap-2.0.8

# make

# ln -s libtermcap.so.2.0.8 libtermcap.so.2
```

```
# cp libtermcap.so* /home/lxz/rootfs/lib
```

这样, Shell 启动所需要的基本库就都备齐了。但是, 这些库里面还含有调试信息, 体积稍大, 可以把这些信息去掉 (当然不去掉也没有什么影响)。

```
# cd /home/lxz/rootfs/lib
```

```
# arm-linux-strip *.so*
```

至此, 库就制作好了。

• 脚本

有了以上的东西, Shell 还是不能正常工作。可能会是内核提示 Free init memory: XXX 之后就什么输出也没有了, 这时候向终端敲入文字, 可以显示; 就是没有终端提示符, 不理睬输入的命令。这是因为初始化脚本没有启动 Shell。下面介绍这些脚本。

首先是/etc/inittab。内核启动、根文件系统挂载之后所必须的一个文件, 其中列举了 Shell 和整个系统初始化、关闭所需的命令。如果你想让 Shell 出现, 那么只需要加入这么一行

“::askfirst:/bin/ash”。当然如果在编译 Busybox 的时候选择的 shell 不是 “ash”, 而是 hush、lash、msh 之类, 那就改成相应的东西。除了启动 Shell, inittab 还干了很多事情, 我就用我的 inittab 来说明了。注意, 在编译 Busybox 的时候要选上 touch、syslogd、klogd 等命令。

```
# Startup the system
null::sysinit:/bin/mount -o remount,rw /
null::sysinit:/bin/mount -t proc proc /proc
null::sysinit:/bin/mount -a
null::sysinit:/bin/hostname -F /etc/hostname

# Now run any rc scripts
::sysinit:/etc/init.d/rcS

# Now invoke shell
::askfirst:/bin/ash

# Logging junk
null::sysinit:/bin/touch /var/log/messages
```

```

null::respawn:/sbin/syslogd -n -m 0
null::respawn:/sbin/klogd -n

# Stuff to do for the 3-finger salute
::ctrlaltdel:/sbin/reboot

# Stuff to do before rebooting
null::shutdown:/usr/bin/killall klogd
null::shutdown:/usr/bin/killall syslogd
null::shutdown:/bin/umount -a -r
null::shutdown:/sbin/swapoff -a

```

好, 把上面这些储存为 inittab, 启动系统。应该出现两个提示, 没有/etc/fstab 和 /etc/init.d/rcS。目前我的理解/etc/fstab 是用来执行 mount -a 命令的, 里面是文件系统的挂载表。还是用我的 fstab 来说明。

#	<file system>	<mount pt>	<type>	<options>	<dump>	<pass>
	/dev/root	/	ext2	rw,noauto	0	1
	proc	/proc	proc	defaults	0	0
	devpts	/dev/pts	devpts	defaults,gid=5,mode=620	0	0
	tmpfs	/tmp	tmpfs	defaults	0	0
	sysfs	/sys	sysfs	defaults	0	0

还有/etc/init.d/rcS, 这在 PC 上是用来执行/etc/init.d 下所有初始化脚本的一个脚本, 呵呵, 请原谅我不懂得怎么写这种脚本, 对于嵌入式系统, 根本不需要这么复杂, 直接写在/etc/init.d/rcS 里面了。还是用我的/etc/init.d/rcS 来说明, 其中启动 udev 的那些指令对于使用静态设备节点和 devfs 的系统不适用。

```

# Start udev
/bin/mount -t tmpfs tmpfs /dev
/sbin/udev --daemon
/sbin/udevstart

# Configure net interface
/sbin/ifconfig lo 127.0.0.1 up
/sbin/route add -net 127.0.0.0 netmask 255.0.0.0 lo
/sbin/ifconfig eth0 192.168.2.25 netmask 255.255.255.0
/sbin/route add default gw 192.168.2.1

```

如果用的是 udev, 还必须有 udev 的配置脚本。这个写起来有些麻烦, 我们可以直接用 udev 自己带的那些脚本, 位置在 udev 目录下的 etc/udev/udev.conf 和 etc/udev/rules.d 里面的文件。把这些放到根文件系统中去, etc/udev/udev.conf 变为根文件系统的/etc/udev/udev.conf, etc/udev/rules.d 里面的文件变为/etc/udev/rules.d 里面的文件。

至此, 一个可用的最简单文件系统就完成了。之后, 可以使用 mkcramfs、mkyaffs 之类的工具制作文件映像, 这我就不多说了。需要注意的是, cramfs 文件系统是只读的, 就算用的是 initrd 加载的 cramfs 也一样是只读的, 这对根文件系统的/var 目录有一定的影响。除了本文中说的办法, 还可以用 buildroot 这个工具来建, 但是现在我还在尝试中, 以后再说。

***** 《基本步骤总结》

1. 准备工作

下载 busybox 工具 说明: 用于制作可执行命令工具集

2. 开发环境

- 1) 主机: RedHat 9
- 2) 交叉编译工具路径: /usr/local/arm/3.3.2/ (用于编译 busybox1.2.0)
把/usr/local/arm/3.3.2/bin 路径添加到/etc/profile 文件中
- 3) 开发板: 友善之臂 SBC2410
- 4) 开发板分别使用的 Linux2.4.18 内核和 Linux2.6.14 内核验证文件系统

3. 建立目标板空根目录文件夹及根目录下的文件夹

```
[root@190 friendly-arm]# mkdir myroots  
  
[root@190 friendly-arm]# pwd  
  
/friendly-arm/myroots  
  
[root@190 friendly-arm]# cd myroots
```

```
[root@190 myroots]#
```

```
[root@190 myroots]# mkdir bin sbin usr lib dev mnt opt root etc home proc tmp var
```

```
[root@190 myroots]# mkdir etc/init.d
```

进入 etc/init.d 目录下, 建立一个脚本文件, 并命名为 rcS, 用 gedit 打开, 添加如下内容:

```
#!/bin/sh
```

```
PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin:
```

```
runlevel=S
```

```
prevlevel=N
```

```
umask 022
```

```
export PATH runlevel prevlevel
```

```
#
```

```
# Trap CTRL-C &c only in this shell so we can interrupt subprocesses.
```

```
#
```

```
trap ":" INT QUIT TSTP
```

```
[root@190 myroots]# cd ../lib
```

也就是进入 lib 目录, 添加相应的库文件, 具体操作略。

4. 移植 Busybox

进入到压缩文件存放的目录下, 并解压。然后进入解压后的 busybox 目录文件夹, 随后配置 busybox 。

```
[root@190 busybox-1.2.0]# make menuconfig
```

执行之后, 将出现如下图所示的配置界面



由于每个版本的配置选项组织有所不同。不管怎样, 我们注意以下选项就行了:

- 1) Support for devfs
- 2) Build BusyBox as a static binary (no shared libs) //将 busybox 编译成静态链接
- 3) Do you want to build busybox with a Cross Compile?
(/usr/local/arm/3.3.2/bin/arm-linux-) Cross Compile prefix //指定交叉编译器
- 4) init
- 5) Support reading an inittab file //支持 init 读取/etc/inittab 配置文件
- 6) (X) ash 选中 ash //建立的 rcS 脚本才能执行
- 7) ash
- 8) cp cat ls mkdir mv //可执行命令工具的选择, 自己看着办吧, 需要用到的就选上
- 9) mount
- 10) umount
- 11) Support loopback mounts

- 12) Support for the old /etc/mtab file
- 13) insmod
- 14) Support version 2.2.x to 2.4.x Linux kernels
- 15) Support version 2.6.x Linux kernels
- 16) vi

以上内容必须选上, 其他可按默认值; 如果要支持其他功能, 如网络支持等, 可按需选择, 英语不是很烂的话, 这些都没有问题。

配置好之后, 保存退出。然后对其编译和安装到刚才建立的根文件系统目录下:

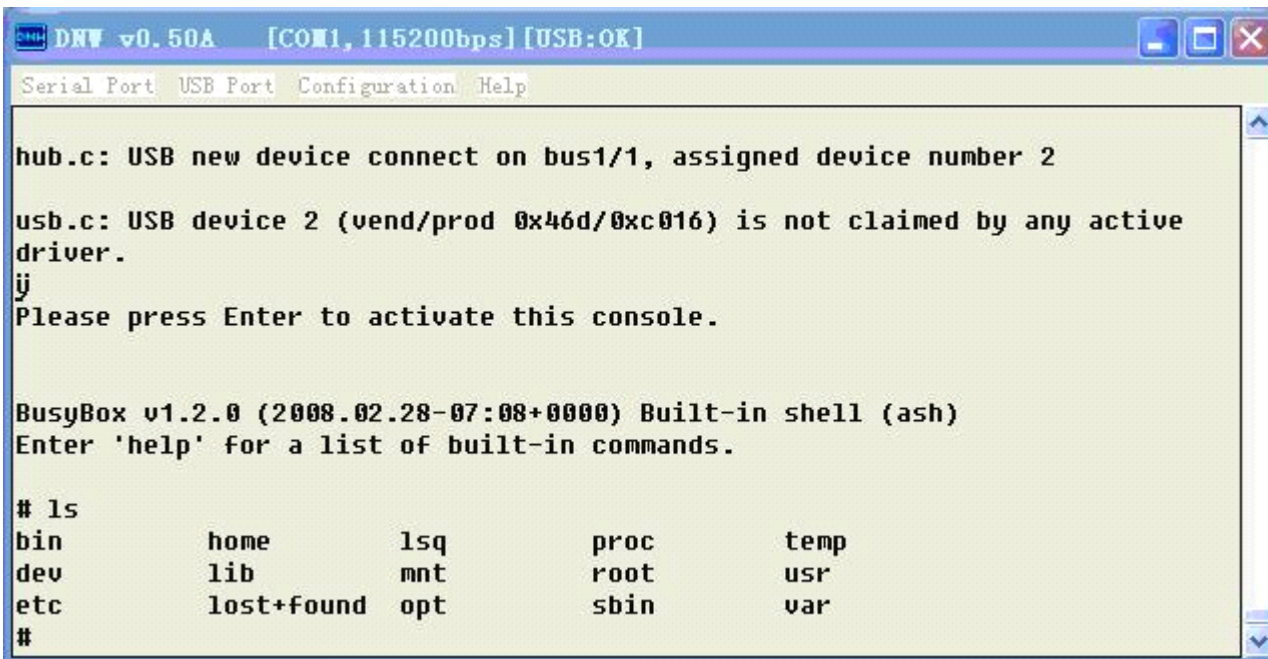
```
[root@190 busybox-1.2.0] make TARGET_ARCH=arm CROSS=arm-linux- \ PREFIX=/friendly-arm/myroots/ all install
```

安装好之后, 将相应的二进制文件拷贝到根文件系统相应的目录下。

5. 制作 yaffs 文件系统包

```
[root@190 friendly-arm]# mkyaffsimage myroots myroots.img
```

6. 下载根文件系统包到开发板上, 并运行, 其结果如图所示



```
DNU v0.50A [COM1, 115200bps] [USB:OK]
Serial Port USB Port Configuration Help

hub.c: USB new device connect on bus1/1, assigned device number 2
usb.c: USB device 2 (vend/prod 0x46d/0xc016) is not claimed by any active
driver.
ij
Please press Enter to activate this console.

BusyBox v1.2.0 (2008.02.28-07:08+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# ls
bin          home        lsq         proc        temp
dev          lib         mnt         root        usr
etc          lost+found  opt         sbin        var
#
```

7. 补充

首先, 本文档制作的根文件系统, 是用 mkyaffsimage 打包成了 yaffs 镜像文件, 也可以使用其他打包工具制作成其他形式的根文件系统, 但必须是内核要有相应文件系统的支持, 否则, 将无法挂上根文件系统。

其次, 我们可以根据项目需要, 在相应的目录下, 添加相应的文件, 并配置相应的服务。如内核动态加载模块可放在 lib 下, 在启动脚本里挂载相应的模块; 在 etc 目录下, 可移植 tinylogin 登录界面应用程序, 等等。

ARM 汇编程序中常见的伪指令分析

ARM 汇编程序分析过程中, 比较难理解的是他的伪操作、宏指令和伪指令。在读 vivi 时遇到很多不懂

的, 所以在此对引导程序中出现伪操作、宏指令和伪指令进行总结,

一、GET option.s

// GET 和 INCLUDE 功能相同

功能: 引进一个被编译过的文件。

格式: GET filename

其中: filename 汇编时引入的文件名, 可以有路径名。

GET 符号在汇编时对宏定义, EQU 符号以及存储映射时是很有用的, 在引入文件汇编完以后, 汇编将从 GET 符号后开始。在被引入的文件中可能有 GET 符号再引入其他的文件。GET 符号不能用来引入目标文件。

二、INTPND EQU 0x01e00004

//EQU 可以用“*”代替, 在阅读源程序时注意。

功能: 对一个数字常量赋予一个符号名。

格式: name EQU expression

其中: name 符号名。Expression 寄存器相关或者程序相关的固定值。

使用 EQU 定义常量, 与 C 语言中用#define 定义一个常量相同。

例: num EQU 2 ; 数字 2 赋予符号 num

三、GBLL THUMBCODE

```
[ {CONFIG} = 16
    THUMBCODE SETL {TRUE}
    CODE32
|
    THUMBCODE SETL {FALSE}
]
[ THUMBCODE
    CODE32 ;for start-up code for Thumb mode
]
```

//其中[=IF, |=ELSE, |=ENDIF, CODE32 表明一下操作都在 ARM 状态。这些都是伪操作这段理解为设定 THUMBCODE 的值, 然后确定, 用户的程序是在 ARM 状态还是 THUMB 状态。

四、MACRO

```
$HandlerLabel HANDLER $HandleLabel
$HandlerLabel
sub sp,sp,#4 ;decrement sp(to store jump address)
stmfd sp!,{r0} ;PUSH the work register to stack
ldr r0,$HandlerLabel;load the address of HandleXXX to r0
ldr r0,[r0] ;load the contents(service routine start address) of HandleXXX
str r0,[sp,#4] ;store the contents(ISR) of HandleXXX to stack
ldmfd sp!,{r0,pc} ;POP the work register and pc(jump to ISR)
MEND
```

//MACRO.....MEND

功能: 标志一下宏的定义。

格式: MACRO

Macro_prototype
MEND

宏表达式的格式如下:

{ \$label } macroname { \$ parameter {,parameter2} ... }

其中:

\$ label 参数, 在宏使用时, 被给定的符号替代。

Macroname 宏的名称, 并不一定以一条指令或者符号名开始。

\$parameter 在宏使用时, 被替代的参数, 格式为: \$parameter="default value"

在宏体中, 参数如: \$parameter 和变量一样使用, 在被宏引用时, 被赋予新值, 参数必须用"\$"符号加于区别。\$label 在宏定义内部符号 时很有用, 可以看作宏的参数。使用"|"符号作为使用一个参数缺省值的变量, 如果使用的是一个空格字符串, 将省去该变量。在使用内部标志的宏定义中, 将内部 标志定义为带后缀的标志, 将会很有用。如果在扩展中空间不够, 可以作为参数和后续文字之间或者参数之间使用圆点隔开, 但在文本和后续参数之间不能使用圆点。宏可以定义局部变量的范围。宏还可以嵌套使用。

例:

MACRO

\$label xmac \$p1,\$p2
 LCLS err

\$labell,loop1
 BGE \$p1

\$labell,loop2
 BL \$p1
 BEG \$p1
 BEG \$labell,loop2

MEND

五、\$和\$\$

//\$临时变量替换, 若程序中需要用字符\$则用\$\$来表示, 通常情况下, 包含在两个||之间的\$并不表示进行变量替换, 但是如果\$是在双引号内, 则将进行变量替换。用"."来分割出变量名的用法,

GBLS STR1

GBLS STR2

STR1 SETS "AAA"

STR2 SETS "BBB\$\$STR1.CCC" //汇编后 STR2 的值为 bbAAACCC

六、IMPORT Main ; The main entry of mon program

//该伪操作告诉编译器当前的符号不是在本文件中定义的, 在本源文件中可能引用该符号, 而不论该源文件是否使用该符号, 该符号都将被加入到本源文件中。

格式:

IMPORT symbol {[WEAK]}

symbol 引用的符号的名称, 他是区分大小写的, [WEAK]指定这个选项后, 如果 symbol 所在的源文件中没有被定义, 编译器也不会报错。他和 EXTERN 作用相同, 不同之处在于, 如果本源文件没有实际引用该符号, 该符号将不会被加入到本源文件的符号表中。

七、AREA Init, CODE, READONLY ENTRY

//功能: 指示汇编器汇编一段新的代码或新的数据区。

格式:

name 给出的特定段名。以数字开头, 必须加竖线, 否则, 将报错, 例如: |l_Data-Area|。某些名字已保留, 如: |C\$\$code| 已经被 C 编译器用作代码, 或者用作与 C 库相连的代码段。

Attr 段名属性, 下列属性是有效的:

ALIGN=expression

缺省状态下, AOF 段将按 4 个字节对准, expression 可以是 2~31 之间的整数, 该段将按 2 (上标为 expression) 字节对准。例如, espression 等于

10, 该段将按 1KB 对准。

CODE 特定机器指令, 缺省为 READONLY。

COMDEF 通用段定义。该 AOF 段可能包括代码和数据, 但必须与其他段名相区别。

COMMON 通用数据段, 无须再注释定义任何代码和数据, 通常由链接器初始化为零。

DATA 包含数据, 但是不包含指令, 缺省为 READWRITE

INTERWORK 表明代码段可以适用 ARM/Thumb interworking 功能。

NOINIT 表明数据段可以初始化为零, 只包含指示符。

PIC 表明定位独立段, 可以不修改情况下, 在任意地址执行。

READONLY 表明该段可读可写。

汇编时, 必须至少有一个 AREA 指示符。使用 AREA 符号可以将源程序区分, 但是必须不重名。通常需要独立的 AOF 段做为代码或者数据段, 较大程序 可以分为多个代码段。AOF 段可以定义局部标签的范围, 可以使用 ROUT 符号。如果没有任何的 AREA 指示符定义, 汇编器将会产生名为|\$\$\$\$\$\$| 的 AOF 段和一条诊断信息, 将限制由于缺少指示符而产生的错误信息, 但是并不一定会成功汇编。

八、LORG

//LORG 是在此指令出现的地方放一个文本池(literal pool). 在 ARM 汇编中常用到

ldr r0, =instruction 将地址 instruction 载入 r0

此时编译器将 ldr 尽可能的转变成 mov 或 mvn 指令。如果转变不成, 将产生一个 ldr 指令, 通过 pc 相对地址从一块保存常数的内存区读出 instruction 的值。此内存区既是文本池。一般的, 文本池放在 END 指令之后的地方。但是, 如果偏移地址大于 4k 空间, ldr 指令会出错(因为 ldr 的相对偏移地址为 12-bit 的值)。此时使用 LORG 放到会出错的 ldr 指令附近, 以解决此问题。编译器会收集没有分配的 ldr 的值放到此文本池中。所以必须在 LDR 指令前后 4KB 的范围内用 LORG 显式地在代码段中添加一个文字池。

九、LDR r0,=WTCON ;watch dog disable

LDR r1,=0x0

功能: 将一个 32 位常量或地址读取至寄存器。

格式:

LDR{condition} register,=[expression|Label-expression]

其中:

condition 可选的条件代码。

register 读取的寄存器。

expression 数字常量:

如果该数字常量在 MOV 或 MVN 指令的范围中, 汇编器会产生合适的指令;

如果该数字量不在 MOV 或 MVN 指令的范围中, 汇编器把该常量于程序后, 用程序相关的 LDR 伪指

令读取, PC 与该常量的偏移量不得超过 4KB。

Label-expression 程序相关的或外部的表达式。汇编器将其存放在程序后的常量库 (称为文字池 (literal pool)) 中, 用程序相关的 LDR 伪指令读取, PC 与与该常量的偏移量不得超过 4KB。

LDR 伪指令的使用有两个目的:

对于不能被 MOV 和 MVN 指令所读取的立即数, 将其变成常量, 进行读取。

将一个程序相关的或外部的表达式读取进寄存器中。

例:

```
LDR    R1, =0xffff
```

```
LDR    R2, =place
```

```
*****
```

十、DCD 0x11110090

;Bank0=OM[1:0], Bank1~Bank7=16bit, bank2=8bit;

//DCD 或"&"

功能: 分配一个或多个字, 从 4 个字节边界开始。

格式:

```
{label}DCD    expression{,expression}...
```

其中:

expression 可以是:

一个数字表达式;

一个程序相关的表达式。

如果在 Thumb 代码中, 使用 DCD 符号定义带标志的数据时则必须使用 DATA 符号。

按 4 个字节对准时, DCD 符号会在第一个字节之前插入 3 个字节的空字符, 如果无须对准的话, 可以使用 DCDU 符号。

例:

```
data1    DCD    1,5,20
```

```
data2    DCD    mem06
```

```
data3    DCD    glb+4
```

```
*****
```

十一、ALIGN

//功能: 从 1 个字边界开始。

格式:

```
ALIGN    {expression    {,offset-expression} }
```

其中:

expression 2(上标为 0)到 2(上标为 31)之间的任意数幂, 当前按 2(上标为 n)字节对准, 如果该参数没有指定, ALIGN 将按字对准。

Offset-expression 定义 expression 指定的对准方式的字节偏移量。

使用 ALIGN 符号, 保证程序正确对准。对于 Thumb 地址, 使用 ALIGN 符号保证其按字对准, 例如: ADR Thumb 伪指令只能读取字对准的地址。

在代码段出现数据定义符时, 使用 ALIGE 符号。当在代码段使用数据定义符 (DCB, DCW, DCWU, DCDU 和%), 程序计数器 PC 并不一定按字对准。

汇编器会在下一条指令时插入 3 个字节, 保证:

ARM 状态下按字对准;

Thumb 状态下按半字对准。

在 Thumb 状态下, 可以使用 ALIGN2 对 Thumb 代码按半字对准。

使用 ALIGN 状态下, 还可以充分利用一些 ARM 处理器的 Cache, 例如, ARM940T 有一个每行 4 字的 Cache, 使用 ALIGN16 按 16 字节对准, 从而最大限度使用 Cache。

十二、^_ISR_STARTADDRESS

//MAP 与"^"

MAP 用于定义一个结构化的内存表 (StorageMAP) 的首地址。此时, 内存表的位置计数器{VAR} (汇编器的内置变量) 设置成该地址值。MAP 可以用"^"代替。

语法: MAP expr {,base-register}

其中, expr 为数字表达式或者是程序中已经定义过的标号。Base-register 为一个寄存器。当指令中没有 Base-register 时, expr 为结构化内存表的首地址。此时, 内存表的位置计数器{VAR}设置成该地址值。当指令中包含这一项时, 结构化内存表的首地址为 expr 和 Base-register 寄存器内容的和。

使用说明: MAP 伪操作和 FIELD 伪操作配合使用来定义结构化的内存表结构。

举例: MAP 伪操作

MAP fun ;fun 就是内存表的首地址

MAP 0x100,R9 ;内存表的首地址为 R9+0x100

十三、HandleReset # 4

HandleUndef # 4

HandleSWI # 4

//FIELD 和"#"

FIELD 用于定义一个结构化的内存表中的数据域。FIELD 可用"#"代替。

语法: {label} FIELD expr

其中: {label}为可选的。当指令中包含这一项时, label 的值为当前内存表的位置计数器{VAR}的值。汇编编译器处理了这条 FIELD 伪操作后。

内存表计数器的值将加上 expr.expr 表示本数据域在内存中所占的字节数。

使用说明: MAP 伪操作和 FIELD 伪操作配合使用来定义结构化的内存表结构。MAP 伪操作定义内存表的首地址。FIELD 伪操作定义内存表的数据域的字节长度, 并可以为每一格数据域指定一个标号, 其他指令可以引用该标号。

MAP 伪操作中的 Base-register 寄存器值队以以后所有 FIELD 伪操作定义的数据域是默认使用的, 直到遇到新的包含 Base-register 项的 MAP 伪操作需要特别注意的是, MAP 伪操作和 FIELD 伪操作仅仅是定义数据结构, 他们并不实际分配内存单元。由 MAP 伪操作和 FIELD 伪操作配合 定义的内存表有 3 种: 基于绝对地址的内存表, 基于相对地址的内存表和基于 PC 的内存表。

举例: 基于绝对地址的内存表

用伪操作序列定义一个内存表, 其首地址为固定的地址 8192 (0X2000), 该内存表中包括 5 个数据域。

Consta 长度为 4 个字节; constb 长为 4 个字节, x 长为 8 字节; y 长为 8 字节; string 长为 16 字节。这种内存表成为基于绝对地址的内存表。

MAP 8192 ; //内存表的首地址 8192 (0x2000)

Consta FIELD 4 ; //consta 长为 4 字节, 相对位置为 0

Constb FIELD 4; //constb 长为 4 字节, 相对位置为 4

X FIELD 8; // X 长为 8 字节, 相对位置为 8

Y FIELD 8; // y 长为 8 字节, 相对位置为 16

String FIELD 16 ;// String 为 16 字节, 相对位置为 24

在指令中, 可以这样引用内存表中的数据域;

LDR R0, consta; //将 consta 地址处对应内存加载到 R0 上面的指令仅仅可以访问 LDR 指令前后 4KB 地址范围的数据域。

举例: 相对绝对地址的内存表

下面的伪操作序列定义一个内存表, 其首地址为 0 与 R9 寄存器值得和, 该内存表中包含 5 个数据域。这种表称为相对地址的内存表。

MAP 0, R9; //内存表的首地址寄存器 R9 的值

Consta FIELD 4; //consta 长为 4 字节, 相对位置为 0

Constb FIELD 4; //constb 长为 4 字节, 相对位置为 4

X FIELD 8; //X 长为 8 字节, 相对位置为 8

Y FIELD 8; //y 长为 8 字节, 相对位置为 16

String FIELD 16; //String 为 16 字节, 相对位置为 24

可以通过下面的指令访问地址范围超过 4KB 的数据;

ADR R9, Field; //伪指令

LDR R5, Constb; //相当于 LDR R5, [R9, #4]

在这里, 内存表中的数据都是相对于 R9 寄存器的内容, 而不是相对于一个固定的地址。通过在 LDR 中指定不同的基址寄存器的值, 定义的内存表结构可以在程序中有多个实例。可多次使用 LDR 指令, 用以实现不同的程序实例。

举例: 基于 PC 的内存表

Data SPACE 100; //分配 100 字节的内存单元, 并初始化为 0

MAP Data; //内存表的首地址为 Datastruc 内存单元

Consta FIELD 4; //consta 长为 4 字节, 相对位置为 0

Constb FIELD 4; //constb 长为 4 字节, 相对位置为 4

X FIELD 8; //X 长为 8 字节, 相对位置为 8

Y FIELD 8; //y 长为 8 字节, 相对位置为 16

String FIELD 16; //String 为 16 字节, 相对位置为 24

可以通过下面的指令访问范围不超过 4kb 的数据;

LDR R5, constb; 相当于 LDR R5,[PC,offset]*****

十四、RN

在局部标号中:

%表示引用操作

F 指示编译器只向前搜索。

B 指示编译器只向后搜索。

A 指示编译器搜索宏的所有嵌套层。

T 指示编译器搜索宏的当前层次。

若 F、B 没有指定则先向前搜索, 再向后搜索。

若 A、T 都没有指定则先从当前层到最高层, 比当前层低的不再搜索

ARM 体系各种异常分析

1.复位异常

(1) 当内核的 nRESET 信号被拉低时, ARM 处理器放弃正在执行的指令, 当 nRESET 信号再次变高时, ARM 处理器进行复位操作;

(2) 系统复位后, 进入管理模式对系统进行初始化, 复位后, 只有 PC (0×00000000) 和 CPSR (nzcvcIfI_t_SVC) 的值是固定的, 另外寄存器的值是随机的。

2.IRQ 异常

(1) 当 CPSR 中的相应的中断屏蔽被清除时, 内核的 nIRQ 信号被拉低时可产生 IRQ 异常;

(2) 由于 ARM 处理器的三级流水线结构, 当异常发生时, PC 的值等于当前执行指令的地址+8 (即正确的中断返回地址+4), 因此 R14 保存的值是 中断返回地址+4, 所以当异常要返回时须执行以下指令:

```
SUBS    PC, R14_irq,#4          ; PC=R14 - 4
```

注意: 在 SUB 指令尾部有个 S, 并且 PC 是目标寄存器, 所以程序返回时 CPSR 将自动从 SPSR 寄存器中恢复;

(3) 将用户模式下的 CPSR 保存到 SPSR_irq 中;

(4) 设置 PC 为 IRQ 异常处理程序的中断入口向量地址, 在 IRQ 模式下该向量地址为 0×00000018 。

3.FIR 异常

(1) 当 CPSR 中的相应 F 位被清零时, 内核的 nFIR 信号被拉低时可产生 FIR 异常, FIQ 异常是优先级最高的中断;

(2) FIQ 异常的进入和退出与 IRQ 异常类似;

(3) 快速中断模式有 8 个专用的寄存器, 可用来满足寄存器保护的需要, 因此从其他模式进入 FIQ 模式时这些寄存器不用压栈了, 提高程序运行的速度, 且在中断入口地址的安排上, FIQ 处于所有异常入口的最后, 这是为了让用户可以从 FIQ 异常入口处 ($0 \times 1c$) 就开始安排中断服务程序, 而不需要再次跳转。

4.未定义指令异常

(1) 当 ARM 在对一条未定义指令进行译码时, 发现这是一条自己和系统内任何协处理器都无法执行的指令时, 就会发生未定义指令异常;

(2) 由于是在对未定义指令译码时发生异常, 所以 PC 的值等于未定义指令的地址+4 (即刚好为中断返回地址), 因此 R14 保存的值是 中断返回地址, 所以当异常要返回时可执行以下指令:

```
MOVS    PC, R14_und
```

5.中止异常

中止表示当前存储器的访问不能完成, 是由外部的 ABOUT 输入信号引起的异常, 分为两类:

(1) 预取指中止: 由程序存储器引起的中止异常;

(2) 数据中止: 由数据存储器引起的中止异常;

5.1 预取指中止异常

当程序发生预取指中止时, ARM 内核将预取的指令标记为无效, 但在指令到达流水线的执行阶段时才进入异常, 因此当前 PC 的值为当前执行指令的地址+8 (即正确的中断返回地址+4), 因此 R14 保存的值是中断返回地址+4, 所以当修复了产生中止的原因后, 不管在什么操作状态, 处理器都会执行以下指令:

```
SUBS    PC, R14_abt,#4          ; PC=R14 - 4
```

5.2 数据中止异常

当发生数据中止异常时, 异常会在“导致异常的指令”执行后的下一条指令时才发生, 因此当前 PC 的值为“导致异常的指令”执行后的下一条指令的地址+8 (即正确的中断返回地址+8), 因此 R14 保存的值是中断返回地址+8, 所以当修复了产生中止的原因后, 不管在什么操作状态, 处理器都会执行以下指令:

```
SUBS    PC, R14_abt,#8          ; PC=R14 - 8
```

注意: LPC2000 系列 ARM 是基于 ARM7TDMI 内核的, 不具有 MMU, 所以不应该发生中止异常, 初学者时常会发生中止异常, 大多数是因为编写的程序的问题。

6.SWI 软件中断异常

(1) 所有的任务都是运行在用户模式下的, 因此任务只能读 CPSR 而不能写 SPSR。任务切换到特权模式下唯一的途径就是使用一个 SWI 指令调用, SWI 指令强迫处理器从用户模式切换到 SVC 管理模式, 并且 IRQ 自动关闭, 所以软件中断方式常被用于系统调用。

(2) 系统调用的具体过程还是看有关 uc/os-II 等操作系统书, 那里比较详细。

(3) SWI 处理程序通过执行下面的指令返回:

```
MOVS    PC, R14_svc
```

具体为什么偏移量为 0, 我现在也还没有搞懂, 请看到的大虾多多指点, 留个言, 谢谢了!!!

经高手指点后明白了原来这么多异常的返回地址问题只要一句话: 除了数据中止以外, 所有异常发生时 R14 保存的值都是跳转时的 PC-4, 只是软件原因引起的异常时执行时(PC 为该指令地址+8)就发生异常跳转了, 而硬件引起的异常为了保证程序安全必须等到当前指令完成后(执行目标已经指向下一个指令, 即 PC 为该指令地址+12)才会发生跳转。

ARM 伪指令讲解

在 ARM 汇编语言程序里, 有一些特殊指令助记符, 这些助记符与指令系统的助记符不同, 没有相对应的操作码, 通常称这些特殊指令助记符为伪指令, 他们所完成的操作称为伪操作。伪指令在源程序中的作用是为完成汇编程序作各种准备工作的, 这些伪指令仅在汇编过程中起作用, 一旦汇编结束, 伪指令的使命就完成。

在 ARM 的汇编程序中, 有如下几种伪指令: 符号定义伪指令、数据定义伪指令、汇编控制伪指令、宏指令以及其他伪指令。

符号定义 (Symbol Definition) 伪指令

符号定义伪指令用于定义 ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等操作。

常见的符号定义伪指令有如下几种:

- 用于定义全局变量的 GBLA 、 GBLL 和 GBLS 。
- 用于定义局部变量的 LCLA 、 LCLL 和 LCLS 。
- 用于对变量赋值的 SETA 、 SETL 、 SETS 。
- 为通用寄存器列表定义名称的 RLIST 。

1、GBLA、GBLL 和 GBLS

语法格式:

GBLA (GBLL 或 GBLS) 全局变量名

GBLA 、 GBLL 和 GBLS 伪指令用于定义一个 ARM 程序中的全局变量, 并将其初始化。其中:

GBLA 伪指令用于定义一个全局的数字变量, 并初始化为 0 ;
GBLL 伪指令用于定义一个全局的逻辑变量, 并初始化为 F (假);
GBLS 伪指令用于定义一个全局的字符串变量, 并初始化为空;
由于以上三条伪指令用于定义全局变量, 因此在整个程序范围内变量名必须唯一。
使用示例:

GBLA Test1 ; 定义一个全局的数字变量, 变量名为 Test1
Test1 SETA 0xaa ; 将该变量赋值为 0xaa
GBLL Test2 ; 定义一个全局的逻辑变量, 变量名为 Test2
Test2 SETL {TRUE} ; 将该变量赋值为真
GBLS Test3 ; 定义一个全局的字符串变量, 变量名为 Test3
Test3 SETS "Testing" ; 将该变量赋值为 "Testing"

2、LCLA、LCLL 和 LCLS

语法格式:

LCLA (LCLL 或 LCLS) 局部变量名

LCLA 、 LCLL 和 LCLS 伪指令用于定义一个 ARM 程序中的局部变量, 并将其初始化。其中:

LCLA 伪指令用于定义一个局部的数字变量, 并初始化为 0 ;
LCLL 伪指令用于定义一个局部的逻辑变量, 并初始化为 F (假);
LCLS 伪指令用于定义一个局部的字符串变量, 并初始化为空;
以上三条伪指令用于声明局部变量, 在其作用范围内变量名必须唯一。
使用示例:

LCLA Test4 ; 声明一个局部的数字变量, 变量名为 Test4
Test3 SETA 0xaa ; 将该变量赋值为 0xaa
LCLL Test5 ; 声明一个局部的逻辑变量, 变量名为 Test5
Test4 SETL {TRUE} ; 将该变量赋值为真
LCLS Test6 ; 定义一个局部的字符串变量, 变量名为 Test6
Test6 SETS "Testing" ; 将该变量赋值为 "Testing"

3、SETA、SETL 和 SETS

语法格式:

变量名 SETA (SETL 或 SETS) 表达式

伪指令 SETA 、 SETL 、 SETS 用于给一个已经定义的全局变量或局部变量赋值。

SETA 伪指令用于给一个数字变量赋值;

SETL 伪指令用于给一个逻辑变量赋值;

SETS 伪指令用于给一个字符串变量赋值;

其中, 变量名为已经定义过的全局变量或局部变量, 表达式为将要赋给变量的值。

使用示例:

LCLA Test3 ; 声明一个局部的数字变量, 变量名为 Test3
Test3 SETA 0xaa ; 将该变量赋值为 0xaa
LCLL Test4 ; 声明一个局部的逻辑变量, 变量名为 Test4
Test4 SETL {TRUE} ; 将该变量赋值为真

4 、 RLIST

语法格式:

名称 RLIST { 寄存器列表 }

RLIST 伪指令可用于对一个通用寄存器列表定义名称, 使用该伪指令定义的名称可在 ARM 指令 LDM/STM 中使用。在 LDM/STM 指令中, 列表中的寄存器访问次序为根据寄存器的编号由低到高, 而与列表中的寄存器排列次序无关。

使用示例:

RegList RLIST {R0-R5 , R8 , R10} ; 将寄存器列表名称定义为 RegList , 可在 ARM 指令 LDM/STM 中通过该名称访问寄存器列表。

数据定义 (Data Definition) 伪指令

数据定义伪指令一般用于为特定的数据分配存储单元, 同时可完成已分配存储单元的初始化。

常见的数据定义伪指令有如下几种:

— DCB 用于分配一片连续的字节存储单元并用指定的数据初始化。

— DCW (DCWU) 用于分配一片连续的半字存储单元并用指定的数据初始化。

— DCD (DCDU) 用于分配一片连续的字存储单元并用指定的数据初始化。

— DCFD (DCFDU) 用于为双精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。

— DCFS (DCFSU) 用于为单精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。

— DCQ (DCQU) 用于分配一片以 8 字节为单位的连续的存储单元并用指定的数据初始化。

— SPACE 用于分配一片连续的存储单元

— MAP 用于定义一个结构化的内存表首地址

— FIELD 用于定义一个结构化的内存表的数据域

1、 DCB

语法格式:

标号 DCB 表达式

DCB 伪指令用于分配一片连续的字节存储单元并用伪指令中指定的表达式初始化。其中, 表达式可以为 0 ~ 255 的数字或字符串。DCB 也可用 “=” 代替。

使用示例:

Str DCB “ This is a test ! ” ; 分配一片连续的字节存储单元并初始化。

2、 DCW (或 DCWU)

语法格式:

标号 DCW (或 DCWU) 表达式

DCW (或 DCWU) 伪指令用于分配一片连续的半字存储单元并用伪指令中指定的表达式初始化。

其中, 表达式可以为程序标号或数字表达式。。

用 DCW 分配的字存储单元是半字对齐的, 而用 DCWU 分配的字存储单元并不严格半字对齐。

使用示例:

DataTest DCW 1 , 2 , 3 ; 分配一片连续的半字存储单元并初始化。

3、DCD (或 DCDU)

语法格式:

标号 DCD (或 DCDU) 表达式

DCD (或 DCDU) 伪指令用于分配一片连续的字存储单元并用伪指令中指定的表达式初始化。其中, 表达式可以为程序标号或数字表达式。DCD 也可用 “&” 代替。

用 DCD 分配的字存储单元是字对齐的, 而用 DCDU 分配的字存储单元并不严格字对齐。

使用示例:

DataTest DCD 4 , 5 , 6 ; 分配一片连续的字存储单元并初始化。

4、DCFD (或 DCFDU)

语法格式:

标号 DCFD (或 DCFDU) 表达式

DCFD (或 DCFDU) 伪指令用于为双精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个双精度的浮点数占据两个字单元。用 DCFD 分配的字存储单元是字对齐的, 而用 DCFDU 分配的字存储单元并不严格字对齐。

使用示例:

FDataTest DCFD 2E115 , -5E7 ; 分配一片连续的字存储单元并初始化为指定的双精度数。

5、DCFS (或 DCFSU)

语法格式:

标号 DCFS (或 DCFSU) 表达式

DCFS (或 DCFSU) 伪指令用于为单精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个单精度的浮点数占据一个字单元。用 DCFS 分配的字存储单元是字对齐的, 而用 DCFSU 分配的字存储单元并不严格字对齐。

使用示例:

FDataTest DCFS 2E5 , -5E - 7 ; 分配一片连续的字存储单元并初始化为指定的单精度数。

6、DCQ(或 DCQU)

语法格式:

标号 DCQ (或 DCQU) 表达式

DCQ (或 DCQU) 伪指令用于分配一片以 8 个字节为单位的连续存储区域并用伪指令中指定的表达式初始化。

用 DCQ 分配的存储单元是字对齐的, 而用 DCQU 分配的存储单元并不严格字对齐。

使用示例:

DataTest DCQ 100 ; 分配一片连续的存储单元并初始化为指定的值。

7、SPACE

语法格式:

标号 SPACE 表达式

SPACE 伪指令用于分配一片连续的存储区域并初始化为 0 。其中, 表达式为要分配的字节数。

SPACE 也可用 “ % ” 代替。

使用示例:

DataSpace SPACE 100 ; 分配连续 100 字节的存储单元并初始化为 0 。

8、MAP

语法格式:

MAP 表达式 { , 基址寄存器 }

MAP 伪指令用于定义一个结构化的内存表的首地址。MAP 也可用 “ ^ ” 代替。

表达式可以为程序中的标号或数学表达式, 基址寄存器为可选项, 当基址寄存器选项不存在时, 表达式的值即为内存表的首地址, 当该选项存在时, 内存表的首地址为表达式的值与基址寄存器的和。

MAP 伪指令通常与 FIELD 伪指令配合使用来定义结构化的内存表。

使用示例:

MAP 0x100 , R0 ; 定义结构化内存表首地址的值为 0x100 + R0 。

9、FIELD

语法格式:

标号 FIELD 表达式

FIELD 伪指令用于定义一个结构化内存表中的数据域。FIELD 也可用 “ # ” 代替。

表达式的值为当前数据域在内存表中所占的字节数。

FIELD 伪指令常与 MAP 伪指令配合使用来定义结构化的内存表。MAP 伪指令定义内存表的首地址, FIELD 伪指令定义内存表中的各个数据域, 并可以为每个数据域指定一个标号供其他的指令引用。

注意 MAP 和 FIELD 伪指令仅用于定义数据结构, 并不实际分配存储单元。

使用示例:

MAP 0x100 ; 定义结构化内存表首地址的值为 0x100 。

A FIELD 16 ; 定义 A 的长度为 16 字节, 位置为 0x100

B FIELD 32 ; 定义 B 的长度为 32 字节, 位置为 0x110

S FIELD 256 ; 定义 S 的长度为 256 字节, 位置为 0x130

汇编控制 (Assembly Control) 伪指令

汇编控制伪指令用于控制汇编程序的执行流程, 常用的汇编控制伪指令包括以下几条:

— IF 、 ELSE 、 ENDIF

— WHILE 、 WEND

— MACRO 、 MEND

— MEXIT

1、IF、ELSE、ENDIF

语法格式:

IF 逻辑表达式

指令序列 1

ELSE

指令序列 2

ENDIF

IF 、 ELSE 、 ENDIF 伪指令能根据条件的成立与否决定是否执行某个指令序列。当 IF 后面的逻辑表达式为真, 则执行指令序列 1 , 否则执行指令序列 2 。其中, ELSE 及指令序列 2 可以没有, 此时, 当 IF 后面的逻辑表达式为真, 则执行指令序列 1 , 否则继续执行后面的指令。

IF 、 ELSE 、 ENDIF 伪指令可以嵌套使用。

使用示例:

GBLL Test ; 声明一个全局的逻辑变量, 变量名为 Test.....

IF Test = TRUE

指令序列 1

ELSE

指令序列 2

ENDIF

2、 WHILE、 WEND

语法格式:

WHILE 逻辑表达式

指令序列

WEND

WHILE 、 WEND 伪指令能根据条件的成立与否决定是否循环执行某个指令序列。当 WHILE 后面的逻辑表达式为真, 则执行指令序列, 该指令序列执行完毕后, 再判断逻辑表达式的值, 若为真则继续执行, 一直到逻辑表达式的值为假。

WHILE 、 WEND 伪指令可以嵌套使用。

使用示例:

GBLA Counter ; 声明一个全局的数学变量, 变量名为 Counter

Counter SETA 3 ; 由变量 Counter 控制循环次数

.....

WHILE Counter < 10

指令序列

WEND

3、 MACRO、 MEND

语法格式:

\$ 标号 宏名 \$ 参数 1 , \$ 参数 2 ,

指令序列

MEND

MACRO 、 MEND 伪指令可以将一段代码定义为一个整体, 称为宏指令, 然后就可以在程序中通过宏指令多次调用该段代码。其中, \$ 标号在宏指令被展开时, 标号会被替换为用户定义的符号, 宏指令可以使用一个或多个参数, 当宏指令被展开时, 这些参数被相应的值替换。

宏指令的使用方式和功能与子程序有些相似, 子程序可以提供模块化的程序设计、节省存储空间并提高运行速度。但在使用子程序结构时需要保护现场, 从而增加了系统的开销, 因此, 在代码较短且需要传递的参数较多时, 可以使用宏指令代替子程序。

包含在 MACRO 和 MEND 之间的指令序列称为宏定义体, 在宏定义体的第一行应声明宏的原型(包含宏名、所需的参数), 然后就可以在汇编程序中通过宏名来调用该指令序列。在源程序被编译时, 汇编器将宏调用展开, 用宏定义中的指令序列代替程序中的宏调用, 并将实际参数的值传递给宏定义中的形式参数。

MACRO 、 MEND 伪指令可以嵌套使用。

4、 MEXIT

语法格式:

MEXIT

MEXIT 用于从宏定义中跳转出去。

其他常用的伪指令

还有一些其他的伪指令, 在汇编程序中经常会被使用, 包括以下几条:

- AREA
- ALIGN
- CODE16 、 CODE32
- ENTRY
- END
- EQU
- EXPORT (或 GLOBAL)
- IMPORT
- EXTERN
- GET (或 INCLUDE)
- INCBIN
- RN
- ROUT

1、 AREA

语法格式:

AREA 段名 属性 1 , 属性 2 ,

AREA 伪指令用于定义一个代码段或数据段。其中, 段名若以数字开头, 则该段名需用“|”括起来, 如 |1_test|。

属性字段表示该代码段(或数据段)的相关属性, 多个属性用逗号分隔。常用的属性如下:

- CODE 属性: 用于定义代码段, 默认为 READONLY。
- DATA 属性: 用于定义数据段, 默认为 READWRITE。
- READONLY 属性: 指定本段为只读, 代码段默认为 READONLY。
- READWRITE 属性: 指定本段为可读可写, 数据段的默认属性为 READWRITE。
- ALIGN 属性: 使用方式为 ALIGN 表达式。在默认时, ELF (可执行连接文件)的代码段和数据段是按字对齐的, 表达式的取值范围为 0 ~ 31, 相应的对齐方式为 2 表达式次方。

— COMMON 属性: 该属性定义一个通用的段, 不包含任何的用户代码和数据。各源文件中同名的 COMMON 段共享同一段存储单元。

一个汇编语言程序至少要包含一个段, 当程序太长时, 也可以将程序分为多个代码段和数据段。

使用示例:

AREA Init , CODE , READONLY

该伪指令定义了一个代码段, 段名为 Init , 属性为只读

2、 ALIGN

语法格式:

ALIGN { 表达式 { , 偏移量 } }

ALIGN 伪指令可通过添加填充字节的方式, 使当前位置满足一定的对其方式 | 。其中, 表达式的值用于指定对齐方式, 可能的取值为 2 的幂, 如 1 、 2 、 4 、 8 、 16 等。若未指定表达式, 则将当前位置对齐到下一个字的位置。偏移量也为一个数字表达式, 若使用该字段, 则当前位置的对齐方式为: 2 的表达式次幂+偏移量。

使用示例:

AREA Init , CODE , READONLY , ALIEN = 3 ; 指定后面的指令为 8 字节对齐。

指令序列

END

3、CODE16、CODE32

语法格式:

CODE16 (或 CODE32)

CODE16 伪指令通知编译器, 其后的指令序列为 16 位的 Thumb 指令。

CODE32 伪指令通知编译器, 其后的指令序列为 32 位的 ARM 指令。

若在汇编源程序中同时包含 ARM 指令和 Thumb 指令时, 可用 CODE16 伪指令通知编译器其后的指令序列为 16 位的 Thumb 指令, CODE32 伪指令通知编译器其后的指令序列为 32 位的 ARM 指令。因此, 在使用 ARM 指令和 Thumb 指令混合编程的代码里, 可用这两条伪指令进行切换, 但注意他们只通知编译器其后指令的类型, 并不能对处理器进行状态的切换。

使用示例:

AREA Init , CODE , READONLY

.....

CODE32 ; 通知编译器其后的指令为 32 位的 ARM 指令

LDR R0 , = NEXT + 1 ; 将跳转地址放入寄存器 R0

BX R0 ; 程序跳转到新的位置执行, 并将处理器切换到 Thumb 工作状态

.....

CODE16 ; 通知编译器其后的指令为 16 位的 Thumb 指令

NEXT LDR R3, =0x3FF

.....

END ; 程序结束

4、ENTRY

语法格式:

ENTRY

ENTRY 伪指令用于指定汇编程序的入口点。在一个完整的汇编程序中至少要有一个 ENTRY (也可以有多个, 当有多个 ENTRY 时, 程序的真正入口点由链接器指定), 但在一个源文件里最多只能有一个 ENTRY (可以没有)。

使用示例:

AREA Init , CODE , READONLY

ENTRY ; 指定应用程序的入口点

.....

5、END

语法格式:

END

END 伪指令用于通知编译器已经到了源程序的结尾。

使用示例:

AREA Init , CODE , READONLY

.....

END ; 指定应用程序的结尾

6、EQU

语法格式:

名称 EQU 表达式 { , 类型 }

EQU 伪指令用于为程序中的常量、标号等定义一个等效的字符名称, 类似于 C 语言中的 # define 。

其中 EQU 可用 “*” 代替。

名称为 EQU 伪指令定义的字符名称, 当表达式为 32 位的常量时, 可以指定表达式的数据类型, 可以有以下三种类型:

CODE16 、 CODE32 和 DATA

使用示例:

Test EQU 50 ; 定义标号 Test 的值为 50

Addr EQU 0x55 , CODE32 ; 定义 Addr 的值为 0x55 , 且该处为 32 位的 ARM 指令。

7、EXPORT (或 GLOBAL)

语法格式:

EXPORT 标号 {[WEAK]}

EXPORT 伪指令用于在程序中声明一个全局的标号, 该标号可在其他的文件中引用。EXPORT 可用 GLOBAL 代替。标号在程序中区分大小写, [WEAK] 选项声明其他的同名标号优先于该标号被引用。

使用示例:

AREA Init , CODE , READONLY

EXPORT Stest ; 声明一个可全局引用的标号 Stest.....

END

8、IMPORT

语法格式:

IMPORT 标号 {[WEAK]}

IMPORT 伪指令用于通知编译器要使用的标号在其他的源文件中定义, 但要在当前源文件中引用, 而且无论当前源文件是否引用该标号, 该标号均会被加入到当前源文件的符号表中。

标号在程序中区分大小写, [WEAK] 选项表示当所有的源文件都没有定义这样一个标号时, 编译器也不给出错误信息, 在多数情况下将该标号置为 0 , 若该标号为 B 或 BL 指令引用, 则将 B 或 BL 指令置为 NOP 操作。

使用示例:

AREA Init , CODE , READONLY

IMPORT Main ; 通知编译器当前文件要引用标号 Main, 但 Main 在其他源文件中定义.....

END

9、EXTERN

语法格式:

EXTERN 标号 {[WEAK]}

EXTERN 伪指令用于通知编译器要使用的标号在其他的源文件中定义, 但要在当前源文件中引用, 如果当前源文件实际并未引用该标号, 该标号就不会被加入到当前源文件的符号表中。标号在程序中区分大小写, [WEAK] 选项表示当所有的源文件都没有定义这样一个标号时, 编译器也不给出错误信息, 在多数情况下将该标号置为 0, 若该标号为 B 或 BL 指令引用, 则将 B 或 BL 指令置为 NOP 操作。

使用示例:

```
AREA Init , CODE , READONLY
```

```
EXTERN Main ; 通知编译器当前文件要引用标号 Main, 但 Main 在其他源文件中定义.....
```

```
END
```

10、 GET (或 INCLUDE)

语法格式:

GET 文件名

GET 伪指令用于将一个源文件包含到当前的源文件中, 并将被包含的源文件在当前位置进行汇编处理。可以使用 INCLUDE 代替 GET。

汇编程序中常用的方法是在某源文件中定义一些宏指令, 用 EQU 定义常量的符号名称, 用 MAP 和 FIELDED 定义结构化的数据类型, 然后用 GET 伪指令将这个源文件包含到其他的源文件中。使用方法与 C 语言中的 “include” 相似。

GET 伪指令只能用于包含源文件, 包含目标文件需要使用 INCBIN 伪指令

使用示例:

```
AREA Init , CODE , READONLY
```

```
GET a1.s ; 通知编译器当前源文件包含源文件 a1.s
```

```
GET C: \a2.s ; 通知编译器当前源文件包含源文件 C: \a2.s .....
```

```
END
```

11、 INCBIN

语法格式:

INCBIN 文件名

INCBIN 伪指令用于将一个目标文件或数据文件包含到当前的源文件中, 被包含的文件不作任何变动的存放在当前文件中, 编译器从其后开始继续处理。

使用示例:

```
AREA Init , CODE , READONLY
```

```
INCBIN a1.dat ; 通知编译器当前源文件包含文件 a1.dat
```

```
INCBIN C: \a2.txt ; 通知编译器当前源文件包含文件 C: \a2.txt .....
```

```
END
```

12、 RN

语法格式:

名称 RN 表达式

RN 伪指令用于给一个寄存器定义一个别名。采用这种方式可以方便程序员记忆该寄存器的功能。其中, 名称为给寄存器定义的别名, 表达式为寄存器的编码。

使用示例:

```
Temp RN R0 ; 将 R0 定义一个别名 Temp
```

13、 ROUT

语法格式:

```
{ 名称 } ROUT
```

ROUT 伪指令用于给一个局部变量定义作用范围。在程序中未使用该伪指令时, 局部变量的作用范围为所在的 AREA, 而使用 ROUT 后, 局部变量的作用范围为当前 ROUT 和下一个 ROUT 之间

C 语言最大难点揭秘

本文将带您了解一些良好的和内存相关的编码实践, 以将内存错误保持在控制范围内。内存错误是 C 和 C++ 编程的祸根: 它们很普遍, 认识其严重性已有二十多年, 但始终没有彻底解决, 它们可能严重影响应用程序, 并且很少有开发团队对其制定明确的管理计划。但好消息是, 它们并不怎么神秘。

C 和 C++ 程序中的内存错误非常有害: 它们很常见, 并且可能导致严重的后果。来自计算机应急响应小组 (请参见参考资料) 和供应商的许多最严重的安全公告都是由简单的内存错误造成的。自从 70 年代末期以来, C 程序员就一直讨论此类错误, 但其影响在 2007 年仍然很大。更糟的是, 如果按我的思路考虑, 当今的许多 C 和 C++ 程序员可能都会认为内存错误是不可控制而又神秘的顽症, 它们只能纠正, 无法预防。

但事实并非如此。本文将让您在短时间内理解与良好内存相关的编码的所有本质:

正确的内存管理的重要性

内存错误的类别

内存编程的策略

正确的内存管理的重要性

存在内存错误的 C 和 C++ 程序会导致各种问题。如果它们泄漏内存, 则运行速度会逐渐变慢, 并最终停止运行; 如果覆盖内存, 则会变得非常脆弱, 很容易受到恶意用户的攻击。从 1988 年著名的莫里斯蠕虫 攻击到有关 Flash Player 和其他关键的零售级程序的最新安全警报都与缓冲区溢出有关: “大多数计算机安全漏洞都是缓冲区溢出”, Rodney Bates 在 2004 年写道。

在可以使用 C 或 C++ 的地方, 也广泛支持使用其他许多通用语言 (如 Java?、Ruby、Haskell、C#、Perl、Smalltalk 等), 每种语言都有众多的爱好者和各自的优点。但是, 从计算角度来看, 每种编程语言优于 C 或 C++ 的主要优点都与便于内存管理密切相关。与内存相关的编程是如此重要, 而在实践中正确应用又是如此困难, 以致于它支配着面向对象编程语

言、功能性编程语言、高级编程语言、声明性编程语言和另外一些编程语言的所有其他变量或理论。

与少数其他类型的常见错误一样, 内存错误还是一种隐性危害: 它们很难再现, 症状通常不能在相应的源代码中找到。例如, 无论何时何地发生内存泄漏, 都可能表现为应用程序完全无法接受, 同时内存泄漏不是显而易见。

因此, 出于所有这些原因, 需要特别关注 C 和 C++ 编程的内存问题。让我们看一看如何解决这些问题, 先不谈是哪种语言。

内存错误的类别

首先, 不要失去信心。有很多办法可以对付内存问题。我们先列出所有可能存在的实际问题:

内存泄漏

错误分配, 包括大量增加 free() 释放的内存和未初始化的引用

悬空指针

数组边界违规

这是所有类型。即使迁移到 C++ 面向对象的语言, 这些类型也不会有明显变化; 无论数据是简单类型还是 C 语言的 struct 或 C++ 的类, C 和 C++ 中内存管理和引用的模型在原理上都是相同的。以下内容绝大部分是“纯 C”语言, 对于扩展到 C++ 主要留作练习使用。

内存泄漏

在分配资源时会发生内存泄漏, 但是它从不回收。下面是一个可能出错的模型 (请参见清单 1) :

清单 1. 简单的潜在堆内存丢失和缓冲区覆盖 复制内容到剪贴板 代码:

```
void f1(char *explanation)
{
    char p1;

    p1 = malloc(100);
    (void) sprintf(p1,
                  "The f1 error occurred because of '%s'.",
                  explanation);
    local_log(p1);
}
```


您看到问题了吗? 除非 `local_log()` 对 `free()` 释放的内存具有不寻常的响应能力, 否则每次对 `f1` 的调用都会泄漏 100 字节。在内存棒增量分发数兆字节内存时, 一次泄漏是微不足道的, 但是连续操作数小时后, 即使如此小的泄漏也会削弱应用程序。

在实际的 C 和 C++ 编程中, 这不足以影响您对 `malloc()` 或 `new` 的使用, 本部分开头的句子提到了“资源”不是仅指“内存”, 因为还有类似以下内容的示例 (请参见清单 2)。FILE 句柄可能与内存块不同, 但是必须对它们给予同等关注:

清单 2. 来自资源错误管理的潜在堆内存丢失 复制内容到剪贴板 代码:

```
int getkey(char *filename)
{
    FILE *fp;
    int key;

    fp = fopen(filename, "r");
    fscanf(fp, "%d", &key);
    return key;
}
```

`fopen` 的语义需要补充性的 `fclose`。在没有 `fclose()` 的情况下, C 标准不能指定发生的情况时, 很可能是内存泄漏。其他资源 (如信号量、网络句柄、数据库连接等) 同样值得考虑。

内存错误分配

错误分配的管理不是很困难。下面是一个示例 (请参见清单 3): 复制内容到剪贴板 代码:清单 3. 未初始化的指针

```
void f2(int datum)
{
    int *p2;

    /* Uh-oh! No one has initialized p2. */
    *p2 = datum;
    ...
}
```

关于此类错误的好消息是, 它们一般具有显著结果。在 AIX? 下, 对未初始化指针的分配通常会立即导致 `segmentation fault` 错误。它的好处是任何此类错误都会被快速地检测到; 与花费数月时间才能确定且难以再现的错误相比, 检测此类错误的代价要小得多。

在此错误类型中存在多个变种。`free()` 释放的内存比 `malloc()` 更频繁 (请参见清单 4):

清单 4. 两个错误的内存释放 复制内容到剪贴板 代码:

```
/* Allocate once, free twice. */
void f3()
{
    char *p;

    p = malloc(10);
    ...
    free(p);
    ...
    free(p);
}

/* Allocate zero times, free once. */
void f4()
{
    char *p;

    /* Note that p remains uninitialized here. */
    free(p);
}
```

这些错误通常也不太严重。尽管 C 标准在这些情形中没有定义具体行为, 但典型的实现将忽略错误, 或者快速而明确地对它们进行标记; 总之, 这些都是安全情形。

悬空指针

悬空指针比较棘手。当程序员在内存资源释放后使用资源时会发生悬空指针 (请参见清单 5) :

清单 5. 悬空指针 复制内容到剪贴板 代码:

```
void f8()
{
    struct x *xp;

    xp = (struct x *) malloc(sizeof (struct x));
    xp.q = 13;
    ...
    free(xp);
    ...
    /* Problem! There's no guarantee that
       the memory block to which xp points
       hasn't been overwritten. */
    return xp.q;
}
```

传统的“调试”难以隔离悬空指针。由于下面两个明显原因, 它们很难再现:

即使影响提前释放内存范围的代码已本地化, 内存的使用仍然可能取决于应用程序甚至(在极端情况下)不同进程中的其他执行位置。

悬空指针可能发生在以微妙方式使用内存的代码中。结果是, 即使内存存在释放后立即被覆盖, 并且新指向的值不同于预期值, 也很难识别出新值是错误值。

悬空指针不断威胁着 C 或 C++ 程序的运行状态。

数组边界违规

数组边界违规十分危险, 它是内存错误管理的最后一个主要类别。回头看一下清单 1; 如果 explanation 的长度超过 80, 则会发生什么情况? 回答: 难以预料, 但是它可能与良好情形相差甚远。特别是, C 复制一个字符串, 该字符串不适用于为它分配的 100 个字符。在任何常规实现中, “超过的”字符会覆盖内存中的其他数据。内存中数据分配的布局非常复杂并且难以再现, 所以任何症状都不可能追溯到源代码级别的具体错误。这些错误通常会导致数百万美元的损失。

内存编程的策略

勤奋和自律可以让这些错误造成的影响降至最低限度。下面我们介绍一下您可以采用的几个特定步骤; 我在各种组织中处理它们的经验是, 至少可以按一定的数量级持续减少内存错误。

编码风格

编码风格是最重要的, 我还从没有看到过其他任何作者对此加以强调。影响资源(特别是内存)的函数和方法需要显式地解释本身。下面是有关标头、注释或名称的一些示例(请参见清单 6)。

清单 6. 识别资源的源代码示例 复制内容到剪贴板 代码:

```
/******  
 * ...  
 *  
 * Note that any function invoking protected_file_read()  
 * assumes responsibility eventually to fclose() its  
 * return value, UNLESS that value is NULL.  
 *  
 *****/  
FILE *protected_file_read(char *filename)  
{  
    FILE *fp;  
  
    fp = fopen(filename, "r");  
    if (fp) {  
        ...  
    } else {
```

```
        ...
    }
    return fp;
}

/*****
 * ...
 *
 * Note that the return value of get_message points to a
 * fixed memory location.  Do NOT free() it; remember to
 * make a copy if it must be retained ...
 *
 *****/
char *get_message()
{
    static char this_buffer[400];

    ...
    (void) sprintf(this_buffer, ...);
    return this_buffer;
}

/*****
 * ...
 * While this function uses heap memory, and so
 * temporarily might expand the over-all memory
 * footprint, it properly cleans up after itself.
 *
 *****/
int f6(char *item1)
{
    my_class c1;
    int result;

    ...
    c1 = new my_class(item1);
    ...
    result = c1.x;
    delete c1;
    return result;
}

/*****
 * ...
 * Note that f8() is documented to return a value
```

```
* which needs to be returned to heap; as f7 thinly
* wraps f8, any code which invokes f7() must be
* careful to free() the return value.
*
*****/
int *f7()
{
    int *p;

    p = f8(...);
    ...
    return p;
}
```

使这些格式元素成为您日常工作的一部分。可以使用各种方法解决内存问题:

专用库

语言

软件工具

硬件检查器

在这整个领域中, 我始终认为最有用并且投资回报率最大的是考虑改进源代码的风格。它不需要昂贵的代价或严格的形式; 可以始终取消与内存无关的段的注释, 但影响内存的定义当然需要显式注释。添加几个简单的单词可使内存结果更清楚, 并且内存编程会得到改进。

我没有做受控实验来验证此风格的效果。如果您的经历与我一样, 您将发现没有说明资源影响的策略简直无法忍受。这样做很简单, 但带来的好处太多了。

检测

检测是编码标准的补充。二者各有裨益, 但结合使用效果特别好。机灵的 C 或 C++ 专业人员甚至可以浏览不熟悉的源代码, 并以极低的成本检测内存问题。通过少量的实践和适当的文本搜索, 您能够快速验证平衡的 *alloc() 和 free() 或者 new 和 delete 的源主体。人工查看此类内容通常会出现像清单 7 中一样的问题。

清单 7. 棘手的内存泄漏 复制内容到剪贴板 代码:

```
static char *important_pointer = NULL;
void f9()
{
    if (!important_pointer)
        important_pointer = malloc(IMPORTANT_SIZE);
    ...
    if (condition)
        /* Oops! We just lost the reference
```

```
        important_pointer already held. */
        important_pointer = malloc(DIFFERENT_SIZE);
    ...
}
```

如果 condition 为真, 简单使用自动运行时工具不能检测发生的内存泄漏。仔细进行源分析可以从此类条件推理出证实正确的结论。我重复一下我写的关于风格的内容: 尽管大量发布的内存问题描述都强调工具和语言, 对于我来说, 最大的收获来自“软的”以开发人员为中心的流程变更。您在风格和检测上所做的任何改进都可以帮助您理解由自动化工具产生的诊断。

静态的自动语法分析

当然, 并不是只有人类才能读取源代码。您还应使静态语法分析 成为开发流程的一部分。静态语法分析是 lint、严格编译 和几种商业产品执行的内容: 扫描编译器接受的源文本和目标项, 但这可能是错误的症状。

希望让您的代码无 lint。尽管 lint 已过时, 并有一定的局限性, 但是, 没有使用它(或其较高级的后代)的许多程序员犯了很大的错误。通常情况下, 您能够编写忽略 lint 的优秀专业质量代码, 但努力这样做的结果通常会发生重大错误。其中一些错误影响内存的正确性。与让客户首先发现内存错误的代价相比, 即使对这种类别的产品支付最昂贵的许可费也失去了意义。清除源代码。现在, 即使 lint 标记的编码可能向您提供所需的功能, 但很可能存在更简单的方法, 该方法可满足 lint, 并且比较强键又可移植。

内存库

补救方法的最后两个类别与前三个明显不同。前者是轻量级 的; 一个人可以容易地理解并实现它们。另一方面, 内存库和工具通常具有较高的许可费用, 对部分开发人员来说, 它们需要进一步完善和调整。有效地使用库和工具的程序员是理解轻量级的静态 方法的人员。可用的库和工具给人的印象很深: 其作为组的质量很高。但是, 即使最优秀的编程人员也可能会被忽略内存管理基本原则的非常任性的编程人员搅乱。据我观察, 普通的编程人员在尝试利用内存库和工具进行隔离工作时也只能感到灰心。

由于这些原因, 我们催促 C 和 C++ 程序员为解决内存问题先了解一下自己的源。在这完成之后, 才去考虑库。

使用几个库能够编写常规的 C 或 C++ 代码, 并保证改进内存管理。Jonathan Bartlett 在 developerWorks 的 2004 评论专栏中介绍了主要的候选项, 可以在下面的参考资料部分获得。库可以解决多种不同的内存问题, 以致于直接对它们进行比较是非常困难的; 这方面的常见主题包括垃圾收集、智能指针 和 智能容器。大体上说, 库可以自动进行较多的内存管理, 这样程序员可以犯更少的错误。

我对内存库有各种感受。他们在努力工作, 但我看到他们在项目中获得的成功比预期要小, 尤其在 C 方面。我尚未对这些令人失望的结果进行仔细分析。例如, 业绩应该与相应的手动 内存管理一样好, 但是这是一个灰色区域——尤其在垃圾收集库处理速度缓慢的情况下。通过这方面的实践得出的最明确的结论是, 与 C 关注的代码组相比, C++ 似乎可以较好地接受智能指针。

内存工具

开发真正基于 C 的应用程序的开发团队需要运行时内存工具作为其开发策略的一部分。已介绍的技术很有价值, 而且不可或缺。在您亲自尝试使用内存工具之前, 其质量和功能您可能还不了解。

本文主要讨论了基于软件的内存工具。还有硬件内存调试器; 在非常特殊的情况下 (主要是在使用不支持其他工具的专用主机时) 才考虑它们。

市场上的软件内存工具包括专有工具 (如 IBM Rational? Purify 和 Electric Fence) 和其他开放源代码工具。其中有许多可以很好地与 AIX 和其他操作系统一起使用。

所有内存工具的功能基本相同: 构建可执行文件的特定版本 (很像在编译时通过使用 -g 标记生成的调试版本)、练习相关应用程序和研究由工具自动生成的报告。请考虑如清单 8 所示的程序。

清单 8. 示例错误 复制内容到剪贴板 代码:

```
int main()
{
    char p[5];
    strcpy(p, "Hello, world.");
    puts(p);
}
```

此程序可以在许多环境中 “运行”, 它编译、执行并将 “Hello, world.\n” 打印到屏幕。使用内存工具运行相同应用程序会在第四行产生一个数组边界违规的报告。在了解软件错误 (将十四个字符复制到了只能容纳五个字符的空间中) 方面, 这种方法比在客户处查找错误症状的花费小得多。这是内存工具的功劳。

作为一名成熟的 C 或 C++ 程序员, 您认识到内存问题值得特别关注。通过制订一些计划和实践, 可以找到控制内存错误的方法。学习内存使用的正确模式, 快速发现可能发生的错误, 使本文介绍的技术成为您日常工作的一部分。您可以在开始时就消除应用程序中的症状, 否则可能要花费数天或数周时间来调试

C 语言中文件输入输出函数

1.2 文件的输入输出函数

键盘、显示器、打印机、磁盘驱动器等逻辑设备, 其输入输出都可以通过文件管理的方法来完成。而在编程时使用最多的要算是磁盘文件, 因此本节主要以磁盘文件为主, 详细介绍 Turbo C2.0 提供的文件操作函数, 当然这些对文件的操作函数也适合于非磁盘文件的情况。

另外, Turbo C2.0 提供了两类关于文件的函数。一类称做标准文件函数也称缓冲型文件函数, 这是 ANSI 标准定义的函数; 另一类叫非标准文件函数, 也称非缓冲型文件函数。这类函数最早公用于 UNIX 操作系统, 但现在 MS-DOS3.0 以上版本的操作系统也可以使用。下面分别进行介绍。

1.2.1 标准文件函数

标准文件函数主要包括文件的打开、关闭、读和写等函数。不象 BASIC 、FORTRAN 语方有顺序文件和随机文件之分, 在打开时就应按不同的方式确定。Turbo C2.0 并不区分这两种文件, 但提供了两组函数, 即顺序读写函数和随机读写函数。

一、文件的打开和关闭

任何一个文件在使用之前和使用之后, 必须要进行打开和关闭, 这是因为操作系统对于同时打开的文件数目是有限制的, DOS 操作系统中, 可以在 DEVICE.SYS 中定义允许同时打开的文件数 n(用 files=n 定义)。其中 n 为可同时打开的文件数, 一般 n<=20。因此在使用文件前应打开文件, 才可对其中的信息进行存取。用完之后需要关闭, 否则将会出现一些意想不到的错误。Turbo C2.0 提供了打开和关闭文件的函数。

1. fopen()函数

fopen 函数用于打开文件, 其调用格式为: FILE *fopen(char *filename, *type);

在介绍这个函数之前, 先了解一下下面的知识。

(1) 流(stream)和文件(file)

流和文件 在 Turbo C2.0 中是有区别的, Turbo C2.0 为编程者和被访问的设备之间提供了一层抽象的东西, 称之为"流", 而将具体的实际设备叫做文件。流是一个逻辑设备, 具有相同的行为。因此, 用来进行磁盘文件写的函数也同样 可以用来进行打印机的写入。在 Turbo C2.0 中有两种性质的流: 文字流(text stream)和二进制(binary stream)。对磁盘来说就是文本文件和二进制文件。本软件为了便于让读者易理解 Turbo C2.0 语言而没有对流和文件作特别区分。

(2) 文件指针 FILE

实际上 FILE 是一个新的数据类型。它是 Turbo C2.0 的基本数据类型的集合,称之为结构指针。有关结构的概念将在第四节中详细介绍, 这里只要将 FILE 理解为一个包括了文件管理有关信息的数据结构, 即在打开文件时必须先定义一个文件指针。

(3) 以后介绍的函数调用格式将直接写出形式参数的数据类型和函数返回值的数据类型。例如: 上面打开文件的函数, 返回一个文件指针, 其中形式参数有两个, 均为字符型变量(字符串数组或字符串指针)。本软件不再对函数的调用格式作详细说明。

现在再来看打开文件函数的用法。

fopen()函数中第一个形式参数表示文件名, 可以包含路径和文件名两部分。如:

"B:TEST.DAT"
"C:\\TC\\TEST.DAT"

如果将路径写成"C:TC\\TEST.DAT"是不正确的, 这一点要特别注意。 第二个形式参数表示打开文件的类型。关于文件类型的规定参见下表。

表 文件操作类型

字符	含义
"r"	打开文字文件只读

"w"	创建文字文件只写
"a"	增补, 如果文件不存在则创建一个
"r+"	打开一个文字文件读/写
"w+"	创建一个文字文件读/写
"a+"	打开或创建一个文件增补
"b"	二进制文件(可以和上面每一项合用)
"t"	文这文件(默认项)

如果要打开一个 CCDOS 子目录中, 文件名为 CLIB 的二进制文件, 可写成:

```
fopen("c:\\ccdos\\clib", "rb");
```

如果成功的打开一个文件, fopen()函数返回文件指针, 否则返回空指针(NULL)。由此可判断文件打开是否成功。

2. fclose()函数

fclose()函数用来关闭一个由 fopen()函数打开的文件, 其调用格式为: `int fclose(FILE *stream);`

该函数返回一个整型数。当文件关闭成功时, 返回 0, 否则返回一个非零值。可以根据函数的返回值判断文件是否关闭成功。

例 10:

```
#include<stdio.h>
main()
{
    FILE *fp;           /*定义一个文件指针*/
    int i;
    fp=fopen("CLIB", "rb"); /*打开当前目录名为 CLIB 的文件只读*/
    if(fp==NULL)         /*判断文件是否打开成功*/
        puts("File open error");/*提示打开不成功*/
    i=fclose(fp);        /*关闭打开的文件*/
    if(i==0)             /*判断文件是否关闭成功*/
        printf("O,K");    /*提示关闭成功*/
    else
        puts("File close error");/*提示关闭不成功*/
}
```

二、有关文件操作的函数

本节所讲的文件读写函数均是指顺序读写, 即读写了一条信息后, 指针自动加 1。下面分别介绍写操作函数和读操作函数。

1. 文件的顺序写函数

fprintf()、fputs()和 fputc()函数

函数 fprintf()、fputs()和 fputc()均为文件的顺序写操作函数, 其调用格式如下:

```
int fprintf(FILE *stream, char *format, <variable-list>);
int fputs(char *string, FILE *steam);
int fputc(int ch, FILE *steam);
```

上述三个函数的返回值均为整型量。fprintf() 函数的返回值为实际写入文件中的字符个数(字节数)。如果写错误, 则返回一个负数, fputs()函数返回 0 时表明将 string 指针所指的字符串写入文件中的操作成功, 返回非 0 时, 表明写操作失败。putc()函数返回一个向文件所写字符的值, 此时写操作成功, 否则返回 EOF(文件结束其值为-1, 在 stdio.h 中定义)表示写操作错误。

fprintf() 函数中格式化的规定与 printf() 函数相同, 所不同的只是 fprintf()函数是向文件中写入。而 printf()是向屏幕输出。

下面介绍一个例子, 运行后产生一个 test.dat 的文件。

例 11:

```
#include<stdio.h>
main()
{
    char *s="That's good news"); /*定义字符串指针并初始化*/
    int i=617; /*定义整型变量并初始化*/
    FILE *fp; /*定义文件指针*/
    fp=fopen("test.dat", "w"); /*建立一个文字文件只写*/
    fputs("Your score of TOEFL is", fp); /*向所建文件写入一串字符*/
    fputc(':', fp); /*向所建文件写冒号*/
    fprintf(fp, "%d\n", i); /*向所建文件写一整型数*/
    fprintf(fp, "%s", s); /*向所建文件写一字符串*/
    fclose(fp); /*关闭文件*/
}
```

用 DOS 的 TYPE 命令显示 TEST.DAT 的内容如下所示:

屏幕显示

```
Your score of TOEFL is: 617
That's good news
```

2. 文件的顺序读操作函数

fscanf()、fgets()和 fgetc()函数

函数 fscanf()、fgets()和 fgetc()均为文件的顺序读操作函数, 其调用格式 如下:

```
int fscanf(FILE *stream, char *format, <address-list>);
char fgets(char *string, int n, FILE *stream);
int fgetc(FILE *stream);
```

fscanf()函数的用法与 scanf()函数相似, 只是它是从文件中读到信息。fscanf()函数的返回值为 EOF(即-1), 表明读错误, 否则读数据成功。fgets()函数从文件中读取至多 n-1 个字符(n 用来指定字符数), 并把它们放入 string 指向的字符串中, 在读入之后自动向字符串末尾加一个空字符, 读成功返回 string 指针, 失败返回一个空指针。fgetc()函数返回文件当前位置的一个字符, 读错误时返回 EOF。

下面程序读取例 11 产生的 test.dat 文件, 并将读出的结果显示在屏幕上。

例 12

```
#include<stdio.h>
main()
{
```

```
char *s, m[20];
int i;
FILE *fp;
fp=fopen("test.dat", "r");    /*打开文字文件只读*/
fgets(s, 24, fp);             /*从文件中读取 23 个字符*/
printf("%s", s);              /*输出所读的字符串*/
fscanf(fp, "%d", &i);          /*读取整型数*/
printf("%d", i);              /*输出所读整型数*/
putchar(fgetc(fp));           /*读取一个字符同时输出*/
fgets(m, 17, fp);             /*读取 16 个字符*/
puts(m);                      /*输出所读字符串*/
fclose(fp);                   /*关闭文件*/
getch();                      /*等待任一键*/
}
```

运行后屏幕显示:
Your score of TOEFL is: 617
That's good news

如果将上例中 `fscanf(fp, "%d", &i)` 改为 `fscanf(fp, "%s", m)`, 再将其后的输出语句改为 `printf("%s", m)`, 则可得出同样的结果。由此可见 Turbo C2.0 中只要是读文字文件, 则不论是字符还是数字都将按其 ASCII 值处理。另外还要说明的一点就是 `fscanf()` 函数读到空白符时, 便自动结束, 在使用时要特别注意。

3. 文件的随机读写

有时用户想直接读取文件中间某处的信息, 若用文件的顺序读写必须从文件头开始直到要求的文件位置再读, 这显然不方便。Turbo C2.0 提供了一组文件的随机读写函数, 即可以将文件位置指针定位在所要求读写的地方直接读写。

文件的随机读写函数如下:

```
int fseek (FILE *stream, long offset, int fromwhere);
int fread(void *buf, int size, int count, FILE *stream);
int fwrite(void *buf, int size, int count, FILE *stream);
long ftell(FILE *stream);
```

`fseek()` 函数的作用是将文件的位置指针设置到从 `fromwhere` 开始的第 `offset` 字节的位置上, 其中 `fromwhere` 是下列几个宏定义之一:

文件位置指针起始计算位置 `fromwhere`

符号常数	数值	含义
SEEK_SET	0	从文件开头
SEEK_CUR	1	从文件指针的现行位置
SEEK_END	2	从文件末尾

`offset` 是指文件位置指针从指定开始位置(`fromwhere` 指出的位置)跳过的字节数。它是一个长整型量, 以支持大于 64K 字节的文件。`fseek()` 函数一般用于对二进制文件进行操作。当 `fseek()` 函数返回 0 时表明操作成功, 返回非 0 表示失败。

下面程序从二进制文件 `test_b.dat` 中读取第 8 个字节。

例 13:

```
#include<stdio.h>
main()
{
    FILE *fp;
    if((fp=fopen("test_b.dat", "rb"))==NULL)
    {
        printf("Can't open file");
        exit(1);
    }
    fseek(fp, 8, 1, SEEK_SET);
    fgetc(fp);
    fclose(fp);
}
```

fread()函数是从文件中读 count 个字段, 每个字段长度为 size 个字节, 并把 它们存放到 buf 指针所指的缓冲器中。

fwrite()函数是把 buf 指针所指的缓冲器中, 长度为 size 个字节的 count 个字段写到 stream 指向的文件中去。随着读和写字节数的增大, 文件位置指示器也增大, 读多少个字节, 文件位置指示器相应也跳过多少个字节。读写完毕函数返回所读和所写的字段个数。

ftell()函数返回文件位置指示器的当前值, 这个值是指示器从文件头开始算起的字节数, 返回的数为长整型数, 当返回-1 时, 表明出现错误。

下面程序把一个浮点数组以二进制方式写入文件 test_b.dat 中。

例 14:

```
#include <stdio.h>
main()
{
    float f[6]={3.2, -4.34, 25.04, 0.1, 50.56, 80.5};
    /*定义浮点数组并初始化*/

    int i;
    FILE *fp;
    fp=fopen("test_b.dat", "wb"); /*创建一个二进制文件只写*/
    fwrite(f, sizeof(float), 6, fp); /*将 6 个浮点数写入文件中*/
    fclose(fp); /*关闭文件*/
}
```

下面例子从 test_b.dat 文件中读 100 个整型数, 并把它们放到 dat 数组中。

例 15:

```
#include <stdio.h>
main()
{
    FILE *fp;
    int dat[100];
    fp=fopen("test_b.dat", "rb"); /*打开一个二进制文件只读*/
    if(fread(dat, sizeof(int), 100, fp)!=100)
        /*判断是否读了 100 个数*/
    {
        if(feof(fp))
```

```
printf("End of file"); /*不到 100 个数文件结束*/
else
printf("Read error"); /*读数错误*/
fclose(fp); /*关闭文件*/
}
```

注意:

当用标准文件函数对文件进行读写操作时, 首先将所读写的内容放进缓冲区, 写函数只对输出缓冲区进行操作, 读函数只对输入缓冲区进行操作。例如向一个文件写入内容, 所写的内容将首先放在输出缓冲区中, 直到输出缓冲区存满或使用 `fclose()` 函数关闭文件时, 缓冲区的内容才会写入文件中。若无 `fclose()` 函数, 则不会向文件中存入所写的内容或写入的文件内容不全。有一个对缓冲区 进行刷新的函数, 即 `fflush()`, 其调用格式为:

```
int fflush(FILE *stream);
该函数将输出缓冲区的内容实际写入文件中, 而将输入缓冲区的内容清除掉
```

4. `feof()`和 `rewind()`函数

这两个函数的调用格式为:

```
int feof(FILE *stream);
int rewind(FILE *stream);
```

`feof()`函数检测文件位置指示器是否到达了文件结尾, 若是则返回一个非 0 值, 否则返回 0。这个函数对二进制文件操作特别有用, 因为二进制文件中, 文件结尾标志 EOF 也是一个合法的二进制数, 只简单的检查读入字符的值来判断文件是否结束是不行的。如果那样的话, 可能会造成文件未结尾而被认为结尾, 所以就必须有 `feof()`函数。

下面的这条语句是常用的判断文件是否结束的方法。

```
while(!feof(fp))
fgetc(fp);
```

`while` 为循环语句, 将在下面介绍。

`rewind()`函数用于把文件位置指示器移到文件的起点处, 成功时返回 0, 否则, 返回非 0 值。

1.2.2 非标准文件函数

这类函数最早用于 UNIX 操作系统, ANSI 标准未定义, 但有时也经常用到,DOS 3.0 以上版本支持这些函数。它们的头文件为 `io.h`。

一、文件的打开和关闭

1. `open()`函数

`open()`函数的作用是打开文件, 其调用格式为:

```
int open(char *filename, int access);
```

该函数表示按 `access` 的要求打开名为 `filename` 的文件, 返回值为文件描述字,其中 `access` 有两部分内容: 基本模式和修饰符, 两者用 " ("或")"方式连接。修饰符可以有多个, 但基本模式只能有一个。`access` 的规定如表 3-2。

表 3-2 access 的规定

基本模式	含义	修饰符	含 义
O_RDONLY	只读	O_APPEND	文件指针指向末尾

O_WRONLY	只写	O_CREAT	文件不存在时创建文件,
			属性按基本模式属性
O_RDWR	读写	O_TRUNC	若文件存在, 将其长度
			缩为 0, 属性不变
		O_BINARY	打开一个二进制文件
		O_TEXT	打开一个文字文件

open()函数打开成功, 返回值就是文件描述字的值(非负值), 否则返回-1。

2. close()函数

close()函数的作用是关闭由 open()函数打开的文件, 其调用格式为:

```
int close(int handle);
```

该函数关闭文件描述字 handle 相连的文件。

二、读写函数

1. read()函数

read()函数的调用格式为:

```
int read(int handle, void *buf, int count);
```

read()函数从 handle(文件描述字)相连的文件中, 读取 count 个字节放到 buf 所指的缓冲区中, 返回值为实际所读字节数, 返回-1 表示出错。返回 0 表示文件结束。

2. write()函数

write()函数的调用格式为:

```
int write(int handle, void *buf, int count);
```

write()函数把 count 个字节从 buf 指向的缓冲区写入与 handle 相连的文件中, 返回值为实际写入的字节数。

三、随机定位函数

1. lseek()函数

lseek()函数的调用格式为:

```
int lseek(int handle, long offset, int fromwhere);
```

该函数对与 handle 相连的文件位置指针进行定位, 功能和用法与 fseek() 函数相同。

2. tell()函数

tell()函数的调用格式为:

```
long tell(int handle);
```

该函数返回与 handle 相连的文件现生位置指针, 功能和用法与 ftell()相同

C 语言高效编程的几招

编写高效简洁的 C 语言代码, 是许多软件工程师追求的目标。本文就工作中的一些体会和经验做相关的阐述, 不对的地方请各位指教。

第 1 招：以空间换时间

计算机程序中最大的矛盾是空间和时间的矛盾，那么，从这个角度出发逆向思维来考虑程序的效率问题，我们就有了解决问题的第 1 招 - 以空间换时间。

例如：字符串的赋值。

方法 A，通常的办法：

```
#define LEN 32

char string1 [LEN];

memset (string1,0,LEN);

strcpy (string1," This is an example!!")
```

方法 B:

```
const char string2[LEN]=" This is an example!"

char*cp;

cp=string2;
```

(使用的时候可以直接用指针来操作。)

从上面的例子可以看出，A 和 B 的效率是不能比的。在同样的存储空间下，B 直接使用指针就可以操作了，而 A 需要调用两个字符函数才能完成。B 的缺点在于灵活性没有 A 好。在需要频繁更改一个字符串内容的时候，A 具有更好的灵活性；如果采用方法 B，则需要预存许多字符串，虽然占用了大量的内存，但是获得了程序执行的高效率。

如果系统的实时性要求很高，内存还有一些，那我推荐你使用该招数。

该招数的边招 - 使用宏函数而不是函数。举例如下：

方法 C:

```
#define bwMCDR2_ADDRESS 4

#define bsMCDR2_ADDRESS 17

int BIT_MASK (int _bf)

{

    return ((IU<<(bw##_bf))-1)<<(bs##_bf);
```

```

}

void SET_BITS(int_dst,int_bf,int_val)
{
    _dst=(_dst & ~(BIT_MASK(_bf)))| (((_val)<<(bs##_bf))&(BIT_MASK(_bf)))
}

SET_BITS(MCDR2,MCDR2_ADDRESS,RegisterNumber);

```

方法 D:

```

#define bwMCDR2_ADDRESS 4

#define bsMCDR2_ADDRESS 17

#define bmMCDR2_ADDRESS BIT_MASK (MCDR2_ADDRESS)

#define BIT_MASK(_bf)((1U<<(bw##_bf))-1)<< (bs##_bf)

#define SET_BITS(_dst,_bf,_val)\
(((_dst)=((_dst)&~(BIT_MASK(_bf)))| (((_val)<<(bs##_bf))&(BIT_MASK(_bf))))

SET_BITS(MCDR2,MCDR2_ADDRESS,RegisterNumber);

```

函数和宏函数的区别就在于, 宏函数占用了大量的空间, 而函数占用了时间。大家要知道的是, 函数调用是要使用系统的栈来保存数据的, 如果编译器里有栈检查选项, 一般在函数的头会嵌入一些汇编语句对当前栈进行检查; 同时, CPU 也要在函数调用时保存和恢复当前的现场, 进行压栈和弹栈操作, 所以, 函数调用需要一些 CPU 时间。而宏函数不存在这个问题。宏函数仅仅作为预先写好的代码嵌入到当前程序, 不会产生函数调用, 所以仅仅是占用了空间, 在频繁调用同一个宏函数的时候, 该现象尤其突出。

D 方法是我看到的最好的置位操作函数, 是 ARM 公司源码的一部分, 在短短的三行内实现了很多功能, 几乎涵盖了所有的位操作功能。C 方法是其变体, 其中滋味还需大家仔细体会。

第 2 招: 数学方法解决问题

现在我们演绎高效 C 语言编写的第二招 - 采用数学方法来解决这个问题。

数学是计算机之母, 没有数学的依据和基础, 就没有计算机的发展, 所以在编写程序的时候, 采用一些数学方法会对程序的执行效率有数量级的提高。

举例如下, 求 1~100 的和。

方法 E

```
int I,j;
```

```
for (I=1; I<=100; I++)
```

```
{
```

```
    j+=I;
```

```
}
```

方法 F

```
int I;
```

```
I=(100*(1+100))/2
```

这个例子是我印象最深的一个数学用例, 是我的计算机启蒙老师考我的。当时我只有小学三年级, 可惜我当时不知道用公式 $N \times (N+1) / 2$ 来解决这个问题。方法 E 循环了 100 次才解决问题, 也就是说最少用了 100 个赋值、100 个判断、200 个加法(I 和 j); 而方法 F 仅仅用了 1 个加法、1 个乘法、1 次除法。效果自然不言而喻。所以, 现在我在编程的时候, 更多的是动脑筋找规律, 最大限度地发挥数学的威力来提高程序运行的效率。

第 3 招: 使用位操作

实现高效的 C 语言编写的第三招 - 使用位操作, 减少除法和取模的运算。

在计算机程序中, 数据的位是可以操作的最小数据单位, 理论上可以用“位运算”来完成所有的运算和操作。一般的位操作是用来控制硬件的, 或者做数据变换使用, 但是, 灵活的位操作可以有效地提高程序运行的效率。举例如下:

方法 G

```
int I,J;
```

```
I=257/8;
```

```
J=456%32;
```

方法 H

```
int I,J;
```

```
I=257>>3;
```

```
J=456-(456>>4<<4);
```

在字面上好象 H 比 G 麻烦了好多, 但是, 仔细查看产生的汇编代码就会明白, 方法 G 调用了基本的取模函数和除法函数, 既有函数调用, 还有很多汇编代码和寄存器参与运算; 而方法 H 则仅仅是几句相关的汇编, 代码更简洁、效率更高。当然, 由于编译器的不同, 可能效率的差距不大, 但是, 以我目前遇到的 MS C, ARM C 来看, 效率的差距还是不小。相关汇编代码就不在这里列举了。

运用这招需要注意的是, 因为 CPU 的不同而产生的问题。比如说, 在 PC 上用这招编写的程序, 并在 PC 上调试通过, 在移植到一个 16 位机平台上的时候, 可能会产生代码隐患。所以只有在一定技术进阶的基础下才可以使用这招。

第 4 招: 汇编嵌入

高效 C 语言编程的必杀技, 第四招 - 嵌入汇编。

“在熟悉汇编语言的人眼里, C 语言编写的程序都是垃圾”。这种说法虽然偏激了一些, 但是却有它的道理。汇编语言是效率最高的计算机语言, 但是, 不可能靠着它来写一个操作系统吧? 所以, 为了获得程序的高效率, 我们只好采用变通的方法 - 嵌入汇编、混合编程。

举例如下, 将数组一赋值给数组二, 要求每一个字节都相符。char string1[1024], string2[1024];

方法 I

```
int I;
```

```
for (I=0; I<1024; I++)
```

```
*(string2+I)=*(string1+I)
```

方法 J

```
#int I;
```

```
for(I=0; I<1024; I++)
```

```
*(string2+I)=*(string1+I);
```

```
#else
```

```
#ifdef _ARM_
```

```
_asm
```

```
{
```

```
    MOV R0,string1
```

```
    MOV R1,string2
```

```
    MOV R2,#0
```

```
loop:
```

```
    LDMIA R0!,[R3-R11]
```

```
    STMIA R1!, [R3-R11]

    ADD R2, R2, #8

    CMP R2, #400

    BNE loop

}

#endif
```

方法 I 是最常见的方法, 使用了 1024 次循环; 方法 J 则根据平台不同做了区分, 在 ARM 平台下, 用嵌入汇编仅用 128 次循环就完成了同样的操作。这里有朋友会说, 为什么不用标准的内存拷贝函数呢? 这是因为在源数据里可能含有数据为 0 的字节, 这样的话, 标准库函数会提前结束而不会完成我们要求的操作。这个例程典型应用于 LCD 数据的拷贝过程。根据不同的 CPU, 熟练使用相应的嵌入汇编, 可以大大提高程序执行的效率。

虽然是必杀技, 但是如果轻易使用会付出惨重的代价。这是因为, 使用了嵌入汇编, 便限制了程序的可移植性, 使程序在不同平台移植的过程中, 卧虎藏龙、险象环生! 同时该招数也与现代软件工程的思想相违背, 只有在迫不得已的情况下才可以采用。切记。

使用 C 语言进行高效率编程, 我的体会仅此而已。在此已本文抛砖引玉, 还请各位高手共同切磋。希望各位能给出更好的方法, 大家一起提高我们的编程技巧。

C 语言宏定义学习

1 防止一个头文件重复定义

格式如下

```
#ifndef COMDEF_H
#define COMDEF_H
//头文件内容
#endif
```

这个在很多的头文件的开头都有看到, 就是弄不明白, 什么叫重复定义??? 试个程序看看

• 例题 1 test1.c

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("lsdkfla\n");
}
```

• 例题 2 test2.c

```
#include <stdio.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("lsdkfla\n");
}
```

现用动态文件的方法编辑

```
#gcc test1.c -Wall
```

编译结构, 没有错误, 肯定的。。。

```
#gcc test2.c -Wall
```

哦, 也没有错误??? 那个有什么用呢!!! 不理解, 是不是能在 test2.c 编译的结构中 a.out 中会有两个标准库 stdio.h 的库连接呢, 我编译静态。编译成静态文件:

```
#gcc -Wall -static test1.c -o test1
```

编译结果, 没有错误!

```
#gcc -Wall -static test2.c -o test2
```

编译结构, 也没有错误阿。。。那用 ls -l 查看文件大小, 哦, 一样大。。。那个功能是什么作用呢, 想了好久。。。突然想起来自己写一个头文件看看

• 例题 3 test.h main.c

```
/*test.h*/
#include <stdio.h>
int test(int a)
{
    printf("test ...\n");
}
```

```
/*main.c*/
#include "test.h"
int main(int argc, char *argv[])
{
```

```
    test(4);  
}
```

编译方法:

```
#gcc -Wall main.c
```

编译结果, 没有错误。。。 继续试验。。。。。

• 例题 4 test.h main.c

```
/*test.h*/
```

```
#include <stdio.h>  
int test(int a)  
{  
    printf("test ...\n");  
}
```

```
/*main.c*/
```

```
#include "test.h"  
#include "test.h"  
int main(int argc, char *argv[])  
{  
    test(4);  
}
```

编译结果: 终于出错了阿。。。呵呵。。。

```
test.h:3: 错误:  'test'  重定义  
test.h:3: 错误:  'test'  的上一个定义在此
```

终于弄明白了。。。。那么我加上宏定义看看。。。。 例题 5 test.h main.c /*test.h*/

```
#ifndef _TEST_H  
#define _TEST_H  
#include <stdio.h>  
int test(int a)  
{  
    printf("test ...\n");  
}  
#endif /*结束_TEST_H*/
```

```
/*main.c*/
```

```
#include "test.h"  
#include "test.h"
```



```
int main(int argc,char *argv[])
{
    test(4);
}
```

编译方法:

```
#gcc -Wall main.c
```

编译结果: 程序正常完成编译, 没有出错信息。。。

2 防止由于各种平台和编译器的不同, 而产生的类型字节数

```
typedef unsigned char    boolean;    /* Boolean value type. */
typedef unsigned long int uint32;    /* Unsigned 32 bit value */
typedef unsigned short   uint16;    /* Unsigned 16 bit value */
typedef unsigned char     uint8;    /* Unsigned 8 bit value */
typedef signed long int   int32;    /* Signed 32 bit value */
typedef signed short      int16;    /* Signed 16 bit value */
typedef signed char       int8;    /* Signed 8 bit value */
//下面的不建议使用
typedef unsigned char     byte;    /* Unsigned 8 bit value type. */
typedef unsigned short    word;    /* Unsinged 16 bit value type. */
typedef unsigned long     dword;    /* Unsigned 32 bit value type. */
```

3 得到指定地址上的一个字节或字

取得指定地址的一个字

```
#define MEM_B(x) (*((byte *)(x)))
```

取得指定地址的一个字

```
#define MEM_W(x) (*((word *)(x)))
```

例题 1 test.c

```
#include <stdio.h>
#define MEM_B(x) (*((byte *)(x)))
int main(int argc,char *argv[])
{
    char a='c';
    printf("&a==%c \n",MEM_B(&a));
}
```

编译方法:

```
#gcc -Wall test.c
```

执行结果:

```
&a==c
```

4 求最大值和最小值

定义方法:

```
#define MAX( x, y ) ( ((x) > (y)) ? (x) : (y) )  
#define MIN( x, y ) ( ((x) < (y)) ? (x) : (y) )
```

例题: test.c

```
#include <stdio.h>  
#define MAX(x,y) (((x)>(y))?(x):(y))  
int main(int argc,char *argv[])  
{  
    printf("max==%d \n",MAX(4,5));  
}
```

编译方式:

```
#gcc -Wall test.c
```

执行结果:

```
#max==5
```

5 得到一个 field 在结构体(struct)中的偏移量

定义方法:

```
#define FPOS( type, field ) ( (dword) &((( type *) )0)-> field )
```

这个是最郁闷的一个, 终于搞定了。。。郁闷的 typedef 和运算规则 例题 test.c

```
#include <stdio.h>  
typedef unsigned long dword;  
#define FPOS( type,field) (dword)&(((type *)0)->field)  
typedef struct test  
{
```

```
    int a;
    int b;
    char c ;
}d;
int main(int argc,char *argv[])
{
    printf("offset==%d \n",FPOS(d,b));
}
```

编译方法:

```
#gcc -Wall test.c
```

执行结果

```
#offset==0x4
```

“&”的运算级别小于“->”运算级别

特别推荐 C 语言运算符表

6 得到一个结构体中 field 所占用的字节数

定义方法:

```
#define FSIZ(type,field) sizeof(((type *) 0)->field )
```

例题: test.c

```
#include <stdio.h>
#define FSIZ(type,field) sizeof(((type *) 0)->field)
struct test
{
    int a;
    int b;
    char c;
};
int main(int argc,char *argv[])
{
    printf("sizeof==%d \n",FSIZ(struct test,b);
}
```

编译方法:

```
#gcc -Wall test.c
```

执行结果:

```
#sizeof=4
```

7 按照 LSB 格式把两个字节转化为一个 Word

提示: LSB 和 MSB 的定义

- LSB(Least Significant Bit)是 “意义最小的字位”。
- MSB(Most Significant Bit)是 “意义最大的字位”

定义格式

```
#define FLIPW(ray) (((word)(ray)[0])* 256)+(ray)[1])
```

例题 test.c

```
#include <stdio.h>
typedef unsigned short word;
#define FLIPW(ray) (((word)(ray)[0])* 256)+(ray)[1])
int main(int argc, char *argv[])
{
    char test[2]={0x06,0x07};
    printf("LSB==%d \n",FLIPW(test));
}
```

编译方法:

```
#gcc -Wall test.c
```

执行结构

```
#1541
```

这个数是 $6 \times 256 + 5 = 1541$ 突然见有了个想法, 可以不可以变成大字节的顺序, 看一下。。。 例题: test.c

```
1 include <stdio.h>
```

```
typedef unsigned short word;
#define FLIPW(ray) (((word)(ray)[1])* 256)+(ray)[2])
int main(int argc, char *argv[])
{
    char test[2]={0x06,0x07};
    printf("LSB==%d \n",FLIPW(test));
}
```

编译方法:

```
#gcc -Wall test.c
```

执行结构

```
#1286
```

这个数是 $5*256+6=1286$

8 按照 LSB 格式把一个 Word 转化为两个字节

定义方式:

```
#define FLOPW( ray, val ) (ray)[0] = ((val) / 256); (ray)[1] = ((val) & 0xFF)
```

[[编辑](#)]

9 参考资料

资料 1: c 语言的宏定义技巧

<http://www.ednchina.com/blog/levension/21415/message.aspx>

资料 2:

<http://gawaine.itpub.net/>

资料 3:

<http://www.yesky.com/309/1853309.shtml>

运算符	解释	结合方式
() [] -> .	括号（函数等），数组，两种结构成员访问	由左向右
! ~ ++ -- + - * & (类型) sizeof</td>	否定，按位否定，增量，减量，正负号，<p>间接，取地址，类型转换，求大小	由右向左
* / %	乘，除，取模	由左向右
+ -	加，减	由左向右
<< >>	左移，右移	由左向右
< <= >= >	小于，小于等于，大于等于，大于	由左向右
== !=	等于，不等于	由左向右

&	按位与	由左向右
^	按位异或	由左向右
	按位或	由左向右
&&	逻辑与	由左向右
	逻辑或	由左向右
?:	条件	由右向左
= += -= *= /= <p>&= ^= = <<= >>=	各种赋值	由右向左
,	逗号 (顺序)	由左向右

Nand Flash 与 Nor Flash

Nand Flash 与 Nor Flash Nand Flash 与 Nor Flash 经常在一些地方被提到, 一直没认真去理解它们的区别, 因此, 今天花了一段时间仔细理解了一下, 下面把我的笔记放在这里:)

1、NOR 的特点是芯片内执行(XIP,eXecute In Place), 这样应用程序可以直接在 flash 闪存内运行, 不必再把代码读到系统 RAM 中。优点是可以直接从 FLASH 中运行程序, 但是工艺复杂, 价格比较贵, NOR 的传输效率很高, 在 1~4MB 的小容量时具有很高的成本效益, 但是很低的写入和擦除速度大大影响了它的性能。

NAND 结构能提供极高的单元密度, 可以达到高存储密度, 并且写入和擦除的速度也很快。应用 NAND 的困难在于 flash 的管理和需要特殊的系统接口。优点: 大存储容量, 而且便宜。缺点, 就是无法寻址直接运行程序, 只能存储数据。另外 NAND FLASH 非常容易出现坏区, 所以需要有校验的算法。

任何 flash 器件的写入操作只能在空或已擦除的单元内进行

(1) 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 1。

(2) 擦除 NOR 器件时是以 64~128KB 的块进行的, 执行一个写入/擦除操作的时间为 5s, NORFLASHSECTOR 擦除时间视品牌、大小不同而不同, 比如, 4MFLASH, 有的 SECTOR 擦除时间为 60ms, 而有的需要最大 6S。与此相反, 擦除 NAND 器件是以 8~32KB 的块进行的, 执行相同的操作最多只需要 4ms

(3) 当选择存储解决方案时, 设计师必须权衡以下的各项因素。

- NOR 的读速度比 NAND 稍快一些。
- NAND 的写入速度比 NOR 快很多。
- NAND 的 4ms 擦除速度远比 NOR 的 5s 快。
- 大多数写入操作需要先进行擦除操作。
- NAND 的擦除单元更小, 相应的擦除电路更少。

(4) 接口差别

NORflash 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。NAND 读和写操作采用 512 字节的块, 这一点有点像硬盘管理此类操作, 因此, 基于 NAND 的存储器就可以取代硬盘或其他块设备。

(5) 容量差别:

NORflash 占据了容量为 1~16MB 闪存市场的大部分, 而 NANDflash 只是用在 8~128MB 的产品当中, 这也说明 NOR 主要应用在代码存储介质中, NAND 适合于数据存储。

(6) 可靠性和耐用性

—寿命(耐用性)

在 NAND 闪存中每个块的最大擦写次数是一百万次, 而 NOR 的擦写次数是十万次。NAND 存储器除了具有 10 比 1 的块擦除周期优势, 典型的 NAND 块尺寸要比 NOR 器件小 8 倍, 每个 NAND 存储器块在给定的时间内的删除次数要少一些。

—位交换

所有 flash 器件都受位交换现象的困扰。位真的改变了, 就必须采用错误探测/错误更正(EDC/ECC)算法。位反转的问题更多见于 NAND 闪存, 在使用 NAND 闪存的时候, 应使用 EDC/ECC 算法。用 NAND 存储多媒体信息时倒不是致命的。当然, 如果用本地存储设备来存储操作系统、配置文件或其他敏感信息时, 必须使用 EDC/ECC 系统以确保可靠性。

—坏块处理

NAND 器件中的坏块是随机分布的, NAND 器件需要对介质进行初始化扫描以发现坏块, 并将坏块标记为不可用。在已制成的器件中, 如果通过可靠的方法不能进行这项处理, 将导致高故障率。

(7) 易于使用

可以非常直接地使用基于 NOR 的闪存。在使用 NAND 器件时, 必须先写入驱动程序, 才能继续执行其他操作。向 NAND 器件写入信息需要相当的技巧, 因为设计师绝不能向坏块写入, 这就意味着在 NAND 器件上自始至终都必须进行虚拟映射。

(8) 软件支持

在 NOR 器件上运行代码不需要任何的软件支持, 在 NAND 器件上进行同样操作时, 通常需要驱动程序, 也就是内存技术驱动程序(MTD), NAND 和 NOR 器件在进行写入和擦除操作时都需要 MTD。

使用 NOR 器件时所需要的 MTD 要相对少一些, 许多厂商都提供用于 NOR 器件的更高级软件, 其中包括 M-System 的 TrueFFS 驱动, 该驱动被 WindRiverSystem、Microsoft、QNXSoftwareSystem、Symbian 和 Intel 等厂商所采用。驱动还用于对 DiskOnChip 产品进行仿真和 NAND 闪存的管理, 包括纠错、坏块处理和损耗平衡。

(9) 在掌上电脑里要使用 NAND FLASH 存储数据和程序, 但是必须有 NOR FLASH 来启动。除了 SAMSUNG 处理器, 其他用在掌上电脑的主流处理器还不支持直接由 NAND FLASH 启动程序。因此, 必须先一片小的 NOR FLASH 启动机器, 在把 OS 等软件从 NAND FLASH 载入 SDRAM 中运行才行

NOR 和 NAND 是市场上两种主要的非易失闪存技术, 分别由 Intel 在 1988 年和东芝在 1989 年开发。NOR 的特点是芯片内执行(XIP, eXecute In Place), 这样运用程序可以直接在 flash 闪存内运行, 不必把代码读入到系统 RAM 中。NOR 的传输效率高, 但是写入和擦除速度很慢。NAND 结构提供高的单元密度, 写入和擦除的速度很快。苦难在于 flash 的管理和需要特殊的系统接口。Flash 可以对称块为块的存储器单元块进行擦写和再编程。任何 flash 器件的写入操作只能在空或者已擦除的单元内进行。故写入操作前必须先执行擦除。NAND 器件擦除简单, 而 NOR 则要求进行擦除前先将目标快内所有的位写为 0。NOR 和 NAND 有如下区别:

1. NOR 的读速度比 NAND 稍快一些
2. NAND 的写入速度比 NOR 快很多
3. NAND 的 4ms 擦除速度比 NOR 的 5s 快
4. 大多数写入操作需要先进行擦除操作
5. NAND 的擦除单元更小, 相应的擦除电路更少。

接口上的差别在于: 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 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。NAND 读和写操作采用 512 字节的块, 有点像磁盘管理此类操作, 很自然, 基于 NAND 的存储器就可以取代硬盘或其他块设备。NAND 在容量和成本方面占据优势。NOR 主要运用在代码存储介质中, NAND 适用于数据存储, NAND 在 CompactFlash, Secure Digital, PC Cards 和 MMC 存储卡市场上份额最大。NAND 的擦除次数比 NOR 多; NOR 和 NAND 都受位交换现象的困扰。使用 NAND 时, 一般需要采用错误探测/错误更正(EDC/ECC)算法。对于使用本地存储设备来存储操作系统、配置

文件或其他敏感信息时, 必须使用 EDC/ECC 系统以确保可靠性。

NOR Flash 容易使用, 可以像其他存储器那样连接, 并可以在上面直接运行代码。NAND 在需要 I/O 接口, 使用时, 必须先写入驱动程序, 才能继续执行其他操作, 向 NAND 写入信息不能向坏块写入, 这意味 NAND 器件上需要进行虚拟映射。

NOR 上运行代码不需要任何软件支持, 在 NAND 器件上进行同样的操作时, 需要驱动程序, 即内存技术驱动程序(MTD), NAND 和 NOR 器件在进行写入和擦除操作时都需要 MTD。NOR 使用 MTD 相对少一些, 许多厂商都提供用于 NOR 器件的更高级的软件, 包括 M-System 的 TrueFFS 驱动, 该驱动被 Wind River System, Microsoft, QNX Software System, Symbian 和 Intel 等采用。

近年来, NAND 的使用越来越多。据 iSuppli 公司估计, 2005 年第一季度, NAND 闪存受益首次超过 NOR 闪存。消费电子对低成本固态存储器贪婪的需求将继续推动整个产业对 NAND 的需求。据估计, 2007 年, NAND 将占到闪存的 61%。目前 NAND 的需求主要在移动闪存卡, USB 驱动器和 MP3 播放器。而 NOR 闪存主要集中在无线和嵌入式产品两大广阔市场, 其中高密度和高平均售价(ASP)NOR 闪存集中于移动电话行业的推动, 其主要供应商有 Intel, Spansion 和 ST。近年来, NAND 一直鼓吹自己是移动电话中 NOR 的替代者, 但是 iSuppli 分析师认为, 大批 NOR 供应商推出移动电话领域, 并没有足够的 NAND 来填补。因为 NAND 供应商目前生产针对移动存储市场的高密度器件, 对移动电话使用较低密度 NAND(即 128Mb 到 1Gb)的供给正在萎缩, 一旦可能, 低密度 NAND 器件会抬高价格。

Nand Flash 读写操作

正如硬盘的盘片被分为磁道, 每个磁道又分为若干扇区, 一块 nand flash 也分为若干 block, 每个 block 分为如干 page。一般而言, block、page 之间的关系随着芯片的不同而不同, 典型的分配是这样的:

1block = 32page

1page = 512bytes(datafield) + 16bytes(oob)

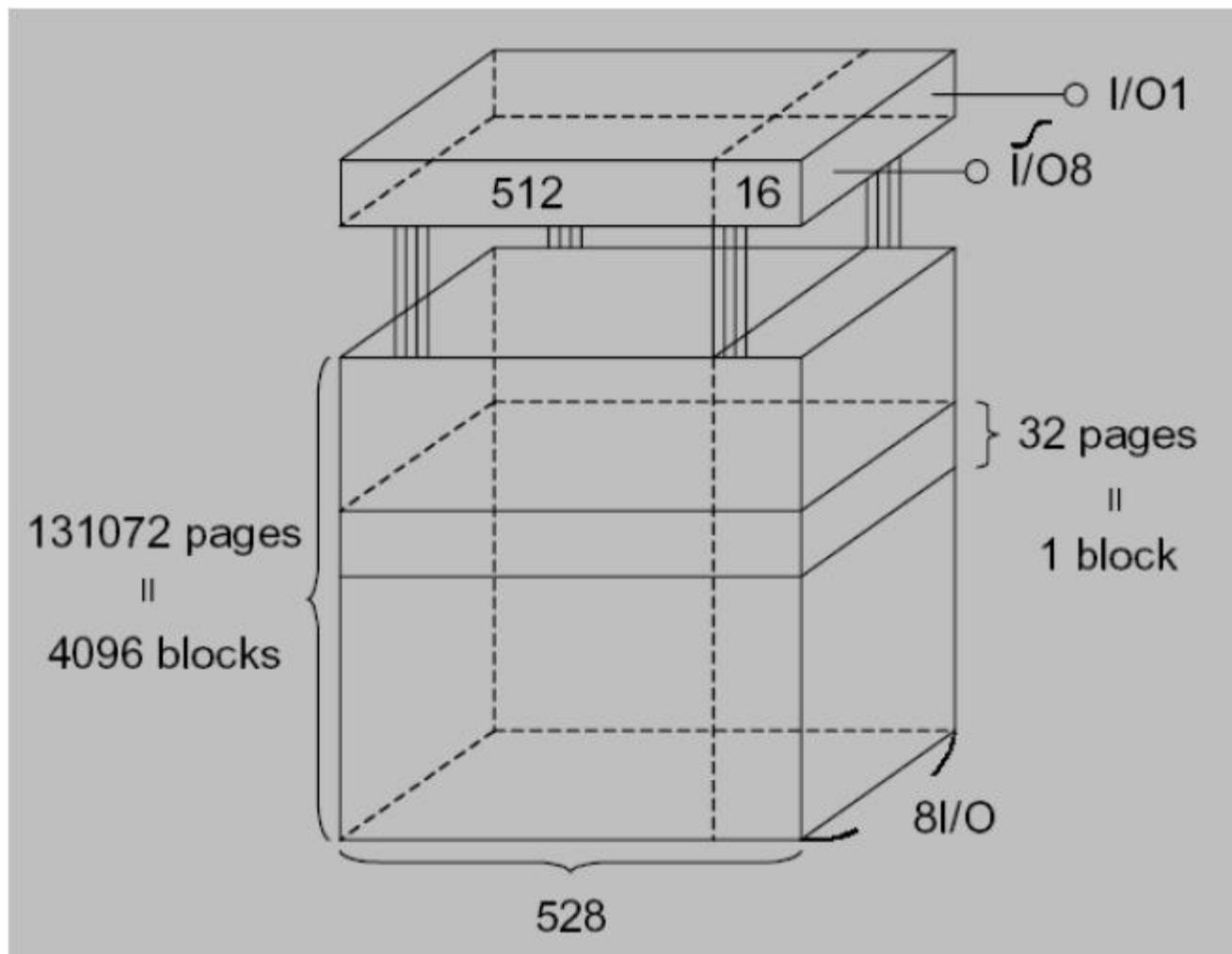


图 1 典型 Nand Flash 组织结构

需要注意的是, 对于 flash 的读写都是以一个 page 开始的, 但是在读写之前必须进行 flash 的擦写, 而擦写则是以一个 block 为单位的。同时必须提醒的是, 512bytes 理论上被分为 1st half 和 2nd half, 每个 half 各占 256 个字节。

我们讨论的 K9F1208U0B 总共有 4096 个 Blocks, 故我们可以知道这块 flash 的容量为 $4096 * (32 * 528) = 69206016 \text{ Bytes} = 66 \text{ MB}$

但事实上每个 Page 上的最后 16Bytes 是用于存储检验码和其他信息用的, 并不能存放实际的数据, 所以实际上我们可以操作的芯片容量为 $4096 * (32 * 512) = 67108864 \text{ Bytes} = 64 \text{ MB}$ 由上图所示, 1 个 Page 总共由 528 Bytes 组成, 这 528 个字节按顺序由上而下以列为单位进行排列 (1 列代表一个 Byte。第 0 行为第 0 Byte, 第 1 行为第 1 Byte, 以此类推, 每个行又由 8 个位组成, 每个位表示 1 个 Byte 里面的 1bit)。这 528 Bytes 按功能分为两大部分, 分别是 Data Field 和 Spare Field, 其中 Spare Field 占 528 Bytes 里的 16 Bytes, 这 16 Bytes 是用于在读写操作的

时候存放校验码用的, 一般不用做普通数据的存储区, 除去这 16Bytes, 剩下的512Bytes 便是我们用于存放数据用的 Data Field, 所以一个 Page 上虽然有528个 Bytes, 但我们只按 512Bytes 进行容量的计算。

读 命令有两个, 分别是 Read1, Read2其中 Read1用于读取 Data Field 的数据, 而 Read2则是用于读取 Spare Field 的数据。对于 Nand Flash 来说, 读操作的最小操作单位为 Page, 也就是说当我们给定了读取的起始位置后, 读操作将从该位置开始, 连续读取到本 Page 的最后一个 Byte 为止 (可以包括 Spare Field)

Nand Flash 的寻址

Nand Flash 的地址寄存器把一个完整的 Nand Flash 地址分解成 Column Address 与 Page Address. 进行寻址。

Column Address: 列地址。Column Address 其实就是指指定 Page 上的某个 Byte, 指定这个 Byte 其实也就是指定此页的读写起始地址。

Page Address: 页地址。由于页地址总是以512Bytes 对齐的, 所以它的低9位总是0。确定读写操作是在 Flash 上的哪个页进行的。

Read1命令

当我们得到一个 Nand Flash 地址 src_addr 时我们可以这样分解出 Column Address 和 Page Address

```
column_addr=src_addr%512;           // column address
page_address=(src_addr>>9);         // page address
```

也可以这么认为, 一个 Nand Flash 地址的 A0~A7是它的 column_addr, A9~A25是它的 Page Address。(注意地址位 A8并没有出现, 也就是 A8被忽略, 在下面你将了解到这是什么原因) Read1 命令的操作分为4个 Cycle, 发送完读命令00h 或01h (00h 与01h 的区别请见下文描述) 之后将分4个 Cycle 发送参数, 1st.Cycle 是 发送 Column Address。

2nd.Cycle , 3rd.Cycle 和4th.Cycle 则是指定 Page Address (每次向地址寄存器发送的数据只能是8位, 所以17位的 Page Address 必须分成3次进行发送 Read1的 命令里面出现了两个命令选项, 分别是00h 和01h。这里出现了两个读命令是否令你意识到什么呢? 是的, 00h 是用于读写1st half 的命令, 而01h 是用于读取2nd half 的命令。现在我可以结合上图给你说明为什么 K9F1208U0B 的 DataField 被分为2个 half 了。

如上文我所提及的, Read1的1st.Cycle 是发送 Column Address, 假设我现在指定的 Column Address 是0, 那么读操作将从此页的第0号 Byte 开始一直读取到此页的最后一个 Byte(包括 Spare Field), 如果我指定的 Column Address 是127, 情况也与前面一样, 但不知道你发现没有, 用于传递 Column Address 的数据线有8条 (I/00~I/07, 对应 A0~A7, 这也是 A8 为什么不出现在我们传递的地址位中), 也就是说我们能够指定的 Column Address 范围为 0~255, 但不要忘了, 1个 Page 的 DataField 是由512个 Byte 组成的, 假设现在我要指定读命令

从第256个字节处 开始读取此页, 那将会发生什么情景? 我必须把 Column Address 设置为 256, 但 Column Address 最大只能是255, 这就造成数据溢出。。。正是因为这个原因我们才把 Data Field 分为两个半区, 当要读取的起始地址 (Column Address) 在0~255内时我们用00h 命令, 当读取的起始地址是在256~511时, 则使用01h 命令. 假设现在我要指定从第256个 byte 开始读取此页, 那么我将这样发送命令串

```
column_addr=256;
NF_CMD=0x01; B           从2nd half 开始读取
NF_ADDR=column_addr&0xff; 1st Cycle
NF_ADDR=page_address&0xff; 2nd.Cycle
NF_ADDR=(page_address>>8)&0xff; 3rd.Cycle
NF_ADDR=(page_address>>16)&0xff; 4th.Cycle
```

其中 NF_CMD 和 NF_ADDR 分别是 NandFlash 的命令寄存器和地址寄存器的地址解引用, 我一般这样定义它们,

```
#define rNFCMD (*(volatile unsigned char *)0x4e000004) //NADD Flash command
#define rNFADDR (*(volatile unsigned char *)0x4e000008) //NAND Flash address
```

事实上, 当 NF_CMD=0x01时, 地址寄存器中的第8位 (A8) 将被设置为1 (如上文分析, A8 位不在我们传递的地址中, 这个位其实就是硬件电路根据 01h 或是00h 这两个命令来置高位或是置低位), 这样我们传递 column_addr 的值256随然由于数据溢出变为1, 但 A8位已经由于 NF_CMD =0x01的关系被置为1了, 所以我们传到地址寄存器里的值变成了

```
A0 A1 A2 A3 A4 A5 A6 A7 A8
1   0   0   0   0   0   0   0   1
```

这8个位所表示的正好是256, 这样读操作将从此页的第256号 byte (2nd half 的第0号 byte) 开始读取数据。 nand_flash.c 中包含3个函数

```
void nf_reset(void);
```

```
void nf_init(void);
```

```
void nf_read(unsigned int src_addr,unsigned char *desc_addr,int size);
```

nf_reset() 将被 nf_init() 调用。nf_init() 是 nand_flash 的初始化函数, 在对 nand flash 进行任何操作之前, nf_init() 必须被调用。

nf_read(unsigned int src_addr,unsigned char *desc_addr,int size);为读函数, src_addr 是 nand flash 上的地址, desc_addr 是内存地址, size 是读取文件的长度。

在 nf_reset 和 nf_read 函数中存在两个宏

```
NF_nFCE_L();
```

```
NF_nFCE_H();
```

你可以看到当每次对 Nand Flash 进行操作之前 NF_nFCE_L() 必定被调用, 操作结束之时 NF_nFCE_H() 必定被调用。这两个宏用于启动和关闭 Flash 芯片的工作 (片选/取消片选)。至于 nf_reset() 中的

```
rNFCNF=(1<<15)|(1<<14)|(1<<13)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0);
```

这一行代码是对 NandFlash 的控制寄存器进行初始化配置, rNFCNF 是 Nand Flash 的配置寄存器, 各个位的具体功能请参阅 s3c2410数据手册。

现在举一个例子, 假设我要从 Nand Flash 中的第5000字节处开始读取1024个字节到内存的 0x30000000处, 我们这样调用 read 函数

```
nf_read(5000, 0x30000000, 1024);
```

我们来分析5000这个 src_addr.

根据

```
column_addr=src_addr%512;
```

```
page_address=(src_addr>>9);
```

我们可得出 column_addr=5000%512=392

```
page_address=(5000>>9)=9
```

于是我们可以知道5000这个地址是在第9页的第392个字节处, 于是我们的 nf_read 函数将这样发送命令和参数

```
column_addr=5000%512;
```

```
>page_address=(5000>>9);
```

```
NF_CMD=0x01;
```

从2nd half 开始读取

```
NF_ADDR= column_addr &0xff;
```

1st Cycle

```
NF_ADDR=page_address&0xff;
```

2nd. Cycle

```
NF_ADDR=(page_address>>8)&0xff;
```

3rd. Cycle

```
NF_ADDR=(page_address>>16)&0xff;
```

4th. Cycle

向 NandFlash 的命令寄存器和地址寄存器发送完以上命令和参数之后, 我们就可以从 rNFDATA 寄存器 (NandFlash 数据寄存器) 读取数据了.

我用下面的代码进行数据的读取.

```
for(i=column_addr;i<512;i++)
```

```
{
```

```
    *buf++=NF_RDDATA();
```

```
}
```

每当读取完一个 Page 之后, 数据指针会落在下一个 Page 的0号 Column(0号 Byte).

下面是源代码:

```
/*
```

```
    www.another-prj.com
```

```
    author: caiyuqing
```

本代码只属于交流学习, 不得用于商业开发

```
*/
```

```
#include "s3c2410.h"
```

```

#include "nand_flash.h"
static unsigned char seBuf[16]={0xff};
//-----
unsigned short nf_checkId(void)
{
    int i;
    unsigned short id;
    NF_nFCE_L();          //chip enable

    NF_CMD(0x90);          //Read ID
    NF_ADDR(0x0);
    for(i=0;i<10;i++);    //wait tWB(100ns)

    id=NF_RDDATA()<<8;    // Maker code(K9S1208V:0xec)
    id|=NF_RDDATA();      // Devide code(K9S1208V:0x76)

    NF_nFCE_H();          //chip enable
    return id;
}
//-----
static void nf_reset(void)
{
    int i;
    NF_nFCE_L();          //chip enable
    NF_CMD(0xFF);          //reset command
    for(i=0;i<10;i++);    //tWB = 100ns.
    NF_WAITRB();          //wait 200~500us;
    NF_nFCE_H();          //chip disable
}
//-----
void nf_init(void)
{
    rNFCONF=(1<<15)|(1<<14)|(1<<13)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0);
    //      1      1      1      1      1      xxx      r xxx,      r xxx
    //      En      r      r      ECCR      nFCE=H tACLS      tWRPH0      tWRPH1
    nf_reset();
}

```



```
//-----

void nf_read(unsigned int src_addr, unsigned char *desc_addr, int size)
{
    int i;
    unsigned int column_addr = src_addr % 512;          // column address
    unsigned int page_address = (src_addr >> 9);        // page address
    unsigned char *buf = desc_addr;
    while((unsigned int)buf < (unsigned int)(desc_addr) + size)
    {
        NF_nFCE_L();                                   // enable chip

        /*NF_ADDR 和 NF_CMD 为 nand_flash 的地址和命令寄存器的解引用*/
        if(column_addr > 255)                            // 2end halft
            NF_CMD(0x01);                                // Read2 command.    cmd 0x01: Read
command(start from 2end half page)
        else
            NF_CMD(0x00);                                // 1st halft?

        NF_ADDR(column_addr & 0xff);                     // Column Address
        NF_ADDR(page_address & 0xff);                     // Page Address
        NF_ADDR((page_address >> 8) & 0xff);               // ...
        NF_ADDR((page_address >> 16) & 0xff);              // ..
        for(i = 0; i < 10; i++);                          // wait tWB(100ns)//////??????
        NF_WAITRB();                                       // Wait tR(max 12us)

        // Read from main area
        for(i = column_addr; i < 512; i++)
        {
            *buf++ = NF_RDDATA();
        }
        NF_nFCE_H();                                     // disable chip
        column_addr = 0;
        page_address++;
    }
    return ;
}
```

作者: 蔡于清

www.another-prj.com

(约有修改) Trackback: <http://tb.blog.csdn.net/TrackBack.aspx?PostId=2060887>

The ASCII Chart

The ASCII Chart

The following chart contains the first 128 ASCII decimal, octal, hexadecimal and character codes.

Decimal	Octal	Hex	Character	Description
0	0	00	NUL	
1	1	01	SOH	start of header
2	2	02	STX	start of text
3	3	03	ETX	end of text
4	4	04	EOT	end of transmission
5	5	05	ENQ	enquiry
6	6	06	ACK	acknowledge
7	7	07	BEL	bell
8	10	08	BS	backspace
9	11	09	HT	horizontal tab
10	12	0A	LF	line feed
11	13	0B	VT	vertical tab
12	14	0C	FF	form feed
13	15	0D	CR	carriage return
14	16	0E	S0	shift out
15	17	0F	SI	shift in
16	20	10	DLE	data link escape
17	21	11	DC1	no assignment, but usually XON

18	22	12	DC2	
19	23	13	DC3	no assignment, but usually XOFF
20	24	14	DC4	
21	25	15	NAK	negative acknowledge
22	26	16	SYN	synchronous idle
23	27	17	ETB	end of transmission block
24	30	18	CAN	cancel
25	31	19	EM	end of medium
26	32	1A	SUB	substitute
27	33	1B	ESC	escape
28	34	1C	FS	file separator
29	35	1D	GS	group separator
30	36	1E	RS	record separator
31	37	1F	US	unit separator
32	40	20	SPC	space
33	41	21	!	
34	42	22	“	
35	43	23	#	
36	44	24	\$	
37	45	25	%	
38	46	26	&	
39	47	27	'	
40	50	28	(
41	51	29)	
42	52	2A	*	
43	53	2B	+	
44	54	2C	,	
45	55	2D	-	
46	56	2E	.	
47	57	2F	/	
48	60	30	0	
49	61	31	1	
50	62	32	2	
51	63	33	3	

52	64	34	4
53	65	35	5
54	66	36	6
55	67	37	7
56	70	38	8
57	71	39	9
58	72	3A	:
59	73	3B	;
60	74	3C	<
61	75	3D	=
62	76	3E	>
63	77	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U

86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D]
94	136	5E	^
95	137	5F	_
96	140	60	`
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w

120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	
125	175	7D	}
126	176	7E	~
127	177	7F	DEL delete

嵌入式 Linux 启动脚本整理

这几周一直在研究目标机上 Linux 系统启动过程, 先将涉及到的脚本内容及启动具体操作整理一下。

系统启动参数设定启动脚本为/linuxrc, 内容如下:

```
#!/bin/bash
```

```
echo "mount /etc as ramfs"
```

```
#/bin/mount -n -t ramfs ramfs /etc
```

```
#/bin/cp -a /mnt/etc/* /etc
```

```
echo "re-create the /etc/mtab entries"
```

```
# re-create the /etc/mtab entries
```

```
/bin/insmod /usr/sd_mod.ko
```

//加载驱动

```
/bin/insmod /usr/usb-storage.ko
```

```
# /bin/insmod /usr/videodev.o
```

```
/bin/insmod /usr/usbvideo.ko
```

```
/bin/insmod /usr/ov511.ko
```

```
/bin/mknod /dev/video0 c 81 0
```

```
/bin/mount -f -t cramfs -o remount,ro /dev/bon/2 /
```

```
/bin/mount -t ramfs ramfs /var
```

```
/bin/mkdir -p /var/tmp
```

```
/bin/mkdir -p /var/run
```

```
/bin/mkdir -p /var/log
```

```
/bin/mkdir -p /var/lock
```

```
/bin/mkdir -p /var/empty
```

```
#/bin/mount -t usbdevfs none /proc/bus/usb
```

```
exec /sbin/init
```

接下来启动 init 进程, init 进程是系统的第一个进程, 它的 PID 为 1, 是所有进程的父进程。Init 进程将用到系统引导配置文件/etc/inittab 中的信息, 根据该信息完成操作系统初始化工作。

```
# This is run first except when booting
```

```
::sysinit:/etc/init.d/rcS // ①
```

```
# Start an "askfirst" shell on the console
```

```
::askfirst:/bin/bash
```

```
::askfirst:/bin/bash
```

```
# Stuff to do when restarting the init process
```

```
::restart:/sbin/init
```

```
::once:/sbin/raja.sh
```

```
::respawn:/sbin/iom
```

```
::once:/usr/etc/rc.local // ②
```


Stuff to do before rebooting

::ctrlaltdel:/sbin/reboot

::shutdown:/bin/umount -a -r

① **/etc/init.d/rcS** 内容:

#!/bin/bash

/bin/mount -a

/usr/etc/profile

/***** profile 内容 *****/

#!/bin/bash

DISPLAY=unix:0.0

PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/**local**/bin

LD_LIBRARY_PATH=/lib:/usr/lib:/Qtopia/qt-2.3.2/lib

#QWS_KEYBOARD=TTY

LANG=C

HOME=/tmp //路径设置

KMOD=/lib/modules/2.4.19-rmk4-pxa2

export PATH LD_LIBRARY_PATH QTDIR LINUETTEDIR QWS_KEYBOARD LANG **HOME** DISPLAY
KDEDIR KMOD

*****/

#echo "make /var/tmp in /dev/shm"

```
#mkdir /dev/shm/tmp

#cp -a /etc/var /dev/shm

grep -q 'dnmode=multi' /proc/cmdline 2>/dev/null      // grep 为查找函数, 不太明白

if [ "$?" = "0" ]; then
    /bin/canna
    exit 0
fi

grep -q 'dnmode=single' /proc/cmdline 2>/dev/null

if [ "$?" = "0" ]; then
    /bin/canna -s
    exit 0
fi

grep -q 'resetmode=hard' /proc/cmdline 2>/dev/null
```

② **/usr/etc/rc.local** 内容, 用户可以把自已的初始化脚本程序放在这里:

```
#!/bin/bash

. /usr/etc/profile

/sbin/ifconfig lo 127.0.0.1 up

/sbin/ifconfig eth0 192.168.2.223 netmask 255.255.255.0 up

/bin/route add default gw 192.168.2.1 eth0

/sbin/inetd

/usr/sbin/makelinks
```

```
source /.bashrc
```

```
/bin/cp -rf /Qtopia/qtopia-free-1.7.0/wjluv/* /tmp/

# /bin/cp -rf /usr/config/* /tmp/

/bin/boa -c /home/httpd

/bin/mkdir /tmp/udisk

/bin/mkdir /tmp/images

/bin/mkdir /tmp/flashdisk

/bin/mkdir /tmp/sdcard

/bin/mkdir /tmp/mplayer

# /bin/mount -t yaffs /dev/mtdblock/3 /tmp/flashdisk/      //可挂载 FLASH

# /bin/mount -t yaffs /dev/nfblock/4 /tmp/mplayer/

if test -e "/tmp/flashdisk/userconfig"; then

/bin/chmod u+x /tmp/flashdisk/userconfig

. /tmp/flashdisk/userconfig

fi

. /testshell

# /Qtopia/demo/bin/transerver 1800 -stopplay
```

./testshell 内容, qpe 启动。

```
#!/bin/sh

if test -e "/dev/input/mouse0"; then

export QWS_MOUSE_PROTO=MouseMan:/dev/input/mice

exec /Qtopia/qtopia-free-1.7.0/bin/qpe-mouse -qws

else
```

```
exec /Qtopia/qtopia-free-1.7.0/bin/qpe -qws
```

```
fi
```

识别复杂变量的声明

```
/* 1. */ void * (*(fp1)(int)) [10];

/* 2. */ float (*(fp2)(int, int, float)) (int);

/* 3. */ typedef double (*(fp3)()) [10] ();

fp3 a;

/* 4. */ int (*(f4())[10]) ();
```

Number 1 says “fp1 is a pointer to a function that takes an integer argumnet and returns a pointer to an array of 10 void pointers.”

Number 2 says “fp1 is a pointer to a function that takes three arguments(int, int, and float) and returns a pointer to a function that takes an integer argument and returns a flaot.”

Number 3 says “an fp3 is a pointer to a function that takes no arguments and returns a pointer to an array of 10 pointers to functions thant take no arguments and return doubles.”

Number 4 says “f4 is a function that returns a pointer to an array of 10 pointers to functions that returns integers.”

第4个声明的是一个函数, 这个函数没有参数, 返回一个指针 p1

```
int ((*f4())[10]) ();  
  
int ((*p1)[10]) ();  
  
int (* p2 [10]) ();
```

p1 指向指针数组, 大小为10, 每个元素指向一个没有参数, 返回值为 int 的函数.

http://www.cppreference.com/wiki/data_types

Reading Type Declarations

Simple type declarations are easy to understand:

```
int i
```

However, it can be tricky to parse more complicated type declarations:

```
double **d[8]           // hmm..  
  
char *(*(**foo [][8])())[8] // augh! what is foo?
```

To understand the above declarations, follow three rules:

Start at the variable name (d or foo in the examples above)

End with the data type (double or char above)

Go right when you can, and left when you must. (Grouping parentheses can cause you to bounce left.)

For example:

Expression	Meaning
double **d[8];	

<code>double **d[8];</code>	d is ... double
<code>double **d[8];</code>	d is an array of 8 ... double
<code>double **d[8];</code>	d is an array of 8 pointer to ... double
<code>double **d[8];</code>	d is an array of 8 pointer to pointer to double

Another example:

Expression	Meaning
<code>char *(*(**foo [][8])())[]</code>	
<code>char *(*(**foo [][8])())[]</code>	foo is ... char
<code>char *(*(**foo [][8])())[]</code>	foo is an array of ... char
<code>char *(*(**foo [][8])())[]</code>	foo is an array of an array of 8 ... char
<code>char *(*(**foo [][8])())[]</code>	foo is an array of an array of 8 pointer to ... char
<code>char *(*(**foo [][8])())[]</code>	foo is an array of an array of 8 pointer to pointer to ... char
<code>char *(*(**foo [][8])())[]</code>	foo is an array of an array of 8 pointer to pointer to function returning ... char
<code>char *(*(**foo [][8])())[]</code>	foo is an array of an array of 8 pointer to pointer to function returning pointer to ... char
<code>char *(*(**foo [][8])())[]</code>	foo is an array of an array of 8 pointer to pointer to function returning pointer to array of ... char
<code>char *(*(**foo [][8])())[]</code>	foo is an array of an array of 8 pointer to pointer to function returning pointer to array of pointer to char

交叉编译环境搭建

2005 年 9 月 01 日

在进行嵌入式开发之前, 首先要建立一个交叉编译环境, 这是一套编译器、连接器和 libc 库等组成的开发环境。文章通过一个具体的例子说明了这些嵌入式交叉编译开发工具的制作过程。

随着消费类电子产品的大量开发和应用和 Linux 操作系统的不断健壮和强大, 嵌入式系统越来越多的进入人们的生活之中, 应用范围越来越广。

在裁减和定制 Linux, 运用于你的嵌入式系统之前, 由于一般嵌入式开发系统存储大小有限, 通常你都要在你的强大的 pc 机上建立一个用于目标机的交叉编译环境。这是一个由编译器、连接器和解释器组成的综合开发环境。交叉编译工具主要由 binutils、gcc 和 glibc 几个部分组成。有时出于减小 libc 库大小的考虑, 你也可以用别的 c 库来代替 glibc, 例如 uClibc、dietlibc 和 newlib。建立一个交叉编译工具链是一个相当复杂的过程, 如果你不想自己经历复杂的编译过程, 网上有一些编译好的可用的交叉编译工具链可以下载。

下面我们将以建立针对 arm 的交叉编译开发环境为例来解说整个过程, 其他的体系结构与这个相类似, 只要作一些对应的改动。我的开发环境是, 宿主机 i386-redhat-7.2, 目标机 arm。

这个过程如下

1. 下载源文件、补丁和建立编译的目录
2. 建立内核头文件
3. 建立二进制工具 (binutils)
4. 建立初始编译器 (bootstrap gcc)
5. 建立 c 库(glibc)
6. 建立全套编译器 (full gcc)

下载源文件、补丁和建立编译的目录

1. 选定软件版本号

选择软件版本号时, 先看看 glibc 源代码中的 INSTALL 文件。那里列举了该版本的 glibc 编译时所需的 binutils 和 gcc 的版本号。例如在 glibc-2.2.3/INSTALL 文件中推荐 gcc 用 2.95以上, binutils 用 2.10.1 以上版本。

我选的各个软件的版本是:


```
linux-2.4.21+rmk2
binutils-2.10.1
gcc-2.95.3
glibc-2.2.3
glibc-linuxthreads-2.2.3
```

如果你选的 glibc 的版本号低于 2.2, 你还要下载一个叫 glibc-crypt 的文件, 例如 glibc-crypt-2.1.tar.gz。Linux 内核你可以从 www.kernel.org 或它的镜像下载。

Binutils、gcc 和 glibc 你可以从 FSF 的 FTP 站点 <ftp://ftp.gnu.org/gnu/> 或它的镜像去下载。在编译 glibc 时, 要用到 Linux 内核中的 include 目录的内核头文件。如果你发现有变量没有定义而导致编译失败, 你就改变你的内核版本号。例如我开始用 linux-2.4.25+vrs2, 编译 glibc-2.2.3 时报 BUS_ISA 没定义, 后来发现在 2.4.23 开始它的名字被改为 CTL_BUS_ISA。如果你没有完全的把握保证你改的内核改完全了, 就不要动内核, 而是把你的 Linux 内核的版本号降低或升高, 来适应 glibc。

Gcc 的版本号, 推荐用 gcc-2.95 以上的。太老的版本编译可能会出问题。Gcc-2.95.3 是一个比较稳定的版本, 也是内核开发人员推荐用的一个 gcc 版本。

如果你发现无法编译过去, 有可能是你选用的软件中有的加入了一些新的特性而其他所选软件不支持的原因, 就相应降低该软件的版本号。例如我开始用 gcc-3.3.2, 发现编译不过, 报 as、ld 等版本太老, 我就把 gcc 降为 2.95.3。太新的版本大多没经过大量的测试, 建议不要选用。

2. 建立工作目录

首先, 我们建立几个用来工作的目录:

在你的用户目录, 我用的是用户 liang, 因此用户目录为 /home/liang, 先建立一个项目目录 embedded。

```
$pwd

/home/liang

$mkdir embedded
```

再在这个项目目录 embedded 下建立三个目录 build-tools、kernel 和 tools。

build-tools-用来存放你下载的 binutils、gcc 和 glibc 的源代码和用来编译这些源代码的目录。

kernel-用来存放你的内核源代码和内核补丁。

tools-用来存放编译好的交叉编译工具和库文件。

```
$cd embedded  
  
$mkdir build-tools kernel tools
```

执行完后目录结构如下:

```
$ls embedded  
  
build-tools kernel tools
```

3. 输出和环境变量

我们输出如下的环境变量方便我们编译。

```
$export PRJROOT=/home/liang/embedded  
  
$export TARGET=arm-linux  
  
$export PREFIX=$PRJROOT/tools  
  
$export TARGET_PREFIX=$PREFIX/$TARGET  
  
$export PATH=$PREFIX/bin:$PATH
```

如果你不惯用环境变量的, 你可以直接用绝对或相对路径。我如果不用环境变量, 一般都用绝对路径, 相对路径有时会失败。环境变量也可以定义在 .bashrc 文件中, 这样当你 logout 或换了控制台时, 就不用老是 export 这些变量了。

体系结构和你的 TARGET 变量的对应如下表

体系结构	TARGET 变量的值
PowerPC	Powerpc-linux
ARM	arm-linux
MIPS(big endian)	mips-linux
MIPS(little endian)	mipsel-linux
MIPS64	mips64-linux
SuperH3	sh3-linux
SuperH4	sh4-linux
I386	i386-linux
Ia64	ia64-linux
M68k	m68k-linux
M88k	m88k-linux
Alpha	alpha-linux
Sparc	sparc-linux
Sparc64	sparc64-linux

你可以在通过 glibc 下的 config.sub 脚本来知道, 你的 TARGET 变量是否被支持, 例如:

```
./config.sub arm-linux  
  
arm-unknown-linux-gnu
```

在我的环境中, config.sub 在 glibc-2.2.3/scripts 目录下。

网上还有一些 HOWTO 可以参考, ARM 体系结构的《The GNU Toolchain for ARM Target HOWTO》, PowerPC 体系结构的《Linux for PowerPC Embedded Systems HOWTO》等。对 TARGET 的选取可能有帮助。

4. 建立编译目录

为了把源码和编译时生成的文件分开, 一般的编译工作不在的源码目录中, 要另建一个目录来专门用于编译。用以下的命令来建立编译你下载的 binutils、gcc 和 glibc 的源代码的目录。

```
$cd $PRJROOT/build-tools  
  
$mkdir build-binutils build-boot-gcc build-gcc build-glibc gcc-patch
```

build-binutils-编译 binutils 的目录
build-boot-gcc-编译 gcc 启动部分的目录
build-glibc-编译 glibc 的目录
build-gcc-编译 gcc 全部的目录
gcc-patch-放 gcc 的补丁的目录

gcc-2.95.3 的补丁有 gcc-2.95.3-2.patch、gcc-2.95.3-no-fixinc.patch 和 gcc-2.95.3-returntype-fix.patch, 可以从 <http://www.linuxfromscratch.org/> 下载到这些补丁。

再将你下载的 binutils-2.10.1、gcc-2.95.3、glibc-2.2.3 和 glibc-linuxthreads-2.2.3 的源代码放入 build-tools 目录中

看一下你的 build-tools 目录, 有以下内容:

```
$ls  
  
binutils-2.10.1.tar.bz2      build-gcc      gcc-patch  
build-binutils              build-glibc     glibc-2.2.3.tar.gz  
build-boot-gcc              gcc-2.95.3.tar.gz  glibc-linuxthreads-2.2.3.tar.gz
```

建立内核头文件

把你从 www.kernel.org 下载的内核源代码放入 \$PRJROOT /kernel 目录

进入你的 kernel 目录:

```
$cd $PRJROOT /kernel
```

解开内核源代码

```
$tar -xzvf linux-2.4.21.tar.gz
```

或

```
$tar -xjvf linux-2.4.21.tar.bz2
```

小于 2.4.19 的内核版本解开会生成一个 linux 目录, 没带版本号, 就将其改名。

```
$mv linux linux-2.4.x
```

给 Linux 内核打上你的补丁

```
$cd linux-2.4.21
```

```
$patch -p1 < ../patch-2.4.21-rmk2
```

编译内核生成头文件

```
$make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

你也可以用 `config` 和 `xconfig` 来代替 `menuconfig`, 但这样用可能会没有设置某些配置文件选项和没有生成下面编译所需的头文件。推荐大家用 `make menuconfig`, 这也是内核开发人员用的最多的配置方法。配置完退出并保存, 检查一下的内核目录中的 `include/linux/version.h` 和 `include/linux/autoconf.h` 文件是不是生成了, 这是编译 `glibc` 是要用到的, `version.h` 和 `autoconf.h` 文件的存在, 也说明了你生成了正确的头文件。

还要建立几个正确的链接

```
$cd include

$ln -s asm-arm asm

$cd asm

$ln -s arch-epxa arch

$ln -s proc-armv proc
```

接下来为你的交叉编译环境建立你的内核头文件的链接

```
$mkdir -p $TARGET_PREFIX/include

$ln -s $PRJROOT/kernel/linux-2.4.21/include/linux
$TARGET_PREFIX/include/linux

$ln -s $PRJROOT/kernel/linux-2.4.21/include/asm-arm
$TARGET_PREFIX/include/asm
```

也可以把 Linux 内核头文件拷贝过来用

```
$mkdir -p $TARGET_PREFIX/include  
  
$cp -r $PRJROOT/kernel/linux-2.4.21/include/linux $TARGET_PREFIX/include  
  
$cp -r $PRJROOT/kernel/linux-2.4.21/include/asm-arm $TARGET_PREFIX/include
```

建立二进制工具 (binutils)

binutils 是一些二进制工具的集合, 其中包含了我们常用到的 as 和 ld。

首先, 我们解压我们下载的 binutils 源文件。

```
$cd $PRJROOT/build-tools  
  
$tar -xvjf binutils-2.10.1.tar.bz2
```

然后进入 build-binutils 目录配置和编译 binutils。

```
$cd build-binutils  
  
$../binutils-2.10.1/configure --target=$TARGET --prefix=$PREFIX
```

--target 选项是指出我们生成的是 arm-linux 的工具, --prefix 是指出我们可执行文件安装的位置。

会出现很多 check, 最后产生 Makefile 文件。

有了 Makefile 后, 我们来编译并安装 binutils, 命令很简单。

```
$make
```

```
$make install
```

看一下我们 \$PREFIX/bin 下生成的文件

```
$ls $PREFIX/bin

arm-linux-addr2line arm-linux-gasp arm-linux-objdump arm-linux-strings
arm-linux-ar arm-linux-ld arm-linux-ranlib arm-linux-strip
arm-linux-as arm-linux-nm arm-linux-readelf
arm-linux-c++filt arm-linux-objcopy arm-linux-size
```

我们来解释一下上面生成的可执行文件都是用来干什么的

addr2line - 将你要找的地址转成文件和行号, 它要使用 debug 信息。

Ar-产生、修改和解开一个存档文件

As-gnu 的汇编器

C++filt-C++ 和 java 中有一种重载函数, 所用的重载函数最后会被编译转化成汇编的标号, c++filt 就是实现这种反向的转化, 根据标号得到函数名。

Gasp-gnu 汇编器预编译器。

Ld-gnu 的连接器

Nm-列出目标文件的符号和对应的地址

Objcopy-将某种格式的目标文件转化成另外格式的目标文件

Objdump-显示目标文件的信息

Ranlib-为一个存档文件产生一个索引, 并将这个索引存入存档文件中

Readelf-显示 elf 格式的目标文件的信息

Size-显示目标文件各个节的大小和目标文件的大小

Strings-打印出目标文件中可以打印的字符串, 有个默认的长度, 为4

Strip-剥掉目标文件的所有的符号信息

建立初始编译器 (bootstrap gcc)

首先进入 build-tools 目录, 将下载 gcc 源代码解压

```
$cd $PRJROOT/build-tools  
$tar -xvzf gcc-2.95.3.tar.gz
```

然后进入 gcc-2.95.3 目录给 gcc 打上补丁

```
$cd gcc-2.95.3  
$patch -p1< ../gcc-patch/gcc-2.95.3-2.patch  
$patch -p1< ../gcc-patch/gcc-2.95.3-no-fixinc.patch  
$patch -p1< ../gcc-patch/gcc-2.95.3-returntype-fix.patch  
echo timestamp > gcc/cstamp-h.in
```

在我们编译并安装 gcc 前, 我们先要改一个文件 \$PRJROOT/gcc/config/arm/t-linux, 把
TARGET_LIBGCC2-CFLAGS = -fomit-frame-pointer -fPIC
这一行改为
TARGET_LIBGCC2-CFLAGS = -fomit-frame-pointer -fPIC -D__gthr_posix_h

你如果没定义 `-Dinhbit`, 编译时将会报如下的错误

```
../../gcc-2.95.3/gcc/libgcc2.c:41: stdlib.h: No such file or directory
../../gcc-2.95.3/gcc/libgcc2.c:42: unistd.h: No such file or directory
make[3]: *** [libgcc2.a] Error 1
make[2]: *** [stamp-multilib-sub] Error 2
make[1]: *** [stamp-multilib] Error 1
make: *** [all-gcc] Error 2
```

如果没有定义 `-D__gthr_posix_h`, 编译时会报如下的错误

```
In file included from gthr-default.h:1,
                  from ../../gcc-2.95.3/gcc/gthr.h:98,
                  from ../../gcc-2.95.3/gcc/libgcc2.c:3034:
../../gcc-2.95.3/gcc/gthr-posix.h:37: pthread.h: No such file or directory
make[3]: *** [libgcc2.a] Error 1
make[2]: *** [stamp-multilib-sub] Error 2
make[1]: *** [stamp-multilib] Error 1
make: *** [all-gcc] Error 2
```

还有一种与 `-Dinhbit` 同等效果的方法, 那就是在你配置 `configure` 时多加一个参数 `-with-newlib`, 这个选项不会迫使我们必须使用 `newlib`。我们编译了 `bootstrap-gcc` 后, 仍然可以选择任何 `c` 库。

接着就是配置 `bootstrap gcc`, 后面要用 `bootstrap gcc` 来编译 `glibc` 库。

```
$cd ../; cd build-boot-gcc  
  
$../gcc-2.95.3/configure --target=$TARGET --prefix=$PREFIX \  
  
>--without-headers --enable-languages=c --disable-threads
```

这条命令中的 `-target`、`--prefix` 和配置 `binutils` 的含义是相同的, `--without-headers` 就是指不需要头文件, 因为是交叉编译工具, 不需要本机上的头文件。`--enable-languages=c` 是指我们的 `boot-gcc` 只支持 `c` 语言。`--disable-threads` 是去掉 `thread` 功能, 这个功能需要 `glibc` 的支持。

接着我们编译并安装 `boot-gcc`

```
$make all-gcc  
  
$make install-gcc
```

我们来看看 `$PREFIX/bin` 里面多了哪些东西

```
$ls $PREFIX/bin
```

你会发现多了 `arm-linux-gcc`、`arm-linux-unprotoize`、`cpp` 和 `gcov` 几个文件。

`Gcc-gnu` 的 `C` 语言编译器

`Unprotoize`-将 `ANSI C` 的源码转化为 `K&R C` 的形式, 去掉函数原型中的参数类型。

`Cpp-gnu` 的 `C` 的预编译器

`Gcov-gcc` 的辅助测试工具, 可以用它来分析和优程序。

使用 gcc3.2 以及 gcc3.2 以上版本时, 配置 boot-gcc 不能使用 --without-headers 选项, 而需要使用 glibc 的头文件。

建立 c 库(glibc)

首先解压 glibc-2.2.3.tar.gz 和 glibc-linuxthreads-2.2.3.tar.gz 源代码

```
$cd $PRJROOT/build-tools  
  
$tar -xvzf glibc-2.2.3.tar.gz  
  
$tar -xvzf glibc-linuxthreads-2.2.3.tar.gz --directory=glibc-2.2.3
```

然后进入 build-glibc 目录配置 glibc

```
$cd build-glibc  
  
$CC=arm-linux-gcc ../glibc-2.2.3/configure --host=$TARGET --prefix="/usr"  
  
--enable-add-ons --with-headers=$TARGET_PREFIX/include
```

CC=arm-linux-gcc 是把 CC 变量设成你刚编译完的 bootstrap gcc, 用它来编译你的 glibc。--enable-add-ons 是告诉 glibc 用 linuxthreads 包, 在上面我们已经将它放入了 glibc 源码目录中, 这个选项等价于 -enable-add-ons=linuxthreads。--with-headers 告诉 glibc 我们的 linux 内核头文件的目录位置。

配置完后就可以编译和安装 glibc

```
$make  
  
$make install_root=$TARGET_PREFIX prefix="" install
```

然后你还要修改 libc.so 文件

将

```
GROUP ( /lib/libc.so.6 /lib/libc_nonshared.a)
```

改为

```
GROUP ( libc.so.6 libc_nonshared.a)
```

这样连接程序 ld 就会在 libc.so 所在的目录查找它需要的库, 因为你的机子的/lib 目录可能已经装了一个相同名字的库, 一个为编译可以在你的宿主机上运行的程序的库, 而不是用于交叉编译的。

建立全套编译器 (full gcc)

在建立 boot-gcc 的时候, 我们只支持了 C。到这里, 我们就要建立全套编译器, 来支持 C 和 C++。

```
$cd $PRJROOT/build-tools/build-gcc

$./gcc-2.95.3/configure --target=$TARGET --prefix=$PREFIX --enable-
languages=c,c++
```

--enable-languages=c,c++ 告诉 full gcc 支持 c 和 c++ 语言。

然后编译和安装你的 full gcc

```
$make all

$make install
```

我们再来看看 \$PREFIX/bin 里面多了哪些东西

```
$ls $PREFIX/bin
```

你会发现多了 arm-linux-g++ 、 arm-linux-protoize 和 arm-linux-c++ 几个文件。

G++-gnu 的 c++ 编译器。

Protoize-与 Unprotoize 相反, 将 K&R C 的源码转化为 ANSI C 的形式, 函数原型中加入参数类型。

C++-gnu 的 c++ 编译器。

到这里你的交叉编译工具就算做完了, 简单验证一下你的交叉编译工具。

用它来编译一个很简单的程序 helloworld.c

```
#include <stdio.h>

int main(void)
{
    printf("hello world\n");

    return 0;
}

$arm-linux-gcc helloworld.c -o helloworld

$file helloworld

helloworld: ELF 32-bit LSB executable, ARM, version 1,
dynamically linked (uses shared libs), not stripped
```

上面的输出说明你编译了一个能在 arm 体系结构下运行的 helloworld, 证明你的编译工具做成功了。

参考资料

● Wookey ,Chris Rutter, Jeff Sutherland, Paul Webb , 《The GNU Toolchain for ARM Target HOWTO》

- Karim Yaghmour, 《Building Embedded Linux Systems》, USA: O'Reilly, 2003

开发板上移植可执行程序

宿主机上编译生成的可执行文件(名字为 black)通过超级终端发送到目标板后, 可用终端执行。但若直接在 Qtopia 桌面环境中右键运行, 则会在终端中显示如下信息:

```
[root@(none) workspace]# Doing slow search for image: black
QImage::smoothScale: Image is a null image
QPixmap::convertFromImage: Cannot convert a null image
QDir::readDirEntries: Cannot read the directory: /Qtopia/qtopia-free-1.7.0/services/Open/application/octet-stream
could not open for writing `.new'
```

必须重新交叉编译 QPE 才能运行。

具体步骤如下:

1、解包 qtopia-free-1.7.0.tar.gz

```
tar zxvf qtopia-free-1.7.0.tar.gz
cd qtopia-free-1.7.0
export QTDIR=$QTDIR
export QPEDIR=$PWD
export PATH=$QPEDIR/bin:$PATH
```

注意: QTDIR 应设成 **qt-2.3.7-target**, 否则会出现 cannot find -lqte 和 cannot find -lqpe 的错误。

制作 32*32 的图标文件 black.png 存放在 \$QPEDIR/pics/inline 目录下, 将 \$QPEDIR/pics/inline 目录下的所有图形文件转换成为一个 c 语言的头文件, 这个头文件包含了该目录下的图形文件的 rgb 信息。

```
qembed --images $QPEDIR/pics/inline/*. * >
$QPEDIR/src/libraries/qtopia/inlinepics_p.h
```

2、在\$QPEDIR 目录下交叉编译:

```
cd src
./configure -platform linux-arm-g++
make
```

运行./configure -platform linux-arm-g++后会生成 Makefile 文件, 若已生成过, 不需要重复运行该指令。

编译过程中会出现如下错误:

```
make -C 3rdparty/libraries/libavcodec
make[1]: Entering directory `/usr/local/trolltech/qtopia-free-
1.7.0/src/3rdparty/libraries/libavcodec'
Makefile:10: *** missing separator. Stop.
make[1]: Leaving directory `/usr/local/trolltech/qtopia-free-
1.7.0/src/3rdparty/libraries/libavcodec'
make: *** [3rdparty/libraries/libavcodec] Error 2
```

解决方法: 因为 make 版本的问题, 修改 `/qtopia-1.7-tp/qtopia-free-1.7.0/src/3rdparty/libraries/libavcodec/Makefile 文件, 将前面3个-e 删了就 OK 了。

该编译过程所需时间较长, 约10分钟, 若不需要添加桌面图标生成新的桌面环境, 可不毕进行该步骤。以后测试时可直接使用已编译好的三个测试图标 (black、white、char)。

3、建立应用启动器 (.desktop) 文件

方法是: 建立一个文本文件, 在文件中添加以下的内容, 这些内容指明了应用的名称, 图标名等信息, 然后将文件更名为 xxxx.desktop, 保存在\$QPEDIR/apps/applications 目录下。以下是例子程序的启动器文件 (black.desktop):

```
[Desktop Entry]
Comment=A LCD test Program
Exec=black
Icon=black
Type=Application
Name=black
```

注: 其中 Exec 是可执行文件名称, Icon 是 png 图标名称, Name 是桌面上图标下显示名称。

4、建立根文件系统

把包括启动器文件, 包含了图标的库文件 libqte.so.*, 和应用程序的可执行文件添加到根文件系统中, 若根文件系统在文件夹 temp 下;

```
cp black.desktop temp/Qtopia/qtopia-free-1.7.0/apps/Applications/  
cp $QPEDIR/lib/libqpe.so.* temp/Qtopia/qtopia-free-1.7.0/lib/  
cp black temp/Qtopia/qtopia-free-1.7.0/bin
```

mkcramfs temp xxxxxx.cramfs (生成新的根文件系统)

将生成的新的根文件系统烧写到目标板的 FLASH 根文件系统区, 复位, 就可以看到 QPE 里有我们编写的应用的图标了, 点击这个图标, 程序就成功运行了。

目前存在问题: 不知如何结束该应用程序。程序运行时按动触摸屏仍会启动其他程序。

内核编译安装过程

《安装过程》

下载内核, 解压到/usr/src 目录下。备份原来/usr/include 目录下的 asm、Linux、scsi 文件夹。

在/usr/include 目录下建立 asm、Linux、scsi 的软链接分别到新内核目录 /usr/src/linux/include/下对应的文件夹。转入内核所在目录, 通过下列步骤开始编译新内核:

```
【make mrproper】  
【make menuconfig】  
【make dep】  
【make zImage】  
【make modules】  
【make modules_install depmod -a】  
【make install】
```

```
1. download linux-2.6.5.tar.gz
2. cp linux-2.6.5.tar.gz /usr/src
3. cd /usr/src
4. mv linux linux_old
5. tar xvzf linux-2.6.5.tar.gz
bzip2 -d linux-2.6.5.tar.bz2
tar -xvf linux-2.6.5.tar
或者 tar jxvf linux-2.6.5.tar.bz2
6. cd /usr/include
rm -rf asm linux scsi
ln -s /usr/src/linux/include/asm-i386 asm
ln -s /usr/src/linux/include/linux linux
ln -s /usr/src/linux/include/scsi scsi
7. cd /usr/src/linux
make mrproper (检查有无不正确的.o 文件和依赖关系, 使用刚下载的完整的源程序包进行编译, 所以本步可以省略。而如果你多次使用了这些源程序编译内核, 那么最好要先运行一下这个命令。)
8. make config 或 make menuconfig (推荐) 或 make xconfig
选择完返回 MainMenu, 选择 save and exit, 退出配置窗口并关闭超级用户终端。现在可以退出 xwindows 了, 回到文本环境 (这样做可以加快内核编译速度) 执行最后操作
9. make dep
10. make zImage
若想做成启动盘的形式可 make zdisk, 若 kernel 大于 512K, 应用 make bzImage (推荐)
11. make modules
make modules_install depmod -a
12. cp /usr/src/linux/arch/i386/boot/zImage /boot (用的是 make zImage)
或
cp /usr/src/linux/arch/i386/boot/bzImage /boot (用的是 make bzImage)
13. 修改 lilo.conf 或 grub.conf (添加)
image=/boot/zImage (或 bzImage)
label=new
root=...
14. lilo & grub
lilo -v 使改动生效。建议保留旧内核的项目, 避免编译失败。通常 grub 配置文件在下面三个地方
(根据发行版的不同):
```

```
/etc/grub.conf  
/boot/grub/menu.lst  
/boot/grub/grub.conf  
15. shutdown -r now
```

如果你的 kernel 支持内存镜像, 就用 `mkinitrd` 命令制作一个镜像文件, 然后拷贝到 `/boot` 路径下。

要这么做, 也要记得在 `make menu` 的时候记得选择内核镜像 `ram image`, 并且不能安装为模块, 否则

`initrd` 就不会运作。生成映像文件, 否则有时系统启动时会找不到/根分区。

```
mkinitrd /boot/initrd-2.4.20-8.img 2.4.20-8
```

`initrd.img` 就是 `initrd-2.4.20-8.img` 文件的链接

```
#ln -s initrd-2.4.20-8.img initrd.img
```

有的时候 `mkinitrd` 失败, 只要加上: `--without-scsi-module` 就可以通过了。

把旧内核源文件下的 `.config` 文件拷贝到新内核源文件下,

```
如 cp /usr/src/linux-2.4.22/.config /usr/src/linux-2.6.5/
```

然后在新内核目录下运行 `make oldconfig`, 如果想改动什么, 可以 `make menuconfig`, 但是一定要先 `make oldconfig`。然后就编译内核, 很容易: `make bzImage make modules make modules_install`

```
make install    然后到/boot 看看是不是已经有新内核文件了, 再看  
/boot/grub/grub.conf 是否已
```

经加载了新内核。

```
shutdown -r now
```

```
cp /usr/src/linux/arch/i386/boot/bzImage /boot/vmlinuz-2.6.5
```

(新内核的 `bzImage` 的位置也有可能在 `/usr/src/linux/i386/` 下)

```
cp /usr/src/linux/System.map /boot/System.map-2.6.5
```

```
make clean
```

没有 `make clean` 之前, 如果增加了新的模块, 可以直接 `make menuconfig && make moduels && make`

`modules_install`, 不需要全部重新来过。如果你以后还要利用这次编译的成果, 也可以省略这一步, 如果你想直接删除源代码目录, 也可以省略这一步。

一些升级内核前的备份过程:

```
cd /boot/  
mv System.map System.map.Old    【备份 System.Map && vmlinuz】  
mv vmlinuz vmlinuz.Old
```

```
cd /usr/src/  
ln -s linux-2.6.5 linux
```

进入/usr/include 目录下, 将 asm、scsi、linux 三个目录改名

【备份/usr/include 下的 asm、scsi、linux】

```
[root@localhost /usr/include]# mv asm asm.OFF  
[root@localhost /usr/include]# mv scsi scsi.OFF  
[root@localhost /usr/include]# mv linux linux.OFF
```

建立指向新内核的位置到第三步中被改名的原目录

```
[root@localhost /usr/include]# ln -s /usr/src/linux/include/asm-i386 asm  
[root@localhost /usr/include]# ln -s /usr/src/linux/include/scsi scsi  
[root@localhost /usr/include]# ln -s /usr/src/linux/include/linux linux
```

检验# ls /boot

如果看到了 vmlinuz-2.6.0 和 System.map-2.6.0 , 那么恭喜你, 成功了!

如果你是用 make install 安装的, 还会看到个 config-2.6.0 文件

```
cd /boot/  
ln -s vmlinuz-2.6.5 vmlinuz  
ln -s System.map-2.6.5 System.Map
```

(上面的两步的作用是保护真正的内核镜像不受损坏)

vmlinuz 是压缩的内核二进制可执行文件

#du -sh vmlinuz-2.6.5 用来查看新的内核有多大

如果你的 Kernel 配置支持 Modules 的话, 解决那些问题是比较简单的

只要编译那些.o 文件就可以啦.

比如我的声卡经常 Irq 和 IO 不对, 我就到/usr/src/linux/drivers/sound 目录下

```
gcc -o configure configure.c
```

```
./configure
```

选好 IO, IRQ 等等,

```
make
cp sound.o /lib/modules/2.0.34/misc
rmmod sound
insmod sound init_trace=1
测试一下. 直到成功为止. 比在/usr/src/linux 下, make menuconfig ;
make dep;make clean;make zImage;.... 好多了.
```

怎么给内核打补丁?

```
# cd /usr/src/linux
# bzip2 -dc patch-xxx.bz2 | patch
PATCH 文件拷贝到/usr/src 下:
```

```
#patch -p0 < patch-2.2.16
#gzip -cd patch-2.4.x-pre2-ac1.gz|patch -p1 -s -N -E -d 源码目录
可以使用这种方式来安装任何补丁, 而不用管它的文件名了
用 lsmod 看当前装入的 module, insmod/rmmod 增删 module. 对于 fs, network 方面的
module, kernel 是可以
自动动态装载的(通过 kernel.d, 要用时自动 insmod) 而对于与硬件相关的 module(如网卡,
SCSI 卡等), 则需
自己手工加入, 或用 RedHat Control Panel 中的 kernelcfg 来选择 module(写于
/etc/conf.modules(?), 那
样则 boot kernel 时会自动加入
```

嵌入式经典面试题目

C 语言测试是招聘嵌入式系统程序员过程中必须而且有效的方法。这些年, 我既参加也组织了许多这种测试, 在这过程中我意识到这些测试能为面试者和被面试者提供许多有用信息, 此外, 撇开面试的压力不谈, 这种测试也是相当有趣的。

从被面试者的角度来讲, 你能了解许多关于出题者或监考者的情况。这个测试只是出题者为显示其对 ANSI 标准细节的知识而不是技术技巧而设计吗? 这是个愚蠢的问题吗? 如要你答出某个字符的 ASCII 值。这些问题着重考察你的系统调用和内存分配策略方面的能力吗? 这标志着出题者也许花时间在微机上而不是在嵌入

式系统上。如果上述任何问题的答案是”是”的话,那么我知道我得认真考虑我是否应该去做这份工作。从面试者的角度来讲,一个测试也许能从多方面揭示应试者的素质:最基本的,你能了解应试者 C 语言的水平。不管怎么样,看一下这人如何回答他不会的问题也是满有趣。应试者是以好的直觉做出明智的选择,还是只是瞎蒙呢?当应试者在某个问题上卡住时是找借口呢,还是表现出对问题的真正的好奇心,把这看成学习的机会呢?我发现这些信息与他们的测试成绩一样有用。

有了这些想法,我决定出一些真正针对嵌入式系统的考题,希望这些令人头痛的考题能给正在找工作的人一点帮助。这些问题都是我这些年实际碰到的。其中有些题很难,但它们应该都能给你一点启迪。这个测试适于不同水平的应试者,大多数初级水平的应试者的成绩会很差,经验丰富的程序员应该有很好的成绩。为了让你自己能自己决定某些问题的偏好,每个问题没有分配分数,如果选择这些考题为你所用,请自行按你的意思分配分数。

预处理器 (Preprocessor)

1. 用预处理指令#define 声明一个常数,用以表明1年中有多少秒(忽略闰年问题)

```
#define SECONDS_PER_YEAR (60 * 60 * 24 * 365)UL
```

我在这想看到几件事情:

- 1) #define 语法的基本知识(例如:不能以分号结束,括号的使用,等等)
- 2) 懂得预处理器将为你计算常数表达式的值,因此,直接写出你是如何计算一年中有多少秒而不是计算出实际的值,是更清晰而没有代价的。
- 3) 意识到这个表达式将使一个16位机的整型数溢出-因此要用到长整型符号 L,告诉编译器这个常数是长整型数。
- 4) 如果你在表达式中用到 UL (表示无符号长整型),那么你有了一个好的起点。记住,第一印象很重要。

2. 写一个”标准”宏 MIN,这个宏输入两个参数并返回较小的一个。

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

这个测试是为下面的目的而设的:

- 1) 标识#define 在宏中的应用的基本知识。这是很重要的。因为在 嵌入(inline)操作符 变为标准 C 的一部分之前,宏是方便产生嵌入代码的唯一方法,对于嵌入式系统来说,为了能达到要求的性能,嵌入代码经常是必须的方法。
- 2) 三重条件操作符的知识。这个操作符存在 C 语言中的原因是它使得编译器能产生比 if-then-else 更优化的代码,了解这个用法是很重要的。
- 3) 懂得在宏中小心地把参数用括号括起来
- 4) 我也用这个问题开始讨论宏的副作用,例如:当你写下面的代码时会发生什么事?
least = MIN(*p++, b);

3. 预处理器标识#error 的目的是什么?

如果你不知道答案,请看参考文献1。这问题对区分一个正常的伙计和一个书呆子是很有用的。只有书呆子才会读 C 语言课本的附录去找象这种问题的答案。当然如果你不是在找一个书呆子,那么应试者最好希望自己不要知道答案。

死循环 (Infinite loops)

4. 嵌入式系统中经常要用到无限循环, 你怎么样用 C 编写死循环呢?
这个问题用几个解决方案。我首选的方案是:

```
while(1)
{

}
```

一些程序员更喜欢如下方案:

```
for(;;)
{

}
```

这个实现方式让我为难, 因为这个语法没有确切表达到底怎么回事。如果一个应试者给出这个作为方案, 我将用这个作为一个机会去探究他们这样做的基本原理。如果他们的基本答案是: ”我被教着这样做, 但从没有想到过为什么。” 这会给我留下一个坏印象。

第三个方案是用 goto

Loop:

...

goto Loop;

应试者如给出上面的方案, 这说明或者他是一个汇编语言程序员 (这也许是好事) 或者他是一个想进入新领域的 BASIC/FORTRAN 程序员。

数据声明 (Data declarations)

5. 用变量 a 给出下面的定义

- a) 一个整型数 (An integer)
- b) 一个指向整型数的指针 (A pointer to an integer)
- c) 一个指向指针的指针, 它指向的指针是指向一个整型数 (A pointer to a pointer to an integer)
- d) 一个有10个整型数的数组 (An array of 10 integers)
- e) 一个有10个指针的数组, 该指针是指向一个整型数的。 (An array of 10 pointers to integers)
- f) 一个指向有10个整型数数组的指针 (A pointer to an array of 10 integers)
- g) 一个指向函数的指针, 该函数有一个整型参数并返回一个整型数 (A pointer to a function that takes an integer as an argument and returns an integer)
- h) 一个有10个指针的数组, 该指针指向一个函数, 该函数有一个整型参数并返回一个整型数 (An array of ten pointers to functions that take an integer argument and return an integer)

答案是:

- a) int a; // An integer

```
b) int *a; // A pointer to an integer
c) int **a; // A pointer to a pointer to an integer
d) int a[10]; // An array of 10 integers
e) int *a[10]; // An array of 10 pointers to integers
f) int (*a)[10]; // A pointer to an array of 10 integers
g) int (*a)(int); // A pointer to a function a that takes an integer argument and returns an integer
h) int (*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer
```

人们经常声称这里有几个问题是那种要翻一下书才能回答的问题, 我同意这种说法。当我写这篇文章时, 为了确定语法的正确性, 我的确查了一下书。但是当我被面试的时候, 我期望被问到这个问题(或者相近的问题)。因为在被面试的这段时间里, 我确定我知道这个问题的答案。应试者如果不知道所有的答案(或至少大部分答案), 那么也就没有为这次面试做准备, 如果该面试者没有为这次面试做准备, 那么他又能为什么出准备呢?

Static

6. 关键字 static 的作用是什么?

这个问题很少有人能回答完全。在 C 语言中, 关键字 static 有三个明显的作用:

- 1) 在函数体, 一个被声明为静态的变量在这一函数被调用过程中维持其值不变。
- 2) 在模块内(但在函数体外), 一个被声明为静态的变量可以被模块内所用函数访问, 但不能被模块外其它函数访问。它是一个本地的全局变量。
- 3) 在模块内, 一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是, 这个函数被限制在声明它的模块的本地范围内使用。

大多数应试者能正确回答第一部分, 一部分能正确回答第二部分, 同是很少的人能懂得第三部分。这是一个应试者的严重的缺点, 因为他显然不懂得本地化数据和代码范围的好处和重要性。

Const

7. 关键字 const 有什么含意?

我只要一听到被面试者说: "const 意味着常数", 我就知道我正在和一个业余者打交道。去年 Dan Saks 已经在他的文章里完全概括了 const 的所有用法, 因此 ESP(译者: Embedded Systems Programming)的每一位读者应该非常熟悉 const 能做什么和不能做什么. 如果你从没有读到那篇文章, 只要能说出 const 意味着"只读"就可以了。尽管这个答案不是完全的答案, 但我接受它作为一个正确的答案。(如果你想知道更详细的答案, 仔细读一下 Saks 的文章吧。)

如果应试者能正确回答这个问题, 我将问他一个附加的问题:

下面的声明都是什么意思?

```
const int a;
int const a;
```



```
const int *a;
int * const a;
int const * a const;
```

```
/******/
```

前两个的作用是一样, a 是一个常整型数。第三个意味着 a 是一个指向常整型数的指针 (也就是, 整型数是不可修改的, 但指针可以)。第四个意思 a 是一个指向整型数的常指针 (也就是说, 指针指向的整型数是可以修改的, 但指针是不可修改的)。最后一个意味着 a 是一个指向常整型数的常指针 (也就是说, 指针指向的整型数是不可修改的, 同时指针也是不可修改的)。如果应试者能正确回答这些问题, 那么他就给我留下了一个好印象。顺带提一句, 也许你可能会问, 即使不用关键字 const, 也还是能很容易写出功能正确的程序, 那么我为什么还要如此看重关键字 const 呢? 我也如下的几下理由:

- 1) 关键字 const 的作用是为给读你代码的人传达非常有用的信息, 实际上, 声明一个参数为常量是为了告诉了用户这个参数的应用目的。如果你曾花很多时间清理其它人留下的垃圾, 你就会很快学会感谢这点多余的信息。(当然, 懂得用 const 的程序员很少会留下的垃圾让别人来清理的。)
- 2) 通过给优化器一些附加的信息, 使用关键字 const 也许能产生更紧凑的代码。
- 3) 合理地使用关键字 const 可以使编译器很自然地保护那些不希望被改变的参数, 防止其被无意的代码修改。简而言之, 这样可以减少 bug 的出现。

Volatile

8. 关键字 volatile 有什么含意?并给出三个不同的例子。

一个定义为 volatile 的变量是说这变量可能会被意想不到地改变, 这样, 编译器就不会去假设这个变量的值了。精确地说就是, 优化器在用到这个变量时必须每次都小心地重新读取这个变量的值, 而不是使用保存在寄存器里的备份。下面是 volatile 变量的几个例子:

- 1) 并行设备的硬件寄存器 (如: 状态寄存器)
- 2) 一个中断服务子程序中会访问到的非自动变量 (Non-automatic variables)
- 3) 多线程应用中被几个任务共享的变量

回答不出这个问题的人是不会被雇佣的。我认为这是区分 C 程序员和嵌入式系统程序员的最基本的问题。搞嵌入式的家伙们经常同硬件、中断、RTOS 等等打交道, 所有这些都要用到 volatile 变量。不懂得 volatile 的内容将会带来灾难。

假设被面试者正确地回答了这是问题 (嗯, 怀疑是否会是这样), 我将稍微深究一下, 看一下这家伙是不是真正懂得 volatile 完全的重要性。

- 1) 一个参数既可以是 const 还可以是 volatile 吗? 解释为什么。
- 2); 一个指针可以是 volatile 吗? 解释为什么。
- 3); 下面的函数有什么错误:

```
int square(volatile int *ptr)
{
return *ptr * *ptr;
}
```

下面是答案:

- 1) 是的。一个例子是只读的状态寄存器。它是 `volatile` 因为它可能被意想不到地改变。它是 `const` 因为程序不应该试图去修改它。
- 2); 是的。尽管这并不很常见。一个例子是当一个中服务子程序修该一个指向一个 `buffer` 的指针时。
- 3) 这段代码有点变态。这段代码的目的是用来返指针 `*ptr` 指向值的平方, 但是, 由于 `*ptr` 指向一个 `volatile` 型参数, 编译器将产生类似下面的代码:

```
int square(volatile int *ptr)
{
    int a,b;
    a = *ptr;
    b = *ptr;
    return a * b;
}
```

由于 `*ptr` 的值可能被意想不到地该变, 因此 `a` 和 `b` 可能是不同的。结果, 这段代码可能返不是你所期望的平方值! 正确的代码如下:

```
long square(volatile int *ptr)
{
    int a;
    a = *ptr;
    return a * a;
}
```

位操作 (Bit manipulation)

9. 嵌入式系统总是要用户对变量或寄存器进行位操作。给定一个整型变量 `a`, 写两段代码, 第一个设置 `a` 的 bit 3, 第二个清除 `a` 的 bit 3。在以上两个操作中, 要保持其它位不变。

对这个问题有三种基本的反应

- 1) 不知道如何下手。该被面者从没做过任何嵌入式系统的工作。
- 2) 用 `bit fields`。`Bit fields` 是被扔到 C 语言死角的东西, 它保证你的代码在不同编译器之间是不可移植的, 同时也保证了你的代码是不可重用的。我最近不幸看到 Infineon 为其较复杂的通信芯片写的驱动程序, 它用到了 `bit fields` 因此完全对我无用, 因为我的编译器用其它的方式来实现 `bit fields` 的。从道德讲: 永远不要让一个非嵌入式的东西粘实际硬件的边。
- 3) 用 `#defines` 和 `bit masks` 操作。这是一个有极高可移植性的方法, 是应该被用到的方法。最佳的解决方案如下:

```
#define BIT3 (0x1 << 3)
static int a;
```

```
void set_bit3(void)
{
a |= BIT3;
}
void clear_bit3(void)
{
a &= ~BIT3;
}
```

一些人喜欢为设置和清除值而定义一个掩码同时定义一些说明常数, 这也是可以接受的。我希望看到几个要点: 说明常数、|=和&=~操作。

访问固定的内存位置 (Accessing fixed memory locations)

10. 嵌入式系统经常具有要求程序员去访问某特定的内存位置的特点。在某工程中, 要求设置一绝对地址为 $0 \times 67a9$ 的整型变量的值为 $0xaa66$ 。编译器是一个纯粹的 ANSI 编译器。写代码去完成这一任务。这一问题测试你是否知道为了访问一绝对地址把一个整型数强制转换 (typecast) 为一指针是合法的。这一问题的实现方式随着个人风格不同而不同。典型的类似代码如下:

```
int *ptr;
ptr = (int *) $0 \times 67a9$ ;
*ptr =  $0xaa55$ ;
```

A more obscure approach is:

一个较晦涩的方法是:

```
*(int * const)( $0 \times 67a9$ ) =  $0xaa55$ ;
```

即使你的品味更接近第二种方案, 但我建议你在面试时使用第一种方案。

中断 (Interrupts)

11. 中断是嵌入式系统中重要的组成部分, 这导致了很多编译开发商提供一种扩展—让标准 C 支持中断。具代表事实是, 产生了一个新的关键字 __interrupt。下面的代码就使用了 __interrupt 关键字去定义了一个中断服务子程序 (ISR), 请评论一下这段代码的。

```
__interrupt double compute_area (double radius)
{
double area = PI * radius * radius;
printf(“\nArea = %f”, area);
return area;
}
```

这个函数有太多的错误了, 以至让人不知从何说起了:

- 1) ISR 不能返回一个值。如果你不懂这个, 那么你不会被雇用的。
- 2) ISR 不能传递参数。如果你没有看到这一点, 你被雇用的机会等同第一项。
- 3) 在许多的处理器/编译器中, 浮点一般都是不可重入的。有些处理器/编译器需要让额处的寄存器入栈, 有些处理器/编译器就是不允许在 ISR 中做浮点运算。此外, ISR 应该是短而有效率的, 在 ISR 中做浮点运算是不明智的。
- 4) 与第三点一脉相承, printf() 经常有重入和性能上的问题。如果你丢掉了第三和第四点, 我不会太为难你的。不用说, 如果你能得到后两点, 那么你的被雇用前景越来越光明了。

代码例子 (Code examples)

12. 下面的代码输出是什么, 为什么?

```
void foo(void)
{
    unsigned int a = 6;
    int b = -20;
    (a+b > 6) ? puts( "> 6" ) : puts( "<= 6" );
}
```

这个问题测试你是否懂得 C 语言中的整数自动转换原则, 我发现有些开发者懂得极少这些东西。不管如何, 这无符号整型问题的答案是输出是 “>6”。原因是当表达式中存在有符号类型和无符号类型时所有的操作数都自动转换为无符号类型。因此-20变成了一个非常大的正整数, 所以该表达式计算出的结果大于6。这一点对于应当频繁用到无符号数据类型的嵌入式系统来说是非常重要的。如果你答错了这个问题, 你也就到了得不到这份工作的边缘。

13. 评价下面的代码片断:

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF;
/*1's complement of zero */
```

对于一个 int 型不是16位的处理器为说, 上面的代码是不正确的。应编写如下:

```
unsigned int compzero = ~0;
```

这一问题真正能揭露出应试者是否懂得处理器字长的重要性。在我的经验里, 好的嵌入式程序员非常准确地明白硬件的细节和它的局限, 然而 PC 机程序往往把硬件作为一个无法避免的烦恼。

到了这个阶段, 应试者或者完全垂头丧气了或者信心满满志在必得。如果显然应试者不是很好, 那么这个测试就在这里结束了。但如果显然应试者做得不错, 那么我就扔出下面的追加问题, 这些问题是比较难的, 我想仅仅非常优秀的应试者能做得不错。提出这些问题, 我希望更多看到应试者应付问题的方法, 而不是答案。不管如何, 你就当是这个娱乐吧...

动态内存分配 (Dynamic memory allocation)

14. 尽管不像非嵌入式计算机那么常见, 嵌入式系统还是有从堆 (heap) 中动态分配内存的过程的。那么嵌入式系统中, 动态分配内存可能发生的问题是什么?

这里, 我期望应试者能提到内存碎片, 碎片收集的问题, 变量的持行时间等等。这个主题已经在 ESP 杂志中被广泛地讨论过了 (主要是 P.J. Plauger, 他的解释远远超过我这里能提到的任何解释), 所有回过头看一下这些杂志吧! 让应试者进入一种虚假的安全感觉后, 我拿出这么一个小节目:

下面的代码片段的输出是什么, 为什么?

```
char *ptr;
if ((ptr = (char *)malloc(0)) == NULL)
puts( "Got a null pointer" );
else
puts( "Got a valid pointer" );
```

这是一个有趣的问题。最近在我的一个同事不经意把0值传给了函数 malloc, 得到了一个合法的指针之后, 我才想到这个问题。这就是上面的代码, 该代码的输出是 "Got a valid pointer"。我用这个来开始讨论这样的一问题, 看看被面试者是否想到库例程这样做是正确。得到正确的答案固然重要, 但解决问题的方法和你做决定的基本原理更重要些。

Typedef

15 Typedef 在 C 语言中频繁用以声明一个已经存在的数据类型的同义字。也可以用预处理器做类似的事。例如, 思考一下下面的例子:

```
#define dPS struct s *
typedef struct s * tPS;
```

以上两种情况的意图都是要定义 dPS 和 tPS 作为一个指向结构 s 指针。哪种方法更好呢? (如果有的话) 为什么?

这是一个非常微妙的问题, 任何人答对这个问题 (正当的原因) 是应当被恭喜的。答案是: typedef 更好。思考下面的例子:

```
dPS p1, p2;
tPS p3, p4;
```

第一个扩展为

```
struct s * p1, p2;
```

上面的代码定义 p1 为一个指向结构的指, p2 为一个实际的结构, 这也许不是你想要的。第二个例子正确地定义了 p3 和 p4 两个指针。

晦涩的语法

16 . C 语言同意一些令人震惊的结构, 下面的结构是合法的吗, 如果是它做些什么 ?

```
int a = 5, b = 7, c;  
c = a+++b;
```

这个问题将做为这个测验的一个愉快的结尾。不管你相不相信, 上面的例子是完全合乎语法的。问题是编译器如何处理它? 水平不高的编译作者实际上会争论这个问题, 根据最处理原则, 编译器应当能处理尽可能所有合法的用法。因此, 上面的代码被处理成:

```
c = a++ + b;
```

因此, 这段代码持行后 $a = 6$, $b = 7$, $c = 12$ 。

如果你知道答案, 或猜出正确答案, 做得好。如果你不知道答案, 我也不把这个当作问题。我发现这个问题的最大好处是这是一个关于代码编写风格, 代码的可读性, 代码的可修改性的好的话题。

好了, 伙计们, 你现在已经做完所有的测试了。这就是我出的 C 语言测试题, 我怀着愉快的心情写完它, 希望你以同样的心情读完它。如果是认为这是一个好的测试, 那么尽量都用到你的找工作的过程中去吧。天知道也许过个一两年, 我就不做现在的工作, 也需要找一个。

作者介绍:

Nigel Jones 是一个顾问, 现在住在 Maryland, 当他不在水下时, 你能在多个范围的嵌入项目中找到他。他很高兴能收到读者的来信, 他的 email 地址是: NAJones@compuserve.com

参考文献

- 1) Jones, Nigel, " In Praise of the #error directive, " Embedded Systems Programming, September 1999, p. 114.
- 2) Jones, Nigel, " Efficient C Code for Eight-bit MCUs , " Embedded Systems Programming, November 1998, p. 66.

清理 Linux 老内核

Ubuntu 用久了, 升级了几次内核以后, 就免不了需要清理一下老的内核, 毕竟一套内核就有 170 多 M 的空间呢。但是每次都打一堆字来清理也烦了。虽然 Ubuntu 命令技巧 (这里再罗嗦几句: 如果还有 Ubuntu 没看过这个, 强烈建议看一遍。) 里有删除旧内核的命令, 而且就一行:

```
sudo aptitude purge `linux-image-.*\(!`uname -r`\)
```

但是我却不是很喜欢, 因为这这个没有把 linux-headers-xxx 删除掉。
于是自己写了个繁的, 需要的可以自己保存, 以便不时之需。

```
#!/bin/sh

# 清理 Ubuntu 的老内核

# by bones7456

# http://li2z.cn

CURRENT=`uname -r | awk -F"." '{print $1"-"$2}'`

HEADERS=""

IMAGES=""

for HEADER in `dpkg --get-selections | grep ^linux-headers | \

grep -vE "(generic|386|virtual)" | awk '{gsub(/linux-headers-/,"",$1);print $1}'`

do

    if [[ "$CURRENT" < "$HEADER" ]]

    then

        echo "正在运行的内核不是最新的。 $CURRENT < $HEADER"

        echo "Running kernel is not the newest. $CURRENT < $HEADER"

    exit 1
```

```
else

    [[ "$CURRENT" != "$HEADER" ]] && {

        HEADERS="$${HEADERS} linux-headers-${HEADER}"

        IMAGE="`dpkg --get-selections | grep ^linux-image | \

            grep "${HEADER}" | awk '{print $1}'`"

        IMAGES="$${IMAGES} $IMAGE"

    }

fi

done

if [[ x"$HEADERS" == x"" ]]

then

    echo "没有要清理的老内核."

    echo "No old kernel need to clean."

    exit 0

fi

CMD="sudo apt-get purge $HEADERS $IMAGES"
```



```
echo "$CMD"

if [ "$1" == "-e" ]

then

    sh -c "$CMD"

else

    echo "请确定以上命令是否正确, 然后输入 $0 -e 来执行以上命令。"

    echo "Be sure this command is right, then type $0 -e to execute."

fi

from:li2z.cn
```

如何做嵌入式 Linux 操作系统

做一个嵌入式 Linux 系统究竟要做哪些工作

做一个嵌入式 Linux 系统究竟需要做哪些工作? 也就是本文究竟要讲述哪些内容? 我先介绍一个脉络, 可以做为我们后面工作的一个总的提纲:

第一步、建立交叉编译环境

没有交叉开发经验的读者, 可能一时很难接受这个概念。首先, 要明白两个概念: 一般我们工作的机器, 称为开发机、主机; 我们制作好的系统将要放到某台机器, 如手机或另一台 PC 机, 这台机我们称为目标主机。

我们一般开发机上已经有一套开发工具, 我们称之为原生开发套件, 我们一般就是用它们来写程序, 那么, 那什么又是交叉编译环境呢? 其实一点也不神秘, 也就是在开发机上再安装一套开发工具, 这套开发工具编译出来的程序, 如内核、系统工作或者我们自己的程序, 是放在目标主机上运行的。

那么或许有初学者会问, 直接用原生开发工具为目标主机编译程序不就完了? 至少我当初是这么想的。一般来说, 我们的开发机都是 X86 平台, 原生开发套件开发的工具, 也针对 X86 平台, 而我们的目标主机可能是 PowerPC、IXP、MIPS.....所以, 我们的交叉编译环境是针对某一类具体平台的。

一般来讲, 交叉开发环境需要二进制工具程序、编译器、C 链接库, 嵌入式开发常用的这三类软件是:

Binutils

Gcc

uClibc

当然, GNU 包含的工具套件不仅于此, 你还要以根据实际需要, 进行选择

第二步、编译内核

开发工具是针对某一类硬件平台, 内核同样也是。这一步, 我们需要用第一步中建立的工具, 对内核进行编译, 对于有内核编译经验的人来说, 这是非常简单的;

第三步、建立根文件系统

也就是建立我们平常看到的 bin、dev、proc.....这一大堆目录, 以及一些必备的文件; 另外, 我们还需要为我们的目标系统安装一些常用的工具软件, 如 ls、ifconfig.....当然, 一个办法是找到这些工具的源代码, 用第一步建立的交叉编译工具来编译, 但是这些软件一是数量多, 二是某些体积较大, 不适合嵌入式系统, 这一步, 我们一般都是用 busybox 来完成的, 包括系统引导软件 init;

最后, 我们为系统还需要建立初始化的引导文件, 如 inittab.....

第四步、启动系统

在这一步, 我们把建立好的目标、文件、程序、内核及模块全部拷贝到目标机存储器上, 如硬盘。然后为系统安装 bootloader, 对于嵌入式系统, 有许多引导程序可供我们使用。不过它们许多都有硬件平台的限制。当然, 如果你是工作在 X86, 可以直接用 lilo 来引导, 事实上, 本文就是采用的 lilo。

做到这一步, 将目标存储设备挂上目标机, 如果顺利, 就可以启动系统了。

当然, 针对某些特别的平台, 不能像硬盘这样拷贝了, 需要读卡器、烧录.....但是基本的方法是相通的!

第五步、优化和个性化系统

通过前四步, 我们已经得到了一个可以正常工作的系统。在这一步里, 就是发挥你想像的时候了.....

本文的工作环境

项目根目录/home/kendo/project ----->;我将它指定至 PATH:\$PRJROOT

子目录及说明

目录 内容

bootloader 目标板的引导加载程序, 如 lilo 等

build-tools 建立交叉编译平台的工具源码

debug 调试工具及所有相关包

doc 项目中用到的所有文档

images 编译好的内核映像, 以及根文件系统

kernel 各个版本的 Linux 内核源码

rootfs 制作好的根文件系统

sysapps 目标板将要用到的系统应用系统, 比如 tthttpd, udhcpd 等

tmp 存放临时文件

tools 编译好的跨平台开发工具链以及 C 链接库

工作的脚本

```
#!/usr/bin
```

```
export PROJECT=skynet
```

```
export PRJROOT=/home/${PROJECT}
```

```
export TARGET=i386-linux
```

```
export PREFIX=${PRJROOT}/tools
```

```
export TARGET_PREFIX=${PREFIX}/${TARGET}
```

```
export PATH=${PREFIX}/bin:/bin:/sbin:/usr/bin:/usr/sbin
```

```
cd $PRJROOT
```

第二章 建立交叉编译环境

在 CU 中发表的另一篇同名的贴子里, 我讲述了一个全手工创建交叉编译环境的方法。目前, 创建交叉编译环境, 包括建立根文件, 一般来讲, 有两种方法:

手功创建

可以得到最大程序的个性化定制, 缺点是过程繁杂, 特别是极易出错, 注意这个“极”字, 包括有经验的开发人员;

自动创建

无它, 方便而。

因为前一篇文章中, 已经讲述了全手工创建交叉编译环境的一般性方法, 本文就不打算再 重复这个步骤了, 感兴趣的朋友, 可以再去搜索那篇贴子, 提醒一点的就是, 在准备工具链的时候, 要注意各个工具版本之间的搭配、每个工具需要哪些补丁, 我建议你在 google 上针对这两项搜索一下, 准备一个清单, 否则.....

本章要讲述的是自动创建交叉编译环境的方法。目标, 针对商业硬件平台, 厂家都会为你提供 一个开发包, 我用过 XX 厂家的 IXP425 和 MIPS 的, 非常地方便, 记得我第一次接触嵌入式开发, 拿着这个开发包自动化创建交叉编译环境、编译内核、建立根文件系统、创建 Ram Disk, 我反复做了三四次, 结果还不知道自己究竟做了些什么, 呵呵, 够傻吧.....

所以, 建议没有这方面经验的读者, 还是首先尝试一下手工创建的方法吧, 而本章接下来的内容, 是送给曾经被它深深伤害而不想再次去亲历这项工作而又想提高交率而又在通用平台上工作没有商业开发包的朋友。

建立交叉开发工具链

准备工具:

buildroot-0.9.27.tar.tar

只需要一个软件? 对, 其它的不用准备了, buildroot 事实上是一个脚本与补丁的集合, 其它需要用到的软件, 如 gcc、uClibc, 你只需在 buildroot 中指明相应的版本, 它会自动去给你下载。

事实上, buildroot 到网上去下载所需的所有工作是需要时间的, 除非你的带宽 足够, 否则下载软件时间或许会占去 80%, 而我在做这项工作之间, 所需的工作链全部都在我本地硬盘上, 我解压开 buildroot 后, 新建 dl 文件夹, 将 所有工具源码的压缩包拷贝进去, 呵呵, buildroot 就不用去网上下载了。

我的软件清单:

Linux-libc-headers-2.4.27.tar.bz2

Gcc-3.3.4.tar.bz2

binutils 2.15.91.0.2.tar.bz2

uClibc 0.9.27.tar.bz2

genext2fs_1.3.orig.tar.gz

ccache-2.3.tar.gz

将它拷贝到\${PRJROOT}/build-tools下, 解压

```
[root@skynet build-tools]# tar jxvf buildroot-0.9.27.tar.tar
```

```
[root@skynet build-tools]# cd buildroot
```

配置它:

```
[root@skynet build-tools]#make menuconfig
```

Target Architecture (i386) --->; 选择硬件平台, 我的是 i386

Build options --->; 编译选项

这个选项下重要的是(`${PRJROOT}/tools`) Toolchain and header file location?编译好的工具链放在哪儿?

如果你像我一样, 所有工具包都在本地, 不需它到网上自动下载, 可以把 `wget command` 选项清空;

Toolchain Options --->; 工具链选项

--- Kernel Header Options 头文件它会自动去下载, 不过应该保证与你将要用的内核是同一个版本;

☐ Use the daily snapshot of uClibc? 使用最近的 uClibc 的 snapshot

Binutils Version (binutils 2.15.91.0.2) --->; Binutils 的版本

GCC compiler Version (gcc 3.4.2) --->; gcc 版本

☒ Build/install c++ compiler and libstdc++?

☐ Build/install java compiler and libgcj? 支持的语言, 我没有选择 java

☐ Enable ccache support? 启用 ccache 的支持, 它用于编译时头文件的缓存处理, 用它来编译程序, 第一次会有点慢, 但是以后的速度可就很理想了, 呵呵.....

--- Gdb Options 根据你的需要, 选择 gdb 的支持

Package Selection for the target --->;

这一项我没有选择任意一项, 因为我打算根文件系统及 busybox 等工具链创建成工, 手工来做。

Target Options --->; 文件系统类型, 根据实际需要选, 我用的 ext2;

配置完成后, 编译它:

```
[root@skynet build-tools]#make
```

这一项工作是非常花时间的, 我的工具包全部在本地, 也花去我一小时十三分的时间, 如果全要下载, 我估计网速正常也要多花一两个钟头。

经过漫长的等待 (事实上并不漫长, 去打了几把游戏, 很快过去了):

.....

```
make[1]: Leaving directory `/home/skynet/build-tools/buildroot/build_i386/genext2fs-1.3'
```

```
touch -c /home/skynet/build-tools/buildroot/build_i386/genext2fs-1.3/genext2fs

#-@find /home/skynet/build-tools/buildroot/build_i386/root/lib -type f -name \*.so\* | xargs
/home/skynet/tools/bin/i386-linux-uclibc-strip --remove-section=.comment --remove-
section=.note --strip-unneeded 2>/dev/null || true;

/home/skynet/build-tools/buildroot/build_i386/genext2fs-1.3/genext2fs -i 503 -b 1056 \
-d /home/skynet/build-tools/buildroot/build_i386/root -q -D target/default/device_table.txt
/home/skynet/build-tools/buildroot/root_fs_i386.ext2
```

大功告成!!!

清点战利品

让我来看看它究竟做了哪些事情吧:

```
[root@skynet skynet]# cd tools
```

```
[root@skynet tools]# ls
```

```
bin bin-ccache i386-linux i386-linux-uclibc include info lib libex!ec man usr
```

bin: 所有的编译工具, 如 gcc, 都在这儿了, 只是加了些指定的前缀;

bin-ccache: 如果在 Toolchain optaion 中没有选择对 ccache 的支持, 就没有这一项了;

i386-linux: 链接文件; 实际指向 include

i386-linux-uclibc: uclibc 的相关工具;

include: 供交叉开发工具使用的头文件;

info: gcc 的 info 文件;

lib: 供交叉开发工具使用的链接库文件;

.....

现在可以把编译工具所在目录 XXX/bin 添加至 PATH 了

测试工具链

如果你现在写一个程序, 用 i386-linux-gcc 来编译, 运行的程序会告诉你:

```
./test: linked against GNU libc
```

因为程序运行库会寻到默认的/lib:/usr/lib 上面去, 而我们目前的 uclibc 的库并不在那里(虽然对于目标机来讲, 这是没有错的), 所以, 也只能暂时静态编译, 试试它能否工作了。当然, 你也可以在建好根文件系统后, 试试用 chroot.....

第三章 编译内核

本章的工作, 是为目标机建立一个合适的内核, 对于建立内核, 我想有两点值得考虑的:

- 1、功能上的选择, 应该能够满足需要的情况下, 尽量地小;
- 2、小不是最终目的, 稳定才是;

所以, 最好编译内核前有一份目标机硬件平台清单以及所需功能清单, 这样, 才能更合理地裁减内核。

准备工具

Linux 内核源码, 我选用的是 Linux-2.4.27.tar.bz2

编译内核

将 Linux-2.4.27.tar.bz2 拷贝至\${PRJROOT}/kernel, 解压

```
#cd linux-2.4.27
```

```
//配置
```

```
# make ARCH=i386 CROSS_COMPILE=i386-linux- menuconfig
```

```
//建立源码的依存关系
```

```
# make ARCH=i386 CROSS_COMPILE=i386-linux- clean dep
```

```
//建立内核映像
```

```
# make ARCH=i386 CROSS_COMPILE=i386-linux- bzImage
```

ARCH 指明了硬件平台, CROSS_COMPILE 指明了这是交叉编译, 且编译器的名称为 i386-linux-XXX, 这里没有为编译器指明路径, 是因为我前面已将其加入至环境变量 PATH。

又是一个漫长的等待.....

OK, 编译完成, 673K, 稍微大了点, 要移到其它平台, 或许得想办法做到 512 以下才好, 回头来想办法做这个工作。

安装内核

内核编译好后, 将内核及配置文件拷贝至\${PRJROOT}/images 下。

```
# cp arch/i386/boot/bzImage ${PRJROOT}/images/bzImage-2.4.27-rmk5
```

```
# cp vmlinux ${PRJROOT}/images/vmlinux-2.4.27-rmk5
```

```
# cp System.map ${PRJROOT}/images/System-2.4.27-rmk5
```

```
# cp .config ${PRJROOT}/images/2.4.27-rmk5
```

我采用了后缀名的方式重命名, 以便管理多个不同版本的内核, 当然, 你也可以不用这样, 单独为每个版本的内核在 images 下新建对应文件夹也是可行的。

安装内核模块

完整内核的编译后, 剩下的工作就是建立及安装模块了, 因为我的内核并没有选择模块的支持 (这样扩展性差了一点, 但是对于我的系统来说, 功能基本上定死了, 这样影响也不太大), 所以, 剩下的步骤也省去了, 如果你还需要模块的支持, 应该:

//建立模块

```
#make ARCH=i386 CROSS_COMPILE=i386-linux- modules
```

//安装内核模块至\${PRJROOT}/images

```
#make ARCH=i386 CROSS_COMPILE= i386-linux- \
```

```
>;INSTALL_MOD_PATH=${PRJROOT}/images/modules-2.4.18-rmk5 \
```

```
>;modules_install
```

最后一步是为模块建立依存关系, 不能使用原生的 depmod 来建立, 而需要使用交叉 编译工具。需要用到 busybox 中的 depmod.pl 脚本, 很可惜, 我在 busybox1.0.0 中, 并没有找到这个脚本, 所以, 还是借用了 busybox0.63 中 scripts 中的 depmod.pl。

将 depmod.pl 拷贝至\${PREFIX}/bin 目录中, 也就是交叉编译工具链的 bin 目录。

```
#depmod.pl \
```

```
>;-k ./vmlinux -F ./System.map \
```

```
>;-b ${PRJROOT}/images/modules-2.4.27-rmk5/lib/modules >; \
```

```
>;${PRJROOT}/images/modules-2.4.27-rmk5/lib/modules/2.4.27-rmk5/modules.dep
```

注: 后面讨论移植内核和模块内容时, 我只会提到内核的拷贝, 因为我的系统并没有模块的支持。如果你需要使用模块, 只需按相同方法将其拷贝至相应目录即可。

附, 内核编译清单

附, 内核选择:

内核编译记录:

Code maturity level options 不选

Loadable module support 不选

Processor type and features 根据实际, 选择处理器类型

General setup --->;

[*] Networking support

[*] PCI support

(Any) PCI access mode

[*] PCI device name database

[*] System V IPC

[*] Sysctl support

(ELF) Kernel core (/proc/kcore) format

[*] Kernel support for ELF binaries

[*] Power Management support

Memory Technology Devices (MTD) --->; MTD 设备, 我用 CF 卡, 不选

Parallel port support --->; 不选

Plug and Play configuration --->; 我的系统用不着即插即用, 不选

Block devices --->;

[*] Loopback device support

[*] RAM disk support

(4096) Default RAM disk size (NEW)

[*] Initial RAM disk (initrd) support

Multi-device support (RAID and LVM) --->; 不选

Networking options --->; 基本上都选了

ATA/IDE/MFM/RLL support --->; 用了默认的

Telephony Support --->; 不选

SCSI support --->; 不选

Fusion MPT device support --->; 不选

I2O device support --->; 不选

Network device support --->; 根据实际情况选择

Amateur Radio support --->; 不选

IrDA (infrared) support --->; 不选

ISDN subsystem --->; 不选

Old CD-ROM drivers (not SCSI, not IDE) --->; 不选

Input core support --->; 不选

Character devices --->;

[*] Virtual terminal

[*] Support for console on virtual terminal

[*] Standard/generic (8250/16550 and compatible UARTs) serial support

[*] Support for console on serial port

Multimedia devices --->; 不选

File systems --->;

[*] Kernel automounter version 4 support (also supports v3)

[*] Virtual memory file system support (former shm fs)

[*] /proc file system support

[*] Second extended fs support

Console drivers --->;

[*] VGA text console 调试时接显示器用

剩下三个都不要

Sound --->;

USB support --->;

Kernel hacking --->;

第四章 建立根文件系统

1、建立目录

构建工作空间时, rootfs 文件夹用来存放根文件系统,

```
#cd rootfs
```

根据根文件系统的基本结构, 建立各个对应的目录:

```
# mkdir bin dev etc lib proc sbin tmp usr var root home
```

```
# chmod 1777 tmp
```

```
# mkdir usr/bin usr/lib usr/sbin
```

```
# ls
```

```
dev etc lib proc sbin tmp usr var
```

```
# mkdir var/lib var/lock var/log var/run var/tmp
```

```
# chmod 1777 var/tmp
```

对于单用户系统来说, root 和 home 并不是必须的。

准备好根文件系统的骨架后, 把前面建立的文件安装到对应的目录中去。

2、拷贝链接库

把 uclibc 的库文件拷贝到刚才建立的 lib 文件夹中:

```
# cd ${PREFIX}/lib
```

```
[root@skynet lib]# cp *-*.so ${PRJROOT}/rootfs/lib
```

```
[root@skynet lib]# cp -d *.so.[*0-9] ${PRJROOT}/rootfs/lib
```

3、拷贝内核映像和内核模块

因为没有模块, 所以拷贝模块就省了,

新建 boot 目录, 把刚才建立好的内核拷贝过来

```
# cd /home/kendo/control-project/daq-module/rootfs/
```

```
# mkdir boot
```

```
# cd ${PRJROOT}/images
```

```
# cp bzImages-2.4.18-rmk5 /home/kendo/control-project/daq-module/rootfs/boot
```

4、建立/dev 下边的设备文件

在 linux 中, 所有的设备文件都存放在/dev 中, 使用 mknod 命令创建基本的设备文件。

mknod 命令需要 root 权限, 不过偶本身就是用的 root 用户, 本来是新建了一个用户专门用于嵌入式制作的, 不过后来忘记用了.....

```
# mknod -m 600 mem c 1 1
# mknod -m 666 null c 1 3
# mknod -m 666 zero c 1 5
# mknod -m 644 random c 1 8
# mknod -m 600 tty0 c 4 0
# mknod -m 600 tty1 c 4 1
# mknod -m 600 ttyS0 c 4 64
# mknod -m 666 tty c 5 0
# mknod -m 600 console c 5 1
```

基本的设备文件建立好后, 再创建必要的符号链接:

```
# ln -s /proc/self/fd fd
# ln -s fd/0 stdin
# ln -s fd/1 stdout
# ln -s fd/2 stderr
# ls
console fd mem null random stderr stdin stdout tty tty0 tty1 ttyS0 zero
```

设备文件也可以不用手动创建, 听说 RedHat /dev 下的脚本 MAKEDEV 可以实现这一功能, 不过没有试过.....

基本上差不多了, 不过打算用硬盘/CF 卡来做存储设备, 还需要为它们建立相关文件, 因为我的 CF 在目标机器上是 CF-to-IDE, 可以把它们等同来对待, 先看看 Redhat 下边 had 的相关属性:

```
# ls -l /dev/hda
brw-rw---- 1 root disk 3, 0 Jan 30 2003 /dev/hda
# ls -l /dev/hda1
brw-rw---- 1 root disk 3, 1 Jan 30 2003 /dev/hda1
```

对比一下, 可以看出, had 类型是 b, 即块设备, 主编号为 3, 次编号从 0 递增, 根限位是

rw-rw----, 即 660, 所以:

```
# mknod -m 660 hda b 3 0
```

```
# mknod -m 660 hda1 b 3 1
```

```
# mknod -m 660 hda2 b 3 2
```

```
# mknod -m 660 hda3 b 3 3
```

5、添加基本的应用程序

未来系统的应用程序, 基本上可以分为三类:

基本系统工具, 如 ls、ifconfig 这些.....

一些服务程序, 管理工具, 如 WEB、Telnet.....

自己开发的应用程序

这里先添加基本的系统工具, 有想过把这些工具的代码下载下来交叉编译, 不过实在是麻烦, 用 BusyBox, 又精简又好用.....

将 busybox-1.00.tar.gz 下载至 sysapps 目录下, 解压:

```
#tar zxvf busybox-1.00.tar.gz
```

```
#cd busybox-1.00
```

```
//进入配置菜单
```

```
#make TARGET_ARCH=i386 CROSS=i386-linux- PREFIX=${PRJROOT}/rootfs menuconfig
```

```
//建立依存关系
```

```
#make TARGET_ARCH=i386 CROSS= i386-linux- PREFIX=${PRJROOT}/rootfs dep
```

```
//编译
```

```
#make TARGET_ARCH=i386 CROSS= i386-linux- PREFIX=${PRJROOT}/rootfs
```

```
//安装
```

```
#make TARGET_ARCH=i386 CROSS= i386-linux- PREFIX=${PRJROOT}/rootfs install
```

```
# cd ${PRJROOT}/rootfs/bin
```

```
# ls
```

```
addgroup busybox chown delgroup echo kill ls mv ping rm sleep
```

```
adduser chgrp cp deluser grep ln mkdir netstat ps rmdir umount
```

```
ash chmod date dmesg hostname login mount pidof pwd sh vi
```

一下子多了这么多命令.....

配置 busybox 的说明:

A、如果编译时选择了:

Runtime SUID/SGID configuration via /etc/busybox.conf

系统每次运行命令时, 都会出现 “Using fallback suid method ”

可以将它去掉, 不过我还是在/etc 为其建了一个文件 busybox.conf 搞定;

B、 [*] Do you want to build BusyBox with a Cross Compiler? (i386-linux-gcc) Cross Compiler prefix

这个指明交叉编译器名称 (其实在编译时的命令行已指定过了.....)

C、安装选项下的(\${PRJROOT}/rootfs) BusyBox installation prefix, 这个指明了编译好后的工具的安
装目录。

D、静态编译好还是动态编译好? 即是否选择

[] Build BusyBox as a static binary (no shared libs)

动态编译的最大好处是节省了宝贵空间, 一般来说都是用动态编译, 不过我以前动态编译出过问题 (其实是库的问题, 不关 busybox 的事), 出于惯性, 我选择了静态编译, 为此多付出了 107KB 的空间。

E、其它命令, 根据需要, 自行权衡。

6、系统初始化文件

内核启动时, 最后一个初始化动作就是启动 init 程序, 当然, 大多数发行套件的 Linux 都使用了与 System V init 相仿的 init, 可以在网上下载 System V init 套件, 下载下来交叉编译。另外, 我也找到一篇写得非常不错的讲解如何编写初始化文件的文件, bsd-init, 回头附在后面。不过, 对于嵌入式系 统来讲, BusyBox init 可能更为合适, 在第 6 步中选择命令的时候, 应该把 init 编译进去。

```
#cd ${PRJROOT}/rootfs/etc
```

```
#vi inittab
```

我的 inittab 文件如下:

```
#指定初始化文件
```

```
::sysinit:/etc/init.d/rcS
```

#打开一个串口, 波特率为 9600

```
::respawn:/sbin/getty 9600 ttyS0
```

#启动时执行的 shell

```
::respawn:/bin/sh
```

#重启时动作

```
::restart:/sbin/init
```

#关机时动作, 卸载所有文件系统

```
::shutdown:/bin/umount -a -r
```

保存退出;

再来编写 rcS 脚本:

```
#mkdir ${PRJROOT}/rootfs/etc/init.d
```

```
#cd ${PRJROOT}/rootfs/etc/init.d
```

```
#vi rcS
```

我的脚本如下:

```
#!/bin/sh
```

```
#Set Path
```

```
PATH=/sbin:/bin
```

```
export PATH
```

```
syslogd -m 60
```

```
klogd
```

```
#install /proc
```

```
mount -n -t proc none /proc
```

```
#reinstall root file system by read/write mode(need: /etc/fstab)
```

```
mount -n -o remount,rw /
```

```
#reinstall /proc
```

```
mount -n -o remount,rw -t proc none /proc
```

```
#set lo ip address
```

```
ifconfig lo 127.0.0.1
```

```
#set eth0 ip address
```

#当然, 这样子做只是权宜之计, 最后做的应该是在这一步引导网络启动脚本, 像 RedHat

#那样, 自动读取所有指定的配置文件来启动

```
ifconfig eth0 192.168.0.68 netmask 255.255.255.0
```

```
#set route
```

#同样的, 最终这里应该是运行启动路由的脚本, 读取路由配置文件

```
route add default gw 192.168.0.1
```

#还差一个运行服务程序的脚本, 哪位有现成的么?

#网卡/路由/服务这三步, 事实上可以合在一步, 在 rcS 这一步中, 做一个循环, 运行指定启动目录下的所有脚, 先将就着这么做吧, 确保系统能够正常启动了, 再来写这个脚本。

```
#set hostname
```

```
hostname MyLinux
```

保存退出。

编写 fstab 文件

```
#vi fstab
```

我的 fstab 很简单:

```
/dev/hda1 / ext2 defaults 1 1
```

```
none /proc proc defaults 0 0
```

第五章 让 MyLinux 能够启动

前一章, 我们把编译好的内核、应用程序、配置文件都拷贝至 rootfs 目录对应的子 目录中去了, 这一步, 就是把这些文件移植至目标机的存储器。这里, 我是先另外拿一块硬盘, 挂在我的开发机上做的测试, 因为我的本本用来写文档, PC 机用来 做开发机, 已经没有另外的机器了.....但是本章只是讲述一个一般性的过程, 并不影响你直接在目标主机上的工作。

因为以后目标机识别硬盘序号都是 hda, 而我现在直接挂上去, 则会是 hdb、hdc.....这样, 安装 lilo 时有点麻烦 (虽然也可以实现)。所以我想了一个办法:

把新硬盘挂在 IDE0 的 primary 上, 进入 linux 后, 会被认为是 had; v

原来主机的装 Redhat 的硬盘, 我将它从 IDE0 的 primary 上变到了 IDE1 的 primary, 因为它的 lilo 早已装好, 基本上不影响系统的使用; v

分区和格式化

BIOS 中改为从第二个硬盘启动; 也就是从我原来开发机启动, 新的硬盘被识别成了 hda。

```
#fdisk /dev/hda
```

用 d 参数删除已存在的所有分区

用 n 参数新建一个分区, 也是就/dev/hda1

格式化

```
#mkfs.ext2 /dev/hda1
```

安装 bootloader

因为我是 X86 平台, 所以直接用了 lilo, 如果你是其这平台, 当然, 有许多优秀的 bootloader 供你选择, 你只需查看其相应的说明就可以了。

编译 lilo 配置文件, 我的配置文件名为 target.lilo.conf, 置于\${PRJROOT}/rootfs/etc 目录。内容如下所示:

```
boot=/dev/hda
```

```
disk=/dev/hda
```

```
bios=0x80
```

```
image=/boot/bzImage-2.4.18-rmk5
```

```
label=Linux
```

```
root=/dev/hda1
```

```
append="root=/dev/hda1"
```

```
read-only
```

```
//新建文件夹, 为 mount 做新准备
```

```
#mkdir /mnt/cf
```

```
//把目标硬盘 mount 上来
```

```
#mount -t ext2 /dev/hdc1 /mnt/cf
```

```
回到 rootfs
```

```
#cd ${PRJROOT}/rootfs
```

拷贝所有文件至目标硬盘

```
#cp -r * /mnt/cf
```

这样, 我们所有的文件都被安装至目标硬盘了, 当然, 它还不能引导, 因为没有 bootloader。使用如下命令:

```
# lilo -r /mnt/cf -C etc/target.lilo.conf
```

Warning: LBA32 addressing assumed

Added Linux *

-r : 改变根目标为/mnt/cf , 这样配置文件其实就是/mnt/cf/etc/target.lilo.conf, 也就是我们先前建立的文件。

当然, 完成这一步, 需要 lilo22.3 及以后版本, 如果你的版本太旧, 比如 Redhat9.0 自带的, 就会出现下面的信息:

```
#lilo -r /mnt/cf -C etc/target.lilo.conf
```

Fatal: open /boot/boot.b: No such file or directory

这时, 你需要升级你的 lilo, 或者重新安装一个。

启动系统

```
#umount /mnt/cf
```

```
#reboot
```

将 BIOS 改为从 IDE0 启动, 也就是目标硬盘。如果一切顺利, 你将顺利进入一个属于你的系统。

回头再来看看我们的工作空间吧

```
[root@skynet lib]# df /dev/hda1
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
------------	-----------	------	-----------	------	------------

/dev/hda1	3953036	1628	3750600	1%	/mnt/cf
-----------	---------	------	---------	----	---------

总共花去了我 1628KB 的空间, 看来是没有办法放到软盘里边去了^o^, 不过一味求小, 并不是我的目标。

```
[root@skynet skynet]# ls ${PRJROOT}
```

bootloader build-tools debug doc images kernel rootfs sysapps tmp tools

这几个目录中的文件, 呵呵, 与本文一开头规划的一样

```
[root@skynet skynet]# ls build-tools/
```

```
buildroot buildroot-0.9.27.tar.tar
```

包含了 buildroot 源码及压缩包, 事实上 buildroot 下边还包括了 GNU 其它工具的源码、编译文件等诸多内容, 是我们最重要的一个文件夹, 不过到现在它已经没有多大用处了, 如果你喜欢, 可以将它删除掉 (不建议)。

```
[root@skynet skynet]# ls images
```

```
2.4.18-rmk5 bzImage-2.4.18-rmk5 System-2.4.18-rmk5 vmlinux-2.4.18-rmk5
```

内核映像及配置文件等, 如果你有模块, 因为还有相应的目录

```
[root@skynet skynet]# ls kernel/
```

```
linux-2.4.27 linux-2.4.27.tar.bz2
```

内核源码及压缩包

```
[root@skynet skynet]# ls rootfs/
```

```
bin boot dev etc home lib linuxrc proc root sbin tmp usr var
```

制作好的根文件系统, 重中之重, 注意备份.....

```
[root@skynet skynet]# ls sysapps/
```

```
busybox-1.00 busybox-1.00.tar.gz
```

busybox-1.00 源码包, 或许你还要继续添加/删除一些命令.....

```
[root@skynet skynet]# ls tools
```

```
bin i386-linux i386-linux-uclibc include info lib man
```

这个也很重要, 我们制作好的交叉开发工具链。如果你要继续开发程序, 这个目录重要性就很高了。

其它目录暂时是空的。

第六章 完善 MyLinux

关于进一步的调试, 你可以在开发机上使用 `chroot /mnt/cf /bin/sh` 这样的命令, 以使我们在目标根文件系统上工作。

支持多用户

因为我在编译 busybox 时, 已经将它的多用户那一大堆命令编译了进来。现在关键是要为其建立相应的文件;

进入原来的开发机, 进入 rootfs 目录, 切换根目录

```
#chroot rootfs/ /bin/sh
```

A、建立/etc/passwd 文件, 我的文件内容如下:

```
root:x:0:0:root:/root:/bin/bash
```

B、建立/etc/group 文件, 我的文件内容如下:

```
root:x:0:
```

```
bin:x:1:
```

```
sys:x:2:
```

```
kmem:x:3:
```

```
tty:x:4:
```

```
tape:x:5:
```

```
daemon:x:6:
```

```
disk:x:7:
```

C、为 root 建立密码

```
#passwd root
```

试试用 addgroup/addusr.....这堆命令。然后重启, 从目标硬盘上启动; 从 console □, 9600 登陆试试 (因为我在 inittab 中启用了 ttyS0, 我未来的目标机, 是没有显卡的, 需要从 console □或 SSH 进去管理)

```
MyLinux login: root
```

```
Password:
```

```
BusyBox v1.00 (2004.10.10-04:43+0000) Built-in shell (ash)
```

```
Enter 'help' for a list of built-in commands.
```

```
~ #
```

成功了.....

增加 WEB Server

Busybox 里边有 httpd 选项, 不过我编译时并没有选择, 所以还是自己来安装。我使用的软件是 tthttpd-2.25b.tar.gz, 将它移至 sysapps 目录下。

```
[root@skynet sysapps]# tar zxvf thttpd-2.25b.tar.gz
```

```
[root@skynet sysapps]# cd thttpd-2.25b
```

//配置

```
[root@skynet thttpd-2.25b]# CC=i386-linux-gcc ./configure --host=$TARGET
```

.....

```
i386-linux-gcc -static httpasswd.o -o httpasswd -lcrypt
```

```
make[1]: warning: Clock skew detected. Your build may be incomplete.
```

```
make[1]: Leaving directory `/home/skynet/sysapps/thttpd-2.25b/extras'
```

//拷贝至根文件目录

```
[root@skynet thttpd-2.25b]# cp thttpd ${PRJROOT}/rootfs/usr/sbin
```

//trip 处理

```
[root@skynet thttpd-2.25b]# i386-linux-strip ${PRJROOT}/rootfs/usr/sbin/thttpd
```

剩下的, 就发挥各人的想像吧.....

独孤九贱 2005-11-1 02:34

继续补完

不好意思, 最近工作太忙, 好久没有来搞这个东东了, 加之心爱的手机丢了, 心情又不太好, 不过一切都过去了, 继续来将它补充完整。

修改启动脚本

在前面写 rcS 启动脚本中。当时只是为了系统能够正常地启动, 在启动网卡/路由/服务等时, 有如下语句:

```
-----  
#set lo ip address
```

```
ifconfig lo 127.0.0.1
```

```
#set eth0 ip address
```

#当然, 这样子做只是权宜之计, 最后做的应该是在这一步引导网络启动脚本, 像 RedHat

#那样, 自动读取所有指定的配置文件来启动

```
ifconfig eth0 192.168.0.68 netmask 255.255.255.0

#set route

#同样的, 最终这里应该是运行启动路由的脚本, 读取路由配置文件

route add default gw 192.168.0.1

.....
```

这样配置的最大坏处就是不能根据配置文件自定义, 每次开机都定死了, 现在来修改它, 将这段语句删除之, 换成如下语句:

```
for i in /etc/start/S??* ;do

# Ignore dangling symlinks (if any).

[ ! -f "$i" ] && continue

echo "Running $i ."

case "$i" in

*.sh)

# Source shell script for speed.

(

trap - INT QUIT TSTP

set start

. $i

)

;;

*)

# No sh extension, so fork subprocess.

$i start

;;
```

```
esac

echo "Done $i ."

echo

done
```

解释一下, 这段语句的作用, 就是启动/etc/start/目录下, 所有以 S 开头的脚本文件, 可以启动两类, 以 sh 结尾或没有 sh 后缀的。

这样, 我们在/etc/目录下再新建一目录 start/, 这里面就是我们启动时需要的脚本的。先来启动网卡。

修改网卡配置文件

我是根据 Red hat 的作法, 把网卡配置放在/etc/sysconfig/network-scripts目录下, 类似于 ifcfg-ethXX 这样子, 它们的语法是:

```
DEVICE=eth0

BOOTPROTO=static

BROADCAST=88.88.88.255

IPADDR=88.88.88.44

NETMASK=255.255.255.0

NETWORK=88.88.88.0

ONBOOT=yes
```

好, 建立这些目录和文件, 我共有两个文件 ifcfg-ethXX。回到/etc/start 目录, 建立网卡的启动脚本 S01interface:

```
#!/bin/sh

. /etc/sysconfig/network

#enable ip_forward

echo >1 /proc/sys/net/ipv4/ip_forward

#enable syn_cookie

echo >1 /proc/sys/net/ipv4/tcp_syncookies

#enable loopback interface

/sbin/ifconfig lo 127.0.0.1
```

```
#enable ethernet interface

/usr/sbin/bootife

#set hostname

if [ -z "$HOSTNAME" -o "$HOSTNAME" = "(none)" ]

then

HOSTNAME=localhost

fi

/bin/hostname ${HOSTNAME}
```

请注意这个脚本文件, 有两个地方:

1、包含了另一个配置文件/etc/sysconfig/network, 在这里, 我也是照抄了 redhat, 我的 /etc/sysconfig/network 这个文件的内容如下:

```
NETWORKING=yes

HOSTNAME=skynet

GATEWAY=88.88.88.2
```

2、在启动网卡时, 我使用了

```
#enable ethernet interface

/usr/sbin/bootife
```

bootife 是我自己写的一个 C 程序, 作用是读取/etc/sysconfig/network-scripts/下面的 ifcfg-ethXX 文件, 并配置之, 本来 这里就该用 shell 来完成更合适一点, 无奈, 偶 shell 功底实在差了一点(刚学几天), 就是想从 Redhat 中照抄过来, 反复试了几次也没有成功。所以被逼无奈用 C 来完成之, 后面我会附上我的 C 的源码, 也希望哪位大哥能够写一段 Shell 的程序代替它, 放上来大家共享之。

修改路由启动文件

同样的, 我在/etc/start 下, 建立新脚本 S02route.sh, 它的作用是启动所有配置的静态路由:

```
#!/bin/bash

. /etc/sysconfig/network

# Add non interface-specific static-routes.

if [ -f /etc/sysconfig/static-routes ]
```



```
then

grep "eth*" /etc/sysconfig/static-routes | while read ignore args ; do

# echo "/sbin/route add -"$args

/sbin/route add - $args

done

fi

#Add default gw

/sbin/route add default gw ${GATEWAY}
```

OK, 启动时读取的配置文件是/etc/sysconfig/static-routes , 它的语法和 Redhat 是一样的, 请参照建立此文件。

启动服务程序

新建启动脚本 S03server:

```
#!/bin/sh

#-----

#-- Source

#-- Author(s) : kendo

#-- Email: [email]kendo999@sohu.com[/email]

#-- http://www.skynet.org.cn

#-- 2005/10/31

#-----

. /etc/sysconfig/bootserver

if [ "$enable_httpd" = 1 ] ; then

. /etc/scripts/httpd.sh $1

fi

if [ "$enable_adsl" = 1 ] ; then

.....
```

```

fi

if [ "$enable_udhcpd" = 1 ] ; then

.....

fi

```

很简单, 根据相应变量的值, 调用相应的脚本。

1、这些启动标志变量, 我定义在了/etc/sysconfig/bootserver 当中, 其内容如下:

```

#start server on system boot

#1:yes 0:no

enable_httpd=1

enable_adsl=1

enable_udhcpd=1

```

2、每种服务对应的脚本, 我都放在了/etc/scripts 下面。这些脚本, 取决于你打算使用哪些服务程序了。脚本的来源, 可以自己编写, 有可能其源码中自带有, 也可以到网上查找.....我就不再一一赘述了,

OK, 基本上, 脚本的修改就完成了, 下一步, 将是建立 RamDisk。

附, 读取网卡配置文件, 启动网卡的 C 源码:

```

/*****

** author:kendo

** date:2005/10/26

*****/

#include

#include

#include

#include

#include

#include

```

```
#define NETCFGDIR "/etc/sysconfig/network-scripts/"

struct _ifcfg{

char device[8];

char bootproto[8];

char br[16];

char netmask[16];

char ip[16];

char network[16];

int onboot;

};

void ParseKey(struct _ifcfg *ifcfg, char *key, char *value)

{

if(!strcmp(key, "DEVICE"))

{

strcpy(ifcfg->device, value);

}

else if(!strcmp(key, "BOOTPROTO"))

{

strcpy(ifcfg->bootproto, value);

}

else if(!strcmp(key, "BROADCAST"))

{

strcpy(ifcfg->br, value);

}

else if(!strcmp(key, "IPADDR"))

{
```

```
strcpy(ifcfg->ip, value);
}

else if(!strcmp(key, "NETMASK"))
{
strcpy(ifcfg->netmask, value);
}

else if(!strcmp(key, "NETWORK"))
{
strcpy(ifcfg->network, value);
}

else if(!strcmp(key, "ONBOOT"))
{
ifcfg->onboot=(strcmp(value, "yes") ? 0 : 1);
}
}

int main(int argc, char **argv)
{
FILE *fp;
DIR *dir;
int i;
char filename[50], buf[80];
char *index, *key, *value, *p;
struct _ifcfg *ifcfg;
struct dirent *ptr;
ifcfg=(struct _ifcfg *)malloc(sizeof(struct _ifcfg));
memset(ifcfg, 0, sizeof(struct _ifcfg));
```

```
dir=opendir(NETCFGDIR); /*打开脚本目录*/

while((ptr=readdir(dir))!=NULL) /*读取所有文件*/
{
    if(strncmp(ptr->d_name,"ifcfg-eth",9)) /*这里, 只启动了以太网卡^o^*/
    {
        continue;
    }

    memset(filename,0,sizeof(filename));

    sprintf(filename,"%s%s",NETCFGDIR,ptr->d_name);

    if((fp=fopen(filename,"r"))==NULL) /*打开配置文件*/
    {
        continue;
    }

    while(!feof(fp))
    {
        memset(buf,0,sizeof(buf));

        if(fgets(buf,80,fp)!=NULL) /*逐行读取分析*/
        {
            p=strchr(buf,'n');

            if(p)
            {
                *p=' ';
            }

            index=buf;

            key=strtok(index,"="); /*读取配置变量*/

            value=strtok(NULL,"="); /*读取变量的值*/
```

```
ParseKey(ifcfg, key, value); /*分析之, 存入结构 ifcfg 中*/
}
}

/*构建相应的命令*/
memset(buf, 0, 80);
strcpy(buf, "/sbin/ifconfig");
if(ifcfg->onboot)
{
    sprintf(buf, "%s %s %s netmask %s broadcast %s",
    buf,
    ifcfg->device,
    ifcfg->ip,
    ifcfg->netmask,
    ifcfg->br);
    /*直接调用 system 来实现, 当然也可以自己通过 ioctl 来设置, 相应源码, 我以前在 c/c++版发过*/
    system(buf);
}
}

free(ife);
return 0;
}

platinum 2005-11-1 02:52

[code]

memset(buf, 0, 80);
strcpy(buf, "/sbin/ifconfig");
if(ifcfg->onboot)
```

```
{  
  
    sprintf(buf, "%s %s %s netmask %s broadcast %s",  
  
    buf,  
  
    ifcfg->device,  
  
    ifcfg->ip,  
  
    ifcfg->netmask,  
  
    ifcfg->br);  
  
    /*直接调用 system 来实现, 当然也可以自己通过 ioctl 来设置, 相应源码, 我以前在 c/c++版发过*/  
  
    system(buf);  
  
[/code]
```

两个问题

1、从这段代码看, 实际调用了 /sbin/ifconfig 来完成网卡的设置, 那么, 这个程序是否必须用 root 来执行?

2、为何不用 system("command") 呢?

独孤九贱 2005-11-1 03:03

回复 19 楼 platinum 的帖子

1、ifconfig 本身运行, 应该不需要 root 吧? 而至于在 shell 中的运行权限, 要看看相应的权限位了, 事实上它已经能够在我的系统中很好的运行了, 测试过很多次的。不过现在我的系统, 其实没有用这种方法的, 我是自己封装了一个网卡管理的库, 也就是重写了 ifconfig, 不过要把这些代码发上来, 太麻烦了, 所以, 就用了解 system 简单了一点。

2、我不是很理解 “为何不用 system("command") 呢?” 这句话的含义, 我用的是 system(buf);你说的是不是为什么要去构建一个 buf, 而不是直接用 system("/sbin/ifconfig ethXX.....")? 清楚一点.....^o^

独孤九贱 2005-11-16 01:44

继续工作, 交叉编译 SNMP

一般系统都会有 SNMP 的支持, 下载了 net-snmp-5.1.3.1, 先看看 INSTALL 和 FAQ 文档 (因为以前从来没有碰过这个东东, 见笑了.....), 按照说明, 在原生主机上安装了一回, 安装完成后, 发现在指定安装目录下主要包括了几块文件:

bin: SNMP 的一些功能脚本和程序;

sbin:主要的代理程序和 trap 程序: snmpd 和 snmptrap

include/lib:自身兼容及第三方开发所需的头文件及库文件;

share: 主要是 MIB 文件;

然后回到安装目录下, 运行./configure --help, 仔细查看了其安装编译选项, 因为我定位的小型系统, 只需具备基本的 SNMP 功能即可, 所以:

那些 bin 目录下的功能程序也不需要, 对应--disable-applications

bin 下的脚本也是不需要的, 对应: --disable-scripts

用户手册也不需要: --disable-manuals

关闭 ipv6 支持: --disable-ipv6

还有一个--enable-mini-agent 选项, 说明是编译出一个最小化的 snmpd, 比较有趣, 试试先。

对于交叉编译, 还需要用--host 指明目标平台。

OK, 看完了帮助说明, 开始编译了:

1、配置, 根据以上确定的选项:

```
[root@skynet root]# CC=i386-linux-gcc ./configure --host=$TARGET --enable-mini-agent --  
disable-ipv6 --with-endianness=little --disable-applications --disable-manuals --disable-  
scripts --disable-ucd-snmp-compatibility
```

CC 指明了编译器; --host 指明了我的目标平台, 这个环境变量在我前面定义的 devedaq 脚本中。

还算顺利, 继续编译它:

```
[root@skynet net-snmp-5.1.3.1]# make LDFLAGS="-static"
```

呵呵, 因为没有装 lib 库, 所以我用了-static 选项, 指明是静态编译;

3、安装

安装就需要指明安装路径了, 路径可以在.config 的时候指定, 因为那个时候, 那串东东太长了, 我在 install 时指定也不迟:

```
#make prefix=${TARGET_PREFIX} exec_prefix=${TARGET_PREFIX} install
```

4、检查一下:

```
[root@skynet net-snmp-5.1.3.1]# ls -l ${TARGET_PREFIX}/sbin
```

```
total 2120
```

```
-rwxr-xr-x 1 root root 2164301 Nov 16 09:22 snmpd
```


snmpd 就是我们要的代理主程序了, 大约静态编译有 2M。

```
[root@skynet net-snmp-5.1.3.1]# ls -l ${TARGET_PREFIX}/bin
```

```
total 4380
```

```
-rwxr-xr-x 2 root root 391980 Oct 14 2004 ar
```

```
-rwxr-xr-x 2 root root 581228 Oct 14 2004 as
```

```
.....
```

呵呵, 那堆程序和脚本没有安装, 如 snmpwalk.....

```
ls ${TARGET_PREFIX}/lib
```

```
ls -l ${TARGET_PREFIX}/include
```

看看我们需要的 mib 文件:

```
[root@skynet net-snmp-5.1.3.1]# ls ${TARGET_PREFIX}/share/snmp
```

```
mib2c.access_functions.conf mib2c.column_defines.conf mib2c.int_watch.conf mib2c.old-api.conf
```

```
mib2c.array-user.conf mib2c.column_enums.conf mib2c.iterate_access.conf mib2c.scalar.conf
```

```
mib2c.check_values.conf mib2c.conf mib2c.iterate.conf mibs
```

```
mib2c.check_values_local.conf mib2c.create-dataset.conf mib2c.notify.conf snmpconf-data
```

5、移植

基本完成了, 因为 snmpd 太大了点, 对它进行 strip 处理:

先备个份:

```
[root@skynet net-snmp-5.1.3.1]# cp ${TARGET_PREFIX}/sbin/snmpd  
${TARGET_PREFIX}/sbin/snmpd.bak
```

```
[root@skynet net-snmp-5.1.3.1]# i386-linux-strip ${TARGET_PREFIX}/sbin/snmpd
```

```
[root@skynet net-snmp-5.1.3.1]# ls -l ${TARGET_PREFIX}/sbin/snmpd
```

```
-rwxr-xr-x 1 root root 503300 Nov 16 09:30 /home/skynet/tools/i386-linux/sbin/snmpd
```

经过处理后, 还有近 500KB 了。

因为只有 SNMP agent 功能, 即 snmpd 程序, 其它的都可以忽略。用了静态编译, lib 下边那些 libnetsnmp 文件都可以不需要了, 程序运行

需要 MIB 库, 也就是 share 下的内容, 把这两个东东拷到 rootfs 相应的目录中去:

```
[root@skynet net-snmp-5.1.3.1]# cp ${TARGET_PREFIX}/sbin/snmpd ${PRJROOT}/rootfs/usr/sbin
```

```
[root@skynet net-snmp-5.1.3.1]# mkdir -p ${PRJROOT}/rootfs/usr/local/share
```

```
[root@skynet net-snmp-5.1.3.1]# cp -r ${TARGET_PREFIX}/share/snmp  
${PRJROOT}/rootfs/usr/local/share
```

```
[root@skynet net-snmp-5.1.3.1]# cp EXAMPLE.conf  
${PRJROOT}/rootfs/usr/local/share/snmp/snmpd.conf
```

最后一步是把安装目录下的配置文件范例拷到 snmpd 启动时默认搜索目录中去。

6、测试

打开 snmpd.conf 看看:

```
[root@skynet net-snmp-5.1.3.1]# vi ${PRJROOT}/rootfs/usr/local/share/snmp/snmpd.conf
```

有如下语句:

```
# sec.name source community  
  
com2sec local localhost COMMUNITY  
  
com2sec mynetwork NETWORK/24 COMMUNITY
```

定义了两个用户, 本地及网络的, 以及它们的通读密钥, 按自己的需要修改一下, 如:

```
# sec.name source community  
  
com2sec local 127.0.0.1 public  
  
com2sec mynetwork 0.0.0.0 public
```

后面是定义用户的用户组等一大堆东东, 事实上不用修改它们了。运行它:

```
[root@skynet net-snmp-5.1.3.1]# chroot ${PRJROOT}/rootfs /bin/sh
```

```
BusyBox v1.00 (2004.10.13-06:32+0000) Built-in shell (ash)
```

```
Enter 'help' for a list of built-in commands.
```

```
/ # snmpd
```

```
/ # exit
```

在我们自己的根文件系统环境下运行它, 然后退出来。用 ps 查看:

```
#ps -aux
```

```
.....
```

```
root 32270 0.0 0.3 1212 936 ? S 09:38 0:00 snmpd
```

```
[root@skynet net-snmp-5.1.3.1]# netstat -anu
```

```
.....
```

```
udp 0 0 0.0.0.0:161 0.0.0.0:*
```

呵呵, 已经成功启动了。用一个 SNMP 管理软件试试, 可以成功地获取到信息。OK!

总结一下:

1、主程序+MIB 库大了点, 共计约 2M, 不过我确实没有办法再小了, 而且一味求小, 也不是我的目的。

2、功能稍微简单了些, 只有 agent, 如果需要, 可以类似地把其它程序加上去就可以了。

3、第一次玩 net-snmp, 还是有点生疏, 比如我静态编译二进制程序, 并不需要 include/lib 下的文件, 但是如何关闭它们呢? 我试过--disable-ucd-snmp-compatibility, 不过好像不是这个选项.....下次改进了.....

独孤九贱 2005-11-16 05:38

继续工作, 使用 ramdisk

前提: 内核编译时得选相应的支持选项, 前文已有叙述。

1、rootfs 中的/boot 文件夹删除;

2、建立 ramdisk:

使用 dd 命令建立一个空的文件系统映像:

```
# dd if=/dev/zero of=images/initrd.img bs=1k count=8192
```

大小\8192K, 用/dev/zero 对其初始化;

利用刚才的空的文件系统映像, 建立文件系统并安装它, 使用了 mke2fs 命令:

```
# /sbin/mke2fs -F -v -m0 images/initrd.img
```

新建一个临时文件夹做 mount 之用:

```
# mkdir tmp/initrd
```

把建好的文件系统 mount 上来:

```
#mount -o loop images/initrd.img tmp/initrd
```

把根文件系统拷贝过来:

```
#cp -av rootfs/* tmp/initrd
```

```
# umount tmp/initrd
```

压缩:

```
# gzip -9 images/initrd.bin
```

这样, 就得到了 images/initrd.bin

把目标盘 mount 上来:

```
#mount -t ext2 /dev/hda1 /mnt/cf
```

新建一个/boot

```
#mkdir /mnt/cf/boot
```

把刚才建立的 ramdisk 镜像拷过来。然后把内核文件 bzImage-2.4.27-rmk5 也拷进去。

这样, boot 文件夹里边有两个文件

initrd.bin

bzImage-2.4.27-rmk5

这个时候还不能安装 lilo, 因为 lilo 的配置文件中 /dev/hda..... 这样的东东, 而目标盘上还没有..... 所以, 临时建一个:

```
#mkdir /mnt/cf/dev
```

```
#cp -rf ${PRJROOT}/rootfs/dev/hda* /mnt/cf/dev
```

修改 \${PRJROOT}/rootfs/etc/target.lilo.conf, 我的配置文件如下:

```
boot=/dev/hda
```

```
disk=/dev/hda
```

```
bios=0x80
```

```
image=/boot/bzImage-2.4.27-rmk5
```

```
initrd=/boot/initrd.bin
```

```
root=/dev/hda1
```

```
append="root=/dev/hda1"
```

```
# label=MyLinux
```

```
read-only
```

相比以前的, 只是加了一句: `initrd=/boot/initrd.bin`, 另外把 `label` 去掉了, 因为否则 `lilo` 会报怨说语法错误。

好了, 可以安装 `lilo` 了。以前我们的语句是:

```
lilo -r /mnt/cf -C etc/target.lilo.conf
```

现在我们的目标盘上没有 `etc` 这个目录了, 更不用说 `target.lilo.conf`, 可以借助于工程目录中的了原文件, 当然, 我在目标硬盘上新建了 `/dev`, 然后把 `target.lilo.conf` 拷过去, 还是用这句命令安装 `lilo`。

这样, 整个系统就完成了。

移植 U-Boot 到 S3C2440

本文是针对在友善之臂公司出品的以 S3C2440 为核心的 mini2440 开发板上实现 U-Boot-2008.10 的移植。其中存储介质为一片 64 MB 的 NAND Flash(K9F1208), 一片 2MB 的 NOR Flash(SST-39VF1601), 两片 32 MB 的 SDRAM(HY57V561620FTP), 网卡芯片为 DM9000。

1. U-Boot 移植

U-Boot 的源码是通过 GCC 和 Makefile 组织编译的。顶层目录下的 Makefile 首先设置开发板的定义, 然后递归地调用各级子目录下的 Makefile, 最后把编译过的程序链接成 U-Boot 映像。

移植 U-Boot 工作就是添加开发板相关的文件、配置选项, 然后配置编译。开发移植 U-Boot 前, 要熟悉硬件电路板和处理器。确认 U-Boot 是否已经支持新开发板的处理器和 I/O 设备, 比较出硬件配置最接近的开发板。选择的原则是, 首先处理器相同, 其次是处理器体系结构相同, 然后是以太网等外围接口。

U-Boot 移植过程如下:

1.1 移植准备

修改 Makefile 文件, 在 U-Boot 中建立自己的开发板文件(以友善之臂的 sbc2410x 为基础)。

1.1.1 添加开发板的配置选项

进入 U-Boot 根目录, 修改 Makefile 文件, 参考 smdk2410 的配置选项修改如下:

```
smdk2410_config      :      unconfig
    @$(MKCONFIG) $(@:_config=) arm arm920t smdk2410 NULL s3c24x0

mini2440_config      :      unconfig
    @$(MKCONFIG) $(@:_config=) arm arm920t mini2440 NULL s3c24x0
```

各项的意思如下:

arm : CPU 的架构 (ARCH)
arm920t : CPU 的类型 (CPU), 其对应于 cpu/arm920t 子目录。
mini2440: 开发板的型号 (BOARD), 对应于 board/ mini2440 目录。
NULL: 开发者/或经销商 (vender), NULL 为没有。
s3c24x0 : 片上系统 (SOC)。

同时在 “ifndef CROSS_COMPILE ” 之前加上自己交叉编译器的路径, 比如我使用 crosstool-0.43 制作的基于 2.6.27.4 内核和 gcc-4.1.1-glibc-2.3.2 的 ARM9TDMI 交叉编译器, 则:

```
CROSS_COMPILE=arm-9tdmi-linux-gnu-
```

特别注意:

在 u-boot1.3.3 及以上版本 Makefile 有一定的变化, 使得对于 24x0 处理器从 nand 启动的遇到问题。也就是网上有人说的: 无法运行过 lowlevel_init。其实这个问题是由于编译器将我们自己添加的用于 nandboot 的子函数 nand_read_ll 放到了 4K 之后造成的 (到这不理解的话, 请仔细看看 24x0 处理器 nandboot 原理)。运行失败后, 利用 mini2440 的 4 个 LED 调试发现 u-boot 根本没有完成自我拷贝, 然后看 uboot 根目录下的 System.map 文件可知道原因。解决办法其实很简单, 将下面这个语句:

```
__LIBS := $(subst $(obj),,$(LIBS)) $(subst $(obj),,$(LIBBOARD))
```

改为:

```
__LIBS := $(subst $(obj),,$(LIBBOARD)) $(subst $(obj),,$(LIBS))
```

1.1.2 在/board 子目录中建立自己的开发板 mini2440 目录

目录结构为 board/mini2440。如果开发者/经销商(vender)不为 NULL, 则目录结构为 board/vender_name/mini2440, 否则编译会出错。然后, 将 smdk2410 目录下的文件考入此目录中, 并将其中的 smdk2410.c 改名为 mini2440.c。同时还得修改 board/mini240/Makefile 文件:

```
COBJS := mini2440.o flash.o
```

1.1.3 在 include/configs/中建立配置头文件

将 smdk2410.h 复制一份在相同目录下, 并改名为 mini2440.h。

1.1.4 测试编译能否成功

回到 U-Boot 主目录, (若之前有编译过, 最好执行一下 make clean) make mini2440_config, 再 make, 编译生成 u-boot.bin 成功。

1.2 修改 U-Boot 中的文件, 以同时匹配 2440 和 2410

1.2.1 修改/cpu/arm920t/start.S

(1) 删除 AT91RM9200 使用的 LED 代码

```
#include <config.h>
#include <version.h>
// #include <status_led.h> /*这是针对 AT91RM9200DK 开发板的 LED 代码, 注释掉。*/
.....
start_code:
    /*
     * set the cpu to SVC32 mode
     */
    mrs r0, cpsr
    bic r0, r0, #0x1f
    orr r0, r0, #0xd3
    msr cpsr, r0
    // bl coloured_LED_init
    // bl red_LED_on
```

(2) 修改编译条件支持2440, 修改寄存器地址定义

```
#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
    /* turn off the watchdog */

#if defined(CONFIG_S3C2400)
#define pWTCON 0x15300000
#define INTMSK 0x14400008 /* Interrupt-Controller base addresses */
#define CLKDIVN 0x14800014 /* clock divisor register */
#else
#define pWTCON 0x53000000
#define INTMSK 0x4A000008 /* Interrupt-Controller base addresses */
#define INTSUBMSK 0x4A00001C
#define CLKDIVN 0x4C000014 /* clock divisor register */
#endif
#define CLK_CTL_BASE 0x4C000000
#define MDIV_405 0x7f << 12
#define PSDIV_405 0x21
```



```
#define UPLL_MDIV_48 0x38 << 12
```

```
#define UPLL_PSDIV_48 0x22
```

```
#define MDIV_200 0xa1 << 12
```

```
#define PSDIV_200 0x31
```

(3)修改中断禁止部分

```
#if defined(CONFIG_S3C2410)
```

```
    ldr r1, =0x7ff /*根据2410芯片手册, INTSUBMSK 有11位可用*/
```

```
    ldr r0, =INTSUBMSK
```

```
    str r1, [r0]
```

```
#endif
```

```
#if defined(CONFIG_S3C2440)
```

```
    ldr r1, =0x7fff /*根据2440芯片手册, INTSUBMSK 有15位可用*/
```

```
    ldr r0, =INTSUBMSK
```

```
    str r1, [r0]
```

```
#endif
```

(4)修改时钟设置 (2440的主频为405MHz。)

注释掉原来的代码:

```
/* FCLK:HCLK:PCLK = 1:2:4 */
```

```
/* default FCLK is 120 MHz ! */
```

```
//ldr    r0, =CLKDIVN
```

```
//mov    r1, #3
```

```
//str    r1, [r0]
```

```
//#endif      /* CONFIG_S3C2400 || CONFIG_S3C2410 */
```

改为:

```
# if defined(CONFIG_S3C2440)
```

```
/* FCLK:HCLK:PCLK = 1:4:8 */
```

```
ldr r0, =CLKDIVN
```

```
mov r1, #5
```

```
str r1, [r0]
```

```
mrc p15, 0, r1, c1, c0, 0 /*read ctrl register */
```

```
orr r1, r1, #0xc0000000 /*Asynchronous */
```

```
mcr p15, 0, r1, c1, c0, 0 /*write ctrl register */
```

```
/*now, CPU clock is 405.00 Mhz */
```

```
mov r1, #CLK_CTL_BASE
```

```
mov r2, #UPLL_MDIV_48    /*UPLL */
```

```

add r2, r2, #UPLL_PSDIV_48
str r2, [r1, #0x08] /*write UPLL first, 48MHz */

mov r2, #MDIV_405 /* mpll_405mhz */
add r2, r2, #PSDIV_405 /* mpll_405mhz */
str r2, [r1, #0x04] /* MPLLCON */

#else
/* FCLK:HCLK:PCLK = 1:2:4 */
/* default FCLK is 12 MHz ! 在这里 U-Boot 有一个错误: 以为默认时钟为120MHz。
其实如果没有添加以下设置 FCLK 的语句, 芯片内部的 PLL 是无效的, 即 FCLK 为12MHz。
S3C24x0的芯片手册说得很明白。*/
ldr    r0, =CLKDIVN
mov     r1, #3
str     r1, [r0]

mrc p15, 0, r1, c1, c0, 0 /*read ctrl register */
orr r1, r1, #0xc0000000 /*Asynchronous */
mcr p15, 0, r1, c1, c0, 0 /*write ctrl register */

/*now, CPU clock is 202.8 Mhz */
mov r1, #CLK_CTL_BASE
mov r2, #MDIV_200 /* mpll_200mhz */
add r2, r2, #PSDIV_200 /* mpll_200mhz */
str r2, [r1, #0x04]

# endif
#endif          /*CONFIG_S3C2400 || CONFIG_S3C2410 || CONFIG_S3C2440*/

```

(5) 将从 Flash 启动改成从 NAND Flash 启动

在以下 U-Boot 的设置堆栈语句段(作用是将 u-boot 的源代码从 nor flash 到 sdram 中):

```

/* Set up the stack */
stack_setup:
    ldr r0, _TEXT_BASE          /* upper 128 KiB: relocated uboot
    的前面添加上以下的 nand boot 代码:
#ifdef CONFIG_S3C2440_NAND_BOOT
    @ reset NAND
    mov     r1, #NAND_CTL_BASE
    ldr     r2, =( (7<<12) | (7<<8) | (7<<4) | (0<<0) )
    str     r2, [r1, #oNFCNF]

```

```
    ldr    r2, [r1, #oNFCNF]

    ldr    r2, =( (1<<4) | (0<<1) | (1<<0) ) @ Active low CE Control
    str    r2, [r1, #oNFCNT]
    ldr    r2, [r1, #oNFCNT]

    ldr    r2, =(0x6) @ RnB Clear
    str    r2, [r1, #oNFSTAT]
    ldr    r2, [r1, #oNFSTAT]

    mov    r2, #0xff @ RESET command
    strb   r2, [r1, #oNFCMD]

    mov    r3, #0 @ wait
nand1:
    add    r3, r3, #0x1
    cmp    r3, #0xa
    blt    nand1

nand2:
    ldr    r2, [r1, #oNFSTAT] @ wait ready
    tst    r2, #0x4
    beq    nand2

    ldr    r2, [r1, #oNFCNT]
    orr    r2, r2, #0x2 @ Flash Memory Chip Disable
    str    r2, [r1, #oNFCNT]

    @ get read to call C functions (for nand_read())
    ldr    sp, DW_STACK_START @ setup stack pointer
    mov    fp, #0 @ no previous frame, so fp=0

    @ copy U-Boot to RAM
    ldr    r0, =TEXT_BASE
    mov    r1, #0x0
    mov    r2, #0x30000

    bl     nand_read_ll

    tst    r0, #0x0
```

```
        beq      ok_nand_read

bad_nand_read:
loop2:
        b        loop2    @ infinite loop

ok_nand_read:
        @ verify
        mov      r0, #0
        ldr      r1, =TEXT_BASE
        mov      r2, #0x400    @ 4 bytes * 1024 = 4K-bytes
go_next:
        ldr      r3, [r0], #4
        ldr      r4, [r1], #4
        teq      r3, r4
        bne      notmatch
        subs     r2, r2, #4
        beq      stack_setup
        bne      go_next

notmatch:
loop3:
        b        loop3    @ infinite loop

#endif

#ifdef CONFIG_S3C2410_NAND_BOOT
        @ reset NAND
        mov      r1, #NAND_CTL_BASE
        ldr      r2, =0xf830    @ initial value
        str      r2, [r1, #oNFCNF]
        ldr      r2, [r1, #oNFCNF]
        bic      r2, r2, #0x800 @ enable chip
        str      r2, [r1, #oNFCNF]
        mov      r2, #0xff      @ RESET command
        strb     r2, [r1, #oNFCMD]

        mov      r3, #0    @ wait
nand1:
        add      r3, r3, #0x1
```

```
        cmp     r3, #0xa
        blt     nand1

nand2:
        ldr     r2, [r1, #oNFSTAT]      @ wait ready
        tst     r2, #0x1
        beq     nand2

        ldr     r2, [r1, #oNFCONF]
        orr     r2, r2, #0x800 @ disable chip
        str     r2, [r1, #oNFCONF]

        @ get read to call C functions (for nand_read())
        ldr     sp, DW_STACK_START      @ setup stack pointer
        mov     fp, #0 @ no previous frame, so fp=0

        @ copy U-Boot to RAM
        ldr     r0, =TEXT_BASE
        mov     r1, #0x0 @start address
        mov     r2, #0x30000 @code size
        bl      nand_read_ll
        tst     r0, #0x0
        beq     ok_nand_read

bad_nand_read:
loop2:
        b       loop2 @ infinite loop

ok_nand_read:
        @ verify
        mov     r0, #0
        ldr     r1, =TEXT_BASE
        mov     r2, #0x400 @ 4 bytes * 1024 = 4K-bytes
go_next:
        ldr     r3, [r0], #4
        ldr     r4, [r1], #4
        teq     r3, r4
        bne     notmatch
        subs    r2, r2, #4
        beq     stack_setup
```

```

        bne      go_next

notmatch:
loop3:
        b        loop3    @ infinite loop

#endif
在“ldr    pc, _start_armboot”之前加入:
#if defined(CONFIG_mini2440_LED)
        @ LED1 on u-boot stage 1 is ok!
        mov     r1, #GPIO_CTL_BASE
        add     r1, r1, #oGPIO_B
        ldr     r2, =0x155aa
        str     r2, [r1, #oGPIO_CON]
        mov     r2, #0xff
        str     r2, [r1, #oGPIO_UP]
        mov     r2, #0x1c0
        str     r2, [r1, #oGPIO_DAT]
#endif

```

修改目的: 如果看到只有 LED1亮了, 并响起蜂鸣器, 说明 U-Boot 的第一阶段已完成! (针对 mini2440, 若不是这块开发板的, 必须修改或不添加)。

在 “ _start_armboot: .word start_armboot ” 后加入:

```

        .align 2
DW_STACK_START: .word    STACK_BASE+STACK_SIZE-4

```

1.2.2 添加 Nand Flash 读取函数

在 board/ mini2440加入 NAND Flash 读取函数 (start.S 中需要的 nand_read_ll 函数) 文件 `nand_read.c` , 内容如下:

```

#include <config.h>

#define __REGb(x) (*(volatile unsigned char *) (x))
#define __REGi(x) (*(volatile unsigned int *) (x))
#define NF_BASE 0x4e000000

```

```
#if defined(CONFIG_S3C2440)

#define NFCONF __REGi(NF_BASE + 0x0)
#define NFCONT __REGi(NF_BASE + 0x4)
#define NFCMD __REGb(NF_BASE + 0x8)
#define NFADDR __REGb(NF_BASE + 0xC)
#define NFDATA __REGb(NF_BASE + 0x10)
#define NFSTAT __REGb(NF_BASE + 0x20)

//#define GPDAT __REGi(GPIO_CTL_BASE+oGPIO_F+oGPIO_DAT)

#define NAND_CHIP_ENABLE (NFCONT &= ~(1<<1))
#define NAND_CHIP_DISABLE (NFCONT |= (1<<1))
#define NAND_CLEAR_RB (NFSTAT |= (1<<2))
#define NAND_DETECT_RB { while(! (NFSTAT&(1<<2))) ; }

#define BUSY 4
inline void wait_idle(void) {
    while(!(NFSTAT & BUSY));
    NFSTAT |= BUSY;
}

#define NAND_SECTOR_SIZE 512
#define NAND_BLOCK_MASK (NAND_SECTOR_SIZE - 1)

/* low level nand read function */
int
nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
{
    int i, j;

    if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK)) {
        return -1; /* invalid alignment */
    }

    NAND_CHIP_ENABLE;

    for(i=start_addr; i < (start_addr + size);) {
```

```
/* READ0 */
    NAND_CLEAR_RB;
    NFCMD = 0;

    /* Write Address */
    NFADDR = i & 0xff;
    NFADDR = (i >> 9) & 0xff;
    NFADDR = (i >> 17) & 0xff;
    NFADDR = (i >> 25) & 0xff;

    NAND_DETECT_RB;

    for(j=0; j < NAND_SECTOR_SIZE; j++, i++) {
        *buf = (NFDATA & 0xff);
        buf++;
    }
}
NAND_CHIP_DISABLE;
return 0;
}
#endif

#if defined(CONFIG_S3C2410)

#define NFCONF __REGi(NF_BASE + 0x0)
#define NFCMD __REGb(NF_BASE + 0x4)
#define NFADDR __REGb(NF_BASE + 0x8)
#define NFDATA __REGb(NF_BASE + 0xc)
#define NFSTAT __REGb(NF_BASE + 0x10)
#define BUSY 1

#define NAND_SECTOR_SIZE 512
#define NAND_BLOCK_MASK (NAND_SECTOR_SIZE - 1)

inline void wait_idle(void) {
    int i;
    while(!(NFSTAT & BUSY))
        for(i=0; i<10; i++);
}
```



```
}
/* low level nand read function */
int
nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
{
    int i, j;
    if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK)) {
        return -1;    /* invalid alignment */
    }
    /* chip Enable */
    NFCONF &= ~0x800;
    for(i=0; i<10; i++);
    for(i=start_addr; i < (start_addr + size);) {
        /* READ0 */
        NFCMD = 0;
        /* Write Address */
        NFADDR = i & 0xff;
        NFADDR = (i >> 9) & 0xff;
        NFADDR = (i >> 17) & 0xff;
        NFADDR = (i >> 25) & 0xff;
        wait_idle();
        for(j=0; j < NAND_SECTOR_SIZE; j++, i++) {
            *buf = (NFDATA & 0xff);
            buf++;
        }
    }
    /* chip Disable */
    NFCONF |= 0x800; /* chip disable */
    return 0;
}
#endif
```

另外还要修改 board/ /mini2440/Makefile 文件, 将 nand_read.c 编译进 u-boot:

OBJS := mini2440.o nand_read.o flash.o

1.2.3 修改 board/mini2440/lowlevel_init.S 文件

(1)修改 BWSCON, mini2440 BANK0接 NOR Flash, BANK4接 DM9000, BANK6接 RAM, 对于 mini2440, 只要修改 B4_BWSCON 即可:

```
#define B3_BWSCON          (DW16 + WAIT + UBLB)
//#define B4_BWSCON        (DW16)
#define B4_BWSCON          (DW16 + WAIT + UBLB)
#define B5_BWSCON          (DW16)
.....

#define B4_Tacs             0x0      /* 0clk */
#define B4_Tcos             0x3      /* 4clk */
#define B4_Tacc             0x7      /* 14clk */
#define B4_Tcoh             0x1      /* 1clk */
#define B4_Tah              0x3      /* 4clk */
#define B4_Tacp             0x6      /* 6clk */
#define B4_PMC              0x0      /* normal */
#define B5_Tacs             0x0      /* 0clk */
```

(2)然后修改 REFRESH 的刷新周期:

```
#define B7_SCAN            0x1      /* 9bit */
/* REFRESH parameter */
#define REFEN              0x1      /* Refresh enable */
#define TREFMD             0x0      /* CBR(CAS before RAS)/Auto refresh */
#define Trc                0x3      /* 7clk */
#define Tchr               0x2      /* 3clk */
//#define REFCNT            1113    /* period=15.6us, HCLK=60Mhz,
(2048+1-15.6*60) */

# if defined(CONFIG_S3C2440)
#define Trp                0x2      /* 4clk */
#define REFCNT             1012
#else
#define Trp                0x0      /* 2clk */
#define REFCNT             0x4f4 /* period=7.8125us, HCLK=100Mhz, (2048+1-7.8125*100)
*/
```

```
#endif
_TEXT_BASE:
    .word    TEXT_BASE
```

1.2.4 修改 GPIO 和 PLL 的配置

在/board/ mini2440/mini2440.c 中对 GPIO 和 PLL 的配置进行修改(参阅开发板的硬件说明和芯片手册):

```
#elif FCLK_SPEED==1                /* Fout = 202.8MHz */
    /*#define M_MDIV        0xA1
#define M_PDIV 0x3
#define M_SDIV 0x1*/

    #if defined(CONFIG_S3C2410)
        /* Fout = 202.8MHz */
        #define M_MDIV    0xA1
        #define M_PDIV    0x3
        #define M_SDIV    0x1
    #endif
    #if defined(CONFIG_S3C2440)
        /* Fout = 405MHz */
        #define M_MDIV 0x7f
        #define M_PDIV 0x2
        #define M_SDIV 0x1
    #endif
    /*-----*/
#endif
.....

#elif USB_CLOCK==1
    /*#define U_M_MDIV        0x48
#define U_M_PDIV        0x3*/
    #if defined(CONFIG_S3C2410)
        #define U_M_MDIV    0x48
        #define U_M_PDIV    0x3
```

```
#endif

#if defined(CONFIG_S3C2440)
#define U_M_MDIV 0x38
#define U_M_PDIV 0x2
#endif
#define U_M_SDIV 0x2
#endif
```

为连接 LED 和蜂鸣器的 GPIO 修改配置寄存器:

```
/* set up the I/O ports */
gpio->GPACON = 0x007FFFFFFF;
/*gpio->GPBCON = 0x00044555;*/
#if defined(CONFIG_mini2440_LED)
    gpio->GPBCON = 0x00055555;
#else
    gpio->GPBCON = 0x00044555;
#endif

gpio->GPBUP = 0x000007FF;
```

为引导 linux 内核, 修改开发板的类型代码:

```
gpio->GPHUP = 0x000007FF;
/* arch number of SMDK2410-Board */
//gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;

#if defined(CONFIG_S3C2410)
/* arch number of SMDK2410-Board */
gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;
#endif
#if defined(CONFIG_S3C2440)
/* arch number of S3C2440 -Board */
gd->bd->bi_arch_number = MACH_TYPE_mini2440 ;
#endif

/* adress of boot parameters */
```

为使 int board_init (void)设置完成后, LED1和 LED2同时亮起, 在 int board_init (void)的最后添加:

```
icache_enable();
dcache_enable();
```

```
# if defined(CONFIG_mini2440_LED)
    gpio->GPBDAT = 0x180;

//int board_init (void)设置完成后, LED1和 LED2会亮起!
#endif

return 0;
```

1.2.5 修改 u-boot 支持烧写 yaffs 映像文件

(1) 在/common/cmd_nand.c 中 do_nand 函数中, 加入代码, 实现对 nand write.yaffs 命令的支持。在对 jffs2操作的下面加入, 如下:

```
在 else if (s != NULL && !strcmp(s, ".oob"))前加入
else if(s != NULL && !strcmp(s, ".yaffs"))
{
    if(read)
    {
        printf("nand read.yaffs is not provide!"); /*不支持读取 yaffs*/
    }
    else
    {
        nand->writeoob = 1;
        ret = nand_write_skip_bad(nand, off, &size, (u_char
*)addr); //写入 yaffs

        nand->writeoob = 0;
    }
}
```

在 nand 的命令中加入对 nand write.yaffs 的描述, 加入如下3行:

```
"    to/from memory address 'addr', skipping bad blocks.\n"
"nand write.yaffs - addr off|partition size\n"
"    write 'size' bytes starting at offset 'off'\n"
"    to/from yaffs image in memory address 'addr', skipping bad
blocks.\n"
"nand erase [clean] [off size] - erase 'size' bytes from\n"
```

(2) 在 include/linux/mtd/mtd.h 的 mtd_info 结构体定义中加入两个变量如下:

```
struct mtd_info {
    u_char writeoob;
    u_char skipfirstblock;
    u_char type;
    u_int32_t flags;
    u_int32_t size; /* Total size of the MTD */
}
```

(3) 在 `drivers/mtd/nand/nand_util.c` 的 `nand_write_skip_bad` 函数中加两段程序, 一段是为了计算正常数据的长度, 一段是为了在写入一段数据后, 数据指针能正常跳到下一段数据, 修改如下:

```
size_t left_to_write = *length;
size_t len_incl_bad;
u_char *p_buffer = buffer;
```

/*这段程序主要是从 yaffs 映像中提取要写入的正常数据的长度, 即不包括 oob 数据的数据长度*/

```
if(nand->writeoob==1)
{
    size_t oobsize = nand->oobsize; //定义 oobsize 的大小
    size_t datasize = nand->writesize;

    int datapages = 0;

    //长度不是528整数倍, 认为数据出错。
    if (((*length)%(nand->oobsize+nand->writesize)) != 0) {
        printf ("Attempt to write error length data!\n");
        return -EINVAL;
    }

    datapages = *length/(datasize+oobsize);

    *length = datapages*datasize;
    left_to_write = *length;
    nand->skipfirstblock=1;
}
```

```
/* Reject writes, which are not page aligned */
```

以下的程序程序本意是没有坏块时, 不进行坏块检查直接写, 但没有达到目的, 所以注释掉:

```

    /*if (len_incl_bad == *length) {
        printf("first write\n");
        rval = nand_write (nand, offset, length, buffer);
        if (rval != 0) {
            printf ("NAND write to offset %x failed %d\n",
                    offset, rval);
            return rval;
        }
    }*/

```

在 while(left_to_write>0)循环中添加:

```

        offset += nand->erasesize - block_offset;
        continue;
    }

```

块。

```

    {
        nand->skipfirstblock=0;
        printf ("Skip the first good block 0x%08x\n",
                offset & ~(nand->erasesize - 1));
        offset += nand->erasesize - block_offset;
        continue;
    }
    if (left_to_write < (nand->erasesize - block_offset))
        write_size = left_to_write;

```

修改 p_buffer 处的代码如下:

```

        left_to_write -= write_size;
        printf("%ld%% is complete.", 100-(left_to_write/(*length/100)));
        offset += write_size;
        //p_buffer += write_size;
        if(nand->writeoob==1)
        {
            p_buffer += write_size+(write_size/nand->writesize*nand->oobsize);
        }
        else
        {
            p_buffer += write_size;
        }
    }

```

(4)在/drivers/mtd/nand/nand_base.c 的 nand_write 函数中, 加入一段把正常数据与 oob 数据分离的代码, 再加入页写时的模式设置为 MTD_OOB_RAW, 在页写时, 不进行 ECC 的校验, ECC 的校验在 yaffs 的 oob 数据中已自带了, 不能重写。此模式下, 写入正常数据后, 会把 oob 的数据写入 nand 的 oob 区。修改后如下:

```
static int nand_write(struct mtd_info *mtd, loff_t to, size_t len,
    size_t *retlen, const uint8_t *buf)
{
    struct nand_chip *chip = mtd->priv;
    int ret;
    //printf("0 len is %d ", len);
    //此段数据是即将写入的数据中的正常数据移到 buf 中的前段, 把 oob 数据移到后
    段。

    int oldopsmode = 0;
    if(mtd->writeoob==1)
    {
        size_t oobsize = mtd->oobsize; //定义 oobsize 的大小
        size_t datasize = mtd->writesize;
        int i = 0;
        uint8_t oobtemp[16];
        int datapages = 0;
        datapages = len/(datasize); //传进来的 len 是没有包括 oob 的数据长度
        for(i=0;i<(datapages);i++)
        {
            memcpy(oobtemp, buf+datasize*(i+1), oobsize);
            memmove(buf+datasize*(i+1), buf+datasize*(i+1)+oobsize, (datapages-
            (i+1))*(datasize)+(datapages-1)*oobsize);
            memcpy(buf+(datapages)*(datasize+oobsize)-
            oobsize, oobtemp, oobsize);
        }

        //printf("1 len is %d ", len);
        /* Do not allow reads past end of device */
        if ((to + len) > mtd->size)
            return -EINVAL;
        if (!len)
            //return 0;
    }
}
```



```

        {printf("Write data length is %d ", len); return 0;}

nand_get_device(chip, mtd, FL_WRITING);

chip->ops.len = len;
chip->ops.datbuf = (uint8_t *)buf;
//chip->ops.oobbuf = NULL;

if(mtd->writeoob!=1)
{
    chip->ops.oobbuf = NULL;
}
else
{
    chip->ops.oobbuf = (uint8_t *) (buf+len); //将 oob 缓存的指针指向
buf 的后段, 即 oob 数据区的起始地址。
    chip->ops.ooblen = mtd->oobsize;
    oldopsmode = chip->ops.mode;
    chip->ops.mode = MTD_OOB_RAW; //将写入模式改为直接书写
oob 区, 即写入数据时, 不进行 ECC 校验的计算和写入。 (yaffs 映像的 oob 数据中, 本身就带
有 ECC 校验)
}

ret = nand_do_write_ops(mtd, to, &chip->ops);

*retlen = chip->ops.retlen;

nand_release_device(mtd);
chip->ops.mode = oldopsmode; //恢复原模式
return ret;
}
.....

/* Do not replace user supplied command function ! */
if (mtd->writesize > 512 && chip->cmdfunc == nand_command)
    chip->cmdfunc = nand_command_lp;

//MTDDEBUG (MTD_DEBUG_LEVEL0, "NAND device: Manufacturer ID:"
//      " 0x%02x, Chip ID: 0x%02x (%s %s)\n", *maf_id, dev_id,
//      nand_manuf_ids[maf_idx].name, type->name);

```

```
printf("NAND device: Manufacturer ID:"  
      " 0x%02x, Chip ID: 0x%02x (%s %s)\n", *maf_id, dev_id,  
      nand_manuf_ids[maf_idx].name, type->name);  
return type;  
}
```

1.2.6 修改 DM9000 驱动

修改文件 u-boot-2008.10/drivers/net/dm9000x.c

```
/* Enable TX/RX interrupt mask */  
DM9000_iow(DM9000_IMR, IMR_PAR);  
  
/* 注释掉与 MII 接口相关的语句----*/  
#if 0  
    i = 0;  
    while (!(phy_read(1) & 0x20)) {          /* autonegation complete bit */  
        udelay(1000);  
        .....  
        break;  
    }  
    printf("mode\n");  
#endif  
return 0;  
}
```

1.2.7 Nand Flash 驱动

首先, 要说明一下 CFG_NAND_LEGACY 的使用。在 u-boot 的 drivers/mtd/下有两个目录, 分别是 nand 和 nand_legacy。在 nand 目录下的是 nand 的初始化函数和 nand 的操作读写函数, 是使用 linux 的 mtd 构架。此目录下的文件, 只有在定义了 CFG_CMD_NAND 宏和没有定义 CFG_NAND_LEGACY 宏的情况下才会被编译。在 nand_legacy 目录下的文件也是是实现 nand 相关

操作命令, 如 read, write 等命令的功能, 但不是使用 linux 的 mtd 构架。此目录下的文件, 只有在定义了 CFG_CMD_NAND 和定义了 CFG_NAND_LEGACY 宏的情况下才会定义。此目录下的文件 u-boot 已不推荐使用。在本移植过程中采用不定义 CFG_NAND_LEGACY 的方式。

(1) 修改/cpu/arm920t/s3c24x0/nand. c

先加入 S3C2440 NAND flash 控制器的地址定义, 修改后如下:

```
#define __REGb(x)      (*(volatile unsigned char *) (x))
#define __REGi(x)      (*(volatile unsigned int *) (x))
```

```
#if defined(CONFIG_S3C2410)
```

```
#define      NF_BASE      0x4e000000
#define      NFCONF      __REGi(NF_BASE + 0x0)
#define      NFCMD      __REGb(NF_BASE + 0x4)
.....
```

```
#define NFEOCC0      __REGb(NF_BASE + 0x14)
#define NFEOCC1      __REGb(NF_BASE + 0x15)
#define NFEOCC2      __REGb(NF_BASE + 0x16)
```

```
#else
```

```
#if defined(CONFIG_S3C2440)
```

```
#define NF_BASE      0x4e000000
#define NFCONF      __REGi(NF_BASE + 0x0)
#define NFCONT      __REGi(NF_BASE + 0x4)
#define NFCMD      __REGb(NF_BASE + 0x8)
#define NFADDR      __REGb(NF_BASE + 0xc)
#define NFDATA      __REGb(NF_BASE + 0x10)
#define NFMECCD0      __REGi(NF_BASE + 0x14)
#define NFMECCD1      __REGi(NF_BASE + 0x18)
#define NFSECCD      __REGi(NF_BASE + 0x1c)
#define NFSTAT      __REGb(NF_BASE + 0x20)
#define NFSTAT0      __REGi(NF_BASE + 0x24)
#define NFSTAT1      __REGi(NF_BASE + 0x28)
#define NFMECC0      __REGi(NF_BASE + 0x2c)
#define NFMECC1      __REGi(NF_BASE + 0x30)
#define NFSECC      __REGi(NF_BASE + 0x34)
#define NFSBLK      __REGi(NF_BASE + 0x38)
#define NFECLK      __REGi(NF_BASE + 0x3c)
```

```

#define S3C2440_NFCNT_nCE      (1<<1)
#define S3C2440_ADDR_NALE 0x0c
#define S3C2440_ADDR_NCLE 0x08

#endif
#endif
#define S3C2410_NFCNTF_EN      (1<<15)
#define S3C2410_NFCNTF_512BYTE (1<<14)

```

u-boot. 2008. 10自带的 S3C2410的 s3c2410_hwcontrol 函数有错。在此函数中, 把 chip->IO_ADDR_W 值改写了, 导致在写数据时出现错误。解决方法是使用一全局变量代替 chip->IO_ADDR_W。在 s3c2410_hwcontrol 函数上一行定义这个全局变量, 然后修改 s3c2410_hwcontrol 函数, 让它支持 S3C2440, 修改后如下:

```

#define S3C2410_ADDR_NCLE 8

ulong IO_ADDR_W = NF_BASE;
static void s3c2410_hwcontrol(struct mtd_info *mtd, int cmd, unsigned int ctrl)
{
    struct nand_chip *chip = mtd->priv;

    DEBUGN("hwcontrol(): 0x%02x 0x%02x\n", cmd, ctrl);

#ifdef CONFIG_S3C2410
    if (ctrl & NAND_CTRL_CHANGE) {
        //ulong IO_ADDR_W = NF_BASE;
        IO_ADDR_W = NF_BASE;
        if (!(ctrl & NAND_CLE))
            IO_ADDR_W |= S3C2410_ADDR_NCLE;
        if (!(ctrl & NAND_ALE))
            IO_ADDR_W |= S3C2410_ADDR_NALE;

        //chip->IO_ADDR_W = (void *)IO_ADDR_W;

        if (ctrl & NAND_NCE)
            NFCNTF &= ~S3C2410_NFCNTF_nFCE;
        .....
    }

    if (cmd != NAND_CMD_NONE)

```

```
        //writeb(cmd, chip->IO_ADDR_W);
        writeb(cmd, (void *)IO_ADDR_W);
    #endif
    #if defined(CONFIG_S3C2440)
        if (ctrl & NAND_CTRL_CHANGE) {
            IO_ADDR_W = NF_BASE;
            if (!(ctrl & NAND_CLE)) //要写的是地址
            {
                IO_ADDR_W |= S3C2440_ADDR_NALE;}
            if (!(ctrl & NAND_ALE)) //要写的是命令
            {
                IO_ADDR_W |= S3C2440_ADDR_NCLE;}

            //chip->IO_ADDR_W = (void *)IO_ADDR_W;

            if (ctrl & NAND_NCE)
                {NFCONT &= ~S3C2440_NFCONT_nCE; //使能 nand flash
                //DEBUGN("NFCONT is 0x%x ", NFCONT);
                //DEBUGN("nand Enable ");
                }
            else
                {NFCONT |= S3C2440_NFCONT_nCE; //禁止 nand flash
                //DEBUGN("nand disable ");
                }
        }

        if (cmd != NAND_CMD_NONE)
            writeb(cmd, (void *)IO_ADDR_W);
            //writeb(cmd, chip->IO_ADDR_W);
    #endif
}
```

修改 board_nand_init 函数如下:

```
    DEBUGN("board_nand_init()\n");

    clk_power->CLKCON |= (1 << 4);

    #if defined(CONFIG_S3C2410)
        DEBUGN("CONFIG_S3C2410\n");
```

```
/* initialize hardware */
twrph0 = 3; twrph1 = 0; tacls = 0;

.....

nand->cmd_ctrl = s3c2410_hwcontrol;

nand->dev_ready = s3c2410_dev_ready;
#else
#if defined(CONFIG_S3C2440)
    DEBUGN("CONFIG_S3C2440\n");
    twrph0 = 4; twrph1 = 2; tacls = 0;
    cfg = (tacls<<12) | (twrph0<<8) | (twrph1<<4);
    NFCONF = cfg;
    //DEBUGN("cfg is %x\n", cfg);
    //DEBUGN("NFCONF is %lx\n", NFCONF);
    cfg = (1<<6) | (1<<4) | (0<<1) | (1<<0);
    NFCONT = cfg;
    //DEBUGN("cfg is %lx\n", cfg);
    //DEBUGN("NFCONT is %x\n", NFCONT);
/* initialize nand_chip data structure */
    nand->IO_ADDR_R = nand->IO_ADDR_W = (void *)0x4e000010;

    /* read_buf and write_buf are default */
    /* read_byte and write_byte are default */

    /* hwcontrol always must be implemented */
    nand->cmd_ctrl = s3c2410_hwcontrol;

    nand->dev_ready = s3c2410_dev_ready;
#endif
#endif

#ifdef CONFIG_S3C2410_NAND_HWECC
```

1.2.8 修改 u-boot-2008.10/cpu/arm920t/s3c24x0/interrupts.c 文件

(1)在有 S3C2410的宏定义开关里加入对 S3C2440的支持:

```
#include <common.h>
#if defined(CONFIG_S3C2400) || defined (CONFIG_S3C2410) || defined (CONFIG_TRAB)
|| defined (CONFIG_S3C2440)
#include <arm920t.h>
#if defined(CONFIG_S3C2400)
#include <s3c2400.h>
#elif defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
#include <s3c2410.h>
#endif
```

(2)加入对 mini2440的支持:

```
#elif defined(CONFIG_SBC2410X) || \
    defined(CONFIG_SMDK2410) || \
    defined(CONFIG_VCMA9) || \
    defined(CONFIG_mini2440)

    tbclk = CFG_HZ;
#else
#    error "tbclk not configured"
```

1.2.9 修改 cpu\arm920t\s3c24x0\speed.c , 修改根据时钟寄存器来计算时钟的方法。

(1)头文件对 S3C2440的支持:

```
#include <common.h>
#if defined(CONFIG_S3C2400) || defined (CONFIG_S3C2410) || defined (CONFIG_TRAB)
|| defined (CONFIG_S3C2440)
#include <arm920t.h>
#if defined(CONFIG_S3C2400)
#include <s3c2400.h>
#elif defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
#include <s3c2410.h>
#endif
```

(2) 由于 S3C2410 和 S3C2440 的 MPLL、UPLL 计算公式不一样, 所以 get_PLLCLK 函数也需要修改:

```
m = ((r & 0xFF000) >> 12) + 8;
p = ((r & 0x003F0) >> 4) + 2;
s = r & 0x3;
#ifdef CONFIG_S3C2440
    if (pllreg == MPLL)
        return((CONFIG_SYS_CLK_FREQ * m * 2) / (p << s));
    else if (pllreg == UPLL)
        return((CONFIG_SYS_CLK_FREQ * m) / (p << s));
#endif
```

(3) 由于 S3C2410 和 S3C2440 的设置方法也不一样, 所以 get_HCLK 函数也需要修改:

```
ulong get_HCLK(void)
{
    S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
#ifdef CONFIG_S3C2440
    if (clk_power->CLKDIVN & 0x6)
    {
        if ((clk_power->CLKDIVN & 0x6)==2) return(get_FCLK()/2);
        if ((clk_power->CLKDIVN & 0x6)==6) return((clk_power->CAMDIVN &
0x100) ? get_FCLK()/6 : get_FCLK()/3);
        if ((clk_power->CLKDIVN & 0x6)==4) return((clk_power->CAMDIVN &
0x200) ? get_FCLK()/8 : get_FCLK()/4);
        return(get_FCLK());
    }

    else return(get_FCLK());
#else
    return((clk_power->CLKDIVN & 0x2) ? get_FCLK()/2 : get_FCLK());
#endif

    //return((clk_power->CLKDIVN & 0x2) ? get_FCLK()/2 : get_FCLK());
}
/* return PCLK frequency */
```


1.2.10 修改 u-boot-2008.10/drivers/usb/usb_ohci.c , 添加对 2440 的支持

```
defined(CONFIG_S3C2400) || \
defined(CONFIG_S3C2410) || \
defined(CONFIG_S3C2440) || \
defined(CONFIG_S3C6400) || \
defined(CONFIG_440EP) || \
defined(CONFIG_PCI_OHCI) || \
defined(CONFIG_MPC5200) || \
defined(CFG_OHCI_USE_NPS)
# define OHCI_USE_NPS          /* force NoPowerSwitching mode */
#endif
```

1.2.11 添加 mini2440 的机器 ID

在 u-boot-2008.10/include/asm-arm/mach-types.h 文件中添加 (需要和内核的 ID 保持一致) :

```
#define MACH_TYPE_IMX27IPCAM          1871
#define MACH_TYPE_NEMOC              1872
#define MACH_TYPE_GENEVA             1873
#define MACH_TYPE_mini2440           782
```

1.2.12 修改 u-boot-2008.10/include/configs/mini2440.h 头文件

(1)添加对2440的宏定义:

```
#define CONFIG_ARM920T                1          /* This is an ARM920T Core      */
#define CONFIG_S3C2440                1 /* in a SAMSUNG S3C2440 SoC */
```

```
#define CONFIG_mini2440          1 /* on a SAMSUNG mini2440 Board */
#define CONFIG_mini2440_LED      1 /* Use the LED on Board */
//#define CONFIG_S3C2410 1 /* in a SAMSUNG S3C2410 SoC */
//#define CONFIG_SMDK2410 1 /* on a SAMSUNG SMDK2410 Board */
/* input clock of PLL */
```

(2)修改网卡的宏定义, 注释掉 CS8900的部分, 添加对 DM9000的支持:

```
/*
 * Hardware drivers
 */
//#define CONFIG_DRIVER_CS8900 1 /* we have a CS8900 on-board */
//#define CS8900_BASE          0x19000300
//#define CS8900_BUS16        1 /* the Linux driver does accesses as shorts */

#define CONFIG_DRIVER_DM9000          1
#define CONFIG_DM9000_USE_16BIT      1
#define CONFIG_DM9000_BASE          0x20000300
#define DM9000_IO                    0x20000300
#define DM9000_DATA                  0x20000304
```

(3)添加支持 Nand 启动等宏定义, 将相关 IP 设置的注释去掉, 并修改 IP 设置, 顺便修改下启动参数的宏设置:

```
#define CONFIG_CMD_CACHE
#define CONFIG_CMD_DATE
#define CONFIG_CMD_ELF

#define CONFIG_CMD_PING

#define CONFIG_SETUP_MEMORY_TAGS
#define CONFIG_INITRD_TAG
#define CONFIG_CMDLINE_TAG
/*向 linux 内核传递函数所需的宏*/

//nand Flash param

#define CONFIG_CMD_NAND

#define CONFIG_CMDLINE_EDITING

#ifdef CONFIG_CMDLINE_EDITING
#undef CONFIG_AUTO_COMPLETE
```

```

#else
#define CONFIG_AUTO_COMPLETE
#endif

//#define CONFIG_NAND_LEGACY
//不使用 LEGACY, 以使用自带的 nand flash 驱动
/*
 * NAND flash settings
 */
#if defined(CONFIG_CMD_NAND)
#define CFG_NAND_BASE 0x4E000000
/* NandFlash 控制器在 SFR 区起始寄存器地址 */
#define CFG_MAX_NAND_DEVICE 1
/* Max number of NAND devices */

#define CONFIG_MTD_NAND_VERIFY_WRITE 1 //使能 flash 写校验

/* #undef CONFIG_MTD_NAND_VERIFY_WRITE */
#endif /* CONFIG_CMD_NAND */

#define CONFIG_BOOTDELAY 2
#define CONFIG_BOOTARGS "noinitrd root=/dev/nfs
nfsroot=192.168.1.10:/opt/root_nfs
ip=192.168.1.70:192.168.1.10:192.168.1.10:255.255.255.0:mini.arm9.net:eth0:off
init=linuxrc console=ttySAC0"
#define CONFIG_ETHADDR 08:00:3e:26:0a:5b
#define CONFIG_NETMASK 255.255.255.0
#define CONFIG_IPADDR 192.168.1.70 /*改变默认的 IP 地址*/
#define CONFIG_SERVERIP 192.168.1.80 /*改变原服务器 IP 地址*/
/*#define CONFIG_BOOTFILE "elinos-lart" */
#define CONFIG_BOOTCOMMAND "tftp 0x31000000 uImage;bootm 0x31000000"
.....
#undef CFG_CLKS_IN_HZ /* everything, incl board info, in Hz */

#define CFG_LOAD_ADDR 0x31000000 /* default load address bootm use
it*/
.....

/*-----

```

```

* FLASH and environment organization
*/
/*#define CONFIG_AMD_LV400      1      /* uncomment this if you have a LV400
flash */
/*#if 0
/*#define CONFIG_AMD_LV800      1      /* uncomment this if you have a LV800
flash */
/*#endif
#define CONFIG_AMD_LV800          /*mini2440上用的是2M 的 Nor Flash, 暂时用这
个也可以从 Nor Flash 启动 */
/*-----*/

```

(5)在 Nand Flash 中保存 u-boot 参数 (saveenv 功能):

```

/* timeout values are in ticks */
#define CFG_FLASH_ERASE_TOUT    (5*CFG_HZ) /* Timeout for Flash Erase */
#define CFG_FLASH_WRITE_TOUT    (5*CFG_HZ) /* Timeout for Flash Write */

#define CONFIG_ENV_IS_IN_NAND    1
#define CONFIG_ENV_OFFSET        0x30000
/*#define CFG_ENV_OFFSET 0X30000
/*#define CONFIG_ENV_IS_IN_FLASH 1
#define CONFIG_ENV_SIZE          0x10000 /* Total Size of Environment Sector */

```

(6)为了在 U-Boot 运行阶段对一些寄存器进行操作, 在此文件后面添加代码:

```

# if defined(CONFIG_mini2440_LED)
/* GPIO */
#define GPIO_CTL_BASE 0x56000000
#define oGPIO_B 0x10
#define oGPIO_CON 0x0 /* R/W, Configures the pins of the port */
#define oGPIO_DAT 0x4 /* R/W, Data register for port */
#define oGPIO_UP 0x8 /* R/W, Pull-up disable register */

#endif

#define STACK_BASE 0x33f00000
#define STACK_SIZE 0x8000

/* NAND Flash Controller */
#define NAND_CTL_BASE 0x4E000000
#define bINT_CTL(Nb) __REG(INT_CTL_BASE + (Nb))

```

```
/* Offset */
#define oNFCNF 0x00

# if defined(CONFIG_S3C2440)
#define CONFIG_S3C2440_NAND_BOOT 1
/* Offset */
#define oNFCNT 0x04
#define oNFCMD 0x08
#define oNFADDR 0x0c
#define oNFDATA 0x10
#define oNFSTAT 0x20
#define oNFECC 0x2c
#define rNFCNF (*(volatile unsigned int *)0x4e000000)
#define rNFCNT (*(volatile unsigned int *)0x4e000004)
#define rNFCMD (*(volatile unsigned char *)0x4e000008)
#define rNFADDR (*(volatile unsigned char *)0x4e00000c)
#define rNFDATA (*(volatile unsigned char *)0x4e000010)
#define rNFSTAT (*(volatile unsigned int *)0x4e000020)
#define rNFECC (*(volatile unsigned int *)0x4e00002c)
#endif

# if defined(CONFIG_S3C2410)
#define CONFIG_S3C2410_NAND_BOOT 1
/* Offset */
#define oNFCNF 0x00
#define oNFCMD 0x04
#define oNFADDR 0x08
#define oNFDATA 0x0c
#define oNFSTAT 0x10
#define oNFECC 0x14
#define rNFCNF (*(volatile unsigned int *)0x4e000000)
#define rNFCMD (*(volatile unsigned char *)0x4e000004)
#define rNFADDR (*(volatile unsigned char *)0x4e000008)
#define rNFDATA (*(volatile unsigned char *)0x4e00000c)
#define rNFSTAT (*(volatile unsigned int *)0x4e000010)
#define rNFECC (*(volatile unsigned int *)0x4e000014)
#define rNFECC0 (*(volatile unsigned char *)0x4e000014)
#define rNFECC1 (*(volatile unsigned char *)0x4e000015)
#define rNFECC2 (*(volatile unsigned char *)0x4e000016)
```

```
#endif
#endif /* __CONFIG_H */
```

1.2.13 修改 u-boot-2008.10/lib_arm/board.c 文件

(1) 添加头文件引用:

```
#include <nand.h>
#include <onenand_uboot.h>
#include <s3c2410.h>
```

(2) 添加 LED 功能, 指示进度:

```
static int display_banner (void)
{
    # if defined(CONFIG_mini2440_LED)
        S3C24X0_GPIO * const gpio = S3C24X0_GetBase_GPIO();
        gpio->GPBDAT = 0x100;

        //在串口初始化和 console 初始化完成, 串口输出信息之前, LED1、LED2、LED3会亮起!

    #endif

    #if !defined(CONFIG_mini2440_LED)
        S3C24X0_GPIO * const gpio = S3C24X0_GetBase_GPIO();

    #endif

    printf ("\n\n%s\n\n", version_string);

    .....

    reset_phy();
    #endif
    #endif

    #if defined(CONFIG_mini2440_LED)
        S3C24X0_GPIO * const gpio = S3C24X0_GetBase_GPIO();
    #endif
```

```
# if defined(CONFIG_mini2440_LED)
    gpio->GPBDAT = 0x0;

//在进入命令提示符之前, 四个 LED 会同时亮起!

#endif

/* main_loop() can return to retry autoboot, if so just run it again. */
```

1.2.14 搜索以下文件, 把支持 S3C2410 的宏定义改成同时支持 S3C2410 和 S3C2440

(1) 文件 u-boot-2008.10/cpu/arm920t/s3c24x0/i2c.c :

```
#include <s3c2400.h>
//#elif defined(CONFIG_S3C2410)
#elif defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
#include <s3c2410.h>

.....
S3C24X0_GPIO * const gpio = S3C24X0_GetBase_GPIO();
//#ifdef CONFIG_S3C2410
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
    return (gpio->GPEDAT & 0x8000) >> 15;
.....
S3C24X0_GPIO * const gpio = S3C24X0_GetBase_GPIO();
//#ifdef CONFIG_S3C2410
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
    gpio->GPEDAT = (gpio->GPEDAT & ~0x4000) | (x&1) << 14;
#endif
.....
if ((status & I2CSTAT_BSY) || GetI2CSDA () == 0) {
//#ifdef CONFIG_S3C2410
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
    ulong old_gpecon = gpio->GPECON;
#endif
.....
    /* bus still busy probably by (most) previously interrupted transfer
*/
//#ifdef CONFIG_S3C2410
```

```

#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
    /* set I2CSDA and I2CSCL (GPE15, GPE14) to GPIO */
    .....
    /* restore pin functions */
//#ifdef CONFIG_S3C2410
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
    gpio->GPECON = old_gpecon;
#endif

```

(2) 文件 u-boot-2008.10/cpu/arm920t/s3c24x0/serial.c :

```

#include <common.h>
//#if defined(CONFIG_S3C2400) || defined (CONFIG_S3C2410) || defined (CONFIG_TRAB)
#if defined(CONFIG_S3C2400) || defined (CONFIG_S3C2410) || defined (CONFIG_TRAB)
|| defined(CONFIG_S3C2440)

    #if defined(CONFIG_S3C2400) || defined(CONFIG_TRAB)
    #include <s3c2400.h>
//#elif defined(CONFIG_S3C2410)
#elif defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
#include <s3c2410.h>
#endif

```

(3) 以下三个文件的修改相同:

u-boot-2008.10/cpu/arm920t/s3c24x0/usb.c u-boot-
2008.10/cpu/arm920t/s3c24x0/usb_ohci.c u-boot-2008.10/drivers/rtc/s3c24x0_rtc.c

修改头文件的引用:

```

#if defined(CONFIG_S3C2400)
# include <s3c2400.h>
//#elif defined(CONFIG_S3C2410)
#elif defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
# include <s3c2410.h>
#endif

```

(4) 文件 u-boot-2008.10/include/common.h :

```

ulong get_OPB_freq (void);
ulong get_PCI_freq (void);
#endif
//#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410) || \

```



```

        //defined(CONFIG_LH7A40X) || defined(CONFIG_S3C6400)
        #if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410) || \
            defined(CONFIG_LH7A40X) || defined(CONFIG_S3C6400) ||
defined(CONFIG_S3C2440)
void    s3c2410_irq(void);
#define ARM920_IRQ_CALLBACK s3c2410_irq
ulong  get_FCLK (void);

```

(5) 文件 u-boot-2008.10/include/s3c24x0.h :

```

        S3C24X0_REG32  PRIORITY;
        S3C24X0_REG32  INTPND;
        S3C24X0_REG32  INTOFFSET;
//#ifdef CONFIG_S3C2410
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
        S3C24X0_REG32  SUBSRCPND;
        S3C24X0_REG32  INTSUBMSK;
#endif
.....
/* DMAS (see manual chapter 8) */
typedef struct {
        S3C24X0_REG32  DISRC;
//#ifdef CONFIG_S3C2410
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
        S3C24X0_REG32  DISRCC;
#endif
        S3C24X0_REG32  DIDST;
//#ifdef CONFIG_S3C2410
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
        S3C24X0_REG32  DIDSTC;
#endif
        S3C24X0_REG32  DCON;
.....
#ifdef CONFIG_S3C2400
        S3C24X0_REG32  res[1];
#endif
//#ifdef CONFIG_S3C2410
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
        S3C24X0_REG32  res[7];
#endif

```

```

    } /*__attribute__((__packed__))*/ S3C24X0_DMA;
.....
    S3C24X0_REG32  CLKCON;
    S3C24X0_REG32  CLKSLOW;
    S3C24X0_REG32  CLKDIVN;

    #if defined (CONFIG_S3C2440)
        S3C24X0_REG32  CAMDIVN;
    #endif
    } /*__attribute__((__packed__))*/ S3C24X0_CLOCK_POWER;
.....
    S3C24X0_REG32  res[8];
    S3C24X0_REG32  DITHMODE;
    S3C24X0_REG32  TPAL;
    // #ifdef CONFIG_S3C2410
    #if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
        S3C24X0_REG32  LCDINTPND;
        S3C24X0_REG32  LCDSRCPND;
        S3C24X0_REG32  LCDINTMSK;
.....
        S3C24X0_REG32  MISCCR;
        S3C24X0_REG32  EXTINT;
    #endif
    // #ifdef CONFIG_S3C2410
    #if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
        S3C24X0_REG32  GPACON;
        S3C24X0_REG32  GPADAT;
        S3C24X0_REG32  res1[2];

```

移植成功后, U-BOOT 在嵌入式系统中运行良好。同时支持 Nand Flash 和 Nor Flash 启动(仍需改进)。

附: U-Boot 各个版本的下载地址 <ftp://ftp.denx.de/pub/u-boot/>

参考博文:

<http://blog.csdn.net/hugerat/archive/2009/01/21/3847025.aspx>

<http://expowinzax.blog.163.com/blog/static/12408013200922222038567/>

<http://blog.chinaunix.net/u1/34474/showart.php?id=1808274>