# 最简单**bootloader**的实现与分析 **- 21ic**中国电子网

学习嵌入式，我是从**bootloader**入手的。前些日子写了一个bootloader，趁今天有时间发出来，以记录自己实现的过程，巩固所学到的知识，并且希望给需要帮助的人带来一些灵感，如果有不对的地方，还望大家能给予指正。

操作系统：Ubuntu 11.04 开发板：友善之臂mini2440 （如果用其它s3c2440或s3c2410 cpu的也差不多，大同小异）
串口调试终端：minicom 编译器：GNU工具链

先修知识：arm汇编，c语言，GNU汇编的一些特殊伪指令，makefile，链接脚本等知识。对于我的这个bootloader，这些知识除C语言外，其它的能看得懂，会一些基本的东西就足够了。

学习一门知识最好的方法莫过于实践，只有通过自己的亲身体会，才能对知识有更加深刻的理解，才能更好的运用。现在的bootloader已经很强大了，比如最出名的u-boot，支持多种cpu架构和不同的开发板。我们听到最多，学得最多的也是bootloader的移植，但是为什么要这样移植，就不见得所有人都知道了。我之所以要亲手实现这样的一个很简单的bootloader也就是为了能够更好的掌握bootloader的原理。先说下我的bootloader所实现的功能：目前只是最基本的功能，支持串口调试、支持命令的交互，但是具体的命令由于对掌握bootloader原理没太大帮助就没有实现，对于linux内核的引导过程相对复杂许多，在这个bootloader中也不作实现。我的想法是越简单越好，不想做得太复杂。

代码组织结构模仿了u-boot，如下：

- bootloader

- board 存放与开发板相关的目录

- s3c2440 存放s3c2440 cpu 的一些与寄存器相关的定义文件

- cpu 存放不同cpu架构的目录

- arm920t 存放依赖于arm920t的相关文件

- drivers 存放一些驱动文件

- include 存放一些用到的头文件

程序源代码：http://download.csdn.net/detail/tianfangk/3621598

有关开发环境的配置等一些知识网上一大堆，这里就不再赘述，直接从bootloader执行过程的角度开始分析。首先，要对一个程序进行分析，必然要先看它的入口函数。对于如何找到入口函数，就要看程序的链接脚本了。每一个链接过程都由链接脚本(linker script，一般以lds作为文件的后缀名)控制。链接脚本主要用于规定如何把输入文件内的section放入输出文件内，并控制输出文件内各部分在程序地址空间内的布局。如果在程序的链接过程中没有指定链接脚本，则会使用连接器的默认内置连接脚本。我用得是自己的链接脚本link.lds文件，从中可以看出程序的入口函数在cpu/arm920t/init.S文件中，所以先从这个文件开始分析。

```
_start:
/* Interrupt Vector Table */
b start @ 0x00
ldr pc, undefined @ 0x04
ldr pc, software_interrupt @ 0x08
ldr pc, prefetch_abort @ 0x0C
ldr pc, data_abort @ 0x10
ldr pc, not_used @ 0x14
ldr pc, irq @ 0x18
ldr pc, fiq @ 0x1C
```

这一段是中断向量表，arm规定从0x00地址开始到0x1C为中断向量表，当程序被中断后，就会自动跳到这个地方，执行相应的中断处理程序。s3c2440 cpu上电后要执行的第一条指令在0x00000000处，所以将执行第一条指令：b start

接着pc就跳到start处：

```
start:
bl svc32_mod
bl off_wtdog
bl off_int
bl init_clk
bl init_cpu
bl init_sdram
```

```
bl init_gpb
#ifdef CONFIG_DEBUG
bl set_uart
#endif
bl copy_code
bl jmp_ram
```

这里作一些硬件的初始化工作，设置cpu的工作模式为svc32、关闭看门狗、屏蔽所有中断、初始化时钟、关闭mmu、初始化内存控制寄存器、初始化与led灯相关的gpio、如果要用到打印调试的话，还要初始化串口。说了这么多感觉好像有点让人眼花缭乱，其实原则只有一点，那就是你要用到什么硬件，就把它初始化到你想要的状态。只要把握住这一点就会觉得做这一切都十分合理，十分清晰。具体怎么初始化，就要参看cpu的芯片手册了，上面说得很详细。下面对部分需要说明的地方进行说明：

接下来要做的事情就是将flash上的代码拷贝到内存中运行了。关于这一部分，有必要说明一点，这也是s3c2440这块芯片的特别之处。通常我们是将程序烧写在nor flash 上，因为nor flash有独立的地址线和数据线，可以直接寻址，所以程序可以直接在nor flash上运行。但是nand flash 不同， 它没有独立的地址线，因此不能直接寻址。所以s3c2440为了支持nand flash启动，在内部设有一块4K大小的SRAM，在S3C2440上电后， Nand Flash控制器会自动的把Nand Flash上的前4K数据搬移到内部SRAM中，并把这块SRAM映射到0x0地址处。由于我的这个bootloader总大小在4K之内，所以全部代码都可以直接被加载到内部SRAM中，为了简化过程，在copy_code过程中，我没有再进行对nand flash的操作，直接从SRAM也就是0x0地址处将代码copy到内存中，之后就执行jmp_ram这一过程，跳转到内存中运行。

但是这里有一个问题，也是我至今仍在困惑的问题，希望明白的朋友给解释一下。当代码从SRAM拷贝到内存完成的那一刻，存在了两处完全一样的代码，一处在SRAM中，一处在内存中。当cpu继续执行的时候，它是如何知道自己要从内存中去取那一条指令而不是SRAM？另外，代码复制到内存中，必然经过了一个重定向的工作，那么这一工作又是在何时完成的呢？

这一问题，先不管，接下来就要跳转到main函数中去了，这是一个C语言函数，必然会用到堆栈，所以在这之前要将堆栈指针设置好。C语言函数的可读性要强很多，就不用多说了。在main函数中，主要进行了对串口的初始化工作，最终程序将跳入到一个死循环wait_command中，反复重复一个动作：等待用户输入命令，然后执行命令。

这里还要说一下GPIO的问题，最初我在对串口进行初始化的时候，没有对相应的GPIO进行初始化（对于s3c2440，连接物理串口0的是GPH0~GPH7），导致无法将信息送到物理串口上，纠结了很长时间才查出错误。

总结一下：

1. 用到什么，就初始化什么；

2. 每写一行代码，都要保证其运行情况在你的掌控之下，千万不要写模棱两可的代码。

3. cpu的工作方式很简单：取指令，执行指令。不要让它猜你的意图。

做到这些，基本上可以保证程序不会出现大的错误。

到这里，对于这个bootloader的分析就完了，至于如何加载并启动内核，待以后有时间再续……