嵌入式 BootLoader 技术内幕

本文详细地介绍了基于嵌入式系统中的 OS 启动加载程序 ——Boot Loader 的概念、软件设计的主要任务以及结构框架等内容。

一、引言

在专用的嵌入式板子运行 GNU/Linux 系统已经变得越来越流行。一个嵌入式 Linux 系统从软件的角度看通常可以分为四个层次：

1. 引导加载程序。包括固化在固件(firmware)中的 boot 代码(可选)，和 Boot Loader 两大部分。
2. Linux 内核。特定于嵌入式板子的定制内核以及内核的启动参数。
3. 文件系统。包括根文件系统和建立于 Flash 内存设备之上文件系统。通常用 ram disk 来作为 root fs。
4. 用户应用程序。特定于用户的应用程序。有时在用户应用程序和内核层之间可能还会包括一个嵌入式图形用户界面。常用的嵌入式 GUI 有：MicroWindows 和 MiniGUI 懂。

引导加载程序是系统加电后运行的第一段软件代码。回忆一下 PC 的体系结构我们可以知道，PC 机中的引导加载程序由 BIOS(其本质就是一段固件程序)和位于硬盘 MBR 中的 OSBoot Loader（比如，LILO 和 GRUB 等）一起组成。BIOS 在完成硬件检测和资源分配后，将硬盘 MBR 中的 Boot Loader 读到系统的 RAM 中，然后将控制权交给 OS Boot Loader。Boot Loader 的主要运行任务就是将内核映象从硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，也即开始启动操作系统。

而在嵌入式系统中，通常并没有像 BIOS 那样的固件程序（注，有的嵌入式 CPU 也会内嵌一段短小的启动程序），因此整个系统的加载启动任务就完全由 Boot Loader 来完成。比如在一个基于 ARM7TDMI core 的嵌入式系统中，系统在上电或复位时通常都从地址 0x00000000 处开始执行，而在这个地址处安排的通常就是系统的 Boot Loader 程序。

本文将从 Boot Loader 的概念、Boot Loader 的主要任务、Boot Loader 的框架结构以及 Boot Loader 的安装等四个方面来讨论嵌入式系统的 Boot Loader。

二、 Boot Loader 的概念

简单地说，Boot Loader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

通常，Boot Loader 是严重地依赖于硬件而实现的，特别是在嵌入式世界。因此，在嵌入式世界里建立一个通用的 Boot Loader 几乎是不可能的。尽管如此，我们仍然可以对 Boot Loader 归纳出一些通用的概念来，以指导用户特定的 Boot Loader 设计与实现。

1. Boot Loader 所支持的 CPU 和嵌入式板

每种不同的 CPU 体系结构都有不同的 Boot Loader。有些 Boot Loader 也支持多种体系结构的 CPU，比如 U-Boot 就同时支持 ARM 体系结构和 MIPS 体系结构。除了依赖于 CPU 的体系结构外，Boot Loader 实际上也依赖于具体的嵌入式板级设备的配置。这也就是说，对于两块不同的嵌入式板而言，即使它们是基于同一种 CPU 而构建的，要想让运行在一块板子上的 Boot Loader 程序也能运行在另一块板子上，通常也都需要修改 Boot Loader 的源程序。

2. Boot Loader 的安装媒介（Installation Medium）

系统加电或复位后，所有的 CPU 通常都从某个由 CPU 制造商预先安排的地址上取指令。比如，基于 ARM7TDMI core 的 CPU 在复位时通常都从地址 0x00000000 取它的第一条指令。而基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备(比如：ROM、EEPROM 或 FLASH 等)被映射到这个预先安排的地址上。因此在系统加电后，CPU 将首先执行 Boot Loader 程序。

下图 1 就是一个同时装有 Boot Loader、内核的启动参数、内核映像和根文件系统映像的固态存储设备的典型空间分配结构图。

http://tech.ccidnet.com/pub/attachm...3/12/267984.gif

图 1 固态存储设备的典型空间分配结构

3. 用来控制 Boot Loader 的设备或机制

主机和目标机之间一般通过串口建立连接，Boot Loader 软件在执行时通常会通过串口来进行 I/O，比如：输出打印信息到串口，从串口读取用户控制字符等。

4. Boot Loader 的启动过程是单阶段（Single Stage）还是多阶段（Multi-Stage）通常多阶段的 Boot Loader 能提供更为复杂的功能，以及更好的可移植性。从固态存储设备上启动的 Boot Loader 大多都是 2 阶段的启动过程，也即启动过程可以分为 stage 1 和 stage 2 两部分。而至于在 stage 1 和 stage 2 具体完成哪些任务将在下面几篇讨论。

5. Boot Loader 的操作模式 (Operation Mode)

大多数 Boot Loader 都包含两种不同的操作模式："启动加载"模式和"下载"模式，这种区别仅对于开发人员才有意义。但从最终用户的角度看，Boot Loader 的作用就是用来加载操作系统，而并不存在所谓的启动加载模式与下载工作模式的区别。

启动加载（Boot loading）模式：这种模式也称为"自主"（Autonomous）模式。也即 Boot Loader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行，整个过程并没有用户的介入。这种模式是 Boot Loader 的正常工作模式，因此在嵌入式产品发布的时侯，Boot Loader 显然必须工作在这种模式下。

下载（Downloading）模式：在这种模式下，目标机上的 Boot Loader 将通过串口连接或网络连接等通信手段从主机（Host）下载文件，比如：下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 Boot Loader 保存到目标机的 RAM 中，然后再被 BootLoader 写到目标机上的 FLASH 类固态存储设备中。Boot Loader 的这种模式通常在第一次安装内核与根文件系统时被使用；此外，以后的系统更新也会使用 Boot Loader 的这种工作模式。工作于这种模式下的 Boot Loader 通常都会向它的终端用户提供一个简单的命令行接口。像 Blob 或 U-Boot 等这样功能强大的 Boot Loader 通常同时支持这两种工作模式，而且允许用户在这两种工作模式之间进行切换。比如，Blob 在启动时处于正常的启动加载模式,但是它会延时 10 秒等待终端用户按下任意键而将 blob 切换到下载模式。如果在 10 秒内没有用户按键，则 blob 继续启动 Linux 内核。

6. BootLoader 与主机之间进行文件传输所用的通信设备及协议最常见的情况就是，目标机上的 Boot Loader 通过串口与主机之间进行文件传输，传输协议通常是 xmodem / ymodem / zmodem 协议中的一种。但是，串口传输的速度是有限的，因此通过以太网连接并借助 TFTP 协议来下载文件是个更好的选择。

此外，在论及这个话题时，主机方所用的软件也要考虑。比如，在通过以太网连接和 TFTP 协议来下载文件时，主机方必须有一个软件用来的提供 TFTP 服务。在讨论了 BootLoader 的上述概念后，下面我们来具体看看 BootLoader 的应该完成哪些任务。

三、Boot Loader 的主要任务与典型结构框架

在继续本节的讨论之前，首先我们做一个假定，那就是：假定内核映像与根文件系统映像都被加载到 RAM 中运行。之所以提出这样一个假设前提是因为，在嵌入式系统中内核映像与根文件系统映像也可以直接在 ROM 或 Flash 这样的固态存储设备中直接运行。但这种做法无疑是以运行速度的牺牲为代价的。从操作系统的角度看，Boot Loader 的总目标就是正确地调用内核来执行。

另外，由于 Boot Loader 的实现依赖于 CPU 的体系结构，因此大多数 Boot Loader 都分为 stage1 和 stage2 两大部分。依赖于 CPU 体系结构的代码，比如设备初始化代码等，通常都放在 stage1 中，而且通常都用汇编语言来实现，以达到短小精悍的目的。而 stage2 则通常用 C 语言来实现，这样可以实现给复杂的功能，而且代码会具有更好的可读性和可移植性。

Boot Loader 的 stage1 通常包括以下步骤(以执行的先后顺序)：

- 硬件设备初始化。
- 为加载 Boot Loader 的 stage2 准备 RAM 空间。
- 拷贝 Boot Loader 的 stage2 到 RAM 空间中。
- 设置好堆栈。
- 跳转到 stage2 的 C 入口点。

Boot Loader 的 stage2 通常包括以下步骤(以执行的先后顺序)：

• 初始化本阶段要使用到的硬件设备。
• 检测系统内存映射(memory map)。
• 将 kernel 映像和根文件系统映像从 flash 上读到 RAM 空间中。
• 为内核设置启动参数。
• 调用内核。

3.1 Boot Loader 的 stage1

3.1.1 基本的硬件初始化

是 Boot Loader 一开始就执行的操作，其目的是为 stage2 的执行以及随后的 kernel 的执行准备好一些基本的硬件环境。它通常包括以下步骤（以执行的先后顺序）：

1. 屏蔽所有的中断。为中断提供服务通常是 OS 设备驱动程序的责任，因此在 Boot Loader 的执行全过程中可以不必响应任何中断。中断屏蔽可以通过写 CPU 的中断屏蔽寄存器或状态寄存器（比如 ARM 的 CPSR 寄存器）来完成。

2. 设置 CPU 的速度和时钟频率。

3. RAM 初始化。包括正确地设置系统的内存控制器的功能寄存器以及各内存库控制寄存器等。

4. 初始化 LED。典型地，通过 GPIO 来驱动 LED，其目的是表明系统的状态是 OK 还是 Error。如果板子上没有 LED，那么也可以通过初始化 UART 向串口打印 Boot Loader 的 Logo 字符信息来完成这一点。

5. 关闭 CPU 内部指令 / 数据 cache。

3.1.2 为加载 stage2 准备 RAM 空间

为了获得更快的执行速度，通常把 stage2 加载到 RAM 空间中来执行，因此必须为加载 Boot Loader 的 stage2 准备好一段可用的 RAM 空间范围。由于 stage2 通常是 C 语言执行代码，因此在考虑空间大小时，除了 stage2 可执行映象的大小外，还必须把堆栈空间也考虑进来。此外，空间大小最好是 memory page 大小(通常是 4KB)的倍数。一般而言，1M 的 RAM 空间已经足够了。具体的地址范围可以任意安排,比如 blob 就将它的 stage2 可执行映像安排到从系统 RAM 起始地址 0xc0200000 开始的 1M 空间内执行。但是，将 stage2 安排到整个 RAM 空间的最顶 1MB(也即(RamEnd-1MB) - RamEnd)是一种值得推荐的方法。

为了后面的叙述方便，这里把所安排的 RAM 空间范围的大小记为：stage2_size(字节)，把起始地址和终止地址分别记为：stage2_start 和 stage2_end(这两个地址均以 4 字节边界对齐)。因此：stage2_end＝stage2_start＋stage2_size

另外，还必须确保所安排的地址范围的的确确是可读写的 RAM 空间，因此，必须对你所安排的地址范围进行测试。具体的测试方法可以采用类似于 blob 的方法，也即：以 memory page 为被测试单位，测试每个 memory page 开始的两个字是否是可读写的。为了后面叙述的方便，我们记这个检测算法为：test_mempage，其具体步骤如下：

1. 先保存 memory page 一开始两个字的内容。

2. 向这两个字中写入任意的数字。比如：向第一个字写入 0x55，第 2 个字写入 0xaa。

3. 然后，立即将这两个字的内容读回。显然，我们读到的内容应该分别是 0x55 和 0xaa。如果不是，则说明这个 memory page 所占据的地址范围不是一段有效的 RAM 空间。

4. 再向这两个字中写入任意的数字。比如：向第一个字写入 0xaa，第 2 个字中写入 0x55。

5. 然后，立即将这两个字的内容立即读回。显然，我们读到的内容应该分别是 0xaa 和 0x55。如果不是，则说明这个 memory page 所占据的地址范围不是一段有效的 RAM 空间。

6. 恢复这两个字的原始内容。测试完毕。

为了得到一段干净的 RAM 空间范围，我们也可以将所安排的 RAM 空间范围进行清零操作。

3.1.3 拷贝 stage2 到 RAM 中

拷贝时要确定两点：

(1) stage2 的可执行映象在固态存储设备的存放起始地址和终止地址；

(2) RAM 空间的起始地址。

3.1.4 设置堆栈指针 sp

堆栈指针的设置是为了执行 C 语言代码作好准备。通常我们可以把 sp 的值设置为(stage2_end-4)，也即在 3.1.2 节所安排的那个 1MB 的 RAM 空间的最顶端(堆栈向下生长)。此外，在设置堆栈指针 sp 之前，也可以关闭 led 灯，以提示用户我们准备跳转到 stage2。经过上述这些执行步骤后，系统的物理内存布局应该如下图 2 所示。

3.1.5 跳转到 stage2 的 C 入口点

在上述一切都就绪后，就可以跳转到 Boot Loader 的 stage2 去执行了。比如，在 ARM系统中，这可以通过修改 PC 寄存器为合适的地址来实现。

http://tech.ccidnet.com/pub/attachm...3/12/268047.gif

图 2 bootloader 的 stage2 可执行映象刚被拷贝到 RAM 空间时的系统内存布局

3.2 Boot Loader 的 stage2

正如前面所说，stage2 的代码通常用 C 语言来实现，以便于实现更复杂的功能和取得更好的代码可读性和可移植性。但是与普通 C 语言应用程序不同的是，在编译和链接 bootloader 这样的程序时，我们不能使用 glibc 库中的任何支持函数。其原因是显而易见的。这就给我们带来一个问题，那就是从那里跳转进 main() 函数呢？直接把 main() 函数的起始地址作为整个 stage2 执行映像的入口点或许是最直接的想法。但是这样做有两个缺点：1)无法通过 main() 函数传递函数参数；2)无法处理 main() 函数返回的情况。一种更为巧妙的方法是利用 trampoline(弹簧床)的概念。也即，用汇编语言写一段 trampoline 小程序，并将这段 trampoline 小程序来作为 stage2 可执行映象的执行入口点。然后我们可以在 trampoline 汇编小程序中用 CPU 跳转指令跳入 main() 函数中去执行；而当 main() 函数返回时，CPU 执行路径显然再次回到我们的 trampoline 程序。简而言之，这种方法的思想就是：用这段 trampoline 小程序来作为 main() 函数的外部包裹(external wrapper)。

下面给出一个简单的 trampoline 程序示例(来自 blob)：

```
.text
.globl _trampoline
trampoline:
bl main
/* if main ever returns we just call it again */
b _trampoline
```

可以看出，当 main() 函数返回后，我们又用一条跳转指令重新执行 trampoline 程序——当然也就重新执行 main() 函数，这也就是 trampoline(弹簧床)一词的意思所在。

3.2.1 初始化本阶段要使用到的硬件设备

这通常包括：

（1）初始化至少一个串口，以便和终端用户进行 I/O 输出信息；

（2）初始化计时器等。在初始化这些设备之前，也可以重新把 LED 灯点亮，以表明我们已经进入 main() 函数执行。设备初始化完成后，可以输出一些打印信息，程序名字字符串、版本号等。

3.2.2 检测系统的内存映射（memory map）

所谓内存映射就是指在整个 4GB 物理地址空间中有哪些地址范围被分配用来寻址系统的 RAM 单元。比如，在 SA-1100 CPU 中，从 0xC000,0000 开始的 512M 地址空间被用作系统的 RAM 地址空间，而在 Samsung S3C44B0X CPU 中，从 0x0c00,0000 到 0x1000,0000 之间的 64M 地址空间被用作系统的 RAM 地址空间。虽然 CPU 通常预留出一大段足够的地址空间给系统 RAM，但是在搭建具体的嵌入式系统时却不一定会实现 CPU 预留的全部 RAM 地址空间。也就是说，具体的嵌入式系统往往只把 CPU 预留的全部 RAM 地址空间中的一部分映射到 RAM 单元上，而让剩下的那部分预留 RAM 地址空间处于未使用状态。由于上述这个事实，因此 Boot Loader 的 stage2 必须在它想干点什么（比如，将存储在

flash 上的内核映像读到 RAM 空间中) 之前检测整个系统的内存映射情况,也即它必须知道 CPU 预留的全部 RAM 地址空间中的哪些被真正映射到 RAM 地址单元，哪些是处于 “unused” 状态的。

(1) 内存映射的描述

可以用如下数据结构来描述 RAM 地址空间中的一段连续(continuous)的地址范围：

```
typedef struct memory_area_struct {
    u32 start;     /* the base address of the memory region */
    u32 size;      /* the byte number of the memory region */
    int used;
} memory_area_t;
```

这段 RAM 地址空间中的连续地址范围可以处于两种状态之一：(1)used=1，则说明这段连续的地址范围已被实现，也即真正地被映射到 RAM 单元上。(2)used=0，则说明这段连续的地址范围并未被系统所实现，而是处于未使用状态。

基于上述 memory_area_t 数据结构，整个 CPU 预留的 RAM 地址空间可以用一个 memory_area_t 类型的数组来表示，如下所示：

```
memory_area_t memory_map[NUM_MEM_AREAS] = {
    [0 ... (NUM_MEM_AREAS - 1)] = {
        .start = 0,
        .size = 0,
        .used = 0
    },
};
```

(2) 内存映射的检测

下面我们给出一个可用来检测整个 RAM 地址空间内存映射情况的简单而有效的算法：

```
/* 数组初始化 */
for(i = 0; i < NUM_MEM_AREAS; i++)
    memory_map[i].used = 0;
/* first write a 0 to all memory locations */
for(addr = MEM_START; addr < MEM_END; addr += PAGE_SIZE)
    * (u32 *)addr = 0;
for(i = 0, addr = MEM_START; addr < MEM_END; addr += PAGE_SIZE) {
/*
* 检测从基地址 MEM_START+i*PAGE_SIZE 开始,大小为
* PAGE_SIZE 的地址空间是否是有效的 RAM 地址空间。
*/
调用 3.1.2 节中的算法 test_mempage();
    if ( current memory page isnot a valid ram page) {
    /* no RAM here */
        if(memory_map[i].used )
            i++;
        continue;
}
    /*
     当前页已经是一个被映射到 RAM 的有效地址范围
    * 但是还要看看当前页是否只是 4GB 地址空间中某个地址页的别名？
    */
    if(* (u32 *)addr != 0) { /* alias? */
    /* 这个内存页是 4GB 地址空间中某个地址页的别名 */
```

```
            if ( memory_map[i].used )
                i++;
            continue;
        }
        /*
        * 当前页已经是一个被映射到 RAM 的有效地址范围
        * 而且它也不是 4GB 地址空间中某个地址页的别名。
        */
        if (memory_map[i].used == 0) {
            memory_map[i].start = addr;
            memory_map[i].size = PAGE_SIZE;
            memory_map[i].used = 1;
        }
        else {
            memory_map[i].size += PAGE_SIZE;
        }
} /* end of for (…) */
```

在用上述算法检测完系统的内存映射情况后，Boot Loader 也可以将内存映射的详细信息打印到串口。

3.2.3 加载内核映像和根文件系统映像

(1) 规划内存占用的布局

这里包括两个方面：(1)内核映像所占用的内存范围；(2) 根文件系统所占用的内存范围。在规划内存占用的布局时，主要考虑基地址和映像的大小两个方面。对于内核映像，一般将其拷贝到从(MEM_START＋0x8000) 这个基地址开始的大约 1MB 大小的内存范围内(嵌入式 Linux 的内核一般都不操过 1MB)。为什么要把从 MEM_START 到 MEM_START＋0x8000 这段 32KB 大小的内存空出来呢？这是因为 Linux 内核要在这段内存中放置一些全局数据结构，如：启动参数和内核页表等信息。而对于根文件系统映像，则一般将其拷贝到 MEM_START+0x0010,0000 开始的地方。如果用 Ramdisk 作为根文件系统映像，则其解压后的大小一般是 1MB。

(2) 从 Flash 上拷贝

由于像 ARM 这样的嵌入式 CPU 通常都是在统一的内存地址空间中寻址 Flash 等固态存储设备的，因此从 Flash 上读取数据与从 RAM 单元中读取数据并没有什么不同。用一个简单的循环就可以完成从 Flash 设备上拷贝映像的工作：

```
while(count) {
    *dest++ = *src++; /* they are all aligned with word boundary */
    count -= 4; /* byte number */
};
```

3.2.4 设置内核的启动参数

应该说，在将内核映像和根文件系统映像拷贝到 RAM 空间中后，就可以准备启动 Linux 内核了。但是在调用内核之前，应该作一步准备工作，即：设置 Linux 内核的启动参数。

Linux 2.4.x 以后的内核都期望以标记列表(tagged list)的形式来传递启动参数。启动参数标记列表以标记 ATAG_CORE 开始，以标记 ATAG_NONE 结束。每个标记由标识被传递参数的 tag_header 结构以及随后的参数值数据结构来组成。数据结构 tag 和 tag_header 定义在 Linux 内核源码的 include/asm/setup.h 头文件中：

```
/* The list ends with an ATAG_NONE node. */
#define ATAG_NONE 0x00000000
struct tag_header {
```

```
    u32 size; /* 注意，这里 size 是字数为单位的 */
    u32 tag;
};
……
struct tag {
    struct tag_header hdr;
    union {
        struct tag_core core;
        struct tag_mem32 mem;
        struct tag_videotext videotext;
        struct tag_ramdisk ramdisk;
        struct tag_initrd initrd;
        struct tag_serialnr serialnr;
        struct tag_revision revision;
        struct tag_videolfb videolfb;
        struct tag_cmdline cmdline;
        /*
        * Acorn specific
        */
        struct tag_acorn acorn;
        /*
        * DC21285 specific
        */
        struct tag_memclk memclk;
    } u;
};
```

在嵌入式 Linux 系统中，通常需要由 Boot Loader 设置的常见启动参数有：ATAG_CORE、ATAG_MEM、ATAG_CMDLINE、ATAG_RAMDISK、ATAG_INITRD 等。比如，设置 ATAG_CORE 的代码如下：

```
params = (struct tag *)BOOT_PARAMS;
params->hdr.tag = ATAG_CORE;
params->hdr.size = tag_size(tag_core);
params->u.core.flags = 0;
params->u.core.pagesize = 0;
params->u.core.rootdev = 0;
params = tag_next(params);
```

其中，BOOT_PARAMS 表示内核启动参数在内存中的起始基地址，指针 params 是一个 struct tag 类型的指针。宏 tag_next() 将以指向当前标记的指针为参数，计算紧临当前标记的下一个标记的起始地址。注意，内核的根文件系统所在的设备 ID 就是在这里设置的。

下面是设置内存映射情况的示例代码：

```
for(i = 0; i < NUM_MEM_AREAS; i++) {
    if(memory_map[i].used) {
        params->hdr.tag = ATAG_MEM;
        params->hdr.size = tag_size(tag_mem32);
        params->u.mem.start = memory_map[i].start;
        params->u.mem.size = memory_map[i].size;
        params = tag_next(params);
    }
```

```
}
```

可以看出，在 memory_map [] 数组中，每一个有效的内存段都对应一个 ATAG_MEM 参数标记。

Linux 内核在启动时可以以命令行参数的形式来接收信息，利用这一点我们可以向内核提供那些内核不能自己检测的硬件参数信息，或者重载(override)内核自己检测到的信息。

比如，我们用这样一个命令行参数字符串"console=ttyS0,115200n8"来通知内核以 ttyS0 作为控制台，且串口采用 "115200bps、无奇偶校验、8 位数据位"这样的设置。下面是一段设置调用内核命令行参数字符串的示例代码：

```
char *p;
/* eat leading white space */
for(p = commandline; *p == ' '; p++)
    ;
/* skip non-existent command lines so the kernel will still
* use its default command line.
*/
if(*p == '\0')
    return;
params->hdr.tag = ATAG_CMDLINE;
params->hdr.size = (sizeof(struct tag_header) + strlen(p) + 1 + 4) >> 2;
strcpy(params->u.cmdline.cmdline, p);
params = tag_next(params);
```

请注意在上述代码中，设置 tag_header 的大小时，必须包括字符串的终止符'\0'，此外还要将字节数向上圆整 4 个字节，因为 tag_header 结构中的 size 成员表示的是字数。

下面是设置 ATAG_INITRD 的示例代码，它告诉内核在 RAM 中的什么地方可以找到 initrd 映象(压缩格式)以及它的大小：

```
params->hdr.tag = ATAG_INITRD2;
params->hdr.size = tag_size(tag_initrd);
params->u.initrd.start = RAMDISK_RAM_BASE;
params->u.initrd.size = INITRD_LEN;
params = tag_next(params);
```

下面是设置 ATAG_RAMDISK 的示例代码，它告诉内核解压后的 Ramdisk 有多大（单位是 KB）：

```
params->hdr.tag = ATAG_RAMDISK;
params->hdr.size = tag_size(tag_ramdisk);
params->u.ramdisk.start = 0;
params->u.ramdisk.size = RAMDISK_SIZE; /* 请注意，单位是 KB */
params->u.ramdisk.flags = 1; /* automatically load ramdisk */
params = tag_next(params);
```

最后，设置 ATAG_NONE 标记，结束整个启动参数列表：

```
static void setup_end_tag(void)
{
    params->hdr.tag = ATAG_NONE;
    params->hdr.size = 0;
}
```

3.2.5 调用内核

Boot Loader 调用 Linux 内核的方法是直接跳转到内核的第一条指令处，也即直接跳转到 MEM_START＋0x8000 地址处。在跳转时，下列条件要满足：

1. CPU 寄存器的设置：

• R0＝0；

@R1＝机器类型 ID；关于 Machine Type Number，可以参见 linux/arch/arm/tools/mach-types。

@R2＝启动参数标记列表在 RAM 中起始基地址；

2. CPU 模式：

• 必须禁止中断（IRQs 和 FIQs）；

• CPU 必须 SVC 模式；

3. Cache 和 MMU 的设置：

• MMU 必须关闭；

• 指令 Cache 可以打开也可以关闭；

• 数据 Cache 必须关闭；

如果用 C 语言，可以像下列示例代码这样来调用内核：

```
void (*theKernel)(int zero, int arch, u32 params_addr) = (void (*)(int, int, u32))KERNEL_RAM_BASE;
……
theKernel(0, ARCH_NUMBER, (u32) kernel_params_start);
```

注意，theKernel()函数调用应该永远不返回的。如果这个调用返回，则说明出错。

四、 关于串口终端

在 boot loader 程序的设计与实现中，没有什么能够比从串口终端正确地收到打印信息能更令人激动了。此外，向串口终端打印信息也是一个非常重要而又有效的调试手段。但是，我们经常会碰到串口终端显示乱码或根本没有显示的问题。造成这个问题主要有两种原因：(1) boot loader 对串口的初始化设置不正确。(2) 运行在 host 端的终端仿真程序对串口的设置不正确，这包括：波特率、奇偶校验、数据位和停止位等方面的设置。

此外，有时也会碰到这样的问题，那就是：在 boot loader 的运行过程中我们可以正确地向串口终端输出信息，但当 boot loader 启动内核后却无法看到内核的启动输出信息。对这一问题的原因可以从以下几个方面来考虑：

(1) 首先请确认你的内核在编译时配置了对串口终端的支持，并配置了正确的串口驱动程序。

(2) 你的 boot loader 对串口的初始化设置可能会和内核对串口的初始化设置不一致。此外，对于诸如 s3c44b0x 这样的 CPU，CPU 时钟频率的设置也会影响串口，因此如果 boot loader 和内核对其 CPU 时钟频率的设置不一致，也会使串口终端无法正确显示信息。

(3) 最后，还要确认 boot loader 所用的内核基地址必须和内核映像在编译时所用的运行基地址一致，尤其是对于 uClinux 而言。假设你的内核映像在编译时用的基地址是 0xc0008000，但你的 boot loader 却将它加载到 0xc0010000 处去执行，那么内核映像当然不能正确地执行了。

五、 结束语

Boot Loader 的设计与实现是一个非常复杂的过程。如果不能从串口收到那激动人心的“uncompressing linux ..... done,booting the kernel……”。

Linux bootloader 编写方法

对于移植 linux 到其它开发板的人来说，编写 boot loader 是一个不可避免的过程。对于学习 linux 的人来讲，编写 bootloader 也是一个很有挑战性的工作。本文通过对 linux 引导协议进行分析，详细阐述了如何编写一个可以在 i386 机器上引导 2.4.20 内核的基本的 bootloader。

1.概述

linux 运行在保护模式下，但是当机器启动复位的时候却处于实模式下。所以写 bootloader 做的

工作也是在实模式之下的。

linux 的内核有多种格式，老式的 zImage 和新型的 bzImage。它们之间最大的差别是对于内核体积大小的限制。由于 zImage 内核需要放在实模式 1MB 的内存之内，所以其体积受到了限制。目前采用的内核格式大多为 bzImage，这种格式没有 1MB 内存限制。本文以下部分主要以 bzImage 为例进行分析。

2.bzImage 格式内核的结构

bzImage 内核从前向后分为 3 个部分，前 512 字节被称为 bootsect，这就是软盘引导 linux 时用到的 bootloader，如果不从软盘引导，这部分就没有用，其中存储了一些编译时生成的内核启动选项的默认值。从 512 个字节开始的 512*n 个字节称为 setup 部分，这是 linux 内核的实模式部分，这部分在实模式下运行,主要功能是为保护模式的 linux 内核启动准备环境。这个部分最后会切换进入保护模式，跳转到保护模式的内核执行。最后的部分就是保护模式的内核，也就是真正意义上的 linux 内核。其中 n 的大小可以从 bootsect 后半部得到，详细地址可以参阅 linux boot protocol。

3.引导过程概述

第一步，打开冰箱门；第二步把大象放到冰箱里……不要笑，过程就是这么简单。首先需要把 linux 内核的 setup 部分拷贝到 9020H:0 开始的地址，然后把保护模式内核拷贝到 1MB 开始的地址，然后根据 Linux Boot Protocol 2.03 的内容设定参数区的内容，基地址就是 9000H：0，最后使用一条 ljmp $0x9020,$0 跳转到 setup 段，剩下的事情就是 linux 自己的了^_^,果然简单吧！

4.THE LINUX/I386 BOOT PROTOCOL

这个就是我们引导 linux 所使用的协议，它的位置在：Documetation/i386/boot.txt 中。里面详细的写了引导 linux 所需要知道的一切知识，对于其它体系结构的 CPU，也一定存在着类似的东东，仿照本文的方法就可以了。

5.细节一：基本引导参数

当然我们不指定任何参数 linux 内核也可以启动，但是这样有可能启动进入一个我们不支持的 framebuffer 模式，导致没有任何屏幕显示；也可能 mount 了错误的根分区失败，导致 No Init Found 的 kernel panic。所以我们必须要指定一些东西。

如果你像我一样是一个懒人，那么可以直接把 bootsect 拷到 9000H：0 的位置，使用软盘引导时它会把自己复制到这个地方的，这里面有些默认的设置，详情请见 boot.txt。

首先是 root 的位置，这里 bootsect_pos 指向的是 9000H:0 的地址。

bootsect_pos[0x1fc] = root_minor;

bootsect_pos[0x1fd] = root_major;

其中 root_minor 和 root_major 分别是 root 的主设备号和次设备号。

当前显示模式：

bootsect_pos[0x1fa] = 0xff;

bootsect_pos[0x1fb] = 0xff;

这两个数值相当于引导参数 vga=0xHHH 的值，两个 0xff 代表文本模式。

bootsect_pos[0x210] = 0xff;

这是在设定你的 bootloader 的类型，其实只要不是 0 就行，因为 0 代表的 loader 太旧无法引导新的内核，setup 发现这个后就会停下来。按照规范你应该写成 0xff，这表示未知的 boot loader，如果你的 bootloader 已经得到了一个官方分配的 type id，那就写上自己的数值。

6.细节二：如何加载内核

如果你现在的环境是一无所有，那么必须使用 bios 中断或者 ATA 指令去读硬盘了，不过如果你手中如果有基本的 DOS 系统，那么就可以使用 DOS 的程序了。为了能够操作整个 4GB 的地址空间，我使用了 WATCOM C 写了个小程序读内核，不过你可以仿照 bootsect 里面的做法，在实模式中读一部分，然后进入到保护模式拷贝到 1MB 以上,然后再从实模式读一部分……需要注意 1:9000H:0 也是 DOS 占用的地

址空间，所以读完内核后就不要返回 DOS 了，否则会有问题；

注意 2:一定保证是纯 DOS，不要加载 HIMEM 或者 EMM386 这样的东西，它们会使上面的引导过程失败。loadlin 倒是可以来者通吃几乎所有的 DOS，不过它的作者也是这方面的大牛，对 DOS 下的内存管理非常的熟悉。我们现在研究这些古老的东西很难找资料了，况且我们是在写 bootloader，不是 DOS killer^_^。

7.引导时的高级功能

1)initrd

initrd 是启动时的一个小虚拟盘，一般用它来实现模块化的内核。引导 initrd 的方法主要有两个要点：

第一，把 initrd 读入内存，我们可以仿照大多数 boot loader 的方法把它放在内存的最高端；

第二，设定 initrd 的起始位置和长度 bootsect_pos[0x218]开始的 4 个字节放的是起始物理地址，bootsect_pos[0x21c]开始的 4 个字节放的是 initrd 的长度。

2)command_line 支持

用 command_line 你可以给内核传一些参数，自己定制内核的行为。我是这样做的，首先把 command_line 放在 9900H:0 的地址里，然后把 9900H:0 的物理地址存放在 bootsect_pos[0x228]开始的 4 个字节里面。注意一定是物理地址，所以你应该放 99000H 这个数，然后内核就会识别你的 command_line 了。

8.结束语

写本文的目的主要是为了用最少的语言和最短的时间说明bootloader的原理，真正的权威资料还是要看linux内核源码和boot.txt文件。我曾经写过一个例子loaderx，使用WATCOM C和TASM，WATCOM C是一个可以在DOS下生成能访问 4GB物理地址程序的C编译器，里面也有详细的注释和文档说明。可以从下面的地址下载： loaderx.tar.gz

参考资料

THE LINUX/I386 BOOT PROTOCOL 2.03

关于作者

范晓炬，联想(北京)有限公司软件设计中心嵌入式研发处开发工程师，研究兴趣为 Linux 内核，网络安全，XWindow 系统，Linux 桌面应用，人工智能系统。你可以通过 xiaoju_f@263.net联系他。

U-BOOT 的启动流程及移植

摘要：嵌入式系统一般没有通用的 bootloader,u-boot 是功能强大的 bootloader 开发软件，但相对也比较复杂。文中对 u-boot 的启动流程作了介绍，详细给出了 u-boot 在 S3C44B0 开发板上的移植方法和步骤。(MIPS 也是个缩写字，全称是 Microprocessor without Interlocked Pipeline Stages)

关键词：bootloader；u-boot；嵌入式系统；移植；S3C44B0

1 Bootloader 及 u-boot 简介

BootLoader 代码是芯片复位后进入操作系统之前执行的一段代码，主要用于完成由硬件启动到操作系统启动的过渡，从而为操作系统提供基本的运行环境，如初始化 CPU、 堆栈、存储器系统等。BootLoader 代码与 CPU 芯片的内核结构、具体型号、应用系统的配置及使用的操作系统等因素有关，其功能类似于 PC 机的 BIOS 程序。由于 BootLoader 和 CPU 及电路板的配置情况有关，因此不可能有通用的 BootLoader ，开发时需要用户根据具体情况进行移植。嵌入式 Linux 系统中常用的 BootLoader 有 armboot、redboot、blob、u-boot 等，其中 u-boot 是当前比较流行，功能比较强大的 BootLoader，可以支持多种体系结构，但相对也比较复杂。BootLoader 的实现依赖于 CPU 的体系结构，大多数 BootLoader 都分为 stage1 和 stage2 两大部分。BootLoader 的基本原理见参考文献。

U-boot 是 sourceforge 网站上的一个开放源代码的项目。它可对 PowerPc、MPC5xx、MPC8xx、MPC82xx、 MPC7xx、MPC74xx、ARM（ARM7、ARM9、Strong ARM、Xscale）、MIPS（4kc、5kc）、X86 等处理器提供支持，支持的嵌入式操作系统有 Linux、Vx-Works、NetBSD、QNX、RTEMS、ARTOS、LynxOS 等，主要用来开发嵌入式系统初始化代码 bootloader。软件的主站点是 http://sourceforge.net/projects/u-boot。u-boot 最初是由 denx www.denx.de 的 PPC-boot 发展而来的，它对 PowerPC 系列处理器的支持最完善，对 Linux 操作系统的支持最好。源代码开放的 u-boot 软件项目经常更新，是学习硬件底层代码开发的很好样例。

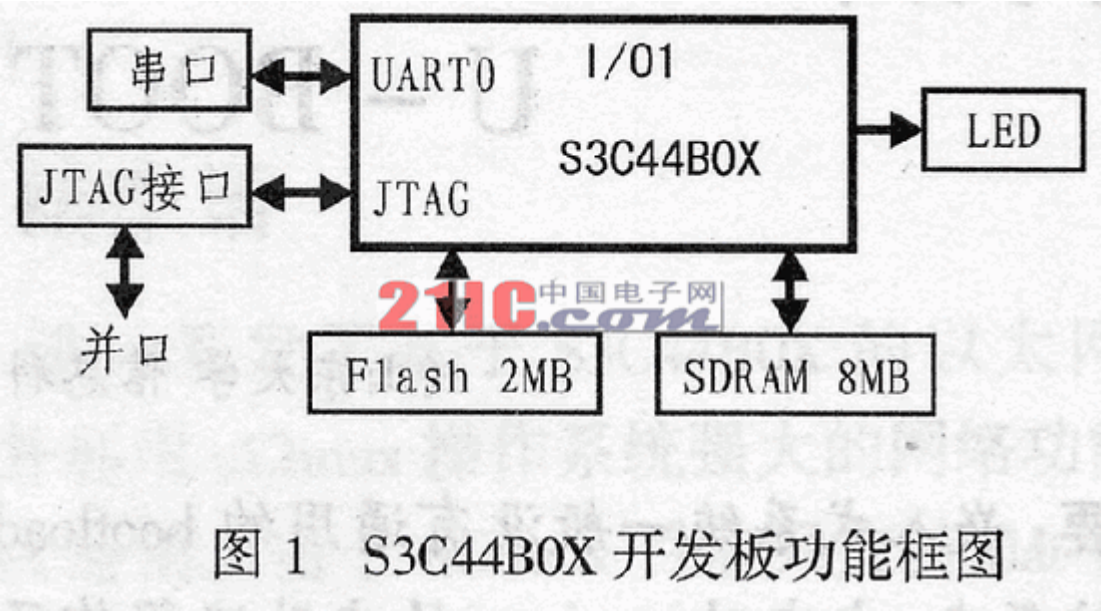

图 1 S3C44B0X 开发板功能框图

2 u-boot 系统启动流程

大多数 bootloader 都分为 stage1 和 stage2 两大部分，u-boot 也不例外。依赖于 CPU 体系结构的代码（如设备初始化代码等）通常都放在 stage1 且可以用汇编语言来实现，而 stage2 则通常用 C 语言来实现，这样可以实现复杂的功能，而且有更好的可读性和移植性。

2．1 stage1 (stage.s 代码结构)

u-boot 的 stage1 代码通常放在 stage.s 文件中，它用汇编语言写成，其主要代码部分如下：

（1） 定义入口 。由于一个可执行的 Image 必须有一个入口点，并且只能有一个全局入口，通常这个入口放在 ROM(Flash)的 0x0 地址，因此，必须通知编译器以使其知道这个入口，该工作可通过修改连接器脚本来完成。

（2）设置异常向量(Exception Vector)。

（3）设置 CPU 的速度、时钟频率及中断控制寄存器。

（4）初始化内存控制器 。

（5）将 ROM 中的程序复制到 RAM 中。

（6）初始化堆栈 。

（7）转到 RAM 中执行，该工作可使用指令 ldrpc 来完成。

2．2 stage2 C 语言代码部分

Lib arm/board.c 中的 start armboot 是 C 语言开始的函数，也是整个启动代码中 C 语言的主函数，同时还是整个 u-boot（armboot）的主函数，该函数主要完成如下操作：

（1）调用一系列的初始化函数。

（2）初始化 Flash 设备。

（3）初始化系统内存分配函数。

（4）如果目标系统拥有 NAND 设备，则初始化 NAND 设备。

（5）如果目标系统有显示设备，则初始化该类设备。

（6）初始化相关网络设备，填写 IP、MAC 地址等。

（7）进入命令循环（即整个 boot 的工作循环），接受用户从串口输入的命令，然后进行相应的工作。

3 移植实例

本系统开发板主要由 S3C44B0X 嵌入式微处理器、2MB 的 Flash（SST39VF160）、8MB 的 SDRAM (HY57V641620)、4 个 LED 以及 ARM JTAG 接口组成。该开发板上与 S3C44B0X 相关部分的功能框图如图 1 所示。

３． 1 u-boot 文件下载

u-boot 文件的下载有两种方法,第一种是在Ｌ ｉ ｎ ｕ ｘ环境下通过ＣＶＳ下载最新的文件,方法是：

$cvs-d pserver anonymous@cvs.sourceforge.net/cvsroot/u-boot login

当要求输入匿名登录的密码时，可直接按回车键

$cvs-z6-d pserver anonymous@cvs.source forge.net/cvsroot/u-boot\co.Pmodulename

第二种是通过 ftp://ftp.denx.de/pub/u-boot/下载正式发布的压缩文件。

３． 2 u-boot 文件的结构

初次下载的文件有很多，解压后存放在 u-boot 文件目录下，具体内容已在 readme 文件中做了详细的介绍，其中与移植相关的主要文件夹有：

（１）CPU 它的每个子文件夹里都有如下文件：

makefile

config.mk

cpu.c 和处理器相关的代码

interrupts.c 中断处理代码

serial.c 串口初始化代码

start.s 全局开始启动代码

（２）BOARD 它的每个子文件夹里都有如下文件：

makefile

config.mk

smdk2410.c 和板子相关的代码（以 smdk2410 为例）

flash.c Flash 操作代码

memsetup.s 初始化 SDRAM 代码

u-boot.lds 对应的连接文件

（3）lib arm 体系结构下的相关实现代码，比如 memcpy 等的汇编语言的优化实现。

3．3 交叉编译环境的建立

要得到下载到目标板的 u-boot 二进制启动代码，还需要对下载的 u-boot1.1.1 进行编译。u-boot 的编译一般在 Linux 系统下进行，可用 ARM-LINUX-GCC 进行编译。一步一步建立交叉编译环境通常比较复杂，最简单的方法是使用别人编译好的交叉编译工具，方法如下：

（1）在http://handhelds.org/download/toolchain下载 arm-Linux-gcc-3.3.2.tar.bz2

（2）以用户名 root 登录，将 arm-linux-gcc-3.3.2.tar.bz2 解压到 /root 目录下

#tar jxvf arm-linux-gcc-3.3.2.tar.bz2

（3）在http://handhelds.org/download/toolchain下载 arm-linux-toolchain-post-2.2.13.tar.gz 只是用了它的头文件而已，主要来自内核/linux-x.x/include下

（4）将 arm-linux-toolchain-post-2.2.13.tar.gz 解压到 /skiff/local 下

#tar zxvf arm-linux-toolchain-post-2.2.13.tar.gz

（5）拷贝头文件到/root/usr/3.3.2/arm-linux/下然后删除 /skiff

#cp -dR/skiff/local/arm-linux/include/root/usr/3.3.2/arm-linux

#rm -rf /skiff

这样就建立了 arm linux 交叉编译环境。

（6）增加/root/usr/local/arm/3.3.2/bin 到路径环境变量

Path = $path:/root/usr/local/arm/3.3.2/bin 可以检查路径变量是否设置正确。#echo $path

3．4 移植的预先编译

移植 u-boot 到新的开发板上仅需要修改与硬件相关的部分即可。主要包括两个层面的移植，第一层是针对 CPU 的移植，第二层是针对 BOARD 的移植。由于 u-boot1.1.1 里面已经包含 S3C44B0 的移植，所以笔者对板子 myboard 的移植主要是针对 BOARD 的移植。移植之前需要仔细阅读 u-boot 目录下的 readme 文件，其中对如何移植做了简要的介绍。为了减少移植的工作量，可以在 include/config 目录下选一个和要移植的硬件相似的开发板，笔者选的是 b2 开发板。具体步骤如下：

（1）u-boot1.1.1 下的ＣＰＵ文件夹里已经包括了 S3C44B0 的目录，其下已经有 start.s 、interrupts.c 以及 cpu.c、 serial.c 几个文件，因而不需要建立与 CPU 相关的目录。

（2）在 board 目录下创建 myboard 目录以及 my-board.c、flash.c、memsetup.s 和 u-boot.lds 等文件。不需要从零开始创建，只需选择一个相似的目录直接复制过来，然后修改文件名及内容即可。笔者在移植 u-boot 过程中选择的是 u-boot1.1.1 / board / dave / B2 目录。

（3）在 include/configs 目录下添加 myboard.h，在这里可放入全局的宏定义等　也不需要从头创建，可以在 include/configs 目录下寻找相似的 CPU 的头文件进行复制，这里笔者用的是 B2.h 文件来进行相关的修改。

（4） 对 u-boot 根目录下的 makefile 文件进行修改，加入

Myboard_config:unconfig@./mkconfig$(@:_config =)arm S3C44B0 myboard

（5） 修改 u-boot 根目录下的 makefile 文件，加入对板子的申明。然后在 makefile 中加入 myboard、LIST ARM7 = “B2 ep7312 impa7 myboard”。

（6）运行 make clobber，删除错误的 depend 文件。

（7）运行 make myboard config。

（8）执行到此处即表示整个软件的 makefile 已建立，这时可修改生成的 makefile 中的交叉编译选项，然后打开 makefile 文件，并找到其中的语句：

Ifeq($(ARCH), arm)

CROSS_COMPILE = arm-linux-end if

接着将其改成

Ifeq($(ARCH), arm)

CROSS COMPILE = /root/usr/local/3.3.2/bin/arm-linux-end if

这一步和上面的设置环境变量只要有一个就可以了。

执行 make，报告有一个错误，修改 myboard/flash.c 中的#include “../common/flash.c”为“u-boot/board/dave/common/flash.c,重新编译即可通过。

4 移植时的具体修改要点

若预先编译没有错误就可以开始硬件相关代码的移植，首先必须要对移植的硬件有清楚地了解，如 CPU、CPU 的控制寄存器及启动各阶段程序在 Flash SDRAM 中的布局等。

笔者在移植过程中先修改/include/config/my-board.h 头文件中的大部分参数（大部分的宏定义都在这里设置），然后按照 u-boot 的启动流程逐步修改。修改时应熟悉 ARM 汇编语言和 C 语言，同时也应对 u-boot 启动流程代码有深入的了解。 B 2 板的 C P U 频率为 75MHZ、Flash 为 4Mbit、SDRAM 为 16Mbit、串口波特率为 115200bit/s、环境变量放在 EEPROM 中。根据两个开发板的不同，需要修改的有：CPU 的频率、Flash 和 SDRAM 容量的大小、环境变量的位置等。由于参考板已经有了大部分的代码，因此只需要针对 myboard 进行相应的修改就可以了。与之相关的文件有/include/config/myboard.h(大部分的宏定义都在这里设置）、/board/myboard/flash.c Flash 的驱动序 、/board/myboard/myboard.c（SDRAM 的驱动程序）、/CPU/S3C44B0/serial.c（串口的驱动使能部分）等。

/include/config/myboard.h 是全局宏定义的地方，主要的修改有：

将#define CONFIG_S3C44B0_CLOCK_SPEED 75 改为

＃#define CONFIG_S3C44B0_CLOCK_SPEED 64;

将 #define PHYS_SDRAM_1_SIZE 0x01000000 /* 16MB */ 改为

＃#define PHYS_SDRAM_1_SIZE 0x00800000 /* 8MB */;

将 #define PHIY_FLASH_SIZE 0x00400000 /* 4MB */改为

#define PHIY_FLASH_SIZE 0x00200000 /* 2MB */;

将 #define CFG_MAX_FLASH_SECT 256 /* max number of sectors on one chip*/ 改为

#define CFG_MAX_FLASH_SECT 35 ;

将 #define CFG_ENV_IS_IN_EEPROM 1　/* use EEPROM for environment vars */ 改为

#define CFG_ENV_IS_IN_FLASH 1

其它（如堆栈的大小等）可根据需要修改。

由于 Flash、SDRAM 的容量会发生变化，故应对启动阶段程序在 Flash、SDRAM 中的位置重新作出安排。笔者将 Flash 中的 u-boot 代码放在 0x0 开始的地方，而将复制到 SDRAM 中的 u-boot 代码安排在 0xc700000 开始的地方。

Flash 的修改不仅和容量有关，还和具体型号有关，Flash 存储器的烧写和擦除一般不具有通用性，应查看厂家的使用说明书，针对不同型号的存储器作出相应的修改。修改过程中，需要了解 Flash 擦写特定寄存器的写入地址、数据命令以及扇区的大小和位置，以便进行正确的设置。

SDRAM 要修改的地方主要是初始化内存控制器部分，由 start.s 文件中的 cpu init crit 完成 CPU cache 的设置，并由 board/myboard/memsetup.s 中的 memsetup 完成初始化 SDRAM。S3C44B0 提供有 SDRAM 控制器，与一些 CPU 需要 UPM 表编程相比，它只需进行相关寄存器的设置修改即可，因而降低了开发的难度。

串口波特率不需要修改（都是 115200bit/s），直接用 B2 板的串口驱动即可。串口的设置主要包括初始化串口部分，值得注意的是：串口的波特率与时钟 MCLK 有很大关系，详见 CPU 用户手册。

配置好以后，便可以重新编译 u-boot 代码。将得到的 u-boot.bin 通过 JTAG 口下载到目标板后，如果能从串口输出正确的启动信息，就表明移植基本成功。实际过程中会由于考虑不周而需要多次修改。移植成功后，也可以添加一些其它功能（如 LCD 驱动等），在此基础上添加功能相对比较容易。

5 结束语

u-boot 是一个功能强大的 bootloader 开发软件，适用的 CPU 平台及支持的嵌入式操作系统很多。本文是笔者在实际开发过程中根据相关资料进行摸索，并在成功移植了 u-boot 的基础上总结出来的。对于不同的 CPU 和开发板，其基本的方法和步骤是相同的，希望能对相关嵌入式系统的设计人员有所帮助。