

EDB9315A 之简化版 BootLoader

作者: Luocheng.sz@hotmail.com

目录

说明	2
第 0 章 开发环境	2
0.1 EDB9315A	2
0.2 J-Link	2
0.3 ARM RDS V2.2	2
第 1 章 EP9315 及开发板简介.....	3
1.1 EP9315: 基于 ARM920T 的 SOC.....	3
1.2 启动模式设置	3
1.3 SDRAM	4
1.4 FLASH	5
第 2 章 BootLoader 的工作原理.....	5
2.1 接管 CPU 的思路	5
2.2 Real View CodeWarrior 设置.....	6
2.3 最小程序	8
2.4 编辑二进制文件	10
2.5 使用 J-Flash ARM 将文件下载到 FLASH.....	10
第 3 章 硬件初始化	14
3.1 与 ARM920T Core 有关的设置.....	15
3.2 系统控制器(System Controller).....	16
3.3 静态内存控制器(Static Memory Controller).....	18
3.4 SDRAM 控制器	19
3.5 串口初始化	26
3.6 硬件初始化小结	29
第 4 章 软件初始化	30
4.1 将程序搬运到 SDRAM 中去运行.....	30
4.2 设置内核标记列表.....	35
4.3 加载 zImage(内核镜像)和 RAMDISK.....	42
4.3.1 将 zImage 和 RAMDISK 下载到 FLASH 中.....	42
4.3.2 将 zImage 和 RAMDISK 装入 SDRAM 中.....	45
4.4 最后一跳!	45

说明

我写的这个 BootLoader 功能有限，完全出于学习目的。目前已实现的功能：

- | 系统启动后初始化硬件
- | 通过串口输出启动信息
- | 引导 ARM Linux 内核启动

未实现的功能：

- | 网络接口
- | FLASH 编程

第 0 章 开发环境

0.1 EDB9315A

EDB9315A 是 Cirrus Logic 公司于 2006 年左右推出的一款基于 EP9315 的开发板。EDB9315A 开发板的详细硬件技术细节请参考 Cirrus Logic 公司的各项**技术手册**：<http://www.cirrus.com/en/products/pro/detail/P1052.html>，建议重点研究的资料：

- | EDB9315A 开发板电路原理图：[EDB9315A_Tech_Ref_Manual.pdf](#)
- | 用户开发指南：[EP93xx_Users_Guide_UM1.pdf](#)

0.2 J-Link

使用了 J-Link(v7)之后，我才体会到了好的工具对于 ARM 嵌入式开发的重要性。J-Link 提供了一整套软硬件工具，通过 ARM 处理器的 JTAG 接口，可以很方便地查看/修改 CPU/外设的寄存器/存储器（J-Link Commander），以及对开发板上的 nor FLASH 进行在线编程（J-Flash ARM）。

0.3 ARM RDS V2.2

RealView Developer Suite v2.2，是 ARM 公司的软件/调式开发套件，提供了 CodeWarrior 集成开发环境（IDE）和 AXD 等在线调式器。AXD 结合 J-Link，就可以进行源代码级的在线调式。

第 1 章 EP9315 及开发板简介

1.1 EP9315: 基于 ARM920T 的 SoC

EP9315 的功能结构框图如图 1.1 所示。(其中红色方框是我加上的,表示在写 BootLoader 的过程中需要重点研究的部分)

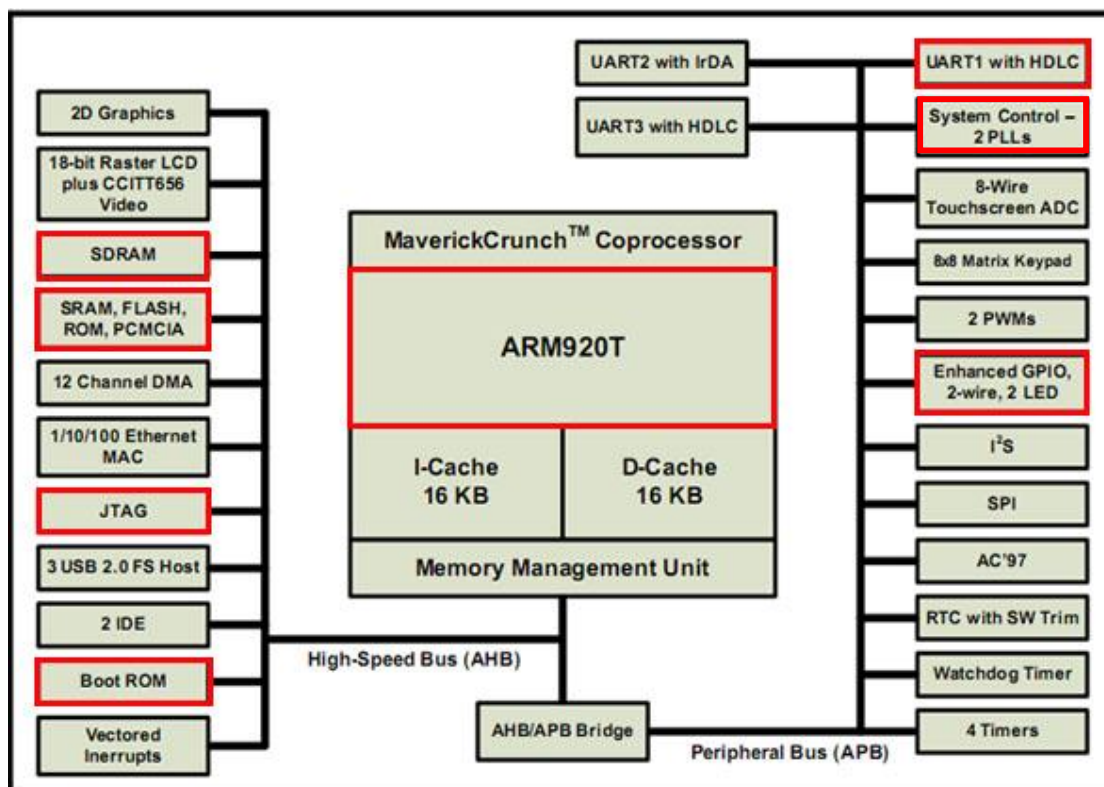


图 1.1 (本图引用自 [Cirrus Logic 公司的《用户开发指南》](#))

1.2 启动模式设置

[《EDB9315A_Tech_Ref_Manual.pdf》](#) 第 12 页:

There are **10 signals** that **determine how the EP9315 will boot and operate**. They are all shown on this page except for BOOT1. BOOT1 is connected to GND. BOOT1 is used for factory testing only. The other nine signals are either pulled up or down external to the EP9315 device. The boot configuration shown sets the EP9315 device to perform an **Internal Async boot** from a **16-bit-wide** memory.

以上节选的这段文字说明了 EDB9315A 开发板将启动模式配置为: Internal Async boot (内部异步启动), 并且启动代码是存放在数据线宽度为 16-bit 的存储器中。在设计基于 EP93XX 系列处理器的硬件电路时, 请参看完整文档(《[EP93xx_Users_Guide_UM1.pdf](#)》第 4 和第 5 章), 仔细研究如何设置启动模式的相关部分内容。

1.3 SDRAM

《EP93xx Users Guide UM1.pdf》第 499 页：

Four pins, SDCSn[3:0], are used to as chip-selects (domain selects) for the four synchronous memory domains.

在设计硬件电路时，需要注意 SDRAM 的片选信号的配置与地址空间的映射问题。

《EDB9315A Tech Ref Manual.pdf》第 12 页：

The SDRAM interface is comprised of **two 16-bit** SDRAM devices to **form a 32-bit** SDRAM bus. The SDRAM is connected to /SDCS0 and is located at physical memory address **0xC000_0000**.

《EP93xx Users Guide UM1.pdf》第 499 页：

Because of the row/column/bank architecture of synchronous memory devices, the mapping of these memories into the processor's memory space is not always obvious, typically because the memory inside a synchronous device **does not appear to the processor to be continuous**. For example, a 32-Mbyte SDRAM device may be visible as **four** 4-Mbyte banks.

译文：

由于同步存储器内部的行/列/块结构，使得映射到处理器的地址空间可能不是很直观，例如一片 SDRAM，映射到处理器的地址空间后，会被分为地址不连续的 2 块或多块。比如，一片 32M 字节的 SDRAM，可能会被分为 8 个块，每块的大小为 4M 字节。

EDB9315A 开发板上的两片 SDRAM 总共为 64M 字节，被分为物理地址不连续的两块：

0xC0000000 ~ 0xC1FFFFFF 32M Bytes

0xC4000000 ~ 0xC5FFFFFF 32M Bytes

手册里也大概解释了地址不连续的原因（部分参见以上的引文，红色部分应该是原文有误，我作了修改），我也没有仔细去研究，希望对此有研究的朋友能不吝赐教。在进行软件开发的时候，可以不用去关心原因，但一定要知道地址不连续的这个事实。

这两片 SDRAM（单片为 16 位）组成的数据总线宽度为 32 位。我们知道 ARM 的数据总线为 32 位，但一般在描述存储器容量或指令地址的时候又是以字节（Byte/8 位）为单位，这对于初学者来说，有时可能会造成困扰。我们不妨这样来看：ARM 每次读入 32bit 的数据，其实可理解为一次性读入了 4 个字节的数据，CPU 内部会自动对这 4 个字节的数据（或指令）进行相应处理，而存储器地址是按照字节来计算的，因此读完一次后存储器地址要加 4，表示每次读出 4 个字节。这也就是所谓“4 字节对齐”的含义，因为**数据或指令在存储器中是以 4 字节为单位存放的**。而像 8051 那样的 8 位 CPU 的数据或指令是以一个字节为单位存放的，那么如果只读入半个字节的内容就会出问题了。

1.4 FLASH

《EDB9315A Tech Ref Manual.pdf》第 12 页：

The Flash interface is made from a single **16-bit** device.

The Flash device is connected to /CS6 and is located at physical memory address **0x6000_0000**. The reference design uses only one Flash device. However, the EP9315 can be designed with a 32-bit Flash interface as well. The Flash device installed is a 128Mbit, **16Mbyte**, device.

开发板上的 nor FLASH 容量为 16M 字节（型号为：Intel 28F128J3D），物理起始地址为 0x60000000（这块 16M 字节地址是连续的）。数据线宽度为 16-bit。这样看来 ARM9 要读两次，才能将一个单位（32-bit）的数据读入。其实这个工作是由 EP9315 内部的 Static Memory Controller（静态存储控制器）来完成的（细节请参考《EP93xx_Users_Guide_UM1.pdf》第 12 章）。通过配置这个控制器，可以外接多种类型的 FLASH，可以将数据总线宽度设置为 8 位，16 位或者 32 位。在进行硬件设计，特别是改用其它类型的 FLASH 时，请务必仔细研究这部分的内容。

第 2 章 BootLoader 的工作原理

目前在 ARM9 平台上比较常见的开源 BootLoader 有 U-Boot 和 Redboot 等，其中 Redboot 是 ecos 实时操作系统的一部分。这些开源 BootLoader 的通用性很强，功能也很全面。当我们这些初学者研究这些开源软件的代码时，往往会觉得抓不住重点，因为为了实现软件的通用性，大部分的代码都在进行各种繁琐的配置工作。而实现软件的通用性（或者称为“可移植性”）其实已经是另外一个层次的功力了（这当然已属于高手的实力范围），像我这样初级水平的工程师其实更愿意把精力放在手头的平台上，因此我决定自己为手上的这块 ARM9 开发板写一个 BootLoader。这样做的目的是为了要找出重点，然后理解并掌握重点。

BootLoader 的功能可以概括为：

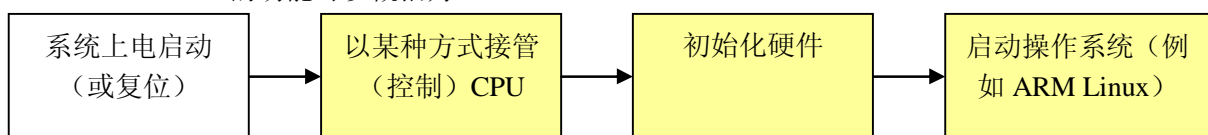


图 2.1

上图中黄色部分为 BootLoader 所要完成的工作。

2.1 接管 CPU 的思路

所有的 ARM 教科书上都会告诉你，CPU 复位后从地址 0 开始运行。那么我们如何才能将自己的代码放到地址 0 处呢？ARM 教科书上没有告诉我，因为这个问题没有标准答案。各个厂家实现的方法各不相同。

EP9315 有它自己的存放启动代码的方案。它在芯片的内部固化了一段启动代码，叫做“Boot ROM”（请参看图 1.1），当系统的启动模式配置为 Internal Async boot，复位后会将 Boot ROM 的地址自动映射为 0x00000000，然后就执行 Boot ROM 中的代码。其实可以将

Boot ROM 理解为一个已经固化了的 BootLoader，具体的技术细节请参考《EP93xx_Users_Guide_UM1.pdf》第 4 章。当然我们可以用 AXD + J-Link 来查看这段代码（Boot ROM 的物理起始地址为 0x80090000），网络上也早有人贴了出来：

<http://www.freelists.org/post/linux-cirrus/EP9312-internal-ROM.4>

其实我自己正是从这段代码开始学习 EP9315 的。Boot ROM 所做的工作相对比较简单，还无法满足启动操作系统的要求，因此还要在此基础上完成功能更加复杂的 BootLoader。

还有一个重要的必须知道的细节就是，EP9315 内部有一块大小为 2K 字节的 SRAM（它默认的功能是作为以太网接口的缓存），物理起始地址为：0x80014000。这块 SRAM 在 BootROM 启动过程中发挥了重要的作用。

Boot ROM 的工作流程大致可以分为如下几个主要部分：

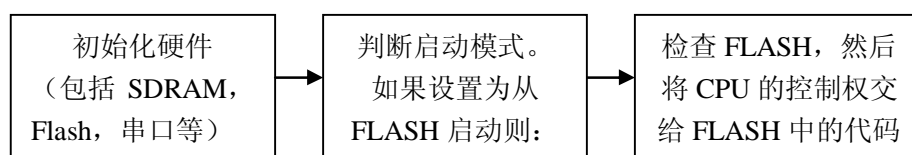


图 2.2

看到这里就基本上已经找到了如何接管 CPU 的线索了。那就是将我们自己的代码放在 FLASH 中的合适的地方。那么哪里才是合适的地方呢？技术手册里已经明确地给出了答案：

《EP93xx Users Guide UM1.pdf》第 121 页：

```
Attempt to read the "CRUS" or "SURC" HeaderID in ASCII in FLASH memory at FLASH Base + 0x0000 ( and Base + 0x1000 ), and verify the HeaderID.
Jump to the start of FLASH memory plus four bytes
```

大意是说，Boot ROM 会在 FLASH 的起始地址处查找有效的 HeaderID：就是字符串“CRUS”或“SURC”，如果找到了，就跳转到字符串后的第一条指令去执行。如果没有找到 HeaderID，就会到 FLASH 起始地址+0x1000 的地方去再找一次。

那么我们就在 FLASH 的起始地址处放上字符串“CRUS”或者“SURC”，然后紧接着放我们自己的代码。注意，这个字符串的长度是 4 个字节，正好满足了“4 字节对齐”的要求。

2.2 RealView CodeWarrior 设置

关于 RealView Developer Suite 和 J-Link 的安装与配置在这里不作详细介绍，网络上已有很多资料了（其实最权威和详细的资料请参看各个工具所自带的官方文档，我觉得这些才是获取相关知识与灵感的最可靠的来源）。我着重谈一下我的思路：编写一个小程序，然后用 J-Link 下载到 FLASH 中，系统复位后能够自动运行这段程序，以达到接管 CPU 的目的。

首先运行 CodeWarrior，新建一个工程：startup_LED。

该工程默认为 Debug 模式，我们就在该模式下进行“Debug Settings”设置：

- I 在“Language Settings”里将“Assembler”(汇编)和“Compiler”(C 编译器)的“Architecture or Processor”都设置为 ARM920T。
- I 在“Linker”下的“RealView Linker”里，将“RO Base”的值设为：0x60000000。表示实际运行时(Runtime)，代码是从地址 0x60000000 (FLASH 的起始地址)开始存放的。如图 2.3 所示。（这里的信息其实是提供给“加载”程序用的，请参考第 4 章）
- I 在“RealView FromELF”中，“Output file name”设置为“startup.bin”。CodeWarrior 编

译和链接完成后，默认生成的是 ARM ELF 格式的可执行文件（AXF 文件），这种文件是依靠 AXD 等工具来进行加载的，而我们所要的是二进制（或者说是机器码+数据）文件，可以直接下载到 FLASH 中去。这里的设置就是让 CodeWarrior 调用 FromELF 工具生成一个二进制文件。如图 2.4 所示。

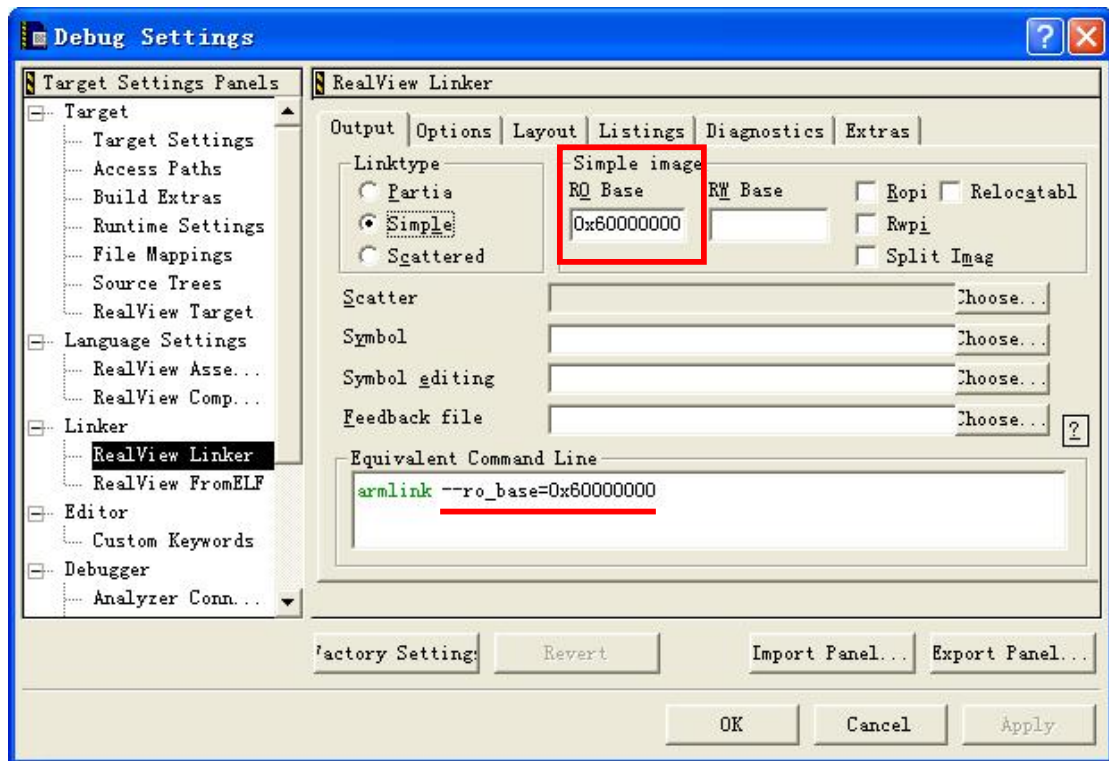


图 2.3

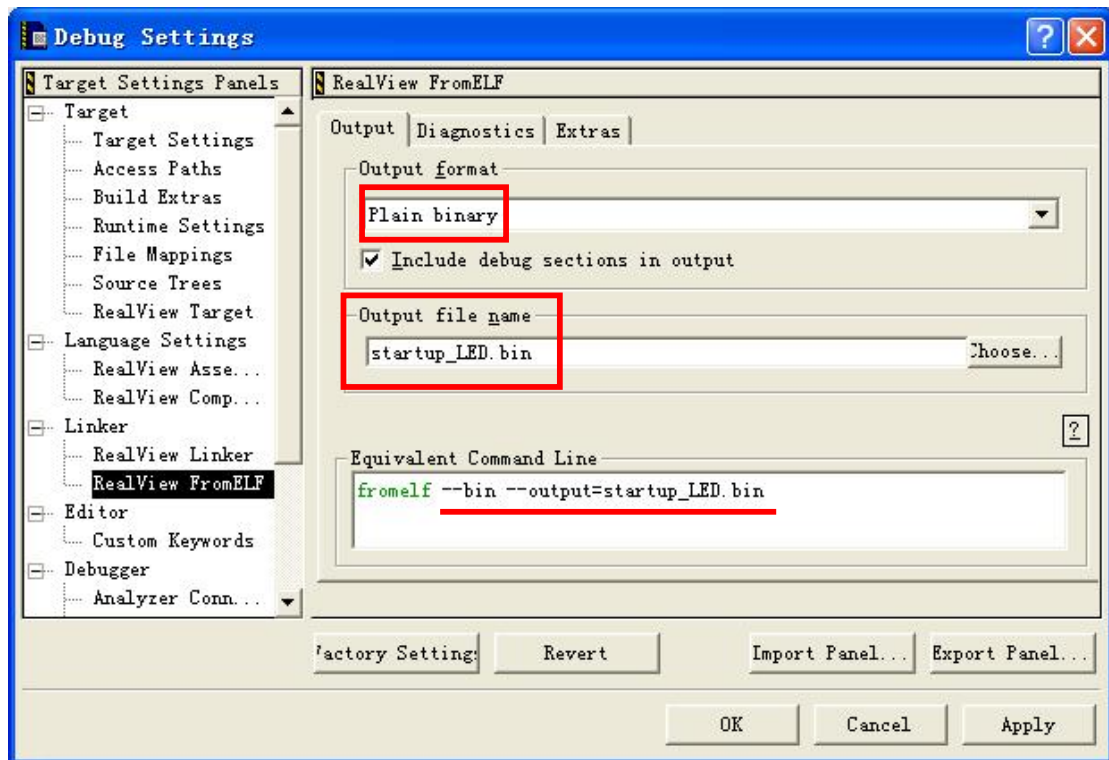


图 2.4

2.3 最小程序

这个程序很简单，就是要告诉大家：“我运行起来了！”那么最直观的方式就是让板子上的 LED 闪烁。

EDB9315A 开发板上有两个 LED，一绿一红。其中红色的 LED 还有一个“奇特”的功能，就是上电/复位时充当看门狗 (WatchDog) 的角色。其实这是因为 EP93XX 处理器有一个非常严重的 BUG，就是上电/复位不稳定，会出现无法启动的情况！（请参看 Cirrus Logic 的官方技术说明 [AN258REV2.pdf](#)）。作为解决方案，就是用红色 LED 的控制脚再配合外部电路做成一个看门狗（具体电路细节请参看 [《EDB9315A Tech Ref Manual.pdf》](#) 第 32 页的电路原理图），在正常启动时，红色 LED 点亮的时候很短，如果其点亮的时间过长，就说明启动失败，要重新复位。为什么要在这里说这么多关于 LED 的话题呢？目的有两点：

1. 本开发板上的红色 LED 的点亮时间不能太长，否则系统会被外部硬件自动复位。
2. Cirrus 解决这个 BUG 的思路和方法很有意思，值得我们参考。

我们的小程序就是让红绿两个 LED 轮流点亮，其中红色的点亮时间很短（当然，最安全的做法就是程序运行后立刻关掉红色 LED，在这里点亮只是为了演示的目的），绿色的点亮时间较长。这两个 LED 是由 EP9315 的两个 GPIO 管脚控制的，对应的 GPIO 数据寄存器物理地址为：0x80840020，其中位 0 控制绿色 LED，位 1 控制红色 LED（GPIO 的细节请参考 [《EP93xx Users Guide UM1.pdf》](#) 第 28 章）。

程序流程图如下：

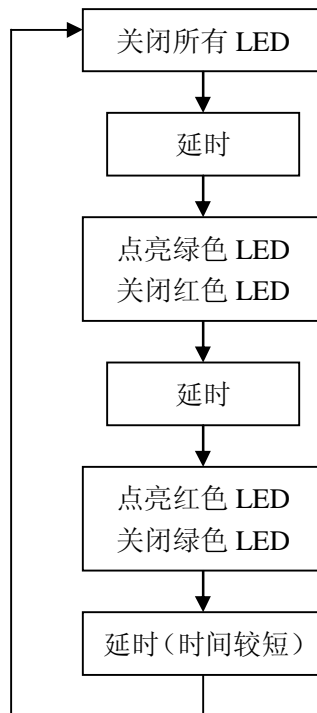


图 2.5

汇编语言源程序如下：


```

1      AREA    startup_LED, CODE, READONLY
2      ENTRY
3
4      Main
5      LDR     r0, =0x80840020      ; GPIO Register
6      LDR     r1, [r0, #0]
7      Loop
8      LDR     r2, =0xFFFFFFFF
9      AND     r1, r1, r2
10     STR     r1, [r0, #0]        ; Turn off all LED
11
12     MOV     r3, #0x20000
13     Loop_1
14     SUBS    r3, r3, #1          ; Delay
15     BNE     Loop_1
16
17     MOV     r2, #0x01
18     ORR     r1, r1, r2
19     STR     r1, [r0, #0]        ; Turn on green LED, and off red
20
21     MOV     r3, #0x20000
22     Loop_2
23     SUBS    r3, r3, #1          ; Delay
24     BNE     Loop_2
25
26     MOV     r2, #0x02
27     ORR     r1, r1, r2
28     STR     r1, [r0, #0]        ; Turn on red LED, and off green
29
30     MOV     r3, #0x2000
31     Loop_3
32     SUBS    r3, r3, #1          ; Delay
33     BNE     Loop_3
34
35     B       Loop
36
37     END

```

源代码 2.1 startup_LED.s

编译链接成功后，会在工程目录下生成二进制文件：startup_LED.bin。

2.4 编辑二进制文件

如果直接将该二进制文件下载到 FLASH 中还是无法运行的，因为前面已经看到，Boot ROM 会去查找 Header ID，所以我们用手工的方式将字符串“CRUS”添加到文件的最前端。我们用 UltraEdit 来进行这项工作。

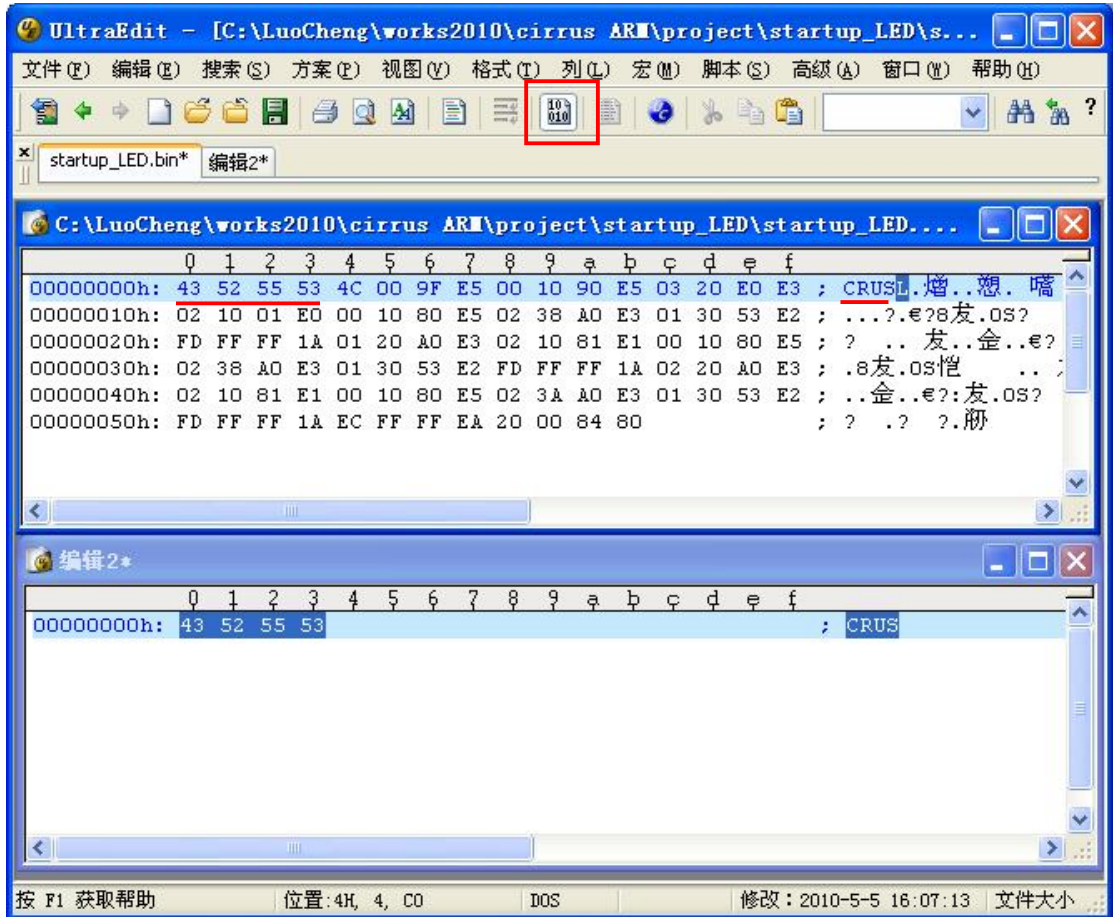


图 2.6

注意要在“十六进制模式”下进行“复制”和“粘贴”（图中红框部分）。如果你还不熟悉 UltraEdit 的使用，建议你花时间研究一下，因为在嵌入式开发中，这是个很有用的工具。

2.5 使用 J-Flash ARM 将文件下载到 FLASH

J-Link 通过 JTAG 接口与 EDB9315A 开发板连接，硬件连接好后，我们运行 J-Link 的 FLASH 编程工具：J-Flash ARM，这个工具使用前需要做一些设置。首先设置 CPU，如下图所示：

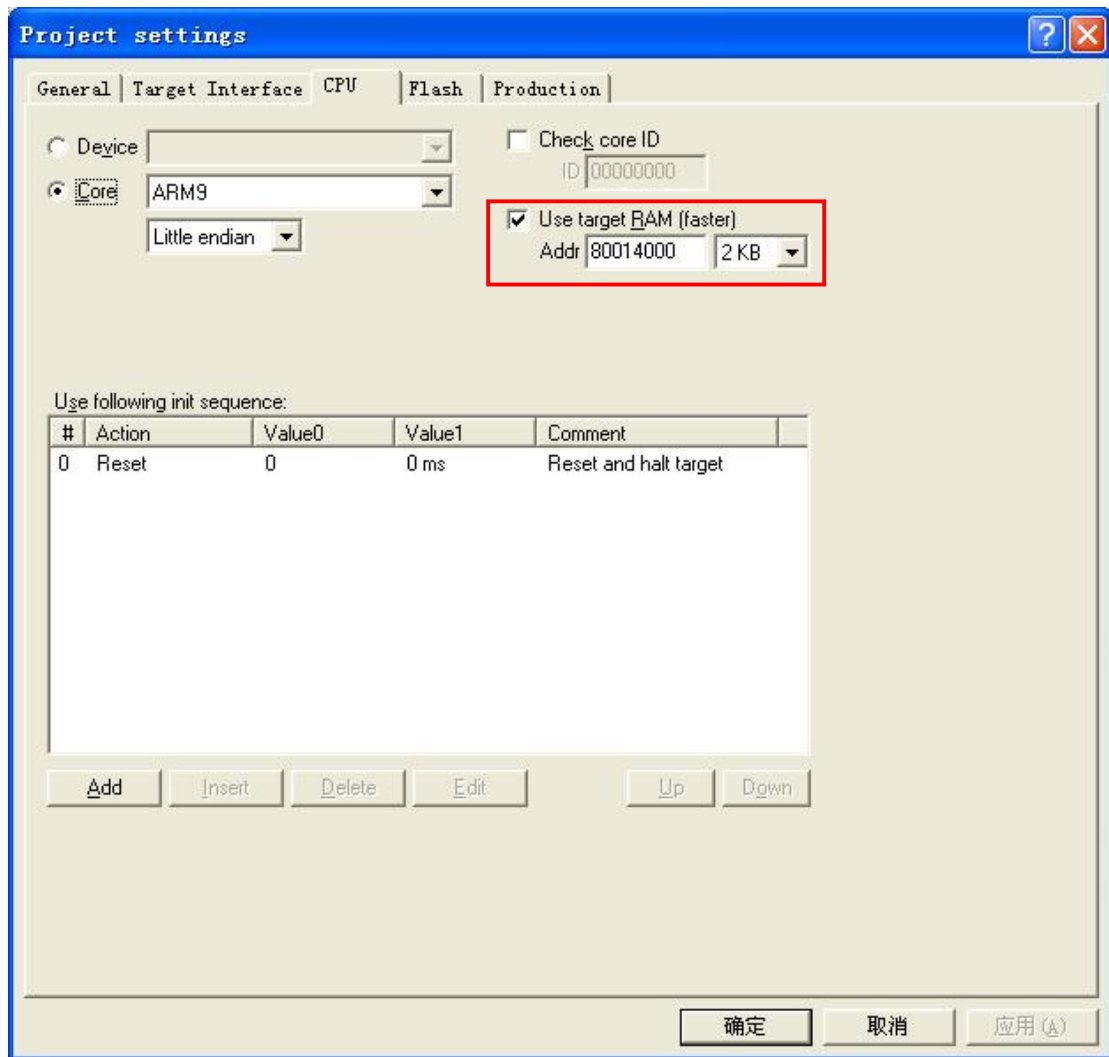


图 2.7

CPU 内核选 ARM9，默认为小端（Little endian）模式。使用目标 RAM，地址设为“80014000”，大小为 2KB（前面已经介绍过这块 SRAM 了）。这个设置很重要，如果不使用片内 SRAM 的话，FLASH 的下载速度将慢的无非忍受！

然后设置 Flash 选项：

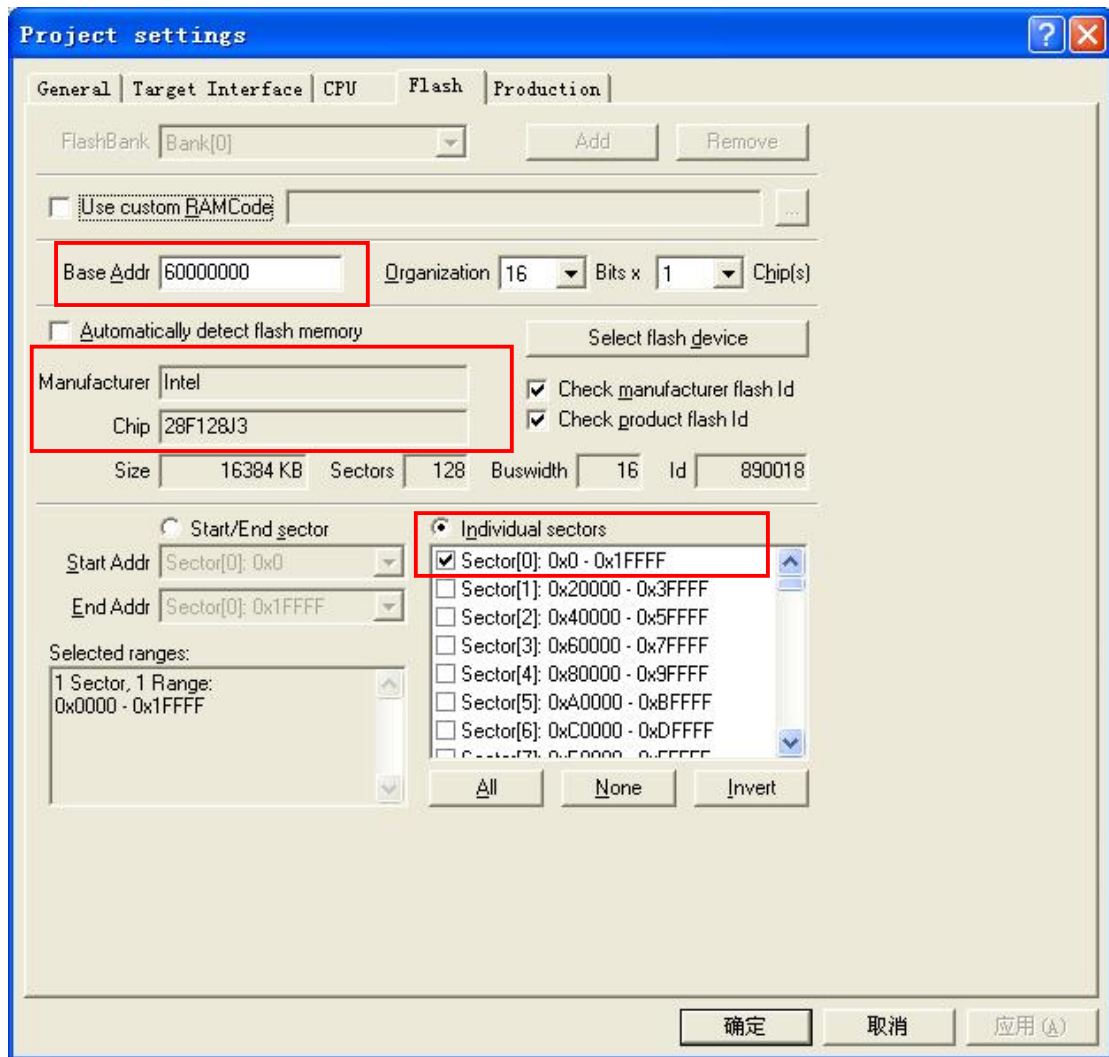


图 2.8

选择 FLASH 的厂家和型号，基地址（Base Addr）设为“60000000”（开发板上 FLASH 的物理起始地址）。该 FLASH 共分为 128 块，我们目前只需要（也必须要）对第 1 块进行编程，所以选中 Sector[0]就可以了，这样可以大大加快下载的速度。

然后调入已修改好的二进制文件：startup_LED.bin，从 FLASH 基地址处开始编程。在对 FLASH 编程之前，必须要先进行擦除，然后进行编程与校验（如下图 2.9 所示）。

还有一点需要说明，有时 J-Flash ARM 会报错：无法擦除或下载，这时需要手动将开发板复位，然后再进行擦除和下载。（这可能和 J-Link 的兼容性有关）

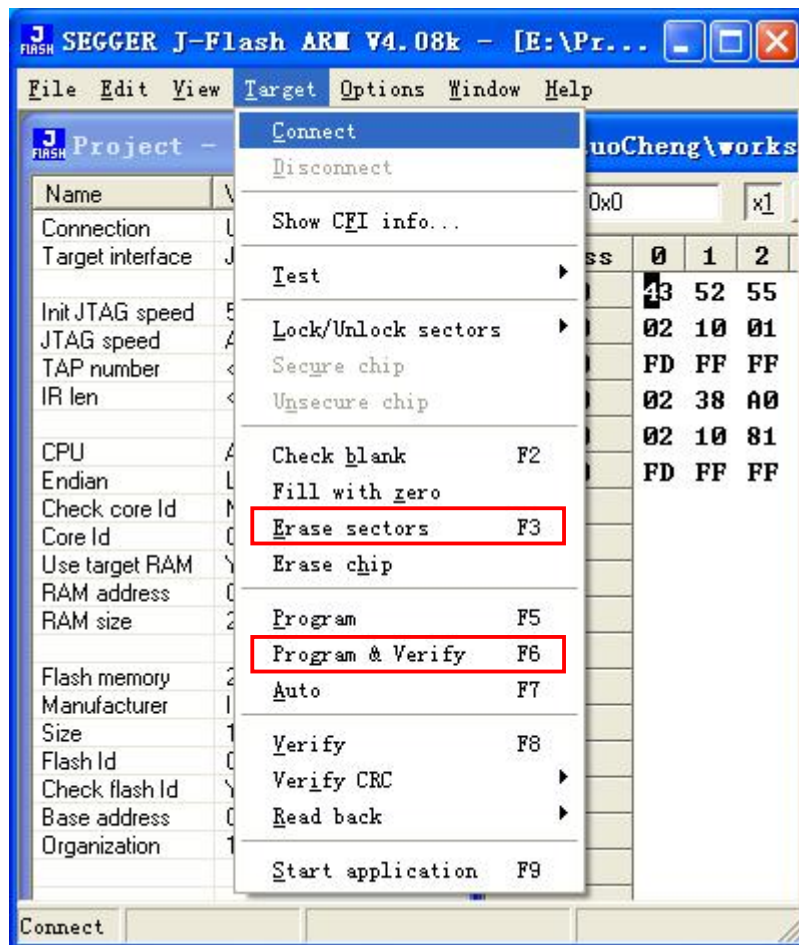


图 2.9

下载成功后，复位开发板，这时应该就可以看到两个 LED 在连续交替地闪烁了，这也表示我们接管 CPU 的工作成功了！

谈点关于 JTAG 的话题

JTAG 在嵌入式开发中有着举足轻重的作用。作为使用者来说，我们不仅要会熟练地运用各种基于 JTAG 的开发工具，也需要掌握一些 JTAG 的基本原理，这样在开发过程中对于出现的一些问题就能更好的理解和解决了。

我个人把 JTAG 理解为一个处于 CPU 与外设之间的“中间件”，通过硬件和 JTAG 协议激活这个“中间件”后，它就开始发挥其“欺上瞒下”的功能了。在 CPU 看来，这个“中间件”就是完整的外设；而在外设看来，它俨然就是发号施令的 CPU。

在进行调试时，通常“中间件”首先要“停止”CPU 的运转，然后才可以读出各个寄存器的内容，通过 JTAG 接口发给使用者，然后再让 CPU 单步或连续运行。所以有时候我们看到很多 JTAG 工具报错的原因，其实是无法正确地“停止”CPU，那么我们就要想办法让它在正确的“时间和地点”去“停止”CPU。

至于利用它来控制外设，比如查看/修改 SDRAM 的内容，下载程序到 Flash 等等，也就容易理解和掌握了。

第 3 章 硬件初始化

当我们接管了系统之后，就要开始进行硬件初始化。在没有时间与能力做“高大全”的 BootLoader 的情况下，我们需要找出重点，只做有针对性的工作。那么具体针对什么呢？就是要满足 ARM Linux 内核启动的条件。

ARM Linux 内核是由 Russell King 维护的，将开源操作系统 Linux 移植到了 ARM 平台上（在此向这些做出过无私奉献的高手们表示敬意）。官方网站是：

<http://www.arm.linux.org.uk>

其中明确提出了 [ARM Linux Kernel Boot Requirements](#)（ARM Linux 内核启动要求），这篇文章不长，值得我们仔细阅读和体会。作者建议 BootLoader 要做的工作有：

1. Setup and initialise the RAM. 设置和初始化 RAM
2. Initialise one serial port. 初始化一个串口
3. Detect the machine type. 检测机器的类型
4. Setup the kernel tagged list. 设置内核标记列表
5. Call the kernel image. 调用（启动）内核镜像

然后在最后还列出了启动 ARM Linux 内核镜像必须满足的条件：

In either case, the following conditions must be met:

* CPU register settings

r0 = 0.

r1 = machine type number discovered in (3) above.

r2 = physical address of tagged list in system RAM.

* CPU mode

All forms of interrupts must be disabled (IRQs and FIQs.)

The CPU must be in SVC mode. (A special exception exists for Angel.)

* Caches, MMUs

The MMU must be off.

Instruction cache may be on or off.

Data cache must be off and must not contain any stale data.

* Devices

DMA to/from devices should be quiesced.

- * The boot loader is expected to call the kernel image by jumping directly to the first instruction of the kernel image.

大致翻译如下：

在任何情况下都必须满足以下条件：

I CPU 寄存器设置

r0 = 0

r1 = 机器类型码（各个厂家的 ARM CPU 和官方的开发板都有各自的机器类型码）

r2 = 标记列表在系统内存中的物理地址

I CPU 的模式

所有类型的中断都必须关闭（包括 IRQs 和 FIQs）

CPU 必须处于 SVC 模式（只对于 Angel 调试器有所例外）

I 高速缓存和 MMUs

MMU 必须关闭

指令高速缓存可以打开或关闭

数据高速缓存必须关闭，并且当中不能残留任何无用的数据

l 设备

连接设备的 DMA 通道应当被关闭

l boot loader 调用内核镜像的方法应当是：直接跳转到内核镜像的第一条指令

另外一个重要的也是必要的学习 BootLoader 的方法就是阅读成熟的开源 BootLoader(比如 U-Boot 和 RedBoot) 的源代码。其实我的 BootLoader 的代码几乎全部来自于上述两个开源软件的源码。我在前面说过，开源软件的源码并不容易读，尤其像我们这些母语非英语的初学者，很多思路完全不同于那些用英语思考和写作的高手，再加上我们受过的专业训练和工程化开发方法的缺失，的确造成了很多的障碍（我一直以为高校的工程专业课应该完全用英语教学，并且着重增加软件开发工程学的英文课程。当然在目前的体制下，这些都是幻想）。但是不容易读还是要读，这需要我们想出各种办法去逐步读懂它们，这也是我写这个最小 BootLoader 的初衷之一。

已经移植完成的 EDB9315A 的 U-Boot 和 RedBoot 的源代码在 Cirrus Logic 的网站上都下载到：<http://arm.cirrus.com>。具体在 Linux 环境下的安装编译过程我就不再多说，这方面的资料已经很多了，而且这些都是需要反复实践和体会的“体力活”，光看不练没法搞懂，真的是“谁用谁知道”。

3.1 与 ARM920T Core 有关的设置

关于 ARM920T Core 的功能，请参考《[EP93xx Users Guide UM1.pdf](#)》第 2 章，以及 ARM 公司的官方文档《[ARM920T Technical Reference Manual](#)》。

修改 ARM920T 的一些寄存器及协处理器的寄存器都需要在特权模式下进行。而 CPU 复位后自动进入特权模式。我们在这里要进行的设置有：

l 设置 CPSR（当前程序状态寄存器），关闭所有中断，并让 CPU 进入 SVC 模式

l 设置协处理器 CP15 的寄存器，关闭 MMU 和 Cache（高速缓存）

l 设置协处理器 CP15 的寄存器，使处理器时钟模式处于 Sync mode（同步模式）

（请参考《[EP93xx Users Guide UM1.pdf](#)》第 2 章和第 5 章）

具体代码如下：

```

1 Setup_920T
2 ;*****
3 ; Make sure the processor is in SVC32 mode with IRQ and FIQ disabled.
4 ;*****
5 MOV      r0, #0xd3
6 MSR      cpsr_cf, r0
7
8 ;*****
9 ; Make sure caches are off and invalidated.
10 ;*****
11 MOV      r0, #0
12 MCR      p15, 0x0, r0, c1, c0, 0
13 MCR      p15, 0x0, r0, c7, c7, 0 ; turn off I/DCache
14 MCR      p15, 0x0, r0, c8, c7, 0 ; turn off I/D TLBs
15 NOP
16 NOP
17 NOP
18 NOP
19 NOP
20
21 ;*****
22 ; Force the nF bit on .Set the Processor clocking modes to Sync mode
23 ;*****
24 MRC      p15, 0, r0, c1, c0 ; get control register v4
25 ORR      r0, r0, #0x40000000 ; set nF
26 MCR      p15, 0, r0, c1, c0
27 ;////////////////////
28 ; End of Setup_920T
29 ;////////////////////

```

源代码 3.1

3.2 系统控制器(System Controller)

EP9315 系统控制器的详细功能请参见 [《EP93xx Users Guide UM1.pdf》](#) 第 5 章，其中主要包括：

- I 系统时钟控制
- I 电源管理
- I 系统配置管理

初次接触这些概念，可能会觉得有些抽象，但是动手实践过之后，就会发现其实都很具体和实用。

关于系统时钟，首先要了解的就是，EP9315 内部有两个相互独立的 PLL（锁相环电路），由这两个 PLL 将外部的 14.7456MHz 时钟倍频后再生成各种时钟信号，供 CPU，总线及外设使用（详细内容请参考 [《EP93xx Users Guide UM1.pdf》](#) 第 130 页）。

我们再来看一段文档:

[《EP93xx Users Guide UM1.pdf》第 130 页](#)

The EP93xx uses a flexible system to generate required clocks. The clock system generates up to 20 independent clock frequencies, some with very tight accuracy requirements, all from a single external low-frequency crystal or other external clock source.

EP93XX 使用了一个灵活的系统去生成所需的各种时钟。该时钟系统产生了多达 20 种的独立的、不同频率的时钟信号,其中包括一些对精度要求很高的时钟信号,所有这些信号都源自于一个外部的低频晶体,或是外部的时钟源。

The ARM Core is designed so that once it has been configured, its CPU speed, bus speeds, and video clocks may be set to a number of different speeds without affecting the speeds of other clocks in the processor.

一旦 ARM 内核配置完成,它的 CPU 运行速度,总线速度和视频时钟都可以各自工作在不同的频率上,而且不会对处理器中的其它时钟信号造成影响。

从这里我们可以看到,系统时钟配置的优点是非常灵活,而缺点就是我们在初始化的时候,可能会搞不清楚怎样才是最好的配置方案。幸好有开源的代码可以参考,以下就是我参考了 U-Boot 和 RedBoot 之后写的系统时钟的配置代码:

```
1 Setup_Clock
2 ;*****
3 ; Set the PLL1 and processor clock.
4 ;*****
5 LDR    r0, =0x80930000
6 LDR    r1, =0x02a4e39e
7 STR    r1, [r0, #0x20]
8 NOP
9 NOP
10 NOP
11 NOP
12 NOP
13 ;*****
14 ; Set the PLL2 and USB clock.
15 ;*****
16 LDR    r1, =0x300dc317
17 STR    r1, [r0, #0x24]
18 ;//////////
19 ; End of Setup_Clock
20 ;//////////
```

源代码 3.2

建议大家仔细看一下介绍系统时钟的这一章,因为里面涉及到了各种外设的时钟来源与配置,了解这些原理对于今后处理外设出现的问题时会有很大的帮助。

系统配置中还包括了对 WatchDog (看门狗) 的控制。所有的 BootLoader 都要求尽可能地关闭 WatchDog, 而在 EP93XX 系统中, 这个关狗的工作已经被 Boot ROM 做过了, 所以这段程序我就省略了。

3.3 静态存储控制器(Static Memory Controller)

开发板上使用的是单片 16-bit 容量为 16M 字节的 nor FLASH。这里主要是对 FLASH 控制器进行设置 (详见 [《EP93xx Users Guide UM1.pdf》](#) 第 12 章)。

其实在 Boot ROM 运行的时候, 已经对 Flash 接口进行过设置了, 因为它能够正确找到我们放在 FLASH 中的代码, 而且我们的代码接管系统后, CPU 也是直接从 FLASH 里将每一条指令正确读出并执行的 (这就是所谓的程序 “在 FLASH 中运行”)。

但是 RedBoot 的程序员好像对 Boot ROM 中设置 FLASH 接口的这段代码不太满意, 因此他重写了一段, 并且还抱怨了几句。我全部照搬了 RedBoot 的这段代码:

```
1 Setup_Flash
2 ;*****
3 ; Undo the silly static memory controller programming
4 ; performed by the boot rom.
5 ;*****
6 LDR r0, =0x80080000 ; SRAM Controller, SMC Bank Configuration registers
7 LDR r1, =0x0000fbe0
8 LDR r2, [r0, #0]
9 ORR r2, r2, r1
10 STR r2, [r0, #0]
11 LDR r2, [r0, #4]
12 ORR r2, r2, r1
13 STR r2, [r0, #4]
14 LDR r2, [r0, #8]
15 ORR r2, r2, r1
16 STR r2, [r0, #8]
17 LDR r2, [r0, #0xc]
18 ORR r2, r2, r1
19 STR r2, [r0, #0xc]
20 LDR r2, [r0, #0x18]
21 ORR r2, r2, r1
22 STR r2, [r0, #0x18]
23 LDR r2, [r0, #0x1c]
24 ORR r2, r2, r1
25 STR r2, [r0, #0x1c]
26 ;////////////////////
27 ; End of Setup_Flash
28 ;////////////////////
```

源代码 3.3

在初始化的过程中,必须要将 **FLASH** 初始化放在时钟初始化之前,否则程序会“跑飞”,我猜测是因为 Boot ROM 的这部分代码有些问题。至于 Boot ROM 中的代码到底“傻”在了哪里,我还没有仔细对比过,希望有兴趣的朋友可以研究一下,并和大家分享一下体会。

3.4 SDRAM 控制器

EDB9315A 开发板上有两片 SDRAM,数据线为 32-bit,共 64M 字节,被分为物理地址不连续的两块:

0xC0000000 ~ 0xC1FFFFFF	32M Bytes
0xC4000000 ~ 0xC5FFFFFF	32M Bytes

SDRAM 控制器详细信息参见 [《EP93xx Users Guide UM1.pdf》](#) 第 13 章。

SDRAM 芯片内部其实可理解为由多个模块组成的复杂系统,除了存储器部分之外,还有一些控制模块,包括时钟/时序分配,地址复用,数据读写,数据刷新等等。当 CPU 访问 SDRAM 的时候,就先要对这些控制模块进行参数设置,让这些模块正确运转起来后,才能对存储器部分进行正常地读写。但是不同厂家生产的不同型号的 SDRAM,其内部控制模块的参数又各有差异,这就对 CPU 使用 SDRAM 造成了很大的麻烦。所以就在 CPU 之外集成了一个 SDRAM 控制器,那些设置参数的工作就让控制器来完成。

了解一些 SDRAM 的原理后,对 SDRAM 控制器的设置过程就容易理解一些了。

以下的设置代码全部引用自 RedBoot:

```
1 Setup_SDRAM
2     LDR        r11,=0xc0000000    ; Physic Addr of SDRAM in EDB9315A
3     ;*****
4     ; Set 32-bit wide configuration of SDRAM.
5     ;*****
6     LDR        r0,=0x00210028
7     LDR        r1,=0x00400000
8     ORR        r2,r11,#0x00008800
9     LDR        r3,=0x80060000
10    STR        r0,[r3,#0x0010]
11
12    ;*****
13    ; Set the Initialize and MRS bits
14    ;*****
15    LDR        r4,=0x80000003
16    STR        r4,[r3,#0x0004]
17
18    ;*****
19    ; Delay for 200us.
20    ;*****
21    MOV        r4,#0x3000
22 Delay1
23    SUBS        r4,r4,#1
24    BNE        Delay1
```

```

25 ;*****
26 ; Clear the MRS bit to issue a precharge all.
27 ;*****
28 LDR      r4, =0x80000001
29 STR      r4, [r3, #0x0004]
30 ;*****
31 ; Temporarily set the refresh timer to 0x10.
32 ; Make it really low so that refresh cycles are generated.
33 ;*****
34 LDR      r4, =0x10
35 STR      r4, [r3, #0x0008]
36 ;*****
37 ; Delay for at least 80 SDRAM clock cycles.
38 ;*****
39 MOV      r4, #80
40 Delay2
41 SUBS     r4, r4, #1
42 BNE      Delay2
43 ;*****
44 ; Set the refresh timer to the fastest required
45 ; for any device that might be used.
46 ;*****
47 LDR      r4, =0x01e0
48 STR      r4, [r3, #0x0008]
49 ;*****
50 ; Select mode register update mode.
51 ;*****
52 LDR      r4, =0x80000002
53 STR      r4, [r3, #0x0004]
54 ;*****
55 ; Program the mode register on the SDRAM.
56 ;*****
57 LDR      r4, [r2]
58 ;*****
59 ; Select normal operating mode.
60 ;*****
61 LDR      r4, =0x80000000
62 STR      r4, [r3, #0x0004]

```



```

64 ;*****
65 ; Determine the size of the SDRAM.
66 ;*****
67 MOV      r0, r11
68 BL      SDRAMSize
69
70 ;*****
71 ; Save the SDRAM characteristics.
72 ;*****
73 MOV      r8, r0
74 MOV      r9, r1
75 MOV      r10, r2
76
77 ;*****
78 ; Compute an appropriate refresh rate based on the memory size.
79 ; It may be possible to refresh less often for particular SDRAMs,
80 ; but these values are appropriate for the majority of SDRAMs.
81 ;*****
82 LDR      r0, =0x80060000      ; EDB93XX_SDRAM CTRL
83 Mul      r1, r8, r10
84 LDR      r2, [r0, #0x0010]
85 TST      r2, #0x00000004
86 MOVEQ    r1, r1, lsr #1
87 CMP      r1, #0x02000000
88 MOVLT    r1, #0x0600
89 MOVGE    r1, #0x2f0
90 STR      r1, [r0, #0x0008]
91 ;//////////
92 ; End of Setup_SDRAM
93 ;//////////

```

源代码 3.4.1

其中 SDRAMSize 是子程序，功能为测试 SDRAM 的大小，代码如下：

```

1 ;*****
2 ; Subroutine: SDRAMSize
3 ; Determine the size of the SDRAM.
4 ; Use data=address for the scan.
5 ;*****
6 SDRAMSize
7     ;
8     ; Store zero at offset zero.
9     ;
10    STR    r0, [r0]
11
12    ;
13    ; Start checking for an alias at 1MB into SDRAM.
14    ;
15    LDR    r1, =0x00100000
16
17    ;
18    ; Store the offset at the current offset.
19    ;
20    check_block_size
21    STR    r1, [r0, r1]
22
23    ;
24    ; Read back from zero.
25    ;
26    LDR    r2, [r0]
27
28    ;
29    ; Stop searching of an alias was found.
30    ;
31    CMP    r1, r2
32    BEQ    found_block_size
33
34    ;
35    ; Advance to the next power of two boundary.
36    ;
37    MOV    r1, r1, lsl #1
38
39    ;
40    ; Loop back if the size has not reached 256MB.
41    ;
42    CMP    r1, #0x10000000
43    BNE    check_block_size

```

```

44     ;
45     ; A full 256MB of memory was found, so return it now.
46     ;
47     LDR    r0, =0x10000000
48     LDR    r1, =0x00000000
49     LDR    r2, =0x00000001
50     MOV    pc, lr
51
52     ;
53     ; An alias was found. See if the first block is 128MB in size.
54     ;
55 found_block_size
56     CMP    r1, #0x08000000
57
58     ;
59     ; The first block is 128MB, so there is no further memory. Return it
60     ; now.
61     LDREQ   r0, =0x08000000
62     LDREQ   r1, =0x00000000
63     LDREQ   r2, =0x00000001
64     MOVEQ   pc, lr
65
66     ;
67     ; Save the block size, set the block address bits to zero, and initialize
68     ; the block count to one.
69     ;
70     MOV    r3, r1
71     LDR    r4, =0x00000000
72     LDR    r5, =0x00000001
73
74     ;
75     ; Look for additional blocks of memory by searching for non-aliases.
76     ; Store zero back to address zero.
77     ;
78 find_blocks
79     STR    r0, [r0]
80
81     ;
82     ; Advance to the next power of two boundary.
83     ;
84     MOV    r1, r1, lsl #1

```

```

85     ;
86     ; Store the offset at the current offset.
87     ;
88     STR    r1, [r0, r1]
89
90     ;
91     ; Read back from zero.
92     ;
93     LDR    r2, [r0]
94
95     ;
96     ; See if a non-alias was found.
97     ;
98     CMP    r1, r2
99     ;
100    ; If a non-alias was found, then or in the block address bit and
101    ; multiply the block count by two (since there are two unique
102    ; blocks, one with this bit zero and one with it one).
103    ;
104    ORRNE   r4, r4, r1
105    MOVNE   r5, r5, lsl #1
106    ;
107    ; Continue searching if there are more address bits to check.
108    ;
109    CMP    r1, #0x08000000
110    BNE    find_blocks
111    ;
112    ; Return the block size, address mask, and count.
113    ;
114    MOV    r0, r3
115    MOV    r1, r4
116    MOV    r2, r5
117    ;
118    ; Return to the caller.
119    ;
120    MOV    pc, lr
121    ;////////////////////
122    ; End of Subroutine:
123    ; SDRAMSize
124    ;////////////////////

```

源代码 3.4.2

当设置完成后，我们要对 SDRAM 进行测试，以验证配置是否成功。测试的思路很简单，就是向 SDRAM 的某些地址写入 32-bit 的数据，然后再读出这些数据，如果内容相符，则说明 SDRAM 已经可以正常使用了。我们已经知道了开发板上 SDRAM 的物理地址的范围，为了程序简单起见，只选择四个地址：0xC0000000，0xC1FFFFFFC，0xC4000000 和 0xC5FFFFFFC 进行测试，具体代码如下：

```
1      ;*****
2      ; Test SDRAM
3      ;*****
4      LDR      r0, =0x55aaaa55
5      LDR      r11, =0xc0000000
6      STR      r0, [r11,#0]
7      LDR      r1, [r11,#0]
8      CMP      r0, r1
9      BNE      SDRAM_ERROR
10
11     LDR      r11, =0xc1ffffffc
12     STR      r0, [r11, #0]
13     STR      r0, [r11,#0]
14     LDR      r1, [r11,#0]
15     CMP      r0, r1
16     BNE      SDRAM_ERROR
17
18     LDR      r11, =0xc4000000
19     STR      r0, [r11, #0]
20     STR      r0, [r11,#0]
21     LDR      r1, [r11,#0]
22     CMP      r0, r1
23     BNE      SDRAM_ERROR
24
25     LDR      r11, =0xc5ffffffc
26     STR      r0, [r11, #0]
27     STR      r0, [r11,#0]
28     LDR      r1, [r11,#0]
29     CMP      r0, r1
30     BNE      SDRAM_ERROR
```

源代码 3.4.3

3.5 串口初始化

串口在嵌入式开发中起着重要的作用。串口的软硬件实现相对其它通信方式来说非常的简单。而网络或 USB 等接口则需要复杂的硬件及软件栈的实现。

EP9315 的串口设置详细信息请参考 [《EP93xx Users Guide UM1.pdf》](#) 第 14 章。

这里着重谈一下串口的时钟源设置。串口的时钟信号由外部晶体产生的 14.7456MHz 信号直接分频产生（[《EP93xx Users Guide UM1.pdf》](#) 第 5 章），可以用下图来理解：

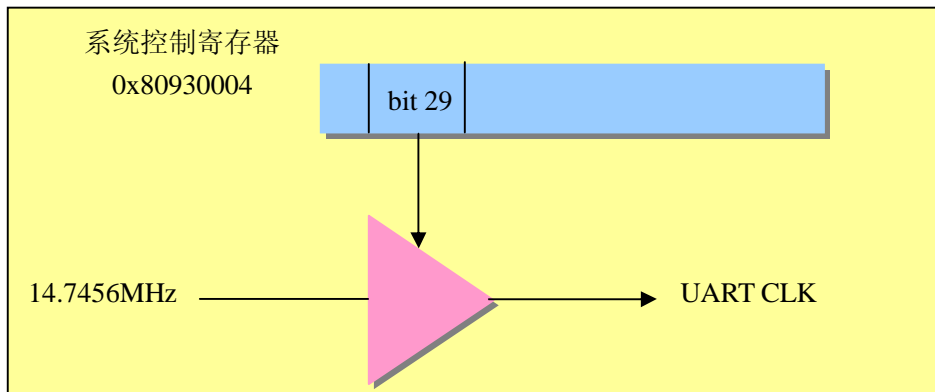


图 3.1

系统控制寄存器 0x80930004 的第 29 位，也被称为 UARTBAUD（波特率控制）位。

[《EP93xx Users Guide UM1.pdf》](#) 第 142 页：

UARTBAUD:

This bit controls the clock input to the UARTs. When cleared, the UARTs are driven by the 14.7456MHz clock divided by 2 (7.3728MHz). This gives a maximum baud rate of 230Kbps. When set, the UARTs are driven by the 14.7456MHz clock directly, giving an increased maximum baud rate of 460Kbps. **This bit is 0 on reset.**

大概的意思就是，当该位为 0 时，串口的时钟源为 14.7456MHz 的一半：7.3728MHz。当该位为 1 时，串口的时钟源为 14.7456MHz（不分频）。系统复位后，该位自动被设置为 0。因此在设置串口波特率之前，我们要知道该位的设置情况。而且还有一点要了解的就是，ARM Linux 内核启动后，默认该位为 1，所以我们在初始化时就要将该位设置为 1，否则 ARM Linux 内核设置的波特率就会出错。

ARM Linux 内核默认将串口波特率设置为 57600，其实是因为 EP9315 的外接晶体产生的时钟频率为 14.7456MHz，波特率设为 57600 时产生的误差最小（具体请参考 [《EP93xx Users Guide UM1.pdf》](#) 第 534 页）。因此我们在初始化时也将波特率设为 57600（8 位数据位，一位停止位，无校验位）。

串口初始化代码如下：


```

1 Setup_UART1
2 ;*****
3 ; 设置 UART 的时钟源
4 ;*****
5 LDR      r0, =0x80930004
6 LDR      r1, =0x20000000
7 LDR      r2, [r0, #0]
8 ORR      r1, r1, r2
9 STR      r1, [r0, #0]
10
11 LDR      r0, =0x80930000
12 MOV      r1, #0xaa
13 STR      r1, [r0, #0xc0]
14 MOV      r1, #0x40000
15 STR      r1, [r0, #0x80]          ; UART1 baud rate clock is active.
16
17 ;*****
18 ; 初始化 UART1
19 ;*****
20 LDR      r12, =0x808c0000
21 MOV      r1, #0
22 STR      r1, [r12, #4]          ; UART1 Receive Status Register
23 STR      r1, [r12, #0xc]       ; Line Control Register - Middle Byte
24 MOV      r1, #0x2e
25 STR      r1, [r12, #0x10]      ; Line Control Register - Low Byte
26 MOV      r1, #0x60
27 STR      r1, [r12, #8]        ; Line Control Register - High Byte
28 MOV      r1, #3
29 STR      r1, [r12, #0x100]     ; UART1 Modem Control Register
30 MOV      r1, #1
31 STR      r1, [r12, #0x14]     ; UART1 Control Register
32
33 SetBaud
34 LDR      r3, =0x808c0000
35
36 ;*****
37 ; Set Baud 57600
38 ;*****
39 MOV      r1, #15
40 STR      r1, [r3, #0x10]

```

```

41 ;*****
42 ; Clear the upper baud divisor register
43 ;*****
44 MOV     r1, #0
45 STR     r1, [r3, #0xc]
46
47 ;*****
48 ; Set the data length to 8 bits per character
49 ; and enable the FIFO. Set the baud rate divider.
50 ; Set the mode to N-8-1
51 ;*****
52 MOV     r1, #0x70
53 STR     r1, [r3, #0x8]
54 ;////////////////////////////////////
55 ; End of Setup_UART1
56 ;////////////////////////////////////

```

源代码 3.5.1

串口初始化完毕之后，我们就可以发出字符串，表示串口已经可以正常工作了。下面是发送单个字符的子程序：

```

1 ;*****
2 ; Subroutine: SendChar
3 ; Sends a character to UART1.
4 ; The character is stored in r0.
5 ;*****
6 SendChar
7     LDR     r1, =0x808c0000    ; base address of the UART 1 registers.
8     ;*****
9     ; Wait until the FIFO is empty.
10    ;*****
11    MOV     r3, #0x80
12 Send_wait
13    LDR     r2, [r1, #0x18]
14    TST     r2, r3
15    BEQ     Send_wait
16    ;*****
17    ; Write the character to UART1.
18    ;*****
19    STR     r0, [r1, #0]

```

```
20 ; *****
21 ; Return to the caller.
22 ; *****
23 MOV    pc, lr
24
25 ;////////////////////
26 ; End of Subroutine:
27 ; SendChar
28 ;////////////////////
```

源代码 3.5.2

3.6 硬件初始化小结

写到这里，我们再回过头去看看图 1.1，会发现红色方框标出的部分，其实就是为了满足 ARM Linux 内核启动所需要设置的硬件。我们可以把以上这些初始化的“片断”串连起来，组成一个完整的硬件初始化程序。关于程序的设置和运行的方法请见第 2 章的介绍。而且现在已经不只限于使用 LED，还可以输出字符串，用串口来显示调试信息了。

第 4 章 软件初始化

4.1 将程序搬运到 SDRAM 中去运行

几乎所有的 BootLoader 都会把启动的过程分为两个部分：

- I 第一部分就是我们在第 3 章里所介绍的硬件初始化，这部分完全用汇编言语来实现（因为 C 言语编译器无法产生一些涉及到 ARM CPU 特权级的指令），并且是在“FLASH 中运行”完成。
- I 第二部分我把它称为“软件初始化”，这么叫只是为了和第一部分有所区别，并没有什么严格的定义。这部分可以用 C 语言来实现，而且代码通常是在 SDRAM 中来运行的。

那么这里又出涉及到了另外两个问题：

- I 在“FLASH 中运行”的程序要将 FLASH 中的部分代码搬运（拷贝）到 SDRAM 中去。
- I 然后跳转到 SDRAM 中去运行。

具体来说就是，当汇编语言编写的这部分代码运行完毕后，需要将 C 语言编写的那部分代码拷贝到 SDRAM 中去，然后跳转到 SDRAM 中去执行 C 语言编写的那部分内容。

我们在 CodeWarrior 中新建一个工程：“startup”，添加两个文件：

startup.s（汇编语言程序）和 startup_main.c（C 语言程序）。如下图所示：

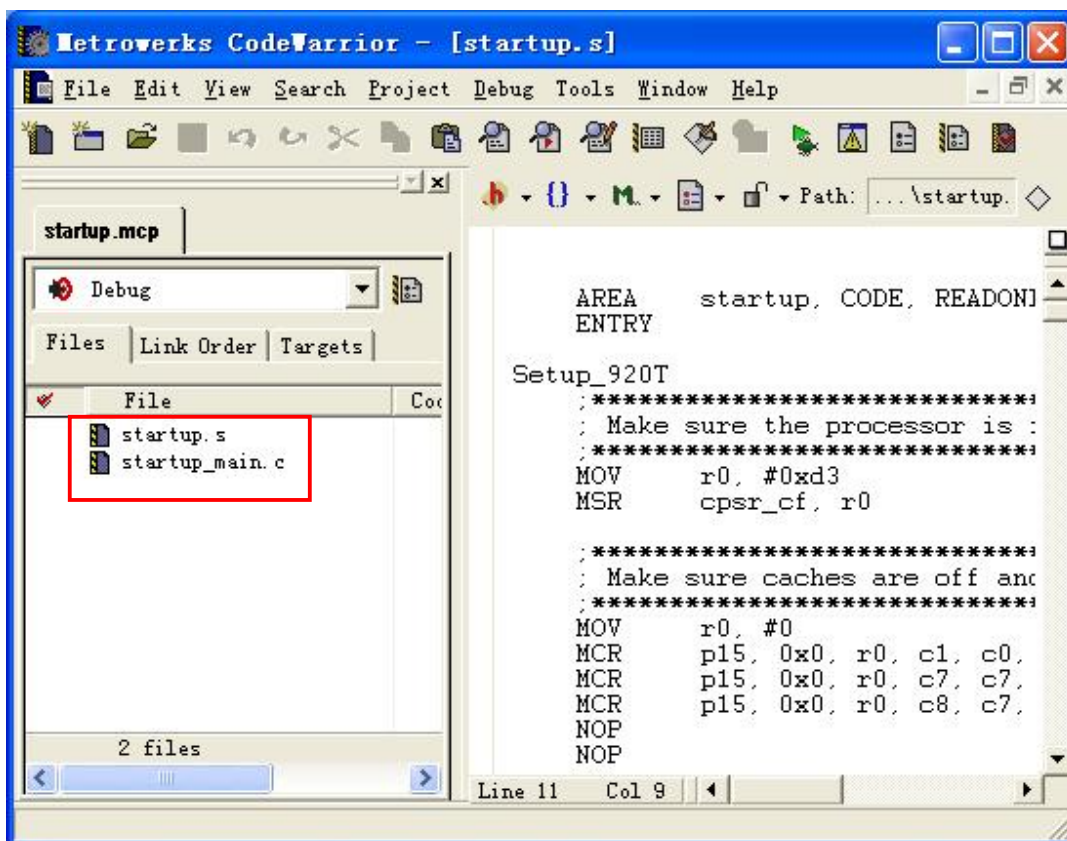


图 4.1

工程的设置请参看第 2 章的内容，这里也假设程序是从 FLASH 中运行的 (0x60000000)。

此外，这里还要将“RW Base”设置为：0xC0000000，如下图所示：

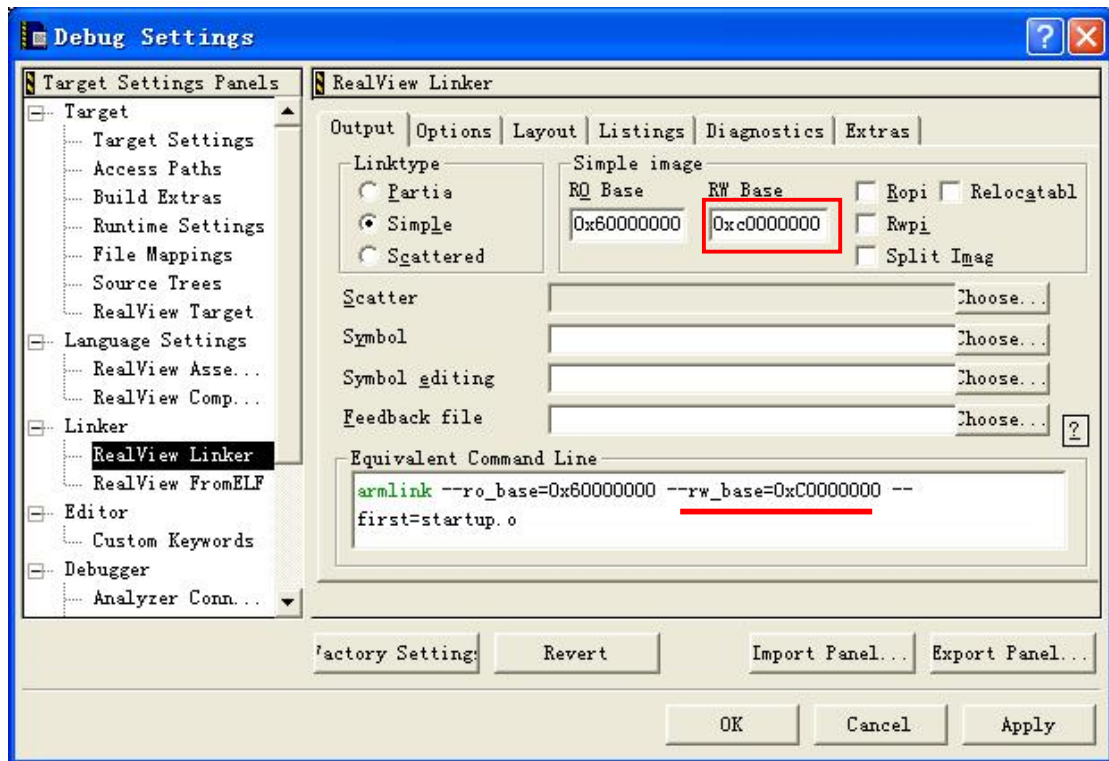


图 4.2

在前面硬件初始化时，我们没有强调“RW Base”的设置问题。“RW Base”表示“Read/Write Base Address”，可以理解为：这里指定的一块内存是用来存放和修改程序运行过程中所用到的数据结构和变量，其实就是一块缓存。我们将该地址设置到 SDRAM 的开头部分 (0xC0000000)。而在硬件初始化的过程中，SDRAM 控制器可能还未设置完成，所以指定一块内存给程序使用反而会出现隐患。因此硬件初始化的代码往往都是在 FLASH 中运行，并且不使用复杂的数据结构和大量的中间变量，这样代码在运行过程中就不需要使用 SDRAM 作为缓存。

当硬件初始化结束后，进行软件初始化时，就需要使用复杂的数据结构和大量的中间变量，这时编译器就可以利用 SDRAM 来作为缓存使用了。

汇编程序 `startup.s` 中的内容就是第 3 章介绍的硬件初始化部分。而 `startup_main.c` 中的内容就是我们在 SDRAM 中运行的部分。

这里我们举个简单的例子：在 C 程序中向串口输出一个字符串，表示程序已经在 SDRAM 中正确运行了。具体代码如下：

```

1 // 文件名: startup_main.c
2 #define UART1_PORT          0x808c0000 // UART1 数据寄存器
3 #define UART1_FLAG         0x808c0018 // UART1 标志寄存器
4
5 // 函数声明
6 void Send_char(unsigned char);
7 void Send_string(unsigned char*);
8
9 /*****
10 * 主程序
11 *****/
12 void startup_main(void)
13 {
14     Send_string("I'm running in SDRAM now!\n\r");
15 }
16 /*****
17 * 函数定义: Send_char
18 * 向 UART1 发送一个字符
19 *****/
20 void Send_char(unsigned char ch)
21 {
22     // waiting for FIFO empty
23     while((*(unsigned char*)UART1_FLAG & 0x80) == 0);
24     // then send data to uart port
25     *(unsigned char*)UART1_PORT = ch;
26 }
27 /*****
28 * 函数定义: Send_String
29 * 向 UART1 发送字符串
30 *****/
31 void Send_string(unsigned char *str)
32 {
33     while(*str != 0)
34         Send_char(*str++);
35 }
36 // End of File: startup_main.c

```

源代码 4.1.1

为了使这段代码能够在 SDRAM 中运行，我们要将该部分编译链接后生成的二进制代码从 FLASH 拷贝到 SDRAM 中去。这需要在汇编程序中增加下面这段代码：

```
1 ;*****
2 ; Copy code to SDRAM
3 ;*****
4 Copy_to_sdram
5     MOV     r6, #1012           ; 要拷贝的代码长度
6     LDR     r1, =0xC0002000    ; 目的地址
7     ADR     r3, Call_CFunction ; 源地址
8
9 Copy_loop
10    ;*****
11    ; Read 4 bytes from FLASH
12    ;*****
13    LDR     r0, [r3]
14    ADD     r3, r3, #4
15    ;*****
16    ; Write 4 bytes to SDRAM
17    ;*****
18    STR     r0, [r1]
19    ADD     r1, r1, #4
20    ;*****
21    ; Decrement the count of 4 bytes
22    ;*****
23    SUBS    r6, r6, #4
24    BNE     Copy_loop
25    ;////////////////////
26    ; End of Copy_to_sdram
27    ;////////////////////
28
29 ;*****
30 ; go to SDRAM
31 ;*****
32     LDR     r13, =0xC0003FFC   ; 将堆栈指针设在 16K 字节处
33     LDR     r5, =0xC0002000
34     MOV     pc, r5
```

源代码 4.1.2

这段代码很简单，就是将指定长度的 FLASH 中的代码，拷贝到起始地址为 0xC0002000 的 SDRAM 中，然后修改 pc，跳转到 SDRAM 中去运行。但是代码的长度该如何确定呢？一个比较直接（或者说“比较笨”）的办法就是去查看项目编译完成后生成的“startup.bin”文件的大小。因为这个二进制文件是由两部分组成的（startup.o + starup_main.o。当硬件初

始化的那部分汇编程序固定后，startup.o 的长度就不会发生变化。而我们修改 C 程序时，就只是 startup_main.o 这部分在发生变化了），所以我们就把拷贝的长度设为该文件的大小，这样做的目的是防止出现拷贝的代码不完整的问题（虽然这样做会多拷贝了一些无用的数据，但这并不会影响 C 程序的运行，所以我选择了“宁滥勿缺”）。

我们没有把代码拷贝到 SDRAM 的开头，而是放在了 0xC0002000 处，这是因为我们将来会把一些 ARM Linux 内核启动所需要的参数放在 SDRAM 的开头部分，并且前面也说过，我们将 SDRAM 的开头部分设置成了程序运行时的缓存。

在上面程序的第 7 行，给出了开始拷贝的地址：Call_CFunction，那么把 Call_CFunction 放在哪里合适呢？从以上的分析可以看出，其实把 Call_CFunction 放在哪里都可以（但要在 startup.s 内部，不能定义到 C 程序中去），Call_CFunction 只做了一件事，就是跳转到 C 函数去运行。为了拷贝的无用数据尽可能地少，我们把 Call_CFunction 放在程序的最后，关键字“END”之前。具体代码如下：

```
1      PRESERVE8
2      IMPORT      startup_main
3      ;*****
4      ; 跳转到 C 函数运行
5      ;*****
6      Call_CFunction
7      BL          startup_main
8
9      END
10     ;//////////
11     ;End of startup.s
12     ;//////////
```

源代码 4.1.3

说了这么多，其实要点总结起来很简单：就是要严格区分清楚哪部分代码是在 FLASH 中运行的，哪部分代码是在 SDRAM 中运行。

下面来看一下对该工程的“RealView Linker”的设置：

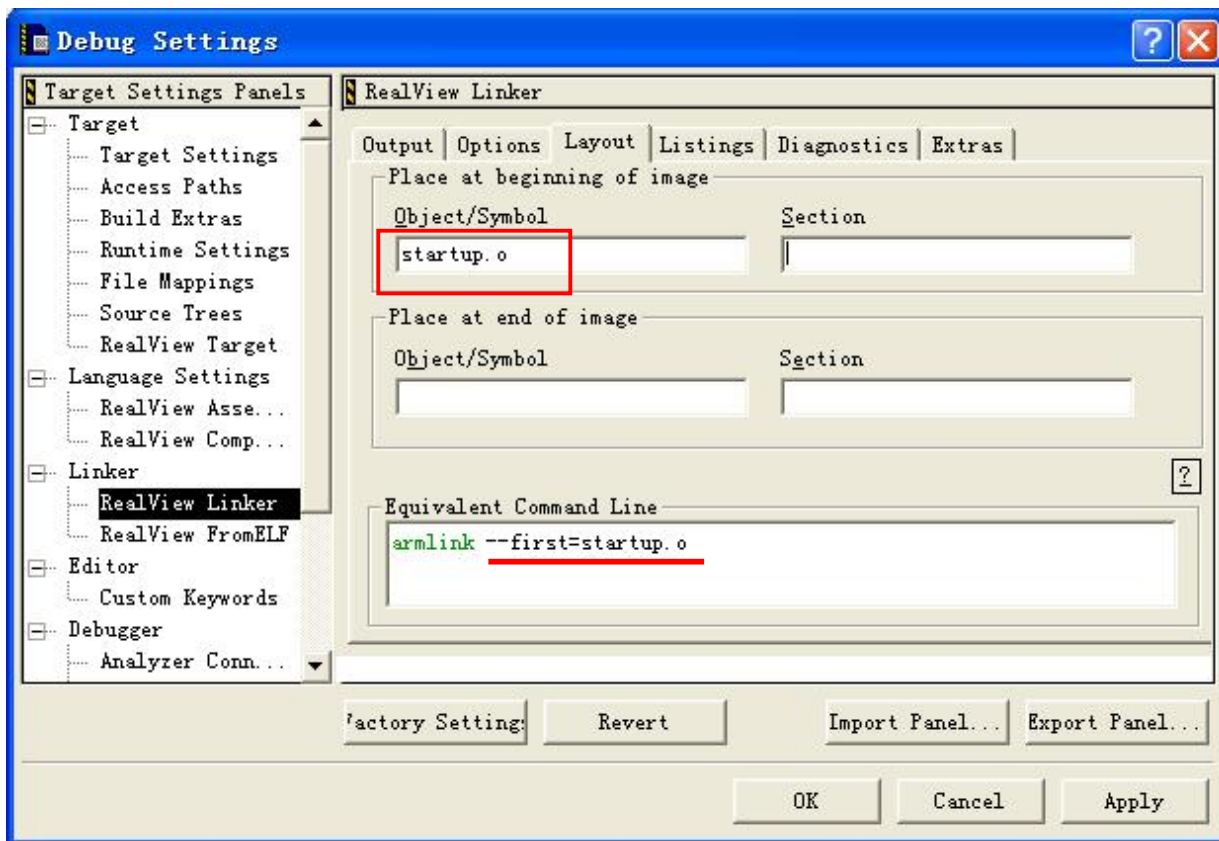


图 4.3

这个设置是要求链接工具生成二进制代码时，将汇编程序的部分放在 C 程序之前。

4.2 设置内核标记列表

现在再来复习一下在第 3 章介绍过的，BootLoader 要做的工作：

1. Setup and initialise the RAM. 设置和初始化 RAM
2. Initialise one serial port. 初始化一个串口
3. Detect the machine type. 检测机器的类型（我们已经知道本开发板的型号）
4. Setup the kernel tagged list. 设置内核标记列表
5. Call the kernel image. 调用（启动）内核镜像

蓝色表示目前已经完成的的部分。现在就来介绍一下“内核标记列表”（kernel tagged list）。

我们还是从 ARM Linux 的官方网站的文档看起：

<http://www.arm.linux.org.uk/developer/booting.php>

现引用其中一段关于“kernel tagged list”的原文如下：

4. Setup the kernel tagged list

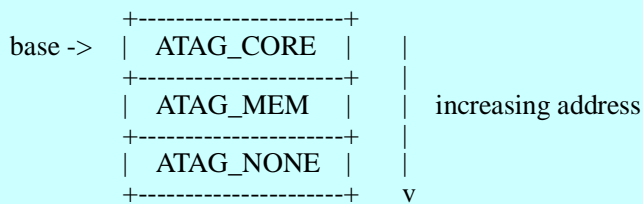
Existing boot loaders: OPTIONAL, HIGHLY RECOMMENDED

New boot loaders: MANDATORY

The boot loader must create and initialise the kernel tagged list. A valid tagged list starts with ATAG_CORE and ends with ATAG_NONE. The ATAG_CORE tag may or may not be empty. An empty ATAG_CORE tag has the size field set to '2' (0x00000002). The ATAG_NONE must set the size field to zero.

Any number of tags can be placed in the list. It is undefined whether a repeated tag appends to the information carried by the previous tag, or whether it replaces the information in its entirety; some tags behave as the former, others the latter.

The boot loader must pass at a minimum the size and location of the system memory, and root filesystem location. Therefore, the minimum tagged list should look:



The tagged list should be stored in system RAM.

The tagged list must be placed in a region of memory where neither the kernel decompressor nor initrd 'bootp' program will overwrite it. The recommended placement is in the first 16KiB of RAM.

其中主要的意思是说：BootLoader 最少要向 ARM Linux 内核提供三块 TAG：ATAG_CORE, ATAG_MEM 和 ATAG_NONE；而且它们必须放在系统 RAM (SDRAM) 中；并且在 ARM Linux 内核启动的过程中，它们不能被覆盖；建议将它们放在系统内存的前 16K 字节的范围内。

所谓“内核标记列表”，就是 BootLoader 提供给 ARM Linux 内核的系统硬件信息。就是 BootLoader 所收集到的，而在 ARM Linux 内核启动时必须知道的信息。其中主要包括：

- I ATAG_MEM, 系统内存信息：内存的大小，物理地址。如果内存不连续，就要分别标出各个分块的起始物理地址和大小。
- I ATAG_INITRD, RAMDISK: ARM Linux 的根文件系统在内存中的起始物理地址和大小。
- I ATAG_CMDLINE, 命令行参数：ARM Linux 内核启动时所需要的命令行参数。

除此之外还有其它很多可选的信息，但只要能够正确提供以上几条信息，ARM Linux 内核就可以正常启动了。

有关“内核标记列表” (kernel tagged list) 更全面的介绍，请查看 ARM Linux 内核源代码树下的“Documentation”目录下的“arm”子目录中的文档。在我使用的内核版本中为：<kernel/linux-2.6.20.4/Documentation/arm>

而用 C 语言来实现这些信息的传递，就是初始化一些数据结构，并把它们放到内存的适当位置上，然后设置其中各种数据成员的参数。

有关“内核标记列表”数据结构的完整定义，请查看 ARM Linux 内核源代码树下的源文件：<kernel/linux-2.6.20.4/include/asm-arm/setup.h>

请仔细阅读这些“原始文档和源代码”，因为这些都是正在各种硬件中运行的，经过了实践检验的东西。很多时候我们可能会“舍本逐末”地去寻找专家或资料，希望能得到问题的答案，殊不知真正的答案就在这些“原始文档和源代码”当中。

先来看看下面这几个数据结构：

```
1 struct tag {
2     struct tag_header hdr;
3     union {
4         struct tag_core core;
5         struct tag_mem32 mem;
6         struct tag_videotext videotext;
7         struct tag_ramdisk ramdisk;
8         struct tag_initrd initrd;
9         struct tag_serialnr serialnr;
10        struct tag_revision revision;
11        struct tag_videolfb videolfb;
12        struct tag_cmdline cmdline;
13        /*
14         * Acorn specific
15         */
16        struct tag_acorn acorn;
17        /*
18         * DC21285 specific
19         */
20        struct tag_memclk memclk;
21    } u;
22 };
```

源代码 4.2.1

```
1 struct tag_header {
2     __u32 size;
3     __u32 tag;
4 };
```

源代码 4.2.2

```
1 /* it is allowed to have multiple ATAG_MEM nodes */
2 struct tag_mem32 {
3     __u32 size;
4     __u32 start; /* physical start address */
5 };
```

源代码 4.2.3

```

1 struct tag_initrd {
2     __u32 start; /* physical start address */
3     __u32 size; /* size of compressed ramdisk image in bytes */
4 };

```

源代码 4.2.4

```

1 struct tag_cmdline {
2     char cmdline[1]; /* this is the minimum size */
3 };

```

源代码 4.2.5

```

1 #define ATAG_CORE          0x54410001
2 #define ATAG_MEM          0x54410002
3 #define ATAG_INITRD2     0x54420005
4 #define ATAG_CMDLINE     0x54410009

```

源代码 4.2.6

以上是各个“信息块”的数据结构。

下面来看一下初始化数据结构的具体过程：

```

1 #define BOOT_PARAMS      (0xc0000100)
2
3 static struct tag *params;
4
5 static void setup_start_tag(void)
6 {
7     params = (struct tag *)BOOT_PARAMS;
8
9     params->hdr.tag = ATAG_CORE;
10    params->hdr.size = tag_size(tag_core);
11
12    params->u.core.flags = 0;
13    params->u.core.pagesize = 0;
14    params->u.core.rootdev = 0;
15
16    params = tag_next(params);
17 }

```

源代码 4.2.7

第一个“信息块”必须是 ATAG_CORE。上面代码的第 7 行：将指针 params 指到 SDRAM

地址 0xC0000100 处，这里就是“内核标记列表”的默认起始物理地址，ARM Linux 内核启动时会到这里来读取“内核标记列表”。第 9 行：将标记 ATG_CORE 写到 SDRAM 中，说明了本信息块的类型。第 10 行是计算出本信息块的大小，然后写入 SDRAM。

计算信息块大小是由宏 tag_size () 来完成的：

```
1  #define tag_size(type) \  
2  ((sizeof(struct tag_header) + sizeof(struct type)) >> 2)
```

源代码 4.2.8

需要注意的是，先得到的信息块的大小单位为字节，然后再右移 2 位（除以 4），表明最后是以 4 字节为单位来表示信息块的大小。那么会不会出现因为除不尽而出现误差的情况呢？我们查看一下，发现除了 tag_cmdline，其它的信息块的结构成员的数据类型都是“__u32”，都是以 4 字节为单位的，所以不会出现除不尽的情况。那么就来看一下设置 tag_cmdline 时是如何来处理这个问题的：

```
1  static void setup_commandline_tag(char *commandline)  
2  {  
3      char *p;  
4  
5      /* eat leading white space */  
6      // 去掉参数之前的空格  
7      for(p = commandline; *p == ' '; p++);  
8  
9      /* skip non-existent command lines so the kernel will still  
10     * use its default command line.  
11     */  
12     // 如果没有参数，就返回，使用系统默认的参数  
13     if(*p == '\0')  
14         return;  
15     params->hdr.tag = ATAG_CMDLINE;  
16     params->hdr.size = (sizeof(struct tag_header) + strlen(p) + 1 + 4) >> 2;  
17  
18     strcpy(params->u.cmdline.cmdline, p);  
19  
20     params = tag_next(params);  
21 }
```

源代码 4.2.9

上面代码的第 16 行就是计算信息块 ATAG_CMDLINE 的大小，它使用了函数 strlen()，这个函数比较简单，返回字符串的长度（除了最后的“0”之外），然后加 1，把字符串中最后的“0”补算进来，再加 4，最后再除以 4。其实这样还是可能会出现除不尽的情况，但是因为做了“加 4”这个补救的动作，所以即使出现除不尽的情况，最后取整得到的值也一定大于等于实际的大小。（这里也可以理解为“宁滥勿缺”）

我们要设置的，传递给内核的命令行参数为：

```
1 char commandline[]="root=/dev/ram console=ttyAM";
```

源代码 4.2.10

再来看一下第 20 行的宏定义 `tag_next()`:

```
1 #define tag_next(t) \  
2 ((struct tag *)((__u32 *) (t) + (t)->hdr.size))
```

源代码 4.2.11

这里可以理解为: `params = params + params->hdr.size`, 就是将指针跳过本信息块, 指向紧接本块之后的, 未被使用的内存。这是链表式数据结构常用的方法。

下面介绍 `ATAG_MEM` (系统内存参数) 的设置。在 `U-Boot` 和 `RedBoot` 等通用程序中, 都是使用函数去仔细检测全部的系统内存, 然后再设置相应的参数。我们在已知开发板内存配置的境况下, 就可以“偷一下懒”(其实在硬件初始化时也已做过简单的内存测试), 直接设置两块内存信息参数:

```
1 static void setup_memory_tags(void)  
2 {  
3     // The first MEM block  
4     params->hdr.tag = ATAG_MEM;  
5     params->hdr.size = tag_size(tag_mem32);  
6  
7     params->u.mem.start = 0xC0000000;  
8     params->u.mem.size = 0x02000000; // 32M Bytes  
9  
10    params = tag_next(params);  
11  
12    // The second MEM block  
13    params->hdr.tag = ATAG_MEM;  
14    params->hdr.size = tag_size(tag_mem32);  
15  
16    params->u.mem.start = 0xC4000000;  
17    params->u.mem.size = 0x02000000; // 32M Bytes  
18  
19    params = tag_next(params);  
20 }
```

源代码 4.2.12

接着要设置的信息块就是 `ATAG_INITRD`:


```

1  #define RAMDISK_RAM_BASE    (0xC0A00000)
2  #define INITRD_LEN          (0x00400000)
3
4  static void setup_initrd_tag(void)
5  {
6      // an ATAG_INITRD node tells the kernel where the compressed
7      // ramdisk can be found. ATAG_RDIMG is a better name, actually.
8      //
9      params->hdr.tag = ATAG_INITRD2;
10     params->hdr.size = tag_size(tag_initrd);
11
12     params->u.initrd.start = RAMDISK_RAM_BASE;
13     params->u.initrd.size = INITRD_LEN;
14
15     params = tag_next(params);
16 }

```

源代码 4.2.13

这就是告诉 ARM Linux 内核，RAMDISK 放在了内存的什么地方。我们把 RAMDISK 放在内存物理地址为 0xC0A00000（10M 字节处）的地方，RAMDISK 占用的内存空间为 4M 字节。其实对 RAMDISK 放在哪里并没有硬性规定，但是要注意 RAMDISK 的大小，以免超出了内存有效地址的范围（越界）。这里我们把 RAMDISK 放在内存 10M 字节处，大小为 4M 字节，而这一块内存的大小为 16M 字节，所以没有越界。

最后设置 ATAG_NONE，表示“内核标记列表”到此为止了。

```

1  static void setup_end_tag(void)
2  {
3      params->hdr.tag = ATAG_NONE;
4      params->hdr.size = 0;
5  }

```

源代码 4.2.14

4.3 加载 zImage(内核镜像)和 RAMDISK

一般来说, zImage 和 RAMDISK 是在 Linux 环境下, 通过交叉编译得到的。先说明一下我的开发环境:

WindowsXP SP3 + VMware Workstation7.0 + Ubuntu8.10

EDB9315A 的 ARM Linux 开发环境的设置请参考: <http://arm.cirrus.com/>

其中要说明的是, 在 VMware Workstation7.0 + Ubuntu8.10 环境下, 当设置完交叉编译环境, 对 ARM Linux 进行编译成功后, 可能会造成重新启动 Ubuntu8.10 时无法进入系统的情况。我的解决办法是, 每次交叉编译完成后, 执行以下一条命令:

```
sudo chmod 777 /tmp
```

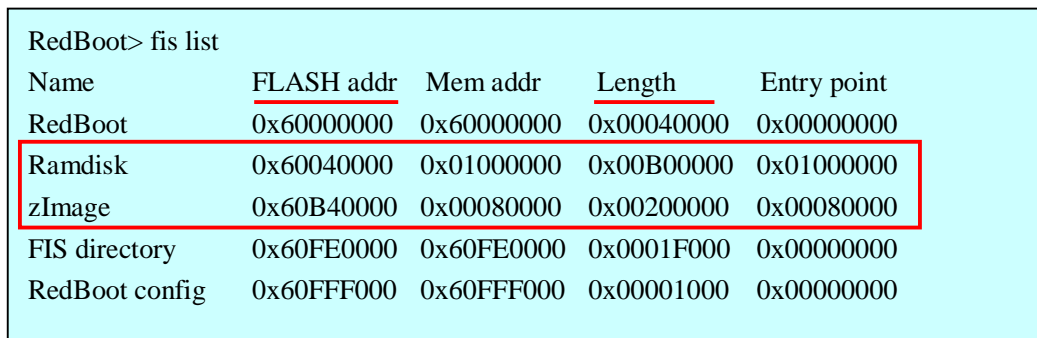
就是重新设置 tmp 目录的属性。(VMware 的 vmtools 存放了很多重要程序在 tmp 目录下, 而交叉编译时可能使用并修改了 tmp 目录的属性, 造成了下次启动时 vmtools 无法正常使用)

4.3.1 将 zImage 和 RAMDISK 下载到 FLASH 中

一般通用的 BootLoader 都是利用网络接口将 zImage 和 RAMDISK 下载到内存, 然后再从内存写入 FLASH 中。这样做是因为这两个文件太大, 用串口下载太费时间(我估算了一下, EDB9315A 的 RAMDISK 如果用串口下载到内存, 波特率设为 57600 时, 大概需要 15 分钟左右)。

而我选择使用 J-Flash ARM, 通过 JTAG 接口来下载 zImage 和 RAMDISK。

还有一个问题就是, 这两个文件要放到 FLASH 中的什么地方才合适呢? 我们可以参考一下 RedBoot 是怎么做的:



Name	FLASH addr	Mem addr	Length	Entry point
RedBoot	0x60000000	0x60000000	0x00040000	0x00000000
Ramdisk	0x60040000	0x01000000	0x00B00000	0x01000000
zImage	0x60B40000	0x00080000	0x00200000	0x00080000
FIS directory	0x60FE0000	0x60FE0000	0x0001F000	0x00000000
RedBoot config	0x60FFF000	0x60FFF000	0x00001000	0x00000000

图 4.4

在 RedBoot 下查看 FLASH 的使用情况, 可以看到 RAMDISK 放在 0x60040000 处, 长度为 0x00B00000 (11M 字节) (这个文件很大, 我在重新编译时去掉了视频部分, 生成的 RAMDISK 大小不到 4M), 之后紧接着存放 zImage, 长度为 0x00200000 (2M 字节)。

因此首先要知道自己编译得到的这两个文件的大小, 确认它们在 FLASH 中的物理起始地址和所占的空间。我编译得到的两个文件分别是 (文件扩展名设为 “bin”, 是因为 J-Flash ARM 打开时要判断文件的扩展名):

```
zImage.bin          1.77 MBytes
ramdisk.gz.bin      3.35 MBytes
```

以此为根据而制定的存放方案如下: 将 ramdisk.gz.bin 放在 FLASH 的物理起始地址 0x60040000 处, 占用空间为 4M 字节; 将 zImage.bin 放在 FLASH 的物理起始地址 0x60440000 处 (紧接着 ramdisk.gz.bin 存放), 占用空间为 2M 字节。

然后运行 J-Flash ARM 进行设置:

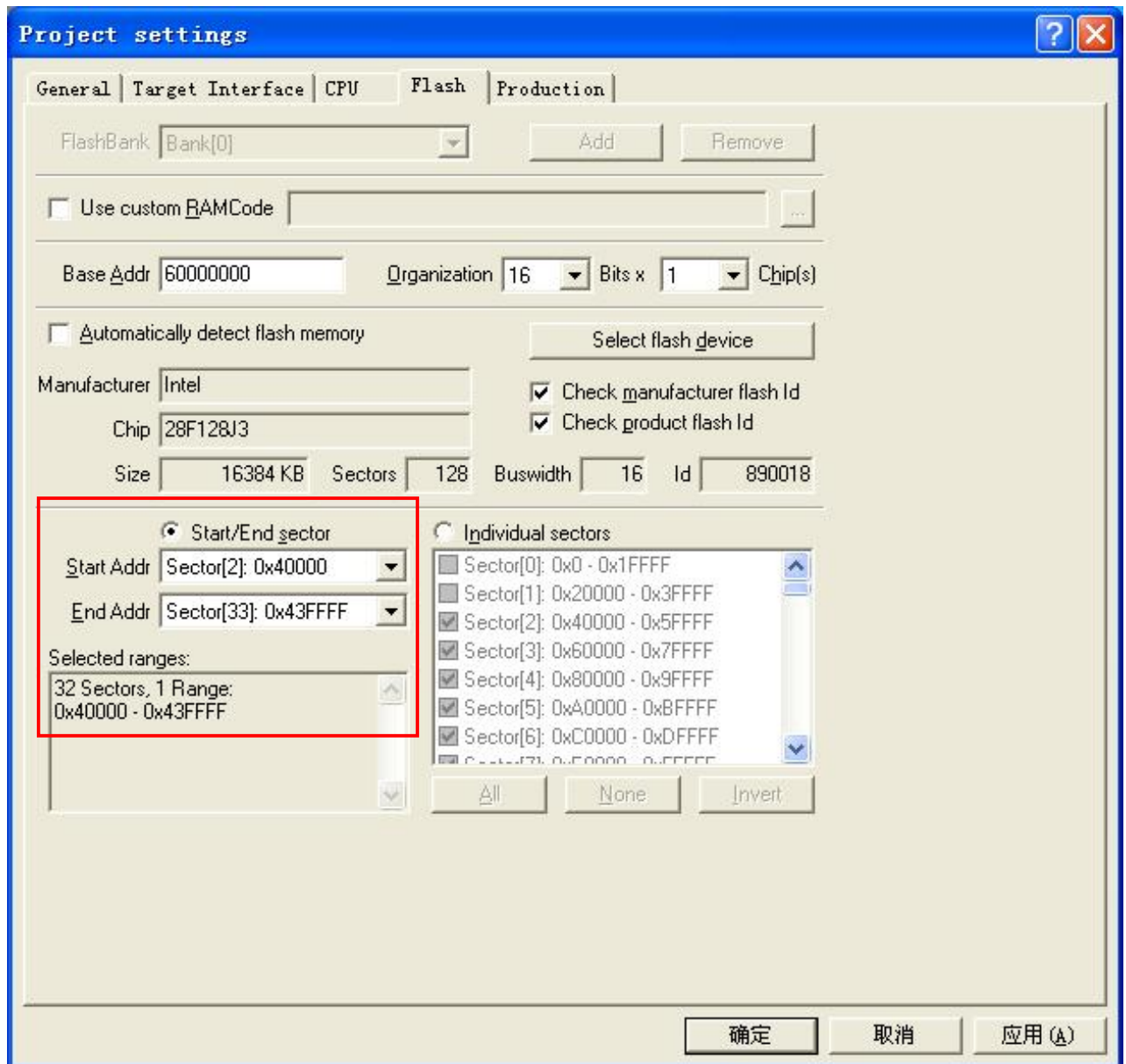


图 4.5

整个 FLASH 的 16M 字节其实是分成了 128 个 Sector (块), 编程时是按 Sector 为单位进行操作的。每个 Sector 大小为 128K 字节。在 J-Flash ARM 中设置 RAMDISK 所要占用的 Sectors, 如上图所示, 请注意开始和结束地址, 共选中了 32 个 Sector, 正好是 4M 字节。设置完成后就可以读入文件 ramdisk.gz.bin, 然后对 FLASH 中的这些 Sectors 进行擦除和编程了。

对 zImage.bin 编程的所做的设置见下图:

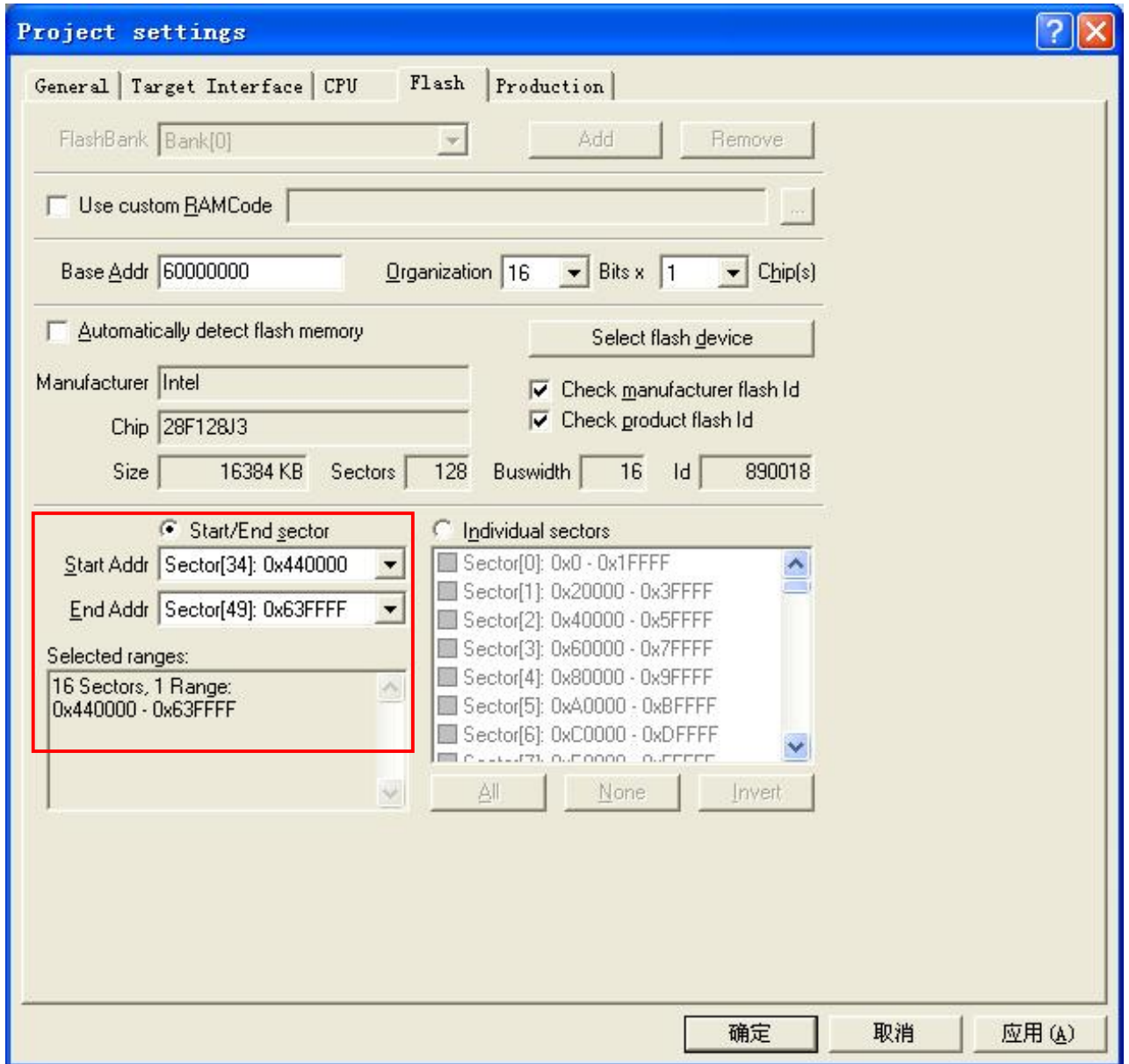


图 4.6

关于 J-Flash ARM 的其它设置请参看第 2 章所做的介绍。J-Flash ARM 的下载速度很快，完全可以在进行批量生产时作为 FLASH 的烧写工具使用。

4.3.2 将 zImage 和 RAMDISK 装入 SDRAM 中

和前面的动作相反，我们现在又要把 zImage 和 RAMDISK 从 FLASH 中给读出来，放到内存中去。我们现在已经知道了这两个文件在 FLASH 中的位置和大小，现在需要知道的就是，应该把这两个文件分别加载（Load）到内存中的什么地方去？查看一下前面的[源代码 4.2.13](#)，就可以知道要将 RAMDISK 加载到物理地址为：**0xC0A00000** 处。那么 zImage 呢？其实内核镜像（zImage）是有默认的内存加载地址的，就是：**0xC0008000**。我们就把 zImage 加载到这里去。当然，你也可以将 zImage 放到其它的地方，只要不越界和与 RAMDISK 冲突就可以了。关于装入 SDRAM 的这段程序很简单，这里就不介绍了，完整的代码见附件中的程序压缩包。

4.4 最后一跳！

最后要做的工作就是跳转到 zImage，启动 ARM Linux 内核。

```
1 #define KERNEL_RAM_BASE      (0xC0008000)
2 void (*theKernel)(int zero,int arch,int param) = (void (*)(int, int,int))KERNEL_RAM_BASE;
```

源代码 4.4.1

这里定义了一个函数指针，并将指针指向 KERNEL_RAM_BASE，就是 ARM Linux 内核在内存中的地址（**把内核看作一个函数**）。然后在程序中以调用函数的方式将 CPU 的控制权交给 ARM Linux 内核。

```
1 #define ARCH_NUMBER          772
2
3 theKernel(0, ARCH_NUMBER, BOOT_PARAMS);
```

源代码 4.4.2

要注意的就是函数传递参数的方式。在默认情况下，C 编译器用寄存器 R0 传递参数 1，R1 传递参数 2，R2 传递参数 3。这正好满足了“[ARM Linux Kernel Boot Requirements](#)”文档中的要求。（其实第 3 个参数 BOOT_PARAMS 并不是必须的，我尝试过不加或者用其它的值来替代都不影响内核的启动，这说明 ARM Linux 内核在启动时会先使用默认参数，就是说 BOOT_PARAMS 有一个系统默认的值，那就是：RAM_BASE+0x100。只有当默认参数无效时，内核才会到指定参数提供的地方去查找信息块）