ARM 映象文件及执行机理

ARM 学习报告 001 2004-5-7

这几天为了弄清楚 ARM 系统是如何运行相应的可执行程序可谓费尽心机，整个五一假期都没有休息，其中由于烧写 flash 的软件出了些问题，使得理解 ARM 映象文件及执行机理更加曲折，不过还好在自己的努力和论坛上兄弟的帮助下，终于还是成功了。作为进入 ARM 系统设计的重要一步，我觉得这个过程是应该写下来的，既是为了自己的总结，也为了后来进入 ARM 的人可以少走些弯路。

我的开发板配置如下：

- CPU：S3C4510B 芯片(ARM7TDMI 核)
- FLASH：1 片 16×1M 位数据宽度的 FLASH，共 2M 字节 Flash（MX29LV160BTC），速度 70/90ns； 16 位模式。
- 内存 SDRAM：2 片 4M ×16 位数据宽度的 SDRAM（HY57V651620B TC-10S）构成，共 16M 字节 SDRAM。2 片 16 位拼做 32 位使用，共用一个片选。
- 简易 JTAG 调试，两个串口，一个以太网口

从我个人的学习经过认为，初学者最难突破的应该是以下三个方面。所以，本文基本上从这三个方面来阐述。

- ARM 映象文件(包括 axf 文件和 bin 文件)的生成和组成
- 映象文件下在 flash 中的状态和运行时的状态（加载域和运行时域）
- 地址重映射 remap

本文基本围绕附录给出的那个串口发送程序 MySComm4510b002.mcp 而展开的。这个程序的初始化和地址重映射部分参考了 twentyone 的程序（http://bbs.edw.com.cn/dispbbs.asp?boardID=20&ID=27980&page=1），串口发送部分参考了 sofa 的程序（那我自己写了什么呢？☺）。

首先我们应该建立这样一种概念，对于一个裸机（Flash 里什么都没有）来说，所有的细节都得自己设置，不要指望芯片或开发工具可以为你做些什么！所以从第一条指令开始，你就必须负责所有的工作。对于 S3C4510B，上电或复位后从 0x00000000 开始执行指令，而硬件上我们把 Flash 接在了 CPU 的 ROMCON0 处，所以 CPU 就是从 Flash 的 0x00000000 处开始取指令，那么我们就必须保证 CPU 一开始可以取到正确的指令。

小插曲：

我前几天就是被我的 flash 下载程序害死了，由于下载程序的时序问题，开始的 40 多个字节里不能正常烧写，结果老是为 0xff，而我刚刚开始也不懂得 ARM 映象文件的内涵，结果以为 axf 文件就是可执行文件，将其烧入 flash，所以程序真正的第一条指令在 0x34 开始，不能正确烧写的部分正好是 axf 文件的头，没有影响到真正的指令。程序有时也可以执行（因为 0xff 相当于空指令一样，程序也可以执行到真正的第一条指令），所以串口输出的都是不对的乱码或字母。

一 ARM 的映象文件

1.1 初步认识 axf 和 bin 文件

这里我先谈谈 ARM 的映象文件（即可执行文件）的概念。我们生成的 ARM 的映象文件有 axf 格式和 bin 格式两种，有时容易被二者混淆。其实 bin 文件是真正的可执行文件，而 axf 文件是 ARM 特有的调试文件，里面除了包含 bin 文件的内容之外，还附加了许多其他调试信息。首先让我们来看看 axf 和 bin 的区别,图 1～图 4 是用 ultraedit 打开的 axf 文件和 bin 文件的头部和尾部。这两个文件都是 MyScomm4510b002.mcp 用 ADS1.2 生成的（选中 target setting 中的 post-linker:fromelf 和 ARM Fromelf 中的 output format：plain binary 这两个选项，就可以同时生成相应的 axf 和 bin 文件）。

在生成这两个映象文件时，请设置 ARMlinker 的选项：ro_base 设为 0x0000，rw_base 设为 0xa00000，作用将在本文后面解释。

图 1 是 axf 文件，其中反显处才是真正的第一条指令，开始的 52 个字节都是 axf 文件头，而图 2 的 bin 文件从 00000000h 就是真正的第一条指令（所以为什么说 bin 文件才是真正的可执行文件）。关键是从第一条指令开始的二个文件的比较了，哦，竟然是完全相同的，一直到 bin 文件的尾部,见图 3 和图 4。即 bin 文件就是 axf 文件的 00000034h～00000323h。不过，二者的相应指令或数据的地址就不一样了，先记住这一点伏笔。Bin 文件结束（图 4 反显处），但 axf 文件还没有结束（图 3 中反显指示 bin 文件相应结束部分），其后还有很多相关的调试信息，这些调试信息可以用在 ADW 或 AXF 的 load image 的调试中。

由于我们的烧写工具是原封不动地将映象文件烧入 flash 中，所以，axf 文件是不可以烧入的，否则 flash 的 0x00000000 处就不是真正的第一条指令，而是 axf 头部分。我们应该将 bin 文件烧入，保证 flash 的 0x00000000 处是第一条指令。

我个人猜测，axf 文件应该可以通过“ADW 中的下载功能”下载到 ROM 中运行，在下载的过程中可能经过 ADW 相应的内部处理而导致真正烧入 ROM 中的还是 bin 文件。

到现在为止，我们只能感性认识一下映象文件：axf 文件和 bin 文件的不同，也就是说，**bin 文件是 axf 文件的一部分，最精华的一部分**，那么到底哪部分是最精华的呢？bin 文件中包含了哪些内容呢？指令和数据到底放在映象文件的什么地方呢？下面接着分析。

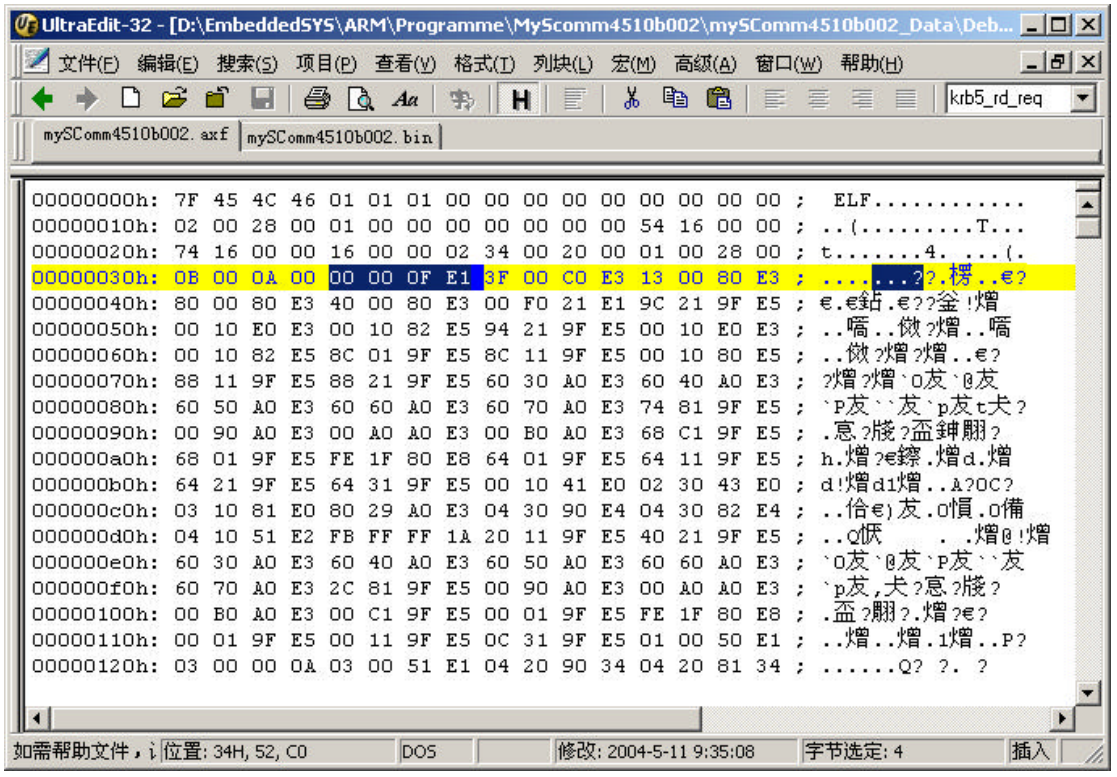

图 1 ultraedit 中打开的 axf 文件二进制形式（头部部分）

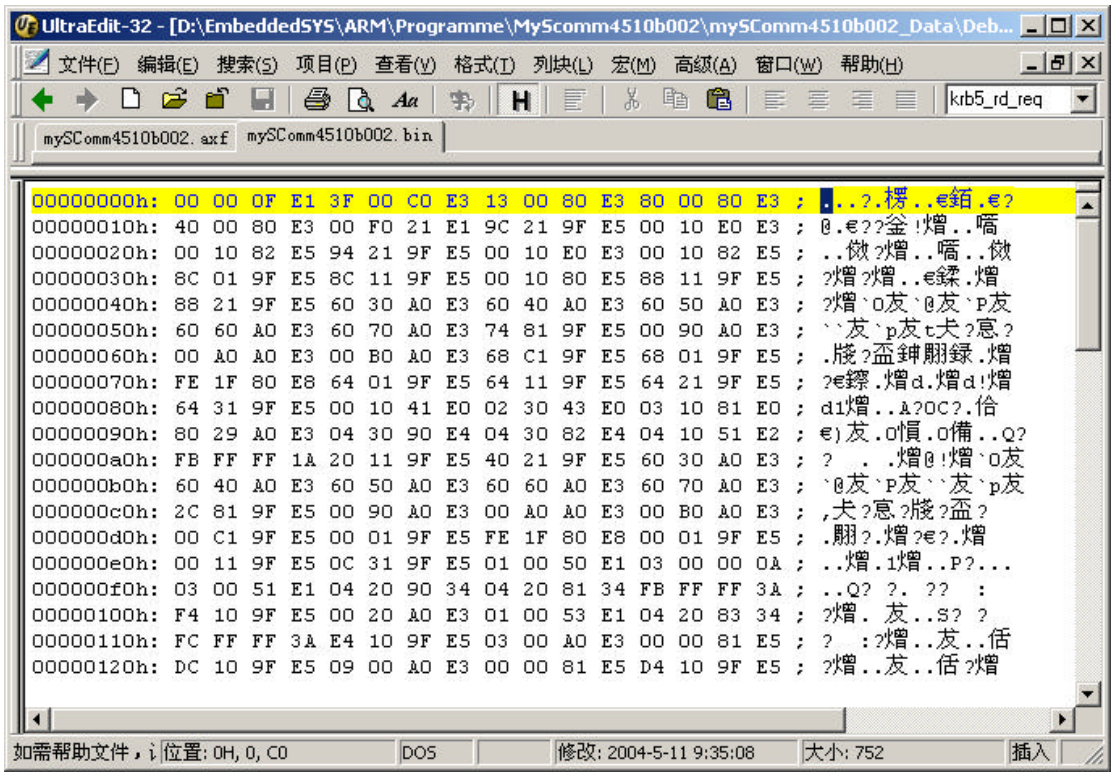

图 2 ultraedit 中打开的 bin 文件二进制形式（头部部分）

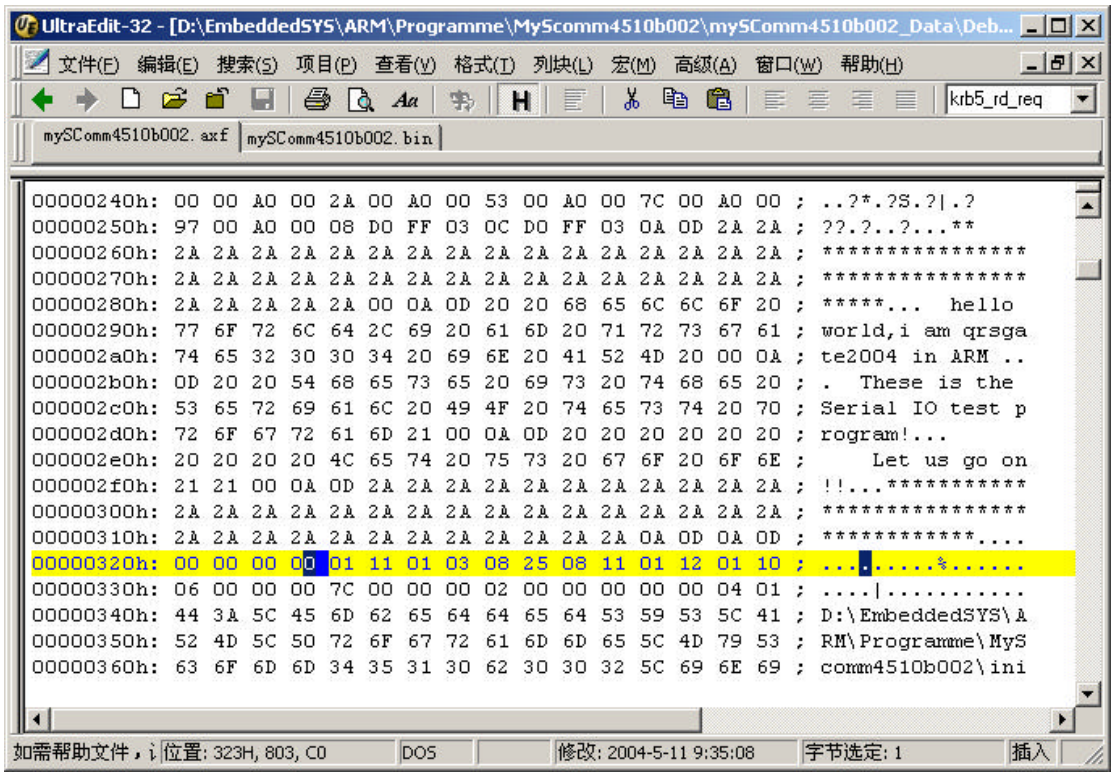

图 3 ultraedit 中打开的 axf 文件二进制形式（中间部分）

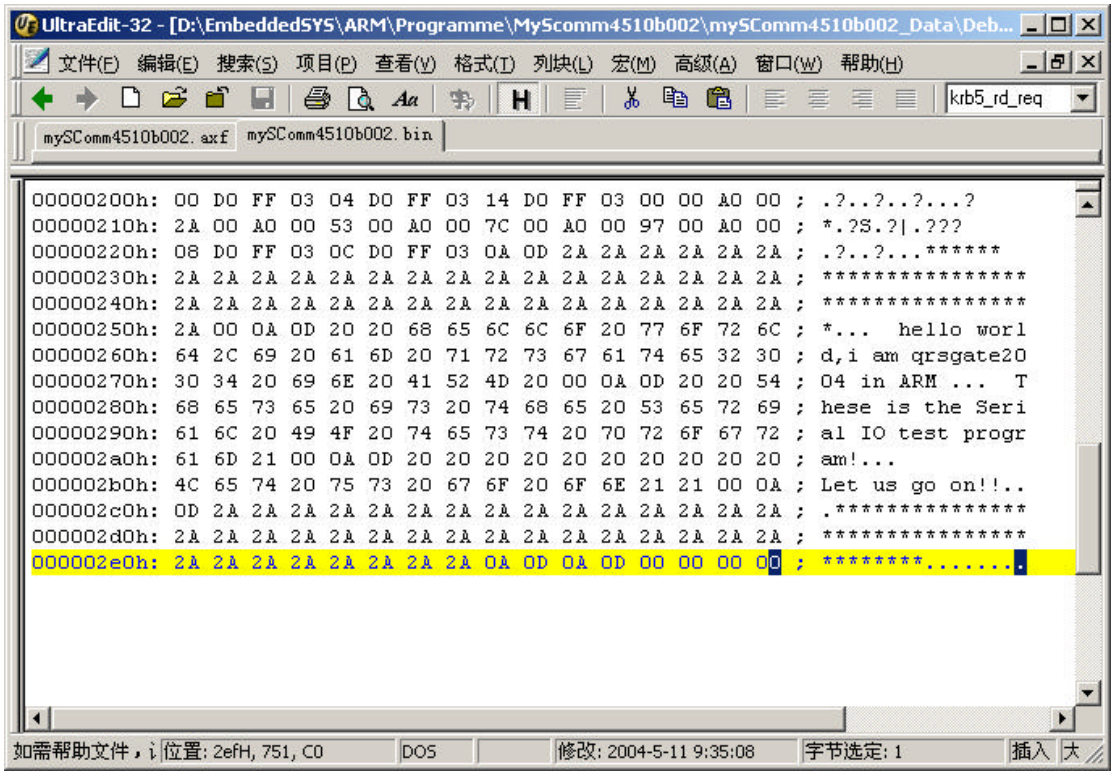

图 4 ultraedit 中打开的 bin 文件二进制形式（尾部部分）

1.2 ARM 映象文件的内容

讨论到映象文件的内容，就应该打开源程序并理解源程序的内容了，这个程序比较简单，我们这里讲解的都是初级知识，给大家一个入门的启示，复杂的以后可以再慢慢学。所以大家看到这篇文章不要老是想到复杂的情况，那样本文有些阐述可能就不太对头了 :)

首先我们可以先不看程序具体做了什么，我们总应该可以看出这个简单程序分为两个部分，一个是 CODE 部分，即指令代码部分；另一部分是 DATA 部分，即数据部分。

```
        AREA    Init, CODE, READONLY        (代码部分 RO)
        CODE32
        GET snds.s

        ENTRY
Start
    ..............
        AREA PRINTLINEOUT,DATA,READWRITE     (数据部分 RW)
LINE1   DCB     &A,&D,"****************************************",0
    ..................
        END
```

书上说，映象文件一般由域组成，域由最多三个输出段（RO,RW,ZI）组成，输出段又由输入段组成。那么我们来看看生成的映象文件 MySComm4510b002.bin 到底怎么符合书上的概念，又怎么和上面源程序的代码和数据部分对应起来的。

先来看看图 5 吧，我觉得实物最可以给人以说服力。

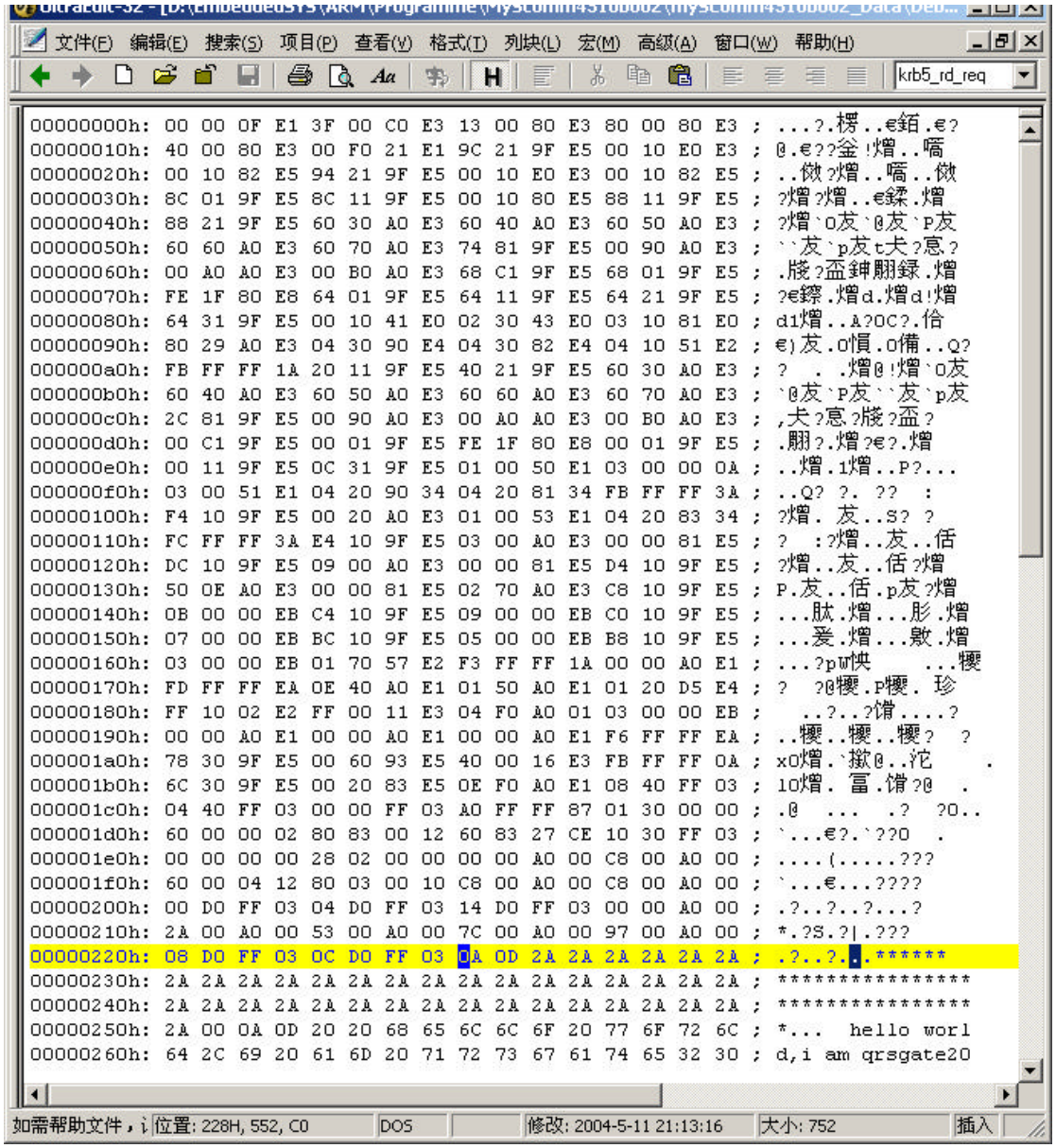

图 5 ultraedit 中打开的 bin 格式映象文件二进制形式（大部分）

大家看到了吧，蓝色反显的前部其实就是源程序对应的指令代码部分，反显后对应的就是源程序的数据部分。反显处就是指令部分 RO 和数据部分 RW 的分界处。大家仔细看看，反显开始的地方往后其实就是要从串口输出的字符数据（由于屏幕关系，没有完全显示出，大家可以自己用 ultraedit 打开看看）；而从 00000000h 开始至 00000227h 处是程序中的指令编码（编译好的二进制编码）。

所以，可以这么说，

- 域：整个 bin 映象文件，也就是说这个简单程序的映象文件只有一个域（加载域），其实大部分程序都是只有一个加载域
- 输出段：有两个输出段，RO 和 RW，没有 ZI。这个我们从源程序和图 5 也可以看出。
- 输入段：两个输入段，即源程序的 CODE 部分和 DATA 部分。CODE 部分是 READONLY，属于 RO 输出段，DATA 部分是 READWRITE，属于 RW 输出段。

所以，域、输出段和输入段的关系如下图：

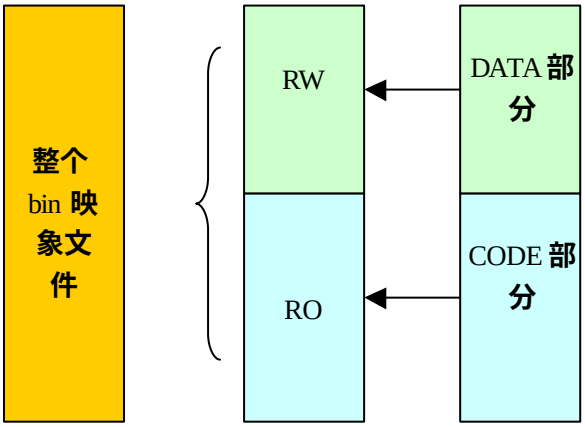

图 6 例子程序的域、输出段和输入段的关系

在加载域中，RW 直接跟在 RO 后面，那我们刚刚设置的连接选项：ro_base=0x0000，rw_base=0xa00000 有什么用呢？RO 的确是从 0x00000000 开始，可是 RW 却从 0x00000228 开始，和 rw_base 不同阿？到底这是怎么回事呢？

二 ARM 的映象文件的加载域和运行时域

所有这些，都是“ARM 连接”和“系统存储器多样化”惹的祸！ARM 连接器一个很重要的工作就是要解析目标文件中各种符号，也就是得到各符号的值。

还是先来看看图 7 吧

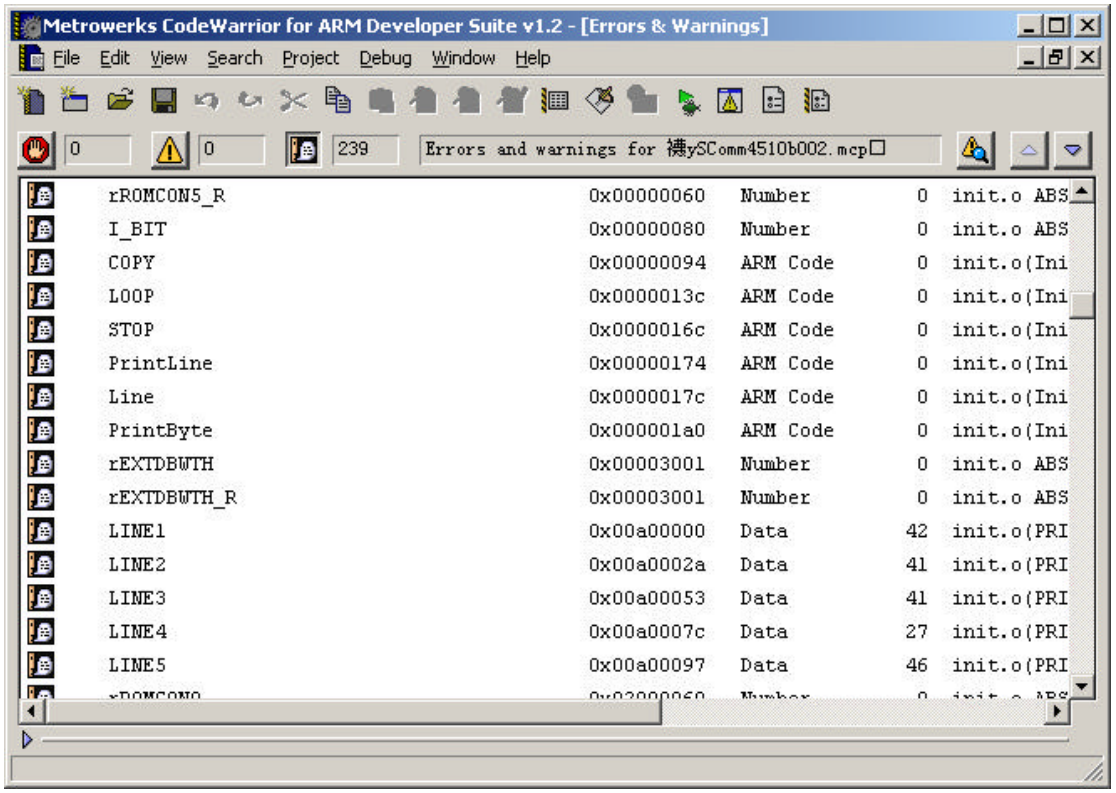

图 7 例子程序 ADS 编译连接的符号映射图（一部分）

这张图是例子程序在 ADS1.2 中编译连接后显示出的符号映射 MAP 的一部分（选中

ARMlinker 中 listing 选项卡的 image map 和 symbols 即可），其中设置了 ro_base=0x0000，rw_base=0xa00000,可以看到象 COPY,LOOP,STOP,PrintLine 等符号的值都是相对于 ro_base 而定位的（在 RO 段中），而 LINE1、LINE2、LINE3、LINE4 和 LINE5 却是相对于 rw_base 而定位的（在 RW 段中）。

这些符号值是在映象文件开始运行时才起作用。如果仅仅是放在 flash 中或仅仅运行前面几条初始化指令（加载域状态），那么这些值还暂时不起作用，否则问题就麻烦了。所以真正运行时，就必须保证这些值是对的！！于是引出了“数据移动”。

2.1 概念

什么是加载域和运行时域呢？我个人认为用“域”这个词实在很容易混淆。其实简单明了的说，加载域就是最原始的 bin 文件，就是映象文件烧入 flash 时的状态，如图 5 所示，RW 跟在 RO 后面；而运行时域就是经过了改头换面的映象文件，即由于运行了相应的初始化程序而把 RW 或 ZI 拷贝到相应的地方，这是映象文件已经四分五裂成至多三个部分，图 8 示意例子程序映象文件的两个不同状态。

当然运行时域也可以和加载域相同，就是不设置 rw_base,那么运行时不需要移动，依然跟在 RO 后面。

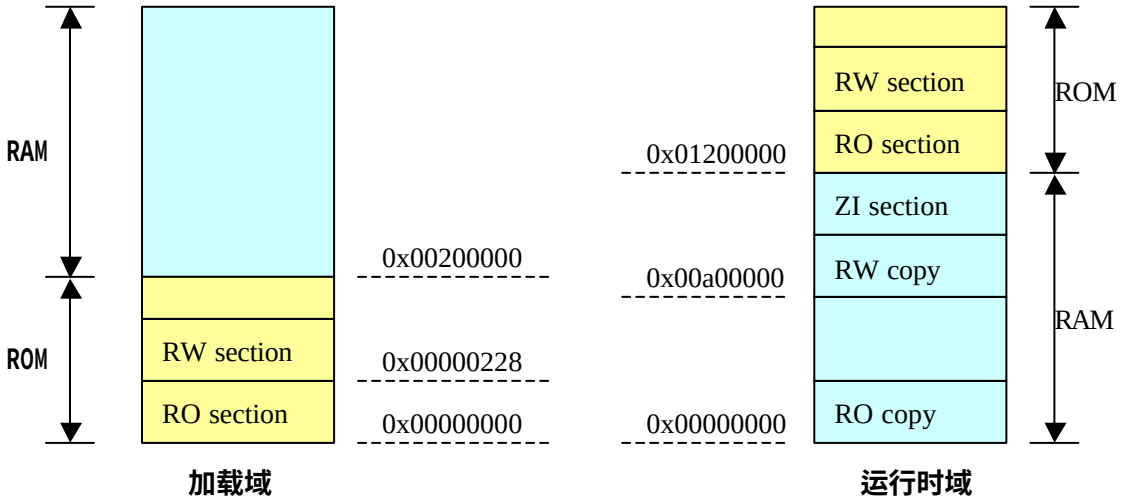

图 8 例子程序加载域和运行时域比较示意图

在加载域状态，RO 地址是正确的，意味着指令可以正确执行；但 RW 是不正确的(从图 7 可知)，但是暂时的错误不影响程序初始化，源程序的 part1～part6 都暂时没有用到 RW，所以程序还是可以正确运行，即使 RW section 在 0x00000228 处。程序终究是要用到 RW 的，那么怎么办？其实只要来个“乾坤大挪移”就好了，在用到 RW 之前将 RW 移动到正确的位置即可。Part7 完成了这个“挪移”过程。当然移动前进行了地址重映射，不过，地址重映射和 RW 的数据移动其实是两回事！

在 twentyone 的 bootloader 中（即例子程序的 part5～part6），在映射前，将 RO 的备份拷到 RAM 中，然后将 RAM 重映射为 0x00000000，非常巧，指令运行不受任何影响！重映射后 flash 的地址为 0x01200000，此时主动权已经到 RAM 中的 RO copy 手上了！然后再将 RW 移动到相应的位置。哪里才是 RW 相应的正确位置呢？

2.2 RW 的数据移动

有一个概念很重要：“加载域状态”的映象文件中 RO、RW 和 ZI 的所在地址都是临时的，他们所在的真正位置（即连接时设置的地址值）都必须在程序初始化时由相应程序，将他们

移动到相应的地方。这个数据移动过程由 Bootloader 完成 !!!

源程序由 8 个部分组成：part1～part7 主要是执行 CPU 的初始化和内存重映射，相关资料可以参考文章(http://bbs.edw.com.cn/dispbbs.asp?boardID=20&ID=27980&page=1)。Part8 主要是通过串口发送几行字符，这里主要解释一下 part3 和 part5。

```
;Part 3
;*************************************************************
;   Import some important variables for later use

    IMPORT |Image$$RO$$Base|
    IMPORT |Image$$RO$$Limit|
    IMPORT |Image$$RW$$Base|
    IMPORT |Image$$RW$$Limit|
    IMPORT |Image$$ZI$$Base|
    IMPORT |Image$$ZI$$Limit|
```

这六个全局变量分别是 RO 输出段运行时起始地址，RO 运行时存储区域界限，RW 输出段运行时起始地址，RW 运行时存储区域界限，ZI 输出段运行时起始地址，ZI 运行时存储区域界限。引入的这六个变量可说是在映象文件编译连接中起到至关重要的作用，这里我们就不说得非常复杂了，就简单用例子程序来举例说明。

现在再让我们回头看看前面编译连接时 ADS1.2 中的 map 及 symbols 图的另外一部分。如图 9。

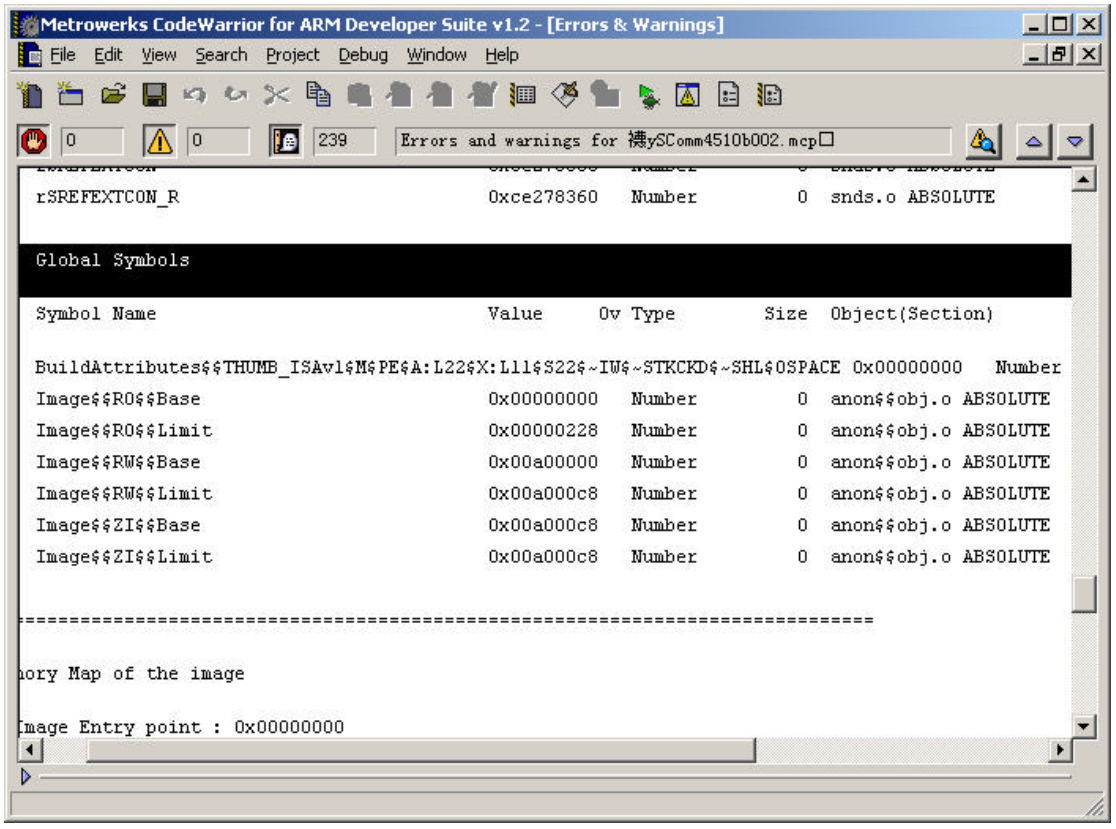

图 9 ADS1.2 编译连接后的映射表一部分

其中：

- Image$$RO$$Base =0x00000000
- Image$$RO$$Limit=0x00000228
- Image$$RW$$Base =0x00a00000
- Image$$RW$$Limit=0x00a000c8
- Image$$ZI$$Base =0x00a000c8 （其实 ZI 是不存在的）
- Image$$ZI$$Limit=0x00a000c8

大家把这些地址再和图 5 的 bin 文件（映象文件）对照看看，有哪些相似之处吗？

RO 的 base 和 limit 和在 ultraedit 中打开的映象文件一样，也就是说和烧在 flash 中的一样了！ Image$$RW$$Base 和 Image$$RW$$Limit 明显不对，不过，仔细看看，除了地址不对外，其实相对值还是一样的，也就是说，只要把 Flash 中的 RW 移到 Image$$RW$$Base 所指示的位置即可。

Image$$RO$$Limit 是 RO 的存储地址界限，其实是 flash 中 RW 的开始地址（加载域状态时），因为加载域状态时 RW 就是紧跟在 RO 后面！而 Image$$RW$$Base 是 RW 运行时的开始地址，二者是不同的。

Part7 中正是通过判断 Image$$RO$$Limit 和 Image$$RW$$Base 是否相等来决定程序运行时 RW 是否需要“搬迁”，这个例子中，二者是不等的，就是说，“程序运行时”RW 不应该在 flash 中的这个位置——RO 的后面，那么初始化程序就必须在未使用 RW 前将 RW 移到正确的位置，例子程序中则将 RW 移到了 0xa00000 处。从而使得编译连接时和 RW 相关的标号值都正确。

```
        LDR r0, =|Image$$RO$$Limit|
        LDR r1, =|Image$$RW$$Base|
        LDR r3, =|Image$$ZI$$Base|

        CMP r0, r1
        BEQ %1

0       CMP r1, r3                      ; Copy init data
        LDRCC r2, [r0], #4
        STRCC r2, [r1], #4
```

一旦 RW 被移动到 0xa00000 处，那么图 7 中的 LINE1、LINE2、LINE3、LINE4 和 LINE5 等 RW 段的符号值都是正确的了！

ADS 在生成加载域——也就是映象文件（bin 文件）时，是把 RW 紧接在 RO 后面，这并不是 RW 的实际地址，而仅仅是加载时，也就是烧写到 flash 时的代码和数据的临时地址。而程序真正开始运行起来时——也就是运行时域，一切情况又都发生了变化，该到什么地方的，就应该去什么地方，所以 RW 应该拷到 0xa00000 处，以保证连接时的符号值都是正确的。

三 ARM 系统的地址重映射

其实如果真正懂了上面两个原理，那么理解地址重映射就很容易了，虽然他们之间没有什么联系，但是，很多人不能理解地址重映射就是因为和映象文件的编译连接地址搞得一团糊！

地址重映射就是通过系统的存储管理部件改变系统中各个存储器的映射地址，在

S3C4510B 中，通过改变系统控制寄存器组来改变存储器的映射地址。如果你对芯片的存储管理部件还不是很熟的话，应该先把芯片资料再看看。

一般的 S3C4510B 开发板都是将 flash 连接到 ROMCON0 处，这样上电复位时 CPU 读的第一条指令是 flash 的 0x00000000 处的值。例子程序中 part1～part5 程序在运行时都没有映射，在 part4 为了 part5 的 bootloader 的复制，特意初始化了 ROM 和 SDRAM，具体设置如图 8 的加载域所示，flash 为 2M，从 0x00000000～0x0001fffff ；SDRAM 为 16M，从 0x00200000～0x01200000，其他系统控制寄存器则设置为 0x03ff0000～0x03ffffff（这个由 part2 设置）。

```
;Part 4
;*************************************************************
;Initalize the memory as followa:
;   FLASH           @ 0  ~ 2 M
;   SDRAM           @ 2  ~ 18M


    LDR r1, =rEXTDBWTH              ;EXTDBWTH
    LDR r2, =rROMCON0               ;ROMCON0    @ 0M ~ 2M
    LDR r3, =rROMCON1               ;ROMCON1    @ DISABLED
    LDR r4, =rROMCON2               ;ROMCON1    @ DISABLED
    LDR r5, =rROMCON3               ;ROMCON1    @ DISABLED
    LDR r6, =rROMCON4               ;ROMCON1    @ DISABLED
    LDR r7, =rROMCON5               ;ROMCON1    @ DISABLED
    LDR r8, =rSDRAMCON0             ;SDRAMCON0  @ 2M ~ 18M
    LDR r9, =rSDRAMCON1             ;SDRAMCON1  @ DISABLED
    LDR r10,=rSDRAMCON2             ;SDRAMCON2  @ DISABLED
    LDR r11,=rSDRAMCON3             ;SDRAMCON3  @ DISABLED
    LDR r12,=rSREFEXTCON

    LDR r0, =ARM7_EXTDBWTH
    STMIA   r0, {r1-r12}
```

3.1 地址重映射

先来看看例子程序 part6：

```
;Part 6
*************************************************************
;Remap the memory
;   FLASH   @ 16 ~ 18M
;   SDRAM   @ 0  ~ 16M


    LDR r1, =rEXTDBWTH_R            ;EXTDBWTH
    LDR r2, =rROMCON0_R             ;ROMCON0    @ 16M ~ 18M
    LDR r3, =rROMCON1_R             ;ROMCON1    @ DISABLED
    LDR r4, =rROMCON2_R             ;ROMCON2    @ DISABLED
    LDR r5, =rROMCON3_R             ;ROMCON3    @ DISABLED
```

```
    LDR r6, =rROMCON4_R             ;ROMCON4    @ DISABLED
    LDR r7, =rROMCON5_R             ;ROMCON4    @ DISABLED
    LDR r8, =rSDRAMCON0_R           ;SDRAMCON0  @ 0M ~ 16M
    LDR r9, =rSDRAMCON1_R           ;SDRAMCON1  @ DISABLED
    LDR r10,=rSDRAMCON2_R           ;SDRAMCON2  @ DISABLED
    LDR r11,=rSDRAMCON3_R           ;SDRAMCON3  @ DISABLED
    LDR r12,=rSREFEXTCON_R

    LDR r0, =ARM7_EXTDBWTH
    STMIA   r0, {r1-r12}
```

在前面的 5 个部分的准备后，特别是将 FLASH 中的内容复制到原先的 0x00200000 处，系统已经进入了可以重映射的状态了。重映射后，系统开始在 SDRAM 中运行了，而不在 flash 中，由于第五部分的拷贝，导致映射后的指令切换的无缝连接 !! 大家可以琢磨一下！

至于映射后，映象文件是否可以正常运行，以及下一步要做什么，我看我就不多说了……

让我们在 ARM 学习报告 002 再见 !!

感谢我的小猫，在五一节陪伴着我 !!

很多东西，如果你懂，它就很简单；如果你不懂，仿佛简单的它也会很难