电 子 科 技 大 学

嵌入式智能计算研究团队

珊瑚-I 启动过程

ACORAL-I BOOT PROCESS



**版本号1.1**

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本号** | **内容** | **日期** | **负责人** |
| 0.1 | 开始编写，修改 latex 模板，确定大纲 | 2022.05.15 | 王彬浩 |
| 1.0 | 第一版 aCoral 启动手册编写完成 | 2022.06.12 | 王彬浩 |
| 1.1 | 更新acoral\_module\_init代码 | 2022.07.23 | 王彬浩 |

目　录

[第一章 Bootloader 1](#_Toc10331)

[1.1 什么是 Bootloader 1](#_Toc20521)

[1.2 loader 1](#_Toc14316)

[第二章 aCoral 启动 4](#_Toc30990)

[2.1 启动-第一阶段 4](#_Toc19098)

[2.1.1 禁用看门狗 5](#_Toc669)

[2.1.2 关中断 6](#_Toc28070)

[2.1.3 时钟初始化 7](#_Toc525)

[2.1.4 存储空间初始化 9](#_Toc15550)

[2.1.5 栈初始化 10](#_Toc31658)

[2.1.6 自我拷贝（loader） 11](#_Toc11740)

[2.1.7 bss 段清零 12](#_Toc22166)

[2.1.8 跳转至下一阶段 13](#_Toc14823)

[2.2 启动-第二阶段 13](#_Toc25893)

[2.2.1 acoral-start 13](#_Toc23242)

[2.2.2 acoral-core-cpu-start 14](#_Toc28403)

[第三章 aCoral 内存分布 17](#_Toc414)

[3.1 存储地址和运行时地址 17](#_Toc4317)

[3.2 aCoral 链接脚本 1](#_Toc12113)9

# 

# 第一章 Bootloader

## 什么是 Bootloader

我们将 Bootloader 拆开来看，一个是 boot, 一个是 loader，可见 Bootloader 有两个主要特性:

(1) Boot。Boot 的原意为靴子，在计算机领域引申为启动，也就是说系统启动时会从这里启动，具体一点就是当我们按开机键，cpu 执行的第一条指令就应该是Bootloader 的代码。

(2) Loader。Loader是加载的意思。那自然就可以引出五个问题：

i. 加载什么（what）

当然是加载代码程序，这个程序就是我们经常谈到的内核映像——像 linux 内

核，window 内核，亦或是 aCoral 的内核等。

ii. 怎么加载（how）

加载说白了就是复制，将内核代码原封不动的从一个地方复制到另一个地方。

iii. 从哪里加载（from）

自然是从放着内核代码的地方加载了，常见的存储介质有 flash、SD 卡等。

iiii. 加载到哪里去（to）

自然是加载到内核要运行的地方——内存，常见的有 SDRAM、DDR 等。

iiiii. 为什么要加载（why）

这就不是一句两句能说清楚的了。简单来讲，因为 flash、SD 卡这些非易失性存储器不能用来运行程序，或者不适合运行程序，但是他们便宜，容量大，断电也不会丢失数据，适合存放程序，我们一般称为外存。而 SDRAM、DDR 这些易失性存储器不能用来在断电的情况下存放程序，但是在通电之后运行速度快，适合运行代码，一般称为内存。所以，上电之后需要通过 loader 来把内核程序从外存复制到内存中，这样内核程序就能正常运行了。

在嵌入式系统中，常见的 Bootloader 有vivi,uboot ，这些程序都是开机时就启动的，它启动后，会从 flash 或 sd 卡等存储设备中将内核程序代码拷贝到 sdram,然后执行内核代码。

如果你曾经接触过vivi、uboot这些开源的 bootloader，就会发现，这些bootloader似乎都没有这么简单，大小也都往往几 MB 往上，这是为什么呢？有两个主要原因：(1) 它们都支持多平台，它们都可以看成一个通用的 Bootloader。(2) 除了提供上面启动，加载两个功能外，它们还支持更多功能，比如支持各种命令，比如操作 nandflash,norflash,EEPROM，支持 ftp,tftp,nfs 等网络协议，又或者支持 usb 下载等功能。有些 Bootloader 如 arm 公司的 bootmonitor 还支持文件系统，能以文件系统的方式管理 nandflash,sdcard,compact card 上的数据。有了上面两大类的支持后，Bootloader 不再是纯粹的 Bootloader, 都有了操作系统的一些功能，只是不支持操作系统支持的任务切换功能。aCoral 的 bootloader 其实就是

h a l / s 3c 2 4 4 0 / s r c / s t a r t . S

这个文件。它就没有这么多复杂的功能了，仅仅做了 bootloader 最本职的两个工

作：启动、加载。关于这个文件的解析，将在后面详细阐述。

## loader

关于为什么要加载（why）这个问题，这一小节就来细说。如果你对这一部分没有特别强烈的求知欲，可以暂时先跳过，以后再来阅读。我们的程序代码存放或者运行的地方称为存储介质。储存介质选择的主要参考：速度，尺寸，价格。存储介质按照不同的方法，可以分为不同的种类。

1) 按存取方式分类如果存储器中任何存储单元的内容都能被随机存取，且存取时间和存储单元的物理位置无关，这种存储器称为随机存储器。半导体存储器和磁芯存储器都是随机存储器。如果存储器只能按某种顺序来存取，也就是说存取时间和存储单元的物理位置无关，这种存储器称为顺序存储器。例如，磁带存储器就是顺序存储器。一般来说，顺序存储器的存取周期较长。磁盘存储器是半顺序存储器。

2) 按存储器的读写功能分类有些半导体存储器存储的内容是固定不变的，即只能读出而不能写入，因此这种半导体存储器称为只读存储器 (ROM)。既能读出又能写入的半导体存储器，称为随机存储器 (RAM)。

3) 按信息的可保存性分类断电后信息即消失的存储器，称为易失性存储器，或者非永久记忆的存储器。断电后仍能保存信息的存储器，称为非易失性存储器，或永久性记忆的存储器。磁性材料做成的存储器是永久性存储器，半导体读写存储器 RAM 是非永久性存储器。

我们常见的储存介质大类有：磁带，硬盘，ROM，RAM, 具体到嵌入式：经常用到是 norflash,nandflash,sdcard,TF 卡，compact 卡，sdram,ram 等。为什么会出现这么多种类？这个是价格和需求平衡的结果。比如, 我们知道程序最后运行必须要有随机可读写存储器来存储变量，且速度要快，这个导致了 RAM 的产生，但是RAM 价格昂贵，又导致了 sdram 的产生，sdram 和 ram 的区别就是它是靠电容的值来保存 0，1 信息，时间一长就会丢失数据，故需要周期性刷新，这个在 sdram控制器芯片的控制下能很好解决，且不太影响性能，但是它速度比 ram 低一些，且复杂些，但是价格低很多，且容易做到很大，故是一种很好的储存器，因此目前无论是嵌入式还是 pc 设备都广泛使用到了 sdram。虽然 sdram 解决了可读写问题，且速度问题。但是它们都是非永久记忆的存储器，断电后信息即消失的存储

器，明显不能满足我们要求永久保存我们代码的需求，你总不至于，每次启动电脑都要下载一次程序吧，于是就产生 nandflash, 硬盘这些永久记忆的存储器（硬盘太大，很少用在嵌入式系统中），这些储存器是永久，且能做到很大容量，但是速度慢，不过还是可以承受的，因为我们有办法解决这个问题？如何解决，就是前面说的加载，就是说在启动阶段，Bootloader 启动后就从这些储存介质拷贝程序到sdram，这样真正运行时，代码和数据是从 sdram 中读取的，也就没有速度问题了，这也是为啥要 Bootloader 的原因。

有了 nandflash, 硬盘这些永久记忆的存储器还不够，为啥？因为它们是按块访问的，而不是按地址访问，这种块模式访问往往需要有硬件控制器，而硬件控制器又需要由程序来控制，那这个控制器的程序从何而来？这就是鸡生蛋，蛋生鸡的问题，正因为这样又出来一种存储器——ROM，比如 norflash, 只读存储器，这种储存器也是永久记忆的存储器，但它和 nandflash 等不一样，它是按地址随机访问的，也就是说不需要驱动，和 sdram 的访问方式一样，可以很简单的访问数据，这就解决了这个问题，但是这种按地址访问的永久记忆的存储器相比有点贵，且不能做到很大。其实也没必要过多的使用这中存储器，为啥？因为它是只读的，没法修改，不会过多使用，所以只要能够容下 Bootloader 这些程序就可以了，其他的代码交给廉价的可写的 nandflash 吧，当然对于代码还是可以一直放在 rom 中的，这样可以减少 Sdam 的使用。

也许你会说为啥不出产一种按地址访问的可读写永久记忆的存储器，是可以啊，但是代价太高，没必要，只要合理搭配，就可以满足需求，当然不排除有一天，按地址访问的可读写永久记忆的存储器很便宜了，但是这个世界没有完美的东西，优点越多，缺点也越多。

下面就来说下嵌入式储存器搭配问题：硬盘肯定是不到万不得已，是不选择的，因为这个家伙体积大，功耗大，也不安静，不过对于需要储存上 10G 的数据的应用，还不得不用它。Bootloader 程序的储存器肯定得要是按地址随机存取的永久性记忆的存储器，当然对于支持 nandflash 启动的 soc，也可以储存在 nandflash, 比如 s3c2410,2440, 同时又比如 omap3530 是支持 sdcard 启动的，这样的 SOC 芯片也是可以将 Bootloader 放在 sdcard 上的。也许到了这里，你会有一种强烈的好奇性？刚才不是说 nandflash,sdcard 都是需要控制器才能访问数据的啊，控制器又需要程序，上面的 s3c2410,omap3530 等芯片是如何做到从这些地方启动的。其实解决方法和我们上面探讨的一样，就是必须有一个拷贝动作，这个拷贝动作可以有三种方式，一种是硬件方式，另一种是软件方式 1）硬件方式：就是硬件实现储存设备控制器的控制，读取指定大小的数据，它没法做到控制器的驱动程序那样，可以随机读取任意大小的数据，但是只要能够拷贝指定地址指定大小的数据，就已经够了，硬件可以看成是简化版的驱动 2）软件方式：这个就更简单了，芯片自带一个 ROM, 往往是片内 ROM, 这个 ROM 里装有驱动程序，这个驱动程序负责将我们的 Bootloader 从 nandflash 或 sdcard 等储存器拷贝到 sdram 或 ram 后，然后跳到我们的 Bootloader 运行，这样其实和我们将 Bootloader 储存在 rom 是一样的，只不过板子自带了一个 Bootloader, 这个简单的 Bootloader 先于我们的 Bootloader 运行，主要实现小量数据（至少包括我们的 Bootloader 的自我拷贝代码）拷贝。 其实还有另外一种启动技巧，那就是内存映射：就是说当用户使用跳线选择方式后，硬件自动开启了内存映射，将其他内存地址映射到 cpu 启动地址，比如pb11mpcore，cpu 的启动地址是 0x0, 如果配置为 Norflash 启动，则可将 norflash 的地址 0x40000000 0x43FFFFFF 映射到 0x0 0x3ffffff, 这样就相当于从 norflash 启动，这种方式是经常用的方式。由于内存映射到地址 0x0 了，导致地址 0x0 对应的的内存没法使用，因此启动后需将这个映射取消这种和上面的方式不同，这种需要有按地址随机存取存储器的支持，即将储存启动代码的存储器的地址映射到启动地址。

说完 Bootloader 的储存介质，就得说说操作系统（通常叫 kernel）映像文件的储存介质。嵌入式操作系统这类操作系统一般比较小，选择余地有很多，可以放在rom 中，也可以放在 nandflash 中，因为不论放在哪里，只要 Bootloader 能找到，拷贝到 sdram 就可以了，所以关键看 Bootloader 是否强大，对于很强大的 Bootloader，其实操作系统都可以放在主机上，然后 Bootloader 可以通过网络将操作系统下载到 sdram，然后启动操作系统。对于 Bootloader 和内核链在一起的操作系统，操作系统肯定就是跟 Bootloader 一起储存在一种储存介质中了啊。

# aCoral 启动

2440 上电之后，CPU 将从 0 地址处开始取指执行指令。如果将 aCoral 程序存放在 Norflash 中，并通过 mini2440 开关 S2 选择 Norflash 启动，则 Norflash 从硬件层面上被映射为 0 地址开始的一段地址；如果将 aCoral 放在 Nandflash 中，并且通过 S2 开关选择 Nandflash 启动，则开发板在上电后自动将 Nandflash 前 4KB 内容复制到开发板上的一块 SRAM 中。这块 SRAM 我们称为 Stepping Stone（垫脚石），并且 Stepping Stone 的地址就是从 0 地址开始。所以，不论选择何种启动方式，2440 都将从 0 地址开始执行第一行代码。

图2-1显示了复位后 S3C2440A 的存储器映射情况，详细请参考《S3C2440 中文手册》第五章。

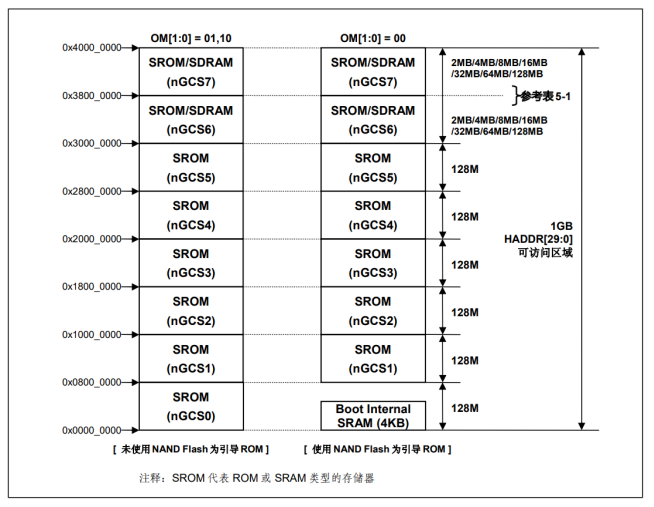


图 2-1 复位后 S3C2440A 的存储器映射

## 启动-第一阶段

之前说到，aCoral 的 bootloader 其实就是

h a l / s 3c 2 4 4 0 / s r c / s t a r t . S

我们将其称为 aCoral 的启动文件。启动文件中，这行跳转指令就是整个 aCoral

的入口，将被烧录在 Nandflash 或 Norflash 的 0 地址。

\_\_ENTRY:

b R e s et H a n dl e r

这句跳转程序将跳转到 ResetHandler 标号处，执行一些上电之后的硬件初始化工作，包括关闭看门狗、配置时钟、堆栈初始化、复制 OS 到 SDRAM 等。我们一点点来看这些代码。

PS: 请准备好《S3C2440 中文手册》

### 禁用看门狗

@ d i s a b l e watch dog t i m e r

mov r1 , #0 x53000000

Mov r2 , #0 x0

s t r r2 , [ r 1 ]

看门狗 WatchDog 的名字形象的描述了它的工作原理，看门狗每隔一段时间（比如：3 个小时）它就会饥饿，每次饥饿时都叫，如果不想让它叫，只要我们保证在 3 个小时内喂狗一次就行。因此我们要及时的对看门狗控制器执行喂狗操作。看门狗定时器内部有一个递减计数器，当该计数器递减为 0 的时候，就会自动重启控制器，如果我们写有这样的程序，该程序在定时器计数器递减为 0 之前，将其递减计数器重新设置一下（喂狗），那么就不会产生重启操作。假如机器设备出现异常情况下如死机，CPU 执行出错，程序跑飞等情况，CPU 就会陷入非正常的执行流程，就不会去执行重置计数器的程序，当计数器递减为 0 时，会产生复位控制器信号，机器就会重新启动，恢复正常执行流程。这样的设计原理就解决了很多环境恶劣的情况下，对服务器进行重启的任务。上面的重置倒计数的操作通常叫做“喂狗”。为了避免看门狗带来的影响，简化系统，我们选择关闭看门狗。上述代码向地址 0x53000000（r1），也就是看门狗定时器控制寄存器（WTCON）写入了 0x0000（r2），即 16 个 0。结合图2-2，可以知道，这样配置的结果就是禁止了看门狗，系统也就不需要定时去喂狗了。当然了，在比较正式的系统中，看门狗是必须要开启的，防止系统一直死机。这里由于我们只是在开发 aCoral，所以暂时关闭。

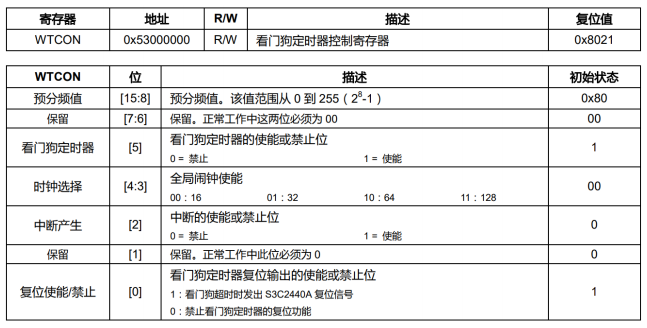


图 2-2 看门狗定时器控制（WTCON）寄存器（手册第 18 章）

### 关中断

@ d i s a b l e a l l i n t e r r u p t s

mov r1 , #INT\_CTL\_BASE

mov r2 , #0 x f f f f f f f f

s t r r2 , [ r1 , #oINTMSK]

l d r r2 , =0 x 7 f f

s t r r2 , [ r1 , #oINTSUBMSK]

系统刚上电启动，这个时候，aCoral 的中断系统还没有初始化，此时发生中断我们无法处理，所以要关闭中断。代码中，立即数 INT\_CTL\_BASE = 0x4A000000 ，立即数 oINTMSK = 0x08 ，两个立即数相加的地址即指向中断屏蔽寄存器。中断屏蔽寄存器由 32 位组成，其每一位都都涉及一个中断源。如果某个指定为被设置为 1，则 CPU 不会去服务来自相应中断源（请注意即使在这种情况中，SRCPND 寄存器的相应位也设置为 1）的中断请求。如果屏蔽位为 0，则可以服务中断请求，如图2-3。oINTSUBMSK 指向的次级中断屏蔽寄存器类似。

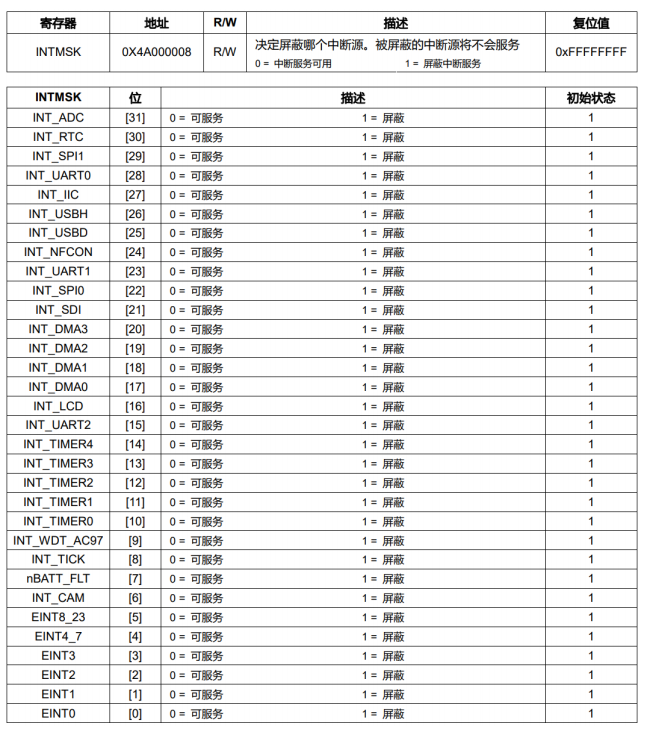


图 2-3 中断屏蔽寄存器（手册第 14 章）

### 时钟初始化

@ i n i t i a l i s e s y st em c l o c k s

Mov r1 , #CLK\_CTL\_BASE

mvn r2 , #0 x f f 0 0 0 0 0 0

s t r r2 , [ r1 , #oLOCKTIME]

mov r1 , #oCLKDIVN

mov r2 , #M\_DIVN

s t r r2 , [ r1 , #oCLKDIVN]

mrc p15 , 0 , r1 , c1 , c0 , 0

@ r e a d c t r l r e g i s t e r

o r r r1 , r1 , #0 xc0000000 @ A s y nc h r o n o u s

mcr p15 , 0 , r1 , c1 , c0 , 0 @ w r i t e c t r l r e g i s t e r

mov r1 , #CLK\_CTL\_BASE

l d r r2 , =vMPLLCON

@ c l o c k u s e r s e t

s t r r2 , [ r1 , #oMPLLCON]

mini2440 上电后，系统工作在板载晶振 12MHz 的频率下。这个频率比较低，系统的性能还没有完全得到发挥，所以需要激发一下潜能，使用一种叫做锁相环PLL 的东西来倍频，加倍之后的频率再给 CPU 和板子上的其他设备使用。mini2440 有两个锁相环，MPLL 和 UPLL。MPLL 的输出频率直接给 CPU 使用，称为 FCLK，同时经过上面初始化系统时钟后得到另外两个频率 HCLK 和PCLK，分别给板载高速硬件和低速外设使用。将 M\_DIVN = 0x5 写入 oCLKDIVN + oCLKDIVN = 0x4C000014 寄存器后，三者的比例为 FCLK：HCLK：PCLK=8：2：1，如图2-4 .

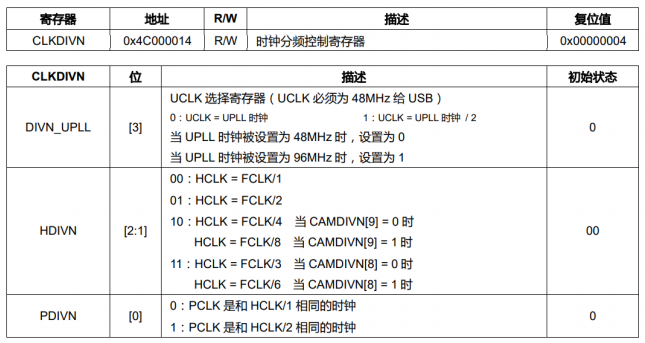


图 2-4 时钟分频控制寄存器（手册第 7 章）

将vMPLLCON = 0x1FC0021 写入 oMPLLCON + CLK\_CTL\_BASE = 0x4C000004寄存器后，得到 FCLK≈400MHz，再根据之前的比例得到 HCLK 和 PCLK 分别为100MHz 和 50MHz。计算过程见图2-5，其中 Fin 就是晶振的频率 =12MHz。

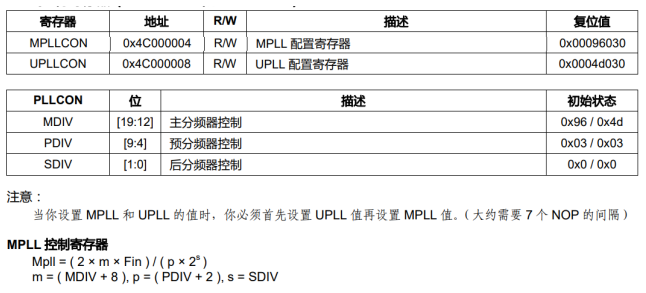


图 2-5 PLL 控制寄存器（手册第 7 章）

UPLL 的输出频率没有配置，直接给 USB 使用。

### 存储空间初始化

b l memsetup

@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@ i n i t i a l i s e t h e s t a t i c memory

@ s e t memory c o n t r o l

r e g i s t e r s

@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

memsetup :

mov r1 , #MEM\_CTL\_BASE

a d r l r2 , mem\_c fg\_val

add r3 , r1 , #52

l d r r4 , [ r 2 ] , #4

s t r r4 , [ r 1 ] , #4

cmp r1 , r 3

bne 1b

mov pc , l r

memsetup 就是初始化一下 mini2440 的存储器（BANK0 BANK7），具体自行阅读《S3C2440 中文手册》第 5 章。

### 栈初始化

b l I n i t S t a c k s

@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@ 堆 栈 初 始 化

@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

I n i t S t a c k s :

mov r2 , r

mrs r0 , c p s r

b i c r0 , r0 , #MODE\_MASK

o r r r1 , r0 , #UND\_MODE| NOINT

msr c p s r \_ c x s f , r 1

@UndefMode

l d r sp , = UDF\_stack

@ U n d e f St a c k =0x33FF\_5C00

o r r r1 , r0 , #ABT\_MODE| NOINT

msr c p s r \_ c x s f , r 1

@AbortMode

l d r sp , = ABT\_stack

@ A b o rt St a c k = 0 x33FF\_6000

o r r r1 , r0 , #IRQ\_MODE | NOINT

msr c p s r \_ c x s f , r 1

@IRQMode

l d r sp , = IRQ \_ st a c k

@ IRQStack =0 x33FF\_7000

o r r r1 , r0 , #FIQ\_MODE | NOINT

msr c p s r \_ c x s f , r 1

@FIQMode

l d r sp , = F I Q \_ st a c k

@ F IQStac k = 0 x33FF\_8000

b i c r0 , r0 , #MODE\_MASK| NOINT

o r r r1 , r0 , #SVC\_MODE

msr c p s r \_ c x s f , r 1

@SVCMode

l d r sp , = SVC\_stack

@ SVCStack = 0 x33FF\_5800

mrs r0 , cp s r

b i c r0 , r0 , #MODE\_MASK

o r r r1 , r0 , #SYS\_MODE| NOINT

msr c p s r \_ c x s f , r 1

@ use rMode

l d r sp , = SYS \_ stac k

mov pc , r 2

InitStacks 就是初始化一下 arm 处理器各个模式的栈。因为影子寄存器的存在，每个模式的 sp 寄存器都是独立的（除了系统和用户模式）。具体的做法就是通过修改 cpsr 寄存器的 [5:0] 位，进入每一个模式并修改 sp 寄存器位该模式的栈底地址。每个模式的栈底地址定义于链接脚本。链接脚本将在下一节进行讲解。cpsr 寄存器各位编排如图2-6。

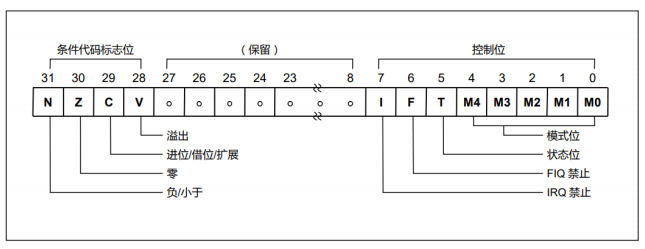


图 2-6 程序状态寄存器格式（手册第 2 章）

PS：关于影子寄存器（也称分组寄存器），请查看《S3C2440 中文手册》第 2章图 2-3。

### 自我拷贝（loader）

a d r r0 , \_\_ENTRY

l d r r1 , \_ t e x t \_ s t a r t

cmp r0 , r 1

bl n e c o p y \_ s e l f

c o p y \_ s e l f :

l d r r1 , = ( (4 < <28 ) | ( 2 < < 4 ) | ( 3 < < 2 ) ) /\* a d d r e s s o f I n t e r n a l SRAM

0x4000002C\* /

mov r0 , #0

s t r r0 , [ r 1 ]

Mov r1 , #0 x2c

/\* a d d r e s s o f men 0x0000002C\* /

l d r r0 , [ r 1 ]

Cmp r0 , #0

bne copy\_ f rom\_ rom

l d r r0 , = ( 2 4 4 0 )

l d r r1 , = ( (4 < <28 ) | ( 2 < < 4 ) | ( 3 < < 2 ) )

s t r r0 , [ r 1 ]

b c o p y \_ f r om \_ na n d

这一步非常重要。如果 aCoral 是烧写在 nor flash 中，启动时是从 0 地址开始运行的，那 aCoral 怎么到内存 SDRAM（起始地址为 0x30000000）中运行呢？这项任务是由 aCoral 自己来完成的，也就是我搬起了我自己。我们先想一下，aCoral 在上电运行的时候，怎么知道自己是在 flash 还是 sdram中运行的呢？答案是看一下现在自己运行的地址是多少就行了，换句话说就是查看 pc 寄存器的值。基于这种思路，aCoral 使用了一种更严谨的做法：使用 adr 相对地址指令。

通过查看反汇编，

a d r r0 , \_\_ENTRY

被汇编成

s u b r0 , pc , #188

这就表示，无论程序在哪里运行，r0 永远是程序的实际起始地址。比如说，当程序从 0 地址的 flash 开始运行，那 r0 的值就等于 0；而当程序在 0x30000000 的sdram 中运行时，r0 寄存器就等于 0x30000000。

l d r r1 , \_ t e x t \_ s t a r t

而 \_text\_start 处存放的是定值 0x30000000，所以 r1=0x30000000。这样通过比较 r0 与 r1 寄存器，如果 r0=r1=0x30000000，就说明程序当前正在 sdram 中运行,就不要 copy\_self；反之则需要。copy\_self 的过程自行阅读。

### bss 段清零

l d r r0 , \_ b s s \_ s t a r t

l d r r1 , \_ b s s \_ e n d

b l mem\_clea r

. . . . . . .

@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@ c l e a r memory

@ r 0 : s t a r t a d d r e s s

@ r 1 : l e n g t h

@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

mem\_clea r :

mov r2 , # 0

1: s t r r2 , [ r 0 ] , # 4

cmp r0 , r 1

b l t 1b

mov pc , l r

bss 段中的数据都是未初始化或者初始值为 0 的，而 sdram 本身是有一些随即初始值的，所以需要对 bss 段对应的内存进行清零。mem\_clear 的两个参数分别为bss 段的起始地址和结束地址，都定义在链接脚本中。

### 跳转至下一阶段

l d r pc , = a c o r a l \_ s t a r t

代码最终将寄存器 pc 设置为 acoral\_start 的值，表示 CPU 将跳转到 acoral\_start函数处执行。acoral\_start 函数位于

k e r n e l \ s r c \ c o r e . c

## 启动-第二阶段

硬件初始化完成后，就需要对系统的软件部分进行初始化。现在我们来具体看一下，aCoral 内核启动的第二阶段到底做了什么。

### acoral-start

core.c 中的 acoral\_start() 函数如下

a c o r a l \_ t h r e a d \_ t

o r i g \_ t h r e a d ;

v oi d

a c o r a l \_ s t a r t ( ) {

o r i g \_ t h r e a d . c o n s o l e \_ i d =ACORAL\_DEV\_ERR\_ID;

a c o r a l \_ s e t \_ o r i g \_ t h r e a d (& o r i g \_ t h r e a d ) ;

/\* 板 子 初 始 化 \* /

HAL\_BOARD\_INIT ( ) ;

/\* 内 核 模 块 初 始 化 \* /

a c o r a l \_ m o d u l e \_ i n i t ( ) ;

/\* 串 口 终 端 应 该 初 始 化 好 了， 将 根 线 程 的 终 端 i d 设 置 为 串 口 终 端 \* /

# i f d e f CFG\_DRIVER

o r i g \_ t h r e a d . c o n s o l e \_ i d = a c o r al \_ d e v \_ o p e n ( ” c o n s o l e ” ) ;

# e n d i f

/\* 主 cpu 开 始 函 数 \* /

a c o r a l \_ c o r e \_ c p u \_ s t a r t ( ) ;

}

可以看到 acoral\_start 代码并不多，概括一下就是三件事：

(1) 内核模块初始化。

这部分代码中，最重要的就是 acoral\_module\_init() 这个函数。这个函数将初始化 aCoral 的各个模块，包括中断、内存、线程等嵌入式操作系统必需的模块。

v oi d a c o r a l \_ m o d u l e \_ i n i t ( ) {

/\* 中 断 系 统 初 始 化 \* /

a c o r a l \_ i n t r \_ s y s \_ i n i t ( ) ;

/\* 内 存 管 理 系 统 初 始 化 \* /

a c o r a l \_ m e m \_ s y s \_ i n i t ( ) ;

/\* 线 程 管 理 系 统 初 始 化 \* /

a c o r a l \_ t h r e a d \_ s y s \_ i n i t ( ) ;

/\* 时 钟 管 理 系 统 初 始 化 \* /

a c o r a l \_ t i m e \_ s y s \_ i n i t ( ) ;

/\* 事 件 管 理 系 统 初 始 化 \* /

a c o r a l \_ e v t \_ s y s \_ i n i t ( ) ;

# i f d e f CFG\_DRIVER

/\* 驱 动 管 理 系 统 初 始 化 \* /

a c o r a l \_ d r v \_ s y s \_ i n i t ( ) ;

# e n d i f

}

(2) 设置 orig 线程。

关于 orig 线程，有两个问题：这个线程是怎么创建的呢？为什么要设置一个orig 线程呢？第一个问题比较好回答，这个时候 aCoral 的线程模块还没有初始化，创建线程的那些函数都是用不来的，所以就直接定义了一个 acoral\_thread\_t 类型的orig\_thread。这个变量类型就是 aCoral 中的 Task Control Block（TCB），在《aCoral内核手册》将会介绍。给成员 console\_id 赋值，并将这个 orig 线程设置为当前正在运行的线程，如代码：

o r i g \_ t h r e a d . c o n s o l e \_ i d =ACORAL\_DEV\_ERR\_ID;

a c o r a l \_ s e t \_ o r i g \_ t h r e a d (& o r i g \_ t h r e a d ) ;

那么为什么要设置这个 orig 线程呢？我们先看一下 console\_id 被置为了什么。

o r i g \_ t h r e a d . c o n s o l e \_ i d = a c o r al \_ d e v \_ o p e n ( ” c o n s o l e ” ) ;

这个 orig 线程的 console\_id 指向一个叫”console” 的设备。console 控制台是干什么用的？console 本质上就是 UART 串口，我们在串口工具的那个黑黑的界面上看到的信息，都是从串口线传过来的。aCoral 中我们经常见到的的函数acoral\_prints()，也是要把信息通过 UART 串口，显示到电脑的串口工具中。打开这个 console 设备，本质上就是在注册的驱动中找到 UART 串口驱动。话又说回来，那 orig 线程需要这个 console 干嘛呢？orig 线程虽然不打印或接收信息，但是它需要这个 console，是为了之后由它创建的线程（子线程、孙子线程等等）都能继承到这个 console，换句话说，它的子孙们都能找到从哪去接收信息，把信息打印打哪里去。这点在 acoral\_thread\_init() 函数，也就是创建线程的过程中有体现。具体会在《aCoral 内核手册》中介绍。

从这里也可以看出，orig 是真正意义上的，aCoral 中的第一个线程（祖宗）。

1. 调用 acoral\_core\_cpu\_start() 进行下一步初始化工作。

### acoral-core-cpu-start

v oi d a c o r a l \_ c o r e \_ c p u \_ s t a r t ( ) {

a c o r a l \_ c o m m \_ p o l i c y \_ d a t a \_ t d a t a ;

/\* 创 建 空 闲 线 程 \* /

a c o r a l \_ s t a r t \_ s c h e d = f a l s e ;

d a t a . cpu= a c o r a l \_ c u r r e n t \_ c p u ;

d a t a . p r i o =ACORAL\_IDLE\_PRIO ;

d a t a . p r i o \_ t y p e =ACORAL\_ABSOLUTE\_PRIO;

i d l e \_ i d = a c o r a l \_ c r e a t e \_ t h r e a d \_ e x t ( i d l e , IDLE\_STACK\_SIZE ,NULL, ”

i d l e ” ,NULL,ACORAL\_SCHED\_POLICY\_COMM,& d a t a ) ;

i f ( i d l e \_ i d == −1)

w hil e ( 1 ) ;

/\* 创 建 初 始 化 线程 , 这 个 调 用 层 次 比 较 多， 需 要 多 谢 堆 栈 \* /

d a t a . p r i o =ACORAL\_INIT\_PRIO ;

/\* 动 态 堆 栈 \* /

i n i t \_ i d = a c o r a l \_ c r e a t e \_ t h r e a d \_ e x t ( i n i t , ACORAL\_TEST\_STACK\_SIZE, ”

i n

i n i t ” , ” i n i t ” ,NULL,ACORAL\_SCHED\_POLICY\_COMM,& d a t a ) ;

i f ( i n i t \_ i d == −1)

w hil e ( 1 ) ;

a c o r a l \_ s t a r t \_ o s ( ) ;

}

这里，acoral\_core\_cpu\_start() 概况起来也是做了三件事：

(1) 创建 idle 线程。

orig 线程的亲儿子，继承了 orig 线程的 console。idle 线程又名守护线程，说白了就是，CPU 没事干了，又不能真的闲下来，那就去执行 idle 线程，所以 idle 线程的优先级是最低的。idle 线程的工作函数如下，可以看到，idle 线程就是一个无限循环，CPU 没事干了就跑 while 循环摆烂。

v oi d i d l e ( v oi d \* a r g s ) {

w hil e ( 1 ) {

}

}

(2) 创建 init 线程。

orig 线程的亲儿子，继承了 orig 线程的 console。init 线程做了很多初始化工作，包括时钟中断的注册、创建 daem 回收线程、shell 线程初始化、插件初始化等。这里解释一下 shell。串口之所以能接收我们的指令并返回结果，就是 shell 这个线程在工作。init 线程的工作函数如下。

v oi d i n i t ( v oi d \* a r g s ) {

a c o r a l \_ p r i n t s ( ” i n

i n i t

s p g ” ) ;

a c o r a l \_ c o m m \_ p o l i c y \_ d a t a \_ t d a t a ;

a c o r a l \_ t i c k s \_ i n i t ( ) ;

/\* t i c k s 中 断 初 始 化 函 数 \* /

a c o r a l \_ s t a r t \_ s c h e d = t r u e ;

/\* 软 件 延 时 初 始 化 函 数 \* /

# i f d e f CFG\_SOFT\_DELAY

s o f t \_ d e l a y \_ i n i t ( ) ;

# e n d i f

/\* 创 建daem回 收 线 程 \* /

a c o r a l \_ i n i t \_ l i s t (& a c o r a l \_ r e s \_ r e l e a s e \_ q u e u e . hea d ) ;

a c o r a l \_ s p i n \_ i n i t (& a c o r a l \_ r e s \_ r e l e a s e \_ q u e u e . hea d . l o c k ) ;

d a t a . cpu= a c o r a l \_ c u r r e n t \_ c p u ;

d a t a . p r i o =ACORAL\_DAEMON\_PRIO;

d a t a . p r i o \_ t y p e =ACORAL\_ABSOLUTE\_PRIO;

daemon\_id= a c o r a l \_ c r e a t e \_ t h r e a d \_ e x t ( daem , DAEM\_STACK\_SIZE,NULL, ”

daemon ” ,NULL,ACORAL\_SCHED\_POLICY\_COMM,& d a t a ) ;

t h r e a d = ( a c o r a l \_ t h r e a d \_ t \* ) a c o r a l \_ g e t \_ r e s \_ b y \_ i d ( daemon\_id ) ;

i f ( daemon\_id == −1)

w hil e ( 1 ) ;

/\* 应 用 级 相 关 服 务 初 始化 , 应 用 级 不 要 使 用 延 时 函 数， 没 有 效 果 的 \* /

# i f d e f CFG\_SHELL

a c o r a l \_ s h e l l \_ i n i t ( ) ;

# e n d i f

p l u g i n \_ i n i t ( ) ;

a p p \_ e n t e r \_ p o l i c y \_ i n i t ( ) ;

u s e r \_m ai n ( ) ;

# i f d e f CFG\_TEST

t e s t \_ i n i t ( ) ;

# e n d i f

a p p \_ e x i t \_ p o l i c y \_ i n i t ( ) ;

}

(3) 调用 acoral\_start\_os() 正式开始运行系统。

最后一件事，无非就是初始化完成了，开始调度了，这个时候 aCoral 就正式开始运行了，一去不复返……

# aCoral 内存分布

前一章中，多次提到了 aCoral 最终会在 0x30000000 这个地址运行，也就是内存 sdram 所在的地址。那为什么一定是这个地址呢？aCoral 在执行 copy\_self 的时候，copy 到 0x30000001 行不行？要解释这个问题，我们要先了解两个概念：存储地址和运行时地址。

## 存储地址和运行时地址

存储地址也可以叫加载地址。因为我们的程序是要存储在非易失性存储器上的，比如 nor flash 和 nand flash。程序存储在这些存储器的地址就叫做存储地址。比如 aCoral 被烧写到 nand flash 的 0 地址后，它的存储地址就是从 0 开始的一连串地址。

运行时地址是程序运行时应该在的地址，注意是应该，表示程序最好在这个地址运行。如果程序全部由位置无关代码构成，那即使不在运行时地址运行，程序也不会出错。但是只要有位置相关代码，程序就 g 了。运行时地址表示程序人员对代码运行时的一种期待和要求。aCoral 在 mini2440 开发板的运行时地址为0x30000000，所以当我们把 flash 中的 acoral 整个复制到 0x30000000 之后的区域后，我们就说 aCoral 此时已经来到了它的运行时地址。

不同的开发板，内存的地址不同，aCoral 的运行时地址就会不同。mini2440就是 0x30000000。那我们在哪里指定这个地址呢？这就是链接脚本做的事了。当然链接脚本不仅指定了运行时地址，还决定了整个 aCoral 镜像的结构，也就是aCoral 被复制到 sdram 中之后的内存分布。链接脚本位于

h a l \ s 3c 2 4 4 0 \ a c o r a l . l d s

## aCoral 链接脚本

链接脚本是给链接器看的。链接器会按照链接脚本定义的规则，把 aCoral 源码中编译出来的一个个可重定位文件 (.o) 进行链接，最终得到 acoral.bin。链接脚本内容如下：

ENTRY(\_\_ENTRY)

MEMORY

{

ram (wx ) : o r g = 0 x030000000 ,

l e n = 64M

}

SECTIONS

{

. t e x t : {

t e x t \_ s t a r t = . ;

\* ( . t e x t )

\* ( . i n i t . t e x t )

\* ( . r o d a t a \* )

}>ram

. d a t a ALIGN( 4 ) :

{

\* ( . a c o r a l 1 . c a l l )

\* ( . a c o r a l 2 . c a l l )

\* ( . a c o r a l 3 . c a l l )

\* ( . a c o r a l 4 . c a l l )

\* ( . a c o r a l 5 . c a l l )

\* ( . a c o r a l 6 . c a l l )

\* ( . a c o r a l 7 . c a l l )

\* ( . a c o r a l 8 . c a l l )

\* ( . a c o r a l 9 . c a l l )

\* ( . a c o r a l 1 0 . c a l l )

\* ( . d a t a )

\* ( . d a t a . r e l )

\* ( . g ot )

\* ( . g ot . p l t )

} >ram

. b s s ALIGN( 4 ) :

{

b s s \_ s t a r t = . ;

\* ( . b s s )

. = ALIGN( 4 ) ;

} >ram

b s s \_ e n d = . ;

s t a c k \_ b a s e = 0 x 3 3 f f f f 0 0 ;

MMU\_base = 0 x 3 3 f 0 0 0 0 0 ;

S Y S \_ st a c k \_ si z e = 0 x200 ;

S VC \_ st a c k \_ si z e = 0 x200 ;

U n d e f \_ s t a c k \_ s i z e = 0 x100 ;

A b o r t \_ s t a c k \_ s i z e = 0 x100 ;

I R Q \_ st a c k \_ si z e = 0 x200 ;

F I Q \_ s t a c k \_ s i z e = 0x0 ;

F I Q \_ st a c k = s t a c k \_ b a s e ;

IRQ \_ st a c k = F I Q \_ st a c k − F I Q \_ s t a c k \_ s i z e ;

ABT\_stack = IRQ \_ st a c k − I R Q \_ st a c k \_ si z e ;

UDF\_stack = ABT\_stack − A b o r t \_ s t a c k \_ s i z e ;

SVC\_stack = UDF\_stack − U n d e f \_ s t a c k \_ s i z e ;

SYS \_ stac k = SVC\_stack − S VC \_ st a c k \_ si z e ;

h e a p \_ s t a r t = ( b s s \_ e n d + 3 ) &( ~ 3 ) ;

hea p \_e n d = MMU\_base − 0 x1000 ;

}

ENTRY 关键字定义了整个源码的入口就是标号 \_ENTRY。这个标号定义在

h a l \ s 3c 2 4 4 0 \ s r c \ s t a r t . S

所以 start.S 中 \_ENTRY 标号处的

b R e s et H a n dl e r

就是 aCoral 上电后执行的第一行代码。

MEMORY 关键字将目标开发板的内存 sdram 分块，这里只分为一块区域，名为 ram，起始地址就是 0x30000000。

SECTIONS 关键字中，镜像文件将被分段，包括我们通常说的代码段.text，数据段.data，未定义数据段.bss 等。.text 段的最后使用“>ram”指定了.text 段将被放在内存的 ram 区域，也就是说代码段理论上应该在 0x30000000 的地址开始运行，这就是运行时地址的由来。之后的数据段和 bss 段也都是放在 ram 这个内存区域，紧接着代码段放置。

之后还定义了 arm 各个模式的栈空间，最后定义了 heap 堆的地址空间。这里heap 堆的起始地址 heap\_start 和终止地址 heap\_end，其实就是 aCoral 可用的内存空间，大小大概是 64M 不到一点。aCoral 下载到内存后的整个内存分布的具体细节如图3-1所示。

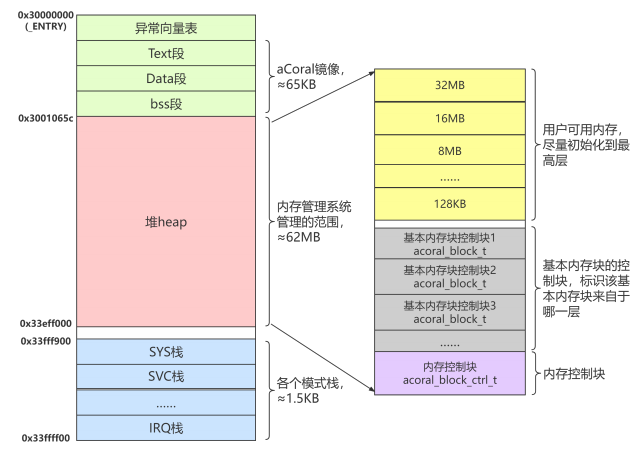


图 3-1 aCoral 内存分布