电 子 科 技 大 学

嵌入式智能计算研究团队

珊瑚-II 启动过程

ACORAL-II BOOT PROCESS

版本号　　1.0

修订历史

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-II 启动手册编写完成	2022.06.28	胡博文

目 录

第一章 BootLoader

1.1 什么是 BootLoader

我们将 BootLoader 拆开来看，一个是 Boot，一个是 Loader。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；Loader 的意思和字面一样，就是加载的意思。那自然就可以引出五个问题：

1. 加载什么?（what）

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。

2. 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。

3. 从哪里加载?（from）

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。

4. 加载到哪里去?（to）

加载到内核要运行的地方——内存，常见的有 SDRAM、DDR 等。

5. 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，compactcard 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。

1.2 Loader

关于为什么要加载这个问题，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，还有一种是内存映射。先来说说硬件方式和软件方式：

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。

最后是内存映射，内存映射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。

第二章 aCoral 启动

2.1 平台启动

zynq7020 有 4 种启动方式，分别是 JTAG、NAND FLASH、QSPI FLASH 以及 SD CARD，而正点原子只适配了三种，不支持 NAND FLASH 启动。若 zynq7020 是从 QSPI FLASH 和 SD CARD 启动，那么其启动时会先运行一段固化程序 bootrom 来引导 fsbl 代码至内部 ram，然后接下来由 fsbl 代码引导操作系统镜像文件至 DDR3 内存空间运行；若它从 JTAG 模式下启动，则进行一段简单初始化后，直接把操作系统镜像文件引导到 DDR3 上运行。具体细节请参考 ZYNQ 官方文档 UG585，更详细的内容诸如 bootrom 运作流程，fsbl 工程流程等请参考 ZYNQ 官方文档 UG821。

2.2 操作系统启动

上一章中已经详细讲解了 BootLoader，上一小节中平台启动其实做的就是 BootLoader 的一部分内容，接下来需要做的事情就和操作系统代码直接相关了。

xilinx 提供了完善的底层硬件库，因此可以通过 IDE 直接配置想要的底层设计来得到空的裸机工程，移植工作就可以在此基础上进行。xilinx 把启动部分的代码分为了许多文件，比较重要的就是 boot.s 与 asm_vector.s。其中 asm_vector.s 就是中断向量表相关代码，boot.s 则是具体的初始化过程。boot.s 在确认了当前 cpu 后，首先设置 vector_table 地址，其中 vector_base 即是主核的中断向量表基地址，代码如下：

```
1 ldr r0, =vector_base
2 mcr p15, 0, r0, c12, c0, 0
```

接下来则是关闭 SCU，SCU 全称为 Snoop Control Unit，用来保持双核之间的数据 Cache 的一致性。第一个 A9 处理器写存储时，只是写在了缓存里，没有进主存，如果第二个 A9 读操作，涉及到第一个写脏了的数据段，SCU 要保证第二个 A9 的缓存里是最新的数据。如果第二个 A9 写同样数据段的数据，需要在第一个中体现出写的内容。SCU 的存在，才使得两个核成互相联系的“双核”，才能成为 MPsoC（acoral 目前没有开启缓存，所以没有用上 SCU）。其代码如下：

```
1 /*invalidate scu*/
```

```
2 ldr r7, =0xf8f0000c
3 ldr r6, =0xffff
4 str r6, [r7]
```

然后是关闭 cache 和 MMU，代码如下：

```
1 /* Invalidate caches and TLBs */
2 mov r0,#0
3 mcr p15, 0, r0, c8, c7, 0
4 mcr p15, 0, r0, c7, c5, 0
5 mcr p15, 0, r0, c7, c5, 6
6 bl invalidate_dcache
7
8 /* Disable MMU, if enabled */
9 mrc p15, 0, r0, c1, c0, 0
10 bic r0, r0, #0x1
11 mcr p15, 0, r0, c1, c0, 0
```

再后面比较关键的地方就是栈空间的初始化。栈其实一般是在链接脚本中定义的一段内存空间，我们只需要让处理器知道地址即可，但是 ARM 拥有很多种模式，各个模式在正常情况下都需要栈空间来保存一些类似函数跳转前的状态信息，像 pc 值、寄存器值、cpsr 值等等，而 ARM 处理器切换模式也需要操作，所以需要分模式来依次进行栈的初始化。这里示例代码为 SYS 模式下的栈初始化，代码如下：

```
1 mrs r0, cpsr /* get the current PSR */
2 mvn r1, #0x1f /* set up the system stack pointer
    */
3 and r2, r1, r0
4 orr r2, r2, #0x1F /* SYS mode */
5 msr cpsr, r2
6 ldr r13,=SYS_stack /* SYS stack pointer */
```

剩下的是一些处理器上的设置，类似什么流控制、vfp、Auxiliary Control、异步中止异常等，最后跳转到 c 语言函数 _start，这个函数在进行一些 xilinx 库函数初始化后就会跳转到 main。

2.3 多核启动

zynq7020 启动时，主核所运行的流程就如上一小节所写，而次核则在上电之后循环等待主核发送启动事件来进行启动，即 wfe 模式下。主核发送事件的指令是 sev，这是一条 ARM 汇编指令，次核在接收到启动信号后会读取 0xFFFFFFF0 地址上的值，来作为下一条指令的 PC 值，所以主核在发送 sev 之前需要把次核的启动代码起始地址写入 0xFFFFFFF0。

主核在通过 boot.s 中的代码进入 main 函数后，就开始了一些基本外设的初始化，最后进入 acoral_ start 函数中；次核也是需要运行一段 boot 代码的，这段代码需要自己编写，代码内容就不如主核那么丰富，也不需要进行更多操作，次核的 boot 只需要初始化堆栈、设置 vector_table 地址，关掉 MMU 与 Cache，然后直接跳转到 acoral_start 函数，值得一提的是，次核的堆栈空间需要改写链接脚本来进行定义。

acoral_start 函数首先使用了一个静态变量来进行两个 cpu 的区分，方便两个核都能使用此函数作为操作系统启动入口，其次就是主核和次核分别执行不同的内容了。主核运行一些内存、中断、线程池等初始化后就准备启动次核，而次核就是创建一个 idle_follow 线程后响应主核，主核接收到回应后就开始调度线程。代码如下：

```
void acoral_start(){
#ifdef CFG_CMP
    static int core_cpu=1;
    if(!core_cpu){
        acoral_set_orig_thread(&orig_thread);
        acoral_follow_cpu_start();
    }
    core_cpu=0;
    HAL_CORE_CPU_INIT();
#endif
    orig_thread.console_id=-1;
    acoral_set_orig_thread(&orig_thread);
    acoral_module_init();
#ifdef CFG_CMP
    acoral_cmp_init();
```

```
16 #endif
17     acoral_core_cpu_start();
18 }
```

第三章 aCoral 内存分布

内存分布见图 3-1

Table 4-1: **System-Level Address Map**

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000_0000 to 0003_FFFF[2]	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU[3]
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF[4]	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF[2]	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

图 3-1 aCoral-II 内存分布

先了解两个概念：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 norflash、nandflash 或 sd 卡。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 nandflash 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 zynq 开发板的运行时地址为 ddr 内存的起始地址 0x100000，所以当我们把 flash 或 sd 卡中的 acoral 整个复制到之后的区域后，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。zynq 就是 0x100000。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。

xilinx 的 IDE 会帮助我们生成初始的链接文件，由于使用到了多核，所以需要进行一些修改，链接文件内容太长，请自行打开文件查看。里面包含了 ddr、heap、stack，还有一些文件段的映射，主要修改的部分就是增加了多核的栈空间。