|  |
| --- |
| **黑色：正文内容**  **红色：要剔除的代码**  **黄色：这段代码的学习目的**  **蓝色：要提供的数据结构、api或者文档等** |

# 嵌入式操作系统基础

## 嵌入式系统简介

嵌入式系统是以应用为中心，以计算机技术为基础，软硬件可配置，对功能、可靠性、成本、体积、功耗有严格约束的专用系统。

* **嵌入式系统具有以下特点**：

1. 实时性　　环境特定；快速启动
2. 异步事件的并发处理 多任务；随机性
3. 应用／操作系统一体化 VxWorks；Linux
4. 应用固化　　不可修改性
5. 实用性　面向行业；定向开发
6. 适用性　可裁减，适应应用
7. 鲁棒性　容错
8. 够用即可　成本、资源
9. 可信性　安全、防危、可靠等

* 嵌入式系统基本结构



图 .1 嵌入式系统基本结构

## 嵌入式软件系统简介

在早期以及现在的一些特定嵌入式系统，还没有图1.1中的嵌入式操作系统，只有一些更为简单的软件系统，包括轮询系统、前后台系统等。而这门课要研究的就是上图1.1中的嵌入式操作系统部分，接下来所有的内容都将围绕其嵌入式操作系统展开。为了对嵌入式操作系统有一个更系统的认识，接下来会介绍包括嵌入式操作系统在内的几种常见的嵌入式软件系统。

### 轮询系统

即在裸机编程时，先初始化相关硬件，让主程序在一个死循环里面不断循环，顺序地处理各种事件。不能说轮询是低端的，轮询系统是一种非常简单的软件结构，但适用于仅需要顺序执行代码且不需要外部事件来驱动就能完成的事情，这会变得简单可靠。

缺点：如果加入按键操作等需要检测外部信号的事件，整个系统的实时响应能力就会体现不好。试想一下，但按键按下时，程序正在运行顺序1程序，而且顺序1程序占用的程序时间片比较长，系统就有可能错过对按键的检测（直到按键松开），实时性极差，用户无法接受。

轮询系统程序框架如下：

|  |
| --- |
| int main(){  /\* 硬件初始化 \*/  HardWareInit();  /\* 进入轮询 \*/  while(true){  fun1(); //顺序程序1  fun2(); //顺序程序2  fun3(); //顺序程序3  }  return 0;  } |

代码1.1

### 前后台系统

相比轮询系统，前后台系统在轮询系统的基础上加入了中断的概念，外部事件的响应在中断里面完成，事件的处理还是回到轮询系统中完成，中断我们称之为前台，main()函数中的无限循环称为后台。

在顺序执行后台程序时，如果有中断，那么中断会打断后台程序的正常执行流，转而去执行中断服务程序，在中断服务程序中标记事件。如果事件要处理的事情很简短，则可在中断服务程序里面处理，如果事件要处理的事情比较多，则返回后台程序处理。通过中断可以大大提供程序的实时响应能力，避免造成外部事件的丢失。

前后台系统程序框架如下：

|  |
| --- |
| /\* 前台中断服务程序 \*/  void isr(){}  /\* 后台轮询程序 \*/  int main(){  /\* 硬件初始化 \*/  HardWareInit();  /\* 注册中断 \*/  isrregister(isr);  /\* 进入轮询 \*/  while(true){  fun1(); //顺序程序1  fun2(); //顺序程序2  fun3(); //顺序程序3  }  return 0;  } |

代码1.2

### 嵌入式操作系统

由于轮询系统和前后台系统存在的不足，且随着嵌入式系统复杂性的增加，这时轮询系统或者前后台系统已经很难满足用户对功能和性能的要求。因此，工程师们设计了嵌入式操作系统，以解决事务的不可剥夺、不可阻塞性，实现多任务的并发执行。嵌入式操作系统（Embedded Operating System，简称：EOS）是指用于嵌入式系统的操作系统。嵌入式操作系统是一种用途广泛的系统软件，通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面、标准化浏览器等。嵌入式操作系统负责嵌入式系统的全部软、硬件资源的分配、任务调度，控制、协调并发活动。它必须体现其所在系统的特征，能够通过装卸某些模块来达到系统所要求的功能。目前在嵌入式领域广泛使用的操作系统有：嵌入式实时操作系统µC/OS-II、嵌入式Linux、Windows Embedded、VxWorks等，以及应用在智能手机和平板电脑的Android、iOS等。

### 实时操作系统

什么是实时操作系统？ 实时操作系统（RTOS）是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程或对处理系统作出快速响应，并控制所有实时任务协调一致运行的操作系统。 其特点是及时响应和高可靠性。由于嵌入式操作系统及其应用软件往往被嵌入到特定的控制设备或者仪器中，用于实时响应并处理外部事件，所以嵌入式操作系统也往往被称为实时操作系统。另一方面，由于RTOS往往也存在于嵌入式系统中，因此约定，下文提到的RTOS代表的就是实时操作系统或者嵌入式实时操作系统。

* 广义上，RTOS≈EOS

### RTOS与通用OS比较

|  |  |  |
| --- | --- | --- |
|  | RTOS | 通用OS |
| 体系结构 | 整体、层次、微内核、构件化 | 整体、层次、微内核、构件化 |
| TCB结构 | 增加时间参数：周期、截止时间等 | 名称、ID、优先级、当前状态、上下文等 |
| 任务调度 | 静态调度表、固定优先级（RM类）、动态优先级（EDF类）；原则上用户不能改变优先级 | 基于优先级的调度；时间片轮转调度；系统态优先于用户态；优先级可用户改变 |
| 内存管理 | 主要使用静态内存划分 | 采用虚拟存储机制 |
| 中断管理 | 多数中断处理转化为周期性查询任务；执行次序由统一考虑的优先级决定 | 中断处理优先于任务执行 |
| 体系结构 | 整体、层次、微内核、构件化 | 整体、层次、微内核、构件化 |
| TCB结构 | 增加时间参数：周期、截止时间等 | 名称、ID、优先级、当前状态、上下文等 |
| 任务调度 | 静态调度表、固定优先级（RM类）、动态优先级（EDF类）；原则上用户不能改变优先级 | 基于优先级的调度；时间片轮转调度；系统态优先于用户态；优先级可用户改变 |
| 内存管理 | 主要使用静态内存划分 | 采用虚拟存储机制 |
| 中断管理 | 多数中断处理转化为周期性查询任务；执行次序由统一考虑的优先级决定 | 中断处理优先于任务执行 |

## 硬件简介

上一节介绍了嵌入式系统中的软件部分，本节将对与之对应的硬件部分进行介绍，包括使用的开发板概况、2440处理器寄存器内容以及开发环境的搭建。

### mini2440简介

本门课程将使用mini2440作为硬件开发环境，对应图1.1中的外围设备和嵌入式处理器部分。本小节需要结合《s3c2440中文手册》学习。

* Mini2440外观

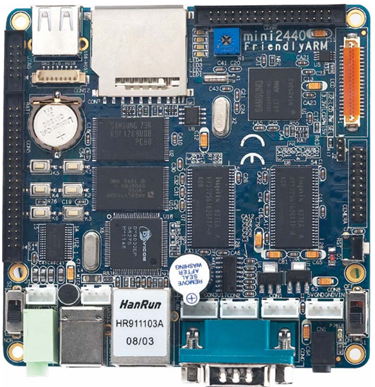


图1.2 mini2440外观

* Mini2440硬件资源介绍

**CPU 处理器**

- Samsung S3C2440A，主频 400MHz，最高 533Mhz

**SDRAM 内存**

- 在板 64M SDRAM

- 32bit 数据总线- SDRAM 时钟频率高达 100MHz

**FLASH 存储**

在板 256M/1GB Nand Flash, 掉电非易失(用户可定制 64M/128M/256M/512M/1G)

在板 2M Nor Flash，掉电非易失，已经安装 BIOS

**LCD 显示**

- 板上集成 4 线电阻式触摸屏接口，可以直接连接四线电阻触摸屏

- 支持黑白、4 级灰度、16 级灰度、256 色、4096 色 STN 液晶屏，尺寸从 3.5 寸到 12.1 寸，屏幕分辨率可

以达到 1024x768 象素；

- 支持黑白、4 级灰度、16 级灰度、256 色、64K 色、真彩色 TFT 液晶屏，尺寸从 3.5 寸到 12.1 寸，屏幕

分辨率可以达到 1024x768 象素；

- 标准配置为统宝 3.5” 真彩 LCD，分别率 240x320，带触摸屏；

**接口和资源**

- 1 个 100M 以太网 RJ-45 接口(采用 DM9000 网络芯片)

- 3 个串行口

- 1 个 USB Host

- 1 个 USB Slave B 型接口

- 1 个 SD 卡存储接口

- 1 路立体声音频输出接口，一路麦克风接口；

- 1 个 2.0mm 间距 10 针 JTAG 接口

- 4 USER Leds

- 6 USER buttons(带引出座)

- 1 个 PWM 控制蜂鸣器

- 1 个可调电阻，用于 AD 模数转换测试

- 1 个 I2C 总线 AT24C08 芯片，用于 I2C 总线测试

- 1 个 2.0 mm 间距 20pin 摄像头接口

- 板载实时时钟电池

- 电源接口(5V)，带电源开关和指示灯

**系统时钟源**

- 12M 无源晶振

**实时时钟**

- 内部实时时钟（带后备锂电池）

**扩展接口**

1 个 34 pin 2.0mmGPIO 接口

1 个 40 pin 2.0mm 系统总线接口

**规格尺寸**

- 100 x 100(mm)

**操作系统支持**

- Linux2.6.32.2 + Qtopia-2.2.0+QtE-4.6.1(独创双图形系统共存，无缝切换)

- WindowsCE.NET 6.0(R3)

### 简单裸板程序编写

这一小节，我们先来写一个简单的裸板程序，来熟悉一下mini2440开发板上寄存器的操作。参考s3c2440中文手册，写一个程序，要求如下：**按下开发板上的按键K1，两秒后LED1亮起。**

* 获取按键K1状态

首先，我们需要知道我们要从哪个寄存器里，去获得K1的状态，这需要查看mini2440原理图，如图1.3和1.4：

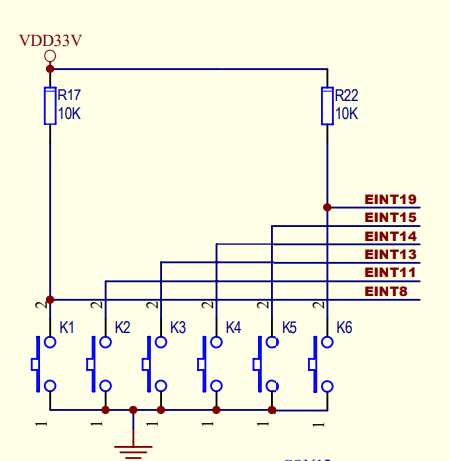


图1.3

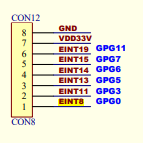


图1.4

由上可知，K1对应EINT8，EINT对应GPG0，至于EINT8是什么，后面会讲。2440中的GPA、GPB一直到GPJ，都为通用输入输出端口（General Purpose Input/Output Port）。GPA为一组，GPB为一组，以此类推。GPG0指的是GPG这一组中的第0个端口。在s3c2440中文手册中查找GPG0，可以找到如下内容，如图1.5：

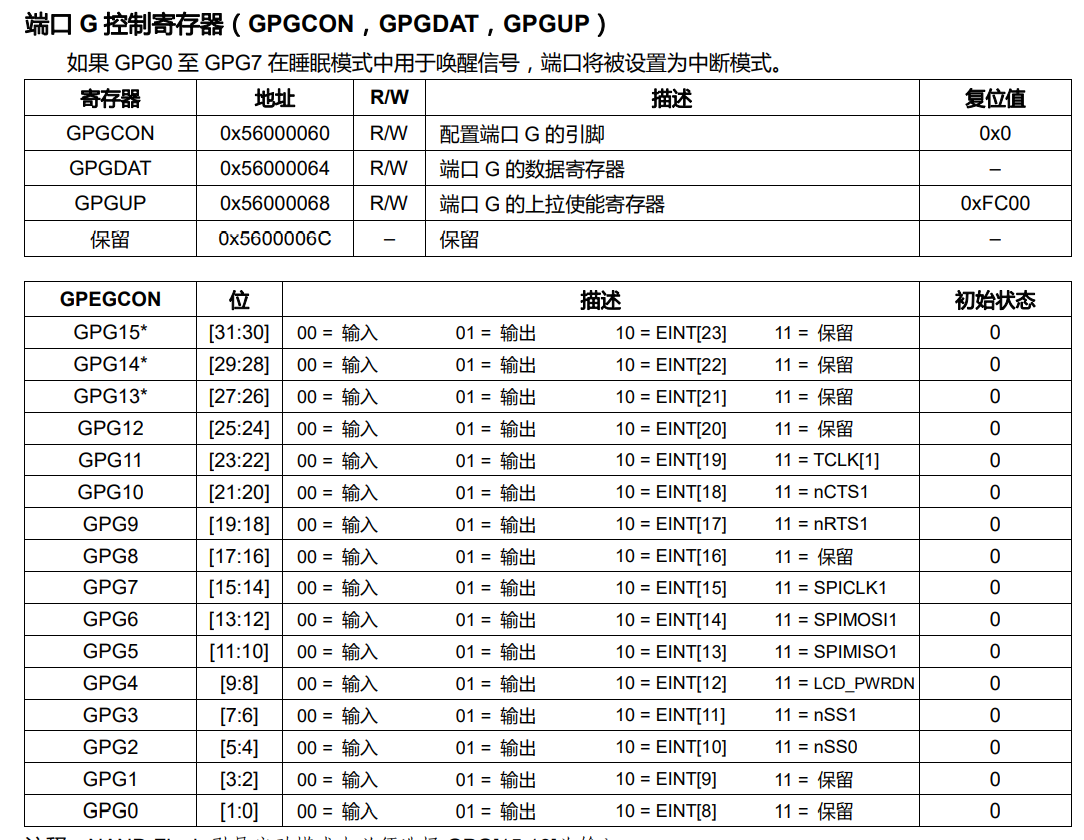


图1.5

我们需要把GPGCON寄存器的[1：0]位写为00，就可以将按键K1连接的引脚定义为输入引脚，我们就可以通过读取GPGDAT寄存器来获取K1的状态，由图1.3可知，按下为低电平0，松开为高电平1。

根据以上描述，写出按键K1初始化和状态判断代码

|  |
| --- |
| rGPGCON = 0;  while(1)  {  if((rGPGDAT & 0x1)==0)  break;  } |

代码1.3

* 定时器计时

按键K1按下两秒后，灯亮。这里的两秒延时，可以通过mini2440开发板上的定时器来实现。众所周知，开发板上所有的设备都需要依靠时钟信号来协同工作，就像跑操的时候大家一起喊“一二一“一样。定时器也不例外，它依赖于开发板上的的PCLK时钟信号，关于PCLK的详细知识，会在第七章介绍，这里只需要知道，经过代码1.4的设置后，PCLK频率为50MHz。

|  |
| --- |
| rCLKDIVN = 0X5;  rMPLLCON = (0X7f<<12) | (0X2<<4) | (0X1); |

代码1.4

有了PCLK作为定时器的输入频率，我们就可以配置定时器了。mini2440上有5个定时器，这里我们选用定时器0。定时器的工作方式大致可以概括为：我们先给定时器指定一个计数值，定时器开始工作后，便开始在每一个时钟信号来临时，将计数值减一，直到减到0。而我们也有一个窗口，可以观察计时器内部的计数值减到多少了。具体到步骤如下：

**第一步**。首先我们要给定时器一个输入频率。这里的输入频率并不是直接使用PCLK，而是要经过一些分频处理，得到一个较低的频率供定时器使用。定时器输入频率配置如图1.6



图1.6

这里我们对定时器0的预分频值选择为249，于是往TCFG0寄存器写入0xF9。分频值选择为16，所以往TCG1寄存器写入0x3。这样，定时器0的输入时钟频率=50MHz/（249+1）/16=25000MHz，即定时器0每秒会收到25000个时钟频率。

**第二步**。这时我们就要给定时器0一个计数值用来倒数。计数值的设置如图1.7

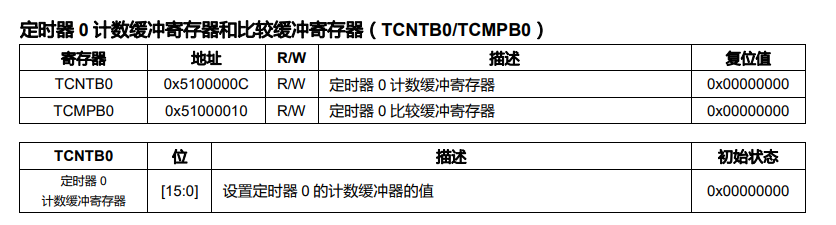


图1.7

因为我们要的是2秒的延时，而定时器0每秒收到25000个时钟信号，每行收到一个时钟信号计数值减一，所以我们要将计数值设为50000，这样减到0就正好是2秒。于是往寄存器TCNTB0写入0xC350。

**第三步**。有一点要注意，mini2440中，定时器0实际上减的是TCNT0寄存器的值，TCNTB0只不过是一个缓冲器，我们在计时开始前，需要手动将TCNTB0的值写入TCNT0中。如图1.8



图1.8

由图1.8可知，我们可以通过将寄存器TCON的[1]位置为1，来将TCNTB0的值写入TCNT0，于是我们往寄存器TCON写入0x2。

**最后一步**，就是要开始计时了。由图1.8可知，我们通过将寄存器TCON的[0]位置为1，就可以启动定时器0。所以我们往寄存器TCON写入0x1

根据以上描述步骤，写出定时器0的相关代码。

|  |
| --- |
| rTCFG0 |= 0xF9; /\* prescaler等于249\*/  rTCFG1 |= 0x3; /\*divider等于16\*/  rTCNTB0 = 0XC350; /\*计数值50000\*/  rTCON = rTCON & (~0xf) |0x02; /\* 更新TCNTB0到TCNT0\*/  rTCON = rTCON & (~0xf) |0x01; /\* start\*/ |

代码1.5

还有一点需要注意，因为TCNT寄存器我们不能直接观察它的值，所以我们需要一个窗口来观察，这个窗口就是TCNTO0计数器监视寄存器。如图1.9

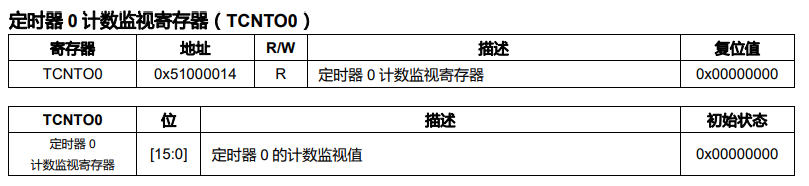


图1.9

写出读取TCNTO0来获取计数器值的代码

|  |
| --- |
| while(1){  if(rTCNTO0 ==0)  break;  } |

代码1.6

* 点亮LED1

两秒延时结束后，就要点亮LED了。同样先查看mini2440原理图，如图1.10



图1.10

由图可知，led1对应GPB5（另外三个LED分别对应GPB6\7\8，可自行考证），在s3c2440中文手册查找GPB5，可以找到如下内容：

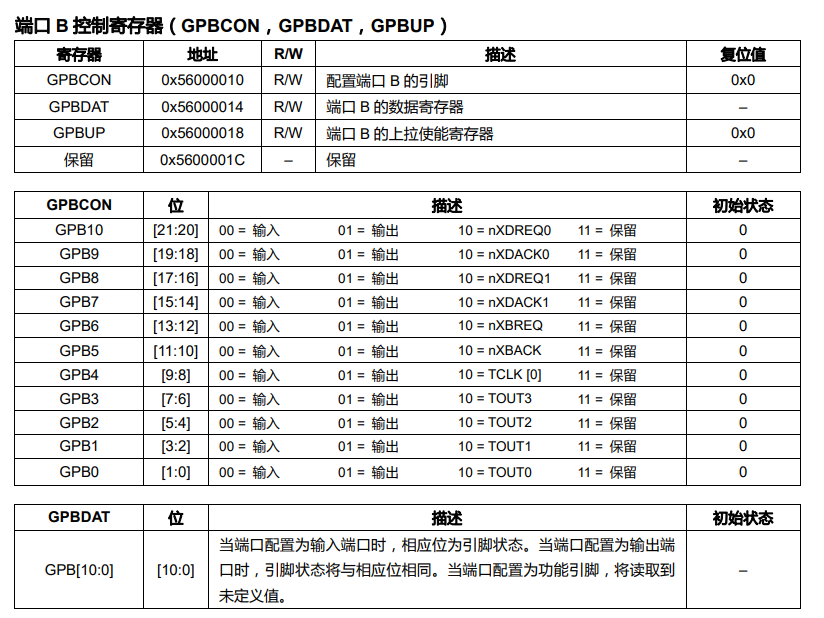


图1.11

和按键K1不同的是，LED1是要发光的，所以要将GPB5设置为输出端口。再根据图1.12

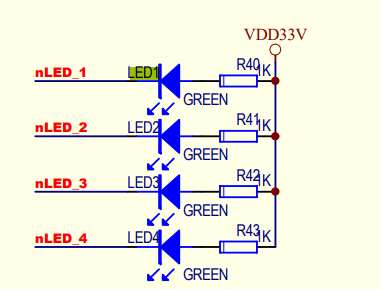


图1.12

nLED\_1这端要为低电平，才能使led1发光，所以要往GPBDAT中的最低位写0,而其他三个led对应的位要写1。根据以上描述，写出led1配置和发光代码。

|  |
| --- |
| rGPBCON = 0x400;  rGPBDAT = 0x1C0; |

代码1.7

* 总结

将代码1.3~1.7组合起来，就可以得到所需的完整的程序，如代码1.8

|  |
| --- |
| int main()  {  rCLKDIVN = 0X5;  rMPLLCON = (0X7f<<12) | (0X2<<4) | (0X1);  rGPGCON = 0;  while(1)  {  if((rGPGDAT & 0x1)==0)  break;  }  rTCFG0 |= 0xF9; /\* prescaler等于249\*/  rTCFG1 |= 0x3; /\*divider等于16\*/  rTCNTB0 = 0XC350; /\*计数值50000\*/  rTCON = rTCON & (~0xf) |0x02; /\* 更新TCNT0\*/  rTCON = rTCON & (~0xf) |0x01; /\* start\*/  while(1){  if(rTCNTO0 ==0)  break;  }  rGPBCON = 0x400;  rGPBDAT = 0x1C0;  } |

代码1.8

### S3C2440简介·

上文提到mini2440使用S3C2440芯片作为SOC，而S3C2440又使用ARM920T处理器。下面就来简单介绍一下S3C2440。

* **S3C2440运行模式** （详见s3c2440中文手册）

ARM920T 支持 7 种运行模式：

1. 用户（usr）：正常 ARM 程序执行状态
2. 快中断（fiq）：为支持数据传输或通道处理设计
3. 中断（irq）：用于一般用途的中断处理
4. 管理（svc）：操作系统保护模式
5. 中止（abt）: 数据或指令预取中止后进入
6. 系统（sys）：操作系统的特权用户模式
7. 未定义（und）：执行了一个未定义指令时进入

模式的改变可由软件控制，或者由外部中断或进入异常引起。大部分应用程序都将在用户模式执行。 被称为 特权模式的非用户模式，都将进入到中断服务或异常中去，或者访问受保护的资源。

* S3C2440通用寄存器介绍（详见s3c2440中文手册）

ARM920T 总共有 37 个寄存器，其中 31 通用 32 位寄存器和 6 个状态寄存器，但不能在同一时刻对所有的寄 存器可见。处理器状态和运行模式决定了哪些寄存器对程序员可见。 ARM 状态时内部寄存器集 在 ARM 状态，16 个通用寄存器和一个状态寄存器在任意时刻都可见。在特权（非用户）模式下，将切换到指 定模式的分组（banked）寄存器。图 1.3 显示了哪些寄存器在各模式下是可见的：分组寄存器被标记了阴影三角形。 ARM 状态时寄存器被设为包含 16 个直接可以访问的寄存器：R0 到 R15。除了 R15，其他全部寄存器都为通 用寄存器，如可能用于保存数据或地址值。除此之外，还有第 17 个寄存器用于存储状态信息。

|  |  |
| --- | --- |
| 寄存器14 | 此寄存器被用作子程序的链接寄存器。当执行分支和链接（BL）指令时该寄存器接收 R15 的备份。  其余时间也可作为通用寄存器处理。相对应的分组寄存器 R14\_svc，R14\_irq，R14\_fiq，R14\_abt  和 R14\_und 也类似用于中断或异常发生时 R15 的返回值，或者在中断或异常例程中执行了分支和  链接中断。 |
| 寄存器15 | 此寄存器存放着程序计数器（PC）。在 ARM 状态，R15 的位[1:0]都为 0，位[31:2]为 PC 值。在 Thumb  状态，只有位[0]为 0，位[31:1]为 PC 值。 |
| 寄存器16 | 此寄存器为 CPSR（当前程序状态寄存器）。它包含条件码标志位和当前模式位。 |

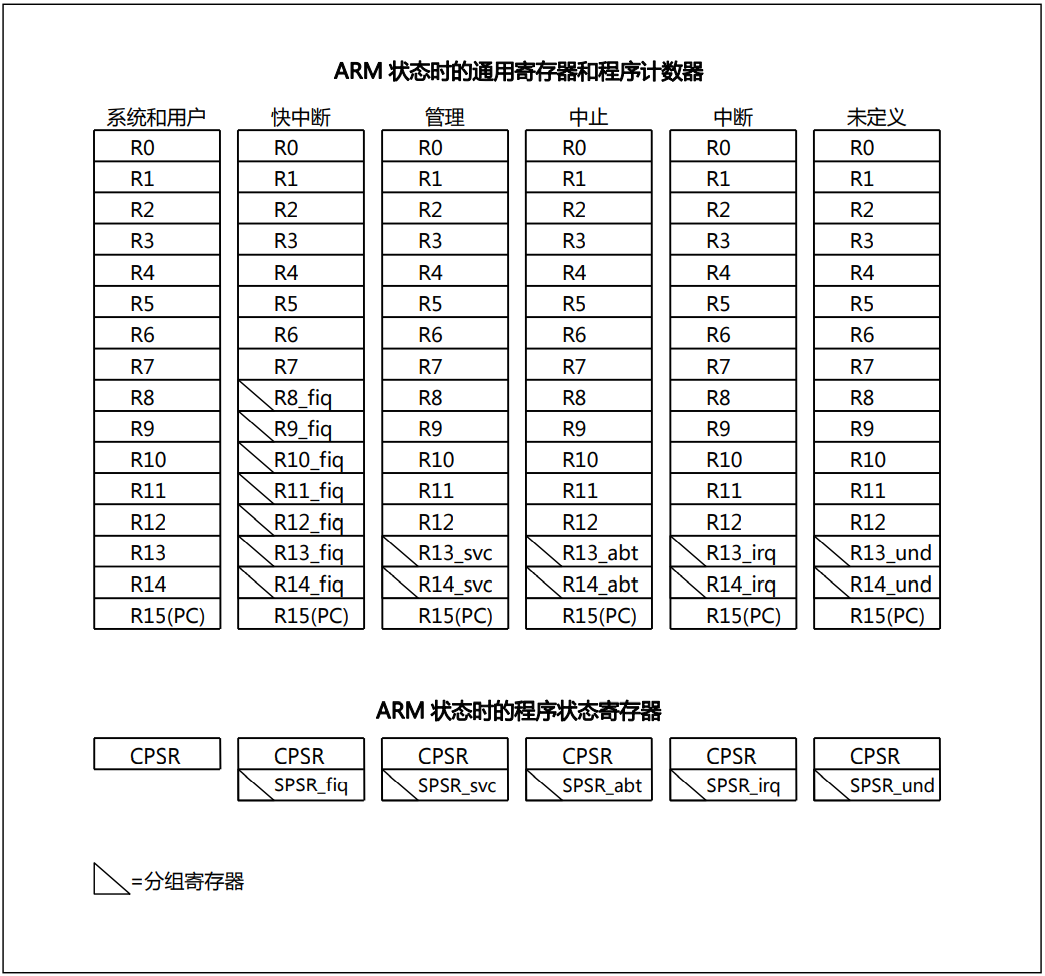


图1.13 ARM 状态时寄存器组织

* S3C2440特殊寄存器介绍（详见s3c2440中文手册）

除了CPU使用的通用寄存器，S3C2440还有很多特殊寄存器，用以控制、配置时钟、IO、存储设备等外设。每个寄存器都有自己的物理地址，如BWSCON寄存器的地址为0x48000000，用来设置总线宽度和等待状态控制。操作系统就通过这些十六进制的地址访问、读取、修改这些寄存器，从而达到控制设备的目的。

之前在做按键亮灯的裸板程序时，用到的那些寄存器都属于这类特殊寄存器。

* S3C 2440存储空间

S3C2440A总共有8个存储器Bank，可将一个Bank理解为一块存储空间。6个存储器Bank为ROM（只读），SRAM等；其余2个存储器Bank为ROM，SRAM，SDRAM 等。前7个存储器Bank的起始地址固定，第8个存储器Bank起始地址可变并Bank大小可编程。下图为2440存储器地址分配。

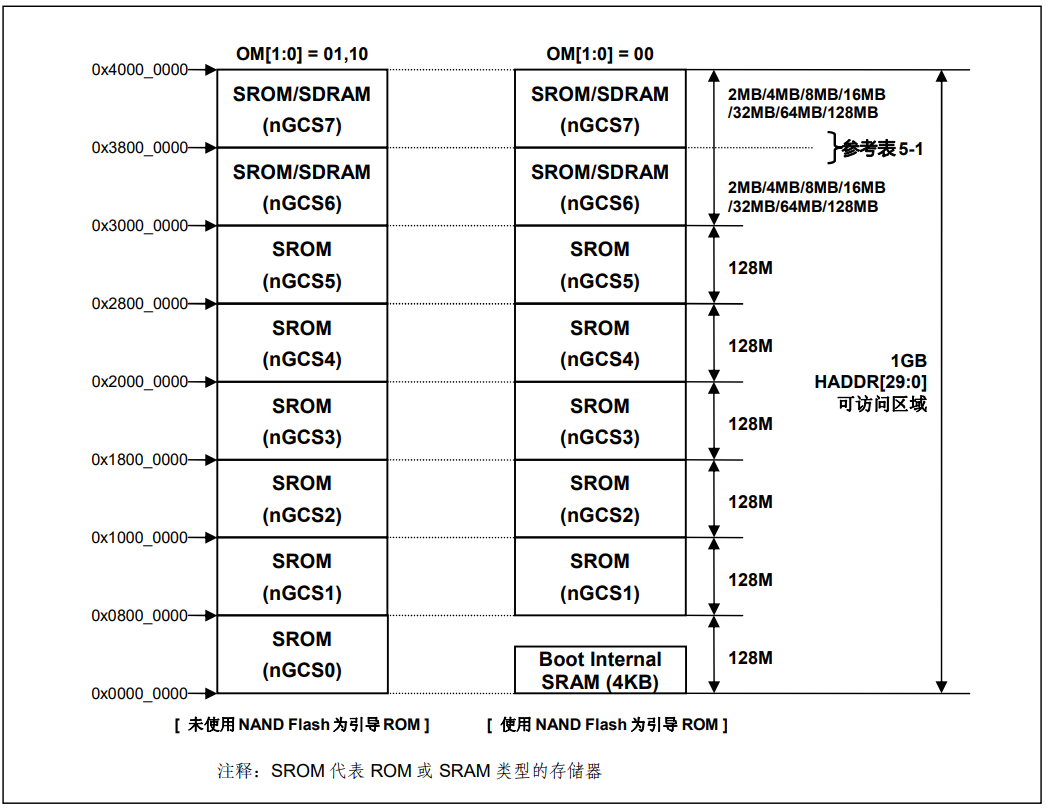


图1.14 S3C2440存储器映射

上图中，左边表示从NOR FLASH启动系统的地址分布，右边表示从NAND FLASH启动系统的地址分布。两种FLASH的区别自行查阅。由图中可知一个重要信息——S3C2440的SDRAM，也就是我们常说的内存，挂载在Bank6，起始地址为0X30000000，请同学们务必记住这个地址。同时，结合芯片手册，我们还可以发现，特殊寄存器的地址是从0X48000000开始的，与上图中的地址互不干涉，因此CPU统一使用一套地址规则访问特殊寄存器或者上图中的存储空间。

### 开发环境简介（软硬件如何结合，涉及到宿主机与客户机、交叉编译链、串口回显、jtag调试与烧写）

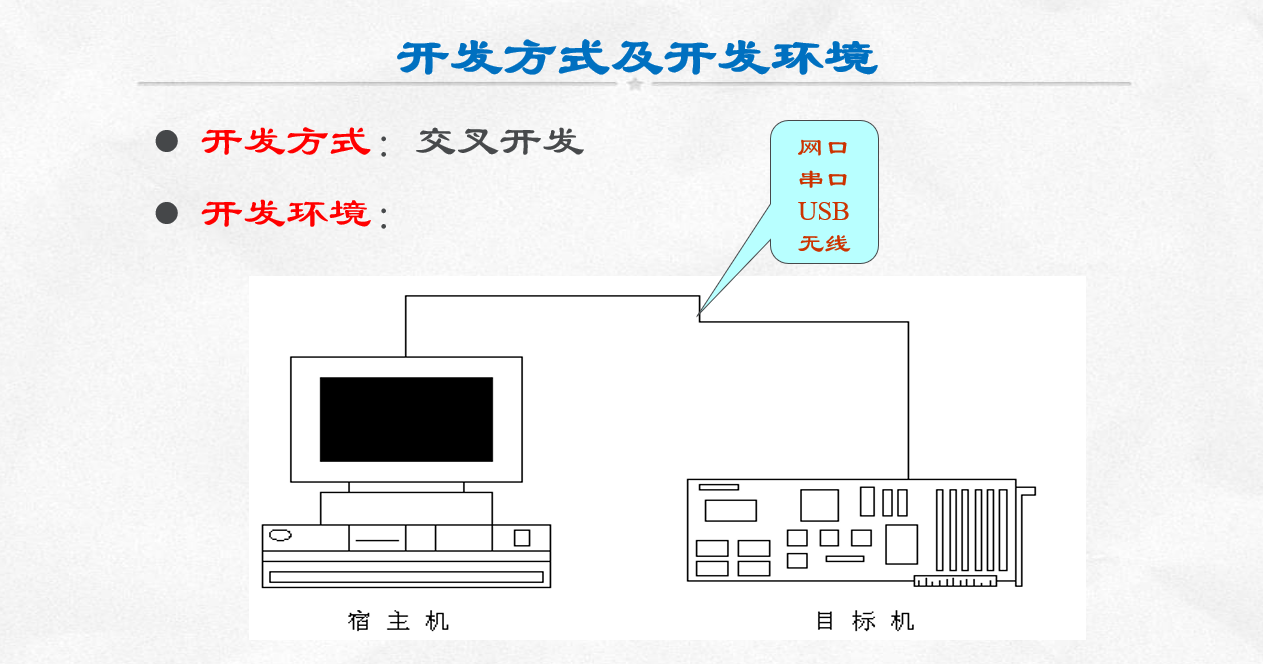


图1.15

## aCoral

本门课程将要学习的嵌入式实时操作系统就是aCoral，对应图1.1中的嵌入式操作系统部分。

### aCoral简介

aCoral是电子科技大学计算机学院嵌入式实时计算实验室开发的一款嵌入式多核实时操作系统,开源,支持多核（SMP：Symmetry Multiple Process）的嵌入式实时操作系统，具有高可配，高扩展性。

aCoral对于主流的开发平台都有支持，像s3c2440,s3c2410,s3c44b0，lpc2313,lpc2200,stm3210,同时支持ARM11多核平台pb11mpcore，还有两个PC虚拟平台emu-single,emu-cmp,只要你有一台运行Linux的PC机，你就可以学习，运行，调试aCoral。同时文档中讲解的移植模板更是给想做移植的人一个动手实践的方向，任何人都可以参照代码，写出自己的内核。尤其是多核，大家对多核可能都有所了解，但是嵌入式多核估计就了解的少了，对应的多核操作系统就更少了，aCoral源码及附带文档能让你深入了解多核，最后你会发现，多核其实就是那么一回事。

### aCoral启动入口

本节主要介绍aCoral在mini2440中启动的过程，即2440上电执行的是哪行代码，之后又做了哪些事。

* start.S启动代码（硬件相关初始化）

aCoral上电之后执行的第一行代码就位于**hal\s3c2440\src\start.s**中，标号**\_\_ENTRY**处。至于为什么是这个地方是入口，可以查看**hal\s3c2440\acoral.lds**,称为链接脚本，如下

|  |
| --- |
| ENTRY(\_\_ENTRY) |

这行代码指示了**\_\_ENTRY**标号将被放在整个系统镜像的最前面，即0x0地址位置。在**start.S**中，**\_ENTRY**标号处的代码如下

|  |
| --- |
| \_\_ENTRY:  b ResetHandler |

2440上电之后，CPU将从0地址处开始取指执行指令。而0地址处正是这句跳转指令。具体异常处理机制将在第二章介绍。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，代码如下

|  |
| --- |
|  |

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**kernel\include\core.c**中

* 跳到C语言部分，core.c

**core.c**中的**acoral\_start()**函数如下

|  |
| --- |
| void acoral\_start(){  orig\_thread.console\_id=ACORAL\_DEV\_ERR\_ID;  acoral\_set\_orig\_thread(&orig\_thread);  /\*内核模块初始化\*/  **acoral\_module\_init();**  /\*串口终端应该初始化好了，将根线程的终端id设置为串口终端\*/  #ifdef CFG\_DRIVER  orig\_thread.console\_id=acoral\_dev\_open("console");;  #endif  /\*主cpu开始函数\*/  acoral\_core\_cpu\_start();  } |

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。初始化完成后，执行**acoral\_core\_cpu\_start()**函数，aCoral开始运行。

* 课程主线——core.c中的acoral\_module\_init()函数

|  |
| --- |
| void acoral\_module\_init(){  /\*中断模块初始化\*/  acoral\_intr\_sys\_init();  /\*内存管理系模块初始化\*/  acoral\_mem\_sys\_init();  /\*资源管理模块初始化\*/  acoral\_res\_sys\_init();    /\*线程管理系统初始化\*/  acoral\_thread\_sys\_init();  /\*时钟管理模块初始化\*/  acoral\_time\_sys\_init();  /\*事件管理模块初始化,这个必须要，因为内存管理系统用到了\*/  acoral\_evt\_sys\_init();  /\*驱动管理模块初始化\*/  #ifdef CFG\_DRIVER  acoral\_drv\_sys\_init();  #endif  } |

接下来的整个课程所有内容都是以这个函数为主线，依次介绍aCoral的每个模块。

1. 中断模块——第二章
2. 内存管理模块——第三章前两节
3. 资源管理模块——第三章第三节
4. 线程管理模块——第四章、第五章
5. 时钟管理模块暂时不介绍
6. 事件管理模块——第六章

# aCoral中断系统

中断系统是acoral\_module\_init()函数中初始化的第一个模块，本章就将针对中断系统的特点以及aCoral的中断系统是如何工作的进行介绍与学习。

## 中断简介与前后台系统的实现

* 中断介绍

中断是指由于CPU外部的原因而改变程序执行流程的过程。属于异步事件，又称为硬件中断。比如mini2440上的按键就可以触发一个外部中断，或者2440内部的timer所计的时间到了也可以触发一个中断等等。中断根据不同种类可分为如下几种：

**按可否屏蔽**：可屏蔽中断和不可屏蔽中断。

**按中断信号的产生方式**：边缘触发中断和电平触发中断。

**按中断服务程序的调用方式**：向量中断、直接中断和间接中断

**按中断源的位置**：外部中断、内部中断

* 中断与异常的异同

在S3C2440中，除了中断，还有一个概念叫异常。在2440中，中断作为异常的一种，除了中断之外，还有复位、未定义指令、软中断、快中断（FIQ）等异常类型。这里要注意区分中断（IRQ）和快中断（FIQ）是两回事。异常作为一种硬件机制，意思是在发生某种异常后，CPU通过硬件机制强制到某个固定的地址取指处理异常。每一种异常都对应一个固定的地址，我们称这些地址为异常向量表。2440的异常向量表如下：



图2.1 2440异常向量表

由于2440上电复位后将触发一个复位异常，根据异常向量表可知，CPU将从0地址开始取指执行，这就解释了第一章第三节中为什么2440上电后是从**\_ENTRY**标号处开始执行。

* 中断源介绍（结合s3c2440中文手册）

S3C2440中可以触发中断的东西称为中断源。S3C2440有60个中断源，如下图：

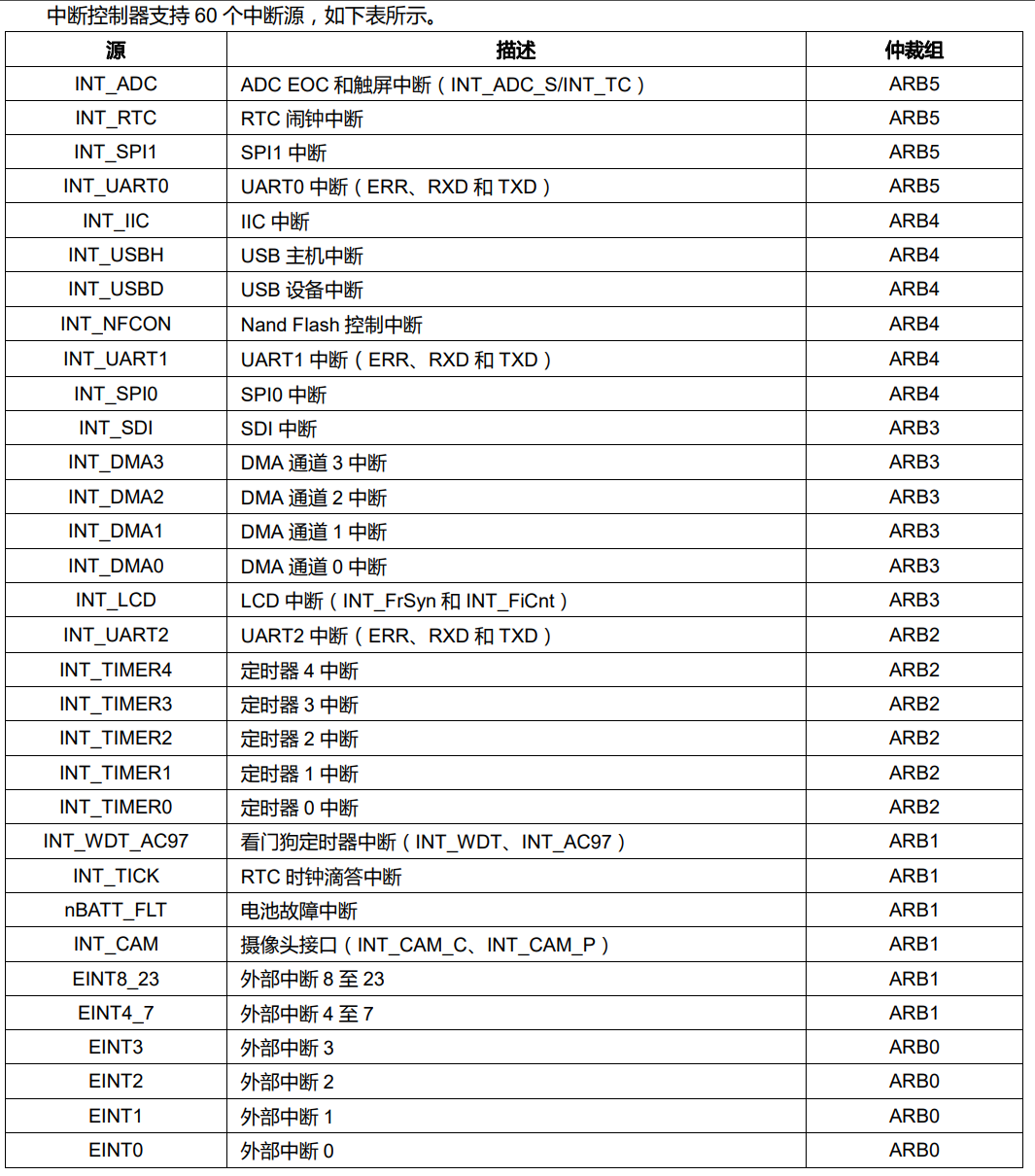


图2.2 2440中断源

其中，EINT0~23属于外部中断（External Interruption），用于mini2440开发板上的按键产生的中断等；其余的中断属于内部中断。形如EINT8~23这样的中断源称为中断复用，即多个中断产生复用一个中断源。要区分复用中断到底是由哪个中断源产生的，则需要进一步结合其他寄存器进行判断，具体将在之后的小节学习。

* 中断控制器特殊寄存器

想让CPU知道发生了中断，这需要许多寄存器的合作。

此处中断控制器中有5个最重要的控制寄存器：源挂起寄存器（SRCPND）、屏蔽寄存器(INTMSK)、中断模式寄存器（INTMOD）、优先级寄存器（PRIORITY）和中断挂起寄存器（INTPND）。各个寄存器的逻辑关系如下图。

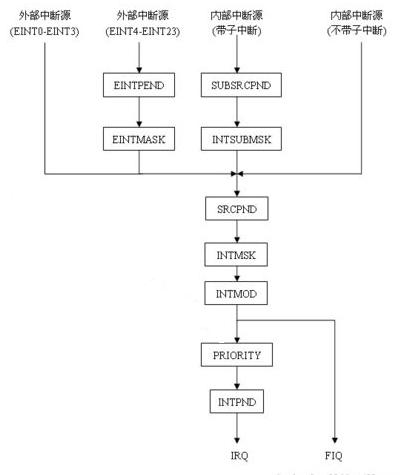
。

图2.3 中断相关寄存器的关系

所有来自中断源的中断请求首先被记录到源挂起寄存器（SRCPND）中，经过屏蔽寄存器（INTMSK）屏蔽某一些中断，中断模式寄存器（INTMOD）判断当前中断源被设置为中断（IRQ）还是快中断（FIQ），如果是中断，则需在经过优先级寄存器（PRIORITY）仲裁优先级，某一时间优先级最高的普通中断在中断挂起寄存器（INTPND）中对应的位将被置1，并产生中断请求给CPU。如果程序状态寄存器（CPSR）中的IRQ中断禁止位没有被置1，则进入中断处理流程，CPU将通过硬件机制跳转到异常向量表的第七项，即地址0x00000018，开始处理IRQ中断。

* 中断处理流程

1. **寻找中断向量**
2. **保护现场**
3. **执行中断服务程序**
4. **恢复现场**

* 前后台系统

撇开比较复杂的aCoral，先让大家写一个嵌入式软件系统中比较简单的前后台系统。让大家理清楚中断整个的原理、流程。

提供第一章第三节提到的启动文件start.S（**还要改，中断向量表要放到异常向量表后面**），进行系统硬件初始化，完成之后跳转到前后台系统开始运行。

|  |
| --- |
|  |

代码2.1

## aCoral中断介绍

* aCoral中断处理与前后台系统的异同

**相同之处：**同样都是先跳转到异常向量表中的中断项（0x18），再跳转到公共入口中，读取INTOFFSET判断中断源，

**不同之处**：之前的前后台系统没有进一步区分次级中断源，或者说没有区分复用的4~7，8~23外部中断源，并且中断向量表直接使用汇编进行编写，放在了异常向量表的后面。而aCoral在**hal\_all\_entry**函数中，进一步区分了次级中断源，并且中断向量表**intr\_table**使用了c语言编写，作为一个数组存放在系统镜像的代码段。本质上而言两者的中断向量表没有区别，只是表现形式略有不同罢了。

关于2440的中断复用，或者说次级中断源，请看下图。



图2.4

* aCoral三种中断类型

1. **实时中断**

直接调用中断处理程序，不经过：HAL→hal\_all→intr→中断处理程序

1. **专家中断（ACORAL\_EXPERT\_INTR）**

需要用户在中断处理程序中自己处理中断响应相关的操作，如清中断位等：特殊中断

1. **普通中断（ACORAL\_COMM\_INTR）**

不需要任何中断硬件相关的操作，只需编写中断要做什么

## aCoral中断注册

* aCoral中断向量表

aCoral中断向量表**intr\_table**定义如下

|  |
| --- |
| acoral\_intr\_ctr\_t intr\_table[HAL\_INTR\_NUM]; |

代码2.2

数据结构**acoral\_intr\_ctr\_t**定义如下

|  |
| --- |
| typedef struct {  acoral\_u32 index; //中断号  acoral\_u8 state; //中断状态  acoral\_u8 type; //中断类型，包括普通模式、专家模式  void (\*isr)(acoral\_vector); //中断服务程序  void (\*enter)(acoral\_vector); //中断进入处理程序，如清中断  void (\*exit)(acoral\_vector); //中断退出处理程序  void (\*mask)(acoral\_vector);  void (\*unmask)(acoral\_vector);  }acoral\_intr\_ctr\_t; |

代码2.3

* aCoral中断初始化

在aCoral能够正常响应中断之前，即在系统初始化阶段，必然要先对中断系统进行初始化——中断注册。中断注册，顾名思义就是对每个中断源产生的中断编写中断服务程序，也就是为中断向量表intr\_table数组的每个成员赋值。这也是之前提到的主线中第一个模块**acoral\_intr\_sys\_init()（回到主线那里再提醒一下学生）**所做的事情。

下面请尝试补全中断注册代码：

* **目的：在中断向量**表中为每个中断源注册中断服务程序，并设置中断类型
* 提供：aCoral中断向量表intr\_table
* 提供：aCoral默认中断服务程序acoral\_default\_isr
* 提供aCoral三种中断类型：ACORAL\_COMM\_INTR、ACORAL\_EXPERT\_INTR、ACORAL\_RT\_INTR

|  |
| --- |
| void acoral\_intr\_sys\_init(){  acoral\_u32 i;  acoral\_vector index;  /\*关中断\*/  acoral\_intr\_disable();  /\*中断嵌套标志初始化\*/  HAL\_INTR\_NESTING\_INIT();  /\*中断底层初始化函数\*/  HAL\_INTR\_INIT();  /\*对于每个中断，设置默认的服务处理程序，然后屏蔽该中断\*/  for(i=HAL\_INTR\_MIN;i<=HAL\_INTR\_MAX;i++){  HAL\_TRANSLATE\_VECTOR(i,index);  intr\_table[index].isr=acoral\_default\_isr;  intr\_table[index].type=ACORAL\_COMM\_INTR;  acoral\_intr\_mask(i);  }  } |

代码2.4

* aCoral关中断

代码2.4中，使用了函数acoral\_intr\_disable()来关闭中断。在中断初始化完成之前，aCoral还不具备响应中断的能力，所以初始化的第一步就是要关闭中断。aCoral通过将CPSR寄存器的IRQ中断位置1，来关闭中断。这样即使发生了中断请求，INTPND寄存器中的某一位被置为1，CPU也还是不会跳入中断。CPSR寄存器格式如下图

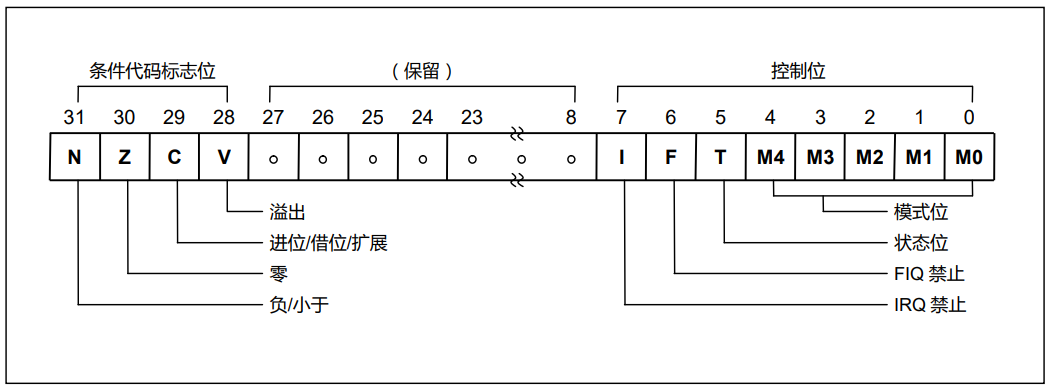


图2.5 程序状态寄存器格式

参照图2.5，请尝试补全关中断acoral\_intr\_disable()的代码。

* 目的：将CPSR寄存器的IRQ位置1，以屏蔽中断。
* 提供汇编指令：mrs、msr

|  |
| --- |
| HAL\_INTR\_DISABLE:  mrs r0,cpsr  mov r1,r0  orr r1,r1,#0xc0  msr cpsr\_cxsf,r1  mov pc ,lr |

代码2.5

* 中断底层初始化

代码2.4中，使用了函数HAL\_INTR\_INIT()来完成底层硬件相关的中断初始化，也就是对之前提到的几个中断控制寄存器进行赋值。具体包括：

1. 配置PRIORITY寄存器，使用默认优先级
2. 配置INTMOD寄存器，每个中断都为IRQ中断
3. 配置INTMSK寄存器，屏蔽所有中断。这里与代码2.5关闭中断有所不同，这里屏蔽所有中断，INTPND寄存器就不会有任何一位被置为1；而代码2.5所做的是，即使INTPND有一位为1，也不会跳入中断。这两种做法任何一种其实都能完全避免aCoral陷入中断，aCoral只是选择了两种都做。
4. 配置EINTMSK寄存器，屏蔽所有外部中断。同3。

除了对这些寄存器赋值以外，还要完善中断向量表每个表项的成员。我们再来看一下中断向量的结构

|  |
| --- |
| typedef struct {  acoral\_u32 index; //中断号  acoral\_u8 state; //中断状态  acoral\_u8 type; //中断类型，包括普通模式、专家模式  void (\*isr)(acoral\_vector); //中断服务程序  void (\*enter)(acoral\_vector); //中断进入处理程序，如清中断  void (\*exit)(acoral\_vector); //中断退出处理程序  void (\*mask)(acoral\_vector);  void (\*unmask)(acoral\_vector);  }acoral\_intr\_ctr\_t; |

代码2.6

代码2.4仅仅为isr和type成员赋了值，那其他几个成员的赋值也是在hal\_intr\_init中完成的。其中enter负责在中断开始之前清中断，exit负责中断之后结束之后处理，mask负责屏蔽某一个中断（修改INTMSK寄存器的值），unmask是maks的逆操作。

* 目的：设置四个中断控制寄存器，使用默认优先级，所有中断为IRQ中断，屏蔽所有中断
* 提供：中文手册

|  |
| --- |
| void hal\_intr\_init(){  acoral\_u32 i;  rPRIORITY = 0x00000000; /\* 使用默认的固定的优先级\*/  rINTMOD = 0x00000000; /\* 所有中断均为IRQ中断\*/  rEINTMSK= 0xffffffff; /\*屏蔽所有外部中断\*/  rINTMSK = 0xffffffff; /\*屏蔽所有中断\*/  for(i=HAL\_INTR\_MIN;i<=HAL\_INTR\_MAX;i++){  acoral\_set\_intr\_enter(i,hal\_intr\_ack);  acoral\_set\_intr\_exit(i,NULL);  acoral\_set\_intr\_mask(i,hal\_intr\_mask);  acoral\_set\_intr\_unmask(i,hal\_intr\_unmask);  }  } |

代码2.7

其实不管是代码2.5还是代码2.7，目的都只有一个，就是在中断初始化完成，系统有能力处理中断之前，屏蔽一切中断。，

* 注册按键K1中断服务程序

在学习了aCoral如何注册一个中断服务函数后，请为mini2440开发板上的按键K1注册中断服务函数。在注册中断之前，我们先需要知道按键K1对应哪个中断源。这时就需要查阅mini2440原理图了，如图

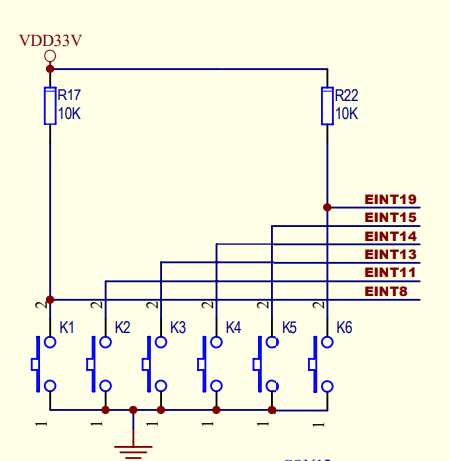


图2.6

由图中可知，按键K1对应中断源EINT8，再通过查阅S3C2440中文手册可知，EINT8~23复用中断号5，也就是中断偏移量为5，如图2.7。但是aCoral会区分中断复用，也就是说intr\_table中的第0项对应EINT0，第4项对应EINT4，第7项对应EINT7，以此类推，intr\_table的每一项都对应一个（次级）中断源。

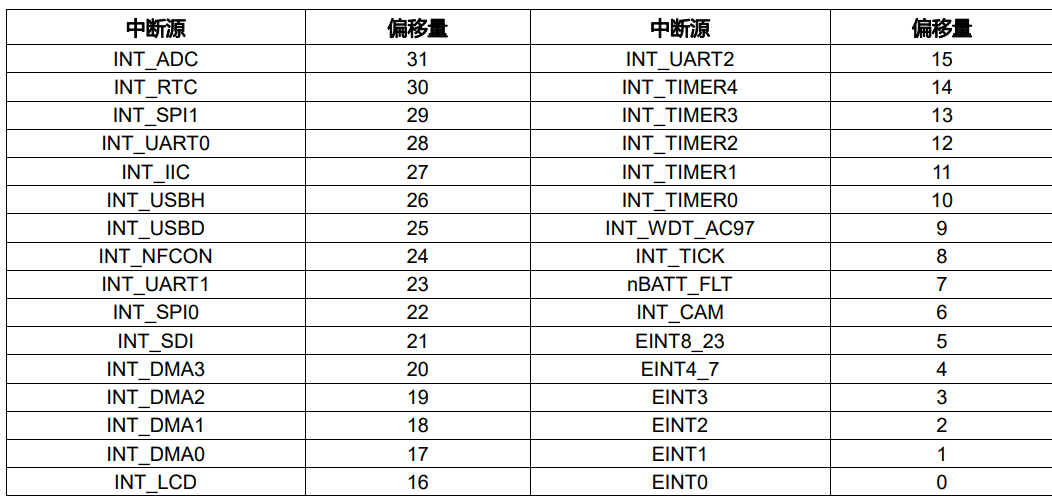


图 2.7

* 目的：为按键K1注册中断服务程序k1\_isr
* 提供mini2440原理图
* 提供S3C2440中文手册
* 提供K1的中断服务程序k1\_isr

|  |
| --- |
| for(i=HAL\_INTR\_MIN;i<=HAL\_INTR\_MAX;i++){  HAL\_TRANSLATE\_VECTOR(i,index);  intr\_table[index].isr=acoral\_default\_isr;  intr\_table[index].type=ACORAL\_COMM\_INTR;  acoral\_intr\_mask(i);  }  /\*为按键K1注册中断服务程序\*/  intr\_table[8].isr=k1\_isr; |

代码2.8

但是，仅仅为EINT8注册了中断服务程序还不够，还需要将按键k1对应的引脚配置为中断引脚。通过查阅s3c2440中文手册可知，只需设置GPGCON寄存器的[1：0]=10即可，如图

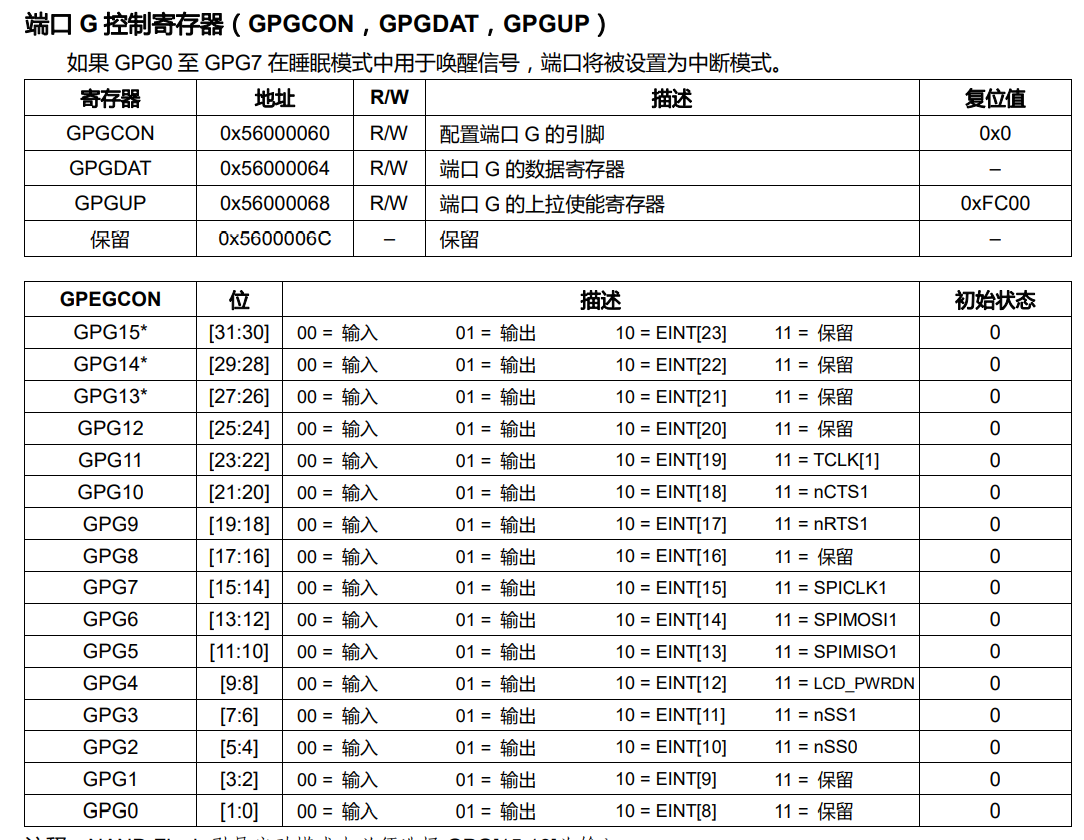


图2.8

## aCoral中断发生及响应

上一节讲解了aCoral如何注册中断服务程序。那在发生中断后，如何响应中断并调用已经注册的中断服务程序呢？这个流程和之前写的前后台系统响应中断的流程又有什么不同呢？这就是这一节要学习的内容。

* HAL层——中断汇总入口HAL\_INTR\_ENTRY代码

2440发生中断后，通过硬件机制到0x18取指。aCoral中0x18处的代码位于hal\s3c2440\src\start.S中，如下：

|  |
| --- |
| b HandleIRQ |

代码2.9

这是一行跳转指令，CPU跳转到HandleIRQ标号处执行。此处的代码如下

|  |
| --- |
| HandleIRQ:  ldr pc,=HAL\_INTR\_ENTRY |

代码2.10

HandleIRQ处依旧是一条跳转指令，CPU 跳到HAL\_INTR\_ENTRY标号处执行。HAL\_INTR\_ENTRY称为aCoral的中断硬件层公共入口，所有的中断都会跳转到这里执行，进行现场的保存、调用中断处理函数，最后再恢复中断以前的现场。所谓的现场，又或者叫上下文环境（context），指的就是当前CPU那些通用寄存器（R0~R15、CPSR）的值；而保存环境指的就是把这些寄存器的值保存到当前SP寄存器指向的堆栈中，以便之后能够恢复并继续执行被中断的任务。

下面请尝试根据以上内容，补全HAL\_INTR\_ENTRY处的代码。

* 目的：保存通用寄存器的值，进入内核层进行下一步处理，最后恢复通用寄存器的值

|  |
| --- |
| HAL\_INTR\_ENTRY:  /\*保存上下文\*/  stmfd sp!, {r0-r12,lr} @保护通用寄存器及PC  mrs r1, spsr  stmfd sp!, {r1} @保护spsr，以支持中断嵌套  msr cpsr\_c, #SVC\_MODE|NOIRQ @进入SVC\_MODE,以便允许中断嵌套，同时关闭中断  stmfd sp!, {lr} @保存SVc模式的专用寄存器lr    /\*进入内核层处理\*/  ldr r0, =INTOFFSET @读取中断向量号  ldr r0, [r0]  mov lr, pc @求得lr的值  ldr pc, =hal\_all\_entry @中断内核层处理  ldmfd sp!, {lr} @恢复svc模式下的lr,  msr cpsr\_c,#IRQ\_MODE|NOINT @更新cpsr,进入IRQ模式并禁止中断    /\*恢复上下文\*/  ldmfd sp!,{r0} @spsr->r0  msr spsr\_cxsf,r0 @恢复spsr  ldmfd sp!,{r0-r12,lr}  subs pc,lr,#4 @此后，中断被重新打开 |

代码2.11

* 内核层1——中断向量号复用处理入口函数hal\_all\_entry

代码2.11中的hal\_all\_entry就是对中断向量号进行进一步处理，分离出复用的向量号，以便之后调用对应的中断服务程序。

对于中断号复用的处理，对于S3C2440具体来说，就是将中断EINT4~7、8~23这些若干个中断复用一个中断号的情况，区分出具体是4~7或者8~23中的哪一个发生了中断。在分离出了复用的中断号后，就相当于原本的4~7、8~23两个中断源变成了20个中断源，因此在EINT23之后的中断源的中断号就需要加上20-2=18，例如INT\_CAM中断源的中断号就从6号变成了24号。我们称这些处理过的中断号为内核层中断号。中断公共服务入口函数——acoral\_intr\_entry将使用内核层中断号，调用相应的中断服务程序。

* 目的：根据传入的INTOFFSET寄存器的中断偏移量vector，分离出复用的4~7、8~23号中断，并对EINT23之后的中断号进行处理，最后使用这些内核层中断号作为acoral\_intr\_entry（）的参数。
* 提供寄存器——rEINTPND，详见s3c2440中文手册
* 提供中断公共服务入口函数——acoral\_intr\_entry()

|  |
| --- |
| void hal\_all\_entry(acoral\_vector vector){  unsigned long eint;  unsigned long irq=4;  if(vector==4||vector==5){  eint=rEINTPND;  for(;irq<24;irq++){  if(eint & (1<<irq)){  acoral\_intr\_entry(irq);  return;  }  }  }  if(vector>5)  vector+=18;  if(vector==4)  acoral\_prints("DErr\n");  acoral\_intr\_entry(vector);  } |

代码2.12

* 内核层2——中断响应实际入口

代码2.12中使用的中断公共服务入口函数——acoral\_intr\_entry，本质上就是根据传入的内核层中断号index，调用intr\_table中断向量表中对应的中断服务程序。同时注意到，之前在保存上下文环境时，关闭了中断，所以为了支持中断嵌套，在调用中断服务程序之前还要打开中断。

还有一点要注意的是，中断发生后，INTPND寄存器中的某一位被置为1，CPU就会收到中断请求。但INTPND中的这一位并不会自动归0，也就是说CPU会不停接收到来自这一位发出的中断请求。所以我们需要手动向INTPND的这一位写1来清0。intr\_table中的每个成员不仅包含了中断服务程序，也包含了中断进入时程序enter，即用来将INTPND寄存器对应中断位清0的函数（清中断）。intr\_table定义见代码2.3。

请根据以上描述尝试补全代码2.13，acoral\_intr\_entry函数

* 目的：清中断以防止CPU不断接收到中断请求、开中断以支持中断嵌套、调用中断服务函数、关中断，最后调用中断退出函数。
* 提供：开中断acoral\_intr\_enable();
* 提供：关中断acoral\_intr\_disable();
* 提供：中断退出函数acoral\_intr\_exit()
* 参考数据结构：中断向量表

|  |
| --- |
| acoral\_intr\_ctr\_t intr\_table[HAL\_INTR\_NUM];  typedef struct {  acoral\_u32 index; //中断号  acoral\_u8 state; //中断状态  acoral\_u8 type; //中断类型，包括普通模式、专家模式  void (\*isr)(acoral\_vector); //中断服务程序  void (\*enter)(acoral\_vector); //中断进入处理程序，如清中断  void (\*exit)(acoral\_vector); //中断退出处理程序  void (\*mask)(acoral\_vector);  void (\*unmask)(acoral\_vector);  }acoral\_intr\_ctr\_t; |

|  |
| --- |
| void acoral\_intr\_entry(acoral\_vector vector){  #ifdef CFG\_DEBUG  acoral\_print("isr in cpu:%d\n",acoral\_current\_cpu);  #endif  acoral\_intr\_nesting\_inc(); //增减中断嵌套数  if(intr\_table[index].type==ACORAL\_EXPERT\_INTR){  intr\_table[vector].isr(vector);  acoral\_intr\_disable();  }else{  /\*清中断\*/  if(intr\_table[vector].enter!=NULL)  intr\_table[vector].enter(vector);    acoral\_intr\_mask(vector);  /\*开中断\*/  acoral\_intr\_enable();  /\*调用该中断的服务处理函数\*/  intr\_table[vector].isr(vector);    /\*关中断\*/  acoral\_intr\_disable();    acoral\_intr\_unmask(vector);  /\*调用中断退出函数\*/  if(intr\_table[vector].exit!=NULL)  intr\_table[vector].exit(vector);  }  acoral\_intr\_nesting\_dec(); //减少中断嵌套数  acoral\_intr\_exit();  } |

代码2.13

* 中断退出

代码2.13最后的acoral\_intr\_exit()函数，就是所有中断处理完成后的退出函数。其实就是判断是否需要调度，如果要则重调度；如果不要则继续之前被中断的任务。

|  |
| --- |
| void acoral\_intr\_exit(){  if(!acoral\_need\_sched)  return;  if(acoral\_intr\_nesting)  return;  if(acoral\_sched\_is\_lock)  return;  if (!acoral\_start\_sched)  return;    /\*如果需要调度，则调用此函数\*/  HAL\_INTR\_EXIT\_BRIDGE();  } |

代码2.14

* aCoral中断响应、处理流程小结



图2.9

## aCoral时钟中断

在RTOS中，时钟具有非常重要的作用，通过时钟可以实现延时任务、周期性触发任务执行、任务有限等待的计时、软定时器的定时管理、确认超时以及与时间相关的调度操作，如时间片轮转调度等。时钟管理是处理实时系统必不可少的内容，因此，实时内核必须提供时间管理机制。

大多数嵌入式系统有两种时钟源，分别为实时时钟RTC和定时器TIMER。实时时钟一般是靠电池供电，即使系统断电，也可以维持日期和时间，如ARM9（S3C2440）提供的RTC.由于实时时钟独立于操作系统，因此也被称为硬件时钟。另外一种时钟源就是定时器。嵌入式系统一般集成了多个定时器，实时内核需要一个定时器作为系统时钟，并由内核控制系统时钟工作，系统时钟的最小粒度是由应用和操作系统的特点决定的。

aCoral使用2440的TIMER0作为系统时钟，并由TIMER0产生时钟中断。TIMER0基于PCLK。所谓时钟中断，就是时钟（这里是timer0）计时到某一数值时，产生一个中断，然后处理系统中相关的任务。例如对于分时策略，每当一个ticks中断到来时，其中断服务程序就会将当前分时策略线程的时间片slice成员值减1，当slice为0时，就会触发调度。由S3C2440中文手册可知，2440中的timer0对应中断源INT\_TIMER0，中断偏移量10。由于有18个中断复用，所以在aCoral中断向量表中的下标为28。时钟中断的注册是在aCoral所有模块初始化完成之后，即acoral\_module\_init()函数执行完毕后，创建的init线程中完成的，注册代码如代码5.5所示。

* 目的：注册时钟中断服务程序，初始化计时器timer0.
* 提供：中断服务程序注册函数acoral\_32 acoral\_intr\_attach(acoral\_vector vector,void (\*isr)(acoral\_vector))
* 提供：时钟中断号HAL\_TICKS\_INTR

|  |
| --- |
| void acoral\_ticks\_init(){  ticks=0; /\*初始化滴答时钟计数器\*/  acoral\_intr\_attach(HAL\_TICKS\_INTR,acoral\_ticks\_entry);/\*这个用于注册ticks的处理函数\*/  HAL\_TICKS\_INIT(); /\*这个主要用于将用于ticks的时钟初始化\*/  acoral\_intr\_unmask(HAL\_TICKS\_INTR);  return;  } |

代码5.5

代码5.5中的HAL\_TICKS\_INIT()函数，就是为了初始化计时器timer0。这部分代码清同学们结合s3c2440中文手册第10章自行理解。

|  |
| --- |
| void hal\_ticks\_init(){  rTCON = rTCON & (~0xf) ; /\* clear manual update bit, stop Timer0\*/  rTCFG0 &= 0xFFFF00;  rTCFG0 |= 0xF9; /\* prescaler等于249\*/  rTCFG1 &= ~0x0000F;  rTCFG1 |= 0x2; /\*divider等于8，则设置定时器4的时钟频率为25kHz\*/  rTCNTB0 = PCLK /(8\*(249+1)\*ACORAL\_TICKS\_PER\_SEC);  rTCON = rTCON & (~0xf) |0x02; /\* updata\*/  rTCON = rTCON & (~0xf) |0x09; /\* star\*/  } |

代码5.6

时钟中断ticks中断服务程序如代码5.7所示.其中acoral\_policy\_delay\_deal()就是处理和时间相关的策略线程的。

|  |
| --- |
| void acoral\_ticks\_entry(acoral\_vector vector){  ticks++;  acoral\_printdbg("In ticks isr\n");  if(acoral\_start\_sched==true){  time\_delay\_deal();  acoral\_policy\_delay\_deal();  /\*--------------------\*/  /\* 超时链表处理函数\*/  /\* pegasus 0719\*/  /\*--------------------\*/  timeout\_delay\_deal();  }  } |

代码5.7

# aCoral内存管理系统

aCoral的内存管理系统分为两级，分别作为acoral\_module\_init()函数中初始化的第二个和第三个模块。本章就将针对aCoral内存管理系统进行介绍与学习。

## 内存管理机制介绍

* 静态内存分配
* 动态内存分配之固定大小存储管理
* 动态内存分配之可变大小存储管理
* aCoral内存管理机制

## aCoral第一级内存管理机制——伙伴系统

伙伴系统作为acoral\_module\_init()函数中初始化的第二个模块，直接与物理内存相关。

* 伙伴系统初始化
* 目的：初始化伙伴系统

|  |
| --- |
| acoral\_err buddy\_init(acoral\_u32 start\_adr,acoral\_u32 end\_adr){  acoral\_32 i,k;  acoral\_5u32 resize\_size;  acoral\_u32 save\_adr; //为  acoral\_u32 index;  acoral\_u32 num=1;  acoral\_u32 adjust\_level=1;  acoral\_32 level=0;  acoral\_u32 max\_num,o\_num;  acoral\_u32 cur;  start\_adr+=3;  start\_adr&=~(4-1);  end\_adr&=~(4-1);  resize\_size=BLOCK\_SIZE;  end\_adr=end\_adr - sizeof(acoral\_block\_ctr\_t);  end\_adr&=~(4-1);  acoral\_mem\_ctrl=(acoral\_block\_ctr\_t \*)end\_adr;  /\*如果内存这么少，不适合分配\*/  if(start\_adr>end\_adr||end\_adr-start\_adr<BLOCK\_SIZE){  acoral\_mem\_ctrl->state=MEM\_NO\_ALLOC;  return -1;  }  acoral\_mem\_ctrl->state=MEM\_OK;  /\*确定伙伴系统层数num\*/  while(1){  if(end\_adr<=start\_adr+resize\_size)  break;  resize\_size=resize\_size<<1;  num=num<<1;  adjust\_level++;  }  acoral\_mem\_blocks=(acoral\_block\_t \*)end\_adr-num;  save\_adr=(acoral\_u32)acoral\_mem\_blocks;  level=adjust\_level;  if(adjust\_level>LEVEL)  level=LEVEL;  num=num/32;  for(i=0;i<level-1;i++){  num=num>>1;  if(num==0)  num=1;  save\_adr-=num\*4;  save\_adr&=~(4-1);  acoral\_mem\_ctrl->bitmap[i]=(acoral\_u32 \*)save\_adr;  acoral\_mem\_ctrl->num[i]=num;  save\_adr-=num\*4;  save\_adr&=~(4-1);  acoral\_mem\_ctrl->free\_list[i]=(acoral\_32 \*)save\_adr;  for(k=0;k<num;k++){  acoral\_mem\_ctrl->bitmap[i][k]=0;;  acoral\_mem\_ctrl->free\_list[i][k]=-1;  }  acoral\_mem\_ctrl->free\_cur[i]=-1;  }  if(num==0)  num=1;  save\_adr-=num\*4;  save\_adr&=~(4-1);  acoral\_mem\_ctrl->bitmap[i]=(acoral\_u32 \*)save\_adr;  acoral\_mem\_ctrl->num[i]=num;  save\_adr-=num\*4;  save\_adr&=~(4-1);  acoral\_mem\_ctrl->free\_list[i]=(acoral\_32 \*)save\_adr;  for(k=0;k<num;k++){  acoral\_mem\_ctrl->bitmap[i][k]=0;;  acoral\_mem\_ctrl->free\_list[i][k]=-1;  }  acoral\_mem\_ctrl->free\_cur[i]=-1;  /\*如果减去刚才bitmap用的内存刚好是下一level\*/  if(save\_adr<=(start\_adr+(resize\_size>>1)))  adjust\_level--;  if(adjust\_level>LEVEL)  level=LEVEL;  acoral\_mem\_ctrl->level=level;  acoral\_mem\_ctrl->start\_adr=start\_adr;  num=(save\_adr-start\_adr)>>BLOCK\_SHIFT;  acoral\_mem\_ctrl->end\_adr=start\_adr+(num<<BLOCK\_SHIFT);  acoral\_mem\_ctrl->block\_num=num;  acoral\_mem\_ctrl->free\_num=num;  acoral\_mem\_ctrl->block\_size=BLOCK\_SIZE;  i=0;  max\_num=1<<level-1;  o\_num=0;  if(num>0)  acoral\_mem\_ctrl->free\_cur[level-1]=0;  else  acoral\_mem\_ctrl->free\_cur[level-1]=-1;  while(num>=max\_num\*32){  acoral\_mem\_ctrl->bitmap[level-1][i]=-1;;  acoral\_mem\_ctrl->free\_list[level-1][i]=i+1;  num-=max\_num\*32;  o\_num+=max\_num\*32;  i++;  }  if(num==0)  acoral\_mem\_ctrl->free\_list[level-1][i-1]=-1;  while(num>=max\_num){  index=o\_num>>level-1;  acoral\_set\_bit(index,acoral\_mem\_ctrl->bitmap[level-1]);  num-=max\_num;  o\_num+=max\_num;  }  acoral\_mem\_ctrl->free\_list[level-1][i]=-1;    while(--level>0){  index=o\_num>>level;  if(num==0)  break;  cur=index/32;  max\_num=1<<level-1;  if(num>=max\_num){  acoral\_mem\_blocks[BLOCK\_INDEX(o\_num)].level=-1;  acoral\_set\_bit(index,acoral\_mem\_ctrl->bitmap[level-1]);  acoral\_mem\_ctrl->free\_list[level-1][cur]=-1;  acoral\_mem\_ctrl->free\_cur[level-1]=cur;  o\_num+=max\_num;  num-=max\_num;  }  }  acoral\_spin\_init(&acoral\_mem\_ctrl->lock);  return 0;  } |

代码3.1

* 伙伴系统的内存分配

|  |
| --- |
| static void \*r\_malloc(acoral\_u8 level){  acoral\_sr cpu\_sr;  acoral\_u32 index;  acoral\_32 num,cur;  HAL\_ENTER\_CRITICAL();  acoral\_spin\_lock(&acoral\_mem\_ctrl->lock);  acoral\_mem\_ctrl->free\_num-=1<<level;  cur=acoral\_mem\_ctrl->free\_cur[level];  if(cur<0){  num=recus\_malloc(level+1);  if(num<0){  acoral\_spin\_unlock(&acoral\_mem\_ctrl->lock);  HAL\_EXIT\_CRITICAL();  return NULL;  }  index=num>>level+1;  cur=index/32;  acoral\_set\_bit(index,acoral\_mem\_ctrl->bitmap[level]);  acoral\_mem\_ctrl->free\_list[level][cur]=-1;  acoral\_mem\_ctrl->free\_cur[level]=cur;  if((num&0x1)==0)  acoral\_mem\_blocks[BLOCK\_INDEX(num)].level=level;  #ifdef CFG\_TEST  acoral\_print("Malloc-level:%d,num:%d\n",level,num);  #endif  #ifdef CFG\_TEST\_MEM  buddy\_scan();  #endif  acoral\_spin\_unlock(&acoral\_mem\_ctrl->lock);  HAL\_EXIT\_CRITICAL();  return (void \*)(acoral\_mem\_ctrl->start\_adr+(num<<BLOCK\_SHIFT));  }  index=acoral\_ffs(acoral\_mem\_ctrl->bitmap[level][cur]);  index=index+cur\*32;  acoral\_clear\_bit(index,acoral\_mem\_ctrl->bitmap[level]);  if(acoral\_mem\_ctrl->bitmap[level][cur]==0){  acoral\_mem\_ctrl->free\_cur[level]=acoral\_mem\_ctrl->free\_list[level][cur];  }  if(level==acoral\_mem\_ctrl->level-1){  num=index<<level;  if(num+(1<<level)>acoral\_mem\_ctrl->block\_num){  acoral\_spin\_unlock(&acoral\_mem\_ctrl->lock);  HAL\_EXIT\_CRITICAL();  return NULL;  }  }  else{  num=index<<level+1;  if(acoral\_mem\_blocks[BLOCK\_INDEX(num)].level>=0)  num+=(1<<level);  }  if((num&0x1)==0)  acoral\_mem\_blocks[BLOCK\_INDEX(num)].level=level;  #ifdef CFG\_TEST  acoral\_print("malloc-level:%d,num:%d\n",level,num);  #endif  #ifdef CFG\_TEST\_MEM  buddy\_scan();  #endif  acoral\_spin\_unlock(&acoral\_mem\_ctrl->lock);  HAL\_EXIT\_CRITICAL();  return (void \*)(acoral\_mem\_ctrl->start\_adr+(num<<BLOCK\_SHIFT));  } |

## aCoral第二级内存管理机制——资源池系统

资源池系统作为acoral\_module\_init()函数中初始化的第三个模块，并不直接与物理内存相关，而是在伙伴系统的基础上，对内存进行进一步管理。

* 创建资源内存池

|  |
| --- |
| acoral\_err acoral\_create\_pool(acoral\_pool\_ctrl\_t \*pool\_ctrl){  acoral\_pool\_t \*pool;  if(pool\_ctrl->num>=pool\_ctrl->max\_pools)  return ACORAL\_RES\_MAX\_POOL;  pool=acoral\_get\_free\_pool();  if(pool==NULL)  return ACORAL\_RES\_NO\_POOL;  pool->id=pool\_ctrl->type<<ACORAL\_RES\_TYPE\_BIT|pool->id;  pool->size=pool\_ctrl->size;  pool->num=pool\_ctrl->num\_per\_pool;  pool->base\_adr=(void \*)acoral\_malloc(pool->size\*pool->num);  if(pool->base\_adr==NULL)  return ACORAL\_RES\_NO\_MEM;  pool->res\_free=pool->base\_adr;  pool->free\_num=pool->num;  pool->ctrl=pool\_ctrl;  acoral\_pool\_res\_init(pool);  acoral\_list\_add2\_tail(&pool->ctrl\_list,pool\_ctrl->pools);  acoral\_list\_add2\_tail(&pool->free\_list,pool\_ctrl->free\_pools);  pool\_ctrl->num++;  return 0;  } |

# aCoral线程调度机制

线程系统是acoral\_module\_init()函数中初始化的第四个模块，其中包含两个小模块：线程调度机制和线程调度策略。如下代码所示

|  |
| --- |
| void acoral\_thread\_sys\_init(){  acoral\_sched\_mechanism\_init();  acoral\_sched\_policy\_init();  } |

代码4.1

线程调度机制是与硬件相关的一些代码部分，而调度策略就是纯粹软件算法层面的东西。本章就将针对第一个小模块——aCoral线程调度机制进行介绍与学习。

## aCoral线程介绍

* 任务与线程

aCoral执行、调度的基本单位是线程，也就是说aCoral在执行程序的时候一定是在执行某一个线程（中断除外）。aCoral的一个线程，也可称为一个任务。说到线程，估计大家会想到进程。进程是许多操作系统调度和资源分配的基本单位，但是aCoral并没有这样做，而是选择了线程，原因就在于，线程所消耗的资源更小，线程间通信也比进程简单方便，更加容易满足嵌入式操作系统的实时性、适用性等要求。

之前在中断章节提到过，中断发生后需要保存上下文环境。同样的，发生线程切换时也需要保存上下文环境。中断发生后上下文环境是保存在中断模式独有的堆栈里，那么线程切换时1·，上下文环境应该保存到哪里呢？自然是保存到被打断的线程的私有空间里。线程的这一私有空间我们就称之为线程控制块，Thread Control Block.

TCB除了用来在被打断（切换）时保存上下文环境，还保存了一些运行前或运行时需要的参数，包括线程的状态，用来指示线程是在运行还是挂起等；线程优先级，嵌入式实时操作系统有时需要根据优先级来选择优先执行的任务；线程调度策略，嵌入式实时操作系统可以选择不同的调度策略来调度线程……

* 线程描述TCB

正如进程使用PCB描述一样，线程通常使用TCB（Thread Controll Block）描述。TCB其实本质上就是内存中的一块区域，用来存放线程的一些参数。TCB一般包含的内容如图4.1。



图4.1 TCB的一般结构

aCoral中的TCB结构如图4.2所示。**注意，在之后的课程中，此图需要常驻给学生看。**



图4.2

介绍一下几个重要的结构体成员。

res：在aCoral系统中，线程是一种资源，因此有一个称为res的结构体成员。

state：目前，aCoral的线程状态有7种，ACORAL\_THREAD\_STATE\_READY，ACORAL\_THREAD\_STATE\_SUSPEND，ACORAL\_THREAD\_STATE\_RUNNING，ACORAL\_THREAD\_STATE\_EXIT，ACORAL\_THREAD\_STATE\_RELEASE，ACORAL\_THREAD\_STATE\_DELAY，ACORAL\_THREAD\_STATE\_MOVE

prio：优先级

policy：线程调度策略。aCoral中现有的调度策略有ACORAL\_SCHED\_POLICY\_COMM、ACORAL\_SCHED\_POLICY\_PERIOD、ACORAL\_SCHED\_POLICY\_RM、ACORAL\_SCHED\_POLICY\_SLICE、ACORAL\_SCHED\_POLICY\_POSIX。当然用户可以添加自己的调度策略。关于调度策略的内容，具体将在下一章进行学习。

ready：aCoral线程就绪队列。当线程就绪之后，就会被挂载到系统就绪队列acoral\_ready\_queues上，以待CPU调度。

timeout：aCoral超时等待队列。线程申请的互斥量等共享资源被占用时，会将线程挂载到系统超时队列timeout\_queue上。

waiting：等待队列。这里的等待，可能是等待一定的时间，如周期策略线程，或者调用了delay相关接口等，waiting成员就会被挂载到系统延时队列time\_delay\_queue上；还有可能是等待某个资源，那waiting成员就会被挂载的这个资源的等待队列上。

stack：线程堆栈指针，指向的堆栈用于线程切换时保存上下文环境。

* aCoral线程优先级与就绪队列

线程优先级，顾名思义，就是优先级越高的线程，越早被调度。aCoral中的线程优先级为0~ACORAL\_MAX\_PRIO\_NUM，数字越小优先级越高，且不同线程允许拥有相同的优先级。ACORAL\_MAX\_PRIO\_NUM取决于编译时指定的最大线程数量以及是否支持POSIX线程。

就绪队列，顾名思义就是处于ready状态的线程组成的一个集合。aCoral将就绪队列本身定义为一个数组，数组的下标代表优先级，数组的每个元素是同一个优先级的线程组成的链表队列。

|  |
| --- |
| acoral\_queue\_t queue[ACORAL\_MAX\_PRIO\_NUM]; |

同时，为了更快的找到哪个最高的优先级上有就绪的线程，aCoral还定义了一个优先级位图数组bitmap[]，位图的每一位（注意是每一位而不是每个元素）代表了一个优先级，0代表这一优先级没有就绪线程，1则有之。

|  |
| --- |
| acoral\_u32 bitmap[PRIO\_BITMAP\_SIZE]; |

再加上就绪任务总数num，这样，aCoral的一个就绪队列acoral\_rdy\_queue\_t;就定义完成了

|  |
| --- |
| struct acoral\_prio\_array {  acoral\_u32 num; //就绪任务总数  acoral\_u32 bitmap[PRIO\_BITMAP\_SIZE]; //优先级位图  acoral\_queue\_t queue[ACORAL\_MAX\_PRIO\_NUM]; //相同优先级线程队列数组  };  typedef struct acoral\_prio\_array acoral\_prio\_array\_t;  typedef struct{  acoral\_prio\_array\_t array;  }acoral\_rdy\_queue\_t; |

那有因为aCoral为了支持多核，也就是多个CPU的情况，所以定义了就绪队列数组acoral\_ready\_queues，数组的个数就是CPU的个数。

|  |
| --- |
| static acoral\_rdy\_queue\_t acoral\_ready\_queues[HAL\_MAX\_CPU]; |

概括一下，aCoral就绪队列实际上是一个数组acoral\_ready\_queues，每个成员都是一个CPU的私有就绪队列。每个私有就绪队列又是一个数组queue，每个成员都是同样优先级的线程队列。这样，整个aCoral的就绪队列就介绍完毕了。

* aCoral调度机制初始化

在创建线程之前，我们需要对线程机制进行初始化，这就是代码4.1所做的事。具体来说，就是初始化了线程资源池和线程就绪队列。初始化线程资源池之后，创建线程的时候就可以直接从资源池中取出资源作为TCB；初始化就绪队列之后，就可以将线程挂载上去供CPU调度。关于这二者的具体内容，请继续下一节的学习。

## 通用策略线程创建

在上一节学习了aCoral中线程的基本知识后，我们就要来学习aCoral是如何创建一个新的线程的。因为关于调度策略的部分我们还没有学习，所以就先从最简单的通用调度策略线程开始。

* 创建通用策略线程1——创建通用策略线程

创建一个线程，说白了就是，分配一块内存用作TCB，并给TCB中的属性赋值，最后将其挂载到就绪队列上以供CPU调度。

分配TCB所需要的内存空间的方法，已经在第三章第三节资源池处做过介绍，具体就不做详细说明。函数acoral\_alloc\_thread()就是用于从线程资源池中取得一个资源，返回值作为TCB。

需要赋值的属性中，线程的名字、堆栈大小、优先级、CPU都由调用函数传入，调度策略因为是创建策略策略线程，所以就使用ACORAL\_SCHED\_POLICY\_COMM。

函数最终需要返回线程的id。

根据以上描述，尝试补全创建通用策略线程的代码。其中acoral\_comm\_policy\_data\_t结构用于存储普通策略线程的策略相关参数，定义如下

|  |
| --- |
| typedef struct{  acoral\_8 cpu;  acoral\_u8 prio;  acoral\_u8 prio\_type;  }acoral\_comm\_policy\_data\_t; |

* 目的：给线程分配TCB空间，并为其中的字段赋值，完成通用策略线程创建。
* 提供分配TCB函数：acoral\_thread\_t \* acoral\_alloc\_thread()
* 提供通用策略初始化函数：acoral\_id comm\_policy\_thread\_init(acoral\_thread\_t \*thread,void (\*route)(void \*args),void \*args,void \*data)
* 提供常量： ACORAL\_SCHED\_POLICY\_COMM

|  |
| --- |
| acoral\_id create\_comm\_thread(void (\*route)(void \*args),acoral\_u32 stack\_size,void \*args,acoral\_char \*name,acoral\_u8 prio,acoral\_8 cpu){  acoral\_comm\_policy\_data\_t policy\_ctrl;  acoral\_thread\_t \*thread;  /\*分配tcb数据块\*/  thread=acoral\_alloc\_thread();  if(NULL==thread){  acoral\_printerr("Alloc thread:%s fail\n",name);  acoral\_printk("No Mem Space or Beyond the max thread\n");  return -1;  }  thread->name=name;    /\*线程堆栈参数\*/  stack\_size=stack\_size&(~3);  thread->stack\_size=stack\_size;  thread->stack\_buttom=NULL;  /\*设置线程要运行的cpu核心\*/  policy\_ctrl.cpu=cpu;  /\*设置线程的优先级\*/  policy\_ctrl.prio=prio;  policy\_ctrl.prio\_type=ACORAL\_BASE\_PRIO;  thread->policy=ACORAL\_SCHED\_POLICY\_COMM;  thread->cpu\_mask=-1;  return comm\_policy\_thread\_init(thread,route,args,&policy\_ctrl);  } |

代码4.2

* 创建通用策略线程2——通用策略初始化

代码4.2的最后一步就是使用赋值好的通用策略控制块进一步为TCB赋值。其实也可以在上一步中不使用通用策略控制块，直接将那些值赋值给TCB。代码4.2这样的做法只是为了更加统一。赋值完成后，就需要根据指定的堆栈大小，向伙伴系统索要对应的空间。如果返回值不为0，说明堆栈空间分配失败，需要释放已经分配的TCB空间，创建线程失败。

如果堆栈分配成功，则将线程挂载到就绪队列上，函数最终返回资源id，作为线程的id返回给上层函数。请根据上述描述，尝试补全代码4.3。

* 目的使用通用线程策略控制块进一步为TCB赋值，并调用堆栈初始化函数和挂载就绪队列函数。
* 提供线程堆栈初始化函数：acoral\_err acoral\_thread\_init(acoral\_thread\_t \*thread,void (\*route)(void \*args),void (\*exit)(void),void \*args)
* 提供挂载到就绪队列函数：void acoral\_resume\_thread\_by\_id(acoral\_u32 thread\_id)

|  |
| --- |
| acoral\_id comm\_policy\_thread\_init(acoral\_thread\_t \*thread,void (\*route)(void \*args),void \*args,void \*data){  acoral\_sr cpu\_sr;  acoral\_u32 prio;  acoral\_comm\_policy\_data\_t \*policy\_data;  policy\_data=(acoral\_comm\_policy\_data\_t \*)data;  thread->cpu=policy\_data->cpu;  prio=policy\_data->prio;  if(policy\_data->prio\_type==ACORAL\_BASE\_PRIO){  prio+=ACORAL\_BASE\_PRIO\_MIN;  if(prio>=ACORAL\_BASE\_PRIO\_MAX)  prio=ACORAL\_BASE\_PRIO\_MAX-1;  }  thread->prio=prio;  /\*分配线程堆栈空间\*/  if(acoral\_thread\_init(thread,route,acoral\_thread\_exit,args)!=0){  acoral\_printerr("No thread stack:%s\n",thread->name);  HAL\_ENTER\_CRITICAL();  acoral\_release\_res((acoral\_res\_t \*)thread);  HAL\_EXIT\_CRITICAL();  return -1;  }  /\*将线程就绪，并重新调度\*/  acoral\_resume\_thread(thread);  return thread->res.id;  } |

代码4.3

* 创建通用策略线程3——线程堆栈初始化

再来具体看看代码4.3中的线程堆栈初始化函数acoral\_thread\_init()具体做了什么。可能有同学会有疑问了——代码4.3中的acoral\_thread\_init()函数接受的参数thread中，只有关于堆栈的大小和栈底信息（详见代码4.2 注释*线程堆栈参数处*）。那aCoral究竟是如何确定堆栈的呢？这里涉及一个常识，就是堆栈往往是向上生长的，或者说是向低地址生长的，栈顶指针往往是在高地址，入栈操作之后栈顶指针会自减，这就是所谓的向上生长。有了这个常识之后，就可以来看代码4.4了

aCoral中的线程堆栈空间有一个最小值，即必须保证能至少放得下上下文环境。这个最小值设置为ACORAL\_MIN\_STACK\_SIZE，如果传入的堆栈大小小于这个值，就需要增加之。

最后还需要将线程的状态设置为挂起状态，表示还没有被挂载到就绪队列上。

请根据以上描述，尝试补全线程堆栈初始化函数acoral\_thread\_init()。

* 目的：为线程分配堆栈空间，设置线程状态
* 提供分配内存空间函数：acoral\_malloc(size)
* 提供线程状态

|  |
| --- |
| acoral\_err acoral\_thread\_init(acoral\_thread\_t \*thread,void (\*route)(void \*args),void (\*exit)(void),void \*args){  acoral\_sr cpu\_sr;  acoral\_u32 stack\_size=thread->stack\_size;  if(thread->stack\_buttom==NULL){  if(stack\_size<ACORAL\_MIN\_STACK\_SIZE)  stack\_size=ACORAL\_MIN\_STACK\_SIZE;  thread->stack\_buttom=(acoral\_u32 \*)acoral\_malloc(stack\_size);  if(thread->stack\_buttom==NULL)  return ACORAL\_ERR\_THREAD\_NO\_STACK;  thread->stack\_size=stack\_size;  }  thread->stack=(acoral\_u32 \*)((acoral\_8 \*)thread->stack\_buttom+stack\_size-4);  HAL\_STACK\_INIT(&thread->stack,route,exit,args);  /\*cpu\_mask\*/  if(thread->cpu\_mask==-1)  thread->cpu\_mask=0xefffffff;  if(thread->cpu<0)  thread->cpu=acoral\_get\_idle\_maskcpu(thread->cpu\_mask);  if(thread->cpu>=HAL\_MAX\_CPU)  thread->cpu=HAL\_MAX\_CPU-1;  thread->data=NULL;  thread->state=ACORAL\_THREAD\_STATE\_SUSPEND;  /\*继承父线程的console\_id\*/  thread->console\_id=acoral\_cur\_thread->console\_id;  return 0;  } |

代码4.4

* 创建普通策略线程4——线程寄存器环境模拟

代码4.4中，HAL\_STACK\_INIT()函数用于模拟线程寄存器上下文环境。

|  |
| --- |
| HAL\_STACK\_INIT(&thread->stack,route,exit,args); |

我们已经知道，线程在执行时的上下文环境，本质上就是一组寄存器，所以在被调度之前，要先把寄存器的值准备好，以便于被调度执行时，可以直接将这些准备好的值赋给CPU的寄存器。准备这些值，就是将这些值保存在TCB堆栈指针指向的内存中。准备这些值的过程称为线程寄存器环境模拟。

aCoral为了模拟线程上下文环境，定义了结构体hal\_ctx\_t，定义如下：

|  |
| --- |
| typedef struct {  acoral\_u32 cpsr;  acoral\_u32 r0;  acoral\_u32 r1;  acoral\_u32 r2;  acoral\_u32 r3;  acoral\_u32 r4;  acoral\_u32 r5;  acoral\_u32 r6;  acoral\_u32 r7;  acoral\_u32 r8;  acoral\_u32 r9;  acoral\_u32 r10;  acoral\_u32 r11;  acoral\_u32 r12;  acoral\_u32 lr;  acoral\_u32 pc;  }hal\_ctx\_t; |

可以发现，hal\_ctx\_t中的每个成员都对应了CPU的一个寄存器，在调度线程时，只需要按照一定顺序，就可以将结构体中的值一个一个赋值给对应的寄存器。

在这些寄存器中，R0~R12比较简单，赋值可以任意赋。

lr用来存放线程退出函数acoral\_thread\_exit()。这个函数有什么作用呢？请同学们思考一下，一个线程如果执行完了，但还没有到调度的时候该怎么办？接下来CPU不就没有事情干了么？但是CPU是一定要去取指干活的。于是为了避免CPU无事可干的情况，就必须对线程退出后的情况予以处理。线程退出后，通常需需要进行如下工作：释放线程占用的资源、将线程从就绪丢队列上取下、重新调度新的线程。acoral\_thread\_exit()函数就是做这些工作的。那线程退出后，CPU如何自动开始执行这个函数呢？就是通过将lr指向这个函数，而线程函数执行的最后一句程序一定是return语句，这就会将lr寄存器的值赋值给pc，让CPU自动自动跳转到acoral\_thread\_exit()函数开始执行。

pc用来存放线程函数的指针。之前可能会有细心的同学发现，aCoral的TCB结构体中并没有存放线程函数，但是线程一定是要有东西可以执行的，也就是说线程函数是必须的。于是aCoral采用的做法就是将pc的值直接赋值为线程函数，这样，当线程被调度执行时，CPU就会自动执行pc中的指令，即跳转到线程函数中了。

cpsr用来保存程序状态寄存器的值。

最后将结构体整个压入堆栈，线程被调度时会去堆栈指针指向的空间内取得这些模拟值，并送入对应的寄存器中。

请根据以上描述，尝试补全线程寄存器环境模拟函数hal\_stack\_init()。

* 目的：在堆栈中模拟线程上下文环境。
* 提供数据结构：hal\_ctx\_t

|  |
| --- |
| void hal\_stack\_init(acoral\_u32 \*\*stk,void (\*route)(),void (\*exit)(),void \*args){  hal\_ctx\_t \*ctx=(hal\_ctx\_t \*)\*stk;  ctx--;  ctx=(hal\_ctx\_t \*)((acoral\_u32 \*)ctx+1);  ctx->r0=(acoral\_u32)args;  ctx->r1=1;  ctx->r2=2;  ctx->r3=3;  ctx->r4=4;  ctx->r5=5;  ctx->r6=6;  ctx->r7=7;  ctx->r8=8;  ctx->r9=9;  ctx->r10=10;  ctx->r11=11;  ctx->r12=12;  ctx->lr=(acoral\_u32)exit;  ctx->pc=(acoral\_u32)route;  ctx->cpsr=0x0000001fL;  \*stk=(acoral\_u32 \*)ctx;  } |

代码4.5

* 创建通用策略线程5——将线程就绪

创建线程的最后一步，也就是代码4.3的倒数第二行，就是将已经创建好的线程就绪。首先需要判断线程状态是否为挂起，如果不是，就没必要或者不能将线程就绪。而线程挂起状态的设置在代码4.4已经做好了，所以没有问题。

之后就正式需要将线程挂载到就绪队列上了，这里使用函数acoral\_rdyqueue\_add。在挂载这一步时，需要使系统进入临界状态，以防止外界中断破坏新创建的线程的内容。至于acoral\_rdyqueue\_add函数如何将线程挂载到队列上，无非就是将TCB的ready成员插入到aCoral的就绪队列中相应优先级的队列链表，aCroal在这里使用的是尾插法，再将bitmap对应位置1（如果原本没有就绪线程的话）。具体实现请自行阅读aCoral源码。

最后的最后，调用acoral\_sched()，触发调度。根据以上内容尝试补全代码4.6。

* 目的：将线程挂载到就绪队列。
* 提供挂载到就绪队列函数：void acoral\_rdyqueue\_add(acoral\_thread\_t \*new)

|  |
| --- |
| void acoral\_resume\_thread(acoral\_thread\_t \*thread){  acoral\_sr cpu\_sr;  acoral\_8 cpu;  if(!(thread->state&ACORAL\_THREAD\_STATE\_SUSPEND))  return;  #ifdef CFG\_CMP  cpu=thread->cpu;  /\*resumed \*/  if(cpu!=acoral\_current\_cpu){  acoral\_ipi\_cmd\_send(cpu,ACORAL\_IPI\_THREAD\_RESUME,thread->res.id,NULL);  return;  }  #endif  /\*进入临界状态\*/  HAL\_ENTER\_CRITICAL();  acoral\_rdyqueue\_add(thread);  /\*退出临界状态\*/  HAL\_EXIT\_CRITICAL();  acoral\_sched();  } |

代码4.6

* 通用策略线程创建过程小结

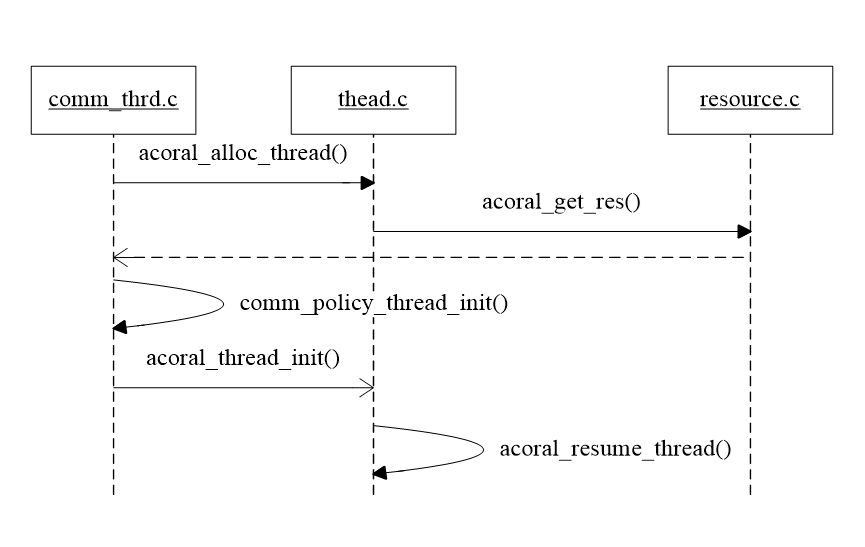


图 4.3

## 线程调度和其他线程机制

上一节中，创建一个通用策略线程的最后的最后一步，就是acoral\_sched()触发调度。所谓线程调度，就是从一个线程切换到另一个线程的过程。那aCoral是如何调度线程的呢？

* 调度前准备

线程调度之前，要先做一些初步的准备工作

1. 判断aCoral是否需要调度。标志：acoral\_need\_sched。aCoral之前4.2小节中的acoral\_resume\_thread()中，就会设置这一标志为true。这个标志很重要，一般在线程正常执行的过程中是不会被置为true的，以保证线程可以正常运行。

|  |
| --- |
| acoral\_set\_need\_sched(true); |

1. 判断aCoral是否处于中断中。标志：acoral\_intr\_nesting。如果系统处于中断，acoral\_intr\_nesting为true，那在中断结束之前是不能触发系统调度的，可以简单理解为中断优先级比所有的线程都高。
2. 判断调度是否被锁。标志：acoral\_sched\_is\_lock。如果调度被锁，acoral\_sched\_is\_lock为true，则不能调度。
3. 判断调度是否开始了。标志：acoral\_start\_sched。aCoral只有在系统所有模块初始化完成后，才能开始调度。在这之前，acoral\_start\_sched为false。

在这些判断都通过之后，aCoral才能开始进行真正的线程调度。请根据以上描述，尝试补全代码4.7。

* 目的：调度前的准备工作，判断是否可以进行调度。
* 提供标志：acoral\_need\_sched、acoral\_intr\_nesting、acoral\_intr\_nesting、acoral\_start\_sched。

|  |
| --- |
| void acoral\_sched(){  /\*如果不需要调度，则返回\*/  if(!acoral\_need\_sched)  return;  if(acoral\_intr\_nesting)  return;  if(acoral\_sched\_is\_lock)  return;  /\*如果还没有开始调度，则返回\*/  if(!acoral\_start\_sched)  return;  /\*这个函数进行简单处理后会直接或间接调用acoral\_real\_sched,或者acoral\_real\_intr\_sched\*/  HAL\_SCHED\_BRIDGE();  return;  } |

代码4.7

* 线程调度1——acoral\_real\_sched

代码4.6的最后一步HAL\_SCHED\_BRIDGE()就是aCoral系统真正的调度，定义如下：

|  |
| --- |
| #define HAL\_SCHED\_BRIDGE() hal\_sched\_bridge\_comm()  void hal\_sched\_bridge\_comm(){  acoral\_sr cpu\_sr;  HAL\_ENTER\_CRITICAL();  acoral\_real\_sched();  HAL\_EXIT\_CRITICAL();  } |

需要注意的是，在调度的过程中，是需要关闭中断的，所以调用了HAL\_ENTER\_CRITICAL()

在调度完成后，再通过HAL\_EXIT\_CRITICAL()打开中断。

现在我们已经可以开始真正的线程调度了。我们首先自然需要关闭调度，防止在调度的过程中被其他条件意外再次触发调度产生问题。然后我们就要去aCoral的就绪队列中去寻找优先级最高的线程，并和当前的线程进行一个切换。

线程切换分为两种情况：

1. 如果旧线程已经执行完毕，那就不需要保存其上下文环境了，直接将新线程的上下文环境写入CPU寄存器就可；这种情况下，aCoral提供了函数HAL\_SWITCH\_TO，参数就是新线程的堆栈栈顶指针。
2. 如果旧线程还未执行完毕，则需要先在旧线程的堆栈中保存旧线程的上下文环境，再向寄存器写入新线程的上下文环境。这种情况下，aCoral提供了函数HAL\_CONTEXT\_SWITCH，参数一为旧线程的堆栈栈顶指针，参数二为新线程的堆栈栈顶指针。

根据以上描述，尝试补全代码4.8。

* 目的：选择就绪队列中最高优先级的线程，触发调度。
* 提供全局变量：aCoral当前线程为acoral\_cur\_thread
* 提供全局变量：aCoral中选择出来的最高优先级线程为acoral\_ready\_thread

|  |
| --- |
| void acoral\_real\_sched(){  acoral\_thread\_t \*prev;  acoral\_thread\_t \*next;  acoral\_set\_need\_sched(false);  prev=acoral\_cur\_thread;  /\*选择最高优先级线程\*/  acoral\_select\_thread();  next=acoral\_ready\_thread;  if(prev!=next){  acoral\_set\_running\_thread(next);  if(prev->state==ACORAL\_THREAD\_STATE\_EXIT){  prev->state=ACORAL\_THREAD\_STATE\_RELEASE;  HAL\_SWITCH\_TO(&next->stack);  return;  }  #ifdef CFG\_CMP  if(prev->state&ACORAL\_THREAD\_STATE\_MOVE){  /\*这个函数开lock后不能使用prev的堆栈\*/  prev->state&=~ACORAL\_THREAD\_STATE\_MOVE;  HAL\_MOVE\_SWITCH\_TO(&prev->move\_lock,0,&next->stack);  return;  }  #endif  /\*线程切换\*/  HAL\_CONTEXT\_SWITCH(&prev->stack,&next->stack);  }  } |

代码4.8

* 选择最高优先级线程

代码4.8中，acoral\_select\_thread()函数就是从就绪队列中寻找到最高优先级的线程，并赋值给系统全局变量acoral\_ready\_thread。

具体来说，寻找的过程如下：

1. 在就绪队列数组中选出当前CPU对应的就绪队列
2. 就绪队列实际上是一个数组，每个成员都是同样优先级的线程队列。所以取出数组并作为函数acoral\_get\_highprio()的参数，返回找到的队列的下标序号。
3. 取出这个队列中的第一个线程并置为就绪线程。

请根据以上描述，尝试补全代码4.9。

* 目的：寻找绪队列中优先级最高的线程并置为就绪线程。
* 提供全局变量：aCoral就绪队列acoral\_ready\_queues
* 提供在优先级数组中寻找最高优先级的函数

acoral\_u32 acoral\_get\_highprio(acoral\_prio\_array\_t \*array)

|  |
| --- |
| void acoral\_select\_thread(){  acoral\_u8 cpu;  acoral\_u32 index;  acoral\_rdy\_queue\_t \*rdy\_queue;  acoral\_prio\_array\_t \*array;  acoral\_list\_t \*head;  acoral\_thread\_t \*thread;  acoral\_queue\_t \*queue;  cpu=acoral\_current\_cpu;  /\*选出当前CPU的就绪队列\*/  rdy\_queue=acoral\_ready\_queues+cpu;  array=&rdy\_queue->array;  /\*找出就绪队列数组中优先级最高的存在就绪线程的成员的序号\*/  index = acoral\_get\_highprio(array);  queue = array->queue + index;  head=&queue->head;  /\*取出队列中的第一个线程\*/  thread=list\_entry(head->next, acoral\_thread\_t, ready);  ACORAL\_ASSERT(thread,"Aseert:In select thread");  /\*设置系统当前已经准备好的线程\*/  acoral\_set\_ready\_thread(thread);  } |

代码4.9

这里有必要解释一下代码4.9中的list\_entry（）函数。它的作用是，返回第一个参数所在的结构体，什么意思呢？然我们看一下list\_entry的代码

|  |
| --- |
| #define list\_entry(ptr, type, member) \  container\_of(ptr, type, member) |

代码4.10

代码4.10中，可以看出，list\_entry被定义为container\_of()这个函数，字面上就可以看出，这是在找一个容器container，即ptr指向的这个member所在的container。代码4.11可以证明这一点。

|  |
| --- |
| #define container\_of(ptr, type, member) ((type \*)((acoral\_8 \*)ptr - offsetof(type,member))) |

代码4.11

说白了就是根据结构体成员，反过来找结构体。比如代码4.9中的list\_entry，就是根据指定的ready成员来找到他所属的acoral\_thread\_t变量。

* 找出就绪队列中优先级最高的线程的优先级（这个并没有标准答案，可以自己想一种可行的算法）

aCoral中的bitmap，存在的目的是为了更快的找到，就绪队列中哪个优先级上有就绪线程。aCoral中总共有130个优先级，所以bitmap的大小应该是至少130位，即5个acoral\_u32的32位变量，所以bitmap实际是一个5\*32的表格，等价于int bitmap[5]。bitmap[0]的第0位（最低位）=0，表示优先级0没有就绪队列；bitmap[1]的第31位（最高位）=0，表示优先级63有就绪队列，以此类推。

* 目的：设计算法，找出32位的优先级位图bitmap中第一个1
* 提供：常量PRIO\_BITMAP\_SIZE等于5。

|  |
| --- |
| acoral\_u32 acoral\_get\_highprio(acoral\_prio\_array\_t \*array){  return acoral\_find\_first\_bit(array->bitmap,PRIO\_BITMAP\_SIZE);  }  acoral\_u32 acoral\_find\_first\_bit(const acoral\_u32 \*b,acoral\_u32 length)  {  acoral\_u32 v;  acoral\_u32 off;  for (off = 0; v = b[off], off < length; off++) {  if (v)  break;  }  return acoral\_ffs(v) + off \* 32;  } |

代码4.12

* 线程阻塞（挂起）

线程阻塞挂起，引起的原因有很多，可以分为外因和内因两大里类。外因就是别的线程调用

* 线程退出

之前在创建线程中，线程寄存器环境模拟这一步提到过，线程执行完成后，会自动去执行acoral\_thread\_exit()这个函数，完成线程退出后的一些工作。acoral\_thread\_exit()其实就是调用了acoral\_kill\_thread()函数。这些工作包括线程占用的资源释放、将线程加入系统的acoral\_res\_release\_queue资源回收队列等待daem线程回收、将TCB资源还给资源池系统等。这部分代码请同学们自行阅读。

|  |
| --- |
| void acoral\_kill\_thread(acoral\_thread\_t \*thread){  acoral\_sr cpu\_sr;  acoral\_8 cpu;  acoral\_evt\_t \*evt;  #ifdef CFG\_CMP  cpu=thread->cpu;  /\*kill \*/  if(cpu!=acoral\_current\_cpu){  acoral\_ipi\_cmd\_send(cpu,ACORAL\_IPI\_THREAD\_KILL,thread->res.id,NULL);  return;  }  #endif  HAL\_ENTER\_CRITICAL();  /\* \*/  /\* \*/  if(thread->state&ACORAL\_THREAD\_STATE\_SUSPEND){  evt=thread->evt;  /\*\*/  if(thread->state&ACORAL\_THREAD\_STATE\_DELAY){  acoral\_spin\_lock(&thread->waiting.prev->lock);  acoral\_spin\_lock(&thread->waiting.lock);  acoral\_list\_del(&thread->waiting);  acoral\_spin\_unlock(&thread->waiting.lock);  acoral\_spin\_unlock(&thread->waiting.prev->lock);  }else  {  /\*\*/  if(evt!=NULL){  acoral\_spin\_lock(&evt->spin\_lock);  acoral\_evt\_queue\_del(thread);  acoral\_spin\_unlock(&evt->spin\_lock);  }  }  }  acoral\_unrdy\_thread(thread);  acoral\_release\_thread1(thread);  HAL\_EXIT\_CRITICAL();  acoral\_sched();  } |

代码4.14

* 线程状态小结

学到这里，我觉得有必要总结一下aCoral中，线程状态的转换。和一般我们看到的，别的系统的线程状态转换有所不同，aCoral的线程状态与其说是转换，不如说是状态的叠加与删除。也就是说，aCoral的一个线程，可以同时拥有几种状态。操作上来说，你可以把aCoral的所有线程状态全部叠加到一个线程上，当然这会出现逻辑上的错误，比如一个线程不可以既是运行中RUNNING，又是挂起SUSPEND。但一些不存在逻辑冲突的状态是完全可以叠加的，比如一个线程，从准备READY状态，到被调度开始运行的RUNNING状态，本质上是在READY状态上叠加了RUNNING状态。同理，我们将一个运行中的线程挂起，本质上是将READY与RUNNING状态删除，添加了SUSPEND状态。那么这些添加或者删除状态是怎么做到的呢？aCoral的实现方法是，每一种状态都用1位来表示，且都有自己的掩码，每次添加或者删除状态，都是通过这些掩码来对TCB的状态成员state来进行位操作。各个状态的掩码定义如下：

|  |
| --- |
| #define ACORAL\_THREAD\_STATE\_MINI 0  #define ACORAL\_THREAD\_STATE\_READY (1<<ACORAL\_THREAD\_STATE\_MINI)  #define ACORAL\_THREAD\_STATE\_SUSPEND (1<<(ACORAL\_THREAD\_STATE\_MINI+1))  #define ACORAL\_THREAD\_STATE\_RUNNING (1<<(ACORAL\_THREAD\_STATE\_MINI+2))  #define ACORAL\_THREAD\_STATE\_EXIT (1<<(ACORAL\_THREAD\_STATE\_MINI+3))  #define ACORAL\_THREAD\_STATE\_RELEASE (1<<(ACORAL\_THREAD\_STATE\_MINI+4))  #define ACORAL\_THREAD\_STATE\_DELAY (1<<(ACORAL\_THREAD\_STATE\_MINI+5))  #define ACORAL\_THREAD\_STATE\_MOVE (1<<(ACORAL\_THREAD\_STATE\_MINI+6)) |

代码

图4.4展示了部分函数，对aCoral线程状态的影响。

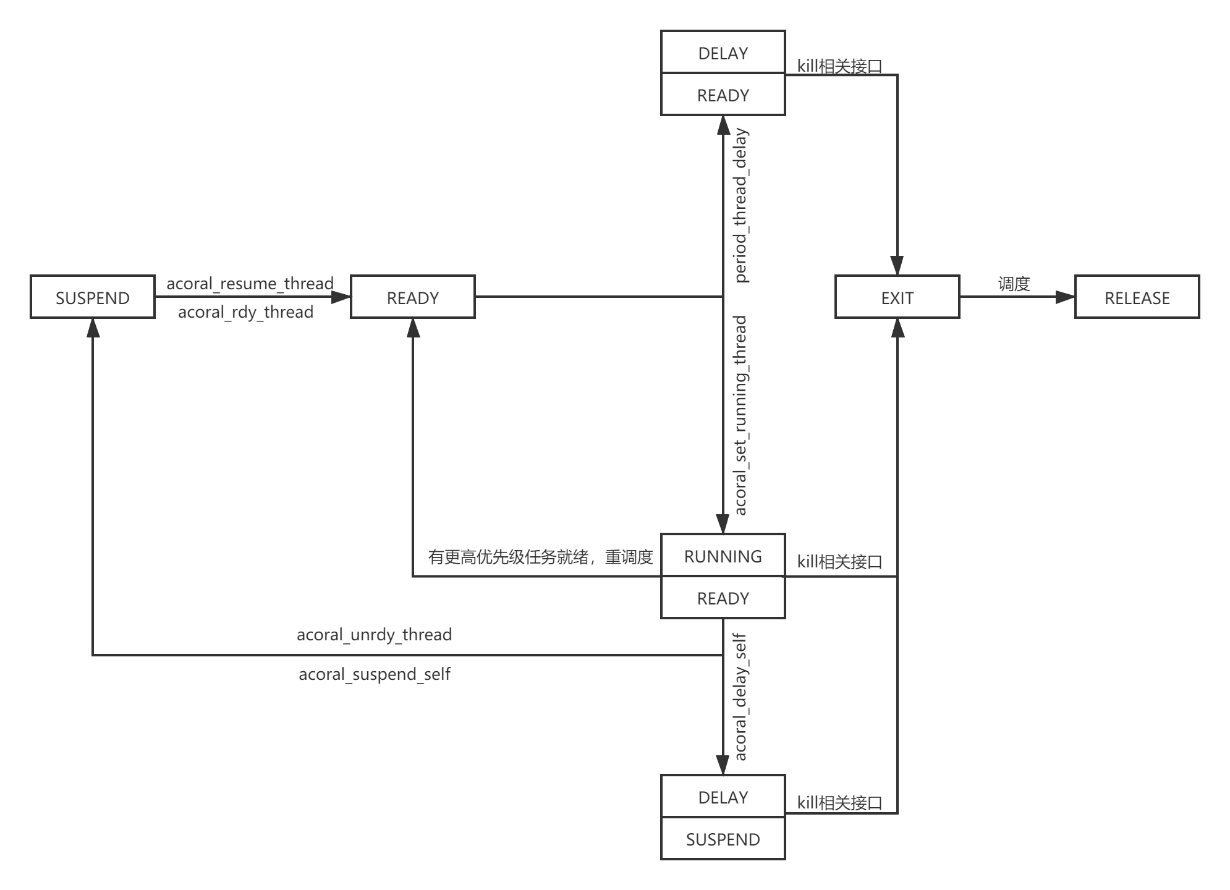


图4.4

# aCoral线程调度策略

本章将针对acoral\_module\_init()函数中的第三个模块——线程模块中的第二部分内容，即aCoral线程调度策略进行介绍与学习。

## 调度策略介绍

调度策略，也可以叫作调度算法，说白了就是什么时候调度哪个线程，或者说如何分配CPU。上一章中，创建通用策略线程的调度策略就是先来先调度，具体来说就是相同优先级的线程先就绪的先被调度，不同优先级的线程优先级高的先被调度。但是这种调度策略过于简单，满足不了嵌入式系统实时性的要求，所以就有了分时策略、周期策略、RM策略等更加复杂、更加优秀的调度策略。本节将介绍线程调度策略相关的基本概念、线程调度策略和调度机制的关系，以及aCoral是如何创建一个策略线程的。

本章中我们约定，除了第四章的通用策略线程，其他策略的线程我们统称为策略线程。

* 基本概念

**任务执行的特性**：

动态性：任务的运行状态不断变化 → 状态变迁图

并发性：多任务并发执行

异步独立性：任务间相互独立，不存在前驱与后继关系

同步性：任务之间非独立，相互依赖

调度点的位置：中断结束、等待资源、周期起止、...

**时间约束：**

截止时间：任务必须完成的时间

执行周期：任务重复出现并处理完成的时间间隔

延迟时间：任务允许等待的时间

RTOS追求的目标：简单、确定性 → 确保实时性约束

通用调度是NP问题；复杂调度往往只用于理论研究aCoral中的策略控制块

* 调度机制与调度策略

调度机制：根据给定调度策略来安排任务的具体执行，如创建任务、就绪任务、挂起任务、任务间通信、...。

调度策略：针对有限的CPU资源，确定哪个任务先执行、在哪个CPU上执行、执行时间、等等的策略。→ 以调度算法的形式体现

调度策略的本质：为任务确定优先级

调度算法：在特定时刻，确定将要运行的任务的一组规则

调度机制和调度策略共同完成内核的的调度功能

## aCoral调度策略实现

* aCoral策略控制块

策略控制块是aCoral中的一种全局变量。在aCoral中，每一种策略对应一个策略控制块。策略控制块中存储了这种策略的控制信息，包括策略线程初始化函数、释放函数、延时函数等信息。策略控制块结构如下：

|  |
| --- |
| typedef struct{  acoral\_list\_t list; //策略链表结点  acoral\_u8 type; //策略类型  acoral\_id (\*policy\_thread\_init)(acoral\_thread\_t \*,void (\*route)(void \*args),void \*,void \*); //策略线程初始化函数  void (\*policy\_thread\_release)(acoral\_thread\_t \*); //策略线程释放函数  void (\*delay\_deal)(); //与延时相关的处理函数，如周期、轮转等策略都要用到类似的延时机制  acoral\_char \*name; //策略参数  }acoral\_sched\_policy\_t; |

代码5.1

创建某一种策略线程的时候，就需要根据策略名找到对应的策略控制块，并使用其中的初始化函数。

在之前的普通线程创建中，并没有用到普通线程策略控制块comm\_policy，可能是因为编写aCoral的时候，觉得普通线程的策略过于简单，没有必要使用策略控制块，也可能是忘了。

* aCoral调度策略初始化

aCoral的调度策略初始化就是代码4.1的第二个函数。调度策略初始化所做的工作就是，定义每一种策略的策略控制块并为其赋值，最后将其挂载到策略链表policy\_list。这样，之后创建策略线程的时候就可以从策略链表上取下策略控制块，调用其中的函数。

* aCoral创建策略线程

与第4章创建通用策略线程不同，现在我们要创建一个其他策略的线程。创建的过程和创建通用策略线程大致差不多，都是先分配TCB空间，再根据传入的参数为TCB成员赋值，最后调用线程初始化函数。不同点在于，最后一步中，创建通用策略函数是直接调用通用策略线程的初始化函数comm\_policy\_thread\_init（）函数完成初始化，而创建其他策略线程，因为策略的不同，则需要调用acoral\_policy\_thread\_init（）函数，根据不同的策略完成相应的初始化。

尝试根据以上描述，补全代码5.2。

* 目的：创建策略线程
* 提供分配TCB函数：acoral\_thread\_t \* acoral\_alloc\_thread()
* 提供策略线程初始化函数：acoral\_id acoral\_policy\_thread\_init(acoral\_u32 policy,acoral\_thread\_t \*thread,void (\*route)(void \*args),void \*args,void \*data)

|  |
| --- |
| acoral\_id create\_thread\_ext(void (\*route)(void \*args),acoral\_u32 stack\_size,void \*args,acoral\_char \*name,void \*stack,acoral\_u32 sched\_policy,void \*data){  acoral\_thread\_t \*thread;  /\*分配tcb数据块\*/  thread=acoral\_alloc\_thread();  if(NULL==thread){  acoral\_printerr("Alloc thread:%s fail\n",name);  acoral\_printk("No Mem Space or Beyond the max thread\n");  return -1;  }  thread->name=name;  stack\_size=stack\_size&(~3); //地址四位对齐  thread->stack\_size=stack\_size;  if(stack!=NULL)  thread->stack\_buttom=(acoral\_u32 \*)stack;  else  thread->stack\_buttom=NULL;  thread->policy=sched\_policy;  return acoral\_policy\_thread\_init(sched\_policy,thread,route,args,data);  } |

代码5.2

* 初始化策略线程

代码5.2的最后，调用了acoral\_policy\_thread\_init()函数。这个函数将根据传入的调度策略名称，调用acoral\_get\_policy\_ctrl()函数，找到aCoral中对应的策略控制块，再调用策略控制块中的初始化函数完成线程堆栈的初始化，并将线程挂载到就绪队列上。这里的初始化函数和代码4.2的作用基本类似，区别就在于代码4.2是专用于通用策略线程的初始化函数，而这里是某一策略控制块中用于某一种策略的初始化函数。关于策略初始化函数policy\_thread\_init，将在下一小节进行介绍。

尝试根据以上描述，补全代码5.3。

* 目的：查找策略控制块并调用策略控制块中某一策略的初始化函数。
* 提供查找策略控块函数acoral\_sched\_policy\_t \*acoral\_get\_policy\_ctrl(acoral\_u8 type)

|  |
| --- |
| acoral\_id acoral\_policy\_thread\_init(acoral\_u32 policy,acoral\_thread\_t \*thread,void (\*route)(void \*args),void \*args,void \*data){  acoral\_sr cpu\_sr;  acoral\_sched\_policy\_t \*policy\_ctrl;.。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。·  policy\_ctrl=acoral\_get\_policy\_ctrl(policy);  /\*没有找到策略控制块\*/  if(policy\_ctrl==NULL||policy\_ctrl->policy\_thread\_init==NULL){  /\*释放资源\*/  HAL\_ENTER\_CRITICAL();  acoral\_release\_res((acoral\_res\_t \*)thread);  HAL\_EXIT\_CRITICAL();  acoral\_printerr("No thread policy support:%d\n",thread->policy);  return -1;  }  return policy\_ctrl->policy\_thread\_init(thread,route,args,data);  } |

代码5.3

* 根据策略名查找策略控制块

代码5.3中，函数acoral\_get\_policy\_ctrl()根据传入的参数type，遍历策略链表policy\_list，找到对应的策略控制块。

根据以上描述，尝试补全代码5.4

* 目的：根据策略类型type找到策略控制块
* 提供：aCoral的策略链表policy\_list

|  |
| --- |
| acoral\_sched\_policy\_t \*acoral\_get\_policy\_ctrl(acoral\_u8 type){  acoral\_list\_t \*tmp,\*head;  acoral\_sched\_policy\_t \*policy\_ctrl;  head=&policy\_list.head;  tmp=head;  for(tmp=head->next;tmp!=head;tmp=tmp->next){  /\*根据策略控制块的list成员找到所属的策略控制块\*/  policy\_ctrl=list\_entry(tmp,acoral\_sched\_policy\_t,list);  if(policy\_ctrl->type==type)  return policy\_ctrl;  }  return NULL;  } |

代码5.4

## aCoral的分时策略

在创建策略线程之前，aCoral得认识我们指定的这一种调度策略。所以我们要先往aCoral中注册调度策略。注册调度策略的做法就是把赋值好的调度策略控制块挂载到策略链表上。

本小节，我们将学习分时策略，尝试将其注册到aCoral中去。同时，分时策略还依赖于aCoral中一种叫做时钟中断的东西，我们也将进行学习。

* 分时策略

分时策略又称为时间片轮转策略。时间片即CPU分配给各个程序的时间，每个线程被分配一个时间段，称作它的时间片，即该进程允许运行的时间，使各个程序从表面上看是同时进行的。如果在时间片结束时进程还在运行，则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则CPU当即进行切换。而不会造成CPU资源浪费。

由此可知，分时策略最重要的指标就是时间片的大小。时间片太大，单个线程执行时间过长，其他线程就会得不到及时的调度；时间片太小，线程切换过于频繁，而线程切换是需要消耗时间和空间资源的，过于频繁的切换会造成资源的浪费。所以，为每个线程分配大小适合的时间片是非常重要的。目前，aCoral还是需要用户创建线程的时候，手动为线程指定一个大小的时间片。

* 注册分时策略控制块

在能注册分时策略线程之前，需要先往aCoral中注册分时策略。之前说过，注册策略就是将策略控制块赋值，再挂载到策略队列policy\_list。

请根据以上描述尝试补全代码5.5和5.6。

* 目的：注册分时策略
* 提供：常量ACORAL\_SCHED\_POLICY\_SLICE
* 提供：策略队列policy\_list

|  |
| --- |
| acoral\_sched\_policy\_t slice\_policy;  void slice\_policy\_init(){  slice\_policy.type=ACORAL\_SCHED\_POLICY\_SLICE;  slice\_policy.policy\_thread\_release=slice\_policy\_thread\_release;  slice\_policy.policy\_thread\_init=slice\_policy\_thread\_init;  slice\_policy.delay\_deal=slice\_delay\_deal;  slice\_policy.name="slice";  acoral\_register\_sched\_policy(&slice\_policy);  } |

代码5.5

|  |
| --- |
| void acoral\_register\_sched\_policy(acoral\_sched\_policy\_t \*policy){  acoral\_list\_add2\_tail(&policy->list,&policy\_list.head);  } |

代码5.6

* 分时策略线程初始化函数

代码5.3中，假如创建的是分时策略线程，那么找到的就是分时策略控制块slice\_policy，调用的policy\_thread\_init()函数就是代码5.5中注册的slice\_policy\_thread\_init()。

其实每个策略的初始化函数所做的工作都差不多，都是在代码4.3的基础上，加入对某个特定策略相关的一些参数的赋值。比如代码5.4中就加入了将指定的时间片赋值给TCB中的slice成员，其他都与代码4.3大同小异。CPU就会根据TCB中的slice成员值，对分时策略线程进行调度。

要注意的是，用户创建线程的时候，传入的时间片单位是毫秒，而CPU对分时、周期这类和时间有关的策略线程的处理，是参照系统时钟中断tick的，所以需要把时间片从毫秒转换成tick，这个工作由函数TIME\_TO\_TICKS完成。

* 目的：为TCB中的成员赋值，并调用堆栈初始化函数和挂载就绪队列函数。
* 提供：时间到tick转换函数TIME\_TO\_TICKS(time)

|  |
| --- |
| acoral\_id slice\_policy\_thread\_init(acoral\_thread\_t \*thread,void (\*route)(void \*args),void \*args,void \*data){  acoral\_sr cpu\_sr;  acoral\_u32 prio;  acoral\_slice\_policy\_data\_t \*policy\_data;  slice\_policy\_data\_t \*private\_data;  if(thread->policy==ACORAL\_SCHED\_POLICY\_SLICE){  policy\_data=(acoral\_slice\_policy\_data\_t \*)data;  thread->cpu=policy\_data->cpu;  prio=policy\_data->prio;  if(policy\_data->prio\_type==ACORAL\_BASE\_PRIO){  prio+=ACORAL\_BASE\_PRIO\_MIN;  if(prio>=ACORAL\_BASE\_PRIO\_MAX)  prio=ACORAL\_BASE\_PRIO\_MAX-1;  }  thread->prio=prio;  private\_data=(slice\_policy\_data\_t \*)acoral\_malloc2(sizeof(slice\_policy\_data\_t));  if(private\_data==NULL){  acoral\_printerr("No level2 mem space for private\_data:%s\n",thread->name);  HAL\_ENTER\_CRITICAL();  acoral\_release\_res((acoral\_res\_t \*)thread);  HAL\_EXIT\_CRITICAL();  return -1;  }  private\_data->slice\_ld=TIME\_TO\_TICKS(policy\_data->slice);  thread->slice=private\_data->slice\_ld;  thread->private\_data=private\_data;  thread->cpu\_mask=-1;  }  /\*堆栈初始化函数\*/  if(acoral\_thread\_init(thread,route,acoral\_thread\_exit,args)!=0){  acoral\_printerr("No thread stack:%s\n",thread->name);  HAL\_ENTER\_CRITICAL();  acoral\_release\_res((acoral\_res\_t \*)thread);  HAL\_EXIT\_CRITICAL();  return -1;  }  /\*将线程就绪，并重新调度\*/  acoral\_resume\_thread(thread);  return thread->res.id;  } |

代码5.4

* 时钟中断ticks中的处理

上文中提到，CPU会根据TCB中的slice成员值，调度分时策略线程。那具体是怎么调度的呢？CPU怎么知道当前线程（假设是分时线程）的时间片用完了呢？这就是时钟中断ticks所做的工作了。

时钟中断ticks中断服务程序如代码5.7所示.其中acoral\_policy\_delay\_deal()就是处理和时间相关的策略线程的。acoral\_policy\_delay\_deal()定义如代码5.8所示。

|  |
| --- |
| void acoral\_ticks\_entry(acoral\_vector vector){  ticks++;  acoral\_printdbg("In ticks isr\n");  if(acoral\_start\_sched==true){  time\_delay\_deal();  acoral\_policy\_delay\_deal();  /\*--------------------\*/  /\* 超时链表处理函数\*/  /\* pegasus 0719\*/  /\*--------------------\*/  timeout\_delay\_deal();  }  } |

代码5.7

|  |
| --- |
| void acoral\_policy\_delay\_deal(){  acoral\_list\_t \*tmp,\*head;  acoral\_sched\_policy\_t \*policy\_ctrl;  head=&policy\_list.head;  tmp=head;  for(tmp=head->next;tmp!=head;tmp=tmp->next){  policy\_ctrl=list\_entry(tmp,acoral\_sched\_policy\_t,list);  if(policy\_ctrl->delay\_deal!=NULL)  policy\_ctrl->delay\_deal();  }  } |

代码5.8

代码5.8中，将去遍历策略控制块链表，调用每个策略控制块的delay\_deal()成员函数（如果存在的话）。继续以分时策略为例，代码5.5中，分时策略的delay\_deal()成员函数就是slice\_delay\_deal函数。slice\_delay\_deal（）函数将会判断当前线程是不是分时策略线程。如果是的话，则将当前线程TCB的slice成员值减1。如果减到0，就表示当前线程时间片已经用完，触发调度。根据以上描述尝试补全代码5.9

* 目的：将分时策略线程的时间片slice减1，如果用完slice，则触发调度
* 提供：当前线程指针acoral\_cur\_thread

|  |
| --- |
| void slice\_delay\_deal(){  acoral\_thread\_t \*cur;  slice\_policy\_data\_t \*data;  cur=acoral\_cur\_thread;  if(cur->policy==ACORAL\_SCHED\_POLICY\_SLICE){  cur->slice--;  if(cur->slice<=0){  data=(slice\_policy\_data\_t \*)cur->private\_data;  cur->slice=data->slice\_ld;  acoral\_thread\_move2\_tail(cur);  }  }  } |

代码5.9

代码5.9的最后，acoral\_thread\_move2\_tail(cur)将当前线程挂载到就绪队列队尾，并触发调度。这样，分时策略线程在用完一个时间片后，就退居二线，等待下一次调度。

现在，我们回过头来看本课程的主线——acoral\_module\_init()函数，其中初始化的第五个模块——时钟模块，就与ticks时钟中断有密切的关系。这个模块初始化主要就是对aCoral中除了就绪队列以外的的另外两个队列——延时队列和超时队列进行初始化。代码5.7 time\_delay\_deal()和timeout\_delay\_deal()这两个函数就是用来处理这两个队列上的线程的。具体关于这两个队列的内容，请同学们自行学习。

## aCoral的RM策略

* RM策略介绍
* 注册RM策略控制块
* 分时策略线程初始化函数

# aCoral线程交互系统

aCoral线程交互系统是acoral\_module\_init()函数中初始化的第六个模块，本章就将针对aCoral线程交互系统进行介绍与学习。

aCoral在acoral\_module\_init()中初始化线程交互系统代码如代码6.1和代码6.2：

|  |
| --- |
| void acoral\_evt\_sys\_init(){  acoral\_evt\_pool\_init();  } |

代码6.1

|  |
| --- |
| void acoral\_evt\_pool\_init(){  acoral\_evt\_pool\_ctrl.type=ACORAL\_RES\_EVENT;  acoral\_evt\_pool\_ctrl.size=sizeof(acoral\_evt\_t);  acoral\_evt\_pool\_ctrl.num\_per\_pool=8;  acoral\_evt\_pool\_ctrl.num=0;  acoral\_evt\_pool\_ctrl.max\_pools=4;  acoral\_pool\_ctrl\_init(&acoral\_evt\_pool\_ctrl);  #ifdef CFG\_MSG  acoral\_msg\_sys\_init();  #endif  } |

代码6.2

由上可知，线程交互系统的初始化其实就是在初始化事件evt资源池，以及如果定义CFG\_MSG标志，则还会初始化消息系统。本章我们主要学习事件evt资源在线程交互中的使用，消息我们会大致介绍一下。

首先我们先来看一下事件控制块（ECB）acoral\_evt\_t的定义，如代码6.3

|  |
| --- |
| typedef struct {  acoral\_res\_t res;  acoral\_u8 type;  acoral\_spinlock\_t spin\_lock;  acoral\_32 count;  acoral\_queue\_t wait\_queue;  acoral\_char \*name;  void \*data;  }acoral\_evt\_t; |

代码6.3

各个成员的作用如下：

1. res：事件控制块类似TCB，是一种资源
2. type：ECB类型：互斥量、信号量、邮箱
3. spin\_lock：确保多核访问互不干扰
4. count：互斥量的状态或优先级，或临界资源实例数量
5. wait\_queue：等待该事件的任务队列：优先级排序
6. name：ECB名称
7. data：占有该互斥量的任务指针，或挂载传递的邮件

其中，type类型所指示的三种类型，互斥量专用于线程之间的互斥机制，信号量可以用于线程之间的互斥和同步，邮箱专用于线程之间的通信机制。此外代码6.2中初始化的消息系统也用于线程之间的通信。所以，根据这个顺序，我们先来介绍aCoral中的互斥机制——互斥量，而信号量我们主要放到同步机制一节进行学习。

## 互斥机制——互斥量

* 概念

mutex，一个可以处于两态之一的变量 —— 解锁和加锁：只需要一个二进制位表示 → 0、1

* 特征

使用两个过程控制临界区访问；mutex\_lock、mutex\_unlock

* 理解

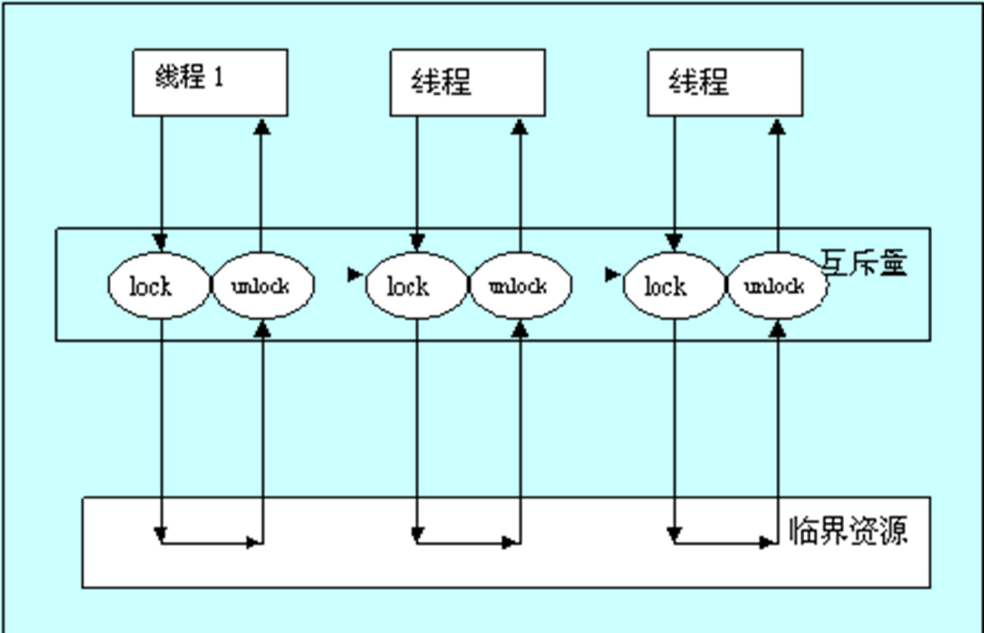


图6.1

* 基本思想

多个线程访问同一临界资源时，必须互斥访问

* 临界资源

全局变量、表格、队列、外设（如打印机）、...

* 控制方式

互斥量 —— 在同一个时刻，只能有一个任务拥有互斥量。

* 临界点

任务刚开始执行临界区代码和退出临界区代码的时刻 —— HAL\_ENTER\_CRITICAL、HAL\_EXIT\_CRITICAL

* aCoral互斥量的count成员

在创建互斥量之前，我觉得有必要先解释一下互斥量ECB中的count成员的含义，这对于我们理解aCoral中互斥量的一些操作和后面优先级反转的现象有帮助。

首先，请同学们思考一下，之前说过，互斥量mutex是一个可以处于两态之一的变量。它决定了当前线程能否申请到临界资源。那么，互斥量就肯定会有一个方法，用来判定这个互斥量是否已经被占用了，对吧？最简单的，就是直接将互斥量定义为一个整型变量，0就是被占用，1就是没被占用。但是这样的做法过于简单，无法满足一些其他的需求。所以，在aCoral的ECB中，定义了一个count成员用来判定互斥量是否被占用。除此以外，count还有另外的一些作用。count成员是一个32位的变量，但实际上它起作用的只有23~0位，这点记住即可。那么count就可以表示如下：

其中23~16位表示这个互斥量的优先级天花板。至于什么是优先级天花板，它有什么作用，我们后面会解释，这里只需要记住，这个值在互斥量被创建的时候就确定了且不会改变。

15~8位表示这个互斥量已经被占用时，因为尝试申请互斥量而被阻塞的线程中最高的优先级。

7~0位在互斥量没有被上锁时，为全1，表示互斥量可用；在被上锁，也就是被占用时，会被赋值为占用它的线程的原始优先级。之所以说是原始优先级，是因为占用线程在使用互斥量的过程中可能被提升优先级，那在释放互斥量之后就要恢复之前的优先级，那就是从count成员的7~0位取值。

在搞清楚count的含义后，我们就可以来尝试创建一个互斥量了。

* aCoral创建互斥量

创建互斥量，本质上来说和创建线程等操作没有区别，都是先从资源池系统处获得一片固定大小的资源，再为其中的成员赋值。对于互斥量来说，就是获得一片内存空间作为ECB，并为几个关键成员，count、type、data赋值，再初始化等待队列wait\_queue。

尝试根据以上描述，结合代码6.3中ECB各个成员的含义，补全代码6.4.

* 目的：给互斥量分配ECB空间，为其中的字段赋值，完成互斥量的创建。
* 提供：事件资源ECB分配函数acoral\_alloc\_evt()
* 提供：常量ACORAL\_EVENT\_MUTEX，表示事件为互斥量。

|  |
| --- |
| acoral\_evt\_t \*acoral\_mutex\_create(acoral\_u8 prio, acoral\_u32 \*err)  {  acoral\_evt\_t \*evt;  /\* 是否大于最小优先线，和是否该优先级已占用\*/  evt = acoral\_alloc\_evt(); /\* alloc a RAM for the event\*/  if (NULL == evt)  {  \*err = MUTEX\_ERR\_NULL;  return NULL;  }  evt->count = (prio << 16) | MUTEX\_AVAI | MUTEX\_U\_MASK;  evt->type = ACORAL\_EVENT\_MUTEX;  evt->data = NULL;  acoral\_evt\_init(evt);  return evt;  } |

代码6.4

* aCoral初始化互斥量

代码6.4中，acoral\_evt\_init(evt)就是为了初始化ECB中wait\_queue成员。wait\_queue成员为等待该事件的任务队列，并且按优先级排序。函数实现如代码6.5：

|  |
| --- |
| void acoral\_evt\_init(acoral\_evt\_t \*evt){  acoral\_spin\_init(&evt->spin\_lock);  acoral\_list\_init(&evt->wait\_queue.head);  } |

代码6.5

* aCoral申请互斥量1——非阻塞式

在创建完了互斥量之后，我们就可以申请互斥量了。要记得，申请互斥量的主体是线程，申请的目的是为了线程访问临界资源时不被其他线程打扰。

我们先来看一种比较简单的申请互斥量的实现。在linux中，有阻塞式与非阻塞式IO两种IO方式，在aCoral中，申请互斥量也有阻塞与非阻塞两种。非阻塞式相比另一种，实现上相对简单一点，在申请的互斥量已经被占用的情况下，申请线程不会被阻塞，就当无事发生。

互斥量能否被线程申请到，取决于ECB的count成员低8位。当互斥量没有被占用，即申请成功之后，count成员的低8位将被置位，具体请参考《aCoral互斥量的count成员》这一知识点。另外，ECB的data成员也将指向申请互斥量的线程，也就是当前线程了。当互斥量被占用时，则申请失败，申请线程不会被阻塞。

要注意的是，以上申请互斥量的过程，必须都在临界区执行。

请根据以上描述，结合知识点**“aCoral互斥量的count成员”**补全代码6.6

* 目的：非阻塞式申请互斥量
* 提供：常量MUTEX\_L\_MASK= 0x00FF
* 提供：常量MUTEX\_AVAI= 0x00FF
* 提供：常量MUTEX\_U\_MASK =0xFF00

|  |
| --- |
| acoral\_u32 acoral\_mutex\_trypend(acoral\_evt\_t \*evt)  {  acoral\_sr cpu\_sr;  acoral\_thread\_t \*cur;  if(acoral\_intr\_nesting>0)  return MUTEX\_ERR\_INTR;    cur=acoral\_cur\_thread;    /\*进入临界区\*/  HAL\_ENTER\_CRITICAL();  acoral\_spin\_lock(&evt->spin\_lock);  if (NULL== evt)  {  acoral\_spin\_unlock(&evt->spin\_lock);  HAL\_EXIT\_CRITICAL();  return MUTEX\_ERR\_NULL;  }    if ((acoral\_u8)(evt->count & MUTEX\_L\_MASK) == MUTEX\_AVAI)  {  /\* 申请成功\*/  evt->count &= MUTEX\_U\_MASK;  evt->count |= cur->prio;  evt->data = (void\*)cur;  acoral\_spin\_unlock(&evt->spin\_lock);  /\*退出临界区\*/  HAL\_EXIT\_CRITICAL();  return MUTEX\_SUCCED;  }  acoral\_spin\_unlock(&evt->spin\_lock);  /\*退出临界区\*/  HAL\_EXIT\_CRITICAL();  return MUTEX\_ERR\_TIMEOUT;  } |

代码6.6

* aCoral申请互斥量2——阻塞式

代码6.6中，如果互斥量被占用，申请线程不会被阻塞。这纵然是一种比较温和的处理方式，但是很多情况下，线程申请不到自己需要的资源，它就无法继续执行下去，这时线程只能将自己挂起，不然不就是占着CPU不干活了么？所以阻塞式申请互斥量才是更加常见的申请互斥量的方式。

在阻塞式申请互斥量中，当互斥量没有被占用时，做法和非阻塞式基本一致。当互斥量已被占用，情况就比较复杂了。首先要处理优先级反转的问题。关于优先级反转的内容，将在后面进行学习。然后就要将当前申请互斥量的线程阻塞，并加入互斥量的等待队列wait\_queue和系统的超时队列timeout\_queue上，触发调度。

这里重点讲一下系统超时队列。aCoral中的超时队列起到如下作用：比如在申请互斥量时，互斥量已被占用，则申请线程被挂起，加入超时队列，在到达指定的超时时间timeout之前，如果互斥量被释放了，并且刚刚被挂起的线程是这个互斥量等待队列中优先级最高的，则这个线程依然可以获得这个互斥量，申请互斥量成功。代码6.7申请互斥量函数的第二个参数timeout就是这个超时时间，它将被赋值给线程TCB中的delay成员，并且在每个时钟中断到来时被减一，这个操作是在代码5.7的timeout\_delay\_deal()中完成的。超时机制本质上就是一种容错机制，可以让线程申请互斥量的成功率更高。另外提一下，timeout=0时，意为无限等待直到互斥量释放。

在触发调度之后，等到再次调度到刚刚被阻塞的线程时，无非两种可能。第一种是互斥量被释放，被阻塞线程在互斥量等待队列中优先级最高，被调度了；第二种是指定的timeout超时时间到了，被阻塞线程被从超时队列上取下，然后被调度。第一种情况当然是申请成功了，而第二种就属于申请超时，失败了。所以，在触发调度之后的代码中，我们就要马上对申请是否超时进行判断

尝试根据以上描述，补全代码6.7

* 目的：阻塞式申请互斥量
* 提供：线程阻塞函数void acoral\_unrdy\_thread(acoral\_thread\_t \*thread)
* 提供：互斥量等待队列添加函数 void acoral\_evt\_queue\_add(acoral\_evt\_t \*evt,acoral\_thread\_t \*new)
* 提供：系统超时队列添加函数 void timeout\_queue\_add(acoral\_thread\_t\*)
* 提供：系统超时队列删除函数 void timeout\_queue\_del(acoral\_thread\_t\*)

|  |
| --- |
| acoral\_u32 acoral\_mutex\_pend(acoral\_evt\_t \*evt, acoral\_time timeout){  acoral\_sr cpu\_sr;  acoral\_u8 highPrio;  acoral\_u8 ownerPrio;  acoral\_thread\_t \*thread;  acoral\_thread\_t \*cur;  if(acoral\_intr\_nesting>0)  return MUTEX\_ERR\_INTR;  cur=acoral\_cur\_thread;  HAL\_ENTER\_CRITICAL();  acoral\_spin\_lock(&evt->spin\_lock);  if (NULL== evt){  acoral\_spin\_unlock(&evt->spin\_lock);  HAL\_EXIT\_CRITICAL();  return MUTEX\_ERR\_NULL;  }  if ((acoral\_u8)(evt->count & MUTEX\_L\_MASK) == MUTEX\_AVAI){  /\* 申请成功\*/  evt->count &= MUTEX\_U\_MASK;  evt->count |= cur->prio;  evt->data = (void\*)cur;  acoral\_spin\_unlock(&evt->spin\_lock);  HAL\_EXIT\_CRITICAL();  return MUTEX\_SUCCED;  }    /\* 处理优先级反转\*/  highPrio = (acoral\_u8)(evt->count >> 8);  ownerPrio = (acoral\_u8)(evt->count & MUTEX\_L\_MASK);  thread = (acoral\_thread\_t\*)evt->data;  if (thread->prio>cur->prio)  {  /\*有可能优先级反转，继承最高优先级\*/  if(cur->prio<highPrio)  {  highPrio=cur->prio;  evt->count &= ~MUTEX\_U\_MASK;  evt->count |= highPrio << 8;  }  acoral\_thread\_change\_prio(thread,highPrio);  }  /\*阻塞线程\*/  acoral\_unrdy\_thread(cur);  /\*加入互斥量等待队列\*/  acoral\_evt\_queue\_add(evt,cur);  if (timeout > 0)  {  /\*加载到超时队列\*/  cur->delay = TIME\_TO\_TICKS(timeout);  timeout\_queue\_add(cur);  }  acoral\_spin\_unlock(&evt->spin\_lock);  HAL\_EXIT\_CRITICAL();  /\*触发调度\*/  acoral\_sched();  HAL\_ENTER\_CRITICAL();  acoral\_spin\_lock(&evt->spin\_lock);  /\*申请互斥量超时\*/  if(evt->data!=cur&&timeout>0&&cur->delay<=0){  acoral\_printk("Time Out Return\n");  acoral\_evt\_queue\_del(cur);  acoral\_spin\_unlock(&evt->spin\_lock);  HAL\_EXIT\_CRITICAL();  return MUTEX\_ERR\_TIMEOUT;  }  /\*在超时时间内申请成功\*/  timeout\_queue\_del(cur);  if(evt->data!=cur){  acoral\_printk("Err Ready Return\n");  acoral\_evt\_queue\_del(cur);  acoral\_spin\_unlock(&evt->spin\_lock);  HAL\_EXIT\_CRITICAL();  return MUTEX\_ERR\_RDY;  }    return MUTEX\_SUCCED;  } |

代码6.7

* timeout\_delay\_deal
* 优先级反转与优先级继承

上一个知识点中，申请信号量时提到了对于优先级反转的操作。那么什么是优先级反转呢？我们来看图6.2。



图6.2

任务优先级T1>T2>T3。

* 0时刻T3到达并开始执行，t0时刻获得共享资源S。
* t1时刻T1到达并抢占T3开始执行，t2时刻T1想要获取S，但是S已经被T3上锁，T1被阻塞，T3继续占有S执行。
* t3时刻T2到达并抢占T3开始执行，t4时刻T2执行完毕，T3继续执行到t5，释放S，唤醒T1继续执行到结束。

上面这个情境中，任务T1的优先级最高，但因为被T3占用的S阻塞，低优先级的T2反而获得了CPU，延迟了T1的执行，这种现象就叫做优先级反转。优先级反转，本质上就是高优先级任务被低优先级任务阻塞，中优先级任务渔翁得利。

RTOS中，任务的优先级反映了任务的紧急程度。如果高优先级任务反而执行权得不到保障，这就会对系统的实时性造成影响，这是不能接受的。aCoral为了解决优先级反转，采用了一种方案——优先级继承。

传统的优先级继承做法，就是在线程申请的互斥量已被占用且占用线程的优先级低于申请线程时，提高占用线程的优先级到互斥量等待队列中最高的优先级，如图6.3



图6.3

任务优先级T1>T2>T3。

* 0时刻T3到达并开始执行，t0时刻获得共享资源S。
* t1时刻T1到达并抢占T3开始执行，t2时刻T1想要获取S，但是S已经被T3上锁，T1被阻塞，此时提高T3的优先级，因为S的等待队列中优先级最高的为T1，所以T3的优先级被提高到和T1一样，并继续占有S执行
* t3时刻，T2到达，但是T3的优先级已经被提高到T1，大于T2，所以T2被阻塞，T3继续执行。
* t4时刻，T3释放S，唤醒T1，T1获取S执行到t6结束，T2执行。

再回过头去看代码6.7，处理优先级的部分我截出来放在代码6.8

|  |
| --- |
| /\* 处理优先级反转\*/  highPrio = (acoral\_u8)(evt->count >> 8);  thread = (acoral\_thread\_t\*)evt->data;  if (thread->prio>cur->prio)  {  /\*有可能优先级反转，继承最高优先级\*/  if(cur->prio<highPrio)  {  highPrio=cur->prio;  evt->count &= ~MUTEX\_U\_MASK;  evt->count |= highPrio << 8;  }  acoral\_thread\_change\_prio(thread,highPrio);  } |

代码6.8

先解释一下各个变量的含义

* highPrio：互斥量被阻塞线程中最高的优先级
* thread：占用互斥量的线程

根据之前的描述，可以知道，优先级反转可能发生的条件就是，申请线程优先级高于占用线程。因为aCoral是数字越大，优先级越低，所以判断条件就是thread->prio>cur->prio。那有同学可能会问了，为什么不能用count成员的7~0位去和申请线程的优先级比较呢？那是因为，count成员7~0位是不变的，表示占用线程的原始优先级，而占用线程的优先级可能在之前就已经被提升过了，所以不能用于判断。具体的情景请同学自己思考。那判断出可能会发生优先级反转之后，就要提升占用线程的优先级了。具体做法就是：

* 比较申请线程的优先级和highPrio，找到小的（优先级高）那个

if(cur->prio<highPrio)

* 如果申请线程优先级更高，更新highPrio的值，此时highPrio就是被阻塞线程中的最高优先级

highPrio=cur->prio;

* 更新count成员23~16位的值

evt->count |= highPrio << 8;

* 提升占用线程优先级，继承被阻塞线程中的最高优先级

acoral\_thread\_change\_prio(thread,highPrio);

这样，当高优先级线程申请的互斥量已被占用后，占用线程就会继承被阻塞线程中最高的优先级一直执行下去，这样即使中间出现中优先级线程也不会抢占CPU执行权，也就避免了优先级反转的现象。之后申请线程便被阻塞（或者进入超时等待队列在超时时间内等待互斥量释放）。

在互斥量被释放后，占用线程的值将恢复原始优先级，也就是count成员的7~0位。

## 同步机制——信号量

除了互斥量，信号量也可以用于线程之间的互斥。同时信号量也可以用于线程同步。本节就主要介绍信号量的线程同步功能。在此之前，我们先来了解一下什么是信号量。

* 概述

信号量用于实现任务与任务之间、任务与中断处理程序之间的同步与互斥。

按功能划分类型：三种

互斥信号量：任务对临界资源互斥访问；不处理优先级反转；相当于轻量级互斥量，比互斥量耗费更少的CPU资源

同步信号量：用于线程间同步；初始值为0——同步事件未发生

计数信号量：控制系统中临界资源的多个实例使用；初始值为n——要管理的最大实例个数

* aCoral创建信号量

和互斥量一样，aCoral的线程在申请使用信号量之前，也要先创建一个信号量。创建的过程和互斥量基本类似，只有ECB的count和type成员赋值不同。

type成员将被赋值为ACORAL\_EVENT\_SEM，指示这个ECB是一个信号量。count成员将被赋值为（1-semNum），其中semNum为信号量创建函数的输入参数。

* 目的：创建信号量
* 提供：常量ACORAL\_EVENT\_SEM，表示事件为互斥量。

|  |
| --- |
| acoral\_evt\_t \*acoral\_sem\_create(acoral\_u32 semNum)  {  acoral\_evt\_t \*evt;  evt = acoral\_alloc\_evt();  if (NULL == evt)  {  return NULL;  }  semNum = 1 - semNum; /\* 拥有多个资源，0,一个 -1 两个， -2 三个 ....\*/  evt->count = semNum;  evt->type = ACORAL\_EVENT\_SEM;  evt->data = NULL;  acoral\_evt\_init(evt);  return evt;  } |

代码6.9

* aCoral申请信号量

**？？？为什么要1-sem**

## 通信机制——邮箱

* 任务间的通信方式

1. 直接通信: 在通信过程中双方必须明确地知道（命名）对方

send (P，message)：发送一个消息到任务P

receive(Q，message) ：从任务Q接收一个消息

1. 间接通信：通信双方不需要指出消息的来源或去向，而通过中间机制来通信，如邮箱、共享存储、...。

send(A，message) ：发送一个消息给邮箱A

receive(A，message)： 从邮箱A接收一个消息

1. 极端情况：用于任务间同步

# 内核编译、烧写与启动

到这里，aCoral内核中的几大模块都已经介绍完毕。那要将我们手上已经有的aCoral源码，变成mini2440开发板上可以运行的嵌入式操作系统，还需要经过编译、烧写两个步骤，最后才能启动aCoral。本章就将针对编译、烧写、启动三个过程进行讲解。

## 内核编译

内核编写完成之后的第一步就是编译。编译本质上就是将我们编写的文本代码转变成arm处理器能够执行的二进制编码。对于aCoral这类嵌入式操作系统，我们通常在linux环境下，使用交叉编译工具对源码进行编译。交叉编译工具，例如arm-linux-gcc，本质上和我们之前接触过的gcc.exe没有区别，无非前者使用在PC环境下，编译出来的对象在目标板上运行，所以称为交叉编译；而后者无论编译还是运行都在PC机中。又因为我们需要编译的是一整个操作系统，涉及到的文件非常之多，如果还是像之前编译单个c文件时，在命令行一条一条编译命令敲写，就会十分麻烦。所以我们会用到Makefile这个东西，一次编译所涉及到的所有命令都提前在makefile中写好了，我们编译时只需要执行make命令就行了。而为了学习Makefile，我们就需要对aCoral整个源码的目录结构有所了解。

### 目录结构

aCoral的一级目录结构如图7.1所示

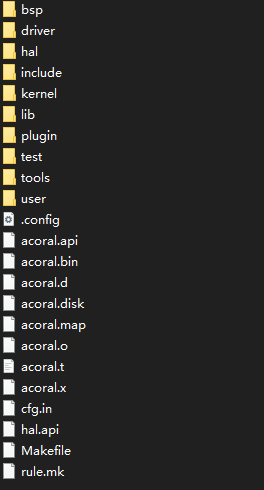


图7.1

其中一些重要的文件或者目录的作用如下：

* **driver：**不同开发板的外设驱动程序，例如UART串口、网卡、显示屏等
* **hal：**硬件抽象层（Hardware Abstraction Layer），将不同开发板的硬件资源访问封装成统一的接口，供上层操作系统使用，这样就可以将底层与硬件相关的代码和上层操作系统相关的代码分开，便于在不同硬件资源的开发板之间移植。源码中我们看到的HAL开头的函数，多数来自于这个文件夹。我们第一章中，aCoral启动入口部分提到的start.S文件就来自于\hal\s3c2440\src这个目录中。
* **include：**主要就是几个与系统相关的头文件定义，比如aCoral中的各种数据类型定义（acoral\_u8等），处理器型号（S3C2440），还有aCoral某个功能是否启用等。这个文件夹中的autocfg.h头文件是在make menuconfig之后自动生成的，与根目录的.config配置文件中的内容对应。
* **kernel：**整个aCoral的内核源码最核心的部分，也就是刚刚提到的所谓上层操作系统部分，我们之前第二到第六章看到的几个模块的源码也基本上来自于这个文件夹。第一章aCoral启动入口部分提到的core.c就来自于\kernel\src这个目录中。
* **lib：**定义了上层操作系统中常用的一些数据结构，如list、queue、bitmap等
* **test：**操作系统启动之后，对系统的一些测试函数
* **.config：**内核编译配置，每次执行make命令，都会读取这个文件中的一些配置，包括使用的处理器型号等。每次执行make menuconfig命令调整系统配置时都会将保存的配置写入这个文件。
* **Makefile：**编译命令集合

### Makefile

Makefile就是我们每次在终端中敲入make命令之后执行的文件。Makefile其中就是一连串命令的集合，它会读取.config文件中的配置，编译出我们需要的aCoral内核镜像。我么首先来学习一下Makefile的基本语法

想要书写一个完整的 Makefile文件，需要了解 Makefile 的相关的书写规则。我们已经知道了 Makefile 描述的是文件编译的相关规则，它的规则主要是两个部分组成，分别是依赖的关系和执行的命令，其结构如下所示：

|  |
| --- |
| targets : prerequisites  command |

代码7.1

相关说明如下：

* targets：规则的目标，可以是 Object File（一般称它为中间文件），也可以是可执行文件，还可以是一个标签；
* prerequisites：是我们的依赖文件，要生成 targets 需要的文件或者是目标。可以是多个，也可以是没有；
* command：make 需要执行的命令（任意的 shell 命令）。可以有多条命令，每一条命令占一行。

注意：我们的目标和依赖文件之间要使用冒号分隔开，命令的开始一定要使用Tab键。

通过下面的例子来具体使用一下 Makefile 的规则，Makefile文件中添代码如下：

|  |
| --- |
| test:test.c  gcc -o test test.c |

代码7.2

在默认的方式下，也就是我们只输入make命令。那么，make会在当前目录下找名字叫“Makefile”或“makefile”的文件。

如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“test”这个文件，并把这个文件作为最终的目标文件。

如果test文件不存在，或是test所依赖的后面的.c文件的文件修改时间要比test这个文件新，那么，他就会执行后面所定义的命令来生成test这个文件，需要执行的命令是gcc -o test test.c。这就是 Makefile 的基本的语法规则的使用。

那现在可以来看一下aCoral的Makefile文件了

待补充

### 编译操作

在解读完了aCoral的Makefile之后，我们就要来进行真正的编译操作了。编译环境为Ubuntu 18.04.5 LTS，编译交叉工具链为arm-2010q1。

* 第一步，将编译器加入系统环境变量。在终端输入“gedit /etc/profile”，在打开的文件最后输入“PATH=/home/acoral/Desktop/arm-2010q1/bin:$PATH”，其中“/home/acoral/Desktop/”为交叉编译链文件所在路径。然后保存文件，在终端输入“source etc/profile”，让“/etc/profile“文件立即生效。再在终端输入“arm-none-eabi-gcc -v”，查看版本号，确定交叉编译工具是否安装成功。
* 第二步，修改编译所需的编译器。进入aCoral顶层目录，修改makefile文件中的交叉编译路径，如图7.2所示



图7.2

其中“CROSS\_COMPILE=”为自己的交叉编译工具路径。

* 第三步，make menuconfig。在终端输入“make menuconfig”，配置完成之后保存，可以得到一个.config文件（同时会在include文件夹内自动生成一个对应的autocfg.h头文件）。这里我们为了节约时间，已经提供了一个.config文件，所以这步可以暂时跳过。
* 第四步，make。在终端输入“make“，等待一会儿之后，就会得到我们要烧写的aCoral镜像文件acoral.bin以及一些其他能够帮助我们了解aCoral的文件。

## aCoral内核烧写

在得到acoral.bin镜像文件后，我们就可以将其烧写到开发板上了。理论上我们可以直接把acoral.bin烧写到开发板的nor flash或者nand flash上启动，aCoral可以自我引导，即将自己复制到sdram内存中执行。

但是这有一个问题，烧写到nor flash或者nand flash的速度是很慢的，如果我们每次在修改aCoral的源码后，直接烧写到开发上进行调试，就会浪费大量时间在等待烧写完成上。对于这个问题，我们有一种解决方案，就是在nor flash或者nand flash中烧写一个bootloader，每次修改源码得到新镜像后，使用bootloader，将新编译得到的内核镜像直接复制到sdram中执行，这样就可以节省很多时间。这个bootloader可以是uboot，也可以是supervivi等等。我们已经在开发板的norflash中提前烧写好了supervivi，接下来就可以正式开始烧写。注意，接下来的步骤，我是在windows中进行的，ubuntu中的过程大致类似，只不过使用的工具可能有所不同，具体参考《嵌入式实时操作系统的设计与开发》一书。

第一步，将开发板左下角的开关向下拨到nor，等会儿就可以从norflash启动supervivi，拨动开关，示意图如图7.3所示

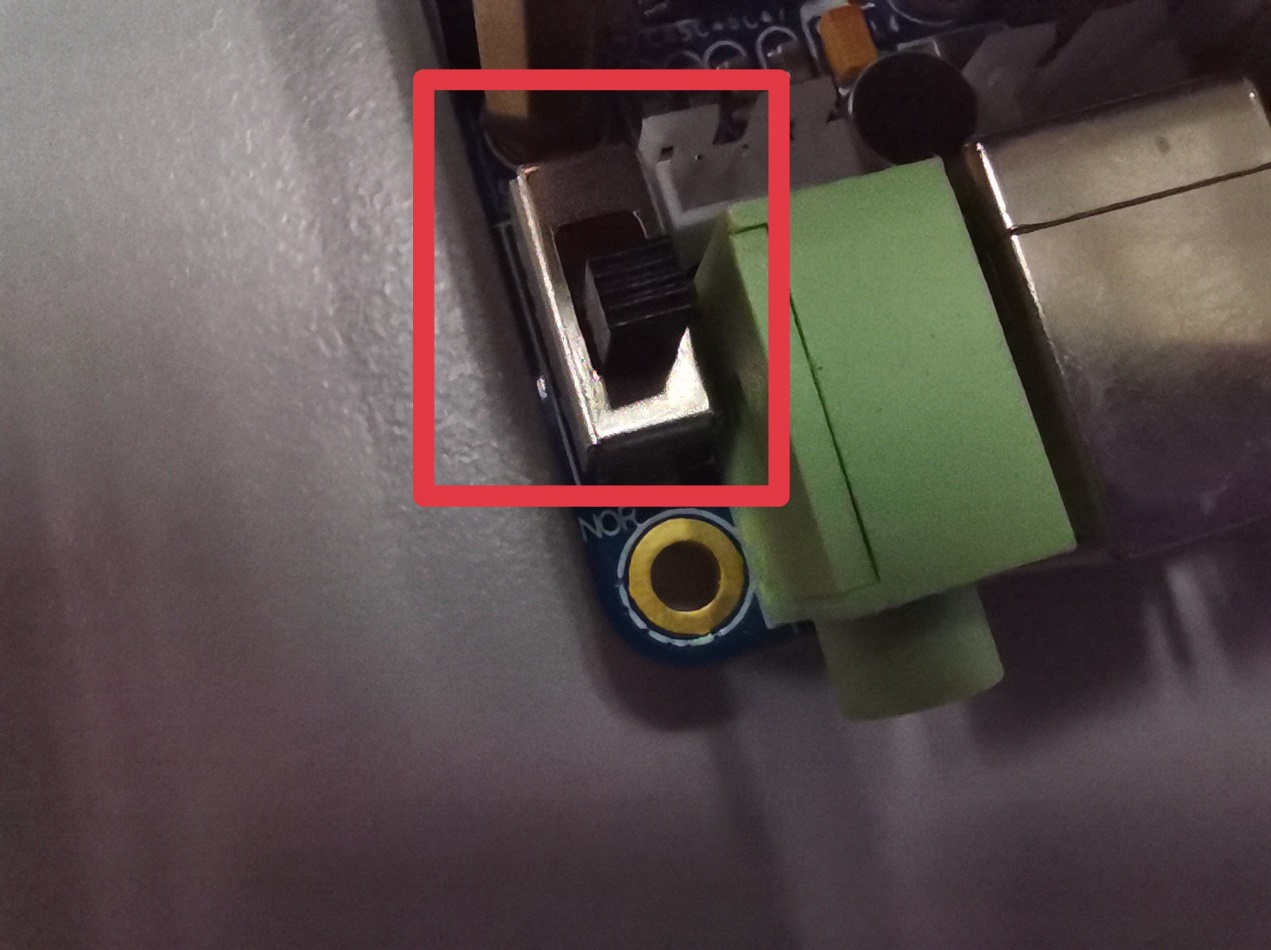


图7.3

第二步，接线。依次从左到右插上USB下载线、串口线、电源线，如图7.4所示

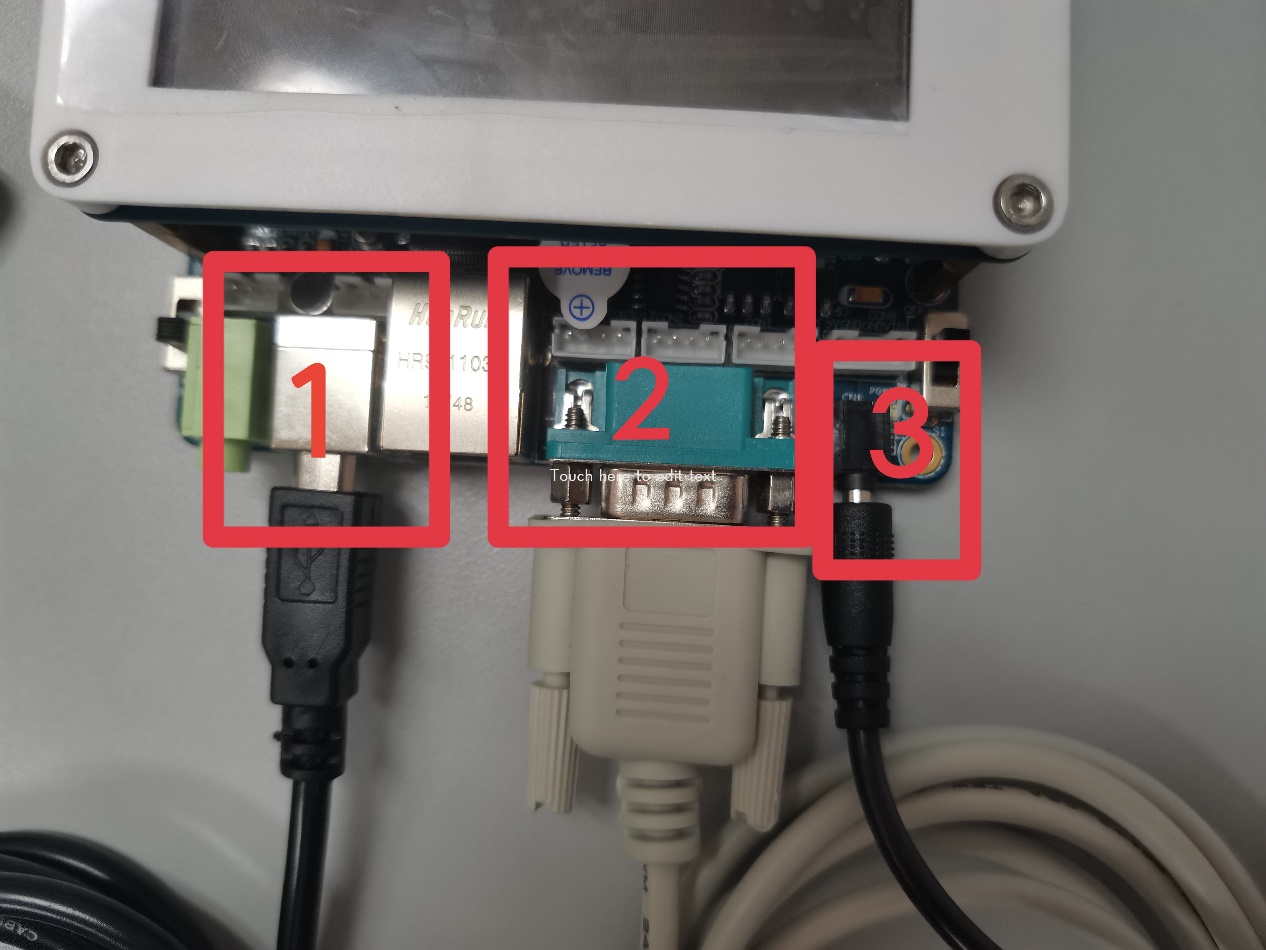


图7.4

另一端USB接口自然是插到电脑上。其中USB下载线用于将镜像下载到sdram中，串口线用于在开发板和PC之间传递字符信息，比如命令行命令就是由串口线传到开发板上，执行结果再由串口回显到PC上。大家电脑上肯定没有串口了，所以需要一个USB转串口，USB转串口的驱动在课程文件夹中。驱动的安装需要关闭win10强制驱动签名，具体自查。

第三步，打开课程文件夹中的SuperVivi-Transfer-Tool-Complete\Program\SuperVivi-USB-Transfer-Tool.exe，并将右上角端口号选择为插入串口的那个端口号（具体从设备管理器看），如图7.5.

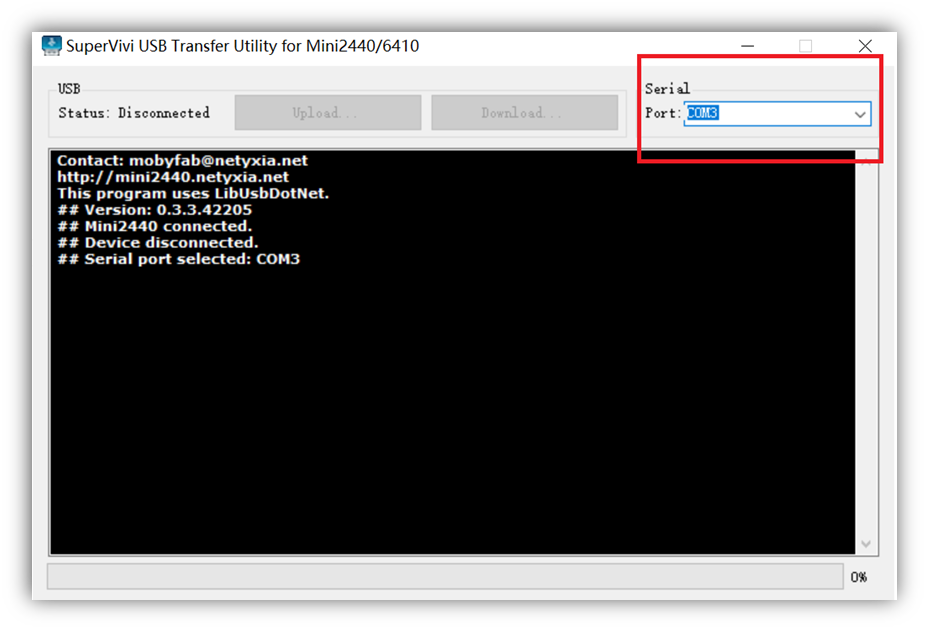


图7.5

第四步，打开开关，之后如果一切顺利，可以看到如图7.6所示的界面。

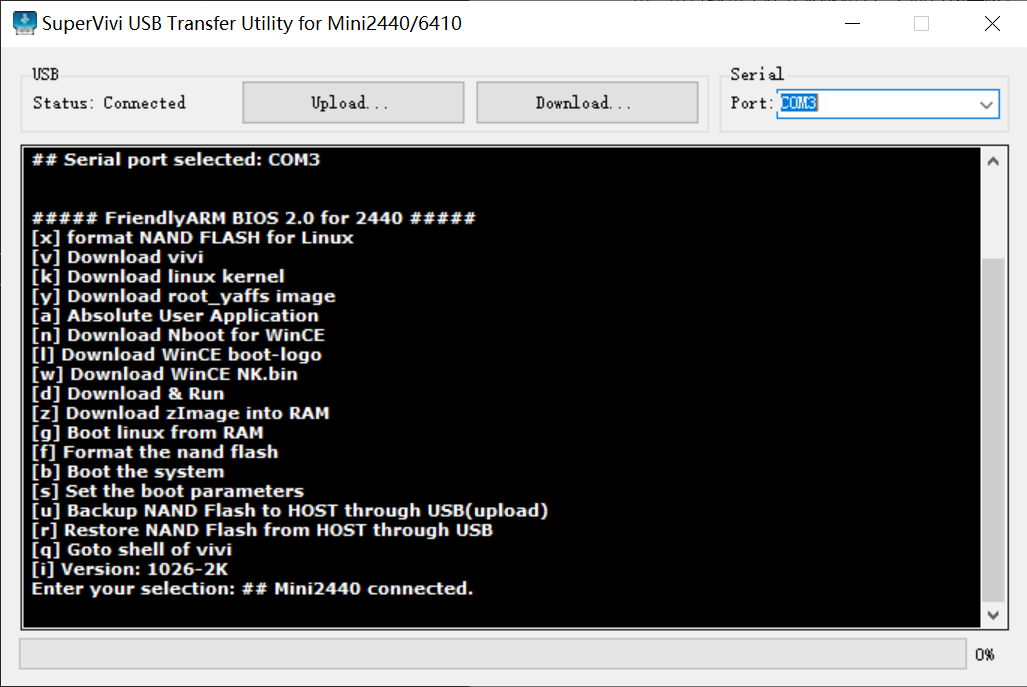


图7.6

第五步，在终端中输入d，并点击Upload，选择之前编译出来的acoral.bin文件（之前是在Ubuntu虚拟机中编译的，这里需要自己想办法拿到windows中）。等待进度条一闪而过，烧写完成，终端上显示出“aCoral:>“，就大功告成了。如图7.7所示

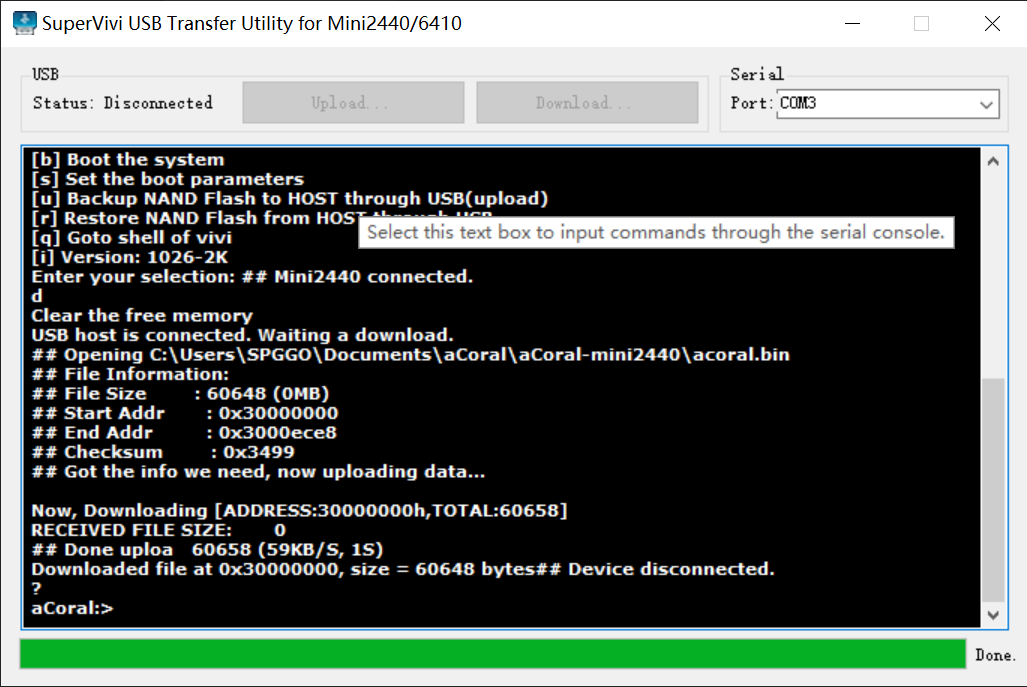


图7.7

大家可以在终端中输入help命令，看看aCoral现在支持的指令（其实没几个）。至此，aCoral的烧写完成，之后如果又重新编译了新的镜像，都可以按照这个步骤来。需要注意的是，这种使用bootloader直接往sdram烧写的镜像，关机之后没了，需要重新烧写。

## aCoral内核启动

现在我们已经完成了aCoral的第一次启动。之前几章的学习中，我们也已经学习了aCoral的运行机制和底层代码，那aCoral在上电后究竟是怎么运行起来的呢？运行的第一行代码在哪里？只有知道了这些，aCoral对于我们来说才能彻底从黑盒变成白盒。

### 内存分布

这一小节标题为内存分布，指的是aCoral镜像在sdram中的结构与分布，又或者说编译得到的acoral.bin的组织结构，这二者是等价的，因为后者是要被一摸一样复制到内存中运行的嘛。那我们要先了解两个概念：运行时地址和存储地址。

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。但是这些非易失性存储器往往不能运行程序（可读不可写），所以我们需要把这些程序复制到sdram（可读可写），复制完之后程序继续运行。

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，不在这个地址可能会，也可能不会发生错误。这只是表示程序人员对代码运行时的一种期待和要求，并不是说，程序在运行的时候一定在运行时地址，只是最好在而已。aCoral在mini2440开发板的运行时地址为0x30000000开始的一连串地址，所以当我们把nand flash中的acoral系统整个复制到sdram中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。

将一个操作系统移植到一块开发板上，我们就要根据开发板上sdram的不同来指定运行时地址，mini2440就是0x30000000。那我们在哪里指定这个地址呢？这就是我们之前在Makefile中为链接器ld指定的链接脚本做的事了。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。那我们就来看一下\hal\s3c2440\acoral.lds链接脚本吧。

|  |
| --- |
| ENTRY(\_\_ENTRY)  MEMORY  {  ram (wx) : org = 0x030000000, len = 64M  }  SECTIONS  {  .text :  {  text\_start = .;  \* (.text)  \* (.init.text)  \* (.rodata\*)  }>ram    .data ALIGN(4):  {  \*(.acoral1.call)  \*(.acoral2.call)  \*(.acoral3.call)  \*(.acoral4.call)  \*(.acoral5.call)  \*(.acoral6.call)  \*(.acoral7.call)  \*(.acoral8.call)  \*(.acoral9.call)  \*(.acoral10.call)  \*(.data)  \*(.data.rel)  \*(.got)  \*(.got.plt)  } >ram    .bss ALIGN(4):  {  bss\_start = .;  \* (.bss)  . = ALIGN(4) ;  } >ram  bss\_end = .;  stack\_base = 0x33ffff00;  MMU\_base = 0x33f00000;  SYS\_stack\_size = 0x200;  SVC\_stack\_size = 0x200;  Undef\_stack\_size = 0x100;  Abort\_stack\_size = 0x100;  IRQ\_stack\_size = 0x200;  FIQ\_stack\_size = 0x0;  FIQ\_stack = stack\_base;  IRQ\_stack = FIQ\_stack - FIQ\_stack\_size;  ABT\_stack = IRQ\_stack - IRQ\_stack\_size;  UDF\_stack = ABT\_stack - Abort\_stack\_size;  SVC\_stack = UDF\_stack - Undef\_stack\_size;  SYS\_stack = SVC\_stack - SVC\_stack\_size;  heap\_start = (bss\_end + 3)&( ~ 3);  heap\_end = MMU\_base - 0x1000;  } |

代码7.3

ENTRY关键字定义了整个源码的入口就是标号\_ENTRY。这个标号被定义在hal\s3c2440\src\start.S中，所以start.S中\_ENTRY标号处的“b ResetHandler”就是aCoral上电后执行的第一行代码。这点1.3中就讲到过了。

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。我们也可以自定义段。.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，其实就是第三章中，伙伴系统初始化（代码3.1）的两个输入参数，也就是说，这里的整个heap就是伙伴系统管理的内存空间，大小大概是64M不到一点。

关于aCoral的整个内存分布的具体细节，同学们可以自行查看源码根目录下的acoral.map和acoral.d文件。

### aCoral内核启动第一阶段——start.s

了解aCoral的内存分布，最直接的目的当然还是找到整个系统的入口，也就是start.S中\_ENTRY标号处的“b ResetHandler”指令。1.3中大概说明了ResetHandler是做了一些硬件初始化的操作，最后跳转到第二阶段——acoral\_start执行。但是具体ResetHandler做了什么呢？看代码7.4

|  |
| --- |
| ResetHandler:  **@关闭看门狗**  mov r1, #0x53000000  mov r2, #0x0  str r2, [r1]  **@ 关中断**  mov r1, #INT\_CTL\_BASE  mov r2, #0xffffffff  str r2, [r1, #oINTMSK]  ldr r2, =0x7ff  str r2, [r1, #oINTSUBMSK]  @ **初始化系统时钟**  mov r1, #CLK\_CTL\_BASE  mvn r2, #0xff000000  str r2, [r1, #oLOCKTIME]  mov r1, #CLK\_CTL\_BASE  mov r2, #M\_DIVN  str r2, [r1, #oCLKDIVN]  mrc p15, 0, r1, c1, c0, 0 @ read ctrl register  orr r1, r1, #0xc0000000 @ Asynchronous  mcr p15, 0, r1, c1, c0, 0 @ write ctrl register  mov r1, #CLK\_CTL\_BASE  ldr r2, =vMPLLCON @ clock user set  str r2, [r1, #oMPLLCON]    @ **初始化内存和栈**  bl memsetup  bl InitStacks  @ **复制自己到sdram，0x30000000运行时地址处**  adr r0,\_\_ENTRY  ldr r1,\_text\_start  cmp r0,r1  blne copy\_self  @ **清空bss段**  ldr r0,\_bss\_start  ldr r1,\_bss\_end  bl mem\_clear  **@ 跳转到内核启动第二阶段**  **ldr pc,=acoral\_start** |

代码7.4

* 第一步，关闭看门狗。

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（如下图所示），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。关闭的方法就是往0x53000000地址的WTCON寄存器写0。WTCON寄存器说明如图7.8

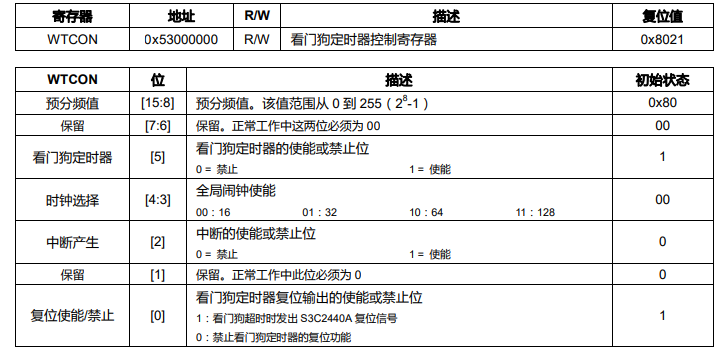


图7.8

* 第二步，关中断

因为这个时候，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。具体自己对照s3c2440中文手册。

* 第三步，初始化系统时钟

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。上面的初始化完成后，三者的比例为FCLK：HCLK：PCLK=8：2：1，其中FCLK≈400MHz，正好就是第一章中介绍的S3C2440处理器的主频。

UPLL的输出频率没有配置，直接给USB使用。

具体初始化涉及到的寄存器和数值，参考s3c2440中文手册。

* 第四步，初始化内存和栈

memsetup就是初始化一下mini2440的存储器（BANK0~BANK7），具体自行阅读。

InitStacks就是初始化一下arm处理器各个模式的栈，具体的做法就是修改cpsr寄存器，进入每一个模式，因为影子寄存器的缘故，每个模式的sp寄存器都是独立的（除了系统和用户模式），所以在每个模式中修改sp寄存器的值就行了。修改的值，也就是每个模式的栈的地址，来自于链接脚本。

* 第五步，复制自己到sdram，0x30000000运行时地址处

这一步非常重要。之前说过运行时地址的概念，也就是aCoral最好应该运行的位置。那如果aCoral是烧写在nor flash这类存储器中，启动时是从0地址开始运行aCoral的，那aCoral怎么到它的运行时地址来运行呢？这项任务是由aCoral自己来完成的，也就是我搬起了我自己。可以看到copy\_self的执行条件**？？？**

* 第六步，清空bss段

bss段中的数据都是未初始化或者初始值为0的，所以清空bss段就是将其中的内存全部赋值为0。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。

* 最后，跳转到内核启动第二阶段

PC寄存器被赋值为acoral\_start函数的地址，这个函数定义在\kernel\src\core.c中，作为硬件初始化完成后，aCoral启动的第二阶段。

### aCoral内核启动第二阶段——core.c

硬件初始化完成后，就需要对系统的软件部分进行初始化，其中就包括本课程的主线——acoral\_module\_init()函数。我们之前几章的几个模块的初始化基本都是在这个函数中完成的。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。

|  |
| --- |
| void acoral\_start(){  orig\_thread.console\_id=ACORAL\_DEV\_ERR\_ID;  acoral\_set\_orig\_thread(&orig\_thread);  /\*内核模块初始化\*/  acoral\_module\_init();  /\*串口终端应该初始化好了，将根线程的终端id设置为串口终端\*/  #ifdef CFG\_DRIVER  orig\_thread.console\_id=acoral\_dev\_open("console");;  #endif  /\*主cpu开始函数\*/  acoral\_core\_cpu\_start();  } |

代码7.5

可以看到acoral\_start代码并不多，概括一下就是三件事：内核模块初始化、设置orig线程、调用acoral\_core\_cpu\_start()进行下一步初始化工作。

第一件事，内核模块初始化。这个不用多说，前几章都在围绕这个讲。

第一件事，为什么要设置一个orig线程呢？这个线程是怎么创建的呢？第二个问题比较好回答，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，所以这里就直接定义了一个TCB，给里面的console\_id成员赋了个值，并将这个orig线程设置为当前正在运行的线程，如代码7.6.

|  |
| --- |
| void acoral\_set\_orig\_thread(acoral\_thread\_t \*thread){  running\_thread[acoral\_current\_cpu]=thread;  } |

代码7.6

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。它被赋值为“acoral\_dev\_open("console")”，也就是说这个orig线程的console控制台是一个叫“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的UART串口驱动，并返回一个它的res\_id，以便之后使用UART。话又说回来，那orig线程需要这个console干嘛呢？orig线程又不打印或接收信息，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，如图7.9



图7.9

第三件事，就是调用acoral\_core\_cpu\_start()进行下一步初始化工作，代码7.7

|  |
| --- |
| void acoral\_core\_cpu\_start(){      acoral\_comm\_policy\_data\_t data;      /\*创建空闲线程\*/      acoral\_start\_sched=false;      data.cpu=acoral\_current\_cpu;      data.prio=ACORAL\_IDLE\_PRIO;  data.prio\_type=ACORAL\_ABSOLUTE\_PRIO;   idle\_id=acoral\_create\_thread\_ext(idle,IDLE\_STACK\_SIZE,NULL,"idle",NULL,ACORAL\_SCHED\_POLICY\_COMM,&data);      if(idle\_id==-1)          while(1);      /\*创建初始化线程,这个调用层次比较多，需要多谢堆栈\*/      data.prio=ACORAL\_INIT\_PRIO;      /\*动态堆栈\*/      init\_id=acoral\_create\_thread\_ext(init,ACORAL\_TEST\_STACK\_SIZE,"in init","init",NULL,ACORAL\_SCHED\_POLICY\_COMM,&data);      if(init\_id==-1)          while(1);      acoral\_start\_os();  } |

代码7.7

这里，acoral\_core\_cpu\_start()概况起来也是做了三件事，创建idle线程、创建init线程、调用acoral\_start\_os()正式开始运行系统。

第一件事和第二件事中的两个线程，就是之前提到的，orig线程的亲儿子，他们继承了orig线程的console，所以它们，尤其是init线程，才能正确使用UART串口实现和我们的交互。

关于这两个线程，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；init线程做了很多初始化工作，包括5.2提到的时钟中断的注册，就是在init线程中完成的。init线程还初始化了shell。shell又是什么呢？你可以理解为，串口工具中那个黑黑的界面，能接收我们的指令并返回结果的程序就是shell。shell运行于开发板中，并通过console（UART串口）与我们交互。

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会怎么运行，就看我们给它什么指令了。

### aCoral用户交互

上文提到的shell，界面就如图7.10

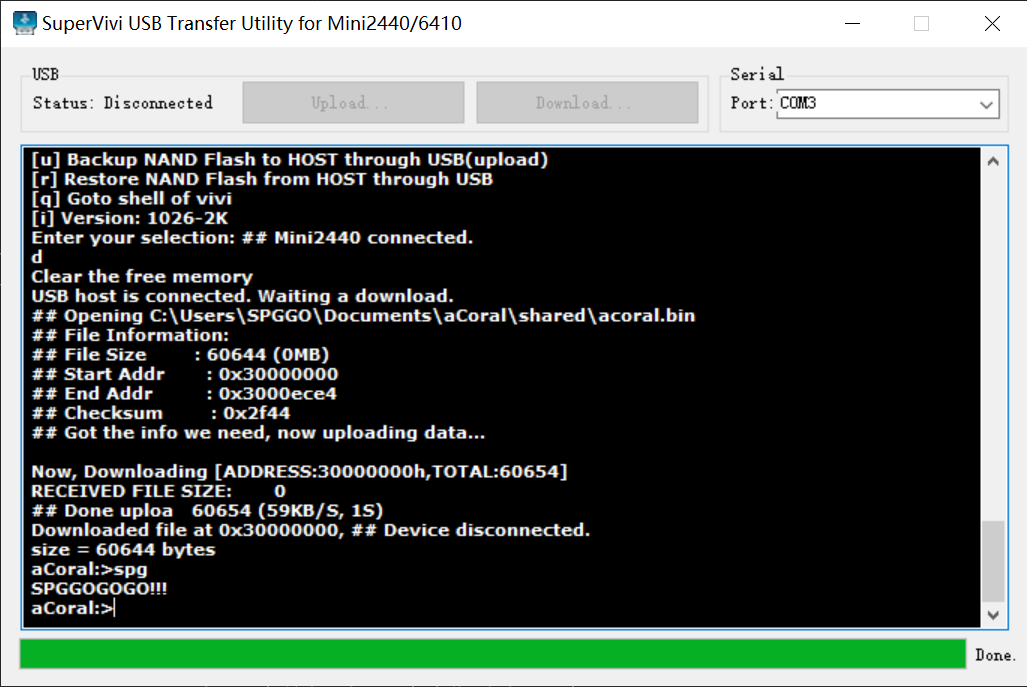


图7.10

这就是aCoral复制到内存后，成功运行的界面，或者说shell就长这样。那shell如何读懂我们的指令呢？比如图7.10中，我输入了一条指令“spg”，aCoral是如何知道要回应我“SPGGOGOGO”的呢？我们如何知道系统中有哪些可用指令呢？

首先，我们可以输入“help”指令，来查看系统中所有可用指令。而这些指令定义在\user\src\cmd.c中。这部分内容比较简单，同学们自行学习。

# 性能评估