第10章

# Cortex-M3的低层编程

* 概览
* 汇编与C的接口
* 典型的开发流程
* 第一步工作
* 与外界互动
* 使用数据存储器
* 使用互斥访问实现互斥锁操作
* 使用位带实现互斥锁操作
* 使用位段提取与查表跳转

## 10.1 概览

在CM3上编程，既可以使用C也可以使用汇编。可能还有其它语言的编译器，但是大多数人还是会在C与汇编的世界里游弋。C与汇编都“尺有所短，寸有所长”，不能互相取代。使用C能开发大型程序，而汇编则用于执行特种任务。

在使用不同的工具链和芯片时，有大量的用法和用量都随之不同。因此，本书不会深入讲解怎样精通一个具体的工具链，也不会大谈如何把程序烧到板子上。在第19章和第20章会提到一些入门知识，具体内容还需查阅相关的文献和在线帮助文档。

### 10.1.1 使用汇编

如果工程比较小，使用纯汇编常常是可行的，而且能使我们随心所欲地优化和控制程序。不过，这么一来的开发周期会变长。尤其是当工程变大，需要处理比较复杂的数据结构，以及要管理函数库时，汇编那狰狞的真面目就会渐显出来：各种地址和间接引用千头万绪；bug劈头盖脸；甚至好几天都改不完，工作量激增，简直就是自虐。当然，如果你想成为系统开发的大虾，就必须以“我不下地狱谁下地狱”的决心，去勇敢面对，后天下乐而乐，百炼成钢。

不论如何，时间宝贵。我们应该以C来实现程序的大框架，而本着好钢用在刀刃上的原则来使用汇编，因为只有在不多的特殊场合是适合使用汇编，甚至是非使用汇编语言不可的，它们包括：

* + 无法用C写成的函数，如操作特殊功能寄存器，以及实施互斥访问。
  + 在危急关头执行处理的子程（如，NMI服务例程）。
  + 存储器极度受限，只有使用汇编才可能把程序或数据挤进去。
  + 执行频率非常高的子程，如操作系统的调度程序。
  + 与处理器体系结构相关的子程，如上下文切换。
  + 对性能要求极高的应用，如防空炮的火控系统。

### 10.1.2 使用C

用C写的程序可以移植，并且操作复杂数据结构时远远比汇编方便。但因为C是一种通用语言——至少是低等高级语言，它并不指定如何初始化具体的处理器（用于在main执行前准备好执行环境）。在解决这个问题时，不同的工具链都有自己的一套，因此最聪明的办法就是看一看工具链附带的示例程序。如果使用RealView开发套件（RVDS）或者KEIL 的RealView微控制器开发套件(RVMDK)，则编译器和汇编器是ARM提供的，而且它们中都附带了很多示例。如果使用了GNU的工具链，则第19章以CodeSourcery GNU工具链为例，给出一个简单的示例（其它示例可以去网上找）。

尽管在使用了C后，大大加速了开发，但是底层的系统控制往往还需要汇编代码。很多编译器都允许我们直接在C代码中插汇编，称为“内联汇编”；另外还允许我们写独立的汇编模块，与编译后的C模块一起连接。以往，使用内联汇编的作法比较多，但是在ARM编译器中，不支持对Thumb-2指令的内联汇编。取而代之的，是从RealView C编译器的3.0版开始，新增了所谓“嵌入式汇编”的功能，它支持Thumb-2指令。它让我们可以在C程序中插入使用汇编语言编写的函数，例如：

**\_\_asm** void SetFaultMask(unsigned int new\_value)

{

//在这里使用汇编代码实现本函数

MSR FAULTMASK, new\_value // 把new\_value写入FAULTMASK中

BX LR // 返回主程序（不可省略）

}

RealView C编译器对嵌入式汇编的详细论述，在《RVCT 3.0 Compiler and Library Guide(Ref6)》中给出。

在CM3中，嵌入式汇编还是比较需要的，因为常常会有访问特殊功能寄存器的时候。比如，在设置堆栈时，就要使用MRS/MSR指令。对于其它不能由编译器产生的指令，比如WFI/WFE、互斥访问、存储器隔离等指令，也必须用汇编显式给出。

在以前的ARM处理器中，因为支持ARM/Thumb双重状态，往往需要所谓的“interworking”，且不同的源文件可能需要编译成不同状态下的代码。在CM3中不再有此需求，因为只使用了Thumb状态，从而工程管理清爽多了。

当使用C开发程序时，推荐开启CM3的双字对齐管理机制（在NVIC配置与控制寄存器中，把STKALIGN置位），代码形如：

#define NVIC\_CCR ((volatile unsigned long \*)(0xE000ED14))

\*NVIC\_CCR ＝ \*NVIC\_CCR | 0x200; //设置STKALIGN位

这是用于确保系统能严格遵守AAPCS过程调用标准，个中细节请参阅第12章。

## 10.2 汇编与C的接口

在很多情况下，都需要让C程序模块与汇编程序模块互相交互，它们包括：

* + 在C代码中使用了嵌入式汇编（或者是在GNU工具下，使用了内联汇编）
  + C程序呼叫了汇编程序，这些汇编程序是在独立的汇编源文件中实现的
  + 汇编程序调用了C程序

在这些情况下，必须知晓参数是如何传递的，以及值是如何返回的，才能在主调函数与子程序之间协同工作。这些交互的机制在ARM中有明确的规定，由文档《ARM Architecture Procedure Call Standard(AAPCS, Ref5)》给出。

不过，在大多数场合下的情况都比较简单：当主调函数需要传递参数（实参）时，它们使用R0-R3。其中R0传递第一个，R1传递第2个……在返回时，把返回值写到R0中。在子程序中，可以随心所欲地使用R0-R3，以及R12（回顾第9章，想想为什么会PUSH它们）。但若使用R4-R11，则必须在使用之前先PUSH它们，使用后POP回来。

可见，汇编程序使用R0-R3, R12时会很舒服。但是如果换个立场——汇编要呼叫C函数，则考虑问题的方式就有所不同：必须意识到子程序可以随心所欲地改写R0-R3, R12，却决不会改变R4-R11。因此，如果在调用后还需要使用R0-R3,R12，则在调用之前，必须先PUSH，从C函数返回后再POP它们，对R4-R11则不用操心。在本章的示例程序中，绝大多数只是调用汇编子程序，它们只影响少量寄存器，或者会在返回前恢复寄存器的内容，所以往往没有严格遵守AAPCS。这主要是为了突出其它重点，简化程序，请读者不要钻牛角尖。

## 10.3 典型的开发流程

在开发基于CM3的应用程序时，常常有多种源程序和库，有些是自己写的，有些是别人已经写好的（尤其是底层的软件）。上述这些开发工具生成代码的流程都差不离。对于最基本的应用，也至少需要C编译器，连接器以及二进制文件处理工具。如果使用的是ARM的工具，如RVDS或RealView 编译器工具（RVCT），则它们的流程如图10.1所示。其中的“分散加载脚本”是可选的，但是当存储器映射变得比较复杂时，则需要它。

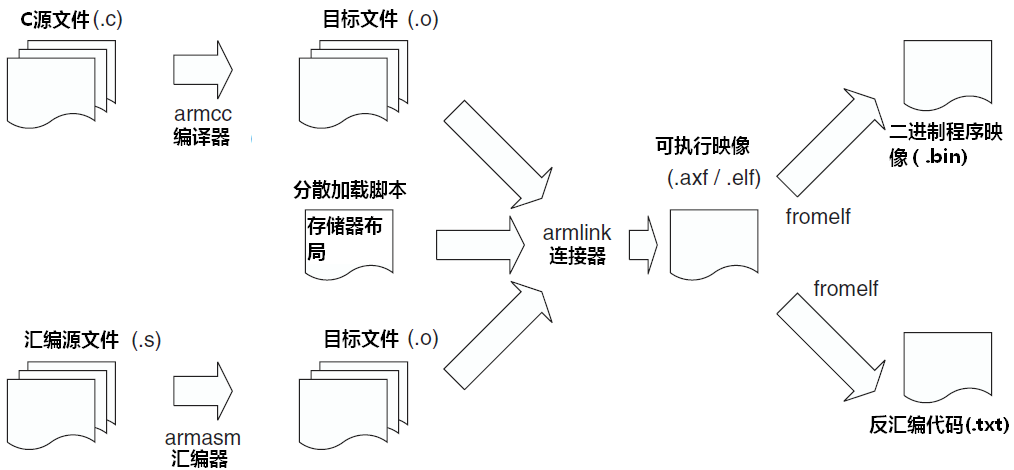


图10.1 使用ARM工具链时的典型开发流程

在上述基本工具之外，RVDS还提供了大量的其它实用程序，比如一个集成开发环境(IDE)以及调试器。欲知详情，可登录ARM网站（[www.arm.com](http://www.arm.com)）。

## 10.4 第一步工作

本章为提供了若干汇编写的例子，在实际应用中，这些程序都会用C写。但是以汇编的方式呈现，有助于让读者更深更好地理解CM3的工作内幕，以便在以后用C开发时，已经是过来人，心里更有底。这里给的程序都用ARM的汇编器(armasm)来汇编，其它工具可能对语法格式有些不同的要求。而且实际上，开发工具几乎都会把启动工作做好，让我们根本不用去想还有启动代码的事（不过，这也妨碍了我们学习得更深入）。下面，就隆重请出本书第一个完整的示例程序（请参考向量表来阅读）：

STACK\_TOP EQU 0x20002000 ; SP初始值，常数

AREA |Header Code|, CODE

DCD STACK\_TOP ; 栈顶（MSP的）

DCD Start ; 复位向量

ENTRY ; 指示程序从这里开始执行

Start ; 主程序开始

; 初始化寄存器

MOV r0, #10 ; 加载循环变量的初值

MOV r1, #0 ; 初始化运算结果的值

; 计算 10+9+8+...+1

loop

ADD r1, r0 ; R1 += R0

SUBS r0, #1 ; R0自减，并且根据结果更新标志（有”S”后缀）

BNE loop ; if (R0!=0) goto loop;

; 现在，运算结果在R1中

deadloop

B deadloop ; 工作完成后，进入无穷循环

END ; 标记文件结束

这个例子非常简单，它只初始化了SP以及PC，以及初始化了需要使用的寄存器，然后就执行连加循环中。

使用ARM工具来汇编该程序，命令为：

$> armasm --cpu cortex-m3 -o test1.o test1.s

命令行中的“-o”指示后面的是输出文件名——也就是test1.o，它也就是目标文件。接下来，我们就要使用连接器，连接各目标文件（本例中只有一个）并创建出一个可执行的映像（ELF），命令为：

$> armlink --rw\_base 0x20000000 --ro\_base 0x0 --map -o test1.elf test1.o

这里，“--ro\_base 0x0”的意思是说，把只读区（也就是程序ROM）的起始地址设为0；而“--rw\_base 0x20000000”则指定读写区（数据存储器）从0x20000000开始（在本例中，我们没有定义任何RAM数据）。“--map”选项则要求连接器给出存储器分配映射表，通过它，可以查看编译后的映像中内存的布局。

最后，我们要生成二进制烧写文件，命令行为：

$> fromelf --bin --output test1.bin test1.elf

如果想要看看生成的映像是否确实是我们想要的，还可以像这样对它做反汇编：

$> fromelf -c --output test1.list test1.elf

（其实基本上很少会做上步——译者注）

如果一切都好，就可以把ELF映像或者二进制代码烧写到器件中了，也可使用模拟器来测试。

## 10.5 与外界互动

如果能把自己的单片机与外面的世界联系起来，那该是多么令人兴奋和值得期待呀！我们常常从点亮LED开始，仿佛是前进路上的明灯，尽管它提供的信息非常有限，但闪烁的灯光常给人“它活着”的印象。如果要输出更多的信息，则最容易上手的方式就是往一个终端发送文本。在嵌入式产品开发中，通常是把一个UART接到电脑上来实现的。例如，运行Windows的电脑大多会有一个附送的“超级终端”程序，通过它可以很方便地让电脑扮演字符终端的角色。

CM3内核没有包含UART接口，但基于CM3的单片机都会有的，而且基本还不止一个。不同芯片的UART用法不同，但是本书就不讲具体的UART驱动了，扯得太远也跑题。在下一个例子中，我们假设系统中有一个UART，UART中有一个“状态位”，用于指示输出缓冲是否已经准备好接收新数据。另外，还需要一个电平转换器件（如MAX3232），用于把单片机I/O口使用的电平转换成RS-232使用的电平。其实，UART并不是输出文本的唯一选择，在CM3中有很多调试组件，它们提供了一系列输出调试消息的方法：

* + 半主机（Semihosting）：取决于调试器与代码库的支持，可以通过NVIC的调试寄存器来做Semihosting（通过调试探测设备，以printf的形式输出消息），第15章还要深入讨论这个主题。使用时，你要在C程序中使用printf函数，然后其输出就会显示在终端，或者显示在调试软件标准输出（STDOUT）上，具体细节还是请参阅第15章。
  + 硬件水平上支持的跟踪：如果使用的CM3单片机提供了一个跟踪接口，并且有一台外部的跟踪接口分析仪（TPA）的话，则可以解放出UART，而使用ITM来做形如printf的调试。跟踪端口就是为了这种调试而生的，它可比UART专业多了——速度快而且能提供多条信道。
  + 硬件水平上支持的跟踪——通过串行线查看器：作为后备方案，CM3的TPIU还提供了“串行线查看器（SWV）”操作模式。有了它，就可以使用远比TPA便宜的硬件来捕获从ITM发来的消息。不过，在SWV模式下，带宽并不富余，因此在需要输出大量数据时，本法就显得有些力不从心。

### 10.5.1 “Hello World”示例程序

这次来真格的了。不过在开始前，先要指出使用何种形式把一个字符发给UART。可以把发送字符的代码做成一个子程序，由其它函数呼叫来输出数据。这样的好处在于，如果输出设备变了，则只需重写这个子程序，就可以使用不同的设备，这种修改动作也有自己的术语，叫“retargeting”（“目标重选”？这词还真不好翻译~）。在大型程序中，这是一个很重要的思想——软件分层，而且这也是“设备无关性”和“可移植性”的前提。

让我们看看一个简单的字符输出子程是啥模样：

UART0\_BASE EQU 0x4000C000

UART0\_FLAG EQU UART0\_BASE+0x018

UART0\_DATA EQU UART0\_BASE+0x000

Putc ; 该子程使用UART来发一个字符

; 入口条件： R0 = 需要发的字符

PUSH {R1,R2, LR} ; 保存寄存器

LDR R1, =UART0\_FLAG

PutcWaitLoop

LDR R2, [R1] ; 读取状态标志

TST R2, #0x20 ; 检查“发送缓冲满”标志

BNE PutcWaitLoop ; 若已满则重试（若UART当掉了，则可能死循环）

LDR R1, =UART0\_DATA ; 有空位时，就把UART发送站寄存器地址加载

STRB R0, [R1] ; 然后通过它把字符送给输出缓冲区

POP {R1,R2, PC} ; 子程返回

在这里的UART是虚构的，其寄存器的地址和位定义都只是为了演示，抛砖引玉。如果雷同，纯属巧合。在实战时，还需要根据自己使用的UART来重塑代码，有些UART还要求更精密地检查状态位。另外，还需要一个用于初始化UART的子程——至少得设置波特率吧。我们为了突出主题，这些细节就不多谈了。在第20章中，有一个具体的例子。

现在，我们就可以通过这个基础设施一般的子程，来构造一系列的消息显示函数，它们都与输出字符的具体硬件无关了。

Puts ; 该子程往UART送一个字符串

; 入口条件：R0 = 待输出字符串的起始地址

; 这个字符串必须以零结尾（C语言格式）

PUSH {R0 ,R1, LR} ; 先保存寄存器

MOV R1, R0 ; 把地址拷贝到R1，因为待会儿调用Putc时还要用

; R0来传参数

PutsLoop

LDRB R0, [R1], #1 ; 读取一个字符，再自增地址

CBZ R0, PutsLoopExit ; 若已到达零字符，则执行完毕，退出

BL Putc ; 把这个字符送往UART

B PutsLoop ; 循环，以输出下一个字符

PutsLoopExit

POP {R0, R1, PC} ; 子程序返回

有了这个Puts，现在终于可以正式请大牌出场了——“Hello World”主程序：

STACK\_TOP EQU 0x20002000 ; SP初始值

UART0\_BASE EQU 0x4000C000

UART0\_FLAG EQU UART0\_BASE+0x018

UART0\_DATA EQU UART0\_BASE+0x000

AREA | Header Code|, CODE

DCD STACK\_TOP ; MSP初始值

DCD Start ; 复位向量

ENTRY

Start ; 主程序入口点

MOV r0, #0 ; 初始化各寄存器

MOV r1, #0

MOV r2, #0

MOV r3, #0

MOV r4, #0

BL Uart0Initialize ; 初始化UART0

LDR r0, =HELLO\_TXT ; 让R0指向客串的起始地址

BL Puts

deadend

B deadend ; 做完了工作，在这里原地打转

;-------------------------------------------------------------

; 各个子程序

;-------------------------------------------------------------

Puts ; 该子程往UART送一个字符串

; 入口条件：R0 = 待输出字符串的起始地址

; 这个字符串必须以零结尾（C语言格式）

PUSH {R0 ,R1, LR} ; 先保存寄存器

MOV R1, R0 ; 把地址拷贝到R1，因为待会儿调用Putc时还要用

; R0来传参数

PutsLoop

LDRB R0, [R1], #1 ; 读取一个字符，再自增地址

CBZ R0, PutsLoopExit ; 若已到达零字符，则执行完毕，退出

BL Putc ; 把这个字符送往UART

B PutsLoop ; 循环，以输出下一个字符

PutsLoopExit

POP {R0, R1, PC} ; 子程序返回

;-------------------------------------------------------------

Putc ; 该子程使用UART来发一个字符

; 入口条件： R0 = 需要发的字符

PUSH {R1,R2, LR} ; 保存寄存器

LDR R1, =UART0\_FLAG

PutcWaitLoop

LDR R2, [R1] ; 读取状态标志

TST R2, #0x20 ; 检查“发送缓冲满”标志

BNE PutcWaitLoop ; 若已满则重试（若UART当掉了，则可能死循环）

LDR R1, =UART0\_DATA ; 有空位时，就把UART发送站寄存器地址加载

STRB R0, [R1] ; 然后通过它把字符送给输出缓冲区

POP {R1,R2, PC} ; 子程返回

;-------------------------------------------------------------

Uart0Initialize

; 与具体硬件有关，也不是主题，故而略

BX LR ; 子程序返回

;-------------------------------------------------------------

HELLO\_TXT

DCB “Hello world\n”,0 ; 定义零结尾的“Hello world”

END ; 本文件结束

本示例代码在各CM3单片机之间都是高度可移植，高度与硬件无关的。事实上，我们只需要自己写Uart0Initialize子程，并调整Putc。之所以日子这么好过，是因为Putc与Puts已经完成了实质的工作。为了锦上添花，最好再提供几个子程，用于输出寄存器的值。首先是输出16进制数的子程。

PutHex ; 以16进制输出寄存器的值

; 入口条件：R0=要显示的值

PUSH {R0-R3,LR}

MOV R3, R0 ; 把R0的值拷贝到R3，以便待会使用R0传递参数给Putc

MOV R0, #’0’ ; 先显示“0x”前缀

BL Putc

MOV R0, #’x’

BL Putc

MOV R1, #8 ; 初始化循环变量

MOV R2, #28 ; 圆圈移位偏移量

PutHexLoop

ROR R3, R2 ; 圆圈右移28格——相当于圆圈左移4格

AND R0, R3, #0xF ; 此时最高4位移至最低4位，提取它们

CMP R0, #0xA ; 转换成ASCII码

ITE GE

ADDGE R0, #55 ; 若大于等于10,则使用字母A-F表示

ORRLT R0, #0x30 ; 否则转换到0-9（原文使用ADDLT，效果相同）

BL Putc ; 输出一个hex字符

SUBS R1, #1 ; 循环变量自减

BNE PutHexLoop ; 检查循环变量是否已减到0，从而循环8次

POP {R0-R3,PC} ; 显示完毕，子程返回

使用这个子程来输出寄存器的值很方便，如果在笔试的时候遇到这个题目，就可以直接抄上去啦！但如果是让你输出10进制数，可就不像乍一看的那么好对付了，而且它还很黄很暴力，能放倒一大批人——要计算32位乘除法（考官阴笑：小样傻眼了吧）！好在CM3下凡后，带出来两颗大力丸——硬件乘除法指令。服下它们，转身以后你会练成护体神功，看见蟑螂也不怕不怕啦！不过，可别神经比较大，因为考官还下了另一个小套儿等你钻呢：在计算期间，我们计算出的字符会是逆序的——即如果不采取措施，0x7B(123)会以321的顺序输出！因此，只好另开一个缓冲区来保存中间结果——先把所有的字符逆序放到这个缓冲区中，来个负负得正，最后使用Puts来一步到位地显示整个结果。在本例中，使用栈空间来存储这个缓冲区，用完即释放——在C编程中，这叫自动局部数组变量。

PutDec ; 以10进制输出寄存器的值

; 入口条件：R0=要显示的值

; 因为是32位宽，最大值（0xffff\_ffff）需要10个10进制位表示，再加上零结尾，共需11字节

PUSH {R0-R5, LR} ; 保存寄存器的值

MOV R3, SP ; 把当前堆栈指针拷贝到R3

SUB SP, SP, #12 ; 为文本缓冲区保留出11个字节（因为是满栈）

MOV R1, #0 ; NULL字符

STRB R1, [R3, #-1]! ; 先把NULL字符写到字符串的结尾（把各字符逆序输出，

; 好“负负得正”）。这里使用了更新基址的预索引

MOV R5, #10 ; R5保存除数

PutDecLoop

**UDIV R4, R0, R5 ; R4 = R0 / 10**

**MUL R1, R4, R5 ; R1 = R4 \* 10**

SUB R2, R0, R1 ; R2 = （ R0-(R0/10)\*10），即个位

ADD R2, 0x30 ; 转换成ASCII（因为R2只能是0-9），亦可使用ORR

STRB R2, [R3, #-1]! ; 把ascii字符送进缓冲区

MOVS R0, R4 ; R0 = 商，并且根更新标志位以检查商是否为零

BNE PutDecLoop ; 若商为零，则已经把所有10进制位都求出

MOV R0, R3 ; R0指向文本缓冲区的起始地址

BL Puts ; 使用Puts显示结果

ADD SP, SP, #12 ; 恢复SP指针

POP {R0-R5, PC} ; 子程返回

怎么样，这下考官得赏你一个麻花两个鸡蛋了吧！如果读者还看过不使用乘除法指令实现该子程的代码，再对比一下两者的执行速度（甚至能相差数百倍），一定会被震憾到，留下刻骨铭心的记忆的。

## 10.6 使用数据存储器

重温一下我们的第一个例子：在我们做到程序连接这一步时，我们手工指定了读/写区的位置。那么我们该如何把数据放到那里呢？正点的解决方法是：在汇编源文件中定义一个相应的数据区。让连接器把数据区中的内容分派到我们指定的位置——从0x2000\_0000(SRAM区的起始)处开始的内存。

回顾当时使用的连接命令：

$> armlink --rw\_base 0x20000000 --ro\_base 0x0 --map -o test1.elf test1.o

STACK\_TOP EQU 0x20002000 ; SP初始值，常数

AREA |Header Code|, CODE

DCD STACK\_TOP ; 栈顶（MSP的）

DCD Start ; 复位向量

ENTRY ; 指示程序从这里开始执行

Start ; 主程序开始

; 初始化寄存器

MOV r0, #10 ; 加载循环变量的初值

MOV r1, #0 ; 初始化运算结果的值

; 计算 10+9+8+...+1

loop

ADD r1, r0 ; R1 += R0

SUBS r0, #1 ; R0自减，并且根据结果更新标志（有”S”后缀）

BNE loop ; if (R0!=0) goto loop;

; 现在，运算结果在R1中

LDR r0, =**MyData1**

STR r1, [r0] ;把结果存入MyData1

deadloop

B deadloop ; 工作完成后，进入无穷循环

**;定义数据区**

**AREA | Header Data|, DATA**

**ALIGN 4**

**MyData1**

**DCD 0 ; Destination of calculation result**

**MyData2**

**DCD 0**

END ; 文件结束标记

在连接阶段，连接器要把DATA区放入读/写存储器中，因此MyData1的地址就将是我们指定的0x2000\_0000。

## 10.7 使用互斥访问实现信号量操作

互斥访问是新出来的，并且专门用于信号量的操作中。最常见的用途，就是确保需要互斥使用的共享资源只被一个任务拥有。

让我们举个例子。记DeviceALocked是一个位于内存中的R/W变量，用于指示设备A是否已经在使用中。任何一个任务，若欲使用设备A，都必须先检查这个变量的值。如果它的值为零，则表示设备可以使用。在任务获取到设备A后，它要把DeviceALocked的值改为1，表示设备A已经被占用。在设备A使用完毕后，该任务通过重新清零DeviceALocked来释放设备A，从而使其它任务可以使用此设备。

看起来这是个如意算盘。不过可否想过，如果两个任务都想访问设备A，是否有潜在的危险？比如，在任务1读取了DeviceALocked后，发现是零于是准备使用此设备，但还没来得及把它改为1，就不巧被调度器切出（比如，轮转调度），然后调度器让任务2执行，于是任务2也读到零，从而它使用设备A。但是在任务2在用完设备A之前，调度器又切回任务1。由于任务1早先读回来的是零，所以它认为设备A是空闲的，于是使用设备A，这时就违背了设备A必须互斥访问的限制，使系统出现紊乱危象！如果设备A是台打印机，则把两个文档的内容打在了一起；如果设备A是油门控制器，则可能使汽车失控或熄火，后果不堪设想。

为避免此问题，必须也保证DeviceALocked的互斥访问。回顾一下第5章，STREX指令是有返回值的，指示访问是成功还是被“驳回”。接上例，如果任务#1和任务#2都使用STREX，则任务#1的STREX将被驳回——返回1，从而任务1知道这期间已经发生了很多事，设备A已被他人占有，就避免了紊乱危象。互斥访问的模式图如图10.3所示。

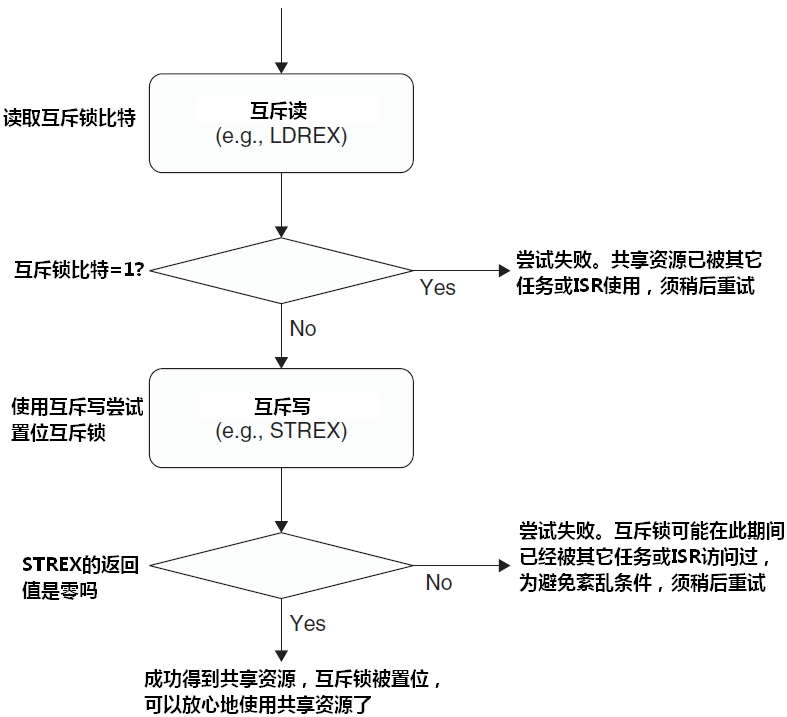


图10.3 使用互斥访问来实现信号量（互斥锁）的操作

上述操作可通过下面的示例代码实现。理解的关键在于，如果互斥访问监视器返回了失败的状态，STREX就被驳回。此时，不会执行写操作，从而保护了互斥锁在访问尝试失败时不被更改。

LockDeviceA

; 一个简单的函数，演示如何尝试锁住设备A

; 返回值：R0=0表示成功，R0=1表示失败

; 如果访问成功，则将把DeviceALocked的值改为1

PUSH {R1, R2, LR}

TryToLockDeviceA

LDR R1, =DeviceALocked

LDREX R2, [R1] ;使用互斥读来标记对互斥锁的访问

CMP R2, #0 ; 检查是否已被锁住

BNE LockDeviceAFailed

DeviceAIsNotLocked

MOV R0, #1 ; 准备锁住设备A

STREX R2, R0, [R1] ; 互斥写

CMP R2, #0

BNE LockDeviceAFailed ; STREX失败，设备A可能已被锁

LockDeviceASucceed

MOV R0, #0 ; 准备返回成功值

POP {R1, R2, PC} ; 子程序返回

LockDeviceAFailed

MOV R0, #1 ; 准备返回失败值

POP {R1, R2, PC} ; 子程序返回

如果返回的是1，则为了避免紊乱危象，任务必须重试。在单处理机系统中，互斥访问主要用在ISR与主程序之间，用以保护它们共享的，并且需要互斥访问的资源（如，一块内存，一个外设）。此时，引起互斥写失败的唯一原因，就是在读写期间曾响应过中断。如果代码在特权级下运行，还可以通过设置PRIMASK，在“测试——置位”期间暂时把中断给掐了。

在多处理机系统中，情况会变得更复杂。此时，除了本机的中断，其它处理机对同一块内存的访问也可以使互斥写操作失败。为了检测到其它处理机对内存的访问，总线系统中必须加入一个“互斥访问监视”的硬件基础设施。它负责检测在互斥读写期间，总线上是否有其它主机访问了互斥锁及其临近的“高危地带”。事实上，在绝大多数低成本的CM3单片机中，都只包含了一个核，因此无需此监视器。

有了这个机制，我们就可以确信共享资源一定能互斥地使用，不会发生紊乱危象。如果一个共享资源在多次尝试时依然无法获取，则可能必须放弃对此资源的请求，有可能先前锁住该资源的任务已经崩溃了。

## 10.8 使用位带实现互斥锁操作

如果存储器系统支持“锁定传送”（locked transfers），或者总线上只有一个主机，还可以使用CM3的位带功能来实现互斥锁的操作。通过使用位带，则可以在C程序中实现互斥锁，但是操作过程与互斥访问是不同的。在使用位带来做资源分配的控制机制时，需要使用位带存储区的内存单元（比如，一个字），该内存单元的每个位表示资源正被特定的任务使用。

在位带别名区的读写实质上是锁定的“读-改-写”（在传送期间总线不能被其它主机占有）。因此，只要每个任务都仅修改分配给它们自己的锁定位，其它任务锁定位的值就不会丢失，即使是两个任务同时写自己的锁定位也不怕，如图10.4所示。

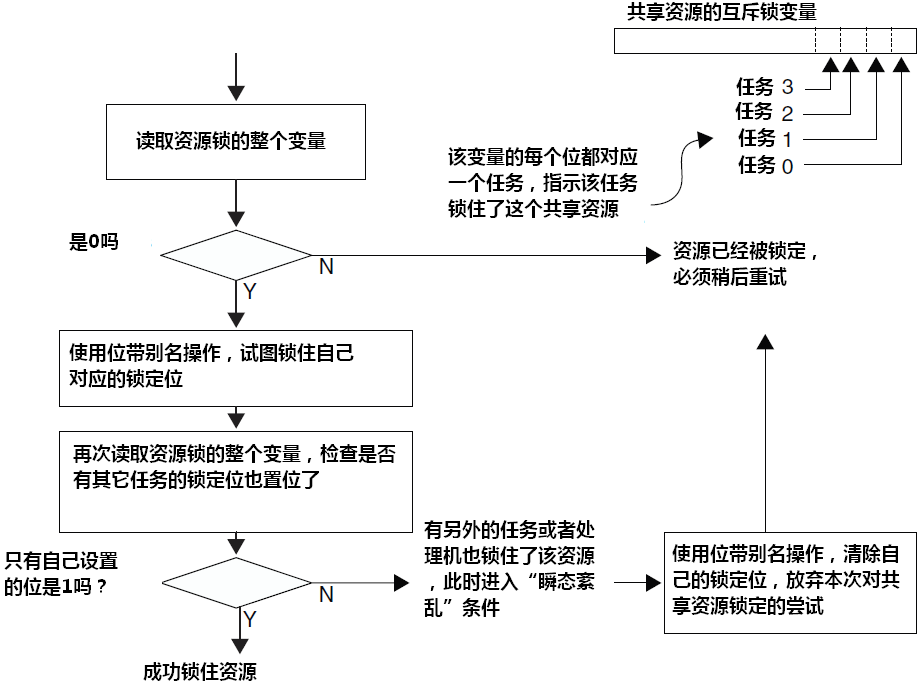


图10.4 使用位带实现互日斥锁的工作流程图

从图10.4我们可以看出，位带操作有可能使共享资源在一个短期内被“多重锁定”，从而有“瞬态紊乱”。但是它不会造成危害，因为任务一定能检测到这个冲突，从而释放自己的锁。

其实，对于“测试并设置”这种互斥锁的简单操作，也可以使用“关中临界区”来保护——也就是在操作前关中断，操作后开中断。这种关中的时间是很短的，因为其它原因导致的关中通常都比这个长得多。只是有时为了无限追求实时性，有一丝希望也会尽最大的努力，就像这两种互斥锁操作那样。

## 10.9 使用位段提取与查表跳转

在第4章中，我们考察了位段提取指令（UBFX）和查表跳转指令（TBB/TBH）。这两条指令可以配合工作，以构建一个非常强大的“跳转树”。这对于电表及数据通信应用程序非常有意义，常使这类程序得到戏剧般地优化。这类程序在工作时，经常要判断各种各样的情况，并且“分类讨论”。有时，还需要进一步细化，作二级甚至多级的比较判断。例如，下图就演示了一个“判决树”，它根据输入量A的各位段编码，来决定启动的任务。

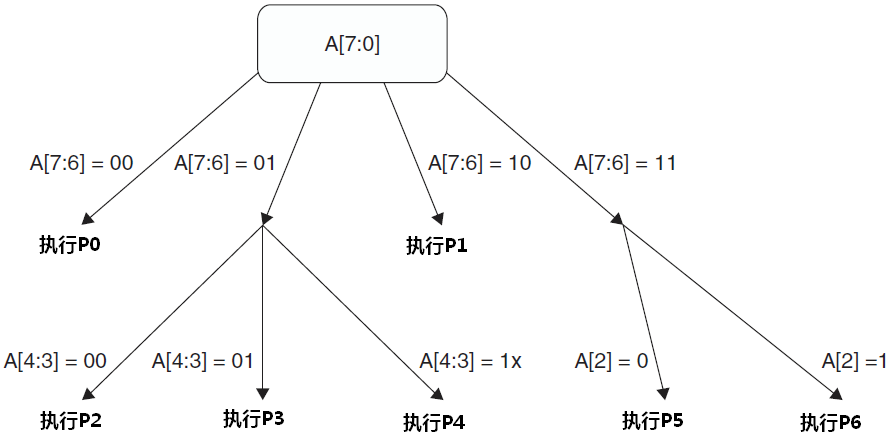


图10.5 通过各位段编码决定操作的判决树示例

DecodeA

LDR R0,=A ; 从内存中读取A的值

LDR R0, [R0]

UBFX R1, R0, #6, #2 ; R1=R0[7:6]

TBB [PC, R1]

BrTable1

DCB ((P0 -BrTable1)/2) ; 如果 A[7:6] = 00 则跳至P0

DCB ((DecodeA1-BrTable1)/2) ; 如果 A[7:6] = 01 则跳至DecodeA1，继续解码

DCB ((P1 -BrTable1)/2) ; 如果 A[7:6] = 10 则跳至P1

DCB ((DecodeA2-BrTable1)/2) ; 如果 A[7:6] = 10 则跳至DecodeA2

DecodeA1

UBFX R1, R0, #3, #2 ; R1=R0[4:3]，准备二级解码

TBB [PC, R1]

BrTable2

DCB ((P2 -BrTable2)/2) ; 如果 A[4:3] = 00 则跳至P2

DCB ((P3 -BrTable2)/2) ; 如果 A[4:3] = 01 则跳至P3

DCB ((P4 -BrTable2)/2) ; 如果 A[4:3] = 10 则跳至P4

DCB ((P4 -BrTable2)/2) ; 如果 A[4:3] = 11 则也跳至P4

DecodeA2

TST R0, #4 ; 只需检测一个位，因此无需UBFX

BEQ P5

B P6

P0 ... ; P0

P1 ... ; P1

P2 ... ; P2

P3 ... ; P3

P4 ... ; P4

P5 ... ; P5

P6 ... ; P6

看，如果使用C来写这个这个程序，则需要使用嵌套的switch和大量的位操作；可现在却干净利落得如此爽快！如果跳转目标更远，可以使用TBH指令。