ARM 芯片的地址重映射

by winday

映射就是一一对应的意思。重映射就是重新分配这种一一对应的关系。

我们可以把存储器看成一个具有输出和输入口的黑盒子。如下图所示，输入量是地址，输出的是对应地址上存储的数据。当然这个黑盒子是由很复杂的半导体电路具现的，具体的实现的方式我们现在不管。存储单位一般是字节。这样，每个字节的存储单元对应一个地址，**当一个合法地址从存储器的地址总线输入后，该地址对应的存储单元上存储的数据就会出现在数据总线上面。**

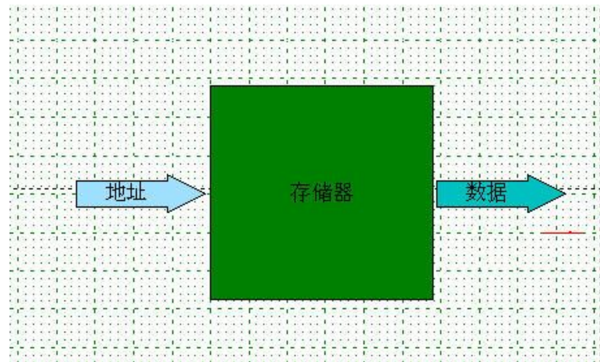

图 1

普通的单片机把可执行代码和数据存放到存储器中。单片机中的 CPU 从储器中取指令代码和数据。其中存储器中每个物理存储单元与其地址是一一对应而且是不可变的。如图 1，CPU 读取 0x00000000 地址上存储单元的过程。

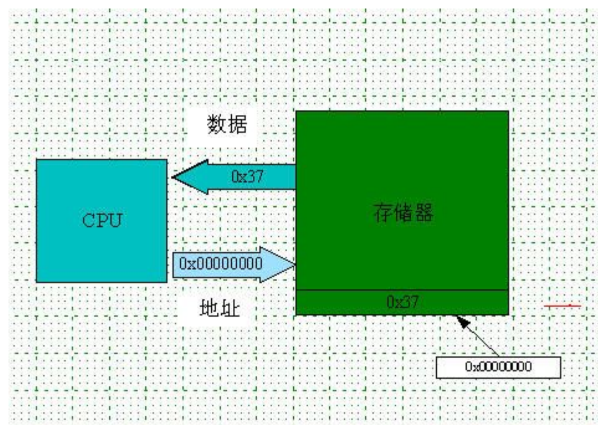

图 2

ARM 比较复杂。ARM 芯片与普通单片机在存储器地址方面的不同在于：ARM 芯片中有些物理存储单元的地址可以根据设置变换。就是说一个物理存储单元现在对应一个地址，经过设置以后，这个存储单元就对应了另外一个地址了。图 3 是随意举了个例子（不要与 ARM 芯片对应），旨在说明地址重映射的过程。图 3 表示把 0x00000000 地址上的存储单元映射到新的地址 0x00000007 上。CPU 存取 0x00000007 就是存取 0x00000000 上的物理存储单元。

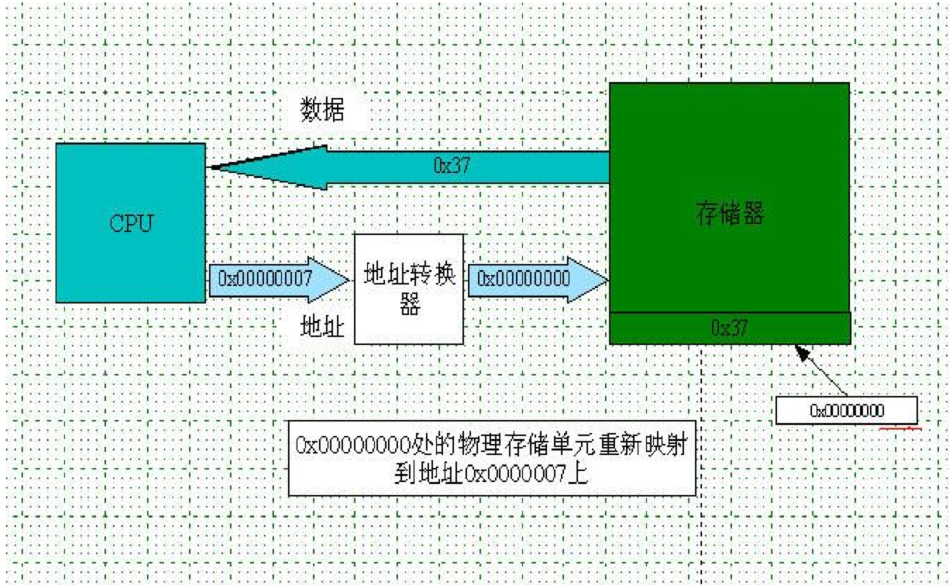

图 3

图 4，图 5 是对 ARM 芯片的两种地址重映射方式的图示。图 4 假设我们的应用程序存放在外扩 FLASH 当中，那么应用程序的异常向量表就存放在 0x80000000 起始的 64 个（其中有 32 个存放异常向量）物理存储单元中。但是 ARM 核发生异常（中断）后是从 0x00000000~0x0000003F 地址范围取异常向量的。所以要把 0x80000000~0x8000003F 范围内的存储单元重新映射到 0x00000000~0x0000003F 地址范围上。以后 CPU 存取 0x00000000~0x0000003F 地址就是存取 0x80000000~0x8000003F 范围内的存储单元。图 4 只显示出第一个异常向量的地址重映射，整个异常向量表的地址重映射等同这个过程。

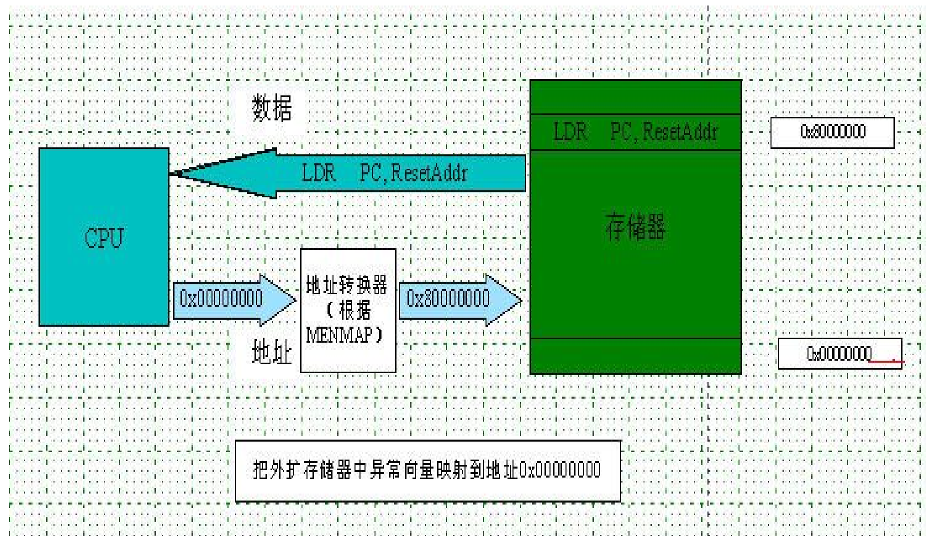

图 4

图 5 图示了 ARM 芯片的另外一种映射方式。这个映射可以由用户决定采用还是不采用（相关代码在工程文件的 startup.s 中，这个文件是第三方提供，用户可以修改）。这个映射主要是为了提高应用程序异常相应得速度。当我们把应用程序存放在片内 FLASH 的时候，异常向量表存放在 0x00000000~0x0000003F 存储单元内。每次发生异常，CPU 从 0x00000000~0x0000003F 地址上取异常向量。但是对 RAM 的存取速度远高于对 FLASH 的存取速度，所以为了提高异常相应速度我们采取以下做法：

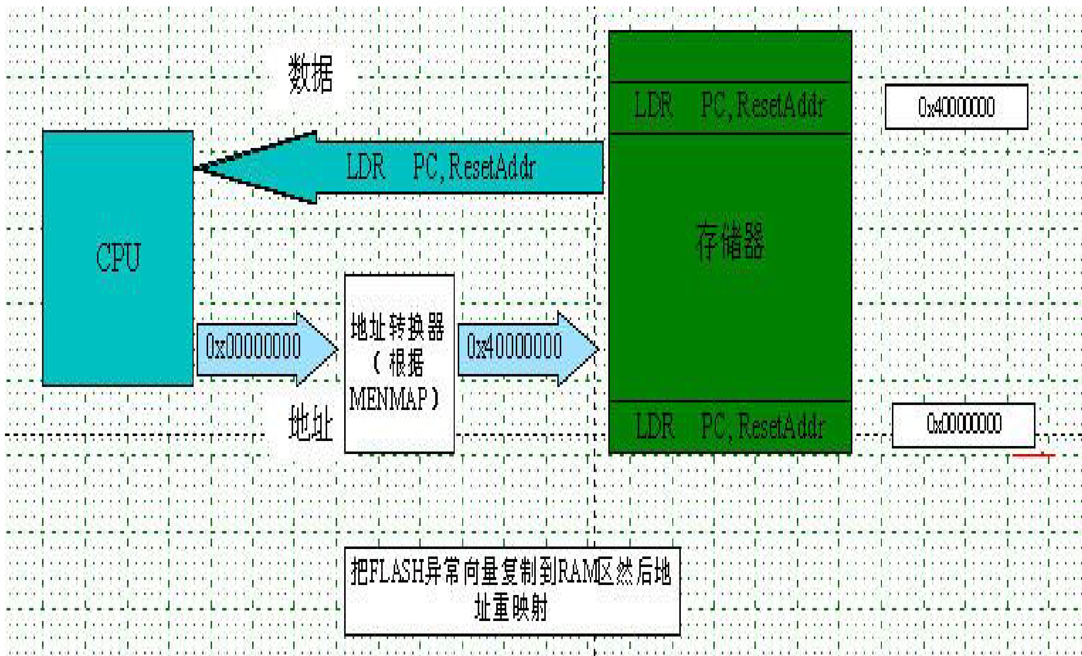

图 5

（1）先把 0x00000000~0x0000003F（FLASH）存储单元内的异常向量表复制到 0x40000000~0x4000003F（片内 RAM 的最低端 64 个字节的存储单元）范围内存储单元中。

（2）把 0x40000000~0x4000003F 范围内存储单元地址重新映射到 0x00000000~0x0000003F 地址范围。

这样做了以后，当异常发生的时候，CPU 取异常向量就是从 RAM 区中的异常向量表中区，速度快了。比如复位中断发生，CPU 从地址 0x00000000 取指令，但此时由于已经过地址重新映射，这个 0x00000000 被地址转换器转换成 0x40000000，CPU 实际上是取的 RAM 区中 0x400000000 这个存储单元内的指令（异常向量）。

当然用户可以不进行这种映射。片内 FLASH 中 0x00000000~0x0000003F 存储单元具有一模一样的异常向量表。只不过不进行这种处理，异常相应速度慢一点。但是这种速度上的差别很多情况下是不必要在意的。

图中的地址转换器受控制寄存器 MENMAP 的控制，用户可以设置 MENMAP 实现对地址重映射的控制。这个地址转换器显然是通过内部硬件电路实现的。

存储器的映射

存储器映射是指把芯片中或芯片外的 FLASH，RAM，外设，BOOTBLOCK 等进行统一编址。即用地址来表示对象。这个地址绝大多数是由厂家规定好的，用户只能用而不能改。用户只能在挂外部 RAM 或 FLASH 的情况下可进行自定义。

ARM7TDMI 的存储器映射可以有 0X00000000~0XFFFFFFFF 的空间，即 4G 的映射空间，但所有器件加起来肯定是填不满的。一般来说，0X00000000 依次开始存放 FLASH——0X00000000，SRAM——0X40000000，BOOTBLOCK，外部存储器 0X80000000，VPB（低速外设地址，如 GPIO，UART）——0XE0000000，AHB（高速外设：向量中断控制器，外部存储器控制器）——从 0XFFFFFFFF 回头。他们都是从固定位置开始编址的，而占用空间又不大，如 AHB 只占 2MB，所以从中间有很大部分是空白区域，用户若使用这些空白区域，或者定义野指针，就可能出现取指令中止或者取数据中止。

由于系统在上电复位时要从 0X00000000 开始运行，而第一要运行的就是厂家固化在片子里的 BOOTBLOCK，这是判断运行哪个存储器上的程序，检查用户代码是否有效，判断芯片是否加密，芯片是否 IAP（在应用编程），芯片是否 ISP（在系统编程），所以这个 BOOTBLOCK 要首先执行。而芯片中的 BOOTBLOCK 不能放在 FLASH 的头部，因为那要存放用户的异常向量表的，以便在运行、中断时跳到这来找入口，所以 BOOTBLOCK 只能放在 FLSAH 尾部才能好找到，呵呵。而 ARM7 的各芯片的 FLASH 大小又不一致，厂家为了 BOOTBLOCK 在芯片中的位置固定，就在编址的 2G 靠前编址的位置虚拟划分一个区域作为 BOOTBLOCK 区域，这就是重映射，这样访问<2G 即<0X80000000 的位置时，就可以访问到在 FLASH 尾部的 BOOTBLOCK 区了。

BOOTBLOCK 运行完就是要运行用户自己写的启动代码了，而启动代码中最重要的就是异常向量表，这个表是放在 FLASH 的头部首先执行的，而异常向量表中要处理多方面的事情，包括复位、未定义指令、软中断、预取指中止、数据中止、IRQ（中断）,FIQ（快速中断），而这个异常向量表是总表，还包括许多分散的异常向量表，比如在外部存储器，BOOTBLOCK，SRAM 中固化的，不可能都由用户直接定义，所以还是需要重映射把那些异常向量表的地址映到总表中。

ARM 启动代码分析—philips 的 LPC2xxx 系列

```
/*******************************************************************************************
*File:       startup.s
*Author:   Embest  w.h.xie        2005.02.21
*Desc:      lpc22xx\lpc212x\lpc211x\lpc210x  startup  code
*History:
*note  modify:   cui  jian  jie         2006-4-25
*comment:
*******************************************************************************************/
# 处理器的七种工作方式的常量定义
.EQU        Mode_USR,                    0x10              #用户模式
.EQU        Mode_FIQ,                    0x11              #FIQ 模式
.EQU        Mode_IRQ,                    0x12              #IRQ 模式
.EQU        Mode_SVC,                    0x13              #超级用户模式
.EQU        Mode_ABT,                    0x17              #终止模式
.EQU        Mode_UND,                    0x1B              #未定义模式
.EQU        Mode_SYS,                    0x1F              #系统模式

# 中断屏蔽位
.EQU        I_Bit,                  0x80             //IRQ 中断控制位，当被置位时，IRQ 中断被禁止
.EQU        F_Bit,                  0x40             //FIQ 中断控制位，当被置位时，FIQ 中断被禁止

# 状态屏蔽位
.EQU        T_bit,                  0x20        //T 位，置位时在 Thumb 模式下运行，清零时在 ARM 下运行

# 定义程序入口点
.globl  _start                         global?
                .code  32

                .TEXT

_start:

# 中断向量表

Vectors:
                LDR        PC,  Reset_Addr             //把 Reset_Addr 地址处的内容放入 PC 中
                LDR        PC,  Undef_Addr
                LDR        PC,  SWI_Addr
                LDR        PC,  PAbt_Addr
                LDR        PC,  DAbt_Addr
                .long      0xb9205f80           @  keep  interrupt  vectors  sum  is  0
```

```
                LDR     PC, [PC, #-0xff0]     //当前 PC 值减去 0xFF0 等于 IRQ 中断入口地址
                LDR     PC, FIQ_Addr
#地址表
Reset_Addr:                                  #该地址标号存放 Reset_Handler 程序段的入口地址
        .long   Reset_Handler
Undef_Addr:                                  #该地址标号存放 Undef_Handler 程序段的入口地址
        .long   Undef_Handler
SWI_Addr:                                    #该地址标号存放 SWI_Handler 程序段的入口地址
        .long   SWI_Handler
PAbt_Addr:                                   #该地址标号存放 PAbt_Handler 程序段的入口地址
        .long   PAbt_Handler
DAbt_Addr:
        .long   DAbt_Handler
        .long   0
IRQ_Addr:                                    #地址标号处存放一个无效的数据
        .long   0
FIQ_Addr:                                    #该地址标号存放 FIQ_Handler 程序段的入口地址
        .long   FIQ_Handler

Undef_Handler:
        B        Undef_Handler
PAbt_Handler:
        B        PAbt_Handler
DAbt_Handler:
        B        DAbt_Handler

#软中断的中断服务子程序入口地址
SWI_Handler:
                STMFD   sp!, {r0-r3, r12, lr}   //入栈，现场数据保护
                MOV     r1, sp                  //把堆栈指针 SP 存入 R1 中
                MRS     r0, spsr             //把 SPSR 值存入 R0，SPSR 值为产生软中断时的 CPSR
                TST     r0, #T_bit              //判断 R0（SPSR）的 T 位是否为 0
                #SPSR 的 T 位不为 0，工作在 Thumb 模式下
                LDRNEH  r0, [lr,#-2]            //SPSR 的 T 位不为 0，则[lr-2]-〉r0
                BICNE   r0, r0, #0xFF00         // SPSR 的 T 位不为 0，清除 r0 的 Bit8~Bit15 位
                # SPSR 的 T 位为 0，工作在 ARM 模式下
                LDREQ   r0, [lr,#-4]            // SPSR 的 T 位为 0，则[lr-4] -〉r0
                BICEQ   r0, r0, #0xFF000000     // SPSR 的 T 位为 0，清除 r0 的 Bit24~Bit131 位

                # R0 is interrupt number        //R0 是中断号
                # R1 is stack point                 //R1 是堆栈指针

                BL      SWI_Exception           //进入软中断处理程序
                LDMFD   sp!, {r0-r3, r12, pc}^  //出栈，现场数据恢复
```

```
# 快速响应中断的中断服务自程序的入口地址
FIQ_Handler:
                STMFD     SP!, {R0-R3, LR}          //入栈的现场保护
#                BL          FIQ_Exception           //进入 FIQ 的中断处理程序
                LDMFD     SP!, {R0-R3, LR}          //出栈，恢复现场
                SUBS       PC, LR, #4                //返回到主程序

# 复位后程序处理的入口地址
Reset_Handler:
                BL        RemapSRAM           //进行存储器映射的操作
#下面几行代码用来判断当前的工作模式
                MRS       R0, CPSR            //读 CPSR 到寄存器 R0
                AND       R0, R0, #0x1F       //R0 = R0 AND 0x1F
                CMP       R0,  #Mode_USR      //比较 R0 和 #Mode_USR，二者相减
//如果相等则说明当前处在用户模式下，需要通过产生 11 号软中断进入系统模式。因为下面的初始化堆栈
//需要在不同的工作模式下切换，而在用户模式下不能直接切换，只有系统模式可以，所以要通过产生 11
//号软中断切换到用户模式。
                SWIEQ     #11

                BL        InitStack           //进行堆栈初始化工作
#------------------------------------------------------------------------------
#- 初始化 C 变量
#-----------------------
#- 下表由连接器自动产生
#- RO: 只读=代码区。
#- RW: 可读可写=预先初始化的数据(初始化的全局变量)和预先被清零的数据(未初始化的全局变量)。.
#- ZI: 预先被清零的数据区(未初始化的全局变量)
#- 预先被初始化的数据区定位在代码区之后。
#- 预先被清零的数据区定位在预先被初始化的数据区之后。
#- 注意数据区的位置 :
#- I 如果用 ARM SDT, 当链接器选择 no -rw-base 时, 数据区被映射在代码区之后
#- 你可以把数据区房子内部的 SRAM( -rw-base=0x40 or 0x34)中
#- 或者放在外部的 SRAM( -rw-base=0x2000000 )中。
#- 注意：为了提高代码的密度，预先被初始化的数据必须尽可能的少。
#------------------------------------------------------------------------------
#该部分程序功能：先判断当前是在 RAM 中运行还是在 FLASH 中运行，如果在 FLASH 中运行，先把 FLASH
#中的预先赋值的 RW 段数据和未赋值的 ZI 段数据都搬移到 RAM 区中，再把 ZI 段数据全部清零；如果程#序就是在 RAM 中运行，则直接把 ZI 段数据清零。
                .extern      Image_RO_Limit     /* ROM 区中数据段的起始地址*/
                .extern      Image_RW_Base      /* RW 段起始地址 */
                .extern      Image_ZI_Base      /* ZI 段的起始地址*/
                .extern      Image_ZI_Limit     /* ZI 段的结束地址加 1 */

                ldr         r0, =Image_RO_Limit  /* 取 ROM 区中数据段的首地址 */
```

```
            ldr         r1, =Image_RW_Base      /* 取 RAM 区中 RW 段的目标首地址*/
            ldr         r3, =Image_ZI_Base      /*取 RAM 区中 ZI 段的首地址 */
            cmp          r0, r1                 /* 比较 ROM 区中数据段首地址和 RAM 区中 RW 段目标首地址 */
            beq          NoRW                   /*相等代表当前是在 RAM 中运行*/
LoopRw:     cmp          r1, r3                 /*不相等则和 RAM 区中 ZI 段的目标地址比较*/
            ldrcc       r2, [r0], #4   /*如果 r1<r3，则把 r0 地址上的数据读出到 r2 中，然后 r0=r0+4*/
            strcc       r2, [r1], #4   /*如果 r1<r3，则把 r2 内数据写入道 r1 地址中，然后 r1=r1+4*/
            bcc          LoopRw    /*如果 r1<r3，则跳转到 LoopRw 继续执行*/
NoRW:       ldr          r1, =Image_ZI_Limit    /* 取 ZI 段的结束地址 */
            mov          r2, #0                 /*将 r2 赋 0*/
LoopZI:     cmp          r3, r1                 /* 将 ZI 段清零*/
            strcc        r2, [r3], #4    /*如果 r3<r1，将 r2 内容写入到 r3 地址单元中，然后 r3=r3+1*/
            bcc          LoopZI      /*如果 r3<r1(即 C=0)，则跳转到 LoopZI */

            .extern   Main                      /*声明外部变量*/
            B            Main                   /*t 跳转到用户的主程序入口*/
# 为每一种模式建立堆栈，ARM 堆栈指针向下生长
InitStack:
                MOV      R1, LR                 //把该子程序返回地址保留在 R1 中

                LDR      R0, =Top_Stack             //取栈定地址到 R0 中
#进入未定义模式，并禁止 FIQ 中断和 IRQ 中断
                MSR      CPSR_c, #Mode_UND|I_Bit|F_Bit
#设置未定义模式下堆栈的栈顶指针
                MOV      SP, R0
                SUB      R0, R0, #UND_Stack_Size    #未定义模式下堆栈深度

#进入终止模式，并禁止禁止 FIQ 中断和 IRQ 中断
                MSR      CPSR_c, #Mode_ABT|I_Bit|F_Bit
#紧接着未定义模式下的堆栈，设置终止模式下栈顶指针
                MOV      SP, R0
                SUB      R0, R0, #ABT_Stack_Size        #终止模式下堆栈深度

#进入 FIQ 模式，并禁止 FIQ 中断和 IRQ 中断
                MSR      CPSR_c, #Mode_FIQ|I_Bit|F_Bit
#紧接着终止模式下的堆栈，设置下 FIQ 模式下栈顶指针
                MOV      SP, R0
                SUB      R0, R0, #FIQ_Stack_Size        #FIQ 模式下的堆栈深度

#进入 IRQ 模式，并禁止 FIQ 中断和 IRQ 中断
                MSR      CPSR_c, #Mode_IRQ|I_Bit|F_Bit
#紧接着 FIQ 模式下的堆栈，设置 IRQ 模式下的栈顶指针
                MOV      SP, R0
                SUB      R0, R0, #IRQ_Stack_Size        #IRQ 模式下的堆栈深度
```

```
#进入超级用户模式，并禁止 FIQ 中断和 IRQ 中断
            MSR      CPSR_c, #Mode_SVC|I_Bit|F_Bit
#紧接着 IRQ 模式下的堆栈，设置超级用户下的栈顶指针
            MOV      SP, R0
            SUB      R0, R0, #SVC_Stack_Size          #超级用户下的堆栈深度

#设置进入用户模式
            MSR      CPSR_c, #Mode_USR
#紧接着超级用户模式下的堆栈，设置用户模式下的栈顶指针，剩余的空间都开辟为堆栈
            MOV      SP, R0

            MOV      PC, R1                    #堆栈初始化子程序返回
# 重映射 SRAM 区
RemapSRAM:

           MOV     R0, #0x40000000        //RAM 区首地址
           LDR     R1, =Vectors           //向量表首地址
#下面一段程序是把从 0x00000000 开始的 64 个字节（FLASH 中的中断向量表和地址表）搬移到以
#0x40000000 为首地址的 RAM 区中
           LDMIA   R1!, {R2-R9}        //把以[R1]为首地址的 32 个字节数据装载到 R2-R9 中
           STMIA   R0!, {R2-R9}        //把 R2-R9 中的数据存入以[R0]为首地址的单元中
           LDMIA   R1!, {R2-R9}        //把以[R1]为首地址的 32 个字节数据装载到 R2-R9 中
           STMIA   R0!, {R2-R9}        ////把 R2-R9 中的数据存入以[R0]为首地址的单元中
#下面几行代码设置存储器映射控制寄存器
           LDR     R0, =MEMMAP      //取 MEMMAP 地址到 R0
           MOV     R1, #0x02
           STR     R1, [R0]       //给 MEMMAP 赋值为 0x02，设置中断向量从 RAM 区从新映射

           mov     pc,  lr              //跳转到主程序

#下面一段程序代码是进入软中断来切换系统的工作模式，当希望从一种模式切换入另一种模式时，可以通
#过调用下面对应标号的程序段进入软中断。在软中断处理程序中会根据所给定的中断号处理，执行 SWI #num 后软中断号被存
入 R0 中。
.globl    disable_IRQ
.globl    restore_IRQ
.globl    ToSys
.globl    ToUser

# 禁止 IRQ

disable_IRQ:
          STMFD    SP!, {LR}                    //把 LR 值压入堆栈
          swi      #0                           //产生 0 号软中断，  0 -〉R0
```

```
        LDMFD    SP!, {pc}                //恢复 PC 值，返回

# 恢复 IRQ

restore_IRQ:
        STMFD    SP!, {LR}                //把 LR 值压入堆栈
        swi      #1                       //产生 1 号软中断，1 –〉R0
        LDMFD    SP!, {pc}                //恢复 PC 值，返回

#进入系统工作模式

ToSys:
        STMFD    SP!, {LR}                //把 LR 值压入堆栈
        swi      #11                      //产生 11 号软中断，11 –〉R0
        LDMFD    SP!, {pc}                //恢复 PC 值，返回

# 进入用户工作模式

ToUser:
        STMFD    SP!, {LR}                //把 LR 值压入堆栈
        swi      #12                      //产生 12 号软中断，11 –〉R0
        LDMFD    SP!, {pc}                //恢复 PC 值，返回
# 软中断处理代码

SWI_Exception:
        STMFD    SP!, {R2-R3,LR}      //把 R2，R3，LR 值入栈
        #0 号软中断的处理程序
        CMP      R0,  #0              //将 R0 和 0 比较
        //以下 4 行带 EQ 条件的代码均为当 R0 为 0 时应该执行的语句
        MRSEQ    R2,  SPSR            //把 SPSR 读入到 R2 中
        STREQ    R2,  [R1]            /把 R2 的值存入到[R1]中
        ORREQ    R2,  R2, #0x80       //把 R2 的 Bit7 位置 1
        MSREQ    SPSR_c, R2           //把 R2 的值写入到 SPSR_c 中，即禁止 IRQ
        #1 号软中断的处理程序
        CMP      R0,  #1              //比较 R0 值和 1
        LDREQ    R2,  [R1]            //相等则把[R1]中的数据存入 R2 中
        MSREQ    SPSR_c, R2           //相等把 R2 的值写入到 SPSR_c 中，恢复 IRQ

       #11 号软中断的处理程序
        CMP      R0,  #11             //比较 R0 的值和 11
        MRSEQ    R2,  SPSR            //相等则把 SPSR 的值转存入到 R2 中
        BICEQ    R2,  R2, #0x1F       //相等则把 R2 的 Bit0~Bit4 全部清零
        ORREQ    R2,  R2, #Mode_SYS //相等则把 R2 与#Mode_SYS 相与再存入 R2
        MSREQ    SPSR_c, R2          //相等则把 R2 的值存入 SPSR_c 中，即进入系统模式
```

#12 号软中断的处理程序

```
CMP       R0,   #12                //比较 R0 的值和 12
MRSEQ     R2,   SPSR               //相等则把 SPSR 的值存入 R2
BICEQ     R2,   R2, #0x1F          //相等则把 R2 的 Bit0~Bit4 清零
ORREQ     R2,   R2, #Mode_USR      //相等则把 R2 与#Mode_USR 相与再存入 R2 中
MSREQ     SPSR_c, R2               //相等则把 R2 存入 SPSR_c，即进入用户模式

LDMFD     SP!, {R2-R3,PC}          //恢复 R2、R3、PC 值，返回


.END                               //汇编代码段结束
```