



10年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要良心教育、做专业教育，更要做受人尊敬的职业教育。

《从实践中学 ARM 体系结构与接口技术》

作者：华清远见

专业始于专注 卓识源于远见

第 7 章 ARM 异常中断处理及编程

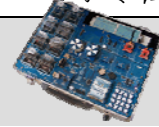
本章简介

几乎每种处理器都支持特定异常处理，中断也是异常的一种。了解处理器的异常处理相关知识，是学习一种处理器的重要环节。本章主要内容：

- ARM 异常中断处理概述
- ARM 体系异常种类
- ARM 异常的优先级
- ARM 处理器模式和异常
- ARM 异常响应和处理程序返回
- ARM 应用系统中异常中断处理程序的安装
- ARM 的 SWI 异常中断处理程序设计
- FIQ 和 IRQ 异常中断程序设计
- 基于 ARM9 芯片 S3C2410X 异常中断程序设计

7.1

ARM 异常中断处理概述



1. 中断的概念

什么是中断？我们从一个生活中的例子引入。你正在家中看书，突然电话铃响了，你放下书本去接电话，和来电话的人交谈，然后放下电话，回来继续看你的书。这就是生活中的“中断”的现象，也就是正常的工作过程被外部的事件打断了。

在处理器中，所谓中断，是一个过程，即 CPU 在正常执行程序的过程中，遇到外部 / 内部的紧急事件需要处理，暂时中断（中止）当前程序的执行，而转去为事件服务，待服务完毕，再返回到暂停处（断点）继续执行原来的程序。为事件服务的程序称为中断服务程序或中断处理程序。严格地说，上面的描述是针对硬件事件引起的中断而言的。用软件方法也可以引起中断，即事先在程序中安排特殊的指令，CPU 执行到该类指令时，转去执行相应的一段预先安排好的程序，然后再返回去执行原来的程序，这可称为软中断。把软中断考虑进去，可给中断再下一个定义：中断是一个过程，是 CPU 在执行当前程序的过程中因硬件或软件的原因插入了另一段程序运行的过程。因硬件原因引起的中断过程的出现是不可预测的，即随机的，而软中断是事先安排的。

2. 中断源的概念

仔细研究一下生活中的中断，对于理解中断的概念也很有好处。什么可以引起中断，其实生活中很多事件可以引起中断：有人按了门铃，电话铃响了，你的闹钟响了，你烧的水开了……诸如此类的事件，我们把可以引起中断的信号源称为中断源。

3. 中断优先级的概念

设想一下，我们正在看书，电话铃响了，同时又有人按了门铃，你该先做哪样呢？如果你正在等一个很重要的电话，你一般不会去理会门铃；而反之，你正在等一个重要的客人，则可能就不会去理会电话了。如果不是这两者（即不是等电话，也不是等人上门），你可能会按你通常的习惯去处理。总之，这里存在一个优先级的概念，处理器中也是如此，也有优先级的概念，即同时有多个中断源递交中断申请时的中断控制器对中断源的响应优先级别。需要注意的是，优先级的概念不仅仅发生在两个中断同时产生的情况，也发生在一个中断已产生而又有一个中断产生的情况。比如你正在接电话，有人按门铃的情况；或你正开门与人交谈，又有电话响了的情况。这时也需要根据中断源的优先级来决定下一动作。

ARM 处理器中有 7 种类型的异常，按优先级从高到低的排列如下：复位异常（Reset）、数据异常（Data Abort）、快速中断异常（FIQ）、外部中断异常（IRQ）、预取异常（Prefetch Abort）、软件中断异常（SWI）和未定义指令异常（Undefined instruction）。

注意，在 ARM 处理器中异常（Exception）和中断（Interrupt）有些差别，异常主要是从处理器被动接受异常的角度出发，而中断带有向处理器主动申请的色彩。在本书中，对“异常”和“中断”不做严格区分，两者都是指请求处理器打断正常的程序执行流程，进入特定程序循环的一种机制。

7.2

ARM 体系异常种类



ARM 体系结构中，存在 7 种异常处理。当异常发生时，处理器会把 PC 设置为一个特定的存储器地址。这一地址放在被称为向量表（vector table）的特定地址范围内。向量表的入口是一些跳转指令，跳转到专门处理某个异常或中断的子程序。

存储器映射地址 0x00000000 是为向量表（一组 32 位字）保留的。在有些处理器中，向量表可以选择定位在存储空间的高地址（从偏移量 0xffff0000 开始）。一些嵌入式操作系统，如 Linux 和 Windows CE 就利用了这一特性。

如表 7-1 所示列出了 ARM 的 7 种异常。

表 7-1 ARM 的 7 种异常

异常类型	处理器模式	执行低地址	执行高地址
复位异常 (Reset)	特权模式	0x00000000	0xFFFF0000
未定义指令异常 (Undefined interrupt)	未定义指令中止模式	0x00000004	0xFFFF0004
软中断异常 (Software Abort)	特权模式	0x00000008	0xFFFF0008
预取异常 (Prefetch Abort)	数据访问中止模式	0x0000000C	0xFFFF000C
数据异常 (Data Abort)	数据访问中止模式	0x00000010	0xFFFF0010
外部中断请求 (IRQ)	外部中断请求模式	0x00000018	0xFFFF0018
快速中断请求 (FIQ)	快速中断请求模式	0x0000001C	0xFFFF001C

异常处理向量表如图 7-1 所示。

当异常发生时，分组寄存器 r14 和 SPSR 用于保存处理器状态，操作伪指令如下：

```
R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4:0] = exception mode number
CPSR[5] = 0 /*进入 ARM 状态*/
If <exception_mode> == reset or FIQ then
    CPSR[6] = 1 /*屏蔽快速中断 FIQ*/
    CPSR[7] = 1 /*屏蔽外部中断 IRQ*/
PC = exception vector address
```

异常返回时，SPSR 内容恢复到 CPSR，连接寄存器 r14 的内容恢复到程序计数器 PC。



图 7-1 异常处理向量表

1. 复位异常

当处理器的复位引脚有效时，系统产生复位异常中断，程序跳转到复位异常中断处理程序处执行。复位异常中断通常用在以下两种情况下：

- (1) 系统上电。
- (2) 系统复位。

当复位异常时，系统（处理器自动执行的，以下几个异常相同）执行下列伪操作。

```
R14_svc = UNPREDICTABLE value
SPSR_svc = UNPREDICTABLE value
CPSR[4:0] = 0b10011 /*进入特权模式*/
CPSR[5] = 0 /*处理器进入 ARM 状态*/
CPSR[6] = 1 /*禁止快速中断*/
```

```

CPSR[7] = 1          /*禁止外设中断*/
If high vectors configured then
    PC = 0xffff0000
Else
    PC = 0x00000000
    
```

复位异常中断处理程序将进行一些初始化工作，内容与具体系统相关。下面是复位异常中断处理程序的主要功能：

- (1) 设置异常中断向量表。
- (2) 初始化数据栈和寄存器。
- (3) 初始化存储系统，如系统中的 MMU 等。
- (4) 初始化关键的 I/O 设备。
- (5) 使能中断。
- (6) 处理器切换到合适的模式。
- (7) 初始化 C 变量，跳转到应用程序执行。

2. 未定义指令异常

当 ARM 处理器执行协处理器指令时，它必须等待一个外部协处理器应答后，才能真正执行这条指令。若协处理器没有响应，则发生未定义指令异常。未定义指令异常可用于在没有物理协处理器的系统上，对协处理器进行软件仿真，或通过软件仿真实现指令集扩展。例如，在一个不包含浮点运算的系统中，CPU 遇到浮点运算指令时，将发生未定义指令异常中断，在该未定义指令异常中断的处理程序中可以通过其他指令序列仿真浮点运算指令。

仿真功能可以通过下面的步骤实现。

(1) 将仿真程序入口地址链接到向量表中未定义指令异常中断入口处 (0x00000004 或 0xffff0004)，并保存原来的中断处理程序。

(2) 读取该未定义指令的 bits[27:24]，判断其是否是一条协处理器指令。如果 bits[27:24] 值为 0b1110 或 0b110x，该指令是一条协处理器指令；否则，由软件仿真实现协处理器功能，可以通过 bits[11:8] 来判断要仿真的协处理器功能（类似于 SWI 异常实现机制）。

(3) 如果不仿真该未定义指令，程序跳转到原来的未定义指令异常中断的中断处理程序执行。

当未定义异常发生时，系统执行下列伪操作。

```

r14_und = address of next instruction after the undefined instruction
SPSR_und = CPSR
CPSR[4:0] = 0b11011          /*进入未定义指令模式*/
CPSR[5] = 0                  /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1                  /*禁止外设中断*/
If high vectors configured then
    PC = 0xffff0004
Else
    PC = 0x00000004
    
```

3. 软中断 SWI

软中断异常发生时，处理器进入特权模式，执行一些特权模式下的操作系统功能。软中断异常发生时，处理器执行下列伪操作。

```

r14_svc = address of next instruction after the SWI instruction
SPSR_und = CPSR
CPSR[4:0] = 0b10011          /*进入特权模式*/
CPSR[5] = 0                  /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1                  /*禁止外设中断*/
If high vectors configured then
    PC = 0xffff0008
Else
    PC = 0x00000008
    
```

4. 预取指令异常

预取指令异常是由系统存储器报告的。当处理器试图去取一条被标记为预取无效的指令时，发生预取异常。

如果系统中不包含 MMU 时，指令预取异常中断处理程序只是简单地报告错误并退出。若包含 MMU，引起异常的指令的物理地址被存储到内存中。

预取异常发生时，处理器执行下列伪操作。

```
r14_svc = address of the aborted instruction + 4
SPSR_und = CPSR
CPSR[4:0] = 0b10111          /*进入特权模式*/
CPSR[5] = 0                  /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1                  /*禁止外设中断*/
If high vectors configured then
    PC = 0xffff000C
Else
    PC = 0x0000000C
```

5. 数据访问中止异常

数据访问中止异常是由存储器发出数据中止信号，它由存储器访问指令 Load/Store 产生。当数据访问指令的目标地址不存在或者该地址不允许当前指令访问时，处理器产生数据访问中止异常。

当数据访问中止异常发生时，处理器执行下列伪操作。

```
r14_abt = address of the aborted instruction + 8
SPSR_abt = CPSR
CPSR[4:0] = 0b10111
CPSR[5] = 0
/*CPSR[6]保持不变*/
CPSR[7] = 1                  /*禁止外设中断*/
If high vectors configured then
    PC = 0xffff000C10
Else
    PC = 0x00000010
```

当数据访问中止异常发生时，寄存器的值将根据以下规则进行修改：

- (1) 返回地址寄存器 r14 的值只与发生数据异常的指令地址有关，与 PC 值无关。
- (2) 如果指令中没有指定基址寄存器回写，则基址寄存器的值不变。
- (3) 如果指令中指定了基址寄存器回写，则寄存器的值和具体芯片的 Abort Models 有关，由芯片的生产商指定。
- (4) 如果指令只加载一个通用寄存器的值，则通用寄存器的值不变。
- (5) 如果是批量加载指令，则寄存器中的值是不可预知的值。
- (6) 如果指令加载协处理器寄存器的值，则被加载寄存器的值不可预知。

6. 外部中断 IRQ

当处理器的外部中断请求引脚有效，而且 CPSR 寄存器的 I 控制位被清除时，处理器产生外部中断 IRQ 异常。系统中各外部设备通常通过该异常中断请求处理器服务。

当外部中断 IRQ 发生时，处理器执行下列伪操作。

```
r14_irq = address of next instruction to be executed + 4
SPSR_irq = CPSR
CPSR[4:0] = 0b10010        /*进入特权模式*/
CPSR[5] = 0                /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1                /*禁止外设中断*/
If high vectors configured then
    PC = 0xffff0018
Else
    PC = 0x00000018
```

7. 快速中断 FIQ

当处理器的快速中断请求引脚有效且 CPSR 寄存器的 F 控制位被清除时,处理器产生快速中断请求 FIQ 异常。

当快速中断异常发生时,处理器执行下列伪操作。

```
r14_fiq = address of next instruction to be executed + 4
SPSR_fiq = CPSR
CPSR[4:0] = 0b10001 /*进入 FIQ 模式*/
CPSR[5] = 0
CPSR[6] = 1
CPSR[7] = 1
If high vectors configured then
    PC= 0xffff001c
Else
    PC = 0x0000001c
```

7.3 ARM 异常的优先级



每一种异常按表 7-2 中设置的优先级得到处理。

表 7-2 异常优先级

优 先 级	异 常
最高	1 复位异常
	2 数据中止
	3 快速中断请求
	4 外部中断请求
	5 预取指令异常
	6 软中断
最低	7 未定义指令

异常可以同时发生,处理器则按表 7-2 中设置的优先级顺序处理异常。例如,处理器上电时发生复位异常,复位异常的优先级最高,所以当产生复位时,它将优先于其他异常得到处理。同样,当一个数据访问中止异常发生时,它将优先于除复位异常外的其他所有异常而得到处理。

优先级最低的两种异常是软中断和未定义指令异常。因为正在执行的指令不可能既是一条 SWI 指令,又是一条未定义指令,所以软件中断异常和未定义指令异常享有相同的优先级。

7.4 ARM 处理器模式和异常



每一种异常都会导致内核进入一种特定的模式。如表 7-3 所示显示了 ARM 处理器异常及其对应的模式。此外,也可以通过编程改变 CPSR,进入任何一种 ARM 处理器模式。

注意 和系统模式是仅有的不可通过异常进入的两种模式,也就是说,要进入这两种模式,必须通过编程改变 CPSR。

表 7-3 ARM 处理器异常及其对应模式

异 常	模 式	用 途
快速中断请求	FIQ	进行快速中断请求处理
外部中断请求	IRQ	进行外部中断请求处理

SWI	SVC	进行操作系统的高级处理
复位	SVC	进行操作系统的高级处理
预取指令中止异常	ABORT	虚存和存储器保护
数据中止异常	ABORT	虚存和存储器保护
未定义指令	Undefined	软件模拟硬件协处理器

7.5 ARM 异常响应和处理程序返回

7.5.1 中断响应的概念

中断的响应过程：当有事件产生，进入中断之前我们必须先记住现在看到书的第几页了，或拿一个书签放在当前页的位置，然后去处理不同的事情（因为处理完了，我们还要回来继续看书）。电话铃响我们要到放电话的地方去，门铃响我们要到门那边去，也就说是不同的中断，我们要在不同的地点处理，而这个地点通常是固定的。

通常，中断响应大致可以分为以下几个步骤：（1）保护断点，即保存下一个将要执行的指令的地址，就是把地址送入堆栈；（2）寻找中断入口，根据不同的中断源所产生的中断，查找不同的入口地址；（3）执行中断处理程序；（4）中断返回。执行完中断指令后，就从中断处返回到主程序，继续执行。

7.5.2 ARM 异常响应流程

1. 判断处理器状态

当异常发生时，处理器自动切换到 ARM 状态，所以在异常处理函数中要判断在异常发生前处理器是 ARM 状态还是 Thumb 状态。这可以通过检测 SPSR 的 T 位来判断。

通常情况下，只有在 SWI 处理函数中才需要知道异常发生前处理器的状态。所以在 Thumb 状态下，调用 SWI 软中断异常必须注意以下两点：

- （1）发生异常的指令地址为 lr-2 而不是 lr-4。
- （2）Thumb 状态下的指令是 16 位的，在判断中断向量号时使用半字加载指令 LDRH。

下面的例子显示了一个标准的 SWI 处理函数，在函数中通过 SPSR 的 T 位判断异常发生前的处理器状态。

```

T_bit EQU 0x20                ;bit 5. SPSR 中的 ARM/Thumb 状态位
:
:
SWIHandler
STMFD sp!, {r0-r3,r12,lr}    ;寄存器压栈，保护程序现场
MRS r0, spsr                 ;读 SPSR 寄存器，判断异常发生前的处理器状态
TST r0, #T_bit               ;检测 SPSR 的 T 位，判断异常发生前是否为 Thumb 状态
LDRNEH r0,[lr,#-2]           ;如果是 Thumb 状态，使用半字加载指令读取发生异常的指令地址
BICNE r0,r0,#0xFF00          ;提取中断向量号
LDREQ r0,[lr,#-4]            ;如果是 ARM 状态，使用字加载指令，读取发生异常的指令地址
BICEQ r0,r0,#0xFF000000      ;提取中断向量号并将中断向量号存入 r0
; r0 存储中断向量号
CMP r0, #MaxSWI              ;判断中断是否超出范围
LDRLS pc, [pc, r0, LSL#2]    ;如果未超出范围，跳转到软中断向量表 Switable
B SWIOutOfRange              ;如果超出范围，跳转到软中断越界处理程序
switable
DCD do_swi_1
DCD do_swi_2
:
:
do_swi_1
; 1 号软中断处理函数
    
```

```
LDMFD sp!, {r0-r3,r12,pc}^ ; Restore the registers and return.
                                ;恢复寄存器并返回
do_swi_2                        ;2号软中断处理函数
:
```

2. 向量表

如前面介绍向量表时提到的，每一个异常发生时总是从异常向量表开始跳转。最简单的一种情况是向量表中的每一条指令直接跳向对应的异常处理函数。其中快速中断处理函数 `FIQ_Handler()` 可以直接从地址 `0x1C` 处开始，省去了一条跳转指令，如图 7-2 所示。

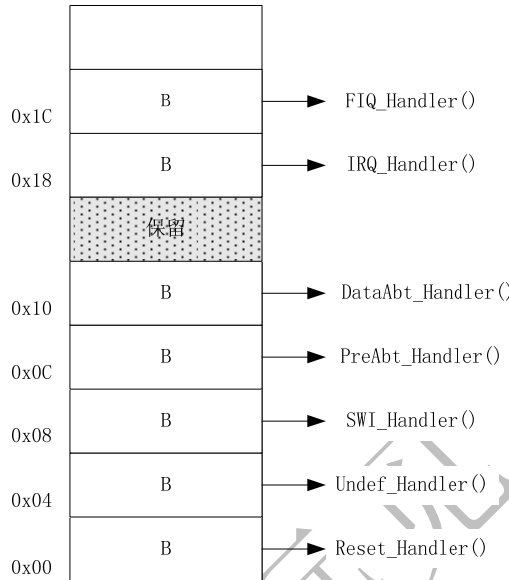


图 7-2 异常处理向量表

跳转指令 `B` 的跳转范围为 $\pm 32\text{MB}$ ，但很多情况下不能保证所有的异常处理函数都定位在向量的 32MB 范围内，需要更大范围的跳转；而且由于向量表空间的限制，只能由一条指令完成。具体实现方法有下面两种。

(1) 方法一：

```
MOV PC, #imme_value
```

这种办法将目标地址直接赋值给 `PC`。但这种方法受格式限制不能处理任意立即数。这个立即数由一个 8 位数值循环右移偶数位得到。

(2) 方法二：

```
LDR PC, [PC+offset]
```

把目标地址先存储在某一个合适的地址空间，然后把这个存储器单元的 32 位数据传送给 `PC` 来实现跳转。

这种方法对目标地址值没有要求，但是存储目标地址的存储器单元必须在当前指令的 $\pm 4\text{KB}$ 空间范围内。

注意

在计算指令中引用 `offset` 数值的时候，要考虑处理器流水线中指令预取对 `PC` 值的影响。

7.5.3 从异常处理程序中返回

当一个 ARM 异常处理返回时，一共有 3 件事情需要处理：通用寄存器的恢复、状态寄存器的恢复及 PC 指针的恢复。通用寄存器的恢复采用一般的堆栈操作指令即可，下面重点介绍状态寄存器的恢复及 PC 指针的恢复。

1. 恢复被中断程序的处理器状态

PC 和 CPSR 的恢复可以通过一条指令来实现，下面是 3 个例子。

```
MOVS PC,LR
SUBS PC,LR,#4
LDMFD SP!,{PC}^
```

这几条指令是普通的数据处理指令，特殊之处在于它们把程序计数器寄存器 PC 作为目标寄存器，并且带了特殊的后缀“S”或“^”。其中“S”或“^”的作用就是使指令在执行时，同时完成从 SPSR 到 CPSR 的复制，达到恢复状态寄存器的目的。

2. 异常的返回地址

异常返回时，另一个非常重要的问题就是返回地址的确定。前面提到过，处理器进入异常时会有一个保存 LR 的动作，但是该保持值并不一定是正确中断的返回地址。以一个简单的指令执行流水状态图来对此加以说明，如图 7-3 所示。

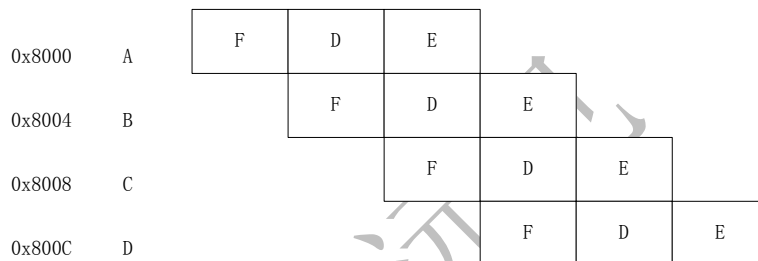


图 7-3 3 级流水线示例

在 ARM 架构中，PC 值指向当前执行指令地址加 8。也就是说，当执行指令 A（地址 0x8000）时，PC 等于 0x8000+8=0x8008，即等于指令 C 的地址。假设指令 A 是 BL 指令，则当执行时，会把 PC 值（0x8008）保存到 LR 寄存器。但是，接下来处理器会对 LR 进行一次自动调整，使 LR=LR-0x4。所以，最终保存在 LR 里面的是图 7-3 中所示的 B 指令地址。所以当从 BL 返回时，LR 里面正好是正确的返回地址。

同样的跳转机制在所有的 LR 自动保存操作中都存在。当进入中断响应时，处理器对保存的 LR 也进行一次自动调整，并且跳转动作也是 LR=LR-0x04。由此，就可以对不同异常类型的返回地址依次比较。

假设在指令 B 处（地址 0x8004）发生了异常，进入异常响应后，LR 经过跳转保存的地址值应该是 C 的地址 0x8008。

(1) 软中断异常。如果发生软中断异常，即指令 B 为 SWI 指令，从 SWI 中断返回后下一条执行指令就是 C，正好是 LR 寄存器保存的地址，所以直接把 LR 恢复给 PC 即可。

(2) IRQ 或 FIQ 异常。如果发生的是 IRQ 或 FIQ 异常，因为外部中断请求中断了正在执行的指令 B，当中断返回后，需要重新回到 B 指令执行。也就是说，返回地址应该是 B（0x8004），需要把 LR 减 4 送 PC。

(3) Data Abort 数据中止异常。在指令 B 处进入数据异常的响应，但导致数据异常的原因却应该是上一条指令 A。当中断处理程序恢复数据异常后，要回到 A 重新执行导致数据异常的指令，因此返回地址应该是 LR 加 8。


为方便起见，如表 7-4 所示总结了各异常和返回地址的关系。

表 7-4 异常和返回地址

异常	地址	用途
复位	—	复位没有定义 LR
数据中止	LR-8	指向导致数据中止异常的指令

FIQ	LR-4	指向发生异常时正在执行的指令
IRQ	LR-4	指向发生异常时正在执行的指令
预取指令中止	LR-4	指向导致预取指令异常的指令
SWI	LR	执行 SWI 指令的下一条指令
未定义指令	LR	指向未定义指令的下一条指令

7.6 ARM 应用系统中异常中断处理程序的安装



7.6.1 使用汇编语言安装异常处理程序

可以使用汇编语言在系统启动时直接安装异常处理程序。

下面的例子显示了系统从 0x0 地址启动，直接安装异常处理程序的方法。

```

Vector_Init_Block
    LDR PC, Reset_Addr
    LDR PC, Undefined_Addr
    LDR PC, SWI_Addr
    LDR PC, Prefetch_Addr
    LDR PC, Abort_Addr
    NOP ;保留向量
    LDR PC, IRQ_Addr
    LDR PC, FIQ_Addr

Reset_Addr DCD Start_Boot
Undefined_Addr DCD Undefined_Handler
SWI_Addr DCD SWI_Handler
Prefetch_Addr DCD Prefetch_Handler
Abort_Addr DCD Abort_Handler
           DCD 0 ;保留向量
IRQ_Addr DCD IRQ_Handler
FIQ_Addr DCD FIQ_Handler
    
```

有些情况下，系统 0x0 地址不一定是 ROM。如果 0x0 地址为 RAM，那么就需要系统将中断向量表从 ROM 复制到 RAM，下面的例子显示了这样一个过程。

```

MOV R8, #0
ADR R9, Vector_Init_Block
LDMIA R9!, {r0-r7} ;复制中断向量表 (8 字)
STMIA R8!, {r0-r7}
LDMIA R9!, {r0-r7} ;复制由伪操作 DCD 定义的地址
STMIA R8!, {r0-r7}
    
```

7.6.2 使用 C 语言编写安装处理函数

程序中有时需要在 main() 函数中使用 C 语言安装中断向量表，这就要求指令经编译后的解码能安装在内存的正确位置。

1. 向量表中使用跳转指令的情况

如果在向量表中使用跳转指令，使用下面的步骤完成向量表的安装。

- (1) 读取异常处理程序的地址。
- (2) 从异常处理程序地址中减去向量表中的偏移。
- (3) 为适应指令流水线，将上一步得到的地址减 8。
- (4) 将得到的结果右移 2 位，得到以字为单位的地址偏移量。

(5) 将结果的高 8 位清零，得到跳转指令的 24 位偏移量。

(6) 将上一步得到的结果和 0xea000000（无条件跳转指令编码）做逻辑与操作，从而得到要写到向量表中的跳转指令的正确编码。

下面的例子显示了这样一个标准过程。

```
unsigned Install_Handler (unsigned routine, unsigned *vector)
{ unsigned vec, oldvec;
vec = ((routine - (unsigned)vector - 0x8)>>2);
if ((vec & 0xFF000000))
{
/*判断错误*/
printf ("Installation of Handler failed");
exit (1);
}
vec = 0xEA000000 | vec;
oldvec = *vector;
*vector = vec;
return (oldvec);
}
```

2. 在向量表中使用加载 PC 指令

在向量表中使用加载 PC 指令，按照下面的步骤完成。

(1) 读取异常处理程序地址。

(2) 从异常处理程序地址中减去向量表中的偏移。

(3) 为适应指令流水线，将上一步得到的地址减 8。

(4) 保留结果的后 12 位。

(5) 将结果与 0xe59ff000（LDR PC, [PC,#offset]）做逻辑或操作，从而得到要写到向量表中的跳转指令的正确编码。

(6) 将异常处理程序的地址放到相应的存储单元。

下面的例子显示了一个标准的 C 语言过程。

```
unsigned Install_Handler (unsigned location, unsigned *vector)
{ unsigned vec, oldvec;
vec = ((unsigned)location - (unsigned)vector - 0x8) | 0xe59ff000;
oldvec = *vector;
*vector = vec;
return (oldvec);
}
```

7.7

ARM 的 SWI 异常中断处理程序设计



本节主要介绍编写 SWI 处理程序时需要注意的几个问题，包括判断 SWI 中断号，使用汇编语言编写 SWI 异常处理函数，使用 C 语言编写 SWI 异常处理函数，在特权模式下使用 SWI 异常中断处理，从应用程序中调用 SWI。

1. 判断 SWI 中断号

当发生 SWI 异常，进入异常处理程序时，异常处理程序必须提取 SWI 中断号，从而得到用户请求的特定 SWI 功能。

在 SWI 指令的编码格式中，后 24 位称为指令的 Comment field。该域保存的 24 位数，即为 SWI 指令的中断号，如图 7-4 所示。

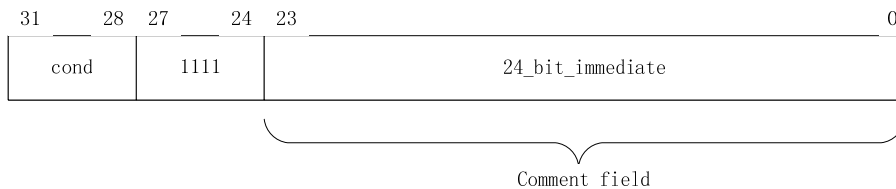


图 7-4 SWI 指令编码格式

第一级的 SWI 处理函数通过 LR 寄存器内容得到 SWI 指令地址，并从存储器中得到 SWI 指令编码。通常这些工作通过汇编语言、内嵌汇编来完成。

下面的例子显示了提取中断向量号的标准过程。

```
PRESERVE8
AREA TopLevelSwi, CODE, READONLY ;第一级 SWI 处理函数
EXPORT SWI_Handler
SWI_Handler
STMFD sp!, {r0-r12,lr} ;保存寄存器
LDR r0, [lr, #-4] ;计算 SWI 指令地址
BIC r0, r0, #0xff000000 ;提取指令编码的后 24 位
;
;提取出的中断号放到 r0 寄存器，函数返回
;
LDMFD sp!, {r0-r12,pc}^ ;恢复寄存器
END
```

例子中，使用 lr-4 得到 SWI 指令的地址，再通过“BIC r0, r0, #0xFF000000”指令提取 SWI 指令中断号。

2. 使用汇编语言编写 SWI 异常处理函数

最简单的方法是利用得到的中断向量号，使用跳转表直接跳转到实现相应 SWI 功能的处理程序。

下面的例子使用汇编语言实现了这种跳转。

```
CMP r0, #MaxSWI ;中断向量范围检测
LDRLS pc, [pc, r0, LSL #2]
B SWIOutOfRange
SWIJumpTable
DCD SWInum0
DCD SWInum1
;使用 DCD 定义各功能函数入口地址
SWInum0 ;0 号中断
B EndofSWI
SWInum1 ;1 号中断
B EndofSWI
;
EndofSWI
```

3. 使用 C 语言编写 SWI 异常处理函数

虽然第一级 SWI 处理函数（完成中断向量号的提取）必须用汇编语言完成，但第二级中断处理函数（根据提取的中断向量号，跳转到具体的处理函数）就可以使用 C 语言来完成。

因为第一级的中断处理函数已经将中断号提取到寄存器 r0 中，所以根据 AAPCS 函数调用规则，可以直接使用 BL 指令跳转到 C 语言函数，而且中断向量号作为第一个参数被传递到 C 函数。

例如汇编中使用了“BL C_SWI_Handler”跳转到 C 语言的第二级处理函数，则第二级的 C 语言函数示例如下。

```
void C_SWI_handler (unsigned number)
{
    switch (number)
    {
        case 0 : /*软件中断 0*/
            break;
        case 1 : /*软件中断 1*/
```

```
break;
...
default : /*未知中断 - 报告错误*/
}
}
```

另外，如果需要传递的参数多于一个，那么可以使用堆栈，将堆栈指针作为函数的参数传递给 C 类型的二级中断处理程序，这样就可以实现在两级中断之间传递多个参数。

例如：

```
MOV r1, sp ;将传递的第二个参数（堆栈指针）放到 r1 中
BL C_SWI_Handler ;调用 C 函数
```

相应的 C 函数的入口变为：

```
void C_SWI_handler(unsigned number, unsigned *reg)
```

同时，C 函数也可以通过堆栈返回操作的结果。

4. 从应用程序中调用 SWI

可从汇编语言或 C/C++ 中调用 SWI。

(1) 从汇编应用程序中调用 SWI。从汇编应用程序中调用 SWI，只要遵循 AAPCS 标准即可。调用前，设定所有必需的值并发出相关的 SWI。例如：

```
MOV r0, #65 ;将软中断的子功能号放到 r0 中
SWI 0x0
```

注意

SWI 指令和其他所有 ARM 指令一样，都可以被条件执行。

(2) 从 C 应用程序中调用 SWI。在 C 或 C++ 应用程序中调用 SWI，要将 C 语言的子程序用编译器扩展 `_swi` 声明。例如：

```
__swi(0) void my_swi(int);
...
...
...
my_swi(65);
```

编译器扩展 `_swi` 确保了 SWI 以内联方式进行编译，而没有额外的开销，但有如下的 AAPCS 限制：

- 函数调用参数只能使用 `r0~r3` 传递。
- 函数返回值只能通过 `r0~r3` 传递。

向内联的 SWI 函数传递参数和向实际的子函数传递参数基本类似，但返回值的情况比较复杂。如果有 2~4 个返回值，则必须告诉编译程序返回值是以结构形式返回的，并使用 `__value_in_regs` 伪操作声明。这是因为基于结构值的函数通常被处理为一个 `void`（空）型函数，且第一个自变量必须为存放结果结构的地址。

7.8

FIQ 和 IRQ 异常中断程序设计



1. 中断分支

ARM 内核只有两个外部中断输入信号 `nFIQ` 和 `nIRQ`。但对于一个系统来说，中断源可能多达几十个。为此，在系统集成的时候，一般都会有一个异常控制器来处理异常信号，如图 7-5 所示。

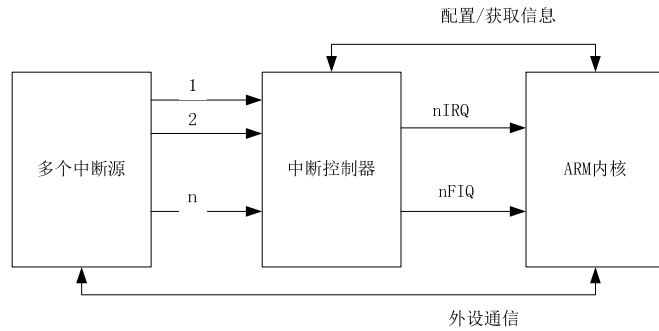


图 7-5 中断系统

这时候用户程序可能存在多个 IRQ/FIQ 的中断处理函数。为了使从向量表开始的跳转始终能找到正确的处理函数入口，需要设置一套处理机制和方法。

多数情况下是由软件来处理异常分支的，因为软件可以通过读取中断控制器来获得中断源的信息，如图 7-6 所示。

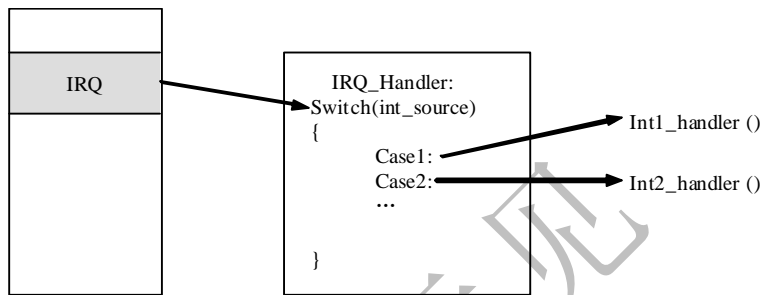


图 7-6 软件控制中断分支

有些芯片可能支持特殊的硬件分支功能，这需要查看具体的芯片说明。

因为软件的灵活性，可以设计出比图 7-6 更好的流程控制方法，如图 7-7 所示。Int_vector_table 是用户自己开辟的一块存储器空间，里面按次序存放异常处理函数的地址。IRQ_Handler()从中断控制器获取中断源信息，然后再从 Int_vector_table 中的对应地址单元得到异常处理函数的入口地址，完成一次异常响应的跳转。这种方法的好处是用户程序在运行过程中，能够很方便地动态改变异常服务内容。

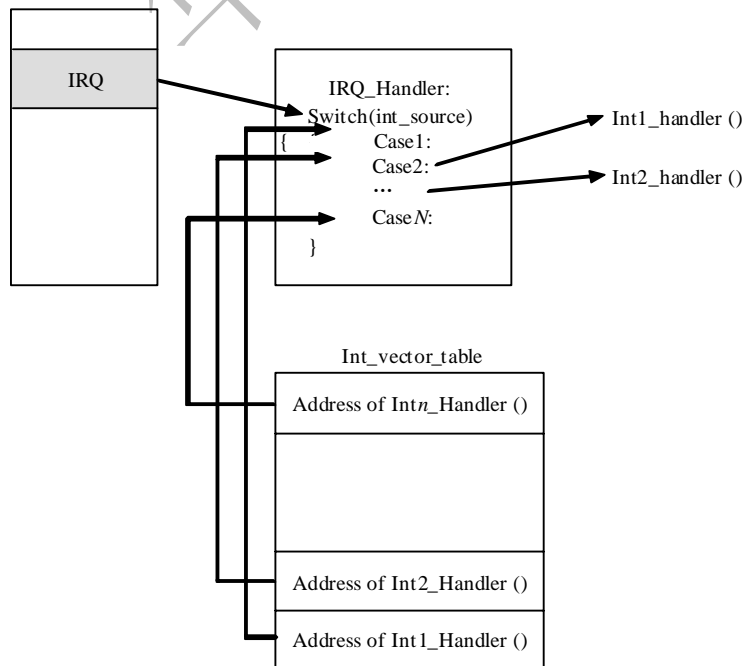


图 7-7 灵活的软件分支设计

进入异常处理程序后,用户可以完全按照自己的意愿来进行程序设计,包括调用 Thumb 状态的函数等。但对于绝大多数的系统来说,有两个步骤必须处理:一是现场保护;二是要把中断控制器中对应的中断状态标识清除,表明该中断请求已经得到响应,否则,中断函数退出以后,又会被再一次触发,从而进入周而复始的死循环。

2. ARM 编译器对中断处理函数编写的扩展

考虑到中断处理函数在现场保护和返回地址的处理上与普通函数的不同之处,不能直接把普通函数体连接到异常向量表上,需要在上面加上一层封装,下面是一个例子。

```

IRQ_Handler                ;中断相应函数
    STMFD    SP!,{r0-r12,lr} ;保护现场,一般只需要保护{r0-r3,LR}
    BL      IrqHandler      ;进入普通处理函数,C或汇编均可
...
    LDMFD    sp!,{r0-r12,LR} ;恢复现场
    SUBS    pc,lr,#4        ;中断返回,注意返回地址
    
```

为了方便使用高级语言直接编写异常处理函数,ARM 编译器对此做了特定的扩展,可以使用函数声明关键字 `__irq`,这样编译出来的函数就可以满足异常响应对现场保护和恢复的需要,并且自动加入 LR 减 4 的处理,符合 IQR 和 FIQ 中断处理的要求。

下面的例子显示了使用 `__irq` 对中断处理函数产生的影响。

C 语言源程序如下:

```

__irq void IRQHandler (void)
{
    volatile unsigned int *base = (unsigned int *) 0x80000000;
    if (*base == 1)
    {
        /*调用C语言中断处理函数*/
        C_int_handler();
    }
    /*清除中断标志*/
    *(base+1) = 0;
}
    
```

使用 ARM 编译器编译出的汇编代码如下:

```

STMFD sp!,{r0-r4,r12,lr}
MOV r4,#0x80000000
LDR r0,[r4,#0]
SUB sp,sp,#4
CMP r0,#1
BLEQ C_int_handler
MOV r0,#0
STR r0,[r4,#4]
ADD sp,sp,#4
LDMFD sp!,{r0-r4,r12,lr}
SUBS pc,lr,#4
    
```

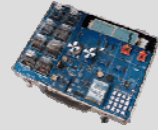
如果不使用 `__irq` 子程序声明关键字,编译出的汇编代码如下:

```

STMFD sp!,{r4,lr}
MOV r4,#0x80000000
LDR r0,[r4,#0]
CMP r0,#1
BLEQ C_int_handler
MOV r0,#0
STR r0,[r4,#4]
LDMFD sp!,{r4,pc}
    
```

7.9

基于 ARM9 芯片 S3C2410X 异常中断程序设计



7.9.1 S3C2410X 中断机制分析

1. S3C2410X 中断概述

S3C2410X 的中断控制器可以接受多达 56 个中断源的中断请求。S3C2410X 的中断源可以由片内外设提供，比如 DMA、UART、IIC 等，其中 UARTn 中断和 EINTn 中断是逻辑或的关系，它们共用一条中断请求线。

S3C2410X 中断控制器为了提供对更多（大于 32 个）中断的支持，还增加了子中断功能部分，这部分中断源如下所示（11 个）。

- (1) INT_ADC: A/D 转换中断。
- (2) INT_TC: 触摸屏中断。
- (3) INT_ERR2: UART2 收发错误中断。
- (4) INT_TXD2: UART2 发送中断。
- (5) INT_RXD2: UART2 接收中断。
- (6) INT_ERR1: UART1 收发错误中断。
- (7) INT_TXD1: UART1 发送中断。
- (8) INT_RXD1: UART1 接收中断。
- (9) INT_ERR0: UART0 收发错误中断。
- (10) INT_TXD0: UART0 发送中断。
- (11) INT_RXD0: UART0 接收中断。

当 S3C2410X 收到来自片内外设和外部中断请求引脚的多个中断请求时，S3C2410X 的中断控制器在中断仲裁过程后向 S3C2410X 内核请求 FIQ 或 IRQ 中断。中断仲裁过程依靠处理器的硬件优先级逻辑，处理器在仲裁过程结束后将仲裁结果记录到 INTPND 寄存器，以告知用户中断由哪个中断源产生。S3C2410X 的中断控制器的处理过程如图 7-8 所示。

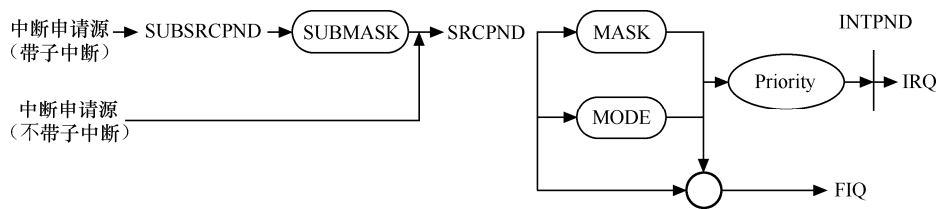


图 7-8 S3C2410X 的中断控制器

S3C2410X 的中断控制器的任务是在有多个中断发生时，选择其中一个中断通过 IRQ 或 FIQ 向 CPU 内核发出中断请求。

实际上，最初 CPU 内核只有 FIQ（快速中断请求）和 IRQ（通用中断请求）两种中断，其他中断都是各个芯片厂家在设计芯片时，通过加入一个中断控制器来扩展定义的，这些中断根据中断的优先级高低来进行处理，更符合实际应用系统中要求提供多个中断源的要求。例如，如果你定义所有的中断源为 IRQ 中断（通过中断模式寄存器设置），并且同时有 10 个中断发出请求，这时可以通过读中断优先级寄存器来确定哪一个中断将被优先执行。

S3C2410X 的中断处理流程如下：当有中断源请求中断时，中断控制器处理中断请求，并根据处理结果向 CPU 内核发出 IRQ 请求或 FIQ 请求，同时，CPU 的程序指针 PC 将指向 IRQ 异常入口 (0x18) 或 FIQ 异常入口 (0x1C)，程序从 IRQ 异常入口 (0x18) 或 FIQ 异常入口 (0x1C) 开始执行。

外部中断 EINT4~7、EINT8~23 是通过 S3C2410 IO 控制器中的中断管理功能扩展出的两组外部中断，用到 S3C2410 IO 控制器中的 EXTINTn、EXTMASK、EINTPND 等寄存器。本节稍后会介绍这几个寄存器的用法。

2. S3C2410X 中断控制

(1) 程序状态寄存器的 F 位和 I 位。如果 CPSR 程序状态寄存器的 F 位被设置为 1，那么 CPU 将不接受来自中断控制器的 FIQ（快速中断请求）；如果 CPSR 程序状态寄存器的 I 位被设置为 1，那么 CPU 将不接受来自中断控制器的 IRQ（中断请求）。因此，为了使能 FIQ 和 IRQ，必须先将 CPSR 程序状态寄存器的 F 位和 I 位清零，并且中断屏蔽寄存器 INTMSK 中相应的位也要清零。

(2) 中断模式（INTMOD）。ARM920T 提供了两种中断模式，即 FIQ 模式和 IRQ 模式。所有的中断源在中断请求时都要确定使用哪一种中断模式。

(3) 中断挂起寄存器（INTPND）。S3C2410X 有两个中断挂起寄存器：源中断挂起寄存器（SRCPND）和中断挂起寄存器（INTPND），用于指示对应的中断是否被激活。当中断源请求中断的时候，SRCPND 寄存器的相应位被置 1，同时 INTPND 寄存器中也有唯一的一位在仲裁程序后被自动置 1。如果屏蔽位被设置为 1，相应的 SRCPND 位会被置 1，但是 INTPND 寄存器不会有变化；如果 INTPND 被置位，只要标志 I 或标志 F 一旦被清零，就会执行相应的中断服务程序。在中断服务子程序中要先向 SRCPND 中的相应位写 1 来清除源挂起状态，再用同样的方法来清除 INTPND 的相应位的挂起状态。

可以通过“INTPND = INTPND;”来实现清零，以避免写入不正确的数据引起错误。

(4) 中断屏蔽寄存器（INTMSK）。当 INTMSK 寄存器的屏蔽位为 1 时，对应的中断被禁止；当 INTMSK 寄存器的屏蔽位为 0 时，则对应的中断正常执行。如果一个中断的屏蔽位为 1，在该中断发出请求时挂起位还是会被设置为 1，但中断请求都不被受理。

3. S3C2410X 中断源

在 56 个中断源中，有 30 个中断源提供给中断控制器，其中外部中断 EINT4~7 通过逻辑“或”的形式提供给中断控制器，EINT8~EINT23 也通过逻辑“或”的形式提供给中断控制器。S3C2410X 的中断源如表 7-5 所示。

表 7-5 S3C2410X 的中断源

中 断 源	描 述	中断仲裁组
INT_ADC	ADC EOC and Touch interrupt (INT_ADC/INT_TC)	ARB5
INT_RTC	RTC alarm interrupt	ARB5
INT_SPI1	SPI1 interrupt	ARB5
INT_UART0	UART0 Interrupt (ERR、RXD and TXD)	ARB5
INT_IIC	IIC interrupt	ARB4
INT_USBH	USB Host interrupt	ARB4
INT_USBD	USB Device interrupt	ARB4
Reserved	Reserved	ARB4
INT_UART1	UART1 Interrupt (ERR、RXD and TXD)	ARB4
INT_SPI0	SPI0 interrupt	ARB4
INT_SDI	SDI interrupt	ARB3
INT_DMA3	DMA channel 3 interrupt	ARB3
INT_DMA2	DMA channel 2 interrupt	ARB3
INT_DMA1	DMA channel 1 interrupt	ARB3
INT_DMA0	DMA channel 0 interrupt	ARB3
INT_LCD	LCD interrupt (INT_FrSyn and INT_FiCnt)	ARB3
INT_UART2	UART2 Interrupt (ERR、RXD and TXD)	ARB2
INT_TIMER4	Timer4 interrupt	ARB2
INT_TIMER3	Timer3 interrupt	ARB2

INT_TIMER2	Timer2 interrupt	ARB2
INT_TIMER1	Timer1 interrupt	ARB2
INT_TIMER0	Timer0 interrupt	ARB2
INT_WDT	Watch-Dog timer interrupt	ARB1

续表

中 断 源	描 述	中断仲裁组
INT_TICK	RTC Time tick interrupt	ARB1
nBATT_FLT	Battery Fault interrupt	ARB1
Reserved	Reserved	ARB1
EINT8_23	External interrupt 8~23	ARB1
EINT4_7	External interrupt 4~7	ARB1
EINT3	External interrupt 3	ARB0
EINT2	External interrupt 2	ARB0
EINT1	External interrupt 1	ARB0
EINT0	External interrupt 0	ARB0

4. S3C2410X 中断控制寄存器

S3C2410X 的中断控制器有 5 个控制寄存器：源挂起寄存器 (SRCPND)、中断模式寄存器 (INTMOD)、中断屏蔽寄存器 (INTMSK)、中断优先级寄存器 (PRIORITY)、中断挂起寄存器 (INTPND)。中断源发出的中断请求首先被寄存器在中断源挂起寄存器 (SRCPND) 中，INTMOD 把中断请求分为两组：快速中断请求 (FIQ) 和中断请求 (IRQ)，PRIORITY 处理中断的优先级。

(1) 源挂起寄存器 (SRCPND)。中断控制寄存器 INTCON 共有 32 位，每一位对应着一个中断源，当中断源发出中断请求的时候，就会置为源挂起寄存器的相应位；反之，中断的挂起寄存器的值为 0。SRCPND 描述如表 7-6 所示。

表 7-6 SRCPND 描述

寄 存 器	地 址	读/写	描 述	复 位 值
SRCPND	0x4A000000	R/W	0: 中断没有发出请求 1: 中断源发出中断请求	0x00000000

(2) 中断模式寄存器 (INTMOD)。中断模式寄存器 INTMOD 共有 32 位，每一位对应着一个中断源，当中断源的模式位设置为 1 时，对应的中断会由 ARM920T 内核以 FIQ 模式来处理；相反，当模式位设置为 0 时，中断会以 IRQ 模式来处理。INTMOD 描述如表 7-7 所示。

表 7-7 INTMOD 描述

寄 存 器	地 址	读/写	描 述	复 位 值
INTMOD	0x4A000004	R/W	0: IRQ 模式 1: FIQ 模式	0x00000000

注 意

中断控制寄存器中只有一个中断源可以被设置为 FIQ 模式，因此只能在紧急情况下使用 FIQ。如果 INTMOD 寄存器把某个中断设为 FIQ 模式，FIQ 中断不影响 INTPND 和 INTOFFSET 寄存器，因此，这两个寄存器只对 IRQ 模式中断有效。

(3) 中断屏蔽寄存器 (INTMSK)。这个寄存器有 32 位，分别对应一个中断源。当中断源的屏蔽位设置为 1 时，CPU 不响应该中断源的中断请求；反之，等于 0 时，CPU 能响应该中断源的中断请求。INTMSK 描述如表 7-8 所示。

表 7-8 INTMSK 描述

寄 存 器	地 址	读/写	描 述	复 位 值
INTMSK	0x4A000008	R/W	0: 允许响应中断请求 1: 中断请求被屏蔽	0xFFFFFFFF

(4) 中断挂起寄存器 (INTPND)。中断挂起寄存器 INTPND 共有 32 位, 每一位对应着一个中断源, 当中断请求被响应的时候, 相应的位会被设置为 1。在某一时刻只有一个位能为 1, 因此在中断服务子程序中可以通过判断 INTPND 来判断哪个中断正在被响应。在中断服务子程序中必须在清零 SRCPND 中相应位后清零相应的中断挂起位, 清零方法和 SRCPND 相同。INTPND 描述如表 7-9 所示。

表 7-9 INTPND 描述

寄存器	地址	读/写	描述	复位值
INTPND	0x4A000010	R/W	0: 未发出中断请求 1: 中断源发出中断请求	0x00000000

注意

① IRQ 响应的时候不会影响 INTPND 相应的标志位。

② 向 INTPND 等于“1”的位写入“0”时, INTPND 寄存器和 INTOFFSET 寄存器会有无法预知的结果, 因此, 千万不要向 INTPND 的“1”位写入“0”。推荐的清零方法是把 INTPND 的值重新写入 INTPND。

(5) IRQ 偏移寄存器。中断偏移寄存器给出 INTPND 寄存器中哪个是 IRQ 模式的中断请求, 如表 7-10 所示。

表 7-10 INTOFFSET 描述

寄存器	地址	读/写	描述	复位值
INTOFFSET	0x4A000014	R	指示中断请求源的中断号	0x00000000

INTOFFSET 的值对应 INTPND 中的中断挂起位, 取值范围为 0~31。

(6) 外部中断控制寄存器 (EXTINTn)。S3C2410X 的 24 个外部中断有几种中断触发方式, EXTINTn 配置外部中断的触发类型是电平触发、边沿触发及触发的极性。具体配置参考数据手册。

(7) 外部中断屏蔽寄存器 (EINTMASK)。EINTMASK 描述如表 7-11 所示。

表 7-11 EINTMASK 描述

寄存器	地址	读/写	描述	复位值
EINTMASK	0x560000A4	R/W	外部中断屏蔽标志	0x00FFFFFF0

EXTMASK[23:4]分别对应外部中断 23~4, 等于 1, 对应的中断被屏蔽; 等于 0, 允许外部中断。EXTMASK[3:0]保留。

(8) 外部中断挂起寄存器 (EINTPND)。EINTPND 描述如表 7-12 所示。

表 7-12 EINTPND 描述

寄存器	地址	读/写	描述	复位值
EINTPND	0x560000A8	R/W	0: 未发出中断请求 1: 中断源发出中断请求	0x00000000

中断挂起寄存器 INTPND 共有 32 位, 前 4 位保留(因为 EINT0~EINT3 对应的挂起位在寄存器 INTPND 中), 4~23 位对应着一个中断源, 当中断请求被响应的时候, 相应的位会被设置为 1。在中断服务子程序中可以通过判断 EINTPND 来判断哪个中断在提起申请。

7.9.2 S3C2410X 中断处理程序实例

下面列举一个中断实例, 目的是为了读者更好地掌握 S3C2410X 中断的原理及中断程序的编写。内容是利用 S3C2410X 的外部中断 0 和外部中断 11 实现两个按键功能。

1. 电路原理

本实验选择的是外部中断 EXTINT0 和 EXTINT11。中断的产生分别来自按钮 SB1202 和 SB1203, 当按钮按下时, EXTINT0 (对应管脚 GPF0) 或 EXTINT11 (对应管脚 GPG3) 和地连接, 输入低电平, 从而

向 CPU 发出中断请求。当 CPU 受理中断后，进入相应的中断服务程序，通过超级终端的主窗口显示当前进入的中断号。电路原理图如图 7-9 所示。

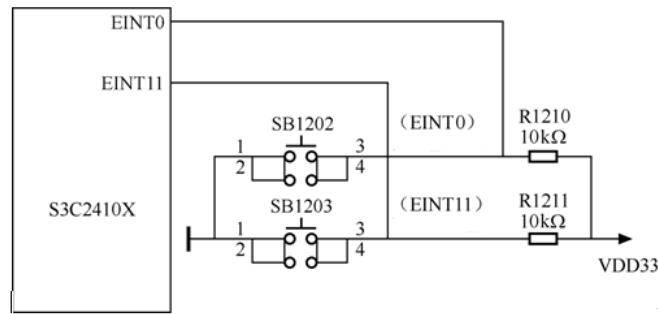


图 7-9 S3C2410X 中断实验电路图

2. 寄存器设置

(1) 配置 GPF0、GPG3 管脚为中断属性。

① 将 GPFCON 的[1:0]设置为“10”，从而将 GPF0 配置为 EINT0 属性。

② 将 GPGCON 的[7:6]设置为“10”，从而将 GPG3 配置为 EINT11 属性。

(2) 配置 EINT0、EINT11 为下降沿触发方式。

① 将 EXTINT0 的[2:0]设置为“010”。

② 将 EXTINT1 的[14:12]设置为“010”。

(3) 初始化 EINTPEND、SRCPND、INTPND 挂起寄存器中对应外部中断 0 和 11 的位。

① 设置 EINTPEND 的[11]为 0。

② 设置 SRCPND 的[0]和[5]为 0。

③ 设置 INTPND 的[0]和[5]为 0。

(4) 打开 EINTMASK 和 INTMSK 中对应外部中断 0 和 11 的屏蔽位。

① 设置 EINTMASK 的[11]为 0。

② 设置 INTMSK 的[0]和[5]为 0。

按上面的步骤配置完寄存器后，就可以响应中断了。当然还需要正确注册中断处理函数。中断处理函数详见程序编写部分。

3. 程序编写

程序在例程目录 MDK\uart_test 子目录下，工程名称是 Uart_test.Uv2，其中串口测试程序的核心代码如下。

(1) 相关寄存器定义：

例程 MDK\common\inc\2410addr.h

```
#define rGPFCON    (*(volatile unsigned *)0x56000050) //端口 F 的控制寄存器
#define rGPGCON    (*(volatile unsigned *)0x56000060) //端口 G 的控制寄存器
#define rEXTINT0    (*(volatile unsigned *)0x56000088) //外部中断控制寄存器 0
#define rEXTINT1    (*(volatile unsigned *)0x5600008c) //外部中断控制寄存器 1
#define rEINTMASK  (*(volatile unsigned *)0x560000a4) //外部中断屏蔽寄存器
#define rEINTPEND  (*(volatile unsigned *)0x560000a8) //外部中断挂起寄存器
#define rINTMSK    (*(volatile unsigned *)0x4a000008) //中断屏蔽寄存器
#define rSRCPND    (*(volatile unsigned *)0x4a000000) //源挂起寄存器
#define rINTPND    (*(volatile unsigned *)0x4a000010) //中断挂起寄存器
```

(2) 中断初始化：

```
void int_init(void)
{
    rGPFCON=(rGPFCON & ~(3<<0))|(0x2<<0); //将 GPF0 配置为 EINT0
    rGPGCON=(rGPGCON & ~(3<<6))|(0x2<<6); //将 GPG3 配置为 EINT11
    pISR_EINT0=(UINT32T)int0_int; //注册中断处理函数
    pISR_EINT8_23 = (UINT32T)int11_int; //注册外部中断 11 处理函数，EINT8~23 共用此
    //函数需要在处理函数中加入对中断源的判断
```

```

rEINTPEND = 0xfffff; //清除所有外部中断挂起状态
rSRCPND = BIT_EINT0 | BIT_EINT8_23; //清除源的挂起状态
rINTPND = BIT_EINT0 | BIT_EINT8_23; //清除挂起状态
rEXTINT0 = (rEXTINT0 & ~(7<<0)) | (0x2<<0); //EINT0 下降沿触发
rEXTINT1 = (rEXTINT1 & ~(7<<12)) | (0x2<<12); //EINT11 下降沿触发
rEINTMASK &= ~(1<<11); //打开外部中断 11
rINTMSK &= ~(BIT_EINT0 | BIT_EINT8_23); //打开 INTMSK 中的中断 0 和外部中断 8~23
}

```

(3) 中断处理函数:

```

void __irq int0_int(void) //外部中断 0 处理函数
{
    uart_printf(" EINT0 interrupt occurred.\n");
    ClearPending(BIT_EINT0); //清除中断源
}
void __irq int11_int(void) //外部中断 11 处理函数
{
    if(rEINTPEND==(1<<11)) //判断外部中断挂起寄存器, 确定是否是外部中断 11
    {
        uart_printf(" EINT11 interrupt occurred.\n");
        rEINTPEND=(1<<11); //清除中断挂起寄存器
    }
    ClearPending(BIT_EINT8_23);
}

#define ClearPending(bit) {\
    rSRCPND = bit;\
    rINTPND = rINTPND;} //precent write wrong data
} //清除中断源, 注意清除的顺序, 要从源头开始清除

```

7.10

本章小结



本章讲解了 ARM 处理器核的异常原理及 S3C2410X 的中断控制器的工作机制, 读者需要结合实验来加深对异常处理的理解。本书后面章节涉及的控制器的, 大都和中断处理有关系。

7.11

本章习题



1. ARM 有几种异常, 每种异常对应的处理器工作模式是什么?
2. 当执行 SWI 时, 会发生什么?
3. 利用 SWI 指令, 实现一个从用户态到系统态的系统调用过程。
4. 在你的硬件开发平台上选择一个可以产生中断的按键, 编写程序实现中断处理。

联系方式

集团官网: www.hqyj.com

嵌入式学院: www.embedu.org

移动互联网学院: www.3g-edu.org

企业学院: www.farsight.com.cn

物联网学院: www.topsight.cn

研发中心: dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路银海大厦 A 座 8 层，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

华清远见