华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

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做良心教育、做专业教育，更要做受人尊敬的职业教育。

《ARM 系列处理器应用技术完全手册》

作者：华清远见

专业始于专注 卓识源于远见

第 3 章 ARM 微处理器的编程模型

专业始于专注 卓识源于远见

3.1 数据类型

3.1.1 ARM 的基本数据类型

ARM 采用的是 32 位架构，ARM 的基本数据类型有以下 3 种。

- Byte：字节，8bit。
- Halfword：半字，16bit（半字必须于 2 字节边界对齐）。
- Word：字，32bit（字必须于 4 字节边界对齐）。

存储器可以看作是序号为 0～$2^{32}-1$ 的线性字节阵列。图 3.1 所示为 ARM 存储器的组织结构。

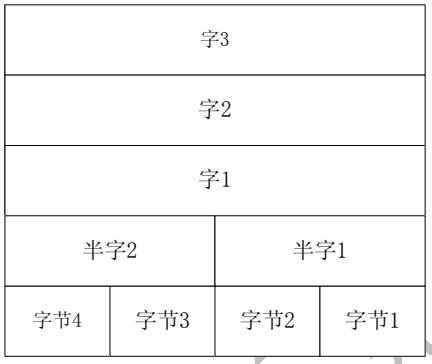

图 3.1　ARM 存储器组织结构

图 3.1 所示为存储器的一小片区域，其中每一个字节都有惟一的地址。字节可以占用任一位置，图中给出了几个例子。长度为 1 个字的数据项占用一组 4 字节的位置，该位置开始于 4 的倍数的字节地址（地址最末两位为 00）。图 3.1 中包含了 3 个这样的例子。半字占有两个字节的位置，该位置开始于偶数字节地址（地址最末一位为 0）。

注意

① ARM 系统结构 v4 以上版本支持以上 3 种数据类型，v4 以前版本仅支持字节和字。

② 当将这些数据类型中的任意一种声明成 unsigned 类型时，N 位数据值表示范围为 $0 \sim 2^n-1$ 的非负数，通常使用二进制格式。

③ 当将这些数据类型的任意一种声明成 signed 类型时，N 位数据值表示范围为 $-2^{n-1} \sim 2^{n-1}-1$ 的整数，使用二进制的补码格式。

④ 所有数据类型指令的操作数都是字类型的，如“ADD r1，r0，＃0x1”中的操作数“0x1”就是以字类型数据处理的。

⑤ Load/Store 数据传输指令可以从存储器存取传输数据，这些数据可以是字节、半字、字。加载时自动进行字节或半字的零扩展或符号扩展。对应的指令分别为 LDR/BSTRB（字节操作）、LDRH/STRH（半字操作）、LDR/STR（字操作）。详见后面的指令参考。

⑥ ARM 指令编译后是 4 个字节（与字边界对齐）。Thumb 指令编译后是 2 个字节（与半字边界对齐）。

3.1.2 浮点数据类型

浮点运算使用在 ARM 硬件指令集中未定义的数据类型。

尽管如此，但 ARM 公司在协处理器指令空间定义了一系列浮点指令。通常这些指令全部可以通过未定义指令异常（此异常收集所有硬件协处理器不接受的协处理器指令）在软件中实现，但是其中的一小部分也可以由浮点运算协处理器 FPA10 以硬件方式实现。

另外，ARM 公司还提供了用 C 语言编写的浮点库作为 ARM 浮点指令集的替代方法（Thumb 代码只能使用浮点指令集）。该库支持 IEEE 标准的单精度和双精度格式。C 编译器有一个关键字标志来选择这个历程。它产生的代码与软件仿真（通过避免中断、译码和浮点指令仿真）相比既快又紧凑。

3.1.3 存储器大/小端

从软件角度看，内存相对于一个大的字节数组，其中每个数组元素（字节）都是可寻址的。

ARM 支持大端模式（big-endian）和小端模式（little-endian）两种内存模式。
图 3.2 和图 3.3 分别显示了内存的大端模式和小端模式。

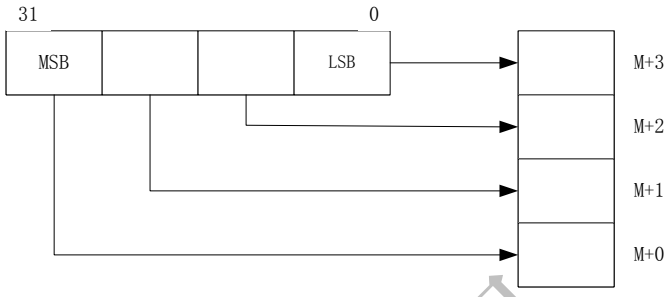

图 3.2 大端模式

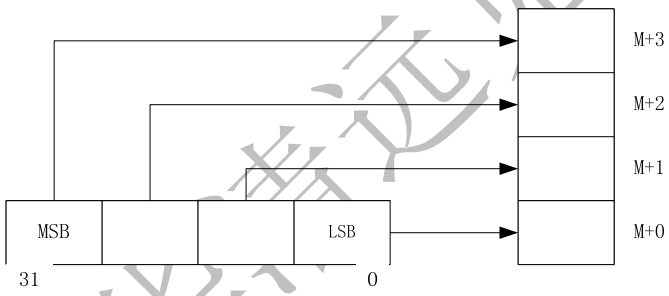

图 3.3 小端模式

下面的例子显示了使用内存大/小端（big/little endian）的存取格式。
【例 3.1】
程序执行前：

```
r0=0x11223344
```

执行指令：

```
r1=0x100
STR r0，[r1]
LDRB r2，[r1]
```

执行后：

```
小端模式下：r2=0x44
大端模式下：r2=0x11
```

上面的例子向我们提示了一个潜在的编程隐患。在大端模式下，一个字的高地址放的是数据的低位，而在小端模式下，数据的低位放在内存中的低地址。要小心对待存储器中一个字内字节的顺序。

3.2 处理器工作模式

ARM 处理器共有 7 种工作模式，如表 3.1 所示。

表 3.1　　ARM 处理器的工作模式

处理器工作模式	简　写	描　　述
用户模式（User）	usr	正常程序执行模式，大部分任务执行在这种模式下
快速中断模式（FIQ）	fiq	当一个高优先级（fast）中断产生时将会进入这种模式，一般用于高速数据传输和通道处理
外部中断模式（IRQ）	irq	当一个低优先级（normal）中断产生时将会进入这种模式，一般用于通常的中断处理
特权模式（Supervisor）	svc	当复位或软中断指令执行时进入这种模式，是一种供操作系统使用的保护模式
数据访问中止模式（Abort）	abt	当存取异常时将会进入这种模式，用于虚拟存储或存储保护
未定义指令中止模式（Undef）	und	当执行未定义指令时进入这种模式，有时用于通过软件仿真协处理器硬件的工作方式
系统模式（System）	sys	使用和 User 模式相同寄存器集的模式，用于运行特权级操作系统任务

除用户模式外的其他 6 种处理器模式称为特权模式（Privileged Modes）。在这些模式下，程序可以访问所有的系统资源，也可以任意地进行处理器模式切换。其中的 5 种又称为异常模式，分别为：

- FIQ(Fast Interrupt reQuest)；
- IRQ(Interrupt request)；
- 管理（Supervisor）；
- 中止（Abort）；
- 未定义（Undefined）。

处理器模式可以通过软件控制进行切换，也可以通过外部中断或异常处理过程进行切换。

大多数的用户程序运行在用户模式下。当处理器工作在用户模式时，应用程序不能够访问受操作系统保护的一些系统资源，应用程序也不能直接进行处理器模式切换。当需要进行处理器模式切换时，应用程序可以产生异常处理，在异常处理过程中进行处理器模式切换。这种体系结构可以使操作系统控制整个系统资源的使用。

当应用程序发生异常中断时，处理器进入相应的异常模式。在每一种异常模式中都有一组专用寄存器以供相应的异常处理程序使用，这样就可以保证在进入异常模式时用户模式下的寄存器（保存程序运行状态）不被破坏。

系统模式，不能有任何异常进入。仅 ARM 体系结构 v4 及以上版本有该模式。它和用户模式具有完全相同的寄存器。但是系统模式属于特权模式，可以访问所有的系统资源，也可以直接进行处理器模式切换，它主要供操作系统任务使用。通常操作系统的任务需要访问所有的系统资源，同时该任务仍然使用用户模式的寄存器组而不是异常模式下相应的寄存器组，这样可以保证当异常中断发生时任务状态不被破坏。

3.3　ARM 寄存器组织

ARM 处理器有 37 个 32 位长的寄存器。

- 1 个用作 PC（Program Counter）。
- 1 个用作 CPSR(Current Program Status Register)。
- 5 个用作 SPSR（Saved Program Status Registers）。
- 30 个用作通用寄存器。

注意　以上 37 个寄存器中，1 个 CPSR 和 5 个 SPSR 通称为状态寄存器，虽然这些寄存器是 32 位的，但目前只使用了其中的 12 位。除了这 6 个状态寄存器外，其余的 31 个寄存器又称为通用寄存器。

ARM 处理器共有 7 种不同的处理器模式，在每一种处理器模式中有一组相应的寄存器组。表 3.2 显示了 ARM 的寄存器组织概要。

表 3.2　　寄存器组织概要

User	FIQ	IRQ	SVC	Undef	Abort
R0	User mode R0～R7,R15,and CPSR	User mode R0～R12,R15 and CPSR	User mode R0～R12,R15 and CPSR	User mode R0～R12,R15 and CPSR	User mode R0～R12,R15 and CPSR
R1					
R2					
R3					
R4					
R5					
R6					
R7					
R8	R8				
R9	R9				
R10	R10				

续表

User	FIQ	IRQ	SVC	Undef	Abort
R11	R11				
R12	R12				
R13(SP)	R13(SP)	R13	R13	R13	R13
R14(LR)	R14(LR)	R14	R14	R14	R14
R15(PC)					
CPSR					
	SPSR	SPSR	SPSR	SPSR	SPSR

注意　System 模式使用和 User 模式相同的寄存器集

当前处理器的模式决定着哪组寄存器可操作，任何模式都可以存取。

- 相应的 r0～r12。
- 相应的 r13（the stack pointer, sp）和 r14（the link register, lr）。
- 相应的 r15（the program counter, pc）。
- 相应的 CPSR（current program status register, cpsr）。

特权模式（除 System 模式）还可以存取。

- 相应的 SPSR（saved program status register）。

3.3.1　通用寄存器

通用寄存器根据其分组与否和使用目的分为以下 3 类。

- 未分组寄存器（The unbanked registers），包括 r0～r7。
- 分组寄存器（The banked register），包括 r8～r14。
- 程序计数器（Program Counter），即 r15。

1．未分组寄存器

未分组寄存器包括 r0～r7。顾名思义，在所有处理器模式下对于每一个未分组寄存器来说，指的都是同一个物理寄存器。未分组寄存器没有被系统用于特殊的用途，任何可采用通用寄存器的应用场合都可以使用未分组寄存器。但由于其通用性，在异常中断所引起的处理器模式切换时，其使用的是相同的物理寄存器，所以也就很容易使寄存器中的数据被破坏。

2. 分组寄存器

r8～r14 是分组寄存器，它们每一个访问的物理寄存器取决于当前的处理器模式。

对于这些分组寄存器 r8～r12 来说，每个寄存器对应两个不同的物理寄存器。一组用于除 FIQ 模式外的所有处理器模式，而另一组则专门用于 FIQ 模式。这样的结构设计有利于加快 FIQ 的处理速度。不同模式下寄存器的使用，要使用寄存器名后缀加以区分，例如，当使用 FIQ 模式下的寄存器时，寄存器 r8 和寄存器 r9 分别记做 r8_fiq, r9_fiq；当使用用户模式下的寄存器时，寄存器 r8 和 r9 分别记做 r8_usr, r9_usr 等。

在 ARM 体系结构中，r8～r12 没有任何指定的其他的用途，所以当 FIQ 中断到达时，不用保存这些通用寄存器，也就是说 FIQ 处理程序可以不必执行保存和恢复中断现场的指令，从而可以使中断处理过程非常迅速。所以 FIQ 模式常被用来处理一些时间紧急的任务，如 DMA 处理。

对于分组寄存器 r13 和 r14 来说，每个寄存器对应 6 个不同的物理寄存器。其中的一个是用户模式和系统模式公用的，而另外 5 个分别用于 5 种异常模式。访问时需要指定它们的模式。名字形式如下：

- r13_<mode>
- r14_<mode>

其中<mode>可以是以下几种模式之一：usr、svc、abt、und、irp 及 fiq。

r13 寄存器在 ARM 中常用作堆栈指针，称为 SP。当然，这只是一种习惯用法，并没有任何指令强制性的使用 r13 作为堆栈指针，用户完全可以使用其他寄存器作为堆栈指针。而在 Thumb 指令集中，有一些指令强制性的将 r13 作为堆栈指针，如堆栈操作指令。

每一种异常模式拥有自己的 r13。异常处理程序负责初始化自己的 r13，使其指向该异常模式专用的栈地址。在异常处理程序入口处，将用到的其他寄存器的值保存在堆栈中，返回时，重新将这些值加载到寄存器。通过这种保护程序现场的方法，异常不会破坏被其中断的程序现场。

寄存器 r14 又被称为连接寄存器（Link Register，LR），在 ARM 体系结构中具有下面两种特殊的作用。

（1）每一种处理器模式用自己的 r14 存放当前子程序的返回地址。当通过 BL 或 BLX 指令调用子程序时，r14 被设置成该子程序的返回地址。在子程序返回时，把 r14 的值复制到程序计数器 PC。典型的做法是使用下列两种方法之一。

- 执行下面任何一条指令。

```
MOV  PC, LR
BX  LR
```

- 在子程序入口处使用下面的指令将 PC 保存到堆栈中。

```
STMFD  SP!, {<register>,LR}
```

在子程序返回时，使用如下相应的配套指令返回。

```
LDMFD  SP!, {<register>,PC}
```

（2）当异常中断发生时，该异常模式特定的物理寄存器 r14 被设置成该异常模式的返回地址，对于有些模式 r14 的值可能与返回地址有一个常数的偏移量（如数据异常使用 SUB PC, LR,#8 返回）。具体的返回方式与上面的子程序返回方式基本相同，但使用的指令稍微有些不同，以保证当异常出现时正在执行的程序的状态被完整保存。

R14 也可以被用作通用寄存器使用。

注意

当嵌套中断被允许时（即异常可重入），r13 和 r14 的使用要特别小心。例如，在用户模式下一个 IRQ 中断发生，这时两种模式分别使用不同的 r13 和 r14，换句话讲，用户模式使用 r13_usr 和 r14_usr，而 IRQ 模式使用 r13_irq 和 r14_irq，这样不会造成寄存器使用冲突。但是，当程序运行在 IRQ 模式下，又有 IRQ 中断进入，此时，第二级中断使用 r13_irq 和 r14_irq，冲掉了第一级 IRQ 的堆栈指针和返回地址，导致程序异常。

解决办法是在第二级中断发生前，将第一级中断用到的寄存器压栈。

3.3.2 程序计数器 r15

程序计算器 r15 又被记为 PC。它有时可以被和 r0－r14 一样用作通用寄存器，但很多特殊的指令在使用 r15 时有些限制。当违反了这些指令的使用限制时，指令的执行结果是不可预知的。

程序计数器在下面两种情况下用于特殊的目的。

- 读程序计数器。
- 写程序计数器。

1．程序计数器读操作

由于 ARM 的流水线机制，指令读出的 r15 的值是指令地址加上 8 个字节。由于 ARM 指令始终是字对齐的，所以读出的结果位[1：0]始终是 0（但在 Thumb 状态下，指令为 2 字节对齐，bit[0]=0）。

读 PC 主要用于快速地对临近的指令或数据进行位置无关寻址，包括程序中的位置无关分支。

需要注意的是，当使用指令 STR 或 STM 对 r15 进行保存时，保存的可能是当前指令地址加 8 或当前指令地址加 12。到底是哪种方式，取决于芯片的具体设计方式。当然，在同一个芯片中，要么采用当前指令地址加 8，要么采用当前指令地址加 12，不可能有些指令采用当前地址加 8，有些采用当前地址加 12。程序开发人员应尽量避免使用 STR 或 STM 指令来对 r15 进行操作。当不可避免要使用这种方式时，可以先通过一小段程序来确定所使用的芯片是使用哪种方式实现的。例如：

```
SUB  R1,PC,#4          ;r1 中存放 STR 指令地址
STR  PC,[R0]           ;将 PC=STR 地址+offset 保存到 r0 中
LDR  R0,[R0]
SUB  R0,R0,R1          ;offset=PC－STR 地址
```

2．程序计数器写操作

当指令向 r15 写入地址数据时，如果指令成功返回，它将使程序跳转到该地址执行。由于 ARM 指令是字对齐的，写入 r15 的地址值应满足 bit[1：0]=0b00，具体的规则根据 ARM 版本的不同也有所不同：

- 对于 ARM 版本 3 以及更低的版本，写入 r15 的地址值 bit[1：0]被忽略，即写入 r15 的地址值将与 0xFFFFFFFC 做与操作。
- 对于 ARM 版本 4 以及更高的版本，程序必须保证写入 r15 寄存器的地址值的 bit[1：0]为 0b00，否则将会产生不可预知的结果。

对于 Thumb 指令集来说，指令是半字对齐的。处理器将忽略 bit[0]，即写入 r15 寄存器的值在写入前要先和 0XFFFFFFFE 做与操作。

有些指令对 r15 的操作有特殊的要求。比如，指令 BX 利用 bit[0]来确定需要跳转到的子程序是 ARM 状态还是 Thumb 状态。

注意

这种读取 PC 值和写入 PC 值的不对称操作需要特别注意。

3.3.3 程序状态寄存器

当前程序状态寄存器 CPSR（Current Program Status Register）可以在任何处理器模式下被访问，它包含下列内容。

- ALU（Arithmetic Logic Unit）状态标志的备份。
- 当前的处理器模式。
- 中断使能标志。
- 设置处理器的状态（只在 4T 架构）。

每一种处理器模式下都有一个专用的物理寄存器作备份程序状态寄存器 SPSR（Saved Program Status Register）。当特定的异常中断发生时，这个物理寄存器负责存放当前程序状态寄存器的内容。当异常处理程序返回时，再将其内容恢复到当前程序状态寄存器。

注意 由于用户模式和系统模式不属于异常中断模式，所以它们没有 SPSR。当在用户模式或系统模式中访问 SPSR，将会产生不可预知的结果。

CPSR 寄存器（和保存它的 SPSR 寄存器）中的位分配如图 3.4 所示。

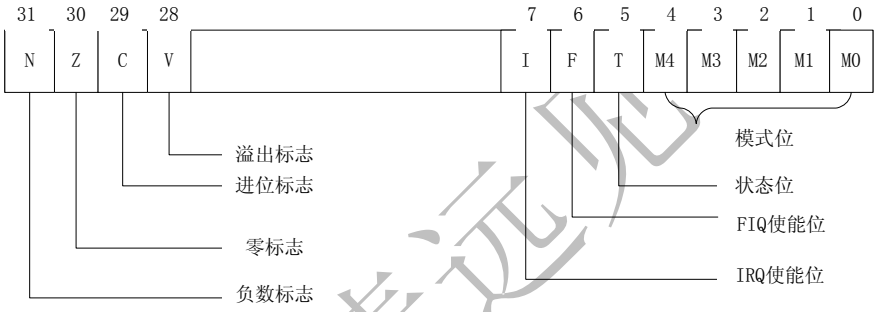

图 3.4 程序状态寄存器格式

1．标志位

N（Negative）、Z（Zero）、C（Carry）和 V（oVerflow）通称为条件标志位。这些条件标志位会根据程序中的算术或逻辑指令的执行结果进行修改，而且这些条件标志位可由大多数指令检测以决定指令是否执行。

在 ARM 4T 架构中，所有的 ARM 指令都可以条件执行，而 Thumb 指令却不能。

各条件标志位的具体含义如下。

- N

本位设置成当前指令运行结果的 bit[31]的值。当两个由补码表示的有符号整数运算时，N=1 表示运算的结果为负数；N=0 表示结果为正数或零。

- Z

Z=1 表示运算的结果为零，Z=0 表示运算的结果不为零。

注意 对于 CMP 指令，Z=1 表示进行比较的两个数相等。

- C

下面分 4 种情况讨论 C 的设置方法。

① 在加法指令中（包括比较指令 CMN），当结果产生了进位，则 C=1，表示无符号数运算发生上溢出；其他情况下 C=0。

② 在减法指令中（包括比较指令 CMP），当运算中发生错位（即无符号数运算发生下溢出），则 C=0,；其他情况下 C=1。

③ 对于在操作数中包含移位操作的运算指令（非加/减法指令），C 被设置成被移位寄存器最后移出去的位。

④ 对于其他非加/减法运算指令，C 的值通常不受影响。

- V

下面分两种情况讨论 V 的设置方法。

① 对于加/减运算指令，当操作数和运算结果都是以二进制的补码表示的带符号的数时，V=1 表示符号位溢出。

② 对于非加/减法指令，通常不改变标志位 V 的值（具体可参照 ARM 指令手册）。

尽管以上 C 和 V 的定义看起来颇为复杂，但使用时在大多数情况下用一个简单的条件测试指令即可，不需要程序员计算出条件码的精确值即可得到需要的结果。

注意

下面两种情况会对 CPSR 的条件标志位产生影响。

1. 比较指令（CMN、CMP、TEQ、TST）。
2. 目的寄存器不是 r15 的算术逻辑运算和数据传输指令。这些指令可以通过在指令末尾加标志“S”来通知处理器指令的执行结果影响标志位。

【例 3.2】

使用 SUBS 指令从寄存器 r1 中减去常量 1，然后把结果写回到 r1，其中 CPSR 的 Z 位将受到影响。

指令执行前：

```
CPSR 中 Z=0
r1=0x00000001

SUBS  r1，r1，#1
```

SUB 指令执行结束后：

```
r1=0x0
CPSR 中 Z=1
```

目的寄存器是 r15 的带“位设置”的算术和逻辑运算指令，也可以将 SPSR 的值复制到 CPSR 中，这种操作主要用于从异常中断程序中返回。

用 MSR 指令向 CPSR/SPSR 写进新值。

目的寄存器位 r15 的 MRC 协处理器指令通过这条指令可以将协处理器产生的条件标志位的值传送到 ARM 处理器。

在中断返回时，使用 LDR 指令的变种指令可以将 SPSR 的值复制到 CPSR 中。

2．Q 标志位

在带 DSP 指令扩展的 ARM v5 及更高版本中，bit[27]被指定用于指示增强的 DAP 指令是否发生了溢出，因此也就被称为 Q 标志位。同样，在 SPSR 中 bit[27]也被称为 Q 标志位，用于在异常中断发生时保存和恢复 CPSR 中的 Q 标志位。

在 ARM v5 以前的版本及 ARM v5 的非 E 系列处理器中，Q 标志位没有被定义。属于待扩展的位。

3．控制位

CPSR 的低 8 位（I、F、T 及 M[4：0]）统称为控制位。当异常发生时，这些位的值将发生相应的变化。另外，如果在特权模式下，也可以通过软件编程来修改这些位的值。

① 中断禁止位

I=1，IRQ 被禁止。
F=1，FIQ 被禁止。
② 状态控制位
T 位是处理器的状态控制位。
T=0，处理器处于 ARM 状态（即正在执行 32 位的 ARM 指令）。
T=1，处理器处于 Thumb 状态（即正在执行 16 位的 Thumb 指令）。
当然，T 位只有在 T 系列的 ARM 处理器上才有效，在非 T 系列的 ARM 版本中，T 位将始终为 0。
③ 模式控制位
M[4：0]作为位模式控制位，这些位的组合确定了处理器处于哪种状态。表 3.3 列出了其具体含义。只有表中列出的组合是有效的，其他组合无效。

表 3.3　　状态控制位 M[4：0]

M[4：0]	处理器模式	可以访问的寄存器
0b10000	User	PC，r14～r0，CPSR
0b10001	FIQ	PC，r14_fiq～r8_fiq，r7～r0，CPSR，SPSR_fiq
0b10010	IRQ	PC，r14_irq～r13_irq，r12～r0，CPSR，SPSR_irq
0b10011	Supervisor	PC，r14_svc～r13_svc，r12～r0，CPSR，SPSR_svc
0b10111	Abort	PC，r14_abt～r13_abt，r12～r0，CPSR，SPSR_abt
0b11011	Undefined	PC，r14_und～r13_und，r12～r0，CPSR，SPSR_und
0b11111	System	PC，r14～r0，CPSR(ARM v4 及更高版本)

注意 由于用户模式（User）和系统模式（System）是非异常模式，所以没有单独的 SPSR 保存程序状态字。在用户模式或系统模式下，读 SPSR 将返回一个不可预知的值，而写 SPSR 将被忽略。

3.4 异常中断处理

异常或中断是用户程序中最基本的一种执行流程和形态。这部分主要对 ARM 架构下的异常中断做详细说明。
ARM 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(SWI)和未定义指令异常（Undefined instruction）。

注意 在 ARM 文档中，使用术语 Exception 来描述异常。Exception 主要是从处理器被动接受异常的角度出发，而 Interrupt 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。

3.4.1 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 就要利用这一特性。

表 3.4 列出了 ARM 的 7 种异常。

表 3.4　　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 3.5 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
```

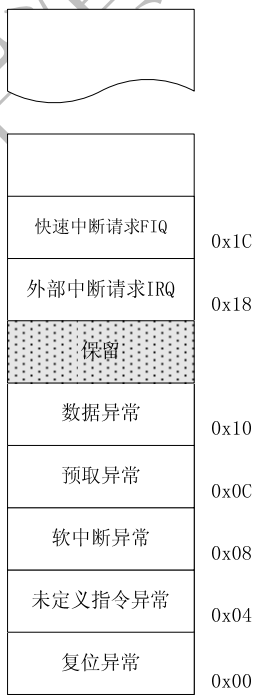

图 3.5　异常处理向量表

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。

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下面两种情况下。

- 系统上电。
- 系统复位。

当复位异常时，系统执行下列伪操作。

```
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。

- 设置异常中断向量表。
- 初始化数据栈和寄存器。
- 初始化存储系统，如系统中的 MMU 等。
- 初始化关键的 I/O 设备。
- 使能中断。
- 处理器切换到合适的模式。
- 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。若协处理器没有相应，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，可以同过 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，系统执行下列的伪操作。

```
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

预取指令异常使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
```

3.4.2 异常优先级

每一种异常按表 3.5 中设置的优先级得到处理。

表 3.5 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	中断请求
	5	预取指令异常
	6	软件中断
最低	7	未定义指令

异常可以同时发生，处理器按表 3.5 的优先级顺序处理异常。例如，复位异常的优先级最高，处理器上电时发生复位异常。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。

优先级最低的 2 种异常是软件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 SWI 和未定义指令异常有相同的优先级。

3.4.3 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。表 3.6 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 用户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 3.6 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

3.4.4 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里面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，省下一条跳转指令，如图 3.6 所示。

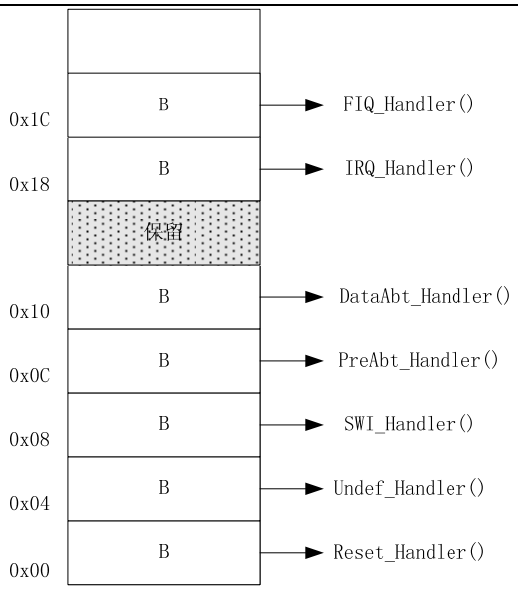

图 3.6　异常处理向量表

但跳转指令 B 的跳转范围为±32MB，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）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）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±4KB 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。

3.4.5　从异常处理程序中返回

当一个异常处理返回时，一共有 3 件事情需要处理：通用寄存器的恢复、状态寄存器的恢复以及 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以及 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 的拷贝，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 3.7 所示。

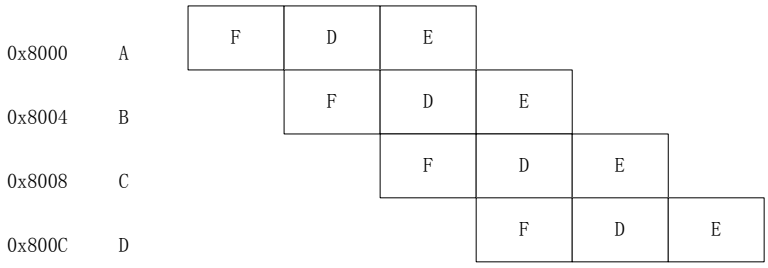

图 3.7　3 级流水线示例

在 ARM 架构里，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 3.5 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，进入异常相应后，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，所以只有直接把 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 处进入数据异常的相应，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，表 3.7 总结了各异常和返回地址的关系

表 3.7　　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那条指令
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

3.4.6　在应用程序中安装异常处理程序

1．使用汇编语言安装异常处理程序

如果系统启动不依赖于 Debug 或 Debug monitor 软件，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，那么就系统将中断向量表从 ROM 复制 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 words)
    STMIA R8!,{r0-r7}
    LDMIA R9!,{r0-r7}                ;复制由伪操作 DCD 定义的地址
    STMIA R8!,{r0-r7}
```

注意 可以使用 Scatter 文件定义加载向量表的地址，这样上述代码的拷贝工作由 C 库函数完成。

2．使用 C 语言安装异常处理程序

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))
{
/* diagnose the fault */
prinf ("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);
}
```

3.4.7 FIQ 和 IRQ 中断处理函数的设计

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 3.8 所示。

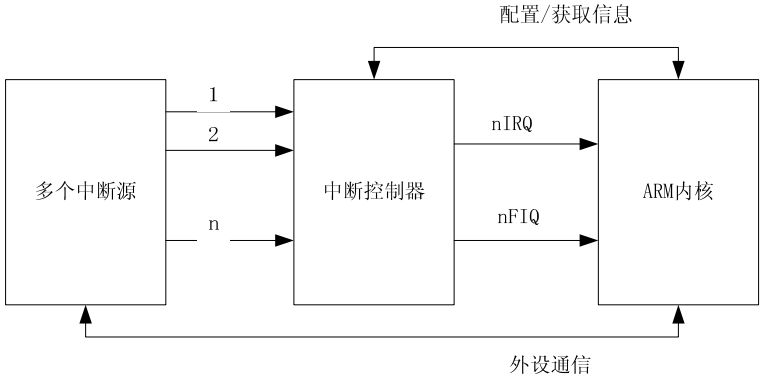

图 3.8　中断系统

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 3.9 所示。

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 3.9 更好的流程控制方法，如图 3.10 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。

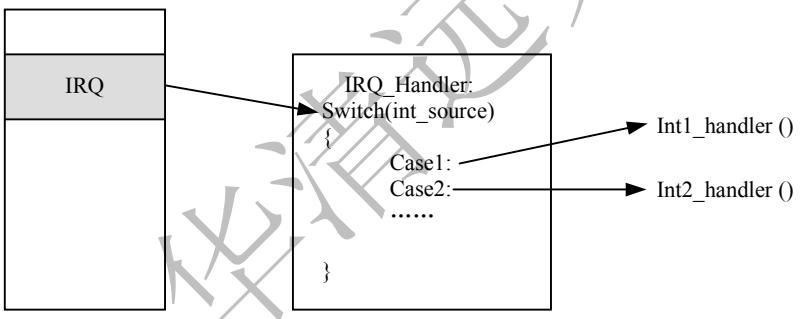

图 3.9　软件控制中断分支

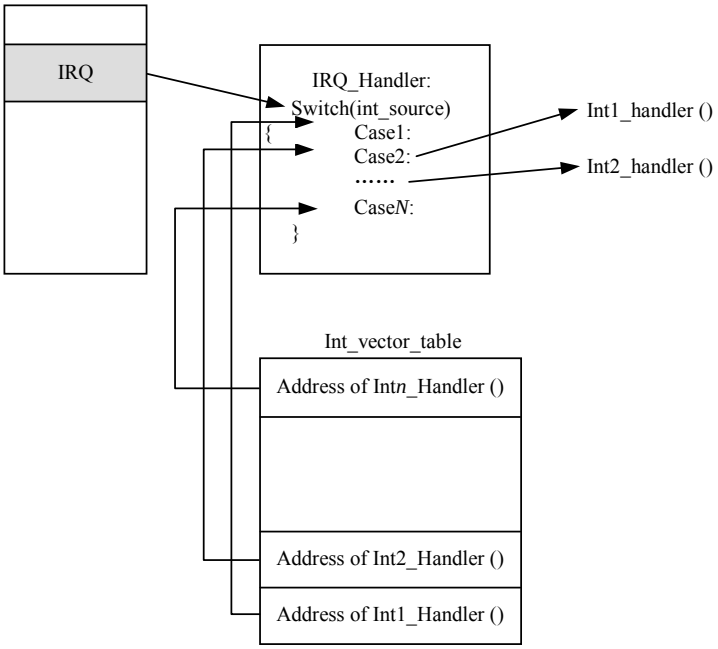

图 3.10　灵活的软件分支设计

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();
}
/*清楚中断标志*/
*(base+1) = 0;
}
```

使用 armcc 编译出的汇编代码如下。

```
IRQHandler PROC
    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
    ENDP
```

如果不使用_irq 子程序声明关键字，编译出的汇编代码如下。

```
IRQHandler PROC
    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}
    ENDP
```

3．可重入中断设计

在缺省情况下，ARM 中断是不可重入的。因为一旦进入异常响应状态，ARM 自动关闭中断使能。如果在异常处理过程中，简单地打开中断使能而发生中断嵌套时，显然新的异常处理将破坏原来的中断现场而导致出错。但有时需要中断必须是可重入的，因此要通过程序设计来解决这个问题。其中有两个关键问题。

① 新中断使能之前，必须要保护好前一个中断的现场信息。比如 LR_irq 和 SPSR_irq 等，这一点比较容易做的。

② 中断处理过程中对 BL 进行保护。

在中断处理函数中发生函数调用 BL 是很常见的，假设有下面一种情况。

```
IRQ_Handler:
    ……
    BL    Foo
    ADD
```

其中，

```
Foo:
    STMFD  SP!,{r0-r3，LR}
    ……
    LDMFD  SP!{r0-r3，PC}
```

上述程序，在 IRQ 处理函数 IRQ_Handler()中调用了函数 Foo()。若是在 IRQ_Handler()里面中断可重入的话，可能发生问题，考察下面的情况：当新的中断请求恰好在“BL　Foo”指令执行完成后发生。这时候 LR_irq 寄存器（因在 IRQ 模式下，所以是 LR_irq）的值将调整为 BL 指令的下一条指令（ADD）地址，使其能从 Foo()正确返回；但是因为这时候发生了中断请求，接下来要进行新中断的响应，处理器在新中断响应过程中也要进行 LR_irq 保存。这次对 LR_irq 的操作发生了冲突，当新中断返回后，往下执行 STMFD 指令，这时候压栈的 LR 已不是原来的 ADD 指令地址，从而使子程序 Foo()无法正确返回。

这个问题无法通过增加额外的现场保护指令来解决。一个办法就是在重新使能中断之前改变处理器模式，也就是使上面程序的“BL　Foo”指令不要运行在 IRQ 模式下。这样当新的中断发生时，就不会造成 LR 寄存器的冲突。考虑 ARM 的所有运行模式，采用 SYSTEM 模式是比较合适的，因为它是特权模式，不是 IRQ 模式，与中断响应无关。

下面的例子显示了标准的 IRQ/FIQ 异常中断处理程序。

```
    PRESERVE8
    AREA INTERRUPT, CODE, READONLY
    IMPORT C_irq_handler
IRQ
```

```
    SUB lr, lr, #4              ;跳转返回地址
STMFD sp!, {lr}                 ;保存返回地址
MRS r14, SPSR                   ;读取 SPSR
STMFD sp!, {r12, r14}           ;保存寄存器
; 清除中断源
MSR CPSR_c, #0x1F               ;切换到 SYSTEM 模式,
STMFD sp!, {r0-r3, lr}          ;保存 lr_USR 和其他使用到的寄存器
BL C_irq_handler                ;跳转到 C 中断处理函数
LDMFD sp!, {r0-r3, lr}          ;恢复用户模式寄存器
MSR CPSR_c, #0x92               ;切换回 irq 模式
LDMFD sp!, {r12, r14}
MSR SPSR_cf, r14
LDMFD sp!, {pc}^
END
```

3.4.8 SWI 异常处理函数的设计

本小节主要介绍编写 SWI 处理程序时需要注意的几个问题，包括下面内容。

- 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。
- 从应用程序中动态调用 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 3.11 所示。

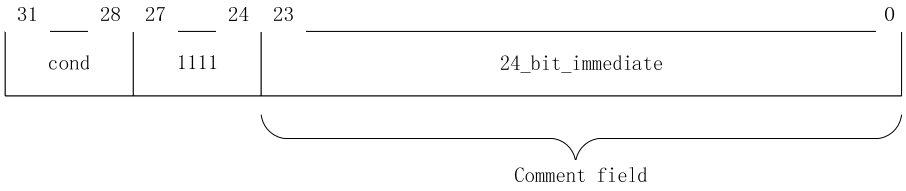

图 3.11　SWI 指令编码格式

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 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．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 语言函数示例如下所示。

```
void C_SWI_handler (unsigned number)
{
    switch (number)
     {
      case 0 : /* SWI number 0 code */
      break;
      case 1 : /* SWI number 1 code */
      break;
      ...
      default : /* Unknown SWI - report error */
```

```
    }
}
```

另外，如果需要传递的参数多于 1 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，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．在特权模式下使用 SWI 异常处理

在特权模式下使用 SWI 异常处理，和 IRQ/FIQ 中断嵌套基本类似。当执行 SWI 指令后，处理器执行下面操作。
① 处理器进入特权模式。
② 将程序状态字内容 CPSR 保存到 SPSR_svc。
③ 返回地址放入 LR_svc。
如果处理器已经处于特权模式，再发生 SWI 异常，则 LR_svc 和 SPSR_svc 寄存器的值将丢失。
所以在特权模式下，调用 SWI 软中断异常，必须先将 LR_svc 和 SPSR_svc 寄存器的值压栈保护。下面的例子显示了一个可以在特权模式下调用的 SWI 处理函数。

```
    AREA SWI_Area, CODE, READONLY
    PRESERVE8
    EXPORT SWI_Handler
    IMPORT C_SWI_Handler
                T_bit EQU 0x20
SWI_Handler
    STMFD sp!,{r0-r3,r12,lr}        ;寄存器压栈保护
    MOV r1, sp                      ;堆栈指针放 r1 作为参数传递.
    MRS r0, spsr                    ;读取 spsr.
    STMFD sp!, {r0, r3}             ;将 spsr 压栈保护
    ;
    ;
    LDR r0,[lr,#-4]                 ;计算 SWI 指令地址.
    BIC r0,r0,#0xFF000000           ;读取 SWI 中断向量号.
    ; r0 存放中断向量号
    ; r1 堆栈指针
    BL C_SWI_Handler                ;调用 C 程序的 SWI 处理函数.
    LDMFD sp!, {r0, r3}             ;从堆栈中读取 spsr.
    MSR spsr_cf, r0                 ;恢复 spcr
    LDMFD sp!, {r0-r3,r12,pc}^      ;恢复其他寄存器并返回.
    END
```

5．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

从汇编语言程序中调用 SWI，只要遵循 AAPCS 标准即可。调用前，设定所有必须的值并发出相关的 SWI。例如：

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

注意 SWI 指令和其他所以 ARM 指令一样，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。如果有两到四个返回值，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。下面的例子显示了对编号为 0x0、0x1、0x2 和 0x3 的 SWI 软中断的调用。其中，SWI0x0 和 SWI0x1 传递两个整型参数并返回一个单一结果；SWI0x2 传递 4 个参数并返回一个单一结果；而 SWI0x3 传递 4 个参数并通过结构体返回 4 个结果。

```
#include <stdio.h>
#include "swi.h"
unsigned *swi_vec = (unsigned *)0x08;
extern void SWI_Handler(void);
int main( void )
{
int result1, result2;
struct four_results res_3;
Install_Handler( (unsigned) SWI_Handler, swi_vec );
printf("result1 = multiply_two(2,4) = %d\n", result1 = multiply_two(2,4));
printf("result2 = multiply_two(3,6) = %d\n", result2 = multiply_two(3,6));
printf("add_two( result1, result2 ) = %d\n", add_two( result1, result2 ));
printf("add_multiply_two(2,4,3,6) = %d\n", add_multiply_two(2,4,3,6));
res_3 = many_operations( 12, 4, 3, 1 );
printf("res_3.a = %d\n", res_3.a );
printf("res_3.b = %d\n", res_3.b );
printf("res_3.c = %d\n", res_3.c );
printf("res_3.d = %d\n", res_3.d );
return 0;
}
__swi(0) int multiply_two(int, int);
__swi(1) int add_two(int, int);
```

```
__swi(2) int add_multiply_two(int, int, int, int);
struct four_results
{
int a;
int b;
int c;
int d;
};
__swi(3) __value_in_regs struct four_results many_operations(int, int, int, int);
```

（3）应用程序中动态调用 SWI

在某些情形下，需要调用直到运行时才会知道其编号的 SWI。例如，当有很多相关操作可在同一目标上执行，并且每一个操作都有其自己的 SWI 时，就会发生这种情况。在此情况下，上一小节的方法不适用。解决的方法有两种。

- 在运行时得到 SWI 功能号，然后构造出相应的 SWI 指令的编码，将该编码保存在某个存储单元中，将 PC 指针指向该单元，执行指令。
- 使用一个通用的 SWI 异常中断处理程序，将运行时需要调用的 SWI 功能号作为参数传递给该通用的 SWI 异常处理程序，通用的 SWI 异常中断处理程序根据参数值调用相应的 SWI 处理程序完成需要的操作。

通过汇编语言可以实现第二种解决办法：通过寄存器（通常为 r0 或 r12）传递所需要的操作数，这样可以重新编写 SWI 处理程序，对相应寄存器中的值进行处理。

但有些情况下，为了节省程序开销，需要直接使用 SWI 中断号对程序调用。例如，操作系统可能会使用单一的一条 SWI 指令并用寄存器来传递所需运算的编号。这使得其他 SWI 空间可用于特定应用程序的 SWI。在一个特定的应用程序中，如果从指令中提取 SWI 编号的开销太大，就可使用这个方法。ARM（0x123456）和 Thumb（0xAB）半主机方式的 SWI 就是这样实现的。

下面的例子显示了如何使用_swi 将 C 函数调用映射到半主机方式的 SWI。

```
#ifdef __thumb
/* Thumb 状态的 Semihosting 软中断处理*/
#define SemiSWI 0xAB
#else
/* ARM 状态下的 Semihosting 的软中断处理*/
#define SemiSWI 0x123456
#endif
/* 使用 Semihosting 软中断输出一个字符*/
__swi(SemiSWI) void Semihosting(unsigned op, char *c);
#define WriteC(c) Semihosting (0x3,c)
void write_a_character(int ch)
{
char tempch = ch;
WriteC( &tempch );
}
```

编译程序含有一个机制，用以支持使用 r12 来传递所需运算的值。根据 AAPCS 标准，r12 为 IP 寄存器，并且专用于函数调用。其他时间内可将其用作暂存寄存器。如前面所述，通用 SWI 参数和返回值通过 r0～r3 寄存器传递。而 r12 可用于传递通用 SWI 调用的中断功能编号。

下面的例子显示了通用 SWI 的 C 语言程序框架。

```
__swi_indirect(0x80)
unsigned SWI_ManipulateObject(unsigned operationNumber,
unsigned object,unsigned parameter);
unsigned DoSelectedManipulation(unsigned object,
```

```
    unsigned parameter, unsigned operation)
    {
    return SWI_ManipulateObject(operation, object, parameter);
    }
```

生成的汇编代码如下。

```
    DoSelectedManipulation PROC
    STMFD sp!,{r3,lr}
    MOV r12,r2
    SWI 0x80
    LDMFD sp!,{r3,pc}
ENDP
```

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