第3章 体系结构

3.1 概述

Cortex-M0 使用的 ARMv6-M 体系结构涵盖了诸多方面,《ARMv6-M 体系结构参考手册》(ARMv6-M Architecture Reference Manual)(参考文档[3])中有体系结构的具体细节。在 ARM 官方网站上注册后,就可以下载到该文档。使用 Cortex-M0 微控制器,用户无须关注体系结构的相关细节。而使用 C 语言开发 Cortex-M0 设备,只需了解存储器映射、外设操作方法、异常处理机制和系统模型的一部分。

本章将会涉及系统模型、存储器映射以及异常中断的基本概况。由于大部分人都使用 C 语言操作 Cortex-M0 处理器,因此,底层的系统模型对程序代码是不可见的。即便这样,了解一下体系结构的细节内容也是很有必要的,它可以在设备调试以及本书内容的理解等方面提供帮助。

3.2 系统模型

3.2.1 操作模式和状态

Cortex-M0 处理器包含两种操作模式和两种状态(见图 3.1)。

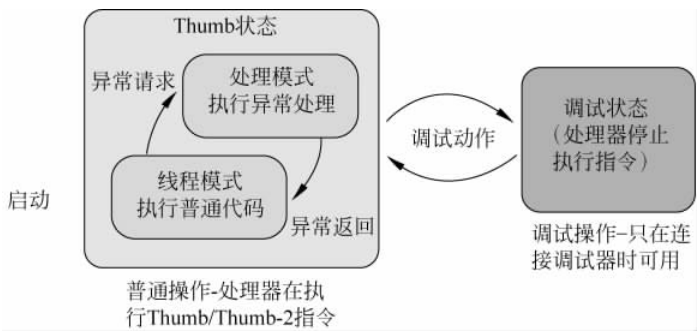

图 3.1 Cortex-M0 处理器的模式和状态

处理器在运行程序时处于 Thumb 状态(Thumb state),在这种状态下,处理器可以处在线程模式(Thread mode),也可以处在处理模式(Handler mode)。在 ARMv6-M 的体系结构里,线程模式和处理模式的系统模型几乎完全一样,唯一的不同在于,线程模式通过配置 CONTROL 特殊寄存器,可以使用影子栈指针(见图 3.7,本章稍后介绍)。栈指针的选择问题,在本章后面的内容中会有所体现。

调试状态(Debug state)仅用于调试操作,暂停处理器内核后,指令将不再执行,这时也就进入了调试状态。在这种状态下,调试器可以读取甚至改变内核寄存器的值。在 Thumb 状态或是调试状态下,调试器都可以访问系统存储器空间。

处理器上电以后,默认处在 Thumb 状态和线程模式。

3.2.2 寄存器和特殊寄存器

数据解析和控制处理的过程中,需要处理器内核的多个寄存器参与。如果需要处理存储器中的数据,这些数据就需要首先被加载到处理器内核的寄存器(寄存器组中的某个),处理完后,如果有必要,它们还将会被送到存储器中,这种方式通常被称为"加载-存储架构"(load-store architecture)。只要在寄存器组中拥有足够的寄存器,这种机制是很容易使用的,而且可以用 C 语言实现。C 编译器很容易地就能将 C 代码编译成机器码,并使其具有良好的性能。使用内部寄存器实现短期的数据存储时,存储器操作的次数也会减少。

Cortex-M0 的寄存器组中,包含了 13 个 32 位的通用目的寄存器,以及多个特殊寄存器(见图 3.2)。

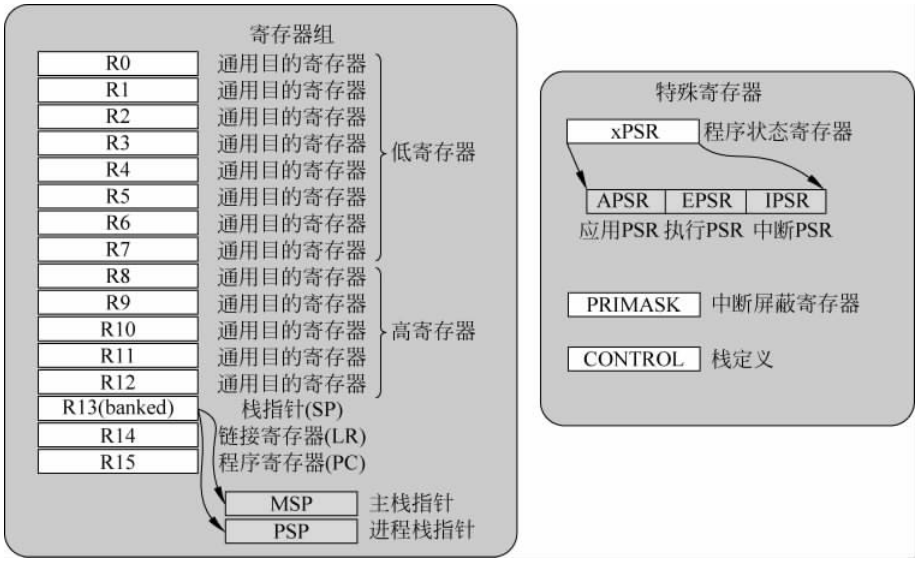

图 3.2 Cortex-M0 的寄存器

寄存器组包含16个32位的寄存器,它们大都是通用的,有些是具有特殊用途的,关于这些寄存器的细节如下所述。

3.2.3 R0-R12

R0-R12为通用寄存器,由于16位的Thumb指令集在空间上的限制,许多Thumb指令只能操作R0到R7,它们也被称作低寄存器(low registers),而像MOV之类的一些指令则可以使用全部的寄存器。在使用ARM汇编器之类的ARM开发工具操作这些寄存器时,寄存器的写法可以用大写(如R0),也可以用小写(如r0)。R0～R12寄存器的初始值未定义。

3.2.4 R13,栈指针(SP)

R13为栈指针,用于对栈空间的存取操作(通过PUSH和POP指令),Cortex-M0在不同的物理位置上存在两个栈指针。主栈指针(MSP,在ARM文档中也被称作SP_main)为上电后的默认指针,用于异常处理。另外一个称作进程栈指针(PSP,在ARM文档也被称作SP_process),只能用在线程模式(Thread mode)。可以通过配置CONTROL寄存器,选择使用哪个栈指针,CONTROL寄存器之后将会介绍。

使用ARM开发工具编写程序代码时,程序员可以使用“R13”或者“SP”来操作栈指针。在ARM处理器中,由于寄存器是32位的,故PUSH和POP指令永远是32位操作,而且存取的地址须是32位字对齐的。在处理器上电流程中,中断向量表的头4字节会被取出,然后填充到MSP,作为MSP的初始值。PSP的初始值未定义。

PSP一般是没有必要使用的,对于许多应用,系统完全依赖MSP。使用操作系统的设计通常会用到PSP,这是因为操作系统内核的栈空间和线程级的应用程序的栈空间是相互独立的。

3.2.5 R14,链接寄存器(LR)

R14为链接寄存器,用于存储子程序或者函数调用的返回地址。子程序或函数执行完毕,存储在LR中的返回地址将被装载到程序计数器(PC)中,以便调用程序可以继续执行。当发生异常中断时,LR会提供一个特定值,用于中断返回机制。在使用ARM开发工具编写代码时,我们可以通过R14或LR访问链接寄存器,而且不区分大小写(可以书写为r14或lr)。

尽管Cortex-M0处理器的函数返回地址始终是偶数(最低位为0,因为最小的指令都是16位,也就是半字对齐的),LR的0位却是可读可写的。对于ARMv6-M体系结构,为了指明当前处于Thumb状态,一些指令需要函数地址的最低位为1。

3.2.6 R15,程序计数器(PC)

R15为程序计数器,并且可读可写。读操作返回当前正在执行的指令地址加上4(这是

由流水线的特性决定的),而写入 R15 会导致程序跳转执行(和函数调用不同,链接寄存器不会更新)。

在 ARM 汇编器中,可以用 R15 或 PC 来操作程序计数器,大小写均可(也可以写作 r15 或 pc)。Cortex-M0 处理器的指令地址须是半字(也就是 16 位)对齐的,这也就意味着 PC 寄存器的最低位必须始终为 0。不过,在使用跳转指令(BX 或 BLX)执行程序跳转时,PC 的最低位应该被置为 1,以表明目标分支处于 Thumb 程序区域。如果试图切换至 Cortex-M0 未知的 ARM 状态,错误异常中断就会被触发。

3.2.7 xPSR,组合程序状态寄存器

组合程序状态寄存器提供了程序执行信息和 ALU(算术逻辑单元)标志,该寄存器由三个程序状态寄存器(PSR)组成(见图 3.3):

- 应用程序状态寄存器(APSR);
- 中断程序状态寄存器(IPSR);
- 执行程序状态寄存器(EPSR)。

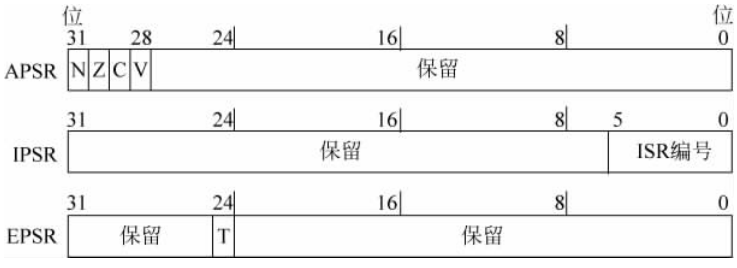

图 3.3 APSR、IPSR、EPSR

APSR 包含了 ALU 标志:N(负号标志)、Z(零标志)、C(进位或借位标志)和 V(溢出标志)。它们位于 APSR 的最高 4 位,一般用于控制条件跳转。

IPSR 中包含了当前正在执行的中断服务程序(ISR)编号,Cortex-M0 的每个异常中断都会有一个特定的中断编号(表示中断类型)。这对调试时识别当前中断非常有用,而且在多个中断共用一个中断处理的情况下,可以看出发生的是哪个中断。

Cortex-M0 的 EPSR 包含了 T 位,该位用以指示当前是否处于 Thumb 状态。由于 Cortex-M0 处理器只支持 Thumb 状态,故 T 位一般为 1。清除该位后,执行下一条指令会触发硬件异常中断。

这三个寄存器可以作为一个寄存器 xPSR 来访问(见图 3.4)。比如,发生中断时,xPSR 也会被自动压入栈中,从中断返回时,数据会自动恢复。在压栈和出栈的过程中,xPSR 始终被当做一个寄存器。

要实现对程序状态寄存器的直接访问,只能使用特殊寄存器操作指令。不过,APSR 寄存器中的值能够影响条件跳转,而且一些数据处理指令也可能会用到进位标志。

图 3.4 xPSR

3.2.8 应用程序状态寄存器(APSR)的行为

数据处理指令不仅能改变目标寄存器,也能影响 APSR,这在其他处理器架构中也被称作 ALU 状态标志。APSR 对条件跳转控制尤为重要,另外,寄存器中的 C 位(进位或借位)也会用在加法和减法操作中。

Cortex-M0 处理器的 APSR 有四位,表 3.1 中有对它们的描述。

表 3.1 Cortex-M0 处理器的 ALU 标志

标 志	描 述
N(第 31 位)	设置到执行指令结果的 31 位,为"1"时结果为负数(被解析为有符号整数时),设为"0"时,结果为整数或等 0
Z(第 30 位)	如果执行指令的结果为 0 置位,两个相同数值经比较后也会置"1"
C(第 29 位)	结果的进位,对于无符号加法,如果产生了无符号溢出,该位置"1";对于无符号减法,该位为借位输出状态取反
V(第 28 位)	结果溢出,对于有符号加法和减法,如果发生了有符号溢出,该位置"1"

ALU 标志的几个例子如表 3.2 所示。

表 3.2 ALU 标志示例

操 作	结果,标志
0x70000000 + 0x70000000	结果 = 0xE0000000,N=1,Z=0,C=0,V=1
0x90000000 + 0x90000000	结果 = 0x30000000,N=0,Z=0,C=1,V=1
0x80000000 + 0x80000000	结果 = 0x00000000,N=0,Z=1,C=1,V=1
0x00001234−0x00001000	结果 = 0x00000234,N=0,Z=0,C=1,V=0
0x00000004−0x00000005	结果 = 0xFFFFFFFF,N=1,Z=0,C=0,V=0
0xFFFFFFFF−0xFFFFFFFC	结果 = 0x00000003,N=0,Z=0,C=1,V=0
0x80000005−0x80000004	结果 = 0x00000001,N=0,Z=0,C=1,V=0
0x70000000−0xF0000000	结果 = 0x80000000,N=1,Z=0,C=0,V=1
0xA0000000−0xA0000000	结果 = 0x00000000,N=0,Z=1,C=1,V=0

在 Cortex-M0 中,几乎所有的数据处理指令都会更改 APSR,有些指令不会对 V 标志和 C 标志产生影响。例如,MULS(乘法)指令只会改变 N 标志和 Z 标志。

ALU 标志也可用于处理超过 32 位的数据,例如要实现 64 位的加法,可以将其拆为两个 32 位的加法。该处理过程可以描述如下:

```
//计算 Z = X + Y,并且 X、Y 和 Z 都是 64 位的
Z[31:0] = X[31:0] + Y[31:0];                    //计算低 32 位相加,进位标志会更新
Z[63:32] = X[63:32] + Y[63:32] + 进位;          //计算高 32 位相加
```

第 6 章中有用汇编指令实现 64 位加法的例子。

APSR 还经常用作跳转控制,相关内容将会在第 4 章中介绍,而且会涉及条件跳转指令的相关细节。

3.2.9 PRIMASK:中断屏蔽特殊寄存器

PRIMASK 仅有一位宽,被称作中断屏蔽寄存器(见图 3.5),置位后,除了不可屏蔽中断(NMI)和硬件错误异常外的其他中断都会被屏蔽掉。实际上,此时当前中断优先级被置为了 0,这也是可以配置的最高等级。

图 3.5 PRIMASK

要访问 PRIMASK 寄存器,可以通过特殊寄存器操作指令(MSR 和 MRS),也可以使用“改变处理器状态”指令(CPS)。在处理对时间敏感的应用时,需要操作 PRIMASK 寄存器。

3.2.10 CONTROL:特殊寄存器

就像前面所提到的,Cortex-M0 处理器具有两个栈指针。处理器模式决定了使用的栈指针,而处理器模式依赖于 CONTROL 寄存器的配置(见图 3.6)。

图 3.6 CONTROL

复位以后,系统默认使用主栈指针(MSP),在线程模式下,通过将 CONTROL 寄存器的第 1 位置 1,处理器也可以切换至使用进程栈指针(PSP)(前提是当前不是处在异常中断处理中)。在处理异常中断时(运行在处理模式下),系统只使用 MSP,CONTROL 寄存器读出值为 0。要改变 CONTROL 寄存器的值,应该在线程模式下操作,或者借助异常中断进入和返回机制。

为了同 Cortex-M3 处理器保持兼容,CONTROL 寄存器的第 0 位保留。在 Cortex-M3 中,第 0 位用于将处理器切换至用户模式(非特权模式),而这个特性在 Cortex-M0 处理器

中已不复存在。

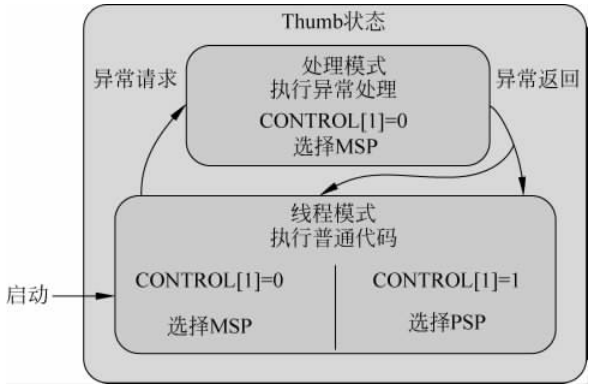

图 3.7 栈指针选择

3.3 存储器系统

Cortex-M0 处理器具有 4GB 的存储器地址空间(见图 3.8)。在体系结构上,存储器空间被分作一系列的区域,每个区域都有推荐的用途,以提高不同设备间的可移植性。

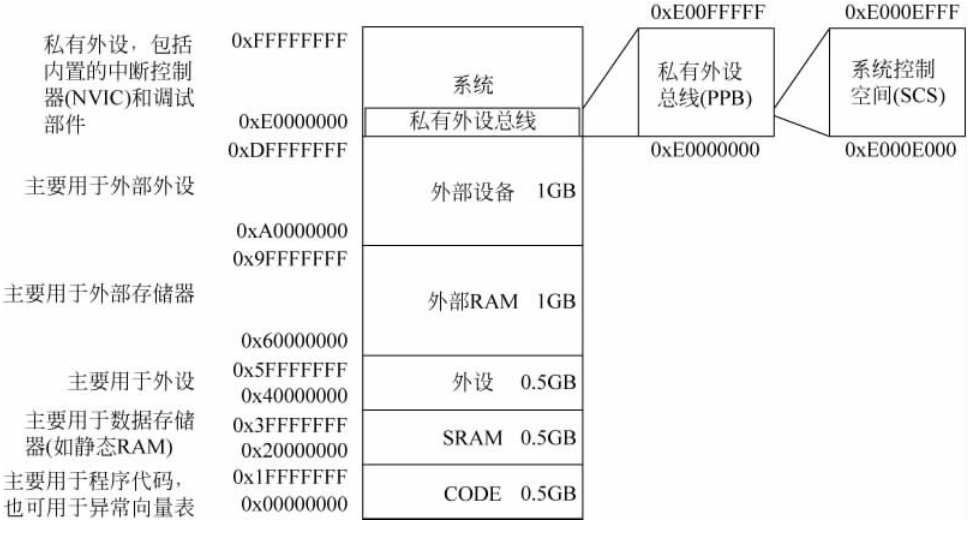

图 3.8 存储器映射

Cortex-M0 处理器内置了各种部件,例如 NVIC(嵌套向量中断控制器)和一些调试部件,它们都被映射到系统空间的固定地址上。因此所有基于 Cortex-M0 的设备在中断控制和调试方面,都具有相同的编程模型。这种处理有利于软件移植,也方便调试工具供应商为基于 Cortex-M0 的微控制器和片上系统(SoC)提供开发调试方案。

大多数情况下，连接到 Cortex-M0 的存储器都是 32 位的；当然，选择合适的硬件接口，Cortex-M0 处理器也可以配备其他数据宽度的存储器。Cortex-M0 的存储器系统支持各种大小的数据传输，包括字节(8 位)、半字(16 位)和字(32 位)。Cortex-M0 支持大端和小端操作，使用相应的配置即可选择，但已经成型的设计不能在两者间切换。

由于 Cortex-M0 存储器系统和外设由微控制器供应商和片上系统(SoC)设计者提供，因此基于 Cortex-M0 的产品的存储器就存在各种大小和类型。

3.4 栈空间操作

栈空间作为一种存储器使用机制，是先入先出的结构，在系统空间中用作临时数据存储。栈空间操作的关键点之一为栈指针寄存器，每次执行栈操作时，栈指针的内容自动调整。在 Cortex-M0 处理器中，栈指针为 R13，而且物理上存在着两个栈指针，但每次只会使用一个，这是由 CONTROL 寄存器以及处理器的状态决定的(见图 3.7)。

按照通常的说法，向栈中存储数据称为“压栈”(使用 PUSH 指令)，恢复数据则称作“出栈”(使用 POP 指令)。根据所使用架构的不同，有些处理器在向栈存入数据时地址会自动增加，而有些则会减小。Cortex-M0 处理器的栈操作基于“满递减”(full-descending)的栈模型，这就意味着栈指针始终指向栈空间的最后一个数据，在执行数据存储前(PUSH)，栈指针会首先减小。

PUSH 和 POP 通常用在函数或子程序的开始和结尾处。在函数开始执行时，PUSH 操作将寄存器的当前内容存入栈空间；执行结束前，POP 又将栈空间存储的数据恢复。一般说来，对每个寄存器的 PUSH 操作都应相应地进行 POP 操作。否则恢复的数据可能无法对应之前的寄存器，这样会导致无法预期的后果，比如栈溢出。

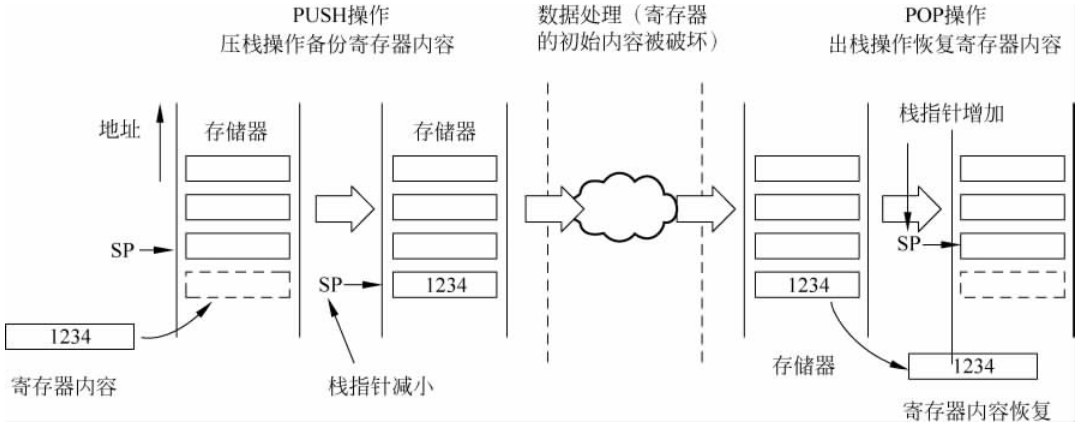

图 3.9 Cortex-M0 的 PUSH 和 POP 操作

每次出栈以及压栈操作的最小单位是4字节(32位),还可以使用一条指令实现对多个寄存器的压栈和出栈操作。Cortex-M0的栈空间被设计为字对齐的(地址值必须是4的倍数,比如0x0、0x4、0x8等),由于这个原因,栈指针的最低两位(BITS[1:0])在硬件上被置为了0,因此读出固定为0。

可以通过R13或SP访问R13或SP,根据处理器状态和CONTROL寄存器值的不同,访问的栈指针可以是主栈指针(MSP),也可以是进程栈指针(PSP)。许多简单的应用只会用到一个栈指针,一般默认是主栈指针(MSP),进程栈指针通常只用于嵌入式应用的操作系统(OS)。

表3.3 栈指针用法定义

处理器状态	CONTROL[1]=0(默认设置)	CONTROL[1]=1(OS已启动)
线程模式	使用MSP(R13为MSP)	使用PSP(R13为PSP)
处理模式	使用MSP(R13为MSP)	使用MSP(R13为MSP)

在一个具有操作系统的典型嵌入式应用中,操作系统内核使用MSP,而应用程序进程则使用PSP。这样就使得内核的栈空间与应用程序进程的栈空间相互独立,而操作系统的上下文切换也会非常迅速(在应用程序进程间相互切换)。即便操作系统内核只使用MSP作为其栈指针,它也可以通过特殊寄存器操作指令(MRS和MSR)访问PSP寄存器。

由于栈是向下生长的(满递减),内存的上边界通常会被用作栈指针的初始值。例如,如果内存区域为0x20000000~0x20007FFF,我们可以将栈指针的初始值设为0x20008000。在这种情况下,第一次压栈操作会将数据存至0x20007FFC开始的字中,这也是内存的最高4字节。

MSP的初始值位于程序空间的开头部分,这里也有中断向量表(相关内容将会在3.5节中介绍)。PSP的初始值则没有定义,它需要通过软件初始化。

3.5 异常和中断

异常会引起程序控制的变化。在异常发生时,处理器停止当前的任务,转而执行被称作异常处理的程序;异常处理完成后,还会继续执行刚才暂停的正常程序流程。异常分为很多种,中断只是其中的一种。Cortex-M0处理器最多支持32个外部中断(通常称作IRQ)和一个被称作不可屏蔽中断(NMI)的特殊中断,中断事件的异常处理通常被称作中断服务程序(ISR),中断一般由片上外设或者IO口的外部输入产生。Cortex-M0处理器上可用的中断数量不定,不同的微控制器的数量可能不一样。如果系统的外设较多,由于中断数目的限制,多个中断源可能会共用同一个中断连接。

除了NIM和IRQ,Cortex-M0处理器还支持许多系统异常,它们主要用于操作系统和错误处理(见表3.4)。

表 3.4 异常类型

异常类型	异常编号	描述
Reset	1	上电复位或系统复位
NMI	2	不可屏蔽中断，最高优先级且不能被禁止，用于高安全性的事件
Hard fault	3	用于错误处理，系统检测到错误后被激活
SVCall	11	请求管理调用，在执行 SVC 指令时被激活，主要用于操作系统
PendSV	14	可挂起服务(系统)调用
SysTick	15	系统节拍定时器异常，一般在 OS 中用作周期系统节拍异常，SysTick 定时器在 Cortex-M0 处理器中是可选的
IRQ0-IRQ31	16—47	中断，可来自外部也可来自片上外设

每一个异常都对应一个异常编号，这在包括 IPSR 在内的许多寄存器中都有所体现，而且这个异常编号还指明了异常向量的地址。需要注意的是，在设备驱动库中，异常编号和中断编号是相互独立的。系统异常使用负数定义，而中断则使用从 0～31 的正数定义。

复位是一类特殊的异常。如果发生复位的话，Cortex-M0 处理器将会退出主程序，并且在线程模式中执行复位处理(不必从处理模式返回到线程模式)。另外，数值为 1 的异常号在 IPSR 中是不可见的。

除了 NMI、硬件错误和复位，其他所有异常的优先级都是可编程的。NMI 和硬件错误的优先级都是固定的，并且比其他异常的都要高，第 8 章中有这方面的更多细节。

3.6 嵌套向量中断控制器(NVIC)

为了管理中断请求的优先级并且处理其他异常，Cortex-M0 处理器内置了嵌套向量中断控制器(NVIC)。NVIC 的一些可编程寄存器控制着中断管理功能，这些寄存器被映射到系统地址空间里，它们所处的区域称为系统控制空间(SCS)(见图 3.8)。

NVIC 具有以下特性：

- 灵活的中断管理；
- 支持嵌套中断；
- 向量化的异常入口；
- 中断屏蔽。

3.6.1 灵活的中断管理

Cortex-M0 处理器中，每一个外部中断都可以被使能或禁止，并且可以被设置为挂起状态或清除状态。处理器的中断可以是信号级的(在中断服务程序清除中断请求以前，外设的请求会一直保持)，也可以是脉冲形式的(最小一个时钟周期)，这样中断控制器可以处理任何中断源。

3.6.2 支持嵌套中断

Cortex-M0 处理器的任何中断都有一个固定或者可编程的中断优先级。当外部中断之类的异常发生时,NVIC 将该异常的优先级与当前级别相比较。如果新的优先级更高,当前任务就会被暂停,一些寄存器将会被压栈处理,然后处理器开始执行新异常的处理程序,这个过程也被称为“抢占”。高优先级的中断完成后,异常返回就会执行,处理器随后将自动进行出栈操作来恢复寄存器的值,并且继续运行之前的任务。这种机制在允许中断服务嵌套的同时,并没有带来软件开销。

3.6.3 向量化的异常入口

异常发生时,处理器需要定位对应异常处理程序的入口。按照传统的处理方式,就像 ARM7TDMI 等处理器那样,这个过程需要软件来实现。而 Cortex-M0 则会从存储器的向量表中,自动定位异常处理的入口。这样一来,从异常产生到异常处理执行的时间就缩短了。

3.6.4 中断屏蔽

Cortex-M0 中的 NVIC 通过 PRIMASK 特殊寄存器提供了一种中断屏蔽特性,NVIC 可以屏蔽掉除了硬件错误和 NMI 之外的所有异常。有些操作,比如时间敏感控制任务或实时多媒体解码任务,不应被打断,此时中断屏蔽就能表现出其作用了。

以上的 NVIC 特性,降低了 Cortex-M0 处理器的使用难度,提供了更优的反应时间,而且由于在 NVIC 硬件中处理异常,也减少了程序代码量。

3.7 系统控制块(SCB)

除了 NVIC,系统控制空间(SCS)中也包含了许多系统管理的寄存器,这些寄存器被称为系统控制块(SCB)。其中有些寄存器控制休眠模式和系统异常配置,另外还有个寄存器中包含了处理器的识别代码(调试器可以使用该代码识别处理器的类型)。

调试系统

尽管已经是 ARM 系列中最小的处理器,Cortex-M0 仍然具有多种调试特性。处理器内核实现的功能包括停止模式调试、单步、寄存器访问,另外还有单独的调试模块提供了断点单元(BPU)和数据监视点(DWT)单元。BPU 最多支持四个硬件断点,DWT 最多支持 2 个监视点。

为了能让调试器控制之前提到的调试部件并执行调试操作,Cortex-M0 处理器实现了调试接口单元。调试接口单元可以使用 JTAG 协议,也可以使用串行线调试(SWD)协议。微控制器供应商可以实现支持 JTAG 和 SWD 两种协议的调试接口,而大多数应用为了节

省引脚，只实现了一种接口。

串行线调试协议是 ARM 开发的新标准，它仅需两个信号线，在没有损失性能的前提下，却可以实现和 JTAG 相同的功能。

串行线调试接口与 JTAG 接口使用相同的引脚：串行时钟与 JTAG 的 TCK 信号复用，而串行线数据与 JTAG 的 TMS 信号复用（见图 3.10）。许多调试仿真器都可用于 ARM 微控制器，包括 ULINK2（Keil）和 JLink（SEGGER），而且它们都已经支持串行线调试接口。

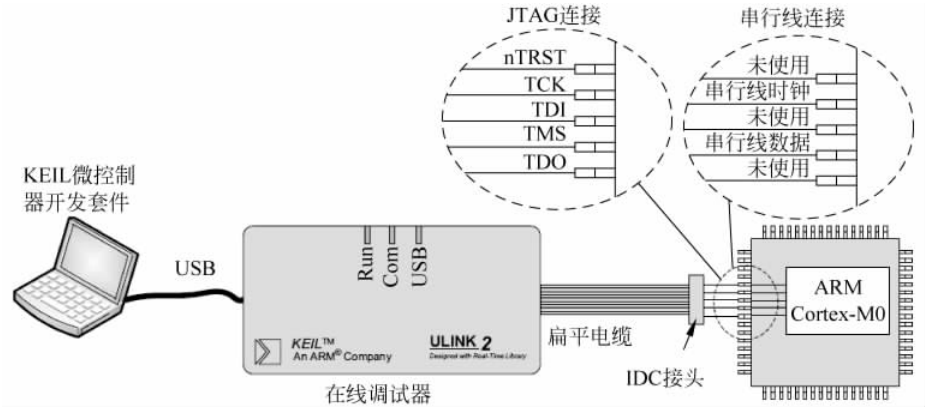

图 3.10　调试接口可以使用 JTAG 或 SWD

3.8　程序映像和启动流程

要理解 Cortex-M0 的启动流程，需要首先大致浏览一下程序映像。通常，Cortex-M0 处理器的程序映像是从地址 0x00000000 开始的。

程序映像的开始处为向量表（见图 3.11），其中包含了异常的起始地址（向量），每个中断向量的地址都等于“异常号×4”。例如，外部 IRQ＃0 的异常类型为 16，因此 IRQ＃0 的向量地址为 16×4＝0x40。这些向量的最低位都被置 1，表明异常处理执行时使用 Thumb 指令。向量表的大小由实际使用的中断个数决定。

向量表中还包含了主栈指针（MSP）的初始值，它存储在向量表的头四个字节。

复位时，处理器首先读取向量表前两个字（即前 8 字节），第一个字为 MSP 的初始值；第二个字为复位向量（见图 3.12），它表示程序执行的起始地址（复位处理）。

例如，如果启动代码位于地址 0x000000C0，就需要在复位向量处写入这个地址，并且将地址的最低位置为 1，以表明当前为 Thumb 代码。因此，地址 0x00000004 处的值被置为 0x000000C1（见图 3.13）。在取得复位向量值以后，处理器将开始从这个地址处执行程序代码。传统的 ARM 处理器（例如 ARM7TDMI）在这方面的处理是不同的，它们往往从地址 0x00000000 处开始执行程序；而且向量表中为跳转指令，Cortex-M 处理器中则是地址值。

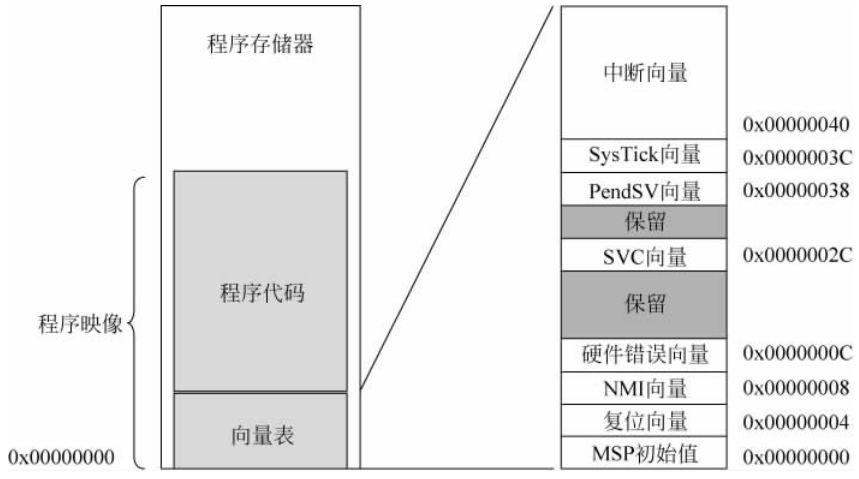

图 3.11　程序映像中的向量表

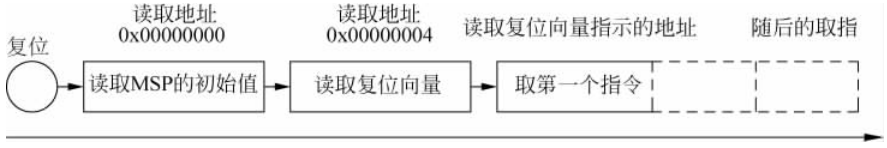

图 3.12　复位流程

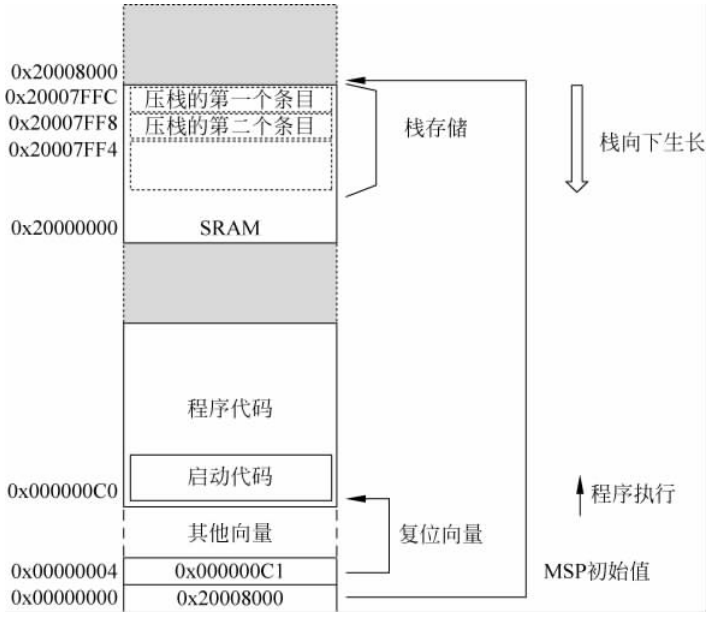

图 3.13　MSP 和 PC 初始化示例

复位流程也会初始化主栈指针(MSP),假定内存位于 0x2000000～0x20007FFF,可以将 0x20008000 写在地址 0x0000000 处,这样就实现了把主栈置于内存的顶部(见图 3.13)。

和 Cortex-M0 的固定地址栈指针初始化不同,传统的 ARM 处理器和其他许多微控制器都是通过软件初始化的方式来实现的。

如果要使用进程栈指针(PSP),在配置 CONTROL 寄存器切换栈指针前,必须首先通过软件代码将其初始化。因为复位流程只初始化 MSP,而非 PSP。

不同的软件开发工具在指定栈指针初始值、复位及异常向量时的方式不同,大多数的开发工具都会提供相应的例程,以供开发者参考。而且对于大部分编译工具,向量表都可以用 C 语言实现。