第二章 ARM Cortex-M4F 处理器

到目前为止，ARM 共推出 8 个版本的指令集体系结构版本。V4 版架构是 ARM7 和 ARM9 两个产品系列采用的体系结构；而 ARM11 则是 V6 版本的架构；今天的 ARM Cortex 则是 V7 版本的架构。

ARM Cortex 系列又分成 3 个子系列，分别是 Cortex A、Cortex M 和 Cortex R。Cortex A 系列面向高性能应用，能运行较大的操作系统，能提供多媒体交互，如智能手机，平板电脑。Cortex M 系列用于嵌入式控制器系统，能耗效率高、成本低，便于使用，如各种工业控制系统。Cortex R 系列针对实时高、性能要求高的应用，如汽车系统。如今已有 15 家芯片厂商购买了 Cortex M 系列的 CPU，在此基础上设计出自己的微控制器销售。

Cortex-M4 是一个 32 位处理器内核，其内部的数据总线、寄存器、存储器接口也都是 32 位的。Cortex -M4 采用三级流水线的哈佛结构，拥有独立的指令总线和数据总线，可以让取指与数据访问并行执行。Cortex-M4 提供一个可选的存储保护单元(MPU)，为操作系统提供特权权限访问。此外，Cortex-M4 还集成了一个高性能的中断控制器 NVIC（可嵌套向量中断控制器）。Cortex-M4 处理器如果还包含了单精度浮点运算单元(FPU)，则被称为 Cortex-M4F 处理器。

ARM Cortex-M4 采用了 16 位和 32 位指令并存的 Thumb-2 指令集，这样处理器就不用在 ARM 和 Thumb 两种状态间来回切换了。

1 Cortex-M4 处理器和基于 Cortex-M4 的 MCU

Cortex-M4 处理器是 MCU 的中央处理单元（CPU）。完整的基于 Cortex-M4 的 MCU 还需要很多其它组件。芯片供应商得到 Cortex-M4 处理器内核的使用授权后，它们就可以把 Cortex-M4 处理器用在自己的芯片设计中，添加存储器，外设，I/O 以及其它功能模块。不同厂家设计出的微控制器会有不同的配置，包括存储器的容量、外设类型等都各具特色。

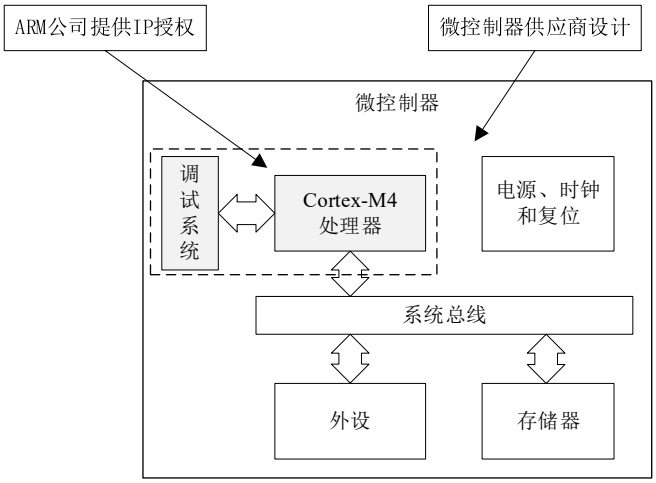

图 2-1 Cortex-M4 处理器内核和 MCU

2 Cortex-M4F 处理器结构

Cortex-M4F 处理器包括一个 Cortex-M4 处理器内核、私有外设、调试接口及多条总线接口。处理器内部的外设被称为私有外设，NVIC、MPU、SysTick 和调试组件都可以被称为私

有外设。

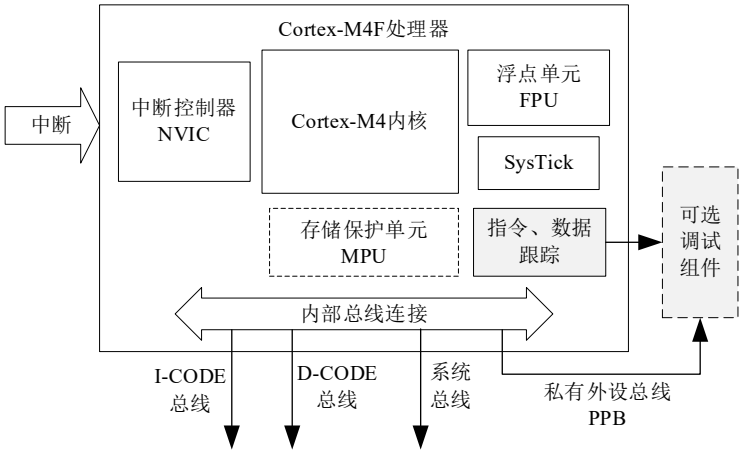

图 2-2 Cortex-M4F 处理器结构框图

1) SysTick

24 位的递减定时器，可被用作实时操作系统 (RTOS) 的节拍定时器，或者作为一个简单的定时器。

2) 嵌套式向量化中断控制器(NVIC)

一个嵌入的中断控制器，支持低延迟中断处理。

3) 存储器保护单元 (MPU)

Cortex-M4 有一个可选的存储器保护单元，用于定义不同存储区域的存储器访问权限（如只支持特权下的访问或全访问）和存储器属性（如可缓冲、可缓存）。配置 MPU 后，就可以对特权级访问和用户级访问存储器分别施加不同的访问限制。

操作系统可以使用 MPU 来保护操作系统内核使用的栈和存储区域，避免被用户程序读取和改写；MPU 支持 8 个可编程的存储器区域，可以给多个用户任务分配独立的存储区域，防止它们互相干扰；还可以将关键数据和外设保护起来，使其只能读取，或者不能被用户程序访问，以阻止因意外访问造成的破坏，使系统更加安全和可靠。MPU 默认为禁止。

4) 浮点单元 (FPU)

完全支持单精度浮点的加、减、乘、除、乘加以及平方根操作。它还可用于定点数和浮点数的相互转换。

5) 调试跟踪接口

Cortex-M4 提供一个完整的硬件调试方案，可以实现程序执行控制，包括停机(halting)、单步执行(stepping)、指令断点、数据监视点、寄存器和存储器访问、性能测试(profiling)以及各种跟踪机制。调试跟踪组件包括 Flash 补丁和断点单元(FPB)、指令追踪宏单元(ITM)、数据观察点和跟踪单元(DWT)、嵌入式跟踪宏单元(ETM)、跟踪端口接口单元(TPIU)和 ROM 表。

6) 总线接口

Cortex-M4 内部有若干个总线接口，以使处理器能同时取指和访问内存。

I-Code 总线和 D-Code 总线负责对代码存储区的访问，I-Code 总线用于取指和取异常向量；D-Code 总线用于取数据（如查表）和调试器访问等操作。

系统总线(System Bus)用于访问内存和外设，覆盖的区域包括 SRAM，片上外设，片外 RAM，片外扩展设备，以及系统级存储区的部分空间。

私有外设总线(PPB)负责一部分私有外设的访问，主要就是访问调试组件。

3 存储器映射

Cortex-M4 支持 32 位的存储空间，其地址范围可达 4GB。存储空间采用了统一编址，即指令存储器、数据存储器和外设都规划在一个地址空间内，而且被规定成若干相对固定的区域。

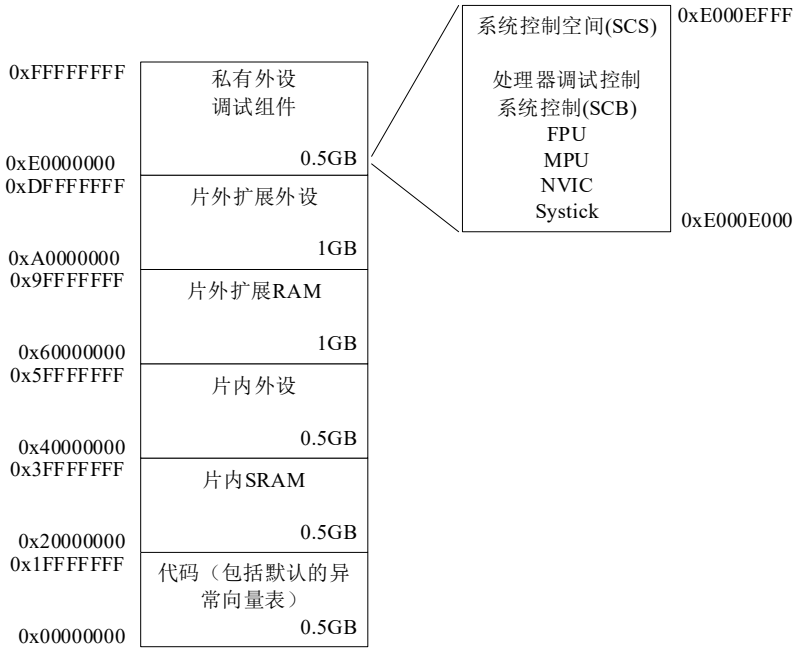

图 2-3 Cortex-M4 存储器映射

在更早的 ARM 架构中，存储器映射由微控制器厂商自己定义，而 Cortex-M4 采用了不同的方式，预先为不同类型的存储器和设备划定好了映射区域。微控制器厂商可以根据不同的应用需求，在相应的映射区域配置不同类型的外设和不同大小的存储器。

为存储空间划定一个“粗线条”的框架，其好处是为应用软件在不同型号的微控制器间移植提供了方便。例如：在地址最高的系统区，Cortex-M4 内核的私有外设(NVIC、MPU、SysTick 和 FPU)和调试组件的地址是固定的，不同型号的 MCU 都可以采用相同的代码对这些系统空间进行访问。

4 处理器模式和软件执行的权限级别

Cortex-M4 处理器具有两种处理器操作模式，还支持两级特权级别。

两种操作模式为：处理模式(handler mode)和线程模式(thread mode)。引入两个模式的本意，是用于区别普通状态程序的代码和异常服务程序的代码。

◆ 线程模式(Thread Mode)：执行普通程序代码时（没有执行异常服务程序），处理器在线程模式下。

◆ 处理模式（Handler Mode）：正在处理异常（正在执行异常服务程序时），处理器在处理模式，当处理器完成异常的处理后会返回到线程模式。

另外，Cortex-M4 有两级权限级别：特权级和用户级。

◆ 用户级：禁止对系统控制空间(SCS，0xE000E000—0xE000EFFF，包括 NVIC、SysTick、MPU 以及代码调试控制)大部分寄存器的访问。除此之外，还禁止使用 MRS／MSR 指令访问除了 APSR 之外的特殊功能寄存器。

◆ 特权级：在特权级下，软件可以使用所有的指令和访问所有的资源。

在线程模式下，软件执行可以处于特权级，也可以处于用户级。在处理模式下，软件执行一定是在特权级。线程模式一旦进入异常处理，则必然进入特权级。在刚刚上电复位时，处理器是处于线程模式+特权级下。

	特权级	用户级
异常处理的代码	处理模式	不可能
普通程序的代码	线程模式	线程模式

特权级可通过设置 CONTROL 寄存器来进入用户级。CONTROL 寄存器只有在特权级下才能访问，用户级下的代码不能自己切换到特权级。用户级的程序如想进入特权级，通常都是使用 “系统服务呼叫 (SVC)”指令来触发“SVC 异常”，在该异常服务函数中可以视具体情况而修改 CONTROL 寄存器，才能在返回到线程模式后进入特权级。

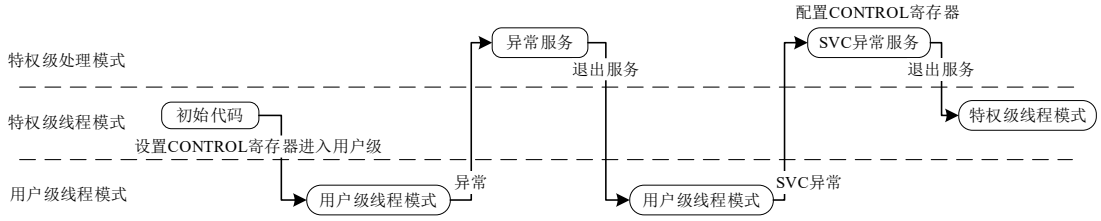

图 2-4 特权级和用户级转换

Cortex-M4 通过区分特权级和用户级为操作系统提供了支持。权限机制实现了一种安全模型，限制用户程序对系统要害配置的访问，防止由于其随意修改，而导致系统崩溃。权限机制可以和 MPU 配合使用，防止某个用户程序对操作系统的存储空间和其它用户存储空间进行访问，而造成信息泄露和其它更严重的破坏。

对于有嵌入式操作系统的 MCU 应用，一般让操作系统内核和异常处理工作在特权级，具备最高权限，能使用所有指令，能对所有区域进行访问。而对于用户程序，则只让其工作在用户级，禁止其对系统空间的访问。至于没有操作系统的简单应用程序，往往无须进入用户级，而一直处于特权级。

5 内核寄存器

Cortex-M4 通过其内核寄存器实现数据处理和相关控制。Cortex-M4 的内核寄存器如图 2-5 所示，其寄存器组共有 16 个寄存器，包括 13 个通用寄存器(R0-R12)，栈指针 SP、链接寄存器 LR、程序计数器 PC。除了寄存器组，处理器还有多个特殊寄存器（状态寄存器 PSR、控制寄存器(CONTROL)、优先级屏蔽寄存器(PRIMASK)、故障屏蔽寄存器(FAULTMASK)、基本优先级屏蔽寄存器(BASEPRI)。在 MCU 的数据手册（如 TM4C1231H6PGE 中文数据手册的第 2 章 Cortex-M4F 处理器的 2.3.4 节寄存器描述）中可以查询到所有内核寄存器的详细信息。

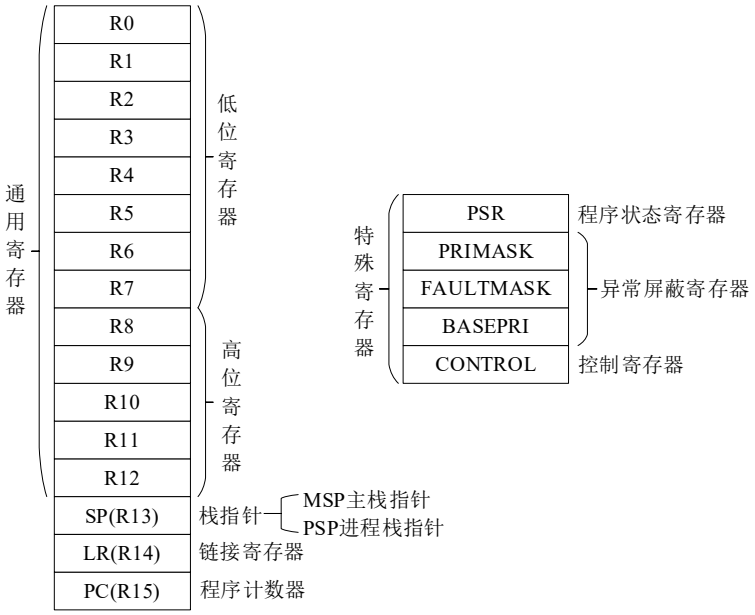

图 2-5 内核寄存器

1) 通用寄存器 R0～R12

通用寄存器在处理器运行过程中作为数据的缓存。R0~R7 被称为低位寄存器，许多 16 位指令只能访问低位寄存器。只有少数 16 位指令能访问高位寄存器(R8~R12)，32 位指令能访问所有通用寄存器。

2) 栈指针 SP(R13)

R13 为栈指针寄存器：栈指针是用于访问堆栈。Cortex-M4 物理上有两个堆栈指针: 主堆栈指针(MSP)和进程堆栈指针(PSP)，R13 在任何时刻只能是其中一个。在复位后或处于处理模式时，使用 MSP；而 PSP 只能用于线程模式。栈指针的选择由控制寄存器(CONTORL)来决定。

在没有操作系统的简单应用程序中，往往只使用 MSP，而没有必要使用 PSP。而在嵌入式实时操作系统中，我们为用户程序专门设置了一个堆栈，让它和操作系统内核的堆栈独立使用，以避免可能发生的错误。在这种管理机制下，运行在线程模式的用户代码使用 PSP，而操作系统内核和异常服务函数则使用 MSP。

3) 链接寄存器 LR

R14 链接寄存器(LR)：当调用函数或子程序时，由 LR 存储子程序运行完后返回的地址。在异常处理的时候，返回地址不会保存到 LR 中，而是保存到内存（栈）中。LR 会自动保存 EXC_RETURN（异常返回）数值，用于在异常处理结束后触发异常返回。

ARM 为了减少对内存的访问（提高流水线的效率），把返回地址直接存储在寄存器中。这样做使得第一级子程序调用不需要访问内存，从而提高了程序运行的效率。如果子程序调用不止一级，则需要把前一级的 LR 的值压到堆栈里，再使用 LR 保存当前调用函数的返回地址。访问寄存器比访问内存的速度更快，使用 LR 寄存器能使函数返回过程更快。

4) 程序计数器(PC)

R15 程序计数器(PC)：是程序运行的基础，具有自加的功能。该寄存器的 0 位始终为 0，因此，指令始终与字或半字边界对齐。

5) 特殊寄存器

特殊寄存器包括处理器状态、对操作模式和权限的设置、以及高级中断/异常屏蔽。在开发简单应用时，一般不需要访问这些寄存器。在使用嵌入式操作系统时，或者要使用高级/中断屏蔽特性时，就需要用到这些特殊寄存器。

xPSR 程序状态寄存器：处理器状态可分为 3 类，应用状态寄存器 (APSR)、中断状态寄存器(IPSR)、执行状态寄存器(EPSR)，它们可以在一个 32 位的寄存器内访问，统称为 xPSR，如表 2-1 所示。

表 2-1 xPSR 程序状态寄存器

	位															
	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q				GE							
IPSR										异常编号						
EPSR						ICI/IT	T			ICI/IT						

表 2-2 xPSR 寄存器各位的功能

位	描述
N	负标志：1：计算结果小于 0；0：计算结果大于 0
Z	零标志：1：计算结果等于 0；0：计算结果不等于 0
C	进位/借位标志：1：计算结果有进位或借位；0：没有进位或借位
V	溢出标志：1：计算结果溢出；0：没溢出
Q	饱和标志：1：饱和；0：没饱和
GE[3:0]	大于或等于标志，对应 4 个字节
ICT/CT	ICT 中断继续指令位；CT:IF-THEN 指令状态位
T	Thumb 状态
异常编号	正在处理的异常的编号

中断/异常屏蔽寄存器：分为三组，它们分别是 PRIMASK、FAULTMASK、BASEPRI。

PRIMASK 为片上可配置优先级异常的总开关，该寄存器只有位 0 有效，当该位为 0 时响应所有可配置优先级异常; 当该位为 1 时禁止除 NMI 和硬件错误(HardFault)外的所有异常。

FAULTMASK 寄存器为管理系统错误的总开关，该寄存器中有位 0 有效，当该位为 0 时，响应所有的异常; 为 1 时只响应 NMI，禁止其他所有异常。

BASEPRI 寄存器用来禁止优先级等于和低于某个优先级的异常。

控制寄存器(CONTROL)：有两个作用，其一用于定义线程模式的权限级别，其二用于选择栈指针，如表所示。

表 2-3 控制寄存器 CONTROL

位	功能
nPRIV(0 位）	线程模式的权限级别：0：特权级；1：用户级
SPSEL（第 1 位）	选择线程模式下的栈指针：0：使用 MSP；1：使用 PSP

6 异常和中断

异常(Exception)是指打断程序正常执行的事件。发生异常时，处理器会暂停当前的程序，而去执行异常服务程序，异常服务结束后，又会回来继续执行原来的程序。像我们前面学习的外设中断，就是一种异常。除了外设产生的中断外，还有系统执行时发生的错误事件、软

件产生的异常以及支持操作系统的异常。在不严格的时候，异常与中断也常常混称。

Cortex-M4 中的嵌套向量中断控制器 NVIC (Nested Vectored Interrupt Controller)，负责处理异常和中断配置、优先级以及中断屏蔽。NVIC 定义了 16 种系统异常和 240 路外设中断。通常具体芯片上的外设中断都不会用满 240 路。

表 2-4 Cortex-M4F 异常类型

异常类型	异常编号	优先级	向量地址	描述
-	0	-	0x0000 0000	
复位	1	-3(最高)	0x0000 0004	上电或复位，具有最高优先级
不可屏蔽的中断(NMI)	2	-2	0x0000 0008	外部 NMI 引脚发出或者使用中断控制及状态(INTCTRL)寄存器触发
硬件错误	3	-1	0x0000 000C	若其它错误由于禁止或屏蔽而未被激活，则触发此错误
存储器管理错误	4	可编程	0x0000 0010	由 MPU 触发或者其他存储器的非法访问
总线错误	5	可编程	0x0000 0014	由总线系统发出，如指令预取错误或数据访问错误
使用错误	6	可编程	0x0000 0018	指令执行的相关故障
-	7--10	-	-	保留
SVCall	11	可编程	0x0000 002C	由 SVC 指令触发。在 OS 环境，用户程序可以使用 SVC 指令来请求访问 OS 内核函数和请求管理调用
调试监控器	12	可编程	0x0000 0030	由调试监视器引起的
-	13	-	-	保留
PendSV	14	可编程	0x0000 0038	可悬挂系统服务请求
SysTick	15	可编程	0x0000 003C	由系统定时器 SysTick 产生
IRQ0	16	可编程	0x0000 0040	外设中断 0
IRQ1	17	可编程	0x0000 0044	外设中断 1
….	…	…	…	…
IRQ239	255	可编程	0x0000 03FC	外设中断 239

6.1 优先级

当多个中断同时发生时，中断的优先级决定了中断响应的先后次序。优先级的数值越小，则优先级越高。在表 2-4 中，复位、不可屏蔽中断和硬件错误有着为负数的优先级数值，优先级高于其它中断，且不可更改。其它中断（异常编号 4-255）的优先级是可以编程的。

在 Cortex-M4 中，如果没有对中断优先级编程，则中断优先级由其编号顺序决定。NVIC 支持通过软件修改中断的优先级。外设中断在 NVIC 中的中断优先级寄存器 PRIn 中写入新的优先级，数值越小，优先级越高。设置软件优先级后，原来的优先级则会无效。例如: 如果将 0 号外设中断指定为优先级 1，31 号外设中断指定为优先级 0，则 0 号外设中断的优先级比 31 号外设中断低。

Cortex-M4 支持抢占式中断，即高优先级中断可以抢占低优先级中断。Cortex-M4 支持 128 级抢占，但绝大多数微控制器芯片都会精简设计，通常的做法是裁掉表达优先级的几个低端有效位，减少优先级的级数。例如：Tiva_C 系列微控制器中只有 8 级（0 到 7）可编程优先级，则其优先级可以用一个 3 位的二进制数表示，在 PRIn 寄存器中的高 3 位[7：5]中设置，而低 5 位则未使用。

Cortex-M4 支持优先级分组，以更好的对大量中断优先级进行管理。将优先级按优先级高低分成若干个组，例如将 8 级优先级分为 4 个组，0-1 分到 0 组、2-3 分到 1 组、4-5 分到 2 组、6-7 分到 3 组。

8 级优先级可以用一个 3 位的二进制数表示，从 000B 到 111B。在上例的 0 组中，000B 和 001B 的高 2 位都为 00B，是组号，也是其组(抢占)优先级；而最低位分别为 0B 和 1B，称为子优先级。同样，在其他组中同组的高 2 位也都是相同的，是其组(抢占)优先级，最低位是子优先级。

组优先级高的中断可以抢占组优先级低的中断。例如：0 组的中断 0 的组优先级高于 1 组的中断 2 的组优先级，则中断 0 可以抢占中断 2。即在执行中断 2 的服务程序的过程中，如果中断 0 到来，则 0 可以打断 2 的中断服务，先执行完中断服务程序 0 再继续执行中断服务程序 2。

同组的中断不能互相抢占；当多个同组的中断同时产生或挂起时，子优先级高的中断会优先响应。例如：如果 2 组中的中断 5 服务还未完成时，又发生了中断 4，则会等待到待中断 5 的服务结束后，再执行中断 4 的服务函数；如果中断 4 和中断 5 同时发生，则会先执行中断 4 的服务。

优先级分组在 APINT 寄存器中的 PRIGROUP 位域中进行设置。Tiva_C 中的 8 级优先级在 PRIn 寄存器的[7:5]位中表示。PRIGROUP 决定了如何分组，其分组设置如表 2-5 所示：例如当 PRIGROUP 为 0x0-0x04 时，优先级 3 位全为组（抢占）优先级，则有 8 级组（抢占）优先级，只有 1 级子优先级。

表 2-5 Tiva_C 中组（抢占）优先级和子优先级分组位置的关系

PRIGROUP	二进制表示	组优先级位	子优先级位	组优先级数	子优先级数
0x0-0x4	xxx	[7:5]	无	8	1
0x5	xxy	[7:6]	[5]	4	2
0x6	xyy	[7]	[6:5]	2	4
0x7	yyy	无	[7:5]	1	8

第二列是表示 8 级优先级的三位二进制数，其中 x 表示组（抢占）优先级位，y 表示子优先级位

6.2 异常向量表

当发生了中断并要对其响应时，处理器会跳转到该中断的服务函数执行，这时候 Cortex-M4 需要知道该函数的起始地址。异常向量表就存储了所有异常/中断服务函数的起始地址，这些函数起始地址被称为异常/中断向量。向量表在存储空间的位置是固定的，所有中断向量在向量表中的位置也是固定的。当某个中断发生时，处理器会在向量表的相应位置读取它的中断向量（其服务函数的入口地址），再跳转执行中断服务函数。

向量表默认从地址 0x0000 0000 开始，每一个向量占用 4 字节，复位后的向量如表 2-6 所示。

表 2-6 异常向量表

异常编号	地址	描述
NA	0x0000 0000	MSP 初始值
1	0x0000 0004	复位
2	0x0000 0008	NMI
3	0x0000 000c	硬件错误
4	0x0000 0010	存储管理错误
…	…	…
15	0x0000 003C	SysTick
16	0x0000 0040	IRQ0
17	0x0000 0044	IRQ1
18~255	0x0000 0048~0x0000 03FF	IRQ2~IRQ239

值得注意的是，向量表中的第 1 个字(地址 0x0000 0000)是主栈指针(MSP)的初始值，而不是某个中断服务函数的起始地址，复位时处理器读取该地址的数据来初始化主堆栈。

发生中断时，Cortex-M4 会从中断向量确定中断服务函数的地址，然后跳到中断服务函数执行。例如复位向量（地址 0x0000_0004）保存的是复位中断服务函数的起始地址。MCU 上电或复位时，会首先到地址 0x0000_0004 找到复位中断服务函数的地址，然后跳去执行。

向量表重定位

向量表的默认地址（0x0000 0000）处一般为启动存储器，是只读存储器 FLASH 或 ROM 设备，在程序运行时是不能修改的。不过，有些应用有可能需要在运行时修改向量表，因此 Cortex-M4 可以将向量表重定位，将向量表复制到可读写存储器 SRAM 区，从而可以通过程序修改。

在 TIVA_C 中，通过设置向量偏移寄存器 VTABLE 来更改向量表的位置，如表 2-7 所示。使用向量表重定位，向量表地址必须和 2 的整数次方对齐。先求出系统中共有多少个向量，再把这个数字向上增大到 2 的整次幂，而起始地址必须对齐到后者的边界上。例如，如果一共有 32 个中断，则共有 32+16（系统异常）=48 个向量，向上增大到 2 的整次幂后值为 64，因此地址必须能被 64*4=256 整除，从而合法的起始地址是：0x0、0x100、0x200 等。TM4C123GH6PM 芯片共有 138 个异常/中断，因此需要 1024 对齐，则地址的 0:9 位为 0。

表 2-7 向量偏移寄存器 VTABLE

31:10 位	9:0 位
偏移地址	保留

6.3 异常的进入与退出

1) 中断进入

压栈：当发生异常/中断时，会先保护现场，由处理器自动把 8 个寄存器(xPSR、PC、LR、R0~R3、R12)压入栈。

取向量：紧接着根据向量表找出相应的异常/中断向量，然后在服务程序的入口处预取指。由于 Cortex-M4 能将数据存取与取指通过不同的总线（系统总线和 I-CODE 总线）操作，所以压栈与取指这两项工作能并行进行，以便快速进入中断。

更新寄存器：压栈和取向量操作完成之后，在执行服务程序之前，还必须更新一系列寄

存器，包括栈指针 SP、程序状态寄存器 PSR、链接寄存器 LR、程序计数器 PC，以及 NVIC 中的相关寄存器。LR 被设置为 EXC_RETURN 值，其中保存了异常流程的相关状态（如压栈使用的是 MSP 还是 PSP），以在异常返回时使用。

2) 中断退出

在 Cortex-M4 中，是通过把 LR 中的 EXC_RETURN 加载到 PC 里来触发中断服务结束的。在启动了中断返回序列后，需要恢复现场：

出栈：将先前压入栈中的寄存器在这里恢复，把栈内的数据弹出写回到寄存器中，出栈顺序与入栈时的相对应，栈指针的值也改回先前的值。

更新 NVIC 的相关寄存器：伴随着中断的返回，中断的活跃位也被硬件清除。

3) 尾链机制

当处理器在响应某中断 1 时，如果又发生其它中断 2，但中断 2 优先级不够高，则被挂起。那么在 ISR1（中断服务函数 1）执行返回后，系统处理 ISR2 时，倘若还是先把相关寄存器出栈，然后又把出栈的数据重新压栈，会白白浪费 CPU 时间。因此，在 ISR1 执行完后，Cortex-M4 不会执行出栈操作，而是在重新取向量和更新少量寄存器（6 个时钟周期）后，直接执行 ISR2。这么做前后只执行了一次压栈与出栈操作，两个中断花费的总时间少了很多，如图 2-7 所示。

图 2-7 尾链

4) 中断迟来

在中断压栈时，或者还没来得及执行其服务函数时，如果此时收到了高优先级中断的请求，则本次入栈就成了为高优先级中断所做的了。入栈后，将先执行高优先级中断的服务函数。

比如，若在响应某低优先级中断 1 的早期，检测到了高优先级中断 2，则只要中断 2 没有太晚，就能以“中断迟来”的方式处理：在压栈完毕后先执行 ISR2，在 ISR2 执行完毕后，再以“尾链”方式，来启动 ISR1 的执行。如果中断 2 来得太晚，已经开始执行 ISR1 的服务了，则按普通的抢占处理，这会需要更多的处理器时间（多一次压栈和出栈）和额外的栈空间。

6.4 异常的挂起与活跃状态

当异常/中断发生时，如果正在处理同级或高优先级中断，或者被某个屏蔽寄存器屏蔽，则中断服务不能立即执行，此时中断被挂起。中断挂起确认后，即使后面中断请求消失，中断的挂起记录也会继续保持。到了系统中它的优先级最高的时候，或者屏蔽被取消，就会得到响应。但如果在中断得到响应前，手工清除了中断挂起标志（写解挂寄存器），则中断被取消。

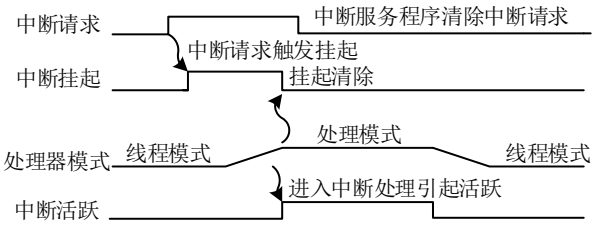

图 2-8 中断的挂起和活跃状态

中断的挂起状态可以通过读置位挂起寄存器(PENDn)和中断解挂寄存器(UNPENDn)来查看，还可以通过写它们来手工挂起和清除挂起中断。

当某中断的服务函数一旦开始执行，就称此中断进入了“活跃”状态，中断的活跃状态可以在活跃位寄存器(ACTIVEn)中查询。如图2-8所示，此时硬件会自动清除其挂起标志，中断请求信号也会被自动清除。

如果在某个中断挂起时，又来了多个同一中断的请求，这些中断请求会被忽略。

如果中断请求一直保持，则该中断会在中断服务函数返回后再次被置为挂起状态。

如果在服务函数执行时，中断请求被释放了，但是在中断服务函数返回前，又来了中断请求，则会重新挂起该中断。

6.5 NVIC 的寄存器

在嵌入式中，对外设的操作主要是通过寄存器来实现的，包括查看外设的工作状态和控制外设工作。外设寄存器都被分配了地址，可以象内存一样访问。NVIC寄存器的访问起始地址是0xE000_E000。大部分NVIC的中断控制/状态寄存器都只能在特权级下访问，除了软件触发中断寄存器可以在用户级下访问，用来产生软件中断。

NVIC的寄存器有：

- 使能与禁止寄存器(ENn 和 DISn)
- 挂起与解挂寄存器(PENDn 和 UNPENDn)
- 优先级寄存器(PRIn)
- 活跃状态寄存器(ACTIVEn)
- 软件触发中断寄存器(SWTRIG)

这些寄存器的操作都很相似，对相应位进行读写操作即可。例如使能与禁止寄存器（ENn 和 DISn），是对片上 240 路外设中断的进行使能和禁止控制，使能是向 ENn 寄存器对应位写“1”，禁止是向 DISn 寄存器对应位写“1”。

另外，下列寄存器也对中断处理有重大影响：

- 中断掩蔽寄存器(PRIMASK, FAULTMASK 以及 BASEPRI)
- 向量表偏移寄存器(VTABLE)（改变向量表位置）
- 应用程序中断及复位控制寄存器(APINT)（优先级分组）

6.6 中断编程实例

6.6.1 位运算和位域

1) 位运算

在计算机中，数据是以二进制的形式存储的，位运算就是直接对整数的二进制位进行操作。在对数据修改时，很多时候需要用到位运算。

表 2-8 C 语言中的位运算符

名称	符号	说明	运算实例
按位与	&	相同位的两个数字都为 1，则为 1；若有一个不为 1，则为 0	0000 1101 & 0000 1010 = 0000 1000
按位或	|	相同位只要一个为 1 即为 1	0000 0011 | 0000 0111 = 0000 0111
按位异或	^	相同位不同则为 1，相同则为 0	0000 1100 ^ 0000 0111 = 0000 1011
按位取反	~	取反	～0000 1001= 1111 0110
左移	<<	左移，高位丢弃，低位补 0	0000 0101 << 4 = 0101 0000
右移	>>	右移(对有符号数的处理有差异)	0101 0000 >> 4 = 0000 0101

2） 位域

寄存器往往有多个功能字段，用寄存器的若干个位（bit）来实现某个功能。寄存器中这些表示不同的功能区域的字段叫做位域。例如：软件触发中断寄存器（SWTRIG）中的0-7位为INTID位域，而8-31位没有使用（保留）。向INTID位域写入一个8位的数值，会产生一个外设中断号为该数值的中断，例如：写入01，则在IRQ1上产生中断（tm4c1231h6pge中文数据手册第134页）。

表 2-9 软件触发中断寄存器 SWTRIG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								IMTID							

Tivaware目录下的inc/hw_nvic.h文件中宏定义了NVIC相关寄存器和寄存器的各个位域，指出了各寄存器的地址以及寄存器的位域。

例如：SWTRIG 寄存器的宏定义如下，其中 0xE000EF00 为 SWTRIG 寄存器在存储空间的地址。：

```
#define NVIC_SW_TRIG 0xE000EF00  // Software Trigger Interrupt
```

SWTRIG寄存器INTID位域的宏定义，因为INTID位域为0-7位，相应位取1，其他位取0，故为0x000000FF：

```
#define NVIC_SW_TRIG_INTID_M 0x000000FF  // Interrupt ID
```

3） 寄存器的读写

Tivaware 目录下的 inc/hw_types.h 定义了对硬件地址直接读写的方法，

```
#define HWREG(x)       (*((volatile uint32_t *)(x)))
#define HWREGH(x)      (*((volatile uint16_t *)(x)))
#define HWREGB(x)      (*((volatile uint8_t *)(x)))
```

这是一个宏调用定义，其实是用指针的方法进行操作。其中x是寄存器宏定义，即寄存器地址。例如：

读取SWTRIG寄存器的值到变量temp中：

```
temp = HWREG(NVIC_SW_TRIG);
```

改写SWTRIG寄存器的值为变量temp中的值：

```
HWREG(NVIC_SW_TRIG) = temp;
```

如果将SWTRIG寄存器中的值改写为1，则：

```
HWREG(NVIC_SW_TRIG) = 0x01；
```

4） 位域的读写

对寄存器的访问往往只想访问寄存器的某个位域，比如只读取寄存器的一个位域，或者只改写寄存器的一个位域，而不影响其他位域。

a） 只读取某位域的值，将寄存器的值直接和位域宏定义相与，则只读出该位域的值，其它位读出的为0，例如：

```
temp = HWREG(NVIC_SW_TRIG) & NVIC_SW_TRIG_INTID_M;
```

假定SWTRIG的值为0x00002078，则：temp = 0x00002078 & 0x000000ff = 0x00000078

b） 要将寄存器某位域全置1，其他位保持不变，则将寄存器的值和位域宏定义相或，例如：

```
HWREG(NVIC_SW_TRIG) |= NVIC_SW_TRIG_INTID_M;
```

假定 NVIC_SW_TRIG 的值为0x00002078，则：SWTRIG的值 = 0x00002078 | 0x000000ff = 0x000020ff

c） 要将寄存器某位域清0，其他位保持不变，则将该位域宏定义取反后，和寄存器的值相与，例如：

```
HWREG(NVIC_SW_TRIG) &= ~(NVIC_SW_TRIG_INTID_M);
```

假定SWTRIG的值为0x00002078，则：SWTRIG的值 = 0x00002078 & ~(0x000000ff) = 0x00002000

d） 要将寄存器某位域赋值，其他位保持不变，则需先将该位域的值清零，再与赋值相或（赋值需位移到该位域的位置），例如将位域改写为变量temp中相应位的值，其它位不变：

```
HWREG(NVIC_SW_TRIG) &= ~(NVIC_SW_TRIG_INTID_M);
HWREG(NVIC_SW_TRIG) |= temp & NVIC_SW_TRIG_INTID_M;
```

假定SWTRIG的值为0x00002078，其INTID位域要改为temp=0x00030057中相应位的值：

SWTRIG的值 = 0x00002078 & ~(0x000000ff) = 0x00002000

SWTRIG的值 = 0x00002000 | (0x00003057 & 0x000000ff) = 0x00002057

6.6.2 中断编程实例

中断程序的编写包括：

1） 包括中断的初始化（配置和使能），确定什么情况下产生中断。

2） 编写中断服务函数，产生中断后，会跳去执行中断服务函数，中断服务函数执行完后会返回到程序原来的位置，继续执行。

3） 修改中断向量表，告知中断服务函数的地址，使得发生中断时能找到中断服务函数。

下面来编写一个IRQ1软件触发中断程序。向软件触发中断寄存器（SWTRIG）写入相应IRQ的号码，即可触发相应的IRQ。这里写入1，触发IRQ1中断。

1） main.c（修改lab2）文件如下，在while循环中，每延时一段时间，触发1次IRQ1中断。

```
//-------------------------------------------------------------------------------------------
//main.c文件
#include <stdint.h>
#include <stdbool.h>
```

```
#include "inc/tm4c123gh6pm.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_nvic.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);                              //初始化GPIOF端口
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3); //设置F1、F2、F3输出
    HWREG(NVIC_EN0) |= 0x00000002;      //使能IRQ1
    while(1)
    {
        HWREG(NVIC_SW_TRIG) = 1;        //触发中断IRQ1
        SysCtlDelay(2000000);           //延时
    }
}
//-------------------------------------------------------------------------------------------
```

2） 修改tm4c1231h6pm_startup_ccs.c文件，添加中断服务函数和中断服务函数的声明。

在文件的前面添加包含头文件和中断服务函数的声明。

```
//-------------------------------------------------------------------------------------------
//修改tm4c1231h6pm_startup_ccs.c文件，在前面添加IRQ1IntHandler中断服务函数的声明
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"

static void IRQ1IntHandler(void);
//-------------------------------------------------------------------------------------------
```

在文件的后面添加 IRQ1IntHandler 中断服务函数，每进一次中断服务函数，都会改变一次 LED 灯的状态。

```
//-------------------------------------------------------------------------------------------
//修改tm4c1231h6pm_startup_ccs.c文件，在后面添加IRQ1IntHandler中断服务函数
static void IRQ1IntHandler(void)
{
    static unsigned char led_on;

    if(led_on == 0)
     {
          GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0);//操作GPIO点亮LED灯
          led_on = 1;
     }
```

```
        else
        {
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 4);              //操作GPIO熄灭LED灯
            led_on = 0;
        }
    }
    //-----------------------------------------------------------------------------------------
```

3） 我们需要修改中断向量表，将IRQ1的中断向量指向IRQ1的中断服务函数。

在tm4c1231h6pm_startup_ccs.c文件中，定义了中断向量表和中断服务函数。可以看到，一般的中断向量都被定义为IntDefaultHandler，在文件后面我们可以找到IntDefaultHandler中断服务函数的定义，这是一个不做任何处理的服务函数，进入后会陷入死循环。将IRQ1的中断向量改为IRQ1IntHandler。

```
    //-----------------------------------------------------------------------------------------
    //修改tm4c1231h6pm_startup_ccs.c文件中的向量表，将IRQ1的中断向量（第18个）改为IRQ1IntHandler
    #pragma DATA_SECTION(g_pfnVectors, ".intvecs")
    void (* const g_pfnVectors[])(void) =
    {
        (void (*)(void))((uint32_t)&__STACK_TOP),  // The initial stack pointer
        ResetISR,                               // The reset handler
        NmiSR,                                  // The NMI handler
        FaultISR,                               // The hard fault handler
        IntDefaultHandler,                       // The MPU fault handler
        IntDefaultHandler,                       // The bus fault handler
        IntDefaultHandler,                       // The usage fault handler
        0,                                      // Reserved
        0,                                      // Reserved
        0,                                      // Reserved
        0,                                      // Reserved
        IntDefaultHandler,                       // SVCall handler
        IntDefaultHandler,                       // Debug monitor handler
        0,                                       // Reserved
        IntDefaultHandler,                       // The PendSV handler
        IntDefaultHandler,                       // The SysTick handler
        IntDefaultHandler,                       // GPIO Port A
        IRQ1IntHandler,                         // IRQ1的中断向量，修改为IRQ1IntHandler
        ……
    }
    //-----------------------------------------------------------------------------------------
```

7 SysTick 定时器

SysTick 定时器是 Cortex-M4 处理器中集成的一个 24 位、递减计数的定时器，主要用来为操作系统产生“滴答”时基。在现代操作系统中，为了让多个用户任务“并行”运行，会将时间切成许多小的“时间片”，每个任务一次运行一个“时间片”就让给下一个任务，多

个任务轮流运行。如此，从宏观上看好像是多个任务同时运行。SysTick 定时器就是供操作系统产生“时间片”，进行任务管理和上下文切换的。SysTick 定时器只能由有特权级权限的操作系统控制，而用户程序是不能访问的。在没有操作系统的简单应用中，SysTick 定时器可以作为一个简单定时器使用。

TM4C123GH6PM 的 SysTick 定时器有 3 个寄存器：STRELOAD、STCURRENT 和 STCTRL。SysTick 可以周期性的产生一个定时信号。SysTick 中的计数器装入 STRELOAD 中保存的 RELOAD 数值并开始递减计数，每个 SysTick 时钟减 1。当计数器递减到 0 时，产生一个定时中断信号。然后，计数器重新装入 RELOAD 值并开始递减计数，如此不断循环，周期性的产生定时信号。定时的时间长度由 SysTick 时钟和 RELOAD 值决定。

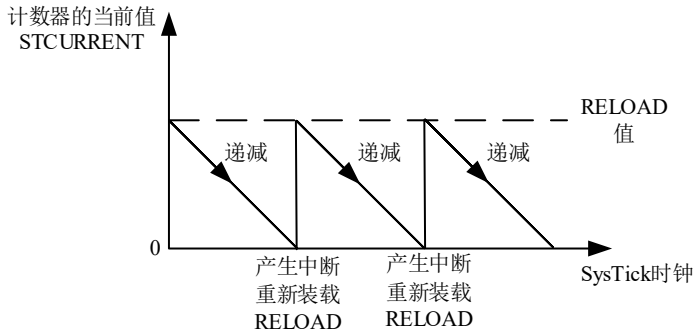

图 2-9：systick 定时中断

SysTick 重载值寄存器(STRELOAD)，当计数至零时将被重新装载的值(RELOAD)。

SysTick 当前值寄存器(STCURRENT)，读取时返回当前计数的值。

SysTick 控制及状态寄存器(STCTRL)，使能 SysTick，使能中断，选择 SysTick 时钟，见表 2-9。

表 2-9 SysTick 控制及状态寄存器(STCTRL)

位域	名称	类型	复位值	描述
16	COUNT	RO	0	如果在上次读取本寄存器后，SysTick 已经计数到了 0，则该位为 1。如果读取该位，该位将自动清零
2	CLK_SRC	R/W	1	0：时钟使用精确内部振荡器(PIOSC) 4 分频 1：时钟使用系统时钟
1	INTEN	R/W	0	1 ：使能 SysTick 中断
0	ENABLE	R/W	0	0：禁用计数器 1：使 SysTick 周期性定时

RO:只能读（read only）
R/W:可读写（read/write）

时钟选择：CLK_SRC 决定了 SysTick 时钟采用 PIOSC4 分频(4MHz)，还是系统时钟。例如：在 TIVA_C 中，如果设置 STRELOAD 为 1000000；CLK_SRC 设为 0，则时钟为精确内部振荡器 (PIOSC) 4 分频，即时钟频率为 4M；INTEN 设为 1，SysTick 中断启用；ENABLE 设为 1，定时器开始工作；SysTick 定时器将每 0.25 秒（1000000 次计数）产生 1 次中断。

习题

1） 请上网搜索，列举 3 个不同厂商、不同型号的 ARM Cortex-M4 内核的 MCU 型号，对它们工作主频，内置存储器容量，内置外设类型进行说明。

2) 在 Tivaware 目录下的 inc/hw_nvic.h 中找到 SysTick 的各个寄存器宏定义和寄存器 STCTRL 的位域宏定义并记录。

3) 参考本章中断例程，编写一个 systick 中断程序，其时钟源设为精确内部振荡器(PIOSC) 4 分频（此时频率为 4MHz），使其每次定时时间为 0.5 秒，让 led 亮 0.5 秒，熄 0.5 秒。

4) 查阅 Tivaware 目录下的 doc\SW-TM4C-DRL-UG-2.1.4.178.pdf 中 28 章 System Tick (SysTick)章节，列出 SysTick 的所有库函数，并简略说明其功能（Tivaware 目录下的 driverlib\systick.c 有相关库函数的源代码）。

5) 使用 SysTick 的库函数完成课堂习题第 2 题的编程，并运行验证。