LPC1100 系列微控制器

第二十二章 ARM Cortex-M0 参考资料

用户手册 Rev1.00

广州周立功单片机发展有限公司

地址：广州市天河北路 689 号光大银行大厦 12 楼 F4

网址：http://www.zlgmcu.com

销售与服务网络（一）

广州周立功单片机发展有限公司

地址：广州市天河北路 689 号光大银行大厦 12 楼 F4
邮编：510630
电话：(020)38730916　38730917　38730972　38730976　38730977
传真：(020)38730925
网址：www.zlgmcu.com

广州专卖店

地址：广州市天河区新赛格电子城 203-204 室
电话：(020)87578634　87569917
传真：(020)87578842

南京周立功

地址：南京市珠江路 280 号珠江大厦 1501 室
电话：(025) 68123901　68123902
传真：(025) 68123900

北京周立功

地址：北京市海淀区知春路 113 号银网中心 A 座 1207-1208 室（中发电子市场斜对面）
电话：(010)62536178　62536179　82628073
传真：(010)82614433

重庆周立功

地址：重庆市石桥铺科园一路二号大西洋国际大厦（赛格电子市场）1611 室
电话：(023)68796438　68796439
传真：(023)68796439

杭州周立功

地址：杭州市天目山路 217 号江南电子大厦 502 室
电话：(0571)89719480　89719481　89719482
89719483　89719484　89719485
传真：(0571)89719494

成都周立功

地址：成都市一环路南二段 1 号数码科技大厦 403 室
电话：(028)85439836　85437446
传真：(028)85437896

深圳周立功

地址：深圳市深南中路 2070 号电子科技大厦 C 座 4 楼 D 室
电话：(0755)83781788（5 线）
传真：(0755)83793285

武汉周立功

地址：武汉市洪山区广埠屯珞瑜路 158 号 12128 室（华中电脑数码市场）
电话：(027)87168497　87168297　87168397
传真：(027)87163755

上海周立功

地址：上海市北京东路 668 号科技京城东座 7E 室
电话：(021)53083452　53083453　53083496
传真：(021)53083491

西安办事处

地址：西安市长安北路 54 号太平洋大厦 1201 室
电话：(029)87881296　83063000　87881295
传真：(029)87880865

销售与服务网络（二）

广州致远电子有限公司

地址：广州市天河区车陂路黄洲工业区 3 栋 2 楼
邮编：510660
传真：(020)38601859
网址：www.embedtools.com （嵌入式系统事业部）
www.embedcontrol.com （工控网络事业部）
www.ecardsys.com （楼宇自动化事业部）

技术支持：

CAN-bus：

电话：(020)22644381 22644382 22644253
邮箱：can.support@embedcontrol.com

iCAN 及数据采集：

电话：(020)28872344 22644373
邮箱：ican@embedcontrol.com

MiniARM：

电话：(020)28872684 28267813
邮箱：miniarm.support@embedtools.com

以太网：

电话：(020)22644380 22644385
邮箱：ethernet.support@embedcontrol.com

无线通讯：

电话：(020) 22644386
邮箱：wireless@embedcontrol.com

串行通讯：

电话：(020)28267800 22644385
邮箱：serial@embedcontrol.com

编程器：

电话：(020)22644371
邮箱：programmer@embedtools.com

分析仪器：

电话：(020)22644375
邮箱：tools@embedtools.com

ARM 嵌入式系统：

电话：(020) 22644383 22644384
邮箱：NXPARM@zlgmcu.com

楼宇自动化：

电话：(020)22644376 22644389 28267806
邮箱：mjs.support@ecardsys.com
mifare.support@zlgmcu.com

销售：

电话：(020)22644249 22644399 22644372 22644261 28872524
28872342 28872349 28872569 28872573 38601786

维修：

电话：(020)22644245

目　录

第22章 ARM Cortex-M0 参考资料

22.1 简介

下面的参考资料以 ARM Cortex-M0 用户指南（ARM Cortex-M0 User Guide）为蓝本。只针对 LPC111x Cortex-M0 的具体实现做了细微的改动。

22.2 Cortex-M0 的处理器和内核外设

Cortex-M0 处理器是一个入门级（entry-level）的 32 位 ARM Cortex 处理器，设计用在更宽范围的嵌入式应用中。该处理器包含以下特性，给开发者提供了极大的便利：

- 结构简单，容易学习和编程；
- 功耗极低，运算效率高；
- 出色的代码密度；
- 确定、高性能的中断处理；
- 向上与 Cortex-M 处理器系列兼容。

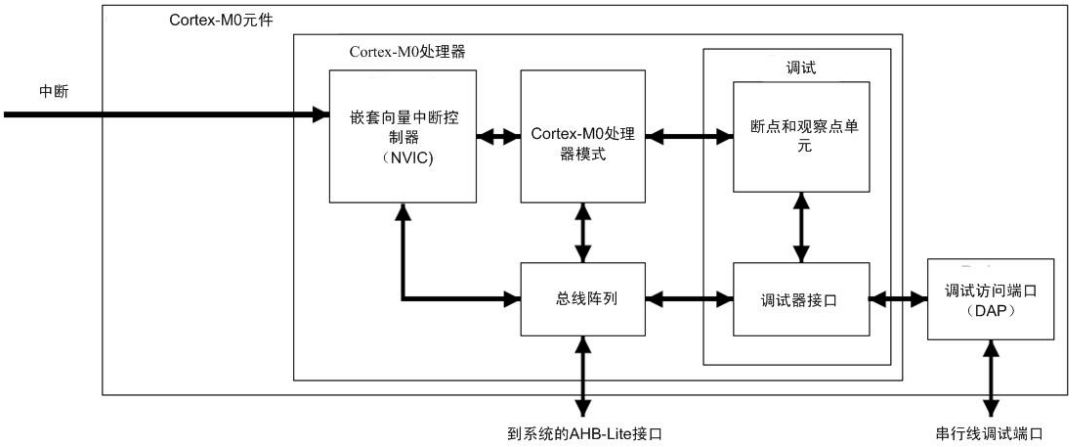

图 22.1 Cortex-M0 的具体实现

Cortex-M0 处理器基于一个高集成度、低功耗的 32 位处理器内核，采用一个 3 级流水线冯·诺伊曼结构（von Neumann architecture）。通过简单、功能强大的指令集以及全面优化的设计（提供包括一个单周期乘法器在内的高端处理硬件），Cortex-M0 处理器可实现极高的能效。

Cortex-M0 处理器采用 ARMv6-M 结构，基于 16 位的 Thumb 指令集，并包含 Thumb-2 技术。提供了一个现代 32 位结构所希望的出色性能，代码密度比其他 8 位和 16 位微控制器都要高。

Cortex-M0 处理器紧密集成了一个可配置的嵌套向量中断处理器（NVIC），提供业界领先的中断性能。NVIC 具有以下功能：

- 包含一个不可屏蔽的中断（NMI）。NMI 在 LPC111x 上不能实现；
- 提供零抖动中断选项；
- 提供 4 个中断优先级。

处理器内核和 NVIC 的紧密结合使得中断服务程序（ISR）可以快速执行，极大地缩短了中断延迟。这是通过寄存器的硬件堆栈以及加载-乘和存储-乘操作的停止和重启来获得

的。中断处理程序不需要任何汇编封装代码，不用消耗任何 ISR 代码。末尾连锁的优化还极大地降低了一个 ISR 切换到另一个 ISR 时的开销。

为了优化低功耗设计，NVIC 还与睡眠模式相结合，提供一个深度睡眠功能，使整个器件迅速掉电。

22.2.1 系统级接口

Cortex-M0 处理器提供一个简单的系统级接口，使用 AMBA 技术来提供高速、低延迟的存储器访问。

22.2.2 集成的可配置调试

Cortex-M0 处理器执行一个完整的硬件调试方案，带有大量的硬件断点和观察点选项。通过一个非常适合微控制器和其他小型封装器件的 2 脚串行线调试（SWD）端口，提供了高系统透明度的处理器、存储器和外设执行。

22.2.3 Cortex-M0 处理器的特性小结

- 高代码集成度，具有 32 位的性能；
- 工具和二进制代码与 Cortex-M 处理器系列向上兼容；
- 集成了极低功耗的睡眠模式；
- 高效的代码执行允许处理器时钟更低，或者延长睡眠模式的时间；
- 单周期的 32 位硬件乘法器；
- 零抖动的中断处理；
- 宽范围的调试功能。

22.2.4 Cortex-M0 内核外设

Cortex-M0 内核外设有：

NVIC —— NVIC 是一个嵌入式中断控制器，支持低延迟的中断处理。

系统控制块 —— 系统控制块（SCB）是到处理器的编程模型接口。它提供系统执行信息和系统控制，包括配置、控制和系统异常的报告。

系统定时器 —— 系统定时器，SysTick，是一个 24 位的递减定时器。可以将其用作一个实时操作系统（RTOS）的节拍定时器，或者用作一个简单的计数器。

22.3 处理器

22.3.1 编程模型

本节描述了 Cortex-M0 的编程模型。除了个别内核寄存器的描述之外，本节还包含处理器模式和堆栈的相关信息。

1. 处理器模式

处理器模式有：

线程模式 —— 用来执行应用软件。处理器在退出复位时进入线程模式。

处理器模式 —— 用来处理异常。处理器在完成所有的异常处理后返回到线程模式。

2. 堆栈

处理器使用一个满递减堆栈。这就意味着堆栈指针指向堆栈存储器中的最后一个堆栈项。当处理器将一个新的项压入堆栈时，堆栈指针递减，然后将该项写入新的存储器单元。

处理器执行两个堆栈，主堆栈和进程堆栈，两个堆栈有自己独立的堆栈指针副本，见本章“堆栈指针”小节。

在线程模式下，CONTROL 寄存器控制着处理器使用主堆栈还是进程堆栈，见本章“CONTROL 寄存器”小节。在处理器模式下，处理器总是使用主堆栈。处理器操作的选择如下：

表 22.1 处理器模式和堆栈使用的选择

处理器模式	用来执行	使用的堆栈
线程模式	应用程序	主堆栈或进程堆栈，见本章“CONTROL 寄存器”小节
处理器模式	异常处理程序	主堆栈

3. 内核寄存器

处理器内核寄存器有：

表 22.2 内核寄存器组小结

名称	类型[1]	复位值	描述
R0-R12	R/W	不可知	见本章“通用寄存器”小节
MSP	R/W	见文中描述	见本章“堆栈指针”小节
PSP	R/W	不可知	见本章“堆栈指针”小节
LR	R/W	不可知	见本章“链接寄存器”小节
PC	R/W	见文中描述	见本章“程序计数器”小节
PSR	R/W	不可知[2]	见表 22.3
APSR	R/W	不可知	见表 22.4
IPSR	RO	0x00000000	见表 22.5
EPSR	RO	不可知[2]	见表 22.6
PRIMASK	R/W	0x00000000	见表 22.7
CONTROL	R/W	0x00000000	见表 22.8

[1] 描述线程模式和处理器模式下程序执行过程中的访问类型。调试访问可以不同。

[2] Bit[24]是 T-bit，从复位向量的 bit[0]加载进来。

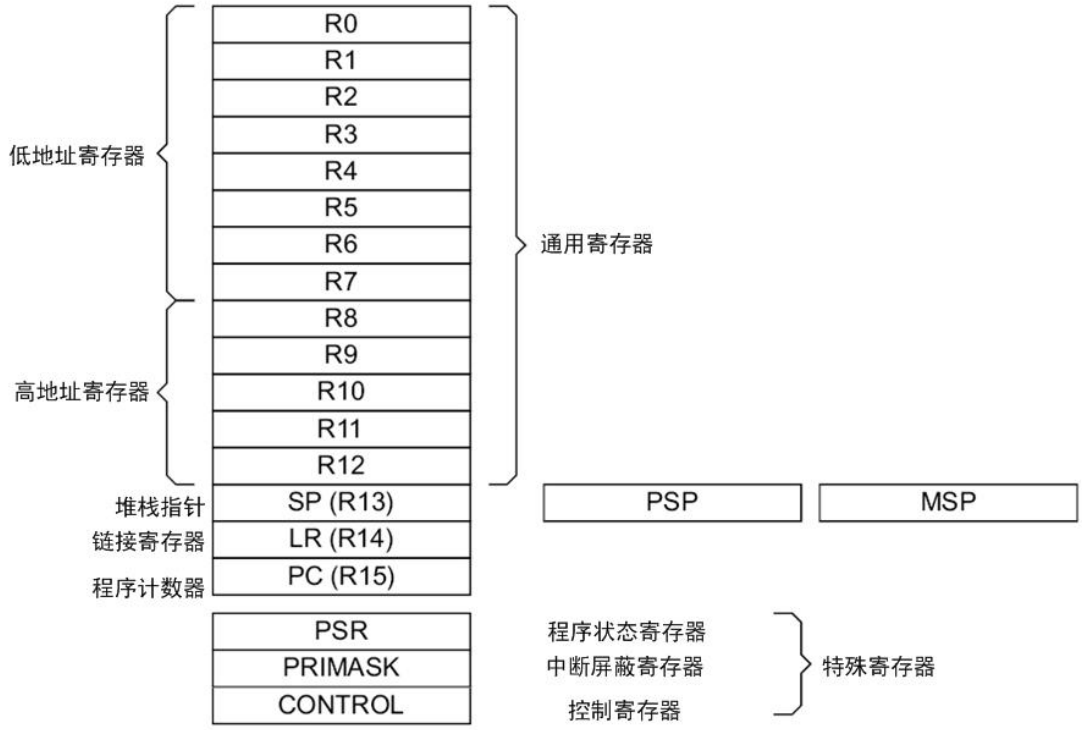

图 22.2 处理器内核寄存器组

（1）通用寄存器

R0-R12 是供数据操作使用的 32 位通用寄存器。

（2）堆栈指针

堆栈指针（SP）是寄存器 R13。在线程模式中，CONTROL 寄存器的 bit[1]指示了堆栈指针的使用情况：

- 0 = **主堆栈指针**（MSP）。这是复位值。
- 1 = **进程堆栈指针**（PSP）。

复位时，处理器将地址 0x00000000 的值加载到 MSP 中。

（3）链接寄存器

链接寄存器（LR）是寄存器 R14。它保存子程序、函数调用和异常的返回信息。复位时，LR 的值不可知。

（4）程序计数器

程序计数器（PC）是寄存器 R15。它包含当前的程序地址。复位时，处理器将复位向量（地址：0x00000004）的值加载到 PC。值的 bit[0]复位时被加载到 EPSR 的 T 位，必须为 1。

（5）程序状态寄存器

程序状态寄存器（PSR）由下列 3 种寄存器组合而成：

- 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）

在 32 位的 PSR 中，这 3 个寄存器的位域分配互斥。PSR 的位域分配如下：

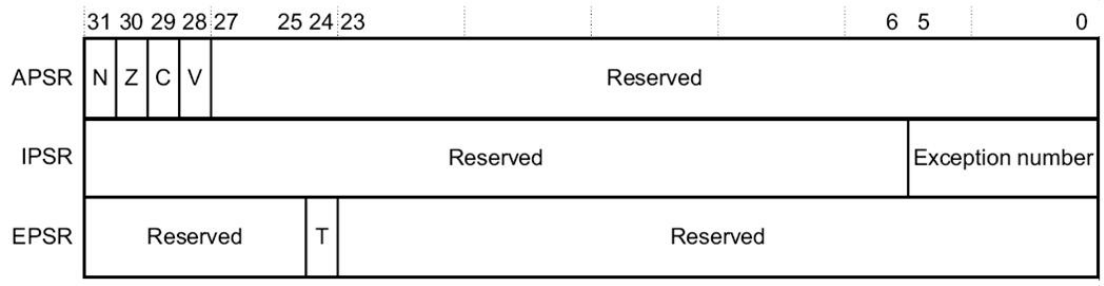

图 22.3 APSR、IPSR、EPSR 寄存器的位分配

这 3 个寄存器可以单独访问，也可以 2 个一组或 3 个一组进行访问，访问时，将寄存器名称作为 MSR 或 MRS 指令的一个变量。例如：

- 使用寄存器名称 PSR，用 MRS 指令来读所有寄存器
- 使用寄存器名称 APSR，用 MSR 指令来写 APSR

PSR 的组合和属性如下所示：

表 22.3 PSR 寄存器组合

寄存器	类型	组合
PSR	R/W[1][2]	APSR，EPSR 和 IPSR
IEPSR	RO	EPSR 和 IPSR
IAPSR	R/W[1]	APSR 和 IPSR
EAPSR	R/W[2]	APSR 和 EPSR

[1] 处理器忽略对 IPSR 位的写操作。

[2] 读 EPSR 位时返回零，处理器忽略对 EPSR 位的写操作。

有关访问程序状态寄存器的更多信息请参考本章“指令”一节中的描述。

应用程序状态寄存器：APSR 包含执行完前面的指令后条件标志的当前状态。有关寄存器的属性请见表 22.2。寄存器的位分配如下所示：

表 22.4 APSR 的位分配

位	名称	功能
[31]	N	负值标志
[30]	Z	零值标志
[29]	C	进位或借位标志
[28]	V	溢出标志
[27:0]	-	保留

有关 APSR 的负值、零值、进位或借位以及溢出标志的更多信息请参考本章“指令”一节。

中断程序状态寄存器：ISP 包含当前**中断服务程序**（ISR）的异常编号。有关寄存器的属性请见表 22.2。寄存器的位分配如下：

表 22.5 IPSR 的位分配

位	名称	功能
[31:6]	-	保留
[5:0]	异常编号	这是当前异常的编号: 0 = 线程模式 1 = 保留 2 = NMI 3 = HardFault 4-10 = 保留 11 = SVCall 12,13 = 保留 14 = PendSV 15 = SysTick 16 = IRQ0 . . . 47 = IRQ31 48-63 = 保留。 更多信息请见本章“异常类型”小节。

执行程序状态寄存器：EPSR 包含 Thumb 状态位。

有关 EPSR 属性请见表 22.2。EPSR 的位分配如下：

表 22.6 EPSR 的位分配

位	名称	功能
[31:25]	-	保留
[24]	T	Thumb 状态位
[23:0]	-	保留

如果应用软件使用 MRS 指令直接读取 EPSR 将始终返回零。利用 MSR 指令来写 EPSR 的操作会被忽略。故障处理程序可以检查入栈的 PSR 的 EPSR 值来确定故障的原因。请见本章“异常的进入和返回”小节。下面的操作可以清除 T~0 位：

[1] 指令 BLX，BX 和 POP{PC}

[2] 异常返回时恢复被压入栈中的 xPSR 值

[3] 进入异常时向量值的 bit[0]

在 T 位为 0 时尝试执行指令会导致 HardFault 或锁定故障，更多信息请见本章“锁定”小节。

可中断-可重启的指令：可中断-可重启的指令有 LDM 和 STM。如果在执行这两条中的其中一条指令的过程中出现中断，处理器就终止指令的执行。

在处理完中断后，处理器再从头开始重新执行指令。

（6）异常屏蔽寄存器

异常屏蔽寄存器禁止处理器处理异常。当异常可能影响到实时任务或要求连续执行的代码序列时，异常就被禁能。

可以使用 MSR 和 MRS 指令、或 CPS 指令改变 PRIMASK 的值来禁能或重新使能异常。更多信息本章“MRS”和“MSR”和“CPS”小节。

优先级屏蔽寄存器：PRIMASK 寄存器阻止优先级可配置的所有异常被激活。有关寄存器的属性请见表 22.2。寄存器的位分配如下：

表 22.7 PRIMASK 寄存器的位分配

位	名称	功能
[31:1]	-	保留
[0]	PRIMASK	0 = 无影响。 1 = 阻止优先级可配置的所有异常被激活。

（7）CONTROL 寄存器

CONTROL 寄存器控制着处理器处于线程模式时所使用的堆栈。寄存器的属性请见表 22.2。CONTROL 寄存器的位分配如下：

表 22.8 CONTROL 寄存器的位分配

位	名称	功能
[31:2]	-	保留
[1]	有效堆栈指针	定义当前的堆栈: 0 = MSP 是当前的堆栈指针 1 = PSP 是当前的堆栈指针 在处理器模式中，这个位读出为零，写操作被忽略。
[0]	-	保留

处理器模式始终使用 MSP，因此，在处理器模式下，处理器忽略对 CONTROL 寄存器的有效堆栈指针位执行的明确的写操作。异常进入和返回机制会将 CONTROL 寄存器更新。

在一个 OS 环境中，推荐运行在线程模式中的线程使用进程堆栈，内核和异常处理器使用主堆栈。

默认情况下，线程模式使用 MSP。要将线程模式中使用的堆栈指针切换成 PSP，只需要使用 MSR 指令将有效堆栈位设置为 1，见本章“MRS”小节。

备注：当更改堆栈指针时，软件必须在 MSR 指令后立刻使用一个 ISB 指令。这样来保证 ISB 之后的指令执行时使用新的堆栈指针，见本章“ISB”小节。

4. 异常和中断

Cortex-M0 处理器支持中断和系统异常。处理器和**嵌套向量中断控制器**（NVIC）划分所有异常的优先级，并对所有异常进行处理。一个中断或异常会改变软件控制的正常流程。处理器使用处理器模式来处理除复位之外的所有异常，更多信息见本章“异常的进入”小节和本章“异常返回”小节。

NVIC 寄存器控制中断处理。更多信息请见“嵌套向量中断控制器”小节。

5. 数据类型

处理器：

[1] 支持下列数据类型：

— 32 位字

— 16 位半字

— 8 位字节

[2] 管理所有数据存储器访问都采用小端模式。指令存储器和**专用外设总线**（PPB）访问始终是小端模式。更多信息请见本章“存储区、类型和属性”小节。

6. Cortex 微控制器软件接口标准

ARM 为编程 Cortex-M0 微控制器提供了 Cortex 微控制器软件接口标准（CMSIS）。CMSIS 是器件驱动库的一个组成部分。

CMSIS 为 Cortex-M0 微控制器系统定义了：

[1] 一种通用的方法来：

— 访问外设寄存器

— 定义异常向量

[2] 内核外设寄存器的名称和内核异常向量的名称

[3] 一个 RTOS 内核的器件独立的接口

CMSIS 包含 Cortex-M0 处理器中内核外设的地址定义和数据结构。还包含有组成 TCP/IP 堆栈和 Flash 文件系统的中间件元件的可选接口。

通过使能模板代码的重复使用以及将不同中间件厂商提供的符合 CMSIS 的软件组件组合起来，CMSIS 大大简化了整个软件开发过程。软件厂商可以扩展 CMSIS，使其包含各个厂商的外设定义以及这些外设的访问函数。

本文档包含了 CMSIS 定义的寄存器名称，并对处理器内核和内核外设相关的 CMSIS 函数进行了简单描述。

备注：本文档使用 CMSIS 定义的寄存器缩略名称。在某些情况下，这些名称与其它文档中可能用到的结构缩略名称不同。

下面各节给出了有关 CMSIS 的更多信息：

[1] 本章“电源管理编程提示”小节；

[2] 本章“内部函数”小节；

[3] 本章“使用 CMSIS 访问和 Cortex-M0 NVIC 寄存器”小节；

[4] 本章“NVIC 编程提示”小节。

22.3.2 存储器模型

本节描述处理器存储器映射以及存储器访问的行为。处理器有一个固定的存储器映射，提供有高达 4GB 的可寻址存储空间。存储器映射是：

图 22.4 Cortex-M3 存储器映射

处理器为内核外设寄存器保留了**专用外设总线**（PPB）地址范围空间，见本章的“Cortex-M0 的处理器和内核外设”一节。

1. 存储区、类型和属性

存储器映射分成多个区域。每个区域有一个定义好的存储器类型，某些区域还有附加的存储器属性。存储器类型和属性决定了各个区域的访问行为。

存储器类型是：

常规存储器 — 处理器为了提高效率，可以重新对交易进行排序，或者刻意地进行读取。

Device 存储器 — 处理器保护与 Device 或强秩序存储器（Strong-ordered memory）的其它交易相关的交易秩序。

强秩序存储器 — 处理器保护与所有其他交易相关的交易秩序。

Device 存储器和强秩序存储器的不同秩序要求意味着，存储器系统可以缓冲一个对 Device 存储器的写操作，但不准缓冲对强秩序存储器的写操作。

附加的存储器属性包括：

永不执行（XN） — 表示处理器阻止指令访问。当执行从存储器的 XN 区提取出来的指令时，产生一个 HardFault 异常。

2. 存储器系统的存储器访问秩序

对于大多数由明确的存储器访问指令引发的存储器访问，存储器系统都不保证访问秩序与指令的编写顺序完全一致，只要所有访问秩序的重新安排不影响指令序列的操作就行。一般情况下，如果两个存储器访问的顺序必须与两条存储器访问指令编写的顺序完全一致程序才能正确执行，软件就必须在两条存储器访问指令之间插入一条内存屏障指令，见本章“软件的存储器访问秩序”小节。

但是，存储器系统不保证 Device 存储器和强秩序存储器的一些访问秩序。对于两条存储器访问指令 A1 和 A2，如果 A1 的编写顺序在前，两条指令所引发的存储器访问顺序为：

A1 \ A2	常规访问	器件访问		强秩序访问
		不可共享	可共享	
常规访问	-	-	-	-
器件访问，不可共享	-	<	-	<
器件访问，可共享	-	-	<	<
强秩序访问	-	<	<	<

图 22.5 存储器排序限制

在表中：

- — 表示存储器系统不保证访问秩序。

< — 表示观察到访问顺序与指令编写顺序一致，即，A1 总是在 A2 之前。

3. 存储器访问的行为

存储器映射中每个区域的访问行为如下：

表 22.9 存储器访问的行为

地址范围	存储区域	存储器类型[1]	XN[1]	描述
0x00000000-0x1FFFFFFF	Code	常规存储器	-	程序代码的可执行区域。也可以把数据保存到这里
0x20000000-0x3FFFFFFF	SRAM	常规存储器	-	数据的可执行区域。也可以把代码保存到这里
0x40000000-0x5FFFFFFF	外设	Device 存储器	XN	外部设备存储器
0x60000000-0x9FFFFFFF	外部 RAM	常规存储器	-	数据的可执行区域
0xA0000000-0xDFFFFFFF	外部设备	Device 存储器	XN	外部设备存储器
0xE0000000-0xE00FFFFF	专用外设总线	强秩序存储器	XN	这个区域包括 NVIC、系统定时器和系统控制块。这个区域只能使用字访问
0xE0100000-0xFFFFFFFF	Device	Device 存储器	XN	厂商提供的特定存储器

[1] 更多信息请见本章“存储区、类型和属性”小节。

Code、SRAM 和外部 RAM 区域可以保存程序。

4. 软件的存储器访问秩序

程序流程的指令顺序不能担保相应的存储器交易顺序。这是因为：

[1] 为了提高效率，处理器可以将一些存储器访问的顺序重新安排，只要不影响指令序列的操作就行；
[2] 存储器映射中的存储器或设备可能有不同的等待状态；
[3] 某些存储器访问被缓冲，或者是刻意为之的。

“存储器系统的存储器访问秩序”小节描述了存储器系统在哪些情况下能保证存储器访问的秩序。但是，如果存储器访问的秩序十分重要，软件就必须插入一些内存屏障指令来强制保持存储器访问的秩序。处理器提供了以下内存屏障指令：

DMB — **数据存储器屏障**（DMB）指令保证先完成未处理的存储器交易，再执行后面的存储器交易，见本章“DMB”小节。

DSB — **数据同步屏障**（DSB）指令保证先完成未处理的存储器交易，再执行后面的指令，见本章“DSB”小节。

ISB — **指令同步屏障**（ISB）保证所有已完成的存储器交易的结果后面的指令都能辨认出来，见本章“ISB”小节。

下面是内存屏障指令使用的一些例子：

向量表 — 如果程序改变了向量表中的一项，然后又使能了相应的异常，那么就在两个操作之间插入一条 DMB 指令。这就确保了，如果异常在被使能后立刻被采纳，处理器能使用新的异常向量。

自修改代码 — 如果一个程序包含自修改代码，代码修改之后在程序中立刻使用一条 ISB 指令。这就确保了后面的指令执行使用的是更新后的程序。

存储器映射切换 — 如果系统包含一个存储器映射切换机制，在切换存储器映射之后使用一条 DSB 指令。这就确保了后面的指令执行使用的是更新后的存储器映射。

对强秩序存储器（例如，系统控制块）执行的存储器访问不需要使用 DMB 指令。

处理器保护与所有其他交易相关的交易秩序。

5. 存储器的字节存储顺序（Memory endianness）

处理器看到的存储器是一个从零开始、编号逐次递增的字节集合。例如，字节 0~3 存放第一个保存的字，字节 4~7 存放第二个保存的字。“（1）小端格式”小节描述了数据的字在存储器中是如何存放的。

（1）小端格式

在小端格式中，处理器将字的**最低有效字节**（lsbyte）保存在编号最小的字节中，**最高有效字节**（msbyte）保存在编号最大的字节中。例如：

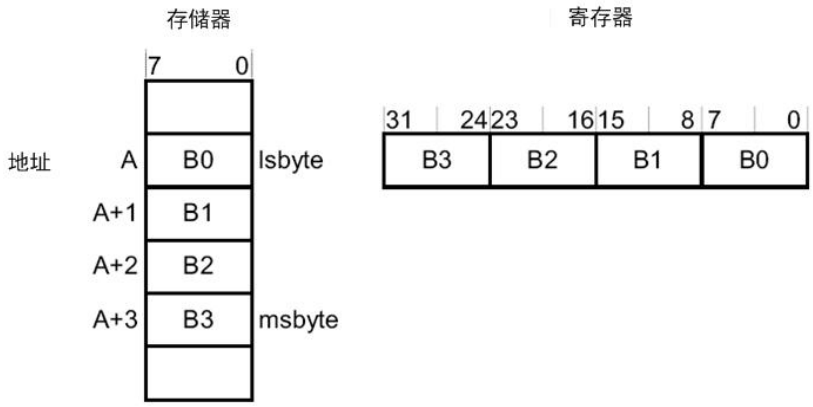

图 22.6 小端格式

22.3.3 异常模型

本节描述异常模型。

1. 异常状态

每个异常都处于下面其中一种状态：

无效 — 异常无效，未挂起。

挂起 — 异常正在等待处理器处理。

一个外设或软件的中断请求可以改变相应的挂起中断的状态。

有效 — 一个异常正在被处理器处理，但处理尚未结束。

一个异常处理程序可以中止另一个异常处理程序的执行。在这种情况下，两个异常都处于有效状态。

有效和挂起 — 异常正在被处理器处理，而且有一个来自同一个异常源的异常正在等待处理。

2. 异常类型

异常类型有：

备注：LPC111x 没有 NMI。

复位 — 复位在上电或热复位时启动。异常模型将复位当做一种特殊形式的异常来对待。当复位产生时，处理器的操作停止（可能停止在一条指令的任何一点上）。当复位撤销时，从向量表中复位项提供的地址处重新起动执行。执行在线程模式下重新起动。

NMI — 一个**不可屏蔽中断**（NMI）可以由外设产生，也可以由软件来触发。这是除复位之外优先级最高的异常。NMI 永远使能，优先级固定为 2。NMI 不能：

[1] 被屏蔽，它的执行也不能被其他任何异常中止；
[2] 被除复位之外的任何异常抢占。

HardFault — HardFault 是由于在正常操作过程中或在异常处理过程中出错而出现的一个异常。HardFault 的优先级固定为-1，表明它的优先级要高于任何优先级可配置的异常。

SVCall — **管理程序调用**（SVC）异常是一个由 SVC 指令触发的异常。在 OS 环境下，应用程序可以使用 SVC 指令来访问 OS 内核函数和器件驱动。

PendSV — PendSV 是一个中断驱动的系统级服务请求。在 OS 环境下，当没有其它异常有效时，使用 PendSV 来进行任务切换。

SysTick — SysTick 是一个系统定时器到达零时产生的异常。软件也可以产生一个 SysTick 异常。在 OS 环境下，处理器可以将这个异常用作系统节拍。

中断（IRQ） — 中断（或 IRQ）是外设发出的一个异常，或者是由软件请求产生的一个异常。所有中断都与指令执行不同步。在系统中，外设使用中断来与处理器通信。

表 22.10 各种异常类型的特性

异常编号[1]	IRQ 编号[1]	异常类型	优先级	向量地址[2]
1	-	复位	-3，优先级最高	0x00000004
2	-14	NMI	-2	0x00000008
3	-13	HardFault	-1	0x0000000C
4-10	-	保留	-	-
11	-5	SVCall	可配置[3]	0x0000002C
12-13	-	保留	-	-
14	-2	PendSV	可配置[3]	0x00000038
15	-1	SysTick	可配置[3]	0x0000003C
16 和大于 16 的值	0 和大于 0 的值	中断（IRQ）	可配置[3]	0x00000040 以及更高的地址[4]

[1] 为了简化软件层，CMSIS 只使用 IRQ 编号，因此，对除中断外的其他异常都使用负值。IPSR 返回异常编号，请见表 22.5。

[2] 更多信息请见“向量表”小节。

[3] 见“中断优先级寄存器”小节。

[4] 地址值以 4 为步长，逐次递增。

对于除复位之外的异步异常，在异常被触发和处理器进入异常处理程序之间，处理器可以执行条件指令。

被特许的软件可以将表 22.10 中列出的优先级可配置的异常禁能，见本章“中断清除-使能寄存器”小节。

有关 HardFault 的更多信息，请见本章“故障处理”小节。

3. 异常处理程序

处理器使用以下处理程序来处理异常：

中断服务程序（ISR） — 中断 IRQ0~IRQ31 是由 ISR 来处理的异常。

故障处理程序 — HardFault 是唯一一个由故障处理程序来处理的异常。

系统处理程序 — NMI、PendSV，SVCall、SysTick 和 HardFault 是由系统处理程序来处理的全部异常。

4. 向量表

向量表包含堆栈指针的复位值以及所有向量处理程序的起始地址（也称为异常向量）。图 22.7 显示了异常向量在向量表中的放置顺序。每个向量的最低有效位必须为 1，表明异常处理程序都是用 Thumb 代码编写的。

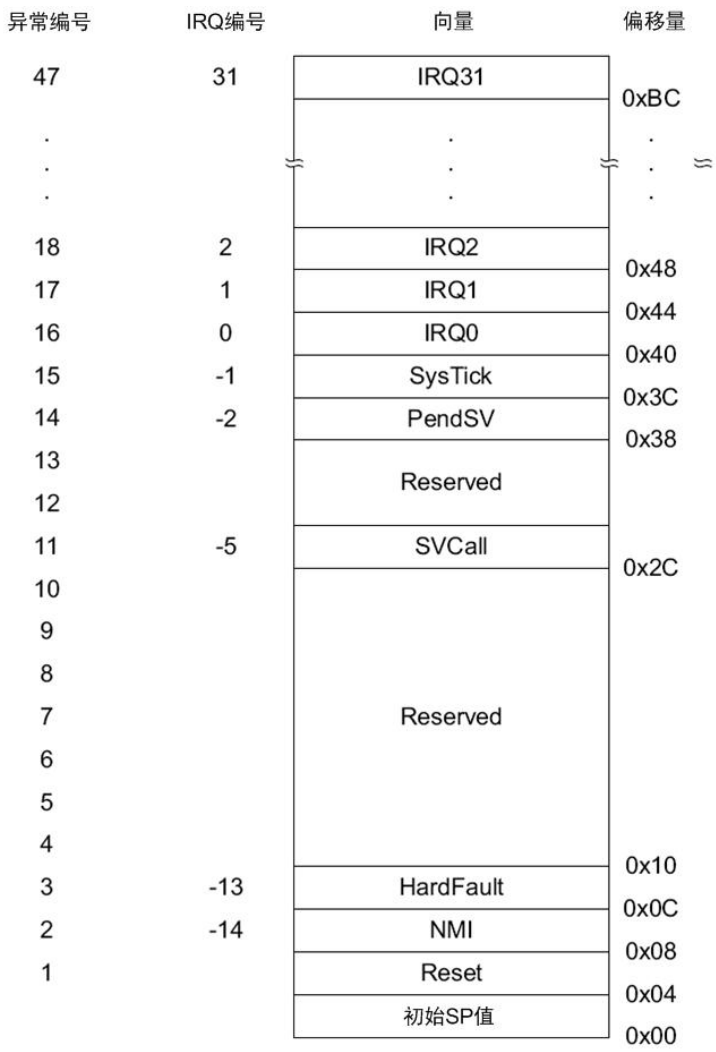

图 22.7 向量表

向量表的地址固定为 0x00000000。

5. 异常优先级

如表 22.10 所示，每个异常都有对应的优先级：

[1] 越小的优先级值指示一个更高的优先级;

[2] 除复位、HardFault 和 NMI 之外，所有异常的优先级都是可配置的;

如果软件不配置任何优先级，那么，所有优先级可配置的异常的优先级就都为 0。有关配置异常优先级的信息请见：

[1] 本章“系统处理程序优先级寄存器”小节

[2] 本章“中断优先级寄存器”小节

优先级值的配置范围为 0 – 192，各值以 64 为间距。复位、HardFault 和 NMI 这些有固定的负优先级值的异常的优先级高于任何其他的异常。

给 IRQ[0]分配一个高优先级值、给 IRQ[1]分配一个低优先级值就意味着 IRQ[1]的优先级高于 IRQ[0]。如果 IRQ[1]和 IRQ[0]都有效，先处理 IRQ[1]。

如果多个挂起的异常具有相同的优先级，异常编号越小的挂起异常优先处理。例如，如果 IRQ[0]和 IRQ[1]正在挂起，并且两者的优先级相同，那么先处理 IRQ[0]。

当处理器正在执行一个异常处理程序时，如果出现一个更高优先级的异常，那么这个异常就被抢占。如果出现的异常的优先级和正在处理的异常的优先级相同，这个异常就不会被抢占，与异常的编号大小无关。但是，新中断的状态就变为挂起。

6. 异常的进入和返回

描述异常处理时使用了下列术语：

抢占 — 当处理器正在执行一个异常处理程序时，如果一个异常的优先级比正在处理的异常的优先级更高，那么低优先级的异常就被抢占。

当一个异常抢占另一个异常时，这些异常就被称为嵌套异常。更多信息请见本章“异常的进入”小节。

返回 — 当异常处理程序结束，并且满足以下条件时，异常就返回。

[1] 没有优先级足够高的挂起异常要处理

[2] 已结束的异常处理程序没有在处理一个迟来的异常

处理器弹出堆栈，处理器状态恢复到中断出现之前的状态，更多信息请见“（2）异常返回”小节。

末尾连锁 — 这个机制加速了异常的处理。当一个异常处理程序结束时，如果一个挂起的异常满足异常进入的要求，就跳过堆栈弹出，控制权移交给新的异常处理程序。

迟来（Late-arriving） — 这个机制加速了抢占的处理。如果一个高优先级的异常在前一个异常正在保存状态的过程中出现，处理器就转去处理更高优先级的异常，开始提取这个异常的向量。状态保存不受迟来异常的影响，因为两个异常保存的状态相同。从迟来异常的异常处理程序返回时，要遵守正常的末尾连锁规则。

（1）异常的进入

当有一个优先级足够高的挂起异常存在，并且满足下面的任何一个条件，就进入异常处理：

[1] 处理器处于线程模式

[2] 新异常的优先级高于正在处理的异常，这时，新异常就抢占了正在处理的异常

当一个异常抢占了另一个异常时，异常就被嵌套。

优先级足够高的意思是该异常的优先级比屏蔽寄存器中所限制的任何一个异常组的优先级都要高，见本章“异常屏蔽寄存器”小节。优先级比这个异常要低的异常被挂起，但不被处理器处理。

当处理器处理异常时，除非异常是一个末尾连锁异常或迟来的异常，否则，处理器都把信息压入到当前的堆栈中。这个操作被称为**入栈**（stacking），8 个数据字的结构被称为**栈帧**（stack frame）。栈帧包含以下信息：

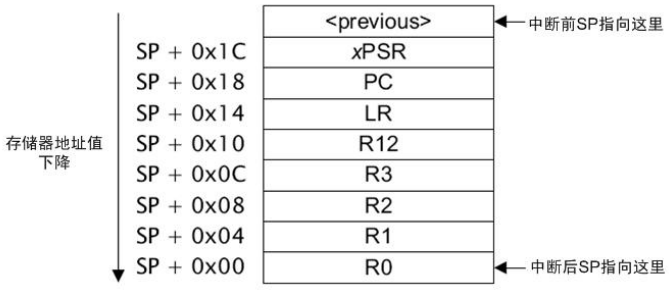

图 22.8 异常进入时堆栈的内容

入栈后，堆栈指针立刻指向栈帧的最低地址单元。栈帧按照双字地址对齐。

栈帧包含返回地址。这是被中止的程序中下条指令的地址。这个值在异常返回时返还给 PC，使被中止的程序恢复执行。

处理器执行一次向量提取，从向量表中读出异常处理程序的起始地址。当入栈结束时，处理器开始执行异常处理程序。同时，处理器向 LR 写入一个 EXC_RETURN 值。这个值指示了栈帧对应哪个堆栈指针以及在异常出现之前处理器处于什么工作模式。

如果在异常进入的过程中没有更高优先级的异常出现，处理器就开始执行异常处理程序，并自动将相应的挂起中断的状态变为有效。

如果在异常进入的过程中有另一个优先级更高的异常出现，处理器就开始执行这个高优先级异常的异常处理程序，不改变前一个异常的挂起状态。这是一个迟来异常的情况。

（2）异常返回

当处理器处于处理器模式并且执行下面其中一条指令尝试将 PC 设为 EXC_RETURN 值时，出现异常返回：

[1] POP 指令，用来加载 PC

[2] BX 指令，用来使用任意的寄存器

在异常进入时处理器将一个 EXC_RETURN 值保存到 LR 中。异常机制依靠这个值来检测处理器何时执行完一个异常处理程序。EXC_RETURN 值的 bit[31:4]为 0xFFFFFFF。当处理器将一个相应的这种形式的值加载到 PC 时，它将检测到这个操作并不是一个正常的分支操作，而是异常已经结束。因此，处理器启动异常返回序列。EXC_RETURN 的 bit[3:0]指出了所需的返回堆栈和处理器模式，如表 22.11 所示。

表 22.11 异常返回的行为

EXC_RETURN	描述
0xFFFFFFF1	返回到处理器模式 异常返回获得主堆栈的状态 返回后执行使用 MSP
0xFFFFFFF9	返回到线程模式 异常返回获得 MSP 的状态 返回后执行使用 MSP
0xFFFFFFFD	返回到线程模式 异常返回获得 PSP 的状态 返回后执行使用 PSP
所有其他值	保留

22.3.4 故障处理

故障是异常的一个子集，见 22.3.3 节。所有的故障都导致 HardFault 异常被处理，或者，如果故障在 NMI 或 HardFault 处理程序中出现，会导致锁定。发生以下情况会导致出现故障：

[1] 在一个优先级等于或高于 SVCall 的地方执行 SVC 指令；

[2] 在没有调试器的情况下执行 BKPT 指令；

[3] 在加载或存储时出现一个系统产生的总线错误；

[4] 执行一个 XN 存储器地址中的指令；

[5] 从系统产生了一个总线故障的地址单元中执行指令；

[6] 在提取向量时出现了一个系统产生的总线错误；

[7] 执行一个未定义的指令；

[8] 由于 T 位之前被清零而导致不再处于 Thumb 状态的情况下执行一条指令；

[9] 尝试对一个不对齐的地址执行加载或存储操作。

只有复位和 NMI 可以抢占优先级固定的 HardFault 处理程序。HardFault 可以抢占除复位、NMI 或其他硬故障之外的任何异常。

1. 锁定

如果在执行 NMI 或 HardFault 处理程序时出现故障，或者，在一个使用 MSP 的异常返回时出栈的却是 PSR 的时候系统产生一个总线错误，处理器进入一个锁定状态。当处理器处于锁定状态时，它不执行任何指令。处理器保持处于锁定状态，直到下面任何一种情况出现：

[1] 出现复位；

[2] 调试器将锁定状态终止；

[3] 出现一个 NMI，以及当前的锁定处于 HardFault 处理程序中。

如果锁定状态出现在 NMI 处理程序中，后面的 NMI 就无法使处理器离开锁定状态。

22.3.5 电源管理

Cortex-M0 处理器的睡眠模式可以降低功耗，睡眠模式包含 2 种：

[1] 睡眠模式：停止处理器时钟

[2] 深度睡眠模式（见第 3 章“电源管理”小节）

SCR 的 SLEEPDEEP 位选择使用哪种睡眠模式，见本章“系统控制寄存器”小节。

本节描述了进入睡眠模式的机制和将器件从睡眠模式唤醒的条件。

1. 进入睡眠模式

本节描述了软件可以用来使处理器进入睡眠模式的一种机制。

系统可以产生伪唤醒事件，例如，一个调试操作唤醒处理器。因此，软件必须能够在这样的事件之后使处理器重新回到睡眠模式。程序中可以有一个空闲循环使得器件回到睡眠模式。

（1）等待中断

等待中断指令（WFI）使器件立刻进入睡眠模式。当执行一个 WFI 指令时，处理器停止执行指令，进入睡眠模式。更多信息见本章“WFI”小节。

（2）等待事件

备注：WFE 指令不能在 LPC111x 上使用。

等待事件指令（WFE）根据一个一位的事件寄存器的值来进入睡眠模式。处理器执行一个 WFE 指令时检查事件寄存器的值：

0 — 处理器停止执行指令，进入睡眠模式

1 — 处理器将寄存器的值设为 0，并继续执行指令，不进入睡眠模式

更多信息请见本章“WFE”小节。

如果事件寄存器为 1，表明处理器在执行 WFE 指令时不必进入睡眠模式。通常的原因是出现了一个外部事件，或者系统中的另一个处理器已经执行了 SEV 指令，见本章“SEV”小节。软件不能直接访问这个寄存器。

（3）Sleep-on-exit

如果 SCR 的 SLEEPONEXIT 位被设为 1，当处理器完成一个异常处理程序的执行并返回到线程模式时，处理器立刻进入睡眠模式。如果应用只要求处理器在中断出现时运行，就可以使用这种机制。

2. 从睡眠模式唤醒

处理器的唤醒条件取决于使处理器进入睡眠模式所采用的机制。

（1）从 WFI 或 sleep-on-exit 唤醒

通常，只有当检测到一个优先级足够高的异常导致进入异常处理时，处理器才唤醒。

某些嵌入式系统在处理器唤醒之后可能必须先执行系统恢复任务，然后再执行中断处理程序。通过将 PRIMASK 位置位来实现这个操作。如果到来的中断被使能，并且优先级高于当前的异常优先级，处理器就唤醒，但不执行中断处理程序，直至处理器将 PRIMASK 设为 0，见本章“异常屏蔽寄存器”小节。

（2）从 WFE 唤醒

如果出现以下情况，处理器就唤醒：

[1] 处理器检测到一个优先级足够高的异常导致进入异常处理

[2] 在一个多处理器的系统中，系统中的另一个处理器执行了 SEV 指令

另外，如果 SCR 的 SEVONPEND 位被设为 1，那么任何新的挂起中断都能触发一个事件和唤醒处理器，即使这个中断被禁能，或者这个中断的优先级不够高而导致无法进入异常处理。有关 SCR 的更多信息请见本章“系统控制寄存器”小节。

3. 电源管理编程提示

ISO/IEC C 不能直接产生 WFI、WFE 和 SEV 指令。CMSIS 为这些指令提供了以下内在函数：

```
void __WFE(void)                    /* 等待事件   */
void __WFI(void)                    /* 等待中断   */
void __SEV(void)                    /* 发送事件   */
```

22.4 指令集

22.4.1 指令集汇总

处理器执行这种版本的指令集。表 22.12 列出了所支持的指令。

表 22.12 的备注如下：

- 尖括号< >括着操作数的备用格式;
- 大括号{}括着可选的操作数和助记符部分;
- 操作数列所列出的操作数不完全。

有关指令和操作数的信息，详见指令描述。

表 22.12 Cortex-M0 指令

助记符	操作数	简述	标志
ADCS	{Rd,} Rn, Rm	进位加法	N,Z,C,V
ADD{S}	{Rd,} Rn,<Rm|#imm>	加法	N,Z,C,V
ADR	Rd, label	将基于 PC 相对偏移的地址读到寄存器	-
ANDS	{Rd,} Rn, Rm	位与操作	N,Z
ASRS	{Rd,}Rm, <Rs|#imm>	算术右移	N,Z,C
B{cc}	label	跳转{有条件}	-
BICS	{Rd,} Rn, Rm	位清除	N,Z
BKPT	#imm	断点	-
BL	label	带链接的跳转	-
BLX	Rm	带链接的间接跳转	-
BX	Rm	间接跳转	-
CMN	Rn, Rm	比较负值	N,Z,C,V
CMP	Rn, <Rm|#imm>	比较	N,Z,C,V
CPSID	i	更改处理器状态，关闭中断	-
CPSIE	i	更改处理器状态，使能中断	-
DMB	-	数据内存屏障	-
DSB	-	数据同步屏障	-
EORS	{Rd,} Rn, Rm	异或	N,Z
ISB	-	指令同步屏障	-
LDM	Rn{!}, reglist	加载多个寄存器，访问之后会递增地址	-
LDR	Rt, label	从基于 PC 相对偏移地址上加载寄存器	-
LDR	Rt, [Rn, <Rm|#imm>]	用字加载寄存器	-
LDRB	Rt, [Rn, <Rm|#imm>]	用字节加载寄存器	-

续上表

助记符	操作数	简述	标志
LDRH	Rt, [Rn, <Rm|#imm>]	用半字加载寄存器	-
LDRSB	Rt, [Rn, <Rm|#imm>]	用有符号的字节加载寄存器	-
LDRSH	Rt, [Rn, <Rm|#imm>]	用有符号的半字加载寄存器	-
LSLS	{Rd,} Rn, <Rs|#imm>	逻辑左移	N,Z,C
U	{Rd,} Rn, <Rs|#imm>	逻辑右移	N,Z,C
MOV{S}	Rd, Rm	传输	N,Z
MRS	Rd, spec_reg	从特别寄存器传输到通用寄存器	-
MSR	spec_reg, Rm	从通用寄存器传输到特别寄存器	N,Z,C,V
MULS	Rd, Rn, Rm	乘法，32 位结果值	N,Z
MVNS	Rd, Rm	位非	N,Z
NOP	-	无操作	-
ORRS	{Rd,} Rn, Rm	逻辑或	N,Z
POP	reglist	出栈，将堆栈的内容放入寄存器	-
PUSH	reglist	压栈，将寄存器的内容压入堆栈	-
REV	Rd, Rm	反转字时面的字节顺序	-
REV16	Rd, Rm	反转每半字里面的字节顺序	-
REVSH	Rd, Rm	反转有符号半字里面的字节顺序	-
RORS	{Rd,} Rn, Rs	循环右移	N,Z,C
RSBS	{Rd,} Rn, #0	反向减法	N,Z,C,V
SBCS	{Rd,} Rn, Rm	进位减法	N,Z,C,V
SEV	-	发送事件	-
STM	Rn!, reglist	存储多个寄存器，在访问后地址递增	-
STR	Rt, [Rn, <Rm|#imm>]	将寄存器作为字来存储	-
STRB	Rt, [Rn, <Rm|#imm>]	将寄存器作为字节来存储	-
STRH	Rt, [Rn, <Rm|#imm>]	将寄存器作为半字来存储	-
SUB{S}	{Rd,} Rn,<Rm|#imm>	减法	N,Z,C,V
SVC	#imm	超级用户调用	-
SXTB	Rd, Rm	符号扩展字节	-
SXTH	Rd, Rm	符号扩展半字	-
TST	Rn, Rm	基于测试的逻辑与	N,Z
UXTB	Rd, Rm	0 扩展字节	-
UXTH	Rd, Rm	0 扩展半字	-

续上表

助记符	操作数	简述	标志
WFE	-	等待事件	-
WFI	-	等待中断	-

22.4.2 内部函数

ISO/IEC C 代码不能直接访问某些 Cortex-M0 指令。本章节对可以产生这些指令的内部函数进行了描述，内部函数可由 CMSIS 或有可能由 C 编译器提供。若 C 编译器不支持相关的内部函数，则用户可能需要使用内联汇编程序来访问相关的指令。

CMSIS 提供下列的内部函数来产生 ISO/IEC C 代码不能直接访问的指令：

表 22.13 产生某些 Cortex-M0 指令的 CMSIS 内部函数

指令	CMSIS 指令集
CPSIE i	void __enable_irq(void)
CPSID i	void __disable_irq(void)
ISB	void __ISB(void)
DSB	void __DSB(void)
DMB	void __DMB(void)
NOP	void __NOP(void)
REV	uint32_t __REV(uint32_t int value)
REV16	uint32_t __REV16(uint32_t int value)
REVSH	uint32_t __REVSH(uint32_t int value)
SEV	void __SEV(void)
WFE	void __WFE(void)
WFI	void __WFI(void)

CMSIS 还提供使用 MRS 和 MSR 指令来访问特别寄存器的函数：

表 22.14 访问特别寄存器的内部函数

特定寄存器	访问	CMSIS 函数
PRIMASK	读	uint32_t __get_PRIMASK (void)
	写	void __set_PRIMASK (uint32_t value)
CONTROL	读	uint32_t __get_CONTROL (void)
	写	void __set_CONTROL (uint32_t value)
MSP	读	uint32_t __get_MSP (void)
	写	void __set_MSP (uint32_t TopOfMainStack)
PSP	读	uint32_t __get_PSP (void)
	写	void __set_PSP (uint32_t TopOfProcStack)

22.4.3 关于指令的描述

下列小节对如何使用指令进行了更为详细的描述：

[1] 操作数；
[2] 使用 PC 或 SP 时的限制；
[3] 移位操作；
[4] 地址对齐；
[5] PC 相对的表达式；

[6] 条件执行。

1. 操作数

指令操作数可以是 ARM 寄存器，常量或其它的指令特定参数。指令在操作数上操作，并经常将结果存放在目的寄存器中。当指令中存在目的寄存器时，它通常会在其它操作数之前被指定。

2. 使用 PC 或 SP 时的限制

对于用于操作数或目的寄的程序计数器（PC）或堆栈指针，许多指令都不能使用它们，或者存在着用户能否使用它们的限制。更多信息，详见指令的描述。

备注：当使用 BX、BLX 或 POP 指令来更新 PC 时，为正确执行程序，任何地址的位 0 都必须为 1.这是因为该位指示目标指令集，且 Cortex-M0 处理器只支持 Thumb 指令。当 BL 或 BLX 指令将位 0 的值写入 LR 时，值 1 会被自动分配。

3. 移位操作

寄存器移位操作通过特定的位数（移位长度）来实现寄存器位的左右移位操作。寄存器移位可以由指令 ASR、LSR、LSL 和 ROR 直接执行，且结果会被写入到目的寄存器中。允许的移位长度由移位类型和指令决定，请参考各个指令的描述。若移位长度为 0，则不发生移位操作。寄存器移位操作会更新进位标志，当移位长度被指定为 0 时除外。下列各子节描述了各种的移位操作以及它们是如何影响进位标志。在这些描述中，Rm 是包含着移位值的寄存器，n 是移位长度。

（1）ASR

算术右移 n 位的操作是将 Rm 寄存器左边的 32-n 个位向右移动 n 位，结果是寄存器右边有 32-n 个位，然后再将寄存器位[31]的原始值复制到结果寄存器左边的 n 个位中。请参考图 22.9。

用户可以使用 ASR 对寄存器 Rm 的符号值进行 2^n 除法操作，得到的结果为负无穷大。

当指令为 ASRS 时，进位标志会被更新为最后移出的位值，即寄存器 Rm 的位[n-1]。

备注：

- 如果 n 为 32 或大于 32，那么结果中的所有位都会被置为 Rm 中位[31]的值。
- 如果 n 为 32 或大于 32，那么进位标志被更新为 Rm 位[31]的值。

图 22.9 ASR #3

（2）LSR

逻辑右移 n 位的操作是将 Rm 寄存器 Rm 左边的 32-n 个位向右移动 n 位，结果是寄存器右边有 32-n 个位，然后再将结果寄存器左边的 n 个位设为 0。请参考图 22.10。

用户可以使用 LSR 操作来对寄存器 Rm 的值进行 2^n 除法操作，如果值为无符号的整数。

当指令为 LSRS 时，进位标志会被更新为最后移出的位值，即寄存器 Rm 的位[n-1]。

备注：

- 如果 n 为 32 或大于 32，那么结果中的所有位都会被清除为 0。
- 如果 n 为 33 或大于 33，那么进位标志被更新为 0。

图 22.10 LSR #3

（3）LSL

逻辑左移 n 位的操作是将 Rm 寄存器右边的 32-n 个位向左移动 n 位，结果是寄存器左边有 32-n 个位，然后将结果中的寄存器右边的 n 个位设为 0。请参考图 22.11。

用户可以使用 LSL 操作来对寄存器 Rm 的值与 2^n 进行乘法操作，如果值为无符号的整数或是有符号 2 的补码整数值。溢出会在无警告的提示下发生。

当指令为 LSLS 时，进位标志会被更新为最后移出的位值，即寄存器 Rm 的位[32-n]。当使用 LSL #0 时，这些指令不会影响进位标志。

备注：

- 如果 n 为 32 或大于 32，那么结果中的所有位都会被清除为 0。
- 如果 n 为 33 或大于 33，那么进位标志被更新为 0。

图 22.11 LSL #3

（4）ROR

循环右移 n 位的操作是将 Rm 寄存器左边的 32-n 个位向右移动 n 位，结果是寄存器右边有 32-n 个位，然后再将寄存器右边的 n 个位移动到结果寄存器的左边的 n 个位中。请参考图 22.12。

当指令为 RORS 时，进位标志会被更新为最后循环出的位值，即寄存器 Rm 的位[n-1]。

备注：

- 如果 n 为 32，那么结果中的所有位值都相同于 Rm 中的值，且如果进位标志被更新，则会被更新为 Rm 的位[31]的值。
- 移位长度 n 大于 32 的 ROR 与移位长度为 n-32 的 ROR 操作得到的结果相同。

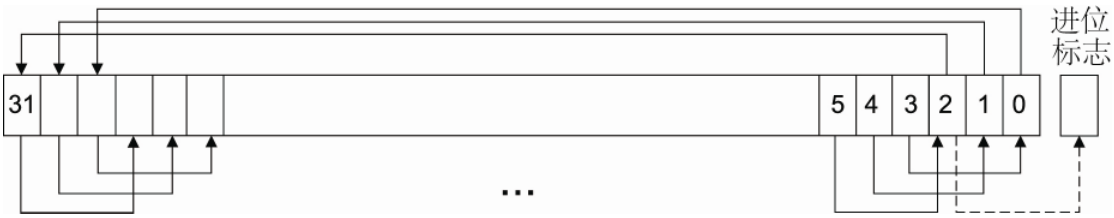

图 22.12 ROR #3

4. 地址对齐

对齐访问是这样的一个操作：字对齐地址是用于字或多字访问，或者半字对齐地址是用于半字访问。字节访问通常是对齐访问的。

Cortex-M0 处理器不支持非对齐地址的访问。任何尝试执行一个非对齐的存储器访问操作都会导致 HardFault 异常。

5. PC 相对的表达式

相对 PC 表达或标签是一个代表着指令或文字数据的地址的符号。在指令中它被表示为 PC 值加上或减去一个数字偏移量。汇编器从标签和当前指令的地址中计算出所要求的偏移量。如果偏移量太大，则汇编器会产生一个错误。

备注:

- 对于大数多指令，PC 的值就是当前指令的地址加上 4 个字节;
- 汇编器可能允许用其它语法来表示 PC 相对表达式，如标签加上或减去一个数值，或者用[PC, #imm]格式表示。

6. 条件执行

大多数数据处理指令依据操作的结果在应用程序状态寄存器（APSR）中更新条件标志。某些指令更新所有标志，而某些指令则仅更新子集。如果标志不被更新，则原始值被保留。指令对标志的影响，请参考指令的描述。

在如下的情况里，用户可以在另一个指令中设置条件标志的基础上：

[1] 在指令更新标志后可立即执行条件性的跳转指令；
[2] 在经过任意数量的不会更新标志的间隔数指令后，可以执行条件性的跳性指令。

在 Cortex-M0 处理器上，通过使用条件性的跳转指令，就可以实现条件性的执行操作。

本小节描述了以下内容：

[1] 条件标志；
[2] 条件代码后缀。

（1）条件标志

APSR 包含了下列的条件标志：

N — 当操作的结果为负值时置为 1，否则清除为 0。

Z — 当操作的结果为 0 时置为 1，否则清除为 0。

C — 当操作的结果导致要进位时置为 1，否则清除为 0

V — 当操作引发溢出时置为 1，否则清除为 0。

有关 APSR 的信息，详情请参考本章“程序状态寄存器”小节。

当出现下列情况时，会发生进位操作：

[1] 如果加法的结果大于或等于 2^{32}；
[2] 如果减法的结果为正或等于 0；
[3] 由于移位指令或循环指令而发生的进位操作。

当位[31]中结果的符号值不与在无穷精度中所执行操作的结果符号值匹配时，溢出发生，例如：

[1] 如果二个负值相加得出一个正值；
[2] 如果二个正值相加得出一个负值；

[3] 如果从一个负值减去一个正值得到一个正值；

[4] 如果从一个正值减去一个负值得到一个负值。

对于 CMP，比较操作与减法操作相同，对于 CMN，则加法操作相同，结果值会被丢弃除外。更多信息，详情请参考指令描述。

（2）条件代码后缀

条件性跳转在语法描述显示为 B{cond}。只有 APSR 的条件代码标志符合指定的条件时，才能执行带有条件代码的跳转指令，否则要忽略跳转指令。表 22.15 显示了使用的条件代码，同时还显示了条件代码后缀和 N、Z、C 和 V 标志之关的联系。

表 22.15 条件代码后缀

后缀	标志	所表示的意思
EQ	Z = 1	等效，最后标志设置结果为 0
NE	Z = 0	不等效，最后标志设置结果为非 0
CS 或 HS	C = 1	更高或相同，无符号
CC 或 LO	C = 0	更低，无符号
MI	N = 1	负数
PL	N = 0	正数或 0
VS	V = 1	溢出
VC	V = 0	无溢出
HI	C = 1 和 Z = 0	更高，无符号
LS	C = 0 或 Z = 1	更低或相同，无符号
GE	N = V	大于或等效，有符号
LT	N != V	少于，有符号
GT	Z = 0 和 N = V	大于，有符号
LE	Z = 1 和 N != V	少于或等于，有符号
AL	可以为任意值	总是。当没有指定后缀时，这是默认的操作

22.4.4 存储器访问指令

表 22.16 所示为存储器访问指令：

表 22.16 访问指令

助记符	简述
LDR{type}	使用寄存器偏移量来加载寄存器
LDR	基于 PC 相对地址来加载寄存器
POP	出栈，将栈中的内容放入寄存器
PUSH	压栈，将寄存器的内容压入堆栈
STM	存储多个寄存器
STR{type}	使用立即数偏移量来存储寄存器
STR{type}	使用寄存器偏移量来存储寄存器

1. ADR

产生一个 PC 相对地址。

（1）语法

ADR Rd, label

其中：

Rd 是目标寄存器。

Label 是 PC 相对表达式。见本章“相对 PC 的表达式”小节。

（2）操作

ADR 通过将立即数值加入到 PC 中来产生一个地址，并将得到的地址结果写入到目的寄存器中。

ADR 产生区域独立代码非常便利，因为地址是 PC 相对地址。

如果用户使用 ADR 来产生 BX 或 BLX 指令的目标地址，为了能正确执行程序，必须要保证将产生的地址的位[0]设置为 1。

（3）限制

在该指令中，Rd 必须指定 R0-R7。地址数据值必须是字对齐，且不能超出当前 PC 的 1020 字节。

（4）条件标志

该指令不会改变标志。

（5）示例

```
ADR R1, TextMessage          ; 将被标签为 TextMessage 单元上的地址值写入到 R1 中
ADR   R3, [PC,#996]          ;将 R3 的值设为 PC + 996。
```

2. LDR 和 STR，立即数偏移量

具有立即数偏移量的加载和存储。

（1）语法

LDR Rt, [<Rn | SP> {, #imm}]

LDR<B|H> Rt, [Rn {, #imm}]

STR Rt, [<Rn | SP>, {,#imm}]

STR<B|H> Rt, [Rn {,#imm}]

其中：

Rt 是加载或存储的寄存器。

Rn 是基于存储器地址上的寄存器。

Imm 是 Rn 的偏移量。如果 imm 被省略，则假设它为 0。

（2）操作

LDR、LDRB 和 LDRH 指令将存储器中的字、字节或半字数据值加载到 Rt 指定的寄存器中。在将数据写入 Rt 指定的寄存器之前，长度少于字的数据要用 0 扩充到 32 位的长度。

STR、STRB 和 STRH 指令将 Rt 寄存器指定的单个寄存器中所包含的字，最低位字节或低半字存放到存储器中。从加载的存储器地址或用于存放的存储器地址是 Rn 或 SP 所指定的寄存器的值与立即数 imm 的和。

（3）限制

在这些指令中，

[1] Rt 和 Rn 必须只指定 R0-R7 的值；

[2] Imm 的值必须要符合下列要求：

- 0 到 1020 之间，对于 LDR 和 STR 操作，在将 SP 用作基址寄存器时，其值必须是 4 的整数倍；

- 0 到 124 之间，对于 LDR 和 STR 操作，在将 R0-R7 用作基址寄存器时，其值必须是 4 的整数倍；
- 0 到 62 之间，对于 LDRH 和 STRH 操作，其值必须是 2 的整数倍；
- 0 到 31 之间，对于 LDRB 和 STRB；

[3] 计算出的地址必须能够被事务中的字节数整除，见“地址对齐”小节。

（4）条件标志

这些指令不会改变标志。

（5）示例

```
LDR R4, [R7                    ;将 R7 的地址载入到 R4。
STR R2, [R0,#const-struc]      ; const-struc  是评估处于 0-1020 范围内的常量的表达式。
```

3. LDR 和 STR，寄存器偏移量

带寄存器偏移量的加载和存储。

（1）语法

LDR Rt, [Rn, Rm]

LDR<B|H> Rt, [Rn, Rm]

LDR<SB|SH> Rt, [Rn, Rm]

STR Rt, [Rn, Rm]

STR<B|H> Rt, [Rn, Rm]

其中：

Rt 是加载或存储的寄存器。

Rn 是基于存储器地址的寄存器。

Rm 是含有用作偏移量的值的寄存器。

（2）操作

LDR、LDRB、U、LDRSB 和 LDRSH 将存储器中的字、0 扩展字节、0 扩展半字、符号扩展字节或符号扩展半字加载到 Rt 指定的寄存器中。

STR、STRB 和 STRH 指令将 Rt 寄存器指定的单个寄存器中所包含的字，最低位字节或低半字存放到存储器中。

从加载的存储器地址或用于存放的存储器地址是 Rn 和 Rm 所指定的寄存器中的值之和。

（3）限制

在这些指令中，

[1] Rt、Rn 和 Rm 必须指定 R0-R7；

[2] 计算出的地址必须能够被加载或存储的字节数整除。见“地址对齐”小节。

（4）条件标志

这些指令不会改变标志。

（5）示例

```
STR R0, [R5, R1]          ; 将 R0 的值存储到 R5 加 R1 得出的地址中；
LDRSH   R1, [R2, R3]      ; 从(R2 + R3)所指定的存储器地址中加载半字数据，符号扩展到 32 位并将
                          ; 其写入到 R1 中。
```

4. LDR，PC 相对

从存储器中加载寄存器（文字数据）。

（1）语法

LDR Rt, label

其中:

Rt 是加载的寄存器。

Label 是 PC 相对表达式，见“相对 PC 的表达式”小节。

（2）操作

将 label 所指定的存储器中的字加载到 Rt 所指定的寄存器中。

（3）限制

在这些指令中，label 的大小必须位于当前 PC 的 1020 字节范围之内，且是字对齐的。

（4）条件标志

这些指令不会更改标志。

（5）示例

```
LDR R0, LookUpTable          ;将标签为 LookUpTable 的地址中的字数据加载到 R0 中；
LDR R3, [PC, #100]           ;将(PC + 100)上的存储器字加载到 R3 中。
```

5. LDM 和 STM

加载和存储多个寄存器。

（1）语法

LDM Rn{!}, reglist

STM Rn!, reglist

其中：

Rn 是基于存储器地址的寄存器。

!回写后缀。

reglist 是被加载或存储的一个或多个寄存器的列表，用大括号括住。它包含着寄存器范围。若它包含着多于一个的寄存器或寄存器范围，必须要将其用逗号隔开，见本小节的“示例”。

对于 LDM，LDMIA 和 LDMFD 相近。LDMIA 为每次访问后都会递增的基址寄存器。LDMFD 用法是将数据从满的递减堆栈中移出。

对于 STM，STMIA 和 STMEA 相近。STMIA 为每次访问后都会递增的基址寄存器。STMEA 用法是将数据压入空的递增堆栈中。

（2）操作

LDM 指令将基于 Rn 上的存储器地址的字值加载到 reglist 的寄存器中。

STM 指令将 reglist 中的寄存器的字值存放到基于 Rn 的存储器地址中。

用于访问的存储器地址为 4 字节间隔，其范围为 Rn 所指定的寄存器的值至 Rn + 4 * (n-1) 所指定的寄存器的值，这里的 n 是 reglist 中的寄存器数量。访问的顺序是按照寄存器的编号从低到高发生，最低编号的寄存器使用最低的存储器地址，最高编号的寄存器使用最高的存储器地址。如果写回后缀被指定，则 Rn + 4 *n 所指定的寄存器的值会被写回到 Rn 所指定的寄存器中。

（3）限制

在这些指令中：

[1] reglist 和 Rn 限制为 R0-R7；

[2] 必须要使用写回后缀，除非指令是 LDM 指令，在 LDM 里，reglist 也含有 Rn，在这种情况下，要紧记不能使用写回后缀；

[3] Rn 所指定的寄存器的值必须是字对齐的。详情请参考“地址对齐”小节；

[4] 对于 STM，如果 reglist 中存在着 Rn，那么 Rn 必须是列表中的第一个寄存器。

（4）条件标志

这些指令不会更改标志。

（5）示例

```
LDM R0,{R0,R3,R4} ; LDMIA 相近于 LDM
STMIA R1!,{R2-R4,R6}
```

（6）错误示例

```
STM R5!,{R4,R5,R6}；存放于 R5 的值是不可预测的
LDM R2,{}；在列表中至少要存在着一个寄存器。
```

6. PUSH 和 POP

将寄存器压入满递减堆栈和将满递减堆栈中的内容移入寄存器。

（1）语法

PUSH reglist

POP reglist

其中：

Reglist 是非空的寄存器列表。用大括号括着。它包含着寄存器范围。若它包含着多于一个的寄存器或寄存器范围，必须要将其用逗号隔开。

（2）操作

PUSH 将寄存器存放到堆栈中，最低编号的寄存器使用低存储器地址，最高编号的寄存器使用高存储器地址。

POP 将堆栈中的内容加载到寄存器中，最低编号的寄存器使用最低存储器地址，最高编号的寄存器使用最高存储器地址。

PUSH 将 SP 寄存器的值减去 4 所得的值用作最高存储器地址，POP 将 SP 寄存器的值用作最低的存储器地址来执行满递减堆栈操作。当操作完成时，PUSH 会更新 SP 寄存器来指向最低存储值的单元，而 POP 则会更新 SP 寄存器来指向高于所加载的最高单元的单元。

如果 POP 在它的 reglist 中包含了 PC，则当 POP 指令完成时，会在该单元上执行一个跳转操作。为 PC 所读出的 Bit[0 值用来更新 APSR T-位。该位必须为 1，以确保能正确执行程序。

（3）限制

在这些指令中：

[1] reglist 必须只为 R0-R7；

[2] 对于 PUSH 和 POP，异常情况分别是 LR 和 PC。

（4）条件标志

这些指令不会更改标志。

（5）示例

```
PUSH {R0,R4-R7}                    ; 将 R0,R4,R5,R6,R7 压入堆栈
PUSH {R2,LR}                       ;将 R2 和链接寄存器压入堆栈
POP {R0,R6,PC}                     ;令 r0,r6 和 PC 出栈，然后跳转到新的 PC 值.
```

通用数据处理指令表 22.17 显示了数据处理指令：

表 22.17 数据处理指令

助记符	简述
ADCS	进位加法
ADD{S}	加法
ANDS	逻辑与
ASRS	算术右移
BICS	位清零
CMN	比较负值
CMP	比较
EORS	异或
LSLS	逻辑左移
LSRS	逻辑右移
MOV{S}	传输
MULS	乘法
MVNS	取反传输
ORRS	逻辑或
REV	反转字里面的字节顺序
REV16	反转每半字里面的字节顺序
REVSH	反转低半字中的字节顺序，并进行符号扩展
RORS	循环右移
RSBS	反向减法
SBCS	带进位的减法
SUBS	减法
SXTB	符号扩展字节
SXTH	符号扩展半字
UXTB	零扩展字节
UXTH	零扩展半字
TST	测试

7. ADC、ADD、RSB 和 SUB

进位加法、加法、反向减法、进位减法、减法。

（1）语法

```
ADCS   {Rd,} Rn, Rm
ADD{S} {Rd,} Rn, <Rm|#imm>
RSBS {Rd,} Rn, Rm, #0
SBCS   {Rd,} Rn, Rm
SUB{S} {Rd,} Rn,
```

<Rm|#imm>

其中：

S 会令 ADD 或 SUB 指令更新标志

Rd 指定结果寄存器

Rn 指定首个源寄存器

Rm 指定第二个源寄存器

Imm 指定一个常量立即数值

当省略了可选的 Rd 寄存器限定符时，会假定其值与 Rn 相同，例如，ADDS R1,R2 与 ADDS R1、R1、R2 相同。

（2）操作

ADCS 指令将 Rn 中的值加到 Rm 的值中，如果进位标志被置位，则加多一个 1，并将结果存放在 Rd 所指定寄存器里，同时更新 N、Z、C 和 V 标志。

ADD 指令将 Rn 的值加到 Rm 的值或 imm 指定的立即数中，并将结果存放到 Rd 所指定的寄存器中。

ADDS 指令执行的操作与 ADD 相同，并还可以更新 N、Z、C 和 V 标志。

RSBS 指令是用 0 减去 Rn 中的值，得到一个负数，然后将结果值存放在 Rd 所指定的寄存器中，并更新 N、Z、C 和 V 标志。

SBCS 指令是用 Rn 的值减去 Rm 的值，如果进位标志置位，则减去一个 1。指令会将结果值存放到 Rd 所指定的寄存器中，并更新 N、Z、C 和 V 标志。

SUB 指令会减去 Rm 的值或 imm 所指定的立即数。指令把结果值存放到 Rd 所指定的寄存器中。

SUBS 指令执行的操作与 SUB 相同，同时它还可以更新 N、Z、C 和 V 标志。

如何使用 ADC 和 SBC 来综合处理多字算术，请参考下面的“示例”小节。

还可以参考“ADR”小节。

（3）限制

表 22.18 列出了寄存器指示符的合法组合和每一个指令可以使用的立即数。

表 22.18　ADC、ADD、RSB、SBC 和 SUB 操作数限制

指令	Rd	Rn	Rm	imm	限制
ADCS	R0-R7	R0-R7	R0-R7	-	Rd 和 Rn 必须指定相同的寄存器
ADD	R0-R15	R0-R15	R0-PC	-	Rd 和 Rn 必须指定相同的寄存器 Rd 和 Rm 必须不能同时指定 PC
	R0-R7	SP 或 PC	-	0-1020	立即数必须为 4 的整数倍
	SP	SP	-	0-508	立即数必须为 4 的整数倍
ADDS	R0-R7	R0-R7	-	0-7	-
	R0-R7	R0-R7	-	0-255	Rd 和 Rn 必须指定相同的寄存器
	R0-R7	R0-R7	R0-R7	-	-
RSBS	R0-R7	R0-R7	-	-	-
SBCS	R0-R7	R0-R7	R0-R7	-	Rd 和 Rn 必须指定相同的寄存器
SUB	SP	SP	-	0-508	立即数必须为 4 的整数倍
SUBS	R0-R7	R0-R7	-	0-7	-

续上表

指令	Rd	Rn	Rm	imm	限制
	R0-R7	R0-R7	-	0-255	Rd 和 Rn 必须指定相同的寄存器
	R0-R7	R0-R7	R0-R7	-	-

（4）示例

下例所示为二个指令将 R0 和 R1 所包含的 64 位整数值加到 R2 和 R3 所包含的另一个 64 位整数值中，并将结果存放到 R0 和 R1 中。

```
64 位加法：
    ADDS R0, R0, R2 ； 加上最低位的字
    ADCS R1, R1, R3 ;加上最高位的字，带进位
```

多字的值无需要使用连续的寄存器。下面示例为指令会令 R4、R5 和 R6 所包含的 96 位整数值减去 R1、R2 和 R3 所包含的 96 位整数值。该例将结果值存放在 R4、R5 和 R6 中。

```
96 位减法：
    SUBS R4, R4, R1 ； 减去最低位字
    SBCS R5, R5, R2 ； 减去中间的字，带进位
    SBCS R6, R6, R3 ； 减去最高位字，带进位
```

下列所示的 RSBS 指令是用来执行单个寄存器 1 的补码的操作。

```
算术负值运算:
    RSBS   R7, R7, #0 ； 用 0 减去 R7
```

8. AND、ORR、EOR 和 BIC

逻辑 AND、OR、异或和位清除。

（1）语法

ANDS {Rd,} Rn, Rm

ORRS {Rd,} Rn, Rm

EORS {Rd,} Rn, Rm

BICS {Rd,} Rn, Rm

其中：

Rd 是目标寄存器。

Rn 是保存第一个操作数的寄存器，且还是与目标寄存器相同的寄存器。

Rm 是第二个寄存器。

（2）操作

AND、EOR 和 ORR 对 Rn 和 Rm 的值按位执行 AND、异或、或操作。

BIC 指令对 Rn 上的位执行 AND 操作，对 Rm 值上的相应位执行逻辑非操作。

条件代码标志会根据操作的结果被更新，见“条件标志”小节。

（3）限制

在这些指令中，Rd、Rn 和 Rm 必须指定 R0-R7。

（4）条件标志

这些指令会：

[1] 根据结果值来更新 N 和 Z 标志；

[2] 不会影响 C 或 V 标志。

（5）示例

```
ANDS    R2, R2, R1
ORRS    R2, R2, R5
ANDS    R5, R5, R8
EORS    R7, R7, R6
BICS    R0, R0, R1
```

9. ASR, LSL, LSR 和 ROR

算术右移，逻辑左移，逻辑右移，循环右移。

（1）语法

ASRS {Rd,} Rm, Rs

ASRS {Rd,} Rm, #imm

LSLS {Rd,} Rm, Rs

LSLS {Rd,} Rm, #imm

LSRS {Rd,} Rm, Rs

LSRS {Rd,} Rm, #imm

RORS {Rd,} Rm, Rs

其中：

Rd 是目的寄存器。如果 Rd 被省略，则假定它的值与 Rm 相同。

Rm 是保存要移位的值的寄存器。

Rs 是保存着移位长度（该长度要应用到 Rm 中的值）的寄存器

Imm 是移位长度。

移位长度要由指令来决定：

ASR — 移位长度为 1 到 32

LSL — 移位长度为 0 到 31

LSR — 移位长度为 1 到 32.

注：MOVS Rd, Rm 是 LSLS Rd, Rm, #0 的假名。

（2）操作

ASR、LSL、LSR 和 ROR 对立即数 imm 所指定的长度而锁定的 Rm 寄存器的位或者 Rs 所指定的寄存器的最低位字节值执行算术左移、逻辑左移、逻辑右移或循环右移。

关于不同的指令会产生什么样的结果，详情请参考“移位操作”小节。

（3）限制

在这些指令中，Rd、Rm 和 Rs 必须只可以指定 R0-R7。对于非立即数指令，Rd 和 Rm 必须指定相同的寄存器。

（4）条件标志

这些指令根据结果值来更新 N 和 Z 标志。

C 标志被更新为最后移出的位。当移位长度为 0 时例外，见“移位操作”小节。V 标志不变。

（5）示例

```
ASRS    R7, R5, #9 ; 算术右移 9 位
LSLS    R1, R2, #3 ; 逻辑左移 3 位，并更新标志
LSRS    R4, R5, #6 ; 逻辑右移 6 位
RORS    R4, R4, R6 ; 循环右移 R6 低字节中的值
```

10. CMP 和 CMN

比较和比较负值。

（1）语法

CMN Rn, Rm

CMP Rn, #imm

CMP Rn, Rm

其中：

Rn 是保存第一个操作数的寄存器。

Rm 是用于比较的寄存器。

Imm 是用于比较的立即数值。

（2）操作

这些指令将一个寄存器中的值与另一个寄存器中的值或立即数进行比较。指令会根据结果值来更新条件标志，但不会将结果写入寄存器。

CMP 指令将 Rn 的值减去 Rm 所指定的寄存器值或立即数 imm，并更新标志。这操作与 SUBS 指令相同，不同的是结果值会被丢弃。

CMN 指令将 Rm 的值加到 Rn 的值中，并更新标志。这操作与 ADDS 指令相同，不同的是结果值会被丢弃。

（3）限制

对于：

[1] CMN 指令

指令 Rn、Rm 必须只能指定 R0-R7；

[2] CMP 指令：

- Rn 和 Rm 可以指定 R0-R14；
- 立即数的范围为 0-255。

（4）条件标志

这些指令根据结果值来更新 N、Z、C 和 V 标志。

（5）示例

```
CMP     R2, R9
CMN     R0, R2
```

11. MOV 和 MVN

传输和取反传输。

（1）语法

MOV{S} Rd, Rm

MOVS Rd, #imm

MVNS Rd, Rm

其中：

S 是可选后缀。如果指定了 S，则会根据操作的结果值来更新条件代码标志，见“条件执行”小节。

Rd 是目的寄存器。

Rm 是寄存器。

Imm 可以是 0-255 范围内的任何一个值。

（2）操作

MOV 指令将 Rm 的值复制到 Rd 中。

MOVS 指令执行的操作与 MOV 指令相同，但是它会更新 N 和 Z 标志。

MVNS 指令采用 Rm 的值，对该值执行按位的逻辑取反操作，并将结果存放到 Rd 中。

（3）限制

在这些指令中，Rd 和 Rm 必须指定 R0-R7。

当在 MOV 指令里 Rd 是 PC 时：

[1] 结果值的位[0]被丢弃；

[2] 在通过将结果值的位[0]强制为 0 来所创造的地址上执行跳转操作。T-位保持不变。

备注：尽管可以将 MOV 用作跳转指令，但是为了软件的可移植性，AMR 强烈建推荐使用 BX 或 BLX 指令来执行跳转操作。

（4）条件标志

如果 S 被指定，则这些指令会：

[1] 根据结果值更新 N 和 Z 标志；

[2] 不会影响 C 或 V 标志。

（5）示例

```
MOVS R0, #0x000B                ; 将 0x000B 写入 R0，更新标志
MOVS R1, #0x0                   ; 将 0 写入 R1，更新标志
MOV  R10, R12                   ; 将 R12 的值写入 R10，不更新标志
MOVS R3, #23                    ; 将 23 写入 R3
MOV  R8, SP                     ; 将堆栈指针的值写入 R8
MVNS R2, R0                     ; 将 R0 取反写入 R2 并更新标志
```

12. MULS

使用 32 位操作数的乘法，产生 32 位的结果值。

（1）语法

MULS Rd, Rn, Rm

其中：

Rd 是目的寄存器。

Rn、Rm 是保存进行乘法操作值的寄存器。

（2）操作

MUL 指令将 Rn 和 Rm 所指定的寄存器的值进行乘法操作，并将结果值的最低 32 位存放在 Rd 中。条件代码标志会按照操作的结果值而被更新，见“条件执行”小节。

该指令的结果并不是由操作数是有符号还是无符号来决定。

（3）限制

在该指令中：

[1] Rd、Rn 和 Rm 必须只能指定 R0-R7；

[2] Rd 必须要和 Rm 相同。

（4）条件标志

该指令会：

[1] 根据结果值来更新 N 和 Z 标志；

[2] 不会影响 C 或 V 标志。

（5）示例

```
MULS R0, R2, R0 ;乘法操作，标志被更新, R0 = R0 x R2
```

13. REV、REV16 和 REVSH

反转字节。

（1）语法

REV Rd, Rn

REV16 Rd, Rn

REVSH Rd, Rn

其中：

Rd 是目的寄存器。

Rn 是源寄存器。

（2）操作

使用这些指令来改变数据的端点排序：

REV — 将 32 位大端数据转换成小端的数据或将 32 位小端的数据转换成大端数据。

REV16 — 将二个打包的 16 位大端数据转换成小端的数据或将二个打包的小端的数据转换成大端数据。

REVSH —将 16 位有符号的大端数据转换成 32 位有符号小端数据或将 16 位有符号小端数据转换 32 位有符号大端数据。

（3）限制

在这些指令中，Rd 和 Rn 必须只可以指定 R0-R7。

（4）条件标志

这些指令不会更改标志。

（5）示例

```
REV R3, R7                ; 反转 R7 值的字节顺序，并将其写入 R3
REV16 R0, R0              ; 反转 R0 中的每一个 16 位半字的字节顺序
REVSH R0, R5              ; 反转有符号的半字
```

14. SXT and UXT

符号扩展和 0 扩展。

（1）语法

SXTB Rd, Rm

SXTH Rd, Rm

UXTB Rd, Rm

UXTH Rd, Rm

其中：

Rd 是目的寄存器。

Rm 是寄存器，其保存的值会被扩展。

（2）操作

这些指令从结果值中提取位：

[1] SXTB 提取位[7:0]并将值进行符号扩展到 32 位；
[2] UXTB 提取位[7:0]并将值用 0 扩展到 32 位；
[3] SXTH 提取[15:0]并将值进行符号扩展到 32 位；
[4] UXTH 提取[15:0]并将值用 0 扩展到 32 位。

（3）限制

在这些指令中，Rd 和 Rm 必须只可以指定 R0-R7。

（4）条件标志

这些指令不会影响标志。

（5）示例

```
SXTH R4, R6          ;获取 R6 的低半字，然后将其进行符号扩展到 32 位，并将结果写入 R4
UXTB R3, R1          ;获取 R10 最低位字节，并用 0 扩展，最后将结果写入 R3
```

15. TST

测试位。

（1）语法

TST Rn, Rm

其中：

Rn 是保存第一个操作数的寄存器。

Rm 是测试的寄存器。

（2）操作

该指令将一个寄存器的值与别一个寄存器中的值进行测试。它会根据结果值来更新条件标志，但是不会将结果值写入寄存器。

TST 指令对 Rn 中的值和 Rm 中的值执行位与操作。这是与 ANDS 指令相同的操作，不同的是它会丢弃结果值。

为了测试 Rn 中的某个位是 0 还是 1，要使用 TST 指令，且寄存器的该位要设为 1，其它所有位被清除为 0。

（3）限制

在这些指令中，Rn 和 Rm 必须只能指定 R0-R7。

（4）条件标志

这些指令：

[1] 会根据结果来更新 N 和 Z 标志；

[2] 不会影响 C 或 V 标志。

（5）示例

```
TST R0, R1          ；对 R0 值和 R1 值执行位与操作，更新条件代码标志，但结果值会被丢弃。
```

22.4.5 跳转和控制指令

表 22.19 所示为跳转和控制指令。

表 22.19 跳转和控制指令

助记符	简述
B{cc}	跳转（有条件）
BL	带链接的跳转
BLX	带链接的间接跳转
BX	间接跳转

1. B、BL、BX 和 BLX

跳转指令。

（1）语法

B{cond} label

BL label

BX Rm

BLX Rm

其中：

Cond 是可选的条件代码，见“条件执行”小节。

Label 是 PC 相对表达式，见“相对 PC 的表达式”小节。

Rm 是提供跳转地址的寄存器。

（2）操作

所有这些指令都会对 label 所指示的地址或在 Rm 所指定的寄存器中包含地址上执行跳转操作。另外：

[1] BL 和 BLX 指令将下一个指令的地址写入 LR，链接寄存器 R14；

[2] 如果 Rm 的位[0]是 0，则 BX 和 BLX 指令会导致 HardFault 异常。

BL 和 BLX 指令还会将 LR 的位[0]设置为 1。这就确保了该值适合由后续 POP {PC}或 BX 指令使用其来执行成功的返回跳转操作。

表 22.20 所示为适用于各种跳转指令的跳转范围。

表 22.20 跳转范围

指令	跳转范围
B label	−2 KB 至+2 KB
Bcond label	−256 字节至+254 字节
BL label	−16 MB 至+16 MB
BX Rm	寄存器中的任意值

续上表

指令	跳转范围
BLX Rm	寄存器中的任意值

（3）限制

在这些指令里：

[1] 不要在 BX 或 BLX 指令里使用 SP 或 PC；

[2] 对于 BX 和 BLX，为实现正确的执行操作，Rm 的位[0]必须为 1。位[0]用于更新 EPSR T-位，并会被从目标地址上丢弃。

备注：Bcond 是在 Cortex-M0 处理器上唯一的条件指令。

（4）条件标志

这些指令不会更改标志。

（5）示例

```
B     loopA                ;跳转到 loopA
BL    funC                 ;对函数 funC 进行带链接的跳转（调用）,返回存放在 LR 的地址
BX    LR                   ; 从函数调用中返回
BLX   R0                   ; 带链接的跳转，并从（调用）中更改为存放在 R0 所存放的地址
BEQ   labelD  ; 条件跳转到 abelD，如果最后的标志被设置，指令设置 Z 标志，否则不执行跳转
```

22.4.6 综合指令

表 22.21 所示为余下的 Cortex-M0 指令：

表 22.21 综合指令

助记符	简述
BKPT	断点
CPSID	更改处理器状态，禁止中断
CPSIE	更改处理器状态，使能中断
DMB	数据内存屏障
DSB	数据同步屏障
ISB	指令同步屏障
MRS	从特别寄存器传输到寄存器
MSR	从寄存器传输到特别寄存器
NOP	无操作
SEV	发送事件
SVC	超级用户调用
WFE	等待事件
WFI	等待中断

1. BKPT

断点。

（1）语法

BKPT #imm

其中：

imm 是 0-255 范围内的整数。

（2）操作

BKPT 指令会令处理器进入调试状态。当指令到达特定的地址时，调试工具可以使用该指令来调查系统状态。处理器会忽略 imm。如有需要，调试器可以使用它来存放断点的其它信息。

如果在执行 BKPT 指令时调试器没有连接上，那么处理器还有可能会产生 HardFault 或进入锁定状态。详情见本章“锁定”小节。

（3）限制

该指令没有限制。

（4）条件标志

该指令不会更改标志。

（5）示例

```
BKPT #0    ；立即数值设为 0x0 的断点。
```

2. CPS

更改处理器状态。

（1）语法

CPSID i

CPSIE i

（2）操作

CPS 更改 PRIMASK 特别寄存器值。通过设置 PRIMASK，CPSID 可令中断被关闭。而通过清除 PRIMASK，CPSIE 则可使能中断。关于这些寄存器的详细描述，请参考本章“异常屏蔽寄存器”小节。

（3）限制

该指令没有限制。

（4）条件标志

该指令不会更改条件标志。

（5）示例

```
CPSID i                    ;关闭所有的中断，NMI 除外(设置 PRIMASK)
CPSIE i                    ；使能中断  (清除 PRIMASK)
```

3. DMB

数据内存屏障。

（1）语法

DMB

（2）操作

DMB 用作数据内存屏障。它可确保会先检测到程序中位于 DMB 指令前的所有显式内存访问指令，然后再检测到程序中位于 DMB 指令后的显式内存访问指令。它不影响其他指令（不访问内存的指令）在处理器上的执行顺序。

（3）限制

该指令没有限制。

（4）条件标志

该指令不会更改标志。

（5）示例

```
DMB                                                    ; 数据内存屏障
```

4. DSB

数据同步屏障。

（1）语法

DSB

（2）操作

DSB 用作特别数据同步内存屏障，只有当此指令执行完毕后，才会执行程序中位于此指令后的指令。位于此指令前的所有显式内存访问均完成时，DSB 指令才会完成。

（3）限制

该指令没有限制。

（4）条件标志

该指令不会更改标志。

（5）示例

```
DSB                                                    ; 数据同步屏障
```

5. ISB

指令同步屏障。

（1）语法

ISB

（2）操作

ISB 用作指令同步屏障。它会刷新处理器的管道，因此在完成了 ISB 指令后，需要再次将 ISB 之后的所有指令从高速缓存或内存中提取出来。

（3）限制

该指令没有限制。

（4）条件标志

该指令不会更改标志。

（5）示例

```
ISB                                                    ; 指令同步屏障
```

6. MRS

将特别寄存器的内容移动到通用寄存器中。

（1）语法

MRS Rd, spec_reg

其中：

Rd 是通用目的寄存器。

spec_reg 是其中一个特别寄存器：APSR、IPSR、EPSR、IEPSR、IAPSR、EAPSR、PSR、MSP、PSP、PRIMASK 或 CONTROL。

（2）操作

MRS 将特别寄存器的内容存放到通用寄存器中。MRS 指令可以结合 MR 指令来产生读-修改-写序列，这适用于在 PSR 中修改特别标志。

见“MSR”小节。

（3）限制

在该指令中，Rd 必须不能是 SP 或 PC。

（4）条件标志

该指令不会更改标志。

（5）示例

```
MRS R0, PRIMASK                    ; 读取 PRIMASK 值并将其写入 R0
```

7. MSR

将通用寄存器的内容传移到指定的特别寄存器中。

（1）语法

MSR spec_reg, Rn

其中：

Rn 是通用源寄存器。

spec_reg 是特别目的寄存器：APSR,、IPSR、EPSR、IEPSR、IAPSR、EAPSR、PSR、MSP、PSP、PRIMASK 或 CONTROL。

（2）操作

MSR 使用 Rn 所指定的寄存器的值来更新其中一个特别寄存器。

见“MRS”小节。

（3）限制

在该指令里，Rn 必须不能为 SP 和 PC。

（4）条件标志

该指令明确地根据 Rn 中的值来更新标志。

（5）示例

```
MSR CONTROL, R1                    ;读取 R1 的值，并将其写入 CONTROL 寄存器。
```

8. NOP

空操作

（1）语法

NOP

（2）操作

NOP 执行的是无操作，且不能保证会占用指令时间。处理器可在它到达执行阶段之前将其从管道中移除。

使用 NOP 指令来进行填充，例如，在 64 位边界上放置后续指令。

（3）限制

该指令没有限制。

（4）条件标志

该指令不会更改标志。

（5）示例

```
NOP                                                    ；无操作
```

9. SEV

发送事件。

（1）语法

SEV

（2）操作

SEV 将事件信号发送到一个多处理器系统内的所有处理器中。它还可设置局部事件寄存器。见本章“电源管理”小节。

也可参考“WFE”小节。

（3）限制

该指令没有限制。

（4）条件标志

该指令不会更改标志。

（5）示例

```
SEV                                                    ；发送事件
```

10. SVC

超级用户调用。

（1）语法

SVC #imm

其中：

Imm 是 0-255 范围内的整数。

（2）操作

SVC 指令会引发 SVC 异常。

处理器会忽略 imm。如果有需要，可以通过异常处理程序获取 imm 来决定要请求什么样的服务程序。

（3）限制

该指令没有限制。

（4）条件标志

该指令不会更改标志。

（5）示例

```
SVC #0x32 ；超级用户调用(SVC 处理程度使用堆栈的 PC 来锁定立即数的位置，然后将其提取出来)
```

11. WFE

等待事件。

备注：WFE 指令不会在 LPCIdesit 上执行。

（1）语法

WFE

（2）操作

如果事件寄存器为 0，则 WFE 挂起执行，直至发生下列其中的一个事件：

[1] 出现异常，除非异常屏蔽寄存器或当前优先级级别将其屏蔽；
[2] 异常进入挂起状态，如果系统控制寄存器的 SEVONPEND 置位；
[3] 存在调试进入请求，如果调试使能的话；
[4] 外设或多处理器系统里另一个处理器通过使用 SEV 指令来发出事件信号。

如果事件寄存器为 1，则 WFE 将其清除为 0 并立即完成操作。

详情请参考本章“电源管理”小节。

备注：WFE 的目的只是用于节约功率。当写软件时，假定 WFE 作为 NOP 运行。

（3）限制

该指令没有限制。

（4）条件标志

该指令不会更改标志。

（5）示例

```
WFE                                    ；等待事件
```

12. WFI

等待中断。

（1）语法

WFI

（2）操作

WFI

挂起执行，直至发生下列其中的一个事件：

[1] 异常；
[2] 中断变为挂起状态，如果 PRIMASK 被清除，则该中断占用优先权；
[3] 存在调试进入请求，无论调试是否使能。

备注：WFI 的目的只是用于节约功率。当写软件时，假定 WFI 作为 NOP 运行。

（3）限制

该指令没有限制。

（4）条件标志

该指令不会更改标志。

（5）示例

```
WFI                                    ；等待中断。
```

22.5 外设

22.5.1 关于 ARM Cortex-M0

专用外设总线（PPB）的地址映射为：

表 22.22 内核外设寄存器区

地址	内核外设	描述
0xE000E008-0xE000E00F	系统控制块	表 22.31
0xE000E010-0xE000E01F	系统定时器	表 22.40
0xE000E100-0xE000E4EF	嵌套向量中断控制器	表 22.23
0xE000ED00-0xE000ED3F	系统控制块	表 22.31
0xE000EF00-0xE000EF03	嵌套向量中断控制器	表 22.23

在寄存器描述中，寄存器的**类型**有以下几种：

R/W — 可读和可写

RO — 只读

WO — 只写

22.5.2 嵌套向量中断控制器

本节描述**嵌套向量中断控制器**（NVIC）以及它使用的寄存器。NVIC 支持：

[1] 32 个中断；

[2] 每个中断的优先级可编程为 0~3 四种级别。级别越高对应的优先级越低。因此，级别 0 是最高的中断优先级。

[3] 中断信号的电平和脉冲检测；

[4] 中断末尾连锁；

[5] 一个**外部不可屏蔽中断**（NMI）。LPC111x 没有 NMI。

处理器在异常进入时自动使它的状态入栈，在异常退出时自动使它的状态出栈，无需采用任何指令。这就实现了低延迟的异常处理。NVIC 的硬件寄存器有：

表 22.23 NVIC 寄存器小结

地址	名称	类型	复位值	描述
0xE000E100	ISER	R/W	0x00000000	本章“中断设置-使能寄存器”小节
0xE000E180	ICER	R/W	0x00000000	本章“中断清除-使能寄存器”小节
0xE000E200	ISPR	R/W	0x00000000	本章“中断设置-挂起寄存器”小节
0xE000E280	ICPR	R/W	0x00000000	本章“中断清除-挂起寄存器”小节
0xE000E400-0xE000E41C	IPR0-7	R/W	0x00000000	本章“中断优先级寄存器”小节

1. 使用 CMSIS 访问 Cortex-M0 NVIC 寄存器

CMSIS 函数允许在不同的 Cortex-M 系列中进行软件移植。

当利用 CMSIS 来访问 NVIC 寄存器时要用到以下函数：

表 22.24 CMSIS 访问 NVIC 函数

CMSIS 函数	描述
void NVIC_EnableIRQ(IRQn_Type IRQn)[1]	使能一个中断或异常
void NVIC_DisableIRQ(IRQn_Type IRQn) [1]	禁能一个中断或异常
void NVIC_SetPendingIRQ(IRQn_Type IRQn) [1]	将中断或异常的挂起状态设为 1
void NVIC_ClearPendingIRQ(IRQn_Type IRQn) [1]	将中断或异常的挂起状态清零
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn) [1]	读取中断或异常的挂起状态。如果挂起状态被设为 1，这个函数就返回非零值
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority) [1]	将一个优先级可配置的中断或异常的优先级设置为级别 1
uint32_t NVIC_GetPriority(IRQn_Type IRQn) [1]	读取一个优先级可配置的中断或异常的优先级。这个函数返回当前的优先级级别。

[1] 输入参数 IRQn 是 IRQ 编号，更多信息请见表 22.10。

2. 中断设置-使能寄存器

ISER 使能中断，并显示哪些中断被使能。有关寄存器属性请见表 22.23 的寄存器小结。

该寄存器的位分配如下：

表 22.25 ISER 的位分配

位	名称	功能
[31:0]	SETENA	中断设置-使能位。 写：0 = 无影响 1 = 使能中断 读：0 = 中断被禁能 1 = 中断被使能

如果一个挂起中断被使能，NVIC 就根据它的优先级来激活该中断。如果一个中断未被使能，使该中断的中断信号有效可将中断的状态变成挂起，但是，不管这个中断的优先级如何，NVIC 都不会激活该中断。

3. 中断清除-使能寄存器

ICER 禁能中断，并显示哪些中断被使能。有关寄存器属性请见表 22.23 的寄存器小结。

这个寄存器的位分配如下：

表 22.26 ICER 的位分配

位	名称	功能
[31:0]	CLRENA	中断清除-使能位。 写：0 = 无影响 1 = 禁能中断 读：0 = 中断被禁能 1 = 中断被使能

4. 中断设置-挂起寄存器

ISPR 强制中断进入挂起状态，并显示哪些中断正在挂起。有关寄存器属性请见表 22.23 的寄存器小结。

这个寄存器的位分配如下：

表 22.27 ISPR 的位分配

位	名称	功能
[31:0]	SETPEND	中断设置-挂起位。 写：0 = 无影响 1 = 将中断状态变为挂起 读：0 = 中断没有挂起 1 = 中断正在挂起

备注：向 ISPR 位写 1 相当于下面两种情况:

- 正在挂起的中断不会有任何影响
- 被禁能的中断会将中断的状态设置成挂起

5. 中断清除-挂起寄存器

ICPR 使中断离开挂起状态，并显示哪些中断正在挂起。寄存器属性请见表 22.23 的寄存器小结。

这个寄存器的位分配如下：

表 22.28 ICPR 的位分配

位	名称	功能
[31:0]	CLRPEND	中断清除-挂起位。 写：0 = 无影响　　1 = 清除中断的挂起状态 读：0 = 中断没有挂起　　1 = 中断正在挂起

备注：向 ICPR 位写 1 不影响相应中断的有效状态。

6. 中断优先级寄存器

IPR0-IPR7 寄存器为每个中断提供了一个两位的优先级域。这些寄存器只能字访问。关于它们的属性请见表 22.23 的寄存器小结。

每个寄存器包含 4 个优先级域，如下所示：

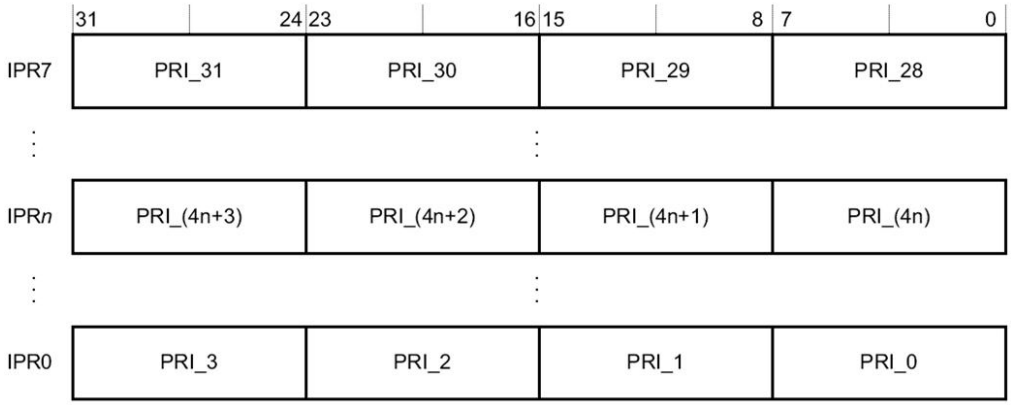

图 22.13 IPR 寄存器

表 22.29 IPR 的位分配

位	名称	功能
[31:24]	优先级，字节偏移量 3	每个优先级域保存一个优先级值（0~3）。值越小，对应中断的优先级越高。处理器只使用每个域的 bit[7:6]，bit[5:0]读出为 0，写操作被忽略。
[23:16]	优先级，字节偏移量 2	
[15:8]	优先级，字节偏移量 1	
[7:0]	优先级，字节偏移量 0	

有关中断优先级阵列（提供了中断优先级的软件视角）访问的更多信息请参考本章“使用 CMSIS 访问 Cortex-M0 NVIC 寄存器”小节。

使用下面的方法为中断 M 找出 IPR 编号和字节偏移量：

- 相应的 IPR 编号 N，通过等式 N = N/4 得出
- 这个寄存器中所需优先级域的字节偏移量是 M MOD 4（M 除以 4 取余），在这里：
  — 字节偏移量 0 指的是寄存器位[7:0]

— 字节偏移量 1 指的是寄存器位[15:8]

— 字节偏移量 2 指的是寄存器位[23:16]

— 字节偏移量 3 指的是寄存器位[31:24]

7. 电平有效的中断和脉冲中断

处理器支持电平有效的中断和脉冲中断。脉冲中断也被描述成边沿触发的中断。

一个电平有效的中断一直保持有效，直至外设将中断信号撤销。通常，发生这种情况的原因是 ISR 访问外设导致外设将中断请求清除。脉冲中断是在处理器时钟的上升沿同时采样到的一个中断信号。为了确保 NVIC 检测到中断，外设必须使中断信号至少在一个时钟周期内保持有效，在这段时间内 NVIC 检测脉冲并锁存中断。

当处理器进入 ISP 时，它自动消除中断的挂起状态，见“（1）中断的硬件和软件控制”小节。对于一个电平有效的中断，如果在处理器从 ISR 返回之前中断信号未被撤销，中断就再次变成挂起，处理器必须再次执行 ISR。这就表示，外设可以一直使中断信号保持有效，直到它不再需要服务为止。

（1）中断的硬件和软件控制

Cortex-M0 锁存所有的中断。外设中断会由于下面的其中一个原因而变为挂起：

- NVIC 检测到中断信号有效，而相应的中断无效；
- NVIC 检测到中断信号的一个上升沿；
- 软件向相应的中断设置-挂起寄存器位写入值，见本章“中断设置-挂起寄存器”小节。

挂起的中断一直保持挂起，直到出现以下其中一种情况：

- 处理器进入中断的 ISR。这就使中断的状态从挂起变为有效。而且：
  - — 对于电平有效的中断，当处理器从 ISR 返回时，NVIC 采样中断信号。如果中断信号有效，中断的状态变回挂起，这可能使得处理器立刻再次进入 ISR。否则，中断的状态变为无效。
  - — 对于脉冲中断，NVIC 继续监测中断信号，如果这个中断信号一直处于脉冲状态，中断的状态就变成挂起和有效。在这种情况下，当处理器从 ISR 返回时，中断的状态变为挂起，这可能使得处理器立刻重新进入 ISR。

    如果当处理器在处理 ISR 时中断信号的脉冲就不存在了，那么，当处理器从 ISR 返回时中断的状态变为无效。
- 利用软件向相应的中断清除-挂起寄存器位写入值。

  对于电平有效的中断，如果中断信号仍然有效，中断的状态不改变。否则，中断的状态变为无效。

  对于脉冲中断，中断的状态变为：

  — 无效（如果中断之前的状态是挂起）

  — 有效（如果中断之前的状态是有效和挂起）

8. NVIC 的使用提示和技巧

保证软件正确使用对齐的寄存器访问。处理器不支持非对齐的 NVIC 寄存器访问。

中断即使被禁能也可以进入挂起状态。禁能一个中断只阻止处理器处理中断。

（1）NVIC 编程提示

软件使用 CPSIE i 和指令来使能和禁能中断。CMSIS 为这些指令提供以下内在函数：

```
void __disable_irq(void)   // 禁能中断
void __enable_irq(void)    // 使能中断
```

另外，CMSIS 提供了许多 NVIC 控制函数，包括：

表 22.30 CMSIS 的 NVIC 控制函数

CMSIS 中断控制函数	描述
void NVIC_EnableIRQ(IRQn_t IRQn)	使能 IRQn
void NVIC_DisableIRQ(IRQn_t IRQn)	禁能 IRQn
uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)	如果 IRQn 正在挂起，返回真（1）
void NVIC_SetPendingIRQ (IRQn_t IRQn)	设置 IRQn 挂起状态
void NVIC_ClearPendingIRQ (IRQn_t IRQn)	清除 IRQn 挂起状态
void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)	设置 IRQn 的优先级
uint32_t NVIC_GetPriority (IRQn_t IRQn)	读取 IRQn 的优先级
void NVIC_SystemReset (void)	复位系统

输入参数 IRQn 是 IRQ 编号，更多信息请见表 22.10。有关这些函数的更多信息，请见 CMSIS 的资料。

22.5.3 系统控制块

系统控制块（SCB）提供了系统执行信息和系统控制，包括配置、控制和系统异常的报告。SCB 寄存器有：

表 22.31 SCB 寄存器小结

地址	名称	类型	复位值	描述
0xE000ED00	CPUID	RO	0x410CC200	本章“CPUID 寄存器”小节
0xE000ED04	ICSR	RW[1]	0x00000000	本章“中断控制和状态寄存器”小节
0xE000ED0C	AIRCR	RW[1]	0xFA050000	本章“应用中断和复位控制寄存器”小节
0xE000ED10	SCR	RW	0x00000000	本章“系统控制寄存器”小节
0xE000ED14	CCR	RO	0x00000204	本章“配置和控制寄存器”小节
0xE000ED1C	SHPR2	RW	0x00000000	本章“系统处理程序优先级寄存器 2”小节
0xE000ED20	SHPR3	RW	0x00000000	本章“系统处理程序优先级寄存器 3”小节

[1] 更多信息请见寄存器描述。

1. Cortex-M0 SCB 寄存器的 CMSIS 映射

为了提高软件效率，CMSIS 简化了 SCB 寄存器的表现形式。在 CMSIS 中，阵列 SHP[1] 对应寄存器 SHPR2-SHPR3。

2. CPUID 寄存器

CPUID 寄存器包含处理器的型号、版本和实现信息。有关它的属性请见寄存器小结。CPUID 的位分配如下：

表 22.32 CPUID 寄存器的位分配

位	名称	功能
[31:24]	Implementer	实现代码：0x41 = ARM

位	名称	功能
[23:20]	Variant	更新编号，产品版本标识符 rnpn 中 r 的值：0x0 = 版本 0
[19:16]	Constant	定义处理器结构的常量；读取的结果是：0xC = ARMv6-M 结构
[15:4]	Partno	处理器的型号：0xC20 = Cortex-M0
[3:0]	Revision	修订编号，产品版本标识符 rnpn 中 p 的值：0x0 = Patch 0

3. 中断控制和状态寄存器

ICSR：

[6] 提供了：

— 为不可屏蔽中断（NMI）异常提供了一个设置-挂起位；

— 为 PendSV 和 SysTick 异常提供了设置-挂起位和清除-挂起位。

[7] 指明了：

— 正在处理的异常的异常编号；

— 是否有被抢占的有效异常；

— 最高优先级挂起异常的异常编号；

— 是否有任何异常正在挂起。

有关 ICSR 的属性请见表 22.31 的寄存器小结。ICSR 的位分配如下：

表 22.33 ICSR 的位分配

位	名称	类型	功能
[31]	NMIPENDSET[2]	R/W	NMI 设置-挂起位。 写：0 = 无影响 1 = 将 NMI 异常的状态变为挂起 读：0 = NMI 异常未挂起 1 = NMI 异常正在挂起 由于 NMI 是优先级最高的异常，因此，一般情况下，处理器一旦检测到向该位写 1 就立刻进入 NMI 异常处理程序。处理器进入处理程序后将该位清零。这就表示，只有当 NMI 信号在处理器正在执行 NMI 异常处理程序的过程中再次有效，通过异常处理程序读取这个位才返回 1。
[30:29]	-	-	保留。
[28]	PENDSVSET	R/W	PendSV 设置-挂起位。 写：0 = 无影响 1 = 将 PendSV 异常的状态变为挂起 读：0 = PendSV 异常未挂起 1 = PendSV 异常正在挂起 向该位写 1 是将 PendSV 异常状态设为挂起的唯一方法。
[27]	PENDSVCLR	WO	PendSV 清除-挂起位。 写：0 = 无影响 1 = 撤销 PendSV 异常的挂起状态
[26]	PENDSTSET	R/W	SysTick 异常设置-挂起位。 写：0 = 无影响 1 = 将 SysTick 异常的状态变为挂起 读：0 = SysTick 异常未挂起 1 = SysTick 异常正在挂起

续上表

位	名称	类型	功能
[25]	PENDSTCLR	WO	SysTick 异常清除-挂起位。 写：0 = 无影响 1 = 撤销 SysTick 异常的挂起状态 该位只可写。当对这个寄存器执行读操作时，该位读出的值不可知。
[24:23]	-	-	保留。
[22]	ISRPENDING	RO	除 NMI 和故障之外的中断的挂起标志: 0 = 中断未挂起 1 = 中断正在挂起
[21:18]	-	-	保留
[17:12]	VECTPENDING	RO	指示优先级最高的、正在挂起的并且使能的异常的异常编号: 0 = 没有正在挂起的异常 非零 = 优先级最高的、正在挂起的并且使能的异常的异常编号。
[11:6]	-	-	保留
[5:0]	VECTACTIVE[1]	RO	包含有效的异常编号: 0 = 线程模式 非零 = 当前有效异常的异常编号[1]。 备注：这个值减去 16 得到 CMSIS IRQ 编号，编号标识出对应在中断清除-使能、设置-使能、清除-挂起、设置-挂起以及优先级寄存器中的位，请见表 22.5。

[1] 这个值与 IPSR 位[5:0]的值相同，请见表 22.5。

[2] NMI 不能在 LPC111x 上实现。

写 ICSR 时，如果执行下列操作，结果将不可知：

[1] 写 1 到 PENDSVSET 位和写 1 到 PENDSVCLR 位

[2] 写 1 到 PENDSTSET 位和写 1 到 PENDSTCLR 位

4. 应用中断和复位控制寄存器

AIRCR 提供了数据访问的字节顺序状态和系统的复位控制信息。有关寄存器的属性请见表 22.31 和表 22.34 的寄存器小结。

如果要写这个寄存器，必须先向 VECTKEY 域写入 0x05FA，否则，处理器会将写操作忽略。

AIRCR 的位分配如下：

表 22.34 AIRCR 的位分配

位	名称	类型	功能
[31:16]	读：保留 写：VECTKEY	RW	寄存器码: 读出的值不可知。 执行写操作时将 0x05FA 写入 VECTKEY，否则写操作被忽略。
[15]	ENDIANESS	RO	采用的数据字节存储顺序: 0 = 小端 1 = 大端

续上表

位	名称	类型	功能
[14:3]	-	-	保留
[2]	SYSRESETREQ	WO	系统复位请求: 0 = 无影响 1 = 请求一个系统级复位 这个位读出为 0。
[1]	VECTCLRACTIVE	WO	保留供调试使用。这个位读出为 0。当写这个寄存器时，必须向这个位写 0，否则操作将不可预知。
[0]	-	-	保留

5. 系统控制寄存器

SCR 控制着低功耗状态的进入和退出特性。有关寄存器的属性请见表 22.31 的寄存器小结。SCR 的位分配如下：

表 22.35 SCR 的位分配

位	名称	功能
[31:5]	-	保留
[4]	SEVONPEND	挂起时发送事件位: 0 = 只有使能的中断或事件能够唤醒处理器。不接受禁能中断的唤醒。 1 = 使能的事件和包括禁能中断在内的所有中断都能唤醒处理器。 当一个事件或中断进入挂起状态时，事件信号将处理器从 WFE 唤醒。 如果处理器并未在等待一个事件，事件被记录，影响下个 WFE。 处理器也可以在执行 SEV 指令时唤醒。
[3]	-	保留
[2]	SLEEPDEEP	控制处理器是将睡眠模式还是深度睡眠模式作为低功耗模式: 0 = 睡眠 1 = 深度睡眠
[1]	SLEEPONEXIT	指示当从处理器模式返回到线程模式时 sleep-on-exit（退出时进入睡眠）: 0 = 处理器返回到线程模式时不进入睡眠 1 = 处理器从 ISR 返回到线程模式时进入睡眠或深度睡眠 将该位设为 1 允许一个中断驱动的应用程序避免返回到一个空的主应用程序。
[0]	-	保留

6. 配置和控制寄存器

CCR 是一个只读寄存器，指出了 Cortex-M0 处理器行为的一些情况。有关 CCR 属性请见表 22.31 的寄存器小结。

CCR 的位分配如下：

表 22.36 CCR 的位分配

位	名称	功能
[31:10]	-	保留
[9]	STKALIGN	该位读出总是为 0，指示进入异常时堆栈按 8 字节对齐。 进入异常时，处理器使用入栈的 PSR 的 bit[9]来指示栈对齐。从异常中返回时，处理器使用这个入栈的位来恢复正确的栈对齐。
[8:4]	-	保留
[3]	UNALIGN_TRP	该位读出总是为 0，指示所有未对齐的访问产生一个 HardFault。
[2:0]	-	保留

7. 系统处理程序优先级寄存器

SHPR2-SHPR3 寄存器设置优先级可配置的异常处理程序的优先级级别（0~3）。

SHPR2-SHPR3 是字可访问的。有关它们的属性请见表 22.31 的寄存器小结。

利用 CMSIS 访问系统异常的优先级级别要用到以下 CMSIS 函数：

[1] uint32_t NVIC_GetPriority(IRQn_Type IRQn)

[2] void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)

输入参数 IRQn 是 IRQ 编号，更多信息请见表 22.10。

系统故障处理程序、优先级域以及每个处理程序的寄存器如下所示：

表 22.37 系统故障处理程序优先级域

处理程序	域	寄存器描述
SVCall	PRI_11	见本章“系统处理程序优先级寄存器”小节。
PendSV	PRI_14	见本章“系统处理程序优先级寄存器”小节。
SysTick	PRI_15	

每个 PRI_N 域 8 位宽，但处理器只使用每个域的 bit[7:6]；bit[5:0]读出为 0，写操作被忽略。

（1）系统处理程序优先级寄存器 2

该寄存器的位分配如下：

表 22.38 SHPR2 寄存器的位分配

位	名称	功能
[31:24]	PRI_11	系统处理程序 11（SVCall）的优先级
[23:0]	-	保留

（2）系统处理程序优先级寄存器 3

该寄存器的位分配如下：

表 22.39 SYST_CSR 的位分配

位	名称	功能
[31:24]	PRI_15	系统处理程序 15（SysTick 异常）的优先级
[23:16]	PRI_14	系统处理程序 14（PendSV）的优先级
[15:0]	-	保留

8. SCB 使用的提示和技巧

保证软件使用对齐的 32 位字大小传输来访问所有的 SCB 寄存器。

22.5.4 系统定时器，SysTick

当系统定时器被使能时，定时器从重装值开始递减计数到零，下个时钟周期再将 SYST_RVR 的值重新加载到定时器（一个回合），然后在后面的时钟周期下继续开始递减计数。向 SYST_RVR 写零会使计数器在下个回合禁能。当计数器跳变到零时，COUNTFLAG 状态位被设为 1。读 SYST_CSR 将 COUNTFLAG 位清零。

写 SYST_CVR 会将该寄存器和 COUNTFLAG 状态位都清零。这个写操作不触发 SysTick 异常逻辑。读取 SYST_CVR 寄存器返回的是读取操作执行当下的寄存器值。

备注：当寄存器由于调试而被终止时，计数器不递减计数。

系统定时器寄存器有：

表 22.40 系统定时器寄存器小结

地址	名称	类型	复位值	描述
0xE000E010	SYST_CSR	R/W	0x00000000	本章“SysTick 控制和状态寄存器”小节
0xE000E014	SYST_RVR	R/W	不可知	本章“SysTick 重装值寄存器”小节
0xE000E018	SYST_CVR	R/W	不可知	本章“SysTick 当前值寄存器”小节
0xE000E01C	SYST_CALIB	RO	0xC0000000[1]	本章“SysTick 校准值寄存器”小节

[1] SysTick 的校准值

1. SysTick 控制和状态寄存器

SYST_CSR 使能 SysTick 特性。有关寄存器的属性请见表 22.40 的寄存器小结。该寄存器的位分配为：

表 22.41 SYST_CSR 的位分配

位	名称	功能
[31:17]	-	保留
[16]	COUNTFLAG	如果自从上次读这个寄存器之后定时器计数到 0，该位就返回 1
[15:3]	-	保留
[2]	CLKSOURCE	选择 SysTick 定时器的时钟源: 0 = 外部基准时钟 1 = 处理器时钟
[1]	TICKINT	使能 SysTick 异常请求: 0 = 计数到零不提交 SysTick 异常请求 1 = 计数到零提交 SysTick 异常请求
[0]	ENABLE	使能计数器: 0 = 计数器被禁能 1 = 计数器被使能

2. SysTick 重装值寄存器

SYST_RVR 设定了加载到 SYST_CVR 的起始值。有关寄存器的属性请见表 22.40 的寄存器小结。该寄存器的位分配为：

表 22.42 SYST_RVR 的位分配

位	名称	功能
[31:24]	-	保留
[23:0]	RELOAD	当计数器被使能且计数值到达 0 时加载到 SYST_CVR 的值，请见本章“计算 RELOAD 值”小节。

（1）计算 RELOAD 值

RELOAD 值可以是 0x00000001-0x00FFFFFF 范围内的任何值。您可以将 RELOAD 的值设为 0，这不会产生任何影响，因为计数值从 1 变为 0 时 SysTick 异常请求和 COUNTFLAG 都被激活了。

如果要产生一个周期为 N 个处理器时钟周期的多次触发定时器，就可以将 RELOAD 值设为 N-1。例如，如果要求每隔 100 个时钟脉冲就触发一次 SysTick 中断，RELOAD 就被设为 99。

3. SysTick 当前值寄存器

SYST_VCR 包含 SysTick 计数器的当前值。有关寄存器的属性请见表 22.40 的寄存器小结。该寄存器的位分配如下：

表 22.43 SYST_CVR 的位分配

位	名称	功能
[31:24]	-	保留
[23:0]	CURRENT	读取时返回 SysTick 计数器的当前值。 向这个域写入任何值都会将该域清零，还会清零 SYST_CSR 的 COUNTFLAG 位。

4. SysTick 校准值寄存器

SYST_CALIB 寄存器指明了 SysTick 的校准特性。有关寄存器的属性请见表 22.40 的寄存器小结。该寄存器的位分配如下：

表 22.44 SYST_CALIB 寄存器的位分配

位	名称	功能
[31]	NOREF	该位读出为 0。该位指明不提供独立的基准时钟
[30]	SKEW	该位读出为 0。由于 TENMS 不可知，因此，10ms 不精确计时的校准值不能确定。这会影响 SysTick 作为软件实时时钟的适用性
[29:24]	-	保留
[23:0]	TENMS	该位读出为 0。该域指明校准值不可知。

如果校准信息不可知，就通过处理器时钟或外部时钟的频率来计算所需的校准值。

5. SysTick 使用的提示和技巧

利用中断控制器时钟来更新 SysTick 计数器。如果这个时钟信号由于进入低功耗模式而终止，SysTick 计数器就停止计数。

确保软件使用字访问来访问 SysTick 寄存器。

如果在复位时没有定义 SysTick 计数器的重装值和当前值，正确的 SysTick 计数器初始化序列如下：

第 1 步：设置重装值

第 2 步：清除当前值

第 3 步：设置控制和状态寄存器