原创 上海_... 已于 2023-08-08 21:11:03 修改 58 收藏 1 版权

文章标签： arm开发

ARM Cortex-M3 权威指南-概览和基础

1 ARM Cortex-M3 处理器初探

1.1 Cortex-M3简评

1.1.1 高性能

- 许多指令都是单周期的——包括乘法相关指令。并且从整体性能上，Cortex-M3 比得过绝大多数其它的架构。
- 指令总线和数据总线被分开，取值和访内可以并行不悖
- Thumb-2 的到来告别了状态切换的旧世代，再也不需要花时间来切换于 32 位 ARM 状态和16 位 Thumb 状态之间了。这简化了软件开发和代码维护，使产品面市更快。
- Thumb-2 指令集为编程带来了更多的灵活性。许多数据操作现在能用更短的代码搞定，这意味着 Cortex-M3 的代码密度更高，也就对存储器的需求更少。
- 取指都按 32 位处理。同一周期最多可以取出两条指令，留下了更多的带宽给数据传输。
- Cortex-M3 的设计允许单片机高频运行（现代半导体制造技术能保证 100MHz 以上的速度）。即使在相同的速度下运行， CM3 的每指令周期数(CPI)也更低，于是同样的 MHz 下可以做更多的工作；另一方面，也使同一个应用在 CM3 上需要更低的主频。

1.1.2 先进的中断处理功能

- 内建的嵌套向量中断控制器支持多达 240 条外部中断输入。向量化的中断功能剧烈地缩短了中断延迟，因为不再需要软件去判断中断源。中断的嵌套也是在硬件水平上实现的，不需要软件代码来实现。
- Cortex-M3 在进入异常服务例程时，自动压栈了 R0-R3, R12, LR, PSR 和 PC，并且在返回时自动弹出它们，这多清爽！既加速了中断的响应，也再不需要汇编语言代码了。
- NVIC 支持对每一路中断设置不同的优先级，使得中断管理极富弹性。最粗线条的实现也至少要支持 8 级优先级，而且还能动态地被修改。
- 优化中断响应还有两招，它们分别是“咬尾中断机制”和“晚到中断机制”。
- 有些需要较多周期才能执行完的指令，是可以被中断 - 继续的——就好

POP。

除非系统被彻底地锁定， NMI（不可屏蔽中断）会在收到请求的第一时间予以响应。对于很多安全-关键(safety-critical)的应用， NMI 都是必不可少的。

1.1.3 调试支持

在支持传统的 JTAG 基础上，还支持更新更好的串行线调试接口。

基于 CoreSight 调试解决方案，使得处理器哪怕是在运行时，也能访问处理器状态和存储器内容。

内建了对多达 6 个断点和 4 个数据观察点的支持。

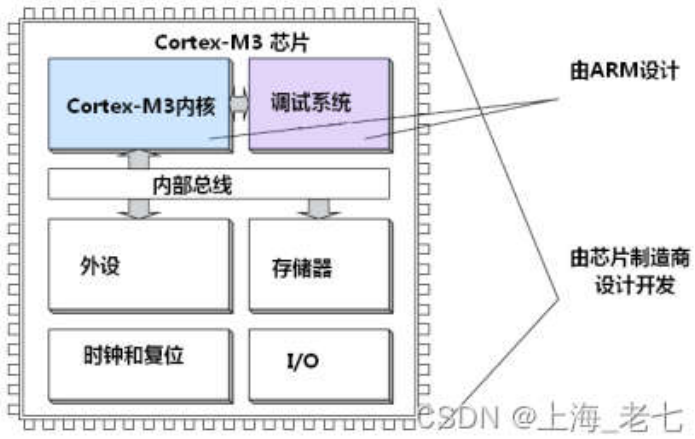
可以选配一个 ETM，用于指令跟踪。数据的跟踪可以使用 DWT

在调试方面还加入了以下的新特性，包括 fault 状态寄存器，新的 fault 异常，以及闪存修补 （patch）操作，使得调试大幅简化。

可选 ITM 模块，测试代码可以通过它输出调试信息，而且“拎包即可入住”般地方便使用。

1.2 基于cortex-M3的芯片设计

Cortex-M3处理器内核是芯片的中央处理单元，完整的MCU还需要很多其他组件，例如存储，外设，IO等。芯片设计商得到CM3核的授权后，就会把CM3用到自己的芯片中，做一些定制化的设计，所以不同的厂商有不同的配置，想了解具体型号的处理器需要查阅厂家提供的文档，比如stm32，nxp，ti，Freescale。基于ARM低成本和高效的处理器设计方案，得到授权的厂商生产了多种多样的的处理器、 单片机以及片上系统(SoC)。这种商业模式就是所谓的“知识产权授权”。



1.3 ARM发展历史

ARMv7架构的闪亮登场。在这个版本中，内核架构首次从单一款式变成3种款式。

款式A：设计用于高性能的“开放应用平台”——越来越接近电脑了

款式R：用于高端的嵌入式系统，尤其是那些带有实时要求的——又要快又要实时。

款式M：用于深度嵌入的，单片机风格的系统中——本书的主角。

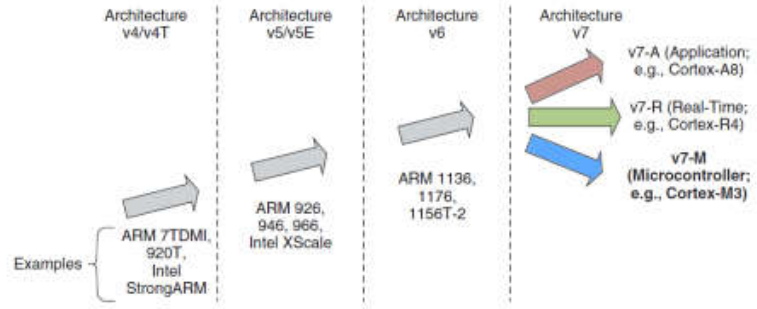


图1.2 ARM处理器架构进化史

2 Cortex-M3 概览和基础

2.1 M3架构图

Cortex-M3 是一个32位处理器内核。内部的数据路径是32位的，寄存器是32位的，存储器接口也是32位的。 CM3采用了哈佛结构，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不悖。这样一来数据访问不再占用指令总线，从而提升了性能。Both 小端模式和大端模式都是支持的。

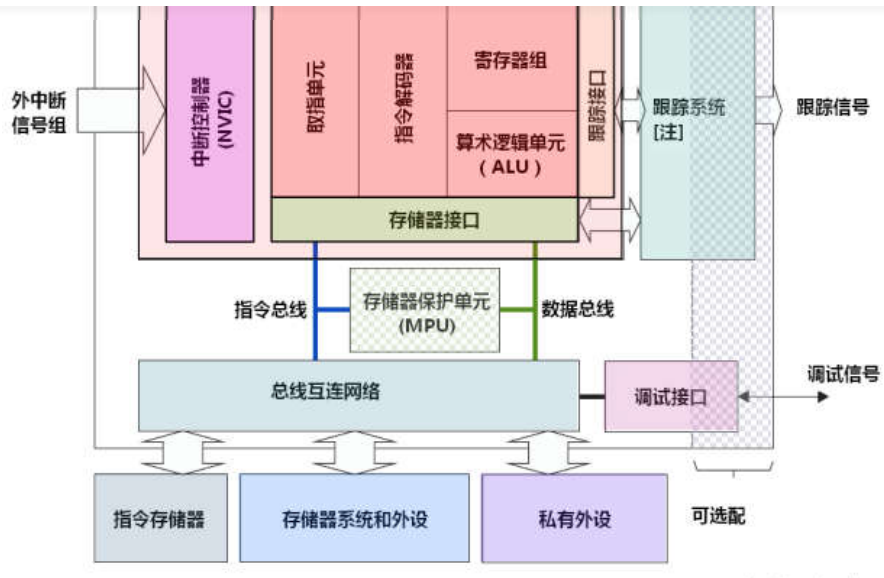


图 2.1 Cortex-M3 的一个简化视图

2.2 寄存器组

Cortex-M3 处理器拥有 R0-R15 的寄存器组。其中 R13 作为堆栈指针 SP。 SP 有两个，但在同一时刻只能有一个可以看到，这也就是所谓的“banked”寄存器。复位后，寄存器默认值不确定。

R0-R12通用寄存器：R0-R12 都是32位通用寄存器，用于数据操作。但是注意：绝大多数16位 Thumb 指令只能访问 R0-R7，而 32 位 Thumb-2 指令可以访问所有寄存器。

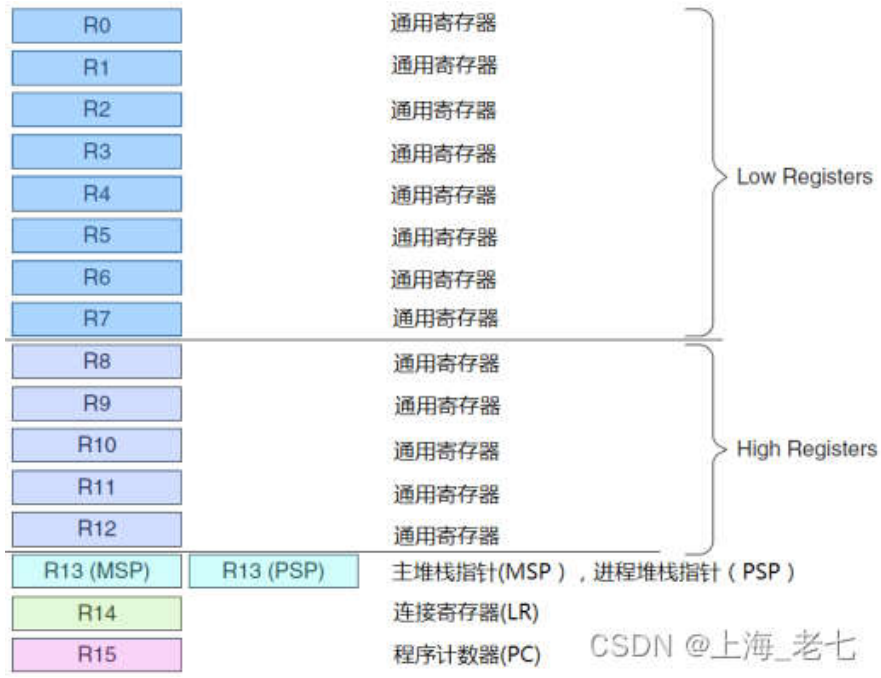
Banked R13两个堆栈指针：Cortex-M3 拥有两个堆栈指针，然而它们是 banked，因此任一时刻只能使用其中的一个。

- 主堆栈指针（MSP）：复位后缺省使用的堆栈指针，用于操作系统内核以及异常处理例程（包括中断服务例程）
- 进程堆栈指针（PSP）：由用户的应用程序代码使用。

堆栈指针的最低两位永远是 0，这意味着堆栈总是 4 字节对齐的。

R14链接寄存器：当呼叫一个子程序时，由 R14 存储返回地址。

R15程序计数寄存器：指向当前的程序地址。如果修改它的值，就能改变程序的执行流（很多高级技巧就在这里面）。



简单总结：
R0-R12 通用寄存器
R13-------SP（两种）
R14-------LR
R15 -------PC

特殊功能寄存器：
状态
控制： 模式控制，别的文件有：

特殊功能寄存器，Cortex-M3 还在内核水平上搭载了若干特殊功能寄存器，包括：

- 程序状态字寄存器组（PSRs）
- 中断屏蔽寄存器组（PRIMASK, FAULTMASK, BASEPRI）
- 控制寄存器（CONTROL）

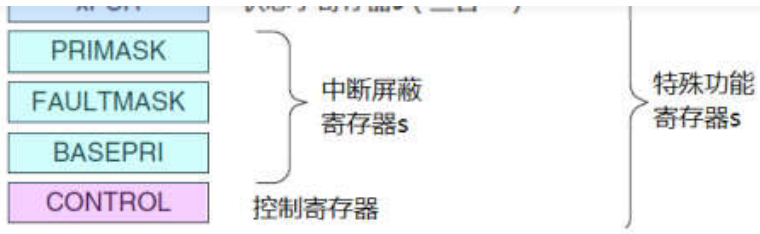


图 2.3： Cortex-M3 中的特殊功能寄存器集合 CSDN @上海_老七

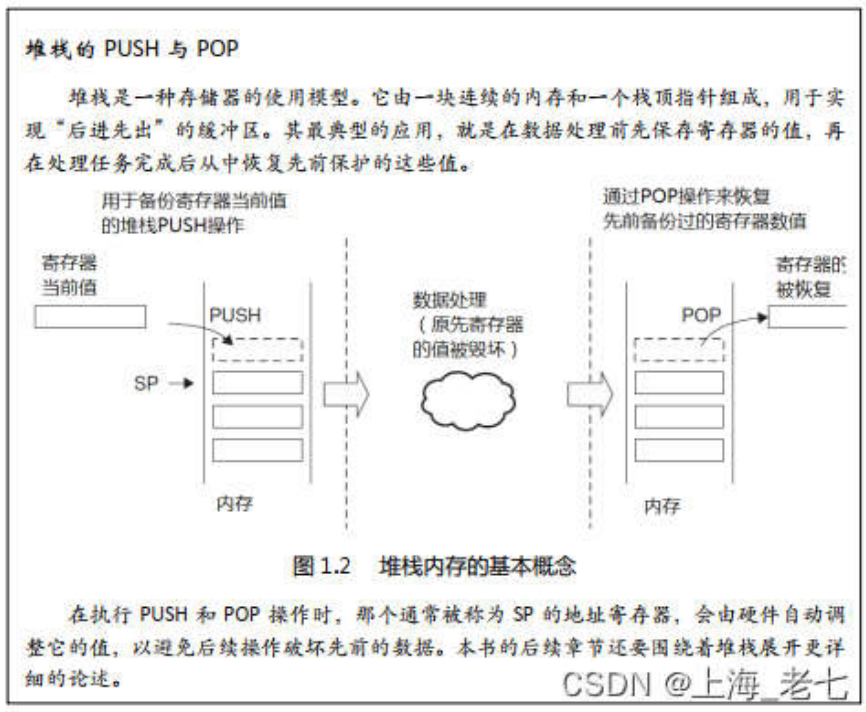
Cortex-M3 中的特殊能寄存器只能被专用的 MSR/MRS 指令访问，而且它们也没有与之相关联的访问地址。

```
MRS <gp_reg>, <special_reg> ;读特殊功能寄存器的值到通用寄存器
MSR <special_reg>, <gp_reg> ;写通用寄存器的值到特殊功能寄存器
```

寄存器	功能
xPSR	记录 ALU 标志（0 标志，进位标志，负数标志，溢出标志），执行状态，以及当前正服务的中断号
PRIMASK	除能所有的中断——当然了，不可屏蔽中断（NMI）才不甩它呢。
FAULTMASK	除能所有的 fault——NMI 依然不受影响，而且被除能的 faults 会“上访”，见后续章节的叙述。
BASEPRI	除能所有优先级不高于某个具体数值的中断。
CONTROL	定义特权状态（见后续章节对特权的叙述），并且负责使用第二个堆栈指针

2.2.1 R13堆栈指针

要注意的是，并不是每个程序都要用齐两个堆栈指针才算圆满。简单的应用程序只使用 MSP 就够了。堆栈指针用于访问堆栈，并且 PUSH 指令和 POP 指令默认使用 SP。



通常在进入一个子程序后，第一件事就是把寄存器的值先 PUSH 入堆栈中，在子程序退出前再 POP 曾经 PUSH 的那些寄存器。另外， PUSH 和 POP 还能一次操作多个寄存器。

操作一个寄存器

```
PUSH    {R0}        ; *(--R13)=R0。R13是long*的指针
POP     {R0}        ; R0= *R13++
```


操作多个寄存器

```
PUSH    {R0-R7, R12, R14}     ; 保存寄存器列表
…                             ; 执行处理
POP {R0-R7, R12, R14}         ; 恢复寄存器列表
BX R14                        ; 返回到主调函数
```


Cortex-M3 的堆栈实现

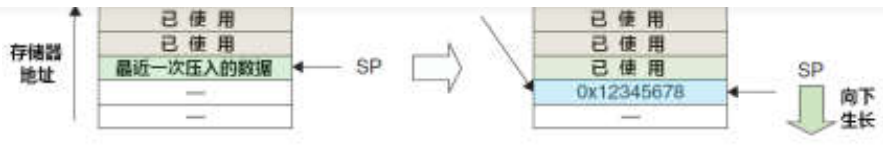


图 3.13 Cortex-M3 堆栈的 PUSH 实现方式

POP 操作刚好相反：先从 SP 指针处读出上一次被压入的值，再把 SP 指针自增 4。

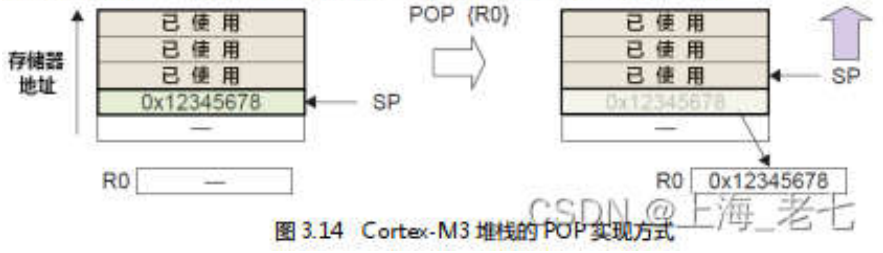


图 3.14 Cortex-M3 堆栈的 POP 实现方式

2.2.2 R14：连接寄存器（LR）

R14 是连接寄存器（LR）。在一个汇编程序中，你可以把它写作 both LR 和 R14。 LR 用于在调用子程序时存储返回地址。例如，当你在使用 BL(分支并连接， Branch and Link)指令时，就自动填充 LR的值。

```
main        ;主程序
    …
    BL  function1      ; 使用“分支并连接”指令呼叫 function1
                   ; PC= function1，并且 LR=main 的下一条指令地址
    …

Function1
    …              ; function1 的代码
    BX  LR         ; 函数返回（如果 function1 要使用 LR，必须在禁用前 PUSH，
                   ; 否则返回时程序就可能跑飞了——译注）
```


赋值PC/LR时，LSB总是1，用以表明是Thumb状态下执行，倘若写了0，就会产生fault异；但是读取PC时LSB是0，LR的LSR是1。

2.2.3 程序状态寄存器组（PSRs或曰PSR）

程序状态寄存器在其内部又被分为三个子状态寄存器：

- 应用程序 PSR（APSR）
- 中断号 PSR（IPSR）
- 执行 PSR（EPSR）

通过 MRS/MSR 指令，这 3 个 PSRs 即可以单独访问，也可以组合访问（2 个组合，3 个组合都可以）。当使用三合一的方式访问时，应使用名字“xPSR”或者“PSR”。

	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											
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

图 3.3 Cortex-M3 中的程序状态寄存器（xPSR）

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception Number				

图 3.4 合体后的程序状态寄存器(xPSR)

2.2.4 中断屏蔽寄存器组PRIMASK, FAULTMASK 和 BASEPRI

名字	功能描述
PRIMASK	这是个只有单一比特的寄存器。在它被置 1 后，就关掉所有可屏蔽的异常，只剩下 NMI 和硬 fault 可以响应。它的缺省值是 0，表示没有关中断。
FAULTMASK	这是个只有 1 个位的寄存器。当它置 1 时，只有 NMI 才能响应，所有其它的异常，甚至是硬 fault，也通通闭嘴。它的缺省值也是 0，表示没有关异常。
BASEPRI	这个寄存器最多有 9 位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成 0，则不关闭任何中断，0 也是缺省值。

对于时间-关键任务而言，恰如其分地使用 PRIMASK 和 BASEPRI 来暂时关闭一些中断是非常重要的。而 FAULTMASK 则可以被 OS 用于暂时关闭 fault 处理机能，这种处理在某个任务崩溃时可能需要。因为在任务崩溃时，常常伴随着一大堆 faults。在系统料理“后事”时，通常不再需要响应这些 fault——人死帐清。总之 FAULTMASK 就是专门留给 OS 用的。

注意：如果是在idle流程中观察PRIMASK，由于idle流程锁中断了，连接仿真器时刚从WFI指令退出，还停留在idle流程中，因此PRIMASK是1。

2.2.5 控制寄存器（CONTROL）

控制寄存器有两个用途，1用于定义特权级别，其二用于选择当前使用哪

CONTROL[1]：在 Cortex-M3 的 handler 模式中，CONTROL[1]总是 0。在线程模式中则可以为 0 或 1。因此，仅当处于特权级的线程模式下，此位才可写，其它场合下禁止写此位。
CONTROL[0]：仅当在特权级下操作时才允许写该位。一旦进入了用户级，唯一返回特权级的途径，就是触发一个（软）中断，再由服务例程改写该位。

表 3.3 Cortex-M3 的 CONTROL 寄存器

位	功能
CONTROL[1]	堆栈指针选择 0=选择主堆栈指针 MSP（复位后的缺省值） 1=选择进程堆栈指针 PSP 在线程或基础级（没有在响应异常——译注），可以使用 PSP。在 handler 模式下，只允许使用 MSP，所以此时不得往该位写 1。
CONTROL[0]	0=特权级的线程模式 1=用户级的线程模式 **Handler 模式永远都是特权级的。**

2.3 操作模式和特权级别

Cortex-M3 支持 2 个模式和两个特权等级。

图 2.4 Cortex-M3 下的操作模式和特权级别

操作模式：处理者模式(handler mode)和线程模式（thread mode）。引入两个模式的本意，是用于区别普通应用程序的代码和异常服务例程的代码——包括中断服务例程的代码。
特权分级：特权级和用户级。这可以提供一种存储器访问的保护机制，使得普通的用户程序代码不能意外地，甚至是恶意地执行涉及到要害的操作。处理器支持两种特权级，这也是一个基本的安全模型。
特权级和用户级区别：在 CM3 运行主应用程序时（线程模式），既可以使用特权级，也可以使用用户级；但是异常服务例程必须在特权级下执行。复位后，处理器默认进入线程模式，特权极访问。在特权级下，程序可以访问所有范围的存储器并且可以执行所有指令。
特权级和用户级切换：在特权级下的程序可以为所欲为，但也可能会把自己给玩进去——切换到用户级。一旦进入用户级，再想回来就得走“法律程序”了——用户级的程序不能简简单单地试图改写 CONTROL 寄存器就回到特权级，它必须先“申诉”：执行一条系统调用指令(SVC)。这会触发 SVC 异常，然后由异常服务例程（通常是操作系统的一部分）接管，如果批准了进入，则异常服务例程修改 CONTROL 寄存器，才能在用户级的线程模式下重新进入特权级。事实上，从用户级到特权级的唯一途径就是异常：如果在程序执行过程中触发了一个异常，处理器总是先切换入特权级， 并且在异常服务例程执行完毕退出时，返回先前的状态。
引入特权级和用户级目的：能够在硬件水平上限制某些不受信任的或者还没有调试好的程序，不让它们随便地配置涉及要害的寄存器，因而系统的可靠性得到了提高。

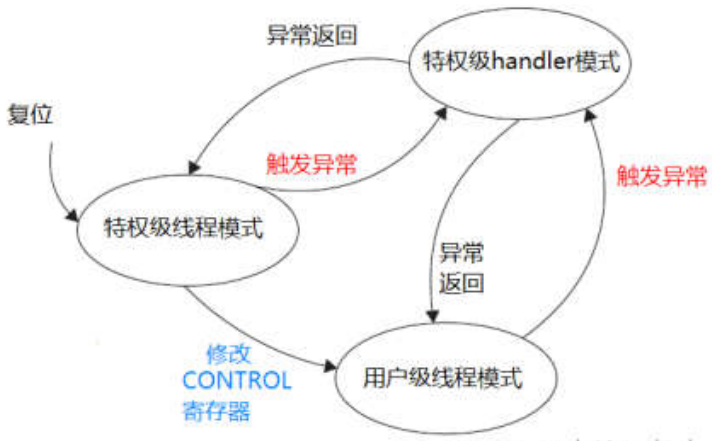


图 2.5 合法的操作模式转换图

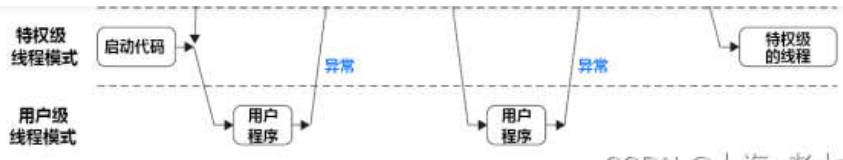


图 3.7 特权级和处理器模式转换图

2.4 嵌套向量中断控制器

NVIC 共支持 1 至 240 个外部中断输入（通常外部中断写作 IRQs）。具体的数值由芯片厂商在设计芯片时决定。此外，NVIC 还支持一个“永垂不朽”的不可屏蔽中断（NMI）输入。NMI 的实际功能亦由芯片制造商决定。在某些情况下，NMI 无法由外部中断源控制。

可嵌套中断支持
可嵌套中断支持的作用范围很广，覆盖了所有的外部中断和绝大多数系统异常。外在表现是，这些异常都可以被赋予不同的优先级。当前优先级被存储在 xPSR 的专用字段中。当一个异常发生时，硬件会自动比较该异常的优先级是否比当前的异常优先级更高。如果发现来了更高优先级的异常，处理器就会中断当前的中断服务例程（或者是普通程序），而服务新来的异常——即立即抢占。

向量中断支持
向量中断与直接中断的好处：当开始响应一个中断后， CM3 会自动定位一张向量表，并且根据中断号从表中找出 ISR 的入口地址，然后跳转过去执行。不需要像以前的 ARM 那样， 由软件来分辨到底是哪个中断发生了，也无需半导体厂商提供私有的中断控制器来完成这种工作。这么一来，中断延迟时间大为缩短。

中断延迟大大缩短
Cortex-M3 为了缩短中断延迟，引入了好几个新特性。包括自动的现场保护和恢复，以及其它的措施，用于缩短中断嵌套时的 ISR 间延迟。咬尾中断、晚到中断。

动态优先级调整支持
软件可以在运行时期更改中断的优先级。如果在某ISR中修改了自己所对应中断的优先级，而且这个中断又有新的实例处于悬起中（pending），也不会自己打断自己，从而没有重入(reentry) 风险。

中断可屏蔽
既可以屏蔽优先级低于某个阈值的中断/异常(设置BASEPRI寄存器)，也可以全体封杀(设置PRIMASK和FAULTMASK寄存器)。这是为了让时间关键（ time-critical）的任务能在死线 (deadline，或曰最后期限)到来前完成，而不被干扰。

Cortex-M3 支持大量异常，包括 16-4-1=11 个系统异常，和最多 240 个外部中断——简称 IRQ。具体使用了这 240 个中断源中的多少个，则由芯片制造商决定。由外设产生的中断信号，除了 SysTick的之外，全都连接到 NVIC 的中断输入信号线。典型情况下，处理器一般支持 16 到 32 个中断，当然也有在此之外的。

作为中断功能的强化， NVIC 还有一条 NMI 输入信号线。 NMI 究竟被拿去做什么，还要视处理器的设计而定。在多数情况下， NMI 会被连接到一个看门狗定时器，有时也会是电压监视功能块，以便在电压掉至危险级别后警告处理器。 NMI 可以在任何时间被激活，甚至是在处理器刚刚复位之后。
表 3.4 列出了 Cortex-M3 可以支持的所有异常。有一定数量的系统异常是用于 fault 处理的，它们可以由多种错误条件引发。 NVIC 还提供了一些 fault 状态寄存器，以便于 fault 服务例程找出导致异常的具体原因。

0	N/A	N/A	没有异常在运行
1	复位	-3（最高）	复位
2	NMI	-2	不可屏蔽中断（来自外部 NMI 输入脚）
3	硬(hard) fault	-1	所有被除能的 fault，都将“上访”成硬 fault。除能的原因包括当前被禁用，或者被 PRIMASK 或 BASPRI 掩蔽。
4	MemManage fault	可编程	存储器管理 fault，MPU 访问犯规以及访问非法位置均可引发。企图在“非执行区”取指也会引发此 fault
5	总线 fault	可编程	从总线系统收到了错误响应，原因可以是预取流产（Abort）或数据流产，或者企图访问协处理器
6	用法(usage) Fault	可编程	由于程序错误导致的异常。通常是使用了一条无效指令，或者是非法的状态转换，例如尝试切换到 ARM 状态
7-10	保留	N/A	N/A
11	SVCall	可编程	执行系统服务调用指令（SVC）引发的异常
12	调试监视器	可编程	调试监视器（断点，数据观察点，或者是外部调试请求
13	保留	N/A	N/A
14	PendSV	可编程	为系统设备而设的“可悬挂请求”（pendable request）
15	SysTick	可编程	系统滴答定时器（也就是周期性溢出的时基定时器——译注）
16	IRQ #0	可编程	外中断#0
17	IRQ #1	可编程	外中断#1
...	...	...	...
255	IRQ #239	可编程	外中断#239

2.5 存储器映射

待补充

2.6 总线接口

Cortex-M3 内部有若干个总线接口，以使 CM3 能同时取址和访内（访问内存），它们是：

指令和存储总线（两条）
系统总线
私有外设总线
指令和存储总线：有两条代码存储区总线负责对代码存储区的访问，分别是 I-Code 总线和 D-Code 总线。前者用于取指，后者用于查表等操作，它们按最佳执行速度进行优化。
系统总线：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：负责一部分私有外设的访问，主要就是访问调试组件。它们也在系统级存储区。

2.7 存储器保护单元（MPU）

Cortex-M3 有一个可选的存储器保护单元。配上它之后，就可以对特权级访问和用户级访问分别施加不同的访问限制。当检测到犯规（violated）时， MPU 就会产生一个 fault 异常，可以由fault 异常的服务例程来分析该错误，并且在可能时改正它。
MPU 有很多玩法。最常见的就是由操作系统使用 MPU，以使特权级代码的数据，包括操作系统本身的数据不被其它用户程序弄坏。 MPU 在保护内存时是按区管理的(“区”的原文是 region，以后不再中译此名词——译注)。它可以把某些内存 region 设置成只读，从而避免了那里的内容意外被更改；还可以在多任务系统中把不同任务之间的数据区隔离。一句话，它会使嵌入式系统变得更加健壮，更加可靠（很多行业标准，尤其是航空的，就规定了必须使用 MPU 来行使保护职能——译注）。

2.8 指令集

32位的ARM指令集 。对应处理器状态：ARM状态。
16位的Thumb指令集。对应处理器状态：Thumb状态，代码密度高，功能相对来说就少一些。
Thumb-2真不愧是一个突破性的指令集。它强大，它易用，它轻佻，它高效。Thumb-2是16位Thumb指令集的一个超集，在Thumb-2中， 16位指令首次与32位指令并存，结果在Thumb状态下可以做的事情一下子丰富了许多，同样工作需要的指令周期数也明显下降。 在支持了both 16位和32位指令之后，就无需烦心地把处理器状态在Thumb和ARM之间来回的切换了。

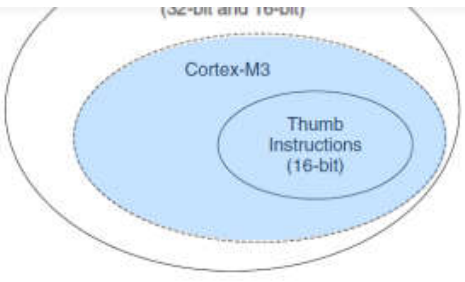


图1.4 Thumb-2指令集与Thumb指令集的关系

Cortex-M3 只使用Thumb-2指令集。这是个了不起的突破，因为它允许 32 位指令和 16 位指令水乳交融，代码密度与处理性能两手抓，两手都硬。而且虽然它很强大，却依然易于使用。注意：CM3 并不支持所有的 Thumb-2 指令。

在过去，做 ARM 开发必须处理好两个状态。这两个状态是井水不犯河水的，它们是：32 位的ARM 状态和16 位的 Thumb 状态。当处理器在 ARM 状态下时，所有的指令均是 32 位的（哪怕只是个"NOP"指令），此时性能相当高。而在 Thumb 状态下，所有的指令均是16 位的，代码密度提高了一倍。不过，thumb 状态下的指令功能只是 ARM 下的一个子集，结果可能需要更多条的指令去完成相同的工作，导致处理性能下降。

为了取长补短，很多应用程序都混合使用 ARM 和 Thumb 代码段。然而，这种混合使用是有额外开销（overhead）的，时间上的和空间上的都有，主要发生在状态切换之时。另一方面， ARM 代码和 Thumb 代码需要以不同的方式编译，这也增加了软件开发管理的复杂度。

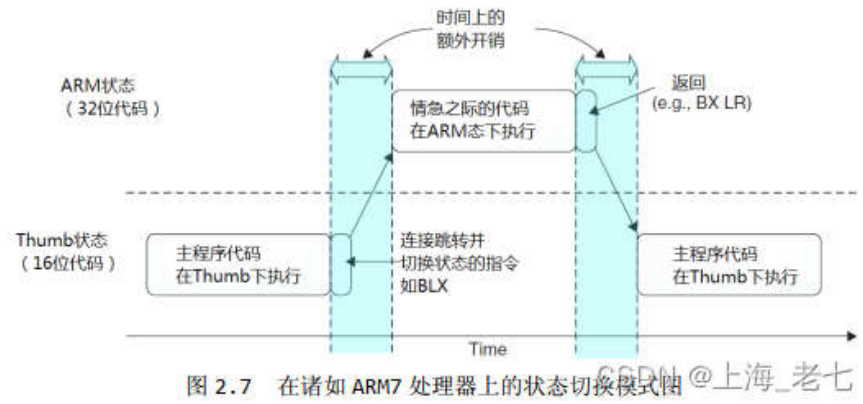


图 2.7 在诸如 ARM7 处理器上的状态切换模式图

伴随着 Thumb-2 指令集的横空出世，终于可以在单一的操作模式下搞定所有处理了，再也没有来回切换的事来烦你了。 事实上， Cortex-M3 内核干脆都不支持 ARM 指令，中断也在 Thumb 态下处理（以前的 ARM 总是在 ARM 状态下处理所有的中断和异常）。这可不是小便宜，它使 CM3 在好几个方面都比传统的ARM处理器更先进：

消灭了状态切换的额外开销，节省了 both 执行时间和指令空间。

消除文件编译管理：不再需要把源代码文件分成按ARM编译的和按Thumb编译的，软件开发的管理大大减负。

无需再反复地求证和测试：究竟该在何时何地切换到何种状态下，我的程序才最有效率。开发软件容易多了。

2.9 调试支持

Cortex-M3 在内核水平上搭载了若干种调试相关的特性。最主要的就是程序执行控制，包括停机(halting)、单步执行(stepping)、指令断点、数据观察点、寄存器和存储器访问、性能速写（profiling）以及各种跟踪机制。

Cortex-M3 的调试系统基于ARM最新的 CoreSight 架构。不同于以往的 ARM 处理器，内核本身不再含有 JTAG 接口。取而代之的，是 CPU 提供称为"调试访问接口(DAP)"的总线接口。通过这个总线接口，可以访问芯片的寄存器，也可以访问系统存储器，甚至是在内核运行的时候访问！对此总线接口的使用，是由一个调试端口(DP)设备完成的。 调试端口DPs 不属于CM3内核，但它们是在芯片的内部实现的。目前可用的 DPs 包括 SWJ-DP(既支持传统的 JTAG 调试，也支持新的串行线调试协议)，另一个 SW-DP 则去掉了对 JTAG 的支持。另外，也可以使用 ARM CoreSignt 产品家族的 JTAG-DP模块。这下就有 3 个 DPs 可以选了，芯片制造商可以从中选择一个，以提供具体的调试接口（通常都是选 SWJ-DP）。

此外， CM3 还能挂载一个所谓的"嵌入式跟踪宏单元（ETM）"。 ETM 可以不断地发出跟踪信息，这些信息通过一个被称为"跟踪端口接口单元（TPIU）"的模块而送到内核的外部，再在芯片外面使用一个"跟踪信息分析仪"，就可以把 TIPU 输出的"已执行指令信息"捕捉到，并且送给调试主机——也就是 PC。所有这些调试组件都可以由 DAP 总线接口来控制，CM3 内核提供 DAP 接口。此外，运行中的程序也能控制它们。所有的跟踪信息都能通过 TPIU 来访问到。