目 录

第 5 章 ARM 处理器体系结构和编程模型 1

5.1 ARM 体系结构与 ARM 处理器概述 1

5.1.1 指令集体系结构与微架构 1

5.1.2 ARM 处理器体系结构简介 3

5.1.3 ARM 处理器主要产品系列简介 9

5.2 CORTEX-M3/M4 处理器结构 23

5.2.1 Cortex-M3/M4 处理器概述及指令集架构 23

5.2.2 Cortex-M3/M4 处理器结构 25

5.2.3 存储器管理 30

5.2.4 总线系统 31

5.2.5 异常与中断处理 34

5.3 CORTEX-M3/M4 的编程模型 38

5.3.1 操作状态与操作模式 38

5.3.2 常规寄存器 40

5.3.3 特殊寄存器 43

5.3.4 堆栈结构 49

5.4 CORTEX-M 处理器存储系统 52

5.4.1 存储器映射 52

5.4.2 连接存储器和外设 53

5.4.3 存储器的端模式 54

5.4.4 非对准数据的访问 56

5.4.5 位段操作 57

5.4.6 存储器访问权限 61

5.4.7 存储器访问属性 62

5.4.8 排他访问 64

5.4.9 存储器屏障 65

5.5 CORTEX-M 处理器的异常处理 65

5.5.1 Cortex-M 异常管理模型 66

5.5.2 向量表重定位机制 71

5.5.3 异常请求和挂起 72

5.5.4 NVIC 寄存器 73

5.5.5 SCB 寄存器 76

5.6 习题 78

第5章 ARM 处理器体系结构和编程模型

本章主要介绍 ARM 处理器的体系结构和编程模型。5.1 节对 ARM 处理器体系结构版本的演进过程、典型处理器产品及主要特性做简要概述。5.2 ~ 5.5 节详细介绍目前获得广泛的 ARM Cortex-M3 和 ARM Cortex-M4 处理器，其中 5.2 节介绍这两款处理器的结构以及主要部件的功能；5.3 节介绍 Cortex-M3/4 处理器的编程模型，包括处理器的工作模式（Operation Modes）、寄存器组织、特殊寄存器以及堆栈机制，并与经典的 ARM 处理器进行对比；5.4 节介绍 Cortex-M3/4 处理器的存储系统，包括存储空间的区域划分、操作特性、访问权限、访问属性以及总线连接方式；5.5 节介绍 Cortex-M3/4 处理器的异常与中断管理，包括异常管理模型，向量表重定位机制、异常挂起，以及相关的特殊寄存器。

5.1 ARM 体系结构与 ARM 处理器概述

5.1.1 指令集体系结构与微架构

本书第二章已经介绍了计算机体系结构的基本含义。广义上，计算机体系结构是指对计算机的的逻辑特征、原理特征、结构特征和功能特征的一种抽象描述。而在狭义上，计算机体系结构可以认为就是处理器的指令集体系结构（Instruction Set Architecture，ISA），也就是从程序员角度所看到的计算机概念结构和功能特征。而计算机各个功能单元的逻辑设计、硬件实现和部件之间的互联组织则属于计算机组成所要研究的内容。

1. 指令集体系结构

计算机的核心是处理器，而处理器的各种功能主要通过指令来体现。描述处理器指令及其功能、组织方式的规范称为指令集体系架构 ISA。

具体地说，ISA 是描述软件如何使用硬件的一种规范和约定，是从编程者的角度对处理器的一种逻辑抽象，是指程序员在使用该处理器编程时，能看到或者能用到的处理器资源以及使用方式、工作原理及其相互间的关系。ISA 主要规定了以下内容：

- 可执行指令集合，包括指令格式、操作种类以及每种操作所对应的操作数规范；
- 指令可以接受的操作数类型；
- 寄存器组结构，包括每个寄存器的名称、编号、长度和用途；
- 存储空间的大小和编址方式；
- 操作数在存储空间的存放格式，如大端还是小端；
- 操作数的寻址方式；
- 指令执行过程的控制方式，包括程序计数器和条件码定义等。

以上内容可以归纳为指令系统和寄存器组模型两个部分，也可以理解为通常所说的基本编程模型。ISA 告诉软件程序员，处理器能做什么事；ISA 则告诉硬件设计师，处理器应该完成的任务以及需要实现的功能。ISA 在计算机系统中的地位和作用可用图 5.1 表示，从图中可以看出，ISA 是介于硬件和软件之间的中间抽象层，也是硬件和软件之间的桥梁和纽带。

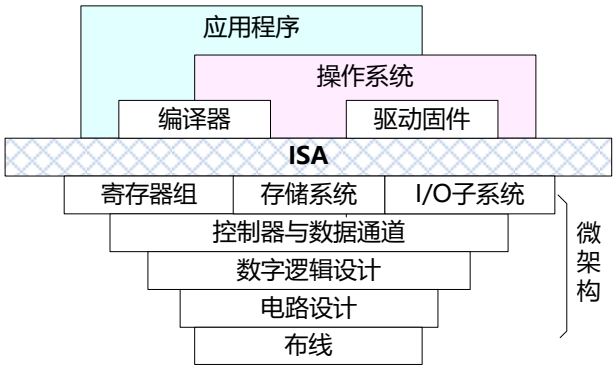

图 5.1 指令集体系结构在计算机中所处的位置

2. 微架构

ISA 的硬件实现方式称为微架构（Microarchitecture），即数字电路以何种方式来实现处理器的各种功能，包括运算器、控制器、流水线、超标量和存储系统结构等内容，也就是计算机的组织和实现技术。

基于相同 ISA 的处理器可以有不同的硬件实现方案，例如采用不同的流水线、不同的存储和缓存结构等。也就是说，同一个 ISA 可以通过不同的微架构来实现。但是，只要基于同一个 ISA，即使是采用不同微架构实现的各种处理器，在软件层面可以做到相互兼容。例如，本书 2.7 节所介绍的 ARM7TDMI 和 ARM9TDMI 两款 ARM 处理器，都是基于 ARMv4T 系统结构版本（相同的 ISA），两者的微架构却有较大区别。ARM7TDMI 采用的是冯·诺依曼结构，流水线分为三级；而 ARM9TDMI 采用的是哈佛结构，流水线分为五级。但是两者在软件层面可以做到完全兼容。又如，同样支持 x86 指令集体系结构的 AMD 处理器，其设计和实现方式与 Intel 处理器存在许多差别，所采用的微架构与 Intel 并不相同，但同样都可以安装和运行 Windows 操作系统或其他用户软件，用户在使用过程中感觉不到彼此之间的差异。

3. 处理器体系结构的分类

如前所述，ISA 偏向于处理器的软件层面，而微架构偏向于处理器的硬件层面。因此，处理器体系结构有基于 ISA 和基于微架构的两种分类方法。基于 ISA，处理器可以分为 CISC 和 RISC 两大类。基于微架构，处理器可以分为冯·诺依曼结构和哈佛结构两大类，如图 5.2 所示。

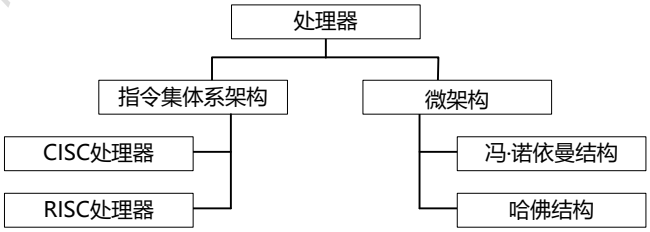

图 5.2 处理器体系结构分类

本书 2.1.2 节已对冯 • 诺依曼结构做了详细介绍，该结构的特点是将程序存储器和数据存储器合并在一起，统一编址，共享同一条总线。冯 • 诺依曼结构的取指和取操作数都使用同一条总线上，通过分时复用的方式进行的，不能同时读取指令和操作数，因而不利于采用流水线作业方式。但是冯 •诺依曼结构电路简单，易于实现，Intel x86、MIPS 以及 ARM7 系列和 Cortex-M 系列中的部分低成本产品都采用这种结构。

哈佛结构是将程序和数据存储在不同的存储空间中，即程序存储器和数据存储器是两个

相互独立的存储器，每个存储器独立编址、独立访问。同时，系统中设置了两条相互独立的总线（包括地址总线、数据总线和控制总线），分别连接程序存储器和数据存储器，指令和数据可在两条总线上同时并行传送。采用哈佛结构的流水线处理器，可以消除流水线上取指和取操作数之间的资源相关。

但是哈佛结构较为复杂，与外设以及外部扩展存储器的连接难度较大，故早期通用 CPU 较少采用这种结构。此外，哈佛结构因其指令和数据分开存储，各自有不同总线，所以指令和数据宽度可以不同。例如 Microchip 公司的 PIC16 处理器，其指令位宽是 14 位，数据位宽是 8 位。采用哈佛结构的处理器主要有 Motorola 公司的 MC68 系列和 Zilog 公司 Z8 系列处理器。ARM 公司 ARM9 系列之后的高性能处理器大都采用的也是哈佛结构。

为了兼具冯•诺依曼结构与哈佛结构的优点，有些处理器对哈佛结构进行了一些简化。例如，曾经获得广泛应用的 Intel MCS-51 单片机，使用了两个独立的存储器模块分别存储指令和数据，每个存储模块不允许指令和数据并存，但是程序存储器和数据存储器还是共用一条公共总线，仍然采用分时方式访问程序存储器和数据存储器。还有些处理器在内部分别配置了指令和数据 Cache、TCM[1]（Tightly Coupled Memory，紧耦合存储器）或者 SRAM，内部使用两条独立的总线对这些片内存储器进行访问。但是处理器对外只配置了一条总线，在片外仍然将指令和数据存储在一个存储器中，以减少硬件实现的复杂性。

ARM 处理器也可以采用图 5.2 所示的分类方法，从 ISA 角度进行分类，ARM 处理器属于 RISC 处理器，但有多个体系结构版本；而在微架构方面，ARM 处理器有基于冯诺依曼结构和哈佛结构两种不同实现方式。

ARM 体系结构往往也被认为是 ARM 指令集体系结构 ISA，但是本书所介绍的 ARM 处理器体系结构除了 ISA 以外，还包括了编程模型、存储器组织以及异常/中断处理机制等内容。

5.1.2 ARM 处理器体系结构简介

1. ARM 处理器的起源

1978 年，剑桥大学的奥地利籍物理学家 Herman Hauser 和他的同事 Andy Hopper，以及另一位就职于 Sinclair Research 的朋友 Chris Curry 合作创办了一家名为 CPU 的公司（Cambridge Processor Unit），他们的第一个产品是博彩机上的控制器，取名为 Acorn System 1。在产品取得成功之后，公司也更名为 Acorn Computer Ltd。因 Acorn 公司产品研发的需要，急需一款质优价廉的 CPU，在遍寻无着的情况下，只能依靠两位刚刚毕业于剑桥大学的博士 Sophie Wilson（女）和 Steve Furber 自行研发。两位博士也曾试图寻求 Motorola 公司和 Intel 公司的帮助，结果却是无功而返。但是他俩从 MIPS 项目中深受启发，回到剑桥后两人开始着手研制基于 RISC 架构的处理器。Sophie Wilson 负责 ISA 设计，Steve Furber 负责微架构实现。在历时 17 个月之后，两人领衔研发的首款 32 位商用 RISC 处理器加电运行获得成功。ARM 公司将新的 CPU 命名为 Acorn RISC Machine 1（简称 ARM1）。

[1] 紧耦合内存 TCM 是一种通过专用总线与 CPU 相连的高速存储器，因为使用的是专用总线，所以可以实现处理器对 TCM 的快速访问，性能与 Cache 相当。现在 TCM 都与处理器集成在一块芯片上，早期也有置于片外的，但需要紧密地部署在处理器周边。TCM 与 Cache 的主要区别包括：TCM 具有物理地址、需要占用内存空间、没有 Cache 特有的不可预测性。TCM 一般用来存放需要快速执行的程序，如异常处理程序和实时任务程序。此外，TCM 还用来存放寄存器数据以及局部属性不适合高速缓存的数据，如中断向量表和堆栈等。

在对 ARM1 进行的一系列测试过程中，发现其功耗非常低。事实上，低功耗并非是 ARM 1 最初的设计目标，这纯属是无心插柳之举，究其原因应该归功于 ARM1 简单的结构以及较少的门电路数量。几乎与此同时，Apple 公司为了开发手持式 PDA（Personal Digital Assistant）产品，正在寻找一款低功耗处理器，于是 ARM1 顺理成章地成为了 Apple 公司第一款 PDA 产品 Newton 中的处理器。虽然 Newton 后来的销售情况并不理想，但是与 Apple 的合作迅速提高了 Acorn 公司的知名度。在 ARM 1 之后，Acorn 公司又陆续推出了 ARM 2 和 ARM 3，并逐渐坚定了自己的设计理念：低成本、低功耗和高性能（low-cost、low-power and high-performance）。

1990 年 11 月，为了进一步加深与 Apple 公司的合作，Acorn 公司对 ARM 处理器研发部门实行了剥离，并将其作为出资与 Apple 和 VLSI（ARM 芯片代工厂商）合资成立了 ARM 公司，其产品名称也由 Acorn RISC Machine 更名为 Advanced RISC Machine。ARM 公司成立之后，其商业模式逐步转向专门从事基于 RISC 技术的芯片设计、开发和授权业务，现已成为全球最大的 IP 供应商。因 ARM 处理器所具有的高性能、低成本和低功耗等特点，目前全世界超过 95%的智能手机和平板电脑处理器采用的都是基于 ARM 架构的处理器 。

从 ARM1 诞生之日起，ARM 公司到目前为此先后推出了 ARMv1~ARMv8 共 8 个大的体系结构版本（可以看作是 ISA），以下对各个版本的主要特点做简要介绍。

2. ARM 体系结构版本特性简介

1） V1 版

ARMv1 版架构是与 ARM 1 原型机同时诞生的，仅用于 ARM 1 原型机。ARM 1 原型机虽然是 32 位处理器，但是仅设计了 26 条地址总线，内存寻址空间只有 64MB，不过 64MB 已经超过了当时许多小型机的内存容量。ARMv1 版架构只提供了一些基本指令，如：

- 基本数据处理指令，没有乘法运算指令；
- 数据存取指令，可以对字节、半字和字数据进行存取；
- 控制转移指令，包括子程序调用及链接指令[2]；
- 供操作系统使用的软件中断（Software Interrupt，SWI）指令。

2） v2 版架构

为了提高数据处理能力，v2 版增加了乘法运算和乘加运算指令，以及若干协处理器操作指令。所谓乘加运算是指只用一条指令即可完成类似 a＋b×c→a 的运算。为了满足实时性应用的需求，v2 版增加了快速中断模式 FIQ（Fast Interrupt Request），以便 CPU 能够快速处理某些优先级较高的中断请求。此外，v2 版还增加了存储器与寄存器之间进行数据交换的 SWP（swap）指令，包括字交换指令 SWP 和字节交换指令 SWPB。但是 ARM 2 的内存寻址空间仍然只有 64MB。

基于 ARMv2（包括 v2a）版架构的 ARM 处理器有 ARM2、ARM2aS 和 ARM3。

3） v3 版架构

ARMv3 版架构对 ARM 体系结构进行了较大的升级和完善，将地址总线的数量增加到 32 条，内存寻址空间从 64MB 扩展到了 4GB，使得 ARM 处理器与同期其他主流 32 位处理器的内存寻址能力一致。此外，ARMv3 版架构还增加了若干新的寄存器和新的指令。在微架构方

[2] 所谓链接指令是指在转移时，将转移指令的下一条指令存入 R14 寄存器。需要返回时，使用 MOV 指令将 R14 寄存器的内容写入 PC，即可实现程序的返回。

面也首次集成了高速缓存 Cache。v3 版的主要改进包括：

- 增加一个新的程序状态寄存器（Current Program Status Register，CPSR），不再使用原来的 R15 寄存器保存程序执行状态；
- 增加一个新的程序状态保存寄存器（Saved Program Status Register，SPSR），在异常处理时用于保存 CPSR 的状态，以便异常返回后能够恢复原来的工作状态；
- 增加了 MRS 和 MSR 指令， 专门用来访问新增的 CPSR 和 SPSR 寄存器；
- 增加了从异常处理返回的指令功能；
- 新定义了中止（Abort）和未定义（Undefined）两种新的处理器工作模式；
- 将乘法运算和乘加运算指令扩展为 32 位长乘法运算和长乘加运算（长乘法运算：32 位×32 位＝64 位，长乘加运算：32 位×32 位+32 位＝64 位）。

4） v4 版架构

ARMv4 版架构包括 ARMv4 和 ARMv4T 两个子版本。基于 ARMv4 子版本架构的处理器主要是 ARM8 和 StrongARM（现归属于 Intel 公司），而 ARMv4T 版本被认为是 ARM 处理器发展史上的一个重要里程碑。本书 2.7 节介绍的 ARM7TDMI 和 ARM920T 处理器都是基于 ARMv4T 版本，尤其 ARM7T 系列处理器当属 ARM 公司的一款功勋产品，帮助 ARM 公司取代 MIPS 公司一举成为嵌入式处理器市场的领导者。

从 ARMv4T 版本开始，ARM 处理器引入了 Thumb 指令集。所谓 Thumb 指令集是 32 位的 ARM 指令集中最常用的一部分经过重新编码和压缩形成的一个 16 位指令集。Thumb 指令在执行时，被实时并且透明地解压成完整的 32 位 ARM 指令，并且没有任何性能损失。Thumb 指令集可以看作 ARM 指令集的一个子集，但在指令功能方面不如 ARM 指令集全面，某些复杂操作可能需要使用较多 Thumb 指令。

采用 Thumb 指令集主要有以下两方面的好处。其一是可以提高代码密度，Thumb 指令的代码容量一般只有 ARM 指令的 65%，减小存储容量意味着可以降低系统成本。其二是大多数外设的 I/O 接口位宽只有 8 位或者 16 位，还有一些存储系统的接口位宽也只有 8 位或者 16 位，在这种情况下，使用 Thumb 指令可以提高数据传送的效率。例如，当外部存储系统数据总线的位宽为 16 位时，采用 Thumb 指令的性能是 ARM 指令集的 160%。

在增加 Thumb 指令的同时，v4T 版将处理器的运行状态分为两个操作状态（Operation states，也称为工作状态）。执行 Thumb 指令集的状态称为 Thumb 状态，将执行 ARM 指令的状态称为 ARM 状态。处理器的操作状态是通过当前状态寄存器 CPSR 设定的，CPSR 中的 T 位（CPSR 的 bit[5]）置位为 1，处理器工作在 Thumb 状态；若 T 位为 0，处理器工作在 ARM 状态。在 Thumb 状态下，ARM 处理器仍然是 32 位的，只不过指令码长度只有 16 位，处理器取指、译码和执行的都是 Thumb 指令集。在程序运行过程中，可根据需要对 CPSR 中的 T 位进行修改，实现两种状态的自由切换。处理器的操作状态切换不会影响其它寄存器的内容和工作模式。无论处理器处于何种工作状态，在响应异常时，都自动进入 ARM 工作状态。

v4T 版在 v3 版的基础上还做了其他一些完善和扩展，例如：

- 增加了系统模式（sys），可以运行具有特权的操作系统程序对系统进行管理。在该模式下可以使用用户模式（usr）下的所有寄存器，并具有直接切换到其他模式的特权。
- 对软件中断指令 SWI 的功能做了完善；
- 增加了无符号字节以及半字数据的加载/存储指令（LDRB/STRB、LDRH/STRH）；
- 增加了有符号字节以及半字数据的加载指令（LDRSB、LDRSH）；
- 把没有使用的指令空间都作为“未定义指令”，并在“未定义指令（UND）”异常中

对其进行处理。

在 ARMv4T 版本架构发布的同时，ARM 处理器的微架构也做了较大的改进，主要体现在内核全面采用了流水线技术。基于该版本的处理器有 ARM7T 系列和 ARM9T 系列。

5） v5 版架构

ARMv5 版架构也有 ARMv5TE 和 ARMv5TEJ 两个子版本，子版本名称后缀的字母 T、E 和 J 分别表示支持 Thumb 指令、增强的 DSP 指令和 Java 硬件加速技术。不同的子版本所对应的微架构配置了相应的功能部件。

v5 版中增加了数字信号处理指令，并为协处理器提供了更多可选择的指令，同时也更加严格地定义了乘法指令对条件标志位的影响。v5 版架构在 v4 版的基础上增加了以下一些新的指令：

- 增加了支持有符号数的加减饱和运算指令。所谓饱和运算是指出现溢出时，结果等于最大或者最小可表示范围，避免出现更大的误差，对于音视频信号处理，饱和运算尤为重要；
- BLX 指令，将可返回的转移指令和状态切换指令合二为一；
- CLZ 指令：用于计算寄存器操作数最高位 0（前导 0）的个数，该指令可提高归一化运算、浮点运算以及整数除法运算的性能；也使中断优先级排队操作更为有效；
- BRK 指令，软件断点指令。

为实现 ARMv5 版架构新增加的各项功能，从 v5 版开始，ARM 处理器在微架构方面增加了或者可以选配如下部件：

- DSP 部件，使得 MCU 兼具 DSP 功能，可将 MCU 和 DSP 合二为一。
- 可以选配的矢量浮点处理器（Vector Float Processor，VFP），为 ARM 处理器提供浮点计算能力。
- 可以选配的 TCM 或者 TCM 接口，对于内核采用哈佛结构的处理器，可以同时选配指令 TCM 和数据 TCM。
- 名为 Jazelle DBX（Direct Bytecode eXecution）的 Java 硬件加速器。带有 Jazelle DBX 处理器可以直接运行 Java 虚拟机的字节码程序，可以更加流畅快捷地运行采用 Java 语言编写的各种手机游戏程序。
- 带有 AHB 或者 AHB Lite 总线接口。

基于 ARMv5 版架构的 ARM 处理器包含 ARM7EJ、ARM9E 和 ARM10E 三个系列的多款产品，可在性能、功耗和实时性等方面满足不同需求。这些处理器的名称和主要特性参见表 5.1，本节稍后将对具体产品做更详细的介绍。

6） v6 版架构

ARMv6 版架构发布于 2001 年，也是 ARM 体系结构演进过程中的一个重要里程碑。ARMv6 包含了 ARMv5TEJ 的所有指令，并引入了许多突破性的新技术，在多媒体处理、存储器管理、多处理器支持、数据处理、异常和中断响应等方面做了较大改进。

在 ISA 方面，ARMv6 版架构增加了以下功能和指令：

- 首次引入 Thumb-2 指令集（v6T2 增强版），可以混合执行 32 位的 AMR 指令和 16 位的 Thumb 指令，同时兼具 ARM 指令集的性能和 Thumb 指令集的代码密度等优点，这也是 ARMv6 版架构中最前卫的技术之一；
- 增加了 SIMD 指令。SIMD（Single Instruction Multiple Data，单指令流多数据流）普

遍应用于音视频信号处理。例如，使用 SIMD 指令可以同时计算立体声的左右声道；在视频和图形图像处理领域，每个像素的 RGB（红绿蓝三基色）元素可以用 8 位数据表示，使用 SIMD 指令可以对其进行并行处理。新增的 SIMD 指令使 ARM 处理器的音视频信号处理能力较之前提高了 2~4 倍；

- ❒ 支持混合大小端（Mixed-endian）和非对准（Unaligned）存储访问；
- ❒ 采用了 TrustZone 安全技术（v6Z 版）。在处理器硬件层面划分了可信区域和不可信区域，两个区域里运行的代码有不同的权限，经认证的代码运行在可信区域，未经认证的代码运行在不可信区域，从而提高了系统的安全性。

基于 v6 架构的 ARM 处理器主要是 ARM11 产品系列。在微架构实现方面，ARM11 系列处理器将流水线级数增加到 8 级或 9 级（v6T2），并采用了动态和静态相结合的转移预测方式，预测准确率可以达到 85%。ARM11 内核与 Cache 之间，以及内核与协处理器之间的数据通路扩展到 64 位，每个流水线周期可以读入两条指令或存取两个连续的字数据，成倍提高了取指和数据访问的速度。ARM11 内部还增加了增强型的数字信号处理器以及用于功耗管理的 IEM（Intelligent Energy Management）部件，在性能和功耗两个方面又有新的突破。

v6 版本架构因其所具有的优良特性从而备受青睐。2007 年，已是 v6 下一代版本 v7 面世的 3 年之后，ARM 公司仍然继续推出了 v6 版本的增强版 v6-M，并先后发布了三款基于该版本的 Cortex-M 系列处理器，均属于低成本、高性能、超小体积和超低功耗产品。关于 Cortex-M 处理器的主要特性将在下一小节中进一步介绍。

7) v7 版架构

ARMv7 版架构诞生于 2004 年，该版本可以看作 ARM 体系结构的一道分水岭，而在此之前的 ARM 处理器被 ARM 公司称为传统或者经典（Classical）处理器。与经典 ARM 处理器相比，ARMv7 版架构在处理器操作状态、工作模式、寄存器结构、存储器组织和映射、异常和中断管理等方面做了较大的调整和改进。同时，ARMv7 版架构也注意与经典 ARM 处理器软件保持一定的兼容性。

ARMv7 版架构全面支持 Thumb-2 技术和 Thumb-2 指令集。新的 Thumb-2 指令集比原来的 32 位 ARM 指令集减少了 31％的内存使用量，其性能比原来的 Thumb 指令集提升了 38％，并且无需在 ARM 指令集和 Thumb 指令集之间来回切换。此外，为了满足日益增长的 3D 图形与手机游戏的应用需求，v7 架构支持 128 位（16 字节）的 SIMD 扩展，可以选配名为 NEON 的多媒体数据处理引擎（DSP+SIMD），将 DSP 和媒体处理能力提高了近 4 倍。ARMv7 还支持改良的运行环境，以迎合即时编译（Just In Time，JIT）以及自适应动态编译（Dynamic Adaptive Compilation，DAC）技术的需求。

从 ARMv7 版本架构开始，ARM 将体系结构划分为三种类型（architecture profile[3]），分别是类型 A（A-profile，Application-profile）、类型 R（R-profile，Real Time-profile）和类型 M（M-profile，Microcontroller-profile）。ARMv7 版架构相应地也分为三类，分别是 ARMv7-A、ARMv7-R 和 ARMv7-M。同时，ARM 公司采用了新的处理器产品命名方式，摒弃了以往复杂的并且需要解析的命名规则，将基于三种体系结构类型的处理器产品划分为三条产品线或者三个产品系列，分别冠名为 Cortex-A、Cortex-R 和 Cortex-M，以期改变之前体系结构版本与产品名称之间较为混乱的对应关系。本书 1.4.5 小节已对 Cortex-A、Cortex-R 和 Cortex-M 的适

[3] “architecture profile”可以理解为体系结构的某个“剖面”，从某个角度看体系结构所呈现出的“剪影”，或者匹配某类应用的体系结构“类型”。本书将“architecture profile”统一称为体系结构类型。

用领域做过介绍，此处不再赘述。

从 ARMv4T 版架构到 ARMv7 版架构，ARM 指令集体系结构的演进过程可以用图 5.3 来表示。

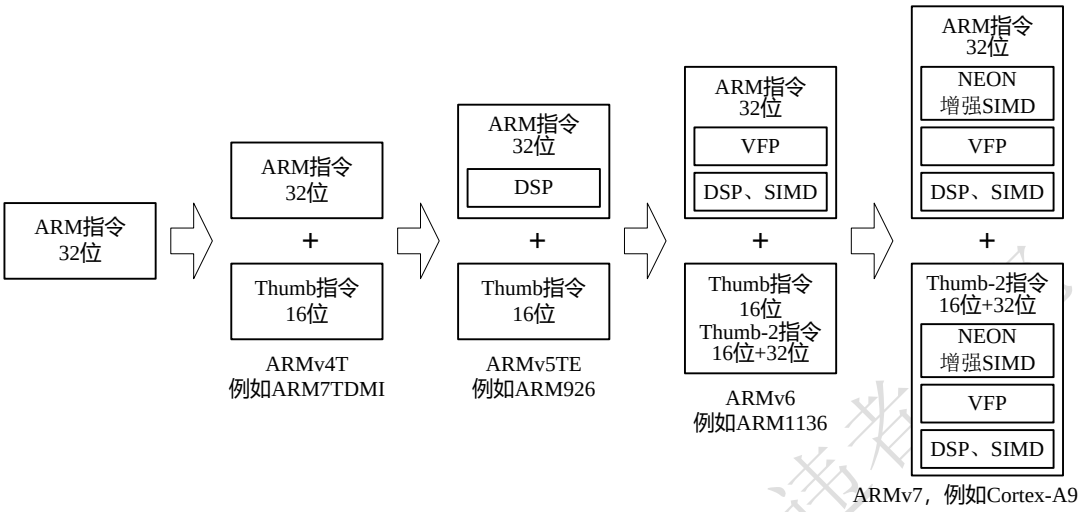

图 5.3 ARM 指令集体系结构（ISA）的演进过程

本书在 5.2 节之后，将详细介绍基于 ARMv7-M 版本架构的 ARM Cortex-M3 和 ARM Cortex-M4 处理器，并将其与经典处理器进行对比，以便读者能够更深入地了解 v7-M 版的新特性。

8) v8 版架构

ARMv8 是 ARM 公司首款支持 64 位指令集的 ISA 架构。ARMv8 版架构新增了一套 64 位指令集，称作 A64，并且继续支持原有的 ARM 指令集和 Thumb-2 指令集，同时将其重新命名为 A32 指令集和 T32 指令集。v8 版架构新定义了 AArch64 和 AArch32 两种运行状态，分别执行 64 位和 32 位指令集。ARMv8 的新特性包括：

- 在 ARMv7 安全扩展的基础上，新增加了安全模式，支持与安全相关的应用需求；
- 在 ARMv7 虚拟化扩展的基础上，提供完整的虚拟化框架，在硬件层面上提供对虚拟化的支持；
- AArch64 对异常等级赋予新的内涵，并重新解释了处理器运行模式和特权等级概念。

ARMv8 继续沿用 v7 版本所采用的体系结构类型划分方法，ARMv8 版本也分为 ARMv8-A、ARMv8-R 和 ARMv8-M 三种类型，他们各自的主要特性如下：

- ARMv8-A 架构是类型 A 中最新一代 ARM 架构，支持 64 位的 AArch64 和 32 位的 AArch32 两种运行状态，软件可以根据需要在两种运行状态之间进行切换。除了智能手机和平板电脑等传统高性能嵌入式应用以外，ARMv8-A 架构的目标市场也开始觊觎高端服务器市场。在这一领域内，ARM 公司的合作伙伴有中国的华为和飞腾等，以及国外的 AMD、三星、高通、英伟达和博通等传统半导体公司，甚至还包括谷歌、亚马逊和脸书在内的互联网公司。
- ARMv8-R 架构是类型 R 中最新一代 ARM 架构，该架构具有快速中断响应能力和确定的中断响应时延，采用容错设计并配置了内存保护单元 MPU，支持 A32 和 T32 指令集。Armv8-R 架构的应用领域包括无人机、车辆自动驾驶、医疗设备、各种专业机器人以及 4G/5G 应用。这些应用对实时性和可靠性都有着极高的要求。
- ARMv8-M 架构是类型 M 中最新一代 ARM 架构，使用了与其他类型不同的异常处

理模型，并且只支持 T32 指令集。ARMv8-M 架构面向的是低成本、小体积、低功耗、低中断延迟以及高性能的嵌入式应用。

ARMv8 版本架构诞生之后，ARM 公司对基于上述三种体系结构类型的处理器产品，仍然按照 Cortex-A、Cortex-R 和 Cortex-M 三个系列进行划分。

3. ARM 体系结构的增强型版本

为了满足不同类型的应用需求，ARM 公司在多个体系结构版本的基础上还定义了若干增强型版本。同时，在微架构方面也增加了或者可以选配相应的功能部件，例如 Cache、TSM、DSP、VFP、Java 加速器、多媒体处理单元以及各种跟踪调试部件等，形成了多种具有不同增强功能的处理器产品。在 ARMv7 版本架构之前，ARM 公司在版本名称后面增加若干英文字母后缀，以区分不同的增强型版本，例如：

1） T 增强版本

表示该版本支持 Thumb 指令集。

2） E 增强版本

支持增强的 DSP 算法。增加的主要指令包括：

- ❒ 增加了几条用于 16 位数据乘法运算和乘加运算指令；
- ❒ 增加有符号数的加减饱和运算指令；
- ❒ 增加了双字数据操作指令，包括双字加载指令 LDRD，双字存储指令 STRD 和协处理器寄存器传输指令 MCRR/MRRC；
- ❒ 增加了 Cache 预取（PLD）指令。

3） J 增强版本

支持基于 Jazelle　DBX 的 Java 硬件加速技术。与普通的 Java 虚拟机相比，Jazelle 技术可以使基于 Java 字节码的程序运行速度提高 8 倍，而功耗却可降低 80%。

4） S 增强版本

提供用于多媒体信号处理的 SIMD 指令。

5） F 增强版本

支持矢量浮点处理单元。

6） T2 增强版本

支持 Thumb-2 指令集

此外，在 ARMv6 版本架构中还有 Z、K 和 M 三个增强版本。其中 Z 表示支持 TrustZone 安全增强，K 表示支持多核（最多 4 个），而 ARMv6-M 则是用于 Cortex-M0、Cortex-M0+和 Cortex-M1 的增强版本。

在 ARMv7 版本之后，ARM 公司按照产品的应用领域，将体系结构分为 A、R 和 M 三种类型，以上用来表示版本增强特性的后缀基本上不再使用。

5.1.3 ARM 处理器主要产品系列简介

1. ARM 处理器的特点

本书第二章已经简要介绍了 ARM 处理器以及 ARM 处理器指令的主要特点，事实上，ARM 处理器以及 ARM 处理器指令还具有以下特色：

- 统一和固定长度的操作码，可以简化指令译码操作，便于指令流水线设计和实现；
- 具有多个通用寄存器，指令不局限在某个特定的寄存器上执行，可以实现对寄存器组的均匀访问；
- 操作数地址均由寄存器内容或者指令码的位域指定，具有地址自动增减寻址模式，寻址方式灵活，可以优化程序循环结构，程序执行效率高；
- 每条数据处理指令都可对算术逻辑单元和移位器进行控制，可最大程序地发挥 ALU 和移位器的效能；
- 具有多寄存器加载和存储指令，可增加处理器的数据吞吐速率；
- 所有指令都可以条件执行，以提升代码执行速度和执行效率；
- 支持精巧的 Thumb 指令集（包括 Thumb 和 Thumb-2），提高代码密度，减少所需的系统存储容量；
- 基于不同体系结构版本的 ARM 处理器，所支持的指令集在功能方面有所不同。但是只要基于同一种系统结构，即使在微架构和实现方式上有区别，所集成的功能部件存在差异，产品型号不同甚至属于不同系列产品，都可以做到软件相互兼容。

随着 ARM 体系结构版本的不断进步，ARM 公司面向不同的应用领域，推出了各具特色的系列处理器产品，分别满足用户在性能、成本、功耗、实时性、安全性、容错性、可综合、便于调试以及 DSP、多媒体处理和人工智能等方面的需求，ARM 处理器展现出越来越多的特点，同时 ARM 处理器指令功能也日渐丰富，本书后续内容将对此做进一步介绍。

2. ARM 处理器相关产品的层次关系

本书 2.7 节已经介绍了 ARM 内核、ARM 处理器以及基于 ARM 处理器的 MCU 或者 SOC 芯片等概念，事实上，基于 ARM 处理器的相关产品涵盖了从简单到复杂，从基本到综合，从相对单一的内核到多种部件集成的 SOC 芯片，呈现一种逐级扩展的层次结构。在 v7 版本以后，基于 ARM 内核的嵌入式系统一般结构如图 5.4 所示，图中虚线框表示的部件在某些版本架构中可以作为可选件，在另外一些版本架构中可能不支持，而外设以及外设接口均由芯片制造商根据需要进行选配。

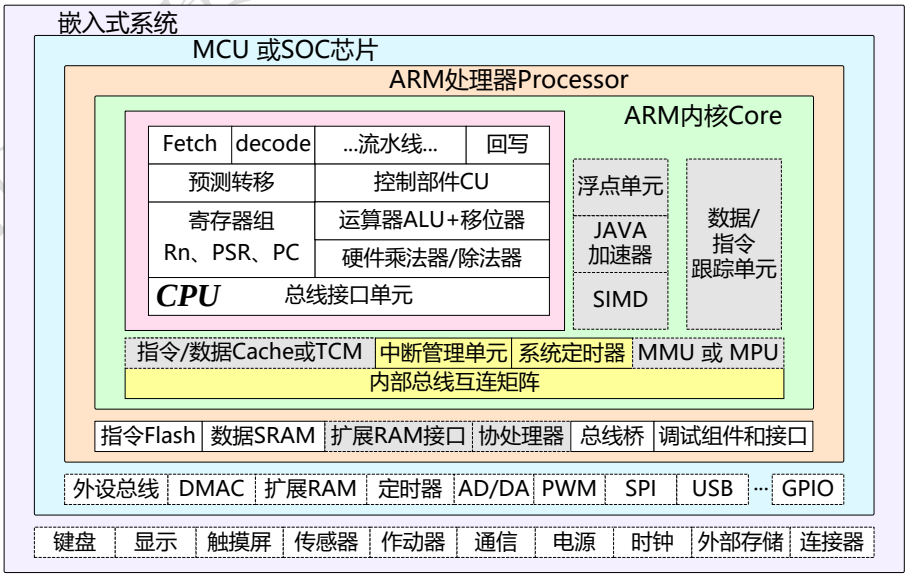

图 5.4 嵌入式系统层次结构

从图 5.4 中可以看出，在 ARM 内核（Core）中，除了 CPU 以外，必不可少的核心部件包

括中断管理部件、系统定时器和内部总线互连矩阵；其中总线互连矩阵可以让指令和多种数据在不同的总线上同时传送。有些内核可以选配指令和数据 Cache 或 TCM、MMU 或者 MPU、浮点单元、Java 硬件加速器、多媒体处理部件以及指令和数据跟踪单元调试组件。

ARM 处理器（Processor）是以 ARM 内核为核心，通过总线将程序存储器、数据存储器、协处理器、外设总线桥接器、各种调试组件和调试访问接口等部件在片内进行互连，构成了功能相对完整的微处理器。如果需要进一步扩展存储，ARM 处理器可以通过扩展 RAM 接口进一步扩展存储容量。

获得 ARM 授权的芯片制造商在 ARM 处理器的基础上，再集成诸如 DMA 传送控制器、ADC 和 DAC、定时器/计数器、脉宽调制器以及多种多样的接口电路，形成了各具特色的 MCU 或者 SOC 芯片。嵌入式系统开发商或者用户利用这些芯片，再配置所需人机接口设备、传感器、作动器、通信接口和电源等部件，就构成了具有特定功能的专用计算机系统，即嵌入式应用系统。

图 5.4 中的各种功能部件是通过不同的总线进行互连的。在 ARM 内核中，有指令和数据可以同时并行传送的总线互连矩阵；在 ARM 处理器中，有高速的系统总线和相对低速的外设总线，系统总线和外设总线之间有总线桥接器，两者之间的关系以及各自所连接的功能部件如图 5.5 所示。

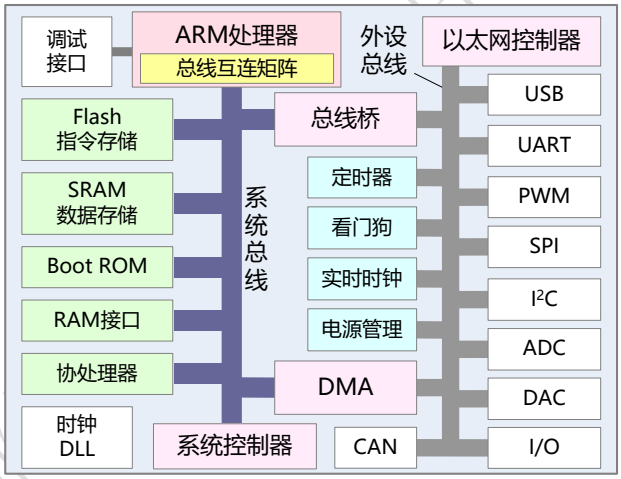

图 5.5 基于 ARM 处理器的 MCU/SOC 一般结构

从图 5.4 和图 5.5 中还可以看出，在基于 ARM 体系结构的嵌入式系统中，核心是 ARM 内核或者 ARM 处理器。事实上，ARM 内核与 ARM 处理器都是由 ARM 公司设计的，两者的区别只是所包含的功能部件有所不同，彼此之间并没有明显的界限。因此，除非有特别说明，本书对 ARM 内核与 ARM 处理器也不进行严格区分。

3. ARM 处理器产品命名规则

ARM 公司所发布的 ARM 体系结构（ISA）版本号与 ARM 内核或者 ARM 处理器名称采用了两种不同的标识方式。自 v3 版本之后，基于同一个架构版本有多个 ARM 处理器产品。经典 ARM 处理器的名称与体系结构版本编号之间的对应关系较为复杂，在 ARMv7 版以后，这种局面有所改观。目前仍在使用的 ARM 处理器产品家族与 ARM 体系架构版本之间的对应关系如图 5.6 所示。

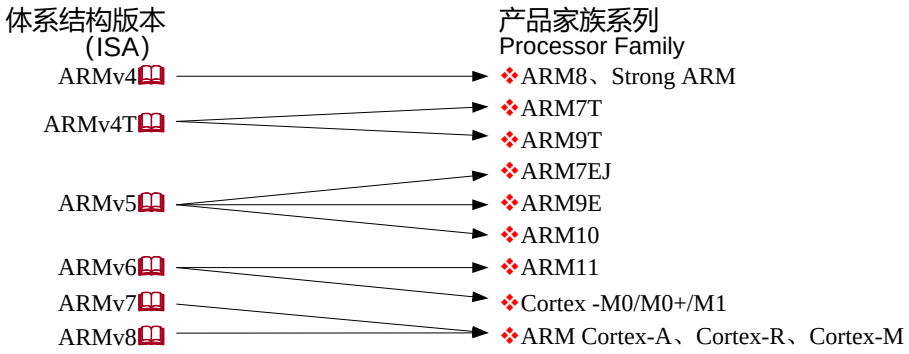

图 5.6 ARM 体系结构版本与 ARM 处理器产品家族之间的对应关系

在 ARMv7 版架构发布之前，经典 ARM 处理器产品的命名规则如下：

1) 处理器名称后缀中的英文字母含义

T、E、J、S、F、T2、Z 和 K 等字母表示所基于体系结构增强版本。在基于 ARMv4T 的 ARM7TDMI 以及 ARM9TDMI 产品名称中，还出现过 D、M 和 I 三个字母后缀，分别表示带有 Debug 调试接口、硬件乘法器和嵌入式 ICE，后来这些都成了 ARM 处理器的标准配置，不再使用后缀进行标识。

2) 第一个（组）数字的含义

ARM 后面第一个（组）数字表示产品系列，如 ARM7TDMI 和 ARM920T 中的“7”和“9”分别对应 ARM7 系列和 ARM9 系列，但在 ARM10 和 ARM11 产品系列中，“10”和“11”两位数字被视为一组。

3) 第二个数字的含义

2 表示带有 MMU，如 ARM720T 和 ARM920T；

3 带有改良型的 MMU；

4 表示带有 MPU，如 ARM946ES；

6 表示无 MMU 和 MPU，如 ARM966E-S 和 ARM968EJ-S。

4) 第三个数字的含义

0 表示标准容量 Cache；

2 表示减小容量的 Cache。

6 表示带有紧耦合内存 TCM

5) 名称最后的“-S”

表示是可综合版本，亦即软核。

在 v7 版架构发布之后，很多增强型部件都成了 ARM 处理器的标准配置或者可选配置，同时 ARM 也开始使用新的基于体系结构类型的产品分类方法，ARM 处理器名称中除了体系结构类型和产品序号之外，不再使用表示处理器特性的复杂编号。新的命名方式将 Cortex 作为整体品牌名称，用 A、R 和 M 表示体系结构类型，同一系列处理器只使用序号命名，以消除架构版本和处理器编号不一致所引起的混乱。例如，Cortex-A8、Cortex-R5 和 Cortex-M3 是分别属于 A、R 和 M 三个体系架构类型的产品。但是，仍有个别产品名称带有字母后缀，表示其所适用的应用领域。

4. ARM7 系列

ARM7 系列分为 ARM7 和 ARM7T 两个子系列，前者基于 ARMv3 版本架构，后者是基于 ARMv4T 版本架构。真正获得广泛应用的是 ARM7T 产品系列。

ARM7T 系列的基本型产品是 ARM7TDMI 以及可综合版 ARM7TDMI-s，其内部结构参见本书2.7.3小节的介绍。ARM7T其他几款处理器都是以ARM7TDMI为核心，另外增加了Cache、MMU 或者 MPU、以及 ASB 总线接口等功能部件。其中 MMU 和 MPU 都是用于内存管理，两者的功能和主要区别简介如下。

在执行多任务的计算机系统中，必须提供一种机制来保证正在运行的任务不破坏其他任务的操作。在嵌入式系统中，MPU 就是实现这种机制的一种专门硬件部件，MPU 将内存空间划分成若干个域（regions）进行管理，域具有与存储空间一一对应的属性。MPU 将域的大小、起始地址和访问权限等存放在内部寄存器中。域的访问权限有可读/可写、只读和不可访问等。当程序代码试图访问存储器的一个域时，MPU 将该程序所具有的特权等级与该域的访问权限属性进行比较，如果符合域的访问标准，则 MPU 允许该程序进行访问；如果不符号域的访问标准，将产生一个异常。

MMU 在内存保护方面的功能以及原理与 MPU 基本相同，也是将内存划分为若干个域并设置了不同的访问权限属性。但除了提供内存保护功能以外，MMU 的主要使命是负责虚拟地址和物理地址之间的转换。在多任务计算机系统中，往往使用虚拟地址来编写大型程序。当 MMU 被使能之后，MMU 拦截处理器输出的虚拟地址，将其分解为页号和页内偏移地址，再通过查找存放在主存中的二级页表，生成访问内存所需的物理地址并输出到地址总线上（这部分内容参见本书第三章有关内容）。为了提高虚实地址的转换效率，MMU 中通常会设置一个转换旁视缓冲器，（Translation Lookaside Buffer），其作用类似于高速缓存，保存最近被访问过的页表项。如果访问的页不在主存中，MMU 将产生一个异常，请求将所需的页从辅存中调往内存。假如 MMU 没有使能（被关闭），处理器输出的地址将作为物理地址，直接送往地址总线。

MMU 对内存采用分页管理，减少了内存碎片，提高了内存的利用率。但是，虚拟地址到物理地址的转换过程可能会产生不可预期的延迟。因此，尽管使用虚拟地址有诸多优点，但并不适合某些对实时性有较高要求的应用系统。所以，在面向实时应用的 ARM 处理器中往往配置的是 MPU 而不是 MMU。不过，即便在实时系统中使用了带有 MMU 的处理器，也可以选择关闭 MMU 的功能。

常见的 ARM7T 系列处理器的主要特性见表 5.1。ARM7T 系列的工作时钟频率小于 200 MHz，Dhrystone 基准测试结果为 0.9 DMIPS/MHz。采用 ARM7T 系列处理器为核心的 MCU/SOC 芯片主要有恩智浦（NXP）公司的 LPC20 系列、三星（Samsung）公司的 S3C44BO 系列以及意法（ST）半导体的 STR7 系列等。

5. ARM9 系列处理器

ARM9 系列也分为 ARM9T 和 ARM9E 两个子系列，前者与 ARM7T 相同，也是基于 ARMv4T 版本架构，但在微架构方面有较多改进。ARM9T 采用的是哈佛结构，流水线为 5 级。除了本书 2.7.3 简介的 ARM9TDMI 和 ARM920T 以外，ARM9T 系列中还有 ARM922T 和 ARM940T，这两款与 ARM920T 基本相同，只是 Cache 的大小有区别。

ARM9E 系列是基于 ARMv5TE 和 ARMv5TEJ 版本架构，根据上一小节关于 ARM 体系结

构版本的的介绍，通过版本名称的后缀不难看出它们之间的差别。ARM9E 系列处理器采用的也是哈佛结构，基本型产品为 ARM9E-S 和 ARM9EJ-S，前者带有 DSP 和可选的 VFP，后者还增加了 Jazelle，但是作为单个产品，ARM9E-S 和 ARM9EJ-S 现在均已退出使用。目前仍在使用的 ARM9E 系列产品有 ARM926EJ-S、ARM946E-S 和 ARM968E-S，分别适用于无线设备、数字消费品、成像设备、工业控制装置、存储系统控制器和网络设备等对实时性有较高要求的应用场合。

ARM926EJ-S 将 ARM9EJ-S 作为内核，另外增加了大小可选配的指令 Cache 和数据 Cache，并带有 TCM 接口、Jazelle DBX、DSP、ETM、AHB 总线接口以及外部协处理器接口等功能部件。ARM926EJ-S 配置了 MMU，可使用虚拟地址编程，支持多任务操作系统。

ARM946E-S 的内核是 ARM9E-S。与 ARM926EJ-S 的配置相比，也带有大小可选的指令 Cache 和数据 Cache，并且处理器内部还集成了 2 个 SRAM，分别作为指令 TCM 和数据 TCM，SRAM 的容量从 1KB ~ 1MB 可选。ARM946E-S 带有 DSP 部件，但是没有 Jazelle DBX。ARM946E-S 没有使用 MMU，只有 MPU，因此只具有内存保护功能，不支持虚拟地址，应用领域多为实时任务系统。

ARM968E-S 的内核也是 ARM9E-S，但是没有 Cache 和片内 TCM，只配置了指令和数据 TCM 接口，TCM 的容量从 1KB 到 4MB（按照 2^n 配置）可选。ARM968E-S 既没有 MMU 也没有 MPU 和 AHB，总线也简配为 AHB Lite（只能支持一个 master 主设备），并将 ETM 作为选件，应属于 ARM9E 系列中最低端的产品。此外，还有一款已经不再使用的产品 ARM966E-S，与 ARM968E-S 极为类似，并且配置较 ARM968E-S 还略强一些。两者的主要区别是 ARM966E-S 的总线接口仍然是 AHB，TCM 的容量范围为 1KB~64MB。

常见的 ARM9 系列处理器的主要特性见表 5.1。ARM9 系列的典型工作时钟频率为 400HHz，Dhrystone 基准测试结果均为 1.1 DMIPS/MHz。采用 ARM9 系列处理器为核心的 MCU/SOC 芯片主要有三星公司的 S3C24 系列、飞思卡尔（Freescale，2015 年与恩智浦合并）公司的 i.MX27 系列、TI（德州仪器）公司的 OMAP 1 系列、以及恩智浦公司的 LPC3200（ARM926EJ）和 LPC2900（ARM968E-S）系列等。

6. ARM11 系列处理器

ARM11 系列处理器基于 ARMv6 版本架构，该系列产品都是软核。共有 ARM1136J(F)-S、ARM1156T2(F)-S、ARM1176JZ(F)-S 和 ARM11MPCore 四款产品。产品名称中的“(F)”表示可选配矢量浮点单元 VFP。通过上一小节关于 ARMv6 版本架构的介绍，可以大致了解这几款 ARM11 系列处理器所具有的主要特点，例如：带有转移预测功能的 8 级或 9 级流水线、内核与 Cache 之间以及内核与协处理器之间具有 64 位数据通路、带有增强型数字信号处理器、支持混合大小端和非对准存储访问等。但是每款处理器又具有一些个性化的功能，这能从产品名称中得到一些反映。几款处理器的主要特性简介如下：

- ARM1136J(F)-S，基于 ARMv6 版本，主要特性有 SIMD、Thumb、Jazelle、MMU 以及可选配的 VFP。
- ARM1156T2(F)-S，基于 ARMv6T2 版本，主要特性有：Thumb-2 指令集、SIMD、可选配的 VFP，没有 MMU，只有可选配的 MPU。
- ARM1176JZ(F)-S，基于 ARMv6Z 版本，在 ARM1136J(F)-S 基础上增加了 TrustZone 安全功能和 IEM 部件。
- ARM11MPCore，可以配置 1~4 颗 ARM1136J(F)-S 内核的全对称多核处理器。

ARM11 处理器的流水线仍然属于单发射的标量流水线，但在流水线的后几级，算逻单元 ALU、乘积累加单元（Multiply Accumulate，MAC）和数据加载/存储单元（LSU）采用了并行部署方式，属于一种分叉或者分支结构的流水线。ARM1136J(F)-S 和 ARM1176JZ(F)-S 的流水线结构如图 5.7 所示。ARM1156T2(F)-S 的流水线增加了一级 Thumb-2 指令取指，其结构如图 5.8 所示。

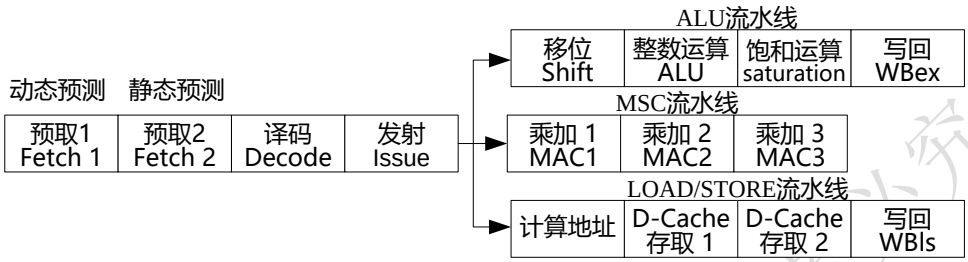

图 5.7 ARM1136J(F)-S 和 ARM1176JZ(F)-S 流水线结构[4]

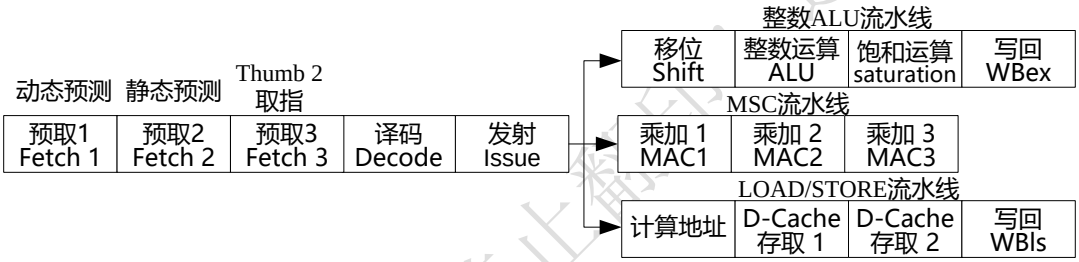

图 5.8 ARM1156T2(F)-S 流水线结构

在上述分叉结构的流水线中，当指令完成译码和取操作数之后，不同类型的任务被分发到不同的流水线分支上执行。由于分支流水线上各个功能部件的分工更加具体，电路更加简单，执行速度更快，所需的流水线周期更短，从而可以提高整条流水线的时钟频率。ARM11 的工作时钟频率可以提高到 500MHZ 以上，最高可达到 1GHz（取决于芯片所采用的工艺制程）。

通过以上新技术的应用，ARM11 系列处理器的性能较上一代产品有了较大幅度的提高。在时钟频率为 772MHz 时，Dhrystone 基准测试指标为 965 DMIPS。配置 4 颗 ARM1136J(F)-S 内核的 ARM11MPCore，可以达到 2600 DMIPS。

ARM 处理器的设计理念并不是单纯追求高性能，而是要在性能、功耗、芯片面积和成本之间进行均衡。ARM11 可以通过软件动态调整时钟频率和工作电压，实现性能和功耗之间的平衡。采用 130nm 工艺制作的 ARM11，当电源电压为 1.2v 时，功耗仅为 0.4mW/MHz。除了正常的工作模式以外，配置了 IEM 的 ARM1176JZ(F)-S 还支持休眠模式、待机模式和关机模式，进一步降低了系统功耗。

将 ARM11 系列处理器作为核心的 MCU/SOC 芯片主要有三星的 S3C64 系列、飞思卡尔公司的 i.MX31 系列、TI 公司的 OMAP2 系列，以及高通骁龙 S1 系列手机处理器中的部分产品。

常见的基于 ARMv1~ARMv6（不包括 ARMv6-M）版本架构的 ARM 处理器产品名称、体

[4] WBex（Write back of data from the multiply or main execution pipelines），倍乘或执行流水线数据写回；WBls （Write back of data from the Load Store Unit），加载/存储单元数据回写。

系结构版本和主要特性如表 5.1[5]所示。

[5] 每款 ARM 处理器还有若干个小版本，以 ARM1136J(F)-S 为例，从 2002 年 12 月到 2009 年 2 月，先后有 r0p0，r0p1、r0p2、r1p1、r1p3 和 r1p5 等多个小版本，每个小版本的细节都有所不同，ARM 公司的产品手册中都会对小版本之间差异进行说明。

表 5.1 常见的经典 ARM 处理器产品特性一览表

产品家族	结构版本	内核或处理器型号	主要特性	Cache Or TCM	典型性能 MIPS@MHz
ARM1	ARMv1	ARM1	原型机，只有简单的基本指令，26 位地址	-	
ARM2	ARMv2	ARM2	增加了乘法指令	-	0.5, 0.33D
	ARMv2a	ARM250	MMU、图形及 I/O 处理器、SWP/SWPB 指令		0.58
ARM3		ARM3	首次增加了 Cache	4K 统一	0.48, 0.5D
ARM6	ARMv3	ARM60	32 位地址、长乘法和长乘加指令	-	0.83
		ARM600	Cache、协处理器（浮点单元）接口	4K 统一	0.85
		ARM610	同 ARM60，Cache，无协处理器接口	4K 统一	0.85, 0.65D
ARM7		ARM700		8K 统一	40MHz
		ARM710	同 ARM700，协处理器接口	8K 统一	40MHz
		ARM710a	同 ARM70	8K 统一	40MHz, 0.68D
ARM7T	ARMv4T	ARM7 TDMI(-S)	3 级流水线、Thumb 指令集	-	≤200MHz 0.89D
		ARM710T	同 ARM7TDMI，Cache、MMU	8K 统一	0.9D
		ARM720T	同 ARM710T，ASB（r4 以后 AHB）	8K 统一	1D
		ARM740T	同 ARM7TDMI，MPU	-	
ARM7EJ	ARMv5TEJ	ARM7EJ-S	5 级流水线、Thumb、Jazelle、DSP	-	
ARM8	ARMv4	ARM810	5 级流水线、静态转移预测、MMU	8K 统一	1.17D
ARM9T	ARMv4T	ARM9 TDMI(-S)	5 级流水线、哈佛结构、Thumb	-	典型时钟频率 400MHz
		ARM920T	同 ARM9TDMI、Cache、MMU、ASB	16K+16K	1.11D
		ARM922T	同 ARM9TDMI、Cache、MMU、ASB	8K+8K	
		ARM940T	同 ARM9TDMI、Cache、MPU、ASB	4K+4K	
ARM9E	ARMv5TE	ARM946E-S	Thumb、DSP、容量可变 Cache、MPU、AHB	I/D TCM	
		ARM966E-S	Thumb、DSP、无 Cache（停止使用）	I/D TCM	
		ARM968E-S	DMA、比 ARM966E-S 尺寸小 20%，功耗小	I/D TCM	
		ARM996HS	Clockless 处理器，同 ARM966EJ-S	I/D TCM	
	ARMv5TEJ	ARM926EJ-S	Thumb、DSP、Jazelle、VFP、MMU、I/D TCM 接口、协处理器接口	4~128K I/D Cache	1.1
ARM10E		ARM1026EJ-S	DSP、Jazelle、VFP、MMU or MPU	容量可变	
	ARMv5TE	ARM1020E	6 级流水线、Thumb、DSP、VFP、MMU	32K+32K	
		ARM1022E	同 ARM1020E	16K+16K	
ARM11	ARMv6	ARM1136 J(F)-S	8 级流水线、SIMD、Thumb、Jazelle DBX、DSP、	TCM+ Cache	400M~1GHZ ≥ 1.3
	ARMv6T2	ARM1156 T2(F)-S	9 级流水线、SIMD、Thumb-2、Jazelle DBX、DSP、可选 VFP，可选 MPU	TCM+可选 Cache	
	ARMv6Z	ARM1176 JZ(F)-S	同 ARM1136J(F)-S，TrustZone	TCM+ Cache	
	ARMv6K	ARM11MPcore	同 ARM1136J(F)-S，1~4 核	容量可变	2600D@4 核

7. Cortex-A

Cortex-A 是面向移动计算、智能手机、数字电视、企业网络和服务器的高性能处理器。Cortex-A 支持可伸缩（Scalable）的异构或者同构多核处理器架构，内置 VFP 和 NEON 以支持浮点运算和 SIMD 多媒体数据处理，时钟频率超过 1 GHz，支持 Linux、安卓和微软视窗等大型操作系统。Cortex-A 系列的首个处理器是 Cortex-A8，发布于 2005 年 10 月。截止到 2020 年 2 月，ARM 公司 Cortex-A 产品系列已有 A5~A77 共 20 款处理器。如表 5.2 所示。

表 5.2 Cortex-A 系列处理器型号以及体系结构版本

体系结构版本	处理器型号	支持指令集
ARMv7-A	A5、A7、A8、A9、A15、A17	A32 和 T32
ARMv8-A	A32	A32 和 T32
ARMv8-A	A34、A35、A53、A55、A58、A65、A72、A73、A75、A76、A77	A64、A32 和 T32
ARMv8-A	A65AE、A76AE（AE 后缀表示汽车增强型）	A64、A32 和 T32

其中 A5 到 A17 基于 32 位 ARMv7-A 体系结构类型；A32 虽然基于 ARMv8-A 体系结构类型，但只有 AArch32 运行状态；A34 以后的产品都基于 64 位 ARMv8-A 体系结构类型，同时支持 AArch32 和 AArch64 两种运行模式，其中还有 A65AE 和 A75AE 两款是专门用于车辆控制的汽车增强型（Automotive Enhanced）产品。

上述处理器除了体系结构版本类型不同之外，其内部微架构和所配置的功能部件也有所不同。2011 年，ARM 公司推出了 Cortex-A 处理器的“bL”（big.LITTLE，大小核）技术，允许处理器搭载两种不同类型的内核。在采用 bL 技术的处理器中，内核的性能有高有低，功耗有大有小，轻负载时运行低功耗内核，高负载情况下运行高性能内核，以期在性能和节能之间寻求平衡。现代智能手机处理器 98%以上都采用了 bL 架构，以期同时兼顾手机性能和电池的续航时间。2017 年，ARM 公司将 bL 技术进行升级，并命名为 DynamIQ 架构，该架构允许最多 8 个内核构成一个簇（cluster），单个处理器最多可搭载 32 个簇以及 256 个内核，进一步提高了大小核架构的灵活性与可扩展性。

2017 年面世的 A55 是 ARM 公司推出的第一款基于 DynamIQ 技术的小核。A55 采用 ARMv8.2 架构，引入神经网络算法以提高转移预测的准确性，并对 NEON、VFP 和缓存结构做了改进，同时也进一步降低了功率消耗。A55 可以作为一款高性能低功耗的 64 位处理器单独使用，应用领域包括数字电视、虚拟现实（Virtual Reality，VR）和增强现实（Augmented Reality，AR）等。但是，A55 设计目标更多地是作为 DynamIQ 结构中的小核，例如，在华为麒麟 980/990、高通骁龙 865/855、三星 Exynos98 系列以及联发科 Helio G 系列等目前较高端的手机处理器中，有些虽然采用了自研的大核，但小核几乎毫无例外地都选用了 A55。

A76 和 A77 是 ARM 公司分别于 2018 年和 2019 年发布的两款基于 DynamIQ 技术的“大核”，分别刷新了当时 Cortex-A 系列处理器的最高性能记录。A76 和 A77 不仅可以用于通用计算，还可以担负大部分设备的机器学习、虚拟现实和增强现实任务。

A76 采用 ARMv8.2 版本架构，在微架构实现方面使用了可以乱序执行的超标量结构，拥有 4 个解码发射单元和 8 条并行流水线，流水线级数为 13 级。采用 7nm 制程工艺时，A76 的峰值时钟频率可达 3GHz。据 ARM 公司公布的基准测试数据，A76 的性能已经超过 Intel 酷睿 i5-7300U，是名副其实的“大核”。使用 A76 作为大核的处理器，可以通过 DynamIQ 技术实现多种大小核配置方案，例如“1 大+7 小”、“2 大+6 小”或者“4 大+4 小”，如图 5.9 所示。

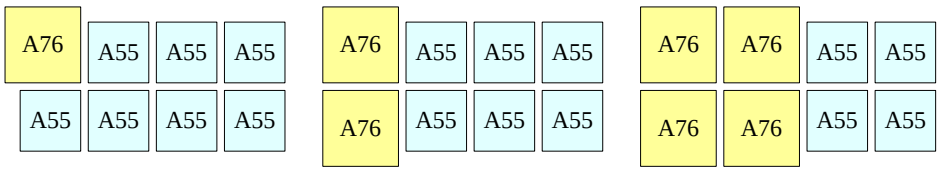

图 5.9 基于 DynamIQ 的大小核搭配方式

ARM 公司于 2019 年 5 月 27 日发布的 A77，仍然基于 ARMv8.2 版本架构。与 A76 相比，A77 的基本配置没有太大变化。但是，A77 的设计目标是增加 IPC，其改进几乎触及了微体系结构的所有部分，包括采用具有更大带宽的前端（取指+译码+转移预测+发射），使用新的指令缓存技术、新的整数 ALU 单元和改进的加载/存储队列，在相同制程工艺和时钟频率条件下，

A77 的 IPC 比 A76 提高了 20%，可以看作是 A76 的改良版。虽然 A77 的设计目标是希望通过 DynamIQ 技术与更多的同构或者异构内核进行集成，提供更强大的计算能力和实现更高效的能源利用效率。但是 2019 年 9 月 6 日，在华为发布的海思麒麟 990 5G 智能手机处理器中，所使用的大核仍然是 A76，并没有选用 A77。对此华为的官方解释是 A77 的性能已经超出了当时手机处理器的性能需求。

在通用计算领域，基于 ARM 架构和 Cortex-A 内核的高性能处理器也开始崭露头角。例如，采用 ARM 架构的国产飞腾系列和鲲鹏系列处理器已开始装备国产通用台式计算机和服务器，为实现我国信息化安全自主可控的目标迈出了坚定的一步。相信未来随着软件生态的不断改善，这类产品必定会得到广泛应用。

8. Cortex-R

Cortex-R 系列聚焦于高性能实时应用，所谓实时性是指严格的时间确定性。Cortex-R 的应用领域包括汽车电子、医疗设备、工业控制装置、智能手机的基带调制解调器、硬盘驱动器和企业级网络设备等。这些应用对处理截止时间都有着极其严格的要求，必须在规定的时间内完成规定的操作。此外，在汽车电子、医疗设备和重要的工业过程控制装置中，对安全稳健（Safety-Critical）也有极高的要求，绝对不允许因为处理器或数据错误而出现误操作。Cortex-R 的设计目标包括低延时、高可靠、可信赖、安全性、高性能和容错性，以确保嵌入式系统的时间确定性和行为确定性。

如前所述，在具有实时性的计算机系统中，一般不采用虚拟地址，避免虚拟地址与物理地址转换过程可能出现的访问延迟。此外，这类系统中的异常/中断处理程序一般也存储在内核的私有存储器（如 TCM）中，避免在使用物理地址访问公用内存时与其他内核产生冲突。

Cortex-R 系列现有 5 款处理器，分别是基于 ARMv7-R 版本架构的 R4、R5、R7 和 R8，以及基于 ARMv8-R 版本架构的 R52。其中 R4 是 Cortex-R 系列的第一款产品，而 R52 虽然是基于 ARMv8-R 版本架构，但是只能运行在 AArch32 状态。所有 Cortex-R 系列处理器都支持 A32 和 T32 指令集，可以实现二进制代码兼容。

为了实现容错性，所有 Cortex-R 处理器都支持多处理器锁步（lock-step）技术。所谓锁步技术是一种利用硬件冗余设计以提高计算机系统的容错能力。如果使用两颗处理器进行锁步配置，两颗处理器分别作为主处理器和监控处理器，时间上严格同步并执行相同的指令。主处理器承担系统处理任务并负责驱动输出，监控处理器连续监测主处理器总线上的数据、地址和状态等信息，一旦发现两者之间出现不一致，说明某颗处理器出现了差错。由于仅有两颗处理器，无法判断究竟是哪颗处理器出现了问题，所以只能采用故障静默方式，本次计算结果不输出。如果采用 3 颗或 3 颗以上处理器进行锁步配置，在单颗处理器出现故障时，则可以通过硬件表决方式判断出现故障的处理器，并对其进行屏蔽，由其他处理器负责任务执行，从而实现故障的实时恢复。

在现有的 5 款 Cortex-R 系列处理器中，Cortex-R4 是单内核处理器，两颗 R4 可以通过锁步配置组成高可靠性的双核。R5 升级为一主一备的异构双内核，单颗处理器就可实现能够独立运行的锁步双核。R7 在异构双核中增加了 QoS[6]（服务质量保证）功能，另外又增加一对同构双核，以提高处理器的性能。R8 和 R52 在 R7 的基础上，将锁步配置的内核数量增加到 3 颗或者 4 颗，可以实现故障恢复。R8 和 R52 也增加了同构内核的数量。Cortex-R 系列各款处

[6] 这里的 QoS（Quality of Service）可以理解为将需要处理任务按照轻重缓急进行分类，确保高优先级的任务能够得到优先处理。

理器的特性和内核配置情况参见表 5.3。

表 5.3 Cortex-R 系列处理器特性对比表

型号 / 特性	R4	R5	R7	R8	R52
ISA	ARMv7-R	ARMv7-R	ARMv7-R	ARMv7-R	ARMv8-R
DMIPS	1.67 / 2.01 / 2.45 DMIPS/MHz		2.50 / 2.90 / 3.77		2.04 / 2.6 / 5.07
CoreMark	3.47		4.35	4.62	4.2
锁步	有				
内核	单内核	异构双核	异构双核+QoS 同构双核	2~4 异构多核+QoS 2~4 同构多核	
TCM	有				
LLPP	-	有			
处理器	双发射 8 级流水线、指令预取 转移预测		超标量 11 级流水线、乱序执行 静态+动态转移预测		双发射 8 级、指令预取、转移预
Cache	I-Cache 和 D-Cache				
	硬件除法器、 SIMD、DSP				硬件除法器 NEON
FPU	双精度	双精度或者优化的单精度			
MPU	8 或 12 区域	12 或 16 区域		12、16、20 或 24 区域	第一级&第二级 0 或 16 区域
ECC 和 奇偶校验	Cache 和/或 TCM ECC&Parity	Cache 和/或 TCM ECC&Parity， AXI 接口 ECC	Cache 和/或 TCM ECC&Parity， AXI 接口 ECC， 基于错误库的出错管理		Cache 和/或 TCM ECC&Parity， AXI 接口 ECC， 互连交叉开关 总线保护
VIC 或 GIC	矢量中断控制器 VIC 或 通用中断控制器 GIC		集成 GIC		集成 GIC 32~960 个中断源

同样是为了提高可靠性，Cortex-R 系列所有处理器的 Cache 或 TCM 都支持 ECC（Error Correcting Code）或奇偶校验（Parity），对于每次进出 Cache 或 TCM 的数据，可以进行 2bit 的检错和 1bit 的纠错。除了 R4 以外，R5~R52 还支持对总线接口的 ECC 纠错和检错，R52 则进一步增加了对内核交叉开关互连总线（Switch Fabric）的错误防护功能。

为了满足实时性要求，所有 5 款 Cortex-R 处理器都采用指令和数据分离的哈佛结构，均配置了指令和数据 Cache、TCM 存储器、硬件除法器、双精度 FPU、MPU、内存和中断控制器。R4~R8 处理器搭载了 SIMD 单元和 DSP，R52 处理器增加了 NEON 引擎以提高多媒体数据处理能力。为了减少访问 I/O 端口时间，R5~R52 还带有低时延的外设接口（Low Latency Peripheral Port，LLPP），可以实现快速的外设读取和写入。

Cortex-R4 的性能为 2.45 DMIPS/MHz，峰值时钟频率为 600MHz。R7 的性能为 3.77 DMIPS/MHz，峰值时钟频率可超过 1GHz。R8 的异构和同构内核数量比 R7 翻了一番，每个内核都能够非对称运行，各自都带有电源管理部件，可以单独关闭某个内核以减少功率消耗。另外，R8 的每个核心都扩充了 Cache 和 TCM 的容量，峰值时钟频率可超过 1.5 GHz，总体性能是 Cortex-R7 的两倍。

基于 ARMv8-R 架构的 R52 是 Cortex-R 系列最新一款微处理器，支持硬件虚拟化技术，可以看作 Cortex-R5 的升级版。但 R52 与 R7 和 R8 在应用领域上有区别，R52 面向汽车、工业自动化和医疗设备领域，R7 和 R8 在存储低延迟和调制解调器（Modem）方面进行了强化，目标市场是车联网、物联网以及 4G 和 5G 解决方案。

9. Cortex-M

Cortex-M 系列面向的是低成本、低功耗和高性能应用领域。该系列第一个产品是基于 ARMv7-M 版本架构的 Cortex-M3，诞生于 2005 年（2006 年开始有芯片出现）。为了降低功耗，Cortex-M3 的流水线只有 3 级，但带有硬件除法器，支持 Thumb-2 指令集，并提供了全面的调试和跟踪功能，是一款高性能低成本的 MCU。2010 年，ARM 公司在 M3 基础上，推出了可选配单精度浮点单元 FPU 的 Cortex-M4。除了可选配 FPU 以外，Cortex-M4 还增加了 SIMD、饱和运算以及快速乘加运算指令，使其可以执行一些数字信号处理程序。Cortex-M3 和 Cortex-M4 处理器的应用非常广泛，有多家芯片厂商生产了众多基于这两款处理器的 SOC 产品。本书后续部分将详细介绍 Cortex-M3 和 Cortex-M4（以下合称 Cortex-M3/M4）的主要组成、编程模型、存储器结构、中断处理过程、寻址方式、指令系统以及软硬件开发技术。

2007 年，在 ARMv7-M 版架构面世三年之后，ARM 公司没有舍弃 v6 版架构，又发布了支持 Thumb-2 指令集的 ARMv6-M 增强版。ARM 公司与 FPGA 器件的主要生产厂家赛灵思（Xilinx）公司合作，于 2007 年 3 月推出了面向 FPGA 设计的基于 v6-M 版本的 Cortex-M1 处理器。2009 年 2 月，ARM 公司又发布了一款同样是基于 v6-M 版架构的 Cortex-M0 处理器。为了降低功耗和减少系统复杂性，Cortex-M0 采用冯.诺依曼结构，流水线只有 3 级，配置了嵌套向量中断控制器（Nested Vectored Interrupt Controller，NVIC）、能源管理部件和 AHB Lite 总线接口，并可选配唤醒中断控制器（Wakeup Interrupt Controller，WIC）、单周期硬件乘法器、系统定时时钟 SysTick 和调试组件。Cortex-M0 的门电路数不到 12 000 个，如果采用 40 纳米低耗电（40LP）制程，布线面积仅有 0.008 mm^2，功耗仅为 5.3 μW/MHz @1.1v/25℃，性能却可达到 0.87~1.27 DMIPS/MHz，属于一种极低功耗的 32 位高性能处理器。M0 的内部结构如图 5.10 所示。

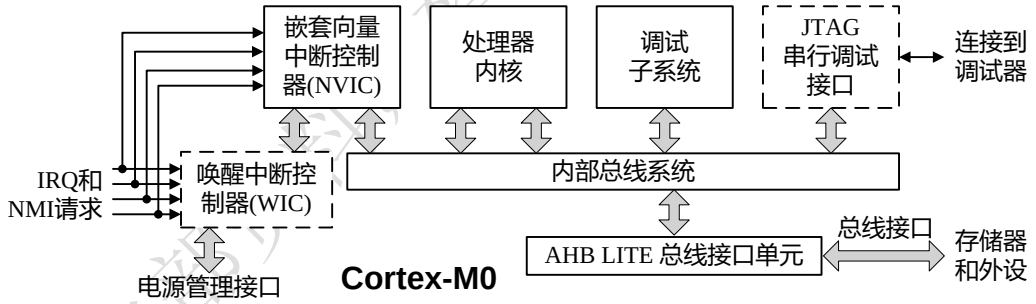

图 5.10 Cortex-M0 系统结构

2012 年 3 月，ARM 公司在中国上海发布了 M0 的改进版 ARM Cortex-M0+。M0+在 M0 的基础上增加了单周期 I/O 接口和微跟踪缓冲（Micro Trace Buffer MTB）特性。采用 40LP 制程的 M0+布线面积仅有 0.0066 mm^2，功耗控制在 3.8 μW/MHz @1.1v/25℃以内，性能却比 M0 提高 8%以上，把能效比推向一个新的高度。

2014 年 9 月问世的 Cortex-M7 则是在 Cortex-M4 的基础上，将流水线级数增加到 6 级，浮点单元升级为双精度，提供可选的 Cache 或者 TCM，以满足高端微控制器和密集型数据处理的需求。此后，ARM 公司又陆续推出了基于 ARMv8-M 版架构的 M23、M33 和 M35P，支持新的增强指令集并增加了 TrustZone 安全扩展。其中 M23 具有与 M0 类似的低成本和小体积特点；M33 与 M3 和 M4 类似，但系统设计更灵活，能效比更高；M35P 产品型号中的“P”表示具有物理（Physical）防篡改（tamper-resistant）功能，包含了多项防范物理攻击的安全特性。

2020 年 2 月 10 日，ARM 公司发布了基于 ARMv8.1 版本的 Cortex-M55 处理器，可看作 Cortex-M33 的下一代产品。M55 首次使用了名为 Helium 的矢量数据处理引擎，该引擎基于 MVE（M-Profile Vector Extension，矢量扩展）技术，将执行 SIMD 指令和 DSP 处理能力提升了 5 倍，机器学习能力提升了 15 倍。与此同时，ARM 公司还发布了一款与 Cortex-M55 配套使用的嵌入式 NPU（Neural network Processor Unit，神经网络处理器）Ethos-U55。在此之前 ARM 公司已有 Ethos-N37，Ethos-N57 和 Ethos-N77 三款 NPU，而 Ethos-U55 是专为下一代 Cortex-M 处理器定制的，具有低功耗和低成本特性。Cortex-M55 搭配 Ethos-U55 将进一步增强 ARM 公司在未来物联网应用领域的产品竞争力。

Cortex-M 系列处理器的主要特性和主要部件的配置参见表 5.4。虽然 Cortex-M 系列在性能、可靠性和实时性方面不如 Cortex-A 和 Cortex-R 系列，但是凭借其所具有的低功耗、小体积和高性价比的特点，问世之后一直是 ARM 公司销售量最大的产品。据 ARM 公司 2019 财年第四季度财报数据显示，ARM 公司的合作厂商在一个季度内总共销售了 64 亿片基于 ARM 处理器的各种芯片，其中有 42 亿片属于 Cortex-M 系列，占出货总量的 66%，其余才是 Cortex-A 系列、Cortex-R 系列、经典处理器以及安全芯片等产品。

表 5.4 Cortex-M 系列处理器特性对比表

型号 / 特性	M0	M0+	M1	M23	M3	M4	M33	M35P	M55	M7
ISA	v6-M			v8-M Baseline	v7-M	v7E-M	v8-M Mainline		v8.1-M Mainline Helium	v7-M
DMIPS	0.87~1.27	0.95~1.36	0.8	0.98	1.25~1.89	1.25~1.95	1.5		1.6	2.14~3.23
CoreMark	2.33	2.46	1.85	2.64	3.34	3.42	4.02		4.2	5.01
流水线级数	3	2	3	2	3				4	6
MPU	-	可选	-	可选(2x)	可选		可选(2x)			可选
MPU 区域	-	8	-	16	8		16			
跟踪单元	-	MTB 可选	-	MTB 或 ETMv3 可选	ETMv3 (可选)		MTB 和/或 ETMv4 可选		ETMv4 可选	
DSP 扩展	-					有	可选			有
浮点单元	-					标量单精度			标量双精度、矢量单精度	标量单/双精度
SysTick Timer	可选			2 x	有		2 x			有
内置 Cache	-							仅 I，可选	I-Cache / D-Cache 可选 4~64kB	
TCM	-								I-TCM / D-TCM 可选 0~16MB	
TrustZone	-			可选	-		可选			-
CP 接口	–						可选			-
总线协议	AHB Lite	AHB Lite Fast I/O	AHB Lite	AHB5 Fast I/O	AHB Lite, APB		AHB5, APB		AXI5 AHB APB	AXI4, AHB Lite, APB
WIC 支持	有			有						
NVIC	有									
外部中断数	32			240			480			240
硬件除法器	-			有						
单周期乘法	可选			可选						
双核锁步	-			有	-		有		-	有

5.2 Cortex-M3/M4 处理器结构

Cortex-M3 与 Cortex-M4 两款处理器极为相似，主要区别是 Cortex-M4 增加了一些增强的 DSP 运算指令，并且可选配浮点处理单元 FPU。Cortex-M3 与 Cortex-M4 是目前应用最为广泛的 ARM 处理器，因此本章将以 Cortex-M3 与 Cortex-M4 为例（以下统称为 Cortex-M3/M4），详细介绍这两款处理器的主要特性、指令集架构、内部组成结构、编程模型、存储器管理以及异常/中断处理。

5.2.1 Cortex-M3/M4 处理器概述及指令集架构

虽然 Cortex-M3 和 Cortex-M4 分别基于 ARMv7-M 和 ARMv7E-M 版本架构，但是两者非常类似。Cortex-M3/M4 的内部各有一条三级流水线（取指、译码和执行），采用指令与数据分离的哈佛结构。这两款处理器共有的主要特性如下：

- 32 位处理器，可以处理 8 位、16 位和 32 位数据；
- 处理器本身不包含存储器，但提供了连接不同存储器的总线接口；
- 多种总线接口，可分别连接存储器、外设以及调试接口；
- 紧耦合的嵌套向量中断控制器 NVIC，能够以确定的周期快速响应中断；
- 丰富的调试和跟踪组件以及外部调试接口；
- 可选配 MPU，实现内存的分区保护；
- 低功耗，低成本（基于 M3 的低端 SOC 芯片单价只需几元人民币）；
- 具有丰富的开发调试工具。

作为低成本 MCU，除了 Cortex-M4 可以选配 FPU 以外，两款处理器内部没有其他协处理器，也没有协处理器接口。

与 Cortex-M3 不同的是，Cortex-M4 可以选配符合 IEEE754 标准的单精度浮点单元 FPU，其功能主要包括：

- 提供多条浮点运算指令，以及多条单精度和半精度浮点数据之间的转换指令。
- 浮点单元支持融合乘加（MAC）运算，所谓融合乘加运算是指在计算 a＋b×c→a 的浮点数时，可以在全部计算完成之后再进行浮点数的舍入（Rounding），减少舍入误差以提高 MAC 结果的精度。
- 如果不需要浮点单元，可以将其关闭从而降低功耗。

即使没有选配 FPU，Cortex-M4 也具有 Cortex-M3 所没有的一些特性，两者在指令集方面的差异如图 5.11 所示。

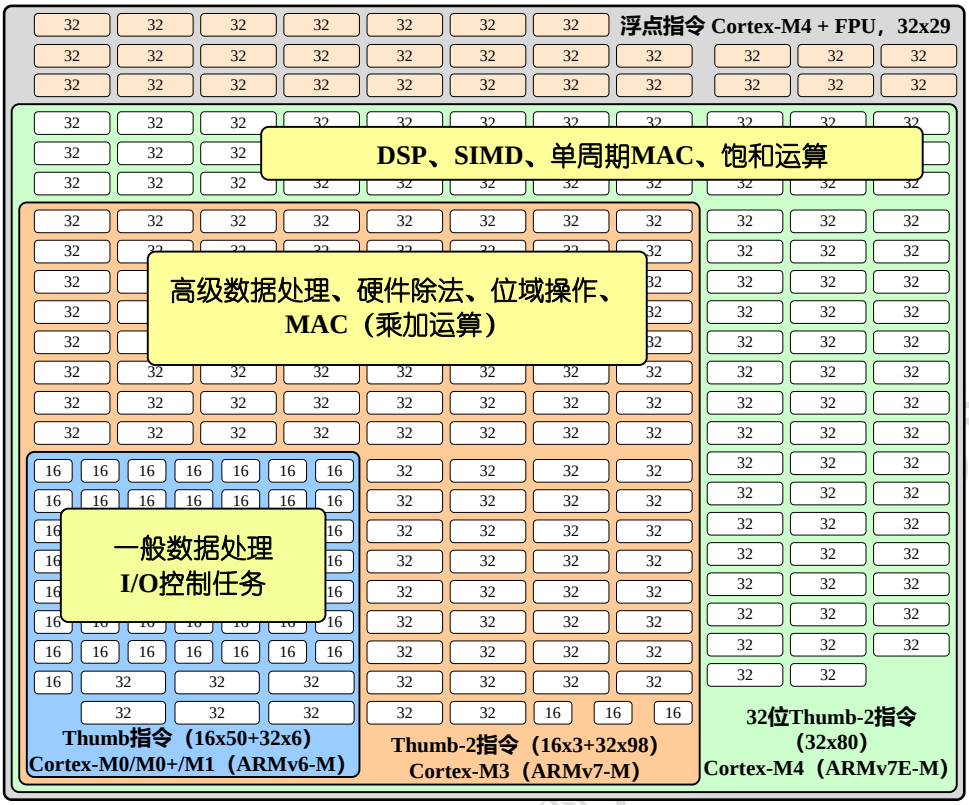

图 5.11 Cortex-M3/M4 处理器在指令集方面的差异

从图 5.11 中可以看出，Cortex-M3/M4 支持 Cortex-M0/M0+/M1 的所有指令，这意味着 Cortex-M0/M0+/M1 处理器可以向上兼容。Cortex-M4 支持 Cortex-M3 的所有指令，但是另外增加了 DSP 扩展功能，例如：

- 增加了支持 8 位和 16 位数据的 SIMD 指令，允许对多个数据同时进行并行处理；
- 支持多个（包括 SIMD 在内的）饱和运算指令，避免在出现上溢出和下溢出时计算结果出现较大畸变；
- 支持单周期 16 位、双 16 位以及 32 位乘加（MAC）运算。

尽管 Cortex-M3 也有几条 MAC 指令，而 Cortex-M4 的 MAC 指令则具有更多选项。其中包括寄存器高低 16 位多种组合的乘法以及 SIMD 形式的 16 位 MAC。Cortex-M4 处理器中的 MAC 运算可以在单周期内快速完成，而 Cortex-M3 则需要花费几个周期。

为了支持额外的浮点指令以及满足 DSP 的高性能需求，Cortex M4 内部的数据通路和 Cortex-M3 处理器也有所不同。由于这些差异，Cortex-M4 某些指令执行所需的时钟周期要少于 Cortex-M3。因此，从某种意义上说，Cortex-M4 可以看作 Cortex-M3 的增强版。表 5.5 是 Cortex-M4 与 Cortex-M3，以及与 Cortex-M0/M0+/M1 所支持的指令功能范围。

表 5.5 几种 Cortex-M 处理器的指令功能范围

指令组	Cortex-M0/M0+/M1	Cortex-M3	Cortex-M4	带有 FPU 的 Cortex-M4
16 位 ARMv6-M 指令	●	●	●	●
32 位间接跳转链接指令	●	●	●	●
32 位系统指令	●	●	●	●
16 位 ARMv6-M 指令		●	●	●
32 位 ARMv6-M 指令		●	●	●
DSP 扩展指令			●	●
浮点指令				●

Cortex-M3/M4 处理器在设计时考虑了对嵌入式操作系统的高效支持，在处理器内核集成了一个系统节拍定时器 SysTick，可以为操作系统所需的定时提供周期性定时中断。SysTick 属于 Cortex-M3/M4 内核设备，所有 Cortex-M3/M4 都带有 SysTick 定时器，提高了在嵌入式操作系统环境下应用软件的通用性。

Cortex-M3/M4 具有两个堆栈指针，操作系统内核和异常处理使用主栈指针 MSP（Main Stack Point），用户程序使用进程栈指针 PSP（Process Stack Point）。这样设计的目的是为了将操作系统内核使用的堆栈与普通应用任务的堆栈进行分离，以提高系统可靠性，并且可以优化堆栈空间。对于不使用操作系统的简单应用，可以只使用 MSP。

为了进一步提高系统可靠性，Cortex-M3/M4 支持独立的特权和非特权访问等级。处理器启动后默认处于特权访问等级。当使用操作系统时，操作系统和中断处理任务具有特权访问等级，而普通用户任务只具有非特权访问等级。划分访问等级的目的是为了增加某些限制，如阻止非特权普通用户任务对某些特殊寄存器的访问，避免其可能对系统正常运行造成不利影响。特权和非特权访问等级还可以与 MPU 配合，防止非特权任务访问某些存储器区域，防止用户任务破坏内核以及其他任务的数据，以提高系统健壮性。对于高可靠性嵌入式应用系统，通过特权和非特权任务的分离，还可以实现应用软件故障隔离。当某个非特权任务出错后，系统和其他应用可能还会继续执行。但是，对于大多数没有操作系统的简单应用，没有必要使用非特权模式。

5.2.2 Cortex-M3/M4 处理器结构

ARM 公司给出的 Cortex-M3/4 处理器系统结构如图 5.12 所示。按照各个部件的作用以及与总线系统的连接关系，整个处理器系统可以分为内核、处理器和系统三个层级，图中用虚线框表示的是可选配部件。

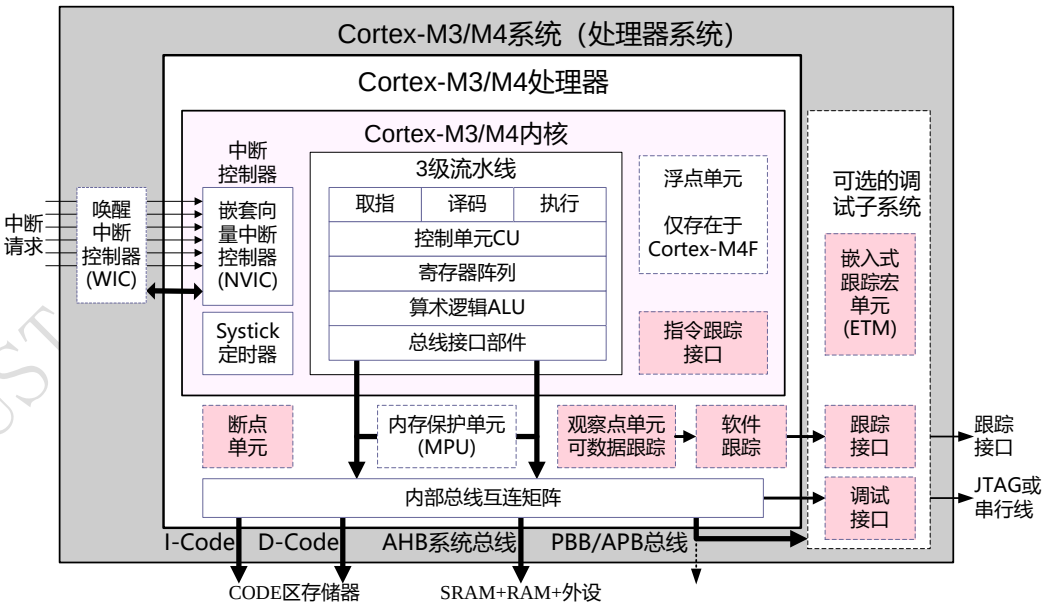

图 5.12 Cortex-M3 和 Cortex-M4 处理器结构框图

1. 内核

Cortex-M3/M4 处理器内核包括 CPU、嵌套向量中断控制器 NVIC、系统节拍定时器 SysTick

以及可选的指令跟踪接口，Cortex- M4 内核还可以选配一个浮点单元（具有浮点单元的 Cortex-M4 产品型号为 Cortex-M4F）。

1） CPU

Cortex-M3/M4 的 CPU 是一个 32 位 RISC 微处理器，内部有一条三级（取指、译码和执行）流水线以及三个关键部件：算术逻辑运算单元 ALU、控制单元 CU 和寄存器阵列。本书第二章已对 ALU、控制单元以及流水线的工作原理做了详细介绍，关于 Cortex-M3/M4 中寄存器组织请参见本书 5.3.4 小节。Cortex-M3/M4 的 CPU 采用哈佛结构，其总线接口部件有取指和数据存取两条总线。

2） NVIC 和 SysTick

在经典 ARM 处理器中，只有 7 种异常类型，分成了 6 个优先等级。这 7 种异常类型以及优先级排序如图 5.13 所示，其中优先级数值越小，优先级越高。

优先级	1	2	3	4	5	6	
异常名称	复位 Reset	数据中止 Data Abort	快速中断 Fast RQ	外部中断 IRQ	预取中止 Prefetch Abort	未定义指令 Undefined	软件中断 SWI

图 5.13 经典 ARM 处理器异常类型和优先级

如果需要管理的外部中断数不止一个，经典 ARM 处理器在内核之外可以增加矢量中断控制器 VIC（Vector Interrupt Controller）。与 ARM7TDMI 配套的 VIC 最多可管理 32 个外部中断，VIC 的功能包括中断排队、优先级管理、向量中断和中断屏蔽等。

Cortex-M 系列处理器在异常处理方面做了较大改进。Cortex-M 系列处理器在内核中集成了一个与 CPU 紧耦合的嵌套中断控制器 NVIC，对系统异常、不可屏蔽中断 NMI 以及外部中断 IRQ 进行全面管理。此外，Cortex-M 系列处理器内核还集成了一个简单的倒计时计数器 SysTick，专门负责产生类型号为 15 的系统定时异常（中断）。鉴于异常/中断的重要性，本书将在 5.2.5 小节对 NVIC 和 SysTick 的特性进行概述，并在 5.5 节详细介绍 Cortex-M3/M4 的异常/中断处理机制和处理过程。

3） FPU

Cortex-M4 与 Cortex-M3 的主要区别之一是增加了一个可选的浮点运算单元 FPU，大大提高了 Cortex-M4 的浮点运算功能。Cortex-M4 浮点运算单元支持符合 IEEE 754-2008 标准的单精度浮点运算。该浮点单元主要功能如下：

- ❒ 浮点寄存器组包含 32 个 32 位寄存器，即可以作为 32 个寄存器单独使用，也可以两两配对作为 16 个双字寄存器使用；
- ❒ 支持的转换指令包括“整数↔单精度浮点”、“定点↔单精度浮点”、“半精度↔单精度浮点”；
- ❒ 支持单精度数据和双字数据在浮点寄存器组和存储器之间传输；
- ❒ 支持单精度数据在浮点寄存器组和整数寄存器组之间传输。

但是，Cortex-M4 浮点单元不支持双精度浮点运算、浮点余数以及二进制与十进制之间的转换，如若需要这些功能只能另外编程实现。Cortex-M4 浮点运算单元的版本为 FPv4-SP，是 ARMv7-A 和 ARMv7-R 中的矢量浮点单元 VFPv4-D16 的子集。这两个浮点单元版本中大部分指令是通用的，因此有时将 Cortex-M4 浮点单元 FPU 也称为 VFP，并且浮点运算指令的助记符都是以 V 开头。

在 ARM 处理器架构中，浮点单元一般位于处理器内核设备与协处理器之间，具有和其协

处理器类似的状态控制寄存器，用以标明浮点运算单元的当前运行状态，并可通过编程对其进行控制。为了与其他处理器中的协处理器编程控制方式保持一致，浮点单元也被看作 CPU 的一个协处理器。在 Cortex-M 系列处理器的系统控制块 SCB（System Control Block）中，有一个协处理器访问控制寄存器 CPACR（Co-processor Access Control Register），其中 CP10 和 CP11 两位负责对 FPU 进行管理和控制，所以有时也将 FPU 视为协处理器 CP10 和 CP11。通过对 CPACR 寄存器中 CP10 和 CP11 两位置位或者复位操作，可以使能或禁用 FPU。

在经典 ARM 处理器（如 ARM7 和 ARM9）中，有单独的协处理器寄存器访问指令（MCR 和 MRC），通过这些专门的传送指令向协处理器中的寄存器传送数据和操作码，从而实现对协处理器的操作控制。Cortex-M4 对 FPU 的操作控制方式与经典 ARM 处理器完全不同。Cortex-M4 有专门的浮点运算指令和数据传送指令，通过这些指令可以把数据从存储器取出并传送到 FPU 的寄存器中，在完成浮点运算之后再把运算结果写回存储器。在 Cortex-M4 的三级流水线中，浮点处理单元 FPU 和 CPU 共用取指部件，在译码和执行阶段则二者并行执行，如图 5.14 所示。

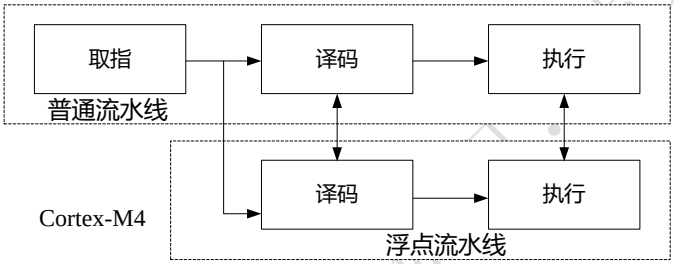

图 5.14　FPU 与 CPU 并行流水线

2. 处理器

Cortex-M3/M4 处理器除了内核以外，最重要的部件就是总线交换矩阵（Bus Matrix）。总线交换矩阵是一个基于 AHB 总线协议的交换网络。Cortex-M3/M4 通过总线交换矩阵，面向各种存储器，片上和片外不同类型的设备以及调试组件提供了多条总线，可以让数据和指令在不同的总线上并行传送（只要不是访问同一个器件）。Cortex-M3/M4 的总线矩阵还包含一个写缓冲区，可以加快存储器写操作速度。Cortex-M3/M4 处理器的系统总线基于 AHB-Lite 总线协议，属于 AHB 总线的“轻量级”版本，总线上只有一个主设备（主机），无需使用总线仲裁。本书第四章已对 AMBA 总线规范做了介绍，此处不再赘述。关于 Cortex-M3/M4 总线系统将在 5.2.4 小节介绍。

Cortex-M3/M4 都可以选配内存保护单元 MPU，通过 MPU 把存储空间划分为最多 8 个区域，区域之间也可以重叠。各个区域的存储特性和访问权限可以通过编程定义。如果使用了嵌入式操作系统，MPU 由操作系统进行管理，给每个任务分配不同存储区域以及访问权限，防止某个应用任务对操作系统或者其他任务的数据造成破坏。

在没有操作系统的简单应用中，MPU 也可以通过应用软件编程来设定需要保护的存储区域，例如，把某个存储空间设置为只读属性，或者把某个区域设置为只有特权用户才能访问，以提高系统安全性和可靠性。

3. 处理器系统

从图 5.12 可以看出，在处理器的基础上，再选配唤醒中断控制器 WIC（Wake-up interrupt

controller）以及若干调试组件之后，就构成 Cortex-M3/M4 处理器系统。关于 WIC 和调试组件的功能和作用简介如下。

1） WIC

为了减少功耗，Cortex-M 系列处理器引入了“睡眠”和“深度睡眠”两种模式。当处理器处于睡眠模式时，大部分功能模块/部件的时钟都被停掉；在深度睡眠模式时，甚至系统时钟和 SysTick 也被关闭。有些版本的 Cortex-M4 系列处理器采用了名为状态保持功率门（State Retention Power Gating，SRPG）的电路，在深度睡眠模式时，包括内核和 NVIC 在内的大部分电路都处于掉电状态，以进一步减少功率消耗。处理器在几种不同工作方式下的电流消耗差异情况可以用图 5.15 表示。

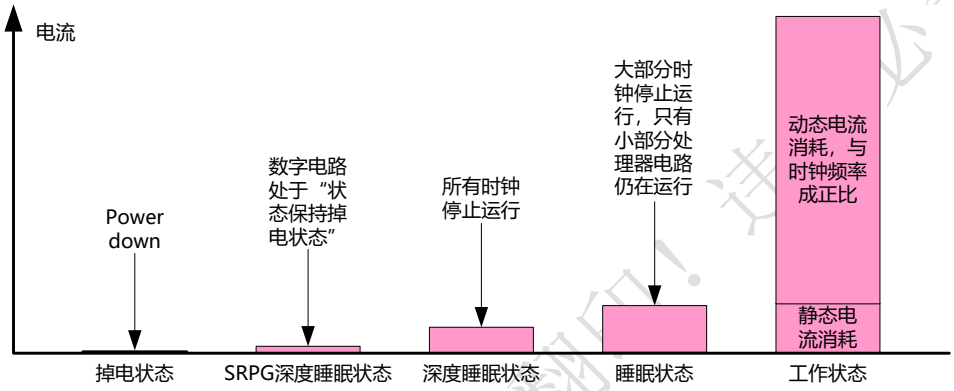

图 5.15 几种不同工作方式下的电流消耗情况

但是，处理器处于深度睡眠模式时，所有时钟都停止运行，甚至所有的电路都处于“状态保持掉电”状态，无法再检测到中断信号。为了使处理器在深度睡眠模式时能够“醒来”，支持深度睡眠的 Cortex-M 处理器[7]引入了与之配套的 WIC 特性。

WIC 的电路非常小巧，通过专有接口与 NVIC 以及电源管理单元 PMU（Power Management Unit）相连。当处理器进入深度睡眠模式时，WIC 立即使能。如果在深度睡眠模式下出现了 NMI 或者未被屏蔽的 IRQ 时，WIC 触发电源管理单元 PMU，将整个系统唤醒。WIC 与系统部件之间的连接如图 5.16 所示。

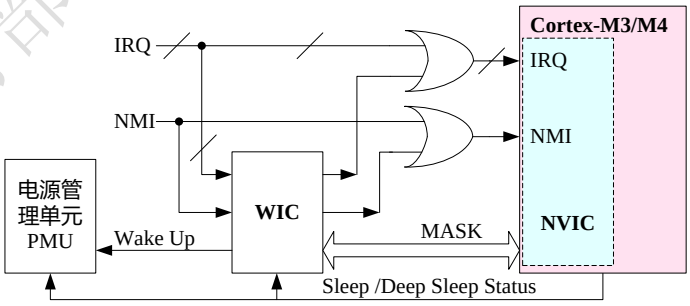

图 5.16 WIC 与系统部件的连接

2） 调试组件

经典 ARM 处理器中也有多种调试组件，大都带有 JTAG 接口，并通过 JTAG 接口实现对寄存器和存储器的访问。Cortex-M 系列处理器采用了一种全新的调试架构，ARM 公司将其命名为 CoreSight。CoreSight 调试架构涵盖了调试接口协议、调试总线协议、调试部件控制、安

[7] r1p1 以后版本的 Cortex-M3 才支持 WIC。

全特性以及跟踪接口等。

在 Cortex-M3/M4 处理器中，调试过程是由 NVIC 和若干调试组件协作完成的。NVIC 中有一些用于调试的寄存器，通过这些寄存器对处理器的调试动作进行控制，如停机（halting）和单步执行（stepping）等。Cortex-M3/M4 可以选配多种调试组件。根据调试组件的作用和功能，不同的组件被连接到不同的总线上。各种调试组件与总线系统的连接关系如 5.2.4 节图 5.20 所示。这些调试组件提供了指令断点、数据观察点、寄存器和存储器访问、性能分析（profiling）以及各种跟踪机制。这些调试组件中除了本书 2.7 节提到过的 ETM（嵌入式跟踪宏单元）以外，还包括：

- ITM：Instrumentation Trace Macrocell Unit，指令跟踪宏单元；
- DWT：Data Watchpoint and Trace Unit，数据观察点和跟踪单元；
- FPB：Flash Patch and Breakpoint Unit，Flash 地址重载和断点单元；
- TPIU：Trace Port Interface Unit，跟踪端口接口单元。

所谓调试（Debug）是指通过调试接口读取或修改处理器内部的寄存器或存储器的内容，或者发布一些调试命令，让处理器执行某些调试动作，如暂停、单步执行或者断点执行等。而跟踪（Trace）是指在程序运行期间，无需停止处理器正常的指令执行流程，由相关的跟踪组件实时收集处理器在指令执行过程中产生的各种运行信息，并通过跟踪接口实时输出到外部调试主机中，再由调试分析软件（如 Keil）对这些信息进行分析，以便系统开发者了解系统的详细运行情况。

在 CoreSight 的调试架构中，ETM、DWT 和 ITM 等调试组件都属于跟踪数据源（Trace Source），由这些组件收集处理器运行过程中产生的各种调试信息，经调试总线（在 Cortex-M3/M4 中是 APB）发往 TPIU 进行汇集。TPIU 属于 CoreSight 的调试架构中 Trace Sink（调试信号汇集点）部件，TPIU 将汇集的调试信息进行格式转换和打包之后，再通过跟踪端口输出到外部的调试主机（PC 机）。Cortex-M3/M4 处理器中的调试信息流如图 5.17 所示。

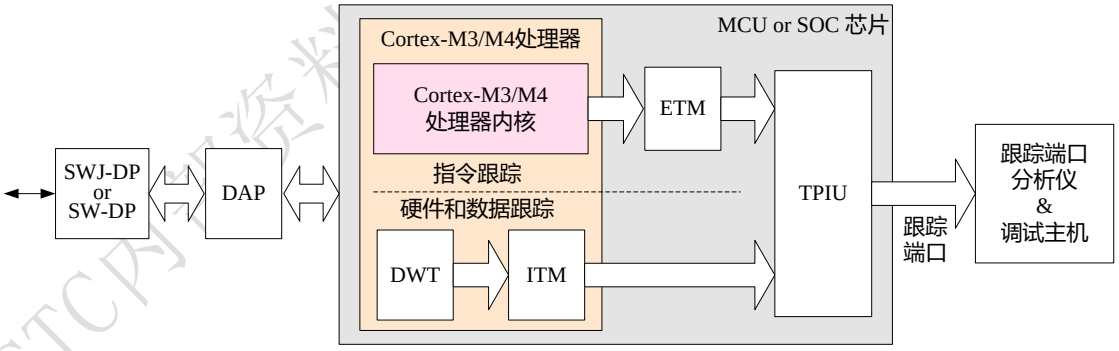

图 5.17 调试信息流

在 CoreSight 调试架构中，定义了处理器内部的调试访问接口 AP（Access Port）与外部调试端口 DP（Debug Port）。AP 的功能是接收来自 DP 的调试命令，并将这些命令转换成对处理器内部各寄存器和存储单元的访问。DP 和 AP 合称为调试访问端口 DAP（Debug Access Port），DAP 与系统的连接方式参见 5.2.4 小节的图 5.22。

Cortex-M3/M4 处理器仍然支持传统的基于 JTAG 协议的 SWJ-DP（Serial Wire JTAG DP）接口。由于 JTAG 接口有 4 条线（模式 TMS、时钟 TCK、数据输入 TDI 和数据输出 TDO），为了减少芯片引脚，Cortex-M3/M4 还支持只有 2 条线的 SW-DP（Serial Wire DP）接口，如图 5.18 所示，芯片制造商可以根据需要选择其中的一种或者两种。

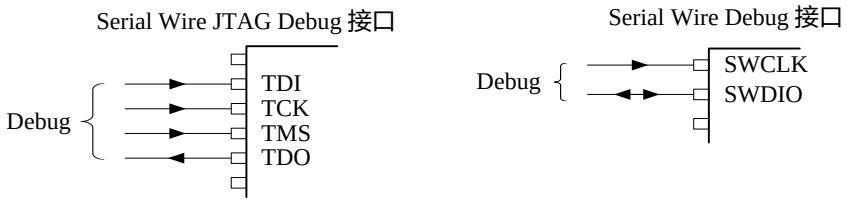

图 5.18 SWJ-DP 和 SW-DP 接口

以上各种调试组件都是可选件，不同芯片制造商针对不同的应用背景可以选配不同的调试组件。芯片制造商将所选的调试组件配置信息，都记录在一个称为 ROM 表的非易失存储器中，ROM 表的作用类似于注册表。

关于 ARM 处理器具体调试方法以及 CoreSight 的更多内容请参阅相关资料。

5.2.3 存储器管理

从图 5.12 所示 Cortex-M3/M4 处理器结构图中可以看出，Cortex-M3/M4 处理器内核没有 Cache 或者 TCM 之类的高速存储器，而是通过多条总线连接片上的各种不同类型和容量的存储器件。如果需要进一步扩展存储容量，还可以通过存储器接口控制器，连接片外大容量存储器。本小节只对 Cortex-M3/M4 在存储管理方面的特性以及存储器的映射关系作简单介绍，更加具体的内容详见本书 5.4 节。

1. Cortex-M3/M4 存储器管理特性

在存储器管理方面，Cortex-M3/M4 处理器具有以下特性：

- 4GB 线性地址空间。虽然 AHB-Lite 总线属于 32 位总线，但是通过适当的存储器接口控制器，可以连接 32 位、16 位和 8 位的存储器件。
- 确定的存储器映射关系定义，旨在优化处理器的性能。4GB 存储空间被划分为多个区域，分别用于不同的存储器和外设。例如，Cortex-M3/M4 处理器具有多个总线接口，允许同时访问程序代码区以及 SRAM 区或者外设区域。
- 支持小端和大端的存储器系统。但是芯片制造商可能只选择其中一种配置类型。
- Bit-Band Operations（位带操作，也称为位段或者位域操作，可选）。在经典 ARM 处理器中，如果想修改某个存储单元或者 I/O 端口的某一个 bit 位而不影响其它位，需要顺序执行读出、修改和写入三个步骤。而位带操作可以使用地址直接对这个 bit 位直接进行操作，无需担心影响其他位。但是具体的 MCU 或者 SOC 芯片是否带有位带操作特性则由芯片制造商决定。
- 写缓冲。Cortex-M3/M4 处理器内置写缓冲，可以提高程序的执行速度。
- 存储器保护单元 MPU（可选）。Cortex-M3/M4 处理器中的 MPU 支持 8 个可编程区域，可在嵌入式操作系统环境下提高系统健壮性。
- 非对准传送。所有基于 ARMv7-M 的 Cortex-M 系列处理器都支持非对准传送，但是非对准传送将额外增加总线传送次数，影响数据传送效率。

2. 存储器映射

所有基于 ARMv7-M 的 Cortex-M 系列处理器，都采用图 5.19 所示的存储器映射关系方式。Cortex-M 系列处理器将 4GB 地址空间分成以下几个区域：

- 程序代码访问区域（CODE 区域），多采用 flash 器件存储程序代码；

- 数据存储区域，又分为片内 SRAM 以及片内或者片外 RAM 区域；
- 外设端口区域，也分为片内设备或者片外设备两个区域；
- 处理器的内部控制和调试部件区域。

在 Cortex-M 系列处理器中，虽然有明确和清晰的存储区域划分，但是仍然不失灵活性。例如，程序代码既可以存放 CODE 区域，也可以存放在 AHB-Lite 总线所连接的 SRAM（主存）中；芯片制造商也可以在 CODE 区域使用 SRAM 存储器。针对嵌入式系统的实际应用背景，芯片制造商往往只会使用每个区域的一小部分用于程序 flash、SRAM 和外设，有些区域可能不会用到。不同的 MCU 或者 SOC 芯片使用的存储器大小和外设地址可能各不相同，芯片制造商的数据手册会对此有详细描述。

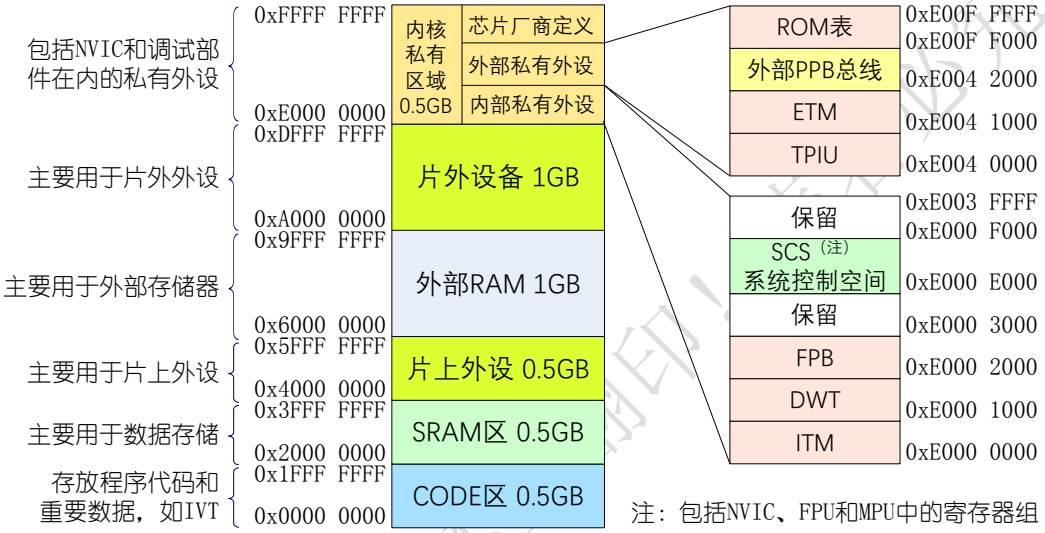

图 5.19 Cortex-M 系列处理器的存储器映射

Cortex-M 系列处理器将位于 0xE000 0000~0xFFFF FFFF 之间的空间划为内核私有区域，该区域又分为内部私有设备、外部私有设备和芯片厂商定义三个子区域。其中，位于内部私有外设子区域的 SCS（System Control Space，系统控制空间）区，包括 NVIC、SysTick、系统控制块 SCB、FPU 和 MPU 等在内的各种系统部件寄存器组。

此外，Cortex-M 系列处理器中各种调试组件，如 ITM、DWT、FPB、TPIU、ETM 和 ROM 表，也都映射在图 5.19 所示的内部私有外设以及外部私有外设子区域。在所有 Cortex-M 系列处理器中，映射到内核私有区域的各类寄存器都有相同地址。这种统一的地址映射方案有助于提高基于 Cortex-M 设备之间的软件可移植性和代码可重用性。

5.2.4 总线系统

Cortex-M3/M4 处理器总线系统的结构如图 5.20 所示，其核心是基于 AHB 总线协议的总线交换矩阵，总线交换矩阵所连接的各类总线的作用以及主要特性简介如下

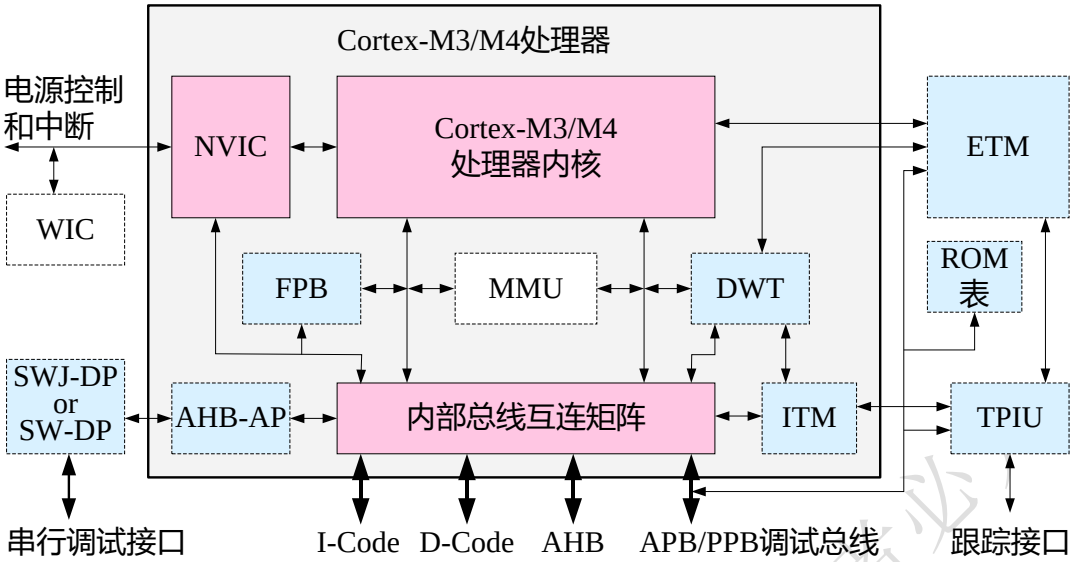

图 5.20 Cortex-M3/M4 总线结构

- I-Code 总线：基于 AHB-Lite 总线协议的 32 位总线，负责在 0x0000 0000~0x1FFF FFFF 之间（CODE 区）进行取指操作。取指是以字为单位，对于 16 位 Thumb 指令，一次取指操作可以取出两条指令。
- D-Code 总线：也是基于 AHB-Lite 总线协议的 32 位总线，所能访问的区域与 I-Code 总线相同，也是 0x0000 0000~0x1FFF FFFF 之间的 CODE 区，负责对位于该区域的数据进行访问。

从以上两条总线可访问的地址范围可以看出，尽管 Cortex- M3/M4 的取指和数据读写是在两条总线上分别进行，但是，I-Code 和 D-Code 只能访问 CODE 区内共用的 512MB 空间。两条总线在物理上互相独立，彼此之间有一个仲裁器。当 I-Code 和 D-Code 同时访问同一个存储部件或者存储区域时，此时将发生冲突，仲裁器判定 D-CODE 优先，I-Code 等待。如果将代码和数据分别存放于 CODE 区内两个地址不同的存储器中，则可以避免上述冲突。CODE 区的程序代码通常使用 flash 型闪存，而数据一般使用 SRAM 型存储器。

- System（系统总线），基于 AHB-Lite 总线协议的 32 位总线，有时也称为 AHB 总线，负责在 0x2000 0000~0xDFFF FFFF 以及 0xE010 0000~0xFFFF FFFF 之间的所有数据传送。
- APB/PPB（Private Peripheral Bus），32 位 APB 总线，负责连接 0xE004 0000~0xE00F FFFF 之间的外部私有外设。从图 5.19 所示的内存映射表中可以看出，在外部私有外设子区域中，有一部分空间已被 ETM、TPIU 等调试组件所占用，只有 0xE004 2000~0xE00F EFFF 之间可用于外部私有设备连接。但是，该总线是专用的，一般不用于普通外设，否则将会因特权管理问题出现各种错误。该接口包含了一条名为“PADDR31”的信号，以标识发起传送的“源”方。若该信号为 0，则表示是内部软件发起的传送操作；若为 1，则表示是调试硬件产生了传送操作。外设可以根据这个信号有选择地作出响应。例如，只响应调试硬件发起的请求，或者只部分响应软件发起数据传送请求。

以上 I-Code、D-Code、System 和 PPB 四条总线所能访问的区域如图 5.21 所示。注意到 0xE000 0000~0xE003 FFFF 是内核私有外设（参见图 5.19）区域，CPU 对该区域的访问不经过

上述四条总线，而是通过内部总线互连矩阵直接对该区域进行访问。

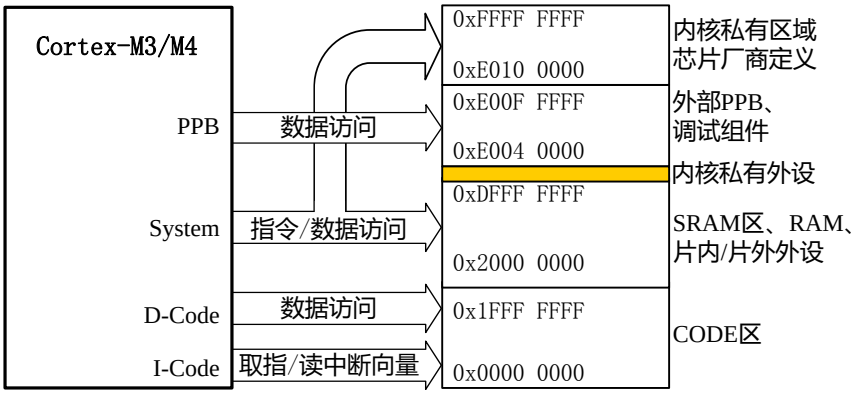

图 5.21 四条总线的访问区域

除了上述四条总线以外，Cortex-M3/M4 处理器还有一个调试访问端口（DAP），用于连接处理器内部的调试访问端口 AHB-AP 与外部调试端口 DP，如 SWJ-DP 和 SW-DP。AHB-AP 与内部总线互连矩阵之间有一条基于“增强型 APB 规格”的 32 位总线，其连接方式如图 5.22 所示。

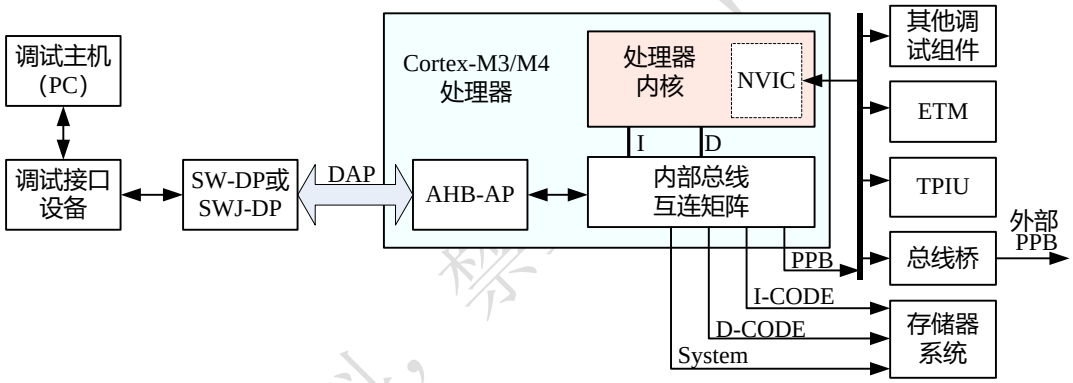

图 5.22 DP 与 AP 的连接

Cortex-M3/M4 各类总线与各种存储器以及外设的连接方式如图 5.23 所示。需要指出的是，图 5.23 中的 AHB 总线互联矩阵属于处理器内核设备，而 CODE 区的总线矩阵或者总线复用器是 ARM 公司或者其他 IP 供应商和芯片制造商提供的选件，其作用是让 I-Code 和 D-Code 都能够访问 CODE 区的 flash 和 SRAM。

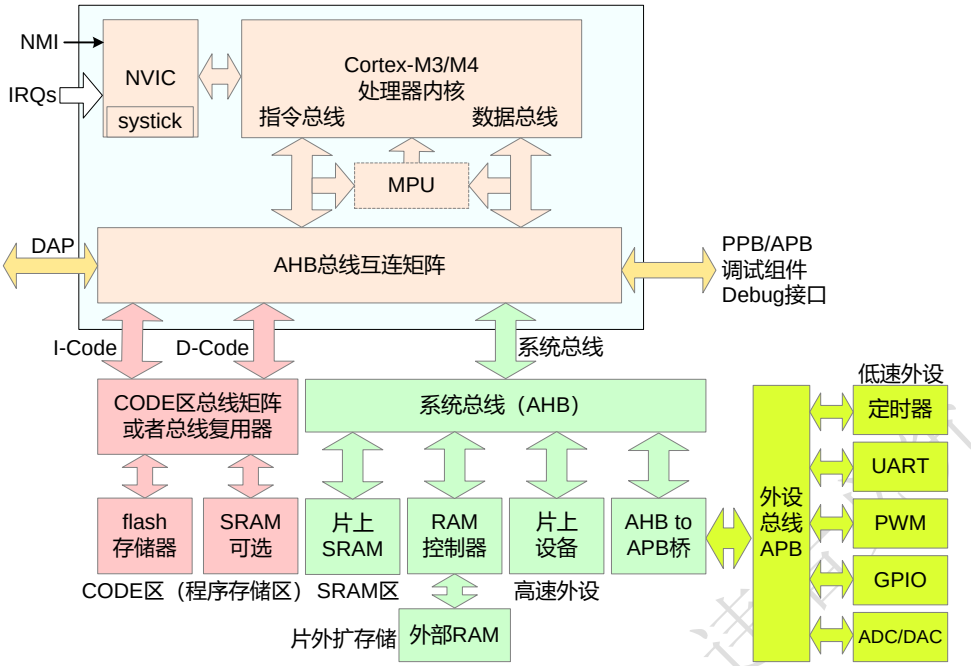

图 5.23 Cortex-M3/M4 总线与各个部件的连接方式

还需要说明的是，CODE 区的总线矩阵和总线复用器是两种不同的选件，并具有不同的特性。如果选用总线矩阵，I-Code 对 flash 的取指操作与 D-Code 对 SRAM 的数据存取操作可以同时进行；如果选用的是总线复用器，I-Code 和 D-Code 对 CODE 区的访问只能分时进行，I-Code 和 D-Code 上的数据传送不再有具有并行性，但是可以减少芯片中的电路数以及芯片面积。也有一些芯片制造商取消了原本应该放到 CODE 区的 SRAM，将数据统一存放到由系统总线连接的片上 SRAM 中，利用系统总线与 I-Code 总线的并行性进行数据传送。这样不仅可以减少一块 SRAM，而且 CODE 区只需使用相对简单和低成本的总线复用器即可。

片上 SRAM 也称为主存储器，必须连接到系统总线上，使其位于 SRAM 区，以便可以使用位带操作。关于位带区所处位置以及位带操作的具体内容详见本书 5.4 节相关内容。

如果需要扩展内存容量而使用片外存储，必须使用片外 RAM 控制器作为接口，Cortex-M3/M4 系统总线不能直接连接片外存储器。

图 5.23 中的总线矩阵或者总线复用器、片外 RAM 控制器、AHB 到 APB 的总线桥、AHB-AP 调试访问端口以及 UART 等各种外设接口都可以根据需要选配。除了 ARM 公司以外，还有其他 IP 供应商也能提供。

5.2.5 异常与中断处理

1. 嵌套向量中断控制器 NVIC

本书第四章已对中断和异常的概念做了介绍。在 ARM 处理器中，中断被看作一种特殊的异常类型，通常由外设或者外部输入触发，或者由软件设置触发。异常和中断处理程序通常也被称为中断服务程序 ISR（Interrupt Service Routine）。

与经典 ARM 处理器相比（参见图 5.13），ARMv7-M 版本架构采用了一种全新的异常/中断处理机制，取消了 FIQ，取而代之的是逻辑关系更加清晰的中断优先级机制，并且全面支持嵌套中断管理。Cortex-M3/M4 处理器集成了一个与 CPU 紧耦合的嵌套中断控制器 NVIC，总共可以管理从 IRQ#0 到 IRQ#239 共 240 个外部中断、一个不可屏蔽中断 NMI 以及多个系统

异常。其中，类型号为 0~15 的异常分配给系统异常（实际上只使用了 11 个，包括 NMI，有 5 个作为保留）；类型号从 16 到 255 的 240 个异常分配给了外部中断 IRQ。异常类型号的具体分配方式以及优先级安排如表 5.6 所示。

表 5.6 异常类型分配和优先级一览表

编号	类型	优先级	简介
0	N/A	N/A	没有异常在运行
1	复位 Reset	-3（最高）	复位
2	NMI	-2	来自 NMI 引脚，一般由看门狗或者掉电监测单元 BOD 产生
3	HardFault 硬件错误	-1	如果相应的异常处理未使能，所有错误都可能引发此异常
4	MemManage 错误	可编程	访问内存的行为违法了 MPU 定义的规则
5	总线错误	可编程	AHB 收到从总线的错误响应，如指令预取和数据读写被终止
6	用法错误	可编程	无效指令或试图访问未配置的部件，如访问 M3/M4 没有的协处理器
7~10	保留		
11	SVC	可编程	有 OS 时，应用程序可藉此调用系统服务（类似于 DOS 调用）
12	调试监视	可编程	使用基于软件的调试时，断点和数据观察点等调试事件的异常
13	保留		
14	PendSV	可编程	可挂起（缓期执行）的 SVC，常用于多任务 OS 的上下文切换
15	SYSTICK	可编程	系统节拍定时器产生的周期性异常，例如任务之间的切换定时
16	IRQ#0	可编程	由片上外设或者外设中断源产生
17	IRQ#1	可编程	
	... ...	可编程	
255	IRQ#239	可编程	

虽然 Cortex-M3/M4 处理器可以管理 240 个外部中断，但是，为了支持数量众多的外部中断需要增加相应门电路和芯片引脚，同时也增加了产品成本和芯片功耗。因此，芯片制造商一般根据具体应用场景选择合适的外部中断数量。实际 MCU 或者 SOC 产品的外部中断数量一般只有 8 个或者 16 个，很少超过 64 个。另外，某些产品也没有提供外部 NMI 引脚。

需要注意的是，在调用 CMSIS[8]设备驱动库编程时，CMSIS 库的中断类型编号范围是-15 ~ 239，CMSIS 库的中断编号 0 对应的是 IRQ#0，中断编号 239 对应的是 IRQ#239；而编号为负数的-15 到-1 依次对应的是复位到 SYSTICK 等系统异常。

NVIC 是 Cortex-M3/M4 内核不可分割的一部分，NVIC 在内核中的连接方式如图 5.24 所示。

[8] 微控制器软件接口标准（Cortex Microcontroller Software Interface Standard），由 ARM 公司联合多家芯片和软件供应商合作定义，包含多个组件。其中，内核和设备访问库函数 CMSIS-core 提供了多种函数，可对 CPU 内部各种寄存器以及内核设备进行访问。关于 CMSIS 参见本书第七章相关内容。

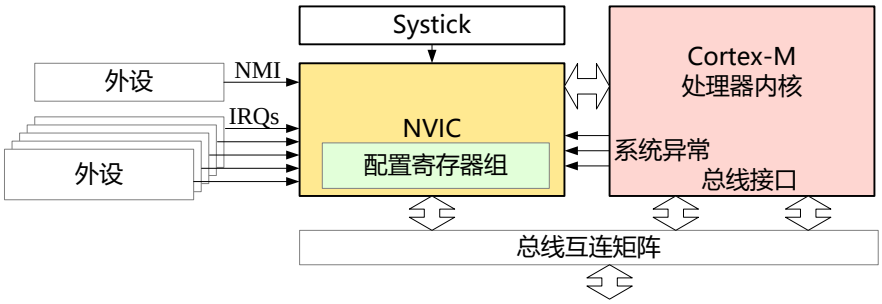

图 5.24 NVIC 在 Cortex-M 处理器中的连接方式

Cortex-M3/M4 通过 NVIC，可对 240 个外部以及大部分系统异常进行全面和细腻的集约化管理，其中包括优先级设定、中断屏蔽、嵌套（抢占）管理、中断状态识别、中断挂起[9]（Pending）和解挂等事务。在异常处理过程中，NVIC 与内核相互配合，使得 Cortex-M3/M4 的异常处理机制更加高效。Cortex-M3/M4 处理器在异常/中断处理方面的特性还包括：

- 除了复位和 NMI 之外，其它所有异常/中断都可以被屏蔽；
- 除了复位、NMI 和硬件错误之外，其它所有异常/中断都可以单独使能或禁止；
- 除了复位、NMI 和硬件错误具有固定的（高）优先级之外，其它所有异常/中断都具有多达 256 级可编程优先级，以及最多 128 个可抢占（嵌套）优先级（关于中断处理以及中断抢占等内容将在本书 5.5 节介绍）；
- 支持优先级的动态修改（Cortex-M0/M0+无此特性）；
- 支持向量中断/异常方式，中断响应时自动给出中断/异常处理程序入口地址；
- 向量表可以重定位在存储器中的其他区域；
- 低中断处理延迟，对于零等待的存储器系统，中断处理延迟仅为 12 个时钟周期；
- 中断和多个异常可由软件触发；
- 可以按照优先级对中断进行屏蔽；
- 进入中断/异常服务程序时可自动保存包括 PSR 在内的多个寄存器，异常返回时自动恢复，无需另外编程；
- 可选配唤醒中断控制器 WIC（Wake-up Interrupt Controller），支持睡眠（Sleep）以及深度睡眠（Deep Sleep）。

在经典 ARM 处理器中，所有外部 IRQ 同属一个优先级，外部 IRQ 之间的优先级以及中断嵌套只能通过外部 VIC 或者软件编程实现。而 Cortex-M3/M4 最多可以设置 128 个可抢占（嵌套）优先级，可通过 NVIC 对所有的外部中断和大多数系统异常实现硬件嵌套管理，无需另外编程控制。当前正在处理的异常/中断优先级被存储在程序状态寄存器的专用字段中。在允许抢占（嵌套）的情况下，当出现新的异常/中断时，硬件电路自动将其可抢占优先级与当前正在处理的异常/中断进行比较，如果新异常/中断的抢占优先级更高，就会中断当前正在执行的中断服务程序，转而处理新的异常/中断，从而实现中断嵌套。

为了支持这些特性，Cortex-M3/M4 处理器以及 NVIC 使用了多个可编程寄存器。这些寄存器被映射在内存的 SCS 区，可通过地址和汇编指令对其进行访问。此外，CMSIS-Core 提供了丰富的访问函数（API），可以对这些寄存器进行定义，实现常规的中断控制任务。这些访问函数易于使用，而且其中大多数可同样用于包括 Cortex-M0 在内的的其他 Cortex-M 处理器。

[9] 由于优先级等原因，某个中断服务请求不能立刻得到处理，称为中断被挂起。

2. 中断向量表

Cortex-M3/M4 处理器将所有异常/中断处理程序的入口地址集中存放在中断向量表中。Cortex-M3/M4 处理器的中断向量表只有异常/中断服务程序的入口地址，每个入口地址为 32 位，占用 4 个字节，这与经典 ARM 处理器有很大的不同。在经典 ARM 处理器中，不仅所能处理的异常和中断数量较少，而且中断向量表中还包含了跳转到中断服务程序入口的转移指令。Cortex-M3/M4 处理器总共有 256 个异常/中断类型，所以中断向量表的容量为 4x256=1024 个字节（1 KB）。中断向量表作为一类系统表，起始地址默认位于存储器空间最开始位置（地址 0x0）。Cortex-M3/M4 处理器的中断向量表中的具体内容如表 5.7 所示。

表 5.7 Cortex-M3/M4 处理器中断向量表

异常类型	CMSIS 中断编号	地址偏移	中断向量
N/A	N/A	0x00	主堆栈 MSP 初始值
1	N/A	0x04	Reset 复位
2	-14	0x08	NMI 不可屏蔽中断
3	-13	0x0C	Hard Fault 硬件错误
4	-12	0x10	MemManage Fault 存储管理错误
5	-11	0x14	Bus Fault 总线错误
6	-10	0x18	Usage Fault 用法错误
N/A	N/A	0x1C	保留
N/A	N/A	0x20	保留
N/A	N/A	0x24	保留
N/A	N/A	0x28	保留
11	-5	0x2C	SVC 系统服务调用
12	-4	0x30	Debug Monitor 调试监视
N/A	N/A	0x34	保留
14	-2	0x38	PendSV 可挂起请求
15	-1	0x3C	SYSTICK 系统定时
16	0	0x40	IRQ#0
17	1	0x44	IRQ#1
18~255	2~239	0x48~0x3FF	IRQ#2~IRQ239

为了加快 Cortex-M3/M4 处理器的中断处理速度，在中断响应时，读取中断向量以及中断服务程序的取指操作应与保护断点的寄存器压栈操作同时进行。由于中断向量表位于存储空间最开始的位置，而这片区域属于存放程序代码的 CODE 区，在中断响应时，应由 I-Code 总线负责读取中断向量。而中断响应时的压栈操作理论上可由 D-Code 或者系统总线完成，以充分发挥哈佛结构的优势。但是，有些芯片制造商为了减少一片 CODE 区的 SRAM，把数据都存放在由系统总线负责管理的 SRAM 区，因此压栈操作只能由系统总线完成。

在某些时候，中断向量表在系统中的存放位置可能需要改变，这种改变称为中断向量表的重定位（relocate）。Cortex-M3/M4 提供了中断向量表重定位特性，中断向量表可以移至 CODE 区其他位置或 SRAM 主存储区。关于中断向量表重定位将在本书 5.5 节介绍。

3. 系统节拍定时器 SysTick

多任务操作系统大都采用分时处理的原则，将处理器时间划分成若干个时间片，在不同的时间片内，处理器轮流为每个任务进行服务。因此，系统中需要一个定时器来产生周期性的定时信号，“提醒”操作系统对任务进行切换。此外，计算机中还有其他系统事务也需要定时信号，例如 DRAM 的刷新定时和实时时钟等。

为了支持多任务操作系统，Cortex-M 系列处理器集成了一个系统节拍定时器 SysTick，其

作用是产生周期性的 SYSTICK 中断（异常号#15）。SysTick 定时器与 NVIC 捆绑在一起，也可以认为是 NVIC 的一部分，都属于内核设备。SysTick 内部包含一个 24 位递减计数器和如下 4 个寄存器：

- ❒ 状态控制寄存器 STCSR（SysTick Control and Status Register）；
- ❒ 加载值寄存器 STRVR（SysTick Reload Value Register）；
- ❒ 当前计数值寄存器 STCVR（SysTick Current Value Register）；
- ❒ 校准值寄存器 STCR（SysTick Calibration Value Register）。

可以通过对这些寄存器的编程操作实现对 SysTick 的管理。例如，当 SysTick 被使能后，便从 STRVR 的数值开始，按照时钟周期进行递减计数，减至 0 后在下个时钟周期又从 STRELOAD 中重新加载，重复递减计数，每次减至 0 便产生一个标志事件，从而产生一个周期性的中断触发。

在没有使用操作系统的嵌入式系统中，SysTick 定时器可以作为普通定时器，用于产生周期中断、延时和时间测量等。

SysTick 设计时考虑了系统保护，非特权访问等级的用户程序不能随意访问 SysTick 的寄存器，以免影响其产生正常的周期性中断信号。若编程中使用了 CMSIS-Core，具有特权访问等级的用户程序可调用 SysTick_Config 函数对 SysTick 进行配置和管理。

5.3 Cortex-M3/M4 的编程模型

Cortex-M3/M4 处理器的编程模型（Programmer's model）与经典 ARM 处理器有较大差别。例如，在以 ARM7T 系列为代表的经典 ARM 处理器中，具有 ARM 和 Thumb 两种操作状态，在不同的状态下有各自不同的指令系统。两种状态可以切换，但是状态切换会有一定的时间开销。而 Cortex-M3/M4 支持 16 位和 32 位指令并存的 Thumb-2 技术，没有 ARM 和 Thumb 两种操作状态之分，不再有状态切换所导致的时间开销。经典 ARM 处理器的运行模式有 7 种之多，如表 5.8 所示，Cortex-M3/M4 对这些运行模式进行了化简与合并，使得 Cortex-M3/M4 的操作状态和运行模式的逻辑关系更加清晰。本节将从编程的角度，介绍 Cortex-M3/M4 处理器的操作状态与运行模式、各类寄存器以及堆栈处理机制。

表 5.8 经典 ARM 处理器的运行模式

处理器模式			说明
1、用户模式（usr）			User，正常的程序执行模式，不能直接修改 CPSR 切换到其他模式
特权模式	2、系统（sys）		System，运行具有特权的操作系统任务，可直接切换到其他模式
	异常模式	3、管理（svc）	Supervisor，操作系统使用的保护模式，复位和软件中断进入该模式
		4、终止（abt）	Abort，当数据或指令预取遇到终止时进入该模式
		5、未定义指令（und）	Undefined，当执行未定义或不支持的指令时进入该模式
		6、中断（irq）	Interrupt request，通用外部中断处理，IRQ 异常响应时进入该模式
		7、快速中断（fiq）	Fast Interrupt request，快速中断请求处理，FIQ 异常响应时进入该模式

5.3.1 操作状态与操作模式

Cortex-M3/M4 处理器只有两种操作状态和两个操作模式，另外还有特权和非特权两种访问等级。

1. 操作状态

Cortex-M3/M4 处理器具有以下两种操作状态。

1） Thumb 状态

若处理器执行 Thumb 指令程序代码，就会处于 Thumb 状态。由于 Cortex-M 系列处理器不支持 ARM 指令集，所以没有经典处理器中的 ARM 状态。

2） 调试状态

当处理器被暂停后，例如通过调试器发布暂停命令或触发程序断点后，就会进入调试状态，并停止指令执行。

调试状态仅用于调试操作，可以通过两种方式进入调试状态，调试器发起暂停请求，或处理器中调试部件产生调试事件。在调试状态下，调试器可以访问或修改处理器中寄存器数值。无论在 Thumb 状态还是调试状态下，调试器都可以访问系统存储器，包括位于处理器片内和片外的各种外设。

2. 操作模式

操作模式也称为处理器工作模式或者运行模式，Cortex-M3/M4 处理器将经典 ARM 处理器的 7 种运行模式归并成以下两种操作模式。

1） 处理模式（Handler mode）

也就是异常处理模式。在该模式下执行的是异常/中断服务程序（ISR）。处理模式相当于将经典处理器中的 5 种异常模式整合为一种模式。

2） 线程模式（Thread mode）

除了处理模式以外的所有运行模式。

3. 访问等级

在 Cortex-M3/M4 处理器中，有特权和非特权两种访问等级。两种访问等级可以访问的寄存器类型和存储区域存在差异。

如果处理器处于处理模式，正在执行异常/中断服务程序，此时具有特权访问等级，可以访问处理器中的所有资源。

而线程模式则根据特权访问等级分为两种类型，第一种是具有特权访问等级的线程模式，简称为特权线程模式；第二种是只有非特权访问等级的线程模式，简称非特权线程模式。如果与经典 ARM 处理器进行类比，特权线程模式相当于系统（sys）模式所具有的访问权限；非特权线程模式则类似于用户模式（usr）所具有的访问权限。

Cortex-M3/M4 处理器操作状态与操作模式之间的迁移关系可由图 5.25 表示。

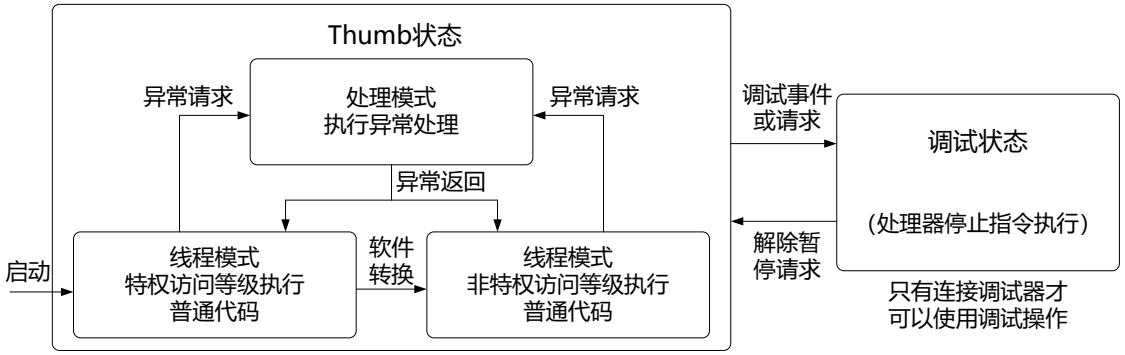

图 5.25 Cortex-M3/M4 处理器的操作状态与操作模式

Cortex-M 系列处理器启动后默认处于 Thumb 状态以及特权线程模式。当执行普通应用程序代码时，处理器可以处于特权线程模式，也可以处于非特权线程模式。处于线程模式下，处理器所具有的特权访问等级由特殊寄存器 CONTROL 控制。关于 Cortex-M3/M4 的特殊寄存器将在本节稍后部分介绍。在特权线程模式下，通过对 CONTROL 寄存器的写操作，可以将处理器从特权线程模式切换到非特权线程模式。但是在非特权线程模式下，处理器无法访问 CONTRL 寄存器，因此也无法从非特权等级切换到特权等级。如若需要进行切换，只能借助异常机制才可实现。

非特权访问等级与特权访问等级的编程模型基本一致，仅在以下几个方面存在差异：

- ❒ 有几条指令只有特权访问等级才能执行，使用这些指令将引起用法错误（Usage Fault）异常；
- ❒ 非特权访问等级不能访问大部分的内核私有区域，也不能访问某些特殊寄存器，例如，Cortex-M3/M4 处理器所有的 NVIC 寄存器仅支持特权访问等级；
- ❒ 如果系统中配置了 MPU，并且 MPU 划定某些区域只允许特权等级访问，非特权等级也不能访问这些区域，否则将引起 MemManage Fault 异常。

线程模式和处理模式的编程模型也很类似。不过线程可以使用进程栈指针 PSP，也可以选择使用主栈指针 MSP，而处理模式只能使用 MSP，这种设计使得应用任务的栈空间和操作系统的主栈空间相互独立，提高了系统可靠性。但是对于许多简单的应用，一般不会使用非特权线程模式和进程栈指针。

特权和非特权访问等级是一种最基本的安全模型，系统设计者可以借此对关键区域提供必要的保护机制，以提高嵌入式应用系统的健壮性。例如，系统运行的程序可能既有具有特权访问等级的嵌入式操作系统，又有非特权访问等级的应用任务。可以通过特权等级划分限制应用程序可访问区域，避免不可靠应用任务破坏操作系统内核，或者破坏其他任务使用的存储区域和外设。如果某个应用程序出现崩溃，不至于影响到操作系统内核和其他应用任务继续运行。

基于 ARMv6-M 版本架构的 Cortex-M0 处理器不支持非特权线程模式，而 Cortex-M0+处理器只是将其作为一个可选项。

5.3.2 常规寄存器

在经典 ARM 处理器（如 ARM7TDMI）中，虽然程序可访问的寄存器总数达 30 多个，但在不同的状态和不同的工作模式下只能看到其中的一部分，如表 5.9 所示。例如，在 ARM7TDMI 中，在 usr 和 sys 模式下只能看到其中的 17 个，其他模式可以看到 18 个。

表 5.9 经典 ARM 处理器中的寄存器

寄存器 \ 模式	usr	sys	svc	abt	und	irq	fiq
通用寄存器	R0						
	R1						
	R2						
	R3						
	R4						
	R5						
	R6						
	R7						
	R8						R8-fiq

	R9					R9-fiq
	R10					R10-fiq
	R11					R11-fiq
	R12					R12-fiq
	R13（SP）	R13_svc	R13_abt	R13_und	R13_irq	R13-fiq
	R14（LR）	R14_svc	R14_abt	R14_und	R14_irq	R14-fiq
程序计数器	R15（PC）					
状态寄存器	CPSR					
	无	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

与经典处理器相比，Cortex-M3/M4 处理器将工作模式的数量从 7 种减为 2 种。同时，在力求与经典处理器保持相对兼容的前提下，Cortex-M3/M4 处理器对各类寄存器也进行了调整和梳理，并重新进行了分组，使得各类寄存器的组织结构更加清晰，功能更加全面。

Cortex-M3/M4 处理器中的常规寄存器组中共有 16 个寄存器，其中 13 个为 32 位通用寄存器，其他 3 个分别为堆栈指针、链接寄存器和程序计数器。与经典处理器相比，Cortex-M3/M4 处理器的常规寄存器少了 CPSR 和 SPSR，而是改用一种新程序状态信息保护机制，本小节稍后将介绍新的状态寄存器。Cortex-M3/M4 处理器各个寄存器名称如图 5.26 所示。

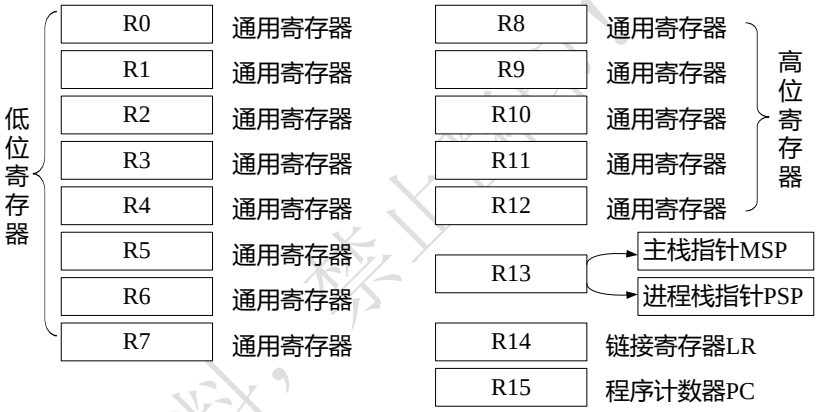

图 5.26 Cortex-M3/M4 处理器中的寄存器

1. R0~R12 通用寄存器

编号 R0~R12 的寄存器为通用寄存器，其中前 8 个（R0~R7）也被称作低位寄存器。由于受指令编码空间的限制，许多 16 位 Thumb 指令只能访问低位寄存器。R8~R12 称为高位寄存器，可用于 32 位指令和少数几个 16 位指令（如 MOV 指令）。系统复位之后，R0~R12 的初始值均未定义。

虽然 R0~R12 都是通用寄存器，但是为了能够实现汇编程序与 C 语言程序的相互调用，ARM 公司制定了 AAPCS[10]规范，规定 R0~R3 用于子程序之间的参数传递，R4~R11 用于保存子程序的局部变量，R12 作为子程序调用的中间寄存器。因此，在编写需要进行参数传递的子程序时，应注意遵守上述寄存器使用规则。关于 ARM 汇编语言编程请参见本书第七章相关内容。

2. R13 栈指针

R13 为堆栈指针。Cortex-M3/M4 处理器采用了双堆栈设计，有两个物理栈指针，也就是

[10] ARM Architecture Procedure Call Standard，ARM 公司制定的子程序调用规范（标准），包括寄存器使用规则、数据栈使用规则以及参数传递规则。

说实际上有两个 R13 寄存器，一个是主栈指针 MSP，另一个是进程栈指针 PSP。但是对于一般程序而言，两个堆栈指针寄存器只有一个可见。其中 MSP 为默认栈指针，在系统复位后或处理器处于处理模式时，处理器使用 MSP，而 PSP 只能用于线程模式。栈指针的选择是通过特殊寄存器 CONTROL 实现的。

在大多情况下，对于不需要嵌入式操作系统的应用，没有必要使用 PSP，许多简单应用只需使用 MSP 即可。系统复位之后，PSP 的初值未定义，而 MSP 的初值存放在整个存储空间的第一个字中（参见 5.2 节表 5.7），在系统初始化时，需要将其取出并对 MSP 进行赋值。

需要注意的是，尽管 v8 版架构之前的 ARM 处理器都是 32 位，PUSH 和 POP 也是以字为单位进行操作，在理论上堆栈采用字（4 字节）对齐方式即可。但是考虑到 64 位双精度浮点数的压栈问题，AAPCS 规范要求堆栈应该采用双字（8 字节）对齐方式。如果没有遵守 AAPCS 的规范，在发生函数调用时可能会因对齐问题出现难以预料的错误。在 Cortex-M3/M4 处理器中，通过系统控制块 SCB（System Control Block）中的配置控制寄存器 CCR（Configuration Control Register），可以使能或禁止双字栈对齐特性。

无论是采用字对齐还是双字对齐，MSP 和 PSP 的最低两位总是为 00，对这两位进行写操作不起作用。

3. R14 链接寄存器

R14 也被称作链接寄存器（Link Register，LR），用于保存函数或者子程序调用时的返回地址。当函数或子程序运行结束时，可以将 LR 中的数值加载到程序计数器 PC 中，返回到调用程序处并继续执行。

在异常/中断处理时，LR 将自动保存一个名为 EXC_RETURN 的特殊值，指示异常/中断处理结束时的返回行为。如果是嵌套（被调用的函数或者子程序又调用另外一个函数或子程序）调用，调用前需要将 LR 中的数值压栈保存，否则在嵌套调用之后 LR 原来保存的信息丢失从而无法正常返回主程序。本书第 5.5 节将对异常/中断处理过程做更加深入的介绍。

由于 Cortex-M3/M4 中有些指令字长为 16 位，指令有时会对齐到半字地址，即便如此，返回地址总是偶数，LR 的最低位为 0。不过 LR 的第 0 位是可读可写的，有些转移/调用操作需要将 LR 的第 0 位置 1，用以表示处于 Thumb 状态。但是无论 LR 的最低位是什么数值，作为返回地址来说，总是默认其为 0。

4. R15 程序计数器

R15 为程序计数器 PC，对于 ARM 处理器，PC 不仅可读而且可写，读操作返回的是当前指令地址加 4，这是因为流水线的特性以及与 ARM7T 系列处理器兼容的需要。对 PC 的写操作可以使用 MOVE 指令以及数据处理指令，以实现程序的跳转。而在 Intel 8086 处理器中，不允许对代码段寄存器 CS 以及指令指针 IP 进行写操作，CS 和 IP 也不能作为数据传送和数据处理指令的目的操作数，程序转移或者过程调用只能通过专门的指令来实现。

在 Cortex-M3/M4 处理器中，由于指令必须要对齐到半字或字地址，作为代码地址，PC 的最低位总是认为是 0。但是，无论是使用跳转指令还是直接写 PC 寄存器，写入值必须是奇数，确保其最低位是 1，以表示其处于 Thumb 状态，否则将被认为试图转入 ARM 模式，从而导致出现错误异常。如果使用高级编程语言（包括 C 和 C++），编译器会自动将跳转目标的最低位置 1，无需担心出错。PC 最低位为 1，只是表示其处于 Thumb 状态，作为指令地址，PC 最低位始终被认为是 0。

在多数情况下，跳转和调用操作都由专门指令实现，利用数据传输和数据处理指令更新 PC 的情况较为少见。在访问位于程序存储器中的字符数据时，经常将 PC 作为基地址寄存器，而偏移地址由指令中的立即数给出。

5. 关于寄存器的名称

使用 ARM 汇编指令编程时，在汇编代码中出现的上述寄存器可以使用不同的名称，如大写、小写或者大小写混用，如表 5.10 所示。常用的汇编工具（如 Keil MDK-ARM 和 ARM ADS）都能识别。

表 5.10 汇编代码中可以使用的寄存器名称

寄存器	可以使用的寄存器名称	备注
R0 ~ R12	R0、R1...R12、r0、r1...r12	
R13	R13、r13、SP、sp、Sp	MSP 和 PSP 只能用于特殊寄存器访问指令 MRS 和 MSR
R14	R14、r14、LR、lr、Lr	
R15	R15、r15、PC、pc、Pc	

5.3.3 特殊寄存器

Cortex-M3/M3 处理器中除了上述寄存器之外，还有几个特殊寄存器，如图 5.27 所示。这些寄存器用于表示处理器状态、定义处理器操作状态以及设置异常/中断屏蔽。在使用 C 或 C++ 等高级编程语言开发简单应用时，可能不太需要这些特殊寄存器。但在编写需要在嵌入式操作系统环境下运行的应用程序，或者需要使用高级中断屏蔽特性时，就需要访问这些特殊寄存器。

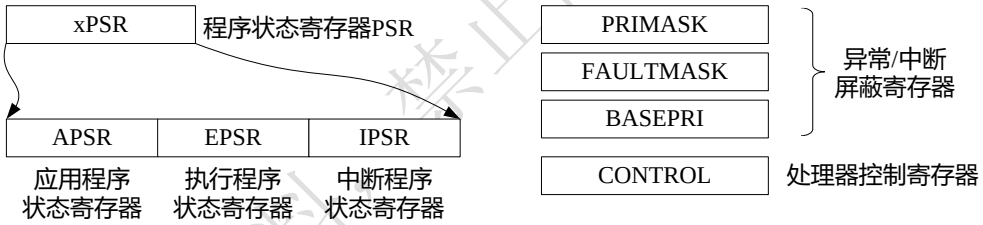

图 5.27 Cortex-M3/M3 处理器中的特殊寄存器

特殊寄存器没有经过存储器映射，只能利用 MSR/MRS 指令通过寄存器的名称对其进行访问。具体格式为：

```
MRS <reg>,  <special_reg>        ；将特殊寄存器读入某个通用寄存器 reg
MSR <special_reg>,  <reg>        ；将通用寄存器 reg 的内容写入特殊寄存器
```

CMSIS-core 也提供了几个用于访问特殊寄存器的函数。但应注意不要将特殊寄存器与其他 MCU 中的“特殊功能寄存器（SFR）”混淆，后者一般用于 I/O 控制。

1. 程序状态寄存器

以 ARM7TDMI 为代表的经典 ARM 处理器中，采用了 CPSR 和 SPSR 两个程序状态寄存器，其中，CPSR 用于保存当前程序状态，SPSR 用于异常/中断处理时保存 CPSR 的状态，以便异常返回后能够恢复处理器的工作状态。但在出现中断嵌套时，SPSR 的内容还必须使用堆栈保存，否则将会丢失。

从 ARMv7 版架构开始，ARM 采用新的程序状态寄存器 PSR 替代 CPSR，并且取消了 SPSR，在出现异常时，使用 L14（LR）寄存器或者堆栈（出现中断嵌套时）保存 PSR 的内容。

PSR 实际上包含了 APSR（应用程序状态寄存器）、EPSR（执行程序状态寄存器）和 IPSR

（中断程序状态寄存器）三个寄存器，如图 5.27 所示。读取 PSR 的结果包含了 APSR、EPSR 和 IPSR 三个寄存器的内容。所以 PSR 又被称为 xPSR。

例如，使用如下 ARM 汇编指令可以读写组合程序状态字，所读到的 PSR 内容如表 5.11 所示。表中各个标志位的含义参见表 5.12。

```
MRS     r0,  PSR      ；读组合程序状态字到 R0 寄存器
MSR     PSR, r0       ；将 R0 寄存器的内容写入组合程序状态字
```

表 5.11 PSR 组合状态字与 APSR、EPSR 和 IPSR

	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[3:0]*							
EPSR						ICI/IT	T			ICI/IT						
IPSR												异常/中断编号				
PSR	N	Z	C	V	Q	ICI/IT	T		GE[3:0]*	ICI/IT		异常/中断编号				

*：仅 Cortex-M4 处理器中有 GE[3:0]，Cortex-M3 没有此位域；

除了上述整体访问方式以外，使用下列 ARM 汇编指令可以单独访问 APSR 和 IPSR 两个状态寄存器，但是 IPSR 只能读出，写入操作对其无效。而 EPSR 不能使用 MRS（读出全是 0）和 MSR 指令对其进行单独访问，只能作为 PSR 中的一个组成部分被整体访问。

例 5.1 读写程序状态字

```
MRS     r1,  APSR     ；读应用程序状态字到 R1 寄存器
MRS     r2,  IPSR     ；读当前正在处理的中断类型号 R2 寄存器
MSR     APSR, r0      ；将 R0 寄存器的内容写入应用程序状态寄存器
```

表 5.12 Cortex-M3/M4 处理器程序状态字各位的含义

位	描 述
N	N=1，结果为负
Z	Z=1，结果为零
C	C=1，出现进位
V	V=1，出现溢出
Q	Q=1，出现饱和。基于 ARMv6-M 版架构的 Cortex-M0/M0+/M1 不存在此位
GE[3:0]	大于或等于标志，分别对应每个字节的数据通路，ARMv6-M 版架构以及 Cortex-M3 中没有
ICI/IT	Interrupt-Continuable Instruction/ IF-THEN 指令状态标志位，用于指令条件执行，ARMv6-M 无
T	Thumb 指令标志。由于总是处于 Thumb 状态，所以该位总是 1，清除此位会引起错误异常
异常编号	表示正在处理的异常/中断编号，是新的异常/中断能否实现抢占（嵌套）的主要依据

表 5.12 所示 APSR 和 EPSR 的某些位域在 ARMv6-M 版架构（如 Cortex-M0）中不可用，甚至在 Cortex-M3 中也没有。ARMv7 版结构的程序状态寄存器与经典 ARM 处理器已有很大差异。若将 Cortex-M3/M3 处理器的 PSR 与 ARM7TDMI 的 CPSR 进行比较，就会发现 ARM7TDMI 中反映工作模式的 M[4:0]位已被取消，Thumb 指令标志 T 更换了位置，中断标志 I 和 F 被中断屏蔽寄存器（PRIMASK）所取代。为了便于比较，表 5.13 列出了 Cortex-M3/M4 和经典 ARM 处理器以及 Cortex-A/R 处理器的程序状态字格式。

表 5.13 几种不同架构 ARM 处理器的程序状态字格式

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
Cortex-A/R	N	Z	C	V	Q	IT	J	保留	GE[3:0]	IT	E	A	I	F	T	M[4:0]
ARMv4T	N	Z	C	V	保留								I	F	T	M[4:0]
Cortex-M4	N	Z	C	V	Q	ICI/IT	T		GE[3:0]	ICI/IT		异常编号				
Cortex-M3	N	Z	C	V	Q	ICI/IT	T			ICI/IT		异常编号				
ARMv6-M	N	Z	C	V			T									异常编号

2. PRIMASK、 FAULTMASK 和 BASEPRI 寄存器

PRIMASK、FAULTMASK 和 BASEPRI 三个寄存器用于实现基于优先权等级的异常/中断屏蔽。

在 Cortex-M3/M4 处理器中，复位、NMI 和 HardFault 硬件错误三个系统异常具有固定的优先权等级，其中复位的优先级为-3，具有最高的优先权等级（参见表 5.6）；紧随其后的是 NMI（优先权等级为-2）和硬件错误（优先权等级为-1）。可以看出优先权等级的数值越小，优先级越高；而数值越大，优先级越低。

除了上述三个具有固定优先级的系统异常之外，可以通过对 NVIC 相关寄存器的设置，为其他所有异常/中断都分配一个优先权等级，优先权等级的数值范围从 0~255。

当每个异常/中断都有了优先权之后，如何通过 PRIMASK、FAULTMASK 和 BASEPRI 这三个寄存器对异常/中断进行屏蔽管理呢？首先让我们了解一下这三个寄存器的编程模型（如图 5.28 所示）。

	31:8	7:1	0
PRIMASK			PRIMASK
FAULTMASK			FAULTMASK
BASEPRI		3~8位可伸缩 ◄——►	5~0位无用

图 5.28 寄存器 PRIMASK、FAULTMASK 和 BASEPRI 的编程模型

从图 5.28 中可以看出，PRIMASK 和 FAULTMASK 寄存器各只有一位。如果 PRIMASK 的最低位被置位（写入 1），则系统将屏蔽除复位、NMI 和硬件错误以外的所有系统异常和外部中断，相当于优先级数值大于等于 0 的所有系统异常/中断均被屏蔽。当处理某些对时间有特殊要求的紧急任务时，往往需要这样做。这种方式有点类似 x86 系统的“关中断”，但是 x86 系统的关中断只能屏蔽外部中断，无法屏蔽系统异常。需要注意的是，当这些任务处理完之后，要将 PRIMASK 的最低位进行复位（清零），以便恢复对异常/中断的处理，此操作类似于 x86 系统的“开中断”。

FAULTMASK 与 PRIMASK 类似，如果将 FAULTMASK 的最低位置 1，硬件错误异常也被屏蔽，相当于把异常/中断的屏蔽门槛提高到优先级“-1”。这在执行负责错误处理的中断服务程序中较为常用。因为既然出现了错误并且正在处理，就无需理会此刻出现的包括硬件错误在内的其他异常，以便错误处理程序能够“专心致志”地进行错误修复。与 PRIMASK 不同的是，FAULTMASK 无需清理，当负责错误处理的异常处理程序返回时，会自动复位 FAULTMASK。但是，ARMv6-M 中没有 FAULTMASK 寄存器。

为使异常/中断管理功能更加灵活和细腻，ARMv7-M 版架构增加了 BASEPRI 寄存器（ARMv6-M 没有），可以按照具体优先数值对中断屏蔽进行管理。BASEPRI 寄存器的最低 8 位采用了如图 5.28 所示的可伸缩设计，其具体宽度取决于芯片制造商实际配置的中断优先级数量。芯片制造商配置的中断优先级至少为 8 级，此时 BASEPRI 中 7:5 这 3 位用于设置中断屏蔽；如果配置了 256 级中断，则 BASEPRI 中 7:0 共有 8 位用于设置中断屏蔽；如果实际配置了 32 级中断，BASEPRI 的宽度为 7:3 共有 5 位。假如 7:3 这 5 位的数值是 0b1 0000，将屏蔽优先级数值大于等于 0b1 0000 的所有中断。但是，如果 7:3 这 5 位的数值是 0b0 0000，其含义则是对所有中断都不屏蔽。如果需要屏蔽所有优先级数值大于等于 0 的中断，应该使用 PRIMASK 寄存器。

显然，只有特权访问等级才可以对这些特殊寄存器进行读写访问，非特权状态下的写操作会被忽略，而读操作返回数值为 0。具有特权访问级别时，可以使用如下的汇编指令访问这些异常/中断屏蔽寄存器。

```
MRS     r0,  BASEPRI          ; 将 BASEPRI 寄存器读入 R0
MRS     r0,  PRIMASK          ; 将 PRIMASK 寄存器读入 R0
MRS     r0,  FAULTMASK        ; 将 FAULTMASK 寄存器读入 R0
MSR     BASEPRI,  r0          ; 将 R0 写入 BASEPRI 寄存器
MSR     PRIMASK,  r0          ; 将 R0 写入 PRIMASK 寄存器
MSR     FAULTMASK,  r0        ; 将 R0 写入 FAULTMASK 寄存器
```

例 5.2 关中断命令

```
MOV     r0,  #1
MSR     PRIMASK,     R0
```

例 5.3 开中断命令

```
MOV     R0,  #0
MSR     PRIMASK,     r0
```

例 5.4 假设系统中共有 16 级中断，BASEPRI 寄存器的宽度为 7:4 共 4 位，现在需要屏蔽优先级大于等于 5 的所有中断。

```
MOV     R0,  #0b101
MSR     BASEPRI, R0
```

例 5.5 取消 BASEPRI 寄存器的中断屏蔽设置

```
MOV     R0,  #0
MSR     BASEPRI, R0
```

在特权访问等级下，也可以使用 CMSIS-Core 提供的多个 C 语言函数访问这三个屏蔽寄存器，例如：

```
x = _get_BASEPRI() ;          //读 BASEPRI 寄存器
x =_get_PRIMARK();            //读 PRIMASK 寄存器
x =_get_FAULTMASK();          //读 EAULTMASK 寄存器
_set_BASEPRI(x);              //设置 BASEPRI 的新数值
_set_PRIMASK(x);              //置位 PRIMASK
_set_FAULTMASK(x);            //置位 FAULTMASK
_disable_irq();               // 置位 PRIMASK，禁止异常/中断，相当于
_set_PRIMASK(1)
_enable_irq();                // 清除 PRIMASK，使能异常/中断，相当于
_set_PRIMASK(0)
```

此外，利用修改处理器状态 CPS 指令，也可以很方便地设置或清除 PRIMASK 和 FAULTMASK 的数值。例如：

```
CPSIE i         ;使能异常/中断（清除 PRIMASK 位）
CPSID i         ;禁止异常/中断（置位 PRIMASK 位）
CPSIE f         ;使能异常/中断（清除 FAULTMASK 位）
CPSID f         ;禁止异常/中断（置位 FAULTMASK 位)
```

3. CONTROL 寄存器

CONTROL 寄存器用于选择线程模式的特权访问等级以及栈指针。对于带有 FPU 的 Cortex-M4 处理器，还有一位用于指示当前正在执行的代码是否使用 FPU。CONTROL 寄存器的编程模型如表 5.14 所示。为了便于比较，表 5.14 中还给出了 Cortex-M0 和 M0+的 CONTROL 寄存器结构。

表 5.14 几款 Cortex-M 系列处理器的 CONTROL 寄存器

	31:3	2	1	0
Cortex-M3/M4			SPSEL	nPRIV
具有 FPU 的 Cortex-M4		FPCA	SPSEL	nPRIV
Cortex-M0			SPSEL	
Cortex-M0+			SPSEL	nPRIV 可选

表 5.14 中的各位含义如下：

- nPRIV，设置处理器线程模式下的特权访问等级。若该位为 0，处理器进入特权线程模式；若该位为 1，处理器则处于非特权线程模式。
- SPSE，选择线程模式下所使用的栈指针类型。当该位为 0 时，线程模式使用主栈指针 MSP；当该位为 1 时，线程模式使用进程栈指针 PSP；而在处理模式下，该位始终为 0，并且忽略对其所做的写操作。
- FPCA，只有选配了 FPU 的 Cortex-M4 中才有此位。当发生异常时，异常处理程序利用该位确定浮点单元中的寄存器是否需要压栈保存。当该位为 1 时，如果当前代码使用的是浮点指令，则需要压栈保存浮点寄存器。执行浮点指令时 FPCA 位自动置位，在异常入口处该位被硬件自动清除。

复位后，CONTROL 寄存器被恢复成默认值 0。这意味着在复位之后，处理器处于线程模式、具有特权访问等级并且使用主栈指针。在特权线程模式下，可以通过写 CONTROL 寄存器进入非特权线程模式。不过，当 CONTROL 寄存器的最低位 nPRIV 位被置位后，非特权线程模式下不能继续访问 CONTROL 寄存器了，也无法再切换回特权线程模式了。

上述机制可以为嵌入式系统提供一个简单的基本安全保护。例如，嵌入式系统中可能会运行一些不受信任的应用程序，应该对这些程序的权限进行限制，防止不可靠程序对系统可能造成的损害。

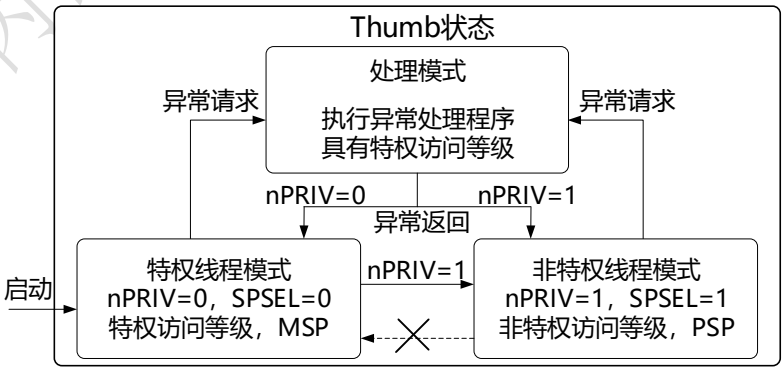

图 5.29 特权访问等级和栈指针选择的状态迁移图

如果需要将非特权线程模式切换回特权线程模式，则必须通过异常处理机制才能实现。在异常处理时，处理器处于处理模式并具有特权访问等级，可以清除 nPRIV 位，再从异常返回线程模式后（如图 5.30 所示），处理器就会进入特权线程模式。

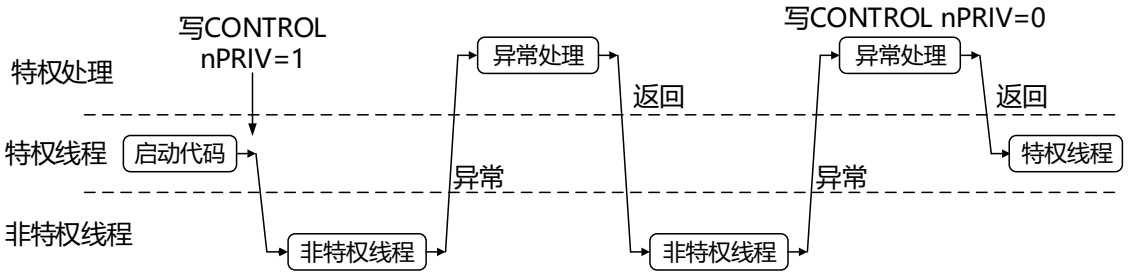

图 5.30 特权线程模式与非特权线程模式之间的切换

大多数简单应用都没有使用嵌入式操作系统，在这种情形下无须修改 CONTROL 寄存器的数值，整个应用可以一直运行在特权线程模式下，并且只使用 MSP，如图所示。

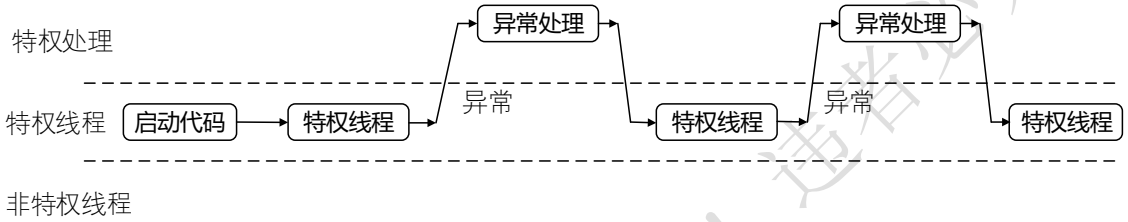

图 5.31 简单应用不需要非特权线程模式

除了可使用汇编指令 MRS/MSR 对 CONTROL 寄存器进行读写之外，CMSIS-Core 也提供了 CONTROL 的 C 语言访问函数。

```
x =_get_CONTROL();          //读取 CONTROL 寄存器的当前值
_set_CONTROL(x);            //设置 CONTROL 寄存器的数值为 x
```

4. 系统控制块 SCB

系统控制块 SCB（System Control Block）是由多个寄存器组成的一个数据结构，属于 Cortex-M3/M4 内核的一部分并被综合在 NVIC 中。SCB 中各类寄存器的作用包括：

- ❒ 对处理器进行配置，如之前所述的双字栈对齐使能控制，以及低功耗模式设置等；
- ❒ 提供错误状态信息，SCB 中包含了多个反映不同类型错误的状态寄存器；
- ❒ 异常/中断管理，例如异常的挂起与解挂控制、优先级设置、优先级分组以及中断向量表的重定位；
- ❒ 反映处理器特性、指令集特性、存储模块特性以及调试特性（只读寄存器）。

对于配置了 FPU 的 Cortex-M4 处理器，还有一个协处理器访问控制寄存器。SCB 中的各类寄存器与 NVIC 中的其他寄存器相同，都被映射在系统控制空间 SCS 中，其地址范围为 0xE000 ED00 ~ 0xE000 ED88。如果使用汇编语言，只能通过地址对这些寄存器进行访问。不过 CMSIS-Core 为每个寄存器分配了一个符号，类似于将这些寄存器都赋予一个名称。例如，地址为 0xE000 ED04 的中断控制和状态寄存器（ICSR），CMSIS-Core 的符号为“SCB->ICSR”；地址为 0xE000 ED10 的系统控制寄存器（SCR），CMSIS-Core 的符号为“SCB->SCR”。使用 C 语言并利用 CMSIS-Core 定义的这些符号，可以更方便地对 SCB 中的各个寄存器进行访问。

SCB 中与异常/中断管理有关的寄存器将在 5.5 节详细介绍。

5.3.4 堆栈结构

与几乎所有的处理器相同，Cortex-M3/M4 处理器也需要使用堆栈和堆栈指针 SP。并且为了提高系统的健壮性，Cortex-M3/M4 处理器采用了双堆栈结构，分别使用了 MSP 和 PSP 两个堆栈指针。

1. 堆栈的作用和堆栈类型

堆栈是一种特殊的数据结构，是一种只能在一端进行插入和删除操作的线型表。堆栈的数据存取操作按照“后进先出（LIFO）”的原则，并通过堆栈指针指示当前的操作位置。使用压栈指令 PUSH 向堆栈中增加数据，使用出栈指令 POP 指令从堆栈中提取数据。每次 PUSH 和 POP 操作后，当前使用的堆栈指针都会自动进行调整。

ARM 处理器的堆栈可用于：

- ❒ 在异常/中断响应时，保存被中断程序的下一条指令的地址（断点），以及处理器状态寄存器的内容，以便在异常/中断返回之后处理器能够从断点处继续运行；
- ❒ 异常/中断服务程序，或者正在执行的函数或者子程序如果需要使用某些寄存器，可以使用堆栈保存这些寄存器原来的内容（保存现场），以便异常返回或者函数或子程序处理结束时可以恢复现场；
- ❒ 实现主程序与函数或者与子程序之间的参数传递；
- ❒ 用于存储局部变量。

按照堆栈区域在存储器中的地址增长方向，可以分为递增栈（Ascending Stack）和递减栈（Descending Stack）两种。所谓递增栈是指向堆栈写入数据时，堆栈区由低地址向高地址生长；而递减堆栈是指向堆栈写入数据时，堆栈区是由高地址向低地址生长。

按照堆栈指针 SP 所指示的位置，堆栈又可以分为满堆栈（Full Stack）和空堆栈（Empty Stack）两种。所谓满堆栈是指堆栈指针 SP 始终指向栈顶元素，也就是指向堆栈最后一个已使用的地址（满位置）。而空堆栈的 SP 始终指向下一个将要放入元素的位置，也就是指向堆栈的第一个没有使用的地址（空位置）。

组合上述两种地址增长方向和两种堆栈指针指示位置，可以得到 4 种基本的堆栈类型：

- ❒ 满递增（FA）：SP 指向最后压入的数据，且由低地址向高地址生长。
- ❒ 满递减（FD）：SP 指向最后压入的数据，且由高地址向低地址生长。
- ❒ 空递增（EA）：SP 指向下一个可用空位置，且由低地址向高地址生长。
- ❒ 空递减（ED）：SP 指向下一个可用空位置，且由高地址向低地址生长。

2. Cortex-M 系列处理器的堆栈模型

许多经典 ARM 处理器以及 ARM T32 指令集可以支持以上四种堆栈类型，但在 Cortex-M 系列处理器只能使用满递减类型。处理器在启动或者复位后，系统初始化程序从位于系统 CODE 区的 flash 中，取出地址为 0x0000 0000 的第一个字，作为主栈指针 MSP 的初始值。以后每次 PUSH 操作时，处理器首先移动 SP（SP 的值减去 4），然后将需要压栈保存的数据存储在 SP 指向的存储器位置，如图 5.32 所示。

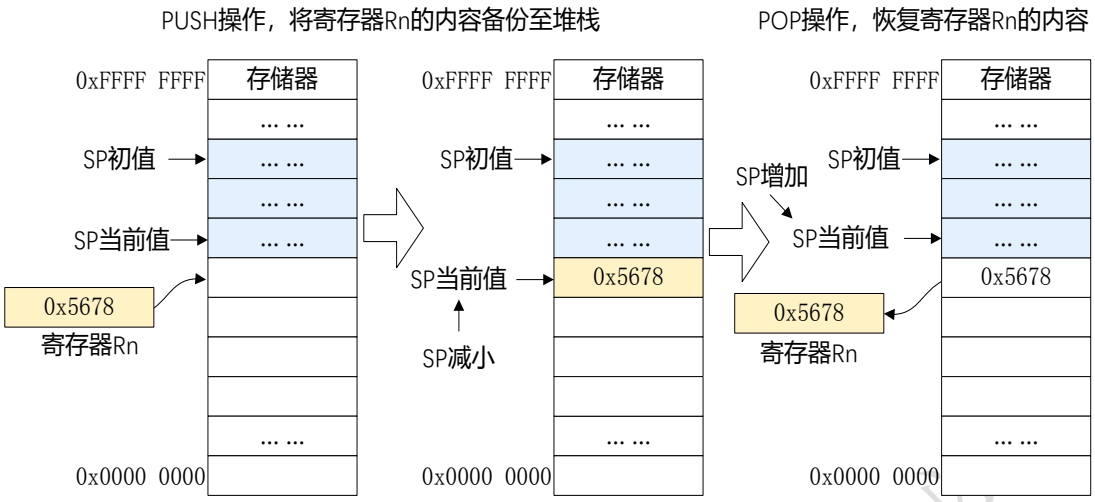

图 5.32 满递减堆栈的 PUSH 和 POP 操作

对于 POP 操作，SP 所指向的堆栈数据被读出，然后 SP 的数值自动加 4，指向 POP 之后新的栈顶位置。从图 5.32 还可以看出，POP 操作从堆栈读出的数据也可以存放到其他寄存器中，具体存放在何处取决于 POP 指令中的目的操作数，从而实现寄存器之间的数据交换。但在保护现场和恢复现场操作时，POP 指令和 PUSH 指令必须配对使用，保证所恢复的现场与原来一致。另外，在 POP 之后原来存放在堆栈中的数据依然存在，但无需理会，因为随后可能出现的 PUSH 操作将会将其覆盖。更多关于堆栈操作的指令请参见本书第六章相关内容。

如果 Cortex-M3/M4 处理器使能了双字栈对齐模式，当出现异常时，假如压栈操作之后堆栈指针没有对齐到双字边界，处理器会自动插入一个“空”字，强制堆栈对齐在双字边界上。同时，已经入栈保存的 xPSR 寄存器中的第 9 位将被置为 1（参见表 5.11），表示堆栈指针发生过调整。出栈时硬件电路对 xPSR 第 9 位进行检查，若为 1 则说明最后入栈的字是为了对齐双字边界插入的，出栈应将其丢弃。虽然双字对齐可能会造成堆栈空间的一点点浪费，但是为了提高软件的标准化程度以及便于程序之间的相互调用，建议使用双字对齐模式。

3. Cortex-M3/M4 处理器中的双堆栈

基于 ARMv7-M 版本架构的 Cortex-M3/M4 采用了双堆栈设计，并且有 MSP 和 PSP 两个堆栈指针，分别服务于不同的操作模式和特权访问等级。根据 CONTROL 寄存器中 nPRIV 和 SPSEL 位的不同组合，两个堆栈共有如表 5.15 所示的 4 种场景，其中前三种比较常见。

表 5.15 主堆栈和进程栈的应用场景

nPRIV	SPSEL	应用场景
0	0	运行在特权访问等级，使用主堆栈和 MSP。大多数简单应用都选择这种模式。
0	1	具有特权访问等级的线程模式，选择使用进程栈和 PSP，主栈用于操作系统内核以及处理模式。常见于搭载了嵌入式操作系统的应用。
1	1	非特权线程模式，只能使用进程栈和 PSP。
1	0	这种情形只可能出现在处理模式，使用主栈和 MSP，但是只有非特权访问等级。线程模式不会出现这种情况。

Cortex-M 系列处理器堆栈空间一般位于 SRAM 区的系统主存储器中，也可以放置在 CODE 区中通过 D-Code 总线连接 SRAM 上。如果需要使用双堆栈，应该通过 MPU 在上述的 SRAM 中建立两个区域（region），其中一个定义为特权级，该区域的一部分用作主栈存储区；

另一个定义为非特权级，其中一部分用于进程栈存储区。注意到 Cortex-M3/M4 的堆栈是满递减类型，因此两个栈指针的初始值应该是如图 5.33 所示的两个区域的最大地址。

如果按照 AAPCS 要求，堆栈采用双字对齐，主栈和进程栈的栈顶都应该位于双字的边界（8 的整数倍）上，MSP 和 PSP 的最低三位应该是 000。

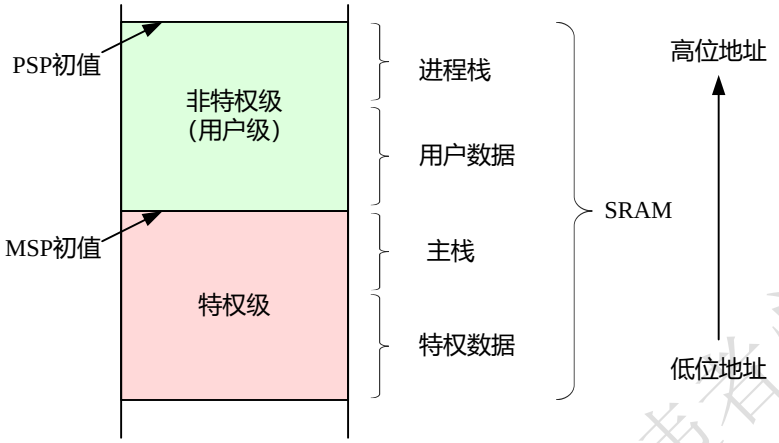

图 5.33 双堆栈的存储结构

用于堆栈的存储区需要事先做好规划，预留足够的空间，以防出现堆栈溢出。MSP 的初始值作为中断向量表的第一项，存放在整个存储空间的第一个字中，该位置通常位于 I-Code 总线所连接的 flash 中。系统上电或者复位之后，系统初始化程序将从中断向量表中取出第一个字完成 MSP 的初始化，但是 PSP 的初始化需要另外编程实现。初始化 PSP 宜使用汇编指令代码，不仅简单高效，而且不易出错。

例 5.6 在特权线程模式下，调用 MPU 设置子程序划分 region，并将进程栈指针设在 SRAM 区的最大位置，然后转到非特权线程模式并启用进程栈。

```
BL      Mpusetup            ; 调用 MPU 设置子程序，建立 region，并使能存储器保护
MOV     r0,  #0x4000 0000   ; 设置进程栈栈顶
MSR     PSP,  r0            ; 初始化进程栈指针
MOV     r0,  #0x3           ; 准备置位 CONTROL 寄存器的 SPSEL 和 nPRIV
MSR     CONTROL, r0         ; 完成 CONTROL 寄存器的更改，切换到非特权线程模式
B       UserAppStart        ; 已进入非特权线程模式，跳转到用户程序入口
```

在以上示例中，BL 是可返回的子程序调用指令，B 是分支指令，两者类似于模型机指令集中（参见第二章表 2.1）的 CALL 指令和 JMP 指令。关于示例中各条汇编指令的完整含义和功能将在本书第六章和第七章详细介绍，这里只是说明创建进程栈的方法和步骤。

5.4 Cortex-M 处理器存储系统

如 5.2.3 小节所述，Cortex-M 处理器可寻址的 4GB 存储器地址空间划分成不同的区域，其地址映射方案如图 5.19 所示。存储空间的不同区域在访问权限、访问属性、总线连接方式、所支持的操作特性等方面均存在差异。本小节将分析上述差异，进行详细说明。

5.4.1 存储器映射

ARM 处理器支持非常灵活的存储器配置，因此基于 Cortex-M3 和 Cortex-M4 的微控制器产品往往具有许多不同的存储器大小和存储器映射。一些微控制器产品中还集成了一些设备相关的存储器特性，如存储器地址重映射。

依据图 5.19 地址映射方式，4GB 可寻址的存储器空间中，有些区域被预定义为片内设备、有些被预定义为片外设备，有些用于代码区，有些用于数据区，还有些用来存储芯片信息。虽然预定义的存储器映射关系是固定的，但整体架构具有高度的灵活性，芯片生产厂家可以在产品中加入不同类型的存储器和外设。

图 5.19 所示各个存储器区域的地址范围和用途说明如表 5.16。

表 5.16 存储器区域的地址范围和用途说明

区域，地址范围	存储器区域的用途
代码 0x0000 0000 ~ 0x1FFF FFFF	512MB 空间，一般连接 Flash 或 ROM，通过 ICode 总线获取指令，通过 Dcode 总线获取数据。代码区主要用于存放程序代码、中断向量表，也可以存放数据。
SRAM 0x2000 0000 ~ 0x3FFF FFFF	512MB 空间，连接 SRAM（一般是片上 SRAM，也可连接其他类型的存储器），主要用于存放数据，也可以存放程序代码。 SRAM 区第一个 1MB 区域支持位段操作，映射至位段别名区域。
片上外设 0x4000 0000 ~ 0x5FFF FFFF	512MB 空间，用于片上外设，连接系统总线。 和 SRAM 区类似，第一个 1MB 区域支持位段操作。
外部 RAM 0x6000 0000 ~ 0x9FFF FFFF	包括两个 512MB 的空间共 1GB，连接片外存储器等其他 RAM，可用于存放程序代码或数据。两个 512MB 空间支持的缓存特性不同。
片外外设 0xA000 0000 ~ 0xDFFF FFFF	包括两个 512M 空间共 1GB，连接片外外设等其他类型存储器。两个 512MB 空间对共享性的支持不同。
系统 0xE000 0000 ~ 0xFFFF FFFF	系统区分为几个部分： ①内部私有外设总线，0xE000 0000 ~ 0xE003 FFFF 用于访问 NVIC，SysTick、MPU 等内置系统部件及调试部件。一般情况下该存储器空间只能由运行在特权等级的代码访问。 ②外部私有外设总线，0xE004 0000 ~ 0xE00F FFFF 用于其他的可选调试部件，芯片厂家可增加独有的调试部件或其他部件。该存储器空间只能由运行在特权等级的代码访问。 ③厂家定义区域，0xE010 0000 ~ 0xFFFF FFFF 用于供应商自定义的部件，多数产品中未使用。

需要执行的程序建议存放在代码区。尽管可以将程序存放在 SRAM 和 RAM 区域，但处理器从这些区域读取指令时需要一个额外的周期。因此，通过系统总线从 SRAM 和 RAM 域执行程序代码时性能会稍微低些。此外，程序不允许在外设区域和系统存储器区域中执行。

图 5.19 所示存储器映射中存在多个内部部件，各个部件的功能描述如表 5.17 所示。

表 5.17 Cortex-M3/M4 存储器映射中私有区域的内置部件

部件	描述
SCS	系统控制空间区域，包括与 NVIC、Systick、FPU 和 MPU 等系统部件相关的寄存器
FPB	Flash 补丁和断点单元，用于调试，包含 8 个服务于硬件断点事件的比较器
DWT	数据监视和跟踪单元，用于调试和跟踪，包含 4 个服务于数据监视点事件的比较器
ITM	指令跟踪宏单元，用于调试和跟踪，软件代码可利用它产生可被跟踪接口捕获的数据跟踪激励
ETM	嵌入式跟踪宏单元，用于产生调试软件可用的指令跟踪
TPIU	跟踪端口接口单元，将跟踪源产生的数据包转换至跟踪接口协议以减少调试引脚数目
ROM 表	调试工具用的查找表，存放调试和跟踪部件的地址，调试工具借此来识别系统中可用的调试部件，此外还提供了系统 ID 寄存器

表 5.17 中系统控制空间（SCS）的具体地址分配和功能描述如表 5.18 所示。本章的后续小节将会介绍这些部件内寄存器组的详细信息。

表 5.18 Cortex-M3/M4 存储器映射的 SCS 区域

部件	地址范围	描述
系统控制和 ID 寄存器	0xE000E000-0xE000E00F	中断控制器类型及辅助控制寄存器
	0xE000ED00-0xE000ED8F	系统控制块 SCB，用于控制处理器行为的寄存器组
	0xE000EDF0-0xE000EEFF	SCS 区调试寄存器
	0xE000EF00-0xE000EF4F	软件触发中断寄存器
	0xE000EF50-0xE000EF8F	缓存和分支预测维护
	0xE000EF90-0xE000EFCF	实现方式定义
	0xE000EFD0-0xE000EFFF	微控制器相关 ID
SysTick	0xE000E010-0xE000E0FF	系统节拍定时器，用于 OS 或用户程序产生周期性的中断
NVIC	0xE000E100-0xE000ECFF	嵌套向量中断控制器，用于异常/中断管理的寄存器组
MPU	0xE000ED90-0xE000EDEF	存储器保护单元，用于设置各存储器区域的访问权限和访问属性

5.4.2 连接存储器和外设

如 5.2 节所述，Cortex-M 处理器主要的总线接口使用 AHB Lite 协议，而私有外设总线使用 APB 协议，各类总线通过总线交换矩阵连接后的总线结构如图 5.20 所示。基于这样的总线结构，Cortex-M3/M4 不同总线与各类存储器及外设的连接方式如图 5.23 所示。

用户可通过 AHB Lite 总线接口连接不同类型的数据存储器。尽管总线宽度固定为 32 位，使用适当的转换硬件后，也可以连接其他宽度的存储器（如 8 位、16 位、64 位及 128 位）。虽然使用了 SRAM 和 RAM 等作为存储器区域的名称，但可连接的存储器的种类是没有限制的，如，可以是 SDRAM 或 DDR DRAM。

同时，连接到代码区的程序存储器的类型也没有限制。如，程序代码可以位于 Flash 存储器、E^2PROM 或 OTP ROM 等。程序存储器的大小也非常灵活，在一些低成本的 Cortex-M 微控制器中，仅有 8KB 的 Flash 和 4KB 的片上 SRAM。

由于图 5.23 所示 CODE 区的总线矩阵/总线复用器是 ARM 公司提供的选件，一些微控制器供应商常在微控制器芯片内部放置自行设计的 Flash 访问加速器，以提升代码执行的效率。例如，意法半导体（ST Microelectronics）的 STM32F2（基于 Cortex-M3 的微控制器）和 STM32F4

（基于 Cortex-M4 的微控制器）就在 I-Code 和 D-Code 总线接口处实现了 Flash 访问加速器，如图 5.34 所示。有两套缓冲连接到这两条总线上，而且缓冲还对每个总线的访问类型进行了优化。

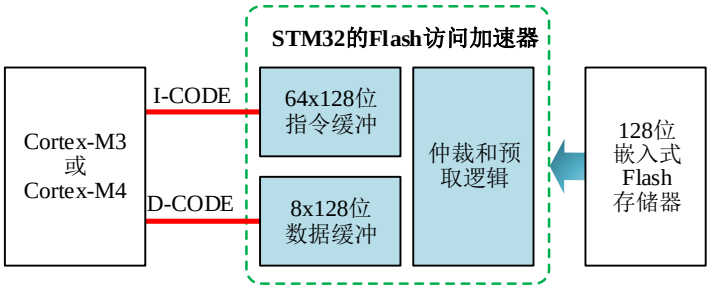

图 5.34 STM32F2 和 STM32F4 的 Flash 访问加速器示意图

AHB Lite 协议仅适用于单主控设备的情形，但很多基于 Cortex-M3 和 Cortex-M4 的微控制器产品中，会出现多个总线主控设备，如 DMA 控制器、以太网控制器或 USB 控制器等。这些产品往往用“总线矩阵”或“多层 AHB”等术语来描述芯片内部总线系统。例如，STM32F4 的总线矩阵就使用了图 5.35 所示的设计，多个总线主设备可以同时访问不同的存储器或外设。

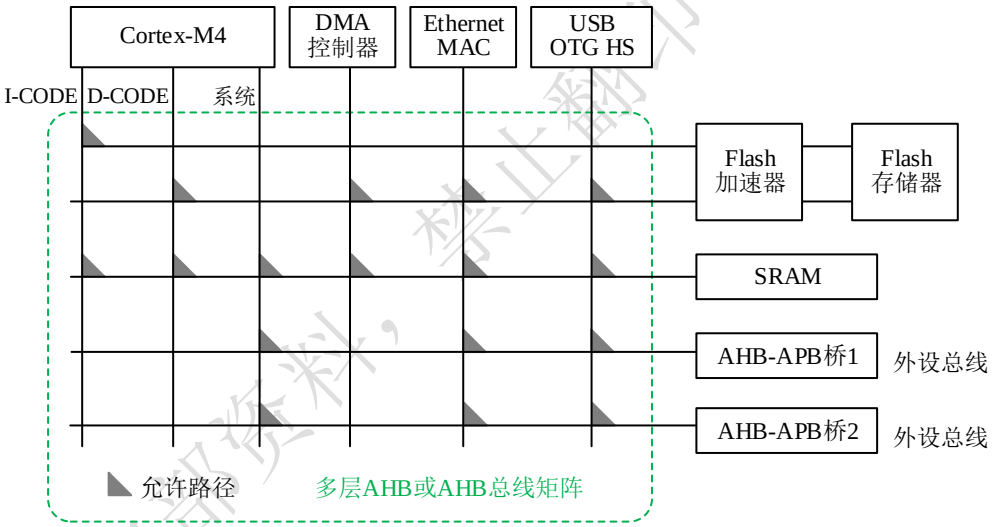

图 5.35 STM32F4 的多层 AHB

图 5.35 所示的多层 AHB 系统中，连接在总线矩阵上的各个从设备可以被不同的总线主设备访问。若出现两个主设备同时访问同一个从设备，则由总线矩阵内部的仲裁逻辑协调高优先级的主设备先访问。

为了支持多种高性能外设与 Cortex-M 处理器的连接，许多微控制器厂家采用了类似上述多层 AHB 的方法。但由于 AHB Lite 协议中并未定义对多主控设备支持的具体实现方式，不同厂家总线接口的实现方式会有差异，通常描述的术语也存在差异。对于较简单的应用场景，软件开发人员无须了解 AHB 操作的细节；但是如果开发高性能的嵌入式应用，开发人员则需要充分了解存储器映射和编程模型。

5.4.3 存储器的端模式

我们在第 2 章 2.2.3 小节中已经学习了大端和小端模式的定义，本小节中结合 Cortex-M

处理器了解更多细节。Cortex-M3/M4 处理器同时支持小端和大端的存储器系统，但不少微控制器厂商把存储器系统设计为只支持小端和大端中的一种。Cortex-M3/M4 处理器在复位时确定存储器系统的端配置；直至下次复位，存储器的端配置都不会改变。

Cortex-M 微控制器多数是小端的，具有小端的存储器系统和外设。对于小端的存储器系统，字数据中四个字节在存储单元中高低字节存放的顺序如表 5.19 所示。

表 5.19 小端存储示例

地址	31~24 位	23~16 位	15~8 位	7~0 位
0x0003 ~ 0x0000	字节 0x0003	字节 0x0002	字节 0x0001	字节 0x0000
0x0007 ~ 0x0004	字节 0x0007	字节 0x0006	字节 0x0005	字节 0x0004
…	…	…	…	…
0x1003 ~ 0x1000	字节 0x1003	字节 0x1002	字节 0x1001	字节 0x1000
0x1007 ~ 0x1004	字节 0x1007	字节 0x1006	字节 0x1005	字节 0x1004
…	…	…	…	…

也可以将 Cortex-M 处理器的存储器系统设计为大端的，此时，字数据中四个字节在存储单元中高低字节存放的顺序如表 5.20 所示。

表 5.20 大端存储示例

地址	31~24 位	23~16 位	15~8 位	7~0 位
0x0003 ~ 0x0000	字节 0x0000	字节 0x0001	字节 0x0002	字节 0x0003
0x0007 ~ 0x0004	字节 0x0004	字节 0x0005	字节 0x0006	字节 0x0007
…	…	…	…	…
0x1003 ~ 0x1000	字节 0x1000	字节 0x1001	字节 0x1002	字节 0x1003
0x1007 ~ 0x1004	字节 0x1004	字节 0x1005	字节 0x1006	字节 0x1007
…	…	…	…	…

Cortex-M3/M4 处理器的大端体系被称作**字节不变大端**（或称为 BE-8）。ARMv6、ARMv6-M、ARMv7 及 ARMv7M 等架构版本支持字节不变大端体系。与 BE-8 不同，在 ARM7TDMI 等经典 ARM 处理器中，使用的大端体系被称作**字不变大端**（或称为 BE-32）。这两种体系在数据传输时字节通道不同，字节通道指高低字节与总线上高低位数据线的映射关系。表 5.21 列出了 BE-8 的 AHB 字节通道，而表 5.22 则是 BE-32 的 AHB 字节通道。

表 5.21 Cortex-M3/M4（字节不变大端）AHB 上的数据

地址，大小	31~24 位	23~16 位	15~8 位	7~0 位
0x1000，字	数据 bit[7:0]	数据 bit[15:8]	数据 bit[23:16]	数据 bit[31:24]
0x1000，半字	—	—	数据 bit[7:0]	数据 bit[15:8]
0x1002，半字	数据 bit[7:0]	数据 bit[15:8]	—	—
0x1000，字节	—	—	—	数据 bit[7:0]
0x1001，字节	—	—	数据 bit[7:0]	—
0x1002，字节	—	数据 bit[7:0]	—	—
0x1003，字节	数据 bit[7:0]	—	—	—

表 5.22 ARM7TDMI（字不变大端）AHB 上的数据

地址，大小	31~24 位	23~16 位	15~8 位	7~0 位
0x1000，字	数据 bit[7:0]	数据 bit[15:8]	数据 bit[23:16]	数据 bit[31:24]
0x1000，半字	数据 bit[7:0]	数据 bit[15:8]	—	—
0x1002，半字	—	—	数据 bit[7:0]	数据 bit[15:8]
0x1000，字节	数据 bit[7:0]	—	—	—

地址，大小	31~24 位	23~16 位	15~8 位	7~0 位
0x1001，字节	—	数据 bit[7:0]	—	—
0x1002，字节	—	—	数据 bit[7:0]	—
0x1003，字节	—	—	—	数据 bit[7:0]

虽然 Cortex-M3/M4 处理器使用的大端系统与经典 ARM 处理器不同，但对于小端系统，Cortex-M3/M4 和经典 ARM 处理器的字节通道都是相同的。如表 5.23 所示。

表 5.23 小端模式下 AHB 上的数据

地址，大小	31~24 位	23~16 位	15~8 位	7~0 位
0x1000，字	数据 bit[31:24]	数据 bit[23:16]	数据 bit[15:8]	数据 bit[7:0]
0x1000，半字	—	—	数据 bit[15:8]	数据 bit[7:0]
0x1002，半字	数据 bit[15:8]	数据 bit[7:0]	—	—
0x1000，字节	—	—	—	数据 bit[7:0]
0x1001，字节	—	—	数据 bit[7:0]	—
0x1002，字节	—	数据 bit[7:0]	—	—
0x1003，字节	数据 bit[7:0]	—	—	—

在 Cortex-M 处理器中：①取指令总是处于小端模式。②对包括系统控制空间（SCS）、调试部件和私有外设总线（PPB）在内的 0xE000 0000 ~ 0xE00F FFFF 区域的访问总是小端的。③若软件应用需要处理大端数据，而所使用的微控制器却是小端的，可以使用 REV、REVSH 和 REV16 等指令将数据在大端和小端间转换。

有些情况下，从一些外设的寄存器中获得的数据，其大小端配置可能会与处理器不同。此时，需要在程序代码中将外设寄存器数据转换为正确的端。

5.4.4 非对准数据的访问

Cortex-M 的存储器系统是 32 位的，但处理器访问或处理的数据的大小可能是 32 位（4 字节，或字），也可能是 16 位（2 字节，或半字）、甚至是八位（字节）。在第 2 章中我们学习了字的对准存放，接下来分析处理器访问对准存放数据/非对准存放数据的具体情形。

对准存放数据可进行对齐传输。对齐传输是指访问存储器时地址值为传输数据大小的整数倍。例如，字大小的对齐传输可以执行的地址为 0x0000 0000、0x0000 0004、…、0x0000 0100 等。类似地，半字大小的对齐传输可以执行的地址则为 0x0000 0000、0x0000 0002、…、0x00000102 等。与对齐传输相反，若访问存储器时地址值不是传输数据大小的整数倍，称为非对齐传输。对齐和非对齐传输的实例如图 5.36 所示。

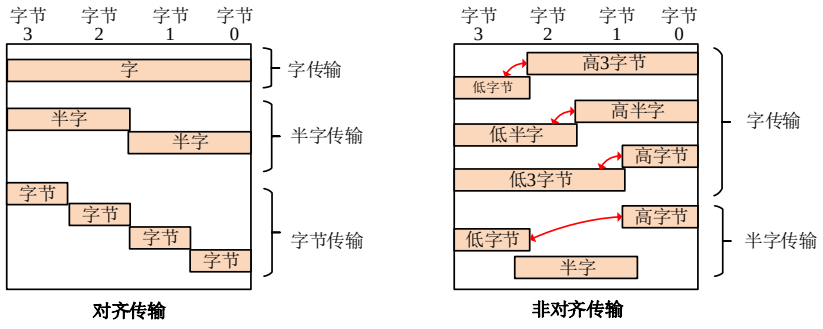

图 5.36 小端存储器系统的对齐访问和非对齐访问

大部分经典 ARM 处理器只允许对齐传输。对齐传输访问存储器时，字地址的 bit[1]和 bit[0]为 0，而半字地址的 bit[0]为 0。例如，字数据的地址可以是 0x2000 或 0x2004，而不能是 0x2001、0x2002 或 0x2003。半字数据的地址可以是 0x2000 或 0x2002，而不能是 0x2001 或 0x2003。另外，所有的字节传输都是对齐的。

对于 Cortex-M3/M4 处理器，大部分存储器访问指令是支持非对齐数据传输的。发生非对齐传输时，数据传输过程会被总线接口电路单元转换为多个对齐传输。这个转换对程序员是不可见的，故而软件开发人员可以不考虑代码所访问的数据是否对准存放。然而，因为非对齐传输会被拆分为几个对齐传输，会导致数据访问花费更多的时间，降低了存储器的访问效率。在高性能应用中，确保数据的对准存放与对齐传输是必要的。

注意，并非所有 Cortex-M3/M4 的存储器访问指令都支持非对齐的数据传输。例如，多加载/存储指令、栈操作指令、排他访问指令必须是对齐；位段操作也不支持非对齐传输。另外，可以设置 Cortex-M3/M4 处理器在非对齐传输出现时触发异常。需要设置的是，配置控制寄存器 CCR 的 UNALIGN_TRP（非对齐陷阱）位。这种设置有利于软件开发过程中，程序员测试程序是否会产生非对齐传输。

5.4.5 位段操作

1. 位段与位段别名

利用位段（Bit-Band，也称作位带）操作，一次存储器操作可以只访问一个位。Cortex-M3 或 Cortex-M4 处理器中，有两个预定义的存储器区域支持这种操作，其中一个位于 SRAM 区域的最低 1MB（0x2000 0000～0x200F FFFF），另一个位于外设区域的最低 1MB（0x4000 0000 ~ 0x400F FFFF）。注意，位段特性在 Cortex-M3 和 Cortex-M4 上是可选的。

这两个区域被称作位段区域，位段区域中的每一个字被映射为一个位段别名（alias）区域。换言之，一个位段别名地址将对应位段区域中特定字的某一个位，SRAM 区域内 1MB 位段区与 32MB 位段别名区域之间的映射关系如图 5.37 所示。图 5.37 中，0x2000 0000 地址的字节数据的八个比特被映射到了 0x2200 0000 ~ 0x2200 001C 处的别名区域；而 0x200F FFFF 地址的字节数据的八个比特被映射到了 0x23FF FFE0 ~ 0x23FF FFFC 处的别名区域。

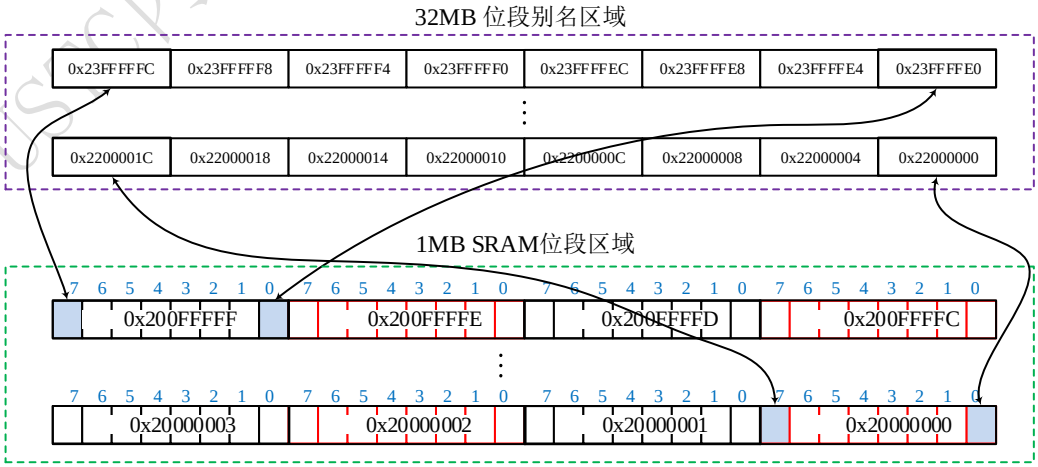

图 5.37 SRAM 区域位段与位段别名的映射关系

位段区域的存储单元可以像普通存储器一样访问，还可以通过名为位段别名的一块独立的存储器区域进行位段访问。当使用位段别名地址访问存储器时，所得到的字数据的 LSB 即对应位段区域中某个特定的位。外设区域的位段别名地址如图 5.38 所示，若读取别名地址 0x4200 002C，则返回结果可能是 0x0000 0000 或 0x0000 0001，分别代表 0x4000 0000 位置字数据的第 11 位取值为“0”和“1”。

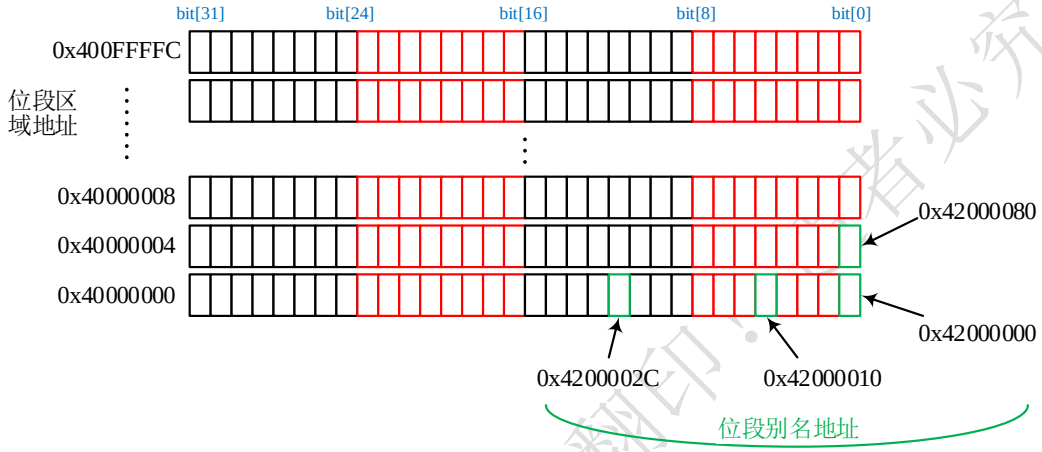

图 5.38 通过位段别名对位段区域进行位访问（外设区域）

例如，要设置地址 0x40000000 处字数据的第 2 位，在处理器不支持位段操作的情形下，需要“读→修改→写[11]”的流程：用三条指令读取数据、设置位，然后将结果写回。

```
❒  LDR   R0, =0x40000000[12]    ;设置地址
❒  LDR   R1, [R0]               ;读字数据
❒  ORR.W  R1, #0x4              ;修改字数据中的第2位
❒  STR   R1, [R0]               ;写回结果
```

而在支持位段操作的处理器中，可以仅使用一条单独的指令。Cortex-M3/M4 处理器帮用户完成了上述“读→修改→写”的流程。

```
❒  LDR   R0, =0x42000008        ;设置别名地址
❒  MOV   R1, #1                 ;设置数据位
❒  STR   R1, [R0]               ;写
```

类似地，若需要读出地址 0x40000000 处字数据的第 2 位，位段特性也可以简化读位操作的程序代码。在处理器不支持位段操作的情形下，程序员需要单独提取需要的位，代码如下。

```
❒  LDR   R0, =0x40000000        ;设置地址
❒  LDR   R1, [R0]               ;读出子数据
❒  UBFX.W  R2, R1, #2, #1       ;提取bit[2]至R2
```

而在支持位段操作的处理器中，可以仅使用一条单独的指令，代码如下。

```
❒  LDR   R0, =0x42000008        ;设置别名地址
❒  LDR   R1, [R0]               ;读
```

[11] 读-修改-写（Read-Modify-Write）：存储器操作指令同时读取或写数据总线上的所有位，即便用户仅需要修改其中一个位，写入的时候也需要写全部位，故而在写入某个位前需要先读取其他所有数据位。

[12] “LDR Rd,=const”是 ARM 汇编中的伪指令，对于较大的立即数，该伪指令可自动将立即数保存到存储器单元，然后通过 LDR 指令装载到寄存器。

事实上，位段操作并不是 Cortex-M 处理器独有的设计。8051 等经典八位微控制器就有类似特性：可位寻址的数据属于特殊数据类型，需要通过特殊的指令来访问位数据。Cortex-M3/M4 处理器中位操作有关的指令只能用于通用寄存器，虽然没有设计读/写存储器的特殊位操作指令，但对位段区域的数据访问会被自动转换为位段操作。

2. 位段操作的优点

从上述例子可看出，利用位段操作可以方便地进行位数据的读取和写入，相比传统“读→修改→写”流程进行位数据操作，代码更为简洁，有利于提高一些指令的位操作速度。除此之外，Cortex-M3/M4 处理器位段操作还有一个重要的特性：能够保证操作的原子性。

原子性指对数据位进行“读→修改→写”的流程不被其他操作打断。若没有这种特性，在进行“读→修改→写”的软件流程时，可能会出现下面的问题：假定数据端口的 bit[1]被主程序使用而 bit[0]被中断处理使用，如图 5.39 所示，基于“读→修改→写”操作的软件可能会引起数据丢失。

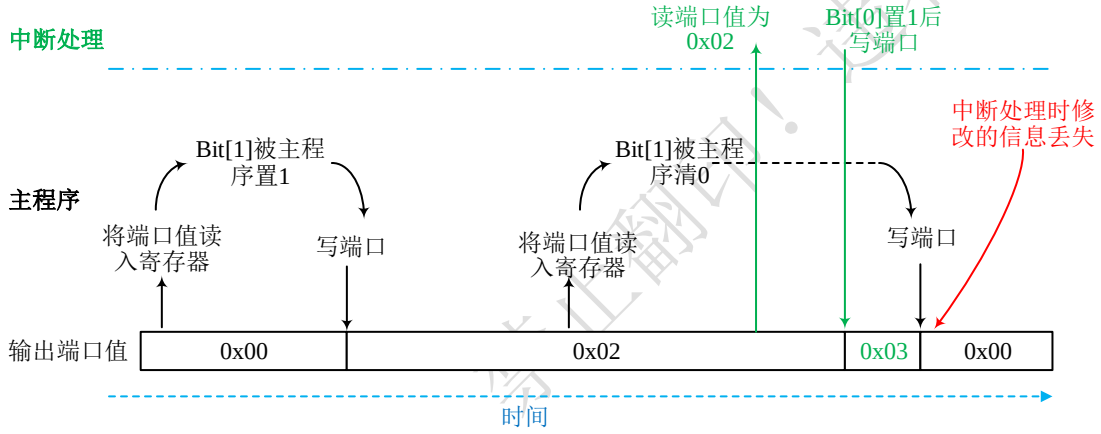

图 5.39 中断处理程序修改的位信息丢失

由于位段操作中“读→修改→写”是在硬件等级执行的，是原子性的，故可以避免这种主程序和中断处理程序之间对位数据资源的竞争。如图 5.40 所示，采用位段操作后，主程序修改 bit[1]和中断处理程序修改 bit[0]不会形成冲突。

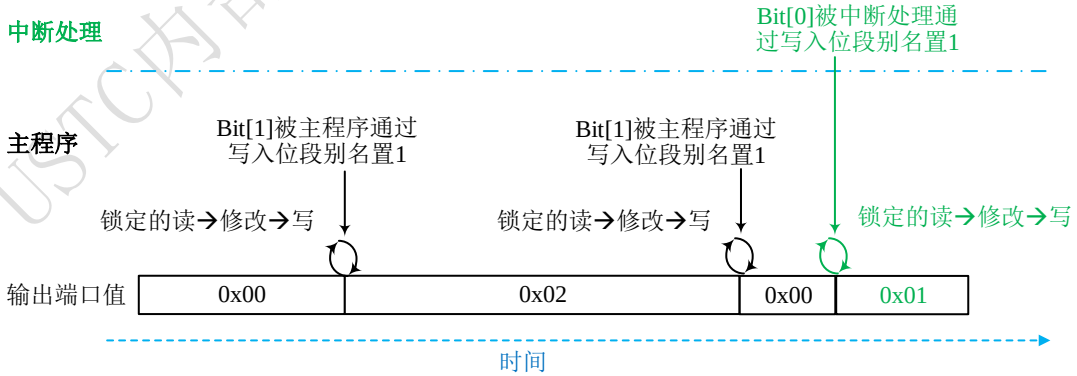

图 5.40 利用位段操作锁定传输以避免数据丢失

多任务系统中也存在类似的问题。例如，若数据端口的 bit[0]被进程 A 使用而 bit[1]被进程 B 使用，基于软件的“读→修改→写”可能会引起数据冲突，如图 5.41 所示。

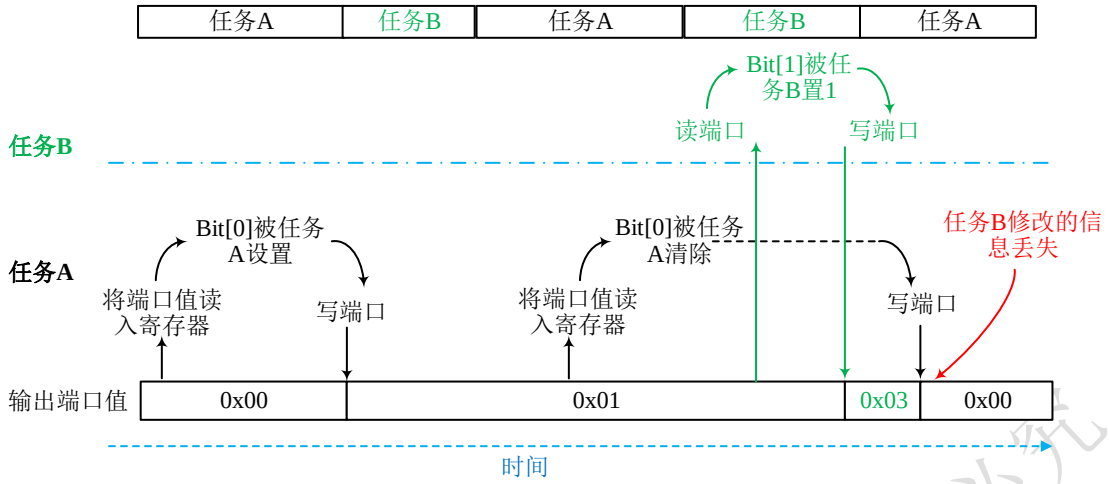

图 5.41 当不同任务修改共享存储器位置时出现数据丢失

与前述主程序和中断处理程序访问共享位置时类似，位段特性可以确保任务 A 和任务 B 的位访问独立，不会产生数据冲突，其操作过程如图 5.42 所示。

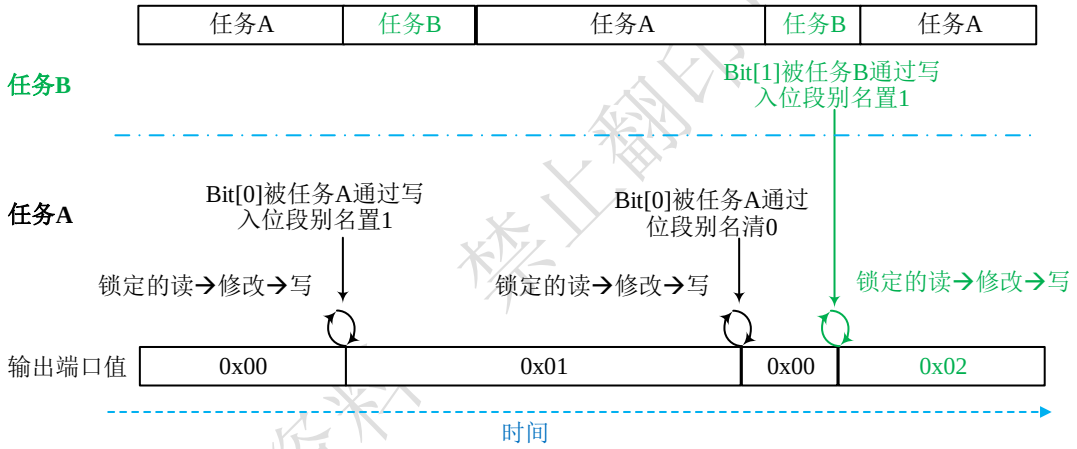

图 5.42 利用位段特性锁定传输以避免多任务写位数据的冲突

除了上述示例中将位段操作应用于 I/O 端口访问，还可以基于位段特性管理 SRAM 区域中的布尔型数据。如，多个布尔型数据可被合并到同一个存储器位置的不同数据位，这样可以节省存储器空间。

3. C 程序实现位段操作

C/C++本身不支持位段操作，这是因为 C 编译器不知道能用两个不同的地址来寻址同一个存储器位置。另外，C 编译器也不知道对位段别名的访问只会操作存储器位置数值的 LSB。如果需要在 C 程序中实现位段特性，可以在程序中声明存储器位置的地址和位段别名。例如，需要写 0x40000000 存储单元的 bit[1]时，代码如下：

```
#define DEVICE_REG0        *(( volatile unsigned long*) (0x40000000))
#define DEVICE_REG0_BIT0   *(( volatile unsigned long*) (0x42000000))
#define DEVICE_REG0_BIT1   *(( volatile unsigned long*) (0x42000004))
…
DEVICE_REG0 = DEVICE_REG0 | 0x2;      //未使用位段特性设置 bit[1]
…
```

```
DEVICE_REG0_BIT1 = 0x1;                    //利用位段特性通过位段别名地址设置 bit[1]
```

也可以利用 C 语言的宏定义进一步简化基于位段别名地址的访问。如，定义一个宏将位段地址和位数转换为位段别名地址，定义另一个宏将地址作为一个指针来访问存储器地址。如下所示。

```
//将位段地址和位编号转换为位段别名地址，其中 addr 是位段地址，bitnum 指示要访问的位
//用左移运算“<<5”实现了乘 32，左移运算“<<2”实现了乘以 4
#define BIT_BAND(addr, bitnum) ((addr & 0xF0000000)+0x2000000 \
        +((addr & 0xFFFFF)<<5)+(bitnum <<2))
//将地址转换为指针，其中 addr 是拟访问存储单元地址
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))
```

前述写 0x40000000 存储单元 bit[1]的例子，代码可以重写如下：

```
#define DEVICE_REG0 0x40000000
#define BIT BAND(addr, bitnum) ((addr & 0xF0000000)+0x02000000 \
        +((addr & 0xFFFFF)<<5)+(bitnum<<2))
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))
…
//未使用位段特性设置第 1 位
MEM_ADDR(DEVICE_REG0) = MEM_ADDR(DEVICE_REG0) | 0x2;
…
//利用位段特性设置第 1 位
MEM_ADDR(BIT_BAND(DEVICE_REG0,1)) = 0x1;
```

注意，使用位段特性时需要把被访问的变量定义为 volatile[13]，由于 C 编译器不知道同一个数据能以两个不同的地址访问，为防止编译器自动执行不期望的优化，需要利用 volatile 属性。

5.4.6 存储器访问权限

Cortex-M3 和 Cortex-M4 的处理器映射有默认的存储器访问权限配置，非特权用户程序不允许访问 SCS 空间。在没有 MPU 的情形，或有 MPU 但未使能时，使用默认的存储器访问权限，如表 5.24 所示。若 MPU 存在且被使能，则用户访问各存储器区域的权限也会受 MPU 设置制约。

表 5.24 默认的存储器访问权限

存储器区域	地址	用户程序的非特权访问
供应商定义	0xE0100000–0xFFFFFFFF	可访问
ROM 表	0xE00FF000–0xE00FFFFF	禁止访问，非特权访问将导致总线错误
外部 PPB	0xE0042000–0xE00FEFFF	禁止访问，非特权访问将导致总线错误
ETM	0xE0041000–0xE0041FFF	禁止访问，非特权访问将导致总线错误
TPU	0xE0040000–0xE0040FFF	禁止访问，非特权访问将导致总线错误
内部 PPB	0xE000F000–0xE003FFFF	禁止访问，非特权访问将导致总线错误
NVIC	0xE000E000–0xE000EFFF	除了软件触发中断寄存器可被编程为允许用户访问，其他寄存器禁止访问，非特权访问将导致总线错误
FPB	0xE0002000–0xE0003FFF	禁止访问，非特权访问将导致总线错误
DWT	0xE0001000–0xE0001FFF	禁止访问，非特权访问将导致总线错误
ITM	0xE0000000–0xE0000FFF	除了存放跟踪信息的端口可非特权访问，其他寄存器为读允许，写忽略

[13] C 语言中，volatile 的作用是作为指令关键字，确保本条指令不会因编译器的优化而省略。

存储器区域	地址	用户程序的非特权访问
外部设备	0xA0000000–0xDFFFFFFF	可访问
外部 RAM	0x60000000–0x9FFFFFFF	可访问
外设	0x40000000–0x5FFFFFFF	可访问
SRAM	0x20000000–0x3FFFFFFF	可访问
代码	0x00000000–0x1FFFFFFF	可访问

当非特权用户程序访问了仅特权用户可访问的区域时，会产生异常。依据总线错误异常是否使能和优先级的配置，可能会触发硬件错误异常或总线错误异常。

5.4.7 存储器访问属性

存储器的不同区域功能不同，和总线连接关系不同，而且每个存储器区域的存储器访问属性也不同。Cortex-M3/M4 处理器中定义的存储器访问属性包括：

- 可缓冲（Bufferable）：对存储器的写操作可由写缓冲执行，处理器不等待当前指令写操作完成就继续执行下一条指令。
- 可缓存（Cacheable）：读存储器所得到的数据可被复制到缓存，下次再访问时可以从缓存中取出这个数值从而加快程序执行。
- 可执行（Executable）：处理器可以从存储器区域读取并执行程序代码。
- 可共享（Sharable）：这种存储器区域的数据可被多个总线主设备共用。存储器系统需要确保不同总线主设备之间数据的一致性。

处理器从存储器读取或写入数据时，存储器系统会检查所访问区域的存储器访问属性。若 MPU 存在且 MPU 配置和默认的存储器属性不同，则默认的存储器属性设置会被覆盖。

可缓冲属性用于处理器内部。为了提供更优越的性能，Cortex-M3/M4 处理器支持总线接口的写缓冲。即使总线接口上的实际传输需要多个时钟周期才能完成，对可缓冲存储器区域的写操作可在单个时钟周期内执行，接下来会继续后续指令的执行。如图 5.43 所示，STR 指令需要写存储器，拟写入存储器的数据写入写缓冲后，继续执行 STR 后的指令，而实际数据写入操作完成时间滞后于 STR 指令开始执行的时间。

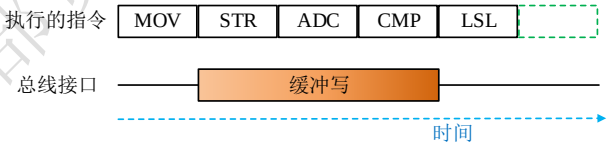

图 5.43 缓冲写操作

可缓存属性需要硬件上缓存单元的支持。虽然 Cortex-M3/M4 处理器中并不存在缓存和缓存控制器，但多数微控制器厂商在其产品中设计了缓存单元。

根据可缓冲和可缓存特性的支持与否，可以确定存储器的不同类型，表 5.25 所示为 Cortex-M3 和 Cortex-M4 用户手册中对存储器类型的定义。

表 5.25 与存储器类型相关的存储器属性

可缓冲	可缓存	存储器类型及特点
0	0	**强序**（Strongly-ordered）存储器。处理器在继续下一个操作前会等待总线接口上的传输完成。
1	0	**设备**。如果下一条指令不是存储器访问指令，则当前指令的写存储器操作可以交由写缓冲执行，继续执行下一条指令。
0	1	具有写通（WT，Write Through）缓存的普通存储器。

可缓冲	可缓存	存储器类型及特点
1	1	具有写回（WB，Write Back）缓存的普通存储器。

若系统中存在多个处理器，且具备缓存一致性控制单元，如图 5.44 所示，就需要用到可共享属性。当数据为各个处理器的共享信息时，某个处理器缓存内的数据可能会被另外一个处理器缓存并修改，故需要缓存控制器来确保该数据在各个缓存单元均是一致的。

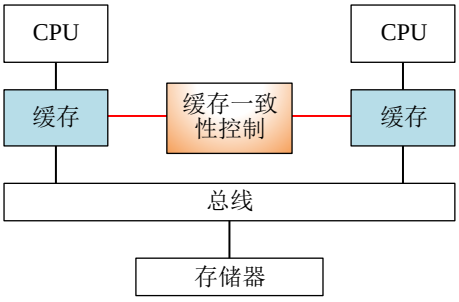

图 5.44 多处理器系统中的缓存一致性需要共享属性

对于大多数指令的存储器访问，存储器系统并不确保多个存储器访问完成的顺序与指令的顺序一致。虽然存储器系统不会改变指令执行的顺序，但是，多条顺序执行的指令中的存储器访问操作完成的时间不一定和指令的顺序一致。如果程序员需要确保存储器访问顺序和指令顺序完全一致，需要在程序中使用 5.4.9 小节所描述的存储器屏障指令。

但是，存储器系统会确保“强序”类型存储器和“设备”类型存储器（如表 5.25 所示）的访问顺序。例如，A1 和 A2 两条存储器访问指令，程序中 A1 指令在前，A2 指令在后，当 A1、A2 指令所访问的存储器类型不同时，这两条指令实际存储器访问结束的先后顺序如表 5.26 所示。表 5.26 中“—”表示存储器系统不确保 A1 和 A2 的访问顺序；“<”表示存储器操作的顺序与代码一致，即 A1 的存储器访问先结束。

表 5.26 A1、A2 指令实际完成存储器访问的顺序

A1 \ A2	访问普通存储器	访问设备		访问强序存储器
		不可共享	可共享	
访问普通存储器	—	—	—	—
访问设备，不可共享	—	<	—	<
访问设备，可共享	—	—	<	<
访问强序存储器	—	<	<	<

每个存储器区域的默认访问属性如表 5.27 所示。其中，XN（eXecute Never）表示永不执行，这也就意味着该区域不允许程序执行。

表 5.27 默认的存储器属性

区域，地址范围	存储器类型	XN	缓存	备注
代码 0x0000 0000 ~ 0x1FFF FFFF	普通	—	WT[14]	内部写缓冲使能，输出存储器属性总是可缓存，不可缓冲
SRAM	普通	—	WB-WA[14]	写回，写分配

[14] Write-allocate (WA)，写分配：回写数据时若缓存未命中则分配一个缓存行。
Write Through (WT)，写通：CPU 写数据时，写入到缓存的同时也写入主存，详见第 3 章。

0x2000 0000 ~ 0x3FFF FFFF				
外设 0x4000 0000 ~ 0x5FFF FFFF	设备	Y	—	可缓冲，不可缓存
RAM 0x6000 0000 ~ 0x7FFF FFFF	普通	—	WB-WA[14]	写回，写分配
RAM 0x8000 0000 ~ 0x9FFF FFFF	普通	—	WT[14]	写通
设备 0xA000 0000 ~ 0xBFFF FFFF	设备	Y	—	可缓冲，不可缓存
设备 0xC000 0000 ~ 0xDFFF FFFF	设备	Y	—	可缓冲，不可缓存
系统（PPB） 0xE000 0000 ~ 0xE00F FFFF	强序	Y	—	不可缓冲，不可缓存
系统（供应商定义） 0xE010 0000 ~ 0xFFFF FFFF	设备	Y	—	可缓冲，不可缓存

5.4.8 排他访问

当某个特定资源共享给多个用户使用时，常利用信号量来协调多个用户对该资源的占用。特别是，若某个共享资源只能满足一个用户使用时，该资源被称为互斥体（Mutex）。这种情况下，若某个资源被一个用户占用，它就会被锁定，在锁定解除前其他用户无法使用该资源。要创建互斥信号量，需要将互斥资源定义为锁定状态，以表示资源已被一个用户锁定。每个用户在使用资源前，需要先检查资源是否已被锁定，若未被使用，则设置为锁定状态，再开始使用资源。

ARM7TDMI 等传统的 ARM 处理器，锁定状态的访问由 SWP 指令执行，该指令可确保读写锁定状态的原子性，避免资源被两个用户同时锁定。较新的 ARM 处理器，如 Cortex-M3 和 Cortex-M4，读/写访问由独立的总线执行。但是 SWP 指令的锁定流程中只适用于读写复用同一总线的场景，此时使用 SWP 指令无法保证存储器访问的原子性，故已被新的排他（exclusive）访问指令取代了。

排他访问需要软硬件配合才能完成，一次排他写过程需要先用排他读指令（LDREX）获取拟访问地址的锁定状态；未被锁定时才可进行排他写（STREX）并设置锁定状态，该访问过程如图 5.45 所示。某特定资源没有被其他用户排他读或排他写时，处理器才能获得该资源的使用权。排他存储失败后，存储器中不会进行实际的写操作，其存储器操作会被处理器内核或外部硬件阻止。

Write Back(WB)，写回：CPU 写数据只写到缓存而主存中的数据不变，详见第 3 章。

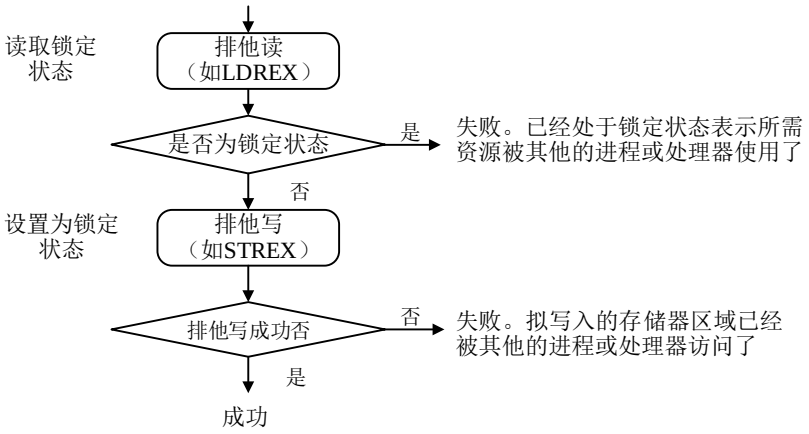

图 5.45 互斥体的排他访问

多处理器环境中使用存储器排他访问，需要增加“排他传输监控”硬件。该监控硬件检查共享地址单元的数据传输，指示处理器排他访问是否成功。处理器总线接口向监控硬件提供额外的控制信号，指示传输是否为排他访问。若存储器设备已经被某一个总线主设备访问，且处于排他读或排他写状态，那么试图进行排他写时，排他访问监控硬件会产生一个排他失败状态。

5.4.9 存储器屏障

Cortex-M3/M4 处理器支持存储器屏障指令（如 ISB、DSB 和 DMB）。利用存储器屏障指令，程序员可以控制不同指令存储器访问的先后顺序。

由于 Cortex-M3/M4 处理器不会调整程序中指令执行的顺序（在超标量处理或支持乱序执行的高性能处理器中可能会出现调整），所以多数应用程序中即便不使用存储器屏障指令，通常存储器操作的先后顺序也是和指令的先后顺序一致的。同时，由于 AHB Lite 和 APB 协议自身较为简单，不允许在前面的传输还未完成时就开始新的传输。故而绝大多数情况下，程序员不使用存储器屏障指令不会引起任何问题。

但处理器内部有一个写缓冲，数据写操作可能会和下一条指令的操作同步执行。在一些特殊的情况下，程序员可能需要保证下一条指令的操作不会在当前指令写操作完成前执行，就需要用到存储器屏障指令。例如，有些微控制器具有存储器重映射特性，存储器映射切换时，在写入重映射控制寄存器后，可使用 DSB 指令来保证缓冲写实际完成前，其他指令的存储器操作都不被执行。

5.5 Cortex-M 处理器的异常处理

如 5.2.5 小节所述，Cortex-M 处理器集成了一个嵌套中断控制器 NVIC，NVIC 能够响应外部中断、NMI、系统节拍定时器、系统异常等不同来源的异常，可以管理 256 种异常类型（如表 5.6 所示）。

由于第 4 章已经学习了中断的相关原理，本小节主要结合 Cortex-M 处理器的异常管理模型，来讨论在工程实践中常遇到的中断优先级的控制问题、向量表的重定位问题、异常/中断

的挂起和激活状态转换问题，最后给出了 NVIC 和 SCB 中部分基本寄存器的描述。同第 4 章一样，这里我们对术语“中断”和“异常”不作严格区分，统一以“异常处理”来描述 Cortex-M 处理器中 NVIC 技术。

5.5.1 Cortex-M 异常管理模型

基于 NVIC 的 Cortex-M 处理器可以响应不同来源的异常，根据中断类型号在异常向量表中查询对应的异常处理程序入口地址，优先级高的异常可以打断优先级低的异常处理程序。为了实现对上述过程的有效管理，NVIC 中定义了异常的优先级和优先级分组，定义了异常程序可能的工作状态，定义了异常挂起和解除挂起的条件。本小节依据 Cortex-M 系列处理器的用户手册描述，对异常管理中用到的概念逐一进行解释。

1. 异常类型

如 5.2.5 小节所述，Cortex-M 处理器支持的多种异常类型，不同类型的异常及其产生原因分析如表 5.6，此处不再赘述。

2. 异常状态

在 Cortex-M 处理器中，每个异常都会处于某个状态（激活、非激活、挂起、激活并挂起），各状态的含义如表 5.28 所示。

表 5.28 Cortex-M 处理器定义的异常状态

异常状态	英文全称	状态的含义
非激活状态	Inactive	异常既不在激活状态也不在挂起状态
挂起状态	Pending	异常源发出了服务请求，正在等待处理器
激活状态	Active	正在接受处理器服务但未结束的异常（如果某异常处理程序被更高优先级的异常服务打断，则两个异常均处于激活状态）
激活并挂起状态	Active and pending	异常正在接受处理器服务，而相同异常源又产生了异常请求

3. 异常处理程序

Cortex-M 处理器在响应异常后，可以进入不同类别的异常处理程序（Exception handlers），如表 5.29 所示。表 5.6 所示不同来源的异常由不同类别的异常处理程序处理。

表 5.29 Cortex-M 处理器可服务的异常处理程序类别

异常处理程序	英文名称	功能描述
中断服务子程序	ISR	片上外设或者外设中断源产生 IRQ 由 ISR 处理
错误程序	Fault handlers	硬件错误、MemManage 错误、总线错误、用法错误产生的异常由错误程序处理
系统异常程序	System handlers	NMI、PendSV、SVCall、SysTick 产生的异常由系统异常程序处理

4. 异常向量表

当 Cortex-M3/M4 处理器接受了某异常请求后，处理器需要确定该异常对应的异常处理程序的起始地址。该信息位于存储器内的异常（中断）向量表中，向量表默认从地址 0x0000 0000 开始，按照异常类型号依次存放各个异常的入口地址，如表 5.7 所示。

表 5.7 所示向量表的开始（0x0000 0000）4 个字节存放的是主栈指针（MSP）的初始值，其后依次存放各个异常源对应的处理程序入口地址（该入口地址也称作异常向量）。异常向量

的存放地址为异常编号乘 4，如，总线错误异常类型号为 5，总线错误向量的地址为 0x0000 0014。此外，异常向量的 LSB 必须是“1”，表示异常处理程序是 Thumb 代码[15]。

5. 异常的优先级

在 Cortex-M 处理器中，异常是否能被处理器接受以及何时进入异常处理程序，是与异常的优先级相关的。高优先级的异常（优先级编号更小）可以抢占低优先级的异常（优先级编号更大），这就是异常/中断嵌套的情形。有些异常（复位、NMI 和 HardFault）具有固定的优先级，其优先级编号为负数，因而这些异常的优先级总是比其他异常的优先级高。其他异常则具有可编程的优先级，范围为 0~255。

Cortex-M3/M4 处理器设计了 3 个固定的最高优先级（如表 5.6 所示：复位为-3、NMI 为-2、硬件错误为-1）以及 256 个可编程优先级，芯片支持的可编程优先级的实际数量由芯片厂商决定。考虑到大的优先级数量会增加 NVIC 复杂度并增加功耗，一般情况下厂商倾向于使用较少的优先级数量（如 16）。优先级的减少是通过去除优先级配置寄存器的最低位（LSB）实现的。

每个异常都有一个优先级寄存器与之对应。异常的优先级通过优先级寄存器配置，宽度为 3~8 位。例如，若芯片设计中只实现了 3 位优先级（可设置 8 个可编程优先级），优先级配置寄存器如图 5.46 所示。此时寄存器低 5 位未实现，故读出总是为 0，对这些位的写操作会被忽略。根据这种设置，可能的优先级为 0x00（高优先级）、0x20、0x40、0x60、0x80、0xA0、0xC0 以及 0xE0（最低），如图 5.47 所示。

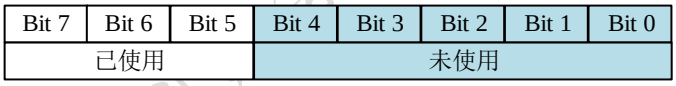

图 5.46 三位优先级寄存器

类似地，若设计中实现了 4 位优先级，会得到 16 个可编程优先级，可能的优先级如图 5.47 所示。

[15] Cortex-M 系列的处理器支持的指令有 16 比特长和 32 比特长度两种，指令所占用的存储单元最小单位是 2 个字节，因而程序入口地址的 LSB 被认为是“0”。最低位为 1，只是表示处于 Thumb 状态。

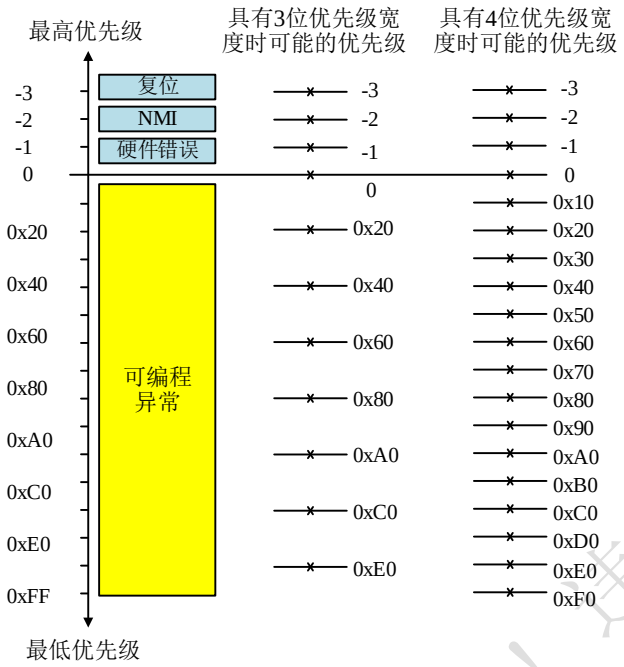

图 5.47 三位或四位优先级宽度可用的优先级

优先级配置寄存器中实现的位数越多，可用的优先级数量就越多。对于 ARMv7-M 架构，宽度最少为 3 位（8 个等级）。

图 5.46 **错误!未找到引用源。**所示优先级寄存器实现了较高的位而不是较低位，即弃用了最低的位。这样的处理使 Cortex-M 设备间移植软件更加容易。按这种方式，在具有 4 位优先级配置寄存器设备上写的程序，可直接在具有 3 位优先级配置寄存器的设备上运行。若弃用最高的位，则在 Cortex-M 设备间移植应用程序时优先级的配置可能会相反。例如，若对于某应用程序，IRQ#0 的优先级是 0x09，IRQ#1 的优先级是 0x03，那么 IRQ#1 的优先级高。若弃用最高位 bit3，则 IRQ#0 的优先级变为 0x01，且优先级大于 IRQ#1。

注意：复位后所有可配置中断都处于禁止状态，默认的优先级为 0。如果多个相同优先级的异常均处于等待状态，处理器会先处理异常类型号低的。例如，IRQ#0 和 IRQ#1 均为等待状态，处理器会先处理 IRQ#0 的 ISR。这种依据异常类型号大小决定处理顺序的机制常被称作**自然顺序优先级**。

6. 异常优先级分组

为了提高对异常优先级的控制能力，NVIC 设计了优先级分组的机制。八位的优先级配置寄存器被分为两个部分：分组优先级[16]（group priority）和（组内的）子优先级（subpriority）。

通过设置系统控制块（SCB）中的应用中断和复位控制寄存器（AIRCR）的 PRIGROUP 域（AIRCR[10:8]，共三位，对应八种优先级分组），优先级配置寄存器可被分为两部分。高位的部分（左边的位）为分组（抢占）优先级，而低位的部分（右边的位）则为子优先级，不同优先级分组时优先级寄存器位定义如表 5.30 所示。

表 5.30 优先级寄存器中抢占优先级域和子优先级域定义

优先级分组	抢占优先级域	子优先级域
0（默认）	Bit[7:1]	Bit[0]

[16] 在 ARM 公司早期的技术资料中，分组优先级称为抢占优先级（preempt priority）。

1	Bit[7:2]	Bit[1:0]
2	Bit[7:3]	Bit[2:0]
3	Bit[7:4]	Bit[3:0]
4	Bit[7:5]	Bit[4:0]
5	Bit[7:6]	Bit[5:0]
6	Bit[7]	Bit[6:0]
7	无	Bit[7:0]

图 5.48 给出了表 5.30 中优先级分组设置为 2、7、6、0 情形下优先级寄存器配置的直观示意图。由于抢占优先级域和子优先级域可配置为不同的宽度组合，提高了异常优先级配置管理的灵活度。

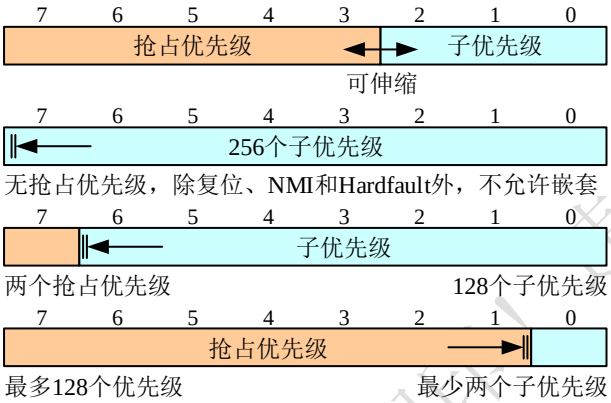

图 5.48 优先级域和子优先级域定义的不同情形

正在运行的一个异常处理是否会被新产生的异常打断，是由异常的抢占优先级决定的。抢占优先级高的异常可以打断抢占优先级低的异常。若新产生异常的抢占优先级没有比正在运行异常处理的抢占优先级高，则新产生的异常不会打断已经运行的异常。

子优先级只会用在两个抢占优先级相同的异常同时产生的情形，此时，具有更高子优先级（数值更小）的异常会被首先处理。

在确定实际的分组优先级和子优先级时，需要考虑：①优先级配置寄存器实际实现的位宽度，②优先级分组设置。例如，若优先级配置寄存器的宽度为 3（第 7~5 位可用）且优先级分组为 5，则会有 4 个分组/抢占优先级（第 7~6 位），而且每个分组/抢占优先级具有两个子优先级，如图 5.49 所示。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
抢占优先级		子优先级	未使用				

图 5.49 3 位优先级寄存器中优先级分组为 5 时的位定义

对于相同的优先级配置寄存器设计（实现了 3 位），若优先级分组设置为 1，则只会有 8 个分组优先级，且每个抢占等级中没有进一步的子优先级（优先级寄存器的 bit[1:0]总是 0）。优先级配置寄存器的位定义如图 5.50 所示。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
抢占优先级[5:3]			抢占优先级[2:0]（总是为0）			子优先级[1:0]（总是为0）	

图 5.50 3 位优先级寄存器中优先级分组为 1 时的位定义

若 Cortex-M3/M4 实现了优先级配置寄存器中的所有 8 位，优先级分组设置为 0，则有 128

个抢占优先级，每个抢占优先级有 2 个子优先级。此时优先级配置寄存器的位定义如图 5.51 所示。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
抢占优先级							子优先级

图 5.51 3 位优先级寄存器中优先级分组为 0 时的位定义

若两个异常同时产生，且它们有相同的抢占优先级和子优先级，则异常编号更小的中断的优先级更高（IRQ#0 的优先级高于 IRQ#1），即前述自然顺序优先级机制。

7. 异常流程

异常处理的过程包括：处理器接受异常请求；进入异常处理；执行异常处理程序；异常返回。

1) 异常请求的接受

若异常没有被屏蔽、处于使能状态（NMI 和硬件错误总是使能的），且异常的优先级高于当前等级，处理器会接受异常请求。注意，如果异常处理程序中出现了 SVC 指令，而该异常的优先级不低于 SVC 的优先级，就会触发硬件错误，从而进入硬件错误的处理程序。

2) 异常进入流程

异常进入流程包括如下操作：

①多个寄存器和返回地址被压入当前使用的栈。在不保护浮点运算寄存器组情形下，被压入堆栈的寄存器包括 PSR、PC、LR、R0~R3、R12，共 8 个字；如果需要保护浮点运算单元状态，则有 26 字的状态信息会被压栈。若处理器处于线程模式且正使用进程栈指针（PSP），则压栈过程使用 PSP 指向的栈区域，否则就会使用主栈指针（MSP）指向的栈区域。

②更新内核寄存器和多个 NVIC 寄存器。从异常（中断）向量表中取出的异常向量存入 PC；根据压栈时使用的栈，MSP 或 PSP 的数值会在异常处理开始前自动调整；LR 被更新为特殊值 EXC_RETURN。EXC_RETURN 数值共 32 位，高 27 位为 1，低 5 位用于指示进入异常是保存的状态信息（即指示使用的是 MSP 还是 PSP，哪些寄存器被压入栈），其可能数值如表 5.31 所示。

3) 执行异常处理程序

进入异常处理程序内部后，处理器进入处理模式，并运行于特权访问等级，栈操作使用 MSP。此过程中如果有更高优先级的异常产生，处理器会接受新的异常，当前正在执行的处理被更高优先级的处理抢占而进入挂起状态，此即异常嵌套。若执行过程中产生的其他异常具有相同或更低的优先级，新产生的异常就会进入挂起状态，待当前异常处理完成后才可能被处理。

4) 异常返回

把 PC 设置为 EXC_RETURN 数值会触发异常返回流程，该数值在异常进入时产生且被存储在 LR 中。EXC_RETURN 取值不同的时候，返回行为的差异如表 5.31 所示。

表 5.31 异常返回的行为

EXC_RETURN[31:0]	**返回行为的描述**
0xFFFFFFF1	返回至处理模式，使用 MSP 恢复非浮点状态信息[17]

[17] 使能了 FPU 时，进入异常处理的现场保护过程会保存更多寄存器信息到栈中，故返回的时候也需要恢复

0xFFFFFFF9	返回至线程模式，使用 MSP 恢复非浮点状态信息
0xFFFFFFFD	返回至线程模式，使用 PSP 恢复非浮点状态信息
0xFFFFFFE1	返回至处理模式，使用 MSP 恢复浮点状态信息
0xFFFFFFE9	返回至线程模式，使用 MSP 恢复浮点状态信息
0xFFFFFFED	返回至线程模式，使用 PSP 恢复浮点状态信息

异常返回可由表 5.32 中所示的指令产生。异常返回机制被触发后，进入异常时被压入栈中的寄存器值会被恢复到寄存器组中，NVIC 的多个寄存器和处理器内核中的多个寄存器都会被更新。

表 5.32 可用于触发异常返回的指令

返回指令	描述
BX　LR	若 EXC_RETURN 数值仍在 LR 中，在异常处理结束时使用 BX　LR 指令将 EXC_RETURN 写入 PC，触发异常返回
POP　PC	若异常处理过程把 LR 值压入栈，可使用 POP 指令将 EXC_RETURN 放到 PC 中，触发异常返回
LDR 或 LDM	以 PC 为目标寄存器的 LDR 或 LDM 指令也可以触发异常返回

5.5.2 向量表重定位机制

向量表默认从地址 0x0000 0000 开始，但有些应用可能需要在运行时修改向量表，Cortex-M3/M4 处理器设计了向量表重定位机制。向量表重定位使用 VTOR（Vector Table Offset Register，向量表偏移寄存器）指示向量表的位置。VTOR 中保存向量表相对于存储器的起始地址（0x0000 0000）的偏移量。Cortex-M3/M4 处理器中，VTOR 的低七位（VTOR[6:0]）总是为零，其高 25 位（VTOR[31:7]，称为 TBLOFF）可配置。特别是，当 VTOR[29]位为“0”时，VTOR 所表示的向量表地址处于代码区，VTOR[29]位为“1”时，则表示向量表地址在 SRAM 区，故很多资料中把 VTOR[29]称作 TBLBASE 位。

在设置 VTOR 的时候，要注意 TBLOFF 的值需要和向量表中异常向量数目匹配。TBLOFF 的最小值为 32 字（VTOR[6:0]总是为“0”），考虑每个中断向量占一个字（四字节），此时可以支持 16 个系统异常，以及 16 个 IRQ。Cortex-M3 和 Cortex-M4 技术手册中规定，TBLOFF 的值（以字为单位）必须为 *2* 的整数次幂，且大于中断向量数。

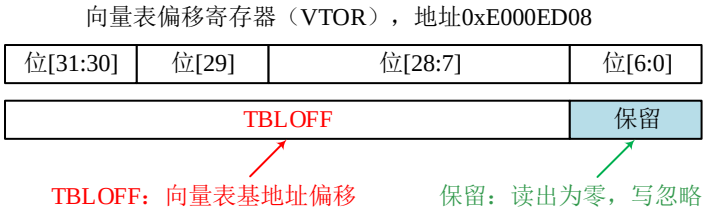

图 5.52 向量表偏移寄存器

例如，微控制器有 21 个中断源。向量表大小为：21（用于中断）+16（用于系统异常空间）= 37 字。取大于 37 的 2 的整数次幂最小为 64，故向量表大小取 64 字 = 256 字节，因此，向量表的地址可被设置为 0x0000 0000、0x0000 0100 以及 0x0000 0200 等。

更多的寄存器信息。未使能 FPU 时压栈信息共 8 字，而使能 FPU 后压栈的信息有 26 字。

向量表重定位机制非常有利于实现灵活的启动方式。例如，图 5.53 所示为典型的具有启动代码（Bootloader）的设备中向量表的重定位过程。多数微控制器芯片有多个程序存储器：启动 ROM 和 Flash 存储器。芯片厂商将 Bootloader 预先写入启动 ROM，故微控制器启动时，启动 ROM 中的 Bootloader 先执行，从 Bootloader 跳转到 Flash 存储器中的用户应用程序前，设置 VTOR 指向用户 Flash 存储器的开始处，从而向量表切换为用户 Flash 中的向量表。

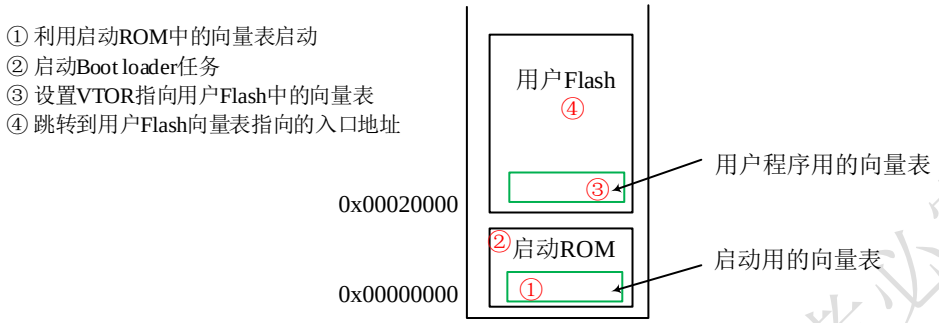

图 5.53 向量表在启动 ROM 和用户 Flash 间切换

有些微控制器芯片启动时，支持外部设备（如 SD 卡，或通过网络接口下载而来）加载用户程序。这种情形下，存储在芯片内的启动程序初始化相关硬件，从外部设备复制应用程序（含向量表）复制到 RAM，再更新 VTOR，然后依据载入的向量表执行已加载至 RAM 的程序。图 5.54 所示即为从外部存储卡加载应用并重定位向量表的过程。

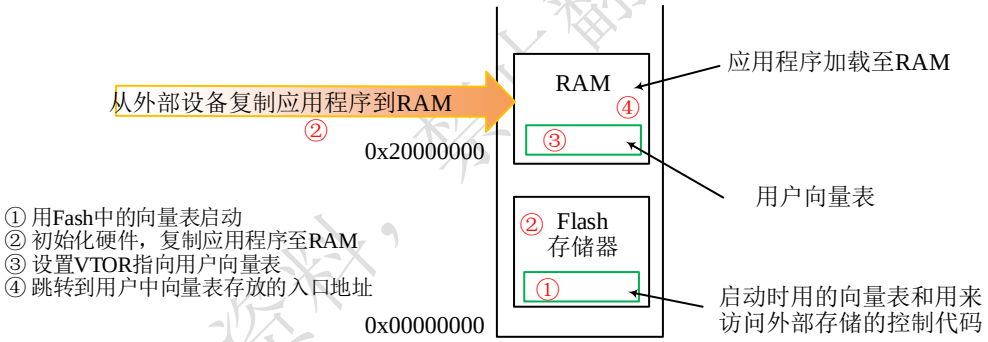

图 5.54 从外部存储加载的应用程序

5.5.3 异常请求和挂起

每个异常/中断都会处于表 5.28 所示的某个状态：挂起（等待服务的请求）或解除挂起，激活（正在处理）或非激活状态。本小节讨论不同状态之间转换的场景和条件。

在传统的 ARM 处理器中，若设备产生了中断请求，在得到处理前需要一直保持中断请求信号。而 Cortex-M 处理器的 NVIC 中，设计了用于保存中断请求的挂起请求寄存器，即便请求中断服务的源设备没有一直维持请求信号，已产生的中断仍会被处理，在被处理前该中断保持在挂起状态。

如果处理器空闲，处于挂起状态的中断请求会马上得到处理，此时，中断的挂起状态会被自动清除。但如果处理器正在处理另外一个更高优先级或同等优先级的中断，或者产生请求的中断源被屏蔽了（通过设置中断屏蔽寄存器），那么在其他中断处理结束前或中断屏蔽被清除前，该中断会一直保持在挂起状态。

当某个中断被处理时，该中断就会进入激活状态。NVIC 设计了中断激活状态寄存器来保存每个中断的激活状态，只有在中断服务完成，处理器执行了异常返回后，中断激活状态寄存器中对应位才会被清除（自动完成）。

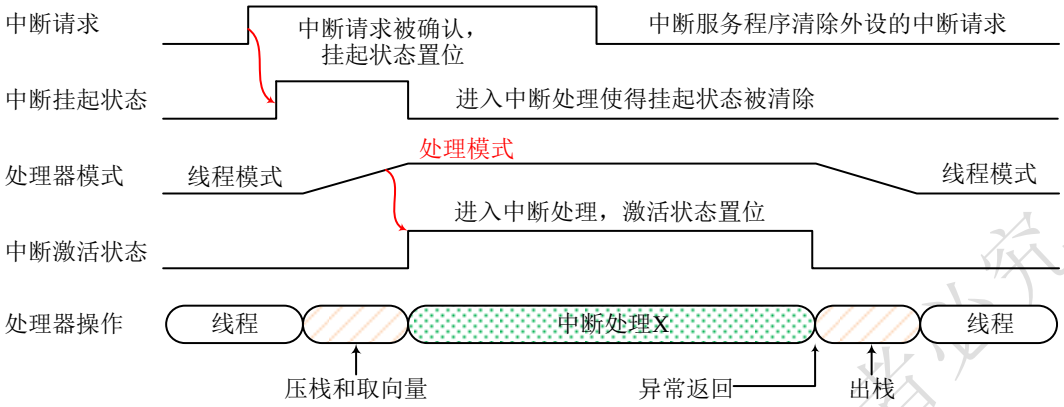

图 5.55 中断挂起和激活行为的简单情况

图 5.55 所示中断从挂起状态、挂起状态清除、进入激活状态、清除激活状态的过程中，处理器经历了线程模式到处理模式再回线程模式的切换。在处理器从线程模式切换到处理模式时，多个寄存器会被自动压入栈中，同时 ISR 的起始地址会被从向量表中取出。而从处理模式回到线程模式时，之前自动压栈的寄存器会被恢复，继续执行此前被打断的程序。

实际使用异常/中断处理时，还有很多细节的工程问题需要注意。例如，当某个特定中断处于激活状态时，同一个中断源如果再次产生中断请求，新的中断请求在本次中断服务结束前会持续保持在挂起状态。再如，对于脉冲形式的中断请求，若在处理器开始处理前，中断请求信号产生了多次，它们会被当作一次中断请求。还有些情况下，中断被禁止了，但其中断请求仍会引起挂起寄存器中对应位被置位，若稍后该中断被使能了，它仍可以被触发进入激活状态得到服务。

5.5.4 NVIC 寄存器

NVIC 中有多个中断控制寄存器，这些寄存器位于系统控制空间（SCS）地址区域。这些寄存器服务于异常类型 16~255 的外部中断，但是不能管理 NMI、SysTick 等系统异常，SCB 中的寄存器才可用于系统异常管理。NVIC 中各寄存器名称与功能描述如表 5.33 所示。

表 5.33 用于中断控制的 NVIC 寄存器列表

地址	寄存器全称	寄存器名称	功能
0xE000E200~ 0xE000E11C	中断设置使能寄存器 （Interrupt Set-enable Registers）	NVIC_ISER0~ NVIC_ISER7	写 1 设置使能，系统复位后清零
0xE000E180~ 0xE000E19C	中断清除使能寄存器 （Interrupt Clear-enable Registers）	NVIC_ICER0~ NVIC_ICER7	写 1 清除使能，系统复位后清零
0xE000E200~ 0xE000E21C	中断设置挂起寄存器 （Interrupt Set-pending Registers）	NVIC_ISPR0~ NVIC_ISPR7	写1设置挂起状态，系统复位后清零
0xE000E280~ 0xE000E29C	中断清除挂起寄存器 （Interrupt Clear-pending Registers）	NVIC_ICPR0~ NVIC_ICPR7	写 1 清除挂起状态
0xE000E300~ 0xE000E31C	中断激活位寄存器 （Interrupt Active Bit Registers）	NVIC_IABR0~ NVIC_IABR7	激活状态位，只读
0xE000E400~ 0xE000E4EF	中断优先级寄存器 （Interrupt Priority Registers）	NVIC_IPR0~ NVIC_IPR59	每个中断的中断优先级

0xE000EF00	软件触发中断寄存器 （Software Trigger Interrupt Register）	STIR	写中断类型号触发相应中断

表 5.33 中除了软件触发中断寄存器（STIR）外，其他所有寄存器都只能在特权等级访问。STIR 默认只能在特权等级访问，但可配置为非特权等级可访问。

1. 中断使能寄存器

在一些传统的微控制器芯片中，中断使能和禁止用一个寄存器实现。如，某特定位为 1 表示对应的中断被使能，反之该中断被禁止。Cortex-M 处理器中，中断使能和禁止通过两个寄存器进行配置。设置特定中断源的使能需要写入 NVIC_ISERn 寄存器的相应位；清除使能（禁止中断）则需要写入 NVIC_ICERn 寄存器的相应位。通过这样的设计，使能或禁止某个中断的时候不会影响其他中断的使能状态。

每个 ISER/ICER 寄存器都是 32 位宽，每个位对应一个中断输入。若芯片厂家实现的 Cortex-M3/M4 处理器支持超过 32 个外部中断源，则需要多个 ISER 和 ICER 寄存器，如 NVIC_ISER0 和 NVIC_ISER1 等，寄存器的位定义如表 5.34 所示。

表 5.34 中断使能设置和清除寄存器

地址	名称	类型	描述
0xE000E100	NVIC_ISER0	R/W	写 1 使能中断#0~#31，写 0 无作用 读出值指示当前使能状态 Bit[0]用于中断#0（异常类型号#16） … Bit[31]用于中断#31（异常类型号#47）
0xE000E104	NVIC_ISER1	R/W	写 1 使能中断#32~#63，写 0 无作用 读出值指示当前使能状态
…	…	…	…
0xE000E180	NVIC_ICER0	R/W	清零中断#0~#31 的使能 写 1 清除使能，写 0 无作用 读出值指示当前使能状态 Bit[0]用于中断#0（异常类型号#16） … Bit[31]用于中断#31（异常类型号#47）
0xE000E184	NVIC_ICER1	R/W	清零中断#32~#63 的使能 写 1 清除使能，写 0 无作用 读出值指示当前使能状态
…	…	…	…

2. 设置中断挂起和清除中断挂起寄存器

Cortex-M 处理器中，设置中断挂起、清除中断挂起（常简称为解挂）通过两个寄存器实现。若中断请求产生但没有立即执行，就会进入挂起状态。中断挂起状态可以通过访问中断设置挂起寄存器（NVIC_ISPRn）和清除中断挂起（NVIC_ICPRn）寄存器获取。与使能寄存器类似，若存在超过 32 个的外部中断输入，挂起状态控制寄存器就会不止一个。

挂起状态寄存器可软件修改，如通过设置 NVIC_ICPRn 取消一个当前被挂起的异常，或通过设置 NVIC_ISPRn 产生软件中断，寄存器的位定义如表 5.35 所示。

表 5.35 中断挂起设置和清除寄存器

地址	名称	类型	描述
0xE000E200	NVIC_ISPR0	R/W	设置中断#0~#31 的挂起 写 1 即设置挂起，写 0 无作用 读出值指示当前挂起状态 Bit[0]用于中断#0（异常类型号#16） Bit[1]用于中断#1（异常类型号#17） … Bit[31]用于中断#31（异常类型号#47）
0xE000E204	NVIC_ISPR1	R/W	设置中断#32~#63 的挂起 写 1 即设置挂起，写 0 无作用 读出值指示当前挂起状态
…	…	…	…
0xE000E280	NVIC_ICPR0	R/W	清零中断#0~#31 的挂起 写 1 即清零挂起，写 0 无作用 读出值指示当前挂起状态 Bit[0]用于中断#0（异常类型号#16） Bit[1]用于中断#1（异常类型号#17） … Bit[31]用于中断#31（异常类型号#47）
0xE000E284	NVIC_ICPR1	R/W	清零中断#32~#63 的挂起 写 1 即清零挂起，写 0 无作用 读出值指示当前挂起状态
…	…	…	…

3. 激活状态寄存器

每个外部中断都在中断激活状态寄存器中有一个激活状态位。处理器执行中断处理时，该位会被置 1，该位在执行中断返回时会被清零。ISR 执行期间，如果产生更高优先级的中断请求，可能会发生抢占，如果发生抢占，需要暂停当前的 ISR 去执行更高优先级的 ISR。在此期间，尽管处理器在执行优先级高的中断处理，之前较低优先级的中断仍处于激活状态。程序员可以通过中断设置挂起寄存器（NVIC_ISPRn）获知特定的异常/中断是否处于挂起状态，通过中断激活状态寄存器（NVIC_IABRn）获知特定的异常/中断是否处于激活状态。

中断激活状态寄存器为 32 位宽。若外部中断的数量超过 32，则激活状态寄存器会不止一个。外部中断的激活状态寄存器是只读的，寄存器位的定义如表 5.36 所示。

表 5.36 中断激活状态寄存器

地址	名称	类型	描述
0xE000E300	NVIC_IABR0	R	外部中断#0~#31 的激活状态 Bit[0]用于中断#0（异常类型号#16） Bit[1]用于中断#1（异常类型号#17） … Bit[31]用于中断#31（异常类型号#47）
0xE000E304	NVIC_IABR1	R	外部中断#32~#63 的激活状态
…	…	…	…

4. 优先级寄存器

每个中断都有各自的优先级寄存器，寄存器最大宽度为 8 位，最小为 3 位。每个寄存器可以根据优先级分组设置被进一步划分为分组优先级和子优先级，详见 5.5.1 小节中“异常的优先级”。优先级寄存器可以通过字节、半字或字访问。优先级寄存器的数量取决于芯片中实际存在的外部中断数，如表 5.37 所示。

表 5.37 中断优先级寄存器

地址	名称	类型	描述
0xE000E400	NVIC_IPR0	R/W	外部中断#0~#3 的优先级
0xE000E401	NVIC_IPR1	R/W	外部中断#4~#7 的优先级
…	…	…	…
0xE000E4EF	NVIC_IPR59	R/W	外部中断#236~#239 的优先级

5. 软件触发中断寄存器

除了 NVIC_ISPRn 寄存器外，还可以通过写软件触发中断寄存器（STIR，见表 5.38）来触发中断。Cortex-M 处理器中 STIR 只实现了八位，写入中断类型号即可触发对应的中断。

表 5.38 软件触发中断寄存器

位	名称	类型	复位值	描述
8:0	STIR	W	—	写中断编号可以设置对应编号中断的挂起位

NVIC_ISPRn 的访问只能由特权等级代码访问，但 STIR 可由非特权代码访问。非特权代码访问访问 STIR 的前提是，设置了配置控制寄存器中的 USERSETMPEND 域。由于 USERSETMPEND 位默认为清零状态，故未配置时只有特权等级代码才能访问 STIR。

5.5.5 SCB 寄存器

系统控制块（SCB）包含了一些用于中断控制的寄存器，表 5.39 为 SCB 中的寄存器列表。表 5.39 所示寄存器中只有一部分与中断或异常控制有关。本小节仅讨论 ICSR、AIRCR、SHPR 和 SHCRS 寄存器的功能，其他寄存器的详细信息可参阅 Cortex-M3 或 Cortex-M4 用户手册。

表 5.39 SCB 中的寄存器

地址	寄存器全称	寄存器简称	功能
0xE000ED00	CPU ID	CPUID	指示处理器类型和版本的 ID
0xE000ED04	中断控制和状态寄存器	ICSR	系统异常的控制和状态
0xE000ED08	向量表偏移寄存器	VTOR	指示向量表相对于存储器的起始地址（0x0000 0000）的偏移量
0xE000ED0C	应用中断/复位控制寄存器	AIRCR	优先级分组、端配置和复位控制
0xE000ED10	系统控制寄存器	SCR	配置休眠模式和低功耗特性
0xE000ED14	配置控制寄存器	CCR	进入线程模式的高级特性配置
0xE000ED18	系统处理优先级寄存器	SHPR1	系统异常的优先级设置
0xE000ED1C	系统处理优先级寄存器	SHPR2	系统异常的优先级设置
0xE000ED20	系统处理优先级寄存器	SHPR3	系统异常的优先级设置
0xE000ED24	系统处理控制和状态寄存器	SHCRS	使能错误异常和系统异常状态控制
0xE000ED28	可配置错误状态寄存器	CFSR	指示错误异常信息
0xE000ED2C	硬件错误状态寄存器	HFSR	指示硬件错误异常信息
0xE000ED30	调试错误状态寄存器	DFSR	指示调试事件信息
0xE000ED34	存储器管理错误寄存器	MMFAR	指示存储器管理错误的地址值
0xE000ED38	总线错误寄存器	BFAR	指示总线错误的地址值
0xE000ED3C	辅助错误状态寄存器	AFSR	指示设备相关错误状态信息
0xE000ED88	协处理器访问控制寄存器	CPACR	指示协处理器的访问权限

1. 中断控制和状态寄存器

ICSR（Interrupt Control and State Register，中断控制和状态寄存器）用于设置和清除系统

异常的挂起状态。通过 ICSR 可以获知当前正接受处理异常的类型号、当前挂起异常中优先级最高者的类型号、是否发生了抢占等信息。ICSR 的位定义如表 5.40 所示。

表 5.40 中断控制和状态寄存器

位	名称	类型	描述
31	NMIPENDSET	R/W	写 1 挂起 NMI，读出值表示 NMI 挂起状态
28	PENDSVSET	R/W	写 1 挂起 PendSV，读出值表示挂起状态
27	PENDSVCLR	W	写 1 清除 PendSV 挂起状态
26	PENDSTSET	R/W	写 1 挂起 SysTick，读出值表示挂起状态
25	PENDSTCLR	W	写 1 清除 SysTick 挂起状态
23	ISRPREEMPT	R	指示调试退出后是否执行被挂起的异常
22	ISRPENDING	R	指示是否有外部异常处于挂起状态
20:12	VECTPENDING	R	指示被挂起的异常中优先级最高者的类型号
11	RETTOBASE	R	指示是否发生了激活异常的抢占
8:0	VECTACTIVE	R	当前执行的异常类型号

2. 应用中断和复位控制寄存器

AIRCR（Application Interrupt and Reset Control Register，应用中断和复位控制寄存器）用于控制异常/中断优先级管理中的优先级分组，指示系统的端配置信息，提供自复位特性。AIRCR 位定义如表 5.41 所示。

表 5.41 应用中断和复位控制寄存器

位	域	类型	描述
31:16	VECTKEY VECTKEYSTAT	W R	写 AIRCR 时必须要将 0x05FA 写入 VECTKEY 域，读出时读取 VECTKEYSTAT
15	ENDIANESS	R	1 表示系统为大端，0 则表示系统为小端
10:8	PRIGROUP	R/W	优先级分组
2	SYSRESETREQ	W	写入 1 引起芯片系统复位
1	VECTCLRACTIVE	W	写入 1 清除异常的激活状态
0	VECTRESET	W	写入 1 引起处理器局部复位，但不复位外设

3. 系统处理优先级寄存器

SHPR（System Handler Priority Register，系统处理优先级寄存器）共四个：SHPR1 到 SHPR3。SHPR 的位域定义与中断优先级寄存器定义相同，差别在于 SHPR 用于系统异常。这些寄存器并未实现所有的位，已实现的位及其定义如表 5.42 所示。

表 5.42 系统处理优先级寄存器

地址	名称	类型	描述
0xE000ED18	SHPR1[7:0]	R/W	存储器管理错误的优先级
0xE000ED19	SHPR1[15:8]	R/W	总线错误的优先级
0xE000ED1A	SHPR1[23:16]	R/W	用法错误的优先级
0xE000ED1B	SHPR1[31:24]	—	未实现
0xE000ED1C	SHPR2[7:0]	—	未实现
0xE000ED1D	SHPR2[15:8]	—	未实现
0xE000ED1E	SHPR2[23:16]	—	未实现
0xE000ED1F	SHPR2[31:24]	R/W	SVC 的优先级
0xE000ED20	SHPR3[7:0]	R/W	调试监控的优先级
0xE000ED21	SHPR3[15:8]	—	—
0xE000ED22	SHPR3[23:16]	R/W	PendSV 的优先级
0xE000ED23	SHPR3[31:24]	R/W	SysTick 的优先级

4. 系统处理控制和状态寄存器

SHCSR（System Handler Control and State Register，系统处理控制和状态寄存器）可以使能的异常包括：用法错误、存储器管理错误和总线错误异常。上述异常及 SVC 异常的挂起状态和多数系统异常的激活状态也可从 SHCSR 获得，SHCSR 的位定义如表 5.43 所示。

表 5.43 系统处理控制和状态寄存器

位	名称	类型	描述
18	USGFAULTENA	R/W	用法错误异常使能
17	BUSFAULTENA	R/W	总线错误异常使能
16	MEMFAULTENA	R/W	存储器管理错误异常使能
15	SVCALLPENDED	R/W	指示 SVC 是否处于挂起状态
14	BUSFAULTPENDED	R/W	指示总线错误异常是否处于挂起状态
13	MEMFAULTPENDED	R/W	指示存储器管理错误异常是否处于挂起状态
12	USGFAULTPENDED	R/W	指示用法错误异常是否处于挂起状态
11	SYSTICKACT	R/W	指示 SysTick 异常是否激活
10	PENDSVACT	R/W	指示 PendSV 异常是否激活
8	MONITORACT	R/W	指示调试监控异常是否激活
7	SVCALLACT	R/W	指示 SVC 异常是否激活
3	USGFAULTACT	R/W	指示用法错误异常是否激活
1	BUSFAULTACT	R/W	指示总线错误异常是否激活
0	MEMFAULTACT	R/W	指示存储器管理异常是否激活

5.6 习题

5.1 计算机中的“ISA”和“μarch”各是什么意思？两者之间有何联系？

5.2 请简述哈佛结构的主要优缺点。

5.3 TCM 与高速缓存 Cache 有什么区别？

5.4 什么是饱和运算？试举例说明采用饱和运算的必要性。

5.5 ARM 指令集、Thumb 指令集和 Thumb-2 指令集之间的主要区别是什么？

5.6 MMU 和 MPU 的功能有何异同？

5.7 Cortex-R5、R7、R8 和 R52 处理器中，采用异构双核或者异构多核结构的主要目的是什么？

5.8 除了可以选配 FPU 以外，Cortex-M4 与 Cortex-M3 在指令功能上还有哪些不同？

5.9 Cortex-M 系列处理器定义的存储器映射关系是固定不变的，这样做有何利弊？

5.10 Cortex-M3 与 Cortex-M4 使用两个堆栈的目的是什么？在中断响应时，程序断点和程序状态寄存器的内容保存在哪个堆栈中？

5.11 Cortex-M3/M4 的 CODE 区选用总线互连矩阵与总线复用器有什么区别？

5.12 有些芯片制造商将所有的数据集中存储在 SRAM 区，试分析这种方案的利弊。

5.13 Cortex-M3/M4 从 SRAM 域读取指令执行时有什么缺点？

5.14 I-Code 和 D-Code 总线全部连接到同一片 Flash 芯片上会有什么问题？

5.15 私有外设总线（Private Peripheral Bus，PPB）基于哪种总线协议，有何特点？

5.16 如果非特权线程试图访问内核私有区域，将会导致哪一类异常？如果 Cortex-M3 使用了一条 SIMD 运算指令，结果又将如何？

5.17 在 Cortex-M3/M4 中，寄存器 R0~R12 有何异同？如果这些寄存器都是空闲的，你觉得首先使用哪些？为什么？

5.18 写出如下术语的中英文全称：NVIC、WIC、SCB、FPU、DWT、ETM。

5.19 某段程序需要跳转到 0x0100 0000 执行，有人写了如下两行汇编指令代码：

```
MOV     R0,     #0x0100 0000
MOV     R15,    R0
```

请问这样会有什么问题？

5.20 请说明特殊寄存器 PRIMASK 和 FAULTMASK 寄存器的异同。

5.21 Cortex-M 系列处理器的异常处理中有哪几个固定不可修改的系统异常优先级？

5.22 某基于 Cortex-M4 的 SOC 芯片共有 64 级外部中断，BASEPRI 寄存器的宽度共有几位？如果想屏蔽所有优先级大于 16 的中断，请写出对 BASEPRI 寄存器进行设置的汇编指令。如果想屏蔽所有优先级大于 0 的中断，又该如何设置？

5.23 有人写了一段对 Cortex-M4 的进程栈进行初始化的代码，其中 PSP 的初始值设为 0x8765 4321，并且使用了如下一条语句：“MOV PSP, R0”对 PSP 进行赋值（其中 R0=0x8765 4321）。这样做存在哪些问题？请逐一说明。

5.24 如果堆栈采用了双字对齐方式，当出现异常时，若需要压栈数据只有奇数个字，请简述处理器是如何实现双字对齐的？

5.25 在特权线程模式下如何切换到非特权线程模式？在非特权线程模式下能否采用类似方法切换到特权线程模式？为什么？

5.26 为何 Cortex-M 系列处理器中外部中断的类型号从 16 开始编号，而不是从 0 开始？

5.27 字节不变大端（BE-8）和字不变大端（BE-32）的区别是什么？

5.28 什么情况下需要程序员自己进行大端和小端的数据转换？

5.29 Cortex-M3 存储空间的哪些区域支持位段（bit-band）操作？

5.30 举例说明位段操作有哪些好处？

5.31 写出利用位段操作读取 0x4000 1000 的第 3 位的代码。

5.32 存储器访问属性包括哪些？

5.33 Cortex-M 处理器内部的写缓冲在什么情况下会带来好处？

5.34 什么情况下需要在程序中使用排他访问指令？

5.35 Cortex-M 系列处理器不会改变代码的执行顺序，因而不需要存储器屏障指令，这个观点对吗？为什么？

5.36 处理器进入异常处理子程序之前保护断点需要把哪些寄存器的值保护起来？

5.37 为何异常向量是 32 比特长度？

5.38 解释 Cortex-M 处理器的中断优先级分组机制。

5.39 解释向量表重定位机制。

5.40 中断使能和中断禁止通过两个寄存器进行配置有什么好处？

5.41 假设某 MCU 实际上有 16 个外部中断，试分析当 AIRCR 中位[10:8]分别为 3、4 和 5 时，分组优先级和抢占优先级的数量（可画图表示）。

5.42 假设某 MCU 实际上有 32 个外部中断，如果中断向量表需要重定位，试分析中断向量表合法的起始位置是哪些（说明需要对齐的边界）。