一本系统性探讨嵌入式设计思想的书

玩转嵌入式多任务程序设计

——RT-Thread 权威指南

DEMO

傻孩子图书工作室作品


操作系统只是一个工具，就好比画笔一样，你对画笔的结构研究的再透彻，对如何作画帮助也是非常有限的。
操作系统并不能保证实时性？！
操作系统
如何确保
系统实时性
FSM
如何实现
How
什么是多任务
What
什么是实时性
What
多任务 | 实时性

目录

什么是“上下文”

[Put Introduction Here]

1.1 关于内核你必须知道的事情

1.1.1 内核（Core）和中央处理器（CPU）是什么关系

从现代意义上的计算机诞生至今，无论规模如何，计算机的五大基本组成部分就未曾改变过：控制器（Controller）、运算器（ALU）、存储器（Memory）、输入设备（Input）和输出设备（Output）——麻雀虽小五脏俱全，嵌入式微控制器（MCU）也是如此。

虽然计算机科学中，我们把控制器和运算器在一起合称中央处理单元（CPU），但从嵌入式的视角看来，PC 机的每一个组成部分几乎无处不是嵌入式系统——即便是 PC 机使用的 CPU 本身也是一个 MCU——内其部也包含了计算机系统的五大组成部分。为了避免由术语系统的模糊造成的麻烦，我们习惯上将 MCU 中的运算器和控制器合称为内核（Core），它仅仅负责指令的执行和算术逻辑运算——ARM 公司的产品从早期的 ARM7、ARM9 到后来的 Cortex M 系列和 A 系列都是这样的内核。

现在主流 PC 机 CPU 使用的都是 x86 指令集，属于复杂指令集计算机（CISC）。其中，每一条复杂指令集指令都是通过众多微指令（Micro Instruction）所编写的程序来实现的。实际上，在这些 CPU 的核心部分都包含有一个或多个使用精简指令集（RISC）的内核，这些内核的指令就是微指令，CPU 正是通过使用微指令编写的程序来解释和执行复杂指令集指令的。这些小的内核工作当然也需要存储器、输入设备和输出设备。因此我们说，中央处理器实际上已经是一个最小的计算机系统了。

1.1.2 内核指令执行总共分几步

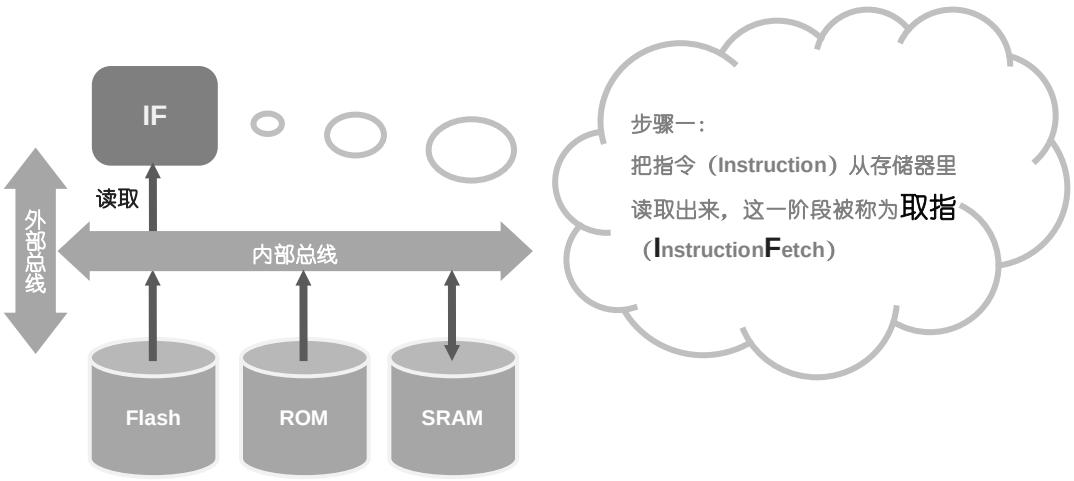
内核负责指令（Instruction）的执行。我们的程序无论用何种高级语言编写而成——C、C++还是罕见的 Basic 语言——最终都会被编译器翻译成内核能直接执行的机器码(也就是指令)、保存在存储器中交由内核来执行。那么问题来了：内核执行指令总共分几步呢？

- 步骤一：将指令从存储器里读取出来

指令（也就是程序）是保存在各种各样的存储器中的，内部的常见有 Flash，SRAM，ROM，外部的常见有 SDRAM、Nand-Flash 等等。无论是何种存储器、内部的还是外部的，它们都像是像葡萄藤上的葡萄一样挂接在总线上（如**图 1.2.1** 所示）——内部的挂接在内部总线上，外部的挂接在外部总线上——最终内核都是通过总线（BUS）来对它们进行访问的。

内核通常有一个专门的逻辑电路负责从总线上读取指令，简称取指（Instruction Fetch），习惯上我们用取址的英文单词首字母组成的缩写 IF 来标记这个部分。又由于取指是内核流水线的第一个阶段（Stage），我们称其为取指阶段（IF Stage）。简单来说，取指就是通过总线读取存储器中的指令，它对总线的操作显然是只读的，**图 1.2.1** 展示的就是内核在取指阶段所涉及到的所有要素。

图 1.2.1 将指令从存储器里读取出来



AMBA（Advanced Microcontroller Bus Architecture）是 ARM 公司提出的总线标准，由于 ARM 在嵌入式领域的影响力，AMBA 已经成为事实上的行业标准而被人们所遵循。AMBA 是一个主从（M/S）结构的总线，由主机（Master）发起对从机（Slave）的访问请求，理论上一个主机可以挂接任意数量的从机。由于 AMBA 支持在同一个总线上存在多个主机，因而当多个主机同时对同一个从机进行访问时，从机将根据事先规定好的策略进行仲裁（Arbitration）。AMBA 3 提供两个实际的总线协议，AHB 和 AHB-Lite，后者是前者的裁剪版本，在低端内核中使用，例如 Cortex M0/M0+。对程序员来说，总线的物理结构是“半透明”的：一方面程序员需要知晓目标主机（Master）上具体挂接了哪些从机（Slave）；另一方面，在编写程序时程序员看到的只是一个扁平的地址空间，通过地址就可以直接访问对应的外设。

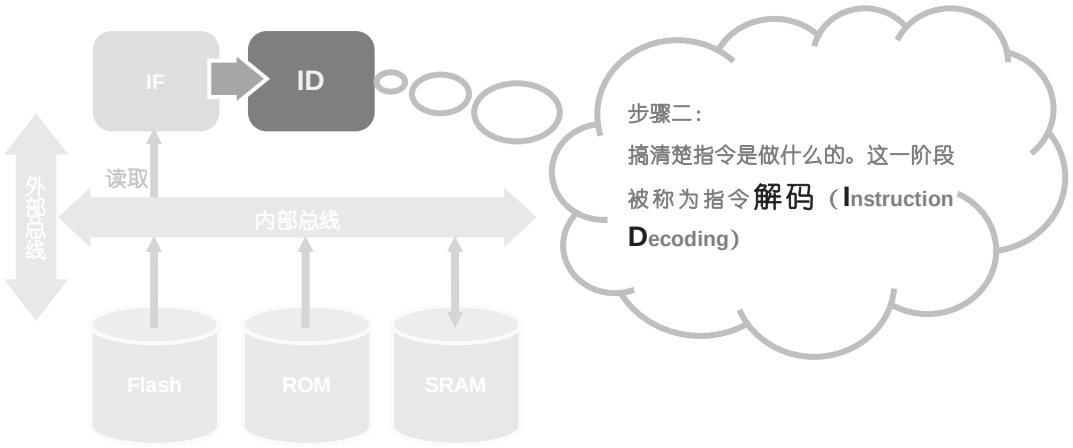
常见的主机有内核、DMA、USB、以太网控制器等等，主机可以根据自己的需要主动的访问从机；常见的从机有，各类存储器，定时器（Timer）、通信类外设（SPI、SDIO、USART、I2C）等等。需要注意的是，不能想当然的认为每个主机都能访问所有的从机——对成本敏感的 MCU 来说，这样做的成本是巨大的——所以，通常 MCU 的厂商会在芯片的数据手册上提供一个主机和从机之间的链接关系。当主机对一个没有链接的从机进行访问时，其结果是不确定的，要么读取到的都是 0，要么直接发生 Hardfault 之类的异常、严重的直接当机。

常见的内部存储器有 SRAM、ROM、Flash 等等，他们是直接挂接在 AHB 总线上的。外部存储器诸如 Nand-Flash、SDRAM 等等是通过外部总线接口（EBI，External Bus Interface）连接在 AHB 上，EBI 通常在 AHB 地址空间中占用很大一块，用于将外部存储器直接映射到其中，方便用户程序直接访问。程序是否可以在某个存储器中运行完全是由内核的总线链接方式所决定的。例如，假设内核用于取指（IF）的主机（Master）连接了 EBI，那么程序就可以在外部存储器中执行，反之则不行。

- 步骤二：搞清楚指令是做什么的

当内核读取到所需的指令后，就着手搞清楚这个指令要求内核做什么事情，具体包括搞清楚指令会用到哪些内核寄存器，是普通的数值运算还是对总线进行读取等等——习惯上我们把这一过程称为指令解码（Instruction Decoding），以缩写 ID 进行标注。指令解码是内核流水线的第二个阶段，因此又称为解码阶段（ID Stage）。**图 1.1.2** 展示了 ID 阶段在整个芯片架构中所处的位置，深色部分的 ID 阶段位于 IF 阶段的后面，并未与图中灰色的其它部分发生关联。指令解码是内核的一个内部过程，主要为指令的执行做必要的准备工作，ARM Cortex M0+甚至让其与 IF 阶段合用同一个内核时钟，只不过 IF 阶段使用的是时钟的上升沿，而 ID 阶段使用的是时钟的下降沿。

图 1.1.2 搞清楚指令是做什么的

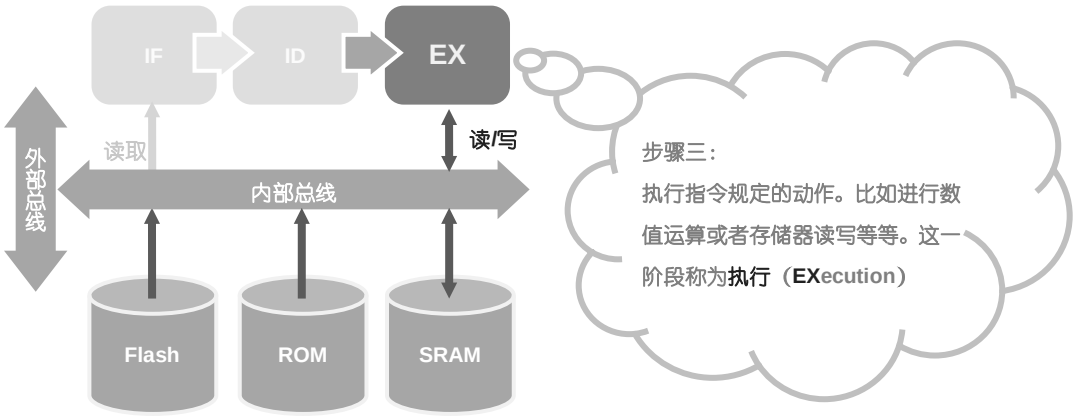


- 步骤三：执行指令要求的动作

在搞清楚指令是做什么以后，内核就进入了指令的执行阶段（Execution），一般缩写为 EX。在某些

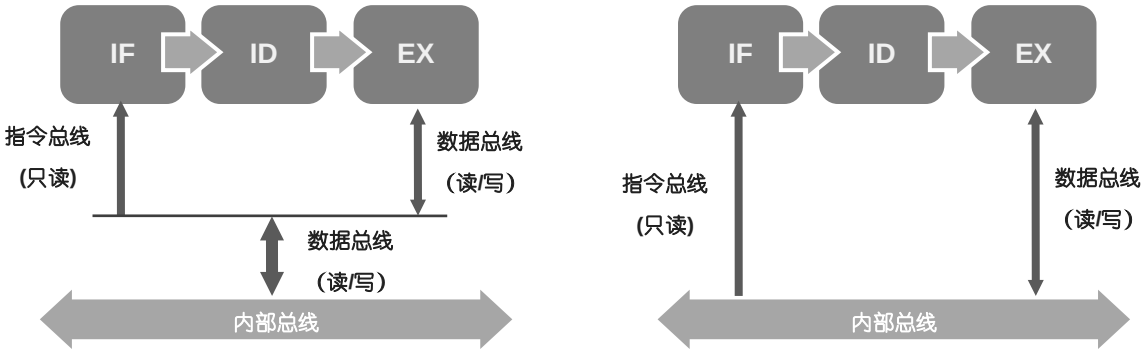
低端的内核比如 Cortex M0 / M0+，M3 和 M4 上，EX 是一个独立的流水线阶段，此时，IF、ID 和 EX 就组成了最基本的三级流水线结构；在追求指令吞吐量的内核流水线上，指令的执行（EX）往往被进一步的拆分以追求流水线的效率，但无论如何拆分，EX 作为整体所完成的功能并无不同，因此我们将不再对这一话题继续展开。

图 1.1.3 执行指令规定的动作



无论看起来如何神秘，EX 阶段所做的事情主要分为两大类：数据的存取和数据的运算。数据的存取简单来说又分为：1）从外部存储器读取数据到内核寄存器；2）将内核寄存器的数据写回到外部存储器；3）在内核寄存器之间进行数据的拷贝和移动。除 3）是内核内部的操作以外，1）和 2）都要涉及到针对总线（BUS）的“主动”访问，因此 EX 阶段明确需要一个总线访问的主机接口（Bus Master）。

图 1.1.4 哈弗结构和冯诺依曼结构



a. 冯诺依曼结构　　**b. 哈弗结构**

● “哈弗结构”和“冯诺依曼结构”是怎么回事？

通过前面的讨论我们知道，IF 阶段需要一个只读的总线接口，用于从存储器上读取指令，这是一个主机接口；由于执行指令的要求，EX 阶段需要一个总线接口，用于“从存储器上读取数据到内核寄存器”或“将内核寄存器中的数据写回存储器”，这也是一个主机接口。于是，计算机架构史上最著名的两大结构：哈佛结构（Harvard Structure）和冯诺依曼结构（Von Neumann Structure）就被提了出来。

哈弗结构为 IF 和 EX 提供了独立的总线接口(如**图 1.1.4 b** 所示),分别称为指令总线接口(Instruction Bus Interface）和地址总线（Data Bus Interface)。由于指令的读取和数据的访问使用的是不同的主机

（Master），因此，当程序和数据分别位于不同的存储器时，这两个总线接口的数据访问是并行的，能够为内核提供总线的理论最大带宽，从而实现内核的高吞吐量。常见的哈弗结构 MCU，如 Cortex M3、M4，其程序往往保存在 Flash 中，而数据则保存在 SRAM 中，在这种情况下，内核通常能够达到较高的性能。

冯诺依曼结构强调，指令只是一种特殊的数据，因而让 IF 和 EX 阶段共用同一个总线接口（如**图 1.1.4 a** 所示），这种设计一方面为程序设计带来了巨大的灵活性，另一方面也由于节省了额外的总线接口从而获得了较小的内核尺寸。缺点是明显的，由于 IF 和 EX 共享同一个总线接口，内核的实际指令吞吐量将受到一定的限制——这一限制就相当于哈弗结构的 MCU，其程序和数据始终保存在同一个存储器上。ARM Cortex M0/M0+和鲜有耳闻的 M1 都是冯诺依曼结构的内核，以换取内核尺寸和性能的折中。

请摒弃“存在即合理”的思维定势，让我们来认真思考下两种架构的存在意义——其实它们分别代表了计算机科学中常见的两大策略：“**用空间换时间**”和“**用时间换空间**”。从冯诺依曼结构的角度来考虑，由于其 IF 和 EX 共享同一个总线接口，以牺牲总线访问效率为代价在空间上节省了内核面积——这是典型的用时间换空间的策略——因此，冯诺依曼结构非常适合对内核尺寸敏感对性能没有特别要求的低端 MCU；从哈弗结构的角度来考虑，由于其为 IF 和 EX 分别提供了专用的总线接口，从而以牺牲内核面积为代价在时间上提高了内核单位时间内的吞吐能力——这是典型的用空间换时间的策略——因此哈弗结构非常适合对性能有所要求而对内核面积有所容忍的应用场合。

EX 阶段所能做另外一个事情就是数值运算，其功能的核心是运算器，又被称为数字逻辑运算单元（ALU），本质上来说，它就是个非常低级的计算器，只会做一些整数的加减法、逻辑与或非运算和非常呆板的数值比较——如果你足够细心会发现，很多内核甚至非常自豪的宣称它们将硬件乘法器作为了“标配”，这就好比卖笔记本电脑商家在 2012 年前后宣称“标配了”固态硬盘（SSD）一样（这段时间机械硬盘仍然占主导地位），暗示了乘法器根本不是 ALU 的基本组成部分，因而可以作为宣传的噱头大书特书——更不用说，硬件除法器和 DSP 协处理器了，这些都是“高配”的 MCU 才会有的。在随后的章节，我们还会对 ALU 进行详细讨论。总结来说，“数据的存取”和“低级的计算器功能”构成了流水线 EX 阶段的基本功能。

最后，我们回过头来看“读取指令的 IF”、“解码指令的 ID”和“执行指令的 EX”，可以发现：内核既不神秘，也不高级，它是非常具体的实实在在的结构，每个阶段所做的事情都非常简单。我们要在心里放下对内核的敬畏，尝试去体会和理解内核的结构，这对日后理解系统的行为是非常有必要的。

了解了程序执行的基本步骤，我们很自然的就引入了“指令流”（Instruction Flow）的概念，它不光由穿过流水线的指令组成，也实际定义了程序的执行流程。容易想到，除了从头到尾顺序流动外，指令流还有分支的概念，下面我们来一起看看程序的分支跳转是如何实现的。

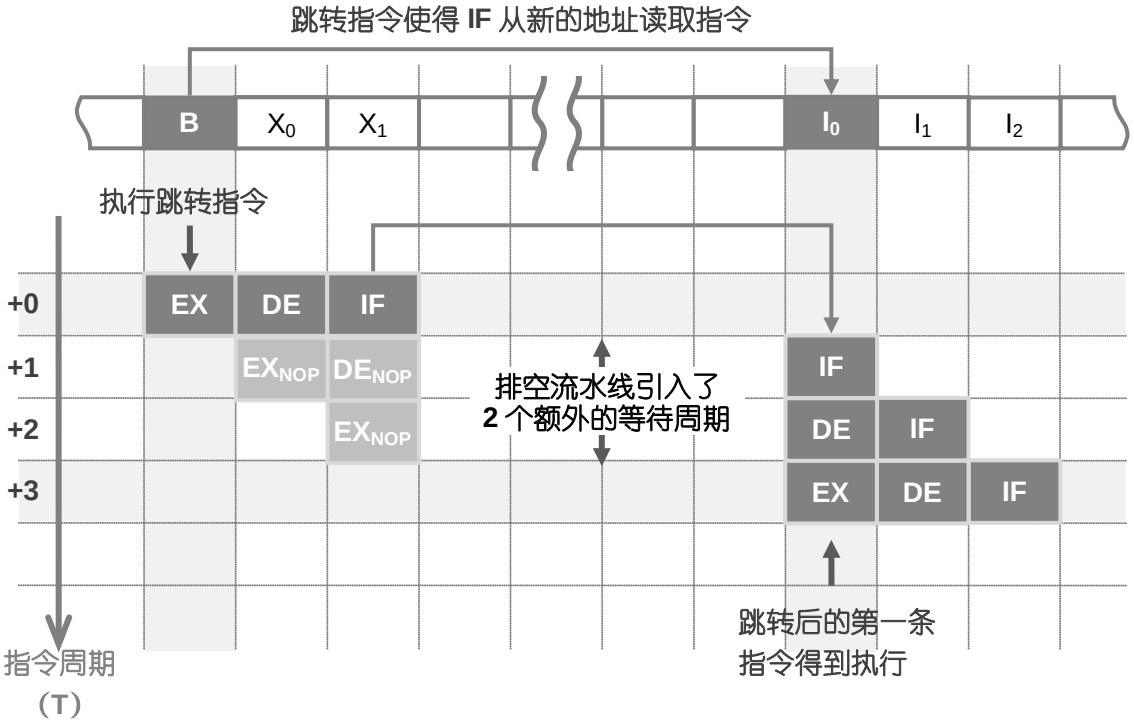
1.2 程序分支是怎么实现的

程序的顺序执行是指内核 IF 阶段顺着存储器的地址依次读取指令并执行的过程，如果中途因为某些原因突然跳转到别的地址去执行，这就叫程序的分支（Branch）。导致分支的原因主要有两大类：1）由条件跳转或者无条件跳转指令导致的分支；以及 2）由中断/异常处理导致的分支。为了方便讨论，我们不妨把前者称为“跳转”而后者称为“中断”。

虽然导致程序分支的原因不同，但内核 IF 阶段对“跳转”和“中断”的处理方法却是相同的。**图 1.2.1** 以跳转为例，展示了三级流水线内核在程序分支时的行为。图中纵轴表示时间（T），沿着自上而下的顺序内核的指令周期是递增的；图中横轴虽然没有特别标注，但很容易看出它表示的是地址空间，地址以从左到右的顺序递增，指令以小方格的形式对齐在地址空间中。

1. 在跳转指令开始执行的时刻（+0），流水线的三个阶段 IF、ID 和 EX 分别对齐到其所对应的指令：EX 阶段对应指令 B（B 是“分支”的英文单词 branch 的缩写），ID 和 IF 则顺次对应 B 随后的两条指令。这表示：IF 刚刚读取 B 后的第二条指令（图中 X_1），DE 正在解码紧随着 B 后的第一条指令（图中 X_0），而当前 EX 正在执行的是分支（Branch）指令。
2. 由于分支指令的执行，在随后的第一个时钟周期内（+1），IF 立即去读取分支目标地址处的指令（I_0），而先前已经在流水线内的两条指令（原先在 IF 和 ID 阶段的两条指令）则由于跳转指令的存在，逻辑上需要被忽略掉，因而内核直接将其无效化了——可以理解为直接将他们替换为 NOP 指令（图中以浅灰色填充的矩形框表示）。
3. 在分支指令执行后的第二个时钟周期（+2），IF 继续向后读取指令（I_1），目标指令（I_0）则进入解码阶段（ID），而 EX 阶段则处于排空阶段（等效于执行 NOP 指令）。
4. 在分支指令执行后的第三个时钟周期（+3），流水线继续正常向后推进：IF 读取指令（I_2），ID 解码指令（I_1），目标指令（I_0）进入执行阶段（EX）。至此，在跳转指令后的第三个周期，目标指令终于得到了执行。

图 1.2.1 三级流水线在跳转分支指令下的行为



对于上述模型，如果我们把程序存储器看做是一个巨大的平面的话，内核流水线执行程序更像是一个在该平面上奔驰的“固定长度”的“贪吃蛇”，只不过在程序跳转的时候，跳转语句就成为一个“任意门”，而目标程序所在的位置就是“任意门”的另外一端。从这一角度理解、重新审视**图 1.2.1** 是不是有了更形象的感官呢？

上述过程不仅描述了流水线在程序分支时的行为，更重要的是解释了为什么三级流水线为什么会在分支指令之后、目标指令执行之前引入额外的两个周期延迟——其实内核从未停歇，只不过排空流水线内已有的内容需要时间而已——三级流水线需要 2 个时钟周期，以此类推，5 级流水线就需要 4 个时钟周期。

IF 阶段从什么地址读取指令是由 PC 指针控制的，修改其值就可以实现程序的分支。

1.2.1 函数调用是怎么实现的

单纯依靠修改 PC 指针只能让程序像“满是 goto 写成的”一样跳来跳去且“有去无回”，如果要实现“函数调用”的效果，就需要跳转“有去有回”——简单的说就是内核在跳转到目标地址执行程序的同时，还要记录下紧接着“起跳点”的下一条指令的地址（也就是“返回点”的地址），以方便函数执行完成时返回。

原理上，我们可以借助栈来实现函数返回地址的保存，具体步骤如下：

1. 将返回点的地址压栈；
2. 修改 PC 指针，跳转到目标地址执行；
3. 返回时，直接将返回地址从栈中弹出到 PC 指针，实现函数的返回

虽然使用栈原理简单，支持函数的“递归”和“子函数”的调用且调用深度仅受栈深度的限制，但单纯使用栈来实现存在效率上的缺陷——结合**图 1.1.3** 容易发现，栈保存于 SRAM 中，而 SRAM 是挂接在总线上的从机（Slave），内核对栈的访问必然要经过总线，如果流水线 EX 阶段的主机接口（Master）和别的主机正好同时访问同一块 SRAM（比如 DMA 正在操作 SRAM 进行数据搬运），内核当前的操作就会收到干扰！——简而言之：多了一重访问在效率上就多了一重不确定性。

如果一个子函数（subroutine）不再调用别的函数，习惯上就被称之为叶子函数（leaf subroutine）。实际应用中，编译器会生成大量的叶子函数，程序的热点（hot point）往往也集中在叶子函数里，比如，存储器的拷贝函数即是程序的热点，也是叶子函数。这里，热点是指大量占用处理器时间的代码小片段。以下面的函数为例：

```
void u32_mem_copy(uint32_t *pwSrc, uint32_t *pwDes, uint16_t hwSize)
{
    do {
        *pwDes++ = *pwSrc++;
    } while(--hwSize);
}
```

为了提高处理程序热点时的效率，内核引入了一个硬件寄存器 LR（Link Register），专门用于保存函数的返回地址。LR 本质上相当于一个深度为 1 的硬件栈，支持且仅支持 1 级函数调用。借助 LR，内核对叶子函数的调用过程如下：

1. 将返回点的地址放入 LR 中（这是由跳转指令自动完成的）；
2. 修改 PC 指针，跳转到目标地址执行（这实际上也是同一条跳转指令完成的）；
3. 返回时，直接将 LR 的值赋值给到 PC 指针，实现函数的返回

在 ARM Cortex M 指令集中，B 表示跳转 branch；BL 表示跳转到固定地址，并自动将返回地址保存到 LR 寄存器中；BX 表示根据寄存器进行跳转，这里的 X 指代寄存器；BLX 表示跳转到由寄存器指定的地址，并自动将返回地址放入 LR 寄存器。

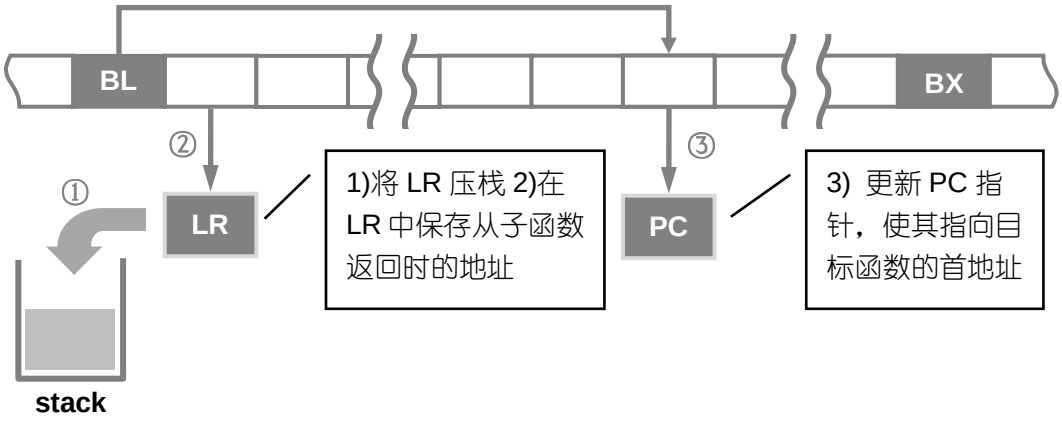
对于叶子函数的调用，由于绕开了对总线的操作的瓶颈和时间上的不确定性，内核的效率得以发挥，

时序的稳定性也得到了保证。如果目标函数不是叶子函数怎么办呢？这时，栈的使用不可避免，同时为了确保在子函数中仍然有机会利用 LR 提高叶子函数的调用效率，在函数调用时，LR 寄存器的值会被压入栈中。针对非叶子函数的、更为通用的调用过程如下（如**图 1.2.2**）所示：

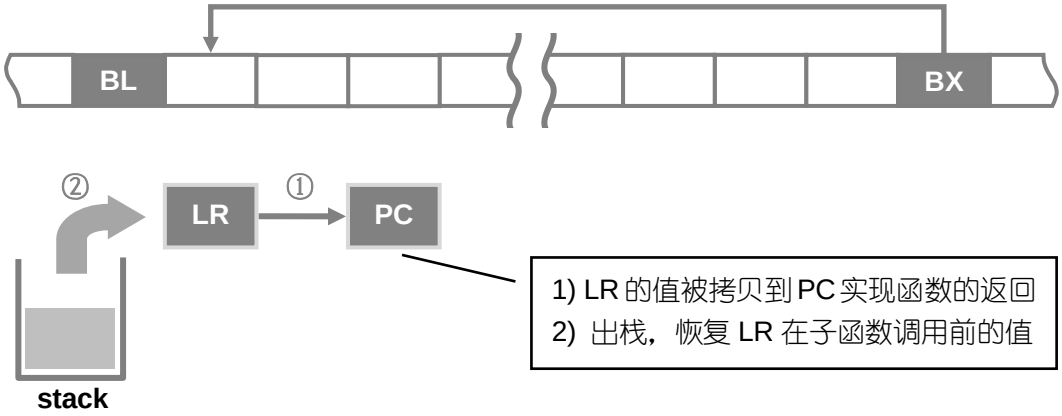
1. 将当前 LR 的值压栈；
2. 将返回点的地址放入 LR 中（这是由跳转指令自动完成的）；
3. 修改 PC 指针，跳转到目标地址执行（这实际上也是同一条跳转指令完成的）；
4. 返回时，直接将 LR 的值赋值给到 PC 指针，实现函数的返回；
5. 将先前保存在栈中的 LR 值出栈。

容易发现，相对叶子函数的调用，普通子函数的调用只是在原有步骤的基础上额外增加了对 LR 的入栈和出栈操作。需要补充说明的是，LR 是内核提供的一个辅助寄存器，在函数调用和返回时时是否对其进行压栈和出栈完全由编译器决定。因为只有编译器知道目标函数是叶子函数还是普通函数，所有的调用都是编译器事先计划好的（在“编译时刻”就确定好的）。

图 1.2.2 函数调用示意图



(a) 调用目标函数



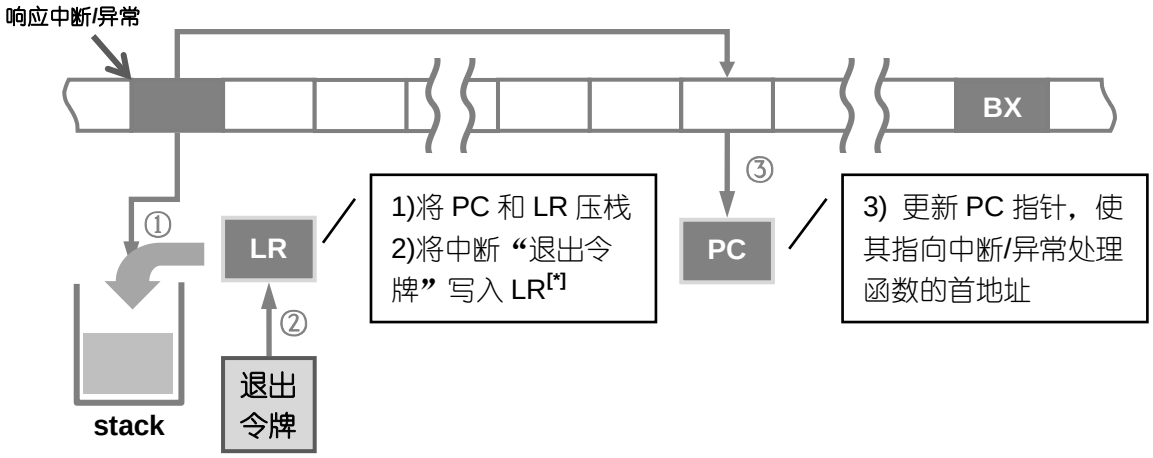
(a) 从目标函数返回

1.2.2 中断/异常处理程序是如何得到执行的

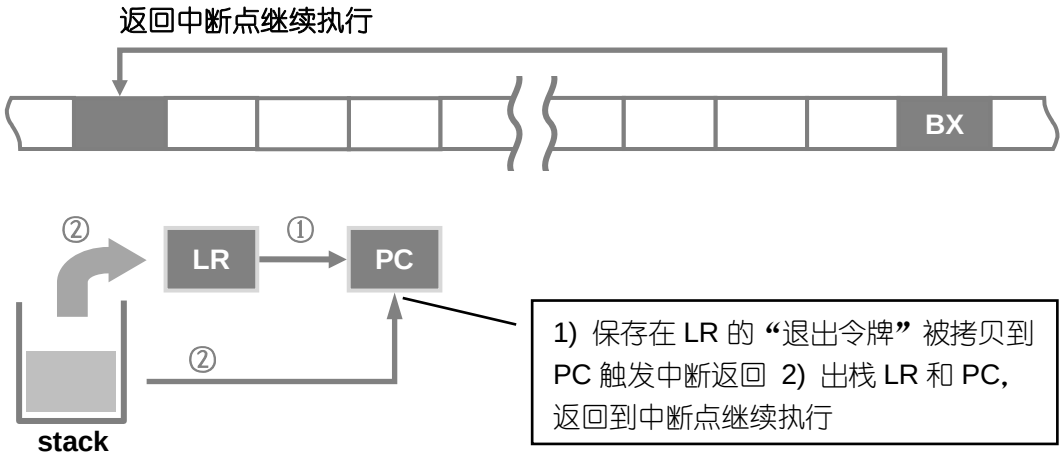
相对编译时刻（Compile-time）就“计划好”的函数调用，中断/异常处理是运行时刻（run-time）“突然”发生的程序分支。当中断/异常请求被响应时，就如同**图 1.2.1** 展示的那样，内核 IF 阶段会立即跳转到中断/异常处理程序去执行，只不过触发这一过程的契机不同：对函数调用来说，是跳转指令触发的；对中断/异常处理来说，是未被屏蔽的中断/异常请求触发的。

考察上一节中介绍的函数调用方法，借助栈和 LR，似乎我们也可以很自然的处理中断/异常处理程序的“分支”行为。然后 ARM Cortex M 系列内核却引入了“退出令牌（EXC_RETURN）”这一概念，这又是为什么呢？让我们首先来看它的中断/异常进入和退出的过程（仅讨论与 PC 和 LR 有关的部分，状态寄存器 xPSR 不做讨论），如**图 1.2.3** 所示：

图 1.2.3 ARMv6-M / ARMv7-M 架构下中断/异常处理程序的进入和退出示意图



(a) 调用中断/异常处理函数



(b) 从中断/异常处理函数返回

注：中断/异常退出令牌（EXC_RETURN）是高 4 位为 0xF 的特殊值，在中断/异常处理模式下（Handler mode），将退出令牌写入 PC 将触发中断/异常退出操作。

1. 将当前“返回点的地址”和 LR 的值压栈（这是内核自动完成的）；
2. 将“退出令牌”放入 LR 中（这是内核自动完成的）；
3. 修改 PC 指针，跳转到中断/异常处理程序的首地址处执行（这是内核自动完成的）；需要注意的是，中断处理程序中实现的函数调用仍然遵循普通的函数调用方法，并无特殊处理。

4. 返回时，直接将 LR 的值赋值给到 PC 指针，由于当前 LR 内保存的是“退出令牌”，触发内核中断/异常处理模式（这是内核自动完成的）
5. 将先前保存在栈中的 LR 值和“返回点地址”出栈，其中“返回点地址”被弹出到 PC 指针中，从先前的中断点继续执行（这是内核自动完成的）

相对普通的函数调用，中断/异常处理不仅是“分支”操作，还可能涉及到内核工作模式的变化。

为了给 RTOS 提供便利，ARM Cortex M 系列从 M0+开始对用户的操作权限提供了限制手段，提出了“特权（Privileged）操作”和“非特权（Unprivileged）操作”的概念。前者拥有最高的自由度，可以访问所有的系统资源；后者则受到了严格的限制，不仅针对普通地址空间的访问要受到 MPU 的重重审核，针对内核特殊功能寄存器的访问也受到了全面的禁止。这是从权利约束的角度来说的。

从工作模式的角度来看，ARM Cortex M 系列引入了专门用于异常/中断处理的“（异常）处理模式（Handler Mode）”以及用于普通代码执行的“线程模式（Thread Mode）”。前者所有的操作都是“特权操作”；后者的操作可以被指定为“特权”或“非特权”。这种结构显然是针对 RTOS 的应用场景设计的，“线程模式”用于执行用户任务，通常运行在“非特权”模式下，以确保任务在规定的安全范围内执行，而不会影响其它任务乃至整个系统；“（异常）处理模式”主要提供给 RTOS 的系统代码使用，用于完成上下文切换、调度、资源管理等系统级任务。

需要特别说明的是，RT-Thread 并没有将用户任务限制为“非特权”操作；裸机环境下，用户超级循环中的代码也是运行在“线程模式”中，所有的操作默认都是“特权操作”。

在某些操作系统环境下，用户任务都运行在“线程模式”下，当内核响应中断/异常请求时，系统会自动切换到“（异常）处理模式”；当内核完成了所有的中断/异常处理而退出时，内核会退回到用户任务原先所在的“线程模式”下继续执行——这一过程涉及到了模式的切换，显然只通过普通的函数调用是无法实现的。

为了解决这个问题，借助 LR，令牌的概念应运而生：在响应中断/异常时，内核自动把 LR 和程序的返回点压栈，并将令牌写入 LR 中；在随后的执行中，只要内核发现退出令牌被写入了 PC 指针（通常是中断处理程序的最后，从 LR 中拷贝到 PC 指针里），就会触发中断/异常的退出，由内核完成后续的步骤，返回原先的被打断的地点继续执行。

1.3 简陋的计算器

在前面的章节中我们提到过，内核流水线的 EX 阶段主要有两大功能：数据的存取以及数据的运算。其中数据的运算主要依靠一个“简陋的计算器”——数字逻辑运算单元 ALU，通常简称运算器。运算器不仅功能简单——只支持加法、位运算和基础的逻辑操作（与、或、非、异或等等）——使用起来也很简单。甚至连 Windows 自带的简易计算器都要比它强大许多。

1.3.1 简易计算器的使用

为了方便理解，我们首先来看看 Windows 自带简易计算器的使用方法：

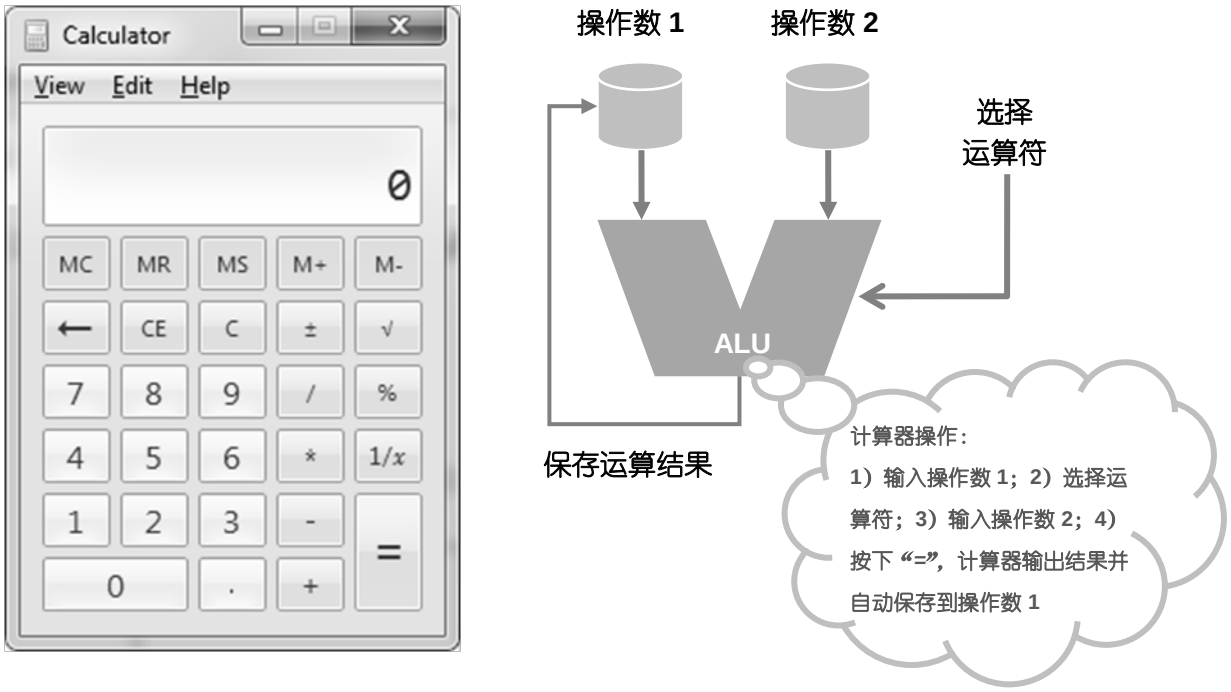
1. 输入第一个操作数；
2. 选择所要做的运算
3. 输入第二个操作数
4. 单击等于号按钮“=”，通过小窗口获取运算结果

类比 Windows 计算器，ALU 的操作步骤如下：

1. 告诉 ALU 分别从哪两个内核寄存器中获取两个操作数
2. 告诉 ALU 运算结果放到哪个内核寄存器中

3. 选择所要做的运算
4. 告诉 ALU 执行运算（相当于按下 **“=”**），ALU 会将运算结果输出到事先约定好的内核寄存器中

图 1.3.1 Windows 自带的简易计算器



容易发现，除了操作数的输入方式和观察结果的方式不同以外，ALU 和 windows 计算器在操作上是几乎一致的。只不过 Windows 默认将计算结果作为后续运算的第一个操作数，这也是计算器支持连续运算的原因，而 ALU 需要用户指定保存结果的内核寄存器，当然，如果用户将保存第一个操作数的寄存器同时也作为输出结果的寄存器，就和 Windows 计算器没有什么不同了，**图 1.3.1** 通过类比的方法体现了这一点。

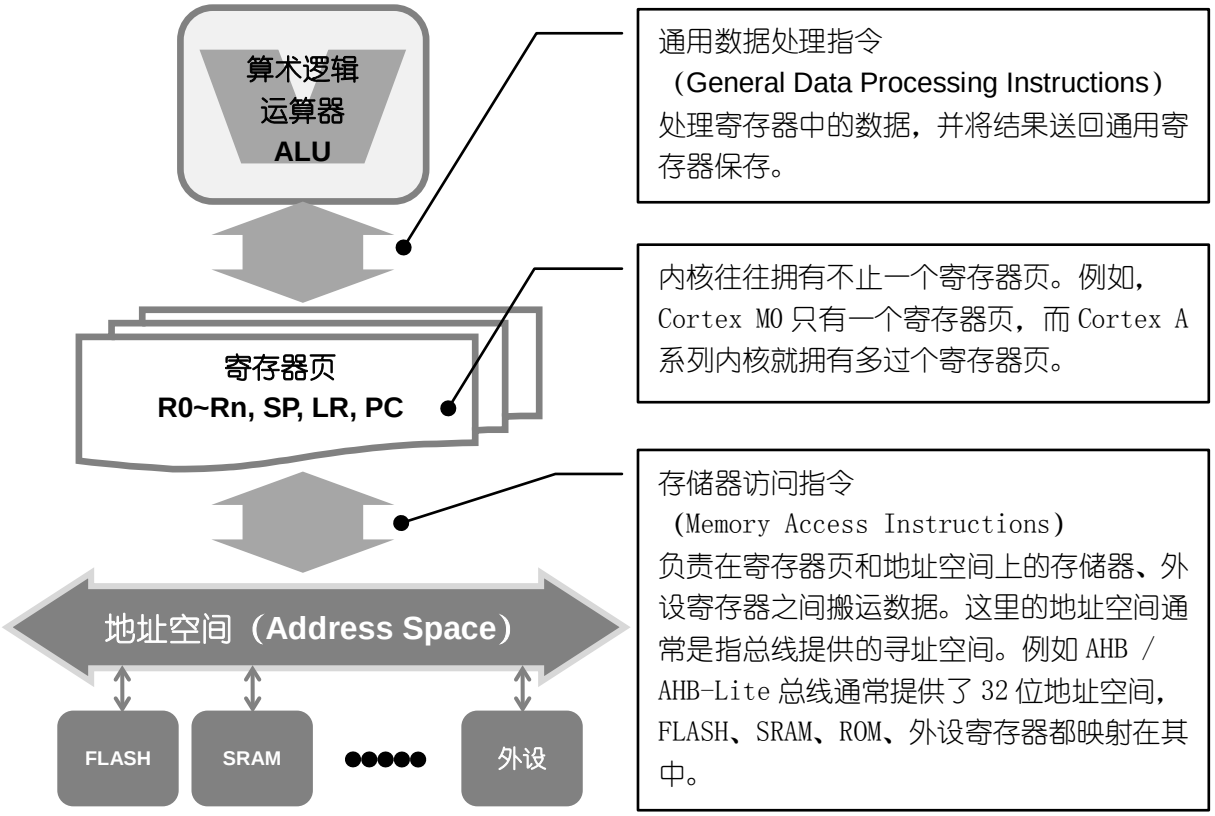
1.3.2 通用寄存器的接口作用

Windows 计算器的接口是键盘（用于输入）和显示数字的小视窗（用于输出），而对 ALU 来说输入输出的接口都是内核寄存器——不仅运算的操作数要在这些内核寄存器中读取，运算的结果也写回内核寄存器——可以说 ALU 是仅针对内核寄存器进行操作的，它既不能直接从外部存储器中读取操作数，也不能把运算结果直接输出到总线上。

为了解决这一问题，除了 ALU 相关的运算类指令（通用数据处理指令，General Data Processing Instructions）外，内核专门引入了另外一类指令（存储器访问指令，Memory Access Instructions），专门用于在由总线提供的“外部地址空间”和内核寄存器之间进行数据搬运（如**图 1.3.2** 所示），这样的设计简洁明快，分工明确。习惯上，我们把 ALU 可以直接访问的寄存器称之为“通用寄存器”，这些通用寄存器在一起形成的集合，称之为“寄存器页”。一个内核至少有一套寄存器页，但很多内核提供了多套寄存器页，具体原因我们将在后面的章节中详细讨论。

寄存器页是算术逻辑单元与外部地址空间进行数据交换的唯一接口。

图 1.3.2 寄存器页——算数逻辑单元与外部地址空间的唯一接口



1.3.3 寄存器页里都保存了什么

- 通用寄存器 Rn

寄存器页中通常提供了若干个通用寄存器供用户使用，习惯上用 R0~Rn 表示，这里 n 的取值根据内核的不同而不同，比如 ARM Cortex M 系列内核提供了 15 个通用寄存器，分别用 R0~R14 表示。那么除了保存 ALU 运算所需的操作数和操作结果，其它什么场合下会用到通用寄存器呢？

首先，我们知道，C 语言使用栈分配局部变量，在嵌入式 C 语言中，编译器会优先使用通用寄存器来保存局部变量，仅当通用寄存器无法满足要求时，才使用栈进行局部变量的分配，这样可以有效的减少总线操作（SRAM 挂接在总线上）从而提高了数据的访问效率提高了内核的运算效率。

其次，当我们使用指针时，内核实际上是通过间接寻址来实现的，常见的间接寻址都需要借助通用寄存器进行实际地址的运算和表达。例如，下面的指令分别实现了间接寻址和变址寻址：

```
STR    r1,   [r3]              ; 将 R1 中保存的数据存储到 R3 指定的地址中
STR    r2,   [r3,  #0x4]       ; 将 R2 中保存的数据存储到以 R3 为基地址，偏移量为 4 的地址中
```

显然，上述代码中，R3 是一个指针，保存的是外部存储器中某个对象的基地址，这个对象可能是数组，也可能是结构体。

再次，对于静态变量（静态局部变量和全局变量），在开启优化的情况下，编译器生成的代码并不会针对每一个语句都生成对应的总线操作，而是会根据变量在代码某个局部的使用情况尽可能利用通用寄存器对其进行优化，例如：

```
//! 4 级别的电池状态
typedefenum {
    BATTERY_EMPTY = 0,                                  //!<空电池
```

```
    BATTERY_LOW,                                        //!<电池快没电了
    BATTERY_HIGH,                                       //!<电池电量高
    BATTERY_FULL                                        //!<电池是满的
} battery_status_t;

static uint32_t s_wBatteryVoltage;                      //!<保存了当前的电池电压

//! 函数根据当前电池电压评估电池的容量情况
battery_status_testimate_battery_status(void)
{
    if (VOLTAGE_BATTERY_EMPTY <= s_wBatteryVoltage) {
        //! 电池电压低于 EMPTY 的门限
        return BATTERY_EMPTY;
    } else if (VOLTAGE_BATTERY_LOW <= s_wBatteryVoltage ) {
        //! 电池电压在 LOW 和 EMPTY 的区间内
        return BATTERY_LOW;
    } else if (VOLTAGE_BATTERY_HIGH <= s_wBatteryVoltage) {
        //! 电池电压在 HIGH 和 LOW 的区间内
        return BATTERY_HIGH;
    }
    //! 电池电压在 FULL 的区间内

    return BATTERY_FULL;
}
```

在上述例子中，静态变量 s_wBatteryVoltage 虽然在函数中被多次使用，但实际上，它仅仅涉及到了读取操作，实际上并不需要每次都通过总线操作实际读取一次变量的值，因而在开启优化的情况下，编译器所生成的代码仅仅会在函数的一开头将 s_wBatteryVoltage 读取到某个通用寄存器中，并在随后的操作中直接使用对应的通用寄存器进行比较——这就是著名的窥孔优化（Peephole Optimize），它在带来更小代码尺寸的同时，也为我们带来的不小的麻烦，例如下面的例子：

```
static uint32_t s_wDelayCounter = 0;

//! System Tick 异常处理程序
ISR(Systick_Handler)
{
    //! System Tick 的异常处理程序，已经被配置为 1ms 触发一次
    if (s_wDelayCounter) {                  //!<我们维护了一个倒计时计数器
        s_wDelayCounter--;
    }
}

//! 一个阻塞的精确毫秒延时函数
voiddelay_ms(uint32_t wDelay)
{
```

```
    s_wDelayCounter = wDelay;
    while(s_wDelayCounter);                    //!<死等延时完成
}

//! 一个 LED 闪烁的例子
void main(void)
{
    system_init();                             //!<初始化系统，包括 System tick
    while(1) {
        LED_ON();                              //!<点亮 LED
        delay_ms(500);                         //!<延迟 500ms
        LED_OFF();                             //!<熄灭 LED
        delay_ms(500);                         //!<延迟 500ms
    }
}
```

这是一个 LED 闪烁的例子，它通过 System Tick 产生了一个 1ms 的异常（Exception）请求，并通过该异常的处理程序实现了一个精确的毫秒倒计时器。原理虽然简单，在某些编译器下（比如 IAR），开启 优化或关闭优化，其运行结果却很可能是不同的——在关闭优化的情况下，LED 以 1Hz 为频率正确的闪烁；而在开启优化的情况下，LED 却是常亮的。这又是为什么呢？

通过仿真，我们发现，开启优化后，系统一旦进入函数 delay_ms(500);就不会退出，并在语句 while(s_wDelayCounter);处死循环。奇怪的是，通过观察 s_wDelayCounter 变量的值我们会发现，变量已经被正确的倒计时为 0，为什么系统仍然无法退出 while 循环呢？——这就是窥孔优化捣的鬼——编译器在尝试优化时发现 delay_ms 函数在对 s_wDelayCounter 赋值后，一直处于对该变量的读取状态，而不再有任何写入操作，换句话说，单纯从 delay_ms 函数来看，s_wDelayCounter 在赋值之后，其值是不会发生任何改变的。因而，编译器片面得出结论，认为完全没有必要每次读取操作都通过总线对变量的实际值进行读取，而是直接在赋值后，将 s_wDelayCounter 读取到某个通用寄存器 Rx 中，之后所有的 while 判断也都是基于该通用寄存器 Rx 进行的。显然 Rx 的值不会随着 s_wDelayCounter 的改变而改变，delay_ms 在窥孔优化下变成了一去不回的死循环。为了解决这一问题，我们需要在变量声明时引入关键字 volatile，用以告诉编译器：s_wDelayCounter 的值是经常变化的，应该关闭窥孔优化，每次对变量的读取操作都应该通过实际的总线操作来进行：

```
//! 引入 volatile 关键字，告知编译器关闭针对目标变量的窥孔优化
static volatile uint32_t s_wDelayCounter = 0;
```

通过观察现象证实，加入 volatile 以后，无论是何种优化方式，LED 都能正确的进行闪烁。

- 如何对 volatile 修饰的变量进行手工优化

volatile 的使用，实际上阻断了编译器利用通用寄存器对静态变量的操作进行优化，虽然能保证操作的正确性，却无法在某些可以优化的地方提升性能。例如：

```
static volatile uint32_t s_wVPort = 0;

void set_vport_u8(uint8_t chValue, uint8_t chOffset)
{
    uint32_t wMask = 0xFF <<chOffset;              //!<获取正确的掩码
    s_wVPort&= ~wMask;                            //!<步骤 1：将掩码对应的位置清零
```

```
    s_wVPort |= ((uint32_t)chValue<<chOffset);  //!<步骤 2: 设置新值到虚拟端口
}
```

由于 volatile 的存在，步骤 1 和步骤 2 这样的“读改写操作”都会独立生成针对 s_wVPort 的读写操作，因此上述代码等效为：

```
void set_vport_u8(uint8_t chValue, uint8_t chOffset)
{
    uint32_t wMask = 0xFF <<chOffset;               //!<获取正确的掩码

    //! s_wVPort&= ~wMask; 的等效展开
    uint32_t wTemp1 = s_wVPort;                     //!<步骤 1.1 读取 s_wVPort
    wTemp1&= ~wMask;                                //!<步骤 1.2 改写 wTemp1
    s_wVPort = wTemp1;                              //!<步骤 1.3 将 wTemp1 写回 s_wVPort

    //! s_wVPort |= ((uint32_t)chValue<<chOffset);的等效展开
    uint32_t wTemp1 = s_wVPort;                     //!<步骤 2.1 读取 s_wVPort
    wTemp1 |= ((uint32_t)chValue<<chOffset);        //!<步骤 2.2 改写 wTemp1
    s_wVPort = wTemp1;                              //!<步骤 2.3 将 wTemp1 写回 s_wVPort
}
```

显然，步骤 1.3 和 2.1 是多余的，我们可以手工将其优化为：

```
void set_vport_u8(uint8_t chValue, uint8_t chOffset)
{
    uint32_t wMask = 0xFF <<chOffset;               //!<获取正确的掩码

    //! 将 s_wVPort 读取到通用寄存器中（wTemp1 编译器会用通用寄存器来保存）
    uint32_t wTemp1 = s_wVPort;                     //!<步骤 1.1 读取 s_wVPort

    //! 对保存在通用寄存器中的值进行统一修改
    wTemp1&= ~wMask;                                //!<步骤 1.2 改写 wTemp1
    wTemp1 |= ((uint32_t)chValue<<chOffset);        //!<步骤 2.2 改写 wTemp1

    //! 将修改后的值写回 s_wVPort
    s_wVPort = wTemp1;                              //!<步骤 2.3 将 wTemp1 写回 s_wVPort
}
```

这就是一个手工对 volatile 修饰的变量进行局部优化的例子，本质上就是替代编译器在合适的位置使用通用寄存器对静态变量进行“手工窥孔优化”。需要注意的是，需要 volatile 进行修饰的变量通常与多任务或者中断/异常有关，因此，进行手工窥孔优化时，尤其需要注“意确保数据操作的完整性”，相关内容，我们将在第四章“什么是共享资源”进行详细分析和介绍。

volatile 的应用范围非常广泛，尤其是在嵌入式系统中，几乎所有的外设寄存器都可以表述为如下的形式：

```
//! 已知某 32 位外设寄存器的地址为 XXXXX_IO_REG_BASE_ADDRESS，则对应的寄存器可以定义为
#define XXXXX_IO_REG    ( *((volatile uint32_t *)XXXX_IO_REG_BASE_ADDRESS) )
```

考虑到这种情况，实际应用中很多针对外设寄存器的连续操作都可以通过“手工窥孔优化”来大幅度提高效率。

- 栈顶指针 SP

栈顶指针（Stack Pointer）是寄存器页的核心，用以指向系统栈的栈顶位置，某些情况下也可以作为通用寄存器来使用，例如，在 ARM Cortex M 内核中，SP 可以作为 R13 来使用。由于栈是函数式语言的核心，在操作系统中 SP 的地位举足轻重，以 RT-Thread 为例，每个用户任务都有独享的栈，任务的切换几乎就是栈的切换，也就是栈顶指针的切换，我们可以毫不夸张的说：栈顶指针就是每个任务的生命线。

- PC 和 LR

PC 指针（Program Counter）和 LR 指针（Link Return）是寄存器页的核心，用于实现流水线的执行和分支，详细内容我们在本章的开头已经详细讨论过。LR 寄存器在某些情况下也可以作为通用寄存器来使用，例如，在 ARM Cortex M 内核中，LR 可以作为 R14 来使用。

- 内核状态寄存器 SR

内核状态机寄存器（Status Register）是寄存器页的核心，用于保存 ALU 的状态标志（例如经典的 N、Z、C、V 标志）以及内核的运行状态。

对不同的内核架构来说，寄存器页中可能还包含其它寄存器，但就基本结构来说，一个标准的寄存器页通常包含，通用寄存器 R0~Rn，PC 和 LR，栈顶指针 SP，以及内核状态寄存器 SR。寄存器页是内核与外界进行据交换的唯一接口。

1.5 什么是“上下文”

“任务上下文”，简称“上下文”是英文单词“Context”的直接中文翻译，由于人们很难从字面上直观的推断“上下文”的意义，使得这一概念较为抽象。为了深入浅出的“曝光”上下文的本质，我们不妨抛开这个概念，先来讨论下任务和任务所使用到的资源。

1.5.1 从任务和资源说起

每个任务（Task）都要使用资源（Resource），不光有保存代码的存储器资源（ROM），保存变量的内存资源（RAM），还包括内核（Core）和外设资源（Peripheral）。当多个任务都必须使用同一个资源时，为了确保所有任务都有机会正确的执行，系统不得不在多个任务之间寻找某种方式共享（Share）这一资源。

由于资源是紧缺的，有时候是唯一的，因而这种共享策略必然是“排它（Exclusive）”的或者说“互斥（Mutually Exclusive）”的。僧多粥少，当所有的任务都拥挤到资源门口“吵吵嚷嚷”时，必须有人站出来协调才能维持系统的秩序，避免效率的牺牲和资源的分配的不合理。和处理人类相同问题的情形类似，这里有两种策略：

- 依靠任务自觉合作

在这种策略下，每个任务都要有“公德心”——不仅要知晓有别的任务与自己竞争，还要自觉的遵守与其它任务约定好的共享规则——无论是先到先得的排队策略，还是尊老扶幼的优先级策略——在这

种环境下，任务对资源访问的秩序完全依靠任务的“公众素质”，一旦出现了不守规矩的任务，或者是不了解规矩“新来的”，整个系统的稳定性就会受到毁灭性的打击。通常意义上多任务之间的通信和同步、数据的交换采用的就是这种方式，我们将在第四章进行详细讨论。

- 依靠第三方维护

在这种策略下，第三方服务机构被引入了进来，专门负责维护多任务之间资源的共享问题；对每个任务来说，由于服务机构的存在，它们只需要服从机构的安排就可以“自由”的使用资源，而不用关心“资源是不是共享的”、“资源是如何共享的”以及“谁和我共享这个资源”的问题——简而言之，第三方服务机构对所有的任务隐藏了资源共享的具体方法和操作细节，使每个任务都可以获得“自己独占资源”的错觉。

这就好比当我们出门叫出租车时，只需通过手机客户端发出请求即可——至于打车软件如何调度车辆、如何处理付费、如何平衡供求关系——这些问题对用户来说都是透明的（看不见的）。在这个例子中，用户下载打车软件并选择使用该软件预约出租的行为可以被视作为“服从机构安排，使用第三方机构分配的车辆”。在嵌入式系统中，一个任务要怎么做才算是“服从机构的安排”呢？

事实上，嵌入式系统中这种服从通常是强制的——任务是在不知不觉中被动的服从机构的安排，因为这些任务往往是直接建立在机构提供的基础之上，从一开始就无法对机构作出反抗。例如，操作系统提供了建立和管理任务的 API，以这种方式创造出的任务自然无法违抗（察觉）操作系统的调度。

1.5.2 服务机构的资源管理

“独占资源”、“资源管理是透明的”对任务（准确的说是任务的编写者）来说实在太诱人了——简直就是懒人的天堂啊！正因为如此，在多任务系统中，内核（Core）、内存（RAM）甚至是外设（Peripheral）这类资源的共享，通常是委托第三方——也就是操作系统——进行管理的。那么第三方是如何处理多任务与共享资源之间关系的呢？

- 服务机构只关心共享资源

任务所使用的资源很多，但并非所有资源都是共享资源，比如保存在 ROM 中的代码，这些都是每个任务独占的。需要注意的是，保存在 ROM 中的内容虽然其他任务也可以访问，但这并不能改变对应的存储区域由某个任务“永久独占”的事实，这里的独占强调的是占有，而并不排斥访问。所谓共享资源，强调的是每个任务都可以在一定时间内（Temporal）排他性（Exclusive）的占有和使用，但无法永久（Permanent）的排他性的占有和使用。比如内核，任务的执行需要排他性的占有内核资源，但每个任务都无法永久的占用内核，否则所谓的多任务就无从谈起了（这里通常假设任务的数量多于内核的数量——这也是大多数应用中常见的情形）。

- 服务机构以某一共享资源为核心考虑多任务的切换

当第三方服务机构管理某一个共享资源时，它要考虑的就是如何让多个任务能够在资源上“透明”的进行切换——当任务对资源的使用被打断时，应该：

1. 挂起/暂停任务的执行
2. 保护资源的使用现象

当任务获得资源的使用权时，应该：

1. 恢复先前保护的资源使用现场
2. 恢复任务的执行

简单说，如果把共享资源看做一个房间，那么当需要任务离开房间时，首先要让他在房间里睡着，

这样它就不知道随后发生的所有事情，然后通过拍照的方法记录房间现在的样子，以方便下次任务重新进入房间时能够恢复现场，最后直接将任务抱出房间；当需要任务重新进入房间时，先要根据之前记录的快照恢复房间的布置，然后再将任务重新抱进房间、最后唤醒。

这里，我们容易注意到：

对任务来说，每一个共享资源都有一个“工作现场”需要保护和恢复。

对任务来说，有多少共享资源就有多少工作现场。那么一个普通的操作系统任务究竟使用了哪些共享资源，对应存在哪些“工作现场”需要保护和恢复呢？

表 1.5.1 典型任务的共享资源和工作现场

共享资源	工作现场	保护/恢复	说明
系统栈	栈顶指针	保存在任务控制块中	用于处理函数调用、局部变量分配以及工作现场的保护
内核	寄存器页（R0~Rn, LR, SR, SP）中的值	压栈 / 出栈	指令的执行
协处理器 [1]	协处理器寄存器页中的值	压栈 / 出栈	各类协处理器包括：浮点运算器，DSP 协处理器等等

NOTE: 1、在没有协处理器的系统中，任务的共享资源也不包含协处理器。

需要注意的是，虽然外设（Peripheral）也是任务的共享资源，但由于外设寄存器页往往并不具有可保存和恢复的特性，因而无法将其视为工作现场。多任务之间共享外设更多的采用“合作”的方式，或者由第三方将其重新包装后伪装成非共享资源。例如，操作系统可以为每个任务都提供独立的 FIFO，从而将 USART 共享问题转化为 FIFO 的独享形式。

我们在讨论“共享资源”和“工作现场”时，有一个“陷阱”需要特别注意：不要把“工作现场”和“工作现场的容器”混为一谈。什么是容器呢？比如内核作为共享资源，它的寄存器页就是容器，而保存在寄存器页里的“内容”才是“工作现场”——这就好比，我们常说将某某寄存器入栈，真正被压入栈中的是保存在寄存器里的值而不是寄存器本身一样。

1.5.3 什么是上下文

上下文是一个集合，包含了任务所有共享资源的“工作现场”。

当操作系统对任务进行切换时，只要进行上下文的切换，就可以保证任务所有的共享资源现场都得到正确的切换，从而确保了任务的正确执行。所以，我们通常说：

任务的切换就是上下文的切换。

对照**表 1.5.1** 容易发现，一个常见的任务上下文通常由内核寄存器页、栈顶指针以及协处理器的寄存器页构成。又由于内核寄存器页（和协处理器寄存器页）实际上是通过系统栈进行保护和恢复的，而系统栈的栈顶指针是通过任务的控制块进行保存的，不难看出：

上下文的切换可以浓缩为栈顶指针的切换。

关于上下文切换的内容，我们将在第二章为您详细展开。

1.6 “空间”和“时间”的游戏

通过前面的讨论，我们知道，上下文是唯一确定任务当前执行状态的最小关键信息，任务的切换就是上下文的切换。上下文的范围由共享资源的工作现场所决定。通常情况下，寄存器页是上下文的最小容器，此时上下文的切换实际上要解决的是“内核如何利用寄存器页作为容器来承载不同任务上下文”的问题。从程序员的角度来说，上下文的切换当然是越快越好；但从芯片成的角度来说，实现最快的上下文切换往往意味着更高的代价。因此，这里也存在着一个“时间”和“空间”的协调问题。

1.6.1 以时间换空间的“上下文切换”技术

在成本敏感的 MCU 系统中，内核仅提供一套寄存器页作为上下文的容器，这就意味着不同的任务必须共享同一个容器。上下文的切换需要借助栈的力量，通过“保护现场”和“恢复现场”的方法实现任务上下文的切换。这里“保护现场”是指将当前任务的上下文从寄存器页保存到系统栈中；“恢复现场”是指从系统栈中将目标任务的上下文恢复到寄存器页中——由于借住了栈的帮助，因而上下文的切换需要消耗较多的时间。这是一个典型的以切换时间换取较小内核尺寸（空间）的策略，在成本敏感的系统中较为常见，如 ARM Cortex M 系列内核。

1.6.2 以空间换时间的硬件“多线程技”技术

在性能敏感的 MCU 系统中，内核为特定的任务——中断/异常处理程序提供了多套寄存器页，这就意味着中断/异常处理程序在上下文切换时并不需要借助栈的力量，而直接使用专用的寄存器页即可，从而节省了切换时间，提高了中断/异常请求的响应速度。内核为中断/异常处理程序提供额外的寄存器页是以牺牲内核面积为代价的，是一个典型的以空间换时间的策略，在性能敏感的高端处理器中较为常见，如 ARM Cortex A 系列内核。

1.7 由切换带来的原子性问题

在 20 世纪初叶，人们曾经一度认为原子是物质的最小组成单位，原子不可再分。虽然很快人们就发现这是一个谬误，原子不仅可以再分，由质子、中字、电子组成，事实上这些微观粒子仍然是可以继续分割的，但计算机科学借用了“原子不可再分”的说法，提出了操作原子性（Atomicity）的概念，即：

对一个由多个步骤构成的操作来说，当操作进行时，针对全部或者某些特定的任务，系统无法进行上下文切换，我们就说该操作对其它全部任务或者某些特定的任务具有原子性，是一个原子操作，反之，该操作不具备原子性，不是原子操作。

简单来说，所谓原子操作就是当这个操作执行的时候，其他任务没法打断它，就好像原子无法再分一样——如果存在任务有能力将当前的操作从中间一刀切开，我们就说这个操作不是原子操作。很容易想到，当多个任务共享统一资源时，对资源访问操作的原子性是非常值得讨论的。详细内容，我们将在第四章“什么是共享资源”中详细为您展开。

以 ARM Cortex M 为例，通常我们所说的一个内核是 8 位、16 位还是 32 位并不是指地址总线的宽

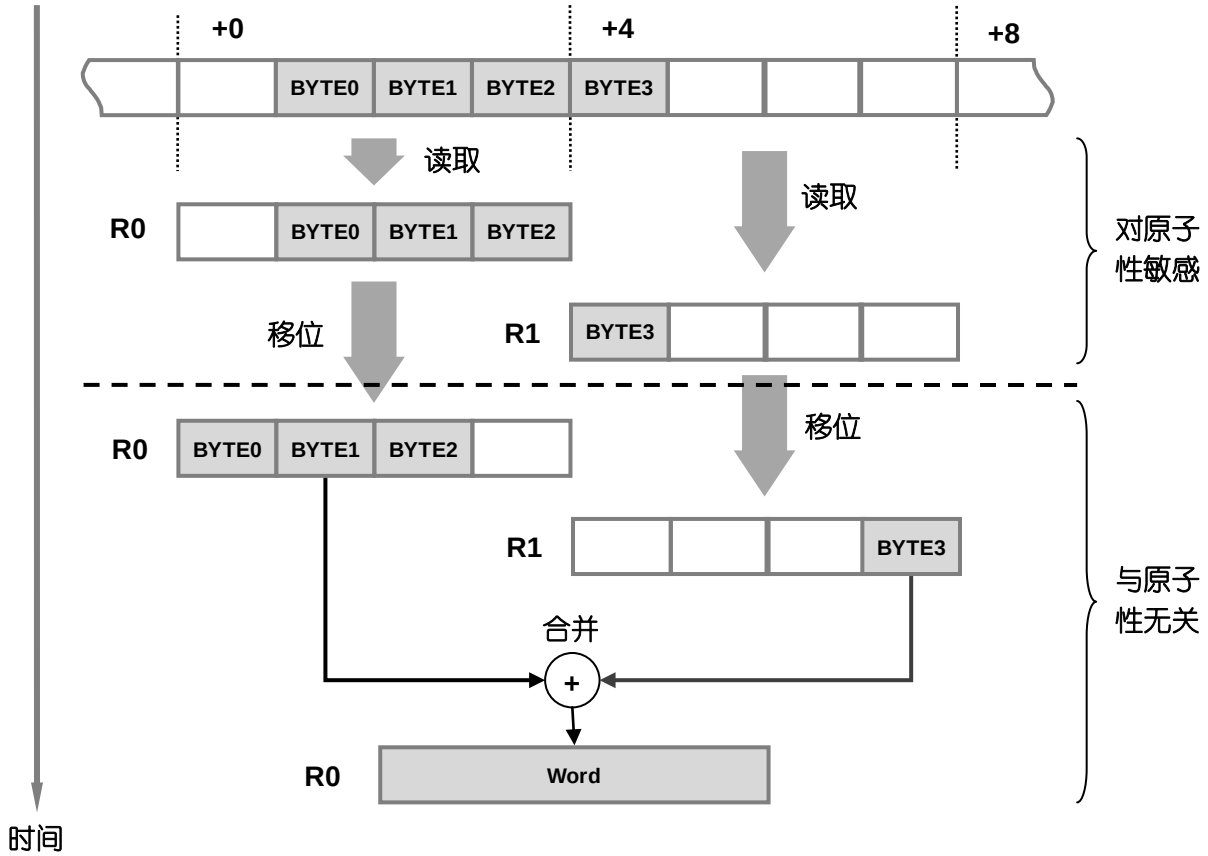
度，而是 ALU 操作数的位宽，习惯上又称之为字长。对 8 位机来说，ALU 一次可以进行 8 位运算，当我们针对一个 32 位的数据进行操作时就要拆成 4 次。对 32 位机来说，ALU 一次就可以完成 32 位的运算。比较二者的区别，除了操作次数不同以外还隐含着原子性的信息，即对 8 位机来说，操作 32 位数据要分 4 个步骤来完成，这期间如果发生中断/异常，操作是会被打断的，因此不具备原子性；对 32 位机来说，由于 ALU 一次运算的过程是不可打断的，因而针对 32 位数据的运算天然具有原子性，我们称之为天然原子性。

ALU 对相同字长数据的处理具有天然原子性。也就是说，16 位机对 16 位数据的处理具有天然原子性；64 位机对 64 位数据的处理具有天然原子性。实际应用中，天然原子性的体现还要受到数据读写对齐方式的限制：

1、 为了提升效率，内核往往规定，数据的读写地址需要对齐到字。当且仅当数据的地址对齐到字长时，内核的天然原子性才能得到表达。以 32 位机为例，当 32 位数据地址对齐到字（32 位）时，内核针对该数据的读写具有天然原子性；对于 **图 1.7.1** 所示的情形，由于 32 位数据的地址并未对齐到字，编译器可能会将针对该数据的读写需要拆解成四个步骤：1）读取变量所涉及的前一个 WORD（32 位）；2）读取变量所涉及的后一个 WORD（32 位）；3）从这两个 WORD 中提取出目标变量——针对该变量的操作由多个步骤构成，且中途有可能被别的任务打断，因而不具备原子性。

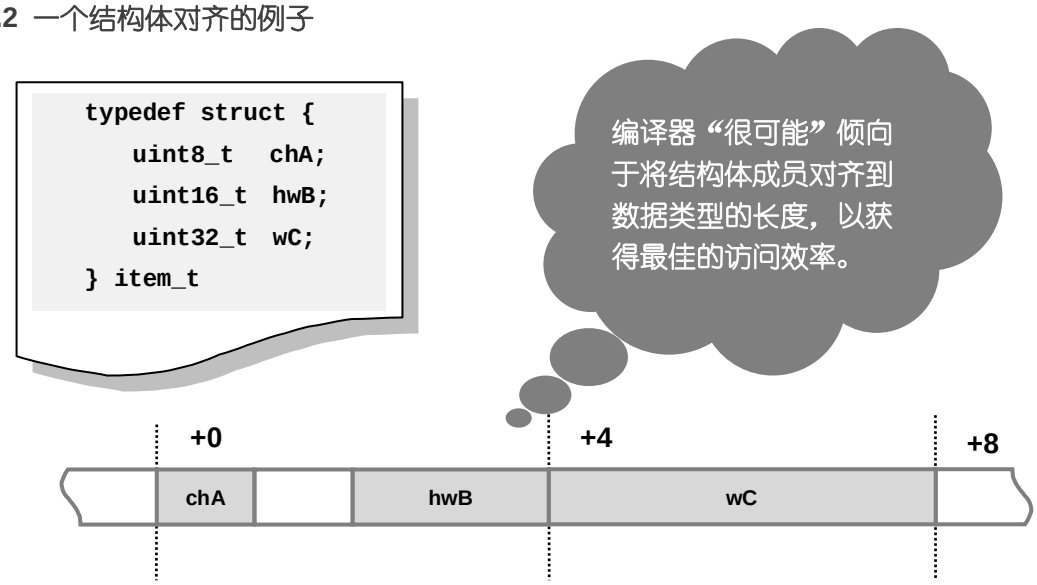
2、 对于小于字长的变量，如果也对齐到了自己的长度(比如,uint16_t 对齐到了偶数地址)字长、则内核针对该变量的操作也具有天然原子性（因为有专门的指令与之对应，例如 LDRB/STRB 针对字节读写，LDRH/STRH 针对半字存取）。很多编译器为了提高内核的访问效率，在默认情况下，对结构体的变量就采取了同样的策略——每个成员的地址都各自对齐到了与自己类型相同的字长（如图

图 1.7.1 针对未对齐到 4 字节地址的 WORD 操作



1.7.2 所示）

图 **1.7.2** 一个结构体对齐的例子



2

什么是“任务”

[Put Introduction Here]

2.1 上下文切换总共有几类

上一章我们通过研究“上下文的容器”间接的了解了什么是上下文，并仔细的剖析了上下文的构成要素。通过这些讨论我们很容易想到，其实在处理器眼中是没有我们所说的“任务”这一概念的，它能看到的就只有上下文，它所能做的也只有上下文切换。至于存储器里还有哪些资源是隶属于任务的，对它来说都是没有实在的意义的——它不仅分不清楚，而且会抱怨说“在那里又怎样，我又能做什么？关我什么事？”——所以我们可以站在处理器的视角狭隘的说：

任务的核心就是上下文；任务的切换就是上下文的切换。

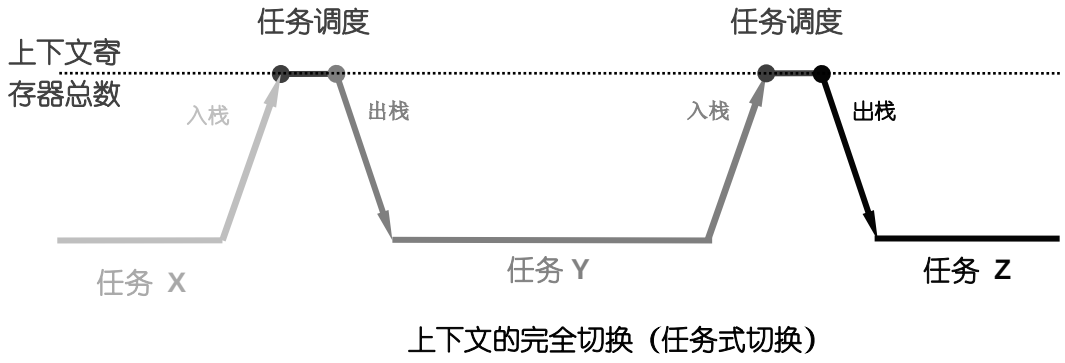
2.1.1 完全切换

如**图 2.1.1** 所示，一个简单的上下文切换方式由三个步骤组成：

- 将整个上下文入栈
- 运行调度算法，找到下一个要运行的任务，取出该任务的栈指针并更新 SP 寄存器
- 将目标任务的上下文出栈

由于在这个过程中，出入栈的上下文是完整的，因此被称为“完全切换”。完全切换是最安全、最直接、最彻底的方式。因为要做的事情明确，基本不用考虑什么特殊情况，完全切换也习惯上被认为是“最简单”的上下文切换方式——是的，只要你实现了完全切换，随便写个什么调度算法（比如轮转）你就可以骄傲的宣布“我的第一个操作系统诞生了！”

图 2.1.1 上下文的“完全”切换



上下文的完全切换（任务式切换）

注：本图仅展示任务之间的上下文切换关系，除斜线长度表示上下文切换的程度外，横线段的长度并无实际意义。

完全切换也是有缺点的，由于大部分时候出入栈操作都需要人为编写代码来实现（哪怕是使用汇编代码），完整的上下文出入栈就意味着固定且相对较大的时间开销。我们知道，“任务切换时间”是一个操作系统的关键指标，用户对它的关心程度就好比购买笔记本时对 CPU 性能的关心程度差不多。

调度算法的时间 + 出入栈的时间 = 任务切换时间

一般来说，出入栈的时间决定了任务切换时间的最小值——也就是理论上的最快切换时间——在这种情况下调度算法执行的时间最短，用以处理时效性要求最高的紧急任务。一个好的操作系统，调度算法的执行时间应该是常量，又由于出入栈的时间是常量，这就最终确保了操作系统调度时间的确定性。实际上，很多人一直存在认识上的误区，以为任务的切换时间只要越快就越好。从实时操作系统（RTOS）

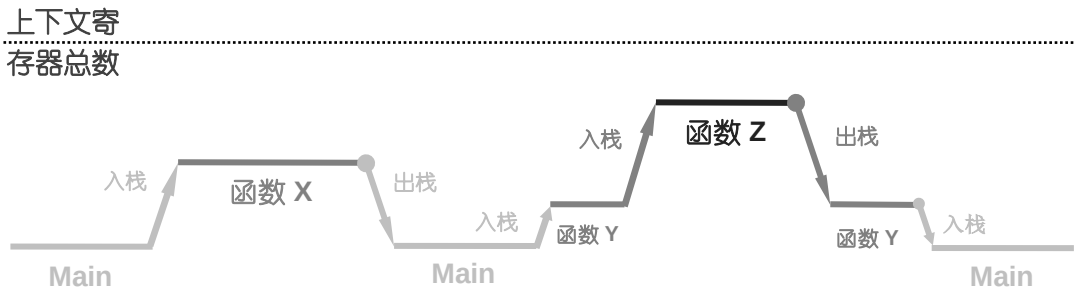
的角度来说这是片面的。实时操作系统首先确保的是任务切换时间的确定性，其次，在这一基础上才会考虑“越快越好”的问题。通俗的说，一个好的实时操作系统，任务切换时间必须是一个常量，在这个基础上这个常量当然是越小越好了。

2.1.2 部分切换

与上下文的完全切换相对，还有一类更为常见的切换方式，称为“部分切换”。所谓“部分切换”，顾名思义就是只有部分上下文在任务切换时被从栈中压入或弹出，比如图 **2.1.1** 展示的就是下面代码的部分切换示意图：

```
void func_X(void)
{
   ...
}

extern void func_Z(void);

void func_Y(void)
{
   ...
   func_Z();
   ...
}

void main(void)
{
   While(1) {
      ...
      func_X();
      func_Z();
      ...
   }
}
```

图 **2.1.1** 上下文的 **“部分”** 切换——合作式上下文切换



合作式上下文切换（函数调用）

注：本图仅展示任务（函数）之间的上下文切换关系，除斜线长度表示上下文切换对栈的消耗的程度外，横线段的长度并无实际意义。

在这个代码例子中，主函数分别调用了两个任务函数 func_X()和 func_Z()，它们显然是共享上下文的。你要说，这哪是任务分明是函数调用嘛！实际上任务的实现方式有很多，裸机环境下通常就是以这种方式来实现多任务的。换一个角度来说，并非只有任务的执行会用到上下文切换，函数的调用也会用到上下文切换，只不过这种切换是由多和任务或者函数彼此合作进行的，这种合作体现在：

- 每个函数/任务都很清楚自己用到了上下文中的哪些部分。
- 每个函数/任务在得到执行时都会自动将自己用到的那部分上下文入栈——因为它用到的这部分上下文会覆盖原有的内容，所以需要保存现场。
- 当完成了部分上下文的现场保护后，函数/任务本身实际上是独占整个上下文的。
- 当函数/任务在退出时，会根据自己用到的那部分上下文，将原本入栈保护的内容出栈从而恢复现场。

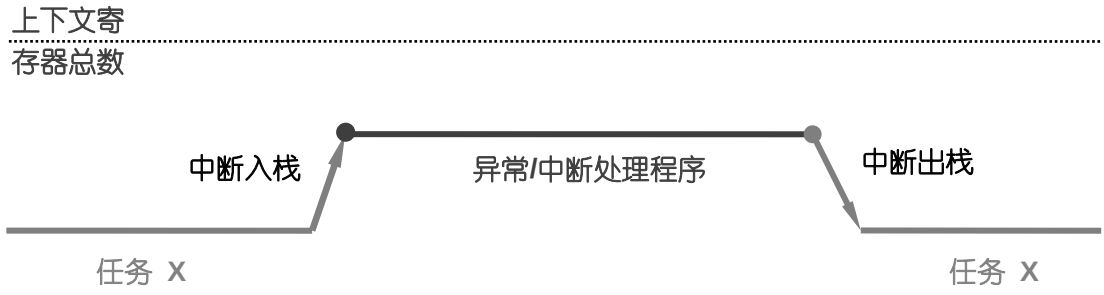
正因为函数/任务在编译器的协调下（编译时刻就决统筹安排好了的）都遵守着上面列出的规则——就仿佛是有一间单人工作室，每个获得机会使用工作室的人都会在进入前保存好前人留下的现场，而在退出后打扫一遍房间一样——所有的任务/函数都是以一种合作的态度在协作，因而，这种部分上下文的切换方式又被称为“合作式切换”。关于合作式上下文切换，还有几点值得注意：

- 上下文的切换是编译器通过产生代码来实现的，是一种软件切换。
- 每次上下文的切换都需要消耗额外的代码资源。函数越多，消耗在切换上的资源也越多。而完全切换是通过公共的切换代码来实现的，消耗在任务切换上的代码资源是固定的。
- 每个函数对上下文的占用是本着“按需分配”的原则进行的，是时间效率最高的上下文切换方式。
- 每个函数只根据自己对上下文的使用情况进行切换，与前后执行的函数/任务无关

在前面的例子中，函数 Y 中调用了函数 Z。根据以上规则，函数 Z 很清楚自己要用到多少上下文资源，因而有针对性的进行了现场保护，这与函数 Y 使用了哪些上下文是无关的。

中断/异常处理所涉及到的上下文切换是另外一类“部分切换”。如果说函数的合作式切换是编译器计划好的，那么中断/异常处理所带来的对处理器资源的抢占式切换就是计划外“临时发生”的，我们不妨称之为“临时切换”（如**图 2.1.2** 所示）。临时切换具有以下特点：

图 2.1.2 上下文的“部分”切换——临时上下文切换



临时上下文切换（中断/异常处理）

注：本图仅展示任务（函数）之间的上下文切换关系，除斜线长度表示上下文切换对栈的消耗的程度外，横线段的长度并无实际意义。

- 临时切换首先是一种“部分”上下文切换。系统根据中断/异常处理程序所使用的上下文范围来确定所需出入栈上下文的范围；
- 临时切换由能产生中断/异常请求的硬件逻辑发起（外设、内核都能发起中断/异常请求），由内核响

应并执行。相对合作式上下文切换来说，临时上下文切换是突然发生的，相对当前正在运行的代码，其发生的时间和位置都是随机的。

- “临时切换”在追求处理能力的中高端处理器中是完全由硬件实现的，以获取最快的响应速率；在另外一些处理器中，为了获得效率与灵活性的平衡，处理器只进行最小的现场保护和恢复——仅通过硬件对 PC，SP，LR 以及部分通用寄存器（比如 ARM Cortex Mv6 / Mv7 的 R0~R4）进行出入栈，剩下的工作则交由中断/异常处理程序来完成。总的来说，我们可以认为“临时切换”是一种硬件主导软件配合的上下文切换方式。

2.2 进程，线程，中断处理程序是任务吗

与一般意义上的认识不同，学过操作系统的人反而对任务的概念比较模糊。他们也许了解什么是进程、什么是线程、什么是中断处理程序——这都是由他们用什么操作系统决定的——但对于什么是任务，往往模棱两可。这并不是说任务是一个伪概念，而具体的进程、线程才是专业人士使用的概念。恰恰相反，进程、线程、中断处理程序甚至是很多人并不熟悉的协程都是任务的一种体现形式，他们都是任务，只不过形态和特点不同。这就好比，鲫鱼黄鱼都是鱼，你不能说鱼是伪概念，必须具体谈论某一种鱼才是正确的。

2.2.1 什么是任务（Task）

广义的任务其实比较简单，就是“做事情”、“完成一个任务目标”，具体谁去做，怎么去做，什么时间，什么地点用什么工具则是依据具体情况各个不同了。太笼统的概念来头太大，对实际工作没有太多的指导意义。嵌入式系统中的任务，虽说仍然是“为了某个明确的目标做事情”，但范围已经有所限定，具体来说就是内核，通过寄存器操作的方式，以给定的时钟频率，实现时序控制或者数值运算。这是任务的主体，但根据对存储器资源和内核资源（处理器时间）的占用方式不同，任务又存在几种具体的表现形式，如下面要依次介绍的进程（Process）、线程（Thread）、中断处理程序（ISR）、协程（Coroutine）、状态机（FSM）等等。

2.2.2 进程（Process）

相对并不是很长的计算机发展史，进程是一个相当古老的概念，一直可以追溯到用“灯泡造恐龙”的时代。那个时候的任务要比现在“具体”的多——由于程序都是通过类似现在英语考试的机读卡一样的硬纸板记录的，通常一抽屉的硬纸板就是一个任务（早期叫作业）。一个完整的作业包括将机读卡通过抽屉送到输入设备用以读取程序和数据；运行程序；将程序运行结果以机读卡的形式输出出来。你可以看到，对一个作业来说，当其获得处理器时，它是实实在在占有整个电脑的！这句话的意义包括，当前作业占有整个电脑的存储器，处理器，输入设备和输出设备——简而言之，程序员在编写程序时可以认为在作业的生命周期内它是独占整个系统的。随着计算机技术的发展，多道、批处理、分时复用等等多任务技术相继被引入进来，但“任务独占整个系统”的习惯却被保留了下来——因为对程序员来说，这样编写代码最简单。

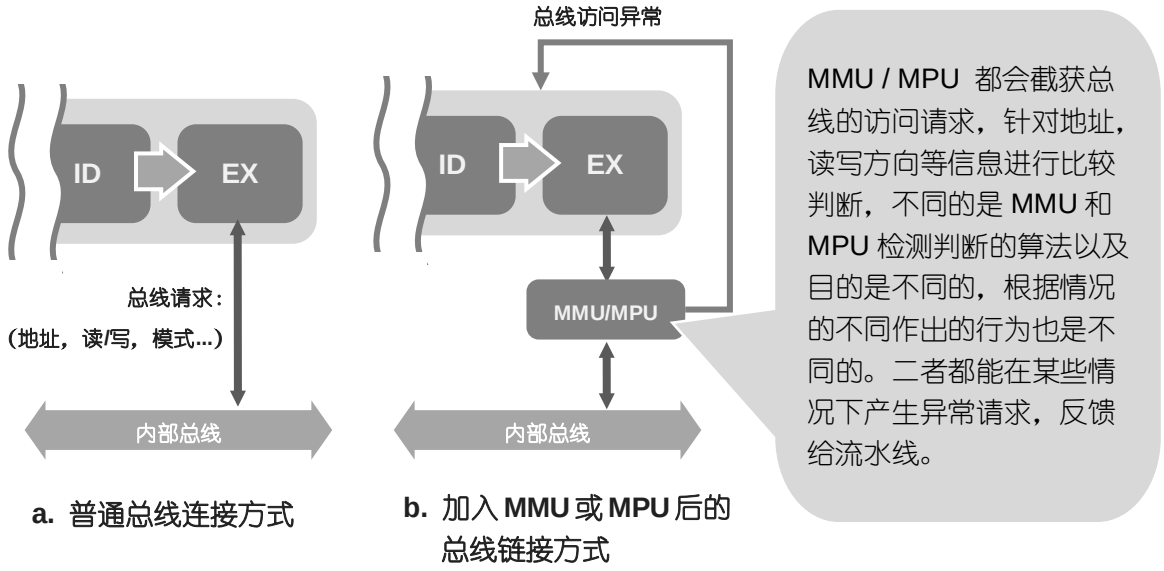
这是一种古老的欺骗术：对任务来说，当且仅当它被分配了处理器时间，我们才说这个任务是活跃（Active）的。简单的说，只要处理器不执行对应任务，那么它的时间就静止了——因此当我们站在任务的视角会发现：不借助外物，任务永远无法感知到时间的停顿，它自始至终认为自己占用的时间就是处理器的全部时间——换句话说，每个任务都认为自己完全占用处理器。这是从时间角度来说的。

从空间角度来说，不借助特殊的手段，很难实现“每个任务独占整个地址空间”的错觉，道理很简单，任务 A 要用地址 0x12345678，任务 B 也要用 0x12345678，这肯定打架的。为了营造“每个任务都独占整个地址空间”的错觉，虚拟地址空间技术被引入进来。这里虚拟地址空间是相对原本的实际物理地址空间（又叫做实地址空间）来说的，每个任务都独占一块虚拟地址空间，而虚拟地址空间中的地址与实际的物理存储地址是无关的，这就使得即便每个任务都使用了虚拟地址空间中的地址

0x12345678，而实际每个任务的 0x12345678 都被映射在了完全不同的物理地址上。

虚拟地址空间技术是怎么实现的呢？原理其实并不复杂，其基本思想可以用“偷梁换柱”来概括。**图 2.2.2** 展示了一个内核流水线的局部。和我们在第一章中看到的内核流水线一样，这是一个简化了的

图 2.2.1MMU 和 MPU 的引入



a. 普通总线连接方式

b. 加入 MMU 或 MPU 后的总线链接方式

三级流水线，其中指令直接阶段（EX）会通过总线访问地址空间内的存储器。这里，每一次总线访问都包含了一些基本信息，例如，操作的目标地址是多少、操作的方向是读取还是写入等等（如**图 2.2.2** a 所示）。如果我们在总线操作的过程中将其“截获”，并篡改其中的地址信息，就能实现将“虚拟地址”映射到指定“物理地址”的功能。这与武侠小说中常见的“掉包桥段”有着异曲同工之妙，通过对信使设伏，“将其麻翻之后”对信件的内容加以篡改，再“神不知鬼不觉的放回原处”，无论是送信人还是收信人对此都毫无知觉，一场好戏就此开始。这里，我们截获内核“EX 阶段”对总线访问所引入的模块叫做 MMU（存储器管理模块 Memory Management Unit），它的基本功能是：

- 根据截获到的访问地址查表获取当前地址与物理地址的映射关系。
- 如果映射关系存在，则直接将目标地址替换为新地址后发出总线请求。
- 如果映射关系不存在，则产生异常（通常叫**“缺页异常”**）报告给内核——由内核调用相应的处理代码来更新映射关系（实际上映射关系是一张表格，表格很大，并不能完全保存在 MMU 中，所以当 MMU 查询不到映射关系时，需要产生异常，通过内核来更新 MMU 映射表）。

容易注意到，当发生“缺页异常”时，MMU 需要内核的协助才能继续工作，这里立即就有两个问题了：1）如何快速的查找到目标地址对应的映射表；2）由于缺页的代价是巨大的，不仅要停下原本的操作，还需要内核运行算法才能找到正确的映射表，因而，如何设计映射关系，如何在随机的地址访问行为中降低缺页率就成为内存管理算法的用武之地。“段式”管理，“页式”管理以及“段页式”管理，都是经典的算法，这是操作系统类书籍通常都会涉及到的内容，这里就不再赘述。

进程的前身就是作业，是一种拥有虚拟地址空间的任务。现代的进程几乎已经不再作为操作系统调度的基本单位了，线程是调度的最小单位。一个进程包含一个或多个线程。显然同一个进程中的线程共享同一片虚拟地址空间。关于二者的关系，我们可以简单的加以概括：“进程管资源（虚拟地址空间），线程管调度”。

在图 2.2.2 中还展示了另外一种存储器管理模块 MPU（存储器保护单元 Memory Protection Unit），从原理上说，它几乎是一个简化版的 MMU。MPU 并不进行地址置换（或者说“掉包”），它与 MMU 一样截获内核发出的总线访问请求，与之不同的是，MPU 是个边境检查站，它只关心总线访问的地址是否是“禁区”，访问的属性是否受到限制。MPU 也有一张表，这张表保存了若干事先划定好的存储器范围，这些范围都被特别标记了限制条款。当 MPU 截获到总线访问请求时，它会比对这张表，如果访问“违法”，MPU 就会产生异常，告知内核进行处理；如果“合法”请求则被直接放行。MMU 通过虚拟地址空间的方法来隔离不同的任务，使它们相互在存储器使用上不会发生干涉，MMU 逻辑复杂，体积较大，成本较高，一般只在高档的 CPU 中使用。MPU 同样是为了隔离不同的任务，使它们不会相互干涉，但与允许任务“任性”的随便使用虚拟地址空间的 MMU 不同，MPU 的策略是限制每个任务的活动范围，严格监管它们的行为，使它们不能越雷池一步。MMU 是内核是否支持 Linux 的关键。

2.2.3 线程（Thread）

对很多轻量级的实时操作系统来说，由于其面向的芯片并不具有 MMU，因而并未引入进程的概念。线程是这类操作系统的最小调度单位。RT-Thread、μCOS、FreeRTOS、RTX 都是这类操作系统的典型代表。

2.2.3.1 线程的特点

作为任务的一种常见表现形式，线程间的切换采用的是上下文的“完全切换”。一方面，完全切换确保了任务的切换时间是固定的，另一方面，完全切换也可以理解为一种无奈之举。从原理上说，任务间上下文的切换应该按需分配，仅切换下一个任务实际使用到的部分——就像合作式切换那样——然而，这对线程来说是不现实的。因为合作式切换的协调者或者说组织者是编译器，编译器掌握着每个函数/任务对上下文使用的详细信息，而线程任务切换的执行者是运行时刻的调度器，当它在任意时刻打断正在执行的程序时，很难知道当前任务此时此刻使用了上下文的哪些部分——因为这些信息并没有被记录下来，这就好比你突然冲进实验室，发现试验台上仍然有很多连好线的仪表和装置，实验室里没人，也许他们吃饭去了，因此你根本无从知道桌子上哪些设备和装置是别人正在使用的，对你来说现在最保险的方式就是把整个桌面上的所有装置都小心翼翼的挪动到别的地方（假设这是允许的），以便你用完实验台后可以恢复——这也正是调度器所采用的策略，它会选择切换整个上下文，以保证任何任何情况下当被切换的线程重新获得执行的机会时，都可以安全的恢复现场，使其察觉不到任何异样。

为什么没有一种操作系统试图记录每个线程的上下文使用情况，以方便上下文切换时做到“按需分配”呢？具体原因笔者并不知道，这其实是一个理论可行的方案，但为什么鲜有实践者，我猜大体是因为这种方式操作起来太麻烦了。我不排除很多人“光想想就觉得麻烦，因而打消了这个念头”，实际上存在的困难的也确不小：

首先，我们需要设计一种机制来记录每个任务的上下文使用情况，如果要做到“及时”记录，显然是开销巨大的——在任务控制块中建立一个类似上下文使用情况的 Bitmap 消耗并不大，也许一两个 uint32_t 的变量就足够了，但更新这一 Bitmap 的代码却是需要占用体积的，而且根据你“更新”的算法不同，其开销也会是天壤之别。

其次，这种机制需要很好的执行手段，简单说，我们如果把更新“当前任务上下文使用情况”的代码抽象为__update_context_usage()这样的系统原语，要么 我们获得编译器的支持——在合适的位置（比如上下文的使用发生变化的地方）自动插入到任务中；要么我们就需要程序员人为的插入这些信息到“他们觉得合适的地方”——这对调度器来说是显然是个灾难。

最后，从任务切换的角度来说，显然只有在“上下文使用情况更新之后”且“上下文实际使用情况变化之前”这段时间，切换任务是安全的。如果我们能做到“每次上下文使用情况变化时都能及时的更新”，那么显然任意时刻进行任务切换都是安全的；如果我们做不到，对调度起来说，显然只有在__update_context_usage()原语执行后立即进行切换才是安全的。那么问题又来了，如何方便恰当的在

任务中放置__update_context_usage()就成了这个方案成败的关键。

从上面的讨论中很容易看出，问题的关键在于信息不对称——任务调度的实施者，也就是调度器，并不了解每个任务对上下文的使用情况——如何将这一信息传递给它就成了“按需分配”式上下文切换的关键。那么谁拥有这些信息呢？程序员、编译器以及内核。严格来说程序员并不真的清楚高级语言代码与上下文使用情况的细节关联，因而让程序员去插入__update_context_usage()原语实际上是强人所难，如果真的采用这种方式，显然是很难推广的。编译器是高级语言和机器语言之间的桥梁，由它来做原语的插入当然是合适的，只是哪家编译器厂商愿意为了这种非主流的方案冒着牺牲代码尺寸空间和效率的风险来做这样的支持呢？最后只剩下内核。是的，内核拥有所有上下文的使用信息，因为它是最终的实施者。对内核来说，首先指令是以寄存器为接口进行设计的，无论是运算类的指令还是存取类的指令（详情请参考第一章），对它，如果维护一个 Bitmap，在上下文中的寄存器被读取或者写入的时候对 Bitmap 进行简单的标记（比如将对应的二进制位置位）简直就是举手之劳。我们甚至可以设计以下的规则，通过硬件实现全自动的“智能上下文切换”：

- 在内核寄存器中，增加一个记录上下文使用状况的 32 位寄存器，每个二进制位对应上下文中除 SP 以外的一个寄存器：PC、LR、R0~Rn 等等。我们不妨起个名字叫做 CTBM（Context Bitmap）。
- 当普通指令读取或者写入上下文中的某个寄存器时，内核将 CTBM 中对应的二进制位置位。
- PUSH，POP 指令被视作普通指令
- 当内核响应中断/异常，自动将某些上下文寄存器压入栈中时，内核应该将 CTBM 中对应的二进制位清零，当内核从中断/异常处理中返回、自动将某些寄存器的值从栈中弹出时，内核应该将 CTBM 中对应的二进制位置位。
- 当调度器读取 CTBM 时，内核应该在返回寄存器值以后，将被标记的上下文寄存器依次压入栈中，并最终将 CTBM 清零，显然调度器会将当前任务的 CTBM 值作为任务控制块的一部分保存下来；当调度器将任务控制块中先前保存 CTBM 值写回 CTBM 时，内核应该对应的寄存器依次出栈。

如果你对内核的基本原理有所了解的话应该能意识到，上述方法实际上实现了一个上下文使用情况的自动登记机制，并利用记录下来的信息自动完成了上下文的“按需切换”。这种方案不仅对编译器没有任何要求，对现有的开发工具和开发习惯来说是毫无影响的，因为所有上下文信息的追中和操作都是内核自动完成的，对内核的使用者（对编译器和最终用户来说）是透明的。操作系统的编写者可以选择完全无视这种机制的存在，使用传统的上下文切换方法，也可以欣然的选择通过读写 CTBM 寄存器来完成原本相对繁琐的上下文切换工作。这种“按需切换”也许对普通 MCU 的上下文切换帮助有限，但对于支持协处理器（例如，浮点运算单元，FPU）的系统来说，协处理器的寄存器页也是上下文的一个组成部分，这种情形下，上下文完全切换的开销就非常惊人了（ARM Cortex M4 浮点运算单元的寄存器页有 32 个 WORD 那么大），在这种应用场景下，“按需切换”带来的收益相当可观。

一个看似不可能的事情，因为内核的举手之劳突然间就变得非常具有可行性，甚至是革命性的，是不是很神奇呢？这就是软硬结合、全局思维的力量。虽然现在还没有这样的内核，但也许读者中会出现有能力将其实现的人。我们通过小字加入的这段内容目的是让读者学会思维的发散和扩展，有时候看似异想天开的方案，只要原理上可行，剩下的就在于找到问题的症结，对症下药、披荆斩棘，说不定就走出一条从无人走过的新路子——这就是创新。

2.2.3.2 如何在 RT-Thread 中创建任务

通过上面的讲解我们会发现：RT-Thread 中所谈论的任务其实就是线程，而创建任务本质上就是创建线程。在 RT-Thread 中，线程的创建有“动态”和“静态”两种方式，其目的是相同的：1）为线程分配栈空间用以上下文的存储和局部变量的分配，2）配置线程的各类调度参数。我们首先用一个例子来说明如何动态地创建一个线程，并启动它：

```
#define DEMO_STACK_SIZE                (512)
#define DEMO_PRIORITY                  (30)

/* 线程入口 */
void rt_demo_thread_entry(void* parameter)
{
    while (1) {
        rt_kprintf("This is the demo thread!\r\n");
        rt_thread_delay(100);        /* 线程睡眠100个系统节拍 */
    }
}

/* RT-Thread 用户应用入口函数 */
int rt_application_init()
{
    /* 指向线程控制块的指针 */
    rt_thread_t tThreadId;
    /* 创建线程 */
    tThreadId = rt_thread_create(
        "dynamic demo",              /* 线程名称      */
        rt_demo_thread_entry,        /* 线程入口      */
        RT_NULL,                     /* 线程入口参数  */
        DEMO_STACK_SIZE,             /* 栈大小        */
        DEMO_PRIORITY,               /* 线程优先级 */
        20);                         /* 线程时间片 */

    /* 判断是否创建成功 */
    if (tThreadId != RT_NULL) {
        /* 将线程加入就绪队列 */
        rt_thread_startup(tThreadId);
    } else {
        return -1;
    }

    return 0;
}
```

在 RT-Thread 中 “rt_application_init()”被认为是用户应用的入口函数。要创建一个线程，我们首先需要定义一个指向线程控制块的指针 tThreadId，使用“rt_thread_create()”接口创建一个线程，将该线程控制块的地址记录在 tThreadId 中。在创建线程的过程中，rt_thread_create()函数需要知道下面的信息：

- 线程名称

以一个字符串表示线程的名称，字符串长度最大为 RT_NAME_MAX；

- 线程入口

线程入口即为这个线程首次运行的入口，这里填入口函数的地址；

- 线程入口参数

线程入口函数的参数

- 栈大小

动态创建线程时，系统需要从动态堆中分配栈空间，该参数以字节为单位指定栈空间大小。分配出来的栈空间将在任务建立时自动被对齐到 8 字节；

- 线程优先级

指定线程优先级，范围是 0~RT_THREAD_PRIORITY_MAX-1，数字越小优先级越高；

- 线程时间片

当系统中存在相同优先级的线程时，该参数指定一次调度后线程最大能运行的时间长度，单位是系统节拍数。该指定时间运行结束后，系统自动选择下一个就绪的同优先级任务运行。针对任务优先级和同优先级任务的调度问题，我们将在第五章以“调度模型”的方式详细为您展开。

使用“rt_thread_create()”动态创建一个线程后，该线程处于“初始态（Initialized）”（还没有处于就绪状态），不会开始运行。使用“rt_thread_startup()”将更改线程初始态为“就绪状态（Ready）”后该线程将参与调度器调度——如果新启动的线程比当前线程优先级还高，系统会立即运行这一线程。

静态创建一个线程使用接口“rt_thread_init()”，仍然使用上面的示例，如果用静态创建的方法，那么代码如下：

```
#define DEMO_STACK_SIZE                (512)
#define DEMO_PRIORITY                  (30)

static struct rt_thread tDemoThread;                        /* 线程控制块 */

ALIGN(8)    /* 预编译宏，将后续变量的首地址对其到 8 字节 */
static rt_uint8_t s_chStack[DEMO_STACK_SIZE];               /* 用户定义的栈空间 */

void rt_demo_thread_entry(void* parameter)                  /* 线程入口 */
{
    while (1) {
        rt_kprintf("This is the demo thread!\r\n");
        rt_thread_delay(100);                               /*线程睡眠 100 个系统节拍*/
    }
}
/* RT-Thread 用户应用入口函数 */
int rt_application_init()
{
    rt_err_t tResult;
    /*初始化线程*/
    tResult = rt_thread_init(
        &tDemoThread,                                       /*线程控制块地址*/
        "static demo",                                      /*线程名称*/
```

```
        rt_demo_thread_entry,                    /*线程入口*/
        RT_NULL,                                 /*线程入口参数*/
        &s_chStack[0],                           /*栈地址*/
        sizeof(s_chStack),                       /*栈大小*/
        DEMO_PRIORITY,                           /*线程优先级*/
        20);                                     /*线程时间片*/
    /*判断是否初始化成功*/
    if (tResult == RT_EOK) {
        /*将线程加入就绪队列*/
        rt_thread_startup(&tDemoThread);
    } else {
        return -1;
    }
    return 0;
}
```

对比可知，静态创建任务与动态创建任务的区别就在于用户需要自己提供线程控制块、线程栈空间，其它部分与动态创建任务并无异。需要注意的是，用户提供的栈空间的对齐方式依赖于硬件，32 位 ARM 处理器一般要求栈空间以 4 字节地址对齐。

- 线程控制块地址

用户定义的线程控制块的地址；

- 栈地址

用户定义的栈空间的地址；不同硬件平台对栈空间的对齐方式要求不同，一般对齐于系统总线。在普通 32 位 ARM 处理器中，要求以 4 字节对齐。

静态创建中使用的“任务句柄（Task Handler）”和“任务栈（Task Stack）”是静态变量、由用户指定并在编译时刻（Compile-Time）就确定下来的。相对线程的“静态”创建，“动态”创建更加灵活——“任务句柄”、“任务栈”是系统运行时刻（Run-Time）由 RT-Thread 内核从堆（Heap）中申请的。

无论用户使用动态创建还是静态创建任务，当栈空间分配了以后实际上系统内核做的关键一步是初始化栈，将任务入口、入口参数、栈顶指针和任务退出后要运行的地址压入栈中——制造一份初始的上下文信息。读者有兴趣的话可以阅读以下 Cortex-M3 移植版本上的栈初始化代码：

```
/* 异常/中断内核自动处理的上下文信息 */
struct exception_stack_frame
{
    rt_uint32_t r0;
    rt_uint32_t r1;
    rt_uint32_t r2;
    rt_uint32_t r3;
    rt_uint32_t r12;
    rt_uint32_t lr;
    rt_uint32_t pc;
    rt_uint32_t psr;
```

```
};
/* Cortex-M3 完整的上下文信息 */
struct stack_frame
{
    /* r4 ~ r11 register */
    rt_uint32_t r4;
    rt_uint32_t r5;
    rt_uint32_t r6;
    rt_uint32_t r7;
    rt_uint32_t r8;
    rt_uint32_t r9;
    rt_uint32_t r10;
    rt_uint32_t r11;
    struct exception_stack_frame exception_stack_frame;
};
/* 栈初始化函数 */
rt_uint8_t *rt_hw_stack_init(   void        *tentry,        /* 任务入口地址 */
                                void        *parameter,     /* 任务入口参数指针 */
                                rt_uint8_t  *stack_addr,    /* 栈顶初始地址 */
                                void        *texit)         /* 任务退出地址 */
{
    struct stack_frame   *stack_frame;
    rt_uint8_t           *stk;
    unsigned long        i;

    stk  = stack_addr + sizeof(rt_uint32_t);
    stk  = (rt_uint8_t *)RT_ALIGN_DOWN((rt_uint32_t)stk, 8);
    stk -= sizeof(struct stack_frame);

    stack_frame = (struct stack_frame *)stk;

    /* 初始化所有上下文寄存器*/
    for (i = 0; i < sizeof(struct stack_frame) / sizeof(rt_uint32_t); i ++)
    {
        ((rt_uint32_t *)stack_frame)[i] = 0xdeadbeef;
    }

    /* r0 : 任务入口参数 */
    stack_frame->exception_stack_frame.r0  = (unsigned long)parameter;
    stack_frame->exception_stack_frame.r1  = 0;                     /* r1 */
    stack_frame->exception_stack_frame.r2  = 0;                     /* r2 */
    stack_frame->exception_stack_frame.r3  = 0;                     /* r3 */
    stack_frame->exception_stack_frame.r12 = 0;                     /* r12 */
    /* lr 任务退出运行的地址*/
```

```
    stack_frame->exception_stack_frame.lr  = (unsigned long)texit;
    /* pc 任务入口地址 */
    stack_frame->exception_stack_frame.pc  = (unsigned long)tentry;
    /* PSR 更改内核状态*/
    stack_frame->exception_stack_frame.psr = 0x01000000L;

    /* 返回初始化完毕后当前栈顶指针 */
    return stk;
}
```

本节只讨论如何创建任务，至于如何在创建任务时给定恰当的优先级、合适的时间片，属于多任务设计方法的范畴，将在以后章节详细讨论。

2.2.4 中断/异常处理程序（ISR）

中断/异常处理程序是内核原生态支持的一种抢占式任务，用来快速响应那些“紧急且重要”的事件。是的，并非所有“紧急”的事件都值得用中断来处理，“重要”的事情也要根据时效性来进行区分。虽然中断的特点就是“抢占”、“响应速度快”，当使用中断/异常方式来执行的任务增多时，事件的响应时间也变得不稳定，系统的实时性也很难得到保证。好钢用在刀刃上，养成习惯只用中断/异常来处理那些“紧急且重要”的任务是写出高实时性系统的开始。

2.2.4.1 共享上下文的中断异常处理

中断/异常处理程序采用的上下文切换方式与内核提供的资源息息相关。为了减小内核尺寸，很多 8 位、16 位和低端 32 位内核仅提供一套寄存器页——无论是否实用操作系统，包括中断/异常处理程序在内的所有任务都共享同一个寄存器页。这种情况下，中断/异常处理采用临时上下文切换策略——即由硬件在中断/异常响应时完成关键上下文寄存器的压栈；在中断/异常处理程序退出时完成这些寄存器的自动出栈。

默认的情况下，中断/异常处理程序与当前正在执行的任务共享同一个栈，栈顶指针 SP 指向该栈（对裸机来说，中断/异常处理程序与超级循环中执行的任务共享同一个栈；对操作系统来说，中断/异常处理程序与当前被打断时正在执行的用户任务共享同一个栈）；如果用户在中断/异常处理程序中人为的对硬件未做处理的上下文进行保护，则有能力为每个中断/异常处理程序都提供完全独立的上下文和栈，具体操作步骤如下：

- 内核将部分核心的上下文寄存器压栈。（以 ARM Cortex M0 为例，内核会将 SR、PC、LR、R12 以及 R0~R3 寄存器自动压栈）
- 中断/异常处理程序将剩下的上下文寄存器入栈（以 ARM Cortex M0 为例，中断处理程序应该将 R4~R11 寄存器入栈）
- 中断/异常处理程序保存当前的 SP 指针，以备返回任务上下文时使用，并将自己专用栈的栈顶指针写入 SP 寄存器
- 中断/异常处理程序执行自己的任务
- 中断/异常处理程序将先前记录的任务栈顶指针重新写回 SP 寄存器
- 中断/异常处理程序将先前手工入栈的寄存器值弹出
- 中断/异常处理程序退出，内核将核心上下文寄存器出栈（ARM Cortex M0 会将 R3~R0、R12、LR、PC 和 SP 自动出栈），任务从中断点继续执行

在上述过程中，步骤 a 和 j 是内核原生支持的“临时上下文切换”；步骤 b、c 是我们人为加入的步骤，用以实现完全上下文切换，使用中断/异常处理程序的独享栈；步骤 e、f 是一个相反的步骤，用于将栈切换回原先的任务栈并通过手工配合硬件的方式实现完全上下文切换。实际上，无论是仅有 a、d、g 这样的原生态中断/异常处理，还是 a~g 这样的完整过程，步骤 d 里所执行的实际任务都是感觉不到任何差异的。

尽管要付出额外的存储器空间，即便从节省空间的角度考虑，操作系统环境下，为中断/异常处理程序提供独立的上下文和栈的优点也是明显的。假设某个系统中，中断/异常处理程序对栈的最大消耗为 X，对用某个户任务来说，如果仅仅满足任务本身的需求其栈大小为 Y，考虑当任务达到其最大栈深度时有可能发生中断/异常，为了保证系统仍然正常执行，该任务的栈大小应该为 Y+X。显然，如果系统中有 n 个用户任务，其满足自身需求的最小栈消耗为 Y_n，那么，为了保证系统正常运行，每个任务都要为异常/中断处理程序的栈开销买单，总体消耗为$\sum(Y_n+X)$，即$\sum Y_n+nX$。同样一个系统，如果我们为中断/异常处理提供独立的栈，则系统总体消耗为$\sum Y_n+X$。相比之下，节省了（n-1）X 的空间大小，当任务较多或者 X 较大时，这样处理是非常划算的。

2.2.4.2 仅提供独享栈的中断异常处理

考虑到栈在上下文切换中所起到的举足轻重的作用，有的内核专门提供了两个指针供中断/异常处理程序和用户任务区别使用。以 ARM Cortex M 系列微处理器为例，内核提供了两个栈顶指针 SP_main 和 SP_process，前者可以认为就是前文所述的 SP 指针——中断/异常处理程序和用户任务都可以使用；后者则专供用户任务使用。当芯片复位的时候，中断/异常处理程序和用户任务默认使用 SP_main 指针，这可以看做是对前文所述内核的一种向下兼容——很多裸机环境使用的就是这种模式，SP_process 并未得到利用。在操作系统环境下，SP_main 是供中断/异常处理程序专用的，用户任务则使用 SP_process 来维护自己专属的任务栈，在这种情况下，内核直接为中断/异常处理程序提供了独立的栈，当中断/异常得到响应时：

- 内核将部分核心的上下文寄存器压入用户栈中（由 SP_process 指向）。(以 ARM Cortex M3 为例，内核会将 SR、PC、LR、R12 以及 R0~R3 寄存器作为核心上下文自动压栈)。
- 内核切换当前栈顶指针为 SP_main，并在内核状态寄存器中记录先前使用的栈顶指针类型
- 如果中断/异常处理程序使用到了上下文中（除核心上下文外）剩下的部分，则会将涉及到的通用寄存器压入由 SP_main 指向的栈中；
- 中断/异常处理程序执行自己的任务
- 如果中断/异常处理程序使用到了上下文中（除核心上下文外）剩下的部分，则会将涉及到的通用寄存器的值从 SP_main 指向的栈中弹出；
- 内核将当前栈顶指针切换为用户栈（由 SP_process 指向）
- 中断/异常处理程序退出，内核将核心上下文寄存器出栈（ARM Cortex M0 会将 R3~R0、R12、LR、PC 和 SP 自动出栈），任务从中断点继续执行

通过上述步骤我们看到：在步骤 b 和步骤 f 中，原先由中断/异常处理程序手工实现的栈切换过程由内核硬件自动完成了。对比前文我们可以看出，ARM Cortex M 系列内核所提供的两个栈顶指针，虽然在中断/异常上下文切换的过程中所节省的时间非常有限，但在简化用户代码开发的效果上却是非常明显的——原本需要手工加入的代码现在由内核直接完成了，不能不说是一种进步。

不知细心的你有没有发现，同样是 a~g 的一个流程，共享栈的内核（只提供 SP 的内核）和双栈顶指针的内核（提供 SP_main 和 SP_process）在中断/异常上下文的保护上其实是本质不同的。前者实现的是完全上下文切换；而后者，由硬件压入栈中的“核心上下文”与由中断/异常处理压入栈中的“剩余部分”保存在不同栈中的，（例如，如果用户任务被打断时，核心上下文保存在由 SP_process 指向的栈中，而剩下的上下文则保存在由 SP_main 指向的栈中）——这根本不算是一次完全上下文切换，因

为上下文切换原本就是针对同一栈而言的，一次所谓的上下文切换，内容的不同部分保存在不同的栈中，这算怎么回事呢？

为了解决这个问题，我们可以对上述部分做一个小小的修改：

- 将步骤 c 修改为：将上下文中（除核心上下文外）剩下的部分全部压入 SP_process 所指向的用户栈中；
- 将步骤 e 修改为：将上下文中（除核心上下文外）剩下的部分全部从 SP_process 所指向的用户栈中弹出；

上述修改，利用了内核硬件“中断/异常处理”实现了双 SP 指针环境下的“完整上下文的切换”。RT-Thread 在针对 ARM Cortex M 系列内核的移植中正是利用这一方法实现任务上下文切换的，具体细节请参考 2.3.3 节的描述。

2.2.4.2 独享上下文的中断异常处理

高端的 MCU 为了追求中断/异常处理的响应速度，为中断/异常处理程序提供了完全独立的上下文寄存器页，这比单纯提供两个 SP 指针更近一步：将原本通过栈作为媒介进行的上下文内容的切换替换为以寄存器页为单位的直接切换，简单说就是用户任务一个寄存器页；中断处理程序一个寄存器页——这样大家终于不用抢了。

这是一个以空间换时间的策略——内核的尺寸更大，但上下文切换的时间更快。具体操作上，不同的内核仍然存在差异，有的内核以中断/异常优先级为单位，为每一个中断/异常优先级提供独立的寄存器页，相同优先级的中断/异常处理程序之间仍然需要共享同一个寄存器页，在出现同优先级中断/异常嵌套时，仍然需要借助栈进行必要的上下文内容切换；有的内核以中断/异常向量为单位，为每一个中断/异常向量都提供了独立的寄存器页，这种方式解决问题最为彻底，但同时空间消耗也最为巨大。

2.2.5 协程（Coroutine）

协程是一类很有趣的任务类型。首先，协程是一个概念，无论协程有多少种表现形式（实现方法），我们不能只抓住一种实现方法就当做是协程的本来面目。本文介绍了协程在嵌入式系统中的典型实现方式——即每一个任务都有自己独立的上下文和栈，与线程类似，协程任务的编写可以像独占了处理器那样使用超级循环。

与线程不同的是，协程是一种讲求协作的任务，这里的协作强调的是协程会根据需要“主动放弃对处理器的控制，并指认自己的后继者”。你可以把每一个协程都理解为分工合作的作坊伙计，大家不仅共享一个工作台，为了完成最终的作品，伙计们必须在完成自己的工序后让出工作台，交给别人继续处理。在使用协程的系统中，抢占是一个被弱化的概念，之所以被弱化而不是完全摒弃则从实用角度出发所做出的妥协——有些场合，如果确实需要用到抢占，协程也能几乎无缝的进行支持。

你也许注意到了，协程间的协作实际上是一种串行化的过程，前一道工序的人如果没有完成后一道工序就无法继续。也许你会说，流水线的协作要比串行化的协作高效的多，但流水线的并行是建立在所有工序都能同时工作的基础之上的，就好比每个人都有一个工作台。我们的内核通常只有一个，只能实现串行化的协作。即便你用多任务的方式在逻辑上做到了并行流水线，实际执行效果上，多个任务仍然是串行的。所以，从理论上说协程并不比线程效率更低。

协程间的任务切换，完全是由协程自己调用专门的原语__yield()实现的。这里的__yield()实现的正是任务间的上下文切换。需要强调的是，协程的__yield()通常具有一个参数，用于指定“将 CPU 控制权交给谁”。这种类似“禅让”的结构，不仅强化了任务“协作”的特点，更节省了调度算法的开支（代码空间和处理器时间的消耗）。对协程来说，不光任务间的协作关系是程序员指定的，就连任务间的执行顺序也是程序员在代码编写时刻就固化下来的；这与线程不同，线程与线程间的关系较为松散，虽然任务间的通信协作关系是程序员指定的——通过信号量（semaphore）、邮箱 （mailbox）、事件 （event）和临界区（mutex）等等来实现的——但线程的实际执行顺序是运行时刻由调度器通过调度算法动态决

策的。

我们不妨引用维基百科中的例子（伪代码），来说明协程工作特点：有一对生产者和消费者，使用队列作为缓冲。生产者将产品放入队列中，当队列为满而不能继续时，生产者会利用__yield()方法将控制权交给消费者。消费者从队列中取出产品消费，当队列为空而不能继续时，消费者会利用__yield()方法重新将控制权交还给生产者。

```
varq := newqueue

coroutine produce
    loop
        while q is not full
            create some new item
            add the item to q
        yield to consume

coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to procude
```

站在程序员的角度，协程的这一特点可以被视作为程序员“带来了空前的控制力”，也可以被认为是带来了更多的任务规划上的麻烦。很多程序员会说，既然我们已经采用了完全上下文切换，为什么我们还要用协程来给你自己找麻烦呢？为什么不让调度器根据优先级以及任务间的信号量同步来决定任务的执行顺序呢？协程相对线程有什么不同呢？

- 协程使得程序员对任务的行为有更强的控制力，因为每个协程都可以指定自己的后继者；
- 协程的编写和线程一样简单，因为每个线程都拥有自己独立的上下文和栈，你可以像编写线程一样轻松的编写协程；
- 协程只保留了任务间切换的原语__yield()，省去了任务的调度算法。
- 协程也能够使用信号量、互斥量、事件、邮箱和消息队列这些常见的任务间同步通信的手段。

总结来说，如果系统资源较为紧张，你又想对系统的行为有更强的掌控，如果还能想享受线程般编写的便利，何乐而不为呢？协程是你不二的选择。

2.2.6 状态机（FSM）

状态机和协程一样都是强调“协作”的任务形式，对程序员来说都具有“极强的掌控力”。它们的区别是，状态机任务之间共享同一个栈，使用“合作式”上下文切换（部分切换）；而每个协程都拥有自己的独立上下文和栈，采用“完全切换”的上下文切换形式。由于状态机任务间共享同一个栈，不需要与协程或者线程一样为每个任务分配足够大小的栈存储空间，因而能够节省更多的内存资源。

状态机和协程类似，都要求程序员手工的将具体的任务拆分成细小的执行单元，由它们协同合作完成整个任务。不同的是，协程拥有独立的上下文和栈，其任务的编写非常简单——代码可以使用非常自然的“阻塞”形式进行组织；状态机则强调每一个状态都必须是“非阻塞”的，当出现等待或较大的延

时时，应该释放 CPU 给其他状态机使用。

以 LED 闪烁为例，我们来说明一下阻塞代码和非阻塞代码的区别。首先，我们以阻塞代式来编写一个延时函数：

```
//! 阻塞形式的延时代码
void delay( uint32_t wTime )
{
    while(wTime--) {
        NOP();                  //!<加入 nop 可以有效防止编译器对改循环进行优化
    }
}
```

接下来，我们在超级循环中实现一个简单的 LED 闪烁功能：

```
extern void led_on(void);       //!<硬件无关的 LED 操作函数，点亮 LED
extern void led_off(void);      //!<硬件无关的 LED 操作函数，熄灭 LED

void led_task(void)
{
    led_on();                   //!<点亮 LED
    delay( DELAY_500MS );       //!<延时 500 毫秒，DELAY_500MS 是一个宏
    led_off();                  //!<熄灭 LED
    delay( DELAY_500MS );       //!<延时 500 毫秒
}

void main(void)
{
    ...
    while(1) {
        ...
        led_task();             //!<这是一个阻塞任务，只要它不完成一次灯的闪烁就不会退出
        ...
    }
}
```

这是一个典型的例子，堪称嵌入式学习的“hello world”。这也是一个阻塞型代码，当系统调用 delay() 函数进行延时的时候，CPU 处于一种“死等”的状态，无论你的内核多么强劲，死等都会把你“一句打回解放前”——这就是阻塞代码的威力，一夫当关万夫莫开，“阻塞”的含义体现的淋漓尽致。在操作系统环境下，情况得到了改观，即便某个任务以阻塞代码的形式编写，且没有使用 OS 提供的带有任务休眠功能的延时函数 sleep()，当高优先级任务就绪时，系统也会毫不犹豫的打断当前“阻塞”的任务，保证了整个系统其它任务的流畅运行。如果我们用状态机的非阻塞形式来实现上述功能，则是另外一番景象。首先，我们实现一个非阻塞的延时函数：

```
//! 状态量
```

```
typedef enum {
    fsm_rt_on_going      = 0,               //!<状态量on going，表示状态机正在执行
    fsm_rt_cpl           = 1,               //!<状态量complete，表示状态机执行完毕
} fsm_rt_t;

#define RESET_FSM()     do { s_tState = START; } while(0)

//! 状态机编写的非阻塞延时函数
fsm_rt_t   delay_fsm( uint32_t wTime )
{
    static enum {
        START = 0,
        WAIT_FOR_DELAY,
    } s_tState = START;                     //!<定义一个状态变量，用以表示状态机当前的状态
    static uint32_t  s_wDelay;              //!<实际用于延时的静态局部变量

    switch (s_tState) {
    /* START 状态机的进入“事件”，在状态机复位后首次运行时，运行且仅运行一次，用于初始化
    状态机环境（比如状态机用到的各类变量） */
        case START:
            s_wDelay = wTime;               //!<记录要延时的时间长度
            s_tState = WAIT_FOR_DELAY;      //!<切换状态
            //break;                        //!<使用fall-through 的方法，提高效率

        /* 实际用于延时的状态 */
        case WAIT_FOR_DELAY:
            if (0 == s_wDelay) {
                RESET_FSM();                //!<复位状态机
                return fsm_rt_cpl;          //!<说明状态机执行完成
            }
            s_wDelay --;                    //!<更新计数器
            break;
    }

     return fsm_rt_on_going;                //!<默认情况下返回状态机正在执行
}
```

接下来我们在超级循环中实现一个非阻塞的状态机，同样实现LED的闪烁功能：

```
fsm_rt_t    led_task_fsm(void)
{
    static enum {
        START = 0,
        LED_ON,
```

```
        DELAY_1,
        LED_OFF,
        DELAY_2
    } s_tState = START;

    switch (s_tState) {
        case START:                     //!< START 作为基本格式保留
            s_tState = LED_ON;          //!<切换状态
            //break;

        case LED_ON:
            led_on();                   //!<点亮 LED
            s_tState = DELAY_1;         //!<切换状态
            break;

        case LED_DELAY1:
            //! 调用延时子状态机
            if (fsm_rt_cpl == delay_fsm( DELAY_500MS )) {
                //! 如果延时完成了
                s_tState = LED_OFF;     //!<切换状态
            }
            break;

        case LED_OFF:
            led_on();                   //!<熄灭 LED
            s_tState = DELAY_2;         //!<切换状态
            break;

        case LED_DELAY2:
            //! 调用延时子状态机
            if (fsm_rt_cpl == delay_fsm( DELAY_500MS )) {
                //! 如果延时完成了
                RESET_FSM();            //!<复位状态机
                return fsm_rt_cpl;      //!<返回状态机完成
            }
            break;
    }

    return fsm_rt_on_going;             //!<默认返回状态机正在执行
}

void main(void)
{
    ...
```

```
    while(1) {
        ...
        //! 这是一个非阻塞的任务，无论是否在延时，状态机都会很快释放 CPU。
        led_task_fsm();
        ...
    }
}
```

由于状态机具有非阻塞的特性，因而多任务协同时往往表现出较好的实时性，这是缺乏调度算法的协程所难以实现的。我们可以简单的认为：相比线程，状态机编写更复杂，但掌控性较强，资源的占用较小；相比协程，状态机的编写更复杂，但在提高任务实时性方面较为容易。

2.2.7 小结

进程、线程、协程、中断/异常处理和状态机都是任务的不同表现形式。**图 2.2.2** 主要从上下文切换和地址空间的视角对这五类任务进行了划分。**表 2.2.1** 以表格的形式列举了五类任务的特点，并对他们进行了纵向的比较。

总结来说，进程需要在内核中引入 MMU 的硬件支持，提供完全独占主机的开发环境；线程则是传统嵌入式实时操作系统的最小调度单元，允许程序员在任务中使用阻塞代码，开发较为容易；状态机是裸机环境下最常见的多任务实现形式，尽管资源消耗较小，由于强制程序员必须编写非阻塞代码，因而开发难度较大；协程在资源消耗上与线程接近，开发难度上却与状态机类似（需要手动规划任务的执行细节，甚至很多时候还不如状态机灵活），因而在嵌入式开发中并不常见；几乎在所有的系统中，中断/异常处理都会得到应用，用于专门处理“紧急且重要的事情”。

图 2.2.2 根据上下文切换划分的任务类型

<table>
<tr><td colspan="4">实地址空间</td><td>虚拟地址空间</td></tr>
<tr><td colspan="4">上下文切换（Context Switching）</td><td rowspan="4">进程
（Process）</td></tr>
<tr><td colspan="2">部分切换
（所有任务共享栈）</td><td colspan="2">完全切换
（每个任务独享栈）</td></tr>
<tr><td>合作式切换</td><td>抢占式切换
（临时切换）</td><td>合作式切换</td><td>抢占式切换</td></tr>
<tr><td>状态机
（FSM）</td><td>中断/异常处理
（ISR）</td><td>协程
（Coroutine）</td><td>线程
（Thread）</td></tr>
</table>

表 2.2.1 任务的多种形式

任务形式	上下文	栈	虚拟地址空间[1]	调度方式	编写难度	
进程（Process）	独占	独占	独占	抢占式	★	独占主机
线程（Thread）	独占	独占	共享	抢占式	★★	自动调度
协程（Coroutine）	独占	独占	共享	合作式	★★★★	手动切换
中断（ISR）	共享[2]	共享[2]	共享	抢占式	☆	内核原生支持
状态机（FSM）	共享	共享	共享	合作式	★★★★★	必须非阻塞

NOTE 1、仅针对使用 MMU 提供虚拟地址空间的操作系统。
2、有些微处理器为了提高中断/异常响应的速度在硬件上为每一个中断向量都提供了独立的寄存器页，在这种情况下，中断其实是独占硬件上下文的，栈也是独占的。

2.3 如何实现上下文切换

2.3.1 如何实现完全切换

人常说“顺藤摸瓜”，根据第一章对上下文内容的描述，如果上下文是瓜，那么栈顶指针（SP）就是瓜的藤了。任务的切换离不开上下文的保护和恢复，很多书籍又将其称为“现场”的保护和恢复，无论是现场还是上下文，他们说的都是同一个事物。与其说，栈顶指针是上下文的瓜藤，不如说它是栈的瓜藤，只不过由于上下文通常是暂存于任务专属的栈里，它的地位才显得如此关键——保护现场时，调度器将上下文压入栈中；恢复现场时，调度器将上下文从栈中弹出——栈在任务切换中扮演着举足轻重的关键角色。

思考一个问题，现场保护时是否要将栈顶指针（SP）也一起压入栈中呢？我们不妨来打个比方：假设你有一个文具袋，就是那种很常见的敞口布口袋，袋口的边沿被向内缝合，留出一个小小的通道，一根鞋带样的吊绳从通道里穿过，使得你拉起吊绳的两头就可以一口气把袋口收紧，顺手就拎走了。我们可以把这样的文具袋看作是栈，里面可以保存了你自习所需的文房四宝。当你准备离开座位时，将桌上的东西放入袋中，我们可以视作上下文的保护。当你把一切收好，拉紧袋口时，我们可以认为整个自习任务都收纳在这个带子里了——表示上下文的文具已经压入了由文具袋代表的栈中，而这根吊绳就是拎起笔袋的栈顶指针（SP）。假设你有很多这样的文具袋，每个文具袋中保存了一门科目（你可以理解为任务），当你拎起或者展开不同的文具袋时，实际上代表的就是任务的切换。既然吊绳是我们拎起文具袋的关键，那么保护现场时，如果将吊绳也塞进袋子里我们还如何“拎走”袋子呢？

很容易想到，栈顶指针是任务切换的关键——对线程来说：

任务的切换就是上下文的切换，上下文的切换就是栈顶指针的切换。

上下文的切换中，并不需要将栈顶指针压入或从栈中弹出。线程的任务控制块中，至少包含当前任务的栈顶指针。

```
//! RT-Thread 的任务控制块
struct rt_thread
{
    /* rt object */
    char     name[RT_NAME_MAX];                 /**<任务名称 */
    ...
    /* stack point and entry */
    void         *sp;                           /**<栈顶指针 */
    void         *entry;                        /**<任务入口 */
    void         *parameter;                    /**<任务入口参数 */
    void         *stack_addr;                   /**<栈地址*/
    rt_uint16_t stack_size;                     /**<栈大小 */
    ...
```

```
};
```

RT-Thread 上下文切换函数的两个形参实际上就是“被切换”和“切换到”两个任务的栈指针，它展示了栈顶指针在任务切换时所起的关键作用，详细内容将在后面的章节中详细讨论。

```
/* 保护当前任务的上下文，伪代码 */
extern void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
```

2.3.2 如何利用临时切换

由上文我们知道，当中断/异常发生时，内核会将当前的关键上下文自动压栈，如果我们的操作系统能利用这一点，只手工压栈剩余的上下文信息来执行完全切换，就能实现事半功倍的效果。要实现这一点，我们必须有能手工触发中断/异常，并在该中断/异常处理函数里完成上下文切换动作。

- 什么是 PendSV 异常，它和 SVCall 有什么区别？

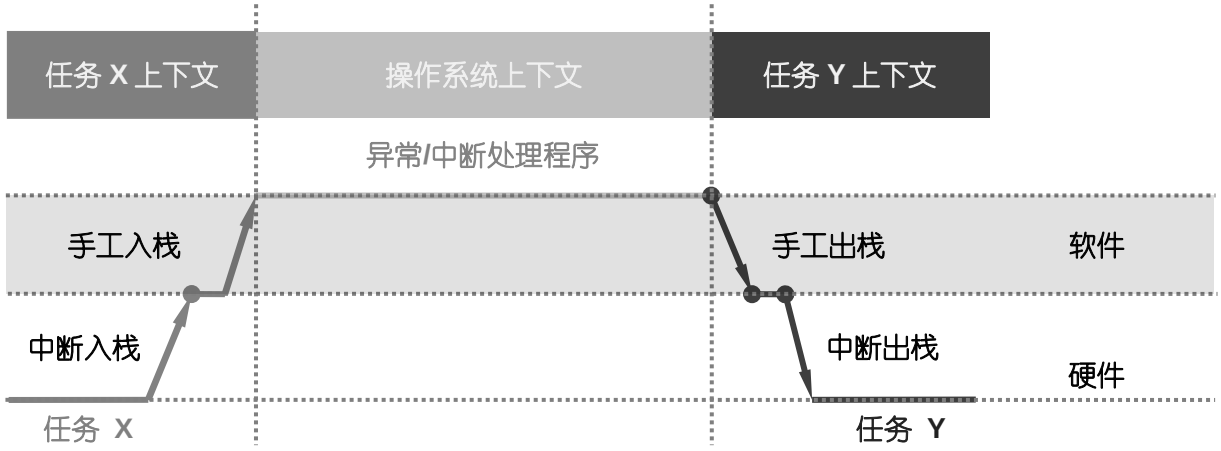
（您所阅读的仅仅是样章，并不包含所有内容，展现的内容也并不是最终形式，详细内容请参考正式书籍）

当然，SVC 并不只用来做“上下文切换”，很多情况下它作为系统特权操作的入口，需要通过一个参数承担多种系统服务的请求——这常常被用在操作系统内核态与用户态的隔离上。在特定平台上，是否使用 SVC 与 PendSV 做上下文切换，往往根据硬件平台特点和资源是否丰富决定。关于 SVC 与 PendSV 的更详细信息请参阅 ARM“编译工具开发指南”等相关文献。

图 2.3.1 描绘了巧用中断/异常进行上下文切换的情景。要实现图中的切换方法，无非类似于“将大象装进冰箱”的几个步骤。假设我们要从线程 A 切换到线程 B：

1. 获得当前线程 A 的栈指针 from_thread->sp 和目标线程 B 的栈指针 to_thread->sp
2. 触发某中断/异常，内核自动将线程 A 关键上下文压栈
3. 手工对线程 A 的非自动部分上下文压栈
4. 手工将线程 B 的非自动部分上下文出栈
5. 将栈顶指针指向线程 B 的栈顶
6. 退出该中断/异常，内核自动将线程 B 关键上下文出栈

图 2.3.1 巧用“临时（异常/中断）”上下文切换实现任务切换



注：本图仅展示任务（函数）之间的上下文切换关系，除斜线长度表示上下文切换的程度外，横线段的长度并无实际意义。

2.3.2.1 利用 PendSV 的完全上下文切换（从用户任务发起的上下文切换请求）

在 RT-Thread 的 Cortex-M 内核移植版本上，都是利用 PendSV 异常来处理上下文切换的。按照**图 2.3.1** 中描绘的切换步骤，下面演示最一般情况的上下文切换过程的实现代码。

1. 获得当前线程 A 的栈指针 from_thread->sp 和目标线程 B 的栈指针 to_thread->sp

```
rt_uint32_t rt_interrupt_from_thread, rt_interrupt_to_thread;
…

IMPORT rt_interrupt_from_thread             ; 记录“from”线程栈顶指针的变量
IMPORT rt_interrupt_to_thread               ; 记录“to”线程栈顶指针的变量

; * C 代码中调用上下文切换的接口
; * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; *r0 中保存 from 参数，r1 中保存 to 参数
EXPORT rt_hw_context_switch
rt_hw_context_switch:
;读取“from”栈指针到变量 rt_interrupt_from_thread
LDR     r2, =rt_interrupt_from_thread
STR     r0, [r2]
; 读取“to”栈指针到变量 rt_interrupt_to_thread
LDR     r2, =rt_interrupt_to_thread
STR     r1, [r2]
```

2. 触发 PendSV 异常，内核自动将线程 A 关键上下文压

```
LDR     r0, =NVIC_INT_CTRL              ; 人为触发 PendSV 异常，让内核硬件自动压栈
LDR     r1, =NVIC_PENDSVSET
STR     r1, [r0]
BX      LR

EXPORT PendSV_Handler
; 进入 PendSV 中断，psr, pc, lr, r12, r3, r2, r1, r0 被芯片内核自动压栈到“from”
PendSV_Handler:
…
```

3. 手工对线程 A 的非自动部分上下文压栈

```
PendSV_Handler:
LDR     r0, =rt_interrupt_from_thread
LDR     r1, [r0]
MRS     r1, psp           ; 获取“from”栈顶指针
STMFD   r1!, {r4 - r11}  ; 将内核的 r4~r11（非自动部分）压栈到“from”栈，并更新 r1 的值
LDR     r0, [r0]
```

```
STR     r1, [r0]          ;将更新后的栈顶指针写回线程“from”控制块，“from”线程压栈完毕
```

4. 手工将线程 B 的非自动部分上下文出栈

```
LDR     r1, =rt_interrupt_to_thread
LDR     r1, [r1]
LDR     r1, [r1]          ; 读取 rt_interrupt_to_thread 变量中保存的“to”栈顶指针到 r1
LDMFD   r1!, {r4 - r11}   ; 从“to”栈出栈 r4 ~ r11（非自动部分）到内核，并更新 r1
```

5. 将栈顶指针指向线程 B 的栈顶

```
MSR     psp, r1                ; 将当前“to”栈栈顶指针写入内核栈顶指针，手工出栈并切换完毕
```

6. 退出该中断/异常，内核自动将线程 B 关键上下文出栈

```
ORR    lr, lr, #0x04

; 退出 PendSV 中断，芯片内核自动完成 psr, pc, lr, r12, r3, r2, r1, r0 出栈
BX      lr
```

实现了这段上下文切换代码以后，即可在 C 语言源文件这样切换线程：

```
struct rt_thread *to_thread;
struct rt_thread *from_thread;
…
// 从“from”线程切换到“to”线程的调用语法示意
rt_hw_context_switch((rt_uint32_t)&from_thread->sp, (rt_uint32_t)&to_thread->sp);
```

上面六个步骤演示了如何保护线程 A 的上下文，然后将内核上下文切换到线程 B 中，并恢复线程 B 的上下文。之所以说这六个步骤是“最一般”的线程切换的实现，是因为实际应用当中有很多特殊情况要考虑。接下来的章节为您呈现各种特殊情况的处理，最后为您展现 RT-Thread 中完整的线程切换的实现。

2.3.2.2 第一次系统调度的特殊情况

一旦用户第一次尝试使用上文中的汇编代码进行线程切换时，马上就会发现一个问题：操作系统初始化后要切换到第一个用户线程时，“from”线程时不存在的。因此线程切换的代码需要考虑系统初始化之后第一次调度的情况。我们约定第一次调度时，“from”参数填“NULL”，那么只需要在“from”参数是 0 时跳过“from”线程的压栈操作即可。

```
LDR     r0, =rt_interrupt_from_thread
LDR     r1, [r0]
CBZ     r1, switch_to_thread     ; “from”栈指针为 0，跳过对“from”手工压栈

; 对“from”线程的压栈操作
…
```

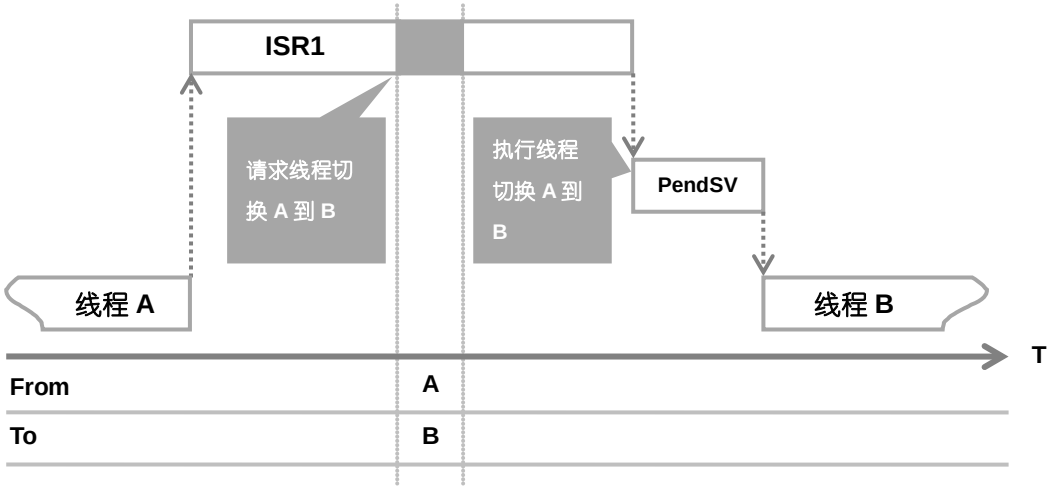
```
switch_to_thread:
; 对“to”线程的出栈操作
…
```

2.3.2.3 如何处理从中断/异常处理发起的上下文切换请求

上文中给出了从用户线程 A 中发起切换，切换到线程 B 的代码，并考虑了系统首次切换的情况。还有一种情况是在用户 ISR 中发起的线程切换请求。这时会发生什么情况呢？由上文中的讨论我们知道，使用 PendSV 做真正的上下文切换时，真正的切换操作发生在所有用户 ISR 都运行完毕的时候，假如用户的线程 A 被第一个 ISR 打断，这个 ISR 中请求线程切换到 B，但是在真正的切换发生前（PendSV 总是在其他 ISR 运行完最后得到响应），另一个 ISR 又请求 C 切换到 D，那么操作系统的调度器应该怎么做呢？最终是从 A 切到 B，还是从 A 切到 D，还是其他的行为呢？

首先，我们看一下正常的从中断中请求线程切换的情况，如**图 2.3.2** 所示。中断请求 1 打断了线程 A，并在 ISR1 中请求了系统调度，要求切换到线程 B。这时线程 A 的栈顶指针被保存在“from”变量中，线程 B 的栈顶指针被保存在变量“to”变量中。用户 ISR1 执行完毕，最低优先级的 PendSV 得以执行，真正的上下文切换在这里发生，用户线程从 A 切换到了 B。最终皆大欢喜，用户 ISR1 得到了及时的响应，在中断中请求的线程切换也得到了正确执行。

图 2.3.2 从中断处理程序中请求一次线程切换



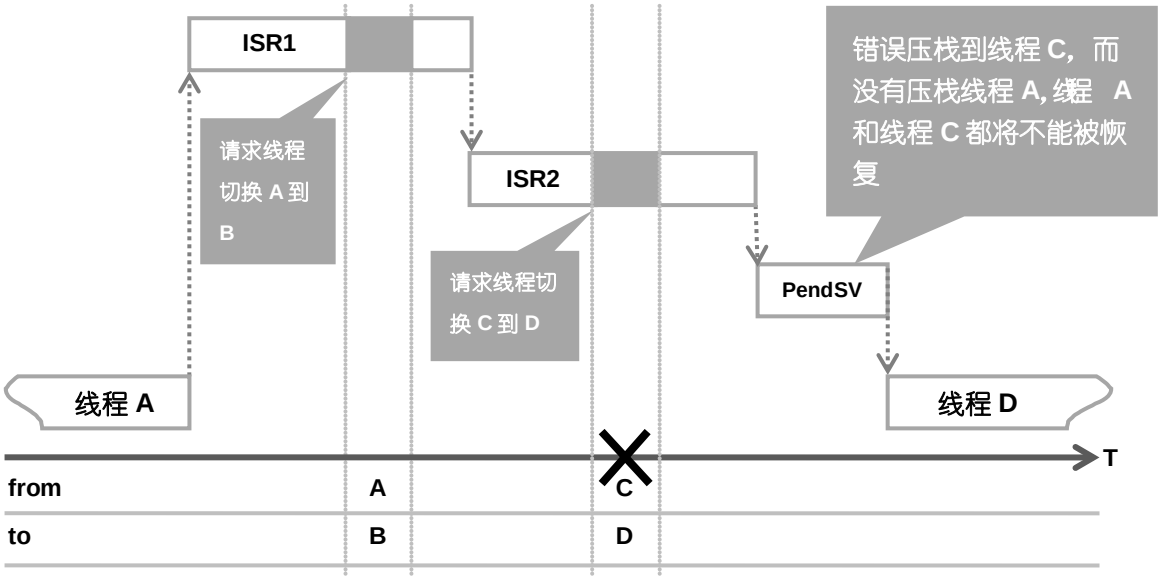
其次，我们来分析我们担心的情况，假如在 ISR1 中请求线程切换之后，ISR1 又被另一个中断请求打断了去运行 ISR2，而在 ISR2 中，又请求了一次线程切换，请求从线程 C 切换到线程 D，那么等到最终真正运行优先级最低的 PendSV 时 “from”变量里就保存着线程 C 的栈顶了，而“to”变量里就保存着线程 D 了。“to”里保存线程 D 的栈顶没问题，因为切换到最近一次请求的线程本来就是正确的做法；但是要被压栈的线程只能是线程 A，因为切换前的当前线程就是 A，如果不去压栈 A 而是去压栈 C，那么线程 A 和线程 C 的上下文都被破坏了——A 的上下文没有被压栈保护，而 C 的上下文被多余压栈了一次，如**图 2.3.3** 所示。

要避免这样的问题，上下文切换代码中，变量“from”只能记录第一调用线程切换时的“当前栈顶指针”，而在 PendSV 之前，重复调用上下文切换代码时，只有“to”变量被更新，“from”变量不能被更新。我们修改上下文切换代码，用一个标志变量来标记是否重复切换：

```
// C 代码
rt_uint32_t rt_interrupt_from_thread, rt_interrupt_to_thread;
```

```
rt_uint32_t rt_thread_switch_interrupt_flag// 用来标记重复切换的标志
```

图 2.3.3 在 ISR 中切换线程有重复请求切换造成压栈错误的可能

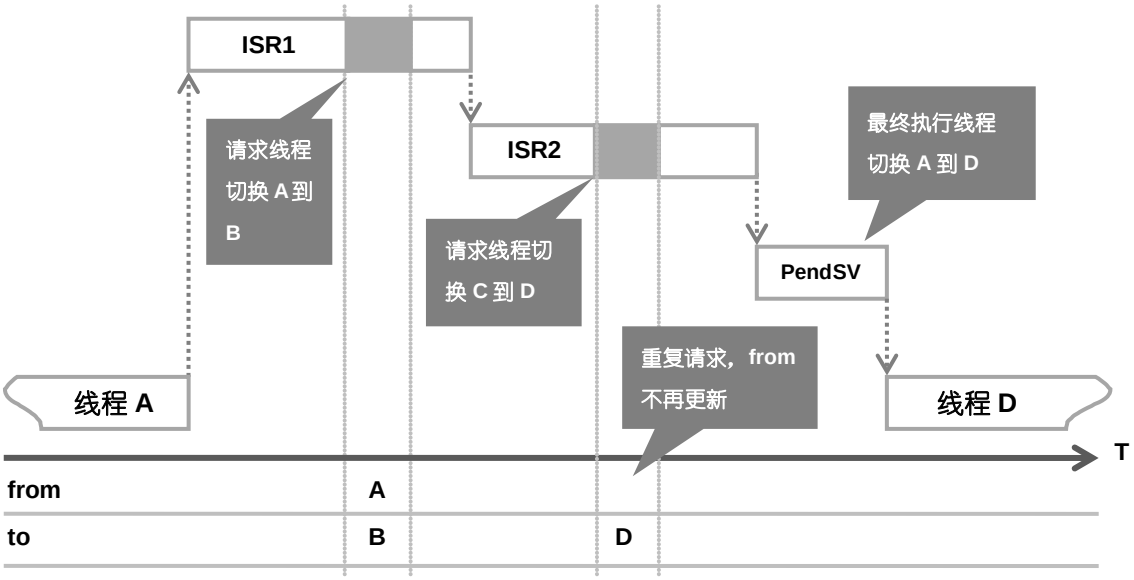



```
; 汇编代码
IMPORT rt_thread_switch_interrupt_flag ; 用来标记重复切换的标志
IMPORT rt_interrupt_from_thread
IMPORT rt_interrupt_to_thread

; * C 代码中调用上下文切换的接口
; * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; *r0 中保存 from 参数，r1 中保存 to 参数
EXPORT rt_hw_context_switch
rt_hw_context_switch:

    LDR     r2, =rt_thread_switch_interrupt_flag
    LDR     r3, [r2]
    CMP     r3, #1
    ; 若标志为 1，表示重复切换，跳过更改“from”指针的代码
    BEQ     _reswitch
    MOV     r3, #1
    STR     r3, [r2]                            ; 若标志不为 1，表示没有重复切换，将标志置为 1
    ;读取“from”栈指针到变量 rt_interrupt_from_thread
    LDR     r2, =rt_interrupt_from_thread
    STR     r0, [r2]

_reswitch:
    ; 读取“to”栈指针到变量 rt_interrupt_to_thread
    LDR     r2, =rt_interrupt_to_thread
```

```
    STR     r1, [r2]
```

修改后的上下文切换代码避免了重复切换的问题，就算发生了上述多个中断请求重复请求切换的情况，最终的切换结果是：线程从最先请求的“from”线程，切换到最后请求的“to”线程。示意图如**图 2.3.4** 所示。

图 2.3.4 正确记录当前线程指针以避免重复错误切换



细心的读者可能发现：“rt_hw_context_switch”的汇编代码里并没有关闭全局中断，会不会在允许中断嵌套时导致数据完整性问题呢？其实，读者大可不必担心。因为线程切换“rt_hw_context_switch”函数并不是由用户直接调用的，而是由操作系统调度器调用的。操作系统调度必须不被打断，因此调度请求发生时调度器第一时间就关闭了全局中断，然后运行调度算法，找到要切换到的线程，然后运行线程切换请求“rt_hw_context_switch”。因此，“rt_hw_context_switch”本身已经在临界区里运行了，不用担心数据完整性问题。反而，真正执行线程切换的 PendSV 服务程序才需要临界区保护——进入 PendSV 后首先记录当前中断状态、关闭全局中断，然后才进行手工压栈、出栈操作。

这里给出 RT-Thread 1.2.0 版本在 Cortex-M3 平台上完整的上下文切换代码，笔者加了一些中文注释在里面。有兴趣阅读最新源码的读者请参阅最新版本 RT-Thread 源代码中，关于上下文切换和系统调度部分。

```
IMPORT rt_thread_switch_interrupt_flag      ; 是否从中断中请求切换的标志
IMPORT rt_interrupt_from_thread             ; 记录“from”线程栈顶指针的变量
IMPORT rt_interrupt_to_thread               ; 记录“to”线程栈顶指针的变量

;/* C 中调用上下文切换的接口
; * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; * r0 中保存 from 参数
; * r1 中保存 to 参数
; */
    EXPORT rt_hw_context_switch_interrupt
```

```
    EXPORT rt_hw_context_switch
rt_hw_context_switch_interrupt:
rt_hw_context_switch:
    ; 将 rt_thread_switch_interrupt_flag 置 1
    LDR     r2, =rt_thread_switch_interrupt_flag
    LDR     r3, [r2]
    CMP     r3, #1
    BEQ     _reswitch     ; 若标志已经为 1，则直接读取“to”栈指针（“from”已被中断自动压栈）
    MOV     r3, #1
    STR     r3, [r2]
    ;读取“from”栈指针到变量 rt_interrupt_from_thread
    LDR     r2, =rt_interrupt_from_thread
    STR     r0, [r2]

_reswitch
    ; 读取“to”栈指针到变量 rt_interrupt_to_thread
    LDR     r2, =rt_interrupt_to_thread
    STR     r1, [r2]

    LDR     r0, =NVIC_INT_CTRL            ; 人为触发 PendSV 异常，让内核硬件自动压栈
    LDR     r1, =NVIC_PENDSVSET
    STR     r1, [r0]
    BX      LR

EXPORT PendSV_Handler
; 进入 PendSV 中断，psr, pc, lr, r12, r3, r2, r1, r0 被芯片内核自动压栈到“from”
PendSV_Handler:

    ; 关闭中断，保护上下文切换
    MRS     r2, PRIMASK
    CPSID   I

    ; 读取上下文切换标志 rt_thread_switch_interrupt_flag
    LDR     r0, =rt_thread_switch_interrupt_flag
    LDR     r1, [r0]
    CBZ     r1, pendsv_exit         ; 上下文切换标志若为 0，则已经切换完毕，直接退出异常处理

    ; 切换标志非 0，则将其清零
    MOV     r1, #0x00
    STR     r1, [r0]

    LDR     r0, =rt_interrupt_from_thread
    LDR     r1, [r0]
    ; “from”栈指针为 0，表示系统第一次调度，跳过对“from”手工压栈
```

```
    CBZ     r1, switch_to_thread

    MRS     r1, psp                 ; 获取“from”栈顶指针
    ; 将内核的r4~r11（非自动部分）压栈到“from”栈，并更新r1的值
    STMFD   r1!, {r4 - r11}
    LDR     r0, [r0]
    STR     r1, [r0]                ;将更新后的栈顶指针写回线程“from”控制块，“from”线程压栈完毕

switch_to_thread
    LDR     r1, =rt_interrupt_to_thread
    LDR     r1, [r1]
    ; 读取rt_interrupt_to_thread变量中保存的“to”栈顶指针到r1
    LDR     r1, [r1]
    LDMFD   r1!, {r4 - r11}         ; 从“to”栈出栈 r4 ~ r11（非自动部分）到内核，并更新r1
    MSR     psp, r1                 ; 将当前“to”栈栈顶指针写入内核栈顶指针，手工出栈并切换完毕

pendsv_exit
    ; 恢复中断状态
    MSR     PRIMASK, r2
    ORR     lr, lr, #0x04
    ; 退出PendSV中断，芯片内核自动完成psr, pc, lr, r12, r3, r2, r1, r0出栈
    BX      lr
; 切换完毕
```

2.4 RT-Thread 完整的移植范例

（您所阅读的仅仅是样章，并不包含所有内容，展现的内容也并不是最终形式，详细内容请参考正式书籍）

3

什么是“多任务”

[Put Introduction Here]

3.1 假并行，真并发

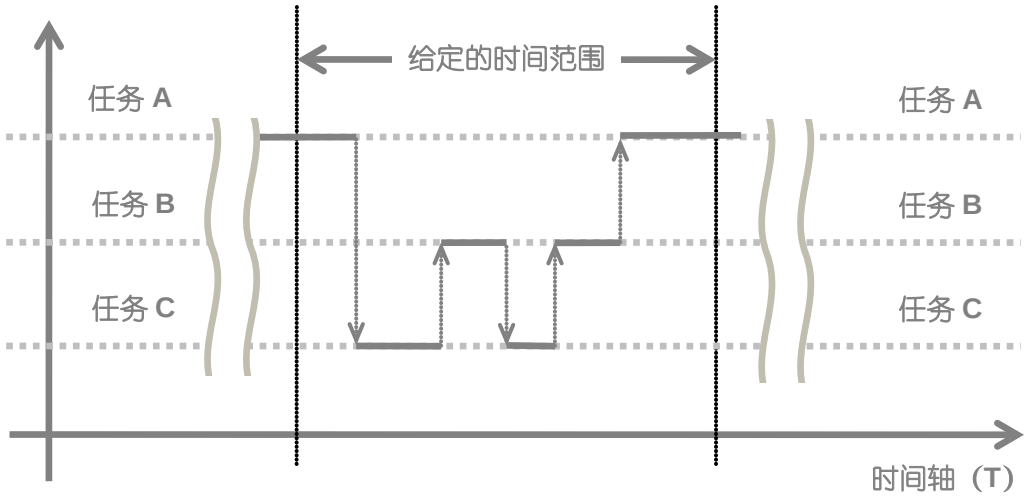
在开始详细讨论多任务的概念之前，我们首先要搞清楚两个概念：任务的并行（Parallel）和并发（Concurrent）。什么是任务的并行呢？

所谓的并行就是在给定的时间范围内，多个目标任务同时（或者有能力）同时得到执行。

如图 3.1.1a 所示，在给定的时间范围内，任务 A、B、C 同时在执行，我们就可以说这三个任务在该时间范围内是并行的。这种并行任务的定义方式是显而易见的，也较为严格。实际上，通常情况下在任意时刻多个目标任务有可能（或者说允许）同时执行，我们就可以判定这些任务是并行的。比如假设任务 A、B、C 实际上是执行在不同的内核上，大家就可以很自然的知道这些任务是并行的（或者说有能力并行的）而不需要每时每刻或者随机性的去划定一个时间范围，抽查他们是否真的在同时执行。

图 3.1.1 并行与并发

a. 给定时间范围内并行执行的任务



b. 给定时间范围内并发执行的任务

相对并行来说，任务的并发是一个较为宽松的概念，它只要求

在给定的时间范围内，目标任务都得到了执行，就可以判定这些任务是并发的。

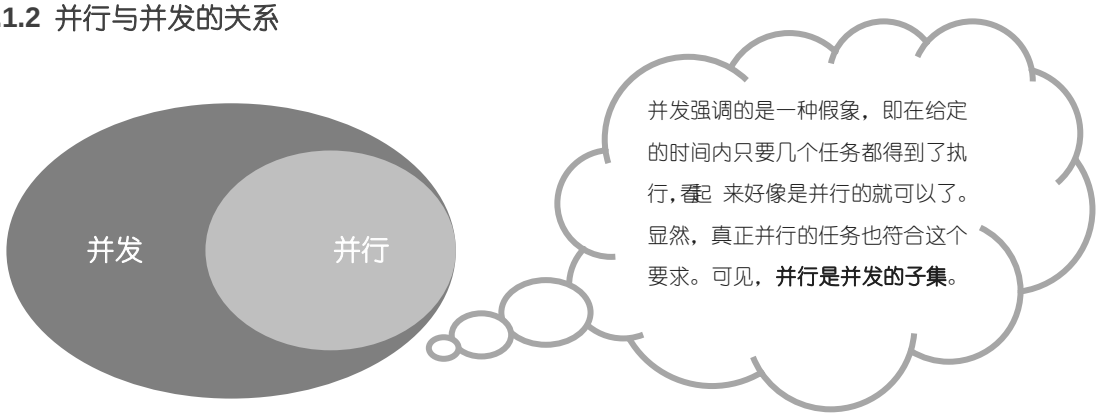
如**图 3.1.1b** 所示，任务 A、B、C 是以某种方式交替轮转执行的，任意时刻有且仅有一个任务得到执行——它们是串行的，但在给定的时间范围内，它们都得到了执行，因此任务 A、B、C 在这一时间范围内是并发的。

并发是一种假象，是一种虚假的并行的。在一个相对宏观的时间范围内由于多个任务都得到了执行从而产生了他们是“并行”的错觉。比如在 500ms 内你按下计算器的按钮，相应的数字显示在液晶屏幕上，从用户的视角来看，键盘扫描任务和 LCD 的刷新任务似乎是并行的，而实际情况是，这两个任务很可能只是超级循环里面轮转执行的两个函数。

```
// 计算器系统中多任务的并发
void main(void)
{
    system_init();                                  // 系统初始化

    while(1) {                                      // 超级循环
        ...
        scan_key_board();                           // 键盘扫描任务
        refresh_lcd_display();                          // 液晶刷新任务
        ...
    }
}
```

并发真正关心的是在给定的时间范围内多个任务是否都得到了执行，至于这些任务究竟是串行轮转的（Round-Robin）还是并行执行的或者部分任务是串行部分任务并行的——并发都不介意。容易想到，并行其实是并发的一个子集。严格并行的任务当然是并发的，因为在给定的时间内多个任务都得到了执行；但并发的任务却并不一定是并行的，因为并行强调的是多个任务在时间上发生重叠的能力。

图 3.1.2 并行与并发的关系



我们在选购笔记本电脑或台式机的时候，时常会听到 CPU 多核多线程的说法。比如 Intel(R) Core™ i7 L620 是 2 核 4 线程的处理器，这里 2 核表示 CPU 中存在两条流水线，可以并行执行两个线程；4 线

程表示 CPU 总共提供了 4 个寄存器页，可以同时保存 4 套不同的上下文，实现了我们在第一章中讨论的用空间换时间的策略从而节省了上下文切换的时间、提高了 CPU 的执行效率。在这样一个 CPU 中，任意时刻有且仅有两个线程是可以并行执行的，而四个线程则是并发执行的。

3.2 CPU 究竟跑的有多快

之所以用并发的方式来实现多任务是因为相对人的感官来说 CPU 跑的太快了，即便是人们常常用来描述时间短暂的“一眨眼功夫”对 CPU 来说也是及其“漫长”的好几百毫秒了——仔细想想有几个人能在一秒钟内连续眨十次眼睛呢？正因为如此，即便是超级循环里面顺次执行的多个任务，在人类看来也往往是“一瞬间就执行完了”。那么 CPU 究竟跑的有多快呢？是很快、非常快还是快得不得了？如果我们继续站在人类的视角考虑这个问题，其抽象程度无异于思考“无穷大究竟是多大”。让我们想象着周围的时间相对你突然都慢了下来，从微处理器的视角重新审视这个世界。要做到这一点，首先要借助一个等效的概念：

“1MHz 就是 1us”

“1MHz 就是 1us”是一个基准概念，通过修改思考方式，我们就可以利用它快速而有效的解决很多实际问题。作为练习，我们来尝试依次快速的回答以下几个问题：

假设每个时钟脉冲都对应一个指令周期：

- 已知系统频率是 1MHz，请问 1us 内有几个指令周期？
- 已知系统频率是 12MHz，请问 1us 内有几个指令周期？
- 已知系统频率是 11.3728MHz，请问 1us 内有几个指令周期？
- 已知系统频率是 500KHz，请问 1us 内有几个指令周期？

很显然，如果你试图首先计算出系统周期（$T = \frac{1}{f_{cpu}}$ S），再用 1us 去相除（$N_{cycles} = \frac{0.000001}{T}$），这个过程已经慢了。让我们来换一种思维模式，既然 1MHz 对应 1us（也就是 1us 对应一个指令周期），那么 12MHz 就是 1MHz 的 12 倍，1us 时间内就有 12 个指令周期；同理可得，当系统频率分别是 11.3728MHz 和 500KHz（0.5MHz）的时候，1us 时间内对应的指令周期数分别是 11.3728 个和 0.5 个。

借助这个等效，我们就可以对 CPU 的处理能力建立更多量化的感官，比如 1ms 的时间内，CPU 能做多少事情呢？由于 1ms 等于 1000us，对 1MHz 的系统来说，1ms 可以完成 1000 个指令周期，12MHz 的系统可以完成 12000 个指令周期。然而 1000 和 12000 这样的数字对于只有十个手指的人类大脑来说还是太抽象了，因此我们更进一步，把指令周期换算成等效的代码尺寸：

- 由于主流的微控制器其指令集中大多是单周期指令，我们不妨假设所有指令都是单指令周期的，这样 1 个指令周期就对应一条指令；
- 假设每条指令都是 2 个字节大小（16 位指令）；

这样，1ms 时间内 1MHz 的系统可以运行大约 2KB 的代码，一个 12MHz 的系统可以运行 24KB 的

代码，依次类推。那么 2KB 是什么概念呢？如果你平时有留意编译后的代码尺寸，2KB 大约是一个基础驱动库的尺寸，可以包含一个 USART 的驱动或者实现电源管理；而 24KB 几乎是一个小型工程应用的尺寸了。

借助这些非常具体的数字，我们很容易拿它们和中断处理程序进行比较，建立直观的认识——比如中断处理程序“执行的是不是足够快”、“丢中断的风险究竟有多大”等等——可以肯定的是，这种忽略循环和条件分支的评估方法几乎是一个代码的最差情况，也就是说，在 1MHz 的系统中对于一个 1KHz 的毫秒中断，中断处理程序越接近 2KB，就说明系统越可能“丢中断”。在这种情况下，除非你通过编译器提供的等效汇编代码仔细的计算过实际的周期数,确信时间上处理周期不会大于 1ms 且这期间不会存在其它中断处理程序，否则你的中断处理程序还是比 2KB 越小越好。

我们来看一个实例：评估一个 10KHz 的外中断，中断处理程序允许的理论最大安全尺寸是多少？

首先，我们要搞清楚系统的指令大小和指令集的周期数情况。以 ARM Cortex M3 为例，其指令大部分为单周期指令，支持 16 位指令和 32 位指令。为了评估中断处理程序的尺寸上线，我们可以分别以 16 位指令和 32 位指令为基础计算出两个结果作为参考范围；

其次，我们要搞清楚系统频率。假设系统频率为 72MHz，已知 10KHz 等效于 100us，则中断处理程序的理论最大尺寸范围是（72 * 100 * 2）字节到（72 * 100 * 4）字节，即 14.4KB 到 28.8K 之间。取最小值 14.4KB。

结论，中断处理程序及其调用的子函数，其尺寸总和至少要小于 14.4KB 才能确保 10KHz 的中断得到及时的响应。由于未考虑循环、分支以及其它任务的存在，以上结果仅用于粗略的快速评估，实际代码通常应该远小于这一上线值。当实际尺寸接近或者超过 14.4KB 时基本可以判定系统无法及时稳定的响应中断。

“1MHz 就是 1us”的等效为我们提供了一个基准，建立了关于“CPU 跑多快”最直观的感受，同时也为评估代码尺寸、系统可靠性提供了有力的参考。掌握了这个基准，作为一个合格的程序员，不应该仅凭人类的感觉毫无依据评价 CPU 的处理能力了，“72MHz 足够快了吧？”“我已经用了芯片的最高频率”这种话再也不能轻易说了，我们应该定量而不是定性的去看待这类问题。

3.3 什么是多任务

对于只有一条流水线的单核系统来说，任意时刻只有一段代码能够得到执行，以（OpCode）指令为载体的多个任务必然是串行执行的，无论它们使用何种策略进行调度和轮转（即便中断也不例外），这些任务都是并发的。简而言之，

对单核单流水线微控制器来说，并发是实现多任务的唯一方式。

那么在微控制器中是否存在真正并行执行的任务呢？答案是肯定的。抛开 ARM Cortex A 系列的大小核（big.LITTLE）结构不说，即便是单流水线单内核的“单片机”系统，也存在真正的并行任务。要理解这一概念，首先要搞清楚什么是“任务”。

3.3.1 任务和任务的载体

任务其实是一个简单的概念，表示“需要完成或者实现的某件事情”——简而言之“要做的事情”。研究如何完成任务的过程就是程序设计。任务的完成有多种不同的实现方式，或者说任务的载体可以是多种多样的。比如“使用 SPI 通信协议发送一串数据”这是一个任务，要实现这个任务我们至少有两种

方式：1）使用专门的 SPI 硬件外设来发送数据；2）使用软件模拟的 SPI 驱动来发送数据。对前者来说任务实现的载体是硬件外设（Peripheral），而对后者来说任务实现的载体是内核指令（OpCode）。

当我们拥有多个任务载体时，并行就成为可能。显然，我们经常一边通过硬件 SPI 与外界通信，一边使用内核执行别的任务，这两个任务毫无疑问是并行的。回头来看，其实无论是增加内核数量，还是增加外设资源，本质上都是增加任务的载体，最终让更多的任务并行起来换取更高的系统性能。也许你已经习惯了将外设看做是资源（Resources）而不是任务的执行者，但只要你对嵌入式微控制器的发展历史稍加研究就会发现，每一次 MCU 系统性能的提升都是通过增加任务的载体（比如外设）——让更多的任务并行起来——实现的。当你体会到这一点，就会很快意识到，让内核死等某个外设完成它的任务是多么“愚蠢”的行为。

作为任务载体，外设通过牺牲灵活性的方式换取最大的执行效率，因为他们就是“为了追求效率而固化下来的确定的逻辑（Determined Logic）”与之相对，内核则是以牺牲效率来换取最大的灵活性——它几乎为程序员提供了无限的可能，使得他们可以编写代码以适应各种各样不同的任务。从诞生之初，外设和内核走的就是截然不同的两个方向，所以单论效率来说，理论上内核永远无法超越专注于本职工作的外设。

对外设来说，效率已成定局；对内核来说，如何提高效率，则是我们关注的焦点。有太多的文章和模型讨论单个任务的设计实现和效率问题。本文关注的是多任务的设计实现和效率问题。这也是我们随后章节讨论的出发点。

3.3.2 多任务是怎么实现的

内核流水线是依靠内核时钟（CORE_CLK）来驱动的，因而，

在内核上实现“多任务”不过是一个处理器时间（CPU Time）的资源分配问题，

对于这种分配我们有一个专业的说法叫做“调度”（Schedule）。

前面我们已经说过，在单内核单流水线的系统中，多任务必然是通过并发的方式实现的。简单地说就是在一个较小的时间范围内，多个任务都得到了执行，从而产生了这些任务是“并行执行”的错觉。这实际上已经告诉我们通过并发来实现多任务的具体操作方式：

- 将原本连续执行的任务拆分成很多细小的片段
- 在一个系统周期内，每个任务都只执行一个片段，使得在这个系统周期内多个任务都能得到执行从而实现并发
- 重复这个系统周期，直到完成所有任务

对于并发的原理，这里已经非常清晰，无需更多的讨论。但容易注意到的是 1）如何将原本连续的任务拆分成细小的片段，以及 2）在一个系统周期内如何选择参与执行的任务，则是需要进一步讨论的。针对这两个问题的讨论就是针对“如何分配处理器时间”的讨论，也就是针对“如何调度任务”问题的讨论。我们把处理这两类问题的策略称之为“调度策略”。概念上存在两类极端的调度策略：

- 手动调度（Compile-time Schedule）

  1) 手动调度，顾名思义，手动地进行任务调度。那又是“谁”手动的进行任务调度呢？答案是：程序员——程序员在代码编写的时候人为的将任务拆分成细小的片段，并决定任务的调度策略，如轮询（Round-Robin）等等。由于任务的拆分、任务的调度策略在程序的编译时

刻就已经由程序员通过代码固化了下来，因此又称为编译时刻调度（Compile-time Schedule）。手动调度的应用范围及其广泛，由广大嵌入式程序员自觉和不自觉的在裸机系统中大规模应用和实践。手动调度中用于任务拆分的方法很多，比如有使用限状态机（FSM），或者使用 protoThread 代码模板辅助等等，这里就不再赘述。

2)

- 自动调度（Runtime Schedule）

3) 自动调度是指借助操作系统（OS）在运行时刻（Runtime）通过某种调度算法自动的进行任务拆分和调度的方式。由于任务的拆分工作是在运行时刻由操作系统自动完成的，因此又称为运行时刻调度（Runtime Schedule）。本质上来说，自动调度并没有比手动调度做更多的事情，它只是借助操作系统代码通过消耗一定的系统资源，将程序员从手动拆分任务的繁重劳动中解放出来——自动的进行任务拆分，从而大大简化了多任务开发的难度，降低了系统开发的门槛。

比较两种调度方式，有几点需要特别注意：

- 多任务的实现只与处理器时间的分配有关，与是否使用操作系统无关。裸机也可以支持多任务。
- 手动调度是将任务调度的工作交给了程序员去完成，提高了代码开发的复杂度和难度但是降低了系统资源的消耗（无需额外的 FLASH 和 SRAM 开销用于实现操作系统，也无需额外的处理器时间用于运行调度算法）——通俗的说，手动分配就是苦了程序员轻松了处理器。在资源相对紧缺的环境中，手动分配是相当划算的。
- 自动调度以系统资源消耗为代价实现了傻瓜化的任务拆分和调度，降低了多任务系统的开发难度，简单说就是，轻松了程序员苦了处理器。在资源相对宽松的环境中，自动调度是不二的选择。

多任务是一个概念，与是否使用操作系统无关。

如果非要用操作系统的概念去理解的话，可以认为手动调度是将操作系统安装在了程序员的大脑里，由程序员通过代码编写的方式固化处理器时间的分配策略；自动调度是将操作系统实际安装在系统中，由处理器消耗自己的系统资源（包含处理器时间）在运行时刻实现对任务的拆分和调度。实际应用中，常常存在介于裸机和操作系统环境的应用场景，比如使用非抢占的合作式调度器的环境、使用状态机调度器的环境等等。我们可以简单的把这类环境理解为介于手动调度和自动调度之间的情况——既然手动调度是将操作系统安装在程序员的大脑里，而自动调度是将操作系统安装在芯片里，那么介于二者之间的情况就是对二者特点或多或少的折中，理解为程序员借助某些实体程序来协助自己进行处理器时间的分配和管理。

3.3.3 多任务系统设计的两大核心问题

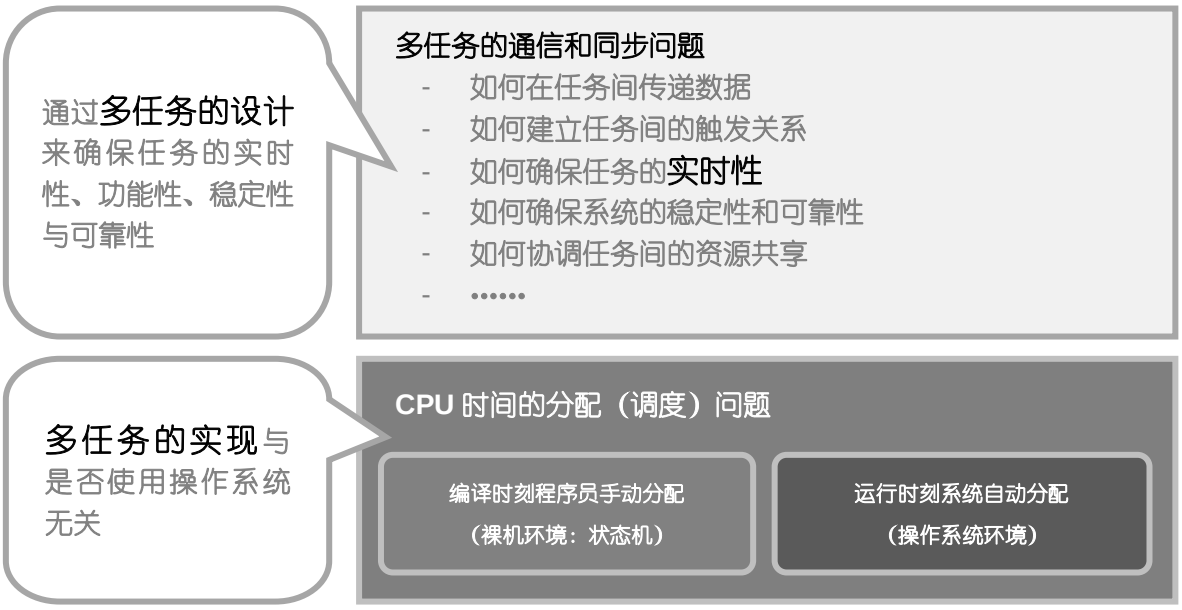
通过前面的章节，我们了解到实现“多任务”只是一个处理器时间的资源分配问题，无论是通过手动还是自动的方式，或者说无论是裸机还是操作系统环境下，我们都能实现多任务——这只解决了多任务系统设计所面临的**第一个核心问题**。

多任务，就像团队中的不同成员，他们如何沟通、如何交流、如何相互协调来完成一个更大的任务目标，是多任务系统设计的**第二大核心问题**，我们通常把这类问题统称为“多任务的通信和同步问题”。如图 3.3.2 所示，多任务系统设计的两大问题存在明显的层次关系。需要注意的是，无论以何种方式实现调度，多任务的通信和同步问题都是存在的，它是更高层面上的问题，是与具体多任务的实现方式（裸机还是操作系统）无关的。

多任务的通信和同步问题是一个概念问题，是一套通用的思维方式，与具体的实现方式无关。

多任务的通信同步问题究竟解决哪些具体问题呢？通常来说主要包含以下几个方面：

图 3.3.2 “多任务”涉及到的两大核心问题



- 任务之间如何安全的传递数据？
- 一个任务如何触发其它一个和多个任务？
- 任务的实时性如何得到保证？
- 如何能确信的说，这么多任务一起执行的情况下系统是稳定的？
- 如何确保多任务环境是可靠的？
- 多个任务竞争有限资源的时候怎么办？
- 在一个已有的多任务应用中，如何安全的修改和更新已有的任务？
- 如何在一个多任务应用中，安全的增加新的任务？
- ……

3.4 小结

多任务程序设计是应用设计的工作主体，除了上述列举的问题以外，我们还会遇到更多更具体更细节的技术难点，而实际中问题通常是结伴出现的。多任务系统的设计是困难的，它的调试更像是从一个揉乱的毛线球中找出断掉的那一根，而很多时候系统已经复杂到足以让我们放弃“把毛线球梳理清晰”这一想法的程度——对做工程来说，快刀斩乱麻只是一个笑话，不相信你斩一个试试？——根本不是老板找你麻烦的问题，而是你根本不知道该从什么地方下手。这本书编写的意义就在于，它试图从两个方向简化多任务系统设计：

- 从正向开发的角度来说，本书试图通过介绍多任务设计的思维方式、原则和理论模型，使得读者有能力在系统设计时保持思路清晰：不仅能快速实现功能，且确保系统在效率、质量、稳定性、可维护性上都能做出符合任务场景需求的平衡。

这不禁让我想到，做了多年工程之后终于明白一个即能实现应用需求又同时在效率、尺寸、质量、稳定性、可维护性等等所有软件工程书籍上能找到的技术指标都同时做得好的系统，在实际工程实践中是不存在的——它们只是某些学术论文摘要上修饰性的文字。

- 从系统分析的角度来说，本书试图通介绍实用的多任务系统建模方法来帮助维护人员快速的掌握系统的结构，并利用模型的便利，精确而迅速的检验系统的可靠性，定位出存在的问题，并通过由模型得出的参考意见给出最直接的改进方案。

这不是一本介绍操作系统原理的书，这不是一本教会你如何动手编写自己操作系统的书，这是一本系统介绍多任务系统设计思维和方法的书。它解决最实际的工程问题，为读者补充从事多任务开发所必备的理论知识。严格意义上说，这里才是本书前言的结尾，经历了整整三个章节，我们才刚刚有能力一层一层的为您解释清楚摆在我们面前的是怎样一个工程问题。我想您现在应该能够理解，在充斥着那么多的误解，充斥着那么多可能的知识点缺失的情况下，直接的去看操作系统的源代码，其实并不解决任何实际问题——是的，实际问题就是，我们做工程，我们必须要用多任务的方法来完成应用设计，我们并不是要学会如何编写自己的操作系统。

千里之行始于足下，使用多任务系统进行应用设计，随后的章节我们将从一个简单的命题开始逐步为您拆解多任务应用设计的要点、难点和常见问题，力求为大家建立一套切实可行的方法和思维方式。

什么是“共享资源”

[Put Introduction Here]

4.1 怎么研究多任务的行为

思考多任务程序设计，研究多任务的同步和通信问题，首先要研究的就是多任务的行为。简单地说，研究一个问题，首先要能观察到现象，排除干扰因素，然后才能知道问题出在哪里，才有可能思考解决方案——这是发现问题的阶段。在解决问题的阶段，我们通常会根据先前的观察做出自己的判断，提出解决方案，然后将这些方案放到同样的环境中加以实现，通过观察结果对方案进行验证。这是一个迭代的过程：

1. 通过观察发现问题；
2. 提出解决方案并加以实现；
3. 通过观察来判断方案是否有效。如果发现新的问题，则跳转到步骤 2)；如果发现所有问题都得到了满意的答案，则整个过程结束。

不难发现，对现象的观察在整个过程中起着举足轻重的作用。然而，程序设计原本就是一个抽象的过程，程序的行为无法像自然科学那样通过实验直接进行观察；多任务又是一个复杂的多元关系，普通的调试方式（单步/断点调试、Trace 调试等等）都无法直接的将其中的关系清晰的揭示出来，那么，在这种情况下，我们又如何去观察多任务的行为呢？——工欲善其事必先利其器，我们首先要做的是拥有一个多任务系统的模型。在接下来的篇幅中，我们将会以任务间上下文的切换关系为视角，建立一套可视化的“任务平面模型”，从而为日后研究和讨论多任务同步和通信问题提供基础和依据。

4.2 什么是任务平面（Task Plane）

任务平面是一种对任务的分类方法。在解释任务平面的概念之前，我们首先来下一个定义：

在任意时刻，当任务 A 运行的时候，任务 B 无法将其打断，我们就说任务 A 对任务 B 具有原子性。

这个概念很好理解，裸机环境下，假设有两个任务函数 A 和 B：任务 A 在中断处理程序中被调用，任务 B 在超级循环中被调用。任意时刻，当中断处理程中的任务 A 执行的时候，超级循环里的任务 B 都没有可能会打断任务 A 的执行。RT-Thread 环境下，假设有两个优先级不同的任务 A 和 B，其中 A 优先级较高。任意时刻，当任务 A 执行的时候（未被阻塞），低优先级的任务 B 都不可能打断任务 A 的执行。

如果存在任务 A 对任务 B 具有原子性，且任务 B 对任务 A 也同时具有原子性（即，任意时刻任务 A 和任务 B 都相互无法打断），则我们称任务 A 和任务 B 互有原子性。

互有原子性的任务在同一个任务平面上。

换句话说，由彼此互有原子性的任务构成的集合称之为任务平面。任务平面是一个集合，由一堆彼此都不能相互打断的任务构成。通过任务平面的定义，回过头来重新审视嵌入式软件开发，我们容易发现：

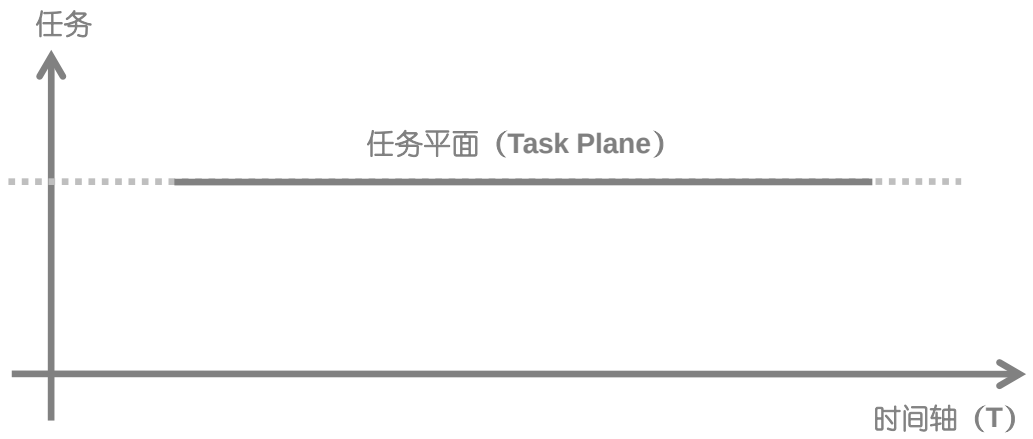
- 裸机环境下，在未开启中断嵌套的情况下，具有相同优先级的中断处理程序在同一个任务平面上。每个中断优先级都是一个任务平面。
- 裸机环境下，超级循环里阻塞运行的任务，一个任务完成之前不允许别的任务运行，因此相互具有原子性——在同一个任务平面上。
- 裸机环境下，超级循环里并发运行的任务，在每个循环周期内所有的任务均只执行一小部分，从每

个任务的视角来看，任务执行的过程不停的被打断，因而这些任务彼此之间不具有原子性——处于不同的任务平面上，且每个任务都独占一个任务平面。

- 操作系统环境下，如果任务的优先级都必须不同，则每个任务都独占一个任务平面，每个优先级都是一个任务平面，例如 µcOSII。
- 操作系统环境下，如果任务的优先级允许相同，且同优先级任务彼此不能打断时，拥有相同优先级的任务在同一个任务平面上，每个任务优先级都是一个任务平面。
- 操作系统环境下，如果任务的优先级允许相同，且同优先级任务彼此能够打断，则所有任务都独占一个任务平面。这种情况我们在后续的模型中有专门的分类和讨论。

类似的例子还可以举出很多，由于任务平面本质上是从上下文切换的视角定义的（通过允许或不允许上下文切换来划分），容易证明所有的多任务系统都可以使用任务平面的概念进行建模。我们通常将任务（Task）相对时间（T）为函数的图像称之为“TT 图”，其中横轴是时间轴，表示处理器时间；纵轴是任务轴，用来标注不同的任务。如**图 3.1.1** 就是一个典型的 TT 图。引入任务平面的概念以后，由于同一任务平面上的任务在时间轴上不可能发生重叠，因而习惯上将这些任务标注在相同的任务轴高度上（如虚线所示），这样所谓的任务“平面”在 TT 图上就变成了 “直线”（如**图 4.2.1** 所示）。任务平面实际上将 TT 图上的任务加以归类，引入了层次的概念，更便于观察和分析，TT 图也因此成为我们分析任务结构模型的利器。

图 4.2.1 任务平面的 TT 图（任务/时间函数图）



针对任务平面的定义，很多人也许会思考一种特殊的情况：任务 A 和任务 B 互有原子性，在同一个任务平面上；任务 B 和任务 C 互有原子性，在同一个任务平面上；是否可以推断出 A，B，C 一定在同一个任务平面上呢？答案是否定的。

任务平面的定义不具备传递性。

怎么理解这个概念呢？首先针对这个例子，任务 A 和任务 B 互有原子性，说明 A 和 B 相互不可打断，这是简单的二元关系；同理，对 B 和 C 来说也是这样。这并不能决定任务 A 和任务 C 也互有原子性。实际应用中，上述情况很可能是 A 和 B 通过信号量 x 进行了互斥（mutex），任务 B 和任务 C 通过信号量 b 进行了互斥，而 A 和 C 之间并不存在任何互斥量——当 B 结束运行时，理论上 A 和 C 都可以

运行（信号量 a 和信号量 b 都得到了释放），A 和 C 之间在缺乏约束的情况下当然会存在“一方被另外一方”或者“相互打断”的情况。

那么，在这个例子中 A、B、C 应该如何划分任务平面呢？有两种方式：A 和 B 属于同一个任务平面，而将 C 划归在另外一个任务平面上；或者 B 和 C 属于同一个任务平面，而将 A 独立出来。然而，从任务开发的角度来说，处理这类问题存在一个重要的原则，称为“任务多元化原则”，简单说就是进行多任务开发时，为了保证系统的可靠性，应该假设每个任务彼此都是可以相互打断的，在这种“最坏”假设下开发出来的代码显然适应所有的情况。关于“任务多元化原则”的具体内容将在随后的章节详细为您展开。

通过对任务的初步划分容易发现：某些任务平面具有相同的特性（比如具有优先级的层次特征）；某些任务平面的行为特征是完全不同的（比如有些任务平面有多个任务，有些则只有一个任务）。在随后的章节中，我们将首先对任务平面进行分类，然后基于这些分类深入的讨论它们的性质和用途。

4.2.1 小结

- 任务平面是一种任务的分类方法，我们将那些运行时彼此不能打断的任务放在一起，叫做一个任务平面。通过分类，我们就能获得一个或多个任务平面。
- 任务平面的引入让我们从研究任务与任务之间的关系变成研究一堆任务与另外一堆任务之间的关系。
- **“**一个任务平面上的任务打断了另外一个任务平面上的任务**”** 这种说法太麻烦了，以后我们就简单的说 **“**一个任务平面打断了另外一个任务平面**”**。
- 同一个任务平面上的任务 **“**谁也不招惹谁**”“**和平相处**”**，**“**你做你的，我做我的**”**，因而关系简单，没什么好操心的。
- 多个任务平面堆叠在一起就像是很多个平行宇宙——某个世界中保险箱里的奶酪，对另外一个世界的 **“**人**”** 来说可能就放在身边的桌子上。你可以体会一下这种感觉，然后再去思考不同任务平面上的任务如何看待他们共同访问的资源的——比如全局变量，比如外设寄存器等等。
- 我们更关注任务平面与任务平面之间的关系，而不那么在意同一个任务平面上任务与任务之间的关系——因为同一个任务平面上的任务彼此之间的关系太简单了——**“**彼此不能打断、互不干涉**”**——这也是为什么我们将这些关系简单的任务归类在一起并给了它们一个统一的名字 **“**任务平面**”**。

4.3 一切从数据完整性开始

嵌入式系统中，有一类资源具有以下特点：

- 对资源的访问由多个子步骤组成；
- 子步骤之间是可以打断的，即内核对资源的访问不具有原子性；
- 读取的步骤和写入的步骤是分开的，我们用 $READ_n$ 表示标号为 n 的读取子步骤；用 $WRITE_n$ 表示标号为 n 的写入子步骤，n 取大于 2 的整数。

由于对这类资源的访问是可以打断的——进行中的读取操作可能会被打断并插入写入操作，反之亦然——在操作缺乏原子性保证的情况下，很难设想这些读、写操作的子步骤交错在一起会造成怎样的后果（如**图 4.3.1** 所示）：

图 4.3.1 读取和写入子步骤发生不期望的交错

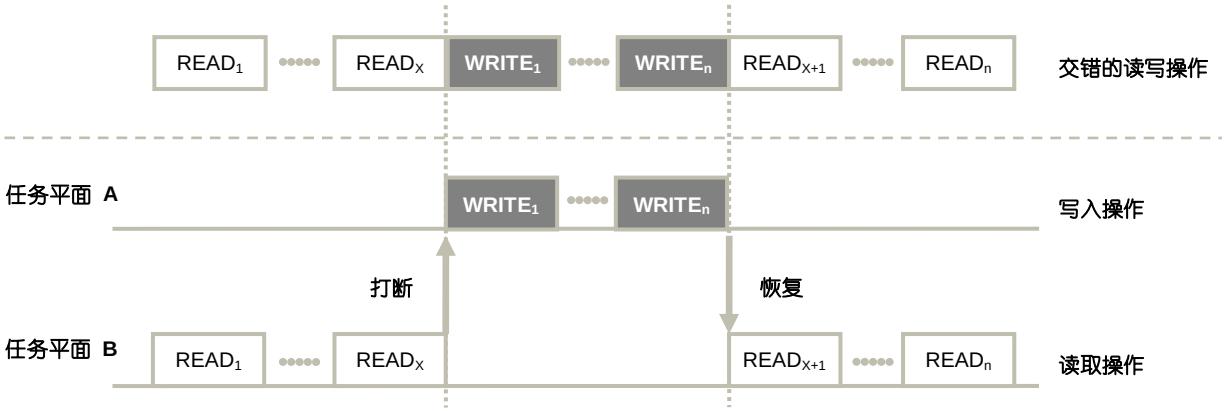


图 4.3.1 展示了读写交错的一个典型例子。首先，针对该资源的读取和写入操作都需要多个子步骤来完成，分别由 $READ_1$ 到 $READ_n$ 和 $WRITE_1$ 到 $WRITE_n$ 表示；其次，操作该资源的任务分别位于两个不同的任务平面上（图中任务平面 A 和任务平面 B），读写操作的原子性无法保证——因此一个很可能发生的情形是：WRITE 步骤打断了正在进行中的 $READ_X$，更新了目标对象的值，$READ_X$ 读取到的是这个资源的旧版本，而 $READ_{X+1}$ 及其之后的子步骤读取到的是该资源的新版本，此时我们说这个资源的值在读取的过程中发生了篡改（更新），数据完整性遭到了破坏。

“我的程序逻辑没有问题，但读到的数据莫名其妙的被改掉了。”
“我的显示程序总是随机的显示错误的值。”

这是程序员经常抱怨的一类问题，其中很大一部分是由数据完整性遭到破坏所引起的。让我们来看一个现实应用中的例子：

在 8 位环境下，受到字长的限制，内核处理多字节类型数据需要多个步骤（指令），比如操作 16 位数据要 2 个步骤，操作 32 数据要 4 个步骤。这些步骤不具有原子性。前后台系统（二阶任务平面）中中断处理程序 ISR(ADC_vect)负责从模数转换器 ADC 读取 16 位的采样结果，存放到静态变量 s_hwADCValue 中；运行在超级循环里（主任务平面上）的函数 refresh_screen_task()读取静态变量 s_hwADCValue 的值以获取采样结果，实时地显示在屏幕上，其核心代码如下：

```
static uint16_t s_hwADCValue = 0;
…
/*! AD 转换的中断处理程序 */
ISR(ADC_vect)
{
    s_hwADCValue = get_adc_result();        /*!<获取 AD 采样结果 */
}
…
void main(void)
{
```

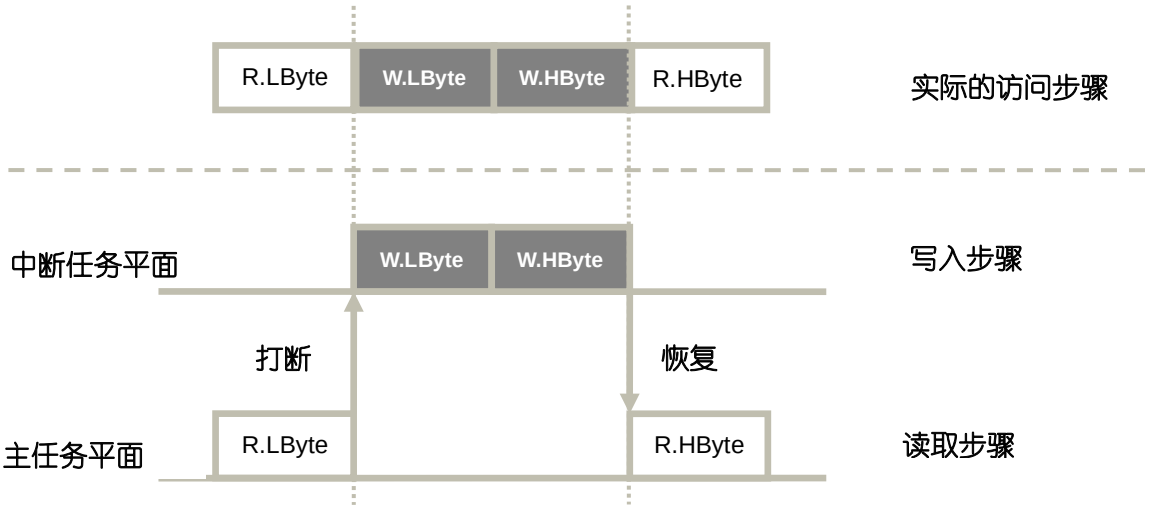
```
…
    /*! 主循环 */
    while (true) {
        …
        refresh_screen_task(s_hwADCValue);   /*!<刷新显示输出 */
        …
    }
}
```

由于字长的限制，8 位机对 16 位数据的访问需要分为高低字节独立进行：中断处理程序需要两个子步骤将数据写入 s_hwADCValue；显示刷新函数 refresh_screen_task 也需要两个子步骤来完成变量的读取——这就满足了“对资源的访问由多个子步骤够成”的条件。由于中断处理程序可以打断主循环的执行，则存在一种可能（如**图 4.3.2** 所示）：

- refresh_screen_task()在读取 s_hwADCValue 低 8 位时被 ADC 中断程序打断；
- 中断处理程序对 s_hwADCValue 进行写入操作，更新了原有内容；
- 系统从中断处理程序返回，refresh_screen_task()继续之前被打断的了的操作——读取 s_hwADCValue 的高 8 位；

最终 refresh_screen_task()函数实际读取到的变量值是由老版本 s_hwADCValue 的低 8 位和新数据的高 8 位拼接起来的无效数据——我们说变量 s_hwADCValue 的数据的完整性被破坏了。

图 4.3.2 一个典型的前后台系统下数据完整性被破坏的例子



默认情况下，编译器喜欢将变量放置在对齐到处理器字宽的地址上，这是因为很多内核被设计成访问小于或等于字宽的数据类型时“仅需要一个操作”以提高数据处理效率，这样的操作很可能是不可打断的，因而具有“天然原子性”，例如，16 位机的字宽是 16 比特，当变量对齐到偶数地址时，内核对 bool、uint16_t、int16_t、uint8_t 和 int8_t 类型的数据进行读写操作时仅需一个操作；同理，32 位字宽的系统，例如 ARM，当变量的字节地址对齐到 4 的倍数时，内核对小于或等于 32 位宽度的数据访问时有很大的可能具有天然原子性。天然原子性现阶段还不是事实上的标准，得不到指令集的保证，因而编

写强调可移植性的代码时不可以依赖它。天然原子性仅作为系统行为分析的一种现象在代码分析和 DEBUG 中存在意义。

不仅限于变量，某些数据结构实体资源也会由于读写交错造成数据完整性问题，例如环形队列：环形队列的正常运行依赖于一个“头指针”（Head）和一个“尾指针”（Tail），“队 ”和“出队”操作分别要正确维护这两个指针的值才能保证环形队列不会出错。一个环形队列的入队示例代码如下：

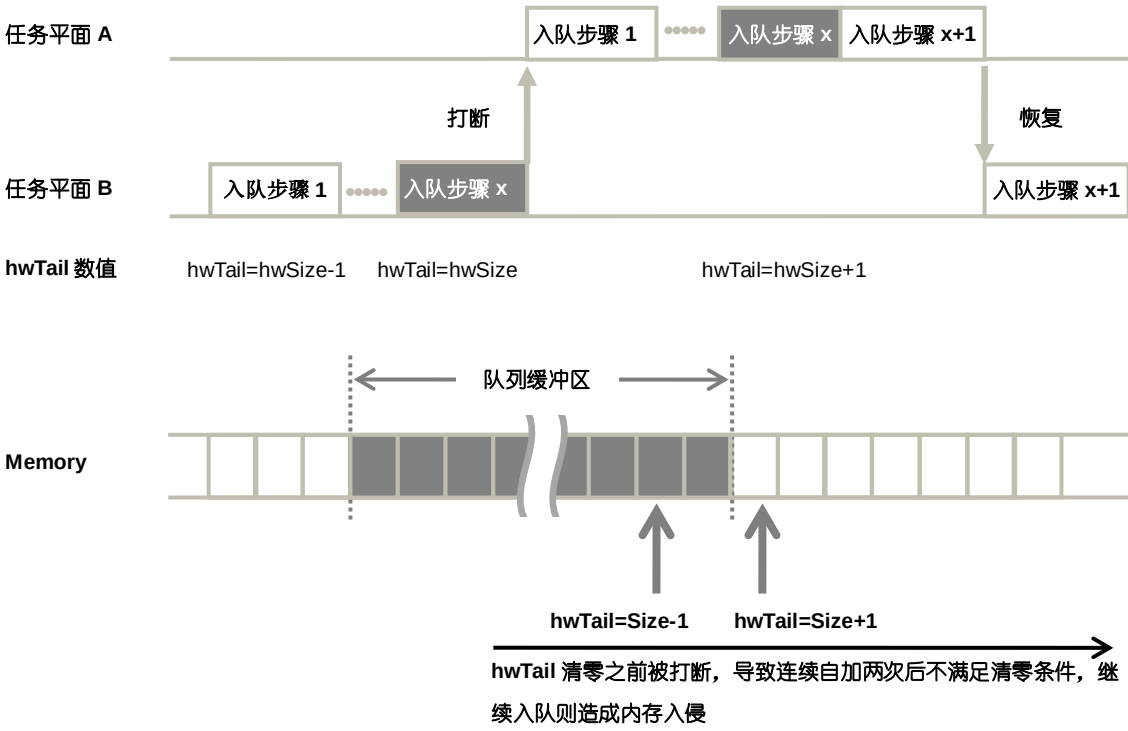
```
//! 入队操作的实现
bool enqueue(queue_t * ptQueue,uint8_t chByte)
{
    if(ptQueue == NULL) {
        return false;
    }
    if((ptQueue->hwLength > 0) && (ptQueue->hwTail == ptQueue->hwHead)) {
        return false;
    } else {
        *(ptQueue->pchBuffer + ptQueue->hwTail) = chByte;
        ptQueue->hwLength++;
        ptQueue->hwTail++;                          /*!<步骤 x */
        if (ptQueue->hwTail == ptQueue->hwSize) {   /*!<步骤 x+1 */
            ptQueue->hwTail = 0;
        }
    }
    return true;
}
```

入队操作需要多个步骤才能完成，其中对于尾指针“hwTail”的关键操作我们在示例代码中记做步骤 x 和步骤 x+1。在步骤 x 中 hwTail 指针自加，步骤 x+1 判断 hwTail 是否将要溢出，如果溢出则在步骤 x+1 中将尾指针清零复位。以 hwTail 等于 hwSize-1 的情况为例，由于入队操作本身不具有原子性，那么必然存在一种可能性：

1. 步骤 x 运行之后，hwTail 自加，其值等于 hwSize，而这时被高阶任务平面打断；
2. 高阶任务平面中，又进行了一次入队操作，运行到步骤 x，这时 hwTail 的值等于 hwSize+1，步骤 x+1 中条件不满足，hwTail 没有清零复位；
3. 系统从高阶任务平面返回，继续运行之前被打断了的操作——原任务平面中的步骤 x+1，而此时 hwTail 的值等于 hwSize+1，仍然不会复位清零；

问题这时已经发生了：环形队列的缓冲区大小只有 hwSize 大小，而队列的尾指针 hwTail 值已经大于 hwSize，之后每一次入队操作都会造成内存入侵而导致程序出错，而且 hwTail 的值只会越来越大，内存入侵越来越深，直至程序跑飞。“队列”这个资源的数据完整性被破坏了，如图 **4.3.3** 所示。

图 4.3.3 队列的数据完整性被破坏示例



以 RT-Thread 操作系统下具体的应用场景为例：某系统中有一个 sensor hub 模块——将众多传感器数据用统一的抽象数据类型 sensor_data_t 集合在一个队列中，供其他模块使用；每个传感器有自己的采样任务 sensor_task_n，用以将数据采样并入队汇总；enqueue 函数是 sensor_data_t 类型的入队函数；各个传感器任务处于不同的任务平面上，以满足不同传感器各自的优先级。这个 sensor hub 的实现代码如下：

```
/* Sensor_1 采样任务 */
void sensor_task_1(void* wParameter)
{
    sensor_data_t tData;
    while(1) {
        sensor1_get_data(&tData);                /*!<获取传感器 1 数据 */
        enqueue(&Queue,tData);
        rt_thread_delay(10);                  /*!<任务休眠 10 个 OS tick */
    }
}
/* Sensor_2 采样任务 */
void sensor_task_2(void* wParameter)
{
    sensor_data_t tData;
    while(1) {
```

```
        sensor2_get_data(&tData);                /*!<获取传感器 2 数据 */
        enqueue(&tQueue,tData);
        rt_thread_delay(10);                  /*!<任务休眠 10 个 OS tick */
    }
}

/* Sensor_n 采样任务 */
...

void rt_application_init()
{
    rt_thread_t thread;
    /* 创建 sensor_task_1 任务*/
    thread = rt_thread_create("Task1", sensor_task_1, NULL,512, 20, 5);
    if (thread != RT_NULL) {
        rt_thread_startup(thread);
    }
    /* 创建 sensor_task_2 任务*/
    thread = rt_thread_create("Task2", sensor_task_2, NULL,512, 10, 5);
    if (thread != RT_NULL) {
        rt_thread_startup(thread);
    }

    ...
    /* 创建 sensor_task_n 任务*/
    ...
}
```

示例中，任务 sensor_task_1 首先通过 sensor1_get_data 函数获取对应传感器数据到 tData，然后由 enqueue 函数将 tData 入队至传感器数据队列，入队完成后该任务休眠 10 个 OS tick 以实现固定的采样周期；共有 n 个这样的任务，并且它们的优先级不尽相同——各个任务处于不同的任务平面，示例中 sensor_task_2 优先级高于 sensor_task_1，因此存在如下可能性：

- 运行一段时间后，队列尾指针 hwTail 等于 hwSize-1；
- sensor_task_1 中的入队函数在执行“ptQueue->hwTail++”之后 hwTail 等于 hwSize，此时被 sensor_task_2 打断；
- sensor_task_2 又运行一次 enqueue 入队函数，hwTail 在执行“ptQueue->hwTail++”之后等于 hwSize+1，不符合复位条件“ptQueue->hwTail == ptQueue->hwSize”，hwTail 没有清零复位；
- sensor_task_2 执行完毕返回低优先级的 sensor_task_1，继续之前被打断了的操作——判断清零条件“ptQueue->hwTail == ptQueue->hwSize”不满足，hwTail 仍然不会复位清零；
- hwTail 的值已经大于 hwSize，即队列尾指针已经指向了队列缓冲区之外，而且永远不满足清零条件了。

一旦出现上面的情形，任何后续的入队操作都会造成内存入侵，对系统造成可谓灾难性后果——我们说数据的完整性被破坏了。

在多个任务平面间，由针对目标资源的非原子性操作发生交错所引发的读写冲突问题，我们称之为**数据完整性问题**。数据完整性问题的本质是操作的原子性问题，是由任务并发特性引起的。通过任务平面模型我们可以很清晰的观察到：

- 在裸机环境中，共享于主循环和中断处理程序之间的资源，实际上是充当前后台系统的二阶任务平面间的共享资源，中断任务平面上的任务可以打断主任务平面上针对资源的访问，存在读写交错的可能性；
- RT-Thread 操作系统环境下，基于优先级的抢占式调度引入了多阶任务平面集合，同优先级任务间基于时间片轮转的调度引入了多元任务平面集合——这些相比二阶任务平面看似引入了更加复杂的关系，实际上，数据完整性问题仍然是简单的操作原子性问题。

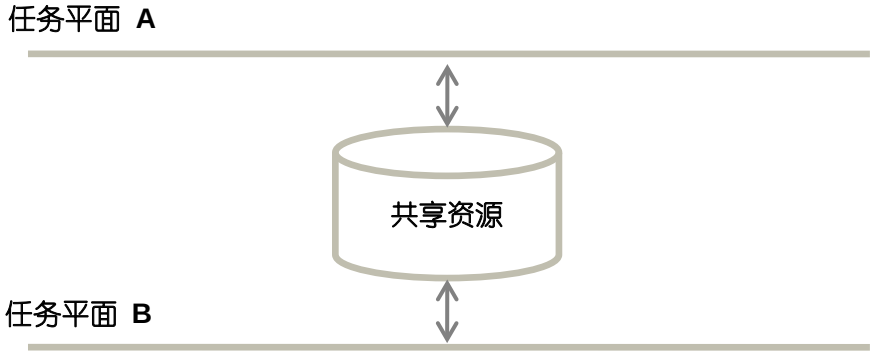
任务平面的引入可以让我们方便的观察到系统中哪些地方、针对哪些资源的操作存在原子性的问题。一旦准确的定位了它们，通过确保操作的原子性就可以轻松的解决数据完整性问题——具体的操作方式，由此引发的一系列具体问题，我们将在随后的内容中为您详细展开。

4.4 什么是共享资源

一般意义上的共享资源是指多个任务都会访问的资源，这里的资源包括一切可直接访问的寄存器、存储器。根据任务平面的定义，在同一个任务平面上的任务彼此不会打断对方，因而他们之间共同访问的资源也不会存在数据完整性的问题，我们可以把这类资源换一个更贴切的说法，例如“合作资源”，合作资源不在本文的讨论之列。共享资源是指不同任务平面上的任务都会访问的资源，因为不同任务平面间的任务存在彼此打断的风险，因而目标数据完整性得不到保证。这类资源，在本书中将统称为“共享资源”。

所谓共享资源，就是多个任务平面都会访问的资源，如**图 4.4.1** 所示。这里的共享不是任务间的共享，而是任务平面间的共享。

图 4.4.1 一个最简单的共享资源模型



在**图 4.4.1** 所示的共享资源模型中，任务平面 Task Space A 和 Task Space B 都会访问的资源就是共享资源。嵌入式系统中，共享资源是无处不在的，例如，裸机环境下，中断服务函数和主循环都会访问到的全局变量是共享资源；RT-Thread 操作系统环境下，高优先级的任务（高阶任务平面集中的任务）

和低优先级的任务（低阶任务平面集中的任务）都会访问到的资源，中断处理程序和用户任务都会访问的资源，都是共享资源；由于 RT-Thread 支持同优先级的任务时间片流转调度，即多元任务平面集，因此同优先级的任务也符合**图 4.4.1** 的模型，共享资源的定义仍然适用。

访问“合作资源”资源的任务在同一任务平面上，他们不会相互打断，不会出现数据完整性问题，因此这类资源的访问不需要特殊保护。例如，裸机环境下，一个全局变量被主循环中的两个函数（任务）访问，如果针对该变量的访问步骤不可分割，则不需要特殊的保护机制，因为主循环里调用的函数（任务）处于同一任务平面上。

共享资源是多任务系统中应用实现必不可少的关键，它主要有以下职责：

- 共享资源担负着不同任务间同步和通信的重任，例如中断和某一任务间的同步标志（Flag）；
- 共享资源用于保存和传递多任务的公共数据，例如各类数据缓冲区；
- 共享资源充当着某些任务的关键数据结构，例如各类跨任务的对象；

共享资源的访问存在风险，无法像合作资源那样可以不加保护的随意访问。那么如何规避共享资源的风险呢？共享资源的风险是由任务平面间的切换导致的数据完整性问题引起的，具体来说就是任务平面间彼此“打断”造成了读写操作交错，进而破坏了数据完整性。造成“打断”的原因很多，比如：

- 内核响应中断请求；
- 操作系统中，高优先级的任务被唤醒，抢占了正在运行的低优先级任务；
- 操作系统中，同优先级的任务时间片流转

无论是在裸机还是操作系统环境下，所有这些“打断”操作的本质都是上下文切换（详细内容请参考本书第一章的描述）。响应中断请求是上下文切换，操作系统调度也是上下文切换。显然，保护共享资源的方法只有一条：在访问共享资源时屏蔽上下文切换。

4.4.1 针对中断的上下文切换保护

关闭中断无疑是最简单的一种保护共享资源的策略。在裸机编程中，关闭全局中断后的系统只剩下主循环这一个单级任务平面——中断任务平面不再有能力打断主循环中运行的任务——这就达到了屏蔽上下文切换的目的，共享资源的访问得到了保护。

为了便于讨论，我们将使用平台无关的宏来表示开关全局中断响应的动作：

```
/*! 关闭全局中断响应 */
DISABLE_GLOBAL_INTERRUPT()

/*! 开启全局中断响应 */
ENABLE_GLOBAL_INTERRUPT()
```

回头看**图 4.3.2** 中，由于中断导致读写交错而产生共享资源数据完整性丢失的问题，使用关闭中断的策略可以保护共享资源访问步骤不被打断，进而保护了数据完整性。**图 4.3.2** 对应的例子代码直接保护代码如下：

```
static uint16_t s_hwADCValue = 0;
…
/*! AD 转换的中断处理程序 */
ISR(ADC_vect)
{
    s_hwADCValue = get_adc_result();        /*!<任务 A 获取 AD 采样结果 */
}
…
void main(void)
{
    …
    /*! 主程序中的大循环 */
    while (true) {
        …
        DISABLE_GLOBAL_INTERRUPT();          /*!<关闭全局中断，保护共享资源*/
        refresh_screen_task(s_hwADCValue);   /*!<任务 B 刷新显示器 */
        ENABLE_GLOBAL_INTERRUPT();               /*!<打开全局中断*/
        …
    }
}
```

在优先级低的主循环任务中，访问共享资源 s_hwADCValue 之前关闭全局中断，保证了对它的两个读取步骤中不会有写入操作来打断，共享资源的数据完整性得到了有效的保护。

像上面例子那样，访问共享资源前关闭全局中断，访问后直接打开全局中断的策略存在着一个比较严重的问题，即无论系统之前的全局中断开关是什么状态，只要访问完一个受保护的共享资源，全局中断就会被打开。显然，如果之前系统的全局中断是关闭的，那么全局中断被无条件打开就属于严重的误操作。因此，在共享资源访问完毕后，通常不是无条件打开全局中断，而是恢复至共享资源访问前的中断状态——之前的中断状态应该被记录下来。获取中断状态和恢复中断状态的一段平台无关的代码如下，用户可以根据自身平台自己实现具体的宏。

```
/*! 读取当前全局中断的开启状态，并保存在专门的 istate_t 类型变量中 */
istate_t tstate = GET_GLOBAL_INTERRUPT_STATE();

/*! 根据输入的 istate_t 类型变量设置全局中断开启状态 */
SET_GLOBAL_INTERRUPT_STATE(tstate);
```

有了上面的中断操作宏，在共享资源访问之前就可以记录中断状态，共享资源访问后，可以恢复之前的中断状态，即，之前全局中断是打开的，则此时打开全局中断，之前全局中断是关闭的，则此时关闭全局中断。一个典型的针对中断的上下文切换保护步骤归纳如下。

1. 获取当前中断状态，关闭全局中断
2. 访问共享资源
3. 恢复访问之前全局中断状态

既然关闭中断来屏蔽上下文切换的方法非常常用，而且需要固定的操作步骤，我们不妨将其固化下来，沉淀成一个实用的原子操作宏：SAFE_ATOM_CODE。

```
/*! 原子操作宏 */
# define SAFE_ATOM_CODE(...)    {\
        istate_t tState = GET_GLOBAL_INTERRUPT_STATE();\
        DISABLE_GLOBAL_INTERRUPT();\
        __VA_ARGS__;\
        SET_GLOBAL_INTERRUPT_STATE(tState);\
    }
```

这个宏中，依赖了三个宏函数和一个“istate_t”类型，这些宏函数和类型可根据具体的平台由用户自己定义。这里以 IAR 环境下尽量使用编译器自带的资源为例介绍一组定义：

```
#define GET_GLOBAL_INTERRUPT_STATE()        __get_interrupt_state()
#define SET_GLOBAL_INTERRUPT_STATE(__STATE) __set_interrupt_state(__STATE)
typedef __istate_t   istate_t;
```

有了这个宏函数，则针对中断的共享资源安全访问就变的非常简单，只需要将对共享资源的访问代码放在宏函数的参数中就可以了。示例如下：

```
SAFE_ATOM_CODE (
    shared_resources_access();                  /*<共享资源访问代码 */
);
```

有了这个原子操作宏，**图 4.3.2** 对应的 8 位裸机示例代码可以这样保护：

```
static uint16_t s_hwBuf = 0;

/*! AD 转换的中断处理程序 */
ISR(Timer0_OverFlow)
{
    s_hwBuf = ad_converse();                    /*!<任务 A 获取 AD 采样结果 */
}

void main(void)
{
    /*! 主循环 */
    while (true) {
        …
        SAFE_ATOM_CODE (                        /*!<受保护的共享资源操作*/
            refresh_screen_task(s_hwADCValue);  /*!<任务 B 刷新显示器 */
```

```
        );
        …
    }
}
```

关闭全局中断进行共享资源保护的方法简单实用，而且不会导致死锁问题，对于占用处理器时间资源较少的共享资源来说是最有效的保护方法。然而关闭中断会带来实时性能的降低的问题，关闭中断期间中断事件将得不到及时响应，详细的讨论和解决方法将在本书第六章详细讨论。

4.4.2 针对操作系统的上下文切换保护

在操作系统中，关闭全局中断后，低优先级任务不但不会被中断打断，而且也不会被高优先级任务打断，因为此时操作系统已经不能够响应任何触发任务调度的事件了，除非这个正在运行的任务主动放弃处理器控制权。在 RT-Thread 中，保护中断任务平面和其他任务平面间的共享资源可以通过直接开关中断来实现，RT-Thread 的 BSP 层提供了两个相关接口：

```
/*!关闭全局中断 */
rt_base_t rt_hw_interrupt_disable(void);

/*!恢复全局中断状态 */
void rt_hw_interrupt_enable(rt_base_t level);
```

这里恢复中断是指恢复到关闭中断前的状态，level 参数是 rt_hw_interrupt_disable 函数的返回值。在关闭了全局中断来访问共享资源期间，系统不再响应任何中断请求，但对共享资源的保护有时并不需要连同中断也屏蔽——中断任务平面外，操作系统不同优先级任务之间的共享资源保护，只需要屏蔽操作系统任务调度就可以。因此许多操作系统提供单独的屏蔽系统任务上下文切换方法，称之为“调度器锁”。RT-Thread 操作系统就有专门的调度器锁开关函数：

```
void rt_enter_critical(void); /*!打开调度锁*/

void rt_exit_critical(void); /*!关闭调度锁*/
```

在 RT-Thread 中，打开调度器锁以后，操作系统当前运行的任务将不会被换出，直到调度器解锁。但调度器上锁以后系统仍然能够响应中断请求，中断服务正常运行。使用调度器锁来保护 4.3 节中 Sensor-Hub 例子中的共享资源操作，应修改队列中对共享资源访问的代码，优先考虑数据完整性而暂不考虑实时性要求，修改后的安全代码如下：

```
/* Sensor_n 采样任务 */
void sensor_task_n(void* wParameter)
{
```

```
    sensor_data_t tData;
    while(1) {
        sensor_n_get_data(&tData);
        rt_enter_critical();                    /*! 打开调度锁*/
        enqueue(&Queue,tData);                  /*! 安全的共享资源访问*/
        rt_exit_critical();                     /*! 关闭调度锁*/
        rt_thread_delay(10);
    }
}
```

这样，入队操作的多个步骤之间不再会被其他任务打断，hwTail 指针不会错误越界而导致内存入侵，从而避免了因读写交错的数据完整性问题。

注意，RT-Thread 中 rt_enter_critical 和 rt_exit_critical 函数可嵌套，但必须成对出现。每调用一次 rt_enter_critical 则对应必须调用一次 rt_exit_critical 函数，否则调度器将被人为死锁。

对操作系统进行上下文切换保护，并不是说就不用考虑针对中断的上下文切换了。对于中断任务平面和普通任务平面间的共享资源，仍然需要关闭中断来进行保护，调度器锁只能对中断任务平面外的多阶、多元任务平面进行共享资源保护，两种保护方法是并存的，威力不同，应用场合不同，需要根据应用场合合理使用。

4.4.3 小结

屏蔽上下文切换，对于中断来说，就是关闭全局中断响应。在访问共享资源时，一旦关闭全局中断响应，那么对共享资源的操作就不会被中断处理程序打断，也就不会出现数据完整性遭到破坏的风险；对于操作系统来说，屏蔽上下文切换就是屏蔽操作系统的任务调度，即在某一个任务平面中访问共享资源时，操作系统不被允许进行任务切换，保护了共享资源的访问。对于某些平台来说，对应操作系统的任务调度也是经由中断实现的，因此屏蔽了全局中断也就一起屏蔽了操作系统的任务调度。

如何评估共享资源的代码安全?

实际应用中，一项工作往往会被分配到不同任务平面中协同工作，完成设计目标。无论在裸机还是操作系统环境下，多任务的安全问题都伴随共享资源而存在着。毋庸置疑，共享资源的安全几乎决定了整个多任务系统的安全。因此，针对共享资源的代码安全评估非常重要。

对于只有超级循环和中断系统的二阶任务平面系统(裸机),检查共享资源的隐患可归纳为以下步骤:首先，找出所有共享资源。由于超级循环中的任务处于同一任务平面，只要找出中断处理程序（ISR）所访问的资源，共享资源就是这些资源的一个子集；然后，分析这些资源在 ISR 之外的使用情况，在大循环的任务平面上也被访问的资源就是共享资源。其次，找出有风险的共享资源。有一类共享资源是天然安全的：每处访问都只有读操作，没有写操作，这样的共享资源不需要保护。最后，对于需要保护的共享资源，要在低阶任务平面中进行保护--在访问前禁止中断，访问后恢复原来的中断状态。只要没有采用这样的共享资源保护机制的代码，被认为是不安全的。下面的代码演示了处理方法：

```
/*! 中断服务程序 */
ISR(xxxx_vector)
```

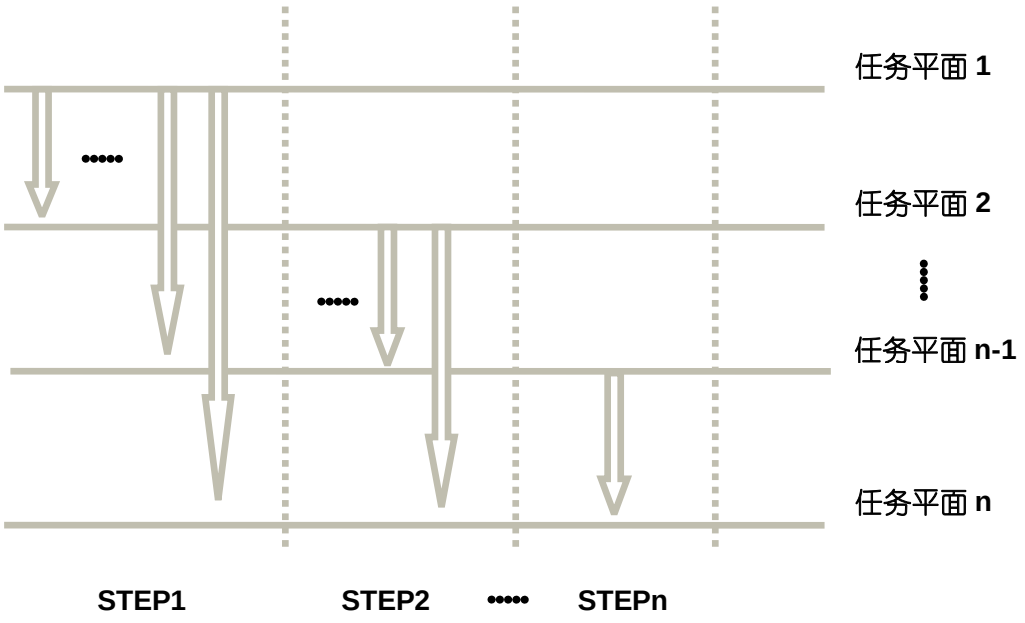
```
{
    shared_resource_access_1();/*!<共享资源访问 */
}
/*! main 中的超级循环 */
void main(void)
{
    while(1) {
        istate_t tState = GET_GLOBAL_INTERRUPT_STATE();
        DISABLE_GLOBAL_INTERRUPT();
        shared_resource_access_2();              /*!<受保护的共享资源访问 */
        SET_GLOBAL_INTERRUPT_STATE(tState);
    }
}
```

对于具有多阶任务平面的操作系统来说,共享资源安全评估步骤较多,但只运用与裸机同样的原则。首先，同样先找出共享资源：由最高阶任务开始，依次找出最高阶任务和所有比它低阶的任务间的共享资源，在所有低阶任务平面中的共享资源应予以保护。这样就完成了最高优先级任务相关的共享资源安全评估。然后，使用同样的方法找到次高阶任务平面向下所有共享资源，检查保护情况。最后，由高到低任务平面，依次重复这一过程，直到进行到最低优先级的任务，完成所有共享资源的安全评估，如**图 4.4.2** 所示。

对于同时具有多阶和多元任务平面的系统来说，找出共享资源的方法和纯多阶系统几乎一致，只是不仅需要找出高阶任务和比他低阶任务共同访问的资源，还需要找出与同阶其他任务平面都会访问的资源，认定为共享资源。

找出了代码中共享资源的保护情况，我们很容易判断出被评估代码是否是安全的。对于不安全的代码，要合理运用上文描述的共享资源保护机制，明白它们之间的异同。然而，通用的保护方法并不是万能的，对于中断和调度锁的特点分析和利弊，请参见后面章节。

图 4.4.2 在多阶任务平面环境下找出所有共享资源的步骤



4.5 小结

4.6 扩展阅读

什么是“实时性”

[Put Introduction Here]

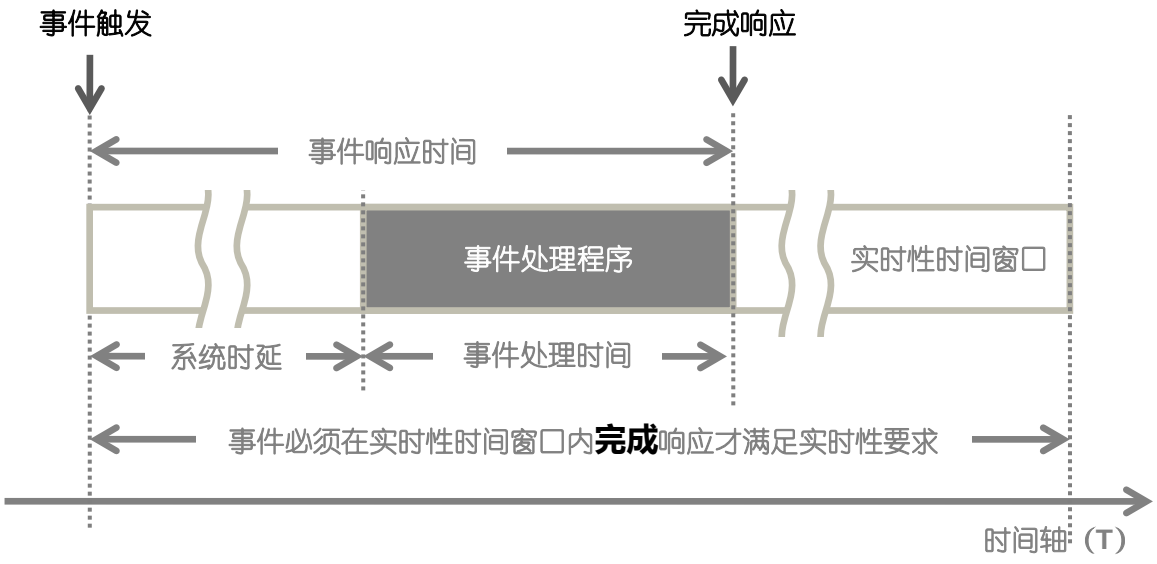
6.1 什么是实时性

和想象的不同，实时性（Real Time）并不是一个强调越快越好的概念。打比方说：团队老大在某个周五的例会上给你布置了一个 3 天工作量的任务，要求在下周五结束之前必须完成。这里，老大给出了任务的工作量（3 天）并提出了明确的时间窗口，即：从现在开始到下周五结束之前。在这个例子中，只要我们在时间窗口内，也就是“最迟下周五结束之前”完成任务，就是满足实时性要求的——无论你是接到任务后立即马不停蹄，能多快就多快，在周三完成任务，还是慢慢悠悠赶在周五最后时间之前完成任务——毫无疑问，都满足老大提出的实时性需求。

> 在给定的时间窗口内完成了确定工作量的任务，我们就可以说实时性得到了满足。

实时性的概念可以用**图 6.1.1** 清晰的加以展示。首先，我们注意到**实时性时间窗口**是由**事件触发**开始计算的一段时间来定义的，这个窗口是判断实时性是否得到满足的硬性指标。其次，我们注意到图中灰色的部分标注了事件处理程序处理该事件所需的时间，我们称之为**事件处理时间**。容易想到，从**实时性时间窗**口中扣去**事件处理时间**，得到的就是所谓的“**最迟响应时间**”——简而言之，到了这个时间点还不处理任务就真的来不及了。值得注意的是，图中系统的**事件响应时间**是指从**事件触发**开始到**完成响应**所需的实际时间。如果说**实时性时间窗口**是一种要求的话，那么**事件响应时间**就是程序实际对该要求的答卷。考虑到**系统时延**的存在，容易知道，**事件响应时间**最小值等于**事件处理时间**（**系统时延**为 0）；最大值就是**实时性时间窗口**（**系统时延**为“**最迟响应时间**”）。

图 6.1.1 事件触发、实时性要求与事件处理时间



6.1.1 中断能保证实时性么

在嵌入式软件领域，有相当一部分微控制器程序员存在这样一个误区，即，中断可以确保实时性。这种观点来源于程序员在日常系统开发中对运行结果最直接的观察，是一种直觉上的判断，然而从实时性定义的角度出发，这种观点并不妥当。

中断确保的是“即时性”，这是一个伪概念。确保即时性并不一定能确保“实时性”。

即时性，意思是越快越好，中断响应确保的就是这一点——当中断被触发时，系统会极尽所能尽快加以响应。然而，这里的极尽所能实际上已经暗示了存在导致中断事件响应延迟的因素，例如：

- 全局中断响应被屏蔽；
- 存在更高优先级的中断；
- 当前的中断优先级被屏蔽；
- 存在操作系统，且操作系统劫持了中断向量；
- 芯片之前处于深度休眠状态，由于时钟系统恢复工作需要从微秒级别到毫秒级别不等的时间，导致内核无法及时响应中断；
- ……

即时性是一种“追求”而不是一种保证，这种“越快越好”的概念实际上建立的是一种“根据当时情况做出最佳选择”的操作准则——即，系统要尽力去做到最快，但究竟能做到多快是没有保证的。即时性和实时性追求的是完全不同的东西，即时性强调的是一种相对当时具体环境条件的“努力”，而实时性强调的则是无关条件的强制的“硬性指标”——从工程的角度来说，“硬性指标”要比“努力”更容易衡量和控制。

实时性与即时性其实并不矛盾，就前面的例子来说，团队老大在周五的会议中规定了任务的实时性窗口——下周五结束之前完成——但并未规定你要做出何种的“努力”，换句话说，老大并没有对即时性做出要求；如果老大要求你必须尽快完成任务且最迟不超过下周五结束，那么这就是同时对实时性和即时性做出了要求。

6.1.2 操作系统能确保实时性么

通过前面的讲解，其实我们已经很清楚结论了。操作系统只是多任务的一种实现方式，是对处理器时间的自动分配。操作系统实现了任务的自动拆分，简化了程序员开发的难度。从本质上来说，很难找到实时性的保证与操作系统原理的直接关联。

操作系统的存在是保证系统实时性的充分但非必要条件。

操作系统本身并不能保证系统的实时性，但提供了一系列多任务环境下所必须的通用工具和手段，在有效使用的情况下，可以被用来确保系统的实时性。具体来说，操作系统提供了多任务同步和通信的完整解决方案，如信号量、互斥、临界区保护、邮箱和消息队列等等，正确使用这些工具进行多任务设计，是保证系统实时性的前提条件。

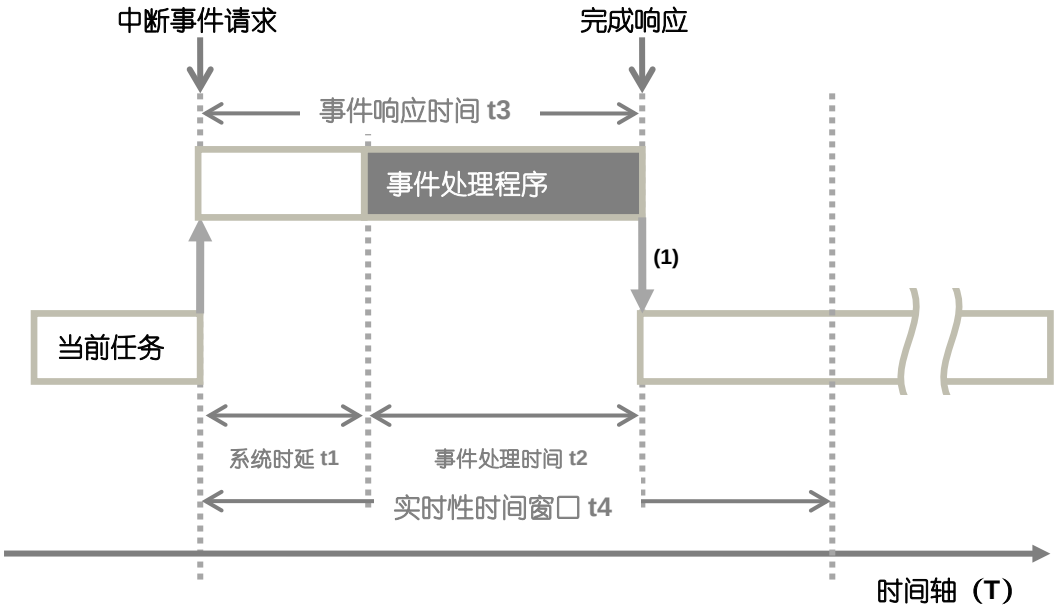
在裸机环境下，多任务是依靠手动调度实现的。手动调度同样不能保证系统的实时性。相对操作系统环境下完善的多任务同步和通信机制，裸机系统甚至需要用户自己去实现这样一套机制。所幸多任务同步和通信的原理是不变的，与多任务的实现方式无关，本书所讨论的多任务同步和通信的内容不仅对 RT-Thread 这样的操作系统适用，对裸机环境也具有同样的参考意义。

6.2 细思极恐”的通杀策略

使用“中断锁”和“调度器锁”保护共享资源是威力最强、应用场合最广的两种策略，但两种策略均没有指向性和针对性，使用不当非常容易破坏系统的实时性。

大部分使用“中断锁”的共享资源保护策略通过屏蔽全局中断来实现。关闭中断进行共享资源访问期间，系统将不再响应任何中断请求，也就不能响应任何外部事件。而这时如果有事件触发，系统对该事件的响应会丢失，或者被延迟到共享资源访问结束——重新恢复中断的时刻。少数处理器的架构支持屏蔽指定优先级的中断请求，例如，屏蔽某一中间优先级的中断后，低于此优先级的中断请求全部被屏蔽，只有高于此优先级的中断能够被响应。这种处理器屏蔽中断造成的效果和普通处理器殊途同归，因此不再展开讨论。

图 6.2.1 正常中断事件响应和实时性要求



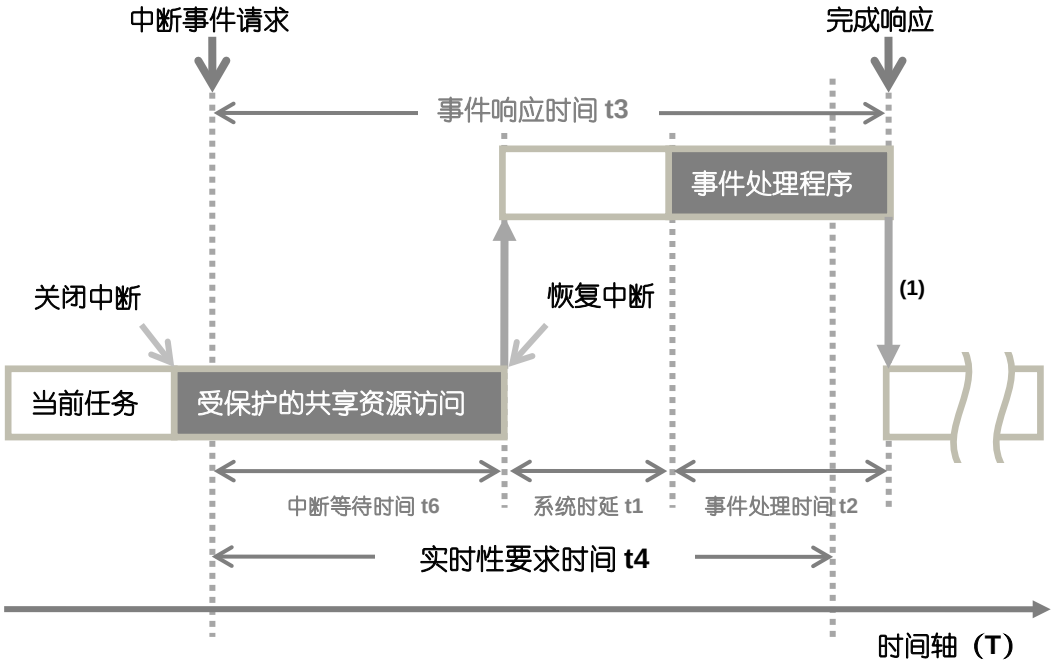
注：1、 操作系统退出 ISR 时并不一定切换回原任务，而是立即进行一次任务调度，优先级最高的任务会被运行，例如 RT-Thread。

没有屏蔽中断的正常事件响应如**图 6.2.1** 所示。中断事件请求后被立即触发，经过系统时延 t1 后进入 ISR 事件处理程序，ISR 程序完成标志着事件完成响应，从事件触发到完成响应的总时间 t3（称为事件响应时间）等于 t1 加上 t2，t3 小于系统的实时性设计时间要求 t4，则满足了实时性系统设计。

使用了屏蔽中断策略的一种情况如**图 6.2.2** 所示。在共享资源访问期间，由于屏蔽了全局中断，事件触发请求发生后，系统无法及时响应——直到共享资源访问结束恢复全局中断。事件响应时间 t3 等于中断等待时间 t6 加系统时延 t1 再加事件处理时间 t2，只要共享资源访问导致的中断等待时间 t6 大于实时性时间窗口 t4，那么系统对这次事件的响应就不满足实时性了。

如第三章描述，实时系统是指在固定的时间内（deadline）能正确的对外部事件做出响应的系统。如果超过了实时性要求时间而系统没有完成事件响应，则称该系统不符合实时性设计要求。一台自平衡无人机，经常见到一种裸机编程方法是将传感器事件处理程序设计在中断任务平面（ISR），而功能性程序被放在主任务平面（主循环）中。假设传感器数据必须在 5 毫秒内处理完毕，否则可能发生坠机。为了保证代码健壮性，如果在主程序中对共享资源使用中断锁保护，就可能出现如下情况：假设处理器响应中断的固定时延 t1 为 0.1ms，中断处理程序用时 t2 为 3ms，而如果共享资源被保护时间大于 1.9ms，事件响应时间 t3 就有可能大于 5 毫秒——超过实时性设计要求。只要系统中存在着不满足实时性设计要求的可能性，系统设计就是不合格的，因为我们永远无法预测最坏的情况什么时候发生。

图 6.2.2 使用中断锁保护共享资源导致实时性问题



注：1、 操作系统退出 ISR 时并不一定切换回原任务，而是根据情况立即进行一次任务调度，例如 RT-Thread 操作系统环境下，在 ISR 中调用了会触发任务调度的系统资源（释放信号量等）的情况。

总结“中断锁”策略的利弊，不难发现其优势：

- 最强大有效的共享资源保护策略，一旦屏蔽中断，ISR 任务平面和其他依赖中断的任务调度都无法打断对共享资源的操作，共享资源是最安全的；
- 最高效的共享资源保护策略，所需指令条数最少。

然而，“中断锁”的弊端也显而易见：

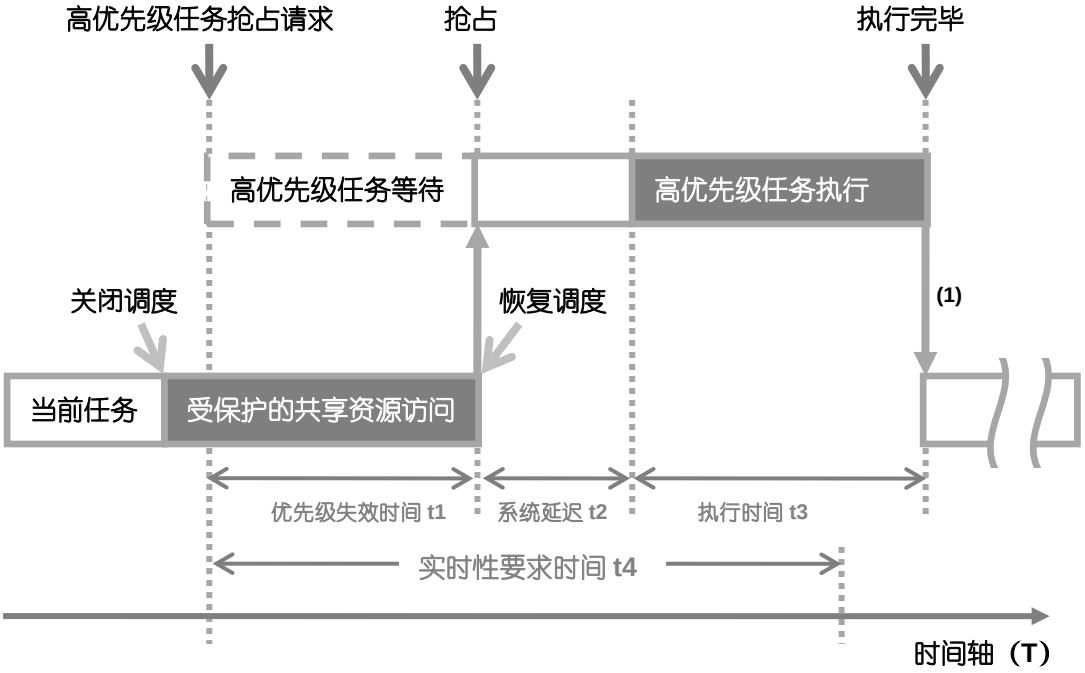
- “一刀切”的保护，为了保护某一个共享资源，影响其他中断响应
- 使用不当容易造成中断事件丢失，或中断事件响应时间延长，实时性能下降
- 对于依赖中断进行任务调度的操作系统来说，屏蔽中断同时误屏蔽了任务调度

“调度器锁”的力度不如“中断锁”强，它只能够屏蔽系统不同任务平面间的任务调度，但仍然会有“中断锁”类似的问题。RT-Thread 的调度器是基于优先级的可抢占任务调度器，开启“调度器锁”后，高阶任务平面的任务将不能抢占当前正在执行的低优先级任务，直到关闭“调度器锁”——在使用“调度器锁”保护共享资源访问期间，发生了优先级倒置，如**图 6.2.3** 所示。优先级倒置的结果是系统的实时性可能得不到满足。“调度器锁”的优点：

- 较强大的共享资源保护策略，开启调度器锁后不同任务平面任务不会相互打断（包括多阶、多元任务平面），对于不牵涉 ISR 任务平面的共享资源是安全的保护措施；
- 量级比“中断锁”轻，不会对系统中断响应造成负担，系统仍然能够响应中断请求。
- “调度器锁”的缺点也很明显：
- 不能保护 ISR 任务平面与其他任务平面间的共享资源
- 使用不当容易造成高阶任务平面无法抢占，任务优先级临时失效甚至人为死锁，实时性能下降

“中断锁”和“调度器锁”的实质都是使用处理器硬件支持的方式屏蔽上下文切换，其作用都是要创建一段不被打断的代码区域，保护不同任务平面间共享资源的访问不出现数据完整性问题。它们共同的优点是对自身适用范围内的共享资源保护是可靠的，但也都会导致实时性能下降，引入不确定性。究竟如何在保护共享资源的同时保证实时性要求呢？在 6.3 节中我们来详细讨论这个问题。

图 6.2.3 使用调度器锁保护共享资源导致临时优先级失效



(1) 操作系统退出某优先级任务时并不一定切换回原任务，而是根据情况立即进行一次任务调度，优先级最高的任务将会被调用。

6.3 挽救“实时性”

（您所阅读的仅仅是样章，并不包含所有内容，展现的内容也并不是最终形式，详细内容请参考正式书籍）

6.4 “以小换大”的通用优化策略

（您所阅读的仅仅是样章，并不包含所有内容，展现的内容也并不是最终形式，详细内容请参考正式书籍）

6.5 如何确保“实时性”

实时性是工程应用对嵌入式系统设计和实现所提出的硬性指标，无论是越快越好的“中断”、提供抢占式服务的“操作系统”，还是一切尽在掌握的“手动调度”，其本身都不能确保系统的实时性要求得到满足。“中断确保实时性”、“操作系统确保实时性”甚至是“状态机确保实时性”都是一种常见的认识误区。那么如何才能确保实时性呢？

首先思考单任务的环境，实时性的确保只与系统本身的特性有关，例如，硬件是否能及时响应，处理器的性能是否足够等等。一般来说，只要硬件系统设计得当，单个任务的实时性是很容易得到满足的。接下来思考多任务环境，确保实时性仍然与系统的性能有关，即理论上硬件的性能必须同时满足给定数量的多个任务对实时性的要求——巧妇难为无米之炊，实际应用中，系统往往要在性能上为突发情况留下足够的富余，才能使系统的实时性需求得到保证。这里，预留富余的多少通常与嵌入式系统的成本（包含方案成本，人力成本和时间成本）直接相关。如何精准的评估应用对硬件性能的需求，尽可能将这种“富余”降到最低，是考验嵌入式系统设计能力的试金石。

当系统的硬件性能得到保证时，实时性的保证就与多任务设计方式直接相关了。多任务系统中，最核心的关系是多任务之间的协调关系，最核心的矛盾是多任务之间对资源的竞争。简单的说多任务设计最头疼的就是处理任务与任务间相互抢资源的问题。很多时候，单任务环境下能够满足的实时性要求，多个任务“搅和在一起”对有限的资源“一阵哄抢”“乱成一锅粥”就很难兑现了。研究多任务的同步和通信问题，本质上就是研究如何让任务彼此“好好的合作”缓解“竞争矛盾”力求最大限度利用有限的资源。多任务的程序设计就是多任务间同步和通信关系的设计。

系统实时性的保证是依靠合理的多任务程序设计来实现的。

8

“完美逻辑”

[Put Introduction Here]

8.1 “生产者”与“消费者”模型

“生产者（Producer）”和“消费者（Consumer）”问题是多任务系统的经典命题，理解起来也并不复杂：

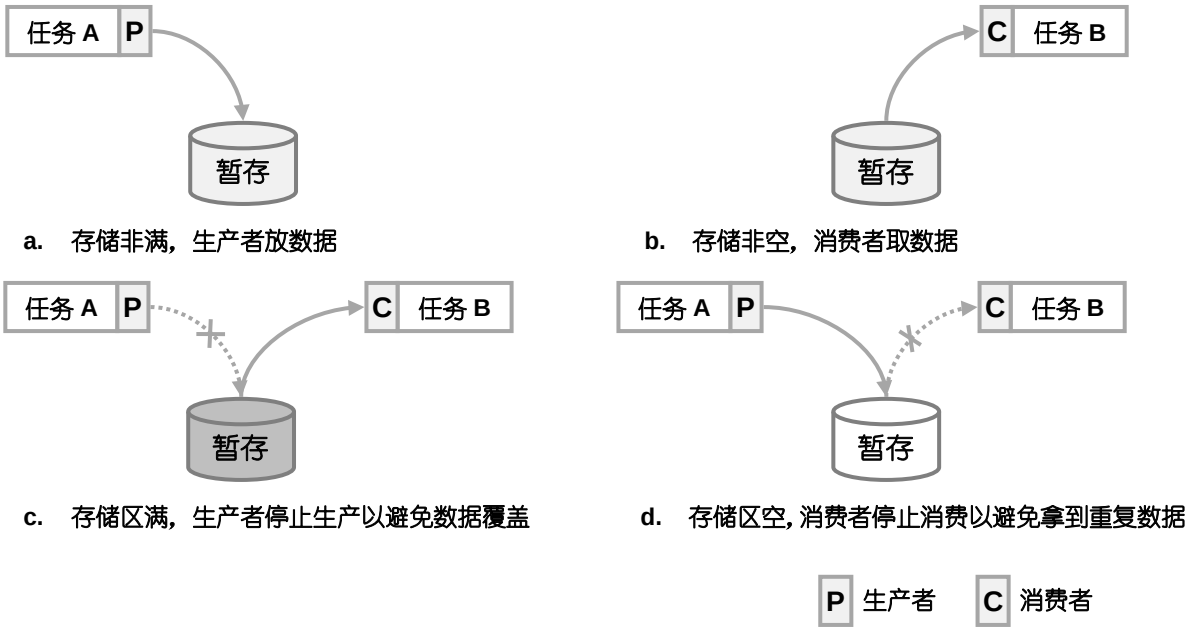
> 生产者就是内容的产出者；消费者就是内容的使用者（消耗者）。

简单来说，“生产者”就是多任务系统中，从事“内容”生产的任务。这里的“内容”包罗万象——可以是从 USART 设备输入的字节流，可以是算法的运算结果，甚至可以是抽象的事件或消息等等。“消费者”是某一“内容”的使用者，该任务既可以直接将“内容”消费掉，例如输出到终端；也可以对数据进行“再加工”，进而传递给下一级消费者（需要注意的是，此时该任务即是上一级“生产者”的“消费者”，也是下一级“消费者”的“生产者”）。

基本的“生产者”与“消费者”问题描述了如下情景（如**图 8.1** 所示）：有一个或者多个任务生产某种类型的数据，放置在特定的暂存区域中；有一个或者多个任务从该暂存区域中获取数据，进行消费；暂存区是临界资源，同一时刻只有一个任务允许访问。简单来说，“生产者”和“消费者”必须遵循以下三个规则才能有序运作：

- 生产者负责持续生产，直到存储区满；
- 消费者负责持续消费，直到存储区空；
- 存储区同一时刻只能被一个任务访问；

图 8.1 经典“生产者”与“消费者”模型



- 早餐店的故事

故事一：一家刚开业的早餐店雇佣了小金，由于只有一个人，他既要负责在厨房包包子，又要负责在门面前照顾客人，因此当小金照顾客人的时候就无法做包子、做包子的时候就无法招呼客人。早餐店刚开张的时候客流较小，小金还算忙得过来，但不久以后，由于小金手艺很好，回头客很多，每天高峰时段早餐点门口都会拍起长队，一些等了太久的顾客最后只好放弃，就这样渐渐

漏掉了不少生意。

故事二：看到生意好，早餐店增加了一个售卖员小梁，这样小金就可以专注于做包子，而小梁只负责售卖。情况果然好转了，队伍的长度也大大缩短。由于生意太好，小金不得不半夜就起床提前准备好包子，以应对高峰时段人们包子的需求。然而，好景不长，冬天到了，由于缺乏保温设备，包子做好以后如果不及时卖掉就会很快变冷影响口感，早来的客人常常抱怨小金早早做好得包子已经冷了，迟来的客人如果错过了新出笼包子的时间又需要等很久。看到队伍排得老长，小梁抱怨小金做包子偷懒，他告诉老板，很多客人因为包子冷了不好吃或者干脆等不到包子渐渐不来了。小金觉得很冤枉，于是决定宁可少卖一点，也要保证每个客人都能吃到热腾腾的包子。就这样，早餐店的效益似乎遇到了瓶颈，一直无法再提高了。

故事三：为了解决这一问题，早餐店花重金购置了一个保温柜，这样小金就可以开足马做好包子，新出的包子只要柜子没满就直接放在柜子里，完全不用和外部进行沟通,；小梁售卖时也不必催促小金，只要柜子没空就直接在柜子里拿。自此以后，排队的现象渐渐消失了，人人都能吃到热腾腾的包子，早餐店的生意红红火火。

“生产者/消费者”模型的提出使得我们有能力在任务间纷繁复杂的关系中单独将数据的流动关系提取出来，方便进一步的研究和讨论。

例如，从上面早餐店的实际情况来看：“故事一”中小金同时兼任了“生产者”和“消费者”两个角色，任务是串行的——效率不高。这也只有在早期生意清淡时勉强应付，一旦顾客需求加大，任务“串行”的弊端就会立即凸现出来。“故事二”是对“故事一”任务的拆分—— “生产者”和“消费者”的角色被独立了出来，使得二者可以并行工作，提高了早餐店的运作效率。实际应用中，“消费者”的消费能力和“生产者”的生产能力不仅存在差异，而且这种差异会随着时间的变化而不同，例如故事二中，包子的销售能力在高峰期和非高峰期是存在明显差异的；而在引入“让所有人都吃到热腾腾的包子”这一限定条件下，在高峰期小金全力生产也不能及时满足销售的需求，而非高峰期（比如半夜），小金必须控制包子的制作速度，避免由于顾客稀少导致出笼的包子变凉。“故事三”正是针对这一点，对“故事二”进行了改进，引入了缓冲区（保温箱）的概念——即确保了全力生产，又减少了消费者的平均等待时间，进一步提高了效率、并而外的降低了耦合度（包子的生产和包子的销售互不相关，单纯以保温柜作为衔接）。

其实针对“生产者/消费者”问题进行讨论，无非是从三个方面进行展开：

- 为什么要拆分出“生产者”和“消费者”到不同任务？什么情况下需要拆分？
- 拆分出“生产者”和“消费者”后会出现哪些问题？如何解决？
- 解决问题的方法有哪些？要借助什么工具？工具是否有缺陷？

为了把抽象的问题讲清楚，抛开生活中的例子，我们将借助一个看似简单的程序案例，对 “生产者”与“消费者”的模型继续挖掘，并对由此衍生的两种经典范式进行分析，而掌握这些范式对于洞悉任务间数据流动、设计出高效可靠的多任务系统是必不可少的。

8.2 “完美逻辑”

我们考虑这样一种经过简化的需求：某软件模块需要通过串口接收数据（波特率规定不高于 9600），当检测到数据流中存在字符串“apple”时立即输出字符串“Apple is red!\r\n”作为响应。考虑更一般的情况：该设备使用异步串行接口（UART）连接电脑，外设 UART 无硬件缓冲，也不使用中断来进行数

据接收；应用通过 UART 驱动接口 get_char()以轮询的方式依次获取字节，一旦检测到数据流中存在字符串“apple”，立即回发“Apple is red!\r\n”作为响应。例如：我们通过 PC 端的超级终端连接设备串口，一旦输入“apple”，超级终端就会立即显示出设备回复的字符串“Apple is red!\r\n”。

需要特别特别强调的是，这里我们假设“字符输入设备无硬件缓冲”且“软件在接收中不使用中断”，这是为了方便讨论，分别从硬件和软件两个方面对模型进行了简化，随后在理解了问题的本质以后我们会发现，“加入硬件缓冲”或者“使用中断方式来接收数据”仅仅只能起到有限制的缓解作用，本质问题仍然存在。

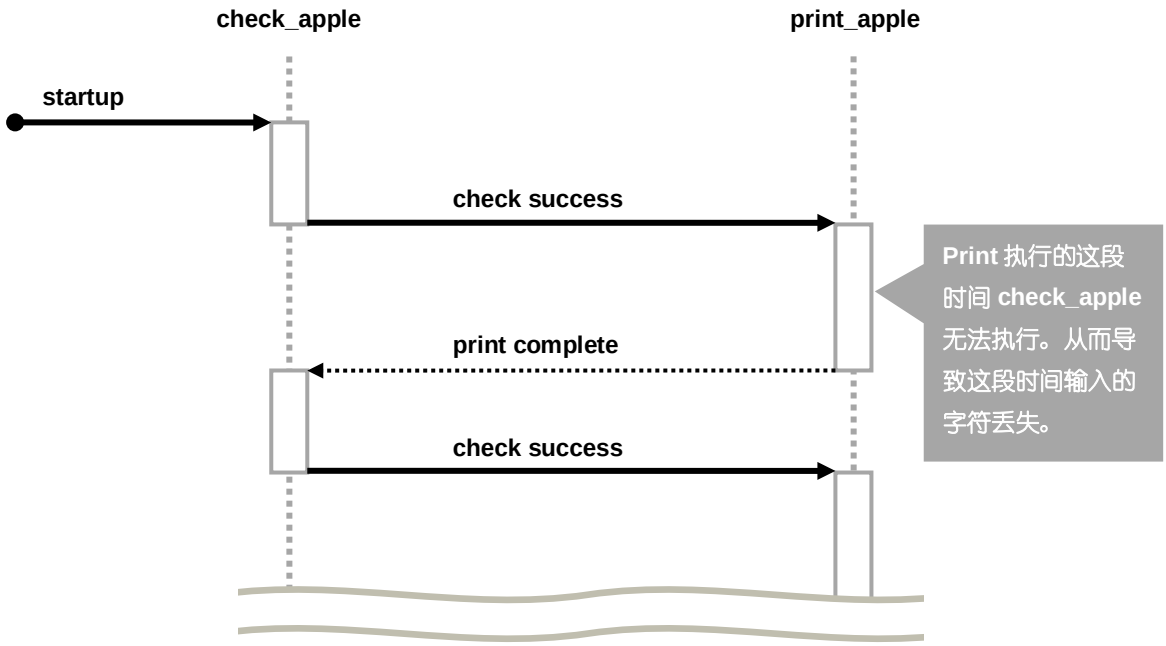
针对上述的需求，我们很容易得到如下的代码：

```
//! 以阻塞的方式从 UART 读取一个字符
extern char get_char(void);
//! 以阻塞的方式通过 UART 输出一个字符
extern void put_char(void);

//! 字符串检测函数
static void check_apple(void)
{
    static const char c_String[] = {“apple”};
    const char *pchStr = c_String;

    //! 一个一个字符的检测，直到完整的字符串被识别出来
    do {
        char cLetter = get_char();       //!< 从 UART 阻塞的读取一个字符
        if (cLetter != *pchStr) {
            pchStr = c_String;           //!< 识别失败，从头重新开始
        } else {
            pchStr++;
        }
    } while(‘\0’ != *pchStr);
    //! 字符串被识别了出来
}

//! 字符串输出函数
static void print_apple(void)
{
    const char *pchStr = “Apple is red!\r\n”;
    do {
        put_char(*pchStr++);             //!< 从 UART 阻塞地输出一个字符
    } while(‘\0’ != *pchStr);
}

// 任务入口
static void check_and_print_entry(void* parameter)
```

```
{
    while(1) {
        check_apple();          // 从UART0上查询输入字符，循环检测“apple”，检测成功后返回
        print_apple();          // 输出“Apple is red!”，完成输出后返回
    }
}
```

这段代码中，检测字符串函数“check_apple”从 UART0 上读取用户通过超级终端输入的数据，并循环检测是否匹配“apple”，一旦匹配立即返回，执行“print_apple”，在 UART0 上输出“Apple is red!\r\n”，字符串最终将显示在超级终端上。需求很简单，代码逻辑也不复杂，对应的序列图如下：

图 8.2 check_apple 和 print_apple 运行序列图



从**图 8.2** 可以清楚的看到，当“check_apple”检测到“apple”后，“print_apple”将会按顺序得到运行，而在“print_apple”运行期间，“check_apple”的任务是无法执行的，换句话说，在这段时间内由于缓冲区的缺失，输入的数据都会被丢失——当用户输入速度较慢时尚能应付，一旦数据吞吐较大，比如使用超级终端直接发送文件，漏数据的问题就会暴露无疑，这就是串行执行的致命缺点，和本章开始第一个故事所描述的情形是一致的。

说“明明有硬件缓冲为什么不用”和“明明可以用中断来实现软件缓冲”的同学请先留步，如果你没法按照我们先前的约定来思考，我觉得有必要就事论事的解释一下 1）为什么我们要假设硬件缓冲并不存在，或者说禁用硬件缓冲；2）为什么我们在这里要抛开中断接受方式进行讨论。

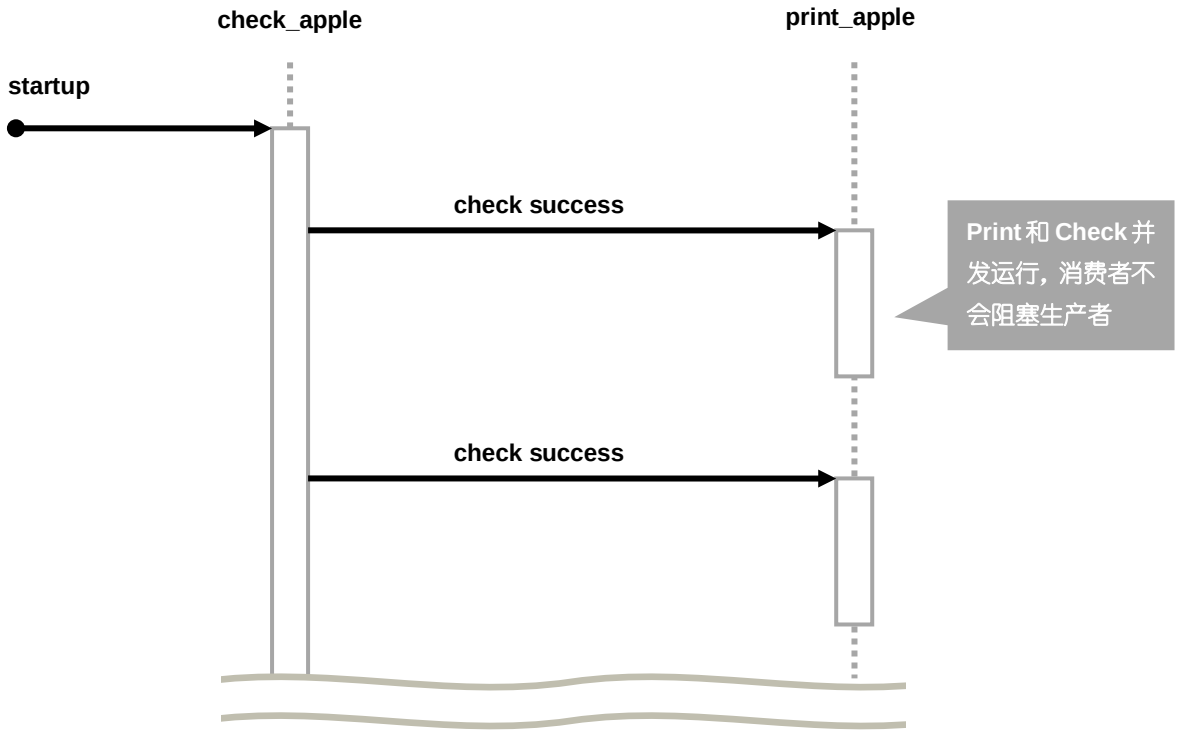
针对第一个问题，首先，很多芯片的串行外设并不一定包含硬件缓冲，早期流行的 STM32 系列就是一个很好的范例，为了保证讨论的一般性，我们应该尽可能选取最差情况。其次，缓冲无论“软硬”其本质和作用是相同的，我们在本章随后的内容中将会详细讲解，您不妨耐着性子，等拿到“口实”再来有理有据的批判笔者也不迟。或许看完后续的章节，您也就会站到我这一边来。实在不行，您大可保留意见，不必将我们一砖头拍死。

针对第二个问题，在前面的章节里我们已经通过“多阶任务平面集合”的模型详细论述过：中断相

对操作系统用户任务只是较高阶的任务平面，本质上与一个在比当前任务（所在平面）阶值更高（平面上的）的任务并无区别。因而，这里我们就要考虑，如何安排任务优先级的高低？为什么要这么安排？随意安排高优先级任务对整个系统（二不仅仅是局部任务的）实时性有哪些影响？考虑这些问题实际上是将任务优先级（或者说任务平面的阶）作为变量引入到讨论中来。这些我们在本书随后的章节中也会逐一讨论，而在此之前，为了方便研究，我们暂时先将这一因素从模型中抹去，单纯从来讨论应用设计中的数据供求关系。

简化模型，仅保留我们关心的变量，是工程学中有针对性研究模型某一类现象的常用方法，作为工程学延伸的嵌入式系统研究也不例外。

图 8.3 拆分 check_apple 和 print_apple 到不同的任务



“check_apple”是生产者，“print_apple”是消费者，当前代码设计的失败之处在于：**串行的代码结构使得“实时性要求较低”的“消费者”干扰了“实时性要求较高”的“生产者”。**皮之不存毛将焉附？更何况消费者的字符串输出任务是一个并不紧急的事情，“print_apple”不仅没有实时性上的要求，甚至其功能实现所依赖的可能就是一个“相对低速”的设备；相对来说，生产者“check_apple”的就存在硬性的实时性要求——它必须在下一个字符到来前完成对当前字符的处理，因而，抛开输入输出设备速度的问题不谈，从更一般的角度来看，一个无实时性要求的任务与存在实时性要求的任务串行在一起，这种设计本身就是不合理的。

意识到问题的本质，解决方法就非常明确了。对于这个例子，由于生产和消费行为不能并发执行，导致消费者阻塞了生产者，容易想到将“生产者”和“消费者”拆分到不同的任务中（如**图 8.3** 所示），利用任务的并发特性——即便“print_apple”仍然需要很长的时间来打印字符串，只要“check_apple”在这段时间内有机会被执行——避免了字符被漏检的情况。

要做到这一点，通常有两种方法：

1）让“check_apple”所在任务优先级高于“print_apple”，并可根据源自 UART0 的事件来触发“check_apple”任务的执行；需要注意的是，“check_apple”需要“在无法获取字符时阻塞自己、在

获取字符时唤醒任务”——这通常由 get_char()函数自动实现——否则可能永远也轮不到“print_apple”来执行。

2）当“check_apple”与“print_apple”优先级相同时，除了 1）所要求的除优先级以外的内容外，还需要“print_apple”所依赖的底层输出函数本身有能力在外设 busy 时阻塞任务、并能在输出完成后重新唤醒任务——这通常由 put_char()函数自动实现。单纯将每个任务的时间片设置为最小值“1ms”并不能达到预期的效果——因为按照时间来调度的“粒度”相对大数据通信时的数据间隔来说仍然太大。因此，只有使用“在 busy 时主动放弃处理器、在设备 idle 时自动唤醒任务”的 put_char()函数才能根本性的解决问题。

3）保持“check_apple”和“print_apple”处于同一线程中，以 FSM 的形式在该线程中建立两个“亚”任务。此时，“check_apple”和“print_apple”都必须依赖非阻塞的外设输入输出函数，同时以状态机的形式重新描述其内部的逻辑，以满足 FSM 非阻塞的结构要求。详情请参考第二章相关章节的内容。

其中，方法 1）和方法 2）对函数 get_char()和 put_char()都有特殊要求，而实现这种特殊要求都免不了要引入 UART0 的中断，配合 semaphore 进行辅助（发送完成中断通过 semaphore 唤醒阻塞在 put_char 中的任务；接收完成中断通过 semaphore 唤醒阻塞在 get_char 中的任务）。为了简化过程，专注于我们要讨论的问题本身，这里有意将波特率限制在 9600 之内，以确保 RT-Thread 的时间“粒度（1ms）”比连续数据发送时的时间间隔（略大于 1ms）要小。这里特此说明，让我们重新回到讨论中来。

“check”任务和“print”任务拆分成两个以后，根据前一章的内容，我们很容易想到使用信号量（semaphore）由“check”任务来触发“print”任务。需要注意的是，此时，“check”和“print”作为“生产者”和“消费者”生产和消费的产品是“事件”。改进后的代码如下：

```
//! 字符输入及检测“apple”任务入口
static void check(void* parameter)
{
    while (1){
        check_apple();                                  //!< 查询输入字符，并检测“apple”
        rt_sem_release(&sem_trigger);                   //!< 发布一个 check 成功信号
    }
}

//! 输出字符串任务入口
static void print(void* parameter)
{
    while (1){
        rt_sem_take(&sem_trigger, RT_WAITING_FOREVER);  //!< 获取一个 check 成功信号
        print_apple();                                  //!< 输出“Apple is red!\r\n”
    }
}
```

假设 RT-Thread 的事件片调度最小单位是 1ms；“check”和“print”优先级相同，且都时间片（Interval）都设置为最小单位 1；当 UART0 波特率为 9600 的情况下，RT-Thread 调度的时间间隔已经小于通讯字节的时间间隔，我们是否可以认为每一个“apple”的输入都会得到一个对应的“Apple is red!”的输出呢？答案是否定的。实际上，即便假设 check_apple()是百分百可靠的，只要存在 3 个以上连续的“apple”

一定会在输出数量上存在不匹配——简单来说，三个连续的“apple”可能只能看到两个“Apple is red!\r\n”。这又是为什么呢？问题比你想象的复杂。

在讨论这一问题之前，我们首先要解决一个更本质的问题，即：什么情况下我们才需要将一个普通的串行数据加工拆分成“生产者”和“消费者”两个任务——使得它们并发起来呢？

当一个串行的数据加工中存在前后的两个工序有明显不同的实时性要求时，我们就应该将它们分开——分别作为“生产者”和“消费者”拆分到不同的任务中来。这只是解决问题的第一步。

理解这段话，需要注意以下几点：

- 首先，数据加工中（Process）可以进行拆分的两个“工序”必须是串行的，或者说他们存在先后顺序，且后者对前者有依赖关系。
- 其次，作为拆分候选的两个工序，必须拥有不同的实时性要求，或者二者拥有不同的数据吞吐能力。
- 再次，这里提到的数据加工（Process）并不一定只包含两个“工序”，它可以是一串或者多串呈现不同拓扑结构的工序。我们可以从中找到符合要求的部分，进而对整个数据加工进行拆分。比如，一个简单的 5 工序串行数据处理，其中第二和第三工序符合拆分条件，则当我们将它们分开时，整个数据加工也一同被拆分成两个部分：工序 1、2 组成前一部分；工序 3、4、5 作为一个整体组成后一个部分。

满足了这些要求，我们就有理由或者说必须要将一个数据加工中原本串行的两个工序拆分开来，以独立的“生产者”和“消费者”身份存在。我们也可以简单的说：

“生产者”与“消费者”的拆分首先是为了解决由“耦合”带来的实时性问题。

8.2.1 问题比你想象的复杂

“拆分”从本质上解除了限制“生产者/消费者”实时性的前提，但距离问题的解决仍然有相当长的路要走。在前一小结中，我们提到，即便作为“生产者”的 check 任务百分百可靠——不会错过任何一个有效的字符串（“apple”）输入——但理论上反映“消费者”print 消费行为的字符串输出（“Apple is red!\r\n”），在数量上仍然与“apple”存在差异——现象上，仿佛有一些“apple”丢失了。

究竟是谁偷走了我们的“苹果”？为什么我们可以非常确定的从理论角度就确认这种丢失行为一定是存在的呢？

TBD（对上述案例进行分析，说明在 UART 输入输出速度相同的客观前提下，输入输出字符数目不同导致实际上 check 和 print 的吞吐量存在差异）

是“速度差”——是“生产者”的生产能力与“消费者”的消费能力之间所存在的“速度差”——偷走了我们的苹果。实际上，这里存在至少四种情况需要讨论：

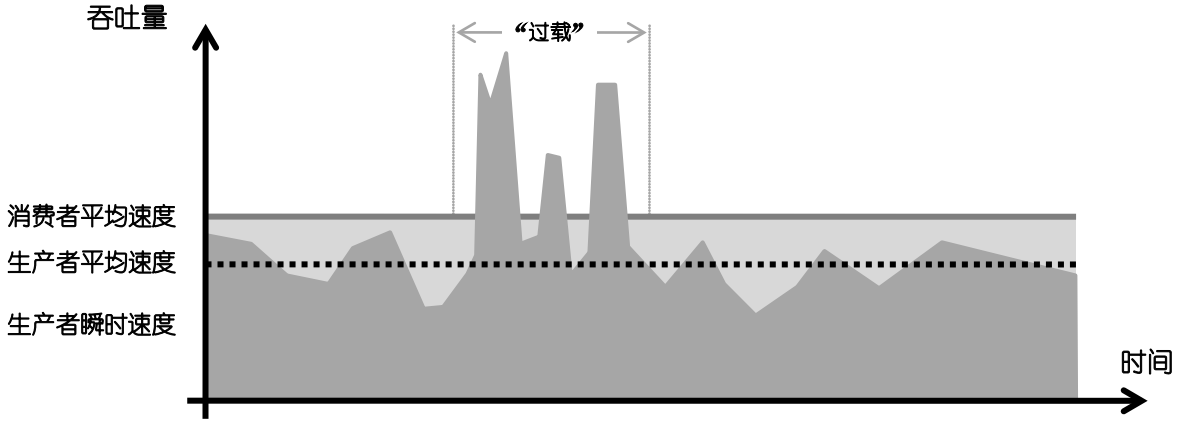
- “生产者”的速度恒小于等于“消费者”的速度

显然，在这种情况下，“消费者”的消费能力是超过“生产者”的，处于吃不饱或勉强吃饱的状态。理论上不存在数据丢失的问题。

- **“生产者”的平均速度小于等于“消费者”的平均速度，但瞬间速度大于消费者的最大处理能力**

当“生产者”的平均产能小于“消费者”的平均消耗时，系统在理论上是有机会解决产品丢失问题的。然而，“平均速度”概念的存在就意味着一定有高于平均值的“瞬时速度”存在，一旦“生产者”的瞬时速度大于“消费者”的最大瞬时吞吐能力时，必然出现数据的丢失，这是显而易见的。缓冲区正是为了解决这一问题而存在的——它通过以“空间换时间”的方法将消费者无法处理的“瞬间”数据暂存下来，基于“生产者”平均速度小于等于“消费者”平均速度这一事实，瞬间“过载”意味着未来一段时间内“生产者”必然存在着较低的瞬时速度（如**图 8.4** 所示），在这期间，满负荷运转的消费者可以从容的将所有数据都消化掉，避免了数据的丢失。

图 8.4 瞬间过载的“生产者”和“消费者”



实际应用中，使用缓冲区的原因很多，但其最本质的作用还是通过“以空间换时间”的策略解决（或者说缓解）“生产者”和“消费者”之间的“瞬时过载”问题。需要强调的是，当且仅当“生产者”的平均速度小于等于“消费者”的平均速度时，使用缓冲区才有可能解决问题，也只有这种情况下，缓冲区的大小取值范围才有明确的计算方法（而不是单纯凭感觉或者依靠所谓“工程师的经验”）；否则，缓冲区的使用只能算是“掩耳盗铃”，最多只能说是“缓解”了速度差异带来的矛盾，而完全无法保证解决这类问题。毫不客气地说，相当多的嵌入式软件工程师都是“鸵鸟”，每逢遇到速度差异的问题，非常喜欢一头扎到“缓冲区”的沙子里面。

关于缓冲区的使用，我们还将在 **8.4 节**以“缓冲范式”的形式详细讨论。这里就不再赘述。

- **“生产者”的平均速度恒大于“消费者”的平均速度**

当“生产者”的平均产能恒大于“消费者”的平均消耗能力时，数据丢失不可避免——即便引入了缓冲，也不过是拖延时间而以，就好像经典的游泳池放水问题所描述的那样：进水速率大于出水速率时游泳池的水位一定是上涨的，并最终漫溢出来——缓冲区撑爆了，数据也还是难逃丢失的厄运。造成这种局面的原因通常是系统设计之初考虑不足。这种情况下，要么重新设计——保持“生产者”和“消费者”能力匹配；要么在不得已或者不允许重新设计的情况下，重新评估消费者的功能，将其拆分成若干子消费者，并找到其中的核心功能（“核心消费者”），以牺牲其它次要功能消费者的代价，尽可能满足核心任务——这就是“偏心范式”的基本思想，我们将在 **8.3 节**的讨论中为您详细展开。

“生产者”与“消费者”的速度差异是客观存在的，是多任务设计中，数据流处理要面对的首要问题。因而，程序员们总是孜孜不倦的尝试去寻找一种程序“逻辑”，用来“完美”的解决“生产者”和“消费者”的速度差异问题。是否存在一个特定的编程范式，可以完美的解决生产者与消费者之间的各种速度差异问题？“完美逻辑”是否存在呢？这是读者需要思考的，其中道理一定要自己想明白。

这里的讨论都是基于一个假设展开的，即，我们已经明确的知道“生产者”和“消费者”各自的能力，因而我们可以分情况进行讨论。然而，实际应用中，更多情况下我们并不能立即知道“生产者”和“消费者”的这些吞吐量信息，这种情况下我们又该如何看待这一问题呢？答案很简单：按最坏情况进行考虑，即“生产者”的平均速度恒大于“消费者”的平均速度。

当然，这也是要分情况的。我们首先需要对消费者进行拆分，并将这些子任务按功能重要性进行排序。如果抛开其他所有任务，最终要的“核心任务”其平均消费能力仍然小于“生产者”，则这个系统本身就是无可救药的，直接放弃好了；如果“核心任务”的平均消费能力大于等于“生产者”，则可以应用“偏心范式”，以“丢车保帅”的姿态来设计系统，牺牲次要任务，确保“核心任务”的功能——这当然是最坏情况，或者说是极限情况（Corner Case），实际上，以这种思路设计的系统往往可以在大部分情形下近似实现“完美逻辑”——所谓“最坏情况下，我们都能确保核心任务得到满足，谁还会害怕平时较为宽松的情况么？”

8.2.2 具体的案例分析

（您所阅读的仅仅是样章，并不包含所有内容，展现的内容也并不是最终形式，详细内容请参考正式书籍）

8.2.3 小结

从前面列举出来的三种速度差异关系来看，只有生产者的速度恒小于消费者速度的时候，从理论上才可能存在可以设计出来的程序逻辑，实现完美的生产行为和消费行为。当两者的速度关系不满足这一点的时候，从理论上就不可能设计出一种程序逻辑，来实现完美的生产和消费行为。即便你偏心——次要消费者得不到保障；即便你缓冲——缓冲总是有限的，总有不够用的时候。从反证的角度来看，产品经理给出的“完美”的命题，总可以找出一个速度差异关系证明其为“伪”。

总之，完美逻辑是不存在的。

作为开发者，当然不能期望设计出一套完美的逻辑来满足所有生产者与消费者速度差异关系。开发者能做的，就是在有限的资源下找到平衡点，做出合理的妥协；未雨绸缪，在未知条件较多时，尽可能以最坏的情况为参考进行设计。大方向来说，有两种常用的妥协思想可以帮助应对生产者与消费者的速度差异问题——偏心范式和缓冲范式。

8.3 偏心范式

（您所阅读的仅仅是样章，并不包含所有内容，展现的内容也并不是最终形式，详细内容请参考正式书籍）

完美的逻辑并不存在，生产者速度恒大于核心任务的情况无法用软件的方式解决。偏心范式讨论的前提就是：

生产者平均速度小于等于核心消费者平均速度

（您所阅读的仅仅是样章，并不包含所有内容，展现的内容也并不是最终形式，详细内容请参考正式书籍）

8.4 缓冲范式

我们已经知道，由于生产者与消费者的速度差问题恒存在，永远无法找到一种完美解决所有速度差异的方法，来实现完美的逻辑。但工程思维和理论思维是不同的，理论思维方式讲究百分之百，讲究“充分又必要”，而工程思维最大差别就是工程中往往没有能达到完美的条件，而是在有限的条件下去满足需求。

我们接下来讨论如何解决这一类速度差异问题，讨论的前提是这样的普遍状况：

生产者的平均速度小于消费者，但瞬时速度可能大于消费者。