# 第一章 嵌入式内核基础

本章主要介绍嵌入式内核的概念和整体结构，并对全书涉及的重要知识做介绍，为我们在以后各章的嵌入式内核分析和学习做好准备。本章假设读者已经对嵌入式系统有初步了解，不再介绍例如发展历史、机制特点这些基本知识。

## 1.1 嵌入式软件系统结构

目前常见的嵌入式软件结构可以分为轮询系统、前后台系统和多任务系统。

### 1.1.1 轮询系统

这是最简单的一种软件结构，主程序是一段无限循环的代码，在循环中顺序查询各个条件，如果满足就执行相应的操作。这种方案的好处是实现简单，逻辑清晰，便于开

发人员掌握。但是每个事件的查询和处理时间是不能确定的。假如前面的操作时间较长，那么后面的操作必然会被延迟。

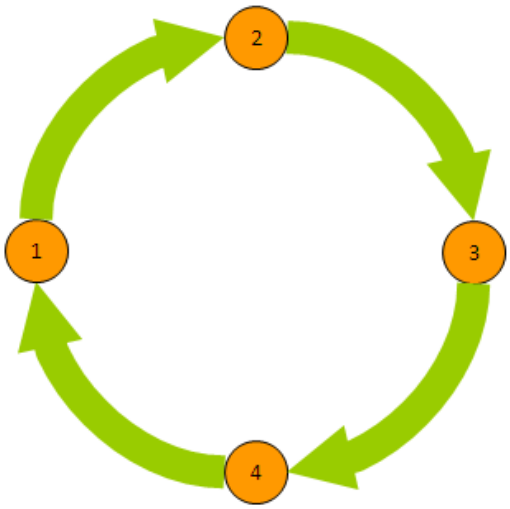



**图 1-1 轮询系统结构**

在上图中，假如步骤 1 操作需要很久，那么步骤 2 必然得不到及时处理，如果步骤 2 的工作很重要或者很紧急，那么系统的性能和响应能力就很差了。

## 1.1.2 前后台系统

相对轮询系统，前后台系统对外部事件的处理做了优化。前后台系统是由中断驱动的。主程序依然是一段无限循环的代码，称为后台程序，而事件的响应则由中断来完成，称为前台程序。在后台程序执行的时候，如果有外部事件发生，则前台的中断程序会打断后台程序。在完成必要的事件响应之后，前台中断程序退出并通知后台程序来继续操作。由后台程序完成事件的后继处理，比如数据的分析等操作。从代码功能上讲，事件的响应和处理分为了两个部分。因为中断自身有优先级和嵌套的功能，所以优先级高的事件能够得到及时响应。但后台程序仍然需要按顺序的处理各个事件的后继事务。

前后台系统演示如下图所示：

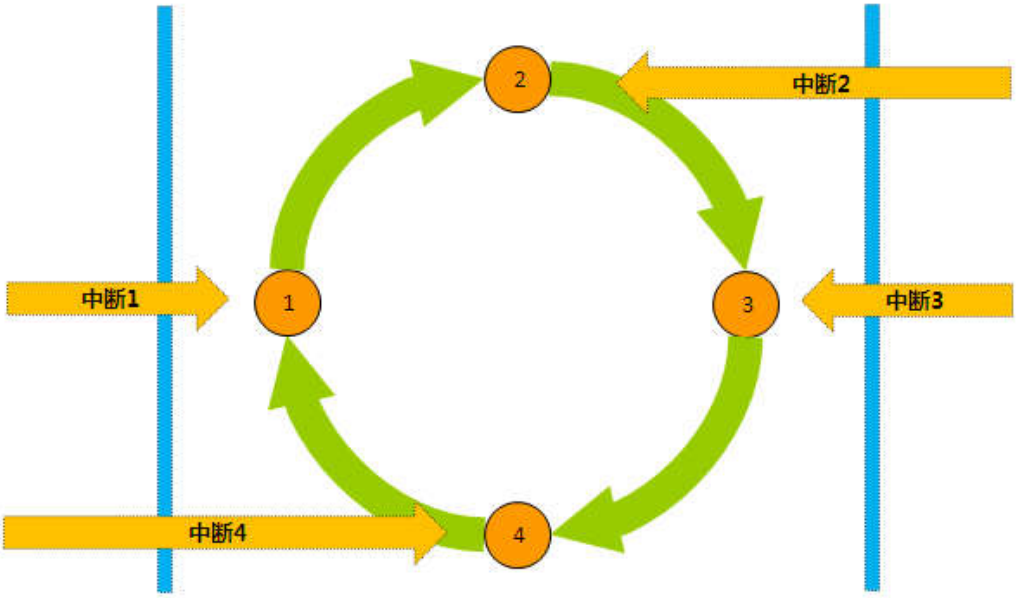



**图 1-2 前后台系统结构**

上图所示，在中断间有优先级的概念。中断 ISR 会首先响应事件，简单的事件可以在 ISR 之中直接处理，复杂的情况下则记录下必要数据和状态标记。等所有中断处理结束后，将由后台主函数顺序处理各个事务。也就是说，事件的响应是支持优先级的，但事件的最终处理却是顺序的。使用中断来代替轮询方案中事件的查询操作，所以相对轮询方案，前后台系统对事件的响应能力有较大改善。

### 1.1.3 多任务系统

和前后台系统相比，多任务系统在响应事件的时候，同样是由多个中断处理程序完成的。但是对于事件的后继操作则是由多个任务来处理的。也就是说每个任务处理它所负责的事件。在基于优先级的多任务系统中，因为任务间优先级的关系，那么优先级高的任务可得到优先处理。这样优先级高的事件就能及时得到处理；在基于分时机制的多任务系统中，则任务间按比例轮流占用处理器。

多任务机制如下图所示：

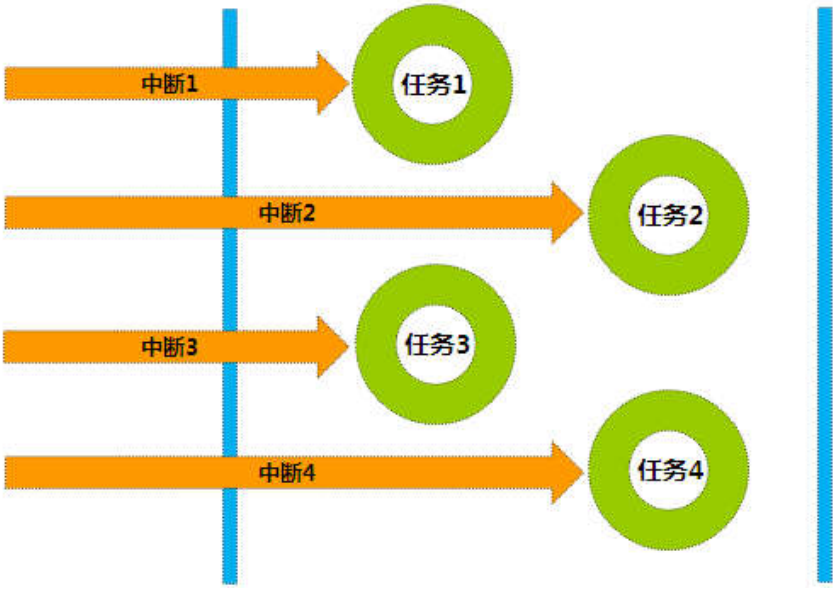



**图 1-3 多任务系统结构**

在上图中，中断用来响应事件，事件的后续操作则由任务来完成。中断和任务都有优先级的概念。假如其中中断 2 和任务 2 处理的事件是紧急的或者重要的，那么当中断 2 发生时，即使其他任务或者中断正在处理，那么也会被抢占，最终任务 2 会得到运行机会。

因为多任务内核允许将具体的应用系统分成若干个相对独立的任务来管理，所以多任务内核的使用可以简化应用应用程序的设计，系统也变得简洁和便于维护和扩展。对实时性要求严格的事件都能得到及时可靠的处理。不过多任务内核自身将占用部分处理器、存储器等硬件资源，这是引入内核机制的必要的代价。

从事件和数据处理的角度考虑，可以把整个应用流程简化为事件响应和事件处理两个阶段。正是对着两个阶段的采用的不同技术手段，才逐步发展出上面介绍的这三种软件结构方案。从这个发展路线可以明显感觉到的就是解决问题的思路越来越清晰，结构和层次越来越合理。

以下是对三种软件结构的比较

| 模型 | 事件响应 | 事件处理 | 特点 |
|---|---|---|---|
| **轮询系统** | 主程序 | 主程序 | 轮询响应事件，轮询处理事件 |
| **前后台系统** | 前台多个中断程序 | 后台单个主程序 | 实时响应事件，轮询处理事件 |
| **多任务系统** | 多个中断程序 | 多个任务 | 实时响应事件，实时处理事件 |

**表 1-1 常见嵌入式软件模型**

通过上面的比较，我们可以清楚的看到嵌入式软件结构上的不同和发展，但这并不是系统结构好坏的标准。每种方案都有它产生的年代、硬件资源的发展阶段和所适合的

应用领域，只能说是技术的发展，是针对应用方案的细分。

多任务系统，指的就是基于多任务操作系统的应用开发模型。本书介绍的就是嵌入式的操作系统的核心部件：嵌入式实时操作系统内核的设计和实现。它的主要功能包括：任务管理、任务调度、任务同步、互斥和通讯、设备管理、中断管理、时间管理等。而像图形用户接口、文件系统、TCP/IP 协议、嵌入式数据库引擎等，则可以归为内核层之上的嵌入式操作系统的功能模块。

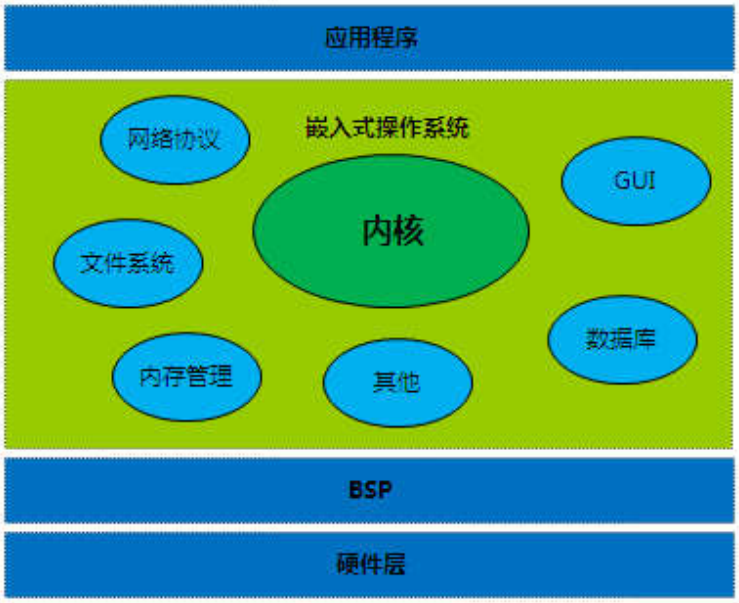



**图 1-4　多任务模型下 RTOS 组成**

关于嵌入式内核，有很多常见的技术概念，熟悉这些知识是我们学习嵌入式内核的基础。在本书内容的编排上，会把各种功能模块的概念放在各章起始，首先介绍其原理，然后分析设计和实现。注意我们是以单处理器多任务模型来讲解的。

## 1.2 多任务机制概述

在前面我们曾介绍了多任务系统是如何演化的。和前后台系统相比较，多任务可以理解为有多个后台程序的前后台系统。每个任务都专注自己处理的问题。下面将详细介绍一下和多任务相关的一些基本概念。

### 1.2.1 时钟节拍

时钟节拍是多任务系统的基础，它指明了把处理器时间以多大的频率分割成固定长度的时间片段。作为多任务系统运行的时间尺度，时钟节拍是通过特定的硬件定时器产生的。硬件定时器会产生周期的中断，在相应的中断处理函数中，内核代码得以运行，从而进行任务调度和定时器时间处理等内核工作。

处理器的时钟节拍如下图演示：

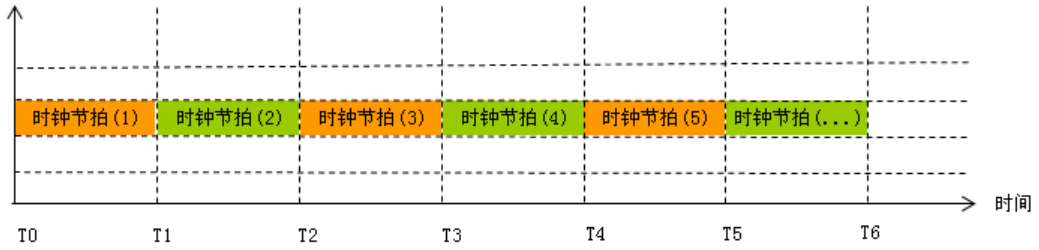



**图 1-5 处理器时钟节拍**

硬件定时器中断的时间间隔取决于不同的内核设计，一般是毫秒级的。时钟节拍越快，内核函数介入系统运行的几率就大，中断响应次数越多，内核占用的处理器时间越长。相反，如果时钟节拍太慢，则任务的切换间隔时间过长，进而影响到系统对事件的响应效果。

下图演示了多任务系统中，中断处理程序和任务在时间上的关系：

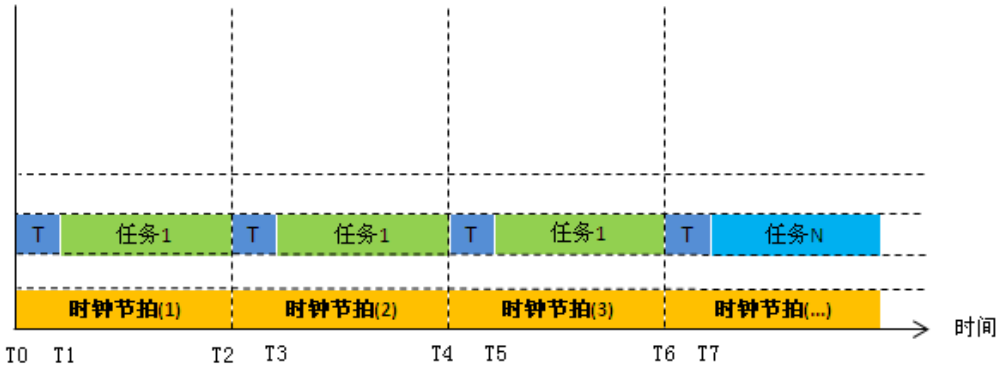



**图 1-6 多任务系统中的中断和任务的时间关系**

如上图所示，硬件定时器按照固定的时间间隔产生中断，然后在 ISR 中（图中 T 标记）处理内核的工作。在 T0～T1 这段时间是内核占用的时间（时钟节拍处理程序），T1～T2 这段时间是任务占用的时间。而 T0～T2 则是一次时钟节拍的全部时间。从上图中可以看出，任务 1 的本轮执行占用了 3 个时钟节拍。

## 2.1.2 多任务机制

在单处理器的计算机系统中，在某一具体时刻处理器只能运行一个任务，但是可以通过将处理器运行时间分成小的时间段，多个任务按照一定的原则分享这些时间段的方法，轮流加载执行各个任务的方法，从而使得在宏观上看，有多个任务在处理器上同时执行，这就是单处理器系统上的多任务机制的原理。如下图所示：

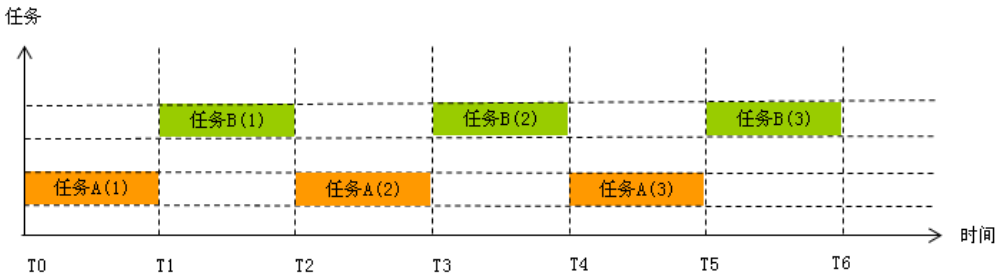



**图 1-7 多任务机制演示**

在上图中，任务 A 和任务 B 按照等长时间轮流占用处理器，在单处理器上造成多个任务同时运行的假象。

另外，因为不同任务的运行路径不同，在某一时刻有些任务可能需要等待一些资源，这时可以通过某种方案，使得当前任务让出处理器，从而避免因为任务等待资源而长期占有处理器，使得其他任务得以运行。这样多任务机制可以使处理器的利用率得到提高并提高了系统的响应时间。

在多任务操作系统内核中,必须提供解决并发任务的机制。通用内核一般以“进程”、“线程”等为单位来管理用户任务。在相关资料中，也会明确指出“进程”与“线程”的区别。但在很多嵌入式内核中，并没有区分进程和线程，只是把整个内核当作一个大的运行实体，其中运行着很多任务。任务通常作为调度的基本单位。

### 1.2.3 任务上下文

任务可以看作是用户程序在处理器等硬件上的运行，是一个动态的概念。任务在处理器上运行的某一时刻，有它自己的状态，即处理器所有的寄存器的数据，这个叫做任务的上下文，可以理解为是处理器的“寄存器数据快照”。通过这些数据，内核可以随时打断任务的运行或者加载新的任务，从而实现不同任务的切换运行。

任务上下文是跟处理器密切相关的概念，不同的处理器有不同的处理器上下文定义。比如在 Cortex-M3 处理器中的寄存器是这样的：

- 拥有 R0-R15 寄存器组
- R0-R12 是通用寄存器。
- R13 作为堆栈指针 SP 有两个，但在同一时刻只有一个起作用。
- R14 连接寄存器。
- R15 程序计数器，指向当前的程序地址。

另外还有特殊功能寄存器：

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

在 RTOS 设计任务上下文的时候经常会把大部分硬件寄存器做为任务上下文的内

容，这点在介绍内核移植的时候会做详细介绍。

### 1.2.4 任务切换

任务切换又叫做任务上下文切换。当内核需要运行其它的任务时，内核首先会保存和当前任务相关的寄存器的内容到当前任务的栈中；然后从将要被加载的任务的栈中取出之前保存的全部寄存器的内容并加载到相关的寄存器中，从而继续被加载任务的运行。这个过程叫做任务切换。

任务基本的切换过程如下图所示：

假设系统中有两个任务 A、B。当前处理器正在运行 A 任务。此时任务 A 的栈顶在变量 An 处：

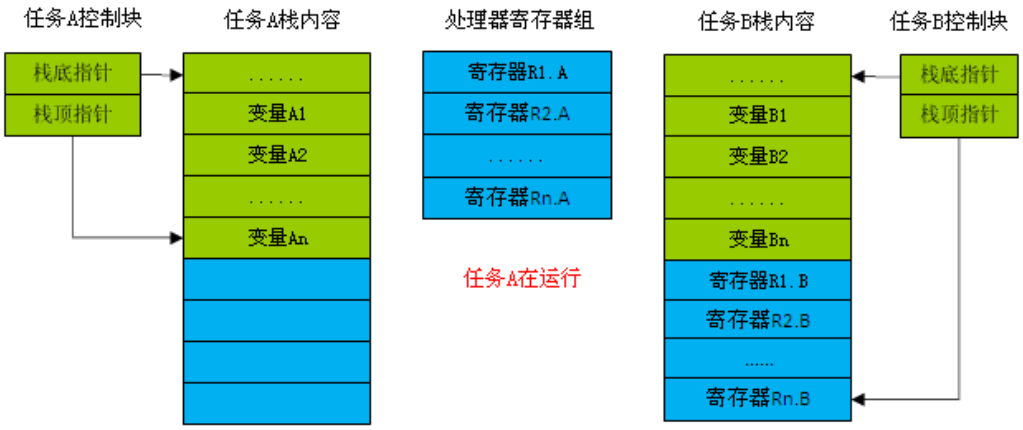



**图 1-8 任务 A 运行时的上下文情况**

然后发生任务调度，需要首先保存当前的处理器寄存器组的内容到任务 A 的栈中：

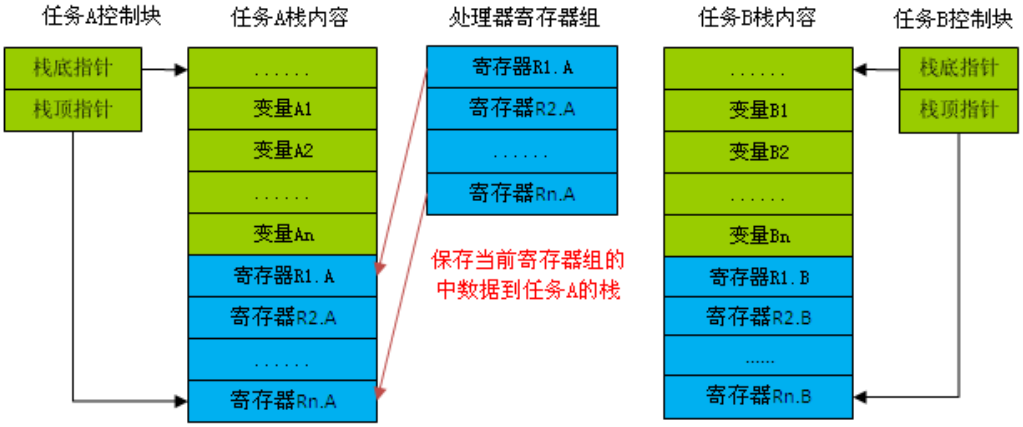



**图 1-9 任务 A 上下文保存**

接下来的操作就是把保存在任务 B 栈中的处理器寄存器组的内容调入到处理器寄存器组中：

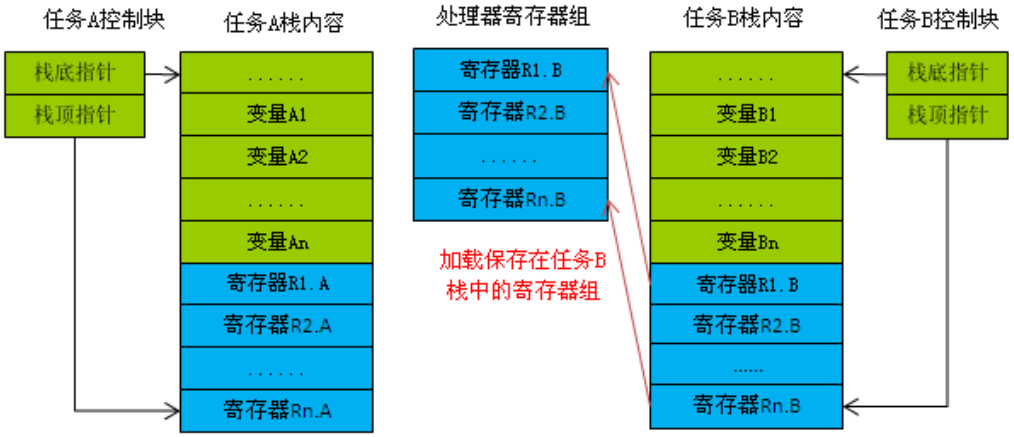



**图 1-10 任务 B 上下文恢复**

最后，处理器开始继续执行任务 B：

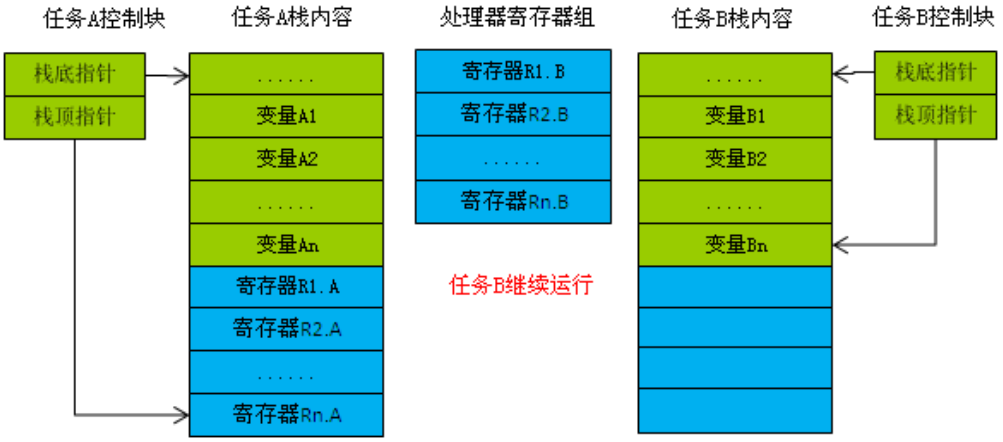



**图 1-11 任务 B 运行时的上下文情况**

这里给读者留两个问题：

- 保存在任务 B 的栈中的最初始的寄存器组的内容从哪里来的？
- 保存在最先执行的任务的栈中的寄存器组的数据是如何加载到处理器寄存器组的？

## 1.2.5 任务的时间片和优先级

时间片指的是任务一次投入运行，在不被抢占或者中断的情况下，能够连续执行的最长时间（以时钟节拍计数）。时间片的长度由具体内核规定，有些内核中不同任务可以有不同的时间片长度，或者是在运行过程中可以动态改变时间片长度。

时间片长度是时钟节拍的整数倍。如下图所示：

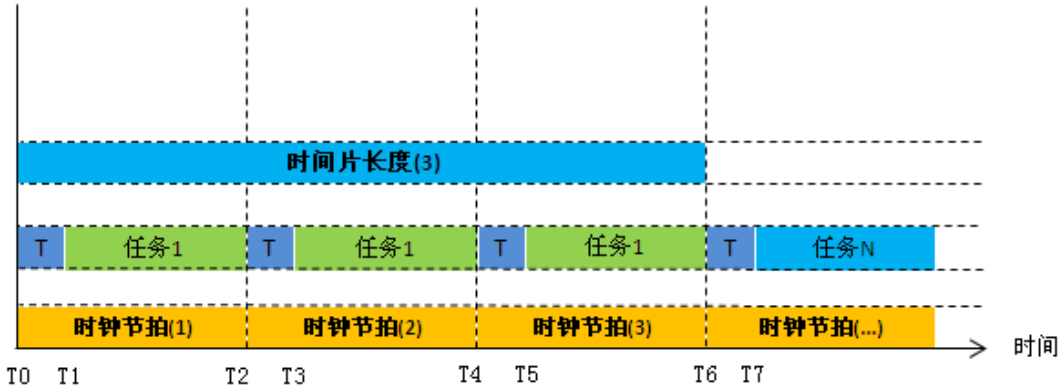



**图 1-13 时间片和时钟节拍关系**

从上图中可以清楚的看出，任务时间片、时钟节拍、定时器 ISR 之间的时间关系：时间片长度是时钟节拍的整数倍，一个时钟节拍被定时器 ISR 和任务共享。

任务的优先级用来安排系统中各个任务的执行次序，它说明了任务的重要性，任务越重要，它的优先级应越高，越应该获得处理器资源。任务优先级的安排有两种方式：静态优先级和动态优先级。如果任务优先级在运行的过程中不能改变，则称为静态优先级。静态优先级是在任务初始化时候决定的；反之如果任务优先级是可以改变的，则称为动态优先级。

时间片和优先级是任务的两个重要参数，分别描述了任务竞争处理器资源的能力和持有处理器时间长短的能力。这两者同时是任务抢占的重要参数。因任务时间片运行完毕而引起的任务调度可以理解为时间片调度，而因为内核中最高就绪优先级的变化而引起的调度则为优先级调度。

### 1.2.6 任务调度和调度方式

任务调度是内核的主要功能之一，在任务需要调度的时候，内核会根据具体的调度算法和策略选择合适的任务，替换当前任务占有处理器等硬件资源。根据调度原理的不同，任务调度方式可分为可抢占调度和不可抢占式调度两类。

- 对于基于优先级的系统而言，可抢占型调度是指内核可以剥夺正在运行任务的处理器使用权并交给拥有更高优先级的就绪任务，让别的任务运行。

- 对于基于分时机制的系统而言，每个任务都能持续占用处理器一段时间，每轮时间用完之后内核就剥夺处理器给别的任务来执行。可以把这种调度方式理解为时间引起的抢占。

- 在不可抢占型调度方式下，如果某任务占有了处理器，那它就一直执行，直到它主动将处理器控制权让给别的任务使用。内核不会强制它释放处理器资源。

基于优先级的可抢占式调度的实时性好，任何优先级高的任务只要具备了运行的条件，即进入了就绪态，就可以立即得到调度和运行。任务在运行过程中都随时可能被比

它优先级高的任务抢占。这种方式的任务调度保证了系统的实时性。

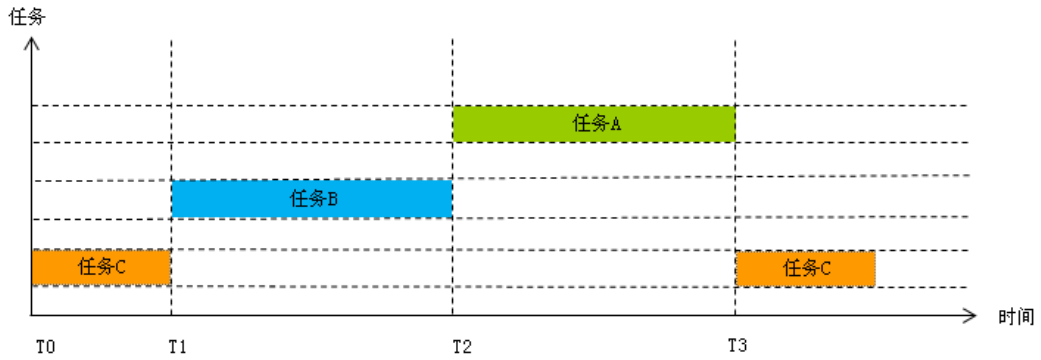



**图 1-12 不可抢占内核任务执行情况**

上图演示了不可抢占内核中的任务执行情况。例子中有三个任务需要运行。系统的执行流程如下：

- 在 T0 时刻任务 C 得到处理器，它开始执行。在它执行过程中，任务 B 和任务 C 就绪，但是因为内核不支持抢占式调度，所以它们只好等待机会。
- T1 时刻任务 C 完成操作，主动让出处理器。内核选择任务 B 来运行
- T2 时刻任务 B 完成操作，主动让出处理器。内核选择任务 A 来运行
- T3 时刻任务 A 完成操作，主动让出处理器。内核再次选择任务 C 来运行

对于抢占式调度方式，下面章节会有详细的图示。

### 1.2.7 任务调度算法

常见的任务调度机制主要有时间片轮转调度算法（时分式）、优先级调度算法（抢占式）和基于优先级的时间片调度算法。

**时间片调度算法**

指的是内核先让某个任务运行一个时间片（多个时钟节拍），然后再切换给另一个任务。这种算法保证每个任务都能够轮流占有处理器。因为不是在每个时钟节拍都做任务调度，对处理器资源的消耗又做到了最少。时间片调度算法缺点是在任务占有处理器的时间段内，即是有更紧急任务就绪，也不能立刻执行它。

时间片的调度如下图所示：

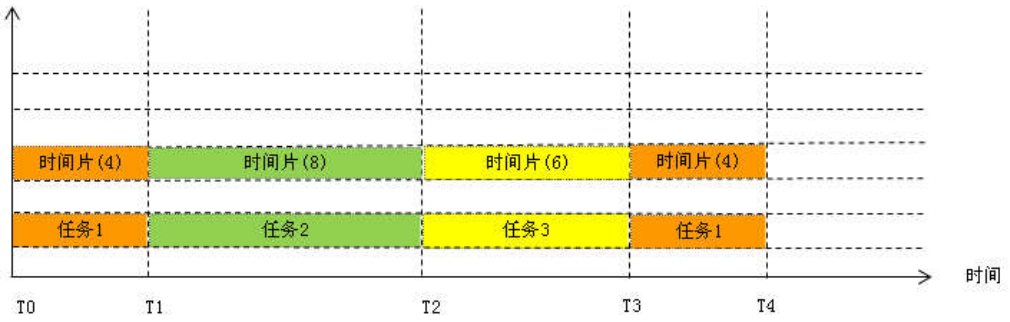


**图 1-14 时间片调度机制演示**

如上图所示，

- 假设任务 1 时间片长度为 4；任务 2 时间片长度为 8；任务 3 时间片长度为 6；
- T0 时刻任务 1 得到运行；
- T1 时刻任务 1 时间片结束，本轮运行结束；任务 2 开始运行；
- T2 时刻任务 2 时间片结束，本轮运行结束；任务 3 开始运行；
- T3 时刻任务 3 时间片结束，本轮运行结束；任务 1 开始新一轮运行；

**优先级调度算法**

指的是内核总是让具有最高优先级的就绪任务优先运行。即当有任务的优先级高于当前任务优先级并且就绪后，就一定会发生任务调度，这种内核最大的提升了系统的实时性。

优先级调度算法如下图演示：

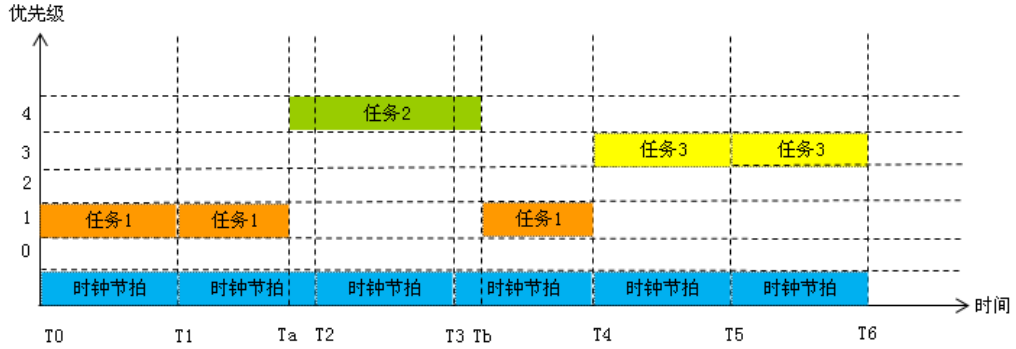



**图 1-15 优先级调度算法演示**

上图只是演示了在时钟节拍中进行优先级调度的情况

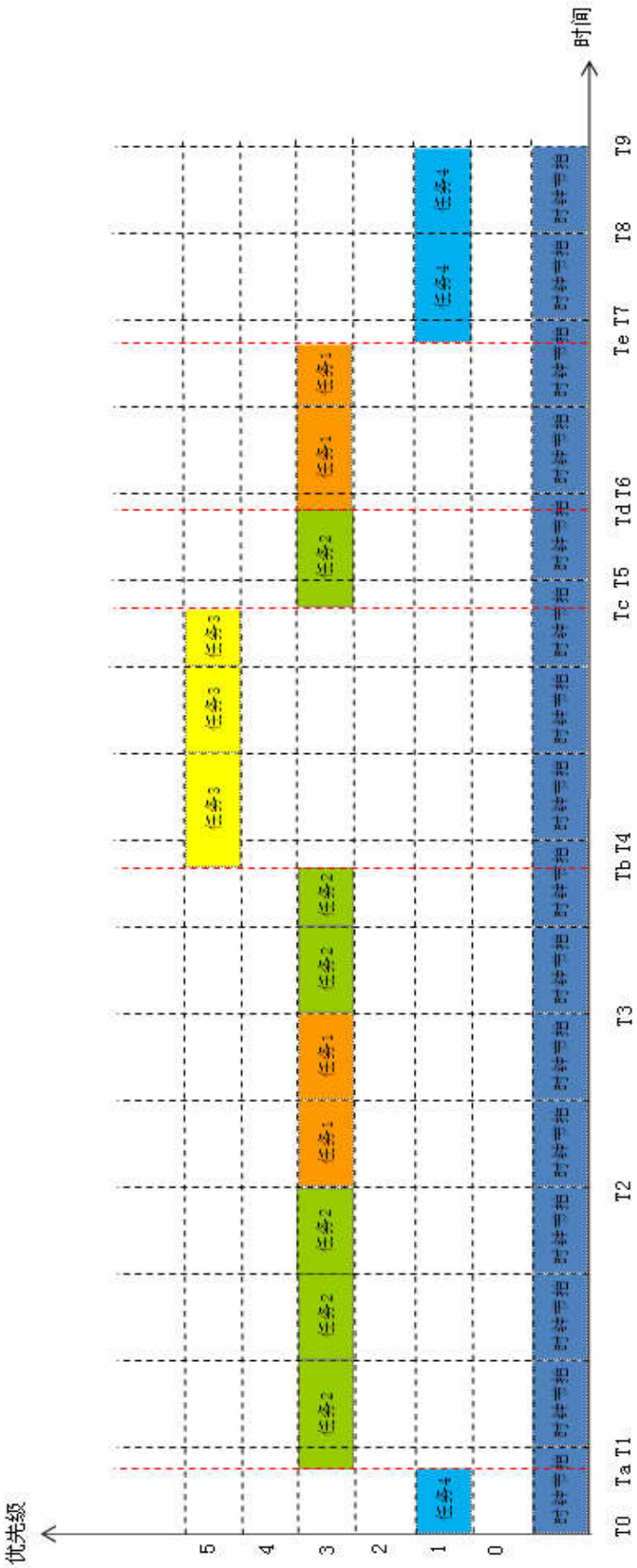
- 假设任务 1 优先级为 1，任务 2 优先级为 4，任务 3 优先级为 3；数值越大优先级越高。
- T0 时刻系统中只有任务 1 就绪，所以首先得到调度运行;
- T1 时刻因为没有别的任务就绪，所以任务 1 继续运行
- Ta 时刻任务 2 就绪，因为任务 2 优先级高，所以它抢占任务 1 开始运行。**抢占发生在 T1-T2 时间节拍之间**。
- T2 时刻因为没有更高优先级任务就绪，所以任务 2 继续运行
- T3 时刻因为没有更高优先级任务就绪，所以任务 2 继续运行
- Tb 时刻任务 2 放弃了处理器(比如任务被阻塞)，任务 1 得到运行。
- T4 时刻，假设此时在时钟节拍 ISR 里唤醒了任务 3，因为任务 3 优先级高，所以任务 3 抢占任务 1。**抢占发生在时钟节拍处。**
- T5 时刻，任务 3 继续运行。

优先级调度算法的缺点是，当最高优先级任务在运行时，它将持续占有处理器直到任务结束或者阻塞，否则其它任务无法获得运行的机会。

**基于优先级的时间片调度算法**

吸收了以上两种算法的优点，同时又解决了它们的不足。这种算法为每个任务都安排了优先级和时间片。在不同优先级的任务间采用优先级调度算法，在相同优先级的任务间使用时间片轮转调度算法。任务调度策略首先考虑任务的优先级，优先级高的任务必定会抢占低优先级的任务。相同优先级的任务则按照时间片长度比例共享处理器时间。这样既保证了能够尽快响应紧急任务，又保证相优先级的任务都有机会轮流占有处理器。

该算法如下图所示：




Trochili RTOS 源码链接 https://github.com/liuxuming/trochili

**图 1-16 基于优先级的时间片调度算法**

如上图所示

- 任务 1 优先级为 3，时间片长度为 2 个时钟节拍；任务 2 的优先级为 3，时间片长度为 3；任务 3 优先级为 5，时间片长度为 1 个时钟节拍；任务 4 优先级为 1，时间片长度为 2 个时钟节拍；
- T0 时刻，系统中只有任务 4 就绪，所以任务 4 得到处理器，运行；
- Ta 时刻，任务 2 和任务 1 就绪，任务 2 因优先级抢占得到处理器，开始运行；
- T2 时刻，任务 2 时间片耗尽，只好释放处理器；任务 1 得到运行的机会；
- T3 时刻，任务 1 时间片耗尽；任务 2 得到运行的机会；
- Tb 时刻，任务 3 就绪，发生优先级抢占；任务 3 得到运行的机会；
- Tc 时刻，任务 3 处理完自己的工作，主动释放处理器；任务 2 得到运行的机会；
- Td 时刻，任务 2 处理完自己的工作，主动释放处理器；任务 1 得到运行的机会；
- Te 时刻，任务 1 处理完自己的工作，主动释放处理器；任务 4 得到运行的机会；

### 1.2.8 任务状态

在多任务的系统中，任务一般具有多种状态，反映任务不同的执行阶段。常见任务状态主要有 3 种：

- 就绪状态　任务已经获得除处理器之外的一切需要的资源，等待任务调度
- 运行状态　任务正在运行中，它得到了所有需要的资源
- 等待状态　任务缺少某些必须的运行条件或资源而不能参与任务调度

不同的 RTOS 可能有不同的状态定义，可能会更细致的定义一些状态。

- 

## 1.3 同步、互斥和通讯

在多任务系统中，在任务间、ISR 和任务间必然存在着处理器交替抢占，轮流执行的情况。除此之外，这些可执行对象也存在着其它关系，仔细观察这些对象，它们总是要“走走停停、互相照应”，这也正是多任务系统的特点，只有这样设计系统才能使得硬件资源得到最大的利用。可以把它们间的关系总结如下：

- 共享资源的竞争

  任务或者 ISR 访问共享资源时是互相竞争的，只能被一个任务或者 ISR 访问，并且操作时不能被打断。强调的是“互斥”的概念。

- 运行同步

  任务间或者任务和 ISR 间互相协作，按照规定的路线执行，也就是对它们的执行步骤和顺序有要求。强调的是“同步”的概念。同步可以是单向的也可以是双向的。

- 数据通信

  任务间或者任务和 ISR 间的数据传输，常见的模式是一方提供数据，另一方处理数据，共同完成某些功能。强调的是“通信”的概念。

  任务间的数据传输，可以是直接的，也可以是间接的。

  - 在直接数据传输方式下，一个任务可以把数据直接发给指定的任务，发送过程很明确的说明了哪个任务把数据传给了那个任务。
  - 间接方式指的是数据交互的双方，约定一个数据缓冲区，发送数据的任务首先会把数据发往该缓冲区，然后通知接收数据的任务则从该缓冲区取得数据。从内核角度来考虑，内核不关心这些数据的含意，只当普通的数据来处理。

在本文后面，经常会采用 IPC(Inter-Process Communication)来代表以上各种关系的操作。从字面含义上讲，IPC 这个词并不是很恰当。常见的 IPC 机制包括信号量、邮箱、消息队列、事件集合、条件变量、管道等。

### 1.3.1 任务等待和唤醒机制

当任务在试图访问 IPC 对象时候，经常会因为运行条件不足而失败，被迫返回或者阻塞在该 IPC 对象的任务阻塞队列。而当有任务释放资源从而使得资源条件可以满足时，内核将会唤醒 IPC 对象上的阻塞任务，使得被唤醒任务继续运行。不同的访问等待机制和唤醒机制是各种内核的重要区别。

用于任务访问 IPC 对象的等待机制主要有三种：

- **直接返回结果**

  任务直接返回访问结果，或者成功或者失败；注意因为 ISR 不像任务那样能够被阻塞，所以 ISR 必须采用本模式。

- **阻塞等待模式**

  任务如果访问 IPC 对象失败，则进入该 IPC 对象的等待队列，直到明确得到处理。

- **时限等待模式**

  任务如果得不到 IPC 对象，则进入等待状态并开始计时。如果在等待期间得到了 IPC 对象则返回操作成功；如果当计时结束的时候任务仍然没有成功，那么它并不会继续等下去，而是返回失败的结果。

当任务不能获得资源而进入资源的等待队列之后，如果某个时刻资源可用，那么内核就该决定怎么处理这些等待任务。这就涉及到了内核任务唤醒机制。

内核唤醒机制主要有的三种模型：

- 当资源可以可使用时，唤醒该资源的全部等待任务。让这些任务与系统中的其他任务平等竞争资源。这种策略会使系统瞬间繁忙，在参与竞争资源的所有任务中，最

终只有一个任务获取到资源，没有得到资源的任务将再次进入资源的等待队列。

- 将该资源等待队列中的一个合适的任务唤醒。这个任务将和系统中可能访问该资源的其他任务一起竞争这个资源。如果这个任务最终没有竞争到资源，它会再次进入该资源的等待队列。

- 内核从等待队列中找到一个最佳的任务并立刻把资源交给它，这样该任务直接从释放资源的那个任务那里获得资源。

目前主流嵌入式内核都采用第三种方案。

### 1.3.2 任务互斥和优先级反转

在系统中，有些资源必须是独占使用的，多个任务对这样的资源的并发访问将导致错误的发生。一般来说，对需要独占使用的资源必须使用互斥方法来将对其的并发访问串行化。

在优先级多任务系统中引入互斥方案，会导致任务优先级反转的问题：假如某时低优先级的任务占有资源，然后又有高优先级的任务申请资源，但因为不能满足而被挂起了，即低优先级任务阻塞了高优先级任务的运行。假如这时又有一个中优先级任务，那么它会把低优先级任务抢占。最终高优先级任务会间接的被中优先级任务抢占了。这种现象叫做优先级反转。举例说明：

假如 A、C、D 三个任务优先级从高到底排列，任务 A 和 C 共享互斥信号量 R，如果某一时刻任务 C 已经获得互斥信号量 R，而任务 A 此时尝试占用 R，那么任务 A 会因为得不到 R 而阻塞在 R 的任务等待队列中。再假设此时任务 D 因为优先级高于任务 C 从而抢占了 C，进而长期占有处理器资源，那么就相当于低优先级的任务 D 间接阻塞了高优先级的任务 A 的运行。

这个过程可以用下图来说明：

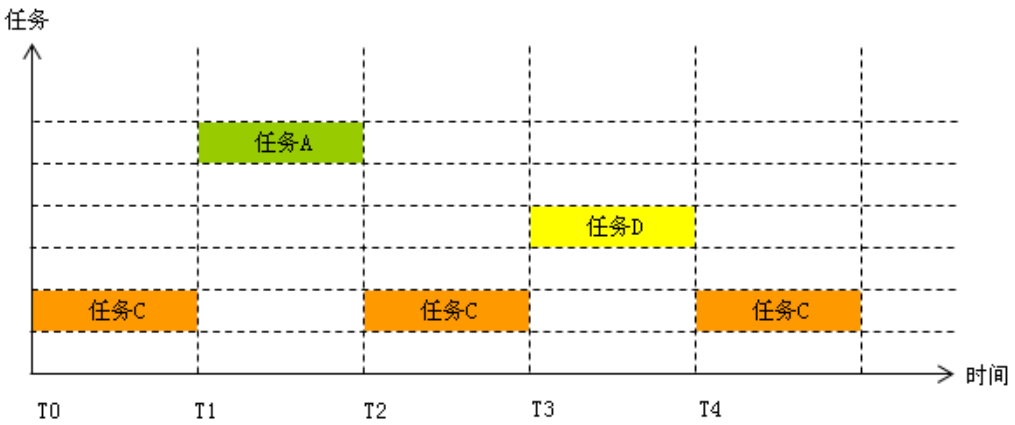



**图 1-17 优先级反转**

- T0 时刻，任务 C 处于运行状态，运行过程中，任务 C 获得了共享资源 R；
- T1 时刻，任务 A 就绪。由于任务 A 优先级高于任务 C，所以它抢占了任务 C，任务 A 被调度执行；
- T2 时刻，任务 A 需要共享资源 R，但 R 被更低优先级的任务 C 所拥有，所以任务 A 被阻塞等待该资源。任务 C 得到执行；
- **T3 时刻，此时任务 D 就绪，由于任务 D 优先级高于任务 C，所以它抢占了任务 C，任务 D 被调度执行；**

从整个流程上看，T3 时刻，高优先级任务 A 被低优先级任务 D 间接的抢占了。此时优先级最高的任务 A 不仅要等任务 C 运行完，还要等优先级低的任务 D 运行完才能被调度，如果任务 D 和任务 C 需要执行很长时间，那么任务 A 的执行就不能得到保证，整个系统的实时性能就很差了。

优先级翻转现象对基于优先级调度的实时系统则有很大的影响。在基于优先级调度的系统中，处理器资源是按照优先级分配给任务的，就绪的高优先级任务必须实时获得处理器。系统中的各种资源，如果采用按照任务优先级分配的原则，那么高优先级的任务应该是首先被考虑的。优先级反转的问题则将打乱这些原则。

优先级反转是由来已久的问题，自从出现多任务系统后，就一直困扰着开发人员，很典型的一个优先级翻转引起的问题是在 1997 年 7 月美国探路者火星车（Pathfinder）发生的在火星表面不停重启的故障，读者如果有兴趣的话可以自己去找些资料看看。

### 1.3.3 优先级天花板和优先级继承

优先级反转问题的核心原因在于共享资源的访问规则，即共享资源只能被一个任务占用，被占用后其它任务不能强制使用这个资源。在优先级反转问题上，高优先级任务被低优先级任务阻塞是必定的，但被中优先级阻塞则是很无奈的。为了避免因为中优先级任务挟持低优先级任务从而阻塞高优先级任务的现象，可以采用一些必要的算法。

有两种经典的防止优先级反转的方法：

- 优先级继承策略(Priority inheritance)
  指的是当一个任务占有了资源并且随后阻塞了其他申请该资源的任务的时候，该任务将临时改变它的优先级为所有申请该资源的任务中的最高的优先级。并以这个临时优先级在临界区执行。当任务释放资源后，则恢复它原有的优先级。从行为上看，占有资源的任务的优先级将是“水涨船高”式的多次改变，因为他的优先级最高，所以它将不被曾经比它优先级高的那些任务的抢占。内核从优先级角度安排它尽快执行，尽快释放资源。但是这样做内核却牺牲了中等优先级任务的调度机会。

- 优先级天花板策略(Priority ceilings):
  优先级天花板是指将申请（占有）资源的任务的优先级提升到可能访问该资源的所有任务的最高优先级。(这个最高优先级称为该资源的优先级天花板)

### 优先级继承协议

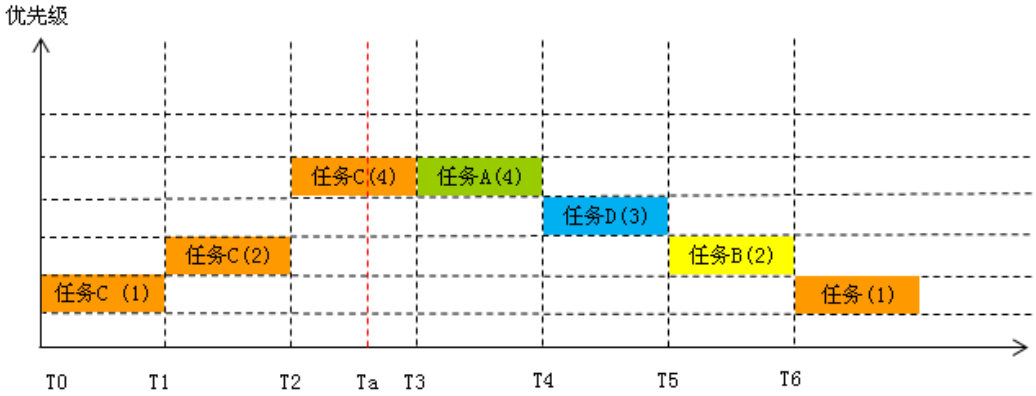



**图 1-18 优先级继承协议**

- T0 时刻，只有任务 C 处于运行状态，运行过程中，任务 C 得到共享资源 R；
- T1 时刻，任务 B 抢占任务 C，并尝试获得资源 R，因为优先级继承的原因，任务 C 的优先级被提升到任务 B 的优先级；任务 B 被阻塞。
- T2 时刻，任务 A 抢占任务 C，并尝试获得资源 R，因为优先级继承的原因，任务 C 的优先级被提升到任务 A 的优先级；任务 A 被阻塞。
- **在 Ta 时刻，任务 D 就绪，但因为此时任务 C 的优先级已经被提升并且比任务 D 优先级高，所以任务 D 不能抢占任务 C，任务 C 继续运行。**
- T3 时刻，任务 C 释放资源，它的优先级恢复到原有优先级。任务 A 得到资源,并因为优先级原因抢占任务 C。
- T4 时刻，任务 A 释放资源，结束运行。任务 B 得到资源。但此时因为任务 D 优先级高于任务 B，所以任务 D 得到执行。
- T5 时刻，任务 D 结束运行，任务 B 开始执行。

### 优先级天花板协议

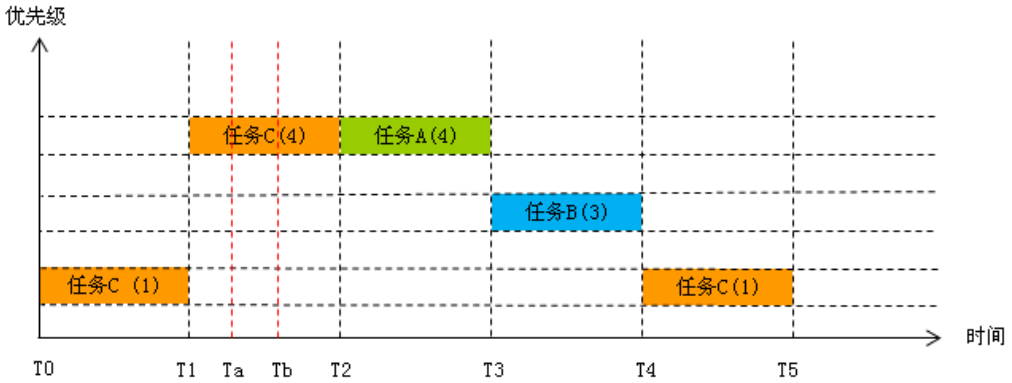



**图 1-19 优先级天花板协议**

- T0 时刻，只有任务 C 处于运行状态。
- T1 时刻，任务 C 得到共享资源 R，因为优先级天花板策略的原因，任务 C 的优先级提升到全部可能访问该资源的任务的最高优先级。
- Ta 时刻，任务 A 抢占任务 C 执行，随后尝试获得资源 R，但是失败并阻塞。任务 C 继续运行。

- Tb 时刻，任务 B 就绪，但是因为任务 C 优先级更高，所以只能等待执行
- T2 时刻任务 C 释放资源 R，任务 A 得到资源 R。因为优先级天花板策略，任务 C 优先级恢复到原有优先级。任务 A 抢占任务 C 开始运行。
- T3 时刻，任务 A 结束运行。任务 B 开始执行。
- T4 时刻，任务 B 结束运行。任务 C 开始继续运行。

优先级继承策略对任务执行流程的影响相对较小，因为只有当高优先级任务申请已被低优先级任务占有的共享资源这一事实发生时，才提升低优先级任务的优先级。而天花板策略是谁占有就直接升到最高。形象一些说，优先级继承协议是“水涨船高”，而优先级天花板协议则是“一次到位”。

## 1.4 中断机制

中断机制是处理器的重要基础设施。它用来对各种事件的响应和处理。当外设或者处理器自身有事件发生时，处理器会暂停执行当前的代码，并转向处理这些中断事务。在处理器与外设间的交互大多采用中断来完成，中断系统能极大提高系统的效率。

发出中断请求的来源叫做中断源。根据中断源的不同，可以把中断分为以下三类：

- 外部中断

  一般是指由系统外设发出的中断请求，如：串口数据的接收、键盘的敲击、打印机中断、定时器时间到达等。外部中断大多是可以屏蔽的，程序可以根据具体需要，通过中断控制器来屏蔽这些中断请求。

- 内部中断

  指因处理器自身的原因引起的异常事件，如非法指令，总线错误（取指）或者运算出错(除 0)等。内部中断基本是不可屏蔽的中断。

- 软件中断

  这是一种特殊的中断，它是程序通过软件指令触发的，从而主动引起程序流程的变化。比如在用户级运行的程序在某时刻需要访问处理器中受到保护的寄存

**器，则可以通过软件中断进入系统级，实现权限的提升。**

在不同的处理器中，以上三种中断可能有不同的名称或者概念，但从技术层面划分，各种中断基本都属于这三类。所以读者不必纠结于不同处理器上的具体称谓。比如在 ARM Cortex M3 内核中，将软件中断和内部中断统称为异常，把外部中断称为中断，并通过中断向量表把这些中断和异常组织在一起，如下表所示，

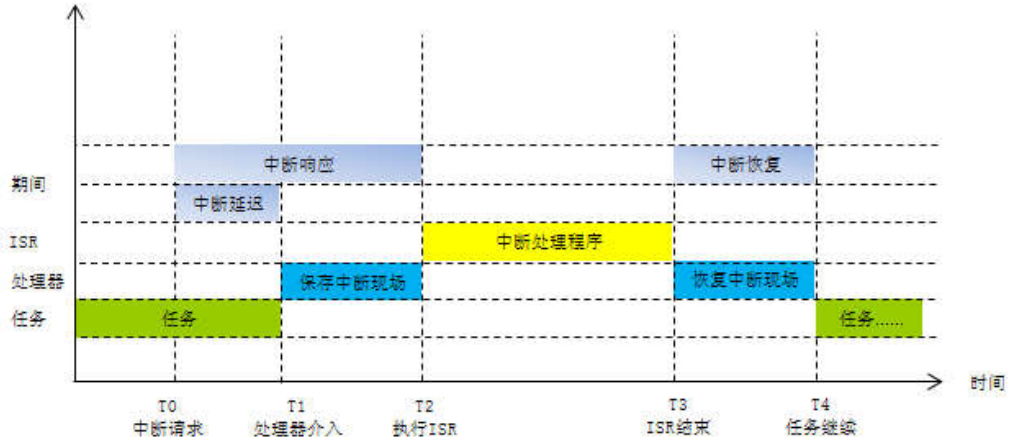
Cortex M3 在内核中拥有一个异常处理系统，用来支持各种系统异常和外部中断。

| 编号 | 类型 | 介绍 |
|---|---|---|
| 0 | N/A | 没有异常 |
| 1 | 复位 | 系统复位 |
| 2 | NMI | 不可屏蔽中断 |
| 3 | 硬 fault | 所有不被处理的 fault 都统一引起硬 fault |
| 4 | 内存管理 fault | |
| 5 | 总线 fault | |
| 6 | 用法 fault | |
| 7-10 | 保留 | |
| 11 | SVC Call | 由系统调用指令 SVC 引起 |
| 12 | 调试器 | |
| 13 | 保留 | |
| 14 | PendSV | 系统级中断服务 |
| 15 | SysTick | 系统时钟中断 |
| 16 | IRQ#0 | 0 号外中断 |
| 17 | IRQ#1 | 1 号外中断 |
| … | | |
| 255 | IRQ#255 | 255 号外中断 |

其中编号为 1－15 的对应系统异常，其余则是外部中断。

### 1.4.1 中断流程概述

不同的处理器上的中断处理流程大致是相同的，但也有些微差别。所以这里我们只介绍中断大概的几个阶段。




如上图所示

- T0 时刻，用户程序正在执行，此时有外部设备产生中断请求
- T1 时刻，处理器开始处理外部中断，保存中断现场
- T2 时刻，处理器开始执行中断处理器函数
- T3 时刻，用户中断处理函数结束，处理器开始恢复中断现场
- T4 时刻，被中断的用户程序继续执行

其中从 T0 时刻中断产生到 T1 时刻处理器开始处理中断的这段时间称为中断延时，这个和处理器是相关的。也就是说，外部中断并不是一发生就被立刻处理的。

从 T1 时刻开始，处理开始处理中断，一般来说，此时最重要的工作就是保留中断现场，以保证能够正确恢复任务的执行。中断现场主要的内容就是处理器上下文，而数据一般是保存到某个具体的栈中。这个同样和处理器是相关的。

T2 时刻处理器就可以正常去执行用户定义的中断处理函数。而从 T0 到 T2 这段时间被称为中断响应时间。

T3 时刻中断处理函数完毕，通过特殊的指令或者流程，处理器从栈中开始恢复本次中断的现场。T3 到 T4 的时间称为中断恢复时间。

以上流程做了最简单的介绍，其实中断流程还有很多细节问题，比如中断的嵌套的问题，处理器保存上下文的问题，RTOS 内核介入中断处理的问题。

### 1.4.2 中断优先级

当几个中断同时产生的时候，首先响应那个中断就是个值得考虑的问题。当前绝大多数的处理器都支持中断优先级的概念，也就是说，为不同的中断源配置不同的优先级，

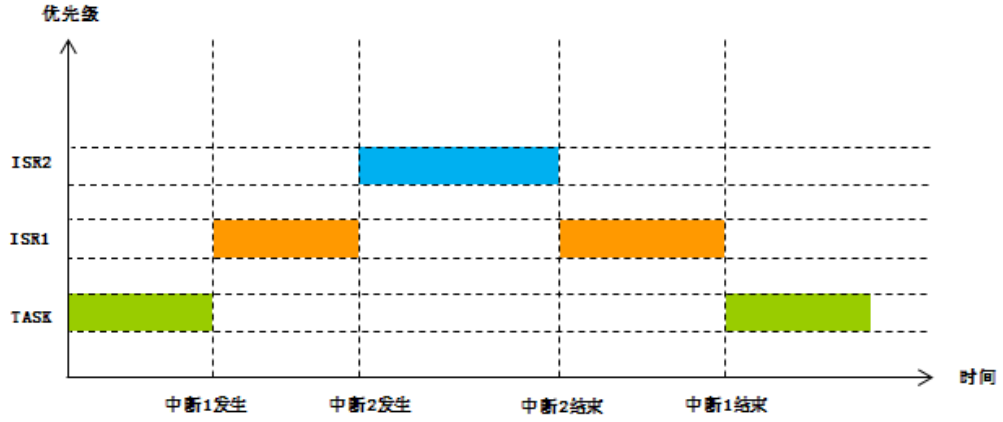
优先级是固定的或者可以通过软件配置的。当多个中断同时产生(或者说需要处理)的时候，首先响应优先级高的中断，这样就能优先处理高优先级的事件。不同的处理器可能有不同的中断优先级策略。

### 1.4.3 中断嵌套

与中断相关的另一个重要的问题是中断嵌套。当处理器正在处理某个中断的时候，如果有其他中断发生，那么就得仔细考虑如何处理新的中断。对于简单的处理器来说，可能本身并不支持中断嵌套，在中断响应阶段就把中断给关闭了。而当前大多数的处理器是支持中断嵌套的，方案基本是结合中断优先级的嵌套方式，原则是：

- 高优先级中断可以抢占低优先级中断
- 同级中断不可抢占（包括自身）
- 不能被立刻响应的中断会被悬挂(Pending)，等待高优先级中断退出后才执行

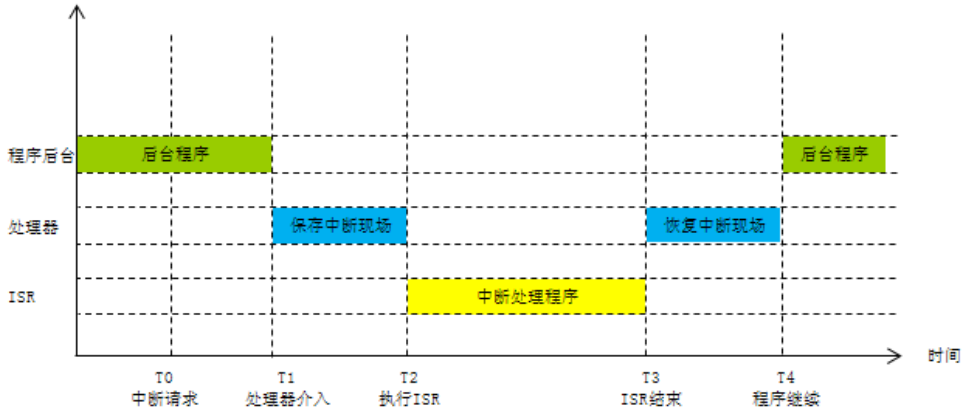
中断嵌套行为的如下图所示




### 1.4.4 中断时序

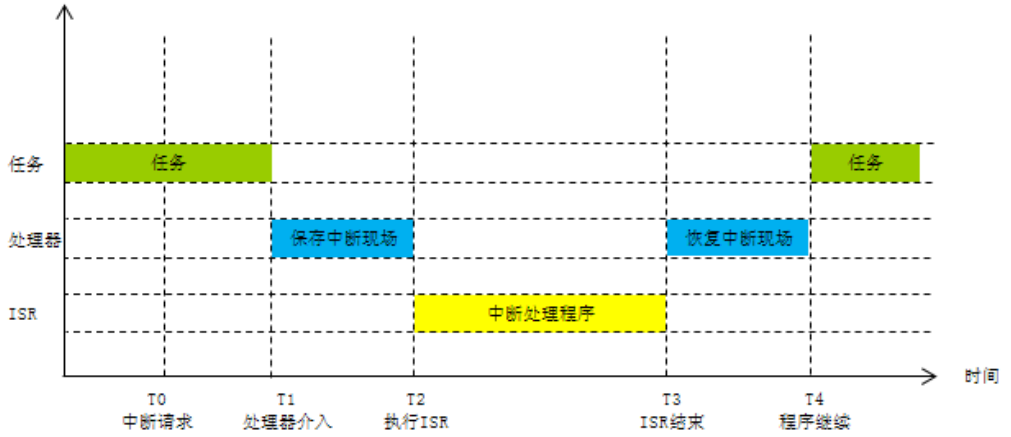
在前面我们介绍了中断的基本流程，结合前面章节介绍的嵌入式软件结构和实时内核的抢占方式，我们来具体分析一下中断的时序的问题。

首先是前后台系统的中断时序，如下图所示：



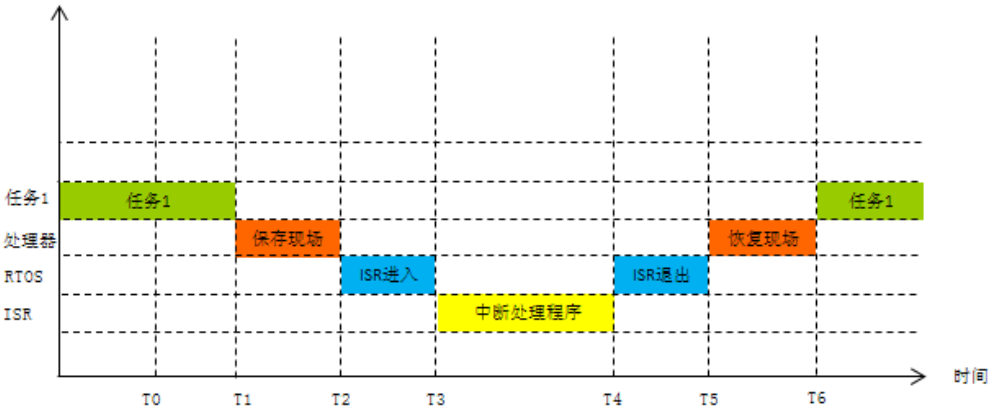

可以看出前后台系统的中断时序很简单，这也说明了前后台的特点：实时响应事件，轮询处理事务。

下图演示了不可抢占内核的中断处理流程。




可以看出不可抢占多任务系统的中断时序和前后台系统的中断时序很相似，需要注意的是前后台系统只有一个后台任务，所有中断打断的都是这个唯一的任务。而不可抢占内核的多任务系统则有很多任务，每个任务都可能被中断打断。

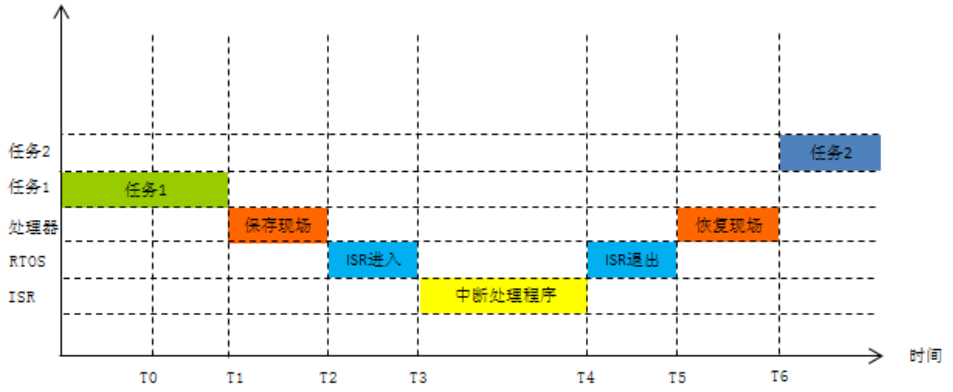
下图演示了可抢占内核的中断处理流程：




如上图所示

- T0 时刻，用户任务正在执行，此时有外部设备产生中断请求

- T1 时刻，处理器开始处理外部中断，保存中断现场
- T2 时刻，RTOS 介入，进行中断预处理，记录当前中断嵌套深度，查找用户登记的中断处理程序。
- T3 时刻，处理器开始执行中断处理器函数
- **T4 时刻，RTOS 介入，进行中断退出处理，如果当前中断是所有嵌套中断的最后一个，则进行任务调度处理，此时可能发生任务切换的准备**
- T5 时刻，处理器开始恢复中断现场，此时恢复的现场未必是 T1 时刻保存的任务现场，而是在 T4 时刻选择的最高就绪优先级的任务
- T6 时刻，如果没有进行任务调度和切换，则被中断的用户任务继续执行，如上图。如果进行了任务切换，则如下图所示。




这里需要解释一下为什么 RTOS 介入可抢占内核的中断流程的问题。因为在可抢占内核中，当从中断返回时会处理任务抢占的问题，所以在每次中断进入时都要对中断嵌套计数做加法；在每次中断退出时都要对嵌套计数做减法，然后再继续检查中断嵌套计数是否为 0，如果是，则说明中断彻底退出了，需要进行任务调度的处理。这些操作都是由 RTOS 完成的。而不可抢占内核则没有这些事情要做，所以不必对中断流程做特殊处理，基本和前后台系统的中断流程一致。

## 1.5 Trochili　内核介绍

TROCHILI 是一个简洁精炼的适用于嵌入式领域的实时操作系统内核。支持多线程、多优先级、抢占式调度。内核代码大部分由 C 语言开发，代码结构清晰，可维护性好。TROCHILI 取蜂鸟之意，意味着体积小巧、动作灵敏。

**多线程机制特点**

- 灵活的线程调度机制，支持优先级抢占式调度。
- 可配置的多优先级数目，默认支持 32 优先级(可扩展) 。
- 不同优先级线程采用优先级调度，相同优先级线程间采用时间片调度。
- 可动态调整线程优先级和时间片。
- 支持多种线程管理功能(线程激活和休眠、线程挂起和唤醒、线程延时、修改线程优先级或时间片等) 。

**IPC 机制特点**

- 支持常见 IPC 机制(信号量、互斥量、消息队列、邮箱、事件标记等) ，充分总结以上各种 IPC 机制的共性和特性，基于通用 IPC 控制结构和操作流程，做了完整、规则并且简洁的实现。
- 可配置的 IPC 调度机制，支持 FIFO 和优先级两种方式的线程等待队列(如在消息队列里，可以配置普通消息操作按照 FIFO 策略来处理，而紧急消息可采用优先级机制)。
- 消息队列、邮箱支持紧急消息传递。
- 互斥量支持优先级天花板协议。
- 信号量、消息队列、邮箱、事件标记支持 ISR 代码调用操作。

**定时器机制特点**

- 支持用户定时器、线程延时定时器和 IPC 访问的时限定时器。
- 内核内置定时器处理线程，可以配置选择在 ISR 或者线程里完成具体的用户定时操作。

**ISR 机制特点**

- 提供灵活的中断处理机制，支持中断下半部处理机制。
- 中断请求可以在 ISR 里完成，也可以内核内置中断处理线程完成，更可以在用户自己提供的中断处理线程完成。

**内存管理特点**

- 提供基于链表的动态内存块管理。
- 提供基于 BUDDY 算法的动态内存管理模块。

**其他特点**

- 提供完整简洁的 API 功能接口。
- 代码结构完整，层次清晰，易于理解维护，强调工程性。
- 资料完整，有大量的实验指导资料和正式出版的图书。
- 代码注释完备，十分详尽的中文注释，可以说是注释最详细的内核代码。

**参数统计**

TROCHILI 非常适合各种资源紧张的 MCU 系统。基于 GD32F150 的处理器(72M 主频、8K SRAM)，采用全功能默认配置，移植后的内核参数如下:

- ROM 代码 小于 10KB
- RAM 代码 约 2.8KB
- 各个 API 调用时间分布在 3~15us

**最新代码会在 Github 上持续更新，代码仓库地址 https://github.com/liuxuming/Trochili**

Trochili 内核是作者业余时间学习的成果。它基本实现了上面介绍的 RTOS 的知识点，并且有自己的独特实现。目前它还不是一个成熟的商业代码，但作为学习和理解 RTOS 确实不可多得的好资料。无论从代码结构还是文档注释上，它更像作者的学习笔记，提供了大量的图表和大段的中文注释，这些是在别的内核代码中看不到。

Trochili 内核主要实现了这样几个功能

- 线程管理和调度
- 信号量和互斥量
- 邮箱和消息队列
- 定时器和其守护线程
- 内核移植内核启动
- 内核全局功能函数
- 调试选项

在本书的以下各章作者将按照原理、设计、实现和应用的思路逐步介绍 Trochili 内核的各个功能模块。多个角度向读者充分展示 RTOS 的核心和细节。希望读者不仅能从中理解 RTOS 的概念和使用，更希望读者能有自己的想法和改进方案，然后付诸行动。对于真正对技术感兴趣的读者来说，下面各章的内容一定不会让你失望。