第 3 章

FreeRTOS 任务管理

本章深入探讨了 FreeRTOS 的任务管理机制，全面覆盖了多任务运行的核心机制、任务状态的演变、优先级的分配及其调度影响、空闲任务的角色与实现以及基础时钟与嘀嗒信号在任务切换中的关键作用。同时，本章还详细介绍了 FreeRTOS 的任务调度方法，特别是时间片抢占式调度的原理与应用，为读者提供了深入的理解。

重点内容：

(1) 任务管理概述。

① 多任务运行基本机制：深入解析多任务系统的基本运行机制，为读者提供清晰的概念框架。

② 任务的状态：详细介绍任务从创建到删除的各个状态，帮助读者理解任务的生命周期。

③ 任务的优先级：深入描述任务优先级的分配原则及其对调度的影响，助力读者掌握优先级调度的精髓。

④ 空闲任务：深入探讨空闲任务的作用和实现方式，揭示其在系统中的重要地位。

⑤ 基础时钟与嘀嗒信号：详细解释嘀嗒信号对任务切换的重要性，帮助读者理解时间管理在任务调度中的关键作用。

(2) FreeRTOS 的任务调度。

① 任务调度方法概述：总结并对比不同的任务调度算法，为读者提供全面的调度方法视野。

② 使用时间片的抢占式调度方法：具体阐述时间片抢占式调度的原理、应用场景及优势，助力读者掌握这一核心调度技术。

(3) FreeRTOS 任务管理：全面介绍 FreeRTOS 中任务的创建、删除和管理方法，为读者提供实用的任务管理指南。

(4) 任务管理相关函数：详细列出并解释与任务管理相关的 API 函数，助力读者快速上手 FreeRTOS 任务管理开发。

(5) FreeRTOS 任务的设计要点：深入探讨在设计 FreeRTOS 任务时需要考虑的关键因素，帮助读者规避常见设计陷阱。

(6) FreeRTOS 任务管理应用实例：通过具体实例演示如何在实际项目中实现任务管理，助力读者将理论知识转换为实践能力。

3.1 任务管理概述

在 FreeRTOS 中，任务(task)是基本的执行单位，类似于传统操作系统中的线程。每个任务相当于一个独立运行的函数，可以独立配置各种属性，如优先级、堆栈大小等。任务之间通过时间共享和优先级调度机制进行切换。

关键特性和概念包括：

(1) 独立执行：每个任务都是一个独立的程序，它们可以分别执行不同的功能。

(2) 优先级：任务可以被赋予不同的优先级，调度器根据优先级来决定任务的执行顺序和时间。

(3) 调度：FreeRTOS 包含一个实时调度器，根据任务的优先级和可运行状态来调度任务。

(4) 堆栈：每个任务有自己独立的堆栈空间，用来存储局部变量、函数调用返回地址以及上下文信息。

(5) 生命周期：任务在创建后进入就绪(ready)状态，调度器选择就绪状态的高优先级任务执行。当任务在等待事件(如延时、信号量、消息队列)时，将进入阻塞(Blocked)状态，事件发生后重新进入就绪状态。

3.1.1 多任务运行基本机制

在 FreeRTOS 中，任务就是实现某种功能的一个函数。通常，任务函数内部包含一个无限循环结构。在任务函数中不允许使用 return 语句退出，如果需要结束任务，可以跳出循环并调用 vTaskDelete 函数自我删除任务，也可以在其他任务中调用 vTaskDelete 函数来删除该任务。

用户可以在 FreeRTOS 中创建多个任务。每个任务需要分配一个栈(Stack)空间以及一个任务控制块(Task Control Block，TCB)空间。此外，每个任务还需要设置一个优先级，优先级的数字越小表示优先级越低。

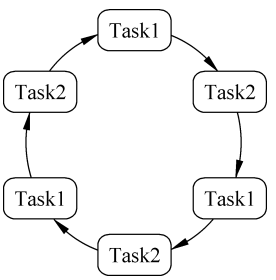

图 3-1 最简单的基于时间片的多任务运行原理

在单核处理器上，任意时刻只能有一个任务占用 CPU 并运行。然而，在 RTOS 系统中运行多个任务时，看起来像是有多个任务在同时运行。其实，这归功于 RTOS 的任务调度机制，使得多个任务能够分时共享 CPU 资源，从而实现所谓的“同时运行”。

最简单的基于时间片的多任务运行原理如图 3-1 所示。

假设系统中只有两个任务：Task1 和 Task2，并且它们具有相同的优先级。可以将CPU 时间想象成一个圆周，类似于钟表的一圈。RTOS 将 CPU 时间分成基本的时间片(time slice)，例如，FreeRTOS 默认的时间片长度是 1ms，也就是 SysTick 定时器的定时周期。在一个时间片内，只有一个任务占用 CPU 并执行。

假设当前运行的任务是 Task1。当一个时间片结束时(SysTick 定时器发生中断时)，RTOS 会进行任务调度。由于 Task1 和 Task2 具有相同的优先级，RTOS 会将 CPU 使用权交给 Task2。当 Task1 交出 CPU 使用权时，它会将当前 CPU 的状态(包括各个核心寄存器的值)保存到自己的栈空间中。而当 Task2 获取 CPU 使用权时，它会从自己的栈空间中恢复 CPU 状态，从上次运行的状态继续执行。

基于时间片的多任务调度就是通过这种方式来控制多个相同优先级的任务，实现 CPU 的分时复用，从而实现多任务运行。由于时间片的长度很短(默认是 1ms)，任务切换的速度非常快，因此在程序运行时，给用户的感觉是多个任务在同时运行。

当多个任务的优先级不同时，FreeRTOS 会使用基于优先级的抢占式任务调度方法。在这种情况下，每个任务获得的 CPU 使用时间长度可以是不一样的。

3.1.2　任务的状态

由单核 CPU 的多任务运行机制可知，任何时刻，只能有一个任务占用 CPU 并运行，这个任务的状态称为运行(running)状态，其他未占用 CPU 的任务的状态都可称为非运行(norunning)状态。非运行状态又可以细分为 3 个状态，任务的各个状态以及状态之间的转换如图 3-2 所示。

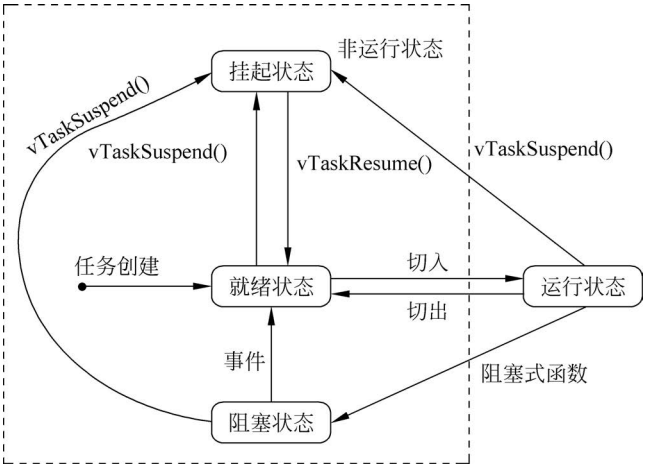

图 3-2　任务的状态以及状态之间的转换

FreeRTOS 的任务调度有两种方式：抢占式(pre-emptive)和合作式(co-operative)。通常使用基于任务优先级的抢占式任务调度方法。关于任务调度的各种方式将在后文详细介绍。这里以抢占式任务调度为例，解释图 3-2 所示的原理。

1. 就绪状态

任务被创建后首先处于就绪状态。每次基础时钟中断时，FreeRTOS 的任务调度器会进行一次任务调度。根据抢占式任务调度的特点，任务调度的结果有以下几种情况。

(1) 如果当前没有其他正在运行的任务，处于就绪状态的任务将进入运行状态。

(2) 如果就绪任务的优先级高于或等于当前运行任务的优先级，处于就绪状态的任务将进入运行状态。

(3) 如果就绪任务的优先级低于当前运行任务的优先级，处于就绪状态的任务将无法获得 CPU 使用权，继续处于就绪状态。

就绪任务获取 CPU 使用权并进入运行状态的过程称为“切入”(switch in)。相应地，处于运行状态的任务被调度器调度为就绪状态的过程称为“切出”(switch out)。

2. 运行状态

在单核处理器上，占用 CPU 并执行的任务处于运行状态。高优先级任务如果一直运行，将持续占用 CPU，导致低优先级的就绪任务无法获得 CPU 使用权。因此，运行状态的任务应在空闲时主动让出 CPU。

处于运行状态的任务可以通过两种方式主动让出 CPU 使用权。

(1) 执行函数 vTaskSuspend 进入挂起状态。

(2) 执行阻塞式函数进入阻塞状态。

这两种状态都是非运行状态，使得任务调度器可以选取其他就绪状态的任务进入运行状态。

3. 阻塞状态

阻塞(blocked)状态表示任务暂时让出 CPU 使用权，处于等待状态。运行状态的任务可以通过调用以下两类函数进入阻塞状态。

(1) 时间延迟函数：如 vTaskDelay 或 vTaskDelayUntil。任务调用这些函数后会进入阻塞状态，并延迟指定的时间。延迟时间到后，任务会重新进入就绪状态，然后通过任务调度再次进入运行状态。

(2) 事件请求函数：用于进程间通信，例如请求信号量的函数 xSemaphoreTake。当任务执行 xSemaphoreTake 后会进入阻塞状态。如果其他任务释放了信号量或等待的超时时间到，任务将从阻塞状态进入就绪状态。

在运行状态的任务中调用 vTaskSuspend 可以将一个处于阻塞状态的任务转入挂起状态。

4. 挂起状态

挂起(suspended)状态表示任务暂停执行，不参与调度器的调度。其他三种状态的任务都可以通过调用 vTaskSuspend 进入挂起状态。处于挂起状态的任务不能自动退出挂起状态，必须在其他任务中调用 vTaskResume，才能将挂起任务变为就绪状态。

3.1.3 任务的优先级

在 FreeRTOS 中，每个任务都必须设置一个优先级。总的优先级个数由文件

FreeRTOSConfig.h 中的宏 configMAX_PRIORITIES 定义，默认值为 56。优先级数字越小，优先级越低。因此，最低优先级是 0，最高优先级是 configMAX_PRIORITIES－1。在创建任务时，用户必须为任务设置初始优先级，在任务运行中还可以动态修改其优先级。同时，多个任务可以拥有相同的优先级。

此外，参数 configMAX_PRIORITIES 可设置的最大值，以及调度器选择哪个就绪任务进入运行状态，还与参数 configUSE_PORT_OPTIMISED_TASK_SELECTION 的取值有关。根据这个参数的值，任务调度器的实现方法有两种。

1. 通用方法

如果 configUSE_PORT_OPTIMISED_TASK_SELECTION 设置为 0，则使用通用方法。这种方法是用 C 语言实现的，可以在所有 FreeRTOS 移植版本上使用，并且 configMAX_PRIORITIES 的最大值没有限制。

2. 架构优化方法

如果 configUSE_PORT_OPTIMISED_TASK_SELECTION 设置为 1，则使用架构优化方法。这种方法的部分代码是用汇编语言编写的，因此运行速度比通用方法更快。不过，使用这种方法时，configMAX_PRIORITIES 的最大值不能超过 32。此外，在使用 Cortex-M0 架构或 CMSIS-RTOS V2 接口时，不能使用架构优化方法。

本书使用的开发板搭载的是 STM32F407ZGT6 处理器，而 FreeRTOS 的接口通常设置为 CMSIS-RTOS V2。因此，在 STM32CubeMX 中，参数 USE_PORT_OPTIMISED_TASK_SELECTION 是不可修改的，总是被禁用(disabled)。

3.1.4　空闲任务

在 main 函数中，当调用 osKernelStart 启动 FreeRTOS 的任务调度器时，FreeRTOS 会自动创建一个优先级为 0(最低优先级)的空闲任务(idle task)。

在 FreeRTOS 中，任何时候都需要有一个任务占用 CPU 并处于运行状态。如果用户创建的任务都不处于运行状态，例如，都处于阻塞状态，空闲任务就会占用 CPU 并进入运行状态。

空闲任务在系统中非常重要，并且有多种用途。与空闲任务相关的配置参数如下。

(1) configUSE_IDLE_HOOK：该参数决定是否使用空闲任务钩子函数。如果配置为 1，则可以在系统空闲时利用空闲任务钩子函数进行一些处理。

(2) configIDLE_SHOULD_YIELD：该参数决定空闲任务是否对同等优先级的用户任务主动让出 CPU 使用权，这会影响任务调度的结果。

(3) configUSE_TICKLESS_IDLE：该参数决定是否使用 tickless 低功耗模式。如果设置为 1，则可以实现系统的低功耗。

这样，空闲任务不仅保障系统在所有任务都处于阻塞状态时的正常运行，还可以通过配置增强系统的功能和性能。

3.1.5 基础时钟与嘀嗒信号

在 FreeRTOS 中，系统自动使用 SysTick 定时器作为基础时钟。SysTick 定时器只有定时中断功能，其中断频率由配置参数 configTICK_RATE_HZ 指定，默认值是 1000，这意味着每 1ms 产生一次中断。

在 FreeRTOS 中，有一个全局变量 xTickCount。每当 SysTick 中断发生时，这个变量就加 1，也就是每 1ms 增加一次。所谓的 FreeRTOS 嘀嗒信号，就是指全局变量 xTickCount 的变化，因此嘀嗒信号的变化周期为 1ms。通过调用函数 xTaskGetTickCount 可以获取 xTickCount 的当前值。延时函数 vTaskDelay 和 vTaskDelayUntil 就是通过嘀嗒信号实现毫秒级延时的。

除了产生嘀嗒信号，SysTick 定时器中断还用于产生任务切换的请求。这意味着，每次 SysTick 中断发生时，FreeRTOS 都有机会检查是否需要进行任务切换，以实现任务调度。

通过这些机制，SysTick 定时器在 FreeRTOS 中起到了时钟基础和任务调度的重要作用。

3.2 FreeRTOS 的任务调度

本节讲述 FreeRTOS 的任务调度机制，包括任务调度方法的整体概述和具体的时间片抢占式调度方法。讲述如何通过分配时间片来实现多个任务的并发执行，以确保系统的实时性和效率。通过该调度机制，FreeRTOS 能够有效地管理和切换多个任务，优化系统资源的使用。

3.2.1 任务调度方法概述

FreeRTOS 有两种任务调度算法，基于优先级的抢占式(pre-emptive)调度算法和合作式(co-operative)调度算法。其中，抢占式调度算法可以使用时间片，也可以不使用时间片。通过参数的设置，用户可以选择具体的调度算法。FreeRTOS 的任务调度方法有 3 种，其对应的参数名称、取值及特点如表 3-1 所示。

表 3-1 FreeRTOS 的任务调度方法

调度方式	宏定义参数	值	特点
抢占式（使用时间片）	configUSE_PREEMPTION	1	基于优先级的抢占式任务调度，同优先级任务使用时间片轮流进入运行状态(默认模式)
	configUSE_TIME_SLICING	1	
抢占式（不使用时间片）	configUSE_PREEMPTION	1	基于优先级的抢占式任务调度，同优先级任务不使用时间片调度
	configUSE_TIME_SLICING	0	
合作式	configUSE_PREEMPTION	0	只有当运行状态的任务进入阻塞状态，或显式地调用要求执行任务调度的函数 taskYIELD，FreeRTOS 才会发生任务调度，选择就绪状态的高优先级任务进入运行状态
	configUSE_TIME_SLICING	任意	

在 FreeRTOS 中，默认的是使用带有时间片的抢占式任务调度方法。在 STM32CubeMX 中，用户不能设置参数 configUSE_TIME_SLICING，其默认值为 1。

3.2.2　使用时间片的抢占式调度方法

抢占式任务调度方法，是 FreeRTOS 主动进行任务调度，分为使用时间片和不使用时间片两种情况。

FreeRTOS 基础时钟的一个定时周期称为一个时间片（time slice），FreeRTOS 的基础时钟是 SysTick 定时器。基础时钟的定时周期由参数 configTICK_RATE_HZ 决定，默认值为 1000Hz，所以时间片长度为 1ms。当使用时间片时，在基础时钟的每次中断里，系统会要求进行一次上下文切换（context switching）。文件 port.c 中的函数 xPortSysTickHandler 就是 SysTick 定时中断的处理函数，其代码如下：

```
void xPortSysTickHandler(void)
 {/* SysTick 中断的抢占优先级是 15,优先级最低 */
portDISABLE_INTERRUPTS();
//禁用所有中断
 {
If(xTaskIncrementTick()! = pdFALSE) //增加 RTOS 嘀嗒计数器的值
 {
/* 将 PendSV 中断的挂起标志位置位,申请进行上下文切换,在 PendSV 中断里处理上下文切换 */
portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
}
portENABLE_INTERRUPTS();
//使能中断
```

这个函数的功能就是将 PendSV（pendable request for system service，可挂起的系统服务请求）中断的挂起标志位置位，也就是发起上下文切换的请求，而进行上下文切换是在 PendSV 的中断服务函数里完成的。文件 port.c 中的函数 xPortPendSVHandler 是 FreeRTOS 的 PendSV 中断服务函数，其功能就是根据任务调度计算的结果，选择下一个任务进入运行状态。这个函数的代码是用汇编语言写的，这里就不展示和分析其源代码了。

在 STM32CubeMX 中，一个项目使用了 FreeRTOS 后，会自动对 NVIC 作一些设置。系统自动将优先级分组方案设置为 4 位全部用于抢占优先级，SysTick 和 PendSV 中断的抢占优先级都是 15，也就是最低优先级。FreeRTOS 在最低优先级的 PendSV 的中断服务函数里进行上下文切换，所以 FreeRTOS 的任务切换的优先级总是低于系统中断的优先级。

使用时间片的抢占式调度方法的特点如下。

（1）在基础时钟每个中断里发起一次任务调度请求。

（2）在 PendSV 中断服务函数里进行上下文切换。

（3）在上下文切换时，高优先级的就绪任务获得 CPU（Central Processing Unit，中央处理器）的使用权。

（4）若多个就绪状态的任务的优先级相同，则将轮流获得 CPU 的使用权。

图 3-3 所示的是使用带时间片的抢占式任务调度方法时，3 个任务运行的时序图。图中横轴是时间轴，纵轴是系统中的任务。垂直方向的虚线表示发生任务切换的时间点，水平为的实心矩形表示任务占据 CPU 处于运行状态的时间段，水平方向的虚线表示任务处于就绪的时间段，水平方向的空白段表示任务处于阻塞状态或挂起状态的时间段。

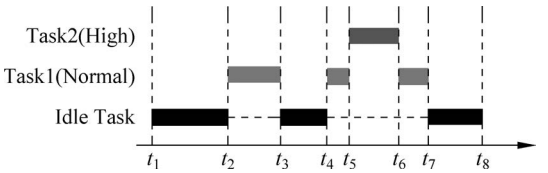

图 3-3 任务运行时序图(带时间片的抢占式任务调度方法)

图 3-3 可以说明带时间片的抢占式任务调度方法的特点。假设 Task2 具有高优先级，Task1 具有正常优先级，且这两个任务的优先级都高于空闲任务的优先级。从这个时序图可以看到这 3 个任务的运行和任务切换的过程。

(1) t_1 时刻开始是空闲任务在运行，这时候系统里没有其他任务处于就绪状态。

(2) 在 t_2 时刻进行调度时，Task1 抢占 CPU 开始运行，因为 Task1 的优先级高于空闲任务。

(3) 在 t_3 时刻，Task1 进入阻塞状态，让出了 CPU 的使用权，空闲任务又进入运行状态。

(4) 在 t_4 时刻，Task1 又进入运行状态。

(5) 在 t_5 时刻，更高优先级的 Task2 抢占了 CPU 开始运行，Task1 进入就绪状态。

(6) 在 t_6 时刻，Task2 运行后进入阻塞状态，让出 CPU 使用权，Task1 从就绪状态变为运行状态。

(7) 在 t_7 时刻，Task1 进入阻塞状态，主动让出 CPU 使用权，空闲任务又进入运行状态。从图 3-3 的多任务运行过程可以看出，在低优先级任务运行时，高优先级的任务能抢占获得 CPU 的使用权。在没有其他用户任务运行时，空闲任务处于运行状态，否则空闲任务处于就绪状态。

当多个就绪状态的任务优先级相同时，它们将轮流获得 CPU 的使用权，每个任务占用 CPU 运行 1 个时间片的时间。如果就绪任务的优先级与空闲任务的优先级都相同，参数 configIDLE_SHOULD_YIELD 就会影响任务调度的结果。

(1) 如果 configIDLE_SHOULD_YIELD 设置为 0，表示空闲任务不会主动让出 CPU 的使用权，空闲任务与其他优先级为 0 的就绪任务轮流使用 CPU。

(2) 如果 configIDLE_SHOULD_YIELD 设置为 1，表示空闲任务会主动让出 CPU 的使用权，空闲任务不会占用 CPU。

参数 configIDLE_SHOULD_YIELD 的默认值为 1。设计用户任务时，用户任务的优先级一般要高于空闲任务。

3.3　任务管理的应用场合

在 FreeRTOS 中，任务管理是操作系统核心功能之一，其应用场景十分广泛，可以用来处理并发操作、时间关键的任务和异步事件等。以下是一些典型的任务管理应用场合。

(1) 并发处理：在需要并行执行多个任务的场景下，例如处理多种传感器数据、同时进行通信和数据处理等，任务管理可以有效将这些任务分配到不同的任务中，由 FreeRTOS 负责调度。

(2) 时间控制：在需要时间精度和及时响应的系统，如机器人控制、工业自动化、医疗设备等，任务管理可以确保高优先级的任务及时执行，响应外部事件。

(3) 界面更新：在图形或文本用户界面中，不同的 UI 组件可以独立运行各自的任务，进行数据刷新、用户输入处理等，确保界面的流畅性和响应速度。

(4) 通信及协议栈实现：在网络通信、无线通信等场景中，各种协议栈(如 TCP/IP、Bluetooth 等)通常会运行在独立的任务中，处理数据包的接收和发送、连接维护等功能。

(5) 线程数据处理：在高性能需求的场景，如音视频处理、图像识别、数据采集等，不同的数据处理步骤可以拆分到不同的任务中，利用多核处理器的能力进行并行处理，提高系统性能。

(6) 异步事件处理：在需要处理异步事件的场景下，如中断服务程序(ISR)触发的事件，可以将复杂的处理逻辑移至任务中执行，使 ISR 保持轻量级，提高系统响应效率。

(7) 系统资源管理：在系统资源(如内存、外设、文件系统等)管理中，不同资源的管理任务可以独立运行，按需调度，避免资源竞争。

(8) 故障隔离与恢复：在高可靠性要求的系统中，不同的任务可以独立运行，使得单个任务的崩溃不会影响整个系统的运行，同时通过看门狗任务进行监控，发生故障时能快速恢复。

FreeRTOS 提供了一系列 API 用于任务管理，包括任务创建、删除、调度、优先级设置、延时等。

以下是一个简单的任务管理示例。

```
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>

// 任务 1 函数
void vTask1(void *pvParameters) {
    for (;;) {
        printf("Task 1 is running\n");
        vTaskDelay(pdMS_TO_TICKS(1000));          // 延时 1s
    }
}
```

```
// 任务 2 函数
void vTask2(void * pvParameters) {
    for (;;) {
        printf("Task 2 is running\n");
        vTaskDelay(pdMS_TO_TICKS(2000));          // 延时 2s
    }
}

// 主函数
int main(void) {
    // 创建任务 1
    xTaskCreate(vTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

    // 创建任务 2
    xTaskCreate(vTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

    // 启动调度器
    vTaskStartScheduler();

    // 如果 vTaskStartScheduler()调用失败,将执行此段代码
    for (;;) {
    }

    return 0;
}
```

在这个示例中,两个任务分别以不同的时间间隔打印消息并延时运行。这展示了FreeRTOS如何通过任务管理来实现并发操作。通过合理地划分任务和设定优先级,可以根据具体需求定制嵌入式系统的行为和性能。

3.4 任务管理相关函数

在FreeRTOS中,任务的管理主要包括任务的创建、删除、挂起、恢复等操作,还包括任务调度器的启动、挂起与恢复,以及使任务进入阻塞状态的延迟函数等。

FreeRTOS中任务管理相关的函数都在文件task.h中定义,在文件tasks.c中实现。在CMSIS-RTOS中还有一些函数,对FreeRTOS的函数进行了封装,也就是调用相应的FreeRTOS函数实现相同的功能,这些标准接口函数的定义在文件cmsis_os.h和cmsis_os2.h中。CubeMX生成的代码一般使用CMSIS-RTOS标准接口函数,在用户自己编写的程序中,一般直接使用FreeRTOS的函数。

任务管理常用的一些函数及其功能描述如表3-2所示。这里只列出了函数名,省略了输入/输出参数。如需了解每个函数的参数定义和功能说明,可以查看其源代码,或参考FreeRTOS官网的在线文档,或查阅FreeRTOS参考手册文档The FreeRTOS Reference Manual。

表 3-2　任务管理常用的一些函数及其功能描述

分　组	FreeRTOS 函数	函 数 功 能
任务管理	xTaskCreate	创建一个任务，动态分配内存
	xTaskCreateStatic	创建一个任务，静态分配内存
	vTaskDelete	删除当前任务或另一个任务
	vTaskSuspend	挂起当前任务或另一个任务
	vTaskResume	恢复另一个挂起任务的运行
	xTaskResumeFromISR	在中断服务程序中，用于恢复被挂起的任务
调度器管理	vTaskStartScheduler	开启任务调度器
	vTaskSuspendAll	挂起调度器，但不禁止中断。调度器被挂起后不会再进行上下文切换
	vTaskResumeAll	恢复调度器的执行，但是不会解除用函数 vTaskSuspend 单独挂起的任务的挂起状态
	vTaskStepTick	用于在 tickless 低功耗模式时补足系统时钟计数节拍
延时与调度	vTaskDelay	当前任务延时指定节拍数，并进入阻塞状态
	vTaskDelayUntil	当前任务延时到指定的时间，并进入阻塞状态，用于精确延时的周期性任务
	xTaskGetTickCount	返回基础时钟定时器的当前计数值
	xTaskAbortDelay	终止另一个任务的延时，使其立刻退出阻塞状态
	taskYIELD	请求进行一次上下文切换，用于合作式任务调度

1. 任务挂起函数

(1) void vTaskSuspend(TaskHandle_t xTaskToSuspend)。

void vTaskSuspend(TaskHandle_t xTaskToSuspend)挂起指定任务。被挂起的任务绝不会得到 CPU 的使用权，不管该任务具有什么优先级。

任务可以通过调用 void vTaskSuspend(TaskHandle_t xTaskToSuspend)函数将处于任何状态的任务挂起，被挂起的任务得不到 CPU 的使用权，也不会参与调度，它相对于调度器而言是不可见的，除非它从挂起态中解除。

vTaskSuspend 是 FreeRTOS 提供的一个函数，用于挂起(暂停)一个指定的任务。当一个任务被挂起后，除非再次调用 vTaskResume 将其恢复，否则该任务不会被调度运行。这个函数对于需要控制任务的执行顺序或者临时暂停低优先级任务的情况非常有用。

① 函数原型。

```
void vTaskSuspend(TaskHandle_t xTaskToSuspend);
```

参数 xTaskToSuspend：要挂起的任务的句柄。如果传递 NULL 给这个参数，则挂起调用该函数的任务自身。

② 应用实例。

下面是一个简单的应用实例，演示如何使用 vTaskSuspend 函数来挂起任务。

假设有两个任务：Task A 和 Task B。在这个示例中，Task A 将在启动后的 5s 内挂起 Task B，然后 5s 后恢复 Task B。

```
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
// 任务句柄
TaskHandle_t xTaskBHandle = NULL;

void vTaskA(void *pvParameters) {
    // 延迟 5s
    vTaskDelay(pdMS_TO_TICKS(5000));
    printf("Suspending Task B\n");
    // 挂起 Task B
    vTaskSuspend(xTaskBHandle);

    // 延迟 5s
    vTaskDelay(pdMS_TO_TICKS(5000));
    printf("Resuming Task B\n");
    // 恢复 Task B
    vTaskResume(xTaskBHandle);

    // 删除任务自身
    vTaskDelete(NULL);
}

void vTaskB(void *pvParameters) {
    while (1) {
        printf("Task B is running\n");
        // 每秒打印一次
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

int main() {
    // 创建 Task A
    xTaskCreate(vTaskA, "Task A", configMINIMAL_STACK_SIZE, NULL, 2, NULL);

    // 创建 Task B 并保存其句柄,以便后续用于挂起和恢复
    xTaskCreate(vTaskB, "Task B", configMINIMAL_STACK_SIZE, NULL, 1, &xTaskBHandle);

    // 启动调度器
    vTaskStartScheduler();

    // 如果系统一切正常,调度器将不再返回此处
    for (;;);
    return 0;
}
```

③ 函数解释。

任务句柄：定义了一个 xTaskBHandle 用于保存 Task B 的句柄。

Task A：

首先，延迟 5s，以让 Task B 有足够时间运行。

然后，打印出 Suspending Task B 并调用 vTaskSuspend(xTaskBHandle)挂起 Task B。

再次延迟 5s，并调用 vTaskResume(xTaskBHandle)恢复 Task B。

最后，Task A 删除自己。

Task B：

使用一个无限循环，每秒钟打印一次 Task B is running。

在这个实例中，Task A 会在启动后的 5s 内挂起 Task B，然后在接下来的 5s 后恢复 Task B。这个简单的示例展示了如何使用 vTaskSuspend 和 vTaskResume 来控制任务的执行。

(2) vTaskSuspendAll。

这个函数就是比较有意思的，将所有的任务都挂起，其实源码很简单，也很有意思，不管三七二十一将调度器锁定，并且这个函数是可以进行嵌套的，即挂起所有任务就是挂起任务调度器。调度器被挂起后则不能进行上下文切换，但是中断还是使能的。当调度器被挂起的时候，如果有中断需要进行上下文切换，那么这个任务将会被挂起，在调度器恢复之后才执行切换任务。调度器恢复可调用 xTaskResumeAll 函数，调用了多少次的 vTaskSuspendAll 就要调用多少次 xTaskResumeAll 进行恢复。

vTaskSuspendAll()源码如下：

```
void vTaskSuspendAll(void)
 {
   ++uxSchedulerSuspended; (1)
  }
```

2. 任务恢复函数

(1) void vTaskResume(TaskHandle_t xTaskToResume)。

既然有任务的挂起，那么当然一样有恢复，不然任务无法恢复。任务恢复就是让挂起的任务重新进入就绪状态，恢复的任务会保留挂起前的状态信息，在恢复的时候根据挂起时的状态继续运行。如果被恢复任务在所有就绪态任务中，处于最高优先级列表的第一位，那么系统将进行任务上下文的切换。

vTaskResume 是 FreeRTOS 提供的一个函数，用于恢复一个被挂起的任务。被挂起的任务在调用 vTaskResume 之前不会被调度运行。恢复后，该任务将继续执行。

① 函数原型。

```
void vTaskResume(TaskHandle_t xTaskToResume);
```

参数 xTaskToResume：要恢复的任务的句柄。

② 应用实例。

下面是一个简单的应用实例，演示如何使用 vTaskResume 函数来恢复任务。

假设有两个任务：Task A 和 Task B。在这个示例中，Task A 将在启动后的 5s 内挂起 Task B，然后 5s 后恢复 Task B。

```
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
// 任务句柄
TaskHandle_t xTaskBHandle = NULL;

void vTaskA(void * pvParameters) {
    // 延迟 5s
    vTaskDelay(pdMS_TO_TICKS(5000));
    printf("Suspending Task B\n");
    // 挂起 Task B
    vTaskSuspend(xTaskBHandle);

    // 延迟 5s
    vTaskDelay(pdMS_TO_TICKS(5000));
    printf("Resuming Task B\n");
    // 恢复 Task B
    vTaskResume(xTaskBHandle);

    // 删除任务自身
    vTaskDelete(NULL);
}

void vTaskB(void * pvParameters) {
    while (1) {
        printf("Task B is running\n");
        // 每秒打印一次
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

int main() {
    // 创建 Task A
    xTaskCreate(vTaskA, "Task A", configMINIMAL_STACK_SIZE, NULL, 2, NULL);

    // 创建 Task B 并保存其句柄,以便后续用于挂起和恢复
    xTaskCreate(vTaskB, "Task B", configMINIMAL_STACK_SIZE, NULL, 1, &xTaskBHandle);

    // 启动调度器
    vTaskStartScheduler();

    // 如果系统一切正常,调度器将不再返回此处
    for (;;);
    return 0;
}
```

③ 函数解释。

任务句柄：定义了一个 xTaskBHandle 用于保存 Task B 的句柄。

Task A：

首先，延迟 5s，以让任务 B 有足够时间运行。

然后，打印出 Suspending Task B 并调用 vTaskSuspend(xTaskBHandle)挂起 Task B。

再次延迟 5s，并打印 Resuming Task B。

最后，调用 vTaskResume(xTaskBHandle)恢复 Task B。

Task A 运行结束后删除自身。

Task B：

使用一个无限循环，每秒钟打印一次 Task B is running。

在这个示例中，Task A 会在启动后的 5s 内挂起 Task B，然后在接下来的 5s 后恢复 Task B。Task B 会在其未被挂起时每秒钟打印一次信息。本示例通过 vTaskSuspend 和 vTaskResume 清晰地控制了任务的执行和挂起，展示了这两个函数的实际用法。

(2) xTaskResumeFromISR(TaskHandle_t xTaskToResume)。

xTaskResumeFromISR(TaskHandle_t xTaskToResume)与 void vTaskResume(TaskHandle_t xTaskToResume)一样都是用于恢复被挂起的任务，不一样的是 xTaskResumeFromISR(TaskHandle_t xTaskToResume)专门用在中断服务程序中。无论通过调用几次 TaskSuspend 函数挂起任务，都只需调用一次 xTaskResumeFromISR(TaskHandle_t xTaskToResume)函数即可解挂。要想使用该函数，必须在 FreeRTOSConfig.h 中把 INCLUDE_vTaskSuspend 和 INCLUDE_vTaskResumeFromISR 都定义为 1 才有效。任务还没有处于挂起态的时候，调用 xTaskResumeFromISR 函数是没有任何意义的。

xTaskResumeFromISR 是 FreeRTOS 提供的一个函数，用于从中断服务程序(ISR)中恢复一个被挂起的任务。与 vTaskResume 不同的是，它专门用于在中断上下文中调用。

① 函数原型。

```
BaseType_t xTaskResumeFromISR(TaskHandle_t xTaskToResume);
```

参数 xTaskToResume：要恢复的任务的句柄。

返回值：如果任务被成功恢复并且此恢复操作导致需要进行上下文切换，则返回 pdTRUE，否则返回 pdFALSE。

② 应用实例。

下面是一个简单的应用实例，演示如何使用 xTaskResumeFromISR 函数来从中断中恢复任务。

假设有两个任务：Task A 和 Task B。Task B 将在系统启动后立即被挂起，并在一个模拟的定时器中断中恢复。

```
#include "FreeRTOS.h"
#include "task.h"
#include "timers.h"
#include <stdio.h>
```

```
// 任务句柄
TaskHandle_t xTaskBHandle = NULL;

// 模拟的定时器中断服务程序
void vTimerCallback(TimerHandle_t xTimer) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    printf("Interrupt: Resuming Task B from ISR\n");
    // 从 ISR 中恢复 Task B
    xTaskResumeFromISR(xTaskBHandle, &xHigherPriorityTaskWoken);

    // 如果需要进行上下文切换,执行 portYIELD_FROM_ISR 宏
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}

void vTaskA(void * pvParameters) {
    printf("Task A starting\n");

    // 挂起 Task B
    printf("Suspending Task B\n");
    vTaskSuspend(xTaskBHandle);

    // 创建并启动一个模拟的定时器,2s 后触发中断
    TimerHandle_t xTimer = xTimerCreate(
        "Timer",
        pdMS_TO_TICKS(2000),
        pdFALSE,
        (void * )0,
        vTimerCallback
    );

    if (xTimer != NULL) {
        xTimerStart(xTimer, 0);
    }

    // 删除任务自身
    vTaskDelete(NULL);
}

void vTaskB(void * pvParameters) {
    while (1) {
        printf("Task B is running\n");
        // 每秒打印一次
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

int main() {
    // 创建 Task A
    xTaskCreate(vTaskA, "Task A", configMINIMAL_STACK_SIZE, NULL, 2, NULL);
```

```
    // 创建 Task B 并保存其句柄,以便后续用于挂起和恢复
    xTaskCreate(vTaskB, "Task B", configMINIMAL_STACK_SIZE, NULL, 1, &xTaskBHandle);

    // 启动调度器
    vTaskStartScheduler();

    // 如果系统一切正常,调度器将不再返回此处
    for (;;);
    return 0;
}
```

③ 函数解释。

任务句柄：定义了一个 xTaskBHandle 用于保存 Task B 的句柄。

Task A：

首先打印 Task A starting 以表明任务开始运行。

然后打印 Suspending Task B 并调用 vTaskSuspend(xTaskBHandle)挂起 Task B。

创建并启动一个一次性的定时器,设定为 2s 后触发其回调函数 vTimerCallback。

Task A 运行结束后删除自身。

Task B：

使用一个无限循环,每秒钟打印一次 Task B is running。

定时器中断服务程序 vTimerCallback：

打印 Interrupt：Resuming Task B from ISR 以表明进入中断。

调用 xTaskResumeFromISR 恢复 Task B,并检查 xHigherPriorityTaskWoken 是否需要进行上下文切换。如果需要,则调用 portYIELD_FROM_ISR 宏进行上下文切换。

在这个实例中,Task A 会启动定时器并立即被删除。Task B 被创建后立即被挂起,然后在 2s 后的定时器中断中从 ISR 中恢复并继续运行。通过这个示例,可以清晰地看到如何在中断中使用 xTaskResumeFromISR。

(3) xTaskResumeAll。

前面讲解过 vTaskSuspendAll 函数,当调用 vTaskSuspendAll 函数将调度器挂起时,若要恢复调度器就需要调用 xTaskResumeAll 函数。

3. void vTaskDelete(TaskHandle_t xTaskToDelete)

void vTaskDelete(TaskHandle_t xTaskToDelete)用于删除一个任务。当一个任务删除另外一个任务时,形参为要删除任务创建时返回的任务句柄,如果是删除自身,则形参为 NULL。要想使用该函数必须在 reeRTOSConfig.h 中把 INCLUDE_vTaskDelete 定义为 1。

vTaskDelete 是 FreeRTOS 提供的一个函数,用于删除一个任务。删除任务后,任务的堆栈和控制块会被回收,系统资源得到释放。这个函数可以用于从任务内部删除自身,也可以用于删除其他任务。

① 函数原型。

```
void vTaskDelete(TaskHandle_t xTaskToDelete);
```

参数 TaskToDelete：要删除的任务的句柄。如果传递 NULL 给这个参数，则删除调用此函数的任务自身。

② 应用实例。

下面是一个简单的应用实例，演示如何使用 vTaskDelete 函数来删除任务。

假设有两个任务：Task A 和 Task B。Task A 启动后运行一定时间然后删除自己，同时启动 Task B，并在 Task B 中删除 Task A。

```
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>

// 任务句柄
TaskHandle_t xTaskAHandle = NULL;

void vTaskA(void * pvParameters) {
    while(1) {
        printf("Task A is running\n");
        // 模拟运行一段时间
        vTaskDelay(pdMS_TO_TICKS(1000));

        // Task A 决定删除自己
        printf("Task A is deleting itself\n");
        vTaskDelete(NULL);
    }
}

void vTaskB(void * pvParameters) {
    printf("Task B is running and will delete Task A\n");

    // 删除 Task A
    if (xTaskAHandle != NULL) {
        vTaskDelete(xTaskAHandle);
        printf("Task A has been deleted by Task B\n");
    }

    // Task B 自身循环打印
    while (1) {
        printf("Task B is running\n");
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

int main() {
    // 创建 Task A，并保存其句柄以便后续用于删除
    xTaskCreate(vTaskA, "Task A", configMINIMAL_STACK_SIZE, NULL, 2, &xTaskAHandle);

    // 创建 Task B
    xTaskCreate(vTaskB, "Task B", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
```

```
    // 启动调度器
    vTaskStartScheduler();

    // 如果系统一切正常,则调度器将不再返回此处
    for (;;);
    return 0;
}
```

③ 函数解释。

任务句柄：定义了一个 xTaskAHandle 用于保存 Task A 的句柄。

Task A：

持续打印 Task A is running 来模拟执行任务操作，并延迟 1s。

最后，打印 Task A is deleting itself 并调用 vTaskDelete(NULL)删除自身。

Taak B：

开始时打印 Task B is running and will delete Task A。

检查 xTaskAHandle 是否非空，如果是，则调用 vTaskDelete(xTaskAHandle)删除 Task A，并打印 Task A has been deleted by Task B。

进入一个无限循环，每 2s 打印一次 Task B is running。

主函数：

创建 Task A，并保存其句柄 xTaskAHandle。

创建 Task B。

启动调度器 vTaskStartScheduler。

在这个实例中，Task A 会在运行一段时间后删除自身。另外，Task B 会在启动时删除 Task A，并在自身内部进入一个无限循环继续运行。这展示了如何使用 vTaskDelete 函数来删除任务，不论是从任务内部还是从其他任务中删除。

4. 任务延时函数

(1) void vTaskDelay(const TickType_t xTicksToDelay)。

void vTaskDelay(const TickType_t xTicksToDelay)在任务中用得非常之多，每个任务都必须是死循环，并且是必须要有阻塞的情况，否则低优先级的任务就无法被运行了。要想使用 FreeRTOS 中的 vTaskDelay 函数必须在 FreeRTOSConfig. h 中把 INCLUDE_vTaskDelay 定义为 1 来使能。

vTaskDelay 用于阻塞延时，调用该函数后，任务将进入阻塞状态，进入阻塞态的任务将让出 CPU 资源。延时的时长由形参 xTicksToDelay 决定，单位为系统节拍周期，比如系统的时钟节拍周期为 1ms，那么调用 vTaskDelay(1)的延时时间则为 1ms。

vTaskDelay 延时是相对性的延时，它指定的延时时间是从调用 vTaskDelay 结束后开始计算的，经过指定的时间后延时结束。比如 vTaskDelay(100)，从调用 vTaskDelay 结束后，任务进入阻塞状态，经过 100 个系统时钟节拍周期后，任务解除阻塞。因此，TaskDelay 并不适用于周期性执行任务的场合。此外，其他任务和中断活动，也会影响 vTaskDelay 的调用(比如调用前高优先级任务抢占了当前任务)，进而影响到任务的下一次执行的时间。

(2) vTaskDelayUntil。

在 FreeRTOS 中,除了相对延时函数,还有绝对延时函数“void vTaskDelayUntil (TickType_t * const pxPreviousWakeTime, const TickType_t xTimeIncrement);”,这个绝对延时常用于较精确的周期运行任务。比如,有一个任务,希望它以固定频率定期执行,而不受外部的影响,任务从上一次运行开始到下一次运行开始的时间间隔是绝对的,而不是相对的。

3.5 FreeRTOS 任务的设计要点

一个优秀的嵌入式开发人员必须对自己设计的系统有非常深入的理解,包括任务的优先级、中断处理、任务的运行时间、行为和状态等。以下是设计嵌入式系统时需要考虑的重要因素。

在 FreeRTOS 中,程序运行的上下文包括:中断服务函数、普通任务、空闲任务。

1. 中断服务函数

中断服务函数在嵌入式系统中是一种特殊的上下文环境。它运行在非任务执行环境下,通常是芯片的特殊运行模式(特权模式)。在中断服务函数中,有几点需要特别注意。

(1) 不能挂起任务:不允许调用任何会阻塞的 API 函数。

(2) 保持简洁:中断服务函数应该尽可能简短,一般只用于标记事件并通知任务处理,因为中断的优先级高于所有任务,如果中断处理时间过长,会阻碍系统中其他任务的执行。

(3) 考虑中断频率和处理时间:设计时必须考虑中断的频率和处理时间,以确保整个系统的任务能正常运行。

2. 普通任务

普通任务的执行没有太多限制,似乎可以执行所有操作。但是,在实时系统中,有几个关键点需要注意。

(1) 避免死循环:如果一个任务出现了没有阻塞机制的死循环,低优先级的任务包括空闲任务都不能运行。这是因为处于死循环的任务不会主动让出 CPU。

(2) 设计阻塞机制:任务在不活跃时应进入阻塞态,以让出 CPU 使用权,确保低优先级任务能正常运行。紧急事件处理任务的优先级通常设置较高。

3. 空闲任务

空闲任务是 FreeRTOS 系统在没有其他工作时自动进入的任务。处理器始终需要执行代码,因此至少需要一个任务处于运行态。FreeRTOS 会在调用 vTaskStartScheduler 时自动创建一个空闲任务,空闲任务具有以下特点。

(1) 额外功能:用户可以通过空闲任务的钩子函数在系统空闲时执行额外的功能,如系统状态指示或省电模式。

(2) 资源回收:FreeRTOS 用空闲任务来执行系统资源回收,如删除任务的内存释放。

(3) 永不阻塞:空闲任务不能被阻塞,确保系统始终有任务可运行。空闲钩子函数应

满足不挂起空闲任务且不陷入死循环。

4. 任务的执行时间

任务的执行时间包括两方面：

(1) 任务运行时间：任务从开始到结束所需的时间。

(2) 任务周期：任务的运行周期。

在设计系统时需同时考虑这两方面：

(1) 对于事件 A 的服务任务 Ta，系统要求的实时响应时间是 10ms，而 Ta 的最大运行时间是 1ms。这样，10ms 是任务 Ta 的周期，1ms 是其运行时间。

(2) 假设系统中还有一个每 50ms 运行一次的任务 Tb，每次运行的最大时间是 100μs，即使 Tb 的优先级高于 Ta，也不会影响系统的实时性，因为它不会占用太多时间。

(3) 如果系统中还有一个任务 Tc，运行时间为 20ms，且优先级高于 Ta，那么 Tc 可能会导致 Ta 无法在 10ms 内完成对事件 A 的响应，这是不允许的。

在设计嵌入式系统时，应将处理时间更短的任务设置为更高优先级，合理安排任务的上下文环境和执行时间，以确保系统高效和稳定地运行。

3.6　FreeRTOS 任务管理应用实例

任务管理实验是将任务常用的函数进行一次实验，在野火 STM32 开发板上进行该试验，创建两个任务，一个是 LED 任务，另一个是按键任务，LED 任务显示任务运行的状态，而按键任务通过检测按键的按下与否来进行对 LED 任务的挂起与恢复。

1. 任务管理源代码

任务管理源代码如下：

```
/**********************************************************************
  * @file    main.c
  *
  * @brief   FreeRTOS V9.0.0 + STM32 任务管理
  * 实验平台:野火 STM32F407 霸天虎开发板
  **********************************************************************/
/**********************************************************************
 *                                包含的头文件
 **********************************************************************/
/* FreeRTOS 头文件 */
#include "FreeRTOS.h"
#include "task.h"
/* 开发板硬件 BSP 头文件 */
#include "bsp_led.h"
#include "bsp_debug_usart.h"
#include "bsp_key.h"
/**************************** 任务句柄 ********************************/
/*
```

```
 * 任务句柄是一个指针,用于指向一个任务,当任务创建好之后,它就具有了一个任务句柄
 * 以后要想操作这个任务都需要通过这个任务句柄,如果是自身的任务操作自己,那么
 * 这个句柄可以为 NULL
 */
static TaskHandle_t AppTaskCreate_Handle = NULL;            /* 创建任务句柄 */
static TaskHandle_t LED_Task_Handle = NULL;                 /* LED 任务句柄 */
static TaskHandle_t KEY_Task_Handle = NULL;                 /* KEY 任务句柄 */

/******************************* 内核对象句柄 ********************************/
/*
 * 信号量,消息队列,事件标志组,软件定时器这些都属于内核的对象,要想使用这些内核
 * 对象,必须先创建,创建成功之后会返回一个相应的句柄。实际上就是一个指针,后续
 * 就可以通过这个句柄操作这些内核对象
 *
 * 内核对象就是一种全局的数据结构,通过这些数据结构可以实现任务间的通信,
 * 任务间的事件同步等各种功能。这些功能的实现是通过调用这些内核对象的函数
 * 来完成的
 *
 */
/******************************* 全局变量声明 ********************************/
/*
 * 当在写应用程序的时候,可能需要用到一些全局变量
 */

/*************************************************************************
 *                              函数声明
 *************************************************************************/
static void AppTaskCreate(void);                    /* 用于创建任务 */

static void LED_Task(void* pvParameters);           /* LED_Task 任务实现 */
static void KEY_Task(void* pvParameters);           /* KEY_Task 任务实现 */

static void BSP_Init(void);                         /* 用于初始化板载相关资源 */

/*************************************************************************
  * @brief 主函数
  * @param 无
  * @retval 无
  * @note  第一步:开发板硬件初始化
           第二步:创建 App 应用任务
           第三步:启动 FreeRTOS,开始多任务调度
  ************************************************************************/
int main(void)
{
  BaseType_t xReturn = pdPASS;          /* 定义一个创建信息返回值,默认为 pdPASS */

  /* 开发板硬件初始化 */
  BSP_Init();
```

```
  printf("这是一个 FreeRTOS 任务管理实例!\n\n");
  printf("按下 KEY1 挂起任务,按下 KEY2 恢复任务\n");

  /* 创建 AppTaskCreate 任务 */
  xReturn = xTaskCreate((TaskFunction_t)AppTaskCreate,          /* 任务入口函数 */
                        (const char*   )"AppTaskCreate",       /* 任务名字 */
                        (uint16_t      )512,                   /* 任务栈大小 */
                        (void*         )NULL,                  /* 任务入口函数参数 */
                        (UBaseType_t    )1,                    /* 任务的优先级 */
                        (TaskHandle_t*  )&AppTaskCreate_Handle);  /* 任务控制块指针 */
  /* 启动任务调度 */
  if(pdPASS == xReturn)
    vTaskStartScheduler();          /* 启动任务,开启调度 */
  else
    return -1;

  while(1);                         /* 正常不会执行到这里 */
}

/**********************************************************************
  * @ 函数名  : AppTaskCreate
  * @ 功能说明: 为了方便管理,所有的任务创建函数都放在这个函数里面
  * @ 参数    : 无
  * @ 返回值  : 无
  **********************************************************************/
static void AppTaskCreate(void)
{
  BaseType_t xReturn = pdPASS;      /* 定义一个创建信息返回值,默认为 pdPASS */

  taskENTER_CRITICAL();             //进入临界区

  /* 创建 LED_Task 任务 */
  xReturn = xTaskCreate((TaskFunction_t)LED_Task,             /* 任务入口函数 */
                        (const char*   )"LED_Task",           /* 任务名字 */
                        (uint16_t      )512,                  /* 任务栈大小 */
                        (void*         )NULL,                 /* 任务入口函数参数 */
                        (UBaseType_t    )2,                   /* 任务的优先级 */
                        (TaskHandle_t*  )&LED_Task_Handle);   /* 任务控制块指针 */
  if(pdPASS == xReturn)
   printf("创建 LED_Task 任务成功!\r\n");
  /* 创建 KEY_Task 任务 */
  xReturn = xTaskCreate((TaskFunction_t)KEY_Task,             /* 任务入口函数 */
                        (const char*   )"KEY_Task",           /* 任务名字 */
                        (uint16_t      )512,                  /* 任务栈大小 */
                        (void*         )NULL,                 /* 任务入口函数参数 */
                        (UBaseType_t    )3,                   /* 任务的优先级 */
                        (TaskHandle_t*  )&KEY_Task_Handle);   /* 任务控制块指针 */
  if(pdPASS == xReturn)
```

```
    printf("创建 KEY_Task 任务成功!\r\n");

  vTaskDelete(AppTaskCreate_Handle);     //删除 AppTaskCreate 任务

  taskEXIT_CRITICAL();                   //退出临界区
}



/**********************************************************************
  * @ 函数名  : LED_Task
  * @ 功能说明: LED_Task 任务主体
  * @ 参数    : 无
  * @ 返回值  : 无
  **********************************************************************/
static void LED_Task(void* parameter)
{
  while (1)
  {
    LED1_ON;
    printf("LED_Task Running,LED1_ON\r\n");
    vTaskDelay(500);                     /* 延时 500 个 tick */

    LED1_OFF;
    printf("LED_Task Running,LED1_OFF\r\n");
    vTaskDelay(500);                     /* 延时 500 个 tick */
  }
}

/**********************************************************************
  * @ 函数名  : LED_Task
  * @ 功能说明: LED_Task 任务主体
  * @ 参数    : 无
  * @ 返回值  : 无
  **********************************************************************/
static void KEY_Task(void* parameter)
{
  while (1)
  {
    if(Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON)
    {/* K1 被按下 */
      printf("挂起 LED 任务!\n");
      vTaskSuspend(LED_Task_Handle);     /* 挂起 LED 任务 */
      printf("挂起 LED 任务成功!\n");
    }
    if(Key_Scan(KEY2_GPIO_PORT,KEY2_PIN) == KEY_ON)
    {/* K2 被按下 */
      printf("恢复 LED 任务!\n");
```

```
      vTaskResume(LED_Task_Handle);      /* 恢复 LED 任务! */
      printf("恢复 LED 任务成功!\n");
    }
    vTaskDelay(20);                      /* 延时 20 个 tick */
  }
}

/**********************************************************************
  * @ 函数名  : BSP_Init
  * @ 功能说明: 板级外设初始化,所有板子上的初始化均可放在这个函数里面
  * @ 参数    : 无
  * @ 返回值  : 无
  **********************************************************************/
static void BSP_Init(void)
{
/*
 * STM32 中断优先级分组为 4,即 4bit 都用来表示抢占优先级,范围为:0～15
 * 优先级分组只需要分组一次即可,以后如果有其他的任务需要用到中断,
 * 都统一用这个优先级分组,千万不要再分组
 */
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);

 /* LED 初始化 */
 LED_GPIO_Config();

 /* 串口初始化 */
 Debug_USART_Config();

 /* 按键初始化 */
 Key_GPIO_Config();
}
/****************************** END OF FILE ***********************/
```

以上的代码实现了在 STM32 开发板上使用 FreeRTOS 进行简单的任务管理。FreeRTOS 是一个实时操作系统内核,能够调度多个任务并且支持任务间通信。

(1) 头文件和任务句柄声明。

代码包含 FreeRTOS 和板级支持包(Board Support Package,BSP)相关的头文件。任务句柄用于区分和控制不同任务。

(2) 任务创建和管理。

ppTaskCreate_Handle 用于创建任务。

LED_Task_Handle 和 KEY_Task_Handle 分别用于控制 LED 和按键任务。

(3) 任务实现。

AppTaskCreate 函数负责统一创建其他所有任务,并在创建成功后删除自身,以节约资源。

LED_Task: 控制板载 LED 的开关,LED 每隔 500 个 tick 闪烁一次。

KEY_Task：通过扫描按键来挂起或恢复 LED 任务。按下 KEY1 挂起 LED 任务，按下 KEY2 恢复 LED 任务。

（4）系统初始化。

SP_Init 函数初始化所有板载硬件，包括 LED、串口和按键，同时设置中断优先级组。

（5）主函数。

硬件初始化后，创建 AppTaskCreate 任务，并启动调度器。一旦调度器启动，系统进入多任务处理阶段。

这段代码演示了如何在 STM32 板上应用 FreeRTOS 来管理不同任务，通过按键控制任务的挂起与恢复，从而实现简单的任务间同步操作。

2. 任务管理实例下载与运行结果

将程序编译好，用 USB 线连接计算机和 STM32 开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（这里为野火霸天虎 STM32F407 开发板），任务管理程序下载界面如图 3-4 所示。

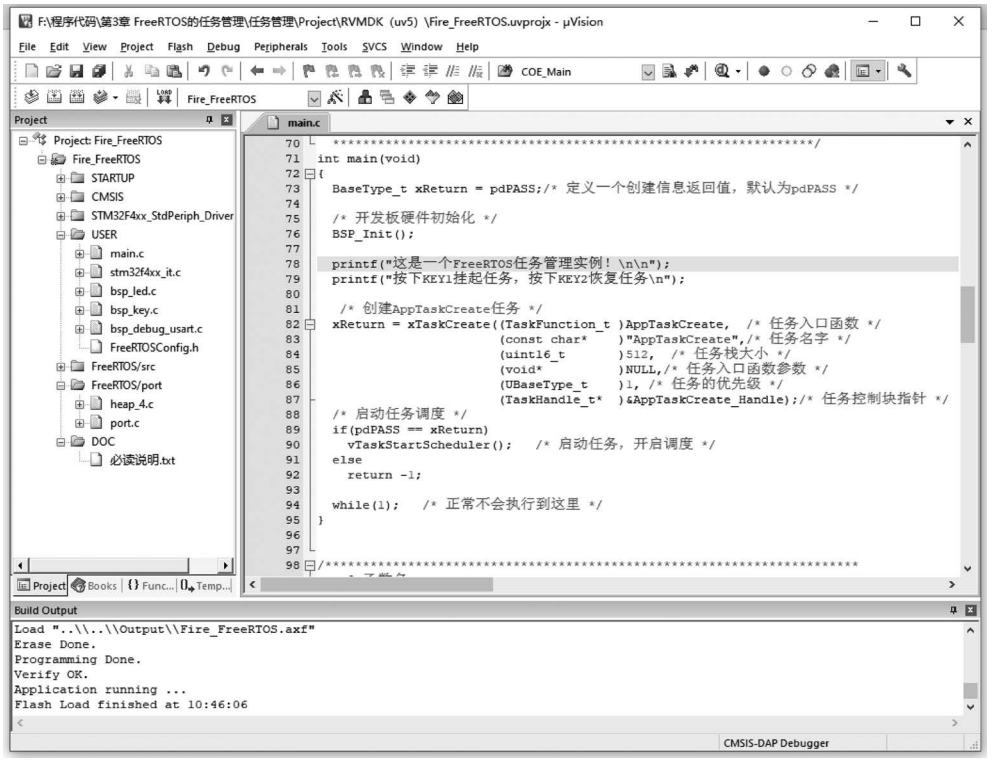

图 3-4　任务管理程序下载界面

在计算机上打开野火串口调试助手 FireTools，然后复位开发板就可以在调试助手中看到串口的打印信息。在 STM32 开发板可以看到，LED 在闪烁，按下开发板的 KEY1 按键挂起任务，按下 KEY2 按键恢复任务。按下 KEY1，可以看到开发板上的灯也不闪烁了，同时

在串口调试助手也输出了相应的信息，说明任务已经被挂起；按下 KEY2，可以看到开发板上的灯也恢复闪烁了，同时在串口调试助手也输出了相应的信息，说明任务已经被恢复。任务管理实例运行结果如图 3-5 所示。

图 3-5　任务管理实例运行结果