第3章 嵌入式实时操作系统 FreeRTOS 原理

嵌入式操作系统(Embedded Operating System,EOS)是一种用于嵌入式系统的系统软件。嵌入式系统由底层硬件和上层应用软件组成,具体为基本外设、外部辅助设备和各类通信协议。与需要 Windows、Linux 或 macOS 等操作系统来运行基本应用程序的台式计算机相似,嵌入式系统也需要一个操作系统来促进其功能。例如,所有的移动电话都有一个集成的嵌入式操作系统软件,如安卓(Android)或 iOS,在手机开机时启动。嵌入式操作系统具有以下一些共同特征:省电、较少的存储能力、较小的处理能力、快速和轻量级、I/O 设备灵活以及可针对实际任务进行定制。

嵌入式操作系统通常是以实现特定功能为目的而设计的,它通过提供基本的驱动程序、辅助工具和库函数,实现合理调度系统资源、协调管理硬件等功能,对特定的任务具有较高的资源调度效率和较强可靠性,因而具有十分广泛的用途。目前在工商业界广泛使用的嵌入式操作系统有 Embedded Linux 、QNX、WinCE、VxWorks 以及智能手机上的 Android 和 iOS 等。

嵌入式操作系统可以根据其对外界事件的响应能力分为分时操作系统和实时操作系统。假如一台计算机能够在操作系统的驱动下采用时间片轮转的方式同时为几个、几十个甚至几百个用户服务,那么这种操作系统就被称为分时操作系统;假如一台计算机能够在操作系统的驱动下在一个规定的时间内完成对外界事件的处理,并及时响应事件处理的结果,同时能控制所有实时设备和实时任务协调运行,那么这种系统就称为实时操作系统(Real Time Operating System,RTOS)。本章将介绍的 FreeRTOS 就是一种嵌入式实时操作系统。

3.1 FreeRTOS 概述

FreeRTOS 是由 Real Time Engineers 公司开发的一款微型实时操作系统,是一个可移植的开放源码软件,可运行在微控制器(MCU)上。作为开源的微内核操作系统,FreeRTOS 主要应用于一些资源有限的小型嵌入式系统,目前主要为用户提供硬件基础驱动服务。作为一款优秀的实时操作系统,FreeRTOS 具备了 RTOS 所有的基础功能,如任务管理、任务调度、内存管理、时间管理、消息队列等。得益于其代码量小、驱动效果好等特点,FreeRTOS 深受广大研发人员的喜爱。

由于在开发、迭代以及维护过程中受到高效严格的管理,FreeRTOS 具有极高的代码规范性和健壮性,并且具有较高的移植性,可以应用到多种处理器和开发板上。同时,FreeRTOS 配备丰富的练习实例能使开发人员能够迅速掌握并熟练运用,从而提高开发效

率，是初学者入门嵌入式操作系统的佳选之一。除以上几点，FreeRTOS 操作系统还有以下一些主要的优点。

1. 支持多任务

FreeRTOS 支持多任务调度，并对系统内的任务总数不设限制，任务量根据需求而定。FreeRTOS 支持多种调度方式：抢占式调度、时间片调度以及合作式调度，可以灵活地为不同需求的任务设置不同优先级。同时，FreeRTOS 会分配独立的栈（Stack）空间给每个任务，并为这些任务栈空间设定最小值，且允许用户根据实际需求调整空间大小。此外，用户还可以根据 FreeRTOS 关于栈空间合理性的特征标准，对空间大小进行调整以减少系统对 RAM 的占用。

2. 时间确定性

FreeRTOS 软件定时器本质上是一个周期性的任务或单次执行的任务，在被创建之后，当经过设定的时钟计数值后，会触发用户定义的回调函数，由于它的实现不需要使用任何硬件定时器，故被称为“软件计时器”。同时，FreeRTOS 中大部分函数具有固定的调用和执行时间，并且执行时间的长短不会受应用程序量的多少而变化，具有时间的确定性。

3. 内核调度方式

在 FreeRTOS 中，用户可以根据自身需求设置内核中任务的调度方式：当调度方式设置为抢占式时，若内核中有比当前任务更高优先级的任务进入就绪态，则抢占当前任务的 CPU 使用权，并将当前任务挂起，这种方式能最优化任务的响应时间，使优先级越高的任务越先执行，从而保证系统的实时性要求；当调度方式设置为非抢占式时，内核中优先级更高的任务进入就绪态时无法抢占当前任务的 CPU 使用权，必须等待当前任务执行结束，主动释放 CPU 后才可以开始执行，与抢占式相比拥有更快的中断响应。

4. 通信服务

FreeRTOS 具有多种通信方式，其中队列是任务之间最主要也是最基础的通信方式，在任务-任务、中断-任务之间进行消息的传递。

5. 中断管理

中断指的是 CPU 在执行事务时暂停当前正在执行的事件而对更高优先级事件处理，处理完高优先级的事件后再返回中断位置继续执行的过程。中断的方式能有效提高 CPU 的利用效率，但不当的中断使用会导致系统的不稳定。FreeRTOS 系统支持中断嵌套，并能通过有效的中断管理以及合理的配置增强系统的稳定性。

6. 模块可配置性

得益于高内聚、低耦合的代码设计，FreeRTOS 的不同组成模块之间相互独立性高。在实际应用中，可以根据用户需求在配置文件中配置不同的功能模块，目前大多数的配置选项都在文件 FreeRTOS.h 中，用户通过简单修改其中的宏定义值便可以实现模块的配置，便捷、灵活地适应项目需求。

FreeRTOS 系统也有缺点，该系统不会根据任务的特点对时间片进行划分，只是机械性地随机分配时间片，并且只能根据优先级来调度任务。因此，在保证系统性能较可靠、提升系统功能模块处理效率的情况下，任务量的增多以及处理时间的延长会导致任务间的切换次数明显增多，严重浪费系统的处理时间。

3.2 FreeRTOS 体系结构

3.1 节提及 FreeRTOS 的高移植性，目前它已经被移植到 32 个硬件架构中，同时支持大量开发工具。其内核代码文件占用的内存空间非常小，仅由 3 或 4 个.c 文件和一些汇编函数构成，只占了 4KB～9KB 的内存。其中 Source 文件夹包含了 FreeRTOS 的全部内核源文件，内核的主要组成文件是 tasks.c、list.c、queue.c 和 croutine.c。

1. tasks.c

与任务调度相关，包含内核所需的各个任务功能的实现，通过任务切换实现包括任务就绪、任务延时、任务休眠以及任务唤醒等功能，给予任务不同的调度策略。

2. list.c

与内核调度相关。list 就是数据结构中的链表，list.h 中定义了 3 种链表节点的数据结构，而 list.c 中则定义了 5 个函数，用于具体实现系统的链表，提供内核调度器。

3. queue.c

与实现队列相关。队列是内部信息交流的主要方式，是其他通信方式的基础。queue.c 源文件能够实现队列、控制信号量，并且支持中断环境。

4. croutine.c

与实现联合程序相关，这是堆栈的另外一种组织形式，可以看作一个临时的数据寄存、交换的内存区，不同任务可以共享同一堆栈内的存储空间，使得系统不用过度依赖 RAM。

FreeRTOS 的内核中不仅包含以上 4 个源文件，还有一个名为 timers.c 的文件，用于实现内核的时钟统一。include 文件夹中主要包含系统所需的头文件。portable 文件夹包含移植实例中各个平台的相关移植文件和内存管理文件，是系统的移植文件夹。

由于 FreeRTOS 的大多数源代码是用 C 语言编写的，所以它的可读性强、可移植性强，并易于维护与扩展，这也是其受欢迎的重要原因之一。图 3-1 展示的就是 FreeRTOS 的代码结构，也展示了 FreeRTOS 在嵌入式系统中的角色。

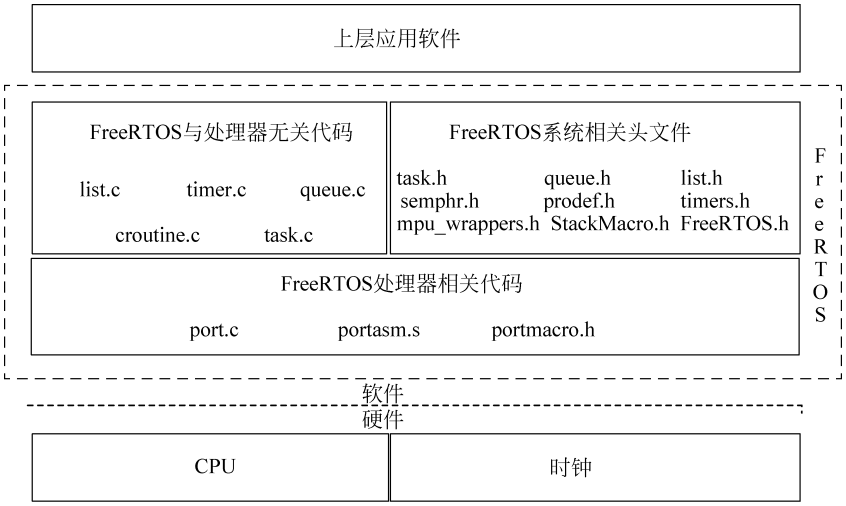

图 3-1 FreeRTOS 代码结构

在对 FreeRTOS 的代码有了大致了解后，接下来将介绍 FreeRTOS 中的几个主要模块：任务管理模块、时间管理模块、内存管理模块、协同例程管理模块，如图 3-2 所示。

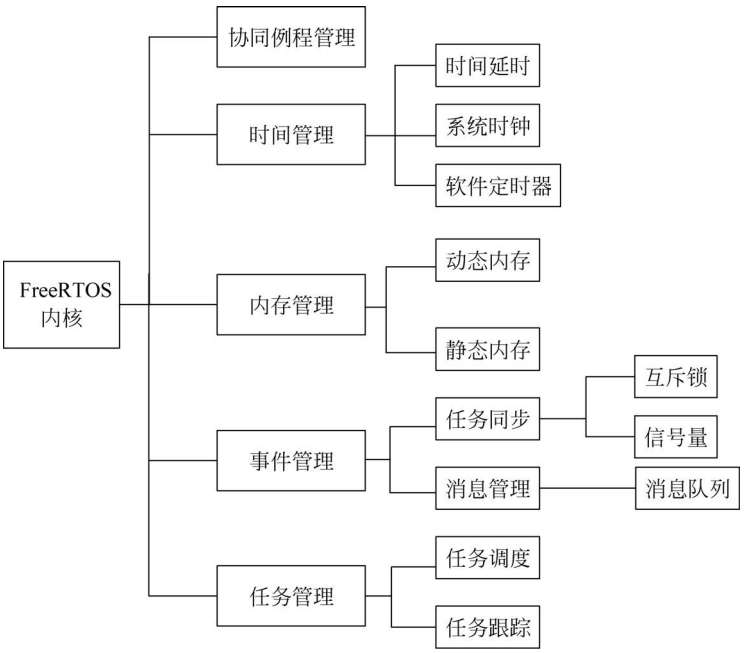

图 3-2　FreeRTOS 内核架构

3.2.1　任务管理模块

操作系统中的任务(Task)指的是一个简单程序或一个线程，是受处理器调度的工作单元。在 FreeRTOS 中，任务和线程不作区分，即一个任务就是一个线程。在 RTOS 中，为保证程序的实时性，通常会将一个目标分为多个小任务，并为各任务分配适当的优先级以及独立的寄存器和堆栈空间等，提高了程序的模块化和可维护性。CPU 通过在多任务之间的调度与切换，依次执行多任务中的每一个任务，以此来实现多任务的运行，从而提高了 CPU 的运行效率，保证系统实时性。tasks.c 文件中实现了 FreeRTOS 的任务管理模块，借助功能函数，FreeRTOS 实现了多任务的有效管理。下面对其中的主要功能函数进行介绍。

1. 任务创建：xTaskCreate()函数

任务创建指操作系统创建一个新的任务，以便于后续进行调度或执行。任务创建可以在调度前，也可以发生在调度后执行过程中，但在执行任务调度之前必须创建至少一个任务，同时 FreeRTOS 的任务控制块也随着系统创建任务的过程中进行动态分配。FreeRTOS 操作系统中任务创建函数是所有函数中最为复杂的。

xTaskCreate()是 FreeRTOS 中创建任务的函数，包含一个指向任务方法初始化接口的指针、任务的优先级、任务的句柄、任务的描述等参数，具体函数声明如下：

```
portBASE_TYPE xTaskCreate(
        pdTASK_CODE pvTaskCode,                    //指向任务的实现函数的指针
        const portCHAR * const pcName,             //任务名
        unsigned portSHORT usStackDepth,           //内核为任务分配的栈空间大小
```

```
        void * pvParameters,                    //指向 void,用于将参数传递给任务
        Unsigned portBASE_TYPE uxPriority,      //任务的优先级
        xTaskHandle * pvCreatedTask             //传出任务的句柄
);
```

函数有两个返回值，若创建任务成功则返回 pdTRUE，若创建任务失败则返回 errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY，创建任务失败的原因可能为无法分配足够的空间保存任务的数据和堆栈等上下文信息。

在参数传入之后，函数首先检测任务的优先级是否有效，优先级必须小于 configMAX_PRIORITIES，且设定在规定的范围之内；优先级检测通过后，函数会检测系统剩余的堆栈空间，根据堆栈增长方向设置堆栈的开始指针并分配堆栈空间，从而为任务分配可信计算基(Trusted Computing Base，TCB)和堆栈所需内存。之后初始化 TCB 的任务名称、优先级和堆栈深度，初始化 TCB 的堆栈、任务列表并将任务添加到就绪列表中。当系统设置为抢占式内核时，创建任务成功后，会检查新创建任务是否为创建的第一个任务，若不是则会判断该任务与当前执行任务的优先级高低，若该任务优先级高于当前执行任务则会调用 portYIELD_WITHIN_API()函数进行任务切换。同时由于 xTaskCreate()函数是 xTaskGenericCreate()函数的宏定义，因此创建任务时实际调用的是 xTaskGenericCreate()函数。堆栈缓存和内存管理是 xTaskGenericCreate()函数中特有的两个参数。

2. 任务删除：vTaskDelete()函数

任务删除指操作系统将现存的任务从任务列表中剔除，在 FreeRTOS 中通过调用 vTaskDelete()函数实现，任务删除的函数声明如下：

```
void vTaskDelete( xTaskHandle pxTaskToDelete );     //任务删除函数声明
```

pxTaskToDelete 是任务删除所需的参数，当这个参数设为 NULL 时表示删除本身。这个函数只包含一个任务句柄，通过此句柄来查找对应的任务并完成删除相关工作。任务在被删除之后就直接从就绪态、挂起态等任务列表中被删除，不会再次进入运行态。但删除任务时，系统不会自动释放被删除任务所占用的系统资源，因此在删除任务时必须主动释放任务所占内存，避免内存冗余。FreeRTOS 的任务删除操作分两步完成：第一步在 vTaskDelete()函数中完成，FreeRTOS 先把要删除的任务从就绪任务链表和事件等待链表中删除，然后把此任务添加到任务删除链表(即 xTasksWaitingTermination)，若删除的任务是当前运行任务，系统将强制执行任务调度函数；第二步则是在空闲任务中完成的，当空闲任务运行时，会检查 xTasksWaitingTermination 链表，如果有任务在这个表中，就释放该任务占用的内存空间，并把该任务从任务删除链表中删除，此时才算真正意义上的任务删除。在进行任务删除时，系统要留给空闲任务一定的执行时间，避免内存空间释放失败造成内存冗余。

3. 任务挂起与恢复：vTaskSuspend()函数、vTaskResume()函数

任务挂起指当前任务从就绪态进入挂起态，暂停运行。FreeRTOS 中通过调用 vTaskSuspend()函数来实现任务的挂起，任务的恢复指的是解除挂起态，使任务回到就绪态，FreeRTOS 中任务的恢复函数是 vTaskResume()。任务挂起与恢复的函数声明如下：

```
void vTaskSuspend(xTaskHandle pxTaskToSuspend);     //任务挂起
void vTaskResume(xTaskHandle pxTaskToResume);       //任务恢复
```

1）任务挂起函数

pxTaskToSuspend是任务挂起所需的参数，该参数设置为NULL时表示挂起本身。与任务删除类似，当调用vTaskSuspend()函数时，首先判断pxTaskToSuspend是否为NULL，即判断是否为挂起当前任务本身；接着FreeRTOS把要挂起的任务从就绪任务链表、延时任务链表和事件等待链表中删除，并将此任务添加进任务挂起链表；若挂起的任务是当前运行任务，系统就强制执行任务调度函数。在任务挂起过程中，需要注意不能调用FreeRTOS的API函数，因为一旦挂起任务它的延时和对事件的等待都会被取消。

如果任务被确定为挂起态，就意味着这个任务即使具有很高的优先级也不可能得到CPU的使用权。同时，挂起操作不会重复累计，无论调用多少次vTaskSuspend()函数，解除挂起的函数只需要执行一次。另外，任务挂起与任务退出和任务进入所实现的临界区不同，挂起操作能够保证区间内的代码在执行过程中不被其他任务中断，因此通过任务挂起实现的“临界区”可以有效提高长代码的执行效率、并且该“区”不适合简单中断来实现任务切换的情况。

2）任务恢复函数

FreeRTOS中的任务恢复时也需要将要被解除挂起态的任务作为参数进行执行，pxTaskToResume是任务恢复所需的参数。任务恢复与挂起相反，首先将任务根据其优先级依次添加到就绪任务链表中，若恢复的任务比当前正在执行的任务优先级高，则系统将强制执行任务调度函数，使高优先级任务率先执行。

3.2.2 时间管理模块

FreeRTOS中的时间管理分为两方面：一方面是系统内部所需的时钟节拍；另一方面是系统任务调度时所需的延时函数。对于一个操作系统来说时钟节拍是不可或缺的，它是操作系统中的最小时间单位，通常由硬件定时器实现。FreeRTOS的任务调度主要就是通过时钟节拍来控制延时、超时等最基本的功能，以实现对时间的管理。

如果将操作系统看成一个人，时钟节拍可以比作操作系统的心跳，它产生一个周期性的中断，不同应用的中断周期不同，大部分都在1～100ms范围内。中断会导致内核中的任务被延迟若干时钟节拍。

而对于实时操作系统来说，时间管理中另一个重要的部分就是时间延迟：在时间片轮转调度策略中，对于需要周期性停止运行的任务可以通过延迟处理函数来提供所需的延迟。

通过调用系统提供的延时函数任务可以进入非运行态，例如进入阻塞态或就绪态等。若要重新进入运行态，系统将通过查询方式来激活相应的任务，并规定一个确定的时间来延时任务，而这个延时时间则取决于系统提供的时钟频率。系统可以依据该时钟频率来计算延时所需的实际时间。对于如何管理任务延迟，FreeRTOS中提供了两个API函数：一个是vTaskDelay()；另一个是vTaskDelayUntil()。下面具体展示与FreeRTOS时间管理模块相关的3类函数。

1. 任务延迟：vTaskDelay()函数

```
/* 延迟时间长度 */
void vTaskDelay(const TickType_t xTicksToDelay);   //任务延迟函数
```

vTaskDelay()函数延时的时间是相对的，这个API函数会受到其他任务和中断的执行

影响。上述代码中，参数 xTicksToDelay 表示延迟的时钟节拍个数，范围是 1～0xFFFFFFFF，portmacro.h 文件中定义了延迟时间的最大值。

2. 周期性延迟：vTaskDelayUntil()函数

```
/* 周期性延迟时间 */
void vTaskDelayUntil(TickType_t   * pxPreviousWakeTime,
                     const TickType_t xTimeIncrement);   //周期性延迟函数
```

vTaskDelayUntil()函数与 vTaskDelay()函数不同，实现的是周期性延迟，该函数可以为任务提供一个精确的阻塞延时，此延迟可以被定为一个固定的周期调用来执行任务，保证任务执行频率的恒定。上述代码中，入口参数 pxPreviousWakeTime 指定的是任务从阻塞态中唤醒进入就绪态的时刻，xTimeIncrement 表示周期性延迟时长。使用该函数时要注意以下两点。

(1) 需要在 FreeRTOSConfig.h 配置文件中配置如下宏定义为：

```
#define INCLUDE_vTaskDelayUntil 1
```

(2) 注意对比与 vTaskDelay()函数的区别，从而在不同情况下选择更合适的函数：vTaskDelayUntil()函数可以指定一段精确的相对延时时间，而 vTaskDeclay()函数只可以提供一个相对时间。因此，在时间要求较严格的情况下，例如任务需要在的固定周期内执行，就应该选用 vTaskDelayUntil()函数而非 vTaskDelay()函数。

3. 获取时钟节拍数：xTaskGetTickCount()函数、xTaskGetTickCountFromISR()函数

```
volatile TickType_t xTaskGetTickCount(void);
volatile TickType_t xTaskGetTickCountFromISR(void);
```

FreeRTOS 系统用 xTaskGetTickCount()函数和 xTaskGetTickCountFromISR()函数来获取当前运行的时钟节拍数。使用时注意辨清两个函数的区别，不可混用。前者主要用于任务代码里面的调用，后者主要在中断服务程序里面调用。

3.2.3 内存管理模块

内存管理模块主要是对内存进行动态分配，这其实是一个 C 编程概念，并非 FreeRTOS 所特有，但却是 FreeRTOS 中非常重要的部分。FreeRTOS 中内核对象都是动态分配的，动态内存的申请和释放涉及两个重要函数：pvPortMalloc()函数和 vPortFree()函数，pvPortMalloc()函数和 vPortFree()函数具有与标准 C 库 malloc()函数和 free()函数相同的原型。当 FreeRTOS 需要 RAM 时，会调用 pvPortMalloc()函数，当 RAM 被释放时，内核会调用 vPortFree()函数。

1. 函数

1) pvPortMalloc(size_t xSize)

pvPortMalloc()函数是由 FreeRTOS 规定的内存分配函数，开发者或用户如果想要使用内存，只能通过该函数进行申请内存分配。

2) vPortFree(void * pv)

vPortFree()函数是由 FreeRTOS 规定的内存释放函数，开发者或用户如果想要释放内存，只能通过该函数进行申请释放内存。需要注意的是，在 heap_1.c 方案中，内存一旦申请分配将无法释放。

2. 内存分配

FreeRTOS还提供了5种内存分配方案，分别存储在源文件 Heap_1.c、Heap_2.c、Heap_3.c、Heap_4.c 以及 Heap_5.c 中，使用者可以根据实际项目的情况来选择合适的内存分配方案，以下对不同的内存分配特点进行介绍。

1) Heap_1.c

最简单的内存分配算法。这种算法将 FreeRTOS 的内存空间视为一个简单数组，当需要分配内存时，将数组细分，取出所需要的内存块，也可以在配置文件中自定义 FreeRTOS 的内存空间。这种方法虽简单，但内存分片完成之后便不允许分配的内存被释放。

2) Heap_2.c

最佳匹配的内存分配算法。与第一种算法不同，该算法支持动态内存分配与释放，甚至比标准 C 库当中的内存分配方案更有效，因此尤其适用于实时性操作系统创建小型动态任务。但这种分配算法使用时会产生一些内存碎片，这些内存碎片可能因为过小而无法使用，造成空间的浪费。

3) Heap_3.c

调用标准库函数的内存分配算法。Heap_3.c 封装了编译器支持的标准 C 函数库中的内存分配和释放函数。得益于这种分配算法的线程安全保护特性，这种分配算法被应用于大多数的项目当中。

4) Heap_4.c

首次匹配的内存分配算法。Heap_4.c 将所有空闲的内存块碎片合并为一个内存块进行回收再利用，以此减缓内存分配和释放时产生的大量内存碎片对系统造成的影响。该分配算法适用于反复分配和删除任务、队列、信号量等情况，此外，这种分配算法也非常适用于在移植层进行内存分配的应用。

5) Heap_5.c

在 Heap_4.c 的基础上进一步优化，链接不连续的内存区作为堆空间，同时仍然可以合并空闲内存碎片，将之合并为内存块进行回收利用，依旧不提供分配为内存块的详细信息。

3.2.4 协同例程管理模块

协同例程是 FreeRTOS 提供的另一种用于实现用户任务的机制，它主要使用在 RAM 较小的处理器上，而在 32 位的处理器上几乎不使用。与任务相比，协同例程具有以下异同点。

1. 不同之处

协同例程只有运行态、就绪态和阻塞态，并没有挂起态。

(1) 协同例程的运行态与任务的运行态意义相同。

(2) 协同例程的就绪态与任务的就绪态意义相同，但条件不同。若任务与协同例程混用，当有任何一个任务处在运行态时，都会导致协同例程无法执行。

(3) 协同例程的阻塞态与任务的阻塞态类似。协同例程在等待时间或者等待外部事件时会进入阻塞态，它会使用 crDELAY()函数来等待一段时间。

2. 相似之处

协同例程的优先级与任务类似。

每个协同例程优先级的取值区间为 0～configMAX_CO_ROUTINE_PRIORITIES－1，并且优先级共享。系统会优先调度到任务上而非协同例程。在一个任务与协同例程混合的系统中，优先级关系可以概括为：高优先级任务＞低优先级任务＞高等级协同例程＞低等级协同例程。

在空闲任务的函数中调度协同例程，会与任务混合使用，这时根据上文提到的先后顺序，会在所有的任务执行完之后才执行协同例程。因此，在有混合任务与协同例程的项目中，应当把重要性低、占用时间短且对实时性要求不高的事情放在协同例程中处理。因为在等数量的任务中协同例程所占用的 RAM 更少，所以协同例程更适合低内存的处理器和应用于小设备上。下面介绍一些协同例程中的注意事项。

（1）FreeRTOS 要求用户定义固定形式的协同例程，实现协同例程的模板代码如下：

```
void ACoRoutiineFunction (CoRoutineHandle_t xHandle, UBaseType_t uxIndex)
{
    crSTART(xHandle):
while(1)
{
    //在此处编写协同例程代码
}
    crEND():
}
```

在定义的过程中，需要注意的是所有的协同例程都必须分别调用 crSTART()函数和 crEND()函数来开始和结束。与任务相同，协同例程代码结构是一个死循环，不允许返回。同一个协同例程模板可以创建多个协同例程，它们彼此之间通过 uxIndex 便可区分。使用协同例程时，调度器会自动创建空闲任务，因此 vCoRountinueSChedule()函数通常在空闲任务的函数中被调用来实现协同例程的调度。

（2）在协同例程处于阻塞态时，协同例程的堆栈并不保持，这种情况下协同例程在栈上申请的变量可能存在值丢失的问题，因此协同例程中必须将需要保持数据的变量定义为 static，代码示例如下：

```
void vACoRoutineFunction(CoRoutineHandle_t xHandle, UBaseType_t uxIndex)
{
static char c = 'a':
     // 协同例程必须以调用 crSTART()函数作为开始
     crSTART(xHandle):
while(1)
{
     //如果在这里将 c 的值设为'b'... c = 'b'
     // ...然后阻塞协同例程... crDELAY(xHandle, 10)
     // ... c 的值只有在声明成静态类型才会一定等于'b'// (as it is here)
}
     //协同例程必须以调用 crEND()函数作为结束
     crEND():
}
```

（3）在程序中，只能由协同例程本身来调用那些可能导致协同例程阻塞的 API，而不能由其内部调用的函数来调用，代码示例如下：

```
void vACoRoutineFunction(CoRoutineHandle_t xHandle, UBaseType_t uxIndex)
{
   //协同例程必须以调用 crSTART()函数作为开始
   crSTART(zHandle);
   while(1)
   {
     //允许在该处进行阻塞调用,如 crDELAY(xHandle ,10)
     vACalledFunction();
   }
   //协同例程必须以调用 crEND()函数作为结束
   crEND();
}
void vACalledFunction(void)
{
   //不允许在该处进行可能会导致阻塞的调用
}
```

(4) FreeRTOS 的默认协同例程实现中只能用 for 语句,不能用 switch 语句,代码示例如下:

```
void vACoRoutineFunction(CoRoutineHandle_t xHandle, UBaseType_t uxIndex)
{
    //协同例程必须以调用 crSTART()函数作为开始
    crSTART(xHandle);
    while(1)
    {
    //允许在该处进行阻塞调用,如  crDELAY(xHandle, 10);
        switch(aVariable)
        {
            case 1 ://不允许在该处进行可能会导致阻塞的调用    break :
            default://这里也不允许
        }
    }
    //协同例程必须以调用 crEND()函数作为结束
    crEND () :
}
```

3.3 FreeRTOS 调度机制

在“嵌入式系统”概念出现之前,程序主要运行于裸机之上,在此基础上有了前后台系统。如今经过多年的发展,实时操作系统也较为成熟,但是任务的本质却没有发生变化。其都是通过在一个不断循环的结构中对相关的函数进行调用,来完成某些指定的功能。

3.3.1 任务结构

在多任务执行的过程中,任务有 4 个不同状态,分别为运行态(Running)、阻塞态(Blocked)、挂起态(Suspended)和就绪态(Ready)。操作系统会根据各个任务优先级的不同对它们进行一定的调度,通过控制不同任务的状态切换来分配它们对 CPU 的使用权。

在 FreeRTOS 中并不区分任务和进程的概念，即任务就是进程。FreeRTOS 中任务具有优先级划分，被分配有独立的堆栈空间并由调度器来调度。每个时间片只能执行一个任务，由 FreeRTOS 调度器使用调度算法来选择当前具体执行哪个任务。每个任务必须有确定的一个任务状态，4 个状态之间的转换如图 3-3 所示。

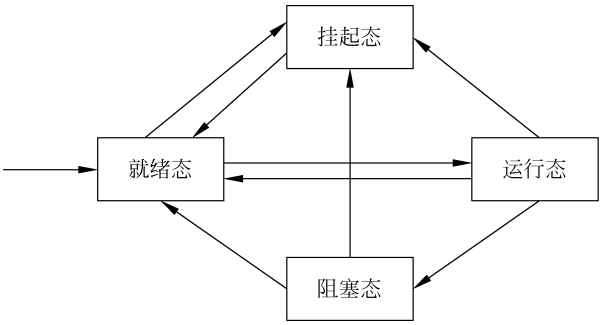

图 3-3　4 个状态之间的转换

运行态指示当前实际正在运行的任务，该任务会占用 CPU 的使用权。对单核微控制器来说处于运行态的任务有且只有一个。

就绪态任务指的是那些未被阻塞和挂起，但因当前有更高优先级的任务在执行而没有运行的任务。刚刚创建的任务会处于就绪态，它们位于就绪列表中，随时等待被调度器调度到运行态。

挂起态属于非运行状态的一个子状态。如果某一任务处于挂起态，那么该任务对于调度器而言是不可见的，调度器无法对之进行管理。要挂起任务，只能通过调用 vTaskSuspend()函数来实现，并且也只有通过调用 vTaskResume()函数才能将处于挂起态的任务唤醒。

阻塞态是指任务由于等待一个外部事件(如信号量、队列、通知、互斥量、事件标志组等)而处于的状态。处于阻塞态的任务不在就绪列表中且不能被调度器调用，同时阻塞态任务一般会设置一个超时计时器，如果超出了计时器设置好的等待时间，该任务就会退出阻塞态。被阻塞的任务会等待以下两种事件以退出阻塞。

(1) 同步事件：该事件来自其他任务或中断。例如，某一任务等待队列中数据来退出阻塞。

(2) 定时事件：该事件可以到达特定时间点或延时时间结束。例如，某一任务进入阻塞态的时间为 25ms，即 25ms 后自动退出阻塞。

3.3.2　任务调度原理

任务调度是操作系统的一门核心技术，也是系统的主要职责之一。任务调度实现了 CPU 对于不同任务在执行时间上的合理分配，而操作系统的实时性恰恰就是体现在任务不同状态间的切换。FreeRTOS 支持不限数目的任务同时进入优先级队列，同时提供了优先级调度算法和轮换调度算法来进行任务调度，这些算法都是通过 task.c 来实现的，表 3-1 是对 task.c 中主要全局变量的描述。

表 3-1　task.c 主要全局变量

序号	名　　称	说　　明
1	pxCurrentTCB	正在运行的任务
2	pxReadyTaskList	Ready List，就绪态，等待被调度任务
3	xDelayedTaskList	被阻塞的任务列表
4	xPendingReadyList	就绪态，调度器停止时无法加入 Ready List 任务

实时操作系统主要以基于优先级调度的方式进行任务的调度。系统会根据任务优先程度的不同，为每个任务赋予不同的优先级，并让高优先级的任务优先运行。根据 CPU 使用权获得方式的不同，在优先级确定的任务调度中，又可以将内核分为两类：抢占式内核和非抢占式内核，又称可剥夺型内核与不可剥夺型内核。

当不可剥夺型内核进行任务调度时，当前运行的任务不必担心其他任务对 CPU 的抢占，即使是高优先级任务也不可中断内核中正在运行的任务或抢占 CPU，只有当前任务自愿放弃 CPU 使用权或运行完毕之后退出内核，才允许其他任务使用 CPU，因此不同任务之间不会互相争夺 CPU，从而引发混乱。相比于前后台系统，这种内核调度方式的优点在于，其调度速度更快，并且由于任务无法抢占 CPU，系统也不需要设置信号量，对当前任务的共享数据进行保护。然而，由于高优先级任务进入就绪态后无法立即得到处理，且等待时间不确定，因而不可剥夺型内核实时性要比可剥夺型内核差。

可剥夺性内核进行任务调度时，在内核运行任务过程中，若有更高优先级的任务进入就绪态，允许其剥夺当前正在运行的任务的 CPU 使用权。相比于不可剥夺内核的调度方式，剥夺型内核调度方式中，当前优先级最高的任务总是具有确定的执行时间，这就保证了系统具有良好的实时性。由于可剥夺型内核在任务调度时，总是优先让高优先级任务先运行，中断当前任务，但中断完成之后内核会重新查找并去执行下一个最高优先级任务，而不一定会再去运行原来被中断的任务，因此可剥夺型内核的程序必须使用信号量等方式来保护共享的数据。

上文提到过 FreeRTOS 的可配置性，系统提供了配置文件 FreeRTOSConfig.h，用户可以通过对该文件进行修改，来对操作系统内核的调度方式进行修改，可以将其在不可剥夺型内核和可剥夺型内核之间切换，实时操作系统在通常情况下会使用可剥夺型内核作为调度方式。同时，用户还可以根据需求，通过 configMAX_PRIORITIE()函数，对系统可以拥有的最大优先级个数进行设置。对于优先级的最大值，FreeRTOS 本身并没有做出限制，任务的优先级可以设置在 0～configMAX_PRIORITIES－1 范围内依次升高，但优先级的最大值与内存空间的消耗是成正比的，因此用户通过自主调整优先级设置可以节省内存空间。FreeRTOS 可以自主选择配置，这让用户可以根据自己的需求，对系统进行定制，使系统更加灵活，可用性更高。

在优先级调度实现上 FreeRTOS 采用了一种双向循环链表结构，此数据结构在 list.c 以及 list.h 中定义，同时支持优先级调度算法和轮转调度算法。借助这种结构，对于优先级不同的任务，FreeRTOS 会采用优先级调度算法对系统任务进行调度，而对于优先级相同的任务，FreeRTOS 则会采用时间片轮转调度算法来进行调度。任务调度如图 3-4 所示。

图 3-4 中的 pxReadyTasksLists[configMAX_PRIORITIES]是 FreeRTOS 定义的就绪任务链表，其中 configMAX_PRIORITIES 是文件 FreeRTOSConfig.h 中定义的系统中最

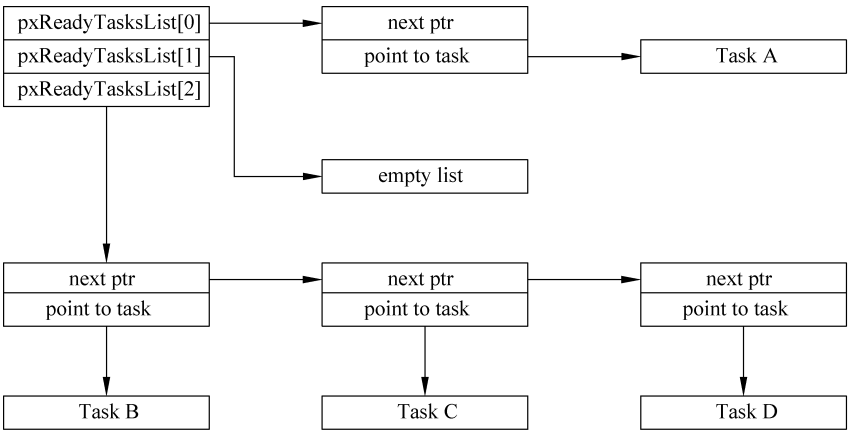

图 3-4　任务调度

大优先级的值。在图 3-4 中系统一共创建了 4 个任务，pxReadyTasksLists[n]中的 n 值就是指优先级，该链表中就是优先级为 n 的任务。FreeRTOS 用于优先级调度实现的链表和链表节点结构定义部分的代码如下：

```
typedef struct xLIST                                    //链表结构定义
{
      volatile unsigned portBASE_TYPE uxNumberOfItems;      //记录链表中有多少个元素
      volatile xMiniListItem xListEnd;                  //指向链表尾节点的指针
      volatile xListItem * pxIndex;                     //用于遍历链表,指向上次访问节点的指针
}  xList;
```

其中，uxNumberOfItems 指的是链表中的节点数量，它代表该链表中处于就绪态任务的个数。该值为 0 代表此链表中无就绪态任务。

```
struct xLIST_ITEM                                       //链表节点结构定义
{
      portTickType xItemValue;                          //存放时间,用于时间管理
      volatile struct xLIST_ITEM * pxPrevious;          //指向链表上一个节点
      volatile struct xLIST_ITEM * pxNext;              //指向链表下一个节点
      void * prContainer;                               //指向本链表节点所在的链表
      void * pvOwner;                                   //指向链表节点指向的任务控制块
};
```

任务创建完成之后，系统会随之创建相应的任务控制块(Task Control Block，TCB)，TCB 中记录了与该任务相关的详细信息，如当前任务名称、优先级、堆栈栈顶指针以及指向下一个 TCB 的指针等。在进行任务调度时，系统会将一个指定任务插入就绪任务链表中。需要注意的是，此时系统并不是将一个指针插入 TCB 中，而是找到该任务对应的 TCB 中的 xGenericListItem 链表节点，然后将该节点插入到任务就绪链表中。链表节点中的 prContainer、pvOwner 则分别记录了该链表节点所属的就绪任务链表和 TCB，使得 FreeRTOS 系统的链表结构使用起来更加灵活方便。

在任务进行调度时，FreeRTOS 会首先采用基于优先级调度的算法，先在任务就绪链表组 pxReadyTasksLists 中查找到第一个 uxNumberOfItems 不为 0 的就绪任务链表，该链表中的任务即为当前系统中优先级最高的任务。若该优先级对应就绪链表中只有一个任务，

则直接运行该任务。若该优先级对应就绪链表中有多个任务，则采用基于时间片的轮转调度算法依次运行所有任务。任务执行结束后，再次在就绪状态的下一级优先队列中重复上述操作，任务转换流程如图 3-5 所示。

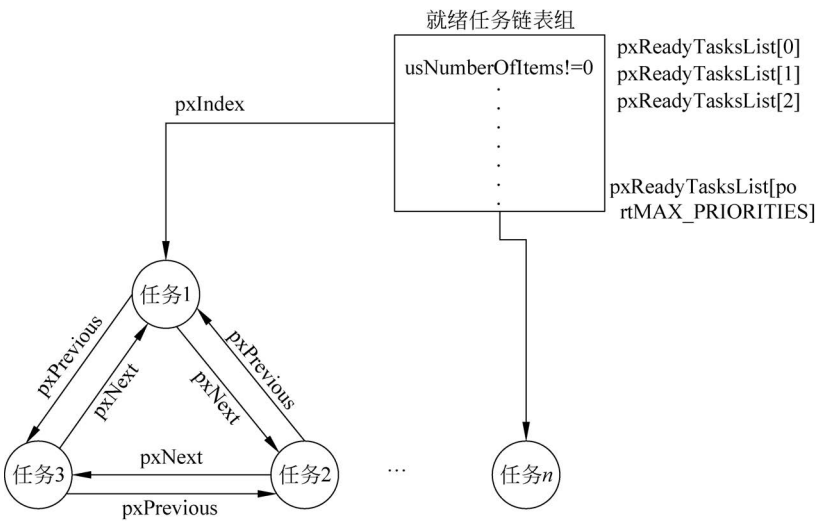

图 3-5　任务转换流程

同时，为了加快任务调度速度，系统专门设置了 uxTopUsedPriority 这一参数，用以跟踪系统中所有任务的最高优先级。在任务加入就绪任务链表之前，系统会将当前 uxTopUsedPriority 参数的值与该任务优先级进行比较，并记录下当前的最高优先级。每个新任务进入就绪态后，最高优先级会进行实时更新。在进行任务调度时，系统便可直接从该最高优先级进行查找，不用从头开始遍历就绪链表组，以此达到加快搜索并缩短内核中断时间的效果。

任务执行时，使用处理器的寄存器、堆栈等资源环境信息被称为上下文信息。在任务调度时 FreeRTOS 需要保存相应的处理器的寄存器与堆栈等环境数据，以便进行上下文的切换。在 FreeRTOS 中，系统会调用 portSAVE_CONTEXT() 函数完成上下文保存。FreeRTOS 为每个任务都分配了独立的堆栈，切换上下文时，系统只需要将每个任务的寄存器值压入各自的堆栈中就能实现上下文的切换工作。

FreeRTOS 默认采用的调度算法是为处于同一优先级的每个任务分配确定的时间片。但实际上，固定大小的时间片无法满足任务所需要的不同时间。为了获取足够的执行时间，系统只能高频率地转换任务，直至执行结束。在这种情况下，系统的执行效率会大大降低。并且，随着相同优先级的任务数量迅速增多，系统将表现得更低效。因此，如果可以指定每个任务的时间片长度，减少任务之间的切换频率，就能够提高 CPU 执行的效率，进而减少任务执行总时间。

3.4　本章小结

本章主要阐述了嵌入式实时操作系统 FreeRTOS 的特点，为介绍本书的物联网软件开发案例做好嵌入式方面的知识铺垫。具体来讲，本章首先概述了 FreeRTOS 的基本概念，

使读者建立起对该操作系统的初步认识；然后介绍了 FreeRTOS 的 4 个模块，即任务管理模块、时间管理模块、内存管理模块以及协同例程管理模块，使读者对 FreeRTOS 的体系结构有了较为清晰的了解；最后讲解了 FreeRTOS 调度机制的原理，至此为读者形成了对 FreeRTOS 的整体认知。

3.5 课后习题

1. 知识点考查

(1) FreeRTOS 操作系统的主要优点有哪些？

(2) FreeRTOS 操作系统的主要功能模块有哪些？各模块有哪些重要的操作？

(3) 从创建任务函数、删除任务函数、延时函数、写入队列函数中选择一个，说明关键函数并尝试用 FreeRTOS 系统进行基础的操作练习。

(4) FreeRTOS 任务调度机制的主要原理是什么？

(5) FreeRTOS 的任务调度机制有哪些优点和缺点？请尝试针对其不足之处，设计新的调度算法，并试着进行验证。

2. 拓展阅读

[1] 何立民. 嵌入式系统的定义与发展历史[J]. 单片机与嵌入式系统应用，2004(1)：6-8.

[2] 张朝. 多核嵌入式实时操作系统(RTOS)综述[J]. 电脑知识与技术，2015，11(12)：248-250.

[3] 夏恒发，黄俊. 物联网终端操作系统中任务调度的研究与设计[J]. 信息通信，2018(2)：168-170.