

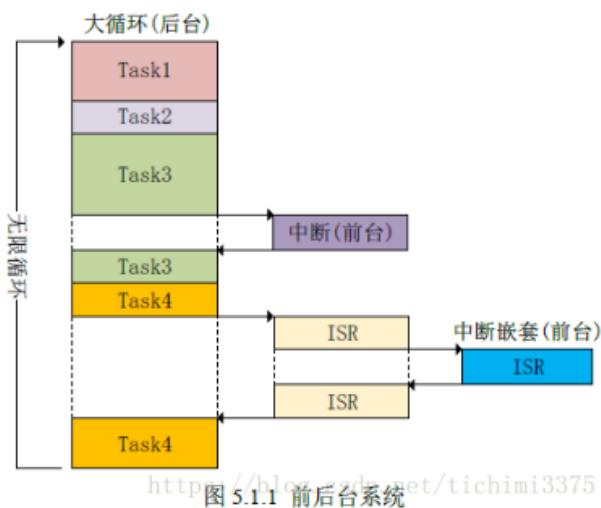
FreeRTOS [TODO]

FreeRTOS

RTOS 系统的核心就是任务管理，FreeRTOS 也不例外，而且大多数学习RTOS系统的工程师或者学生主要就是为了使用RTOS 的多任务处理功能，初步上手RTOS系统首先必须掌握的也是任务的创建、删除、挂起和恢复等操作，由此可见任务管理的重要性。

什么是多任务系统？

回想一下我们以前在使用51、AVR、STM32 单片机裸机(未使用系统)的时候一般都是在main 函数里面用while(1)做一个大循环来完成所有的处理，即应用程序是一个无限的循环，循环中调用相应的函数完成所需的处理。有时候我们也需要中断中完成一些处理。相对于多任务系统而言，这个就是单任务系统，也称作前后台系统，中断服务函数作为前台程序，大循环while(1)作为后台程序，如图5.1.1 所示：



前后台系统的实时性差，前后台系统各个任务(应用程序)都是排队等着轮流执行，不管你这个程序现在有多紧急，没轮到你就只能等着！相当于所有任务(应用程序)的优先级都是一样的。但是前后台系统简单啊，资源消耗也少啊！在稍微大一点的嵌入式应用中前后台系统就明显力不从心了，此时就需要多任务系统出马了。多任务系统会把一个大问题(应用)“分而治之”，把大问题划分成很多个小问题，逐步的把小问题解决掉，大问题也就随之解决了，这些小问题可以单独的作为一个小任务来处理。这些小任务是并发处理的，注意，并不是说同一时刻一起执行很多个任务，而是由于每个任务执行的时间很短，导致看起来像是同一时刻执行了很多个任务一样。多个任务带来了一个新的问题，究竟哪个任务先运行，哪个任务后运行呢？完成这个功能的东西在RTOS 系统中叫做任务调度器。不同的系统其任务调度器的实现方法也不同，比如FreeRTOS 是一个抢占式的实时多任务系统，那么其任务调度器也是抢占式的，运行过程如图5.1.2 所示：

在图5.1.2 中，高优先级的任务可以打断低优先级任务的运行而取得CPU 的使用权，这样就保证了那些紧急任务的运行。这样我们就可以为那些对实时性要求高的任务设置一个很高的优先级，比如自动驾驶中的障碍物检测任务等。高优先级的任务执行完成以后重新把CPU 的使用权归还给低优先级的任务，这个就是抢占式多任务系统的基本原理。

在图5.1.2 中，高优先级的任务可以打断低优先级任务的运行而取得CPU 的使用权，这样就保证了那些紧急任务的运行。这样我们就可以为那些对实时性要求高的任务设置一个很高的优先级，比如自动驾驶中的障碍物检测任务等。高优先级的任务执行完成以后重新把CPU 的使用权归还给低优先级的任务，这个就是抢占式多任务系统的基本原理。

5.2 FreeRTOS 任务与协程

再FreeRTOS 中应用既可以使用任务，也可以使用协程(Co-Routine)，或者两者混合使用。但是任务和协程使用不同的API 函数，因此不能通过队列(或信号量)将数据从任务发送给协程，反之亦然。协程是为那些资源很少的MCU 准备的，其开销很小，但是FreeRTOS 官方已经不打算再更新协程了，所以本教程只讲解任务。

5.2.1 任务(Task)的特性

在使用RTOS 的时候一个实时应用可以作为一个独立的任务。每个任务都有自己的运行环境，不依赖于系统中其他的任务或者RTOS 调度器。任何一个时间点只能有一个任务运行，具体运行哪个任务是由RTOS 调度器来决定的，RTOS 调度器

因此就会重复的开启、关闭每个任务。任务不需要了解**RTOS** 调度器的具体行为，**RTOS** 调度器的职责是确保当一个任务开始执行的时候其上下文环境(寄存器值，堆栈内容等)和任务上一次退出的时候相同。为了做到这一点，每个任务都必须有个堆栈，当任务切换的时候将上下文环境保存在堆栈中，这样当任务再次执行的时候就可以从堆栈中取出上下文环境，任务恢复运行。

任务特性：

- 1、简单。
- 2、没有使用限制。
- 3、支持抢占
- 4、支持优先级
- 5、每个任务都拥有堆栈导致了**RAM** 使用量增大。

6、如果使用抢占的话的必须仔细的考虑重入的问题。

5.2.2 协程(Co-routine)的特性

协程是为那些资源很少的**MCU** 而做的，但是随着**MCU** 的飞速发展，性能越来越强大，现在协程几乎很少用到了！但是**FreeRTOS** 目前还没有把协程移除的计划，但是**FreeRTOS** 是绝对不会更新和维护协程了，因此协程大家了解一下就行了。在概念上协程和任务是相似的，但是有如下根本上的不同：

1、堆栈使用

所有的协程使用同一个堆栈(如果是任务的话每个任务都有自己的堆栈)，这样就比使用任务消耗更少的**RAM**。

2、调度器和优先级

协程使用合作式的调度器，但是可以在使用抢占式的调度器中使用协程。

3、宏实现

协程是通过宏定义来实现的。

4、使用限制

为了降低对**RAM** 的消耗做了很多的限制。

FreeRTOS 操作系统支持三种调度方式：抢占式调度，时间片调度和合作式调度。实际应用主要是抢占式调度和时间片调度，合作式调度用到的很少。

(1) 抢占式调度

每个任务都有不同的优先级，任务会一直运行直到被高优先级任务抢占或者遇到阻塞式的API函数，比如**vTaskDelay**。

(2) 时间片调度

每个任务都有相同的优先级，任务会运行固定的时间片个数或者遇到阻塞式的API函数，比如**vTaskDelay**，才会执行同优先级任务之间的任务切换。

简单的说，调度器就是使用相关的调度算法来决定当前需要执行的任务。所有的调度器有一个共同的特性：

A. 调度器可以区分就绪态任务和挂起任务（由于延迟，信号量等待，邮箱等待，事件组等待等原因而使得任务被挂起）。

B. 调度器可以选择就绪态中的一个任务，然后激活它（通过执行这个任务）。当前正在执行的任务是运行态的任务。

C. 不同调度器之间最大的区别就是如何分配就绪态任务间的完成时间。

嵌入式实时操作系统的核心就是调度器和任务切换，调度器的核心就是调度算法。任务切换的实现在不同的嵌入式实时操作系统中区别不大，基本相同的硬件内核架构，任务切换也是相似的。调度算法就有些区别了。下面我们主要了解一下抢占式调度器和时间片调度器。

抢占式调度器基本概念

在实际的应用中，不同的任务需要不同的响应时间。例如，我们在一个应用中需要使用电机，键盘和**LCD**显示。电机比键盘和**LCD**需要更快速的响应，如果我们使用合作式调度器或者时间片调度，那么电机将无法得到及时的响应，这时抢占式调度是必须的。

如果使用了抢占式调度，最高优先级的任务一旦就绪，总能得到**CPU**的控制权。比如，当一个运行着的任务被其它高优先级的任务抢占，当前任务的**CPU**使用权就被剥夺了，或者说被挂起了，那个高优先级的任务立刻得到了**CPU**的控制权并运行。又比如，如果中断服务程序使一个高优先级的任务进入就绪态，中断完成时，被中断的低优先级任务被挂起，优先级高的那个任务开始运行。

使用抢占式调度器，使得最高优先级的任务什么时候可以得到**CPU**的控制权并运行是可知的，同时使得任务级响应时

间得以最优化。

总的来说，学习抢占式调度要掌握的最关键一点是：每个任务都被分配了不同的优先级，抢占式调度器会获得就绪列表中优先级最高的任务，并运行这个任务。

□如果用户在FreeRTOS的配置文件FreeRTOSConfig.h中禁止使用时间片调度，那么每个任务必须配置不同的优先级。当FreeRTOS多任务启动执行后，基本会按照如下的方式去执行：

□首先执行的最高优先级的任务Task1，Task1会一直运行直到遇到系统阻塞式的API函数，比如延迟，事件标志等待，信号量等待，Task1任务会被挂起，也就是释放CPU的执行权，让低优先级的任务得到执行。

□FreeRTOS操作系统继续执行任务就绪列表中下一个最高优先级的任务Task2，Task2执行过程中有两种情况：

 Task1由于延迟时间到，接收到信号量消息等方面的原因，使得Task1从挂起状态恢复到就绪态，在抢占式调度器的作用下，Task2的执行会被Task1抢占。

 Task2会一直运行直到遇到系统阻塞式的API函数，比如延迟，事件标志等待，信号量等待，Task2任务会被挂起，继而执行就绪列表中下一个最高优先级的任务。

□如果用户创建了多个任务并且采用抢占式调度器的话，基本都是按照上面两条来执行。根据抢占式调度器，当前的任务要么被高优先级任务抢占，要么通过调用阻塞式API来释放CPU使用权让低优先级任务执行，没有用户任务执行时就执行空闲任务。

□如下框图说明抢占式调度在FreeRTOS中的运行过程。

□创建3个任务Task1，Task2和Task3。Task1的优先级为1，Task2的优先级为2，Task3的优先级为3。FreeRTOS操作系统是设置的数值越小任务优先级越低，故Task3的优先级最高，Task1的优先级最低。

□任务Task1在运行中，运行过程中由于Task2就绪，在抢占式调度器的作用下任务Task2抢占Task1的执行。Task2进入到运行态，Task1由运行态进入到就绪态。

□Task2在运行中，运行过程中由于Task3就绪，在抢占式调度器的作用下任务Task3抢占Task2的执行。Task3进入到运行态，Task2由运行态进入到就绪态。

□任务Task3运行过程中调用了阻塞式API函数，比如vTaskDelay，任务Task3被挂起，在抢占式调度器的作用下查找到下一个要执行的最高优先级任务是Task2，任务Task2由就绪态进入到运行态。

□任务Task2在运行中，运行过程中由于Task3再次就绪，在抢占式调度器的作用下任务Task3抢占Task2的执行。Task3进入到运行态，Task2由运行态进入到就绪态。

时间片调度器基本概念

在小型的嵌入式RTOS中，最常用的时间片调度算法就是Round-robin调度算法。这种调度算法可以用于抢占式或者合作式的多任务中。另外，时间片调度适合用于不要求任务实时响应的情况。

实现Round-robin调度算法需要给同优先级的任务分配一个专门的列表，用于记录当前就绪的任务，并为每个任务分配一个时间片（也就是需要运行的时间长度，时间片用完了就进行任务切换）。

□在FreeRTOS操作系统中只有同优先级任务才会使用时间片调度，另外还需要用户在FreeRTOSConfig.h文件中使能宏定义：

```
#define configUSE_TIME_SLICING 1
```

□创建4个同优先级任务Task1，Task2，Task3和Task4。

□每个任务分配的时间片大小是5个系统时钟节拍。

□任务Task3在运行期间调用了阻塞式API函数，调用函数时，虽然5个系统时钟节拍的时间片大小还没有用完，此时依然会通过时间片调度切换到下一个任务Task4。（注意，没有用完的时间片不会再使用，下次任务Task3得到执行还是按照5个系统时钟节拍运行）

□任务Task4运行够5个系统时钟节拍后，通过时间片调度切换到任务Task1。

FreeRTOS 中的任务永远处于下面几个状态中的某一个：

- 运行态

当一个任务正在运行时，那么就说这个任务处于运行态，处于运行态的任务就是当前正在使用处理器的任务。如果使用的 是单核处理器的话那么不管在任何时刻永远都只有一个任务处于运行态。

- 就绪态

处于就绪态的任务是那些已经准备就绪(这些任务没有被阻塞或者挂起)，可以运行的任务，但是处于就绪态的任务还没有运行，因为有一个同优先级或者更高优先级的任务正在运行！

- 阻塞态

如果一个任务当前正在等待某个外部事件的话就说它处于阻塞态，比如说如果某个任务调用了函数vTaskDelay()的话就会进入阻塞态，直到延时周期完成。任务在等待队列、信号量、事件组、通知或互斥信号量的时候也会进入阻塞态。任务进入阻塞态会有一个超时时间，当超过这个超时时间任务就会退出阻塞态，即使所等待的事件还没有来临！

- 挂起态

像阻塞态一样，任务进入挂起态以后也不能被调度器调用进入运行态，但是进入挂起态的任务没有超时时间。任务进入和退出挂起态通过调用函数vTaskSuspend()和xTaskResume()。

RTOS

系统的核心就是任务管理，FreeRTOS 也不例外

多任务处理功能

任务的创建、删除、挂起和恢复等操作

```
// Perform an action every 10 ticks.
void vTaskFunction( void * pvParameters )
{
    TickType_t xLastWakeTime;
    const TickType_t xFrequency = 10;

    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount ();
    for( ;; )
    {
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, xFrequency );

        // Perform action here.
    }
}
```

多任务系统

单片机系统一般在main 函数里面用while(1)做一个大循环来完成所有的处理.单任务系统

任务创建、删除、返回值

创建

```
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,
    const char * const pcName,
    unsigned short usStackDepth,
    void *pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t * pvCreatedTask
);
```

- **pvTaskCode:**指针，指向任务函数的入口。任务永远不会返回（位于死循环内）。该参数类型TaskFunction_t定义在文件projdefs.h中，定义为：

```
typedef void (TaskFunction_t)( void ).
```

- **pcName**：任务描述。主要用于调试。字符串的最大长度由宏**configMAX_TASK_NAME_LEN**指定，该宏位于**FreeRTOSConfig.h**文件中。
- **usStackDepth**：指定任务堆栈大小，能够支持的堆栈变量数量，而不是字节数。比如，在**16**位宽度的堆栈下，**usStackDepth**定义为**100**，则实际使用**200**字节堆栈存储空间。堆栈的宽度乘以深度必须不超过**size_t**类型所能表示的最大值。比如，**size_t**为**16**位，则可以表示的最大值是**65535**。
- **pvParameters**：指针，当任务创建时，作为一个参数传递给任务。
- **uxPriority**：任务的优先级。具有**MPU**支持的系统，可以通过置位优先级参数的**portPRIVILEGE_BIT**位，随意的在特权（系统）模式下创建任务。比如，创建一个优先级为**2**的特权任务，参数**uxPriority**可以设置为(**2 | portPRIVILEGE_BIT**)。
- **pvCreatedTask**：用于回传一个句柄（ID），创建任务后可以使用这个句柄引用任务。

* 返回值 *

如果任务成功创建并加入就绪列表函数返回**pdPASS**，否则函数返回错误码。

errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY：任务创建失败，堆内存不足。具体参见**projdefs.h**。

删除

> void vTaskDelete(TaskHandle_t xTask);

从**RTOS**内核管理器中删除一个任务。任务删除后将会从就绪、阻塞、暂停和事件列表中移除。在文件**FreeRTOSConfig.h**中，必须定义宏**INCLUDE_vTaskDelete** 为**1**，本函数才有效。

注：被删除的任务，其在任务创建时由内核分配的存储空间，会由空闲任务释放。如果有应用程序调用**xTaskDelete()**，必须保证空闲任务获取一定的微控制器处理时间。任务代码自己分配的内存是不会自动释放的，因此删除任务前，应该将这些内存释放。

用法

```
/* 创建任务. */
void vTaskCode( void * pvParameters )
{
    for( ;; )
    {
        /* 任务代码放在这里 */
    }
}
/* 创建任务函数 */
void vOtherFunction( void )
{
    static unsigned char ucParameterToPass;
    xTaskHandle xHandle;

    /* 创建任务，存储句柄。注：传递的参数ucParameterToPass必须和任务具有相同的生存周期，因此这里定义为静态变量。如果它只是一个自动变量，可能不会有太长的生存周期，因为中断和高优先级任务可能会用到它。 */
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE,&ucParameterToPass, tskIDLE_PRIORITY, &xHandle );

    /* 使用句柄删除任务. */
    if( xHandle !=NULL )
    {
        vTaskDelete( xHandle );
    }
}
```

任务控制API

```
> void vTaskDelay( portTickTypexTicksToDelay )
```

- 调用vTaskDelay()函数后，任务会进入阻塞状态，持续时间由vTaskDelay()函数的参数xTicksToDelay指定，单位是系统节拍时钟周期。常量portTICK_RATE_MS 用来辅助计算真实时间，此值是系统节拍时钟中断的周期，单位是毫秒。在文件FreeRTOSConfig.h中，宏INCLUDE_vTaskDelay 必须设置成1，此函数才能有效。

- vTaskDelay()指定的延时时间是从调用vTaskDelay()后开始计算的相对时间。比如vTaskDelay(100)，那么从调用vTaskDelay()后，任务进入阻塞状态，经过100个系统时钟节拍周期，任务解除阻塞。因此，vTaskDelay()并不适用与周期性执行任务的场合。此外，其它任务和中断活动，会影响到vTaskDelay()的调用（比如调用前高优先级任务抢占了当前任务），因此会影响任务下一次执行的时间。API函数vTaskDelayUntil()可用于固定频率的延时，它用来延时一个绝对时间。