title: 从0开始学FreeRTOS-1 top: false cover: false toc: true mathjax: false date: 2019-08-31 19:29:23 password: summary: tags:

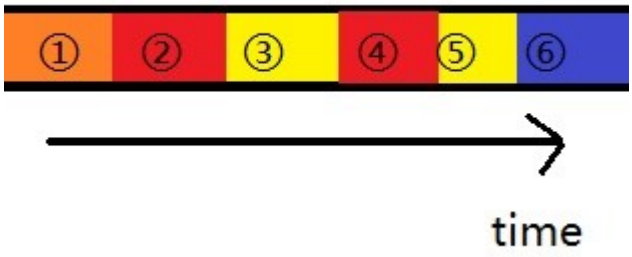
- FreeRTOS
- RTOS
- 操作系统 categories: 操作系统

我们知道，（单核）单片机某一时刻只能干一件事，会造成单片机资源的浪费，而且还有可能响应不够及时，所以，在比较庞大的程序或者是要求实时性比较高的情况下，我们可以移植操作系统。因为这种情况下操作系统比裸机方便很多，效率也高。下面，杰杰将带你们走进FreeRTOS的世界随便看看。

下面正式开始本文内容。

在没有用到操作系统之前，单片机的运行是顺序执行，就是说，很多时候，单片机在执行这件事的时候，无法切换到另一件事。这就造成了资源的浪费，以及错过了突发的信号。那么，用上了操作系统的时候，很容易避免了这样的问题。

很简单，从感觉上，单片机像是同时在干多件事，为什么说像呢，因为单片机的执行速度很快，快到我们根本没办法感觉出来，但是同时做两件事是不可能的，在（单核）单片机中，因为它的硬件结构决定了CPU只能在

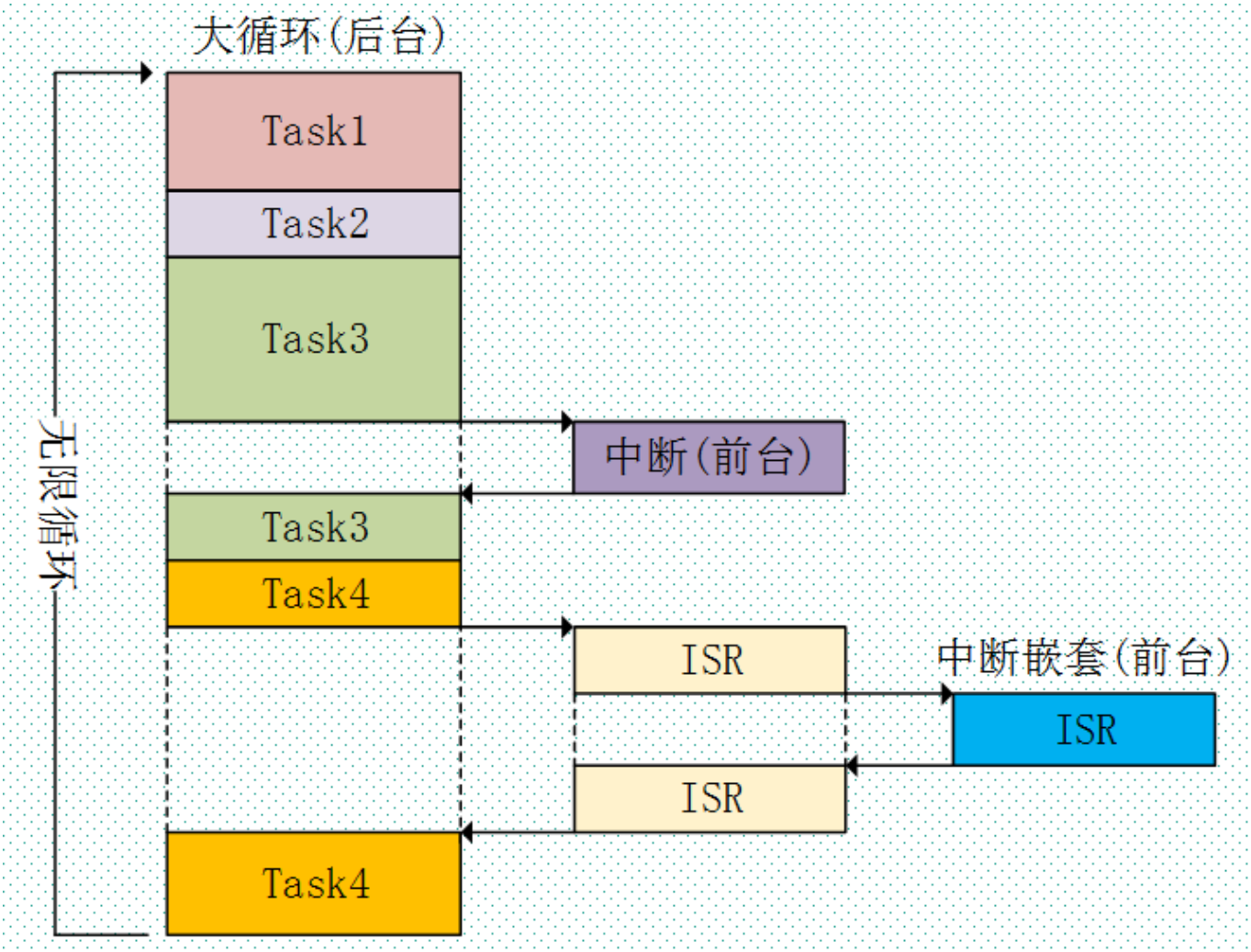


一个时间段做一件事如：

如这张图，都是按照顺序来执行这些事的，假设每个任务（事件）的time无限小，小到我们根本没法分辨出来，那么我们也会感觉单片机在同时做这六件事。

真相就是：所有任务都好像在执行，但实际上在任何一个时刻都只有一个任务在执行

如是加上了中断系统的话，就可以将上图理解为下图：



通常把程序分为两部分：前台系统和后台系统。 简单的小系统通常是前后台系统，这样的程序包括一个死循环和若干个中断服务程序：应用程序是一个无限循环，循环中调用API函数完成所需的操作，这个大循环就叫做后台系统。中断服务程序用于处理系统的异步事件，也就是前台系统。前台是中断级，后台是任务级。简单来说就是程序一直按顺序执行，有中断来了就做中断（前台）的事情。处理完中断（前台）的事情，就回到大循环（后台）继续按顺序执行。

那么问题来了，这样子的系统肯定不是好的系统，我在做第一个任务的时候想做第四个任务，根本做不到啊，其实也能做到，让程序执行的指针cp指向第四个任务就行了。但是任务一旦复杂，那么整个工程的代码的结构，可移植性，及可读性，肯定会差啦。

FreeRTOS

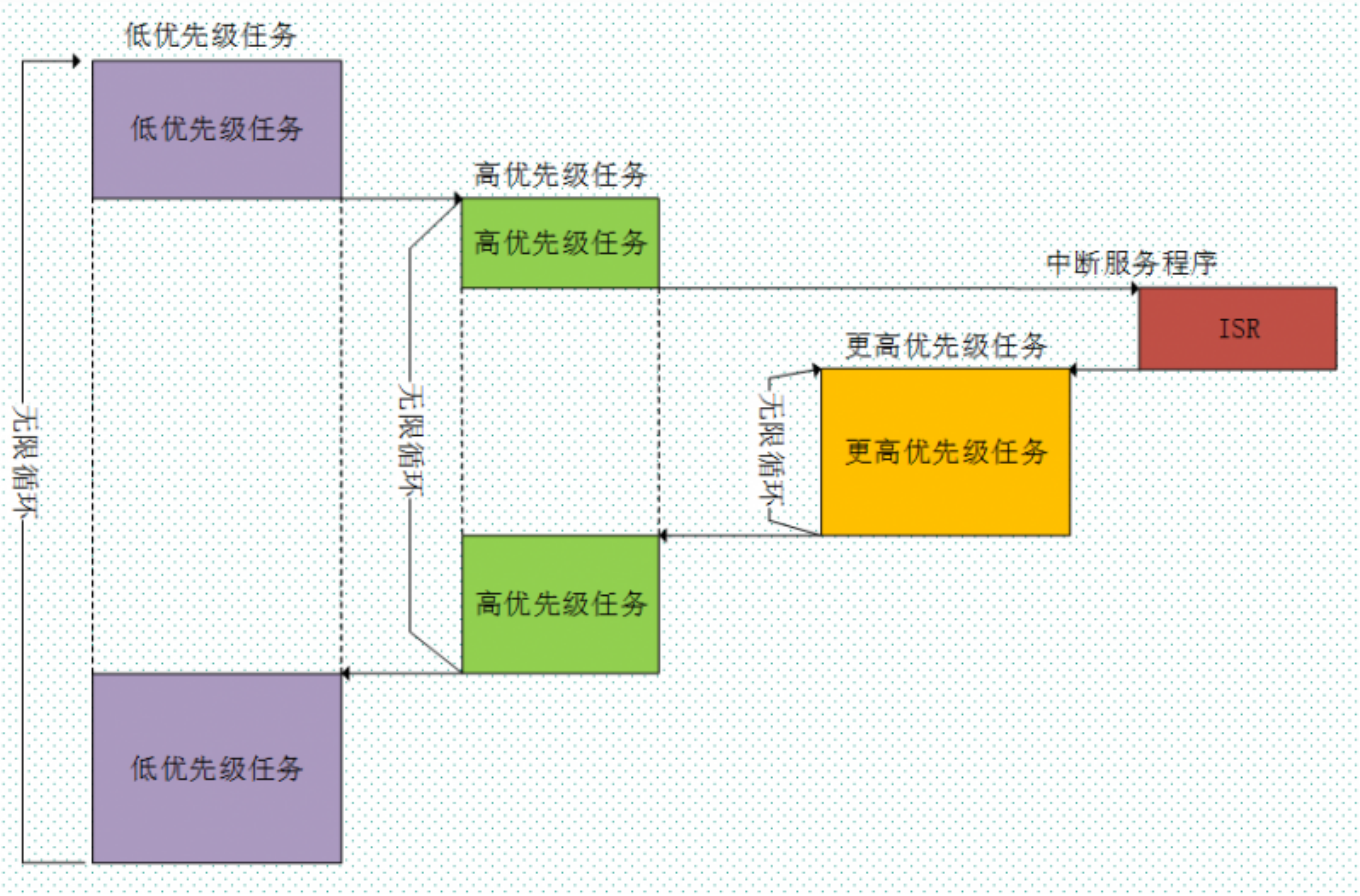
那么操作系统的移植就是不可或缺的了。什么叫RTOS？：Real Time OS，实时操作系统，强调的是实时性，就是要规定什么时间该做什么任务。那么假如同一个时刻，需要执行两个或者多个任务怎么办。那么我们可以人为地把任务划分优先级，哪个任务重要，就先做，因为前面一直强调，单片机无法同时做两件事，在某一个时刻只能做一件事。

那么FreeRTOS是怎么操作的呢？先看看FreeRTOS的内核吧：

FreeRTOS是一个可裁剪、可剥夺型的多任务内核，而且没有任务数限制。FreeRTOS提供了实时操作系统所需的所有功能，包括资源管理、同步、任务通信等。 FreeRTOS是用C和汇编来写的，其中绝大部分都是用C语言编写的，只有极少数的与处理器密切相关的部分代码才是用汇编写的，FreeRTOS结构简洁，可读性很强！RTOS的

内核负责管理所有的任务，内核决定了运行哪个任务，何时停止当前任务切换到其他任务，这个是内核的多任务管理能力。

可剥夺内核顾名思义就是可以剥夺其他任务的CPU使用权，它总是运行就绪任务中的优先级最高的那个任务。



在FreeRTOS中，每个任务都是无限循环的，一般来说任务是不会结束运行的，也不允许有返回值，任务的结构一般都是

```
While(1)
{
/****一直在循环执行*****/
}
```

如果不需要这个任务了，那就把它删除。

移植的教程我就不写了，超级简单的，按照已有的大把教程来做就行了。（如果没有资源，可以在后台找我，我给一份移植的教程/源码）

其实FreeRTOS的运用及其简单，移植成功按照自己的意愿来配置即可，而且FreeRTOS有很多手册，虽然作者英语很差，但是我有谷歌翻译！！！哈哈哈

既然一直都说任务任务，那肯定要有任务啊，创建任务：

```
// task. h  task.c
BaseType_t xTaskCreate(      TaskFunction_t pvTaskCode,
                             const char * const pcName,
```

```
                                uint16_t usStackDepth,
                                void *pvParameters,
                                UBaseType_t uxPriority,
                                TaskHandle_t *pvCreatedTask
                            );
```

函数的原型都有，按照字面的理解

```
TaskFunction_t pvTaskCode         //传递进来的是任务函数
const char * const pcName          //传递进来的是任务Name
uint16_t usStackDepth             //传入的是堆栈的大小
```

在这里要说明一下，在裸机中开发，我们不管局部变量还是全局变量，反正定义了就能用，中断发生时，函数返回地址发哪里，我们也不管。但是在操作系统中，我们必须弄清楚我们的参数是怎么储存的，他们的大小是多大，就需要我们去定义这个堆栈的大小。它就是用来存放我们的这些东西的。太小，导致堆栈溢出，发生异常。（栈是单片机 RAM 里面一段连续的内存空间）

因为在多任务系统中，每个任务都是独立的，互不干扰的，所以要为每个任务都分配独立的栈空间。

```
void *pvParameters               //传递给任务函数的参数
UBaseType_t uxPriority           //任务优先级
TaskHandle_t *pvCreatedTask      //任务句柄
```

任务句柄也是很重要的东西，我们怎么删除任务也是要用到任务句柄，其实说白了，我操作系统怎么知道你是什么任务，靠的就是任务句柄的判断，才知道哪个任务在执行，哪个任务被挂起。下一个要执行的任务是哪个等等，靠的都是任务句柄。

那么要使用这些东西，我们肯定要实现啦，下面就是实现的定义，要定义优先级，堆栈大小，任务句柄，任务函数等。

```
//任务优先级
#define LED_TASK_PRIO           2
//任务堆栈大小
#define LED_STK_SIZE             50
//任务句柄
TaskHandle_t LED_Task_Handler;
//任务函数
void LED_Task(void *pvParameters);
```

创建任务后，可以开启任务调度了，然后系统就开始运行。

```
xTaskCreate((TaskFunction_t )LED_Task,     //任务函数
            (const char*    )"led_task",   //任务名称
            (uint16_t       )LED_STK_SIZE, //任务堆栈大小
```

```
            (void*          )NULL, //传递给任务函数的参数
            (UBaseType_t    )START_TASK_PRIO, //任务优先级
            (TaskHandle_t*  )&LED_Task_Handler);//任务句柄
 vTaskStartScheduler();          //开启任务调度
```

这个创建任务的函数 xTaskCreate 是有返回值的，其返回值的类型是BaseType_t。

我们在描述中看看：

```
// @return pdPASS if the task was successfully created and added to a readylist, o
therwise an error code defined in the file projdefs.h
```

我们其实可以在任务调度的时候判断一下返回值是否为pdPASS从而知道任务创是否建成功。并且打印一个信息作为调试。因为后面使用信号量这些的时候都要知道信号量是否创建成功，使得代码健壮一些。免得有隐藏的bug。

然后就是具体实现我们的任务LED_Task是在做什么的

当然可以实现多个任务。还是很简单的。

```
//LED任务函数
void LED_Task(void *pvParameters)
{
    while(1)
    {
        LED0  =  !LED0;
        vTaskDelay(1000);
    }
}
```

这就是一个简单的操作系统的概述。

下一篇，应该是讲述开启任务调度与任务切换的具体过程。

这个可以参考野火的书籍《从 0 到 1 教你写 uCOS-III》

喜欢就关注我吧！

相关代码可以在公众号后台获取。