title: 从单片机到操作系统⑦——深入了解FreeRTOS的延时机制 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-04 09:23:58 img: coverImg: password: summary: tags:

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

> 没研究过操作系统的源码都不算学过操作系统

FreeRTOS 时间管理

时间管理包括两个方面：系统节拍以及任务延时管理。

系统节拍：

在前面的文章也讲得很多，想要系统正常运行，那么时钟节拍是必不可少的，`FreeRTOS`的时钟节拍通常由`SysTick`提供，它周期性的产生定时中断，所谓的时钟节拍管理的核心就是这个定时中断的服务程序。`FreeRTOS`的时钟节拍isr中核心的工作就是调用`vTaskIncrementTick()`函数。具体见上之前的文章。

延时管理

FreeRTOS提供了两个系统延时函数：

- 相对延时函数`vTaskDelay()`
- 绝对延时函数`vTaskDelayUntil()`。

这些延时函数可不像我们以前用裸机写代码的延时函数操作系统不允许CPU在死等消耗着时间，因为这样效率太低了。

同时，要告诫学操作系统的同学，千万别用裸机的思想去学操作系统。

任务延时

任务可能需要延时，两种情况，一种是任务被`vTaskDelay`或者`vTaskDelayUntil`延时，另外一种情况就是任务等待事件（比如等待某个信号量、或者某个消息队列）时候指定了`timeout`（即最多等待timeout时间，如果等待的事件还没发生，则不再继续等待），在每个任务的循环中都必须要有阻塞的情况出现，否则比该任务优先级低的任务就永远无法运行。

相对延时与绝对延时的区别

相对延时：vTaskDelay()：

相对延时是指每次延时都是从任务执行函数`vTaskDelay()`开始，延时指定的时间结束

绝对延时：vTaskDelayUntil()：

绝对延时是指调用`vTaskDelayUntil()`的任务每隔x时间运行一次。也就是任务周期运行。

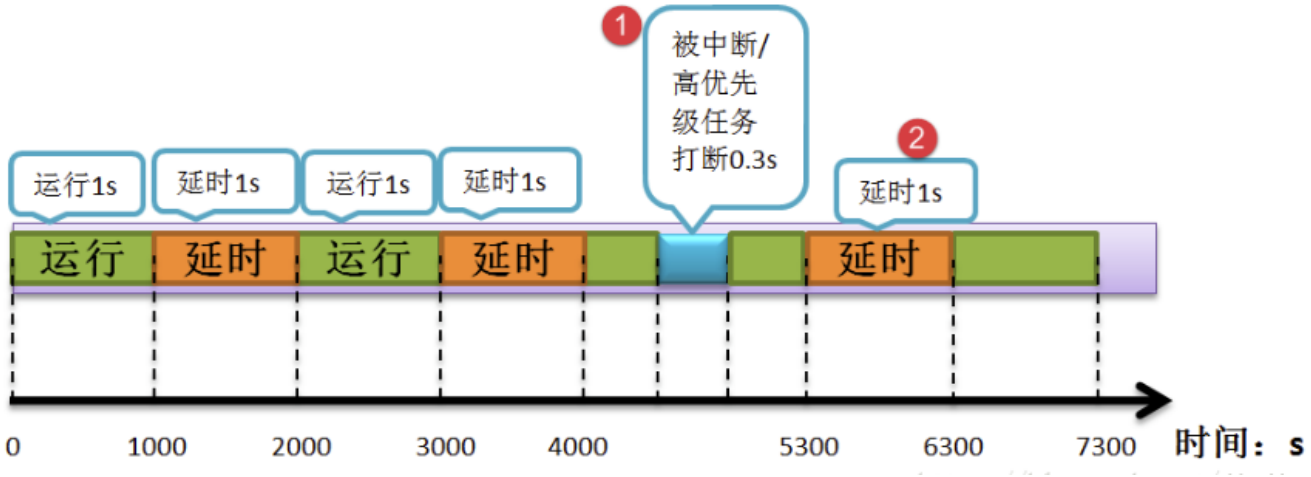
相对延时：vTaskDelay()

相对延时vTaskDelay()是从调用vTaskDelay()这个函数的时候开始延时，但是任务执行的时候，可能发生了中断，导致任务执行时间变长了，但是整个任务的延时时间还是1000个tick，这就不是周期性了，简单看看下面代码：

```
void vTaskA( void * pvParameters )
 {
        while(1)
     {
         //  ...
         //  这里为任务主体代码
         //  ...

         /* 调用相对延时函数,阻塞1000个tick */
         vTaskDelay( 1000 );
     }
}
```

可能说的不够明确，可以看看图解。



当任务运行的时候，假设被某个高级任务或者是中断打断了，那么任务的执行时间就更长了，然而延时还是延时1000个tick这样子，整个系统的时间就混乱了。

如果还不够明确，看看vTaskDelay()的源码

```
void vTaskDelay( const TickType_t xTicksToDelay )
{
    BaseType_t xAlreadyYielded = pdFALSE;

    /* 延迟时间为零只会强制切换任务。 */
    if( xTicksToDelay > ( TickType_t ) 0U )             (1)
    {
        configASSERT( uxSchedulerSuspended == 0 );
        vTaskSuspendAll();                                          (2)
        {
            traceTASK_DELAY();
            /*将当前任务从就绪列表中移除,并根据当前系统节拍
```

```
            计数器值计算唤醒时间,然后将任务加入延时列表 */
            prvAddCurrentTaskToDelayedList( xTicksToDelay, pdFALSE );
        }
        xAlreadyYielded = xTaskResumeAll();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    /* 强制执行一次上下文切换 */
    if( xAlreadyYielded == pdFALSE )
    {
        portYIELD_WITHIN_API();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
```

- (1):如果传递进来的延时时间是0，只能进行强制切换任务了，调用的是portYIELD_WITHIN_API()，它其实是一个宏，真正起作用的是portYIELD()，下面是它的源码：

```
#define portYIELD()
\
{
\
        /* 设置PendSV以请求上下文切换。 */
\
        portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;                         \
        __dsb( portSY_FULL_READ_WRITE );
\
        __isb( portSY_FULL_READ_WRITE );
\
}
```

- (2):挂起当前任务

然后将当前任务从就绪列表删除，然后加入到延时列表。是调用函数prvAddCurrentTaskToDelayedList()完成这一过程的。由于这个函数篇幅过长，就不讲解了，有兴趣可以看看，我就简单说说过程。在FreeRTOS中有这么一个变量，是用来记录systick的值的。

```
PRIVILEGED_DATA static volatile TickType_t xTickCount     = ( TickType_t ) 0U;
```

在每次tick中断时xTickCount加一，它的值表示了系统节拍中断的次数，那么啥时候唤醒被加入延时列表的任务呢？其实很简单，FreeRTOS的做法将xTickCount(当前系统时间) + xTicksToDelay(要延时的时间)即可。

当这个相对的延时时间到了之后就唤醒了，这个(xTickCount+ xTicksToDelay)时间会被记录在该任务的任务控制块中。

看到这肯定有人问，这个变量是TickType_t类型(32位)的，那肯定会溢出啊，没错，是变量都会有溢出的一天，可是FreeRTOS乃是世界第一的操作系统啊，FreeRTOS使用了两个延时列表：

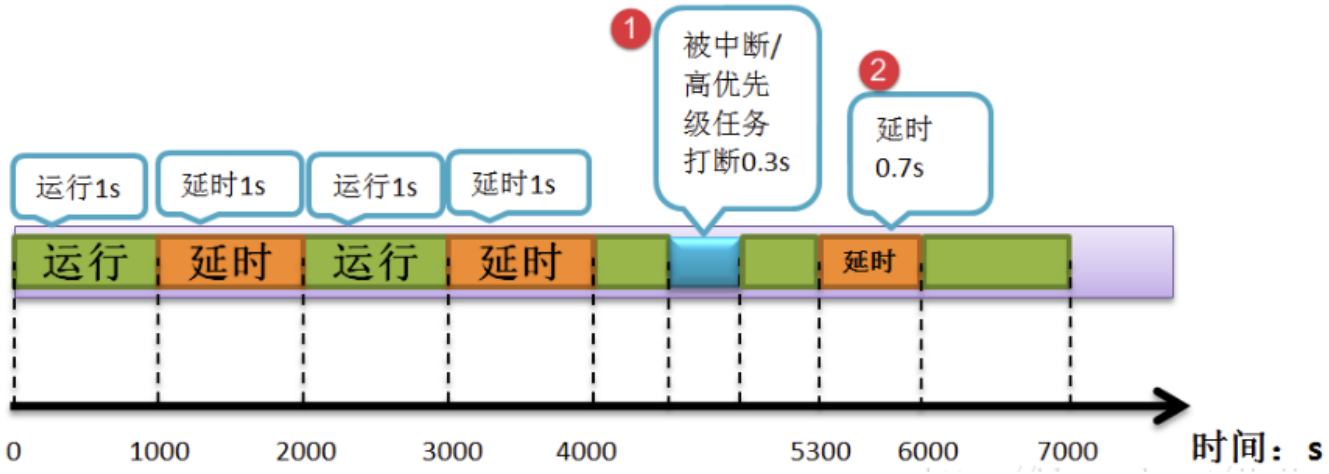
xDelayedTaskList1 和 xDelayedTaskList2

并使用两个列表指针类型变量pxDelayedTaskList和pxOverflowDelayedTaskList分别指向上面的延时列表1和延时列表2（在创建任务时将延时列表指针指向延时列表）如果内核判断出xTickCount+xTicksToDelay溢出，就将当前任务挂接到列表指针 pxOverflowDelayedTaskList指向的列表中，否则就挂接到列表指针pxDelayedTaskList指向的列表中。当时间到了，就会将延时的任务从延时列表中删除，加入就绪列表中，当然这时候就是由调度器觉得任务能不能运行了，如果任务的优先级大于当前运行的任务，那么调度器才会进行任务的调度。

绝对延时：vTaskDelayUntil()

vTaskDelayUntil()的参数指定了确切的滴答计数值

调用vTaskDelayUntil()是希望任务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。假设主体任务被打断0.3s，但是下次唤醒的时间是固定的，所以还是会周期运行。



下面看看vTaskDelayUntil()的使用方法，注意了，这vTaskDelayUntil()的使用方法与vTaskDelay()不一样：

```
void vTaskA( void * pvParameters )
{
    /* 用于保存上次时间。调用后系统自动更新 */
    static portTickType PreviousWakeTime;
    /* 设置延时时间，将时间转为节拍数 */
    const portTickType TimeIncrement = pdMS_TO_TICKS(1000);
    /* 获取当前系统时间 */
    PreviousWakeTime = xTaskGetTickCount();
    while(1)
     {

        /* 调用绝对延时函数,任务时间间隔为1000个tick */
```

```
            vTaskDelayUntil( &PreviousWakeTime, TimeIncrement );

            //  ...
            //  这里为任务主体代码
            //  ...

        }
}
```

在使用的时候要将延时时间转化为系统节拍，在任务主体之前要调用延时函数。

任务会先调用vTaskDelayUntil()使任务进入阻塞态，等到时间到了就从阻塞中解除，然后执行主体代码，任务主体代码执行完毕。会继续调用vTaskDelayUntil()使任务进入阻塞态，然后就是循环这样子执行。即使任务在执行过程中发生中断，那么也不会影响这个任务的运行周期，仅仅是缩短了阻塞的时间而已。

下面来看看vTaskDelayUntil()的源码：

```
void vTaskDelayUntil( TickType_t * const pxPreviousWakeTime, const TickType_t xTim
eIncrement )
{
    TickType_t xTimeToWake;
    BaseType_t xAlreadyYielded, xShouldDelay = pdFALSE;

    configASSERT( pxPreviousWakeTime );
    configASSERT( ( xTimeIncrement > 0U ) );
    configASSERT( uxSchedulerSuspended == 0 );

    vTaskSuspendAll();                                  // (1)
    {
        /* 保存系统节拍中断次数计数器 */
        const TickType_t xConstTickCount = xTickCount;

        /* 生成任务要唤醒的滴答时间。*/
        xTimeToWake = *pxPreviousWakeTime + xTimeIncrement;

        /* pxPreviousWakeTime中保存的是上次唤醒时间,唤醒后需要一定时间执行任务主体代码,
            如果上次唤醒时间大于当前时间,说明节拍计数器溢出了 具体见图片 */
        if( xConstTickCount < *pxPreviousWakeTime )
        {
            /* 由于此功能，滴答计数已溢出持续呼唤。 在这种情况下，我们唯一的时间实际延迟
是如果唤醒时间也溢出，
                唤醒时间大于滴答时间。 当这个就是这样，好像两个时间都没有溢出。*/

            if( ( xTimeToWake < *pxPreviousWakeTime ) && ( xTimeToWake > xConstTick
Count ) )
            {
                xShouldDelay = pdTRUE;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
```

```
            }
        }
        else
        {
            /* 滴答时间没有溢出。 在这种情况下，如果唤醒时间溢出，
               或滴答时间小于唤醒时间，我们将延迟。*/

            if( ( xTimeToWake < *pxPreviousWakeTime ) || ( xTimeToWake > xConstTick
Count ) )
            {
                xShouldDelay = pdTRUE;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
      }

      /* 更新唤醒时间,为下一次调用本函数做准备. */
      *pxPreviousWakeTime = xTimeToWake;

      if( xShouldDelay != pdFALSE )
      {
          traceTASK_DELAY_UNTIL( xTimeToWake );

          /* prvAddCurrentTaskToDelayedList（）需要块时间，而不是唤醒时间，因此减去当
前的滴答计数。 */
          prvAddCurrentTaskToDelayedList( xTimeToWake - xConstTickCount, pdFALSE )
;
      }
      else
      {
          mtCOVERAGE_TEST_MARKER();
      }
  }
  xAlreadyYielded = xTaskResumeAll();

  /* 如果xTaskResumeAll尚未执行重新安排，我们可能会让自己入睡。*/
  if( xAlreadyYielded == pdFALSE )
  {
    portYIELD_WITHIN_API();
  }
  else
  {
    mtCOVERAGE_TEST_MARKER();
  }
}
```

与相对延时函数vTaskDelay不同，本函数增加了一个参数pxPreviousWakeTime用于指向一个变量，变量保存上次任务解除阻塞的时间，此后函数vTaskDelayUntil()在内部自动更新这个变量。由于变量xTickCount可能会溢出，所以程序必须检测各种溢出情况，并且要保证延时周期不得小于任务主体代码执行时间。

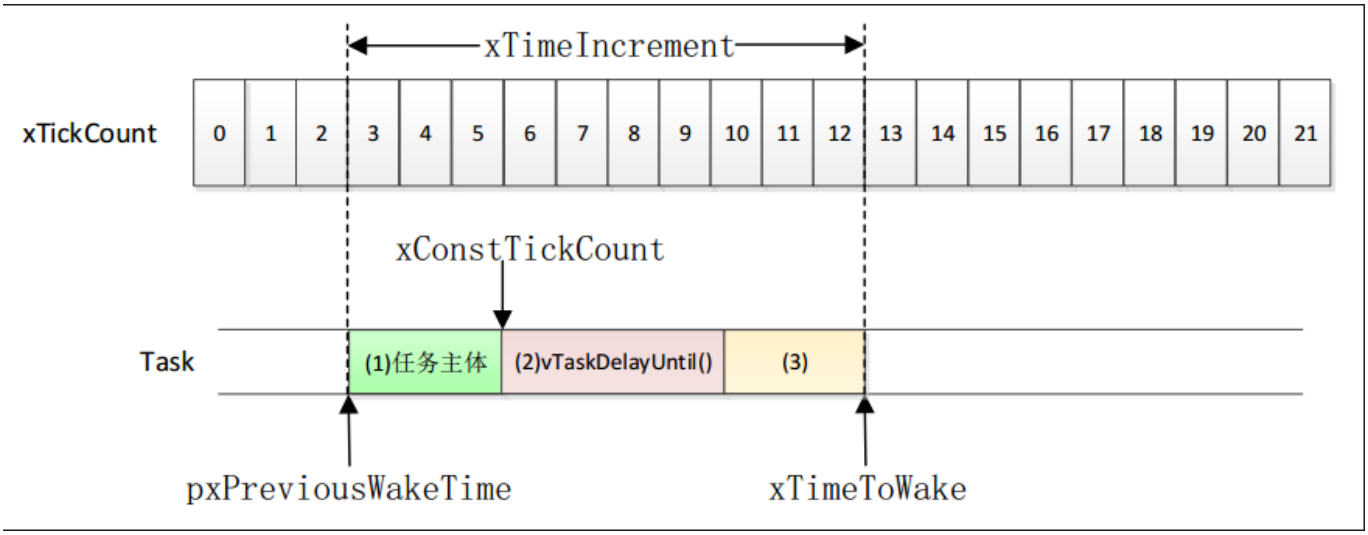
就会有以下3种情况，才能将任务加入延时链表中。

请记住这几个单词的含义：

- `xTimeIncrement`：任务周期时间
- `pxPreviousWakeTime`：上一次唤醒的时间点
- `xTimeToWake`：下一次唤醒的系统时间点
- `xConstTickCount`：进入延时的时间点

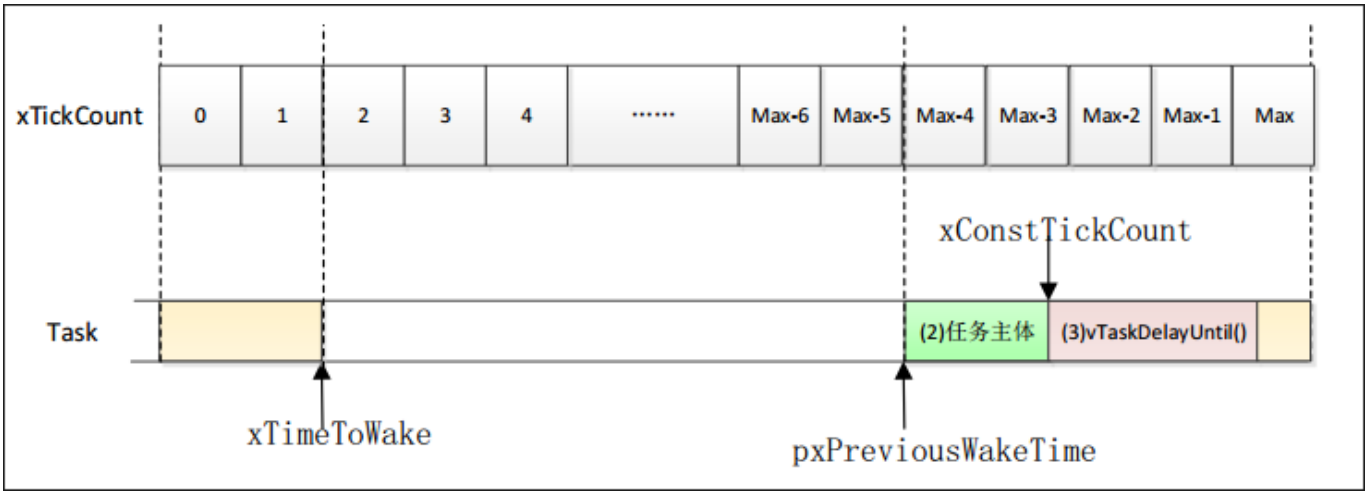
3. 第三种情况：常规无溢出的情况。

以时间为横轴，上一次唤醒的时间点小于下一次唤醒的时间点，这是很正常的情况。



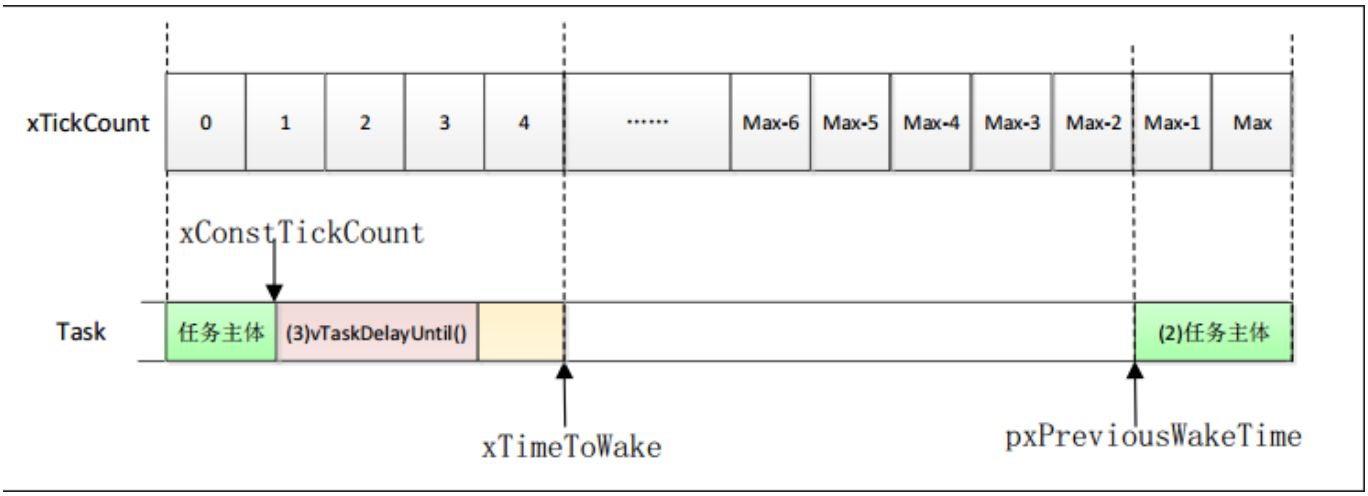
2. 第二种情况：唤醒时间计数器（`xTimeToWake`）溢出情况。

也就是代码中`if( ( xTimeToWake < *pxPreviousWakeTime ) || ( xTimeToWake > xConstTickCount ) )`



1. 第一种情况：唤醒时间（`xTimeToWake`）与进入延时的时间点（`xConstTickCount`）都溢出情况。

也就是代码中`if( ( xTimeToWake < *pxPreviousWakeTime ) && ( xTimeToWake > xConstTickCount ) )`



从图中可以看出不管是溢出还是无溢出，都要求在下次唤醒任务之前，当前任务主体代码必须被执行完。也就是说任务执行的时间不允许大于延时的时间，总不能存在每10ms就要执行一次20ms时间的任务吧。计算的唤醒时间合法后，就将当前任务加入延时列表，同样延时列表也有两个。每次系统节拍中断，中断服务函数都会检查这两个延时列表，查看延时的任务是否到期，如果时间到期，则将任务从延时列表中删除，重新加入就绪列表。如果新加入就绪列表的任务优先级大于当前任务，则会触发一次上下文切换。

总结

如果任务调用相对延时，其运行周期完全是不可测的，如果任务的优先级不是最高的话，其误差更大，就好比一个必须要在5ms内相应的任务，假如使用了相对延时1ms，那么很有可能在该任务执行的时候被更高优先级的任务打断，从而错过5ms内的相应，但是调用绝对延时，则任务会周期性将该任务在阻塞列表中解除，但是，任务能不能运行，还得取决于任务的优先级，如果优先级最高的话，任务周期还是比较精确的（相对vTaskDelay来说），如果想要更加想精确周期性执行某个任务，可以使用系统节拍钩子函数vApplicationTickHook()，它在tick中断服务函数中被调用，因此这个函数中的代码必须简洁，并且不允许出现阻塞的情况。

喜欢就关注我吧！

相关代码可以在公众号后台获取。