目录

简单的任务函数

```
void ATaskFunction( void *pvParameters )
{
    int iVariableExample = 0;

    /* 任务通常实现在一个死循环中。 */
    for( ;; )
    {
        /* 完成任务功能的代码将放在这里。 */
    }
    /* 如果任务的具体实现会跳出上面的死循环，则此任务必须在函数运行完之前删除。传入NULL参数表示删除的是当前任务 */
    vTaskDelete( NULL );
}
```

创建任务函数

```
/* 函数原型 */
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask );
/* 函数示例 */
static TaskHandle_t myTaskHandler = NULL;

xTaskCreate(ATaskFunction, "myTask", 512, NULL, configMAX_PRIORITIES - 4,
&myTaskHandler);
```

删除任务函数

```
/* 函数原型 */
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

任务优先级

低优先级号表示任务的优先级低，优先级号0表示最低优先级。有效的优先级号范围从0到`(configMAX_PRIORITES – 1)`。

函数	说明
vTaskPrioritySet()	可以动态的更改任务优先级
uxTaskPriorityGet()	获取任务优先级
uxTaskPriorityGetFromISR()	在ISR函数中获取任务优先级

延时函数

```
void vTaskDelay( portTickType xTicksToDelay );
```

> xTicksToDelay 表示延时多少个tick
> 该函数会把task切出到阻塞态

精确延时函数

```
void vTaskDelayUntil( portTickType * pxPreviousWakeTime, portTickType
xTimeIncrement );
```

> 调用此函数的任务解除阻塞的时间是绝对时刻，比起vTaskDelay()延时时间更精确

使用方法：

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    portTickType xLastWakeTime;
    pcTaskName = ( char * ) pvParameters;

    /* 变量xLastWakeTime需要被初始化为当前心跳计数值。说明一下，这是该变量唯一一次被显式
赋值。之后，
    xLastWakeTime将在函数vTaskDelayUntil()中自动更新。 */
    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        vPrintString( pcTaskName );
        /* 本任务将精确的以250毫秒为周期执行。同vTaskDelay()函数一样，时间值是以心跳周
期为单位的，
        可以使用常量portTICK_RATE_MS将毫秒转换为心跳周期。变量xLastWakeTime会在
        vTaskDelayUntil()中自动更新，因此不需要应用程序进行显示更新。 */
        vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
    }
}
```

空闲任务

> 调用vTaskStartScheduler()时，调度器会自动创建一个空闲任务
> 空闲任务拥有最低优先级(优先级0)
> 空闲任务是一个非常短小的循环

具体代码在vTaskStartScheduler()中可以看到：

```
void vTaskStartScheduler( void )
{
    BaseType_t xReturn;

    /* Add the idle task at the lowest priority. */
    #if ( INCLUDE_xTaskGetIdleTaskHandle == 1 )
    {
        /* Create the idle task, storing its handle in xIdleTaskHandle so it can
        be returned by the xTaskGetIdleTaskHandle() function. */
        xReturn = xTaskCreate( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE, ( void * )
NULL, ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), &xIdleTaskHandle ); /*lint !e961
MISRA exception, justified as it is not a redundant explicit cast to all supported
compilers. */
    }
    #else
    {
        /* Create the idle task without storing its handle. */
        xReturn = xTaskCreate( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE, ( void * )
NULL, ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), NULL );  /*lint !e961 MISRA
exception, justified as it is not a redundant explicit cast to all supported
compilers. */
    }
    #endif /* INCLUDE_xTaskGetIdleTaskHandle */

    ...
}
```

队列

创建队列

```
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,unsigned
portBASE_TYPE uxItemSize );
```

写入到队列首

```
portBASE_TYPE xQueueSendToFront( xQueueHandle xQueue,
                                 const void * pvItemToQueue,
                                 portTickType xTicksToWait );
```

写入到队列尾

```
portBASE_TYPE xQueueSendToBack( xQueueHandle xQueue,
                                const void * pvItemToQueue,
                                portTickType xTicksToWait );
```

发送队列

- 协程--croutine.c

系统运行过程

协程--croutine.c

4个链表：readyList pendingList delayList（2个）

链表设计--List.c

系统调度

任务管理

内存管理

优先级反转

有优先级为A、B和C三个任务，优先级A>B>C，任务A，B处于挂起状态，等待某一事件发生，任务C正在运行，此时任务C开始使用某一共享资源S。在使用中，任务A等待事件到来，任务A转为就绪态，因为它比任务C优先级高，所以立即执行。当任务A要使用共享资源S时，由于其正在被任务C使用，因此任务A被挂起，任务C开始运行。如果此时任务B等待事件到来，则任务B转为就绪态。由于任务B优先级比任务C高，因此任务B开始运行，直到其运行完毕，任务C才开始运行。直到任务C释放共享资源S后，任务A才得以执行。在这种情况下，优先级发生了翻转，任务B先于任务A运行。

优先级继承

> freertos 使用优先级继承来解决优先级反转问题 当任务A 申请共享资源S 时， 如果S正在被任务C 使用，通过比较任务C 与自身的优先级，如发现任务C 的优先级小于自身的优先级， 则将任务C的优先级提升到自身的优先级， 任务C 释放资源S 后，再恢复任务C 的原优先级。

freertos HOOK函数

- vApplicationIdleHook() idle的hook函数
- vApplicationTickHook() 系统tick的hook函数
- vApplicationMallocFailedHook() malloc失败的hook函数
- vApplicationStackOverflowHook() 栈溢出的hook函数
- vApplicationDaemonTaskStartupHook() vTaskStartScheduler()函数第一次执行的时候调用。

freertos栈溢出检测的2种方法

- 第一种 每次调度都检查一下任务的当前栈有没有超过任务的栈区域。
- 第二种 每个任务栈最下面会有16个字节是特定的内容，每次调度的时候如果发现这16个字节不正常了，就说明有谁踩到了这个栈。