freeRTOS

使用教学

淘宝店铺：

PC 端：

http://n-xytrt8gqu585po94mwj5atokcyd4.taobao.com/index.htm

手机端：

https://shop.m.taobao.com/shop/shop_index.htm?sellerId=755668508&shopId=104493595&inShopPageId=423890608&pathInfo=shop/index2

资料下载地址：

链接：https://pan.baidu.com/s/1kCjD8yktZECSGmHomx_veg?pwd=q8er
提取码：q8er

源码下载地址：

https://gitee.com/vi-iot/esp32-board.git

一、前言

esp-idf 是基于 freeRTOS 的框架，里面用到的组件，以及我们的应用程序都是基于 freeRTOS 来开发的，因此我们必须掌握 freeRTOS 的用法。如果我们不深究原理，只关注于 freeRTOS 的接口使用，我们很快就能掌握。另外，因为 freeRTOS 开源免费的特性，目前大部分芯片产商做的 SDK 都是基于 freeRTOS 系统开发的，因此我们就更有理由要学习 RTOS 了。

二、为什么要使用 RTOS?

在低端设备中，程序基本分为裸机和 RTOS，针对简单的程序，我们用裸机程序完全可以满足，一旦功能复杂，程序模块众多，裸机程序往往很难满足我们的需求。因此我们就要用到 RTOS 系统。一个最简单的例子如下：

裸机程序

```
while(1)
{
    funA();
    funB();
    funC();
}
```

RTOS程序

```
while(1)          while(1)          while(1)
{                 {                 {
    funA();           funB();           funC();
    delay(1);         delay(1);         delay(1);
}                 }                 }
```

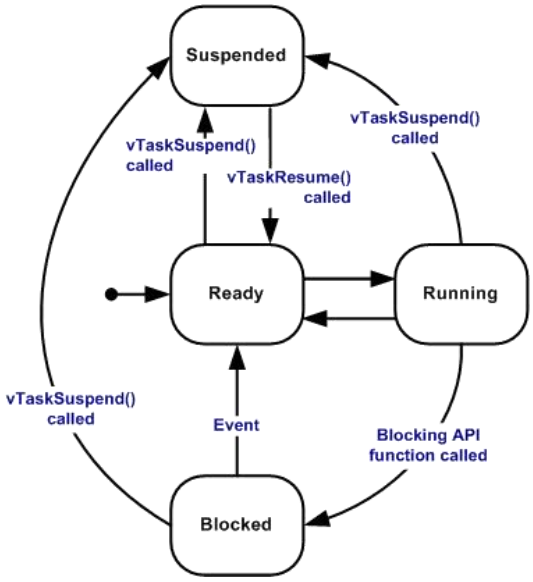
对于裸机程序，我们需要把所有的功能模块代码写在一个大循环里面，可以想象，其中一个功能模块发生阻塞需要延时等待时，其他功能模块均要延时等待，大家可能说可以用状态机来避免这个问题，状态机编程复杂不说，程序可读性也差。

右边是 RTOS 程序，RTOS 程序可以新建多个任务处理不同模块的功能。当模块 A 需要阻塞等待时，它可以放弃自己的执行时间片，把时间片让给其他任务，这样整体程序就可以高效的运行，当然 RTOS 还意味着很多东西，这里只是举一个最直观的例子。

三、freeRTOS 任务概述

使用 FreeRTOS 的实时应用程序可以被构建为一组独立的任务。每个任务在自己的上下文中执行，不依赖于系统内的其他任务或 RTOS 调度器本身。

任务分为四个状态：运行、准备就绪、阻塞、挂起，状态转换如下



运行：

当任务实际执行时，它被称为处于运行状态。 任务当前正在使用处理器。 如果运行 RTOS 的处理器只有一个内核， 那么在任何给定时间内都只能有一个任务处于运行状态。

准备就绪：

准备就绪任务指那些能够执行（它们不处于阻塞或挂起状态），但目前没有执行的任务，因为同等或更高优先级的不同任务已经处于运行状态。

阻塞：

如果任务当前正在等待时间或外部事件，则该任务被认为处于阻塞状态。例如，如果一个任务调用 vTaskDelay()，它将被阻塞（被置于阻塞状态），直到延迟结束-一个时间事件。任务也可以通过阻塞来等待队列、信号量、事件组、通知或信号量事件。处于阻塞状态的任务通常有一个"超时"期， 超时后任务将被超时，并被解除阻塞，即使该任务所等待的事件没有发生。“阻塞”状态下的任务不使用任何处理时间，不能被选择进入运行状态。

挂起：

与“阻塞”状态下的任务一样， “挂起”状态下的任务不能 被选择进入运行状态，但处于挂起状态的任务没有超时。相反，任务只有在分别通过 vTaskSuspend() 和 xTaskResume() API 调用明确命令时 才会进入或退出挂起状态。

优先级：每个任务均被分配了从 0 到 (configMAX_PRIORITIES - 1) 的优先级，其中的 configMAX_PRIORITIES 在 FreeRTOSConfig.h 中定义，低优先级数字表示低优先级任务。 空闲任务的优先级为零。

关于任务，有几个最常见的 API 函数

```
//任务创建
BaseType_t xTaskCreatePinnedToCore(
        TaskFunction_t pvTaskCode,  //任务函数指针，原型是 void fun(void *param)
        const char *constpcName,    //任务的名称，打印调试可能会有用
        const uint32_t usStackDepth,//指定的任务堆栈空间大小（字节）
          void *constpvParameters,  //任务参数
        UBaseType_t uxPriority,     // 优先级，数字越大，优先级越大， 0 到 (configMAX_PRIORITIES - 1)
        TaskHandle_t *constpvCreatedTask,      //传回来的任务句柄
        const BaseType_t xCoreID)   //分配在哪个内核上运行

//延时 xTicksToDelay 个周期
void vTaskDelay( const TickType_t xTicksToDelay )
```

四、队列、信号量、互斥锁

1）队列

队列是任务间通信的主要形式。 它们可以用于在任务之间以及中断和任务之间发送消息。 在大多数情况下，它们作为线程安全的 FIFO（先进先出）缓冲区使用，新数据被发送

到队列的后面， 尽管数据也可以发送到前面。

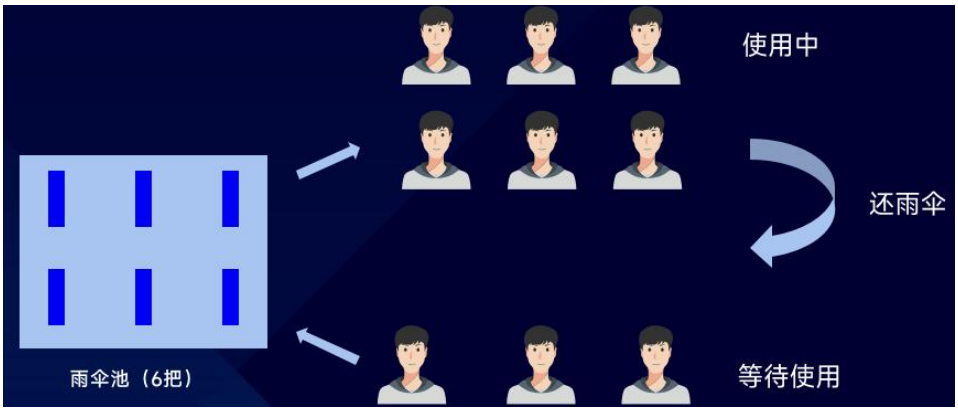
常用如下 API：

```
QueueHandle_t xQueueCreate(     //创建一个队列，成功返回队列句柄
      UBaseType_t uxQueueLength,      //队列容量
      UBaseType_t uxItemSize       //每个队列项所占内存的大小（单位是字节）
);

BaseType_t xQueueSend(              //向队列头部发送一个消息
      QueueHandle_t xQueue,                 //  队列句柄
      const void * pvItemToQueue,    //要发送的消息指针
      TickType_t xTicksToWait );        //等待时间

BaseType_t xQueueSendToBack(              //向队列尾部发送一个消息
      QueueHandle_t xQueue,                 //  队列句柄
      const void * pvItemToQueue,    //要发送的消息指针
      TickType_t xTicksToWait );        //等待时间

BaseType_t xQueueReceive(          //从队列接收一条消息
      QueueHandle_t xQueue,                    //队列句柄
      void * pvBuffer,                                  //指向接收消息缓冲区的指针。
      TickType_t xTicksToWait );              //等待时间

BaseType_t xQueueSendFromISR(              //xQueueSend 的中断版本
      QueueHandle_t xQueue,                 //  队列句柄
      const void * pvItemToQueue,    //要发送的消息指针
      BaseType_t *pxHigherPriorityTaskWoken );        //指出是否有高优先级的任务被
唤醒
```

2）信号量

信号量是用来保护共享资源不会被多个任务并发使用

为了让大家更好的理解信号量，我这边画了个类比图。



这个图中，有一个雨伞池，里面一共有六把雨伞，上面的人是正在使用雨伞的人，当使用人数达到 6 人时雨伞已经没有了，下面是等待使用雨伞的人，因为没有雨伞了，他们只能等待，当有人还雨伞的时候，他们其中一人就可以拿雨伞来用。

这个图里面，雨伞池是我们的信号量句柄，雨伞是信号量，而用户就是任务，任务可以等待是否取得信号量，再去继续执行后续操作，保证资源的正确访问，当使用完信号量后，需要释放信号量。

信号量使用起来比较简单。因为在 freeRTOS 中它本质上就是队列，只不过信号量只关心队列中的数量而不关心队列中的消息内容，在 freeRTOS 中有两种常用的信号量，一是计数信号量，而是二进制信号量。

二进制信号量很简单，就是信号量总数只有 1，也就是这个图中总雨伞数量只有 1。计数信号量则可以自定义总共的信号量

以下是常用的 API 函数

```
//创建二值信号量，成功则返回信号量句柄（二值信号量最大只有 1 个）
SemaphoreHandle_t xSemaphoreCreateBinary( void );

//创建计数信号量，成功则返回信号量句柄
SemaphoreHandle_t xSemaphoreCreateCounting(
    UBaseType_t uxMaxCount,        //最大信号量数
    UBaseType_t uxInitialCount);   //初始信号量数

//获取一个信号量，如果获得信号量，则返回 pdTRUE
xSemaphoreTake( SemaphoreHandle_t xSemaphore,//信号量句柄
                TickType_t xTicksToWait );        //等待时间

//释放一个信号量
xSemaphoreGive( SemaphoreHandle_t xSemaphore ); //信号量句柄

//删除信号量
void vSemaphoreDelete( SemaphoreHandle_t xSemaphore );
```

3）互斥锁

互斥锁：与二进制信号量类似，但会发生优先级翻转

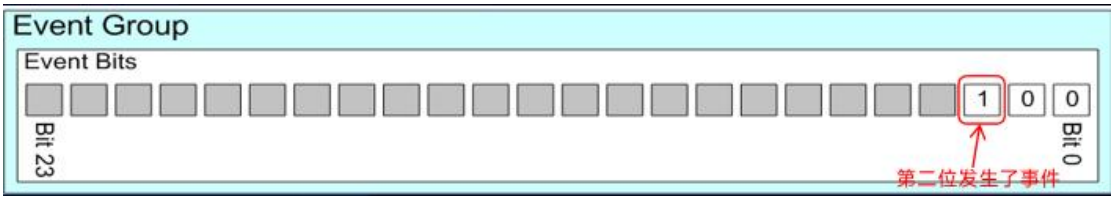
```
//创建一个互斥锁
SemaphoreHandle_t xSemaphoreCreateMutex( void )
```

五、事件组

事件位：用于指示事件是否发生，事件位通常称为事件标志

事件组：就是一组事件位。 事件组中的事件位通过位编号来引用

下图表示一个 24 位事件组， 使用 3 个位来保存前面描述的 3 个示例事件



以下是常用的 API 函数

```
//创建一个事件组，返回事件组句柄，失败返回 NULL
EventGroupHandle_t xEventGroupCreate( void );


//等待事件组中某个标志位,用返回值以确定哪些位已完成设置
EventBits_t xEventGroupWaitBits(
                        const EventGroupHandle_t xEventGroup,   //事件组句柄
                        const EventBits_t uxBitsToWaitFor, //哪些位需要等待
                        const BaseType_t xClearOnExit,          //是否自动清除标志
位
                        const BaseType_t xWaitForAllBits,      //是否等待的标志位都
成功了才返回
                        TickType_t xTicksToWait );                  //最大阻塞时间

//设置标志位
EventBits_t xEventGroupSetBits(
                    EventGroupHandle_t xEventGroup,//事件组句柄
                    const EventBits_t uxBitsToSet ); //设置哪个位

//清除标志位
EventBits_t xEventGroupClearBits(
                                EventGroupHandle_t xEventGroup,   //事件组句柄
                                const EventBits_t uxBitsToClear );      //清除的标志位
```

六、直达任务通知、

定义：每个 RTOS 任务都有一个任务通知数组。 每条任务通知 都有“挂起”或“非挂起”的通知状态， 以及一个 32 位通知值。

直达任务通知是直接发送至任务的事件， 而不是通过中间对象 （如队列、事件组或信号量）间接发送至任务的事件。向任务发送“直达任务通知” 会将目标任务通知设为“挂起”状态（此挂起不是挂起任务）。

以下是最常用的 API

```
//用于将事件直接发送到 RTOS 任务并可能取消该任务的阻塞状态
    BaseType_t xTaskNotify(
            TaskHandle_t xTaskToNotify, //要通知的任务句柄
            uint32_t ulValue,          //携带的通知值
            eNotifyAction eAction ); //执行的操作
```

需要注意的是参数 eAction 如下表所述

eAction 设置	**执行的操作**
eNoAction	目标任务接收事件，但其 通知值未更新。 在这种情况下，不使用 ulValue。
eSetBits	目标任务的通知值 使用 ulValue 按位或运算
eIncrement	目标任务的通知值自增 1（类似信号量的 give 操作）
eSetValueWithOverwrite	目标任务的通知值 无条件设置为 ulValue。
eSetValueWithoutOrwrite	如果目标任务没有 挂起的通知，则其通知值 将设置为 ulValue。如果目标任务已经有 挂起的通知，则不会更新其通知值。

```
    //等待接收任务通知
    BaseType_t xTaskNotifyWait(
            uint32_t ulBitsToClearOnEntry,    //进入函数清除的通知值位
            uint32_t ulBitsToClearOnExit, //退出函数清除的通知值位
            uint32_t *pulNotificationValue,    //通知值
            TickType_t xTicksToWait );    //等待时长
```

关于 freeRTOS 经典例程，源码在于 esp32-board/rtos 中，这个例程 main.c 中包含了 freeRTOS 在实际应用中非常常用的例程，也有丰富的注释，大家看完之后肯定能学会如何使用 freeRTOS。