FreeRTOS

翁恺

FreeRTOS

- FreeRTOS由美国的RichardBarry于2003年发布，RichardBarry是FreeRTOS的拥有者和维护者，在过去的十多年中FreeRTOS历经了9个版本，与众多半导体厂商合作密切，有数百万开发者，是目前市场占有率最高的RTOS

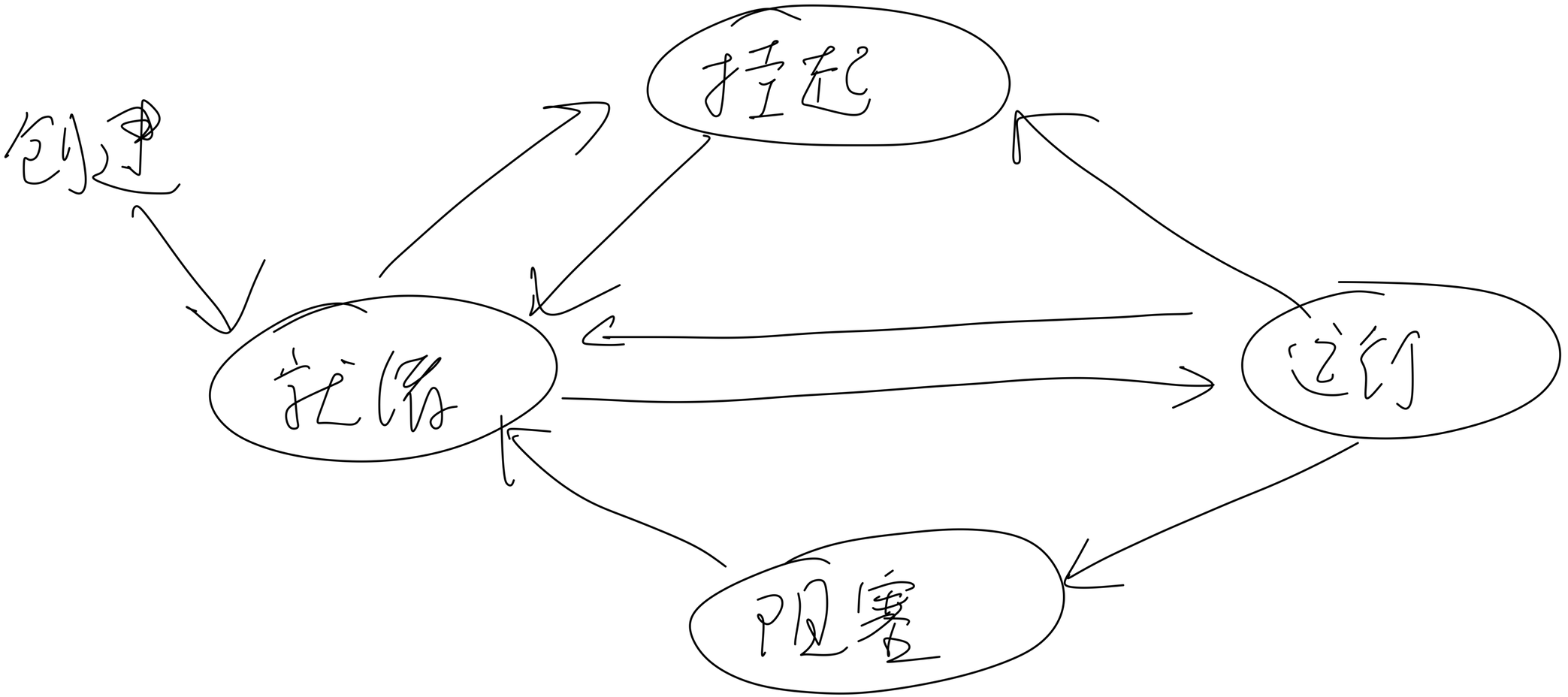
- FreeRTOS于2018年被亚马逊收购，改名为AWSFreeRTOS，版本号升级为V10，且开源协议也由原来的GPLv2+修改为MIT

《FreeRTOS内核实现与应用开发指南》，刘火良、杨森，2019

OpenRTOS?

比较的项目	FreeRTOS	OpenRTOS
是否免费	是	否
可否商业使用	是	是
是否需要版权费	否	否
是否提供技术支持	否	是
是否被法律保护	否	是
是否需要开源工程代码	否	否
是否需要开源修改的内核源码	是	否
是否需要声明产品使用了 FreeRTOS	如果发布源码，则需要声明	否
是否需要提供 FreeRTOS 的整个工程代码	如果发布源码，则需要提供	否

任务状态



任务优先级

- 每个任务都要被指定一个优先级，从0~configMAX_PRIORITIES，configMAX_PRIORITIES定义在FreeRTOSConfig.h中
- 如果某架构硬件支持CLZ，将FreeRTOSConfig.h中configUSE_PORT_OPTIMISED_TASK_SELECTION设置为1，并且最大优先级数目configMAX_PRIORITIES不能大于32
- 低优先级数值代表低优先级。空闲任务（idle task）的优先级为0（tskIDLE_PRIORITY）
- 任何数量的任务可以共享同一个优先级。如果宏configUSE_TIME_SLICING未定义或着宏configUSE_TIME_SLICING定义为1，处于就绪态的多个相同优先级任务将会以时间片切换的方式共享处理器

任务函数

```
void vATaskFunction( void *pvParameters )
{
    for( ;; )
    {
        /*-- 应用程序代码放在这里. --*/
    }

    /* 任务不可以从这个函数返回或退出。在较新的FreeRTOS移植包中，如果
    试图从一个任务中返回，将会调用configASSERT()（如果定义的话）。
    如果一个任务确实要退出函数，那么这个任务应调用vTaskDelete(NULL)
    函数，以便处理一些清理工作。*/
    vTaskDelete( NULL );
}
```

空闲任务和空闲任务钩子

- 空闲任务是启动RTOS调度器时由内核自动创建的任务，这样可以确保至少有一个任务在运行。空闲任务具有最低任务优先级
- 删除任务后，空闲任务用来释放RTOS分配给被删除任务的内存
- 应用程序任务共享空闲任务优先级（tskIDLE_PRIORITY）也是可能的
- 空闲任务钩子是一个函数，每一个空闲任务周期被调用一次
  - void vApplicationIdleHook(void);

创建任务

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

- usStackDepth以字长为单位
- pvCreatedTask带回任务的id，之后要用这个id来表示任务

延时

- void vTaskDelay(portTickType xTicksToDelay)
  - 相对延时，时长单位是系统时钟节拍周期
- void vTaskDelayUntil(TickType_t *... const TickType_t xTimeIncremen...

为什么不能在任务中跑空循环来延时?

定时器

- 硬件上通常具有多个定时器
- 程序逻辑上需要定时器用于：
  - 等待一定时间后再做某事
  - 某个等待（如串口的回答）不能超时
  - 一定时间后需要做某事（不等待）

- pxPreviousWakeTime指向一个持久存储的变量，保存上次的时间
- 当时间等于(*pxPreviousWakeTime + xTimeIncrement)时，任务解除阻塞

绝对延时的例子

```
//每10次系统节拍执行一次
void vTaskFunction(void *pvParameters)
{
    static portTickType xLastWakeTime;
    const portTickType xFrequency = 10;

    // 使用当前时间初始化变量xLastWakeTime
    xLastWakeTime = xTaskGetTickCount();

    for (;;)
    {
        //等待下一个周期
        vTaskDelayUntil(&xLastWakeTime, xFrequency);

        // 需要周期性执行代码放在这里
    }
}
```

优先级

- void vTaskPrioritySet(TaskHandle_t xTask, UBaseType_t uxNewPriority);
  - 如果设置的优先级高于当前运行的任务，在函数返回前会进行一次上下文切换。
  - 在FreeRTOSConfig.h中，宏INCLUDE_vTaskPrioritySet 必须设置成1
- UBaseType_t uxTaskPriorityGet(TaskHandle_t xTask);
  - 宏INCLUDE_vTaskPriorityGet必须设置成1

任务的挂起和恢复

- void vTaskSuspend(TaskHandle_t xTaskToSuspend);
- void vTaskResume(TaskHandle_t xTaskToResume);
- BaseType_t xTaskResumeFromISR(TaskHandle_t xTaskToResume);
  - 如果中断恰巧在任务被挂起之前到达，这就会导致这次的恢复无效

任务状态

```
UBaseType_t uxTaskGetSystemState(TaskStatus_t * const pxTaskStatusArray, const
UBaseType_t uxArraySize, unsigned long * const pulTotalRunTime );

typedef struct xTASK_STATUS
{
    /* 任务句柄*/
    TaskHandle_t xHandle;

    /* 指针，指向任务名*/
    const signed char *pcTaskName;

    /*任务ID，是一个独一无二的数字*/
    UBaseType_t xTaskNumber;

    /*填充结构体时，任务当前的状态（运行、就绪、挂起等等）*/
    eTaskState eCurrentState;

    /*填充结构体时，任务运行（或继承）的优先级。*/
    UBaseType_t uxCurrentPriority;

    /* 当任务因继承而改变优先级时，该变量保存任务最初的优先级。仅当configUSE_MUTEXES定义为1有效。*/
    UBaseType_t uxBasePriority;

    /* 分配给任务的总运行时间。仅当宏configGENERATE_RUN_TIME_STATS为1时有效。*/
    unsigned long ulRunTimeCounter;

    /* 从任务创建起，堆栈剩余的最小数量，这个值越接近0，堆栈溢出的可能越大。 */
    unsigned short usStackHighWaterMark;
}
```

调用此函数会挂起所有任务，直到函数最后才恢复挂起的任务

任务数据

- TaskHandle_t xTaskGetCurrentTaskHandle(void);
- TaskHandle_t xTaskGetIdleTaskHandle(void);
- UBaseType_t uxTaskGetStackHighWaterMark(TaskHandle_t xTask);
  - 返回任务启动后的最小剩余堆栈空间，单位为字
- eTaskState eTaskGetState(TaskHandle_t xTask);
  - eReady, eRunning, eBlocked, eSuspended, eDeleted

调度器数据

- char * pcTaskGetTaskName(TaskHandle_t xTaskToQuery);
- volatile TickType_t xTaskGetTickCount(void);
- volatileTickType_t xTaskGetTickCountFromISR(void)
- BaseType_t xTaskGetSchedulerState(void);
  - taskSCHEDULER_NOT_STARTED（未启动）
  - taskSCHEDULER_RUNNING（正常运行）
  - taskSCHEDULER_SUSPENDED（挂起）

“ps”

- void vTaskList(char *pcWriteBuffer);

```
Name          State   Priority  Stack  Num
*********************************************
Print           R       4       331     29
Math7           R       0       417     7
Math8           R       0       407     8
QConsB2         R       0       53      14
QProdB5         R       0       52      17
QConsB4         R       0       53      16
SEM1            R       0       50      27
SEM1            R       0       50      28
IDLE            R       0       64      0
```


调用这个函数会挂起所有任务，这一过程可能持续较长时间

内核控制宏

- taskYIELD：用于强制上下文切换的宏。在中断服务程序中的等价版本为portYIELD_FROM_ISR
- taskENTER_CRITICAL：用于进入临界区的宏，临界区嵌套计数器增1
- taskEXIT_CRITICAL：用于退出临界区的宏，先将临界区嵌套计数器减1，如果临界区计数器为零，则使能所有RTOS可屏蔽中断
- taskDISABLE_INTERRUPTS：禁止所有RTOS可屏蔽中断
- taskENABLE_INTERRUPTS：使能所有RTOS可屏蔽中断

调度器管理

- void vTaskStartScheduler(void);
- void vTaskSuspendAll(void);
  - 挂起调度器，但不禁止中断。当调度器挂起时，不会进行上下文切换
  - 内核调度器挂起期间，那些可以引起上下文切换的API函数（如vTaskDelayUntil()、xQueueSend()等）决不可使用
- BaseType_t xTaskResumeAll(void);

信号量

- FreeRTOS的信号量包括二进制信号量、计数信号量、互斥信号量和递归互斥信号量
- 互斥量具有优先级继承，信号量没有
- 互斥量不能用在中断服务程序中，信号量可以
- 释放一个空的二值信号量也不会导致任务被挂起，和释放互斥量一样
  - 与Linux的信号量不同
- 创建互斥量和创建信号量的API函数不同，但是共用获取和给出信号API函数

二值信号量

- 信号量API函数允许指定一个阻塞时间。当任务企图获取一个无效信号量时，任务进入阻塞状态，阻塞时间用来确定任务进入阻塞的最大时间，阻塞时间单位为系统节拍周期时间。如果有多个任务阻塞在同一个信号量上，那么当信号量有效时，具有最高优先级别的任务最先解除阻塞

- 如果需要任务来处理外设，使用轮询的方法会浪费CPU资源并且妨碍其它任务执行。更好的做法是任务的大部分时间处于阻塞状态（允许其它任务执行），直到某些事件发生该任务才执行。可以使用二值信号量实现这种应用：当任务取信号量时，因为此时尚未发生特定事件，信号量为空，任务会进入阻塞状态；当外设需要维护时，触发一个中断服务程序，该ISR仅仅给出信号量。中断退出后，任务获得信号量，如果优先级合适，则得到运行

- 中断程序中决不可使用无“FromISR”结尾的API函数

互斥量

- 如果一个互斥量正在被一个低优先级任务使用，此时一个高优先级企图获取这个互斥量，高优先级任务会因为得不到互斥量而进入阻塞状态，正在使用互斥量的低优先级任务会临时将自己的优先级提升，提升后的优先级与与进入阻塞状态的高优先级任务相同。这个优先级提升的过程叫做优先级继承

- 已经获取递归互斥量的任务可以重复获取该递归互斥量。使用 xSemaphoreTakeRecursive() 函数成功获取几次递归互斥量，就要使用 xSemaphoreGiveRecursive()函数返还几次，在此之前递归互斥量都处于无效状态

信号量的创建和删除

- SemaphoreHandle_t xSemaphoreCreateBinary(void);
- SemaphoreHandle_t xSemaphoreCreateCounting (UBaseType_t uxMaxCount, UBaseType_t uxInitialCount)
- SemaphoreHandle_t xSemaphoreCreateMutex(void)
- SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(void)
- void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);

信号量操作

- P(S)—>wait(s):
  - while(s<=0) ; s—;
- V(S)—>singal(s):
  - s++;
- 这两个都必须是原子操作

PV操作

- P
  - xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait)
  - xSemaphoreTakeFromISR(SemaphoreHandle_t xSemaphore, signedBaseType_t *pxHigherPriorityTaskWoken)
  - xSemaphoreTakeRecursive(SemaphoreHandle_t xMutex, TickType_t xTicksToWait);
- V
  - xSemaphoreGive(SemaphoreHandle_t xSemaphore)
  - xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore, signed BaseType_t *pxHigherPriorityTaskWoken)
  - xSemaphoreGiveRecursive(SemaphoreHandle_t xMutex)

信号量数据

- TaskHandle_t xSemaphoreGetMutexHolder(SemaphoreHandle_t xMutex);
- 返回互斥量持有任务的句柄（如果有的话），互斥量由参数xMutex指定
- 如果调用此函数的任务持有互斥量，那么可以可靠的返回任务句柄，但是如果是别的任务持有互斥量，则不总可靠

任务通知

- 每个FreeRTOS任务都有一个32位的通知值，任务创建时，这个值被初始化为0。FreeRTOS任务通知相当于直接向任务发送一个事件，接收到通知的任务可以解除阻塞状态，前提是这个阻塞事件是因等待通知而引起的。发送通知的同时，也可以可选的改变接收任务的通知值
  - 不覆盖接收任务的通知值
  - 覆盖接收任务的通知值
  - 设置接收任务通知值的某些位
  - 增加接收任务的通知值
- 使用任务通知可以快45%、使用更少的RAM

发通知

- BaseType_t xTaskNotify(TaskHandle_txTaskToNotify, uint32_t ulValue, eNotifyAction eAction)
  - eNoAction：不使用ulValue
  - eSetBits：通知值按位或ulValue，类似uc/OS的事件组
  - eIncrement：通知值+1，ulValue不变
  - eSetValueWithOverwrite：写值
  - eSetValueWithoutOverwrite：如果之前的通知值还没有读取则丢弃这次的

TaskNotifyGive

- BaseType_t xTaskNotifyGive(TaskHandle_t xTaskToNotify);
- 本质上相当于xTaskNotify((xTaskToNotify), (0), eIncrement)。可以使用该函数代替二值或计数信号量，但速度更快。在这种情况下，应该使用API函数ulTaskNotifyTake()来等待通知，而不应该使用API函数xTaskNotifyWait()

获取通知

- uint32_t ulTaskNotifyTake(BaseType_txClearCountOnExit, TickType_txTicksToWait);
- 如果在中断中使用，要使用它们的中断保护等价函数vTaskNotifyGiveFromISR()和xTaskNotifyFromISR()

队列

- 队列是主要的任务间通讯方式。可以在任务与任务间、中断和任务间传送信息。大多数情况下，队列用于具有线程保护的FIFO（先进先出）缓冲区
- 任务A使用API函数xQueueSendToBack()向队列发送数据，每次发送一个数据，新入队的数据置于上一次入队数据的后面。任务B使用API函数xQueueReceive()将数据从队列取出，先入队的数据先出队
- 队列内存区域分配由内核完成

队列API

- QueueHandle_t xQueueCreate (UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
- BaseType_t xQueueSend(QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait);
- BaseType_t xQueueReceive(QueueHandle_t xQueue, void *pvBuffer,TickType_t xTicksToWait);
- BaseType_t xQueuePeek(QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait);
- void vQueueUnregisterQueue(QueueHandle_t xQueue);