CooCox CoOS 用户手册

修正版 1.1

2009 年 8 月

www.coocox.org

目录

1 概述 1
1.1 关于 CooCox CoOS 1
1.2 CooCox CoOS 入门指南 3
2 任务管理 9
2.1 任务 9
2.2 任务状态 10
2.3 任务控制块 11
2.4 任务就绪链表 14
2.5 任务调度 15
2.6 临界区 17
2.7 中断 18
3 时间管理 19
3.1 系统节拍 19
3.2 延时管理 21
3.3 软件定时器 22
4 内存管理 24
4.1 静态内存分配 24
4.2 动态内存分配 25
4.3 堆栈溢出检查 29
5 任务间的同步与通信 30
5.1 任务间的同步 30
5.2 任务间的通信 35
6 API 手册 38
6.1 系统管理 38
6.2 任务管理 44
6.3 时间管理 55
6.4 软件定时器 59
6.5 内存管理 66
6.6 互斥区域 73
6.7 信号量 76
6.8 邮箱 83
6.9 消息队列 92
6.10 事件标志 101
6.11 系统工具 113
6.12 其它 116

1 概述

1.1 关于 CooCox CoOS

CooCox CoOS 是一款针对 ARM Cortex-M 系列芯片而设计的实时系统内核。

1.1.1 CooCox CoOS 特征

- Cortex M 系列微控制器定制
- 免费及开源的实时系统内核
- 高度可裁剪性，最小系统内核仅 974Byte
- 自适应任务调度算法
- 支持优先级抢占和时间片轮转
- 零中断延时时间
- 信号量、邮箱、队列、事件标志、互斥等同步通信方式
- 堆栈溢出检测
- 支持多种编译器：ICCARM, ARMCC, GCC

1.1.2 CooCox CoOS 的技术特性

表 1.1.1 时间特性

功能	时间 (无时间片轮转 / 有时间片轮转)
创建已定义的任务（无任务切换）	5.3us / 5.8us
创建已定义的任务（有任务切换）	7.5us / 8.6us
删除任务 (退出任务)	4.8us / 5.2us
任务切换 (切换内容)	1.5us / 1.5 us
任务切换 (在设置事件标志的情况下)	7.5us / 8.1us
任务切换 (在发送信号量的情况下)	6.3us / 7.0us
任务切换 (在发送邮件的情况下)	6.1us / 7.1us
任务切换 (在发送队列的情况下)	7.0us / 7.6us
设置事件标志 （无任务切换）	1.3us / 1.3us
发送信号量 （无任务切换）	1.6us / 1.6us
发送邮件（无任务切换）	1.5us / 1.5us
发送队列（无任务切换）	1.8us / 1.8us
IRQ 中断服务程序的最大中断延迟时间	0 / 0

表 1.1.2 空间特性

描述	空间
内核占 RAM 空间	168 Bytes
内核占代码空间	<1KB
一个任务占 RAM 空间	TaskStackSize + 24 Bytes(MIN) TaskStackSize + 48 Bytes(MAX)
一个邮箱占 RAM 空间	16 Bytes
一个信号量占 RAM 空间	16 Bytes
一个队列占 RAM 空间	32 Bytes
一个互斥体占 RAM 空间	8 Bytes
一个用户定时器占 RAM 空间	24 Bytes

1.1.3 支持的器件（所有 Cortex M0 & Cortex M3 系列，以下仅列出常用的）

- ST STM32 系列
- Atmel ATSAM3U 系列
- NXP LPC17xx LPC13xx LPC11xx 系列
- Toshiba TMPM330 系列
- Luminary LM3S 系列
- Nuvoton NUC1xx 系列
- Energy Micro EFM32 系列

1.1.4 源码下载

如果你想要了解更多关于 CooCox CoOS，你可以从如下网站下载 CooCox CoOS 的源代码：

www.coocox.org

1.2 CooCox CoOS 入门指南

本节介绍 CooCox CoOS 的使用，这里我们使用 Keil 公司的开发利器 Keil RealView MDK 开发工具和 NXP 公司的 EM-LPC1700 开发板做一个很简单的基于 CoOS 的 demo。

有关 Keil 公司，您可以访问相应的网站了解更多详情：

Keil： www.keil.com

我们这里假定您已经会使用 Keil RealView MDK 进行简单的开发和基本的设置。下面为您介绍的是这样一个简单的实例，实例中包括三个任务：

led ：用于 8 个 LED 的循环闪烁，并按固定的间隔时间设置标志来激活另外两个任务；

taskA：循环等待任务标志 a_flag 到来，然后通过串口 1 打印 taskA；

taskB：循环等待任务标志 b_flag 到来，然后通过串口 1 打印 taskB；

总体现象是，评估板上的 8 个 LED 每 0.5 秒变动一次，变动顺序为 LED0 à LED1 à LED2 à … à LED6 à LED7 à LED0 à LED1 à …，串口每 0.5 秒打印一次信息，依次打印：

taskA is running

taskB is running

…

LED 与 GPIO 的对应关系为：

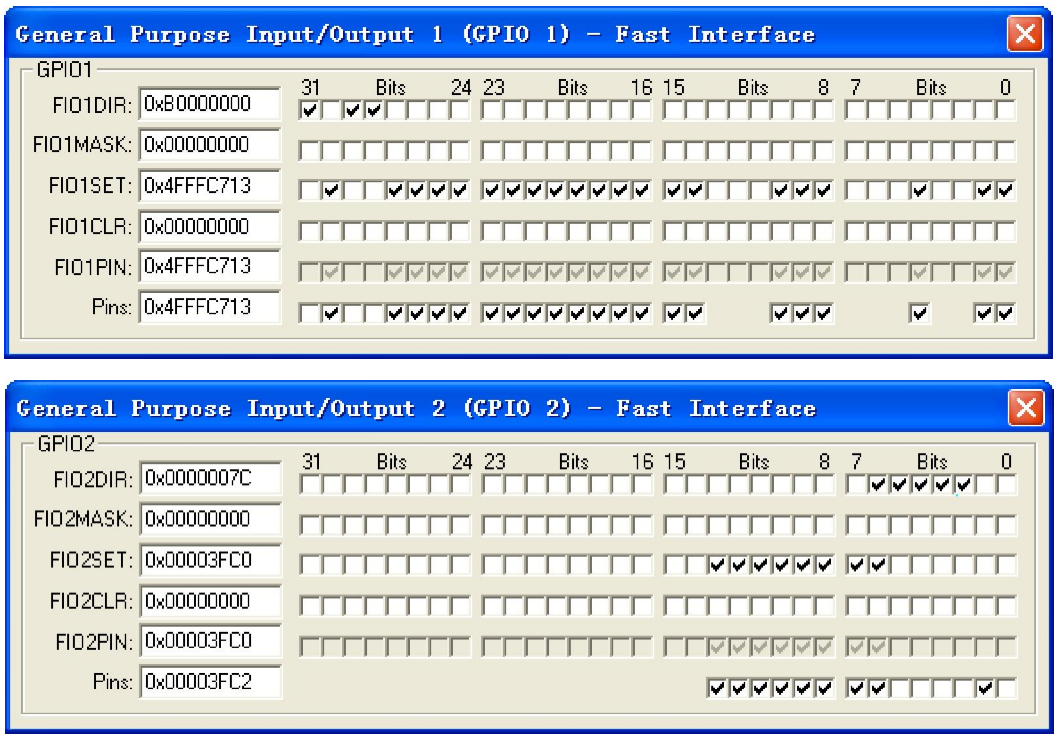
LED0 ß à P1.28

LED1 ß à P1.29

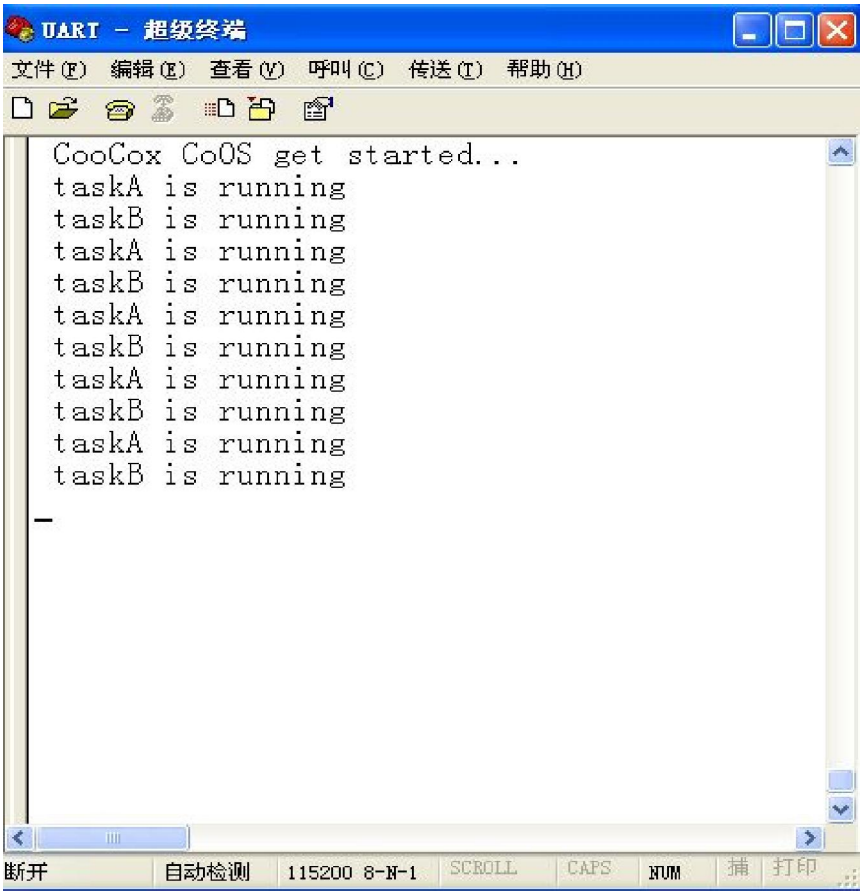
LED2 ß à P1.31

LED[3…7] ß à P[2.2…2.6]

若您在 MDK 模拟器上使用，将会得到如下运行结果：



P1.2.1 GPIO 引脚电平变化



P1.2.2 打印结果输出

接下来介绍 CoOS 是怎样实现以上功能的：

1.2.1 准备工作

1 > 访问 www.coocox.com 网站下载 CoOS 源码包。
2 > 首先创建一个文件夹，命名为 getting_started（注：文件夹名中不要有空格或中文）；
3 > 然后进入 getting_started，分别创建 inc、src 和 ccrtos 文件夹，用于存放头文件和源文件；
4 > 复制文件到工程目录下：

(1) 把 CooCox CoOS 的发布包中的 Demo\Sample\CMSIS 目录下的 LPC17xx.h，system_LPC17xx.h 复制到 inc 文件夹中，把 system_LPC17xx.c，startup_LPC17xx.s(LPC17xx 的启动代码)复制到 src 文件夹中；

(2) 把 Source 文件夹中的 config.h 文件复制到 inc 文件夹中，这个是 CoOS 的裁剪配置文件，把 Demo\Sample\main.c 复制到 src 文件夹中；

(3) 把 Demo\Sample\driver 文件夹中的 LED.c，Retarget.c，Serial.c 复制到 src 文件夹中，把 serial.h，led.h 复制到 inc 文件夹中；

(4) 把 Source\kernel\与 Source\portable\Keil\目录下所有的源文件(除.h 文件外)复制到 ccrtos 文件夹中，所有.h 文件复制到 inc 文件夹中。

1.2.2 创建工程

1 > 用 MDK 软件创建一个空工程，器件选择 NXP 的 LPC1766，内部没有任何文件；
2 > 添加应用驱动代码到工程：
添加 src 文件夹下的所有源文件到工程中；
3 > 添加 CoOS 源码到工程：
添加 ccrtos 文件夹下的所有源文件到工程中(头文件不用包含)；
4 > 配置工程
对工程做适当的配置，在 C/C++中添加包含路径：.\inc。此后您应该已经能成功通过编译，如果不能，请仔细检查步骤和 MDK 设置是否正确。

(如果您想先使用，而不想做前面这些工作来节省您的时间，您可以直接使用我们为你准备的一个现成的工程，存放路径为\Demo\getting_start_sample.rar，但是建议您在使用之前保存一个副本，以备后患)

1.2.3 编写 Application code

打开“main.c”，您可以发现，我们已经为您做了一部分工作，包括初始化时钟，初始化串口 1，初始化用于 LED 闪烁的 GPIO。我们需要做的就是一步步往里面添加任务代码和配置 CoOS。

1 > 包含 CoOS 头文件
要使用 CoOS，首先要将源代码加入您的工程，这个步骤在前面已经完成，接下来要做的就是在您的用户代码中包含 CoOS 的头文件，即在 main.c 中添加如下语句：

```
#include <config.h>              /*!< CoOS configure header file*/
#include <ccrtos.h>              /*!< CoOS header file          */
```

注：最好将#include <config.h>放在#include <ccrtos.h>的前面。

2 > 编写任务代码
任务在创建的时候要为它指定堆栈空间，对于 CoOS，任务的堆栈指针是用户指定的，所以我们要定义三个数组用于三个任务的堆栈：

```
OS_STK      taskA_stk[128];      /*!< define "taskA" task stack  */
OS_STK      taskB_stk[128];      /*!< define "taskB" task stack  */
OS_STK      led_stk  [128];      /*!< define "led" task stack    */
```

其中 taskA，taskB 任务分别等待各自的标志，很显然，我们必须创建两个任务用于任务通讯。另外，taskA，taskB 需要串口打印，串口是一个只能由一个任务占有的互斥型设备，所以我们创建一个互斥体用于保证串口打印时的互斥性。

```
OS_MutexID uart_mutex;          /*!< UART1 mutex id        */
OS_FlagID  a_flag,b_flag;       /*!< Save falg id          */
volatile   unsigned int Cnt = 0;  /*!< A counter           */
```

taskA 任务：taskA 等待一个标志，当标志置位后，打印 taskA is running，然后继续等待标志，如此循环。这里我们设计成 taskA 作为最高优先级任务，在启动系统后第一个得到执行，所以我们在 taskA 中创建后面要用的标志和互斥。任务代码如下：

```
void taskA (void* pdata)
{
  /*!< Create a mutex for uart print */
  uart_mutex = CoCreateMutex ();
  if (uart_mutex == E_CREATE_FAIL)
  {  /*!< If failed to create, print message     */
      printf (" Failed to create Mutex! \n\r");
  }
  /*!< Create two flags to communicate between taskA and taskB     */
  a_flag = CoCreateFlag (TRUE,0);
  if (a_flag == E_CREATE_FAIL)
  {
      printf (" Failed to create the Flag! \n\r");
  }
  b_flag = CoCreateFlag (TRUE,0);
  if (b_flag == E_CREATE_FAIL)
  {
      printf (" Failed to create the Flag ! \n\r");
  }
  for (;;)
 {
      CoWaitForSingleFlag (a_flag,0);
      CoEnterMutexSection (uart_mutex);
      printf (" taskA is running \n\r");
      CoLeaveMutexSection (uart_mutex);
  }
}
```

taskB 任务：taskB 等待 b_flag 标志置位，置位激活后打印 taskB is running，之后继续等待标志，如此循环，任务代码如下：

```
void taskB (void* pdata)
{
  for (;;)
  {
    CoWaitForSingleFlag (b_flag,0);

     CoEnterMutexSection (uart_mutex);
     printf (" taskB is running \n\r");
     CoLeaveMutexSection (uart_mutex);
  }
}
```

led 任务：led 任务控制 led 的变化，这里通过 CoOS 的延时函数 CoTickDelay()来延时任务 0.5 秒，促使 LED 保持点亮状态 0.5 秒后关闭；led 任务同时在合适的时间设置 a_flag，

b_flag 标志，以分别激活 taskA，taskB。程序代码如下：

```
void led (void* pdata)
{
  unsigned int led_num;
  for (;;)
  {
    led_num = 1<< (Cnt%8);
    LED_on (led_num);            /*!< Switch on led    */
    CoTickDelay (50);
    LED_off (led_num);           /*!< Switch off led    */
    if ((Cnt%2) == 0)
    {
       CoSetFlag (a_flag);       /*!< Set "a_flag" flag*/
    }
    else if ((Cnt%2) == 1)
    {
       CoSetFlag (b_flag);       /*!< Set "b_flag" flag*/
    }
    Cnt++;
  }
}
```

1.2.4 创建任务，起始多任务

此刻我们已经完成了所有的任务代码，接下来应该初始化 OS，创建任务，起始多任务调度。在使用 CoOS 之前，也就是调用任何 OS 的 API 之前，必须首先对 CoOS 进行初始化，这个工作通过 CoInitOS()函数来完成。初始化完成之后，就可以调用系统的 API 来创建任务，创建标志、互斥、信号量....，最后，通过 CoStartOS()函数，系统进行第一次调度，系统正式启动。CoStartOS()之后的代码不会得到执行，因为 OS 在第一次调度之后不会返回。

在 main 函数目标初始化的后面，添加以下代码：

```
CoInitOs ();                     /*!< Initial CooCox CoOS          */
  /*!< Create three tasks   */
  CoCreateTask (taskA,0,0,&taskA_stk[128-1],128);
  CoCreateTask (taskB,0,1,&taskB_stk[128-1],128);
  CoCreateTask (led   ,0,2,&led_stk[128-1]   ,128);
  CoStartOS ();                  /*!< Start multitask              */
```

1.2.5 配置、裁剪 CoOS

打开 *OsConfig.h*，这里包括了 CoOS 的所有可配置项和裁剪项目，在修改每一项之前，确保您已经了解了它的真实作用，文件里有详细的注释来解释每一项的用途。

首先，我们来配置几个必须要核查或者修改的项目：

CFG_MAX_USER_TASKS

这个指示用户最多能创建多少个任务，这里我们只有三个任务，所以我们修改它的值为 3 以节省空间。

CFG_CPU_FREQ

这个是你的系统所用的系统时钟，前面的 SystemInit()函数把芯片初始化为 72MHz，所以这里修改成 72000000，对应实际的目标芯片的工作频率。

CFG_SYSTICK_FREQ

这个是系统节拍周期，我们设置为 100，表示 10ms、100Hz 的节拍时钟。

完成了以上工作，你的程序就能够正确运行了。编译您的工程，通过我们的配套的仿真器 Colink 下载到目标板，或者在 MDK 的模拟器上运行您的程序，就可以观察到前述的现象。

2 任务管理

2.1 任务

在基于 OS 的应用开发中，一个应用程序通常由若干个任务组成。在 CooCox CoOS 中，任务通常是一个内部无限循环的 C 函数，同样有返回值和参数，由于任务永远不会返回，所以任务的返回类型必须定义为 void。程序 1 为一个典型的任务体。

程序 1　一个无限循环的任务体

```
void  myTask (void* pdata)
{
    for(;;)
    {
    }
}
```

与普通的 C 函数不同，任务退出是通过调用系统退出的 API 函数来实现的。若只是通过代码执行结束来表示任务退出，这样将会会导致系统崩溃。

在 CooCox CoOS 中，可以调用 ExitTask()和 DelTask(taskID)来删除一个任务。ExitTask()删除当前正在运行的任务；DelTask(taskID)可以删除其他任务，若参数为当前任务 ID，则同 ExitTask()一样删除当前任务。具体用法如程序 2 所示：

程序 2　删除任务

```
void  myTask0 (void* pdata)
{
    CoExitTask();
}
void  myTask1 (void* pdata)
{
    CoDelTask(taskID);
}
```

2.2 任务状态

在 CooCox CoOS 中，一个任务可以存在于以下几种状态之一：

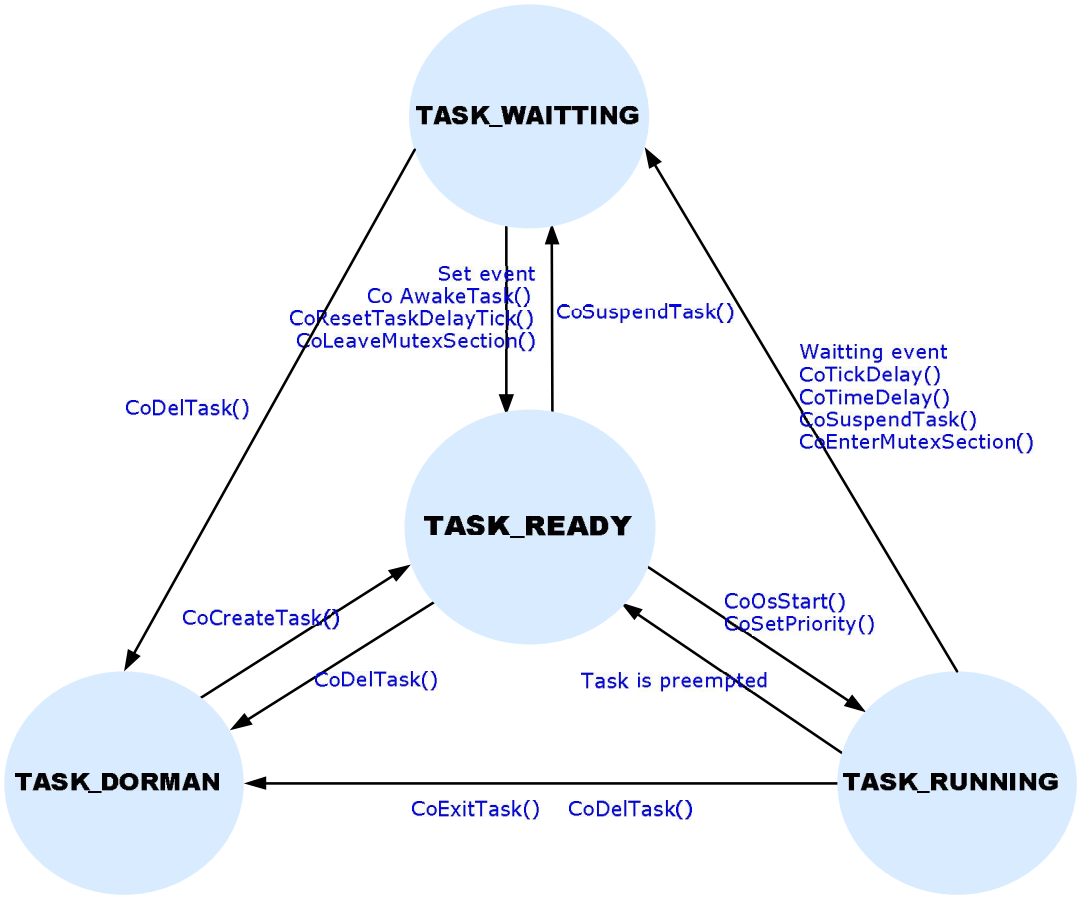
就绪态（TASK_READY）：就绪态任务是指那些可以执行（它们不是处于等待态或休眠态）但因当前有一个同等优先级或更高优先级的任务在运行状态而不能运行的任务。任务一旦创建完成就处于此状态。

运行态（TASK_RUNNING）：当一个任务真正在运行的时候它就处于运行态。它占用当前 CPU。

等待态（TASK_WAITING）：等待某个事件发生。在 CooCox CoOS 中，等待任一事件都将使任务变为等待状态。

休眠态（TASK_DORMANT）：任务已被删除，不再参与任务调度。休眠态不同于等待态：处于等待态中的任务当等待的事件满足时会被重新激活，参与任务调度；而处于休眠态的任务则永远不会被再次激活。

各任务间状态可以进行转换，如下图所示：在 CooCox CoOS 中，调用 CoSuspendTask()可使任务从运行态、就绪态变为等待态，调用 CoAwakeTask()则可使任务从等待态恢复至就绪态。



P2.2.1　任务状态间的切换

2.3 任务控制块

任务控制块是 CooCox CoOS 用来保存任务状态的一种数据结构。CooCox CoOS 每创建一个任务，就会分配一个任务控制块来描述该任务（如程序 3），以确保该任务的 CPU 使用权被剥夺后又重新获得时能丝毫不差地继续执行。

任务控制块作为任务的描述快照一直伴随任务存在，直到任务被删除时才被系统收回。

程序 3　任务控制块

```
typedef struct TCB
{
   OS_STK *stkPtr;                          /*!< The current point of task. */
   U8 prio;                                 /*!< Task priority. */
   U8 state;                                /*!< TaSk status. */
   OS_TID taskID;                           /*!< Task ID. */
#if CFG_MUTEX_EN > 0
   OS_MutexID mutexID;                      /*!< Mutex ID. */
#endif
#if CFG_EVENT_EN > 0
   OS_EventID eventID;                      /*!< Event ID. */
#endif
#if CFG_ROBIN_EN >0
   U16 timeSlice;                           /*!< Task time slice */
#endif
#if CFG_STK_CHECKOUT_EN >0
   OS_STK *stack;                          /*!< The top point of task. */
#endif
#if CFG_EVENT_EN > 0
   void* pmail;                             /*!< Mail to task. */
   struct TCB *waitNext;      /*!< Point to next TCB in the Event waitting list.*/
   struct TCB *waitPrev;       /*!< Point to prev TCB in the Event waitting list.*/
#endif
#if CFG_TASK_SCHEDULE_EN == 0
   FUNCPtr      taskFuc;
   OS_STK       *taskStk;
#endif
#if CFG_FLAG_EN > 0
   void* pnode;                            /*!< Pointer to node of event flag. */
#endif
#if CFG_TASK_WAITTING_EN >0
   U32 delayTick;                          /*!< The number of ticks which delay. */
#endif
   struct TCB *TCBnext;                    /*!< The pointer to next TCB. */
   struct TCB *TCBprev;                    /*!< The pointer to prev TCB. */
}OSTCB,*P_OSTCB;
```

stkPtr： 指向当前任务的栈顶指针。CooCox CoOS 允许每个任务拥有自己的栈，且大小任意。在每次任务切换时，CoOS 通过 stkPtr 所指定的栈来保存 CPU 的当前运行状态，以便在再次获得 CPU 运行时间时，能恢复至上一次的运行状态。由于 Cortex-M3 拥有 16 个 32 位通用寄存器来描述 CPU 的状态，故在 CoOS 中，任务的栈最小为 68 字节（最后 4 个字节用于检查堆栈溢出）。

prio： 用户指定的任务优先级，CooCox CoOS 支持多个任务享有同一优先级。

state： 任务状态。

taskID： 系统分配的任务 ID。在 CooCox CoOS 中，多个任务可以共享同一优先级，优先级并不能作为任务的唯一标识，因此在系统内用任务 ID 来区分不同的任务。

mutexID：任务等待的互斥体 ID。

eventID： 任务等待的事件 ID。

timeSlice：任务时间片的时间。

stack： 堆栈的栈底指针，用于进行堆栈溢出检查。

pmail： 发送给任务的消息指针。

waitNext：事件等待链表的下一个任务 TCB。

waitPrev：事件等待链表的前一个任务 TCB。

taskFuc： 任务的执行函数指针，用于重新激活任务。

taskStk： 任务的起始堆栈指针，用于重新激活任务。

pnode： 事件标志的结点指针。

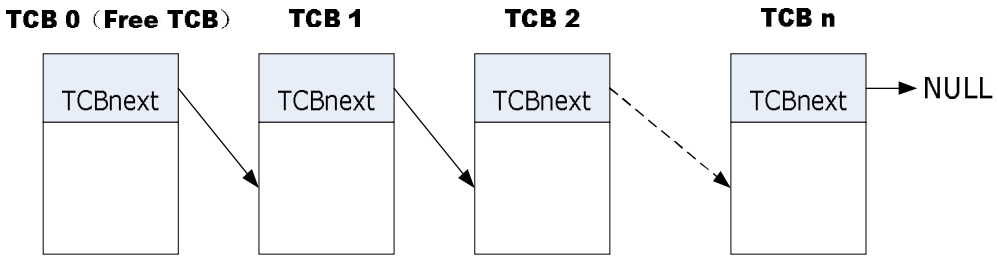
delayTick：任务处于延时状态时与前一个延时事件的时间差值。

TCBnext： 任务在就绪链表/延时链表/互斥体等待链表中的下一个任务 TCB。具体处于哪一链表由用户的任务状态及 TCB 中相关项决定。若当前任务为就绪态，则任务处于就绪链表。若为等待状态，则通过 mutexID 来判断：若 mutexID 不为 0xFFFFFFFF，则处于互斥链表中，否则处于延时链表中。

TCBprev： 任务在就绪链表/延时链表/互斥体等待链表中的前一个任务 TCB。具体处于哪一链表由用户的任务状态及相关事件标志决定。

每一次创建任务，系统都要从当前空闲的任务控制块链表中分配一项给当前任务。在 CooCox CoOS 中，系统通过 FreeTCB 来指定系统的空闲 TCB 指针，若 FreeTCB 为 NULL，则说明系统没有可分配的 TCB，这将导致创建任务失败。

在系统初始化时，CoOS 将系统内部所有可分配的 TCB 资源进行管理排序，通过链表的形式来反映当前的 TCB 状态，如下图所示：



P2.3.1 任务控制块链表

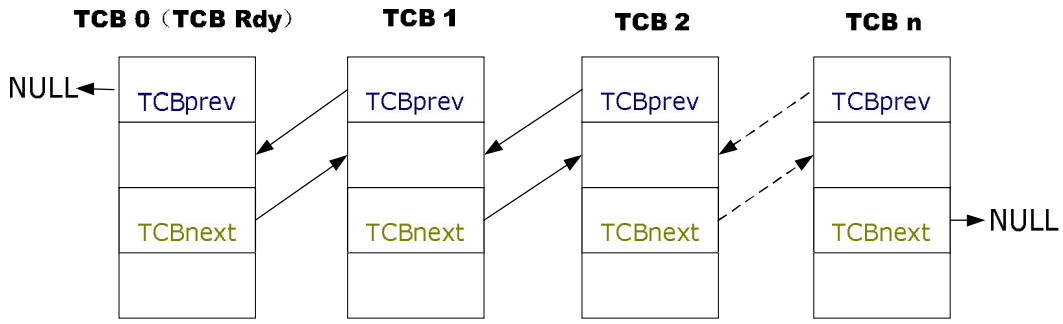
每成功创建一个任务，就将 FreeTCB 作为该任务的 TCB，而将 FreeTCB 的下一项作为新的 FreeTCB，直到 TCBnext==NULL 为止。删除任务和退出任务时，系统回收创建任务时分配出去的 TCB，并将其作为下一次分配的 FreeTCB，由此保证对已删除任务资源的重新利用。

2.4 任务就绪链表

CooCox CoOS 将所有就绪态任务的 TCB 以优先级高低的顺序通过双向链表链接在一起，保证了链表的第一项总是优先级最高、最需要调度的就绪态任务。

CooCox CoOS 允许多个任务共享同一优先级，因此就绪链表中不可避免地会出现相同优先级的任务。对此 CooCox CoOS 遵循“先入先出（FIFO）”的原则，将迟来的任务排在同一优先级的最后，这样所有处于同一优先级的任务就都能获得相应时间片的 CPU 运行时间。

CooCox CoOS用TCBRdy表示就绪链表的开始，即TCBRdy为就绪链表中优先级最高的任务TCB，故每一个任务调度只需要检查TCBRdy所指任务的优先级是否大于当前正在运行任务的优先级，这样可以最大限度地提升任务调度的效率。



P2.4.1 任务就绪链表

2.5 任务调度

CooCox CoOS 支持时间片轮转和优先级抢占两种任务调度机制。不同优先级任务间为优先级抢占调度，同级优先级任务间为时间片轮转调度。

CooCox CoOS 在以下三种情况下会发生任务调度：

1）比当前运行的任务拥有更高优先级的任务转为就绪态时

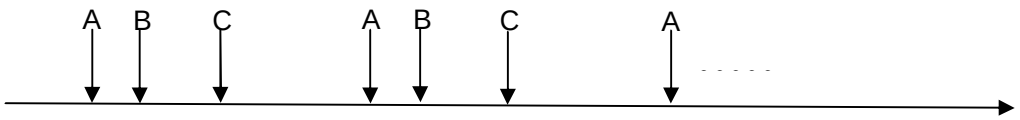
2）正在运行的任务状态发生改变，即从运行态变为等待态或休眠态时

3） 与当前任务是同一优先级的任务处于就绪态，且当前任务的时间片已到时

CooCox CoOS 会在每一个系统节拍中断退出及有任务状态改变时，调用任务调度函数，判断当前是否需要进行任务调度。

对于不同优先级任务之间的调度，CooCox CoOS 根据优先级抢占的原则，永远执行最高优先级的就绪任务。

对于多个同一优先级任务间的调度，则根据其自身任务的时间片进行轮转调度，系统执行完当前任务的时间片长度，就将控制权交给下一个同一优先级任务。下图就描述了三个依次进入就绪状态的相同优先级任务 A,B,C（时间片分别为 1，2，3）的系统运行状态：



P2.5.1 时间片轮询

在 CooCox 源码中，各情况下任务调度的实现方式如下：

程序 4　高优先级的任务就绪

```
/* Is higher PRI task coming in? */
If (RdyPrio < RunPrio)
{
    TCBNext            = pRdyTcb;    /* Yes, set TCBNext and reorder ready list*/
    pCurTcb->state     = TASK_READY;
    pRdyTcb->state     = TASK_RUNNING;
    InsertToTCBRdyList(pCurTcb);
    RemoveFromTCBRdyList(pRdyTcb);
}
```

程序 5　当前任务状态改变

```
/* Does Running task status change */
else if(pCurTcb->state != TASK_RUNNING)
{
    TCBNext             = pRdyTcb;    /* Yes, set TCBNext and reorder ready list*/
    pRdyTcb->state      = TASK_RUNNING;
    RemoveFromTCBRdyList(pRdyTcb);
}
```

程序 6　相同优先级的任务调度

```
/* Is it the time for robinning */
else if((RunPrio == RdyPrio) && (OSCheckTime == OSTickCnt))
{
    TCBNext             = pRdyTcb;     /* Yes, set TCBNext and reorder ready list*/
    pCurTcb->state      = TASK_READY;
    pRdyTcb->state      = TASK_RUNNING;
    InsertToTCBRdyList(pCurTcb);
    RemoveFromTCBRdyList(pRdyTcb);
}
```

2.6 临界区

与其它内核不一样的是：CooCox CoOS并不是通过开关中断来处理临界代码段，而是通过锁定调度器来实现的。因此相对于其它内核，CoOS拥有较短的中断屏蔽时间。

关中断的时间是实时内核开发商应提供的最重要的指标之一，因为这个指标关系到系统对实时事件的响应性。相比其它处理方式，通过锁定调度器能最大程度地提升系统的实时性。

由于CooCox CoOS是通过禁止任务调度来管理临界区的，故在此临界代码段内，用户的应用程序不得使用任何能将现行任务挂起的API。也就是说，在临界区内，系统不能调用CoExitTask()、CoSuspendTask()、CoTickDelay()、CoTimeDelay()、CoEnterMutexSection()、CoPendSem()、CoPendMail()、CoPendQueueMail()、CoWaitForSingleFlag()、CoWaitForMultipleFlags()等函数。

程序7　临界区

```
void Task1（void* pdata）
{
    ……
    CoSchedLock();              // Enter Critical Section
    ……                          // Critical Code
    CoSchedUnlock();            // Exit Critical Section
    ……
}
```

2.7 中断

在 CooCox CoOS 中，中断按照在其内部是否调用了系统 API 函数被划分为两类：调用了操作系统 API 的中断和与操作系统无关的中断。

对于与操作系统无关的中断，CooCox CoOS 不强制其进行任何处理，用户所做的任何操作与没有操作系统时一样。

而调用了操作系统 API 的中断，CooCox CoOS 在中断进入和退出时要求必须调用相关函数，如程序 8：

程序 8　调用系统 API 中断处理程序

```
void  WWDG_IRQHandler(void)
{
    CoEnterISR();            // Enter the interrupt
    isr_SetFlag(flagID);     // API function
    ..................;     // Interrupt service routine

    CoExitISR();             // Exit the interrupt
}
```

在 CooCox CoOS 中，所有可在中断服务程序里调用的系统 API 均以 isr_开头，如 isr_PostSem()、isr_PostMail()、isr_PostQueueMail()、isr_SetFlag()，其它任何 API 的调用，都有可能导致系统运行混乱。

在中断服务程序里调用相应的 API 函数，需要判断当前任务调度是否被锁定。若未被锁定，则正常调用。若被锁定，则需要发送一个相应的服务请求至服务请求列表，等待解锁调度器来响应请求列表里的服务请求。

3 时间管理

3.1 系统节拍

CooCox CoOS用systick中断实现系统节拍，用户需要在config.h文件中配置系统节拍的频率。CFG_CPU_FREQ用于表示CPU的时钟频率，在配置systick中断时，系统需要通过CPU时钟频率来确定具体的参数；CFG_SYSTICK_FREQ表示用户所需的系统节拍频率，CooCox CoOS支持1Hz到1000Hz的频率，实际值要由具体的应用来确定，系统默认为100Hz（即时间间隔为10ms）。

CooCox CoOS在每一次系统节拍中断服务程序中对系统时间进行加一操作，用户可以通过调用CoGetOSTime()来获得当前的系统时间。

除了对系统时间进行加一处理外，CooCox CoOS还会在系统节拍中断中检查延时链表和定时器链表是否为空，若链表不为空，则对链表首项的延时时间进行减一操作，并判断链表首项的等待时间是否到期，若到期，则调用相关的操作函数；否则跳过进行下一步操作。

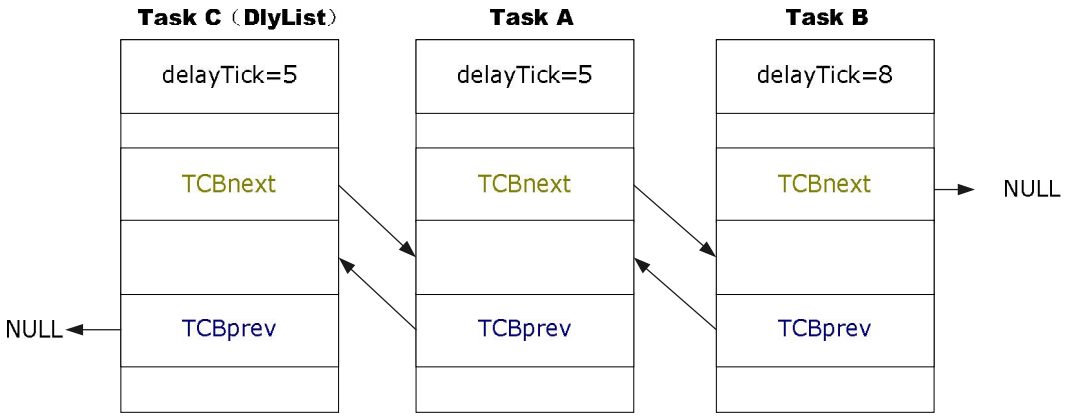
CooCox CoOS在系统节拍中断退出时，调用任务调度函数来判断当前系统内是否需要进行任务调度。

程序1　系统节拍中断处理

```
void SysTick_Handler(void)
{
    OSSchedLock++;                  /* Lock scheduler. */
    OSTickCnt++;                    /* Increment systerm time. */
    if(DlyList != NULL)             /* Have task in delay list? */
    {
        DlyList->delayTick--;       /* Decrease delay time of the list head. */
        if(DlyList->delayTick == 0) /* Delay time == 0? */
        {
            isr_TimeDispose();      /* Call hander for delay time list */
        }
    }
#if CFG_TMR_EN > 0
    if(TmrList != NULL)             /* Have timer in working? */
    {
        TmrList->tmrCnt--;          /* Decrease timer time of the list head. */
        if(TmrList->tmrCnt == 0)    /* Timer time == 0? */
        {
            isr_TmrDispose();       /* Call hander for timer list */
        }
    }
#endif
    OSSchedLock--;                  /* Unlock scheduler. */
    if(OSSchedLock==0)
    {
       Schedule();                  /* Call task scheduler */
    }
}
```

3.2 延时管理

CooCox CoOS通过延时链表管理所有任务的延时和超时。用户调用CoTickDelay()、CoTimeDelay()、CoResetTaskDelayTick()等函数向操作系统申请延时或者调用其他带超时的等待服务时，操作系统将这些延时的时间从短到长排序，然后插入到延时链表中。任务控制块中的delayTick域保存的是本任务与前一项任务延时时间之间的差值。链表第一项为时间延时或超时值，后续链表项中的值均为与前一项的差值。如任务A/B/C分别延时10/18/5，则在延时链表中为如下排序：



P3.2.1 任务延时链表

系统在每次系统节拍中断时会对延时链表的首项进行减一操作，直至其变为0后就将它从延时链表中移入就绪链表。在系统节拍中断中将时间到期的任务移出链表的同时，需要判断该任务是一次延时操作还是超时操作。对于延时操作的任务，CooCox CoOS在其被移出延时链表的同时将其插入就绪链表。而对于超时操作的任务，CooCox CoOS首先判断是由哪一事件所引发的超时，然后将任务从该事件的等待链表中移入就绪链表。

CooCox CoOS的系统延时在以下条件下不能保证延时的精确性：

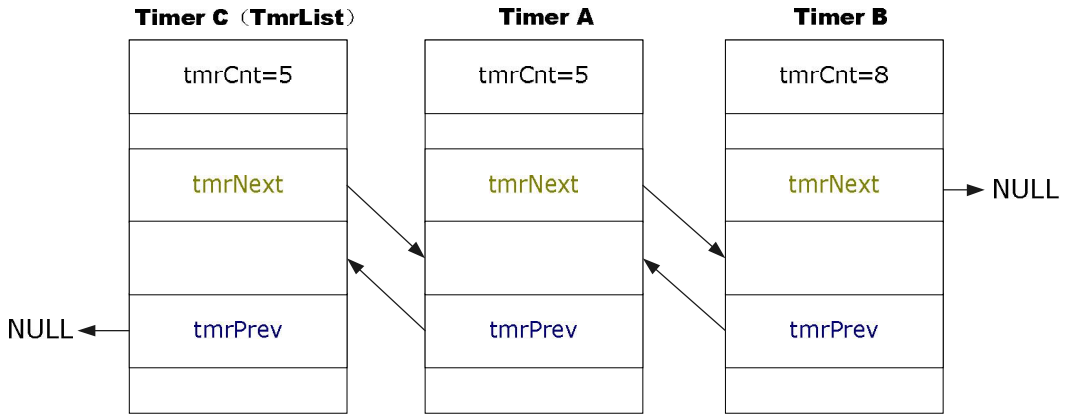
1）在任务A延时的过程中，有高优先级的任务抢占运行，延时时间将取决于高优先级任务的运行时间；

2）在任务A延时的过程中，有与任务A同一优先级的任务B抢占运行，延时时间将取决于就绪链表中与任务A同一优先级任务的任务数和它们各自的时间片长度，以及延时时间到期的时间点。

3.3 软件定时器

软件定时器是CooCox CoOS以系统节拍为基准时钟源的一个高精度定时器。CooCox CoOS最多支持32个软件定时器，每一个定时器都可将其工作模式设定为循环运行模式或单次运行模式。

用户可以调用CoCreateTmr()创建一个软件定时器。创建一个软件定时器时，系统会分配一个相应的定时器控制块来描述定时器当前的状态，创建成功后，定时器默认为停止状态，需要用户调用CoStartTmr()后，定时器才能正常工作。对于正常工作的定时器，CooCox CoOS通过定时器链表进行管理，和延时链表一样，定时链表同样以到期时间的长短进行排序：早到期的任务排在链表前，并将所有定时器的到期时间与链表前一个定时器的到期时间作差后存入定时器控制块。如定时器A/B/C分别定时为10/18/5，则在定时器链表中为如下排序：



P3.3.1 定时器链表

软件定时器启动后就完全独立于其他模块，只与系统节拍中断有关。CooCox CoOS将用户工作的定时器都按照到期时间的长短链入定时器链表。在每次系统节拍中断时，对定时器链表的首项进行减一操作（直至其变为0）。

当定时器等待时间到期时，对于循环运行的定时器，CooCox CoOS会根据用户给定的tmrReload重新设定下一次的定时后再链入定时器链表；而对于单次运行的定时器，则将其移出链表，并将其运行状态设定为停止态。

由上可知，定时器的到期时间只取决于系统节拍的个数，而与是否有高优先级任务在运行或创建该定时器的任务是否处于就绪状态等无关。

软件定时器提供了一个供用户在 systick 中断内部进行操作的函数入口（软件定时器的回调函数）。但是还有一些API是不能调用的，它们的调用会引起一些错误。下面列出不能调用的API及不能调用原因：

1.不能调用功能不相符的API，例如CoEnterISR()、CoExitISR()，因为软件定时器的回调函数不是一个ISR。

2.因为软件定时器的回调函数不是一个任务，而是一个可能在所有任务中被调用的

函数，故会改变当前任务状态的API是不能调用的，例如CoExitTask() \ CoEnterMutexSection() \ CoLeaveMutexSection() \ CoAcceptSem() \ CoPendSem() \ CoAcceptMail() \ CoPendMail() \ CoAcceptQueueMail() \ CoPendQueueMail() \ CoAcceptSingleFlag() \ CoAcceptMultipleFlags() \ CoWaitForSingleFlag() \ CoWaitForMultipleFlags()。

3.由于每一个软件定时器到期时的回调函数均在 systick 中断内部执行，这就要求软件定时器的代码必须精简，不能长时间运行而影响systick 中断的精度，故不能调用CoTimeDelay()和CoTickDelay()，不但会影响精度还可能会导致内核错误。

4 内存管理

4.1 静态内存分配

静态内存分配适用于在代码编译时就已经确定需要占用多少内存的情况，在整个代码运行过程中不能进行释放和重新分配。相对于动态内存分配，静态内存分配不需要消耗 CPU 资源，也不会出现分配不成功的现象（因为分配不成功将直接导致编译失败），因此其速度更快，也更安全。

在 CooCox CoOS 中，各个模块控制块所需的内存都是静态分配的，如任务控制块（TCB）、事件控制块（ECB）、事件标志控制块（FCB）、事件标志结点（FLAG_NODE）等等。

程序 1　CooCox CoOS 任务控制块的空间分配

```
config.h
    #define CFG_MAX_USER_TASKS     (8)     // 确定系统内用户最多任务数
task.h
    #define   SYS_TASK_NUM             (1)     // 确定系统任务数
task.c
                                                // 静态分配 TCB 空间
    OSTCB   TCBTbl[CFG_MAX_USER_TASKS+SYS_TASK_NUM];
```

4.2 动态内存分配

动态内存分配适用于系统运行时内存块大小不能在代码编译时确定，而要根据代码的运行环境来确定的情况。可以说，静态内存分配是按计划分配，动态内存分配是按需要分配。

虽然动态内存分配比较灵活，能极大的提高内存的利用率，但由于每一次的分配和释放都需要消耗 CPU 资源，并且有可能存在分配失败和内存碎片问题，所以每一次动态内存分配都需要程序员来确定是否成功。

下面介绍一下常规的动态内存分配方式——malloc()和 free()的实现方式。在通常的编译器或者系统内核中，内存块根据其是否已被分配分别存放于空闲链表和已分配链表中。

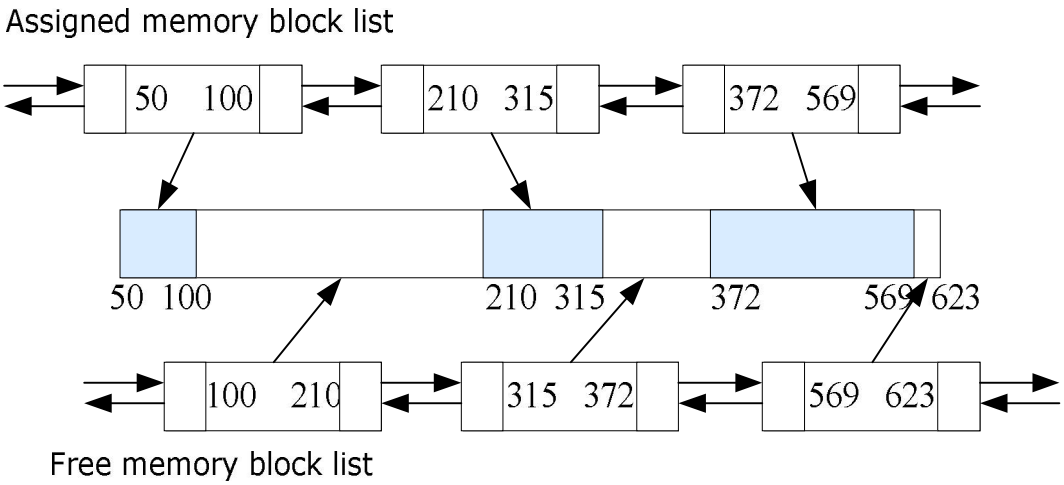
调用 malloc 函数时，系统相关操作步骤如下：

1）沿空闲链表寻找一个大小满足用户需求的内存块。由于查找算法不同，系统所找到的符合用户需求的内存块也不尽相同。而最常用的查找算法为最先匹配算法，即使用第一块满足用户需求的内存块进行分配，这样可以避免每次分配时都遍历所有的空闲链表项。

2）将该内存块一分为二：其中一块的大小与用户请求的大小相等，另一块存储剩余的字节。

3）将分配给用户的那一块内存传给用户，并将另一块（如果有的话）返回到空闲链表上。

调用 free 函数时，系统将用户释放的内存块链接到空闲链表上，并判断该内存块前后的内存块是否空闲，若为空闲则合并成一个大的内存块。



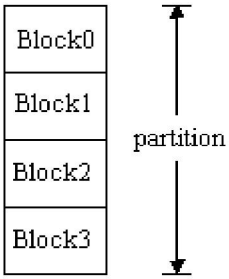
P4.2.1 内存管理链表

由上述可知，多次内存分配和释放后，空闲链表会被分割成很多的小块，若此时用户申请一个大的内存块，则空闲链表上可能没有满足用户需求的片段了，这就产生了所谓的内存碎片。另外，在每次释放内存时，系统都需要从空闲链表第一项开始检查整个链表以确定该内存块需要插回的位置，这就导致释放内存的时间不确定或过于漫长。

为了解决上述问题，CooCox CoOS 提供了两种分区机制：固定长度分区和可变长度分区。

4.2.1 固定长度分区

在 CooCox CoOS 中提供固定长度分区的内存分配方式，系统根据用户给定的参数，将一个大的内存块分成整数个大小固定的内存块，通过链表的形式将这些内存块连接起来，用户可以从中分配和释放固定大小的内存块，这样既保证了分配和释放时间的固定，也解决了内存碎片的问题。



P4.2.2 固定长度分区

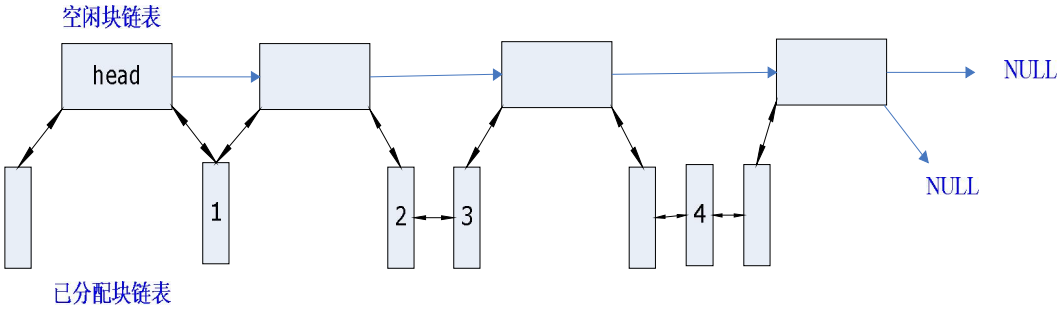
CooCox CoOS 一共可以管理 32 个大小不同的固定长度分区。用户可以通过调用 CoCreateMemPartition()来创建一个固定长度分区，成功创建内存分区后，用户可以调用 CoGetMemoryBuffer()和 CoFreeMemoryBuffer()来进行内存块的分配和释放，也可以调用 CoGetFreeBlockNum()来获得当前内存分区空闲内存块的个数。

程序 2　固定长度分区的创建和使用

```
U8 memPartition[128*20];
OS_MMID memID;
void   myTask (void* pdata)
{
     U8* data;
     memID = CoCreateMemPartition(memPartition,128,20);
     if(CoGetFreeBlockNum(memID ) != 0)
     {
          data = (U8*)CoGetMemoryBuffer(memID );
     }
     ..................
     CoFreeMemoryBuffer(memID ,data);
}
```

4.2.2 可变长度分区

从上文分析的常规动态内存实现可知：内存释放时需要同时操作空闲链表和已分配链表，这要求比较长的操作时间，且影响CPU运行效率。对此CooCox CoOS重新设计链表，使分配和释放内存均只需要搜索一张表即可。



P4.2.3 可变长度分区链表

由上图可知，系统内的所有空闲内存块可单独组成一个单向链表，这样内存分配时，查表比较方便。对于所有的内存块，不管其是已分配还是空闲，CooCox CoOS均通过一个双向链表将其链接起来。因此当一个已分配的内存块释放时，无需再从空闲链表头开始查找确定该内存块的插入点，而只需在该内存块双向链表找到前一个空闲块，即可将此内存块插入至空闲链表，这极大地提高了内存块的释放速度。

在CooCox CoOS中，由于所有的内存块都要求4字节对齐（若需要分配的空间未4字节对齐，强制为4字节对齐），所以内存块head所保存的前后内存块地址的后两位均无效。因此，CooCox CoOS通过最低位来判断该内存块是否为空闲块，若最低位为0，则表示为空闲块，否则为已分配内存块。因此若内存块链表指向的是空闲块，直接获得链表地址即可。若指向的是已分配内存块，则需要进行减一操作。

程序3　已分配内存块head

```
typedef struct UsedMemBlk
{
  void* nextMB;
  void* preMB;
}UMB,*P_UMB;
```

程序4　空闲内存块head

```
typedef struct FreeMemBlk
{
  struct FreeMemBlk* nextFMB;
  struct UsedMemBlk* nextUMB;
  struct UsedMemBlk* preUMB;
}FMB,*P_FMB;
```

对于内存块1和2，当其释放时，内存块本身保存了前一个空闲内存块地址，因此极容易插回空闲链表，这时候程序只需要根据该内存块的后一内存块是否为空闲块来决定是否进行合并。

对于内存块3和4，当其释放时，其前一个内存块并不是空闲内存块，这时还需要通过

双向链表获得此时的前一个空闲内存块地址，才能将其插回至空闲链表。

程序5　获得前一个空闲内存块地址

```
P_FMB GetPreFMB(P_UMB usedMB)
 {

    P_UMB preUMB;
    preUMB = usedMB;
    while(((U32)(preUMB->preMB)&0x1))     /* Is previous MB as FMB?*/
    {                                     /* No,get previous MB */
        preUMB = (P_UMB)((U32)(preUMB->preMB)-1);
    }
    return (P_FMB)(preUMB->preMB);        /* Yes,return previous MB*/
}
```

用户可以在config.h文件中确定是否需要在内核内添加可变长度分区，并同时设置该内存分区的大小。

```
Config.h
#define CFG_KHEAP_EN              (1)
#if CFG_KHEAP_EN >0
#define KHEAP_SIZE                (50)          // size(word)
#endif
```

确定开启可变长度分区后，用户可以在代码里调用 CoKmalloc()和 CoKfree()来实现内存的分配和释放，CoKmalloc()申请的内存大小以字节为单位。

程序 7　可变长度分区的使用

```
void myTask (void* pdata)
{
     void* data;
     data = CoKmalloc(100);
     ......
     CoKfree(data );
}
```

4.3 堆栈溢出检查

堆栈溢出是指任务在运行时使用的堆栈大小超过了分配给任务堆栈的大小，结果导致向堆栈外的内存写入了数据。这样可能导致覆盖了系统或者其他任务的数据，也可能会导致内存访问异常。在多任务内核中，为每一个任务分配的堆栈大小均为固定，在系统运行时，若发生堆栈溢出且没有做处理，则可能导致系统崩溃。

在CooCox CoOS中创建任务时，系统将在任务控制块中保存堆栈的栈底地址，并在栈底地址所对应的内存块中写入一特殊值，用此来判断堆栈是否溢出。CooCox CoOS会在每次任务调度时检查是否发生堆栈溢出。

程序8　堆栈溢出检查

```
if((pCurTcb->stkPtr < pCurTcb->stack)||(*(U32*)(pCurTcb->stack) != MAGIC_WORD))
{
    CoStkOverflowHook(pCurTcb->taskID);          /* Yes,call hander */
}
```

当任务发生堆栈溢出时，系统自动调用CoStkOverflowHook（taskID）函数，用户可以在此函数中添加对堆栈溢出的处理，函数的参数为发生堆栈溢出的任务ID号。

程序9　堆栈溢出处理函数

```
void CoStkOverflowHook(OS_TID taskID)
{
    /* Process stack overflow in here */
    for(; ;)
    {
        ...
    }
}
```

5 任务间的同步与通信

5.1 任务间的同步

任务同步是指一个任务需要等待另一个任务或中断服务程序发送相应的同步信号后才能继续执行。在 CooCox CoOS 中，提供了信号量、互斥区域和事件标志来实现任务间的同步。

5.1.1 信号量

信号量为系统处理临界区和实现任务间同步的问题提供了一种有效的机制。

信号量的行为可以用经典的PV 操作来描述：

```
P操作：while( s==0); s--;
V操作：s++;
```

在 CooCox CoOS 中，用户可以调用 CoCreateSem()来创建一个信号量，成功创建一个信号量之后，用户就可通过调用 CoPendSem()、CoAcceptSem()来获得一个信号量，两者不同的是，对于 CoPendSem()，如果当前没有信号量空闲，则将超时等待到该信号量被释放，而对于 CoAcceptSem()则立刻返回错误。用户也可以在任务体内调用 CoPostSem()或者中断服务程序内调用 isr_PostSem()来释放一个信号量，以实现彼此同步。

程序 1　信号量创建

```
ID0 = CoCreateSem(0,1,EVENT_SORT_TYPE_FIFO); // initCnt=0,maxCnt=1,FIFO
ID1 = CoCreateSem(2,5,EVENT_SORT_TYPE_PRIO); // initCnt=2,maxCnt=5,PRIO
```

程序 2　信号量的使用

```
void myTaskA(void* pdata)
{
    ..........
    semID = CoCreateSem(0,1,EVENT_SORT_TYPE_FIFO);
    CoPendSem(semID,0);
    ..........
}
void myTaskB(void* pdata)
{
    ......
    CoPostSem(semID);
    ......
}
void myISR(void)
{
    CoEnterISR();
    ......
    isr_PostSem(semID);
    CoExitISR();
}
```

5.1.2 互斥区域

在 CooCox CoOS 中，互斥区域解决了“互相排斥”的问题。互斥区域禁止多个任务同时进入受保护的代码“临界区”（critical section）。因此，在任意时刻，只能有一个任务进入这样的代码保护区。

在 CooCox CoOS 中，互斥区域还考虑了优先级反转问题，并通过优先级继承的方法解决了有可能出现优先级反转的问题。

优先级反转是指高优先级任务等待低优先级任务释放资源，而低优先级任务又正在等待中等优先级任务的现象。

目前两种经典的防止反转的方法：

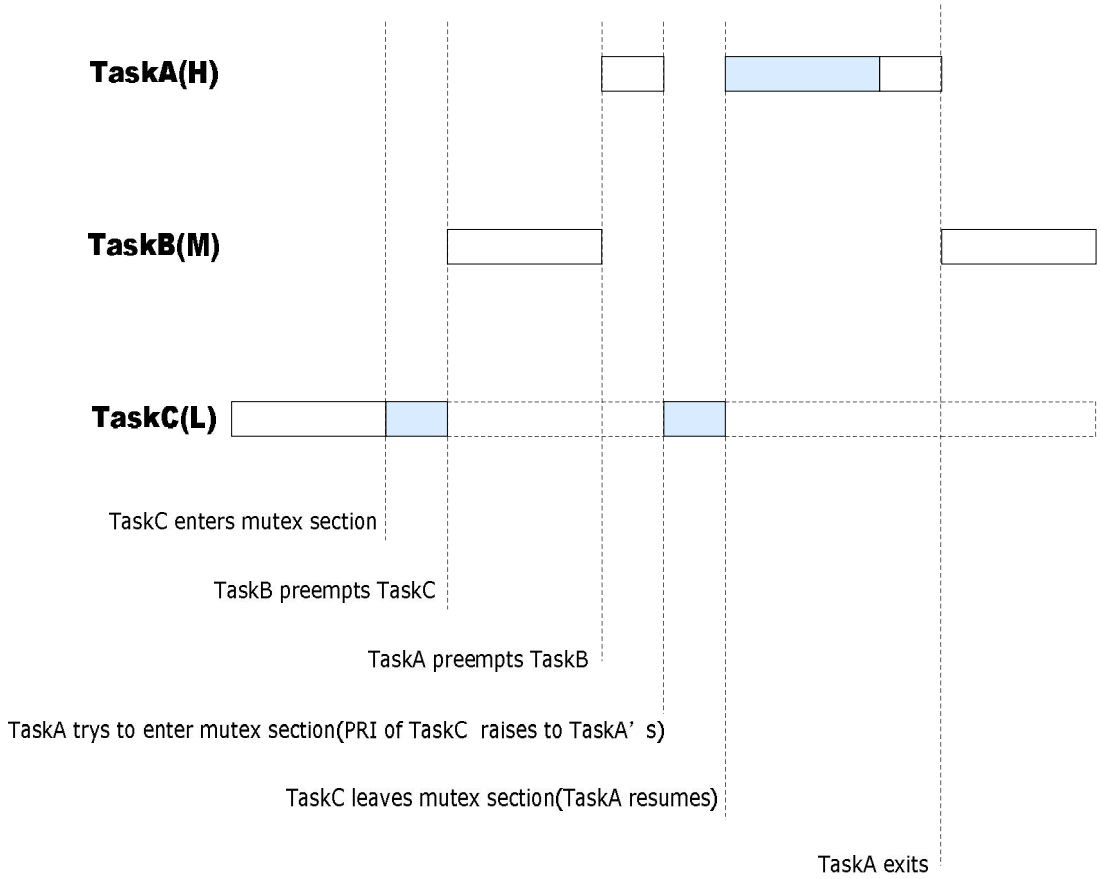
1）优先级继承策略：当前占有临界资源的任务继承所有申请该临界资源任务的最高优先级作为其优先级，当其退出临界区时，恢复至初始优先级。

2）优先级天花板策略：将申请（占有）某资源的任务的优先级提升至可能访问该资源的所有任务中优先级最高的任务的优先级。(这个优先级称为该资源的优先级天花板)

优先级继承策略对任务执行流程的影响相对较小，因为只有当高优先级任务申请已被低优先级任务占有的临界资源这一事实发生时，才抬升低优先级任务的优先级。而天花板策略是谁占有就直接升到最高。

CooCox CoOS通过优先级继承的方法来防止优先级的反转。

下图描述了三个任务在CooCox CoOS中有互斥区域时的任务调度，TaskA拥有最高优先级，TaskC拥有最低优先级，蓝框表示互斥区域。



P5.1.1 有互斥区域时的任务调度

用户在 CooCox CoOS 中，可以调用 CoCreateMutex()来创建一个互斥区域，调用 CoEnterMutexSection()和 CoLeaveMutexSection()进入和离开互斥区域，以实现对临界区代码的保护。

程序 3　互斥区域的使用

```
void myTaskA(void* pdata)
{
     mutexID = CoCreateMutex();
     CoEnterMutexSection(mutexID );    // 进入互斥区域
     .....                             // 临界区代码
     CoLeaveMutexSection(mutexID );    // 离开互斥区域
}
void myTaskB(void* pdata)
{
     CoEnterMutexSection(mutexID );    // 进入互斥区域
     .....                             // 临界区代码
     CoLeaveMutexSection(mutexID );    // 离开互斥区域
}
```

5.1.3 事件标志

当某一任务要与多个事件同步时，需要使用事件标志。若该任务仅与任一个事件同步，称为独立型同步(逻辑或关系)；若与多个事件都发生同步，则称之为关联型同步(逻辑与关系)。

在CooCox CoOS中，最多支持32个事件标志同时存在。CooCox CoOS支持多个任务等待单个事件或多个事件的发生。在CooCox CoOS中，当等待任务所等待的事件标志处于未就绪状态时，这些任务处于不可调度状态。但一旦等待对象变成就绪状态，任务将很快恢复运行。

一个任务成功等待到事件标志后，根据事件标志的类型不同，会有不同的成功等待副作用。在CooCox CoOS中，事件标志有两种类型，人工重置和自动重置。当一个任务成功等待到自动重置事件标志，系统会自动将该事件标志变为未就绪状态；而对于人工重置事件标志，则无副作用。故当人工重置事件就绪后，等待该事件的所有任务均可变为就绪状态，直到用户调用CoClearFlag()将事件标志设置为非就绪态。当一个自动重置事件得到通知，等待该事件标志的任务只有一个变成可调度状态。而且因为事件标志的等待列表按照FIFO的原则排列，因此对于自动重置事件而言，只有等待列表的第一个任务变为就绪状态，其它等待该事件标志的任务仍处于等待状态。

如任务 A/B/C 同时等待事件标志 I，若该事件标志为人工重置事件，那么当事件标志 I 就绪时，同时会通知所有的等待任务，即将任务 A/B/C 均从等待状态变为就绪态，且插入就绪链表。若等待的事件标志 I 为自动重置事件，且任务 A/B/C 按先后顺序排列于等待链表，则当事件标志 I 就绪时，等待其通知任务 A 后，就变为未就绪状态，所以任务 B/C 仍在等待链表中，等待下一次的事件标志 I 就绪。

在 CooCox CoOS 中，用户可以调用 CoCreateFlag()来创建一个事件标志，创建完事件标志，用户可以通过 CoWaitForSingleFlag()、CoWaitForMultipleFlags()来等待单个或多个事件标志。

程序 4　等待单个事件标志

```
void myTaskA(void* pdata)
{
    ..........
    flagID = CoCreateFlag(0,0);        // 人工重置，初始为非就绪态
    CoWaitForSingleFlag(flagID,0);
    ..........
}
void myTaskB(void* pdata)
{
    ......
    CoSetFlag(flagID);
    ......
}
```

程序 5　等待多个事件标志

```
void myTaskA(void* pdata)
{
    U32 flag;
    StatusType err;
    ..........
    flagID1 = CoCreateFlag(0,0);        // 人工重置，初始为非就绪态
    flagID2 = CoCreateFlag(0,0);        // 人工重置，初始为非就绪态
    flagID3 = CoCreateFlag(0,0);        // 人工重置，初始为非就绪态
    flag = flagID1 | flagID2 | flagID3;
    CoWaitForMultipleFlags(flag,OPT_WAIT_ANY,0,&err);
    ..........
}

void myTaskB(void* pdata)
{
    ......
    CoSetFlag(flagID1);
    ......
}

void myISR(void)
{
    CoEnterISR();
    ......
    isr_SetFlag(flagID2);
    CoExitISR();
}
```

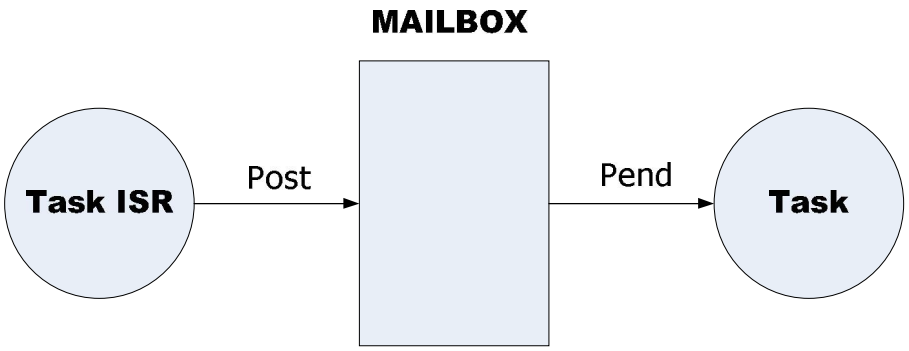
5.2 任务间的通信

任务间或任务与中断服务程序间有时需要进行信息的传递，这种信息传递即为任务间的通信。任务间的通信有两种途径：通过全局变量来实现或发消息给另一个任务。

用全局变量时，必须保证每个任务或中断服务程序独享该变量。在中断服务中保证独享的唯一办法是关中断。如果两个任务共享某变量，各任务要独享该变量可以先关中断再开中断或使用信号量(见5.1节)。请注意，任务只能通过全局变量与中断服务程序通信，而且任务并不知道全局变量什么时候被中断服务程序修改了（除非中断程序以信号量方式向任务发信号或者是该任务以查询方式不断周期性地查询变量的值）。在这种情况下，CooCox CoOS提供了邮箱和消息队列来避免以上问题。

5.2.1 邮箱

系统或用户代码可以通过内核服务来给任务发送消息。典型的消息邮箱也称作交换消息，是指一个任务或一个中断服务程序利用一个指针型变量，通过内核服务来把一则消息(即一个指针)放入邮箱。同样，一个或多个任务可以通过内核服务来接收这则消息。发送消息的任务和接收消息的任务约定，该指针指向的内容就是那则消息。



P5.2.1 邮箱

CooCox CoOS的邮箱就是一个典型的消息邮箱。在CooCox CoOS中，邮箱由两部分组成：一个是邮箱的信息，用一个void指针来表示；另一个是由等待该邮箱的任务组成的等待链表。邮箱的等待链表支持两种排序方式，FIFO和优先级抢占，具体选择哪一种方式，由用户在创建邮箱的时候决定。

在CooCox CoOS中，用户可以调用CoCreateMbox()来创建一个邮箱，成功创建一个邮箱后，邮箱内并无消息存在，用户可在任务体或中断服务程序中调用CoPostMail()或isr_PostMail() 向邮箱发送一则消息，也可以通过CoPendMail()或CoAcceptMail()从邮箱中获得一则消息。

程序6　邮箱的使用

```
void myTaskA(void* pdata)
{
    void* pmail;
    StatusType err;
    ..........
    mboxID = CoCreateMbox(EVENT_SORT_TYPE_PRIO);    // 优先级抢占排序
    pmail = CoPendMail(mboxID,0,&err);
    ..........
}
void myTaskB(void* pdata)
{
    ......
    CoPostMail(mboxID,"hello,world");
    ......
}
void myISR(void)
{
    CoEnterISR();
    ......
    isr_PostMail(mboxID,"hello,CooCox");
    CoExitISR();
}
```

5.2.2 消息队列

消息队列实际上就是邮箱阵列，用于给任务发送消息。通过内核提供的服务，任务或中断服务子程序可以将多个消息(该消息的指针)放入消息队列。同样，一个或多个任务可以通过内核服务从消息队列中取出消息。发送和接收消息的任务约定，传递的消息实际上就是指针所指向的内容。

消息队列不同于邮箱的一点是,邮箱只能存放一个消息,而消息队列则可存放多个消息。在CooCox CoOS里，一个队列可容纳的消息个数是在创建该队列时决定的。

在CooCox CoOS中，消息队列由两部分组成：一个是指示该消息队列的结构体；另一个是由等待该消息队列的任务组成的等待任务表。消息队列的等待链表支持两种排序方式，FIFO和优先级抢占，具体选择哪一种方式，由用户在创建消息队列的时候决定。

在CooCox CoOS中，用户可以调用CoCreateQueue()来创建一个消息队列，成功创建消息队列后，消息队列内并无消息存在，用户可在任务体或中断服务程序中调用CoPostQueueMail()或isr_PostQueueMaill()向消息队列发送一则消息。亦可以通过CoPendQueueMail()或CoAcceptQueueMail()从消息队列中获得一则消息。

程序7　消息队列的使用

```
void myTaskA(void* pdata)
{
     void* pmail;
     Void* queue[5];
     StatusType err;
     ..........
     queueID = CoCreateQueue(queue,5,EVENT_SORT_TYPE_PRIO); //5 级，优先级抢占排序
     pmail = CoPendQueueMail(queueID ,0,&err);
     ..........
}
void myTaskB(void* pdata)
{
     ......
     CoPostQueueMail(queueID ,"hello,world");
     ......
}
void myISR(void)
{
     CoEnterISR();
     ......
     isr_PostQueueMail(queueID ,"hello,CooCox");
     CoExitISR();
}
```

6 API 手册

6.1 系统管理

6.1.1 CoInitOS()

函数原型：

void CoInitOS (void);

功能描述：

系统初始化。

函数参数：

无

返回值：

无

示例：

```
#include "CCRTOS.h"
#define TASK0PRIO    10
OS_STK Task0Stk[100];
OS_TID Task0Id;
void Task0 (void *pdata);
int main(void)
{
  System_init ();
  CoInitOs ();                  // Initialize CoOS
   ...
  Task0Id = CoCreateTask (Task0, (void *)0, TASK0PRIO , Task0Stk[99], 100);
   ...
  CoStartOS ();                    // Start CoOS
  }
void Task0 (void *pdata)
{
  ...
  for(;;)
  {
       ...
  }
}
```

备注：

1）CooCox CoOS 要求用户在调用 CooCox CoOS 任何其他服务之前首先调用系统初始化函数 OsInit()。
2）在调用 OsInit()之前需要设定好CPU时钟，并做好OS的配置选项。
3）在 OS 初始化时锁定调度器，并创建第一个任务 CoIdleTask。

6.1.2 CoStartOS ()

函数原型：

void CoStartOS (void);

功能描述：

系统开始运行。

函数参数：

无

返回值：

无

示例：

```
#include "CCRTOS.h"
#define TASK0PRIO    10
OS_STK Task0Stk[100];
OS_TID Task0Id;
void Task0 (void *pdata);
int main(void)
{
  System_init();
  CoOsInit();                  // Initialize CoOS
    ...
Task0Id = CoCreateTask (Task0, (void *)0, TASK0PRIO , Task0Stk[99], 100);
    ...
  CoOsStart();                 // Start CoOS
  }
void Task0 (void *pdata)
{
  ...
  for(;;)
  {
       ...
  }
}
```

备注：

1）在 CoStartOS()之前，用户必须先创建一个应用任务，否则 OS 将一直处于 IdleTask()函数体内。

2）OS 启动过程中，任务调度器解锁。

6.1.3 CoEnterISR()

函数原型：

void CoEnterISR(void);

功能描述：

系统进入中断。

函数参数：

无

返回值：

无

示例：

```
#include "CCRTOS.h"
void EXTI0_IRQHandler(void)
  {
  CoEnterISR();          // Enter ISR
      ...
  /* Process interrupt here */
      ...
  CoExitISR();           // Exit ISR
}
```

备注：

1）系统进入中断。中断嵌套层数值 OSIntNesting 相应加 1。

2）与 CoExitISR()成对使用。

6.1.4 CoExitISR()

函数原型：

void CoExitISR(void);

功能描述：

系统退出中断。

函数参数：

无

返回值：

无

示例：

```
#include "CCRTOS.h"
void EXTI0_IRQHandler(void)
{
  CoEnterISR();          // Enter ISR
      ...
  /* Process interrupt here */
      ...
  CoExitISR();           // Exit ISR
}
```

备注：

1）若在中断服务程序里调用了系统的 API 函数，则在所有中断完全退出后需要进行一次任务调度；

2）当系统退出中断时，中断嵌套计数器-OSIntNesting 减 1。当 OSIntNesting 减到 0 时，进行一次任务调度。

3）与 CoEnterISR()成对使用。

6.1.5 CoSchedLock()

函数原型：

void CoSchedLock(void);

功能描述：

锁定任务调度器。

函数参数：

无

返回值：

无

示例：

```
#include "CCRTOS.c"
void Task0 (void *pdata)
{
      .....
  CoSchedLock();
      .....
   /* Process critical resources  here */
      .....
  CoSchedUnlock();
      .....
}
```

备注：

1）任务调度被关闭，OSSchedLock 相应加 1。任务调度器被锁期间，被保护的共享数据和资源只能被当前任务使用，这在一定程序上保证了代码的顺序执行。

2） 与 CoSchedUnlockOs()成对使用。

6.1.6 CoSchedUnlock()

函数原型：

void CoSchedUnlock(void);

功能描述：

解锁任务调度器。

函数参数：

无

返回值：

无

示例：

```
#include "CCRTOS.c"
void Task0 (void *pdata)
{
     .....
  CoSchedLock();
     .....
   /* Process critical resources  here */
     .....
  CoSchedUnlock();
     .....
}
```

备注：

1）任务调度被打开，OSSchedLock相应减 1。当OSSchedLock 减至0时，运行任务调度。

2）与OsSchedLock()成对使用。

6.1.7 CoGetOSVersion()

函数原型：

OS_VER CoGetOSVersion(void);

功能描述：

获得当前 CoOS 版本号。

函数参数：

无

返回值：

CoOS 版本号

示例：

```
#include "CCRTOS.H"
void TaskN (void *pdata)
{
    U16 version;
    U8 Major,Minor;
    ....
    version = CoGetOSVersion();
    // Get Major Version
    Major = ((version>>12)&0xF) * 10 + (version>>8)&0xF;
    // Get Minor Version
    Minor = ((version>>4)&0xF) * 10 + version&0xF;
    printf("Current OS Version: %d.%02d\n",Major,Minor);
    ....
}
```

备注：

1) 函数返回值是一个16位的二进制数，需要右移8位才能获得实际版本。例如，0x0101 表示当前 CoOS 版本号为1.01。

6.2 任务管理

6.2.1 CoCreateTask()&CoCreateTaskEx()

CoCreateTask()

函数原型：

```
OS_TID CoCreateTask (
                        FUNCPtr     task,
                        void*       argv,
                        U8          prio,
                        OS_STK*     stk,
                        U16         stkSz,
                        )
```

功能描述：

创建一个任务并返回任务 ID。

函数参数：

[IN] task
　　创建任务的函数体

[IN] argv
　　任务函数体的传入参数列表

[IN] prio
　　任务优先级

[IN] stk
　　任务堆栈起始地址

[IN] stkSz
　　任务堆栈大小（单位为字）

返回值：

任务ID，　创建任务成功
-1，　　创建任务失败

示例：

```
#include "CCRTOS.h"
#define TASKMPRIO         11
#define TASKMSTKSIZE      100
OS_STK TaskMStk[TASKMSTKSIZE];
OS_TID TaskMId;
void TaskM (void *pdata);
void TaskN (void *pdata)
{
      …
      TaskMId = CoCreateTask (TaskM,
                              (void *)0,
                              TASKMPRIO,
                              &TaskMStk[TASKMSTKSIZE-1],
                              TASKMSTKSIZE);
      if (TaskMID==E_CREATE_FAIL)
      {
            printf("Task Create Failed !\n");
      }
      else
      {
            printf("Task ID %d\n",TaskMId);
      }
      …
}
void TaskM (void *pdata)
{
      …
}
```

备注：

1） 每创建一个任务就会分配一个 TCB。

2） 一旦一个任务被成功创建，该任务将处于就绪状态。

3）若当前创建的任务优先级高于正在运行的任务，则立刻转向刚创建的任务，开始运行。

4）stkSz 可取的最大值为 0xfff。

CoCreateTaskEx()

函数原型：

```
OS_TID CoCreateTaskEx (
                        FUNCPtr task,
                        void *argv,
                        U8 prio,
                        OS_STK *stk,
                        U16 stkSz,
                        U16 timeSlice,
                        BOOL   isWaitting
                       )
```

功能描述：

创建一个任务。

函数参数：

[IN] task
　　创建任务的函数体

[IN] argv
　　任务函数体的传入参数列表

[IN] prio
　　任务优先级

[IN] stk
　　任务堆栈起始地址

[IN] stkSz
　　任务堆栈大小

[IN] timeSlice
　　任务运行的时间片，若传入0，则设置为系统默认长度。

[IN] isWaitting
　　任务创建时的起始状态。若为 TRUE，任务创建后处于挂起等待状态。

返回值：

任务ID，　创建任务成功
-1，　　创建任务失败

示例：

```
#include "CCRTOS.h"
#define TASKMPRIO         11
#define TASKMSTKSIZE      100
#define TASKMtimeSLICE    10
OS_STK TaskMStk[TASKMSTKSIZE];
OS_TID TaskMId;
void TaskM (void *pdata);
void TaskN (void *pdata)
{
     ...
     TaskMId = CoCreateTaskEx(TaskM,
                              (void *)0,
                              TASKMPRIO,
                              &TaskMStk[TASKMSTKSIZE-1],
                              TASKMSTKSIZE,
                              TASKMtimeSLICE,
                              FASLE);
     if (TaskMID==E_CREATE_FAIL)
    {
          printf("Task Create Failed !\n");
     }
    else
    {
          printf("Task ID %d\n",TaskMId);
     }
     ...
}
void TaskM (void *pdata)
{
     ...
}
```

备注：

1）每创建一个任务就会分配一个 TCB。

2）一旦一个任务被成功创建，该任务将处于就绪状态。

3）若当前创建的任务优先级高于正在运行的任务，则立刻转向刚创建的任务，开始运行。

4）stkSz 可取的最大值为 0xfff，timeSlice 可取的最大值为 0xfff。

6.2.2 CoExitTask()

函数原型：

void CoExitTask(void);

功能描述：

退出当前正在运行的任务。

函数参数：

无

返回值：

无

示例：

```
#include "CCRTOS.h"
void TaskN (void *pdata)
{
    ...
  CoExitTask();          // Exit current task
}
```

备注：

1）任务退出后，若系统处于动态任务调度模式下，将回收分配此任务资源，以便重新分配。若处于静态任务调度模式，则保留任务资源，等待被重新激活。

2）任务的退出将调用任务调度器，使下一个任务获得系统运行时间。

3）CooCox CoOS 中，任务的退出只能在本任务体内调用。

6.2.3 CoDelTask()

函数原型：

```
StatusType CoDelTask(
                          OS_TID task ID
                        );
```

功能描述：

删除任务。

函数参数：

[in] taskID

需要被删除的任务 ID

返回值：

E_INVALID_ID， 无效任务 ID

E_PROTECTED_TASK， 受系统保护任务，无法删除

E_OK， 删除成功

示例：

```
#include "CCRTOS.h"
OS_TID TaskMId;
void TaskM (void *pdata);
void TaskN (void *pdata)
{
    StatusType result;
    ...
    result = CoDelTask(TaskMId);
    if (result != E_OK)
    {
        if (result==E_INVALID_ID)
        {
            printf("Invalid task ID !\n");
        }
        else if (result==E_PROTECTED_TASK)
        {
            printf("Protected task in OS cannot be deleted !\n");
        }
    }

    ...
}
void TaskM (void *pdata)
{
    ...
}
```

备注：

1）删除任务，若系统处于动态任务调度模式下，将回收分配此任务资源，以便重新分配。若处于静态任务调度模式，则保留任务资源，等待被重新激活。

6.2.4 CoGetCurTaskID()

函数原型：

CoGetCurTaskID(void);

功能描述：

获得当前任务 ID

函数参数：

无

返回值：

当前任务 ID

示例：

```
#include "CCRTOS.h"
void TaskN (void *pdata)
{
    OS_TID tid;
    ...
    tid = CoGetCurTaskID();
    printf("Current task ID is %d !\n",tid);
    ...
}
```

备注：

无

6.2.5 CoSetPriority()

函数原型：

```
StatusType  CoSetPriority(
                          OS_TID    taskID,
                          U8        priority
                         );
```

功能描述：

设定指定任务优先级。

函数参数：

[in] task ID

指定任务ID

[in] Priority

需要重新设定的优先等级

返回值：

E_INVALID_ID， 传入的任务ID为无效ID

E_OK， 设定成功

示例：

```
#include "CCRTOS"
#define NEWPRIO      10
void TaskN (void *pdata)
{
     ...
  SetPriority (TaskMId, NEWPRIO);
     ...
}
void TaskM (void *pdata)
{
     ...
}
```

备注：

1）若设定的任务在等待列表中，且其重新设定后的优先级高于当前运行的任务优先级，则将运行任务调度。

6.2.6 CoSuspendTask()

函数原型：

```
StatusType CoSuspendTask(
                          OS_TID task ID
                         );
```

功能描述：

挂起指定任务。

函数参数：

[in] taskID

指定任务 ID

返回值：

E_INVALID_ID，	传入的任务ID为无效ID
E_PROTECTED_TASK，	不能挂起空闲任务
E_ALREADY_IN_WAITING，	任务已处于等待状态
E_OK，	挂起任务成功

示例：

```
#include "CCRTOS.h"
void TaskN (void *pdata)
{
  StatusType sta;
  sta = CoSuspendTask (TaskMId);  // Suspend TaskM
  if (sta != E_OK)
  {
     if (sta==E_INVALID_ID)
     {
          printf ("TaskM does not exist !\n");
     }
     else if (sta==E_ALREADY_IN_WAITING)
     {
          printf ("TaskM is not ready !\n");
     }
  }
}
void TaskM (void *pdata)
{
     ...
}
```

备注：

1）挂起的任务转至等待（TASK_WAITING）态。

2）调用该函数将运行一次任务调度。

3）与 CoAwakeTask(taskID)成对使用。

6.2.7 CoAwakeTask()

函数原型：

```
StatusType CoAwakeTask(
                          OS_TID task ID
                          );
```

功能描述：

唤醒指定任务。

函数参数：

[in] taskID

指定任务 ID

返回值：

E_INVALID_ID，	传入的任务ID为无效ID；
E_TASK_NOT_WAITING，	指定任务处于非等待状态；
E_TASK_WAIT_OTHER，	该任务正在在等待其它唤醒事件；

E_PROTECTED_TASK，　　空闲任务必须被唤醒；
E_OK，　　唤醒任务成功；

示例：

```
#include "CCRTOS.h"
void TaskI (void *pdata)
{
    ...
  CoSuspendTask (TaskMId);          // Suspend TaskM
    ...
}
void TaskN (void *pdata)
{
  StatusType sta;
    ...
  sta = CoAwakeTask (TaskMId);     // Wakeup TaskM
  if(sta==E_OK) printf("TaskM is ready !\n");
    ...
}
void TaskM (void *pdata)
{
    ...
}
```

备注：

1）若该任务还在等待其它事件，则仍然停留在等待（TASK_WAITING）态。否则返回至就绪（TASK_READY）态。
2）调用该函数将运行一次任务调度。
3）与CoSuspendTask(taskID)成对使用。

6.2.8 CoActivateTask()

函数原型：

```
StatusType CoActivateTask(
                          OS_TID task ID,
                          void *argv
                          );
```

功能描述：

在静态任务调度模式下，激活已退出任务。

函数参数：

[in] taskID
　　指定任务 ID
[in] *argv
　　任务执行函数体的输入参数

返回值：

E_INVALID_ID，　　传入的任务ID为无效ID；

E_OK，　　激活任务成功；

示例：

```
#include "CCRTOS.h"
void TaskI (void *pdata)
{
    ...
  CoDelTask(TaskMId);            // Delete TaskM
    ...
}
void TaskN (void *pdata)
{
  StatusType sta;
    ...
  sta = CoActivateTask(TaskMId,NULL);    // Activate TaskM
  if(sta==E_OK) printf("TaskM is ready !\n");
    ...
}
void TaskM (void *pdata)
{
    ...
}
```

备注：

无

6.3 时间管理

6.3.1 CoGetOSTime()

函数原型：

U64 CoGetOSTime(void);

功能描述：

获取当前系统运行的时间。

函数参数：

无

返回值：

当前系统节拍值

示例：

```
#include "CCRTOS.h"
void TaskN (void *pdata)
{
  U64 ostime;
    ...
  ostime = CoGetOSTime();
    ...
}
```

备注：

无

6.3.2 CoTickDelay()

函数原型：

StatusType CoTickDelay(
U32 ticks
);

功能描述：

延时指定节拍数。

函数参数：

*[in]*ticks

要延迟的系统节拍数

返回值：

E_CALL	在中断服务程序里调用
E_OK	正确执行

示例：

```
#include "CCRTOS.h"
void TaskN (void *pdata)
{
  StatusType sta;
     ...
  sta = CoTickDelay(15);          // Delay 15 system ticks
  if(sta!=E_OK)
  {
     if(sta==E_CALL)
     {
            printf("TickDelay cannot been used in ISR !\n");
     }
  }
     ...
}
```

备注：

1）调用TickDelay()，CooCox CoOS会将当前的任务从运行态（TASK_RUNNING）转换成等待态（TASK_WAITING），并将其插入至delay链表，延时指定系统节拍数数。

6.3.3 CoResetTaskDelayTick()

函数原型：

```
StatusType CoResetTaskDelayTick(
                                    OS_TID taskID,
                                    U32 ticks
                                    );
```

功能描述：

重新设置指定任务的延时节拍。

函数参数：

*[in]*taskID，
　　指定任务ID
*[in]*ticks，
　　重新设定的延时节拍

返回值：

E_INVALID_ID，	无效 ID
E_NOT_IN_DELAY_LIST，	指定任务不在延时链表
E_OK，	正确执行

示例：

```
#include "CCRTOS.h"
OS_TID TaskMId;
OS_TID TaskNId;
void TaskM (void *pdata)
{
    ...
    CoTickDelay (30);
    ...
}
void TaskN (void *pdata)
{
    StatusType result;
    ...
    /* Reset TaskM delay time */
    result = CoResetTaskDelayTick (TaskMId, 61);
    if(result!=E_OK)
    {
        if(result == E_INVALID_ID)
        {
            printf("Invalid task id !\n");
        }
        else if (result == E_NOT_IN_DELAY_LIST)
        {
            printf("TaskM is not in delay list !\n");
        }
    }
    ...
}
```

备注：

1）若重新设置的 tick 数为 0，则将指定任务移出延时链表。

6.3.4 CoTimeDelay()

函数原型：

```
StatusType CoTimeDelay(
                        U8 hour,
                        U8 minute,
                        U8 sec,
                        U16 millsec
                        );
```

功能描述：

延时指定时间。

函数参数：

*[in]*hour

要延迟的小时数；

*[in]*minute

要延迟的分钟数；

*[in]*sec

要延迟的秒数；

*[in]*millsec

要延迟的系统毫秒数；

返回值：

E_CALL，	在中断服务程序里调用；
E_INVALID_PARAMETER，	无效参数；
E_OK，	正确执行；

示例：

```
#include "CCRTOS.h"
void TaskN (void *pdata)
{
  StatusType sta;
     …
  sta = CoTimeDelay(0,0,1,0);              // Delay 1 second
  if(sta!=E_OK)
  {
     if(sta==E_CALL)
     {
          printf("TickDelay cannot been used in ISR !\n");
     }
     else if (sta == E_INVALID_PARAMETER)
     {
          printf("Invalid parameter !\n");
     }
  }
     …
}
```

备注：

1）若输入的时间值不符合标准，则返回错误。

6.4 软件定时器

6.4.1 CoCreateTmr()

函数原型：

```
OS_TCID     CoCreateTmr(
                        U8 tmrType,
                        U32 tmrCnt,
                        U32 tmrReload,
                        vFUNCPtr func
                        );
```

功能描述：

创建定时器。

函数参数：

[IN] tmrtype

定时器类型；

[IN] tmrCnt

第一次计数值（系统节拍数）；

[IN] tmrReload

定时器重新载入值；

[IN] func

回调函数。

返回值：

定时 ID，　　创建成功

-1，　　创建失败

示例：

```
#include "CCRTOS.h"
OS_TCID sftmr;
void SftTmrCallBack(void)
{
    …
}
void TaskN (void *pdata)
{
  StatusType sta;
    …
  sftmr = CoCreateTmr(TMR_TYPE_PERIODIC,
                      100,
                      100,
                      SftTmrCallBack);
  if (sftmr=E_CREATE_FAIL)
  {
    printf("Failed to create the timer!\n");
  }
  else
  {
    printf("Create the timer successfully, Time ID %d\n", sftmr);
  }
    …
}
```

备注：

1）CooCox CoOS 提供两种类型的定时器：周期定时器和一次性定时器。周期性定时器第一次调用回调函数的时间由传入参数 tmrCnt 决定，随后的回调时间由传入参数 tmrReload 决定。而一次性定时器的调用时间只由 tmrCnt 决定，且只调用一次，tmrReload 值无任何作用。

2）一个定时器被创建后， 默认处于停止状态，需要调用 CoStartTmr()启动该定时器方可正常工作。

6.4.2 CoStartTmr()

函数原型：

```
StatusType CoStartTmr(
                      OS_TCID tmrID
                      );
```

功能描述：

启动指定定时器，使其开始正常工作。

函数参数：

[IN] tmrID

指定定时器的ID

返回值：

E_INVALID_ID，　传入的定时器ID为无效ID；

E_OK，　成功启动指定定时器。

示例：

```
#include "CCRTOS.h"
OS_TCID sftmr;
void TaskN (void *pdata)
{
  StatusType sta;
     …
  /* Create Software Timer */
  sftmr = CoCreateTmr(TMR_TYPE_PERIODIC,
                      100,
                      100,
                       SftTmrCallBack);
     …
  /* Start Software Timer */
  sta = CoStartTmr (sftmr);
  if (sta != E_OK)
  {
     if (sta == E_INVALID_ID)
     {
          printf("The timer id passed is invalid, can't start the timer. \n");
     }
  }
     …
}
```

备注：

无

6.4.3 CoStopTmr()

函数原型：

```
StatusType CoStopTmr(
                        OS_TCID tmrID
                        );
```

功能描述：

停止指定定时器，使其停止工作。

函数参数：

[IN] tmrID

指定定时器的ID

返回值：

E_INVALID_ID，　传入的定时器ID为无效ID；
E_OK，　　成功停止指定定时器。

示例：

```
#include "CCRTOS.h"
OS_TCID sftmr;
void TaskN (void *pdata)
{
  StatusType sta;
     …
  /* Stop Software Timer */
  sta = CoStopTmr (sftmr);
  if (sta != E_OK)
  {
     if (sta == E_INVALID_ID)
    {
          printf("The timer id passed is invalid, failed to stop. \n");
     }
  }
     …
}
```

备注：

1）一个软件定时器停止运行后，系统会保留其当前计数值，以便再次激活。

6.4.4 CoDelTmr()

函数原型：

```
StatusType CoDelTmr(
                        OS_TCID tmrID
                      );
```

功能描述：

删除指定定时器，并释放其占用的资源。

函数参数：

[IN] tmrID
　　指定定时器的ID

返回值：

E_INVALID_ID，　传入的定时器ID为无效ID；
E_OK，　　成功删除指定定时器。

示例：

```
#include "CCRTOS.h"
OS_TCID sftmr;
void TaskN (void *pdata)
{
  StatusType sta;
     …
  /* Create Software Timer */
  sftmr = CoCreateTmr(TMR_TYPE_PERIODIC,
                      100,
                      100,
                      SftTmrCallBack);
     …
  /* Delete Software Timer */
  sta = CoDelTmr (sftmr);
  if (sta != E_OK)
  {
     if (sta == E_INVALID_ID)
     {
          printf("The timer id passed is invalid, filed to delete!\n");
     }
  }
     …
}
```

备注：

无

6.4.5 CoGetCurTmrCnt()

函数原型：

```
U32  CoGetCurTmrCnt(
                    OS_TCID        tmrID,
                    StatusType*    perr
                   );
```

功能描述：

获得指定定时器的当前计数值。

函数参数：

[IN] tmrID
指定定时器的ID

[OUT] Perr
返回的错误类型
E_INVALID_ID，　传入的定时器ID为无效ID；
E_OK，　　成功获得指定定时器的计数值。

返回值：

指定软件定时器的当前计数值

示例：

```
#include "CCRTOS.h"
OS_TCID sftmr;
void TaskN (void *pdata)
{
  StatusType sta;
  U32    sftcnt;
     …
  sftcnt = CoGetCurTimerCnt (sftmr, &sta);
  if (sta != E_OK)
  {
     if (sta == E_INVALID_ID)
     {
          printf("The timer id passed is invalid, failed to stop. \n");
     }
  }
  else
  {
     printf("Current Timer Counter : %ld", sftcnt);
  }
     …
}
```

备注：

无

6.4.6 CoSetTmrCnt()

函数原型：

```
StatusType CoSetTmrCnt(
                        OS_TCID tmrID,
                        U32 tmrCnt,
                        U32 tmrReload
                        );
```

功能描述：

设置指定计数器的计数值和定时器导入值。

函数参数：

[IN] tmrID

指定定时器的ID；

[IN] tmrCnt

重新设定的计数值；

[IN] tmrReload

重新设定的定时器导入值。

返回值：

E_INVALID_ID，　传入的定时器ID为无效ID；
E_OK，　成功获得指定定时器的计数值。

示例：

```
#include "CCRTOS.h"
OS_TCID sftmr;
void TaskN (void *pdata)
{
  StatusType sta;
     …
  sta = CoSetTimerCnt (sftmr, 200, 200);
  if (sta != E_OK)
  {
     if (sta == E_INVALID_ID)
     {
          printf("The timer id passed is invalid, failed to stop. \n");
     }
  }
     …
}
```

备注：

无

6.5 内存管理

6.5.1 CoKmalloc()

函数原型：

```
void*  CoKmalloc(
                    U32 size
                    );
```

功能描述：

动态分配长度为 size 的一段内存区域。

函数参数：

[IN] size

需要分配的内存块大小，单位为字节

返回值：

NULL　　分配失败

Others　分配成功，返回内存块首地址指针

示例：

```
#include "CCRTOS.h"
void TaskN (void *pdata)
{
    unsigned int *ptr;
    ...
    /* Allocation 20 word of memory block from kernel heap */
    ptr = (unsigned int *)CoKmalloc(20*4);

    /* process ptr here */
    ...
    /* Release memory block to kernel heap */
    CoKfree(ptr);
    ...
}
```

备注：

1）每调用一次 CoKmalloc（），都将在 size 长度的基础上多消耗 8bytes，来管理该内存块。

6.5.2 CoKfree()

函数原型：

```
void  CoKfree(
                void* memBuf
            );
```

功能描述：

释放首地址为 memBuf 的内存区域

函数参数：

[IN] memBuf

需要释放的内存块首地址

返回值：

无

示例：

```
#include "CCRTOS.h"
void TaskN (void *pdata)
{
    unsigned int *ptr;
    ...
    /* Allocation 20 word of memory block from kernel heap */
    ptr = (unsigned int *)CoKmalloc(20*4);

    /* process ptr here */
    ...
    /* Release memory block to kernel heap */
    CoKfree(ptr);
    ...
}
```

备注：

1）memBuf 必须是通过 CoKmalloc（）所获得的地址，否则将不处理，立即返回。

6.5.3 CoCreateMemPartition()

函数原型：

```
OS_MMID CoCreateMemPartition(
                                    U8* memBuf,
                                    U32 blockSize,
                                    U32 blockNum
                                    );
```

功能描述：

创建一个固定长度分区。

函数参数：

[IN] memBuf

固定长度分区的首地址

[IN] blockSize

固定长度分区的每一个内存块的大小

[IN] blockNum

固定长度分区的内存块个数

返回值：

内存分区ID， 创建分区成功，返回分区ID
-1， 创建分区失败

示例：

```
#include "CCRTOS.h"
#define MEM_BLOCK_NUM   10
OS_MMID mmc;
unsigned int MemoryBlock[100];
void TaskN (void *pdata)
{
    ...
    /* Create a memory partition          */
    /* Memory size: 100*4 bytes,          */
    /* Block num:    10                   */
    /* Block size    100*4/10 = 40 bytes */
    mmc=CoCreateMemPartition((U8*)MemoryBlock,
                             100*4/MEM_BLOCK_NUM,
                             MEM_BLOCK_NUM);
    if (mmc == E_CREATE_FAIL)
    {
        printf("Create memory partition fail !\n");
    }
    else
    {
        printf("Memory Partition ID : %d \n", mmc);
    }
    ...
}
```

备注：

1）成功创建一个内存分区，系统将分配一个内存控制块来实现对该内存块的管理。

2） blockSize 不能为 0，blockNum 必须大于 2。

6.5.4 CoDelMemoryPartition()

函数原型：

```
StatusType CoDelMemoryPartition(
                                OS_MMID mmID
                                );
```

功能描述：

删除指定的固定长度分区

函数参数：

[IN] mmID
内存分区 ID

返回值：

E_INVALID_ID　　无效内存分区 ID
E_OK　　删除内存分区成功

示例：

```
#include "CCRTOS.h"
OS_MMID mmc;
void TaskN (void *pdata)
{
    StatusType result;
    ...
    /* Delete a memory partition   */
    /* mmc: Created by other task */
    result = CoDelMemoryPartition(mmc);
    if (result != E_OK)
    {
        if (result == E_INVALID_ID)
        {
            printf("Invalid memory partition !\n");
        }
    }
    ...
}
```

备注：

1）成功删除内存分区，将释放该内存分区所占用的内存控制块资源。

2）当你想删除一个内存分区的时候，CoOS 不会自动检查该分区是否空闲，你应该在自己的 APP 中管理。

6.5.5 CoGetMemoryBuffer()

函数原型：

```
void*  CoGetMemoryBuffer(
                              OS_MMID mmID
                              );
```

功能描述：

从指定固定长度分区中获得一个内存块

函数参数：

[IN] mmID
　　内存分区 ID

返回值：

NULL　　分配失败
Others　　分配成功，返回内存块首地址指针

示例：

```
#include "CCRTOS.h"
OS_MMID mmc;
void TaskN (void *pdata)
{
    int *ptr;
    ...
    /* Get a memory block from memory partition */
    /* mmc: Created by other task */
    ptr = (int* )CoGetMemoryBuffer(mmc);
    if (ptr == NULL)
    {
        printf("Assign buffer fail !\n");
    }
    else
    {
        ...
        /* Process assigned buffer here */
        ...
        /* Free assigned buffer to memory partition */
        CoFreeMemoryBuffer(mmc, (void* )ptr);
    }
    ...
}
```

备注：

无

6.5.6 CoFreeMemoryBuffer()

函数原型：

```
StatusType  CoFreeMemoryBuffer(
                                    OS_MMID mmID,
                                    void* buf
                                    );
```

功能描述：

释放首地址为 buf 的内存块至指定的固定长度分区。

函数参数：

[IN] mmID

内存分区 ID

[IN] buf

需要释放的内存块首地址

返回值：

E_INVALID_ID	无效内存分区 ID

E_INVALID_PARAMETER　　无效 buf 参数，即所释放的地址并非有效地址
E_OK　　释放内存块成功

示例：

```
#include "CCRTOS.h"
OS_MMID mmc;
void TaskN (void *pdata)
{
    int *ptr;
    ...
    /* Get a memory block from memory partition */
    /* mmc: Created by other task */
    ptr = (int* )CoGetMemoryBuffer(mmc);
    if (ptr == NULL)
    {
        printf("Assign buffer fail !\n");
    }
    else
    {
        ...
        /* Process assigned buffer here */
        ...
        /* Free assigned buffer to memory partition */
        CoFreeMemoryBuffer(mmc, (void* )ptr);
    }
    ...
}
```

备注：

1）buf 必需是通过 CoGetMemoryBuffer（）所获得的地址，否则将出错返回。

6.5.7 CoGetFreeBlockNum()

函数原型：

U32 CoGetFreeBlockNum(OS_MMID mmID,StatusType* perr)

功能描述：

获得指定内存分区空闲块的个数。

函数参数：

[IN]　mmID
信号量 ID

[OUT] perr
错误返回值：

E_INVALID_ID　　无效的内存分区 ID
E_OK　　成功获得当前内存分区空闲块个数

返回值：

fbNum，　　内存分区空闲块个数

示例：

```
#include "CCRTOS.h"
OS_MMID mmc;
void TaskN (void *pdata)
{
    U32 blocknum;
    StatusType result;
    ...
    /* Get free blocks number */
    /* mmc: Created by other task */
    blocknum = CoGetFreeBlockNum(mmc, &result);
    if (result != E_OK)
    {
        if (result == E_INVALID_ID) {
            printf("Invalid ID !\n");
        }
    }
    ...
}
```

备注：

无

6.6 互斥区域

6.6.1 CoCreateMutex()

函数原型：

OS_MutexID CoCreateMutex(void);

功能描述：

创建一个互斥区域。

函数参数：

无

返回值：

互斥区域ID， 创建成功
-1， 创建失败

示例：

```
#include "CCRTOS.h"
OS_MutexID mutex;
void TaskN (void *pdata)
{
      ...
   /* Create a mutex */
  mutex = CoCreateMutex ();
  if (mutex == E_CREATE_FAIL)
  {
     printf("Create mutex fail. \n");
  }
  else
  {
    printf("Mutex ID : %d \n", mutex);
  }
     ...
}
```

备注：

无

6.6.2 CoEnterMutexSection()

函数原型：

StatusType CoEnterMutexSection(
OS_MutexID mutexID
);

功能描述：

进入指定ID的互斥区域。

函数参数：

*[in]*muterID

指定互斥区域ID

返回值：

E_CALL，　　在中断服务程序里调用

E_INVALID_ID，　　传入的互斥区域 ID为无效ID

E_OK，　　进入互斥区域成功

示例：

```
#include "CCRTOS.h"
OS_MutexID mutex;
void TaskN (void *pdata)
{
      ...
    /* Create a mutex */
  mutex = CoCreateMutex ();
    /* Enter critical region */
  CoEnterMutexSection(mutex);
      ...
    /*Excute critical code */
      ...
    /* Exit critical region */
  CoLeaveMutexSection(mutex);
    ...
}
```

备注：

1）若高优先级的任务 A 尝试进入互斥区域，将提升当前已进入互斥区域的任务 B 的优先级至与任务 A 同一级别，并将任务 A 转至等待（TASK_WAITING）态，进行一次任务调度，使已进入互斥区域的任务 B 尽快离开互斥区域。任务 B 在离开互斥区域后，优先级恢复至原先优先级。

2）此函数不能在中断服务程序内使用。

3）与 CoLeaveMutexSection()配对使用。

6.6.3 CoLeaveMutexSection()

函数原型：

```
StatusType  CoLeaveMutexSection(
                                OS_MutexID mutexID
                                );
```

功能描述：

离开指定ID的互斥区域。

函数参数：

*[in]*muterID，指定互斥区域ID

返回值：

E_CALL，	在中断服务程序里调用
E_INVALID_ID，	传入的互斥区域ID为无效ID
E_OK，	离开互斥区域成功

示例：

```
#include "CCRTOS.h"
OS_MutexID mutex;
void TaskN (void *pdata)
{
       ...
     /* Create a mutex */
   mutex = CoCreateMutex ();
     /* Enter critical region */
   CoEnterMutexSection(mutex);
      ...
     /* Excute critical code */
      ...
     /* Exit critical region */
   CoLeaveMutexSection(mutex);
      ...
}
```

备注：

1）若当前有任务在等待进入该互斥区域，在退出互斥区域时，需要进行一次任务调度，并将当前任务的优先级恢复至原有状态。
2）此函数不能在中断服务程序内使用。
3）与 CoEnterMutexSection()配对使用。

6.7 信号量

6.7.1 CoCreateSem()

函数原型：

```
OS_EventID  CoCreateSem(
                        U16 initCnt,
                        U16 maxCnt,
                        U8 sortType
                        );
```

功能描述：

创建一个信号量。

函数参数：

[in] initCnt

初始有效信号量数

[in] maxCnt

信号量的最大值

[in] sortType

排列类型：

EVENT_SORT_TYPE_FIFO， 先入先出顺序

EVENT_SORT_TYPE_PRIO， 优先级抢占顺序

返回值：

信号量ID， 创建成功

-1， 创建失败

示例：

```
#include "CCRTOS.h"
OS_EventID semaphore;
void TaskN (void *pdata)
{
      …
    /* Create a semaphore */
  semaphore = CoCreateSem (1, 10, EVENT_SORT_TYPE_FIFO);
  if (semaphore == E_CREATE_FAIL)
  {
    printf("Create semaphore failed !\n");
  }
  else
  {
     printf("Semaphore ID : %d \n", semaphore);
  }
     …
}
```

备注：

无

6.7.2 CoDelSem()

函数原型：

```
StatusType CoDelSem(
                        OS_EventID   id,
                        U8           opt
                       );
```

功能描述：

删除指定ID信号量。

函数参数：

[in] id

指定信号量ID

[in] opt

删除指定信号量的方式：

EVENT_DEL_NO_PEND,	等待列表为空时删除
EVENT_DEL_ANYWAY,	无条件删除

返回值：

E_INVALID_ID,	传入的信号量ID为无效ID；
E_INVALID_PARAMETER,	无效参数，即ID对应的控制块为空；
E_TASK_WAITING,	等待列表非空；
E_OK,	删除成功；

示例：

```
#include "CCRTOS.h"
OS_EventID semaphore;
void TaskN (void *pdata)
{
  StatusType result;
      ...
     /* Create a semaphore */
  result = CoDeleteSem (semaphore, OPT_DEL_ANYWAY);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
     {
        printf("Invalid event ID !\n");
      }
  }
     ...
}
```

备注：

1) 成功删除则释放该信号量所占用的资源，并进行任务调度，将系统执行时间让给优先级最高的任务。

6.7.3 CoAcceptSem()

函数原型：

```
StatusType  CoAcceptSem(
                        OS_EventID  id
                        );
```

功能描述：

获得一个指定ID信号量资源。

函数参数：

[in] id

指定信号量ID

返回值：

E_INVALID_ID，	传入的信号量ID为无效ID；
E_SEM_EMPTY，	指定ID的信号量资源为空；
E_OK，	成功获得资源；

示例：

```
#include "CCRTOS.h"
OS_EventID semaphore;;
void TaskN (void *pdata)
{
  StatusType result;
      ...
     /* Accept a semaphore without waitting */
  result = CoAcceptSem (semaphore);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
     {
          printf("Invalid event ID !\n");
      }
      else if (result == E_SEM_EMPTY)
     {
          printf("No semaphore exist !\n");
      }
  }
  else
  {
       ...
       /* Process semaphore here */
       ...
  }
     ...
}
```

备注：

1）一次AcceptSem()函数的运行是一次不完整的P操作。它不同于P操作的地方是，当资源数为0时，完整的P操作是一直等待其它任务释放相关资源或超时结束，而此函数中则立刻错误返回。

2）成功获得资源后，该信号量资源数减1。

6.7.4 CoPendSem()

函数原型：

```
StatusType  CoPendSem(
                        OS_EventID  id,
                        U32         timeout
                        );
```

功能描述：

等待信号量。

函数参数：

[in] id

指定信号量ID

[in] timeout

超时时间，0表示无限期等待

返回值：

E_CALL，	在中断服务程序内调用；
E_INVALID_ID，	传入的信号量ID为无效ID；
E_TIMEOUT，	等待资源超时；
E_OK，	成功获得资源；

示例：

```
#include "CCRTOS.h"
OS_EventID semaphore;;
void TaskN (void *pdata)
{
  StatusType result;
      ...
     /* Pend a semaphore, Time out: 20 */
  result = CoPendSem (semaphore, 20);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
      {
          printf("Invalid event ID !\n");
      }
      else if (result == E_CALL)
      {
          printf("Error call in ISR !\n");
      }
      else if (result == E_TIMEOUT)
      {
          printf("Semaphore was not received within the specified 'timeout' time !\n");
      }
  }
  else
  {
       ...
       /* Process semaphore here */
       ...
  }
     ...
}
```

备注：

1）该函数在等待过程中执行一次任务调度，将系统执行时间让给优先级最高的任务。

2）成功获得资源后，该信号量资源数减 1。

3）此函数不能在中断服务程序内使用。

6.7.5 CoPostSem()

函数原型：

```
StatusType CoPostSem(
                        OS_EventID   id
                        );
```

功能描述：

释放一个指定ID信号量资源。

函数参数：

[in] id

指定信号量ID

返回值：

E_INVALID_ID，	传入的信号量ID为无效ID
E_SEM_FULL，	指定信号量已达最大值
E_OK，	成功释放一个信号量资源

示例：

```
#include "CCRTOS.h"
OS_EventID semaphore;
void TaskN (void *pdata)
{
  StatusType result;
      ...
     /* Post a semaphore */
  result = CoPostSem (semaphore);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
      {
          printf("Invalid event ID !\n");
      }
      else if (result == E_NO_SEM_PEND)
      {
          printf("There are no task waitting for the event !\n");
      }
  }
     ...
}
```

备注：

1）此函数不能在中断服务程序内使用。

2）与CoAcceptSem()或 CoPendSem()配对使用。

6.7.6 isr_PostSem()

函数原型：

StatusType isr_PostSem(OS_EventID id)

功能描述：

在中断服务程序中，释放一个指定 ID 信号量资源。

函数参数：

[IN] id

信号量 ID

返回值：

E_SEV_REQ_FULL，	中断服务请求已满
E_INVALID_ID，	无效的信号量 ID
E_SEM_FULL，	指定信号量已达最大值
E_OK，	成功释放信号量资源

示例：

```
#include "CCRTOS.h"
OS_EventID semaphore;
void XXX_IRQHandler(void)
{
     StatusType result;
     EnterISR();          // Enter ISR
          …
     /* Post a semaphore */
     result = isr_PostSem (seamaphore);
     if (result != E_OK) {
          if (result == E_SEV_REQ_FULL) {
               printf("Service requst queue is full !\n");
          }
     }
          …
     ExitISR();        // Exit ISR
}
```

备注：

1）不能在中断服务程序中调用 PostSem（）来实现信号量资源的释放，否则将导致系统混乱。

6.8 邮箱

6.8.1 CoCreateMbox()

函数原型：

```
OS_EventID CoCreateMbox(
                                U8 sortType
                                );
```

功能描述：

创建一个邮箱。

函数参数：

[in] sortType，等待列表排列方式：

EVENT_SORT_TYPE_FIFO，FIFO方式

EVENT_SORT_TYPE_PRIO，优先级抢占方式

返回值：

邮箱ID，	创建成功
-1，	创建失败

示例：

```
#include "CCRTOS.h"
OS_EventID mailbox;;
void TaskN (void *pdata)
{
      ...
     /* Create a mailbox */
  mailbox = CoCreateMbox (EVENT_SORT_TYPE_FIFO);
  if (mailbox != E_OK)
  {
      if (mailbox == E_CREATE_FAIL)
      {
          printf("Create mailbox failed !\n");
      }
  }
  else
  {
      printf("MailBox ID : %d \n", mailbox);
  }
     ...
}
```

备注：

无

6.8.2 CoDelMbox()

函数原型：

```
StatusType CoDelMbox(
                        OS_EventID   id,
                        U8           opt
                        );
```

功能描述：

删除指定ID邮箱。

函数参数：

[in] id，指定邮箱ID；

[in] opt，删除指定邮箱的方式：

EVENT_DEL_NO_PEND， 等待列表为空时删除
EVENT_DEL_ANYWAY， 无条件删除

返回值：

E_INVALID_ID，	传入的邮箱ID为无效ID
E_INVALID_PARAMETER，	无效参数，即ID对应的控制块为空
E_TASK_WAITING，	等待列表非空
E_OK，	删除成功

示例：

```
#include "CCRTOS.h"
OS_EventID mailbox;;
void TaskN (void *pdata)
{
  StatusType result;
      ...
     /* Create a mailbox */
  result = CoDelMbox (mailbox, OPT_DEL_ANYWAY);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
     {
          printf("Invalid event ID !\n");
      }
  }
     ...
}
```

备注：

1）成功删除则释放该邮箱占用的资源，并进行任务调度，将系统执行时间让给优先级最高的任务。

6.8.3 CoAcceptMail()

函数原型：

```
void*  CoAcceptMail(
                    OS_EventID   id,
                    StatusType*   perr
                    );
```

功能描述：

获得一个指定ID邮箱消息。

函数参数：

[in] id

指定邮箱ID；

[in] perr

错误返回类型：

E_INVALID_ID,	传入的邮箱ID为无效ID
E_MBOX_EMPTY,	指定ID的邮箱信息为空
E_OK,	成功获得消息

返回值：

获得邮箱消息指针。

示例：

```
#include "CCRTOS.h"
OS_EventID mailbox;
void TaskN (void *pdata)
{
  StatusType result;
  void *msg;
      ...
     /* Create a mailbox */
  msg = CoAcceptMail (mailbox, &result);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
     {
          printf("Invalid event ID !\n");
      }
  }
  else
  {
          ...
          /* Process mail here */
          ...
  }
     ...
}
```

备注：

1）调用AcceptSem()函数时，若指定邮箱内的消息指针非空，则该函数获得该消息指针，并清空邮箱；否则立刻错误返回。

6.8.4 CoPendMail()

函数原型：

```
void*   CoPendMail(
                    OS_EventID    id,
                    U32           timeout,
                    StatusType*    perr
                    );
```

功能描述：

等待指定邮箱内的消息。

函数参数：

[in] id

指定邮箱ID

[in] Timeout

超时时间，0表示无限期等待

[out] perr，错误返回类型：

E_CALL，	在中断服务程序里调用
E_INVALID_ID，	传入的信号量 ID 为无效 ID
E_TIMEOUT，	等待资源超时
E_OK，	成功获得消息

返回值：

获得邮箱消息指针

示例：

```
#include "CCRTOS.h"
OS_EventID mailbox;
void TaskN (void *pdata)
{
  StatusType result;
  void *msg;
      ...
     /* Create a mailbox */
  msg = CoPendMail (mailbox, 20, &result);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
     {
          printf("Invalid event ID !\n");
      }
  }
  else
  {
      printf("Recived %d \n", *(unsigned int *)msg);
         /* Process mail here */
  }
     ...
}
void TaskM (void *pdata)
{
  StatusType result;
  unsigned int pridat;
     ...
  pridat = 0x49;
     ...
  result = CoPostMail (mailbox, &pridat);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
     {
          printf("Invalid event ID !\n");
      }
  }
     ....
}
```

备注：

1）该函数在等待过程中执行一次任务调度，将系统执行时间让给优先级最高的任务。

2）成功获得邮箱消息后，会清空邮箱。

3）该函数不能在中断服务程序内使用。

6.8.5 CoPostMail()

函数原型：

```
StatusType  CoPostMail(
                        OS_EventID  id,
                        void*       pmail
                        );
```

功能描述：

填充消息至 一个指定 ID 邮箱。

函数参数：

[in] id

指定邮箱ID

[in] pmail

填充的消息指针

返回值：

E_INVALID_ID，	传入的邮箱ID为无效ID
E_MBOX_FULL，	邮箱已满；
E_OK，	成功填充消息至邮箱；

示例：

```
#include "CCRTOS.h"
OS_EventID mailbox;
void TaskN (void *pdata)
{
  StatusType result;
  void *msg;
      ...
     /* Create a mailbox */
  msg = CoPendMail (mailbox, 20, &result);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
     {
          printf("Invalid event ID !\n");
      }
  }
  else
  {
     printf("Recived %d \n", *(unsigned int *)msg);
  }
     ...
}
void TaskM (void *pdata)
{
  StatusType result;
  unsigned int pridat;
      ...
  pridat = 0x49;
      ...
  result = CoPostMail (mailbox, &pridat);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
     {
          printf("Invalid event ID !\n");
      }
  }
      ....
}
```

备注：

1）该函数不能用于中断服务程序中。

2）与 AcceptMail()或PendMail()配对使用。

6.8.6 isr_PostMail()

函数原型：

StatusType isr_PostMail(OS_EventID id,void* pmail)

功能描述：

在中断服务程序中，发送消息至一个指定 ID 的邮箱。

函数参数：

[IN] id

邮箱 ID

[IN] pmail

消息指针

返回值：

E_SEV_REQ_FULL， 中断服务请求已满

E_INVALID_ID， 无效的邮箱 ID

E_MBOX_FULL， 邮箱已满

E_OK， 成功发送消息至邮箱

示例：

```
#include "CCRTOS.h"
OS_EventID mailbox;
int IsrDat;
void XXX_IRQHandler(void)
{
    StatusType result;
    EnterISR();          // Enter ISR
        ...
    IsrDat = 0x90;
    /* Post a mail to Mailbox that created by other tase */
    result = isr_PostMail (mailbox, &IsrDat);
    if (result != E_OK) {
        if (result == E_SEV_REQ_FULL) {
            printf("Service requst queue is full !\n");
        }
    }
        ...
    ExitISR();        // Exit ISR
}
```

备注：

1）用于中断服务程序。

2）不能在中断服务程序中调用 CoPostMail（）来实现邮箱消息的发送，否则将导致系统混乱。

6.9 消息队列

6.9.1 CoCreateQueue()

函数原型：

```
OS_EventID CoCreateQueue(
                          void**   qStart,
                          U16      size ,
                          U8       sortType
                         );
```

功能描述：

创建一个消息队列。

函数参数：

[in] qStart

消息指针存放地址

[in] size

消息队列的最大消息数

[in] sortType，等待列表排列方式 ：

EVENT_SORT_TYPE_FIFO，先进先出排序

EVENT_SORT_TYPE_PRIO，优先级抢占排序

返回值：

队列ID

示例：

```
#include "CCRTOS.h"
OS_EventID queue;
void *MailQueue[8];
void TaskN (void *pdata)
{
      ...
     /* Create a queue */
  queue = CoCreateQueue (MailQueue, 8, EVENT_SORT_TYPE_PRIO);
  if (queue == E_CREATE_FAIL)
  {
     printf("Create a queue fail !\n");
  }
  else
  {
     printf("Queue ID : %d \n", queue);
  }
      ...
}
```

备注：

无

6.9.2 CoDelQueue()

函数原型：

```
StatusType CoDelQueue(
                        OS_EventID   id,
                        U8           opt
                        );
```

功能描述：

删除指定ID消息队列。

函数参数：

[in] id

指定队列ID；

[in] opt

删除指定消息队列的方式：

EVENT_DEL_NO_PEND，　等待列表为空时删除

EVENT_DEL_ANYWAY，　无条件删除

返回值：

E_INVALID_ID，	传入的消息队列ID为无效ID
E_INVALID_PARAMETER，	无效参数，即ID对应的事件控制块不存在
E_TASK_WAITING，	等待列表非空
E_OK，	删除成功

示例：

```
#include "CCRTOS.h"
OS_EventID queue;
void *MailQueue[8];
void TaskN (void *pdata)
{
  StatusType result;
      ...
    /* Delete the specified queue */
  result = CoDelQueue(queue, OPT_DEL_ANYWAY);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
      {
          printf("Invalid Queue ID !\n");
      }
  }
      ...
}
```

备注：

1）成功删除则释放该消息队列所占用的资源，并进行任务调度，将系统执行时间让给优先级最高的任务。

6.9.3 CoAcceptQueueMail()

函数原型：

```
void*  CoAcceptQueueMail(
                        OS_EventID  id,
                        StatusType*  perr
                        );
```

功能描述：

无等待请求指定的消息队列消息。

函数参数：

[in] id

消息队列的ID

[out] perr

错误标志返回：

E_INVALID_ID，	无效的消息队列ID
E_QUEUE_EMPTY，	消息队列为空
E_OK，	成功接收消息

返回值：

NULL，	接收失败
Others，	接收的消息指针

示例：

```
#include "CCRTOS.h"
OS_EventID queue;
void *MailQueue[8];
unsigned int msgdat;
void TaskN (void *pdata)
{
  StatusType result;
  void *msg;
      ...
     /* Receive a mai without waiting */
  msg = CoAcceptQueueMail (queue, &result);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
      {
          printf("Invalid Queue ID !\n");
      }
      else if (result == E_QUEUE_EMPTY)
      {
          printf("Queue is empty !\n");
      }
  }
  else
  {
      if (msg)
      {
          printf("Have recived data : %d \n", *(unsigned int *)msg);
      }
        /* Process mail here */
  }
      ...
}
```

备注：

1) 调用AcceptQueueMail()函数时，若指定消息队列内的消息个数大于0，则获得消息队列里的第一个消息，并对当前消息队列的消息数进行减1操作。否则，若消息个数等于0，则不等待，立刻错误返回。

6.9.4 CoPendQueueMail()

函数原型：

```
void*   CoPendQueueMail(
                        OS_EventID   id,
```

```
                                U32             timeout,
                                StatusType*     perr
                                );
```

功能描述：

获得一个指定ID消息队列消息。

函数参数：

[in] id

指定消息队列ID

[in] timeout

超时时间，0表示无限期等待

[out] perr

错误返回类型：

E_CALL，	在中断服务程序里调用
E_INVALID_ID，	传入的消息队列ID为无效ID
E_TIMEOUT，	等待消息超时
E_OK，	成功获得资源

返回值：

获得队列信息指针。

示例：

```
#include "CCRTOS.h"
OS_EventID queue;
void *MailQueue[8];
unsigned int msgdat;
void TaskN (void *pdata)
{
  StatusType result;
  void *msg;
      ...                                  /* Wait for a mail, time-out:20 */
  msg = CoPendQueueMail (queue, 20, &result);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
          printf("Invalid Queue ID !\n");
  }
  else
  {
      if (msg)
          printf("Have recived data : %d \n", *(unsigned int *)msg);
                                           /* Process mail here */
  }
      ...
}
void TaskM (void *pdata)
{
  StatusType result;
      ...                                    /* Wait for a mail */
  msgdat = 0x61;
  result = CoPostQueueMail (queue, (void *)&msgdat);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
          printf("Invalid Queue ID ! \n");
      else if (result == E_MBOX_FULL)
          printf("The Queue is full !\n");
  }
}
```

备注：

1）该函数在等待过程中执行一次任务调度，将系统执行时间让给优先级最高的任务。

2）成功获得消息队列的消息后，会对当前消息队列的消息数进行减 1 操作。

3）该函数不能用于中断服务程序。

6.9.5 CoPostQueueMail()

函数原型：

```
StatusType CoPostQueueMail(
                            OS_EventID   id,
                            void*        pmail
                            );
```

功能描述：

发送消息至指定的消息队列。

函数参数：

[in] id

消息队列ID

[in] pmail

消息的指针

返回值：

E_INVALID_ID，	无效的消息队列ID
E_QUEUE_FULL，	消息队列已满
E_OK，	消息发送成功

示例：

```
#include "CCRTOS.h"
OS_EventID queue;
void *MailQueue[8];
unsigned int msgdat;
void TaskN (void *pdata)
{
  StatusType result;
  void *msg;
      ...
     /* Wait for a mail, time-out:20 */
  msg = CoPendQueueMail (queue, 20, &result);
  if (result != E_OK)
  {
     if (result == E_INVALID_ID)
         printf("Invalid Queue ID !\n");
  }
  else
  {
      if (msg)
         printf("Have recived data : %d \n", *(unsigned int *)msg);
  }
      ...
}
void TaskM (void *pdata)
{
  StatusType result;
      ...
     /* Wait for a mail */
  msgdat = 0x61;
  result = CoPostQueueMail (queue, (void *)&msgdat);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
         printf("Invalid Queue ID ! \n");
      else if (result == E_MBOX_FULL)
         printf("The Queue is full !\n");
  }
}
```

备注：

1）若指定消息列表内的消息个数等于最大消息个数，则丢弃该消息，错误返回。

6.9.6 isr_PostQueueMail()

函数原型：

```
StatusType isr_PostQueueMail(
                                    OS_EventID   id,
                                    void*            pmail
                                    );
```

功能描述：

在中断服务程序中，发送消息至一个指定 ID 的消息队列。

函数参数：

[IN] id

消息队列 ID

[IN] pmail

消息指针

返回值：

E_SEV_REQ_FULL	中断服务请求已满
E_INVALID_ID	无效的消息队列 ID
E_MBOX_FULL	消息队列已满
E_OK	成功放松消息至消息队列

示例：

```
#include "CCRTOS.h"
OS_EventID mailqueue;
int IsrDat;
void XXX_IRQHandler(void)
{
    StatusType result;
    EnterISR();        // Enter ISR
    ...
    IsrDat = 0x12;
    /* Post a mail to MailQueue that created by other tase */
    result = isr_PostQueueMail (mailqueue, &IsrDat);
    if (result != E_OK) {
        if (result == E_SEV_REQ_FULL) {
            printf("Service requst queue is full !\n");
        }
    }
      ...
    ExitISR();       // Exit ISR
}
```

备注：

1）不能在中断服务程序中调用 CoPostQueueMail（）来实现消息队列消息的发送，否则将导致系统混乱。

6.10 事件标志

6.10.1 CoCreateFlag()

函数原型：

```
OS_FlagID   CoCreateFlag(
                            BOOL bAutoReset,
                            BOOL bInitialState
                            );
```

功能描述：

创建一个事件标志。

函数参数：

[in] bAutoReset

重置方式：

1，自动重置

0，人工重置

[in] bInitialState

初始状态：

1，可通知状态

0，不可通知状态

返回值：

事件标志 ID，　　创建成功

-1，　　创建失败

示例：

```
#include "CCRTOS.h"
OS_FlagID flag;
void TaskN (void *pdata)
{
      ...
     /* Create a flag with auto reset,initial state: 0 */
  flag = CoCreateFlag (1, 0);
  if (result != E_OK)
  {
      if (result == E_CREATE_FAIL)
      {
          printf("Create flag fail !\n");
      }
  }
  else
  {
      printf("Flag ID : %d \n", flag);
  }
      ...
}
```

备注：

无

6.10.2 CoDelFlag()

函数原型：

```
StatusType CoDelFlag(
                        OS_FlagID   id,
                        U8          opt
                        );
```

功能描述：

删除指定事件标志。

函数参数：

[in] id

指定事件标志ID

[in] opt

删除指定事件标志的方式：

EVENT_DEL_NO_PEND， 等待列表为空时删除

EVENT_DEL_ANYWAY， 无条件删除

返回值：

E_INVALID_ID， 传入的事件标志ID为无效ID；

E_TASK_WAITING， 等待列表非空；

E_OK，　　删除成功；

示例：

```
#include "CCRTOS.h"
OS_FlagID flag;
void TaskN (void *pdata)
{
  StatusType result;
      ...
     /* Delete the Flag */
  result = CoDeleteFlag (flag, OPT_DEL_ANYWAY);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
     {
          printf("Invalid Flag ID !\n");
      }
  }
      ...
}
```

备注：

1）成功删除事件标志则释放其所占用的资源，并进行任务调度，将系统执行时间让给优先级最高的任务。

6.10.3 CoAcceptSingleFlag()

函数原型：

```
StatusType CoAcceptSingleFlag(
                                    OS_FlagID   id
                                    );
```

功能描述：

无等待请求单个事件标志。

函数参数：

[in] id

指定事件标志ID

返回值：

E_INVALID_ID，	传入的事件标志ID为无效ID
E_FLAG_NOT_READY，	事件标志处于未通知状态
E_OK，	请求成功

示例：

```
#include "CCRTOS.h"
OS_FlagID flag;
void TaskN (void *pdata)
{
  StatusType result;
      ...
  result = CoAcceptSingleFlag (flag);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
      {
          printf("Invalid Flag ID !\n");
      }
      else if (result == E_FLAG_NOT_READY)
      {
           printf("None get !\n");
      }
  }
  else
  {
          /* process Flag here */
  }
      ...
}
```

备注：

无

6.10.4 CoAcceptMultipleFlags()

函数原型：

```
U32   CoAcceptMultipleFlags(
                              U32          flags,
                              U8           waitType,
                              StatusType*  perr
                              );
```

功能描述：

无等待请求多个事件标志。

函数参数：

[in] flags

需要等待的事件标志

[in] waitType

等待类型：

OPT_WAIT_ALL，　　等待所有事件标志
OPT_WAIT_ANY，　　等待任意一个事件标志

[out] Perr

错误状态返回：

E_INVALID_PARAMETER，　无效参数
E_FLAG_NOT_READY，　　事件标志处于未通知状态
E_OK，　　获取成功

返回值：

触发的事件标志

示例：

```
#include "CCRTOS.h"
OS_FlagID flag;
void TaskN (void *pdata)
{
  StatusType result;
  U32 getFlag
      …
  flagID1 = CoCreateFlag(0,0); // 人工重置，初始为非就绪态
  flagID2 = CoCreateFlag(0,0); // 人工重置，初始为非就绪态
  flagID3 = CoCreateFlag(0,0); // 人工重置，初始为非就绪态
  Flag = 1 << flagID1 | 1 << flagID2 | 1 << flagID3;
  getFlag = CoAcceptMultipleFlags (Flag, OPT_WAIT_ALL, &result);
  if (result != E_OK)
  {
      if (result == E_INVALID_PARAMETER)
     {
          printf("Invalid Parameter !\n");
      }
      else if (result == E_FLAG_NOT_READY)
     {
           printf("Flag not ready !\n");
      }
  }
  else
  {
         /* process Flag here */
  }
      …
}
```

备注：

无

6.10.5 CoWaitForSingleFlag()

函数原型：

```
StatusType  CoWaitForSingleFlag(
                                OS_FlagID  id,
                                U32        timeout
                                );
```

功能描述：

等待一个指定ID事件标志。

函数参数：

[in] id

指定事件标志ID

[in] timeout

超时时间，0表示无限期等待

返回值：

E_CALL，	在中断服务程序里调用
E_INVALID_ID，	传入的事件标志ID为无效ID
E_TIMEOUT，	等待超时
E_OK，	成功获得事件标志

示例：

```
#include "CCRTOS.h"
OS_FlagID flag;
void TaskN (void *pdata)
{
  StatusType result;
      ...
     /* Waiting for a flag, time-out:20 */
  result = CoWaitForSingleFlag (flag, 20);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
      {
          printf("Invalid Flag ID !\n");
      }
      else if (result == E_CALL)
      {
           printf("Error call in ISR !\n");
      }
      else if (result == E_TIMEOUT)
      {
          printf("Time Out !\n");
      }
  }
  else
  {
          /* process Flag here */
  }
      ...
}
```

备注：

无

6.10.6 CoWaitForMultipleFlags()

函数原型：

```
U32 CoWaitForMultipleFlags(
                              U32          flags,
                              U8           waitType,
                              U32          timeout,
                              StatusType*  perr
                              );
```

功能描述：

等待多个事件标志。

函数参数：

[in] flags

需要等待的事件标志；

[in] waitType

等待类型：

OPT_WAIT_ALL，	等待所有事件标志
OPT_WAIT_ANY，	等待任意一个事件标志

[in] timeout

超时时间，0表示无限期等待；

[out] perr

错误状态返回：

E_CALL，	在中断服务程序里调用
E_INVALID_PARAMETER，	无效参数
E_TIMEOUT，	等待超时
E_FLAG_NOT_READY，	事件标志处于未通知状态
E_OK，	等待成功

返回值：

触发任务返回值就绪态的事件标志

示例：

```
#include "CCRTOS.h"
OS_FlagID flag;
void TaskN (void *pdata)
{
  StatusType result;
  U32   getFlag;
      ...
  flagID1 = CoCreateFlag(0,0); // 人工重置，初始为非就绪态
  flagID2 = CoCreateFlag(0,0); // 人工重置，初始为非就绪态
  flagID3 = CoCreateFlag(0,0); // 人工重置，初始为非就绪态
  Flag = 1 << flagID1 | 1 << flagID2 | 1 << flagID3;
  getFlag = CoWaitForMultipleFlags (Flag, OPT_WAIT_ALL, 20, &result);
  if (result != E_OK)
  {
      if (result == E_INVALID_PARAMETER)
      {
          printf("Invalid parameter !\n");
      }
      else if (result == E_CALL)
      {
           printf("Error call in ISR !\n");
      }
      else if (result == E_TIMEOUT)
      {
          printf("Time Out !\n");
      }
  }
  else
  {
          /* process Flag here */
  }
      ...
}
```

备注：

无

6.10.7 CoClearFlag()

函数原型：

```
StatusType   CoClearFlag(
                          OS_FlagID   id
                          );
```

功能描述：

设置一个事件为不可通知状态。

函数参数：

[in] id

指定事件标志ID

返回值：

E_INVALID_ID，　　传入的事件标志ID为无效ID；
E_OK，　　清除成功；

示例：

```
#include "CCRTOS.h"
OS_FlagID flag;
void TaskN (void *pdata)
{
  StatusType result;
      ...
  result = CoClearFlag (flag);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
     {
          printf("Invalid Flag ID !\n");
      }
  }
  else
  {
          /* process Flag here */
  }
      ...
}
```

备注：

1）该函数一般作用于人工重置的事件标志。

6.10.8 CoSetFlag()

函数原型：

```
StatusType  CoSetFlag(
                        OS_FlagID  id
                        );
```

功能描述：

设置一个事件为可通知状态。

函数参数：

[in] id

指定事件标志ID

返回值：

E_INVALID_ID， 传入的事件标志ID为无效ID
E_OK， 设置成功

示例：

```
#include "CCRTOS.h"
OS_FlagID flag;
void TaskN (void *pdata)
{
  StatusType result;
      ...
  result = CoSetFlag (flag);
  if (result != E_OK)
  {
      if (result == E_INVALID_ID)
      {
          printf("Invalid Flag ID !\n");
      }
  }
  else
  {
          /* process Flag here */
  }
      ...
}
```

备注：

1）若成功激活任务，该函数的调用将产生一次任务调度。

6.10.9 isr_SetFlag()

函数原型：

```
StatusType isr_SetFlag(
                        OS_FlagID   id
                        )
```

功能描述：

在中断服务程序中，设置一个指定 ID 的事件标志为 ready 状态。

函数参数：

[IN] id
事件标志 ID

返回值：

E_SEV_REQ_FULL 中断服务请求已满
E_INVALID_ID 无效的事件标志 ID
E_OK 成功设置指定事件标志

示例：

```
#include "CCRTOS.h"
OS_FlagID flag;
void XXX_IRQHandler(void)
{
    StatusType result;
    EnterISR();         // Enter ISR
    ...
    /* Set a flag that created by other test */
    result = isr_SetFlag (flag);
    if (result != E_OK) {
        if (result == E_SEV_REQ_FULL) {
            printf("Service requst queue is full !\n");
        }
    }
      ...
    ExitISR();        // Exit ISR
}
```

备注：

1）该函数用于中断服务程序中。

2）不能在中断服务程序中调用 CoSetFlag（）来实现事件标志的设置，否则将导致系统混乱。

6.11 系统工具

6.11.1 CoTickToTime()

函数原型：

```
void TickToTime(
                    U32  ticks,
                    U8*  hour,
                    U8*  minute,
                    U8*  sec,
                    U16* millsec
                    );
```

功能描述：

将systick数转换成相应的时间。

函数参数：

[in] ticks

systick数

[in] hour

小时

[in] minute

分钟

[in] sec

秒

[in] millsec

毫秒

返回值：

无

示例：

```
#include "CCRTOS.h"
void TaskM (void *pdata)
{
    U8 Hour, Minute, Second, Mircosecond;
    ...
    CoTickToTime (1949,
                  &Hour,
                  &Minute,
                  &Second,
                  &Mircosecond);
    printf ("1949 system ticks = % 2d-% 2d-% 2d-% 3d \n",
    Hour,Minute,Second,Mircosecond);
    ...
}
```

备注：

无

6.11.2 CoTimeToTick()

函数原型：

```
StatusType CoTimeToTick(
                          U8 hour,
                          U8 minute,
                          U8 sec,
                          U16 millsec,
                          U32* ticks
                         );
```

功能描述：

将时间转换成相应的systick 数。

函数参数：

[in] hour

小时

[in] minute

分钟

[in] sec

秒

[in] millsec

毫秒

[in] ticks

systick 数

返回值：

E_INVALID_PARAMETER， 无效参数

E_OK， 转换成功

示例：

```
#include "CCRTOS.h"
void TaskM (void *pdata)
{
    U32 tick;
    StatusType result;
    ...
    result = CoTimeToTick (19,
                           49,
                           10,
                           1,
                           &tick);
    if (result != E_OK)
    {
        if (result == E_INVALID_PARAMETER)
        {
            printf("Invalid parameter be passed and convert fail !\n");
        }
    }
    ...
}
```

备注：

无

6.12 其它

6.12.1 CoIdleTask()

函数原型：

```
void  CoIdleTask(
                  void* pdata
                  );
```

功能描述：

空闲任务函数体。

函数参数：

[IN] pdata

空闲任务函数体的传入参数链表，系统设定 NULL

返回值：

无

示例：

```
void CoIdleTask(void* pdata)
{
 /* 空闲任务处理 */
 for (;;)
 {
 /* 空闲任务处理 */
 }
}
```

备注：

1）空闲任务函数做为系统的驻留函数，不能被删除和自我退出，用户可以利用 CoIdleTask 实现一些系统参数的统计等。

6.12.2 CoStkOverflowHook()

函数原型：

```
void  CoStkOverflowHook(
                         OS_TID  taskID
                         );
```

功能描述：

系统堆栈溢出时的回调函数。

函数参数：

[IN] taskID

引发系统堆栈溢出的任务 ID

返回值：

无

示例：

```
void CoStkOverflowHook(OS_TID taskID)
{
 /* 在此添加对堆栈溢出的处理 */
}
```

备注：

1）此函数作为系统堆栈溢出的处理函数，来方便用户对堆栈溢出做自定义操作，若未对堆栈溢出对相关操作，而退出该函数，将导致系统混乱。