型嵌入式系统，死锁并不是一个大问题，因为系统设计者对整个应用程序都非常清楚，所以能够找出发生死锁的代码区域，并消除死锁问题。

守护任务

守护任务提供了一种干净利落的方法来实现互斥功能，而不用担心会发生优先级反转和死锁。

守护任务是对某个资源具有唯一所有权的任务。只有守护任务才可以直接访问其守护的资源——其它任务要访问该资源只能间接地通过守护任务提供的服务实现。

附：嵌入式实时操作系统TiniOS常用接口函数介绍

嵌入式实时操作系统 TiniOS 在设计时，其文件命名、函数名及变量命名由专用的前缀进行区分：前缀为 OS，表示为 TiniOS 的内核，这些是与平台无关的内核部分，在进行跨平台移植时，无需更改；前缀为 Fit，表示为硬件（芯片类型等）相关的部分，在进行移植时，这一部分的文件、函数及变量需要根据硬件平台（芯片类型等）进行适当的调整；

为了突显嵌入式操作系统配置及裁剪的灵活性，在 TiniOS 系统设计之初就进行了全面的考量，系统中相关参数及功能模块采用了宏定义的方式进行配置，常用的配置参数均保存在文件“OSType.h”中，为便于移植，相应配置信息均可以在“OSPreset.h”文件中进行重新定义，不推荐用户直接更改“OSType.h”文件。系统中常用的配置如下所示：

OSTICK_RATE_HZ：该参数配置了系统运行的“速率”，其决定了系统内核调度的最小时间粒度。默认配置为 1000Hz，最小时间粒度为 1 毫秒。在 Demo 示例工程中，采用 SETOS_TICK_RATE_HZ 配置；
OSNAME_MAX_LEN：系统中名称的最大长度，包括任务名称，软件定时器名称等，默认配置为 10 字节。在 Demo 示例工程中，采用 SETOS_MAX_NAME_LEN 配置；
OSLOWEAST_PRIORITY：系统最低优先级数值，配置为 0；
OSHIGHEAST_PRIORITY：系统可使用的优先级数。原则上讲，系统不会限制优先级数量，不过优先级越多，占用的资源越多，推荐优先级数不超过 32，系统默认优先级数为 8。由于任务最低优先级从 0 开始，则用户实际可使用的优先级范围为 0 到 OSHIGHEAST_PRIORITY-1。在附带的 Demo 示例工程中，通过宏定义变量 SETOS_MAX_PRIORITIES 对优先级数进行配置；
OSTOTAL_HEAP_SIZE：为 TiniOS 操作系统分配的栈空间大小。用户采用系统函数创建的任务、信号量、互斥锁、消息队列、软件定时器等均使用该栈空间。在 Demo 示例工程中，采用 SETOS_TOTAL_HEAP_SIZE 配置。
OSMINIMAL_STACK_SIZE：为任务分配的最小栈空间大小，默认配置 32 个字长。任务使用的栈空间量由具体的任务决定，若任务中局部变量较多，使用空间较大，则需配置更大的栈空间。在 Demo 示例工程中，采用 SETOS_MINIMAL_STACK_SIZE 配置；
OSPEND_FOREVER_VALUE：定义永久挂起/等待的数值，用于信号量、互斥锁、消息队列等永久等待的定义数值，在 32 位宽的芯片中，推荐配置为 0xFFFFFFFF。在 Demo 示例工程中，采用 SETOS_PEND_FOREVER_VALUE 配置；
OS_SEMAPHORE_ON：是否启用系统信号量的标识。数值为 1 则启用信号量功能，数值为 0，则不启用信号量。在 Demo 示例工程中，采用 SETOS_USE_SEMAPHORE 配置；
OS_MSGQ_ON：是否启用消息队列的标识。数值为 1 则启用消息队列，数值为 0，则不启用消息队列。在 Demo 示例工程中，采用 SETOS_USE_MSGQ 配置；
OSMSGQ_MAX_MSGNUM：消息队列中保存的消息数量，默认配置为 5。在向该消息队列发送消息时，若消息数达到该数值时，则消息队列已满，需挂起等待，或者把旧数据覆盖掉。在 Demo 示例工程中，采用 SETOS_MSGQ_MAX_MSGNUM 配置；
OS_MUTEX_ON：是否启用互斥锁的标识。数值为 1 则启用互斥锁，数值为 0，则不启用互斥锁。在 Demo 示例工程中，采用 SETOS_USE_MUTEX 配置；
OS_TIMER_ON：是否启用软件定时器的标识。数值为 1 则启用软件定时器，数值为 0，则不启用软件定时器。需要注意一点的是，系统软件定时器功能依赖消息队列，在使用软件定时器前，需要启用系统的消息队列功能。在 Demo 示例工程中，采用 SETOS_USE_TIMER 配置启用软件定时器；
OSCALLBACK_TASK_PRIO：用于配置软件定时器指令（如定时器启动与停止等）响应任务的优先级，推荐采

用 OSHIGHEAST_PRIORITY-1。在 Demo 示例工程中，采用 SETOS_CALLBACK_TASK_PRIORITY 配置；

OS_LOWPOWER_ON:是否启用低功耗标识。数值为 0，则不启用低功耗模式；数值为非 0，则启用低功耗模式，系统会在空闲任务中进入低功耗状态，直至有效任务恢复执行或者被外部中断唤醒。需要注意的是，采用低功耗模式，有可能会影响系统的时钟精准状态，产生时钟漂移；部分芯片不支持低功耗模式，因此系统默认不支持低功耗。

OS_TASK_SIGNAL_ON：是否启用轻量级同步信号/消息的标识。数值为 0，则不启动轻量级同步信号/消息，数值非 0，则启用轻量级同步信号/消息。轻量级同步信号与轻量级同步消息可以实现旗语 Semaphore 与消息队列 MsgQ 的部分功能，而占用的空间更小，该功能通常在芯片 Ram 非常小的情况下配置使用。

系统 TiniOS 是多任务抢占式操作系统，高优先级任务可以抢占执行，体现了操作系统的实时性。在 TiniOS 系统中，优先级 0 为系统最低优先级，该优先级为空闲任务 OSIdleTask 使用的优先级。用户可以根据任务的重要程度自行配置。

操作系统运行频率（速率）常用 Ticks 表示，Ticks 亦被称为时钟滴答。操作系统“滴答”由硬件以规律性的定时中断产生。在 TiniOS 系统中，时钟“滴答”决定了系统的最小时间粒度，这个参数可以根据硬件平台进行配置。在 TiniOS 系统中，该参数采用宏定义 SETOS_TICK_RATE_HZ 进行配置。在 TiniOS 提供的大部分示例工程中，均配置的为 1000Hz，即每个时钟滴答间隔是 1 毫秒。

操作系统启动后，会按照任务的优先级选择性的执行，最先执行处于等待状态的最高优先级任务，直至该任务让出执行权或者被其它更高优先级的任务抢断。在系统运行的过程中，如果没有符合条件的任务需要执行，则运行系统中预留的 OSIdleTask（空闲任务）。

下面是嵌入式实时操作系统 TiniOS 中经常用到的接口函数，供大家使用时参考；

一、系统中任务相关的 API 函数

```
OSTaskHandle_t OSTaskCreate(OSTaskFunction_t  pxTaskFunction,
                            void*             pvParameter,
                            const uOS16_t     usStackDepth,
                            uOSBase_t         uxPriority,
                            sOS8_t*           pcTaskName)
```

OSTaskCreate 为 TiniOS 系统的任务创建函数，其中参数 OSTaskFunction_t pxTaskFunction 为任务函数，该任务函数定义为 void TaskFunction(void *pParameters)；任务函数中的参数 void *pParameters 亦即 OSTaskCreate 的第二个参数；第三个参数为任务的栈空间 usStackDepth，栈空间需要根据任务占用的空间多少进行调整；第四个参数为任务的优先级，用户可根据该任务的重要程度自行配置其优先级。第五个参数为任务名字，任务名字也就是任务的标签，主要方便在调试时区分不同的任务。

函数 OSTaskCreate 的返回值为 OSTaskHandle_t 类型的任务句柄，该句柄可以被其它系统函数调用，以设置或控制任务的状态；

```
void OSTaskSleep( const uOSTick_t uxTicksToDelay )
```

OSTaskSleep 为 TiniOS 系统中任务延迟执行的设置函数，通过此函数，可以把当前任务休眠若干毫秒的时间。参数 uOSTick_t uxTicksToDelay 代表休眠的时间长短，单位为 Ticks，用户可以通过 OSM2T()把毫秒转换为 Ticks 计数；

OSSchedule()

OSSchedule 函数为 TiniOS 系统中的任务控制类函数，在任务函数中调用，用于让出当前任务的执行权，并切换到下一个处于等待执行状态的任务；函数 OSSchedule 会把当前任务移到等待执行队列的末尾，若只有当前任务处于等待执行状态，则仍然执行该任务；

```
void OSTaskSuspend( OSTaskHandle_t TaskHandle )
```

OSTaskSuspend 函数为系统任务挂起函数，任务挂起后，将不再参与系统的调度，但任务仍然保留在系统中，任务占用的资源不会释放。参数 TaskHandle 用于表示待挂起的任务句柄，该数值若为 OS_NULL，则挂起当前正在运行的任务；

```
void OSTaskResume( OSTaskHandle_t TaskHandle )
```

OSTaskResume 函数用于把挂起的任务恢复（激活），使该任务重新参与系统的调度，参数 TaskHandle 表示待恢复的任务句柄，该参数不允许为 OS_NULL。此函数不是中断安全函数，不允许在中断函数中调用该函数；

```
sOSBase_t OSTaskResumeFromISR( OSTaskHandle_t TaskHandle )
```

OSTaskResumeFromISR 函数用于把挂起的任务恢复（激活），使该任务重新参与系统的调度，参数 TaskHandle 表示待恢复的任务句柄，该参数不允许为 OS_NULL；此函数为中断安全函数，只能在中断服务函数中调用；

```
uOSBase_t OSInit( void )
```

OSInit 函数为 TiniOS 中的系统参数初始化函数；在该函数中，会为系统的堆栈进行初始化，便于为系统的任务、信号量、同步消息与互斥锁等分配空间；同时，会对系统中必要的全局变量进行初始化。尽管大部分变量已经在定义时进行了初始化，但是部分芯片首次上电启动后默认数值仍然不明确，有必要调用此接口函数进行初始化确认；**注：此函数需要在使用 TiniOS 任何接口函数前调用；**

```
uOSBase_t OSStart( void )
```

OSStart 函数是 TiniOS 系统中的任务调度启动函数；在该函数中，系统会设置空闲任务 OSIdleTask 及时钟中断（时钟滴答）；OSIdleTask 任务为系统空闲任务，若系统中无其它需要执行的任务，则会调用该空闲任务，空闲任务可以用于统计当前系统的利用率，及释放处于待删除状态任务的资源；时钟中断则为系统的 ticks 配置，整个系统的运行即依赖此 ticks 驱动运行；调用此函数后，TiniOS 系统就开始启动运行了，因此在正常情况下不会接收到该函数的返回值。一旦接收到该函数的返回值，则表明 TiniOS 系统启动异常，有可能是为系统分配的堆栈空间太小，导致系统无法启动，此时需要进一步排查确认；

二、任务同步信号量相关的 API 函数

```
OSSemHandle_t   OSSemCreate( const uOSBase_t uxInitialCount )
```

函数 OSSemCreate 为信号量（Semaphore）创建函数，用于创建任务间及中断与任务间同步的信号量。信号量创建时，可以通过参数 uxInitialCount 设置该信号量的有效数值，若设置为 0，表示该信号量创建后无有效信号，对应的 OSSemPend（）函数处于阻塞状态，等待 OSSemPost()函数发送有效信号；信号量中的信号容量默认为 0xFF；函数返回值为 OSSemHandle_t 类型的句柄，方便用于对该信号量的操控；

```
sOSBase_t OSSemPend( OSSemHandle_t SemHandle, uOSTick_t xTicksToWait)
```

函数 OSSemPend 为信号量等待函数，在任务执行函数中调用，用于等待相关同步的信号量；参数 OSSemHandle_t SemHandle 为信号量句柄，参数 uOSTick_t xTicksToWait 为任务阻塞时间，单位为 Tick 数，若设置为 OSPEND_FORVER_VALUE，则会永远阻塞，直至指定信号量 SemHandle 获取到有效信号；

```
sOSBase_t OSSemPost( OSSemHandle_t SemHandle)
```

函数 OSSemPost 用于向指定信号量发送有效信号，使处于等待该信号量的任务获取同步信号，以便恢复执行。注：此函数不能在中断服务函数中调用。

```
sOSBase_t OSSemPostFromISR( OSSemHandle_t SemHandle)
```

函数 OSSemPostFromISR 用于向指定信号量发送有效信号，使处于等待该信号量的任务获取同步信号，以便恢复执行。注：此函数只能在中断服务函数中调用。

三、任务同步消息相关的 API 函数

```
OSMsgQHandle_t OSMsgQCreate( const uOSBase_t uxQueueLength, const uOSBase_t uxItemSize)
```

函数 OSMsgQCreate 为消息队列的创建函数，用于创建任务间同步操作的消息队列，参数 uxQueueLength 为消息队列中的消息数的容量（消息数目超过此容量，则发送任务挂起，直到消息队列有空闲位置），第二个参数 uxItemSize 为单个消息的长度。其返回值为 OSMsgQHandle_t 类型的消息队列句柄，方便对消息队列的操控；

```
sOSBase_t OSMsgQReceive( OSMsgQHandle_t MsgQHandle, void * const pvBuffer, uOSTick_t xTicksToWait)
```

函数 OSMsgQReceive 用于在任务中接收指定消息队列的消息，在任务函数中调用。该函数为任务阻塞函数。参数 MsgQHandle 为消息队列句柄，参数 pvBuffer 表示消息的指针，参数 xTicksToWait 为消息队列等待接收时间，单位为 Tick，若设置为 OSPEND_FORVER_VALUE，则会永远等待，直至指定消息队列 MsgQHandle 获取到有效消息为止。函数返回值代表消息接收状态，若为 OS_FALSE 则未接收到有效消息，若为 OS_TRUE 则接收到有效消息；

```
sOSBase_t OSMsgQReceiveFromISR( OSMsgQHandle_t MsgQHandle, void * const pvBuffer)
```

函数 OSMsgQReceiveFromISR 用于在中断函数中接收指定消息队列的消息；参数 MsgQHandle 为消息队列句柄，参数 pvBuffer 表示消息的指针，该函数不会阻塞。函数返回值代表消息接收状态，若为 OS_FALSE 则未接收到有效消息，若为 OS_TRUE 则接收到有效消息；注：此函数只能在中断服务函数中调用。

```
sOSBase_t OSMsgQSend( OSMsgQHandle_t MsgQHandle, const void * const pvItemToQueue, uOSTick_t xTicksToWait)
```

函数 OSMsgQSend 用于向指定的消息队列发送消息，使处于等待该消息的任务获取同步消息，并恢复执行，其中参数 MsgQHandle 表示消息队列，参数 pvItemToQueue 表示消息地址（指针），参数 xTicksToWait 为消息发送等待接收时间，单位为 Tick，若设置为 OSPEND_FORVER_VALUE，则会永远等待，直至指定消息队列 MsgQHandle 有空闲位置。函数返回值代表消息发送状态，若为 OS_FALSE 则消息发送失败，若为 OS_TRUE 则消息发送成功；注：此函数不能在中断服务函数中调用。

```
sOSBase_t OSMsgQSendFromISR( OSMsgQHandle_t MsgQHandle, const void * const pvItemToQueue)
```

函数 OSMsgQSendFromISR 用于向指定的消息队列发送消息，使处于等待该消息的任务获取同步消息，并恢复执行，其中参数 MsgQHandle 表示消息队列，参数 pvItemToQueue 表示消息地址（指针），如果消息队列已满，则函数不会阻塞，直接返回发送失败信息。注：此函数只能在中断服务函数中调用。

四、定时器相关的 API 函数

```
OSTimerHandle_t OSTimerCreate(const uOSBase_t uxTimerTicks, const uOS16_t uiIsPeriod, const OSTimerFunction_t Function, void* pvParameter, sOS8_t* pcName)
```

接口函数 OSTimerCreate 用于创建定时器。其中参数 uxTimerTicks 为定时器周期值，单位为系统滴答 Ticks，可以通过宏定义 OSM2T()把毫秒数值转化为系统 Ticks 数；参数 uiIsPeriod 表示定时器是否为周期性定时器，取值为 OS_OS_TRUE 时为周期性定时器，取值为 OS_OS_FALSE 时为单次定时器；参数 Function 为定时器的服务函数，用于响应定时器，函数 OSTimerFunction_t 的定义类型为 void TimerFunction(void *pParameters)。pvParameter 为定时器服务函数的参数，不用时可以设置为 OS_NULL；参数 pcName 为定时器的名称，方便区分不同的定时器；注意：定时器服务函数中禁止添加信号量等待、消息队列等待等阻塞函数，为不影响系统的性能，定时器服务函数耗时越少越好。

```
uOSBase_t OSTimerStart(OSTimerHandle_t const TimerHandle)
```

定时器创建完毕后并不会自动启动，需要用户调用启动函数 OSTimerStart()，之后定时器才会生效。参数 TimerHandle 为定时器句柄，为定时器创建函数 OSTimerCreate()的返回值；

```
uOSBase_t OSTimerStop(OSTimerHandle_t const TimerHandle)
```

定时启动后，用户可以通过接口函数 OSTimerStop()停止定时。参数 TimerHandle 为定时器句柄，为定时器创建函数 OSTimerCreate()的返回值；

五、轻量级同步信号的 API 函数

```
uOSBool_t OSTaskSignalWait( uOSTick_t const uxTicksToWait)
```

函数 OSTaskSignalWait()为轻量级信号量（Signal）等待函数，用于等待系统中别的任务函数或者中断响应函数发出的同步信号。任务函数在调用该接口函数后，即转入挂起等待状态。参数 uxTicksToWait 为等待时间，单位为 Tick，若设置为 OSPEND_FORVER_VALUE，则会永远等待，直至接收到同步信号。函数返回值代表信号接收状态，若为 OS_FALSE 则未接收到有效信号，若为 OS_TRUE 则接收到有效信号；注意，该函数不允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalEmit( OSTaskHandle_t const TaskHandle )
```

函数 OSTaskSignalEmit()为轻量级信号量（Signal）发射函数，向目标任务函数发送同步信号。参数 TaskHandle 为等待接收信号的目标任务句柄，禁止传入空指针；函数返回值代表信号发送状态，若为 OS_FALSE 则未发送出有效信号，若为 OS_TRUE 则成功发送出有效信号；注意，该函数不允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalEmitFromISR( OSTaskHandle_t const TaskHandle )
```

函数 OSTaskSignalEmitFromISR()为轻量级信号量（Signal）发射函数，向目标任务函数发送同步信号。参数 TaskHandle 为等待接收信号的目标任务句柄，禁止传入空指针；函数返回值代表信号发送状态，若为 OS_FALSE 则未发送出有效信号，若为 OS_TRUE 则成功发送出有效信号；注意，该函数只允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalClear( OSTaskHandle_t const TaskHandle )
```

函数 OSTaskSignalClear()用于对指定任务的信号进行清除，参数 TaskHandle 为等待接收信号的任务句柄，禁止传入空指针；函数返回值代表信号清除状态，若为 OS_FALSE 则未成功清除信号，若为 OS_TRUE 则成功清除对应任务中的信号数值，信号数值进行复位。

六、轻量级同步消息的 API 函数

```
uOSBool_t OSTaskSignalWaitMsg( sOSBase_t xSigValue, uOSTick_t const uxTicksToWait)
```

函数 OSTaskSignalWaitMsg()为轻量级同步消息等待函数，用于等待系统中别的任务函数或者中断响应函数发出的同步消息。任务函数在调用该接口函数后，即转入挂起等待状态。参数 xSigValue 为输出参数，在成功接收到消息后，该参数会获取具体的消息数值信息。参数 uxTicksToWait 为等待时间，单位为 Tick，若设置为 OSPEND_FORVER_VALUE，则会永远等待，直至接收到同步消息。函数返回值代表消息接收状态，若为 OS_FALSE 则未接收到有效消息，若为 OS_TRUE 则接收到有效消息；注意，该函数不允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalEmitMsg( OSTaskHandle_t const TaskHandle, sOSBase_t const xSigValue, uOSBool_t bOverWrite )
```

函数 OSTaskSignalEmitMsg()为轻量级消息发射函数，用于向指定的目标任务发送同步消息。参数 TaskHandle 为等待接收消息的任务句柄，即目标任务句柄，禁止向该参数传入空指针；参数 xSigValue 为待发送出的消息数值；参数 bOverWrite 表示既有的消息数值是否在目标任务未响应的情况下可以覆盖重写，若为 OS_TRUE，则代表可以覆盖重新赋值，若为 OS_FALSE，则该信号数值不允许覆盖，该函数直接返回 OS_FALSE，代表向目标任务发送消息失败。函数返回值代表消息发送状态，若为 OS_FALSE 则未发送出有效消息，若为 OS_TRUE 则发送出有效消息；注意，该函数不允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalEmitMsgFromISR( OSTaskHandle_t const TaskHandle, sOSBase_t const xSigValue, uOSBool_t bOverWrite )
```

函数 OSTaskSignalEmitMsgFromISR()为轻量级消息发射函数，用于向指定的目标任务发送同步消息。参数 TaskHandle 为等待接收消息的任务句柄，即目标任务句柄，禁止向该参数传入空指针；参数 xSigValue 为待发送出的消息数值；参数 bOverWrite 表示既有的消息数值是否在目标任务未响应的情况下可以覆盖重写，若为 OS_TRUE，则代表可以覆盖重新赋值，若为 OS_FALSE，则该信号数值不允许覆盖，该函数直接返回 OS_FALSE，代表向目标任务发送消息失败。函数返回值代表消息发送状态，若为 OS_FALSE 则未发送出有效消息，若为 OS_TRUE 则发送出有效消息；注意，该函数只允许在中断响应函数中使用。

```
uOSBool_t OSTaskSignalClear( OSTaskHandle_t const TaskHandle )
```

函数 OSTaskSignalClear()用于对指定任务的消息进行清除，参数 TaskHandle 为等待接收消息的任务句柄，禁止传入空指针；函数返回值代表消息清除状态，若为 OS_FALSE 则未成功清除消息，若为 OS_TRUE 则成功清除目标任务中的消息数值，消息数值进行复位。

简单示例程序

下面是采用嵌入式实时操作系统 TiniOS 实现的多任务处理演示示例，主要包括任务创建、信号量创建、消息队列创建、任务延时等功能。代码如下：

```
//任务句柄
OSTaskHandle_t      TaskCtrlHandle = OS_NULL;
OSTaskHandle_t      TaskTest1Handle = OS_NULL;
OSTaskHandle_t      TaskTest2Handle = OS_NULL;
//信号量句柄
OSSemHandle_t       SemHandle = OS_NULL;
//消息队列句柄
OSMsgQHandle_t      MsgQHandle = OS_NULL;
//任务函数
static void TaskCtrl( void *pvParameters );
static void TaskTest1( void *pvParameters );
static void TaskTest2( void *pvParameters );

int main( void )
{
    //Initialize the parameter in TiniOS
    OSInit();

    // 创建消息队列，消息容量5个，消息长度为int类长度.
    MsgQHandle = OSMsgQCreate( 5, sizeof( uint32_t ) );
    // 创建信号量
    SemHandle = OSSemCreate(0);
    // 创建任务
    TaskCtrlHandle = OSTaskCreate(TaskCtrl, OS_NULL, OSMINIMAL_STACK_SIZE,
OSLOWEAST_PRIORITY+1, "Ctrl");
    TaskTest1Handle = OSTaskCreate(TaskTest1, OS_NULL, OSMINIMAL_STACK_SIZE,
OSLOWEAST_PRIORITY+1, "Test1");
    TaskTest2Handle = OSTaskCreate(TaskTest2, OS_NULL, OSMINIMAL_STACK_SIZE,
OSLOWEAST_PRIORITY+1, "Test2");

    // 启动系统
    OSStart();
    for( ;; );
}

static void TaskCtrl( void *pvParameters )
{
    unsigned int uiValueToSend = 0;
    for( ;; )
    {
        //发送信号量
        OSSemPost(SemHandle);
        //发送消息队列
```

```
        OSMsgQSend( MsgQHandle, &uiValueToSend, OSPEND_FOREVER_VALUE );
        uiValueToSend += 1;
        //延时等待
        OSTaskSleep(200*OSTICKS_PER_MS);
    }
}

static void TaskTest1( void *pvParameters )
{
    unsigned int uiReceivedValue;
    for( ;; )
    {
        //等待接收消息
        OSMsgQReceive( MsgQHandle, &uiReceivedValue, OSPEND_FOREVER_VALUE );
        //do something here
    }
}

static void TaskTest2( void *pvParameters )
{
    for( ;; )
    {
        //等待信号量
        OSSemPend(SemHandle, OSPEND_FOREVER_VALUE);
        //do something here
    }
}
```