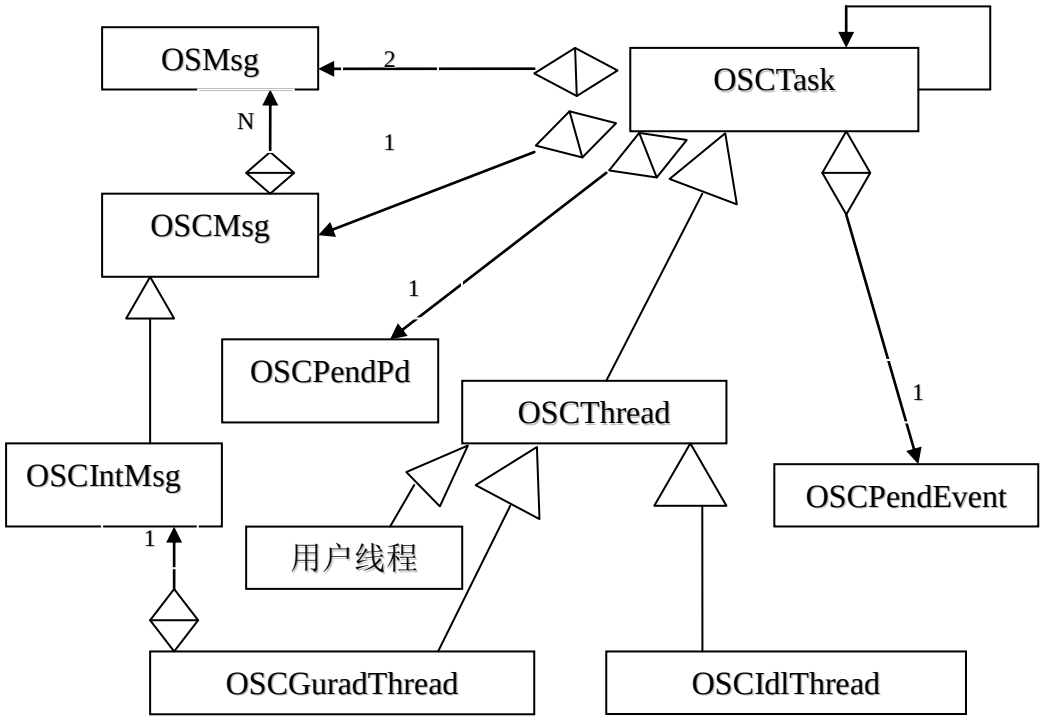
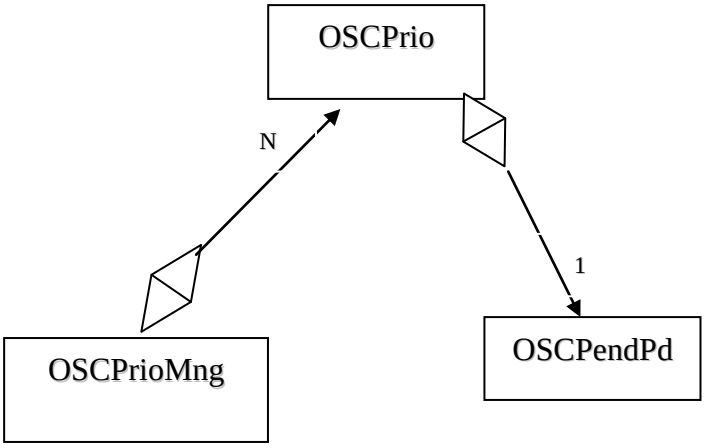
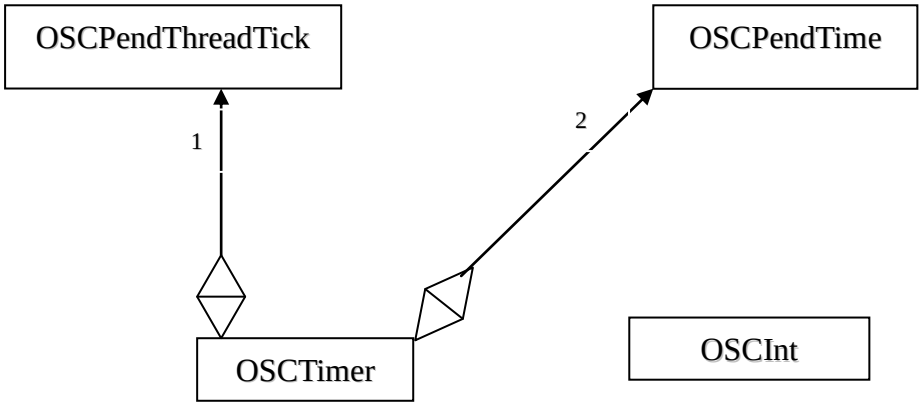
实时操作系统 Worker1.0 使用手册

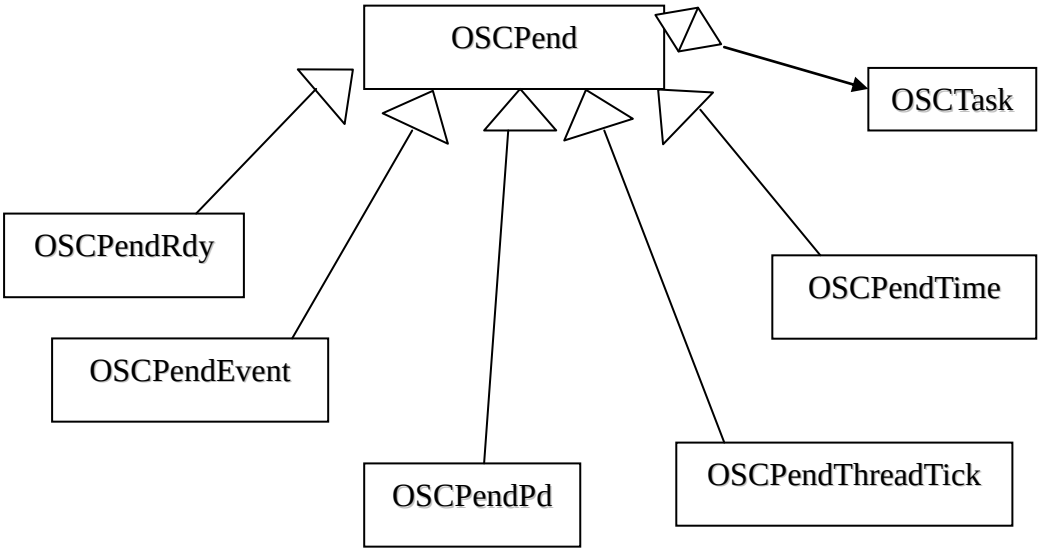
目录：

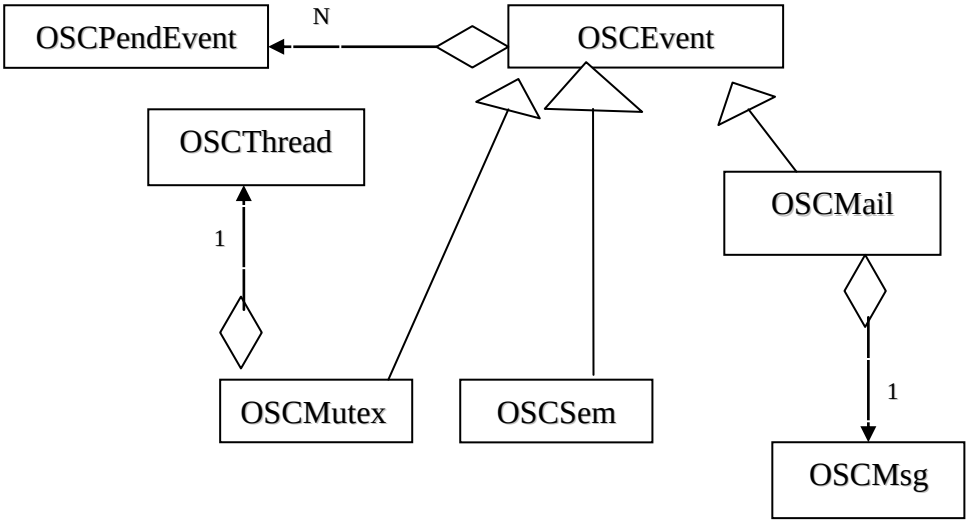
(一) Worker1.0 继承图




OSCPrio
N
1
OSCPrioMng
OSCPendPd


OSCPendThreadTick
OSCPendTime
1
2
OSCTimer
OSCInt





(二) Worker1.0 主要类的简介

类名	类功能简介
OSCTask	这个类是系统的 PCB，记录了线程的各种状态信息，如挂起、阻塞、等待信号量、睡眠 •••••• 还封装了线程通信的主要方法。
OSCPrioMng	线程的就绪队列，具备了运行条件的线程就会被放到这个队列中，这个类提供了用于队列进出的各种操作。但这些方法都不对用户公开的，用户只能通过其他函数的调用来间

	接调用这些方法。
OSCTimer	这个类管理系统需要时间服务的线程，用户可以用这个类提供的注册和卸载方法来决定是否需要系统的时间服务。
OSCInt	管理系统中断，用户通过这个类提供的方法申请中断号、注册中断函数。
OSCMsg	消息队列，这个类提供 FIFO 的消息队列
OSCThread	这个类继承与 OSCTask 类。这个类实现了线程的消息循环，从这个类派生出来的用户线程就已经具备消息循环的能力，用户只要在自己的派生类中重载线程过程函数 ThreadProc 并在这个函数添加消息响应代码就可以驱动线程的运行，线程在创建的时候会自动收到系统发来的消息 TM_CREATE。 用户也可以不使用系统预先设定的消息循环机制，这样用户可以通过重载线程主函数 ThreadMain。
OSCGuardThread	系统守护线程，为系统提供各种服务的线程。转发中断消息、进行时间服务、协助中断函数完成一些必须在任务环境做的工作。
OSCIdlThread	系统空闲时候调用的线程，可以进行各种空闲时候的操作
OSCEvent	为实现代码复用而虚构的类，他是 OSCMail、OSCSem、OSCMutex 三个类的基类。
OSCMail	消息邮箱，但这个邮箱可以存放多个消息，和其他系统讲的消息队列有点像。
OSCSem	信号量，会拿起我这份资料看的人都应该不用我介绍了吧！不支持优先级翻转，创建这个类的对象的时候要传递一个参数，最好是大于 1 的，否则应当使用 OSCMutex
OSCMutex	实现了优先级继承功能，当资源只有一个可用或者没有可用的时候就可以使用这个类。创建类的时候要传递一个参数，只能是 0 或 1
OSMsg	系统消息数据结构。要传递消息的时候，就要对创建这个类的变量，并初始化各种参数，调用各种消息传递的函数，实现消息的发送。

(三) Worker1.0 API

类	接口	功能
OSCTimer	static bool OnTimeInt();	定时器中断函数调用，给系统提供时间服务 返回值：总为 TRUE
	Static bool TimeServerRegister(OSCThread *ThreadPtr,uint32 Tick);	注册应用定时器，注册了的线程在指定的时间到达后就会收到消息 TM_THREADONTIME 【in】Param1：要注册的线程 ID 【in】Param2：要注册的时钟周期 返回值：成功 TRUE，失败 FALSE
	static bool TimeServerUnregister(OSCThread *ThreadPtr);	注销应用定时器 Param1：要取消的线程 ID 返回值：成功 TRUE，失败 FALSE
OSCPrioMng	static uint32 GetRunTime(uint32 Prio)	获取指定优先级的最大运行时间 【in】Param1：指定的优先级 返回值：指定优先级任务的最大运行时间

	static uint32 GetWaitTime(uint32 Prio)	获取指定优先级的最大等待时间 【in】Param1：指定的优先级 返回值:指定优先级任务的最大等待时间
	static uint32 SetRunTime(uint32 Prio , uint32 RunTime);	设置指定优先级的最大运行时间,返回原来时间 【in】Param1：优先级 【in】Param2：新的最大运行时间 返回值：原来的最大运行时间
	static uint32 SetWaitTime(uint32 Prio , uint32 WaitTime)	设置指定优先级的最大等待时间,返回原来时间 【in】Param1：优先级 【in】Param2：新的最大等待时间 返回值：原来的最大等待时间
OSCInt	static void IntEnable()	使能系统总中断
	static void IntDisable()	禁止系统总中断
	static int8 GetIntVector()	申请中断向量 返回值：成功 向量号 ， 失败 -1
	static bool PutIntVector(int8 Vector);	注销中断向量号 【in】Param1：要注销的中断向量号 返回值：成功 TRUE ， 失败 FALSE
	static bool IntRegister(void(*IntServe)(void),int8 Vector,int8 VICChannel)	中断注册函数 【in】Param1：中断函数地址 【in】Param2：中断向量号 【in】Param3：中断通道 返回值：成功 TRUE，失败 FALSE
	static void IntEnterInt();	中断函数调用,在进入中断后使能系统中断。此函数的调用将允许系统中断嵌套
	static void IntEnterNoint();	中断函数调用,在进入中断后防止误开系统中断。此函数的调用将禁止系统中断嵌套
	static void IntExitSwap();	中断函数调用,在退出中断时查看是否要任务调度
	static void IntExitMedi();	中断函数掉用,保证退出中断不会进行任务调度
OSCMail	bool MsgGetSwap(OSMsg *PMsg,uint32 Dly)	调用线程等待邮箱消息 【out】Param1：等到的消息 【in】Param2：最大的等待时间（0 表示无限等待） 返回值：成功 TRUE，失败 FALSE
	bool MsgPutSwap(OSMsg *Msg)	调用线程向邮箱发送消息 【in】Param1：要发送的消息 返回值：有任务被唤醒 TRUE，没有任

		务被唤醒 FALSE
	Bool MsgTeleviseSwap(OSMsg*Msg);	调用线程对等待消息的任务进行广播，如果调用线程优先级低于被唤醒的任务则会被剥夺 【in】Param1：要发送的消息指针 返回值：有任务被唤醒 TRUE，没有任务被唤醒 FALSE
	bool MsgTeleviseMedi(OSMsg*Msg);	调用线程对等待消息的任务进行消息广播，如果被唤醒的任务优先级高于调用线程也不会进行任务切换 返回值：有任务被唤醒 TRUE，没有任务被唤醒 FALSE
	bool MsgGetMedi(OSMsg *PMsg)	调用线程无等待取邮箱消息 【out】Param1：取出的消息 返回值：成功 TRUE，失败 FALSE
	bool MsgPutMedi(OSMsg *TMsg)	调用线程向邮箱发送消息 【in】Param1：要发送的消息 返回值：有任务被唤醒 TRUE，没有任务被唤醒 FALSE
OSCSem	bool SemGetMedi();	调用线程申请信号量，无论成功与否立刻返回 返回值：成功 TRUE，失败 FALSE
	bool SemPutMedi();	调用线程归还信号量，无论成功与否立刻返回 返回值：有任务被唤醒 TRUE，没有任务被唤醒 FALSE
	bool SemGetSwap(uint32 Dly);	调用线程申请信号量 【in】Param1：最大等待时间（0 表示无限等待） 返回值：成功 TRUE，失败 FALSE
	bool SemPutSwap();	调用线程归还信号量，如果被唤醒的线程优先级高则会被抢占 返回值：有任务被唤醒 TRUE，没有任务被唤醒 FALSE
OSCMutex	bool MutexGetMedi();	调用线程申请 Mutex，并立刻返回 返回值：成功 TRUE，失败 FALSE
	bool MutexPutMedi();	调用线程归还 Mutex，并立刻返回 返回值：有任务被唤醒 TRUE，没有任务被唤醒 FALSE
	bool MutexGetSwap(uint32 Dly);	调用线程申请 Mutex Param1：最大等待时间（0 表示无限等待） 返回值：成功 TRUE，失败 FALSE
	bool MutexPutSwap();	调用线程归还 Mutex，如果唤醒的任务

		优先级比调用的高，则会被抢占 返回值：有任务被唤醒 TRUE，没有任务被唤醒 FALSE
OSCMsg	bool IsEmpty()	判断消息队列是否为空 返回值：空 TRUE，不空 FALSE
	bool IsFull()	判断消息队列是否为满， 返回值：满 TRUE，布满 FALSE
	bool Add(OSMsg *TMsg);	向消息队列中添加消息 【in】Param1：要添加的消息内容 返回值：成功 TRUE，失败 FALSE
	bool Delete(OSMsg * TMsg);	删除消息队列中的一个消息 【out】Param1：返回的消息内容 返回值：成功 TRUE，失败 FALSE
OSCIntMsg	bool IsEmpty()	判断消息队列是否为空 返回值：空 TRUE，不空 FALSE
	bool IsFull()	判断消息队列是否为满， 返回值：满 TRUE，布满 FALSE
	bool Add(OSMsg *TMsg);	向消息队列中添加消息 【in】Param1：要添加的消息内容 返回值：成功 TRUE，失败 FALSE
	bool Delete(OSMsg * TMsg);	删除消息队列中的一个消息 【out】Param1：返回的消息内容 返回值：成功 TRUE，失败 FALSE
OSCTask	static void Schedul();	调度任务（建议在重载 ThreadMain 函数设定新的消息循环时才能使用）
	bool TaskPend();	挂起线程到挂起队列中，任务挂起后收到消息就会被唤醒（建议在重载 ThreadMain 函数设定新的消息循环时才能使用） 返回值：成功 TRUE，失败 FALSE
	static void TaskLock();	锁调度器，谨慎使用 返回值：成功 TRUE，失败 FALSE
	static void TaskUnlock();	解锁调度器，谨慎使用 返回值：成功 TRUE，失败 FALSE
	static bool SynMsgPost(OSCTask*TaskP,OSMsg*PMsg);	调用线程发送同步消息，调用者会挂起自身知道消息被取走 【in】Param1：要接受同步消息的线程 【in】Param2：要发送的消息 返回值：成功 TRUE，失败 FALSE
	static bool SynMsgWait(OSMsg*PMsg);	调用线程等待同步消息，调用者会挂起自身知道接收到同步消息 【out】Param1：接收到的消息 返回值：成功 TRUE，失败 FALSE
	static bool AsyMsgTelevise(OSMsg*PMsg);	调用者进行消息广播，在挂起队列中等

		待的线程才能接收到消息 【in】Param1：要广播的消息 返回值：成功 TRUE，失败 FALSE
	bool AsyMsgPost(OSMsg *TMsg)	调用线程发送异步消息，如果接收消息的线程优先级高于调用者，则会发生任务切换 【in】Param1：要发送的消息 返回值：成功 TRUE，失败 FALSE
	bool AsyMsgSend(OSMsg *TMsg);	调用线程发送异步消息，不会切换任务 【in】Param1：要发送的消息 返回值：成功 TRUE，失败 FALSE
	bool AsyMsgAccept(OSMsg *PMsg)	调用者接收异步消息，没有则直接返回 【out】Param1：收到的消息 返回值：成功 TRUE，失败 FALSE
	uint32 ChangePrio(uint32 Prio);	改变线程的优先级 【in】Param1：要设置的优先级 返回值：旧的优先级
	static void Sleep(uint32 Dly)	调用线程睡眠函数 【in】Param1：睡眠时间
	static void SysInit();	系统初始化函数，芯片初始化完成后第一时间调用
	static void StartUp();	启动多任务环境，这个函数调用后不能返回，只能调用一次
OSCThread	virtual bool ThreadTimer()	线程定时器，设定时间到后系统会自动调用这个函数，用户可以重载这个函数实现其他功能。（建议这个函数不能太臃肿）
	virtual void ThreadMain();	这个函数数一个线程主函数，相当于 c 中的 Main 函数，用户可以通过重载这个函数实现新的消息循环，或者其他功能！（建议不要重载这个函数，在系统消息循环的基础上实现自己的功能）
	virtual bool ThreadProc(OSCTask*STaskP,uint32 mParam,WPARAM wParam,LPARAM lParam);	线程的过程函数，当线程消息队列中有消息，系统框架会自动调用这个函数，并把分解后的消息作为参数传递给他。用户只需要重载这个函数，添加各种消息的响应代码，或者自己定义新的消息。但如果 ThreadMain 被重载这个函数就不会被系统框架调用了！ 【in】Param1：指向线程的指针 【in】Param2：消息类型 【in】Param3：消息第一个辅助参数 【in】Param4：消息第二个辅助参数 返回值：消息被处理 TRUE，消息没被

		处理 FALSE
OSCGuradThread	bool IntMsgSendToTask(OSMsg *PMsg);	中断调用这个函数向 OSCGuradThread 线程传输消息。OSCGuradThread 会根据具体消息进行转发或者处理（需要 OSCGuradThread 处理的消息 PMsg –> TaskPtr 一定要等于 OSCGuradThread 的 ID，需要转发的消息 PMsg –> TaskPtr 要等于接收消息的线程 ID） 【in】Param1：要传送的消息 返回值：成功：TURE、失败：FALSE
OSMsg	OSMsg(OSCTask*STaskP = NULL,uint32 TMsg = 0,WPARAM wp = 0,LPARAM lp = 0)	系统消息构造函数 【in】Param1：线程 ID 【in】Param2：消息类型 【in】Param3：消息第一个辅助参数 【in】Param4：消息第二个辅助参数

(四) Worker1.0 移植

系统移植必须实现的数据类型：

系统移植必须实现的数据类型	数据位长度
uint8	无符号 8 位整型变量
int8	有符号 8 位整型变量
uint16	无符号 16 位整型变量
int16	有符号 16 位整型变量
uint32	无符号 32 位整型变量
int32	有符号 32 位整型变量
fp32	单精度浮点数（32 位长度）
fp64	双精度浮点数（64 位长度）

系统移植必须实现的宏定义：

系统移植必须实现的宏定义	宏的功能
OS_TASK_SW()	任务切换：给定指向两个任务结构体的指针 OSTaskCurPtr、OSTaskNextPtr。宏将当前 CPU 状态保存在 OSTaskCurPtr 指向的任务栈中，并把 OSTaskNextPtr 指向任务的 CPU 状态恢复到 CPU 寄存器！同时将 OSTaskCurPtr 指向 OSTaskNextPtr 指向的结构体
OS_SYS_STARTUP()	开启多任务：给定一个任务结构体地址 OSTaskNextPtr，将 OSTaskNextPtr 指向的任务状态从任务栈中恢复到 CPU 寄存器，并用 OSTaskCurPtr 指向这个结构体
OS_EN_IRQ()	开中断：系统中断使能

OS_DIS_IRQ()	关中断：关系统中断
OS_ENTER_CRITICAL()	关中断：关系统中断并且 OSEnterSum++
OS_EXIT_CRITICAL()	开中断：如果 OSEnterSum—等于零，就开启系统中断
Fpclk	VPB 时钟频率

系统移植必须实现的函数：

系统移植必须实现的函数	函数功能
extern "C" void MovTargetInit()	进入主函数后的目标板初始化：可以进行系统时钟、VIC 等等目标板的初始化。也可以在启动系统前，由用户自己编写的代码另外实现，这个函数可以不做任何事，但必须定义
extern "C" OS_STK *MovTaskStkInit (void (*Task)(void *Pd), void *Pdata, OS_STK *Ptop, int PSR);	任务栈初始化 【in】Param1：任务主函数地址 【in】Param2：传给任务主函数的参数 【in】Param3：分配给任务的堆栈（保证足够的大） 【in】Param4：为 CPSR 内容一部分，保证任务在启动时，要开系统中断 返回值：经过初始化后的堆栈地址
Extern "C" void MovIntRegister(void(*IntServe)(void),int8 Vector,int8 IntChannel);	中断注册函数 【in】Param1：中断服务程序地址 【in】Param2：中断向量 【in】Param3：中断通道
extern "C" void MovTimerInit();	定时器启动函数 启动定时器，开定时器中断，保证系统时间模块正常运行

(五) Worker1.0 例程

创建一个线程：

```
#ifndef __CTEST_HPP
#define __CTEST_HPP
class CTest :public OSCThread{
    protected:
    virtual bool ThreadProc(OSCTask*STaskP,uint32 mParam,WPARAM wParam,LPARAM lParam);
    public:
    CTest();
    static CTest *ThreadID;
};
#endif
```

类的派生方法：

从 OSCThread 中派生出自己的线程类，并重载函数 virtual bool ThreadProc(OSCTask*STaskP,uint32 mParam,WPARAM wParam,LPARAM lParam);这样就可以在线程过程函数中添加消息响应代码，或者添加新的消息驱动程序的运行。

最好添加这样一个静态变量 static CTest *ThreadID; 并且在构造函数中把线程的 ID 付给这个变量，这样可以方便的实现任务的通信。

重载函数的编写：

```
#include "config.hpp"
#include "CTest.hpp"
CTest * CTest::ThreadID = NULL;
 CTest::CTest():OSCThread(0)
{
      ThreadID = this;//
 }
 boolCTest::ThreadProc(OSCTask*STaskP,uint32 mParam,WPARAM wParam,LPARAM   lParam)
 {
      static   uint32     Cnt = 0;
      OSMsg     Msg;
      switch(mParam)
      {
            case TM_CREATE:
                        OSCTimer::TimeServerRegister(this,5);
                        //OSCMutex::MutexID -> MutexGetSwap(0);
                        //IOSET = LED2;
                        //OSCMutex::MutexID -> MutexGetSwap(0);
                        //OSCSem::SemID -> SemGetSwap(0);
                        //OSCSem::SemID -> SemGetSwap(0);
                        //OSCSem::SemID -> SemGetSwap(0);
                        //OSCSem::SemID -> SemGetSwap(0);
                        break;
            case TM_THREADONTIME:
                        if(Cnt % 2){
                              IOCLR = LED1;
                        }else {
                              IOSET = LED1;
                        }
                        Cnt ++;
                        //OSCMutex::MutexID -> MutexGetSwap(0);
                        //thisSynMsgWait(&Msg);
                        break;
            case ONLED4: IOCLR = LED4 ; break;
            case OFFLED4: IOSET = LED4 ; break;
            case ONLED5: IOCLR = LED5 ; break;
            case OFFLED5: IOSET = LED5 ; break;
```

```
            default:
                return      OSCThread::ThreadProc(STaskP,mParam,wParam,lParam);
        }
        return   TRUE;
 }
 //END
```

第二个类的实现：

```
#include "config.hpp"
CTest2 *CTest2::ThreadID = NULL;
 CTest2::CTest2():OSCThread(3)
 {
        ThreadID = this;//
 }

  bool      CTest2::ThreadProc(OSCTask*STaskP,uint32 mParam,WPARAM wParam,LPARAM    lParam)
 {
        OSMsg   Msg;
        static   uint32     Cnt = 0;
        switch(mParam)
        {
            case TM_CREATE:
                        //Msg.Msg = TM_CREATE;
                        //MsgSend(&Msg);
                        OSCTimer::TimeServerRegister(this,5);//注册应用定时器
                        //CTest::ThreadID -> MsgPost(&Msg);
                        //OSCMutex::MutexID -> MutexGetSwap(0);
                        //OSCSem::SemID -> SemPutMedi();
                        //OSCSem::SemID -> SemFlushMedi();
                        break;
                case   TM_THREADONTIME:
                        //添加响应应用定时器消息的代码
                        IOCLR = LED8;
                        if(Cnt % 2){//给 CTest  发送点亮 LED4 的消息
                            Msg.MsgID = ONLED4;
                            CTest::ThreadID -> AsyMsgSend(&Msg);
                    }else {//给 CTest  发送熄灭 LED4 的消息
                        Msg.MsgID = OFFLED4;
                            CTest::ThreadID -> AsyMsgSend(&Msg);
                    }
                    Cnt ++;

                        break;
                default:
                    return      OSCThread::ThreadProc(STaskP,mParam,wParam,lParam);
```

```
            }
        return    TRUE;
 }
 //END
```

主函数文件：

```
#include "config.hpp"
extern "C"    int main (void)
{
    // 用户在下面添加自己的程序代码
    OSCTask::SysInit();
    OSCGuardThread     GuradThread;
    OSCIdlThread    IdlTask;
    CTest           TestApp;
    CTest2          Test2App;
    OSCTask::StartUp();
    while(1);
    return 0;
}
```

系统在启动的时候先会进行各种模式下堆栈的初始化，然后初始化系统时钟 PLL，这些工作都完成后就会进入 main 函数初始化内核。

OSCTask::SysInit();函数会首先初始化系统全局变量，如锁调度器标识、多任务是否启动标识、中断锁标识。还有的就是进行系统各种队列的初始化，准备接收将要入列的任务。

上面的工作做完之后就可以创建任务了。OSCGuardThread 、OSCIdlThread 这两个线程是必须创建的，他们是系统必须的线程。OSCGuardThread 线程相当重要，是系统运行的必要条件，是一个优先级最高的任务。他为系统提供各种服务，例如转发中断消息，给每个线程提供时间服务 ••••••

OSCIdlThread 这个线程是系统在空闲的时候运行的任务，优先级最低。

完成这两个线程的创建之后就可以创建用户自己定义的线程了。给出的例子中就创建了两个线程。

完成这些工作之后不要忘记了，调用系统启动函数 OSCTask::StartUp();这个函数调用后，系统才真正进入多任务的状态。这个函数是不会返回的。因此 while(1) ;return 0;这两条语句不会被执行，这样主函数也不会返回，所以才可以通过定义 CTest2 *CTest2::ThreadID = NULL;

这样一个静态的类成员变量实现任务间通信。