本集视频正在制作中，敬请期待…

第四十一集

uC/OS-II 任务同步与通信的理论与应用讲解

§1.31 uC/OS-II同步与通信

1.31.1 信号量和互斥型信号量

通过前面的学习，我们知道嵌入式系统中的各个任务是为同一个大的任务服务的子任务，它们不可避免地要共同使用一些共享资源，并且在处理一些需要多个任务共同协助来完成的工作时，还需要相互的支持和限制。因此，对于一个完善的多任务操作系统来说，系统必须具有完备的同步和通信机制。为了实现各个任务之间的合作和无冲突的运行，在各任务之间必须建立一些制约关系。其中一种制约关系叫做直接制约关系，另一种制约关系叫做间接制约关系。直接制约关系源于任务之间的合作。例如，有任务 A 和任务 B 两个任务，它们需要通过访问同一个数据缓冲区合作完成一项任务，任务 A 负责向缓冲区写入数据，任务 B 负责从缓冲区读取数据。显然，当任务 A 还没有向缓冲区写入数据时（缓冲区为空时），任务 B 因不能从缓冲区得到有效数据而应该处于等待状态；只有等任务 A 向缓冲区写入了数据之后，才应该通知任务 B 去取数据。相反那，当缓冲区的数据还未被任务 B 读取时（缓冲区为满时），任务 A 就不能向缓冲区写入新的数据而应该处于等待状态；只有等任务 B 自缓冲区读取数据后，才应该通知任务 A 写入数据。显然，如果这两个任务不能如此协调工作，将势必造成严重的后果。间接制约关系源于对资源的共享。例如，任务 A 和任务 B 共享一台打印机，如果系统已经把打印机分配给了任务 A，则任务 B 因不能获得打印机的使用权而应该处于等待状态；只有当任务 A 把打印机释放后，系统才能环形任务 B 使其获得打印机的使用权。如果这两个任务不这样做，那么也会造成极大的混乱。

由上可知，在多任务合作工作的过程中，操作系统应该解决两个问题：一是各任务间应该具有一种互斥关系，即对于某个共享资源，如果一个任务正在使用，则其他任务只能等待，等到该任务释放资源后，等待的任务之一才能使用它；二是相关的任务在执行上要有先后次序，一个任务要等其他伙伴发来通知，或建立了某个条件后才能继续执行，否则只能等待。任务之间的这话总制约性的合作运行机制叫做任务间的同步。uC/OS-II 使用信号量、消息邮箱和消息队列这些中间环节来实现任务之间的通信，为了方便起见，这些中间环节都统一被称作“事件”。下面我们给出两个任务通过事件进行通信的示意图，如图 1.301 所示。

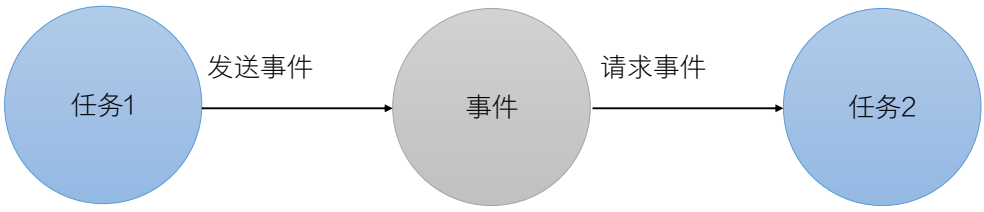

图 1.301 两个任务在使用事件进行通信的示意图

在该图中我们可以看出，任务 1 是发信方，任务 2 是收信方。作为发信方，任务 1 的责任是把信息发送到事件上，这项操作叫做发送事件。作为收信方，任务 2 的责任是通过读事件操作对

事件进行查询：如果有信息，那么任务 2 就会读取信息；如果没有信号，那么任务 2 就会继续等待，通常我们把读事件操作叫做请求事件。在 uC/OS Ⅱ中，我们把任务发送事件、请求事件以及其他对事件的操作全都定义成为全局函数，以供应用程序的所有任务来调用。

信号量就是一类事件，使用信号量的最初目的，是为了给共享资源设立一个标志，该标志表示该共享资源被占用情况。这样，当一个任务在访问共享资源之前，就可以先对这个标志进行查询，从而在了解资源被占用的情况之后，再来决定自己的行为。观察一下人们日常生活中常用的一种共享资源：共用电话亭的使用规则，就会发现这种规则很适合在协调某种资源用户关系时使用。如果一个电话亭只允许一个人进去打电话，俺么电话亭的门上就应该有一个可以变换两种颜色的牌子（例如，用红色表示有人，用绿色表示无人）。当有人进去时，牌子会变成红色；出来时，牌子又会变成绿色。这样来打电话的人就可以根据牌子上的颜色是绿色，那么他就可以进去打电话；如果是红色，那么他只好等待；如果又陆续来了很多人，那么就要排队等待。显然，电话亭上的这个牌子就是一个表示电话亭是否被占用的标志。由于折后再难过标志特别像交叉路口上的交通信号灯，所以人们最初给这种标志起的名称就是信号灯，后来因为它含有了量的概念，所以又叫做信号量。显然，对于上面介绍的红绿标志来说，这是一个二值信号量，而且由于它可以实现共享资源的独占式占用，所以又被叫做互斥型信号量。如果电话亭可以允许多人打电话，那么电话亭门前就不应该是那种只有红色和绿色两种颜色状态的牌子，而应该是一个计数器，该计数器在每进去一个人时会自动减一，而每出去一个人时会自动加1.如果其初值按电话亭的最大容量来设置，那么来人只要见到计数器的值大于 0，就可以进去打电话；否则只好等待。这种技术式的信号叫做信号量。下面我们给出两个任务在使用互斥型信号量进行通信，从而可使这两个任务无冲突地访问一个共享资源的示意图，如图 1.302 和图 1.303 所示。

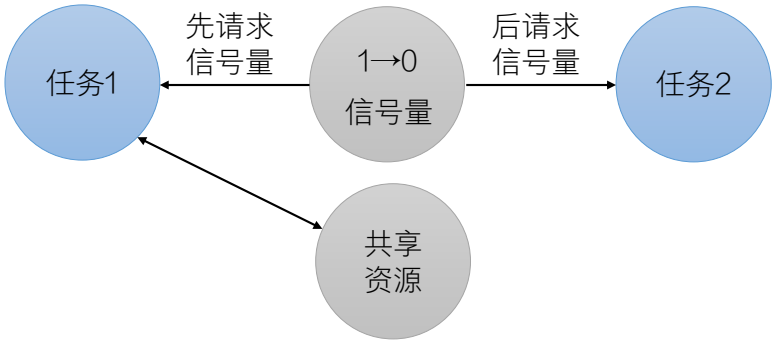

图 1.302 任务 1 先获得信号量并使用共享资源，而任务 2 只能等待信号量

从该图中我们可以看出，任务 1 在访问共享资源之前先进行请求信号量的操作，当任务 1 发现信号量的标志为 1 时，它一方面把信号量的标志由 1 改为 0，另一方面进行共享资源的访问。如果任务 2 在任务 1 已经获得信号之后来请求信号量，那么由于它获得的标志值是 0，所以任务 2 就只有等待而不能访问共享资源了，由此我们可以得出，这种做法可以有效地防止两个任务同时访问同一个共享资源所造成的冲突。那么任务 2 何时可以访问共享资源呢？如图 1.303 所示。

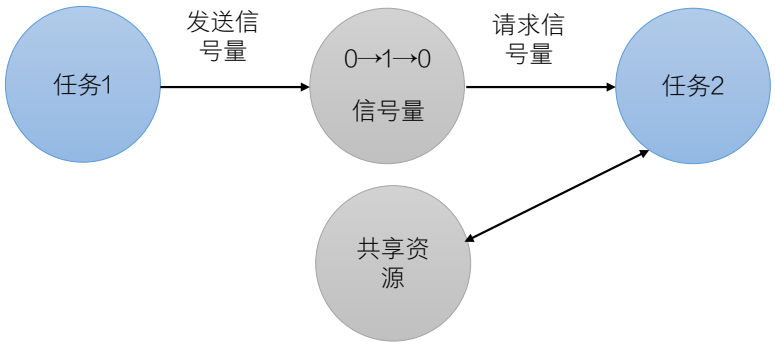

图 1.303 任务 1 释放信号后，任务 2 方可获得信号量并使用共享资源

从该图中我们可以看出，任务 1 使用完共享资源之后，由任务 1 向信号量发信号使信号量标志的值由 0 再变为 1 时，任务 2 就有机会访问共享资源了。与任务 1 一样，任务 2 一旦获得了共享资源的访问权，那么在访问共享资源之前一定要把信号量标志的值由 1 变为 0。

最后我们同样总结给出信号量的函数如表1.72所示，这些函数我们可以在os_sem.c文件中找到。互斥信号量的函数如表 1.73 所示。这些函数我们可以在 os_mutex.c 文件中找到。

表 1.72 信号量的函数

函数	描述
OSSemAccept()	检查和此信号量相关的资源是否可用
OSSemCreate()	创建一个信号量
OSSemDel()	删除一个信号量
OSSemPend()	等待一个信号量
OSSemPendAbord()	取消等待
OSSemPost()	释放一个信号量
OSSemQuery()	查询一个信号量
OSSemSet()	设置一个信号量的值

表 1.73 互斥信号量的函数

函数	描述
OSMutexAccept()	检查互斥信号量相关的资源是否可用
OSMutexCreate()	创建一个互斥信号量
OSMutexDel()	删除一个互斥信号量
OSMutexPend()	等待一个互斥信号量
OSMutexPost()	释放一个互斥信号量
OSMutexQuery()	查询一个互斥信号量
OSMutex_RdyAtPrio ()	使一个任务恢复到指定的优先级别

1.31.2 信号量和互斥型信号量的应用

(1) 功能概述

介绍完了信号量和互斥型信号量，接下来我们再来看看信号量和互斥型信号量的应用，该实验主要设计 6 个任务，这 6 个任务的名称、任务堆栈大小，以及任务功能，如表 1.74 所示。

表 1.74 uC/OS Ⅱ的任务列表

任务名称	任务栈大小	功能
InitTask()	2048 字节	初始化其他任务，然后自我删除
WriterTask1()	2048 字节	与 WriterTask2()竞争共享内存的使用权,得到使用权后向共享内存写入字符串”Writer1 get the semaphore”
WriterTask2()	2048 字节	与 WriterTask1()竞争共享内存的使用权,得到使用权后向共享内存写入字符串”Writer2 get the semaphore”
InputTask()	2048 字节	查询键盘输入,当有键盘输入事件发生时,向 OutputTask()发送信号量
OutputTask()	2048 字节	等待 InputTask()的信号量,取得信号量后输出”Output Task got the semaphore; Key Down n times”
PrintTask()	2048 字节	每 5s 打印一次 WriterTask1()与 WriterTask2()的竞争信息

(2) 硬件框架

讲完了功能概述，接下来我们就来讲解硬件框架，该实验的硬件框架，如图 1.304 所示。

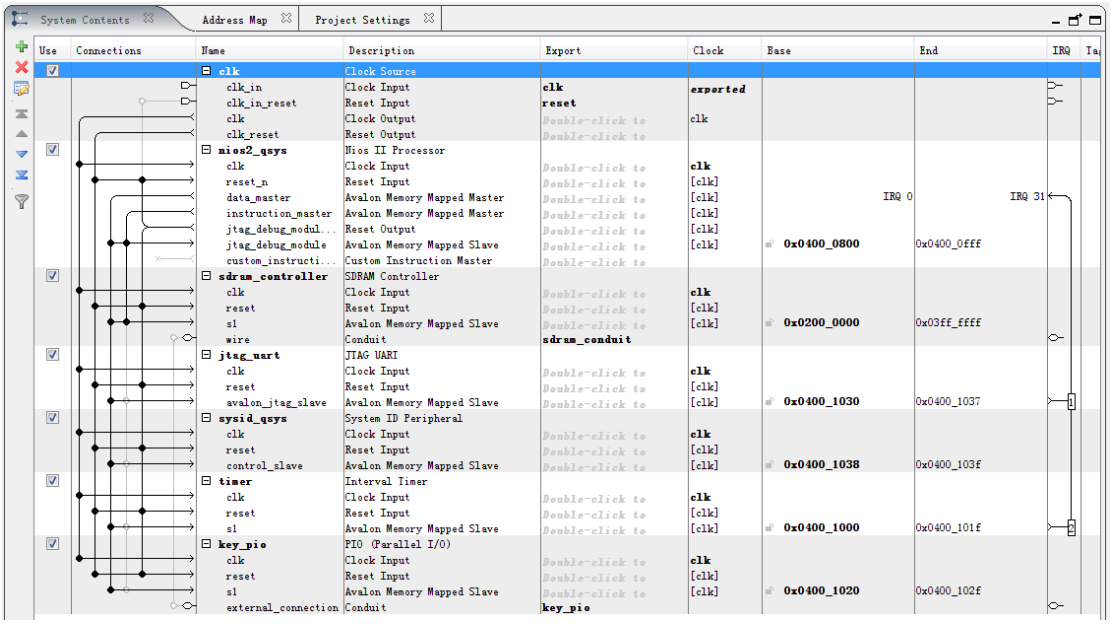

图 1.304 uC/OS II 的硬件框架图

从该图中我们可以看出,我们这里的uC/OS II的硬件框架在我们建立的第一个UC/OS II的硬件框架基础上多添加了一个 PIO,该 PIO 主要是给我们的按键使用的,我们这里就爱那个它配置成了 8 位的输入端口。

(3) 软件工程

讲完了硬件框架，接下来我们就来讲解软件工程，该实验的软件工程代码，如代码 1.91 所示。

代码 1.91 Qsys_Ucosii_Semaphore.c 代码

```
1  //--------------------------------------------------------------------------------
2  //-- 文件名    :  Qsys_Ucosii_Semaphore.c
3  //-- 描述      :  利用信号量,实现任务同步和资源共享
4  //-- 修订历史  :  2014-1-1
5  //-- 作者      :  Zircon Opto-Electronic Technology CO.,Ltd.
```

```
6  //-------------------------------------------------------------------------
7  #include <stdio.h>
8  #include <string.h>
9  #include <unistd.h>
10 #include "includes.h"   /* 声明 MicroC/OS-II 头文件 */
11
12 #define TASK_STACKSIZE 2048 /* 定义任务栈大小 */
13 /* 定义各任务任务栈 */
14 OS_STK initialize_task_stk[TASK_STACKSIZE];
15 OS_STK writer_task1_stk[TASK_STACKSIZE];
16 OS_STK writer_task2_stk[TASK_STACKSIZE];
17 OS_STK input_task_stk[TASK_STACKSIZE];
18 OS_STK output_task_stk[TASK_STACKSIZE];
19 OS_STK print_task_stk[TASK_STACKSIZE];
20
21 /* 分配各任务优先级 */
22 #define INITIALIZE_TASK_PRIORITY   5
23 #define PRINT_TASK_PRIORITY        6
24 #define WRITER_TASK1_PRIORITY      7
25 #define WRITER_TASK2_PRIORITY      8
26 #define INPUT_TASK_PRIORITY        9
27 #define OUTPUT_TASK_PRIORITY       10
28
29 /* 定义信号量 */
30 OS_EVENT *shared_resource_sem;       /*用作共享资源访问的信号量*/
31 OS_EVENT *key_down_sem;              /*用作任务同步的信号量*/
32
33 /* 定义全局变量*/
34 INT32U writer_task1_got_sem = 0;     /*WriterTask1 获取信号量次数记录*/
35 INT32U writer_task2_got_sem = 0;     /*WriterTask2 获取信号量次数记录*/
36 INT32U key_down_num = 0;             /*按键次数记录*/
37 char shared_memory[40];              /*共享内存单元*/
38
39 /* 外部函数申明*/
40 extern INT8U KeyPoll(void);          /*键状态查询函数,有按键按下返回 1*/
41
42 /* 局部函数声明 */
43 int initOSDataStructs(void);         /* 初始化信号量*/
44 int initCreateTasks(void);           /* 初始化任务    */
45
46 /* 每 5 秒打印一次 WriterTask1()和 WriterTask2()的竞争信息 */
47 void PrintTask(void* pdata)
48 {
49   while (1)
```

```
    {
      OSTimeDlyHMSM(0, 0, 5, 0);
      printf("------------------------------------------------------------\n");
      printf("From MicroC/OS-II Running on Nios II. Here is the status:\n");
      printf("The shared memory is written by: %s\n",&shared_memory[0]);
      printf("The Number of times writer_task1 acquired the semaphore %lu\n",
           writer_task1_got_sem);
      printf("The Number of times writer_task2 acquired the semaphore %lu\n",
           writer_task2_got_sem);
      printf("------------------------------------------------------------\n");
      printf("\n");
    }
}

/* 父任务：初始化其他子任务,然后自我删除 */
void  initialize_task(void* pdata)
{
   /* 初始化信号量 */
   initOSDataStructs();
   /* 创建个子任务 */
   initCreateTasks();
   /* 完成用户功能后自我删除 */
   OSTaskDel(OS_PRIO_SELF);

   while(1);
}

/* Writer 任务 1：当获得共享内存使用权后,向共享内存写入字符*/
void WriterTask1(void* pdata)
{
   INT8U return_code = OS_NO_ERR;  /* 系统调用的返回状态  */
   while(1)
   {
      OSSemPend(shared_resource_sem, 0, &return_code);
      strcpy(&shared_memory[0],"Writer1 get the semaphore");
      writer_task1_got_sem++;
      OSSemPost(shared_resource_sem);
      OSTimeDlyHMSM(0, 0, 0, 100); /* 延时 100ms */
   }
}

/* Writer 任务 2：当获得共享内存使用权后,向共享内存写入字符*/
void WriterTask2(void* pdata)
{
```

```
94    INT8U return_code = OS_NO_ERR;  /* 系统调用的返回状态  */
95    while(1)
96    {
97       OSSemPend(shared_resource_sem, 0, &return_code);
98       strcpy(&shared_memory[0],"Writer2 get the semaphore");
99       writer_task2_got_sem++;
100      OSSemPost(shared_resource_sem);
101      OSTimeDlyHMSM(0, 0, 0, 130); /* 延时 130ms */
102   }
103}
104
105/* 键盘输入任务:当有按键下时通知 output 任务*/
106void InputTask(void* pdata)
107{
108   OS_CPU_SR cpu_sr;
109   while(1)
110   {
111      if(KeyPoll() == 1)
112      {  /*利用临界区来保护共享变量*/
113         OS_ENTER_CRITICAL();
114         key_down_num++;
115         OS_EXIT_CRITICAL();
116         OSSemPost(key_down_sem);
117      }
118      else
119         OSTimeDlyHMSM(0, 0, 0, 50); /* 延时 50ms */
120   }
121}
122
123/* 输出任务,当获知键盘事件发生时,输出按键次数*/
124void OutputTask(void* pdata)
125{
126   INT8U return_code = OS_NO_ERR;  /* 系统调用的返回状态  */
127   while(1)
128   {
129      OSSemPend(key_down_sem, 0, &return_code);
130      printf("\n");
131      printf("Output Task got the semaphore\n");
132      printf("Key down %ld times. \n", key_down_num);
133   }
134}
135
136/* 初始化信号量*/
137int initOSDataStructs(void)
```

```
{  /* 用作二值信号量时,信号量初始化为 1 */
   shared_resource_sem = OSSemCreate(1);
   /* 用作事件发生标志时,信号量初始化为 0 */
   key_down_sem = OSSemCreate(0);
   return 0;
}

/* 初始化各子任务*/
int initCreateTasks(void)
{
   /*  创建打印任务  */
   OSTaskCreateExt(PrintTask,
                        NULL,
                        &print_task_stk[TASK_STACKSIZE],
                        PRINT_TASK_PRIORITY,
                        PRINT_TASK_PRIORITY,
                        print_task_stk,
                        TASK_STACKSIZE,
                        NULL,
                        0);

   /*  创建 Writer1 任务  */
   OSTaskCreateExt(WriterTask1,
                        NULL,
                        &writer_task1_stk[TASK_STACKSIZE],
                        WRITER_TASK1_PRIORITY,
                        WRITER_TASK1_PRIORITY,
                        writer_task1_stk,
                        TASK_STACKSIZE,
                        NULL,
                        0);

   /*  创建 Writer2 任务  */
   OSTaskCreateExt(WriterTask2,
                        NULL,
                        &writer_task2_stk[TASK_STACKSIZE],
                        WRITER_TASK2_PRIORITY,
                        WRITER_TASK2_PRIORITY,
                        writer_task2_stk,
                        TASK_STACKSIZE,
                        NULL,
                        0);

   /*  创建键盘输入任务  */
```

```
182     OSTaskCreateExt(InputTask,
183                     NULL,
184                     &input_task_stk[TASK_STACKSIZE],
185                     INPUT_TASK_PRIORITY,
186                     INPUT_TASK_PRIORITY,
187                     input_task_stk,
188                     TASK_STACKSIZE,
189                     NULL,
190                     0);
191
192     /*  创建输出任务  */
193     OSTaskCreateExt(OutputTask,
194                     NULL,
195                     &output_task_stk[TASK_STACKSIZE],
196                     OUTPUT_TASK_PRIORITY,
197                     OUTPUT_TASK_PRIORITY,
198                     output_task_stk,
199                     TASK_STACKSIZE,
200                     NULL,
201                     0);
202
203     return 0;
204 }
205
206 //--------------------------------------------------------------------------
207 //-- 名称     :  main()
208 //-- 功能     :  程序入口
209 //-- 输入参数  :  无
    //-- 输出参数  :  无
210 //--------------------------------------------------------------------------
211 int main (int argc, char* argv[], char* envp[])
212 {
213     /*  创建父任务  */
214     OSTaskCreateExt(initialize_task,
215                     NULL,
216                     &initialize_task_stk[TASK_STACKSIZE],
217                     INITIALIZE_TASK_PRIORITY,
218                     INITIALIZE_TASK_PRIORITY,
219                     initialize_task_stk,
220                     TASK_STACKSIZE,
221                     NULL,
222                     0);
223     OSStart();  /*  启动 OS  */
224
```

```
225     return 0;
226 }
227 /*  This is the end of this file  */
```

下面我们就来给大家讲解一下该程序中的关键点。信号量的用途是什么？信号量（Semaphore）是20世纪60年代中期由Edgser Dijkstra发明的，常用于：1、对一个共享资源（相互排斥）访问的控制；2、表示一个事件的发生；3、让两个任务同步。在本应用中，信号量 shared_resource_sem 用于对一个共享资源（相互排斥）访问的控制，而信号量key_down_sem则用于表示一个事件的发生。

如何操控信号量？对于一个信号量可以进行3个操作：Initialize（初始化或称为Create）、Wait（等待或称Pend）或Signal（发信号或称Post）。一个请求信号量的任务将执行Wait等待。如果该信号量可以使用了（信号量的值大于0），则信号量的值将递减且任务继续进行。如果信号量的值为0，则对该信号量执行Wait操作的任务将被放置在等待列表中。一个任务通过Signal操作来释放信号量。如果没有任何任务等待使用这个信号量，该信号量的值递增。然而，如果有任务在等待这个信号量，则其中的一个将准备运行，且该信号量的值不会递增。对一个信号初始化时，必须提供该信号量的初始值。初始值可以有以下3种：

- 0：当信号量用于表示一个事件发生时，如程序段：key_down_sem = OSSemCreate(0);
- 1：当信号量用于对一个共享资源访问的控制时，如程序段：shared_resource_sem = OSSemCreate(1)；
- n：当信号量用于表示允许任务访问n个相同的资源时。

如果有多个任务在等待信号量时，信号量应该分配给谁？一般来说，信号量对多个等待任务的分配采用两种策略：一是分配给等待任务列表中优先级最高的任务；二是给请求信号量的第一个任务（FIFO）。uC/OS Ⅱ采用的是第一种分配策略。

还有其他系统服务能完成信号量的功能吗？在uC/OS Ⅱ中，除了信号量外，还有多种方法可以保护任务之间的共享数据和提供任务之间的通信。

- 利用宏OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()来关闭中断和打开中断。当两个任务或一个任务和一个中断服务子程序共享某些数据时，可以采用这种方法，这种方法的优点是不会增加调用信号量的系统开销，缺点是降低系统对异步时间的响应能力。例如，考虑由InputTask和PrintTask共享的INT32U型变量key_down_num，当它正在进行加1的操作但还未完成加1操作时被中断打断，uC/OS-Ⅱ开始进行任务调度。如果此时PrintTask任务已处于就绪状态，那么PrintTask任务就会获得CPU的使用权，输出尚未更新完毕的key_down_num的值。为了避免出现这种错误，上述代码中采用了开关中断的方法来保护共享变量。
- 利用函数OSSchedLock()和OSSchekUnlock()对uC/OS-Ⅱ中的任务调度函数上锁和开锁。请读者尝试用该种方法来对共享变量key_down_num进行保护，并分析此种方法的优缺点。
- 邮箱和消息队列。邮箱和消息队列通常用于并发任务间的通信，将在下面进行详述。

这里我们需要注意的是，我们最好将信号量的初始化工作放在 uC/OS Ⅱ启动前，这样可以防止一些莫名奇妙的错误，是一种比较好的编程习惯。

(4) 板级调试

讲完了软件工程，接下来我们就将该实验下载至我们的A4开发板进行验证，首先我们需要在 Quartus Ⅱ 软件中将 Qsys_Ucosii_Semaphore.sof 下载至我们的 A4 开发板，Qsys_Ucosii_Semaphore.sof 下载完成后，我们还需要在 Eclipse 软件中将 Qsys_Ucosii_Semaphore.elf 文件下载至我们的 A4 开发板，Qsys_Ucosii_Semaphore.elf 下载完成以后，我们的 C 程序将会执行在我们的 A4 开发板上，此时，我们可以在 Eclipse 软件的控制台中看到我们的打印信息，如图 1.305 所示。

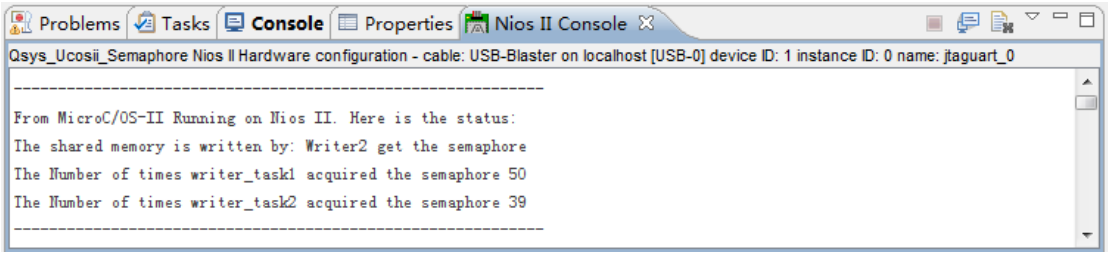

图 1.305 信号量和互斥信号量的控制台打印信息图

这时，我们按下 A4 开发板上的任意一个按键，我们的控制台则会弹出如图 1.306 所示内容。

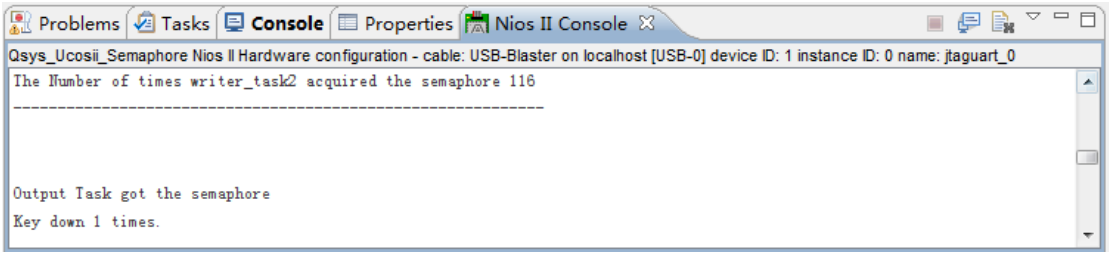

图 1.306 信号量和互斥信号量函数的板级调试图

1.31.3 消息邮箱和消息队列

在多任务操作系统中，常常需要在任务与任务之间通过传递一个数据(这种数据叫做“消息”)的方式来进行通信。为了达到这个目的，可以在内存中创建一个存储空间作为该数据的缓冲区。如果把这个缓冲区叫做消息缓冲区，那么在任务间传递数据(消息)的一个最简单的方法就是传递消息缓冲区的指针。因此，用来传递消息缓冲区指针的数据结构就叫做消息邮箱。下面我们给出两个任务使用消息邮箱进行通信的示意图，如图 1.307 所示。

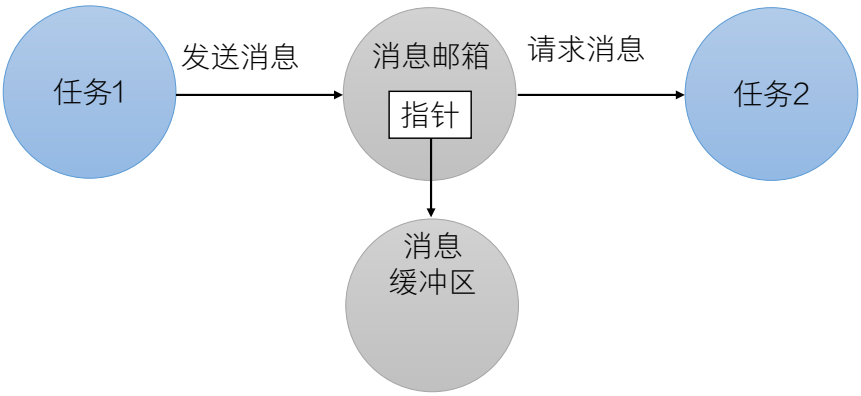

图 1.307 两个任务在使用消息邮箱进行通信的示意图

从该图中我们可以看出，任务 1 在向消息邮箱发送消息，任务 2 在从消息邮箱读取消息。读取消息也叫做请求消息。上面我们谈到的消息邮箱不仅可以用来传递一个消息，而且也可以定义一个指针数组。让数组的每个元素都存放一个消息缓冲区指针，那么任务就可以通过传递这个指针数组指针的方法来传递多个消息了。这种可以传递多个消息的数据结构叫做消息队列。下面我们给出两个任务使用消息队列进行通信的示意图，如图 1.308 所示。

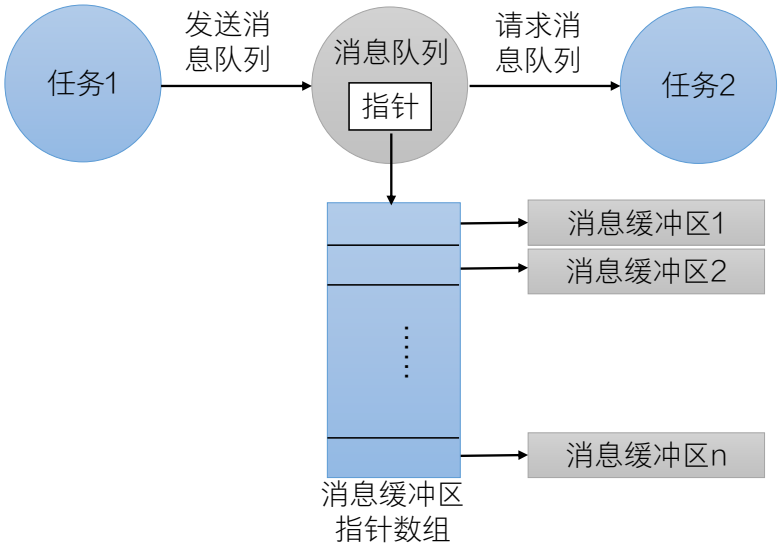

图 1.308 两个任务在使用消息队列进行通信的示意图

从该图中我们可以看出，任务1向消息队列发送消息缓冲区指针数据的指针，这个操作叫做发送消息队列；任务2在从消息队列读取消息缓冲区指针数组的指针，这个操作叫做请求消息队列。最后我们同样总结给出消息邮箱的函数如表1.75所示，这些函数我们可以在os_mbox.c文件中找到。

表 1.75 消息邮箱的函数

函数	描述
OSMboxAccept()	检查邮箱中是否有消息
OSMboxCreate()	创建一个邮箱
OSMboxDel()	删除一个邮箱
OSMboxPend()	等待一个邮箱

OSMboxPendAbort()	取消等待
OSMboxPost()	释放一个邮箱
OSMboxPostOpt()	发送一个消息给一个任务或多个任务
OSMboxQuery()	查询一个邮箱

消息队列的函数如表 1.76 所示。这些函数我们可以在 os_q.c 文件中找到。

表 1.76 消息队列的函数

函数	描述
OSQAccept()	检查一个队列是否有消息可用
OSQCreate()	创建一个消息队列
OSQDel()	删除一个消息队列
OSQFlush()	清除消息队列中的消息
OSQPend()	等待一个消息队列
OSQPendAbort()	取消等待
OSQPost()	将一个消息送到队列中
OSQPostFront()	将一则消息放到一个消息队列中
OSQPostOpt()	发送一则消息给队列
OSQQuery()	查询一个消息队列
OS_QInit()	初始化消息队列模块

1.31.4 消息邮箱和消息队列的应用

(1) 功能概述

介绍完了消息邮箱和消息队列,接下来我们再来看看消息邮箱和消息队列的应用,该实验主要设计 8 个任务，这 8 个任务的名称、任务堆栈大小，以及任务功能，如表 1.77 所示。

表 1.77 uC/OS Ⅱ的任务列表

任务名称	任务栈大小	功能
InitTask()	2048 字节	初始化其他任务及相关数据结构,然后自行删除
MsgSender()	2048 字节	MsgSender()以一定的速率向消息队列发送消息
MsgReceiver1()	2048 字节	MsgReceiver1()以一定的速率取消息
MsgReceiver2()	2048 字节	MsgReceiver2()以不同于 MsgReceiver1()的速度取消息
MailSend1()	2048 字节	MailSend1()以一定的速率从 mailbox1 收取邮件,然后向 maibox2 发送邮件
MailSend2()	2048 字节	MailSend2()以一定的速率从 mailbox2 收取邮件,然后向 maibox3 发送邮件
MailSend3()	2048 字节	MailSend3()以一定的速率从 mailbox3 收取邮件,然后向 maibox1 发送邮件
PrintTask()	2048 字节	每 5s 打印一次使用消息队列进行通信的任务状态信息

(2) 硬件框架

讲完了功能概述，接下来我们就来讲解硬件框架，该实验的硬件框架，如图 1.309 所示。

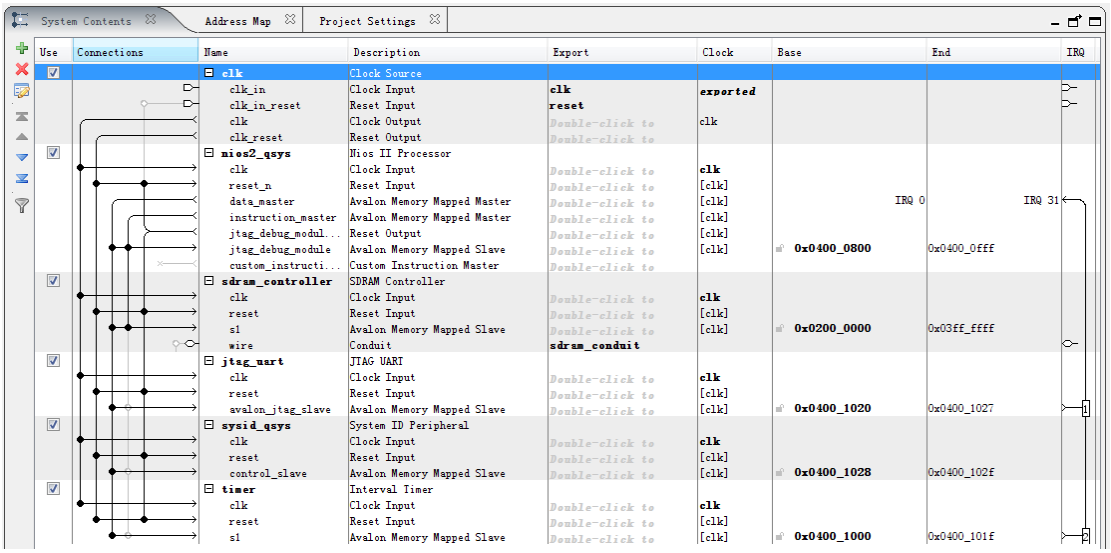

图 1.309 uC/OS II 的硬件框架图

从该图中我们可以看出，我们这里的uC/OS II的硬件框架与我们建立的第一个UC/OS II的硬件框架是一样的。

(3) 软件工程

讲完了硬件框架，接下来我们就来讲解软件工程，该实验的软件工程代码，如代码 1.92 所示。

代码 1.92 Qsys_Ucosii_Mbox_Queue.c 代码

```
1  //--------------------------------------------------------------------------------
2  //-- 文件名    :  Qsys_Ucosii_Mbox_Queue.c
3  //-- 描述      :  利用消息队列和邮箱，实现任务间的通讯。
4  //-- 修订历史  :  2014-1-1
5  //-- 作者      :  Zircon Opto-Electronic Technology CO.,Ltd.
6  //--------------------------------------------------------------------------------
7  #include <stdio.h>
8  #include <unistd.h>
9  #include "includes.h"   /* 声明 MicroC/OS-II 头文件 */
10
11 #define WAIT_FOREVER 0          /* 超时常数，0 表示永远等待 */
12 #define TASK_STACKSIZE 2048     /* 定义任务栈大小          */
13
14 /* 定义各任务任务栈 */
15 OS_STK initialize_task_stk[TASK_STACKSIZE];
16 OS_STK msg_sender_stk[TASK_STACKSIZE];
17 OS_STK msg_receiver1_stk[TASK_STACKSIZE];
18 OS_STK msg_receiver2_stk[TASK_STACKSIZE];
19 OS_STK mail_send1_stk[TASK_STACKSIZE];
20 OS_STK mail_send2_stk[TASK_STACKSIZE];
21 OS_STK mail_send3_stk[TASK_STACKSIZE];
```

```
22  OS_STK print_task_stk[TASK_STACKSIZE];
23
24  /* 分配各任务优先级 */
25  #define INITIALIZE_TASK_PRIORITY   6
26  #define PRINT_TASK_PRIORITY        7
27  #define MSG_SENDER_PRIORITY        8
28  #define MSG_RECEIVER1_PRIORITY     9
29  #define MSG_RECEIVER2_PRIORITY     10
30  #define MAIL_SEND1_PRIORITY        11
31  #define MAIL_SEND2_PRIORITY        12
32  #define MAIL_SEND3_PRIORITY        13
33
34  /* 定义消息队列 */
35  #define MSG_QUEUE_SIZE 30              /* 消息队列大小     */
36  OS_EVENT *msgqueue;                    /* 消息队列事件指针 */
37  void *msgqueueTbl[MSG_QUEUE_SIZE];     /* 队列缓冲区       */
38
39  /* 定义三个邮箱 */
40  OS_EVENT  *mailbox1;
41  OS_EVENT  *mailbox2;
42  OS_EVENT  *mailbox3;
43
44  /* 定义任务状态记录 */
45  INT32U number_of_messages_sent = 0;             /* 发出消息的数量          */
46  INT32U number_of_messages_received_task1 = 0;   /* 接收任务 1 收到消息的数量 */
47  INT32U number_of_messages_received_task2 = 0;   /* 接收任务 2 收到消息的数量 */
48
49  /* 局部函数声明 */
50  int initOSDataStructs(void);   /* 初始化消息队列和邮箱的数据结构 */
51  int initCreateTasks(void);     /* 初始化任务                    */
52
53  /* 每 5 秒打印一次使用消息队列的进行通讯的任务状态信息. */
54  void PrintTask(void* pdata)
55  {
56    while (1)
57    {
58      OSTimeDlyHMSM(0, 0, 5, 0);
59      printf("-------------------------------------------------------------\n");
60      printf("The number of messages sent by the MsgSender:       %lu\n",
61           number_of_messages_sent);
62      printf("The number of messages received by the MsgReceiver1:  %lu\n",
63           number_of_messages_received_task1);
64      printf("The number of messages received by the MsgReceiver2:  %lu\n",
65           number_of_messages_received_task2);
```

```
66        printf("------------------------------------------------------------\n");
67     }
68  }
69
70  /* 消息发送任务： 将消息通过消息队列广播给所有等待消息的任务，当队列满时，延时 1s */
71  void MsgSender(void* pdata)
72  {
73     INT32U  msg = 0;         /* 欲发出的消息        */
74     OS_Q_DATA queue_data;    /* 用于保存从消息队列的事件控制块中获取的数据信息  */
75
76     while (1)
77     {
78        OSQQuery(msgqueue, &queue_data); /* 查询本消息队列的信息    */
79        if(queue_data.OSNMsgs < MSG_QUEUE_SIZE)         /* 查询消息队列是否已满    */
80        {    /* 消息队列未满，将消息(msg)通过消息队列(msgqueue)广播给所有等待消息的任务*/
81           OSQPostOpt(msgqueue, (void *)&msg, OS_POST_OPT_BROADCAST);
82           msg++;                        /* 产生新消息          */
83           number_of_messages_sent++;  /* 记录发出消息的条数  */
84        }
85        else
86        {  /* 消息队列已满，延时 1s */
87           OSTimeDlyHMSM(0, 0, 1, 0);
88        }
89     }
90  }
91
92  /* 消息接收任务 1： 每 300ms 接收一次消息队列的消息*/
93  void MsgReceiver1(void* pdata)
94  {
95     INT8U return_code = OS_NO_ERR;   /* 系统调用的返回状态  */
96     INT32U *msg;                     /* 存储接收到的消息   */
97
98     while (1)
99     {    /* 到 msgqueue 接收消息，如果消息队列为空，则一直等待  */
100       msg = (INT32U *)OSQPend(msgqueue, WAIT_FOREVER, &return_code);
101       number_of_messages_received_task1++;   /* 接到消息，计数器加 1 */
102       OSTimeDlyHMSM(0, 0, 0, 300);           /* 延时 300ms        */
103    }
104 }
105
106 /* 消息接收任务 1： 每 1s 接收一次消息队列的消息*/
107 void MsgReceiver2(void* pdata)
108 {
109    INT8U return_code = OS_NO_ERR;   /* 系统调用的返回状态  */
```

```
110     INT32U *msg;                        /* 存储接收到的消息   */
111
112     while (1)
113     {  /* 到 msgqueue 接收消息，如果消息队列为空，则一直等待  */
114        msg = (INT32U *)OSQPend(msgqueue, WAIT_FOREVER, &return_code);
115        number_of_messages_received_task2++;   /* 接到消息，计数器加 1 */
116        OSTimeDlyHMSM(0, 0, 1, 0);             /* 延时 1s            */
117     }
118 }
119
120 /* 邮件发送任务 1：从 mailbox1 中收取信息，把信息加 1 后发到 mailbox2*/
121 void MailSend1(void* pdata)
122 {
123     INT8U return_code = OS_NO_ERR;   /* 系统调用的返回状态  */
124     INT32U *mbox1_contents;          /* 指向邮件内容的指针  */
125
126     while (1)
127     {  /* 从 mailbox1 接收邮件，如果邮箱为空，则一直等待  */
128        mbox1_contents = (INT32U *)OSMboxPend(mailbox1,WAIT_FOREVER,&return_code);
129
130        /* 输出邮件内容，并把内容加 1  */
131        printf("Task1 received message: %lu in mailbox 1\n",(*mbox1_contents)++);
132        printf("Task1 incremented the message and placed it into mailbox 2\n");
133        printf("\n");
134        OSMboxPost(mailbox2,(void *)mbox1_contents); /* 把邮件发送到 mailbox2  */
135     }
136 }
137
138 /* 邮件发送任务 2：从 mailbox2 中收取信息，把信息加 1 后发到 mailbox3*/
139 void MailSend2(void* pdata)
140 {
141     INT8U return_code = OS_NO_ERR;   /* 系统调用的返回状态  */
142     INT32U *mbox2_contents;          /* 指向邮件内容的指针  */
143
144     while(1)
145     {  /* 从 mailbox1 接收邮件，如果邮箱为空，则一直等待  */
146        mbox2_contents = (INT32U *)OSMboxPend(mailbox2,WAIT_FOREVER,&return_code);
147        /* 输出邮件内容，并把内容加 1  */
148        printf("Task2 received message: %lu in mailbox 2\n", (*mbox2_contents)++);
149        printf("Task2 incremented the message and placed it into mailbox 3\n");
150        printf("\n");
151        /* 把邮件发送到 mailbox3  */
152        OSMboxPost(mailbox3,(void *)mbox2_contents);
153     }
```

```
}

/* 邮件发送任务 3： 从 mailbox3 中收取信息，把信息加 1 后发到 mailbox1*/
void MailSend3(void* pdata)
{
   INT8U return_code = OS_NO_ERR;
   INT32U *mbox3_contents;

   while (1)
   {
      mbox3_contents = (INT32U *)OSMboxPend(mailbox3,WAIT_FOREVER,&return_code);

      printf("Task3 received message: %lu in mailbox 3\n", (*mbox3_contents)++);
      printf("Task3 incremented the message and placed it into mailbox 1\n");
      printf("\n");
      usleep(3000000); /* 延时 3000000us */
      /* 把邮件发送到 mailbox1  */
      OSMboxPost(mailbox1,(void *)mbox3_contents);
   }
}

/* 初始化各子任务 */
void  initialize_task(void* pdata)
{
   INT32U mbox1_contents = 0;
   /* 初始化消息队列和邮箱的数据结构 */
   initOSDataStructs();

   /* 创建个子任务 */
   initCreateTasks();

   /* 向 mailbox1 发送一封邮件*/
   OSMboxPost(mailbox1, (void *)&mbox1_contents);

   OSTaskDel(OS_PRIO_SELF);
   while (1);
}

//--------------------------------------------------------------------------------
//-- 名称     :  main()
//-- 功能     :  程序入口
//-- 输入参数  :  无
//-- 输出参数  :  无
//--------------------------------------------------------------------------------
```

```
198 int main (int argc, char* argv[], char* envp[])
199 {
200   /*  创建父任务  */
201   OSTaskCreateExt(initialize_task,
202                         NULL,
203                         &initialize_task_stk[TASK_STACKSIZE],
204                         INITIALIZE_TASK_PRIORITY,
205                         INITIALIZE_TASK_PRIORITY,
206                         initialize_task_stk,
207                         TASK_STACKSIZE,
208                         NULL,
209                         0);
210   OSStart();  /*  启动 OS  */
211   return 0;
212 }
213 /* This function simply creates the three Mailboxes */
214
215 int initOSDataStructs(void)
216 {  /* 初始化邮箱的数据结构 */
217   mailbox1 = OSMboxCreate((void *)NULL);
218   mailbox2 = OSMboxCreate((void *)NULL);
219   mailbox3 = OSMboxCreate((void *)NULL);
220   /* 初始化消息队列的数据结构 */
221   msgqueue = OSQCreate(&msgqueueTbl[0], MSG_QUEUE_SIZE);
222   return 0;
223 }
224
225 int initCreateTasks(void)
226 {
227   /*  创建打印任务  */
228   OSTaskCreateExt(PrintTask,
229                         NULL,
230                         &print_task_stk[TASK_STACKSIZE],
231                         PRINT_TASK_PRIORITY,
232                         PRINT_TASK_PRIORITY,
233                         print_task_stk,
234                         TASK_STACKSIZE,
235                         NULL,
236                         0);
237
238   /*  创建 MsgSender 任务  */
239   OSTaskCreateExt(MsgSender,
240                         NULL,
241                         &msg_sender_stk[TASK_STACKSIZE],
```

```
                        MSG_SENDER_PRIORITY,
                        MSG_SENDER_PRIORITY,
                        msg_sender_stk,
                        TASK_STACKSIZE,
                        NULL,
                        0);

    /*  创建 MsgReceiver1 任务  */
    OSTaskCreateExt(MsgReceiver1,
                        NULL,
                        &msg_receiver1_stk[TASK_STACKSIZE],
                        MSG_RECEIVER1_PRIORITY,
                        MSG_RECEIVER1_PRIORITY,
                        msg_receiver1_stk,
                        TASK_STACKSIZE,
                        NULL,
                        0);

    /*  创建 MsgReceiver2 任务  */
    OSTaskCreateExt(MsgReceiver2,
                        NULL,
                        &msg_receiver2_stk[TASK_STACKSIZE],
                        MSG_RECEIVER2_PRIORITY,
                        MSG_RECEIVER2_PRIORITY,
                        msg_receiver2_stk,
                        TASK_STACKSIZE,
                        NULL,
                        0);

    /*  创建 MailSend1 任务  */
    OSTaskCreateExt(MailSend1,
                        NULL,
                        &mail_send1_stk[TASK_STACKSIZE],
                        MAIL_SEND1_PRIORITY,
                        MAIL_SEND1_PRIORITY,
                        mail_send1_stk,
                        TASK_STACKSIZE,
                        NULL,
                        0);

    /*  创建 MailSend2 任务  */
    OSTaskCreateExt(MailSend2,
                        NULL,
                        &mail_send2_stk[TASK_STACKSIZE],
```

```
286                         MAIL_SEND2_PRIORITY,
287                         MAIL_SEND2_PRIORITY,
288                         mail_send2_stk,
289                         TASK_STACKSIZE,
290                         NULL,
291                         0);
292
293     /*  创建 MailSend3 任务  */
294     OSTaskCreateExt(MailSend3,
295                         NULL,
296                         &mail_send3_stk[TASK_STACKSIZE],
297                         MAIL_SEND3_PRIORITY,
298                         MAIL_SEND3_PRIORITY,
299                         mail_send3_stk,
300                         TASK_STACKSIZE,
301                         NULL,
302                         0);
303
304     return 0;
305 }
306 /*  This is the end of this file  */
```

下面我们就来给大家讲解一下该程序中的关键点。为什么要使用消息队列或邮箱？从上面的两个应用实例我们可以知道，任务间的通信可以通过全局变量或者信号量来完成。全局变量虽然可以承载通信的内容，但是接收方无法意识到信息的到达，除非发送方向接收方发送一个信号量，或者接收方不断轮询该全局变量；信号量可以立即即使接收方知道某个事件的发送，但无法传递具体内容。用信号量进行通信就像我们只拨通别人的手机而不与之通话；用消息队列或者邮箱进行通信则可达到既拨通别人的手机又与之通话的效果。换句话说，消息队列和邮箱可以及时传送事件的内容。

邮箱通信的机理是什么？发送方通过内核服务把一封邮件投递到邮箱，内核完成投递任务后通知等待列表中的接收方收取邮件。在整个投递过程中，内核充当了邮递员的角色。这里的邮件通常是一个指针，接收方可以用该指针获取邮件内容。

邮箱有些什么基本操作？内核通常提供如下的邮箱服务：

- 初始化邮箱的内容，邮箱最初可以包含或者不包含邮件。
- 把邮件发送到邮箱（Post），如果邮箱已满，则返回错误信息（OS_MBOX_FULL）。
- 以挂起方式接收邮件（Pend），如果邮箱为空，则把取信者挂起；若超过一定时间邮箱仍为空，则返回超时信息。
- 以非挂起方式接收邮件（Accept），如果邮箱为空，则返回一个空指针。

有了邮箱为什么还要增加消息队列服务？当希望一次性向某个任务发送多则消息时，邮箱就优点捉襟见肘了，因为一个邮箱只能装一封信。把多个邮箱集中到一起管理和使用就变成了消息队列，所以消息队列的操作和邮箱很相似。可以简单地认为，消息队列是邮箱数组。

消息队列有些什么基本操作？消息队列的操作基本和邮箱类似：

- 初始化消息队列，初始化后消息队列为空。
- 把消息存入到队列（Post），如果队列已满，则返回错误信息（OS_Q_FULL）。
- 以挂起方式接收信息（Pend），如果队列为空，则把调用者挂起；若超过一定时间队列仍为空，则返回超时信息。
- 以非挂起方式接收信息（Accept），如果队列为空，则返回一个空指针。
- 发送一则紧急消息（Front），一般来说，消息在队列中的传递顺序是先进先出（FIFO）。但如果希望发送一则紧急消息，可以调用（OSQPostFront），把消息直接插到队列的最前端，即后入先出（LIFO）。

当有多个任务同时等待某封邮件时，应该把邮件给谁？如果发送者以一对一的方式发送邮件，则等待列表中的优先级最高的任务将获取邮件；如果发送者以广播的方式发送邮件，则等待该邮件的所有任务将获得此邮件。

通信方式之一句话比较：

- 信号量是最简单并且最快的通信方式，但携带的消息太少。
- 事件标志组比信号量稍微复杂一点，它较之信号量的优势是允许任务同时等待多个时间信号或者这些信号相与（AND）或相或（OR）后的结果。
- 邮箱较之信号量可以携带更多的内容，但是传递速度较慢。
- 消息队列传递信息的内容最多，但是执行速度也最慢。

时间与空间是程序设计中的一对永恒不变的矛盾，矛盾的双方相互作用演化出了各种实际的处理方法。我们应该根据实际情况在时间和空间这对矛盾之间选择一个合理的折中，从而决定选用哪种实现方法。

(4) 板级调试

讲完了软件工程，接下来我们就将该实验下载至我们的A4开发板进行验证，首先我们需要在 Quartus II 软件中将 Qsys_Ucosii_Mbox_Queue.sof 下载至我们的 A4 开发板，Qsys_Ucosii_Mbox_Queue.sof 下载完成后，我们还需要在 Eclipse 软件中将 Qsys_Ucosii_Mbox_Queue.elf 文件下载至我们的A4开发板，Qsys_Ucosii_Mbox_Queue.elf 下载完成以后，我们的 C 程序将会执行在我们的 A4 开发板上，此时，我们可以在 Eclipse 软件的控制台中看到我们的打印信息，如图 1.310 所示。

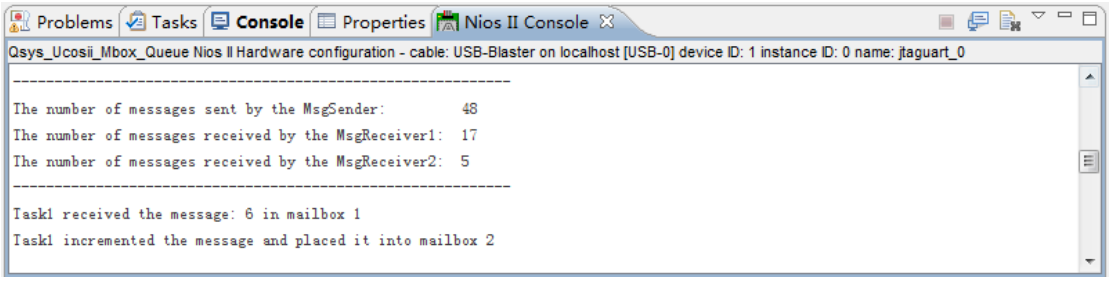

图 1.310 消息邮箱和消息队列的控制台打印信息图