《嵌入式操作系统实验指导书(μCOS-II)》实验指导书

鲍磊

电子与信息工程系计算机应用教研室

2011-11-18

目　录

嵌入式系统教学平台实验教材

第一部分 μC/OS-II

第一章 典型范例

这一部分简要地介绍了一下 μC/OS-II 的相关概念，关于更详细的介绍，读者可以参考《μC/OS-II—源代码公开的实时嵌入式操作系统》这本书。

本章将提供三个范例用来说明如何使用 μC/OS-II。在开始介绍这些例子之前，先来说明一些在这本书里的约定。

这些例子曾经用 Borland C/C++ 编译器（V4.51）编译过，用选择项产生 Intel/AMD80186 处理器（大模式下编译）的代码。这些代码实际上是在 Intel Pentium II PC （300MHz）上运行和测试过，Intel Pentium II PC 可以看成是特别快的 80186。笔者选择 PC 做为目标系统是由于以下几个原因：首先也是最为重要的，以 PC 做为目标系统比起以其他嵌入式环境，如评估板，仿真器等，更容易进行代码的测试，不用不断地烧写 EPROM，不断地向 EPROM 仿真器中下载程序等等。用户只需要简单地编译、链接和执行。其次，使用 Borland C/C++ 产生的 80186 的目标代码（实模式，在大模式下编译）与所有 Intel、AMD、Cyrix 公司的 80x86 CPU 兼容。

1.1 安装 μC/OS-II

实验指导书附带的光盘里面包括了所有的源代码。应当具备 80x86，或者 Pentium-II 以上处理器的 PC 机，并且运行 DOS 或 Windows95 以上的操作系统。至少需要 5Mb 硬盘空间来安装 μC/OS-II。请按照以下步骤安装：

插入附带光盘，打开 μC/OS-II 安装文件，将其中文件复制到目标目录下。

注意：为了在 μC/OS 使用中不会更改过多内容，建议使用 C：＼，D：＼，或 E：＼作为目标目录。

在安装之前请一定阅读一下 READ ME 文件。完成时，用户的目标目录下应该有以下子目录：

\SOFTWARE

这是根目录，所有软件相关的文件都放在这个目录下。

\SOFTWARE\BLOCKS

子程序模块目录。笔者将例子中 μC/OS-II 用到的与 PC 相关的函数模块编译以后放在这个目录下。

\SOFTWARE\HPLISTC

与范例 HPLIST 相关的文件。HPLIST.C 存放在\SOFTWARE\HPLISTC\SOURCE 目录下。DOS 下的可执行文件（HPLIST.EXE）存放在\SOFTWARE\TO\EXE 中。

\SOFTWARE\TO

与范例 TO 相关的文件。源文件 TO.C 存放在\SOFTWARE\TO\SOURCE 中，DOS 下的

可执行文件（TO.EXE）存放在\SOFTWARE\TO\EXE 中。注意 TO 需要一个 TO.TBL 文件，它必须放在根目录下。用户可以在\SOFTWARE\TO\EXE 目录下找到 TO.TBL 文件。如果要运行 TO.EXE，必须将 TO.TBL 复制到根目录下。

\SOFTWARE\uCOS-II

与 μC/OS-II 相关的文件都放在这个目录下。

\SOFTWARE\uCOS-II\EX1_x86L

这个目录里包括例 1 的源代码，可以在 DOS（或 Windows 95 下的 DOS 窗口）下运行。

\SOFTWARE\uCOS-II\EX2_x86L

这个目录里包括例 2 的源代码，可以在 DOS（或 Windows 95 下的 DOS 窗口）下运行。

\SOFTWARE\uCOS-II\EX3_x86L

这个目录里包括例 3 的源代码，可以在 DOS（或 Windows 95 下的 DOS 窗口）下运行。

\SOFTWARE\uCOS-II\Ix86L

这个目录下包括依赖于处理器类型的代码。此时是为在 80x86 处理器上运行 uC/OS-II 而必须的一些代码，实模式，在大模式下编译。

\SOFTWARE\uCOS-II\SOURCE

这个目录里包括与处理器类型无关的源代码。这些代码完全可移植到其它架构的处理器上。

1.2 范例 1

第一个范例可以在\SOFTWARE\uCOS_II\EX1_x86L 目录下找到，它有 13 个任务(包括 μC/OS-II 的空闲任务)。μC/OS-II 增加了两个内部任务：空闲任务和一个计算 CPU 利用率的任务。例 1 建立了 11 个其它任务。TaskStart()任务是在函数 main()中建立的；它的功能是建立其它任务并且在屏幕上显示如下统计信息：

- ✧ 每秒钟任务切换次数；
- ✧ CPU 利用百分率；
- ✧ 寄存器切换次数；
- ✧ 目前日期和时间；
- ✧ μC/OS-II 的版本号；

TaskStart()还检查是否按下 ESC 键，以决定是否返回到 DOS。

其余 10 个任务基于相同的代码——Task()；在屏幕随机的位置上显示一个 0~9 的数字。每个任务只显示同一个数字，一个任务对应一个数字，也就是其中一个任务在随机位置显示 0，另一个显示 1，等等。

运行范例中的.EXE 程序在 DOS 窗口中观察运行结果。约 1s 后，我们可以看到 DOS 窗口几乎被 0~9 的数字随机地填满了，如图 1.1 所示。

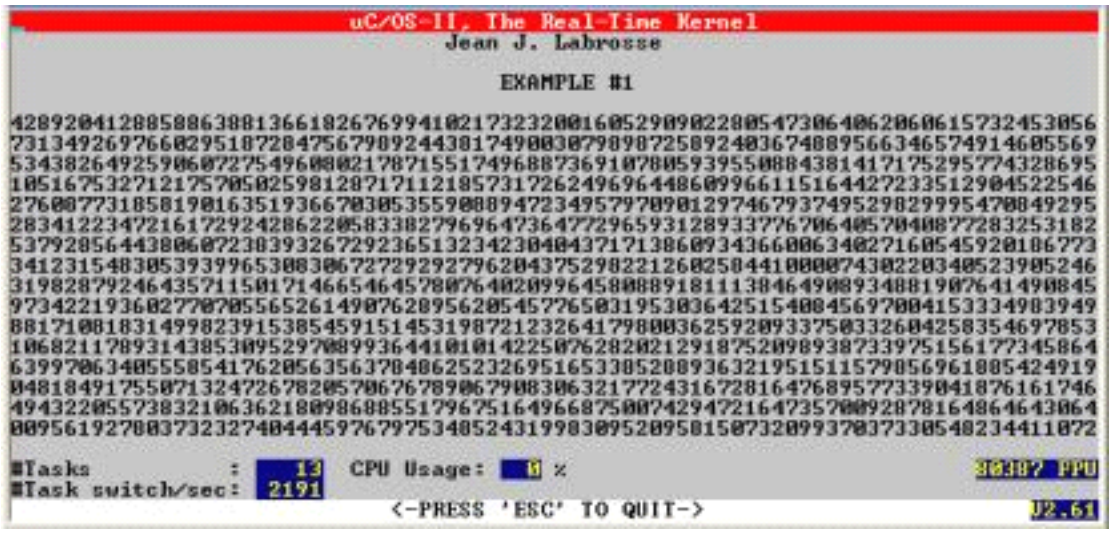

图 1.1　范例 1 的运行窗口

1.2.1 main()

例 1 基本上和最初 μC/OS 中的第一个例子做一样的事，但是笔者整理了其中的代码，并且在屏幕上加了彩色显示。同时笔者使用原来的数据类型（UBYTE, UWORD 等）来说明 μC/OS-II 向下兼容。

main()程序从清整个屏幕开始，为的是保证屏幕上不留有以前的 DOS 下的显示[程序清单 1.1(1)]。注意，笔者定义了白色的字符和黑色的背景色。既然要清屏幕，所以可以只定义背景色而不定义前景色，但是这样在退回 DOS 之后，用户就什么也看不见了。这也是为什么总要定义一个可见的前景色。

μC/OS-II 要用户在使用任何服务之前先调用 OSInit() [程序清单 1.1 (2)]。它会建立两个任务：空闲任务和统计任务，前者在没有其它任务处于就绪态时运行；后者计算 CPU 的利用率。

程序清单 1.1　main（）

```
void main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);                    (1)
    OSInit();                                                            (2)
    PC_DOSSaveReturn();                                                  (3)
    PC_VectSet(uCOS, OSCtxSw);                                           (4)
    RandomSem = OSSemCreate(1);                                          (5)
    OSTaskCreate(TaskStart,                                              (6)
                 (void *)0,
                 (void *)&TaskStartStk[TASK_STK_SIZE-1],
                 0);
    OSStart();                                                           (7)
}
```

当前 DOS 环境是通过调用 PC_DOSSaveReturn()[程序清单 1.1　(3)]来保存的。这使得用户可以返回到没有运行 μC/OS-II 以前的 DOS 环境。跟随清单程序清单 1.2 中的程序可以看到 PC_DOSSaveReturn()做了很多事情。PC_DOSSaveReturn()首先设置 PC_ExitFlag 为

FALSE[程序清单 1.2 (1)]，说明用户不是要返回 DOS，然后初始化 OSTickDOSCtr 为 1[程序清单 1.2(2)]，因为这个变量将在 OSTickISR()中递减，而 0 将使得这个变量在 OSTickISR()中减 1 后变为 255。然后，PC_DOSSaveReturn()将 DOS 的时钟节拍处理（tick handler）存入一个自由向量表入口中[程序清单 1.2(3)-(4)]，以便为 μC/OS-II 的时钟节拍处理所调用。接着 PC_DOSSaveReturn()调用 jmp（）[程序清单 1.2 (5)]，它将处理器状态（即所有寄存器的值）存入被称为 PC_JumpBuf 的结构之中。保存处理器的全部寄存器使得程序返回到 PC_DOSSaveReturn()并且在调用 setjmp（）之后立即执行。因为 PC_ExitFlag 被初始化为 FALSE[程序清单 1.2 (1)]。PC_DOSSaveReturn()跳过 if 状态语句 [程序清单 1.2 (6)–(9)] 回到 main（）函数。如果用户想要返回到 DOS，可以调用 PC_DOSReturn()[程序清单 1.3]，它设置 PC_ExitFlag 为 TRUE，并且执行 longjmp（）语句[程序清单 1.3(2)]，这时处理器将跳回 PC_DOSSaveReturn()[在调用 setjmp()之后] [程序清单 1.2(5)]，此时 PC_ExitFlag 为 TRUE，故 if 语句以后的代码将得以执行。 PC_DOSSaveReturn()将时钟节拍改为 18.2Hz[程序清单 1.2(6)]，恢复 PC 时钟节拍中断服务[程序清单 1.2(7)]，清屏幕[程序清单 1.2(8)]，通过 exit(0)返回 DOS [程序清单 1.2(9)]。

程序清单 1.2 保存 DOS 环境

```
void PC_DOSSaveReturn (void)
{
    PC_ExitFlag    = FALSE;                                          (1)
    OSTickDOSCtr =8;                                                 (2)
    PC_TickISR = PC_VectGet(VECT_TICK);                              (3)
    OS_ENTER_CRITICAL();
    PC_VectSet(VECT_DOS_CHAIN, PC_TickISR);                          (4)
    OS_EXIT_CRITICAL();

    Setjmp(PC_JumpBuf);                                              (5)
    if (PC_ExitFlag == TRUE) {
        OS_ENTER_CRITICAL();
        PC_SetTickRate(18);                                          (6)
        PC_VectSet(VECT_TICK, PC_TickISR);                           (7)
        OS_EXIT_CRITICAL();
        PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);            (8)
        exit(0);                                                     (9)
    }
}
```

程序清单 1.3 设置返回 DOS

```
void PC_DOSReturn (void)
{
    PC_ExitFlag = TRUE;                                              (1)
    longjmp(PC_JumpBuf, 1);                                          (2)
}
```

现在回到 main ）这个函数，在程序清单 1.1 中，main ）调用 PC_VectSet()来设置 μCOS-II 中的 CPU 寄存器切换。任务级的 CPU 寄存器切换由 80x86 INT 指令来分配向量地址。笔者

使用向量 0x80（即 128），因为它未被 DOS 和 BIOS 使用。

这里用了一个信号量来保护 Borland C/C++库中的产生随机数的函数[程序清单 1.1(5)]，之所以使用信号量保护一下，是因为笔者不知道这个函数是否具备可重入性，笔者假设其不具备，初始化将信号量设置为 1，意思是在某一时刻只有一个任务可以调用随机数产生函数。

在开始多任务之前，笔者建立了一个叫做 TaskStart()的任务[程序清单 1.1(6)]，在启动多任务 OSStart()之前用户至少要先建立一个任务,这一点非常重要[程序清单 1.1(7)]。不这样做用户的应用程序将会崩溃。实际上，如果用户要计算 CPU 的利用率时，也需要先建立一个任务。μCOS-II 的统计任务要求在整个一秒钟内没有任何其它任务运行。如果用户在启动多任务之前要建立其它任务，必须保证用户的任务代码监控全局变量 OSStatRdy 和延时程序[即调用 OSTimeDly()]的执行，直到这个变量变成 TRUE。这表明 μC/OS-II 的 CPU 利用率统计函数已经采集到了数据。

1.2.2 TaskStart()

例 1 中的主要工作由 TaskStart()来完成。TaskStart()函数的示意代码如程序清单 1.4 所示。TaskStart()首先在屏幕顶端显示一个标识，说明这是例 1。然后关中断，以改变中断向量，让其指向 μC/OS-II 的时钟节拍处理，而后，改变时钟节拍率，从 DOS 的 18.2Hz 变为 200Hz。在处理器改变中断向量时以及系统没有完全初始化前，当然不希望程序被中断！ 注意 main（）这个函数[程序清单 1.1]在系统初始化的时候并没有将中断向量设置成 μC/OS-II 的时钟节拍处理程序，做嵌入式应用时，用户必须在第一个任务中打开时钟节拍中断。

程序清单 1.4　建立其它任务的任务

```
void TaskStart (void *data)
{
    Prevent compiler warning by assigning ‘data’ to itself;
    Display banner identifying this as EXAMPLE #1;                    (1)

    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                                      (2)
    PC_SetTickRate(200);                                              (3)
    OS_EXIT_CRITICAL();

    Initialize the statistic task by calling ‘OSStatInit()’;          (4)

    Create 10 identical tasks;                                        (5)

    for (;;) {
        Display the number of tasks created;
        Display the % of CPU used;
        Display the number of task switches in 1 second;
        Display uC/OS-II’s version number
        If (key was pressed) {
            if (key pressed was the ESCAPE key) {
                PC_DOSReturn();
```

```
            }
        }
        Delay for 1 Second;
    }
}
```

在建立其他任务之前，必须调用 OSStatInit()[程序清单 1.4(4)]来确定用户的 PC 有多快，如程序清单 1.5 所示。在一开始，OSStatInit()就将自身延时了两个时钟节拍，这样它就可以与时钟节拍中断同步[程序清单 1.5(1)]。因此，OSStatInit()必须在时钟节拍启动之后调用；否则，用户的应用程序就会崩溃。当 μC/OS-II 调用 OSStatInit()时，一个 32 位的计数器 OSIdleCtr 被清为 0 [程序清单 1.5(2)]，并产生另一个延时，这个延时使 OSStatInit()挂起。此时，uCOS-II 没有别的任务可以执行，它只能执行空闲任务（μC/OS-II 的内部任务）。空闲任务是一个无线的循环，它不断的递增 OSIdleCtr[程序清单 1.5(3)]。1 秒以后，uCOS-II 重新开始 OSStatInit()，并且将 OSIdleCtr 保存在 OSIdleMax 中[程序清单 1.5(4)]。所以 OSIdleMax 是 OSIdleCtr 所能达到的最大值。而当用户再增加其他应用代码时，空闲任务就不会占用那样多的 CPU 时间。OSIdleCtr 不可能达到那样多的记数，（如果允许程序每秒复位一次 OSIdleCtr）CPU 利用率的计算由 μC/OS-II 中的 OSStatTask()函数来完成，这个任务每秒执行一次。而当 OSStatRdy 置为 TRUE[程序清单 1.5(5)],表示 μC/OS-II 将统计 CPU 的利用率。

程序清单 1.5 测试 CPU 速度

```
void OSStatInit (void)
{
    OSTimeDly(2);                                          (1)
    OS_ENTER_CRITICAL();
    OSIdleCtr = 0L;                                        (2)
    OS_EXIT_CRITICAL();
    OSTimeDly(OS_TICKS_PER_SEC);                           (3)
    OS_ENTER_CRITICAL();
    OSIdleCtrMax = OSIdleCtr;                              (4)
    OSStatRdy= TRUE;                                       (5)
    OS_EXIT_CRITICAL();
}
```

1.2.3 TaskN（）

OSStatInit()将返回到 TaskStart()。现在，用户可以建立 10 个同样的任务（所有任务共享同一段代码）。所有任务都由 TaskStart()中建立，由于 TaskStart()的优先级为 0（最高），新任务建立后不进行任务调度。当所有任务都建立完成后，TaskStart()将进入无限循环之中，在屏幕上显示统计信息，并检测是否有 ESC 键按下，如果没有按键输入，则延时一秒开始下一次循环；如果在这期间用户按下了 ESC 键，TaskStart()将调用 PC_DOSReturn()返回 DOS 系统。

程序清单 1.6 给出了任务的代码。任务一开始，调用 OSSemPend()获取信号量 RandomSem [程序清单 1.6(1)]（也就是禁止其他任务运行这段代码—译者注），然后调用 Borland C/C++的库函数 random（）来获得一个随机数[程序清单 1.6(2)]，设 random（）函

数是不可重入的，所以 10 个任务将轮流获得信号量，并调用该函数。当计算出 x 和 y 坐标后[程序清单 1.6(3)]，任务释放信号量。随后任务在计算的坐标处显示其任务号（0-9，任务建立时的标识）[程序清单 1.6(4)]。最后，任务延时一个时钟节拍[程序清单 1.6(5)]，等待进入下一次循环。系统中每个任务每秒执行 200 次，10 个任务每秒钟将切换 2000 次。

程序清单 1.6 在屏幕上显示随机位置显示数字的任务

```
void Task (void *data)
{
    UBYTE x;
UBYTE y;

    UBYTE err;

    for (;;) {
        OSSemPend(RandomSem, 0, &err);                                  (1)
        x = random(80);                                                 (2)
        y = random(16);
        OSSemPost(RandomSem);                                           (3)
        PC_DispChar(x, y + 5, *(char *)data, DISP_FGND_LIGHT_GRAY);     (4)
        OSTimeDly(1);                                                   (5)
    }
}
```

1.3 范例 2

第二个范例可以在\SOFTWARE\uCOS-II\EX2_x86L 目录下找到，它包含 9 个任务。加上 uCOS-II 本身的两个任务：空闲任务（idle task）和统计任务。与例 1 一样 TaskStart（）由 main（）函数建立，其功能是建立其他任务并在屏幕上显示如下的统计数据：

- ✧ 每秒种任务切换的次数；
- ✧ CPU 利用率的百分比；
- ✧ 当前日期和时间；
- ✧ μC/OS-II 的版本号；

大约 1s 后，我们可以看到如图 1.2 所示的窗口。

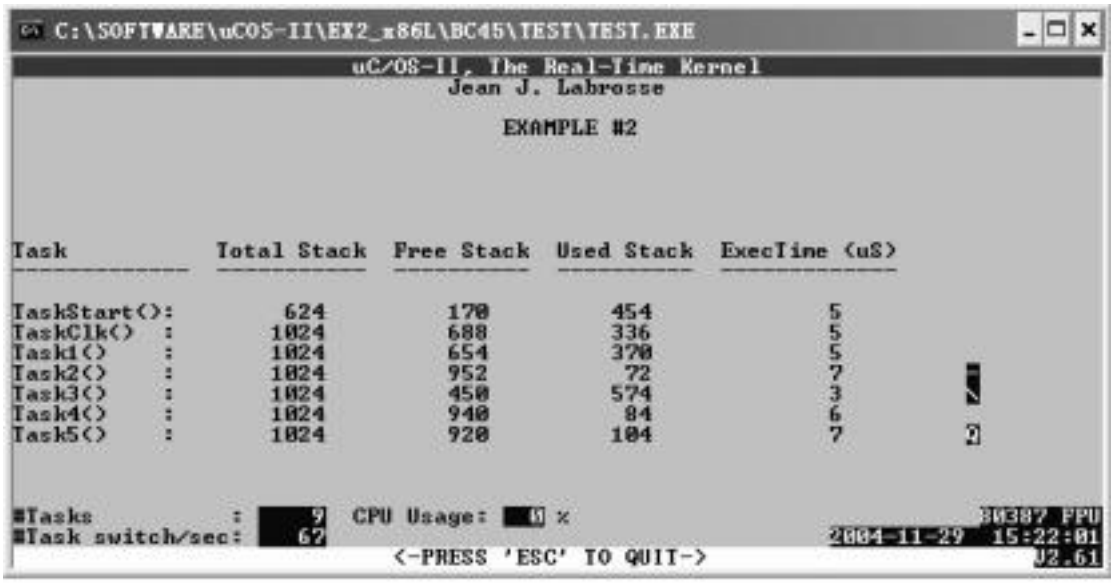

图 1.2　范例 2 的显示

例 2使用了带扩展功能的任务建立函数 OSTaskCreateExt（）和 μC/OS-II 的堆栈检查操作（要使用堆栈检查操作必须用 OSTaskCreateExt（）建立任务）。当用户不知道应该给任务分配多少堆栈空间时，堆栈检查功能是很有用的。在这个例子里，先分配足够的堆栈空间给任务，然后用堆栈检查操作看看任务到底需要多少堆栈空间。显然，任务要运行足够长时间，并要考虑各种情况才能得到正确数据。最后决定的堆栈大小还要考虑系统今后的扩展，一般多分配 10%，25%或者更多。如果系统对稳定性要求高，则应该多一倍以上。

μC/OS-II 的堆栈检查功能要求任务建立时堆栈清零。OSTaskCreateExt()可以执行此项操作（设置选项 OS_TASK_OPT_STK_CHK 和 OS_TASK_OPT_STK_CLR 打开此项操作）。如果任务运行过程中要进行建立、删除任务的操作，应该设置好上述的选项，确保任务建立后堆栈是清空的。同时要意识到 OSTaskCreateExt()进行堆栈清零操作是一项很费时的工作，而且取决于堆栈的大小。执行堆栈检查操作的时候，uCOS-II 从栈底向栈顶搜索非 0 元素(参看图 1.2)，同时用一个计数器记录 0 元素的个数。

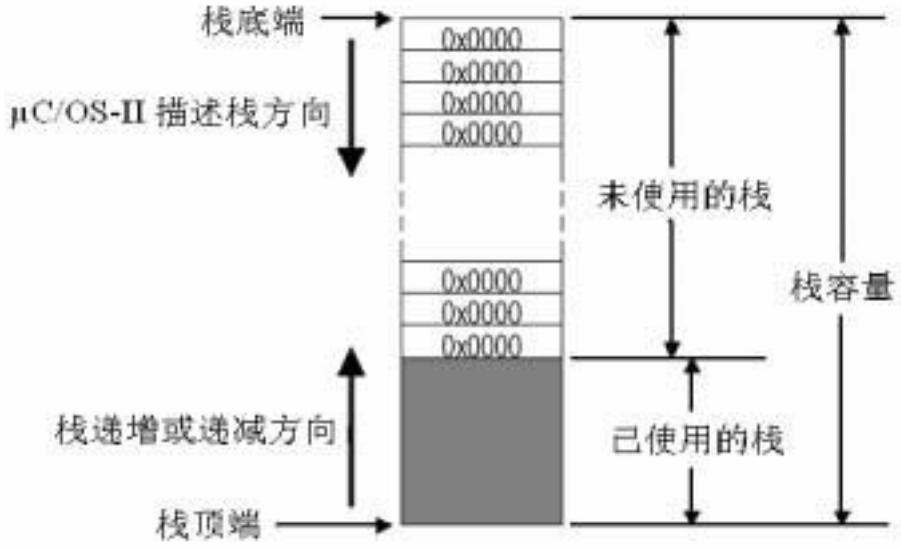

图 1.3　μC/OS-II 堆栈检查

1.3.1 main()

例 2的 main()函数和例 1 的看起来差不多（参看程序清单 1.7），但是有两处不同。第一，main()函数调用 PC_ElapsedInit()[程序清单 1.7(1)]来初始化定时器记录 OSTaskStkChk()的执行时间。第二，所有的任务都使用 OSTaskCreateExt()函数来建立任务[程序清单 1.7(2)][替代老版本的 OSTaskCreate（）]，这使得每一个任务都可进行堆栈检查。

程序清单 1.7　例 2 中的 Main（）函数

```
void main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);
    OSInit();
    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OSCtxSw);
    PC_ElapsedInit();                                          (1)
    OSTaskCreateExt(TaskStart,                                 (2)
                    (void *)0,
                    &TaskStartStk[TASK_STK_SIZE-1],
                    TASK_START_PRIO,
                    TASK_START_ID,
                    &TaskStartStk[0],
                    TASK_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
    OSStart();
}
```

除了 OSTaskCreate()函数的四个参数外，OSTaskCreateExt()还需要五个参数（一共 9 个）：任务的 ID，一个指向任务堆栈栈底的指针，堆栈的大小（以堆栈单元为单位，80X86 中为字），一个指向用户定义的 TCB 扩展数据结构的指针，和一个用于指定对任务操作的变量。该变量的一个选项就是用来设定 uCOS-II 堆栈检查是否允许。例 2 中并没有用到 TCB 扩展数据结构指针。

1.3.2 TaskStart()

程序清单 1.8 列出了 TaskStart()的伪码。前五项操作和例 1 中相同。TaskStart（）建立了两个邮箱，分别提供给任务 4 和任务 5[程序清单 1.8(1)]。除此之外，还建立了一个专门显示时间和日期的任务。

程序清单 1.8　TaskStart()的伪码

```
void TaskStart (void *data)
{
    Prevent compiler warning by assigning ‘data’ to itself;
    Display a banner and non-changing text;
```

```
    Install uC/OS-II's tick handler;
    Change the tick rate to 200 Hz;
    Initialize the statistics task;
    Create 2 mailboxes which are used by Task #4 and #5;                    (1)
    Create a task that will display the date and time on the screen;         (2)
    Create 5 application tasks;
    for (;;) {
        Display #tasks running;
        Display CPU usage in %;
        Display #context switches per seconds;
        Clear the context switch counter;
        Display uC/OS-II's version;
        If (Key was pressed) {
            if (Key pressed was the ESCAPE key) {
                Return to DOS;
            }
        }
        Delay for 1 second;
    }
}
```

1.3.3 TaskN()

任务 1 将检查其他七个任务堆栈的大小，同时记录 OSTackStkChk()函数的执行时间[程序清单 1.9(1)–(2)]，并与堆栈大小一起显示出来。注意所有堆栈的大小都是以字节为单位的。任务 1 每秒执行 10 次[程序清单 1.9(3)]（间隔 100ms）。

程序清单 1.9 例 2，任务 1

```
void   Task1 (void *pdata)
{
    INT8U           err;
    OS_STK_DATA data;
    INT16U          time;
    INT8U           i;
    char            s[80];

    pdata = pdata;
    for (;;) {
        for (i = 0; i < 7; i++) {
            PC_ElapsedStart();                                              (1)
            err = OSTaskStkChk(TASK_START_PRIO+i, &data)
            time = PC_ElapsedStop();                                        (2)
            if (err == OS_NO_ERR) {
                sprintf(s, "%3ld        %3ld        %3ld        %5d",
```

```
                                data.OSFree + data.OSUsed,
                                data.OSFree,
                                data.OSUsed,
                                time);
                        PC_DispStr(19, 12+i, s, DISP_FGND_YELLOW);
                    }
                }
                OSTimeDlyHMSM(0, 0, 0, 100);                                    (3)
            }
        }
```

程序清单 1.10 所示的任务 2 在屏幕上显示一个顺时针旋转的指针（用横线，斜线等字符表示—译者注），每 200ms 旋转一格。

程序清单 1.10　任务 2

```
    void    Task2 (void *data)
    {
        data = data;
        for (;;) {
            PC_DispChar(70, 15, '|',    DISP_FGND_WHITE + DISP_BGND_RED);
            OSTimeDly(10);
            PC_DispChar(70, 15, '/',    DISP_FGND_WHITE + DISP_BGND_RED);
            OSTimeDly(10);
            PC_DispChar(70, 15, '-',    DISP_FGND_WHITE + DISP_BGND_RED);
            OSTimeDly(10);
            PC_DispChar(70, 15, '\\', DISP_FGND_WHITE + DISP_BGND_RED);
            OSTimeDly(10);
        }
    }
```

任务 3(程序清单　1.11)也显示了与任务 2 相同的一个旋转指针，但是旋转的方向不同。任务 3 在堆栈中分配了一个很大的数组，将堆栈填充掉，使得 OSTaskStkChk()只需花费很少的时间来确定堆栈的利用率，尤其是当堆栈已经快满的时候。

程序清单 1.11　任务 3

```
void    Task3 (void *data)
{
    char       dummy[500];
    INT16U     i;
    data = data;
    for (I = 0; i < 499; i++) {
        dummy[i] = '?';
    }
    for (;;) {
```

```
            PC_DispChar(70, 16, '|',    DISP_FGND_WHITE + DISP_BGND_BLUE);
            OSTimeDly(20);
            PC_DispChar(70, 16, '\\', DISP_FGND_WHITE + DISP_BGND_BLUE);
            OSTimeDly(20);
            PC_DispChar(70, 16, '-',    DISP_FGND_WHITE + DISP_BGND_BLUE);
            OSTimeDly(20);
            PC_DispChar(70, 16, '/',    DISP_FGND_WHITE + DISP_BGND_BLUE);
            OSTimeDly(20);
        }
}
```

任务 4(程序清单 1.12)向任务 5 发送消息并等待确认[程序清单 1.12(1)]。发送的消息是一个指向字符的指针。每当任务 4 从任务 5 收到确认[程序清单 1.12(2)],就将传递的 ASCII 码加 1 再发送[程序清单 1.12(3)]，结果是不断的传送“ABCDEFG....”。

程序清单 1.12　　任务 4

```
void    Task4 (void *data)
{
    char     txmsg;
    INT8U    err;

    data    = data;
    txmsg = 'A';
    for (;;) {
        while (txmsg <= 'Z') {
            OSMboxPost(TxMbox, (void *)&txmsg);                    (1)
            OSMboxPend(AckMbox, 0, &err);                          (2)
            txmsg++;                                               (3)
        }
        txmsg = 'A';
    }
}
```

当任务 5 [程序清单 1.13]接收消息后[程序清单 1.13(1)]（发送的字符），就将消息显示到屏幕上[程序清单 1.13(2)]，然后延时 1 秒[程序清单 1.13(3)]，再向任务 4 发送确认信息。

程序清单　1.13　任务 5

```
void    Task5 (void *data)
{
    char    *rxmsg;
    INT8U    err;

    data = data;
```

```
    for (;;) {
        rxmsg = (char *)OSMboxPend(TxMbox, 0, &err);                              (1)
        PC_DispChar(70, 18, *rxmsg, DISP_FGND_YELLOW+DISP_BGND_RED);   (2)
        OSTimeDlyHMSM(0, 0, 1, 0);                                                (3)
        OSMboxPost(AckMbox, (void *)1);                                           (4)
    }
}
```

TaskClk()函数[程序清单 1.14]显示当前日期和时间，每秒更新一次。

程序清单 1.14　时钟显示任务

```
void   TaskClk (void *data)
{
    Struct time now;
    Struct date today;
    char          s[40];


    data = data;
    for (;;) {
        PC_GetDateTime(s);
        PC_DispStr(0, 24, s, DISP_FGND_BLUE + DISP_BGND_CYAN);
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}
```

1.4 范例 3

第三个范例可以在\SOFTWARE\uCOS-II\EX3_x86L 目录下找到，它包括 9 个任务。除了空闲任务（idle task）和统计任务（statistic task ），还有 7 个任务。与例 1，例 2 一样，TaskStart()由 main（）函数建立，其功能是建立其他任务，并显示统计信息。

大约 1s 后，我们可以看到如图 1.4 所示的窗口。例 3 中使用了许多 uCOS-II 提供的附加功能。任务 3 使用了 OSTaskCreateExt()中 TCB 的扩展数据结构，用户定义的任务切换对外接口函数（OSTaskSwHook()），用户定义的统计任务（statistic task）的对外接口函数（OSTaskStatHook()）以及消息队列。

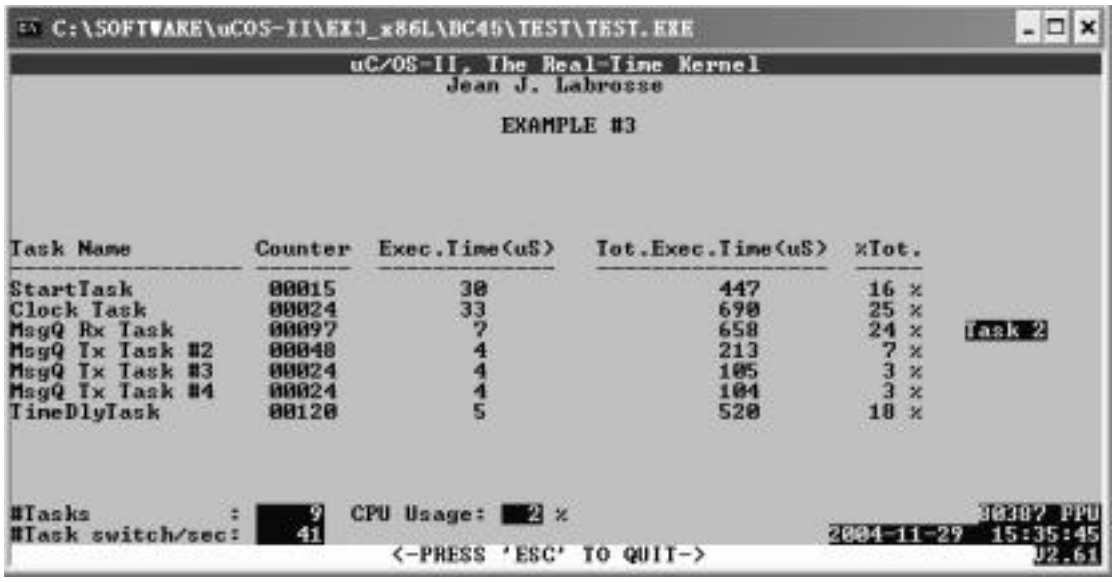

图 1.4 范例 3 的显示

1.4.1 main()

main()函数[程序清单 1.15]和例 2 中的相差不多，不同的是在用户定义的 TCB 扩展数据结构中可以保存每个任务的名称[程序清单 1.15(1)]（扩展结构的声明在 INCLUDES.H 中定义，也可参看程序清单 1.16）。笔者定义了 30 个字节来存放任务名（包括空格）[程序清单 1.16(1)]。本例中没有用到堆栈检查操作，TaskStart()中禁止该操作[程序清单 1.15(2)]。

程序清单 1.15 例 3 的 main()函数

```
void main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);
    OSInit();
    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OSCtxSw);
    PC_ElapsedInit();

    Strcpy(TaskUserData[TASK_START_ID].TaskName, "StartTask");                    (1)
    OSTaskCreateExt(TaskStart,
                    (void *)0,
                    &TaskStartStk[TASK_STK_SIZE-1],
                    TASK_START_PRIO,
                    TASK_START_ID,
                    &TaskStartStk[0],
                    TASK_STK_SIZE,
                    &TaskUserData[TASK_START_ID],
                    0);                                                           (2)
    OSStart();
}
```

程序清单 1.16　TCB 扩展数据结构

```
typedef struct {
    char       TaskName[30];                                    (1)
    INT16U     TaskCtr;
    INT16U     TaskExecTime;
    INT32U     TaskTotExecTime;
} TASK_USER_DATA;
```

1.4.2 任务

TaskStart()的伪码如程序清单 1.17 所示，与例 2 有 3 处不同：

- ✧ 为任务 1，2，3 建立了一个消息队列[程序清单 1.17(1)]；
- ✧ 每个任务都有一个名字，保存在任务的 TCB 扩展数据结构中[程序清单 1.17(2)]；
- ✧ 禁止堆栈检查。

程序清单 1.17　TaskStart()的伪码

```
void TaskStart (void *data)
{
    Prevent compiler warning by assigning 'data' to itself;
    Display a banner and non-changing text;
    Install uC/OS-II's tick handler;
    Change the tick rate to 200 Hz;
    Initialize the statistics task;
    Create a message queue;                                     (1)
    Create a task that will display the date and time on the screen;
    Create 5 application tasks with a name stored in the TCB ext.;   (2)
    for (;;) {
        Display #tasks running;
        Display CPU usage in %;
        Display #context switches per seconds;
        Clear the context switch counter;
        Display uC/OS-II's version;
        If (Key was pressed) {
            if (Key pressed was the ESCAPE key) {
                Return to DOS;
            }
        }
        Delay for 1 second;
    }
}
```

任务 1 向消息队列发送一个消息[程序清单 1.18(1)]，然后延时等待消息发送完成[程序清单 1.18(2)]。这段时间可以让接收消息的任务显示收到的消息。发送的消息有三种。

程序清单 1.18　任务 1

```
void    Task1 (void *data)
{
    char one   = '1';
    char two   = '2';
    char three = '3';


    data = data;
    for (;;) {
        OSQPost(MsgQueue, (void *)&one);                    (1)
        OSTimeDlyHMSM(0, 0, 1,    0);                       (2)
        OSQPost(MsgQueue, (void *)&two);
        OSTimeDlyHMSM(0, 0, 0, 500);
        OSQPost(MsgQueue, (void *)&three);
        OSTimeDlyHMSM(0, 0, 1,    0);
    }
}
```

任务 2 处于等待消息的挂起状态，且不设定最大等待时间[程序清单 1.19(1)]。所以任务 2 将一直等待直到收到消息。当收到消息后，任务 2 显示消息并且延时 500mS[程序清单 1.19(2)]，延时的时间可以使任务 3 检查消息队列。

程序清单 1.19　任务 2

```
void    Task2 (void *data)
{
    INT8U *msg;
    INT8U  err;


    data = data;
    for (;;) {
        msg = (INT8U *)OSQPend(MsgQueue, 0, &err);                  (1)
        PC_DispChar(70, 14, *msg, DISP_FGND_YELLOW+DISP_BGND_BLUE); (2)
        OSTimeDlyHMSM(0, 0, 0, 500);                                (3)
    }
}
```

任务 3 同样处于等待消息的挂起状态，但是它设定了等待结束时间 250ms[程序清单 1.20(1)]。如果有消息来到，任务 3 将显示消息号[程序清单 1.20(3)]，如果超过了等待时间，任务 3 就显示“T”（意为 timeout）[程序清单 1.20(2)]。

程序清单 1.20　任务 3

```
void   Task3 (void *data)
{
    INT8U *msg;
    INT8U   err;


    data = data;
    for (;;) {
        msg = (INT8U *)OSQPend(MsgQueue, OS_TICKS_PER_SEC/4, &err);              (1)
        If (err == OS_TIMEOUT) {
            PC_DispChar(70,15,'T',DISP_FGND_YELLOW+DISP_BGND_RED);          (2)
        } else
            PC_DispChar(70,15,*msg,DISP_FGND_YELLOW+DISP_BGND_BLUE);   (3)
        }
    }
}
```

任务 4 的操作只是从邮箱发送[程序清单 1.21(1)]和接收[程序清单 1.21(2)]，这使得用户可以测量任务在自己 PC 上执行的时间。任务 4 每 10mS 执行一次[程序清单 1.21(3)]。

程序清单 1.21　　任务 4

```
void   Task4 (void *data)
{
    OS_EVENT *mbox;
    INT8U        err;

    data = data;
    mbox = OSMboxCreate((void *)0);
    for (;;) {
        OSMboxPost(mbox, (void *)1);                                    (1)
        OSMboxPend(mbox, 0, &err);                                      (2)
        OSTimeDlyHMSM(0, 0, 0, 10);                                     (3)
    }
}
```

任务 5 除了延时一个时钟节拍以外什么也不做[程序清单 1.22(1)]。注意所有的任务都应该调用 uCOS-II 的函数，等待延时结束或者事件的发生而让出 CPU。如果始终占用 CPU，这将使低优先级的任务无法得到 CPU。

程序清单 1.22　任务 5

```
void   Task5 (void *data)
{
    data = data;
```

```
    for (;;) {
        OSTimeDly(1);                                                    (1)
    }
}
```

同样，TaskClk()函数[程序清单 1.14]显示当前日期和时间。

1.4.3 注意

有些程序的细节只有请您仔细读一读 EX3L.C 才能理解。EX3L.C 中有 OSTaskSwHook()函数的代码，该函数用来测量每个任务的执行时间，可以用来统计每一个任务的调度频率，也可以统计每个任务运行时间的总和。这些信息将存储在每个任务的 TCB 扩展数据结构中。每次任务切换的时候 OSTaskSwHook()都将被调用。

每次任务切换发生的时候，OSTaskSwHook()先调用 PC_ElapsedStop()函数[程序清单 1.23(1)] 来获取任务的运行时间[程序清单 1.23(1)]，PC_ElapsedStop()要和 PC_ElapsedStart()一起使用，上述两个函数用到了 PC 的定时器 2（timer 2）。其中 PC_ElapsedStart()功能为启动定时器开始记数；而 PC_ElapsedStop()功能为获取定时器的值，然后清零，为下一次计数做准备。从定时器取得的计数将拷贝到 time 变量[程序清单 1.23(1)]。然后 OSTaskSwHook()调用 PC_ElapsedStart()重新启动定时器做下一次计数[程序清单 1.23(2)]。需要注意的是，系统启动后，第一次调用 PC_ElapsedStart()是在初始化代码中，所以第一次任务切换调用 PC_ElapsedStop()所得到的计数值没有实际意义，但这没有什么影响。如果任务分配了 TCB 扩展数据结构[程序清单 1.23(4)]，其中的计数器 TaskCtr 进行累加[程序清单 1.23(5)]。TaskCtr 可以统计任务被切换的频繁程度，也可以检查某个任务是否在运行。TaskExecTime [程序清单 1.23(6)]用来记录函数从切入到切出的运行时间，TaskTotExecTime[程序清单 1.23(7)]记录任务总的运行时间。统计每个任务的上述两个变量，可以计算出一段时间内各个任务占用 CPU 的百分比。OSTaskStatHook()函数会显示这些统计信息。

程序清单 1.23　用户定义的 OSTaskSwHook()

```
void OSTaskSwHook (void)
{
    INT16U                  time;
    TASK_USER_DATA *puser;


    time    = PC_ElapsedStop();                                          (1)
    PC_ElapsedStart();                                                   (2)
    puser = OSTCBCur->OSTCBExtPtr;                                       (3)
    if (puser != (void *)0) {                                            (4)
        puser->TaskCtr++;                                                (5)
        puser->TaskExecTime         = time;                              (6)
        puser->TaskTotExecTime += time;                                  (7)
    }
}
```

本例中的统计任务（statistic task）将调用对外接口函数 OSTaskStatHook()（设置OS_CFG.H 文件中的OS_TASK_STAT_EN 为1 允许对外接口函数）。统计任务每秒运行一次，本例中 OSTaskStatHook()用来计算并显示各任务占用CPU 的情况。

OSTaskStatHook()函数中首先计算所有任务的运行时间[程序清单 1.24(1)]，DispTaskStat()用来将数字显示为ASCII 字符[程序清单1.24(2)]。然后是计算每个任务运行时间的百分比[程序清单 1.24(3)]，显示在合适的位置上 [程序清单 1.24(4)]。

程序清单 1.24 用户定义的 OSTaskStatHook()

```
void OSTaskStatHook (void)
{
    char     s[80];
    INT8U    i;
    INT32U total;
    INT8U    pct;

    total = 0L;
    for (i = 0; i < 7; i++) {
        total += TaskUserData[i].TaskTotExecTime;                    (1)
        DispTaskStat(i);                                             (2)
    }
    if (total > 0) {
        for (i = 0; i < 7; i++) {
            pct = 100 * TaskUserData[i].TaskTotExecTime / total;     (3)
            sprintf(s, "%3d %%", pct);
            PC_DispStr(62, i + 11, s, DISP_FGND_YELLOW);             (4)
        }
    }
    if (total > 1000000000L) {
        for (i = 0; i < 7; i++) {
            TaskUserData[i].TaskTotExecTime = 0L;
        }
    }
}
```

第二章 μC/OS-II 的移植

这一章介绍如何将 μC/OS-Ⅱ移植到用户开发板上。所谓移植，就是使一个实时内核能在某个微处理器或微控制器上运行。uC/OS-II 在特定处理器上的移植工作绝大部分集中在多任务切换的实现上，因为这部分代码主要是用来保存和恢复处理器现场，许多操作入读写寄存器操作不能用 C语言，只能使用特定的处理器的汇编语言来完成。

2.1 μC/OS-II 移植概述

2.1.1 μC/OS-Ⅱ移植的要求

要使μC/OS-Ⅱ正常运行，处理器必须满足以下要求：

处理器的 C 编译器能产生可重入代码。

用 C 语言就可以打开和关闭中断。

处理器支持中断，并且能产生定时中断(通常在 10 至 100Hz 之间)。

处理器支持能够容纳一定量数据(可能是几千字节)的硬件堆栈。

处理器有将堆栈指针和其它 CPU 寄存器读出和存储到堆栈或内存中的指令。

S3C2410 处理器完全满足上述要求。

2.1.2 使用 μC/OS-II 系统应注意的问题

第一，μC/OS-II 和 Linux 等分时操作系统不同，不支持时间片轮转法。它是一个基于优先级的实时操作系统。每一个任务的优先级必须不同（分析它的源码会发现，μC/OS-II 把任务的优先级当作任务在标识来使用，如果优先级相同，任务将无法区分）。进入就绪态的优先级最高的任务首先得到 CPU 的使用权，只有等它交出 CPU 的使用权后，其它任务才可以被执行。所以，它只能就是多任务，不能就是多进程，至少不是我们所熟悉的那种多进程。

第二，μC/OS-II 对共享资源提供了保护的机制。μC/OS-II 是一个支持多任务的操作系统。我们可以把一个完整的程序划分成几个任务，不同的任务执行不同的功能。对于共享资源（比如串口），μC/OS-II 也提供了很好的解决办法，一般情况下使用的是信号量方法。我们创建一个信号量并对它进行初始化，当一个任务需要使用一个共享资源时，它必须先申请得到这个信号量。在这个过程中即使有优先权更高的任务进入了就绪态，因为无法得到信号量，也不能使用该资源。在 μC/OS-II 中称为优先级反转。简单地说，就是高优先级任务必须等待低优先级任务的完成。在上述情况下，在两个任务之间发生优先级后转是无法避免的。所以在使用 μC/OS-II 时，必须对所开发的系统了解清楚才能选择对于某种共享资源是否使用信号量。

第三，μC/OS-II 内存管理不够完善。在分析许多 μC/OS-II 的应用实例中发现，任务栈空间和内存分区的创建采用了定义全局数组的方法，这样实现起来固然简单，但不够灵活有效。

编译器会将全局数组作为未初始化的全局变量，放到应用程序映像的数据段。数组的大小是固定的，生成映像后不可能在使用中动态地改变。对于任务栈空间来说，数组定义大了会造成内存浪费；定义小了任务栈溢出，会造成系统崩溃。对于内存分区，在不知道系统初

始化后给用户留下了多少自由内存空间的情况下，很难定义内存分区所使用数组的大小。此外，现在 μC/OS-II 只支持固定大小的内存分区，容易造成内存浪费。μC/OS-II 将来应该被改进以支持可变大小的内存分区。因此，系统初始化后能清楚地掌握自由内存空间的情况是很重要的。所以，应避免使用全局数组分配内存空间，关键是要知道整个应用程序在编译、链接后代码段和数据段的大小，在目标板内存中是如何定位，以及目标板内存的大小。

总之，随着各种智能嵌入式系统的复杂化和系统实时性需求的提高，伴随应用软件朝着系统化发展的加速，功能强大的实时操作系统 μC/OS-II 将会有更大的发展。

2.2 μC/OS-II 移植需要修改的相关文件

将 μC/OS-II 移植到 ARM 处理器上，需要完成的工作非常简单，修改三个和体系结构相关的文件即可，代码量大约是 500 行。这三个文件是 OS_CPU_C.C、OS_CPU_C.H 以及 OS_CPU_A.S，下面将分别说明。

1．OS_CPU_C.h 的移植

1)数据类型定义

该文件中定义了本系统中所使用的数据类型，这部分的修改是和所有的编译器相关的，不同的编译器会使用不同的字节长度来表示同一数据类型，比如 int，同样在 x86 平台上，如果用 GNU 的 gcc 编译器，则编译位 4 bytes，而使用 MS VC++则编译位 2 bytes。我们这里使用的是 GNU 的 gcc。此外，该文件还定义了堆栈单位，它定义了在处理器现场保存和恢复时所使用的数据类型，它必须和处理器的寄存器长度一致。相关的数据类型的定义如下：

```
typedef unsigned  char          BOOLEAN;
typedef unsigned  char          INT8U;
typedef signed    char          INT8S;
typedef unsigned  int           INT16U;
typedef signed    int           INT16S;
typedef unsigned  long          INT32U;
typedef signed    long          INT32S;
typedef float                   FP32;
typedef double                  FP64;
typedef unsigned int            OS_STK;
typedef unsigned int            OS_CPU_SR;
#define BYTE                    INT8S
#define UBYTE                   INT8U
#define WORD                    INT16S
#define UWORD                   INT16U
#define LONG                    INT32S
#define ULONG                   INT32U
```

2)ARM 处理器相关宏定义

该文件还定义了 ARM 处理器中退出临界区和进入临界区的宏定义，如下所示。

```
#define OS_ENTER_CRITICAL()     ARMDisableInt()
#define OS_EXIT_CRITICAL()      ARMEnableInt()
```

3)堆栈增长方向

堆栈增长方向也由该文件定义，堆栈由高地址向低地址增长，这个也是和编译器有关的，当进行函数调用时，入口参数和返回地址一般都会保存在当前任务的堆栈中，编译器的编译

选项和由此生成的堆栈指令就会决定堆栈的增长方向。

#define OS_STK_GROWTH

2．OS_CPU.c 的移植

1)任务堆栈初始化

该函数由 OSTaskCreate（）或 OSTaskCreateExt（）调用，用来初始化任务的堆栈并返回新的堆栈指针 stk。初始状态的堆栈模拟发生一次中断后的堆栈结构。在 ARM 体系结构下，任务堆栈空间由高至低依次将保存着 pc、lr、r12、r11、r10、…r1、r0、CPSR、SPSR，下图(由上到下：高地址->低地址，也是增长的方向)说明了 OSTaskStkInit（）初始化后的也是新创建任务的堆栈内容。堆栈初始化工作结束后，OSTaskStkInit（）返回新的堆栈栈顶指针，OSTaskCreate（）或 OSTaskCreateExt（）将指针保存在任务的 OS_TCB 中。

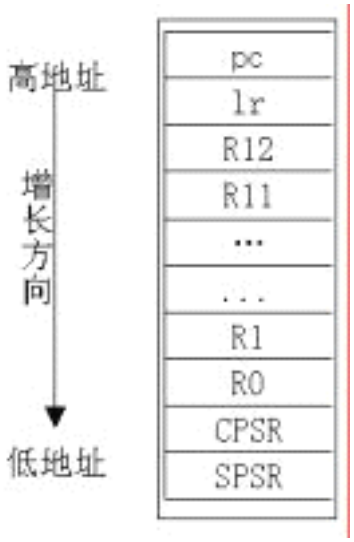

图 2.1 初始化后的堆栈内容

任务堆栈初始化函数如下所示

```
InitStacks
        mrs r0,cpsr
        bic r0,r0,#MODEMASK
        orr r1,r0,#UNDEFMODE|NOINT
        msr cpsr_cxsf,r1            ; UndefMode
        ldr sp,=UndefStack
        orr r1,r0,#ABORTMODE|NOINT
        msr cpsr_cxsf,r1            ; AbortMode
        ldr sp,=AbortStack
        orr r1,r0,#IRQMODE|NOINT
        msr cpsr_cxsf,r1            ; IRQMode
        ldr sp,=IRQStack
        orr r1,r0,#FIQMODE|NOINT
        msr cpsr_cxsf,r1            ; FIQMode
        ldr sp,=FIQStack
        bic r0,r0,#MODEMASK|NOINT
        orr r1,r0,#SVCMODE
        msr cpsr_cxsf,r1            ; SVCMode
```

```
ldr sp,=SVCStack
```

2)系统 hook 函数

在这个文件里面还需要实现几个操作系统规定的 hook 函数，这些函数为用户定义函数，它将在相应的操作系统调用后执行由用户定义的这些 hook 函数，执行特定的用户操作，如果每个特殊需求，则只需要简单地将它们都实现为空函数就可以。这些函数包括：

OSTaskCreateHook（）

OSTaskDelHook（）

OSTaskSwHook（）

OSTaskStatHook（）

OSTimeTickHook（）

3)中断级任务切换函数

该函数由 OSIntExit（）和 OSExIntExit（）调用。它是在时钟中断 ISR（中断服务例程）中发现由高优先级任务等待的时钟信号到来，则需要在中断退出后并不返回被中断任务，二是直接调度就绪的高优先级任务执行。这样做的目的这样是能够尽快地让高优先级地任务得到响应，保证系统地实时性能。该函数通过设置一个全局变量 need_to_swap_context 标志以表示在中断服务程序中进行任务切换，然后在 OSTickISR（）中判断该变量以进行正确地动作。其函数如下：

```
void OSIntCtxSw(void)
{
   need_to_swap_context=1;
}
```

3．OS_CPU.s 的移植

1)时钟节拍中断服务函数

时钟节拍是特定的周期性中断。这个中断可以看作是系统心脏的脉动。时钟的节拍式中断使得内核可以将任务延时若干个整数时钟节拍，以及当任务等待事件发生时，提供等待超时的依据。时钟节拍率越快，系统的额外开销就越大。中断之间的时间间隔取决于不同的应用，本系统使用 S3C2410 的 Timer0 作为时钟节拍源，产生间隔 10ms 的时钟节拍。

OSTickISR（）就是时钟节拍中断服务函数，也就是 S3C2410 的 Timer0 中断处理函数。

OSTickISR（）首先在被中断的任务堆栈中保存 CPU 存储器的值，然后调用 OSIntEnter（）。随后，OSTickISR（）调用 OSTimeTick（），检查所有处于延时等待状态的任务，判断是否有延时结束就绪的任务。OSTickISR（）的最后调用 OSIntExit（），如果在中断中（或其他嵌套的中断）有更高优先级的任务就绪，并且当前中断为中断嵌套的最后一层。OSIntExit（）将进行任务调度。注意如果进行了任务调度，OSIntExit（）将不再返回调用者，而是新任务的堆栈中的寄存器数值恢复 CPU 现场，然后用 IRET 实现任务切换。如果当前中断不是中断嵌套的最后一层，或中断中没有改变任务的就绪状态，OSIntExit（）将返回调用者 OSTickISR（），最后 OSTickISR（）返回被中断的任务。

OSTickISR（）先关闭中断，然后清除 timer0 中断标记（只有清除当前中断标记才能够引发下一次中断）。接着调用 IrqStart（），uC/OS-II 要求在中断服务程序开头将记录中断嵌套层数的全局变量 OSIntNesting 加 1。随后 OSTickISR（）调用 OSTimeTick（），检查所有处于延时等待状态的任务，判断是否有延时结束就绪的任务。然后调用 IrqFinish（）函数，IrqFinish（）将调用 OSIntExit（）函数，如果在中断中（或其他嵌套的中断）有更高优先级的任务就绪，并且当前中断为中断嵌套的最后一层，OSIntExit（）将进行任务调度，并在 OSIntCtxSw（）函数中设置 need_to_swap_context 标记为 1。接下来 OSTickISR（）判断 need_to_swap_context 标记是否为 1，如果为 1 则进行任务调度，将不再返回被中断的任务，

而是用新任务的堆栈中的寄存器数值恢复 CPU 现场，然后实现任务切换。如果当前中断不是中断嵌套的最后一层，或中断中没有改变任务的就绪状态，OSTickISR（）将返回被中断的任务。

2)退出临界区和进入临界区函数

它们分别是退出临界区和进入临界区的宏指令实现。主要用于在进入临界区之前关闭中断，在退出临界区的时候恢复原来的中断状态

进入临界区：关闭中断

```
.GLOBAL     ARMDisableInt
ARMD        isableInt:
STMDB       sp!, {r0}
MRS          r0, CPSR
ORR          r0, r0, #NoInt
MSR          CPSR_cxsf, r0
LDMIA       sp!, {r0}
MOV          pc, lr
```

退出临界区：恢复原来的中断 GLOBAL ARMEnableInt

```
ARMEnableInt：
STMDB       sp!, {r0}
MRS          r0, CPSR
BIC          r0, r0, #NoInt
MSR          CPSR_cxsf, r0
LDMIA       sp!, {r0}
MOV          pc, lr
```

3)任务级上下文切换函数

任务级的上下文切换函数 OS_TASK_SW（），它是当任务因为被阻塞而主动请求 cpu 调度时被执行，由于此时的任务切换都是在非异常模式下进行的，因此区别于中断级别的任务切换。它的工作是先将当前任务的 cpu 现场保存到该任务堆栈中，然后获得最高优先级任务的堆栈指针，从该堆栈中恢复此任务的 cpu 现场，使之继续执行。这样就完成了一次任务切换。

4)OSStartHighRdy

OSStartHighRdy（）函数是在 OSStart（）多任务启动之后，负责从最高优先级任务的 TCB 控制块中获得该任务的堆栈指针 sp，通过 sp 依次将 cpu 现场恢复，这时系统就将控制权交给用户创建的该任务进程，直到该任务被阻塞或者被其他更高优先级的任务抢占 cpu。该函数仅仅在多任务启动时被执行一次，用来启动第一个，也就时高优先级的任务执行。

```
    .GLOBAL   OSStartHighRdy
OSStartHighRdy：
        LDR       r4, addr_OSTCBCur              @ Get current task TCB address
        LDR       r5, addr_OSTCBHighRdy          @ Get highest priority task TCB address
        LDR       r5, [r5]                       @ get stack pointer
        LDR       sp, [r5]                       @ switch to the new stack
        STR   r5, [r4]                         @ set new current task TCB address
        LDMFD   sp!, {r4}
        MSR       SPSR_cxsf, r4
        LDMFD   sp!, {r4}                        @ get new state from top of the stack
```

```
MSR       CPSR_cxsf, r4                    @ CPSR should be SVC32Mode
LDMFD   sp!, {r0-r12, lr, pc }             @ start the new task
```

4．多任务应用程序的编写

一个最基本的uC/OS多任务应用程序的范例主要包括如下两个部分：

1) C语言入口函数

函数Main为C语言入口函数，所有的C程序从这里开始运行。在该函数中将进行如下操作：

a．调用函数ARMTargetInit初始化ARM处理器；

b．调用OSInit进行操作系统初始化；

c．调用OSTaskCreate函数两个任务：TaskLED和TaskSEG;

d．调用ARMTargetStart函数启动时钟节拍中断；

e．调用OSStart启动系统任务调度。

2)任务处理函数

在多任务中，每个任务都对应一个入口函数。这些函数必须为死循环，它们每隔 100个时钟节拍向串口打印一串字符。

第三章 μC/GUI 的几个重要概念

实时系统的特点是，如果逻辑和时序出现偏差将会引起严重后果的系统。有两种类型的实时系统：软实时系统和硬实时系统。在软实时系统中系统的宗旨是使各个任务运行得越快越好，并不要求限定某一任务必须在多长时间内完成。

在硬实时系统中，各任务不仅要执行无误而且要做到准时。大多数实时系统是二者的结合。实时系统的应用涵盖广泛的领域，而多数实时系统又是嵌入式的。这意味着计算机建在系统内部，用户看不到有个计算机在系统里面。

3.1 实时系统中的重要概念

代码的临界段

代码的临界段也称为临界区，指处理时不可分割的代码。一旦这部分代码开始执行，则不允许任何中断打入。为确保临界段代码的执行，在进入临界段之前要关中断，而临界段代码执行完以后要立即开中断。

资源

任何为任务所占用的实体都可称为资源。资源可以是输入输出设备，例如打印机、键盘、显示器，资源也可以是一个变量，一个结构或一个数组等。

共享资源

可以被一个以上任务使用的资源叫做共享资源。为了防止数据被破坏，每个任务在与共享资源打交道时，必须独占该资源。这叫做互斥（*mutual exclusion*）。

多任务

多任务运行的实现实际上是靠 CPU(中央处理单元)在许多任务之间转换、调度。CPU 只有一个，轮番服务于一系列任务中的某一个。多任务运行很像前后台系统，但后台任务有多个。多任务运行使 CPU 的利用率得到最大的发挥，并使应用程序模块化。在实时应用中，多任务化的最大特点是，开发人员可以将很复杂的应用程序层次化。使用多任务，应用程序将更容易设计与维护。

任务

一个任务，也称作一个线程，是一个简单的程序，该程序可以认为 CPU 完全只属该程序自己。实时应用程序的设计过程，包括如何把问题分割成多个任务，每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，有它自己的一套 CPU 寄存器和自己的栈空间。

典型地、每个任务都是一个无限的循环。每个任务都处在以下 5种状态之一的状态下，这 5 种状态是休眠态，就绪态、运行态、挂起态(等待某一事件发生)和被中断态
休眠态相当于该任务驻留在内存中，但并不被多任务内核所调度。就绪意味着该任务已经准备好，可以运行了，但由于该任务的优先级比正在运行的任务的优先级低，还暂时不能运行。

运行态的任务是指该任务掌握了 CPU 的控制权，正在运行中。挂起状态也可以叫做等待事件态 WAITING，指该任务在等待，等待某一事件的发生，（例如等待某外设的 I/O 操作，等待某共享资源由暂不能使用变成能使用状态，等待定时脉冲的到来或等待超时信号的到来以结束目前的等待，等等）。最后，发生中断时，CPU 提供相应的中断服务，原来正在运行的任务暂不能运行，就进入了被中断状态。

内核（Kernel）

多任务系统中，内核负责管理各个任务，或者说为每个任务分配 CPU时间，并且负责任务之间的通讯。内核提供的基本服务是任务切换。之所以使用实时内核可以大大简化应用系统的设计，是因为实时内核允许将应用分成若干个任务，由实时内核来管理它们。内核本身也增加了应用程序的额外负荷，代码空间增加 ROM 的用量，内核本身的数据结构增加了 RAM 的用量。但更主要的是，每个任务要有自己的栈空间，这一块吃起内存来是相当厉害的。内核本身对 CPU 的占用时间一般在 2 到 5 个百分点之间。

单片机一般不能运行实时内核，因为单片机的 RAM 很有限。通过提供必不可缺少 的系统服务，诸如信号量管理，邮箱、消息队列、延时等，实时内核使得 CPU 的利用更为有效。一旦读者用实时内核做过系统设计，将决不再想返回到前后台系统。

调度（*Scheduler*）

调度（Scheduler）,英文还有一词叫 dispatcher，也是调度的意思。这是内核的主要职责之一，就是要决定该轮到哪个任务运行了。多数实时内核是基于优先级调度法的。每个任务根据其重要程度的不同被赋予一定的优先级。基于优先级的调度法指，CPU 总是让处在就绪态的优先级最高的任务先运行。然而，究竟何时让高优先级任务掌握 CPU 的使用权，有两种不同的情况，这要看用的是什么类型的内核，是不可剥夺型的还是可剥夺型内核。

可重入性（Reentrancy）

可重入型函数可以被一个以上的任务调用，而不必担心数据的破坏。可重入型函数任何时候都可以被中断，一段时间以后又可以运行，而相应数据不会丢失。可重入型函数或者只使用局部变量，即变量保存在 CPU 寄存器中或堆栈中。如果使用全局变量，则要对全局变量予以保护。

优先级反转

使用实时内核,优先级反转问题是实时系统中出现得最多的问题。图 3.1 解释优先级反转是如何出现的。如图，任务 1 优先级高于任务 2，任务 2 优先级高于任务 3。任务 1 和任务 2 处于挂起状态，等待某一事件的发生，任务 3 正在运行如[图 3.1(1)]。此时，任务 3 要使用其共享资源。使用共享资源之前，首先必须得到该资源的信号量(Semaphore)(见 2.18.04 信号量)。任务 3 得到了该信号量，并开始使用该共享资源[图 3.1（2）]。由于任务 1 优先级高，它等待的事件到来之后剥夺了任务 3 的 CPU 使用权[图 3.1(3)]，任务 1 开始运行[图 3.1(4)]。运行过程中任务 1 也要使用那个任务 3 正在使用着的资源，由于该资源的信号量还被任务 3 占用着，任务 1 只能进入挂起状态，等待任务 3 释放该信号量[图 3.1(5)]。任务 3 得以继续运行[图 3.1(6)]。由于任务 2 的优先级高于任务 3，当任务 2 等待的事件发生后，任务 2 剥夺了任务 3 的 CPU 的使用权[图 3.1(7)]并开始运行。处理它该处理的事件[图 3.1(8)]，直到处理完之后将 CPU 控制权还给任 3[图 3.1(9)]。任务 3 接着运行[图 3.1(10)]，直到释放那个共享资源的信号量[图 27(11)]。直到此时,由于实时内核知道有个高优先级的任务在等待这个信号量，内核做任务切换，使任务 1 得到该信号量并接着运行[图 3.1(12)]。

在这种情况下,任务 1 优先级实际上降到了任务 3 的优先级水平。因为任务 1 要等,直等到任务 3 释放占有的那个共享资源。由于任务 2 剥夺任务 3 的 CPU 使用权,使任务 1 的状况更加恶化,任务 2 使任务 1 增加了额外的延迟时间。任务 1 和任务 2 的优先级发生了反转。

纠正的方法可以是,在任务 3 使用共享资源时,提升任务 3 的优先级。任务完成时予以恢复。任务 3 的优先级必须升至最高,高于允许使用该资源的任何任务。多任务内核应允许动态改变任务的优先级以避免发生优先级反转现象。然而改变任务的优先级是很花时间的。如果任务 3 并没有先被任务 1 剥夺 CPU 使用权, 又被任务 2 抢走了 CPU 使用权,花很多时间在共享资源使用前提升任务3的优先级,然后又在资源使用后花时间恢复任务3的优先级,则无形中浪费了很多 CPU 时间。真正需要的是,为防止发生优先级反转,内核能自动变换任务的优先级,这叫做优先级继承(Priority inheritance)但μC/OS-Ⅱ不支持优先级继承,一些商业内核有优先级继承功能。

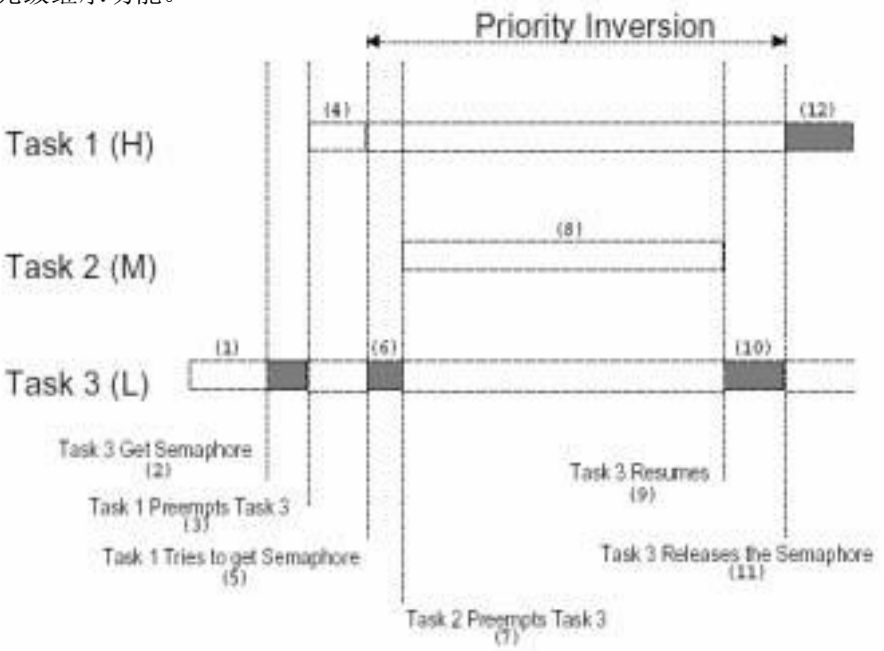

图 3.1 优先级反转问题

图 3.2 解释如果内核支持优先级继承的话,在上述例子中会是怎样一个过程。任务 3 在运行[图 3.2(1)],任务 3 申请信号量以获得共享资源使用权[图 3.2(2)],任务 3 得到并开始使用共享资源[图 3.2(3)]。后来 CPU 使用权被任务 1 剥夺[图 3.2(4)],任务 1 开始运行[图 3.2(5)],任务 1 申请共享资源信号量[图 3.2(6)]。此时,内核知道该信号量被任务 3 占用了,而任务 3 的优先级比任务 1 低,内核于是将任务 3 的优先级升至与任务 1 一样,,然而回到任务 3继续运行,使用该共享资源[图 3.1(7)],直到任务 3 释放共享资源信号量[图 2。8(8)]。这时,内核恢复任务 3 本来的优先级并把信号量交给任务 1,任务 1 得以顺利运行。 [图 3.2(9)],任务 1 完成以后[图 3.2(10)]那些任务优先级在任务 1 与任务 3 之间的任务例如任务 2 才能得到 CPU 使用权,并开始运行 [图 3.2(11)]。注意,任务 2 在从[图 3.2(3)]到[图 3.2(10)]的任何一刻都有可能进入就绪态,并不影响任务 1、任务 3 的完成过程。在某种程度上,任务 2 和任务 3 之间也还是有不可避免的优先级反转。

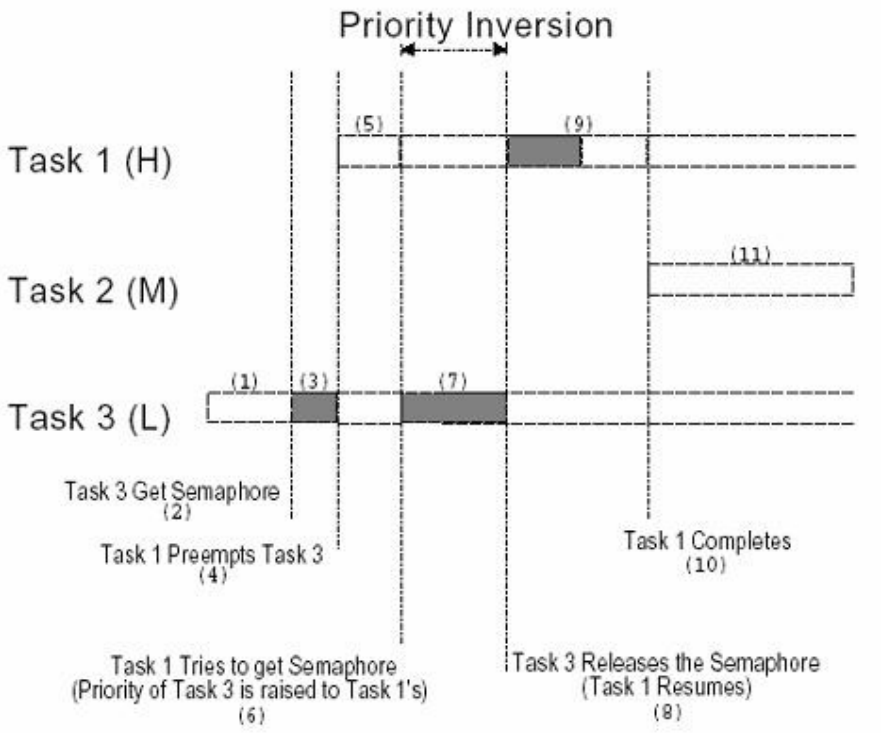

图 3.2 内核优先级继承的支持

信号量(Semaphores)

信号量是60年代中期Edgser Dijkstra 发明的。信号量实际上是一种约定机制，在多任务内核中普遍使用.信号量用于：

- 控制共享资源的使用权(满足互斥条件)
- 标志某事件的发生
- 使两个任务的行为同步

信号像是一把钥匙，任务要运行下去，得先拿到这把钥匙。如果信号已被别的任务占用，该任务只得被挂起，直到信号被当前使用者释放。换句话说，申请信号的任务是在说：“把钥匙给我，如果谁正在用着，我只好等!”信号是只有两个值的变量，信号量是计数式的。只取两个值的信号是只有两个值0和1的量，因此也称之为信号量。计数式信号量的值可以是 0 到 255 或 0 到 65535，或 0 到 4294967295，取决于信号量规约机制使用的是 8 位、16 位还是 32 位。到底是几位，实际上是取决于用的哪种内核。根据信号量的值，内核跟踪那些等待信号量的任务。

任务间的通讯(Intertask Communication)

有时很需要任务间的或中断服务与任务间的通讯。这种信息传递称为任务间的通讯。任务间信息的传递有两个途径：通过全程变量或发消息给另一个任务。

用全程变量时，必须保证每个任务或中断服务程序独享该变量。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，各任务实现独享该变量的办法可以是关中断再开中断，或使用信号量(如前面提到的那样)。请注意，任务只能通过全程变量与中断服务程序通讯，而任务并不知道什么时候全程变量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。要避免这种情况，用户可以考虑使用邮箱或消息队列。

消息邮箱(Message Mail boxes)

通过内核服务可以给任务发送消息。典型的消息邮箱也称作交换消息，是用一个指针型变量，通过内核服务，一个任务或一个中断服务程序可以把一则消息(即一个指针)放到邮箱里去。同样，一个或多个任务可以通过内核服务接收这则消息。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。

每个邮箱有相应的正在等待消息的任务列表，要得到消息的任务会因为邮箱是空的而被挂起，且被记录到等待消息的任务表中，直到收到消息。一般地说，内核允许用户定义等待超时，等待消息的时间超过了，仍然没有收到该消息，这任务进入就绪态，并返回出错信息，报告等待超时错误。消息放入邮箱后，或者是把消息传给等待消息的任务表中优先级最高的那个任务(基于优先级)，或者是将消息传给最先开始等待消息的任务(基于先进先出)。

内核一般提供以下邮箱服务：

- 邮箱内消息的内容初始化，邮箱里最初可以有，也可以没有消息
- 将消息放入邮箱(POST)
- 等待有消息进入邮箱(PEND)
- 如果邮箱内有消息，就接受这则消息。如果邮箱里没有消息，则任务并不被挂起(ACCEPT)，用返回代码表示调用结果，是收到了消息还是没有收到消息。

消息邮箱也可以当作只取两个值的信号量来用。邮箱里有消息，表示资源可以使用，而空邮箱表示资源已被其它任务占用。

消息队列(Message Queue)

消息队列用于给任务发消息。消息队列实际上是邮箱阵列。通过内核提供的服务，任务或中断服务子程序可以将一条消息(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得到消息。发送和接收消息的任务约定，传递的消息实际上是传递的指针指向的内容。通常，先进入消息队列的消息先传给任务，也就是说，任务先得到的是最先进入消息队列的消息，即先进先出原则(FIFO)。然而μC/OS-Ⅱ也允许使用后进先出方式(LIFO)。

像使用邮箱那样，当一个以上的任务要从消息队列接收消息时，每个消息队列有一张等待消息任务的等待列表(Waiting List)。如果消息队列中没有消息，即消息队列是空，等待消息的任务就被挂起并放入等待消息任务列表中，直到有消息到来。通常，内核允许等待消息的任务定义等待超时的时间。如果限定时间内任务没有收到消息，该任务就进入就绪态并开始运行，同时返回出错代码，指出出现等待超时错误。一旦一则消息放入消息队列，该消息将传给等待消息的任务中优先级最高的那个任务，或是最先进入等待消息任务列表的任务。

典型地，内核提供的消息队列服务如下：

消息队列初始化。队列初始化时总是清为空。

放一则消息到队列中去(Post)

等待一则消息的到来(Pend)

如果队列中有消息则任务可以得到消息，但如果此时队列为空，内核并不将该任务挂起(Accept)。如果有消息，则消息从队列中取走。没有消息则用特别的返回代码通知调用者，队列中没有消息。

中断

中断是一种硬件机制，用于通知CPU有个异步事件发生了。中断一旦被识别，CPU保存部分(或全部)现场(Context)即部分或全部寄存器的值，跳转到专门的子程序，称为中断服务子程序(ISR)。中断服务子程序做事件处理，处理完成后，程序回到：

在前后台系统中，程序回到后台程序
对不可剥夺型内核而言，程序回到被中断了的任务
对可剥夺型内核而言，让进入就绪态的优先级最高的任务开始运行

实时系统特点的小结

中断使得 CPU 可以在事件发生时才予以处理，而不必让微处理器连续不断地查询(Polling)是否有事件发生。通过两条特殊指令：关中断(Disable interrupt)和开中断(Enable interrupt)可以让微处理器不响应或响应中断。在实时环境中，关中断的时间应尽量的短。关中断影响中断延迟时间。关中断时间太长可能会引起中断丢失。微处理器一般允许中断嵌套，也就是说在中断服务期间，微处理器可以识别另一个更重要的中断，并服务于那个更重要的中断。

三种类型的实时系统归纳于下表中，这三种实时系统是：前后台系统，不可剥夺型内核和可剥夺型内核。

表 3.1

μC/GUI 实时系统的特点

	Foreground/ Background	***Non-Preemptive Kernel***	***Preemptive Kernel***
Interrupt latency (Time)	MAX(Longest instruction, User int. disable) + Vector to ISR	MAX(Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR	MAX(Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR
Interrupt response (Time)	Int. latency + Save CPU's context	Int. latency + Save CPU's context	Interrupt latency + Save CPU's context + Kernel ISR entry function
Interrupt recovery (Time)	Restore background's context + Return from int.	Restore task's context + Return from int.	Find highest priority task + Restore highest priority task's context + Return from interrupt
Task response (Time)	Background	Longest task + Find highest priority task + Context switch	Find highest priority task + Context switch
ROM size	Application code	Application code + Kernel code	Application code + Kernel code
RAM size	Application code	Application code + Kernel RAM + SUM(Task stacks + MAX(ISR stack))	Application code + Kernel RAM + SUM(Task stacks + MAX(ISR stack))
Services available?	Application code must provide	Yes	Yes

3.2 μC/GUI 的函数和相关的常量

表 3.2 μC/OS-II 函数和相关的常量（#define constant 定义）

表 *μC/OS-II* 函数和相关常量

类型	置 *1*	其他常量
杂相		
OSInit()	无	OS_MAX_EVENTS OS_Q_EN and OS_MAX_QS OS_MEM_EN OS_TASK_IDLE_STK_SIZE OS_TASK_STAT_EN OS_TASK_STAT_STK_SIZE
OSSchedLock()	无	无
OSSchedUnlock()	无	无
OSStart()	无	无
OSStatInit()	OS_TASK_STAT_EN && OS_TASK_CREATE_EXT_EN	OS_TICKS_PER_SEC
OSVersion()	无	无
中断处理		
OSIntEnter()	无	无
OSIntExit()	无	无
消息邮箱		
OSMboxAccept()	OS_MBOX_EN	无
OSMboxCreate()	OS_MBOX_EN	OS_MAX_EVENTS
OSMboxPend()	OS_MBOX_EN	无
OSMboxPost()	OS_MBOX_EN	无
OSMboxQuery()	OS_MBOX_EN	无
内存块管理		
OSMemCreate()	OS_MEM_EN	OS_MAX_MEM_PART
OSMemGet()	OS_MEM_EN	无
OSMemPut()	OS_MEM_EN	无
OSMemQuery()	OS_MEM_EN	无
消息队列		

OSQAccept()	OS_Q_EN	无
OSQCreate()	OS_Q_EN	OS_MAX_EVENTS OS_MAX_QS
OSQFlush()	OS_Q_EN	无
OSQPend()	OS_Q_EN	无
OSQPost()	OS_Q_EN	无
OSQPostFront()	OS_Q_EN	无
OSQQuery()	OS_Q_EN	无
信号量管理		
OSSemAccept()	OS_SEM_EN	无
OSSemCreate()	OS_SEM_EN	OS_MAX_EVENTS
OSSemPend()	OS_SEM_EN	无
OSSemPost()	OS_SEM_EN	无
OSSemQuery()	OS_SEM_EN	无
任务管理		
OSTaskChangePrio()	OS_TASK_CHANGE_PRIO_EN	OS_LOWEST_PRIO
OSTaskCreate()	OS_TASK_CREATE_EN	OS_MAX_TASKS OS_LOWEST_PRIO
OSTaskCreateExt()	OS_TASK_CREATE_EXT_EN	OS_MAX_TASKS OS_STK_GROWTH OS_LOWEST_PRIO
OSTaskDel()	OS_TASK_DEL_EN	OS_LOWEST_PRIO
OSTaskDelReq()	OS_TASK_DEL_EN	OS_LOWEST_PRIO
OSTaskResume()	OS_TASK_SUSPEND_EN	OS_LOWEST_PRIO
OSTaskStkChk()	OS_TASK_CREATE_EXT_EN	OS_LOWEST_PRIO
OSTaskSuspend()	OS_TASK_SUSPEND_EN	OS_LOWEST_PRIO
OSTaskQuery()		OS_LOWEST_PRIO
时钟管理		
OSTimeDly()	无	无
OSTimeDlyHMSM()	无	OS_TICKS_PER_SEC
OSTimeDlyResume()	无	OS_LOWEST_PRIO
OSTimeGet()	无	无
OSTimeSet()	无	无
OSTimeTick()	无	无

用户定义函数		
OSTaskCreateHook()	OS_CPU_HOOKS_EN	无
OSTaskDelHook()	OS_CPU_HOOKS_EN	无
OSTaskStatHook()	OS_CPU_HOOKS_EN	无
OSTaskSwHook()	OS_CPU_HOOKS_EN	无
OSTimeTickHook()	OS_CPU_HOOKS_EN	无

OS_MAX_EVENTS

OS_MAX_EVENTS 定义系统中最大的事件控制块的数量。系统中的每一个消息邮箱，消息队列，信号量都需要一个事件控制块。例如，系统中有 10 个消息邮箱，5 个消息队列，3 个信号量，则 OS_MAX_EVENTS 最小应该为 18。只要程序中用到了消息邮箱，消息队列或是信号量，则 OS_MAX_EVENTS 最小应该设置为 2。

OS_MAX_MEM_PARTS

OS_MAX_MEM_PARTS 定义系统中最大的内存块数，内存块将由内存管理函数操作（定义在文件 OS_MEM.C 中）。如果要使用内存块，OS_MAX_MEM_PARTS 最小应该设置为 2，常量 OS_MEM_EN 也要同时置 1。

OS_MAX_QS

OS_MAX_QS 定义系统中最大的消息队列数。要使用消息队列，常量 OS_Q_EN 也要同时置 1。如果要使用消息队列，OS_MAX_ QS 最小应该设置为 2。

OS_MAX_TASKS

OS_MAX_MEM_TASKS 定义用户程序中最大的任务数。OS_MAX_MEM_TASKS 不能大于 62，这是由于 μC/OS-II 保留了两个系统使用的任务。如果设定 OS_MAX_MEM_TASKS 刚好等于所需任务数，则建立新任务时要注意检查是否超过限定。而 OS_MAX_MEM_TASKS 设定的太大则会浪费内存。

OS_LOWEST_PRIO

OS_LOWEST_PRIO 设定系统中的任务最低优先级（最大优先级数）。设定 OS_LOWEST_PRIO 可以节省用于任务控制块的内存。μC/OS-II 中优先级数从 0（最高优先级）到 63（最低优先级）。设定 OS_LOWEST_PRIO 小于 63 意味着不会建立优先级数大于 OS_LOWEST_PRIO 的任务。μC/OS-II 中保留两个优先级系统自用：OS_LOWEST_PRIO 和 OS_LOWEST_PRIO-1。其中 OS_LOWEST_PRIO 留给系统的空闲任务（Idle task）(OSTaskIdle（）)。OS_LOWEST_PRIO-1 留给统计任务（OSTaskStat（））。用户任务的优先级可以从 0 到 OS_LOWEST_PRIO-2。OS_LOWEST_PRIO 和 OS_MAX_TASKS 之间没有什么关系。例如，可以设 OS_MAX_TASKS 为 10 而

OS_LOWEST_PRIO 为 32。此时系统最多可有 10 个任务，用户任务的优先级可以是 0 到 30。当然，OS_LOWEST_PRIO 设定的优先级也要够用，例如设 OS_MAX_TASKS 为 20，而 OS_LOWEST_PRIO 为 10，优先级就不够用了。

OS_TASK_IDLE_STK_SIZE

OS_TASK_IDLE_STK_SIZE 设置μC/OS-II 中空闲任务（Idle task）堆栈的容量。注意堆栈容量的单位不是字节，而是 OS_STK（μC/OS-II 中堆栈统一用 OS_STK 声明，根据不同的硬件环境，OS_STK 可为不同的长度----译者注）。空闲任务堆栈的容量取决于所使用的处理器，以及预期的最大中断嵌套数。虽然空闲任务几乎不做什么工作，但还是要预留足够的堆栈空间保存 CPU 寄存器的内容，以及可能出现的中断嵌套情况。

OS_TASK_STAT_EN

OS_TASK_STAT_EN 设定系统是否使用μC/OS-II 中的统计任务（statistic task）及其初始化函数。如果设为 1，则使用统计任务 OSTaskStat（）。统计任务每秒运行一次，计算当前系统 CPU 使用率，结果保存在 8 位变量 OSCPUUsage 中。每次运行，OSTaskStat（）都将调用 OSTaskStatHook（）函数，用户自定义的统计功能可以放在这个函数中。详细情况请参考 OS_CORE.C 文件。统计任务 OSTaskStat（）的优先级总是设为 OS_LOWEST_PRIO-1。当 OS_TASK_STAT_EN 设为 0 的时候，全局变量 OSCPUUsage，OSIdleCtrMax，OSIdleCtrRun 和 OSStatRdy 都不声明，以节省内存空间。

OS_TASK_STAT_STK_SIZE

OS_TASK_STAT_STK_SIZE 设置μC/OS-II 中统计任务（statistic task）堆栈的容量。注意单位不是字节，而是 OS_STK（μC/OS-II 中堆栈统一用 OS_STK 声明，根据不同的硬件环境，OS_STK 可为不同的长度----译者注）。统计任务堆栈的容量取决于所使用的处理器类型，以及如下的操作：

- 进行 32 位算术运算所需的堆栈空间。
- 调用 OSTimeDly（）所需的堆栈空间。
- 调用 OSTaskStatHook（）所需的堆栈空间。
- 预计最大的中断嵌套数。

如果想在统计任务中进行堆栈检查，判断实际的堆栈使用，用户需要设 OS_TASK_CREATE_EXT_EN 为 1，并使用 OSTaskCreateExt（）函数建立任务。

OS_CPU_HOOKS_EN

此常量设定是否在文件 OS_CPU_C.C 中声明对外接口函数（hook function），设为 1 为声明。μC/OS-II 中提供了 5 个对外接口函数，可以在文件 OS_CPU_C.C 中声明，也可以在用户自己的代码中声明：

- OSTaskCreateHook（）
- OSTaskDelHook（）
- OSTaskStatHook（）

- OSTaskSwHook（）
- OSTimeTickHook（）

OS_MBOX_EN

OS_MBOX_EN 控制是否使用μC/OS-II 中的消息邮箱函数及其相关数据结构，设为 1 为使用。如果不使用，则关闭此常量节省内存。

OS_MEM_EN

OS_MEM_EN 控制是否使用μC/OS-II 中的内存块管理函数及其相关数据结构，设为 1 为使用。如果不使用，则关闭此常量节省内存。

OS_Q_EN

OS_Q_EN 控制是否使用μC/OS-II 中的消息队列函数及其相关数据结构，设为 1 为使用。如果不使用，则关闭此常量节省内存。如果 OS_Q_EN 设为 0，则语句#define constant OS_MAX_QS 无效。

OS_SEM_EN

OS_SEM_EN 控制是否使用μC/OS-II 中的信号量管理函数及其相关数据结构，设为 1 为使用。如果不使用，则关闭此常量节省内存。

OS_TASK_CHANGE_PRIO_EN

此常量控制是否使用μC/OS-II 中的 OSTaskChangePrio（）函数，设为 1 为使用。如果在应用程序中不需要改变运行任务的优先级，则将此常量设为 0 节省内存。

OS_TASK_CREATE_EN

此常量控制是否使用μC/OS-II 中的 OSTaskCreate（）函数，设为 1 为使用。在μC/OS-II 中推荐用户使用 OSTaskCreateExt（）函数建立任务。如果不使用 OSTaskCreate（）函数，将 OS_TASK_CREATE_EN 设为 0 可以节省内存。注意 OS_TASK_CREATE_EN 和 OS_TASK_CREATE_EXT_EN 至少有一个要为 1，当然如果都使用也可以。

OS_TASK_CREATE_EXT_EN

此常量控制是否使用μC/OS-II 中的 OSTaskCreateExt（）函数，设为 1 为使用。该函数为扩展的，功能更全的任务建立函数。如果不使用该函数，将 OS_TASK_CREATE_EXT_EN 设为 0 可以节省内存。注意，如果要使用堆栈检查函数 OSTaskStkChk（），则必须用 OSTaskCreateExt（）建立任务。

OS_TASK_DEL_EN

此常量控制是否使用μC/OS-II 中的 OSTaskDel（）函数，设为 1 为使用。如果在应用程序中不使用删除任务函数，将 OS_TASK_DEL_EN 设为 0 可以节省内存。

OS_TASK_SUSPEND_EN

此常量控制是否使用μC/OS-II 中的 OSTaskSuspend（）和 OSTaskResume（）函数，设为 1 为使用。如果在应用程序中不使用任务挂起-唤醒函数，将 OS_TASK_SUSPEND_EN 设为 0 可以节省内存。

OS_TICKS_PER_SEC

此常量标识调用 OSTimeTick（）函数的频率。用户需要在自己的初始化程序中保证 OSTimeTick（）按所设定的频率调用（即系统硬件定时器中断发生的频率----译者注）。在函数 OSStatInit ），OSTaskStat ）和 OSTimeDlyHMSM（）中都会用到 OS_TICKS_PER_SEC。

第二部分 μC/GUI

第一章 μC/GUI 简介

现代的嵌入式产品，不论是工业用、医疗用或者是消费性电子，除了一些不需要人类操作或只需几个按钮者，很难说要摆脱人机界面这一环，尤其随着 LCD 面板的普及与价格低下，图形化的界面似乎更是各等级必备条件。也因此，如果一个作业系统，缺乏这样的一套中介软件，可能会无人问津。Micriμm（www.ucos-ii.com）这家公司当然也会想到这点商机，故提出这类的产品称为 μC/GUI Embedded Software Suite，如图 1.1。

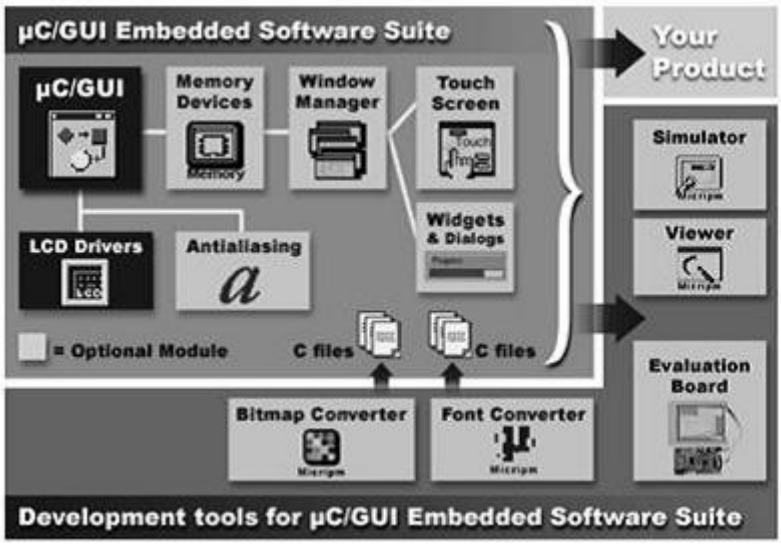

图 1.1 μC/GUI 图形界面支持工具（SOURCE：www.ucos-ii.com ）

这套图形界面软件包含四个阶层，分别为最高阶的视窗管理员、控制程式库、图形程式库与最底层的 LCD 驱动程式，基本上与 CPU 是独立的，且全部由标准的 ANSI-C 设计成，换句话说 μC/GUI 继承了 μC/OS II 的可携性优点。图五中我们可以看到一个简单的人机界面，这是用其视窗管理员所做出来的，其中用到的 Widget 有一般常见的按钮、Checkbox、文字编辑、对话框、进度图、ScrollBar、Slider…等，对于一般的应用已经足足有余了。其他支持如 μC/GUI Simulator 与 μC/GUI View，让我们可以方便的在 PC 上面开发与除错应用程式、做模拟与验证的工作。档案 μC/FS，当然也是阶层式架构，最上层为应用程序呼叫界面、档案系统、逻辑区块到最底层的驱动程序，支持 MS-DOS/Windows 相容 FAT12 和 FAT16 格式，硬件方面支持 RAM Disk、Smart Card、Compact Flash 等。

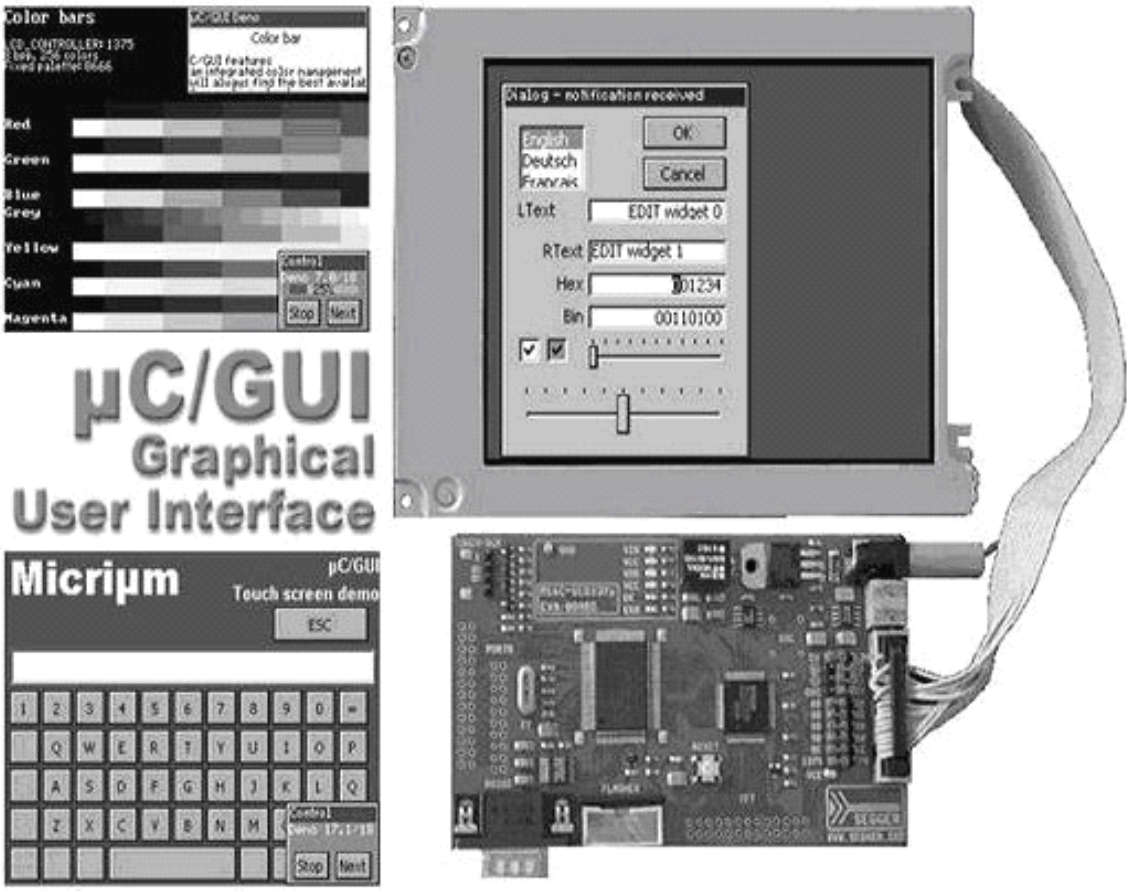

图 1.2　μC/GUI 色彩管理虚拟键盘及视窗模拟器的范例（SOURCE：www.ucos-ii.com ）

虽说与 Symbian OS 或 Windows CE 这类复杂的系统比起来，μC/OS II 家族年轻了许多，但从各类产品的诞生与 Micriμm 持续的研发相关套件看来，一方面代表着固有市场对于目标平台的演化需求，一方面也看出嵌入式中介软件的重要性。而在嵌入式系统微处理器多样化多选择性的时代里，μC/OS II 家族依旧承便于携带的原则，堪称为目前嵌入式软件设计一种典型风格。

下面我们将具体介绍 μC/GUI 的功能特点及使用方法。

1.1 μC/GUI 安装

本实验指导书附带的光盘中也包括 μC/GUI 的所有源代码。同样读者应当具备 80x86，Pentium，或者 Pentium-II 处理器以上的 PC 机，并且运行 DOS 或 Windows95 以上操作系统。至少需要 5Mb 硬盘空间来安装 μC/GUI。请按照以下步骤安装：

打开 μC/GUI 安装文件，将其中文件复制到目标目录下。

在安装之前请一定阅读一下 READ.ME 文件。完成时，用户的目标目录下应该有如 1.3 节的表 1.1，表 1.2 所示的子目录。

另外建议安装 Unicode 字符标准编码计算器，该计算器可以方便 μC/GUI 汉字转换。安装文件在附带光盘中可以找到。

1.2 μC/GUI 结构及功能特点

μC / GUI 是美国 Micriμm 公司出品的一款针对嵌入式系统的优秀图形软件。与 μC / OS 一样，μC / GUI 具有源码公开、可移植、可裁减、稳定性和可靠性高的特点。μC / GUI 函数库为用户程序提供 GUI 接口，包含的函数有文本、数值、二维图形、输入设备以及各种窗口对象。其中，输入设备可以是键盘、鼠标或触摸屏；二维图形包括图片、直线、多边形、圆、椭圆、圆弧等；窗口对象包括按钮、编辑框、进度条、复选框等，可完全产生类似于 Windows 的显示效果。另外，μC / GUI 提供了在 VC 下的仿真库，这使得用户完全可以在 Windows 下仿真 μC / GUI 的各种效果。

μC / GUI 的软件体系结构如图 1.3 所示。

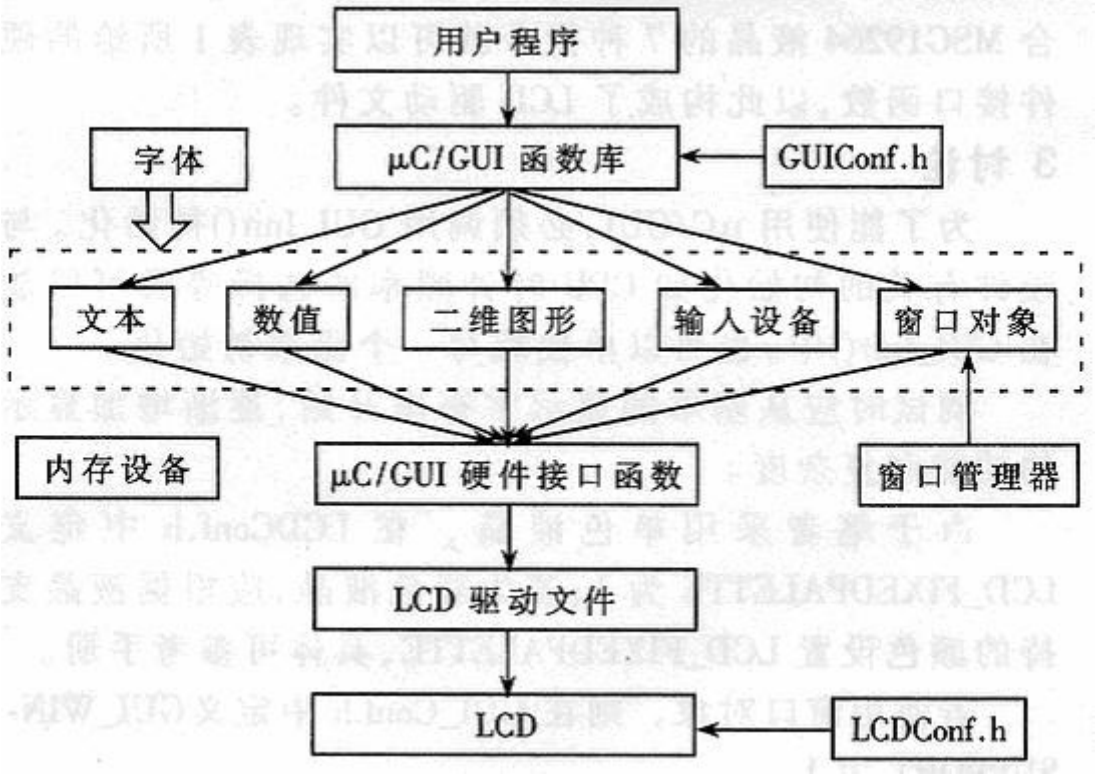

图 1.3　μC / GUI 的软件体系结构

μC/GUI 可选的附加模块允许用户根据应用的需要和性能定制存储器封装。这些模块有：

存储设备：用于避免在绘制重叠图案时显示屏闪烁。

窗口管理：允许用户创建并处理各种大小的窗口

控件：控件是具有对象属性的窗口。比如：按钮，收音机按钮，滚动条，复选框，列表等。它相当于 Windows 下的控件。

对话框：对话框一般是指需要用户输入信息的窗口。它可能包含多个控件，需要用户通过选择输入信息，也可能是消息框和确认按钮，给用户显示一些简单的信息（比如给用户提示或警告）

触摸屏：目前基本的 μC/GUI 包中包含有对触摸屏的支持，该模块包括一个底层驱动，用于处理模拟量输入（模拟量来自于两个 8 位或者更高精度的 A/D 转换通道）以及触摸屏的反跳和校准。该模块是可选的，因为如果用户产品没有触摸屏，就不需要该模块。

反锯齿：通过将前景色和背景色混合得到平滑的曲线和斜线。

多个显示设备:可以在所有层和显示设备上显示窗口和进行画图操作。多个层和多个显示设备的处理方式是一样的(使用相同的 API 例程)，即便是某个嵌入式系统使用了多个显示设备，这些显示设备也只是简单地被当做是多层。μC/GUI 阅读器允许用户查看每个单独的层（显示设备)，但是如果是多层系统，则会查看实际的输出（复合的视图)。

LCD 驱动：Micrium 为目前流行的大多数 LCD 控制器提供驱动。

μC/GUI 产品包中包含有开发工具包，以方便用户的项目开发 μC/GUI，包括 MSVC++ 下的仿真环境。它可以用于在 PC 上编写和测试所有的用户接口（不管采用什么 CPU 和 LCD，所有的例程都与嵌入式应用完全等同)。这将方便调试和开发。产生在 LCD 上的截屏可以作为截图直接加入文档中。

JPEG 采用实际硬件的位图来对目标系统进行显示仿真。

μC/GUI 阅览器，一个独立的程序，即便在调试时也可显示所模拟的 LCD 中的内容。μC/GUI 阅览器界面如图 1.4 所示。

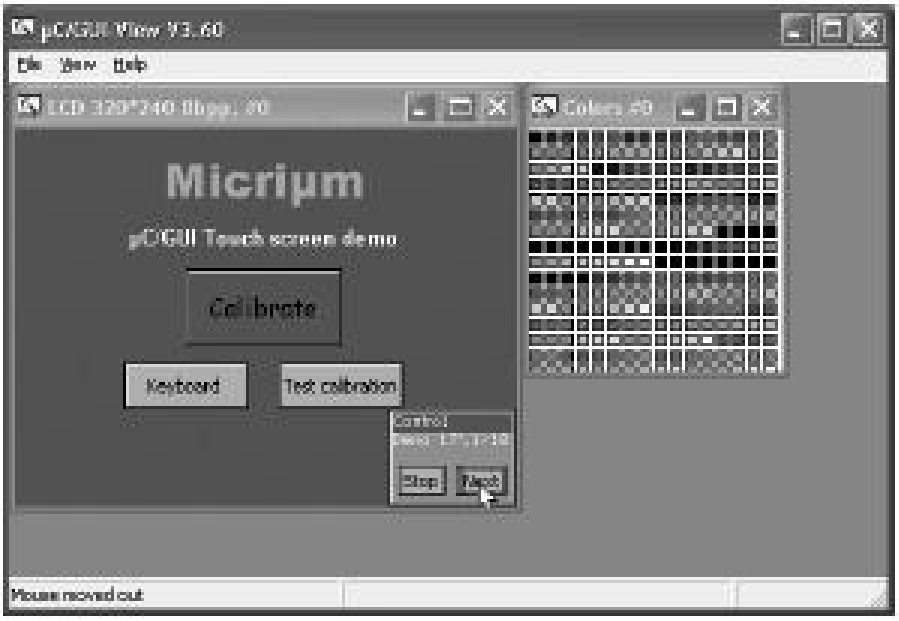

图 1.4　μC/GUI 阅览器界面

μC/GUI 位图转换器：能把所有的位图转换成标准的 C 代码。位图转换器显示要转换的图形。大量例子函数都可以用位图转换器来实现，包括横向或者纵向浏览，旋转位图，以及转换位图的索引或颜色。μC/GUI 位图转换器界面如图 1.5 所示。

嵌入式系统教学平台实验教材

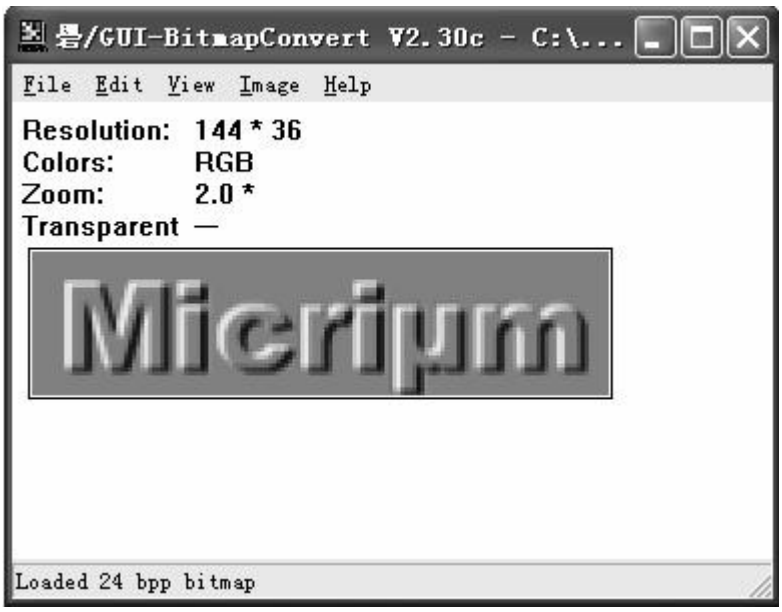

图 1.5　μC/GUI 位图转换器界面

μC/GUI 字体转换器：它可以把任意一种 Windows-PC 下的字体转换成一个"C"文件，在用户的嵌入式应用中可以编译和链接。这样，用 μC/GUI 就可以在 LCD 上显示字体，它支持各种比例大小的字体，同时可以创建外文字体。μC/GUI 字体转换器界面如图 1.6 所示。

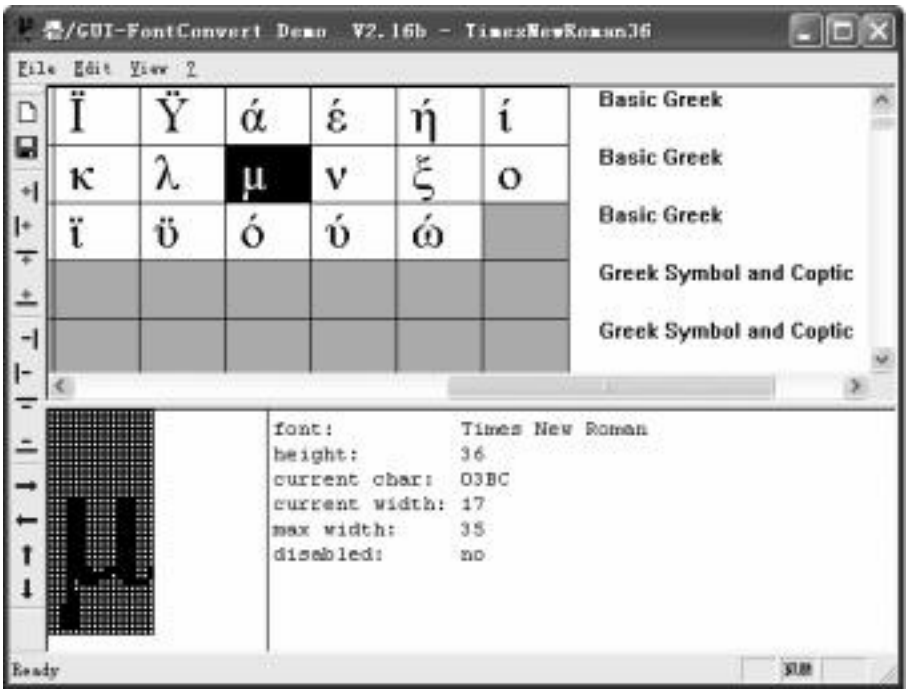

图 1.6　μC/GUI 字体转换器界面

最小化封装：存储器大小要求根据所用的软件部分和目标编译器的效率不同而有所不同，所以不可能指定精确的数值，但是下面的参数适用于典型的系统。

小型系统(没有窗口管理器)

- ✧ RAM：100 bytes
- ✧ 堆栈：500 bytes

- ROM：10-25 kb (与所用的功能有关)

大型系统 (包括窗口管理器和控件)

- RAM：2-6 kb (与要求的窗口数有关)
- 堆栈：1200 bytes
- ROM：30-60 kb (与所用的功能有关)

注意如果用户的应用需要用到很多字体则要求增加 ROM 的大小。上面的所以参数只是粗略估计，并不能完全保证。

1.3 μC/GUI 应用范例

μC/GUI 以源代码的形式发布。在 μC/GUI 产品包中包含了大量源代码的例子，这些例子为 μC/GUI 的一些特性提供了一个简单的介绍。比如：

- 二维图形
- 反锯齿
- 色彩管理
- 窗口/控件/对话框
- 字体
- 仿真 PC 模拟环境下的硬按键……

1.3.1 HelloWorld 范例

接下来介绍一下 uC/GUI 的目录结构和基本配置，并且以编写 HelloWorld 为例介绍使用 uC/GUI 的几个步骤。uC/GUI 文件目录如表 1.1 所示：

表 1.1 uC/GUI 文件目录

GUI\ConvertMono	使用黑白显示设备时，所要使用的灰度转换函数
GUI\ConvertColor	使用彩色显示设备时，使用的色彩转换函数
GUI\Core	uC\GUI 核心代码
GUI\Font	uC\GUI 与字体相关的代码文件
GUI\LCDDriver	LCD 驱动代码文件
GUI\MemDev	内存设备支持文件代码
GUI\Touch	输入设备支持的文件代码
GUI\Widget	uC\GUI 支持的控件代码，包括编辑框、列表框、按钮、选择框等
GUI\WM	uC\GUI 窗口管理部分代码

uC\GUI 的头文件包含在下面几个目录中，如表 1.2 所示：

表 1.2 包含 uC\GUI 的头文件的目录

Config	包含了对 uC\GUI 进行配置的一些文件
GUI\Core	核心部分的头文件
GUI\Widget	控件部分的头文件，在使用控件时需要包含
GUI\WM	窗口管理部分的头文件，在使用窗口管理部分时需要包含

在使用 uC\GUI 时，可以直接使用源代码或者先编译一个库文件，然后将库包含在用户的应用程序中。uC\GUI 中已经提供了批处理，用来制作 uC\GUI 的库文件。

用户使用时可以首先将核心文件、LCD 驱动文件和需要的字体文件包含在用户工程里，然后再根据个人的硬件需要，包含内存设备、输入设备、控件和窗口管理部分。

uC\GUI 的初始化过程通过调用函数 GUI_Init()来完成。

使用 uC\GUI 时，可以按照以下几个步骤来进行：

1．按照需要，定制用户的 uC\GUI；

2．指定硬件设备的地址，编写接口驱动代码。这里需要修改 LCDConf.h 文件；

3．编译，链接，调试例子程序；

4．修改例子程序，并测试，增加用户需要的功能；

5．将 uC\GUI 与操作系统结合；

6．编写自己的应用程序。

下面是 HelloWorld 程序清单：

```
#include "GUI.h"

  void main(void)
  {
    GUI_Init();
    GUI_DispString("Hello world!\n");
    while(1);
  }
```

1.3.2 μC/GUI 字体转换

μC/GUI 字体转换的关键在于用 uC-GUI-FontConvert-Demo 工具把一种 Windows-PC 下的字体转换成一个"C"文件，接下来详细说明其转换步骤。

以添加一种“宋体　常规　16”字体为例。

第一步：在“\uCGUI\Tool”路径下打开 uC-GUI-FontConvert-Demo 字体转换器，屏幕上会弹出一个“Font of options”对话框，按如图 1.7 所示配置点击“OK”确定。

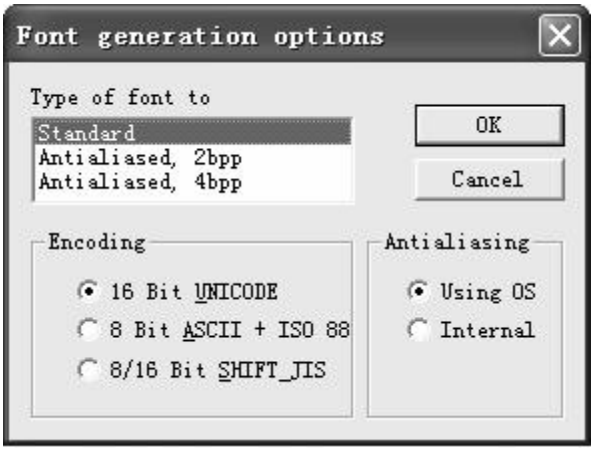

图 1.7 “Font of options”对话框

第二步：在接下来弹出的“字体”对话框里选择我们要添加的字体类型，这里为“宋体　常规　16”如图 1.8 所示，点击“确定”完成。

图 1.8 “字体”对话框

第三步：这时我们看到的是包含在“宋体　常规　16”字体中的所有汉字及字符如图 1.9 所示，图中标记了界面中各部分的含义，其中字符码、符号指针和像素指针是一一对应的。

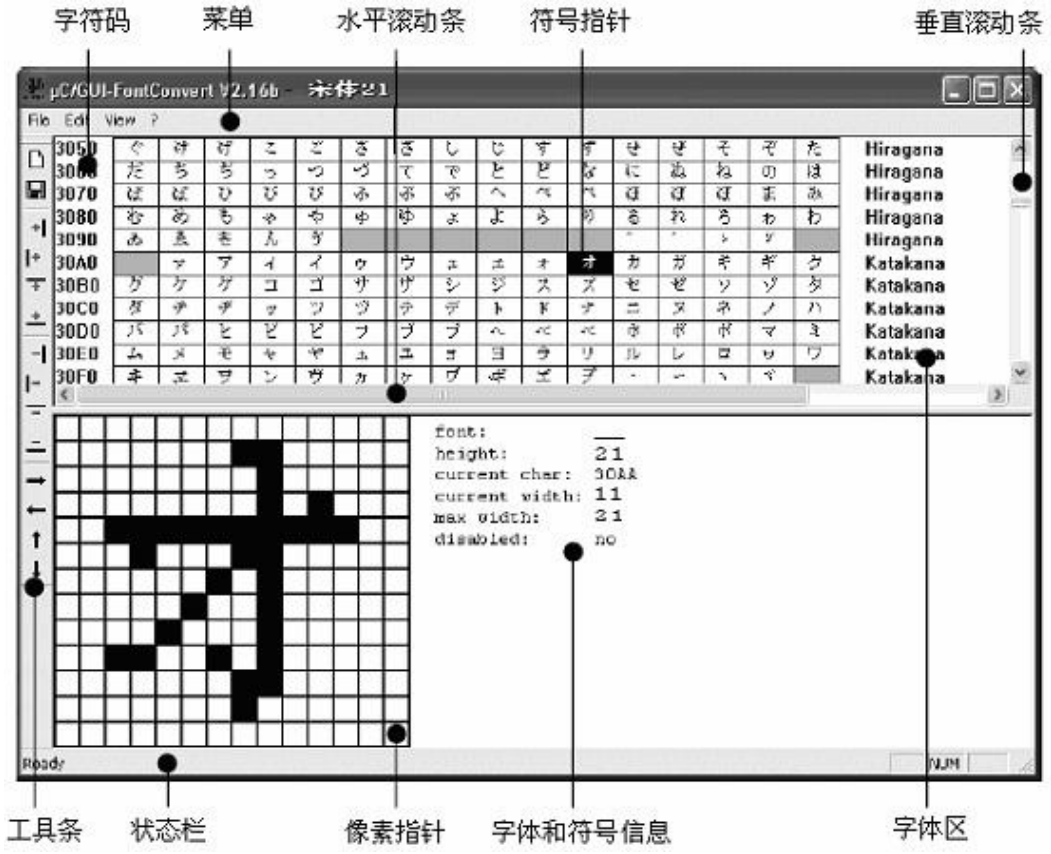

图 1.9　字体启动界面

我们任选一个汉字选择“File→Save As”弹出如图 1.10 所示的对话框，这时意味着 C 文件已经生成，我们把它存放在 Font 目录下，命名为“F16_SONG.c”，点击“保存”。至此，C 文件已经生成。下一步我们需要在头文件中声明它。

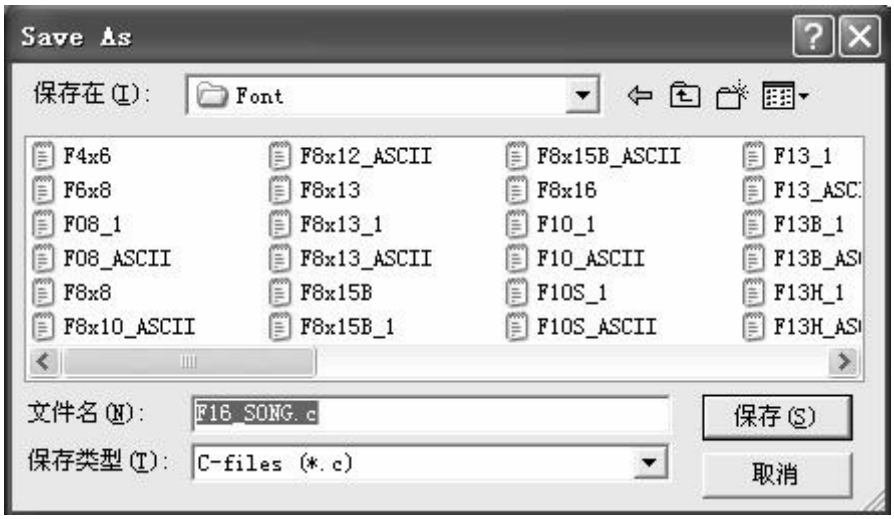

图 1.10　另存对话框

第四步：在＼GUI＼Core＼路径下打开 GUI.h，在其中定义字体部分添加下面一行程序

extern const GUI_FONT GUI_FontF16_SONG;

这样我们就成功的为 μC/GUI 字体中添加了一种中文字体。掌握了这种方法我们还可以给字库中添加日语，并且可以添加不同大小的文字。

1.3.3 μC/GUI 图形转换

μC/GUI 图形转换借助于 uC-GUI-BitmapConvert 工具，μC/GUI 图形转换只支持.bmp 格式的图形文件，其步骤类似于 μC/GUI 字体转换，这里就不做具体讲述了。

1.3.4 小结

μC / GUI 包含的函数有文本、数值、二维图形、输入设备以及各种窗口对象。其中，输入设备可以是键盘、鼠标或触摸屏；二维图形包括图片、直线、多边形、圆、椭圆、圆弧等；窗口对象包括按钮、编辑框、进度条、复选框等，内容极为丰富。因此我们不可能逐一介绍，在此只讲述了字体转换，其他内容读者可参见“uC-GUI_user．pdf”使用手册。

嵌入式系统教学平台实验教材

第二章 μC/GUI 的移植

与 μC / OS 一样，μC / GUI 具有源码公开、可移植、可裁减、稳定性和可靠性高的特点。采用 μC / GUI，开发人员可以很方便地在液晶上显示文本、曲线、图形以及各种窗口对象如按钮、编辑框、滑动条等，可完全产生类似于 Windows 的显示效果。另外，μC / GUI 提供了在 VC 下的仿真库，这使得用户完全可以在 Windows 下仿真 μC / GUI 的各种效果。

采用 μC / GUI，可以大大降低嵌入式系统中显示设计的难度，但 μC / GUI 的使用需针对不同的硬件环境编写相应的驱动程序，即移植，才能在用户的目标系统上运行。uC/GUI 的移植工作主要是针对配置头文件中的宏定义进行修改。这些要修改的宏包括：

1．LCD 宏，定义了显示的尺寸和一些可选择的特性（例如镜像，等等）。

2．LCD 控制器宏，定义了怎样对控制器进行操作。

表 2.1 显示了 uC/GUI 中与移植有关的宏定义：

表 2.1　uC/GUI 中与移植有关的宏定义

类型	宏	缺省值	解释说明
总体配置			
S	LCD_CONTROLLER	…	选择 LCD 控制器
N	LCD_BITSPERPIXEL	…	指定每个像素点的位数
S	LCD_FIXEDPALETTE	…	制定是否采用固定的调色板模式。设置为 0 采用用户设定的调色板。此时 LCD_PHYSCOLORS 宏必须定义。
N	LCD_XSIZE	…	指定行分辨率
N	LCD_XSIZE	…	指定列分辨率
初始化控制器			
F	LCD_INIT_CONTROLLER()	…	LCD 控制器的初始化程序
显示方向			
…… …… ……			
颜色配置			
N	LCD_MAX_LOG_COLORS	65535	数目
A	LCD_PHYSCOLORS	…	当 LCD_FIXEDPALETTE 设置为 0 时，该宏定义了用户所采用的调色板(色彩查找表）
B	LCD_PHYSCOLORS_IN_RAM	0	在用户调色板被定义了的时候可用。将调色板调入 RAM，使得程序在运行当中可以修改调色板颜色

类型	宏	缺省值	解释说明
B	LCD_REVERSE	0	动态颜色反转
B	LCD_SWAP_RB	0	动态红—蓝替换
LCD 显示的放大			
…… …… ……			
简易总线接口配置			
…… …… ……			
全总线接口配置			
F	LCD_READ_MEM(Index)	…	从控制器读出显示缓存中显示内容
F	LCD_READ_REG(Index)	…	从控制器读出配置寄存器内容
F	LCD_WRITE_ MEM(Index,Data)	…	写入数据到显示缓冲区
F	LCD_WRITE_ REG(Index,Data)	…	写入控制器显示配置寄存器的配置值
S	LCD_BUSWIDTH	16	LCD控制器接口总线宽度（位数 8/16）
…… …… ……			
F	LCD_SWAP_BYTE_ORDER	0	采用 16位总线接口时，用来转换大小端模式
LCD 控制器配置：COM/SEG 设置			
…… …… ……			
COM/SEG 查找表			
…… …… ……			
杂项			
…… …… ……			
F	LCD_ON	…	用于点亮 LCD
F	LCD_OFF	…	用于关闭 LCD

2.1 实现方法与步骤

第 1步：配置 uC/GUI

通常是通过修改头文件 LCDConf.h 来配置 uC/GUI。LCDConf.h 中的宏定义描述了 LCD

显示部分硬件特性；根据具体情况修改这些宏定义（例如显示屏的长、宽，每像素点用几位表示，LCD 控制器的类型等参数）。具体程序清单为

```
#ifndef LCDCONF_H
#define LCDCONF_H
/*********************************************************************
*
*          General configuration of LCD
*
**********************************************************************
*/
#define LCD_CONTROLLER      0
#define LCD_XSIZE           240             /*  显示屏的长*/
#define LCD_YSIZE           320             /*  显示屏的宽  */

#define LCD_INTERFACEBITS 8                 /*选择每像素点用 4 或 8 位表示*/
#define LCD_BITSPERPIXEL   16                /* Permitted values here: 1 or 2 */
//#define LCD_PHYSCOLORS 65535

#define LCD_FIXEDPALETTE   565              //RRRRRGGGGGGBBBBB
#define LCD_SWAP_RB        1

#define LCD_TIMERINIT0   1200
#define LCD_TIMERINIT1   1500

#endif /* LCDCONF_H */
```

第 2步：定义 LCD 的底层驱动函数

底层函数包括对 LCD（控制器）的初始化函数，LCD 显示缓冲区的读写函数等，完成对 LCD 显示硬件的直接操作。

对于映射在系统存储器上的 LCD，对显示缓冲区的操作仅需要在 LCDConf.h 中进行定义就可以了。但对于采用 I/O 端口/缓冲区操作的 LCD，就必须定义相应的接口函数了。

第 3步：编译，链接和测试例子代码

uC/GUI 对于单任务和多任务环境下的应用都提供了例子代码。在编程之前，对这些例子代码进行编译、链接和测试，使你能够初步了解这些代码的使用。

第 4步：修改例子程序

对例子代码作少量的修改。逐步添加一些额外的指令，例如显示不同大小的文字，显示多行等等，从而进一步理解代码的应用。

第 5步：把 uC/GUI 的多任务应用加入到操作系统中

如果你的系统有可能用到多个任务同时对显示进行操作，这时就要在 GUITask.C 文件中使用 GUI_MAXTASK 和 GUI_OS 宏。程序清单如下：

```
#include <stddef.h>                 /* needed for definition of NULL */
#include "GUI_Protected.h"
```

```
#include "GUIDEBUG.h"
/*********************************************************************
*
*           Configuration defaults
*
**********************************************************************/
#ifndef GUI_MAXTASK
  #define GUI_MAXTASK (4)
#endif

#if GUI_OS
/*********************************************************************
*
*           Static data
*
**********************************************************************/
static struct {
  U32    TaskID;
  GUI_CONTEXT Context;
} _Save[GUI_MAXTASK];

static int _CurrentTaskNo = -1;
static int _EntranceCnt     =    0;
static U32 _TaskIDLock = 0;
/*********************************************************************
*
*           Static functions
*
**********************************************************************/
static int _GetTaskNo(void) {
  int i;
  for (i=0; i< GUI_MAXTASK; i++) {
    U32 TaskId = GUI_X_GetTaskId();
    if (_Save[i].TaskID == TaskId)
      return i;
    if (_Save[i].TaskID == 0) {
      _Save[i].TaskID = TaskId;
      return i;
    }
  }
  GUI_DEBUG_ERROROUT("No Context available for task ... (increase GUI_MAXTASK)");
  return 0;
}
/*********************************************************************
```

```
*
*           Public functions
*
*********************************************************************************
/
void GUI_Unlock(void) {
  if (--_EntranceCnt == 0) {
    GUI_X_Unlock();
  }
  /* Test if _EntranceCnt is in reasonable range ... Not required in release builds */
  GUI_DEBUG_ERROROUT_IF((_EntranceCnt      <      0),      "GUITASK.c:      GUI_Unlock()
_EntranceCnt underflow ");
}

void GUI_Lock(void) {
  if (_EntranceCnt == 0) {
    GUI_X_Lock();
    _TaskIDLock = GUI_X_GetTaskId();              /* Save task ID */
  } else {
    if (_TaskIDLock != GUI_X_GetTaskId()) {
      GUI_X_Lock();
      _TaskIDLock = GUI_X_GetTaskId();              /* Save task ID */
    }
  }
  if (++_EntranceCnt == 1) {
    int TaskNo = _GetTaskNo();
    if (TaskNo != _CurrentTaskNo) {
      /* Save data of current task */
      if (_CurrentTaskNo>=0) {   /* Make sure _CurrentTaskNo is valid */
        _Save[_CurrentTaskNo].Context = GUI_Context;
      }
      /* Load data of this task */
      GUI_Context = _Save[TaskNo].Context;
      _CurrentTaskNo = TaskNo;
    }
  }
  /* Test if _EntranceCnt is in reasonable range ... Not required in release builds */
  GUI_DEBUG_ERROROUT_IF((_EntranceCnt>12),  "GUITASK.c:  GUI_Lock()  _EntranceCnt
overflow ");
}

void GUITASK_Init(void) {
  int i;
  _CurrentTaskNo =-1;     /* Invalidate */
```

```
  GUI_X_InitOS();
  for (i=0; i<GUI_MAXTASK; i++) {
    _Save[i].Context = GUI_Context;
  }
}
#else

/*********************************************************************************
*
*          Dummy Kernel routines

The routines below are dummies in case configuration tells us not
to use any kernel. In this case the routines below should
not be required, but it can not hurt to have them. The linker
will eliminate them anyhow.

**********************************************************************************/
void GUI_Unlock(void) {}
void GUI_Lock(void) {}
void GUITASK_Init(void) {}
void GUITASK_StoreDefaultContext(void) {}

#endif
```

第 6步：采用 uC/GUI 编写自己的应用

到这一步已经对怎样使用 uC/GUI 有一个清楚的了解了。接下来考虑如何采用 uC/GUI 提供的函数来构建自己的应用，并通过阅读手册来获得各函数更详细的功能和使用上的信息。

第三部分 实验部分

实验一 μC/OS-II 移植实验

1.1 实验目的

1．了解 μC/OS-II 内核的基本原理和主要结构；
2．掌握将 μC/OS-II 内核移植到 ARM 处理器上的基本方法；
3．掌握 μC/OS-II 下基本多任务应用程序的编写。

1.2 实验内容

1．学习 μC/OS-II 在 ARM 处理器上的移植过程；
2．编写简单的多任务应用程序，实现执行任务号 0~9 的依次循环显示。

1.3 预备知识

1．了解嵌入式操作系统的构架，以及具体的 μC/OS-II 的组成；
2．了解操作系统的移植方法。

1.4 实验设备

1．S3C2410 嵌入式开发板，JTAG 仿真器，串口连接线。
2．软件：PC 机操作系统 Win98、Win2000 或 WinXP，ADS1.2 集成开发环境，仿真器驱动程序，超级终端通讯程序。

1.5 基础知识

1．μC/OS-II 移植知识
参见 μC/OS-II 部分的移植一章内容。
2．实验说明
一个最基本的 μC/OS 多任务应用程序的范例主要包括如下两个部分：
1) C 语言入口函数
函数 Main 为 C 语言入口函数，所有的 C 程序从这里开始运行。在该函数中将进行如下操作：
a．调用函数 ARMTargetInit 初始化 ARM 处理器；
b．调用 OSInit 进行操作系统初始化；
c．调用 OSTaskCreate 函数两个任务：TaskLED 和 TaskSEG;
d．调用 ARMTargetStart 函数启动时钟节拍中断；

e．调用 OSStart 启动系统任务调度。

2)任务处理函数

在多任务中，每个任务都对应一个入口函数。这些函数必须为死循环，它们每隔 100 个时钟节拍向串口打印一串字符。

1.6 实验步骤

1．新建一个工程 EX1，将模板中的 Core（C source 和 Core（Assemble）文件夹中的文件加入到工程 EX1 中，修改工程设置。

2．新建一个文件 EX1．C，并将其添加到 EX1 工程的 App 文件夹中。

3．在主函数中建立十个任务，每个任务不停地往串口上打印自己的任务号。

4．编译工程 EX1，如果出错，进行修改后再编译。

5．将 EX1 下载并运行，看结果，正确的结果为 0~9 的依次循环输出，在超级终端中的显示如图 1.1 所示：

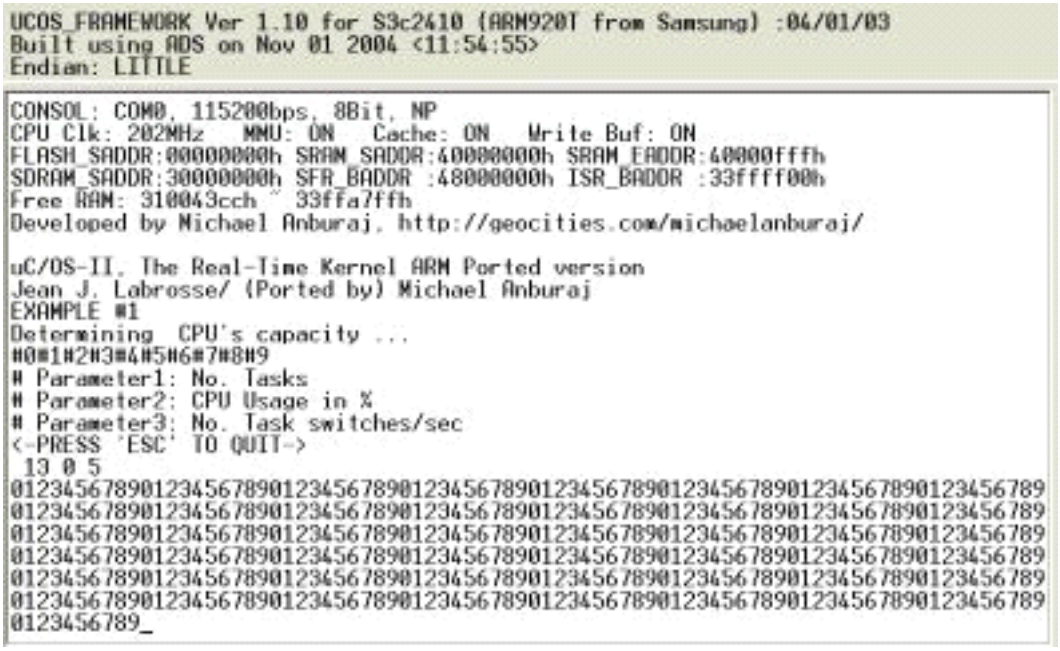

图 1.1 超级终端显示的运行结果

1.7 实验报告要求

1．μC/OS 是一个小型的操作系统，它由那些模块组成？具有哪些功能？

2．简述 μC/OS 是如何进行任务调度的？

实验二 μC/OS-II 任务间通讯实验

2.1 实验目的

1．掌握 μC/OS-II 操作系统下邮箱当作二值信号量的方法及应用
2．掌握 μC/OS-II 操作系统下任务间通讯的方法

2.2 实验内容

1． 使用邮箱实现任务之间的通讯。

2.3 预备知识

1．了解操作系统任务间通讯的机制
2．学习 μC/OS-II 中邮箱的用法

2.4 实验设备

1．S3C2410 嵌入式开发板，JTAG 仿真器，串口连接线。
2．软件：PC 机操作系统 Win98、Win2000 或 WinXP，ADS1.2 集成开发环境，仿真器驱动程序，超级终端通讯程序。

2.5 基础知识

1．μC/OS-II 任务之间的通讯与同步方式在 μC/OS-II 中，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讯。

- ✧ 利用宏 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()来关闭中断和打开中断。当两个任务或者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
- ✧ 利用函数 OSSchedLock()和 OSSchekUnlock()对 μC/OS-II 中的任务调度函数上锁和开锁
- ✧ 利用信号量，邮箱，队列

2．用邮箱作二值信号量

一个邮箱可以被用作二值的信号量。首先，在初始化时，将邮箱设置为一个非零的指针（如 void *1）。这样，一个任务可以调用 OSMboxPend()函数来请求一个信号量，然后通过调用 OSMboxPost()函数来释放一个信号量。程序清单 L6.19 说明了这个过程是如何工作的。如果用户只需要二值信号量和邮箱，这样做可以节省代码空间。这时可以将 OS_SEM_EN 设置为 0，只使用邮箱就可以了。以下是将邮箱用作二值信号量的代码：

```
OS_EVENT *MboxSem;
…
```

```
void Task1 (void *pdata)
{
  INT8U err;
  ...
  for (;;) {
     OSMboxPend(MboxSem, 0, &err);     /*  获得对资源的访问权  */
      …
      … /*  任务获得信号量,对资源进行访问    */
     OSMboxPost(MboxSem, (void*)1);   /*  释放对资源的访问权  */
     }
  }
```

3．μC/OS-II 中使用邮箱邮箱可使一个任务或者中断服务子程序向另一个任务发送一个指针型的变量。该指针指向一个包含了特定“消息”的数据结构。通过调用 OSMboxCreate()函数来创建邮箱，并指定指针的初始值。如果使用邮箱的目的是用来通知一个事件的发生（发送一条消息），那么就要初始化该邮箱为 NULL，因为在开始时，事件还没有发生。如果用户用邮箱来共享某些资源，那么就要初始化该邮箱为一个非 NULL 的指针。在这种情况下，邮箱被当成一个二值信号量使用。使用邮箱同样可以实现任务间的同步。

通过 OSMboxPost()函数发送一个消息到邮箱中，通过 OSMboxPend()函数等待一个邮箱中的消息，如果邮箱中没有可用的消息，OSMboxPend()的调用任务就被挂起，直到邮箱中有了消息或者等待超时。

4．μC/OS-II 中使用邮箱进行任务之间通讯下面的代码每 100 个时钟节拍从 TaskSEG 中发送一个字符串，在 TaskLED 中接收并打印出来。

```
void TaskLED(void *Id)
{
        char      *Msg;
        INT8U    err;
        for (;;)
{
                /* wait for a message from the input mailbox */
                Msg = (char *)OSMboxPend(Mbox1, 0, &err);
                UHALr_printf(Msg);                                  /*print task´s id*/
        }
}
void TaskSEG(void *Id)
{
        char      Msg[100];
        INT8U    err;
        int         nCount = 0;
        for (;;)
        {
                /* post the input message to the output mailbox */
                sprintf (Msg，“TaskSEG %d”, nCount++);
                OSMboxPost (Mbox1, Msg);
                OSTineDly(100);
```

```
        }
    }
```

2.6 实验步骤

1．复制前面我们已经移植好的 ucos 实验，改名为 EX2．mcp，其他的配置文件不变

2．更改主函数中的 APP_vMain 一段程序，建立两个任务，使用信号量实现任务间的同步，同时使用邮箱实现任务之间的通讯，程序另存为 EX2．c。

3．编译工程 EX2 如果出错，进行修改后再编译。

4．更改超级终端，步骤为点击工具栏最右边的 Properties（属性）→ASCII Sending→选择 Echo typed chsrscters locally（回显本地字符）→OK 确定。如图 2.1 所示。更改的目的是让实验者可以对比到输入、输出的一致性。

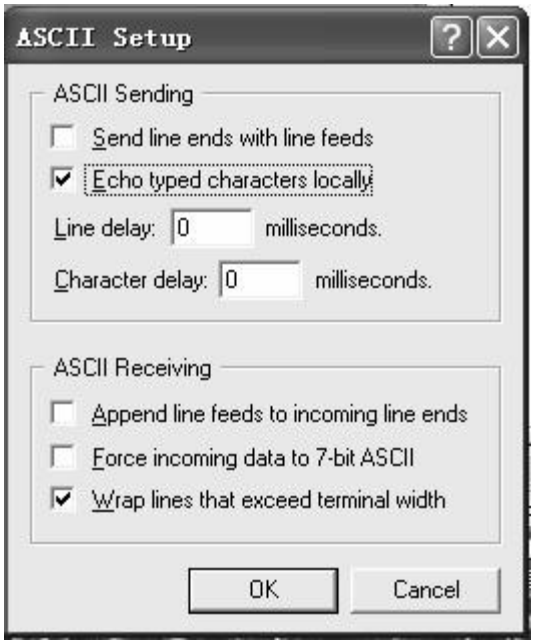

图 2.1　更改超级终端

5．将 EX2 下载并运行，输入一行字符串，以回车结束，在超级终端中的查看运行结果如图 2.2 所示：

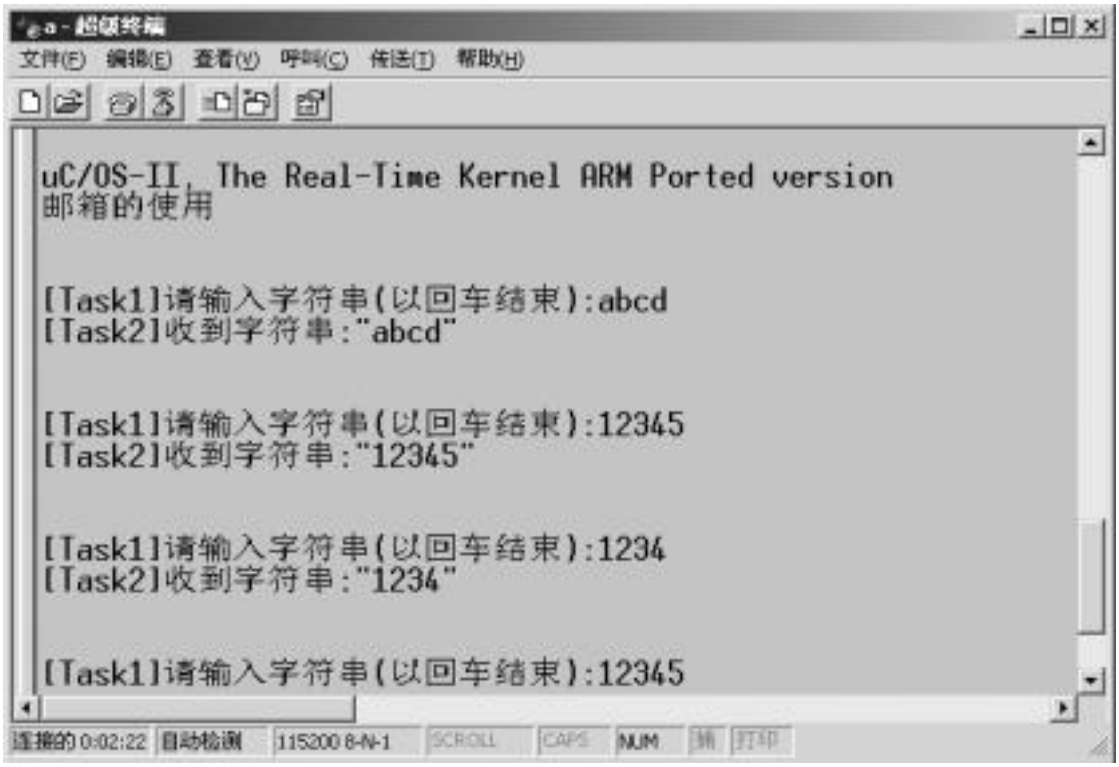

图 2.2 超级终端显示的运行结果

2.7 实验报告要求

1. 简述邮箱的作用?
2. 简述邮箱在 μC/OS 中是如何实现的?

嵌入式系统教学平台实验教材

实验三 μC/OS-II 多任务实验

3.1 实验目的

1．掌握 μC/OS-II 操作系统下使用多任务问题

3.2 实验内容

1．在 μC/OS-II 中的启动多任务环境。

3.3 预备知识

1．学习 μC/OS-II 操作系统下多任务应用程序的编写方法
2．了解操作系统任务调度的机制

3.4 实验设备

1．S3C2410 嵌入式开发板，JTAG 仿真器，串口连接线。
2．软件：PC 机操作系统 Win98、Win2000 或 WinXP，ADS1.2 集成开发环境，仿真器驱动程序，超级终端通讯程序。

3.5 基础知识

1．μC/OS-II 中的多任务多任务运行的实现实际上是靠 CPU(中央处理单元)在许多任务之间转换、调度。CPU 只有一个，轮番服务于一系列任务中的某一个。多任务运行很像前后台系统，但后台任务有多个。多任务运行使 CPU 的利用率得到最大的发挥，并使应用程序模块化。在实时应用中，多任务化的最大特点是，开发人员可以将很复杂的应用程序层次化。使用多任务，应用程序将更容易设计与维护。

一个任务，也称作一个线程，是一个简单的程序，该程序可以认为 CPU 完全只属该程序自己。实时应用程序的设计过程，包括如何把问题分割成多个任务，每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，有它自己的一套 CPU 寄存器和自己的栈空间(如图 3.1 所示)。

2．在 μC/OS-II 中的启动多任务环境。

在调用 OSStart( )之前必须先调用 OSInit ( )。在用户程序中 OSStart( )只能被调用一次。第二次调用 OSStart( )将不进行任何操作。程序如下：

```
void main(void)
{
                                        /*  用户代码                */
…
OSInit();                               /*  初始化  uC/OS-II        */
```

```
…                                          /*  用户代码                  */
OSStart();                                 /*  启动多任务环境*/
    /*写在这里的代码永远不会执行*/
}
```

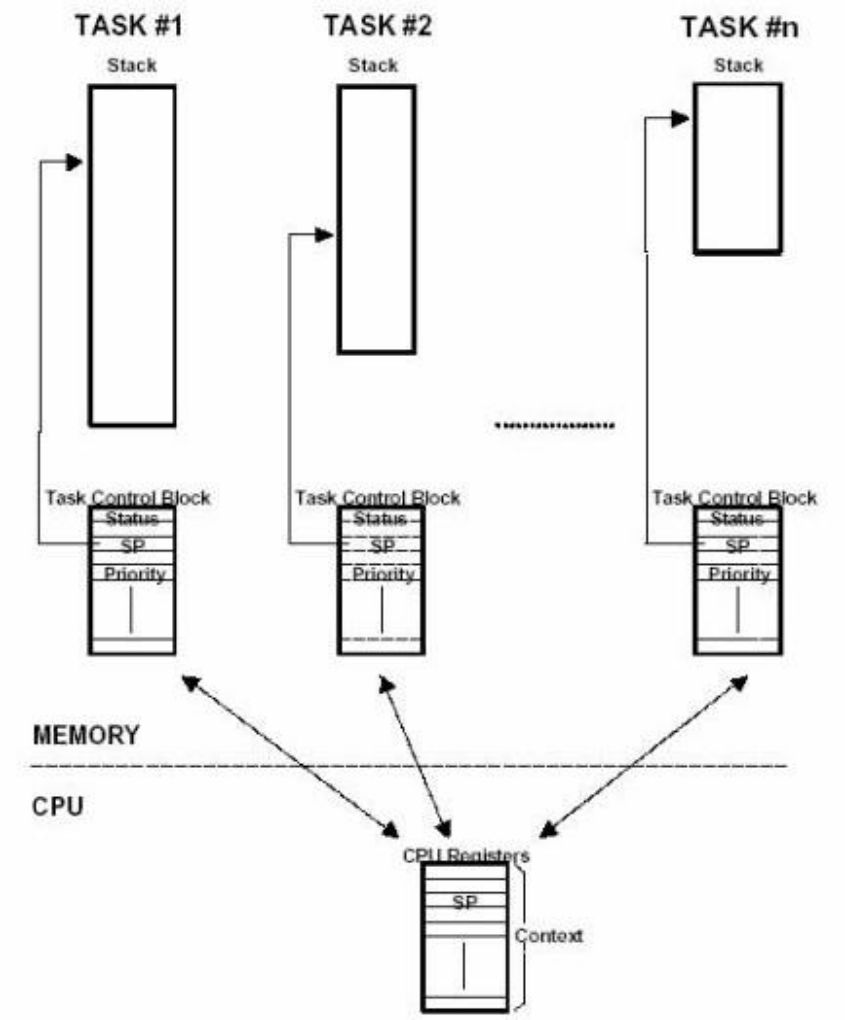

图 3.1　多任务

3.实验说明

主函数为一个菜单程序，包括菜单提示，另外还包含了几段程序的入口。

入口一：“+” 此时会添加一个任务

入口二：“-” 此时会减去一个任务

入口三：“ESC” 退回到主菜单

每个任务为打印自己的任务号，循环执行；通过打印号码可以看到正在执行任务的个数。

3.6 实验步骤

1．复制前面我们已经移植好的 ucos 实验，改名为 EX3．mcp，其他的配置文件不变。

2．更改主函数中的程序，程序另存为 EX3．C，并添加两个函数，一个执行添加任务，一个执行删除任务。

3．编译工程 EX3 如果出错，进行修改后再编译。

4．将 EX3 下载并运行，在超级终端中的查看运行结果如图 3.2 所示：

图 3.2 超级终端显示结果

注释：一串 0 代表此时只有一个任务，按“+”后添加了一个任务，开始打印 10，…同理，最多可以添加到 10 个任务，打印 9876543210。按“-”后删除一个任务，变为 9 个任务，则打印 876543210…以此类推到 0。另外，按“ESC”可以返回到测试菜单。

3.7 实验报告要求

1． 简述 μC/OS-II 中如何启动多任务环境。

嵌入式系统教学平台实验教材

实验四 μC/OS-II +μC/GUI 实验

4.1 实验目的

1．掌握将 μC/OS-II 及 μC/GUI 内核移植到 ARM 处理器上的基本方法；
2．掌握 μC/OS-II 下基本多任务应用程序的编写。
3．掌握 μC/GUI 的简单编程。

4.2 实验内容

利用 μC/OS-II 的多任务功能建立 3 个任务，任务 1 跑马灯，任务 2 是 μC/OS-II 的统计任务，任务 3 在 μC/GUI 的液晶屏中显示图形及文字。

4.3 预备知识

1．学习 μC/OS-II 操作系统下多任务应用程序的编写方法
2．学习 μC/GUI 编程知识，及其移植时 LED 配置部分程序的改写

4.4 实验设备

1．S3C2410 嵌入式开发板，JTAG 仿真器，串口连接线。
2．软件：PC 机操作系统 Win98、Win2000 或 WinXP，ADS1.2 集成开发环境，仿真器驱动程序，超级终端通讯程序。

4.5 基础知识

1．μC/OS-II 的多任务

首先 μC/OS-II 建立 3 个任务，任务 1 跑马灯，任务 2 是 μC/OS-II 的统计任务，任务 3 在 μC/GUI 的液晶屏中显示图形及文字。我们主要看任务 3。

2．任务 3：μC/GUI 显示

μC/GUI 的任务程序如以下代码所示：

```
void Task3 (void *data)
{
        U8 err;

        while(1)
        {
            CONSOL_Printf("Task3    GUI Task!    \n");       /* Display #tasks running */
            OSTimeDly(5);                                    /* Delay 5 clock tick */
            GUIMainTask();
```

```
            OSTimeDly(1000);                                    /* Delay 5 clock tick */
        }
}
```

3．主任务：μC/OS-II 建立多任务环境

任务程序如以下代码所示：

```
void TaskStart (void *data)
{
    U8 i;
    char key;

    data = data;                                        /* Prevent compiler warning */

    CONSOL_Printf("uC/OS-II, The Real-Time Kernel ARM Ported version\n");
    CONSOL_Printf("Task1 #1\n");

    OSStatInit();                                       /* Initialize uC/OS-II's statistics */

    TaskData[0] = '1' ;                  /* Each task will display its own letter */
    TaskData[1] = '2' ;                  /* Each task will display its own letter */
    OSTaskCreate(Task1, (void *)0, (void *)&TaskStk[0][TASK_STK_SIZE - 1], 1);
    OSTaskCreate(Task2, (void *)0, (void *)&TaskStk[1][TASK_STK_SIZE - 1], 2);
    OSTaskCreate(Task3, (void *)0, (void *)&TaskStk[2][TASK_STK_SIZE - 1], 3);

    CONSOL_Printf("<-PRESS 'ESC' TO QUIT->\n");
    while(1)
    {
        CONSOL_Printf(" %d", OSTaskCtr);       /* Display #tasks running */
        CONSOL_Printf(" %d", OSCPUUsage);      /* Display CPU usage in % */
        CONSOL_Printf(" %d\n", OSCtxSwCtr); /* Display #context switches /s*/
        OSCtxSwCtr = 0;

        if(CONSOL_GetChar(&key) == True)
          {                                  /* See if key has been pressed */
                if(key == 0x1B)              /* Yes, see if it's the ESCAPE key */
                        while(1);            /* Stay here for ever */
          }
        OSTimeDlyHMSM(0, 0, 1, 0);           /* Wait one second */
    }
}
```

4.6 实验步骤

1．复制前面我们已经移植好的μC/OS-II 实验，改名为 EX4．mcp，并添加上已经移植好的μC/GUI 配置文件。

2．更改主函数，使它建立 3 个任务，任务 1 跑马灯，任务 2 是μC/OS-II 的统计任务，任务 3 在μC/GUI 的液晶屏中显示图形及文字。程序另存为 EX2．c。

3．编译工程 EX4 如果出错，进行修改后再编译。

4．将 EX4 下载并运行，实验现象为 3 个：

（1）跑马灯现象

（2）在液晶屏中看到μC/GUI 显示图形及文字

（3）在超级终端中显示 μC/OS-II 运行结果及 LED 亮灯位置。超级终端显示如图 4.1 所示：

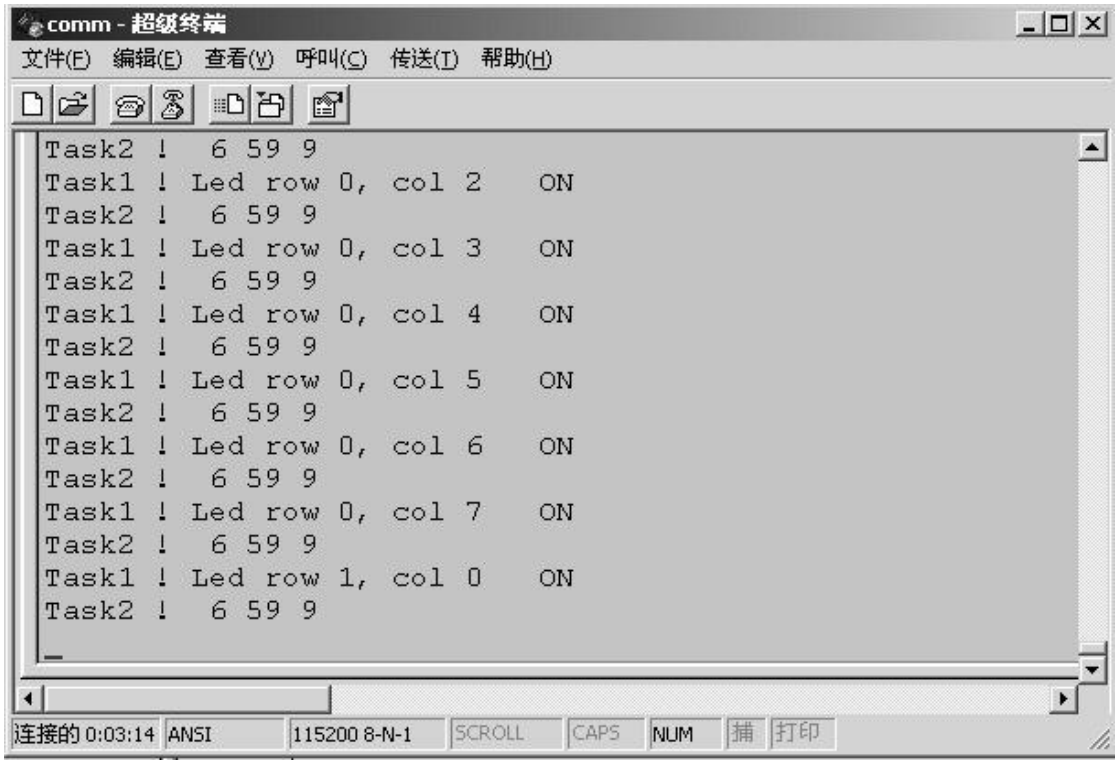

图 4.1 超级终端显示的运行结果

4.7 实验报告要求

1．试着添加其他的任务，验证μC/OS-II 及μC/GUI 丰富的功能