嵌入式RTOS讲座（4）

在80X86上移植μC/OS-II

■ 清华MOTOROLA 单片机应用开发研究中心 邵贝贝 许庆丰 王若鹏

［编者按］ RTOS是开发计算机嵌入式应用产品的有力工具，研究和掌握RTOS的思想方法有益于提高开发人员的水平，从而缩短产品开发周期、提高产品质量。从第7期起，学习园地连载《嵌入式RTOS讲座》，主要内容如下：什么是嵌入式RTOS；RTOS中的一些基本概念；介绍一个源码公开的实时内核；在PC机上实现该实时内核的运行；让该实时内核在8位单片机上运行；如何将该实时内核移植到32位CPU上。

一、μC/OS-II的移植

从这一讲开始，介绍如何将μC/OS-II移植到不同的处理器上。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。为了方便移植，μC/OS-II的大部分代码是用C语言写的，但与处理器相关的代码仍需要用汇编语言写。 由于μC/OS-II在设计时就已经充分考虑了可移植性，所以移植工作并不复杂。

要移植μC/OS-II，目标处理器必须满足以下要求：

(1) 处理器的C编译器能产生可重入代码；

(2) 用C语言就可以打开和关闭中断；

(3) 处理器支持中断，并且能产生定时中断(通常在10~100Hz之间)；

(4) 处理器支持足够的RAM（几千字节），作为多任务环境下的任务堆栈；

(5) 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。

μC/OS-II是面向中小型嵌入式系统的，如果包含全部的功能模块（信号量、消息邮箱、消息队列等及相关函数），编译后的μC/OS-II的内核大约有6~10KB；如果只保留最核心的代码，则最小可压缩到2KB以下。RAM的占用与系统中的任务数有关，任务堆栈要占用大量的RAM空间，堆栈的大小取决于任务的局部变量、缓冲区大小及可能的中断嵌套的层数。对于一般的中小系统，任务堆栈可以取几百字节到几千字节。所以，所要移植的系统中必须有足够的RAM资源；而像MOTOROLA 6805系列的处理器，由于RAM资源太少，且系统总线不能外扩，不能满足上面的第（4）条和第（5）条要求，所以μC/OS-II不能在这类处理器上运行。

如果用户理解了处理器和C编译器的技术细节，移植μC/OS-II的工作实际上是非常简单的。前提是您的处理器和编译器满足了μC/OS-II的要求，并且已经有了必要工具。移植工作包括以下几个内容：

◇ 用#define设置一个常量的值(OS_CPU.H)；

◇ 声明10个数据类型(OS_CPU.H)；

◇ 用#define声明3个宏(OS_CPU.H)；

◇ 用C语言编写6个简单的函数(OS_CPU_C.C)；

◇ 改写4个汇编语言函数(OS_CPU_A.ASM)。

根据处理器的不同，一个移植实例可能需要编写或改写50~300行的代码。移植μC/OS-II需要一个C编译器，并且是针对用户用的CPU的。C编译器还要支持插入汇编语言程序，从而可以插入合适

的处理器指令来允许和禁止中断。

如果已经全部安装了μC/OS-II，可在硬盘上\SOFTWARE\uCOS-II目录下找到μC/OS-II提供的两个现成的移植实例，是针对Intel/AMD的80X86处理器和MOTOROLA 68HC11微控制器。与硬件相关的代码都在下述几个文件中：OS_CPU.H，OS_CPU_C.C，OS_CPU_A.ASM。汇编语言文件OS_CPU_A.ASM是可选择的，因为某些C编译器允许用户在C语言中插入汇编语言，所以用户可以将所需的汇编语言代码直接放到OS_CPU_C.C中。

由于在第4讲中第一次涉及到μC/OS-II的移植，笔者将尽可能详细地介绍移植过程中可能遇到的问题。由于篇幅所限，不可能给出所有的源代码，但在文章中会尽量将要点讲述清楚。本篇中所介绍的关于80X86的源代码在μC/OS-II中都有提供，读者可以从http://www.ucos-II.com上获取。

二、工具和运行环境

下面将介绍如何将μC/OS-II移植到Intel 80X86系列CPU上。μC/OS-II的作者已经为80X86的移植编写了代码和例程，采用的是Borland C/C++ V3.1和Borland Turbo Assembler汇编器完成程序的移植和测试。它可以产生可重入的代码，同时支持在C程序中嵌入汇编语句。编译完成后，程序可在PC机上运行。作者所写的例程是在一台Pentium-II计算机上完成的，操作系统是Microsoft Windows 95。实际上编译器生成的是DOS可执行文件，在Windows的DOS窗口中运行。

这里所介绍的移植和代码都是针对80X86的实模式的，且编译器在大模式下编译和连接。将要介绍的移植过程适用于所有与80X86兼容的CPU，如AMD、Cyrix、NEC (V-系列)等等。

三、移植中需修改的文件

在安装μC/OS-II的时候，安装程序将把和硬件相关的、针对Intel 80X86的代码安装到\SOFTWARE\uCOS-II\Ix86L目录下。代码是80X86实模式，且在编译器大模式下编译的。移植部分的代码可在下述文件中找到：OS_CPU.H, OS_CPU_C.C和OS_CPU_A.ASM。同时需要修改INCLUDES.H文件，将上述文件包含进去。

1. INCLUDES.H 文件

INCLUDES.H是主头文件，在所有后缀名为.C的文件的开始都包含INCLUDES.H文件。对于不同类型的处理器，用户需要改写INCLUDES.H文件，增加自己的头文件，但必须加在文件末尾。为80X86编写的INCLUDES.H文件中增加有：

```
#include "software\ucos-ii-ix861\os_cpu.h"
#include "software\blocks\pc\source\pc.h"
#include "software\ucos-ii\source\usos_ii.h"
```

2. OS_CPU.H 文件

OS_CPU.H文件中包含与处理器相关的常量、宏和结构体的定义，主要有如下定义：

```
typedef unsigned char INT8U;      /*无符号8位数(1)   */
typedef signed   char INT8S;      /*带符号8位数      */
typedef unsigned int  INT16U;     /*无符号16位数     */
typedef signed   int  INT16S;     /*带符号16位数     */
typedef unsigned long INT32U;     /*无符号32位数     */
typedef signed   long INT32S;     /*带符号32位数     */
typedef float         FP32;       /*单精度浮点数     */
typedef double        FP64;       /*双精度浮点数     */
typedef unsigned int  OS_STK; /*堆栈入口宽度为16位*/
#define OS_STK_GROWTH 1       /*堆栈由高地址向低
                                地址增长*/
#define OS_TASK_SW() asm INT  uCOS
OS_CPU_EXT  INT8U  OSTickDOSCtr;
                  /*为调用DOS时钟中断而定义的计数器*/
```

OS_CPU.H文件中定义了与硬件相关的基本信息，有如下几类。

（1）数据类型

由于不同的处理器有不同的字长，μC/OS-II的移植需要重新定义一系列的数据结构。使用

Borland C/C++编译器，整数（int）类型数据为16位，长整形（long）为32位。为了读者方便起见，在源代码中提供了浮点类型的定义。由于在80X86实模式中堆栈都是按字进行操作的，没有字节操作，所以Borland C/C++编译器中堆栈数据类型OS_STK声明为16位。所有的堆栈都必须用OS_STK声明。

（2）代码临界区

临界区是一个非常重要的概念。与其他实时系统一样，μC/OS-II在进入系统临界代码区之前要关闭中断，等到退出临界区后再打开,从而保护核心数据不被多任务环境下的其他任务或中断破坏。Borland C/C++支持嵌入汇编语句，所以加入关闭/打开中断的语句是很方便的。μC/OS-II定义了两个宏用来关闭/打开中断：OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()。对于80X86有两种开关中断的方法，如下所述的方法1和方法2。μC/OS-II作者所提供的例程中采用了方法1。

方法1：

第一种方法，也是最简单的方法，是直接将OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()定义为处理器的关闭（CLI）和打开（STI）中断指令。但这种方法有一个隐患，如果在关闭中断后调用μC/OS-II函数，当函数返回后，中断将被打开！严格意义上的关闭中断应该是执行OS_ENTER_CRITICAL()后中断始终是关闭的，方法1显然不满足要求。但方法1的最大优点是简单，执行速度快（只有一条指令），在此类操作频繁的时候更为突出。如果在任务中并不在意调用函数返回后是否被中断，推荐用户采用方法1。

方法2：

执行OS_ENTER_CRITICAL()的第二种方法是先将中断关闭的状态保存到堆栈中，然后关闭中断。与之对应的OS_EXIT_CRITICAL()的操作是从堆栈中恢复中断状态。采用此方法，不管用户是在中断关闭还是允许的情况下调用μC/OS-II中的函数，在调用过程中都不会改变中断状态。如果用户在中断关闭的情况下调用μC/OS-II函数，其实是延长了中断响应时间。尤其是一些涉及到时间操作的函数，由于需要定时中断，关闭中断有可能使系统崩溃，必须十分小心。所以建议用户调用μC/OS-II的系统函数之前打开中断。

（3）堆栈增长方向

绝大多数的微处理器和微控制器的堆栈是从高地址往低地址递减的，但是某些处理器是用另外一种方式工作的。μC/OS-II设计成两种情况都可以处理，只要在结构常量OS_STK_GROWTH中指定堆栈的生长方式(如下所示)就可以了。

置OS_STK_GROWTH为0，表示堆栈从低地址往高地址递增。

置OS_STK_GROWTH为1，表示堆栈从高地址往低地址递减。

80X86处理器的堆栈是由高地址向低地址方向递减的，所以常量OS_STK_GROWTH必须设置为1。

（4）OS_TASK_SW()函数的定义

在μC/OS-II中，OS_TASK_SW()用来实现任务切换。就绪任务的堆栈初始化应该模拟一次中断发生后的样子，堆栈中应该按进栈次序设置好各个寄存器的内容。OS_TASK_SW()函数模拟一次中断过程，在中断返回的时候进行任务切换。80X86提供了256个软中断源可供选用，中断服务程序（ISR）（也称为例外处理过程）的入口点必须指向汇编函数OSCtxSw()。

在μC/OS-II所提供的例程中使用的是中断号128（0x80）。

（5）设定时钟节拍的发生频率

实时系统中时钟节拍的作用非常重要，凡是涉及到计时的操作都是在时钟中断中完成的。时钟节拍的发生频率应该设置为10~100 Hz。为了方便计算通常设为整数，但由于在PC中，系统缺省的时钟节拍频率是18.20648Hz，在μC/OS-II作者提供的例程中设定时钟节拍频率为200 Hz（间隔5ms）。一方面200 Hz近似18.20648Hz的11倍，可以经过11次延

时再调用DOS中断；另一方面，由于需要在DOS环境下运行，在DOS中，有些操作要求时钟间隔为54.93ms，设定的间隔5ms也可以满足要求。

在文件OS_CPU.H的末尾声明了一个8位变量OSTickDOSCtr，将保存时钟节拍发生的次数，每发生11次，调用DOS的时钟节拍函数一次，从而实现与DOS时钟的同步。OSTickDOSCtr是专门为PC环境而声明的，如果在其他非PC的系统中运行μC/OS-II，就不用这种同步方法，直接设定时钟节拍发生频率即可。

3. OS_CPU_A.ASM文件

μC/OS-II的移植需要用户改写OS_CPU_A.ASM中的4个函数：OSStartHighRdy()、OSCtxSw()、OSIntCtxSw()和OSTickISR()。

（1）OSStartHighRdy()函数

该函数由SStart()函数调用，功能是运行优先级最高的就绪任务。在调用OSStart()之前，用户必须先调用OSInit()，并且已经至少创建了一个任务。为了启动任务，OSStartHighRdy()首先找到当前就绪的优先级最高的任务（OSTCBHighRdy中保存有优先级最高任务的任务控制块——OS_TCB的地址），并从任务的任务控制块（OS_TCB）中找到指向堆栈的指针，然后运行指令POP DS、POP ES、POPA和IRET。由于任务创建是堆栈的结构就是按中断后的堆栈结构初始化的，执行IRET指令后就切换到了新任务（有关μC/OS-II的任务切换机制，请参考本系列讲座的第3讲）。OSStartHighRdy()的汇编语言代码如下所示：

```
_OSStartHighRdy  PROC FAR
MOV   AX, SEG _OSTCBHighRdy
                          ；获取最高优先级任务的TCB地址
MOV   DS, AX              ；
LES   BX, DWORD PTR DS:_OSTCBHighRdy；
MOV   SS, ES:[BX+2]  ；设置堆栈指针
MOV   SP, ES:[BX+0]  ；
POP   DS                  ；恢复任务环境
POP   ES
POPA
IRET                      ；中断返回，运行新任务
```

（2）OSCtxSw()函数

OSCtxSw()是一个任务级的任务切换函数（在任务中调用，区别于在中断程序中调用的OSIntCtxSw()）。在80X86系统上，通过执行一条软中断的指令来实现任务切换。软中断向量指向OSCtxSw()。在μC/OS-II中，如果任务调用了某个函数，而该函数的执行结果可能造成系统任务重新调度（例如，试图唤醒一个优先级更高的任务），则在函数的末尾会调用OSSched()。如果OSSched()将查找当前就绪的优先级最高的任务，如果不是当前任务，则判断需要进行任务调度，会找到该任务控制块OS_TCB的地址，并将该地址拷贝到变量OSTCBHighRdy中，然后通过宏OS_TASK_SW()执行软中断进行任务切换。在此过程中，变量OSTCBCur始终包含一个指向当前运行任务OS_TCB的指针。OSCtxSw()的示意码如下：

```
void OSCtxSw(void)
{保存处理器寄存器;
将当前任务的堆栈指针保存到当前任务的OS_TCB中:
OSTCBCur->OSTCBStkPtr = Stack pointer;
调用用户定义的OSTaskSwHook();
OSTCBCur  = OSTCBHighRdy;
得到需要恢复的任务的堆栈指针:
Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
将所有处理器寄存器从新任务的堆栈中恢复出来;
执行中断返回指令;
}
```

（3）OSIntCtxSw()函数

在μC/OS-II中，由于中断的产生可能会引起任务切换，在中断服务程序的最后会调用OSIntExit()函数检查任务就绪状态，如果需要进行任务切换，将调用OSIntCtxSw()。所以OSIntCtxSw()又称为中断级的任务切换函数。由于在调用OSIntCtxSw()之前已经发生了中断，OSIntCtxSw()将默认CPU寄存器

已经保存在被中断任务的堆栈中了。OSIntCtxSw()的代码与OSCtxSw()的大部分相同，不同之处是：第一，由于中断已经发生，此处不需要再保存CPU寄存器（没有PUSHA、PUSH ES或PUSH DS）；第二，OSIntCtxSw()需要调整堆栈指针，去掉堆栈中一些不需要的内容，以使堆栈中只包含任务的运行环境。

（4）OSTickISR()函数

前面已经提到过实时系统中时钟节拍发生频率的问题，应该在10~100Hz之间；但由于PC环境的特殊性，时钟节拍由硬件产生，间隔54.93ms(18.20648Hz)。我们将时钟节拍频率设为200Hz。PC时钟节拍的中断向量为0x08，μC/OS-II将此向量截取，指向了μC/OS的中断服务函数OSTickISR()，而原先的中断向量保存在中断129(0x81)中。为满足DOS的需要，原先的中断服务还是每隔54.93ms（实际上还要短些）调用一次。

在μC/OS-II中，当调用OSStart()启动多任务环境后，时钟中断的作用是非常重要的；但在PC环境下，启动μC/OS-II之前就已经有时钟中断发生了。实际上我们希望在μC/OS-II初始化完成之后再发生时钟中断，调用OSTickISR()，所以，在DOS程序的开始必须执行下列初始化过程：

①在main()中，首先调用PC_DOSSaveReturn()函数，功能是取得DOS下时钟中断向量，并将其保存在向量0x81中；

②重新设定中断向量0x80，指向任务切换函数OSCtxSw()；

③至少创立一个任务；

④调用OSStart()，启动多任务环境；

⑤在第一个运行的任务中，设定中断向量0x08，指向函数OSTickISR()，将时钟节拍频率从18.20648Hz改为200Hz。OSTickISR()函数的伪码如下：

```
void OSTickISR(void)
{保存处理器寄存器;
调用OSIntEnter()或者直接将 OSIntNesting加1;
调用OSTimeTick();
调用OSIntExit();
恢复处理器寄存器;
执行中断返回指令;
}
```

和μC/OS-II中的其他中断服务程序一样，OSTickISR()首先在被中断任务堆栈中保存CPU寄存器的值，然后调用OSIntEnter()。μC/OS-II要求在中断服务程序开头调用OSIntEnter()，其作用是将记录中断嵌套层数的全局变量OSIntNesting加1。如果不调用OSIntEnter()，直接将OSIntNesting加1也是允许的。接下来计数器OSTickDOSCtr减1，每发生11次中断，OSTickDOSCtr减到0，则调用DOS的时钟中断处理函数，调用间隔大约是54.93ms。如果不调用DOS时钟中断函数，则必须向中断优先级控制器(PIC)发送命令清除中断标志（必须的！否则系统将崩溃）。如果调用了DOS中断，则此项操作可免，因为在DOS的中断程序中已经完成了。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。在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，而是用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。

4. OS_CPU_C.C文件

μC/OS-II的移植需要用户改写OS_CPU_C.C中的6个函数：

```
OSTaskStkInit()
OSTaskCreateHook()
OSTaskDelHook()
```

OSTaskSwHook()

OSTaskStatHook()

OSTimeTickHook()

实际需要修改的只有OSTaskStkInit()函数，其他5个函数需要声明，但不一定有实际内容。这5个函数都是用户定义的接口函数，所以OS_CPU_C.C中没有给出代码。如果用户需要使用这些函数，请将文件OS_CFG.H中的#define constant OS_CPU_HOOKS_EN设为1；设为0表示不使用这些函数。

函数OSTaskStkInit()由任务创建函数OSTaskCreate()或OSTaskCreateExt()调用，用来初始化任务的堆栈。初始状态的堆栈模拟发生一次中断后的堆栈结构。按照中断后的进栈次序预留各个寄存器的存储空间，而中断返回地址指向任务代码的起始地址，当调用OSTaskCreate()或OSTaskCreateExt()创建一个新任务时，需要传递的参数是：任务代码的起始地址、参数指针(pdata)、任务堆栈顶端的地址、任务的优先级。OSTaskCreateExt()还需要一些其他参数，但与OSTaskStkInit()没有关系。OSTaskStkInit()只需要以上提到的3个参数（task、pdata和ptos）。由于80X86堆栈是16位宽的（以字为单位），OSTaskStkInit()将创立一个指向以字为单位内存区域的指针。同时要求堆栈指针指向空堆栈的顶端。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。

其余的几个函数：OSTaskCreateHook()、OSTaskDelHook()、OSTaskSwHook()、OSTaskStatHook()和OSTimeTickHook()均由用户自定义。ME

参考文献

1 Labrosse Jean J. MicroC/OS-II The Real-Time Kernel.

2 邵贝贝译.μC/OS-II——源码公开的实时嵌入式操作系统.北京：中国电力出版社，2001