uCOS51 移植心得

巨龙公司系统集成开发部 杨屹 asdjf@163.com 2002/10/03

引言

自从发表《uCOS51 移植心得》以来，我收到了很多朋友们的来信，大家对公开源码表示鼓励，谢谢大家的支持！很多人对于编写自己的操作系统很感兴趣，uCOS51 是个不错的选择。它的优点是简单易懂，学习成本低，有利于向 32 位 CPU 过渡。目前，嵌入式 BBS 上的热点是：嵌入式实时多任务操作系统、单片机上网、32bitCPU（如 ARM 等）。其实通过 uCOS51 学习完全可以掌握这些热门技术的精髓，而且学习成本低廉。为此我会陆续将我在研发过程中的经验体会写出来与大家交流，共同进步。

我准备讨论以下内容：uCOS51 高效内核、OS 人机界面 SHELL 的编写、51 机开发板的硬件设计、RTL8019AS 网卡驱动程序、51TCP/IP 协议栈设计、应用协议 FTP、PPP、HTTP、SMTP、SNMP……在 51 上的实现技术、51OS 任务划分和应用程序实例、由 51 软件系统向 ARM 的移植以及其他想到的题目。欢迎大家积极参与。

注：开发板原理图、PCB 图、GAL 烧录文件、芯片手册、全部源程序可以来信索取，在整理好后会共享在网上。

讨论 1----uCOS51 高效内核

前一段时间，我参与了一个 SNMP 网管板的项目，我负责硬件设计和单板软件开发。该板的硬件由 MCS51+RTL8019AS 组成，有 64K FLASH 和 64K SRAM。软件部分有操作系统和 TCPIP 协议栈。硬件比较简单，用了一个月就搞定了，协议栈我参考了老古开发板的部分程序又上网找了 SNMP 源代码也很快完成了，但是测试时发现当使用较低时钟频率的 CPU 时（为了降低成本），由于 ASN.1 编解码部分过于庞大，而我的程序又是一个大循环，AGENT 的响应速度受到严重影响，用户界面也反应迟钝。更坏的消息是公司为了适应市场需求，还要在上面跑 PPP 和 HTTP。那样的话，我就得用 40MHz 的 AT89C51RD2 或者人为的把程序断成几部分然后用状态机的方法在运行时再把它们连接起来。不过，我不想增加成本，也不想把程序搞乱，迫不得已，只好使用操作系统。

说实在的，一开始我也不是很有把握，一来我不清楚 51 的 FLASH 是否装得下这么多代码，二来我只做过 OS 应用开发，对于它的移植想都不敢想。不过，我在 BBS 上搜索了一阵儿后还是有了一些头绪。我找到了几个 OS 的源代码（我喜欢用现成的），按照代码大小、实时性、使用人数、众人口碑等标准，最后选定了 uCOS2。我感觉它的实时性有保障，延时可预测，代码据说可小到 2K，网上讨论这个话题的人也比较多，而且它的网站上有针对 KEIL C51 的移植实例。

经过一番查找，我得到了 5 个版本。其中 3 个是用 KEIL 编译的。本来我想直接把 OS 代码嵌到应用程序中，但后来发现没有一个可以直接使用。有的无法用 KEIL 直接编译，有的需要修改 DLL 在软件仿真下使用。而我需要的是能在串口输入输出，不需要修改任何无关软件，能在软件仿真和硬件上运行的实时多任务操作系统。没有办法，我只好硬着头皮去改编。

我分析了自己的劣势：1。KEIL 刚开始使用，不太熟悉；2。混合编程以前从没有作过；3。时间紧迫，要在 1 个月内搞定。而我的优势就是有 5 个移植实例可供参考，可以上网查

资料。一开始，我用“堆栈”、“混合编程”、“汇编”、“ucos”等关键字在C51BBS和老古论坛上检索相关信息并逐条阅读，读过之后，头脑中的思路逐渐清晰了。我了解到在KEIL的HLP目录下有A51.PDF和C51.PDF非常全面的介绍了汇编和C51，是KEIL的权威用户手册；SP初始化、内存清0等操作在STARTUP.A51文件中实现，用户可以改写它；KEIL的变量，子程序等的分配信息可以在.M51文件里查到；KEIL自己的论坛里有很多疑难问题的解答……通过阅读并经过思考，解决了堆栈起点、堆栈空间大小的设定等关键问题。论坛里的问题有些是我没有想到的，这使我发现了自己的疏漏。

在网上获得大量信息后，我开始阅读《uCOSII》中文版，一共读了3遍。第一遍是浏览，了解到uCOSII包括任务调度、时间管理、内存管理、资源管理（信号量、邮箱、消息队列）四大部分，没有文件系统、网络接口、输入输出界面。它的移植只与4个文件相关：汇编文件（OS_CPU_A.ASM）、处理器相关C文件（OS_CPU.H、OS_CPU_C.C）和配置文件（OS_CFG.H）。有64个优先级，系统占用8个，用户可创建56个任务，不支持时间片轮转。第二遍主要是把整个工作过程在头脑里过了一下，不懂的地方有针对性地查书，重点是思考工作原理和流程。我发现其实它的思路挺简单的。就是 “近似地每时每刻总是让优先级最高的就绪任务处于运行状态” 。为了保证这一点，它在调用系统API函数、中断结束、定时中断结束时总是执行调度算法。原作者通过事先计算好数据，简化了运算量，通过精心设计就绪表结构，使得延时可预知。任务的切换是通过模拟一次中断实现的。第三遍重点看了移植部分的内容。对照实例，研究了代码的具体实现方法。

前期准备用了20几天，真正编写代码只用了1.5天，调试用了2天。具体过程如下：

(1)拷贝书后附赠光盘 sourcecode 目录下的内容到 C:\YY 下，删除不必要的文件和EX1L.C，只剩下p187(《uCOSII》)上列出的文件。

(2)改写最简单的OS_CPU.H

数据类型的设定见C51.PDF第176页。注意BOOLEAN要定义成unsigned char类型，因为bit类型为C51特有，不能用在结构体里。

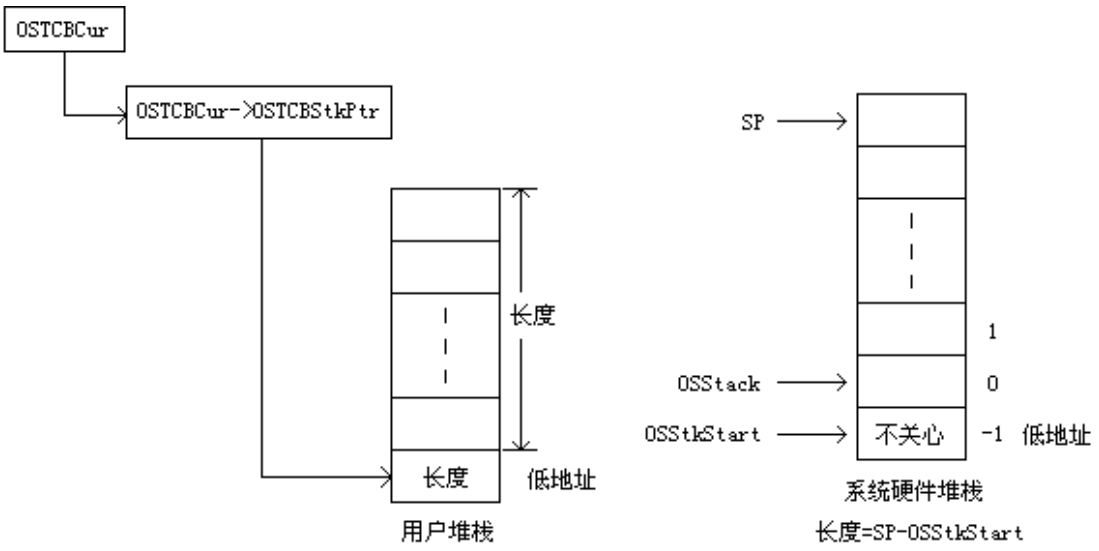
EA=0关中断；EA=1开中断。这样定义即减少了程序行数，又避免了退出临界区后关中断造成的死机。

MCS-51堆栈从下往上增长(1=向下，0=向上)，OS_STK_GROWTH定义为0

#define OS_TASK_SW() OSCtxSw() 因为MCS-51没有软中断指令，所以用程序调用代替。两者的堆栈格式相同，RETI 指令复位中断系统，RET 则没有。实践表明，对于MCS-51，用子程序调用入栈，用中断返回指令RETI出栈是没有问题的，反之中断入栈RET出栈则不行。总之，对于入栈，子程序调用与中断调用效果是一样的，可以混用。在没有中断发生的情况下复位中断系统也不会影响系统正常运行。详见《uC/OS-II》第八章193页第12行

(3)改写OS_CPU_C.C

我设计的堆栈结构如下图所示：



堆栈结构图

TCB 结构体中 OSTCBStkPtr 总是指向用户堆栈最低地址，该地址空间内存放用户堆栈长度，其上空间存放系统堆栈映像，即：用户堆栈空间大小=系统堆栈空间大小+1。

SP 总是先加 1 再存数据，因此，SP 初始时指向系统堆栈起始地址(OSStack)减 1 处(OSStkStart)。很明显系统堆栈存储空间大小=SP-OSStkStart。

任务切换时，先保存当前任务堆栈内容。方法是：用 SP-OSStkStart 得出保存字节数，将其写入用户堆栈最低地址内，以用户堆栈最低地址为起址，以 OSStkStart 为系统堆栈起址，由系统栈向用户栈拷贝数据，循环 SP-OSStkStart 次，每次拷贝前先将各自栈指针增 1。

其次，恢复最高优先级任务系统堆栈。方法是：获得最高优先级任务用户堆栈最低地址，从中取出“长度”，以最高优先级任务用户堆栈最低地址为起址，以 OSStkStart 为系统堆栈起址，由用户栈向系统栈拷贝数据，循环“长度”数值指示的次数，每次拷贝前先将各自栈指针增 1。

用户堆栈初始化时从下向上依次保存：用户堆栈长度（15），PCL，PCH，PSW，ACC，B，DPL，DPH，R0，R1，R2，R3，R4，R5，R6，R7。不保存 SP，任务切换时根据用户堆栈长度计算得出。

OSTaskStkInit 函数总是返回用户栈最低地址。

操作系统 tick 时钟我使用了 51 单片机的 T0 定时器，它的初始化代码用 C 写在了本文件中。

最后还有几点必须注意的事项。本来原则上我们不用修改与处理器无关的代码，但是由于 KEIL 编译器的特殊性，这些代码仍要多处改动。因为 KEIL 缺省情况下编译的代码不可重入，而多任务系统要求并发操作导致重入，所以要在每个 C 函数及其声明后标注 reentrant 关键字。另外，“pdata”、“data”在 uCOS 中用做一些函数的形参，但它同时又是 KEIL 的关键字，会导致编译错误，我通过把“pdata”改成“ppdata”，“data”改成“ddata”解决了此问题。OSTCBCur、OSTCBHighRdy、OSRunning、OSPrioCur、OSPrioHighRdy 这几个变量在汇编程序中用到了，为了使用 Ri 访问而不用 DPTR，应该用 KEIL 扩展关键字 IDATA 将它们定义在内部 RAM 中。

(4)重写 OS_CPU_A.ASM

A51 宏汇编的大致结构如下：

NAME 模块名 ;与文件名无关

;定义重定位段　必须按照 C51 格式定义，汇编遵守 C51 规范。段名格式为：?PR?函数名?模块名

;声明引用全局变量和外部子程序　注意关键字为“EXTRN”没有‘E’

全局变量名直接引用

无参数/无寄存器参数函数 FUNC

带寄存器参数函数 _FUNC

重入函数 _?FUNC

;分配堆栈空间

只关心大小，堆栈起点由 keil 决定，通过标号可以获得 keil 分配的 SP 起点。切莫自己分配堆栈起点，只要用 DS 通知 KEIL 预留堆栈空间即可。

?STACK 段名与 STARTUP.A51 中的段名相同，这意味着 KEIL 在 LINK 时将把两个同名段拼在一起，我预留了 40H 个字节，STARTUP.A51 预留了 1 个字节，LINK 完成后堆栈段总长为 41H。查看 yy.m51 知 KEIL 将堆栈起点定在 21H，长度 41H，处于内部 RAM 中。

;定义宏

宏名 MACRO　实体　ENDM

;子程序

OSStartHighRdy

OSCtxSw

OSIntCtxSw

OSTickISR

SerialISR

END ;声明汇编源文件结束

一般指针占 3 字节。+0 类型+1 高 8 位数据+2 低 8 位数据 详见 C51.PDF 第 178 页

低位地址存高 8 位值，高位地址存低 8 位值。例如 0x1234，基址+0:0x12 基址+1:0x34

(5)移植串口驱动程序

在此之前我写过基于中断的串口驱动程序，包括打印字节/字/长字/字符串，读串口，初始化串口/缓冲区。把它改成重入函数即可直接使用。

系统提供的显示函数是并发的，它不是直接显示到串口，而是先输出到显存，用户不必担心 IO 慢速操作影响程序运行。串口输入也采用了同样的技术，他使得用户在 CPU 忙于处理其他任务时照样可以盲打输入命令。

(6)编写测试程序 Demo(YY.C)

Demo 程序创建了 3 个任务 A、B、C 优先级分别为 2、3、4，A 每秒显示一次，B 每 3 秒显示一次，C 每 6 秒显示一次。从显示结果看，显示 3 个 A 后显示 1 个 B，显示 6 个 A 和 2 个 B 后显示 1 个 C，结果显然正确。

显示结果如下：

AAAAAA111111 is active

AAAAAA111111 is active

AAAAAA111111 is active

BBBBBB333333 is active

AAAAAA111111 is active

AAAAAA111111 is active

AAAAAA111111 is active
BBBBBB333333 is active
CCCCCC666666 is active
AAAAAA111111 is active
AAAAAA111111 is active
AAAAAA111111 is active
BBBBBB333333 is active
AAAAAA111111 is active
AAAAAA111111 is active
AAAAAA111111 is active
BBBBBB333333 is active
CCCCCC666666 is active
Demo 程序经 Keil701 编译后，代码量为 7-8K，可直接在 KeilC51 上仿真运行。
编译时要将 OS_CPU_C.C、UCOS_II.C、OS_CPU_A.ASM、YY.C 加入项目

以上是我这次移植 uCOS51 的一些心得，写出来只是让准备在 51 上运行操作系统的同行们少走弯路并增强使用信心。我强烈推荐大家在自己的 51 系统中使用 uCOS 这个简单实用的自己的操作系统。它的大小应该不是问题，性能上的提高却是显著的。但愿此文能对朋友们有所帮助，错误在所难免，希望各位大虾指正，诸位高手们见笑了！

注：全部源码可来信索要(asdjf@163.com)，以下仅为关键代码部分。

文件名 : OS_CPU_A.ASM

```
$NOMOD51
EA   BIT  0A8H.7
SP         DATA       081H
B          DATA      0F0H
ACC        DATA      0E0H
DPH        DATA      083H
DPL        DATA      082H
PSW        DATA      0D0H
TR0        BIT         088H.4
TH0        DATA      08CH
TL0 DATA     08AH

            NAME OS_CPU_A       ;模块名

;定义重定位段
?PR?OSStartHighRdy?OS_CPU_A         SEGMENT CODE
?PR?OSCtxSw?OS_CPU_A                 SEGMENT CODE
?PR?OSIntCtxSw?OS_CPU_A              SEGMENT CODE
?PR?OSTickISR?OS_CPU_A               SEGMENT CODE
```

```
?PR?_?serial?OS_CPU_A                    SEGMENT CODE

;声明引用全局变量和外部子程序
        EXTRN IDATA (OSTCBCur)
        EXTRN IDATA (OSTCBHighRdy)
        EXTRN IDATA (OSRunning)
        EXTRN IDATA (OSPrioCur)
        EXTRN IDATA (OSPrioHighRdy)

        EXTRN CODE   (_?OSTaskSwHook)
        EXTRN CODE   (_?serial)
        EXTRN CODE   (_?OSIntEnter)
        EXTRN CODE   (_?OSIntExit)
        EXTRN CODE   (_?OSTimeTick)

;对外声明 4 个不可重入函数
        PUBLIC OSStartHighRdy
        PUBLIC OSCtxSw
        PUBLIC OSIntCtxSw
        PUBLIC OSTickISR

        ;PUBLIC SerialISR

;分配堆栈空间。只关心大小，堆栈起点由 keil 决定，通过标号可以获得 keil 分配的 SP 起点。
?STACK SEGMENT IDATA
        RSEG ?STACK
OSStack:
        DS 40H
OSStkStart IDATA OSStack-1

;定义压栈出栈宏
PUSHALL      MACRO
        PUSH PSW
        PUSH ACC
        PUSH B
        PUSH DPL
        PUSH DPH
        MOV   A,R0     ;R0-R7 入栈
        PUSH ACC
        MOV   A,R1
        PUSH ACC
        MOV   A,R2
        PUSH ACC
        MOV   A,R3
```

```
        PUSH ACC
        MOV   A,R4
        PUSH ACC
        MOV   A,R5
        PUSH ACC
        MOV   A,R6
        PUSH ACC
        MOV   A,R7
        PUSH ACC
        ;PUSH SP      ;不必保存 SP，任务切换时由相应程序调整
        ENDM

POPALL      MACRO
        ;POP   ACC    ;不必保存 SP，任务切换时由相应程序调整
        POP   ACC     ;R0-R7 出栈
        MOV   R7,A
        POP   ACC
        MOV   R6,A
        POP   ACC
        MOV   R5,A
        POP   ACC
        MOV   R4,A
        POP   ACC
        MOV   R3,A
        POP   ACC
        MOV   R2,A
        POP   ACC
        MOV   R1,A
        POP   ACC
        MOV   R0,A
        POP   DPH
        POP   DPL
        POP   B
        POP   ACC
        POP   PSW
        ENDM

;子程序
;--------------------------------------------------------------------
        RSEG ?PR?OSStartHighRdy?OS_CPU_A
OSStartHighRdy:
        USING 0      ;上电后 51 自动关中断，此处不必用 CLR EA 指令，因为到此处还未
开中断，本程序退出后，开中断。
        LCALL _?OSTaskSwHook
```

```
OSCtxSw_in:

        ;OSTCBCur ===> DPTR   获得当前 TCB 指针，详见 C51.PDF 第 178 页
        MOV   R0,#LOW  (OSTCBCur) ;获得 OSTCBCur 指针低地址，指针占 3 字节。+0
类型+1 高 8 位数据+2 低 8 位数据
        INC   R0
        MOV   DPH,@R0       ;全局变量 OSTCBCur 在 IDATA 中
        INC   R0
        MOV   DPL,@R0

        ;OSTCBCur->OSTCBStkPtr ===> DPTR   获得用户堆栈指针
        INC   DPTR            ;指针占 3 字节。+0 类型+1 高 8 位数据+2 低 8 位数据
        MOVX A,@DPTR          ;.OSTCBStkPtr 是 void 指针
        MOV   R0,A
        INC   DPTR
        MOVX A,@DPTR
        MOV   R1,A
        MOV   DPH,R0
        MOV   DPL,R1

        ;*UserStkPtr  ===>  R5   用户堆栈起始地址内容(即用户堆栈长度放在此处)   详见
文档说明   指针用法详见 C51.PDF 第 178 页
        MOVX A,@DPTR          ;用户堆栈中是 unsigned char 类型数据
        MOV   R5,A            ;R5=用户堆栈长度

        ;恢复现场堆栈内容
        MOV   R0,#OSStkStart

restore_stack:

        INC   DPTR
        INC   R0
        MOVX A,@DPTR
        MOV   @R0,A
        DJNZ R5,restore_stack

        ;恢复堆栈指针 SP
        MOV   SP,R0

        ;OSRunning=TRUE
        MOV   R0,#LOW (OSRunning)
        MOV   @R0,#01
```

```
        POPALL
        SETB EA       ;开中断
        RETI
;--------------------------------------------------------------------
        RSEG ?PR?OSCtxSw?OS_CPU_A
OSCtxSw:
        PUSHALL

OSIntCtxSw_in:

        ;获得堆栈长度和起址
        MOV   A,SP
        CLR   C
        SUBB A,#OSStkStart
        MOV   R5,A        ;获得堆栈长度

        ;OSTCBCur ===> DPTR   获得当前 TCB 指针，详见 C51.PDF 第 178 页
        MOV   R0,#LOW  (OSTCBCur) ;获得 OSTCBCur 指针低地址，指针占 3 字节。+0
类型+1 高 8 位数据+2 低 8 位数据
        INC   R0
        MOV   DPH,@R0      ;全局变量 OSTCBCur 在 IDATA 中
        INC   R0
        MOV   DPL,@R0

        ;OSTCBCur->OSTCBStkPtr ===> DPTR   获得用户堆栈指针
        INC   DPTR            ;指针占 3 字节。+0 类型+1 高 8 位数据+2 低 8 位数据
        MOVX A,@DPTR        ;.OSTCBStkPtr 是 void 指针
        MOV   R0,A
        INC   DPTR
        MOVX A,@DPTR
        MOV   R1,A
        MOV   DPH,R0
        MOV   DPL,R1

        ;保存堆栈长度
        MOV   A,R5
        MOVX @DPTR,A

        MOV   R0,#OSStkStart   ;获得堆栈起址
save_stack:

        INC   DPTR
        INC   R0
        MOV   A,@R0
```

```
        MOVX @DPTR,A
        DJNZ R5,save_stack

        ;调用用户程序
        LCALL _?OSTaskSwHook

        ;OSTCBCur = OSTCBHighRdy
        MOV   R0,#OSTCBCur
        MOV   R1,#OSTCBHighRdy
        MOV   A,@R1
        MOV   @R0,A
        INC   R0
        INC   R1
        MOV   A,@R1
        MOV   @R0,A
        INC   R0
        INC   R1
        MOV   A,@R1
        MOV   @R0,A

        ;OSPrioCur = OSPrioHighRdy   使用这两个变量主要目的是为了使指针比较变为字节比较，以便节省时间。
        MOV   R0,#OSPrioCur
        MOV   R1,#OSPrioHighRdy
        MOV   A,@R1
        MOV   @R0,A

        LJMP OSCtxSw_in
;--------------------------------------------------------------------
        RSEG ?PR?OSIntCtxSw?OS_CPU_A

OSIntCtxSw:

        ;调整 SP 指针去掉在调用 OSIntExit(),OSIntCtxSw()过程中压入堆栈的多余内容
        ;SP=SP-4

        MOV   A,SP
        CLR   C
        SUBB A,#4
        MOV   SP,A

        LJMP OSIntCtxSw_in
;--------------------------------------------------------------------
        CSEG AT 000BH       ;OSTickISR
```

```
        LJMP OSTickISR     ;使用定时器 0
        RSEG ?PR?OSTickISR?OS_CPU_A

OSTickISR:

        USING 0
        PUSHALL

        CLR    TR0
        MOV    TH0,#70H       ;定义 Tick=50 次/秒(即 0.02 秒/次)
        MOV    TL0,#00H       ;OS_CPU_C.C  和  OS_TICKS_PER_SEC
        SETB TR0

        LCALL _?OSIntEnter
        LCALL _?OSTimeTick
        LCALL _?OSIntExit
        POPALL
        RETI
;--------------------------------------------------------------------
        CSEG AT 0023H      ;串口中断
        LJMP SerialISR     ;工作于系统态，无任务切换。
        RSEG ?PR?_?serial?OS_CPU_A

SerialISR:

        USING 0
        PUSHALL
        CLR    EA
        LCALL _?serial
        SETB EA
        POPALL
        RETI
;--------------------------------------------------------------------
        END
;--------------------------------------------------------------------
```

文件名 : OS_CPU_C.C

```
void *OSTaskStkInit (void (*task)(void *pd), void *ppdata, void *ptos, INT16U opt) reentrant
{
    OS_STK *stk;

    ppdata = ppdata;
    opt    = opt;                                    //opt 没被用到，保留此语句防止告警产生
```

```
    stk       = (OS_STK *)ptos;                    //用户堆栈最低有效地址
    *stk++ = 15;                                   //用户堆栈长度
    *stk++ = (INT16U)task & 0xFF;                  //任务地址低 8 位
    *stk++ = (INT16U)task >> 8;                    //任务地址高 8 位
    *stk++ = 0x00;                                 //PSW
    *stk++ = 0x0A;                                 //ACC
    *stk++ = 0x0B;                                 //B
    *stk++ = 0x00;                                 //DPL
    *stk++ = 0x00;                                 //DPH
    *stk++ = 0x00;                                 //R0
    *stk++ = 0x01;                                 //R1
    *stk++ = 0x02;                                 //R2
    *stk++ = 0x03;                                 //R3
    *stk++ = 0x04;                                 //R4
    *stk++ = 0x05;                                 //R5
    *stk++ = 0x06;                                 //R6
    *stk++ = 0x07;                                 //R7
                                                   //不用保存 SP，任务切换时根据用户堆栈
长度计算得出。
    return ((void *)ptos);
}

#if OS_CPU_HOOKS_EN
void OSTaskCreateHook (OS_TCB *ptcb) reentrant
{
    ptcb = ptcb;                                   /* Prevent compiler warning
*/
}

void OSTaskDelHook (OS_TCB *ptcb) reentrant
{
    ptcb = ptcb;                                   /* Prevent compiler warning
*/
}

void OSTimeTickHook (void) reentrant
{
}
#endif

//初始化定时器 0
void InitTimer0(void) reentrant
{
    TMOD=TMOD&0xF0;
```

```
    TMOD=TMOD|0x01;      //模式 1(16 位定时器)，仅受 TR0 控制
    TH0=0x70;      //定义 Tick=50 次/秒(即 0.02 秒/次)
    TL0=0x00;      //OS_CPU_A.ASM  和  OS_TICKS_PER_SEC
    ET0=1;         //允许 T0 中断
    TR0=1;
}
```

文件名 : YY.C

```
#include <includes.h>

#define MAX_STK_SIZE 64

void TaskStartyya(void *yydata) reentrant;
void TaskStartyyb(void *yydata) reentrant;
void TaskStartyyc(void *yydata) reentrant;

OS_STK TaskStartStkyya[MAX_STK_SIZE+1];//注意：我在 ASM 文件中设置?STACK 空间为
40H 即 64，不要超出范围。
OS_STK TaskStartStkyyb[MAX_STK_SIZE+1];//用户栈多一个字节存长度
OS_STK TaskStartStkyyc[MAX_STK_SIZE+1];

void main(void)
{
    OSInit();

    InitTimer0();
    InitSerial();
    InitSerialBuffer();

    OSTaskCreate(TaskStartyya, (void *)0, &TaskStartStkyya[0],2);
    OSTaskCreate(TaskStartyyb, (void *)0, &TaskStartStkyyb[0],3);
    OSTaskCreate(TaskStartyyc, (void *)0, &TaskStartStkyyc[0],4);

    OSStart();
}


void TaskStartyya(void *yydata) reentrant
{
    yydata=yydata;
    clrscr();
    PrintStr("\n\t\t*******************************\n");
    PrintStr("\t\t*         Hello! The world.           *\n");
```

```
    PrintStr("\t\t*******************************\n\n\n");

    for(;;){
        PrintStr("\tAAAAAA111111 is active.\n");
        OSTimeDly(OS_TICKS_PER_SEC);
    }
}

void TaskStartyyb(void *yydata) reentrant
{
    yydata=yydata;

    for(;;){
        PrintStr("\tBBBBBB333333 is active.\n");
        OSTimeDly(3*OS_TICKS_PER_SEC);
    }
}

void TaskStartyyc(void *yydata) reentrant
{
    yydata=yydata;

    for(;;){
        PrintStr("\tCCCCCC666666 is active.\n");
        OSTimeDly(6*OS_TICKS_PER_SEC);
    }
}
```