|  |
| --- |
| **uCOS-II在ARM7上移植** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **uCOS-II在ARM7上移植**  **1、文件OS\_CPU.H的编写**  **1）定义可移植性强的数据类型**  μC/OS-Ⅱ不使用C语言中的short、int、long等与处理器类型有关的数据类型，而代之以移植性强的数据类型，这样既直观又便于移植，不过这就成了必须移植的代码。根据ADS编译器的特性，在文件OS\_CPU.H中这些数据类型的定义如下：   |  | | --- | | typedef     unsigned char    BOOLEAN;                //布尔类型  typedef     unsigned char    INT8U;                  //无符号8位整型  typedef     signed  char     INT8S;                  //有符号8位整型  typedef     unsigned short    INT16U;                //无符号16位整型  typedef     signed  short     INT16S;                //有符号16位整型  typedef     unsigned int      INT32U;                //无符号32位整型  typedef     signed  int       INT32S;                //有符号32位整型  typedef     float          FP32;                     //单精度浮点类型（32位长度）  typedef     double         FP64;                     //双精度浮点类型（64位长度）  typedef     INT32U        OS\_STK;                    //堆栈是32位宽度 |     **2）堆栈增长类型和其他定义**  虽然ARM处理器对堆栈向上与向下的两种增长方式都予以支持，但由于编译器ADS仅支持堆栈从上往下，并且必须是满递减堆栈，所以在文件中用来定义堆栈增长方式的常量OS\_STK\_GROWTH 的值为1，与其他相关定义如下：   |  | | --- | | #define      OS\_STK\_GROWTH 1                         //堆栈由高地址向低地址增长  #define      USR32Mode         0x10                   //用户模式  #define       SYS32Mode          0x1f                    //系统模式  #define       IRQ32Mode          0x12                   //IRQ模式 |   ARM7处理器核具有用户、系统、管理、中止、未定义、IRQ中断和FIQ中断7种模式，其中后5种模式与异常相关。μC/OS-Ⅱ任务使用的模式一般为用户模式，当发生异常时（如定时器产生了IRQ中断），处理器会进入其他模式执行异常处理代码，所以必须事先保护好用户模式下的现场。在异常模式下保护好用户模式下的现场，需要切换到系统模式才行，所以我们在头文件中定义了这几种模式。    **3）外部函数声明**  在μC/OS-II.h头文件中，有些要移植的函数已经声明，包括：OSTaskStkInit()、OSIntCtxSw(void)、OSStartHighRdy(void)、××HOOK××()。其他未声明的函数需要我们在文件OS\_CPU.H中声明，声明如下：   |  | | --- | | void            OS\_TASK\_SW(void);                              //任务级任务切换函数  void            OS\_ENTER\_CRITICAL(void);              //关中断  void            OS\_EXIT\_CRITICAL(void);                  //开中断  void            OSTickISR(void)                                     //时钟节拍函数 |     **2、文件OS\_CPU\_C.C的编写**  **1）任务堆栈初始化函数OSTaskStkInit()**  在编写任务堆栈初始化函数OSTaskStkInit()之前，必须先根据处理器的结构和特点确定任务的堆栈结构。ARM7处理器在ARM状态中，16个通用寄存器(R0~R12、SP、LR、PC) 和1个状态寄存器CPSR可在任何时候被访问。我们在任务堆栈中需要保存除SP堆栈指针之外其他16个寄存器的内容，移植的堆栈结构如图1所示。  图1  任务堆栈结构    **2) 钩子函数**  μC/OS-II为了用户在系统函数中书写自己的代码而预置了一些函数名带有Hook字样的钩子函数（如OSTimeTickHook），这些函数在移植时可全为空函数。  **3、文件OS\_CPU\_A.S的编写**  **1）开/关中断函数**  μC/OS-II分别使用宏OS\_ENTER\_CRITICAL()和OS\_EXIT\_CRITICAL来关中断和开中断。在ARM7处理器核中可利用改变程序状态寄存器CPSR中的相应控制位实现。当I位（第7位）置位时，IRQ中断被禁止；当I位清零时，IRQ中断被允许。当F位（第6位）置位时，FIQ中断被禁止；当F位清零时，FIQ中断被允许。  关中断OS\_ENTER\_CRITICAL()的实现代码见程序清单1。  **程序清单1  OS\_ENTER\_CRITICAL()**   |  | | --- | | OS\_ENTER\_CRITICAL  MRS R0,CPSR                                         ;R0 = CPSR  ORR R0,R0,#0xC0                                  ;R0的第6、7位置1  MSR CPSR\_c,R0                                     ;CPSR = R0  MOV         PC,LR                                    ;程序返 |   开中断OS\_EXIT\_CRITICAL()的实现代码见程序清单2。  **程序清单2  OS\_EXIT\_CRITICAL()**   |  | | --- | | OS\_EXIT\_CRITICAL  MRS R0,CPSR                                         ;R0 = CPSR  BIC   R0,R0,#0xC0                                  ;R0的第6、7位清0  MSR CPSR\_c,R0                                     ;CPSR = R0  MOV         PC,LR                                    ;程序返回 |     **2）OSStartHighRdy(void)**  OSStart()函数调用OSStartHighRdy()来使就绪态任务中优先级最高的任务开始执行，这个函数的示意性代码如下：  void  OSStartHighRdy(void)  {  调用用户定义的OSTaskSwHook()；  OSRunning = TRUE；  得到将要恢复运行任务的堆栈指针给SP：  SP = OSTCBHighRdy —〉OSTCBStkPtr；           从新任务堆栈中恢复处理器的所有寄存器；  }    **3）OS\_TASK\_SW(void)**  μC/OS-II通过调用OSSched()函数来完成任务调度的，OSSched()先将最高优先级任务的地址装载到OSTCBHighRdy，再通过调用OS\_TASK\_SW()执行任务级的任务切换，OS\_TASK\_SW的示意性代码如下：  void  OS\_TASK\_SW(void)  {  保存处理器寄存器；  在当前任务的任务控制块中保存当前任务的堆栈指针：  OSTCBCur —〉OSTCBStkPtr = SP；  调用用户定义的OSTaskSwHook()；  OSTCBCur = OSTCBHighRdy；  OSPrioCur = OSPrioHighRdy;  得到将要恢复运行任务的堆栈指针给SP：  SP = OSTCBHighRdy —〉OSTCBStkPtr；           从新任务堆栈中恢复处理器的所有寄存器；  }  通过对示意性代码的分析，我们发现OS\_TASK\_SW()函数比OSIntCtxSw()函数多做一份工作，即程序开头需要保存处理器的现场；  **4）OSIntCtxSw(void)**  OSIntExit()通过调用OSIntCtxSw()，在中断服务程序中执行任务切换功能。OSIntCtxSw()是在中断服务程序中被调用的，此时处理器工作在IRQ工作模式下，需要保存用户模式下的处理器寄存器（用户模式下与系统模式下的处理器共享同一套处理器）。OSIntCtxSw()的示意性代码如下：  void  OSIntCtxSw(void)  {  切换到系统模式，保存该模式下的处理器寄存器；  OSPrioCur = OSPrioHighRdy；  在当前任务的任务控制块中保存当前任务的堆栈指针：  OSTCBCur —〉OSTCBStkPtr = SP；  调用用户定义的OSTaskSwHook()；  OSTCBCur = OSTCBHighRdy；  OSPrioCur = OSPrioHighRdy；  得到将要恢复运行任务的堆栈指针给SP：  SP = OSTCBHighRdy —〉OSTCBStkPtr；           从新任务堆栈中恢复处理器的所有寄存器；  }    **5）时钟节拍函数**  在本移植中，只使用了ARM的IRQ中断。由于不同的ARM芯片的中断系统并不完全一样，因此不可能编写出对所有使用ARM核的处理器通用的中断及节拍移植代码。尽管不能编写通用的中断代码，但在进入中断后，有些工作是必须都要完成的，其中断的示意性代码如下：  void      OSTickISR（void）  {  将IRQ模式下的全部寄存器保存到当前模式下的堆栈中；  OSIntEnter();           调用定时器0的中断代码；  OSIntExit()；  从当前模式下的堆栈中恢复全部寄存器；  }  系统进入时钟节拍后，这时ARM芯片工作在IRQ模式下，定时器0的中断代码中调用了OSTimeTick()函数，并通知ARM中断结束。OSTickISR的实现代码见程序清单3。  **程序清单3  OSTickISR()**   |  | | --- | | OSTickISR       STMFD         SP!, {R0-R3,R12,LR}         BL            OSIntEnter       BL            Timer0\_Exception       ; 调用定时器0的中断处理程序，详见下节       BL            OSIntExit         LDMFD         SP!,{R0-R3,R12,LR}         SUBS          PC,LR,#4 |     **4、移植到LPC2220处理器上**  目前，基于ARM7处理器核的各种处理器芯片有很多，它们在存储系统、片内外设设备、中断系统等方面都存在或多或少的差异。因此移植时，在上一节介绍的μC/OS-II在ARM7上移植的通用代码的基础上，还需要一小部分与具体处理器类型相关的代码。这小部分代码主要是解决时钟节拍中断，我们下面以LPC2220为例进行介绍。  LPC2220有两个定时器：定时器0和定时器1。在这里需要用到一个定时器来产生定时中断，作为μC/OS-II的时钟节拍，假设用到定时器0，则要对其进行初始化，代码如下：   |  | | --- | | void Timer0Init(void)  {      T0IR = 0xffffffff;      T0TC = 0;      T0TCR = 0x01;      T0MCR = 0x03;      T0MR0 = (Fpclk / OS\_TICKS\_PER\_SEC);        // 设定uC/OS-II的时钟节拍       } |   定时器0产生定时中断后，需要为μC/OS-II提供时钟节拍服务，所以定时器0的中断服务程序中必须调用OSTimeTick()函数，中断服务程序具体代码如下：   |  | | --- | | void Timer0\_Exception(void)  {   T0IR = 0x01;      VICVectAddr = 0;            // 通知中断控制器中断结束      OSTimeTick();  } |   与裸机下的中断服务程序不同的是，在μC/OS-II中执行中断服务前要保护现场，中断退出前要恢复CPU的寄存器内容，所以运行在μC/OS-II操作系统上的中断服务程序特别要注意以上两点。OSTickISR中断服务程序就是一个例子。  最后我们初始化VIC中断控制器，把时钟节拍中断服务程序挂接到LPC2220的中断系统中去，具体代码如下：   |  | | --- | | void VICInit(void)  {      extern void OSTickISR(void);      VICIntEnClr = 0xffffffff;      VICVectAddr0 = (uint32)OSTickISR;      VICVectCntl0 = (0x20 | 0x04);      VICIntEnable = 1 << 4;   } | |