

μC/OS-II 在 MSP430 上的移植

(目标系统选用 MSP-TEST449 学习板)

产品名称: μC/OS-II 在 MSP430 上的移植代码

编 辑: 周震宇

审 核: 季燕飞

审 批: 梁源

日 期: 2004.05.25

杭州利尔达单片机技术有限公司

目 录

1 开发工具.....	3
2 目录和文件.....	3
3 INCLUDE.H 文件	4
4 OS_CPU.H 文件	4
4.1 数据类型.....	7
4.2 代码临界段.....	7
4.3 堆栈增长方向.....	8
4.4 OS_TASK_SW().....	8
5 OS_CPU_A.ASM.....	8
5.1 OSStartHighRdy().....	8
5.2 OSCtxSw()	9
5.3 OSIntCtxSw()	10
5.4 OSTickISR()	11
6 OS_CPU_C.C.....	12
6.1 OSTaskStkInit()	13
6.2 OSTaskCreateHook()	14
6.3 OSTaskDelHook()	15
6.4 OSTaskSwHook()	15
6.5 OSTaskStatHook()	15
6.6 OSTimeTickHook()	15
7 移植代码正确性验证.....	15
7.1 移植代码在 AQ430 环境下的验证过程	16

本文将介绍如何将实时多任务操作系统 $\mu C/OS-II$ 移植到 TI 的 MSP430 系列 CPU 上。本文内容大致可分为两部分: ①介绍如何编写 $\mu C/OS-II$ 针对 MSP430 的 PORT; ②在 AQ430 的集成编译环境中对移植代码的正确性作简单的验证。

图 0-1 为 MSP430 的存储器结构。

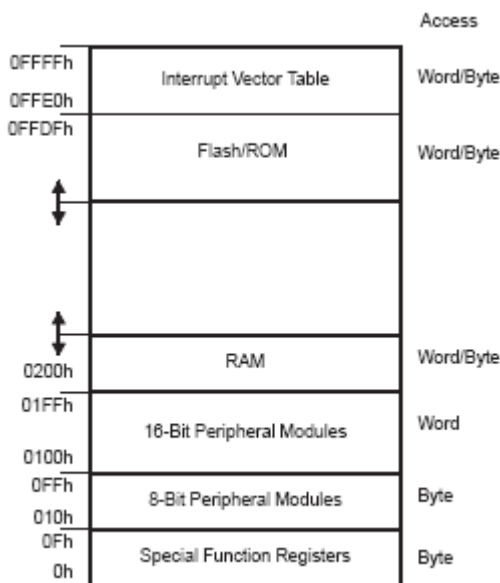


图 0-1 MSP430 的存储器结构

因为 $\mu C/OS-II$ 需要较大的 RAM 空间，我们可以选用 MSP430F149、MSP430F449 等具有 2K RAM 空间的 CPU 来完成我们的移植。

1 开发工具

我们选用 MSP-TEST44X 为目标系统，它是一块基于 MSP430F449 的学习板。软件开发环境采用 AQ430，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。本章所介绍的移植和代码都是针对 AQ430 的，对于其他的 C430 编译器，本章所介绍的移植和代码仅供参考。

2 目录和文件

为了方便实现复制、共享，我们将所有的文件放在名为“PORT_AQ430_449”目录中。具体包括 $\mu C/OS-II$ 的内核源代码、针对 MSP430 CPU 和 AQ430 编译器的移植代码(os_cpu_c.c os_cpu_a.asm os_cpu.h)、包含头文件(include.h)、已通过测试调试的 AQ430 项目文件(uCOS_AQ430_PORT.qpj)。这样做的目的是为了更方便我们和大家一起讨论学习 $\mu C/OS-II$ ，不管从何处下载或复制得到“PORT_AQ430_449”及其所包含的文件，只要打开 AQ430 的项目文件(uCOS_AQ430_PORT.qpj)，不需要任何其他的操作或修改，就立刻可以进行编译调试等实际操作，省去由于头文件路径不对而导致编译通不过的麻烦。当然你也完全可以重新创建项目进行编译调试。重新创建 AQ430 项目的过程请参考后文。

注：若文件为只读，请将只读属性去除，否则 AQ430 项目编译可能报错。

3 INCLUDE.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。使用INCLUDES.H的好处是所有的.C文件都只包含一个头文件，程序简洁，可读性强。缺点是.C文件可能会包含一些它并不需要的头文件，额外的增加编译时间。与优点相比，多一些编译时间还是可以接受的。用户可以改写INCLUDES.H文件，增加自己的头文件，但必须加在文件末尾。程序清单3-1是为MSP430编写的INCLUDES.H文件的内容。

程序清单3-1 INCLUDES.H

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <in430.h>
#include <msp430x44x.h>

#include "OS_CPU.H"
#include "os_cfg.h"
#include "uCOS_II.H"
```

注：此文件中的最后三条包含语句必须用“双引号”，这样AQ430才能在项目文件所在的目录中查找。对于其他文件中包含项目文件所在目录的文件的包含语句都用“双引号”。

4 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。程序清单 4-1 是为MSP430 编写的 OS_CPU.H 文件的内容。

程序清单4-1 OS_CPU.H

```
#ifndef OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT extern
#endif

/*****
*
* 数据类型
*****/
```

```
*
                                (与编译器相关的内容)
*****/

typedef unsigned char  BOOLEAN;

typedef unsigned char  INT8U;                /* Unsigned   8 bit quantity */
typedef signed   char  INT8S;                /* Signed     8 bit quantity */
typedef unsigned int   INT16U;               /* Unsigned   16 bit quantity */
typedef signed   int   INT16S;               /* Signed     16 bit quantity */
typedef unsigned long  INT32U;               /* Unsigned   32 bit quantity */
typedef signed   long  INT32S;               /* Signed     32 bit quantity */
typedef float        FP32;                   /* Single precision floating point */
typedef double       FP64;                   /* Double precision floating point */

typedef unsigned int   OS_STK;                /* Each stack entry is 16-bit wide*/
typedef unsigned int   OS_CPU_SR; /* Define size of CPU status register (SR = 16 bits) */

/*****

*
                                MSP430 (实模式, 大模式编译)
*

*方法 #1:   用简单指令开关中断。

*
                                注意, 用方法1关闭中断, 从调用函数返回后中断会重新打开!

*

*方法 #2:   中断的势能与否与先前的中断状态有关, 比如, 在进入临界段前中断势能关闭的话, 退出临界段时中断势能仍然关闭。

*

*

*方法 #3   中断势能与否与先前的中断状态有关。先将状态寄存器SR存储到局部变量CPU_SR中, 然后关闭中断, UCOS-II在需要关闭中断的地方都分配一个局部变量CPU_SR, 最后通过将CPU_SR的值复制到状态寄存器来恢复中断势能状态。

*****/

#define OS_CRITICAL_METHOD    3

#if      OS_CRITICAL_METHOD == 1

#define OS_ENTER_CRITICAL()  _DINT()    /* Disable interrupts*/
#define OS_EXIT_CRITICAL()   _EINT()    /* Enable interrupts*/
```

```
#endif

#if OS_CRITICAL_METHOD == 2
#define OS_ENTER_CRITICAL()      /* Disable interrupts*/
#define OS_EXIT_CRITICAL()       /* Enable interrupts*/
#endif

#if OS_CRITICAL_METHOD == 3
#define OS_ENTER_CRITICAL() (cpu_sr = OSCPU_SaveSR())
#define OS_EXIT_CRITICAL() (OSCPU_RestoreSR(cpu_sr))
#endif

/*****
 *      MSP430 (实模式, 大模式编译)
 *****/

#define OS_STK_GROWTH 1          /* 堆栈由高地址向低地址增长 */

#define OS_TASK_SW()             OSCtxSw() /*任务切换函数*/

/*****
 *
 *      全局变量
 *****/

OS_CPU_EXT OS_STK *OSISRStkPtr;    /*中断堆栈指针*/

/*****
 *
 *      定义外部函数
 *****/

OS_CPU_SR OSCPU_SaveSR(void);      /*保存状态寄存器SR*/
void OSCPU_RestoreSR(OS_CPU_SR cpu_sr); /*恢复状态寄存器SR*/
```

4.1 数据类型

由于不同的处理器有不同的字长, $\mu\text{C}/\text{OS-II}$ 的移植需要重新定义一系列的数据结构。使用 AQ430 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, 尽管 $\mu\text{C}/\text{OS-II}$ 中没有用到浮点类型的数, 在源代码中笔者还是提供了浮点类型的定义。

由于在 MSP430 实模式中堆栈都是按字进行操作的, 没有字节操作, 所以 AQ430 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。

4.2 代码临界段

与其他实时系统一样, $\mu\text{C}/\text{OS-II}$ 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。啊请 0 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。 $\mu\text{C}/\text{OS-II}$ 定义了两个宏用来关闭/打开中断: OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()。此处, 笔者为用户提供三种种开关中断的方法, 如下所述的方法 1、方法 2 和方法 3。作为一种测试, 本书采用了方法 1。当然, 您可以自由决定采用那种方法。

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 (_DINT()) 和打开 (_EINT()) 中断指令。但这种方法有一个隐患, 如果在关闭中断后调用 $\mu\text{C}/\text{OS-II}$ 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, 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

中断的势能与否与先前的中断状态有关, 不作任何处理。比如, 在进入临界段前中断势能关闭的话, 退出临界段时中断势能仍然关闭。

方法 3

执行 OS_ENTER_CRITICAL() 的第三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 $\mu\text{C}/\text{OS-II}$ 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 $\mu\text{C}/\text{OS-II}$ 函数, 其实是延长了中断响应时间。虽然 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 可以保护代码的临界段。但如此用法要小心, 特别是在调用 OSTimeDly() 一类函数之前关闭了中断。此时任务将处于延时挂起状态, 等待时钟中断, 但此时时钟中断是禁止的! 则系统可能会崩溃。很明显, 所有的 PEND 调用都会涉及到这个问题, 必须十分小心。所以建议用户调用 $\mu\text{C}/\text{OS-II}$ 的系统函数之前打开中断。

4.3 堆栈增长方向

MSP430 处理器的堆栈是由高地址向低地址方向增长的, 所以常量 OS_STK_GROWTH 必须设置为 1。

4.4 OS_TASK_SW()

在 μ C/OS-II 中, 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。中断服务程序 (ISR) 的入口点必须指向汇编函数 OSCtxSw() (请参看文件 OS_CPU_A.ASM)。

5 OS_CPU_A.ASM

μ C/OS-II 的移植需要用户改写 OS_CPU_A.ASM 中的四个函数:

```
OSStartHighRdy()  
OSCtxSw()  
OSIntCtxSw()  
OSTickISR()
```

实际上, 由于 AQ430 允许在 C 语句中内嵌汇编语句, 所以 OS_CPU_A.ASM 中的代码可与 OS_CPU_C.C 合在一起。为了更清楚的理解 uCOS 在 MSP430 上的 PORT, 我们最终还是采用分开形式。

5.1 OSStartHighRdy()

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 (请参考 OSTaskCreate() 和 OSTaskCreateExt() 函数)。OSStartHighRdy() 默认指针 OSTCBHighRdy 指向优先级最高就绪任务的任务控制块 (OS_TCB) (在这之前 OSTCBHighRdy 已由 OSStart() 设置好了)。很明显, OSTCBHighRdy->OSTCBStkPtr 指向的是任务堆栈的顶端。

函数 OSStartHighRdy() 的代码见程序清单 5-1。

程序清单5-1 OSStartHighRdy().

```
.pseg code,common    ;可重定位段  
_OSStartHighRdy  
    call    #_OSTaskSwHook  
    mov.b   #1, &_OSRunning  
    mov.w   SP, &_OSISRStkPtr    ; 保存中断堆栈  
    mov.w   &_OSTCBHighRdy, R13  ; 装载最高优先级任务堆栈
```



```

mov.w    @R13, SP
POPALL                                ; 恢复所有工作寄存器
reti                                           ; 模拟中断返回

```

5.2 OSCtxSw()

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（例如试图唤醒了一个优先级更高的任务），则在函数的末尾会调用OSSched()，如果OSSched()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OSTCBHighRdy，然后通过宏OS_TASK_SW()执行软中断进行任务切换。注意到在此过程中，变量OSTCBCur始终包含一个指向当前运行任务OS_TCB的指针。程序清单5-2为OSCtxSw()的代码。

函数和一般函数的区别在于，调用它时需要保存任务环境，并以中断形式返回，即仿效一次中断。任务环境保存完后，将调用用户定义的对外接口函数OSTaskSwHook()。请注意，此时OSTCBCur指向当前任务OS_TCB，OSTCBHighRdy指向新任务的OS_TCB。在OSTaskSwHook()中，用户可以访问这两个任务的OS_TCB。如果不使用对外接口函数，请在头文件中把相应的开关选项关闭，加快任务切换的速度。

程序清单5-2 OSCtxSw().

```

_OSCtxSw
    push    sr                        ;通过保存状态寄存器效仿中断
    PUSHALL                            ;所有工作寄存器压入堆栈
    mov.w   &_OSTCBCur, R13           ; OSTCBCur->OSTCBStkPtr = SP
    mov.w   SP, 0(R13)
    call    #_OSTaskSwHook           ; 调用用户定义的对外接口函数
    mov.b   &_OSPrioHighRdy, R13      ; OSPrioCur = OSPrioHighRdy
    mov.b   R13, &_OSPrioCur         ;
    mov.w   &_OSTCBHighRdy, R13      ; OSTCBCur = OSTCBHighRdy
    mov.w   R13, &_OSTCBCur          ;
    mov.w   @R13, SP                 ; SP = OSTCBHighRdy->OSTCBStkPtr

    POPALL                            ; 弹出所有工作寄存器
    reti

```

从对外接口函数OSTaskSwHook()返回后，由于任务的更替，变量OSTCBHighRdy被拷贝到OSTCBCur中，同样，OSPrioHighRdy被拷贝到OSPrioCur中。需要注意的是在运行OSCtxSw()和OSTaskSwHook()函数期间，中断是禁止的。

5.3 OSIntCtxSw()

在 $\mu\text{C}/\text{OS-II}$ 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。

程序清单 5-3 给出的代码大部分与 `OSCtxSw()` 的代码相同, 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; 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。

程序清单5-3 OSIntCtxSw()。

```
_OSIntCtxSw
    call    #_OSTaskSwHook
    mov.b   &_OSPrioHighRdy, R13    ; OSPrioCur = OSPrioHighRdy
    mov.b   R13, &_OSPrioCur      ;
    mov.w   &_OSTCBHighRdy, R13    ; OSTCBCur = OSTCBHighRdy
    mov.w   R13, &_OSTCBCur        ;
    mov.w   @R13, SP               ; SP = OSTCBHighRdy->OSTCBStkPtr

    POPALL                          ; pop all registers
    reti
```

当中断发生后, CPU在完成当前指令后, 进入中断处理过程。首先是保存现场, 将返回地址压入当前任务堆栈, 然后保存状态寄存器的内容。接下来CPU从中断向量处找到中断服务程序的入口地址, 运行中断服务程序。在 $\mu\text{C}/\text{OS-II}$ 中, 要求用户的中断服务程序在开头保存CPU其他寄存器的内容。此后, 用户必须调用`OSIntEnter()`或着把全局变量`OSIntNesting`加1。此时, 被中断任务的堆栈中保存了任务的全部运行环境。在中断服务程序中, 有可能引起任务就绪状态的改变而需要任务切换, 例如调用了`OSMboxPost()`, `OSQPostFront()`, `OSQPost()`, 或试图唤醒一个优先级更高的任务(调用`OSTaskResume()`), 还可能调用 `OSTimeTick()`, `OSTimeDlyResume()` 等等。

$\mu\text{C}/\text{OS-II}$ 要求用户在中断服务程序的末尾调用`OSIntExit()`, 以检查任务就绪状态。在调用`OSIntExit()`后, 返回地址会压入堆栈中。

进入`OSIntExit()`后, 由于要访问临界代码区, 首先关闭中断。由于`OS_ENTER_CRITICAL()`可能有不同的操作, 状态寄存器的内容有可能被压入堆栈。如果确实要进行任务切换, 指针`OSTCBHighRdy`将指向新的就绪任务的`OS_TCB`, `OSIntExit()`会调用`OSIntCtxSw()`完成任务切换。注意, 调用`OSIntCtxSw()`会在再一次在堆栈中保存返回地址。在进行任务切换的时候, 我们希望堆栈中只保留一次中断发生的任务环境, 而忽略掉由于函数嵌套调用而压入的一系列返回地址。忽略的方法也很简单, 只要把堆栈指针加一个固定的值就可以了。

一旦堆栈指针重新定位后, 就被保存到将要被挂起的任务`OS_TCB`中, 在 $\mu\text{C}/\text{OS-II}$ 中(包括 $\mu\text{C}/\text{OS}$), `OSIntCtxSw()`是唯一一个与编译器相关的函数, 也是用户问的最多的。如果您的系统移植后运行一段时间后会死机, 就应该怀疑是`OSIntCtxSw()`中堆栈指针重新定位的

问题。

当当前任务的现场保存完毕后,用户定义的对外接口函数OSTaskSwHook()会被调用。注意到OSTCBCur指向当前任务的OS_TCB,OSTCBHighRdy指向新任务的OS_TCB。在函数OSTaskSwHook()中用户可以访问这两个任务的OS_TCB。如果不用对外接口函数,请在头文件中关闭相应的开关选项,提高任务切换的速度。

从对外接口函数OSTaskSwHook()返回后,由于任务的更替,变量OSTCBHighRdy被拷贝到OSTCBCur中,同样,OSPrioHighRdy被拷贝到OSPrioCur中。此时,OSIntCtxSw()将载入新任务的CPU环境,

需要注意的是在运行OSIntCtxSw()和用户定义的OSTaskSwHook()函数期间,中断是禁止的。

5.4 OSTickISR()

时钟节拍由WDT产生,当然,只要你喜欢,也可以用其他方式产生。在这个例子中,我们将时钟节拍间隔设定为32ms。时钟节拍的中断向量为0xFF4,μC/OS-II将此向量截取,指向了μC/OS的中断服务函数OSTickISR()。

在程序清单5-4给出了函数OSTickISR()的伪码。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。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。

程序清单5-5给出了OSTickISR()的完整代码。

程序清单5-4 OSTickISR() 伪码.

```
void OSTickISR (void)
{
    Save processor registers;

    OSIntNesting++;

    OSTickDOSctr--;

    if (OSTickDOSctr == 0) {
        Chain into DOS by executing an 'INT 81H' instruction;
    } else {
        Send EOI command to PIC (Priority Interrupt Controller);
    }
}
```

```

    OSTimeTick();

    OSIntExit();

    Restore processor registers;

    Execute a return from interrupt instruction (IRET);
}

```

程序清单5-5 *OSTickISR()*.

```

_WDT_ISR                                     ; 看门狗定时器中断服务程序

    PUSHALL

    bic.b    #0x01, IE1                     ; 不允许看门狗定时器中断
    cmp.b    #0, &_OSIntNesting             ; if (OSIntNesting == 0)
    jne      WDT_ISR_1
    mov.w    &_OSTCBCur, R13                 ; 保护任务堆栈
    mov.w    SP, 0(R13)
    mov.w    &_OSISRStkPtr, SP              ; 装载中断堆栈

WDT_ISR_1
    inc.b    &_OSIntNesting                 ; OSIntNesting ++
    bis.b    #0x01, IE1                     ; 看门狗定时器中断使能
    EINT
    call     #_OSTimeTick
    DINT
    call     #_OSIntExit
    cmp.b    #0, &_OSIntNesting             ; if (OSIntNesting == 0)
    jne      WDT_ISR_2
    mov.w    &_OSTCBHighRdy, R13            ; 恢复任务堆栈
    mov.w    @R13, SP

WDT_ISR_2
    POPALL
    Reti

.pseg wdt_vector,abs= 0xFFE0+WDT_VECTOR     ; 定义WDT中断向量
.data _WDT_ISR

```

6 OS_CPU_C.C

μC/OS-II 的移植需要用户改写OS_CPU_C.C中的六个函数:

```
OSTaskStkInit()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskSwHook()
OSTaskStatHook()
OSTimeTickHook()
```

实际需要修改的只有 OSTaskStkInit() 函数, 其他五个函数需要声明, 但不一定有实际内容。这五个函数都是用户定义的, 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。

6.1 OSTaskStkInit()

该函数由 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。

当调用 OSTaskCreate() 或 OSTaskCreateExt() 创建一个新任务时, 需要传递的参数是: 任务代码的起使地址, 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() (程序清单6-1) 只需要以上提到的3个参数 (task, pdata, 和ptos)。

程序清单6-1 OSTaskStkInit()。

```
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *p_arg, OS_STK *ptos,
INT16U opt)
{
    INT16U *top;
    opt    = opt;
    top    = (INT16U *)ptos;
    top--;
    *top = (INT16U)task;
    top--;
    *top = (INT16U)task;                      /* Interrupt return pointer */
    top--;
    *top = (INT16U)0x0008;                    /* Status register */
    top--;
    *top = (INT16U)0x0404;
    top--;
    *top = (INT16U)0x0505;
    top--;
    *top = (INT16U)0x0606;
```

```
top--;
*top = (INT16U) 0x0707;
top--;
*top = (INT16U) 0x0808;
top--;
*top = (INT16U) 0x0909;
top--;
*top = (INT16U) 0x1010;
top--;
*top = (INT16U) 0x1111;
top--;
*top = (INT16U) p_arg; /* Pass 'p_arg' through
register R12*/ top--;
*top = (INT16U) 0x1313;
top--;
*top = (INT16U) 0x1414;
top--;
*top = (INT16U) 0x1515;
return ((OS_STK *)top);
}
```

由于MSP430 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。

我们使用的AQ430编译器配置为用堆栈而不是寄存器来传送参数pdata，此时参数pdata的段地址和偏移量都将被保存在堆栈中。

堆栈中紧接着是任务函数的起始地址，理论上，此处应该为任务的返回地址，但在 μ C/OS-II中，任务函数必须为无限循环结构，不能有返回点。

返回地址下面是状态字(SR)，设置状态字也是为了模拟中断发生后的堆栈结构。堆栈中的SR初始化为0x0008。

如果确实需要突破上述限制，可以通过参数pdata向任务传递希望实现的中断状态。如果某个任务选择启动后禁止中断，那么其他的任务在运行的时候需要重新开启中断。同时还要修改OSTaskIdle()和OSTaskStat()函数，在运行时开启中断。如果以上任何一个环节出现问题，系统就会崩溃。

堆栈中还要留出各个寄存器的空间。

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。

6.2 OSTaskCreateHook()

OS_CPU_C.C中未定义，此函数为用户定义。

6.3 OSTaskDelHook()

OS_CPU_C.C中未定义, 此函数为用户定义。

6.4 OSTaskSwHook()

OS_CPU_C.C中未定义, 此函数为用户定义。其用法请参考例程3。

6.5 OSTaskStatHook()

OS_CPU_C.C中未定义, 此函数为用户定义。其用法请参考例程3。

6.6 OSTimeTickHook()

OS_CPU_C.C 中未定义, 此函数为用户定义。

7 移植代码正确性验证

我们所有的移植代码都是针对 AQ430 写的, 这一部分介绍在 AQ430 编译环境下的具体实践。你可以直接打开已经过调试的移植代码测试项目文件 uCOS_AQ430_PORT.qpj (在名为“PORT_AQ430_449”目录中)进行测试。

具体步骤为:

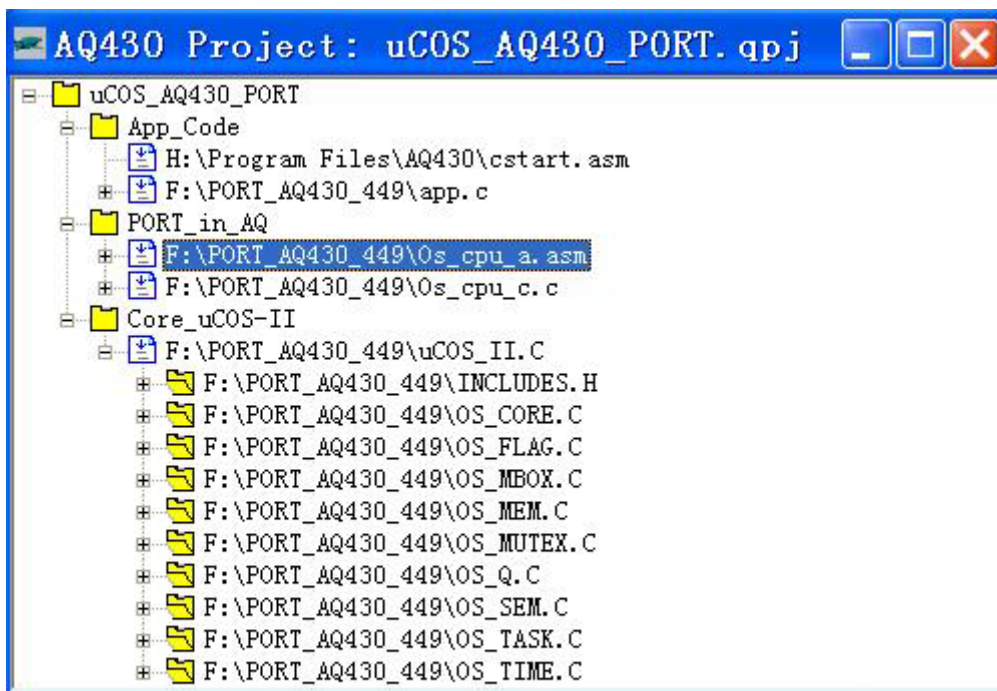


图 7-1

- 1) 选择 **Project->open**, 打开项目文件 `uCOS_AQ430_PORT.qpj`, 出现如图 7-1 所示界面。从图 7-1 可以看到, 我们把代码分为三部分, 分别属于三个不同的组。组 `Core_uCOS-II` 中所包含的是 `uCOS-II` 的内核源代码, 点击 `uCOS-II.c` 下相应的文件可以查看内核源代码。组 `PORT_in_AQ` 中包含的是 `uCOS-II` 在 `AQ430` 环境下的 `MSP430` 移植代码。组 `App_Code` 组中包含的是应用程序, 这里的应用程序没有特别意义, 仅是用来测试移植的正确性。
- 2) 选择 **Build** 菜单下的各选项进行编译、连接、下载等
- 3) 进行调试。

7.1 移植代码在 AQ430 环境下的验证过程

这里说明从创建一个项目到进行调试验证的全过程。移植代码可以在其他编辑环境中编辑, 创建项目中把它们添加进来, 也可在项目建成后在 `AQ430` 环境中进行编辑。

- 1) 打开 `AQ430`, 选择 **Project->New**, 出现如图 7-2 所示窗体。

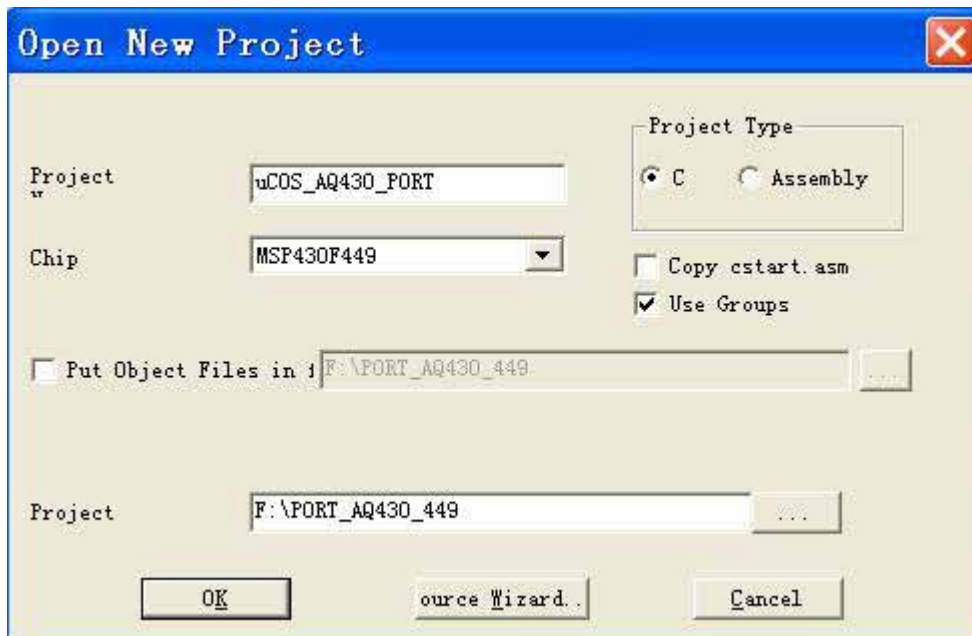


图 7-2

在图 7-2 所示的窗体中, 我们选择芯片信号为 `MSP430F449`, 项目路径为 `F:\PORT_AQ430_449`, 项目名称为 `uCOS_AQ430_PORT`, 选择项目类型为 `C`, 并选择使用组 (选择 `Use Groups`), 使用组后我们可以把代码形式上分开, 如分为不需要修改的内核代码, 需要修改的移植代码合用户自己的应用程序, 便于理解。

- 2) 点击图 7-2 窗体中的 “OK”, 进入如图 7-3 所示界面。

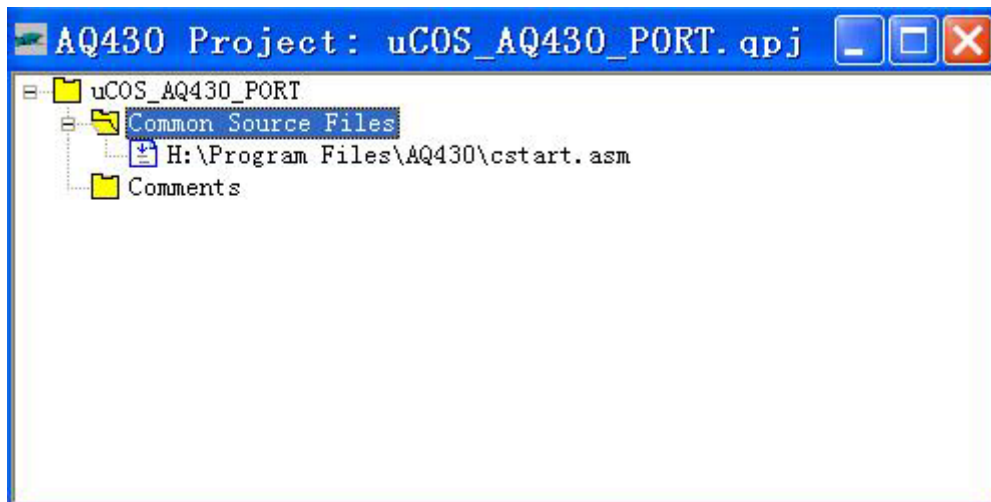


图 7-3

从图 7-3 可以看到, 由于我们选择的项目类型是 C, 所以 AQ430 会自动添加 cstart.asm 文件到项目中。我们可以把组名修改为自己喜欢的名称, 也可以通过选择 **Project->Add Group** 创建新的组。在这个例子中, 我把组分别命名为 Core_uCOS-II、PORT_in_AQ 和 App_Code。命名后通过 **Project->Add Files** 把对应的代码文件添加到各个组中, 如图 7-4 所示。

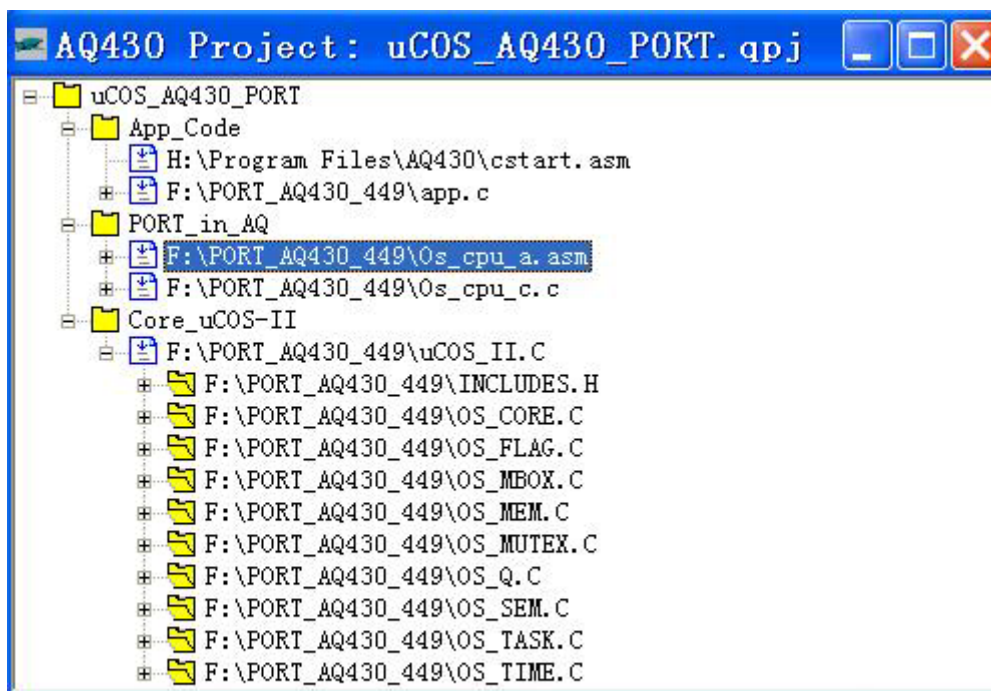


图 7-4

- 3) 选择 **Build->Make, Don't load**, 察看编译是否出错。
- 4) 选择 **Build->Make/Link** 等下载程序调试。

在应用操作系统时, 首先要验证移植代码、内核源代码的正确性。应先编写简单的任务进行调试, 等确定移植无误、操作系统工作正常后, 再进行应用程序的调试, 否则, 遇到问题时不能确定问题在于操作系统还是应用程序, 给调试增加难度。

本例中, 我们创建了一个任务 (目标系统可用我们的 449 学习板), 反复在 P1.0 口输出高低电平, 使接在 P1.0 上的 LED 以固定的时间间隔不停的闪烁, 由此来验证移植代码和操

作系统任务调度等的正确。应用程序在组 App_Code 的 app.c 文件中。