**Link->https://blog.csdn.net/u010383937/article/details/72236671**

**vxWorks内核解读四--中断**

2017年05月16日 11:00:07 [傲世阿龍](https://me.csdn.net/u010383937" \t "_blank) 阅读数：955更多

个人分类： [vxWorks](https://blog.csdn.net/u010383937/article/category/6891568" \t "_blank)

接下来我们讨论一下Wind内核的中断处理模块，中断是操作系统内核设计中非常重要的部分。由于周期性和非周期性任务的按时执行都离不开中断，并且大多数实时任务的调度都是由中断引发的，中断管理对于实时系统来说不仅重要而且要求更高。因此，实时系统要求操作系统具备迅速响应外部中断的能力。

本篇我以x86平台的Pentium处理器为例，介绍Wind内核的中断处理框架，以及中断栈幁的设计。通过本篇的分析，我们可以看出Wind内核作为一款优秀的实时内核，到底优秀在什么地方，其为什么会有快速的中断响应时间。

**4.1 Pentium处理器概述**

**4.1.1 Pentium CPU的中断类型**

有两类事件可引起Pentium处理器挂起当前的指令流，即中断和异常。中断是由外部事件引发的,在程序执行的任何时刻都可能出现;异常也称异常中断,是由内部事件引发的。中断和异常各有两类触发源:

**(1)中断**

**可屏蔽中断:**CPU的INTR引脚收到有效信号,如果Pentium标志寄存器IF位为1,则允许中断,否则信号在CPU内被屏蔽。

**非屏蔽中断:**CPU的NMI引脚收到有效信号而引发的中断,这类中断不能被阻止。

**(2)异常。**

**执行异常:**CPU试图执行一条指令的过程中出现错误、故障等不正常条件而引发的异常中断。

**执行软件中断指令:**Pentium指令系统中包括一些如INTO,INT n这类软件中断指令,执行时产生异常中断。

详细分类的话,Pentium处理器可以识别256种中断和异常。每种中断给予一个编号,即0～255,称为中断向量号(interrupt vector number)。其中NMI、异常以及系统保留占用中断向量号为0～31,而32～255为用户中断向量号,可供INTR和自定义软件中断(如汇编中的INT指令)使用。

我们把x86中0~31号中断称为同步中断(又称软中断)，由于Pentium平台使用PIC是两片i8259A级联而成，因此可用于外部中断线只有16个，因此我们把32~47号中断称为异步中断(又称硬中断)。

同步中断是由指令引发的，发生在引发陷阱指令的执行过程中，由CPU内部或者指令产生的外部信号引发。若在执行第i条指令时引发同步陷阱(软中断)，那么同步陷阱(软中断)返回的仍为第i条指令。

异步陷阱(硬中断)是指外部事件引发的中断，和CPU执行的指令无关，在逻辑上发生在两条指令之间。若在执行第i条指令时发生异步陷阱，那么异步陷阱处理的返回地址是第i+1条指令。

**4.1.2 Pentium CPU的中断响应过程**

中断处理子程序的入口地址信息存于内存中的一个表内,实模式为中断向量表IVT,保护模式为中断描述符表IDT。中断发生时,CPU首先通过某种方式获得中断向量号,再以中断向量号检索此表,即可获取中断服务子程序入口地址,详述如下:

**(1)实模式使用中断向量表。**

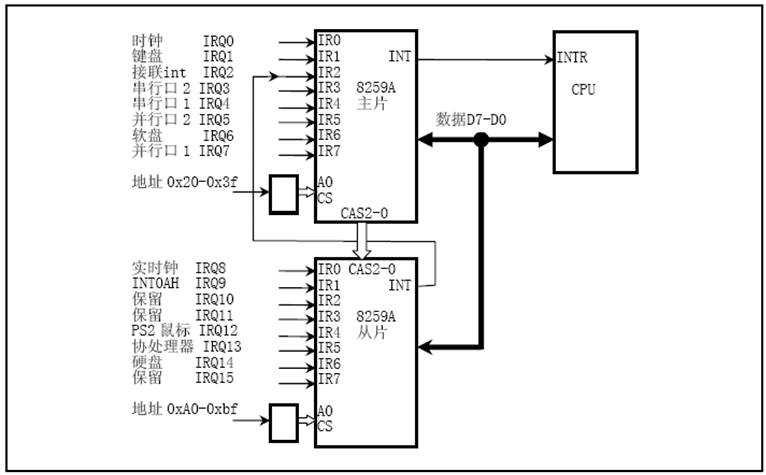
中断向量表IVR的基地址由IDTR(中断描述符寄存器)指定,大小为1kB。中断响应时的查表过程与8086/8088一致,在此不再赘述。

**(2)保护模式使用中断描述符表。**

中断描述符表(IDT)的基地址也由IDTR指定,大小为2kB。中断描述符表每一表项对应一个中断向量号,但表项称为中断门描述符或陷阱门描述符。这些门描述符为8字节长,对应256个中断向量号。以中断向量号乘以8作为访问IDT的偏移,读取相应的中断门/陷阱门描述符表项。门描述符给出中断服务子程序人口地址(段:偏移),其中32位偏移量装入EIP,16位的段值被装入CS寄存器。但此段值是选择符,CPU会自动查GDT或LDT取得代码段描述符并送到相应的描述符寄存器中。

**4.1.3 X86架构的计算机对外部中断的管理**

在嵌入式应用中,人们感兴趣的主要是指由硬件信号触发的非屏蔽中断与可屏蔽中断。在单CPU的X86计算机中,采用两片i8259A级联来管理16个可屏蔽外部中断,由于主i8259A的IRQ2用于级联,所以实际可用的IRQ只有15个,其中又有一些被系统占用,这种逻辑如今已被集成在主板芯片组的南桥中,如图4.1所示。由于传统的PIC提供的中断资源较少,现代PC开始采用APIC(高级可编程中断控制器)管理外部中断,它的一个显著优点是能够扩充系统可用的IRQ资源。本章基于传统的PIC结构(两片i8259A级联的结构)来分析Wind内核的中断处理框架。

**图4.1 两片i8259A级联的中断控制器**

在Wind内核中与中断相关的有以下四个概念：

**外部中断号IRQNumber：0**~15,外部中断号由两片i8259A级联构成的PIC控制器的输入引脚引入，主i8259A的8条中断线对应外部中断0~7，从i8259A对应中断8~15。

**内部中断号INumber:**0~255，是响应的中断描述符在中断描述符表中的索引。

内中断号IRQNumber(i)=i+INT\_NUM\_IRQ0=i+32,其中i=0，… ,15, 在Wind内核中由INT\_NUM\_GET(irq)宏实现外部中断号与内部中断号的转换

**中断向量号iVector：**相应的中断描述符在中断描述符表中的偏移量，由于每个中断描述符为8个字节，所有IVector=INumber\*8

INUM\_TO\_IVEC(intNum)：实现内部中断号与内部向量号之间的转换。

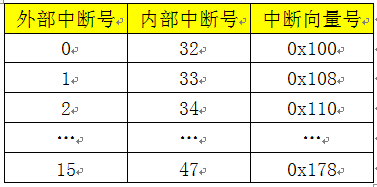
INT\_NUM\_GET(irq)：实现外部中断号与内部中断号的转换

**中断向量号iVector：**相应的中断描述符在中断描述符表中的偏移量，由于每个中断描述符为8个字节，所有IVector=INumber\*8

INUM\_TO\_IVEC(intNum)：实现内部中断号与内部向量号之间的转换。

外部中断号、内部中断号、以及中断向量号的对应关系如表4.1所示。

**表4.1 中断相关概念对照表**

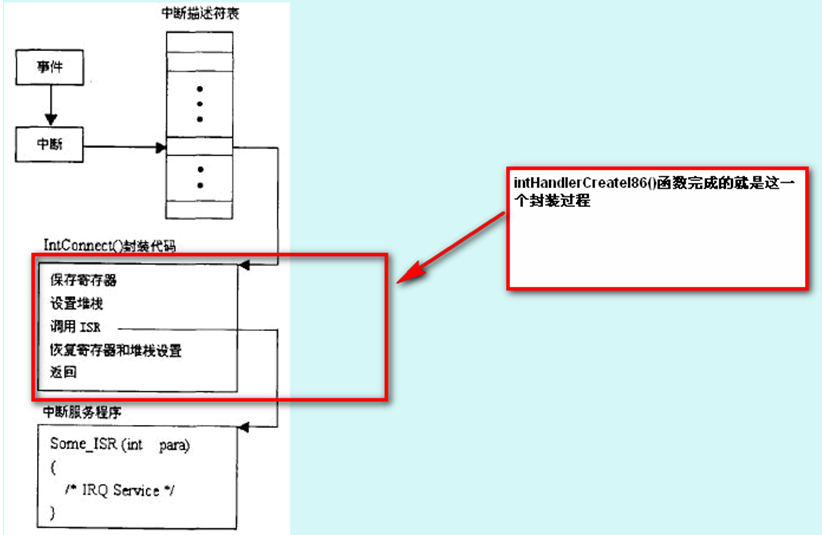


X86架构的计算机中,一些中断资源已经固定地分配给某些外部设备,如系统时钟固定使用IRQ0,所以在选择中断号时首先应参考硬件手册,避免与已用的中断资源冲突。选定中断号后,需要在BIOS中加以设置。避免BIOS在初始化时,把此中断号作为可用资源分配给PCI设备,造成中断冲突。

备注：VxWorks中使用intConnect()挂接中断服务程序,但对于PCI设备,一般采用pciIntConnect()挂接中断,它与intConnect()的主要不同在于intConnect()使用的中断向量是独占的,而pciIntConnect()则可使多个外部中断共享一个中断向量。它在内部使用一个链表管理多个ISR,发生中断时,链接在一个链表上的各个ISR被依次调用,pciIntConnect()要求每个ISR被调用时,应该首先查询是否为自己的设备产生的中断,不是则应立即返回,以继续调用其它ISR。

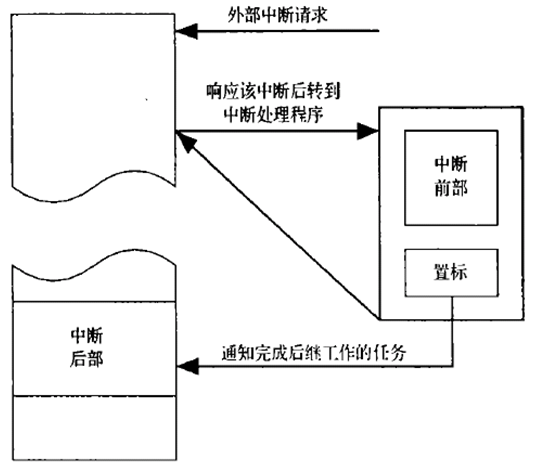
**4.2 Wind内核中断处理流程**

在Pentium平台，Wind内核运行在保护模式下。在VxWorks中,可以采用intConnect()关联中断服务程序至某个中断向量。然而intConnect()并不是直接将用户设计的ISR与中断门描述符相关联,而是对它加了一层封装,然后将封装代码的内存首地址与中断门描述符相关联,中断响应过程如下图4.2所示。



**图4.2 Wind内核中断封装过程**

Wind内核的中断处理模型为“前后段”模型，如图4.3所示，这种中断管理的思想是把中断处理按照重要性分成两部分(前段和后段)，前段的优先级最高，并且在关中断的条件下执行，仅做一些必要的硬件操作，随后设置一些相关的标志位，所以执行的时间非常短；而把流程更复杂、可以延时操作并且影响不大的部分归为后段，这一部分在退出中断处理程序后实施，后段为用户或者系统默认的ISR。通过这样的中断处理机制可以减少中断服务时间，为其它外部事件的中断提供更多的时机。如在一些I/O处理部分，由I/O操作引发的中断处理部分只作标记功能，即只设一个标志或者发一个消息说明外部中断来了，而具体的I/O传输操作放在中断处理外部实施。



**图4.3 "前后段”终端处理模型**

图中“中断前部”完成当外部事件发出中断请求时，系统对其响应所需的必须功能，比如中断现场保护等，“置标”部分主要通知某个任务已经有一个中断发生，且中断前部已经完成；“中断后部”并不是在中断处理程序中完成由接收到标记或者接收到通知的任务完成，主要完成本应该在中断处理程序中完成的工作(其实uC/OS-III也采用了这种处理机制)。

Wind内核的“前后段”处理机制能在最短的时间内对中断作出响应，并且在不影响接收中断的情况下，对已经响应的中断作出及时的处理。

例如采用intConnect()为ISA总线设备关联中断服务程序IRQ\_ISR至IRQ10的程序片断如下:

#define IRQNum 10

……

if(intConneet((VOIDFUNCPTR)INUM\_TO\_IVEC(IRQNum+0x20),IRQ\_ISR,0)==OK)

{

         if(sysIntEnablePIC(IRQNum)==OK)

         {

                   printf("Succeeded.\n");

         }

}

void IRQ\_ISR()

{

         int intLockKey;

         intLockKey=intLock();  //关中断

         ……(critical section)   //执行临界区代码

         intUnlock(intLocKey);  //开中断

}

备注：上面的例子，我演示了一个简单的中断挂接过程，其中中断ISR是IRQ\_ISR()函数，在这个例子为了简化起见我假设IRQ\_ISR()的所有处理代码都是临界区代码，其中中断处理ISR中并不是所有的代码都是要关中断的，可以把其中不需要里面处理的部分作为一个Job，放入内核工作队列中延后处理，这样可以极大的降低中断时延。后面分析的时钟中断处理流程就是采用这种方式。

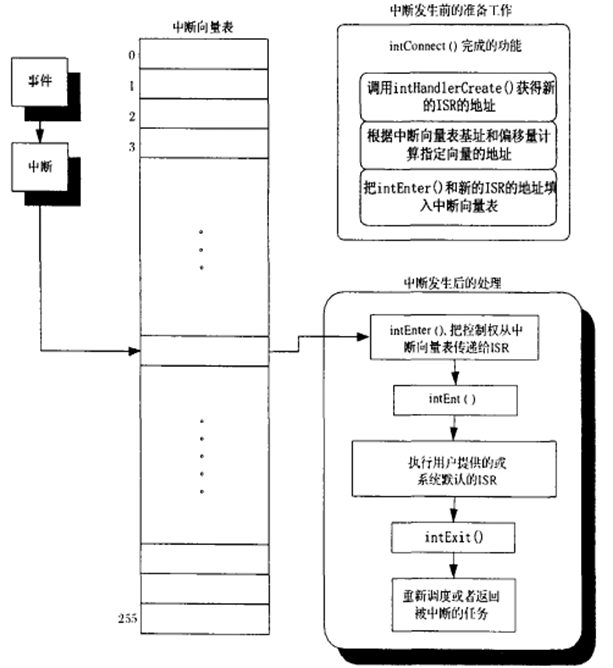
另外用intConneet()挂中断后,还必须用sysIntEnablePIC()使能中断,实际上就是针对某个IRQ,置i8259A中断屏蔽寄存器中的相应位为允许。

接下来我Wind内核的时钟中断为例，来分析Wind内核的中断处理过程。

**4.3 VxWorks中断处理机制**

Wind内核接管中断的处理过程主要包括两个部分：面向应用的编程接口和面向底层的处理。面向应用的编程接口的任务之一是支持用户安装ISR。面向底层的处理分成两个部分：中断向量表部分和中断处理部分。中断向量表部分主要指的是中断向量表的定位和向量表中表项内容的形式，一般在RTOS内核中提供一个中断向量表，其表项的向量号和CPU中描述的向量对应；向量表采取两种形式，一种是在向量表的位置处存储几条跳转指令，转到具体的中断处理部分；另一种形式是中断向量的位置存放具体的ISR，但这种方式仅仅针对向量号之间彼此具有一定的空间，足以存放ISR。

面向底层部分的中断处理是内核中断处理的核心，Wind的中断处理流程如图4.4所示。



**图4.4 Wind内核中断处理流程**

**4.3.1 中断发生前的准备工作**

VxWorks中面向应用的编程接口为intConnect(),其原型如下：

STATUS intConnect( VOIDFUNCPTR \*vector,VOIDFUNCPTR routine,int parameter)

其中

vector：要挂接的中断向量地址；

routine：中断发生时调用ISR；

parameter：传递给IST的参数。

它允许将任务C函数作为ISR挂接到指定的中断向量vector上，VxWorks的Pentium平台挂接时钟中断的代码片段如下：

(void)intConnect (INUM\_TO\_IVEC (INT\_NUM\_GET (PIT0\_INT\_LVL)), sysClkInt, 0);

其中PIT0\_INT\_LVL值为0，

INT\_NUM\_GET (PIT0\_INT\_LVL)值为0+32=32，

INUM\_TO\_IVEC (INT\_NUM\_GET (PIT0\_INT\_LVL)置为32\*8=0x100=256.

即将中断描述符表IDT的第32个表项的入口地址设置位sysClkInt()函数的地址。

intConnect()的具体实现如下：

STATUS intConnect( VOIDFUNCPTR \*vector,  VOIDFUNCPTR routine,          int parameter)

{

    FUNCPTR intDrvRtn;

    VOIDFUNCPTR routineBoi;

    VOIDFUNCPTR routineEoi;

    int parameterBoi;

    int parameterEoi;

    if (intEoiGet == NULL)

         {

        intDrvRtn = intHandlerCreate ((FUNCPTR)routine, parameter);

         }

    else

         {

        (\* intEoiGet) (vector, &routineBoi, &parameterBoi,

                                   &routineEoi, &parameterEoi);

        intDrvRtn = intHandlerCreateI86 ((FUNCPTR)routine, parameter,

                                                (FUNCPTR)routineBoi, parameterBoi,

                                                (FUNCPTR)routineEoi, parameterEoi);

         }

    if (intDrvRtn == NULL)

         return (ERROR);

    /\* make vector point to synthesized code \*/

    intVecSet ((FUNCPTR \*)vector, (FUNCPTR)intDrvRtn);

    return (OK);

}

在Pentium平台，VxWorks同时通过usrInit()->sysHwInit()重新设置了intEoiGet函数指针为：

intEoiGet = sysIntEoiGet;

因此intConnect()首先调用了sysIntEoiGet()函数来获取在中断ISR，即执行sysClkInt()函数调用前后的打桩例程，即在执行sysClkInt()前调用(\*routineBoi)(parameterBoi),执行sysClkInt()后执行(\*routineEoi)(parameterEoi)。

具体到Pentium平台的时钟中断来说routineBoi=NULL，routineEoi=i8259IntEoiMaster，parameterEoi=irqNo其中VOID i8259IntEoiMaster(INT32 irqNo)向主i8259A中断控制器发送EOI(end of interrupt)信号。

intConnect()接着调用intHandlerCreateI86()函数来获取新的ISR的地址；这个新的ISR除了在sysClkInt()前调用(\*routineBoi)(parameterBoi),在sysClkInt()后调用(\*routineEoi)(parameterEoi)，并且在这个新封装的基础上，进一步在前部封装了intEnt(),在后部封装了intExit()。因此新的ISR的执行流程为：

* intEnt()
* (\*routineBoi)(parameterBoi)
* sysClkInt()
* (\*routineEoi)(parameterEoi()
* intExit()

相对于Pentium平台来说就是

* intEnt()
* sysClkInt()
* i8259IntEoiMaster(irqNo)
* intExit()

从上面的分析我们可以看出intHandlerCreateI86()为中断指定一个C函数作为ISR，并通过调用intVecSet()挂接到指定的中断上。intHandlerCreateI86()在ISR的中断“打桩”：进入该函数之前调用intEnt()和(\*routineBoi)(parameterBoi)，退出ISR之后调用(\*routineEoi)(parameterEoi()和intExit()

“打桩”的代码存是一个存储在一个数组中的PIC(Position Independent Code)指令序列，以Pentium平台为例，其实现如下：

LOCAL UCHAR intConnectCode []   =     /\* intConnect stub \*/

{

/\*

 \* 00  e8 kk kk kk kk                call   \_intEnt                              \* tell kernel

 \* 05  50                                   pushl         %eax                       \* save regs

 \* 06  52                                   pushl         %edx

 \* 07  51                                   pushl         %ecx

 \* 08  68 pp pp pp pp             pushl         $\_parameterBoi      \* push BOI param

 \* 13  e8 rr rr rr rr                     call   \_routineBoi                      \* call BOI routine

 \* 18  68 pp pp pp pp             pushl         $\_parameter           \* push param

 \* 23  e8 rr rr rr rr                    call   \_routine                            \* call C routine

 \* 28  68 pp pp pp pp            pushl         $\_parameterEoi       \* push EOI param

 \* 33  e8 rr rr rr rr                    call   \_routineEoi                       \* call EOI routine

 \* 38  83 c4 0c                        addl $12, %esp                         \* pop param

 \* 41  59                                  popl %ecx                                \* restore regs

 \* 42  5a                                  popl %edx

 \* 43  58                                  popl %eax

 \* 44  e9 kk kk kk kk                jmp  \_intExit    \* exit via kernel

 \*/

     0xe8, 0x00, 0x00, 0x00, 0x00,       /\* \_intEnt filled in at runtime \*/

     0x50,

     0x52,

     0x51,

     0x68, 0x00, 0x00, 0x00, 0x00,       /\* BOI parameter filled in at runtime \*/

     0xe8, 0x00, 0x00, 0x00, 0x00,       /\* BOI routine filled in at runtime \*/

     0x68, 0x00, 0x00, 0x00, 0x00,       /\* parameter filled in at runtime \*/

     0xe8, 0x00, 0x00, 0x00, 0x00,       /\* routine filled in at runtime \*/

     0x68, 0x00, 0x00, 0x00, 0x00,       /\* EOI parameter filled in at runtime \*/

     0xe8, 0x00, 0x00, 0x00, 0x00,       /\* EOI routine filled in at runtime \*/

     0x83, 0xc4, 0x0c,                           /\* pop parameters \*/

     0x59,

     0x5a,

     0x58,

     0xe9, 0x00, 0x00, 0x00, 0x00,       /\* \_intExit filled in at runtime \*/

};

对于Pentium平台的中断中断，其打桩之后的执行序列如下：

* intEnt()
* sysClkInt()
* i8259IntEoiMaster(irqNo)
* intExit()

对于的汇编序列：

call   intEnt

pushl         %eax

pushl         %edx

pushl         %ecx

call   sysClkInt

pushl         $\_parameterEoi

call   \_routineEoi

addl $4, %esp

popl %ecx

popl %edx

popl %eax

jmp  intExit

intHandlerCreateI86()将时钟中断对应的中断ISR(即sysClkInt()函数)的打桩序列在内存中的首地址通过intVecSet()设置到中断描述符表IDT的第32表项(下标从0开始)的跳转地址中。

上述工作完成好，就可以等待中断的发生。

关于时钟中断，我还想多说几句，我在本文上面提供，Wind内核通过usrRoot()->sysClkInit()->sysClkConnect()->sysHwInit2()->intConnect()->intHandlerCreateI86()完成的中断sysClkInit()挂接。sysClkInt()实现如下：

void sysClkInt (void)

{

    if (sysClkRoutine != NULL)

         (\* sysClkRoutine) (sysClkArg);

}

通过代码，我们知道sysClkInt()执行的中断处理函数是(\* sysClkRoutine) (sysClkArg);

sysClkRoutine是在usrRoot()->sysClkInit()->sysClkConnect()中被初始化为usrClock()，挂接好中断，对于时钟中断而言，再设置时钟中断的发生频率后，就可以启动中断了，其代码usrRoot()->sysClkInit()实现如下：

void sysClkInit (void)

{

    sysClkConnect ((FUNCPTR) usrClock, 0); /\* 挂接中断ISR \*/

    sysClkRateSet (SYS\_CLK\_RATE);               /\* 设置系统时钟频率 \*/

    sysClkEnable ();        /\* 启动时钟中断 \*/

}

**4.3.2 中断发生后的处理**

当Wind内核响应 中断时，根据中断号从中断向量表中取出对应的中断向量，然后调用intEnt()。该函数把控制权从中断向量传递给中断ISR。intEnt()完成包括中断响应操作、保持系统寄存器、建立一个C语言的上下文环境。由于intEnt()和具体的平台相关，我们仍以Pentium平台为例，来分析intEnt()的具体实现。

intEnt()函数在中断ISR的入口被调用，由intHandlerCreateI86()封装在桩代码中。

假设我们使用中断栈，即

#define INT\_STACK\_USE

vxIntStackEnabled=TRUE

为了方便我表述，我贴出了相关的代码：

FUNC\_LABEL(intEnt)

         cli                                        /\* 关中段 \*/

         pushl         (%esp)               /\* 将esp指向的ret\_addr在入栈一次 \*/

         pushl         %eax

         movl          FUNC(errno),%eax

         movl          %eax,8(%esp)          /\* 将errno保持到战中返回地址所在的位置 \*/

         incl   FUNC(intCnt)                     /\* intCnt++ \*/

         incl   FUNC(intNest)                   /\* intNest++ \*/

#ifdef        INT\_STACK\_USE

         cmpl    $0,FUNC(vxIntStackEnabled) /\* if vxIntStackEnabled == 0 then \*/

         je      intEnt0                                       /\* skip the interrupt stack switch \*/

         movl          ESF0\_CS+12(%esp), %eax      /\* get CS in ESF0 \*/

         cmpw       FUNC(sysCsInt), %ax                /\* 中断嵌套吗? \*/

         je      intEnt0                                              /\*   是的话:跳过下面的部分 \*/

         cmpl          $1,FUNC(intNest)    /\* 是第一层中断吗? \*/

         jne    intEnt0     /\*如果是的话，切换到中断栈；否的话跳过栈切换过程 \*/

         /\* copy the supervisor stack to the interrupt stack. \*/

intEntStackSwitch:

         pushl         %ecx                            /\* save %ecx \*/

         pushl         %esi                            /\* save %esi \*/

         pushl         %edi                           /\* save %edi \*/

         /\* copy ESF0(12 bytes), errno, return addr, %eax \*/

         subl  $ ESF0\_NBYTES+12+4, FUNC(vxIntStackPtr)  /\* alloc \*/

         movl          FUNC(vxIntStackPtr), %eax               /\* get int-stack ptr \*/

         leal   20(%esp), %ecx                                          /\* get addr of errno \*/

         movl          %ecx, ESF0\_NBYTES+12(%eax)         /\* save the original ESP \*/

         leal   12(%esp), %esi                              /\* set the source addr \*/

         movl          %eax, %edi                           /\* set the destination addr \*/

         movl          $ ESF0\_NLONGS+3, %ecx    /\* set number of longs to copy \*/

         cld                                                      /\* set direction ascending order \*/

         rep                                                      /\* repeat next inst \*/

         movsl                                                  /\* copy ESF0\_NLONGS + 3 longs \*/

         popl %edi                  /\* 任务栈恢复 %edi \*/

         popl %esi                   /\* 任务栈恢复 %esi \*/

         popl %ecx                  /\* 任务栈恢复 %ecx \*/

         movl          %eax, %esp       /\* 栈寄存器的切换 \*/

         /\* now, we are in the interrupt stack \*/

#endif       /\* INT\_STACK\_USE \*/

intEnt0:

         pushl         ESF0\_EFLAGS+12(%esp)  /\* push the saved EFLAGS \*/

         popfl                                     /\* UNLOCK INTERRUPT \*/

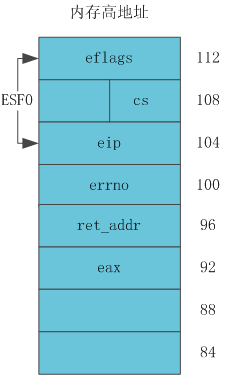
         popl %eax                           /\* restore %eax \*/

         ret

intEn()在逻辑上可以分成三部分：

**1.   在进行中断栈切换之前，特权态下的栈幁布局如下。**

代码执行到语句“#ifdef        INT\_STACK\_USE”之前的部分，其它任务栈布局如图4.5所示。



**图4.5 任务栈布局**

其中的返回地址是在“打桩”的代码中调用函数intEn()压入战中的返回地址。其中的eflags，cs，ip寄存器的值是由硬件自动压入栈；而errno，ret\_addr，eax则是由intEn()手动压入到任务栈中。此时仍处于任务栈中，还没有切换到中断栈。

**2.任务栈到中断栈的切换**

由

#ifdef        INT\_STACK\_USE

…..

#endif

中的代码实现。

首先判断是否开启中断栈；

在开启中断栈的情况下，判断是否是第一次进入中断栈，这个根据代码段选择子和中断段选择的比较来实现的。

在VxWorks的Pentium平台，Wind内核工作在保护模式下的特权态，其全局描述符表的值如下：

FUNC\_LABEL(sysGdt)

         /\* 0(selector=0x0000): Null descriptor \*/

         .word        0x0000

         .word        0x0000

         .byte         0x00

         .byte         0x00

         .byte         0x00

         .byte         0x00

         /\* 1(selector=0x0008): Code descriptor, for the supervisor mode task \*/

         .word        0xffff                            /\* limit: xffff \*/

         .word        0x0000                        /\* base : xxxx0000 \*/

         .byte         0x00                            /\* base : xx00xxxx \*/

         .byte         0x9a                            /\* Code e/r, Present, DPL0 \*/

         .byte         0xcf                    /\* limit: fxxxx, Page Gra, 32bit \*/

         .byte         0x00                            /\* base : 00xxxxxx \*/

         /\* 2(selector=0x0010): Data descriptor \*/

         .word        0xffff                            /\* limit: xffff \*/

         .word        0x0000                        /\* base : xxxx0000 \*/

         .byte         0x00                            /\* base : xx00xxxx \*/

         .byte         0x92                            /\* Data r/w, Present, DPL0 \*/

         .byte         0xcf                    /\* limit: fxxxx, Page Gra, 32bit \*/

         .byte         0x00                            /\* base : 00xxxxxx \*/

         /\* 3(selector=0x0018): Code descriptor, for the exception \*/

         .word        0xffff                            /\* limit: xffff \*/

         .word        0x0000                        /\* base : xxxx0000 \*/

         .byte         0x00                            /\* base : xx00xxxx \*/

         .byte         0x9a                            /\* Code e/r, Present, DPL0 \*/

         .byte         0xcf                    /\* limit: fxxxx, Page Gra, 32bit \*/

         .byte         0x00                            /\* base : 00xxxxxx \*/

         /\* 4(selector=0x0020): Code descriptor, for the interrupt \*/

         .word        0xffff                            /\* limit: xffff \*/

         .word        0x0000                        /\* base : xxxx0000 \*/

         .byte         0x00                            /\* base : xx00xxxx \*/

         .byte         0x9a                            /\* Code e/r, Present, DPL0 \*/

         .byte         0xcf                    /\* limit: fxxxx, Page Gra, 32bit \*/

         .byte         0x00                            /\* base : 00xxxxxx \*/

从全局描述符表我们可以看出，为了区分出任务代码段和中断代码段，特权设置了代码段描述符和中断代码段描述符。

并sysCsSuper指向特权任务段，sysCsInt指向特权中断段。

FUNC\_LABEL(sysCsSuper)

        .long   0x00000008              /\* CS for supervisor mode task \*/

FUNC\_LABEL(sysCsExc)

        .long   0x00000018              /\* CS for exception \*/

FUNC\_LABEL(sysCsInt)

        .long   0x00000020              /\* CS for interrupt \*/

如果是第一次进入中断，并且中断嵌套次数为1，则需要进行中断栈幁的切换。

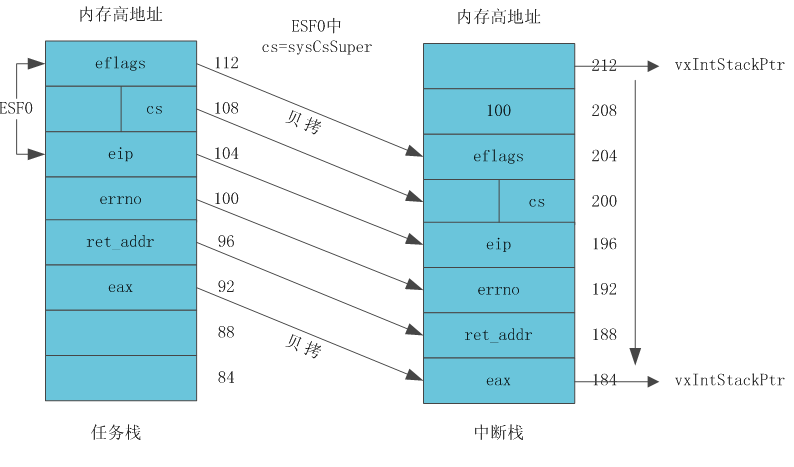
VxWorks的中断栈在usrInit()->kernelInit()函数中分配，大小为4K，其在VxWorks内存中的布局如图4.6所示.



**图4.6 中断栈VxWorks内存中的布局**

vxIntStackPtr通过usrInit()->kernelInit()->windIntStackSet()位置位执行vxIntStackBase

中断栈分配ESF0\_NBYTES+12+4=28字节的空间，然后将创建的任务栈幁拷贝到中断栈幁中，示意图如4.7。



**图4.7 中断不嵌套下的中断栈布局**

**3.intEnt()返回**

intEnt0:

         pushl         ESF0\_EFLAGS+12(%esp)  /\* push the saved EFLAGS \*/

         popfl                                     /\* UNLOCK INTERRUPT \*/

         popl %eax                           /\* restore %eax \*/

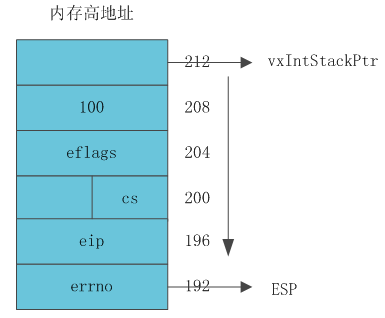
         ret

执行intEnt0指向的代码段，即恢复被中断之前的EFLAGS寄存器的值，由于原来的EFLAGS寄存中IF为没有置位，所有恢复之后，中断打开。

恢复eax寄存器的值；

恢复ret\_addr指向的返回地址。

此时中断栈的布局如图4.8所示。



**图4.8 中断恢复时的中断栈布局**

备注：这里分析的都是第一场中断，没有考虑中断嵌套的情况，如果考虑中断嵌套的话，只需要在上面中断栈的基础上再次保持上下文即可。

从上面代码的分区中我们指定intEnt()函数是在关中断条件下执行，其执行的目的是构造运行中断ISR的栈环境，当前被中断的任务或者上一次中断的上下文还没有保存。

接着执行中断ISR，对于时钟中断来说，最终执行的是：

void tickAnnounce (void)

{

    if (kernelState)                            /\* defer work if in kernel \*/

                  workQAdd0 ((FUNCPTR)windTickAnnounce);

    else

         {

                  kernelState = TRUE;                  /\* KERNEL\_ENT \*/

                  windTickAnnounce ();                /\* advance tick queue \*/

                  windExit ();                          /\* KERNEL\_EXIT \*/

         }

}

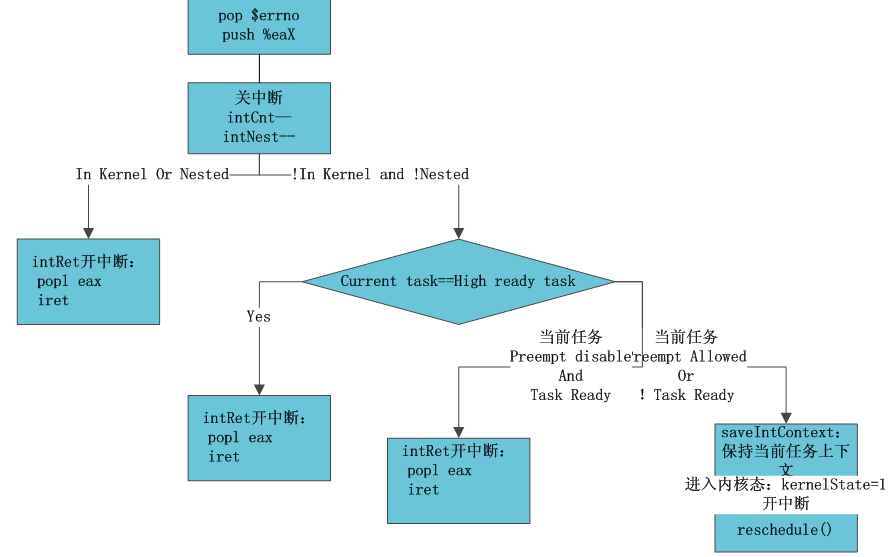
如果当前wind内核的内核态没有被占用，则进入内核态，执行中断处理函数windTickAnnounce();

如果存在代码正在访问内核，则将中断处理例程放入内核处理队列中延后执行。

VxWorks的“前后段”中断处理框架就体现在这里。

接着执行i8259IntEoiMaster(0)，向中断控制器写EOI(End of Interrupt)信号。告诉中断控制器中断已经处理完毕。

最后执行intExit(),intExit()的执行路程如4.9。



**图4.9 中断返回处理IntExit()处理流程**

intExit()检查调度队列，以决定是否需要执行调度器。如果没有更高优先级的任务就绪，并且没有内核队列可做，那么intExit()返回到被中断的任务。

如果调度是必须的，那么此时遍历中断栈，将被中断任务的上下文保存到被中断任务的TCB中。

由于intExit()需要保持所有的寄存器，所有当使用寄存器来检查内核队列是否为空时，使用的寄存器必须实现将其内容备份到栈。

另外在调用reschedule()时，edx必须保持的是taskIdCurrent。

当intExit()检测到当前被中断的任务已经不是优先级最高的任务，并且当前任务运行抢占。则保存当前任务的上下文，进入内核态，并开中断。从这一刻起，系统的运行状态进入内核级。在任何多任务系统中，大量的应用是发生在一个或多个任务的上下文中。然而，有些CPU时间片不在任何任务的上下文。这些时间片发生在内核改变内部队列或决定任务调度。在这些时间片中，CPU在内核级执行，而非任务级。

为了内核安全地操作它的内部的数据结构，必须有互斥操作。内核级没有相关的任务上下文，内核不能使用信号量保护内部链表。内核使用工作延期作为实现互斥的方式。当有内核参与时，中断服务程序调用的函数不是被直接激活，而是被放在内核的工作队列中。内核完成这些请求的执行而清空内核工作队列。

当内核正在执行已经被请求服务时系统将不响应到达内核的函数调用。可以简单地认为内核状态类似于禁止抢占。如前面所讨论的，抢占延时在实时系统中是不期望有的，因为它增加了对于会引起应用任务重新调度的事件的响应时间.

尽管操作系统在内核级（此时禁止抢占）完全避免消耗时间是不可能的，但减少这些时间是很重要的。这是减少由内核执行的函数的数量的主要原因，也是不采用统一结构的系统设计方式的原因。

言归正传，此时intExit()已经进入内核级执行，其进入内核态，开中断，执行调度器schedule().

此刻虽然当前任务（taskIdCurrent指向的任务）的上下文已经保存到当前任务的TCB控制块中。但是taskIdCurrent此时指向的仍然是这个任务。

当进入reschedule()调度器时，正常情况下它会恢复优先级最高的那个任务的上下文；如果到那个时候当前出现的优先级最高的任务由于各种原因变成非就绪态。有限性最高的任务就变为自身的话，它将会重新恢复运行。

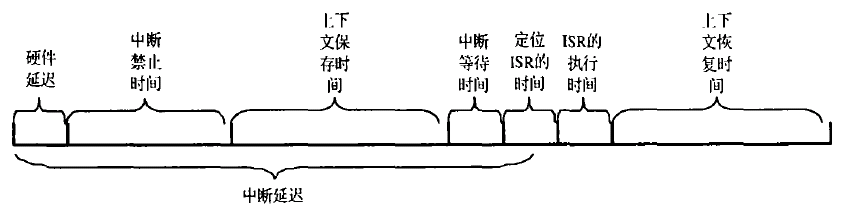
如果那时，自己也会阻塞了，就绪队列将会为空。reschedule()将会空转执行内核队列中的Job。

**4.4 小结**

VxWorks的“前后段”中断处理机制可以有效的对并发中断进行处理。“前半部”只完成必要的处理，在尽可能短的时间内记录已发生的中断，而把原来位于中断处理程序中的某些部分移到了任务中，在尽快完成必要的中断处理的基础上，保证了对并发中断的及时响应。此外，VxWorks的内核工作队列中的Job主要来自于中断(并非全部，因为看门狗定时处理例程也是放在内核Job队列中完成)，这种工作推迟技术将记录中断中要完成的工作，并在退出中断前或任务切换前执行所有内核Job并清空内核工作队列，避免中断的丢失。

VxWorks的内核中有两个堆栈：系统栈和任务栈。系统栈是系统为中断上下文处理而预留的堆栈；任务栈则属于任务本身的私有堆栈，用来存储任务执行过程中的临时变量等信息。当中断发生且非中断嵌套时，堆栈由被中断任务的任务栈切换到系统栈；当在中断处理中又发生中断即发生中断嵌套时，堆栈不再切换，仍用系统栈；当退出最外层中断时，堆栈又由系统栈切换到被中断任务的任务栈。由于中断上下文不存在于任何任务的上下文中，因此可以保证足够快的中断响应时间，有利于满足空间应用实时性强的要求。

通常有几个因素影响中断延迟：中断响应的硬件延迟、中断禁止时间、中断处理前保存

上下文的时间、中断等待时间以及定位ISR的时间，如图4.10所示。

**图4.10 中断延迟组成成分**

一次仅允许一个进程访问的资源称为临界资源，每个进程中访问临界资源的那段代码称临界区。中断禁止时间受临界区长度的直接影响，这是由于在执行临界区代码的时候通常需要关中断。Wind内核的中断“前后段”处理模型只在上下文保存和上下文恢复时关中断，并保存尽可能少的上下文，而在中断等待和定位ISR时可以响应中断，减少了中断延迟。中断等待时间由中断的调度策略决定，Wind内核是完全抢占式的内核，一个高优先级中断到来后立即打断低优先级中断的处理，内核转去处理高优先级的中断，此时中断等待时间为0。

VxWorks对中断嵌套的支持可以保证不丢失高优先级的中断，从而保证了实时系统的实时性和可靠性。

由于ISR不是在任务上下文中运行，它没有TCB，并且所有的中断处理程序共享同一个堆栈，因此它必须遵循一个基本约束：不能调用可能引起阻塞的函数。例如：在ISR中不能试图获取一个信号量，不能通过VxWorks驱动执行I/O操作，由于在intEnter()函数中没有保存浮点寄存器的操作，所以在ISR中也不能调用使用浮点协处理器的函数。

待续.....O(∩\_∩)O~