**->https://blog.csdn.net/redseazhaojianertao/article/details/53843515**

**VxWorks内核解读-6**

2016年12月23日 20:18:23 [redseazhaojianertao](https://me.csdn.net/redseazhaojianertao" \t "_blank) 阅读数：314

本篇文章分析VxWorks的初始化，VxWorks的初始化可以分成两个部分：

1.具体处理器平台相关的硬件初始化：包括CPU内部寄存器、堆栈寄存器的初始化，外设初始化；

2.VxWorks内核初始化：包括核心数据结构的初始化、初始任务的创建，启动多任务等等。

我以Pentium平台为例，来分析VxWorks的初始化过程。

**6.1 处理器平台相关的初始化**

这部分代码初始化CPU内部寄存器，是VxWorks在内存中的入口代码。其主要工作是关中断，初始化CPU内部寄存器，特别是栈寄存器，分配栈空间。为运行第一个C函数usrInit()建立环境。

具体代码如下：

sysInit:

\_sysInit:

         cli                                                                                /\* 关中断 \*/

         movl    $ BOOT\_WARM\_AUTOBOOT,%ebx        /\*设置启动类型 \*/

         movl          $ FUNC(sysInit),%esp                         /\* 初始化栈寄存器 \*/

         movl          $0,%ebp                                               /\* 初始化栈幁寄存器\*/

         ARCH\_REGS\_INIT                      /\*初始化DR[0-7] ,CR0, EFLAGS寄存器 \*/

#if     (CPU == PENTIUM) || (CPU == PENTIUM2) || (CPU == PENTIUM3) || \

         (CPU == PENTIUM4)

         /\* ARCH\_CR4\_INIT          /@ initialize CR4 for P5,6,7 \*/

         xorl  %eax, %eax               /\* 清EAX寄存器 \*/

         movl          %eax, %cr4       /\* 清CR4寄存器 \*/

#endif       /\* (CPU == PENTIUM) || (CPU == PENTIUM[234]) \*/

/\*将全局描述符表拷贝到pSysGdt指向的内存空间处\*/

         movl          $ FUNC(sysGdt),%esi       /\* set src addr (&sysGdt) \*/

         movl          FUNC(pSysGdt),%edi        /\* set dst addr (pSysGdt) \*/

         movl          %edi,%eax

         movl          $ GDT\_ENTRIES,%ecx      /\* number of GDT entries \*/

         movl          %ecx,%edx

         shll   $1,%ecx                      /\* set (nLongs of GDT) to copy \*/

         cld

         rep

         movsl                                    /\* copy GDT from src to dst \*/

/\*构造初始化gdtr寄存器的值\*/

         pushl         %eax                           /\* push the (GDT base addr) \*/

         shll   $3,%edx                              /\* get (nBytes of GDT) \*/

         decl  %edx                                    /\* get (nBytes of GDT) - 1 \*/

         shll   $16,%edx                                      /\* move it to the upper 16 \*/

         pushl         %edx                           /\* push the nBytes of GDT - 1 \*/

         leal   2(%esp),%eax                    /\* get the addr of (size:addr) \*/

         pushl         %eax                           /\* push it as a parameter \*/

         call   FUNC(sysLoadGdt)  /\* load the brand new GDT in RAM \*/

/\*构造一个中断返回的情景\*/

         pushl         %ebx                           /\* push the startType \*/

         movl          $ FUNC(usrInit),%eax

         movl          $ FUNC(sysInit),%edx      /\* push return address \*/

         pushl         %edx                           /\*   for emulation for call \*/

         pushl         $0                       /\* push EFLAGS, 0 \*/

         pushl         $0x0008                     /\* a selector 0x08 is 2nd one \*/

         pushl         %eax                           /\* push EIP,  FUNC(usrInit) \*/

         iret                               /\* iret \*/

代码分析：

1.       sysInit()初始化过程比较直观，但是由于这是一段汇编语句，需要考虑到汇编语言和C语言编程的一些细节。

BOOT\_WARM\_AUTOBOOT是一个宏，其值为0，将一个宏的值放入一个寄存器中时，采用的语句是：

movl    $ BOOT\_WARM\_AUTOBOOT,%ebx

sysInit()是一个函数名字，其所在的地址为sysInit()的入口地址0x30800c：  
0030800c <\_sysInit>:

  30800c:        fa                           cli

  30800d:        bb 00 00 00 00          mov    $0x0,%ebx

  308012:        bc 0c 80 30 00           mov    $0x30800c,%esp

  308017:        bd 00 00 00 00          mov    $0x0,%ebp

  30801c:        31 c0                         xor    %eax,%eax

  30801e:        0f 23 f8               mov    %eax,%db7

  308021:        0f 23 f0               mov    %eax,%db6

  <……………….略…………………>

所以

movl          $ FUNC(sysInit),%esp就是将sysInit所在的地址0x30800放入到寄存器ESP中。

由于：

#define FUNC(sym)           sym

#define FUNC\_LABEL(sym)               sym:

movl          $ FUNC(sysInit),%esp和movl         $ sysInit,%esp是一致的。

由于sysInit是VxWorks的入口地址，把地址赋值给ESP，意味着将sysInit地址往下的地方作为临时栈空间。

2. ARCH\_REGS\_INIT宏分析

ARCH\_REGS\_INIT宏展开如下：

#define ARCH\_REGS\_INIT                                                               \

         xorl  %eax, %eax;              /\* zero EAX \*/                    \

         movl          %eax, %dr7;              /\* initialize DR7 \*/            \

         movl          %eax, %dr6;              /\* initialize DR6 \*/            \

         movl          %eax, %dr3;              /\* initialize DR3 \*/            \

         movl          %eax, %dr2;              /\* initialize DR2 \*/            \

         movl          %eax, %dr1;              /\* initialize DR1 \*/            \

         movl          %eax, %dr0;              /\* initialize DR0 \*/            \

         movl    %cr0, %edx;               /\* get CR0 \*/                      \

         andl    $0x7ffafff1, %edx;     /\* clear PG, AM, WP, TS, EM, MP \*/ \

         movl    %edx, %cr0;               /\* set CR0 \*/                      \

                                                                                    \

         pushl         %eax;                          /\* initialize EFLAGS \*/               \

         popfl;

其用于初始化Pentium平台的调试寄存器，控制寄存器CRO，以及EFLAGS寄存器。

从控制寄存器CRO只保留的PE位，我们可以看出目前Pentium只启用了保护模式。

关键CR0寄存器更详细的解释参考Intel官方编程手册。

3.将全局描述符表拷贝到pSysGdt指定的位置处

全局描述符表sysGdt[]定义如下：

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 \*/

代码中：

movl          $ FUNC(sysGdt),%esi是将sysGdt[]数组的首地址(即全局描述符表sysGdt[]所在内存块的基地址)放入到寄存器esi中，比如sysGdt[]数组所在的地址是0x30380，该条指令将0x30380放入esi寄存器中。

movl          FUNC(pSysGdt),%edi将pSysGdt的值放入到寄存器edi中，这里需要注意的是pSysGdt是一个指针变量，在sysLib.c中定义如下：

GDT \*pSysGdt = (GDT \*)(LOCAL\_MEM\_LOCAL\_ADRS + GDT\_BASE\_OFFSET);

其中

#define LOCAL\_MEM\_LOCAL\_ADRS (0x00100000)

#define GDT\_BASE\_OFFSET         0x1000

所有指针变量pSysGdt的值为0x101000，加载pSysGdt所在的地址为0x339980：

00339980 <pSysGdt>:

339980:        00 10                        add    %dl,(%eax)

339982:        10 00                        adc    %al,(%eax)

那么movl         FUNC(pSysGdt),%edi指令值得效果是将0x101000的值放入edi寄存器中，如果误写成$movl     FUNC(pSysGdt),%edi，将导致将0x339980写入edi寄存器中，从而引发错误。

4.通过构造中断栈幁实现跳转

sysInit()函数的最后，通过中断返回指令iret，实现跳转到第一个C函数usrInit()中，跳转之前sysInit()已经初始化了CPU的栈寄存器ESP为sysInit的入口地址，这意味着将sysInit入口地址向下的地址空间作为usrInit()函数的临时站空间。

要想成功跳转到iret函数中，必须构造中断栈幁：

         pushl         %ebx                           /\* push the startType \*/

         movl          $ FUNC(usrInit),%eax

         movl          $ FUNC(sysInit),%edx      /\* push return address \*/

         pushl         %edx                           /\*   for emulation for call \*/

         pushl         $0                       /\* push EFLAGS, 0 \*/

         pushl         $0x0008                     /\* a selector 0x08 is 2nd one \*/

         pushl         %eax                           /\* push EIP,  FUNC(usrInit) \*/

构造的伪中断栈幁如图6.1所示。

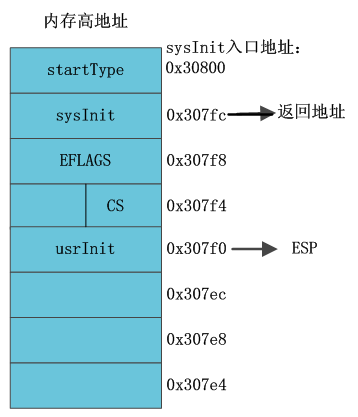


图6.1 临时中断栈帧

当执行完iret指令后，将跳转到usrInit()函数中运行。

**6.2 第一个C函数usrInit()执行**

usrInit()是VxWorks启动之后执行的第一个C函数，由于在跳转到usrInit()函数之前，sysInit()已经进行了关中断操作，因此该函数是在关中断条件下，使用sysInit建立的临时栈空间执行相关硬件的初始化。

其主要完成的工作如下：

1. 清BSS段，将vxWorks内核映像中所有为初始化的全局变量初始化为0；
2. 建立异常向量表；
3. 调用sysHwInit()初始化硬件，这里的sysHwInit()函数是vxWorks的板级支持包BSP的主调用函数；
4. 创建初始化任务taskRoot，由taskRoot任务的主函数usrRoot继续完成vxWorks核心的初始化。

usrInit()的实现跟用户的配置相关，这里我们不考虑Cache的使用，由于我们侧重分析的VxWorks内核的初始化过程，cache的配置和工作机制不是我们研究的重点。

usrInit()实现代码如下：

void usrInit (int startType)

{

    sysStart (startType); /\* 清BSS段，同时设置中断向量表的基地址\*/

    excVecInit ();      /\*构建异常向量表 \*/

    sysHwInit ();       /\*板级支持包BSP的入口函数，vxWorks的设备驱动在这里调用\*/

    usrKernelInit ();   /\* 构造初始化任务taskRoot的上下文，启动taskRoot \*/

}

分析：

1. sysStart (startType)主要完成的工作是清BSS段、设置启动类型，并初始化CPU的中断向量表基地址寄存器。
2. excVecInit()完成初始化构架异常向量表，并用构架的异常向量表的基地址初始化CPU的异常向量基地址寄存器；
3. sysHwInit ()是vxWorks板级支持包BSP的入口完成，用于完成BSP定制的外设的初始化，主要包含以下几个部分：

* 初始化中断控制器和挂接中断的例程，比如Pentium平台：
  + sysIntInitPIC ();                  /\*初始化可编程中断控制器 \*/
  + intEoiGet = sysIntEoiGet;         /\* 用于中断挂接的intConnect()的调用例程 \*/
* 遍历PCI总线，初始化总线上的网络设备；
  + pciConfigForeachFunc (0, TRUE, (PCI\_FOREACH\_FUNC) sysNetPciInit, NULL);
* 遍历PCI总线，寻找USB设备，并添加USB设备映射空间
* 初始化串口设备
* 初始化电源管理设备
* 初始化硬盘设备

usrKernelInit()配置内核数据结构，并调用kernelInit()构造初始化任务taskRoot的上下文，启动taskRoot任务。我们单独分析usrKernelInit()函数。

**6.3 usrKernelInit()函数分析**

usrKernelInit()配置内核数据结构，调用kernelInit()构造初始化任务taskRoot的上下文，启动taskRoot任务,其具体代码实现如下：

void usrKernelInit (void)

{

    classLibInit ();                     /\* initialize class (must be first) \*/

    taskLibInit ();                      /\* initialize task object \*/

    /\* 配置内核就绪队列、活动队列、定时队列 \*/

#ifdef        INCLUDE\_CONSTANT\_RDY\_Q

    qInit (&readyQHead, Q\_PRI\_BMAP, (int)&readyQBMap, 256); /\* 固定优先级队列 \*/

#else

    qInit (&readyQHead, Q\_PRI\_LIST); /\* 简单优先级队列 \*/

#endif       /\* !INCLUDE\_CONSTANT\_RDY\_Q \*/

    qInit (&activeQHead, Q\_FIFO);       /\* 先进先出的活动队列 \*/

    qInit (&tickQHead, Q\_PRI\_LIST);   /\* 简单优先级队列\*/

    workQInit ();                       /\* 内核延时工作队列 \*/

    /\*构架初始化任务taskRoot()上下文，启动taskRoot任务，其主流程为usrRoot \*/

    kernelInit ((FUNCPTR) usrRoot, ROOT\_STACK\_SIZE, MEM\_POOL\_START,

                sysMemTop (), ISR\_STACK\_SIZE, INT\_LOCK\_LEVEL);

}

分析：

在VxWorks中：

就绪队列由全局变量readyQHead指向其头部，该队列中链接的是有资格获取CPU使用权的任务；

定时队列由全局变量readyQHead指向其头部，该队列链接的是所有需要延时的任务；

活动队列由全局变量activeQHead指向其头部，该队列链接的是内核中创建的所有任务，包括就绪队列中的任务、定时队列中需要延时的任务、以及在信号量等待队列中的任务。

内核延时队列是一个大小为64的环形队列；

这四个队列构成了vxWorks内核最核心的资源。位于wind内核的内核态中，由内核全局变量kernelState进行保护。只有在windLib库中的内核态例程wind\*开头的例程才可以访问。非内核态的例程只有进入内核态，才能调用wind\*例程，访问并操作这三个内核队列、以及各种信号量等待队列。

下面我们依次分析者四种队列：

**6.3.1 就绪队列**

在VxWorks的wind内核中就绪队列可以由两种配置方式：

1.  按照优先级的从高低排序，形成一个优先级队列：

qInit (&readyQHead, Q\_PRI\_LIST); /\* 简单优先级队列 \*/

这样的队列虽然比较简单。但是当存在任务就绪时，插入队列的时间跟优先级队列的长度相关，假如优先级队列的长度为n。则插入优先级队列的时间复杂度为O(n)。

2.  另外一种方式是采用才优先级位图形式的优先级队列。这样的话，优先级队列的入队时间只有优先级数相关，而与优先级队列的长度无关，插入优先级队列的时间复杂度为O(1)。

具体的机制如下：

readyQHead类型：

typedef struct           /\* Q\_HEAD \*/

    {

    Q\_NODE  \*pFirstNode;          /\* first node in queue based on key \*/

    UINT     qPriv1;                      /\* use is queue type dependent \*/

    UINT     qPriv2;                      /\* use is queue type dependent \*/

    Q\_CLASS \*pQClass;                   /\* pointer to queue class \*/

    } Q\_HEAD;

Q\_NODE是16个字节的类型：

typedef struct           /\* Q\_NODE \*/

    {

    UINT     qPriv1;                      /\* use is queue type dependent \*/

    UINT     qPriv2;                      /\* use is queue type dependent \*/

    UINT     qPriv3;                      /\* use is queue type dependent \*/

    UINT     qPriv4;                      /\* use is queue type dependent \*/

    } Q\_NODE;

在readyQHead.pFirstNode指向的就绪队列中，每个节点代表一个WIND\_TCB控制块，所以WIND\_TCB控制块必须有一个成员为Q\_NODE类型，

typedef struct windTcb             /\* WIND\_TCB - task control block \*/

    {

    Q\_NODE           qNode;              /\* 0x00: multiway q node: rdy/pend q \*/

    Q\_NODE           tickNode;       /\* 0x10: multiway q node: tick q \*/

    Q\_NODE           activeNode;     /\* 0x20: multiway q node: active q \*/

    OBJ\_CORE                 objCore;   /\* 0x30: object management \*/

    …………….<略>……………

    } WIND\_TCB;

readQHead头节点在

usrKernelInit()->qInit (&readyQHead, &qPriBMapClass, (int)&readyQBMap, 256)中初始化

将readQHead. pQClass初始化为& qPriBMapClass.

这样就可以通过readQHead. pQClass调用rqPriBMapClass .qPriBMapInit()初始化readyQHead.

通过qPriBMapInit()申明部分：

STATUS qPriBMapInit

    (

    Q\_PRI\_BMAP\_HEAD \*    pQPriBMapHead,

    BMAP\_LIST \*              pBMapList,

    UINT                          nPriority           /\* 1 priority to 256 priorities \*/

)

其中：

typedef struct           /\* Q\_PRI\_BMAP\_HEAD \*/

    {

    Q\_PRI\_NODE   \*highNode;                /\* highest priority node \*/

    BMAP\_LIST      \*pBMapList;             /\* pointer to mapped list \*/

    UINT         nPriority;                      /\* priorities in queue (1,256) \*/

} Q\_PRI\_BMAP\_HEAD;

typedef struct           /\* Q\_PRI\_NODE \*/

    {

    DL\_NODE         node;               /\* 0: priority doubly linked node \*/

    ULONG     key;               /\* 8: insertion key (ie. priority) \*/

    } Q\_PRI\_NODE;

typedef struct dlnode                /\* Node of a linked list. \*/

    {

    struct dlnode \*next;         /\* Points at the next node in the list \*/

    struct dlnode \*previous; /\* Points at the previous node in the list \*/

    } DL\_NODE;

typedef struct           /\* BMAP\_LIST \*/

    {

    UINT32    metaBMap;               /\* lookup table for map \*/

    UINT8       bMap [32];                 /\* lookup table for listArray \*/

    DL\_LIST    listArray [256];            /\* doubly linked list head \*/

    } BMAP\_LIST;

typedef struct                    /\* Header for a linked list. \*/

    {

    DL\_NODE \*head;     /\* header of list \*/

    DL\_NODE \*tail;        /\* tail of list \*/

    } DL\_LIST;

readyQHead类型将由Q\_HEAD类型强制装换为Q\_PRI\_BMAP\_HEAD类型：

这样readyQHead. qPriv1将会初始为(int)&readyQBMap，eadQHead. qPriv2被初始化为类255.

readyQHead.pFirstNode被初始化为NULL。

初始化之后的示意图状态如图6.2所示。

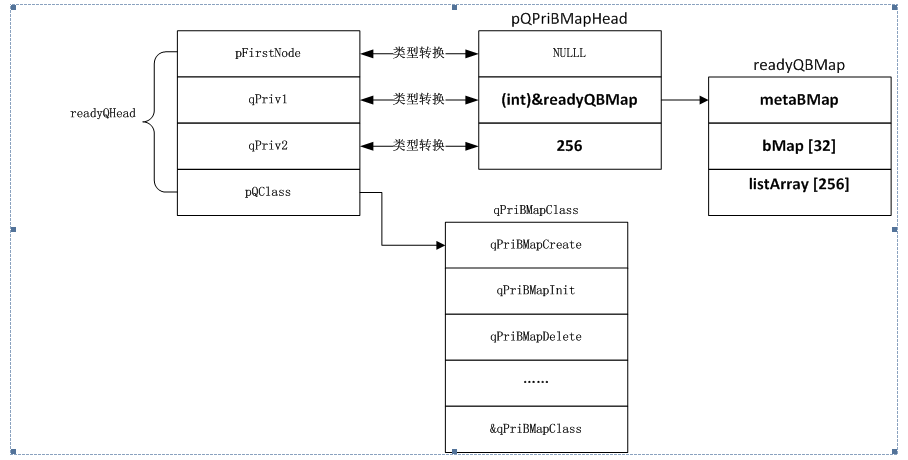


图6.2 就绪队列状态示意图

备注：从图中我们可以看出readyQHead.pFirstNode成员是Q\_NODE类型的指针变量(Q\_NODE类型占据16个字节)，而pQPriBMapHead. highNode成员是Q\_PRI\_NODE类型的指针变量。

这意味着什么呢？

我们可以这样理解，readyQHead.pFirstNode原来是指向16个字节内存区域的指针，经过强制类型装换后，编程了指向12个字节内存区域的指针。

typedef struct                   /\* Q\_PRI\_NODE \*/

    {

    DL\_NODE         node;                /\* 0: priority doubly linked node \*/

    ULONG    key;                  /\* 8: insertion key (ie. priority) \*/

    } Q\_PRI\_NODE;

typedef struct dlnode                /\* Node of a linked list. \*/

    {

    struct dlnode \*next;         /\* Points at the next node in the list \*/

    struct dlnode \*previous; /\* Points at the previous node in the list \*/

    } DL\_NODE;

备注：从Q\_PRI\_NODE的类型我们可以看出，当处理任务的代理人WIND\_TCB是将IWND\_TCB中的Q\_NODE类型的成员变量转换为Q\_PRI\_NODE，这意味着下面图6.3所示映射关系。

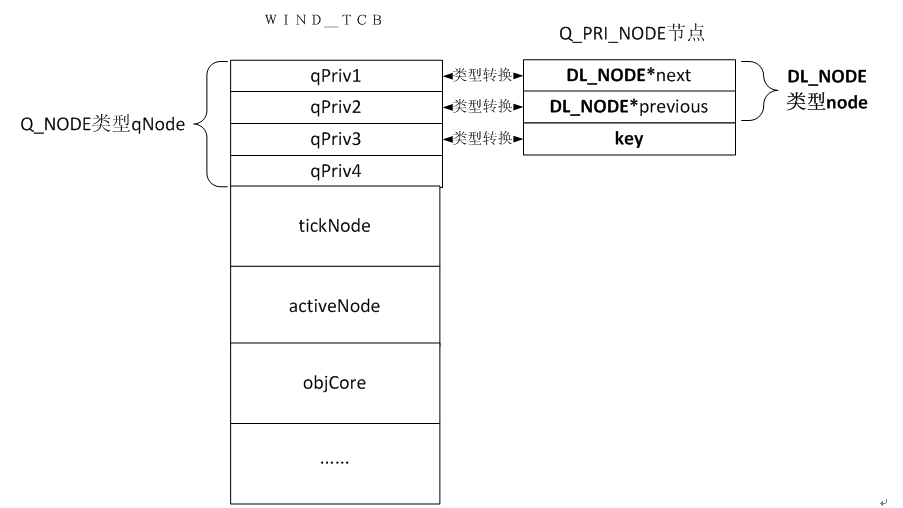


图6.3 Q\_NODE映射关系

从图中，我们可以看出，wind内核将WIND\_TCB中的qNode域转换成Q\_PRI\_NODE节点，放到优先级队列中进行处理。由于qNode节点是WIND\_TCB的第一个成员，该变量的首地址就是相应任务的WIND\_TCB地址，却优先级队列中的Q\_PRI\_NODE需要转化为TCB节点时，只需要做类型转换即可。比如：

taskIdCurrent = (WIND\_TCB \*) Q\_FIRST (&readyQHead)

其中Q\_FIRST宏类型如下：

#define Q\_FIRST(pQHead)                                                               \

((Q\_NODE \*)(((Q\_HEAD \*)(pQHead))->pFirstNode))

这样一切就清楚了。

vxWorks使用基于BIT位图的优先级队列，使用位图(bitmap)和元位图(meta-bitmap)、每个优先级对应一个FIFO队列，这种设计方案可以快速获取的Q\_GET()、Q\_PUT()操作方法，即Q\_GET()、Q\_PUT()操作的时间复杂度为0(1)。

其具体优先级位图状态如图6.4所示。

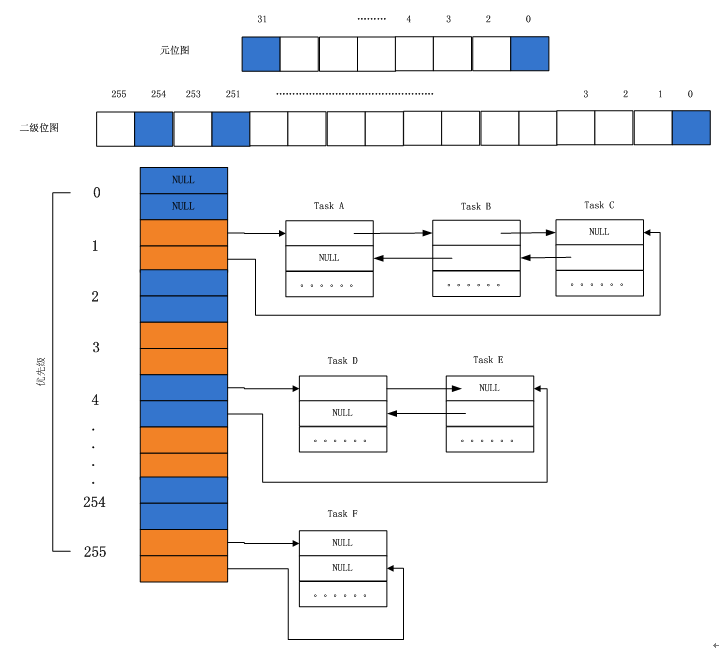


图6.4 优先级位图状态

备注：Task A，Task B, Task C的优先级为1，以对应的元位图的Bit31，二级位图Bit254.

例如当向位图队列中放入Task C时，是放入优先级为1处的FIFO队列的尾部。调整元位图和二级位图的C代码片段如下：

此时priority=1；

priority = 255 - priority;

pBMapList->metaBMap                    |= (1 << (priority >> 3));

pBMapList->bMap [priority >> 3]    |= (1 << (priority & 0x7));

删除位图队列中的TASK F时，调度位图的C代码片段如下：

此时priority=255；

priority = 255 - priority;

pBMapList->bMap [priority >> 3] &= ~(1 << (priority & 0x7));

if (pBMapList->bMap [priority >> 3] == 0)

pBMapList->metaBMap &= ~(1 << (priority >> 3));

此时优先级位图队列的状态如图6.5所示。

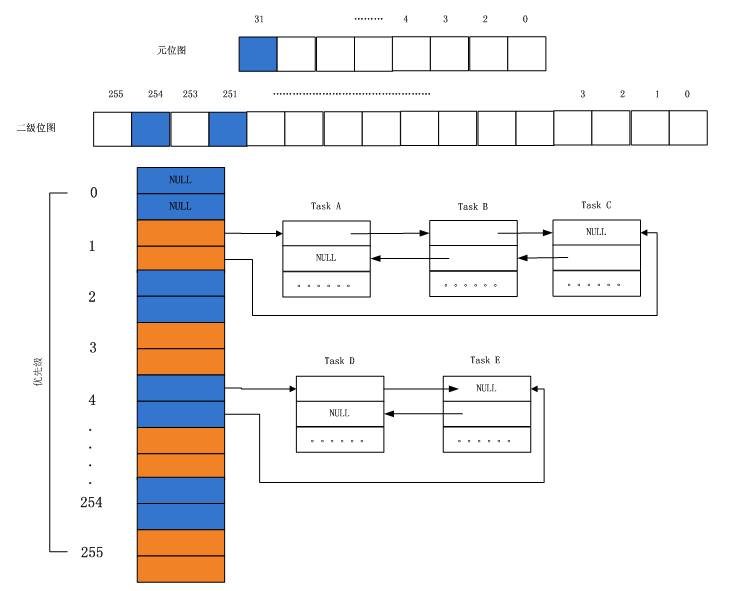


图6.5 优先级位图队列状态

备注：注意元位图中的Bit0位，二级位图的中的Bit255位已经清0,255优先级对应的Task F任务已经从优先级位图队列中清除。

注意：这里需要指出的是元位图、以及二级位图中是以MSB Bit位来索引最高优先级的，这与我们在uC/OS-II中使用的以LSB Bit位来索引最高优先级的方式刚好相反。

**6.3.2 定时队列设计**

定时队列基于全局变量32位的无符号整数vxTicks，来判断定时器队列中的节点(每个节点代表一个WIND\_TCB控制块)的定时时间是否到达。

定时队列在usrKernelInit()函数中北初始化：

qInit (&tickQHead, &qPriListClass);       /\* simple priority semaphore q\*/

tickQHead也是Q\_HEAD类型：

typedef struct           /\* Q\_HEAD \*/

{

Q\_NODE  \*pFirstNode;          /\* first node in queue based on key \*/

UINT     qPriv1;                      /\* use is queue type dependent \*/

UINT     qPriv2;                      /\* use is queue type dependent \*/

Q\_CLASS \*pQClass;                   /\* pointer to queue class \*/

} Q\_HEAD;

qInit()将tickQHead初始化为&qPriListClass，然后利用qPriListInit()初始化tickQHead的其余三个成员变量。

STATUS qPriListInit

(

Q\_PRI\_HEAD \*pQPriHead

)

{

dllInit (pQPriHead); /\* initialize doubly linked list \*/

return (OK);

}

通过qPriListInit()函数的类型，我们可以看出，tickQHead将会被转化为Q\_PRI\_HEAD类型：

typedef DL\_LIST Q\_PRI\_HEAD;

typedef struct                    /\* Header for a linked list. \*/

{

DL\_NODE \*head;     /\* header of list \*/

DL\_NODE \*tail;        /\* tail of list \*/

} DL\_LIST;

其初始化后的定时器队列，在挂入了两个延时任务后的示意如图6.6所示。

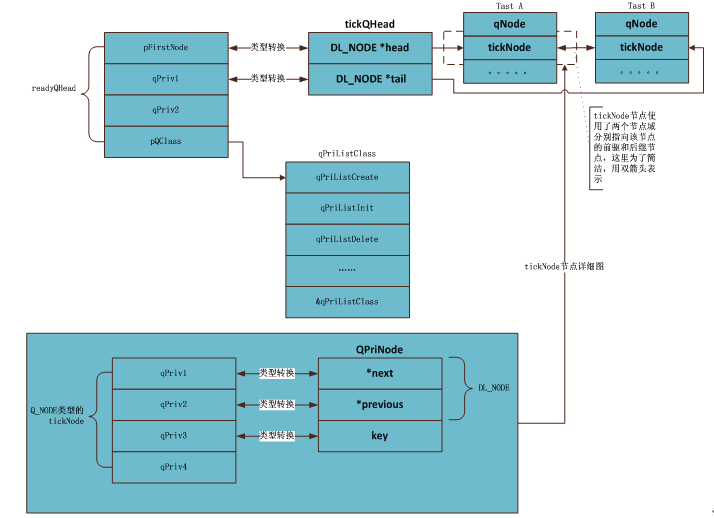


图6.6 定时器队列示意图

备注：WIND\_TCB块的Q\_NODE域的四个成员，目前只是用了三个，没有用的是第四个成员域，定时器队列采用根据定时到期的时刻(该时间存放在qPriv3成员域中，也即key变量的值)的长短排序，到期时刻小的节点排在前面。

tickQHead指向的定时队列中，tickQHead中有两个域pFirstNode，qPriv1分别之前定时队列的头部和尾部。

定时队列的节点QPriNode的两个域在定时队列的第一个节点和最后一个节点，具有一个节点域为NULL。

即第一个节点previous为NULL，最后一个节点next为NULL

我们来分析一下入队操作：当一个任务需要延时时，将通过taskDelay()->windDelay()执行：

Q\_PUT (&tickQHead, &taskIdCurrent->tickNode, timeout + vxTicks)实现。

其中vxTicks存放的是当前滴答数，timeout表现要定时的时长，那么timeout + vxTicks表示的是闹钟闹铃的时刻(这里以时钟滴答作为刻度数)，Q\_PUT()是一个操作宏，即最终调用：

qPriListPut(&tickQHead, &taskIdCurrent->tickNode, timeout + vxTicks)。

由于定时器是按照定时时刻从前往后排序qPriListPut会将这个新的节点放置到第一个小于其时刻值的节点前面。

加入当前的定时队列的排序是：1,3,5,7,7,9

那么新来的6节点插入后的队列是：1,3,5,6,7,7,9

那么新来的7节点插入后的队列是：1,3,5,6,7,7,7,9

备注：如果插入的节点的定时刻和队列中已有节点的定时时刻相同，那么将其插入到相同定时时刻的节点后面。

为方便阅读，我贴出插入代码：

void qPriListPut

    (

    Q\_PRI\_HEAD  \*pQPriHead,

    Q\_PRI\_NODE  \*pQPriNode,

    ULONG        key

    )

    {

    FAST Q\_PRI\_NODE \*pQNode = (Q\_PRI\_NODE \*) DLL\_FIRST (pQPriHead);

    pQPriNode->key = key;

    while (pQNode != NULL)

        {

         if (key < pQNode->key)              /\* it will be last of same priority \*/

             {

             dllInsert (pQPriHead, DLL\_PREVIOUS (&pQNode->node),

                          &pQPriNode->node);

             return;

             }

         pQNode = (Q\_PRI\_NODE \*) DLL\_NEXT (&pQNode->node);

         }

    dllInsert (pQPriHead, (DL\_NODE \*) DLL\_LAST (pQPriHead), &pQPriNode->node);

    }

备注：由此看出将一个延时的任务插入定时队列的时间复杂度（这里指的是最坏时间复杂度）是跟延时队列的长度相关的，即时间复杂度为0(n)。为了保证RTOS的确定性，该插入操作在VxWorks后续版本(比如VxWorks6.8版本)中采用多级差分队列的算法，Linux-2.4之后的内核，uC/OS-III也采用了类似的算法。

出队操作比较简单,在VxWorks的时钟中断处理函数usrClock()->tickAnnounce()->windTickAnnounce()检查是否有任务的定时时间到，如果到的话，将会从定时队列中剔除，相关代码片段如下：

while ((pNode = (Q\_NODE \*) Q\_GET\_EXPIRED (&tickQHead)) != NULL)

{

pTcb = (WIND\_TCB \*) ((int)pNode - OFFSET (WIND\_TCB, tickNode));

。。。。。。。。。。。。。。。。。。。。。

}

Q\_GET\_EXPIRED (&tickQHead)即调用：qPriListGetExpired(&tickQHead)

该函数返回定义检查tickQHead队列的第一个节点是否定时时间到，如果到的话，返回第一个节点的地址，同时将第一个节点从定时队列中删除，让第二个节点成为顶一个节点。

Q\_PRI\_NODE \*qPriListGetExpired

    (

    Q\_PRI\_HEAD \*pQPriHead

    )

    {

    FAST Q\_PRI\_NODE \*pQPriNode = (Q\_PRI\_NODE \*) DLL\_FIRST (pQPriHead);

    if ((pQPriNode != NULL) && (pQPriNode->key <= vxTicks))

         return ((Q\_PRI\_NODE \*) dllGet (pQPriHead));//删除第一个节点，让其后续成为队列头部

    else

         return (NULL);

    }

**5.3.3 活动队列**

活动队列链接了vxWorks内核中所有已经创建的任务，不论其是否为就绪态，都会在链入该队列中。vxWorks内核的提高的系统调用i()、以及shell中的i命令，均是遍历该活动队列来显示系统中的所有创建的任务。

在usrKernelInit()被初始化：

qInit (&activeQHead, &qFifoClass);       /\* FIFO queue for active q \*/

activeQHead类型：

typedef struct           /\* Q\_HEAD \*/

{

Q\_NODE  \*pFirstNode;          /\* first node in queue based on key \*/

UINT     qPriv1;                      /\* use is queue type dependent \*/

UINT     qPriv2;                      /\* use is queue type dependent \*/

Q\_CLASS \*pQClass;                   /\* pointer to queue class \*/

} Q\_HEAD;

qInit ()将activeQHead. pQClass初始化为&qFifoClass，进而调用qFifoInit()初始化activeQHead的前两个域：

STATUS qFifoInit

(

Q\_FIFO\_HEAD \*pQFifoHead

)

{

dllInit (pQFifoHead);

return (OK);

}

pQFifoHead类型：

typedef DL\_LIST Q\_FIFO\_HEAD;               /\* Q\_FIFO\_HEAD \*/

typedef DL\_NODE Q\_FIFO\_NODE;           /\* Q\_FIFO\_NODE \*/

typedef struct dlnode                /\* Node of a linked list. \*/

{

struct dlnode \*next;         /\* Points at the next node in the list \*/

struct dlnode \*previous; /\* Points at the previous node in the list \*/

} DL\_NODE;

typedef struct                    /\* Header for a linked list. \*/

{

DL\_NODE \*head;     /\* header of list \*/

DL\_NODE \*tail;        /\* tail of list \*/

} DL\_LIST;

其初始化后，加入了两个任务的队列如图6.7所示。

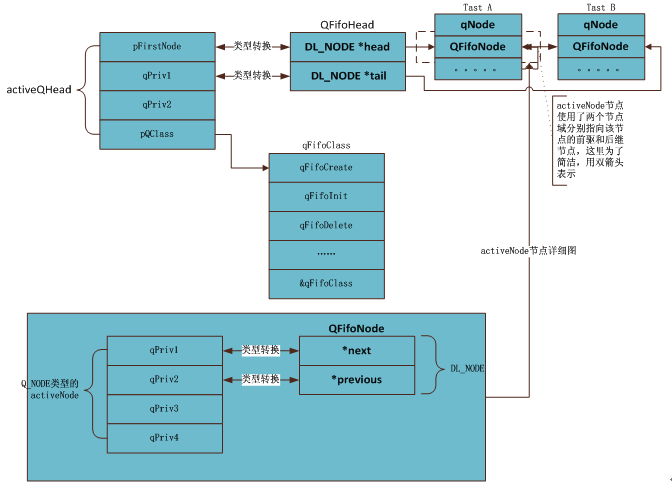


图6.7 活动队列示意图

从图中，我们可以看出活动队列比较简单。由于其是双向队列，可以将其插入到指定节点的任何位置。

例如当创建任务时：

taskSpawn()->taskCreate()->taskInit()->windSpawn()将新创建的任务掺入到活动队列的尾部，代码片段如下：

Q\_PUT (&activeQHead, &pTcb->activeNode, FIFO\_KEY\_TAIL);      /\* in active q\*/

Q\_PUT()是一个宏，进而调用qFifoPut (&activeQHead, &pTcb->activeNode, FIFO\_KEY\_TAIL)

void qFifoPut

    (

    Q\_FIFO\_HEAD \*pQFifoHead,

    Q\_FIFO\_NODE \*pQFifoNode,

    ULONG        key

    )

    {

    if (key == FIFO\_KEY\_HEAD)

         dllInsert (pQFifoHead, (DL\_NODE \*)NULL, pQFifoNode);

    else

         dllAdd (pQFifoHead, pQFifoNode);

    }

将指定的任务从活动队列中删除：

taskDelete()->taskDestroy()->windDelete()

或者taskTerminate()->taskDestroy()->windDelete()

windDelete()中的关键代码如下：

Q\_REMOVE (&activeQHead, &pTcb->activeNode);                  /\* deactivate it \*/

进而调用：qFifoRemove()

STATUS qFifoRemove(&activeQHead, &pTcb->activeNode);           /\* deactivate it \*/

    (

    Q\_FIFO\_HEAD \*pQFifoHead,

    Q\_FIFO\_NODE \*pQFifoNode

    )

    {

    dllRemove (pQFifoHead, pQFifoNode);

    return (OK);

    }

**6.3.4 内核延时队列**

由于wind内核态正在被其它程序访问，当前新的请求内核态例程服务的Job将被放置到内核队列中延时处理。内核工作队列是一个单读者/多写者的环形工作队列。读者总是第一个进入内核态的任务或者中断ISR，读者负责在离开wind内核前清空内核队列(通过执行内核Job)。由于内核写者主要来自于中断ISR(，还有一部分来自于任务)，因此在写操作内核队列期间，CPU必须关中断；但是在读操作期间不需要关中断。

内核队列通过一个大小为1K字节的环形缓冲队列实现，队列中的每一个元素称为Job，占16个字节大小，环形缓冲队列一共有64个Job。选择64个字节大小，是想利用刚好一个字节的数据的索引值可以遍历这个队列。这是因为每遍历一个元素，索引值都需要加4，如果用8个bit位(刚好一个字节大小)的索引值，其回卷到数值0时，刚对内核队列从头开始。不用单独考虑内核队列是否回卷，省去了条件判断的时间。

备注：有两个方面的局限，可能导致未来的wind内核版本中修改内核队列，这是因为64个大小的内核队列，每个队列16个字节是硬编码的，这很有可能不能适应未来的需求，但是就目前来说，这个规模是最有效的机制。

workQInit()完成内核队列的初始化，并将读写索引初始化为0，其代码如下：

void workQInit (void)

    {

    workQReadIx  = workQWriteIx = 0;       /\* initialize the indexes \*/

    workQIsEmpty = TRUE;            /\* the work queue is empty \*/

    }

workQAdd0()添加无参数的Job到内核队列中，当内核被中断时，新的服务请求将会以Job的形式添加到内核队列中。内核队列可以被第一个进入内核的中断ISR或者任务清空，但不管是中断ISR还是任务，最终都以在调度器reschedule()的末尾清空内核队列。

由于内核队列采用单读者/多写者的模式，因此我们必须在写者在向内核队列添加Job的过程中关中断，由于读者从来不会中断写者，因此中断只在写者需要引导队列写索引的时候关闭。

其实现如下：

void workQAdd0( FUNCPTR  func )

{

    int level = intLock ();                   /\* 关中断 \*/

    FAST JOB \*pJob = (JOB \*) &pJobPool [workQWriteIx];

    workQWriteIx += 4;                   /\* 移到写索引 \*/

    if (workQWriteIx == workQReadIx)

                  workQPanic ();                   /\* 如果内核队列满，则在关中断的情况下退出内核 \*/

    intUnlock (level);                         /\* 开中断 \*/

    workQIsEmpty = FALSE;            /\* 标识内核队列现在非空 \*/

    pJob->function = func;               /\*构造Job\*/

}

添加带一个参数的Job到内核队列中：

void workQAdd1 (FUNCPTR func,  int arg1 )

{

    int level = intLock ();                   /\*关中断 \*/

    FAST JOB \*pJob = (JOB \*) &pJobPool [workQWriteIx];

    workQWriteIx += 4;                   /\* 移到写索引\*/

    if (workQWriteIx == workQReadIx)

                  workQPanic ();                   /\* leave interrupts locked \*/

    intUnlock (level);                         /\* 开中断 \*/

    workQIsEmpty = FALSE;            /\* 标识内核队列非空 \*/

    pJob->function = func;               /\*向Job中添加函数 \*/

    pJob->arg1 = arg1;                     /\* 向Job中添加函数参数 \*/

}

添加带两个参数的Job到内核队列中：

void workQAdd2(FUNCPTR func,  int arg1,  int arg2 )

{

    int level = intLock ();                   /\* 关中断 \*/

    FAST JOB \*pJob = (JOB \*) &pJobPool [workQWriteIx];

    workQWriteIx += 4;                   /\* advance write index \*/

    if (workQWriteIx == workQReadIx)

                  workQPanic ();                   /\* leave interrupts locked \*/

    intUnlock (level);                         /\* 开中断 \*/

    workQIsEmpty = FALSE;            /\* we put something in it \*/

    pJob->function = func;               /\* 向Job中添加函数\*/

    pJob->arg1 = arg1;                     /\* 向Job中添加参数\*/

    pJob->arg2 = arg2;                     /\* 向Job中添加参数\*/

}

清空内核队列：

void workQDoWork (void)

{

FAST JOB \*pJob;

    int oldErrno = errno;                           /\* save errno \*/

    while (workQReadIx != workQWriteIx)

         {

        pJob = (JOB \*) &pJobPool [workQReadIx];      /\* get job \*/

         /\* 在执行内核Job函数之前，增加读索引，因为Job函数有可能是时钟处理函数

\* windTickAnnounce () ，它也是通过这个Job函数进行调用。

          \*/

                  workQReadIx += 4;

        (FUNCPTR \*)(pJob->function) (pJob->arg1, pJob->arg2);

                  workQIsEmpty = TRUE;                      /\* 标识内核队列有空位置 \*/

         }

    errno = oldErrno;                                 /\* restore \_errno \*/

}

Wind内核中的三个队列、在加上各种信号量上的等待队列构成了wind内核最核心的资源，位于wind内核的内核态中，由内核全局变量kernelState进行保护。只有在windLib库中的内核态例程wind\*开头的例程才可以访问。非内核态的例程只有进入内核态，才能调用wind\*例程，访问并操作这三个内核队列、以及各种信号量等待队列。

**5.4 kernelInit()构造初始化任务taskRoot上下文**

kernelInit()函数：

    kernelInit ((FUNCPTR) usrRoot, ROOT\_STACK\_SIZE, MEM\_POOL\_START,

                sysMemTop (), ISR\_STACK\_SIZE, INT\_LOCK\_LEVEL);

其中：

#define ROOT\_STACK\_SIZE         10000   /\* size of root's stack, in bytes \*/

#define INT\_LOCK\_LEVEL          0x0     /\* 80x86 interrupt disable mask \*/

#define ISR\_STACK\_SIZE          1000    /\* size of ISR stack, in bytes \*/

MEM\_POOL\_START标识内核映像在内存中的结束位置，通过链接脚本的end来标识。

kernelInit()代码实现如下，我们假设目标平台为Pentium，所有这里删除与Pentium平台无关代码，所有X86平台栈均向下增长。

void kernelInit

(

    FUNCPTR rootRtn,            /\* 用户启动例程 \*/

    unsigned  rootMemSize, /\*给 TCB 和初始任务栈分配的内存 \*/

    char \*       pMemPoolStart,      /\* 内存池的起始地址 \*/

    char \*       pMemPoolEnd,         /\* 内存池的结束地址 \*/

    unsigned  intStackSize,    /\* 中断栈大小 \*/

    int              lockOutLevel    /\* 关中断级别 (1-7) \*/

)

{

    union

         {

                  double   align8;        /\* 8-byte alignment dummy \*/

                  WIND\_TCB initTcb;       /\* context from which to activate root \*/

         } tcbAligned;/\*共用体的使用确保初始任务TCB八字节对齐\*/

    WIND\_TCB \*  pTcb;           /\* pTcb初始任务TCB指针\*/

    unsigned  rootStackSize; /\* 初始任务的实际栈大小 \*/

    unsigned  memPoolSize;  /\* 初始内存池的实际大小\*/

    char \*       pRootStackBase;     /\* 初始任务栈基地址 \*/

    /\* 使得输入参数按照指定的字节(一般4字节对齐) \*/

    rootMemNBytes = STACK\_ROUND\_UP(rootMemSize);

    pMemPoolStart = (char \*) STACK\_ROUND\_UP(pMemPoolStart);

    pMemPoolEnd   = (char \*) STACK\_ROUND\_DOWN(pMemPoolEnd);

    intStackSize  = STACK\_ROUND\_UP(intStackSize);

    /\*初始化vxWorks中断级别\*/

    intLockLevelSet (lockOutLevel);

    /\* 时间片轮转调度模型默认禁止\*/

    roundRobinOn = FALSE;

    /\*时钟滴答初始化为0 \*/

    vxTicks = 0;                                   /\* good morning \*/

#if   (\_STACK\_DIR == \_STACK\_GROWS\_DOWN)

    vxIntStackBase = pMemPoolStart + intStackSize;//设置中断栈基地址

    vxIntStackEnd  = pMemPoolStart;           //设置中断栈尾地址

    bfill (vxIntStackEnd, (int) intStackSize, 0xee);//用0xee填充中断栈

    windIntStackSet (vxIntStackBase);//设置wind内核的中断栈基地址指针vxIntStackPtr

    pMemPoolStart = vxIntStackBase;

#else         /\* \_STACK\_DIR == \_STACK\_GROWS\_UP \*/

<略>

#endif      /\* (\_STACK\_DIR == \_STACK\_GROWS\_UP) \*/

    /\* Carve the root stack and tcb from the end of the memory pool.  We have

     \* to leave room at the very top and bottom of the root task memory for

     \* the memory block headers that are put at the end and beginning of a

     \* free memory block by memLib's memAddToPool() routine.  The root stack

     \* is added to the memory pool with memAddToPool as the root task's

     \* dieing breath.

     \*/

    rootStackSize  = rootMemNBytes - WIND\_TCB\_SIZE - MEM\_TOT\_BLOCK\_SIZE;

    pRootMemStart  = pMemPoolEnd - rootMemNBytes;

#if     (\_STACK\_DIR == \_STACK\_GROWS\_DOWN)

    pRootStackBase = pRootMemStart + rootStackSize + MEM\_BASE\_BLOCK\_SIZE;

    pTcb           = (WIND\_TCB \*) pRootStackBase;

#else         /\* \_STACK\_GROWS\_UP \*/

<略>

#endif       /\* \_STACK\_GROWS\_UP \*/

//这里把taskIdCurrent初始化为0，是因为taskInit()会进入内核态，执行windSpawn()将当前

//初始任务放入活动队列(activceQueue)，然后调用windExit()退出内核态，在windExit()逻辑

//中会判断taskIdCurrent和就绪队列的头readyQHead是否相等，如果相等则说明当前任务

//是优先级最高的任务，不需要进行上下文切换,这我们的情景中taskIdCurrent为NULL，而

//此时内核队列也为空，即readyQHead也为NULL，则不需要进行上下文切换，又由于此时

//内核队列为空，所以windExit()直接放回，这正是我们想要的结果，windExit()判断逻辑如

//下图黄色部分所示。

    taskIdCurrent = (WIND\_TCB \*) NULL;    /\* 初始化化taskIdCurrent为空 \*/

    bfill ((char \*) &tcbAligned.initTcb, sizeof (WIND\_TCB), 0);

    memPoolSize = (unsigned) ((int) pRootMemStart - (int) pMemPoolStart);

//初始化任务，并将初始化任务放入活动队列，此时任务保持挂起(SUSPEND)状态

//注意初始化任务的优先级为0

    taskInit (pTcb, "tRootTask", 0, VX\_UNBREAKABLE | VX\_DEALLOC\_STACK,

               pRootStackBase, (int) rootStackSize, (FUNCPTR) rootRtn,

               (int) pMemPoolStart, (int)memPoolSize, 0, 0, 0, 0, 0, 0, 0, 0);

    rootTaskId = (int) pTcb;                      /\* fill in the root task ID \*/

    /\* Now taskIdCurrent needs to point at a context so when we switch into

     \* the root task, we have some place for windExit () to store the old

     \* context.  We just use a local stack variable to save memory.

     \*/

//现在将taskIdCurrent初始化为一个临时的的TCB控制块，taskActive()进入内核态，调用

//windResume()将初始任务taskRoot放入就绪队列，此时readyQHead指向就绪队列中唯一

//的任务taskRoot初始任务，当taskActive()条用windExit()退出内核态时，由于readyQHead

//和taskIdCurrent不等，windExit()将调用调度器恢复readyQHead指向的队首任务的上下文，

//即恢复taskRoot的上下文。由于windExit()在调用调度器恢复taskRoot任务上下文之前，

//保持当前任务taskIdCurrent的上下文当当前任务的TCB控制块中，所里这里才定义了一

//个临时的上下文空间tcbAligned.initTcb，由于这个临时空间在临时栈中分配，当taskRoot

//任务起来后，临时栈即被舍弃了，因此不需要再回收了。这个情景中windExit()的执行逻

//辑，如下图红色部分所示。

    taskIdCurrent = &tcbAligned.initTcb;        /\* update taskIdCurrent \*/

    taskActivate ((int) pTcb);                   /\* activate root task \*/

}

分析：windExit()的执行流程如图6.8所示。

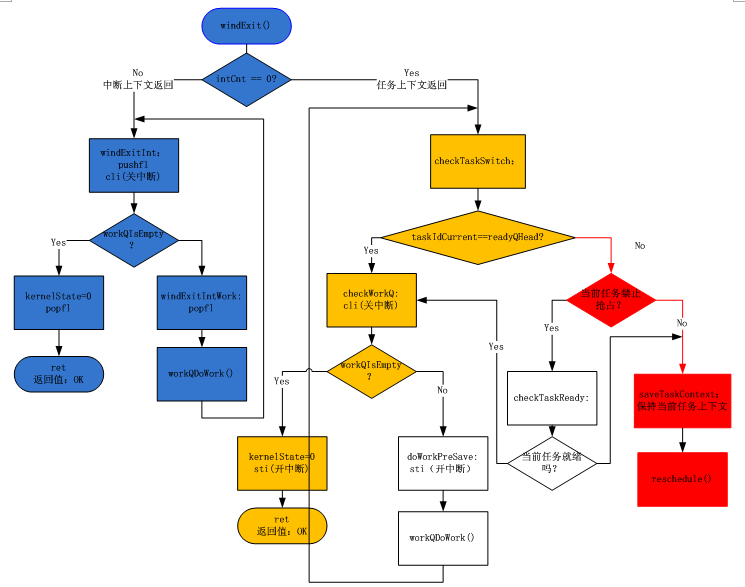


图6.8 windExit()执行流程

我们在前面的博文[VxWorks内核解读-3](http://www.prtos.org/vxworks-wind-scheduler/)已经分析了windExit()的执行流程，这里不再赘述。

备注：这是有一点需要注意，taskActivate()调用windExit()恢复taskRoot的上下文后，启动的任务并不是usrRoot()，而是void    vxTaskEntry ()函数，由vxTaskEntry()来调用usrRoot()函数。

vxTaskEntry()代码如下：

FUNC\_LABEL(vxTaskEntry)

         xorl  %ebp,%ebp               /\* make sure frame pointer is 0 \*/

         movl          FUNC(taskIdCurrent),%eax /\* get current task id \*/

         movl          WIND\_TCB\_ENTRY(%eax),%eax /\* entry point for task is in tcb \*/

         call   \*%eax                         /\* call main routine \*/

         addl $40,%esp                   /\* pop args to main routine \*/

         pushl         %eax                           /\* pass result to exit \*/

         call   FUNC(exit)                 /\* gone for good \*/

这样做的目的有三个：

1. 任务的真正入口函数保存在任务控制块中，很容易通过taskRestart()重新启动；
2. vxTaskEntry()函数的引入，使得任务的主函数体相对于vxTaskEntry()来说是一个普通的函数调用，其任务栈可以被编译器自动清理，也便于调试栈回溯工具处理主函数例程的调用。
3. 从vxTaskEntry()的代码我们可以看出，任务的主函数执行完毕后，将会调用exit()函数回收该任务的资源，这样就编译对删除的任务回收期资源。

现在我们接着分析初始任务taskRoot的主函数例程usrRoot()吧，O(∩\_∩)O~。

**6.5 初始化任务taskRoot的执行**

usrRoot()属于用户自定义的例程，主要完成VxWorks内核的初始化，比如初始化I/O系统，安装驱动，创建设备，建立协议栈等待，这是都是可以通过用户来配置，它也可以创建系统符号表。

我们现在不考虑其他外围组件，只考虑Wind内核的执行，其usrRoot的实现如下：

void usrRoot (char \*pMemPoolStart, unsigned memPoolSize)

 {

usrKernelCoreInit ();               /\* vxWorks核心的初始化 \*/

//vxWorks的核心初始化化包括事件模块、二值信号量模块、互斥信号量模块、计数信

//号量模块、消息队列、看门狗、以及任务创建、删除、上下文切换钩子模块的初始化

    memInit (pMemPoolStart, memPoolSize); /\* 初始化内存分配器 \*/

memPartLibInit (pMemPoolStart, memPoolSize); /\* 初始化核心内存管理单元 \*/

// memInit()以及保护了memPartLibInit()的调用，因此再次显示调试memPartLibInit()其

//实是没有必要的,还好memPartLibInit()用了一个全局变量memPartLibInstalled，借以验

//证memPartLibInit()是否已经被调用过.

   sysClkInit ();      /\* 挂接时钟中断，并初始化时钟\*/

    usrMmuInit();           /\*建立一一对应的MMU映射\*/

    usrAppInit ();     /\* 调用用户自定义例程\*/

}

分析：

由于我们目前仅仅分析vxWorks的wind内核的工作机制，所有vxWorks的其它组件，比如I/O模块，文件系统，shell等等暂不考虑。

至此，到VxWorks运行到usrAppInit()时，vxWorks的wind内核的多任务运行环境，已经运行起来，我们可以在usrAppInit()函数中，创建我们的应用调用vxWorks提供的服务来执行。

比如：

/\*

\* usrAppInit - initialize the users application

\*/

#include "vxWorks.h"

#define DEMO\_PRI 149

extern void windDemo(int iteration);

void usrAppInit (void)

{

#ifdef        USER\_APPL\_INIT

         USER\_APPL\_INIT;              /\* for backwards compatibility \*/

#endif

         printk("hello vxWorks\n");

//创建一个demoTask任务来运行

         taskSpawn("demoTask", DEMO\_PRI, 0x0001, 4000, (FUNCPTR) windDemo, 20, 0,0,0,0,0,0,0,0,0);

    /\* add application specific code here \*/

}