VxWorks

程序员速查手册

周启平 张杨 编著

机械工业出版社
CHINA MACHINE PRESS

VxWorks 程序员速查手册

周启平　张　杨　编著

机 械 工 业 出 版 社

本书介绍了 VxWorks 操作系统的常用函数库，并对库中各函数进行了详细描述。全书共 12 章，主要内容包括：任务管理与调度函数、任务同步与通信函数、时钟管理函数、中断管理函数、内存管理函数、I/O 系统函数、文件系统函数、系统内核函数、用户函数、浮点函数、网络系统函数和错误状态函数等。

本书语言通畅、条理清晰、内容全面且深入浅出，以精选实例加文字说明的方式结合编者多年的实际开发经验编写而成。它是 VxWorks 程序员的案头参考书，无论是 VxWorks 初学者还是资深程序员都能从中受益匪浅。

图书在版编目（CIP）数据

Vx Works 程序员速查手册 / 周启平，张杨编著. —北京：机械工业出版社，2005.2
ISBN 7-111-16123-8

Ⅰ. V… Ⅱ. ①周…②张… Ⅲ. 实时操作系统，Vx Works—技术手册 Ⅳ. TP316.2-62

中国版本图书馆 CIP 数据核字（2005）第 009276 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
责任编辑：吉 玲（E-mail：jiling@mail.machineinfo.gov.cn）
责任印制：石 冉
三河市宏达印刷有限公司印刷 · 新华书店北京发行所发行
2005 年 2 月第 1 版第 1 次印刷
787mm × 1092mm 1/16 · 20.25 印张 · 652 千字
0001—4000 册
定价：36.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
本社购书热线电话（010）68993821、88379646
68326294、68320718
Http://www.machineinfo.gov.cn/book/

前　言

随着 VxWorks 操作系统在嵌入式领域越来越广泛的应用，一种新的书籍需求正在形成。即编程者需要多种相关的速查手册。该书的出版正是为了适应这样的需求，为广大程序员提供最大的便利。为了能做到给程序员提供正确的帮助，书中的函数说明力争可以完全地符合 WindRiver 公司所提供的说明文档，而所有的例子则尽量使用公开的程序片断。

本书适合于使用 VxWorks 操作系统的人，不论是入门级的或是资深级的程序员都可以将本书作为函数速查手册使用。书中提供了一些例子和函数库的详细说明，使得该书也可以作为 VxWorks 的参考手册来使用。在书中，很多函数库的说明较为详细，可以对其他的资料作一定程度的补充。

本书包括了任务的调度和管理、终端管理、系统时钟的管理、内存管理、IO 系统、文件系统、网络系统、用户接口、异常处理等诸多分类的函数。在函数的选材上，我们希望能包含所有的常用的函数，但有很多函数库和函数的使用程度是我们无法预料的，希望读者能将意见及时地反馈给我们。

该书由周启平和张杨合作完成，其中第 1～4、9、11 章由周启平完成，第 5～8、10、12 章由张杨完成。例子的添加与测试，章节的修改、校对等工作，是由周启平和张杨共同完成的。在此，向所有支持本书编写的人员表示感谢。

同时还要感谢出版社的编辑们、我们的亲友和同事在该书出版过程中对我们的帮助和鼓励。尽管抱着认真的态度，但在书中还是难免会出现各种各样的错误，希望各位读者能不吝赐教。

我们的联系方式：
周启平：qpzhou@sina.com
张杨：hwybird@sohu.com

编者

目　录

第1章　任务管理与调度

1.1　任务管理函数

1.1.1　函数库描述

1．库命名

任务管理函数库名称为 taskLib。

2．函数

表 1-1 中列出了任务管理函数。

表 1-1　任务管理函数

函　数	描　述	函　数	描　述
taskSpawn()	创建并激活任务	taskPriorityGet()	获得任务优先级
taskInit()	初始化任务（指定堆栈地址）	taskLock()	禁止任务调度
taskActivate()	激活已经被初始化的任务	taskUnlock()	使能任务调度
exit()	结束任务并释放内存	taskSafe()	保护任务，防止其被非法删除
taskDelete()	删除任务	taskUnsafe()	解除任务的安全保护
taskDeleteForce()	无条件删除任务	taskDelay()	任务延时（参数为时间片）
taskSuspend()	挂起任务	taskIdSelf()	获得当前正在运行任务的任务 ID
taskResume()	恢复任务	taskIdVerify()	核实任务的存在
taskRestart()	重启任务	taskTcb()	获得任务控制块
taskPrioritySet()	改变任务优先级		

3．描述

任务管理库为 VxWorks 的任务管理机制提供了接口。任务控制服务是由 VxWorks 内核提供，其内核包括 kernelLib 库、taskLib 库、semLib 库、tickLib 库、msgQLib 库和 wdLib 库。taskInfo 库提供了访问和调试任务信息的函数。至于更高级别的任务信息显示函数则由 taskShow 库来提供。

4．任务创建

任务创建通常是由 taskSpawn()函数来完成的。其创建过程包括：为任务的堆栈和任务控制块（WIND_TCB）分配内存、初始化任务控制块和启动任务控制块。如果用户有特殊需求，可以使用更低级别函数 taskInit()和 taskActivate()，并且它们就是函数 taskSpawn()的基本构成。

VxWorks 系统采用优先级抢占调度方式，任务执行在体系结构所决定的特权状态中。

对于 VxWorks 系统而言，只要有足够的可用内存能够满足系统分配需求，其创建任务的个数是没有限制的。

usrLib 库提供的 sp()函数是创建任务的一个简单缩写，它使用缺省参数调用 taskSpawn()函数。

5．任务删除

在任务创建期间，如果任务退出它的“主”程序，内核隐含调用 exit()函数删除这个任务。事实上用户可以通过调用 taskDelete()或 exit()函数删除任务。

由于资源回收困难，用户需要小心对待任务删除操作。删除一个任务时，其拥有的临界资源可能会破坏

系统，因为这个资源可能再也不可用。因为系统在删除任务的时候并不知道资源所处的状态，所以在此时简单地释放资源不是可行的解决方案。

采用删除保护方式解决任务删除问题胜于采用其他过度复杂的删除机制。对于未预料到的删除操作，用户可以通过使用 taskSafe()函数来保护任务。当任务处于删除保护的时候，删除者将会被阻塞直到解除任务的删除保护。同样，任务可以通过获得一个互斥信号量来保护自己免于被删除，而这个互斥信号量是在任务创建时选中 SEM_DELETE_SAFE 选项来创建的。无论系统调用 taskSafe()函数还是调用 semTake()函数，系统都需要调用与之对应的 taskUnsafe()和 semGive()函数(有关详细信息请参考 2.1.3 节)。许多 VxWorks 系统资源都采用这种方式来保护。应用程序设计者在可能发生动态的任务删除操作之处也应该考虑使用这种方法。

系统也会使用 sigLib 模块，允许任务在实际终止前执行清除代码。

6．任务控制

在任务创建的时候，系统会给任务分配一个 ID，其后系统对任务的操作是依靠这个 ID 进行的。在任务控制函数中，VxWorks 通过使用指定的任务 ID 控制任务。

下面这些函数可以控制任务状态：taskResume()、taskSuspend()、taskDelay()、taskRestart()、taskPrioritySet()和 taskRegsSet()。

7．任务调度

VxWorks 系统基于优先级抢占方式调度任务。任务可以拥有的优先级范围为 0～255，其中 0 为最高优先级，255 为最低优先级。VxWorks 中任务的优先级是可以动态改变的，用户可以通过调用 taskPrioritySet()函数来改变任务现有的优先级。

8．头文件

任务管理函数声明在 taskLib.h 中。

9．参考

相关信息请参考 taskInfo、taskShow、taskHookLib、taskVarLib、semLib、semMLib、kernelLib 库描述，以及“VxWorks Programmer's Guide”书中的基本操作系统章节。

1.1.2 任务管理函数详细描述

taskSpawn()

函数原型：

```
int taskSpawn
(
char * name,            /* 新任务名称(存储在 pStackBase 中) */
int priority,           /* 新任务优先级 */
int options,            /* 任务选项字 */
int stackSize,          /* 任务所需堆栈大小(字节) */
FUNCPTR entryPt,        /* 新任务入口点 */
int arg1,               /* 以下是传递给任务函数的 10 个参数 */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8,
int arg9,
int arg10
```

)

功能描述：

创建并激活任务。该函数创建并激活一个带有指定优先级和选项字的新任务，同时返回一个系统分配的 ID 号。相关信息参考该函数的组成部分：taskInit()和 taskActivate()。

在任务创建时可以分配一个名称给任务，便于调试。任务名将会出现在各种系统信息函数所产生的显示中，例如 i()函数。任务名的长度和内容可以是任意的，不过当前 VxWorks 系统约定任务名的长度不超过 10 个字符而且任务名第一个字母为“t”。如果 name 参数为 NULL，系统将为该任务分配一个 ASCII 名称，其形式为“*tn*”，在这里“*n*”是一个随着新任务创建而增加的整数。

分配给新任务的堆栈资源大小由 stackSize 参数指定，而资源是从系统内存区中分配得到。堆栈大小应该是一个偶数。任务控制块（TCB）存放在堆栈中，像任务名一样也需要一些内存。剩余的任务堆栈通过 checkStack()函数用变量 0xEE 来填充堆栈的每个字节。相关信息请参考堆栈检验函数 checkStack()。

入口地址 entryPt 是任务主函数的地址。一旦建立了 C 环境，系统将调用该函数。该函数最多可以带有 10 个给定的参数。在主函数将要返回时系统自动地调用 exit()。

值得注意的是 10 个（最多 10 个）参数必须传递给创建函数。

选项字参数中的位可以设置任务运行在如下模式：

VX_FP_TASK (0x0008)：浮点协处理器支持；

VX_PRIVATE_ENV (0x0080)：包含私有环境支持(参考 envLib)；

VX_NO_STACK_FILL (0x0100)：不使用 checkStack()填充堆栈；

VX_UNBREAKABLE (0x0002)：禁止断点调试。

有关选项字定义请参考头文件 taskLib.h。

返回值：

成功创建任务则返回任务 ID，如果内存不足或不能创建任务则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE、S_objLib_OBJ_ID_ERROR、S_memLib_BLOCK_ERROR、S_smObjLib_NOT_INITIALIZED、S_memLib_NOT_ENOUGH_MEMORY。

参考：

相关信息请参考 taskLib 库描述、taskInit()、taskActivate()、sp()函数等。

例：

```
/* 数据定义 */
#define     TASK_PRI150         /* 任务优先级 */
int         taskId;             /* 任务 ID */
void   taskDemo (void);         /* 任务入口函数声明 */
...
void mainDemo(void)
{
/* 创建并启动任务 */
taskId = taskSpawn ("tDemo", TASK_PRI, VX_FP_TASK, 4000,
        (FUNCPTR) taskDemo, 0,0,0,0,0,0,0,0,0,0);
if(taskId == ERROR)
printf("spawn taskDemo failed!\n");
}
/* 演示任务入口函数 */
void taskDemo (void)
```

```
    {
...
    for (;;)
{
printf("taskDemo is running!\n");
...
}
    }
```

taskInit()

函数原型：

```
    STATUS taskInit
    (
    WIND_TCB * pTcb,      /* 新任务的 TCB 地址 */
    char * name,          /* 新任务名称 (存储在 pStackBase 中) */
    int priority,         /* 新任务优先级 */
    int options,          /* 任务选项字 */
    char * pStackBase,    /* 新任务的堆栈基地址 */
    int stackSize,        /* 任务所需堆栈大小(字节) */
    FUNCPTR entryPt,      /* 新任务入口点 */
    int arg1,             /* 以下是传递给任务函数的 10 个参数 */
    int arg2,
    int arg3,
    int arg4,
    int arg5,
    int arg6,
    int arg7,
    int arg8,
    int arg9,
    int arg10
    )
```

功能描述：

初始化任务。该函数初始化用户分配给任务堆栈和控制块的内存区域，代替从系统内存中自动分配的空间（taskSpawn()自动分配内存空间）。该函数将使用指向 WIND_TCB 和堆栈的指针作为任务的组成部分，这样允许一个静态 WIND_TCB 变量的初始化，也允许对特殊的堆栈进行配置用于辅助调试。

在 taskSpawn()中，系统会分配一个名称给任务。taskSpawn()自动为未命名的任务命名，而 taskInit()允许未命名任务的存在。对于其他任务函数而言，任务 ID 是必需的，它实际上是参数 pTcb 的地址。

值得注意的是任务堆栈的基地址由参数 pStackBase 决定，根据目标机体系结构向上或向下增长。

该函数的其他参数与 taskSpawn()中的参数一样。与 taskSpawn()不同的是 taskInit()并不激活任务。任务的激活操作必须是在调用 taskInit()之后调用 taskActivate()函数来完成。

通常，用户使用 taskSpawn()而非 taskInit()启动任务，除非当任务内存分配需要额外控制或希望单独激活任务。

返回值：

成功初始化任务则返回 OK，如果初始化任务失败则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE、S_objLib_OBJ_ID_ERROR。

参考：

相关信息请参考 taskLib 库描述、taskActivate()和 taskSpawn()函数。

例：

```
/* 数据定义 */
#define     TASK_PRI100         /* 任务优先级 */
WIND_TCB    * pTcb;             /* 任务的 TCB 地址 */
int         status;             /* 函数返回状态值 */
void        taskDemo (void);    /* 任务入口函数声明 */
int         pStackBase          /* 堆栈基地址 */
...
void mainDemo(void)
{
pStackBase = 0x2000;

/* 创建并启动任务 */
status = taskInit (pTcb, "tDemo", TASK_PRI, VX_FP_TASK,(char *)pStackBase, 500,
                   (FUNCPTR) taskDemo, 0,0,0,0,0,0,0,0,0,0);
if(status == ERROR)
printf("initialize taskDemo failed!\n");
}
/* 演示任务入口函数 */
void taskDemo (void)
    {
...
    for (;;)
{
printf("taskDemo is running!\n");
...
}
    }
```

taskActivate()

函数原型：

```
STATUS taskActivate
(
int tid  /* 任务 ID */
)
```

功能描述：

激活已经初始化的任务。该函数激活用 taskInit()函数创建的任务。如果任务没有激活，则任务没有资格让调度程序分配 CPU 给它。参数 tid（任务 ID）是任务的 WIND_TCB 地址，且为一个整数：

```
tid = (int) pTcb;
```

taskSpawn()函数是由 taskActivate()和 taskInit()函数组成。通过 taskSpawn()函数创建的任务并不需要激活

任务操作。

返回值：

成功激活任务则返回 OK，如果任务激活失败则返回 ERROR。

参考：

相关信息请参考 taskLib 库描述和 taskInit()函数。

exit()

函数原型：

```
void exit
(
int code            /* 传递给删除钩子函数的代码，存储在 TCB 中 */
)
```

功能描述：

任务退出。这是一个 ANSI C 函数，任务通过调用该函数退出任务。当创建任务的主程序退出时，系统隐含地调用该函数。参数 code 将存储在 WIND_TCB 中，删除钩子函数或后继的调试可能会用到它。

返回值：

无。

参考：

相关信息请参考 taskLib 库描述、taskDelete()函数和“VxWorks Programmer's Guide”中的基本操作系统章节。

taskDelete()

函数原型：

```
STATUS taskDelete
(
int tid         /* 任务 ID */
)
```

功能描述：

删除任务。该函数停止并删除指定的任务，同时重新分配相关的堆栈和 WIND_TCB 内存资源。任务删除前，在被删除任务的上下文中将调用所有通过 taskDeleteHookAdd()函数添加的钩子函数。另外，该函数是 taskSpawn()的对应函数。

返回值：

成功删除任务则返回 OK,任务删除失败则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE、S_objLib_OBJ_DELETED、S_objLib_OBJ_UNAVAILABLE、S_objLib_OBJ_ID_ERROR。

参考：

相关信息请参考 taskLib、excLib 库描述，taskDeleteHookAdd()和 taskSpawn()函数，以及“VxWorks Programmer's Guide”中的基本操作系统章节。

例：

```
int         taskId;             /* 任务 ID */
...
/* 查找任务名为“tDemo”的任务 */
taskId = taskNameToId("tDemo");
```

```
if(taskId == ERROR)
    printf("Don't find tDemo!\n");
else
taskDelete(taskId);                    /* 删除任务 */
...
taskDeleteForce()
```

函数原型：

```
STATUS taskDeleteForce
(
int tid          /* 任务 ID */
)
```

功能描述：

无条件删除任务。即使任务处于删除保护状态，该函数依然可以删除这个任务。它的操作类似 taskDelete()函数。任务删除前，在被删除任务的上下文中将调用所有通过 taskDeleteHookAdd()函数添加的钩子函数。

警告：

该函数作为调试辅助函数，通常情况下不适合用于应用程序中。因为忽视任务的删除保护会使系统处于一种不稳定的状态或导致系统死锁。

调用 taskDeleteForce()函数时，系统起不到保护作用，所以用户使用该函数时要小心。

返回值：

成功删除任务则返回 OK，如果任务删除失败则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE、S_objLib_OBJ_DELETED、S_objLib_OBJ_UNAVAILABLE、S_objLib_OBJ_ID_ERROR。

参考：

相关信息请参考 taskLib 库描述，taskDeleteHookAdd()和 taskDelete()函数。

taskSuspend()

函数原型：

```
STATUS taskSuspend
(
int tid          /* 任务 ID */
)
```

功能描述：

挂起任务。该函数挂起指定的任务。如果任务 ID 值为 0，则表明任务挂起自身。处于挂起状态的任务可以追加其他状态，例如任务可以处于挂起+延迟状态，或挂起+阻塞状态。挂起+延迟状态的任务在延迟到期后任务仍然是处于挂起状态。同样,挂起+阻塞状态的任务在解除阻塞后任务仍然是处于挂起状态。

在任务的异步通信中，该函数的使用应极度小心。当挂起指定的任务时，系统是不会考虑任务当前情形的。譬如任务持有针对某系统资源（例如网络或系统内存块）的互斥信号量，如果在这个时期挂起任务，由于互斥信号量被占用而其他任务得不到该资源，这种情形下系统经常会出现死锁情况。

该函数是调试和异常管理包的基础。然而，像同步机制一样，用户应该更多地采用通用信号量机制，而不是使用该功能。

返回值：

成功挂起任务则返回 OK，如果挂起任务失败则返回 ERROR。

错误码：

S_objLib_OBJ_ID_ERROR

参考：

相关信息请参考 taskLib 库描述。

taskResume()

函数原型：

```
STATUS taskResume
(
int tid         /* 任务 ID */
)
```

功能描述：

恢复任务。该函数恢复指定的任务。取消任务挂起并恢复任务到原来的运行状态。

返回值：

成功恢复任务则返回 OK，如果恢复任务失败则返回 ERROR。

错误码：

S_objLib_OBJ_ID_ERROR

参考：

相关信息请参考 taskLib 库描述。

例：

```
/* 数据定义 */
int         taskAId;            /* 任务 A ID */
int         taskBId;            /* 任务 B ID */
void        taskA (void);       /* 任务 A 入口函数声明 */
void        taskB (void);       /* 任务 B 入口函数声明 */
...
void mainDemo(void)
{
...
/* 创建并启动任务 */
taskAId = taskSpawn ("tTaskA", 100, VX_FP_TASK, 4000,
          (FUNCPTR) taskA, 0,0,0,0,0,0,0,0,0,0);
taskBId = taskSpawn ("tTaskB", 120, VX_FP_TASK, 4000,
          (FUNCPTR) taskB, 0,0,0,0,0,0,0,0,0,0);
}
/* 任务 A 入口函数 */
void taskA (void)
    {
...
    for (;;)
{
printf("taskA is running!\n");
...
taskSuspend(0);         /* 挂起自身 */
```

```
}
    }

/* 任务 B 入口函数 */
void taskB (void)
    {
...
    for (;;)
{
printf("taskB is running!\n");
...
taskResume(taskAId);        /* 恢复任务 A */
}
    }
```

taskRestart()

函数原型：

```
STATUS taskRestart
(
int tid   /* 任务 ID */
)
```

功能描述：

重新启动任务。该函数重新启动指定的任务。该函数首先停止任务，并用相同的任务 ID、优先级、选项字、最初的入口点、堆栈大小和参数重新初始化任务。任务的自身重启是通过异常任务来完成的。当出现异常中断时，shell 利用该函数重新启动自身。

注意：

如果任务的任何一个启动参数被修改，被重新启动的任务将用修改后的参数启动任务。

返回值：

成功重新启动任务则返回 OK，如果输入无效的任务 ID 或任务重新启动失败则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE、S_objLib_OBJ_DELETED、S_objLib_OBJ_UNAVAILABLE、S_objLib_OBJ_ID_ERROR、S_smObjLib_NOT_INITIALIZED、S_memLib_NOT_ENOUGH_MEMORY、S_memLib_BLOCK_ERROR。

参考：

相关信息请参考 taskLib 库描述。

taskPrioritySet()

函数原型：

```
STATUS taskPrioritySet
(
int tid,                    /* 任务 ID */
int newPriority             /* 新的优先级 */
)
```

功能描述：

改变任务优先级。该函数把任务的优先级改变成指定的优先级。其优先级范围从 0~255，其中 0 为最高优先级，255 为最低优先级。

返回值：

成功改变任务优先级则返回 OK，如果输入无效的任务 ID 则返回 ERROR。

错误码：

S_taskLib_ILLEGAL_PRIORITY、S_objLib_OBJ_ID_ERROR。

参考：

相关信息请参考 taskLib 库描述及 taskPriorityGet()函数。

例：

```
int             taskId;                 /* 任务 ID */
...
/* 查找任务名为“tDemo”的任务 */
taskId = taskNameToId("tDemo");
...
if(taskId == ERROR)
    printf("Don't find tDemo!\n");
else
taskPrioritySet(taskId,120);        /* 设置任务优先级 */
...
```

taskPriorityGet()

函数原型：

```
STATUS taskPriorityGet
(
int tid,                            /* 任务 ID */
int * pPriority                     /* 存放返回的优先级 */
)
```

功能描述：

获得任务的优先级。该函数获得指定任务的当前优先级，并复制任务的当前优先级到参数 pPriority 指向的整数中。

返回值：

成功获得任务优先级则返回 OK，如果输入无效的任务 ID 则返回 ERROR。

错误码：

S_objLib_OBJ_ID_ERROR

参考：

相关信息请参考 taskLib 库描述及 taskPrioritySet()函数。

taskLock()

函数原型：

```
STATUS taskLock (void)
```

功能描述：

禁止任务调度。该函数禁止任务上下文切换。任务调用该函数将导致系统只允许该任务执行，除非任务自己明确地放弃 CPU 资源并不再处于就绪状态。该函数与 taskUnlock()函数成对出现，它们共同包围一段临界代码。任务优先级抢占锁允许嵌套，这样应用程序调用多少个 taskLock()函数，直到应用程序调用相同个

数的 taskUnlock()函数时系统才打开优先级抢占锁。

该函数并不禁止中断，禁止中断操作需要调用 intLock()函数。

作为互斥现象的一种方式，taskLock()函数比 intLock()函数更可取，因为中断锁增加了系统潜伏的中断；而与 taskLock()函数相比，semTake()函数更可取，因为优先级抢占锁增加了系统潜伏的优先级抢占。

值得说明的是在中断服务程序中不可以调用 taskLock()函数。

返回值：

成功调用则返回 OK，调用失败则返回 ERROR。

错误码：

S_objLib_OBJ_ID_ERROR、S_intLib_NOT_ISR_CALLABLE

参考：

相关信息请参考 taskLib 库描述、taskUnlock()、intLock()、taskSafe()和 semTake()函数。

例：

```
int             count;                  /* 任务运行计数 */
void   taskDemo (void);                 /* 任务入口函数声明 */
...
/* 演示任务入口函数 */
void taskDemo (void)
    {
...
    for (;;)
    {
    printf("taskDemo is running!\n");
...
    /* 保护全局变量 count */
        taskLock();
    count++;
        taskUnlock();
    }
    }
```

taskUnlock()

函数原型：

STATUS taskUnlock (void)

功能描述：

打开任务调度。该函数减少任务优先级抢占锁个数。该函数调用与 taskLock()函数成对出现，以结束一段临界代码。应用程序调用多个 taskLock()函数，直到应用程序调用相同个数的 taskUnlock()函数时系统才打开优先级抢占锁。当优先级抢占锁个数减少到 0 时，任何符合条件的任务都可以抢占当前任务。

值得说明的是在中断服务程序中不可以调用 taskUnlock()函数。

返回值：

成功调用则返回 OK，调用失败则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE。

参考：

相关信息请参考 taskLib 库描述及 taskLock()函数。

taskSafe()

函数原型：

STATUS taskSafe (void)

功能描述：

保护任务。该函数保护调用它的任务，防止其被非法删除。任何任务试图删除一个处于保护下的任务都将会阻塞，直到调用 taskUnsafe()函数撤消任务删除保护操作。当任务处于非删除保护下时，删除者将不会被阻塞并允许删除该任务。

taskSafe()函数使用一个计数来表明任务保护的嵌套调用。当嵌套出现时，只有在调用相同数量的 taskUnsafe()函数之后任务才变成非删除保护状态。

返回值：

调用后返回 OK。

参考：

相关信息请参考 taskLib 库描述、taskUnsafe()函数以及“VxWorks Programmer's Guide”中的基本操作系统部分。

例：

```
charbuff[10];                    /* 共享缓冲区 */
SEM_ID    semId;                 /* 信号量 ID */
void  taskDemo (void);           /* 任务入口函数声明 */
...
/* 演示任务入口函数 */
void taskDemo (void)
    {
...
    for (;;)
    {
    printf("taskDemo is running!\n");
...
    /* 保护全局变量 count */
        taskSafe();
        semTake (semId, WAIT_FOREVER);     /* 获得信号量 */
    buff = " ";                            /* 缓冲区清空 */
        semGive (semId);                   /* 释放信号量 */
        taskUnsafe();
    }
    }
```

taskUnsafe()

函数原型：

STATUS taskUnsafe (void)

功能描述：

撤消当前任务删除保护。该函数撤消调用任务的删除保护。任何任务试图删除处于保护下的任务都将会阻塞，直到任务撤消删除保护操作。当任务处于非删除保护时，删除者将不会被阻塞并允许删除该任务。

taskUnsafe()函数使用一个计数来表明任务保护的嵌套调用。当嵌套出现时，只有在调用相同数量的

taskUnsafe()函数之后任务才变成非删除保护状态。

返回值：

调用后返回 OK。

参考：

相关信息请参考 taskLib 库描述、tasksafe()函数，以及“VxWorks Programmer's Guide”中的基本操作系统部分。

taskDelay()

函数原型：

```
STATUS taskDelay
(
int ticks       /* tick 数 */
)
```

功能描述：

延迟任务执行。该函数导致调用任务在指定的持续时间(tick)内放弃 CPU 资源。通常该函数是作为任务重新调度的参考函数。当等待某个与中断没有关联的外部条件时，该函数也非常有用。

如果调用任务接收到一个不能阻塞或忽略的信号时，在信号处理程序运行后函数 taskDelay()返回 ERROR 并设置错误号为 EINTR。

返回值：

成功调用则返回 OK，如果中断级调用或调用任务接收到一个不能阻塞或忽略的信号时则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE、EINTR。

参考：

相关信息请参考 taskLib 库描述。

例：

```
SEM_ID          semId;          /* 信号量 ID */
int             highPriId;      /* 任务 tHighPri 的 ID */
...
void taskLowPri (void)
    {
...
    for(;;)
  {
    printf("Low task is runing!\n");
    semGive (semId);            /* 释放信号量 */
    taskResume (highPriId);     /* 唤醒 tHighPri 任务 */
    taskDelay (60);             /* 任务延时 60 个 ticks */
  }
    }
```

taskIdSelf()

函数原型：

```
int taskIdSelf (void)
```

功能描述：

获得运行任务的任务 ID。该函数获得调用任务的任务 ID。如果在中断级调用该函数则任务 ID 将视为无效。

返回值：

调用任务的任务 ID。

参考：

相关信息请参考 taskLib 库描述。

taskIdVerify()

函数原型：

```
STATUS taskIdVerify
(
int tid         /* 任务 ID */
)
```

功能描述：

检验任务的存在。该函数通过确认指定 ID 与任务 ID 是否一样来检验指定任务是否存在。

返回值：

任务存在则返回 OK，如果输入的是无效任务 ID 则返回 ERROR。

错误码：

S_objLib_OBJ_ID_ERROR。

参考：

相关信息请参考 taskLib 库描述。

taskTcb()

函数原型：

```
WIND_TCB *taskTcb
(
int tid         /* 任务 ID */
)
```

功能描述：

根据任务 ID 获得任务控制块。该函数为指定任务返回一个指向该任务控制块的指针。尽管所有任务状态信息包含在任务控制块中，但用户不可以直接修改它。若需要改变记录，可以通过使用 taskRegsSet()和 taskRegsGet()函数实现。

返回值：

成功调用则返回一个指向任务控制块的指针，如果输入的是无效任务 ID 则返回 NULL。

错误码：

S_objLib_OBJ_ID_ERROR。

参考：

相关信息请参考 taskLib 库描述。

1.2 任务信息函数

1.2.1 函数库描述

1. 库命名

任务信息函数库名称为 taskInfo。

2．函数

表 1-2 中列出了任务信息函数。

表 1-2　任务信息函数

函　数	描　述	函　数	描　述
taskOptionsSet()	改变任务选项	taskNameToId()	根据任务名查找任务 ID
taskOptionsGet()	获得任务选项	taskIdDefault()	设置默认任务 ID
taskRegsGet()	从任务控制块中获得任务的寄存器内容	taskIsReady()	检查任务是否处于就绪状态
taskRegsSet()	设置任务的寄存器内容	taskIsSuspended()	检查任务是否被挂起
taskName()	根据任务 ID 获得任务名	taskIdListGet()	获得活动任务的 ID 列表

3．描述

为了获得任务的相关信息，taskInfo 库提供相应的接口函数来满足用户的需求。在应用开发期间，taskInfo 库是辅助调试和用户界面的核心部分。用户通过调用 taskOptionsGet()、taskRegsGet()、taskName()、taskNameToId()、taskIsReady()、taskIsSuspended()和 taskIdListGet()函数可以获得任务信息。taskOptionsSet()、taskRegsSet()和 taskIdDefault()这三个函数提供高级调试功能。

任务信息函数的主要缺点是在信息采集与信息显示之间任务可能已经改变了它们的状态，所以这些函数提供的信息应该被认为是系统某个时刻的快照，而不可以信赖。除非任务被托管给一个已知状态，例如挂起状态。

任务管理和控制函数由 taskLib 库提供。更高级别的任务信息显示函数由 taskShow 库提供。

4．头文件

任务信息函数的声明在头文件 taskLib.h 中。

5．参考

相关信息请参考 taskLib、taskShow、taskHookLib、taskVarLib、semLib 和 kernelLib 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统章节。

1.2.2　任务信息函数详细描述

taskOptionsSet()

函数原型：

```
STATUS taskOptionsSet
(
int tid,                /* 任务 ID */
int mask,               /* 不被设置的选项位位掩码 */
int newOptions          /* 被设置的选项位位掩码 */
)
```

功能描述：

改变任务选项。该函数改变任务执行时的选项。在任务已经被创建后，能够被改变的惟一选项是：

VX_UNBREAKABLE：禁止断点调试。

有关它的定义请阅读头文件 taskLib.h。

返回值：

成功设置选项则返回 OK，如果输入无效任务 ID 则返回 ERROR。

参考：

相关信息请参考 taskInfo 库描述和 taskOptionsGet()函数。

例：

```
int optionsSet
      (
      int taskId                /* 任务 ID */
      )
{
      int toptions;             /* 存放任务选项值 */
      int status;
      int errorStatus = 0;
/* 获得任务选项 */
status = taskOptionsGet(taskId, &toptions);
      if (status == ERROR)
      {
      errorStatus = COMMIT_FAILEDED;
      return errorStatus;
       }
      if ((toptions & VX_UNBREAKABLE) == 0)
      {
      /* 设置任务选项 */
      status = taskOptionsSet (taskId, ~toptions, (toptions | VX_UNBREAKABLE));
            if (status == ERROR)
              {
      errorStatus = COMMIT_FAILEDED;
      return errorStatus;
            }
      }
      else {
 toptions &= ~VX_UNBREAKABLE;
            if (taskOptionsSet (taskId, VX_UNBREAKABLE, toptions) == ERROR)
              {
            errorStatus = COMMIT_FAILEDED;
            return errorStatus;
              }
      }
      return errorStatus;
}
```

taskOptionsGet()

函数原型：

```
      STATUS taskOptionsGet
      (
      int tid,                  /* 任务 ID */
      int * pOptions            /* 存放任务的选项 */
      )
```

功能描述：

获得任务选项。用户通过调用该函数获得指定任务的当前执行选项。该函数返回的选项位说明如下：

VX_FP_TASK：浮点协处理器支持；

VX_PRIVATE_ENV：私有环境支持(查阅 envLib 库)；

VX_NO_STACK_FILL：不调用 checkstack()函数填充堆栈；

VX_UNBREAKABLE：禁止断点调试。

有关上述定义请查阅头文件 taskLib.h。

返回值：

成功获得选项则返回 OK，如果是无效的任务 ID 则返回 ERROR。

参考：

相关信息请参考 taskInfo 库描述和 taskOptionsSet()函数。

taskRegsGet()

函数原型：

```
STATUS taskRegsGet
(
int tid,                    /* 任务 ID */
REG_SET * pRegs             /* 存放任务的寄存器内容 */
)
```

功能描述：

从任务控制块(TCB)中获得任务的寄存器内容。该函数采集保存在 TCB 中的任务内容，并复制任务的寄存器内容到寄存器结构 pRegs 参数中。

注意：

该函数只有在任务处于已知状态而非运行态时才正常工作。

返回值：

成功获得任务的寄存器内容则返回 OK，如果是无效任务 ID 则返回 ERROR。

参考：

相关信息请参考 taskInfo 库描述、taskSuspend()和 taskRegsSet()函数。

taskRegsSet()

函数原型：

```
STATUS taskRegsSet
(
int tid,                    /* 任务 ID */
REG_SET * pRegs             /* 存放设置任务寄存器所需的内容 */
)
```

功能描述：

设置任务的寄存器内容。该函数装载指定的寄存器设置 pRegs 到指定的任务控制块（TCB）中。

注意：

该函数只有在任务处于已知状态而非就绪状态时才正常工作。推荐用户在改变寄存器设置之前对任务进行挂起操作。

返回值：

成功设置任务的寄存器内容则返回 OK，如果是无效任务 ID 则返回 ERROR。

参考：

相关信息请参考 taskInfo 库描述、taskSuspend()和 taskRegsGet()函数。

taskName()

函数原型：

```
char *taskName
(
int tid              /* 需要查找任务名的任务 ID */
)
```

功能描述：

根据任务 ID 获得任务名。如果任务有任务名，则该函数返回一个指向指定任务的任务名指针。如果任务没有任务名，则返回一个空字符串。

返回值：

有任务名则返回一个指向任务名的指针，如果输入无效任务 ID 则返回 NULL。

参考：

相关信息请参考 taskInfo 库描述。

taskNameToId()

函数原型：

```
int taskNameToId
(
char * name        /* 任务名 */
)
```

功能描述：

根据任务名查找任务 ID。该函数返回与指定名称相匹配的任务 ID。通过这种方式提取任务的操作效率较低，因为这个操作执行任务列表的查找。

返回值：

成功查找到任务则返回任务 ID，如果没有发现任务则返回 ERROR。

错误码：

S_taskLib_NAME_NOT_FOUND。

参考：

相关信息请参考 taskInfo 库描述。

例：

```
int                taskId;                /* 任务 ID */
...
/* 查找任务名为“tDemo”的任务 */
taskId = taskNameToId("tDemo");

if(taskId == ERROR)
    printf("Don't find tDemo!\n");
...
```

taskIdDefault()

函数原型：

```
int taskIdDefault
(
```

```
int tid        /* 用户提供的任务 ID；如果为 0，则返回默认 ID。*/
)
```

功能描述：

设置默认任务 ID。该函数维护一个全局的默认任务 ID。当系统没有明确地提供任务 ID 时，系统把默认任务 ID 赋予任务 ID 变量。

如果参数 tid 非 0(例如，用户指定一个任务 ID)，设置默认 ID 为这个值，并返回该值。如果 tid 为 0(例如，用户不指定一个任务 ID)，系统不改变默认 ID 值并返回缺省值。因此，返回的值总是用户指定的最后一次任务 ID。

返回值：

最近的非 0 任务 ID。

参考：

相关信息请参考 taskInfo 库描述、dbgLib 库描述、“VxWorks Programmer's Guide”的目标机 Shell、windsh 章节，以及“Tornado User's Guide”的 Shell 章节。

taskIsReady()

函数原型：

```
BOOL taskIsReady
(
int tid        /* 任务 ID */
)
```

功能描述：

检查任务是否处于就绪状态。该函数检查任务状态字段，确定任务是否处于就绪状态。

返回值：

任务处于就绪状态则返回 TRUE，其他状态则返回 FALSE。

参考：

相关信息请参考 taskInfo 库描述。

例：

```
int          taskId;          /* 任务 ID */
BOOL         status;
...
/* 查找任务名为“tDemo”的任务 */
taskId = taskNameToId("tDemo");
if(taskId == ERROR)
    printf("Don't find tDemo!\n");
else
 {
 /* 检查任务 tDemo 是否处于就绪状态 */
    status = taskIsReady(taskId);

 if(status == TRUE)
     printf("tDemo is ready status!\n");
    }
...
```

taskIsSuspended()

函数原型：

```
BOOL taskIsSuspended
(
int tid        /* 任务 ID */
)
```

功能描述：

检查任务是否处于挂起状态。该函数检查任务状态字段，确定任务是否处于挂起状态。

返回值：

任务处于挂起状态则返回 TRUE，其他状态则返回 FALSE。

参考：

相关信息请参考 taskInfo 库描述。

taskIdListGet()

函数原型：

```
int taskIdListGet
(
int idList[],          /* 任务 ID 数组 */
int maxTasks           /* idList[]能容纳的最大任务数 */
)
```

功能描述：

获得活动任务的 ID 列表。该函数提供所有活动任务列表给调用任务。一个不大于最大任务数 maxTasks 的未分类整理的任务 ID 列表被存放在 idList 数组中。

返回值：

ID 列表中存放的任务数。

参考：

相关信息请参考 taskInfo 库描述。

1.3 任务显示函数

1.3.1 函数库描述

1．库命名

任务显示函数库名称为 taskShow。

2．函数

表 1-3 中列出了任务显示函数。

表 1-3 任务显示函数

函 数	描 述	函 数	描 述
taskShowInit()	初始化任务显示函数模块	taskRegsShow()	显示任务的寄存器内容
taskInfoGet()	获得任务信息	taskStatusString()	获得任务状态字
taskShow()	显示任务控制块（TCB）信息		

3．描述

在应用开发期间，taskShow 库是辅助调试和用户界面的核心部分。用户通过调用 taskInfoGet()、taskShow()、taskRegsShow()和 taskStatusString()函数可以显示任务信息。系统必须在调用 taskShowInit()函数之后，才可以调用这些函数。

使用任务信息函数的主要缺点是在信息采集与信息显示之间，任务可能已经改变状态，所以这些函数提供的信息应该被认为是系统某个时刻的快照，而不可以信赖。除非任务被托管给一个已知状态，例如挂起状态。

任务管理和控制函数由 taskLib 库提供。访问任务信息和调试特性函数由 taskInfo 库提供。

4．头文件

任务显示函数声明在 taskLib.h 头文件中。

5．参考

相关信息请参考 taskLib、taskInfo、taskHookLib、taskVarLib、semLib 和 kernelLib 库描述，“VxWorks Programmer's Guide”中的基本操作系统、目标机 Shell 章节，以及“Tornado User's Guide”的 Shell 章节。

1.3.2　任务显示函数详细描述

taskShowInit()

函数原型：

```
void taskShowInit (void)
```

功能描述：

初始化任务显示函数模块。该函数把任务显示函数模块链接到 VxWorks 系统中。当用户通过下面的方式配置显示模块时，系统将自动调用该函数：

- 用户使用配置头文件，在 config.h 文件中定义 **INCLUDE_SHOW_ROUTINES** 即可；
- 用户使用 Tornado 工程配置工具，选中 **INCLUDE_TASK_SHOW** 即可。

返回值：

无。

参考：

相关信息请参考 taskShow 库描述。

taskInfoGet()

函数原型：

```
STATUS taskInfoGet
(
int tid,                    /* 任务 ID */
TASK_DESC * pTaskDesc   /* 用来填充的任务描述符 */
)
```

功能描述：

获得任务的信息。该函数把指定的任务信息填充在指定的任务描述符里。实际上任务描述符所包含的信息是任务控制块(WIND_TCB)中大部分信息的一个复制。TASK_DESC 结构对通用信息而言非常有用，避免系统直接处理难以操作的 WIND_TCB。

注意：

应当限制 WIND_TCB 的检查，通常情况仅用来调试。

返回值：

成功获得信息则返回 OK，如果任务 ID 无效则返回 ERROR。

参考：

相关信息请参考 taskShow 库描述。

taskShow()

函数原型：

```
STATUS taskShow
(
int tid,        /* 任务 ID */
int level       /* 0 = 摘要, 1 = 详细信息, 2 = 所有任务的摘要*/
)
```

功能描述：

显示任务控制块（TCB）中的信息。该函数显示指定 TCB 内容。如果 level 为 0，显示任务 ID 的摘要。如果 level 为 1，它同时也显示任务选项字和寄存器的值。如果 level 为 2，它显示所有任务的摘要。

该函数显示表 1-4 中所说明的 TCB 域中的内容。

表 1-4 TCB 域说明

域	说 明	域	说 明
NAME	任务名	PC	程序计数器
ENTRY	任务入口处符号名或地址	SP	堆栈指针
TID	任务 ID	ERROR	任务的最近错误代码
PRI	优先级	DELAY	如果任务处于延迟状态，保留在 delay 中时钟 tick 数（0 除外）
STATUS	任务状态，taskStatusString()形成的格式		

返回值：

无。

参考：

相关信息请参考 taskShow 库描述、taskStatusString()函数，“VxWorks Programmer's Guide”中的基本 windSh、目标机 Shell 章节，以及“Tornado User's Guide”的 Shell 章节。

操作示范：

在宿主机 WindSh 工具中，taskShow()函数操作示范如下所示：

```
-> taskShow tWdbTask,0
  NAME        ENTRY       TID         PRI   STATUS      PC          S           ERRNO     DELAY
---------- ---------- ---------- ----- ---------- ---------- ---------- ---------- ----------
tWdbTask    0x417cc4    50f0c08    3     READY      4276be     50f0ae4     d0003     0

-> taskShow tWdbTask,1
  NAME        ENTRY       TID         PRI   STATUS      PC          SP          ERRNO     DELAY
---------- ---------- ---------- ----- ---------- ---------- ---------- ---------- ----------
tWdbTask    0x417cc4    50f0c08    3     READY      4276be     50f0ae4     d0003     0
stack: base 0x50f0c08   end 0x50eccc8   size 16176   high 3840   margin 12336   options: 0xe
VX_UNBREAKABLE VX_DEALLOC_STACK VX_FP_TASK
edi    =  ffffffff      esi  =  50f0fb8     ebp  =  50f0aec     esp  =  50f0ae4
ebx    =  0             edx  =  4276be      ecx  =  10101       eax  =  0
eflags =  212           pc   =  4276be
```

```
-> taskShow tWdbTask,2
  NAME      ENTRY       TID      PRI   STATUS      PC          SP          ERRNO     DELAY
---------- ------------ ---------- ----- ---------- ---------- ---------- ---------- ----------
tExcTask    _excTask    50f9170   0     PEND      4276be      50f908c     d0003     0
tLogTask    _logTask    50f4758   0     PEND      4276be      50f4670     d0003     0
tWdbTask    0x417cc4    50f0c08   3     READY     4276be      50f0ae4     d0003     0
```

taskRegsShow()

函数原型：

```
void taskRegsShow
(
int tid        /* 任务 ID */
)
```

功能描述：

显示任务的寄存器内容。该函数在标准输出上显示指定任务的寄存器内容。

返回值：

无。

参考：

相关信息请参考 taskShow 库描述。

操作示范：

在宿主机 WindSh 工具中，taskRegsShow()函数操作示范如下所示：

```
-> taskRegsShow(tHighPri)
edi    =   0        esi   =   500e7d0     ebp   =   500e72c     esp   =   500e724
ebx    =   0        edx   =   4276be      ecx   =   10101       eax   =   fffffffe
eflags =   206      pc    =   4276be
```

taskStatusString()

函数原型：

```
STATUS taskStatusString
(
int tid,              /* 任务 ID */
char * pString        /* 返回的任务状态字 */
)
```

功能描述：

获得任务的状态字。该函数解释指定的任务在 TCB 中的 WIND 任务状态字，并把它复制到 pString 中。表 1-5 对任务的状态字进行说明。

返回值：

成功获得任务状态字返回 OK，如果输入无效任务 ID 则返回 ERROR。

参考：

相关信息请参考 taskShow 库描述。

操作示范：

在宿主机 WindSh 工具中，taskStatusString()函数操作示范如下所示：

```
-> taskStatusString tWdbTask,status=malloc(10)
new symbol "status" added to symbol table.
```

```
value = 0 = 0x0
-> printf("task status is:<%s>\n",status)
task status is:<PEND>
```

表 1-5 任务状态字说明

状 态 字	说 明	状 态 字	说 明
READY	除 CPU 之外，任务不再等待任何其他资源	PEND+S	任务处于阻塞和挂起的状态
PEND	任务由于某个不可用的资源而阻塞	PEND+T	任务处于带有超时值且阻塞的状态
DELAY	任务在等待一定时间的延时	PEND+S+T	任务处于带有超时值、阻塞且又挂起的状态
SUSPEND	任务得不到执行（非暂停、延迟或阻塞），处于被挂起状态	...+I	任务带有继承的优先级且处于上述任何状态(...表示上述任何状态)
DELAY+S	任务处于延迟和挂起的状态	DEAD	任务不存在

1.4 任务钩子管理函数

1.4.1 函数库描述

1．库命名

任务钩子(hook)函数库名称为 taskHookLib。

2．函数

表 1-6 中列出了任务钩子管理函数。

表 1-6 任务钩子管理函数

函 数	描 述	函 数	描 述
taskHookInit()	初始化任务钩子模块	taskSwitchHookDelete()	删除任务切换钩子程序
taskCreateHookAdd()	添加任务创建钩子程序	taskDeleteHookAdd()	添加任务删除钩子程序
taskCreateHookDelete()	删除任务创建钩子程序	taskDeleteHookDelete()	删除任务删除钩子程序
taskSwitchHookAdd()	添加任务切换钩子程序		

3．描述

taskHookLib 库提供了扩展 VxWorks 任务模块的函数。为了把与任务有关的功能增加到系统中而又不修改内核，内核提供了一些钩子管理函数。通过这些钩子管理函数可以把程序或“钩子”添加到系统中，在任务创建、切换或删除时调用它们。钩子管理函数允许动态地添加和删除钩子。

dbgLib 库使用这个功能是为了提供任务相关的断点和单步操作。为了支持“任务变量”机制，taskVarLib 库也使用该功能。同时，为了支持浮点协处理器，fppLib 库也使用该功能。

4．注意：

在任务钩子函数中可能有从属关系。例如，一个删除钩子函数可能要使用某个资源，而这个资源却已经被另一个删除钩子函数清除或删除。在这种情况下，钩子函数的运行顺序是非常重要的。VxWorks 是按添加顺序运行创建和切换钩子函数，而删除钩子函数的运行顺序与添加时的顺序恰好相反。

5．头文件

任务钩子管理函数声明在 taskHookLib.h 文件中。

6．参考

相关信息请参考 dbgLib、fppLib、taskLib、taskVarLib 库描述，“VxWorks Programmer's Guide”中的基

本操作系统章节。

1.4.2　任务钩子管理函数详细描述

taskHookInit()

函数原型：

```
void taskHookInit (void)
```

功能描述：

初始化任务钩子模块。该函数配置任务钩子包到 VxWorks 系统中。如果定义了配置宏 INCLUDE_TASK_HOOKS，系统将自动调用该函数。系统必须在调用 taskHookInit()函数之后，才可以调用其他的钩子管理函数。

返回值：

无。

参考：

相关信息请参考 taskHookLib 库描述。

taskCreateHookAdd()

函数原型：

```
STATUS taskCreateHookAdd
(
FUNCPTR createHook   /* 在任务创建时调用该函数 */
)
```

功能描述：

添加任务创建钩子程序。该函数添加指定的函数到函数列表中，无论系统何时创建任务，系统将首先调用添加的函数。被添加的函数应声明：

```
void createHook
(
WIND_TCB *pNewTcb /* 指向新任务 TCB 的指针 */
)
```

返回值：

成功添加任务创建钩子程序则返回 OK，如果任务创建钩子函数表已满则返回 ERROR。

参考：

相关信息请参考 taskHookLib 库描述、taskCreateHookDelete()函数。

例：

```
/* 安装 WDB task 创建钩子 */
static STATUS vxTaskCreateHookAdd( void (*hook)() )
    {
    static BOOL initialized = FALSE;
    wdbCreateHook = hook;

    if ((hook == NULL) && initialized)
{
    /* 卸载任务创建钩子 */
    taskCreateHookDelete (__wdbTaskCreateHook);
```

```
initialized = FALSE;
}
    else if (!initialized)
{
    /* 安装任务创建钩子 */
    taskCreateHookAdd (__wdbTaskCreateHook);
initialized = TRUE;
}
    return (OK);
    }

/* __wdbTaskCreateHook - 任务创建钩子函数 */
static int __wdbTaskCreateHook(WIND_TCB * pTcb)
    {
    WDB_CTX  createdCtx;
    WDB_CTX  creationCtx;

    if (wdbCreateHook != NULL)
{
/* 填充 createdCtx 结构 */
createdCtx.contextType = WDB_CTX_TASK;
createdCtx.contextId   = (UINT32)pTcb;

/* 填充 creationCtx 结构 */
creationCtx.contextType      = WDB_CTX_TASK;
creationCtx.contextId   = (UINT32)taskIdCurrent;

(*wdbCreateHook) (&createdCtx, &creationCtx);
}
    return (OK);
    }
```

taskCreateHookDelete()

函数原型：

```
    STATUS taskCreateHookDelete
    (
    FUNCPTR createHook        /* 从任务创建钩子函数列表中将该函数删除 */
    )
```

功能描述：

删除任务创建钩子函数。该函数从任务创建钩子函数列表中卸载指定的函数。

返回值：

成功卸载则返回 OK，如果任务创建钩子函数列表中无指定的函数则返回 ERROR。

参考：

相关信息请参考 taskHookLib 库描述、taskCreateHookAdd()函数。

taskSwitchHookAdd()

函数原型：

```
STATUS taskSwitchHookAdd
(
FUNCPTR switchHook      /* 在任务切换时调用该函数 */
)
```

功能描述：

添加任务切换钩子函数。该函数添加指定的函数到函数列表中，当任务发生切换时，系统将首先调用添加的函数。被添加的函数应声明：

```
void switchHook
(
WIND_TCB *pOldTcb,    /* 指向先前任务 WIND_TCB 的指针 */
WIND_TCB *pNewTcb     /* 指向新任务 WIND_TCB 的指针 */
)
```

注意：

在内核上下文中调用用户安装的任务切换钩子函数。因此，切换钩子函数无权访问所有的 VxWorks 模块。表 1-7 列出了切换钩子函数可调用的函数。

表 1-7　任务切换钩子函数可调用的函数

库	函　数	库	函　数
bLib	所有的函数	mathALib	如果调用了 fppSave()/fppRestore()，所有的函数可被调用
fppArchLib	fppSave()、fppRestore()	rngLib	除 rngCreate()函数之外的所有函数
intLib	intContext()、intCount()、intVecSet()、intVecGet()	taskLib	taskIdVerify()、taskIdDefault()、taskIsReady()、taskIsSuspended()、taskTcb()
lstLib	所有的函数	vxLib	vxTas()

返回值：

成功添加任务切换钩子函数则返回 OK，如果任务切换钩子函数列表已满则返回 ERROR。

参考：

相关信息请参考 taskHookLib 库描述、taskSwitchHookDelete()函数。

taskSwitchHookDelete()

函数原型：

```
STATUS taskSwitchHookDelete
(
FUNCPTR switchHook      /* 从任务切换钩子函数列表中将该函数删除 */
)
```

功能描述：

删除任务切换钩子函数。该函数从任务切换钩子函数列表中卸载指定的函数。

返回值：

成功卸载则返回 OK，如果任务切换钩子函数列表中无指定的函数则返回 ERROR。

参考：

相关信息请参考 taskHookLib 库描述及 taskSwitchHookAdd()函数。

taskDeleteHookAdd()

函数原型：

```
STATUS taskDeleteHookAdd
(
FUNCPTR deleteHook        /* 在任务被删除时该函数被调用 */
)
```

功能描述：

添加任务删除钩子函数。该函数添加指定的函数到函数列表中，无论系统何时删除任务，系统将首先调用添加的钩子函数。被添加的函数应声明：

```
void deleteHook
(
WIND_TCB *pTcb     /* 指向删除任务 WIND_TCB 的指针 */
)
```

返回值：

成功添加任务删除钩子函数则返回 OK，如果任务删除钩子函数列表已满则返回 ERROR。

参考：

相关信息请参考 taskHookLib 库描述及 taskDeleteHookDelete()函数。

taskDeleteHookDelete()

函数原型：

```
STATUS taskDeleteHookDelete
(
FUNCPTR deleteHook        /* 从任务删除钩子函数列表中将该函数删除 */
)
```

功能描述：

删除任务删除钩子函数。该函数从任务删除钩子函数列表中卸载指定的函数。

返回值：

成功卸载则返回 OK，如果任务删除钩子函数列表中无指定的函数则返回 ERROR。

参考：

相关信息请参考 taskHookLib 库描述及 taskDeleteHookAdd()函数。

1.5 任务钩子显示函数

1.5.1 函数库描述

1. 库命名

任务钩子显示函数库名称为 taskHookShow。

2. 函数

表 1-8 中列出了任务钩子显示函数。

3. 描述

taskHookShow 库提供的函数功能是总结已被安装到内核中的钩子函数。每一种类型的内核钩子（任务创

建、任务切换和任务删除）都有专用的显示函数。

表 1-8　任务钩子显示函数

函　数	描　述	函　数	描　述
taskHookShowInit()	初始化任务钩子显示模块	taskSwitchHookShow()	显示任务切换钩子函数列表
taskCreateHookShow()	显示任务创建钩子函数列表	taskDeleteHookShow()	显示任务删除钩子函数列表

系统必须在调用了 taskHookShowInit()函数之后，才可以调用其他钩子显示函数。

4．头文件

任务钩子显示函数声明在 taskHookLib.h 头文件中。

5．参考

相关信息请参考 taskHookLib 库描述，“VxWorks Programmer's Guide”中的基本操作系统章节。

1.5.2　任务钩子显示函数详细描述

taskHookShowInit()

函数原型：

```
void taskHookShowInit (void)
```

功能描述：

初始化任务钩子显示模块。该函数把任务钩子显示模块链接到 VxWorks 系统中。当用户通过下面的方式配置任务钩子显示模块时，系统将自动调用该函数：

- 用户使用配置头文件，则在 config.h 头文件中定义 **INCLUDE_SHOW_ROUTINES** 即可；
- 用户使用 Tornado 工程配置工具，选中 **INCLUDE_TASK_HOOK_SHOW** 即可。

返回值：

无。

参考：

相关信息请参考 taskHookShow 库描述。

taskCreateHookShow()

函数原型：

```
void taskCreateHookShow (void)
```

功能描述：

显示任务创建钩子函数列表。该函数显示所有安装到任务创建钩子列表中的任务创建钩子函数。表中函数顺序按添加先后次序排列。

返回值：

无。

参考：

相关信息请参考 taskHookShow 库描述及 taskCreateHookAdd()函数。

例：

```
/* 钩子函数显示程序 */
void   HookShow(void)
{
/* 显示任务创建钩子函数列表 */
taskCreateHookShow();
```

```
/* 显示任务切换钩子函数列表 */
taskSwitchHookShow();

/* 显示任务删除钩子函数列表 */
taskDeleteHookShow();
}
```

taskSwitchHookShow()

函数原型：

void taskSwitchHookShow (void)

功能描述：

显示任务切换钩子函数列表。该函数显示所有安装到任务切换钩子列表中的任务切换钩子函数。表中函数顺序按添加先后次序排列。

返回值：

无。

参考：

相关信息请参考 taskHookShow 库描述及 taskSwitchHookAdd()函数。

taskDeleteHookShow()

函数原型：

void taskDeleteHookShow (void)

功能描述：

显示任务删除钩子函数列表。该函数显示所有安装到任务删除钩子列表中的任务删除钩子函数。表中函数顺序按添加先后次序排列。

返回值：

无。

参考：

相关信息请参考 taskHookShow 库描述及 taskDeleteHookAdd()函数。

1.6 任务变量函数

1.6.1 函数库描述

1．库命名

任务变量函数库名称为 taskVarLib。

2．函数

表 1-9 中列出了任务变量函数。

表 1-9 任务变量函数

函 数	描 述	函 数	描 述
taskVarInit()	初始化任务变量模块	taskVarGet()	获得任务变量值
taskVarAdd()	把任务变量添加到任务中	taskVarSet()	设置任务变量值
taskVarDelete()	从任务中删除任务变量	taskVarInfo()	获得任务的任务变量列表

3．描述

VxWorks 提供了一个被称为“任务变量”的模块，它允许把 4 个字节的变量添加到任务上下文中，该变量值将会在任务切换时发生切换，也可以通过任务自身来修改。典型情况是几个任务声明同一个变量作为任务变量，并视其为自己的私有变量在内存区中对它进行处理。例如，当一个函数必须被多次创建并激活、并视为几个同时存在的任务时，系统可以使用该模块来实现这个目的。

用户可以使用 taskVarAdd()和 taskVarDelete()函数来添加或删除任务变量。使用 taskVarGet()和 taskVarSet()函数来获得或设置任务变量值。

4．注意：

如果在任务删除钩子中使用任务变量，请阅读有关正确使用 taskVarInit()函数的警告部分。

5．头文件

任务变量函数声明在 taskVarLib.h 头文件中。

6．参考

相关信息请参考 taskVarLib 库描述及“VxWorks Programmer's Guide”中的基本操作系统章节。

1.6.2　任务变量函数详细描述

taskVarInit()

函数原型：

```
STATUS taskVarInit (void)
```

功能描述：

初始化任务变量模块。它安装任务切换和删除钩子所要使用的任务变量。如果没有明确地调用函数 taskVarInit()，当添加第一个任务变量时，系统将自动调用 taskVarInit()函数。

在该函数的第一次调用之后，后来的调用视为无效。

警告：

在 VxWorks 系统中运行删除钩子要非常小心，其运行顺序与添加顺序恰好相反，采取“先进后出”的方式进行。任何拥有删除钩子的模块将使用任务变量以保证在添加自己的删除钩子之前按正确次序调用 taskVarInit()函数。

在创建钩子中添加任务变量时，用户也应当小心操作。如果任务变量包还没有通过 taskVarInit()函数来安装它，而创建钩子企图创建一个钩子，这将会导致系统失败。为了避免这种情况，在 usrRoot()根任务(在 usrCofig.c 中)进行系统初始化期间应当调用 taskVarInit()函数。

返回值：

成功初始化任务变量模块则返回 OK，如果不可以安装任务切换/删除钩子则返回 ERROR。

参考：

相关信息请参考 taskVarLib 库描述。

taskVarAdd()

函数原型：

```
STATUS taskVarAdd
(
int tid,            /* 任务的 ID */
int * pVar          /* 一个指向变量的指针 */
)
```

功能描述：

把任务变量添加到任务中。该函数把一个指定的变量 pvar(占 4 个字节内存区)添加到一个指定任务的上

下文中。调用该函数后，任务把该变量作为自己的私有变量。任务可以访问和修改这个变量，但修改操作不会影响其他任务，并且其他任务针对该变量的修改也不会影响这个任务所“看到”的值。每次任务发生切换或任务自身改变该值时，系统将为其他任务保存和恢复变量的初始值。

当一个函数被重复地创建、激活并视为几个相互独立的任务时，系统可以使用该功能来实现这个目的。尽管每一个任务拥有自己的堆栈和独立的堆栈变量，但他们将完全共享全局变量。为了使一个变量不被共享，函数可以调用 taskVarAdd()为每一个任务分别复制该变量，但所有复制都在同一个物理地址。

值得注意的是任务变量增加了任务切换时间。这样，需要合理地限制任务使用任务变量的个数。有效的方法是任务变量是一个指针变量，它动态地分配结构来存放任务的私有数据。

返回值：

成功添加任务变量则返回 OK，如果内存不能满足任务变量描述符的需求则返回 ERROR。

参考：

相关信息请参考 taskVarLib 库描述、taskVarDelete()、taskVarGet()和 taskVarSet()函数。

例：

```
int         taskId;                 /* 任务 ID */
int         status;
int         taskVar;
...
/* 查找任务名为“tDemo”的任务 */
taskId = taskNameToId("tDemo");
...
if(taskId == ERROR)
    {
      printf("Don't find tDemo!\n");
      return(ERROR);
      }
else
      {

 /* 把任务变量添加到任务中 */
      status = taskVarAdd(taskId, &taskVar);
...
```

taskVarDelete()

函数原型：

```
      STATUS taskVarDelete
      (
      int tid,          /* 任务的 ID */
      int * pVar        /* 一个指向变量的指针 */
      )
```

功能描述：

从任务中删除任务变量。该函数从指定任务的上下文中删除指定的任务变量 pvar。这个变量的私有值也随之丢弃。

返回值：

成功删除任务变量则返回 OK，如果指定的任务中不存在指定的变量则返回 ERROR。

参考：

相关信息请参考 taskVarLib 库描述、taskVarAdd()、taskVarGet()和 taskVarSet()函数。

taskVarGet()

函数原型：

```
int taskVarGet
(
int tid,            /* 任务的 ID */
int * pVar          /* 一个指向任务变量的指针*/
)
```

功能描述：

获得任务变量值。该函数从指定的任务中返回任务变量的私有值。这个指定的任务通常不是调用该函数的任务，它可以直接访问变量而获得变量的私有值。

返回值：

成功获得则返回任务变量的私有值，如果没有找到任务或任务没有这个变量则返回 ERROR。

参考：

相关信息请参考 taskVarLib 库描述、taskVarAdd()、taskVarDelete()和 taskVarSet()函数。

taskVarSet()

函数原型：

```
STATUS taskVarSet
(
int tid,            /* 任务的 ID */
int * pVar,         /* 一个指向变量的指针 */
int value           /* 新的任务变量值 */
)
```

功能描述：

设置任务变量值。该函数为指定的任务设置任务变量的私有值。这个指定的任务通常不是调用该函数的任务，它可以直接修改变量并设置变量的私有值。该函数主要为调试服务。

返回值：

成功设置则返回 OK，如果没有找到任务或任务没有这个变量则返回 ERROR。

参考：

相关信息请参考 taskVarLib 库描述、taskVarAdd()、taskVarDelete()和 taskVarGet()函数。

例：

```
int         taskId;         /* 任务 ID */
int         taskVar;
int         status;
...
/* 设置任务变量的值 */
status = taskVarSet(taskId, &taskVar,100);
...
```

taskVarInfo()

函数原型：

```
int taskVarInfo
```

```
(
int tid,                    /* 任务的 ID */
TASK_VAR varList[],         /* 存放任务变量地址的数组 */
int maxVars                 /* 数组变量 varList[]的最大单元数 */
)
```

功能描述：

获得任务的任务变量列表。该函数提供给调用任务一个指定任务的任务变量列表。未分类整理的任务变量数组被复制到 varList[]中。

警告：

当查找任务变量时，系统调用 taskLock()函数屏蔽内核调度。这样，不能保证所有任务变量依然有效，或 taskVarInfo()函数返回时新任务变量还没有创建。

返回值：

任务变量列表中任务变量数。

参考：

相关信息请参考 taskVarLib 库描述。

操作示范：

在宿主机 WindSh 工具中，taskVarInfo()等函数操作示范如下：

```
-> taskVarAdd tLowPri,taskVar=malloc(10)
new symbol "taskVar" added to symbol table.
value = 0 = 0x0
-> taskVarSet tLowPri,taskVar,100
value = 0 = 0x0
-> taskVarGet tLowPri,taskVar
value = 100 = 0x64 = 'd'
-> taskVarInfo tLowPri,varList=malloc(100),5
new symbol "varList" added to symbol table.
value = 1 = 0x1          /* 表明任务拥有 1 个私有变量 */
-> taskVarAdd tLowPri,taskVar1=malloc(10)
new symbol "taskVar1" added to symbol table.
value = 0 = 0x0
-> taskVarInfo tLowPri,varList=malloc(100),5
value = 2 = 0x2          /* 表明任务拥有 2 个私有变量 */
-> taskVarDelete tLowPri,taskVar1
value = 0 = 0x0
```

1.7 体系结构相关任务管理函数

1.7.1 函数库描述

1. 库命名

体系结构相关任务管理函数库名称为 taskArchLib。

2. 函数

表 1-10 中列出了体系结构相关的任务管理函数。

表 1-10　体系结构相关的任务管理函数

函　数	描　述
taskSRSet()	设置任务状态寄存器（MC680x0、MIPS、i386/i486）
taskSRInit()	初始化任务状态寄存器的默认值（MIPS）

3．描述

taskArchLib 库提供了体系结构相关的任务管理函数，以便设置和检验与体系结构有关的寄存器。与体系结构无关的任务管理函数的详细信息请阅读 taskLib 函数库。

4．注意：

对 SPARC 体系结构而言，taskArchLib 库不为该体系结构提供应用层函数。

5．头文件

体系结构相关任务管理函数声明在 taskArchLib.h 头文件中。

6．参考

相关信息请参考 taskLib 库描述。

1.7.2　体系结构相关的任务管理函数详细描述

taskSRSet()

函数原型：

```
STATUS taskSRSet
(
int tid,            /* 任务 ID */
UINT16 sr           /* 状态寄存器新的值 */
)
```

功能描述：

设置任务状态寄存器。该函数设置非运行状态任务(例如，必须不是调用任务的 TCB)的状态寄存器。调试工具使用该函数设置任务状态寄存器的跟踪位，从而使任务处于单步执行状态。

返回值：

成功设置则返回 OK，如果任务 ID 无效则返回 ERROR。

参考：

相关信息请参考 taskArchLib 库描述。

例：

```
int         taskId;         /* 任务 ID */
int         taskVar;
int         status;
...
/* 设置任务状态寄存器 */
status = taskSRSet(taskId, 100);
if(status ==ERROR)
printf("set the task status register failed!\n");
...
```

taskSRInit()

函数原型：

```
ULONG taskSRInit
```

```
(
ULONG newSRValue /* 任务状态寄存器的默认值 */
)
```

功能描述：

初始化任务状态寄存器的默认值。该函数设置任务状态寄存器的默认值。把所有已经创建任务的状态寄存器设置成该值；这样，系统必须在 kernelInit()函数之前调用该函数。

返回值：

状态寄存器先前的值。

参考：

相关信息请参考 taskArchLib 库描述。

例：

```
#define DEFAULT_SR (SR_CU1 | SR_CU0 | (SR_IMASK0 & ~INT_LVL_FPA) | SR_IEC)
...
void sysHwInit (void)
    {
    /* 设置任务状态寄存器的默认值 */
    taskSRInit (DEFAULT_SR);
    ...
    }
```

第 2 章　任务间同步与通信

2.1　信号量

2.1.1　二值信号量

2.1.1.1　函数库描述

1．库命名

二值信号量函数库名称为 semBLib。

2．函数

表 2-1 中列出了二值信号量函数。

表 2-1　二值信号量函数

函　数	描　述
semBCreate()	创建并初始化一个二值信号量

3．描述

semBLib 库为 VxWorks 的二值信号量提供接口。二值信号量是最通用、最有效、最简单类型的信号量。它们可以被用于：对共享设备或数据结构的互斥访问控制，多任务的同步，或任务级和中断级处理。二值信号量是 VxWorks 功能的基本组成部分。

一个二值信号量可以被看成是内存中的一个单元，它的内容是两种状态中的一种，即空(empty)或者满(full)。当任务使用 semTake()函数获得一个二值信号量时，后来的行为依赖信号量的状态：

（1）如果信号量先前为满，则使信号量现在为空，并且调用任务继续执行。

（2）如果信号量先前为空，则任务将被阻塞，直到信号量可用。如果指定了超时并超时截止，被阻塞的任务将从阻塞任务队列中撤走，并随同一个 ERROR 状态进入就绪状态。一个被阻塞的任务无资格分配 CPU 资源。任何数量的任务可能同时被阻塞在同一个二值信号量上。

当任务使用 semGive()函数释放一个二值信号量时，存在阻塞队列中的第一个任务被解除阻塞。如果没有任务阻塞在这个信号量上，信号量则变为满。

4．注意

如果一个信号量被释放，被解除阻塞的任务的优先级比调用 semGive()函数的任务优先级高，则被解除阻塞的任务将抢占正在执行的任务。

5．互斥操作

使用一个二值信号量作为互斥的一种手段，首先用初始状态为满这种情况来创建它。例如：

```
SEM_ID semMutex;
/* 创建一个二值信号量，其初始状态为满。*/
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

然后，通过使用 semTake()获得信号量来保护临界区或临界资源，而退出临界区或免除临界资源则通过使用 semGive()释放信号量来实现。例如：

```
semTake (semMutex, WAIT_FOREVER);
... /* 临界区，每次仅有一个任务可以访问。*/
semGive (semMutex);
```

当多个任务对同一信号量的释放、获得或刷新没有限制时，确保互斥构造的特有功能是非常重要的。当任何次数的获得信号量操作都不会发生危险时，应当非常谨慎地控制信号量释放操作。如果信号量释放操作是通过一个没有对该信号量进行获得操作的任务来进行，则信号量丧失了互斥操作功能。

6．同步操作

使用二值信号量作为同步的一种手段，首先用初始状态为空这种情况来创建它。任务由于获得信号量而阻塞在同步点上，它将保持阻塞直到另一个任务或中断服务程序释放这个信号量。

对于中断服务程序的同步有特殊的要求。可以在中断服务程序中释放二值信号量，但不可以执行获得信号量操作。这样，任务可以阻塞在同步点 semTake()上，而中断服务程序可以通过 semGive()解除任务的阻塞。

在下面的例子中，当调用 init()时，创建一个二值信号量，一个中断服务程序附在一个事件上，创建一个任务处理这个事件。任务将运行直到调用 semTake()，在这个点，任务将阻塞直到事件导致中断服务程序调用 semGive()。当中断服务程序结束，任务可以执行并处理事件。

例：

```
SEM_ID semSync; /* 同步信号量的 ID */
init ()
{
intConnect (..., eventInterruptSvcRout, ...);                    /* 注册中断 */
semSync = semBCreate (SEM_Q_FIFO, SEM_EMPTY);    /* 创建信号量 */
taskSpawn (..., task1);                                                  /* 创建任务 */
}
task1 ()
{
...
semTake (semSync, WAIT_FOREVER);                                /* 等待信号量 */
... /* 处理事件 */
}
eventInterruptSvcRout ()
{
...
semGive (semSync); /* 释放信号量，让任务 1 处理事件 */
...
}
```

semFlush()函数作用于二值信号量，将自动解锁这个信号量队列中所有被挂起的任务。

7．警告

当任务被挂起或删除时，没有机制来自动归还或回收信号量。因为对被保护资源或区域的状态没有明确的认识，而不计后果的自动收回信号量会将资源保持在一个局部状态里。这样，如果任务意外地停止执行，像总线出错，当前拥有的信号量将不会归还给系统，实际上是再也不能使用保留资源了。semMLib 库提供的互斥信号量对意外的任务删除操作提供保护功能。

8．头文件

二值信号量函数声明在 semLib.h 头文件中。

9．参考

相关信息请参考 semLib、semCLib、semMLib 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统章节。

2.1.1.2 二值信号量函数详细描述

semBCreate()

函数原型：

```
SEM_ID semBCreate
(
```

```
    int options,                    /* 信号量选项字 */
    SEM_B_STATE initialState  /* 信号量初始状态 */
    )
```

功能描述：

创建并初始化一个二值信号量。该函数创建并初始化一个二值信号量。参数 initialState 是信号量的初始状态，其值是 **SEM_FULL**(1)或 **SEM_EMPTY**(0)。

参数 options 指定被阻塞任务的排列类型。任务可以基于优先级或先进先出原则进行排列。这些选项分别是 **SEM_Q_PRIORITY**(0x1)和 **SEM_Q_FIFO**(0x0)。

返回值：

成功创建信号量则返回信号量 ID，如果内存不足则返回 NULL。

参考：

相关信息请参考 semBLib 库描述。

例：

```
/* 数据定义 */
SEM_ID semId1;              /* 二值信号 ID */
...
/* 创建一个二值信号 */
semId1 = semBCreate (SEM_Q_PRIORITY, SEM_FULL);

if (semId1 == NULL)
        {
        perror ("synchronizeDemo: Error in creating semId1 binary semaphore");
        return (ERROR);
        }
...
```

2.1.2　计数信号量

2.1.2.1　函数库描述

1．库命名

计数信号量函数库名称为 semCLib。

2．函数

表 2-2 中列出了计数信号量函数。

表 2-2　计数信号量函数

函　数	描　述
semCCreate()	创建并初始化一个计数信号量

3．描述

semCLib 库为 VxWorks 计数信号量提供接口。当对一个资源进行多次请求操作时，采用计数信号量对这个资源被进行保护操作是非常有用的行为。

可以把一个计数信号量看成是内存中的一个单元，它的内容保留了一个计数记录。当任务使用 semTake()函数获得一个计数信号量时，后来的行为依赖计数状态：

（1）如果计数非 0，则计数减少，而调用任务继续执行。

（2）如果计数为 0，则任务将被阻塞，直到信号量可用。如果指定了超时且超时已经截止，则被阻塞的任务将从阻塞任务队列中撤走，并随同一个 ERROR 状态进入就绪状态。被阻塞的任务无资格分配 CPU 资源。任何数量的任务可能同时被阻塞在同一个计数信号量上。

当任务使用 semGive()函数释放信号量时，存在阻塞队列中的第一个任务被解除阻塞。如果没有任务被

阻塞在这个信号量上，则信号量计数增加。值得注意的是如果一个信号量被释放，被解除阻塞的任务其优先级比调用 semGive()函数的任务优先级高，则被解除阻塞的任务将抢占当前正在执行的任务。

semFlush()函数作用于一个计数信号量，将自动解锁这个信号量队列中所有被挂起的任务。因此所有的任务在执行前将处于就绪状态。不过信号量的计数将保持不变。

4．警告

当任务被挂起或删除时，没有机制来自动归还或回收信号量。因为对被保护资源或区域的状态没有明确的认识，而不计后果地自动收回信号量会将资源保持在一个局部状态里。这样，如果任务意外地停止执行，像总线出错，当前拥有的信号量将不会归还给系统，实际上是再也不能使用保留资源了。semMLib 库提供的互斥信号量对意外的任务删除提供保护功能。

5．头文件

计数信号量函数声明在 semLib.h 中。

6．参考

相关信息请参考 semLib、semCLib、semMLib 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统章节。

2.1.2.2 计数信号量函数详细描述

semCCreate()

函数原型：

```
SEM_ID semCCreate
(
int options,        /* 信号量选项字 */
int initialCount    /* 计数初始值 */
)
```

功能描述：

创建并初始化一个计数信号量。该函数分配并初始化一个计数信号量。

参数 initialCount 是信号量的初始计数值。参数 options 指定被阻塞的任务的排列类型。任务可以基于优先级或先进先出原则进行排列。这些选项分别是 **SEM_Q_PRIORITY**(0x1)和 **SEM_Q_FIFO**(0x0)。

返回值：

成功创建信号量则返回信号量 ID，如果内存不足则返回 NULL。

参考：

相关信息请参考 semCLib 库描述。

例：

```
/* 数据定义 */
SEM_ID semId = NULL;            /* 计数信号量 ID */
...
/* 创建一个计数信号量 */
semId = semCCreate (SEM_Q_PRIORITY, 5);
...
if (semId == NULL)
    {
      perror ("Error in creating counting semaphore");
      return (ERROR);
    }
...
```

2.1.3 互斥信号量

2.1.3.1 函数库描述

1. 库命名

互斥信号量函数库名称为 semMLib。

2. 函数

表 2-3 中列出了互斥信号量函数。

表 2-3 互斥信号量函数

函 数	描 述
semMCreate()	创建并初始化一个互斥信号量
semMGiveFore()	无条件释放一个互斥信号量

3. 描述

semMLib 库为 VxWorks 互斥信号量提供接口。互斥信号量对需要互斥访问的资源提供了便利的选择。典型的应用包括共享设备和保护数据结构。许多高级别的 VxWorks 模块使用互斥信号量功能。

互斥信号量是二值信号量的一个特殊版本，在互斥内部的设计考虑了地址的问题，例如递归访问资源、优先级反转和安全删除。互斥信号量的基本行为与二值信号量相同，但加入了下面的限制：

（1）只可以被用作互斥操作。

（2）只可以被获得它的任务释放。

（3）在中断服务程序中不可以对它进行获得或释放操作。

（4）使用 semFlush()函数对它进行操作是非法行为。

4. 递归资源访问

互斥信号量的一个特点是可以对它进行“递归”获得操作。例如，在最后释放互斥信号量之前，拥有它的任务可以对它进行多次获得操作。对一组程序需要互斥访问一个资源，采用递归式是有用的，但可能需要互相各自调用。

系统追踪任务当前自身拥有的互斥信号量可能会导致递归情况发生。在信号量释放前，对互斥信号量进行了多少次获得操作就必须进行多少次释放操作；这个过程的追踪是依靠一个计数来进行，调用 semTake()函数对它进行增加操作，而 semGive()则对它进行减少操作。

下例是互斥信号量递归举例。函数 funcA()需要获得 semM 才可以访问资源；函数 funcA()也需要调用函数 funcB()，而 funcB()也需要 semM：

例：

```
...
SEM_ID semM;
semM = semMCreate (...);
funcA ()
{
semTake (semM, WAIT_FOREVER);
...
funcB ();
...
semGive (semM);
}
funcB ()
{
semTake (semM, WAIT_FOREVER);
...
semGive (semM);
```

}

5．优先级反转保护

如果在创建互斥信号量时，选项字包含 **SEM_INVERSION_SAFE** 选项，库采用优先级继承协议来解决潜在的“优先级反转”事件。当一个高优先级任务为了等低优先级任务的完成而被迫等待不确定周期的时间时，便发生了优先级反转。

优先级反转是一种不定延期的现象，这种现象普遍存在于共享资源的多任务可抢占系统执行中。做一个假设：当一个高优先级的任务请求访问一个共享资源时，而此时这个资源被分配给一个低优先级的任务。这时高优先级的任务被阻塞，直到低优先级的任务释放这个资源。问题是当低优先级的任务执行时，被一个或多个不需要请求访问这个共享资源的中优先级任务抢占，导致资源迟迟得不到释放，从而使高优先级的任务实际上降到了低优先级任务的优先级水平，这就发生了优先级反转。

优先级继承是解决优先级反转的一种算法，通过调用这种算法把拥有互斥信号量的一个低优先级任务的优先级提升，提升到因等待这个资源而被阻塞的最高优先级任务的优先级水平。直到它释放这个互斥信号量后，它的优先级又回到原来状态。

6．信号量删除

调用 semDelete()函数可以终止一个信号量并重新分配任何关联的内存。信号量的删除将解锁阻塞在这个信号量上的任务。如果删除信号量失败则该函数返回 ERROR。当删除信号量的时候，其操作要非常小心，特别是在互斥操作中。

7．任务删除保护

如果在创建互斥信号量时，选项字包含 **SEM_DELETE_SAFE**，则任务在拥有信号量的时候将防止被删除。删除一个执行在临界区的任务会出现灾难性的后果。资源会保留在一个被破坏的状态，而采用信号量来保护这个资源将不会出现这种情况。

在讨论 taskLib 库时， taskSafe()和 taskUnsafe()函数提供了一种任务删除保护方案，因为这种操作与互斥操作通常同时执行，互斥信号量提供了 **SEM_DELETE_SAFE** 选项，该选项使 taskSafe()函数暗含在 semTake()函数中，taskUnsafe()函数暗含在 semGive()函数中。不过采用这种方案对保护任务安全删除更有效，因为由此产生的代码需要较少的内核函数。

8．警告

任务获得互斥信号量从而会提升任务的优先级到更高的优先级上，直到任务释放该信号量。当涉及到嵌套互斥时，将会发生无限制的优先级反转情况。如果嵌套互斥必不可少，考虑下面方案：

（1）避免重叠临界区；

（2）调整任务的优先级，以便没有任务在中优先级层次上；

（3）调整任务的优先级，以便不需要优先级继承协议；

（4）手动设置最高优先级协议。设置所有阻塞在互斥体上的任务的优先级到最高优先级上，然后获得互斥体。调用 semGive()后，设置优先级回到原有优先级级别上。实际上这种方案把队列转变成一个 FIFO 队列。

9．头文件

互斥信号量函数声明在头文件 semLib.h 中。

10．参考

相关信息请参考 semLib、semCLib、semBLib 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统章节。

2.1.3.2 互斥信号量函数详细描述

semMCreate()

函数原型：

```
SEM_ID semMCreate
(
```

```
    int options,                    /* 互斥信号量选项字 */
    )
```

功能描述：

创建并初始化一个互斥信号量。该函数分配并初始化一个互斥信号量。信号量的初始状态为满(full)。信号量选项包含下列各项：

SEM_Q_PRIORITY (0x1)：队列中的阻塞任务按优先级排列。

SEM_Q_FIFO (0x0)：队列中的阻塞任务按先进先出排列。

SEM_DELETE_SAFE (0x4)：防止拥有该信号量的任务遭受意外的删除。

SEM_INVERSION_SAFE (0x8)：防止系统出现优先级反转。该选项必须与 **SEM_Q_PRIORITY** 排列模式配套使用。

返回值：

成功创建信号量则返回信号量 ID，如果内存不足则返回 NULL。

参考：

相关信息请参考 semMLib、semLib、semBLib 库描述，以及 taskSafe()和 taskUnsafe()函数。

例：

```
/* 数据定义 */
SEM_ID mutexSemId;              /* 互斥信号量 ID */
...
/* 创建一个互斥信号量 */
mutexSemId = semMCreate (SEM_Q_PRIORITY|SEM_DELETE_SAFE|SEM_INVERSION_SAFE);

if (mutexSemId   == NULL)
        {
        perror ("Error in creating mutual exclusion semaphore");
        return (ERROR);
        }
...
```

semMGiveForce()

函数原型：

```
    STATUS semMGiveForce
    (
    SEM_ID semId   /* 信号量 ID */
    )
```

功能描述：

无条件释放一个互斥信号量。该函数释放一个互斥信号量而不管信号量所有权。它只是作为调试辅助功能来使用。

当任务“死亡”时持有某个互斥信号量，采用该函数特别有用，因为这个信号量可以被“复活”。该函数将释放信号量给阻塞队列中的第一个任务或使信号量满(如果没有任务被阻塞)。实际上，如果任务释放了自己所持有的信号量之后任务将继续执行。

警告：

该函数应当只是作为调试辅助功能来使用。

返回值：

成功释放则返回 OK，如果输入的是无效信号量 ID 则返回 ERROR。

参考：

相关信息请参考 semMLib 库描述，以及 semGive()函数。

2.1.4 通用的信号量函数

2.1.4.1 函数库描述

1．库命名

通用的信号量函数库名称为 semLib。

2．函数

表 2-4 中列出了通用的信号量函数。

表 2-4 通用的信号量函数

函 数	描 述	函 数	描 述
semGive()	释放信号量	semFlush()	解锁所有阻塞在信号量上的任务
semTake()	获得信号量	semDelete()	删除信号量

3．描述

semLib 库是 VxWorks 同步和互斥功能的基础。它们强劲而简单，是 VxWorks 众多模块的基本组成部分。

不同类型的信号量满足不同需求，虽然类型的行为不同，但基本接口是相同的。该函数库提供的信号量函数对 VxWorks 的所有类型信号量都可用。其两个基本操作是 semTake()和 semGive()，即获得和释放信号量。

信号量的创建和初始化由其他库来实现，依赖于使用的信号量类型。下面这些库包含了信号量的完整功能描述：

semBLib：二值信号量。

semCLib：计数信号量。

semMLib：互斥信号量。

semSmLib：共享内存信号量(需要 VxMP 模块)。

其中，二值信号量提供了快速和广泛的适用性。

4．信号量控制

semTake()调用获得一个指定的信号量，阻塞调用任务或使信号量不可用。所有类型的信号量在 semTake()上支持超时操作。超时值是指定了信号量所要阻塞的 tick 计数。

WAIT_FOREVER 和 **NO_WAIT** 是常用的超时设置。如果 semTake()超时，它将返回 ERROR。有关该操作的行为描述，参考对应信号量的库描述。

semGive()调用释放一个指定的信号量，解锁一个阻塞任务或使信号量可用。有关该操作的行为描述，参考对应信号量的库描述。

semFlush()调用可用于自动解锁所有挂起在信号量队列上的任务。例如，使所有的任务在允许运行前将处于非阻塞状态。在同步应用中，该函数被视为广播操作。使用 semFlush()函数解锁任务，但并不改变信号量状态；这与 semGive()函数不一样。

5．信号量删除

semDelete()调用终止一个信号量并重新分配任何关联的内存。信号量的删除将解锁阻塞在这个信号量上的任务；如果删除信号量失败则该函数返回 ERROR。当删除信号量的时候，其操作要非常小心，特别是在互斥操作中。为了避免删除一个持有信号量的任务给系统带来危难，应用程序应当采用删除信号量协议保证安全删除任务。

6．信号量信息

semInfo()调用对于调试而言是非常有帮助的，报告了所有阻塞在一个指定信号量上的任务。它捕获的只

是队列某个时刻的一个快照，毕竟信号量的状态是动态的。由于与信号量当前状态有关，所以应该限制挂起任务队列的使用，且只是作为调试来使用。

7．头文件

通用的信号量函数声明在 semLib.h 中。

8．参考

相关信息请参考 taskLib、semLib、semCLib、semBLib、semMLib 和 semSmLib 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统章节。

2.1.4.2　通用的信号量函数详细描述

semGive()

函数原型：

```
STATUS semGive
(
SEM_ID semId    /* 信号量 ID */
)
```

功能描述：

释放信号量。该函数对指定的信号量执行释放操作。视信号量类型而定，信号量的状态和阻塞任务的状态可能会受到影响。另外，semGive()的行为在库中充分进行了描述。

返回值：

成功释放信号量则返回 OK，如果输入无效信号量 ID 返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE，S_objLib_OBJ_ID_ERROR，S_semLib_INVALID_OPERATION

参考：

相关信息请参考 semMLib、semLib、semBLib、semCLib 和 semSmLib 库描述。

semTake()

函数原型：

```
STATUS semTake
(
SEM_ID semId,   /* 信号量 ID */
int timeout         /* 超时设定，时间单位是 tick。*/
)
```

功能描述：

获得信号量。该函数对指定的信号量执行获得操作。视信号量类型而定，信号量的状态和阻塞任务的状态可能会受到影响。另外，semTake()的行为在库中充分进行了描述。

用户可以指定超时设定。如果一个任务等待信号量超时，semTake()将返回 ERROR。

WAIT_FOREVER(-1)和 **NO_WAIT**(0)的超时设定表明永远等待或根本不等待。

当 semTake()由于超时返回时，错误号为 S_objLib_OBJ_TIMEOUT。

值得说明的一点是在中断服务程序中不可以调用 semTake()函数。

返回值：

成功获得信号量则返回 OK，如果信号量 ID 无效或任务超时则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE、S_objLib_OBJ_ID_ERROR、S_objLib_OBJ_UNAVAILABLE。

参考：

相关信息请参考 semMLib、semLib、semBLib、semCLib 和 semSmLib 库描述。

例：

```
/* 数据定义 */
SEM_ID semId1;              /* 信号量 1 */
SEM_ID semId2;              /* 信号量 2 */
int    numTimes;            /* 运行次数 */
...
STATUS taskA ()
    {
    int count;
    for (count = 0; count < numTimes; count++)
        {
     /* 获得信号量 semId1 */
        if (semTake (semId1, WAIT_FOREVER) == ERROR)
            {
            perror ("taskA: Error in semTake");
            return (ERROR);
            }
    printf ("taskA: Started first by taking the semId1 semaphore - %d times\n",
                (count + 1));
    printf("This is task   <%s> : Event A now done\n", taskName (taskIdSelf()));
    printf("taskA: I'm done, taskB can now proceed; Releasing semId2 semaphore\n\n");

     /* 释放信号量 semId2 */
        if (semGive (semId2) == ERROR)
            {
            perror ("taskA: Error in semGive");
            return (ERROR);
            }
        }
        return (OK);
    }
...
```

semFlush()

函数原型：

```
    STATUS semFlush
    (
    SEM_ID semId   /* 信号量 ID */
    )
```

功能描述：

解锁所有阻塞在信号量上的任务。该函数自动解锁所有挂起在指定的信号量上的任务，但信号量的状态将不会发生改变。所有阻塞的任务在有机会运行之前将进入就绪队列中。

在同步应用中刷新操作作为一种广播手段是非常有用的。对于 semMCreate()创建的互斥信号量，该操作

属于非法操作。

返回值：

成功解锁任务则返回 OK，如果输入无效信号量 ID 或信号量不支持该操作则返回 ERROR。

错误码：

S_objLib_OBJ_ID_ERROR。

参考：

相关信息请参考 semMLib、semLib、semBLib、semCLib 和 semSmLib 库描述。

semDelete()

函数原型：

```
STATUS semDelete
(
SEM_ID semId,  /* 信号量 ID */
)
```

功能描述：

删除信号量。该函数停止指定的信号量并重新分配与之关联的内存。任何阻塞在该信号量上的任务将被解锁。

返回值：

成功删除信号量则返回 OK，如果删除失败则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE、S_objLib_OBJ_ID_ERROR、S_smObjLib_NO_OBJECT_DESTROY。

参考：

相关信息请参考 semMLib、semLib、semBLib、semCLib 和 semSmLib 库描述。

2.1.5　信号量显示函数

2.1.5.1　函数库描述

1．库命名

信号量显示函数库名称为 semShow。

2．函数

表 2-5 中列出了信号量显示函数。

表 2-5　信号量显示函数

函　数	描　述
semShowInit()	初始化信号量显示模块
semInfo()	获得阻塞在信号量上的任务列表
semShow()	显示信号量信息

3．描述

semShow 库提供了信号量信息统计显示函数。例如信号量类型、信号量排列方法和任务阻塞列表等。

函数 semShowInit()把信号量显示模块链接到 VxWorks 系统中。当通过下面的方式配置信号量显示模块时，系统将自动调用该函数：

- 如果用户使用配置头文件，在 config.h 头文件中定义 **INCLUDE_SHOW_ROUTINES** 即可；
- 如果用户使用 Tornado 工程配置工具，选中 **INCLUDE_SEM_SHOW** 即可。

值得注意的是系统必须在调用 semShowInit()函数之后，才可以调用其他信号量显示函数。

4．头文件

信号量显示函数声明在 semLib.h 头文件中。

5．参考

相关信息请参考 semLib 库描述，以及“VxWorks Programmer's Guide”。

2.1.5.2　信号量显示函数详细描述

semShowInit()

函数原型：

STATUS semShowInit(void)

功能描述：

初始化信号量显示模块。该函数把信号量显示模块链接到 VxWorks 系统中。

返回值：

无。

参考：

相关信息请参考 semShow 库描述。

semInfo()

函数原型：

```
int semInfo
(
SEM_ID semId,   /* 信号量 ID */
int idList[],   /* 任务 ID 数组 */
int maxTasks    /* idList[]能容纳的最大任务数 */
)
```

功能描述：

获得阻塞在信号量上的任务列表。该函数返回阻塞在指定信号量上的任务数量。不大于 maxTasks 的任务 ID 数被复制到指定的数组 idList 中，这个数组是无序的。

警告：

在 semInfo()返回时，并不能保证所有列出的任务依然有效或没有新任务被阻塞。

返回值：

放置在数组 idList 中阻塞任务的个数。

参考：

相关信息请参考 semShow 库描述。

操作示范：

在宿主机 WindSh 工具中，semInfo()函数操作示范如下：

```
-> semInfo semId,idList=malloc(100),10
value = 1 = 0x1
```

上述结果表明有一个任务阻塞在信号量 semId 上。

semShow()

函数原型：

```
STATUS semShow
(
SEM_ID semId,   /* 信号量 ID */
int level       /* 0 = 摘要, 1 = 详细信息 */
)
```

功能描述：

显示信号量信息。该函数显示信号量的状态和被阻塞的任务。如果参数 level 为 0，它显示信号量的摘要。如果参数 level 为 1，显示阻塞任务的详细信息。

返回值：

成功显示信号量则返回 OK，如果信号量 ID 无效则返回 ERROR。

参考：

相关信息请参考 semShow 库描述，以及“VxWorks Programmer's Guide”中的 Target Shell 和 WindSh 章节。

操作示范：

在宿主机 WindSh 工具中，semShow()函数操作示范如下：

```
-> semShow semId,0
Semaphore Id          : 0x502e9d8
Semaphore Type        : BINARY
Task Queueing         : PRIORITY
Pended Tasks          : 1
State                 : EMPTY
-> semShow semId,1
Semaphore Id          : 0x502e9d8
Semaphore Type        : BINARY
Task Queueing         : PRIORITY
Pended Tasks          : 1
State                 : EMPTY

    NAME        TID      PRI      TIMEOUT
----------  --------  ---  -------
tHighPri    5015898   150      0
```

2.2　消息队列

2.2.1　消息队列管理函数

2.2.1.1　函数库描述

1．库命名

消息队列管理函数库名称为 msgQLib。

2．函数

表 2-6 中列出了消息队列管理函数。

表 2-6　消息队列管理函数

函　数	描　述
msgQCreate()	创建并初始化消息队列
msgQDelete()	删除消息队列
msgQSend()	往消息队列发送消息
msgQReceive()	从消息队列中接收消息
msgQNumMsgs()	获得消息队列中消息的个数

3．描述

msgQLib 库包含创建、删除和使用消息队列的函数，它是单 CPU 内部主要的任务间通信机制。消息队列允许消息按 FIFO 顺序排列在消息队列中。任何任务或中断服务程序可以发送消息给消息队列；任何任务也可以从消息队列中接收消息；多个任务可以从同一个消息队列进行中接收消息和发送消息。在两个任务间进行全双工通信通常需要两个消息队列，每个方向都有各自的消息队列。

4．消息队列的创建和使用

通过调用 msgQCreate()函数可以创建消息队列。它的参数需要指定消息队列中可以存放的最大消息个数和每个消息的最大长度。系统需要预分配足够的缓冲区给消息队列来存放消息。

任务或中断服务程序调用 msgQSend()函数往消息队列发送消息。如果没有任务在这个消息队列上等待消息，消息只是被添加到消息缓冲区中。如果有任务已经在这个消息队列上等待接收消息，则消息立即被交付

给第一个等待的任务。

任务调用 msgQReceive()函数从消息队列中接收消息。如果有消息已经存放在这个消息队列缓冲区中，第一个消息立即出列并返回给调用者。如果消息队列没有消息，调用任务将被阻塞并被添加到等待消息的任务队列中。等待任务的队列可以按任务优先级或 FIFO 顺序排列，当创建消息队列时，在选项参数中指定任务队列排列模式。

5．超时设定

函数 msgQSend()和 msgQReceive()采用超时参数。当发送消息时，如果消息队列中没有可用的缓冲区，超时参数指定等待缓冲区变为可用的最长时间。当接收消息时，如果没有消息立即可用，超时参数指定等待消息的最长时间。参数 timeout 可以拥有特定值 **NO_WAIT**(0)或 **WAIT_FOREVER**(-1)。**NO_WAIT** 意味着函数应当立即返回；**WAIT_FOREVER** 意味着函数永远等待消息，且等待操作永不超时。

6．紧急消息

msgQSend()函数允许指定消息的优先级，即一般级别(**MSG_PRI_NORMAL**)或紧急级别(**MSG_PRI_URGENT**)。一般级别的消息被添加到消息队列的尾部，紧急级别的消息被添加到消息队列的头部。

7．头文件

消息队列管理函数声明在 msgQLib.h 头文件中。

8．参考

相关信息请参考 pipeDrv、msgQSmLib 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统章节。

2.2.1.2 消息队列管理函数详细描述

msgQCreate()

函数原型：

```
MSG_Q_ID msgQCreate
(
int maxMsgs,          /* 消息最大个数 */
int maxMsgLength,     /* 消息最大长度 */
int options           /* 消息队列选项字 */
)
```

功能描述：

创建并初始化消息队列。该函数创建一个有能力容纳 maxMsgs 个消息的消息队列，且每个消息长度可达到 maxMsgLength 个字节。该函数返回消息队列 ID 来识别所创建的消息队列，在所有后来的消息队列函数调用中将通过 ID 对消息队列进行访问。

参数 options 指定被阻塞任务的排列顺序。选项字包含下列各项：

SEM_Q_PRIORITY(0x00)：阻塞的任务按 FIFO 顺序排列；

SEM_Q_FIFO(0x01)：阻塞的任务按任务优先级顺序排列。

返回值：

成功创建信号量则返回消息队列 ID，如果出错则返回 NULL。

错误码：

S_memLib_NOT_ENOUGH_MEMORY、S_intLib_NOT_ISR_CALLABLE。

参考：

相关信息请参考 msgQLib 及 msgQSmLib 库描述。

例：

```
/* 数据定义 */
```

```
#define MAX_MSG          10          /* 消息队列中最多消息个数 */
MSG_Q_IDmsgQId;                      /* 消息队列 ID */
struct msg {                         /* msg 数据结构 */
        int tid;                     /* 任务 id */
        int value;                   /* msg 的变量 */
        };
...
/* 创建一个消息队列 */
msgQId = msgQCreate (MAX_MSG, sizeof (struct msg), MSG_Q_FIFO);
...
if(msgQId == NULL)
        {
        perror ("Error in creating msgQ");
        return (ERROR);
        }
...
```

msgQDelete()

函数原型：

```
STATUS msgQDelete
(
MSG_Q_ID msgQId    /* 消息队列 ID */
)
```

功能描述：

删除消息队列。该函数删除指定的消息队列。任何阻塞在这个消息队列上的任务将被解锁。参数 msgQId 将不再是一个有效的消息队列 ID。

返回值：

成功删除则返回 OK，如果消息队列 ID 无效则返回 ERROR。

错误码：

S_objLib_OBJ_ID_ERROR、S_intLib_NOT_ISR_CALLABLE。

参考：

相关信息请参考 msgQLib 及 msgQSmLib 库描述。

msgQSend()

函数原型：

```
STATUS msgQSend
(
MSG_Q_ID msgQId,    /* 消息队列 ID */
char * buffer,      /* 要发送的消息 */
UINT nBytes,        /* 消息长度 */
int timeout,        /* 等待时间，其单位为 tick */
int priority        /* 消息优先级，MSG_PRI_NORMAL 或 MSG_PRI_URGENT */
)
```

功能描述：

发送消息给消息队列。该函数发送存放在 buffer 中、长度为 nBytes 的消息给消息队列 msgQId 。如果有任务已经在这个队列上等待接收消息，消息将立即被交付给第一个等待的任务。如果没有任务正在等待接收消息，消息将被保存在消息队列中。

如果消息队列满，参数 timeout 指定等待缓冲区变为可用的最长时间。参数 timeout 也可以包含下面的特定值：

NO_WAIT(0)：立即返回，即使没有发送消息；

WAIT_FOREVER(-1)：永远等待，且等待操作永不超时。

参数 priority 指定发送消息的优先级。可能的值是：

MSG_PRI_NORMAL(0)：一般级别；把消息添加到消息队列的尾部；

MSG_PRI_URGENT(1)：紧急级别；把消息添加到消息队列头部。

注意：

在中断服务程序中可以调用该函数，但是参数 timeout 必须是 **NO_WAIT**。

返回值：

成功发送消息则返回 OK，如果发送失败则返回 ERROR。

错误码：

S_distLib_NOT_INITIALIZED、S_objLib_OBJ_ID_ERROR、S_objLib_OBJ_DELETED、S_objLib_OBJ_UNAVAILABLE、S_objLib_OBJ_TIMEOUT、S_msgQLib_INVALID_MSG_LENGTH、S_msgQLib_NON_ZERO_TIMEOUT_AT_INT_LEVEL。

参考：

相关信息请参考 msgQLib 及 msgQSmLib 库描述。

例：

```
/* 数据定义 */
MSG_Q_IDmsgQId;                 /* 消息队列 ID */
struct msg {                    /* msg 数据结构 */
        int tid;                /* 任务 id */
        int value;              /* msg 的变量 */
        };
...
/* 生产任务 */
STATUS producerTask (void)
    {
    int count;
    int value;
    struct msg producedItem;                /* 发送的数据 */
    printf ("producerTask started: task id = %#x \n", taskIdSelf ());

    for (count = 1; count <= 100; count++)
        {
        value = count * 10;     /* produce a value */

        /* 填充 producedItem */
        producedItem.tid = taskIdSelf ();
        producedItem.value = value;
```

```
        /* 发送消息 */
        if ((msgQSend (msgQId, (char *) &producedItem, sizeof (producedItem),
                       WAIT_FOREVER,  MSG_PRI_NORMAL)) == ERROR)
            {
                perror ("Error in sending the message");
                return (ERROR);
            }
        else
            printf ("ProducerTask: tid = %#x, produced value = %d \n",
                                                    taskIdSelf (), value);
        }
    return (OK);
    }
```

msgQReceive()

函数原型：

```
int msgQReceive
(
MSG_Q_ID msgQId,    /* 消息队列 ID */
char * buffer,      /* 存放接收消息的缓冲区 */
UINT maxNBytes,     /* 缓冲区长度 */
int timeout         /* 等待时间，其单位为 tick */
)
```

功能描述：

从消息队列中接收消息。该函数从消息队列 msgQId 中接收消息。把接收到的消息复制到指定的 buffer 中，其缓冲区长度为 maxNBytes 个字节。如果消息长度大于 maxNBytes，系统将会丢弃消息的剩余部分(不返回错误迹象)。

当调用 msgQReceive()函数时，如果消息队列中没有可用消息，参数 timeout 指定等待消息的最长 tick 数。参数 timeout 可以包含下面的特定值：

NO_WAIT(0): 立即返回，即使没有接收到消息；

WAIT_FOREVER(-1): 永远等待，且等待操作永不超时。

警告：

在中断服务程序中不可以调用该函数。

返回值：

成功接收消息则返回存放在 buffer 中的消息字节数，如果接收失败则返回 ERROR。

错误码：

S_distLib_NOT_INITIALIZED、S_smObjLib_NOT_INITIALIZED、S_objLib_OBJ_ID_ERROR、S_objLib_OBJ_DELETED、S_objLib_OBJ_UNAVAILABLE、S_objLib_OBJ_TIMEOUT、S_msgQLib_INVALID_MSG_LENGTH。

参考：

相关信息请参考 msgQLib 及 msgQSmLib 库描述。

例：

```
/* 数据定义 */
MSG_Q_IDmsgQId;                 /* 消息队列 ID */
```

```
struct msg {                    /* msg 数据结构 */
        int tid;                /* 任务 id */
        int value;              /* msg 的变量 */
        };
...
/* 消费任务 */
STATUS consumerTask (void)
    {
    int count;
    struct msg consumedItem;                /* 存放接收到的数据 */

    printf ("\n\nConsumerTask: Started -   task id = %#x\n", taskIdSelf());

    for (count = 1; count <= 100; count++)
        {
        /* 接收消息 */
        if ((msgQReceive (msgQId, (char *) &consumedItem,
                          sizeof (consumedItem), WAIT_FOREVER)) == ERROR)
            {
            perror ("Error in receiving the message");
            return (ERROR);
            }
        else
            printf ("ConsumerTask: Consuming msg of value %d from tid = %#x\n",
                        consumedItem.value, consumedItem.tid);
        }
    return (OK);
    }
```

msgQNumMsgs()

函数原型：

```
int msgQNumMsgs
(
MSG_Q_ID msgQId /* 消息队列 ID */
)
```

功能描述：

获得消息队列中消息的个数。该函数返回指定的消息队列中当前排列的消息个数。

返回值：

成功获得则返回消息数，如果操作失败则返回 ERROR。

错误码：

S_distLib_NOT_INITIALIZED,S_smObjLib_NOT_INITIALIZED,S_objLib_OBJ_ID_ERROR。

参考：

相关信息请参考 msgQLib 及 msgQSmLib 库描述。

2.2.2　消息队列显示函数

2.2.2.1　函数库描述

1．库命名

消息队列显示函数库名称为 msgQShow。

2．函数

表 2-7 中列出了消息队列显示函数。

3．描述

msgQShow 库提供了消息队列信息统计显示函数，例如显示阻塞任务的排列方法、消息的排列和被阻塞的任务等。

表 2-7　消息队列显示函数

函　数	描　述
msgQShowInit()	初始化消息队列显示模块
msgQInfoGet()	获得消息队列信息
msgQShow()	显示消息队列信息

函数 msgQShowInit()把消息队列显示模块链接到 VxWorks 系统中。当通过下面的方式配置消息队列显示模块时，系统将自动调用该函数：

- 如果用户使用配置头文件，在 config.h 头文件中定义 **INCLUDE_SHOW_ROUTINES** 即可；
- 如果用户使用 Tornado 工程配置工具，选中 **INCLUDE_MSG_Q_SHOW** 即可。

值得注意的是系统必须在调用 msgQShowInit()函数之后，才可以调用其他消息队列显示函数。

4．头文件

消息队列显示函数声明在 msgQLib.h 头文件中。

5．参考

相关信息请参考 pipeDrv 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统章节。

2.2.2.2　消息队列管理函数详细描述

msgQShowInit()

函数原型：

```
void msgQShowInit(void)
```

功能描述：

初始化消息队列显示模块。该函数把消息队列显示模块链接到 VxWorks 系统中。

返回值：

无。

参考：

相关信息请参考 msgQShow 库描述。

msgQInfoGet()

函数原型：

```
STATUS msgQInfoGet
(
MSG_Q_ID msgQId,        /* 消息队列 ID */
MSG_Q_INFO * pInfo      /* 返回的消息信息 */
)
```

功能描述：

获得消息队列的信息。该函数获得消息队列的状态和内容。参数 pInfo 是一个指向 MSG_Q_INFO 结构的指针，该结构如下：

```
typedef struct /* MSG_Q_INFO */
{
```

```
int numMsgs;          /* 消息队列中消息的个数 */
int numTasks;         /* 等待消息队列的任务数 */
int sendTimeouts;     /* 发送超时设定的计数 */
int recvTimeouts;     /* 接收超时设定的计数*/
int options;          /* 消息队列选项字 */
int maxMsgs;          /* 最大消息个数 */
int maxMsgLength;     /* 每个消息的最大长度 */
int taskIdListMax;    /* 填充在 taskIdList 中的最大任务数 */
int * taskIdList;     /* 等待消息队列的任务 ID 数组 */
int msgListMax;       /* 填充在消息列表中最大的消息个数 */
char ** msgPtrList;   /* 消息队列中的消息数组 */
int * msgLenList;     /* 消息长度数组 */
} MSG_Q_INFO;
```

如果消息队列为空，可能有任务因接收消息而阻塞。如果消息队列已满，可能有任务因发送消息而阻塞。这些将根据下面信息决定：

- 如果 numMsgs 等于 0，那么 numTasks 显示因接收消息而被阻塞的任务；
- 如果 numMsgs 等于 maxMsgs，那么 numTasks 显示因发送消息而被阻塞的任务；
- 如果 numMsgs 大于 0 但小于 maxMsgs，那么 numTasks 将等于 0。

警告：

在获得信息的时候，该函数在锁中断环境下执行，这样会危及系统中所有潜伏中断的安全。通常情况下，该函数只是作为调试目的使用。

返回值：

成功获得信息则返回 OK，如果失败则返回 ERROR。

错误码：

S_distLib_NOT_INITIALIZED、S_smObjLib_NOT_INITIALIZED、S_objLib_OBJ_ID_ERROR。

参考：

相关信息请参考 msgQShow 库描述。

msgQShow()

函数原型：

```
STATUS msgQShow
(
MSG_Q_ID msgQId,   /* 消息队列 ID */
int      level     /* 0 = 摘要，1 = 详细信息 */
)
```

功能描述：

显示消息队列信息。该函数显示消息队列的状态和用户所选的内容。如果参数 level 为 0，它显示消息队列的摘要。如果参数 level 为 1，显示被阻塞任务的详细信息。

返回值：

成功显示信息则返回 OK，如果显示失败则返回 ERROR。

错误码：

S_distLib_NOT_INITIALIZED、S_smObjLib_NOT_INITIALIZED。

参考：

相关信息请参考 msgQShow 库描述，以及“VxWorks Programmer's Guide”中的 Target Shell、WindSh

章节。

操作示范：

在宿主机 WindSh 工具中，msgQShow()函数操作示范如下：

```
-> msgQShow msgQId, 0
Message Queue Id      : 0x502ef88
Task Queueing         : FIFO
Message Byte Len      : 8
Messages Max          : 1
Messages Queued       : 0
Receivers Blocked     : 1
Send Timeouts         : 0
Receive Timeouts      : 0

-> msgQShow msgQId, 1
Message Queue Id      : 0x502ef88
Task Queueing         : FIFO
Message Byte Len      : 8
Messages Max          : 1
Messages Queued       : 0
Receivers Blocked     : 1
Send Timeouts         : 0
Receive Timeouts      : 0

Receivers Blocked:

    NAME        TID      PRI    TIMEOUT
----------   --------   ---   -------
tHighPri     5015898    150      0
```

2.3　管道

2.3.1　管道 I/O 驱动函数库描述

1．库命名

管道 I/O 驱动函数库名称为 pipeDrv。

2．函数

表 2-8 中列出了管道 I/O 驱动函数。

表 2-8　管道 I/O 驱动函数

函　数	描　述
pipeDrv()	初始化管道驱动
pipeDevCreate()	创建管道设备

3．描述

pipeDrv 库提供一种通信机制，让任务之间通过标准的 I/O 接口进行通信。通过标准的 read()和 write()函数调用可以对管道进行读写操作。pipeDrv()函数初始化管道驱动，而调用 pipeDevCreate()函数则创建管道设备。

管道驱动使用 VxWorks 的消息队列模块来递交消息。通过 I/O 系统，管道驱动完全提供访问消息队列模块。在使用管道和消息队列之间主要区别如下：

- 管道需要命名；

- 管道使用标准的 I/O 函数—open()、close()、read()、write()；消息队列使用 msgQSend()、msgQReceive()函数；
- 管道响应标准的 ioctl()函数；
- 管道可以在 select()调用中被使用；
- 消息队列有更多的灵活性选项：超时设定和消息优先级；
- 管道的效率比消息队列低，因为附加一层 I/O 系统。

4．安装驱动

使用驱动之前，必须调用 pipeDrv()函数初始化和安装管道驱动。该函数也必须在创建任何管道之前被调用。如果定义了配置宏 **INCLUDE_PIPES**，则在根任务 usrRoot()中将自动调用该函数。

5．创建管道

使用管道之前，必须调用 pipeDevCreate()函数创建管道。例如，创建一个设备管道"/pipe/demo"，它可以存放 10 个消息，且每个消息的长度为 100 个字节，正确的调用是：

pipeDevCreate("/pipe/demo" ,10,100);

6．使用管道

一旦管道被创建，应用程序可以对它进行打开、关闭、读和写操作，就像访问其他 I/O 设备一样访问管道。通常从管道中读取的数据和写入管道中的数据是同一种数据结构。下面是读写管道的一个例子

例：

```
struct msg {                    /* msg 数据结构 */
        int tid;                /* 任务 id */
        int value;              /* msg 的变量 */
         };
          ...
        void pipeDemo(void)
        {
        int fd;
        struct msg outMsg;
        struct msg inMsg;
        int len;
        pipeDevCreate("/pipe/demo",10,100);
        fd = open ("/pipe/demo", O_RDWR);
        ...
        write (fd, &outMsg, sizeof (struct msg));
        len = read (fd, &inMsg, sizeof (struct msg));
        close (fd);
        }
```

数据被写入到管道中并作为单个消息保留，同时也将被单个读取。如果 read()函数调用所带的缓冲区比读取的消息小，剩余的消息将被丢弃。这样，管道 I/O 是“消息定向”而不是“流定向”。在这一点上，VxWorks 的管道不同于 UNIX 的“流定向”管道。

7．select()函数调用

管道的一个重要特征是可以在 select()函数调用中被使用。select()函数允许任务等待来自于一组可选 I/O 设备中的任何一个设备的输入。任务可以使用 select()等待来自于管道、套接字(socket)或串行设备的输入。

8．ioctl()功能

管道设备响应下面的 ioctl()功能。系统在 ioLib.h 头文件中定义了这些功能。

FIOGETNAME：获得文件名，并复制到指定的缓冲区。例如：

```
status = ioctl (fd, FIOGETNAME, &nameBuf);
```

FIONREAD：获得管道中第一个消息的字节数，并赋给指定的变量。例如：

```
status = ioctl (fd, FIONREAD, &nBytesUnread);
```

FIONMSGS：获得管道中当前的消息个数，并赋给指定的变量。例如：

```
status = ioctl (fd, FIONMSGS, &nMessages);
```

FIOFLUSH：丢弃管道中所有的消息并释放缓冲区。例如：

```
status = ioctl (fd, FIOFLUSH, 0);
```

9．头文件

管道 I/O 驱动函数声明在 pipeDrv.h 头文件中。

10．参考

相关信息请参考 msgQLib 库描述、select()函数，以及“VxWorks Programmer's Guide”中的 I/O 系统章节。

2.3.2　管道 I/O 驱动函数详细描述

pipeDrv()

函数原型：

```
STATUS pipeDrv (void)
```

功能描述：

初始化管道驱动。该函数初始化并安装管道驱动。在任何管道被创建之前，必须调用该函数。如果定义了配置宏 **INCLUDE_PIPES**，则在根任务 usrRoot()中将自动调用该函数。

返回值：

成功安装驱动则返回 OK，如果安装驱动失败则返回 ERROR。

参考：

相关信息请参考 pipeDrv 库描述。

pipeDevCreate()

函数原型：

```
STATUS pipeDevCreate
(
char * name,        /* 管道名 */
int nMessages,      /* 最大消息数 */
int nBytes          /* 消息长度 */
)
```

功能描述：

创建管道设备。该函数创建管道设备并分配内存给必要的结构和初始化设备所需的内存。当管道已满时，任务企图往管道中写将会被挂起，直到一个消息被读走。如果在中断服务程序中往一个已满的管道中写，则消息将会被丢弃。

返回值：

成功创建管道则返回 OK，如果创建失败返回 ERROR。

参考：

相关信息请参考 pipeDrv 库描述。

例：

```
struct msg {                    /* msg 数据结构 */
```

```
        int tid;            /* 任务 id */
        int value;          /* msg 的变量 */
        };
int         status;
...
/* 创建管道设备"/pipe/server" */
status = pipeDevCreate ("/pipe/server", 10, sizeof (struct msg));
if (status == ERROR)
        {
        perror ("Error in creating pipe");
        return (ERROR);
        }
...
```

2.4 信号

2.4.1 信号管理函数库描述

1．库命名

信号管理函数库名称为 sigLib。

2．函数

表 2-9 中列出了信号管理函数(没有列出 POSIX 信号接口)。

表 2-9 信号管理函数(不包含 POSIX 信号接口)

函 数	描 述	函 数	描 述
sigInit()	初始化信号模块	sigvec()	安装信号处理程序
sigqueueInit()	初始化排队信号模块	sigsetmask()	设置信号掩码
signal()	信号处理程序与信号关联	sigblock	屏蔽信号
sigtimedwait()	等待信号	raise()	向调用任务发送信号
sigwaitinfo()	实时等待信号	sigqueue()	向任务发送排队信号

3．描述

sigLib 库为任务提供信号接口函数。通过内部异步事件或任务上下文间的通信，信号可以用来改变任务的流控制。任何任务或中断服务程序可以向一个指定的任务“发出”(或发送)一个信号。接收到信号的任务立即挂起当前执行的线程，并调用任务指定的“信号处理”程序。信号处理程序是用户提供的程序，它与指定的信号捆绑在一起，只要接收到信息，系统将调用与信号关联的信号处理程序。

与其他的任务间通信机制相比较，信号机制更适合于错误和异常的处理。信号处理程序可以看作是一种“软”中断的服务程序，任何可能导致任务阻塞的函数都不能用在信号处理程序中。由于处理机制和中断服务程序非常类似，信号处理程序中也只能调用那些在中断服务程序中可以安全使用的函数。只有在保证信号处理程序不会出现死锁时，才可以在信号处理程序中调用其他的函数。

sigLib 库支持两种类型的信号：BSD 4.3 和 POSIX 信号接口。POSIX 接口提供一个标准接口，其功能比传统的 BSD 4.3 接口要多。而本小节主要对非 POSIX 接口的信号管理函数进行描述。

4．现场恢复功能

如果一个任务被挂起(例如等待一个信号量)并向这个任务(假定任务已经安装了信号处理程序)发送信号，于是信号处理程序将在获得信号量之前运行。当信号处理程序完成后，任务将返回到挂起点。如果有超时设置，那么当任务从处理程序中返回并回到挂起点时将又使用原始超时值。

下例是一个典型的信号处理程序

例：

```
void sigHandler
(
int sig, /* 信号号码(number) */
)
{
    logMsg ("signal handler is running!\n", 0, 0, 0, 0, 0, 0);
    ...
}
```

在 VxWorks 系统中，如果需要传递附加变量给信号处理程序，则定义如下：

```
void sigHandler
 (
 int sig,       /* 信号号码 */
 int code,      /* 附加码 */
 SIGCONTEXT *sigContext /* 信号前任务的上下文 */
 )
 {
   logMsg ("signal handler is running!\n", 0, 0, 0, 0, 0, 0);
   logMsg ("signal number = %d, additional code = %d!\n", sig, code, 0, 0, 0, 0);
    ...
 }
```

参数 code 只是对硬件异常所发出的信号有效。在这种情况下，它被用来区分信号变体。例如，溢出和除零都发出 SIGFPE(浮点异常)，但是持有的 code 值却不同。

5．信号处理程序定义

信号处理程序必须是随后描述的两种指定格式中的一种。这样，当信号产生时，操作系统便可以正确调用信号处理程序。

传统的信号处理程序接收信号号码作为惟一的输入参数。然而，某些程序产生的信号弥补 POSIX 实时扩展(P1003.b)，支持附加的应用变量传递给信号处理程序。这些包括 sigqueue()、异步 I/O、POSIX 实时时钟、以及 POSIX 消息队列产生的信号。

如果信号处理程序接收附加的参数，sigaction 结构的 sa_flags 项必须设置成 **SA_SIGINFO**，该结构是 sigaction()函数的一个输入参数。其信号处理程序的格式必须如下：

```
void sigHandler (int sigNum, siginfo_t * pInfo, void * pContext);
```

传统的信号处理程序不必把 sa_flags 项设置成 **SA_SIGINFO**，其格式如下：

```
void sigHandler (int sigNum);
```

6．异常处理

当遭遇到硬件异常时，系统将自动发送某信号。这种机制允许安装自定义异常处理程序。对于恢复灾难性的事件(例如总线或运算错误)，采用这种方式是非常有用的。典型情况是在程序中调用 setjmp()函数恢复控制，并在信号处理程序中调用 longjmp()函数恢复上下文。如果没有安装自定义处理程序或已经安装了处理程序应答硬件异常所产生的信号，那么任务将被挂起并发送一个消息到控制台。

在 VxWorks 系统中，有关硬件异常、信号、附加码(additional code)之间的关系请参考附录 A。

7．头文件

信号管理函数声明在 signal.h 头文件中。

8．参考

相关信息请参考 intLib 库描述、IEEEE POSIX 1003.1b，以及“VxWorks Programmer's Guide”中的基本内核章节。

2.4.2 信号管理函数详细描述

sigInit()

函数原型：

```
int sigInit(void)
```

功能描述：

该函数初始化信号模块。通常在系统启动程序 usrInit()中、并且在开中断之前调用该函数。

返回值：

成功初始化信号则返回 OK，如果不可以安装删除钩子则返回 ERROR。

错误码：

S_taskLib_TASK_HOOK_TABLE_FULL。

参考：

相关信息请参考 sigLib 库描述。

sigqueueInit()

函数原型：

```
int sigqueueInit
(
int nQueues             /* 排队信号个数 */
)
```

功能描述：

初始化排队信号模块。该函数初始化 POSIX 排队信号模块。在第一次调用 sigqueue()之前必须调用该函数。通常在系统启动程序 usrInit()中且在 sysInit()之后调用该函数。

分配 nQueues 缓冲区给 sigqueue()调用使用。每次调用 sigqueue()时需使用一个缓冲区；当交付信号后释放缓冲区。

返回值：

成功初始化排队信号则返回 OK，如果分配内存失败则返回 ERROR。

参考：

相关信息请参考 sigLib 库描述。

signal()

函数原型：

```
void (*signal
(
int   signo,                /* 信号号码 */
void(*pHandler) ()          /* 信号处理程序入口 */
)) ()
```

功能描述：

信号与信号处理程序关联。该函数注册信号处理程序。如果参数 pHandler 的值是 SIG_DFL，当 signo 信号出现时执行缺省信号处理程序。如果参数 pHandler 的值是 SIG_IGN，则忽略 signo 信号。参数 pHandler 必须指向一个函数，当 signo 信号出现时将调用这个函数。

返回值：

无。

参考：

相关信息请参考 sigLib 库描述。

例：

```
...
/*安装信号处理程序 */
signal (SIGBUS,   sigHandler);
signal (SIGILL,   sigHandler);
signal (SIGSEGV, sigHandler);
...
/* 信号处理程序 */
void sigHandler (int sig)
 {
 logMsg ("Signal %d received. Restarting"
                 " server\n", sig, 0,0,0,0,0);
 taskRestart (0);
 }
```

sigtimedwait()

函数原型：

```
int sigtimedwait
(
const sigset_t * pSet,                  /* 信号掩码 */
struct siginfo * pInfo,                 /* 返回值 */
const struct timespec * pTimeout        /* 等待时间 */
)
```

功能描述：

等待信号。该函数选择来自于 pSet 中待解决的信号。如果 pSet 中有多个信号等待解决，返回最小号码的信号。如果在调用该函数时 pSet 中没有待解决的信号，任务将被阻塞直到 pSet 中有一个待解决的信号，或直到等待时间 pTimeout 截止。如果 pTimeout 为 NULL，那么永远等待信号。

如果参数 pInfo 非空，经过选择的信号号码保存在 si_signo 成员中，并把产生信号的原因保存在 si_code 成员中。如果信号是排队信号，则值保存在 pInfo 的 si_value 成员中；否则 si_value 的内容不明确。

另外，在头文件 signal.h 中定义了 si_code 的值。相关值描述如下：

SI_USER：kill()函数发送的信号；

SI_QUEUE：sigqueue()函数发送的信号；

SI_TIMER：通过 timer_settime()函数设置定时器产生的信号；

SI_ASYNCIO：异步 I/O 请求产生的信号；

SI_MESGQ：消息到达时产生的信号；

SI_SYNC：硬件产生的信号。

函数 sigtimedwait()为任务等待异步产生的信号提供一个同步机制。任务可以使用 sigprocmask()函数阻塞任何处理同步的信号，并使信号处理程序处于默认情况。任务可以重复调用 sigprocmask()函数卸载任何已经发送的信号。

返回值：

成功调用则返回经过选择的信号号码，其他返回-1 并设置错误码。

错误码：

EINTR、EAGAIN、EINVAL。

参考：

相关信息请参考 sigLib 库描述。

例：

```
siginfo         sig_Info;       /* 返回值 */
timespec        sig_Timeout;    /* 等待时间 */
int             sig_num;        /* 信号号码 */
...
sig_Timeout.tv_sec = 6;         /* 等待 6s */
sig_Timeout.tv_nsec = 0;

/* 等待一个信号 */
sig_num = sigtimedwait(0x8, sig_Info, sig_Timeout);

if (sig_num == ERROR)
    {
     printf("wait signal failed!\n");
     return (ERROR);
    }
...
```

sigwaitinfo()

函数原型：

```
int sigwaitinfo
(
const sigset_t * pSet,                  /* 信号掩码 */
struct siginfo * pInfo,                 /* 返回值 */
)
```

功能描述：

实时等待信号。该函数相当于用 pTimeout=NULL 来调用 sigtimedwait()。相关信息参考 sigtimedwait()函数。

返回值：

成功调用则返回经过选择的信号号码，其他返回-1 并设置错误码。

错误码：

EINTR。

参考：

相关信息请参考 sigLib 库描述。

例：

```
siginfo          sig_Info;        /* 返回值 */
int              sig_num;         /* 信号号码 */
...
/* 实时等待一个信号 */
sig_num = sigwaitinfo(0x8, &sig_Info);
...
```

sigvec()

函数原型：

```
int sigvec
(
int sig,                          /* 信号号码 */
const struct sigvec * pVec,       /* 新的信号处理程序信息 */
struct sigvec * pOvec             /* 先前的信号处理程序信息 */
)
```

功能描述：

安装信号处理程序。该函数把 pVec 引用的信号处理程序与指定的信号 sig 关联在一起。也可以通过该函数来确定与信号 sig 关联的信号处理程序：如果 pVec 设置成 NULL(0)，sigvec()复制信号 sig 的当前信号处理程序到 pOvec 中，且不安装新信号处理程序。

pVec 和 pOvec 是指向 struct sigvec 结构的指针。传递的信息不但包括信号处理程序，而且还包括信号掩码和信号选项字。在头文件 signal.h 中定义了 sigvec 结构。

返回值：

成功注册则返回 OK(0)，如果信号号码无效或分配信号 TCB 失败则返回 ERROR(-1)。

错误码：

EINVAL、ENOMEM。

参考：

相关信息请参考 sigLib 库描述。

例：

```
...
SIGVEC sig;
sig.sv_handler = (VOIDFUNCPTR) sigHandler;
sig.sv_mask = 0;
sig.sv_flags = 0;

/* 安装信号处理程序 */
sigvec (SIGBUS, &sig, NULL);
...
/* 信号处理程序*/
void sigHandler
    (
    int sig,            /* 信号号码 */
    int code,           /* 附加码 */
    SIGCONTEXT *sigContext /* 信号之前的任务上下文 */
    )
```

```
{
if (sig == SIGBUS)
    printErr ("\nSignal SIGBUS (bus error) received");
else
    printErr ("\nSignal %x received", sig);
}
```

sigsetmask()

函数原型：

```
int sigsetmask
(
int mask /* 新的信号掩码 */
)
```

功能描述：

设置信号掩码。该函数设置调用任务的信号掩码为一指定的值。掩码位为 1 表示屏蔽这个信号。通过使用宏 SIGMASK 可以计算出信号号码所对应的掩码。

返回值：

先前的信号掩码值。

参考：

相关信息请参考 sigLib 库描述及 sigprocmask()函数。

sigblock()

函数原型：

```
int sigblock
(
int mask /* 信号掩码 */
)
```

功能描述：

屏蔽信号。该函数在 mask 中增加屏蔽信号，掩码位为 1 表示屏蔽这个信号。通过使用宏 SIGMASK 可以计算出信号号码所对应的掩码。

返回值：

先前的信号掩码值。

参考：

相关信息请参考 sigLib 库描述及 sigprocmask()函数。

raise()

函数原型：

```
int raise
(
int signo /* 信号号码 */
)
```

功能描述：

向调用任务发送信号。该函数向调用它的任务发送信号 signo。

返回值：

成功发送则返回 OK，如果信号号码或任务 ID 无效则返回 ERROR。

错误码：

EINVAL。

参考：

相关信息请参考 sigLib 库描述。

sigqueue()

函数原型：

```
int sigqueue
(
int tid,                    /* 任务 ID */
int signo,                  /* 信号号码 */
const union sigval value    /* 信号参数值 */
)
```

功能描述：

向任务发送排队信号。该函数向指定任务 tid 发送带有信号参数值 value 的信号 signo。

返回值：

成功发送则返回 OK。如果任务 ID 无效，或信号号码无效，或没有可用的排队信号缓冲区则返回 ERROR。

错误码：

EINVAL、EAGAIN。

参考：

相关信息请参考 sigLib 库描述。

例：

```
int         status;
int         taskId;
int         sigvalue;
...
sigvalue = 200;

/* 向任务发送排队信号 */
status = sigqueue(taskId,SIGUSR2,sigvalue);
...
```

第3章 时钟管理

3.1 ANSI 时间管理函数

3.1.1 函数库描述

1．库命名

ANSI 时间管理函数库名称为 ansiTime。

2．函数

表 3-1 中列出了 ANSI 时间管理函数。

表 3-1 ANSI 时间管理函数

函 数	描 述
asctime()	将 tm 型结构的日期和时间转换成包含日期和时间的字符串(ANSI)
asctime_r()	将 tm 型结构的日期和时间转换成包含日期和时间的字符串(POSIX)
clock()	判断程序的执行时间(ANSI)
ctime()	将秒形式的日历时间转换成包含日期和时间的字符串(ANSI)
ctime_t()	将秒形式的日历时间转换成包含日期和时间的字符串(POSIX)
difftime()	计算两个时间的差值(ANSI)
gmtime()	将秒形式的日历时间转换成以世界时间表示的 tm 型结构的日期和时间(ANSI)
gmtime_r()	将秒形式的日历时间转换成以世界时间表示的 tm 型结构的日期和时间(POSIX)
localtime()	将秒形式的日历时间转换成以当地时间表示的 tm 型结构的日期和时间(ANSI)
localtime_r()	将秒形式的日历时间转换成以当地时间表示的 tm 型结构的日期和时间(POSIX)
mktime()	将 tm 型结构的日期和时间转换成秒形式的日历时间(ANSI)
strftime ()	将 tm 型结构的日期和时间转换成格式化日期和时间串(ANSI)
time()	获得当前秒形式的日历时间(ANSI)

3．描述

在头文件 time.h 中定义了两个宏，声明了四个类型和一些函数用来处理时间。一些时间管理函数处理秒形式的日历时间，它包含当前日期(依据阳历[Gregorian calendar])和时间。有一些函数处理当地时间，这个日历时间需要明确具体的时区和夏令时，所以必须定义当地时区和夏令时。

两个宏定义是：NULL 和 CLOCKS_PER_SEC(每秒 tick 数)。

4．类型

time.h 文件中定义了四个类型，即：size_t、clock_t、time_t 和 struct tm。其中，clock_t 和 time_t 是表示时间的算术类型；而 struct tm 是由日期和时间组成的日历时间。该结构原型如下所示：

```
struct tm
    {
```

```
int tm_sec;          /* 秒 - [0, 59] */
int tm_min;          /* 分 - [0, 59] */
int tm_hour;         /* 小时 - [0, 23] */
int tm_mday;         /* 日 - [1, 31] */
int tm_mon;          /* 月 - [0, 11] */
int tm_year;         /* 年 - 1900 以后    */
int tm_wday;         /* 星期 - [0, 6] */
int tm_yday;         /* 天 - [0, 365] */
int tm_isdst;        /* 夏令时(Daylight Saving Time)标识 */
};
```

如果实行夏令时，那么 tm_isdst 是一个正的值；如果不实行夏令时，那么 tm_isdst 的值为 0；如果 tm_isdst 项不可用，那么 tm_isdst 的值为负数。

如果设置了环境变量 TIMEZONE，tm_isdst 的值将从 TIMEZONE 变量中重新获得，否则从当前 tm_isdst 项中获得。TIMEZONE 形式如下：

name_of_zone:(unused):time_in_minutes_from_UTC:daylight_start:daylight_end

为了计算当地时间，time_in_minutes_from_UTC 的值需要减去 UTC(Universal Time Coordinated)；且 time_in_minutes_from_UTC 必须是正数。

环境变量 daylight 的表达形式为 mmddhh(month-day-hour)，例如：

UTC::0:040102:100102

5．重入

在这里有一些成对函数，例如 ctime()和 ctime_r()。只是 xx_r()形式的函数是可重入的，而 xx()形式的函数是不可重入的。

6．头文件

ANSI 时间管理函数声明在 time.h 库文件中。

7．参考

相关信息请参考 ansiLocale 库及美国国家标准 X3.159-1989。

3.1.2 ANSI 时间管理函数详细描述

asctime()

函数原型：

```
char * asctime
(
const struct tm * timeptr /* tm 型结构的日期和时间 */
)
```

功能描述：

将 tm 型结构的日期和时间转换成包含日期和时间的字符串。该函数将 timeptr 指向的日期和时间转换成如下形式的字符串：

SUN SEP 16 01:03:52 1973\n\0

该函数是不可重入的。有关重入型，请参考 asctime_r()函数。

返回值：

指向包含日期和时间的字符串指针。

参考：

相关信息请参考 ansiTime 库描述。

例：

```
void time_test(void)
{
 time_t cur ;
 struct tm newtime;

/* 获得当前日期与时间 */
 cur = time((time_t *)NULL);

/* 将秒钟形式的日历时间转换成日期与时间 */
 newtime = localtime( &cur );

/* 显示日期和时间 */
 printf("%s\n", asctime( &newtime));
}
```

asctime_r()

函数原型：

```
int asctime_r
(
const struct tm * timeptr,          /* tm 型结构的日期和时间 */
char * asctimeBuf,                  /* 存放字符串的缓冲区 */
size_t * buflen                     /* 缓冲区大小 */
)
```

功能描述：

将 tm 型结构的日期和时间转换成包含日期和时间的字符串。该函数将 timeptr 指向的日期和时间转换成如下形式的字符串：

SUN SEP 16 01:03:52 1973\n\0

并复制字符串到 asctimeBuf 中。这个调用是 asctime()函数的 POSIX 可重入型函数。

返回值：

实际字符串大小。

参考：

相关信息请参考 ansiTime 库描述。

例：

```
void time_test(void)
{
 time_t cur ;
 struct tm newtime;
 char     * timeBuf;
 int                num;
/* 获得当前日期与时间 */
 cur = time((time_t *)NULL);

/* 将秒钟形式的日历时间转换成日期与时间 */
```

```
newtime = localtime( &cur );

/* 把日期和时间转换成字符串 */
num = asctime_r( &newtime,timeBuf,20);
printf("%s\n", timeBuf);
}
```

clock()

函数原型：

```
clock_t clock(void)
```

功能描述：

判断程序的执行时间。当用户采取措施提高程序的性能时，可能需要计算程序各个部分耗时多少。这样可以判断程序的哪个部分最耗时。作为原则，应首先优化程序中占用处理器时间最多的部分。为帮助用户判断程序占用的时间，系统提供了 clock()函数，它返回时钟周期数。

通过该函数可以判断程序执行所需的最小处理时间。将 clock()返回的值除以常数 CLOCK_PER_SEC，可以把时间转换成秒的形式。

返回值：

成功获得则返回 tick 数，如果失败则返回 ERROR。

参考：

相关信息请参考 ansiTime 库描述。

例：

```
clock_t   Process(void)
{
 clock_t processor_time;
int i;
int num=0;
 /* 获得处理器当前 tick 数 */
 processor_time = clock();
 for(i=0; i<1000; i++)
{
num=num+1;
}
 processor_time = clock() - processor_time;
 return(processor_time);
}
```

ctime()

函数原型：

```
char * ctime
(
const time_t * timer      /* 秒钟形式的日历时间 */
)
```

功能描述：

将秒钟形式的日历时间转换成包含日期和时间的字符串。该函数将 timer 指向的日历时间转换成字符串。

等价于：

```
asctime (localtime (timer));
```

ctime()函数是不可重入的。有关重入型，请参考 ctime_r()函数。

返回值：

指向包含日期和时间的字符串指针。

参考：

相关信息请参考 ansiTime 库描述，asctime()和 localtime()函数。

例：

```
void ctime_test(void)
{
 time_t cur ;

/* 获得当前日期与时间 */
 cur = time((time_t *)NULL);

/* 显示日期和时间 */
 printf("%s\n", ctime( &cur));
}
```

ctime_r()

函数原型：

```
char * ctime_r
(
const time_t * timer,     /* 秒形式的日历时间 */
char * asctimeBuf,        /* 存放字符串的缓冲区 */
size_t * buflen           /* 缓冲区大小 */
)
```

功能描述：

将秒形式的日历时间转换成包含日期和时间的字符串。该函数将 timer 指向的日历时间转换成字符串。等价于：

```
asctime (localtime (timer));
```

该函数是 ctime()函数的 POSIX 可重入型。

返回值：

指向包含日期和时间的字符串指针。

参考：

相关信息请参考 ansiTime 库描述，asctime()和 localtime()函数。

例：

```
void time_test(void)
{
 time_t cur ;
 char    * timeBuf;
 int                num;
```

```
/* 获得当前日期与时间 */
 cur = time((time_t *)NULL);

/* 显示日期和时间 */
printf("%s\n", ctime_r( &cur,timeBuf,20));
}
```

difftime()

函数原型：

```
double difftime
(
time_t time1,          /* 秒形式的后来时间 */
time_t time0           /* 秒形式的早期时间 */
)
```

功能描述：

计算两个时间的差值。该函数以浮点数的形式返回两个时间的差值：time1 – time0。

返回值：

浮点数形式的时间差值。

参考：

相关信息请参考 ansiTime 库描述。

例：

```
void diff_time(void)
{
time_t start_time;
time_t current_time;

/* 获得当前日期与时间 */
time(&start_time);
printf("About to delay 6 seconds!\n");
do
{
 time(&current_time);
 }while(difftime(current_time,start_time) < 6.0)
printf("End!\n");
}
```

gmtime()

函数原型：

```
struct tm *gmtime
(
const time_t * timer /* 秒形式的日历时间 */
)
```

功能描述：

将秒形式的日历时间转换成以世界时间表示的 tm 型结构的日期和时间。该函数把 timer 指向的日历时间

转换成 tm 型结构的日期和时间。

另外，该函数是不可重入的。有关重入型，请参考 gmtime_r()函数。

返回值：

成功转换则返回一个指向 tm 型结构的指针，如果世界时间无效则返回空指针。

参考：

相关信息请参考 ansiTime 库描述。

例：

```
void gmtime_test(void)
{
struct tm *gm_date;
time_t seconds;

/* 获得当前日期与时间 */
time(&seconds);

/* 将秒形式的日历时间转换成以世界时间表示的日期和时间 */
gm_data = gmtime(&seconds);

printf("Current date: %d-%d-%d\n", gm_date->tm_mon+1, gm_date->tm_mday,
        gm_date->tm_year);
printf("Current time: %02d:%02d\n", gm_date->tm_hour, gm_date->tm_min);
}
```

gmtime_r()

函数原型：

```
int gmtime_r
(
const time_t * timer,   /* 秒形式的日历时间 */
struct tm * timeBuffer /* 存放 tm 型结构的日期和时间的缓冲区 */
)
```

功能描述：

将秒形式的日历时间转换成以世界时间表示的 tm 型结构的日期和时间。该函数把 timer 指向的日历时间转换成 tm 型结构的日期和时间，并存放在 timeBuffer 缓冲区中。

另外，该函数是 gmtime()函数的 POSIX 可重入型。

返回值：

成功转换则返回 OK。

参考：

相关信息请参考 ansiTime 库描述。

例：

```
void gmtime_test(void)
{
struct tm gm_date;
time_t seconds;
int status;
```

```
/* 获得当前日期与时间 */
time(&seconds);

/* 将秒钟形式的日历时间转换成以世界时间表示的日期和时间 */
status = gmtime_r(&seconds,&gm_date);

if(status == OK)
  {
    printf("Current date: %d-%d-%d\n", gm_date.tm_mon+1, gm_date.tm_mday,
        gm_date.tm_year);
    printf("Current time: %02d:%02d\n", gm_date.tm_hour, gm_date.tm_min);
  }
else
    printf("convert calendar time failed!\n");
}
```

localtime()

函数原型：

```
struct tm *localtime
(
const time_t * timer /* 秒钟形式的日历时间 */
)
```

功能描述：

将秒形式的日历时间转换成 tm 型结构的日期和时间。该函数将 timer 指向的日历时间转换成以当地时间表示的 tm 型结构的日期和时间。

返回值：

一个指向 tm 型结构的指针，其结构包含当地日期和时间。

参考：

相关信息请参考 ansiTime 库描述。

例：

```
void Ltime(void)
{
struct tm *current_date;
time_t seconds;

/* 获得当前日期与时间 */
time(&seconds);

/* 将秒形式的日历时间转换成以当地时间表示的 tm 型结构的日期和时间 */
current_date = localtime(&seconds) ;
printf("Current date: %d-%d-%d\n", gm_date.tm_mon+1, gm_date.tm_mday,
        gm_date.tm_year);
printf("Current time: %02d:%02d\n", gm_date.tm_hour, gm_date.tm_min);
```

```
}
```

localtime_r()

函数原型：

```
int localtime_r
(
const time_t * timer,    /* 秒钟形式的日历时间 */
struct tm * timeBuffer   /* 存放 tm 型结构的日期和时间的缓冲区 */
)
```

功能描述：

将秒钟形式的日历时间转换成 tm 型结构的日期和时间。该函数将 timer 指向的日历时间转换成以当地时间表示的 tm 型结构的日期和时间，并存放在 timeBuffer 缓冲区中。

返回值：

成功转换则返回 OK。

参考：

相关信息请参考 ansiTime 库描述。

mktime()

函数原型：

```
time_t mktime
(
struct tm * timeptr /* 一个指向 tm 结构的指针 */
)
```

功能描述：

将 tm 型结构的日期和时间转换成秒形式的日历时间。该函数把 timeptr 指向的日期和时间转换成日历时间值，其值等于 time()函数返回的值。

返回值：

成功转换则返回秒钟形式的日历时间，如果出错则返回 ERROR(-1)。

参考：

相关信息请参考 ansiTime 库描述。

例：

```
void time_test(void)
{
 int status;
 time_t seconds;
 struct tm time_fields;
 time_fields.tm_mday = 31;
 time_fields.tm_mon = 10;
 time_fields.tm_year = 97;
 status = mktime(&time_fields);
 if(status == ERROR)
     printf("Error converting fields!\n");
 else
     printf("Julian date for October 31, 1997 is %d\n", time_fields.tm_yday);
```

```
}
```

strftime()

函数原型：

```
size_t strftime
(
char * s,                    /* 字符串 */
size_t n,                    /* 字符串的最大字符个数 */
const char * format,         /* 字符串输出格式 */
const struct tm * tptr       /* tm 型结构的日期和时间 */
)
```

功能描述：

将 tm 型结构的日期和时间转换成格式化日期和时间串。表 3-2 列出了字符串输出格式。

表 3-2 strftime 函数的格式标识符

格式标识符	描 述	格式标识符	描 述
%a	简写的星期名称	%P	字符 AM 或 PM，用于 12 小时制的时钟
%A	完整的星期名称	%S	用两位十进制表示的秒钟(00～59)
%b	简写的月份名称	%U	用两位十进制表示的星期数目(00～53),以星期日为每周的开始
%B	完整的月份名称	%w	每周的日期(0～6)，在这里星期日为 0
%c	日期和时间	%W	用两位十进制表示的星期数目(00～53),以星期一为每周的开始
%d	用两位十进制表示的日期(0～31)	%x	日期
%H	用两位十进制表示的 24 小时制的小时(00～23)	%X	时间
%I	用两位十进制表示的 12 小时制的小时(01～12)	%y	用两位十进制表示的年份(00～99)
%j	用三位十进制表示的儒略历(Julian calendar)日期(001～366)	%Y	用四位十进制表示的年份
%m	用两位十进制表示的月份(01～12)	%Z	时区名称或缩写
%M	用两位十进制表示的分钟(00～59)	%%	%字符

返回值：

参数 s 中的字符个数。

参考：

相关信息请参考 ansiTime 库描述。

例：

```
void   time_format(void)
{
 char buffer[128];
 struct tm *datetime;
```

```
time_t current_time;

/* 获得当前日期与时间 */
time(&current_time);
datetime = localtime(&current_time);
...
/* 将日期和时间转换成格式化日期和时间串 */
strftime(buffer,sizeof(buffer), "%x   %x", datetime);
printf("Using %%x %%x: %s\n",buffer);
}
```

time()

函数原型：

```
time_t time
(
time_t * timer /* 秒钟形式的日历时间 */
)
```

功能描述：

获得当前秒形式的日历时间。如果参数 timer 非空，它也保存返回值。

返回值：

成功获得则返回当前秒钟形式的日历时间，如果日历时间无效则返回 ERROR。

参考：

相关信息请参考 ansiTime 库描述。

例：

```
time_t current_time;
time_t get_time;
...
/* 获得当前日期与时间 */
get_time = time(&current_time);
if(get_time == ERROR)
  {
   printf("Get current time failed!\n");
   return(ERROR);
  }
...
```

3.2 时钟 tick 管理函数

3.2.1 函数库描述

1．库命名

时钟 tick 管理函数库名称为 tickLib。

2．函数

表 3-3 中列出了时钟 tick 管理函数。

3．描述

tickLib 库为 VxWorks 内核时钟 tick 提供了接口。通过这些接口可以增加、改变、获得 tick 计数器的值，VxWorks 的内核模块、taskDelay()、wdStart()、kernelTimeslice()函数以及信号量超时设定都依赖时钟 tick。在每一种情况中，指定的超时与当前时间有关。调用 tickSet()函数不会影响超时关系，该函数只是改变绝对时间。函数 tickSet()和 tickGet()将系统内核的其余部分与绝对时间隔离开来。

日计时时钟或其他辅助计时基数对于天或更长时间超时设置是较合意的。一些计时基数的精确性非常高，一些外部计时计数甚至可以周期性自身校准。

表 3-3 时钟 tick 管理函数

函 数	描 述
tickAnnounce()	通知内核一个时钟 tick 到
tickSet()	设置内核 tick 计数器的值
tickGet()	获得内核 tick 计数器的值

4．头文件

时钟 tick 管理函数声明在 tickLib.h 头文件中。

5．参考

相关信息请参考 kernelLib、taskLib、semLib 和 wdLib 库描述，以及“VxWorks Programmer's Guide”中的基本内核章节。

3.2.2 时钟 tick 管理函数详细描述

tickAnnounce()

函数原型：

```
void tickAnnounce (void)
```

功能描述：

该函数通知内核一个时钟 tick 到。在自定义的系统时钟 ISR 中应当调用该函数，并且调用 sysClkConnect()函数安装系统时钟 ISR。通常情况下，系统时钟频率范围是 60～100Hz。系统时钟频率超过 600Hz，导致系统把大部分时间花费在时钟上，处理机的工作效率将会很低。默认情况下，usrConfig.c 中的 usrClock()调用这个函数。

返回值：

无。

参考：

相关信息请参考 tickLib、kernelLib、taskLib、semLib 和 wdLib 库描述，以及“VxWorks Programmer's Guide”中的基本内核章节。

例：

```
/* 用户自定义的系统时钟中断服务程序 */
void usrClock (void)
    {
    tickAnnounce();        /* 通知内核一个时钟 tick 到 */
    }
```

tickSet()

函数原型：

```
    void tickSet
    (
    ULONG ticks            /* tick 计数器的值 */
    )
```

功能描述：

设置内核 tick 计数器的值。该函数设置内部 tick 计数器为一指定的值。通过调用 tickGet()函数将反映新

的计数值，但对于任何任务而言，调用 tickSet()将不会改变延迟时间或超时值。例如，如果一个任务延迟 10 个 tick，通过调用 tickSet()函数提升计时值，但被延迟的任务将依然延迟直到 tickAnnounce()调用了 10 次。

返回值：

无。

参考：

相关信息请参考 tickLib 库描述,tickGet()和 tickAnnounce()函数。

tickGet()

函数原型：

ULONG tickGet(void)

功能描述：

获得内核 tick 计数器的值。该函数返回 tick 计数器的当前值。在启动时，这个值被设置成 0；通过调用 tickAnnounce()增加这个值；调用 tickSet()可以改变这个值。

返回值：

tick 计数器的当前值。

参考：

相关信息请参考 tickLib 库描述,tickSet()和 tickAnnounce()函数。

例：

```
void tickDemo(void)
 {

 ULONG  SettickVal=0;
 ULONG   GettickVal=0;

 /* 获得内核当前 tick 计数器的值 */
 GettickVal = tickGet();
 printf("ticker's value is: %ld\n", GettickVal);

 /* 设置内核 tick 计数器的值 */
 SettickVal = 60;
 printf("Set ticker's value is: %ld\n", SettickVal);
 tickSet(SettickVal);

 /* 获得内核当前 tick 计数器的值 */
 tickTime = tickGet ();
 printf("ticker's value is: %ld\n",tickTime);
}
```

3.3 看门狗定时器

3.3.1 看门狗定时器管理函数

3.3.1.1 函数库描述

1．库命名

看门狗定时器管理函数库名称为 wdLib。

2．函数

表 3-4 中列出了看门狗定时器管理函数。

表 3-4 看门狗定时器管理函数

函 数	描 述
wdCreate()	创建看门狗定时器
wdDelete()	删除看门狗定时器
wdStart()	启动看门狗定时器
wdCancel()	取消正在计数的看门狗定时器

3．描述

wdLib 库提供了通用的看门狗定时器模块。任何任务都可以创建看门狗定时器，并在指定延迟之后，使用它在系统时钟 ISR 的上下文运行一个指定的程序。

通过调用 wdCreate()创建一个看门狗定时器，用 wdStart()可以启动它。wdStart()函数指定运行的超时程序及延迟时间等。一旦延迟时间到，系统将执行 wdStart()函数指定的超时程序，除非调用 wdCancel()取消定时器。

每调用一次 wdStart()，超时函数仅执行一次；在超时后，不需要用 wdCancel()来取消定时器或在超时期间内回调自身。

4．注意

系统在中断级调用超时程序，而不是在任务上下文中。这样，对于超时程序将有一些限制，其限制规则与中断服务程序一样。例如，不可以获得信号量、不可以调用 printf()等。

5．头文件

看门狗定时器管理函数声明在 wdLib.h 头文件中。

6．参考

相关信息请参考 logLib 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统章节。

3.3.1.2 看门狗定时器管理函数详细描述

wdCreate()

函数原型：

```
WDOG_ID wdCreate (void)
```

功能描述：

创建看门狗定时器。该函数通过在内存中分配一个 WDOG 结构，创建一个看门狗定时器。

返回值：

成功创建则返回看门狗 ID，如果内存不足则返回 NULL。

参考：

相关信息请参考 wdLib 库描述、wdDelete()函数。

例：

```
WDOG_ID wdId = NULL;                    /* 看门狗定时器 ID */
...
if ((wdId = wdCreate()) == NULL)
        {
        perror ("Error in creating watchdog timer");
        return (ERROR);
        }
...
```

wdDelete()

函数原型：

```
STATUS wdDelete
(
WDOG_ID wdId /* 看门狗 ID */
```

```
)
```

功能描述：

删除看门狗定时器。该函数将从定时器队列上撤走指定的看门狗，并重新分配内存。

返回值：

成功删除则返回 OK，如果删除失败则返回 ERROR。

参考：

相关信息请参考 wdLib 库描述以及 wdCreate()函数。

wdStart()

函数原型：

```
STATUS wdStart
(
WDOG_ID wdId,           /* 看门狗 ID */
int delay,              /* tick 形式的延迟计数 */
FUNCPTR pRoutine,       /* 超时时调用的程序 */
int parameter           /* 传递给超时程序的参数 */
)
```

功能描述：

启动看门狗定时器。该函数把看门狗定时器添加到系统 tick 队列中。在延迟到达后，系统在中断级调用超时程序。

如果需要替换延迟时间或所需执行的超时程序，可以用同样的看门狗 ID 重新调用 wdStart()来实现；在指定的 tick 计数到达前，可以通过调用 wdCancel()来结束看门狗定时器的执行。

看门狗定时器仅执行一次，除非应用程序需要周期性地执行定时器。为了实现这个目的，定时器程序本身必须调用 wdStart()来重新启动定时器。

返回值：

成功启动则返回 OK，如果启动失败则返回 ERROR。

参考：

相关信息请参考 wdLib 库描述以及 wdCancel()函数。

例：

```
#define TIME_BETWEEN_INTERRUPTS          5     /* 5 tick */
WDOG_ID     wdId = NULL;                       /* 看门狗定时器 ID */
STATUS      st;

/* 启动一个看门狗定时器，用看门狗定时器模拟硬件中断 */
st = wdStart(wdId, TIME_BETWEEN_INTERRUPTS, (FUNCPTR) syncISR, 5);

if (st == ERROR)
    {
      perror ("Error in starting watchdog timer ");
      return (ERROR);
    }
...
/* 模拟一个产生中断的硬件设备。 */
```

```
void syncISR
    (
    int times
    )
    {
    semGive (semId);
    times--;
    if (times > 0)
        wdStart (wdId, TIME_BETWEEN_INTERRUPTS, (FUNCPTR) syncISR, times);
    }
```

wdCancel()

函数原型：

```
STATUS wdCancel
(
WDOG_ID wdId /* 看门狗 ID */
)
```

功能描述：

取消正在计数的看门狗定时器。该函数取消正在运行的看门狗定时器，并把延迟计数归0。

返回值：

成功取消看门狗则返回 OK，如果取消失败则返回 ERROR。

参考：

相关信息请参考 wdLib 库描述以及 wdStart()函数。

3.3.2 看门狗显示函数

3.3.2.1 *函数库描述*

1．库命名

看门狗显示函数库名称为 wdShow。

2．函数

表 3-5 中列出了看门狗显示函数。

表 3-5 看门狗显示函数

函 数	描 述
wdShowInit()	初始化看门狗显示模块
wdShow()	显示关于看门狗的信息

3．描述

wdShow 库提供看门狗信息统计显示函数。其信息包括看门狗活动状态、看门狗超时程序等。

函数 wdShowInit()把看门狗显示模块链接到 VxWorks 系统中。当通过下面的方式配置看门狗显示模块后，系统将自动调用该函数：

- 用户使用配置头文件，在 config.h 文件中定义 **INCLUDE_SHOW_ROUTINES**；
- 用户使用 Tornado 工程配置工具，选中 **INCLUDE_WATCHDOGS_SHOW**。

值得说明的是，系统必须在调用了 wdShowInit()函数之后，才可以调用 wdShow()函数。

4．头文件

看门狗显示函数声明在 wdLib.h 头文件中。

5．参考

相关信息请参考 wdLib 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统和 WindSh 章节。

3.3.2.2 看门狗显示函数详细描述

wdShowInit()

函数原型：

```
void wdShowInit(void)
```

功能描述：

初始化看门狗显示模块。该函数把看门狗显示模块链接到 VxWorks 系统中。

返回值：

无。

参考：

相关信息请参考 wdShow 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统和 WindSh 章节。

wdShow()

函数原型：

```
STATUS wdShow
(
WDOG_ID wdId /* 看门狗 ID */
)
```

功能描述：

显示看门狗信息。该函数显示看门狗的状态。

返回值：

成功显示则返回 OK，如果失败则返回 ERROR。

参考：

相关信息请参考 wdShow 库描述，以及“VxWorks Programmer's Guide”中的基本操作系统和 WindSh 章节。

操作示范：

在宿主机 WindSh 工具中，wdShow()函数操作示范如下所示：

```
-> wdShow wdId
Watchdog ID          : 0x502ef38
State                : IN_Q
Ticks Remaining      : 100
Routine              : 0x427354
Parameter            : 0x1
```

3.4 执行计时器函数

3.4.1 函数库描述

1．库命名

执行计时器函数库名称为 timexLib。

2．函数

表 3-6 中列出了执行计时器函数。

表 3-6 执行计时器函数

函 数	描 述	函 数	描 述
timexInit()	包含执行计时器库	timexN()	重复测量一个或一组函数执行的时间
timexClear()	清除被测执行时间的函数列表	timexPost()	指定在测量函数执行时间之后将被调用的函数
timexFunc()	指定被测执行时间的函数	timexPre()	指定在测量函数执行时间之前将被调用的函数
timexHelp()	显示执行计时器模块的纲要		
timex()	测量函数单次执行的时间	timexShow()	显示被测执行时间的函数列表

3．描述

timexLib 库包含的函数用来测量程序、一个函数或一组函数执行的时间，且使用 VxWorks 系统时钟作为时间基数。如果函数的执行时间比时间基数还要短，则可以多次调用该函数从而确定一个令人满意的平均执行时间。

最多可以把 4 个函数作为一组来测量它们的执行时间。另外，可以在测量函数执行时间之前或之后指定 4 个函数，从而在测量函数执行时间之前和之后执行这些函数。函数 timexPre()和 timexPost()用来指定测量之前和之后将被调用的函数，而 timexFunc()则指定被测执行时间的函数。

函数 timex()用来测量最多可带 8 个参数的函数单次执行的时间。如果没有指定参数，函数 timex()使用 timexFunc()、timexPre()和 timexPost()函数创建的函数列表中的函数作为参数。如果指定被测函数，则用指定的函数替代前面的列表。函数 timexN()除重复测量函数的执行时间之外，其他的工作方式与 timex()一样。

4．例子

函数 timex()用于获取 myFunc 函数的单次执行时间：

```
-> timex myFunc, myArg1, myArg2, ...
```

函数 timexN()反复执行 myFunc 函数直到测量误差小于 2%：

```
-> timexN myFunc, myArg1, myArg2, ...
```

函数 timexPre()、timexPost()和 timexFunc()增加一组需要执行的函数列表：

```
-> timexPre 0, myPreFunc1, preArg1, preArg2, ...
-> timexPre 1, myPreFunc2, preArg1, preArg2, ...


-> timexFunc 0, myFunc1, myArg1, myArg2, ...
-> timexFunc 1, myFunc2, myArg1, myArg2, ...
-> timexFunc 2, myFunc3, myArg1, myArg2, ...


-> timexPost 0, myPostFunc, postArg1, postArg2, ...
```

该列表传递给无参数的 timex()或者 timexN()：

```
-> timex
```

或者

```
-> timexN
```

在该例中，myPreFunc1 和 myPreFunc2 根据各自参数调用。然后系统依次调用 myFunc1、myFunc2 和 myFunc3。如果使用函数 timexN()，系统将会循环调用这三个函数直到测量误差小于 2%。最后，系统会调用 myPostFunc。在所有的函数执行之后，系统会回馈函数执行时间信息。

5．注意

执行时间测量的是函数体的执行时间，不包括通常的子程序入口和退出代码时间。也就是说，安装参数

和调用程序不包含在所报告的执行时间中。

6. 头文件

执行计时器函数声明在 timexLib.h 头文件中。

7. 参考

相关信息请参考 spyLib 库描述。

3.4.2 执行计时器函数详细描述

timexInit()

函数原型：

```
void timexInit (void)
```

功能描述：

包含执行计时器库。该函数把 timexLib 库链接到 VxWorks 系统中。如果用户定义了配置宏 INCLUDE_TIMEX，则 usrConfig.c 文件中的 usrRoot()将会调用这个函数。

返回值：

无。

参考：

相关信息请参考 timexLib 库描述。

例：

```
void usrRoot
    (
    char *   pMemPoolStart,          /* 系统内存区起始点 */
    unsigned      memPoolSize        /* 内存池初始大小 */
    )
{
...
#ifdef INCLUDE_TIMEX
    timexInit ();              /* 安装执行计时器库 */
#endif/* INCLUDE_TIMEX */
...
}
```

timexClear()

函数原型：

```
void timexClear (void)
```

功能描述：

该函数清除当前被测执行时间的函数列表。

返回值：

无。

参考：

相关信息请参考 timexLib 库描述。

timexFunc()

函数原型：

```
void timexFunc
```

```
(
int i,              /* 列表中的函数编号 (0，1，2，3) */
FUNCPTR func,  /* 被添加的函数 (如果删除函数则为 NULL) */
int arg1,           /* 以下是传递给给函数的 8 个参数 */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8
)
```

功能描述：

指定被测执行时间的函数。这个函数在被测函数列表中添加或删除函数，并把列表中的函数作为一个整体通过调用 timex()或 timexN()来测量它们的执行时间。被测函数列表中最多可以包含 4 个函数。参数 i 指定函数在执行顺序中的位置。通过指定函数的序列号 i 和设置函数参数 func 为 NULL，则可以从列表中删除序列号为 i 的函数。

返回值：

无。

参考：

相关信息请参考 timexLib 库描述，以及 timex()和 timexN()函数。

例：

```
void timexFunc_demo (void)
{
/* 指定一个被测执行时间的函数 */
timexFunc(0, (FUNCPTR) RunDemo,0,0,0,0,0,0,0,0);
}
…
/* 被测执行时间的函数 */
void RunDemo (void)
{
int i;
for(i=0;i<200;i++)
printf("Testing the function execute time!\n");
}
```

timexHelp()

函数原型：

```
void timexHelp (void)
```

功能描述：

显示执行计时器模块的纲要。该函数显示下面可用的执行计时器函数的纲要：

```
timexHelp                    Print this list.
timex       [func,[args...]]    Time a single execution.
```

```
timexN          [func,[args...]]   Time repeated executions.
timexClear                         Clear all functions.
timexFunc   i,func,[args...]       Add timed function number i (0,1,2,3).
timexPre    i,func,[args...]       Add pre-timing function number i.
timexPost   i,func,[args...]       Add post-timing function number i.
timexShow                          Show all functions to be called.
...
```

返回值：

无。

参考：

相关信息请参考 timexLib 库描述。

timex()

函数原型：

```
void timex
(
FUNCPTR func,  /* 被测函数 */
int arg1,      /* 以下是传递给函数的 8 个参数 */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8
)
```

功能描述：

测量函数单次执行的时间。该函数测量最多可带 8 个参数的函数单次执行的时间。如果没有指定函数，则测量当前被测函数列表中函数的执行时间，而这些列表是通过调用 timexFunc()、timexPre()和 timexPost()函数创建的。如果指定被测函数，则用指定的函数替代整个当前列表。

当执行完成时，timex()显示函数的执行时间和测量误差。如果被测函数执行太快，以至于比时钟频率还快，测得执行时间就没有意义(误差>50%)，这时会显示一个警告信息。对于这种情况，应该使用 timexN()函数测量函数多次执行的时间。

返回值：

无。

参考：

相关信息请参考 timexLib 库描述，以及 timexFunc()、timexPre()、timePost()和 timexN()函数。

例：

```
void timex_demo(void)
{
/* 测量函数单次执行的时间 */
timex((FUNCPTR)RunDemo,0,0,0,0,0,0,0,0);
}
...
```

```
/* 被测执行时间的函数 */
void RunDemo (void)
{
int i;
for(i=0;i<200;i++)
printf("Testing the function execute time!\n");
}
```

timexN()

函数原型：

```
void timexN
(
FUNCPTR func, /* 被测函数 */
int arg1,     /* 以下是传递给函数的 8 个参数 */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8
)
```

功能描述：

重复测量一个或一组函数执行的时间。这个函数除重复测量函数的执行时间之外，其他则与 timex()的方式一样测量当前被测函数列表中函数的执行时间；然而，函数的执行次数是一个不定值，直到测得的执行时间的误差小于 2%为止。

返回值：

无。

参考：

相关信息请参考 timexLib 库描述，以及 timexFunc()和 timex()函数。

例：

```
void timexN_demo(void)
{
/* 重复测量一个或一组函数执行的时间，直到测得的执行时间的误差小于 2%为止。*/
timexN((FUNCPTR)RunDemo,0,0,0,0,0,0,0,0);
}
…
/* 被测执行时间的函数 */
void RunDemo (void)
{
int i;
for(i=0;i<100;i++)
printf("Testing the function execute time!\n");
}
```

timexPost()

函数原型：

```
void timexPost
(
int i,              /* 列表中的函数编号 (0，1，2，3) */
FUNCPTR func,       /* 被添加的函数 (如果删除函数则为 NULL ) */
int arg1,           /* 以下是传递给函数的 8 个参数 */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8
)
```

功能描述：

指定在测量函数执行时间之后将被调用的函数。这个函数在函数列表中添加或删除函数，一旦测量函数执行时间的操作完成后，系统立即调用函数列表中的函数。列表中最多可以包含 4 个函数，且每个函数最多可传递 8 个参数。

返回值：

无。

参考：

相关信息请参考 timexLib 库描述。

例：

```
void timexPost_demo (void)
{
/* 指定在测量函数执行时间之后将被调用的函数 */
timexPost(0, (FUNCPTR)timex_end,0,0,0,0,0,0,0,0);
}
...
/* 在测量函数执行时间之后将被调用的函数 */
void timex_end (void)
{
printf("Test the function execute time ended!\n");
}
```

timexPre()

函数原型：

```
void timexPre
(
int i,              /* 列表中的函数编号 (0，1，2，3) */
FUNCPTR func,       /* 被添加的函数 (如果删除函数则为 NULL ) */
int arg1,           /* 以下是传递给函数的 8 个参数 */
```

```
    int arg2,
    int arg3,
    int arg4,
    int arg5,
    int arg6,
    int arg7,
    int arg8
    )
```

功能描述：

指定在测量函数执行时间之前将被调用的函数。这个函数在函数列表中添加或删除函数，在开始测量函数执行时间，系统首先调用函数列表中的函数，然后才测量时间。列表中最多可以包含4个函数，且每个函数最多可传递8个参数。

返回值：

无。

参考：

相关信息请参考 timexLib 库描述。

例：

```
void timexPre_demo (void)
{
/* 指定一个在测量函数执行时间之前将被调用的函数 */
timexPre(0, (FUNCPTR)timex_begin,0,0,0,0,0,0,0,0);
}
...
/* 在测量函数执行时间之前将被调用的函数 */
void timex_begin (void)
{
printf("Begin test the function execute time!\n");
}
```

timexShow()

函数原型：

```
    void timexShow (void)
```

功能描述：

显示被测执行时间的函数列表。该函数显示当前被测执行时间的函数列表。而这些列表是通过调用 timexPre()、timexFunc()和 timexPost()函数来创建的。

返回值：

无。

参考：

相关信息请参考 timexLib 库描述，以及 timexPre()、timexFunc()和 timexPost()函数。

第4章 中断管理

4.1 与体系结构无关的中断函数

4.1.1 函数库描述

表 4-1 与体系结构无关的中断函数

函 数	描 述
intContext()	确定当前状态在中断上下文中还是在任务上下文中
intCount()	获得当前中断嵌套深度

1. 库命名

与体系结构无关的中断函数库名称为 intLib。

2. 函数

表 4-1 中列出了与体系结构无关的中断函数。

3. 描述

intLib 库为中断提供通用的程序。通过调用 intConnect()函数可以将任何 C 语言程序与任何中断(陷阱)关连，而该函数由 intArchLib 库提供。intCount()和 intContext()函数用来确定 CPU 正运行在中断上下文中还是在一般的任务上下文中。有关与体系结构相关的中断处理，请参考 intArchLib 库描述。

4. 头文件

与体系结构无关的中断函数声明在 intLib.h 头文件中。

5. 参考

相关信息请参考 intArchLib 库描述，以及“VxWorks Programmer's Guide”中的基本内核章节。

4.1.2 与体系结构无关的中断函数详细描述

intContext()

函数原型：

```
BOOL intContext (void)
```

功能描述：

确定当前状态在中断上下文中还是在任务上下文中。

返回值：

如果当前执行状态是在中断上下文中则返回 TRUE，其他则返回 FALSE。

参考：

相关信息请参考 intLib 库描述。

例：

```
BOOL intMode;
...
intMode = intContext ();    /* 确定当前状态在中断上下文中是否 */

if(intMode == TRUE)
  logMsg ("current state is in interrupt context!\n");
else
  logMsg ("current state is in task context!\n");
...
```

intCount()

函数原型：

int intCount (void)

功能描述：

获得当前中断的嵌套深度。该函数返回当前中断的嵌套深度。

返回值：

嵌套的深度。

参考：

相关信息请参考 intLib 库描述。

4.2 与体系结构相关的中断函数

4.2.1 函数库描述

1．库命名

与体系结构相关的中断函数库名称为 intArchLib。

2．函数

表 4-2 中列出了与体系结构相关的中断函数。

表 4-2 与体系结构相关的中断函数

函 数	描 述
intLevelSet()	设置中断级(MC680x0、SPARC、i960、x86、ARM)
intLock()	锁(关)中断
intUnlock()	解锁(开)中断
intEnable()	打开相应的中断位(MIPS、PowerPC、ARM)
intDisable()	屏蔽相应的中断位(MIPS、PowerPC、ARM)
intCRGet()	读 cause 寄存器(MIPS)
intCRSet()	写 cause 寄存器(MIPS)
intSRGet()	读状态寄存器(MIPS)
intSRSet()	更新状态寄存器(MIPS)
intConnect()	将 C 语言的函数连接到中断上
intHandlerCreate()	为 C 语言的函数构造中断服务程序（ISR）(MC680x0、SPARC、i960、x86、MIPS)
intLockLevelSet()	设置当前中断屏蔽级(MC680x0、SPARC、i960、x86、ARM)
intLockLevelGet()	获得当前中断屏蔽级(MC680x0、SPARC、i960、x86、ARM)
intVecBaseSet()	设置中断向量(陷阱)基地址(MC680x0、SPARC、i960、x86、MIPS、ARM)
intVecBaseGet()	获得中断向量(陷阱)基地址(MC680x0、SPARC、i960、x86、MIPS、ARM)
intVecSet()	设置 CPU 向量(陷阱)(MC680x0、SPARC、i960、x86、MIPS)
intVecGet()	获得中断处理程序(MC680x0、SPARC、i960、x86、MIPS)
intVecTableWriteProtect()	对异常向量表写保护(MC680x0、SPARC、i960、x86、ARM)
intUninitVecSet()	对未初始化的向量安装处理程序(ARM)

3．描述

intArchLib 库提供与体系结构相关的函数来处理和连接硬件中断。通过调用 intConnect()函数可以将任何 C 语言函数与任何中断关连。通过函数 intVecSet()和 intVecGet()可以直接访问向量。通过函数 intVecBaseGet()可以访问向量(陷阱)基址寄存器(如果存在的话)。

任务通过调用 intLock()和 intUnlock()函数可以禁止和打开中断。通过函数 intLockLevelSet()和 intLockLevelGet()可以设置和获得中断屏蔽级(只对 MC680x0、SPARC、i960、x86、MIPS、ARM 体系结构有效)。函数 intLevelSet()改变处理器的当前中断级。

4．警告

在中断被禁止后，不要调用 VxWorks 系统函数。违反这个规则可能会重新打开不可预见的中断。

5．中断向量和中断号

该函数库中的大部分函数采用中断向量作为参数，它通常是向量表的字节偏移。在中断向量和中断号之间通过宏进行转换：

IVEC_TO_INUM(intVector)：将中断向量转换成中断号；

INUM_TO_IVEC(intNumber)：将中断号转换成中断向量；

TRAPNUM_TO_IVEC (trapNumber)：将陷阱号转换成中断向量。

6．头文件

与体系结构相关的中断函数声明及相关宏定义在 intLib.h 及 iv.h 头文件中。

7．参考

相关信息请参考 intLib 库描述。

4.2.2 与体系结构相关的中断函数详细描述

intLevelSet()

函数原型：

```
int intLevelSet
(
int level /* 中断级掩码 */
)
```

功能描述：

设置中断级。该函数用 level 指定的值改变状态寄存器中的中断掩码。在这个级别或在这个级别下的中断将被屏蔽。level 的值必须处于下面范围：

MC680x0: 0～7；

SPARC: 0～15；

i960: 0～31；

ARM：由 BSP 指定。

对于 SPARC 系统，在调用前必须打开陷阱。

返回值：

先前的中断级。

参考：

相关信息请参考 intArchLib 库描述。

intLock()

函数原型：

```
int intLock (void)
```

功能描述：

锁(关)中断。该函数禁止中断。intLock()函数返回一个与体系结构相关的屏蔽关键字(lock-out key)表示调用之前的中断级：这个屏蔽关键字可以被传递给 intUnlock()函数以便重新打开中断。

对于 MC680x0、SPARC、i960 和 x86 体系结构，该函数禁止 intLockLevelSet()设置的中断。默认中断屏蔽级是最高的中断级(MC680x0=7、APARC=15、i960=31、i386/i486=1)。

对于 MIPS 处理器，该函数禁止最高级别的中断。这意味着即使不屏蔽状态寄存器的中断屏蔽位(15～8)，中断也不会发生。

对于 ARM 处理器，该函数通过设置 CPSR 中的 I 位来屏蔽中断请求。这意味着没有中断请求发生。

对于 PowerPC 处理器，只有惟一的一个中断向量。当调用 intLock()函数时，系统将会屏蔽外部中断(向量偏移 0x500)。这意味着任何外部事件都不会中断处理器。

屏蔽关键字：

体系结构不同，其屏蔽关键字也不同。如下所示：

MC680x0: 中断屏蔽码；
SPARC: 中断级(0～15)；
i960: 中断级 (0～31)；
MIPS: 状态寄存器；
i386/i486: EFLAGS 寄存器中的中断使能标识(IF)位；
PowerPC: MSR 寄存器的值；
ARM: CPSR 中的 I 位。

警告：

在中断被禁止后，不要调用 VxWorks 系统函数。违反这个规则可能会重新打开不可预见的中断。

在中断级或任务级中都可以调用 intLock()函数。当在任务上下文中调用时，中断屏蔽级是任务上下文的一部分。禁止中断并不会阻止任务调度。这样，如果一个任务禁止中断，但调用内核服务导致任务阻塞(例如：taskSuspend()或 taskDelay())或导致更高优先级的任务处于就绪态(例如：semGive()或 taskResume())，那么将会发生任务调度，并且在其他任务运行的时候会打开中断。通过调用 taskLock()函数可以禁止任务调度，但在调用这个函数之前，必须打开陷阱。

返回值：

与体系结构相关的屏蔽关键字。

参考：

相关信息请参考 intArchLib 库描述，intUnlock()、taskLock()和 intLockLevelSet()函数。

例：

```
int   count;        /* 一个全局计数 */
void intLock_demo (void)
    {
    int lockKey;

    /* 锁中断 */
    lockKey = intLock();
    ++count;    /* 需要保护的临界资源 */
    /* 开中断 */
    intUnlock(lockKey);
    }
```

intUnLock()

函数原型：

```
void intUnlock
(
int lockKey /* intLock()返回的屏蔽关键字 */
)
```

功能描述：

解锁(开)中断。该函数重新打开被 intLock()禁止的中断。参数 lockKey 是先前 intLock()返回的一个与体系结构相关的屏蔽关键字。

返回值：

无。

参考：

相关信息请参考 intArchLib 库描述及 intLock()函数。

intEnable()

函数原型：

```
int intEnable
(
int level /* 中断位(0x00 - 0xff00) */
)
```

功能描述：

打开相应的中断位。该函数打开 MIPS 或 PowerPC 处理器的状态寄存器上，输入的中断位。

注意：

ARM 处理器通常没有片上中断控制器。中断的控制与具体的板级支持包（BSP）有关。该函数调用一个具体的 BSP 程序来打开中断。对于每个中断级，在中断允许之前必须调用该函数。

对于 MIPS，level 是 SR_IBIT1～SR_IBIT8 的一个组合。

返回值：

成功打开则返回 OK，如果失败则返回 ERROR（对于 MIPS 架构来说，调用失败则返回状态寄存器先前的内容）。

参考：

相关信息请参考 intArchLib 库描述。

例：

```
#define          LEVEL    0xff00
int              status;
...
/* 打开相应的中断位 */
status = intEnable (LEVEL);
if(status == ERROR)
    printf("Error in enabling corresponding interrupt bit!\n");
...
```

intDisable()

函数原型：

```
int intDisable
(
int level /* 中断位 (0x0 - 0xff00) */
```

)

功能描述：

屏蔽相应的中断位。对于 MIPS 和 PowerPC 体系结构，该函数屏蔽状态寄存器中相应的中断位。

注意：

ARM 处理器通常没有片上中断控制器。中断的控制与具体的 BSP 有关。该函数调用一个具体的 BSP 程序来屏蔽一个特殊的中断级，而不管当前中断屏蔽级。

对于 MIPS，可以设置宏定义 SR_IBIT1～SR_IBIT8。

返回值：

成功屏蔽则返回 OK，如果失败则返回 ERROR（对于 MIPS 架构来说，调用失败则返回状态寄存器先前的内容）。

参考：

相关信息请参考 intArchLib 库描述。

例：

```
#define          LEVEL    0x0
int              status;
...
/* 屏蔽相应的中断位 */
status = intDisable (LEVEL);
if(status == ERROR)
    printf("Error in disabling corresponding interrupt bit!\n");
...
```

intCRGet()

函数原型：

```
int intCRGet (void)
```

功能描述：

读 cause 寄存器。该函数从 MIPS cause 寄存器中读取内容，并返回读取的内容。

返回值：

cause 寄存器中的内容。

参考：

相关信息请参考 IntArchLib 库描述。

intCRSet()

函数原型：

```
void intCRSet
(
int value    /* 往 cause 寄存器中写入的值 */
)
```

功能描述：

写 cause 寄存器。该函数把 value 值写入到 MIPS cause 寄存器中。

返回值：

无。

参考：

相关信息请参考 intArchLib 库描述。

intSRGet()

函数原型：

```
int intSRGet (void)
```

功能描述：

读 MIPS 状态寄存器。该函数读取 MIPS 状态寄存器中的内容，并返回读取的内容。

返回值：

状态寄存器的内容。

参考：

相关信息请参考 intArchLib 库描述。

intSRSet()

函数原型：

```
int intSRSet
(
int value /* 往状态寄存器中写入的值*/
)
```

功能描述：

更新状态寄存器。该函数把 value 值写入到 MIPS 状态寄存器中，并返回状态寄存器先前的内容。

返回值：

状态寄存器先前的内容。

参考：

相关信息请参考 intArchLib 库描述。

例：

```
int         statusReg;
...
statusReg = intSRGet ();

/* 写 cause 寄存器 */
printf("Set the status register!\n");
causeReg = intSRSet (statusReg | 0xAA);
...
```

intConnect()

函数原型：

```
STATUS intConnect
(
VOIDFUNCPTR * vector,    /* 中断向量 */
VOIDFUNCPTR routine,     /* 函数 */
int parameter            /* 传递给函数的参数 */
)
```

功能描述：

将 C 语言的函数连接到中断上。该函数把指定的 C 语言函数连接到指定的中断向量上。routine 的地址

通常存储在 vector 上，当中断发生时，系统将调用带参数 parameter 的 routine。系统在超级模式的中断级中调用 routine 函数，并且系统会建立一个正确的 C 环境，保存必要的寄存器并安装堆栈。

routine 函数可以是任何普通的 C 代码，但它不能是调用阻塞或执行 I/O 操作的系统函数。

返回值：

成功连接则返回 OK，如果不能建立中断处理程序则返回 ERROR。

参考：

相关信息请参考 intArchLib 库描述,intHandlerCreate()和 intVecSet()函数。

例：

```
int                 intNum;
SEM_ID              semId;              /* 信号量 ID */
STATUS              st;
...
intNum = 0x28;
/* 安装中断服务程序 */
st = intConnect ((VOIDFUNCPTR *) INUM_TO_IVEC (intNum), (VOIDFUNCPTR) intHdlr,semId);
if (st == ERROR)
        {
        perror ("Error in connecting to the ISR");
        return;
        }
...
/* 中断服务程序*/
void intHdlr
    (
    SEM_ID semId
    )
    {
     int level = intLock();
    logMsg ("acknowledging interrupt\n",0,0,0,0,0,0);
    semGive (semId);
     intUnlock(level);
    }
```

intHandlerCreate()

函数原型：

```
    FUNCPTR intHandlerCreate
    (
    FUNCPTR routine,      /* 函数 */
    int parameter         /* 传递给函数的参数 */
    )
```

功能描述：

为 C 语言函数构造中断服务程序。该函数为指定的 C 语言函数建立中断服务程序，这样可以通过调用 intVecSet()函数把这个 ISR 与指定的向量地址关联。系统在超级模式的中断级中调用这个 ISR，并且系统会建立正确的 C 环境，保存必要的寄存器并安装堆栈。

routine 函数可以是任何普通的 C 代码，但它不能是调用阻塞或执行 I/O 操作的系统函数。

返回值：

一个指向新 ISR 的指针，如果内存不够则返回 NULL。

参考：

相关信息请参考 intArchLib 库描述。

例：

```
FUNCPTR newISR;
...
/* 把函数构造成一个 ISR */
newISR = intHandlerCreate((FUNCPTR)intHdlr,0);
if(newISR == NULL)
    printf ("Error in construct an ISR!\n");
...
/* 中断服务程序*/
void intHdlr (void)
    {
    logMsg ("acknowledging interrupt\n",0,0,0,0,0,0);
    }
```

intLockLevelSet()

函数原型：

```
    void intLockLevelSet
    (
    int newLevel /* 中断级 */
    )
```

功能描述：

设置当前中断屏蔽级。该函数设置当前中断屏蔽级并把它存储在全局变量 intLockMask 中。当调用 intLock()禁止中断时，系统屏蔽指定级别的中断。在初始化 VxWorks 的时候，系统通过调用 kernelInit()初始化默认屏蔽级(MC680x0 = 7、SPARC = 15、i960 = 31、i386/i486 =1)。

返回值：

无。

参考：

相关信息请参考 intArchLib 库描述,intLockLevelGet()、intLock()和 taskLock()函数。

intLockLevelGet()

函数原型：

```
    int intLockLevelGet (void)
```

功能描述：

获得当前中断屏蔽级。该函数返回由 intLockLevelSet()设置并存储在全局变量 intLockMask 中的当前中断屏蔽级。

返回值：

存储在中断屏蔽码（intLockMask）中的当前中断级(对于 ARM 体系结构，则永远返回 ERROR)。

参考：

相关信息请参考 intArchLib 库描述及 intLockLevelSet()函数。

intVecBaseSet()

函数原型：

```
void intVecBaseSet
(
FUNCPTR * baseAddr /* 中断向量(trap)基地址 */
)
```

功能描述：

设置中断向量(陷阱)基地址。该函数设置 CPU 的中断向量基址寄存器为一指定的值 baseAddr ，函数 intVecGet()和 intVecSet()将使用这个基地址。中断向量基地址最初为 0(SPARC 为 0x10000)，直到调用该函数修改它的值。

注意：

在 SPARC 处理器上，向量基地址必须是以 4KB 为边界(也就是说，它的末端 12 位必须是 0)。

68000、MIPS 和 ARM 处理器没有中断向量基址寄存器，所以该函数对这些系统无效。

该函数对 i960 系统也无效。中断向量表位于文件 sysLib.c 中，通过调用 intVecBaseSet()移动它将需要重新复位处理器。并且，向量基址隐藏在 PRCB 中的片内，因而该函数不能设置中断向量基地址。

返回值：

无。

参考：

相关信息请参考 intArchLib 库描述，intVecBaseGet()、intVecGet()和 intVecSet()函数。

intVecBaseGet()

函数原型：

```
FUNCPTR *intVecBaseGet (void)
```

功能描述：

获得中断向量(陷阱)基地址。该函数返回由函数 intVecBaseSet()设置的当前中断向量基地址。

返回值：

当前中断向量基地址(对于 i960 系统为 sysLib.c 中 sysIntTable 的值；而 MIPS 和 ARM 系统则总是为 0)。

参考：

相关信息请参考 intArchLib 库描述及 intVecBaseSet()函数。

intVecSet()

函数原型：

```
void intVecSet
(
FUNCPTR * vector,    /* 向量偏移量 */
FUNCPTR function     /* 放置在向量中的地址 */
)
```

功能描述：

该函数把异常/中断/陷阱处理程序连接到指定的向量上。指定的向量作为一个偏移量存放在 CPU 的向量表中。向量表的起始点默认值是：

MC680x0: 0

SPARC: 0x1000

i960: sysIntTable(在 sysLib.c 中)

MIPS: excBsrTbl(在 excArchLib.c 中)

i386/i486: 0

然而，向量表的起始点可能被 intVecBaseSet()设置在任何地址(CPU 能访问的)。在函数 usrInit()中安装向量表。

返回值：

无。

参考：

相关信息请参考 intArchLib 库描述，intVecBaseSet()和 intVecGet()函数。

例：

```
int                 intNum;
...
intNum = 0x0A;

/* 把异常/中断/陷阱处理程序连接到指定的向量上 */
intVecSet ((FUNCPTR *) INUM_TO_IVEC (intNum),( FUNCPTR) intHdlr);
...
/* 中断服务程序*/
void intHdlr (void)
    {
    int level = intLock();
    logMsg ("acknowledging interrupt!\n",0,0,0,0,0,0);
    intUnlock(level);
    }
```

intVecGet()

函数原型：

```
FUNCPTR intVecGet
(
FUNCPTR * vector /* 向量偏移量 */
)
```

功能描述：

获得中断处理程序。该函数返回一个与指定向量所关联的异常/中断处理程序。

注意：

对于 i960 体系结构，在系统启动后，重新初始化中断表位置并存放在 sysIntTable 中。通过 intVecBaseGet()函数可以获得该表的位置。

返回值：

一个指向异常/中断处理程序的指针。

参考：

相关信息请参考 intArchLib 库描述，intVecSet()、intVecBaseGet()和 intVecBaseSet()函数。

例：

```
int                 intNum;
FUNCPTR             ISR_function;
...
intNum = 0x0A;
```

```
/* 获得中断处理程序 */
printf("Get the interrupt handler to the specified vector: %d\n", intNum);
ISR_ function = intVecGet((FUNCPTR *) INUM_TO_IVEC (intNum));
...
```

intVecTableWriteProtect()

函数原型：

```
STATUS intVecTableWriteProtect (void)
```

功能描述：

对异常向量表进行写保护。如果系统包含了存储器管理单元(MMU——Memory Management Unit)支持包(VxVMI)，该函数对异常向量表进行写保护，防止它被意外破坏。

注意：

包含在页中的其他数据结构也将被保护。在 VxWorks 默认配置中，异常向量表被放置在内存中 0 的位置。写保护会影响底板定位、引导配置信息、可能的正文段(假设默认正文位置为 0x1000)。在操作期间，处理这些结构的所有代码已经对可写内存进行了修改。如果用户为异常向量表选择一个不同的地址，必须确认它驻留在一个与其他可写数据结构分开的页中。

返回值：

操作成功则返回 OK，如果内存不可以被写保护则返回 ERROR。

错误码：

S_intLib_VEC_TABLE_WP_UNAVAILABLE

参考：

相关信息请参考 intArchLib 库描述。

intUninitVecSet()

函数原型：

```
void intUninitVecSet
(
VOIDFUNCPTR routine /* 指向用户函数的指针 */
)
```

功能描述：

为未初始化的向量安装处理程序。该函数为未初始化向量安装处理程序，当进入任何未初始化向量时，系统将调用。

返回值：

无。

参考：

相关信息请参考 intArchLib 库描述。

第 5 章　VxWorks 内存管理

5.1　完全内存区管理函数

5.1.1　函数库描述

1．库命名

完全内存区管理函数库名称为 memLib。

2．函数

表 5-1 中列出了完全内存区管理函数。

表 5-1　完全内存区管理函数

函　数	描　述
memPartOptionsSet()	为内存分区设置调试选项
memalign()	分配一块对齐的内存
valloc()	从页边界开始分配内存
memPartRealloc()	在指定的内存分区中重新分配一块内存
memPartFindMax()	寻找最大的空闲内存块
memOptionsSet()	为系统内存分区设置调试选项
calloc()	分配一组内存空间(ANSI)
realloc()	改变指定内存块的大小(ANSI)
cfree()	释放一块内存
memFindMax()	在系统内存分区中查找最大的未使用内存块

3．描述

完全内存区管理库为分配和管理内存分区提供了全面的功能函数。该函数库是 memPartLib 函数库的扩展，提供了更强的功能，其中包括错误处理、对齐内存分配、兼容 ANSI 分配函数接口等。相关内容可以参考 memPartLib 函数库的说明。

当函数 kernelInit()初始化系统内核时，同时创建了内存分区。系统分区的 ID 被存储在全局变量 memSysPartId 中。

这里值得一提的是 memalign()函数，它提供了按指定边界分配内存的功能。

该库中还包含了与 ANSI 兼容的三个函数：calloc()函数用于为一组数据分配内存空间；realloc()函数用于改变指定内存块的大小；cfree()函数释放由 calloc()所分配的内存。

4．错误说明

用户可以使用函数 memPartOptionsSet()以及函数 memOptionsSet()为指定的内存分区设置调试选项。系统可以检测到两类错误：试图分配的内存大于剩余内存、释放内存的时候发现错误的内存块。在出现了这两类情况后，系统将会返回错误状态，下面是四种错误处理的方法，可以分别组合使用：

- **MEM_ALLOC_ERROR_LOG_FLAG**

当分配内存出现错误时记录一条信息；

- **MEM_ALLOC_ERROR_SUSPEND_FLAG**

当分配内存出现错误时挂起任务（但是使用选项 VX_UNBREAKABLE 时，任务不会被挂起）；

- **MEM_BLOCK_ERROR_LOG_FLAG**

当释放内存出现错误时记录一条信息；

- **MEM_BLOCK_ERROR_SUSPEND_FLAG**

当释放内存出现错误时挂起任务（但是使用选项 VX_UNBREAKABLE 时，任务不会被挂起）。

当使用下面的选项来检测内存分区的空闲内存时，memPartLib 库中的 memPartFree()函数和 free()函数会进行一致性检查。如果未指定这个选项，释放内存时则不会进行任何检查。

MEM_BLOCK_CHECK

检查每一个被释放的内存块。设置任何一个 MEM_BLOCK_ERROR 选项，则自动连带设置该选项。

创建内存分区时的默认设置是：

MEM_ALLOC_ERROR_LOG_FLAG

MEM_BLOCK_CHECK

MEM_BLOCK_ERROR_LOG_FLAG

MEM_BLOCK_ERROR_SUSPEND_FLAG

当使用 memPartOptionsSet()或 memOptionsSet()为内存分区设置选项时，应当使用逻辑或组合相应的选项，例如：

```
memPartOptionsSet (myPartId, MEM_ALLOC_ERROR_LOG_FLAG |MEM_BLOCK_CHECK |MEM
_BLOCK_ERROR_LOG_FLAG);
```

5．头文件

完全内存区管理函数声明在 memLib.h 头文件中。

6．参考

相关信息请参考 memPartLib、smMemLib 库描述。

5.1.2　完全内存管理函数详细描述

memPartOptionsSet ()

函数原型：

```
STATUS memPartOptionsSet
(
PART_ID partId,        /* 设置选项的内存块 */
unsigned options       /* 内存管理选项 */
)
```

功能描述：

该函数为指定的内存分区设置调试选项。调试时可以监测到两类错误：试图分配的内存大于剩余内存和释放内存的时候发现错误的内存块。如果发生这两种情况，函数将会返回错误状态。下面是四种错误处理的方法，可以分别组合使用：

- **MEM_ALLOC_ERROR_LOG_FLAG：**

当分配内存出现错误时记录一条信息；

- **MEM_ALLOC_ERROR_SUSPEND_FLAG：**

当分配内存出现错误时挂起任务（但是使用选项.VX_UNBREAKABLE 时，任务不会被挂起）；

- **MEM_BLOCK_ERROR_LOG_FLAG：**

当释放内存出现错误时记录一条信息；

- **MEM_BLOCK_ERROR_SUSPEND_FLAG：**

当释放内存出现错误时挂起任务（但是使用选项 VX_UNBREAKABLE 时，任务不会被挂起）。

返回值：

成功设置选项则返回 OK，如果设置选项失败则返回 ERROR。

错误码：

S_smObjLib_NOT_INITIALIZED

参考：

相关信息请参考 memLib、smMemLib 库描述。

memalign ()

函数原型：

```
void *memalign
(
unsigned alignment,    /* 内存边界 (2 的幂数) */
unsigned size          /* 分配字节数 */
)
```

功能描述：

该函数在系统内存分区中分配 size 参数指定大小的缓冲区。而且，该函数会确保所分配的缓冲区以指定的方式对齐。对齐的参数必须是 2 的幂数。

返回值：

指向新分配的内存块的指针，如果不能分配内存则返回 NULL。

参考：

相关信息请参考 memLib 库描述。

valloc ()

函数原型：

```
void * valloc
(
unsigned size      /* 分配字节数 */
)
```

功能描述：

该函数从系统内存分区中分配 size 参数指定大小的缓冲区。而且，该函数会确保所分配的缓冲区对齐于一个内存页。内存页的大小决定于体系结构。

返回值：

指向所分配内存的指针，如果未能分配内存或者内存管理单元支持库未初始化则返回 NULL。

错误码：

S_memLib_PAGE_SIZE_UNAVAILABLE

参考：

相关信息请参考 memLib 库描述。

memPartRealloc ()

函数原型：

```
void *memPartRealloc
(
PART_ID partId,        /* 内存块 ID */
char * pBlock,         /* 重新分配的内存块 */
unsigned nBytes        /* 新内存块字节大小 */
)
```

功能描述：

该函数改变指定内存块的大小，返回指向新内存块的指针。新内存块中原有的内容不会改变，但是内存对齐的方式可能会与原来的内存块不同。

如果参数 pBlock 为 NULL，则该调用等效于 memPartAlloc()函数。

返回值：
指向新内存块的指针，如果调用失败则返回 NULL。
错误码：
S_smObjLib_NOT_INITIALIZED
参考：
相关信息请参考 memLib、smMemLib 库描述。
memPartFindMax ()
函数原型：
int memPartFindMax
(
PART_ID partId /* 内存块 ID */
)

功能描述：
该函数在内存分区的空闲块列表中查找最大的块，然后返回其大小。
返回值：
以字节计算的空闲内存块的大小。
错误码：
S_smObjLib_NOT_INITIALIZED
参考：
相关信息请参考 memLib、smMemLib 库描述。
例：

```
int          sizenum;
PART_ID mypartId;
...
/* 在内存分区的空闲块列表中查找最大的内存块 */
sizenum = memPartFindMax(mypartId);
printf("The size of the largest available free block: %d\n",sizenum);
...
```

memOptionsSet ()
函数原型：
void memOptionsSet
(
unsigned options /* 系统内存分区的调试选项 */
)

功能描述：
该函数为系统内存分区设置调试选项。调试时可以监测到两类错误：试图分配的内存多于剩余内存和释放内存的时候发现错误的内存块。如果发生这两种情况，可以令函数做出下面的反映：1）返回错误状态。2）记录错误信息并且返回。3）记录错误信息并且挂起对应的任务。
返回值：
无。
参考：
相关信息请参考 memLib 库描述及 memPartOptionsSet()函数。

calloc ()

函数原型：

```
void *calloc
(
size_t elemNum, /* 单元数 */
size_t elemSize   /* 单元字节大小 */
)
```

功能描述：

该函数为一组数分配对应的内存块，所分配的空间内元素被初始化为零。

返回值：

指向内存块的指针，如果出现错误返回 NULL。

参考：

相关信息请参考 memLib 库描述。

例：

```
typedef struct {
    float pos;
    float vel;
    float acc;
    SEM_ID sampleSemaphore;
    int scopeIndex;
} *ControlInfo;
ControlInfo TheCI;
...
/* 分配内存块 */
TheCI = (ControlInfo) calloc(1,sizeof(*TheCI));
if(TheCI == NULL)
    printf("allocate space failed!\n");
...
```

realloc ()

函数原型：

```
void *realloc
(
void * pBlock, /* 需要重新分配的内存块 */
size_t newSize /* 新内存块大小 */
)
```

功能描述：

该函数改变指定内存块的大小，返回指向新内存块的指针。新内存块中原有的内容不会改变，但是内存对齐的方式可能会与原来的内存块不同。

返回值：

指向新的内存块的指针，如果调用失败则返回 NULL。

参考：

相关信息请参考 memLib 库描述。

cfree ()

函数原型：

```
STATUS cfree
(
char * pBlock /*   指向内存块的指针  */
)
```

功能描述：

该函数将先前由 calloc()分配的内存块释放到空闲内存池。如果释放未经分配的内存则会返回错误。

返回值：

成功释放内存块则返回 OK，如果指定的内存块无效则返回 ERROR。

错误码：

无。

参考：

相关信息请参考 memLib 库描述。

例：

```
int             status;
char   * pMemArea;    /*  指向字符串的指针  */
...
/*  释放分配的内存  */
status = cfree (pMemArea);
if(status == ERROR)
   {
      printf("free block of memory failed!\n");
      return(ERROR)
   }
...
```

memFindMax ()

函数原型：

```
int memFindMax (void)
```

功能描述：

该函数在系统内存分区的空闲内存列表中查找最大的空闲内存块并返回其大小。

返回值：

最大的空闲内存块的字节数。

参考：

相关信息请参考 memLib 库描述及 memPartFindMax()函数。

例：

```
int             sizenum;
...
/*  在系统内存分区的空闲块列表中查找最大的内存块  */
sizenum = memFindMax();
printf("The largest free block in the system memory partition: %d\n",sizenum);
...
```

5.2 核心内存区管理函数

5.2.1 函数库描述

1．库命名

核心内存分区管理函数库名称为 memPartLib。

2．函数

表 5-2 中列出了核心内存区管理函数。

表 5-2 核心内存区管理函数

函 数	描 述	函 数	描 述
memPartCreate ()	创建内存分区	memPartFree ()	从指定的内存分区释放一块内存
memPartAddToPool ()	向指定的内存分区中增加内存	memAddToPool ()	向系统内存分区增加内存
memPartAlignedAlloc ()	从指定的内存分区分配对齐的内存	malloc ()	从系统分区的空闲内存列表中分配一块内存
memPartAlloc ()	从指定的内存分区中分配一块内存	free ()	释放内存

3．描述

该函数库为内存分区的分配、管理提供了核心函数。该库在设计上提供了精简的表达方式，对应的完整内存管理模块包含在 memLib 函数库中（详细请参考本章第 5.1 节）。该函数库包含了两组函数：以 memPart 为开头的一组包含了创建、管理、分配、释放内存分区的函数，剩余的一组则提供了与 ANSI 标准相兼容的对系统内存分区操作的接口。

系统内存分区是由 kernelInit()函数在内核初始化时创建的。系统内存分区的 ID 存储在全局变量 memSysPartId 中（该变量在 memLib.h 中声明）。

通常情况下使用的内存分配函数 malloc()与指定内存分区的 memPartAlloc()函数使用相同的“优先适合”算法。在使用 memPartFree()和 free()函数释放内存时，相邻的内存区域会组合在一起。该函数库同样提供了可以分配内存的指定对齐方式的 memPartAlignedAlloc()函数。

4．警告

不同的体系结构的内存对齐限制也相应不同。为了提供最优的性能，malloc()会返回指向适当对齐方式缓冲区的指针。表 5-3 中列出了不同体系结构的内存块边界：

表 5-3 不同体系结构的内存块边界

体系结构	界 限	高 端	体系结构	界 限	高 端
68K	4	8	MIPS	16	12
SPARC	8	12	i960	16	16

5．头文件

核心内存分区管理函数声明在 memLib.h 及 stdlib.h 头文件中。

6．参考

相关信息请参考 memPartLib、memLib、smMemLib 库描述。

5.2.2　核心内存区管理函数详细描述

memPartCreate ()

函数原型：

```
PART_ID memPartCreate
(
char * pPool,           /* 指向内存区的指针 */
unsigned poolSize       /* 内存区大小 */
)
```

功能描述：

该函数会创建包含指定内存池的新的内存分区，它会返回一个 ID 号，使用这个 ID 来管理相应的分区（例如在分区内分配、释放内存空间）。分区可以用于管理很多独立的内存池。

注意：

新分区的描述符是从系统分区之中分配的（例如使用 malloc()来分配）。

返回值：

分区的 ID 号，如果系统分区中没有足够的空间用于新的分区则返回 NULL。

参考：

相关信息请参考 memPartLib, smMemLib 库描述。

memPartAddToPool ()

函数原型：

```
STATUS memPartAddToPool
(
PART_ID partId,         /* 初始内存区 */
char * pPool,           /* 指向内存块的指针 */
unsigned poolSize       /* 内存块字节大小 */
)
```

功能描述：

把内存添加到内存区中。该函数将内存加入到已创建的内存分区中。增加的内存不必与分区中原有的内存连续。

返回值：

OK 或者 ERROR。

错误号：

S_smObjLib_NOT_INITIALIZED、S_memLib_INVALID_NBYTES

参考：

相关信息请参考 memPartLib、smMemLib 库描述和 memPartCreate()函数。

例：

```
#define         VXWORKS_EXT_POOL_SIZE           0x200
char *      pMemUglPool   = NULL;     /* 指向 UGL 内存池的指针 */
char *    pMemExtPool   = NULL;       /* 扩展内存池 */
PART_ID     memUglPartId = NULL;      /* UGL 内存块 ID */
STATUS      st;
...
```

```
/* 分配一块内存 */
pMemUglPool = (char *) malloc (VXWORKS_UGL_POOL_SIZE);
pMemExtPool = (char *) malloc (VXWORKS_EXT_POOL_SIZE);
if ((!pMemUglPool)&(!pMemExtPool))
    return NULL;
else
   /* 创建一个内存分区 */
    memUglPartId = memPartCreate (pMemUglPool, VXWORKS_UGL_POOL_SIZE);
...
/* 将内存加入到已创建的内存分区中 */
st = memPartAddToPool(memUglPartId, pMemExtPool, VXWORKS_EXT_POOL_SIZE)
if(st == ERROR)
  {
   printf("Add memory to a memory partition failed!\n");
   return(ERROR);
  }
...
```

memPartAlignedAlloc ()

函数原型：

```
void *memPartAlignedAlloc
(
PART_ID partId,              /* 从这个内存区中分配内存 */
unsigned nBytes,             /* 分配的内存字节数 */
unsigned alignment           /* 内存边界 */
)
```

功能描述：

该函数从指定的内存分区中分配由 nByte 指定大小的缓冲区，而且它会保证缓冲区的起始地址以指定的方式对齐。指定对齐方式的 alignment 参数必须是 2 的幂。

返回值：

指向新分配的内存块的指针，如果不能分配内存空间则返回 NULL。

参考：

相关信息请参考 memPartLib 库描述。

memPartAlloc ()

函数原型：

```
void *memPartAlloc
(
PART_ID partId,        /* 从这个内存区中分配内存 */
unsigned nBytes        /* 分配字节数 */
)
```

功能描述：

该函数从指定的内存分区中分配一块内存。所分配的内存块将会等于或者大于参数 nBytes 所指定的字节数。指定的分区必须是已经创建的。

返回值：
指向内存块的指针，如果调用失败则返回 NULL。
错误号：
S_smObjLib_NOT_INITIALIZED
参考：
相关信息请参考 memPartLib, smMemLib 库描述以及 memPartCreate()函数。
memPartFree ()
函数原型：

```
STATUS memPartFree
(
PART_ID partId, /* 把释放的内存添加到这个内存分区中 */
char * pBlock    /* 指向块内存的指针 */
)
```

功能描述：
该函数将曾经分配的内存块释放给指定的内存分区。
返回值：
如果内存块无效则返回 ERROR。
错误号：
S_smObjLib_NOT_INITIALIZED
参考：
相关信息请参考 memPartLib, smMemLib 库描述以及 memPartAlloc()函数的。
例：

```
struct DataStruct
{
    int intValue;
    int intIndex;
    char *pCharText;
};
struct          DataStruct *pData;
STATUS      st1,st2;
PART_ID     memId = NULL;           /* 内存块 ID */
...
memPartFree(memId, pData->pCharText);
memPartFree(memId, (char *)pData);
if((st1&st2) == ERROR)
    printf("Free block memory failed!\n");
...
```

memAddToPool ()
函数原型：

```
void memAddToPool
(
char * pPool,           /* 指向内存块的指针 */
unsigned poolSize       /* 内存块字节数 */
```

```
)
```

功能描述：

该函数在初始化系统内存分区之后将内存增加到系统内存分区。

返回值：

无。

参考：

相关信息请参考 memPartLib 库以及 memPartAddToPool()函数的描述。

例：

```
#define ADDED_BOOTMEM_SIZE    0x00200000      /* 2MB */
char   *memTopPhys   = NULL;                  /* 内存顶部 */
...
if ((int)memTopPhys >= (0x00200000 + ADDED_BOOTMEM_SIZE))
                {
                memAddToPool ((char *)memTopPhys – ADDED_BOOTMEM_SIZE,
                                ADDED_BOOTMEM_SIZE);
                }
...
```

malloc ()

函数原型：

```
void *malloc
(
size_t nBytes      /* 分配字节数 */
)
```

功能描述：

该函数从空闲的内存列表中分配一块内存。所分配的内存块将会大于或等于参数 nBytes 所指定的字节数。

返回值：

指向新分配内存块的指针，如果出现错误则返回 NULL。

参考：

相关信息请参考 memPartLib 库描述。

free ()

函数原型：

```
void free
(
void * ptr      /* 指向内存块的指针 */
)
```

功能描述：

该函数将由 malloc()或者 calloc()所分配的内存释放到空闲内存池中。

返回值：

无。

参考：

相关信息请参考 memPartLib 库描述以及 malloc(), calloc()函数。

例：

```
#define BYTES_PER_SECTOR          0x200    /* 512 个字节      */
UCHAR * secBuf;
...
/* 分配内存 */
if ((secBuf = malloc (BYTES_PER_SECTOR)) == NULL)
    {
      printf ("Error during malloc'ing sector buffer!\n");
      return (ERROR);
    }
...
/* 释放内存 */
free (secBuf);
return (OK);
...
```

5.3　内存区显示函数

5.3.1　函数库描述

1．库命名

内存区显示函数库名称为 memShow。

2．函数

表 5-4 中列出了内存区显示函数。

3．描述

该函数库包含了内存分区信息显示函数。在使用这些函数之前，必须首先使用 memShowInit()函数。如果在 VxWorks 配置中包含了 memShow 函数库，系统会在初始化时自动调用该函数。可以使用下面的两种办法包含 memShow 函数库：

1）在 config.h 中定义 INCLUDE_SHOW_ROUTINES；

2）使用 Tornado 工程工具选定 INCLUDE_MEM_SHOW。

4．参考

相关信息请参考 memShow, memLib, memPartLib 库描述。

表 5-4　内存区显示函数

函　数	描　述
memShowInit()	初始化内存分区显示工具
memShow()	显示系统内存分区块的状态
memPartShow()	显示内存分区块的状态
memPartInfoGet()	获取分区信息

5.3.2　内存区显示函数详细描述

memShowInit ()

函数原型：

```
void memShowInit (void)
```

功能描述：

该函数将内存分区显示模块链接到 VxWorks 系统中。如果在 VxWorks 配置中包含了 memShow 函数库，系统会在初始化时自动调用该函数。可以使用下面的两种办法包含 memShow 函数库：

1）在 config.h 中定义 INCLUDE_SHOW_ROUTINES；

2）使用 Tornado 工程工具选定 INCLUDE_MEM_SHOW。

返回值：

无。

参考：

相关信息请参考 memShow 库描述。

memShow ()

函数原型：

```
void memShow
(
int type      /* 1 = 列出空闲列表中所有内存块 */
)
```

功能描述：

该函数显示系统内存分区中空闲的和已分配的内存的状态。这些状态包括字节数、内存块数、空闲的和分配的内存的平均块大小、最大空闲块尺寸、当前分配的块数、已分配的平均块尺寸。

另外，如果 type 参数为 1，该函数将会显示系统分区内空闲内存块的列表。

返回值：

无。

参考：

相关信息请参考 memShow 库描述，memPartShow()函数。以及“VxWorks Programmer's Guide”中的目标机 shell、WindSh 章节。

操作示范：

在宿主机 WindSh 工具中，memShow()函数操作示范如下：

```
-> memShow 1
FREE LIST:
num         addr          size
---       ----------     ----------
 1      0x3fee18            16
2       0x3b1434            20
3       0x4d188          2909400
SUMMARY:
 status       bytes      locks      avg block     max block
 ------      ---------   --------    ----------    ----------
current
     free     2909436      3          969812       2909400
   alloc      969060       16102        60            -
cumulative
   alloc     1143340      16365        69            -
```

memPartShow ()

函数原型：

```
STATUS memPartShow
(
PART_ID partId,          /* 内存区 ID */
int type                 /* 0 = 统计, 1 = 统计并列表 */
)
```

功能描述：

该函数可以显示指定内存分区中空闲的和已分配的内存的状态。这些状态包括字节数、内存块数、空闲的和分配的内存的平均块大小、最大空闲块尺寸、当前分配的块数、已分配的平均块尺寸。

另外，如果 type 参数为 1，该函数将会显示指定内存分区内空闲内存块的列表。

返回值：

OK 或者 ERROR.

错误号：

S_smObjLib_NOT_INITIALIZED

参考：

相关信息请参考 memShow()函数。

操作示范：

在宿主机 WindSh 工具中，memPartShow()函数操作示范如下：

```
-> memPartShow(memId,1)
FREE LIST:
num     addr        size
---     ----------  ----------
 1      0x3fee18          16
SUMMARY:
 status     bytes      locks     avg block   max block
 ------     ---------  --------  ----------  ----------
current
   alloc    969060     16102          60          -
cumulative
   alloc    1143340    16365          69          -
```

memPartInfoGet ()

函数原型：

```
STATUS memPartInfoGet
(
PART_ID partId,                         /* 内存区 ID */
MEM_PART_STATS * ppartStats             /* 内存区状态结构 */
)
```

功能描述：

该函数可以获取指定内存分区的相关信息，并将其添加到 MEM_PART_STATS 结构中。

返回值：

如果获取到有效的数据则返回 OK，否则返回 ERROR。

参考：

相关信息请参考 memShow()函数。

5.4　缓冲区处理函数

5.4.1　函数库描述

1．库命名

缓冲区处理函数库名称为 bLib。

2．函数

表 5-5 中列出了缓冲区处理函数。

3．描述

bLib 函数库包含处理缓冲区的函数。尽管缓冲区的长度是以字节计数，但依然可以按双字形式处理缓冲区。不过，这种情况只是在源缓冲区和目的缓冲区起始地址同为奇数或偶数时才发生。如果一个缓冲区为奇数，而另一个缓冲区为偶数，对缓冲区的操作必须按字节形式进行。

假设某个应用，例如外围设备按字节宽度映射内存，可能需要按字节方式操作内存。这种情况下可以使用 bcopyBytes()和 bfillBytes()函数来代替 bcopy()和 bfill。该库中提供的函数都不会检测字符串的 null 结束符。

4．头文件

缓冲区处理函数声明在 string.h 中。

5．参考

相关信息请参考 ansiString 库描述。

表 5-5 缓冲区处理函数

函 数	描 述	函 数	描 述
bcmp()	缓冲区比较	bcopyBytes()	以字节为单位复制缓冲区
binvert()	颠倒缓冲区中的字节顺序	bcopyWords()	以字为单位复制缓冲区
bswap()	交换缓冲区	bcopyLongs()	以长字为单位复制缓冲区
swab()	交换字节	bfill()	用指定字符填充缓冲区
uswab()	与缓冲区交换字节	bfillBytes()	用指定字符，以字节形式填充缓冲区
bzero()	缓冲区清零	index()	在字符串中查找字符最初出现的位置
bcopy()	复制缓冲区	rindex()	在字符串中查找字符最后出现的位置

5.4.2 缓冲区处理函数详细描述

bcmp()

函数原型：

```
int bcmp
(
char * buf1,      /* 指向第一个缓冲区的指针 */
char * buf2,      /* 指向第二个缓冲区的指针 */
int nbytes        /* 比较字节数 */
)
```

功能描述：

缓冲区比较。该函数比较缓冲区 buf1 与 buf2 中最初的 nbytes 字符。

返回值：

如果缓冲区 buf1 与 buf2 中最初的 nbytes 字符相同则返回 0，如果缓冲区 buf1 小于 buf2 则返回值小于 0，如果缓冲区 buf1 大于 buf2 则返回值大于 0。

参考：

相关信息请参考 bLib 库描述。

binvert()

函数原型：

```
void binvert
(
char * buf,          /* 指向缓冲区的指针 */
int nbytes           /* 缓冲区中字节数 */
)
```

功能描述：

颠倒缓冲区中的字节顺序。该函数一个字节一个字节地颠倒整个缓冲区。例如，缓冲区为[1，2，3，4，5]，颠倒后变为[5，4，3，2，1]。

返回值：

无。

参考：

相关信息请参考 bLib 库描述。

操作示范：

在宿主机 WindSh 工具中，binvert()函数操作示范如下：

```
-> printf("buf1: %s\n",buf1)
buf1: 12345678
value = 15 = 0xf
-> binvert(buf1, 6)
value = 4301876 = 0x41a434 = _bcmp
->   printf("buf1: %s\n",buf1)
buf1: 65432178
value = 15 = 0xf
```

bswap()

函数原型：

```
void bswap
(
char * buf1,         /* 指向第一个缓冲区的指针 */
char * buf2,         /* 指向第二个缓冲区的指针 */
int nbytes           /* 交换字节数 */
)
```

功能描述：

交换缓冲区。该函数调换两个缓冲区中最初的 nbytes 个字符。

返回值：

无。

参考：

相关信息请参考 bLib 库描述。

操作示范：

在宿主机 WindSh 工具中，bswap()函数操作示范如下：

```
-> buf1 = malloc(10);
new symbol "buf1" added to symbol table.
```

```
buf1 = 0x4eaeff4: value = 84467008 = 0x508dd40
-> buf2 = malloc(10)
new symbol "buf2" added to symbol table.
buf2 = 0x4eaefec: value = 84466984 = 0x508dd28
-> buf1 = "abcde"
buf1 = 0x4eaeff4: value = 82505700 = 0x4eaefe4
-> buf2 = "ijklm"
buf2 = 0x4eaefec: value = 82505692 = 0x4eaefdc
-> bswap(buf1,buf2,3)
value = 4235371 = 0x40a06b = _swab + 0x33
-> printf("buf1:%s      buf2: %s\n",buf1,buf2)
buf1:ijkde      buf2: abclm
value = 27 = 0x1b
```

swab()

函数原型：

```
void swab
(
char * source,              /* 指向源缓冲区的指针 */
char * destination,         /* 指向目的缓冲区的指针 */
int nbytes                  /* 调换字节数 */
)
```

功能描述：

交换字节。该函数从 source 缓冲区中获得指定的字节数，邻近奇、偶字节调换，并把结果存放在 destination 缓冲区中。两个缓冲区之间不应该重叠。

注意：

在一些 CPU 上，如果缓冲区没有对齐，函数 swab()将会引起异常。在这种情况下，使用 uswab()函数处理为对齐的缓冲区。当 nbytes 为奇数时会产生异常。

返回值：

无。

参考：

相关信息请参考 bLib 库描述，以及 uswab()函数。

操作示范：

在宿主机 WindSh 工具中，swab()函数操作示范如下：

```
-> printf("buf1:%s      buf2: %s\n",buf1,buf2)
buf1:abcde      buf2:
value = 22 = 0x16
-> swab(buf1,buf2,4)
value = 100 = 0x64 = 'd'
-> printf("buf1:%s      buf2: %s\n",buf1,buf2)
buf1:abcde      buf2: badc
```

uswab()

函数原型：

```
void uswab
```

```
(
char *      source,          /* 指向源缓冲区的指针 */
char * destination,          /* 指向目的缓冲区的指针 */
int     nbytes               /* 交换字节个数 */
)
```

功能描述：

交换字节。该函数从 source 缓冲区中获得指定个数的字节，邻近奇、偶字节调换，并把结果存放在 destination 缓冲区中。

注意：

由于速度原因，在必要的情况下才使用这个函数。使用 swab()函数处理对齐交换。参数 nbytes 应为偶数，否则将出错。

返回值：

无。

参考：

相关信息请参考 bLib 库描述，swab()函数。

bzero()

函数原型：

```
void bzero
(
char * buffer,           /* 指向缓冲区的指针 */
int     nbytes           /* 填充字节数 */
)
```

功能描述：

缓冲区清零。该函数用“0”填充缓冲区最初的 nbytes 个字符。

返回值：

无。

参考：

相关信息请参考 bLib 库描述。

bcopy()

函数原型：

```
void bcopy
(
const char * source,         /* 指向源缓冲区的指针 */
char *       destination,    /* 指向目的缓冲区的指针 */
int          nbytes          /* 复制字节个数 */
)
```

功能描述：

缓冲区复制。该函数从缓冲区 source 中复制最初的 nbytes 字符到缓冲区 destination 中。在一些体系结构上采用这种方式复制字符是一个最有效的方法。通常，如果两个缓冲区是按长字排列，这种复制速度将非常快。

返回值：

无。

参考：

相关信息请参考 bLib 库描述，以及 bcopyBytes()、bcopyWords()和 bcopyLongs()函数。

操作示范：

在宿主机 WindSh 工具中，bcopy()函数操作示范如下所示：

```
-> printf("buf1:%s        buf2: %s\n",buf1,buf2)
buf1:abcdefg        buf2:
value = 24 = 0x18
-> bcopy(buf1,buf2,5)
value = 101 = 0x65 = 'e'
-> printf("buf1:%s        buf2: %s\n",buf1,buf2)
buf1:abcdefg        buf2: abcde
value = 29 = 0x1d
```

bcopyBytes()

函数原型：

```
int bcopyBytes
(
char * source,          /* 指向源缓冲区的指针 */
char * destination,     /* 指向目的缓冲区的指针 */
int   nbytes            /* 复制字节数 */
)
```

功能描述：

以字节形式复制缓冲区。该函数从 source 缓冲区中以字节形式复制 nbytes 个字符到 destination 缓冲区中。如果缓冲区只允许字节指令访问它，外围设备必须按字节宽度映射内存。

返回值：

无。

参考：

相关信息请参考 bLib 库描述，以及 bcopy()函数。

bcopyWords ()

函数原型：

```
int bcopyWords
(
char * source,          /* 指向源缓冲区的指针 */
char * destination,     /* 指向目的缓冲区的指针 */
int   nwords            /* 复制字数 */
)
```

功能描述：

以字形式复制缓冲区。该函数从 source 缓冲区中以字形式复制 nwords 个字到 destination 缓冲区中。如果缓冲区只允许字指令访问它，外围设备必须按字宽度映射内存。且两个缓冲区必须是按字对齐。

返回值：

无。

参考：

相关信息请参考 bLib 库描述，以及 bcopy()函数。

bcopyLongs()

函数原型：

```
int bcopyLongs
(
char * source,          /* 指向源缓冲区的指针 */
char * destination,     /* 指向目的缓冲区的指针 */
int   nlongs            /* 复制长字(long word)个数 */
)
```

功能描述：

以长字形式复制缓冲区。该函数从 source 缓冲区中以长字形式复制 nlongs 个长字到 destination 缓冲区中。如果缓冲区只允许长字指令访问它，外围设备必须按长字宽度映射内存。且两个缓冲区必须是按长字对齐。

返回值：

无。

参考：

相关信息请参考 bLib 库描述，以及 bcopy()函数。

bfill()

函数原型：

```
int bfill
(
char * buf,         /* 指向缓冲区的指针 */
int nbytes          /* 填充字节个数 */
int ch              /* 填充值 */
)
```

功能描述：

用指定的字符填充缓冲区。该函数用字符 ch 填充缓冲区最初的 nbytes 字符。

返回值：

无。

参考：

相关信息请参考 bLib 库描述，以及 bfillBytes()函数。

操作示范：

在宿主机 WindSh 工具中，bfill()函数操作示范如下：

```
-> printf("buf1:%s\n",buf1)
buf1:aaaaaaaa
value = 14 = 0xe
-> bfill(buf1,6,66)
value = 6 = 0x6
-> printf("buf1:%s\n",buf1)
buf1:BBBBBBaa
value = 14 = 0xe
```

bfillBytes()

函数原型：

```
int bfillBytes
(
char * buf,        /* 指向缓冲区的指针 */
int nbytes         /* 填充字节数 */
int ch             /* 填充值 */
)
```

功能描述：

用指定的字符填充缓冲区。该函数以字节形式用字符 ch 填充缓冲区最初的 nbytes 字符。如果缓冲区只允许字节指令访问它，外围设备必须按字节宽度映射内存。

返回值：

无。

参考：

相关信息请参考 bLib 库描述，以及 bfill ()函数。

index()

函数原型：

```
char *index
(
const char * s,    /* 字符串 */
int c              /* 要查找的字符 */
)
```

功能描述：

在字符串中查找字符最初出现的位置。该函数在字符串 s 中查找 c 最初出现的位置。

返回值：

指向定位字符的指针，如果没有发现字符则返回 NULL。

参考：

相关信息请参考 bLib 库描述，以及 strchr()函数。

rindex()

函数原型：

```
char *rindex
(
const char * s,    /* 字符串 */
int c              /* 要查找的字符 */
)
```

功能描述：

在字符串中查找字符最后出现的位置。该函数在字符串 s 中查找 c 最后出现的位置。

返回值：

指向定位字符的指针，如果没有发现字符则返回 NULL。

参考：

相关信息请参考 bLib 库描述。

第6章　VxWorks的I/O系统

6.1　I/O应用接口函数

6.1.1　函数库描述

1．库命名

IO接口函数库名称为ioLib。

2．函数

表6-1中列出了I/O应用接口函数。

表6-1　I/O应用接口函数

函　数	描　述	函　数	描　述
creat()	创建文件	ioDefPathSet()	设置当前默认路径
unlink()	删除文件（POSIX）	ioDefPathGet()	获取当前默认路径
remove()	移除文件（ANSI）	chdir()	设置当前默认路径
open()	打开文件	getcwd()	获取当前默认路径（POSIX）
close()	关闭文件	getwd()	获取当前默认路径
rename()	更改文件名	ioGlobalStdSet()	为全局标准输入/输出/错误信息输出设置文件描述符
read()	从文件或设备中读取内容	ioGlobalStdGet()	获取全局标准输入/输出/错误信息输出文件描述符
write()	向文件中写入内容	ioTaskStdSet()	为任务标准输入/输出/错误信息输出设置文件描述符
ioctl()	执行I/O控制功能	ioTaskStdGet()	获取任务标准输入/输出/错误信息输出文件描述符
lseek()	设置文件读写指针	isatty()	返回设备所使用的底层驱动是否为tty类型

3．描述

该函数库包含基本I/O系统的接口。它包括：

1）由驱动提供的七个基本接口：creat()、remove()、open()、close()、read()、 write()和ioctl()；

2）面向文件系统的接口，包括rename()和lseek()；

3）设置和获取当前工作路径的函数；

4）指派任务或者全局标准文件描述符的函数。

4．文件描述符

在基本I/O一层，文件是由文件描述符来区别的。文件描述符是由open()或者creat()函数返回的一个较小的整数。其他的基本I/O接口则使用该文件描述符作为参数来指定文件。

在系统中保留了三个描述符：

0 (STD_IN)：标准输入；

1 (STD_OUT)：标准输出；

2 (STD_ERR)：标准错误输出。

VxWorks系统允许两级重定向。首先，它允许重定向全局的三个标准文件描述符。默认的情况下，新生

成的任务将会使用全局标准文件描述符作为任务标准输入输出。使用 ioGlobalStdSet() 和 ioGlobalStdGet()函数可以重定向或者获取标准描述符。

其次，系统也可以为每一个任务设置独立的重定向文件描述符。使用 ioTaskStdSet() 和 ioTaskStdGet()函数可以为任务的输入输出设置重定向。

5．头文件

I/O 接口函数声明在 ioLib.h 头文件中。

6．参考

相关信息请参考 iosLib、ansiStdio 库描述,以及“VxWorks Programmer's Guide”中的 I/O 系统部分。

6.1.2 I/O 接口函数详细描述

creat()

函数原型：

```
int creat
(
const char * name,      /*  文件名  */
int flag                /* O_RDONLY、O_WRONLY 或 O_RDWR */
)
```

功能描述：

该函数创建以参数 name 为名称的文件并使用参数 flag 所标识的方式打开。该函数首先会判断在何种设备上打开文件，然后根据设备调用相应的设备驱动。所以很多操作是依赖于设备或者驱动的。

flag 参数可以设置为 O_RDONLY(0)、O_WRONLY(1)、O_RDWR(2)以指示文件打开的方式。如果需要创建 UNIX 的 NFS 文件，可以使用 open()函数并指定相应的参数。

注意：

关于没有文件描述符状态的更多信息参见 iosInit()函数。

返回值：

文件描述符，如果未指定文件名、设备不存在、无可用文件描述符或者驱动程序则返回 ERROR。

参考：

相关信息请参考 ioLib 库描述、open()函数等。

例：

```
 char name[]= "save.txt";
 int fd;
 ...
 /*    新建一个可读写文件  */
 fd = creat(name, O_RDWR);
 ...
```

unlink()

函数原型：

```
STATUS unlink
(
char * name       /*  文件名  */
)
```

功能描述：

该函数删除指定的文件。它与 remove()执行相同的功能并提供了与 POSIX 兼容的接口。

返回值：

如果设备没有删除接口或者驱动程序返回了 OK 则返回 OK。如果没有设备存在或者驱动程序返回 ERROR 则返回 ERROR。

参考：

相关信息请参考 ioLib 库描述、remove()函数。

例：

```
char name[]= "test.txt";
STATUS   st;
...
/* 删除文件 */
st = unlink (name);
if(st == ERROR)
   {
    printf ("Delete a file failed!\n");
    return (ERROR);
   }
...
```

remove()

函数原型：

```
STATUS remove
(
const char * name       /* 文件名 */
)
```

功能描述：

该函数删除指定的文件。该函数将会调用相应设备上的驱动程序完成删除任务。

返回值：

如果设备没有删除接口或者驱动程序返回了 OK 则返回 OK。如果没有设备存在或者驱动程序返回 ERROR 则返回 ERROR。

参考：

相关信息请参考 ioLib 库描述。

例：

```
int fd,status;
char name[]= "save.txt";
...
/*    删除文件 */
if(Delete_F != 0)
{
      if(close(fd) == OK)
      status = remove(name);
...
}
...
```

open()

函数原型：

```
int open
(
const char * name,      /* 文件名 */
int flags,              /* O_RDONLY、O_WRONLY、O_RDWR 或 O_CREAT */
int mode                /* 文件模式 (UNIX chmod 类型) */
)
```

功能描述：

该函数可以用于打开文件并返回其文件描述符。该函数的参数为文件名和访问类型：

O_RDONLY (0) （或者 **READ**）：打开一个只读文件；

O_WRONLY (1)（或者 **WRITE**）：打开一个只写文件；

O_RDWR (2) （或者 **UPDATE**）：打开一个读写文件；

O_CREAT (0x0200) ：创建文件。

通常，open()函数只能打开已经存在的文件，然而，对于 NFS 网络设备来说，open()函数也可以在文件打开标志参数上“或”上 O_CREAT 来创建文件。此时，文件将以 UNIX 类型创建，如：

```
fd = open ("/usr/myFile", O_CREAT | O_RDWR, 0644);
```

只有 NFS 驱动使用 mode 参数。

注意：

关于没有文件描述符状态的更多信息参见 iosInit()函数参考。

返回值：

返回文件描述符号，如果未指定文件名、设备不存在、没有有效的文件描述符或者驱动程序返回 ERROR 时返回 ERROR。

错误码：

ELOOP

参考：

相关信息请参考 ioLib 库描述、creat()函数。

close()

函数原型：

```
STATUS close
(
int fd          /* 文件描述符 */
)
```

功能描述：

该函数关闭指定的文件并释放文件描述符。该函数会调用设备所对应的驱动程序完成工作。

返回值：

驱动程序的状态，如果文件描述符无效则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE、S_objLib_OBJ_DELETED、S_objLib_OBJ_UNAVAILABLE、S_objLib_OBJ_ID_ERROR。

参考：

相关信息请参考 ioLib 库描述。

rename()

函数原型：

```
int rename
(
const char * oldname,        /* 原文件名 */
const char * newname         /* 新的文件名 */
)
```

功能描述：

该函数用于更改文件名。

注意：

只有部分设备支持 rename()函数。可以通过察看 xxDrv 或者 xxFs 中 ioctl 功能列表中是否有 FIORENAME 来确认设备是否支持该函数。例如，dosFs、rt11Fs 支持 rename()，但是 netDrv 和 nfsDrv 则不支持。

返回值：

更名成功返回 OK，如果文件打不开或者不能改名则返回 ERROR。

参考：

相关信息请参考 ioLib 库描述。

例：

```
int          status
...
status = rename("start.txt", "end.txt");
if(status == ERROR)
    printf("Change the name of a file failed!\n");
...
```

read()

函数原型：

```
int read
(
int fd,              /* 文件描述符 */
char * buffer,       /* 指向缓冲区的指针 */
size_t maxbytes      /* 读取的最大字节数 */
)
```

功能描述：

该函数从指定的文件描述符中读取字符并将其放入 buffer 中。该函数会调用驱动程序完成这个步骤。

返回值：

读取到的字节数（1 到 maxbytes 之间，如果文件结束则返回 0）。如果文件描述符不存在、驱动不支持读过程或者驱动返回 ERROR 则返回 ERROR。如果驱动不支持读过程，errno 将会被置为 ENOTSUP。

参考：

相关信息请参考 ioLib 库描述。

write()

函数原型：

```
int write
(
```

```
int fd,              /* 文件描述符 */
char * buffer,       /* 指向写缓冲区的指针*/
size_t nbytes        /* 写字节数 */
)
```

功能描述：

该函数将 buffer 中的 nbytes 个字节写到文件描述符指定的文件中。该函数调用驱动程序运作。

返回值：

写入的字节数（如果与 nbytes 不相同则可能发生了错误）。如果文件描述符不存在、驱动不支持读过程或者驱动返回 ERROR 则返回 ERROR。如果驱动不支持读过程，errno 将会被置为 ENOTSUP。

参考：

相关信息请参考 ioLib 库描述。

例：

```
void R_Write (void)
{
    char name[]= "/myDevice/PORT1";
    char buf[100]= "Test device !";
    int fd,len;

    /*    打开设备 */
    fd = open(name, O_RDWR，0);

    /* 向设备写一串字符 */
    len = write(fd, buf, sizeof(buf))

    /*    关闭设备 */
    if(close(fd) ==ERROR)
        printf ("close device failed!\n");
}
```

ioctl()

函数原型：

```
int ioctl
(
int fd,              /* 文件描述符 */
int function,        /* 功能代码 */
int arg              /* 任意参数 */
)
```

功能描述：

该函数执行设备的 I/O 控制功能。VxWorks 的设备所使用的控制功能定义在头文件 ioLib.h 中，大多数的 I/O 请求都会传递给驱动程序来处理。由于 ioLib.h 中所包含的功能是与驱动程序相关的，所以它们分别在 tyLib、pipeDrv、nfsDrv、dosFsLib、rt11FsLib、rawFsLib 中说明。

例：

```
/* 将文件或目录的名称更改为"newname" */
```

ioctl (fd, FIORENAME, "newname");

注意 FIOGETNAME 功能是在 I/O 接口层处理的而不是传给驱动程序来处理。所以该功能不应该用于用户自定义的设备。

返回值：

驱动程序的返回值，如果文件描述符不存在则返回 ERROR。

参考：

相关信息请参考 ioLib、tyLib、pipeDrv、nfsDrv、dosFsLib、rt11FsLib、rawFsLib 库描述。

lseek()

函数原型：

```
int lseek
(
int fd,             /* 文件描述符 */
long offset,        /* 字节偏移 */
int whence          /* 关联文件位置 */
)
```

功能描述：

该函数设置指定的文件读写位置为 offset。参数 whence 的取值如下：

SEEK_SET(0)：将文件指针设置到 offset；

SEEK_CUR(1)：将文件指针设置到当前位置加 offset；

SEEK_END(2)：将文件指针设置到文件末尾加 offset。

该过程会调用 ioctl()函数的 FIOWHERE、FIONREAD、FIOSEEK 功能。

返回值：

从文件开始算起的新偏移值，其他情况返回 ERROR。

参考：

相关信息请参考 ioLib 库描述。

ioDefPathSet()

函数原型：

```
STATUS ioDefPathSet
(
char * name/* 设备新的默认路径名 */
)
```

功能描述：

该函数设置默认的 I/O 路径。所有 I/O 系统中相关的路径都会以该路径为前提。这里的路径名必须是一个绝对路径，也就是说 name 参数要以一个存在的设备开头。

返回值：

成功设置则返回 OK，如果指定的设备不存在则返回 ERROR。

参考：

相关信息请参考 ioLib 库描述、ioDefPathGet()、chdir()、getcwd()函数。

例：

```
int SetPath(char * name)
{
STATUS   st;
```

```
char * pathname;
...
/* 设置路径 */
st = ioDefPathSet(name);
if(st == ERROR)
    {
 printf("Set new path failed!\n");
     return(ERROR);
 }
}
...
/* 获得新设置的路径名 */
ioDefPathGet(pathname);
printf("The current default path: %s\n",pathname);
}
```

ioDefPathGet()

函数原型：

```
void ioDefPathGet
(
char * pathname /* 默认路径名 */
)
```

功能描述：

该函数将当前默认路径的名称复制到 pathname 参数中。参数 pathname 的长度至少要等于 MAX_FILENAME_LENGTH。

返回值：

无。

错误码：

S_objLib_OBJ_ID_ERROR、S_intLib_NOT_ISR_CALLABLE。

参考：

相关信息请参考 ioLib 库描述、ioDefPathSet()、chdir()、getcwd()函数。

例子请参见上例。

chdir()

函数原型：

```
STATUS chdir
(
char * pathname /* 新默认路径名 */
)
```

功能描述：

该函数设置默认的 I/O 路径。所有 I/O 系统中相关的路径都会以该路径为前提。这里的路径名必须是一个绝对路径，也就是说 name 参数要以一个存在的设备开头。

返回值：

成功设置当前默认路径返回 OK，如果指定的设备不存在则返回 ERROR。

错误码：

S_intLib_NOT_ISR_CALLABLE

参考：

相关信息请参考 ioLib 库描述、ioDefPathSet()、ioDefPathGet()、getcwd()函数。

getcwd()

函数原型：

```
char *getcwd
(
char * buffer,      /* 存放路径名的缓冲区 */
int size            /* 缓冲区中的字节数 */
)
```

功能描述：

该函数将当前默认路径复制到 buffer 中去，它提供了与 ioDefPathGet()相同的功能，同时提供了与 POSIX 兼容的接口。

返回值：

指向所提供缓冲区的指针，当 size 参数小于当前路径名长度则返回 NULL。

参考：

相关信息请参考 ioLib 库描述、ioDefPathSet()、ioDefPathGet()、chdir()函数。

getwd()

函数原型：

```
char *getwd
(
char * pathname /* 存放路径名的缓冲区 */
)
```

功能描述：

该函数将当前默认路径的名称复制到 pathname 参数中。它提供了与 ioDefPathGet()、getcwd()相同的功能，同时提供了与某些其他 POSIX 兼容的接口。参数 pathname 的值要不小于 MAX_FILENAME_LENGTH 的值。

返回值：

指向结果路径名的指针。

参考：

相关信息请参考 ioLib 库描述。

ioGlobalStdSet()

函数原型：

```
void ioGlobalStdSet
(
int stdFd,    /* 标准输入 (0)、输出(1) 或 错误输出(2) */
int newFd     /* 新文件描述符 */
)
```

功能描述：

该函数改变全局标准输入/输出/错误输出的文件描述符。参数 newfd 必须是一个已打开文件（或者设备）的文件描述符。如果没有特别的设置，所有对标准设备输入输出的操作都会指向所设定的文件。如果参数 stdFd

的取值不是 0、1、2，则该函数不作任何响应。

返回值：

无。

参考：

相关信息请参考 ioLib 库描述以及 ioGlobalStdGet()和 ioTaskStdSet()函数。

例：

```
int             consoleFd;        /* 文件描述符 */
...
/* 打开设备"/tyCo/0"*/
consoleFd = open ("/tyCo/0", O_RDWR, 0);
...
/* 设置文件描述符标准输入、输出和错误输出 */
ioGlobalStdSet (STD_IN,  consoleFd);
ioGlobalStdSet (STD_OUT, consoleFd);
ioGlobalStdSet (STD_ERR, consoleFd);
...
```

ioGlobalStdGet()

函数原型：

```
int ioGlobalStdGet
(
int stdFd /* 标准输入 (0)、 输出(1) 或 错误输出(2) */
)
```

功能描述：

该函数获取当前标准输入、输出或者错误输出设备的文件描述符。

返回值：

指定的全局文件描述符。如果参数 stdFd 的取值不是 0、1、2 则返回 ERROR。

参考：

相关信息请参考 ioLib 库以及 ioGlobalStdSet()和 ioTaskStdGet()函数。

ioTaskStdSet()

函数原型：

```
void ioTaskStdSet
(
int taskId,   /* 任务 ID (0 = 自身) */
int stdFd,    /* 标准输入 (0)、 输出(1) 或 错误输出(2) */
int newFd     /* 新文件描述符 */
)
```

功能描述：

该函数将指定任务的标准文件描述符指派为新的文件描述符，新的文件描述符应指向一个已打开的文件。对应的任务将会使用新指定的文件作为标准输入输出设备。如果参数 stdFd 不是 0、1、2 则该函数无响应。

注意：

在中断一级调用该函数时，该函数不作任何响应。

返回值：

无。

参考：

相关信息请参考 ioLib 库描述以及 ioGlobalStdGet()和 ioTaskStdGet()函数。

例：

```
LOCAL int vxTaskCreate
    (
    char *    name,          /* 任务名 */
    int       priority,      /* 任务优先级 */
    int       options,       /* 任务选项字 */
    caddr_t   stackBase,     /* 堆栈基地址 */
    int       stackSize,     /* 堆栈大小 */
    caddr_t   entryPt,       /* 任务入口点 */
    int       arg[10],       /* 传递给任务的参数 */
    int       fdIn,          /* 标准输入 */
    int       fdOut,         /* 标准输出 */
    int       fdErr          /* 错误输出 */
    )
    {
    int tid;
...
    if (stackSize == 0)
 stackSize = WDB_SPAWN_STACK_SIZE;
...
    tid = taskCreat (name, priority, options, stackSize, (int (*)())entryPt,
      arg[0], arg[1], arg[2], arg[3], arg[4], arg[5], arg[6],
      arg[7], arg[8], arg[9]);
      ...
    if (tid == NULL)         /* 任务创建失败 */
 return (ERROR);
...
#ifdef INCLUDE_IO_SYSTEM
    if (fdIn != 0)
    ioTaskStdSet (tid, 0, fdIn);
    if (fdOut != 0)
    ioTaskStdSet (tid, 1, fdOut);
    if (fdErr != 0)
    ioTaskStdSet (tid, 2, fdErr);
#endif/* INCLUDE_IO_SYSTEM */
    return (tid);
    }
```

ioTaskStdGet()

函数原型：

```
int ioTaskStdGet
```

```
(
int taskId,        /* 任务 ID (0 = 自身) */
int stdFd          /* 标准输入 (0), 输出 (1), 或错误输出 (2) */
)
```

功能描述：

该函数返回指定任务的标准输入、输出或者错误输出文件所对应的文件描述符。

返回值：

对应的文件描述符。如果参数 stdFd 的取值不是 0、1、2 或者在中断级调用该函数，则返回 ERROR。

参考：

相关信息请参考 ioLib 库描述以及 ioGlobalStdGet()和 ioTaskStdSet()函数。

isatty()

函数原型：

```
BOOL isatty
(
int fd  /* 文件描述符 */
)
```

功能描述：

该函数执行指定文件描述符的文件所对应驱动的 ioctl()函数的 FIOISATTY 功能。

返回值：

TRUE，如果设备的驱动不是 tty 设备则返回 FALSE

参考：

相关信息请参考 ioLib 库的描述。

6.2 I/O 系统函数

6.2.1 函数库描述

1．库命名

IO 系统函数库名称为 iosLib。

2．函数

表 6-2 中列出了 I/O 系统函数。

表 6-2 I/O 系统函数

函 数	描 述	函 数	描 述
iosInit()	初始化 I/O 系统	iosDevDelete()	从 I/O 系统中删除设备
iosDrvInstall()	安装 I/O 驱动程序	iosDevFind()	在设备列表中寻找 I/O 设备
iosDrvRemove()	删除 I/O 驱动程序	iosFdValue()	确认已打开的文件描述符有效并返回驱动程序对应的值
iosDevAdd()	为 I/O 系统添加设备		

3．描述

该函数库是驱动级 I/O 系统的接口。其主要功能是使用适当的参数将用户的 I/O 请求传递给适当的驱动程序。为了实现这个目的，iosLib 维护着一个有效驱动程序表。

系统应该在调用该函数库中的其他函数之前先使用 iosInit()来初始化 I/O 系统。之后就可以使用 iosDrvInstall()来安装每一组驱动程序并通过 iosDevAdd()添加对应的设备。

4．头文件

IO 系统函数声明在 iosLib.h 头文件中。

5．参考

相关信息请参考 iosLib、intLib、ioLib 库描述。

6.2.2　I/O 系统函数详细描述

iosInit()

函数原型：

```
STATUS iosInit
(
int max_drivers,        /* 最大设备驱动程序个数 */
int max_files,          /* 最大文件打开个数 */
char * nullDevName      /* 空设备名 */
)
```

功能描述：

该函数初始化 I/O 系统。必须在使用 I/O 系统的其他函数之前调用它。

返回值：

成功初始化 I/O 系统则返回 OK，如果内存不足则返回 ERROR。

参考：

相关信息请参考 iosLib 库描述。

iosDrvInstall()

函数原型：

```
int iosDrvInstall
(
FUNCPTR pCreate,    /* 指向驱动程序创建函数的指针 */
FUNCPTR pDelete,    /* 指向驱动程序删除函数的指针*/
FUNCPTR pOpen,      /* 指向驱动程序打开函数的指针 */
FUNCPTR pClose,     /* 指向驱动程序关闭函数的指针 */
FUNCPTR pRead,      /* 指向驱动程序读函数的指针 */
FUNCPTR pWrite,     /* 指向驱动程序写函数的指针 */
FUNCPTR pIoctl      /* 指向驱动程序控制函数的指针 */
)
```

功能描述：

安装 I/O 驱动程序。对于每一组 I/O 驱动程序，该函数应该只能调用一次。该函数的作用是将不同的 I/O 驱动程序分派惟一的驱动号并将其对应的函数入口地址增加到驱动程序表中。

返回值：

对应驱动程序的驱动号，如果驱动表中的空间不足则返回 ERROR。

参考：

相关信息请参考 iosLib 库描述。

例：

```
int   XXDrvNum ;      /* 驱动程序号 */
...
/* 驱动程序安装函数 */
STATUS XXDrvInstall()
{
 if(XXDrvNum > 0) return(OK);
 XXDrvNum = iosDrvInstall ((FUNCPTR) NULL, (FUNCPTR) NULL, XXOpen,
                    XXClose, XXRead, XXWrite, XXIoctl);
 return (XXDrvNum == ERROR ? ERROR : OK);
}
```

iosDrvRemove()

函数原型：

```
STATUS iosDrvRemove
(
int drvnum,              /* 驱动程序号，iosDrvInstall()返回的值 */
BOOL forceClose          /* 如果为 TRUE，强制关闭打开的文件 */
)
```

功能描述：

该函数将由 iosDrvInstall()添加的驱动程序从驱动程序表中移除。

返回值：

成功卸载驱动则返回 OK，如果有打开的文件在使用驱动则返回 ERROR。

参考：

相关信息请参考 iosLib 库描述、iosDrvInstall()函数。

例：

```
int   XXDrvNum ;      /* 驱动程序号 */
...
/* 驱动程序卸载函数 */
STATUS XXDrvUninstall()
{
 if (XXDrvNum != ERROR)
 {
      if(iosDrvRemove(XXDrvNum, TRUE) == ERROR)
      {
           return(ERROR);
      }
      XXDrvNum = ERROR;
 }
 return (OK);
}
```

iosDevAdd()

函数原型：

```
STATUS iosDevAdd
(
```

```
    DEV_HDR * pDevHdr,      /* 指向设备结构的指针 */
    char * name,            /* 设备名 */
    int drvnum              /* 驱动程序号 */
    )
```

功能描述：

该函数将设备添加到 I/O 系统的设备列表中，使用 open()、creat()函数可以打开设备。

参数 pDevHdr 是指向设备头（DEV_HDR）的指针。设备头是设备列表中的一个节点。通常是设备所对应的数据结构的第一个成员。

返回值：

设备添加成功则返回 OK，如果已经有同名的设备存在则返回 ERROR。

参考：

相关信息请参考 iosLib 库描述。

例：

```
typedef struct          XXDEV
{
      DEV_HDR       devHdr;
 BOOL                 isCreate;
      BOOL          isOpen;
      UINT32        ChNo;
      UINT32        Status;
      UINT32        ErrCode;
  }   XXDEV;
int   XXDrvNum ;          /* 驱动程序号 */
...
/* 设备创建函数 */
STATUS XXDevCreate(char *name,int cardno,int chno)
{
 XXDEV pDev;
 if(XXDrvNum == ERROR)
 {
      errnoSet(syncSio_errLib_driver_install_error);
      return (ERROR);
 }
      if(pDev.isCreate == TRUE)
      {
      errnoSet(syncSio_errLib_device_created);
      return (ERROR);
 }

     /* 通道号 */
 pDev.ChNo = chno;
 pDev.isOpen = FALSE;
 ...
```

```
/* 添加设备 */
if(iosDevAdd((DEV_HDR *)pDev.devHdr,name,XXDrvNum) == ERROR)
{
    errnoSet(syncSio_errLib_device_create_error);
    return(ERROR);
}
pDev.isCreate = TRUE;
pDev.ErrCode = 0;
return(OK);
}
```

iosDevDelete()

函数原型：

```
void iosDevDelete
(
DEV_HDR * pDevHdr /* 指向设备结构的指针*/
)
```

功能描述：

该函数从 I/O 系统设备列表中删除设备，使后续的 open()和 creat()函数无法使用该设备。由于该函数不影响驱动程序，任何已经打开的文件或者挂起的操作都不会受到影响。

如果从未增加过相应的设备将会产生不可预期的结果。

返回值：

无。

参考：

相关信息请参考 iosLib 库描述。

iosDevFind()

函数原型：

```
DEV_HDR *iosDevFind
(
char * name,              /* 设备名 */
char * *pNameTail         /* 指向设备名的指针 */
)
```

功能描述：

该函数在系统设备列表中查找与指定设备名匹配的第一个设备。如果找到了设备，iosDevFind()函数会将参数 pNameTail 指向参数 name 的第一个字符，之后跟随着与设备名相同的部分。如果设备未找到，函数返回指向默认设备的指针（当前目录所指向的设备），并且参数 pNameTail 指向 name 参数。如果没有默认设备存在，该函数返回 NULL。

返回值：

指向设备头的指针，如果未找到设备则返回 NULL。

参考：

相关信息请参考 iosLib 库描述。

iosFdValue()

函数原型：

```
int iosFdValue
(
int fd  /* 文件描述符 */
)
```

功能描述：

该函数检查参数所传递的文件描述符是否有效并返回与驱动程序相关的值。

返回值：

驱动程序对应的值，如果文件描述符无效则返回 ERROR。

参考：

相关信息请参考 iosLib 库描述。

例：

```
int             inFd;
...
if (iosFdValue (inFd) == ERROR)
        {
        errno = S_filterDrv_BAD_FILESPEC;
        return (ERROR);
        }
...
```

6.3　I/O 系统显示函数

6.3.1　函数库描述

1．库命名

IO 系统显示函数库命名为 iosShow。

2．函数

表 6-3 中列出了 I/O 系统显示函数。

表 6-3　I/O 系统显示函数

函　数	描　述	函　数	描　述
iosShowInit()	初始化 I/O 系统显示模块	iosDevShow()	显示系统中设备列表
iosDrvShow()	显示系统中驱动列表	iosFdShow()	显示系统中文件描述符列表

3．描述

该函数库包含了 I/O 系统信息显示函数。函数 iosShowInit()会将 I/O 系统信息显示工具连接到系统中。如果 configAll.h 中定义了 INCLUDE_SHOW_ROUTINES，则系统会自动调用该函数。

4．参考

相关信息请参考 iosShow 库描述。

6.3.2　任务显示函数详细描述

iosShowInit()

函数原型：

void iosShowInit (void)

功能描述：

该函数会将 I/O 系统信息显示工具连接到系统中。如果 configAll.h 中定义了 INCLUDE_SHOW_ROUTINES 则系统会自动调用该函数。

返回值：

无。

参考：

相关信息请参考 iosShow 库描述。

iosDrvShow()

函数原型：

void iosDrvShow (void)

功能描述：

该函数显示驱动程序列表中的所有驱动程序。

返回值：

无。

参考：

相关信息请参考 iosShow 库描述。

操作示范：

在宿主机 WindSh 工具中，iosDrvShow()函数操作示范如下：

```
-> iosDrvShow
drv      create   delete   open     close    read     write    ioctl
1        4126dc   0        4126dc   412704   43f2c0   43f1f0   412730
2        0        0        42ad24   0        42ad54   42ad94   42ae80
3        42bf44   42bfec   42c258   42bef0   42c338   42c364   42c0e4
4        41bf88   0        41bf88   41c048   43f2c0   43f1f0   41c0bc
5        41ddb0   41df70   41ddcc   41e064   41e114   41e1b4   41e254
```

iosDevShow()

函数原型：

void iosDevShow (void)

功能描述：

该函数显示设备列表中的所有设备。

返回值：

无。

参考：

相关信息请参考 iosShow 库描述。

操作示范：

在宿主机 WindSh 工具中，iosDevShow()函数操作示范如下：

```
-> iosDevShow
    drv  name
    0    /null
    1    /tyCo/0
```

```
3    host:
4    /vio
5    /tgtsvr
```

iosFdShow()

函数原型：

void iosFdShow (void)

功能描述：

该函数显示系统中所有的文件描述符。

返回值：

无。

参考：

相关信息请参考 iosShow 库描述。

操作示范：

在宿主机 WindSh 工具中，iosFdShow()函数操作示范如下：

```
-> iosFdShow
  fd        name          drv
  3         /tyCo/0       1
  4         /vio/1        4
  5         /vio/2        4
```

6.4　格式化 I/O 函数

6.4.1　函数库描述

1．库命名

格式化 I/O 函数库命名为 fioLib。

2．函数

表 6-4 中列出了格式化 I/O 函数。

表 6-4　格式化 I/O 函数

函　数	描　述	函　数	描　述
fioLibInit()	初始化格式化的 I/O 库	vfdprintf()	向指定文件描述符中写入带有变量声明的格式化字符串
printf()	向标准输出流中写入格式化的字符串		
printErr()	向标准错误输出流中写入格式化的字符串	vsprintf()	向缓冲区中写入带有变量声明的格式化字符串
fdprintf()	向指定文件描述符中写入格式化的字符串	fioFormatV()	转换格式化的字符串
sprintf()	向缓冲区中写入格式化的字符串	fioRead()	读取到指定的缓冲区
vprintf()	向标准输出设备中写入带有变量声明的格式化字符串	fioRdString()	从文件中读取字串
		sscanf()	从 ASCII 字符串中读取并转换字符

3．描述

该函数库提供了基本的格式化输入输出功能。其中包含了兼容 ANSI 标准的部分函数。同时也包含了几个模块函数。

如果希望该函数库支持浮点格式（例如支持 e、E、f、g、G），则需要首先调用 floatInit()函数。如果在

usrConfig.c 中已经定义了宏 INCLUDE_FLOATING_POINT，则根任务会自动调用 floatInit()。

该库中包含的函数不使用标准 I/O 库中提供的缓冲型 I/O 函数，所以即使系统中不包含标准 I/O 包也可以使用这些函数。其中 printf()在大多数的 UNIX 系统中都是缓冲型 I/O 的一部分，但在 VxWorks 中则作为一个非缓冲型的 I/O 实现，原因在于 printf()函数经常用到。这样的设计可以在大多数情况下减小系统的体积。

4．头文件

格式化 I/O 函数声明在 fioLib.h、stdio.h 头文件中。

5．参考

相关信息请参考 fioLib、floatLib 库描述。

6.4.2 格式化的 I/O 函数详细描述

fioLibInit()

函数原型：

void fioLibInit (void)

功能描述：

该函数初始化格式化的输入输出函数库支持。在使用诸如 printf()和 scanf()函数之前应该在 usrRoot()函数中调用一次该函数。

返回值：

无。

参考：

相关信息请参考 fioLib 库描述。

printf()

函数原型：

```
int printf
(
const char * fmt /* 格式字符串 */
)
```

功能描述：

该函数向标准输出设备写入格式化的信息。参数 fmt 包含了原始的字符,也就是直接输出的字符，以及转换信息，这些信息使跟随着 fmt 的声明输出为格式化的字串。

参数的数量是任意的，但是必须对应于 fmt 中的转换。如果参数不足，执行的后果则不可预期。如果参数过多，多出的参数将会被忽略。该函数在遇到字符串结束时返回。

注意：

该函数包含在 fioLib.h 头文件中。

返回值：

输出字符的数量，或者出错的时候返回一个负数。

参考：

相关信息请参考 fioLib 库描述及 fprintf()函数。

printErr()

函数原型：

```
int printErr
(
```

```
const char * fmt   /* 格式字符串 */
)
```

功能描述：

该函数向标准错误输出设备输出格式化的信息。其功能及语法与 printf()相似。

返回值：

输出的字符数，如果在输出过程中出现错误则返回 ERROR。

参考：

相关信息请参考 fioLib 库描述及 printf()函数。

fdprintf()

函数原型：

```
int fdprintf
(
int fd,                 /* 文件描述符 */
const char * fmt        /* 格式字符串 */
)
```

功能描述：

该函数向指定的文件描述符中写入格式化的信息。

返回值：

输出的字符个数，如果输出过程中出现错误则返回 ERROR。

参考：

相关信息请参考 fioLib 库描述及 printf()函数。

例：

```
int         tyfd;
int         num
...
tyfd = open("/tyCo/0",O_RDWR,0)
...
num = fdprintf(tyfd, "Test Info!");
if(num == ERROR)
   {
   printf(" Error in fdprintf function!\n");
   return (ERROR);
  }
...
```

sprintf()

函数原型：

```
int sprintf
(
char * buffer,          /* 写缓冲区 */
const char * fmt        /* 格式字符串 */
)
```

功能描述：

该函数将格式化的信息复制到指定的缓冲区中。其功能及语法与 printf()相似。

返回值：

复制到缓冲区的字符个数，但不包含最后的结束符 NULL。

参考：

相关信息请参考 fioLib 库描述及 printf()函数。

vprintf()

函数原型：

```
int vprintf
(
const char * fmt,       /* 格式化的字符串 */
va_list vaList          /* 格式参数 */
)
```

功能描述：

该函数向标准输出设备输出带有变量声明的格式化信息。除了带有变量声明，其功能与 printf()相似。

返回值：

输出的字符数，如果输出过程中出错则返回 ERROR。

参考：

相关信息请参考 fioLib 库描述及 printf()函数。

vfdprintf()

函数原型：

```
int vfdprintf
(
int fd,                 /* 文件描述符 */
const char * fmt,       /* 格式化的字符串 */
va_list vaList          /* 参数选项 */
)
```

功能描述：

该函数向指定的文件描述符输出带有变量声明的格式化信息。除了带有变量声明外，其功能与 fdprintf()相似。

返回值：

输出的字符数，如果输出过程中出错则返回 ERROR。

参考：

相关信息请参考 fioLib 库描述及 fdprintf()函数。

vsprintf()

函数原型：

```
int vsprintf
(
char * buffer,          /* 写缓冲区 */
const char * fmt,       /* 格式化的字符串 */
va_list vaList          /* 参数选项 */
)
```

功能描述：

该函数将带有变量声明的格式化信息复制到指定的缓冲区。除了带有变量声明外，该函数与 sprintf()相似。

返回值：

复制到缓冲区的字符个数，但不包含结束符 NULL。

参考：

相关信息请参考 fioLib 库描述及 sprintf()函数。

fioFormatV()

函数原型：

```
int fioFormatV
(
const char * fmt,          /* 格式化的字符串 */
va_list vaList,            /* 指向参数的列表 */
FUNCPTR outRoutine,        /* 处理程序 */
int outarg                 /* 处理程序的参数 */
)
```

功能描述：

该函数由 printf()系列函数调用，处理实际的转换工作。其第一个参数就是格式子串，也就是 printf()的第一个参数，第二个参数是一组变量的列表。

当处理完成后，结果将由第三个参数所传递的函数输出。该输出函数将结果输出到一个设备或者一段缓存中。除了输出缓冲区和长度，第四个参数用于第三个参数所指定的函数，该参数可以是一个文件描述符、一段缓冲区地址或者其他的值。输出函数应声明如下：

```
STATUS outRoutine
(
char *buffer,          /* 传递给程序的缓冲区 */
int nchars,            /* 缓冲区长度 */
int outarg             /* 传递给处理程序的任意参数 */
)
```

如果输出函数成功执行应返回 OK，反之则返回 ERROR。

返回值：

输出的字符数，如果输出过程中出错则返回 ERROR。

参考：

相关信息请参考 fioLib 库描述。

fioRead()

函数原型：

```
int fioRead
(
int fd,            /* 文件描述符 */
char * buffer,     /* 输入缓冲区 */
int maxbytes       /* 读取的最大字节数 */
)
```

功能描述：

该函数重复调用 read()函数直到向缓冲区中读入指定的字节。如果中途遇到 EOF，所读入的字节小于 maxbytes。

返回值：

读取的字节数，如果在读操作中遇到错误则返回 ERROR。

参考：

相关信息请参考 fioLib 库描述及 read()函数。

fioRdString()

函数原型：

```
int fioRdString
(
int fd,             /* 设备文件描述符 */
char string[],      /* 输入缓冲区 */
int maxbytes        /* 读取的最大字节数 */
)
```

功能描述：

该函数将一行输入放置到字串中。函数会读取指定的文件直到读到指定的字节数或者遇到 EOF 或 EOS 或者一个新行的字符来到。新行的字符或者 EOF 可以替代 EOS，除非已经读到指定的字节数。

返回值：

读到的字串的长度，包括 EOS 字符（或者 EOF 字符）。

参考：

相关信息请参考 fioLib 库描述。

例：

```
int             tyfd;
charrbuf[100];
int             num;
...
tyfd = open("/tyCo/0",O_RDWR,0)
...
num = fioRdString (tyfd, rbuf, sizefo(rbuf));
```

sscanf()

函数原型：

```
int sscanf
(
const char * str,   /* 字符串 */
const char * fmt    /* 格式化的字符串 */
)
```

功能描述：

该函数从字串中读取字符并且转换为 fmt 所指定的格式。

如果参数不足，执行的后果不可预期。如果参数过多，多余的参数将会被忽略。该函数在遇到字符串结束时返回。

返回值：

返回输入的项目条数，可能比提供的要少，甚至可能是零。或者在遇到输入错误的时候返回 EOF。

参考：

相关信息请参考 fioLib 库描述，fscanf(), scanf()函数。

例：

```
char input[80];
int     num;
...
gets (input);
num = sscanf (input, "%d");
...
```

6.5　select 函数

6.5.1　函数库描述

1．库命名

select 函数库名称为 selectLib。

2．函数

表 6-5 中列出了 select 函数。

表 6-5　select 函数

函　数	描　述	函　数	描　述
selectInit()	初始化 select 函数库	selNodeDelete()	从 select()唤醒列表中寻找并删除节点
select()	阻塞一组文件描述符	selWakeupListInit()	初始化 select()唤醒列表
selWakeup()	唤醒由 select()阻塞的任务	selWakeupListLen()	获得 select()唤醒列表中节点的个数
selWakeupAll()	唤醒所有在 select()唤醒列表中的任务	selWakeupType()	获得 select()唤醒节点的类型
selNodeAdd()	向 select()唤醒列表中增加节点		

3．描述

该函数库提供了与 BSD4.3 相兼容的 select 模块，用于等待一组文件描述符的活动。selectLib 为驱动提供了使任务等待设备活动的机制。这种机制允许驱动的中断服务程序唤醒相对应的任务，而不再使用轮询方式。其支持的最大文件描述符个数为 256。

应用程序可以使用 select()函数对管道、串行设备和 socket 进行操作，支持对文件描述符的读写操作，但是不支持异常处理操作。

通常，应用程序的开发者需要关心 select()函数的调用。然而，如果驱动程序需要支持 select()机制，则其开发者还需要掌握与 select()相关的其他函数。

4．头文件

select 函数声明在 selectLib.h 头文件中。

5．参考

相关信息请参考 selectLib 库描述。

6.5.2　select 函数详细描述

selectInit()

函数原型：

void selectInit (void)

功能描述：

该函数初始化兼容 UNIX BSD4.3 的 select 函数库。通常系统会在根任务中调用它。该函数仅可调用一次，它会在系统中安装一个任务删除钩子函数用于在任务删除时清除在 select()中阻塞的任务。

返回值：

无。

参考：

相关信息请参考 selectLib 库描述。

select()

函数原型：

```
int select
(
int width,                  /* 检查位号 */
fd_set * pReadFds,          /* 读文件描述符组 */
fd_set * pWriteFds,         /* 写文件描述符组 */
fd_set * pExceptFds,        /* 异常文件描述符组(不支持) */
struct timeval * pTimeOut   /* 等待最长时间, NULL = 永远等 */
)
```

功能描述：

该函数可以使任务挂起等待指定的文件描述符变为就绪状态。三个输入参数 pReadFds、pWriteFds 和 pExceptFds：是指向文件描述符组的指针，该文件描述符组的每个值为“1”的位均对应着一个文件描述符。如果 pReadFds 描述符组相应的位为 1，则任务会等待对应文件可以读取到有效数据为止。如果 pWriteFds 描述符组相应的位为 1，则任务会等待对应文件可以写入为止（目前还不支持 pExceptFds，但是为了兼容 UNIX 提供了相应的接口）。

对于设置文件描述符组，可以使用下面的宏定义：

FD_SET(fd, &fdset)

FD_CLR(fd, &fdset)

FD_ZERO(&fdset)

如果 pReadFds 或者 pWriteFds 为空，select()将会忽略它们。参数 width 定义了指定文件描述符的个数，应比文件描述符组中设定的最大文件描述符大 1，或者直接等于 FD_SETSIZE。当 select()函数返回时，它会将文件描述符组清空并设置准备好的文件描述符位。可以使用宏定义 FD_ISSET 来检查文件描述符位的有效性。

如果 pTimeOut 为 NULL，select()函数也会阻塞。如果 pTimeOut 不为空而是指出了时间间隔为零，则 select()函数会立刻轮询文件描述符组中指定的文件描述符并返回。如果有效的时间大于零，select()函数会在指定的时间后返回。

应用程序可以对管道设备、串行设备、网络插口等使用 select()函数。select()函数支持对文件描述符的读写操作，但是不支持异常处理操作。

返回值：

有效文件描述符的个数，如果时间到但没有有效的文件描述符则返回 0，如果当 select()函数调用驱动中的 ioctl()函数出错时则返回 ERROR。

参考：

相关信息请参考 selectLib 库描述。

例：

```
int width;
struct fd_set readFds;
int inFd1;                          /* 管道设备 1 */
int inFd2;                          /* 管道设备 2 */
...
FD_SET (inFd1, &readFds);           /* 打开 inFd1 */
FD_SET (inFd2, &readFds);           /* 打开 inFd2 */
width = (inFd1 > inFd2) ? inFd1 : inFd2;
width++;
...
if ((numFds = select (width, &readFds, NULL, NULL, NULL)) == ERROR)
                {
                perror (" ERROR in select");
                return (ERROR);
                }
    else
        printf (Number of file descriptors ready for reading = %d\n", numFds);
...
```

selWakeup()

函数原型：

```
void selWakeup
(
SEL_WAKEUP_NODE * pWakeupNode       /* 要唤醒的节点 */
)
```

功能描述：

该函数将唤醒由 select()函数挂起的任务。一旦驱动程序的 FIOSELECT 功能在驱动的唤醒列表中安装了一个节点并且设备已就绪，该函数将会解除 select()函数的阻塞。

返回值：

无。

参考：

相关信息请参考 selectLib 库描述。

例：

```
typedef struct      PCI9080DEV
{
 DEV_HDR devHdr;                    /* 设备头结构 */
 UINT32    ChNo;                    /* 通道号 */
 UINT32    RegBase;                 /* 本地寄存器基地址 */
 UINT32    ErrCode;                 /* 错误代码 */
 BOOLisCreate;                      /* 设备创建标志 */
 BOOLisOpen;                        /* 设备打开标志 */
 ...
     SEL_WAKEUP_LIST selWakeupList;   /* select 列表结构 */
```

```
} XXDEV;
...
/* XX 设备驱动程序控制函数 */
int XXIoctl(XXDEV *pDev, int cmd,int Arg)
{
if(pDev == ((XXDEV *)NULL))
{
        errnoSet(Sio_errLib_parameter_error);
        return(ERROR);
}
if(!pDev->isCreate)
{
        errnoSet(Sio_errLib_device_create_error);
        return(ERROR);
}
if(!pDev->isOpen)
{
        errnoSet(Sio_errLib_open_failed);
        return(ERROR);
 }
 switch(cmd)
 {
 case FIOSELECT:
                /* 增加节点到 wakeup 列表上 */
                selNodeAdd (&pDev->selWakeupList, (SEL_WAKEUP_NODE *) Arg);
                if (selWakeupType ((SEL_WAKEUP_NODE *) Arg) == SELREAD )
                {
                    /* 数据可用, 唤醒任务 */
                    selWakeup ((SEL_WAKEUP_NODE *) Arg);
        }
              if (selWakeupType ((SEL_WAKEUP_NODE *) Arg) == SELWRITE)
                {
                    /* 设备就绪等待写, 唤醒任务 */
                    selWakeup ((SEL_WAKEUP_NODE *) Arg);
                }
                break;
  case FIOUNSELECT:
                /* 从 wakeup 列表上删除节点 */
                selNodeDelete (&pDev->selWakeupList, (SEL_WAKEUP_NODE *) Arg);
                break;
        case XX_FUNC_GET_LAST_ERROR:        /* 得到错误代码 */
                return pDev->ErrCode;
                break;
            case XX_RELEASE_SELECT_WAIT:
```

```
            /* 唤醒任务列表上所有等待接收的任务 */
                selWakeupAll (&pDev->selWakeupList, SELREAD);
            break;
    default:
        errnoSet(Sio_errLib_function_not_define);
        return(ERROR);
    }
    return(OK);
    }
```

selWakeupAll()

函数原型：

```
void selWakeupAll
(
SEL_WAKEUP_LIST * pWakeupList,    /* 需要唤醒的任务列表 */
SELECT_TYPE type                  /* 接收者 (SELREAD)或发送者(SELWRITE) */
)
```

功能描述：

该函数将 select()函数阻塞的所有任务唤醒。当设备就绪时，驱动程序调用该函数通知系统。参数 type 指定了需要唤醒的任务是读任务或者是写任务。

返回值：

无。

参考：

相关信息请参考 selectLib 库描述。

selNodeAdd()

函数原型：

```
STATUS selNodeAdd
(
SEL_WAKEUP_LIST * pWakeupList,      /* 需要唤醒的任务列表 */
SEL_WAKEUP_NODE * pWakeupNode       /* 把该节点添加到列表上 */
)
```

功能描述：

该函数为设备的唤醒列表增加一个节点。通常是在驱动程序的 FIOSELECT 功能中调用。

返回值：

向 select()唤醒列表中增加节点成功，返回 OK，如果内存不足则返回 ERROR。

参考：

相关信息请参考 selectLib 库描述。

selNodeDelete()

函数原型：

```
STATUS selNodeDelete
(
SEL_WAKEUP_LIST * pWakeupList,      /* 需要唤醒的任务列表 */
SEL_WAKEUP_NODE * pWakeupNode       /* 从列表上删除该节点 */
```

)

功能描述：

该函数从指定的唤醒列表中删除节点，通常是驱动程序中的 FIOUNSELECT 函数调用。

返回值：

成功从 select()唤醒列表中删除节点，返回 OK，如果在唤醒列表中未发现指定节点则返回 ERROR。

参考：

相关信息请参考 selectLib 库描述。

selWakeupListInit()

函数原型：

```
void selWakeupListInit
(
SEL_WAKEUP_LIST * pWakeupList /* 初始化唤醒列表 */
)
```

功能描述：

该函数用于初始化唤醒列表，应在设备创建过程中调用。

返回值：

无。

参考：

相关信息请参考 selectLib 库描述。

例：

```
int   XXDrvNum ;      /* 驱动程序号 */
...
/* 设备创建函数 */
STATUS XXDevCreate(char *name,int cardno,int chno)
{
 XXDEV pDev;
 ...
 /* 初始化唤醒列表 */
      selWakeupListInit (&pDev->selWakeupList);
 /* 添加设备 */
 if(iosDevAdd((DEV_HDR *)pDev.devHdr,name,XXDrvNum) == ERROR)
 {
       errnoSet(Sio_errLib_device_create_error);
       return(ERROR);
 }
 pDev.isCreate = TRUE;
 pDev.ErrCode = 0;
 return(OK);
}
```

selWakeupListLen()

函数原型：

```
int selWakeupListLen
```

```
(
SEL_WAKEUP_LIST * pWakeupList   /* 唤醒任务列表 */
)
```

功能描述：

该函数返回指定的唤醒列表中节点的个数。在驱动程序中可以用于查看当前是否有任务被阻塞，是否应使用 selWakeupAll()来唤醒这些任务。

返回值：

在 select()唤醒列表中的节点个数，或者 ERROR。

参考：

相关信息请参考 selectLib 库描述。

selWakeupType()

函数原型：

```
SELECT_TYPE selWakeupType
(
SEL_WAKEUP_NODE * pWakeupNode      /* 节点 */
)
```

功能描述：

该函数返回指定唤醒节点的类型。通常用于驱动程序的 FIOSELECT 功能中确定设备所应用的 select()操作类型。

返回值：

SELREAD (读操作)或者 SELWRITE (写操作)。

参考：

相关信息请参考 selectLib 库描述。

第7章　文件系统API

7.1　与MS-DOS系统兼容的文件系统函数

7.1.1　函数库描述

1．库命名

与MS-DOS系统兼容的文件系统也就是dosFs文件系统，其函数库名称为dosFsLib。

2．函数

表7-1中列出了dosFs文件系统函数。

表7-1　dosFs文件系统函数

函　数	描　述	函　数	描　述
dosFsConfigGet()	获取dos卷配置值	dosFsMkfs()	安装设备并创建dosFs文件系统
dosFsConfigInit()	初始化dos卷配置结构	dosFsMkfsOptionsSet()	修改dosFs卷模式
dosFsConfigShow()	显示dos卷配置数据	dosFsModeChange()	为函数dosFsMkfs()指定卷选项
dosFsDateSet()	设置dosFs文件系统日期	dosFsReadyChange()	通知dosFs磁盘状态变更
dosFsDateTimeInstall()	安装用户提供的日期/时间函数	dosFsTimeSet()	设置dosFs文件系统时间
dosFsDevInit()	将块设备与dosFs文件系统相连接	dosFsVolOptionsGet()	获取当前dosFs卷选项
dosFsDevInitOptionsSet()	指定用于dosFsDevInit()函数的初始化数据	dosFsVolOptionsSet()	设置dosFs卷选项
dosFsInit()	初始化dosFs函数库	dosFsVolUnmount()	卸载dosFs卷

3．描述

该函数库提供了兼容MS-DOS®文件标准的函数支持。该模块提供必要的缓冲区管理、维护目录结构和文件系统细节的函数。

4．函数库的使用方法

VxWorks的Dos文件系统(dosFs)所提供的函数可以分成三组：通用初始化、设备初始化、文件系统操作。

函数dosFsInit()负责初始化功能：不论初始化多少dosFs设备，该函数只能调用一次。另外，如果使用了dosFsDateTimeInstall()，也只能调用一次，在执行任何实际文件操作之前调用这个安装函数，用来安装用户提供的日期和时间处理函数。

其他的dosFs函数用于设备初始化。对于每一个dosFs设备，可以调用dosFsDevInit()或者dosFsMkfs()来安装设备并定义其配置。函数dosFsConfigInit()提供了简单的初始化数据结构的功能。但是不一定用该函数来初始化配置数据。

还有一些函数用于设置文件系统的系统环境。dosFsDateSet()和dosFsTimeSet()函数用于设置日期和时间，通常用于未安装用户时间服务函数的情况。dosFsModeChange()函数可以用于修改实际设备的读写模式。

dosFsReadyChange()函数用于通知文件系统磁盘的转换等事件。最后，dosFsVolUnmount()函数将实际设备的同步和卸载事件告知文件系统，用于准备磁盘的变更。

5．初始化 DOSFS 函数库

在调用 dosFs 库中的其他函数之前，应先调用 dosFsInit()函数。该函数指定可以同时打开的 dosFs 文件的最大个数。企图打开多于指定数目的文件时，open()和 creat()函数将返回错误。

如果配置了 INCLUDE_DOSFS，VxWorks 将会通过 usrRoot()任务调用 dosFsInit()函数执行初始化过程。

6．定义 DOSFS 设备

设备描述符结构用来将文件系统用于实际的设备。该设备描述符的第一个成员必须是块设备描述符（BLK_DEV）。在使用 dosFsDevInit()函数之前就要初始化该结构。在 BLK_DEV 结构中，包含了驱动程序必须提供的五个函数的地址：读扇区、写扇区、执行设备 I/O 控制、检查设备状态、重置设备。这些函数将在下面描述。BLK_DEV 结构中同时包含了描述设备物理配置的信息。

dosFsDevInit()函数将设备与 dosFsLib 函数库相联系。它需要三个参数：

1）设备名字符串指针，用于标识设备。这将是 I/O 操作的路径名的一部分。这个名称将会保存在 VxWorks I/O 系统设备表中。使用 iosDevShow()函数可以看到。

2）指向 BLK_DEV 结构的指针，用于描述设备、保存五个必须的文件的地址。在使用 dosFsDevInit()函数前就要初始化该结构中的字段。

3）指向卷配置结构（DOS_VOL_CONFIG）的指针。该结构中包含卷的配置数据，用于文件系统的配置。在使用 dosFsDevInit()函数前就要初始化该结构中的字段。可以使用 dosFsConfigInit()函数初始化 DOS_VOL_CONFIG 结构。

一旦调用了 dosFsDevInit()函数，dosFsLib 每接到一个 I/O 请求，就会调用设备的驱动函数来访问设备。

dosFsMkfs()函数可以替代 dosFsDevInit()函数使用。该函数将会在设备上初始化一个新的 dosFs 文件系统，所以它不适于已经包含了需要保留的数据的磁盘。由于不再使用 DOS_VOL_CONFIG 结构，该函数使用默认的配置参数。

7.多逻辑设备的支持

传递给驱动读写函数的扇区号是绝对扇区号,从设备的零扇区开始编号。如果有必要，驱动可以将逻辑地址的偏移量加到物理设备的起始地址上。可以通过传送偏移量参数到驱动设备结构中去，并将文件系统传递的扇区号加上偏移量来实现。

8．以原始模式访问磁盘

如果在 open()或者 creat()函数的文件名参数中传入空文件名，dosFs 文件系统就会以原始方式访问整个磁盘，而不仅仅访问磁盘上的一个文件（也就是在调用 open()或者 creat()时使用设备名作为文件名）。

原始模式仅仅意味着访问没有文件系统的磁盘。例如，在磁盘上初始化一个新的文件系统，首先打开一个原始磁盘并返回文件描述符，然后使用 ioctl()函数的 FIODISKINIT 功能。有时在使用 ioctl()函数的其他功能时也会用原始模式访问磁盘。（如 FIONFREE, FIOLABELGET）。

若要读取一个没有文件名的磁盘的根目录，可以使用 opendir()。跟着调用 readdir()可以返回根目录下的文件名和子目录名。

在原始模式下将数据写到磁盘使用与通常操作相同的区域。原始模式不会使用磁盘引导扇区、根目录区或者文件分配表所在扇区。

9．设备名和路径名

在一个 MS-DOS 机上，磁盘设备名一般为“A：”，也就是一个字母加上冒号。这样的名字可以使用在 VxWorks 系统中。然而也可以使用更长的设备名，如“DOS1：”或者“/floppy0/”。该设备名由函数 dosFsDevInit()或者 dosFsMkfs()指明。

用于指定文件名的参数可以使用正反斜杠（但是 VxWorks 不允许在文件名中使用正斜杠“/”，这一点值得读者注意）。

当使用 VxWorks 的 Shell 工具调用指定文件名的函数时，用户必须使 C 语法解释器可以正确识别。即使

用引号将其括起，并为“\”使用“\\”，如：

-> copy ("DOS1:\\subdir\\file1", "file2")

而使用 shell 命令时，就不需要这样做。如：

-> copy < DOS1:\subdir\file1

这其中要注意的是，设备名与文件名之间的斜杠是不需要的，也就是说"DOS1:newfile.new"和"DOS1:/newfile.new"是指的同一个文件。

10．使用长文件名

MS-DOS 标准只允许使用 8.3 格式的文件名。但是在远程系统传输文件或者用户应用转换文件名时这会产生巨大的不便。

基于上述原因，dosFs 文件系统提供了可选的长文件名支持。如果使用这个选项，文件名可以包含 40 个 ASCII 字符。并且不需要执行大小写转换，也不限制使用的字符。

注意事项

由于磁盘上使用了特殊的目录，使用长文件名的磁盘不兼容 MS-DOS 系统，不能在 MS-DOS 机上使用。使用长文件名的磁盘必须由 VxWorks dosFs 文件系统初始化（使用 FIODISKINIT），使用 MS-DOS 系统初始化的磁盘则不能使用该选项。

在 DOS_VOL_CONFIG 结构中的 dosvc_options 字段指定 DOS_OPT_LONGNAMES 位用于 dosFsDevInit()函数可以使能长文件名支持。（函数 dosFsMkfs()也可以使能长文件名支持，但是在调用该函数之前必须使用 dosFsMkfsOptionsSet()函数设置 DOS_OPT_LONGNAMES 选项）。

11．网络文件系统支持

当使用函数 dosFsDevInit()或者 dosFsMkfs()初始化带有 DOS_OPT_EXPORT 选项的磁盘时，可以打开文件系统输出支持。如果未清楚地给出磁盘配置，则 dosFs 系统默认支持该选项。

如果所加载的 dosFs 卷是基于 PC 的远程客户端，则应使用 DOS_OPT_LOWERCASE 选项。该选项使文件名全部转化为小写字母（不使用 DOS_OPT_LONGNAMES 选项时）。有很多基于 PC 的 NFS 系统都需要该映射。

使能了 DOS_OPT_EXPORT 选项后，VxWorks NFS 文件系统使用保留的 dosFs 目录来保存 dosFs 文件的惟一标识信息。

任何时候在目录中创建文件，目录的时间戳都会增长，这避免了缓存的不同步。

用户也可以在初始化 NFS 时为用户 ID、组 ID、文件访问权限指定整数值。该值将用于系统中的所有文件。设置 dosFsUserId 用于指定用户 ID 号，默认为 65534。设置 dosFsGroupId 用于指定组 ID 号，默认为 65534。设置 dosFsFileMode 用于指定文件访问号，默认为 777。

12．获取目录项

用户可以使用 opendir()，readdir()，rewinddir()以及 closedir()函数搜索 VxWorks dosFs 卷。这些调用允许确定子目录名或者文件名。

使用 fstat()或者 stat()函数可以获取关于文件的更多信息。除了标准的文件信息，这些函数所使用的结构还会返回文件属性字节。

13．文件的时间和日期

dosFs 卷的目录项都会包含文件或者目录的时间和日期。文件的时间是在创建、关闭更新时设置。目录的时间是在创建时设置。

dosFs 文件系统库使用内部的结构保存时间和日期。由于没有自动的时间更新机制，可以使用两种方法设置时间。

第一种方法是将当前时间通过 dosFsDateSet() 和 dosFsTimeSet()函数设置。

示例如下：

```
dosFsDateSet (1990, 12, 25);        /* 设置日期为 Dec-25-1990 */
dosFsTimeSet (14, 30, 22);          /* 设置时间为 14:30:22 */
```

第二种方法需要用户提供一个钩子函数。如果使用了 dosFsDateTimeInstall()函数安装了时间日期钩子函数，dosFsLib 在需要当前日期时会调用该函数。该工具提供了使用硬件时钟的途径。

日期/时间钩子函数应定义如下：

```
void dateTimeHook
(
DOS_DATE_TIME *pDateTime   /* 指向 dosFs 日期/时间结构的指针 */
)
```

参数中的 DOS_DATE_TIME 结构应该包含最新设置的时间和日期。钩子函数会将其包含的值置为当前的日期和时间。未改变的字段将保持原值。

MS-DOS 规范仅为文件时间提供了 2 秒的精确度。如果有函数 dosFsTimeSet()或者日期/时间钩子函数设置的秒数是奇数，则将会被转为下一秒。

dosFsLib 中使用的时间被初始化为 Jan-01-1980, 00:00:00。

14．文件属性

dosFs 卷的目录项中包含了文件的属性字节。属性定义了下面几种状态：只读文件、隐藏文件、系统文件、卷标、目录标识、文档标识。下面列出了这些属性的 VxWorks 定义：

DOS_ATTR_RDONLY：只读文件

DOS_ATTR_HIDDEN：隐藏文件

DOS_ATTR_SYSTEM：系统文件

DOS_ATTR_VOL_LABEL：卷标

DOS_ATTR_DIRECTORY：目录标识

DOS_ATTR_ARCHIVE：文档标识

属性字节中的所有标识，除了目录标识和卷标标识之外，都可以使用 ioctl()函数的 FIOATTRIBSET 功能来设置或清除。打开文件之后就可以调用该功能，它会将指定的文件属性直接复制到目录项中。如果希望保存已有的标识设置，应首先使用函数 fstat()获取文件属性，然后修改相应的位。如下可以使文件改为只读并保留其他的标识：

```
struct stat fileStat;
fd = open ("file", O_RDONLY, 0);          /* 打开文件 */
fstat (fd, &fileStat);                    /* 获得文件状态 */
ioctl (fd, FIOATTRIBSET, (fileStat.st_attrib | DOS_ATTR_RDONLY));
/* 设置只读标识 */
close (fd);    /* 关闭文件 */
```

15．连续文件支持

VxWorks 的 dosFs 文件系统提供了对连续文件的支持，也就是说支持连续扇区所组成的文件，包含创建连续文件和对连续文件访问的优化。

如果需要创建连续文件，首先以通常的方式创建文件。然后使用返回的文件描述符调用 ioctl()函数的 FIOCONTIG 功能。FIOCONTIG 功能的另一个参数是所要分配的字节数。也可以将该参数设置为 CONTIG_MAX (-1)，为文件申请磁盘上尽可能大的连续空间。

FAT 将会搜索合适的空间，如果找到则将其分配给文件。（如果没有足够的空间可以分配则返回错误 S_dosFsLib_NO_CONTIG_SPACE），然后就可以将文件关闭用于未来的操作了。

例：

```
/* 创建一个连续事件并为其分配 0×10000 字节 */
fd = creat ("file", O_RDWR, 0);                    /* 打开文件 */
status = ioctl (fd, FIOCONTIG, 0x10000);           /* 获得连续区域 */
```

```
if (status != OK)
... /* 错误处理 */
close (fd); /* 关闭文件 */
```

例：

```
/* 创建一个连续文件并为其分配磁盘上尽可能大的字节数 */
fd = creat ("file", O_RDWR, 0);                    /* 打开文件 */
status = ioctl (fd, FIOCONTIG, CONTIG_MAX);        /*获得连续区域 */
if (status != OK)
... /* 执行错误处理 */
close (fd); /* 关闭文件 */
```

ioctl()函数所使用的文件描述符必须是新创建的文件。而且由于文件可能被分配在任何位置，所以必须在向文件写入任何数据之前执行 FIOCONTIG 操作。

使用函数 fstat()检查文件大小可以获取使用 CONTIG_MAX 创建的文件实际分配的字节数。

可以使用 ioctl()函数的 FIOTRUNC 功能释放分配给文件的空间。

目录也可以分配到连续的扇区。像标准文件一样，使用 FIOCONTIG 功能为目录分配空间。所使用的目录应该为空。根目录的分配不能改变。分配给目录的空间在目录删除之前不能使用，而且不可以使用 FIOTRUNC 功能。

任何文件打开时都会检查连续性。如果一个文件连续，系统将会使用很多优化技术来访问文件。不论文件是否使用上述办法分配，都可以享受优化技术提高性能。

16．更换磁盘、卸载磁盘和同步磁盘

目录结构信息和 FAT 表的复制都在内存中。这样会极大地提高访问效率，但是当磁盘变更时必须通知 dosFsLib。有两种机制可以完成这个任务。

1）卸载磁盘卷

首选的方法是通过调用 dosFsVolUnmount()函数卸载磁盘。该调用会将所有缓存中修改的数据保存到磁盘上，并且令所有打开的文件描述符失效。在下一次 I/O 操作前，应重新装载磁盘。装载磁盘的操作可以通过 ioctl()的 FIOUNMOUNT 功能，也可以调用 dosFsVolUnmount()函数，此时可以使用任何文件描述符来执行 ioctl()。

当卸载磁盘卷的时候可能还会有打开的文件或者目录，卸载磁盘后这些文件描述符将会失效。所有试图访问描述符的操作将会返回 S_dosFsLib_FD_OBSOLETE 错误。用户可以使用 close()函数释放这些文件描述符，但是该操作仍然会返回 S_dosFsLib_FD_OBSOLETE 错误。在执行 dosFsVolUnmount()函数后，以原始方式打开的文件描述符将不会被指为无效，并且可以继续使用。

中断处理函数不能直接调用 dosFsVolUnmount()函数，因为该函数会等待设备的就绪。中断处理函数可以使用释放信号量的方式通知任务执行 dosFsVolUnmount()函数（注意函数 dosFsReadyChange()可以用于中断处理函数）。

当调用 dosFsVolUnmount()函数时，它会试图将数据写回到磁盘。但是如果磁盘已经更换则不适合使用该方法（因为旧缓冲区中的数据将会写入到新的磁盘中）。此时就需要用到 dosFsReadyChange()函数。

如果在磁盘更换以后才调用 dosFsVolUnmount()函数，数据写将会出错。然而，该函数仍然会使相关文件描述符失效，而系统仍然会在这之后重新加载磁盘。这时 dosFsVolUnmount()函数将不会返回错误。为了避免这样的情况发生应该在磁盘更换前进行同步操作。

如果磁盘卷是由 usrFdConfig()函数加载，不要对该磁盘使用 dosFsVolUnmount()函数。因为 usrFdConfig()函数并不返回 DOS_VOL_CONFIG 结构。这时可以使用 ioctl()函数的 FIOUNMOUNT 功能卸载磁盘。

2）声明磁盘就绪状态的改变

将磁盘更换的消息通知给 dosFsLib 的第二种方法是通过“就绪状态改变”机制。磁盘就绪状态改变被

dosFsLib 解释为下一次对磁盘的 I/O 访问前需要重新装载磁盘。

总共有三种方法声明磁盘状态的改变。第一，可以直接调用 dosFsReadyChange()函数。其次，可以使用 ioctl()函数的 FIODISKCHANGE 功能。最后，设备驱动可以设置 BLK_DEV 结构中的 bd_readyChanged 字段为 TRUE。这三种办法的效果都是相同的。

就绪状态改变机制不执行数据的同步操作。仅仅是标志磁盘卷需要重新装载，其结果是，缓冲区中的数据（文件的数据、目录条目、FAT 表的修改）将会丢失。可以在申明就绪状态改变之前执行同步操作来避免这种情况（这两步操作的组合等同于 dosFsVolUnmount()函数，但是不会将文件描述符标识置为无效）。

由于不执行数据同步操作，该机制可以用在中断处理函数中。

3）没有更换通知机制的磁盘

在实际的情况中，不可能每次磁盘更换时都会调用 dosFsVolUnmount() 或者 dosFsReadyChange()函数，设备必须在初始化的时候就声称惟一的标识。函数 dosFsDevInit()的一个参数是 DOS_VOL_CONFIG 的地址，用于指定不同的配置参数。如果驱动程序或者应用程序无法保证在每一次磁盘更换的时候都通知系统，那么必须设置 DOS_VOL_CONFIG 结构的 dosvc_options 字段的 DOS_OPT_CHANGENOWARN 选项。

该配置选项会降低性能，因为磁盘配置数据必须有规律地从磁盘中读取，以确认磁盘的更换。另外设置该选项还会自动执行同步操作。

注意

如果要将磁盘的更换通知系统，必须每次都调用 dosFsVolUnmount()或者 dosFsReadyChange()函数。不论在中断处理函数中或者应用程序中都可以。例如，如果应用程序中提供了调用 dosFsVolUnmount()的用户接口，那就不需要设置 DOS_OPT_CHANGENOWARN 选项了。

4）同步操作

磁盘在卸载之前应该执行同步操作。同步操作即是将所有的输出缓冲区中的数据写到磁盘上（包括文件、目录、FAT 表等）。如果使用 dosFsVolUnmount()函数卸载磁盘，则不需要执行同步操作。

调用 dosFsVolUnmount()函数卸载磁盘卷之前，该函数会试图执行同步操作。如果在调用 dosFsVolUnmount()函数的时候磁盘仍然可以读写,则会成功执行同步操作,这时就不需要单独执行同步操作。

然而，在磁盘已经移除之后再执行 dosFsVolUnmount()函数则为时晚矣（这种情况 dosFsVolUnmount()函数会丢弃所有缓存的数据）。所以在磁盘移除前应调用 ioctl()的 FIOFLUSH 或者 FIOSYNC 功能。

5）自动同步模式

dosFs 文件系统提供了自动同步功能。在调用 dosFsDevInit()函数前设置 DOS_VOL_CONFIG 结构的 dosvc_options 字段为 DOS_OPT_AUTOSYNC 即可。当使能该选项时，应用所修改的目录项、FAT 表等数据会立刻写入到磁盘中（通常在文件关闭以前不会将这些数据写回）。当然这降低了性能，但是提高了数据的安全性。

如果磁盘卷无法产生移除通知，系统会自动使能自动同步，例如设置了 DOS_VOL_CONFIG 结构的 dosvc_options 字段的 DOS_OPT_CHANGENOWARN 选项。若对数据的完整性有较高要求的时候，也应该使用该选项。

17. 改变磁盘卷的配置信息

第一次使用 dosFsDevInit()函数初始化 dosFs 设备时，应为磁盘指定不同的配置参数。该数据保存在卷描述符 DOS_VOL_DESC 结构中。但是，有可能使用不同参数初始化的磁盘却使用相同的驱动器。所以当一个新的磁盘加载上之后必须修改卷描述符中的配置数据。

当加载磁盘时，系统将会读取引导扇区的信息。该数据用于更新卷描述符中的配置数据。这个过程发生在卸载上一个磁盘后的第一次对磁盘的 I/O 访问时。

这个自动重新初始化步骤有两个重要的意义：

1）由于卷描述符中的数据有所改变,有可能会忽略 dosFs 配置数据（为 dosFsDevInit()函数指定一个 NULL 的指针而不是 DOS_VOL_CONFIG 结构）。第一次使用磁盘卷必须面向一个已经格式化过并初始化文件系统

的磁盘。

2）当初始化磁盘时将会用到卷描述符数据。FIODISKINIT 功能使用最近加载的磁盘的参数初始化磁盘，而不使用 dosFsDevInit()所使用的原始数据。所以，推荐在调用 dosFsDevInit()函数之后立即执行 FIODISKINIT 功能（设备应以原始模式打开，然后执行 FIODISKINIT 功能，然后关闭设备）。

18．IOCTL 功能字

dosFs 文件系统支持下面的 ioctl()功能。这些功能均定义在 ioLib.h 中。除非特殊说明，这里所用到的文件描述符可以是磁盘卷上的任意描述符。

FIODISKFORMAT

使用适当的硬件设置格式化整个磁盘。但是不在磁盘上初始化任何文件系统。注意这时驱动程序提供的功能。

```
fd = open ("DEV1:", O_WRONLY);
status = ioctl (fd, FIODISKFORMAT, 0);
```

FIODISKINIT

在磁盘卷上初始化 DOS 文件系统。该功能并不格式化磁盘，格式化的操作应由驱动程序来完成。所使用的文件描述符应该是以原始方式打开的磁盘卷。

```
fd = open ("DEV1:", O_WRONLY);
status = ioctl (fd, FIODISKINIT, 0);
```

FIODISKCHANGE

声明媒体更换，它执行与 dosFsReadyChange()相同的功能。该函数可以在中断处理函数中调用。

```
status = ioctl (fd, FIODISKCHANGE, 0);
```

FIOUNMOUNT

卸载磁盘卷，它执行与函数 dosFsVolUnmount()相同的功能。该函数不能在中断处理函数中调用。

```
status = ioctl (fd, FIOUNMOUNT, 0);
```

FIOGETNAME

获取指定文件描述符的文件名，并将其复制到指定的缓冲区中。

```
status = ioctl (fd, FIOGETNAME, &nameBuf );
```

FIORENAME

将文件名或目录名改为新名称。

```
status = ioctl (fd, FIORENAME, "newname");
```

FIOSEEK

设置文件指针的字节偏移量。

```
status = ioctl (fd, FIOSEEK, newOffset);
```

FIOWHERE

返回文件指针的当前位置。这里返回的是将要读写的字节的位置。该功能不需要其他的参数。

```
position = ioctl (fd, FIOWHERE, 0);
```

FIOFLUSH

同步文件输出缓存。该功能保证任何输出都实际写入到磁盘设备中。如果所使用的文件描述符是以原始方式打开的整个磁盘卷，该功能会将所有的缓冲同步，其中包括文件、目录、FAT 表等。

```
status = ioctl (fd, FIOFLUSH, 0);
```

FIOSYNC

执行与 FIOFLUSH 相同的功能，并且重新读取磁盘中的数据。这使得文件可以有多个描述符操作并保持同步。

FIOTRUNC

将指定文件的长度置为 newLength。所指定长度之外的已分配磁盘簇都会被释放。只有文件才可以进行

该操作，如果对目录或者整个磁盘卷尝试该操作将会返回错误。而且该操作仅会截短文件，如果参数 newLength 大于文件长度将会返回错误，其错误号为 S_dosFsLib_INVALID_NUMBER_OF_BYTES。

```
status = ioctl (fd, FIOTRUNC, newLength);
```

FIONREAD

将文件中未读取的字符数量返回到 unreadCount 参数中。

```
status = ioctl (fd, FIONREAD, &unreadCount);
```

FIONFREE

获取剩余空间的字节数。

```
status = ioctl (fd, FIONFREE, &freeCount);
```

FIOMKDIR

创建一个新目录。

```
status = ioctl (fd, FIOMKDIR, "dirName");
```

FIORMDIR

删除由 dirName 参数指定的目录。

```
status = ioctl (fd, FIORMDIR, "dirName");
```

FIOLABELGET

获取磁盘卷标并将其复制到指定的缓冲区。

```
status = ioctl (fd, FIOLABELGET, &labelBuffer);
```

FIOLABELSET

设置卷标，所指定的卷标不能超过 11 个 ASCII 字符。

```
status = ioctl (fd, FIOLABELSET, "newLabel");
```

FIOATTRIBSET

设置文件属性字节。文件描述符参数用于指定相应的文件。

```
status = ioctl (fd, FIOATTRIBSET, newAttrib);
```

FIOCONTIG

为文件或者目录分配连续的磁盘空间。所需空间由参数 bytesRequested 指定。通常在文件创建时直接分配连续空间。

```
status = ioctl (fd, FIOCONTIG, bytesRequested);
```

FIONCONTIG

获取磁盘上最大连续空间的尺寸。

```
status = ioctl (fd, FIONCONTIG, &maxContigBytes);
```

FIOREADDIR

读取下一个目录项，参数 dirStruct 是目录描述符。通常，用户会使用 readdir()函数读取目录，而不是直接使用 FIOREADDIR 功能。

```
DIR dirStruct;
fd = open ("directory", O_RDONLY);
status = ioctl (fd, FIOREADDIR, &dirStruct);
```

FIOFSTATGET

获取文件状态信息（目录项数据）。参数 statStruct 是状态结构的指针。通常，使用 stat() 或者 fstat()获取文件信息，而不是用该功能。

```
struct stat statStruct;
fd = open ("file", O_RDONLY);
status = ioctl (fd, FIOFSTATGET, &statStruct);
```

其他的 ioctl()命令都会传给块设备驱动程序来处理。

19．内存使用

为了使文件系统的内存使用量降到最低，所有的内存使用都被限制在专用的内存分区中。可以通过全局变量 dosFsMemPartId 访问该内存分区。

如果要显示 dosFsLib 所使用的内存总量，可以调用 show(dosFsMemPartId)。

如果希望改变 dosFsLib 的内存管理方式可以修改相应的变量。

如果不存在 dosFsLib 内存分区，系统会在系统内存池中创建一个默认的大小是 8K 的内存分区。可以通过修改全局变量 dosFsMenPartInitSize 来确定所需使用的内存池大小。如果指定内存池，将变量 dosFsMemPartId 设置为函数 memPartCreate()返回的 PART_ID 即可。

修改全局变量 dosFsMemPartIdOptions 可以改变 malloc() 和 free()的错误处理模式。默认的处理方式是 MEM_BLOCK_ERROR_LOG_FLAG，使用该模式将会记录 free()函数所检测到的错误。该选项仅仅影响对 dosFs 内存区的操作。

如果有必要的话，已有的内存分区可以自动增长。系统会从系统内存池中分配新的内存，至少 1K 大小。通过全局变量 dosFsMemPartGrowSize 可以设置该尺寸的默认值。

用户可以通过全局变量 dosFsMemPartCap 限制 dosFs 所使用的最大内存池的尺寸。一旦到达指定的尺寸，dosFs 将不会再从系统内存池中分配内存。该变量的默认值是-1，意味着不限制内存的分配。

最后可以通过设置全局变量 dosFsDebug 使能调试功能。设置该变量为一表示使能 dosFs 内存管理的详细调试信息。

20．头文件

dosFs 文件系统函数声明在 dosFsLib.h 头文件中。

21．参考

相关信息请参考 dosFsLib、ioLib、iosLib、dirLib、ramDrv 库描述。

7.1.2 dosFs 文件系统函数详细描述

dosFsConfigGet()

函数原型：

```
STATUS dosFsConfigGet
(
DOS_VOL_DESC * vdptr,        /* 指向卷描述符的指针 */
DOS_VOL_CONFIG * pConfig     /* 指向配置结构的指针 */
)
```

功能描述：

该函数获取 dosFs 磁盘卷的配置值。所需要的数据来自于指定的磁盘卷的描述符，该函数不对物理设备进行操作。

配置数据放置在 DOS_VOL_CONFIG 结构中，由指针 pConfig 指向，在调用 dosFsConfigGet()函数之前必须为该结构分配空间。

该函数可以从一个已知的磁盘上读取配置数据，并将其初始化为一个新盘（初始化使用 dosFsDevInit()函数）

从磁盘卷描述符中读取数据不会锁定磁盘卷，所以要注意不要使其他任务在读取的同时修改数据。

返回值：

OK 或者 ERROR。

参考：

相关信息请参考 dosFsLib 库描述。

dosFsConfigInit()

函数原型：

```
STATUS dosFsConfigInit
(
DOS_VOL_CONFIG * pConfig,   /* 指向卷配置结构的指针 */
char mediaByte,             /* 媒介描述符字节 */
UINT8 secPerClust,          /* 每个簇包含扇区数 */
short nResrvd,              /* 预留扇区数 */
char nFats,                 /* 文件分配表副本数 */
UINT16 secPerFat,           /* 每个文件分配表副本包含扇区数 */
short maxRootEnts,          /* 根目录下最大子目录数 */
UINT nHidden,               /* 隐藏扇区数 */
UINT options                /* 卷选项 */
)
```

功能描述：

该函数初始化 dosFs 磁盘卷配置结构（DOS_VOL_CONFIG）。dosFsDevInit()函数使用这个结构为磁盘指定文件系统配置。

结构 DOS_VOL_CONFIG 必须在使用该函数之前分配，其地址由参数 pConfig 决定，该结构中包含了指定的配置变量。

该函数仅为方便 DOS_VOL_CONFIG 的初始化而提供。使用其他办法初始化该结构也一样可以用于 dosFsDevInit()函数。

返回值：

成功初始化则返回 OK，如果输入了无效参数或者 pConfig 参数为 NULL 则返回 ERROR。

参考：

相关信息请参考 dosFsLib 库描述、dosFsDevInit()函数。

dosFsConfigShow()

函数原型：

```
STATUS dosFsConfigShow
(
char * devName /* 设备名 */
)
```

功能描述：

该函数获取指定的 dosFs 卷的配置，并将其以一定的格式显示在标准输出设备上。显示的信息主要来自于结构 DOS_VOL_CONFIG 以及其他的配置信息（例如 BLK_DEV 中存储的设备描述信息）。

如果未指定文件名，则获取当前设备的描述。

返回值：

成功获得则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 dosFsLib 库描述。

操作示范：

在宿主机 WindSh 工具中，dosFsConfigShow()函数操作示范如下所示：

```
-> dosFsConfigShow "/ataCtrl0Part1"
```

```
    device name:                /ataCtrl0Part1
    total number of sectors:    2100609
    bytes per sector:           512
    media byte:                 0xf0
    # of sectors per cluster:   33
    # of reserved sectors:      1
    # of FAT tables:            2
    # of sectors per FAT:       249
    max # of root dir entries:  112
    # of hidden sectors:        0
    removable medium:           TRUE
    disk change w/out warning: not enabled
    auto-sync mode:             not enabled
    long file names:            ENABLED
    exportable file system:     not enabled
    lowercase-only filenames:   not enabled
    volume mode:                O_RDWR (read/write)
    available space:            1075244544 bytes
    max avail. contig space:    1075244544 bytes
```

dosFsDateSet()

函数原型：

```
STATUS dosFsDateSet
(
int year,    /* 年 (1980…2099) */
int month,   /* 月 (1…12) */
int day      /* 日 (1…31) */
)
```

功能描述：

该函数为 dosFs 文件系统卷设置日期。所有文件的创建或者修改都会标着这个日期。

注意：

日期不会自动增长，新的日期必须通过该函数自行设置。

返回值：

成功设置则返回 OK，如果日期无效则返回 ERROR。

参考：

相关信息请参考 dosFsLib 库描述、dosFsTimeSet()及 dosFsDateTimeInstall()函数。

dosFsDateTimeInstall()

函数原型：

```
void dosFsDateTimeInstall
(
FUNCPTR pDateTimeFunc  /* 指向用户提供的函数指针 */
)
```

功能描述：

该函数将用户提供的日期/时间函数安装到系统中。一旦安装了这函数，dosFsLib 会在需要获取时间日期时调用该函数。否则，将使用最近一次由 dosFsDateSet()和 dosFsTimeSet()设置的日期和时间。

所使用的用户函数必须只有一个输入参数，也就是 DOS_DATE_TIME 结构的地址指针（在 dosFsLib.h 中定义）用户的函数必须更新该结构中必要的字段，用户函数所不改变的值将保持其原值。

返回值：

无。

参考：

相关信息请参考 dosFsLib 库描述。

例：

```
#define CENTURY        1900
...
dosFsDateTimeInstall (rtcHook);        /* 安装用户提供的日期/时间函数 */
...
/* 日期和时间钩子函数 */
void rtcHook
    (
    DOS_DATE_TIME *pDateTime
    )
    {
    /* 更新本地日期和时间结构 */
    rtcSpIoctl (GET_DATE, FALSE);
    rtcSpIoctl (GET_TIME, FALSE);
...
    /* 给 DOS 日期和时间结构赋值 */
    pDateTime->dosdt_year = CENTURY + rtc_Year;    /* 年 */
    pDateTime->dosdt_month = rtc_Month;            /* 月 */
    pDateTime->dosdt_day = rtc_DayOfMonth;         /* 日 */
    pDateTime->dosdt_hour = rtc_Hour;              /* 时 */
    pDateTime->dosdt_minute = rtc_Minute;          /* 分 */
    pDateTime->dosdt_second = rtc_Second;          /* 秒 */
    }
```

dosFsDevInit()

函数原型：

```
DOS_VOL_DESC *dosFsDevInit
(
char * devName,                    /* 设备名 */
BLK_DEV * pBlkDev,                 /* 指向块设备结构的指针 */
DOS_VOL_CONFIG * pConfig           /* 指向卷配置数据的指针 */
)
```

功能描述：

该函数将设备驱动程序创建的设备结构 BLK_DEV 定义为一个 dosFS 卷。函数执行之后，系统会将所有对设备的高级 I/O 访问指向 dosFsLib，调用其中的函数。参数 pBlkDev 是指向设备 BLK_DEV 结构的指针。

该函数会将指定的 devName 与设备相连并将其安装到 VxWorks I/O 系统设备表中。所使用的驱动号就是

由 dosFsInit()函数获取的驱动号（放置在全局变量 dosFsDrvNum 中）。

BLK_DEV 结构中包含了描述设备所需要的配置数据以及访问设备所使用的 5 个函数的地址。这些函数只有在一系列操作时才会被调用。

参数 pConfig 是指向 DOS_VOL_CONFIG 数据结构的指针，该结构必须使用指定的 dosFs 配置数据预先初始化。该结构可以使用 dosFsConfigInit()函数进行简单的初始化。

如果设备已经被初始化过，pConfig 参数可以为 NULL，此时，系统将会加载磁盘卷并从磁盘的引导扇区中读取配置数据。（如果 pConfig 参数为 NULL，change-no-warn 和 auto-sync options 将会被初始化为禁止，可以使用函数 dosFsVolOptionsSet()将其打开）。

该函数会为设备分配并初始化卷描述符（DOS_VOL_DESC）并返回该结构的指针。

返回值：

指向卷描述符 DOS_VOL_DESC 结构的指针，如果出现错误则返回 ERROR。

参考：

相关信息请参考 dosFsLib 库描述、dosFsMkfs()函数。

dosFsDevInitOptionsSet()

函数原型：

```
STATUS dosFsDevInitOptionsSet
(
UINT options /* 为将来 dosFsDevInit()调用准备的选项 */
)
```

功能描述：

该函数设置 dosFsDevInit()函数执行时所使用的卷选项，而不需要在 DOS_VOL_CONFIG 结构中提供明确的配置信息。通常在加载一个已经有系统数据初始化过的磁盘卷时使用。options 的值将会用于 dosFsDevInit()函数所初始化的所有的磁盘卷，除非另外指定配置信息。

这里所要指定的选项是那些与磁盘数据无关的选项。但是，不能在这里指定长文件名支持选项。如果磁盘支持该选项，其加载时会自动检测。如果指定了无法设置的参数，该函数会将其忽略，其余的有效位将会作为初始化磁盘卷的选项。

例如，使用函数 dosFsDevInit()初始化磁盘卷，设置选项自动同步和文件系统输出，则：

```
status = dosFsDevInitOptionsSet (DOS_OPT_AUTOSYNC |DOS_OPT_EXPORT);
if (status != OK)
return (ERROR);
vdptr = dosFsDevInit ("DEV1:", pBlkDev, NULL);
```

返回值：

成功设置则返回 OK，如果参数无效则返回 ERROR。

参考：

相关信息请参考 dosFsLib 库描述以及 dosFsDevInit(), dosFsVolOptionsSet()函数。

dosFsInit()

函数原型：

```
STATUS dosFsInit
(
int maxFiles        /* 同时打开 dosFs 文件的最大个数 */
)
```

功能描述：

该函数初始化 dosFs 库。在该库中任何其他函数调用之前，必须而且只能调用一次该函数。该函数将 dosFsLib 作为 VxWorks 的驱动安装在 I/O 系统驱动表中，分配并设置必要的内存并初始化信号量。对应于驱动程序表中的驱动号保存在全局变量 dosFsDrvNum 中。

如果定义了 INCLUDE_DOSFS，则系统会在根任务 usrRoot()中调用 dosFsInit()函数。

返回值：

初始化成功则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 dosFsLib 库描述。

dosFsMkfs()

函数原型：

```
DOS_VOL_DESC *dosFsMkfs
(
char * volName,        /* 卷名 */
BLK_DEV * pBlkDev /* 指向块设备结构 */
)
```

功能描述：

该函数提供了在设备上快速创建 dosFs 文件系统的方法。它由调用 ioctl()函数的 FIODISKINIT 功能并且调用 dosFsDevInit()函数这两个步骤组成。

该调用使用 dosFs 配置参数中默认的值（例如，DOS_VOL_CONFIG 中的配置信息），通常是：

2：每簇的扇区数；

1：保留的扇区；

2：FAT 表的备份数量；

112：根目录项；

0xF0：媒体字节值；

0：隐藏字节。

卷选项自动同步、长文件名、不可更换等可以使用函数 dosFsMkfsOptionsSet()设置，并由该函数用于初始化。默认的情况下，这些选项不用于 dosFsMkfs()的初始化。

如果初始化一个较大的磁盘，使用每簇 2 扇区的设置可能导致整个磁盘区不能包含大于 64K 簇的描述。这时，dosFsMkfs()会自动增加每簇的扇区数量，使得 64K 的簇可以包含整个磁盘容量。

每个 FAT 备份数量的扇区数设置为可以包含 FAT 表的最小数量的扇区。

返回值：

成功创建则返回 OK，如果磁盘卷未加载则返回 ERROR

参考：

相关信息请参考 dosFsLib 库描述以及函数 dosFsReadyChange()。

例：

```
BLK_DEV *      pSbdFloppy;          /* 一个指向块设备的指针 */
SCSI_PHYS_DEV *   pSpd31;           /* 一个 SCSI_PHYS_DEV 结构指针 */
IMPORT SCSI_CTRL *pSysScsiCtrl;
...
/* SCSI 设备安装 */
STATUS sysScsiConfig (void)
{
...
```

```
/* 配置设备 busId = 3, LUN(Logical Unit Number) = 1 */
if ((pSpd31 = scsiPhysDevCreate (pSysScsiCtrl, 3, 1, 0, NONE, 0, 0, 0))
 == (SCSI_PHYS_DEV *) NULL)
 {
        printErr ("usrScsiConfig: scsiPhysDevCreate failed.\n");
 return (ERROR);
 }
...
/* 创建块设备 */
    if ((pSbdFloppy = scsiBlkDevCreate (pSpd31, 0, 0)) == NULL)
 {
        printErr ("usrScsiConfig: scsiBlkDevCreate failed.\n");
 return (ERROR);
 }
/* 初始化块设备 */
scsiBlkDevInit ((SCSI_BLK_DEV *) pSbdFloppy, 15, 2);
/* 将块设备与 dosFs 文件系统相连接 */
if (dosFsDevInit ("/fd0/", pSbdFloppy, NULL) == NULL)
 {
        printErr ("usrScsiConfig: dosFsDevInit failed.\n");
 return (ERROR);
 }
/* 在设备上快速创建 dosFs 文件系统 */
if(dosFsMkfs("/fd0/", pSbdFloppy) == (DOS_VOL_DESC *)NULL)
    {
     printErr ("dosFsMkfs failed.\n");
 return (ERROR);
 }
return (OK);
}
```

dosFsMkfsOptionsSet()

函数原型：

```
STATUS dosFsMkfsOptionsSet
(
UINT options     /*  卷选项 */
)
```

功能描述：

该函数允许在调用函数 dosFsMkfs()时设置卷选项。选项的值将用于 dosFsMkfs()初始化的所有磁盘卷。例如，若初始化时希望使用自动同步和长文件名支持选项，则：

```
status = dosFsMkfsOptionsSet (DOS_OPT_AUTOSYNC |
DOS_OPT_LONGNAMES);
if (status != OK)
return (ERROR);
```

vdptr = dosFsMkfs ("DEV1:", pBlkDev);

返回值：

成功设置卷选项则返回 OK，如果选项无效则返回 ERROR。

参考：

相关信息请参考 dosFsLib 库以及函数 dosFsMkfs(), dosFsVolOptionsSet()的描述。

dosFsModeChange()

函数原型：

```
void dosFsModeChange
(
DOS_VOL_DESC * vdptr,   /* 指向卷描述符的指针 */
int newMode             /* O_RDONLY/O_WRONLY/O_RDWR */
)
```

功能描述：

该函数将磁盘卷的模式设置为 newMode 模式。实际的模式值保存在 BLK_DEV 结构中的 bd_mode 字段，所以也会用于设备驱动程序中。直接改变该字段值与调用该函数效果相同。当决定了读写能力之后，模式字段应该相应更新，通常在就绪状态改变之后。参见函数 dosFsReadyChange()的说明。

驱动程序的设备初始化函数应该设置模式字段为O_RDWR（同时支持O_RDONLY 和O_WRONLY）。

返回值：

无。

参考：

相关信息请参考 dosFsLib 库以及函数 dosFsReadyChange()的描述。

dosFsReadyChange()

函数原型：

```
void dosFsReadyChange
(
DOS_VOL_DESC * vdptr    /* 指向卷描述符的指针 */
)
```

功能描述：

该函数设置卷描述符的状态到 DOS_VD_READY_CHANGED。当驱动觉察到设备上线或者离线时都应该调用该函数（例如插上或弹出磁盘）。

该函数调用后，下次尝试对该卷访问时将会重新装载该卷。

该函数与ioctl()函数中的FIODISKCHANGE命令执行相同的功能。

将BLK_DEV结构中的字段bd_readyChanged设置为TRUE会产生与调用该函数相同的结果。

返回值：

无。

参考：

相关信息请参考 dosFsLib 库的描述。

dosFsTimeSet()

函数原型：

```
STATUS dosFsTimeSet
(
int hour,     /* 0 to 23 */
```

```
int minute, /* 0 to 59 */
int second  /* 0 to 59 */
)
```

功能描述：

该函数为dosFs文件系统卷设置时间。所有文件的创建或者修改都会标着上这个时间。

注意：

时间不会自动增长，新的日期必须通过该函数自行设置。

返回值：

成功设置则返回OK，如果时间无效则返回ERROR。

参考：

相关信息请参考 dosFsLib 库以及函数 dosFsDateSet()、dosFsDateTimeInstall()的描述。

dosFsVolOptionsGet()

函数原型：

```
STATUS dosFsVolOptionsGet
(
DOS_VOL_DESC * vdptr,   /* 指向卷描述符的指针 */
UINT * pOptions         /* 存放当前选项值 */
)
```

功能描述：

该函数获取指定 dosFs 卷的当前选项，并将其存储在指针 pOptions 指向的字段中。

每个 FAT 复制数量的扇区数设置为可以包含 FAT 表的最小数量的扇区。

返回值：

总是返回 OK。

参考：

相关信息请参考 dosFsLib 库以及函数 dosFsVolOptionsSet()的描述。

dosFsVolOptionsSet()

函数原型：

```
STATUS dosFsVolOptionsSet
(
DOS_VOL_DESC * vdptr,   /* 指向卷描述符的指针 */
UINT options            /* 新选项值 */
)
```

功能描述：

该函数为一个已经初始化过的dosFs设备设置磁盘卷选项。只有下面的选项可以动态地改变。

DOS_OPT_CHANGENOWARN (0x1)

DOS_OPT_AUTOSYNC (0x2)

选项DOS_OPT_CHANGENOWARN仅对可移动卷有效（即，BLK_DEV结构中的bd_removable必须置为TRUE）。如果指定了不可移动卷，将忽略该选项。当成功设置之后，选项DOS_OPT_CHANGENOWARN使能DOS_OPT_AUTOSYNC选项。

推荐使用函数dosFsVolOptionsGet()获取当前卷选项，修改对应的位，然后使用dosFsVolOptionsSet()设置。

返回值：

OK，如果选项无效或者选项不可动态改变则返回ERROR。

参考：

相关信息请参考 dosFsLib 库以及函数 dosFsDevInitOptionsSet(), dosFsMkfsOptionsSet(),dosFsVolOptionsGet()的描述。

dosFsVolUnmount()

函数原型：

```
STATUS dosFsVolUnmount
(
DOS_VOL_DESC * vdptr /* 指向卷描述符的指针 */
)
```

功能描述：

当用户不再对此磁盘卷进行I/O操作是可以调用该函数将磁盘卸载，来更换可以动磁盘。

执行该操作后，所有该磁盘卷上写缓存中的数据将会被写到设备中，任何打开的文件描述符都被标记为无效同时磁盘卷被标记为未装载。当一系列的I/O操作初始化该磁盘后，磁盘卷将会重新自动装载。

一旦文件描述符被标记为无效，任何对该文件的操作将会返回错误（但是可以用close()函数释放无效的文件描述符，该函数调用会释放掉相应文件描述符并返回错误）。打开整个卷的文件描述符不会只为无效。

该函数所执行的功能与ioctl()的FIOUNMOUNT中执行的功能相同。

该函数不能在中断中调用。

返回值：

OK，如果磁盘卷未加载则返回ERROR。

参考：

相关信息请参考 dosFsLib 库以及函数 dosFsReadyChange()的描述。

7.2　原始文件系统函数

7.2.1　函数库描述

1．库命名

原始文件系统函数库名称为 rawFsLib。

2．函数

表 7-2 中列出了原始文件系统函数。

3．描述

该函数库为磁盘设备提供了不使用标准文件和目录结构的文件系统。整个磁盘卷被假设为单一的文件，每一次读写操作都面向整个磁盘的一部分。该文件系统不支持多个文件或者目录结构。

表 7-2　原始文件系统函数

函　数	描　述
rawFsDevInit()	为块设备连接原始文件系统
rawFsInit()	初始化原始文件系统
rawFsModeChange()	修改原始文件系统设备卷模式
rawFsReadyChange()	将就绪状态的改变告知文件系统
rawFsVolUnmount()	禁用原始文件系统的设备卷

4．使用该函数库

VxWorks 原始文件系统(rawFs)所提供的函数可以分成三组：通用初始化、设备初始化、文件系统操作。

函数 rawFsInit()负责初始化功能；不论初始化多少 rawFs 设备，该函数只能调用一次。

另外还有一个独立的参数用于设备的初始化。对于每一个 rawFs 设备，必须调用 rawFsDevInit()安装设备。

该库还提供了用于将系统环境的改变通知文件系统的函数。rawFsModeChange()函数用于修改涉及设备的读写访问方式。rawFsReadyChange()函数用于通知文件系统磁盘的切换。rawFsVolUnmount()用于通知文件

系统实际设备需要同步并卸载，通常为磁盘的更换做准备。

5．初始化函数库

在使用该库中的其他函数之前，应首先调用 rawFsInit()函数初始化函数库。该调用可以指定原始文件系统设备所能开启的最多文件数目。如果使用 open()或者 creat()函数试图开启更多的文件将返回错误。

rawFsInit()函数会将原始文件系统以驱动程序的形式安装在 VxWorks 的 I/O 系统驱动程序表中。所获取的驱动号将会保存在全局变量 rawFsDrvNum 中。

如果配置了 INCLUDE_RAWFS 宏，VxWorks 将会通过 usrRoot()任务调用 rawFsInit()函数执行初始化过程。

6．定义原始卷设备

设备描述符结构用于将文件系统用于实际的设备。该设备描述符的第一个成员必须是块设备描述符（BLK_DEV）。在使用 rawFsDevInit()函数之前就要初始化该结构。在 BLK_DEV 结构中，包含了驱动程序必须提供的五个函数的地址：读扇区、写扇区、执行设备 I/O 控制、检查设备状态、重置设备。BLK_DEV 结构中同时包含了描述设备物理配置的信息。

rawFsDevInit()函数将设备与 rawFsLib 函数库相联系。volName 参数用于指定设备名字符串，该字符串用于标识设备。该名称会增加到 I/O 系统设备表中，可以使用 iosDevShow()函数查看。

rawFsDevInit()函数的 pBlkDev 参数是指向用于描述设备的 BLK_DEV 结构的指针。函数 rawFsDevInit()语法如下：

```
rawFsDevInit
(
char *volName,          /* 卷名 */
BLK_DEV *pBlkDev  /* 指向设备描述符的指针 */
)
```

与 VxWorks 的 DOS 和 RT-11 文件系统不同，原始磁盘卷不需要 ioctl()的 FIODISKINIT 功能来初始化卷结构（虽然可以这样使用但是不产生任何效果）。所以原始磁盘卷中没有创建文件系统的函数。

当 rawFsLib 从系统中接收到一个 I/O 请求时，它会调用相应的驱动程序来访问设备。

7．支持多逻辑设备

传递给驱动读写函数的扇区号是绝对扇区号,从设备的开始零扇区编号。如果有必要，驱动可以将逻辑地址的偏移量加到物理设备的起始地址上。可以通过传送偏移量参数到驱动设备结构中去，并将文件系统传递的扇区号加上偏移量来实现。

8．卸载磁盘卷

磁盘在移除前首先应该卸载。磁盘卸载后，任何向磁盘中写入的数据都将会被丢弃。用户可以使用 rawFsVolUnmount()函数或者 ioctl()函数的 FIODISKCHANGE 功能来完成磁盘的卸载。

当磁盘卸载时，可能会存在已经打开的文件，这时候，那些打开的文件描述符将被标记为无效。所有试图访问描述符的操作将会返回 S_rawFsLib_FD_OBSOLETE 错误。用户可以使用 close()函数释放这些文件描述符，但是该操作仍然会返回 S_rawFsLib_FD_OBSOLETE 错误。

9．同步操作

磁盘在卸载之前应该执行同步操作。同步操作即是将所有的输出缓冲区中的数据写到磁盘上（打开的文件描述符所对应的写缓冲区）。如果使用 rawFsVolUnmount()函数卸载磁盘，则不需要执行同步操作。

调用 rawFsVolUnmount()函数卸载磁盘卷之前，该函数会试图执行同步操作。然而，在磁盘已经移除之后再执行 dosFsVolUnmount()函数则为时晚矣（这种情况 rawFsVolUnmount()函数会丢弃所有缓存的数据）。所以在磁盘移除前应调用 ioctl()的 FIOSYNC 功能。

如果在调用 rawFsVolUnmount()函数的时候磁盘仍然可以读写,则会成功执行同步操作,这时就不需要单独执行同步操作。其他的情况下，如果卸载前执行同步操作失败则会导致数据的丢失。

10．IOCTL 命令字

rawFs 文件系统支持下面的 ioctl()功能。这些功能均定义在 ioLib.h 头文件中。

FIODISKFORMAT

使用硬件的磁道扇区参数格式化磁盘。格式化之后不创建任何文件系统。注意这是驱动程序提供的功能。

```
fd = open ("DEV1:", O_WRONLY);
status = ioctl (fd, FIODISKFORMAT, 0);
```

FIODISKINIT

在指定的磁盘卷中初始化原始文件系统。由于没有文件结构，该函数执行空操作。仅仅为了兼容其他文件系统而提供。

FIODISKCHANGE

声明媒体更换，与函数 rawFsReadyChange()执行相同的功能。该函数可以在中断级别调用。

```
status = ioctl (fd, FIODISKCHANGE, 0);
```

FIOUNMOUNT

卸载磁盘卷，与函数 rawFsVolUnmount()执行相同的功能。不能在中断级别调用。

```
status = ioctl (fd, FIOUNMOUNT, 0);
```

FIOGETNAME

获取指定文件描述符的文件名。

```
status = ioctl (fd, FIOGETNAME, &nameBuf);
```

FIOSEEK

设置当前文件指针到新的位置。

```
status = ioctl (fd, FIOSEEK, newOffset);
```

FIOWHERE

返回当前指定文件描述符的文件指针相对于文件开始的字节位置。不需要其他参数。

```
position = ioctl (fd, FIOWHERE, 0);
```

FIOFLUSH

将所有修改过的文件描述符的缓冲写入到物理设备中。

```
status = ioctl (fd, FIOFLUSH, 0);
```

FIOSYNC

执行与 FIOFLUSH 相同的功能。

FIONREAD

计算当前文件指针到文件结束的字节数。

```
status = ioctl (fd, FIONREAD, &unreadCount);
```

11．头文件

原始文件系统函数声明在 rawFsLib.h 头文件中。

12．参考

相关信息请参考 rawFsLib、ioLib、iosLib、dosFsLib、rt11FsLib、ramDrv 库描述。

7.2.2　原始文件系统函数详细描述

rawFsDevInit()

函数原型：

```
RAW_VOL_DESC *rawFsDevInit
(
char * volName,          /* 卷名 */
BLK_DEV * pBlkDev  /* 指向块设备信息的指针 */
)
```

功能描述：

该函数将驱动程序创建的块设备定义为原始文件系统卷。在随后对该设备的高级 I/O 访问中，系统将会调用 rawLib 中的函数。

该函数将 volName 与设备相连并将其安装到 VxWorks 的 I/O 系统设备表中。这个设备所使用的驱动号就是由 rawFsInit()函数初始化得到的（相应的驱动号保存在全局变量 rawFsDrvNum 中）。

参数 pBlkDev 所指定的 BLK_DEV 结构用于描述指定的块设备，同时该结构会指定访问设备所使用的 5 个函数。这些函数只有在执行一系列 I/O 操作时才会使用到。

返回值：

指向卷描述符的指针（RAW_VOL_DESC），如果出现错误则返回 NULL。

参考：

相关信息请参考 rawFsLib 库描述。

例：

```
/* 配置设备 busId = 4, LUN = 0 */
...
if ((pSpd40 = scsiPhysDevCreate (pSysScsiCtrl, 4, 0, 0, NONE, 0, 0, 0))
    == (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n");
    }
    else
    {
    /* 创建块设备：一个给 dosFs 系统，另一个给 rawFs 系统 */
    if (((pSbd0 = scsiBlkDevCreate (pSpd40, 0x20000, 0)) == NULL) ||
    ((pSbd1 = scsiBlkDevCreate (pSpd40, 0, 0x20000)) == NULL))
    {
    return (ERROR);
    }
    ...
    /* 初始化 dosFs 和 rawFs 文件系统 */
    if ((dosFsDevInit ("/sd0/", pSbd0, NULL) == NULL) ||
    (rawFsDevInit("/sd1/", pSbd1) == NULL))
    {
    return (ERROR);
    }
    }
    ...
```

rawFsInit()

函数原型：

```
STATUS rawFsInit
(
int maxFiles /* 同时打开的最大文件数 */
)
```

功能描述：

该函数初始化原始卷函数库，在使用库中其他函数之前必须而且只能调用该函数一次。其参数指定了同时可使用在设备上的最大文件描述符的个数。该函数会为其他函数分配和设置必要的内存结构并初始化信号量。

该函数还会将原始卷函数中的函数安装在 VxWorks 的 I/O 系统驱动程序表中，所返回的驱动号将会保存在全局变量 rawFsDrvNum 中。该驱动号可以被后续的操作所使用。

如果在工程中包含了 INCLUDE_RAWFS 定义，则系统会自动在根任务 usrRoot()中执行该初始化。

返回值：

成功初始化则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 rawFsLib 库描述。

rawFsModeChange()

函数原型：

```
void rawFsModeChange
(
RAW_VOL_DESC * vdptr,  /* 指向卷描述符的指针 */
int newMode                    /* O_RDONLY/O_WRONLY/O_RDWR */
)
```

功能描述：

该函数通过修改 BLK_DEV 结构将设备模式设置为 newMode。该函数应在读写操作之前调用，通常是在就绪状态改变之后。

驱动程序的设备初始化函数应将初始模式设置为 O_RDWR（也就是同时包括 O_RDONLY 和 O_WRONLY）。

返回值：

无。

参考：

相关信息请参考 rawFsLib 库描述、rawFsReadyChange()函数。

rawFsReadyChange()

函数原型：

```
void rawFsReadyChange
(
RAW_VOL_DESC * vdptr /* 指向卷描述符的指针 */
)
```

功能描述：

该函数将卷描述符状态设置为 RAW_VD_READY_CHANGED。当驱动程序感知到设备装载或者卸载时应该调用该函数。

该函数被调用后，再次访问磁盘卷将会导致重新安装设备。

返回值：

无。

参考：

相关信息请参考 rawFsLib 库描述。

rawFsVolUnmount()

函数原型：

```
STATUS rawFsVolUnmount
```

```
(
RAW_VOL_DESC * vdptr /* 指向卷描述符的指针 */
)
```

功能描述：

当不再对磁盘卷进行 I/O 操作时可以调用该函数。通常是在更换磁盘前调用该函数。调用该函数会将所有缓冲的数据写入到设备上，所有打开的文件都会作废，同时磁盘卷被标记为卸载状态。

由于该函数会将所有内存中的数据写到物理内存，所以在不可确认磁盘是否卸载的情况下不能使用该函数。这时，应该使用状态改变机制（参见本函数库描述）。

该函数在使用 ioctl()函数的功能 FIOUNMOUNT 时也会被调用。

返回值：

成功操作则返回 OK，如果不能访问磁盘卷则返回 ERROR。

参考：

相关信息请参考 rawFsLib 库描述以及 rawFsReadyChange()函数。

7.3 磁带设备文件系统函数

7.3.1 函数库描述

1．库命名

磁带序列设备文件系统库名称为 tapeFsLib。

2．函数

表 7-3 中列出了磁带序列设备文件系统函数。

表 7-3 磁带序列设备文件系统函数

函 数	描 述
tapeFsDevInit()	将磁带卷功能连接到磁带设备
tapeFsInit()	初始化磁带卷库
tapeFsReadyChange()	通知 tapeFsLib 库转换到未就绪状态
tapeFsVolUnmount()	禁用磁带设备卷

3．描述

该函数库提供了对不使用标准文件、目录结构的磁带设备的基本支持。磁带卷会被系统作为一个大文件处理，可以支持读或者写。但是该库中并不对磁带卷提供文件或者目录的支持，这需要在更高一级的接口中实现。

4．使用磁带序列文件系统

由 VxWorks 的磁带文件系统提供的不同的函数可以分为 3 类：一般的初始化、设备初始化、文件系统操作。

函数 tapeFsInit()是一般的初始化函数。不论使用多少磁带卷设备，都只能调用该函数一次。

每初始化一个 tapeFs 设备都需要运行一次 tapeFsDevInit()函数。

通常都是通过 I/O 接口来使用该函数库的功能：open()、close()、read()、write()、ioctl()。除了这些接口，系统还提供了几个接口用于将系统环境变量的改变通知文件系统。函数 tapeFsVolUnmount()通知文件系统指定的设备应该被卸载了。在使用该函数之前应先做好同步，使用该函数之后则可以准备更换设备了。

函数 tapeFsReadyChange()通知文件系统设备已经更换，下一次磁带操作是面对新的设备的操作。关于 ready-change 的信息也与 SEQ_DEV 结构有关。注意只有在文件关闭之后才可以调用 tapeFsVolUnmount()和 tapeFsReadyChange()。

5．初始化文件系统

在使用 tapeFsLib 函数库中的任何函数之前必须首先调用函数 tapeFsInit()初始化该函数库。这种执行方式假设每卷只有一个文件描述符。但这种约束将在未来中会得到改进。

在初始化的过程中磁带设备库作为驱动程序安装在 I/O 系统驱动程序表中。相应的驱动号则被放置在全

局变量 tapeFsDrvNum 中。

在 BSP 中定义了 INCLUDE_TAPES 宏或者在使用该函数库之前调用 tapeFsDevInit()和 tapeFsInit()将自动执行上述过程。

6．定义磁带设备

设备描述符结构用于将文件系统用于实际的设备。该设备描述符的第 1 个成员必须是序列设备描述符（SEQ_DEV）。在使用 tapeFsDevInit()函数之前就要初始化该结构。在 SEQ_DEV 结构中，包含了驱动程序必须提供的函数的地址：读扇区、写扇区、执行设备 I/O 控制、向磁带设备中写入文件标记、倒回磁带卷、保存磁带设备、释放磁带设备、加载/卸载磁带卷、快进/快退、擦除、检查设备状态、重置设备。SEQ_DEV 结构中同时包含了描述设备物理配置的信息。

7．初始化磁带序列设备

在调用 tapeFs 库中的其他函数之前，应调用 tapeFsInit()函数。参数 volName 指定所创建的设备的名称，同时这个名称将作为文件访问路径的一部分被添加到 I/O 系统设备表中，可以使用 iosDevShow()函数显示。

参数 pSeqDev 是指向设备描述结构 SEQ_DEV 的指针，其中包含了必须的驱动函数。

参数 pTapeConfig 是指向 TAPE_CONFIG 结构的指针，该结构中包含了磁带设备的配置信息。其中的配置项包括块的大小是否可变、是否将设备倒回、文件标记数量的限制等。关于 TAPE_CONFIG 结构的更详细内容可以参考 tapeFsLib.h 文件。

tapeFsDevInit()函数的语法如下：

```
tapeFsDevInit
(
char * volName,                 /* 卷名 */
SEQ_DEV * pSeqDev,              /* 指向设备描述符的指针 */
TAPE_CONFIG * pTapeConfig       /* 指向磁带配置信息的指针 */
)
```

tapeFsDevInit()函数完成之后，tapeFsLib 在接到 I/O 系统的请求时将调用驱动程序的函数访问设备。

8．打开及关闭文件

使用 I/O 系统函数 open()可以打开磁带卷设备。可以打开的文件模式只能是 O_RDONLY 或者 O_WRONLY。该库不支持 O_RDWR。open()函数会初始化文件描述符缓冲区以及状态信息，同时会独占磁带设备，根据配置信息倒回磁带，装载磁带卷。一旦磁带卷被打开，再次关闭前就不允许其他的系统访问设备。同样，在文件关闭前对应的文件描述符被标记为“使用中”，以确认其不会被多次使用。

使用 I/O 调用 close()函数可以关闭磁带设备。发出 close()请求之后，所有未写入的缓冲区都回写入到设备中，然后设备根据配置信息决定是否倒回，最后系统将设备释放。

9．卸载磁带卷设备

磁带卷在移除之前应将其卸载。当卸载磁带卷之前应确保其关联的文件描述符已经被关闭。用户可以直接调用 tapeFsVolUnmount()函数卸载磁带卷。

如果有文件打开，就不应该更换磁带。由于 tapeFs 假设每个设备仅关联一个文件描述符，如果需要重新使用该设备必须先关闭文件描述符，然后再将其打开。

在调用 tapeFsVolUnmount()函数卸载磁带卷之前，应该执行同步操作。调用 ioctl()函数的 FIOSYNC 或者 FIOFLUSH 功能可以在磁带卷卸载前执行磁带文件系统的同步。卸载前同步失败将导致数据丢失。

10．IOCTL 命令字

VxWorks 磁带序列设备文件系统支持下面的 ioctl()功能。这些功能均定义在 tapeFsLib.h 中。

FIOFLUSH

将所有修改过的文件描述符的缓冲区写入到物理设备中。

```
status = ioctl (fd, FIOFLUSH, 0);
```

FIOSYNC

执行与 FIOFLUSH 相同的功能。

FIOBLKSIZEGET

获取物理设备的块尺寸值。

FIOBLKSIZESET

在物理设备上设置指定的块大小并更新 SEQ_DEV 结构和 TAPE_VOL_DESC 结构。如果函数所提供的块值为零，函数仅仅更新结构中的值而不改变实际物理设备的块尺寸。因为结构中的“0”代表着可变块值，所以设备的块值将会被忽略。

MTIOCTOP

允许使用标准的 UNIX MTIO ioctl 操作，其中所使用的 MTOP 结构定义如下：

```
typedef struct mtop
{
short mt_op;      /* 操作 */
int mt_count;     /* 操作号 */
} MTOP;
```

使用方法如下：

```
MTOP mtop;
mtop.mt_op = MTWEOF;
mtop.mt_count = 1;
status = ioctl (fd, MTIOCTOP, (int) &mtop);
```

mt_op 所允许的取值为：

MTWEOF：将文件结束符写到磁带中。文件结束符是一种文件标记。

MTFSF：向前越过文件标记将磁头放置在文件标记和下一块数据之间。如果此时磁带在写模式下，则将缓存的数据写回到磁带中。

MTBSF：向后越过文件标记将磁头放置在文件标记之前。如果此时磁带在写模式下，则将缓存的数据写回到磁带中。

MTFSR：向前越过一块数据将磁头放置在越过的数据和下一块数据之间。如果此时磁带在写模式下，则将缓存的数据写回到磁带中。

MTBSR：向后越过一块数据将磁头放置在越过的数据之前。如果此时磁带在写模式下，则将缓存的数据写回到磁带中。

MTREW：将磁带倒回到开始处。如果此时磁带在写模式下，则将缓存的数据写回到磁带中。

MTOFFL：倒回并卸载磁带。如果此时磁带在写模式下，则将缓存的数据写回到磁带中。

MTNOP：不作任何操作，但是检查设备状态，并设置 SEQ_DEV 结构中相应的字段。

MTRETEN：保持磁带。该命令通常设置磁带状态并使之处在读或写的状态。如果此时磁带在写模式下，则将缓存的数据写回到磁带中。

MTERASE：擦除全部磁带并将其倒回。

MTEOM：将磁带置为磁带的末端并卸载磁带。如果此时磁带在写模式下，则将缓存的数据写回到磁带中。

11．头文件

磁带文件系统函数声明在 tapeFsLib.h 头文件中。

12．参考

相关信息请参考 tapeFsLib、ioLib、iosLib 库描述。

7.3.2 磁带文件系统函数详细描述

tapeFsDevInit()

函数原型：

```
TAPE_VOL_DESC *tapeFsDevInit
(
char * volName,                 /* 卷名 */
SEQ_DEV * pSeqDev,              /* 指向连续设备的信息 */
TAPE_CONFIG * pTapeConfig       /* 指向磁带配置信息 */
)
```

功能描述：

该函数将设备驱动创建的连续设备定义为磁带文件系统卷。该函数执行之后，对该设备的高级 I/O 访问(例如 open()、write()操作)将会通过 tapeFsLib 库实现。

该函数将 volName 与设备相连并将其安装在 VxWorks 的 I/O 系统设备表中。所得到的驱动号保存在全局变量 tapeFsDrvNum 中。该驱动号需要在 tapeFsInit()函数中使用。

由 pSeqDev 指针指向的 SEQ_DEV 结构包含了描述设备的配置信息以及驱动程序的地址，包括：读扇区、写扇区、执行设备 I/O 控制、向磁带设备中写入文件标记、倒回磁带卷、保存磁带设备、释放磁带设备、加载/卸载磁带卷、快进/快退、擦除、检查设备状态、重置设备。TAPE_CONFIG 结构用于定义 TAPE_VOL_DESC 的配置参数。相关配置参数定义在 tapeFsLib.h 头文件中。

返回值：

返回指向 TAPE_VOL_DESC 结构的指针，如果发生错误则返回 NULL。

错误号：

S_tapeFsLib_NO_SEQ_DEV、S_tapeFsLib_ILLEGAL_TAPE_CONFIG_PARM。

参考：

相关信息请参考 tapeFsLib 库描述。

例：

```
STATUS sysScsiConfig (void)
    {
...
/* 创建 SCSI 物理设备和连续设备 */
    if ((pSpd40 = scsiPhysDevCreate (pSysScsiCtrl, scsiId, 0,0,NONE,0,0,0))
            == NULL)
        {
        printErr ("scsiPhysDevCreate failed.\n");
        return (ERROR);
        }
    if ((pSd0 = scsiSeqDevCreate (pSpd40)) == NULL)
        {
        printErr ("scsiSeqDevCreate failed.\n");
        return (ERROR);
        }
    /* 在固定的块上配置可回放的磁带文件系统 */
    pTapeConfig = (TAPE_CONFIG *) calloc (sizeof(TAPE_CONFIG),1);
    pTapeConfig->rewind = TRUE;     /* 回放设备 */
    pTapeConfig->blkSize = 512;     /* 固定的 512 字节块 */
    if (tapeFsDevInit ("/tape1", pSd0, pTapeConfig) == NULL)
{
```

```
    printErr ("tapeFsDevInit failed.\n");
    return (ERROR);
    }
    ...
    }
```

tapeFsInit()

函数原型：

```
STATUS tapeFsInit()
```

功能描述：

该函数初始化磁带卷库。该函数只能调用一次，而且必须在使用该库中的其他函数之前调用。一个磁带卷上只能打开一个文件描述符。

该函数同时也将磁带卷库函数添加到 VxWorks 的 I/O 系统驱动程序表中。对应的驱动号保存在全局变量 tapeFsDrvNum 中。该驱动号将会和 tapeFs 设备上打开的系统文件描述符相连。

使能该初始化只需要调用 tapeFsDevInit()函数即可实现，该函数会自动调用 tapeFsInit()初始化磁带文件系统。

返回值：

初始化成功则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 tapeFsLib 库描述。

tapeFsReadyChange()

函数原型：

```
STATUS tapeFsReadyChange
(
TAPE_VOL_DESC * pTapeVol /* 指向卷描述符的指针 */
)
```

功能描述：

该函数将卷描述符的状态设置为 TAPE_VD_READY_CHANGED。当驱动程序感知到设备连接或设备断开时应调用该函数（例如，插入或者移除磁带时应该使用）。

该函数调用之后，下一次对该卷的访问将会使设备重新安装。

返回值：

如果设置了就绪状态改变则返回 OK，如果文件描述符正在使用中则返回 ERROR。

错误值：

S_tapeFsLib_FILE_DESCRIPTOR_BUSY

参考：

相关信息请参考 tapeFsLib 库描述。

tapeFsVolUnmount()

函数原型：

```
STATUS tapeFsVolUnmount
(
TAPE_VOL_DESC * pTapeVol /* 指向卷描述符的指针 */
)
```

功能描述：

用户不再访问磁带卷的时候应该调用该函数。通常在更换磁带的时候调用该函数。所有对应卷缓存的数据将会写回到设备中去。所有打开的文件描述符都将被标记为无效，并且磁带卷被标记为未装载。

由于该过程将内存中的数据写到物理设备中，所以该函数不能用于磁盘已经更换的情况。在这种情况下，应该使用"就绪状态改变"机制。

该函数与 ioctl()函数中的 FIOUNMOUNT 功能的执行效果相同。

返回值：

操作成功则返回 OK，如果不能访问磁盘卷将会返回 ERROR。

错误值：

S_tapeFsLib_VOLUME_NOT_AVAILABLE、S_tapeFsLib_FILE_DESCRIPTOR_BUSY、S_tapeFsLib_SERVICE_NOT_AVAILABLE。

参考：

相关信息请参考 tapeFsLib 库描述以及 tapeFsReadyChange()函数。

7.4　CD-ROM 文件系统函数

7.4.1　函数库描述

1．库命名

CD-ROM 文件系统函数库名称为 cdromFsLib。

2．函数

表 7-4 中列出了 CD-ROM 文件系统函数。

表 7-4　CD-ROM 文件系统函数

函　数	描　述
cdromFsInit()	初始化 cdromFsLib
cdromFsVolConfigShow()	显示卷配置信息
cdromFsDevCreate()	创建 cdromFsLib 设备

3．描述

该函数库可以允许用户使用标准的 POSIX I/O 调用从符合 ISO 9660 标准的 CD-ROM 格式文件系统中读取数据。

VxWorks 系统提供标准的 BLOCK_DEV 结构访问 CD-ROM 文件系统。

基本的初始化顺序与在 SCSI 设备上安装 DOS 文件系统相类似。

1）初始化 cdrom 文件系统库(更适合在 sysScsi.c 的 sysScsiConfig()函数中执行)：

```
cdromFsInit();
```

2）查找并创建一个 SCSI 物理设备：

```
pPhysDev=scsiPhysDevCreate(pSysScsiCtrl,0,0,0,NONE,1,0,0);
```

3）在物理设备上创建一个 SCSI 块设备：

```
pBlkDev = (SCSI_BLK_DEV *) scsiBlkDevCreate (pPhysDev, 0, 0);
```

4）在块设备上创建 CD-ROM 文件系统：

```
cdVolDesc = cdromFsDevCreate ("cdrom:", (BLK_DEV *) pBlkDev);
```

每连接一个目标 CD-ROM 驱动都需要调用 cdromFsDevCreate()函数。该函数成功返回之后，CD-ROM 文件系统将可以像 DOS 文件系统一样使用，用户可以通过调用 open()、close()、read()、ioctl()、readdir()和 stat()等函数访问 CD-ROM 上的数据。然而调用 write()函数将总会返回错误。

cdromFsLib 函数库支持多个驱动程序，支持多任务的并发访问，支持打开多个文件。

4．文件和目录的命名

严谨的 ISO 9660 标准详细说明了仅允许使用大写的 8.3 格式的文件名。为了支持多版本的相同文件，该标准支持文件的版本号。在 open()函数中指定文件名的时候，可以为文件选择一个十进制数指定文件的版本。如果用户忽略了版本号，cdromFsLib 会为打开的文件使用最近一次使用过的版本号。

为了兼容 MS-DOS，cdromFsLib 允许用户使用小写、大写混用的文件名访问文件。但是混用的文件名只

能打开完全匹配的文件名。

最后，cdromFsLib 使用 ISO 9660 的 8 位表达方式，也就是说，cdromFsLib 允许用户使用拉丁或者亚洲字符作为文件名。

5．IOCTL 命令字

FIOGETNAME

返回指定文件描述符的文件名。

FIOLABELGET

返回卷标，该功能可以用来返回光盘的详细卷说明。

FIOWHERE

确定当前文件位置。

FIOSEEK

改变当前文件位置。

FIONREAD

获取当前文件指针位置与文件结尾之间的字节数。

FIOREADDIR

读取下一个目录条目。

FIODISKCHANGE

声明光盘更换（在块设备驱动程序无法提供该声明时使用）。

FIOUNMOUNT

声明磁盘移除（当前打开的文件描述符均无效）。

FIOFSTATGET

获取文件状态信息。

6．修改 BSP 以便使用 cdrom 文件系统

下面的例子描述了在 SCSI 设备上装载 cdromFs 文件系统。

根据下面的步骤编辑 BSP 的 config.h 文件：

（1）加入下面的宏定义：

```
#define INCLUDE_CDROMFS
```

（2）把下面的注释部分中的 FALSE 改成 TRUE。

```
/* change FALSE to TRUE for SCSI interface */
```

在文件 sysScsi.c 中作下列修改(如果 BSP 中没有 sysScsi.c 则在 sysLib.c 文件中修改)：

1）在文件的顶端加入下面的声明：

```
#ifdef INCLUDE_CDROMFS
#include "cdromFsLib.h"
STATUS cdromFsInit (void);
#endif
```

2）修改 sysScsiInit()的定义，加入下面的部分：

```
#ifdef INCLUDE_CDROMFS
cdromFsInit();
#endif
```

cdromFsInit()函数初始化 cdromFS。该函数只能在调用其他 cdromFsLib 的函数之前调用一次，并且必须完全正确返回。通常是在系统引导时调用。由于使用 SCSI CD-ROM 设备，所以这里很自然在 sysScsiInit()函数中调用 cdromFSInit()函数。

（3）修改 sysScsiConfig()函数的定义，包含下面的内容：

```
/* 配置一个 SCSI CDROM，busId = 6, LUN = 0 */
```

```
#ifdef INCLUDE_CDROMFS
if ((pSpd60 = scsiPhysDevCreate (pSysScsiCtrl, 6, 0, 0, NONE,
0, 0, 0)) ==
(SCSI_PHYS_DEV *) NULL)
{
SCSI_DEBUG_MSG ("sysScsiConfig: scsiPhysDevCreate failed for CDROM.\n",
0, 0, 0, 0, 0, 0);
return (ERROR);
}
else if ((pSbdCd = scsiBlkDevCreate (pSpd60, 0, 0) ) == NULL)
{
SCSI_DEBUG_MSG ("sysScsiConfig: scsiBlkDevCreate failed for CDROM.\n",
0, 0, 0, 0, 0, 0);
return (ERROR);
}
/*
* 在 I/O 系统中创建一个 CD-ROM 设备。
* cdromFsDevCreate() 调用 iosDrvInstall(), 进入
* 在 I/O 驱动程序表中适当的驱动程序。
*/
if ((cdVolDesc = cdromFsDevCreate ("cdrom:", (BLK_DEV *)
pSbdCd )) == NULL)
{
return (ERROR);
}
#endif /* end of #ifdef INCLUDE_CDROMFS */
```

（4）在 sysScsiConfig()函数的定义之前声明下面的内容：

```
SCSI_PHYS_DEV *pSpd60;
BLK_DEV *pSbdCd;
CDROM_VOL_DESC_ID cdVolDesc;
```

上面这一段代码的主要目的就是调用 cdromFsDevCreate()函数。cdromFsDevCreate()函数需要指向块设备的指针作为参数。在上面的例子中，scsiPhysDevCreate()和 scsiBlkDevCreate()函数以 SCSI CD-ROM 设备创建块设备。

在完成函数 cdromFsDevCreate()的执行后，就可以使用标准 I/O 接口访问 cdrom 设备了。

7．头文件

cdrom 文件系统函数声明在 cdromFsLib.h 头文件中。

8．警告

cdromFsLib 并不支持 CD 装置包含多个磁盘。

9．参考

相关信息请参考 cdromFsLib、ioLib 库描述以及 ISO 9660 规范。

7.4.2　cdrom 文件系统函数详细描述

cdromFsInit()

函数原型：

STATUS cdromFsInit (void)

功能描述：

该函数初始化 cdromFsLib。在调用该库中其他函数之前必须要先调用该函数一次，而且只能调用一次。

错误号：

S_cdromFsLib_ALREADY_INIT

返回值：

初始化成功则返回 OK，如果已经初始化了 cdromFsLib 则返回 ERROR。

参考：

相关信息请参考 cdromFsLib 库描述以及 cdromFsDevCreate()函数，iosLib.h 头文件。

cdromFsVolConfigShow()

函数原型：

```
VOID cdromFsVolConfigShow
(
void * arg   /* 设备名或 CDROM_VOL_DESC * */
)
```

功能描述：

该函数获取设备的卷配置并将其打印到标准输入输出设备。获取的信息来自于指定设备的 BLK_DEV 结构。

返回值：

无。

参考：

相关信息请参考 cdromFsLib 库描述。

cdromFsDevCreate()

函数原型：

```
CDROM_VOL_DESC_ID cdromFsDevCreate
(
char * devName,          /* 设备名 */
BLK_DEV * pBlkDev   /* 指向块设备的指针 */
)
```

功能描述：

该函数在 I/O 系统中重新创建一个 cdromFsLib 设备。该函数需要一个指向块设备的 BLK_DEV 结构型指针作为参数。所以在调用该函数之前应调用 scsiBlkDevCreate()函数。

返回值：

创建成功则返回 CDROM_VOL_DESC_ID，如果出错则返回 NULL。

参考：

相关信息请参考 cdromFsLib 库描述、cdromFsInit()函数。

第 8 章　系统内核函数

8.1　系统相关函数

8.1.1　函数库描述

1．库命名

系统相关函数库名称为 sysLib。

2．函数

表 8-1 中列出了系统相关函数。

表 8-1　系统相关函数

函　数	描　述	函　数	描　述
sysClkConnect()	将指定函数连接到系统时钟中断上	sysModel()	返回 CPU 板的模块名称
sysClkDisable()	关闭系统时钟中断	sysBspRev()	返回 BSP 版本和修订号
sysClkEnable()	使能系统时钟中断	sysHwInit()	初始化系统硬件
sysClkRateGet()	获取系统时钟频率	sysPhysMemTop()	获取物理内存顶端地址
sysClkRateSet()	设置系统时钟频率	sysMemTop()	获取逻辑内存顶端地址
sysAuxClkConnect()	将指定函数连接到辅助时钟中断上	sysToMonitor()	将系统控制权传给 ROM 监视程序
sysAuxClkDisable()	关闭辅助时钟中断	sysProcNumGet()	获取处理器号
sysAuxClkEnable()	打开辅助时钟中断	sysProcNumSet()	设置处理器号
sysAuxClkRateGet()	获取辅助时钟频率	sysBusTas()	通过总线测试并设置本地单元
sysAuxClkRateSet()	设置辅助时钟频率	sysScsiBusReset()	驱动 SCSI 总线上的 RST 信号（仅对西部数据的 WD33C93 有效）
sysIntDisable()	禁用指定的总线中断级	sysScsiInit()	初始化并使能板上 SCSI 接口
sysIntEnable()	使能指定的总线中断级	sysScsiConfig()	系统 SCSI 配置
sysBusIntAck()	响应指定的总线中断	sysLocalToBusAdrs()	将指定的本地地址转换为总线地址
sysBusIntGen()	产生总线中断	sysBusToLocalAdrs()	将指定的总线地址转换为本地地址
sysMailboxConnect()	将指定函数连接到邮箱中断上	sysSerialHwInit()	初始化 BSP 串行设备为静止状态
sysMailboxEnable()	使能邮箱中断	sysSerialHwInit2()	连接 BSP 串行设备中断
sysNvRamGet()	获取非易失性 RAM 的内容	sysSerialReset()	重置所有 SIO 设备为静止状态
sysNvRamSet()	写非易失性 RAM	sysSerialChanGet()	获取 SIO_CHAN 设备所连接的串行通道

3．描述

该函数库提供了与目标板相关的函数。

注意：这只是一个通用的 BSP 函数库，这里只包含一般的描述。一些关于特殊的 BSP 描述还要参考特

定的 BSP 相关资料。

文件 sysLib.c 中提供涉及 VxWorks 的板级接口和应用代码可以实现与硬件无关的性能。该文件中包含的函数包括：

- 初始化函数：
  - 将硬件初始化到特定的状态；
  - 识别系统；
  - 初始化诸如 SCSI 或自定义设备的驱动。
- 内存/地址空间函数：
  - 获取板上内存大小；
  - 使板上内存可以被外部总线访问；
  - 映射本地和总线地址空间；
  - 使能/禁用高速缓存；
  - 设置/获取非易失性内存（NVRAM）；
  - 定义板上内存映射（可选）；
  - 如果处理器带有 MMU，则还包含虚拟内存到物理内存声明。
- 总线中断函数：
  - 使能/禁用总线中断级别；
  - 产生总线中断。
- 时钟/计数器函数：
  - 使能/禁用计时器中断；
  - 设置计时器周期
- 邮箱/本地监视函数：
  - 为基于 VME 的板卡使能邮箱/本地监视中断。

对于不同的板卡来说，该函数库支持的函数不同，有些板卡可能对该函数库有所扩展。相反地，有些板卡却不支持所有函数库中的函数；有些板卡为某些函数提供了特定的功能。

通常，该函数库里的大多数函数都不会被用户的应用程序所调用。配置模块 usrConfig.c 和 bootConfig.c 中适当地使用了这些函数。驱动程序也有可能会调用其中的内存映射和总线函数。

注意：

这里只是通用的 BSP 函数库，这里只包含一般的描述。BSP.*的说明将会有更详细的信息。

4．头文件

系统相关函数声明在 sysLib.h 头文件中。

5．参考

相关信息请参考“VxWorks Programmer's Guide”中的配置和构造章节。

8.1.2 系统相关函数详细描述

sysClkConnect()

函数原型：

```
STATUS sysClkConnect
(
FUNCPTR routine,      /* 系统时钟中断处理程序 */
int arg               /* 传递给处理程序的参数 */
)
```

功能描述：

该函数为时钟中断指定中断服务程序。通常由 usrConfig.c 中的 usrRoot()函数将 usrClock()函数连接在系统时钟中断上。

返回值：

操作成功则返回 OK，如果函数不能连接到中断上则会返回 ERROR。

参考：

相关信息请参考 sysLib 库描述、intConnect()、usrClock()、sysClkEnable()函数等。

例：

```
#define             SYS_CLK_RATE60
...
/* usrClock – 用户自定义的系统时钟 ISR */
void usrClock (void)
    {
    tickAnnounce();    /* 通知内核，时钟 tick 到。*/
    }
/*  sysClkInit – 初始化系统时钟 */
void sysClkInit (void)
    {
    /* 安装系统计时器 */
    sysClkConnect ((FUNCPTR) usrClock, 0);    /* 关联 ISR */
    sysClkRateSet (SYS_CLK_RATE);             /* 设置系统时钟频率 */
    sysClkEnable();                           /* 启动系统时钟 */
    }
```

sysClkDisable()

函数原型：

```
void sysClkDisable (void)
```

功能描述：

该函数禁止系统时钟中断

返回值：

无。

参考：

相关信息请参考 sysLib 库描述及 sysClkEnable()函数。

sysClkEnable()

函数原型：

```
void sysClkEnable (void)
```

功能描述：

该函数使能系统时钟中断

返回值：

无。

参考：

相关信息请参考 sysLib 库描述和函数 sysClkConnect(), sysClkDisable(), sysClkRateSet()的说明。

sysClkRateGet()

函数原型：

int sysClkRateGet (void)

功能描述：
该函数获取系统时钟频率
返回值：
系统时钟的每秒 tick 数量。
参考：
相关信息请参考 sysLib 库描述、sysClkEnable()、sysClkRateSet()函数。
例：

```
...
sysClkRateSet(60);
...
taskDelay (sysClkRateGet());      /* 延时 1 秒 */
...
```

sysClkRateSet()
函数原型：

```
STATUS sysClkRateSet
(
int ticksPerSecond /* 每秒钟时钟中断次数 */
)
```

功能描述：
该函数设置系统时钟的中断频率。由 usrConfig.c 中的 usrRoot()函数调用。
返回值：
成功设置则返回 OK，如果频率无效或者始终无法设置则返回 ERROR。
参考：
相关信息请参考 sysLib、clockLib 库描述，sysClkEnable()、sysClkRateGet()函数。
sysAuxClkConnect()
函数原型：

```
STATUS sysAuxClkConnect
(
FUNCPTR routine, /* 辅助时钟 ISR */
int arg                /* 传递给辅助时钟 ISR 的参数 */
)
```

功能描述：
该函数指定辅助时钟中断所关联的中断服务程序。但是该函数并不使能辅助时钟。
返回值：
操作成功则返回 OK，如果函数不能连接到中断则返回 ERROR。
参考：
相关信息请参考 sysLib 库描述、intConnect()、sysAuxClkEnable()函数。
例：

```
SEM_ID   semId;
int      AuxRate;
```

```
...
/* 辅助时钟 ISR */
void Interrupt(void)
{
 semGive(semId);
}
/* 创建信号量并启动辅助时钟 */
void StartInt(void)
{
 semId = semBCreate(0, SEM_EMPTY);
 /* 给信号量命名 */
 objNameSet(semId, "Interrupt");
 /* 初始化辅助时钟 */
 sysAuxClkRateSet(600);
      sysAuxClkConnect((FUNCPTR)Interrupt, 0);
 AuxRate = sysAuxClkRateGet();
 sysAuxClkEnable();
}
```

sysAuxClkDisable()

函数原型：

void sysAuxClkDisable (void)

功能描述：

该函数禁用辅助时钟中断

返回值：

无。

参考：

相关信息请参考 sysLib 库描述以及函数 sysAuxClkEnable()。

sysAuxClkEnable()

函数原型：

void sysAuxClkEnable (void)

功能描述：

该函数使能辅助时钟中断

返回值：

无。

参考：

相关信息请参考 sysLib 库描述 sysAuxClkConnect()、sysAuxClkDisable()、sysAuxClkRateSet()函数。

sysAuxClkRateGet()

函数原型：

int sysAuxClkRateGet (void)

功能描述：

该函数获取辅助时钟中断频率

返回值：
每秒钟辅助时钟的 tick 数量。
参考：
相关信息请参考 sysLib 库描述 sysAuxClkEnable(), sysAuxClkRateSet()函数。
sysAuxClkRateSet()
函数原型：
STATUS sysAuxClkRateSet
(
int ticksPerSecond /* 每秒钟辅助时钟中断次数 */
)

功能描述：
该函数设置辅助时钟中断频率。但它不使能辅助时钟中断。
返回值：
成功设置则返回 OK，如果时钟频率无效或者计数器不能设置则返回 ERROR。
参考：
相关信息请参考 sysLib 库描述、sysAuxClkEnable()、sysAuxClkRateGet()函数。
sysIntDisable()
函数原型：
STATUS sysIntDisable
(
int intLevel /* 屏蔽中断级 */
)

功能描述：
该函数关闭指定的总线中断级。
返回值：
操作成功则返回 OK，如果参数 intLevel 超出范围则返回 ERROR。
参考：
相关信息请参考 sysLib 库描述及 sysIntEnable()函数。
sysIntEnable()
函数原型：
STATUS sysIntEnable
(
int intLevel /* 中断级 (1-7) */
)

功能描述：
该函数使能指定的中断级。
返回值：
操作成功则返回 OK，如果参数 intLevel 超出范围则返回 ERROR。
参考：
相关信息请参考 sysLib 库描述及 sysIntDisable()函数。

sysBusIntAck()

函数原型：

```
int sysBusIntAck
(
int intLevel /* 响应总线中断级 */
)
```

功能描述：

该函数响应指定的 VME 总线中断级别。

返回值：

NULL

参考：

相关信息请参考 sysLib 库描述及 sysBusIntGen()函数。

sysBusIntGen()

函数原型：

```
STATUS sysBusIntGen
(
int intLevel,       /* 总线中断级 */
int vector          /* 中断向量 (0-255) */
)
```

功能描述：

该函数为带有指定中断向量的中断级产生总线中断。

返回值：

调用成功则返回 OK，如果 intLevel 超出范围或者板上无法产生总线中断则返回 ERROR。

参考：

相关信息请参考 sysLib 库描述及 sysBusIntAck()函数。

sysMailboxConnect()

函数原型：

```
STATUS sysMailboxConnect
(
FUNCPTR routine, /* mailbox 中断服务程序 */
int arg              /* 传递给 ISR 的参数 */
)
```

功能描述：

该函数为邮箱中断指定中断服务程序

返回值：

调用成功则返回 OK，如果函数无法连接到中断则返回 ERROR。

参考：

相关信息请参考 sysLib 库描述、intConnect()、sysMailboxEnable()函数。

sysMailboxEnable()

函数原型：

```
STATUS sysMailboxEnable
```

```
(
char * mailboxAdrs /* mailbox 的地址 */
)
```

功能描述：

该函数使能邮箱中断

返回值：

永远返回 OK

参考：

相关信息请参考 sysLib 库描述以及函数 sysMailboxConnect()。

sysNvRamGet()

函数原型：

```
STATUS sysNvRamGet
(
char * string,      /* 将非易失性内存的内容复制给这个字符串 */
int strLen,         /* 最大复制字节数 */
int offset          /* NVRAM 字节偏移 */
)
```

功能描述：

该函数将非易失性内存的内容复制到指定的字符串中。该字符串以一个 EOS 结尾。

返回值：

成功调用则返回 OK，如果访问范围超过非易失性内存的范围则返回 ERROR。

参考：

相关信息请参考 sysLib 库描述以及函数 sysNvRamSet()。

sysNvRamSet()

函数原型：

```
STATUS sysNvRamSet
(
char * string,      /* 把该字符串复制到 NVRAM 中 */
int strLen,         /* 最大复制字节数 */
int offset          /* NVRAM 的字节偏移 */
)
```

功能描述：

该函数将指定的字符串复制到非易失性内存中

返回值：

复制成功则返回 OK，如果访问范围超过了非易失性内存的范围则返回 ERROR。

参考：

相关信息请参考 sysLib 库描述以及函数 sysNvRamGet()。

sysModel()

函数原型：

```
char *sysModel (void)
```

功能描述：

该函数返回 CPU 板的模块名。

返回值：

指向包含板卡名的字符串指针。

参考：

相关信息请参考 sysLib 库描述。

sysBspRev()

函数原型：

char * sysBspRev (void)

功能描述：

该函数返回指向 BSP 版本和修订号的指针。例如，可能会返回 1.0/1。BSP 的修订版本号会连接在 BSP 的版本号后面一并返回。

返回值：

指向 BSP 版本和修订号的指针。

参考：

相关信息请参考 sysLib 库描述。

sysHwInit()

函数原型：

void sysHwInit (void)

功能描述：

该函数初始化系统的硬件。该函数由 usrConfig.c 中的 usrInit()函数调用。

注意：

用户不应该直接调用该函数。

返回值：

无。

参考：

相关信息请参考 sysLib 库描述。

sysPhysMemTop()

函数原型：

char * sysPhysMemTop (void)

功能描述：

获取物理内存顶端地址。通常，物理内存的总量由 LOCAL_MEM_SIZE 来定义。BSP 值在定义了宏 LOCAL_MEM_AUTOSIZE 后支持运行时计算内存大小。如果未定义这个宏，系统会认为 LOCAL_MEM_SIZE 所指的就是真实的物理内存大小。

注意：

可以调整 LOCAL_MEM_SIZE 以便保留内存给应用程序使用。更多关于保留内存的信息请参考函数 sysMemTop()的说明。

返回值：

物理内存的顶端地址。

参考：

相关信息请参考 sysLib 库描述以及函数 sysMemTop()。

sysMemTop()

函数原型：

char *sysMemTop (void)

功能描述：

该函数返回逻辑内存顶部地址。

返回值：

内存顶部的地址。

参考：

相关信息请参考 sysLib 库描述。

sysToMonitor()

函数原型：

```
STATUS sysToMonitor
(
int startType /* 启动类型 */
)
```

功能描述：

该函数将系统控制权传输给 ROM 监视程序。通常，该函数只由 reboot()函数以及中断级的总线错误调用。然而有些情况下，用户可能会使用 startType 参数指定引导类型。

返回值：

该函数不返回。

参考：

相关信息请参考 sysLib 库描述。

sysProcNumGet()

函数原型：

int sysProcNumGet (void)

功能描述：

该函数获取 CPU 板的处理器号，该处理器号由 sysProcNumSet()设置。

返回值：

CPU 板的处理器号。

参考：

相关信息请参考 sysLib 库描述以及函数 sysProcNumSet()。

sysProcNumSet()

函数原型：

```
void sysProcNumSet
(
int procNum /* 处理器号 */
)
```

功能描述：

该函数设置 CPU 板的处理器号，在同一块背板上处理器号应该是相同的。

返回值：

无。

参考：

相关信息请参考 sysLib 库描述以及函数 sysProcNumGet()。

sysBusTas()

函数原型：

```
BOOL sysBusTas
(
char * adrs /*  设置测试地址  */
)
```

功能描述：

该函数跨过背板执行测试和设置指令。

注意：

该函数等效于 vxTas()。

返回值：

如果该值从来没有设置过则返回 TRUE，如果已经设置则返回 FALSE。

参考：

相关信息请参考 sysLib 库描述以及函数 vxTas()。

sysScsiBusReset()

函数原型：

```
void sysScsiBusReset
(
WD_33C93_SCSI_CTRL * pSbic /*  指向  SBIC 信息的指针  */
)
```

功能描述：

该函数驱动 SCSI 总线的 RST 线，将所有连接的设备都置为静止状态。

返回值：

无。

参考：

相关信息请参考 sysLib 库描述。

sysScsiInit()

函数原型：

```
STATUS sysScsiInit (void)
```

功能描述：

该函数创建并初始化 SCSI 控制结构，使能板上 SCSI 接口。该函数同时会将适当的中断服务程序连接到中断上，并将中断使能。

如果支持 SCSI DMA，同时在系统中定义了 INCLUDE_SCSI_DMA，该函数也负责初始化 DMA。

返回值：

成功初始化则返回 OK，如果不能连接控制结构、不能初始化控制器或者 DMA 中断无法连接上则返回 ERROR。

参考：

相关信息请参考 sysLib 库描述。

sysScsiConfig()

函数原型：

STATUS sysScsiConfig (void)

功能描述：

该函数是 SCSI 配置例子函数。

该函数所包含的大多数代码均是 SCSI 外设的配置例子。用户必须根据实际的 SCSI 总线配置来修改该函数。修改过程可以参考 src/config/usrScsi.c 中的例子。

用户可以通过定义 SCSI_AUTO_CONFIG 来测试硬件，定义了这个宏之后系统会扫描总线并显示出找到的设备。任何一个设备的 SCSI 总线 ID 都会不同。

这里将列举三个配置例子。这三个例子分别是 SCSI 硬盘配置、OMTI 3500 软盘配置、磁带驱动器配置。

硬盘配置

这里将硬盘分为两个 32M 字节的分区和一个包含剩余空间的分区。第一个分区初始化为 dosFs 设备。第二个和第三个初始化为 rt11Fs 设备，每个设备都包含 256 个目录条目。

推荐的做法是将块设备上的第一个分区作为 dosFs 设备，这样可以使该设备成为 VxWorks 引导设备。这样的做法将会相当简化操作。

软盘配置

由于软盘是可移动媒体，并且通常只有一个分区，所以为了与 IBM PC 机相兼容，dosFs 是使用该设备文件系统的明智选择。

与硬盘配置不同，该例中的软盘配置非常复杂。注意其中两次调用了 scsiPhysDevCreate()。第一次的调用仅仅是为了获取 scsiModeSelect()函数所需要的句柄，以确保默认的媒体未发生变更。

在正确的硬件配置之后也就是第二次调用 scsiPhysDevCreate()配置硬件之后，该句柄被 scsiPhysDevDelete()函数注销。（在 scsiModeSelect()函数调用之前，硬件配置信息是错误的）。注意在调用 scsiBlkDevCreate()之后，必须将正确的 sectorsPerTrack 和 nHeads 值通过 scsiBlkDevInit()函数写入，以便兼容 IBM PC 机。

磁带驱动器配置

由于 VxWorks 中将适当的设备参数关闭并取代以固定的数据块大小，所以磁带驱动器的配置比较复杂。

dosFsDevInit()函数的最后一个参数是指向 DOS_VOL_CONFIG 结构的指针。如果指定为 NULL，则系统会要求 dosFsDevInit()函数从驱动器中的磁盘读取该信息。如果此时未提供磁盘或者磁盘上没有有效的 dosFs 目录，函数将会运行失败。此时可以使用 dosFsMkfs()函数在磁盘上创建一个新的目录。但是该函数使用了默认的参数，可能不适合用户的应用。所以应该为函数 dosFsDevInit()提供一个指向已经初始化了的结构 DOS_VOL_CONFIG 的指针。如果使用了 dosFsDevInit()，diskInit()函数将会向空磁盘或者可写磁盘中写入一个目录。

注意：

变量 pSbdFloppy 被定义为全局变量，以便上述函数可以在 shell 中调用，如：

-> dosFsMkfs "/fd0/", pSbdFloppy

如果磁盘是新的，可以使用 diskFormat()将其格式化。

返回值：

配置成功则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 sysLib 库描述。

sysLocalToBusAdrs()

函数原型：

STATUS sysLocalToBusAdrs

```
(
int adrsSpace,          /* 总线地址空间 */
char * localAdrs,       /* 本地地址 */
char * *pBusAdrs        /* 存放总线地址 */
)
```

功能描述：

该函数获取指定的本地内存对应的总线地址。

返回值：

成功获取则返回 OK，如果本地地址空间未知或者未映射则返回 ERROR。

参考：

相关信息请参考 sysLib 库描述以及函数 sysBusToLocalAdrs()。

sysBusToLocalAdrs()

函数原型：

```
STATUS sysBusToLocalAdrs
(
int adrsSpace,          /* 总线地址空间 */
char * busAdrs,         /* 总线地址 */
char * *pLocalAdrs      /* 存放本地地址*/
)
```

功能描述：

该函数获取指定总线内存地址所对应的本地地址。

返回值：

获取成功则返回 OK，如果映射错误或内存地址未知则返回 ERROR。

参考：

相关信息请参考 sysLib 库描述以及函数 sysLocalToBusAdrs()。

sysSerialHwInit()

函数原型：

```
void sysSerialHwInit (void)
```

功能描述：

该函数初始化 BSP 串行设备描述符并将设备置为静止状态。该函数由 sysHwInit()函数调用。

返回值：

无。

参考：

相关信息请参考 sysLib 库描述。

sysSerialHwInit2()

函数原型：

```
void sysSerialHwInit2 (void)
```

功能描述：

该函数连接 BSP 串行设备中断，由 sysHwInit2()函数调用。串行设备中断不能在 sysSerialHwInit()中连接，因为当时内核内存分配器还未初始化，而 intConnect()函数调用了 malloc()函数。

返回值：
无。
参考：
相关信息请参考 sysLib 库描述。
sysSerialReset()
函数原型：
void sysSerialReset (void)

功能描述：
该函数由 sysToMonitor()函数调用，用于重置所有的 SIO 设备防止他们产生中断或者执行 DMA 操作。
返回值：
无。
参考：
相关信息请参考 sysLib 库描述。
sysSerialChanGet()
函数原型：

```
SIO_CHAN * sysSerialChanGet
(
int channel  /* 串口通道 */
)
```

功能描述：
该函数获取 SIO_CHAN 设备所连接的串行通道。
返回值：
指向对应通道的 SIO_CHAN 结构的指针，如果通道无效则返回 ERROR。
参考：
相关信息请参考 sysLib 库描述。

8.2 VxWorks 内核函数

8.2.1 函数库描述

1．库命名
VxWorks 核心函数库名称为 kernelLib。
2．函数
表 8-2 中列出了 VxWorks 内核函数。
3．描述
VxWorks 内核为应用提供任务控制服务。这些函数库 kernelLib、taskLib、semLib、tickLib 和 wdLib 组成了内核功能。该函数库是 VxWorks 核心初始化、版本信息、调度控制的接口。

表 8-2 VxWorks 内核函数

函　数	描　述
kernelInit()	初始化 VxWorks 内核
kernelVersion()	返回内核版本号
kernelTimeSlice()	使能时间片轮转方式

4．内核初始化
在任何内核操作执行前必须初始化内核。通常 usrConfig.c 中的系统配置代码 usrInit()会执行内核初始化。
内核初始化包含下列内容：

1）定义系统内存分区的开始地址和大小。函数 malloc()将会使用该分区为 VxWorks 中的分配请求分配内存。

2）为中断堆栈分配内存。通常中断服务程序会使用这段堆栈，除非体系结构不支持独立的中断堆栈(这时中断服务程序将使用中断任务的堆栈)。

3）指定中断锁定级别。VxWorks 在进行任何操作时都不会超过这个级别。中断锁定级别通常用于定义最高可能的优先级。然而，在需要最短中断延迟的时候，中断锁定级别应使系统可以适时地响应中断。处理高于中断锁定级别的中断的服务程序将不能调用任何 VxWorks 函数。

一旦内核初始化完成，系统会以指定入口点的函数生成根任务并指定其堆栈大小。通常根任务的入口点是 usrConfig.c 模块中的 usrRoot()函数。VxWorks 其他的初始化将在 usrRoot()中完成。

5．时间片轮转调度

时间片轮转调度允许相同优先级的任务分享处理器时间。如果不使用时间片轮转调度，当相同优先级的多任务共享处理器时，单一的任务会独占处理器资源直到高优先级的任务抢占处理器资源，而不是将处理器资源释放给同优先级的任务。

默认的状态下时间片轮转调度是禁用的，可以使用函数 kernelTimeSlice()将其打开或关闭。该函数的参数以时间片为单位，决定了每一个任务在一次调度中可以占用的时间片的数量，如果参数为零，则表示关闭时间片轮转调度。如果同时使用了时间片轮转调度和优先级抢占调度，函数 tickAnnounce()可以用于增加正在执行任务的时间片数量。当指定的时间片数量用完之后，系统将会清除时间片计数器同时会将任务放置在同优先级的就绪队列末尾。如果有新的任务生成，它会被加入到同优先级的就绪任务队列末尾并且其时间片计数器会被初始化为零。

如果某个任务在运行时被更高优先级的任务抢占了处理器资源，该任务的时间片计数器不会改变，并且在其得到处理器资源时会继续开始计数。如果在使用时间片轮转调度的同时禁止了优先级抢占式调度，时间片计数器不可以增加。

6．头文件

任务信息函数声明在 kernelLib.h 头文件中。

7．参考

相关信息请参考 kernelLib、taskLib、intLib 库描述。

8.2.2　任务信息函数详细描述

kernelInit()

函数原型：

```
void kernelInit
(
FUNCPTR rootRtn,          /* 用户启动程序 */
unsigned rootMemSize,     /* TCB 和根任务堆栈的内存 */
char * pMemPoolStart,     /* 内存池起始地址 */
char * pMemPoolEnd,       /* 内存池结束地址 */
unsigned intStackSize,    /* 中断堆栈大小 */
int lockOutLevel          /* 中断屏蔽级(1-7) */
)
```

功能描述：

该函数初始化并运行内核。在一次运行中应该只运行一次。参数 rootRtn 指定了用户初始化代码的入口点，通常 rootRtn 会被设置为 usrRoot()。

中断会在 kenelInit()退出之后被打开。

kernelInit()使用参数 pMemPoolStart 和 pMemPoolEnd 初始化系统内存分区。在支持独立中断堆栈的体系结构中系统还会为中断堆栈分配一块从 pMemPoolStart 开始的 intStackSize 大小的内存。

返回值：

无。

参考：

相关信息请参考 kernelLib 库描述和 intLockLevelSet()函数。

kernelVersion()

函数原型：

char *kernelVersion (void)

功能描述：

该函数返回包含内核版本的字符串。该字符串的格式形如“WIND version x.y”，其中 x 代表内核的主版本号，y 代表内核的子版本号。

返回值：

指向格式为“WIND version x.y”的版本字符串的指针。

参考：

相关信息请参考 kernelLib 库描述。

操作示范：

在宿主机 WindSh 工具中，kernelVersion()函数操作示范如下：

```
-> printf("Kernel:%s.\n",kernelVersion())
Kernel:WIND version 2.5.
value = 25 = 0x19
```

kernelTimeSlice()

函数原型：

```
STATUS kernelTimeSlice
(
int ticks      /* 以 ticks 形式表示的时间片或 0（禁用时间片轮转调度）*/
)
```

功能描述：

该函数打开同优先级之间的时间片轮转调度并将系统的时间片设置为 ticks。默认的情况下时间片轮转调度是禁用的。ticks 若取值为“0”则禁用时间片轮转调度。

更详细的信息参见本函数库描述。

返回值：

永远返回 OK。

参考：

相关信息请参考 kernelLib 库描述。

第9章 用户部分

9.1 用户接口子程序

9.1.1 子程序库描述

1．库命名

用户接口子程序库名称为 usrLib。

2．函数

表 9-1 中列出了用户接口子程序。

表 9-1 用户接口子程序

子程序	描　　述	子程序	描　　述
help()	打印可选程序描述纲要	squeeze()	回收 RT-11 卷上的空闲碎片空间
netHelp()	打印网络程序描述纲要	ld()	装载对象模块到内存中
bootChange()	改变引导行参数	ls()	列出目录的内容
periodRun()	周期性调用函数	ll()	详细列出目录的内容
period()	创建任务来周期性调用函数	lsOld()	列出 RT-11 目录的内容
repeatRun()	多次重复调用一个函数	mkdir()	创建目录
repeat()	创建任务多次重复调用一个函数	rmdir()	删除目录
sp()	用默认参数创建任务	rm()	删除文件
checkStack()	打印每个任务的堆栈使用概要	devs()	列出系统知道的所有设备
i()	打印每个任务的 TCB 概要	lkup()	列出符号
ti()	打印任务的 TCB 完整信息	lkAddr()	列出在指定值附近的符号
show()	打印指定对象的信息	mRegs()	修改寄存器
ts()	挂起任务	pc()	返回程序计数器的内容
tr()	恢复挂起的任务	printErrno()	打印指定错误状态值的定义
td()	删除任务	printLog()	打印 VxWorks 标识语
version()	打印 VxWorks 版本信息	logout()	退出 VxWorks 系统
m()	修改内存	h()	显示或设置 shell 历史记录显示大小
d()	显示内存	spyReport()	显示任务活动(activity)数据
cd()	改变默认目录	spyTask()	运行周期性任务进行活动汇报
pwd()	打印当前默认目录	spy()	开始周期性任务进行活动汇报
copy()	文件复制	spyClkStart()	开始收集任务活动数据
copyStreams()	流复制	spyClkStop()	停止收集任务活动数据
diskFormat()	格式化磁盘	spyStop()	停止任务监视和汇报
diskInit()	在块设备上初始化文件系统	spyHelp()	显示任务监视帮助菜单

3．描述

usrLib 库提供一系列可以在 VxWorks shell 上执行的函数。这些函数对于任务监测和执行、系统信息、符号表管理等操作都非常有效。

相对于 VxWorks 中的其他函数，库中的许多函数是简单命令接口。用户可以自由地修改或扩展这个库，并通过创建新的私有库可以实现更好的定制功能，这些库可以作为单个模块链接到系统中。

有一些函数包含可选参数。如果这些参数为 0，shell 将使用默认参数来支持这些函数。

该模块中的许多函数采用可选的任务名或 ID 作为参数。如果忽略参数或为 0，则函数使用“最近的”任务。最近的任务(或默认任务)是最后引用的任务。usrLib 库使用 taskIdDefault()函数设置和获得最后引用的任务 ID，以供 VxWorks 的其他函数使用。

4．头文件

用户接口子程序声明请查阅 usrLib.h 头文件。

5．参考

相关信息请参考 spyLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell、Windsh 章节。

9.1.2 用户接口子程序函数详细描述

help()

函数原型：

```
void help (void)
```

功能描述：

打印可选程序纲要。这个命令打印下面通用程序调用次序列表，且大部分函数包含在 usrLib 库中：

```
help                Print this list
dbgHelp             Print debug help info
...
h [n]               Print (or set) shell history
i [task]            Summary of tasks' TCBs
ti task             Complete info on TCB for task
sp adr,args...      Spawn a task, pri=100, opt=0, stk=20000
...
diskInit "device"   Initialize file system on disk
squeeze "device"    Squeeze free space on RT-11 device
NOTE: Arguments specifying <task> can be either task ID or name.
```

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell、Windsh 章节。

操作示范：在 WindSh 下执行 help()函数

```
-> help
help                    Print this list
h           [n]         Print (or set) shell history
i           [task]      Summary of tasks' TCBs
ti          ask         Complete info on TCB for task
...
NOTE:   Arguments specifying <task> can be either task ID or name.
value = 0 = 0x0
```

netHelp()

函数原型：

```
void netHelp (void)
```

功能描述：

打印网络程序纲要。这个命令打印网络模块纲要，是 shell 中典型的调用。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell。

操作示范：在 target shell 下执行 netHelp()函数

```
-> netHelp
hostAdd     "hostname","inetaddr"     - add a host to remote host table;
                                      "inetaddr" must be in standard
                                      Internet address format e.g. "90.0.0.4"
hostShow                              - print current remote host table
...
EXAMPLE:    -> hostAdd "wrs", "90.0.0.2"
            -> netDevCreate "wrs:", "wrs", 0
            -> iam "fred"
            -> copy <wrs:/etc/passwd        /* copy file from host "wrs" */
            -> rlogin "wrs"                 /* rlogin to host "wrs" */
value = 1 = 0x1
```

bootChange()

函数原型：

```
void bootChange (void)
```

功能描述：

改变引导行参数。这个命令改变引导 ROM 中的引导行参数。在远程登录(login)期间这个命令非常有用。在改变引导参数后，用户可以通过 reboot()命令重启目标机，然后结束登录并再次远程登录。

如果目标机有非易失性 RAM，这个命令把新的引导行参数保存在非易失性 RAM 中。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell。

操作示范：在 target shell 下执行 bootChange()函数

```
-> bootChange
'.' = clear field;   '-' = go to previous field;   ^D = quit

boot device             : nt0
processor number        : 0
host name               : host
...
```

periodRun()

函数原型：

```
void periodRun
(
int secs,              /* 调用间的延迟时间(秒数) */
FUNCPTR func,          /* 周期调用的函数 */
int arg1,              /* 以下是传递给函数的 8 个参数 */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8
)
```

功能描述：

周期性调用函数。这个命令周期性调用指定的函数，其调用之间延迟时间为指定的秒数。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，以及 period()函数和“VxWorks Programmer's Guide”中的目标机 shell。

操作示范：在 target shell 下通过 periodRun()函数每隔 5s 执行“i”命令

```
-> periodRun(5,i,0,0,0,0,0,0,0,0)
```

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	_excTask	50890a0	0	PEND	4358c2	5088fbc	1c0001	0
tLogTask	_logTask	5084688	0	PEND	4358c2	50845a0	0	0
tShell	_shell	507c358	1	READY	41eca0	507b754	0	0
tWdbTask	40e478	5080610	3	PEND	4358c2	50804f4	0	0

...

period()

函数原型：

```
void period
(
int secs,              /* 以秒形式的时间周期 */
FUNCPTR func,          /* 周期调用的函数 */
int arg1,              /* 以下是传递给函数的 8 个参数 */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
```

```
    int arg8
    )
```

功能描述：

创建任务周期性调用一个函数。这个命令创建一个任务周期性调用指定的函数，其调用之间延迟时间为指定的秒数。

注意：

该命令调用 sp()函数创建这个任务。有关信息请阅读 sp()的详细描述。

返回值：

任务 ID，如果创建任务失败则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，periodRun()和 sp()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 target shell 下通过 period()函数创建一个任务每隔 10 秒执行“i”命令

```
-> period(10,i,0,0,0,0,0,0,0,0)
task spawned: id = 0x506da98, name = t1
value = 84335256 = 0x506da98
->
```

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	_excTask	50890a0	0	PEND	4358c2	5088fbc	0	0
tLogTask	_logTask	5084688	0	PEND	4358c2	50845a0	0	0
tShell	_shell	507c358	1	PEND	4358c2	507bfc0	d0001	0
tWdbTask	40e478	5080610	3	PEND	4358c2	50804f4	0	0
t1	_periodRun	506da98	100	READY	41eca0	506d058	0	0

...

repeatRun()

函数原型：

```
    void repeatRun
    (
    int n,              /* 调用次数 */
    FUNCPTR func,       /* 调用的函数 */
    int arg1,           /* 以下是传递给函数的 8 个参数 */
    int arg2,
    int arg3,
    int arg4,
    int arg5,
    int arg6,
    int arg7,
    int arg8
    )
```

功能描述：

多次重复调用一个函数。这个命令 n 次重复调用指定函数。如果 n 为 0，系统将不停地调用这个函数。

通常，这个函数只是被 repeat()调用，它把这个函数作为一个任务来创建。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，repeat()函数、以及“VxWorks Programmer's Guide”中的目标机 shell。

操作示范：在 target shell 下通过 repeatRun()函数重复 3 次调用 checkStack()函数

```
-> repeatRun(3,checkStack,0,0,0,0,0,0,0,0)
```

NAME	ENTRY	TID	SIZE	CUR	HIGH	MARGIN
tExcTask	excTask	50890a0	15984	228	2452	13532
tLogTask	logTask	5084688	12984	232	2100	10884
tShell	shell	507c358	57320	2756	4348	52972
tWdbTask	0x000040e478	5080610	16176	284	3840	12336
INTERRUPT			50000	0	0	50000

...

repeat()

函数原型：

```
void repeat
(
int n,              /* 调用次数 */
FUNCPTR func,       /* 调用的函数 */
int arg1,           /* 以下是传递给函数的 8 个参数 */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8
)
```

功能描述：

创建任务多次重复调用一个函数。这个命令创建一个任务 n 次重复调用指定函数。如果 n 为 0，系统将不停地调用这个函数或直到删除创建的任务。

注意：

该命令调用 sp()函数创建这个任务。有关信息请阅读 sp()的详细描述。

返回值：

任务 ID，如果创建任务失败则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，repeatRun()和 sp()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 target shell 下通过 repeat()函数创建一个任务来重复 3 次调用 version()函数

```
-> repeat(3,i,0,0,0,0,0,0,0,0)
task spawned: id = 0x506da98, name = t2
```

```
value = 84335256 = 0x506da98
->
NAME        ENTRY       TID      PRI  STATUS    PC       SP       ERRNO    DELAY
----------  ----------  -------- ---  --------- -------- -------- -------  -----
tExcTask    _excTask    50890a0  0    PEND      4358c2   5088fbc  1c0001   0
tLogTask    _logTask    5084688  0    PEND      4358c2   50845a0  0        0
tShell      _shell      507c358  1    PEND      4358c2   507bfc0  d0001    0
tWdbTask    40e478      5080610  3    PEND      4358c2   50804f4  0        0
t2          _repeatRun  506da98  100  READY     41eca0   506d058  0        0
...
```

sp()

函数原型：

```
int sp
(
FUNCPTR func,  /* 函数 */
int arg0,
int arg1,          /* 以下是传递给任务的 10 个参数 */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8,
int arg9
)
```

功能描述：

用默认参数创建任务。这个命令用下面默认值创建任务。

priority：100；

stack size：20,000 字节；

task ID：非当前使用的最高级；

task options：VX_FP_TASK（浮点协处理器支持）；

task name：tN 组成的任务名，在这里 N 是一个整数，每创建一个新任务它就增加，例如：t1、t2、t3等。

在任务创建后，系统显示任务 ID。

这个命令是 taskSpawn()函数的缩写，便于用默认参数创建任务。如果不愿意使用默认值，则应该直接调用 taskSpawn()函数。

返回值：

成功创建则返回任务 ID，如果创建任务失败则返回 ERROR。

参考：

相关信息请参考 usrLib、taskLib 库描述，以及 taskSpawn()函数和“VxWorks Programmer's Guide”中的目标机 shell。

操作示范：在 target shell 下通过 sp()函数创建一个任务(入口为 hello)周期显示“Test sp() function!”字符

串

```
-> sp(hello,0,0,0,0,0,0,0,0,0)
task spawned: id = 0x506da98, name = t1
value = 84335256 = 0x506da98
-> i
   NAME   ENTRY       TID      PRI  STATUS      PC       SP       ERRNO    DELAY
  ---------- ------------ -------- ---  ----------  --------  --------  -------  -----
tExcTask   _excTask    50890a0  0    PEND        4358fe   5088fbc  0        0
...
t1         _hello      506da98  100  DELAY       4358fe   506da00  0        404
...
Test sp() function!
Test sp() function!
...
```

其中，hello()函数原型如下：

```
void hello(void)
{
for(;;)
{
printf("Test sp() function!\n");
taskDelay (1000);
}
}
```

checkStack()

函数原型：

```
    void checkStack
    (
    int taskNameOrId      /* 任务名或任务 ID; 0 = 所有任务的概要 */
    )
```

功能描述：

打印每个任务的堆栈使用概要。这个命令显示指定任务或所有任务(不给定参数)的堆栈使用概要。总结包括堆栈总的大小(SIZE)、当前使用的堆栈字节数(CUR)、已用的最大堆栈字节数(HIGH)和从未使用的堆栈字节数(MARGIN = SIZE - HIGH)。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，以及 taskSpawn()函数和"VxWorks Programmer's Guide"中的目标机 shell、windsh。

操作示范：在 WindSh 下执行 checkStack()函数

```
-> checkStack
   NAME       ENTRY       TID      SIZE   CUR    HIGH   MARGIN
---------- ------------ -------- -----  -----  -----  ------
tExcTask   _excTask    50890a0  15984  228    2096   13888
```

tLogTask	_logTask	5084688	12984	232	2100	10884
tShell	_shell	507c358	57320	920	4348	52972
tWdbTask	0x40e4b4	5080610	16176	284	3840	12336

i()

函数原型：

```
void i
(
int taskNameOrId /* 任务名或任务 ID, 0 = 所有任务的概要 */
)
```

功能描述：

打印每个任务的 TCB 摘要。该命令显示系统中所有任务的概要。i()函数在一个指定的任务上可以提供更多的信息。

i()和 ti()函数都使用 taskShow()函数。有关详细信息请阅读 taskShow()函数。

警告：

这个命令只是作为调试目的使用，显示后相应的信息已过时。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，ti()和 taskShow()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 WindSh 下执行 i()函数

```
-> i
  NAME   ENTRY       TID      PRI  STATUS    PC       SP       ERRNO   DELAY
---------- ------------ -------- ---  ----------  --------  --------  -------  -----
...
tDbgTask   _windDemo    5060360  100  SUSPEND   4eaecae   50602e8  0       0
tHighPri   _taskHighPri 50667f0  150  PEND      4358fe    5066744  0       0
tLowPri    _taskLowPri  50635a8  200  DELAY     4eaefa7   5063528  0       0
```

ti()

函数原型：

```
void ti
(
int taskNameOrId        /* 任务名或任务 ID; 0 = 默认 */
)
```

功能描述：

打印任务的 TCB 完整信息。这个命令打印指定任务的任务控制块(TCB)内容，其中包括寄存器。如果忽略 taskNameOrId 或为 0，则假设为最后引用的任务。

ti()函数使用了 taskShow()函数；有关详细信息请阅读 taskShow()函数。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，taskShow()函数，以及“VxWorks Programmer's Guide”中的目标机 shell

和 windsh。

操作示范：在 WindSh 下执行 ti()函数

```
-> ti
  NAME        ENTRY        TID      PRI  STATUS    PC       SP       ERRNO   DELAY
----------  ------------  --------  ---  ----------  --------  --------  -------  -----
tHighPri    _taskHighPri  506da98   150  PEND      4358fe    506d9cc  0        0
stack: base 0x506da98   end 0x506abb8   size 11984    high 2084    margin 9900
options: 0x5
VX_SUPERVISOR_MODE VX_DEALLOC_STACK

edi  = ffffffff   esi  =  508b958   ebp  =  506d9d4   esp  =  506d9cc
ebx  =      0     edx  =    4358fe   ecx  =    10101   eax  =  fffffffe
eflags =   212    pc   =    4358fe
```

show()

函数原型：

```
void show
(
int objId,    /* 对象 ID */
int level     /* 信息级别 */
)
```

功能描述：

打印指定对象的信息。系统对象包括任务、本地和共享信号量、本地和共享消息队列、本地和共享内存块、看门狗和符号表。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，i()、ti()和 lkup()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

操作示范：在 WindSh 下执行 show()函数显示一个信号量对象信息

```
-> show semId
Semaphore Id          : 0x508c5e0
Semaphore Type        : BINARY
Task Queueing         : PRIORITY
Pended Tasks          : 0
State                 : EMPTY
```

ts()

函数原型：

```
void ts
(
int taskNameOrId      /* 任务名或任务 ID */
)
```

功能描述：

挂起任务。这个命令挂起指定任务。它只是调用 taskSuspend()函数。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，tr()和 taskSuspend()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

tr()

函数原型：

```
void tr
(
int taskNameOrId        /* 任务名或任务 ID */
)
```

功能描述：

恢复挂起的任务。这个命令恢复一个挂起任务的执行。它只是调用 taskResume()函数。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，ts()和 taskResume()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

td()

函数原型：

```
void td
(
int taskNameOrId        /* 任务名或任务 ID */
)
```

功能描述：

删除任务。这个命令删除指定的任务。它只是调用 taskDelete()函数。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，taskDelete()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

version()

函数原型：

```
void version (void)
```

功能描述：

打印 VxWorks 版本信息。这个命令打印 VxWorks 的颁布号，内核 wind 颁布号，VxWorks 映像构造日期和其他相关信息。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

m()

函数原型：

```
void m
(
void * adrs,        /* 内存地址 */
int width           /* 修改单元的宽度 (1, 2, 4, 8) */
)
```

功能描述：

修改内存。这个命令提示用户在指定的地址处开始修改内存，并依次打印每一个地址和当前地址的内容。如果 adrs 或 width 为 0 或省略，默认使用前面的值。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，mRegs()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 WindSh 下执行 m()函数

```
-> m(0x41bc98,8)
0041bc98:   83085d8b53e58955-
0041bca0:   5473ff2b75011c7b-
0041bca8:   73ff4c73ff5073ff-.
value = 1 = 0x1
```

d()

函数原型：

```
void d
(
void * adrs,        /* 内存地址 (如果为 0, 显示下一个内存块 */
int nunits,         /* 显示单元数(如果为 0, 使用默认值) */
int width           /* 显示单元的宽度(1, 2, 4, 8) */
)
```

功能描述：

显示内存。这个命令在 adrs 处开始显示内存的内容。如果忽略 adrs 或为 0，则 d()显示下一个内存块，地址从最后 d()命令开始。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，m()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 WindSh 下执行 d()函数

```
-> d(0x41bc98,20,8)
0041bc90:                   8955 53e5 5d8b 8308    *          U..S.]..*
```

```
0041bca0:   1c7b 7501 ff2b 5473 73ff ff50 4c73 73ff     *{..u+.sT.sP.sL.s*
0041bcb0:   ff48 4473 73ff ff40 3c73 73ff ff38 3473     *H.sD.s@.s<.s8.s4*
```

cd()

函数原型：

```
STATUS cd
(
char * name        /* 新目录名 */
)
```

功能描述：

改变默认目录。这个命令设置默认目录为 name，且默认目录是一个设备名。

返回值：

成功设置则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，pwd( )函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

pwd()

函数原型：

```
void pwd (void)
```

功能描述：

打印当前默认目录。这个命令显示当前工作设置/目录。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，cd()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 WindSh 下执行 pwd()函数显示当前默认目录

```
-> pwd
D:/Tornado/target/proj/Project9
value = 0 = 0x0
```

copy()

函数原型：

```
STATUS copy
(
char * in,    /* 读文件的文件名 (如果为 NULL，假设为 stdin) */
char * out    /* 写文件的文件名 (如果为 NULL，假设为 stdout) */
)
```

功能描述：

文件复制。把 in 文件复制到 out 文件中。这个命令从输入文件复制到输出文件中，直到源文件末尾。

返回值：

成功复制则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，copyStreams()和 tyEOFSet()函数，以及“VxWorks Programmer's Guide”

中的目标机 shell。

copyStream()

函数原型：

```
STATUS copyStreams
(
int inFd,    /* 从这个流文件描述符中复制 */
int outFd    /* 复制到这个流文件描述符中 */
)
```

功能描述：

流复制。这个命令从 inFd 中复制到 outFd 中，直到 inFd 中文件的结尾。

返回值：

成功复制则返回 OK，如果读或写有错误则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，copy()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

diskFormat()

函数原型：

```
STATUS diskFormat
(
char * devName          /* 设备名 */
)
```

功能描述：

格式化磁盘。这个命令格式化磁盘，并在上面创建文件系统。设备驱动程序必须已经创建这个设备并用指定的文件系统初始化它，例如 dosFsDevInit()或 rt11FsDevInit()。

这个命令调用 ioctl()来执行 FIODISKFORMAT 功能。

返回值：

成功格式化则返回 OK，如果打开设备或格式化失败则返回 ERROR。

参考：

相关信息请参考 usrLib、dosFsLib 和 rt11FsLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell。

diskInit()

函数原型：

```
STATUS diskInit
(
char * devName   /* 设备名 */
)
```

功能描述：

在块设备上初始化文件系统。这个命令在块设备上创建一个新的、空白的文件系统。设备驱动程序必须已经创建这个设备并用指定的文件系统初始化它，例如 dosFsDevInit()或 rt11FsDevInit()。

这个命令调用 ioctl()来执行 FIODISKINIT 功能。

返回值：

成功初始化则返回 OK，如果打开或初始化设备失败则返回 ERROR。

参考：

相关信息请参考 usrLib、dosFsLib 和 rt11FsLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell。

squeeze()

函数原型：

```
STATUS squeeze
(
char * devName   /* RT-11 设备，例如："/fd0/" */
)
```

功能描述：

回收 RT-11 卷上的空闲碎片空间。这个命令移动 RT-11 卷上的数据，以及合并空闲空间的区域。

注意：

当调用这个程序的时候，不应该打开设备文件。由于不知道一些文件的连续条件，而对它们进行写操作会毁坏整个磁盘。

返回值：

成功回收则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell。

ld()

函数原型：

```
MODULE_ID ld
(
int syms,          /* -1, 0, 或 1 */
BOOL noAbort,  /* TRUE = 忽略错误 */
char * name        /* 对象模块名, NULL = 标准输入 */
)
```

功能描述：

装载对象模块到内存中。这个命令装载来自于文件或标准输入设备的对象模块，且对象模块必须是 UNIX a.out 格式。在装载期间，将解决模块中的外部引用。参数 syms 决定装载什么符号；可能值：

0：添加全局符号到系统符号表中；

1：添加全局和局部符号到系统符号表中；

-1：符号不添加到系统符号表中。

如果在装载期间出现错误(例如：没有定义外部引用)，将调用 shellScriptAbort()函数停止装载。如果 noAbort 为 TRUE，表明错误忽略不计。

在调试期间使用 ld()的通常方式是装载所有的符号，以后只是装载全局符号。

返回值：

成功装载则返回 MOULE_ID，如果读文件出现错误、或对象文件格式无效、或太多符号则返回 NULL。

参考：

相关信息请参考 usrLib、loadLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

ls()

函数原型：

```
STATUS ls
```

```
(
char * dirName,        /* 目录名 */
BOOL doLong            /* 如果为 TRUE，详细地显示列表 */
)
```

功能描述：

列出目录的内容。这个命令类似于 UNIX ls 命令。它可以通过两种格式中的一种格式列出一个目录的内容。如果 doLong 为 FALSE，只是显示指定目录中的文件名(或子目录)。如果 doLong 为 TRUE，将显示文件名、大小、日期和时间。

参数 dirName 指定要显示的目录。如果忽略这个参数或为 NULL，则显示当前工作目录。

注意：

当使用 netDrv 设备(FTP 或 RSH)时，参数 doLong 无效。

返回值：

成功列出则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，ll()、lsOld()和 stat()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

ll()

函数原型：

```
STATUS ll
(
char * dirName    /* 目录名 */
)
```

功能描述：

详细列出目录的内容。这个命令详细列出一个目录的内容。它等价于：

```
-> ls dirName, TRUE
```

返回值：

成功列出则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，ls()和 stat()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

lsOld()

函数原型：

```
STATUS lsOld
(
char * dirName /* 设备名 */
)
```

功能描述：

列出 RT-11 目录的内容。这个命令是 ls()的旧版本，使用旧类型 ioctl()函数的 FIODIRENTRY 功能来获得目录的信息。自 VxWorks 5.0 以后，使用 ls()的新版本。

返回值：

成功操作则返回 OK，如果不能打开目录则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，ls()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

mkdir()

函数原型：

```
STATUS mkdir
(
char * dirName    /* 目录名 */
)
```

功能描述：

创建目录。这个命令在一个分等级的文件系统中创建一个新的目录。

返回值：

成功创建则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，rmdir()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

rmdir()

函数原型：

```
STATUS rmdir
(
char * dirName    /* 目录名 */
)
```

功能描述：

删除目录。这个命令从一个分等级的文件系统中删除一个存在的目录。

返回值：

成功删除则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，mkdir()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

rm()

函数原型：

```
STATUS rm
(
char * fileName   /* 文件名 */
)
```

功能描述：

删除文件。这个命令类似 UNIX 命令，它只是调用 remove()函数。

返回值：

成功删除文件则返回 OK，如果失败则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，remove()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

devs()

函数原型：

```
void devs (void)
```

功能描述：

列出系统知道的所有设备。这个命令显示所知道的 I/O 系统设备。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，iosDevShow()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 WindSh 下执行 devs()函数

```
-> devs
drv name
  0 /null
  1 /tyCo/0
  3 host:
  4 /vio
  5 /tgtsvr
```

lkup()

函数原型：

```
void lkup
(
char * substr       /* 子字符串 */
)
```

功能描述：

列出符号。这个命令列出系统符号表中所有包含 substr 字符串的字符。如果忽略 substr 或为 0，只打印符号表简单摘要。如果 substr 是空字符(“”)，则列出表中所有符号。

返回值：

无。

参考：

相关信息请参考 usrLib、symLib 库描述，symEach()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 target shell 下执行 lkup()函数

```
-> lkup
Number of Symbols     : 2400
Symbol Mutex Id       : 0x51511f0
Symbol Hash Id        : 0x51509c8
Symbol memPartId      : 0x472354
Name Clash Policy     : Allowed
value = 0 = 0x0
```

lkAddr()

函数原型：

```
void lkAddr
(
unsigned int addr  /* 地址 */
)
```

功能描述：

列出在指定值附近的符号。这个命令列出在系统符号表中指定值附近的符号。这个符号显示包括：

- 小于指定值的符号；
- 指定值的符号；
- 随后的符号，直到至少显示 12 个符号。

返回值：

无。

参考：

相关信息请参考 usrLib、symLib 库描述，symEach()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 target shell 下执行 lkAddr()函数

```
-> lkAddr 0x00401168
0x00401120 sysClkConnect          text
0x00401168 sysClkDisable          text
0x0040119c sysClkEnable           text
0x004011e0 sysClkRateGet          text
...
```

mRegs()

函数原型：

```
STATUS mRegs
(
char * regName,        /* 寄存器名, NULL 表示所有寄存器 */
int taskNameOrId       /* 任务名或任务 ID, 0 = 默认任务 */
)
```

功能描述：

修改寄存器。这个命令修改指定任务的指定寄存器。如果忽略 taskNameOrId 或为 0，系统将假设为最后引用的任务。如果没有发现指定的寄存器，将打印有效的寄存器列表并返回 ERROR。如果没有指定寄存器，将提示用户给任务的寄存器赋新的值。依次显示每一个寄存器和寄存器当前的值。用户可以采用下面几种方式作出应答：

RETURN：不改变这个寄存器，但继续提示下一个寄存器；

Number：设置寄存器的值为 Number；

.(点)：不改变这个寄存器，并退出；

EOF：不改变这个寄存器，并退出。

所有输入和显示的数字都是十六进制格式，但要把浮点值除外，它可能是双精度型。

返回值：

成功修改则返回 OK，如果任务或寄存器不存在则返回 ERROR。

参考：

相关信息请参考 usrLib 库描述，m()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 WindSh 下执行 mRegs()函数

```
-> mRegs("eax",t1)
eax     : ffffffffe - ffffffffb
value = 0 = 0x0
```

pc()

函数原型：

```
int pc
(
int task      /* 任务 ID */
)
```

功能描述：

返回程序计数器的内容。这个命令从任务的 TCB 中提取程序程序计数器的内容。如果忽略 task 或 task 为 0，则使用当前任务。

返回值：

程序计数器的内容。

参考：

相关信息请参考 usrLib 库描述，ti()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

操作示范：在 WindSh 下执行 pc()函数

```
-> pc tShell
value = 4413694 = 0x4358fe = _windExit + 0x32
```

printErrno()

函数原型：

```
void printErrno
(
int errNo     /* 错误状态码值 */
)
```

功能描述：

打印指定错误状态值的定义。这个命令根据指定的错误状态值显示错误状态字符串。如果已经构造了错误状态符号表并包含在系统中，则这个命令非常有用。如果 errNo 为 0，则通过调用 errnoGet()函数显示当前任务状态。

返回值：

无。

参考：

相关信息请参考 usrLib、errnoLib 库描述，errnoGet()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

操作示范：在 WindSh 下执行 printErrno()函数

```
-> printErrno(0xd0001)
0xd0001 = S_iosLib_DEVICE_NOT_FOUND
value = 0 = 0x0
```

printLogo()

函数原型：

```
void printLogo (void)
```

功能描述：

打印 VxWorks 标识语。这个命令显示 VxWorks 启动时的标语。它也显示 VxWorks 版本号和内核版本号。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

logout()

函数原型：

```
void logout (void)
```

功能描述：

退出 VxWorks 系统。这个命令退出 VxWorks shell 系统。如果一个远程注册存在，它将被停止并恢复到标准 I/O 控制台。

返回值：

无。

参考：

相关信息请参考 usrLib 库描述，rlogin()、telnet()和 shellLogout()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

h()

函数原型：

```
void h
(
int size      /* 0 = 显示, >0 = 设置历史记录新的大小 */
)
```

功能描述：

显示或设置 shell 历史记录显示大小。如果没有指定参数，则显示 shell 历史记录。如果指定 size，保存并显示最近命令数。size 的最初值为 20。

返回值：

成功显示或设置 shell 历史记录显示大小则返回 OK，如果失败则返回 ERROR。

参考：

相关信息请参考 usrLib、ledLib 库描述，shellHistory()函数，以及“VxWorks Programmer's Guide”中的目标机 shell、windsh。

spyReport()

函数原型：

```
void spyReport (void)
```

功能描述：

显示任务活动(activity)数据。这个函数在中断级报告每个任务使用 CPU 时间总计，中断花费时间总计，内核花费时间总计，空闲时间总计。时间是以 tick 形式和一个百分数形式显示，自最后调用 spyClkStart()和 spyReport()函数以后显示数据。如果自调用 spyReport()函数以后没有发生中断，则不显示任何内容。

返回值：

无。

参考：

相关信息请参考 usrLib、spyLib 库描述，spyClkStart()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

操作示范：在 WindSh 下执行 spyReport()函数

```
-> spyClkStart
value = 0 = 0x0
-> spyReport
NAME            ENTRY       TID       PRI   total % (ticks)      delta % (ticks)
--------        --------    -----     ---   ----------------     ----------------
tExcTask        excTask     50790a0   0     0% (       0)        0% (       0)
...
INTERRUPT                                    0% (       0)        0% (       0)
IDLE                                         99% (    2748)       99% (    1193)
TOTAL                                        99% (    2749)       99% (    1194)
```

spyTask()

函数原型：

```
void spyTask
(
int freq      /* 以秒形式的报告频率 */
)
```

功能描述：

运行周期性任务进行活动汇报。通过 spy()把这个函数创建成一个任务，从而提供周期性任务进行活动汇报。它延迟指定的秒数，重复打印报告。

返回值：

无。

参考：

相关信息请参考 usrLib、spyLib 库描述，spy()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

spy()

函数原型：

```
void spy
(
int freq,              /* 秒形式的报告频率, 0 = 默认值为 5 秒 */
int ticksPerSec        /* 中断时钟频率，0 =默认值为 100 个 ticks */
)
```

功能描述：

开始周期性任务进行活动汇报。这个函数收集任务活动数据并周期运行 spyReport()。参数 ticksPerSec 是每秒钟数据的采集频率，而参数 freq 是报告频率。如果 freq 为 0，默认值为 5 秒。如果 ticksPerSec 为 0 或忽略，则默认值为 100。

这个函数通过 spyTask()执行实际的汇报。在运行 spy()之前，不需要调用 spyClkStart()函数。

返回值：

无。

参考：

相关信息请参考 usrLib、spyLib 库描述，spyClkStart()和 spyTask()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

spyClkStart()

函数原型：

```
STATUS spyClkStart
(
int intsPerSec      /* 定时器中断频率, 0 = 默认值为 100 个 ticks */
)
```

功能描述：

开始收集任务活动数据。这个函数通过使能辅助时钟中断，以 intsPerSec 中断频率开始收集数据。如果 intsPerSec 为 0 或忽略，则频率为 100 个 ticks。先前收集的数据将被清除。

返回值：

成功调用则返回 OK，如果 CPU 没有辅助时钟，或任务创建钩子和删除钩子安装失败则返回 ERROR。

参考：

相关信息请参考 usrLib、spyLib 库描述，sysAuxClkConnect()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

spyClkStop()

函数原型：

```
void spyClkStop (void)
```

功能描述：

停止收集任务活动数据。这个函数屏蔽辅助时钟中断。保留收集的数据直到下次调用 spyClkStart()函数。

返回值：

无。

参考：

相关信息请参考 usrLib、spyLib 库描述，spyClkStart()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

spyStop()

函数原型：

```
void spyStop (void)
```

功能描述：

停止监视和汇报。这个函数调用 spyClkStop()函数。停止通过 spyTask()函数创建的任何周期汇报任务。

返回值：

无。

参考：

相关信息请参考 usrLib、spyLib 库描述，spyClkStop()和 spyTask()函数，以及“VxWorks Programmer's Guide”中的目标机 shell。

spyHelp()

函数原型：

```
void spyHelp (void)
```

功能描述：

显示任务监视帮助菜单。这个函数显示 spyLib 库的摘要。

返回值：

无。

参考：

相关信息请参考 usrLib、spyLib 库描述，以及“VxWorks Programmer's Guide”中的目标机 shell。

9.2 自定义系统配置函数

9.2.1 函数库描述

1．库命名

自定义系统配置函数库名称为 usrConfig。

2．函数

表 9-2 中列出了自定义系统配置函数。

表 9-2 自定义系统配置函数

函　数	描　述
usrInit()	自定义系统初始化函数
usrRoot()	根(root)任务
usrClock()	自定义系统时钟中断函数

3．描述

usrConfig 库是 VxWorks 的配置模块。它包含根任务、主要系统初始化函数、网络初始化函数和时钟中断函数。

头文件 config.h 包含这些函数要使用的与系统相关的参数。

4．头文件

自定义系统配置函数相关宏定义在 config.h 头文件中。

5．参考

相关信息请参考“VxWorks Programmer's Guide”中的配置和构造部分。

9.2.2 自定义系统配置函数详细描述

usrInit()

函数原型：

```
void usrInit
(
int startType       /* 启动类型 */
)
```

功能描述：

自定义系统初始化函数。这个函数是系统引导后首先执行的 C 代码函数。在关闭中断情况下调用该函数，并在这之前，内核处于非多任务环境下。

该函数首先对 bss 赋零；这样把所有的变量初始化成 0，并通过调用 excVecInit()函数初始化所有系统和缺省中断向量；通过调用 sysHwInit()初始化与目标板相关的硬件；安装中断/异常向量；通过调用 usrKernelInit()配置 VxWork 内核；调用 kernelInit()初始化并启动内核多任务环境；usrRoot()是系统的根任务。

另外，这个函数的源代码在 usrConfig.c 文件中。

返回值：

无。

参考：

相关信息请参考 usrConfig，kernelLib 库描述。

usrRoot()

函数原型：

```
void usrRoot
(
char * pMemPoolStart,      /* 系统内存区的起始点 */
unsigned memPoolSize       /* 内存池的初始大小 */
)
```

功能描述：

根任务。这是在多任务环境下第一个运行的任务。它执行所有最终的初始化，然后启动其他任务。

它初始化 I/O 系统，安装设备驱动程序，创建设备（在 configAll.h 和 config.h 中指定），安装网络等。可能创建和装载系统符号表，可能装载和创建另外必要的任务。在默认配置中，它只是初始化 VxWorks shell。

另外，这个函数的源代码在 usrConfig.c 文件中。

返回值：

无。

参考：

相关信息请参考 usrConfig 库描述。

usrClock()

函数原型：

```
void usrClock ()
```

功能描述：

自定义系统时钟中断函数。每次发生时钟中断时系统在中断级调用该函数。在 usrRoot()中通过调用 sysClkConnect()函数安装这个系统时钟中断函数。

如果应用需要在系统时钟中断级上处理某事，可以把它添加到这个函数中。

返回值：

无。

参考：

相关信息请参考 usrConfig 库描述。

第10章 浮点函数

10.1 浮点I/O支持函数

10.1.1 函数库描述

1. 库命名

浮点I/O支持函数库名称为floatLib。

2. 函数

表10-1中列出了浮点I/O支持函数。

表10-1 浮点I/O支持函数

函 数	描 述
floatInit()	初始化浮点格式输入输出支持

3. 描述

该库提供了浮点I/O的格式化输入和输出的支持。浮点的格式输入输出支持不能直接调用，它们都是连接在fioLib库的printf()/scanf()上的。这个工作由函数floatInit()动态完成。如果定义了INCLUDE_FLOATING_POINT宏，系统会在usrConfig.c文件的usrRoot()函数中自动调用floatInit()函数。如果从未执行过floatInit()函数，printf()和scanf()函数将不支持浮点格式的输出。

4. 头文件

浮点支持函数声明在math.h头文件中。

5. 参考

相关信息请参考floatLib, fioLib库描述。

10.1.2 浮点支持函数详细描述

floatInit()

函数原型：

void floatInit (void)

功能描述：

如果需要printf()和scanf()函数支持浮点格式的输入输出，必须调用该函数。如果定义了INCLUDE_FLOATING_POINT，系统会在usrConfig.c文件的usrRoot()函数中自动调用floatInit()函数。

返回值：

无。

参考：

相关信息请参考floatLib库描述。

10.2 与体系结构相关的浮点协处理器支持函数

10.2.1 函数库描述

1. 库命名

与体系结构相关的浮点协处理器支持函数库名称为fppArchLib。

2．函数

表 10-2 中列出了与体系结构相关浮点协处理器支持函数。

3．描述

该函数库中包含了与体系结构相关的浮点协处理器支持。函数 fppSave()和 fppRestore()可以保存和恢复浮点处理器上下文中的内容。函数 fppProbe()检查系统是否存在浮点协处理器。函数 fppTaskRegsSet()和 fppTaskRegsGet()可以获取和设置指定任务的协处理器寄存器内容。

表 10-2 与体系结构相关的浮点协处理器支持函数

函 数	描 述
fppSave()	保存浮点协处理器上下文
fppRestore()	恢复浮点协处理器上下文
fppProbe()	探测当前浮点协处理器
fppTaskRegsGet()	获取 TCB 中的浮点寄存器内容
fppTaskRegsSet()	设置指定任务的浮点寄存器内容

若需要提供浮点处理器支持，必须首先使用函数 fppInit()执行初始化操作。如果用户在 config.h 中定义了 INCLUDE_HW_FP 宏，则系统会在 usrRoot()函数中自动执行该函数。在 I386/I486 体系结构上，VxWorks 屏蔽了 6 个可以发送 IRQ 给 CPU 的 FPU 异常。

4．头文件

与体系结构相关的浮点协处理器支持函数声明在 fppLib.h 头文件中。

5．参考

相关信息请参考 fppLib,fppArchLib 库描述。

10.2.2 与体系结构相关的浮点协处理器支持函数详细描述

fppSave ()

函数原型：

```
void fppSave
    (
    FP_CONTEXT * pFpContext        /* 存放浮点协处理器上下文 */
    )
```

功能描述：

该函数保存浮点协处理器上下文。在不同的体系结构中，保存的上下文是不一样的。

MC680x0：保存寄存器 fpcr、fpsr、fpiar、f0～f7 以及内部状态帧的内容；

SPARC：保存寄存器 fsr、fpq 和 f0～f31 的内容；

i960：保存寄存器 fp0～fp3 的内容；

MIPS：保存寄存器 fpcsr 和 fp0～fp31 的内容；

i386/i486：保存控制字、状态字、备注字、IP 偏移量、CS 选择器、数据操作数偏移量、操作数偏移量以及寄存器 st0～st7 中的内容；

ARM：ARM 体系结构不支持浮点协处理器。

返回值：

无。

参考：

相关信息请参考 fppArchLib 库描述和 fppRestore()函数。

fppRestore ()

函数原型：

```
void fppRestore
    (
```

```
FP_CONTEXT * pFpContext        /* 存放恢复上下文的内容 */
)
```

功能描述：

该函数恢复浮点协处理器上下文。在不同的体系结构中，恢复的上下文是不一样的。

MC680x0：恢复寄存器 fpcr、fpsr、fpiar、f0～f7 以及内部状态帧的内容；

SPARC：恢复寄存器 fsr、fpq 和 f0～f31 的内容；

i960：恢复寄存器 fp0～fp3 的内容；

MIPS：恢复寄存器 fpcsr 和 fp0～fp31 的内容；

i386/i486：恢复控制字、状态字、备注字、IP 偏移量、CS 选择器、数据操作数偏移量、操作数偏移量以及寄存器 st0～st7 中的内容。

ARM：ARM 体系结构不支持浮点协处理器。

返回值：

无。

参考：

相关信息请参考 fppArchLib 库描述和 fppSave()函数。

fppProbe()

函数原型：

```
STATUS fppProbe (void)
```

功能描述：

该函数会探测系统是否支持浮点协处理器。该函数与体系结构相关。

MC680x0，SPARC，i386/i486：该函数向协处理器发送非法的操作代码并执行。如果该操作产生异常，fppProbe()函数将会返回 ERROR。该函数会在执行非法操作之前保存和恢复协处理器的寄存器。该函数仅在第一次调用的时候执行，获取到的结果将会保存在一个静态变量中，之后的调用直接返回该变量。

i960：该函数仅仅检查 VxWorks 在编译的时候是否使用了-DCPU=I960KB 参数。

NIPS：该函数试图读取 R-Series 状态寄存器并判断相应寄存器，检查浮点协处理器是否存在。该标志位必须在 BSP 中正确初始化。

ARM：该函数总是返回 ERROR。

返回值：

系统支持浮点协处理器则返回 OK，不存在浮点协处理器时返回 ERROR。

参考：

相关信息请参考 fppArchLib 库描述。

fppTaskRegsGet()

函数原型：

```
STATUS fppTaskRegsGet
    (
    int             task,           /* 任务 ID */
    FPREG_SET * pFpRegSet    /* 指向浮点寄存器的指针 */
    )
```

功能描述：

该函数将指定任务的浮点寄存器内容以及状态寄存器保存到参数 pFpRegSet 所指向的结构中。浮点寄存器中的内容会被复制到包含所有寄存器内容的数组中。

注意：

该函数只能正确保存其他任务（非调用该函数的任务）的浮点寄存器。如果试图保存自身任务的浮点寄存器的值，将可能会保存一些错误数据。

返回值：

操作成功则返回 OK，在不支持浮点或状态无效时返回 ERROR。

参考：

相关信息请参考 fppArchLib 库描述及 fppTaskRegsSet()函数。

例：

```
int tid;                        /* 任务 ID */
FPREG_SET fpRegSet;             /* 浮点寄存器结构 */
BOOL    fppOk;                  /* FPU 支持标识 */
...
tid = taskIdDefault (tid);
if (fppOk = (fppProbe () == OK))
 {
        if (fppTaskRegsGet (tid, &fpRegSet) != OK)
    return (ERROR);
 }
...
```

fppTaskRegsSet ()

函数原型：

```
STATUS fppTaskRegsSet
    (
    int             task,           /* 任务 ID */
    FPREG_SET * pFpRegSet   /* 指向浮点寄存器的指针 */
    )
```

功能描述：

该函数用指定的值改写指定 TCB 中的浮点寄存器部分。

返回值：

设置成功则返回 OK，在不支持浮点或状态无效时返回 ERROR。

参考：

相关信息请参考 fppArchLib 库描述及 fppTaskRegsGet()函数。

10.3 浮点协处理器支持函数

10.3.1 函数库描述

1．库命名

浮点协处理器支持函数库名称为 fppLib。

2．函数

表 10-3 中列出了浮点协处理器支持函数。

3．描述

该函数库提供浮点处理器的支持。使用 fppInit()函数可以

表 10-3 浮点协处理器支持函数

函 数	描 述
fppInit()	初始化浮点协处理器支持

激活浮点协处理器并使任务支持浮点处理。如果用户在 config.h 中定义了 INCLUDE_HW_FP 宏，则系统会在 usrRoot()函数中自动执行该函数。

在初始化了浮点协处理器后可以使用 VX_FP_TASK 选项保存和恢复任务的浮点寄存器上下文。然而，并不是每个任务都需要保存和恢复浮点寄存器。只有使用了 VX_FP_TASK 选项，任务才会执行这项操作。

系统不会为使用 intConnect()连接的任何中断服务程序保存和恢复浮点寄存器，如果需要，用户可以使用 fppArchLib 库中的函数。

注意：

必须使用 VX_FP_TASK 选项生成进行浮点操作的任务。

4．头文件

浮点协处理器支持函数声明在 fppLib.h 头文件中。

5．参考

相关信息请参考 fppArchLib、fppShow 库描述。

10.3.2 浮点协处理器支持函数详细描述

fppInit()

函数原型：

void fppInit (void)

功能描述：

该函数初始化浮点协处理器支持，必须在使用浮点协处理器之前调用。如果用户在 config.h 中定义了 INCLUDE_HW_FP，则系统会在 usrRoot()函数中自动执行该函数。

返回值：

无。

参考：

相关信息请参考 fppLib 库描述。

10.4 浮点显示函数

10.4.1 函数库描述

1．库命名

浮点显示函数库命名为 fppShow。

2．函数

表 10-4 中列出了浮点显示函数。

表 10-4 浮点显示函数

函 数	描 述
fppShowInit()	初始化浮点显示模块
fppTaskRegsShow()	打印指定任务的浮点寄存器内容

3．描述

该函数库提供了显示任务浮点上下文的函数。在使用这个模块之前，首先应调用 fppShowInit()函数做初始化。该过程是用户配置浮点显示时系统自动完成的。用户只需要在 config.h 头文件中配置 INCLUDE_SHOW_ROUTINES 或者使用 Tornado 工程工具配置 INCLUDE_HW_FP_SHOW 即可。该函数库会增强信息显示函数的功能，例如在使用 ti()函数时可以显示浮点上下文。

4．头文件

任务钩子管理函数声明在 fppLib.h 头文件中。

5．参考

相关信息请参考 fppShow、fppLib 库描述。

10.4.2 浮点显示函数详细描述

fppShowInit()

函数原型：

void fppShowInit (void)

功能描述：

该函数将浮点信息显示模块连接到 VxWorks 系统库。该过程是用户配置浮点显示时系统自动完成的。用户只需要在 config.h 头文件中配置 INCLUDE_SHOW_ROUTINES 或者使用 Tornado 工程工具配置 INCLUDE_HW_FP_SHOW 即可。该函数库会增强信息显示函数的功能，例如在使用 ti()函数时可以显示浮点上下文。

返回值：

无。

参考：

相关信息请参考 fppShow 库描述。

fppTaskRegsShow ()

函数原型：

```
void fppTaskRegsShow
    (
    int task/* 任务 ID */
    )
```

功能描述：

该函数将指定任务的浮点寄存器内容输出到标准输出设备上。

返回值：

无。

参考：

相关信息请参考 fppShow 库描述。

操作示范：

在宿主机 WindSh 工具中，fppTaskRegsShow()函数操作示范如下：

```
-> fppTaskRegsShow(tWdbTask)
fpcr   = 1                fpsr   = b90dfcd8          fptag= a00026a1
st0    = 1.82033e-307     st1    = 3.16202e-322      st2    = 5.92879e-323
st3    = 1.18221e-308     st4    = -7.21931e-34      st5    = 5.72375e-302
st6    = -7.2189e-34      st7    = 1.07396e+250
```

第 11 章　VxWorks 网络系统

11.1　常用 BSD 套接字(socket)函数

11.1.1　常用 BSD 套接字函数库

1．库命名

常用 BSD 套接字函数库名称为 sockLib。

2．函数

表 11-1 中列出了常用 BSD 套接字函数。

表 11-1　常用 BSD 套接字函数

函　数	描　述	函　数	描　述
socket()	打开套接字	recvfrom()	从套接字接收消息
bind()	给套接字分配名字	recv()	从套接字接收数据
listen()	允许连接套接字	recvmsg()	从套接字接收消息
accept()	在套接字上接受连接	setsockopt()	设置套接字选项
connect()	请求连接套接字	getsockopt()	获得套接字选项
connectWithTimeout()	在指定时间内试图连接套接字	getsockname()	获得套接字名
sendto()	向套接字发送消息	getpeername()	获得远程连接的套接字名
send()	向套接字发送数据	shutdown()	关闭网络连接
sendmsg()	向套接字发送消息		

3．描述

sockLib 库提供 UNIX BSD 4.4 兼容的套接字调用。使用这些调用可以打开、关闭、读和写套接字。使用套接字，可以跨接背板、跨接以太网、或者跨接任何连接网络的联合体，也可以处理单个 CPU 内部的通信。在任何联合体的 VxWorks 任务和主机系统处理之间，都可以使用套接字通信。这些函数的调用次序与在 UNIX BSD 4.4 下的调用次序是一样的。

套接字通信的一个最大的优势就是它是一个统一机制：无论是网络中的局部处理或它们运行的是何种操作系统，套接字通信在处理中都是一样严密和正确。

套接字可以根据通信性质分类。应用程序一般仅在同一类的套接口间通信。不过只要底层的通信协议允许，不同类型的套接口间也照样可以通信。用户目前使用两种套接口，即流套接字和数据报套接字。

流套接字采用 TCP 协议与端口约定。一旦两个 TCP 套接字建立连接后，它们之间形成了一个虚拟回路。提供了双向的、有序的、无重复并且无记录边界的数据流服务。

数据报套接字采用 UDP 协议与端口约定。支持双向数据流，但套接字之间是独立的，并不需要建立连接。与 TCP 比较，UDP 像邮件一样，不保证数据包是可靠、有序、无重复的。也就是说，一个从数据报套接字接收信息的任务有可能发现信息重复了，或者和发出时的顺序不同。数据报套接字的一个重要特点是它保留了记录边界。对于这一特点，数据报套接字采用了与现在许多包交换网络（例如以太网）非常类似的模型。

4．地址族

VxWorks 套接字仅支持 Internet 域地址族。在需要 domain 参数的函数中，使用 AF_INET 作为函数参数值。但是，VxWorks 套接字不支持 UNIX 域地址族。

5．ioctl()函数

套接字响应 iotctl()函数的下面功能：

FIONBIO：打开/关闭非块 I/O，例如：

```
On = TRUE;
status = ioctl(sFd, FIONBIO, &on);
```

FIONREAD：报告套接字中现有待读字节个数。在 ioctl()返回中，参数 bytesAvailable 存放套接字中现有待读字节个数。例如：

```
status = ioctl(sFd, FIONREAD, &bytesAvalibale);
```

SIOCATMARK：报告是否有从套接字中禁止读取的(out-of-band)数据。在 ioctl()返回中，如果有禁止的数据则 atMark 为 TRUE(1)。其他则为 FALSE(0)。例如：

```
status = ioctl(sFd, SIOCATMARK, &atMark);
```

6．头文件

常用 BSD 套接字函数、定义等请参阅 types.h、mbuf.h、socket.h、socketVar.h 头文件。

7．参考

相关信息请参考 netLib 库描述，以及“VxWorks Programmer's Guide”中的网络系统章节。

11.1.2　常用 BSD 套接字函数详细描述

socket()

函数原型：

```
int socket
(
int domain,        /* 地址族 (例如, AF_INET) */
int type,          /* 套接字类型（SOCK_STREAM, SOCK_DGRAM 或 SOCK_RAW）*/
int protocol       /* 套接字协议(通常为 0) */
)
```

功能描述：

打开套接字。该函数打开套接字并返回套接字描述符。这个套接字描述符将被传递给其他套接字函数，以便识别这个套接字。套接字描述符是一个标准的 I/O 系统文件描述符(fd，file descriptor)，可以被 close()、read()、write()和 ioctl()函数使用。

可用的套接字类型包括：

SOCK_STREAM：流套接字，采用 TCP 协议与端口约定；

SOCK_DGRAM：数据报套接字，采用 UDP 协议与端口约定；

SOCK_RAW：原始(raw)协议套接字。

返回值：

成功打开则返回一个套接字描述符，如果失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述。

例：

```
int         sock;                   /* socket 文件描述符 */
...
```

```
/* 打开一个 TCP socket */
sock = socket (AF_INET, SOCK_STREAM, 0);
if (sock < 0)
 {
 perror ("Error in opening a TCP socket!");
 return (ERROR);
 }
...
```

bind()

函数原型：

```
STATUS bind
(
int s,                          /* 套接字描述符 */
struct sockaddr * name,         /* 分配给套接字的名字 */
int namelen                     /* 名字长度 */
)
```

功能描述：

给套接字分配名字。该函数分配一个网络地址(也称为“名字”)给指定的套接字，以便其他处理可以连接它或给它发送数据。当使用 socket()创建一个套接字时，它属于一个地址族但并没有分配名字。

返回值：

成功操作则返回 OK，如果套接字无效、地址无效、地址在使用、或套接字已经被捆绑则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述。

例：

```
struct sockaddr_in serverAddr;      /* 服务器的地址 */
int echoServerPort = 7001;          /* 默认端口号 */
int         sock;                   /* socket 文件描述符 */
...
/* 给套接字分配一个 internet 地址，以便客户端可以连接 */
    serverAddr.sin_family           = AF_INET;
    serverAddr.sin_port             = htons (echoServerPort);
    serverAddr.sin_addr.s_addr      = INADDR_ANY;
...
    printf ("\nBinding SERVER :%x\n", serverAddr.sin_port);
    if (bind (sock,(struct sockaddr *) &serverAddr, sizeof (serverAddr)) < 0)
 {
 printf ("Error in binding a TCP socket: %d\n",sock);
 close (sock);
 return (ERROR);
 }
...
```

listen()

函数原型：

```
STATUS listen
(
int s,                          /* 套接字描述符 */
int backlog                     /* 连接次数 */
)
```

功能描述：

允许连接套接字。该函数指定最大的被拒绝连接次数。当用 listen()允许连接后，通过 accept()函数接受连接。

返回值：

成功连接则返回 OK，如果套接字无效或连接失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述。

例：

```
int             sock;                   /* socket 文件描述符 */
...
/* 允许客户端连接服务器. */
printf ("Listening to client！ \n");
if (listen (sock, 5) < 0)
    {
 printf ("Error in Listening a TCP socket: %d\n",sock);
 close (sock);
 return (ERROR);
 }
...
```

accept()

函数原型：

```
int accept
(
int s,                          /* 套接字描述符 */
struct sockaddr * addr,         /* 连接地址 */
int * addrlen                   /* 连接地址的长度 */
)
```

功能描述：

在套接字上接受连接。该函数接受一个连接，并返回为连接而新建的一个套接字。这个套接字必须用 bind()捆绑一个地址，通过调用 listen()允许建立连接。accept()函数用与 s 一样的道具创建一个新的套接字。它阻塞调用者直到出现连接，除非套接字标识为非阻塞方式。

返回值：

成功接受则返回一个套接字描述符，如果这个调用失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述。

例：

```
struct sockaddr_in  newConnAddr;        /* 客户端的地址 */
```

```
int         sock;                   /* socket 文件描述符 */
int         len;                    /* 地址长度 */
int         newConnection;          /* socket 文件描述符 */
...
/* 已经与客户端连接上，则接受连接。*/
len = sizeof (newConnAddr);
newConnection = accept (sock, (struct sockaddr  *) &newConnAddr, &len);
if (newConnection == ERROR)
    {
      printf ("Error in accepting a TCP socket: %d\n", sock);
      close (sock);
      return (ERROR);
     }
...
```

connect()

函数原型：

```
STATUS connect
(
int s,                          /* 套接字描述符 */
struct sockaddr * name,         /* 与 s 连接的套接字地址 */
int namelen                     /* 套接字地址长度(字节) */
)
```

功能描述：

请求连接套接字。如果参数 s 是一个 SOCK_STREAM 类型的套接字，该函数在 s 与另一个套接字 name 之间建立一个虚拟线路。如果 s 是一个 SOCK_DGRAM 类型的套接字，套接字之间是独立的，并不需要建立连接。

返回值：

成功连接则返回 OK，如果连接失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述。

例：

```
struct sockaddr_in  serverAddr;             /* 服务器的地址 */
int         ClientSock;                     /* socket 文件描述符 */
int       Port = 7001;                      /* 默认端口号 */
char     *server;                           /* 服务器的 internet 地址 */
char    defaultServer[16] = "147.11.184.2"; /* 服务器的默认 internet 地址 */
...
bzero ((char  *) &serverAddr, sizeof (serverAddr));
server = defaultServer;
serverAddr.sin_family        = AF_INET;
serverAddr.sin_port          = htons (port);
serverAddr.sin_addr.s_addr   = inet_addr (server);
...
```

```
printf ("Server's address is %x:\n", ntohl (serverAddr.sin_addr.s_addr));
if (connect (echoClientSock, (struct sockaddr  *) &serverAddr, sizeof (serverAddr)) < 0)
  {
  perror ("echoTcpClientSock: connect failed");
  close (echoClientSock);
  return (ERROR);
  }
printf ("Connected...\n");
...
```

connectWithTimeout()

函数原型：

```
STATUS connectWithTimeout
(
int sock,                       /* 套接字 */
struct sockaddr * adrs,         /* 与 sock 连接的套接字地址 */
int adrsLen,                    /* 套接字地址长度(字节) */
struct timeval * timeVal        /* 超时值 */
)
```

功能描述：

在指定时间内尝试连接套接字。除在用户指定时间内建立连接外，该函数基本上与 connect()相同。

如果参数 timeVal 为 NULL 指针，该函数实际上就等同与 connect()。如果参数 timeVal 非 NULL，则试图在指定工作期间内建立连接。在这个时间之后，如果连接失败则该函数报告一个超时错误。

返回值：

成功连接则返回 OK，如果连接失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述，以及 connect()函数。

sendto()

函数原型：

```
int sendto
(
int s,                  /* 发送消息的套接字 */
caddr_t buf,            /* 发送数据缓冲区 */
int bufLen,             /* 缓冲区长度 */
int flags,              /* 协议标识 */
struct sockaddr * to,   /* 接收者的地址 */
int tolen               /* 地址长度 */
)
```

功能描述：

向套接字发送消息。该函数发送消息给数据报套接字 to。套接字 s 是作为发送消息的套接字，且接收者接收来自于套接字 s 上的消息。

参数 buf 的最大长度受 UDP 缓冲区尺寸的限制；有关这个问题请参考 setsockopt()中的 SO_SNDBUF 选项。

参数 flags 的值可以是下列选项的组合：

MSG_OOB(0x1)：处理 out-of-band 数据；

MSG_DONTROUTE(0x4)：发送时不使用路由表。

返回值：

发送成功则返回发送字节数，如果发送失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述，以及 setsockopt()函数。

例：

```
#define BROADCAST_ADDR "147.11.184.255"          /* 广播地址 */
int sockFd;                                      /* socket 文件描述符 */
struct sockaddr_in sendToAddr;                   /* 接收者的地址 */
char buf[] = "Hello, world!!!\n";                /* 广播消息 */
int     port = 7001;                             /* 默认端口号 */
int    sendNum ;
...
sendToAddr.sin_family = AF_INET;
sendToAddr.sin_port = htons (port);
sendToAddr.sin_addr.s_addr = inet_addr (BROADCAST_ADDR);
...
/* 发送广播消息 */
sendNum = sendto (sockFd, buf, sizeof (buf), 0,(struct sockaddr *) &sendToAddr,
          sizeof (struct sockaddr_in));
if ((sendNum) == ERROR)
    {
      printf("sendto broadcast failed!\n");
      return (ERROR);
    }
printf ("%d bytes of broadcast message sent: %s\n", sendNum,
          buf);
...
```

send()

函数原型：

```
int send
(
int s,              /* 发送数据的套接字 */
char * buf,         /* 发送数据缓冲区 */
int bufLen,         /* 缓冲区长度 */
int flags           /* 协议标识 */
)
```

功能描述：

向套接字发送数据。该函数发送数据给一个先前建立连接的流套接字。

参数 buf 的最大长度受 TCP 缓冲区尺寸的限制；有关这个问题请参考 setsockopt()中的 SO_SNDBUF 选项。

参数 flags 的值可以是下列选项的组合：

MSG_OOB(0x1)：处理超出范围的(out-of-band)数据；

MSG_DONTROUTE(0x4)：发送时不使用路由表。

返回值：

发送字节数，如果发送失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述，以及 setsockopt()和 sendmsg()函数。

例：

```
int         newConnection;                  /* socket 文件描述符 */
int         numSend;                        /* 发送字节数 */
char        buffer [] = "Hello, world!!!\n";    /* 数据缓冲区 */
...
/* 发送数据 */
numSend = send (newConnection, buffer, sizeof (buffer), 0);
if ((numSend) == ERROR)
    {
      printf("send data failed!\n");
      return (ERROR);
 }
...
```

sendmsg()

函数原型：

```
int sendmsg
(
int sd,                     /* 发送消息的套接字 */
struct msghdr * mp,         /* scatter-gather 消息头 */
int flags                   /* 协议标识 */
)
```

功能描述：

向套接字发送消息。该函数发送一个消息给数据报套接字。它可以代替 sendto()，对每个消息而言，它可以减少重建消息头结构(msghdr)的头部。

返回值：

发送字节数，如果发送失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述，以及 sendto()函数。

recvfrom()

函数原型：

```
int recvfrom
(
int s,                  /* 从这个套接字中接收消息 */
char * buf,             /* 存放消息的数据缓冲区 */
int bufLen,             /* 缓冲区长度 */
int flags,              /* 协议标识 */
```

```
    struct sockaddr * from,  /* 发送者的地址 */
    int * pFromLen           /* 地址长度 */
    )
```

功能描述：

从套接字中接收消息。该函数从数据报套接字中接收一个消息，而不管套接字是否已经连接。如果 from 非 0，发送者的套接字地址存放在这里。pFromLen 的初始值是 from 缓冲区的长度。函数返回后，它的值是存储在 from 中实际地址大小。

参数 buf 的最大长度受 UDP 缓冲区尺寸的限制；有关这个问题请参考 setsockopt()中的 SO_RCVBUF 选项。

参数 flags 的值可以是下列选项的组合：

MSG_OOB(0x1)：处理超出范围的(out-of-band)数据；

MSG_PEEK(0x2)：返回数据，但不把它从套接字中删除。

返回值：

接收字节数，如果调用失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述，以及 setsockopt()函数。

例：

```
int          sockFd;                 /* socket 文件描述符*/
charrecbuf[1024];                    /* 存放消息的数据缓冲区 */
int          lenbuf;                 /* 缓冲区长度 */
int    clientAddrLength;             /* 客户端地址长度 */
struct  sockaddr_in  clientAddr;     /* 客户端地址 */
...
clientAddrLength = sizeof (clientAddr);
/* 接收消息 */
if (recvfrom (sockFd, recbuf, sizeof(recbuf), 0,
              (struct sockaddr *) &clientAddr,
              &clientAddrLength) < 0)
 {
 close (sockFd);
 printf("recvfrom failed!\n");
 return (ERROR);
 }
...
```

recv()

函数原型：

```
    int recv
    (
    int s,              /* 从这个套接字中接收数据 */
    char * buf,         /* 存放数据的缓冲区 */
    int bufLen,         /* 缓冲区长度 */
    int flags           /* 协议标识 */
    )
```

功能描述：

从套接字中接收数据。该函数从一个连接的流套接字中接收数据。

buf 的最大长度受 TCP 缓冲区尺寸的限制；有关这个问题请参考 setsockopt()中的 SO_RCVBUF 选项。

参数 flags 的值可以是下列选项的组合：

MSG_OOB(0x1)：处理超出范围的(out-of-band)数据；

MSG_PEEK(0x2)：返回数据，但不把它从套接字中删除。

返回值：

接收字节数，如果调用失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述，以及 setsockopt()函数。

例：

```
int         newConnection;      /* socket 文件描述符 */
char        buffer [1024];      /* 数据缓冲区 */
...
/* 接收数据 */
numRead = recv (newConnection, buffer, sizeof (buffer), 0);
printf ("Received data: %s\n", buffer);
if (numRead == ERROR)
  {
    printf("CLIENT recv error!\n");
    return(ERROR);
  }
  ...
```

recvmsg()

函数原型：

```
    int recvmsg
    (
    int sd,                     /* 从这个套接字中接收消息 */
    struct msghdr * mp,         /* scatter-gather 消息头 */
    int flags                   /* 协议标识 */
    )
```

功能描述：

从套接字接收消息。该函数从数据报套接字中接收一个消息。它可以代替 recvfrom()，对每个消息而言，它可以减少分解消息头结构(msghdr)的头部。

返回值：

接收字节数，如果调用失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述。

setsockopt()

函数原型：

```
    STATUS setsockopt
    (
    int s,              /* 套接字 */
```

```
    int level,          /* 选项协议级 */
    int optname,        /* 选项名 */
    char * optval,      /* 选项值 */
    int optlen          /* 选项长度 */
    )
```

功能描述：

设置套接字选项。该函数用分配的选项设置套接字。为了在“套接字”级处理选项，参数 level 应当是 SOL_SOCKET。其他级应该使用适当的协议数。在使用该函数时，用系统分配给参数 optval 的空间大小(sizeof(optval))来初始化参数 optlen。

尽管参数 optval 是作为一个字符传递给函数，但实际上是一个整数或结构的地址，且取决于参数 optname。表 11-2 列出了参数 level、optname、optval 和套接字之间的关系。

表 11-2 参数 level、optname、optval 和套接字之间的关系

level	Optname	Optval	套接字类型
SOL_SOCKET	SO_KEEPALIVE	1 或 0	流套接字
SOL_SOCKET	SO_LINGER	用户指定时间	流套接字
IPPROTO_TCP	TCP_NODELAY	1 或 0	流套接字
SOL_SOCKET	SO_DEBUG	1 或 0	流套接字
SOL_SOCKET	SO_BROADCAST	1 或 0	数据报套接字
IPPROTO_IP	IP_ADD_MEMBERSHIP	一个 ip_mreq 结构	数据报和 raw 套接字
IPPROTO_IP	IP_DROP_MEMBERSHIP	一个 ip_mreq 结构	数据报和 raw 套接字
IPPROTO_IP	IP_MULTICAST_IF	一个 in_addr 结构	数据报和 raw 套接字
IPPROTO_IP	IP_MULTICAST_TTL	0、1、31...(该值与应用有关)	数据报和 raw 套接字
IPPROTO_IP	IP_MULTICAST_LOOP	1 或 0	数据报和 raw 套接字
SOL_SOCKET	SO_REUSEADDR	1 或 0	数据报和流套接字
SOL_SOCKET	SO_SNDBUF	发送缓冲区大小	数据报和流套接字
SOL_SOCKET	SO_RCVBUF	接收缓冲区最大尺寸	数据报和流套接字
SOL_SOCKET	SO_OOBINLINE	—	数据报和流套接字

返回值：

成功设置则返回 OK，如果设置失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述。

例：

```
int         sock;       /* socket 文件描述符 */
int     optionVal;      /* socket 选项值 */
...
optionVal = 1;

/* 设置 socket 选项 */
if (setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, (char *) &optionVal,
                sizeof(optionVal)) == ERROR)
 {
```

```
perror ("echoTcpForever: setsockopt SO_REUSEADDR failed");
close (sock);
return (ERROR);
}
...
```

getsockopt()

函数原型：

```
STATUS getsockopt
(
int s,              /* 套接字 */
int level,          /* 选项协议级 */
int optname,        /* 选项名 */
char * optval,      /* 选项存放处 */
int * optlen        /* 选项长度存放处 */
)
```

功能描述：

获得套接字选项。该函数返回分配给套接字的选项。为了在“套接字”级处理选项，参数 level 应当是 SOL_SOCKET。其他级应该使用适当的协议数。在使用该函数时，用系统分配给参数 optval 的空间大小(sizeof(optval))来初始化参数 optlen。

尽管参数 optval 是作为一个字符传递给函数，但实际上是一个整数或结构的地址，且取决于参数 optname。

返回值：

成功获得套接字选项则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述，以及 setsockopt()函数。

例：

```
int          sock;      /* socket 文件描述符 */
charoptbuf[20];         /* 选项存放处 */
int    optlen           /* 选项长度存放处 */
...
optlen = sizeof (optbuf);
/* 获得套接字选项 */
if (getsockopt (sock, SOL_SOCKET,SO_RCVWAKEUP, optbuf, &optlen) == ERROR)
  {
 printf ("getsockopt failed: %s", strerror (errno));
 return (ERROR);
  }
 ...
```

getsockname()

函数原型：

```
STATUS getsockname
(
int s,                          /* 套接字描述符 */
struct sockaddr * name,         /* 名字存放处 */
```

```
    int * namelen                   /* name 所占空间大小 */
    )
```

功能描述：

获得套接字名字。该函数获得指定的套接字 s 当前名字。在使用该函数时，用系统分配给 name 的空间大小(sizeof(name))初始化参数 namelen 。

返回值：

成功获得套接字名则返回 OK，如果是无效的套接字或没有连接则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述。

例：

```
int             fd;         /* socket 文件描述符*/
STATUS   st;
struct sockaddr_in sockaddr;
int sockaddrLen = sizeof (sockaddr);
...
/* 获得套接字名字 */
st = getsockname (fd, (struct sockaddr *) &sockaddr, &sockaddrLen);
if (st == ERROR)
 {
 printf ("getsockname error (errno = %#x)\n", errnoGet ());
 return(ERROR);
 }
...
```

getpeername()

函数原型：

```
    STATUS getpeername
    (
    int s,                          /* 套接字描述符 */
    struct sockaddr * name,         /* 名字存放处 */
    int * namelen                   /* name 所占空间大小 */
    )
```

功能描述：

获得远程连接的套接字名字。该函数获得与指定的套接字 s 连接的远程套接字的名字。在使用该函数时，用系统分配给 name 的空间大小(sizeof(name))初始化参数 namelen。

返回值：

成功获得则返回 OK，如果是无效的套接字或没有连接则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述。

例：

```
int             fd;         /* socket 文件描述符*/
STATUS   st;
struct sockaddr_in      peer;
```

```
int peerlen = sizeof(peer);
...
/* 获得远程连接的套接字名字 */
st = getpeername(fd, (struct sockaddr_in *) &peer,&peerlen);
if (st == ERROR)
   {
 printf("getpeername error (errno = %#x)\n", errnoGet ());
 return (ERROR);
    }
...
shutdown()
函数原型：
      STATUS shutdown
      (
      int s,    /* 要关闭的套接字 */
      int how /* 0 = 禁止接收 */
              /* 1 = 禁止发送 */
              /* 2 = 禁止发送和接收 */
      )
```

功能描述：

关闭套接字。该函数关闭连接套接字 s 的所有或部分功能。

返回值：

成功关闭则返回 OK，如果是无效套接字或没有连接则返回 ERROR。

参考：

相关信息请参考 sockLib 库描述。

例：

```
int             socketFd;
STATUS   st;
...
/* 关闭套接字 */
st = shutdown(socketFd, 2);
if(st == ERROR)
   {
 printf("shutdown socket error (errno = %#x)\n", errnoGet ());
 return (ERROR);
    }
...
```

11.2　zbuf 套接字接口函数

11.2.1　函数库描述

1. 库命名

zbuf 套接字接口函数库名称为 zbufSockLib。

2．函数

表 11-3 中列出了 zbuf 套接字接口函数。

表 11-3 zbuf 套接字接口函数

函 数	描 述	函 数	描 述
zbufSockLibInit()	初始化 zbuf 套接字函数库	zbufSockBufSendto()	用用户消息创建 zbuf，并把它发送给 UDP 套接字
zbufSockSend()	向 TCP 套接字发送 zbuf 数据		
zbufSockSendto()	向 UDP 套接字发送 zbuf 消息	zbufSockRecv()	从 TCP 套接字中接收数据，并存放在一个 zbuf 中
zbufSockBufSend()	用用户数据创建 zbuf，并把它发送给 TCP 套接字	zbufSockRecvfrom()	从 UDP 套接字中接收消息，并存放在一个 zbuf 中

3．描述

zbufSockLib 库提供一组基于数据抽象的、可选的套接字接口函数，VxWorks 系统把它称为 zbuf 套接字接口。

zbuf 套接字接口允许应用程序，对 BSD 套接字进行读写操作，但在应用程序缓冲区和网络缓冲区之间并不进行数据复制。用户可以基于 TCP 或 UDP 协议使用 zbuf。新的 TCP 子集接口有时称为“zero-copy TCP”（零复制 TCP）。

zbuf 套接字的接口函数可以和标准 BSD 套接字接口相互通信：套接字的另一端没有办法告诉对方，它是使用 zbuf 调用还是使用传统的 BSD 套接字调用。

但是，zbuf 套接字调用，并不兼容标准 BSD 套接字接口。在使用 zbuf 接口时，必须调用不同的套接字接口函数。如果应用程序使用 zbuf 接口，将会降低应用程序的移植性。

使用 zbuf 套接字最简单的方法是调用 zbufSockBufSend()(类似流套接字的 send()) 函数，或者 zbufSockBufSendto()(类似数据报套接字 sendto())函数。在这两者中，需要提供指向应用程序数据缓冲区（该缓冲区包含要发送的数据或消息）的指针，网络协议也将使用这个缓冲区，这种通过指针的方式不需要进行数据复制，胜于从缓冲区复制数据。

与传统的 BSD 套接字相比较，采用 zbuf 套接字最大的优势就是，避免了在应用程序缓冲区和网络缓冲区之间进行数据复制。这样提高了网络通信性能。这种 zbuf 技术目前被广泛应用到一些交换机等设备中。

4．头文件

zbuf 套接字接口函数声明在 zbufSockLib.h 头文件中。

5．参考

相关信息请参考 zbufLib、sockLib 库描述，以及“VxWorks Programmer's Guide”中的网络部分。

11.2.2 zbuf 套接字接口函数详细描述

zbufSockLibInit()

函数原型：

```
STATUS zbufSockLibInit (void)
```

功能描述：

初始化 zbuf 套接字接口函数库。在使用 zbuf 套接字函数之前，必须调用该函数。当我们定义了配置宏 INCLUDE_ZBUF_SOCK（在 configAll.h 文件中定义）时，系统将自动调用该函数。

返回值：

成功初始化则返回 OK，如果初始化 zbuf 套接字接口函数库失败则返回 ERROR。

参考：

相关信息请参考 zbufSockLib 库描述。

例：

```
STATUS usrNetInit
    (
    char *bootString            /* 引导参数 */
    )
{
...
#ifdef   INCLUDE_ZBUF_SOCK
    zbufSockLibInit ();         /* 初始化 zbuf socket 接口 */
#endif   /* INCLUDE_ZBUF_SOCK */
...
}
```

zbufSockSend()

函数原型：

```
    int zbufSockSend
    (
    int s,                      /* 发送数据的套接字 */
    ZBUF_ID zbufId,             /* 要发送的 zbuf */
    int zbufLen,                /* 整个 zbuf 长度 */
    int flags                   /* 协议标识 */
    )
```

功能描述：

向 TCP 套接字发送 zbuf 数据。该函数把 zbufId 中的所有数据发送给先前建立连接的流套接字 s 。

参数 zbufLen 只是被用来决定套接字写缓冲区需要的空间大小，它并不能体现出实际发送字节数。如果用户不知道 zbufId 的长度，用户必须通过调用 zbufLength()函数来确定。

函数把 zbuf 的所有权从用户应用中递交到 VxWorks 网络堆栈中。调用该函数后，系统删除 zbufId 的 ID，且不能再使用这个 zbuf ID，即使该函数返回一个 ERROR 状态也不能再使用这个 zbuf ID。

参数 flags 的值可以是下列选项的组合：

MSG_OOB(0x1)：处理超出范围的(out-of-band)数据；

MSG_DONTROUTE(0x4)：发送时不使用路由表。

返回值：

发送字节数，如果调用失败则返回 ERROR。

参考：

相关信息请参考 zbufSockLib 库描述、zbufLength()、zbufSockBufSend()和 send()函数。

例：

```
int     sFd;                /* 套接字文件描述 */
ZBUF_ID zReplyOrig;         /* 原始应答消息 */
ZBUF_ID zReplyDup;          /* 应答消息副本 */
static char replyMsg[] = "The data from a TCP Server";
...
/* 创建一个基于 TCP 的套接字 */
```

```
if ((sFd = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
{
printf ("socket open failed!\n");
return (ERROR);
}
...
/* 把应答消息插入到 zbuf 中 */
zbufInsertBuf (zReplyOrig, NULL, 0, replyMsg,sizeof (replyMsg), NULL, 0);
...
/* 复制应答消息 zbuf, 保存原有消息 */
zReplyDup = zbufDup (zReplyOrig, NULL, 0, ZBUF_END));
 /* 发送数据 */
 if (zbufSockSend (sFd, zReplyDup, sizeof (replyMsg), 0) == ERROR)
    {
     printf ("zbufSockSend failed!\n");
     return(ERROR);
    }
...
```

zbufSockSendto()

函数原型：

```
int zbufSockSendto
(
int s,                  /* 发送消息的套接字 */
ZBUF_ID zbufId,         /* 要发送的 zbuf */
int zbufLen,            /* 整个 zbuf 长度 */
int flags,              /* 协议标识 */
struct sockaddr * to,   /* 接收者的地址 */
int tolen               /* 地址长度 */
)
```

功能描述：

向 UDP 套接字发送 zbuf 消息。该函数把 zbufId 中的所有消息发送给接收地址 to，且套接字 5 是发送套接字。

参数 zbufLen 只是被用来决定套接字写缓冲区需要的空间大小，它并不能体现出实际发送字节数。如果用户不知道 zbufId 的长度，用户必须通过调用 zbufLength()函数确定。

函数把 zbuf 的所有权从用户应用中递交到 VxWorks 网络堆栈中。调用该函数后，系统删除 zbufId 的 ID，且不能再使用这个 zbuf ID，即使该函数返回一个 ERROR 状态也不能再使用这个 zbuf ID。

参数 flags 的值可以是下列选项的组合：

MSG_OOB(0x1)：处理超出范围的(out-of-band)数据；

MSG_DONTROUTE(0x4)：发送时不使用路由表。

返回值：

发送字节数，如果调用失败则返回 ERROR。

参考：

相关信息请参考 zbufSockLib 库描述、zbufLength()、zbufSockBufSendto()和 sendto()函数。

例：

```
#define BROADCAST_ADDR "147.11.184.255"
int     sFd;                        /* 套接字文件描述 */
ZBUF_ID zReplyOrig;                 /* 原始应答消息 */
ZBUF_ID zReplyDup;                  /* 应答消息副本 */
static char replyMsg[] = "The message from a UDP Server ";
struct sockaddr_in sendToAddr;      /* 接收者的地址 */
int             port = 7001;
...
/* 创建一个基于 UDP 的套接字 */
if ((sFd = socket(AF_INET, SOCK_DGRAM, 0)) == ERROR)
{
printf ("socket open failed!\n");
return (ERROR);
}
...
/* 把应答消息插入到 zbuf 中 */
zbufInsertBuf (zReplyOrig, NULL, 0, replyMsg,sizeof (replyMsg), NULL, 0);
...
/* 复制应答消息 zbuf, 保存原有消息 */
zReplyDup = zbufDup (zReplyOrig, NULL, 0, ZBUF_END));
 sendToAddr.sin_family = AF_INET;
 sendToAddr.sin_port = htons (port);
 sendToAddr.sin_addr.s_addr = inet_addr (BROADCAST_ADDR);
 /* 发送消息 */
 if (zbufSockSend to (sFd, zReplyDup, sizeof (replyMsg), 0,
          (struct sockaddr    *) &sendToAddr,sizeof(sendToAddr)) < 0)
    {
     printf ("zbufSockSendto failed!\n");
     return(ERROR);
    }
...
zbufSockBufSend()
函数原型：
      int zbufSockBufSend
      (
      int s,                        /* 发送套接字 */
      char * buf,                   /* 数据缓冲区 */
      int bufLen,                   /* 发送字节数 */
      VOIDFUNCPTR freeRtn,          /* 释放函数*/
      int freeArg,                  /* 传递给释放函数的参数 */
      int flags                     /* 协议标识 */
      )
```

功能描述：

用用户数据创建 zbuf，并把它发送给 TCP 套接字。

当 TCP/IP 网络堆栈不再使用 buf 时，系统将调用用户提供的释放函数 freeRtn 。应用程序可以使用这个函数来接收释放 buf 的布告。如果 freeRtn 为 NULL，该函数正常运行，只是当网络堆栈释放 buf 时，系统没有办法通知应用程序。释放函数运行在任务的上下文中，下面是释放函数 freeRtn 的声明：

```
void freeCallback          /* 用户自定义的释放函数名 */
(
caddr_t buf,               /* 指向用户缓冲区的指针 */
int freeArg                /* 用户提供给释放函数的参数 */
)
```

参数 flags 的值可以是下列选项的组合：

MSG_OOB(0x1)：处理超出范围的(out-of-band)数据；

MSG_DONTROUTE(0x4)：发送时不使用路由表。

返回值：

发送字节数，如果调用失败则返回 ERROR。

参考：

相关信息请参考 zbufSockLib 库描述，以及 zbufSockSend()和 send()函数。

例：

```
int     sFd;              /* 套接字文件描述 */
static char replyMsg[] = "The message from a UDP Server ";
...
/* 发送数据 */
if(zbufSockBufSend(sFd,replyMsg,sizeof(replyMsg),NULL,0, MSG_DONTROUTE) < 0)
    {
     printf ("zbufSockBufSend failed!\n");
     return(ERROR);
    }
...
```

zbufSockBufSendto()

函数原型：

```
int zbufSockBufSendto
(
int s,                     /* 发送套接字 */
char * buf,                /* 数据缓冲区 */
int bufLen,                /* 发送字节数 */
VOIDFUNCPTR freeRtn,       /* 释放函数 */
int freeArg,               /* 传递给释放函数的参数 */
int flags,                 /* 协议标识 */
struct sockaddr * to,      /* 接收者的地址 */
int tolen                  /* 地址长度 */
)
```

功能描述：

用用户消息创建 zbuf，并把它发送给 UDP 套接字。

当 TCP/IP 网络堆栈不再使用 buf 时，系统将调用用户提供的释放函数 freeRtn 。应用程序可以使用这个函数来接收释放 buf 的布告。如果 freeRtn 为 NULL，该函数正常运行，只是当网络堆栈释放 buf 时，系统没有办法通知应用程序。释放函数运行在任务的上下文中，下面是 freeRtn 的声明：

```
void freeCallback
(
caddr_t buf,      /* 指向用户缓冲区的指针 */
int freeArg       /* 用户提供给释放函数的参数 */
)
```

参数 flags 的值可以是下列选项的组合：

MSG_OOB(0x1)：处理超出范围的(out-of-band)数据；

MSG_DONTROUTE(0x4)：发送时不使用路由表。

返回值：

发送字节数，如果调用失败则返回 ERROR。

参考：

相关信息请参考 zbufSockLib 库描述，以及 zbufSockSendto()和 sendto()函数。

zbufSockRecv()

函数原型：

```
ZBUF_ID zbufSockRecv
(
int s,            /* 从这个套接字中接收数据 */
int flags,        /* 协议标识 */
int * pLen        /* 请求接收字节数 */
)
```

功能描述：

从 TCP 套接字中接收数据，并把数据存放在一个新建的 zbuf 中。

参数 pLen 指出调用者请求接收字节的个数。如果操作成功，接收到的字节数将被复制到 plen 中。

参数 flags 的值可以是下列选项的组合：

MSG_OOB(0x1)：处理超出范围的(out-of-band)数据；

MSG_PEEK(0x2)：返回数据，但并不把它从套接字中删除。

一旦用户应用程序与 zbuf 断绝关系，应用程序应当调用 zbufDelete()把 zbuf 内存回收给 VxWorks 网络堆栈。

返回值：

成功接收则新建的 zbuf ID 包含接收数据，如果操作失败则为 NULL。

参考：

相关信息请参考 zbufSockLib 库描述，以及 recv()函数。

例：

```
int     sFd;            /* 套接字文件描述 */
int     nbytes;         /* 接收字节数 */
ZBUF_ID zRecv;          /* 读取套接字的 zbuf */
...
nbytes = 10;
/* 接收数据 */
zRecv = zbufSockRecv (sFd, 0, &nbytes));
```

```
if(zRecv == NULL)
    {
     printf ("zbufSockRecv failed!\n");
     return(ERROR);
    }
...
```

zbufSockRecvfrom()

函数原型：

```
ZBUF_ID zbufSockRecvfrom
(
int s,                          /* 从这个套接字中接收消息 */
int flags,                      /* 协议标识 */
int * pLen,                     /* 请求接收字节数 */
struct sockaddr * from,         /* 发送者的地址 */
int * pFromLen                  /* 地址长度 */
)
```

功能描述：

从 UDP 套接字中接收数据，并把数据存放在一个新建的 zbuf 中。

当接收消息时，系统不管套接字是否处于连接状态。如果 from 非 0，发送者的地址存放在这里。用 from 缓冲区的大小初始化参数 pFromLen 。函数一旦返回，pFromLen 包含 from 的实际大小。

参数 pLen 指出调用者请求接收字节数。如果操作成功，将把接收到的字节数复制到 plen 中。

参数 flags 的值可以是下列选项的组合：

MSG_OOB(0x1)：处理超出范围的(out-of-band)数据；

MSG_PEEK(0x2)：返回数据，但并不把它从套接字中删除。

一旦用户应用程序与 zbuf 断绝关系，应用程序应当调用 zbufDelete()把 zbuf 内存回收给 VxWorks 网络堆栈。

返回值：

成功接收则新建的 zbuf ID 中包含接收消息，如果操作失败则为 NULL。

参考：

相关信息请参考 zbufSockLib 库描述，以及 recvfrom()函数。

11.3 zbuf 接口函数

11.3.1 函数库描述

1．库命名

zbuf 接口函数库名称为 zbufLib。

2．函数

表 11-4 中列出了 zbuf 接口函数。

3．描述

zbufLib 库包含创建、构造、处理和删除 zbufs 的函数。Zbufs，也称为“Zero copy buffers”，它允许用户通过共享缓冲区来提取数据而不必要复制数据。同时，为了支持数据提取操作，该函数库中的子程序隐藏了 zbuf 的详细执行过程。

表 11-4　zbuf 接口函数

函　数	描　述	函　数	描　述
zbufCreate()	创建空的 zbuf	zbufSplit()	把一个 zbuf 分裂成两个独立的 zbuf
zbufDelete()	删除 zbuf	zbufDup()	复制 zbuf
zbufInsert()	把 zbuf 插入到另一个 zbuf 中	zbufLength()	确定 zbuf 的长度
zbufinsertBuf()	基于缓冲区创建 zbuf 段，并把它插入到 zbuf 中	zbufSegFind()	查找 zbuf 段
		zbufSegNext()	在一个 zbuf 中获得下一个 zbuf 段
zbufInsertCopy()	把缓冲区中的数据复制到 zbuf 中	zbufSegPrev()	在一个 zbuf 中获得前一个 zbuf 段
zbufExtractCopy()	把 zbuf 中的数据复制到缓冲区中	zbufSegData()	确定 zbuf 段中数据的位置
zbufCut()	在 zbuf 中删除字节	zbufSegLength()	确定 zbuf 段的长度

Zbufs 有三个基本属性：第一，zbuf 保持字节排序。第二，组织这些字节成一个或多个连接的数据段。第三，段中的数据可能与其他段共享；也就是说，在某时，可能有多个 zbuf 使用这个数据。

4．zbuf 类型

下面是 zbufs 的数据类型：

ZBUF_ID：一个任意的(但是是惟一的)整数，用来识别一个指定的 zbuf；

ZBUF_SEG：一个任意的(但是是惟一的)整数，用来识别 zbuf 段。

5．寻址字节(addressing byte)

这个字节是通过段(zbufSeg)和偏移量(offset)作为在 zbuf 中的寻址地址。偏移量可能是正数或负数，是从段开始的字节数。

一个段可以指定为 NULL，用来作为一个 zbuf 的起始段。如果段为 NULL，偏移量相对 zbuf 中任何字节而言是绝对偏移量。然而，相对于段而言，包含偏移量来确定 zbuf 字节位置显得更有效。zbufSegFind()函数转换任何段和偏移量到最有效等量。

偏移量为负值总是指的是相应段之前的字节，在它们自己内部通常不是最有效的地址表达。

下面是偏移量的特殊值，并且是一个常数，允许用户指定整个 zbuf 的起始或结尾处，且不管 zbuf 段的值：

ZBUF_BEGIN：整个 zbuf 的起始处；

ZBUF_END：整个 zbuf 的结尾处。

6．插入和限制偏移量

如果偏移量超出 zbuf，则偏移量无效。在一个 N 个字节的 zbuf 中，当前地址数据正确的偏移量相对于第一个段是从 0 到 N-1。

插入函数是一种特殊情况：插入函数遵循通用的惯例，但是在插入结束后要使用偏移量来指定新数据的开始处。由于关系到最初的 zbuf 数据，因此数据总是恰好插入在偏移量值指定的字节位置之前。这种惯例允许在现有的数据之后或之前插入数据。为了在一个 zbuf 段中当前所有的数据之前插入数据，则使用 0 作为偏移量。为了在一个包含 N 个字节的 zbuf 段中所有的数据之后插入数据，则使用 N 作为偏移量。当偏移量为 N-1，则恰好在一个包含 N 个字节的 zbuf 段的最后字节之前插入数据。

偏移量为 0 则表明总是一个有效的插入点；对于一个空 zbuf，0 为惟一的有效偏移量。

7．共享数据

zbufLib 库中的函数无论什么时候只要可能就避免复制数据。这样，系统宁可传递和处理 ZBUF_ID 而不愿复制数据，从而使得多个程序可以更有效地进行通信。但是，每一个程序必须意识到数据是共享的：改变 zbuf 段中的数据意味着所有涉及到这个数据的 zbuf 段都是看得见的。

为了改变用户自己程序的 zbuf 数据而不影响其他程序，首先使用 zbufDup()函数创建一个新的 zbuf；接

着用户可以使用插入或删除函数，例如 zbufInsertBuf()函数，只是在用户自己的程序中添加一个段。一种安全的做法是在添加数据到 zbuf 中之前直接在一个私有缓冲区中处理数据。

一旦把数据缓冲区添加到 zbuf 段中，当任何程序不再使用这个缓冲区时 zbufLib 库有责任汇报这个情况。为了支持这种现象，zbufInSertBuf()函数需要用户指定一个释放函数释放每次在现有的缓冲区上创建的 zbuf 段。当不再使用数据缓冲区时，用户可以使用这个释放函数来通知应用程序。

8. 头文件

zbuf 接口函数声明在 zbufLib.h 头文件中。

9. 参考

相关信息请参考 zbufLib 库描述，以及“VxWorks Programmer's Guide”中的网络部分。

11.3.2 zbuf 接口函数详细描述

zbufCreate()

函数原型：

```
ZBUF_ID zbufCreate (void)
```

功能描述：

创建空的 zbuf。该函数创建一个 zbuf，它首先保持空(没有数据)，直到调用 zbuf 插入函数把段添加到这个 zbuf 中。操作完成后返回一个 zbuf ID。

返回值：

zbuf ID，如果创建失败则返回 NULL。

参考：

相关信息请参考 zbufLib 库描述，以及 zbufDelete()函数。

zbufDelete()

函数原型：

```
STATUS zbufDelete
(
ZBUF_ID zbufId        /* 要删除的 zbuf */
)
```

功能描述：

删除 zbuf。该函数删除指定的 zbuf 中所有的 zbuf 段，然后删除 zbuf ID 自身。在这个函数成功执行后，用户再也不可以使用这个 zbufId 了。

对于任何其他的 zbuf 不再使用的数据缓冲区，函数 zbufDelete()将会调用关联的释放函数向系统报告。

返回值：

成功删除则返回 OK，如果删除失败则返回 ERROR。

参考：

相关信息请参考 zbufLib 库描述,zbufCreate()、zbufInsertBuf()函数。

zbufInsert()

函数原型：

```
ZBUF_SEG zbufInsert
(
ZBUF_ID zbufid1,          /* 把 zbufId2 插入到这个 zbuf 中 */
ZBUF_SEG zbufSeg,         /* 被 offset 使用的 zbuf 段偏移基地址 */
int offset,               /* 相对字节偏移量 */
```

```
    ZBUF_ID zbufId2          /* 把这个 zbuf 插入到 zbufId1 中 */
    )
```

功能描述：

把一个 zbuf 插入到另一个 zbuf 中。该函数把 zbufId2 中所有的 zbuf 段插入到 zbufId1 中所指定的字节位置处。

通过 zbufSeg 和 offset 指定插入位置。有关更多的信息请参考 zbufLib 库描述。值得注意的是，当插入到一个空 zbuf 中时，zbufSeg 和 offset 分别是 NULL 和 0。

在 zbufId2 中所有的 zbuf 段插入到 zbufId1 后，系统会删除 zbufId2。

返回值：

第一个被插入的 zbuf 段，如果操作失败则返回 NULL。

参考：

相关信息请参考 zbufLib 库描述。

例：

```
ZBUF_ID zRecvTotal = NULL;          /* 要返回的 zbuf */
ZBUF_ID zRecv;                      /* 读取套接字的 zbuf */
...
/* 插入 zbuf */
if (zbufInsert (zRecvTotal, NULL, ZBUF_END, zRecv) == NULL)
{
printf ("Insert zbuf failed!\n");
zbufDelete (zRecv);
zbufDelete (zRecvTotal);
return (NULL);
}
...
```

zbufInsertBuf()

函数原型：

```
    ZBUF_SEG zbufInsertBuf
    (
    ZBUF_ID zbufId,          /* 把缓冲区插入到这个 zbuf 中 */
    ZBUF_SEG zbufSeg,        /* 被 offset 使用的 zbuf 段偏移基地址*/
    int offset,              /* 相对的字节偏移量 */
    caddr_t buf,             /* 应用缓冲区 */
    int len,                 /* 插入字节数 */
    VOIDFUNCPTR freeRtn,     /* 释放函数 */
    int freeArg              /* 传递给释放函数的参数 */
    )
```

功能描述：

基于缓冲区创建 zbuf 段，并把它插入到 zbuf 中。

通过 zbufSeg 和 offset 指定插入位置。有关更多的信息请参考 zbufLib 库描述。值得注意的是，当插入到一个空 zbuf 中时，zbufSeg 和 offset 分别是 NULL 和 0。

参数 freeRtn 指定释放函数，当任何 zbuf 段不再使用 buf 时，系统将调用用户提供的释放函数。应用程

序可以使用这个函数来接收释放 buf 的布告。如果 freeRtn 为 NULL，该函数正常运行，只是当网络堆栈释放 buf 时，系统没有办法通知应用程序。释放函数运行在任务的上下文中，下面是释放函数 freeRtn 的声明：

```
void freeCallback
(
caddr_t buf,      /* 指向用户缓冲区的指针 */
int freeArg       /* 用户提供给释放函数的参数 */
)
```

返回值：

插入的 zbuf 段 ID，如果操作失败则返回 NULL。

参考：

相关信息请参考 zbufLib 库描述。

例：

```
ZBUF_ID zReplyOrig;              /* 原始应答消息 */
static char replyMsg[] = "Server received your message";
...
/* 创建原始应答消息 zbuf */
if ((zReplyOrig = zbufCreate ()) == NULL)
{
printf ("zbuf create failed!\n");
return(ERROR);
}
/* 把应答消息插入到 zbuf 中 */
if (zbufInsertBuf (zReplyOrig, NULL, 0, replyMsg,
sizeof (replyMsg), NULL, 0) == NULL)
{
printf ("zbuf insert failed!\n");
zbufDelete (zReplyOrig);
return(ERROR);
}
...
```

zbufInsertCopy()

函数原型：

```
ZBUF_SEG zbufInsertCopy
(
ZBUF_ID zbufId,          /* 把数据复制到这个 zbuf 中 */
ZBUF_SEG zbufSeg,        /* 被 offset 使用的 zbuf 段偏移基地址*/
int offset,              /* 相对的字节偏移量 */
caddr_t buf,             /* 把这个缓冲区中的数据复制到 zbufId 中 */
int len                  /* 复制字节数 */
)
```

功能描述：

把缓冲区中的数据复制到 zbuf 中。该函数从用户缓冲区 buf 中复制 len 个字节数据到 zbufId 中指定的字节位置处。

通过 zbufSeg 和 offset 指定插入位置。有关更多的信息请参考 zbufLib 库描述。值得注意的是，当插入到一个空 zbuf 中时，zbufSeg 和 offset 分别是 NULL 和 0。

返回值：

第一个插入到 zbuf 中的 zbuf 段 ID，如果操作失败则返回 NULL。

参考：

相关信息请参考 zbufLib 库描述。

zbufExtractCopy()

函数原型：

```
int zbufExtractCopy
(
ZBUF_ID zbufId,          /* 把这个 zbuf 中的数据复制到 buf 中 */
ZBUF_SEG zbufSeg,        /* 被 offset 使用的 zbuf 段偏移基地址*/
int offset,              /* 相对的字节偏移量 */
caddr_t buf,             /* 把 zbuf 中的数据复制到这个缓冲区中 */
int len                  /* 复制字节数 */
)
```

功能描述：

把 zbuf 中的数据复制到缓冲区中。该函数从 zbufId 中复制 len 个字节数据到应用缓冲区 buf 中。

通过 zbufSeg 和 offset 指定复制起始位置。有关更多的信息请参考 zbufLib 库描述。值得注意的是，第一个被复制的字节就是 zbufSeg 和 offset 所指定的字节。

复制字节数由参数 len 确定。如果这个参数是负数或大于 zbuf 中指定字节位置后面的字节数，zbuf 的其余数据会被复制，并可能跨度多个段。

返回值：

复制到缓冲区中的字节数，如果操作失败则返回 ERROR。

参考：

相关信息请参考 zbufLib 库描述。

例：

```
ZBUF_ID zRequest;            /* 来自于客户端的请求消息 */
Int     sFd;
int reply;                   /* 应答请求 */
...
/* 客户端到服务器端的一个请求结构 */
struct request
{
int reply;                   /* TRUE = 请求得到服务端应答 */
int msgLen;                  /* 消息文本长度 */
char message[100];           /* 消息缓冲区 */
};
...
/* 接收消息 */
zRequest = zbufFioSockRecv (sFd, sizeof(struct request));
...
/* 提取应答字段到 <reply> */
```

```
(void) zbufExtractCopy (zRequest, NULL, 0,(char *) &reply, sizeof (reply));
/* 在 zRequest 中删除字节 */
(void) zbufCut (zRequest, NULL, 0, sizeof (reply));
...
```

zbufCut()

函数原型：

```
ZBUF_SEG zbufCut
(
ZBUF_ID zbufId,          /* 从这个 zbuf 中删除字节 */
ZBUF_SEG zbufSeg,        /* 被 offset 使用的 zbuf 段偏移基地址*/
int offset,              /* 相对的字节偏移量 */
int len                  /* 删除字节数 */
)
```

功能描述：

在 zbuf 中删除字节。该函数从 zbufId 指定的字节位置开始删除 len 个字节数据。

通过 zbufSeg 和 offset 指定删除起始位置。有关更多的信息请参考 zbufLib 库描述。值得注意的是，第一个被删除的字节就是 zbufSeg 和 offset 所指定的字节。

删除字节数由参数 len 确定。如果这个参数是负数或大于 zbuf 中指定字节位置后面的字节数，zbuf 的其余数据会被删除，并可能跨度多个段。

如果某一段中所有的数据被删除，那么这个段将被删除；如果没有其他 zbuf 段使用这个数据缓冲区，那么系统将释放这个缓冲区。没有哪个包含 0 字节的 zbuf 段能够存在系统中。

函数返回删除字节的 zbuf 段 ID。如果在 zbuf 的末端来删除字节，那么函数将返回 ZBUF_NONE。

返回值：

删除字节的 zbuf 段 ID，如果操作失败则返回 NULL。

参考：

相关信息请参考 zbufLib 库描述。

zbufSplit()

函数原型：

```
ZBUF_ID zbufSplit
(
ZBUF_ID zbufId,          /* 把这个 zbuf 分裂成两个 */
ZBUF_SEG zbufSeg,        /* 被 offset 使用的 zbuf 段偏移基地址 */
int offset               /* 相对的字节偏移量 */
)
```

功能描述：

把一个 zbuf 分裂成两个独立的 zbuf。该函数在指定的字节位置把 zbufId 分裂成两个独立的 zbuf。起始部分保留在 zbufId 中，末端部分返回到新建的 zbuf 中。

通过 zbufSeg 和 offset 指定分裂位置。有关更多的信息请参考 zbufLib 库描述。值得注意的是，返回 zbuf 的第一个字节就是 zbufSeg 和 offset 所指定的字节。

返回值：

包含 zbufId 末端部分的新建 zbuf ID，如果操作失败则返回 NULL。

参考：

相关信息请参考 zbufLib 库描述。

例：

```
ZBUF_ID        Origzbuf;
ZBUF_ID        Newzbuf;
...
/* 把 zbuf 分裂成两个独立的 zbuf */
Newzbuf = zbufSplit(Origzbuf,NULL,0);
if(Newzbuf == NULL)
{
printf ("zbuf split failed!\n");
zbufDelete (Newzbuf);
return(ERROR);
}
...
```

zbufDup()

函数原型：

```
ZBUF_ID zbufDup
(
ZBUF_ID zbufId,          /* 复制这个 zbuf */
ZBUF_SEG zbufSeg,        /* 被 offset 使用的 zbuf 段偏移基地址*/
int offset,              /* 相对的字节偏移量 */
int len                  /* 复制字节数 */
)
```

功能描述：

复制 zbuf。该函数在 zbufId 的指定字节位置处开始复制 len 个字节数据，并返回给新建的 zbuf ID。

通过 zbufSeg 和 offset 指定复制开始位置。有关更多的信息请参考 zbufLib 库描述。值得注意的是，复制的第一个字节就是 zbufSeg 和 offset 所指定的字节。

返回值：

新建的 zbuf ID，如果操作失败则返回 NULL。

参考：

相关信息请参考 zbufLib 库描述。

例：

```
ZBUF_ID zReplyOrig;          /* 原始应答消息 */
ZBUF_ID zReplyDup;           /* 应答消息副本 */
...
/* 复制应答消息 zbuf, 保存原有消息 */
if ((zReplyDup = zbufDup (zReplyOrig, NULL, 0, ZBUF_END)) == NULL)
{
printf ("zbuf duplicate failed!\n");
zbufDelete (zReplyDup);
return(ERROR);
}
...
```

zbufLength()

函数原型：

```
int zbufLength
(
ZBUF_ID zbufId /* 确定这个 zbuf 的长度 */
)
```

功能描述：

确定 zbuf 的长度。该函数返回 zbufId 中字节数。

返回值：

zbuf 中的字节数，如果调用失败则返回 ERROR。

参考：

相关信息请参考 zbufLib 库描述。

zbufSegFind()

函数原型：

```
ZBUF_SEG zbufSegFind
(
ZBUF_ID zbufId,          /* 在这个 zbuf 中查找 */
ZBUF_SEG zbufSeg,        /* 被 pOffset 使用的 zbuf 段偏移基地址 */
int * pOffset            /* 相对的字节偏移量 */
)
```

功能描述：

查找 zbuf 段。该函数在 zbufId 中的指定字节位置处查找 zbuf 段，而字节位置是由 zbufSeg 和 pOffset 来指定，并返回这个 zbuf 段，且相对于返回段的新偏移量存放在 pOffset 中。

如果 zbufSeg 和 pOffset 指定的字节位置超过 zbuf 的末端，或在 zbuf 的起始位置之前，zbufSegFind()将返回 NULL。

返回值：

成功查找则返回 zbuf 段 ID，如果操作失败则返回 NULL。

参考：

相关信息请参考 zbufLib 库描述。

例：

```
ZBUF_ID zbufId;
ZBUF_SEG zbufSeg;
...
/* 查找 zbuf 段 */
if ((zbufSeg = zbufSegFind (zbufId, NULL, 0)) == NULL)
  {
    printf ("find zbuf segment failed!\n");
    return (ERROR);
  }
...
```

zbufSegNext()

函数原型：

```
ZBUF_SEG zbufSegNext
(
ZBUF_ID zbufId,          /* 在这个 zbuf 中查找 */
ZBUF_SEG zbufSeg         /* zbuf 段 */
)
```

功能描述：

在一个 zbuf 中获得下一个 zbuf 段。该函数在 zbufId 中查找处于 zbufSeg 段之后的下一个 zbuf 段。如果 zbufSeg 段为 NULL，函数将返回 zbufId 中处于第一个段之后的下一个 zbuf 段。如果 zbufSeg 段是 zbufId 中最后的 zbuf 段，那么函数将返回 NULL。

返回值：

处于 zbufSeg 段之后的下一个 zbuf 段的 ID，如果操作失败则返回 NULL。

参考：

相关信息请参考 zbufLib 库描述。

例：

```
ZBUF_ID zbufId;
ZBUF_SEG CurrentzbufSeg;
ZBUF_SEG NextzbufSeg;
...
/* 获得下一个 zbuf 段 */
NextzbufSeg = zbufSegNext (zbufId, CurrentzbufSeg);
If(NextzbufSeg == NULL)
  {
    printf ("Get the next zbuf segment failed!\n");
    return (ERROR);
  }
...
```

zbufSegPrev()

函数原型：

```
ZBUF_SEG zbufSegPrev
(
ZBUF_ID zbufId,          /* 在这个 zbuf 中查找 */
ZBUF_SEG zbufSeg         /* zbuf 段 */
)
```

功能描述：

在一个 zbuf 中获得前一个 zbuf 段。该函数在 zbufId 中查找处于 zbufSeg 段之前的前一个 zbuf 段。如果 zbufSeg 段为 NULL 或 zbufSeg 段是 zbufId 中第一个 zbuf 段，那么函数将返回 NULL。

返回值：

处于 zbufSeg 段之前的前一个 zbuf 段的 ID，如果操作失败则返回 NULL。

参考：

相关信息请参考 zbufLib 库描述。

zbufSegData()

函数原型：

```
caddr_t zbufSegData
(
ZBUF_ID zbufId,          /* 在这个 zbuf 中查找 */
ZBUF_SEG zbufSeg         /* zbuf 段 */
)
```

功能描述：

确定 zbuf 段中数据的位置，该函数返回段 zbufSeg 中数据的第一个字节的位置。如果 zbufSeg 为 NULL，则返回 zbufId 的第一个段中数据的位置。

返回值：

指向 zbuf 段中数据的第一个字节的指针，如果操作失败则为 NULL。

参考：

相关信息请参考 zbufLib 库描述。

zbufSegLength()

函数原型：

```
int zbufSegLength
(
ZBUF_ID zbufId,          /* 在这个 zbuf 中查找 */
ZBUF_SEG zbufSeg         /* zbuf 段 */
)
```

功能描述：

确定 zbuf 段的长度。该函数返回段 zbufSeg 中字节数。如果段 zbufSeg 为 NULL，则返回 zbufId 中第一个 zbuf 段的长度。

返回值：

返回段 zbufSeg 中字节数，如果操作失败则为 ERROR。

参考：

相关信息请参考 zbufLib 库描述。

11.4 Internet 地址处理函数

11.4.1 函数库描述

1．库命名

Internet(inet)地址处理函数库名称为 inetLib。

2．函数

表 11-5 中列出了 Internet 地址处理函数。

3．描述

inetLib 库提供处理 Internet 地址的函数，包括 UNIX BSD 4.3 inet_ 函数。这些函数的功能包括在 Internet 标准带点十进制符号和整数地址之间进行转换，从 Internet 地址中提取网络和主机号，以及根据网络和主机号组成 Internet 地址。

所有的 Internet 地址是按网络顺序返回。所有的网络号和本地地址部分是作为整数值返回。

4．头文件

Internet 地址处理函数声明及相关宏定义，请查阅 inetLib.h 和 inet.h 头文件。

表 11-5　Internet 地址处理函数

函　数	描　述	函　数	描　述
inet_addr()	把圆点记法的 Internet 地址转换成长整数	inet_network()	把包含地址的字符串转换成网络地址
inet_lnaof()	从 Internet 地址中获得本地地址(主机号)	inet_ntoa_b()	把网络地址转换成带点的符号，并保存在缓冲区中
inet_makeaddr_b()	根据网络和主机号组成 Internet 地址		
inet_makeaddr()	根据网络和主机号组成 Internet 地址	inet_ntoa()	把网络地址转换成带点的十进制符号
inet_netof()	从 Internet 地址中返回网络号	inet_aton()	把带点的十进制符号转换成网络地址，并保存在一个结构中
inet_netof_string()	提取圆点记法的网络地址		

5．参考

相关信息请参考 UNIX BSD 4.3 inet 手册，以及“VxWorks Programmer's Guide”中的网络部分。

11.4.2　Internet 地址处理函数详细描述

inet_addr()

函数原型：

```
u_long inet_addr
(
char * inetString        /* 包含 Internet 地址的字符串 */
)
```

功能描述：

把圆点记法的 Internet 地址转换成长整数。该函数解释一个 Internet 地址。所有的网络库函数调用这个函数来获得地址。返回值是按网络顺序排列。

返回值：

成功转换则返回 Internet 地址，如果转换失败则返回 ERROR。

参考：

相关信息请参考 inetLib 库描述。

例：

```
#define BROADCAST_ADDR "147.11.184.255"          /* 广播地址 */
struct sockaddr_in sendToAddr;                   /* 接收者的地址 */
int          port = 7001;                        /* 端口号 */
...
sendToAddr.sin_family = AF_INET;
sendToAddr.sin_port = htons (port);
/* "147.11.184.255" => 0x930bb8ff */
sendToAddr.sin_addr.s_addr = inet_addr (BROADCAST_ADDR);
...
```

inet_lnaof()

函数原型：

```
int inet_lnaof
(
int inetAddress    /* Internet 地址 */
)
```

功能描述：

从 Internet 地址中获得本地地址(主机号)。该函数从指定的 Internet 地址 inetAddress 中获得本地网络地址。该函数处理 A、B、C 类网络号。

返回值：

inetAddress 的本地地址。

参考：

相关信息请参考 inetLib 库描述。

例：

```
int hostNum;
...
/* 获得主机号，下面的操作使 hostNum 的值为 2。 */
hostNum = inet_lnaof (0x5a000002);
...
```

inet_makeaddr_b()

函数原型：

```
void inet_makeaddr_b
(
int netAddr,                    /* Internet 地址的网络部分 */
int hostAddr,                   /* Internet 地址的主机部分 */
struct in_addr * pinetAddr      /* Internet 地址 */
)
```

功能描述：

根据网络和主机号组成 Internet 地址。该函数根据网络号 netAddr 和本地主机地址 hostAddr 组成 Internet 地址，并存放在 pInetAddr 中。

返回值：

无。

参考：

相关信息请参考 inetLib 库描述。

例：

```
struct in_addr InetAddr;
...
/* 把地址 0x5a000002 复制到本地地址 InetAddr 中 */
inet_makeaddr_b (0x5a, 2, &InetAddr);
...
```

inet_makeaddr()

函数原型：

```
struct in_addr inet_makeaddr
(
int netAddr, /* Internet 地址的网络部分 */
int hostAddr /* Internet 地址的主机部分 */
)
```

功能描述：

根据网络和主机号组成 Internet 地址。该函数根据网络号 netAddr 和本地主机地址 hostAddr 组成 Internet 地址。

返回值：

网络地址，且保存在一个 in_addr 结构中。

参考：

相关信息请参考 inetLib 库描述，以及 inet_makeaddr_b()函数。

例：

```
struct in_addr netAdd;
...
/* 下面的操作将使 netAdd 的值为 0x5a000002。 */
netAdd = inet_makeaddr (0x5a, 2);
...
```

inet_netof()

函数原型：

```
int inet_netof
(
struct in_addr inetAddress /* Internet 地址 */
)
```

功能描述：

从 Internet 地址中返回网络号。该函数从 Internet 地址 inetAddress 中提取网络号。

返回值：

Internet 地址 inetAddress 中的网络号。

参考：

相关信息请参考 inetLib 库描述。

例：

```
int     inetNum;
...
/* 下面的操作将使 inetNum 的值为 0x5a */
inetNum = inet_netof(0x5a000002);
...
```

inet_netof_string()

函数原型：

```
void inet_netof_string
(
char * inetString,        /* Internet 地址 */
char * netString          /* 网络 Internet 地址 */
)
```

功能描述：

提取圆点记法的网络地址。该函数从主机 Internet 地址中提取网络 Internet 地址。该函数处理 A、B、C 类网络地址。缓冲区 netString 的长度应该是 18 个字节。

返回值：

无。

参考：

相关信息请参考 inetLib 库描述。

例：

```
char   * netAddString;
...
/* 下面的操作将把"90.0.0.0"复制到 netAddString 中 */
inet_netof_string ("90.0.0.2", netAddString);
...
```

inet_network()

函数原型：

```
u_long inet_network
(
char * inetString /* 包含 Internet 地址的字符串 */
)
```

功能描述：

把包含 Internet 地址的字符串转换成网络地址。该函数把包含 Internet 网络号的 ASCII 字符串转换成一个网络地址。

返回值：

Internet 网络号。

参考：

相关信息请参考 inetLib 库描述。

例：

```
u_long      netAdd;
...
/* 下面的操作将把"90"转换成 0x5a */
netAdd = inet_network ("90");
...
```

inet_ntoa_b()

函数原型：

```
void inet_ntoa_b
(
struct in_addr inetAddress,    /* Internet 地址 */
char * pString                 /* ASCII 字符串 */
)
```

功能描述：

把网络地址转换成带点的符号，并保存在缓冲区中。

返回值：

无。

参考：

相关信息请参考 inetLib 库描述。

例：

```
char   * netString;
struct in_addr iaddr;
...
/* 复制字符串 "90.0.0.2" 到 netString 中*/
iaddr.s_addr = 0x5a000002;
inet_ntoa_b (iaddr, netString);
...
```

inet_ntoa()

函数原型：

```
char *inet_ntoa
(
struct in_addr inetAddress /* Internet 地址 */
)
```

功能描述：

把网络地址转换成带点的十进制符号。

返回值：

指向包含 Internet 地址的 ASCII 字符串指针。

参考：

相关信息请参考 inetLib 库描述。

例：

```
struct in_addr iaddr;
char    * inet;
...
/* inet 的内容将为"90.0.0.2" */
iaddr.s_addr = 0x5a000002;
inet = inet_ntoa (iaddr);
...
```

inet_aton()

函数原型：

```
STATUS inet_aton
(
char * pString,                 /* 包含地址的字符串 */
struct in_addr * inetAddress    /* 存放地址的结构 */
)
```

功能描述：

把带点的十进制符号转换成网络地址，并保存在结构中。

返回值：

成功转换则返回 OK，如果转换失败则返回 ERROR。

参考：

相关信息请参考 inetLib 库描述。

例：

```
struct in_addr pinetAddr;
```

```
STATUS    st;
...
/* 结构 pinetAddr 的成员 s_addr 值将为 0x5a000002 */
st = inet_addr ("90.0.0.2", pinetAddr);
if(st == ERROR)
   {
   printf("Error in inet_aton function!\n");
   return(ERROR);
   }
...
```

11.5 网络接口函数

11.5.1 函数库描述

1．库命名

网络接口函数库名称为 ifLib。

2．函数

表 11-6 中列出了网络接口函数。

表 11-6 网络接口函数

函　数	描　述	函　数	描　述
ifAddrAdd()	给网络接口分配 Internet 地址	ifMaskGet()	获得网络接口的子网掩码
ifAddrSet()	给网络接口分配 Internet 地址	ifFlagChange()	改变网络接口标识
ifAddrGet()	获得网络接口的 Internet 地址	ifFlagSet()	指定网络接口标识
ifBroadcastSet()	给网络接口分配广播地址	ifFlagGet()	获得网络接口的标识
ifBroadcastGet()	获得网络接口的广播地址	ifMetricSet()	指定网络接口路由段数
ifDstAddrSet()	指定用于点到点连接的目的地地址	ifMetricGet()	获得网络接口路由段数
ifDstAddrGet()	获得用于点到点连接的目的地地址	ifRouteDelete()	删除与网络接口关联的路由
ifMaskSet()	为网络接口规定一个子网	ifunit()	把接口名映射给一个接口结构指针

3．描述

ifLib 库提供包含配置网络接口参数的函数。通常情况下，每个函数对应 UNIX 命令 ifconfig 的一个函数。

4．头文件

网络接口函数声明在 ifLib.h 头文件中。

5．参考

相关信息请参考 hostLib 库描述，以及“VxWorks Programmer's Guide”中的网络部分。

11.5.2 Internet 地址处理函数详细描述

ifAddrAdd()

函数原型：

```
STATUS ifAddrAdd
(
```

```
char * interfaceName,        /* 网络接口名，例如：ei0 */
char * interfaceAddress,     /* 分配给网络接口的 Internet 地址 */
char * broadcastAddress,     /* 分配给网络接口的广播地址*/
int subnetMask               /* 子网掩码 */
)
```

功能描述：

分配 Internet 地址给网络接口。Internet 地址可以是主机名或标准的 Internet 地址格式(例如，90.0.0.6)。如果指定主机名，则应该首先使用 hostAdd()函数把主机名添加到主机表中。用户必须指定参数 interfaceName、interfaceAddress。broadcastAddress 为可选参数，如果为 NULL，in_ifinit()将通过使用 interfaceAddress 和网络掩码产生一个 broadcastAddress。subnetMask 也是可选参数，如果 subnetMask 为 0，in_ifinit()将设置一个与网络掩码一样的 subnetMask 。如果是点到点连接，则 broadcastAddress 也就是 destAddress(目的地地址）。

返回值：

成功分配地址则返回 OK，如果网络接口不可以被设置则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifAddrGet()、ifDstAddrSet()和 ifDstAddGet()函数。

例：

```
STATUS    st;
...
/* 分配 Internet 地址 "192.168.10.10" 给网络接口 fei0 */
st = ifAddrAdd("fei0","192.168.10.10",NULL,0);
if(st == ERROR)
{
printf("Assigns an Internet address to fei0 failed!\n")
return(ERROR);
}
...
```

ifAddrSet()

函数原型：

```
STATUS ifAddrSet
(
char * interfaceName,    /* 网络接口名，例如：ei0 */
char * interfaceAddress  /* Internet 地址 */
)
```

功能描述：

分配 Internet 地址给网络接口。Internet 地址可以是主机名或标准的 Internet 地址格式(例如，90.0.0.6)。如果指定主机名，则用户应该首先使用 hostAdd()函数把主机名添加到主机表中。

成功调用 ifAddrSet()的结果是增加了新的路由。使用子网掩码可以决定地址的网络部分，而子网掩码则通过 ifMaskSet()函数来设置，如果没有调用 ifMaskSet()函数则系统使用默认类掩码。标准的做法是先调用 ifMaskSet()给网络接口分配子网掩码，后调用 ifAddrSet()给网络接口分配 Internet 地址。

返回值：

成功分配地址则返回 OK，如果网络接口不可以被设置则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifAddrGet()、ifDstAddrSet()和 ifDstAddGet()函数。

例：

```
STATUS    st;
...
/* 分配 Internet 地址 "192.168.10.10" 给网络接口 fei0 */
st = ifAddrSet("fei0","192.168.10.10");
...
```

ifAddrGet()

函数原型：

```
STATUS ifAddrGet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
char * interfaceAddress      /* 存放 Internet 地址的缓冲区 */
)
```

功能描述：

获得网络接口的 Internet 地址。该函数根据指定的网络接口名 interfaceName 获得它的 Internet 地址，并复制到缓冲区 interfaceAddress 中。该缓冲区大小应该足够存放 18 个字节的数据。

返回值：

成功获得地址则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifAddrSet()、ifDstAddrSet()和 ifDstAddGet()函数。

例：

```
STATUS    st;
char   * inetAddr;
...
/* 获得网络接口 fei0 的 Internet 地址 */
st = ifAddrGet("fei0", inetAddr);
...
```

ifBroadcastSet()

函数原型：

```
STATUS ifBroadcastSet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
char * broadcastAddress      /* 广播地址 */
)
```

功能描述：

分配广播地址给网络接口。广播地址必须是标准 Internet 地址格式的字符串(例如，90.0.0.0)。

网络接口的默认广播地址是 Internet 地址其主机部分全为“1”(例如，90.255.255.255)的一个 Internet 地址。这符合当前 ARPA 规范。然而，一些老的系统使用 Internet 地址的主机部分都为“0”的 Internet 地址作为广播地址。

注意：

VxWorks 通常使用主机部分都为“1”的 Internet 地址作为广播地址。这样，一个数据报发送给 Internet

地址“192.255.255.255”，则是对所有在 192 段网络上的主机进行数据报广播。然而，为了兼容其他系统，VxWorks 自动接受主机部分都为“0”的 Internet 地址作为广播地址，并允许通过调用该函数为每一个网络接口重新分配广播地址。

返回值：

成功设置广播地址则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述。

例：

```
STATUS    st;
...
/* 分配广播地址“90.255.255.255”给网络接口“fei0” */
st = ifBroadcastSet ("fei0", "90.255.255.255");
...
```

ifBroadcastGet()

函数原型：

```
STATUS ifBroadcastGet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
char * broadcastAddress      /* 存放广播地址的缓冲区 */
)
```

功能描述：

获得网络接口的广播地址。该函数获得指定的网络接口的广播地址，并把它复制到缓冲区 broadcastAddress 中。

返回值：

成功获得广播地址则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifBroadcaseSet()函数。

例：

```
STATUS    st;
char   * BAddr;
...
/* 获得网络接口“fei0”的广播地址 */
st = ifBroadcastGet ("fei0", BAddr);
...
```

ifDstAddrSet()

函数原型：

```
STATUS ifDstAddrSet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
char * dstAddress            /* 分配给目的地的 Internet 地址 */
)
```

功能描述：

指定用于点对点连接的目的地地址。该函数为连接到点对点网络的机器分配一个目的地地址，如一个SLIP 连接。点对点连接协议(例如，SLIP)需要指定连接的两端地址。

返回值：

成功设置则返回 OK，如果设置失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifAddrSet()和 ifDstAddrGet()函数。

例：

```
STATUS    st;
...
/* 指定网络接口“fei0”的目的地址 */
st = ifDstAddrSet ("fei0", "90.90.22.22");
...
```

ifDstAddrGet()

函数原型：

```
STATUS ifDstAddrGet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
char * dstAddress            /* 存放目的地地址的缓冲区 */
)
```

功能描述：

获得用于点到点连接的目的地地址。该函数获得连接到点对点网络机器的目的地地址， 并把它复制到dstAddress 中。

返回值：

成功获得则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifAddrGet()和 ifDstAddrSet()函数。

例：

```
STATUS    st;
char   * dstAddr;
...
/* 获得网络接口“fei0”的目的地址 */
st = ifDstAddrGet ("fei0", dstAddr);
...
```

ifMaskSet()

函数原型：

```
STATUS ifMaskSet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
int netMask                  /* 子网掩码 (例如：0xff000000) */
)
```

功能描述：

为网络接口规定一个子网。该函数分配一个子网掩码给网络接口，子网掩码由一连串的“1”和一连串

“0”组成。“1”对应于网络号码和子网号码字段，而“0”对应于主机号码字段。

为了正确说明这个地址，应该先给网络接口分配子网掩码，然后使用函数 ifAddrSet()给接口分配 Internet 地址。

返回值：

成功设置则返回 OK，如果设置失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifAddrSet()函数。

例：

```
STATUS    st;
...
/* 为网络接口“fei0”规定一个子网 */
st = ifMaskSet ("fei0", 0xffff0000);
...
```

ifMaskGet()

函数原型：

```
STATUS ifMaskGet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
int * netMask                /* 存放子网掩码的缓冲区 */
)
```

功能描述：

获得网络接口的子网掩码。该函数获得网络接口 interfaceName 的子网掩码，并存放在缓冲区 netMask 中。

返回值：

成功获得则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifAddrGet()和 ifFlagGet()函数。

例：

```
STATUS    st;
char    * subnet;
...
/* 获得网络接口“fei0”的子网掩码 */
st = ifMaskGet ("fei0", subnet);
...
```

ifFlagChange()

函数原型：

```
STATUS ifFlagChange
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
int flags,                   /* 网络标识 */
BOOL on                      /* TRUE = 打开, FALSE = 关闭 */
)
```

功能描述：

改变网络接口标识。该函数改变网络接口 interfaceName 的网络标识。如果参数 on 为 TRUE，则打开指定的标识，其他则关闭。调用函数 ifFlagGet()和 ifFlagSet()进行实际操作。

返回值：

成功改变则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifAddrSet()、ifMaskSet()、ifFlagSet()和 ifFlagGet()函数。

例：

```
STATUS    st;
...
/* 改变网络接口标识 */
st = ifFlagChange ("fei0", IFF_UP | IFF_PROMISC, TRUE);
if(st == ERROR)
{
printf("Change fei0 flag failed!\n")
return(ERROR);
}
...
```

ifFlagSet()

函数原型：

```
STATUS ifFlagSet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
int flags                    /* 网络标识 */
)
```

功能描述：

指定网络接口标识。该函数改变网络接口的标识。其标识可以是下面标识的任意组合：

IFF_UP (0x1)：打开或关闭网络接口；

IFF_DEBUG (0x4)：打开调试；

IFF_LOOPBACK (0x8)：设置回送网络；

IFF_NOTRAILERS (0x20)：总是设置(VxWorks 不使用追踪协议)；

IFF_PROMISC (0x100)：接收所有的包，但不包括广播包和它自身地址包；

IFF_ALLMULTI (0x200)：接收所有的多点传送包；

IFF_NOARP (0x80)：关闭接口 ARP(地址解析协议)。

返回值：

成功设置则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifFlagChange()和 ifFlagGet()函数。

例：

```
STATUS    st;
...
/* 改变网络接口标识 */
st = ifFlagSet ("fei0", IFF_UP | IFF_PROMISC);
```

...

ifFlagGet()

函数原型：

```
STATUS ifFlagGet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
int * flags                  /* 网络标识 */
)
```

功能描述：

获得网络接口的标识。该函数获得网络接口 interfaceName 的标识，并把它存放在参数 flags 中。

返回值：

成功获得则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifFlagSet()函数。

例：

```
STATUS    st;
int       netflag;
...
/* 获得网络接口标识 */
st = ifFlagGet ("fei0", &netflag);
...
```

ifMetricSet()

函数原型：

```
STATUS ifMetricSet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
int metric                   /* 路由段数 */
)
```

功能描述：

指定网络接口路由段数。该函数为网络接口配置从主机到目的地网络的路由段数 metric。IP 路由运算使用这个信息来计算各条通路所包含的相对距离。例如，为 SLIP 接口分配一个较高的路由段数来阻碍一个数据包路由到低速串行线路连接。

注意：

当路由段数为 0 时，IP 路由运算允许直接发送包含 IP 网络地址的数据包，其地址并不需要与本地网络地址相同。

返回值：

成功设置则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifMetricGet()函数。

例：

```
STATUS    st;
...
```

```
/* 指定网络接口路由段数 */
st = ifMetricSet ("fei0", 5);
...
```

ifMetricGet()

函数原型：

```
STATUS ifMetricGet
(
char * interfaceName,        /* 网络接口名，例如：ei0 */
int * pMetric                /* 路由段数 */
)
```

功能描述：

获得网络接口的路由段数。该函数获得网络接口 interfaceName 的路由段数，并把它复制到 pMetric 中。

返回值：

成功获得则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述，以及 ifMetricSet()函数。

例：

```
STATUS    st;
int       Metric;
...
/* 获得网络接口路由段数 */
st = ifMetricGet ("fei0", &Metric);
if(st == ERROR)
{
printf("Error in ifMetricGet function!\n")
return(ERROR);
}
...
```

ifRouteDelete()

函数原型：

```
int ifRouteDelete
(
char * ifName,          /* 网络接口名 */
int unit                /* 单元数 */
)
```

功能描述：

删除与网络接口关联的路由。该函数删除所有与网络接口关联的路由。如果网络接口的目的地等于分配的地址或网络号，一个路由与该接口关联。该函数并不删除通过给定接口的任意目的地的路由。

返回值：

删除的路由数，如果没有指定接口则返回 ERROR。

参考：

相关信息请参考 ifLib 库描述。

ifunit()
函数原型：

```
struct ifnet *ifunit
(
char * ifname      /* 网络接口名 */
)
```

功能描述：
把网络接口名映射给一个网络接口结构指针。如果这个网络接口不存在则返回 NULL。
返回值：
指向网络结构的指针，如果网络接口不存在则返回 NULL。
参考：
相关信息请参考 ifLib、etherLib 库描述。
例：

```
struct ifnet      *ifPtr;
...
if ((ifPtr = ifunit ("feio")) == NULL)
        printf ("Interface not found!\n");
...
```

11.6　IP 过滤钩子函数

11.6.1　函数库描述

1．库命名
IP 过滤钩子函数库名称为 ifFilterLib。
2．函数
表 11-7 中列出了 IP 过滤钩子函数。

表 11-7　IP 过滤钩子函数

函　数	描　述
ipFilterLibInit()	初始化 IP 过滤模块
ipFilterHookAdd()	增加接收所有 Internet 协议包函数
ipFilterHookDelete()	删除 IP 过滤钩子函数

3．描述
ifFilterLib 库提供直接访问 IP 包的函数。通过使用函数 ipFilterHookAdd()可以检查或处理收到的原始 IP 包。输入钩子用来接收原始 IP 包，且是 IP 协议的一部分。过滤钩子也可以用来建立 IP 通信监测和作为测试的工具。

通常情况下，应当通过 sockLib 库提供的高级别套接字接口访问网络。在应用程序中应该极少使用 ipFilterLib 库提供的函数。

在配置文件 configAll.h 中如果定义了 INCLUDE_IP_FILTER 宏，则系统自动调用函数 ipFilterLibInit()把 IP 过滤模块链接到 VxWorks 系统中。

4．头文件
IP 过滤钩子函数声明请查阅 ifLib.h 头文件。
5．参考
相关信息请参考“VxWorks Programmer's Guide”中的网络部分。

11.6.2　IP 过滤钩子函数详细描述

ipFilterLibInit()
函数原型：

void ipFilterLibInit (void)

功能描述：

初始化 IP 过滤模块。该函数把 IP 过滤模块链接到 VxWorks 系统中。如果在配置文件 configAll.h 中定义了 INCLUDE_IP_FILTER 宏，则系统自动调用该函数。

返回值：

无。

参考：

相关信息请参考 ifFilterLib 库描述。

ipFilterHookAdd()

函数原型：

```
STATUS ipFilterHookAdd
(
FUNCPTR ipFilterHook      /* 接收原始 IP 包的函数 */
)
```

功能描述：

增加接收所有 Internet 协议包函数。该函数增加一个钩子函数，当接收每一个 IP 包时都将调用该钩子函数。过滤钩子函数声明如下：

```
BOOL ipFilterHook
(
struct ifnet *pIf,              /* 在这个结构上接收网络数据包 */
struct mbuf **pPtrMbuf,         /* 一个指向 mbuf 链的指针的指针 */
struct ip **pPtrIpHdr,          /* 一个指向 IP 报头的指针的指针*/
int ipHdrLen,                   /* IP 报头长度 */
)
```

如果钩子函数处理输入包，则钩子函数应该返回 TRUE。如果返回 TRUE，钩子函数有责任调用 m_freem(*pPtrMbuf)函数释放 mbuf 链。如果钩子函数没有处理则返回 FALSE，但正常的处理应该依然发生。

IP 包挂在一个 mbuf 链上，该链是作为一个指针的指针参数被传递。这个 mbuf 链可以通过参数 pPtrMbuf 直接访问它。当从钩子中返回后系统可以再次使用这个 mbuf 链。如果钩子函数需要保留输入包，应该复制它并存放在别处。为了达到这个目的，我们可以通过使用宏 copy_from_mbufs(buffer,*pPtrMbuf,len)来实现。该宏在 net/mbuf.h 中有定义。

参数 pPtrIpHdr 是一个指向 IP 报头的指针的指针。通过这个指针可以获得 IP 报头。IP 报头用来检查和处理 IP 报头区。在包进入过滤钩子之前，把 IP 报头中的 ip_len、ip_id 和 ip_offset 从网络字节顺序转换成主机字节顺序。

如果钩子函数返回 FALSE，从钩子函数返回后可以再次使用 pPtrMbuf 和 pPtrIpHdr。通常情况下，用户不需要修改 pPtrMbuf 或 pPtrIpHdr。

返回值：

总是返回 OK。

参考：

相关信息请参考 ipFilterLib 库描述。

ipFilterHookDelete()

函数原型：

void ipFilterHookDelete (void)

功能描述：

删除 IP 过滤钩子函数。

返回值：

无。

参考：

相关信息请参考 ipFilterLib 库描述。

11.7　IP 堆栈管理函数

11.7.1　函数库描述

1．库命名

IP 堆栈管理函数库名称为 ipProto。

2．函数

表 11-8 中列出了 IP 堆栈管理函数。

表 11-8　IP 堆栈管理函数

函　数	描　述
ipAttach()	绑定 TCP/IP 堆栈到 MUX 上的指定的网络接口
ipDetach()	卸载网络接口所关联的 TCP/IP 堆栈模块

3．描述

ifProto 库在伯克利(Berkeley)协议堆栈和 MUX 接口之间提供一个接口。ipAttach()函数绑定 IP 协议给指定的设备。如果定义了 INCLUDE_END，在网络初始化期间系统自动调用该函数。ipDetach()函数卸载 IP 协议堆栈。

4．头文件

IP 堆栈管理函数声明及宏定义请查阅 ipProto.h、end.h、muxLib.h、etherMultilib.h、sys/ioctl.h 和 etherLib.h 头文件。

5．参考

相关信息请参考“VxWorks Programmer's Guide”中的网络部分。

11.7.2　IP 堆栈管理函数详细描述

ipAttach()

函数原型：

```
int ipAttach
(
int unit,              /* 网络设备单元号 */
char * pDevice         /* 网络设备名(例如：ene、fei 等). */
)
```

功能描述：

绑定 TCP/IP 堆栈到 MUX 上的指定的网络接口。该函数获得网络设备单元号和一个 END 驱动的设备名，接着把 TCP/IP 堆栈绑定到这个 MUX 上。如果操作成功，IP 协议将从这个驱动程序中开始接收数据包。

返回值：

成功绑定则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ipProto 库描述。

例：

```
int          status;
...
/* 给第二块网卡“rtl”分配 TCP/IP 堆栈 */
status = ipAttach(1, "rtl");
if(status == ERROR)
{
printf("Attach a IP stack to the sencod rtl failed!\n")
return(ERROR);
}
...
ipDetach()
```

函数原型：

```
    STATUS ipDetach
    (
    int unit,              /* 网络设备单元号 */
    char * pDevice         /* 网络设备名(例如：ene、fei 等). */
    )
```

功能描述：

卸载网络接口所关联的 TCP/IP 堆栈模块。该函数从 MUX 上卸载 TCP/IP 堆栈。如果操作成功，IP 协议将不再从这个设备的 END 驱动程序上接收数据包。

返回值：

成功卸载则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 ipProto 库描述。

例：

```
int          status;
...
/* 卸载第二块网卡“rtl”的 TCP/IP 堆栈 */
status = ipDetach (1, "rtl");
...
```

11.8 主机表子程序函数

11.8.1 子程序函数库描述

1．库命名

主机表子程序函数库名称为 hostLib。

2．函数

表 11-9 中列出了主机表子程序函数。

3．描述

hostLib 库提供存储和访问网络主机数据库函数。主机表包含本地网络上所知道的主机的信息。主机表(用 hostShow()显示)具体包含 Internet 地址、正式的主机名和别名。

表 11-9　主机表子程序函数

函　数	描　述	函　数	描　述
hostTblInit()	初始化网络主机表	hostGetByAddr()	根据 Internet 地址在主机表中查寻主机
hostAdd()	增加一个主机到主机表中	sethostname()	设置机器符号名
hostDelete()	从主机表中删除一个主机	gethostname()	获得机器符号名
hostGetByName()	根据主机名在主机表中查寻主机		

根据协议，网络地址是带点(“.”)的十进制数字。inetLib 库包含 Internet 地址处理程序。主机名和别名可能包含任何可打印的字符。

在使用这个库中任何程序之前，系统必须调用 hostTblInit()函数初始化这个库。如果定义了配置宏 INCLUDE_NET_INIT，VxWorks 系统将自动调用库初始化函数。

4．头文件

主机表子程序函数其声明在 hostLib.h 头文件中。

5．参考

相关信息请参考 inetLib 库描述，以及“VxWorks Programmer's Guide”中的网络部分。

11.8.2　主机表子程序函数详细描述

hostTblInit()

函数原型：

```
void hostTblInit (void)
```

功能描述：

初始化网络主机表。该函数初始化主机列表数据结构。在使用这个库中任何其他程序之前，系统必须调用 hostTblInit()函数初始化这个库。如果定义了配置宏 INCLUDE_NET_INIT，VxWorks 系统将自动调用库初始化函数。

返回值：

无。

参考：

相关信息请参考 hostLib 库描述，以及 usrConfg()函数。

hostAdd()

函数原型：

```
STATUS hostAdd
(
char * hostName,        /* 主机名 */
char * hostAddr         /* 标准 Internet 格式的主机地址 */
)
```

功能描述：

增加一个主机到主机表中。该函数增加主机名到本地主机表中。在打开远程主机上的套接字之前，或通过 netDrv 或 nfsDrv 访问远程主机上的文件之前，必须调用该函数。

主机表每个条目都有一个 Internet 地址。对于一个地址可能使用多个名。附加的主机名作为别名被添加。

返回值：

成功增加则返回 OK，如果主机表满、主机名或 Internet 地址已经记录、无效 Internet 地址、内存不够则

返回 ERROR。

参考：

相关信息请参考 hostLib、netDrv 和 nfsDrv 库描述。

操作示范：

分配主机名“rainbow”给网络地址“192.168.10.10”。在 WindSh 下，操作如下：

```
-> hostAdd "rainbow", "192.168.10.10"
-> hostShow
hostname     inet address          aliases
--------     ------------          -------
localhost    127.0.0.1
rainbow      192.168.10.10
value = 0 = 0x0
```

hostDelete()

函数原型：

```
STATUS hostDelete
(
char * name,      /* 主机名或别名 */
char * addr       /* 标准 Internet 格式的主机地址 */
)
```

功能描述：

从主机表中删除一个主机。如果参数 name 是一个主机名，则删除该主机记录项。如果参数 name 是一个别名，则删除别名。

返回值：

成功删除则返回 OK，如果是无效参数或是未知主机则返回 ERROR。

参考：

相关信息请参考 hostLib 库描述。

例：

```
STATUS          status;
...
/* 删除主机*/
status = hostDelete("rainbow", "192.168.10.10");
...
```

hostGetByName()

函数原型：

```
int hostGetByName
(
char * name /* 主机名 */
)
```

功能描述：

根据主机名在主机表中查寻主机。该函数返回主机的 Internet 地址，而该地址是先前调用 hostAdd()函数增加到主机表中的。如果在 VxWorks 映像中配置了域名服务(DNS——Domain Name Service)库，且又在本地主机表中没有发现该主机名，则将会向 DNS 服务器询问主机 IP 地址。

返回值：

成功找到则返回主机的 Internet 地址，如果是未知主机则返回 ERROR。

参考：

相关信息请参考 hostLib 库描述。

例：

```
int             hostAddr;
...
/* 查寻主机 */
hostAddr = hostGetByName ("rainbow");
if(hostAddr == ERROR)
    printf("The host is unknown!\n")
else
    printf("The host address : %x\n", hostAddr)
...
```

hostGetByAddr()

函数原型：

```
    STATUS hostGetByAddr
    (
    int addr,               /* 主机 Internet 地址 */
    char * name             /* 存放主机名的缓冲区 */
    )
```

功能描述：

根据 Internet 地址在主机表中查寻主机。该函数通过主机 Internet 地址查找主机名并复制到参数 name 中，且系统应该预先分配 65(MAXHOSTNAMELEN+1)个字节的内存给缓冲区 name。如果在 VxWorks 映像中配置了域名服务库，且又在本地主机表中没有发现该主机名，则将会向 DNS 服务器询问主机名。

警告：

该函数并不寻找别名。主机名的长度限制为 64(MAXHOSTNAMELEN)个字符。

返回值：

成功查找则返回 OK，如果缓冲区无效或是未知主机则返回 ERROR。

参考：

相关信息请参考 hostLib 库描述，以及 hostGetByName()函数。

例：

```
char   * hostname;
STATUS    st;
...
/* 查寻主机 */
st = hostGetByAddr (0x5a000002,hostname);
if(st == ERROR)
    printf("invalid or the host is unknown!\n")
else
    printf("The host name : %s\n", hostname)
.....
```

sethostname()

函数原型：

```
int sethostname
(
char * name,      /* 机器名 */
int nameLen       /* 名字长度 */
)
```

功能描述：

设置机器符号名。该函数设置目标机器的符号名，用来识别机器。

返回值：

成功设置则返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 hostLib 库描述。

例：

```
int          status;
char hostname[] = "rainbow";
...
/* 设置机器名 */
status = sethostname(hostname,sizeof(hostname);
```

gethostname()

函数原型：

```
int gethostname
(
char * name,      /* 机器名 */
int nameLen       /* 名字长度 */
)
```

功能描述：

获得机器符号名。该函数获得目标机器的符号名，用来识别机器。

返回值：

成功获得返回 OK，如果操作失败则返回 ERROR。

参考：

相关信息请参考 hostLib 库描述。

11.9 网络信息显示函数

11.9.1 函数库描述

1．库命名

网络信息显示函数库名称为 netShow。

2．函数

表 11-10 中列出了网络信息显示函数。

表 11-10 网络信息显示函数

函 数	描 述	函 数	描 述
ifShow()	显示缚上的(attach)网络接口	netShowInit()	初始化网络显示函数
inetstatShow()	显示所有处于连接状态的 Internet 协议套接字	arpShow()	显示系统 ARP 表中的条目
ipstatShow()	显示 IP 统计	arptabShow()	显示知道的 ARP 条目
netPoolShow()	显示网络数据池统计	routestatShow()	显示路由统计
netStackDataPoolShow()	显示网络堆栈数据池统计	routeShow()	显示主机和网络路由表
netStackSysPoolShow()	显示网络堆栈系统池统计	hostShow()	显示主机表
mbufShow()	显示 mbuf 统计	mRouteShow()	打印路由表的条目

3. 描述

netShow 库提供的函数用来显示各种各样与网络有关联的统计。例如网络接口的配置参数、协议统计、套接字统计等。

理解这些统计需要 Internet 网络协议的详细知识。有关协议信息，可以在下面所列出的书中找到：

- Internetworking with TCP/IP Volume III, by Douglas Comer and David Stevens;
- UNIX Network Programming, by Richard Stevens;
- The Design and Implementation of the 4.3 BSD UNIX Operating System, by Leffler,McKusick, Karels and Quarterman。

netShowInit()函数把网络显示模块链接到 VxWorks 系统中。如果在 configAll.h 文件中定义了 INCLUDE_NET_SHOW 宏，则系统自动执行该函数。

4. 头文件

网络信息显示函数声明请查阅 netShow.h 头文件。

5. 参考

相关信息请参考 netShow、ifLib、icmpShow、igmpShow、tcpShow 和 udpShow 库描述，以及“VxWorks Programmer's Guide”中的网络部分。

11.9.2 网络信息显示函数详细描述

ifShow()

函数原型：

```
void ifShow
(
char * ifName     /* 网络接口名 */
)
```

功能描述：

显示缚上的(attach)网络接口。该函数显示网络接口，其目的是为了调试和诊断。如果给定 ifName，只是显示属于该接口的信息。如果忽略 ifName，则显示所有缚上的网络接口。

对于选择的每个接口，显示的信息包括：Internet 地址、点对点地址、广播地址、网络掩码、子网掩码、以太网地址、路由段数、最大传输单元、接口上发送和接收包的个数、输入和输出错误数和标识(例如：回送、广播、ARP、运行和调试)。

返回值：

无。

参考：
相关信息请参考 netShow、ifLib 库描述，以及 routeShow()函数。
例：

```
...
/* 显示网络接口“fei0”的网络信息*/
printf("Display the net interface fei0!\n");
ifShow("fei0");
...
```

inetstatShow()
函数原型：

```
void inetstatShow (void)
```

功能描述：
显示所有处于连接状态的 Internet 协议套接字。该函数显示格式类似于 UNIX 的 netstat 命令。
返回值：
无。
参考：
相关信息请参考 netShow 库描述。
ipstatShow()
函数原型：

```
void ipstatShow
(
BOOL zero        /* TRUE = 重新复位统计 */
)
```

功能描述：
显示 IP 统计。该函数显示 IP 协议的详细统计。
返回值：
无。
参考：
相关信息请参考 netShow 库描述。
例：

```
void netshow (void)
{
/* 显示所有处于连接状态的 Internet 协议套接字 */
inetstatShow();
...
/* 显示 IP 统计 */
ipstatShow(TRUE);
}
```

netPoolShow()
函数原型：

```
void netPoolShow
(
```

```
    NET_POOL_ID pNetPool    /* 网络数据池 */
    )
```

功能描述：

显示网络数据池统计。该函数显示网络数据池 pNetPool 中的 mblk 和簇分布。

返回值：

无。

参考：

相关信息请参考 netShow 库描述。

netStackDataPoolShow()

函数原型：

```
void netStackDataPoolShow (void)
```

功能描述：

显示网络堆栈数据池统计。该函数显示网络堆栈数据池中 mBlk 和簇的分布。网络数据池只是在通过网络堆栈传输数据时使用。

返回值：

无。

参考：

相关信息请参考 netShow、netBufLib 库描述，以及 netStackSysPoolShow()函数。

netStackSysPoolShow()

函数原型：

```
void netStackSysPoolShow (void)
```

功能描述：

显示网络堆栈系统池统计。该函数显示网络系统池中 mBlk 和簇的分布。网络系统池只是被系统结构所使用，例如套接字、路由、接口地址、协议控制块、多点传输地址和多点传输路由条目。

返回值：

无。

参考：

相关信息请参考 netShow、netBufLib 库描述，以及 netStackDataPoolShow()函数。

mbufShow()

函数原型：

```
void mbufShow (void)
```

功能描述：

显示 mbuf 统计。该函数显示网络中 mbuf 的分布。

返回值：

无。

参考：

相关信息请参考 netShow 库描述。

netShowInit()

函数原型：

```
void netShowInit (void)
```

功能描述：

初始化网络显示函数。该函数把网络显示模块链接到 VxWorks 系统中。如果在 configAll.h 文件中定义了 INCLUDE_NET_SHOW，则系统自动调用该函数。

返回值：

无。

参考：

相关信息请参考 netShow 库描述。

arpShow()

函数原型：

```
void arpShow (void)
```

功能描述：

显示系统 ARP 表中的条目。该函数显示 ARP 表中当前 Internet-to-Ethernet 地址映射。

返回值：

无。

参考：

相关信息请参考 netShow 库描述。

操作示范：

在 WindSh 下执行 arpShow 命令：

```
-> arpShow
LINK LEVEL ARP TABLE
destination        gateway              flags      Refcnt          Use        Interface
-----------------------------------------------------------------------------------------
90.0.0.63          08:00:3e:23:79:e7    405        0               82         lo0
-----------------------------------------------------------------------------------------
```

arptabShow()

函数原型：

```
void arptabShow (void)
```

功能描述：

显示知道的 ARP 条目。该函数显示 ARP 表中当前 Internet-to-Ethernet 地址映射。

返回值：

无。

参考：

相关信息请参考 netShow 库描述。

routestatShow()

函数原型：

```
void routestatShow (void)
```

功能描述：

该函数显示路由统计。

返回值：

无。

参考：

相关信息请参考 netShow 库描述。

routeShow()

函数原型：

```
void routeShow (void)
```

功能描述：

显示主机和网络路由表。该函数显示路由表中当前包含的路由信息。

操作示范：

例如，在 WindSh 下执行 routeShow：

```
-> routeShow
ROUTE NET TABLE
destination   gateway       flags     Refcnt     Use        Interface
-----------   -------       -----     ------     ---        ---------
90.0.0.0      90.0.0.63     1         1          142        enp0

ROUTE HOST TABLE
destination   gateway       flags     Refcnt     Use        Interface
-----------   -------       -----     ------     ---        ---------
127.0.0.1     127.0.0.1     5         0          82         lo0
```

标识域提供标识的十进制值，表 11-11 中描述了当前可用的标识值。

表 11-11　当前可用的标识值

值	描　述	值	描　述
0x1	可用路由	0x100	产生新的路由
0x2	目的地(destination)是一个网关	0x200	外部 daemon 转换名
0x4	主机指定的路由条目	0x400	由 ARP 产生
0x8	主机或网络不能到达的	0x800	手动添加(静态的)
0x10	动态创建(改变方向)	0x1000	恰好丢弃数据包(在更新期间)
0x20	动态修改(改变方向)	0x2000	通过管理协议修改
0x40	消息确定	0x4000	协议特殊的路由标识
0x80	当前子网掩码	0x8000	协议特殊的路由标识

返回值：

无。

参考：

相关信息请参考 netShow 库描述。

hostShow()

函数原型：

```
void hostShow (void)
```

功能描述：

显示主机表。该函数按 Internet 地址和别名顺序显示远程主机的列表。

返回值：

无。

参考：

相关信息请参考 netShow 库描述，以及 hostAdd()函数。

操作示范：

在 WindSh 下执行 hostShow：

```
-> hostAdd "test", "192.168.10.10"
value = 0 = 0x0
-> hostAdd "target", "192.168.10.10"
value = 0 = 0x0
-> hostShow
value = 0 = 0x0
显示的结果如下：
hostname        inet address        aliases
--------        ------------        -------
localhost       127.0.0.1
rainbow         192.168.10.10       test target
value = 0 = 0x0
```

mRouteShow()

函数原型：

```
void mRouteShow (void)
```

功能描述：

打印路由表的条目。该函数打印路由表中的路由条目。

返回值：

无。

参考：

相关信息请参考 netShow 库描述。

11.10 TCP 信息显示函数

11.10.1 函数库描述

1．库命名

TCP 信息显示函数库名称为 tcpShow。

2．函数

表 11-12 中列出了 TCP 信息显示函数。

表 11-12 TCP 信息显示函数

函 数	描 述
tcpShowInit()	初始化 TCP 显示函数
tcpDebugShow()	显示 TCP 的调试信息
tcpstatShow()	显示 TCP 的所有统计

3．描述

tcpShow 库提供的函数用来显示与 TCP 有关联的统计。

理解这些统计需要了解 Internet 网络协议的详细知识。有关协议信息，可以在下面列出的书中找到：

- TCP/IP Illustrated Volume II, The Implementationa, by Richard Stevens；
- The Design and Implementation of the 4.4 BSD UNIX Operating System, by Leffler, McKusick, Karels and Quarterman。

tcpShowInit()函数把 TCP 显示模块链接到 VxWorks 系统中。如果在 configAll.h 文件中定义了

INCLUDE_NET_SHOW，则系统自动执行该函数。

4．头文件

TCP 信息显示函数声明请查阅 netShow.h 头文件。

5．参考

相关信息请参考 netLib、netShow 库描述，以及“VxWorks Programmer's Guide”中的网络部分。

11.10.2　TCP 信息显示函数详细描述

tcpShowInit()

函数原型：

```
void tcpShowInit (void)
```

功能描述：

初始化 TCP 显示函数。该函数把 TCP 显示模块链接到 VxWorks 系统中。如果在 configAll.h 文件中定义了 INCLUDE_NET_SHOW，则系统自动执行该函数。

返回值：

无。

参考：

相关信息请参考 tcpShow 库描述。

tcpDebugShow()

函数原型：

```
void tcpDebugShow
(
int numPrint,       /* 打印条目数, 默认情况(0)显示 20 个条目 */
int verbose         /* 1 = 详细 */
)
```

功能描述：

显示 TCP 协议的调试信息。为了包含 TCP 调试模块，当构造系统映像的时候需要定义 INCLUDE_TCP_DEBUG。为了采集信息，打开相应套接字的 SO_DEBUG 选项。

返回值：

无。

参考：

相关信息请参考 tcpShow 库描述。

tcpstatShow()

函数原型：

```
void tcpstatShow (void)
```

功能描述：

显示 TCP 协议的所有统计。该函数显示 TCP 协议的详细统计。

返回值：

无。

参考：

相关信息请参考 tcpShow 库描述。

第 12 章　错误状态函数

12.1　错误状态函数

12.1.1　函数库描述

1．库命名

错误状态函数库名称为 errnoLib。

2．函数

表 12-1 中列出了错误状态函数。

表 12-1　错误状态函数

函　数	描　述
errnoGet()	获取当前任务错误状态值
errnoOfTaskGet()	获取指定任务的错误状态值
errnoSet()	设置当前任务错误状态值
errnoOfTaskSet()	设置指定任务的错误状态值

3．描述

该函数库包含了设置和获取任务或中断的错误状态值的函数。大多数的 VxWorks 函数在遇到错误的时候返回 ERROR 并且设置错误值以描述所遇到的错误。这个机制与 UNIX 下的错误状态机制相一致，错误值将会被设置到全局的变量 errno 中。同时，为支持多任务机制以及解决中断服务程序共享上下文空间问题，VxWorks 采用了下面两种增强的方式：

1）VxWorks 为在每一个任务的 TCB 中保存独立的 errno 值，并且在任务上下文切换的时候保存和恢复这个值。由于保存在 TCB 中，任务代码就可以随时地直接获取或设置 errno 值。

2）对于中断服务程序，VxWorks 将 errno 作为中断的进入和退出代码的一部分。所以中断服务程序也可以直接获取或设置 errno。

errno 机制用于 VxWorks 的错误报告。按照惯例，当被调用的程序产生了错误，调用者也会继续保留该错误值。推荐开发者在应用模块中也使用相同的机制。

错误状态值是一个四字节的整数。这个四字节整数的高位指出了出现错误的模块，称为模块号。低位则指出该模块所定义的错误代码。VxWorks 保留了模块号 0 以兼容 UNIX 的错误号定义。模块号的 1 到 500 指定了 VxWorks 模块，这些都定义在文件 vwModNum.h 中。该头文件中包含了 VxWorks 的每一个模块所对应的模块号。每一个 VxWorks 模块都包含有自身将会产生的错误值定义。大于 500 的模块号都是用于应用程序的。

VxWorks 可以包含被称为 statSymTbl 的特殊的符号表，用于打印错误信息。包含了该符号表之后，可以使用 printErrno()来打印错误值所对应的信息。

4．头文件

错误状态函数声明在 errnoLib.h 头文件中。

5．参考

相关信息请参考 errnoLib 库描述以及 printErrno()函数。

12.1.2　系统相关函数详细描述

errnoGet ()

函数原型：

int errnoGet (void)

功能描述：

该函数获取 errno 中所保存的错误状态值。为了向下兼容，也可以直接访问 errno 变量。

返回值：

errno 中所保存的错误状态值。

参考：

相关信息请参考 errnoLib 库描述，errnoSet()、errnoOfTaskGet()函数等。

errnoOfTaskGet ()

函数原型：

```
int errnoOfTaskGet
(
    int taskId    /* 任务 ID */
)
```

功能描述：

该函数获取指定任务最近设置的错误状态值。如果参数 taskId 为 0 则表示获取自身任务的错误状态值。

返回值：

返回指定任务的错误状态值，如果指定的任务不存在则返回 ERROR。

参考：

相关信息请参考 errnoLib 库描述，errnoSet()、errnoGet()函数。

errnoSet()

函数原型：

```
STATUS errnoSet
(
    int errorValue    /* 错误状态值 */
)
```

功能描述：

该函数将 errno 变量设置为指定的错误状态值。为了向下兼容，可以直接为 errno 变量赋值。

返回值：

成功设置则返回 OK，如果中断嵌套级别太多则返回 ERROR。

参考：

相关信息请参考 errnoLib 库描述和函数 errnoGet(), errnoOfTaskSet()的说明。

errnoOfTaskSet ()

函数原型：

```
STATUS errnoOfTaskSet
(
    int taskId,       /* 任务 ID */
    int errorValue    /* 错误状态值 */
)
```

功能描述：

该函数为指定的任务设置 errno 变量。如果参数 taskId 为 0 则表示设置自身任务的错误状态值。该函数主要用于调试，在一般的情况下，直接设置任务自身的错误状态值。

返回值：

成功设置则返回 OK，如果指定的任务不存在则返回 ERROR。

参考：

相关信息请参考 errnoLib 库描述，errnoSet()、errnoOfTaskGet()函数。

12.2 错误码速查列表

VxWorks 操作系统中，错误代码的定义都包含在各个模块中，而模块的错误代码标识则保存在文件 vwModNum.h 中。下面表 12-2 就是所有 VxWorks 系统定义的错误代码值。

表 12-2 VxWorks 系统定义的错误代码值

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
30065	196709	S_taskLib_NAME_NOT_FOUND
30066	196710	S_taskLib_TASK_HOOK_TABLE_FULL
30067	196711	S_taskLib_TASK_HOOK_NOT_FOUND
30068	196712	S_taskLib_TASK_SWAP_HOOK_REFERENCED
30069	196713	S_taskLib_TASK_SWAP_HOOK_SET
3006A	196714	S_taskLib_TASK_SWAP_HOOK_CLEAR
3006B	196715	S_taskLib_TASK_VAR_NOT_FOUND
3006C	196716	S_taskLib_TASK_UNDELAYED
3006D	196717	S_taskLib_ILLEGAL_PRIORITY
C0001	786433	S_ioLib_NO_DRIVER
C0002	786434	S_ioLib_UNKNOWN_REQUEST
C0003	786435	S_ioLib_DEVICE_ERROR
C0004	786436	S_ioLib_DEVICE_TIMEOUT
C0005	786437	S_ioLib_WRITE_PROTECTED
C0006	786438	S_ioLib_DISK_NOT_PRESENT
C0007	786439	S_ioLib_NO_FILENAME
C0008	786440	S_ioLib_CANCELLED
C0009	786441	S_ioLib_NO_DEVICE_NAME_IN_PATH
C000A	786442	S_ioLib_NAME_TOO_LONG
C000B	786443	S_ioLib_UNFORMATED
D0001	851969	S_iosLib_DEVICE_NOT_FOUND
D0002	851970	S_iosLib_DRIVER_GLUT
D0003	851971	S_iosLib_INVALID_FILE_DESCRIPTOR
D0004	851972	S_iosLib_TOO_MANY_OPEN_FILES
D0005	851973	S_iosLib_CONTROLLER_NOT_PRESENT
D0006	851974	S_iosLib_DUPLICATE_DEVICE_NAME
D0007	851975	S_iosLib_INVALID_ETHERNET_ADDRESS

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
E0001	917505	S_loadLib_ROUTINE_NOT_INSTALLED
E0002	917506	S_loadLib_TOO_MANY_SYMBOLS
110001	1114113	S_memLib_NOT_ENOUGH_MEMORY
110002	1114114	S_memLib_INVALID_NBYTES
110003	1114115	S_memLib_BLOCK_ERROR
110004	1114116	S_memLib_NO_PARTITION_DESTROY
110005	1114117	S_memLib_PAGE_SIZE_UNAVAILABLE
140001	1310721	S_rt11FsLib_VOLUME_NOT_AVAILABLE
140002	1310722	S_rt11FsLib_DISK_FULL
140003	1310723	S_rt11FsLib_FILE_NOT_FOUND
140004	1310724	S_rt11FsLib_NO_FREE_FILE_DESCRIPTORS
140005	1310725	S_rt11FsLib_INVALID_NUMBER_OF_BYTES
140006	1310726	S_rt11FsLib_FILE_ALREADY_EXISTS
140007	1310727	S_rt11FsLib_BEYOND_FILE_LIMIT
140008	1310728	S_rt11FsLib_INVALID_DEVICE_PARAMETERS
140009	1310729	S_rt11FsLib_NO_MORE_FILES_ALLOWED_ON_DISK
14000A	1310730	S_rt11FsLib_ENTRY_NUMBER_TOO_BIG
14000B	1310731	S_rt11FsLib_NO_BLOCK_DEVICE
160065	1441893	S_semLib_INVALID_STATE
160066	1441894	S_semLib_INVALID_OPTION
160067	1441895	S_semLib_INVALID_QUEUE_TYPE
160068	1441896	S_semLib_INVALID_OPERATION
1C0001	1835009	S_symLib_SYMBOL_NOT_FOUND
1C0002	1835010	S_symLib_NAME_CLASH
1C0003	1835011	S_symLib_TABLE_NOT_EMPTY
1C0004	1835012	S_symLib_SYMBOL_STILL_IN_TABLE
230001	2293761	S_usrLib_NOT_ENOUGH_ARGS
250001	2424833	S_remLib_ALL_PORTS_IN_USE
250002	2424834	S_remLib_RSH_ERROR
250003	2424835	S_remLib_IDENTITY_TOO_BIG
290001	2686977	S_netDrv_INVALID_NUMBER_OF_BYTES
290002	2686978	S_netDrv_SEND_ERROR
290003	2686979	S_netDrv_RECV_ERROR
290004	2686980	S_netDrv_UNKNOWN_REQUEST

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
290005	2686981	S_netDrv_BAD_SEEK
290006	2686982	S_netDrv_SEEK_PAST_EOF_ERROR
290007	2686983	S_netDrv_BAD_EOF_POSITION
290008	2686984	S_netDrv_SEEK_FATAL_ERROR
290009	2686985	S_netDrv_WRITE_ONLY_CANNOT_READ
29000A	2686986	S_netDrv_READ_ONLY_CANNOT_WRITE
29000B	2686987	S_netDrv_READ_ERROR
29000C	2686988	S_netDrv_WRITE_ERROR
29000D	2686989	S_netDrv_NO_SUCH_FILE_OR_DIR
29000E	2686990	S_netDrv_PERMISSION_DENIED
29000F	2686991	S_netDrv_IS_A_DIRECTORY
290010	2686992	S_netDrv_UNIX_FILE_ERROR
2B0001	2818049	S_inetLib_ILLEGAL_INTERNET_ADDRESS
2B0002	2818050	S_inetLib_ILLEGAL_NETWORK_NUMBER
2C0001	2883585	S_routeLib_ILLEGAL_INTERNET_ADDRESS
2C0002	2883586	S_routeLib_ILLEGAL_NETWORK_NUMBER
2D0001	2949121	S_nfsDrv_INVALID_NUMBER_OF_BYTES
2D0002	2949122	S_nfsDrv_BAD_FLAG_MODE
2D0003	2949123	S_nfsDrv_CREATE_NO_FILE_NAME
2D0004	2949124	S_nfsDrv_FATAL_ERR_FLUSH_INVALID_CACHE
2D0005	2949125	S_nfsDrv_WRITE_ONLY_CANNOT_READ
2D0006	2949126	S_nfsDrv_READ_ONLY_CANNOT_WRITE
2D0007	2949127	S_nfsDrv_NOT_AN_NFS_DEVICE
2D0008	2949128	S_nfsDrv_NO_HOST_NAME_SPECIFIED
2E0001	3014657	S_nfsLib_NFS_AUTH_UNIX_FAILED
2E0002	3014658	S_nfsLib_NFS_INAPPLICABLE_FILE_TYPE
2F0000	3080192	S_rpcLib_RPC_SUCCESS
2F0001	3080193	S_rpcLib_RPC_CANTENCODEARGS
2F0002	3080194	S_rpcLib_RPC_CANTDECODERES
2F0003	3080195	S_rpcLib_RPC_CANTSEND
2F0004	3080196	S_rpcLib_RPC_CANTRECV
2F0005	3080197	S_rpcLib_RPC_TIMEDOUT
2F0006	3080198	S_rpcLib_RPC_VERSMISMATCH
2F0007	3080199	S_rpcLib_RPC_AUTHERROR

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
2F0008	3080200	S_rpcLib_RPC_PROGUNAVAIL
2F0009	3080201	S_rpcLib_RPC_PROGVERSMISMATCH
2F000A	3080202	S_rpcLib_RPC_PROCUNAVAIL
2F000B	3080203	S_rpcLib_RPC_CANTDECODEARGS
2F000C	3080204	S_rpcLib_RPC_SYSTEMERROR
2F000D	3080205	S_rpcLib_RPC_UNKNOWNHOST
2F000E	3080206	S_rpcLib_RPC_PMAPFAILURE
2F000F	3080207	S_rpcLib_RPC_PROGNOTREGISTERED
2F0010	3080208	S_rpcLib_RPC_FAILED
300000	3145728	S_nfsLib_NFS_OK
300001	3145729	S_nfsLib_NFSERR_PERM
300002	3145730	S_nfsLib_NFSERR_NOENT
300005	3145733	S_nfsLib_NFSERR_IO
300006	3145734	S_nfsLib_NFSERR_NXIO
30000D	3145741	S_nfsLib_NFSERR_ACCES
300012	3145746	S_nfsLib_NFSERR_EXIST
300014	3145748	S_nfsLib_NFSERR_NODEV
300015	3145749	S_nfsLib_NFSERR_NOTDIR
300016	3145750	S_nfsLib_NFSERR_ISDIR
30001B	3145755	S_nfsLib_NFSERR_FBIG
30001C	3145756	S_nfsLib_NFSERR_NOSPC
30001E	3145758	S_nfsLib_NFSERR_ROFS
30003F	3145791	S_nfsLib_NFSERR_NAMETOOLONG
300042	3145794	S_nfsLib_NFSERR_NOTEMPTY
300045	3145797	S_nfsLib_NFSERR_DQUOT
300046	3145798	S_nfsLib_NFSERR_STALE
300063	3145827	S_nfsLib_NFSERR_WFLUSH
310001	3211265	S_errnoLib_NO_STAT_SYM_TBL
320001	3276801	S_hostLib_UNKNOWN_HOST
320002	3276802	S_hostLib_HOST_ALREADY_ENTERED
320003	3276803	S_hostLib_INVALID_PARAMETER
350001	3473409	S_if_sl_INVALID_UNIT_NUMBER
350002	3473410	S_if_sl_UNIT_UNINITIALIZED
350003	3473411	S_if_sl_UNIT_ALREADY_INITIALIZED

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
360001	3538945	S_loginLib_UNKNOWN_USER
360002	3538946	S_loginLib_USER_ALREADY_EXISTS
360003	3538947	S_loginLib_INVALID_PASSWORD
370001	3604481	S_scsiLib_DEV_NOT_READY
370002	3604482	S_scsiLib_WRITE_PROTECTED
370003	3604483	S_scsiLib_MEDIUM_ERROR
370004	3604484	S_scsiLib_HARDWARE_ERROR
370005	3604485	S_scsiLib_ILLEGAL_REQUEST
370006	3604486	S_scsiLib_BLANK_CHECK
370007	3604487	S_scsiLib_ABORTED_COMMAND
370008	3604488	S_scsiLib_VOLUME_OVERFLOW
370009	3604489	S_scsiLib_UNIT_ATTENTION
37000A	3604490	S_scsiLib_SELECT_TIMEOUT
37000B	3604491	S_scsiLib_LUN_NOT_PRESENT
37000C	3604492	S_scsiLib_ILLEGAL_BUS_ID
37000D	3604493	S_scsiLib_NO_CONTROLLER
37000E	3604494	S_scsiLib_REQ_SENSE_ERROR
37000F	3604495	S_scsiLib_DEV_UNSUPPORTED
370010	3604496	S_scsiLib_ILLEGAL_PARAMETER
370011	3604497	S_scsiLib_INVALID_PHASE
370012	3604498	S_scsiLib_ABORTED
370013	3604499	S_scsiLib_ILLEGAL_OPERATION
370014	3604500	S_scsiLib_DEVICE_EXIST
370015	3604501	S_scsiLib_DISCONNECTED
370016	3604502	S_scsiLib_BUS_RESET
370017	3604503	S_scsiLib_INVALID_TAG_TYPE
370018	3604504	S_scsiLib_SOFTWARE_ERROR
370019	3604505	S_scsiLib_NO_MORE_THREADS
37001A	3604506	S_scsiLib_UNKNOWN_SENSE_DATA
37001B	3604507	S_scsiLib_INVALID_BLOCK_SIZE
380001	3670017	S_dosFsLib_VOLUME_NOT_AVAILABLE
380002	3670018	S_dosFsLib_DISK_FULL
380003	3670019	S_dosFsLib_FILE_NOT_FOUND
380004	3670020	S_dosFsLib_NO_FREE_FILE_DESCRIPTORS

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
380005	3670021	S_dosFsLib_INVALID_NUMBER_OF_BYTES
380006	3670022	S_dosFsLib_FILE_ALREADY_EXISTS
380007	3670023	S_dosFsLib_ILLEGAL_NAME
380008	3670024	S_dosFsLib_CANT_DEL_ROOT
380009	3670025	S_dosFsLib_NOT_FILE
380009	3670025	S_dosFsLib_NOT_DIRECTORY
380009	3670025	S_dosFsLib_NOT_SAME_VOLUME
380009	3670025	S_dosFsLib_READ_ONLY
380009	3670025	S_dosFsLib_ROOT_DIR_FULL
380009	3670025	S_dosFsLib_DIR_NOT_EMPTY
380009	3670025	S_dosFsLib_BAD_DISK
380010	3670032	S_dosFsLib_NO_LABEL
380011	3670033	S_dosFsLib_INVALID_PARAMETER
380012	3670034	S_dosFsLib_NO_CONTIG_SPACE
380013	3670035	S_dosFsLib_CANT_CHANGE_ROOT
380014	3670036	S_dosFsLib_FD_OBSOLETE
380015	3670037	S_dosFsLib_DELETED
380016	3670038	S_dosFsLib_NO_BLOCK_DEVICE
380017	3670039	S_dosFsLib_BAD_SEEK
380018	3670040	S_dosFsLib_INTERNAL_ERROR
380019	3670041	S_dosFsLib_WRITE_ONLY
390001	3735553	S_selectLib_NO_SELECT_SUPPORT_IN_DRIVER
3A0001	3801089	S_hashLib_KEY_CLASH
3B0001	3866625	S_qLib_Q_CLASS_ID_ERROR
3D0001	3997697	S_objLib_OBJ_ID_ERROR
3D0002	3997698	S_objLib_OBJ_UNAVAILABLE
3D0003	3997699	S_objLib_OBJ_DELETED
3D0004	3997700	S_objLib_OBJ_TIMEOUT
3D0005	3997701	S_objLib_OBJ_NO_METHOD
3E0001	4063233	S_qPriHeapLib_NULL_HEAP_ARRAY
3F0001	4128769	S_qPriBMapLib_NULL_BMAP_LIST
410001	4259841	S_msgQLib_INVALID_MSG_LENGTH
410002	4259842	S_msgQLib_NON_ZERO_TIMEOUT_AT_INT_LEVEL
410003	4259843	S_msgQLib_INVALID_QUEUE_TYPE

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
420001	4325377	S_classLib_CLASS_ID_ERROR
420002	4325378	S_classLib_NO_CLASS_DESTROY
430001	4390913	S_intLib_NOT_ISR_CALLABLE
430002	4390914	S_intLib_VEC_TABLE_WP_UNAVAILABLE
450001	4521985	S_cacheLib_INVALID_CACHE
460001	4587521	S_rawFsLib_VOLUME_NOT_AVAILABLE
460002	4587522	S_rawFsLib_END_OF_DEVICE
460003	4587523	S_rawFsLib_NO_FREE_FILE_DESCRIPTORS
460004	4587524	S_rawFsLib_INVALID_NUMBER_OF_BYTES
460005	4587525	S_rawFsLib_ILLEGAL_NAME
460006	4587526	S_rawFsLib_NOT_FILE
460007	4587527	S_rawFsLib_READ_ONLY
460008	4587528	S_rawFsLib_FD_OBSOLETE
460009	4587529	S_rawFsLib_NO_BLOCK_DEVICE
46000A	4587530	S_rawFsLib_BAD_SEEK
470001	4653057	S_arpLib_INVALID_ARGUMENT
470002	4653058	S_arpLib_INVALID_ENET_ADDRESS
470003	4653059	S_arpLib_INVALID_FLAG
480001	4718593	S_smLib_MEMORY_ERROR
480002	4718594	S_smLib_INVALID_CPU_NUMBER
480003	4718595	S_smLib_NOT_ATTACHED
480004	4718596	S_smLib_NO_REGIONS
490001	4784129	S_bootpLib_INVALID_ARGUMENT
490002	4784130	S_bootpLib_INVALID_COOKIE
490003	4784131	S_bootpLib_NO_BROADCASTS
490004	4784132	S_bootpLib_PARSE_ERROR
490005	4784133	S_bootpLib_INVALID_TAG
490006	4784134	S_bootpLib_TIME_OUT
4A0001	4849665	S_icmpLib_TIMEOUT
4A0002	4849666	S_icmpLib_NO_BROADCAST
4A0003	4849667	S_icmpLib_INVALID_INTERFACE
004B0001	4915201	S_tftpLib_INVALID_ARGUMENT
004B0002	4915202	S_tftpLib_INVALID_DESCRIPTOR
004B0003	4915203	S_tftpLib_INVALID_COMMAND

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
004B0004	4915204	S_tftpLib_INVALID_MODE
004B0005	4915205	S_tftpLib_UNKNOWN_HOST
004B0006	4915206	S_tftpLib_NOT_CONNECTED
004B0007	4915207	S_tftpLib_TIMED_OUT
004B0008	4915208	S_tftpLib_TFTP_ERROR
4E0001	5111809	S_smPktLib_SHARED_MEM_TOO_SMALL
4E0002	5111810	S_smPktLib_MEMORY_ERROR
4E0003	5111811	S_smPktLib_DOWN
4E0004	5111812	S_smPktLib_NOT_ATTACHED
4E0005	5111813	S_smPktLib_INVALID_PACKET
4E0006	5111814	S_smPktLib_PACKET_TOO_BIG
4E0007	5111815	S_smPktLib_INVALID_CPU_NUMBER
4E0008	5111816	S_smPktLib_DEST_NOT_ATTACHED
4E0009	5111817	S_smPktLib_INCOMPLETE_BROADCAST
4E000A	5111818	S_smPktLib_LIST_FULL
4E000B	5111819	S_smPktLib_LOCK_TIMEOUT
4C0001	4980737	S_proxyArpLib_INVALID_PARAMETER
4C0002	4980738	S_proxyArpLib_INVALID_INTERFACE
4C0003	4980739	S_proxyArpLib_INVALID_PROXY_NET
4C0004	4980740	S_proxyArpLib_INVALID_CLIENT
4C0005	4980741	S_proxyArpLib_INVALID_ADDRESS
4C0006	4980742	S_proxyArpLib_TIMEOUT
500001	5242881	S_loadAoutLib_TOO_MANY_SYMBOLS
520001	5373953	S_bootLoadLib_ROUTINE_NOT_INSTALLED
530001	5439489	S_loadLib_FILE_READ_ERROR
530002	5439490	S_loadLib_REALLOC_ERROR
530003	5439491	S_loadLib_JMPADDR_ERROR
530004	5439492	S_loadLib_NO_REFLO_PAIR
530005	5439493	S_loadLib_GPREL_REFERENCE
530006	5439494	S_loadLib_UNRECOGNIZED_RELOCENTRY
530007	5439495	S_loadLib_REFHALF_OVFL
530008	5439496	S_loadLib_FILE_ENDIAN_ERROR
530009	5439497	S_loadLib_UNEXPECTED_SYM_CLASS
53000A	5439498	S_loadLib_UNRECOGNIZED_SYM_CLASS

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
53000B	5439499	S_loadLib_HDR_READ
53000C	5439500	S_loadLib_OPTHDR_READ
53000D	5439501	S_loadLib_SCNHDR_READ
53000E	5439502	S_loadLib_READ_SECTIONS
53000F	5439503	S_loadLib_LOAD_SECTIONS
530010	5439504	S_loadLib_RELOC_READ
530011	5439505	S_loadLib_SYMHDR_READ
530012	5439506	S_loadLib_EXTSTR_READ
530013	5439507	S_loadLib_EXTSYM_READ
530014	5439508	S_loadLib_NO_RELOCATION_ROUTINE
540001	5505025	S_vmLib_NOT_PAGE_ALIGNED
540002	5505026	S_vmLib_BAD_STATE_PARAM
540003	5505027	S_vmLib_BAD_MASK_PARAM
540004	5505028	S_vmLib_ADDR_IN_GLOBAL_SPACE
540005	5505029	S_vmLib_TEXT_PROTECTION_UNAVAILABLE
550001	5570561	S_mmuLib_INVALID_PAGE_SIZE
550002	5570562	S_mmuLib_NO_DESCRIPTOR
550003	5570563	S_mmuLib_INVALID_DESCRIPTOR
550005	5570565	S_mmuLib_OUT_OF_PMEGS
550006	5570566	S_mmuLib_VIRT_ADDR_OUT_OF_BOUNDS
560001	5636097	S_moduleLib_MODULE_NOT_FOUND
560002	5636098	S_moduleLib_HOOK_NOT_FOUND
560003	5636099	S_moduleLib_BAD_CHECKSUM
560004	5636100	S_moduleLib_MAX_MODULES_LOADED
570001	5701633	S_unldLib_MODULE_NOT_FOUND
570002	5701634	S_unldLib_TEXT_IN_USE
580001	5767169	S_smObjLib_NOT_INITIALIZED
580002	5767170	S_smObjLib_NOT_A_GLOBAL_ADRS
580003	5767171	S_smObjLib_NOT_A_LOCAL_ADRS
580004	5767172	S_smObjLib_SHARED_MEM_TOO_SMALL
580005	5767173	S_smObjLib_TOO_MANY_CPU
580006	5767174	S_smObjLib_LOCK_TIMEOUT
580007	5767175	S_smObjLib_NO_OBJECT_DESTROY
590001	5832705	S_smNameLib_NOT_INITIALIZED

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
590002	5832706	S_smNameLib_NAME_TOO_LONG
590003	5832707	S_smNameLib_NAME_NOT_FOUND
590004	5832708	S_smNameLib_VALUE_NOT_FOUND
590005	5832709	S_smNameLib_NAME_ALREADY_EXIST
590006	5832710	S_smNameLib_DATABASE_FULL
590007	5832711	S_smNameLib_INVALID_WAIT_TYPE
5B0001	5963777	S_m2Lib_INVALID_PARAMETER
5B0002	5963778	S_m2Lib_ENTRY_NOT_FOUND
5B0003	5963779	S_m2Lib_TCPCONN_FD_NOT_FOUND
5B0004	5963780	S_m2Lib_INVALID_VAR_TO_SET
5B0005	5963781	S_m2Lib_CANT_CREATE_SYS_SEM
5B0006	5963782	S_m2Lib_CANT_CREATE_IF_SEM
5B0007	5963783	S_m2Lib_CANT_CREATE_ROUTE_SEM
5B0008	5963784	S_m2Lib_ARP_PHYSADDR_NOT_SPECIFIED
5B0009	5963785	S_m2Lib_IF_TBL_IS_EMPTY
5B000A	5963786	S_m2Lib_IF_CNFG_CHANGED
5B000B	5963787	S_m2Lib_TOO_BIG
5B000C	5963788	S_m2Lib_BAD_VALUE
5B000D	5963789	S_m2Lib_READ_ONLY
5B000E	5963790	S_m2Lib_GEN_ERR
5C0001	6029313	S_aioPxLib_DRV_NUM_INVALID
5C0002	6029314	S_aioPxLib_AIO_NOT_DEFINED
5C0003	6029315	S_aioPxLib_IOS_NOT_INITIALIZED
5C0004	6029316	S_aioPxLib_NO_AIO_DEVICE
5E0001	6160385	S_mountLib_ILLEGAL_MODE
600001	6291457	S_loadSomCoffLib_HDR_READ
600002	6291458	S_loadSomCoffLib_AUXHDR_READ
600003	6291459	S_loadSomCoffLib_SYM_READ
600004	6291460	S_loadSomCoffLib_SYMSTR_READ
600005	6291461	S_loadSomCoffLib_OBJ_FMT
600006	6291462	S_loadSomCoffLib_SPHDR_ALLOC
600007	6291463	S_loadSomCoffLib_SPHDR_READ
600008	6291464	S_loadSomCoffLib_SUBSPHDR_ALLOC
600009	6291465	S_loadSomCoffLib_SUBSPHDR_READ

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
60000A	6291466	S_loadSomCoffLib_SPSTRING_ALLOC
60000B	6291467	S_loadSomCoffLib_SPSTRING_READ
60000C	6291468	S_loadSomCoffLib_INFO_ALLOC
60000D	6291469	S_loadSomCoffLib_LOAD_SPACE
60000E	6291470	S_loadSomCoffLib_RELOC_ALLOC
60000F	6291471	S_loadSomCoffLib_RELOC_READ
600010	6291472	S_loadSomCoffLib_RELOC_NEW
600011	6291473	S_loadSomCoffLib_RELOC_VERSION
610001	6356993	S_loadElfLib_HDR_READ
610002	6356994	S_loadElfLib_HDR_ERROR
610003	6356995	S_loadElfLib_PHDR_MALLOC
610004	6356996	S_loadElfLib_PHDR_READ
610005	6356997	S_loadElfLib_SHDR_MALLOC
610006	6356998	S_loadElfLib_SHDR_READ
610007	6356999	S_loadElfLib_READ_SECTIONS
610008	6357000	S_loadElfLib_LOAD_SECTIONS
610009	6357001	S_loadElfLib_LOAD_PROG
61000A	6357002	S_loadElfLib_SYMTAB
61000B	6357003	S_loadElfLib_RELA_SECTION
61000C	6357004	S_loadElfLib_SCN_READ
61000D	6357005	S_loadElfLib_SDA_MALLOC
61000F	6357007	S_loadElfLib_SST_READ
610014	6357012	S_loadElfLib_JMPADDR_ERROR
610015	6357013	S_loadElfLib_GPREL_REFERENCE
610016	6357014	S_loadElfLib_UNRECOGNIZED_RELOCENTRY
610017	6357015	S_loadElfLib_RELOC
610018	6357016	S_loadElfLib_UNSUPPORTED
610019	6357017	S_loadElfLib_REL_SECTION
620001	6422529	S_mbufLib_ID_INVALID
620002	6422530	S_mbufLib_ID_EMPTY
620003	6422531	S_mbufLib_SEGMENT_NOT_FOUND
620004	6422532	S_mbufLib_LENGTH_INVALID
620005	6422533	S_mbufLib_OFFSET_INVALID
630001	6488065	S_pingLib_NOT_INITIALIZED

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
630002	6488066	S_pingLib_TIMEOUT
640001	6553601	S_strmLib_DUPLICATE_PROVIDER
640002	6553602	S_strmLib_INVALID_OPEN_STYLE
640003	6553603	S_strmLib_INVALID_SYNC_LEVEL
650001	6619137	S_pppSecretLib_NOT_INITIALIZED
650002	6619138	S_pppSecretLib_SECRET_DOES_NOT_EXIST
650003	6619139	S_pppSecretLib_SECRET_EXISTS
660001	6684673	S_pppHookLib_NOT_INITIALIZED
660002	6684674	S_pppHookLib_HOOK_DELETED
660003	6684675	S_pppHookLib_HOOK_ADDED
660004	6684676	S_pppHookLib_HOOK_UNKNOWN
660005	6684677	S_pppHookLib_INVALID_UNIT
670001	6750209	S_tapeFsLib_NO_SEQ_DEV
670002	6750210	S_tapeFsLib_ILLEGAL_TAPE_CONFIG_PARM
670003	6750211	S_tapeFsLib_SERVICE_NOT_AVAILABLE
670004	6750212	S_tapeFsLib_INVALID_BLOCK_SIZE
670005	6750213	S_tapeFsLib_ILLEGAL_FILE_SYSTEM_NAME
670006	6750214	S_tapeFsLib_ILLEGAL_FLAGS
670007	6750215	S_tapeFsLib_FILE_DESCRIPTOR_BUSY
670008	6750216	S_tapeFsLib_VOLUME_NOT_AVAILABLE
670009	6750217	S_tapeFsLib_BLOCK_SIZE_MISMATCH
67000A	6750218	S_tapeFsLib_INVALID_NUMBER_OF_BYTES
680001	6815745	S_snmpdLib_VIEW_CREATE_FAILURE
680002	6815746	S_snmpdLib_VIEW_INSTALL_FAILURE
680003	6815747	S_snmpdLib_VIEW_MASK_FAILURE
680004	6815748	S_snmpdLib_VIEW_DEINSTALL_FAILURE
680005	6815749	S_snmpdLib_VIEW_LOOKUP_FAILURE
680006	6815750	S_snmpdLib_MIB_ADDITION_FAILURE
680007	6815751	S_snmpdLib_NODE_NOT_FOUND
680008	6815752	S_snmpdLib_INVALID_SNMP_VERSION
680009	6815753	S_snmpdLib_TRAP_CREATE_FAILURE
68000A	6815754	S_snmpdLib_TRAP_BIND_FAILURE
68000B	6815755	S_snmpdLib_TRAP_ENCODE_FAILURE
68000C	6815756	S_snmpdLib_INVALID_OID_SYNTAX

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
690001	6881281	S_pcmciaLib_BATTERY_DEAD
690002	6881282	S_pcmciaLib_BATTERY_WARNING
6A0001	6946817	S_dhcpcLib_NOT_INITIALIZED
6A0002	6946818	S_dhcpcLib_NO_DEVICE
6A0003	6946819	S_dhcpcLib_MAX_LEASES_REACHED
6A0004	6946820	S_dhcpcLib_MEM_ERROR
6A0005	6946821	S_dhcpcLib_BAD_COOKIE
6A0006	6946822	S_dhcpcLib_NOT_BOUND
6A0007	6946823	S_dhcpcLib_BAD_OPTION
6A0008	6946824	S_dhcpcLib_OPTION_NOT_PRESENT
6A0009	6946825	S_dhcpcLib_TIMER_ERROR
6B0001	7012353	S_resolvLib_NO_RECOVERY
6B0002	7012354	S_resolvLib_TRY_AGAIN
6B0003	7012355	S_resolvLib_HOST_NOT_FOUND
6B0004	7012356	S_resolvLib_NO_DATA
6B0005	7012357	S_resolvLib_BUFFER_2_SMALL
6B0006	7012358	S_resolvLib_INVALID_PARAMETER
6D0001	7143425	S_muxLib_LOAD_FAILED
6D0002	7143426	S_muxLib_NO_DEVICE
6D0003	7143427	S_muxLib_INVALID_ARGS
6D0004	7143428	S_muxLib_ALLOC_FAILED
6D0005	7143429	S_muxLib_ALREADY_BOUND
6D0006	7143430	S_muxLib_UNLOAD_FAILED
6E0001	7208961	S_m2RipLib_IFACE_NOT_FOUND
6F0001	7274497	S_ospfLib_MCAST_GROUP_FAILED
6F0002	7274498	S_ospfLib_LOOPBACK_DISABLE_FAILED
6F0003	7274499	S_ospfLib_MCAST_IF_SELECT_FAILED
6F0004	7274500	S_ospfLib_CIRC_CREATE_FAILED
6F0005	7274501	S_ospfLib_NBMA_DST_ADD_FAILED
6F0006	7274502	S_ospfLib_NBMA_DST_DELETE_FAILED
6F0007	7274503	S_ospfLib_EXT_ROUTE_ADD_FAILED
6F0008	7274504	S_ospfLib_EXT_ROUTE_DELETE_FAILED
700001	7340033	S_dhcpsLib_NOT_INITIALIZED
710001	7405569	S_sntpcLib_INVALID_PARAMETER

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
710002	7405570	S_sntpcLib_INVALID_ADDRESS
710003	7405571	S_sntpcLib_TIMEOUT
710004	7405572	S_sntpcLib_VERSION_UNSUPPORTED
710005	7405573	S_sntpcLib_SERVER_UNSYNC
720001	7471105	S_sntpsLib_INVALID_PARAMETER
720002	7471106	S_sntpsLib_INVALID_ADDRESS
730001	7536641	S_netBufLib_MEMSIZE_INVALID
730002	7536642	S_netBufLib_CLSIZE_INVALID
730003	7536643	S_netBufLib_NO_SYSTEM_MEMORY
730004	7536644	S_netBufLib_MEM_UNALIGNED
730005	7536645	S_netBufLib_MEMSIZE_UNALIGNED
730006	7536646	S_netBufLib_MEMAREA_INVALID
730007	7536647	S_netBufLib_MBLK_INVALID
730008	7536648	S_netBufLib_NETPOOL_INVALID
730009	7536649	S_netBufLib_INVALID_ARGUMENT
73000A	7536650	S_netBufLib_NO_POOL_MEMORY
740001	7602177	S_cdromFsLib_ALREADY_INIT
740003	7602179	S_cdromFsLib_DEVICE_REMOVED
740004	7602180	S_cdromFsLib_SUCH_PATH_TABLE_SIZE_NOT_SUPPORTED
740005	7602181	S_cdromFsLib_ONE_OF_VALUES_NOT_POWER_OF_2
740006	7602182	S_cdromFsLib_UNNOWN_FILE_SYSTEM
740007	7602183	S_cdromFsLib_INVAL_VOL_DESCR
740008	7602184	S_cdromFsLib_INVALID_PATH_STRING
740009	7602185	S_cdromFsLib_MAX_DIR_HIERARCHY_LEVEL_OVERFLOW
74000A	7602186	S_cdromFsLib_NO_SUCH_FILE_OR_DIRECTORY
74000B	7602187	S_cdromFsLib_INVALID_DIRECTORY_STRUCTURE
74000C	7602188	S_cdromFsLib_INVALID_FILE_DESCRIPTOR
74000D	7602189	S_cdromFsLib_READ_ONLY_DEVICE
74000E	7602190	S_cdromFsLib_END_OF_FILE
74000F	7602191	S_cdromFsLib_INV_ARG_VALUE
740010	7602192	S_cdromFsLib_SEMTAKE_ERROR
740011	7602193	S_cdromFsLib_SEMGIVE_ERROR
740012	7602194	S_cdromFsLib_VOL_UNMOUNTED
740013	7602195	S_cdromFsLib_INVAL_DIR_OPER

（续）

错误代码（16 进制）	错误代码（10 进制）	错误代码值宏定义
740014	7602196	S_cdromFsLib_READING_FAILURE
740015	7602197	S_cdromFsLib_INVALID_DIR_REC_STRUCT
750001	7667713	S_loadLib_FILE_READ_ERROR
750002	7667714	S_loadLib_REALLOC_ERROR
750003	7667715	S_loadLib_JMPADDR_ERROR
750004	7667716	S_loadLib_NO_REFLO_PAIR
750005	7667717	S_loadLib_GPREL_REFERENCE
750006	7667718	S_loadLib_UNRECOGNIZED_RELOCENTRY
750007	7667719	S_loadLib_REFHALF_OVFL
750008	7667720	S_loadLib_FILE_ENDIAN_ERROR
750009	7667721	S_loadLib_UNEXPECTED_SYM_CLASS
75000A	7667722	S_loadLib_UNRECOGNIZED_SYM_CLASS
75000B	7667723	S_loadLib_HDR_READ
75000C	7667724	S_loadLib_OPTHDR_READ
75000D	7667725	S_loadLib_SCNHDR_READ
75000E	7667726	S_loadLib_READ_SECTIONS
75000F	7667727	S_loadLib_LOAD_SECTIONS
750010	7667728	S_loadLib_RELOC_READ
750011	7667729	S_loadLib_SYMHDR_READ
750012	7667730	S_loadLib_EXTSTR_READ
750013	7667731	S_loadLib_EXTSYM_READ
760001	7733249	S_distLib_NOT_INITIALIZED
760002	7733250	S_distLib_NO_OBJECT_DESTROY
760003	7733251	S_distLib_UNREACHABLE
760004	7733252	S_distLib_UNKNOWN_REQUEST
760005	7733253	S_distLib_OBJ_ID_ERROR
7A0001	7995393	S_if_ul_INVALID_UNIT_NUMBER
7A0002	7995394	S_if_ul_UNIT_UNINITIALIZED
7A0003	7995395	S_if_ul_UNIT_ALREADY_INITIALIZED
7A0004	7995396	S_if_ul_NO_UNIX_DEVICE

附　录

在VxWorks系统中，有关体系结构的硬件异常、信号、附加码之间的关系请参考下列各表格。

附表 1　Motorola 68K

信　号	附加代码	异常描述
SIGSEGV	NULL	bus error
SIGBUS	BUS_ADDERR	address error
SIGILL	ILL_ILLINSTR_FAULT	illegal instruction
SIGFPE	FPE_INTDIV_TRAP	zero divide
SIGFPE	FPE_CHKINST_TRAP	chk trap
SIGFPE	FPE_TRAPV_TRAP	trapv trap
SIGILL	ILL_PRIVVIO_FAULT	privilege violation
SIGTRAP	NULL	trace exception
SIGEMT	EMT_EMU1010	line 1010 emulator
SIGEMT	EMT_EMU1111	line 1111 emulator
SIGILL	ILL_ILLINSTR_FAULT	coprocessor protocol violation
SIGFMT	NULL	format error
SIGFPE	FPE_FLTBSUN_TRAP	compare unordered
SIGFPE	FPE_FLTINEX_TRAP	inexact result
SIGFPE	FPE_FLTDIV_TRAP	divide by zero
SIGFPE	FPE_FLTUND_TRAP	underflow
SIGFPE	FPE_FLTOPERR_TRAP	operand error
SIGFPE	FPE_FLTOVF_TRAP	overflow
SIGFPE	FPE_FLTNAN_TRAP	signaling "Not A Number"

附表 2　SPARC

信　号	附加代码	异常描述
SIGBUS	BUS_INSTR_ACCESS	bus error on instruction fetch
SIGBUS	BUS_ALIGN	address error (bad alignment)
SIGBUS	BUS_DATA_ACCESS	bus error on data access
SIGILL	ILL_ILLINSTR_FAULT	illegal instruction
SIGILL	ILL_PRIVINSTR_FAULT	privilege violation
SIGILL	ILL_COPROC_DISABLED	coprocessor disabled
SIGILL	ILL_COPROC_EXCPTN	coprocessor exception
SIGILL	ILL_TRAP_FAULT(n)	uninitialized user trap

（续）

信　号	附加代码	异常描述
SIGFPE	FPE_FPA_ENABLE	floating point disabled
SIGFPE	FPE_FPA_ERROR	floating point exception
SIGFPE	FPE_INTDIV_TRAP	zero divide
SIGEMT	EMT_TAG	tag overflow

附表 3　Intel i960

信　号	附加代码	异常描述
SIGBUS	BUS_UNALIGNED	address error (bad alignment)
SIGBUS	BUS_BUSERR	bus error
SIGILL	ILL_INVALID_OPCODE	invalid instruction
SIGILL	ILL_UNIMPLEMENTED	instr fetched from on-chip RAM
SIGILL	ILL_INVALID_OPERAND	invalid operand
SIGILL	ILL_CONSTRAINT_RANGE	constraint range failure
SIGILL	ILL_PRIVILEGED	privilege violation
SIGILL	ILL_LENGTH	bad index to sys procedure table
SIGILL	ILL_TYPE_MISMATCH	privilege violation
SIGTRAP	TRAP_INSTRUCTION_TRACE	instruction trace fault
SIGTRAP	TRAP_BRANCH_TRACE	branch trace fault
SIGTRAP	TRAP_CALL_TRACE	call trace fault
SIGTRAP	TRAP_RETURN_TRACE	return trace fault
SIGTRAP	TRAP_PRERETURN_TRACE	pre-return trace fault
SIGTRAP	TRAP_SUPERVISOR_TRACE	supervisor trace fault
SIGTRAP	TRAP_BREAKPOINT_TRACE	breakpoint trace fault
SIGFPE	FPE_INTEGER_OVERFLOW	integer overflow
SIGFPE	FST_ZERO_DIVIDE	integer zero divide
SIGFPE	FPE_FLOATING_OVERFLOW	floating point overflow
SIGFPE	FPE_FLOATING_UNDERFLOW	floating point underflow
SIGFPE	FPE_FLOATING_INVALID_OPERATION	invalid floating point operation
SIGFPE	FPE_FLOATING_ZERO_DIVIDE	floating point zero divide
SIGFPE	FPE_FLOATING_INEXACT	floating point inexact
SIGFPE	FPE_FLOATING_RESERVED_ENCODING	floating point reserved encoding

附表 4　MIPS R3000/R4000

信　号	附加代码	异常描述
SIGBUS	BUS_TLBMOD	TLB modified
SIGBUS	BUS_TLBL	TLB miss on a load instruction

（续）

信 号	附加代码	异常描述
SIGBUS	BUS_TLBS	TLB miss on a store instruction
SIGBUS	BUS_ADEL	address error (bad alignment) on load instr
SIGBUS	BUS_ADES	address error (bad alignment) on store instr
SIGSEGV	SEGV_IBUS	bus error (instruction)
SIGSEGV	SEGV_DBUS	bus error (data)
SIGTRAP	TRAP_SYSCALL	syscall instruction executed
SIGTRAP	TRAP_BP	break instruction executed
SIGILL	ILL_ILLINSTR_FAULT	reserved instruction
SIGILL	ILL_COPROC_UNUSABLE	coprocessor unusable
SIGFPE	FPE_FPA_UIO, SIGFPE	unimplemented FPA operation
SIGFPE	FPE_FLTNAN_TRAP	invalid FPA operation
SIGFPE	FPE_FLTDIV_TRAP	FPA divide by zero
SIGFPE	FPE_FLTOVF_TRAP	FPA overflow exception
SIGFPE	FPE_FLTUND_TRAP	FPA underflow exception
SIGFPE	FPE_FLTINEX_TRAP	FPA inexact operation

附表 5 Intel i386/486

信 号	附加代码	异常描述
SIGILL	ILL_DIVIDE_ERROR	divide error
SIGEMT	EMT_DEBUG	debugger call
SIGILL	ILL_NON_MASKABLE	NMI interrupt
SIGEMT	EMT_BREAKPOINT	breakpoint
SIGILL	ILL_OVERFLOW	INTO-detected overflow
SIGILL	ILL_BOUND	bound range exceeded
SIGILL	ILL_INVALID_OPCODE	invalid opcode
SIGFPE	FPE_NO_DEVICE	device not available
SIGILL	ILL_DOUBLE_FAULT	double fault
SIGFPE	FPE_CP_OVERRUN	coprocessor segment overrun
SIGILL	ILL_INVALID_TSS	invalid task state segment
SIGBUS	BUS_NO_SEGMENT	segment not present
SIGBUS	BUS_STACK_FAULT	stack exception
SIGILL	ILL_PROTECTION_FAULT	general protection
SIGBUS	BUS_PAGE_FAULT	page fault
SIGILL	ILL_RESERVED	(intel reserved)
SIGFPE	FPE_CP_ERROR	coprocessor error
SIGBUS	BUS_ALIGNMENT	alignment check

附表 6 PowerPC

信　号	附加代码	异常描述
SIGBUS	_EXC_OFF_MACH	machine check
SIGBUS	_EXC_OFF_INST	instruction access
SIGBUS	_EXC_OFF_ALIGN	alignment
SIGILL	_EXC_OFF_PROG	program
SIGBUS	_EXC_OFF_DATA	data access
SIGFPE	_EXC_OFF_FPU	floating point unavailable
SIGTRAP	_EXC_OFF_DBG	debug exception (PPC403)
SIGTRAP	_EXC_OFF_INST_BRK	inst. breakpoint (PPC603,PPCEC603,PPC604)
SIGTRAP	_EXC_OFF_TRACE	trace (PPC603, PPCEC603,PPC604,PPC860)
SIGBUS	_EXC_OFF_CRTL	critical interrupt (PPC403)
SIGILL	_EXC_OFF_SYSCALL	system call