**https://blog.csdn.net/redseazhaojianertao/article/details/53843548?utm\_source=blogxgwz4**

**VxWorks内核解读-7**

2016年12月23日 20:20:05 [redseazhaojianertao](https://me.csdn.net/redseazhaojianertao" \t "_blank) 阅读数：3582

为了实现应用层程序的平台无关性，操作系统为应用层提供了一套标准的接口函数，这些接口函数在所有的平台上都保持一致，只是随着平台的变化，底层驱动或接近驱动部分操作系统中间层可能会随着调整。这样可以使用户程序独立于具体的硬件平台，增加了应用层开发的效率，避免了重复编码。通用操作系统GPOS(General Purpose Operating System)比如Unix,Linux，将这套提供给应用层的标准接口函数从操作系统中独立出来，专门以标准库的形式存在，增加了应用程序的平台无关性，平台之间的差别完全被操作系统屏蔽。

和Unix,Linux类似，VxWorks也对应用层提供了一套标准文件操作接口函数，实际上与GPOS提供接口类似，我们将其称作为标准I/O 库，VxWorks 下由ioLib.c 文件提供。ioLib.c 文件提供如下标准接口函数：creat、open、unlink、remove、close、rename、read、write、ioctl、lseek、readv、writev 等。VxWorks 操作系统区别于GPOS的一个很大不同点是：VxWorks 不区分用户态和内核态，用户层程序可以直接对内核函数进行调用，而无须使用陷阱指令之类的机制，以及存在使用权限上的限制。因此VxWorks 提供给应用层的接口无须通过外围库的方式，而是直接以内核文件的形式提供。用户程序可以直接使用ioLib.c 文件定义的如上这些函数，这些函数名称与GPOS标准库下的函数名一致，是VxWorks 对标准库的模拟。

VxWorks的I/O系统给应用程序提供了简单、统一、与下列设备无关的接口：

1. 面向终端的字符设备或者通信线路设备；
2. 随机访问块设备，例如硬盘；
3. 虚拟设备，例如管道和套接字；
4. 用于监控和控制的数据或者模拟I/O设备；
5. 访问远程设备的网络设备。

在VxWorks系统中，应用程序通过文件来访问I/O设备，一个文件通常代表以下两类设备：

1.  无结构的原始(raw)设备，比如串口通信通道，用于任务间通信的管道设备；

2.  包含文件系统的的有结构的、可随机访问设备上的逻辑文件；

例如下面的文件：

/usr/myfile

/pipe/mypipe

/tyCo/0

第一个文件myfile在一个名为/usr的硬盘上；第二个在命名管道上，通常管道以/pipe开始；第三个文件映射到串行通道。I/O系统可以采用文件的方式处理这些设备，在VxWorks中尽管这些设备物理性质诧异巨大，但是它们都称为文件，在这一点上借鉴了Unix的设计思想。

VxWorks系统中的设备由程序模块来处理，这些程序模块被称为驱动(Driver)；使用I/O系统并不需要了解这些驱动和设备的具体实现机制，但VxWorks的I/O系统赋予特定设备的驱动极大的灵活性，尽管所有的I/O系统均是以文件来索引，但是却有两种不同的实现方式：基本I/O实现和带缓存的I/O设想，这两种实现方式的差别在于数据是否缓存、以及系统调用实现的方式，如图1所示。

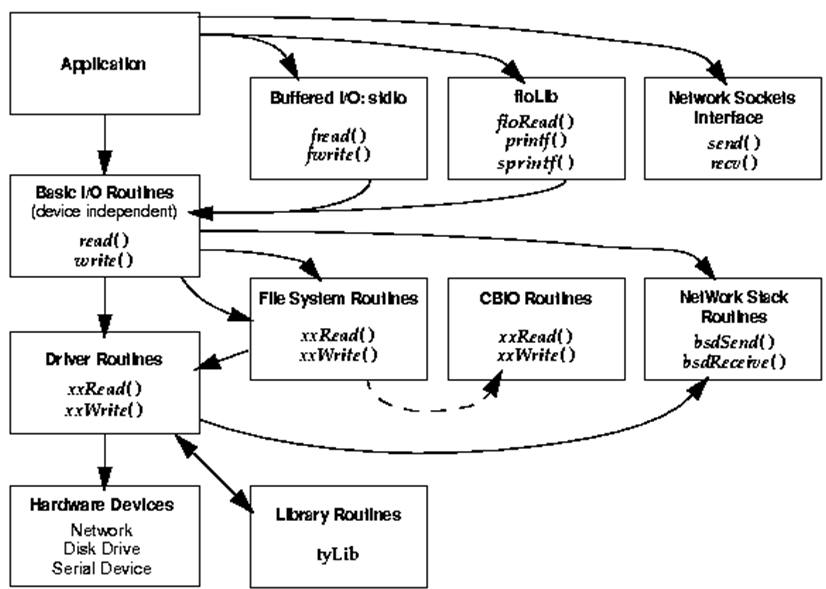


图1 VxWorks I/O系统概述

文件名用一个字符串表示，一个没有结构的文件用一个设备名表示。以文件系统设备为例，设备名字后面紧跟着文件名，比如/tyCo/0用于命名一个指定的串行I/O通道，DEV1：/file1表示一个在设备DEV1:上的一个文件file1。

当在一个I/O调用中指定一个文件名时，I/O系统会根据这个文件名来搜素设备，然后I/O例程定位到这个设备；如果I/O系统找不到文件名指定的设备，I/O例程就会被定向到默认的设备上，我们可以将这个默认的设备指定为系统中的任何一个设备，或者不包含任何设备，如果不包含任务设备，那么当I/O系统找不到文件名指定的设备时，将会返回一个错误。VxWorks提供接口ioDefPathGet()获取当前默认的设备，也可以使用接口ioDefPathSet()设置当前默认的设备。

非块设备在添加到I/O系统上时会被命名，这通常在系统初始化时完成。块设备在它们被指定的文件系统初始化时命名。VxWorks I/O系统在给设备命名的方式上没有限制。除非是在寻找匹配的设备和文件时，否则I/O系统不对设备和文件名做任何解释。

但是采用一个传统的惯例来给设备和文件命名还是非常有用的：大多数的设备名都是以斜杠“/”开始，但是非NFS网络设备和VxWorks DOS设备除外。

按照惯例，基于NFS的网络设备都是以一个斜杠“/”开始的名字来命名挂载点，例如/usr。

非NFS网络设备通常以一个远程的机器名跟一个冒号来命名，例如host:，其余的名字则是远程系统目录中的文件名 。

使用dosFs的文件系统通常使用大写字母和数字组合的形式，并紧跟一个冒号:来命名，例如DEV:

**7.1 VxWorks I/O框架**

VxWorks的I/O系统和Unix或者Linux的I/O系统的不同之处在于，响应用户请求的工作分布在与I/O系统无关的设备和设备驱动本身之间。

在GPOS中，设备驱动提供某些底层的I/O例程用于从字符串定位的设备中读取或者写入字符序列。基于字符设备的高级别通信协议协议例程，通常由I/O系统中独立于设备的部分实现。在驱动例程获得控制权前，用户请求严重依赖于I/O系统的服务。

尽管这种解决方案使得驱动实现起来较为容易，并且驱动的行为表现的尽可能的相似，但是它存在这种缺陷：即当驱动的开发者实现现存的I/O系统中不存在的协议时，将面临极大的困难。而在实时系统中，如果某些设备的吞吐量至关重要时，我们需要绕过标准的协议，或者这类设备本身就不适合标准的模型。

VxWorks系统中，在控制器权转到设备驱动程序之前，用户的请求进行尽可能少的处理。VxWorks I/O系统的角色更像是一个转接开光，负责将用户请求转接到合适的驱动例程上。每一个驱动都能够处理原始的用户请求，到最合适它的设备上。另外，驱动程序开发者也可以利用高级别的库例程来实现基于字符设备或者块设备的标准协议。因此，VxWorks的I/O系统具有两方面的优点：一方面使用尽可能少的使用驱动相关代码就可以为绝大多数设备写成标准的驱动程序，另一方面驱动程序开发者可以在合适的地方使用非标准的程序自主的处理用户请求。

设备一般有两种类型(这里暂时不考虑网络设备)：块设备或者字符设备。块设备用于存放文件系统，在块设备中数据以块的形式存放，块设备采用随机访问的方式，像硬盘或者软盘都属于块设备；不属于块设备范畴的设备被称为字符设备，向串行设备或者图形输入设备都属于字符设备。

VxWorks的I/O系统包含三个元素：驱动、设备和文件。接下来我们以字符设备为例，其中大部分的分析也适用于块设备。当然，块设备必须和vxWorks 文件系统进行交互，所以其在组织结构上和字符设备稍有不同。

**7.2 VxWorks I/O 基本接口**

I/O基本接口是VxWorks系统中最底层的接口，VxWorks的基本I/O接口和标准C库中的I/O接口原语实现了源码级兼容，由VxWorks的ioLib库提供支持，基本接口有7个，如下表1所示。

表1 基本I/O接口



在基本I/O基本，文件是用一个文件描述符fd(file descriptor)来引用，fd是上表中open()或者creat()返回的一个整数。其它的基本I/O接口使用这个fd作为参数来指定操作的文件。

文件描述符fd对系统来说是全局的，例如任务A调用write()操作fd7和任务B调用write()操作fd7，所引用的都是同一个文件。

当打开一个文件时fd将会被分配并返回给用户使用，当文件关闭时，fd也将会被释放。在VxWorks系统中fd的数目是有限的，为了避免超过VxWorks系统的限制，当fd使用完毕后，最好将其关闭。fd的数目在VxWorks系统初始化时设定。

在VxWorks系统中下面三个描述符是系统保留的，具有特殊的意义：

0：标准输入；

1：标准输出；

2：标准错误输出；

VxWorks的基本I/O例程open()和creat()通常不会返回上面三个描述符，标准的描述符使得任务和模块独立于他们实际的I/O分配。如果一个模块发送输出信息到标准输出描述符(fd=1)上，那么输出可以重定向到任何一个文件或者设备上，而不需要修改该模块。

VxWorks允许两级重定向，首先，存在一个全局的分配三个标准文件描述符；其次个别的任务可以重新分配三个标准描述符，将其重定位到只分配给这些任务的设备上。

全局重定向：在VxWorks系统初始化时，标准描述符被重定位到系统终端上。当任务被创建时，它们没有被分配私有的文件描述符fd，只能使用全局标准描述符。VxWorks提供了ioGlobalStdSet()用于重定向全局描述符，例如下面的例子将全局标准输出描述符(fd=1)重定向到文件描述符fileFd指向的一个打开文件：

ioGlobalStdSet(1, fileFd);

如果任务没有私有的文件描述符，都可以使用系统全局标准描述符，例如任务tRlogind调用ioGlobalStdSet()将I/O输出重定向到rlogin会话任务上。

任务级重定向：特定任务的标准描述符可以通过接口ioTaskStdSet()进行重定向，该例程的第一个参数是需要进行重定向的任务ID(ID为0表示自身)，第二个参数是需要重定向的标准描述符，第三个参数是需要重定向到的文件描述符。例如下面的例子，任务A将标准输出重定向到fileFd描述符:

ioTaskStdSet(0,1, fileFd)

**7.3 VxWorks驱动层次结构**

正如我们在前面所述，VxWorks 的I/O框架由ioLib.c 文件提供，但ioLib.c文件提供的函数仅仅是一个最上层的接口，并不能完成具体的用户请求，而是将请求进一步向其他内核模块进行传递，位于ioLib.c模块之下的模块就是iosLib.c。我们将ioLib.c 文件称为上层接口子系统，将iosLib.c文件称为I/O 子系统，注意二者的区别。上层接口子系统直接对用户层可见，而I/O 子系统则一般不可见（当然用户也可以直接调用iosLib.c 中定义的函数，但一般需要做更多的封装，且违背了内核提供的服务层次），其作为上层接口子系统与下层驱动系统的中间层而存在。图2展示了VxWorks系统驱动层次。

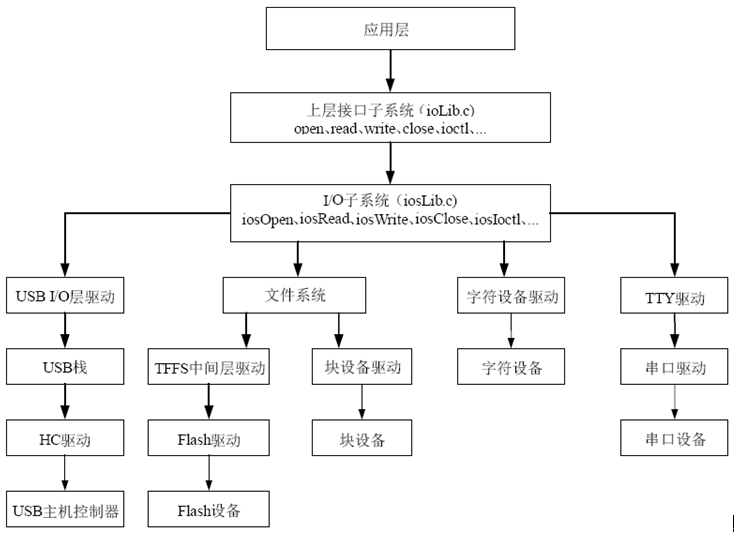


图2 VxWorks内核驱动层次

从图2中可以看出：I/O 子系统在整个驱动层次中起着十分重要的作用，其对下管理着各种类型的设备驱动。换句话说，各种类型（包括网络设备）的设备都必须向I/O 子系统进行注册方可被内核访问。所以在I/O 子系统这一层次，内核维护着三个十分关键的数组用以对设备所属驱动、设备本身以及当前系统文件句柄进行管理。

需要指出的是，VxWorks文件系统在内核驱动层次中实际上是作为块设备驱动层次中的一个中间层而存在的，其向I/O 子系统进行注册，而将底层块设备驱动置于自身的管理之下以提高数据访问的效率。在这些文件系统中，dosFs 和rawFs 是最常用的两种文件系统类型，在VxWorks早期版本就包含对这两种文件系统的支持。

**7.4 VxWorks I/O驱动框架构成**

由于I/O 子系统在整个驱动层次中起着管理的功能，其维护着与系统设备和驱动相关的三张表：设备表，驱动表，文件句柄表。下面我们来介绍这三种表结构，并通过这三种表来一窥VxWorks I/O子系统的 全貌。

我们以一个字符设备为例，假设已经将该字符设备的驱动向I/O 子系统进行了注册，并创建了一个该字符设备的文件节点。下面以用户打开设备操作为例，介绍用户层请求传递到底层驱动的调用流程。

用户在使用一个设备之前必须先打开该设备。以字符设备的文件节点为路径名调用open() 函数，open函数将请求转移给iosOpen()。I/O 子系统维护着当前系统所有的驱动和设备。其根据open() 函数调用时传入的设备节点名从系统设备列表中查询设备，查询到设备后，由设备结构中的相关字段值得知该设备对应的驱动程序索引号，I/O 子系统根据该驱动程序号从系统驱动列表中获取该设备对应的驱动函数集合，调用底层驱动open() 响应函数，底层驱动open() 响应函数将进行中断注册，使能硬件工作，完成用户层打开设备的服务请求。

open() 函数返回一个整型数，我们将其称为文件描述符，I/O 子系统除了系统驱动和系统设备两张表外，其维护的第三张表就是系统当前打开的所有的文件描述符表，该表中每个表项都是一个数据结构，表项在表中的位置索引号就是文件描述符本身，而表项的内容则表明了该文件描述符对应的设备以及驱动。此后对设备的读写、控制、关闭或其他任何操作将以文件描述符为依据。由文件描述符可以直接寻址到被操作设备的驱动程序。

**7.4.1 系统驱动表**

I/O 子系统维护的系统驱动表包含了当前注册到I/O 子系统下的所有驱动。这些驱动可以是直接驱动硬件工作的驱动层，如一般的字符驱动，也可以是驱动中间层，如文件系统中间层、TTY 中间层、USB I/O中间层等。对于中间层驱动，下层硬件驱动将由这些中间层自身负责管理，而不再通过I/O 子系统。如串口底层驱动将通过TTY 中间层进行管理，而不再通过I/O 子系统。

系统驱动表底层实现是一个数组，数组元素数目在VxWorks 内核初始化过程中初始化I/O 子系统时指定。iosInit 函数用以初始化I/O 子系统，iosInit 函数调用原型如下：

STATUS iosInit

(

int max\_drivers,            /\* maximum number of drivers allowed \*/

int max\_files,              /\* max number of files allowed open at once \*/

char \*nullDevName           /\* name of the null device (bit bucket) \*/

)

其中：

参数1（max\_drivers）:指定系统驱动表元素数目，即系统最多支持的驱动数。

参数2（max\_files）:指定系统同时打开的最大文件数，这个参数实际上指定了系统文件描述符表的元素数目。

参数3（nullDevName）:指定了null 设备的设备节点名，一般为“/null”

系统驱动表在内核中由drvTable 表示，其声明如下：

DRV\_ENTRY \* drvTable; /\* driver entry point table \*/

在iosInit 函数中根据传入的最大驱动数目对drvTable 进行初始化，如以下代码所示：

STATUS iosInit

    (

    int max\_drivers,            /\* maximum number of drivers allowed \*/

    int max\_files,              /\* max number of files allowed open at once \*/

    char \*nullDevName           /\* name of the null device (bit bucket) \*/

    )

    {

    int i;

    int size;

    maxDrivers      = max\_drivers;

    maxFiles  = max\_files;

.......略...........

    /\* allocate driver table and make all entries null \*/

    size = maxDrivers \* sizeof (DRV\_ENTRY);

    drvTable = (DRV\_ENTRY \*) malloc ((unsigned) size);

    if (drvTable == NULL)

         return (ERROR);

    bzero ((char \*) drvTable, size);

    for (i = 0; i < maxDrivers; i++)

         drvTable [i].de\_inuse = FALSE;

   ......略..........

    return (OK);

    }

系统驱动表中每个表项都是一个DRV\_ENTRY 类型的结构，该结构定义在h/private/ iosLibP.h 文件中，如下：

typedef struct           /\* DRV\_ENTRY - entries in driver jump table \*/

{

FUNCPTR de\_create;

FUNCPTR de\_delete;

FUNCPTR de\_open;

FUNCPTR de\_close;

FUNCPTR de\_read;

FUNCPTR de\_write;

FUNCPTR de\_ioctl;

BOOL        de\_inuse;

} DRV\_ENTRY;

可以看出，DRV\_ENTRY 实际上就是一个函数指针结构，结构中的每个成员都指向一个完成特定功能的函数，这些函数与用户层提供标准函数的接口一 一对应。成员de\_inuse 用以表示一个表项是否空闲。

iosInit() 函数创建系统驱动表，从以上代码示例来看，该表实际上是由drvTable 指向的一个数组，数组大小由传入iosInit 函数的第一个参数决定，每个表项在drvTable 中的位置索引就作为驱动号。索引号为0的表项被WInd内核预留，专门用做null 设备的驱动号，故驱动号的分配实际上是从1 开始的。

I/O 子系统提供iosDrvInstall() 供驱动程序注册用，iosDrvInstall() 函数调用原型如下。

int iosDrvInstall

(

FUNCPTR pCreate,    /\* pointer to driver create function \*/

FUNCPTR pDelete,    /\* pointer to driver delete function \*/

FUNCPTR pOpen,      /\* pointer to driver open function \*/

FUNCPTR pClose,     /\* pointer to driver close function \*/

FUNCPTR pRead,      /\* pointer to driver read function \*/

FUNCPTR pWrite,     /\* pointer to driver write function \*/

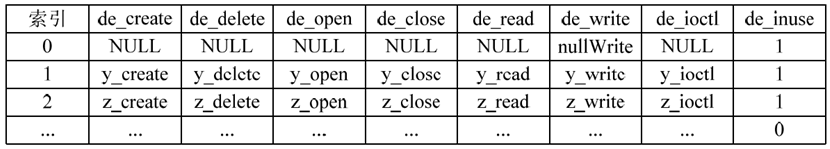
FUNCPTR pIoctl      /\* pointer to driver ioctl function \*/

)

一个设备驱动在初始化过程中一方面完成硬件设备寄存器的配置，另一方面就是向I/O 子系统注册驱动和设备，从而使设备对用户可见。可以看到，iosDrvInstall()函数参数为一系列的函数地址，这些函数对应了为用户层提供的标准接口函数。一个驱动无须提供以上所有函数的实现，对于无须实现的函数，直接传递NULL 指针即可。iosDrvInstall() 函数的基本实现即遍历drvTable 数组，查询一个空闲表项，用传入的函数地址对表项中各成员变量进行初始化，并将de\_inuse 设置为TRUE，最后返回该表项在数组中的索引作为驱动号。设备初始化函数将使用该驱动号调用iosDevAdd() 将设备添加到I/O 子系统中。此后用户就可以使用iosDevAdd 函数调用时设置的设备节点名对设备进行打开操作，打开后进行读写或控制等其他操作，完成用户要求的特定功能。

用户可在命令行下输入iosDrvShow，显示系统驱动表中当前存储的所有驱动。如表2为系统驱动表的一个简单示意。

表2 系统驱动表



一个非块设备的驱动实现基本的I/O函数：creat()、delete()、open()、close()、read()、write()、以及ioctl()。通常情况下，该驱动包含例程来实现这7个基本I/O函数，但是如果这些例程中的某些相对于该驱动对应的设备没有意义的话，相应的实现为NULL。

驱动可以选择允许多个任务在多个文件描述符上等待被激活，这在驱动的ioctl()例程中实现，我们会在分析select()例程时分析。

块设备的驱动程序和文件系统交互，而不是直接跟VxWorks的I/O系统交互。文件系统反过来执行绝大多数的I/O功能，而驱动则仅提供例程来读写块、重置驱动、执行I/O控制、检查设备状态。

当用户调用基本I/O功能时，I/O系统将这些请求路由到指定驱动的适合例程，驱动例程在被调用任务的上下文中执行，看起来好像是应用程序直接调用这些例程。然而，驱动可以自由使用正常情况下对任务可用的功能，包括I/O或者其他的设备，这意味着绝大多数驱动必须包含某些机制以提供对代码临界区的互斥访问，通知的机制是使用semLib库中的信号量机制。

除了实现7个基本I/O功能的例程之外，VxWorks的I/O系统的驱动还包含下面三个例程：

1.  一个初始化例程，典型的命名为xxDrv()，该例程在I/O系统中安装驱动、连接中断到驱动服务的设备、执行一些必要的硬件初始化操作；

2.  一个添加驱动服务的设备到VxWorks的I/O系统中的例程，该例程通常被命名为xxDevCreate();

3.  中断级例程，用于连接驱动服务的设备产生的中断中。

驱动表和安装驱动：I/O系统的功能是将用户的请求路由到合适驱动的合适例程，并由该例程完成用户的请求。I/O系统通过维护一张包含每个驱动的各个例程的入口地址的表来实现这样的功能。通过调用I/O系统内部例程iosDrvIntall()来动态的安装驱动，iosDrvIntall()的参数是将要添加的新驱动的七个基本I/O例程的入口地址。iosDrvIntall()将这七个函数的入口地址填入到驱动表的一个空闲槽中，并返回这个槽的索引。这个索引被称为驱动号(driver number)，将会被该驱动服务的设备所引用。

如果驱动的7个基本I/O功能的入口例程地址被置为NULL，这意味着这个驱动不需要处理这些功能。比如对于非文件系统的驱动，close()和delete()不需要做任何事情。

VxWorks的文件系统(比如dosFsLib)在驱动表中包含他们自己的例程，当文件系统库被初始化时，这些例程被创建。非块设备的驱动初始化过程如图3所示。

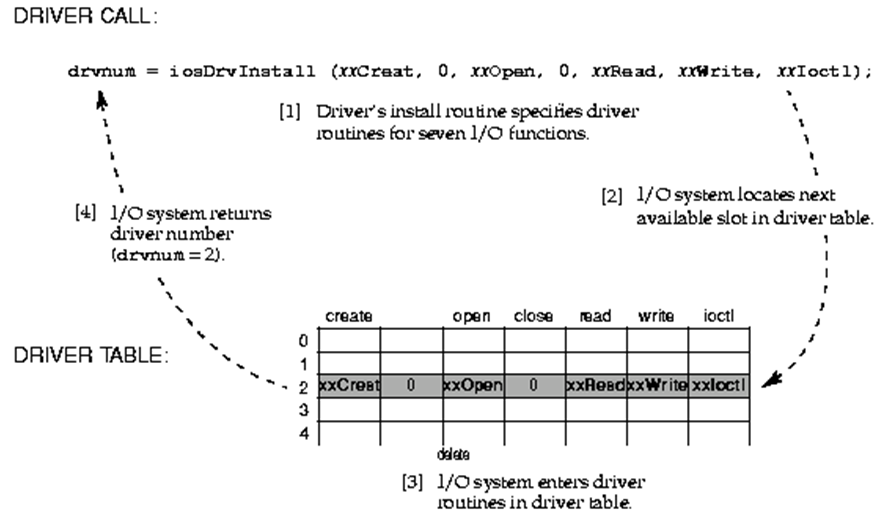


图3 非块设备的初始化过程

图3中显示的是当示例驱动的初始化例程xxDrv()运行时，示例驱动和I/O系统所采取的行动。

驱动调用iosDrvInstall()指定七个基本I/O函数的地址，然后I/O系统执行下面的操作：

1.  定位驱动表中下一个可用的空闲槽，在本例子中，为空闲槽slot2；

2.  在驱动表的空闲槽slot2中填写7个基本I/O函数的地址；

3.  返回新安装的驱动所在的空闲槽slot2的索引2，该索引被称为驱动号(driver number)。

**7.4.2 系统设备表**

某些驱动可以为指定设备的多个实例提供服务。例如，串口通信设备的单个驱动可以处理多个分割的仅参数不同的通道，比如设备地址。

在VxWorks的I/O系统中，设备用一个称之为设备头(DEV\_HDR)的数据结构来定义，这个数据结构包含设备名字符串、服务这个设备的驱动设备号。在VxWorks的系统中，所有设备的设备头都常驻内存，并链成一个设备链表。设备头是特定驱动所确定的一个大的数据结构的起始部分，这个大的数据结构称之为设备描述符(device descriptor)，设备描述符除了设备头之外，还包含诸如设备地址、缓冲区、信号量等与具体设备相关的成员。

VxWorks 内核对每个设备使用DEV\_HDR 数据结构进行表示，该结构定义如下:

typedef struct                   /\* DEV\_HDR - device header for all device structures \*/

    {

    DL\_NODE         node;                 /\* device linked list node \*/

    short         drvNum;            /\* driver number for this device \*/

    char \*       name;                /\* device name \*/

    } DEV\_HDR;

该结构中给出了链接指针（用以将该结构串入队列中）、驱动索引号、设备节点名。内核提供这个结构较为简单，只存储了一些设备关键系统。底层驱动对其驱动的设备都有一个自定义数据结构表示，其中包含了被驱动设备寄存器基地址、中断号、可能的数据缓冲区、保存内核回调函数的指针，以及一些标志位。最关键的一点是DEV\_HDR 内核结构必须是这个自定义数据结构的第一个成员变量，因为这个用户自定义结构最后需要添加到系统设备队列中，故必须能够在用户自定义结构与DEV\_HDR 结构之间进行转换，而将DEV\_HDR 结构设置为用户自定义结构的第一个成员变量就可以达到这个目的。如下代码为一个用户自定义设备结构的简单示例。

typedef struct xxDev

{

DEV\_HDR devHdr; //内核提供的结构，必须是自定义结构的第一个成员变量。

UINT32 regBase;   //设备寄存器基地址。

UINT32 buffPtr;     //数据缓冲区基地址。

BOOL isOpen;        //设备已打开标志位。

UINT8 intLvl;         //设备中断号。

FUNCPTR putData; //内核回调函数指针，该指针指向的函数向内核提供数据。

FUNCPTR getData; //内核回调函数指针，该指针指向的函数从内核获取数据。

… //其他设备参数。

}

为了能够让用户对设备进行操作，驱动程序必须将设备注册到I/O 子系统中，这个过程也被称为创建设备节点。

I/O 子系统提供的iosDevAdd()函数用以被驱动程序调用注册一个设备。该函数调用原型如下:

STATUS iosDevAdd

    (

    DEV\_HDR \*pDevHdr, /\* pointer to device's structure \*/

    char \*name,       /\* name of device \*/

    int drvnum        /\* no. of servicing driver, \*/

                         /\* returned by iosDrvInstall() \*/

    )

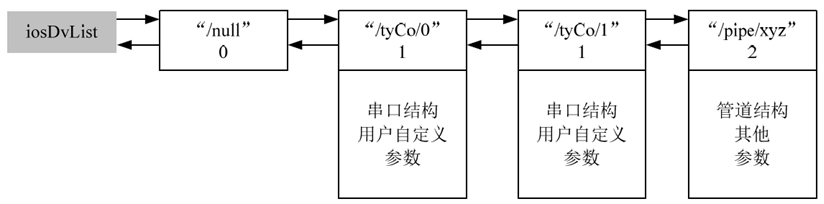
传入iosDevAdd 的参数：

参数1(pDevHdr):是一个DEV\_HDR 结构类型，一般我们将用户自定义结构作为第一个参数传入，这也是必须将DEV\_HDR 结构类型的成员变量作为用户自定义结构的第一个成员的原因所在。

参数2(name):表示设备节点名，这个名称将被用户程序调用作为打开设备时的路径使用。

参数3(drvnum):是设备对应的驱动程序索引号。这个驱动号是iosDrvInstall ()函数的返回值。在设备初始化函数中，我们首先调用iosDrvInstall() 注册驱动，然后使用iosDrvInstall() 函数返回的驱动号调用iosDevAdd 添加设备到系统中，这两步完成之后，设备就可以被用户程序使用了。

osDevAdd() 函数将一个设备添加到由I/O 子系统维护的系统设备列表中，该列表是一个队列，队列中的成员通过指针链接在一起，这是由DEV\_HDR 结构中的node 成员变量完成的。系统设备列表由iosDvList内核变量指向，如图4所示为系统设备列表示意图。

图4 系统设备表示意图

系统设备列表中第一个设备是内核本身添加的，这是一个null 设备，所有写入null 设备的数据都将被直接丢弃，这种机制对于屏蔽一些输出十分有效。null 设备是内核内部设备，驱动号0 被专门预留给null设备。null 设备单独有一个DEV\_HDR 结构表示，不存在其他参数，故图3 中对null 设备只显示了一个DEV\_HDR 结构，而其他设备一般都需要在DEV\_HDR 结构之外定义额外的参数。

图4中还显示了系统中存在的两个串口设备，这两个串口使用相同的驱动，实际上，此处显示的驱动索引号是TTY 驱动的驱动号，还不是真正的底层串口驱动号，底层串口驱动通过TTY 进行管理，故对不同串口的操作在TTY 驱动层才进行分离，所有的串口驱动首先都需要通过相同的TTY 驱动层的处理，而后请求被转发到具体的底层串口驱动。用户可在命令行下输入iosDevShow 或devs，显示系统设备中的所有设备。

用户调用open() 函数打开一个设备文件时，I/O 子系统将以传入的文件路径名匹配系统设备中的设备节点名，匹配方式是最佳匹配，即名称最相近的设备被返回。如输入的文件路径为“/pipe/xyz”，如果系统设备表中存在两个设备：“/pipe/xy”、“/pipe/xyz”，那么“/pipe/xyz”设备将被返回，无论其位置在前还是在后。当然如果传入的文件路径名长度较小，那么此时系统设备表最前面的设备将被返回。例如，如果传入的文件路径名为“/pipe/x”，那么对于系统设备中的“/pipe/xy”和“/pipe/xyz”两个设备，谁位于设备表的前面，谁就被返回。对于路径名比设备名长的情况，在对块设备的操作中比较普遍。一般我们在块设备上创建一个文件系统，我们对块设备创建一个设备节点，而对块设备的所有操作都是在这个根节点下，此时块设备节点就成为判断一个被操作的文件或者目录到底属于哪个块设备（如果系统中存在多个块设备的话）。

正如前面所述，非块设备通过调用内部例程iosDevAdd()被动态添加到I/O系统中，iosDevAdd()的参数包含这个新设备的设备描述符地址、设备名字、以及服务该设备的驱动的驱动号。设备描述符是由驱动所指定的，其中包含必须的与设备相关的信息，但是必须以设备头作为该设备描述符的第一个成员。驱动不需要添加设备描述符的设备头部分，只需要填写设备描述符中与具体设备相关的信息。iosDevAdd()例程负责填写设备头中的设备名、设备号，并把该设备添加到系统设备链表中。

添加一个块设备到I/O系统中，需要调用为块设备的文件系统调用设备初始化例程(比如dosFsDevCreate()或者rawFsDevInit())，设备初始化例程自动调用iosDevAdd()例程。

例程iosDevFind()用于定位设备结构(通过指向DEV\_HDR结构的指针，它是设备描述符结构的第一个成员)，并且验证设备名在设备表中是否存在。下面是使用iosDevFind()的一个例子：

char \*  pTail;                                               /\* 指向设备名devName的尾部 \*/

char devName[6] = "DEV1:";                   /\* 设备名 \*/

DOS\_VOLUME\_DESC \*  pDosVolDesc;   /\* 第一个设备时设备头 DEV\_HDR \*/

    ...

    pDosVolDesc = iosDevFind(devName, (char\*\*)&pTail);

    if (NULL == pDosVolDesc)

        {

        /\* ERROR: 设备名不存在,且没有默认的设备 \*/

        }

    else

        {

        /\* pDosVolDesc是一个合法的设备头DEV\_HDR指针

         \* pTail指向设备名devName的开始部分

         \* 通过pTail检查设备名,以确定该设备名时一个默认的设备名，

\* 还是一个指定的设备名

         \*/

        }

图5展示的是一个驱动的设备创建例程xxDevCreate()通过调用iosDevAdd()添加一个设备到I/O系统中。

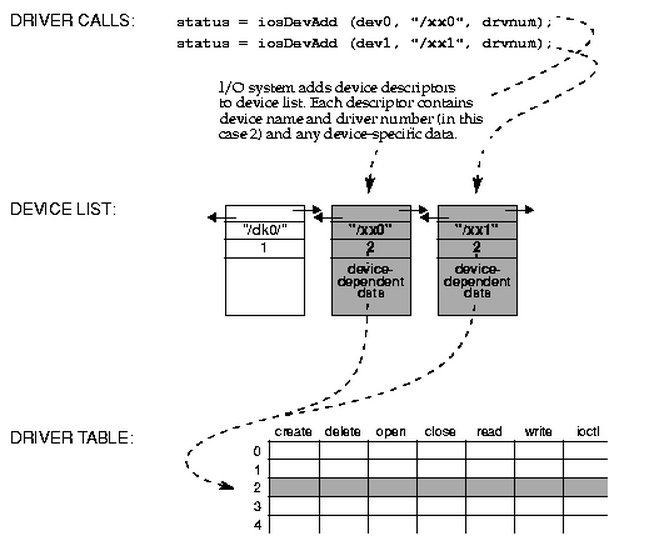


图5添加设备到I/O系统中示意图

**7.4.3 文件描述符**

I/O 子系统维护的第三张表就是系统文件描述符表，即当前系统范围内打开的所有文件描述符都将存储在该表中。文件描述符表底层实现上也是一个数组，正如设备驱动表表项索引用做驱动号，文件描述符表表项索引被用做文件描述符ID，即open 函数返回值。对于文件描述符，有一点需要注意：标准输入、标准输出、标准错误输出虽然使用0、1、2 三个文件描述符，但是可能在系统文件描述符表中只占用一个表项，即都使用同一个表项。VxWorks 内核将0、1、2 三个标准文件描述符与系统文件描述符表中的内容分开进行管理。实际上，系统文件描述符中的内容更多的是针对硬件设备，即使用一次open 函数调用就占用一个表项。0、1、2 三个标准文件描述符虽然占用ID 空间（即其他描述符此时只能从3 开始分配），但是其只使用了一次open 函数调用，此后使用ioGlobalStdSet 函数对open 返回值进行了复制。

在同一个时间内，多个文件描述符可以打开同一个单独的设备，一个设备驱动能够维护和文件描述符fd相关的信息，而不仅仅是I/O系统中设备信息。特别的是，当多个文件对应同一个设备被打开时，必须还有与各个文件描述符fd相关的文件信息，比如文件偏移量等等。在一个非块设备上，我们可以有多个文件描述符fd，比如tty设备。典型的，由于没有其它特别的信息，因此向任务一个文件描述符写数据，都会产生相同的结果。

文件可以通过open()或者creat()打开文件描述符表(Fd Table)，I/O系统搜索设备链表中的设备名，来寻找和用户指定的文件名向匹配的设备名。如果发现匹配的设备名，I/O系统将会使用该设备头中包含的设备号来定位并且调用设备表中的驱动打开例程。

I/O系统必须建立一个在随后的I/O调用中用户使用的文件描述符和执行该服务的驱动之间的连接。另外，驱动必须给每个文件描述符一个关联的数据结构。在非块设备的例子中，这通常是位于 系统中的设备描述符。

I/O系统在一个文件描述符表(fd table)中维护这种关联，这个表中包含驱动号和由该驱动决定的4字节大小的一个数值，我们称之为驱动值。驱动值是一个由驱动打开例程所返回的内部描述符，是一个驱动用于指定文件的一个菲负整数值。在随后的驱动其它I/O函数的调用(read()、write()、ioctl()、以及close())，这个值被驱动在随后的应用基本的I/O调用中替换文件描述符fd。

比如在VxWorks系统中，通过下面的代码对三个标准文件描述符进行初始化，为了方便阅读的方便，代码我进一步做了简化：

int    consoleFd = NONE;      /\* fd of initial console device      \*/

STATUS usrSerialInit (void)

    {

    char tyName [20];

    int ix;

    if (NUM\_TTY > 0)

         {

         ttyDrv();                               /\* install console driver \*/

         for (ix = 0; ix < NUM\_TTY; ix++)        /\* create serial devices \*/

             {

             tyName[0] = '\0';

             strcat (tyName, "/tyCo/");

             strcat (tyName, itos (ix));

             (void) ttyDevCreate (tyName, sysSerialChanGet(ix), 512, 512);

#if  (!(defined(INCLUDE\_PC\_CONSOLE)))

             if (ix == CONSOLE\_TTY)             /\* init the tty console \*/

                   {

                   consoleFd = open (tyName, O\_RDWR, 0);

                   (void) ioctl (consoleFd, FIOBAUDRATE, CONSOLE\_BAUD\_RATE);

                   (void) ioctl (consoleFd, FIOSETOPTIONS, OPT\_TERMINAL);

                   }

#endif /\* INCLUDE\_PC\_CONSOLE \*/

             }

         }

    ioGlobalStdSet (STD\_IN,  consoleFd);

    ioGlobalStdSet (STD\_OUT, consoleFd);

    ioGlobalStdSet (STD\_ERR, consoleFd);

    return (OK);

    }

可以看到，usrSerialInit ()函数中将“/tyCo/0”作为了三个标准输入/输出，此处只使用了一次open() 函数调用，

语句如下：

consoleFd = open (tyName, O\_RDWR, 0);

实际上，consoleFd=3，因为标准输入/输出占用了0、1、2 三个文件描述符，所以系统文件描述符表中存储的描述符最小值就是3。此后使用ioGlobalStdSet 函数将这个描述符“3”指向的设备作为0、1、2三个描述符的默认设备。即此处将串口作为了标准输入/输出设备。内核将0、1、2 三个文件描述符预留给了标准输入/输出，并将其与系统文件描述符表中的表项隔离开来，内核专门使用ioStdFd 数组表示0、1、2 三个文件描述符指向的具体系统文件描述符表中哪个表项。

LOCAL int ioStdFd [3];                /\* global standard input/output/error \*/

所以ioGlobalStdSet（0，consoleFd）; 语句实际上完成如下工作：

ioStdFd[0] = consoleFd；

其他两条语句的实际结果为：

ioStdFd[1] = consoleFd；

ioStdFd[2] = consoleFd；

而consoleFd 等于3，实际上是系统文件描述符表中的第一个表项，其索引为0，但是在作为文件描述符返回时，基于0、1、2 已被预留为标准输入/输出，故做加3 处理，实际上，系统文件描述符表项索引作为文件描述符返回时都做加3 处理。

当使用一个文件描述符进行操作时，如调用write 函数，内核首先检查文件描述符是否是0、1、2 标准输入/输出描述符，如是，则依次为索引查询ioStdFd，以ioStdFd[fd]作为索引查询系统文件描述符表，获得驱动号，进而索引系统驱动表，调用对应表项de\_write 指向的函数，完成对设备的写入操作；如果文件描述符大于2，表示这是一个普通的文件描述符，那么就直接以该描述符作为索引查询系统文件描述表，获得驱动号，进而索引系统驱动表，调用相关函数。

系统文件描述符表中每个表项都是一个FD\_ENTRY 类型的结构，该结构定义在h/private/ iosLibP.h 中，代码如下。

typedef struct           /\* FD\_ENTRY - entries in file table \*/

    {

    DEV\_HDR \*      pDevHdr; /\* device header for this file \*/

    int   value;                 /\* driver's id for this file \*/

    char \*       name;                /\* actual file name \*/

    int              taskId;               /\* task to receive SIGIO when enabled \*/

    BOOL       inuse;                 /\* active entry \*/

    BOOL        obsolete; /\* underlying driver has been deleted \*/

    void \*        auxValue;          /\* driver specific ptr, e.g. socket type \*/

    void \*        reserved; /\* reserved for driver use \*/

} FD\_ENTRY;

备注：FD\_ENTRY 结构的第一个成员就是DEV\_HDR 结构类型，该结构中存储了设备节点名和驱动号。FD\_ENTRY 结构中的value 成员表示驱动附加信息，并非驱动号，实际上这个字段被用以保存底层驱动中open 实现函数的返回值，这个返回值的意义重大，因为其后驱动中read、write 等实现函数被调用时，I/O 子系统就以这个返回值作为这些函数的第一个参数。所以，底层驱动open 实现函数一般返回一个驱动自定义结构句柄。name 成员变量按理应该设置为文件名，但是DEV\_HDR 结构中已经有设备节点名（也就是文件名），故该成员变量当前被设置为NULL，节省了内存空间。

系统文件描述符表由内核变量fdTable 指向，该变量声明如下。

FD\_ENTRY \* fdTable; /\* table of fd entries \*/

fdTable 的初始化在iosInit() 函数中完成，该函数调用原型如前文所示，传入该函数的第二个参数指定了fdTable 数组的大小，该变量的初始化代码示例如下。

size = maxFiles \* sizeof (FD\_ENTRY);

fdTable = (FD\_ENTRY \*) malloc ((unsigned) size);

用户程序每调用一次open ()函数，系统文件描述符表中就增加一个有效表项，直到数组满，此时open函数调用将以失败返回。表项在表中的索引偏移3 后作为文件描述符返回给用户，作为接下来其他所有操作的文件句柄。

用户可在命令行下输入iosFdShow，显示系统文件描述附表中当前所有的有效表项。如图6 所示为系统文件描述符表的一个简单示意图。

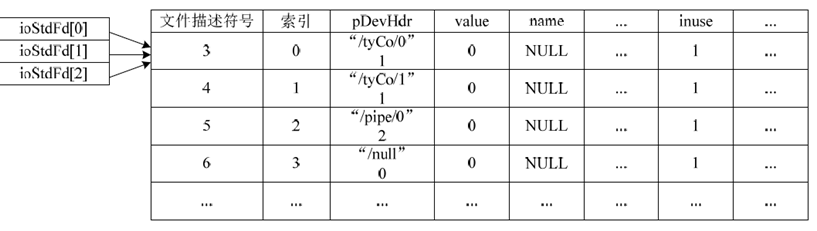


图6 系统文件描述符表示意图

备注：每个文件描述符的确定由对应的设备文件被打开的时机决定，一般而言，串口被用做标准输入/输出是最早被打开的设备，其他设备只在需要时由用户程序打开。当然这也不可一概而论。

图6 中也显示了0、1、2 三个标准文件描述符对系统文件描述符表的索引。此时“/tyCo/0”设备同时也被用于标准输入/输出。ioStdFd 数组有且仅有三个元素，0 号元素表示标准输入，1 号元素表示标准输出，2 号元素表示标准错误输出。ioStdFd 数组索引本身表示文件描述符，而元素内容表示实际操作设备时使用的文件描述符，usrRoot 函数将ioStdFd 三个元素均设置为3，即第一个串口设备对应的文件描述符“3”，也就是使用第一个串口设备作为标准输入/输出设备。

**7.4.4 三张表之间的联系**

下面我们以一个open() 调用为例，介绍I/O 子系统维护的三张表是如何相互协作完成用户请求的。如图7、图8 所示，用户调用open 函数打开“/xx0”文件，VxWorks 内核I/O 子系统将进行如下一系列响应。

1. VxWorks I/O子系统使用文件路径名匹配系统设备表，查询一个匹配设备。此处在设备列表中找到了一个匹配的设备。
2. VxWorks I/O子系统在系统文件描述符表中预留一个空闲项，用以创建一个文件描述符，如果后续调用成功，将以这个空闲项对应的索引（偏移3）值返回给用户，作为文件描述符使用。
3. VxWorks I/O子系统根据设备列表匹配项中的信息得到驱动号，进而以此驱动号为索引从系统驱动表中获取底层设备驱动对应用户层open() 调用的响应函数x\_open()。x\_open() 的第一个参数被设置为对应的硬件设备结构，第二个参数为除去设备名本身余下的部分，此处为NULL，第三、四个参数为用户传入的权限和模式参数。x\_open() 将完成硬件设备的配置、使能工作、注册中断等，为用户接下来可能的读写设备操作做好准备。x\_open() 同时对传入的第一个参数（设备结构）进行初始化，这个结构将在后续操作中一直被底层驱动使用。

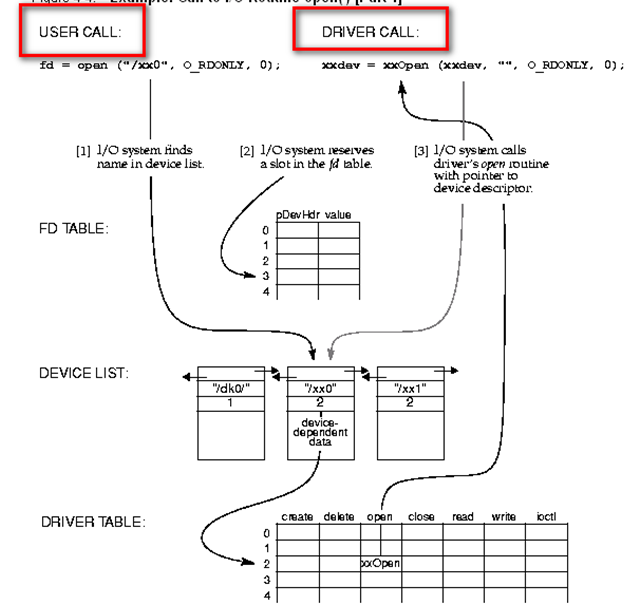


图7 用户请求I/O 服务过程(Part1)

4. 底层驱动返回设备结构，表示底层打开设备成功，否则返回NULL 或ERROR 表示调用失败

5.  I/O 子系统对文件描述符表中预留的空闲项进行初始化，填入驱动号和设备结构。

6.  最后，open 函数调用返回一个文件描述符，这个描述符是文件描述符之前被预留空闲项（现在已得到初始化，被使用）在表中的索引值（偏移3）。此处即文件描述符表中的第一个表项，即fd=0+3=3。

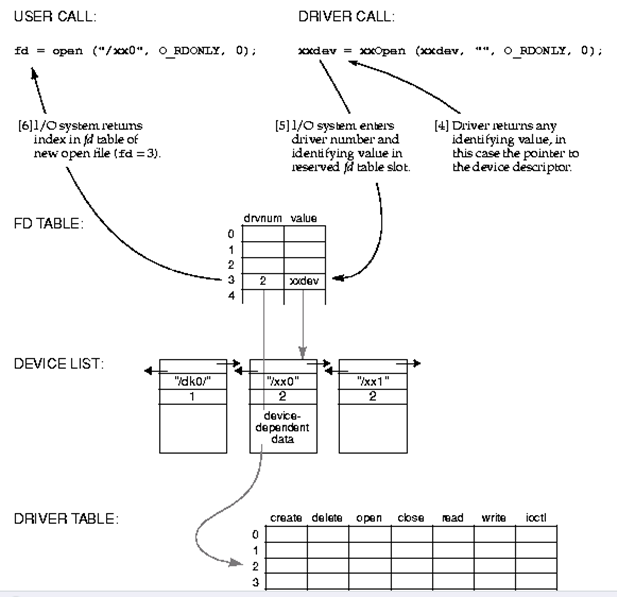


图8 用户请求I/O 服务过程(Part2)

**小结：**

本章我们详细介绍了VxWorks 下的内核驱动层次，着重对I/O 子系统进行了介绍。VxWorks 内核使用三张表对系统内所有的驱动、设备以及打开文件进行管理。这三张表是VxWorks 内核驱动层次的核心。VxWorksI/O 子系统在驱动层次中的作用十分重要，系统内所有的驱动直接或者间接地受I/O 子系统的管理，所有的用户请求都必须通过I/O 子系统进行传递。对于较为复杂的设备（如磁盘设备、Flash 设备、USB 设备），为了简化底层硬件驱动的设备复杂度，VxWorks 内核专门提供了驱动中间层与这些底层驱动接口，此时驱动中间层直接受I/O 子系统的管理，而底层驱动则委托给这些中间层进行管理。

至此，对于VxWorks的I/O驱动框架，将介绍到这里了。关于VxWorks的I/O驱动框架，VxWorks的VxWorks Programmer's Guide有着详细的描述，本章的多个解读和示例代码均取自VxWorks Programmer's Guide。

还是那句话，大家有任何的疑问，欢迎在本站给我留言O(∩\_∩)O~