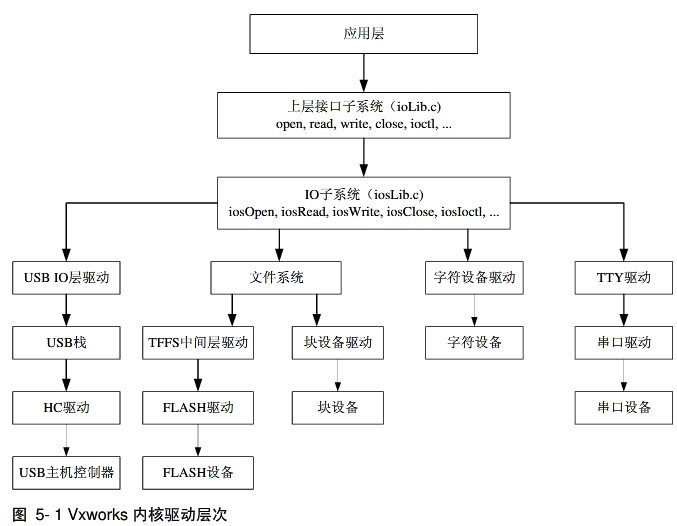
驱动的工作模式

设备驱动从总体上分为两种工作模式：轮询模式和中断模式。关于这两种模式的区别和各自的用处。详见《vxworks设备驱动开发详解--曹桂平版--4.3.4节》。

外设驱动代码编写中有一个问题要特别的注意，不但是外设驱动，所有对外设的寄存器进行操作的代码都必须要注意的一个问题是：对外设的寄存器的操作必须要使用volatile修饰符，虽然在Vxworks提供的sysPhysMemDesc数组初始化的时候将外设的寄存器区间设置为non-cachable，但是还是要使用volatile修饰符，因为MMU机制并非在任何的条件下都会生效。

1. vxworks下设备驱动结构

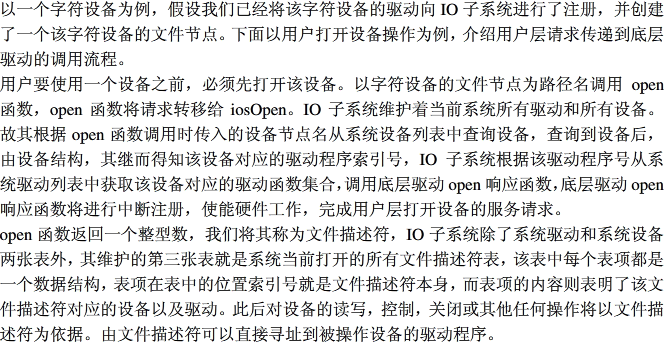
Vxworks对应用层提供了一套标准文件操作接口函数，将其称为标准IO库，vxworks下由ioLib.c文件提供。ioLib.c文件提供如下的标准接口函数：create、open、unlink、remmove、close、rename、read、write、ioctl、lseek、readv、writev等。Vxworks操作系统区别于通用操作系统的一个很大的不同点是vxworks下不区分用户态和内核态，用户层程序可以直接对内核函数进行调用而无需使用指令中断之类的机制或者存在使用权限上的限制。所以vxworks提供给应用层的接口无需通过外围库的方式，而是直接以内核文件的形式提供。这是有操作体统的特点所决定的。



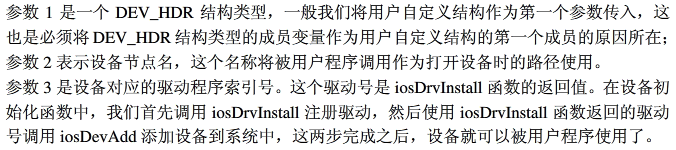
字符设备和块设备均由IO子系统进行管理。而网络设备由于其特殊的工作方式将由另一套的用户标准接口函数（套接字函数）和内核网络栈进行支持。

串口设备本身也是一个字符设备，但是由于串口使用范围广，为了提供串口驱动的编程效率以及串口数据的收发效率，vxworks内核将串口设备区别于一般的字符设备对待，其提供一个TTY中间层驱动对串口数据进行管理（提供缓存）。TTY驱动向IO子系统注册，底层串口驱动对TTY注册。从底层驱动编程难易程度来看，TTY中间层的加入并非显著的降低了底层串口的编程难度，但是TTY中间层提供一个最为关键的优点是极大的提高了数据收发的效率，而且也提供了很大的灵活性，如提供较多的选项用以对串口设备进行控制。

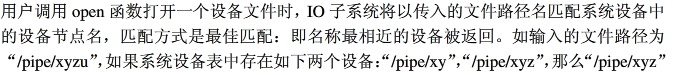
IO子系统在整个的驱动层次当中起着管理的功能，其维护着系统的设备和驱动的关键的三张表。



Vxworks内核对每一个设备使用DEV\_HDR数据结构进行表示，该结构中给出了链接指针、驱动索引号、设备节点名。内核提供的这个结构比较简单，只存储了一些设备的关键系统。底层驱动对其驱动的设备都有一个自定义的数据结构来表示。其中应该包含被驱动的设备的寄存器的基地址、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。且DEV\_HDR内核结构必须要是自定义的结构的第一个成员变量。为了让用户能够对设备进行操作，驱动程序必须将设备注册到IO子系统当中，这个过程也被称为是创建设备节点。IO子系统提供的iosDevAdd函数可以被驱动程序调用来注册一个设备。其中：

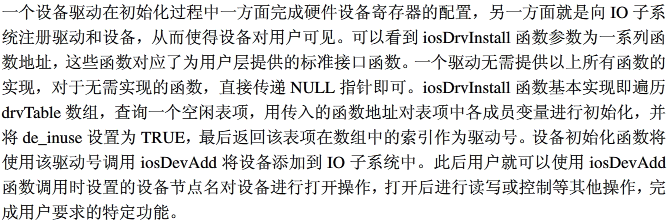


底层的串口驱动是通过TTY来进行管理的，对不同的串口的操作在TTY驱动层才进行分离，所有的串口驱动都要先通过相同的TTY驱动的层的处理，而后请求才被转发（转发的具体过程是怎么样的？？？）到具体的底层的串口驱动。可在命令行中使用iosDevShow或devs显示设备中的所有设备。



5.2.2 系统驱动表

IO子系统维护的系统驱动表包含了当前注册到IO子系统下的所有的驱动，这些驱动可以是直接驱动硬件工作的驱动层，也可以是驱动中间层，如文件系统中间层，TTY中间层，USB IO中间层等。对于中间层驱动，下层的硬件驱动将由这些中间层自身负责管理，而不再通过IO子系统。



5.2.3 系统文件描述符

IO子系统维护的第三张表就是文件系统的描述符，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。文件描述符表底层的实现上也是一个数组，正如设备驱动表表项索引用作驱动号，文件描述符表表项索引被用作文件描述符ID，即open函数的返回值。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，vxworks内核将0，1，2三个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函数就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），但是他只是用了一次open函数调用，此后使用ioGlobalStdSet函数对open函数的返回值进行了复制。如以下usrConfig.c文件中对于三个标准文件描述符的初始化代码。



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，如下语句：consoleFd = open（consoleName，2，0）；

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，并将其与文件系统描述符表中的表项隔离开来，内核专门用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一个表项。

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

所以ioGlobalStdSet（0，consoleFd）；语句实际上完成的工作如下：

ioStdFd[0] = consoleFd;

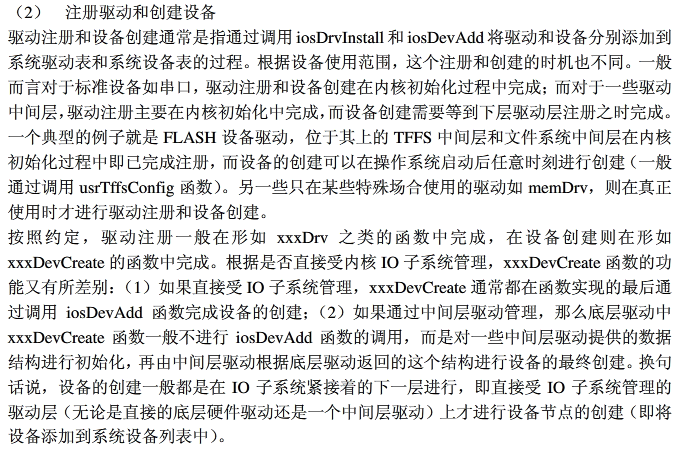
其他两条语句的实际结果即：

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，其索引号为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，故作加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。

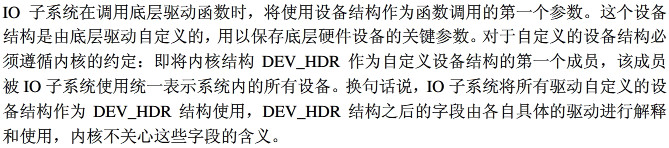
当使用一个文件描述符进行操作时，如调用write函数，内核首先检查文件描述符是否为0，1，2标准输入输出描述符，如果是，则依次为索引查询ioStdFd，以ioStdFd[fd]作为索引查询系统文件描述符表，获得驱动号，进而索引系统驱动表，调用对应表项de\_write指向的函数，完成写入操作。如果文件描述符大于2，表示这是一个普通的文件描述符，那么就直接以该描述符作为索引查询系统文件描述表，获得驱动号，进而索引系统驱动表，调用相关的函数。



1. 字符设备驱动

字符设备以字节流的方式对数据进行操作，数据只能顺序的读写，I2C、SPI、UART等等接口类型的设备都可以作为字符设备进行驱动。字符设备位于Vxworks内核IO子系统直接管理之下，不经过任何中间层，但是内核为了简化某些常用的字符设备的驱动，也会提供一个中间层作为缓冲，如TTY中间层。用户可以自行选择是否使用这些中间层。一般而言，对于UART设备，建议读者使用内核提供的TTY中间层，以提高设备的使用效率，因为UART串口设备在内核启动过程中默认的被设置为标准输入输出通道，在整个的系统运行过程中使用比较频繁。

底层驱动xxxDevCreate进行设备创建时，在每一个设备结构中都存储了该设备的驱动号（xxxDrv函数调用时产生），IO子系统可根据设备列表中设备结构直接查询到该设备对应的驱动程序，当IO子系统以文件路径名在系统设备列表中匹配到一个设备时，其直接根据存储在设备结构中的驱动号在系统的驱动表当中获得对应的设备驱动，并调用open底层驱动的响应函数x\_open，完成用户的打开设备请求。



6.2 驱动注册和设备创建

底层驱动一般提供形如xxxDrv和xxxDevCeate之类的函数完成驱动的注册和设备创建的工作。这些工作的完成实在内核启动过程中进行的，但是并非得一定如此，只要不影响用户层的最终使用，对于注册和设备创建的时机比较随意（可以等到用户需要使用设备时，由用户自己调用xxxDrv和xxxDevCreate函数完成底层驱动的初始化）。一般而言，xxDrv()形式的函数不带参数，如果需要参数，则函数的名称定义如下：

STATUS xxxInit(int arg,...);

6.3底层驱动服务函数

用户在使用一个设备之前必须要先打开这个设备，底层驱动响应函数中根据设备的需要将进行中断注册和使能设备工作配置等操作。

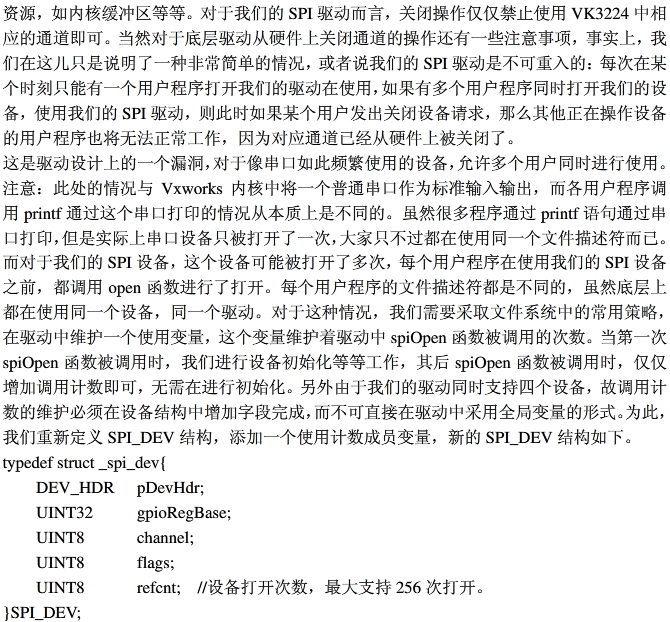
6.3.3设备控制函数

设备控制简单的说用户对于某些工作行为的再配置。基于设备的类型，这些由驱动和IO子系统提供给给用户的再配置参数会有所差别。Vxworks（其它通用操作系统也是如此对各种类型的设备都抽取了一组共同属性作为配置选项，如串口波特率再配置就是一个串口标准属性事实上，虽然有所约定，但是底层驱动完成可以按照自己的标准对这些再配置属性进行选择；可以选择只实现其中某些再配置参数，可以按照特定设备的特殊情况选择对某个再配置选项的响应方式或者转移再配置参数等。。可以说，设备控制函数即提供给了用户控制设备的方便性，也对底层设备的实现提供了极大的方便性，当然，底层驱动程序员不可以欺骗用户，必须要完成用户要求的基本配置要求方可根据需要再做一些辅助性的配置工作，这是底层驱动设备控制实现函数的基本原则。除了vxworks操作系统本身提供的控制参数外，对于一个特定的设备也可以有自己的特定参数，这些也可以作为选项提供给用户进行控制。一般而言，底层驱动需要定一个头文件，将设备的特定参数在其中定义，而后将这个头文件提供给用户程序，当用户对设备进行操作时，其包含这个头文件，使用其中定义的特定参数对设备进行控制。IO子系统实际上不加任何改变的将用户使用的选项参数或者控制命令传递给了底层驱动，由底层驱动完成对选项参数或者控制命令的解释和使用。再配置参数和命令有很多，所有的选项和命令都将统一定义在一个头文件当中，同时提供给驱动本身和用户层使用。

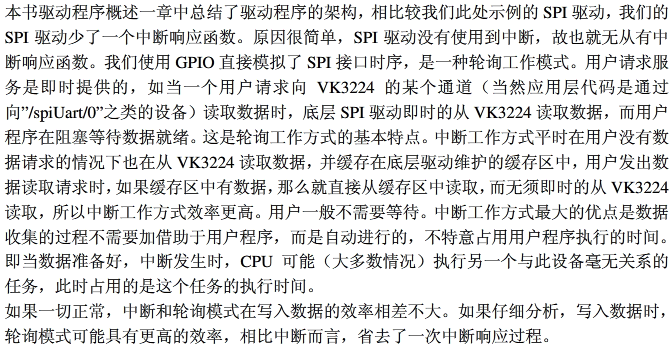
系统中的每一个设备驱动都需要通过头文件的方式提供设备选项或者是设备命令给应用层用户使用，这些选项或命令不需要在数值上做到不同，因为不同的选项将被不同的用户使用，不会在驱动之间造成冲突

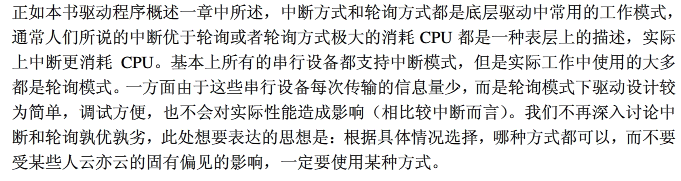
6.3.4 设备关闭函数

对设备操作完成以后，最后需要关闭设备，释放资源，一个设备被关闭，驱动需要做的工作将根据设备类型的不同而差别很大，有一些设备仅仅需要配置硬件相关寄存器将工作使能位清除即可，而有些设备除了硬件相关的寄存器以外还需要释放一些在打开操作中申请的内核资源，如内核缓冲区等等。



有一点需要记住的是spiRead、spiWrite、Ioctl、spiCl;ose函数的第一个参数都是spiOpen函数的返回值，IO子系统将spiOpen的返回值进行保存，并将其作为第一个参数传入以上的四个函数当中，所以spiOpen函数必须要返回一个设备结构类型的指针，这是所有驱动都应该要遵循的约定，否则其后的操作会没有操作对象。当然很多驱动将设备结构作为一个全局变量在驱动的各个函数中使用，这种的编码方式不值得提倡，当一个驱动同时驱动多个相同的设备时，就可能造成问题，当然，驱动此时可以维护一个设备结构数组，对每一个设备使用数组中的一个元素，但是这种方式在进行read、write、ioctl、close操作时依然无法对各个设备进行区分，所以底层驱动中open函数的返回值是至关重要的。驱动程序员要特别注意，必须要使用一个有意义的可以区分设备的参数作为返回值（当然在驱动中使用全局设备结构变量时，可以不用返回设备结构，但是还是需要一个区分设备的参数。）

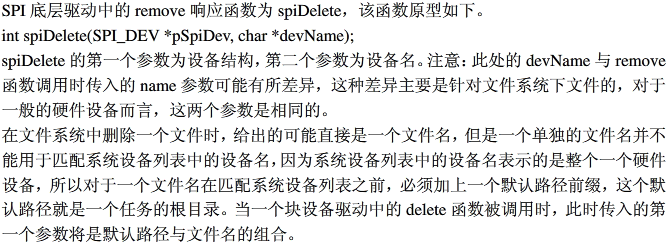




一旦一个设备被创建以后，一般在系统的整个运行期间，是不需要删除的，这个设备将一直存在于系统的设备列表当中，同样设备驱动也将一直存在于系统的驱动列表当中，但在某些特殊情况下，我们在使用完设备以后需将设备从系统设备列表中删除，也将设备设备驱动从系统驱动表中注销。

6.3.5 设备删除函数

删除设备：将设备从系统列表中删除，底层驱动中所有为设备分配的资源将被删除，包括设备结构占用的内存空间。设备的删除函数调用之后，系统中所有该设备相关的信息将被清除，虽然底层驱动中可以提供设备删除函数的实现，但是普通用户通常没有权限删除一个设备。用户层删除一个设备的标准接口函数是remove()；



spiDelete函数需要完成以下的功能：

1. 检查设备是否正常被使用，如果还有用户在操作设备，则直接返回。（要被删除的设备首先得是一个已经关闭了的设备）。
2. 调用iosDevDelete将设备从系统设备列表中删除。
3. 释放底层驱动中为设备分配的其他任何资源
4. 释放设备结构体本身占用的内存资源。

实际上，create、remove一般只用于文件系统管理下的块设备当中，某一个文件的创建或者删除。对于普通的设备文件，底层驱动并不需要提供这两个函数的实现。

6.4 设备的卸载和驱动卸载

虽然在SPI底层驱动中实现了一个设备的删除函数spiDelete，但是这并不是合适的处理方式。事实上系统的驱动表当中de\_delete函数指针指向的实现只在文件系统中间层驱动中一般才有效，用于删除块设备上的一个文件或者目录，而不是用于删除整个的设备。SPI底层驱动是一个字符设备驱动，直接受IO子系统的管理没有通过文件系统中间层，所以实现spiDelete更多的是帮助理解其目的，在实际的应用当中不会这样实现。我们使用spiDevCreate函数创建了一个设备，事实上删除一个设备应有的实现并不属于底层的驱动服务函数集合中，而应该是与spiDevCreate函数位置平行的spiDevRemove函数，该函数用于删除设备，其取消spiDevCreate函数所做的工作。

由于设备总是存在的，软件上无法做到删除一个设备，所以我们给出的更精确的说法是：卸载设备，即设备的软件设施全部销毁。同样在卸载设备之后，我们可以进一步的卸载驱动。

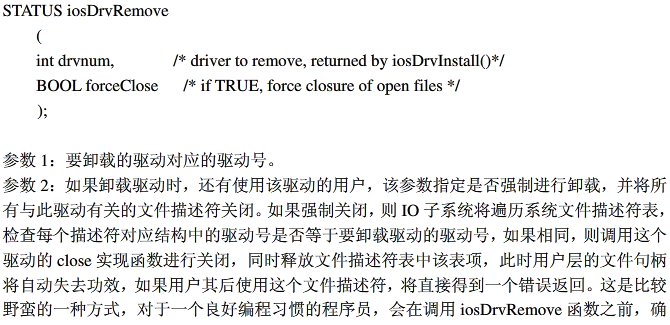
卸载函数完成的功能：

1. 设备节点从系统设备列表中删除
2. 释放xxxDevCreate函数分配的所有资源，包括设备结构本身。



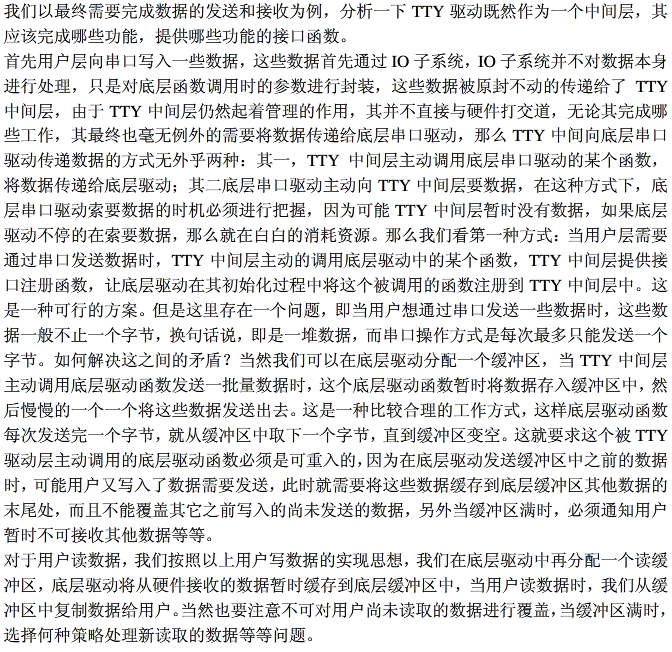
6.4.2 卸载驱动

在spiDrv中我们使用iosDrvInstall函数向IO子系统注册了我们的驱动，同时IO子系统也提供了另外的一个相反作用的函数注销我们的驱动，这个函数就是iosdrvRemove,



1. 串口驱动

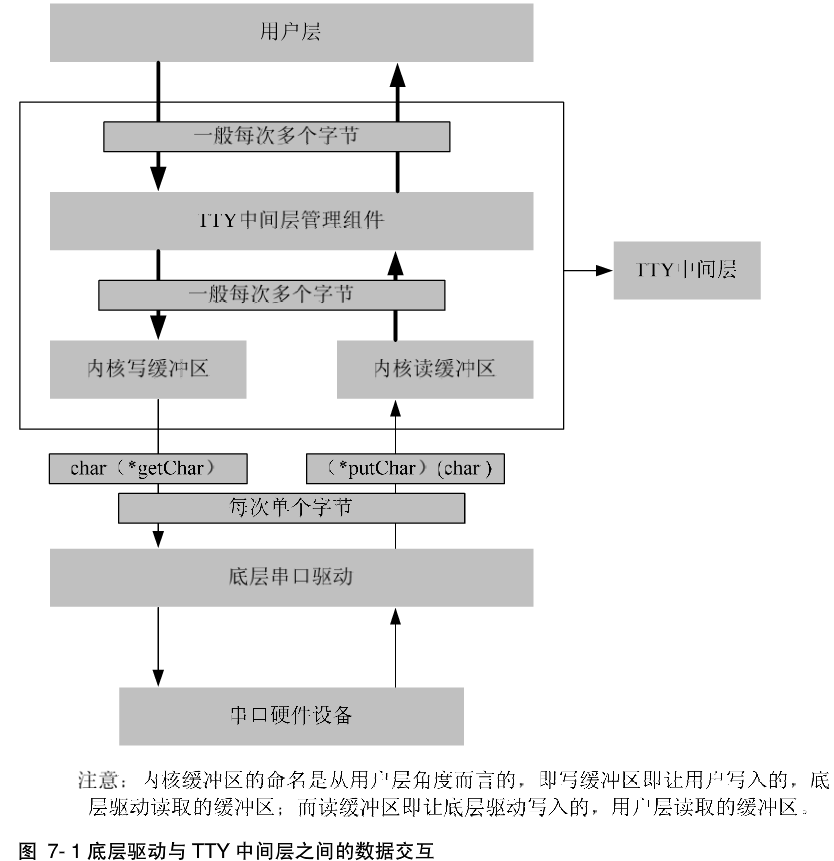
串口是一类非常常用的典型的字符设备，正是由于其常用性，所以Vxworks在IO子系统层之下提供了一个TTY内核驱动中间层，用以管理串口驱动，这就使得底层串口驱动结构与一般的字符设备有所差异。



对于以上的硬件设备的读写操作，现实中很多硬件的速度跟不上用户的读写速度，基本上都是这么实现的，那么问题是，既然底层的驱动需要作这么多工作，而且在效率上也是可行的，那么TTY中间层存在的必要性在哪里，实际上，vxworks中将底层驱动中维护的读写缓冲区从底层驱动中分离了出来改由内核本身进行维护，这个进行维护的模块我们就称为TTY中间层，所以TTY中间层主要是为了简化串口这个常用设备的驱动编写而专门设计的一个内核模块，用它来管理本应该由驱动来管理的数据读写缓冲区，提供回调函数给底层驱动从缓冲区中读数据以及将从硬件接受的数据存入缓冲区中。读写数据的接口函数将在串口驱动的初始化过程当中，由TTY中间层提供给底层驱动。

基于TTY层，重新设计底层串口的实现：

用户向串口写入一些数据，这些数据先传递给IO子系统，IO子系统原封不动的将这些数据传递给了TTY中间层，由于底层串口每一次只能发送一个字节，TTY先将这些数据缓存到他维护的缓冲区中，然后主动调用底层驱动提供的一个数据发送接口函数，发送一个字节，每调用一次就发送一个字节，这样就需要不停地调用底层驱动函数。这种实现基本上可行，但是对于用户要从串口上读取数据时，这种实现就不可行，因为对于用户来说数据的到来时无法确定的，而对于串口这一类的设备而言，很多的时间都处于IDLE的状态，即没有数据的收发行为，如果让TTY不停地调用底层驱动的某个接口函数从硬件读数据，将会很浪费CPU的资源。且效率低下，最好的方式莫过于让底层驱动在有数据到达的时候，主动将其提供给TTY层，此时可以让TTY层提供一个接口函数，该函数接受一个字节，将其放入缓冲区。基于中断的工作方式，每次硬件接受一个字节，底层驱动就调用TTY层提供的这个接口函数将这个字节存入到内核缓冲区，既能做到接收到实时性，也免去了TTY中间层的不断轮询。而实际上对于串口写数据的情况，TTY的实现上也是摈弃了由TTY中间层不断调用底层驱动接口的方式，而是改由底层驱动不断的调用TTY中间层提供的从内核缓冲区读数据的方式。即TTY对串口数据的读写提供了两个接口函数给底层驱动：1.内核写缓冲区读取函数 2.内核读缓冲区写入函数。



从上图可看到，底层驱动维护两个函数指针，存储TTY中间层提供的内核缓冲区读写函数，其中getChar函数指针存储从内核写缓冲区读取的函数地址；putChar函数指针存储向内核读缓冲区写入数据的函数地址。这两个函数的每一次操作都是以一个字节为单位。

最后一个问题是，如何处罚底层驱动主动的发送数据。对与底层驱动接收数据是由串口设备的中断来触发的。每次串口硬件设备从外界接收到一个字节，其发出一个中断，底层的中断响应函数就来处理这个中断，从串口的数据寄存器当中读取这一个字节，再调用putChar指向的内核读缓冲区写入函数将这个字节写入内核读缓冲区。完成串口对外界数据的读取操作。对于向外界发送数据，TTY确实也提供了一个内核写缓冲区读取函数，可以让底层驱动不停地从这个缓冲区中读取数据，进而通过串口发送出去，但是一定要有一个触发点，即第一次从内核写缓冲区读取数据是如何触发的。底层驱动本身不可能进行触发，如由底层驱动触发，那么就又回到轮询模式了。而缓冲区由TTY层来维护，只能通过TTY层的接口函数对缓冲区进行操作。但是实际上TTY只提供了一个写缓冲区读取和一个读缓冲区写入函数，并没有缓冲区为空或者满状态的查询函数，当写缓冲区读取函数返回空的时候表示写缓冲区为空。首先这个触发的目的是为了让底层驱动知道写缓冲区当中现在有数据了，需要将这些数据发送出去。那么其基本实现上定是TTY中间层调用底层驱动当中的某一个函数，这个函数一旦被调用就意味着缓冲区当中存在中数据要发送，此后底层驱动就可以不停的调用getChar函数指针来读取数据直到读取为空为止。这才是真正的TTY中间层当前的实现方式，底层驱动在初始化中除了向TTY层要了两个函数putChar和getChar，还要提供一个函数给TTY层

经以上分析可得出TTY中间层与底层驱动之间的关系：

1. TTY中间层向底层驱动提供两个内核缓冲区读写函数，使得底层驱动可以从TTY层读取需要通过串口发送的数据以及向TTY层写入从串口设备接收的数据；
2. 底层驱动向TTY中间层提供一个数据发送触发函数，从而使得当内核写缓冲区中从空变为非空时，可以通知到底层驱动，将这些数据发送出去。

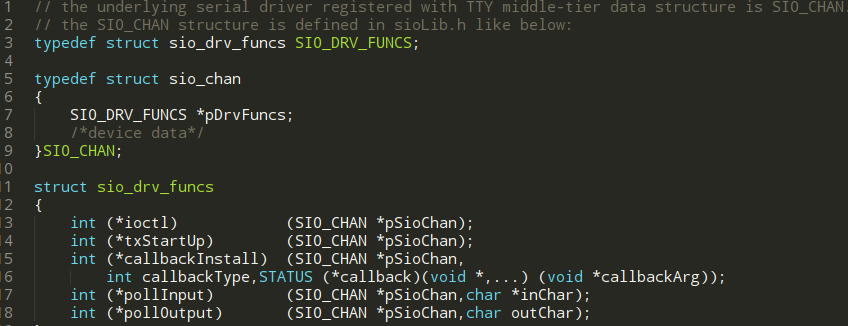
7.1 数据结构

TTY中间层和底层的驱动之间存在一个相互注册的过程。通常非内核组件向内核组件的注册通过以下的两个步骤:（1）初始化内核提供的一个特定数据结构。（2）以这个初始化后的特定的数据结构调用内核提供的接口函数。

内核组件向非内核组件的注册则是直接输出相关信息。

底层串口驱动向TTY中间层注册用的数据结构就是SIO\_CHAN结构。

SIO\_CHAN是一个封装后的数据结构，其实际上就是一个sio\_drv\_funcs结构类型。



1. txStartUp是一个发送触发函数指针，
2. Ioctl：设备控制函数工作机制如同一般的直接受IO子系统管理的字符设备驱动函数。
3. callbackInstall：与数据读写操作相关，由于中间层要向底层串口提供提供两个函数。既然是内核向底层的驱动输出信息，那么通常的实现是内核提供两个接口函数，每次调用输出一个函数地址，由底层驱动保存即可；或者以硬编码的方式，在底层驱动中直接调用这两个函数，因为底层驱动也是作为内核的一部分编入vxworks内核。这不会造成使用上的问题。不过这对系统的升级会造成很大的限制。此处TTY内核中间层的实现采用另外的一种方式，即先将原TTY中间层向底层驱动提供函数的过程变为了由底层驱动向TTY中间层提供函数的过程。由于TTY中间层要向底层的串口驱动提供两个函数，所以必须有一个参数（callbackInstall的第二个参数）指定此次提供的是哪一个函数，其次就是函数本身（第三个参数）是什么，底层驱动提供的这个callbackInstall函数必须在底层驱动中根据函数的不同目的将这两个函数地址保存到底层驱动中，供之后操作内核缓冲区使用。
4. pollInput、pollOutput：这两个函数的目的就是实现上文中分析的由TTY中间层不断的调用底层驱动函数发送缓冲区中数据以及从设备接收数据的功能。即此时底层驱动（1）：不再使用中断主动从设备接收数据，而是完全由TTY中间层来主动索取数据；（2）：不再主动从内核写缓冲区读取数据发送，而是每发送一个字节，都需要TTY中间驱动，而不是院线的单次触发，一直发送。此时不再调用txStartUp指向的触发函数，而是调用pollOutput指向的触发函数，txStartUp只用再单次触发的工作方式之中。这种由TTY中间层完全负责每一个字节的发送和接收过程的工作方式用于使用串口建立主机和目标机之间的调试通道时。如果系统不使用串口作为调试通道，则不需要实现这两个函数。

Sio\_drv\_funcs结构实际上完成了两个方向上的注册功能。作为底层驱动编写的惯例，底层驱动需要一个自己的数据结构用来保存底层硬件固有的信息和关键的参数，当然为了服务内核组件的需要，同时也需要一个内核提供的数据结构，这个结构就是sio\_drv\_funcs结构。即SIO\_CHAN.