VxWorks下所谓的内核态仅仅有一个布尔变量kernelState表示，当kernelState设置为TRUE时，表示此时的代码运行在内核状态，Vxworks内核态的本质即保护内部的数据结构，防止多处代码对内核数据结构同时进行访问，所以它不同于通用操作系统内核态的概念。通用操作系统内核态是为了保护内核数据结构不受应用层的影响。所以VxWorks内核态更多的是一个信号量的概念。

当代码需要对内核的数据结构进行访问的时候，其首先要检查的是kernelState的状态，若kernelState设置为TRUE，则表示当前已经有代码在操作内核的数据结构，故将当前的代码要做的工作加入到内核工作队列当中。稍后再完成。

设备驱动函数是执行在内核环境下的这是一个笼统的说法。实际上，驱动代码具有两个执行环境，一个是任务上下文，一个是中断上下文。除了中断响应函数执行在中断上下文之外。其他所有的函数都是执行在任务上下文中。这个任务上下文对应的任务就是当前对设备进行某种服务请求的用户任务或者是内核任务。在vxworks中处于一个任务的上下文中仅仅是指在执行驱动的代码时可以被挂起，代码使用的栈是任务栈，代码可以被中断抢断，包括其自身中断。所以，如果驱动中的某个函数与驱动中断处理程序共享同一个资源，则要避免形成资源的破坏，在这种情况下，大多使用intLock和taskLock组合保护任务上下文。

Vxworks不支持驱动的动态加载，驱动程序一般实现为内核代码的一部分，故安全性方面的问题较少，但是在编程的时候应该对用户输入的任何参数都必须经过检查以后才能使用，且对不合法的或者是不合理的参数，必须要终止服务。

4.3.4 驱动基本工作模式——轮询和中断

设备驱动从总体上可以分为两种工作模式：轮询和中断。轮询模式通过检测相关寄存器的状态位来决定是否进行下一步操作；而中断则用来通知某一个操作已经完成，可以经行下一个操作。二者的根本区别在于轮询方式在等待操作完成的过程中需要CPU等待，而中断则将CPU从等待中解放出来，在硬件完成一个用户请求的过程中，CPU可以运行其他任务，当硬件完成一个请求以后，发出一个中断，再次引起驱动的注意。驱动可以在中断响应函数中完成下一个操作。

以串口为例，其支持中断工作模式，但是使用中断有如下的缺点：如果没发送或者是接收一个字节就产生一次中断，由于中断响应需要消耗资源，频繁的中断不但不能加快数据的收发数率，而且会极大的影响整个系统的性能。

通常情况下，串口只是用于少量信息传输的实现，vxworks下，Shell通常也是建立在串口之上的，所以需要使用串口传递信息量较少的命令。此时使用FIFO就不合适，故大多数情况下，串口驱动都工作在轮询模式之下。以发送为例，每次驱动将一个字节的数据写入串口发送寄存器后，就不断的检查状态寄存器，查看这个字节是否发送出去。若没有，则不断的进行查询。一段查询到这个字节已成功发送，则取下一个待发送的字节进行发送。直到当前内核串口缓冲中所有的字节都发送完毕。这种方式可以将写入串口的数据及时的通过串口打印出来。

可以总结如下，中断使用在数据量大的场合；轮询使用在数据量较少的场合。

驱动的注册和设备的创建通常是指通过调用iosDrvInstall和iosDevAdd将驱动和设备分别添加到系统的驱动表和系统的设备表的过程。根据设备的使用范围不同，这个注册和创建的时机也不同。一般而言，对于标准设备（串口），驱动注册和设备创建在内核的初始化过程中完成；对于一些驱动中间层，驱动注册要在内核初始化中完成。而设备的创建要在下层驱动注册时完成。

字符设备是一类比较简单的设备，其以字节流的方式对数据进行操作，数据只能顺序的读写。UART等接口类型的设备都可以作为字符设备来进行驱动，字符设备位于Vxworks内核I/O子系统的直接管理之下，不经过任何的中间层，只得注意的是，内核为了简化某一些常用字符设备驱动的设备，也会提供一个中间层作为缓冲，如TTY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，使用TTY层能够提高设备的使用效率，因为UART串口设备在内核的启动过程当中默认被设置为标准的输入输出通道。在整个操作系统的运行过程中使用比较频繁。

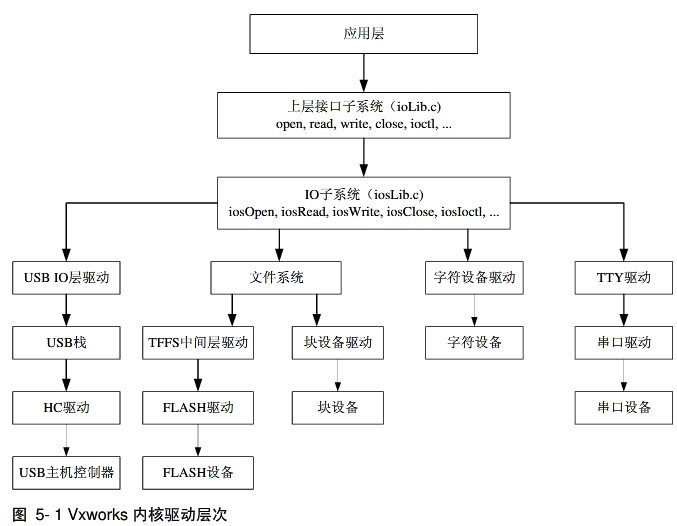
驱动的工作模式

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

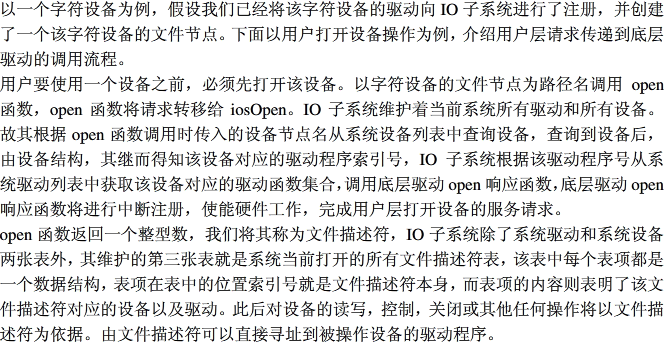
Vxworks对应用层提供了一套标准文件操作接口函数，将其称为标准IO库，vxworks下由ioLib.c文件提供。ioLib.c文件提供如下的标准接口函数：create、open、unlink、remmove、close、rename、read、write、ioctl、lseek、readv、writev等。Vxworks操作系统区别于通用操作系统的一个很大的不同点是vxworks下不区分用户态和内核态，用户层程序可以直接对内核函数进行调用而无需使用指令中断之类的机制或者存在使用权限上的限制。所以vxworks提供给应用层的接口无需通过外围库的方式，而是直接以内核文件的形式提供。这是有操作体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



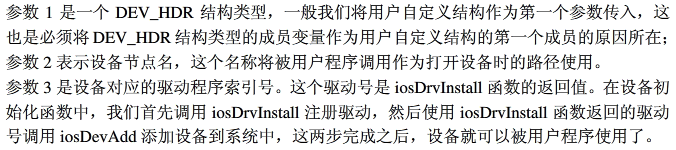
字符设备和块设备均由IO子系统进行管理。而网络设备由于其特殊的工作方式将由另一套的用户标准接口函数（套接字函数）和内核网络栈进行支持。

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。

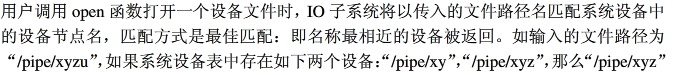
IO子系统在整个的驱动层次当中起着管理的功能，其维护着系统的设备和驱动的关键的三张表。



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：

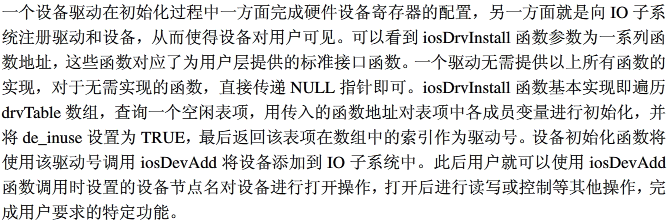


底层的串口驱动是通过TTY来进行管理的，对不同的串口的操作在TTY驱动层才进行分离，所有的串口驱动都要先通过相同的TTY驱动的层的处理，而后请求才被转发（转发的具体过程是怎么样的？？？）到具体的底层的串口驱动。可在命令行中使用iosDevShow或devs显示设备中的所有设备。



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;

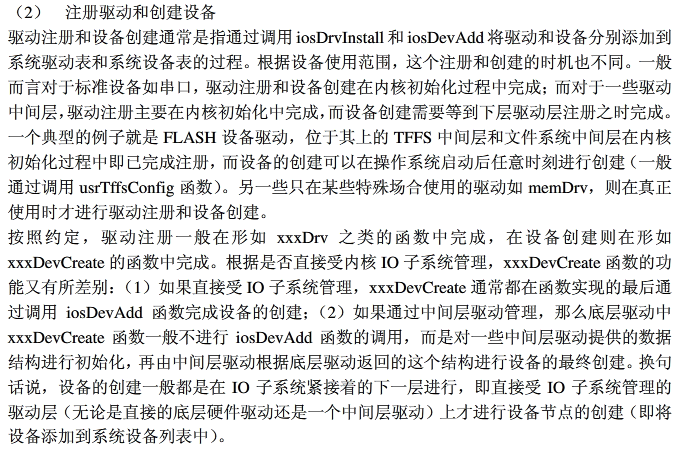
其他两条语句的实际结果即：

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。

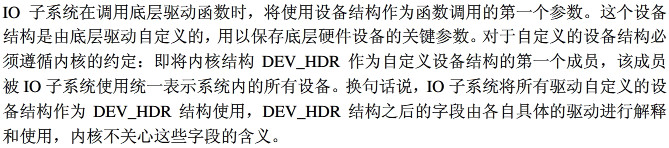
当使用一个文件描述符进行操作时，如调用write函数，内核首先检查文件描述符是否为0，1，2标准输入输出描述符，如果是，则依次为索引查询ioStdFd，以ioStdFd[fd]作为索引查询系统文件描述符表，获得驱动号，进而索引系统驱动表，调用对应表项de\_write指向的函数，完成写入操作。如果文件描述符大于2，表示这是一个普通的文件描述符，那么就直接以该描述符作为索引查询系统文件描述表，获得驱动号，进而索引系统驱动表，调用相关的函数。



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。

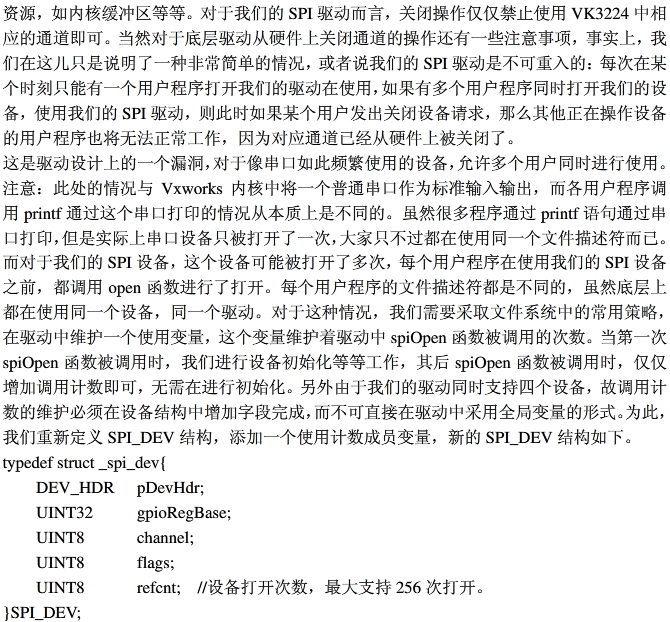
6.3.3设备控制函数

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。

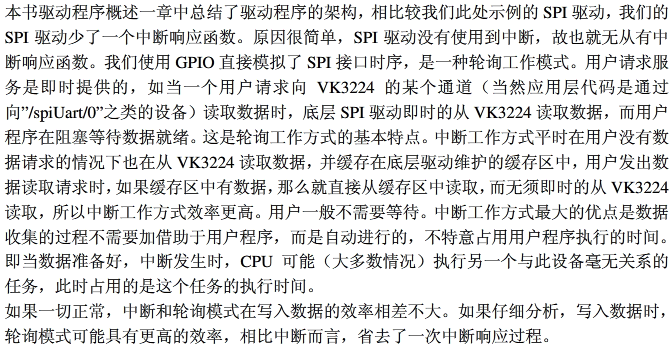
系统中的每一个设备驱动都需要通过头文件的方式提供设备选项或者是设备命令给应用层用户使用，这些选项或命令不需要在数值上做到不同，因为不同的选项将被不同的用户使用，不会在驱动之间造成冲突

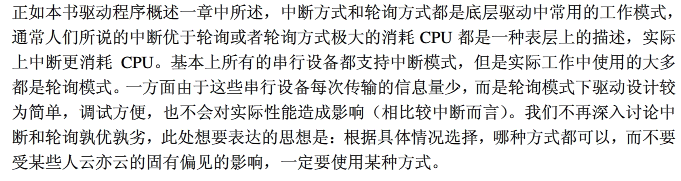
6.3.4 设备关闭函数

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。）

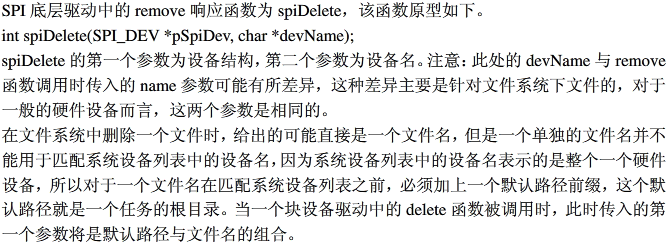




一旦一个设备被创建以后，一般在系统的整个运行期间，是不需要删除的，这个设备将一直存在于系统的设备列表当中，同样设备驱动也将一直存在于系统的驱动列表当中，但在某些特殊情况下，我们在使用完设备以后需将设备从系统设备列表中删除，也将设备设备驱动从系统驱动表中注销。

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。

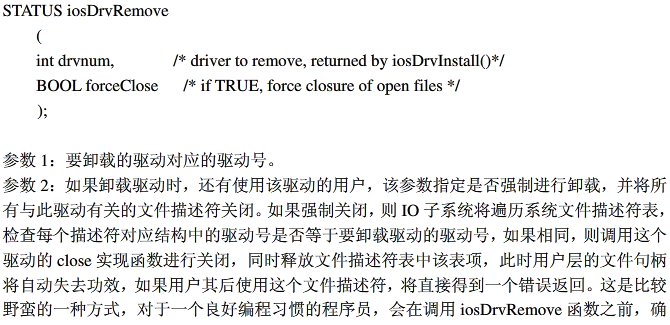
卸载函数完成的功能：

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。



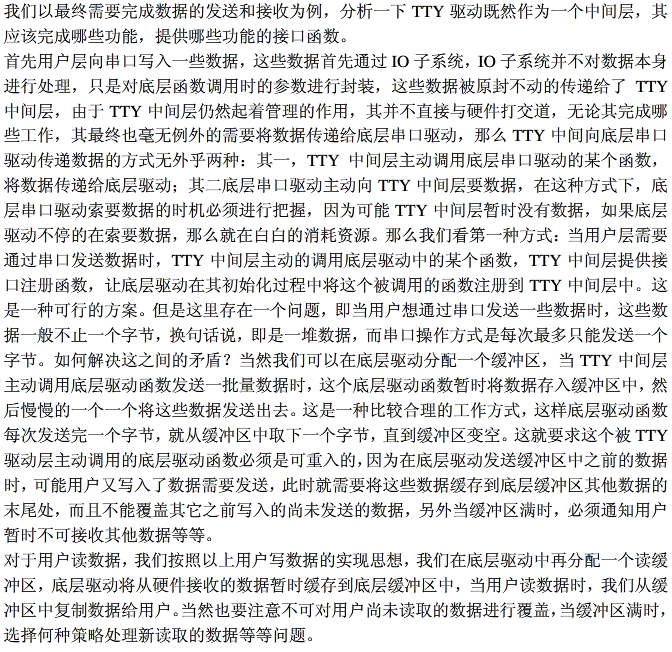
6.4.2 卸载驱动

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

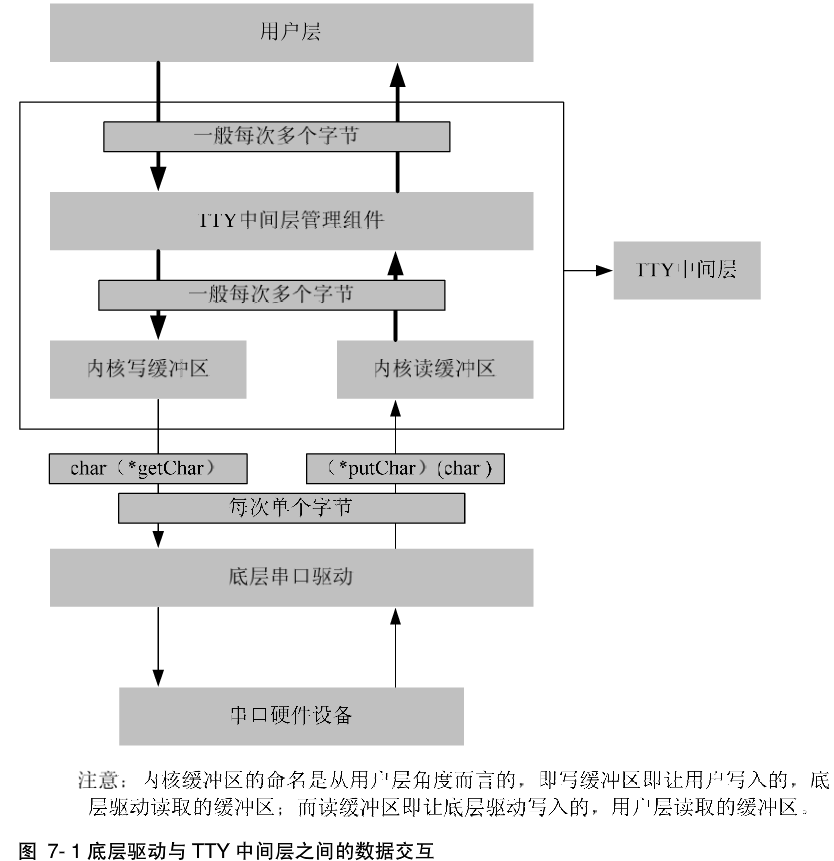
串口是一类非常常用的典型的字符设备，正是由于其常用性，所以Vxworks在IO子系统层之下提供了一个TTY内核驱动中间层，用以管理串口驱动，这就使得底层串口驱动结构与一般的字符设备有所差异。



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。

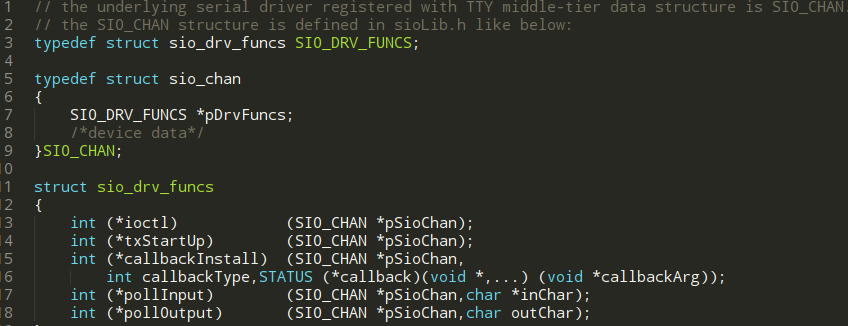
7.1 数据结构

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。

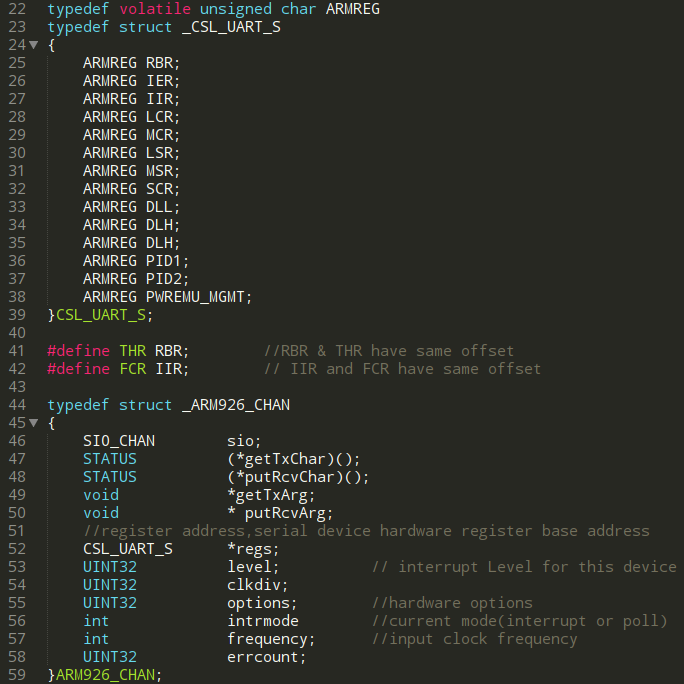
底层串口驱动向TTY中间层注册用的数据结构就是SIO\_CHAN结构。

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.

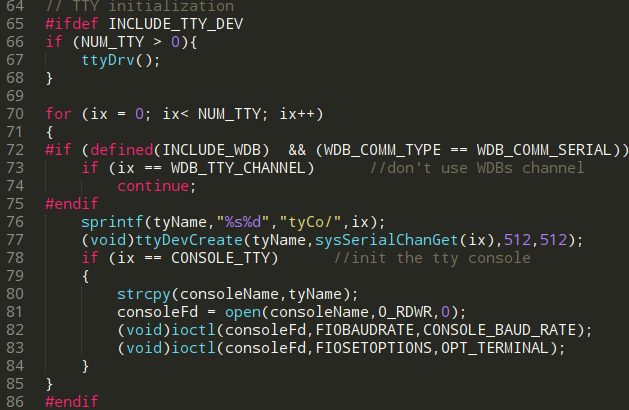


驱动自定义的数据结构如上，Vxworks下内核驱动层次决定了Vxworks下所有的驱动都必须要置于IO子系统的直接或是间接的管理之下，对于串口设备，IO子系统之下是TTY驱动层，TTY之下才是串口驱动，基于IO子系统管理的三张表，首先TTY层要向IO子系统注册注册其驱动中间层，从而使得IO子系统可以访问到TTY层，进而将用户的请求转发给TTY层。IO子系统无法感知到串口驱动的存在，对于所有的串口设备的操作，IO子系统都将调用TTY中间层注册在系统驱动表中的函数进行处理，TTY中间层再将请求传递给底层串口驱动。

因此在介绍真正的串口设备之前，我们必须要对TTY中间层的初始化过程做一定的了解，以及真正的串口驱动是如何的初始化的，串口设备又是如何以及何时添加到系统设备表当中的

7.2 TTY中间层的初始化

TTY作为操作系统的一部分，如同其他内核组件一样，在操作系统的启动当中被初始化，具体的初始化代码定义在usrConfig.c中。如下：



TTY组件的初始化工作由ttyDrv函数完成，该函数的功能是向IO子系统注册驱动。此处即为TTY将自己的驱动函数注册到系统驱动表中，作为所有串口设备的最上层次。

ttyDrv函数执行完成以后，TTY中间层就将中层的驱动注册到系统的驱动表了，现在可以根据平台上串口的数目（TTY\_NUM表示），初始化每一个底层串口驱动，创建对应的设备，并添加到系统设备列表中。这些功能当然是由ttyDevCreate函数完成的。该函数的调用原型为：

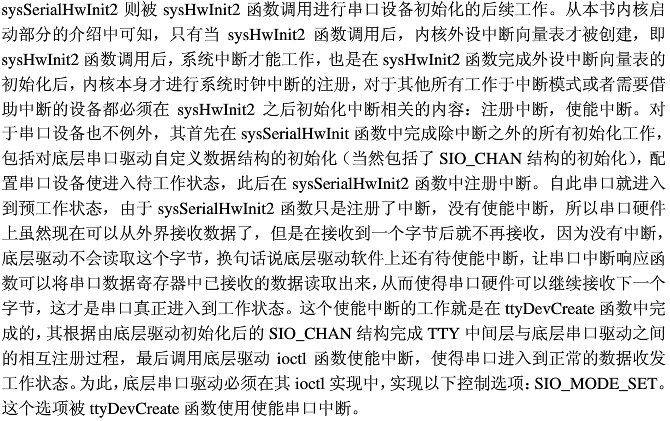


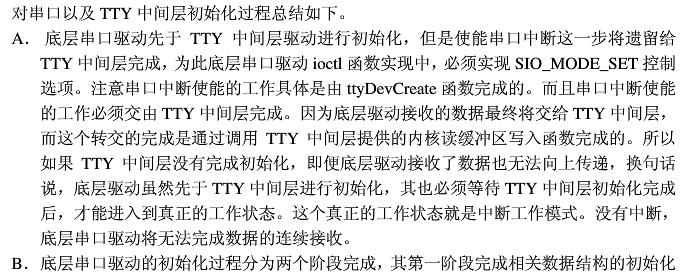
参数一：创建设备时使用的设备节点名

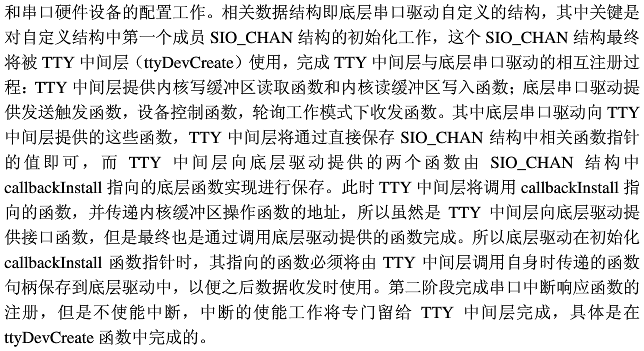
参数二：一个SIO\_CHAN类型的结构指针，事实上，其本质是一个底层驱动自定义的数据结构，SIO\_CHAN结构作为了第一个成员变量，所以同时也可以当做一个SIO\_CHAN使用。需要注意的是，传递给ttyDevCreate的这个SIO\_CHAN必须要是经过底层串口驱动初始化以后的，此时TTY中间层将会根据该结构当中的内容保存底层驱动相关函数的地址，以便此后进行调用，如对发送触发函数（SIO\_CHAN结构当中的tyStartUp函数指针指向）的调用。同时ttyDevCreate还将会调用SIO\_CHAN结构当中的callbackInstall指向函数向底层驱动提供内核缓冲区的两个操作函数。

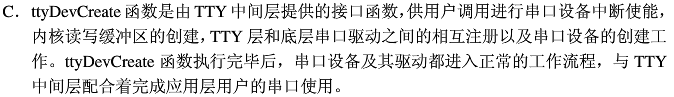
参数三、四：TTY中间层维护的内核读写缓冲区的大小，这两个参数给用户一个指定内核缓冲区大小的手段。

ttyDevCreate函数调用当中最重要的参数是第二个参数：一个SIO\_CHAN结构指针，指向底层驱动自定义的结构。这个SIO\_CHAN是已经经过底层驱动初始化的，即对结构中的函数指针都进行了初始化。在usrConfig.c中调用ttyDevCreate时调用的sysSerialChanGet函数一般定义在sysSerial.c文件中，sysSerial.c文件专门是针对串口设备的实现文件，除了sysSerialChanGet函数外，在这个文件中还定义了如下重要的函数：sysSerialHwInit，sysSerialHwInit2；以及一个数组：sysSioChans，该数组维护着系统内所有的串口设备对应的SIO\_CHAN结构地址。实际上sysSerialChanGet函数实现就是简单的根据输入参数，返回sysSioChan数组中对应的元素。既然调用ttyDevCreate函数时要求传递一个经过底层驱动初始化以后的SIO\_CHAN结构，而sysSerialChanGet函数仅仅返回sysSioChans数组中对应的元素，并不是完成任何的结构初始化工作，这就表示在调用ttyDevCreate之前，底层的驱动就已经初始化好了，并对驱动设备的对应的SIO\_CHAN结构进行了初始化。实际上，vxworks内核启动的过程中，将会通过调用sysHwInit以及sysHwInit2完成平台外设硬件的初始化（对于sysHwInit的具体调用，参见前面vxworks启动流程章节。）（？？？？？内核启动过程完成初始化？？？不支持热插拔么？？？）。而sysSerialHwInit函数即被sysHwInit调用专门对串口设备进行初始化，sysSerialHwInit2则被sysHwInit2调用进行串口设备初始化的后续工作。



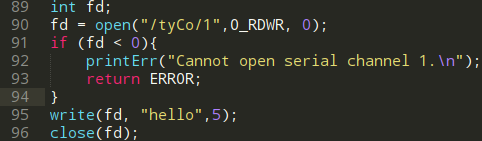




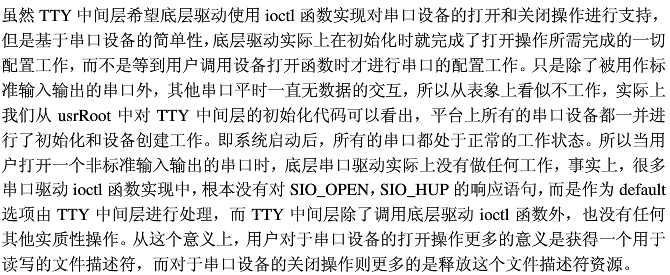


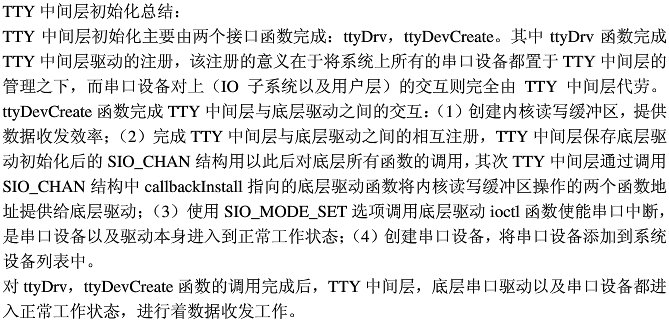
CONSOLE\_TTY表示作为终端使用的串口通道。这个通道将作为所有任务中打印信息的默认输出通道以及系统标准输入输出通道。

注意：ttydevCreate函数将保存传入的第二个参数，此后对于底层驱动的调用都将会使用该参数。这类似于普通的字符设备中对IO子系统调用xxxOpen函数返回参数进行的保存。另外，串口设备通常被作为标准的输入输出设备，即一个任务在调用printf语句打印一些信息到终端的时候，其无需专门打开一个串口设备。但是在系统存在多个串口设备的情况下，用于对于除被用作标准输入输出串口的其他串口进行操作时，就必须要先打开该串口，代码如下：



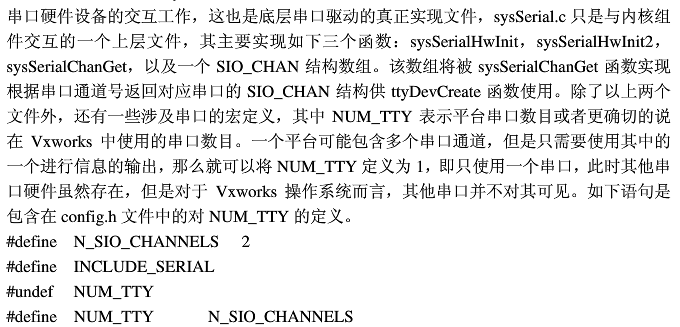
上面的代码段打开串口1（串口0被用作为标准的输入输出），写入“hello”信息。此时我们不能直接使用printf（“hello”）进行输出，因为printf语句将通过串口0将信息发送出去，而不是我们要求的串口1.但是从串口驱动的结构而言，底层的串口驱动并没有open、close函数，那么是怎么实现的呢？实际上，TTY将对串口的open、close请求通过底层驱动中的ioctl函数传递给了底层驱动。用户对一个设备发出open、close请求时，其通过SIO\_OPEN,SIO\_HUP选项分别调用底层驱动的ioctl函数。

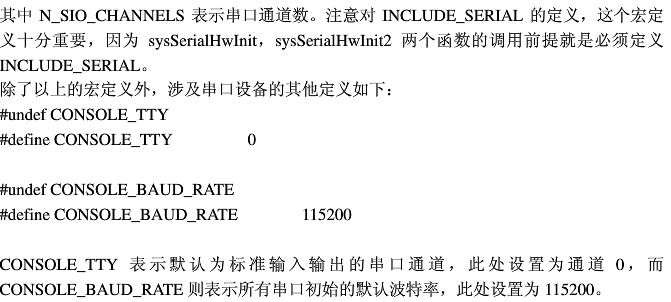




7.3 串口驱动基本的结构

串口驱动设计的文件有sysSerial.c，xxx\_uart.c。其中sysSerial.c文件属于BSP（板级支持包）组成的标准文件之一，一般包含串口设备的平台都需要在BSP中包含这个文件且文件名必须要遵照约定命名为sysSerial.c。Xxx\_uart.c则是具体串口驱动的底层实现代码，完成着与串口硬件设备的交互工作，这也是底层串口驱动的真正的实现文件，





底层串口驱动胡基本结构结构如下：

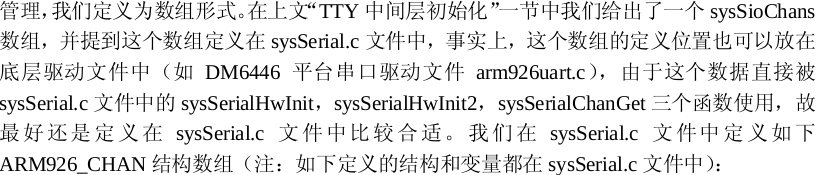
1. ：函数的实现

1：xxxUartInit函数：被sysSerialHwInit函数调用完成驱动自定义结构的初始化，尤其是第一个成员变量SIO\_CHAN结构的初始化，完成串口硬件的配置，使串口进入到待工作的状态．

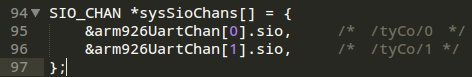
2:xxxUartInit2:被sysSerialHwInit2函数调用完成串口驱动中断响应函数的注册．中断使能延迟到ttyDevCreate函数中进行．

1. xxxUartctl函数：底层驱动ioctl控制函数实现，必须提供SIO\_MODE\_SET选项的实现，该选项的基本功能是使能中断．若底层驱动支持可变的波特率，还必须要实现SIO\_BAUD\_SET选项．关于串口设备的更多选项，查看头文件sioLib.h．
2. xxxUartInit函数：底层中断响应函数，主要实现数据的接收．
3. xxxUartPollInput，xxxUartPollOutput函数：轮询工作方式的实现．
4. xxxUartCallBakInstall 函数：该函数作为SIO\_CHAN结构中callbackInstall成员变量指向的函数，供TTY中间层调用以将ＴＴＹ中间层提供的两个内核环层缓冲区操作函数地址传递给底层驱动．底层驱动此后对用户需要发送的数据获取以及向ＴＴＹ传递接收数据的操作都需要通过这两个函数来完成．底层驱动需要对这两个ＴＴＹ中间层的函数进行妥善的保存．
5. xxxUartTxStartUp函数：由底层驱动提供给ＴＴＹ中间层的发送触发函数．
6. 串口硬件操作函数集合：这些函数用于对串口设备的寄存器进行读写操作。对于这些函数，下面的介绍中我们只交代其实现的功能，而不在给出细致的代码，这部分是与平台相关的。
7. 结构定义：

对于DM6446平台，其上有三个串口通道，我们选用两个进行说明。对于两个通道，要定义两个ARM926\_CHAN结构。为了对所有的串口进行管理，我们定义为数组的形式。

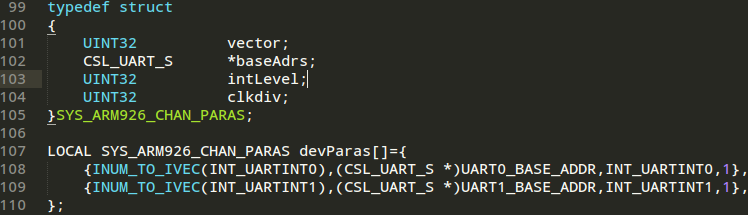


sysSioChans数组的定义如下：



事实上，只要使用arm926UartChan这一个数组就可以满足要求，但是为了表示不同使用方式（sysSioChans数组由sysSerialChanGet函数返回，供ＴＴＹ中间层使用；arm926UartChan数组由sysSerialHwInit和sysSerialHwInit2使用；二者本质上是指向相同内存区域的）。

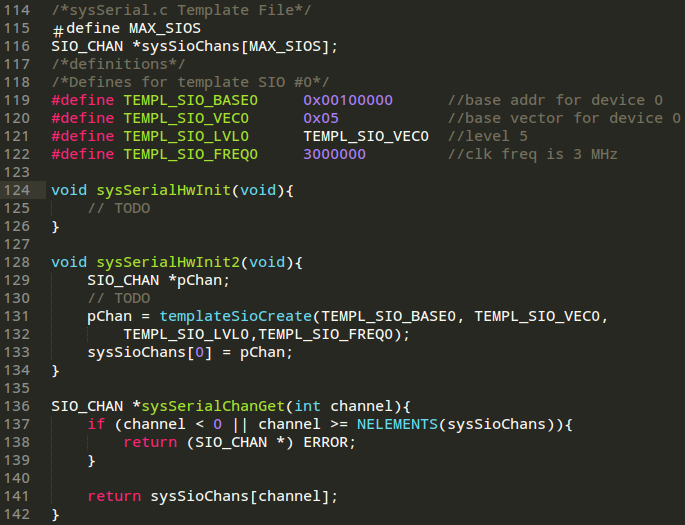
为了能够在sysSerialHwInit和sysSerialHwInit2函数使用数组方式进行初始化，我们将串口设备的参数也用数组的方式表示。故定义以下的结构：



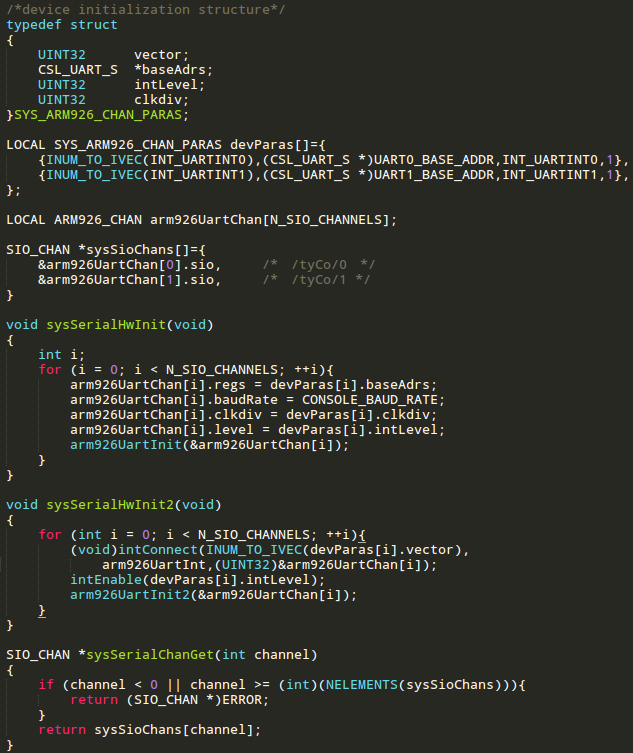
需要注意的是，devParas数组中的类似于INT\_UARTINT0常量定义在平台的头文件（dm6446.h），此处不在给出其具体的定义。

串口的驱动文件一般有三个：sysSerial.c,xxxUartDriver.c,xxxUARTDriver.h.

.h文件主要是一些结构的定义以及寄存器的地址和中断号等等相关参数的定义文件。对于sysSerial.c文件，我们将其称之为串口驱动内核接口文件，这个文件的作用主要是体现在串口驱动的初始化过程中，这个文件的具体实现具有标准的模板，包括函数的命名方式都应该遵照一定的约定。基于ARM平台的sysSerial.c模板文件如下：



针对DM6446平台修改后的sysSerial.c文件的实现如下：



在sysSerial.c中定义的三个函数主要用在用户底层串口驱动的初始化过程中，内核的初始化代码调用sysSerial.c中定义的标准函数接口完成对底层串口驱动的初始化，避免了直接对内核进行修改。sysSerial.c文件中三个函数的实现是与平台相关的，但是基本的代码结构相似，特别是sysSerialChanGet函数基本可以不做改变的使用到任一平台之上。而sysSerialHwInit和而sysSerialHwInit２函数只需要做微小的改动即可，主要是调用底层驱动函数时的改正。

在sysSerialHwInit的实现当中，我们需要初始化每一个串口设备的关键参数：串口设备对应的寄存器基地址，初始波特率，中断号，之后调用底层驱动串口设备初始化函数arm926UartInit,该函数配置串口设备寄存器使得串口处于待工作状态。

sysSerialHwInit 调用完成后,底层驱动已经完成设备硬件的配置工作,并且完成了串口设备SIO\_CHAN 结构的初始化工作。sysSerialHwInit2 函数是串口设备的后续初始化工作。对于使用中断的所有设备,都需要一个二次初始化的过程,在第二次初始化过程中,完成中断注册和使能中断的工作,这个二次初始化的过程是 Vxworks 内核特殊启动方式造成的,而非有意为之,将中断注册和中断使能单独作为第二次初始化过程。具体细节请参考本书前文“Vxworks 启动流程”相关章节。对于一般的设备,如网口,SPI 接口设备,I2C 接口设备等等,在 sysHwInit2 调用期间完成中断的注册和使能,自此设备进入正常工作模式。然而对于串口设备,却有些特殊。虽然串口设备的中断注册确实是在 sysHwInit2 调用期间完成,但是其并不进行使能中断的工作。一个中断的使能原则上包括三个步骤,CPU 系统中断位使能,这个已经在 usrRoot 任务创建时完成,其次中断控制器对应中断通道使能,再次设备中断使能。由于 CPU 可用中断输入管脚有限(通常为 1 个,ARM 为 2 个,但是 Vxworks 只使用 IRQ,不使用 FIQ),所以通常都使用各外部设备管理平台上所有设备的中断,这个外部设备我们就称之为中断控制器,中断控制器内部对每个中断通道都有一个中断使能位对应,要使该通道发出的中断可以传递给CPU 中断管脚,则对应的中断使能位必须开启。另外每个设备都有自己的中断控制寄存器,控制中断的发出与否。所以中断开启一方面需要使能中断控制器的相关通道,还需要使能设备本身。对于串口设备我们可以在 sysHwInit2 调用期间,使能中断控制器,但是设备使能的工作留给 ttyDevCreate 函数,或者将这两个工作都留给 ttyDevCreate 调用时完成。总之,从工作角度而言,却没有进入到中断模式。串口设备的二次初始化具体是在 sysSerialHwInit2 函数中完成的,该函数将被 sysHwInit2 函数调用专门对串口设备进行二次初始化。从此处可以看出,Vxworks 内核对串口设备确实是“情有独钟”,对串口设备驱动提供 TTY 中间层,初始化过程也使用专门的函数(sysSerial.c 文件中定义的 sysSerialHwInit,sysSerialHwInit2)。

在如上代码实现中,sysSerialHwInit2 函数完成了中断的注册以及中断控制器通道的使能工作,而只将设备的中断使能留到 ttyDevCreate 函数调用时完成。至于对于 arm926UartInit2底层驱动函数的调用,则是让底层驱动在二次初始化过程中进行一些其需要完成的额外工作。通常,底层驱动二次初始化函数实现为空,即 arm926UartInit2 函数简单返回,不做任何实质性工作。

sysSerial.c文件中最后一个函数sysSerialChanGet被内核调用，用来获取ttyDevCreate函数调用时所需的设备结构参数，具体的调用语句如下；

注意 sysSerialChanGet 函数名称是预先约定的,换句话说,为了能够让内核启动过程中正常的初始化 TTY 中间层,BSP 程序员必须在 sysSerial.c 文件中定义一个名为 sysSerialChanGet的函数,该函数返回一个 SIO\_CHAN 结构指针,指向由参数指定的串口设备的数据结构。

本节小结:

如果平台使用串口,则必须在平台 BSP 目录下定义一个 sysSerial.c 文件,该文件中必须实现如下三个指定名称的函数:sysSerialHwInit,sysSerialHwInit2,sysSerialChanGet,一般还需要一个 SIO\_CHAN 结构数组。其中 sysSerialChanGet 函数和 SIO\_CHAN 结构数组在 TTY中间层初始化过程中被调用用以获取每个串口设备对应的 SIO\_CHAN 结构,从而使得 TTY中间层完成其与底层驱动之间的交互;sysSerialHwInit 和 sysSerialHwInit2 则在串口设备本身的初始化过程中被调用,完成每个串口设备对应 SIO\_CHAN 结构的初始化以及串口设备硬件寄存器的配置和中断注册过程。

基于串口设备的普遍性和常用性,Vxworks 内核提供了 TTY 中间层对底层串口驱动进行管理。在 Vxworks 启动过程中,为了便于初始化底层串口驱动,Vxworks(应该是 Wind River)提供了 sysSerial.c 内核接口文件。实际上我们也可以将 sysSerial.c 文件视为 TTY 中间层的一个组成部分,从而使得 sysSerial.c 文件显得更为正式一些。在完成对 TTY 中间层初始化以及 sysSerial.c 文件的介绍后,下面就进入到串口驱动代码的实际编写了。上文中我们已经对串口驱动的组成进行了说明,而且对于串口驱动和 TTY 中间层之间的交互关系也做了较为详细的分析,基于这些理解,串口驱动本身的设计就只是编写代码了。下一节我们将以 TI DM6446 平台串口驱动为例,详细介绍底层串口代码的实现。

7.5 串口驱动函数的实现

基于上下层的关系,我们的串口驱动需要实现如下函数:

A. arm926UartInit 函数:被 sysSerialHwInit 函数调用完成驱动自定义结构的初始化,尤其是第一个成员变量 SIO\_CHAN 结构的初始化,完成串口硬件的配置,使串口进入到待工作状态。

B. arm926UartInit2 函数:被 sysSerialHwInit2 函数调用完成串口驱动中断响应函数的注册。中断使能延迟至 ttyDevCreate 函数中进行。

C. arm926UartIoctl 函数:底层驱动 ioctl 控制函数实现,必须提供SIO\_MODE\_SET 选项的实现,该选项基本实现功能是使能中断,包括系统级中断使能以及硬件中断使能两个方面的使能工作(实际上可以将系统级中断使能放入 xxxUartInit2 函数中进行)。如果底层驱动支持可变波特率,还必须提供 SIO\_BAUD\_SET 选项的实现。关于串口设备的更多选项,请查阅内核头文件 h/sioLib.h。

D. arm926UartInt 函数:底层驱动中断响应函数,主要实现数据的接收。

E. arm926UartPollInput,arm926UartPollOutput 函数:底层驱动轮询工作方式实现函数,这两个函数提供了一种工作模式由 TTY 中间层负责每一个字节的接收和发送,此时底层驱动完全处于被动调用。这种模式一般在串口通道被用作调试通道时使用。

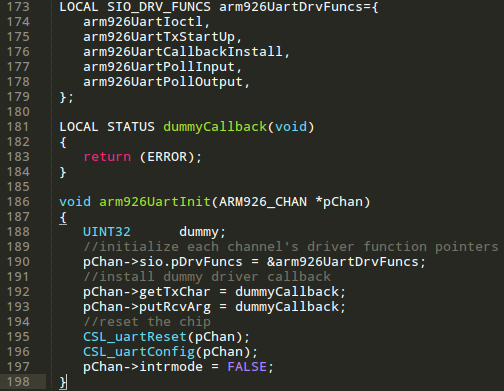
F. arm926UartCallbackInstall 函数:该函数作为 SIO\_CHAN 结构中callbackInstall 成员变量指向的函数,供 TTY 中间层调用用以将 TTY 中间层提供的两个内核缓冲区操作函数地址传递给底层驱动。底层驱动此后对用户需要发送数据的获取以及向 TTY 中间层传递接收数据的操作都需要通过这两个函数完成,底层驱动需要对这两个 TTY 中间层提供的函数进行妥善保存。

G. arm926UartTxStartup 函数:该函数是由底层驱动提供给 TTY 中间层的发送触发函数。这个函数将用以初始化 SIO\_CHAN 结构中 txStartup 函数指针,从而在 ttyDevCreate 函数调用中传递给内核 TTY 中间层。

H. 串口硬件操作函数集合:这些函数用以对串口设备寄存器进行读写操作。对于这些函数,下文介绍中我们只交代其实现的功能,而不再给出细致的代码,这部分代码平台相关。

7.5.1 初始化函数实现

串口驱动中初始化函数主要有两个：arm926UartInit,arm926UartInit2,arm926UartInit将被sysSerialHwInit函数调用完成串口设备结构的初始化工作以及串口设备硬件寄存器的配置，使得串口设备进入待工作的状态，arm926UartInit2被sysSerialHwInit2调用。由于啥意思sysSerialHwInit2函数本身在arm926UartInit2函数之前就已经完成了串口设备的中断注册以及中断控制器相关通道的使能工作，所以arm926UartInit2函数就没有实质性的工作需要完成，实现中将直接返回。除了以上两个比较明显的函数外，在初始化的过程中被TTY中间层直接调用的函数有arm926UartIoctl,arm926UartCallbackInstall。其中arm926UartCallbackInstall被 TTY 中间层使用向底层串口驱动提供两个内核缓冲区操作函数;而 arm926UartIoctl 函数则被调用设置一些串口设备工作参数,如波特率,还有一个最为关键的工作是通过 arm926UartIoctl 函数使能串口设备中断。sysSerialHwInit2 函数中完成的是对中断控制器相关通道的使能,而此处完成的是对设备本身的中断使能。

（1）：Arm926UartInit实现的功能为：初始化串口设备结构，配置串口设备寄存器，具体代码如下：  


SIO\_CHAN结构本质上是一个SIO\_DEV\_FUNCS结构。Arm926UartInit函数实现的第一条语句就是对SIO\_CHAN结构初始化。Arm926UarDrvFuncs变脸是一个驱动内部全局变量，指向驱动内部实现的需要提供给TTY中间层的五个函数：ioctl，txStartUp，callbackInstall，pollInput，pollOutput。每个函数在TTY中间层都有使用的意义。callbackInstall函数是底层驱动提供的一个回调函数，只在初始化的过程中被调用两次，TTY中间层使用其向底层驱动提供两个内核缓冲区操作函数。完成SIO\_CHAN的初始化以后将底层驱动用来保存TTY中间层提供的内核缓冲区操作函数的两个函数指针初始化为指向一个dummy函数，防止内核启动异常可能造成的未初始化问题，此后在使用时造成空指针的引用，此处将其初始化为指向dummy函数可以避免这个问题。而对于ARM926\_CHAN结构中的getTxChar，putRcvChar这两个函数指针的真正的初始化将在TTY中间层调用arm926UartCallbackInstall函数当中完成。最后完成串口设备寄存器的配置工作,这些工作由 CSL\_uartReset 和CSL\_uartConfig 函数完成,这些代码与具体平台相关,我们不在给出其具体实现代码。由于尚未使能中断,故设置 intrmode 成员变量为 FALSE,表示处于非中断工作模式(实际上也不是轮询模式,可以称为待工作模式)。

1. ：arm926UartInit2函数实现。

第二次初始化工作，这次初始化将进行中断的注册以及中断控制器串口通道的中断使能工作。此处由于不做实际的工作，于是直接返回。  

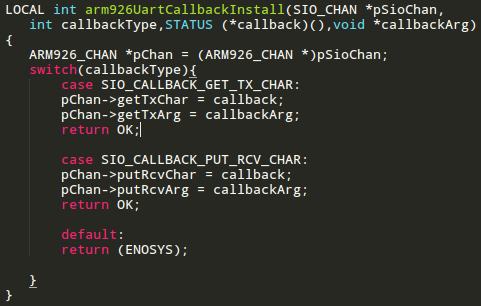

7.5.2 arm926UartCallbackInstall 函数实现

虽然 arm926UartCallbackInstall 函数实际上也是在初始化过程中被调用,但是其

完成的功能比较特殊，Arm926UartCallbackInstall由底层串口驱动提供,被 TTY 中间层调用,将 TTY 中间层实现的两个函数提供给底层串口驱动,即:内核读缓冲区写入函数和内核写缓冲区读取函数。当底层驱动接收到一个字节时,其调用读缓冲区写入函数将这个字节存储读缓冲区中,当用户启动一个串口读操作时,TTY 中间层将直接拷贝读缓冲区中的已有数据给用户,而不会对底层驱动做任何打扰。内核写缓冲区即被用户写入的缓冲区,其数据来源是用户层。对于底层串口驱动而言,其必须读取其中的数据,将它们通过串口一个字节一个字节的发送出去。每次发送完一个字节的数据,底层驱动将使用写缓冲区读取函数从内核写缓冲区中读取一个字节,操作串口硬件设备发送出去,直到写缓冲区清空,底层驱动才终止发送过程。当下一次用户又写入一批数据时,TTY 中间层将调用 arm926UartTxStartup 函数再次启动发送过程,底层驱动继而又不断的读取写缓冲区中的数据进行发送,直到写缓冲区再次清空,如此循环反复的进行着数据的发送行为。注意:前文中已经交代,TTY 中间层对于 arm926UartTxStartup 发送触发函数的调用,只在用户写入数据时,内核写缓冲区为空的情况下才进行,如果用户写入数据时,内核写缓冲区还存在数据,则表示底层驱动当前已经在不断地发送数据,此时就无需对arm926UartTxStartup 发送触发函数进行调用了。

由于内核读写缓冲区由 TTY 中间层进行维护,底层驱动无权直接对其进行操作,故需要 TTY中间层提供两个接口函数给底层驱动,从而完成底层串口驱动与 TTY 中间层之间的数据交互。

Arm926UartCallbackInstall函数实现的功能就是存储TTY中间层对其进行调用时传入的函数句柄（即函数的地址）。其代码如下：



实际上,arm926UartCallbackInstall 函数的实现代码可以被复制到 Vxworks 下所有串口驱动的 callbackInstall 函数实现中,只需对设备结构进行细微调整即可。该函数实现是非常标准的,就是根据调用类型保存函数句柄和对应参数。由于 TTY 中间层需要提供两个接口函数给底层驱动,故 arm926UartCallbackInstall 函数将被 TTY 中间层调用两次,每次传入一个接口函数的地址,至于究竟是哪种函数的句柄,则由第二个参数指定:

SIO\_CALLBACK\_GET\_TX\_CHAR表示此次传入的是内核写缓冲区读取函数；

SIO\_CALLBACK\_PUT\_RCV\_CHAR 则表示此次传入时内核读缓冲区写入函数。

注意在传递函数句柄的同时还有一个参数需要进行保存,该参数在进行对应函数的调用时必须作为第一个参数传入,这个参数被 TTY 中间层内部使用区分不同设备的缓冲区。在arm926UartInit 函数实现中,将 ARM926\_CHAN 结构中的 getTxChar,putRcvChar初始化为指向一个 dummy 函数,此处这两个函数指针才得到真正的初始化。此处保存的两个内核缓冲区操作函数将在串口驱动工作期间频繁的被调用用以与 TTY 中间层进行数据的交互。

本章开始处讨论中,我们说到对于 TTY 中间层提供的这两个函数完全可以以硬编码的方式

提供,即底层驱动直接调用指定名称的函数,无需进行此处繁琐的回调注册过程。但是这种硬编码的工作方式给后期的代码扩展带来了极大的不便,而且对内核代码本身的更新升级也带来了极大地限制。因为内核代码的升级必须保证在建立在原有版本上的驱动代码也可以工作,这就表示所有的后续操作系统版本必须定义早期版本中以硬编码方式提供给驱动的这两个函数。使用函数指针赋值的方式,就彻底解决了以上操作系统代码升级的问题,内核可以任意更换内部实现函数,而不会对底层驱动代码的运行造成任何影响。

7.5.3 arm926UartIoctl函数实现

该函数在初始化的过程中被调用设置初始的波特率以及使能设备的中断，但是不同于以上介绍的函数，这个函数并不是只在初始化的过程中被调用，在设备工作的任何时间点都可以被调用来对设备进行相关的控制。arm926UartIoctl 函数的实现具有最大的灵活性,其在实现基本功能之下可以提供任意的其他功能,也可以不提供其他任意功能。此处定义的基本功能有二:(1)设备中断使能,禁止功能;(2)波特率设置,获取功能。



对于SIO\_HW\_OPTS\_SET ,SIO\_HW\_OPTS\_GET表示的硬件设备选项返回ENOSYS,表示不支持硬件设备选项设置和获取。SIO\_OPEN选项对应用户串口打开操作(open函数调用),TTY中间层使用该选项调用底层驱动ioctl函数,此处底层驱动实现返回ENOSYS,注意返回ENOSYS并不表示串口设备不支持打开操作,实际上对于SIO\_OPEN选项,TTY中间层忽略底层驱动返回的任何参数,换句说话,用户打开串口设备操作总是成功的。SIO\_HUP选项对应用户关闭串口操作(close 函数调用),TTY 中间层同样也不对底层驱动返回值进行检查,也即用户关闭串口操作总是成功的。如果底层驱动需要实现SIO\_OPEN以及SIO\_HUP的功能,则必须进行针对性的响应。不过由于串口设备比较特殊,一般在初始化过后,就已经进入正常工作状态,而且基于串口设备的简单性和常用型,一般底层实现上,也不专门将串口设备打开和关闭作为子操作进行封装。所以用户层打开和关闭串口设备更多的意义是获取一个读写串口的文件描述符,而非需要底层驱动一定进行一个打开和关闭的具体响应。

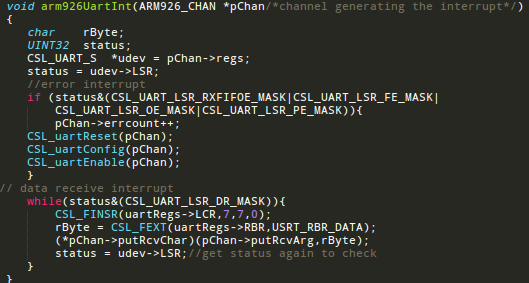
arm926UartIoctl 实现代码比较简单,对于其他平台串口驱动可以复制该函数实现,修改具体底层响应函数即可,这写具体响应的函数实现与平台相关,一般都是配置串口的相关寄存器,修改相关结构参数配合着管理。

7.5.4 arm926UartInt函数的实现

该函数是串口的中断响应函数，可以说是整个驱动的灵魂。驱动简单的说就是一套函数的集合，这套函数要被执行才会有意义。函数被调用的方式通常有两个途径：（1）用户发请求；（2）中断；一般而言用户请求触发的函数执行都是单次的,但是很多请求并不是单次执行一个函数就可以得到满足。对于串口设备(以及其他大多数设备)而言,中断通常有三种来源:(1)设备接收到新数据;(2)设备已发送完需发送的数据;(3)设备在数据收发过程中发生了错误。

对于设备接收到新数据发出的中断,其意义在于及时同时驱动将这个数据从设备内部的硬件缓冲区中(对于串口而言,其内部硬件缓冲区就是一个 8 比特的寄存器)读取出来,以便腾出空间接收下一个新数据,这是设备接收数据的主要和标准工作方式。当然数据的接收也可以使用轮询,但是轮询方式的使用对于设备的工作方式是有条件的,即数据的接收一定是对发送的数据的响应,简单的说,就是两个通信的设备之间具有严格的主从关系,而不是变动的主从关系。这种设备有 I2C 设备,SPI 设备,这些设备间的通信无论是数据发送和接收都是由一端主动发起,而另一端被动响应。此种方式下,在发送一个数据后,由于明确的知道在一定时间内,一定有数据传送过来,而且数据一般都具有确定的数目,故可以不断的轮询读取数据。对于像网口之类的设备,数据的接收并非一定是对发送数据的响应,故很难使用轮询工作方式接收数据。而串口设备则介于两者之间,需要根据具体使用情景确定。不过一般而言,为了提供数据收发效率,串口大多(而且必须)使用中断进行数据的接收,而对于数据的发送则也可以使用非中断方式,此时可能比中断方式具有更高的效率(见下文讨论)。前文中不断提到发送触发函数,针对我们的例子,对应 arm926UartTxStartup 函数,该函数被 TTY 中间层调用,用以触发一个发送过程。其确切的含义当用户写入一批数据,内核写缓冲区由空变为非空时,TTY 中间层将调用 arm926UartTxStartup 函数。该函数并不能一次性将所有的数据都发送出去,基于串口每次最多只能操作一个字节的数据的前提,用户所有数据的发送将分为多次进行。这个多次发送操作的实现可以有两种:中断方式和非中断方式。中断方式即将从内核写缓冲区读取一个字节,操作串口寄存器将这个字节发送出去,当串口硬件完成该字节的发送后,其发出一个中断,中断响应函数 arm926UartInt 监测到此次中断来源是一个数据发送完毕中断,这就表示可以进行下一个字节的发送了,此时中断响应函数将再从内核写缓冲区中读取下一个字节,将这个字节写入串口硬件数据寄存器后返回。串口硬件操作这个字节,当发送完毕后,其再次发出一个中断,如此循环往复,直到内核写缓冲区清空。中断方式下,串口硬件设备每发送完毕一个字节就发出一次中断,对系统的整体性能将造成不利的影响,因为每次中断响应过程都消耗不小的内核资源,而每个中断仅仅处理一个字节实在是“大材小用”。所以对于串口数据的发送,大多并不采用中断方式。当然,考虑到这一点,现在的串口设备很多内部集成一个 FIFO(一般为 16 字节),这就相当于从原来一个字节的硬件缓冲区变为现在的 16 字节硬件缓冲区(读写缓冲区是分开的,每个都是 16 字节),此种方式可以通过配置相关寄存器对中断触发条件进行控制,如每次发送 8个字节才发出一个中断(事实上,串口设备是等到写 FIFO 中累积到 8 个字节才真正启动发送动作)。这种方式部分的缓解了频繁中断给整个系统带来的不利影响,但是引起的一个问题是,如果一个用户单次只写入 4 个字节或者 5 个字节,总之是小于 8 个字节,那么这 4个或 5 个字节就会有不确定时间的延迟,必须等待用户后续字节的“推动”才能将这 4 个或5 个字节发送出去,对于这种不确定的延迟,某些情况下是不可容忍的。基于以上这些原因,虽然现在串口设备都内部集成有 FIFO,但是实际中很少使用。而对于串口数据的发送,则采用轮询发送方式,即当 TTY 中间层调用 arm926UartTxStartup 函数时,该函数进入一个while 循环,每次循环从内核写缓冲区中读取一个字节的数据,操作串口将其发送出去,并等待其发送动作的完成,此后再次从内核写缓冲区中读取一个字节,再次操作串口将其发送出去,直到 while 循环期间内核写缓冲器清空。而中断方式则只用于数据的接收和错误响应。对于数据的接收,FIFO 缓冲区的使用也有类似于发送的问题,此时 FIFO 中必须累积到一定数量的字节,才会触发中断,将这些数据传送给用户,如果外界只发送一个字节的命令,其后就等待回应,那么由于这边串口接收设备需要 FIFO 中累积到比如说 8 个字节,才给出中断,那么这一个字节将无限期等待,两边陷入死锁。所以串口设备接收一般也很少使用 FIFO,但是必须使用中断,虽然还是每次只能读取一个字节,但这是没有办法的事,如果使用轮询方式接收数据,基于串口数据接收时机的不确定性,其消耗的资源更是无法容忍。

综上所述，arm926UartInt函数实现将只对数据接收中断和收发错误中断进行响应。发送中断从硬件的角度被禁止使用，即不会有发送中断产生。实现如下：

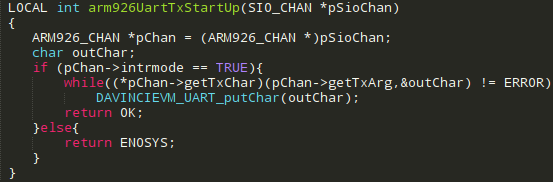


该函数先根据设备结构获取对应设备寄存器的基地址，然后读取中断状态寄存器，检查中断来源:(1)是一个错误中断，则复位串口，重新配置串口工作；(2)是一个数据接收中断，读取串口数据寄存器，调用内核读缓冲区写入函数，将这个字节传入TTY中间层。注意我们对于数据接收中断的响应代码使用的是 while 语句,而不是 if 语句,因为在读取数据之后,将数据传递给内核 TTY 层的过程中,串口可能又完成下一个字节的接收,此时由于处于中断处理函数中,中断被屏蔽,故中断没有给出,但是中断状态寄存器已经显示了更新的信息,所以我们将当前这个字节传递给 TTY 中间层之后,再次读取中断状态寄存器,查看串口是否在此期间已经完成对下一个字节的接收。上面没有发送中断的响应，原因已经介绍过了。

7.5.5 armUartTxStartup 函数实现

该函数的调用条件是：TTY中间层维护的内核写缓冲区由空变为非空。由前面的TTY初始化函数中可知，我们在调用ttyCreate函数的时候，传入了两个参数，指定了内核读写缓冲区的大小，其中写缓冲区用以缓存用户写入的数据，写缓冲区在管理上作为一个环形存储区，如果用户写入的频率较高，那么在缓冲区中已经有数据被发送之前，又有用户数据到来。由于内核的写缓冲区空间有限，如果用户写入数据的频率过高，造成写缓冲区变满，此时上层用户任务讲被设置为挂起状态，等待缓冲区变为非空。用户任务被挂起以及唤醒的操作由tty中间层来负责完成。

Arm926UartTxStartup函数将会实现一个while循环，采用轮询的工作方式将内核写缓冲区中当前数据清空。实现如下：



Arm926UartStartup函数的实现代码比较简单，但是完成了与TTY之间的协调工作。DAVINCIEVM\_UART\_putChar函数是底层发送数据实现函数，该函数每次接受一个字节，操作串口寄存器将数据发送出去，注意：DAVINCIEVM\_UART\_putChar函数每一次发送字节之前，都要检查发送就绪位，只有发送就绪位表示串口可以接收下一个字节时，才将要发送的这一个字节写入相关的（数据）寄存器。所以DAVINCIEVM\_UART\_putChar是一个阻塞函数，而arm926UartTxStartup函数也是一个阻塞函数。也就是说，如果用户写入的频率不高，那么每一次写入内核缓冲区都将为空。此时 arm926UartTxStartup 函数将被 TTY层调用,而用户任务将等待底层 arm926UartTxStartup 函数执行完毕,即用户层调用 write 函数将阻塞于该函数直到写入的数据真正的从串口设备中发送出去。这对于某些仅仅想通过串口输出信息的任务而言可能无法容忍,此时可以使用 logMsg 函数通过默认的串口通道打印,而不需要专门通过 open,write 等标准函数打开一个非标准通道进行信息的输出。如果非要这么做,而用户任务不能容忍等待,那么可以创建一个后台任务,模拟 tLogTask 对信息通过非标准串口通道进行输出。

虽然我们是用的是查询就绪状态位的方式进行数据的发送，而不是采用中断的机制，但是整个的设备仍然是工作在中断模式下，数据的接收和错误报告都是通过中断通知。

底层串口驱动组成中的最后一个部分就是具体操作串口寄存器的函数集合,这些函数平台相关,但是由于串口设备的常用性,基本上当前所有的串口设备都采用相同的寄存器映射方式,操作这些寄存器进行串口数据的收发代码很容易从网上下载或者直接使用开源的 Linux 串口设备的寄存器操作代码,只需进行简单修改即可使用到其他任何操作系统下,因为这部分硬件操作代码是与操作系统本身无关的,故本书不再列举串口设备寄存器操作函数的实现代码。串口驱动底层寄存器操作代码已经非常标准,所以当前串口驱动主要涉及的内容就是如何协调好与特定操作系统的接口层的交互。对于Vxworks 操作系统而言,就是处理与 TTY 中间层的交互工作。无论最终采用何种机制,其基本目的都是一致的:以一种有效地方式完成操作系统与底层驱动之间的数据交互。在这一点上, Vxworks 操作系统较为出色的完成了这个目的,而其中最为关键的内核组件就是 TTY驱动中间层,其具体负责与底层串口驱动之间的数据交互.

TTY中间层详解。

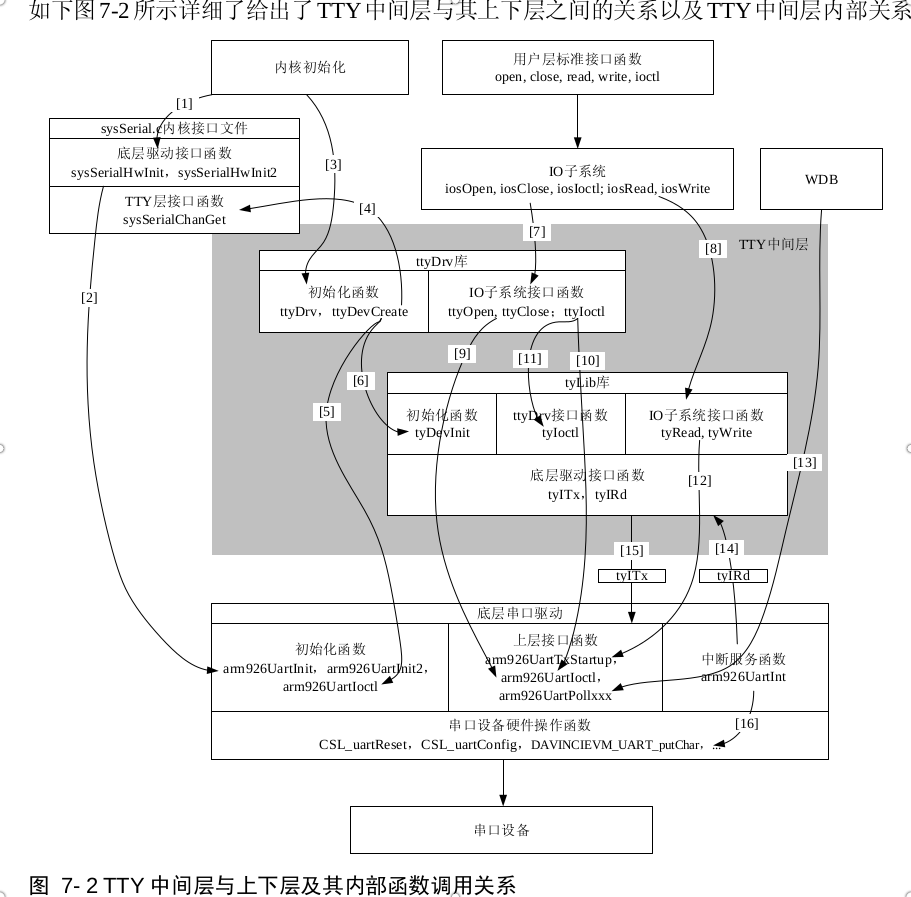
下面我们将深入 TTY 中间层内核组件,较为详细的介绍一些 TTY 中间层的内部组成以及阐述一个用户读写请求是如何通过 IO 子系统,TTY 中间层最终到达底层串口驱动的,我们着重关注一下在 TTY 中间层所做的处理。

在“TTY 中间层初始化”一节中我们介绍了两个重要函数: ttyDrv, ttyDevCreate,其中 ttyDrv完成 TTY 中间驱动层向 IO 子系统的驱动注册,即通过 iosDrvInstall 函数向 IO 子系统注册TTY 中间层驱动函数。自此所有的串口设备请求将通过 IO 子系统传递给 TTY 中间层驱动函数。 TTY 中间层注册的驱动函数为: ttyOpen, ttyClose, tyRead, tyWrite, ttyIoctl。 ttyDevCreate函数完成 TTY 中间层与底层串口驱动之间的相互注册,使能串口中断,使用传入的字符串创建一个串口设备节点并通过调用 iosDevAdd 函数将其添加到系统设备列表中。自此用户就可以通过诸如 open 之类的标准接口函数对串口设备进行打开,读写,关闭,控制等操作。所有的串口设备中,一般第一个注册到 TTY 中间层的串口设备被用作标准输入输出,此时对于该串口的输入输出不用之前调用 open 打开串口设备,Vxworks 操作系统启动过程中已经使用 open 调用打开了这个设备,返回的文件描述符被复制为标准输入输出,错误输出,即 0,1,2 三个系统文件描述符都指向了这个串口设备。Vxworks 操作系统内运行的所有任务都可以通过标准打印语句 printf,logMsg 通过这个串口设备输出信息或者读取信息。除了这个被用作标准输入输出的串口设备外,其他串口设备的操作必须遵循标准的设备操作流程,即打开设备,操作设备,关闭设备。虽然串口设备的打开操作只是上层软件上的一种操作,而不会对底层串口硬件施加行为,但是打开操作后将返回一个重要的参数:文件描述符。用户层此后所有的操作都必须基于这个文件描述符。

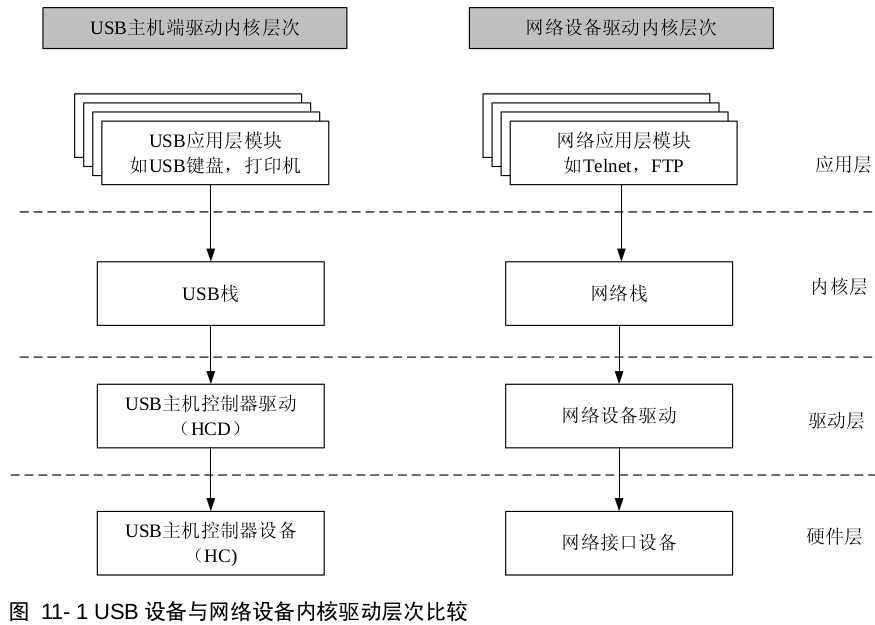
从 ttyDrv 注册的五个函数中我们看到 ttyOpen,ttyClose 就是 TTY 中间层对用户层打开和关闭串口设备的响应函数。而 ttyIoctl 函数则是用户层 ioctl 的 TTY 层响应函数。虽然 open,close,ioctl 标准用户层调用在 TTY 层实现为对应的三个响应函数,但是从前文中底层串口驱动代码的分析中我们看到底层驱动中并没有对应这三个函数的实现,实际上在 TTY 层就进行了转移,即 ttyOpen,ttyClose,ttyIoclt 函数最后都调用了底层串口驱动 ioctl 函数实现,对 应 上 文 的 例 子 就 是 arm926UartIoctl 函 数 。 ttyOpen 将 以 SIO\_OPEN 选 项 调 用arm926UartIoctl,而 ttyClose 将以 SIO\_HUP 选项调用 arm926UartIoctl,而 arm926UartIoclt则是 ttyIoctl 的底层标准实现。

TTY 中间层提供的用户层 read,write 读写响应函数是 tyRead,tyWrite。细心的读者可能发现这两个函数的命名方式不同于以上介绍的其他三个函数,即这两个函数不是命名为ttyRead,ttyWrite。这正是区别所在,实际上 tyRead,tyWrite 由 tyLib 库提供而 ttyOpen,ttyClose,ttyIoctl 包括 ttyDevCreate 函数以及 ttyDrv 函数则是由 ttyDrv 库提供。前文中我们分析出 TTY 中间层需要向底层驱动提供两个函数:内核写缓冲区读取函数,内核读缓冲区写入函数,实际上这两个函数也是由 tyLib 库提供,其中内核写缓冲区读取函数为 tyITx,内核读缓冲区写入函数为 tyIRd 函数,这两个函数都定义在 tyLib 库中, ttyDevCreate 函数只是“代为转交”。

由此可见,TTY 中间层具体有两个部分组成:ttyDrv 库和 tyLib 库。这两个内核组件共同组成 TTY 中间层。二者之间并无上下层关系,而是平行关系。从如上 ttyDrv 函数中调用iosDrvInstall 函数时提供的函数即可看出, ttyDrv 和 tyLib 都贡献了一部分,不过从实现上来看,tyLib 将完成 TTY 中间层基本所有的实质性工作,包括与底层串口驱动的具体的数据交互。除了 ttyDrv 库提供的 ttyIoctl 函数外,tyLib 也提供了一个 tyIoctl 函数,加上底层串口提供的 arm926UartIoctl 函数,这三个函数之间的调用关系是:用户层调用 ioctl 对设备进行配置和控制,这个请求通过 IO 子系统首先传递给 ttyIoctl 函数,ttyIoctl 函数本身并不完成任何实质性工作,其调用其他函数完成用户请求:首先调用底层驱动提供的 arm926UartIoctl函数,如果 arm926UartIoctl 返回值不是 ENOSYS,则直接返回;如果 arm926UartIoctl 返回ENOSYS,即表示底层驱动不支持对应请求,此时 ttyIoctl 将继续调用 tyIoctl 完成用户请求,所以 tyIoctl 作为最终的所有底层未处理请求的“终结地”。



1. USB设备驱动



对于没有接触过USB控制器底层驱动代码编写工作的人来说，对于很多涉及到USB的资料和规范上给出的很多概念哈市无法真正理解的。相比网络栈实现，USB栈的实现要简单的多，不过USB主机端控制器（HC：Host Controller）驱动的设计和实现要比网络设备驱动复杂的多，其复杂性主要体现在与内核USB栈的紧密耦合关系上，故要完成 USB HC驱动的编写,必须对 USB 栈实现本身有一个比较彻底的了解,Vxworks 下USB 栈内核实现代码都是以源码方式提供,这就为自实现 HC 驱动提供了可能。

通用串行总线(USB)仅仅是一个总线接口,定义了一套数据传输的方式和协议。除去 USB本身,我们可以将 USB 通信双方看作一对 FIFO,USB 规范定义了这一对 FIFO 之间数据传输的所需进行的各种配置和通信方式。从系统软件软件的角度,则只需从 FIFO 的角度进行理解即可。底层驱动(Host Controller 驱动)通过中断被通知 FIFO 中接收到另一端发送的数据或者被通知 FIFO 中数据已经被发送,需要向 FIFO 中填入数据准备下一次的发送,仅此而已,所有的发送和接收细节由称为 USB 控制器的硬件完成。

对于 USB 规范中定义的需要默认控制端点(EndPoint)响应的各种请求(如 set\_configuration,set\_interface 等等),则通过 USB 控制器后端的系统软件解析这些请求,并将合适的数据写入发送 FIFO,由 USB 控制器发送到主机完成请求的过程。所以 USB 规范中定义的各种描述符(device,configuration,interface,endpoint)是由系统软件维护的,而不是硬件自动响应的。USB 仅仅是一个接口,用于数据的传送,其并不对数据本身进行解析和处理。通常我们将这些请求以及对这些请求的响应与 USB 本身混为一谈,是因为很多 USB 资料上对此不加区分的进行介绍,造成了很多 USB 新手认为这些请求以及对这些请求的响应是由 USB本身完成的。而这是大错特错的,如此理解方式人为的复杂化了 USB 协议本身。

对于 USB 规范中定义的各种描述符一般存放在非易失介质上如 EEPROM,当主机被(Hub)通知有一个新的 USB 硬件连接到 USB 总线上时,其发送标准请求到这个新连接的 USB 硬件,读取各种描述符,从而获知该硬件具备的基本功能。这些请求通常为一个长度为 8 字节的内容,USB 硬件中内嵌的系统软件通过对这些内容进行解析,之后从非易失介质上读取相应的描述符,写入 USB 硬件的 FIFO 中,由 USB 硬件中的 USB 控制器将这些描述符通过USB 总线发送给主机。

如下是关于 USB 理解关键的几点:

1. USB 仅仅是数据传输的接口,其本身不对数据内容进行解析。数据的解析和应答由 USB控制器后面的系统软件(驱动)完成。系统软件从 USB 控制器 FIFO 中读取请求和数据进行解释和响应。

2. USB 外设各种功能的实现由主机上称为 USB Client 驱动和外设中内嵌的系统软件配合完成。

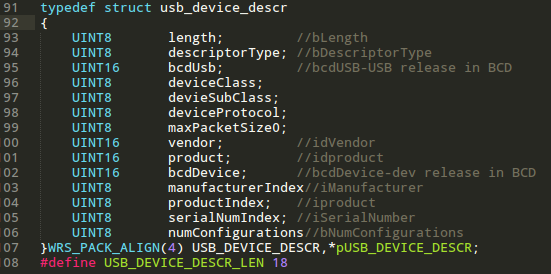
3. USB 规范中定义的各种请求以及相关数据存储在 USB 外设非易失介质上,主机对这些数据的请求由 USB 外设系统软件进行应答:从非易失介质中读取合适的数据并通过 USB控制器发送给主机。

4. 主机和 USB 外设各自具有一个 USB 控制器完成通过 USB 总线的数据交换。主机软件和USB 外设中内嵌系统软件通过 FIFO 和中断与各自的 USB 控制器进行数据交换。

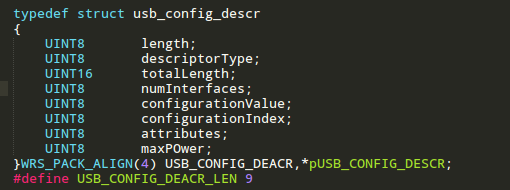
5. USB 外设一般结构为:USB接口,USB 控制器,微控制器(如常见的 8051),系统软件(或称固件,firmware),非易失存储介质。通常 USB 控制器和微控制器作为一个部分出现,名称上却仍然叫做 USB 控制器,这容易引起误解。对于 Mass Storage 类 USB 外设,则存在大容量 Nandflash 作为数据存储介质,其他 USB 外设一般为较少量的 EEPROM存储各种 USB 规范中要求的描述符参数。为了在微控制器(如 8051)中运行系统软件,USB 外设一般包含少量的系统 RAM。

在嵌入式系统下,USB 驱动一般而言是指 USB 控制器驱动,而非 Client 驱动。Client 驱动在 USBD 核心驱动之上,一般操作系统(如 Vxworks)对此的支持比较完善,无须用户作任何更改;而 USB 控制器在嵌入式系统下则五花八门,一般通用的操作系统很难正好包含相应的 HC (Host Controller)驱动,故必须由嵌入式操作系统移植人员完成。 HC 驱动在 USBD核心驱动之下,其驱动 USB 控制器(即 HC)通过 USB 总线进行数据的传输,主要是硬件寄存器的控制和控制器 FIFO 的读写以及中断的响应。

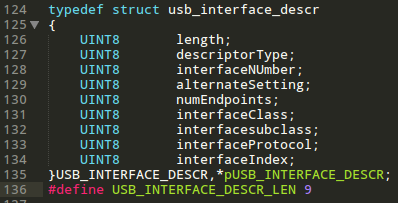
每一个USB外设都必须有且只有一个device descriptor。device descriptor描述整个USB外设，包含的信息是针对整个USB外设，如默认控制管道（EP0）的packet大小，基本结构如下：



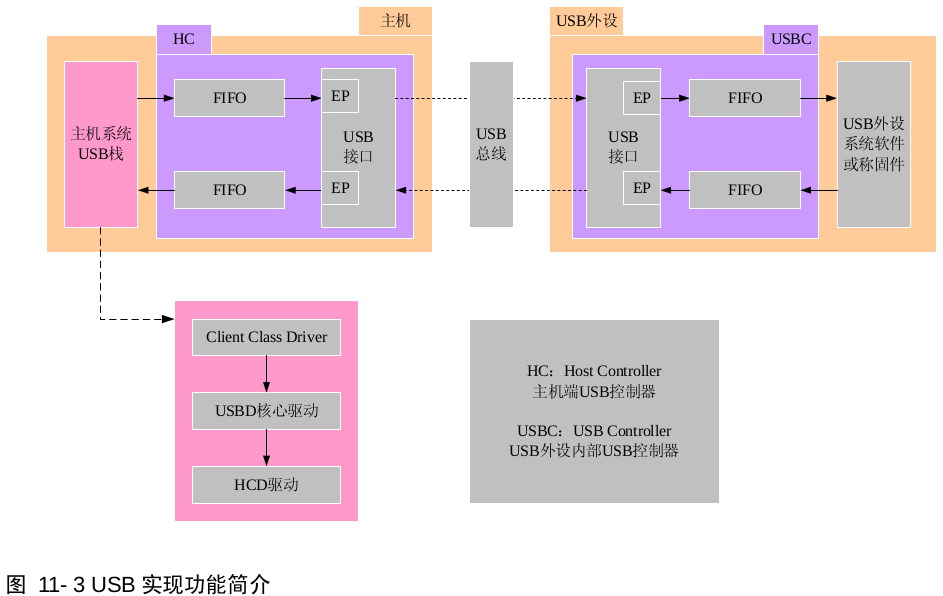
2.主机对配置描述符进行请求时，USB外设返回配置描述符内容外，还同时返回该配置下的所有interface描述符以及endpoint描述符。可以认为configuration描述符和interface，endpoint描述是相绑定的。一种配置体现了 USB 外设一种功能实现方式。如某个外设在一种配置下实现为鼠标,在另一种配置却实现为键盘。每种配置下可以有多个 interface,每种 interface 实现一种子功能。在一个 interface 下可以有多个 endpoint,表示该子功能下有多个数据传输节点。Endpoint 可以理解为网络中的套接字,它仅仅是主机与 USB 外设之间进行通信的一种手段。我们将主机与 USB 外设某个 endpoint 之间建立的数据传输通道称为管道(Pipe)。USB 外设遵循以 endpoint进行通信的方式,为每个 endpoint 都分配数据 FIFO 进行数据缓存,或者说使用 FIFO进行 USB 外设中内部USB 控制器与内嵌系统软件之间的数据交换。所以 endpoint 仅仅是一个叫法,它表示了一个主机与 USB 外设之间数据通信的节点,每个 endpoint 都有一个地址,USB 外设在接收请求时将检查这个 endpoint 地址,将数据转发到对应的endpoint 的 FIFO 中,供内嵌的系统软件作相应的处理。Configuration description定义如下：



1. 接口描述符(interface descriptor)必须作为某个 configuration 描述符的一部分,或者说接口仅仅是 USB 外设某种配置下实现的一个子功能。配置表示了 USB 外设较大方面实现的功能,而接口则表示这个大的功能下实现的某一个小的子功能,而包含在interface下的 endpoint 则是数据传输的节点。每个 configuration 可以有多个interface,即每个实现的大的功能下可以有多个子功能,最少是一个。配置表示的是大的方面,只有到接口这个层次才进行具体功能的描述,故虽然 configuration 描述符表示的是USB 外设实现的大的方面的功能,这只是为便于理解的一种说法,只有在 interface 描述符中方才对功能类进行描述,如 Mass Storage Class。接口描述符结构定义如下：



1. 端点(或称节点)(endpoint)是 USB 中最终的数据传输节点。主机的数据传输将以 USB外设中某个 endpoint 作为数据源或者目的地。Endpoint descriptor 相应的描述了一个endpoint 的地址,属性(如支持的最大传输包大小)等。Endpoint 隶属于某个interface,每个 interface 包括 0 个或多个 endpoint,表示在某种子功能下可以有 0个或多个数据传输的节点。Endpoint 类似于网络编程中使用的套接字,二者本质功能一样,表述方式不同而已。USB 外设为每个 endpoint 分配 FIFO 作为缓冲区,一般而言发送和接收各自具有缓冲区。USB 外设内嵌系统软件(或称为固件 firmware)通过这些 FIFO 与主机进行数据交互。所以本质上讲,endpoint 只是书面上的一种说法,或者说只是便于数据传输的一种机制。每个 endpoint 都具有一个地址用于数据传输中的寻址从而确定本次数据是发送到哪个 FIFO 的。当主机进行读取操作时,其发送一个数据请求 packet 到某个endpoint,USB 外设(内部的 USB 控制器)接收到请求后,将请求写入该 endpoint 对应的接收 FIFO,并发出一个中断,系统软件中断处理程序读取状态寄存器,获知此次中断源,从而到相应 endpoint 对应的 FIFO 中读取数据,并对数据进行解析,而后进行适当的响应。如请求为 get\_configuration 描述符,则系统软件从 USB 外设中非易失介质中读取 configuration,interface,endpoint 描述符信息,写入原先接收请求的 endpoint 对应的发送 FIFO 中,之后配置 USB 控制器相关寄存器,启动 USB 控制器将这些描述符信息发送给主机,从而完成一次主机请求。对于主机写入数据,原理相同。所以从根本上讲,endpoint 只是一种便于描述的书面上的一种用语,可以简单的将其看作是两个 FIFO:接收 FIFO 和发送 FIFO。使用 endpoint 的目的是为数据传输提供一种统一的机制,如寻址机制。USB 控制器并非如想象中的那么“能干”,它仅仅完成将数据写入相关的 FIFO或者从 FIFO 中将数据通过 USB 总线发送给对端的 USB 控制器。由此我们可以简单用下图描述 USB 实现的功能。



如下为endpoint描述符的定义：

