分 类 号_____ 学号_____M201672905_____

学校代码__10487__ 密级_____

華中科技大學

硕 士 学 位 论 文

基于 Vxworks 的调试通道的设计与实现

学位申请人：郑松

学 科 专 业：计算机应用技术

指 导 教 师：张杰 讲师

答 辩 日 期：2018 年 3 月 26 日

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master

A design and implementation of debug channel based on Vxworks.

Student : Song Zheng

Major : Computer Applications Technology

Supervisor : Instructor JieZhang

Huazhong University of Science & Technology

Wuhan 430074, P. R. China

March 26, 2018

独创性声明

本人声明所呈交的学位论文是我个人在导师的指导下进行的研究工作及取得的研究成果。尽我所知，除文中已标明引用的内容外，本论文不包含任何其他人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于　保密 □，在 ____ 年解密后适用本授权书。

　　不保密 □。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

通常在软件开发、移植、运行时都需要输出一些调试或日志信息，对于嵌入式设备而言通常使用串口来进行这些信息的传输，在 VxWorks 的集成开发环境 Tornado 中也有相关的调试工具，但是其比较适合用户软件的开发环境当中，不适用于实际的生产环境当中，同时由于目前大多数的设备都已不再将 RS-232 的串口作为必须的标准接口，而是大量的使用 USB 接口，因此在调试信息的传输过程中只能够使用 USB 接口，本文基于 VxWorks 开发出一个小巧易用的调试信息的传输通道，使用 CP2102 模块来进行 USB 口转串口的实际硬件部分，并编写了一个 VxWorks 下的 USB 口转串口驱动以及上层的应用层接口部分，本项目来源于中船重工的实际需求。

本文首先对课题的背景和需求进行了分析，给出了总体的设计方案，并给出介绍了本次的设计中需要使用的和了解的关键技术。然后从 VxWorks 下的 I/O 系统和驱动程序的关系入手，分析了 VxWorks 下 I/O 系统调用和驱动程序的实现过程，在此基础上实现了我们调试通道的必须的 USB 口转串口驱动，该驱动具有双向缓冲功能，同时具有即插即用、使用方便等特点。接着我们设计了调试通道的传输协议，并给出了两个上层应用的接口，一个 Log 日志接口，一个标准输出重定向接口。

最后我们通过对系统的各个部分进行测试和整体测试，验证了本次的调试通道设计基本完成，系统运行可靠，满足所需完成的要求。

关键词： 嵌入式系统，设备驱动开发，USB 口转串口，VxWorks，CP2102

Abstract

Generally, some debugging or log information needs to be output during software development, porting, and runtime. For embedded devices, serial ports are commonly used for the transmission of this information. There are also related debugging tools in Tornado, an integrated development environment of VxWorks. It is more suitable for the development environment of the user software, does not apply to the actual production environment, and at the same time because most of the current devices no longer use the RS-232 serial port as a required standard interface, but use a lot of USB interfaces. Only the USB interface can be used in the transmission of debugging information. This article has developed a small and easy-to-use debugging information transmission channel based on VxWorks. It uses the CP2102 module to perform the USB port to the actual hardware part of the serial port, and wrote a VxWorks USB port to serial port driver and the upper application layer interface part, this project comes from the actual needs of CSIC(China Shipbuilding Industry Corporation).

This paper first analyzes the background and requirements of the project, gives the overall design plan, and gives the key technologies that are used and understood in this design. Then start with the relationship between I/O system and driver under VxWorks, analyze the process of I/O system call and driver under VxWorks, and realize the necessary USB port to serial port driver of our debugging channel. The driver has two-way buffering function, and has plug-and-play, easy to use and other features. Then we designed the transmission protocol of the debug channel, and gave two upper application interfaces, a Log log interface, and a standard output redirection interface.

Finally, we test and test the various parts of the system to verify that the design of the debug channel is basically completed, the system is running reliably, and the required requirements are fulfilled.

Key words: embedded system, device driver development, USB port to serial port, VxWorks, CP2102

目 录

一 绪论

1.1 课题背景以及意义

当今,在嵌入式领域,嵌入式技术 (Embedded Technology) 已经成为了新的技术热点,随着嵌入式系统的不断发展和应用,针对不同的嵌入式软件的开发也越来越受到重视。在嵌入式软件的设计当中,由于嵌入式独有的特点,其调试、分析一直是一个费时费力的工作。一个好的调试器可以给嵌入式软件开发人员带来很大的帮助,使其达到事半功倍的效果,快速完成软件开发过程中的调试分析过程、软件运行过程中日志信息的定位等工作。

目前国内外已有几十种商业化嵌入式操作系统可以供选择,如 VxWorks、uc/OS-II、Windows Embedded CE、RTLinux、和“女蜗 Hopen”等。其中 VxWorks 以其良好的可靠性和优越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中[1]。而 Linux 操作系统则是完全开源的,在全世界拥有几十万的开源项目,目前主流的 Android 及嵌入式设备都采用 Linux 操作系统。

在我国 VxWorks 大量的应用于我国的军事、国防工业当中,通常在进行 VxWorks 应用程序的开发或者是将 Linux 下的应用程序移植到 VxWorks 中时都需要在程序中加入大量的调试信息,在程序的运行当中也需要输出一些日志信息,方便之后编程人员对程序运行过程中产生的问题进行具体的分析。VxWorks 自身带有一个集成的测试开发调试环境 Tronado,可以用它来完成程序的编辑、编译、调试、系统配置等工作。其带有的 CrossWind 调试器拥有一个驻留在主机端的命令行解释器 WindSh 和 GDB 命令行,但是由于中船重工实际使用中的限制,设备上并没有串口可供使用,而且设备并不具备进行现场调试的环境,他们希望能够在软件运行时直接将调试和日志信息输出保存之后进行一个事后分析的工作。

因此,本论文基于中船重工的实际需求制作了一个基于 VxWorks 的调试通道。

1.2 国内外概况

随着计算机技术的突飞猛进,实时系统无论是在技术上还是在应用领域当中都取得了辉煌的成就,尤其是在最近的二三十年当中,随着物联网的兴起,各种智能设备都需要安装嵌入式操作系统,导致嵌入式操作系统的发展愈加迅猛。而嵌入式实时操作系统属于嵌入式操作系统中定位更加精准的一类,其应用场景更加的专业化。实时嵌入式系统广泛应用在通信、航天、航空等关键型任务控制领域内。实时嵌入式系统是指以计算机技术为基础,以运用为目的的专用系统。系统对硬件和软件都有严格要求,硬件上对外观尺寸、内部可靠性及指令系统都有很高要求,软件在可靠性、实时性

等方面也具有严格要求。美国风河公司的 VxWorks 操作系统, 因具有抢占式调度、中断延迟小、系统内核可剪裁等特点, 在嵌入式应用领域内占据重要地位。在嵌入式系统应用中, 出于成本、尺寸、功能等方面考虑, 广泛应用定制硬件, 这样就要求用户自己开发硬件的驱动程序。驱动程序开发是系统开发中的重要部分, 驱动程序的性能、可靠性制约着应用系统的性能和可靠性。设备驱动本身与操作系统的相关性特别密切, 因此驱动程序开发不仅要求开发者对操作系统有深入的了解, 还要对硬件体系具有相当的了解, 所以难度较大。

目前国内在使用的 RTOS 有上百个, 这些操作系统面向不同的专业领域, 具有各自不同的特性。以下介绍几个具有代表性的操作系统:

- **uc/OS-II**
  是一个由 Micrium 公司提供的可移植、可固化、可裁剪、抢占式多任务实时内核, 在内核之上提供最基本的系统服务, 适用于多种微处理器、微控制器和数字处理芯片。该实时操作系统内核的特点是仅仅包含了任务调度、任务管理、时间管理、内存管理、任务间的通信和同步等基本功能, 没有提供输入输出管理、文件系统、网络等额外的服务。但是由于 uC/OS-II 良好的可扩展性和源码开放性, 这些功能完全可以由用户根据需要分别实现。
- **Windows Embedded CE**
  由 Microsoft 开发出来的一个嵌入式实时操作系统。其内核提供内存管理、抢先多任务和中断处理功能。内核上面是图形用户界面 GUI 和桌面应用程序。在 GUI 的内部运行着所有的应用程序, 其内核具有 32000 个处理器的并发处理能力, 每个处理器有 2GB 虚拟内存寻址空间, 同时还能够保持系统的实时响应[?]。但其缺点是很难实现产品的定制, 而且并不具备真正的实时性能, 没有足够的多任务支持能力。其主要的应用场景为互联网协议机顶盒、全球定位系统、无线投影仪以及各种工业自动化、消费电子、以及医疗设备等。
- **RTLinux**
  由美国墨西哥理工学院开发的嵌入式实时操作系统, 其特殊之处在于开发者并没有针对实时操作系统的特性而重写 Linux 内核, 而是将标准的 Linux 核心作为实时核心的一个进程, 同用户的实时进程一起进行调度。这样对 Linux 内核的改动非常小, 并充分利用了 Linux 下现有的丰富的软件资源。RTLinux 的优点在于: 与 Linux 一样, RTLinux 是开放源码的操作系统, 在网上较易获得所需的资料和技术支持, 使用者可以根据自己的需要进行修改。其主要应用领域包括航天飞机的空间数据采集、科学仪器监控和电影特技图像处理等。
- **Vxworks**
  由美国 Wind River System 公司推出的一个实时操作系统, 并提供了一套实时操作系统开发环境 Tornado, 提供了丰富的调试、仿真环境和工具。VxWorks 具有良好的持续发展能力、高性能的微内核以及友好的用户开发环境。它支持广泛

的网络通信协议、并能够根据用户的需求进行组合,其开放式的结构和工业标准的支持,使得开发者只需要做最少量的工作即可设计出有效的适合于不同的用户要求的系统。因为 VxWorks 良好的可靠性和卓越的实时性,其广泛的被运用于通信、军事、航天等高精尖和实时性要求极高的领域当中。

VxWorks 作为一款强实时性、高可靠性的操作系统,在我国广泛的运用在军工、航空航天、通信等部门[2]。VxWorks 的集成开发调试环境为 Tornado,使用该开发环境可以帮助编程人员轻松的完成程序的编辑、编译、调试、系统配置等工作[3][4]。Tornado 拥有一整套完整的面向嵌入式系统的开发和调试工具,包括 C 和 C++ 远程级调试器、目标和工具管理、系统目标跟踪、内存使用分析和自动配置,所有工具都能够很方便的同时运行,很容易增加扩展和交互式开发[5]。Tornado 的调试器包含有 GDB 命令行接口和 WindSh 工具,能够很好的进行应用程序的现场开发和调试。但是对于调试信息、日志信息的事后分析却没有提供解决办法且该工具要基于 RS-232 串口来使用,而现在大多数的设备都已不再配置 RS-232 串口。

对 USB 口转串口的设计通常可以采用两种方案,一种是以 CY7C68013 芯片为代表,自己从底层的固件开始,进行彻底而全面的系统开发,这种方案的成本和开发难度都很大,通常都不会使用这种方案。另外一个方案是采用类似于 CP2102 等专用的双向 USB 口转串口芯片来进行设计,这种方案简单实用,只需要对芯片的功能进行了解和应用即可,无需深入开发。因此我们在此会选择 CP2102 芯片来进行调试通道的设计。

1.3 论文的主要内容和组织结构

研究目标:在嵌入式实时操作系统 VxWorks 上实现一个能够满足程序的调试信息输出的通道,主要包括两个部分:一个满足特定要求的、实用的 USB 转串口驱动程序,一个上层的日志传输接口封装程序和标准输出重定向接口封装程序。

本文共分为六章,各个章节的具体安排如下:

第一章为绪论,主要介绍了本课题的研究背景和意义、国内外的发展状况以及本文的内容的安排。

第二张介绍了进行调试通道的开发所需要了解的系统知识,主要包括 VxWorks 系统及驱动开发的知识、USB 开发的相关知识,最后给出了一个调试通道的总体设计。

第三章介绍了我们使用 CP2102 模块开发的相关知识和 VxWorks 下的 USB 口转串口驱动的具体实现,包括特定需求下的单设备驱动和多设备支持的驱动

第四章主要介绍了应用层的接口封装部分,主要包括 Log 接口的设计,标准输出重定向接口的设计,以及 PC 客户端的协议解析部分。

第五章主要内容是系统的功能测试部分。

最后在结束语部分对整个的工作进行了总结,指出了本次的工作的不足之处,并对下一步的工作进行了展望。

二　调试通道总体设计与关键技术

2.1　总体设计

VxWorks 的集成开发调试环境为 Tornado, 它使用串口和网口结合的方式来对目标机进行控制和数据传输, 而目前对于大多数的设备而言都已经抛弃了串口, 很多用于军事上的嵌入式设备都是专用设备, 没有联网的需求, 并不会配备网口, 但是对于设备上产生的各种调试信息、日志信息都需要传输到我们的 windows PC 上来进行一个事后分析, 因此我们需要设计一个新的调试通道来进行这些信息的传输, 我们本次设计一个基于 USB 口转串口的底层驱动来实现该调试通道。本项目来源于中船重工在实际的生产环境当中的需求。调试通道的总体结构如图 2-1所示。

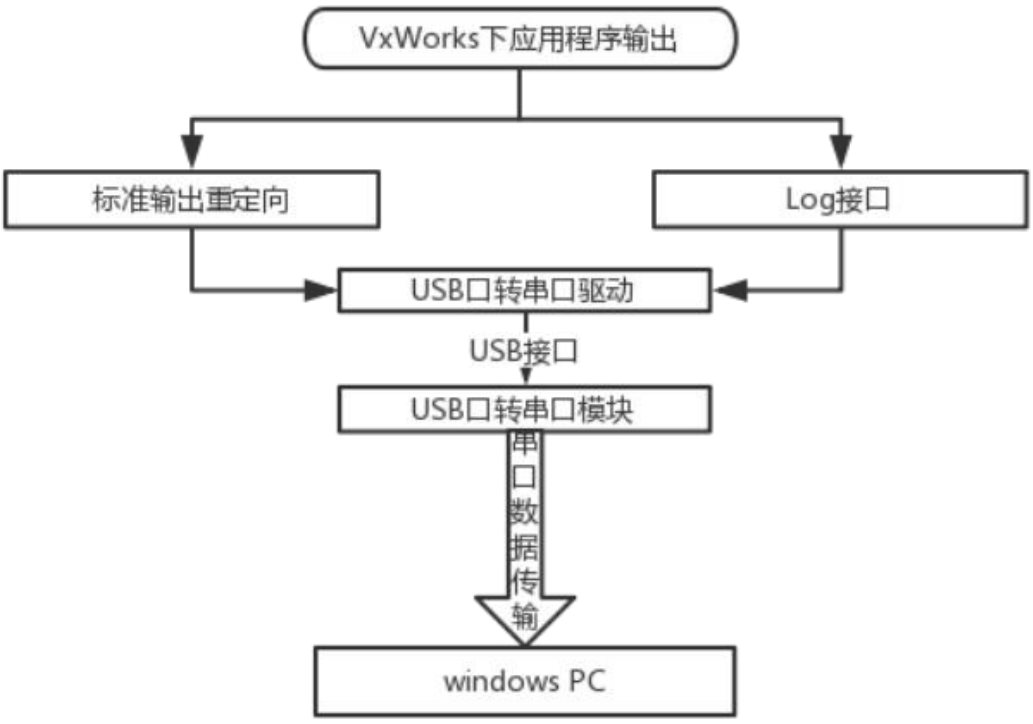


图 2-1　调试通道整体结构图

整个调试通道主要分为两个模块: 应用层的接口模块、USB 口转串口模块。提供给应用层的接口模块负责将系统应用层的输出通过我们的 USB 口转串口驱动程序传输到 windows PC 机, 输出的形式包括特定内容的格式化的输出和普通的重定向的输出, 格式化的输出我们会使用自定义的 Log 接口进行格式控制, 内容包含调试级别、调试信息所在的文件、调试信息所处的行号、输出该条调试信息的时间等。

USB 口转串口模块用于在 VxWorks 上实现一个 USB 口转串口驱动程序, 负责将

上层应用的信息传输到 windows PC, 主要包括驱动程序当中主要包含有驱动程序加载、卸载模块, 设备的打开、关闭、读、写、控制模块。同时在驱动程序中还需要一个数据的管理模块, 我们会使用循环缓冲区来管理数据。

2.2 关键技术

2.2.1 VxWorks 驱动开发

在 VxWorks 当中使用 I/O 子系统来管理设备驱动, I/O 子系统在整个 VxWorks 当中起着承上启下的作用, 各种类型的设备都必须要向 I/O 子系统进行注册才能够被内核访问。I/O 子系统维护着设备驱动中非常重的三个数据结构: 系统设备表、系统驱动表、系统文件描述符表, 设备驱动程序在初始化的时候会完成硬件设备寄存器的配置, 同时也会向 I/O 子系统注册驱动和设备。在 VxWorks 中设备驱动的访问过程如下:

1. 使用 open() 函数打开一个名为 devName 的设备, 此时 VxWorks 的 I/O 系统会在系统的设备表中寻找这个名为 devName 的设备项, 并找到相应的驱动号;
2. I/O 系统在文件描述符当中保留一个文件描述符, 然后在系统的设备驱动表中找到该设备的 xxxOpen() 函数, 调用这个函数, 并返回设备描述符的指针;
3. I/O 系统将设备描述符的指针存储在文件描述符列表的 Device ID 中, 同时将对应的设备驱动号存储在文件描述符的 Driver Num 项。最后 I/O 系统返回该描述符的索引 (即 fd);
4. 应用程序当中使用这个 fd 来调用 read()、write() 函数。系统会根据 fd 自动找到相应的设备驱动号, 进而找到相应的驱动例程。

2.2.1.1 VxWorks I/O 系统

通常操作系统为了平台的无关性都会为应用程序提供一套标准的接口, VxWorks 也不例外, 它为应用层的提供了接口函数 creat()、open()、unlink()、remove()、close()、rename()、read()、write()、ioctl()、lseek()、readv()、writev() 等, 我们将其称作为标准 I/O 库[6][7]。这样就可以通过调整底层驱动或者是接近驱动那部分的操作系统中间层来实现应用程序的通用性, 提高应用层的开发效率, 避免重复编码。通用操作系统 (如 Mac OS、Linux、Windows) 中, 通常会把这套接口从操作系统中独立出来, 以标准库的形式存在, 但是在 VxWorks 中它们是由系统的内核实现的, 都位于 ioLib.c 文件下。VxWorks 与通用操作系统有很大的一个不同点是: VxWorks 不区分用户态和内核态, 用户层可以直接对内核函数进行调用, 而无需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需通过外围库的方式, 而是直接以内核文件的形式提供[7]。对于一般的设备而言, remove() 接口是不需要实现的, 表 2.1 是七个驱动程序接口函数的简单描述。

接口名称	函数原型	描述
open	open(filename,flags,mode)	打开一个新的或已存在的文件
create	create(filename,flags)	创建并打开一个新的文件
read	read(fd,& buf,nBytes)	从文件中读取
write	write(fd,& buf,nBytes)	向文件中写入
ioctl	ioctl(fd,command,arg)	其他控制命令
close	close(fd)	关闭文件
remove	remove(filename)	移除文件

表 2.1　IO 接口函数表

2.2.1.2　系统设备表

系统设备表是 VxWorks 中为了管理系统上的所有设备而使用的一个链表。系统设备表在系统中的连接方式如图 2-2所示。wind 内核规定每一个设备都必须使用一个 DEV_ HDR 的结构来表示该设备，其定义如下：

```
/*h/iosLib.h*/
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;
```

VxWorks 系统提供了一个设备的注册函数 iosDevAdd(DEV_ HDR *pDevHdr, char *name, int drvnum) 来将设备添加到系统设备表当中，系统设备表在每次添加设备时就会增加一个节点，删除设备时就会减少一个节点，它会为 open()、close()、remove() 这三个函数提供文件与设备的连接，当应用程序执行这三个函数中的一个时，IO 系统会通过文件名对设备链表中的项进行匹配，匹配的方式是最佳匹配，匹配成功之后就使用这个设备驱动进行其他的文件操作。

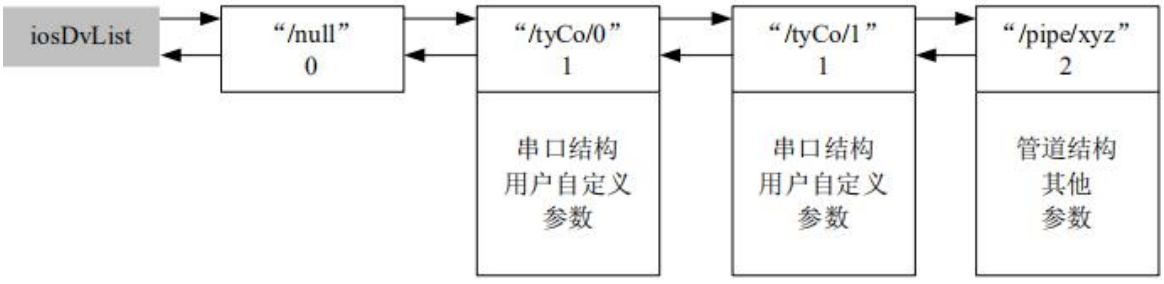


图 2-2　VxWorks 系统设备示意图

2.2.1.3 系统驱动表

系统驱动表用于管理当前注册到 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 IO 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 IO 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 IO 子系统[7][8]。

系统驱动表底层的实现是一个数组，其中的每一个表项都是一个 DRV_ ENTRY 类型的结构，该结构定义在 h/private/iosLibP.h 文件当中，其定义如下：

```
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;
```

DEV_ ENTRY 结构体实际上就是一个函数指针结构，结构中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接口一一对应[9][10]。成员 de_ inuse 用以表示一个表项是否空闲。这个结构体中的函数指针实际指向的内容由驱动调用 iosDrvInstall(FUNCPTR pCreate,FUNCPTR pDelete,FUNCPTR pOpen,FUNCPTR pClose,FUNCPTR pRead,FUNCPTR pWrite,FUNCPTR pIoctl) 来提供。

2.2.1.4 系统文件描述符表

系统描述符表用于管理当前系统打开的所有文件描述符。其底层实现也是一个数组，文件描述符表的表项索引被用作文件描述符的 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附表中只占用一个表项，即都使用同一个表项[11][7]。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[3][12][7]。

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，如下所示。

```
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;
```

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。

2.2.2 VxWorks 中的通信机制

wind 内核提供了三种任务间通信机制 (不包括信号机制): 信号量，消息队列，管道。这三种机制都是使用的共享物理内存机制，这块共享的内存是由内核进行管理的，任务必须通过内核提供的接口函数进行访问，这种保护和管理机制使任务间通信安全有序的进行。

1. **信号量**

   信号量的主要用途是互斥和同步。基于各种资源的不同使用方式，VxWorks 信号量机制具体的提供了三种信号量:通用信号量;互斥信号量;资源计数信号量。通用信号量既可用于同步也可用于资源计数，此时资源数通常为 1(当资源数为 1 时，也可以称之为互斥)。互斥信号量针对在使用过程中一些具体问题(如优先级反转)做了优化，更好的服务于任务间互斥需求;资源计数信号量用于资源数较多，同时可供多个任务使用的场合。

   VxWorks 中信号量是一种指向 semaphore 结构的指针，其定义如下所示:

```
typedef struct semaphore
{
  OBJ_CORE objCore;/*对象管理*/
  UINT8 semType; /*信号量类型*/
  UINT8 options; /*信号量选择*/
  UINT16 recurse; /*信号量重复获取计数器*/
  Q_HEAD qHead; /*阻塞的任务队列头*/
  union{
        UNIT count;/*当前状态*/
        struct windTcb *owner;
  }state;
  EVENTS_RSRC events;/*VxWorks事件*/
}SEMAPHORE;
```

2. **消息队列**

   消息队列内核实现上实际是一个结构数组，数组大小和数组中元素的容量在创建消息队列时被确定。消息队列是 Vxworks 内核提供的任务间传递较多信息的一种机制，不过这种机制存在的很大的局限性，即每个消息的最大长度是固定

的。Vxworks 内核提供的消息机制在创建消息队列时就必须指定单个消息的最大长度以及消息的数量,在消息队列成功创建后,这些参数都是固定不变的。

3. **管道**

   管道相比消息队列提供了一种更为流畅的任务间信息传递机制。消息队列必须将信息分批打包,而且对于每个消息的大小存在限制,而管道可以像文件那样进行读写,是一种流式消息机制。管道通常分为两种:命名管道和非命名管道。任务间通信使用的一般都是命名管道。非命名管道使用在线程意义上,如父子进程,进程关系密切,且某些变量存在继承关系,如文件描述符,一般无法使用在两个执行路线完全不同的任务之间。

任务间特殊的通信机制–信号:信号用于通知一个任务某个事件的发生。Vxworks 中的信号处理机制有些特别之处,当一个任务接收到一个信号时,在这个任务下一次被调度运行之时进行信号的处理(即调用相关信号处理函数)。事实上,由于 Vxworks 在退出内核函数时都会进行任务调度,故一个任务,无论是否是当前执行任务,都将在被调度运行时执行信号的处理。通用操作系统上对于某些信号将不允许用户修改其默认处理函数,如 SIGKILL,SIGSTOP,然而 VxWorks 操作系统中可以对任何信号的处理函数都可以进行更换。

2.2.3 USB 技术

USB(Universial Serial Bus,通用串行总线) 是这十几年来应用在 PC 领域的最新型的接口技术,出现的契机是为了为了解决日益增加的 PC 外设与有限的主板插槽之间的矛盾,其实现是由一些 PC 大厂商 (Microsoft、Intel 等) 定制出来的,自从 1995 年在 Comdex 上展出以来至今已广泛地被各个 PC 厂家所支持。目前已经在各类外部设备中都广泛的采用 USB 接口。USB 接口标准目前有三种:USB1.1,USB2.0 和 USB3.0。USB 接口应用如此的广泛是由其独特和实用的特性决定的。

USB 规范规定了 USB 传输的四种方式,每种方式有各自的用途[13]:

- **控制传输**:控制传输是每一个设备必须要具有的传输方式,也是四种传输方式中过程最复杂的部分,他使得主机能够从设备列出的范围中读取和选择配置和其他设置,也能够发送自定义请求来为任何目的而发送和接收数据,在配置和列举的过程中起到重要的作用。
- **批量传输**:批量传输是为了处理传输速率不是很关键的情况,一般用于打印机和扫描仪。
- **中断传输**:中断传输是为了那些要快速实现主机和设备的交互而准备的,比如适用于鼠标和键盘。
- **等时传输**:等时传输用于必须要按照一个常数传输数据的情况,比如一个需要被实时播放的视频/音频数据流。

所有的传输都是由事务组成,事务又由包组成,而包包含一个包识别器 (PID)、CRC 和

其他的信息[14]。

USB 的体系结构如图 2-3所示

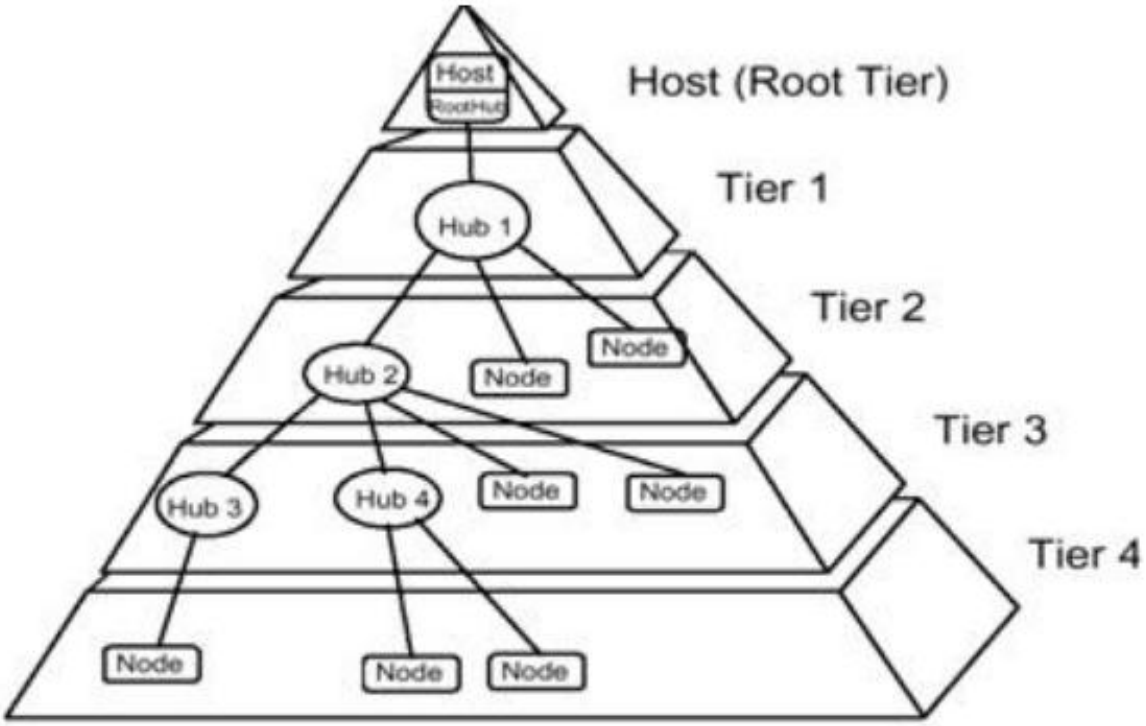


图 2-3　USB 总线拓扑结构

USB 的物理层拓扑为星型结构, 包括三个部分: USB 主机 (Host)、USB 集线器 (Hub)、USB 设备 (Device)。

1. **USB 主机**

   USB 的体系结构只允许系统中有一个主机, 主计算机系统的 USB 接口称之为 USB 主控制器。主控制器可以是硬件、固件或软件的联合体, 其控制着总线上所有 USB 设备和所有集线器的数据通信过程。所有的数据传输都是由 USB 的主机端发起的, 而且如果 USB 主机嵌入在一个计算机系统中, 在数据的传输过程中也不需要计算机的 CPU 参与传输工作。USB 主机通常具有以下的功能:

   - 检测 USB 设备的插拔动作 (通过根集线器来实现)
   - 管理 USB 主机和 USB 设备之间的控制流
   - 管理 USB 主机和 USB 设备之间的数据流
   - 收集 USB 主机的状态和 USB 设备的动作信息

2. **USB 集线器**

   根集线器是集成在主机系统当中的一个特殊集线器, 他可以提供一个或者跟多的接入口。在即插即用的 USB 体系结构当中, 集线器是一种很重要的设备。其极大的简化了 USB 的复杂性, 而且以很低的价格和易用性提供了设备的健壮

性,集线器的最大的连接能力是 127。

3. **USB 设备**

USB 设备是 USB 总线系统的重要组成部分,它们以从属的方式与 USB 主机进行通信,并受 USB 主机的控制。USB 主机端提供的协议软件通过和 USB 设备通信获得设备的信息,并给设备提供驱动程序,相比 USB 主机而言,USB 设备只能够被动的应答,按照 USB 主机的要求接受或者发送数据。USB 设备通过以下的属性来完成主机的要求:

- **描述符**

  USB 协议为 USB 设备定义了一套描述设备功能和属性的固定结构的描述符,通过这些描述符向 USB 主机汇报设备的各种属性,主机通过对这些描述符的访问对设备进行识别、配置并为其提供相应的客户端驱动程序。典型的描述符一般由 USB 标准描述符和 USB 类描述符,或者由 USB 标准描述符和 USB 厂商特定描述符组成。运行于 USB 协议栈上层的客户端驱动程序通过这些信息正确的访问设备并与其进行通信,以实现即插即用的目的。

- **类**

  USB 协议支持许多的外围设备,为了正确的驱动这些设备,USB 主机端要为这些设备提供符合 USB 协议的驱动程序,称为客户端驱动。同时为了避免客户端程序过多,协议通过归纳将设备划分为不同的设备类,把功能相近的设备归为一类,主机端只需要提供类驱动程序便可以驱动大多数的 USB 设备。

- **功能 (Function)/接口 (Interface)**

  在 USB 协议中,Function 被定义为具有某种能力的设备,即相当于传统的单一功能设备。随着 USB 设备应用的发展,物理上几种不同的 Function 可以组成一台设备,只要设备的接口具有某种独立的能力,就称为一个 Function。Function 是从功能角度来说的,从设备的角度来说,它又被称为 Interface。

- **端点 (Endpoint)**

  端点层的各个子模块是 USB 设备与 USB 主机逻辑上的数据传输的最基本单元,即最基本的通信点,因而端点层的每一个逻辑模块被称为端点 (Endpoint)。每一个端点都关联一个相应的端点号和数据传输方向 (IN/OUT)。具有相同端点号和不同传输方向的通信点表示不同的端点,端点 0 被 USB 规范保留用作设备枚举和配置过程中的数据传输端点,与端点 0 对应的管道是默认管道,设备的所有端点共享端点 0。在一个 USB 系统当中,USB 主机对特定设备功能的访问是通过使用不同的端点号,否则,USB 主机将不能对相应的设备进行正确的访问。由于在系统运行时,不同

的设备配置有着互斥性,因而在不同的配置描述中,端点号是可以重复的。

- **管道**

  设备端点与 USB 主机所形成的具有特定数据传输特性 (如数据传输格式、传输带宽、传输方向等) 的数据通道, 被称为管道。管道的物理介质就是 USB 系统中的数据线。端点层的 USB 设备与 USB 主机之间的数据传输都是基于管道进行的。

- **设备地址**

  USB 主机的客户端驱动程序通过描述符来区分不同的设备, 而 USB 主机控制器通过设备地址来区分设备。设备地址共有 7 位, 表示理论上系统可以同时连接 127 个 USB 设备, 但是在实际中, 由于 USB 总线带宽的限制, 这么多设备不可能同时的工作。USB 主机负责为 USB 系统中的设备分配不同的设备地址, 用来表示哪个设备的同时还要指名设备的端点号, 表示使用哪个管道。

一个主机和设备的连接需要一些列的层次和实体之间的交互, USB 接口层在主机和设备间提供物理、信号包的关联, USB 设备层表示 USB 系统程序实现对一个设备进行的总的 USB 操作, 功能层适当匹配的客户服务程序层提供附加的功能给主机。设备和功能层当中各自有逻辑通信, 但是实际的 USB 数据传输是通过 USB 总线接口层来实现的[?][15]。

2.3 本章小结

本章重点介绍了本次的 VxWorks 调试通道的整体架构, 并介绍了介绍了各个部分的设计方案, 最后介绍了在本次的设计当中所需要使用关键技术和所需了解的重要知识, 主要包括 VxWorks 下的驱动开发必须的结构、驱动中所需使用的 VxWorks 的通信机制、缓冲区技术、USB 技术。下面将要讨论 VxWorks 下的调试通道的详细的设计细节和具体的实际机制。

三　驱动程序的设计和实现

3.1　USB 口转串口驱动的设计

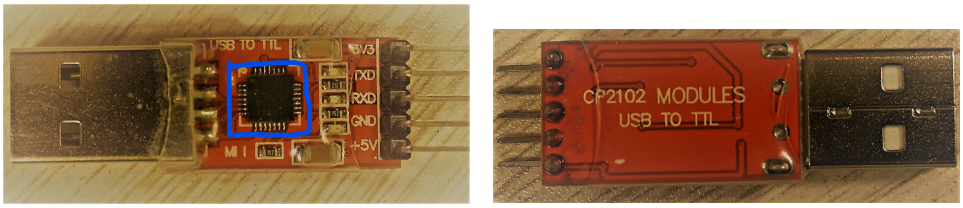
USB 口转串口驱动程序的编写与操作系统的关系密不可分。设备驱动程序在操作系统中如何存在、如何与操作系统的其它部分相联系、如何与操作系统的其他部分相联系、如何为用户提供服务都是操作系统的设计人员在设计操作系统时制定的，系统已经为驱动程序制定好了一个框架，无论驱动程序的开发人员以何种方式控制设备，他们所开发的驱动程序都是以预先设计好的方式存在、与操作系统其他部分相联系和为用户提供服务的[2]。我们此次驱动程序需要实现的框架部分主要包括 cp210xDrvInit()、cp210xDevOpen()、cp210xDevClose()、cp210xDevIoctl()、cp210xDevWrite()、cp210xDevRead()、cp210xDrvUnInit() 等函数，如表 3.1所示，以及用于处理数据的缓冲区、用于进行同步和互斥操作的信号量。

模块	作用
cp210xDrvInit()	这个模块用来初始化驱动程序，主要是与设备无关的一些全局变量并向系统注册该驱动
cp210xDevOpen()	这个模块用来转接 I/O 子系统分发过来的 open() 操作，实现设备的打开，返回文件描述符
cp210xDevClose()	这个模块用来转接 I/O 子系统分发过来的 close() 操作，实现设备的关闭，对设备占用资源进行清理
cp210xDevIoctl()	这个模块用来转接 I/O 子系统分发过来的 ioctl() 操作，实现对设备的一些特定的控制操作
cp210xDevWrite()	这个模块用来转接 I/O 子系统分发过来的 write() 操作，实现对设备进行数据的写入
cp210xDevRead()	这个模块用来转接 I/O 子系统分发过来的 read() 操作，用于从设备读取数据。
cp210xDrvUnInit()	这个模块用来卸载驱动程序，将驱动从系统驱动表中删除，并清理该驱动程序所占用的全部资源

表 3.1　驱动程序的关键模块

另外一个方面，USB 口转串口的驱动程序需要硬件来作为支撑,PC 机上本身并没有 USB/RS-232 的转换器，对于 USB/RS-232 转换器的设计通常有两类实现方式: 一类是采用全面系统的设计，使用包含有 USB 单元的微处理器，要求它具有

内置的通用异步收发器 (UART) 在 USB 和 RS-232 之间进行信号转换, 这样的控制器包括 PCI16C745、68HC705JB4 和 C541U 系列等[16], 也可以采用 USB 接口芯片如 PDIUSBD12、USBN9604 等与微控制器组合工作; 第二类方法是采用专用的 USB/RS-232 双向转换芯片, 如 CP2102、FT232BM 等, 我们在此处的设计即使用了 CP2102 芯片作为 USB/RS-232 转换器, 这样设计的好处是不需要编写转换器芯片的固件, 节约开发时间, 由于这个技术已经很成熟, 大多的 USB 口转串口的解决方案都会采用这种已经设计好的集成芯片来作为转换器。我们本次使用的设备如图 3-1所示。



(a) CP2102 模块正面　　(b) CP2102 模块反面

图 3-1　CP2102 模块正反面

3.1.1　CP2102 开发

CP2102 是 SILICON LABORATORIES 推出的 USB 与 RS232 接口转换芯片, 是一种专门用来进行 USB 转 UART 的高度集成的桥接器, 和其他同类型的芯片相比具有功耗更低、体积更小、集成度更高 (仅需少量外部元件)、价格更低等优点。因此我们此次选择这个芯片作为调通道的设计当中使用的芯片。CP2102 提供一个使用最小化的元件和 PCB 空间实现 RS232 转 USB 的简便的解决方案。CP2102 芯片包含有一个 USB2.0 全速功能控制器, EEPROM, USB 收发器, 振荡器和带有全部的调制解调器控制信号的异步串行数据总线 (UART), CP2102 将全部的部件集成在一个 5mm*5mm MLP-28 封装的 IC 当中[17], CP2102 的电路框图如图 3-2所示。

使用 CP2102 进行串口扩展的时候所需要的外部器件是非常少的, 仅仅需要 2-3 个去耦电容即可, 在 SILICON 给出的文档当中已经帮我们给出了一个最简单的连接电路图, 如图 3-2所示。电路使用 CP2102UART 总线上的 TXD/RXD 两个引脚, 其余的引脚都悬空。CP2102 可以完成 USB/RS-232 双向转换 (需要外接一个 TTL 电平到 RS-232 电平的芯片), 一方面可以从主机接收 USB 数据并将其转换为 RS232 信息格式流发送给外设, 另外一方面可以从 RS232 外设接收数据转换为 USB 数据格式传送回主机。使用时我们只需要将数据通过 USB 的数据包发送给 CP2102 芯片即可, 芯片会自动进行解析和控制。

1. CP2102 的 USB 功能控制器和收发器: CP2102 的 USB 功能控制器是一个符合 USB2.0 协议的全速器件, 这个器件负责管理 USB 和 UART 之间的所有数据传输以及由 USB 主控制器发出的命令请求和用于控制 UART 功能的命令。

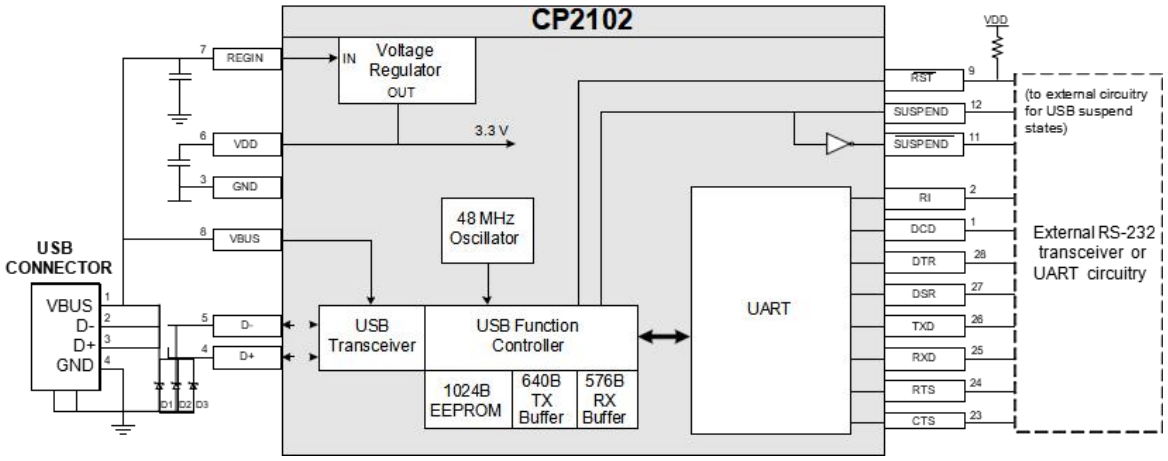


图 3-2　cp2102 电路框图

2. 异步串行数据总线 (UART) 接口: CP2102 的 UART 接口包括 TXD(发送) 和 RXD(接收) 数据信号以及 RTS, CTS, DSR, DTR, DCD 和 RI 控制信号。ART 支持 RTS/CTS, DSR/DTR 和 X-on/X-Off 握手。且支持编程使 UART 支持各种数据格式和波特率。ART 的数据格式和波特率的编程是在 PC 的 COM 口配置期间进行的。可以使用的数据格式和波特率见表 3.2。
3. 内部 EEPROM:CP2102 内部集成了一个 EEPROM 用于存储设备原始制造商定义的 USB 供应商 ID、产品 ID、产品说明、电源参数、器件版本号和器件序列号等信息[17]。USB 配置数据的定义是可选的, 如果 EEPROM 没有被 OEM 的数据所填充的话, 则设备会自动的使用一组默认的数据如表 3.3所示。

数据位	5,6,7,8
停止位	1,1.5, 2
校验位	无校验, 偶校验, 奇校验, 标志校验, 间隔校验
波特率	600,1200,2400,4800,7200,9600,14400,16000,19200,28800, 38400,51200,56000,57600,64000,76800,115200,128000,158600, 230400,250000,256000, 4608000,576000,921600

表 3.2　CP2102 可配置参数

CP2102 当中的协议控制单元会通过接受 USB 接口的命令, 对 UART 接口进行配置 (如配置通信的波特率、数据位、校验位、起始/停止位、流控信号等)。CP2102 当中的接收和发送缓冲区用来临时保存双方在数据传输过程中的数据。以从计算机到外设的数据传输为例。当 USB 转串口设备连接到 PC 的 USB 总线上时, PC 在检测到设备连接之后会对设备进行初始化并启动相关的客户端驱动程序; 之后会由驱动程序给设备发送配置命令, 设置设备的数据传输特性; 最后, 在数据传输的时候, 计算机上的驱

Name	Value
Vendor ID	10C4h
Product ID	EA60h
Power Descriptor(attributes)	80h
Power Descriptor(Max Power)	32h
Release Number	0100h
Serial Number	0001(63 characters maximum)
Product Description String	”CP2102 USB to UART Bridge Controller”(126 characters maximum)”

表 3.3　CP2102 默认配置表

动程序会将数据包传输给 USB 接口 (通常使用批量传输的方式), 设备从 USB 接口提取出数据并保存在数据缓冲区中, UART 接口再从数据缓冲区中将数据取走并发送出去, 从外设传输数据到计算机的方式则相反。

3.1.2　VxWorks 上的 USB 开发

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。VxWorks 的内核驱动层次结构如图 3-3所示。

VxWorks 的 USB 主机驱动程序堆栈满足 USB 协议规定的要求, 提供了一整套服务来操作 USB 以及一些预置 USB 类驱动程序, 以处理特定类型的 USB 设备。在 Wind River 的 VxWorks 中 USB 驱动程序堆栈的开发符合的是通用串行总线规范 2.0 版, USB 系统是一种主从结构, 系统的所有动作都是由 USB 主机发起, 并协调不同的设备动作, 设备端软件在系统中只需要对主机的命令做出响应即可, USB 的主机端由于在系统中的地位比较特殊, 因而其软件结构比较复杂, USB 协议在主机端是分层实现的, 其通信的逻辑结构和 PC 端的软硬件结构如**??**所示。USB 协议由上至下可以分为三层: 客户端驱动程序 (Client Driver)、USB 驱动 (USBD)、主机控制器驱动 (HCD), 每一层完成不同的功能。

客户端驱动程序完成对不同的设备类的设备的功能驱动。本次设计中所要完成的 USB 口转串口的驱动要完成的就是客户端驱动。为了和设备进行正常的通信, 他通过 USB 的 I/O 请求包 (I/O request, IRP) 向 USBD 层发出数据接收或者发送的请求。此外, USB 的传输机制对于客户端的驱动程序而言是完全透明的, 客户端驱动程序所

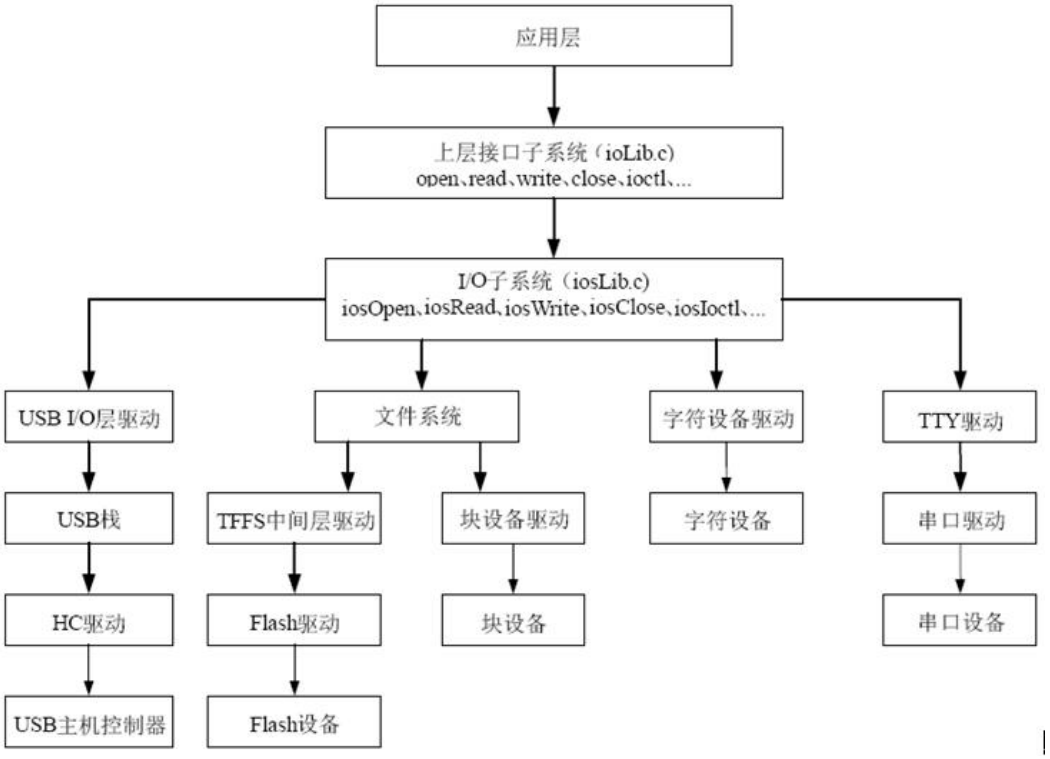
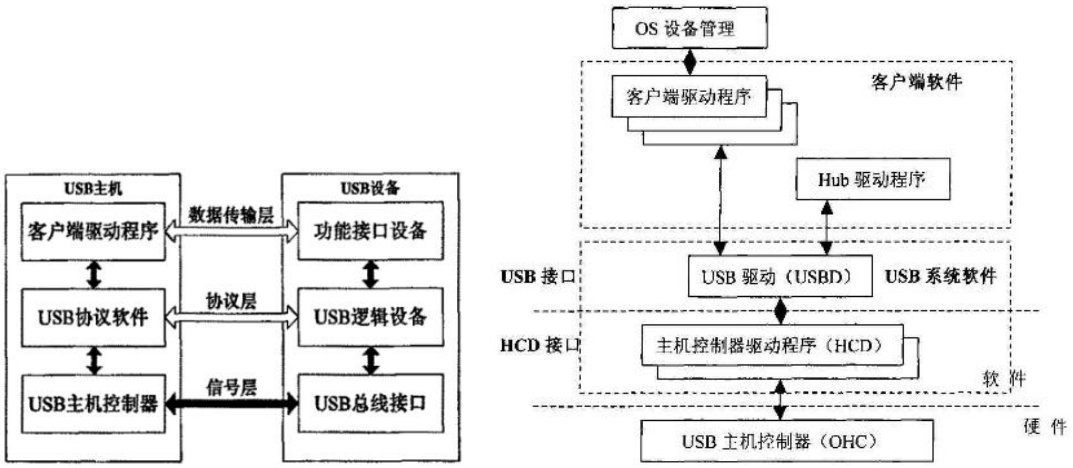


图 3-3　VxWorks 驱动内核层次结构



(a) USB 通信的逻辑结构　　(b) USB 主机端软硬件结构

图 3-4　USB 通信结构

看到的仅仅是具体的设备类,不管设备采用的何种的数据传输方式。另外 IRP 是 USB 协议定义的抽象概念,其结构需要根据协议的具体来实现。

USBD 是 USB 的核心驱动,其提供的功能包括 USB 总线的枚举、总线带宽的分配、传输控制等操作。向上的接口负责处理客户端驱动程序提出的 I/O 请求,他通过 IRP 了解此设备的属性和本次数据通信的具体要求,将此 IRP 转换成 USB 能够识别的一系列的事物处理,交给 HCD 层或直接交给主机控制器处理。USBD 还负责新设

备的动态插拔、USB 电源管理和对客户端驱动程序的维护等操作。

HCD 层的主要功能是与主机控制器合作完成 USB 的各种事物处理。它根据一定的规则调度所有奖杯广播发送到 USB 上的事物处理。调度方法是首先将数据传输类型组成不同的链表，每一种链表包括来自不同的设备驱动程序的同一种类型的数据，然后定义不同的数据类型在传输中所占的带宽比例，交给主机控制器处理，控制器根据规则从立案表上摘下数据块，根据大小为它创建一个或者多个事物处理，完成与设备的数据传输，当事物处理完成时 HCD 将结果交给 USBD 层。此外它还完成对主机控制器和根集线器的配置和驱动等操作。WindRiver 提供了两种类型的主机控制器驱动：usbHcdUhciLib(UHCI 主机控制器驱动) 和 usbHcdOhciLib(OHCI 主机控制器驱动)。VxWorks 的 USBD 和 HCD 之间的接口允许操过一个的底层主控制器，并且 USBD 能够同时连接多个 USB HCD。这样的设计特点可以让开发者建立复杂的 USB 系统。

在 VxWorks 系统当中 UBSD 层的驱动和 HCD 层的驱动都是已经实现好的，我们所需要实现的是客户端的驱动程序，用以驱动特定的 USB 设备。典型的 USB 设备的描述符一般由 USB 标准描述符和 USB 类描述符组成，或者由 USB 标准描述符和 USB 厂商特定描述符组成。任何一个 USB 设备都必须包含 USB 标准描述符，他提供了设备的基本信息和通信方式。为了简化 USB 设备的开发过程，通常会将具有相同的或者是相识的功能的设备归为一类，并指定相关的类规范，这样就能够保证只要按照同样的规范标准，即使是不同的厂商开发的 USB 设备也能够使用相同的驱动程序。针对不同类型的 USB 设备，USB-IF 规定了相关的类描述符，他在标准描述符的基础上进一步说明了特定类型的设备共能以及相关的数据传输方式。但是 USB-IF 规定的设备类描述符并不能够覆盖所有的电子设备，对于没有相关的类描述符的 USB 接口，生产厂商需要利用自己提供的厂商特定功能的类描述符和设备命令对其通信特性做出说明，这些特定功能的描述符和命令的定义和操作完全取决于厂商，要想驱动此类设备就必须要参考厂商提供的这些专有命令。CP2102 模块就属于这种没有相关的类描述符的设备，他不但支持 USB 的标准描述符和 USB 标准命令，还支持自己特定的描述符和命令，我们称这样的设备为非标准类型的 USB 设备。非标准的设备命令和描述符的结构和处理方式与标准设备命令和描述符是一样的，但是它只对特定的功能设备有效。

3.1.3 环形缓冲区的设计

在我们的 USB 口转串口驱动程序当中，我们需要解决低速设备和处理器之间、USB 接口和 RS-232 串口之间的速度匹配问题，如果每次数据传送或接收时系统任务都去操作数据，这样必会造成任务频繁切换，系统效率降低。解决办法之一就是在设备驱动和应用程序之间建立环形数据缓冲，当数据到达时，系统会产生中断，USBD 层会处理中断，然后调用我们注册的输入 IRP 来处理数据，将接受的数据放到我们设计

的缓冲中,应用程序在需要接收数据时,从环形缓冲中的另一端(即最先放入数据的一端)读取数据,是一种先进先出缓冲结构。

应用程序、设备驱动和环形缓冲区的关系如图 3-5所示,通过环形数据缓冲,我们可以解决速度匹配的问题,大大降低系统的开销。我们使用循环队列实现的来实现缓冲区,并设置对头指针和队尾指针,其特点是当一个数据元素被取走以后,其余的数据元素不需要移动其存储位置。在设备初始化时,将设备的环形缓冲区清空,队头指针和队尾指针均设为 0,之后通过操作队头指针与队尾指针来填充数据、移除数据。当驱动中接收到一个数据时,将此数据保存到当前位置 tail++,缓冲区当中需要考虑的关键问题是缓冲区的满和空的判断,因为缓冲区满或者是空都有可能出现读指针或者是写指针指向同一个位置。我们处理这个问题的策略是在缓冲区中保持一个存储单元保存为未使用的状态。即一个大小为 N 的缓冲区中最多只能够存入 N-1 个数据。如果读写指针指向同一个位置那么缓冲区即为空。如果写指针位于读指针的相邻后一个位置,那么缓冲区为满。在测试缓冲区是否满时做取余运算。在缓冲区已满但还是有数据需要写入的时候,此时我们选择的策略是将最开始的数据进行覆盖,这样操作的好处是可以防止最新的数据丢失,而老的数据可能已经失去了时效性。

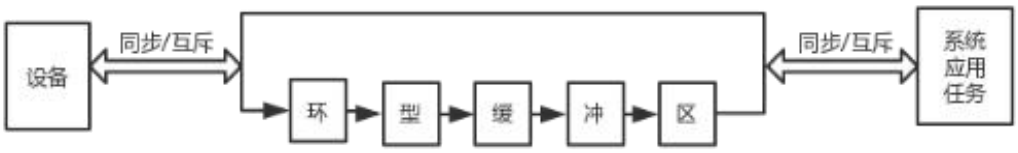


图 3-5　设备数据缓冲

3.2　USB 转串口驱动的实现

在 VxWorks 中驱动程序对上需要匹配操作系统提供的一套规范接口,对下必须驱动硬件设备进行工作,其起着一个关键的中间转换角色,将操作系统的具体请求转换为对硬件的某种操作,让所有的硬件对操作系统的一套内部规范接口进行响应,屏蔽了硬件的所有复杂性,应用层对于某个设备的操作通过操作系统提供的一套标准接口完成,操作系统最终将这些操作请求传递给驱动程序,驱动硬件完成这些请求。VxWorks 调试通道当中运行的最主要的软件平台是嵌入式实时操作系统 VxWorks,作为系统的最底层的软件,要想进行数据的传输,驱动程序是必不可少的。本系统中的硬件设备是基于 USB 总线的,USB 口转串口设备的驱动程序在 Windows 和 Linux 下都有现成可用的,但是在 VxWorks 下需要自己来实现这部分。

在 VxWorks I/ O 当中通常应该经过以下的三个基本步骤来实现一个设备驱动:

1. 实现对实际物理设备的数据结构抽象(即设备的自定义数据结构);
2. 完成 I/ O 系统所需要的各类接口及自身的特殊接口(open、read、write 等);

3. 将驱动集成到操作系统中。

3.2.1 特定需求单设备驱动的实现

由于对于仅支持单设备驱动程序是基于特定的需求而具体定制的,所以该设备的驱动程序的实现流程与通常的支持多设备的驱动的初始化流程存在差异。具体的需求为:

1. 驱动中支持的设备名是固定的,无论具体的设备是否连接上,都可以往这个设备中写入数据。

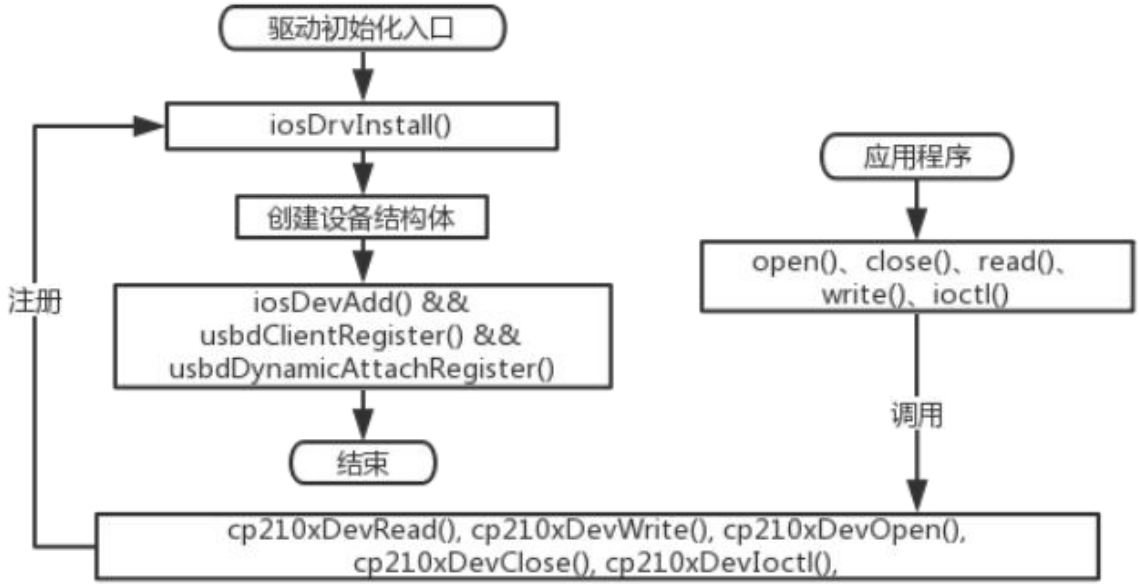
2. 驱动程序中要有缓存一定数据的能力,一旦设备连接之后就能够检查缓冲区中是否有数据,有数据则将其发送出去。

由于需要在设备未连接时就能够往设备中写入数据,且设备名为固定的,那么就必须调整驱动的初始化流程,使得其能够支持这一特性,通常驱动都是在设备加载之后再将其加入到系统设备表和系统驱动表当中,那么此时我们就需要先将一个固定的设备名加入到系统设备表当中。对于该特定需求的单设备驱动的流程图如图 3-6和所示。

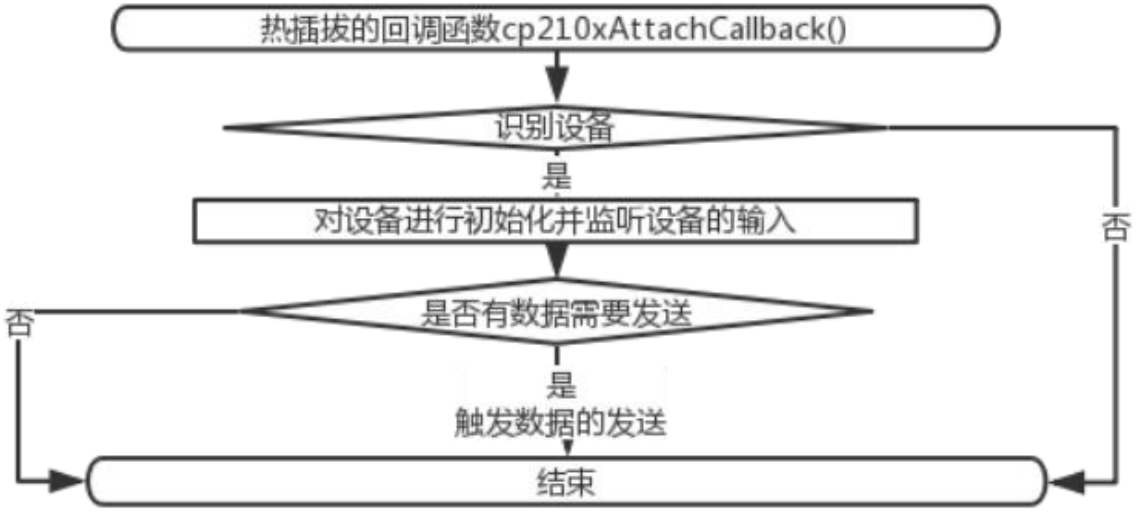
3.2.1.1 设备的自定义结构

底层驱动都要对其驱动的设备维护一个自定义的数据结构,这个结构用来保存所驱动的设备的关键参数,这些信息将随着设备类型的不同而有所差别。对于我们的 USB 口转串口设备需要保存的关键参数有:USB 配置、读写缓冲区的指针、接口配置、端点地址等等。关键数据结构的定义如下:

```
typedef struct cp210x_dev
{
  DEV_HDR cp210xDevHdr;

  UINT16 numOpen;
  USBD_NODE_ID nodeId;
  UINT16 configuration;
  UINT16 interface;
  UINT16 interfaceAltSetting;
  UINT16 vendorId;
  UINT16 productId;

  BOOL connected;
  int trans_len;
  USBD_PIPE_HANDLE outPipeHandle;
  USB_IRP       outIrp;
  BOOL outIrpInUse;
  UINT16  outEpAddr;
  UINT8 trans_buf[64];

  USBD_PIPE_HANDLE inPipeHandle;
  USB_IRP inIrp;
  BOOL inIrpInUse;
```



(a)



(b)

图 3-6　特定需求单设备驱动运行流程图

```
24    UINT8 inBuf[64];
25    UINT16        inEpAddr;
26  } CP210X_DEV, *pCP210XDEV;
```

部分成员的含义如下:

- DEV_ HDR:这一成员必须是自定义设备结构的第一个成员,VxWorks 的 I/O 子系统会把所有的设备结构都看作是这个类型的结构, wind 只会识别到 DEV_ HDR 结构并对其进行管理,在系统的设备列表中,内核只使用 DEV_ HDR 结构当中的成员。

- numOpen: 用来记录设备被打开的次数, 每次调用 open() 函数打开该设备则 numOpen 加一, 调用 close() 函数关闭设备则减一。
- nodeId:用来保存该设备在系统中的唯一 ID 号。
- configure、interface、interfaceAltsetting: 用来保存设备的描述符中的配置、接口、可变接口信息。
- vendorId、productId:保存该设备的厂商 ID 和产品 ID, 用来识别该设备是否适合我们的驱动程序。
- outPipeHandle、inPipeHandle: 设备的输入/输出端点的管道句柄, 每次传输数据时都需要使用该句柄来表明数据传到的哪一个端点。

3.2.1.2 驱动注册和设备创建

底层的驱动程序都要提供一个函数给系统来调用, 以便进行驱动程序的注册和初始化, 这些工作的完成通常是在内核启动过程中进行的 (即内核启动的时候会调用我们的驱动初始化函数)。对于此处的 USB 口转串口驱动我们定义 cp210xDrvInit() 初始化函数, 其主要完成驱动所需要的资源申请和系统的初始化, 包括创建信号量、向系统注册驱动、创建设备、向 USBD 层注册。cp210xDevInit 模块主要代码如下所示:

```
1  STATUS cp210xDrvInit(void)
2  {
3    ...
4    if(OSS_MUTEX_CREATE(&cp210xWriteMutex) != OK || OSS_MUTEX_CREATE(&
         cp210xReadMutex) != OK || OSS_MUTEX_CREATE(&cp210xMutex) != OK || (
         blockReadSem = semBCreate(SEM_Q_FIFO, SEM_EMPTY)) == NULL )
5    ...
6    cp210xDrvNum = iosDrvInstall(NULL,NULL,cp210xDevOpen,cp210xDevClose,
7                           cp210xDevRead,cp210xDevWrite,cp210xDevIoctl);
8    ...
9    if( iosDevAdd(&pCp210xDev->cp210xDevHdr,CP210X_NAME,cp210xDrvNum) != OK
         )
10   ...
11   if(usbdClientRegister (CP210X_CLIENT_NAME, &cp210xHandle) != OK)
12   ...
13   if(usbdDynamicAttachRegister(cp210xHandle,USBD_NOTIFY_ALL,
         USBD_NOTIFY_ALL,USBD_NOTIFY_ALL,TRUE,(USBD_ATTACH_CALLBACK)
         cp210xAttachCallback)!= OK)
14   ...
15 }
```

驱动会在进入初始化的时候首先检查该驱动是否已经安装, 若已经安装了则无需再次安装, 直接退出即可, cp210xDrvInit() 通常是在 usrRoot(usrConfig.c) 中调用, 但是你也可以手动调用这个函数对该驱动进行初始化操作。然后会进行一些驱动所需要的全局资源的初始化, 如信号量、全局变量、看门狗等, 接着调用 iosDrvInstall() 函数安装驱动的 I/O 函数, 将其添加到驱动表当中, 在我们的 USB 口转串口驱动当中不需要实现 delete 函数和 create 函数, 直接将其指针置为 NULL 即可。

注册完成之后还要向系统将该驱动程序添加到 IO 子系统当中, 添加成功后会在

系统的设备列表中显示该命名为 CP210X_ NAME 的设备 (CP210X_ NAME 只是一个宏定义, 设备名可以自己更改)。此处即是我们的驱动程序中的一个特殊的地方, 在没有识别到设备之前就已经创建好设备文件, 只不过此时该设备还只是一个“假”的, 只有软件实现, 没有硬件支撑, 即使此时已经可以打开该设备, 向该设备写入数据, 但是也只是写入到了系统的缓冲区当中而已, 数据并没有发送到任何的硬件上。

接下来 USB 客户端驱动还需要向 USBD 层注册, 注册完成之后会返回一个用于操作 USBD 的客户端 handle, 我们将其保存在 cp210xHandle 这个变量当中。然后还要注册一个动态注册的回调函数, 当 USBD 层发现有 USB 设备的插拔动作时, 就会根据我们注册时选定的设备类和接口类来判断是否需要调用我们注册的回调函数, 由于我们的设备是一个特殊的设备, 并不符合任何标准的 USB 设备类和接口类, 于是我们将这两个参数置为 USBD_ NOTIFY_ ALL, 即任何 USB 设备的插拔都调用我们的注册的回调函数。之后我们在回调函数中根据设备的 v 设备 ID 和厂商 ID 来判断该设备是否能被我们的驱动所支持。在回调函数中我们会发送标准的 USB 设备请求命令来获取该设备的设备描述符, 设备描述符当中包含有设备的类、子类、协议、厂商 ID 等信息, 详细的 USB 设备描述符的信息如 所示;

```
typedef struct usb_device_descr
    {
    UINT8 length;            /* bLength */
    UINT8 descriptorType;            /* bDescriptorType */
    UINT16 bcdUsb;                   /* bcdUSB - USB release in BCD */
    UINT8 deviceClass;               /* bDeviceClass */
    UINT8 deviceSubClass;            /* bDeviceSubClass */
    UINT8 deviceProtocol;            /* bDeviceProtocol */
    UINT8 maxPacketSize0;            /* bMaxPacketSize0 */
    UINT16 vendor;                   /* idVendor */
    UINT16 product;                  /* idProduct */
    UINT16 bcdDevice;                /* bcdDevice - dev release in BCD */
    UINT8 manufacturerIndex;         /* iManufacturer */
    UINT8 productIndex;              /* iProduct */
    UINT8 serialNumberIndex;         /* iSerialNumber */
    UINT8 numConfigurations;         /* bNumConfigurations */
    } WRS_PACK_ALIGN(4) USB_DEVICE_DESCR, *pUSB_DEVICE_DESCR;
...
    usbdDescriptorGet (cp210xHandle, nodeId,USB_RT_STANDARD |
        USB_RT_DEVICE,USB_DESCR_DEVICE,0, 0, sizeof(pBfr), pBfr, &actLen)
        ) != OK)
```

使用 usbdDescriptorGet() 函数来获取设备描述符, 其中第二、第三个参数是 USB 的标准请求命令, 表示我们请求的是设备的标准描述符, 获取到设备标准描述之后提取其中的厂商 ID 和产品 ID, 我们将该驱动支持的设备的厂商 ID 和设备 ID 存储在一个二维数组当中, 然后通过遍历当前插入的设备的 VID 和 PID 是否在我们的数组当中来判断这个设备是否能够被我们的驱动所支持。设备识别的流程如图 3-7所示, 目前支持的设备的设备 ID、产品 ID 的组合如表 3.4所示。

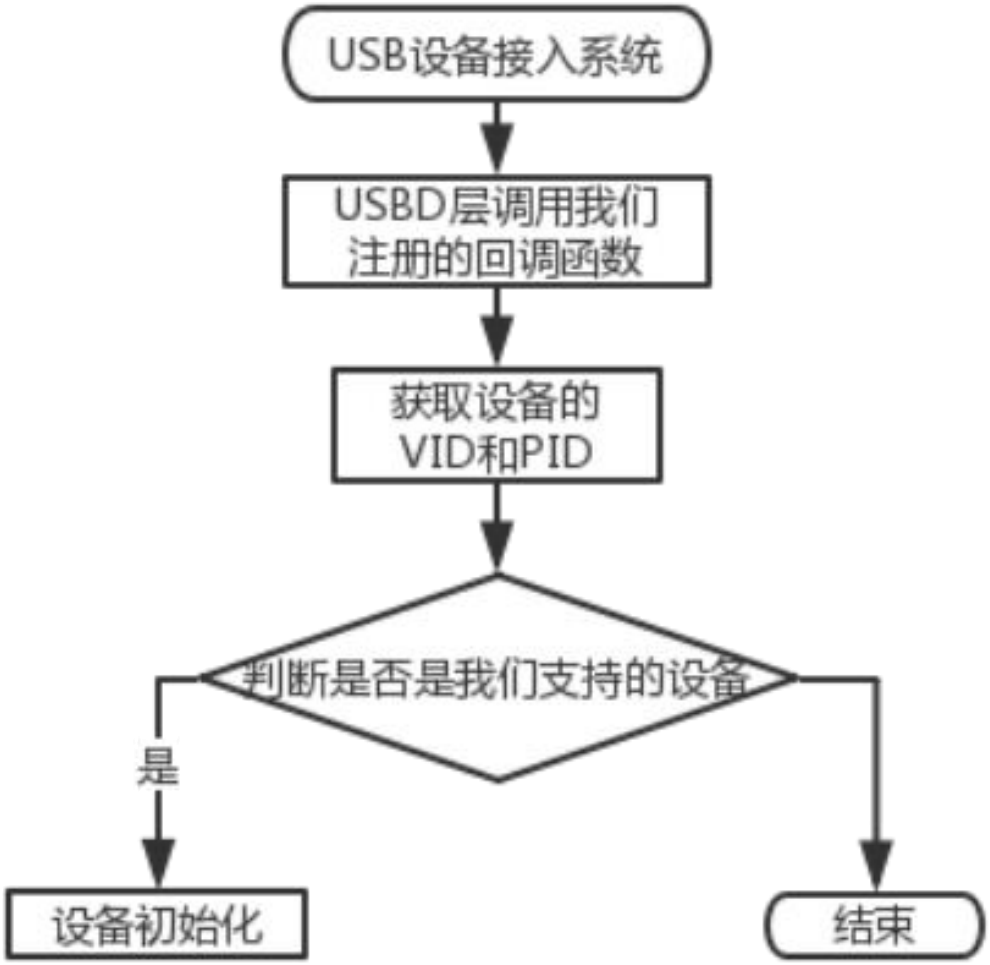


图 3-7　设备识别流程图

PID	0x045B	0x0471	0x0489	0x0489	0x10C4	0x10C4
VID	0x0053	0x066A	0xE000	0xE003	0x80F6	0x8115
PID	0x10C4	0x10C4	0x10C4	0x10C4	0x10C4	0x2405
VID	0xEA60	0x813D	0x813F	0x814A	0x814B	0x0003

表 3.4　目前支持的设备列表

3.2.1.3　驱动缓冲的实现

在驱动的内部我们会建立一个循环缓冲区来接收上层程序写入、从设备接收的数据，若系统中没有 USB 口转串口的设备，那么所有的上层应用往外输出的数据就只会写入到该循环缓冲区当中，当循环缓冲区中的数据已经满了之后，遵循先进先出的原则来覆盖数据。当 USB 口转串口设备连接上时，若循环缓冲区当中已经有数据存在，则会立即启动发送数据的过程，若没有数据，则什么也不做。

在每一个设备初始化的时候我们都会为该设备分配读缓存和写缓存，缓存空间的大小作为以宏的方式定义在头文件当中，方便以后改动，分别定义为：WRITE$_{B}UFFER_{S}IZE$、$READ_{B}UFFER_{S}IZE$。由于计算机的内存是线性地址空间，因此环形缓

在内存中实际开始位置的指针；

在内存中实际结束位置的指针，或者缓冲区的长度；

存储在缓冲区中的有效数据的开始位置 (读指针)；

存储在缓冲区中的有效数据的结束位置的指针 (写指针)。

通过比较我们选择第三种范式方式作为我们底层驱动中缓冲区空/满的判断方式，同时我们会将缓冲区的大小设置成 2 的幂，这样我们就可以不需要镜像指示位和取余操作，只需要通过简单的条件表达式就可以判断缓冲区的空/满，这减少了我们的驱动程序本身的运算量和存储空间的消耗，节省了数据传输过程中的处理时间。

3.2.1.4 设备的初始化

设备的初始化包括获取该 USB 设备的各种描述符信息。包括配置描述符信息、接口描述符信息、端点描述符信息。再通过所获得的这些信息来创建到输出端点的管道和对设备进行设置，我们在此处将设备的波特率初始化为 115200，数据位为 8 位，1 个停止位，没有奇偶校验，没有流控。设备的初始化流程图如图 3-8所示。

这些描述符同样是通过 usbdDescriptorGet() 函数来发送设备的标准描述符命令来获取，在获取到这些信息之后，通过 usbdConfigurationSet()、usbdInterfaceSet() 来配置设备，通过 usbdPipeCreate() 函数来设置设备的输入、输出管道，通过管道来连接设备的输入、输出端点。部分关键代码如下：

```
...
  usbdDescriptorGet (cp210xHandle, pCp210xDev->nodeId,
     USB_RT_STANDARD | USB_RT_DEVICE, USB_DESCR_CONFIGURATION,0, 0,
      USB_MAX_DESCR_LEN, pBfr, &actLen)
  pCfgDescr = usbDescrParse (pBfr, actLen,USB_DESCR_CONFIGURATION)
  pScratchBfr = pBfr;
  ifNo = 0;
  while ((pIfDescr = usbDescrParseSkip (&pScratchBfr,&actLen,
     USB_DESCR_INTERFACE))!= NULL){
    if (ifNo == pCp210xDev->interface)
                break;
        ifNo++;
  }
  pOutEp = findEndpoint(pScratchBfr, actLen, USB_ENDPOINT_OUT)
  pInEp = findEndpoint(pScratchBfr,actLen,USB_ENDPOINT_IN))
...
```

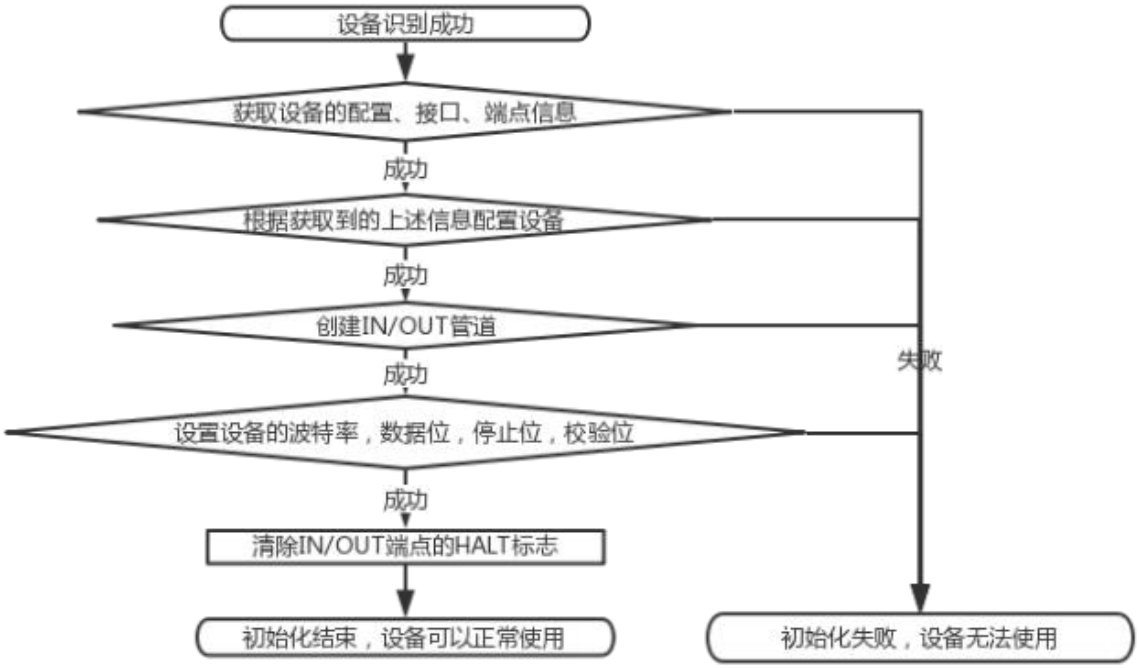


图 3-8　设备初始化流程图

```
14    usbdConfigurationSet (cp210xHandle, pCp210xDev->nodeId,pCfgDescr->
          configurationValue,pCfgDescr->maxPower * USB_POWER_MA_PER_UNIT
          )
15  ...
16    usbdInterfaceSet(cp210xHandle,pCp210xDev->nodeId,pCp210xDev->
          interface,pIfDescr->alternateSetting);
17  ...
18    usbdPipeCreate(cp210xHandle,pCp210xDev->nodeId,pOutEp->
          endpointAddress,pCfgDescr->configurationValue,pCp210xDev->
          interface,USB_XFRTYPE_BULK,USB_DIR_OUT,maxPacketSizeOut,0,0,&
          pCp210xDev->outPipeHandle)
19  ...
20    usbdPipeCreate(cp210xHandle,pCp210xDev->nodeId,pInEp->
          endpointAddress,pCfgDescr->configurationValue,pCp210xDev->
          interface,USB_XFRTYPE_BULK,USB_DIR_IN,maxPacketSizeIn,0,0,&
          pCp210xDev->inPipeHandle)
21  ...
```

3.2.1.5　设备打开/关闭函数

用户在使用一个设备之前必须先对这个设备进行打开，在这个过程当中底层驱动的响应函数通常会将进行中断注册和使能设备工作配置等操作。但是对于我们的 USB 口转串口驱动而言，我们不需要自己管理中断，USBD 层会为我们进行中断的管理工作，而配置工作我们在设备的初始化工作中已经完成，因此此处我们的设备打开工作，只需要简单的记录设备被打开的次数，并返回即可。设备的打开代码如下所示：

```
LOCAL CP210X_DEV * cp210xDevOpen(DEV_HDR *pDevHdr, char *name, int
    flags,int mode)
{
        CP210X_DEV *pCp210xDev;
        pCp210xDev = (CP210X_DEV *)pDevHdr;
        if(!pCp210xDev->connected)
                return (ERROR);

        (pCp210xDev->numOpen)++;
        return (pCp210xDev);
}
```

第一个 DEV_ HDR 类型的参数即是我们之前所说的必须是设备自定义结构的第一个成员，在此时它会由 IO 子系统自动提供，IO 子系统会根据驱动号寻址到对应驱动函数时，然后将对应的系统设备列表中存储的设备结构作为第一个参数来调用 cp210xDevOpen()。我们之后在使用的时候需要首先将这个 DEV_ HDR 结构转换成我们的自定义结构 CP210X_ DEV。但是我们也可以直接将第一个参数的类型设置为自定义结构类型，那么对于我们 USB 口转串口驱动，以上 cp210xDevOpen() 函数的调用原型就变为：LOCAL int cp210xDevOpen(CP210X_ DEV *pCp210xDev, char *name, int flags,int mode) 这并不会造成什么影响。

第二个参数是设备名匹配后的剩余部分，注意不是我们传过来的设备名。VxWorks 中使用最佳匹配的原则来匹配设备名，如我们打开一个设备”/ttyUsb/xyz”，但是系统中并没有这个设备，只有设备”/ttyUsb/x”，那么此处的 name 就是匹配之后剩余的字符串”yz”。我们的应用中 open() 函数调用时的路径名与系统设备列表中的设备名是完全匹配的，此处的 name 就会是一个空字符串。对于在文件系统层下的块设备来说，此处指向的应该是块设备节点名后的子目录和文件名。

第三，四个参数就是用户 open() 调用时传入的第二，三个参数，IO 子系统不会对他们进行更改，只是原封不动的转发给了 cp210xDevOpen() 函数。

该函数的返回值有如下两种值：一个有效的 CP210X_ DEV 结构指针表示 cp210xDevOpen 调用成功，ERROR 则表示 cp210xDevOpen() 调用失败，IO 子系统会根据返回的指针是否有效来决定返回一个文件描述符还是返回一个错误。cp210xDevOpen() 函数的返回值非常重要，这个指针将被 IO 子系统保存，用于其后对驱动中读写，控制函数的调用。这个返回的指针也会作为这些函数的第一个参数。

设备的关闭和设备的打开操作是相反的操作，在关闭操作当中我们只需要对设备记录的打开次数进行减法操作即可。

3.2.1.6 设备的读写

在成功打开一个设备后，用户程序将得到一个 open() 返回的文件描述符，此后用户就可以使用这个文件描述符对设备进行读写，控制操作。我们的 USB 口转串口驱动的读写函数原型如下：

```
1 LOCAL int cp210xDevWrite(CP210X_DEV *pCp210xDev, char *buffer,UINT32
      nBytes)
2 LOCAL int cp210xDevRead(CP210X_DEV *pCp210xDev, char *buffer,UINT32
    nBytes)
```

此处我们将这两个函数的第一个参数类型直接设置为 CP210X_ DEV 结构类型而不是 DEV_ HDR，只是为了向大家展示两种方式都是允许的，在我们的实际编程中应该一种方式一以贯之。

由于我们的驱动程序的初始化方式比较特殊，所以此处对数据的输出操作的流程也需要有相应的改变。在驱动程序中我们设置了一个 4K 的数据缓冲区来接收数据，设备的初始化之后就先查询缓冲区中是否存在数据，若存在数据则将其发送，若不存在则等待其他程序调用 write 写入数据来触发数据的发送操作，系统调用 write() 在该驱动程序中对应的函数为 cp210xDevWrite() 函数，这个函数会接受 write() 发送过来的数据，并将其存入缓冲区中，并判断是否需要触发数据的发送操作。发送数据的触发操作只会在当前没有数据在发送时完成，若调用 write 时设备已经在发送数据，则只是将 write 的数据存入缓冲区，设备发送完当前正在发送的数据之后会去判断缓冲区中是否还有数据，有数据则会继续发送。其流程图如图 3-9 所示。

cp210xDevRead 和 cp210xDevWrite 的实现非常相似，只是更换了一下数据的传输方向。cp210xDevRead 在底层也有一个循环缓冲区用来接收从 USB 口转串口设备发送来的数据，当需要读取 nbytes 个字节而缓冲区内的字节不够时，read 就会阻塞，直到 USBD 层通知你有新的数据到来时才会继续进行读操作。同时也在驱动中启动了一个计时器，如果在计时器时间到了之后，还未能满足需要读取的字节数，则退出本次读写操作，返回当前已处理的字节数。

3.2.1.7 设备的控制操作

设备控制操作用于对设备的某一些工作行为进行再配置，可进行的在配置类别随着设备类型的不同而不同，操作系统当中通常会一种类型的设备的某一组共同属性作为一个配置选项，比如波特率再配置就是串口的一个标准属性，而一般的 USB 设备是不具有该属性的。但是这只是一个约定，并不是所有的设备都必须要完全对照这一准则，底层驱动也可以根据自己的实际需要来对这些再配置属性进行选择，我们可以选择只实现某一些再配置参数，也可以根据具体情

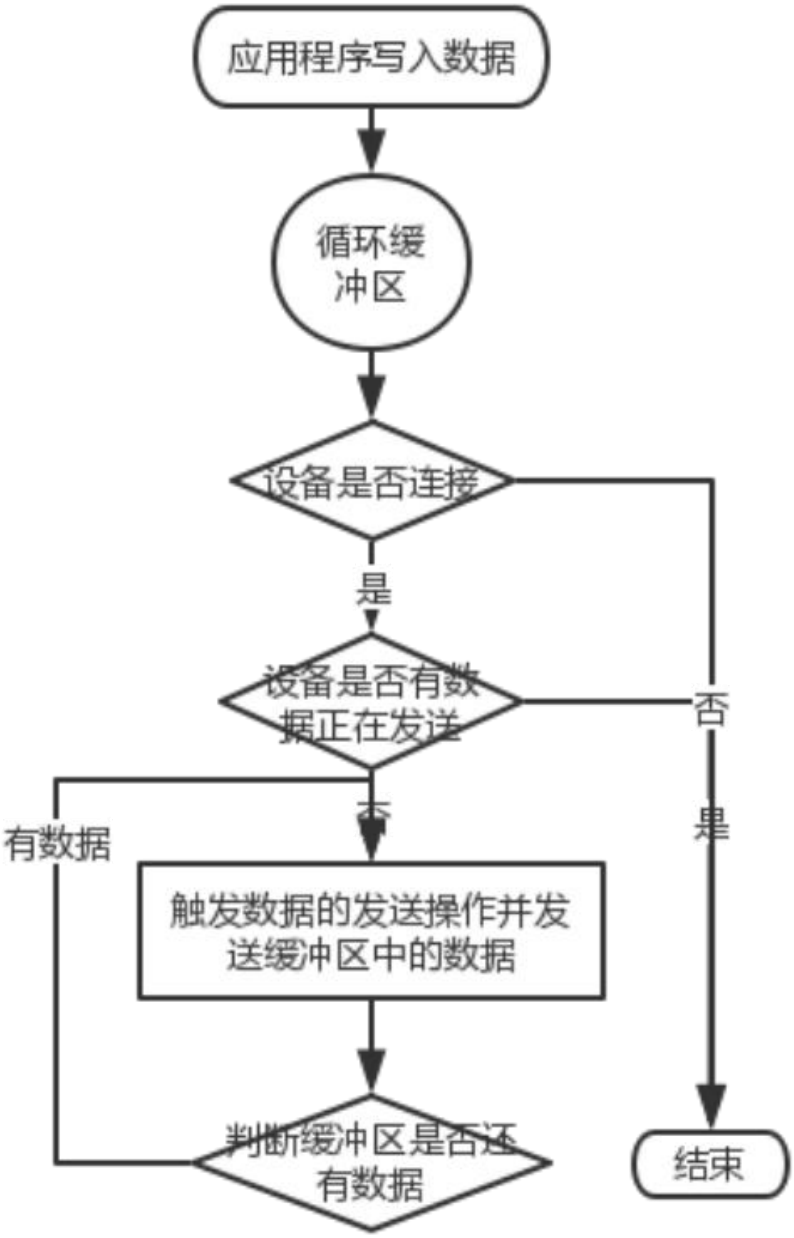


图 3-9　数据发送逻辑图

况对某一个再配置选项进行响应，设备控制函数给用户控制设备提供方便的同时也对底层设备的实现提供了极大的方便性。

对于我们的 USB 口转串口驱动而言，其属于一个特殊的设备，没法归入操作系统的已经分好类的设备当中，我们需要实现一些非约定的配置属性，如配置波特率、流控、数据位等等非 USB 所属的配置选项。我们将 USB 口转串口驱动特定的参数定义在一个头文件当中，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 子系统不会对用户调用的 ioctl() 函数做任何的改变，只会将用户使用的选项参数或者控制命令传递给我们的 cp210xDevIoctl() 函数，然后由这个函

数完成对选项参数或控制命令的解释和使用。设备控制函数原型如下:

```
LOCAL int cp210xDevIoctl(CP210X_DEV *pCp210xDev, int request, void *
    someArg )
```

对于我们的 USB 口转串口驱动, 在实际使用中, 再配置参数和命令有很多, 但是目前我们只提供设备的波特率、数据位、校验位、流控的参数和命令。这些控制对于普通的 USB 设备而言是没有的, 他们在定义上属于 USB 的厂商自定义请求, 在我们的驱动程序的 cp210xDevIoctl() 函数当中最后都要使用 usbdVendorSpecific() 函数来发送 USB 的厂商自定义请求,该函数的原型如下:

```
STATUS usbdVendorSpecific
(
  USBD_CLIENT_HANDLE clientHandle,/* Client handle */
  USBD_NODE_ID nodeId,  /* Node Id of device/hub */
  UINT8 requestType,    /* bmRequestType in USB spec. */
  UINT8 request,                /* bRequest in USB spec. */
  UINT16 value,                 /* wValue in USB spec. */
  UINT16 index,                 /* wIndex in USB spec. */
  UINT16 length,                /* wLength in USB spec. */
  pUINT8 pBfr,                  /* ptr to data buffer */
  pUINT16 pActLen               /* actual length of IN */
)
```

其中第二个参数是请求类型, 表示该类型是从主机到设备的, 还是从设备到主机的, 有四种类型, 分别是 REQTYPE_ HOST_ TO_ INTERFACE(0x41),REQTYPE_ INTERFACE_ TO_ HOST(0xC1),REQTYPE_ HOST_ TO_ DEVICE(0x40),REQTYPE_ DEVICE_ TO_ HOST((0xC0)), 第三个参数是具体的请求, 对于我们的设备而言能响应的部分主要请求如表 3.5所示。

宏名	代码	宏名	代码
CP210X_ IFC_ ENABLE	0x00	CP210X_ SET_ BAUDDIV	0x01
CP210X_ GET_ BAUDDIV	0x02	CP210X_ SET_ LINE_ CTL	0x03
CP210X_ GET_ LINE_ CTL	0x04	CP210X_ SET_ BREAK	0x05
CP210X_ IMM_ CHAR	0x06	CP210X_ SET_ MHS	0x07
CP210X_ SET_ XOFF	0x0A	CP210X_ SET_ XON	0x09
CP210X_ SET_ FLOW	0x13	CP210X_ GET_ FLOW	0x14
CP210X_ EMBED_ EVENTS	0x15	CP210X_ GET_ EVENTSTATE	0x16
CP210X_ SET_ CHARS	0x19	CP210X_ GET_ BAUDRATE	0x1D
CP210X_ SET_ BAUDRATE	0x1E	CP210X_ VENDOR_ SPECIFIC	0xFF

表 3.5　厂商指定请求

3.2.1.8 驱动卸载

在驱动的初始化函数 cp210xDrvInit() 当中我们使用了 iosDrvInstall() 函数向 IO 子系统注册我们的驱动，在 wind 中也提供了另外一个相反作用的函数 iosDrvRemove() 注销我们的驱动。该函数调用原型如下:

```
STATUS iosDrvRemove
 (
 int drvnum, /* driver to remove, returned by iosDrvInstall()*/
 BOOL forceClose /* if TRUE, force closure of open files */
);
```

该函数的第二个参数指定是否强制进行卸载，并将所有与此驱动有关的文件描述符关闭。如果强制关闭，则 IO 子系统将遍历系统文件描述符表，检查每个描述符对应结构中的驱动号是否等于要卸载驱动的驱动号，如果相同，则调用这个驱动的 close 实现函数进行关闭，同时释放文件描述符表中该表项，此时用户层的文件句柄将自动失去功效，如果用户其后使用这个文件描述符，将直接得到一个错误返回。

除了驱动卸载函数之外，我们的驱动初始化时还向 USBD 层进行了注册，在卸载的时候也应该注销 USBD 层的注册，同时注销动态注册的回调函数并从系统的设备表当中删除掉该设备。之后还应该对该驱动占用的所有其他的系统资源进行释放。

至此，我们已经完成了该特定需求下的 USB 口转串口驱动程序的所有组成部分的设计和实现。

3.2.2 通用多设备驱动的实现

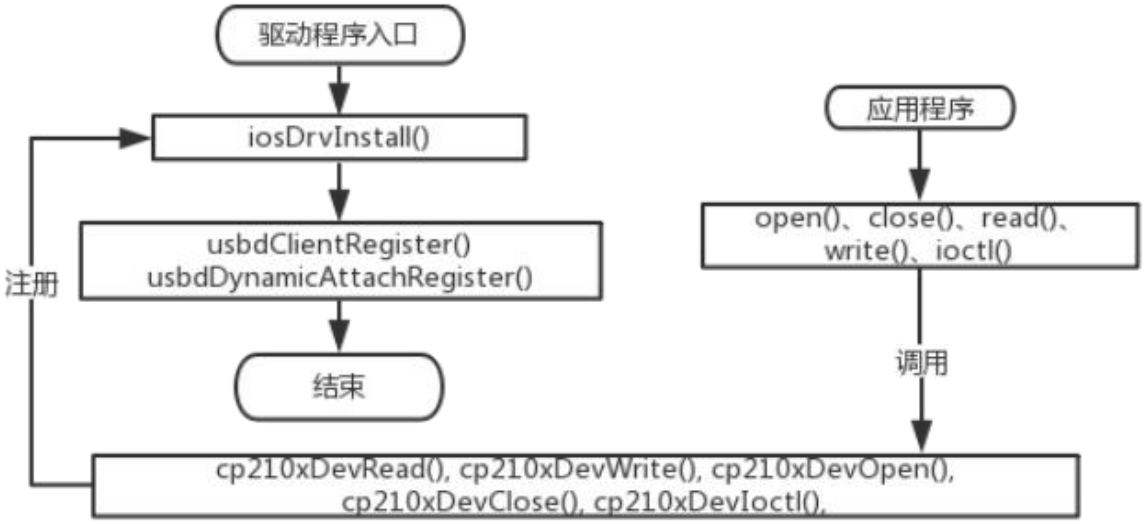
多设备支持的驱动程序在初始化的时候与特定需求单设备支持的驱动初始化的过程是不一样的，需要支持多设备，首先需要在识别设备后给每一个设备分配一个设备名，并加入到系统设备表当中，然后需要给每一个设备初始化自己的设备结构体多设备下的驱动运行流程如图 3-10所示。

与特定需求单设备的驱动相比，这里驱动程序的变化主要在驱动的初始化和读写函数部分，同时设备自定义的数据结构会更加复杂，需要保存更多与具体的设备相关的信息。

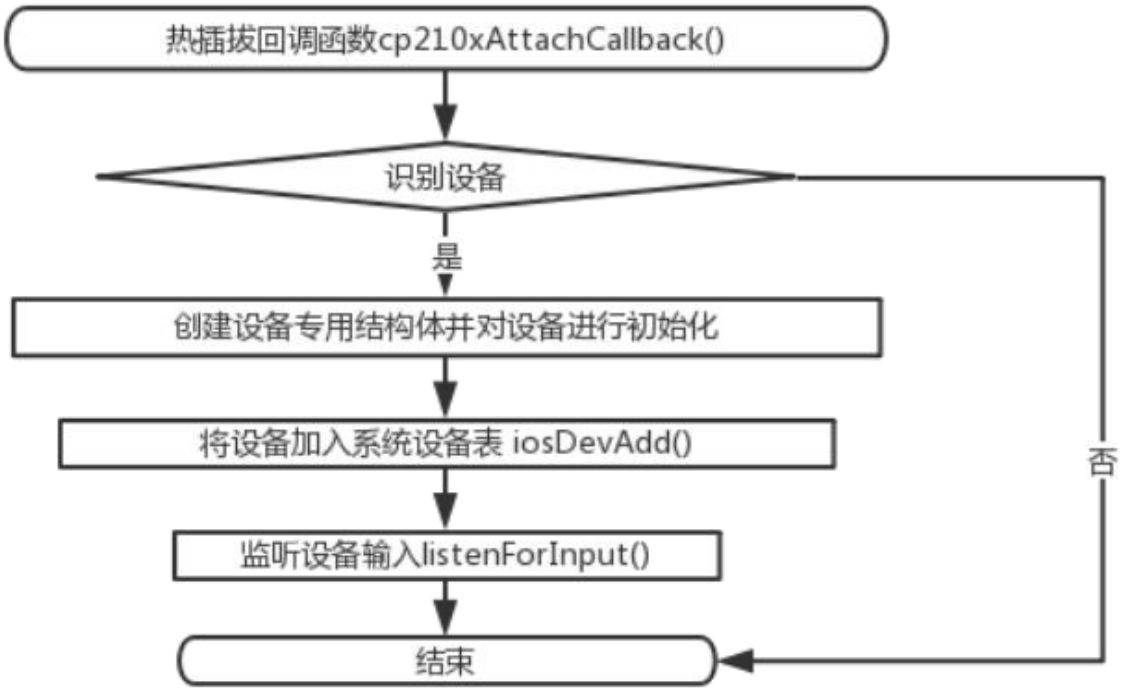
3.2.2.1 设备的自定义结构体

多设备的驱动自定义结构体的定义如下所示:

```
typedef struct cp210x_dev
{
```



(a)



(b)

图 3-10　多设备驱动运行流程图

```
    DEV_HDR cp210xDevHdr; /*must be first field*/
    LINK  devHdrLink; /*linked list of  devhdr structs*/
    UINT16 numOpen;
    USBD_NODE_ID nodeId; /*device nodeID*/
    UINT16 configuration;
    UINT16 interface; /*a interface of this device*/
    UINT16 interfaceAltSetting;
    UINT16 vendorId;
    UINT16 productId;
    BOOL connected;

    int trans_len;
    USBD_PIPE_HANDLE outPipeHandle; /* USBD pipe handle for bulk OUT
        pipe*/
    USB_IRP       outIrp; /*IRP to monitor output to device*/
    BOOL outIrpInUse;
    UINT32 outErrors; /*TRUE while IRP is outstanding*/
    UINT8 trans_buf[64];
    UINT16  outEpAddr;

    USBD_PIPE_HANDLE inPipeHandle;
    USB_IRP inIrp;
    BOOL inIrpInUse;
    UINT8 inBuf[64];
    UINT32 inErrors;
    UINT16 inEpAddr;

    char *writeBuf;
    int writeFront;
    int writeRear;
    char *readBuf;
    int readFront;
    int readRear;
} CP210X_DEV, *pCP210XDEV;
```

在这个结构体当中我们增加了每个设备的读写缓冲区的指针和各个缓冲区的头尾指针。同时使用了一个链表 devHdrLink 来链接在系统上的由该驱动支持的设备,每次检测到新设备时我们可以通过将新添加的设备增加到这个链表当中,之后可以通过 nodeId 来从多个设备中定位我们的设备是否存在,之后我们可以给每一个设备分配一个设备名。部分代码如下所示:

```
...
 usbListLinkProt(&devListHdr,(pVOID)pCp210xDev,(pLINK)&pCp210xDev->
    devHdrLink,LINK_TAIL,cp210xMutex);
...

LOCAL pCP210XDEV findDevHdr(USBD_NODE_ID nodeId)
{
        pCP210XDEV pCp210xDev =  usbListFirst(&devListHdr);

        while(pCp210xDev != NULL)
        {
                if(pCp210xDev->nodeId == nodeId)
                        break;
```

```
                pCp210xDev = usbListNext(&pCp210xDev->devHdrLink);
        }
        return pCp210xDev;
}
```

3.2.2.2 驱动注册和设备创建

比较单设备驱动初始化 (如图 3-6a所示) 和多设备驱动初始化 (如图 3-10a所示), 我们可以看出在驱动注册过程中两者的区别, 在单设备驱动的初始化中我们先完成设备结构体的创建并定好一个设备名, 之后直接将其加入到系统的设备表当中, 即使此时没有设备连接。而在多设备的驱动初始化当中我们是在驱动的回调函数中驱动识别完了设备之后再完成设备结构体的创建和加入系统设备表的, 这种方式是通用的设备驱动常采用的方式。在设备创建时我们会通过判断已连接设备的个数来决定当前设备所采用的设备名, 部分代码如下:

```
LOCAL int getCp210xDeviceNum(CP210X_DEV *pCp210xDev)
{
...
  for (int index=0; index < CP210X_MAX_DEVICE; index++)
    if (pCp210xDevArray[index] == NULL){
      pCp210xDevArray[index] = pCp210xDev;
      return (index);
    }
...
}

LOCAL STATUS cp210xAttachCallback(USBD_NODE_ID nodeId, UINT16
    attachAction,UINT16 configuration,UINT16 interface,UINT16
    deviceClass,UINT16 deviceSubClass, UINT16 deviceProtocol)
{
  ...
  cp210xUnitNum = getCp210xDeviceNum(pCp210xDev);
  sprintf (cp210xName, "%s%d", CP210X_NAME,cp210xUnitNum);
  if(iosDevAdd(&pCp210xDev->cp210xDevHdr,cp210xName,cp210xDrvNum) !=
      OK)
  ...
}
```

3.2.2.3 设备读写

对于通用多设备的写操作, 与设定设备的操作不同的是设备连接上时, 没有一个自动发送缓冲区的数据的过程, 此时设备没有连接上也不可能往缓冲区中写入数据。其基本流程如下:

1. 将数据拷贝到输出循环缓冲区当中, 若缓冲区已满则等待。
2. 判断设备是否仍然处于连接状态。

3. 若处于连接状态，那么是否有数据正在发送当中。若有数据正在发送，则等待。
4. 若没有数据在发送，则触发发送数据的操作。
5. 返回发送的字节数。

对于数据发送的逻辑控制，使用了 VxWoks 的同步和互斥信号量。首先在写入数据的时候需要进行互斥写，因为此时设备有可能正在从缓冲区当中取数据进行输出操作，那么这是写入输出缓冲区就需要等待，否则可能会造成缓冲区的混乱，造成输出结果与输入数据不一致。当设备输出从缓冲区拷贝完成之后就会释放互斥信号量，此时写入操作就可以往输出缓冲区中写入数据。

对于通用多设备的读操作，我们会事先创建好一个用来读取数据的 USB IRP 在这个 IRP 中我们注册一个回调函数，USB 的中断会由 USBD 层来替我们进行管理，当有数据到来时 USBD 层会调用我们注册的回调函数来通知我们。在驱动程序初始化完成之后我们就会启动 listenForInput() 这个函数来注册一个 USBD 的通知过程，一个接收 IRP 使用完成之后，我们需要重新注册一次，因为每一次的 IRP 都是单次有效的，所以在 cp210xIrpCallback() 中我们接受完这一次的 IRP 的数据之后，需要新建另一个 IRP 重新启动下一次的 listenForInput() 过程。部分关键代码如下：

```
1
2  LOCAL STATUS listenForInput(CP210X_DEV *pCp210xDev)
3  {
4  ..
5    pIrp->userPtr = pCp210xDev;
6    pIrp->userCallback = cp210xIrpCallback;
7    pIrp->timeout = USB_TIMEOUT_NONE;
8    pIrp->transferLen = 64;
9    if(usbdTransfer (cp210xHandle, pCp210xDev->inPipeHandle, pIrp) !=
         OK)
10 ...
11 }
12
13 LOCAL void cp210xIrpCallback(pVOID p)
14 {
15 ...
16         if(pIrp == &pCp210xDev->inIrp && pCp210xDev->connected ==
               TRUE)
17         {
18                 if(pIrp->result == OK)
19                         copy_to_readBuf(pCp210xDev);
20
21                 if (pIrp->result != S_usbHcdLib_IRP_CANCELED)
22                         listenForInput(pCp210xDev);
23         }
24 }
```

其他的部分如设备的控制、设备打开/关闭、设备卸载函数与单设备下的相比不

需要做改变即可完成, 此时操作的设备就是我们使用设备名打开的那个设备, IO 子系统会将设备名映射到该设备所对应地驱动。

3.3 小结

本章首先介绍了我们的 USB 口转串口驱动的设计想法, 包括 USB 口转串口所使用转换器的选择, 驱动程序所需要实现的模块, 每个模块的功能是什么。接下来对所选择的转换器 CP2102 的开发进行了介绍, 对 VxWorks 中 USB 的开发进行了介绍, 最后根据我们的设计方案对驱动程序的每一个部分进行了实现。

四　应用层程序接口封装

从 VxWorks6.x 开始引入了 RTP(VxWorks Real Time Process Project) 模式，这种模式的优点是应用程序之间互相独立、互不影响，而且增加了内核的稳定性，缺点是由于“内核态”与“用户态”的内存拷贝，其执行效率有所降低，随着 CPU 速度越来越快，这点效率的牺牲已经越来越不重要。相比较于传统的 DKM（downloadable kernel module project），RTP 适合多个团队独立运作，然后汇总联试，这种模式除了全局函数不能在 shell 里直接调用外，其对应用程序几乎不做任何约束，原有的 DKM 工程代码稍作修改即可正常运行。内核变化较大，需要添加较多的组件，内存需要较好的划分，为保持应用程序直接调用函数调试的习惯，需要封装接口供用户使用。

4.1　应用层接口模块设计

应用层的接口包括两个部分：一是标准输出重定向接口的设计，目的是在程序运行期间直接调用标准输出函数时就能够将输出信息通过我们的串口输出出去，但是这个标准输出输出的信息不是格式化的信息。二是 Log 接口函数，通过调用这个接口函数可以将调试信息格式化输出，输出信息会自动包括调试的级别、产生的时间、所处的文件、行号等信息，便于对调试信息进行分析。

主要的模块包块以下两个：

- ResetStdOut(): 提供给用户选择是否需要重定向标准输出，若参数为 1 则将标准输出进行重定向，若参数为 0，则关闭标准输出重定向，恢复到之前的标准输出。
- Log **接口函数：**提供封装的不同级别的 Log 调试接口，包括 LogE(表示错误信息)、LogD(表示详情信息)、LogW(表示警告信息)、LogI(表示)

由于从 VxWorks6.x 开始引入了 RTP(VxWorks Real Time Process Project) 模式，这种模式的优点是应用程序之间互相独立、互不影响，而且增加了内核的稳定性，缺点是由于“内核态”与“用户态”的内存拷贝，其执行效率有所降低，随着 CPU 速度越来越快，这点效率的牺牲已经越来越不重要。在这种 RTP 模式下对于我们的标准输出重定向接口而言，其实现方式与 task 模式下存在一些差别，在 task 模式下我们需要使用 VxWorks 封装好的 ioTaskStdSet() 函数来实现重定向，而在 RTP 模式下我们无法使用，只能寻找其他的解决办法，在此处我们使用的是 dup2()/dup() 来实现。

4.2 Log 协议的设计

对于 Log 接口函数我们设计为其为设计了专用的输出协议，如图 4-1 所示，在协议中包含了五种不同的输出级别，分别用 LogE(表示错误信息)、LogD(表示调试信息)、LogW(表示警告信息)、LogI(表示详情信息)、Logo(表示其他信息)。同时我们还在协议中包含了一些调试信息所需要的关键信息字段，包括任务 ID 字段、任务名字段、文件名字段、行号字段、时间字段。

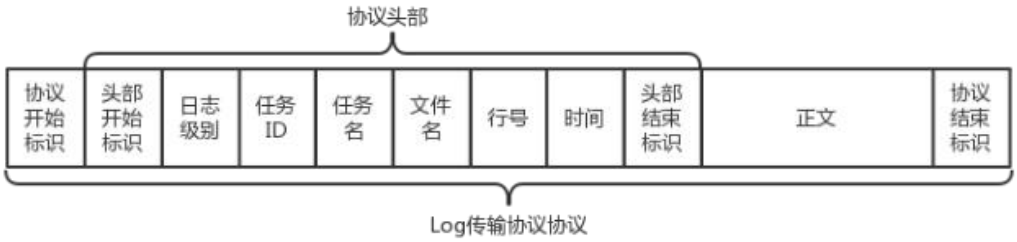


图 4-1 Log 协议字段

我们为调试信息制定的协议格式为：$\backslash 03\backslash 03 < L =$ 日志级别; $PN =$ 任务ID; $P =$ 任务名; $F =$ 文件名; $N =$ 行号; $T =$ 时间 $> contents\backslash 04\backslash 04$
其中各部分的含义如下：

- $\backslash 03\backslash 03$：表示自定义的$Log$协议的数据包的开始；
- **<**: 表示自定义的 Log 协议数据包头部的开始；
- **L**：表示日志的级别，我们在此将日志分为五个级别：
  - e: 表示 error；
  - w: 表示 warning；
  - i: 表示 info；
  - d: 表示 debug；
  - o：表示其他信息。
- **PN**: 此处的内容是输出该条调试信息的任务的任务 ID；
- **P**：此处的内容是输出该条调试信息的任务的任务名；
- **F**：此处的内容是输出该条调试信息的任务所在的文件名；
- **N**：此处的内容是该调试信息语句所在的文件的行号；
- **T**：此处的内容是这条调试信息被输出时候的系统时间；
- **>**: 表示自定义的 Log 协议数据包头部的结束；
- **contents**：这个部分是调试信息的正文部分。
- $\backslash 04\backslash 04$: 表示自定义的Log协议数据包的结束。

4.3 标准输出重定向接口的实现

由于标准输出的重定向无法在 RTP 模式和 task 模式下使用同一种方法来实现，于是我们使用了两种方法来分别实现 RTP 模式和 task 模式下的标准输出重定向。

4.3.1 RTP 模式下标准输出重定向

RTP 模式下的标准输出重定向流程如图 4-2所示。部分关键代码如下：

```
int ResetStdOut(int usb_serial)
{
   ...

   _init_fd();
   if(usb_serial == 1)
   {
     if(dup2(log_fd,STD_OUT) < 0)
     {
       printf("can not reset STDOUT to /hust_use_serial\n");
           return -1;
         }
   }
   else
   {
     if (dup2(STDOUT_FD,STD_OUT) < 0)
     {
           printf("can not reset STDOUT to /hust_use_serial\n");
           return -1;
         }
   }

   ...
}
```

在 RTP 模式下使用 dup2 函数来实现标准输出的重定向，dup2 函数的原型为：

```
int dup2(int oldfd, int newfd);
```

dup2() 用于复制描述符 oldFd 到 newFd 的，其中 oldFd 是要被复制的文件描述符，newFd 是制定的新文件描述符，如果 newFd 已经打开，它将首先被关闭。如果 newFd 等于 oldFd，dup2 会返回 newFd，但是不会关闭它。函数调用成功时会返回新的文件描述符，所返回的新的描述子与参数 oldFd 给定的描述符字引用同一个打开的文件，即共享同一个系统打开文件表项。函数调用失败时会返回-1 并设置 errno。

4.3.2 task 模式下标准输出重定向

在 task 模式下无法使用 dup()/dup2() 函数来进行标准输出的重定向，在 task 模式下 VxWorks 有专用的标准输出接口 ioTaskStdSet()，我们在此模式下只能使用这个借口

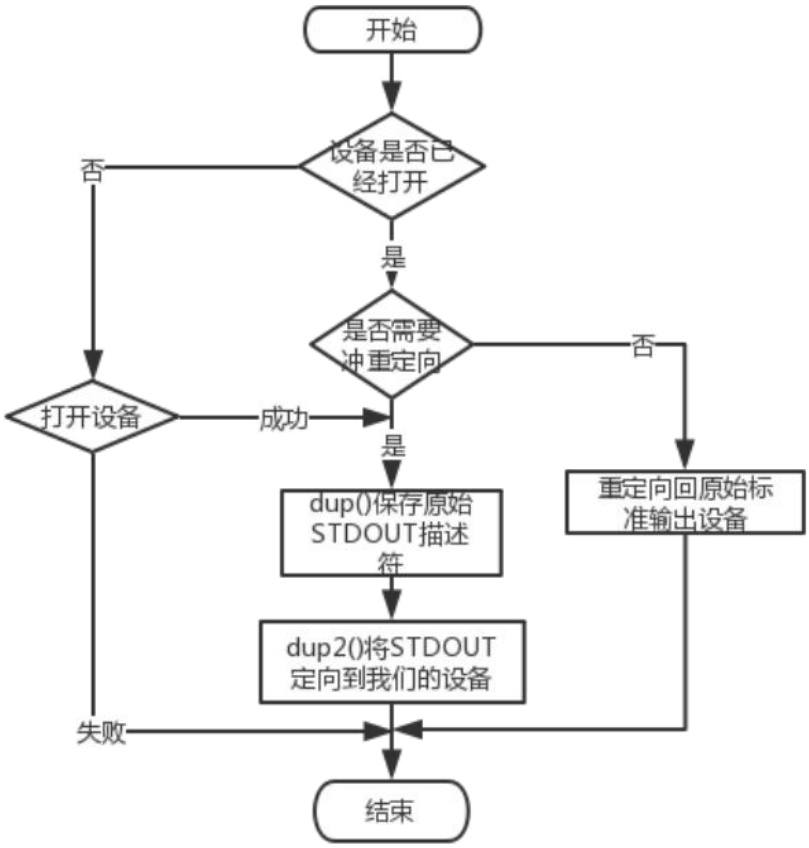


图 4-2　RTP 模式下标准输出重定向流程图

来实现重定向，task 模式下的标准输出重定向如图 4-3所示。部分关键代码如下：

```
int ResetStdOut(int usb_serial)
{
  ...

  _init_fd();
  if(usb_serial == 1)
  {
    ioTaskStdSet(0,STD_OUT,log_fd);
  }
  else
  {
    ioTaskStdSet(0,STD_OUT,STDOUT_FD);
  }

  ...
}
```

ioTaskStdSet() 是 VxWorks 专门用来进行任务级的重定向的函数。其函数原型为：

```
void ioTaskStdSet(int taskId, int stdFd, int newFd);
```

在 VxWorks 中每一个任务都有一个数组 taskStd，用于表明这个任务的标准输入、标准输出、标准错误，函数 ioTaskStdSet() 的功能就是将特定任务的标准描述符重定向到

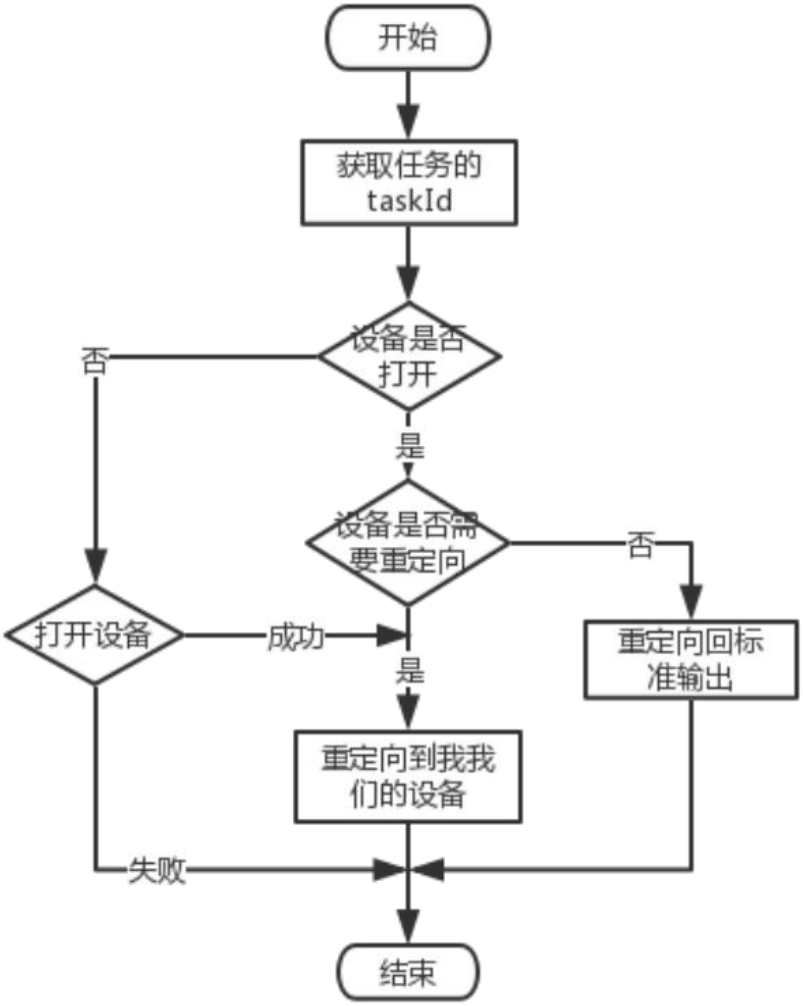


图 4-3　task 模式下标准输出重定向流程图

newFd, newFd 需要是一个文件或者设备的描述符。第一个参数 taskId 表示需要进行重定向的任务的 ID(ID 为 0 表示该任务本身), 第二个参数是需要被重定向的某个标准描述符 (0,1,2), 第三个参数是需要重定向到的文件描述符。该函数没有返回值。

4.4　Log 接口的实现

Log 接口函数用于完成标准格式的 log 的输出, 使用时只需要调用 LogE()、LogW()、LogI()、LogD()、LogO(), 这几个接口均为宏定义, 定义在 usb_ logWrite.h 当中, 在使用时需要包含该头文件, 作用是获取 log 协议所需要的部分信息, 其代码如下所示:

```
1 #define LogE(format, ...) usb_logWrite('e',__FILE__,__LINE__,format
      ,##__VA_ARGS__)
2
3 #define LogD(format, ...) usb_logWrite('d',__FILE__,__LINE__,format
      ,##__VA_ ARGS__)
4
5 #define LogI(format, ...) usb_logWrite('i',__FILE__,__LINE__,format
```

```
    ,##__VA_ ARGS__)

#define LogW(format, ...) usb_logWrite('w',__FILE__,__LINE__,format
    ,##__VA_ ARGS__)

#define LogO(format, ...) usb_logWrite('o',__FILE__,__LINE__,format
    ,##__VA_ ARGS__)

extern int usb_logWrite(char level,char *fileName, int lineNum,
    const char * format, ...);
```

LogE(),LogW(),LogD(),LogO,LogI() 均由 usb_ logWrite() 函数来实现, usb_ logWrite() 函数实现真正的完整的协议封装和调用驱动发送的过程, usb_ logWrite() 完成协议头部信息的获取, 包括日志的级别, 发送该日志的进程号和进程名, 打印该日志的文件的文件名, 该日志在文件中所处的行号。并将这些信息封装在所定义的头部格式当中。最后将用户需要输出的信息放入协议的数据部分, 并添加结束标志, 然后调用驱动程序将该数据包发送出去。usb_ logWrite() 的部分关键代码如下所示:

```
int usb_logWrite(char level,char *fileName, int lineNum, const char
    * format, ...)
{
  ...

  struct timespec tp;
  struct tm timeBuffer;
  time_t nowSec;
  char datetime[64];
  clock_gettime(CLOCK_REALTIME,&tp);
  nowSec = tp.tv_sec;
  localtime_r(&nowSec,&timeBuffer);
  timeLen = strftime(datetime,64,"%Y/%m/%d %H:%M:%S",&timeBuffer);
  sprintf(datetime+timeLen,".%3.3ld",tp.tv_nsec/1000000L);

  _init_fd();
  if(fileName != NULL)
  {
    char *rf = strrchr(fileName, '/');
        if(rf != NULL) fileName = rf+1;
  }

  logWriteBuf[0]=0x03;
  logWriteBuf[1]=0x03;
  n = snprintf(&logWriteBuf[2],LOG_BUF_SIZE-2,"<L=%c;PN=%d;P=%s;F=%s
      ;N=%d;T=%s>",level,Id,name,fileName,lineNum,datetime);
  n+=2;
  va_list argList;
  va_start(argList,format);
  m = vsnprintf(logWriteBuf+n,LOG_BUF_SIZE-n,format,argList);
  va_end(argList);

  if(m <= 0)
  {
        m = snprintf(logWriteBuf+n,LOG_BUF_SIZE-n, "format error\n")
            ;
```

```
34      }
35      n += m;
36
37      if(n > LOG_BUF_SIZE-2) n = LOG_BUF_SIZE-2;
38      logWriteBuf[n++] = 0x04;
39      logWriteBuf[n++] = 0x04;
40
41      write(log_fd, logWriteBuf, n);
42      return n;
43  }
```

usb_ logWrite() 函数的实现在 RTP 模式和 task 模式之下是一样的，在 task 模式下只需包含 usb_ logWrite.h 头文件即可，在 RTP 模式下需要包含 usb_ logWrite.h 和 usb_ logWrite.c 两个文件。

4.5 Windows 下的日志分析工具

由于日志分析工具并不是我们本次论文的介绍重点，此处我们只介绍调试信息的接收部分协议的解析相关的内容，对于其他的部分不做详细介绍；日志分析工具为 Windows 下使用 QT 开发的界面程序，其结构如图 4-4所示。

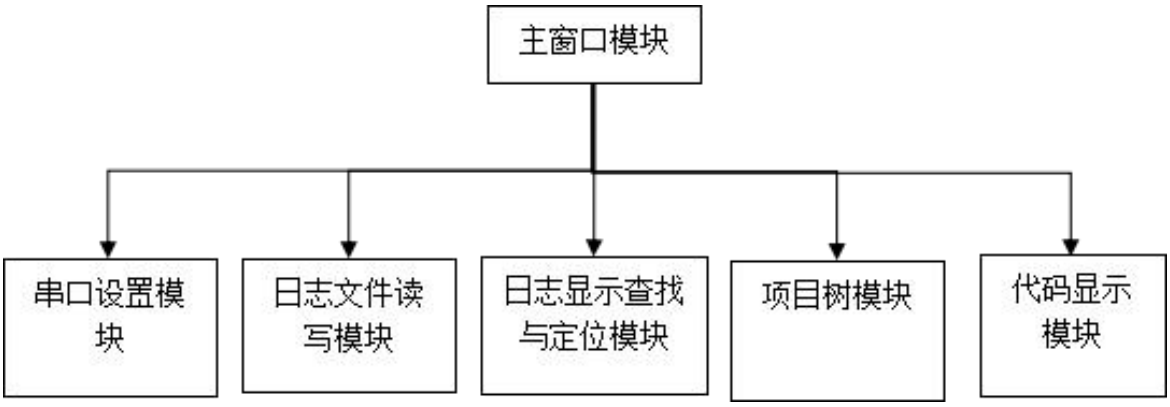


图 4-4　日志分析工具结构图

串口读取流程日志界面的串口读取流程图如串口读取流程图 所示，当用户打开串口后，主窗口中的串口读取函数会对串口中的数据进行非堵塞地读取，并按协议格式进行解析、显示和存盘。对于非协议格式的数据，则按照普通标准输出重定向过来的数据进行显示。此时在日志信息显示框中只会将信息显示在详细内容部分，日志级别、进程号、文件名、行号等内容均为空白。对于按照协议格式发送的信息，会按照信息的级别以不同的底色进行显示。

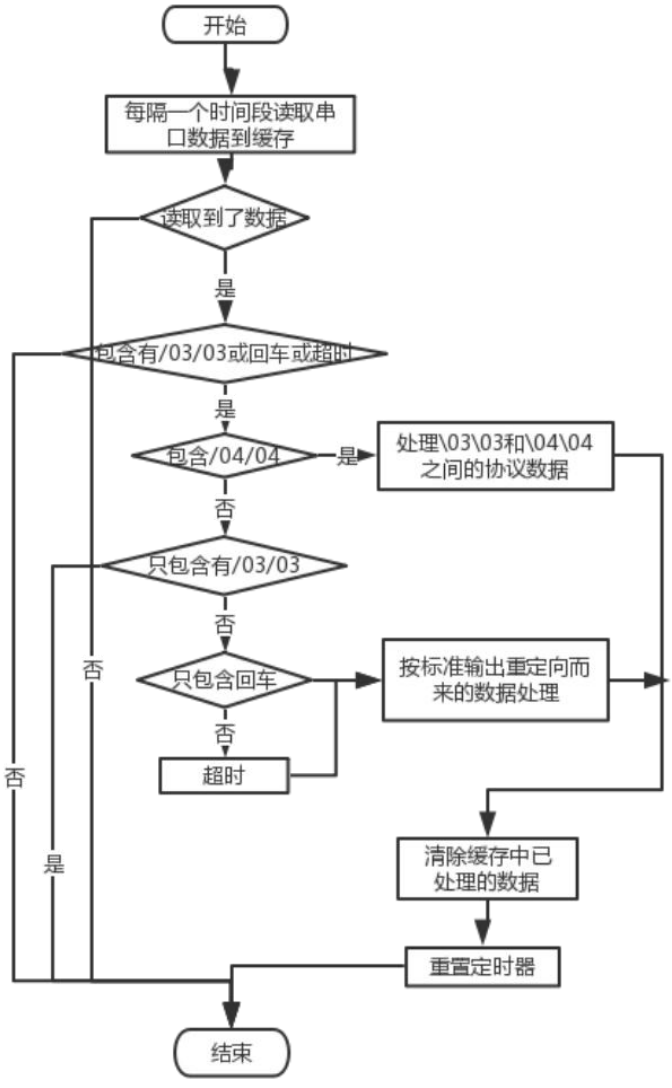


图 4-5　日志界面串口读取流程图

五　系统的功能测试

5.1　设备添加到系统管理表

查看当前的驱动程序是否已经添加到了系统当中，我们自定义的 cp210xDevRead()、cp210xDevWrite()、cp210xDevOpen()、cp210xDevClose()、cp210xDevIoctl() 是否已经和被注册到了 IO 子系统当中。只有成功的注册到了 IO 子系统当中，应用层的软件才能够通过标准接口 open()、close()、read()、write()、ioctl() 来对设备进行操作，完成设备的功能。首先通过 iosDrvShow 查看当前的 cp210xDrvInit 是否已经将驱动程序接口添加到了驱动程序列表当中，驱动加载前的系统设备表如图 5-1所示，驱动加载后的系统设备表如图 5-2所示，显示了加载了驱动驱动之后多出来了一个驱动号为 14 的设备。由于两种情况下进行为不同的设备命名策略，所以显示出两个不一样的设备名。

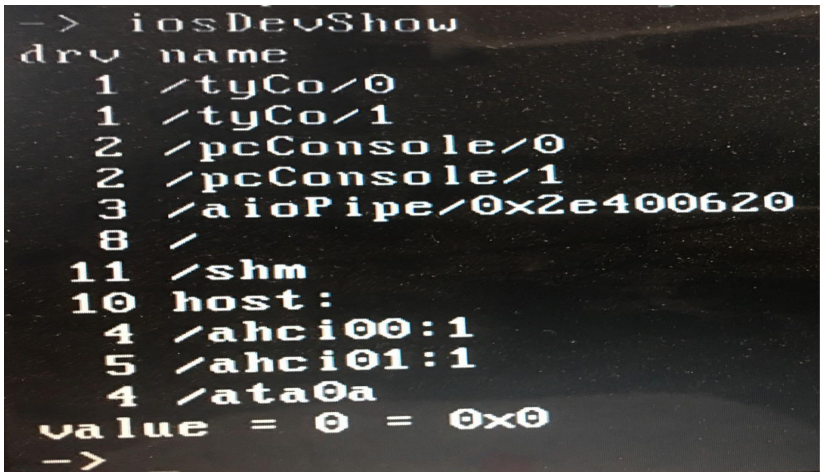
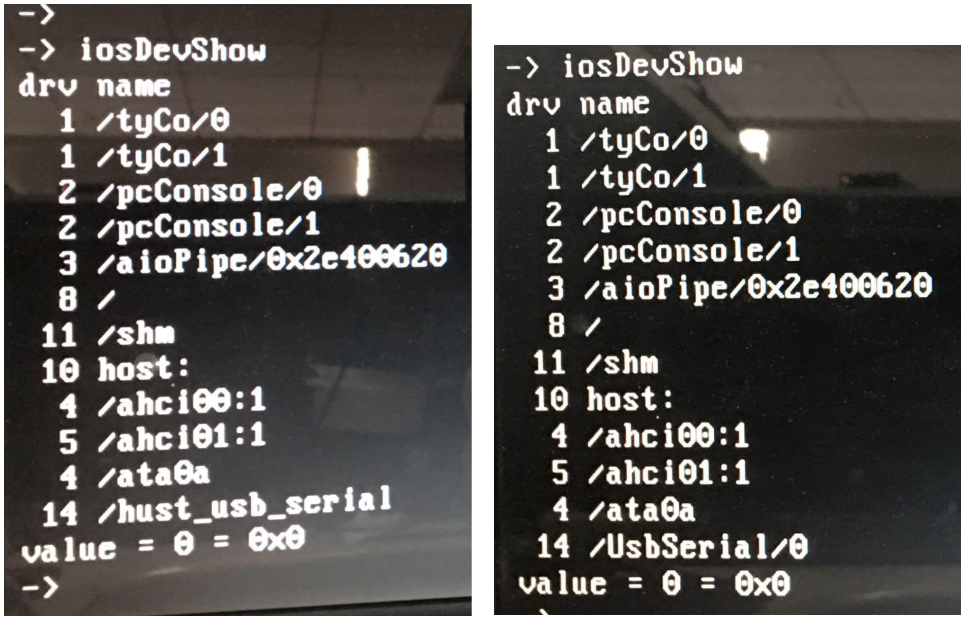


图 5-1　驱动加载前系统设备表

接下来查看设备驱动的接口是否已添加到驱动程序的列表当中，驱动加载前的系统驱动表如图 5-3所示，驱动加载后的系统设备如图 5-4所示。

最后查看一下加载驱动之后是否能够正常的打开设备，打开设备之后应该会将其加入到系统文件描述符表当中，驱动加载前的文件描述符表如图 5-5所示，驱动加载后的文件描述符表如图 5-6所示。

通过以上的验证说明了驱动程序已经被正常的安装，标准系统 IO 接口和驱动程序的 IO 子系统已经挂接成功。下一步将要进行进一步的功能测试，检测内部的各个 IO 接口是否能够正常的完成工作。



(a) 单设备驱动　　(b) 多设备驱动

图 5-2　驱动加载后系统设备表

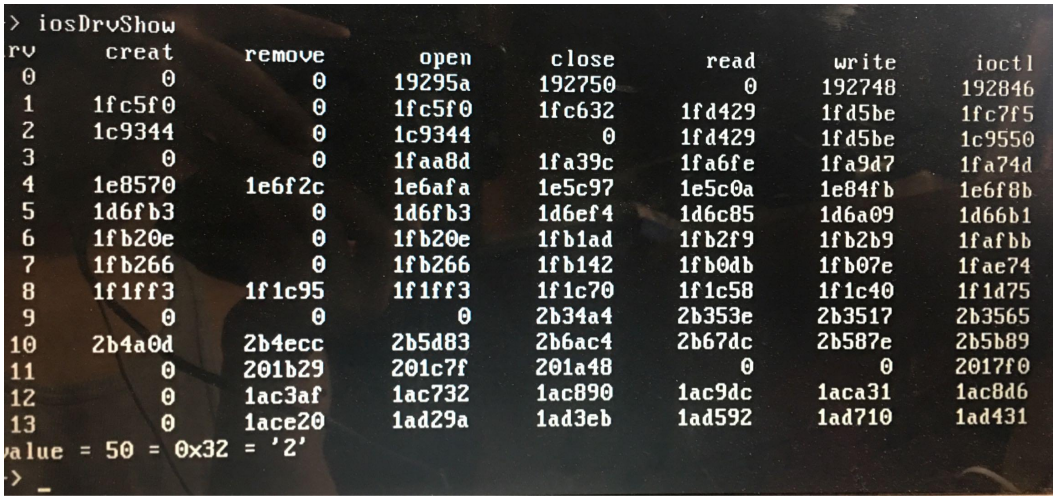


图 5-3　驱动加载前系统驱动表

5.2　驱动程序读写测试

驱动通常会作为操作系统内核的一部分来实现，即便现在很多系统支持驱动的动态加载，但是驱动代码在执行时，依然是以内核代码模式进行执行的，所以如果驱动程序存在 BUG，会导致操作系统的崩溃。所以调试驱动是一项十分关键的工作，必须对驱动进行仔细检查，并需要经受长时间运行考验，其调试过程同时也是对硬件和驱动进行验证的过程。由于 VxWorks 同时存在 RTP 模式和 task 模式，所以在这两种模式下都需要进行读写测试。测试条件: 装有我们编写的 USB 口转串口驱动的 VxWorks PC

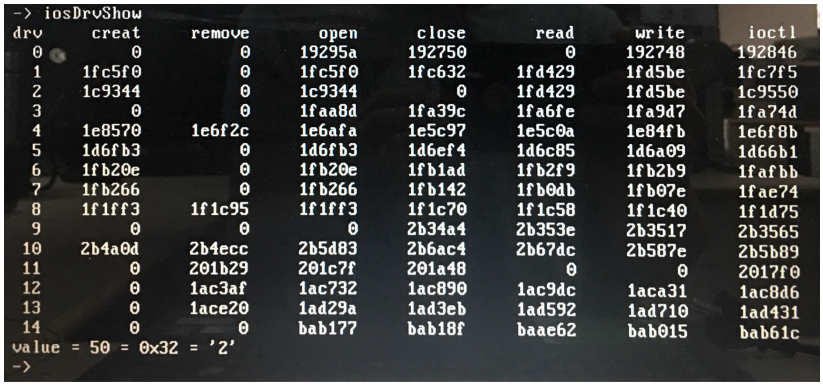


图 5-4　驱动加载系统驱动表

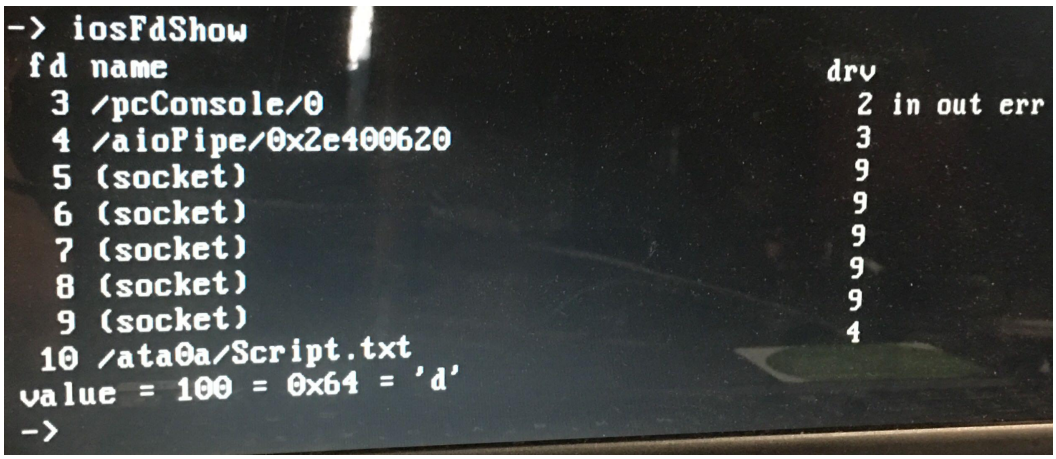


图 5-5　当前系统上的文件描述符表

机，装有串口调试工具的 windows PC 机，USB 转 TTL 模块在两台 PC 之间进行数据传输。

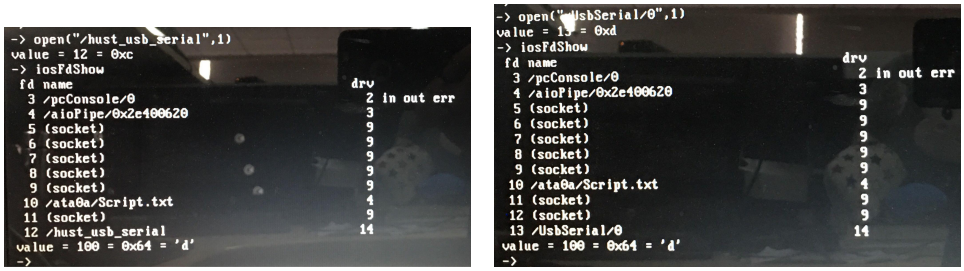


图 5-6　加载驱动后系统上的文件名文件描述符表

5.2.1　RTP 模式下的读写测试

测试报告如表 5.1所示。

波特率	数据位	数据位	数据方向	发送周期	发送次数	信息数量	正确率
9600	8	0	读	0.1S/次	10000	500 单词/次	100%
9600	8	0	写	0.1S/次	10000	500 单词/次	100%
9600	8	0	读 & 写	0.1S/次	10000	500 单词/次	100%
115200	8	0	读	0.1S/次	10000	500 单词/次	100%
115200	8	0	写	0.1S/次	10000	500 单词/次	100%
115200	8	0	读 & 写	0.1S/次	10000	500 单词/次	100%
921600	8	0	读	0.1S/次	10000	500 单词/次	99.8%
921600	8	0	写	0.1S/次	10000	500 单词/次	100%
921600	8	0	读 & 写	0.1S/次	10000	500 单词/次	99.2%

表 5.1　RTP 模式下串口测试

5.2.2　task 模式下的读写测试

测试报告如表 5.2所示。

在两种模式下都出现了在波特率较高的时候出现误码的现象,尤其是在同时进行数据的收发的时候出现误码的概率更大。会有很多的原因导致串口出现误码,例如干扰、接地不好、数据率过高、双方定时不一致等都会导致误码率的升高。

波特率	数据位	数据位	数据方向	发送周期	发送次数	信息数量	正确率
9600	8	0	读	0.1S/次	10000	500 单词/次	100%
9600	8	0	写	0.1S/次	10000	500 单词/次	100%
9600	8	0	读 & 写	0.1S/次	10000	500 单词/次	100%
115200	8	0	读	0.1S/次	10000	500 单词/次	100%
115200	8	0	写	0.1S/次	10000	500 单词/次	100%
115200	8	0	读 & 写	0.1S/次	10000	500 单词/次	100%
921600	8	0	读	0.1S/次	10000	500 单词/次	100%
921600	8	0	写	0.1S/次	10000	500 单词/次	100%
921600	8	0	读 & 写	0.1S/次	10000	500 单词/次	99.6%

表 5.2　task 模式下串口测试

5.3　应用程序接口测试

5.3.1　标准输出重定向测试

测试条件: 装有我们编写的 USB 口转串口驱动的 VxWorks PC 机, 封装好的标准输出重定向接口, 装有串口调试工具的 windows PC 机, USB 转 TTL 模块在两台 PC 之间进行数据传输。

测试同样分为 RTP 模式和 task 模式, 在两种模式下实现重定向的机制并不一样。1. task 模式:

在 VxWorks 中启动一个 task 程序, 先调用标准输出重定向接口, 查看其是否能够在程序中将标准输出重定向到串口. 再定向回来。循环进行下去, 查看输出是否正确。简单的测试程序如下:

```
void test1(void)
{
        int j =0;
        while(j < 10000){
                ResetStdOut(1);
                printf("<task printf test:> shold be in windows\n");
                ResetStdOut(0);
                printf("<task printf test:> shold be in VxWorks\n");
        }
}
```

测试结果显示重定向接口能够在 task 模式下完美的完成重定向的功能。

2. RTP 模式:

RTP 模式下的测试方式与在 task 模式下一样, 在 VxWorks 中启动一个 RTP 程序, 先

调用标准输出重定向接口，查看其是否能够在程序中将标准输出重定向到串口. 再定向回来，循环进行下去，查看输出是不是正确。简单的测试程序如下：

```
int main()
{
        int j =0;
        while(j < 10000){
                ResetStdOut(1);
                printf("<RTP printf test:> shold be in windows\n");
                ResetStdOut(0);
                printf("<RTP printf test:> shold be in VxWorks\n");
        }
}
```

注意这里的两个程序的区别，RTP 模式下是 main 函数开始的，而 task 模式下是以内核模块的方式运行的。测试结果显示 RTP 模式下标准输出重定向也能够正常的完成工作，不会对系统造成混乱。

5.3.2 Log 日志接口测试

Log 接口函数在 RTP 模式下和 task 模式下的实现方法是一样的，所以就不需要区分两种实现方式下的测试结果。我们以在 RTP 模式下的测试结果为例，测试完成的界面如图 5-7所示。在日志信息的显示框我们可以看到不同的颜色标识的不同级别的信息，包括我们自定义的协议的日志级别、进程号、进程名等信息都能正常的进行封装，并被正确的解析出来。

测试报告如表 5.3所示。

波特率	数据位	数据位	数据方向	发送次数	信息类型	正确率
115200	8	0	写	1000	LogE()	100%
115200	8	0	写	1000	LogW()	100%
115200	8	0	写	1000	LogI()	100%
921600	8	0	写	1000	LogO()	100%
921600	8	0	写	1000	LogD()	100%

表 5.3 Log 接口测试

需要注意的是上述的测试数据的是在 1000 次的循环内依次发送 LogE、LogW、LogI、LogO、LogD 的信息测试出来的，并不是每个级别的信息单独一个程序发送的。

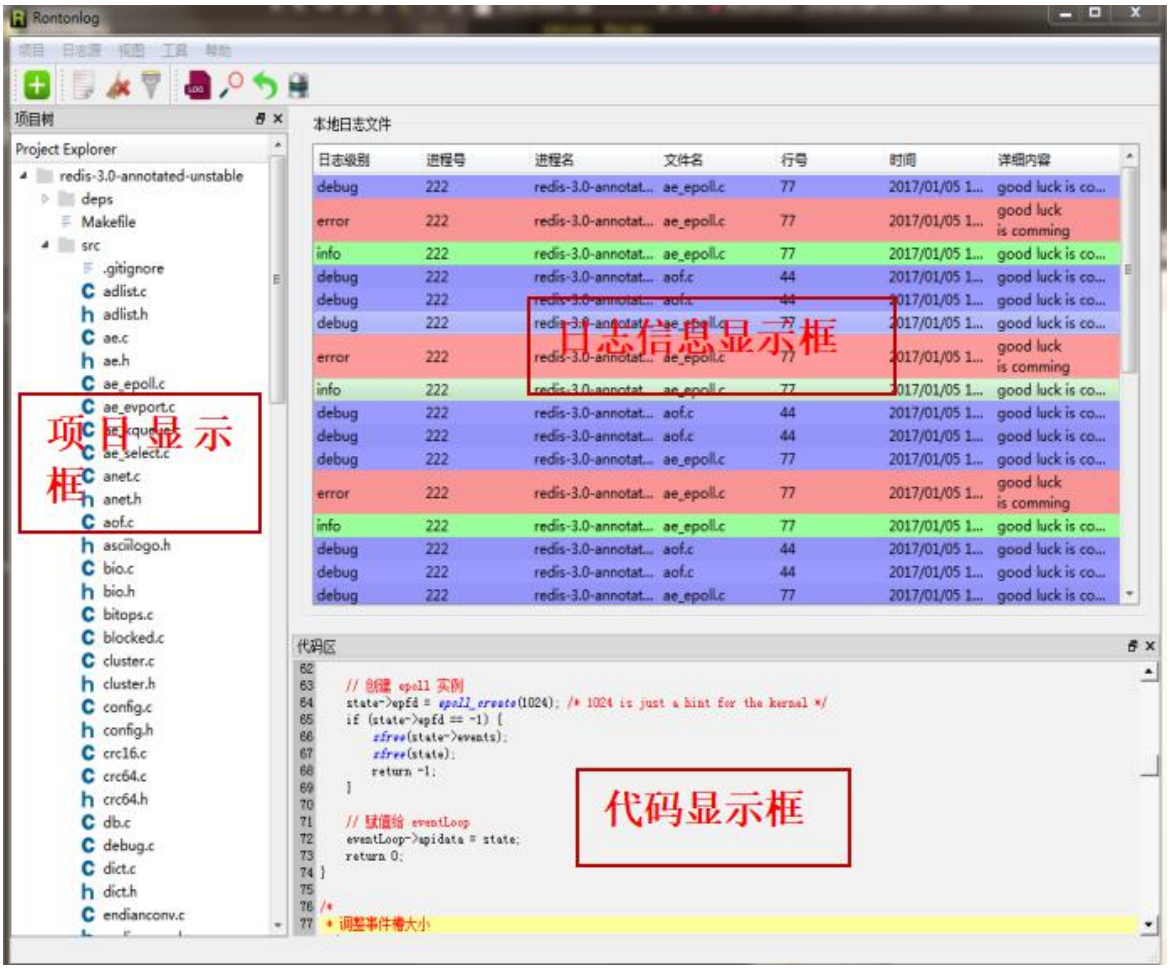


图 5-7　主机端日志分析工具测试界面

六　总结与展望

6.1　全文总结

USB 是一种新兴的外围接口标准，由于它众多的优良特性，且在现代 PC 中广泛使用，已经逐渐替代了其他的接口标准。本文选择了使用在 VxWorks 下使用 USB 口转串口的技术为基础，来实现一个调试通道。USB 口转串口的硬件实现选择了目前市场上通用的 CP2102 芯片，因为芯片有良好的说明文档，并有大量的社区支持，对于驱动程序的编写更加方便。基于这个芯片我们在 VxWorks 上实现了一个针对该芯片的 USB 口转串口驱动。

在本次的研究和开发的过程中，掌握了大量的关于 USB 接口和串口的相关知识，让我深深的体会到了 USB 驱动分层设计的好处与便利。要开发一个 USB 协议栈需要有很强的理论基础，明白各个层次之间的关系，上层应用程序与设备驱动之间的通信原理等。本次的设计是基于 VxWorks 操作系统之下的，之前从未接触过这类强实时性的嵌入式系统。在本次的开发过程中不断学习 VxWorks 系统的同时，也积累了大量的 VxWorks 下开发项目的实际经验。

6.2　展望

通过本次的 VxWorks 系统作为一个高性能的实时操作系统，成功的应用于很多大型的高尖端项目当中。在本次的发开过程当中我也只是学习到了 VxWorks 的冰山一角，对于其系统理论和实际应用的掌握还不全面。同时感受到了 USB 功能的强大和开发难度。但是其优异的性能一定能够使得其在今后的外围串行通信接口当中占有更加重要的地位。随着 PC 和消费电子类产品、数字化产品的不断增多和普及，用户对于数据带宽、数据存储容量、数据传输速率和使用等会有越来越高的要求，鉴于此，USB 技术也在不断地发展和完善，USB OTG、type C 等标准的颁布，更能使得 USB 的应用领域和实用场景强化。因此，研究 USB 协议和开发一些特定功能的 USB 接口将会成为应用 USB 技术的关键。

至此，本次设计的调试通道的功能已经基本实现，能够满足实际的应用中的需求。由于时间和能力有限，对于本次的调试通道的设计还有很多的不足之处，对于 VxWorks 下的 USB 口转串口的驱动程序部分还有很多的可以改进、完善的部分。例如在本系统中没有完成对设备的流控的设置，因为串口的传输速率有限，远远小于 USB 口的传输速率，使得串口速率成为了调试通道中传输速率的瓶颈部分，也许可以选择更好的

数据传输方式来设计此通道。

因为时间的关系,未能将研究工作进行的更加深入,对于不足之处深表遗憾。此外,论文当中难免会有考虑不周之处,恳请老师、同学批评指正。

致 谢

研究生生活即将结束,在此,我要感谢所有教导我的老师和陪伴我一齐成长的同学,他们在我的研究生生涯给予了很大的帮助。本论文能够顺利完成,要特别感谢我的导师张杰老师,张杰老师对该论文从选题,构思到最后定稿的各个环节给予细心指引与教导,正是在张老师的指导和帮助下,我才能够克服各种困难,突破一个又一个的技术难题,最终完成该调试通道的设计,使我得以最终完成毕业论文设计!

在硕士学位论文即将完成之际,我想向身边曾经给我帮助和支持的人们表示最忠心的感谢。首先要感谢的就是我的导师张杰老师,在我的研究生生活的第一天起便给予我许多的帮助。老师对我的严格要求使我始终保持着一丝不苟的态度对待每一次研究和身边发生的每一件事,他自身严谨的治学态度以及为人处世的坦荡都让我钦佩不已,也是我一直向之前进的目标,正是导师的悉心培养才有了今天的我和我的一些研究成果。老师不仅在科研方面给我很多的灵感与帮助,让我在研究的道路上走得很顺畅;同时也教会了我很多关于人生的道理,让我有所成长。我与老师的关系亦师亦友,老师总会是我们最坚实的后盾,让我们能够放心的往前飞。相信在今后的工作与生活中,我都会不断的记起老师的谆谆教诲,也能够因此少走很多弯路,更好的为社会、为国家做出更大的贡献。在此祝愿我的导师身体健康,幸福欢乐!当然我也要感谢许多已经毕业或仍在深造的师哥师姐,他们也给予了我许多的帮助以及支持鼓励。在我刚入门时耐心指导我,不厌其烦的解答我的疑问;也在我能够独当一面时用他们的经验提醒我可能会不注意的一些小细节。于是现在的我也能够很好的为师弟师妹们答疑解惑,是他们给我的帮助和在他们身上看到的善意,使我以更加友善柔和的目光去看待这个看似冷酷实则温柔的世界。

最后我还要感谢我的战友们,我们一起攻克研究过程中遇到的重重难关,在最艰难的时候互相鼓励,互帮互助。在这看似枯燥的道路上,总是能汲取出甘甜的也是我同甘苦的战友们。

参考文献

[1] 马超, 尹长青. VxWorks 嵌入式实时操作系统的结构研究. 电脑知识与技术: 学术交流, 2006, (1):133–134.

[2] 徐媛媛. 嵌入式实时操作系统的设备驱动: [PhD Dissertation]. 华中科技大学, 2003.

[3] 孔祥营. 嵌入式实时操作系统 VxWorks 及其开发环境 Tornado. 中国电力出版社, 2002.

[4] System W R. Tronado Device Driver Workshop. Inc, 1999.

[5] System W R. Tronado BSP Training Workshop. Inc, 1999.

[6] System W R. VxWorks BSP Developer Guide. Inc, 2004.

[7] 曹桂平. VxWorks 设备驱动开发详解. 电子工业出版社, 2011.

[8] 罗国庆. VxWorks 与嵌入式软件开发. 机械工业出版社, 2003.

[9] System W R. VxWorks Driver API Reference 5.5. Inc, 2002.

[10] System W R. VxWorks Programmer Guide. Inc, 2003.

[11] 王金刚. 基于 VxWorks 的嵌入式实时系统设计. 清华大学出版社, 2004.

[12] 史小斌, 孙献璞, 张艳玲. VxWorks 串行设备驱动模式及其实现. 现代电子技术, 2003, (10):72–74.

[13] 张念淮, 江浩. USB 总线接口开发指南. 国防工业出版社, 2001.

[14] Axelson, 陈逸 J. USB 大全. 中国电力出版社, 2001.

[15] 刘荣. 圈圈教你玩 USB. 北京航空航天大学出版社, 2009.

[16] 何源, 顾金良. USB 与 RS232 接口转换器的设计. 指挥控制与仿真, 2006, 28(5):114–117.

[17] Labs S. CP2102 SINGLE-CHIP USB to UART BRIDGE, 2007.

[18] Renard K G, Nichols M H, Woolhiser D A, et al. 1003.1-2008 - IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7. 2008, c1-3826.

[19] Wu Y, Wang C, Cai J. Implementation of IPTV STB Network Driver and Protocol for VxWorks Embedded System. in: Proceedings of International Symposium on Test Automation Instrumentation, 2008.

[20] Zhang Z, Li Y, Zhang Z. Design and implementation of control system software based on VxWorks multi-tasks. in: Proceedings of International Conference on Advanced Computer Theory and Engineering, 2010, V2-285 - V2-288.

[21] Yi-Hua F U, Mei S L. Implementation of 100Mbps Ethernet based on VxWorks in embedded system. Journal of Computer Applications, 2006, 26(9):2190–2191.

[22] 肖竟华, 邱奕敏. 嵌入式实时操作系统 VxWorks 设备驱动程序设计与实现. 电脑与信息技术, 2003, (5):28–30.

[23] 李立志, 张朝阳, 陈文正. 实时操作系统 VxWorks 设备驱动程序的编写. 计算机工程, 2003, 29(4):182–184.

[24] Ghosh K, Mukherjee B, Schwan K. A Survey of Real-Time Operating Systems – Draft. Georgia Institute of Technology, 1993.

[25] 晋成凤, 晋成龙. 基于虚拟串口和串口调试工具的串口编程调试系统设计. 2010.

[26] Meirong X, Ming C, Jinxiang D. Design and Implementation of CAN Driver Based on Real-Time Operation System VxWorks. Computer Application and Rearch, 2006, 23(5):185–188.

[27] System W R. USB Developer's Kit Programmer's Guide 1.1.2. Inc, 2003.

[28] System W R. VxWorks Network Programmer's Guide 5.4. Inc, 2003.
[29] 萧世文, 宋延清. USB 2.0 硬件设计. 清华大学出版社, 2006.
[30] 李肇庆, 韩涛. 串行端口技术. 国防工业出版社, 2004.