

分 类 号 _____

学号 _____ M201672905 _____

学校代码 10487

密级 _____

华中科技大学
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 5 月 27 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 : Lecturer Jie Zhang

Huazhong University of Science & Technology

Wuhan 430074, P. R. China

May 27, 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 保密口,在 ____ 年解密后适用本授权书。
 不保密口。
(请在以上方框内打“√”)

学位论文作者签名:

日期: 年 月 日

指导教师签名:

日期: 年 月 日

摘 要

在嵌入式软件的开发、移植、运行时都需要输出一些调试或日志信息，目前 VxWorks 当中使用 WorkBench 集成开发平台来完成软件的开发、调试工作，它的调试工具使用的是一种典型的在线调试方式，但是在我们项目实际生产环境中希望能够使用一种离线的调试日志的方式来进行程序的调试工作，他们希望在设备运行的大量应用程序当中加入调试信息，这些调试信息都能够自动的传输到普通的 PC 上，之后他们可以查看这些调试信息，对这些信息进行分析。

本文从项目的需求出发，详细地了解了他们所使用的设备的情况和具体的需求，给出了一个总体的设计方案，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，在项目所处的实际环境当中也没有可供使用的串口，因此在调试信息的传输过程中只能够使用 USB 接口，本设计即以 USB 转串口驱动为基础，并配以我们设计的调试接口组成一个实际可用的调试通道，方便它们在程序的开发和移植的过程中进行离线的调试。

根据我们的方案需求，我们分析了 VxWorks 下的设备驱动所需的关键技术，然后在 VxWorks 下设计出了一个特定需求的 USB 转串口的驱动程序和一个普通的 USB 转串口驱动程序，驱动程序具有双向缓冲功能，同时具有即插即用、使用方便等特点。然后我们给上层需要输出调试信息的应用设计了两个调试接口。一个 Log 调试接口，我们为这个接口设计了一个自定义的调试通道的传输协议；另外我们也给出了一个标准输出重定向的调试接口，方便开发人员直接使用标准输出进行调试信息的输出。最后我们通过对系统的整体测试和对各个部分分别进行测试，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： VxWorks 操作系统，设备驱动开发，调试，USB 转串口

Abstract

In the development, migration, and runtime of embedded software, some debugging or log information needs to be output. At present, Workbench integrated development platform is used in VxWorks to complete software development and debugging. Its debugging tool uses a typical online debugging. The way, but in the actual production environment of our project, we hope to use an offline debug log to debug the program. They want to add debugging information to a large number of applications running on the device. These debugging information can be automatic. It is transmitted to a normal PC, after which they can view the debug information and analyze the information.

This article starts from the actual needs of the project, understands in detail the conditions and specific requirements of the equipment they use, and gives an overall design proposal. Since most of the current devices no longer use RS-232 serial ports as The necessary standard interface, but a large number of use of the USB interface, there is no serial port available for use in the actual environment of the project, so only the USB interface can be used during the transmission of debugging information, this design is to USB to serial port Based on the driver, and equipped with the debugging interface we designed to form a practical and available debugging channel, they are convenient for offline debugging during program development and porting.

According to our solution requirements, we analyzed the key technologies needed for device drivers under VxWorks, and then designed a specific USB-to-serial driver and a generic USB to serial driver under VxWorks. Two-way buffer function, with plug and play, easy to use and so on. Then we design two debugging interfaces for the application that needs to output debugging information. A Log debug interface, we have designed a custom debug channel transfer protocol for this interface; in addition we also provide a standard output redirection debug interface to facilitate developers to directly use the standard output for debugging information output. Finally, we test the whole system and test each part separately to verify that the debug channel design is basically completed, the system is running reliably, and the required requirements are fulfilled.

Key words: VxWorks operating system, device driver development,debugging, USB to serial

目 录

摘要	I
1 绪论	1
1.1 课题背景以及来源	1
1.2 国内外概况	2
1.3 论文的主要工作和组织结构	3
2 调试通道总体设计与关键技术	5
2.1 总体设计	5
2.2 关键技术	7
2.3 本章小结	15
3 驱动程序的设计和实现	16
3.1 VxWorks 上的 USB 开发	16
3.2 CP2102 开发	18
3.3 特定需求单设备驱动的实现	21
3.4 通用多设备驱动的实现	35
3.5 小结	39
4 应用层程序接口封装	40
4.1 应用层接口模块设计	40
4.2 Log 协议的设计	40
4.3 标准输出重定向接口的实现	41
4.4 Log 接口的实现	43
4.5 Windows 下的日志分析工具	44
4.6 小结	45
5 系统的功能测试	46
5.1 设备添加到系统管理表	46
5.2 驱动程序读写测试	47
5.3 应用程序接口测试	50
5.4 小结	52

6 总结与展望	53
致谢	54
参考文献	55

— 绪论

1.1 课题背景以及来源

嵌入式系统目前不断发展和壮大, 使用嵌入式的场景越来越多, 同时对嵌入式系统进行软件开发和移植工作也越来越多, 由于每种嵌入式系统都有不同的软件接口和特性。所以在不同的嵌入式系统上进行软件开发和移植工作要根据其特性使用不同的方法, 在我们所熟悉的通用操作系统 (Linux、Windows、MacOS) 下进行软件设计和开发时它们通常都会有现成的集成开发环境和调试工具。然而在嵌入式软件的开发和移植过程当中, 由于嵌入式设备所独有的特点, 对嵌入式系统上的软件的调试、分析一直是一个费时费力的工作。在整个的嵌入式的开发过程当中软件的调试占据了软件开发周期中的大部分时间。这是因为在设计软件时难免会出现各种各样的错误, 这些错误可能需要经过反复的修改才能够达到设计的要求。因此一个好的调试、分析工具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, 软件运行过程中调试信息的定位等工作。

VxWorks 因为其可靠性和实时性被广泛地应用对系统的实时性要求很高的领域当中, 如: 通信、航天、军事等^[1], 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, 在程序的正常运行当中也需要输出一些日志信息, 方便之后对程序运行过程中产生的问题进行具体的分析。目前 VxWorks 当中使用 WorkBench 集成开发平台来完成软件的开发、调试工作, 这是一种典型的在线调试方式^{[2][3]}, 但是在我们项目 的实际生产环境当中他们希望使用一种离线的调试日志的方式来进行程序的调试工作, 他们希望在设备运行的大量应用程序当中加入调试信息, 这些调试信息都能够自动的传输到普通的 PC 上, 他们能够在之后查看其调试信息, 对这些信息分析工作。并且通常对于宿主机和目标机之间的传输都是通过网口或者串口, 然而目前的大多设备都已不再将 RS-232 的串口作为必须的标准接口, 而是大量的使用 USB 接口, 嵌入式设备上又大多没有配备网口的需求, 因此在调试信息的传输过程中只能够使用 USB 接口。

本次基于 VxWorks 的调试通道的设计来源于项目 的实际需求, 我们基于 VxWorks 开发出一个小巧易用的调试信息的传输通道, 方便它们将应用程序的调试信息传输出来, 以进行一个事后的分析工作, 对于底层的信息传输我们使用 USB 转串口来实现, USB 转串口的转换器使用 CP2102 模块来实现, 并在此基础上设计了给应用层使用的调试接口。

1.2 国内外概况

在嵌入式系统上由于其资源有限,所以在嵌入式系统上的调试必须采用交叉调试的方式。用户无法直接在嵌入式系统平台上调试被调试的程序,因此必须要借助于宿主机上丰富的调试资源通过一定的调试通道配合目标机共同完成对被调试程序执行状态的实时跟踪,从而快速有效地对程序错误进行定位,纠正错误,提高调试速率和软件质量。

目前嵌入式系统上主要使用三类的调试技术进行调试。

1. 目标监控程序调试技术

目标监控程序调试技术是在目标机中植入一段特殊的代码即监控程序来实现对目标机的调试控制宿主机和目标机之间可以仅仅通过简单的通讯设备(如网口,串口等进行连接),不需要额外的硬件开销,同时在目标机上也不需要特殊的硬件支持。目标监控程序调试技术又可以根据监控程序的实现方式不同而分为两种:一种是作为一个独立的程序运行在目标机上,如 GDBServer, ROM Monitor, 这种方式下的监控程序主要是作为一个操作系统上的应用程序用于用户应用程序的调试。若要实现对操作系统进行调试,则需要同时具备完成简单的硬件初始化、下载以及调试被调试程序的功能;开发难度较大。另外一种是监控程序编译进操作系统镜像之后一起在目标机上运行,如 GDB Stub、VxWorks 的 target agent、Windows CE 的 KITL 调试技术等等,这种监控方式可以直接调用操作系统中的可用代码(如设备驱动),减少代码的开发量,还可以方便的进行接口功能扩展,丰富调试通道,解决了接口资源受限目标机的调试问题,同时监控程序一般都可以同时用于操作系统以及用户应用程序的调试。

2. 片上调试技术

片上调试技术是随着芯片技术发展特别是处理器的封装越来越表贴化,加大了 ICE 的仿真探头开发难度的情况下发展起来的,是通过在目标机处理内部集成调试模块来接管被调试程序的运行控制完成调试操作的。片上调试技术使用两级模式:运行模式和调试模式,在调试模式下通过目标机上的提供的调试端口同宿主机交互信息完成调试功能。根据芯片调试接口支持不同,常用的片上调试技术包括:BDM 调试技术。BDM 也叫后天调试适配器。BDM 调试是利用处理器内部提供的 BDM,增加一个可以插入后台模式指令激活后台处理的串行口作为在线仿真接口,在定义的执行点上,由在线仿真器将后台调试指令覆盖在原代码上,控制目标处理器的运行和停止。

3. 利用打印信息进行调试

利用打印信息进行调试是最一般的调试技术,这种技术也称为监视,只需要应用程序内部适当的地方加上 printf 调用即可。这样就可以完成对应用程序中相关变量的监视工作,进而推断出错误所在。这种调试方式简单方便,不需要复杂的设置,但是缺点是获取到的信息有限且不能进行断点设置,单步执行等操作。

在这些调试技术领域中,宿主机和目标机之间的通信方式主要有串口方式、以太

网接口,大多数的远程调试中使用的是串口传输方式,但是串口通信存在着速度慢、通信距离受限等弊端,而以太网口的传输方式则可以克服串口方式的不足,不仅可以提供稳定可靠的数据传输,而且无论是传输速度还是传输距离都远远优于串口方式,是一种快速高效的通信方式,但是网络传输需要网络芯片的支持,而很多的嵌入式设备并没有这种需求,所以很多设备上无法实现网络传输的范式来进行调试,目前 USB 已经成为嵌入式平台的通用接口,使用 USB 做为传输方式已经广泛的应用在嵌入式系统的开发当中,逐渐替代了 RS-232 接口,USB 不仅可以实现直接传输,还可以使用 CDC 协议虚拟成其他通信设备进行传输,USB 在调试技术方面的使用也有相关的研究,如 GDB 中的 USB Network 调试的实现,但是 USB 应用在这方面的资料相对较少,因此对其进行研究与实现具有重要的意义。

对于 USB 口转串口的转换器,国内外通常都会采用两种方案:一种是以 CY7C68013 芯片为代表,自己从底层的硬件和固件开始,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^{[4][5]}。因此我们在此会选择 CP2102 芯片来进行调试通道的设计。

1.3 论文的主要工作和组织结构

主要工作: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:一个将 USB 总线技术和 RS-232 接口相结合,设计出一个满足特定要求的、实用的 USB 转串口驱动程序;另一个是设计给应用层调用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。

本文共分为六章来进行描述,对每一个章节我们做了如下的安排:

第一章为绪论部分,主要描述了本次的课题的背景和来源、国内外的发展状况以及本文的结构安排。

第二章介绍了首先介绍对于我们的调试通道的总体设计,然后介绍了调试通道的开发所需要了解的系统知识和关键技术,主要包括 VxWorks 系统及驱动开发的知识、USB 技术相关知识。

第三章介绍了 USB 口转串口驱动程序的设计和实现,包括驱动程序程序当中对于缓冲区和信号量的设计,我们使用 CP2102 模块开发和 VxWorks 下的 USB 开发的内容,然后给出了 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,我们进行了整体测试和各个部分的功能测试。

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

在嵌入式软件的开发、移植、运行时都需要输出一些调试或日志信息，目前 VxWorks 当中使用 WorkBench 集成开发平台来完成软件的开发、调试工作，它所使用的调试方式是一种典型的在线调试方式，但是在项目实际生产环境当中他们希望使用一种离线的调试日志的方式来进行程序的调试工作，它们希望在设备运行的大量应用程序当中加入调试信息，这些程序的调试信息都能够自动的传输到普通的 PC 上，他们能够在之后查看其调试信息，对这些信息分析工作。

基于以上需求，此次我们的工作就是在 VxWorks 下完成一个能够满足他们的需求的调试通道，在主机端我们使用一个分析程序来对目标机上传输过来的调试数据进行分析；在目标机端我们给应用程序提供一个调试接口，将调试信息从通信接口中传输出去，本次设计的调试通道主要应用于 VxWorks 下程序移植时大量调试信息的输出的场景。对于在该应用场景下的整个调试通道的框架如图 2-1 所示。

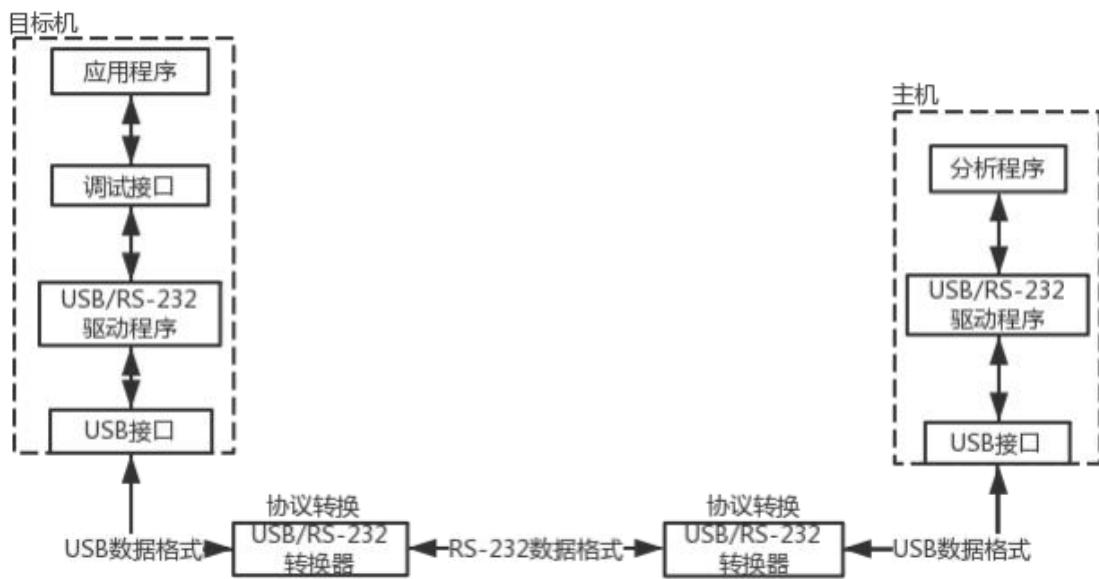


图 2-1 调试通道框架图

在我们所设计的调试通道当中，目标机上的上层应用程序会调用调试通道中所提供的调试接口，通过调试接口将调试信息传递给通信接口，由于设备上没有可用的 RS-232 串口，而 USB 的体系结构又是主从式的，无法直接作为调试通道的通信接口来使用，为了解决这个问题，在我们的调试通道当中使用外部的 USB/RS-232 的转

换器来实现通信接口，同时在系统上使用与设备相对应的 USB/RS-232 设备驱动程序。这样 PC 端的应用软件应用软件仍然是针对 RS-232 串行端口编程的，外设也是以 RS-232 为数据通信通道的，但是 PC 端到外设之间的物理连接却是 USB 总线，其上的数据格式也是 USB 数据格式。

在 VXWorks 中并没有现成可用的 USB 转串口驱动以及 USB 转串口转换器，所以我们需要自己选择一个外部的 USB/RS-232 转换器，以及针对该 USB/RS-232 转换器的 USB 转串口驱动。在主机端都会有已经实现好的 USB 转串口驱动程序，我们只需要提供转换器即可，这样用户在不需要理会 USB 的复杂的内部协议的情况下享受 USB 接口的即插即用、数据可靠传输、扩展方便等优点。分析程序会自动接收串口的数据，完成协议分析、调试信息的定位等工作。

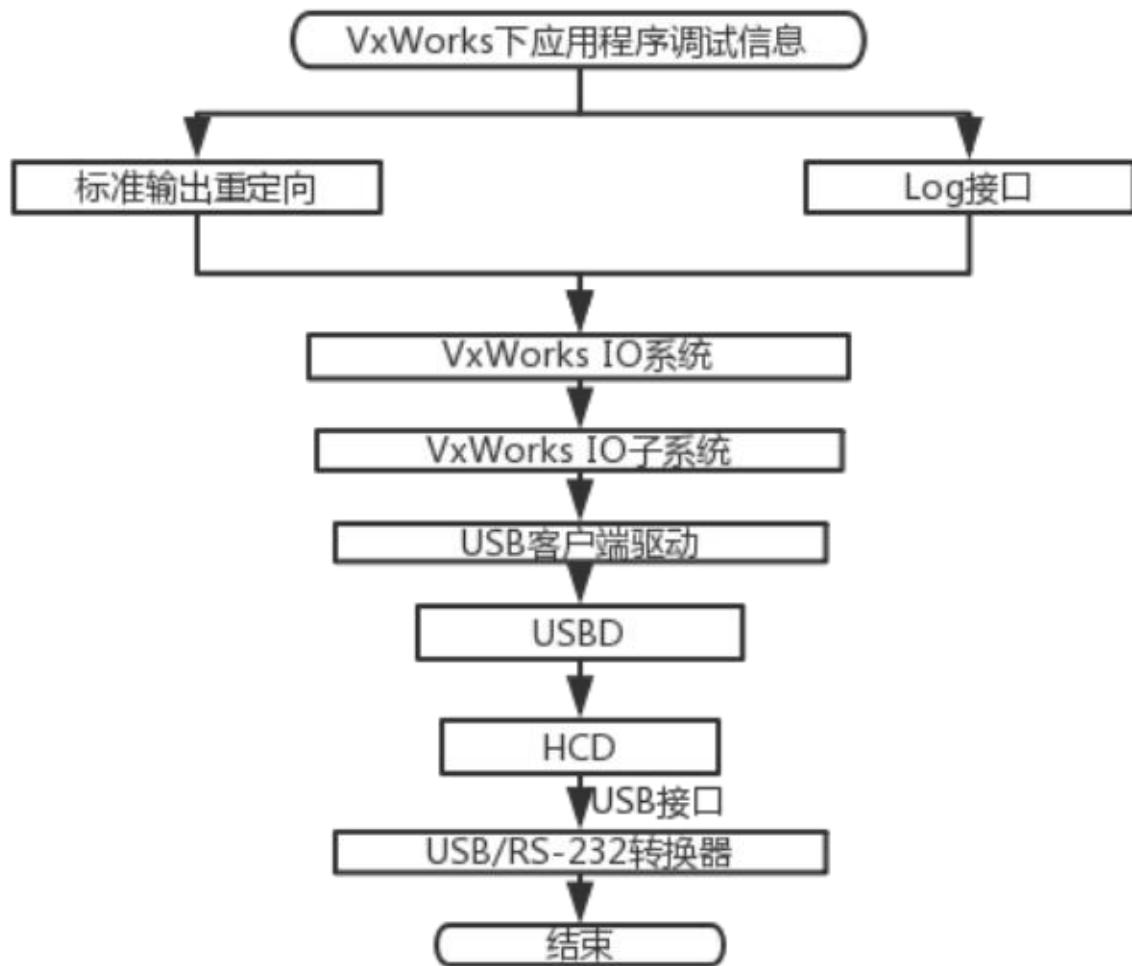


图 2-2 目标机调试通道层次结构图

对于目标机上的调试通道的结构如图 2-2 所示。在 VxWorks 当中 I/O 系统和 I/O 子系统是作为操作系统的重要组成部分已经实现好了的，我们的调试通道所需要设计的部分主要是接口模块和 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,为此我们设计了一个自定义的 Log 协议格式,其中的内容包含有调试级别、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;重定向的输出包括 RTP 模式下的重定向和 task 模式下的重定向,VxWorks 中对于这两种模式需要使用不同的重定向方式。

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,包括一个特定需求的驱动程序的实现和一个普通的驱动程序的实现。对于本次特殊需求的驱动程序相对于普通的驱动程序而言在流程和结构上进行了修改,以使其达到特定的要求,具体的实现我们会在第三节进行介绍。两种实现方式中都会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 子系统在 VxWorks 当中的作用是维护系统设备表、系统驱动表、系统文件描述符表^{[6][7]}。设备驱动在 VxWorks 中就靠这三个数据结构来进行管理,所以对于设备驱动而言非常重要。设备驱动程序初始化时会对硬件完成初始化的配置,同时会向 I/O 子系统注册自己,注册之后 I/O 子系统才能找到该驱动。

2.2.1.1 VxWorks I/O 系统

通常操作系统为了应用程序的平台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有 creat()、open()、unlink()、remove()、close()、rename()、read()、write()、ioctl()、lseek()、readv()、writev() 等^{[2][8][9]},我们通常将其称作为标准 I/O 库函数。使用库函数可以在对应用层的程序进行开发和移植的时候使用同一套接口,这给应用程序的编程人员带来很大的方便,大大提高了其开发效率,避免了重复工作,而对于操作系统而言它需要通过调整底层驱动或者操作系统中间层来为标准 I/O 接口的实现提供支持。

在 Mac OS、Linux、或 Windows 当中会把这套接口以标准库的形式呈现,但是在 VxWorks 中它们是由系统的内核实现的,直接以内核文件的形式提供的,它们都位于 ioLib.c 文件下^[6]。之所以是以内核文件的形式来提供,是因为 VxWorks 当中不会区分用户态和内核态,这是 VxWorks 与通用操作系统的一个很大的不同点。在 Vxworks

当中所有的内核函数都可以不加限制的由用户程序直接进行调用,这样就减少了中断转入内核态再继续执行这一个过程,这对于强实时性系统而言极为重要,因为这样减少了时间开销,但是这样实现的一个缺点是减少了系统使用权限上的限制,为内核带来了很大的不安全性,极易导致内核的崩溃,所以 VxWorks 系统上应用程序的开发和使用对开发人员的要求比较高。

VxWorks 中 I/O 调用结构如图 2-3 所示。

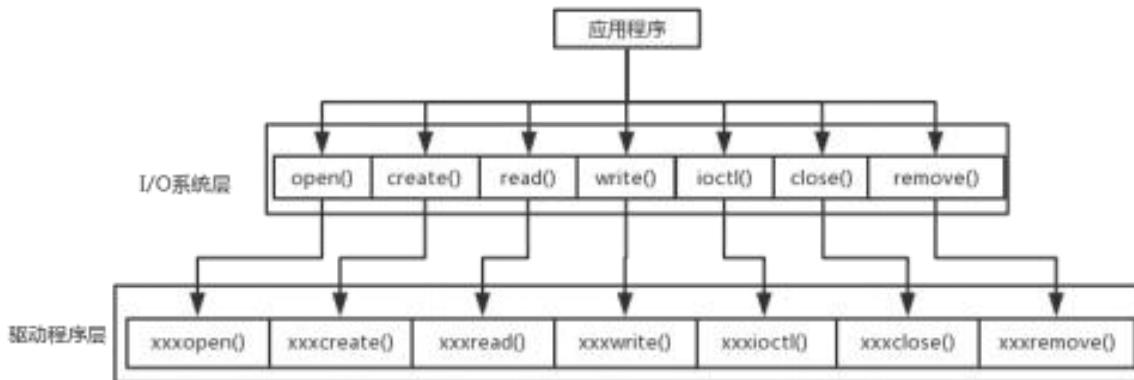


图 2-3 I/O 调用

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,系统设备表中每一个节点都是一个 DEV_HDR 类型的结构体,系统会将每个设备 DEV_HDR 连接在如图 2-4 所示的系统设备表中。DEV_HDR 是 wind 内核规定的每一个设备都必须要具有的一个数据结构,且必须是设备自定义结构的第一个成员,之后系统只会使用这个结构来代表该设备。DEV_HDR 结构体当中只包含有三个成员:一个设备链表节点;一个设备驱动号;一个指向设备名的指针。其定义如图 2-5 所示。

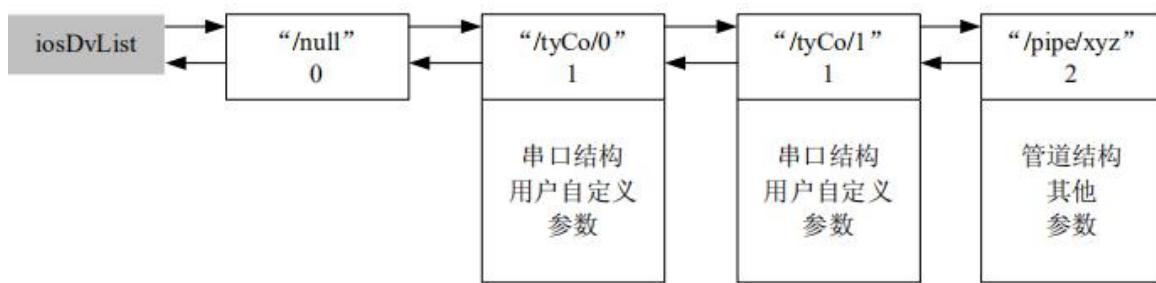


图 2-4 VxWorks 系统设备示意图

同时 VxWorks 系统提供了一个设备的注册函数 iosDevAdd(DEV_HDR *pDevHdr, char *name, int drvnum), 该函数用来将一个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在表中增加一个节点表示该设备,删除设备时就会将该设

```

1  /*h/iosLib.h*/
2  typedef struct
3  {
4      DL_NODE node;
5      short drvNum;
6      char * name;
7  }DEV_HDR; /* 这个结构是所有设备自定义结构体的第一个成员 */

```

图 2-5 DEV_HDR 结构体

备的节点从表中删除,一个设备添加到系统之后,就可以使用 open() 函数对其进行操作,open() 会通过将传递过来的设备名与系统设备表当中进行设备名匹配来完成设备的打开操作,匹配的原则是最佳匹配,匹配成功之后就可以实现文件与设备的连接^[1],之后就可以使用相对应的注册的设备驱动进行其他的文件操作。

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程序,既可以是直接驱动硬件工作的驱动程序,也可以是注册到 I/O 子系统下的驱动中间层^[6]。在 VxWorks 中系统驱动表的底层实现是一个数组,数组中的每一个元素就是一个系统驱动表的表项,每一个表项都是一个 DRV_ENTRY 类型的结构体,该结构定义在内核的头文件 iosLibP.h 当中,其定义如图 2-6 所示。

```

1  typedef struct
2  {
3      FUNCPTR de_create; /* 指向驱动的 create() 函数 */
4      FUNCPTR de_delete; /* 指向驱动的 delete() 函数 */
5      FUNCPTR de_open; /* 指向驱动的 open() 函数 */
6      FUNCPTR de_close; /* 指向驱动的 close() 函数 */
7      FUNCPTR de_read; /* 指向驱动的 read() 函数 */
8      FUNCPTR de_write; /* 指向驱动的 write() 函数 */
9      FUNCPTR de_ioctl; /* 指向驱动的 ioctl() 函数 */
10     BOOL de_inuse; /* 用于指示表项是否空闲 */
11 } DRV_ENTRY; /* 系统驱动表中的条目 */

```

图 2-6 DEV_ENTRY 结构体

DEV_ENTRY 结构体当中的大多数成员都是函数指针,他们用于指向所注册的驱动程序中的一个用于完成特定功能的实际函数,这些函数的功能要符合 IO 系统预定义好的规则,这些函数被加入到系统驱动表中之后就可以完成与用户层提供的标准函数接口对接^{[6][10][11]}。在 DEV_ENTRY 结构体当中唯一不是函数指针的成员是一个布尔类型的 de_inuse 成员,若该成员为 FALSE 则表示该表项目前是未被使用的状态,即该表项没有被任何的驱动所注册。

VxWorks 当中给我们提供了一个驱动的注册函数 iosDrvInstall(), 使用该函数注册我们的驱动之后,系统驱动表就会分配一个未被使用的表项给该驱动,然后使用

iosDrvInstall() 所提供的的参数来填充系统驱动表当中的指针，并将 de_inuse 置为 TRUE 的状态，一个驱动程序不需要实现所有的 IO 函数，对于实现的函数，在注册时直接将其指针置为 NULL 即可。

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中所打开的所有文件描述符，VxWorks 中系统描述符表的底层实现也是一个数组。每次执行 open() 调用成功之后，系统就需要从系统描述符表中分配一个表项给程序使用，并将文件描述符的表项索引作为文件描述符的 ID 返回给应用程序。之后应用程序直接通过这个 ID 就可以对文件进行操控，无需每次都是用文件名。在 VxWorks 中，标准输入、标准输出、标准错误输出虽然使用 0, 1, 2 三个文件描述符来表示，但是它在底层的实现上可能并不是占用了三个文件描述符表的表项，而是只占用一个表项，即三个文件描述符指向同一个文件描述符的表项^{[6][12]}，这一点是需要注意的。

系统文件描述符表中每一个表项都使用 FD_ENTRY 这个结构体来表示，这个结构定义在内核的头文件 iosLibP.h 中，其定义如图 2-7 所示。

```

1  typedef struct
2  {
3      DEV_HDR * pDevHdr; /* 该设备的设备头 */
4      int value; /* 设备的驱动号 */
5      char * name; /* 设备名指针 */
6      int taskId; /* 设备的任务ID */
7      BOOL inuse;
8      BOOL obsolete; /* 底层驱动是否仍然存在 */
9      void * auxValue; /* 驱动自定指针，根据驱动的需求实现 */
10     void * reserved; /* 保存未用 */
11 } FD_ENTRY; /* 系统文件描述符表的表项 */

```

图 2-7 FD_ENTRY 结构体

用户的应用程序每次使用 open() 系统调用系统文件描述符表中就会增加一个有效表项，该表项的 FD_ENTRY 结构体会根据 open() 调用的内容来进行填充，每一个文件能够进行的 open() 调用是有限制的，因为数组的容量是固定的，每个驱动的 FD_ENTRY 结构数组满了之后就无法再对这个设备进行 open() 操作，此时 open() 函数将会失败返回^[6]。系统会在表中的索引偏移 3 (0、1、2 被系统占用) 之后找一个最先找到的未使用的 id 作为文件描述符返回给用户。

2.2.2 VxWorks 中的信号量机制

任务间的通信机制用于协调多个任务之间的活动，VxWorks 内核当中为我们提供了丰富的任务间通信机制，包括共享内存、信号量、消息队列、管道、信号、Sockets 等^{[13][14]}。在我们调试通道的信息传输中需要设计一个特殊需求的 USB 转串口驱动，

在驱动当中需要使用信号量机制来确保对驱动内部缓冲区中的数据正确、有序的读写,因此我们有必要先了解一下 VxWorks 下的信号量机制。

信号量是一种在程序的设计当中最常使用的通信机制,其主要作用是线程间的同步和互斥。VxWorks 中提供 POSIX 信号量的同时还设计了专门的 wind 信号量,POSIX 信号量的使用主要是为了方便程序的移植。和 POSIX 信号量的不同之处在于,VxWorks 中设计的 wind 信号量为 VxWorks 系统进行了高度的优化,使得其更适用于实时操作系统,能够更快的实现任务间通信。VxWorks 中信号量是一个指向 SEMAPHORE 类型的结构指针,提供了二进制信号量、互斥信号量、计数信号量三种类型的信号量机制,他们适用于解决不同类型的问题。

- 二进制信号量

二进制信号量是最快、最通用的信号量,既可以用于同步也可以用于资源计数。

wind 的二进制信号量所需系统开销最少,适用于高性能的需求。二进制信号量在资源可用时标记为 FULL,在资源不可用时标记为 EMPTY。在 VxWorks 中二进制信号量使用函数 semBCreate() 来创建,二进制信号量的提取和释放过程如图 2-8 和图 2-9 所示。

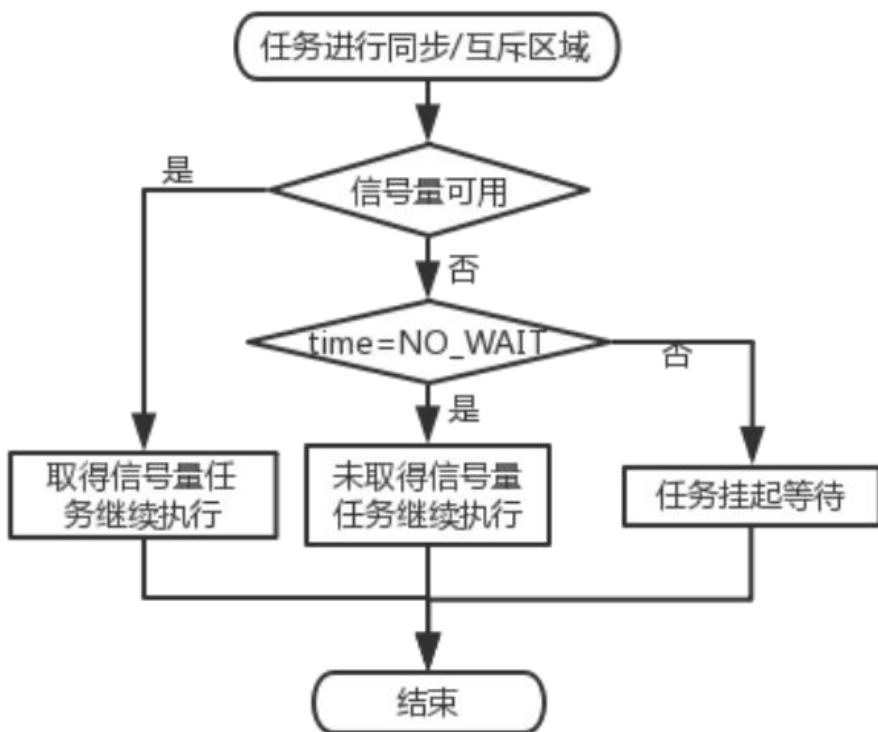


图 2-8 提取信号量

- 互斥信号量

互斥信号量可以看做是一种特殊的二进制信号量(资源数为 1),它优化了互斥、优先级继承、删除安全等问题,这使得它能够更好的服务于任务间的互斥需求;

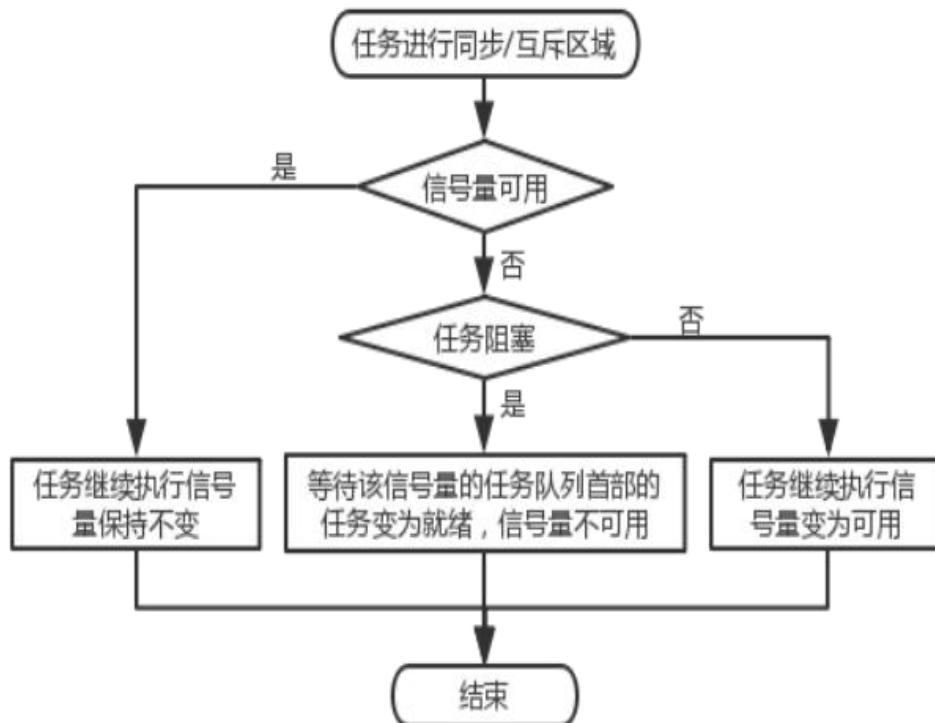


图 2-9 释放信号量

互斥信号量的基本行为和二进制信号量是一致的,但是互斥信号量只能够用于互斥,不能够用于同步,该信号量只能够由获得的该信号量的进程来进行释放,不能够由其它的进程进行释放。它使用 SEM_INVERSION_SAFE 和 SEM_Q_PRIORITY 选项来使得该信号量能继承优先级算法,以此解决优先级的倒置问题;使用 SEM_DELETE_SAFE 选项来解决删除安全问题,在 VxWorks 中互斥信号量使用系统提供的 semMCreate() 函数来创建;

- 资源计数信号量

资源计数信号量也是一种特殊的二进制信号量(资源数较多),它会跟踪信号量增加、删除的次数,每次释放一个信号量,内部的计数器就会执行加一操作,每次提取一个信号量,内部的计数器就会执行减一操作,当计数器为 0 时,表示没有可供使用的资源,此时提取信号量的操作就会被阻塞,在 VxWorks 中资源计数信号量使用系统提供的 semCCreate() 函数来创建。

三种信号量的释放操作都是使用 semGive() 函数;提取操作都是使用 semTake() 函数,在提取信号量是我们可以选择是否允许超时,超时可以作为解决阻塞的一种方法。

2.2.3 USB 技术

USB(Universal Serial Bus)作为PC领域的最新型的接口技术,目前已被各个PC厂家所支持,并且在各类外设当中都广泛的采用USB接口。USB的开发技术也已经很成熟,通用串行总线开发者论坛(USB Implementers Forum,USB IF)目前制定了三种USB接口标准:USB1.1,USB2.0和USB3.0。USB采用菊花链的形式连接所有的设备,最多可以连接127个设备,USB的总线拓扑结构如图2-10所示

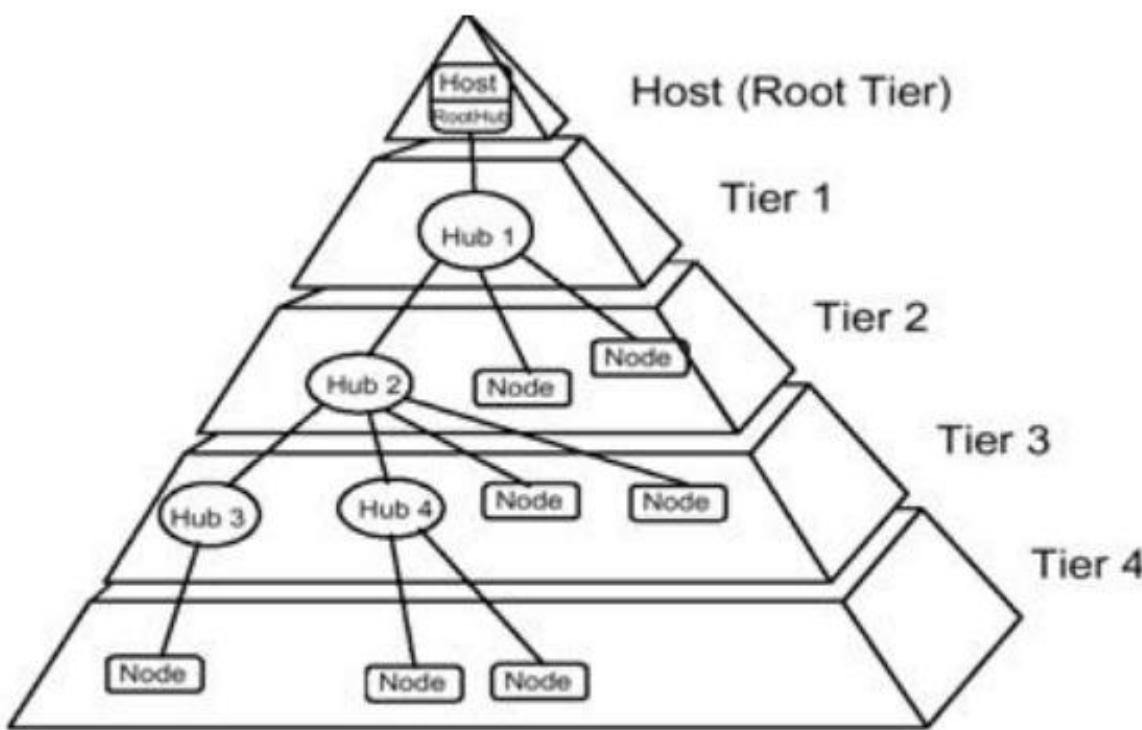


图 2-10 USB 总线拓扑结构

USB的体系结构由三个部分组成,分别是USB主机(Host)、USB集线器(Hub)、USB设备(Device)。其中我们需要了解的关键部分是USB主机和USB设备。

1. USB 主机

USB主机是USB体系中的核心,且系统中只允许一个USB主机存在。USB主机上的USB接口是USB主控制器,其控制着总线上所有USB设备数据通信。对于USB的体系结构而言,其数据的传输都是USB主机端发起的,非主机端(设备端)只能被动的进行响应。USB主机需要完成的功能包括检测设备的热插拔、管理主机和设备之间的信息(控制和数据)流^{[15][16]}。

2. USB 设备

USB设备指的是提供具体功能的外部USB设备,是相对USB主机而言的,它们受USB主机的控制,只能对主机的请求进行被动响应。USB主机端会在检测到USB设备的动作之后会通过默认管道和USB设备进行通信,对其进行必要的初始化

配置，并给设备提供适合的驱动程序（如果有的话），一个 USB 设备会通常会有很多的属性，它会通过这些属性来完成主机的配置要求。一些 USB 设备的属性如下：

- **描述符 (Descriptor) 属性**

描述符是 USB 协议中定义的一套用来描述 USB 设备的功能和属性的固定结构，我们可以通过描述符了解设备的各种属性，描述符又分为设备描述符、配置描述符、接口描述符、端点描述符、字符串描述符^{[17][18]} 除此之外，设备还可以提供自己专用的描述符，分为设备类描述符和供应商自定义描述符，我们使用的 USB 口转串口设备就不属于一个标准的 USB 设备，它会为我们提供供应商自定义的描述符，我们使用需要使用它来对设备进行识别。

- **类 (Class) 属性**

由于 USB 协议支持许多的外围设备，而这些设备又可以根据功能来分成一些相近的类，如打印机类、键盘类等。这样主机端就可以为这些功能相近的设备提供一个类驱动，类驱动可以用于驱动所有属于同一类的设备，不需要再为每一个设备提供一个完整的驱动程序。这大大的方便了设备的制造商，他们的设备只需要符合某一类的驱动，就可以使用该类驱动程序来驱动其设备，之后只需要实现简单的包含有设备特性的客户端驱动即可，若设备没有特殊的特性，则直接使用类驱动即可^[15]。

- **功能 (Function)/接口 (Interface) 属性**

功能或接口是 USB 协议中定义的设备的某种能力，Function 是从功能角度来说的，从设备的角度来说，被称为 Interface。对于一个设备他可以拥有很多个不同的接口，每一个接口负责完成设备的一个特定的功能，并且这个接口具体实现什么样的功能并不是固定的，当 USB 设备处于可配置状态的时候能够通过控制命令来改变某一个接口的功能，一个接口能够具有什么样的功能会在 USB 的接口描述符中进行描述。

- **端点 (Endpoint) 属性**

端点是 USB 设备与 USB 主机逻辑上的通信流的终点，每个设备都拥有一个可独立进行操作的端点集合，且每个端点在使用时都要先初始化其数据传输方向 (IN/OUT)，即使端点号相同但是传输方向不同的通信点也是不同的端点^[15]。

- **管道**

管道可以看做是设备上的一个端点和主机上的软件的联合体，设备和主机间的数据传输要基于管道进行。在 USB 的通信过程中首先要建立一个管道才能够进行数据的传输，USB 设备在和主机通信时都会建立一个默认的管道，这个管道对应的端点是默认端点 0，之后需要自己使用其它的端点来建立我们的数据传输过程中需要使用的输入或输出管道。在我们的 USB 口转串口驱动中会为每一个设备建立两个管道，一个批量输出管道和一个批量输入管道。

- **设备地址**

设备地址用于区分 USB 系统中的一个 USB 设备的特殊标识,设备地址会在设备初始化之后由主机进行分配且是唯一的。设备地址单元共有 7bit,其中地址 0 是缺省地址,在设备初始化的时候使用,理论上系统可以区分 127 个 USB 设备^[15]。

USB 规范规定了 USB 主从设备之间的四种传输方式,每种方式有各自的用途^[19]:

- **控制传输:** 控制传输 USB 传输方式中最重要、最复杂的一种,它适用于少量、对时间和速率无要求的场合,一个 USB 设备插入主机之后就是使用这种传输方式来读取设备的地址和描述符等信息。所有的设备都会在其 0 号端点的缺省管道当中支持控制传输^[17]。
- **批量传输:** 批量传输有两种最基本的事物类型: BULK_IN 和 BULK_OUT, 其主要用于处理对数据传输速率不是很高的情况,批量传输使我们的 USB 口转串口设备所使用的主要传输传输方式,每次有数据需要传输时我们都会构建一个 IRP 使用批量传输将其传出或传入。
- **中断传输:** 中断传输也有两种基本的事务: IN 和 OUT, 其主要是为那些要快速实现主机和设备的交互,但是数据量很小、对服务时间有要求的情况而准备的。
- **等时传输:** 等时传输也是由基本的 IN 和 OUT 两种事务组成,主要用于处理大量、恒速、对时间周期有要求的数据。等时传输只有全速和高速设备才支持,低速设备不支持^[17]。

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

设备驱动程序在操作系统中框架是操作系统的设计师在设计操作系统时已经制定好的,所以在进行驱动程序的开发时需要开发人员对操作系统和设备的硬件体系具有相当的了解,而且驱动程序的性能、可靠性也制约着系统的性能和可靠性。

3.1 VxWorks 上的 USB 开发

在 VxWorks 当中 I/O 框架的实现放在内核文件 ioLib.c 当中,我们将它称为上层接口子系统,因为它负责给上层应用提供 I/O 接口,但是该接口并不会完成具体请求的实现,他只会在进行简单的处理之后将请求转发给 I/O 子系统,具体请求的处理工作会被发送到 I/O 子系统当中来处理。I/O 子系统的定义放在在内核文件 iosLib.c 当中,I/O 子系统对用户层而言通常是透明的,用户层一般不会直接调用 I/O 子系统中的函数,它是作为上层接口子系统与下层驱动系统的中间层来使用的。VxWorks 的当中内核驱动层次结构如图 3-1所示。

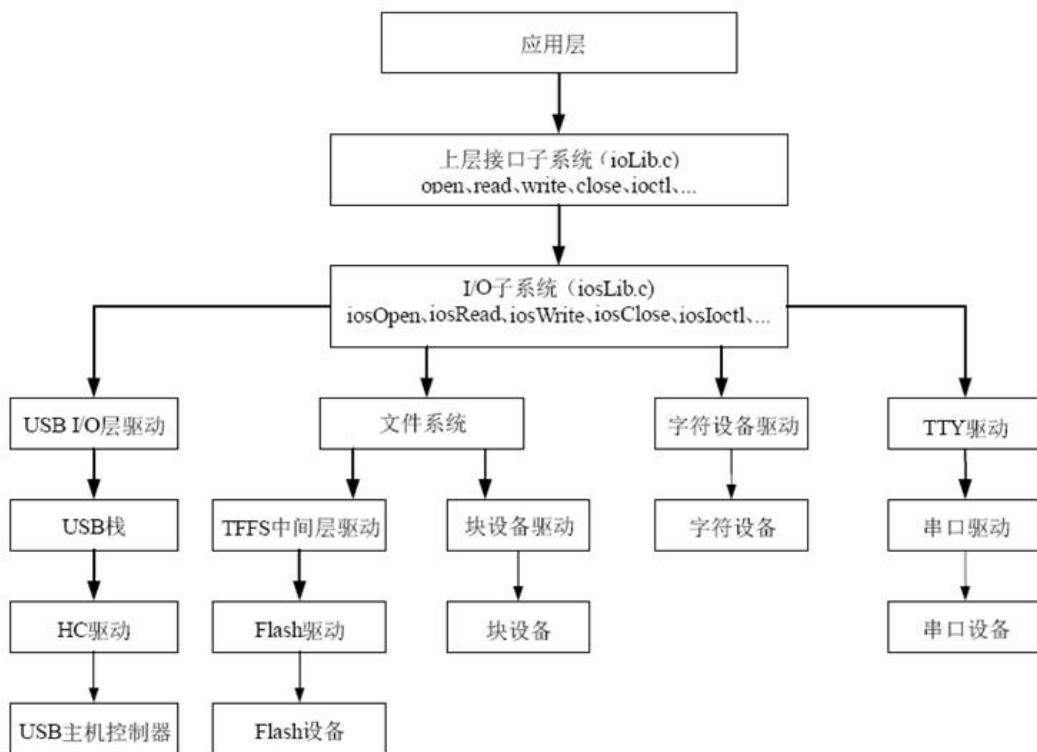


图 3-1 VxWorks 驱动内核层次结构

从图 3-1当中我们可以看出主机端 USB 驱动在 VxWorks 当中的层次结构,但是 VxWorks 作为一个嵌入式系统,其作为 USB 的主/从端的可能性都是存在的,所以

其上也有 USB 从端驱动栈的实现。但我们本次需要使用的是其主机栈,在 VxWorks 当中 USB 驱动程序堆栈的开发符合的是通用串行总线规范 2.0 标准,VxWorks 中的 USB 主机端内核驱动层次如图 3-2 所示。VxWorks 中的将 USB 协议在主机端分成三层来实现,分别是客户端驱动程序 (Client Driver)、USB 驱动 (USBD)、主机控制器驱动 (HCD),每一层完成不同的功能。其通信的逻辑结构和 PC 端的软硬件结构如图 3-3 所示。

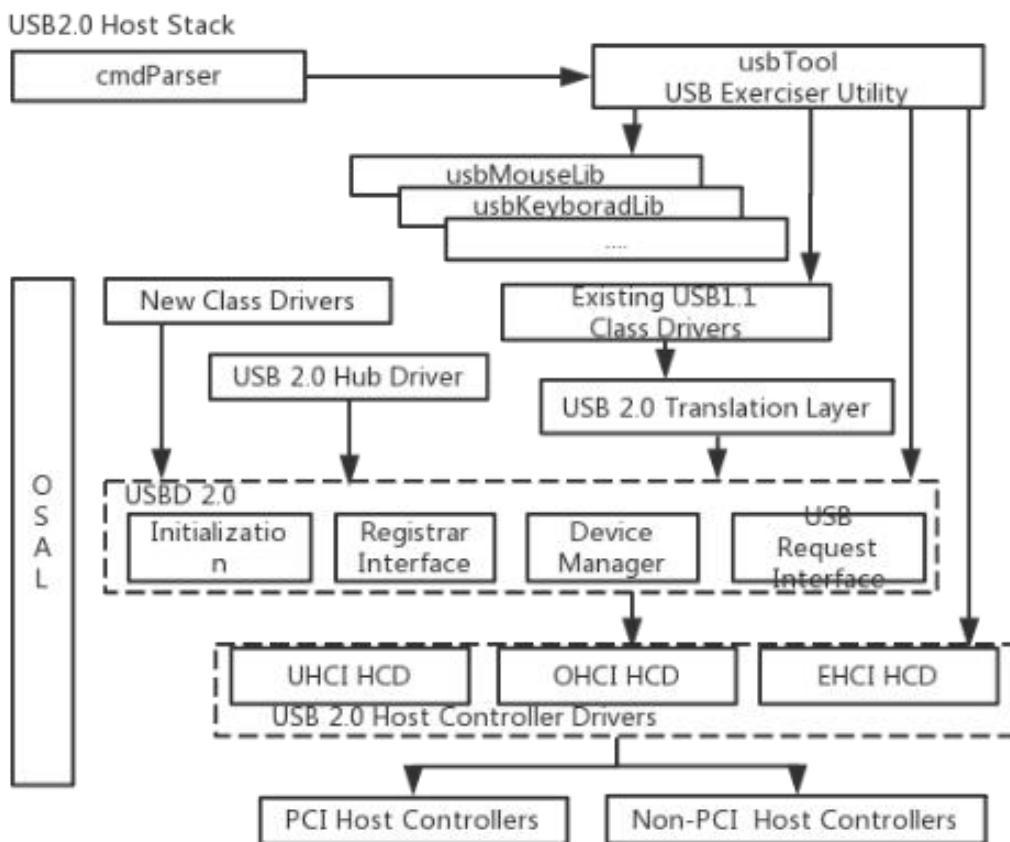


图 3-2 VxWorks 下 USB 内核驱动层次结构

客户端驱动程序的功能是完成对不同类型设备的功能驱动。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。USB 口转串口驱动会构建 USB I/O 请求包 (I/O request, IRP) 来向 USBD 层发出数据接收或者发送的请求,IRP 是 USB 协议中定义的一个抽象概念,我们需要根据所发送的内容和发送的方式来具体的实现一个 IRP^[15]。

USBD 是 USB 的核心驱动,其提供了操作系统组件 (主要是设备驱动) 能够用来访问 USB 设备的方法,其实现是由操作系统决定的。USBD 提供包括 USB 总线的枚举、总线带宽的分配、传输控制等操作,它还会处理客户端驱动程序发送来的 IRP 包,并在对 IRP 请求包进行解析之后将实际的请求映射到适当的 HCD 或者直接交给主机控制器处理。除此之外 USBD 负责的内容还包括新设备的动态插拔、电源管理和对

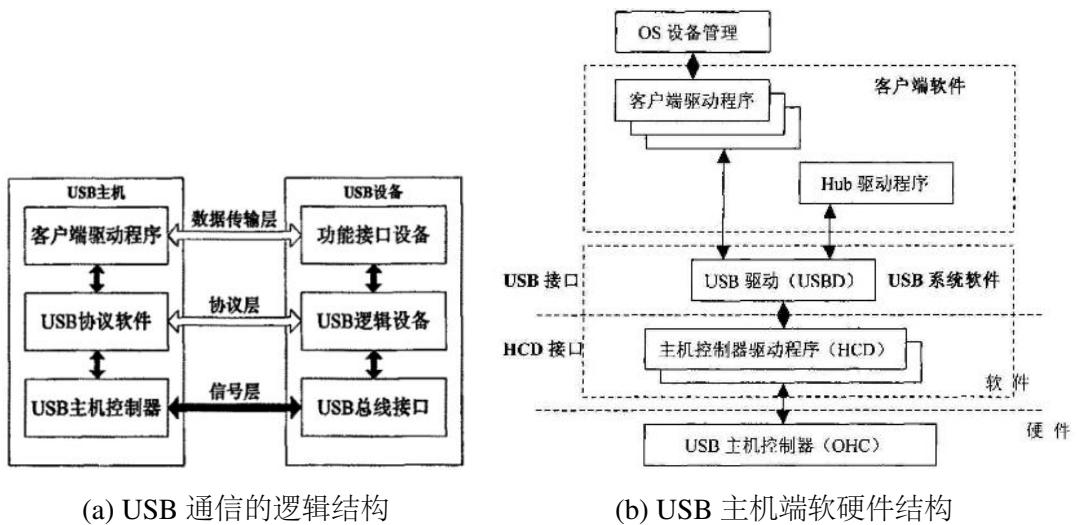


图 3-3 USB 通信结构

客户端驱动程序的维护等。

HCD 层提供一个软件抽象来完成 USB 主控制器的各种事物处理, 用于隐藏主控制器的硬件实现细节, HCD 只服务于我们的 USBD 层, 对上层应用而言其通常不能直接访问的。VxWorks 中 HCD 层只需要向 USBD 层注册一个接口函数, 所有的 USBD 层的请求都是用这个接口函数来完成, 对于 UHCI 主机控制器驱动而言他被命名为 usbdHcdUhciExec, 对于 OHCI 主控器驱动而言被命名为 usbdHcdOhciExec。HCD 层会将 USBD 层传输下来的事务调度给主机控制器进行处理, 当事物处理完成之后 HCD 层会将处理结果返回给 USBD 层。此外它还会完成对主机控制器和根集线器的配置和驱动等操作^[15]。VxWorks 中本身已经集成好了对几种常用的主机控制器驱动的支持, 包括 UHCI 主控器、OHCI 主控器、EHCI 主控器, VxWorks 中暂时没有实现 xHCI 主控驱动。

对于我们的 USB 口转串口驱而言,我们不再需要实现对物理设备的数据结构抽象,因为 VxWorks 中的 USB HCD 层已经为我们实现好了物理层的抽象,同时还给我们提供了 USBD 层,因此我们只需要实现 USB 主机三层结构当中的客户端驱动程序即可,使其能够驱动特定的 USB 设备正常工作。在客户端驱动当中,我们需要完成 IO 系统的各类接口的实现、完成驱动需要实现的特殊功能的实现、将驱动集成到系统中等工作。

3.2 CP2102 开发

除了驱动程序之外,USB 口转串口的实现还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:一类是使用包含有 USB 单元的微处理器从底层的硬件和固件进行全面系统的设计,这样的控制器有 PCI16C745、68HC705JB4 和 C541U 系列等^[20],但是使用这种方式从头

开始设计存在难度大,系统复杂等问题,不符合我们本次设计工作的需求。第二类方法是采用市场上设计好的 USB/RS-232 双向转换芯片,这类芯片有 CH341, 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。

CP2102 是 SILICON LABORATORIES 推出的 USB 与 RS232 接口转换芯片,它包含有一个 USB2.0 全速功能控制器,EEPROM,USB 收发器,振荡器和异步串行数据总线(UART),在 SILICON 给出的文档当中已经帮我们给出了一个最简单 CP2102 的使用方式的电路框图,如图 3-4 所示。

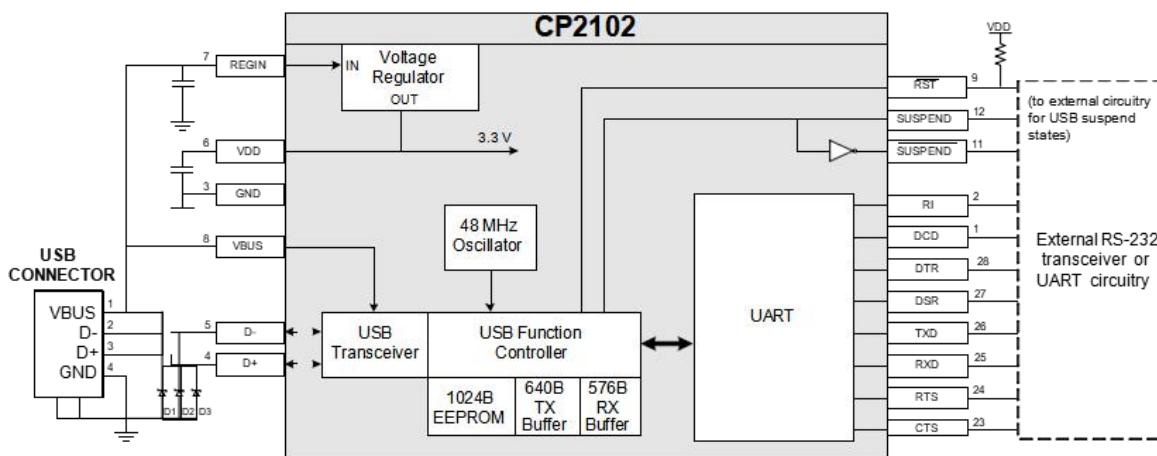


图 3-4 cp2102 电路框图

1. CP2102 的 USB 功能控制器和收发器:CP2102 的内置的 USB 功能控制器符合 USB2.0 协议,它负责管理 USB 和 UART 之间的所有数据和控制传输。
2. 异步串行数据总线(UART)接口:CP2102 的 UART 接口支持接收、发送、控制、握手信号,而且支持对 UART 数据格式和波特率进行编程控制。可以使用的数据格式和波特率见图 3-5。
3. 内部 EEPROM:CP2102 内部集成了一个 1K 的 EEPROM,在其中存储了一些设备的特定信息,包括厂商 ID、产品 ID、产品说明、电源参数、器件版本号和器件序列号等^[21]。如果 OEM 没有为设备的 EEPROM 写入数据的话,那么设备会自动的使用一组默认的数据,如图 3-6 所示。

使用 CP2102 进行串口扩展的时候只需少量外部器件,CP2102 当中的协议控制单元会对来自 USB 接口的命令进行解析,然后对 UART 接口进行配置。UART 的部分可配置参数如图 3-5 所示,CP2102 当中的拥有一个 576B 的接收缓冲区和一个 640B 发送缓冲区,这些缓冲区可以部分的解决 USB 和 RS-232 之间的速率不匹配的问题。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

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

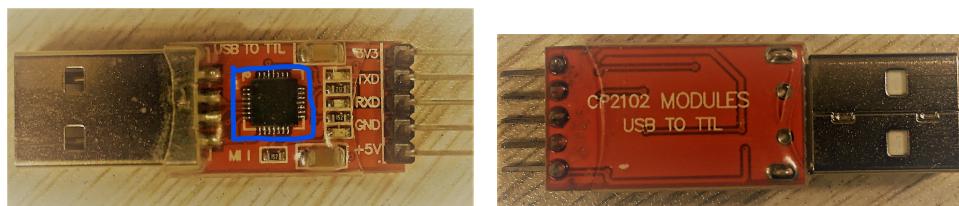
图 3-5 cp2102 可选配置参数

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)"

图 3-6 CP2102 默认配置表

后,PC 会对其进行初始化并在识别出该设备后启动支持该设备的客户端驱动;计算机上的驱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^[15]。

在我们本次的设计当中我们并不会去完成 CP2102 的外部电路的设计工作,而是选择市场上已经封装好的模块。我们本次使用的设备如图 3-7 所示。我们所需要做的是了解 CP2102 的原理和功能,能够对其进行正确的开发工作。



(a) CP2102 模块正面

(b) CP2102 模块反面

图 3-7 CP2102 模块正反面

3.3 特定需求单设备驱动的实现

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。

无论驱动程序的开发人员要完成一个什么样的驱动程序,其都有一些系统规定的标准的模块,以方便控制设备。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,在这些模块当中关键要处理的内容是对数据的传输工作和对设备的控制工作,数据的传输要用到缓冲区和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,返回设备的指针
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

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,并为其初始化好设备的缓冲区,具体的设计如下文所示。

3.3.1 设备的自定义结构

底层驱动都要为其所驱动的每一个设备维护一个设备的自定义数据结构,这个结构的第一个成员必须是 **DEV_HDR** 结构体,**DEV_HDR** 这个结构是给系统维护该设备的驱动时使用的,其余的成员为该设备所需保存的关键参数,这些参数都是我们的驱动程序在设备运行时需要使用的,系统不会使用这些参数。对于我们的 USB 口转串口设备而言,我们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:

```

1 typedef struct cp210x_dev
2 {
3     DEV_HDR cp210xDevHdr;
4
5     UINT16 numOpen;
6     USBD_NODE_ID nodeId;
7     UINT16 configuration;
8     UINT16 interface;
9     UINT16 interfaceAltSetting;
10    UINT16 vendorId;
11    UINT16 productId;
12
13    BOOL connected;
14    int trans_len;
15    USBD_PIPE_HANDLE outPipeHandle;
16    USB_IRP outIrp;
17    BOOL outIrpInUse;
18    UINT16 outEpAddr;
19    UINT8 trans_buf[64];
20
21    USBD_PIPE_HANDLE inPipeHandle;
22    USB_IRP inIrp;
23    BOOL inIrpInUse;
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,系统只会识别到 **DEV_**

HDR 结构并对其进行管理，在系统设备表、系统文件描述符表当中都需要使用该驱动中的 **DEV_HDR** 这个成员。

- **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。

3.3.2 驱动初始化

底层的驱动程序都要提供一个初始化函数给系统来调用，以便进行驱动程序的注册和初始化驱动必须的资源，这些工作通常是在内核启动过程中进行的，你必须将你的驱动加入到内核的启动列表当中，内核在启动的时候就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()` 为初始化函数，其主要完成驱动所需要的资源的申请和全局变量的初始化，包括创建信号量、向系统注册驱动、创建设备、向 USBD 层注册。驱动初始化函数的主要流程如图 3-8 所示。

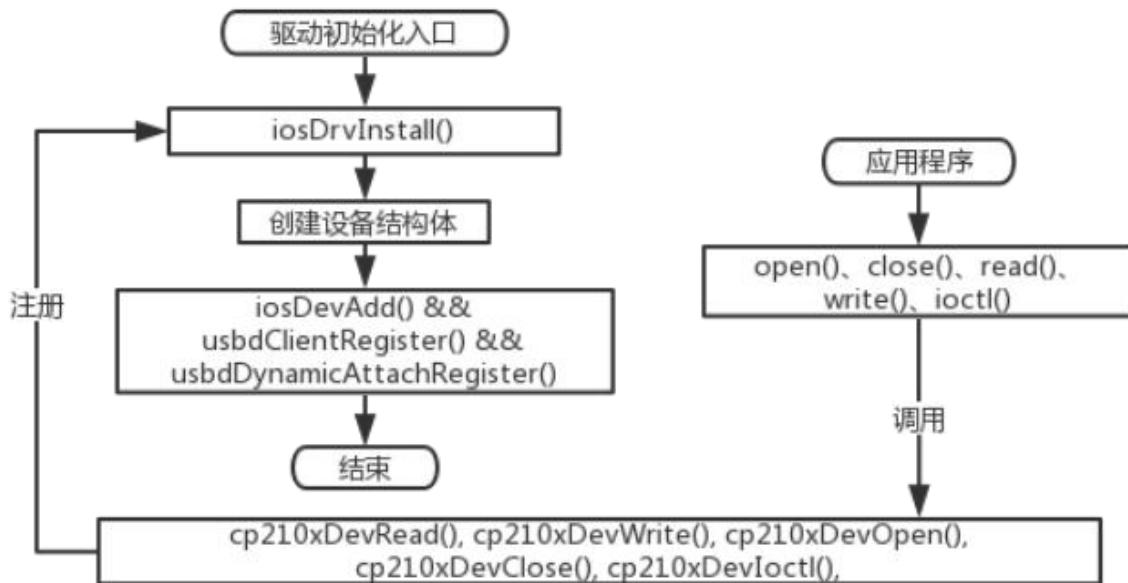


图 3-8 单设备驱动初始化流程

在调用驱动初始化函数时，驱动首先会检查自己是否已经安装过了，若已经安装了则无需再次安装，直接返回即可。我们通常会将驱动的初始化模块放在

usrRoot(usrConfig.c) 中调用,这样驱动就可以随着内核的启动而启动。但是你也可以将其当做是一个核外的模块来进行启动,在 VxWorks 中你需要加载驱动程序所在的库,然后直接运行 cp210xDrvInit() 函数即可。驱动初始化时需要初始化我们用来在驱动的运行过程中使用的各种信号量,还需要调用 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。

此处和普通驱动不同的地方是,我们单设备驱动的初始化函数当中就已经初始化好了驱动所需要的缓冲区,并且已经先将设备添加到了 I/O 子系统当中,此时实际的设备并不存在,添加到系统当中的只是一个“假”设备,只有软件实现,没有实际的硬件支撑,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。而普通的驱动程序中的逻辑应当是在检测到设备插入,并且识别出该设备能够被我们的驱动程序所支持时才给其分配缓冲区、进行设备的初始化工作,最后再调用 iosDevAdd() 将设备注册到 I/O 子系统。

在设备成功的添加到 I/O 子系统之后,系统的设备表当中就会显示该命名为 CP210X_NAME 的设备(CP210X_NAME 只是一个宏定义,设备名可以自己更改),我们可以在系统中使用 iosDevShow 命令来进行查看。

在设备注册完之后我们还要将 USB 客户端驱动还需要向 USBD 层注册,在 3.1 小节中我们介绍过,USB 主机架构将其分为三层来实现,USBD 层就是为我们的客户端驱动所服务的,它负责管理 USB 的中断、热插拔等事件。它给我们提供了注册函数 usbdClientRegister() 和 usbdDynamicAttachRegister(), 使用 usbdClientRegister() 注册之后 USBD 层才能知道我们的客户端驱动的存在,并将我们的驱动加入到它的服务列表当中,它会给我们的驱动分配一个独一无二的句柄 cp210xHandle,以后对 USBD 层进行操作是只需传递 cp210xHandle 这个独一无二的句柄 USBD 层就能区分是哪一个驱动在发送数据。

usbdDynamicAttachRegister() 用于注册我们的驱动所感兴趣的设备的热插拔事件,并像 USBD 层注册一个回调函数 cp210xAttachCallback,当我们所感兴趣的热插拔事件发生时,USBD 层会调用我们所注册的回调函数通知我们。usbdDynamicAttachRegister() 函数的原型如图 3-9 所示。在热插拔的注册函数中我们需要指定我们所关心的设备是属于哪一个类、子类和协议,但是由于我们的 USB 口转串口设备并不属于任何一个标准的类、子类和协议层次,它使用的都是厂商自定义的协议,所以在此处我们将其注册为 USBD_NOTIFY_ALL,即所有的 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。

```

1 STATUS usbdDynamicAttachRegister(
2     USBD_CLIENT_HANDLE clientHandle,           /* Client handle */
3     UINT16 deviceClass,                      /* USB class code */
4     UINT16 deviceSubClass,                   /* USB sub-class code */
5     UINT16 deviceProtocol,                  /* USB device protocol code */
6     USBD_ATTACH_CALLBACK attachCallback /* User-supplied callback */
7 )

```

图 3-9 热插拔注册函数

3.3.3 设备的识别和初始化

设备的初始化工作需要在设备被我们的客户端驱动所识别之后才进行，在驱动初始化时我们注册好了热插拔的回调函数 cp210xAttachCallback()，在回调函数中我们最先做的就是识别设备，识别完成之后再对其进行初始化，识别设备通过发送标准的 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，在 VxWorks 当中对 USB 设备描述符信息的定义如图 3-10 所示；

```

1 typedef struct usb_device_descr
2 {
3     UINT8 length;                         /* bLength */
4     UINT8 descriptorType;                /* bDescriptorType */
5     UINT16 bcdUsb;                      /* bcdUSB - USB release in BCD */
6     UINT8 deviceClass;                  /* bDeviceClass */
7     UINT8 deviceSubClass;                /* bDeviceSubClass */
8     UINT8 deviceProtocol;                /* bDeviceProtocol */
9     UINT8 maxPacketSize0;                /* bMaxPacketSize0 */
10    UINT16 vendor;                     /* idVendor */
11    UINT16 product;                    /* idProduct */
12    UINT16 bcdDevice;                  /* bcdDevice - dev release in BCD */
13    UINT8 manufacturerIndex;          /* iManufacturer */
14    UINT8 productIndex;                /* iProduct */
15    UINT8 serialNumberIndex;          /* iSerialNumber */
16    UINT8 numConfigurations;          /* bNumConfigurations */
17 } WRS_PACK_ALIGN(4) USB_DEVICE_DESCR, *pUSB_DEVICE_DESCR;
18 ...
19 usbdDescriptorGet (cp210xHandle, nodeId,USB_RT_STANDARD |
20     USB_RT_DEVICE,USB_DESCR_DEVICE,0, 0, sizeof(pBfr), pBfr, &actLen)
21 ) != OK)

```

图 3-10 usb 设备描述符信息结构

VxWorks 当中提供了 usbdDescriptorGet() 函数来获取设备描述符，该函数的三、四个参数用于指定所需要获取的描述的类型，对于此处我们需要获取的是设备的标准描述符，于是将其设置为 USB_RT_STANDARD|USB_RT_DEVICE 和 USB_RT_DEVICE。我们将标准设备描述符的信息放在指向 **USB_DESCRIPTOR** 类型的结构体指针 **pBfr** 当中，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、

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 3.2 目前支持的设备列表

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对设备进行接口进行配置、创建输入/输出管道。对于我们的 USB 口转串口设备还有一点特殊的地方在于它还需要配置串口的波特率、数据位、停止位、流控等信息，这是普通的 USB 设备所没有的。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 3-12 所示。

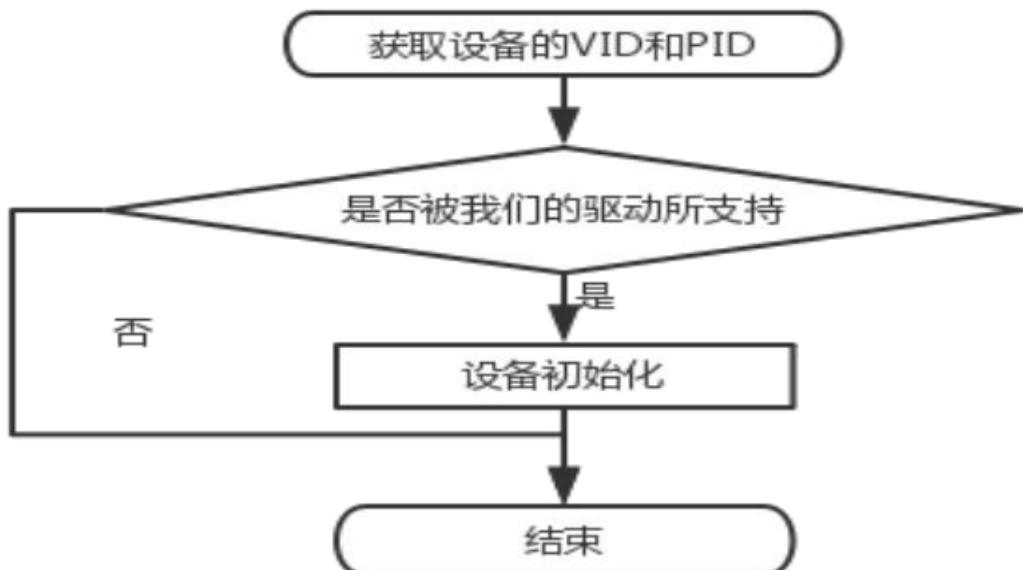


图 3-12 设备初始化流程图

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。

在驱动程序初始化完成之后我们会立即启动 listenForInput() 这个函数来注册一个设备输入的回调函数来监听设备的输入动作，当设备产生中断输入数据时，USBD 层就会通知我们注册的这个回调函数。另外我们会立即执行一个发送操作的触发函数 initOutPut()，当设备的输出缓冲区有数据时会立即执行数据的发送，这是我们的单设备驱动程序中一个特殊的流程。关于数据详细的传输和控制的过程我们会在 3.3.5 小节 设备的读写部分介绍，设备识别和初始化的整个流程在回调函数中

如图 3-13 所示。

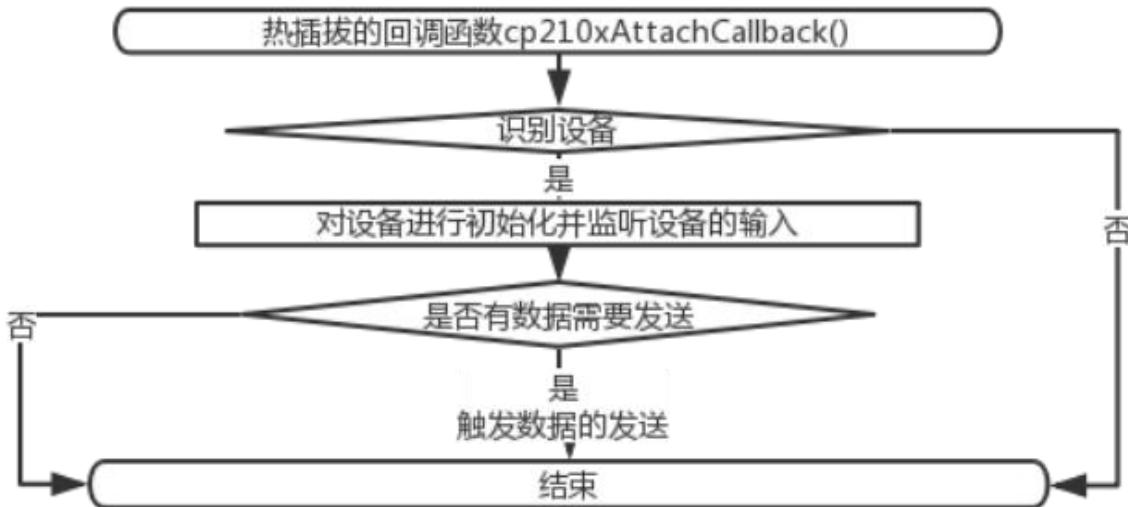


图 3-13 单设备回调函数流程

3.3.4 设备打开/关闭

用户在使用一个设备之前必须先对这个设备进行打开,在这个过程当中通常而言底层驱动的响应函数会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 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时已经完成,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/关闭代码如下所示:

```

1 LOCAL CP210X_DEV * cp210xDevOpen(DEV_HDR *pDevHdr, char *name, int flags,
2                                     int mode)
3 {
4     CP210X_DEV *pCp210xDev;
5     pCp210xDev = (CP210X_DEV *)pDevHdr;
6     pCp210xDev->numOpen++;
7     return (pCp210xDev);
8 }
9 LOCAL CP210X_DEV * cp210xDevClose(CP210X_DEV *pCp210xDev)
10 {
11     --pCp210xDev->numOpen;
12     return (pCp210xDev);
13 }
```

cp210xDevOpen() 函数用来承接系统的 open() 调用,它的第一个 DEV_HDR 类型的参数即是之前所说的必须是设备自定义结构的第一个成员,这个参数会由 I/O 子系统自动提供,I/O 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”, 但是如果系统中并没有完全符合这个设备名的设备, 而是只有名为”/ttyUsb/x” 的设备, 那么系统就会打开这个设备, 而将匹配完成之后剩余的剩余的字符串”yz” 传递给 name。我们的应用中 open() 函数调用时的路径名应该与系统设备列表中的设备名是完全一致的, 此处的 name 就会是一个空字符串, 所以这个规则对于我们并没有什么影响。这是规则的主要用处在于处理文件系统层下的块设备, 此时 name 可以指向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任何更改, 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, I/O 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。

3.3.5 设备的读写

在 open() 操作成功之后, 我们可以得到一个返回值, 这个返回值就是设备的文件描述符, 然后我们就可以使用这个文件描述符对设备进行 read(), write() 等操作。我们的 USB 口转串口驱动的读写函数原型如下:

```

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

```

由于我们特殊驱动程序的要求与普通流程下的驱动有所不同, 其初始化方式比较特殊, 对于数据的传输操作的流程也有相应的需求。我们在此需要对其数据流进行重新的设计。

对于这个特殊流程的驱动程序, 我们会在驱动程序初始化的时候就为其建立好输入输出缓冲区, 而不是在设备初始化的时候为设备创建输入输出缓冲区, 因为我们需要在设备还未连接的时候就能够打开设备, 往设备的输出缓冲区中写入数据。特殊驱动程序的数据的输出操作会分为两个流程来进行。缓存空间的大小是以宏的方式定

义在头文件当中的,方便以后改动,分别定义为:WRITE_BUFFER_SIZE 和 READ_BUFFER_SIZE。

上层应用写入数据时会使用系统调用 write(),该函数在 VxWorks 当中对应于我们驱动程序中的函数为 cp210xDevWrite(),这个函数会接受 write() 发送过来的数据,并将其存入输出缓冲区中,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设备当前没有数据发送时完成,若调用 write() 时设备已经在发送数据,则只需要将数据存入缓冲区即可,设备发送完目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 3-14 所示。

设备在连接上系统并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,一个数据包发送完成之后会再次去查询驱动的输出缓冲区中是否仍然有数据需要发送,若有数据要发送则继续发送,若没有则等待上层应用写入数据来触发数据的发送操作,其输出流程如图 3-15 所示。

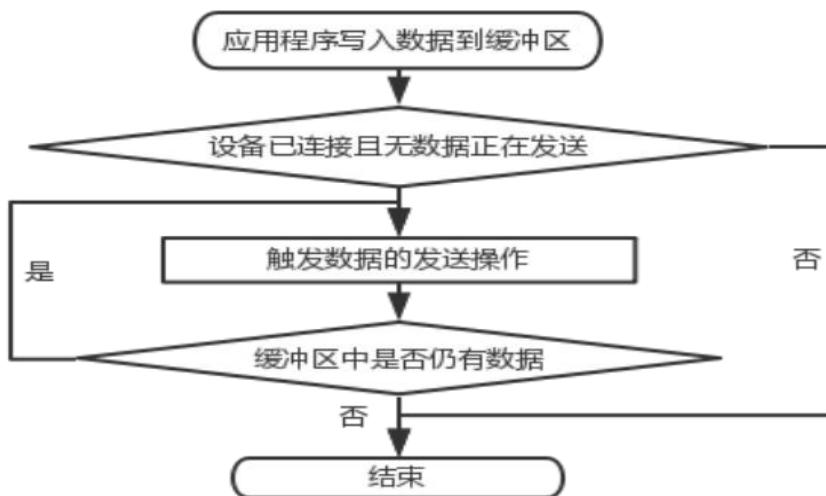


图 3-14 write 操作的输出流程

对于数据的输入操作,因为数据的输入是要依赖于实际的硬件的,不像输出操作一样存在“假”的输入。我们在设备的初始化完成之后我们会立即启动 listenForInput() 这个函数来注册一个设备输入的回调函数来监听设备的输入动作,当设备产生中断输入数据时,USBD 层就会通知我们注册的这个回调函数,在这个回调函数中我们会处理设备输入的数据。

在 listenForInput() 当中我们先创建好一个用来读取数据的 USB IRP,在这个 IRP 当中我们注册回调函数为 cp210xIrpCallback, 获取 IRP 的超时时间为 USB_TIMEOUT_NONE 当有 USB 的输入管道当中有输入数据到来时 USBD 层会调用我们注册的回调函数来通知我们。在处理完一次的回调之后我们需要启动下一次的监听,因为每一次的 IRP 都是单次有效的,所以在 cp210xIrpCallback() 当中我们接收完这一次的 IRP 的数据并将其进行处理之后需要新建另一个 IRP 重新启动下一次的

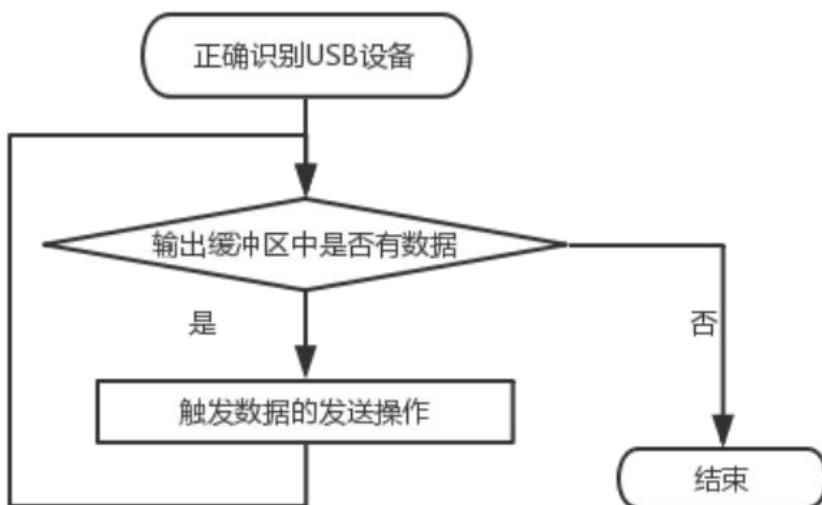


图 3-15 设备连接上系统时输出流程

listenForInput() 过程。

因为我们的驱动的程序的主要功能就在于数据的传输,在数据的传输过程中对于缓冲区的设计和信号量的使用方式直接影响到我们的驱动程序性能,因此有必要对这两部分再进行详细的描述。

3.3.5.1 环形缓冲区的设计和控制

应用程序、设备驱动和环形缓冲区的关系如图 3-16 所示,通过环形数据缓冲,我们可以解决速度匹配的问题和数据丢失的问题,应用程序也无需阻塞等待设备的空闲,只需将数据发送到我们的缓冲区中即可。

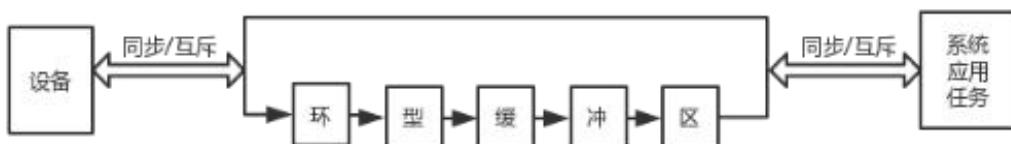


图 3-16 设备数据缓冲

我们使用循环队列实现的来实现环形缓冲区,由于计算机的内存是线性地址空间,因此从逻辑上实现环形缓冲区时通常需要 4 个指针:

- 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,由于我们已经在驱动中定义了缓冲区的长度为 WRITE_BUFFER_SIZE 和 READ_BUFFER_SIZE,所以这个指针我们不再需要;
- 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)。

当数据元素插入或者是删除时,我们不需要移动其余的数据元素存储位置,只需要移动队头和队尾指针即可。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,当缓冲区中接收到一个数据时,将此数据保存到队尾的位置并将队尾指针加 1,当缓冲区中取走一个数据时,我们将队头指针减一。

无论是对于读缓冲区还是写缓冲区,我们都需要解决的一个问题是当循环缓冲区中的数据已经满了之后,再有数据需要进入到缓冲区当中时,我们怎么进行处理,此处我们选择的策略是遵循先进先出的原则来覆盖数据。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。

对于输入缓冲区,我们使用一个读互斥信号量 cp210xReadMutex 和一个读二进制信号量 ReadblockSem 来进行缓冲区的控制, cp210xReadMutex 的作用保证对读缓冲区的正确访问顺序, ReadblockSem 的作用是用于对上层读任务和底层驱动的写任务进行同步。在使用 ReadblockSem 时我们会设置一个超时时间 TIMEOUT, 当时间超过时 semTake() 操作就会返回一个错误, 我们上层读取任务此时就会得到一个错误返回, 说明无法读取到数据。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, 直到超时返回或者 USBD 层通知你有新的数据到来时再继续进行读操作。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。

```

1 cp210xDevRead(...)//上层应用读取任务
2 {
3 ...
4     semTake(cp210xReadMutex, WAIT_FOREVER)
5     if(available == 0)
6     {
7         semGive(cp210xReadMutex)
8         //等待缓冲区写入数据, 当底层往缓冲区中写入数据时会释放
9         ReadblockSem
10        status = semTake(ReadblockSem, TIEMOUT);
11        if(status == ERROR)
12            return ERROR;
13
14        semTake(cp210xReadMutex, WAIT_FOREVER)
15    }
16    copy_from_readBuf; /*从缓冲区读取数据*/
17    semGive(cp210xReadMutex)
18    ...
19
20 cp210xIrpCallback(...)//底层从硬件接收数据
21 {
22 ...
23     semTake(cp210xReadMutex, WAIT_FOREVER)
24     if(available == 0)//缓冲区中没有数据, 则释放一次 ReadblockSem
25         semGive(ReadblockSem);
26     copy_to_readBuf()//向读缓冲区中写入数据
27     semGive(cp210xReadMutex);
28 ...
29 }
```

对于输出缓冲区，我们使用一个写互斥信号量 `cp210xWriteMutex` 来进行缓冲区的读写控制，由于我们这个特殊的驱动程序在往外输出数据时只需要将数据输出到了缓冲区的即可，不管实际的设备是否连接，因此我们不需要使用二进制信号量来进行上层写任务和底层读任务之间的同步操作。此时设备如果连接在系统上则会触发数据的发送操作，若设备没有连接在系统上，那么设备在连接上之后会自动的进行一次判断缓冲区中是否有数据的操作，有数据则触发发送操作。部分关键代码如下：

```

1 cp210xDevWrite(...) //上层应用写任务
2 {
3 ...
4     semTake(cp210xWriteMutex, WAIT_FOR_EVER);
5     copy_to_writeBuf; /*从缓冲区读取数据*/
6     semGive(cp210xWriteMutex);
7 ...
8 }
9
10 initOutPut() //底层从缓冲区读取数据写入设备
11 {
12 ...
13     semTake(cp210xWriteMutex, WAIT_FOR_EVER)
14     if(available == 0)
15     {
16         semGive(cp210xWriteMutex);
17         return;
18     }
19     copy_from_writeBuf() //从输出缓冲区中取出数据发送
20     initOutIrp(); //构建数据发送IRP
21     semGive(cp210xWriteMutex);
22 ...
23 }
```

3.3.6 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，可供执行的再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 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 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:

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

对于我们的 USB 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() 函数当中我们会使用 switch 语句来对请求类型进行分类, 所有的请求最后都要使用 usbdVendorSpecific() 函数来发送 USB 的厂商自定义请求, 该函数的原型如图 3-17 所示。

```
1 STATUS usbdVendorSpecific
2 (
3     USBD_CLIENT_HANDLE clientHandle, /* Client handle */
4     USBD_NODE_ID nodeId, /* Node Id of device/hub */
5     UINT8 requestType, /* bmRequestType in USB spec. */
6     UINT8 request, /* bRequest in USB spec. */
7     UINT16 value, /* wValue in USB spec. */
8     UINT16 index, /* wIndex in USB spec. */
9     UINT16 length, /* wLength in USB spec. */
10    pUINT8 pBfr, /* ptr to data buffer */
11    pUINT16 pActLen /* actual length of IN */
```

图 3-17 usbdVendorSpecific()

其中第三个参数是请求类型, 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)), 第四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图 3-18 所示。

3.3.7 驱动卸载

在驱动的卸载函数当中我们需要完成驱动的反注册, 包括注销向 USBD 层的注册和向系统设备表的注册, 同时还需要销毁具体的设备所占用的各种资源。

在单设备驱动的注销函数中我们应该首先对驱动的热插拔回调函数进行注销, 这样就可以阻止对此后再接入系统上的设备进行响应, 之后应该因此注销 USBD 客户端、系统设备表、系统驱动表, 然后再清理驱动的全局资源和设备对应的资源。若此时由设备连接在系统上且有数据正在发送, 那么应该等待设备当前正在发送的数据发送完之后再清理设备对应的资源, 其流程如图 3-19 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

图 3-18 部分厂商自定义请求

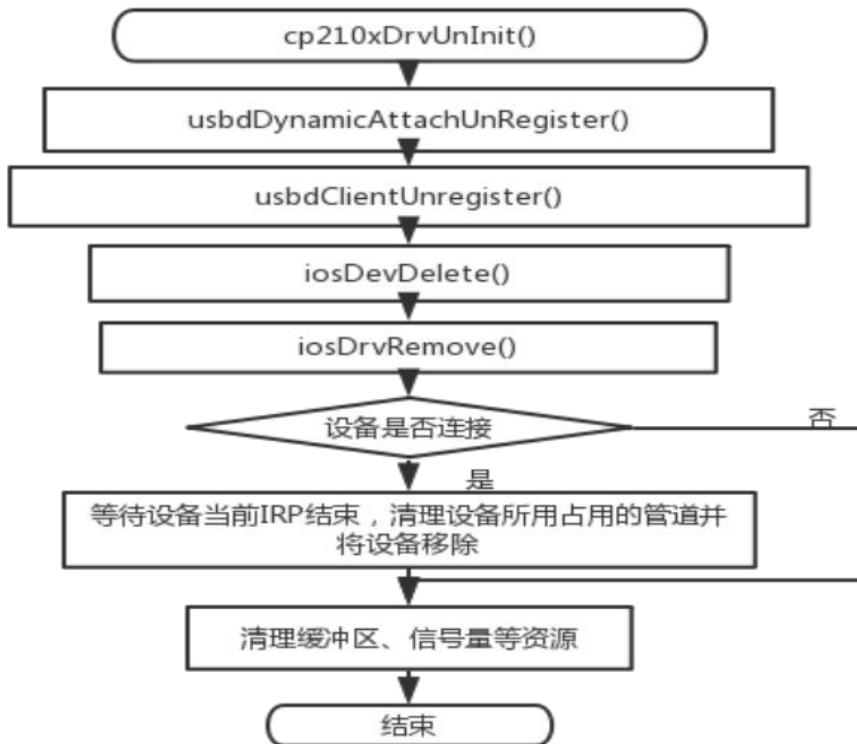


图 3-19 单设备驱动注销流程

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。该函数调用原型如图 3-20 所示。

该函数的第二个参数指定是否对驱动程序进行强制性的卸载，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

```

1 STATUS iosDrvRemove
2 (
3     int drvnum, /* driver to remove, returned by iosDrvInstall() */
4     BOOL forceClose /* if TRUE, force closure of open files */
5 );

```

图 3-20 iosDrvRemove()

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。

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。

3.4 通用多设备驱动的实现

多设备驱动与特殊需求的单设备驱动最明显的不同之处在于整个驱动程序的启动流程不一样, 其次需要支持多设备, 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自定义结构体, 多设备驱动的具体设计如下文所述。

3.4.1 设备的自定义结构体

在多设备的自定义结构体当中除了要保存和单设备结构体当中一样的设备的配置、管道、端点地址等信息之外, 我们还需要保存每一个设备的读写缓冲区的基地址和头尾指针。除此之外我们使用了一个链表 devHdrLink 来链接接入系统上的该驱动支持的 USB 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, 若不存在则给该设备分配一个设备名。多设备的驱动自定义结构体的部分定义如下所示:

```

1 typedef struct cp210x_dev
2 {
3     DEV_HDR cp210xDevHdr; /*must be first field*/
4     LINK devHdrLink; /*linked list of devhdr structs*/
5     ...
6
7     char *writeBuf;
8     int writeFront;
9     int writeRear;
10    char *readBuf;
11    int readFront;
12    int readRear;
13 } CP210X_DEV, *pCP210XDEV;

```

3.4.2 驱动初始化

与单设备驱动程序中的驱动初始化流程不同的是,对于多设备驱动在初始化的时候我们不需要先注册设备和初始化输入输出缓冲区,在多设备驱动初始化的时候我们只需要将驱动程序注册到系统驱动表即可,不需要将其注册系统设备表,注册系统设备表需要在设备初始化时完成,除此之外我们还需要将驱动向 USBD 层进行注册多设备下的驱动初始化流程如图 3-21所示。

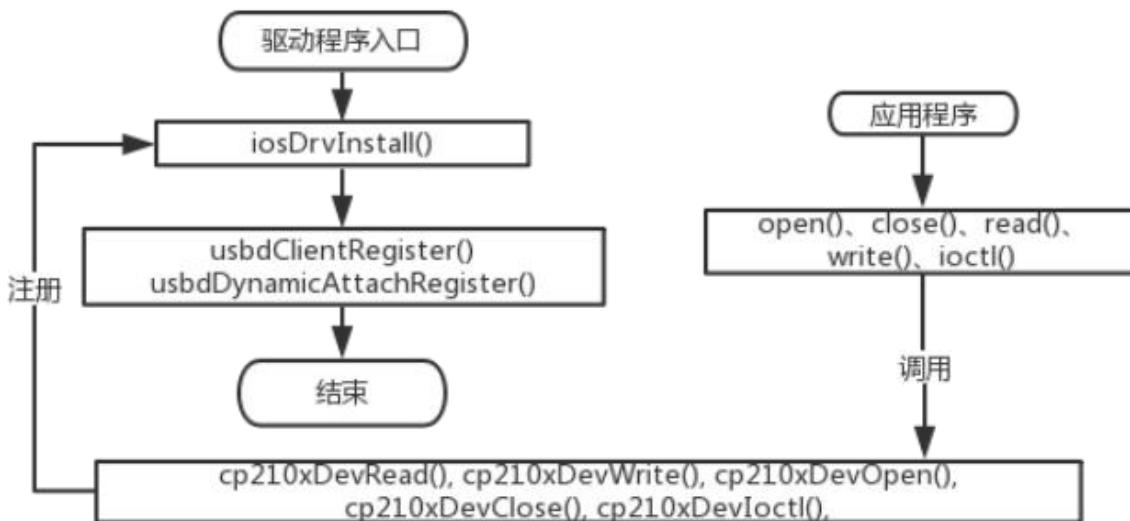


图 3-21 多设备驱动初始化流程

3.4.3 设备的识别和初始化

设备的初始化工作同样是从设备插入系统之后的回调函数开始的,在驱动初始化时我们注册好了热插拔的回调函数 `cp210xAttachCallback()`,和单设备驱动中的识别过程一样,我们通过发送标准的 USB 设备请求命令来获取该设备的设备描述符,然后通过设备的 VID 和 PID 来识别设备,多设备驱动中设备识别和初始化流程如图 3-22所示。

比较单设备驱动中设备初始化(如图 3-13所示)和多设备驱动中设备初始化(如图 3-22所示)的过程,我们可以看出在驱动注册过程中两者的区别,在单设备驱动的设备初始化中我们无需在完成设备添加到系统设备表的过程,因为这一过程已在驱动初始化当中完成。而多设备驱动中设备初始化的过程需要识别每一个设备并给支持的设备分配一个设备名和设备的自有资源,然后再将该设备添加到系统设备表当中。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,使用的设备名诸如“/UsbSerial/0”,“/UsbSerial/1”...。

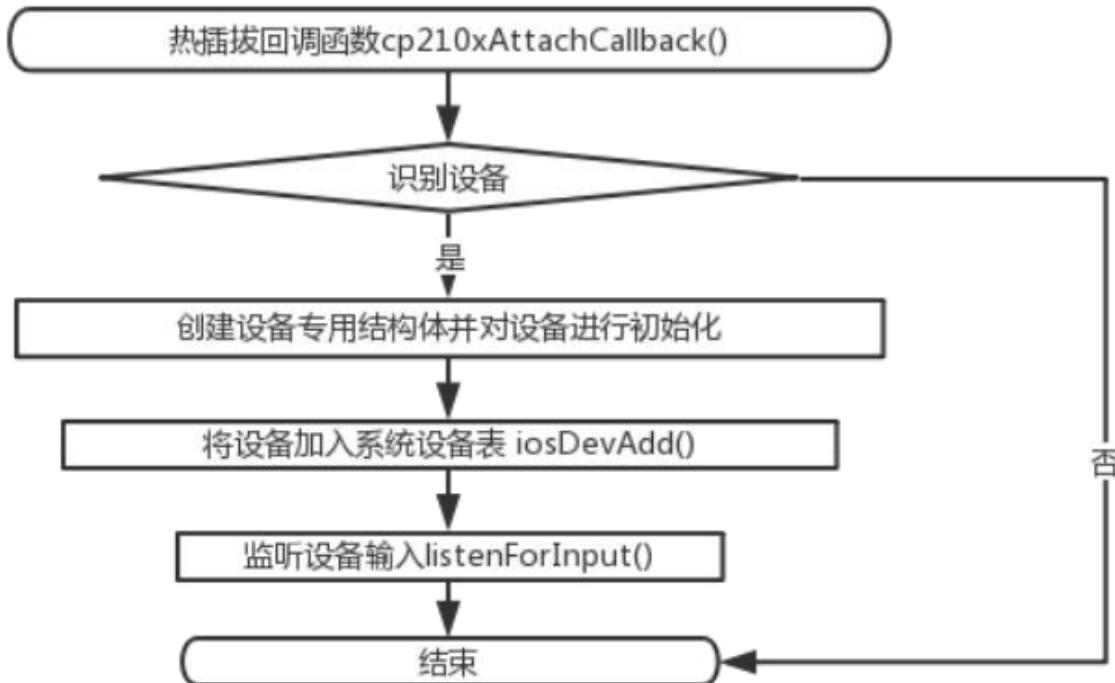


图 3-22 多设备回调函数流程

3.4.4 设备的打开/关闭

通用多设备驱动相对于我们上述的单设备驱动而言在打开/关闭操作的不同之处在于需要在打开或者关闭之前判断设备是否仍然连接在系统上,因为设备可能未连接在系统上或者在 `open()` 之后已经拔出了设备。因为对于通用多设备驱动而言不存在“假”的设备,不能在设备未连接时往缓冲区当中写数据,所以我们需要在执行打开/关闭操作的函数内首先判断设备是否仍然处于连接状态。除此之外只需要记录下设备打开和关闭的次数即可。

3.4.5 设备的读写

在多设备的驱动内部我们会为每一个连接到系统上的设备建立一个输入循环缓冲区和一个输出循环缓冲区,分别用于从设备接收输入数据和接收上层应用的输出数据。每个设备的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_BUFFER_SIZE` 和 `READ_BUFFER_SIZE`。

对于通用多设备的写操作,与单设备驱动的写操作不同的是在设备连接上时,没有一个自动发送缓冲区的数据的过程,因为对于多设备驱动而言在设备连接上之前是不可能往缓冲区中写入数据的,只有在上层应用调用 `write()` 操作之后才会往该设备的缓冲区中写入数据,并触发数据的发送操作。其基本流程如图 3-23 所示。

对于上层应用的 `write()` 操作,我们首先判断该设备是否仍然处于连接状态,若设备不处于连接状态则直接返回错误即可,若正处于连接状态则将数据拷贝到该设备的

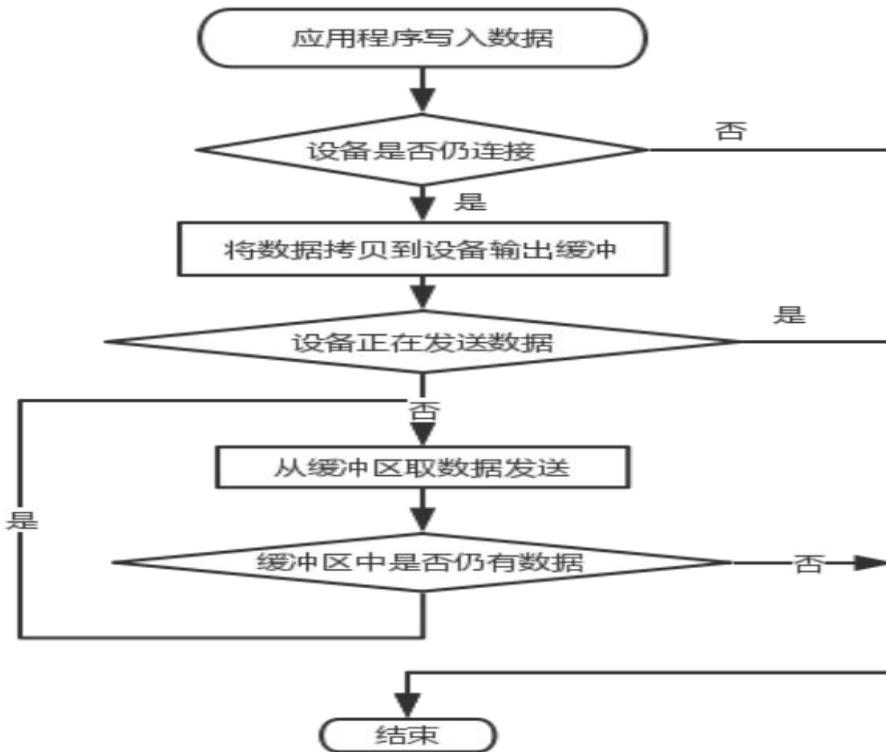


图 3-23 多设备驱动数据发送逻辑图

发送缓冲区,若当前设备正在进行数据的发送,则直接结束即可,若当前设备没有数据正在发送则触发数据的发送操作,设备在发送完一次的数据包之后会检查缓冲区中是否仍然有数据,若有数据则继续发送,若没有数据了则结束。

同样我们在多设备驱动下使用一个互斥信号 `cp210xWriteMutex` 对输出缓冲区进行控制,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,那么这时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。

多设备下的数据的输入操作的实现方式与单设备下的相同,都是在设备接入系统之后就向 `USBD` 层注册一个回调函数和一个 `IRP` 接收设备的输入,即 `listenForInput()`,这个函数的实现与单设备驱动当中的一致,然后在 `cp210xIrpCallback()` 函数当中处理设备的输入,并启动下一次的 `listenForInput()`,

3.4.6 驱动卸载

在多设备驱动当中驱动的初始化流程与所设备中的不同,此时多设备驱动的卸载流程与单设备的驱动卸载流程也有一些相异之处,我们多设备驱动的卸载流程如图 3-24 所示。与多设备驱动不同之处在于此处为每一个连接在系统上的设备执行 `iosDevDelete()` 操作,因此在单设备驱动中设备名是固定的,在驱动初始化时已经加载到系统设备表当中。而多设备驱动中设备名是动态的,需要根据插入系统中的设备数

来决定,需要在设备初始化完成之后才将其加入系统设备表当中,在清理设备所占有的资源的时候同样要先等待设备当前的数据传输完成之后再清理。

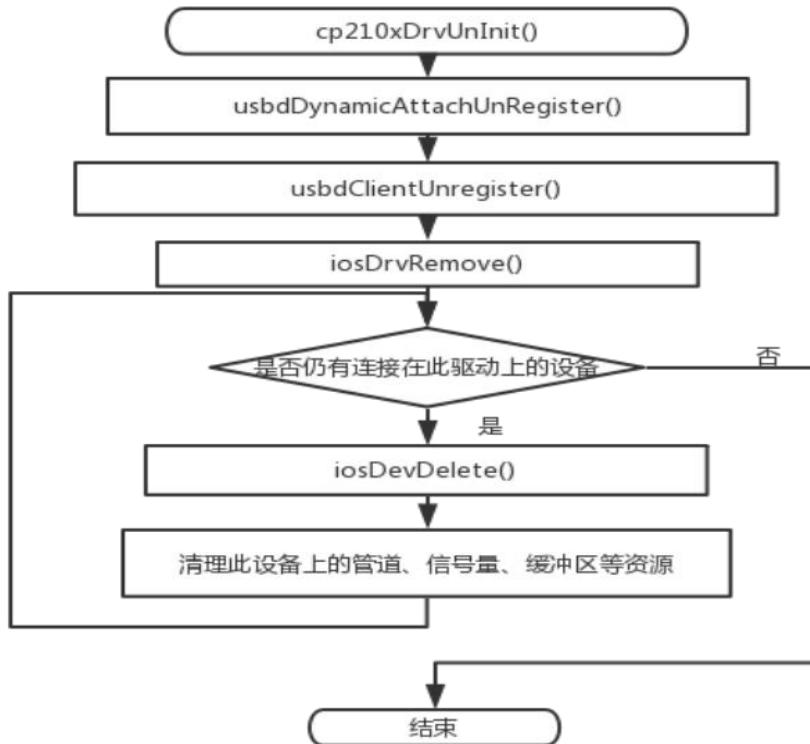


图 3-24 多设备驱动注销流程

对于多设备情况下的驱动的设备控制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。

3.5 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 VxWorks 下进行 USB 驱动开发的层次结构,USB 口转串口所使用转换器的选择和对其进行开发工作所必须的知识。接下来我们讲解了我们的驱动程序的具体的设计和实现的方式,包括驱动中要实现的模块,每个模块的功能是什么,如何实现这些模块。

四 应用层程序接口封装

由于 VxWorks 从 6.x 开始引入了 RTP(VxWorks Real Time Process Project) 模式, RTP 模式对用户而言增加了更多的权限限制,对于应用程序而言也有了较为明显的内核态和用户态的区别,数据需要在内核态和用户态之间进行内存拷贝,执行效率有所降低,不过对于现代 CPU 的速度而言,这点速度的降低并不是很明显,不过对于实时性要求很高的应用而言还是应该使用 task 的模式来构建程序。RTP 模式的优点是应用程序之间互相独立,这有利于增强内核的稳定性和安全性。对于我们而言其最直接的影响就是在 RTP 模式下有很多的内核函数都不能够再直接进行调用,而是需要使用封装的接口调用。

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 RTP 模式的引入,在这种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。

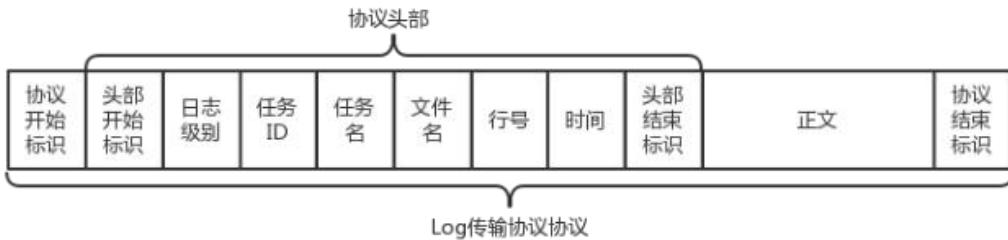


图 4-1 Log 协议字段

我们为调试信息制定的协议格式为:\03\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\04\04

其中各部分的含义如下：

- \03\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。
- \04\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。

在 RTP 模式下使用 dup/dup2 函数来实现标准输出的重定向, 首先需要指定一个

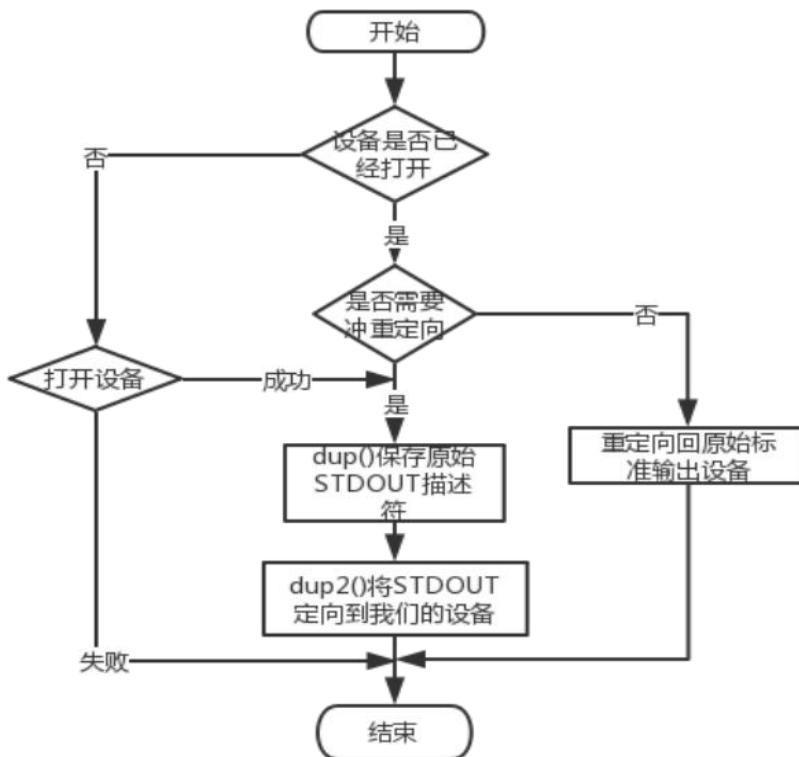


图 4-2 RTP 模式下标准输出重定向流程图

需要重定向到的设备,然后判断该设备是否已经打开,若设备已打开则判断其是要重定向到远程终端还是重定向到本地的标准输出,重定向到远程设备时需要先使用 `dup` 来保存原始的标准输出,方便之后重定向会本地的标准输出,然后使用 `dup2` 将标准输出重定向到远程设备,若设备未打开则先将指定的重定向设备打开再执行重定向操作。

`dup2` 函数的原型为:

```
1 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接口来实现重定向, task 模式下的标准输出重定向如图 4-3 所示。

`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:

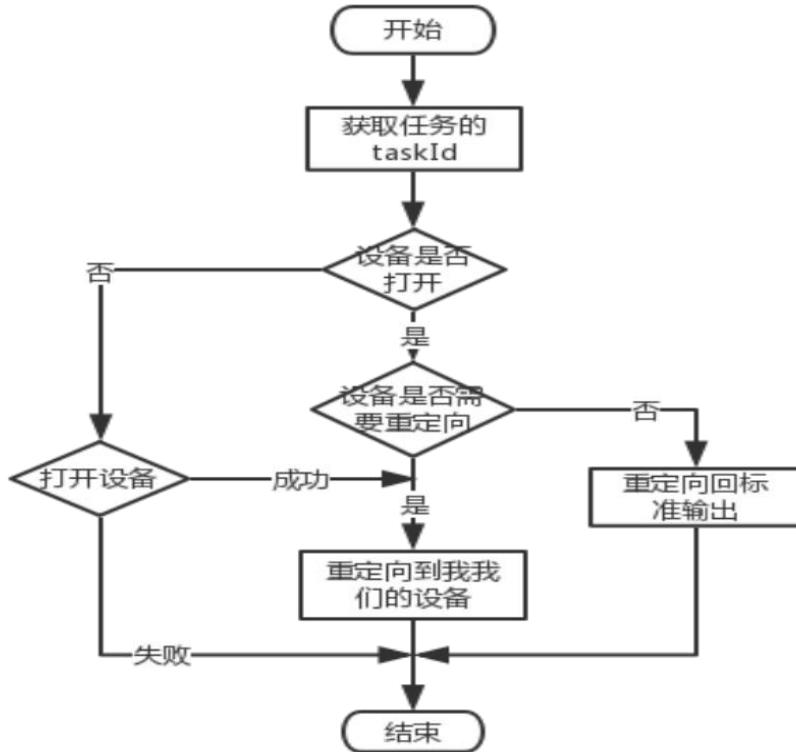


图 4-3 task 模式下标准输出重定向流程图

```
1 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 newFd, newFd 需要是一个文件或者设备的描述符。

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__)
6
7 #define LogW(format, ...) usb_logWrite('w', __FILE__, __LINE__, format, ##_VA_ARGS__)
```

```

8
9 #define LogO(format, ...) usb_logWrite('o', __FILE__, __LINE__, format, ##
10   __VA_ARGS__)
11 extern int usb_logWrite(char level, char *fileName, int lineNumber, const
12   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() 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

我们本次的设计当中会使用到一个自定义 Log 协议, 对与自定义的协议我们需要在 windows PC 端开发出自己的协议解析工具和用户界面。本次的工作中我们使用 QT 在 windows 下开发出了一个调试信息的分析界面其, 结构如图 4-4 所示。但是由于其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;



图 4-4 调试信息分析工具结构图

Log 协议解析流程 日志界面的串口读取流程图如图 4-5 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。

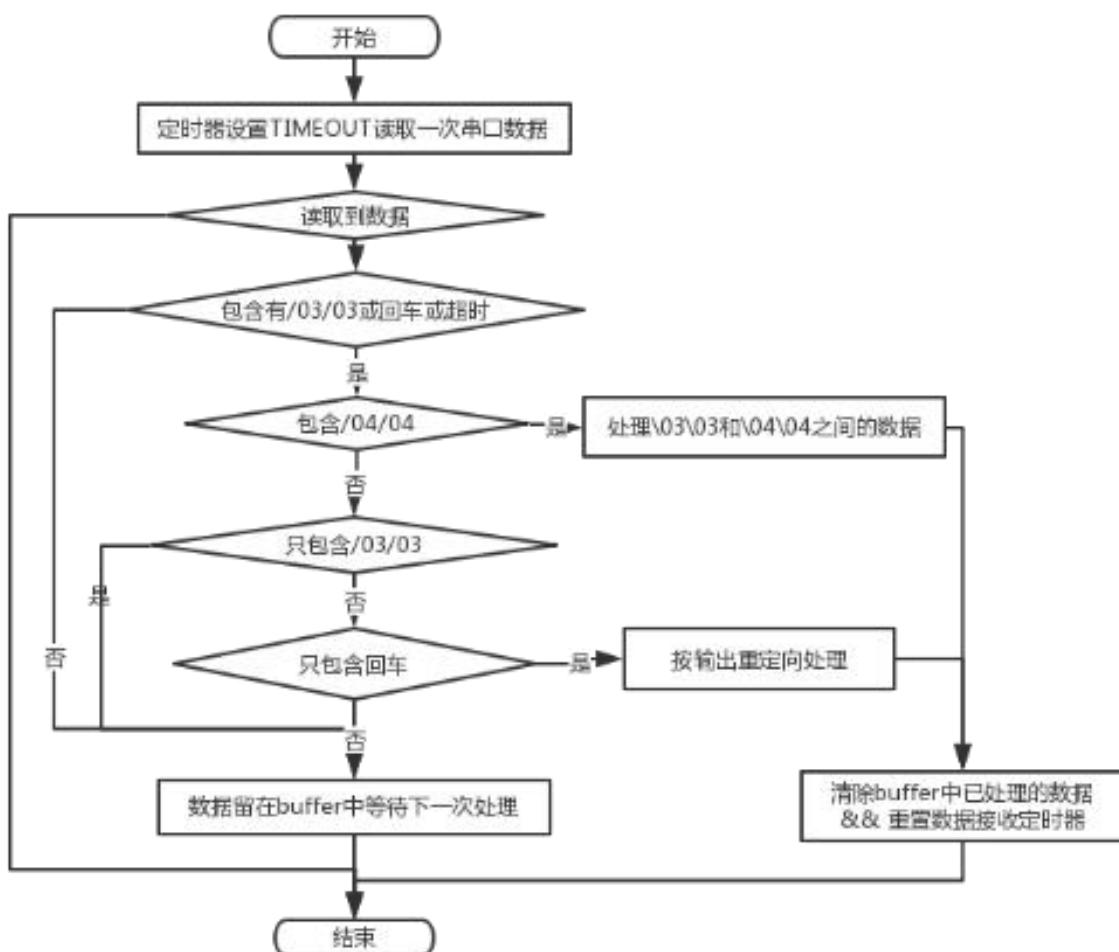


图 4-5 协议解析流程图

4.6 小结

本章首先介绍了我们的接口程序的设计分为了几个部分，每个部分的功能是什么，然后对我们的一个自定义的 Log 协议进行了介绍。最后分别在 RTP 模式和 task 模式下对接口进行了实现。

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。

```
-> iosDevShow
drv name
 1 /tyCo/0
 1 /tyCo/1
 2 /pcConsole/0
 2 /pcConsole/1
 3 /aiopipe/0x2e400620
 8 /
 11 /shm
 10 host:
 4 /ahci00:1
 5 /ahci01:1
 4 /ata0a
value = 0 = 0x0
->
```

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
->
-> iosDevShow
drv name
 1 /tyCo/0
 1 /tyCo/1
 2 /pcConsole/0
 2 /pcConsole/1
 3 /aioPipe/0x2e400620
 8 /
11 /shm
10 host:
 4 /ahci00:1
 5 /ahci01:1
 4 /ata0a
14 /hust_usb_serial
value = 0 = 0x0
->

(b) 多设备驱动
->
-> iosDevShow
drv name
 1 /tyCo/0
 1 /tyCo/1
 2 /pcConsole/0
 2 /pcConsole/1
 3 /aioPipe/0x2e400620
 8 /
11 /shm
10 host:
 4 /ahci00:1
 5 /ahci01:1
 4 /ata0a
14 /UsbSerial/0
value = 0 = 0x0
->

```

图 5-2 驱动加载后系统设备表

rv	creat	remove	open	close	read	write	ioctl	
0	0	0	19295a	192750	0	192748	192846	
1	1fc5f0	0	1fc5f0	1fc632	1fd429	1fd5be	1fc7f5	
2	1c9344	0	1c9344	0	1fd429	1fd5be	1c9550	
3	0	0	1faa8d	1fa39c	1fa6fe	1fa9d7	1fa74d	
4	1e8570	1e6f2c	1e6afa	1e5c97	1e5c0a	1e84fb	1e6f8b	
5	1d6fb3	0	1d6fb3	1d6ef4	1d6c85	1d6a09	1d66b1	
6	1fb20e	0	1fb20e	1fb1ad	1fb2f9	1fb2b9	1fafbb	
7	1fb266	0	1fb266	1fb142	1fb0db	1fb07e	1fae74	
8	1f1ff3	1f1c95	1f1ff3	1f1c70	1f1c58	1f1c40	1f1d75	
9	0	0	0	2b34a4	2b353e	2b3517	2b3565	
10	2b4a0d	2b4ecc	2b5d83	2b6ac4	2b67dc	2b587e	2b5b89	
11	0	201b29	201c7f	201a48	0	0	2017f0	
12	0	1ac3af	1ac732	1ac890	1ac9dc	1aca31	1ac8d6	
13	0	1ace20	1ad29a	1ad3eb	1ad592	1ad710	1ad431	
value = 50 = 0x32 = '2'								

图 5-3 驱动加载前系统驱动表

5.2 驱动程序读写测试

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 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。

5.2.1 RTP 模式下的读写测试

测试报告如表 5.1 所示。

```

-> iosDrvShow
drv   creat      remove     open      close      read      write      ioctl
0       0          0        19295a  192750      0        192748  192846
1     1fc5f0      0        1fc5f0  1fc632  1fd429  1fd5be  1fc7f5
2     1c9344      0        1c9344      0        1fd429  1fd5be  1c9550
3       0          0        1faa8d  1fa39c  1fa6fe  1fa9d7  1fa74d
4     1e8570  1e6f2c      0        1e5c97  1e5c97  1e5c0a  1e84fb  1e6f8b
5     1d6fb3      0        1d6fb3  1d6ef4  1d6c85  1d6a09  1d66b1
6     1fb20e      0        1fb20e  1fb1ad  1fb2f9  1fb2b9  1fafbb
7     1fb266      0        1fb266  1fb142  1fb0db  1fb07e  1fae74
8     1f1ff3  1f1c95      0        1f1c70  1f1c58  1f1c40  1f1d75
9       0          0        0        2b34a4  2b353e  2b3517  2b3565
10    2b4a0d  2b4ecc      0        2b5d83  2b6ac4  2b67dc  2b587e  2b5b89
11    0        201b29  201c7f      0        201a48      0        0        2017f0
12    0        1ac3af  1ac732      1ac890  1ac9dc  1aca31  1ac8d6
13    0        1ace20  1ad29a  1ad3eb  1ad592  1ad710  1ad431
14    0        0        bab177  bab18f  baae62  bab015  bab61c
value = 50 = 0x32 = '2'
->

```

(a) 单设备驱动

```

-> iosDrvShow
drv   creat      remove     open      close      read      write      ioctl
0       0          0        19295a  192750      0        192748  192846
1     1fc5f0      0        1fc5f0  1fc632  1fd429  1fd5be  1fc7f5
2     1c9344      0        1c9344      0        1fd429  1fd5be  1c9550
3       0          0        1faa8d  1fa39c  1fa6fe  1fa9d7  1fa74d
4     1e8570  1e6f2c      0        1e5c97  1e5c97  1e5c0a  1e84fb  1e6f8b
5     1d6fb3      0        1d6fb3  1d6ef4  1d6c85  1d6a09  1d66b1
6     1fb20e      0        1fb20e  1fb1ad  1fb2f9  1fb2b9  1fafbb
7     1fb266      0        1fb266  1fb142  1fb0db  1fb07e  1fae74
8     1f1ff3  1f1c95      0        1f1c70  1f1c58  1f1c40  1f1d75
9       0          0        0        2b34a4  2b353e  2b3517  2b3565
10    2b4a0d  2b4ecc      0        2b5d83  2b6ac4  2b67dc  2b587e  2b5b89
11    0        201b29  201c7f      0        201a48      0        0        2017f0
12    0        1ac3af  1ac732      1ac890  1ac9dc  1aca31  1ac8d6
13    0        1ace20  1ad29a  1ad3eb  1ad592  1ad710  1ad431
14    0        0        bad1b7  bad1cf  bacea2  bad055  bad65c
value = 50 = 0x32 = '2'
->

```

(b) 多设备驱动

图 5-4 驱动加载系统驱动表

```

-> iosFdShow
fd name
3 /pcConsole/0
4 /aioPipe/0x2e100620
5 (socket)
6 (socket)
7 (socket)
8 (socket)
9 (socket)
10 /ata0a/Script.txt
value = 100 = 0x64 = 'd'
->

```

图 5-5 当前系统上的文件描述符表

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

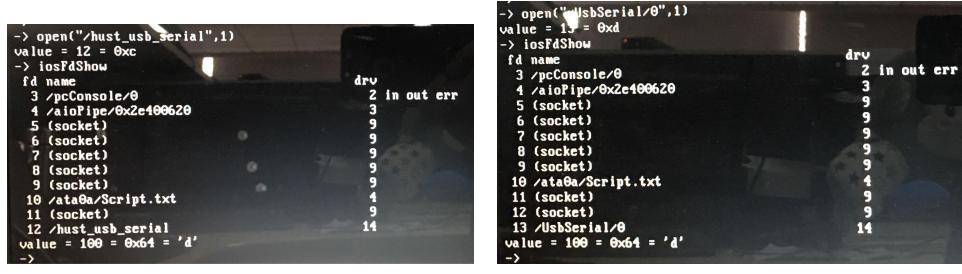


图 5-6 加载驱动后系统上的文件名文件描述符表

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

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

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。

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, VxWorks 下封装好的标准输出重定向接口以及 USB 转串口驱动, 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, 因为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, 如此循环 10000 次, 查看所有的输出是否显示在正确的终端上。测试结果如下:

波特率	数据位	数据位	数据方向	发送周期	发送次数	正确率
9600	8	0	发送到本地终端	0.1S/次	10000	100%
9600	8	0	发送到远程终端	0.1S/次	10000	100%

表 5.3 task 模式重定向测试

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。注意这里的两个程序的区别, RTP 模

波特率	数据位	数据位	数据方向	发送周期	发送次数	正确率
9600	8	0	发送到本地终端	0.1S/次	10000	100%
9600	8	0	发送到远程终端	0.1S/次	10000	100%

表 5.4 RTP 模式重定向测试

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。

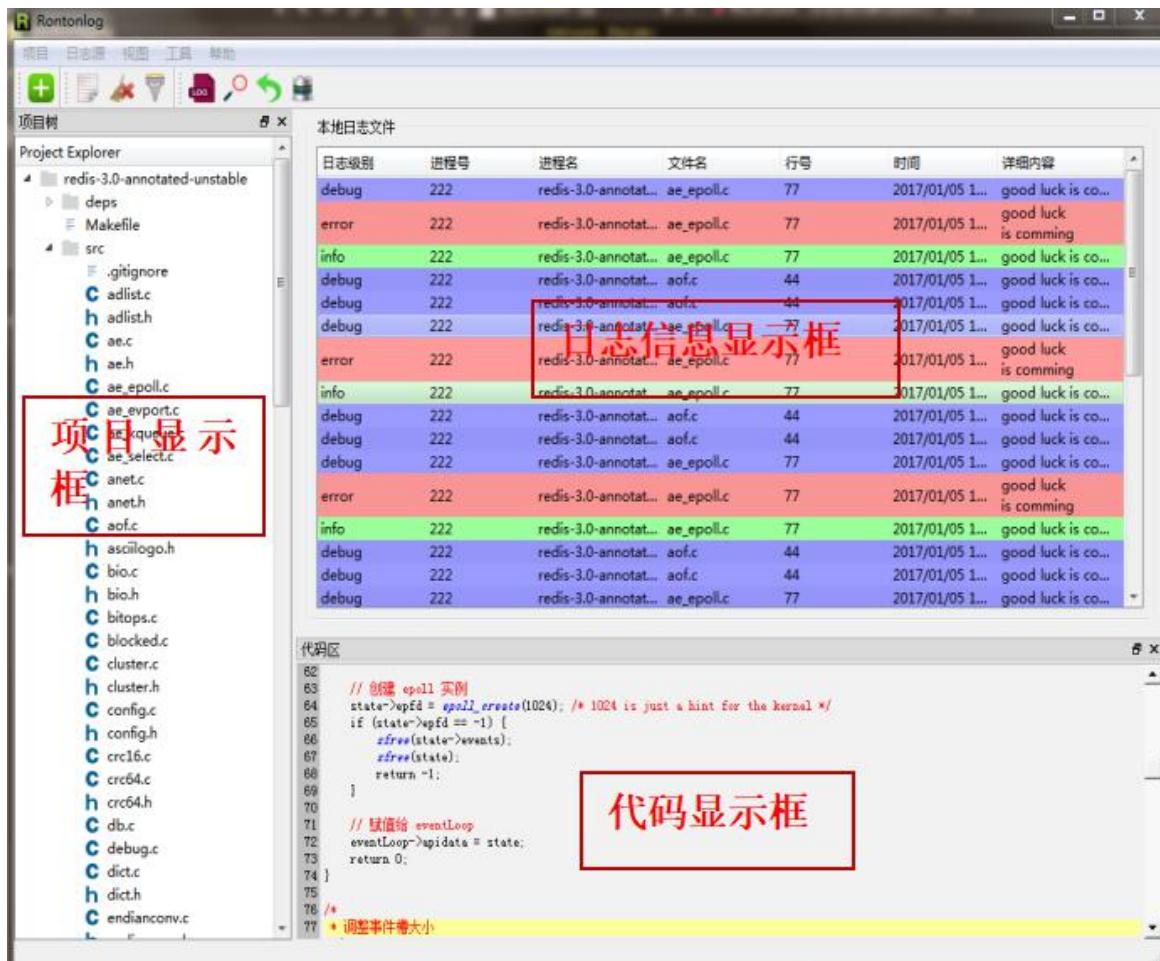


图 5-7 主机端日志分析工具测试界面

测试报告如表 5.5 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 5.5 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。

5.4 小结

本章对我们的驱动程序进行了一个整体的功能测试和各个部分的功能测试，查看系统的各个部分是否能够正常的工作以及整个系统的能够正常工作，是否能够达到设计的要求。通过测试结果我们可以看到系统已经基本达到了本次的设计要求。各个模块都能够正常、稳定的运行。

六 总结与展望

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** 下开发项目实际经验。

通过本次的项目我了解到了 **VxWorks** 的强大之处,其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之大。**USB** 凭借其优异的性能一定会成为更多外设的通用标准,同时用户也会对 **USB** 的带宽、据传输速率提出越来越高的要求,因此以后的 **USB** 技术肯定会更加的强大和完善,随着 **USB OTG**、**type C** 等标准的颁布, **USB** 的应用领域和实用场景进一步得到强化。

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。

致 谢

转眼两年的研究生生活就要结束了，在此要感谢在这两年的学习生活当中教导我的老师和陪伴我一起成长的同学，是他们陪伴着我在科研的路上一直奋斗，他们不断地激励着我、鼓励着我前进。同时还要感谢论文的评阅人员，谢谢你们为我研究生生涯进行最后的把关、检验工作。本此的调试通道的设计工作能够顺利完成，还要特别感谢我的导师张杰老师，张老师从论文选题，构思到最后定稿的每个环节都一直给予我意见、指引与教导，在整个研究生生涯当中张老师都非常严格的要求我，这让我在研究生期间学习到了更多的知识。也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！

参考文献

- [1] 刘小军. 基于 VxWorks 的实时仿真系统软件设计与开发: [PhD Dissertation]. 南京航空航天大学, 2008.
- [2] 陈洋, 胡向宇, 杨坚华. VxWorks 下的内存管理. 计算机工程, 2007, 33(8):94–96.
- [3] 张鹏. 基于 VxWorks 的无人直升机操纵控制系统设计与实现: [PhD Dissertation]. 南京航空航天大学, 2007.
- [4] Yao Weiqing, Zhou Wei. Design of USB-RS232 converter module based on FT2232H. Electronic Design Engineering, 2009.
- [5] Zhou Yiqing, Jiangmen. The Simplest Solution for USB/RS-232 Converter. Journal of Xianning Teachers College, 2002.
- [6] yz2010 的博客. VxWorks 内核解读. <https://blog.csdn.net/yz2010/article/details/68930013>. Accessed April 17, 2018.
- [7] 曹桂平. VxWorks 设备驱动开发详解. 电子工业出版社, 2011.
- [8] Wu Yurong, Wang Congling, Cai Jin. Implementation of IPTV STB Network Driver and Protocol for VxWorks Embedded System. in: Proceedings of International Symposium on Test Automation Instrumentation, 2008.
- [9] Zhang Zhen, Li Yan. 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.
- [10] Wind River System. VxWorks Driver API Reference 5.5. Inc, 2002.
- [11] Wind River System. VxWorks Programmer Guide. Inc, 2003.
- [12] An Junshe. Implementation of an Embedded System Based on VxWorks. Computer Engineering Applications, 2003.
- [13] 胡明民. 基于实时操作系统 VxWorks 的驱动程序开发: [PhD Dissertation]. 西安电子科技大学, 2012.
- [14] 冯云贺. 基于 Simics 全系统仿真环境的嵌入式系统的研究与开发: [PhD Dissertation]. 北京工业大学, 2014.
- [15] 李雪红. USB/RS232 接口转换器的设计: [PhD Dissertation]. 长安大学, 2004.
- [16] 莫宏伟, 柳泉, 赵文华, et al. USB 传输技术及其应用. 应用科技, 2001, 28(10):20–22.
- [17] 张杰. 基于 USB 主机的 USB-RS232 接口转换器的设计与实现: [PhD Dissertation]. 南京理工大学, 2008.
- [18] 边海龙, 贾少华. USB 2.0 设备的设计与开发. 人民邮电出版社, 2004.
- [19] 张念淮, 江浩. USB 总线接口开发指南. 国防工业出版社, 2001.
- [20] 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.
- [21] Labs S. CP2102 SINGLE-CHIP USB to UART BRIDGE, 2007.
- [22] 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.
- [23] 阳得常. 基于 Windows XP 下 USB 转 RS-232 接口转换器驱动程序的开发: [PhD Dissertation]. 西安电子科技大学, 2010.
- [24] 谢强. 基于 LINUX 的嵌入式操作系统实时性研究: [PhD Dissertation]. 西安电子科技大学, 2007.
- [25] 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.

- [26] Barbalace A., Luchetta A., Manduchi G. Performance Comparison of VxWorks, Linux, RTAI and Xenomai in a Hard Real-time Application. in: Proceedings of Real-Time Conference, 2007 Ieee-Npss, 2007, 1-5.
- [27] Kornecki Andrew J., Zalewski Janusz, Eyassu Daniel. Learning Real-Time Programming Concepts through VxWorks Lab Experiments. in: Proceedings of Software Engineering Education Training, 2000. Proceedings. Conference on, 2000, 294-301.
- [28] Neugass H., Espin G., Nunoe H. VxWorks: an interactive development environment and real-time kernel for Gmicro. Tron Symposium Proc Ghth, 1991, pages 196–207.
- [29] 郭春生, 朱兆达. 硬实时操作系统-RTLinux. 电子技术应用, 2002, 28(4):17–19.
- [30] Zhu DyYu, Yan Li, Qiang Wang. RTLinux-based software CNC system. Computer Integrated Manufacturing Systems, 2004, 10(12):1571–1576.
- [31] 解月江, 张梅. VxWorks 下设备驱动技术研究. 航天控制, 2004, 22(6):54–57.
- [32] 马增炜, 马锦儒, 李亚敏. 基于 WIFI 的智能温室监控系统设计. 农机化研究, 2011, 33(2):154–157.
- [33] Yi-Hua F. U., Mei Shunliang. Implementation of 100Mbps Ethernet based on VxWorks in embedded system. Journal of Computer Applications, 2006, 26(9):2190–2191.
- [34] 张军. 基于 Vxworks 实时操作系统的串口通信程序设计与实现. 微计算机信息, 2006, 22(5):98–99.
- [35] 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.
- [36] 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.
- [37] Ghosh Kaushik, Mukherjee Bodhisattwa, Schwan Karsten. A Survey of Real-Time Operating Systems – Draft. Georgia Institute of Technology, 1993.
- [38] 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.
- [39] Xu Meirong, Cai Ming, Dong Jinxiang. Design and Implementation of CAN Driver Based on Real-Time Operation System VxWorks. Computer Application and Rearch, 2006, 23(5):185–188.
- [40] Barbalace A, Luchetta A, Manduchi G, et al. Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in a Hard Real-Time Application. IEEE Transactions on Nuclear Science, 2008, 55(1):435–439.
- [41] 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.
- [42] Wind River System. USB Developer's Kit Programmer's Guide 1.1.2. Inc, 2003.
- [43] Wind River System. Tronado BSP Training Workshop. Inc, 1999.
- [44] Wind River System. Tronado Device Driver Workshop. Inc, 1999.
- [45] Wind River System. VxWorks BSP Developer Guide. Inc, 2004.
- [46] Wind River System. VxWorks Network Programmer's Guide 5.4. Inc, 2003.
- [47] 周启平, 张杨. VxWorks 程序员速查手册. 机械工业出版社, 2005.
- [48] Jan Axelson, 陈逸. USB 大全. 中国电力出版社, 2001.
- [49] 李肇庆, 韩涛. 串行端口技术. 国防工业出版社, 2004.