

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 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 : Joe

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:

日期: 年 月 日

指导教师签名:

日期: 年 月 日

## 摘 要

数据传输是现代通讯过程中的一个重要环节,在数据的传输过程中,不仅仅要求数据传输的准确率高,而且要求速度快,连接方便。传统的 RS232 串口通讯和并口通讯都存在传输速度低,扩展性差、安装麻烦等缺点,而基于 USB 接口的数据传输系统能够较好的解决这些问题。目前 USB 接口以其传输速率高、即插即用、支持热插拔等优点,逐步成为 PC 机的标准接口。

本文中的数据传输系统采用了 USB 接口进行上位机与下位机之间的数据通讯。下位机采用的是 VxWorks 实时操作系统。

**关键词：** 实时操作系统,设备驱动,USB 口转串口,VxWorks,CP2103

## Abstract

This is a  $\text{\LaTeX}$  template example file. This template is used in written thesis for Huazhong Univ. of Sci. & Tech.

This template is published under LPPL v1.3 License.

**Key words:**  $\text{\LaTeX}$ , Huazhong Univ. of Sci. & Tech., Thesis, Template

## 目 录

摘要	I
1 绪论	1
1.1 课题背景以及意义	1
1.2 国内外概况	2
1.3 论文的主要内容和组织结构	4
2 系统分析	5
2.1 实时操作系统	5
2.2 VxWorks 简介	6
2.3 集成开发环境 Tornado	12
2.4 USB 简介	13
2.5 串口通信	17
2.6 软件需求	20
3 驱动程序的设计和实现	23
3.1 VxWorks 设备驱动概述	23
3.2 内核驱动的相关结构	26
3.3 USB 转串口设备驱动程序的实现	30
4 应用层程序接口封装	34
4.1 Log 协议的设计	34
4.2 标准输出重定向接口的设计	35
4.3 Log 接口的设计	37
4.4 Windows 下的日志分析工具	39
5	41
5.1 概述	41
致谢	42
参考文献	43
附录 A 攻读学位期间发表的学术论文	43
附录 B 这是一个附录	44

## 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。嵌入式系统的最典型的特色是它同人们的日常生活紧密相关。小到 MP3、PDA 等微型数字化设备。大到信息家电、智能电视、车载 GPS 等形形色色运用了嵌入式技术的电子产品。目前各种新型嵌入式设备在数量上已经远远超过了通用计算机。在嵌入式设备发展的 30 多年的历史中,嵌入式技术从来没有像现在这样风靡,人类也从来没有像现在这样享受嵌入式技术带来的便利。

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卓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,如卫星通讯、军事演习、弹道制导、飞机导航等。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,方便编程人员对系统进行分析、调试。同时基于应用程序的可移植性的考虑,VxWorks 在其内核 Wind 中也支持 POSIX 1003.1b 的规定和 1003.1 中有关基本系统调用的规定,包括:过程初始化、文件与目录、I/O 初始化、语言服务、目录处理;而且 VxWorks 还支持 POSIX 1003.1b 的实时扩展,主要包括:异步 I/O、记数信号量、消息队列、信号、内存管理和调度控制等<sup>[2]</sup>。当然有些 POSIX 接口也有 VxWorks 下的实现重新实现版本,比如定时器、二进制信号量、消息队列等。VxWorks 下专用的实现版本和 POSIX 兼容的实现版本在性能上有一些差异,POSIX 的接口主要是为简化 Linux 下程序的移植而设立的。对于没有 POSIX 接口的部分则必须将程序修改为 VxWorks 下的接口,对于修改了的代码在 VxWorks 下运行可能会出现各种问题,本文为了给出一个事后对程序进行分析的方法,于是设计了这个基于 VxWorks 下的调试通道。

目前开发人员碰到使用新的硬件方案,厂家基本上不提供 VxWorks 的 BSP,只支持 Linux。这个给开发人员增加了很大的开发成本,所有的程序跟模块都要重新设计,以适应新的系统,增加了开发周期。本文设计一种方案,可以很方便地将 VxWorks 的应用程序搬迁到 Linux 上,为厂家缩短开发周期,减少程序搬迁风险,大大降低了开发成本。本文先介绍 VxWorks 的技术原理,然后分析技术实现细节,最后拿一个应用程序作为例子。

## 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。

### 1.2.1 现有的 RTOS

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,在这个内核之上提供了最基本的系统服务,该内核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 为开发各类功能强大的信息设备而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<sup>[2]</sup>。Windows Embedded CE 以多种方式将一个虚拟的桌面计算机置于掌上或放置于口袋中,可以看做是 Windows98/NT 的微缩版。但是从技术的角度而言,Windows Embedded CE 并不能称得上是一个优秀的 RTOS,首先其很难实现产品的定制,其次它占用过多的 RAM 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,使用者可以根据自己的需要进行修改,同时由于其开源,在网上较易获得所需的资料和技术支持。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。

Linux 操作系统经过几十年的发展,其拥有大量的开发人员,成千上万的开源项目,成为了开发人员的首选。在移动领域借助 Android 与 IOS 平分天下。在嵌入式领域,近年来 CPU 主频的发展迅猛,CPU 主频的增加克服了 Linux 分时操作系统的实时缺陷,使得基于 Linux 的实时操作系统的精度已经完成可以满足大部分实时领域的要求,特别在民用领域。但是在强实时的工业控制,航天,军用领域当中,仍然需要强实时的专用操作系统 VxWorks。

VxWorks 操作系统是美国 WindRiver 公司于 1983 年推出的一种嵌入式实时操作系统 (RTOS),是一个运行在目标机上的高性能、可裁剪的实时操作系统,是一个专门为嵌入式实时系统设计开发的操作系统,其具有良好的持续发展能力、高性能的内核以及友好的用户开发环境,为开发人员提供了高效的实时多任务调度、中断管理、实时的系统资源以及实时的任务间通信。在嵌入式实时操作系统领域当中占有一席之地。VxWorks 支持 X86、PowerPC、ARM 等众多主流的处理器的,且在各种 CPU 平台上提供了统一的编程接口和一致的运行环境。在军事、航空航天、工业控制、通信等高精尖以及实时性要求极高的领域当中,有着更加广泛而深入的应用。应用实例包括 1997 年火星表面登入的火星探测器、爱国者导弹、飞机导航、F-16、FA-18 战斗机等。

### 1.2.2 VxWorks 下驱动程序的开发

VxWorks 作为一款嵌入式操作系统,为了屏蔽硬件的具体操作细节,为上层应用程序提供统一的接口,引入了设备驱动程序的概念。设备驱动程序是用来直接控制设备,以完成设备应有的操作。操作系统通过驱动程序来对设备进行操作,属于一种软件接口。这样,应用程序开发使用统一的软件接口,使得开发人员可以专注于应用程序的开发,不用考虑底层的物理设备。

VxWorks 操作系统下的驱动程序在其开发上有自身规范,且对于不同的设备的驱动程序开发也表现出较大的差异。嵌入式设备的硬件平台千差万别,厂商不可能遇见所有的设备并提供相应的驱动程序,因此要实现 VxWorks 的跨平台移植,用户必须根据硬件平台开发驱动程序。所以,开发 VxWorks 操作系统下外围设备的驱动程序具有很大的实际应用价值。

虽然 VxWorks 作为实时嵌入式系统有着无比强大的性能表现,但是就目前的现

状来看, VxWorks 操作系统的平台成本很高, 因为 VxWorks 是一个专用系统, 使用这个系统的单位或公司都是处于军事, 航天等特殊行业当中, 这使得他们的开发经验和资源不能够拿出来共享, 导致 VxWorks 的参考资料比较少, 没有成熟活跃的社区支持, 且 VxWorks 并不是一个开源的系统, 需要花不菲的价格进行购买和售后支持, 这些都提高了 VxWorks 的应用的开发门槛, 使得一般的开发人员不能够方便的接入到对其的应用研究当中, 增加了开发难度。

### 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高速的、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、国内外的状况以及本文的内容的安排。

第二章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、USB、串口和 VxWorks 的开发工具 Tornado。

第三章介绍了 VxWorks 下的驱动程序的开发和 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 系统分析

本次设计的工作包括一个 USB 口转串口的驱动部分、基于 USB 口转串口驱动的应用层接口部分、windows 下的调试信息分析界面部分。其中的重点在于本论文将要描述的基于 VxWorks 的调试通道部分,包括驱动部分和应用层的接口部分。驱动程序是调试通道的核心部分,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它部分相联系、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,因此要编写 VxWorks 下的驱动程序必须先对 VxWorks 以及 VxWorks 下与驱动编写相关的部分有一定的了解。

### 2.1 实时操作系统

实时操作系统 (Real Time Operation System, 简称 RTOS) 是整个实时系统的核心。POSIX1003.1 标准为 RTOS 下了一个简单的定义:RTOS 是能够在有限的响应时间内为应用提供所要求级别服务的操作系统<sup>[2]</sup>。当外界事件或者是数据产生时,RTOS 需要快速的进行处理,并且处理的结果又能够在规定的时间之内来控制生产过程或者对处理系统做出快速的响应,调度一切可利用的资源来完成实时任务。实时系统按照实时的效果可以分为软实时和硬实时,硬实时要求在规定的时间内必须完成操作,这是通过操作的在设计的时候就得到保证的;软实时只需要按照任务的优先级,尽可能快的完成任务即可。

一个实时操作系统的特征通常包括以下几点:

- **高精度计时系统**

计时精度是影响实时性的一个重要因素,在实时系统当中,经常需要精确确定实时地操作某个设备或执行某个任务,或精确的计算一个时间函数。这些不仅仅依赖于一些硬件提供的时钟精度,也依赖于实时操作系统的高精度计时功能。

- **多级中断机制**

中断是实时操作系统其中的一个关键设施,是用于通知系统发生外部事件的常用机制。一个实时操作系统通常需要处理多种外部信息或事件,但处理的紧迫程度有轻重缓急之分。有的必须立即作出反应,有的则可以延后处理。因此,需要建立多级中断嵌套处理机制,以确保对紧迫程度较高的实时事件进行及时响应和处理。

- **实时调度机制**

实时操作系统不仅要及时响应实时事件中断,同时也要及时调度运行实时任务。但是,处理机调度并不能随心所欲的进行,因为涉及到两个进程之间的切换,只能在确保“安全切换”的时间点上进行,实时调度机制包括两个方面,一是在调

度策略和算法上保证优先调度实时任务;二是建立更多“安全切换”时间点,保证及时调度实时任务。

本次所需完成的调试通道正是基于一个目前业界有名的实时操作系统 VxWorks。

## 2.2 VxWorks 简介

VxWorks 操作系统是美国 Wind River System 公司于 1983 年推出的一个运行在目标机上的高性能、可裁剪的实时操作系统 (RTOS), 该系统专门为嵌入式实时系统领域而设计开发, 其具有良好的持续发展能力、高性能的内核以及友好的用户开发环境, 为开发人员提供了高效的实时多任务调度、中断管理、实时的系统资源以及实时的任务间通信。其拥有 400 多个相对独立的短小而精炼的目标模块, 并且拥有多达 1800 个功能强大的应用程序接口<sup>[7]</sup>。

VxWorks 支持广泛的工业标准, 如 POSIX1003.1b 实时扩展、ANSI C(浮点支持) 和 TCP/IP 网络协议等。这种广泛的协议支持在主机和目标机之间提供了无缝的工作环境, 任务可以通过网络箱其他系统的主机存取文件, 即远程文件存取, 也支持远程的过程调用。这些标准也促进了多种不同产品之间的互通性, 提升了可移植性。同时因为其模块间的相对独立性, 使得其拥有高度的灵活性, 用户可以根据自己的实际需求选择适当的模板来裁剪和配置系统, 对该操作系统进行重新定制或作适当的开发, 以满足自己的实际应用。而系统的连接器可以按照应用的需要自动链接一些目标模块。这样, 通过目标模块之间的按需组合, 可以得到许多满足功能需求的应用。

VxWorks 采用微内核设计, 支持多种硬件环境, 包括 X86、PowerPC、ARM 等众多主流的处理器的, 同时还支持 RISC、DSP 技术, 且在各种 CPU 平台上提供了统一的编程接口和一致的运行环境。VxWorks 凭借着其优异的性能在军事、航空航天、工业控制、通信等高精尖以及实时性要求极高的领域当中, 有着更加广泛而深入的应用。应用实例包括火星探测器、爱国者导弹、飞机导航、F-16、FA-18 战斗机等<sup>[7]</sup>。自从对我国的销售解禁之后, VxWorks 大量的应用于我国的军事、国防工业当中。

VxWorks 的基本构成部件由 5 个部分组成: 板级支持包、wind 内核、网络系统、文件子系统、IO 系统。其结构如图 2-1 所示。

在本次的设计当中会使用到的部分包括 VxWorks 的 wind 内核部分和 IO 子系统部分, 因此有必要介绍这两部分的相关内容。

### 2.2.1 微内核 wind

现代嵌入式操作系统的一个发展趋势是尽可能的从操作系统当中去掉冗余的服务, 保证内核的高效、精简, 只留下一个很小的内核, 由用户进程来实现大多数操作系统的功能如文件服务、进程服务、存储服务等, 即所谓的微内核操作系统, 从而使得系统具有模块化、结构清晰、可移植性强和可裁剪性好的特点。在微内核操作系统当中, 设备驱动程序通常作为核外可裁剪的部分存在。对于操作系统内核来说, 核外驱



图 2-1 VxWorks 系统结构

动就相当于一个普通的应用程序,因此对于系统内核的影响很小,编写和调试都很方便,同时可以避免数据流在不同级间的拷贝。微内核的设计思想首先在 CMU 开发的 MACH 操作系统当中得到了成功的应用,并使得微内核结构成为操作系统研究的热点<sup>[2]</sup>

VxWorks 也是基于微内核的思想设计的操作系统,其微内核被称为 wind。内核作为操作系统的核心,负责控制这计算机上的所有硬件和软件资源,在必要的时候给应用程序分配硬件资源,并执行相应的操作命令。wind 内核的主要功能如下:

- 系统内存管理
- 任务调度
- 任务间同步和通信
- IO 管理
- 文件和网络系统管理

Wind 的主要特点在于高效的任务管理(提供无限数量的的多任务,具有 256 个优先级,灵活快速的调度机制)、方便的任务间通信(提供多种信号量,网络通信以及共享内存等)、高度可裁剪(内核最小可以裁剪到 8K)。由于 VxWorks 操作系统模块化非常好,模块间的耦合度非常低,每个模块对外提供都是单独的头文件,比如任务调度,其头文件为 taskLib.h,任务通讯如果用的队列,那其头文件就是 msgQLib.h,如果是定时器管理,那其头文件就是 timerLib.h,因此也让程序移植提供了很大的便利性。有很多人认为,VxWorks 跟 Linux 操作性的系统头文件差异化太大,因此移植难度成倍速增加,其实不然,就是由于 VxWorks 的高度组件化,让程序移植提供了很大的便利性。

### 2.2.2 wind 的任务调度

VxWorks 下的任务即通用操作系统下所述的进程,是内核的基本运行单位,wind 提供多任务环境,允许实时应用程序以一套独立任务的方式构筑(即程序可以由单个或多个任务来构成,每个任务是程序的一个实例),每一个任务拥有独立的执行线程和一套自己的系统资源,同时由于在 VxWorks 中对内存进行统一编址,在任务间进行资源共享也十分方便,而任务间的通信机制可以使得得这些任务的行为同步、协调。每一个开启的任务都有一个任务控制的数据结构来记录当前的任务的状态,供内核管理调度,这个控制块简记为 TCB。控制块里包含了当前的状态、优先级、要等待的事件或资源、任务程序码的起始地址、初始堆栈指针等任务的上下文。

操作系统的进程调度时机基本上都会选择在同一个地方:从内核状态退出之时,VxWorks 也一样。从内核状态退出的时机主要有两个渠道:系统调用和中断。VxWorks 与于通用操作系统的区别的关键点在于其必须能够迅速的响应外部中断,所以实时操作系统只是在响应速度上更快,而在响应时机上与通用操作系统并无区别。Vxworks 区别于通用操作系统的另一个显著的特点是其划分运行级别,应用层程序可以直接调用内核函数,而不需要通过软中断的方式从应用层进入到内核层。VxWorks 将任务被分为 256 个优先级,0 最高,255 最低;任务在被创建时就会被指定一个优先级,在运行时也可以通过 taskPrioritySet() 函数重新设定优先级。该函数的原型如下:

---

```

1 STATUS taskPrioritySet
2 (
3     int tid, /* task ID */
4     int newPriority /* new priority */
5 )

```

---

在 VxWorks 中任务调度是基于优先级的,而且是可抢占式的调度方式,这样才能够区分实际情况下的不同状态的处理级别,对高优先级的情况进行优先响应。VxWorks 下对于应用层任务,推荐使用 100-250 之间的优先级,驱动层任务可以使用 51-99 之间的优先级。VxWorks 中任务的状态通常有四种,这四种状态可以通过相应的函数控制其相互转化:

- **就绪态**:任务正在等待 CPU 资源,该状态下以优先级为序排列任务。
- **休眠态**:任务正在等待除 CPU 资源之外的其他资源,当竞争使用某个资源,而资源当前不可得,就进入这个状态。
- **延迟态**:任务正在等待一定时间的延时。当调用 taskDelay 函数让任务延迟一段固定的时间时,任务所处的状态,此时任务无资格竞争使用 CPU。
- **悬置态**:任务无法执行,主要是用于调试一种状态,这种状态仅影响任务的执行而不影响任务的转换。

如图 2-3 的 wind 内核任务调度框图展示了内核中构建了四种状态队列,通过 TCB 中任务状态的变化,由内核对其进行调度,并把就绪的任务加入到就绪队列等候

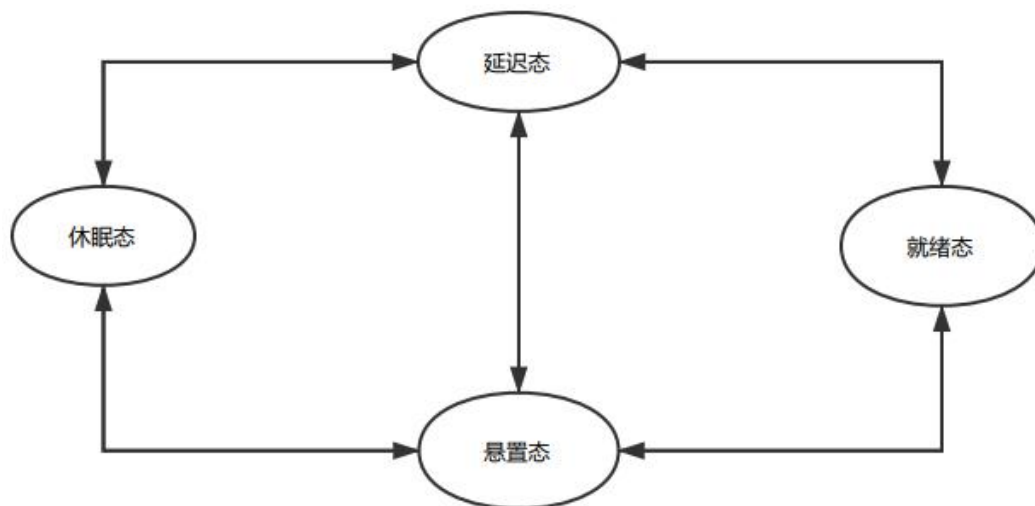


图 2-2 任务状态转换图

CPU 资源。

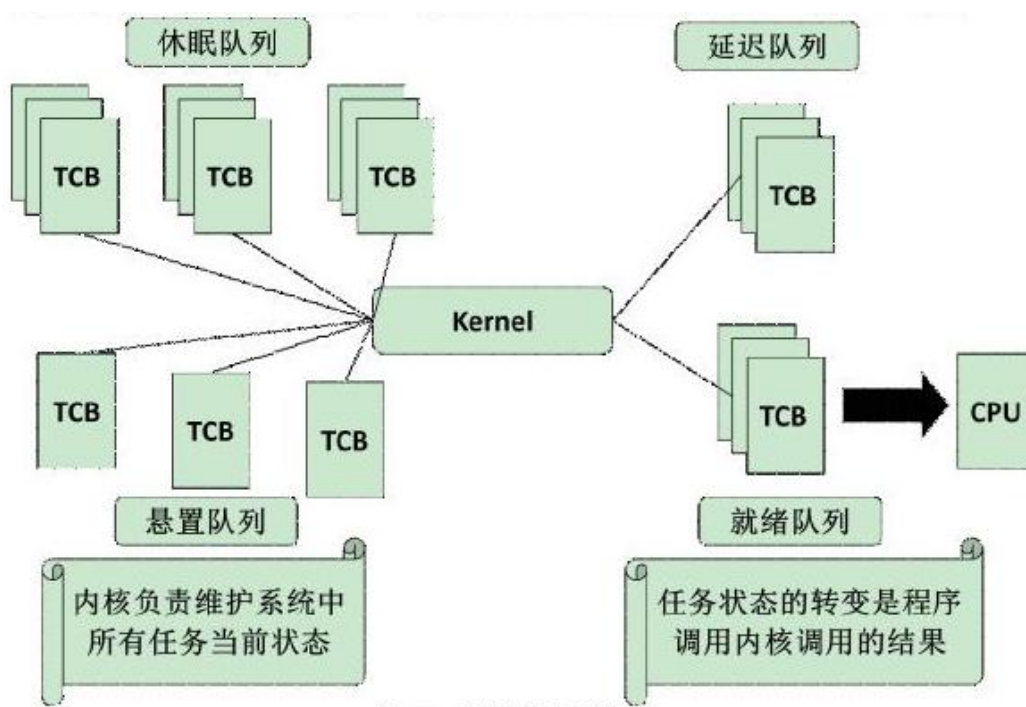


图 2-3 wind 任务调度

### 任务调度机制:

任务调度的基本算法包括:时间片轮转,优先级抢占和独占资源。**wind** 内核这三种算法都有实现,并允许用户自行进行配置。**wind** 将基于优先级的抢占调度作为缺省的调度策略,同级之间的任务依据时间片来进行 CPU 资源的分配。采用优先级抢占式调度算法符合实时系统的实际需求,紧急情况的出现需要及时得到响应。这样用户对于需要尽快处理的任务就可以设置一个更高的优先级,保证其时间需求,而相对无关紧要的任务则设置一个较低的优先级。



### 2.2.3 wind 任务间通信

任务间通信机制可以协调、同步任务之间的活动。在 VxWorks 中,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。这三种机制无一例外的都是在使用共享物理内存机制,只不过这块共享的内存由内核在进行管理,任务无法直接进行访问,而必须通过内核提供的接口函数进行访问,这就提供了一种保护和管理机制,使得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。互斥主要保护资源,即某个时刻只允许只有一个任务在使用该资源。同步则是任务间协同完成某一项共同工作的机制,典型的例子就是生产者和消费者进程,消费者平时处于等待状态,等待生产者完成其资源的生成,而一旦资源产生完毕,此时生产者就会触发同步信号量,让消费者的任务启动(即唤醒)其处理资源的工作。

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:

---

```

1 typedef struct semaphore
2 {
3     OBJ_CORE objCore; /*对象管理*/
4     UINT8 semType; /*信号量类型*/
5     UINT8 options; /*信号量选择*/
6     UINT16 recurse; /*信号量重复获取计数器*/
7     Q_HEAD qHead; /*阻塞的任务队列头*/
8     union{
9         UNIT count; /*当前状态*/
10        struct windTcb *owner;
11    }state;
12    EVENTS_RSRC events; /*VxWorks 事件*/
13 }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。消息队列对于



每个消息的大小存在限制,而且必须将信息分批打包,而管道可以像文件那样进行读写,是一种流式消息机制。管道在底层实现上是一种更为直接的共享物理内存机制。信号量和消息队列还需要对传递的数据进行某种方式的封装,则管道不会传递信息做任何包装,直接分配一块连续的内存空间作为任务间信息交互的中转站。传统意义上,管道分为两种:命名管道和非命名管道。通常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：**信号不是信号量,二者不是一个概念。信号量是一种任务间互斥和同步机制,而信号则用于通知一个任务某个事件的发生。一个信号产生后,对应任务暂停当前执行流程,转而去执行一个特定的函数进行处理。信号机制有些类似于中断,也可以将信号看作是一种用户层提供的软件中断机制(区别于 CPU 本身提供的软件中断指令)。这种用户层软件中断机制相比硬件中断和中断指令而言具有较长的延迟时间,即从信号产生到信号的处理之间大多将经历较长的延迟。Vxworks 下信号处理有些特别,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。

## 2.2.4 I/O 系统

通常为了实现与应用程序的平台无关性,操作系统都会为应用程序提供一套标准的接口,VxWorks 也不例外,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提高应用层开发的效率,避免重复编码。在我们的通用操作系统(如 Mac OS、Linux、Windows)当中,通常会将这套应用层的接口标准从操作系统中独立出来,专门以标准库的形式存在,这样可以屏蔽操作系统之间的差异,增强应用程序的平台无关性。

VxWorks 中为应用层提供了一套标准的文件操作接口,实际上与 GPOS 提供接口类似,我们将其称为标准 I/O 库,VxWorks 下由 ioLib.c 文件提供。ioLib.c 文件提供如下标准接口函数:creat()、open()、unlink()、remove()、close()、rename()、read()、write()、ioctl()、lseek()、readv()、writev() 等<sup>[2]</sup>,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。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

## 2.3 集成开发环境 Tornado

Tornado 是嵌入式实时领域里最新一代的开发调试环境,其系统结构如图 2-4所示。Tornado 提供了高效明晰的图形化的实时应用开发平台,可以帮助轻松的完成程序的编辑、编译、调试、系统配置等工作。它包括一整套完整的面向嵌入式系统的开发和调试工具。Tornado 采用的是主机—目标机的交叉开发模型,应用程序在主机的 Windows 环境下编译链接生成可执行文件,下载到目标机,通过主机上的目标服务器与目标机上的代理程序的通信完成对应用程序的调测、分析。这些工具包括 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。

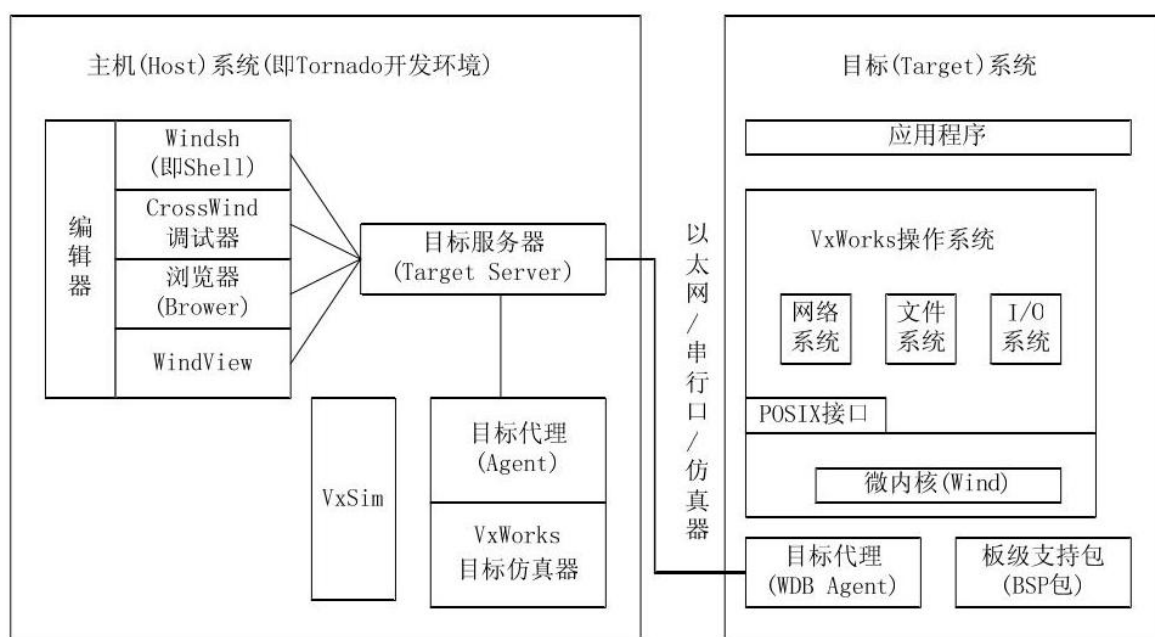


图 2-4 Tornado 开发系统结构

典型的主机开发系统通常有比较大的 RAM、硬盘空间、打印机以及其他外部设

备,而典型的目标机系统只有仅能满足实时应用的资源,除此以外可能还有比较少量的用于测试和调试的额外资源。**Tornado** 开发环境的基本优点是应用模块不需要链接到运行系统库。**Tornado** 直接装载重定位的目标模块,通过每个模块里的符号表来动态解析外部符号的引用,解析符号表是由运行在目标机上的目标服务器来完成的。**Tornado** 在开发过程中会把目标模块的大小减到最小,这样可以缩短开发周期。主机端驻留的 **shell** 和调试器也能够调用和测试独立的应用程序或者是完整的任务。

**Tornado** 是为开发 **VxWorks** 应用程序提供的集成开发环境,其中包含有工程管理软件,可以将用户自己的代码和 **VxWorks** 的核心有效的结合起来,可以按照用户的需要裁剪配置 **VxWorks** 内核;**Tornado** 开发系统包含有三个高度集成的部分:

- 运行在宿主机和目标机上的强有力的交叉开发工具和实用程序;
- 运行在目标机上的高性能、可裁剪的实时操作系统 **VxWorks**;
- 连接宿主机和目标机的多种通信方式,如以太网,串口线, **ICE** 或 **ROM** 仿真器等。

**Tornado** 主机集成开发环境中的主要工具为:

- **工程管理和配置工具**: 提供工作空间用于应用程序的组织和管理,可通过图形化的配置工具对 **VxWorks** 及其组件进行配置。
- **交叉编译器**: 提供 **GNU** 和 **Diab** 两种交叉编译器和类库
- **调试器**: 提供图形界面的调试方式,可以通过编辑窗口的右键菜单来设置断点和查看变量;
- **WindSh**: 一个驻留在主机端的命令行解释器,提供从主机端控制所有运行系统的接口
- **CrossWind 调试器**: 一个远程源码级的调试器,该调试器控制窗口综合了 **GDB** 命令行接口和 **WindSh** 工具。
- **浏览器**: 是一个系统对象的查看器,可以监视目标机的状态。
- **WindView 逻辑分析器**: 一个动态可视化工具,可以提供上下文切换的情况,事件和有关测量对象的信息。
- **VxSim 仿真器**: 用来模拟目标操作系统。

位于主机上的目标服务器和位于目标机上的目标代理通过 **WDB(Wind Debug)** 协议完成主机和目标机之间的通信。二者无论采取哪种链接方式,都是基于 **WDB** 协议的。

## 2.4 USB 简介

**USB(Universal 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 的主要有优点有:

1. 使用方便, 支持热插拔: 连接外设不必再打开主机箱, 而且方便携带, 允许在任何时候 USB 外设的热插拔, 而且不必关闭主机的电源; USB 外设没有需要用户选择的设置; 例如端口地址和中断请求线; 自动配置外设, 当用户连接 USB 外设到一个正在运行的系统时, 系统会自动的检测外设, 载入合适的驱动软件 (如果有的话), 并将其初始化; 以上的特点使得 USB 外设符合便携易用的潮流。
2. 速度快: 目前设备上使用大多数是 USB 2.0 的接口, 最新生产的设备都配备了 USB3.0 的接口。USB2.0 接口的理论速度可以达到 480Mbps(即 60MB/s), 并且可以向下兼容 USB1.1。USB3.0 接口的理论速度可以达到 5.0Gbps(即 640MB/s), 并提供 USB2.0 的兼容。
3. 连接灵活: 一个 USB 主控制器理论上可以连接 127 个设备

USB 规范规定了 USB 传输的四种方式, 每种方式有各自的用途<sup>[2]</sup>:

- **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<sup>[2]</sup>。

USB 的体系结构如图 2-5所示

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

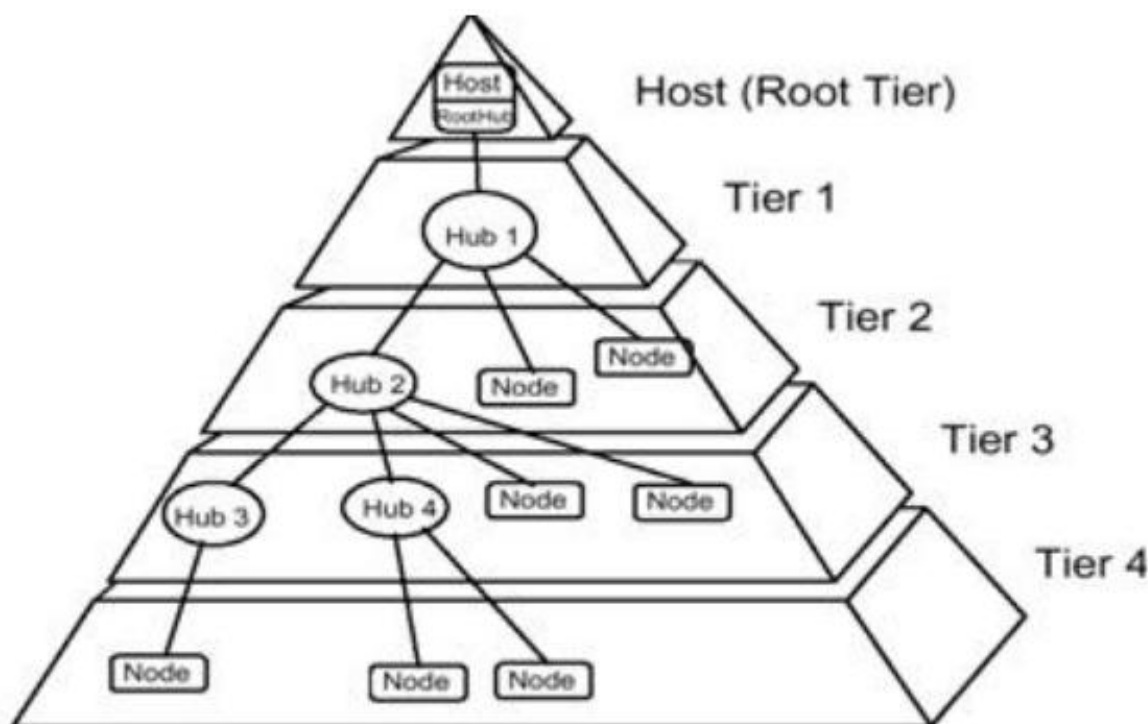


图 2-5 USB 总线拓扑结构

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其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，一个集线器通常会包括两个部分：集线控制器和集线放大器。集线放大器是一种在上游端口 v 和下游端口之间的协议控制开关。集线控制器提供接口寄存器用于和主机之间的通信，允许主机对其特定状态和控制命令进行设置，并监视和控制其端口。一种市场常见的集线器如图 2-6 所示

每个集线器的下游端口都可以连接另外的集线器或功能部件，最大的连接能力是 127，集线器可以检测每一个下游端口的设备的安装或移除，并可以对下游端口的设备分配资源，每一个下有端口都有独立的能力，不论是高速或者是低俗设备均可连接。集线器可以将低俗和高速端口的信号分开。

## 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



图 2-6 USB 集线器

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 and 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层程序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<sup>[2][3]</sup>。

## 2.5 串口通信

在生产实践和科学研究当中经常需要在上位机和下位机中进行数据和控制的传输,这种数据、控制传输都需要通过串口来进行,而现在生产的设备中大多都不再配置串口,仅仅保留了 USB 接口,于是需要通过一些手段将 USB 转换为串口,这时就出现了一些专用的 USB 转 UART 芯片,CP2102 就是这样一种芯片。CP2102 与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试通道的设计当中使用的芯片。

计算机与计算机之间或者是计算机与终端设备之间的通信方式有串口通信,并行通信以及网络通信等。使用串行通信方式的优点是成本低,使用线路少,而且可以解决由于不同的线路特性造成的问题,因此在远距离传输方面也是一种不错的选择。对于特性相异的设备要使用串行通信来连接时,必须保证双方所使用的的标准接口是一致的。一般的串口标准有 RS-232、RS-485、TTL 等。从通信的方向性来看,串口通信有单工、半双工和全双工三种方式。单工通常使用一根导线,通信只在一个方向进行,如监视器、打印机、电视机。半双工可以在两个方向上进行,但是方向切换时有时间延迟,如打印机。全双工可以在两个方向上运行,且时间的切换没有时间延迟,适用于那些不能有时间延迟的交互式应用,如远程监控等。

### 2.5.1 RS232 与 TTL

我们通常见到的串口有两种的物理标准,D 型 9 针(对应于 RS-232 标准)插头和 4 针(对应于 TTL 标准)的杜邦头,4 针的通常也会有第五根引脚-3.3V 电源引脚。RS232 电平标准用正负电压来表示逻辑状态,在 RS-232 标准中正电压是 0,负电压是 1。TTL 电平标准使用高低电平来表示逻辑状态,TTL 标准中低电平是 0,高电平是 1。使用 TTL 标准进行连接时,一般只会使用 GND、RXD、TXD 引脚,不会使用 VCC 或 3.3V 的电源线,避免与目标设备上的供电冲突。PL2303、CP2102 芯片是 USB 转 TTL 串口的芯片,用 USB 来扩展串口(TTL)电平。

### 2.5.2 CP2102 简介

CP2102 是 SILICON LABORATORIES 推出的 USB 与 RS232 接口转换芯片,是一种高度集成的 USB-UART 桥接器,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<sup>[21]</sup>,cp2102 的电路框图如图 2-7所示。

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,这些工作都会由芯片自动完成。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。在我们本次的使用当中只需要将其转换为 TTL 标准即可完成需求,不需要转换成 RS-232 电平,因此 CP2102 模块之外不再需要额外的转换为 RS-232 电平的芯片。

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。



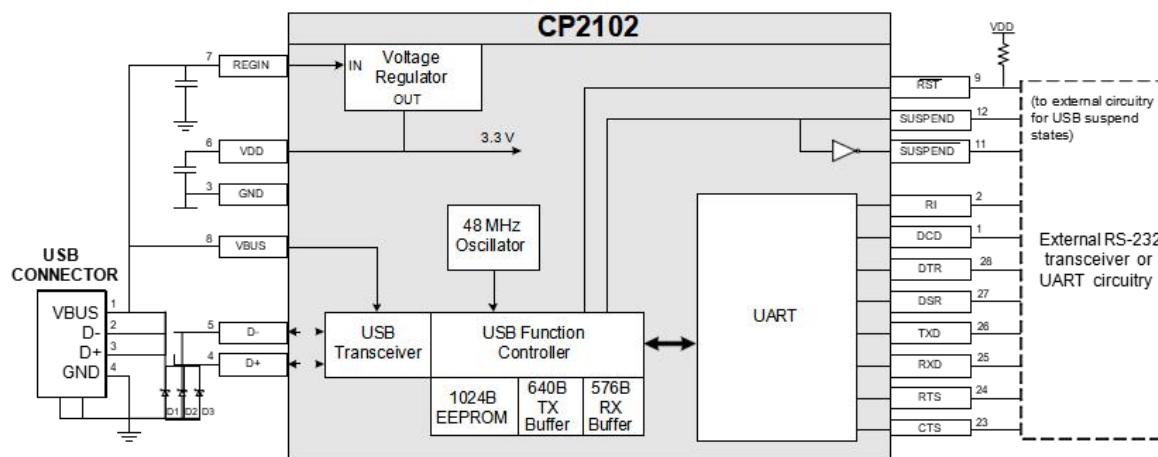


图 2-7 cp2102 电路框图

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 2.2。
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<sup>[2]</sup>。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 2.3 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, 230400,250000,256000,460800,576000,921600

表 2.2 CP2102 可配置参数

### 2.5.3 CP2102 应用

使用 CP2102 开发 USB 口转串口具有电路简单, 运行可靠, 成本低廉的优点, 通常在使用 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 2-7 所示, 由于我们并不需要将 TTL 电平转换为 RS-232 电平, 所以我们不需要额外的电平转换芯片。电路使用 CP2102 UART 总线上的 TXD/RXD 两个引脚,

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 2.3 CP2102 默认配置表

其余的引脚都悬空。此次我们使用的是购买的已经制作好的 CP2102 转 TTL 模块, 如图 2-8所示:

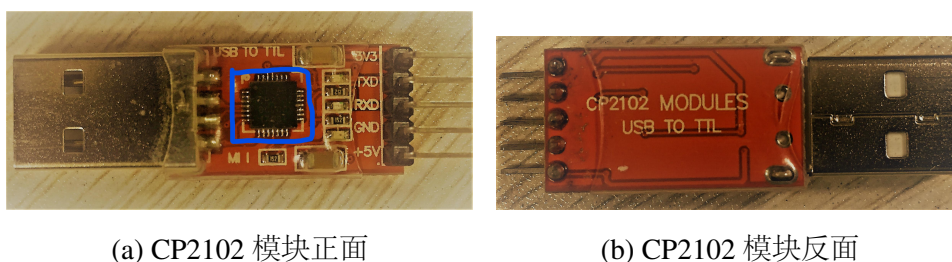


图 2-8 CP2102 模块正反面

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。

## 2.6 软件需求

整个调试通道的设计包括两个部分,一个是应用层的接口的设计,一个 USB 口转串口的驱动程序的设计。整个系统的结构如所示:

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

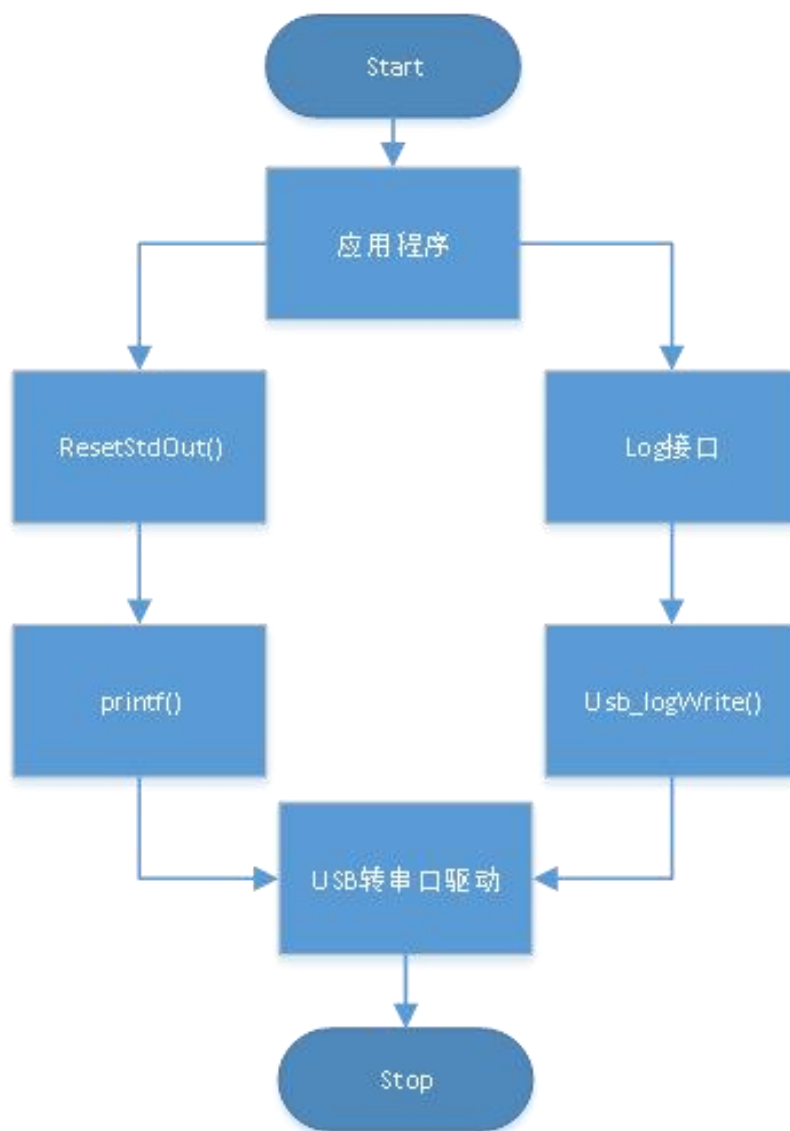


图 2-9 调试通道整体结构图

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。

主要的模块包块一下几个:

- **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)

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。

作为非标准类型的 USB 设备,CP2102 USB-TTL 模块连接到主机之后必须使用一个由开发者自己编写的设备驱动程序来驱动其正常工作,而 PC 端的应用程序则无需任何更改,仍将其当做一个正常的串口设备来使用即可,但是本质上所有针对该串口发起的通信都是通过 USB 总线来传输的。而对于设备一方,收发的都是 TTL 的电平的串行数据。利用现有的 CP2102 模块能够轻松地完成 USB-TTL 的转换,开发者无需考虑总线枚举、数据收发与转换等工作,这些都由芯片自动完成。

在 windows 和 Linux 下都有已经实现好了的基于 CP2102 的 USB 口转串口的驱动,但是在 VxWorks 下并没有这个驱动的实现,我们的调试通道的核心部分真是这个 USB 口转串口的驱动,因此我们需要在 VxWorks 中自己实现这个驱动程序。

### 三 驱动程序的设计和实现

在此我们需要编写一个 VxWorks 下的 USB 口转串口的驱动程序,在编写驱动程序之前我们有必要先了解一下 VxWorks 下驱动的软件结构,USB 口转串口的流程以及相关的工作原理。

#### 3.1 VxWorks 设备驱动概述

应用程序必须要通过驱动程序才能够与硬件设备进行通信,而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。将这种由操作系统的设计人员指定的设备驱动程序结构定义为驱动程序的外部结构,而由于驱动开发人员在开发设备驱动时采用的具体策略不同导致的不同的驱动程序结构称为驱动程序的内部结构。驱动程序的外部结构决定了操作系统的 I/O 体系结构,驱动程序的内部结构决定了不同的设备驱动方式。

驱动程序简单的说就是设置某个硬件完成其固有功能的程序。驱动程序直接与硬件设备交互,其大多数的工作就是操作硬件相关寄存器。首先寄存器也是一种 RAM,在系统下电后,寄存器中的内容都会丢失,系统上电复位过程中,硬件寄存器一般都复位到一个默认值,默认状态下硬件是不能正常工作的,如中断使能被屏蔽,工作使能位也被屏蔽,还有一些决定硬件工作情况的关键控制寄存器也需要被重新配置。而这些工作都有赖于设备驱动完成。驱动一般都作为操作系统内核组成的一部分,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,换句话说,驱动代码具有系统特权级,除了其自身资源,对应硬件设备资源,其还对操作系统资源具有完全的访问权。所以一个驱动程序如果存在 BUG,将直接会导致整个操作系统的崩溃。故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。应用层程序员往往对属于内核编程的外设驱动心存敬畏,认为驱动编程是一项非常复杂的工作。实际上,底层驱动编程往往比应用层编程具有更大的灵活性,就如没有调试不出来的硬件,也没有调试不出来的底层驱动,但是应用层 BUG 有时就是无法调试出来。底层驱动的调试过程是同时对硬件和驱动进行验证的过程。底层驱动很多时候用来定位硬件设计错误或者硬件芯片本身可能的问题,故底层驱动程序员必须对所驱动的设备有一个比较充分的了解,以及对与硬件交互的其他硬件或外界环境也需要有一个比较清楚的理解。

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。驱动程序在操作系统中扮演着一个非常特殊的角色,其类似一个黑盒子,让所有的硬件对操作系统的一套内部规范接口进行响应,驱动程序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。

设备驱动时直接控制设备操作的那部分程序,也是设备上层的一个软件接口。设备驱动程序的功能完成软件层对硬件的访问,实际上从软件工程的角度来说就是介于软件和硬件之间实现软件层标准接口的程序。软件层访问硬件必须要通过调用驱动程序。所以驱动程序不能自动执行,只能被系统或者程序调用。

应用程序必须要通过驱动程序才能够与硬件设备进行通信,而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。将这种由操作系统的设计人员指定的设备驱动程序结构定义为驱动程序的外部结构,而由于驱动开发人员在开发设备驱动时采用的具体策略不同导致的不同的驱动程序结构称为驱动程序的内部结构。驱动程序的外部结构决定了操作系统的 I/O 体系结构,驱动程序的内部结构决定了不同的设备驱动方式。

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

### 3.1.1 设备驱动的功能以及分类

驱动程序是位于用户和硬件之间一个软件层,驱动程序员有完全的决定权决定一个硬件设备以何种形式呈现给用户,不同的驱动程序可以使得同一个硬件设备以不同的方式对用户可见。我们可以将一个实际块设备以字符设备对用户可见,将一个实际 Flash 设备以硬盘设备对用户可见,以何种形式表现一个实际设备完全由底层驱动控制。驱动程序员可以提供一系列方式让用户对设备进行控制,甚至可以让用户直接操作硬件设备的每一个寄存器,由用户在寄存器层次对硬件设备进行操作;或者只提供一个读或写操作,屏蔽其他所有操作等等。

故从宏观角度而言,驱动程序实现的功能即提供一种底层服务机制供用户进行选

择。从微观角度而言,驱动程序需要对下配置硬件寄存器,完成对设备数据的读写,对设备本身的控制,对上使得设备能够响应用户的服务请求,这些服务请求如下:打开设备;读写设备;控制设备;关闭设备。

根据设备的工作方式和数据的存储或者来源不同,可以将设备分为三大类:

- **字符设备类型**

字符设备即以字节流的方式被访问,就如同一个文件,但不同于文件之处是字符设备一般不可以移动文件偏移指针,而只能顺序的访问数据。终端设备以及串口设备都属于字符设备类型。字符设备驱动至少需要实现 `open`, `close`, `read` 和 `write` 四个系统调用底层实现函数。

- **块设备类型**

块设备一般通过文件系统访问。而块设备的最多使用方式也是文件方式。块设备一般不能对单个字节进行访问,而是一个块的方式(如硬盘以一个扇区(512B)为单元进行访问)进行。块设备允许同一数据的反复读取和写入。最典型的块设备就是硬盘,Flash 设备也是一种块设备。

- **网络设备类型**

用于与网络上其他主机进行通信。其数据读取方式有些类似于字符设备,不可以对同一数据进行反复读写,只能顺序读写数据。且该类型设备区别于字符设备和块设备的一个很大的不同是,其不提供文件节点,任务要访问一个网络设备必须使用另一套网络套接字接口函数进行,与文件系统则完全不相关。网络设备底层数据传输上以块的方式进行,但是又不同于块设备中数据块的概念,网络设备中块的大小可以改变,但是有一个区间范围。

以上只是设备类型的划分方式之一,事实上,按以上的划分标准,某些设备接口在某些情况下可以表现为任意以上三种形式之一,如 **USB** 接口,可以是一个字符设备,如 **USB** 串口;也可以是一个块设备,如 **USB** 内存卡;也可以是一个网络设备,如 **USB** 网络接口。

### 3.1.2 VxWorks 设备驱动层次结构

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 3-1 所示。

I/O 子系统在整个驱动层次中起着十分重要的作用,其对下管理着各种类型的设备驱动。换句话说,各种类型(包括网络设备)的设备都必须向 I/O 子系统注册方可被内核访问。所以在 I/O 子系统这一层次,内核维护着三个十分关键的数组用以对



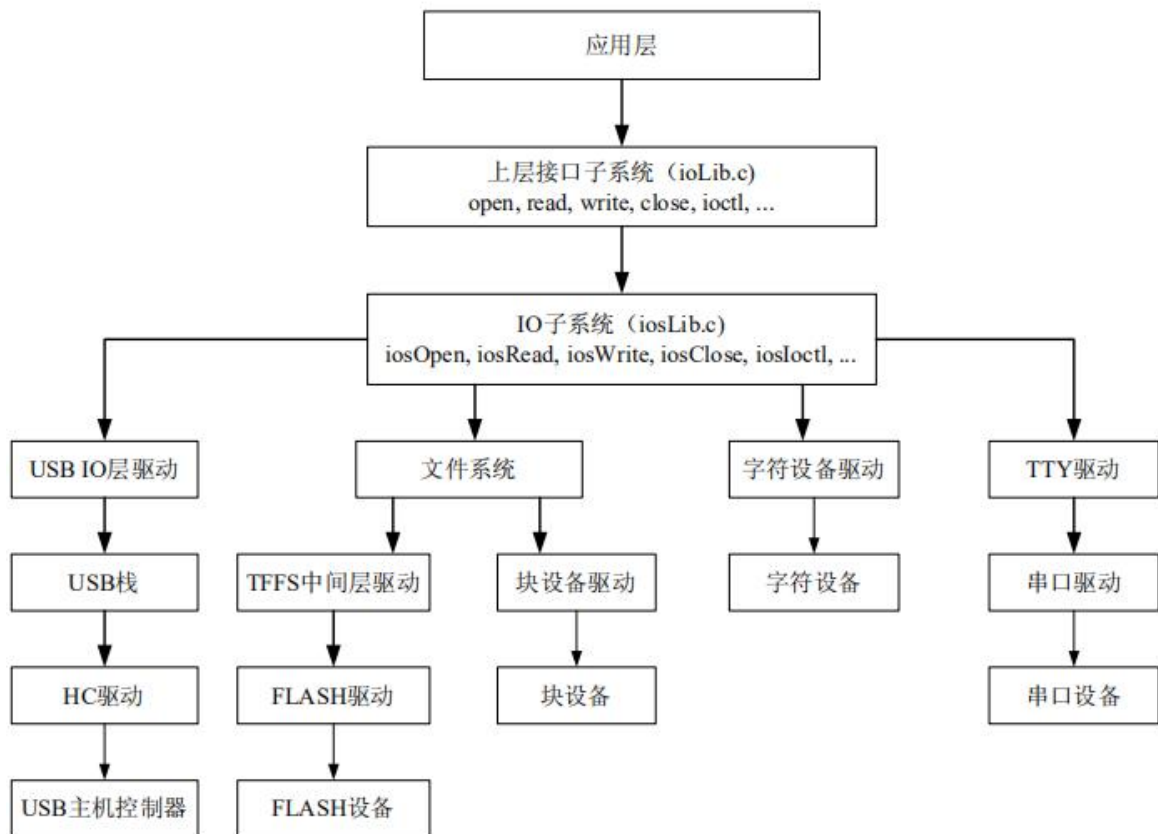


图 3-1 VxWorks 驱动内核层次结构

设备所属驱动、设备本身以及当前系统文件句柄进行管理。

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

## 3.2 内核驱动的相关结构

### 3.2.1 系统设备表

在 VxWorks 中对于每一个设备都会使用 DEV\_HDR 的结构来表示这个设备，DEV\_HDR 的定义如下：

```

1  /*h/iosLib.h*/
2  typedef struct /* DEV_HDR - device header for all device structures */
3  {
4      DL_NODE node; /* device linked list node */
5      short drvNum; /* driver number for this device */
6      char * name; /* device name */
7  }DEV_HDR;
    
```

在该结构中给出了链接指针 node (用以将该结构串入队列中)，驱动索引号



drvNum, 设备节点名 name。内核提供这个结构较为简单, 只存储了一些设备关键部分。底层驱动对其驱动的设备都有一个自定义数据结构表示, 其中包含了被驱动设备寄存器基地址, 中断号, 可能的数据缓冲区, 保存内核回调函数的指针, 以及一些标志位。最为关键的一点是 DEV\_HDR 内核结构必须是这个自定义数据结构的第一个成员变量, 因为这个用户自定义结构最后需要添加到系统设备队列中, 故必须能够在用户自定义结构与 DEV\_HDR 结构之间进行转换, 而将 DEV\_HDR 结构设置为用户自定义结构的第一个成员变量就可以达到这个目的。

为了能够让用户对设备进行操作, 驱动程序必须将设备注册到 IO 子系统中, 这个过程也被称为创建设备节点。IO 子系统提供了一个简单的被驱动程序调用的设备注册函数 iosDevAdd(), 该函数原型如下:

```
1 STATUS iosDevAdd
2 (
3     DEV_HDR *pDevHdr, /* pointer to device's structure */
4     char *name, /* name of device */
5     int drvnum /* no. of servicing driver, returned by iosDrvInstall() */
6 );
```

iosDevAdd 函数将一个设备添加到由 IO 子系统维护的系统设备列表中, 该列表是一个队列, 队列中成员通过指针链接在一起, 这是由 DEV\_HDR 结构中 node 成员变量完成的。系统设备列表由 iosDvList 内核变量指向, 系统设备表在系统中的连接方式如下图 3-2 所示:

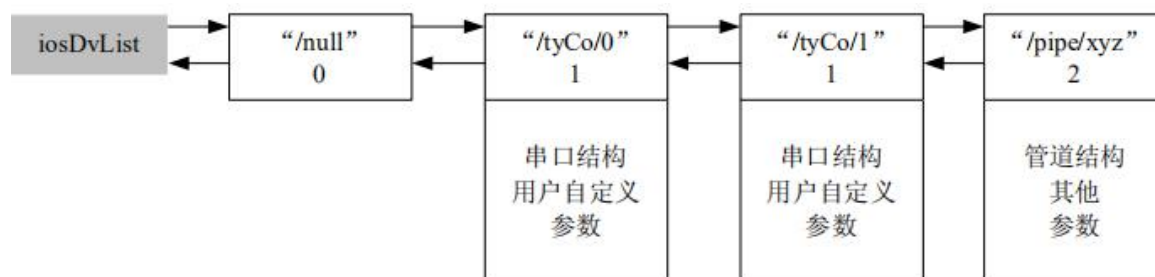


图 3-2 VxWorks 系统设备示意图

用户可以在命令行下使用 iosDevShow 或者是 devs 来显示系统设备的所有设备, 在本次使用的环境中, 存在的设备如图 3-3 所示。

### 3.2.2 系统驱动表

IO 子系统维护的系统驱动表包含有当前注册到 IO 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。

```

-> 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 3-3 当前系统上的所有设备

系统驱动表底层的实现是一个数组, 数组元素数目在 Vxworks 内核初始化过程中初始化 IO 子系统时指定, iosInit() 函数用以初始化 IO 子系统, 我们在编写驱动程序的时候不需要再调用此函数, 因为内核已经帮我们初始化好了 IO 子系统。iosInit 函数调用原型如下:

```

1 STATUS iosInit
2 (
3     int max_drivers, /* maximum number of drivers allowed */
4     int max_files, /* max number of files allowed open at once */
5     char *nullDevName /* name of the null device (bit bucket) */
6 );
    
```

在系统的驱动表中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:

```

1 typedef struct /* DRV_ENTRY - entries in driver jump table */
2 {
3     FUNCPTR de_create;
4     FUNCPTR de_delete;
5     FUNCPTR de_open;
6     FUNCPTR de_close;
7     FUNCPTR de_read;
8     FUNCPTR de_write;
9     FUNCPTR de_ioctl;
10    BOOL de_inuse;
11 } 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。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() 来提供。iosDrvInstall() 是 IO 子系统提供的驱动程序注册函数, 其原型如下:

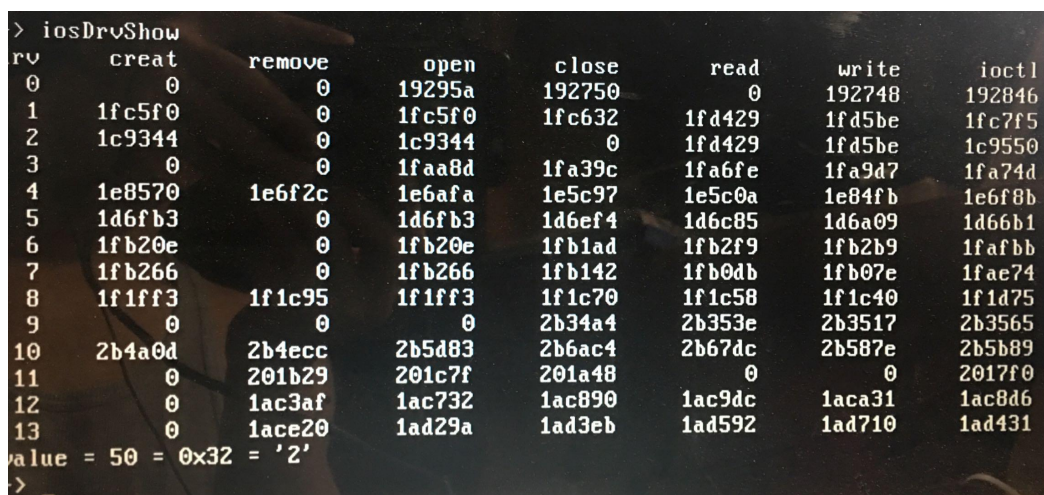
```

1  int iosDrvInstall
2  (
3      FUNCPTR pCreate, /* pointer to driver create function */
4      FUNCPTR pDelete, /* pointer to driver delete function */
5      FUNCPTR pOpen, /* pointer to driver open function */
6      FUNCPTR pClose, /* pointer to driver close function */
7      FUNCPTR pRead, /* pointer to driver read function */
8      FUNCPTR pWrite, /* pointer to driver write function */
9      FUNCPTR pIoctl /* pointer to driver ioctl function */
10 );

```

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

用户可在命令行下输入 iosDrvShow, 显示系统驱动表中当前存储的所有驱动。如图 3-4所示为当前系统中的所有驱动。



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	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 3-4 当前系统上的驱动表

### 3.2.3 系统文件描述符表

系统描述符表存储着当前系统范围内打开的所有文件的描述符。文件描述符表底层实现上也是一个数组,正如设备驱动表表项索引用作驱动号,文件描述符表表项索引被用作文件描述符 ID,即 `open` 函数返回值。对于文件描述符有一点需要注意:标准输入,标准输出,标准错误输出虽然使用 0,1,2 三个文件描述符,但是可能在系统文件描述符表中只占用一个表项,即都使用同一个表项。Vxworks 内核将 0,1,2 三个标准文件描述符与系统文件描述符表中内容分开进行管理。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。

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。

---

```

1  typedef struct /* FD_ENTRY - entries in file table */
2  {
3      DEV_HDR * pDevHdr; /* device header for this file */
4      int value; /* driver's id for this file */
5      char * name; /* actual file name */
6      int taskId; /* task to receive SIGIO when enabled */
7      BOOL inuse; /* active entry */
8      BOOL obsolete; /* underlying driver has been deleted */
9      void * auxValue; /* driver specific ptr, e.g. socket type */
10     void * reserved; /* reserved for driver use */
11 } 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。用户可以通过 `iosFdShow` 来显示系统文件描述符表中当前所有的有效表项,如图 3-5 所示是当前系统下的文件系统描述符表。

## 3.3 USB 转串口设备驱动程序的实现

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。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。



```

-> iosFdShow
fd name                                     drv
3 /pcConsole/0                             2 in out err
4 /aioPipe/0x2e400620                      3
5 (socket)                                9
6 (socket)                                9
7 (socket)                                9
8 (socket)                                9
9 (socket)                                9
10 /ata0a/Script.txt                       4
value = 100 = 0x64 = 'd'
->
    
```

图 3-5 当前系统上的文件描述符表

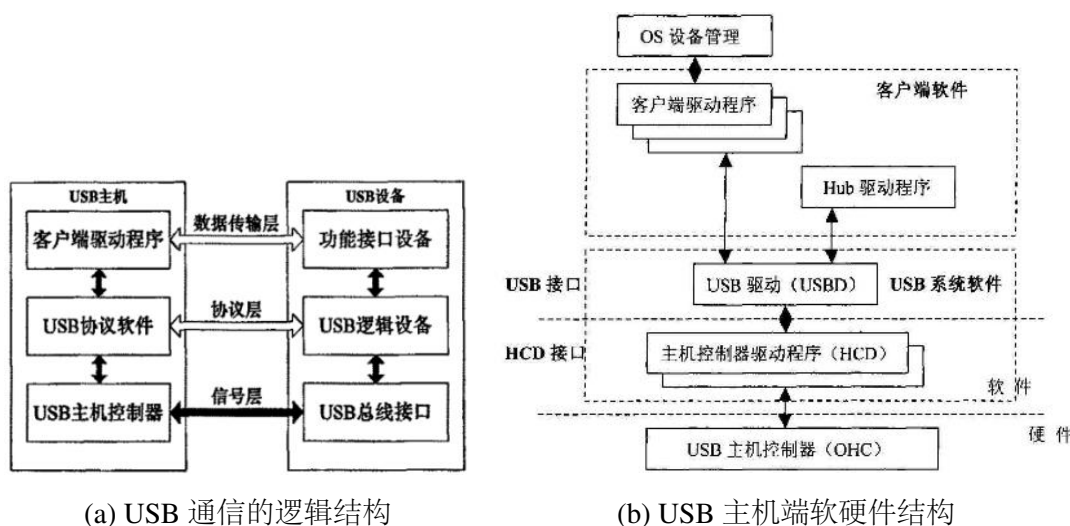


图 3-6 USB 通信结构

由于 USB 设备接口特殊性,其分类可以不按照3.1.1 小节中的分配进行,对 USB 接口的分类应该按照其具体的实现用途进行。在此处由于我们需要用其来实现一个 USB 口转串口设备,所以我们按照字符设备的方式来实现这个驱动。

在应用程序可以与一个设备通信之前,主机需要知道设备支持哪一些传输类型和终端,主机也必须要分配一个地址给设备。主机通过一个被称为列举的信息交换来完成这些工作。以下叙述 USB 列举的基本过程:

### 1. USB 设备与主机系统的交互

MCU 对 USB 芯片进行初始化:设置内部时钟,选择内部连接方式以及是否开通 DMA 传输等;设定设备的工作模式,并设备本设备的初始地址为 0(USB 规范指明当设备接入 PC 的时候都由初始为 0 的地址对主机进行响应,之后再由主机分配一个地址给 USB 设备,设备接收到分配地址命令后,再更改自己的地址,并一直通过这个地址完成后续的通信)。完成初始化工作之后,MCU 将使能 USB 接口,主机系统将因此检测到一个新的 USB 接入而很快与设备进行握手,获取

设备的基本信息,并完成一些列的对设备的配置,其过程为:

- 1.1 USB 上电使能后,主机会向 USB 设备发送 GET DEVICE DESCRIPTOR 的命令,之后主机会收到设备发出的设备描述符,随即为设备分配一个空闲地址,并向设备发送 SET ADDRESS 的命令,这时设备通过地址 0 发送一个长度为 0 的数据包予以应答,然后根据主机的要求更改自身的地址,而且这以后的数据交换都将会通过这个新的地址来进行。
- 1.2 完成地址设备之后,主机将会发送 GET CONFIGURATION DESCRIPTOR, USB 规定当主机发出该命令符的时候,设备必须要同时返回配置所包含的所有接口和接口所包含的所有端点的描述符。
- 1.3 主机获取到 USB 设备的描述符、配置描述符并进行了地址设置之后,设备与主机的握手初步完成,之后会将该设备加入到设备列表当中。

## 2. USB 设备与驱动程序的交互

USB 设备的驱动与传统意义上的硬件驱动不完全相同,他并不与硬件直接通信,而是以创建和发送 URB 请求块的形式把命令传递给操作系统所提供的 USB 总线驱动程序,由总线驱动程序来完成与硬件的直接交互。和主机类似,驱动会首先创建和发送请求得到该设备的 DEVICE DESCRIPTOR 的 URB,并将获取到的信息存储在专用的数据结构当中,接着驱动为了得到完整的设备配置,必须要通过总线驱动发送两次 GET CONFIGURATION DESCRIPTOR 命令得到设备。获取设备的配置信息之后,驱动在启动设备之前还要发送 SET CONFIGURATION、SET INTERFACE 命令。通过以上的几个步骤便可以完成驱动与 USB 设备过程。

当设备接入到 PC 的 USB 接口的时候,在固件和操作系统的支持下会对设备进行枚举操作,

由于我们的 USB 驱动程序不能够支持处理中断,所以只能用查询的方式来连续接收数据。可以使用两种方式实现设备的查询,一是使用计时器,另外就是使用系统线程。在我们的 USB 口转串口驱动程序中,我们使用两个线程实现数据的接收操作。在第一次设置 PID 过滤参数之后,开启一个系统线程,它会不停的从 IN 端口读取数据,然后写入到驱动程序中开辟的缓冲区当中;

### 3.3.1 VxWorks 上的 USB 协议栈

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,VxWorks 中的 USB 主机驱动程序堆栈体系结构如下图所示:

### 3.3.2 特定需求的单设备支持

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标准的设备的驱动的初始化流程存在差异。具体的需求为:驱动中支持的设备名是固定的,驱动程序中要有缓存一定数据的能力,即使设备没有正确的连接也要能够往这个驱动中写入数据,并且一旦设备连接之后就能够将驱动中缓存的数据发送出去。

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

驱动程序各模块的实现:

1. cp210xDevInit 模块

#### 3.3.2.1 USB 设备硬件的初始化

### 3.3.3 通用的多设备支持

## 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 Log 协议的设计

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。



## 4.2 标准输出重定向接口的设计

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。

### 4.2.1 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 4-1所示。部分关键代码如下:

```

1  int ResetStdOut(int usb_serial)
2  {
3      ...
4
5      _init_fd();
6      if(usb_serial == 1)
7      {
8          if(dup2(log_fd,STD_OUT) < 0)
9          {
10             printf("can not reset STDOUT to /hust_use_serial\n");
11             return -1;
12         }
13     }
14     else
15     {
16         if (dup2(STDOUT_FD,STD_OUT) < 0)
17         {
18             printf("can not reset STDOUT to /hust_use_serial\n");
19             return -1;
20         }
21     }
22     ...
23 }
24
```

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:

```

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。

### 4.2.2 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来实现重定向, task 模式下的标准输出重定向如图 4-2所示。部分关键代码如下:

```

1  int ResetStdOut(int usb_serial)
2  {

```

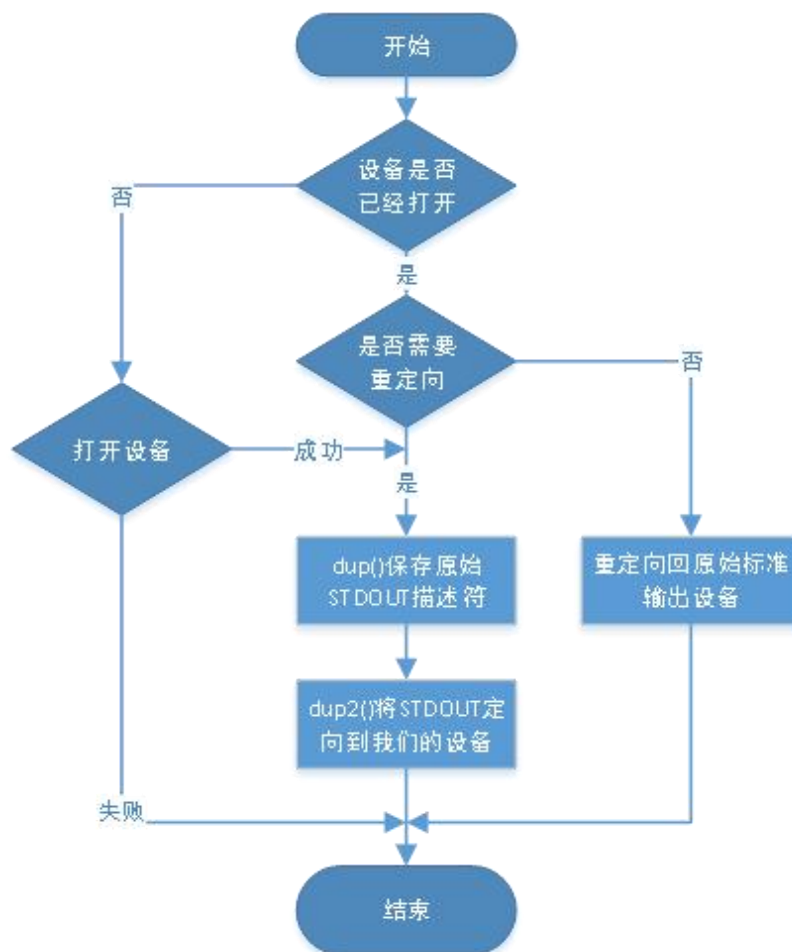


图 4-1 RTP 模式下标准输出重定向流程图

```

3   ...
4
5   _init_fd();
6   if(usb_serial == 1)
7   {
8       ioTaskStdSet(0,STD_OUT,log_fd);
9   }
10  else
11  {
12      ioTaskStdSet(0,STD_OUT,STDOUT_FD);
13  }
14
15  ...
16 }

```

`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:

```
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。第一个参数 `taskId` 表示需要进行

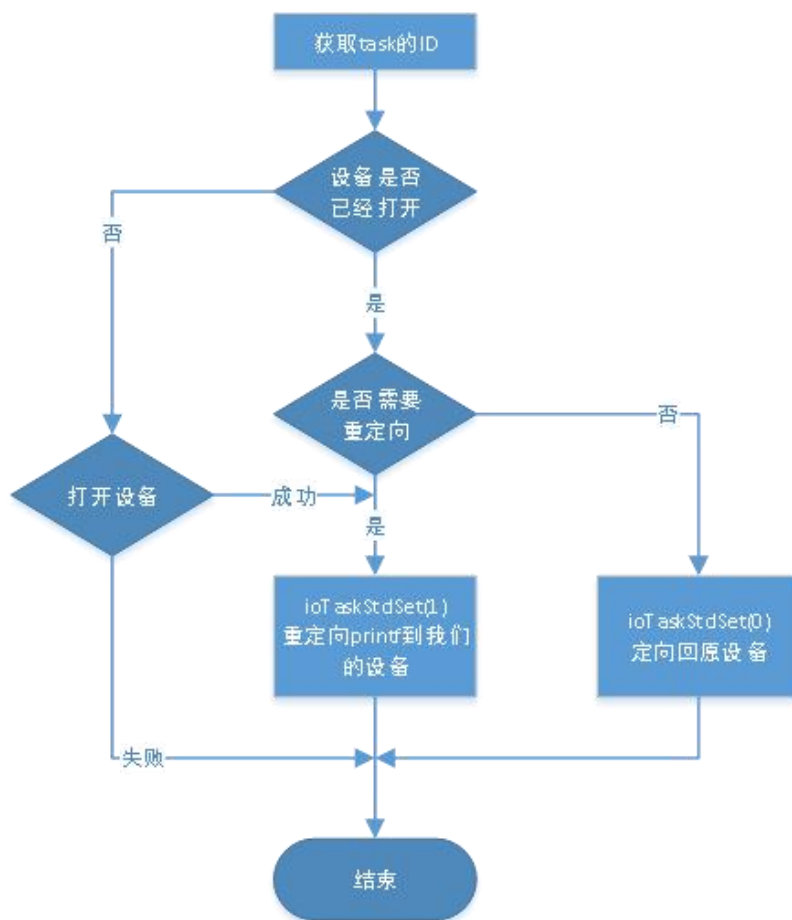


图 4-2 task 模式下标准输出重定向流程图

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。

### 4.3 Log 接口的设计

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, ##
   __VA_ARGS__)
10
11 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:

```

1 int usb_logWrite(char level, char *fileName, int lineNum, const char *
   format, ...)
2 {
3     ...
4
5     struct timespec tp;
6     struct tm timeBuffer;
7     time_t nowSec;
8     char datetime[64];
9     clock_gettime(CLOCK_REALTIME, &tp);
10    nowSec = tp.tv_sec;
11    localtime_r(&nowSec, &timeBuffer);
12    timeLen = strftime(datetime, 64, "%Y/%m/%d %H:%M:%S", &timeBuffer);
13    sprintf(datetime+timeLen, ".%3.3ld", tp.tv_nsec/1000000L);
14
15    _init_fd();
16    if(fileName != NULL)
17    {
18        char *rf = strrchr(fileName, '/');
19        if(rf != NULL) fileName = rf+1;
20    }
21
22    logWriteBuf[0]=0x03;
23    logWriteBuf[1]=0x03;
24    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);
25    n+=2;
26    va_list argList;
27    va_start(argList, format);
28    m = vsnprintf(logWriteBuf+n, LOG_BUF_SIZE-n, format, argList);
29    va_end(argList);
30
31    if(m <= 0)
32    {
33        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。

## 4.4 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 4-3 所示, 主界面如图 4-4 所示。

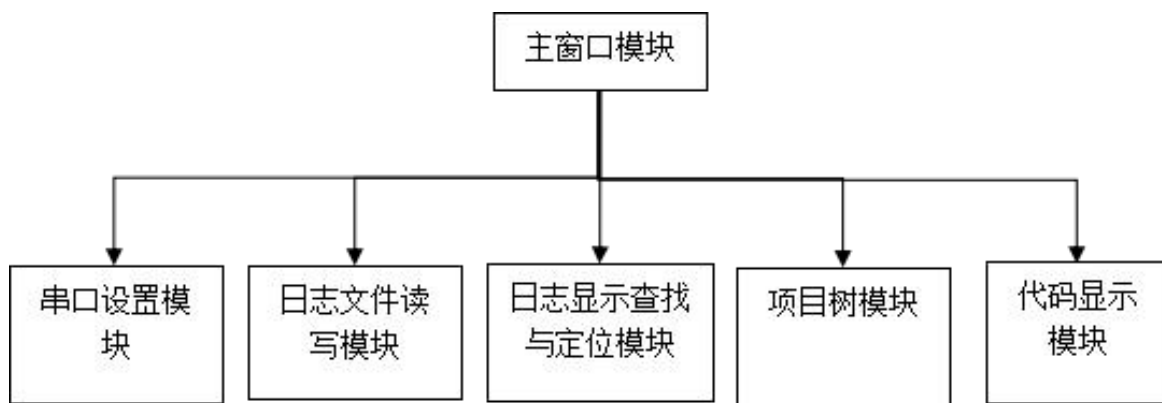


图 4-3 日志分析工具结构图

### 串口读取流程

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。首先, 解析一下协议格式, 每种日志级别的信息在日志显示框中会以不同的底色显示。为了提供系统的可用性, 对不按协议格式发送的日志信息, 本系统也做了合理的容错处理, 主要是针对没有调用 Log 接口而是直接使用标准输出重定向输出的调试信息。对串口读取的日志信息进行一下的分类处理:

每种情况都会寻找包头, 如果找到就会解析, 如果没找到则整体作为日志信息内容部分显示。另外, 在符合协议标准的数据包中, 内容部分是可以包含回车符的。

串口读取和解析的流程图如?? 所示

需要注意的是, 串口读取和解析程序位于串口可读信号对应的槽中, 也就是当串口可读时, 会自动调用。

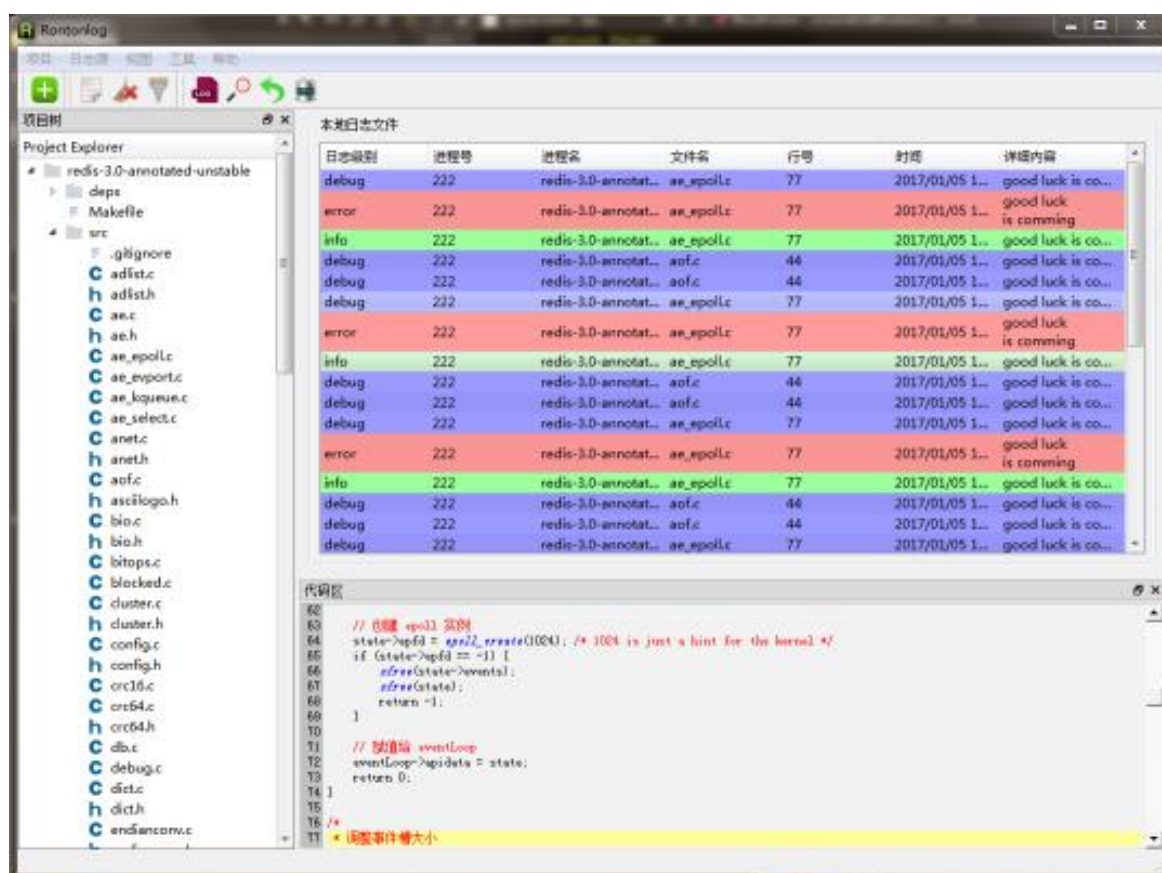


图 4-4 主机端日志分析工具界面

## 五

### 5.1 概述

#### 5.1.1 实时操作系统

## 致 谢

致谢正文。



## 附录 A 攻读学位期间发表的学术论文

[1] 论文 1

[2] 论文 2

## 附录 B 这是一个附录

附录正文。