

北京大学本科学位论文

基于 USB 协议的虚拟文件系统的设计与实现

姓名: 曲 宁

学号: 09708118

系别: 计算机科学技术系

专业: 计算机软件

导师: 程旭教授

摘要

“面向高端消费类电子产品的 32/16 位嵌入式微处理器及其示范原型系统”是由教育部主持国家高技术研究发展计划（863 计划）的重点课题。在该系统中，通过 USB 协议提供原型系统与 PC 间的通讯，并在此基础上提供操作系统中的文件系统所需要的基本支持。

本论文的工作是在“面向高端消费类电子产品的 32/16 位嵌入式微处理器及其示范原型系统”的基础上，针对原型系统上的由 USBN9603 控制芯片为核心的 USB 设备进行了接口设计和驱动程序的编写，同时利用上述的驱动程序完成对虚拟文件系统的支持。整个工作主要分为两个部分，在驱动程序方面涉及到了 USB 设备以及 Windows 下的驱动程序的编程模式；在虚拟文件系统方面主要是通讯协议栈以及文件系统命令集的定制和实现。在文中对这两个部分都做了详细的介绍和分析。

关键词：USB 驱动程序、WDM、USB

目录

摘要.....	1
目录.....	2
第一章 绪论.....	4
1.1 研究背景和论文任务的提出.....	4
1.2 程序设计模型概述.....	4
1.3 论文结构.....	5
第二章 背景知识.....	6
2.1 WDM设备驱动程序.....	6
2.1.1 WDM设备驱动程序概述.....	6
2.1.2 WDM设备驱动程序设计.....	7
2.1.3 WDM设备驱动程序环境.....	8
2.2 USB（通用串行总线）协议与USB设备.....	9
2.2.1 USB背景知识.....	9
2.2.2 USB体系结构概述.....	10
第三章 USB设备驱动程序的设计与实现.....	13
3.1 USB设备驱动程序概述.....	13
3.2 USB设备驱动程序设计需求.....	13
3.3 USB设备的访问与控制的实现.....	14
3.3.1 USB设备的逻辑结构.....	14
3.3.2 USBDI的调用.....	15
3.3.3 初始化过程.....	15
3.3.4 USB设备配置的定义.....	16
3.4 USB驱动程序的结构.....	18
3.4.1 程序的模块划分.....	18
3.4.2 主要的模块程序流程.....	18
3.4.3 主要的数据结构和函数说明.....	19
3.5 USB设备安装文件.....	21
3.6 USB驱动程序的调试.....	21
第四章 虚拟文件系统的设计与实现.....	23
4.1 虚拟文件系统设计概述.....	23
4.2 虚拟文件系统设计需求.....	23
4.3 虚拟文件系统的定义.....	24
4.3.1 基本原理.....	24
4.3.2 层次定义.....	24
4.3.3 传输原语定义.....	25
4.3.4 高层协议定义.....	25
4.4 虚拟文件系统的实现.....	26
4.4.1 程序模块划分.....	26

4.4.2 主要模块的程序流程.....	26
4.4.3 主要数据结构和函数说明.....	28
第五章 工作的总结与展望.....	31
5.1 毕业设计工作的总结.....	31
5.2 对今后工作的展望.....	31
致谢.....	32
附录：从DEBUGPRINT看USB设备驱动程序.....	33
参考文献.....	35

第一章 绪论

“面向高端消费类电子产品的 32/16 位嵌入式微处理器及其示范原型系统”研究成果——“JBCore32 32/16 位微处理器及系统软件和程序开发环境”是国家高技术研究发展计划（863 计划）重点课题。该项目主要包括：JBCORE32 微处理器，可重定目标的编译器 JBRC-2，汇编生成器 JBASM-2，结构级、组成级、信号级微处理器模拟器，操作系统原型，基于 JBCore32 的信息家电示范原型系统。

本篇论文的主要内容是操作系统原型中的基于 USB 协议的虚拟文件系统支持这一部分工作的说明。针对 USB 驱动程序设计以及虚拟文件系统的设计作出了详细的阐述。

1.1 研究背景和论文任务的提出

一个完整的操作系统的实现，需要软硬件上的协同支持。在示范原型系统中，硬件上，已经完成了支持 32 位/16 位指令和中断等功能的微处理器、控制输入输出的 I/O 控制器、支持虚拟地址变换的内存管理器；在软件上，已经完成了编译器，汇编器以及 Windows 环境下模拟器。不过由于系统中缺少 IDE 或者是类似的外存控制器，因此无法实现真正意义上的基于外存的文件系统。最后，选择的替代方案就是构建一个基于 USB 协议的虚拟文件系统。虚拟文件系统的实现是通过在示范原型系统中控制 USBN9603 芯片与普通 PC 机通讯，基于这个通讯功能，在 PC 端可以根据原型系统端发出的命令，进行相应的回应，从而完成操作系统中的文件系统调用。只要保证操作系统中文件系统的接口与标准一致，那么，如果以后实现真正的外存控制器，就可以很容易的进行移植，仅需要修改文件系统中函数接口的内部实现。

基于以上考虑，本次工作决定实现基于 USB 协议的虚拟文件系统。在原型系统端，包括文件系统接口的实现和 USB 控制芯片驱动程序；在 Windows 端，则包括 USB 驱动程序和虚拟文件系统的支持（简称 PC Daemon）。这两端的工作是由杨春和我协同完成的，我的重点是在 Windows 这一方面的工作。

1.2 程序设计模型概述

本次工作由两个部分组成：针对原型系统中以 USBN9603 芯片为核心的 USB 设备所编写的 USB 设备驱动程序以及构建在 USB 设备驱动程序基础上的虚拟文件

系统（PC Daemon）。这两个部分在系统中的调用关系如图 1-1：

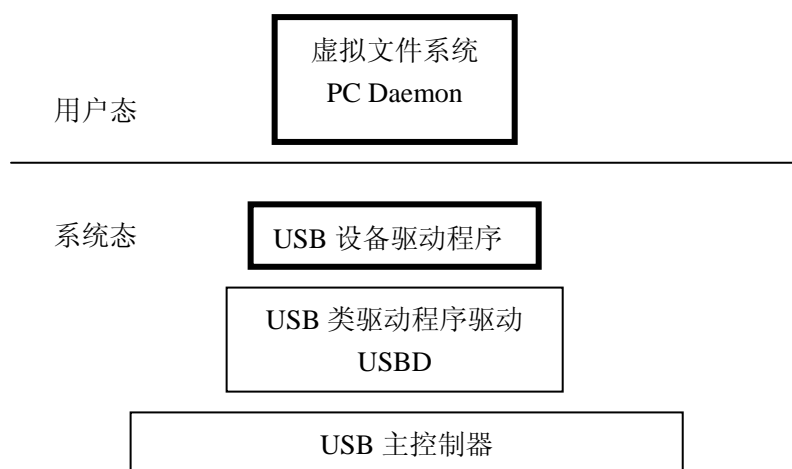


图 1-1 调用关系图

1.3 论文结构

本篇论文主要分为五个部分。第一章为绪论；第二章为背景知识，主要对论文所涉及到的主要的专业知识和技术做一个简要的介绍，包括 Windows 下的驱动程序和 USB 设备两个方面；在第三章中，主要介绍了 USB 驱动程序的设计与实现过程；第四章主要介绍了在完成 USB 驱动程序的基础上，虚拟文件系统支持的设计与实现；第五章主要是对本次工作的总结和对今后工作的展望。

第二章 背景知识

2.1 WDM 设备驱动程序

2.1.1 WDM 设备驱动程序概述

WDM 是 Windows 驱动程序模型 (Windows Driver Model) 的英文缩写, 这是在 Windows98 和 Windows2000 中常见的驱动模型中的一种。驱动程序是一个软件, 在装入后成为操作系统的一部分。它使一个或多个设备可用于用户态程序员, 每个设备代表一个物理的或逻辑的硬件。在 Windows 中, 驱动程序总是使设备看起来像是一个文件。可以打开设备的一个句柄, 然后应用程序可以在设备句柄最后关闭之前发出读写请求。

设备驱动程序由以下几个部分组成:

- 初始化自己
- 创建和删除设备
- 处理 Win32 打开和关闭文件句柄的请求
- 处理 Win32 输入/输出 (I/O) 请求
- 串行化对设备的访问
- 访问硬件
- 调用其他驱动程序
- 取消 I/O 请求
- 超时 I/O 请求
- 处理一个可热插拔的设备被加入或删除的情况
- 处理电源管理请求

严格说, 只有初始化模块一定不能少。在实际的工作中, 所有驱动程序都有分发例程处理用户 I/O 请求。WDM 设备驱动程序需要一个即插即用模块以及一个安装 INF 文件。驱动程序通常在它们的初始化例程中创建它们的设备, 并在一个卸载例程中删除设备。所有其他的模块都是可选的。

图 2-1 描述了设备驱动程序在系统体系结构中的位置以及用户的动作最后如何由设备驱动程序处理的。

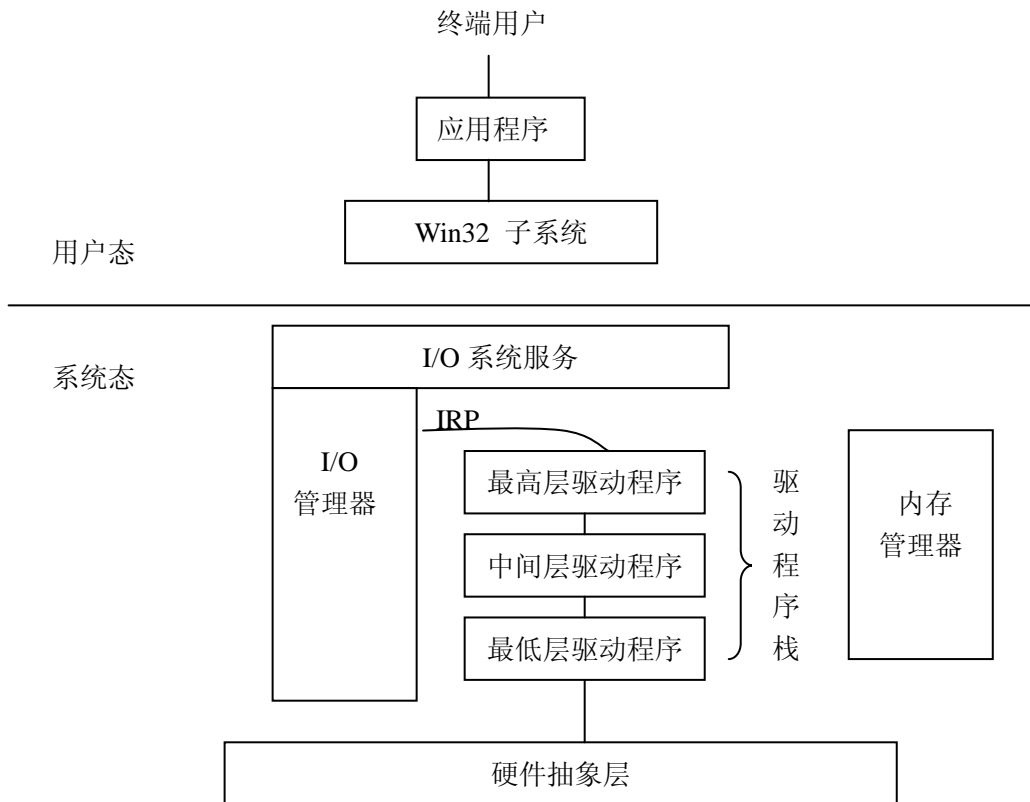


图 2-1 设备驱动程序系统框架

应用程序对设备 I/O 进行 Win32 调用，这个调用由 I/O 系统服务接收。I/O 管理器为这个请求构造一个合适的 I/O 请求包 (IRP)。

在最简单的情况下，I/O 管理器只是把 IRP 传递给一个设备驱动程序，这个驱动程序与硬件打交道，并完成 IRP 的处理。I/O 管理器把数据和结果返回给 Win32 和用户程序。

现在一个 IRP 由一个分层的设备驱动程序栈处理是很常见的。每个驱动程序把该请求划分为更简单的请求。更高层的驱动程序不必知道过多的细节；中间层次的驱动程序进一步处理请求；最后，由最低层的驱动程序与硬件实际打交道。这样也更容易保证高层驱动程序的专用性和底层驱动程序的通用性。

2.1.2 WDM 设备驱动程序设计

Windows 使用分层设计，把请求从终端用户传递到一个驱动程序栈，最终到达硬件。作为操作系统的可信任部分，驱动程序必须很好的适应内核。因此在编写驱动程序过程中有许多需要注意的地方。

- 内核调用：设备驱动程序不能访问任何 C、C++或 Win32 函数，而必须使用大量的内核例程。
- 内核对象：开发文档中描述内核时大量使用“对象”这个词。在最纯正的定义中，内核对象应仅使用内核函数调用来访问。在实际的使用中，每个内核对象通常有许多可直接访问的域，但是一些域应看成是私有的，驱动程序不能够访问。
- 中断级：驱动程序使用 3 个中断级。驱动程序的分发例程在 PASSIVE_LEVEL 级调用（即无中断）；许多驱动程序回调例程在 DISPATCH_LEVEL 级运行（即软件中断）；驱动程序硬件中断服务例程在 DIRQL 级运行（设备中断请求）。驱动程序运行的中断级非常重要，因为它确定可以采取的操作的类型。
- 内存使用：设备驱动程序在分配或访问内存时要特别小心。例如，驱动程序可以分配能够换出的内存，称为分页内存；另外可以分配永久驻留的内存，称为非分页内存。如果试图在 DISPATCH_LEVEL 或者更高的中断级访问分页内存，就回引起缺页故障，内核会崩溃。如果在 PASSIVE_LEVEL 中断级访问非驻留分页内存，内核会阻塞线程，直到内存管理器把内存装回到内存中。
- IRP：I/O 请求包（IRP）是驱动程序操作的中心。IRP 是一个内核对象，即预先定义的数据结构，带有一组对它进行操作的 I/O 管理器例程。I/O 管理器接收一个 I/O 请求，然后把它传递到合适的驱动程序栈中的最高驱动程序之前，分配并初始化一个 IRP。

2.1.3 WDM 设备驱动程序环境

- DDK：DDK 是驱动程序开发工具包的简称。在不同的 Windows 平台，要安装相应的 DDK。例如在 Windows 2000 中安装 Windows 2000 DDK。
- 调试：比较简单的获得有关的报警事件和错误事件信息，可以在 Windows 2000 中使用事件查看器显示驱动程序所写入的系统日志。而真正的内核态调试程序，常见的有几种：
 - ◆ Microsoft WinDbg，这种调试必须在两台 NT 或 Windows 2000 计算机之间使用；
 - ◆ Numega Compuware 的 SoftICE 调试程序，它可以与测试程序在同一台计算机上运行；
 - ◆ PHD Computer Consultants Ltd. 公司的 DebugPrint 软件，本次工作主要使用的就是这一调试工具。

2.2 USB（通用串行总线）协议与 USB 设备

2.2.1 USB 背景知识

USB 是通用串行总线（Universal Serial Bus）的缩写，支持在主机和各式各样的即插即用的外设之间进行数据传输。由主机预定的标准的协议使各种设备分享 USB 带宽，当其它设备和主机在运行时，总线允许添加、设置、使用以及拆除外设。

USB 的工业标准是对 PC 机现有的体系结构的扩充。USB 的设计主要遵循以下几个准则：

- 易于扩充多个外围设备；
- 价格低廉，且支持 12M 比特率的数据传输；
- 对声音音频和压缩视频等实时数据的充分支持；
- 协议灵活，综合了同步和异步数据传输；
- 兼容了不同设备的技术；
- 综合了不同 PC 机的结构和体系特点；
- 提供一个标准接口，广泛接纳各种设备；
- 赋予 PC 机新的功能，使之可以接纳许多新设备。

表 2-1 按照数据传输率（USB 可以达到）进行了分类。可以看到，12M 比特率可以包括中速和低速的情况。总的来说，中速的传输是同步的，低速的数据来自交互的设备，USB 设计的初衷是针对桌面电脑而不是应用于可移动的环境下的。软件体系通过对各种主机控制器提供支持以保证将来对 USB 的扩充。

性能	应用	特性
低速 • 交互设备 • 10-20kb/s	键盘、鼠标、游戏棒	低价格、热插拔、易用性
中速 • 电话、音频、压缩视频 • 500kb/s-10Mb/s	ISBN、PBX、POTS	低价格、易用性、动态插拔、限定带宽和延迟
高速 • 音频、磁盘 • 25-500Mb/s	音频、磁盘	高带宽、限定延迟、易用性

表 2-1

USB 的规范能针对不同的性能价格比要求提供不同的选择，以满足不同的系统和部件及相应不同的功能，其主要特色可归结为以下几点：

- 终端用户的易用性：
 - ◆ 为接缆和接头提供了单一模型；
 - ◆ 电气特性与用户无关；
 - ◆ 自我检测外设，自动地进行设备驱动、设置；
 - ◆ 动态连接，动态重置外设。
- 广泛的应用性：
 - ◆ 适应不同设备，传输速率从几千比特率到几十兆比特率；
 - ◆ 在同一总线上支持同步、异步两种传输模式；
 - ◆ 支持对多个设备的同时操作；
 - ◆ 可同时操作 127 个物理设备；
 - ◆ 在主机和设备之间可以传输多个数据流和信息流；
 - ◆ 支持多功能的设备；
 - ◆ 利用低层协议，提高了总线利用率。
- 灵活性：
 - ◆ 可以传输一系列大小不同的数据包，允许对设备缓冲器大小的选择；
 - ◆ 通过指定数据缓冲区大小和执行时间，支持各种数据传输率；
 - ◆ 通过协议对数据流进行缓冲处理。
- 健壮性：
 - ◆ 出错处理/差错恢复机制在协议中使用；
 - ◆ 对用户感觉而言，热插拔是完全实时的；
 - ◆ 可以对有缺陷设备进行认定。
- 与 PC 产业的一致性：
 - ◆ 协议的易实现性和完整性；
 - ◆ 与 PC 机的即插即用的体系结构的一致；
 - ◆ 对现存操作系统接口的良好衔接。
- 升级方式：
 - ◆ 体系结构的可升级性支持在一个系统中可以有多个 USB 主机控制器。

2.2.2 USB 体系结构概述

一个 USB 系统主要被定义为三个部分：

- USB 的互连；
- USB 的设备；

- USB 的主机。

USB 的互连是指 USB 设备与主机之间进行连接和通信的操作，主要包括以下几方面：

- 总线的拓扑结构：USB 设备与主机之间的各种连接方式；
- 内部层次关系：根据性能考虑，USB 的任务被分配到系统的每一个层次；
- 数据流模式：描述了数据在系统中通过 USB 从产生方到使用方的流动方式；
- USB 的调度：USB 提供了一个共享的连接。对可以使用的连接进行了调度以支持同步数据传输，并且避免了优先级判别的开销。

USB 连接了 USB 设备和 USB 主机，USB 的物理连接是有层次性的星型结构。每个网络集线器是在星型的中心，每条线段是点点连接。从主机到集线器或其功能部件，或从集线器到集线器或其功能部件，从图 2-2 中可看出 USB 的拓扑结构。

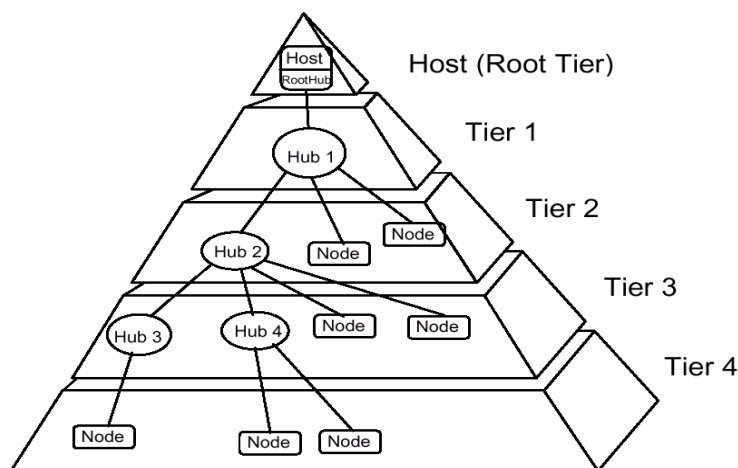


图 2-2 USB 总线的拓扑结构

在任何 USB 系统中，只有一个主机。USB 和主机系统的接口称作主机控制器，主机控制器可由硬件、固件和软件综合实现。根集线器是由主机系统整合的，用以提供更多的连接点。

USB 的主机通过主机控制器与 USB 设备进行交互。主机功能如下：

- 检测 USB 设备的安装和拆卸
- 管理在主机和 USB 设备之间的控制流；
- 管理在主机和 USB 设备之间的数据流；
- 收集状态和动作信息；
- 提供能量给连接的 USB 设备。

主机上 USB 的系统软件管理 USB 设备和主机上该设备软件之间的相互交互，USB 系统软件与设备软件间有三种相互作用方式：

- 设备编号和设置；
- 同步数据传输；
- 异步数据传输；
- 电源管理
- 设备和总线管理信息。

只要可能,USB 系统软件就会使用目前的主机软件接口来管理上述几种方式。

USB 设备主要分为两种设备类:集线器和功能部件。USB 设备需要提供自检和属性设置的信息,USB 设备必须在任何时刻执行与所定义的 USB 设备的状态相一致的操作。

集线器连接到总线上,可让不同性质的设备连接在 USB 上,连接点称作设备端口。每个集线器将一个连接点转化成许多的连接点。并且该体系结构支持多个集线器的级联连接。

一个集线器包括两部分:集线控制器(Controller)和集线放大器(Repeater)。集线放大器是一种在上游端口和下游端口之间的协议控制开关。而且硬件上支持复位、挂起、唤醒的信号。集线控制器提供了接口寄存器用于与主机之间的通信、集线器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。

功能部件是一种通过总线进行发送接收数据和控制信息的 USB 设备,通过一根电缆连接在集线器的某个端口上,功能设备一般是一种相互无关的外设。然而一个物理单元中可以有多个功能部件和一个内置集线器,并利用一根 USB 电缆,这通常被称为复合设备,即一个集线器连向主机,并有一个或多个不可拆卸的 USB 设备连在其上。

第三章 USB 设备驱动程序的设计与实现

3.1 USB 设备驱动程序概述

USB 设备驱动程序是支持即插即用功能的标准 WDM 驱动程序，通常是使用标准 Windows 系统 USB 类驱动程序通过访问 USBDI (Windows USB 驱动程序接口) 来实现的。USB.D.sys 就是 USB 类驱动程序，它使用 UHCD.sys 访问通用主机控制器接口设备，或是使用 OpenHCL.sys 访问开放主机控制器接口设备。USBHUB.sys 是根集线器和外部集线器的驱动程序，USB 集线器驱动程序检测一个新的 USB 设备何时插入。PNP 管理器使用厂商 ID 或设备类信息选择要运行的驱动程序。然后调用设备驱动程序的相应例程，添加设备，然后进行其他的相关操作。USB 设备驱动程序绝不会收到任何硬件资源（如端口或中断），因为 USB 类驱动程序处理了所有的底层 I/O。

根据定义，USB 设备驱动程序在它的下沿进行对 USB 类驱动程序的调用，然后实现所需要的上沿功能。在本次的 USB 设备驱动程序中，实现了设备的读写接口，可以动态选择不同类型的读写，同时也实现了对设备信息的查询和设置。

3.2 USB 设备驱动程序设计需求

本次所设计的 USB 设备驱动程序，是针对原型系统中的由 USBN9603 芯片构成的 USB 设备。设备驱动程序所要完成的基本功能：

- 对于底层，一方面要完成在 Windows 系统中的设备安装，以及对 PNP 功能函数的支持；另一方面需要完成与 USB 设备的所有交互，包括初始化，设置，最基本的读和写功能等。
- 对于上层，需要提供对于 USB 设备信息查询，状态的控制，以及简单的读和写。

同时，针对本次的特定应用，又有一些必须支持的特性：

- 对于普通文件系统的传输支持，需要保证传输的准确性，而对于传输的带宽要求不是特别严格。
- 对于播放音频文件时，要求文件传输的带宽非常高，而对于传输的准确性则可以放松要求。

3.3 USB 设备的访问与控制的实现

3.3.1 USB 设备的逻辑结构

从设备驱动程序的角度来看，USB 设备的逻辑结构如图 3-1：

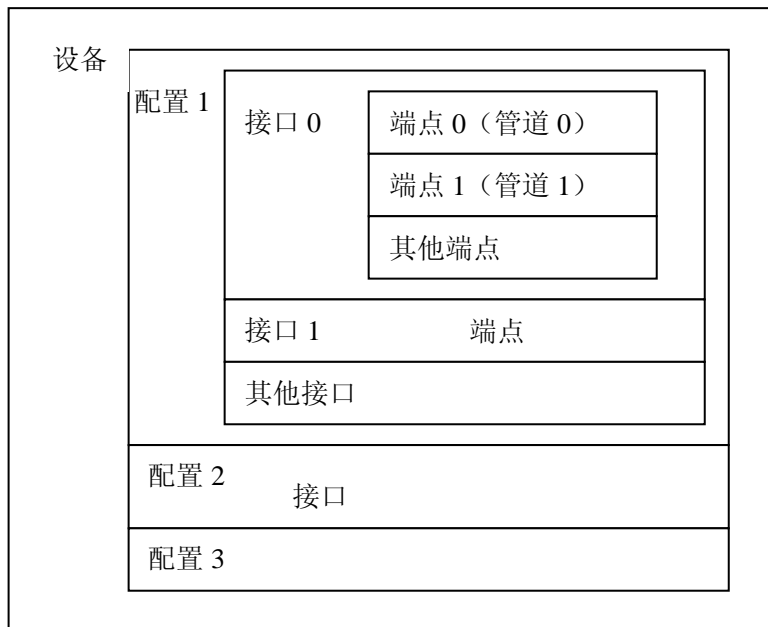


图 3-1 USB 逻辑结构图

每个设备内有一个或多个逻辑连接点，称为端点，每个端点可以定义为下面的传输类型之一：控制传输，中断传输，块传输和等时传输。所有设备都必须有一个端点 0，用于设备的配置和控制。

主机与设备端点之间的连接称为管道，主机 USB 系统软件和设备端点 0 之间的管道称为缺省管道。

一个设备对主机来说，表现为一组端点。一组相关的端点称为一个接口。每个设备可以有多组接口，每组相关的接口称为一个配置，而一个配置确定之后，也就定义好了设备的运行方式。在同一时刻只能有一个配置是有效的，称为活动配置，而在活动配置中可以有多个可用的接口，按照需要进行切换。

USB 设备驱动程序在初始化时通常需要调用底层的 USBDI 以获得设备的信息以及可以使用的配置，通过这些信息，根据实际应用的需要为设备选择一种合适的配置，这样，也就相当于对端点的使用进行了分配。经过配置后，USB 设备就可以进行传输了。

3.3.2 USBDI 的调用

USB 类驱动程序主要是通过使用 USB 类驱动程序接口 (USBDI) 的内部 I/O 控制 (IOCTL) 实现的。最重要的内部 IOCTL 是 IOCTL_INTERNAL_USB_SUBMIT_URB, 它发出 USB 请求块 (URB) 由 USB 类驱动程序处理。有 30 多个不同的 URB 功能代码。USB 驱动程序使用 URB 来完成大多数的工作。

URB 的数据结构是一个联合, 含有 16 个不同的结构。每个功能代码使用其中的一个结构来详细说明它的输入或输出函数。所有结构有一个公共的头结构, 再调用 USB 设备接口之前, 必须填写它的长度和功能域, 其处理结果在状态域中返回。有一些结构中还有一个链接域, 如果是非空的话, 那么在当前的 URB 完成之后, 将会处理所链接的 URB。

本次驱动程序中构造了 CallUSBDI 例程, 用于发出所有内部 IOCTL 给 USB 系统类驱动程序。它有缺省的参数, 使得要求 URB 被处理是非常容易的。

CallUSBDI 必须为内部 IOCTL 创建一个新的 IRP, 然后填写该 IRP, 并沿设备栈向下把这个 IRP 发送到 USB 系统驱动程序, 然后, 等待直到该 IRP 被处理。CallUSBDI 只能在 PASSIVE_LEVEL 这一级别中调用。

如果要调用底层驱动程序, 再次使用一个已存在的 IRP 是最简单的。只要将正确的功能代码和参数填入下一个 IRP 栈单元, 并调用设备栈中的下一个驱动程序就可以完成了。本次驱动程序也可以使用这种方式。但是, 分配新的 IOCTL IRP 也并不麻烦。在这个例程中使用 IoBuildDeviceIoControlRequest 调用, 很容易的构造和发出一个 IOCTL IRP。这样, 传递一个已经初始化的事件, 就可以通过等待该事件触发来等待 IRP 完成, 而不需要设置完成例程。

总之, CallUSBDI 完成了以下的工作:

- 初始化 IRP 完成事件
- 构造一个内部 IOCTL
- 在 IRP 的下一个栈单元中存储 URB 的指针等
- 调用下一个驱动程序
- 如果请求仍然挂起, 等待完成事件的触发。

3.3.3 初始化过程

为了初始化设备的连接, USB 驱动程序必须做几个工作。大多数的 USB 驱动程序要在处理“启动设备”Pnp IRP 时初始化它们的设备。

1. 检查设备是否启动, 如果必要, 复位并启用设备
2. 获取配置信息, 在某个配置中选择一个接口

3. 可能读其他描述符，如类描述符或厂商特定的描述符
4. 访问设备，发出相关的命令，读入设备的状态，以及初始化管道。

当一个程序打开设备的一个句柄时，一般需要对设备的端口进行复位。如果设备有严重的通信问题（如缺省管道的控制传输一直失败），也需要复位端口，但是，如果另一个管道暂停，不要复位该端口。如果管道暂停或 USBDI 检测超时，则该管道停止。这时，连接到当前 URB 的所有 URB 都被取消。停止的管道不能接受任何其他的传输，直到发出“管道复位”URB。但是，缺省管道决不能暂停。如果这里发生超时或暂停，则报告一个错误。暂停条件自动清除。从而提供了在这个管道上继续传输的机会，如果传输一直失败，则必须复位端口。

之后，就可以通过构造和发送 URB 由 USB 类驱动程序来处理的方式来构造各种例程。

接下来，要使用一个设备，驱动程序必须选择一个配置中的一个接口，这个工作相对比较的复杂。大致的步骤如下：

1. 得到配置，接口，端点描述符
2. 查找合适的接口描述符
3. 发出一个选择配置 URB
4. 保存要使用的所有管道的配置句柄

先通过例程获得需要的所有描述符，这个例程发出两次“获得描述符”请求。第一次调用只返回基本配置描述符，其中的一个域标志出要分配多少内存来取出所有相关的描述符。然后在所得到的描述符中查找所需要的接口，查找的条件是可以自定义的。在查找到合适的配置描述符以后，就可以通过这个描述符构造“选择配置”URB，在发出这个 URB 后，如果正常工作，那么返回配置句柄。可以使用这个句柄选择当前配置中的不同的接口设置。同时，还要保存所有的管道句柄，这样，以后就可以使用这些被配置过的管道进行相应的传输了。

3.3.4 USB 设备配置的定义

USB 设备的配置过程从逻辑结构上来看，是对配置描述符，接口描述符，端点描述符这三级描述符的定义与选择。

在配置描述符中定义了所支持的接口数目，供电方式等信息；在接口描述符中定义了这个接口所使用的端点数等信息；端点描述符定义了端点的地址，传输类型，接收数据包的长度限制等信息。

针对 USBN9603 芯片以及原型系统的设计，可以用最简单的方式定义配置描述符和接口描述符。本次的设备的供电方式为自供电，因此不需要通过 USB 总线所提供的电源。USBN9603 芯片的端点数共为 7 个，其中一个是缺省端点，

其他的分别是 3 个输入端点和 3 个输出端点。除了缺省端点是已经预定义好作为设备的控制传输用的，剩下的六个端点都是可以自定义使用的。根据这些信息，只需一个配置描述符和一个接口描述符就足以包括定义以上的所有信息。

然后，需要具体定义接口中的端点描述符，也就是定义端点的属性。在端点属性中，对传输过程有决定性作用的就是传输方式。那么，下面先对可以使用的四种传输方式，控制传输、中断传输、等时传输和块传输进行简要的介绍和分析：

- 控制传输：这种传输方式把命令传输给设备（包括 USB 设置和配置信息），含有沿任一方向的数据传输。常用来在设备与主机间传输少量的数据；
- 中断传输：这种传输方式的通过把设备的特定信息输入主机（如键盘按下等），而 PC 定期查询设备得到任何可用的中断数据，因此数据的传输方向只有设备到主机。其中主机定期查询的时间间隔为 1—225 毫秒，可以在端点的属性中进行设置；
- 等时传输：常用于传输与时间非常相关的数据（如来自电话的声音数据）。这也是四种方式中唯一没有 CRC 校验的方式，也就是说这种方式无法保证数据的绝对准确。同时，这种方式也是单向传输的。USB 把它的可用带宽分成帧，每帧最多传输 1023 字节数据；
- 块传输：这是一种异步数据传输方式，数据的传输是双向的，通常用来传输大量的数据。

通过以上的对比分析，可以比较容易的决定最终使用的传输类型：

在系统中的控制命令都可以使用控制传输方式。而控制传输是对传输利用率最低的方式，因此复用缺省管道就是最佳的选择，这样不会无端的占用系统中的有限的管道资源，同时足以满足系统的需求。

在文件系统的基本传输中首先要求数据的绝对正确，那么在保证校验的前提下，又能兼顾了带宽的要求无疑只有块传输方式。

而对于音频文件的播放，如果 USB 协议没有数据方向的限制的话，其实最适合的方式就是等时传输方式，不过很可惜由于 USB 设备本身是从设备，因此不提供这种从主机到 USB 设备的等时传输，因此只能用块传输方式来替代了。不过由最后的实践证明，块传输方式在系统中还是可以满足需要的。

每个端点的传输方向是唯一的，因此最后定义为所有的对设备的控制信息都通过缺省管道来完成，而把其余的 3 个输入管道和 3 个输出管道均定义为块传输方式。这样，就需要尽可能的利用块传输的特点，最大限度的利用带宽，提高有效数据的利用率。

3.4 USB 驱动程序的结构

3.4.1 程序的模块划分

USB 驱动程序按照功能大体可以分为以下几个模块：

- USB 设备初始化模块。支持 USB 设备的即插即用，读取 USB 设备描述符，读取并选择 USB 配置描述符，获得 USB 的每个端点的描述符，进行相应的设置。
- USB 设备控制模块。按照 USB 协议标准，向设备发送各种控制命令和请求。
- USB 设备接收模块。使用块传输管道进行较大量的数据接收，完成所需的传输。
- USB 设备发送模块。使用块传输管道进行较大量数据的发送。
- USB 设备监视模块。可以对 USB 设备的状态，传输情况等信息进行查询。

3.4.2 主要的模块程序流程

驱动程序最核心的的工作集中在初始化模块，接收模块和发送模块，这些模块包括了一个设备从配置到真正运行的主要过程。由于发送模块的流程与接收模块的流程很相似，因此在下面仅分析初始化和接收这两个过程。

初始化模块流程如图 3-2：

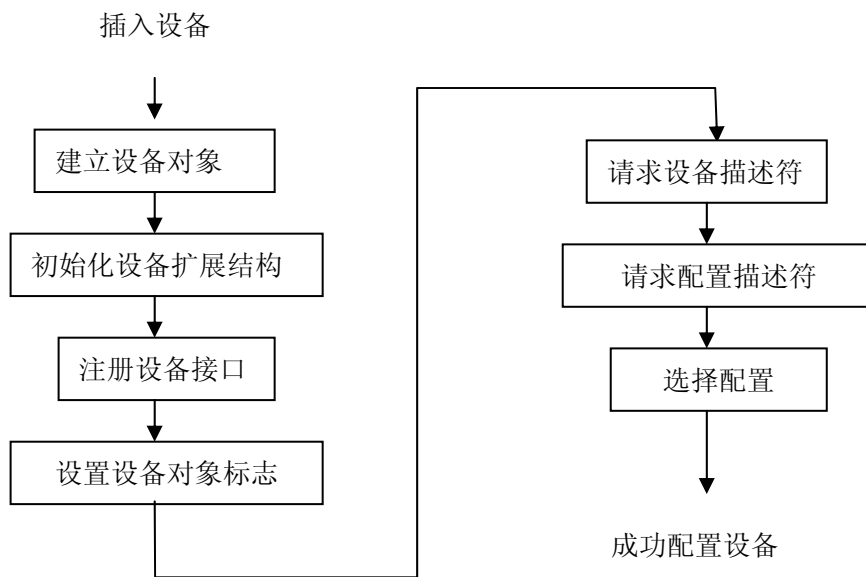


图 3-2 USB 设备初始化

接收模块流程如图 3-3:

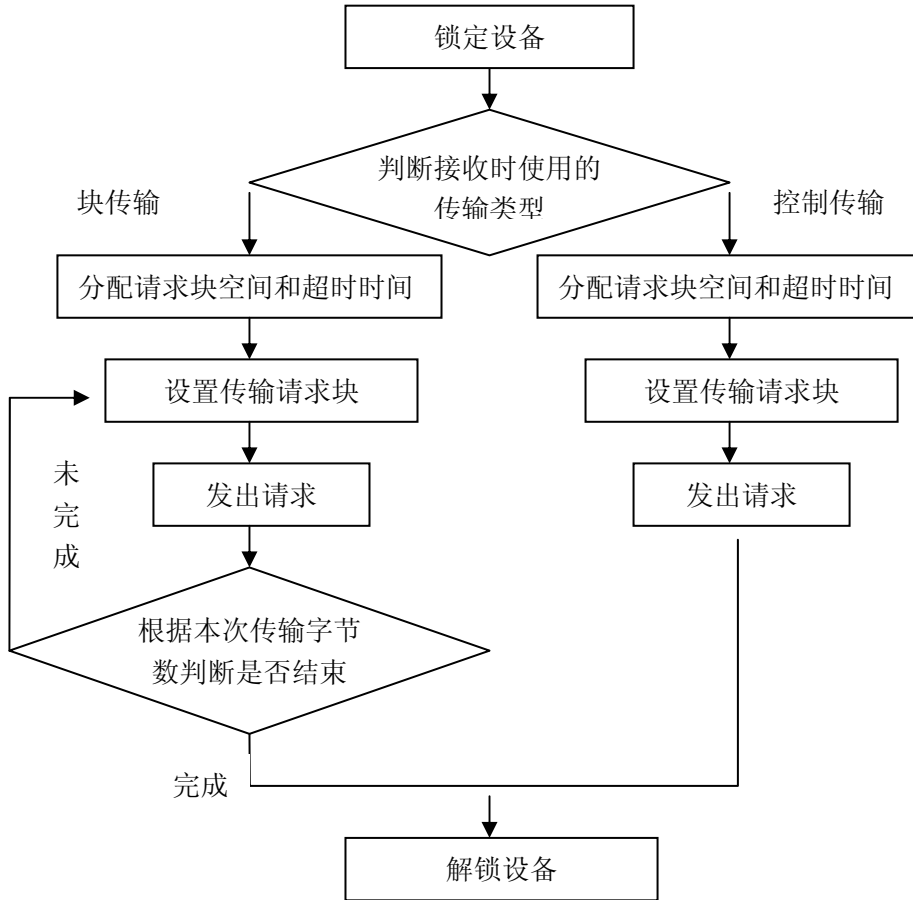


图 3-3 USB 设备接收过程

3.4.3 主要的数据结构和函数说明

程序的最重要的数据结构就是设备扩展结构,这个数据结构几乎在驱动程序的每个函数中都用到,所有配置好的设备信息也都可以其中找到。下面将简单的说明其中的主要内容:

```

typedef struct _USB51_DEVICE_EXTENSION //设备扩展结构
{
    PDEVICE_OBJECT fdo; //驱动程序对象

    PDEVICE_OBJECT NextStackDevice; //驱动程序栈中下面的驱动
    //程序对象

    UNICODE_STRING ifSymLinkName; //符号链接名称
}
    
```

```

bool GotResources;                //设备是否初始化成功

LONG OpenHandleCount;            //设备打开的数目

USBSD_CONFIGURATION_HANDLE UsbConfigurationHandle;
                                //选择的配置句柄

ULONG UsbTimeout;               //传输超时限制

PUSB_DEVICE_DESCRIPTOR pDeviceDesc;
                                //设备描述符

PUSB_CONFIGURATION_DESCRIPTOR pConfigDesc;
                                //配置描述符

_USBD_PIPE_INFORMATION hInPipe[PIPE_COUNT];
                                //输入管道

_USBD_PIPE_INFORMATION hOutPipe[PIPE_COUNT];
                                //输出管道
};

```

主要的函数包括对上层的接口（分发例程），和完成接口所需的具体功能的支持函数。

分发例程分别为：

- 打开设备
NTSTATUS Usb51Create (IN PDEVICE_OBJECT fdo, IN PIRP Irp)
- 关闭设备
NTSTATUS Usb51Close (IN PDEVICE_OBJECT fdo, IN PIRP Irp)
- 从设备读
NTSTATUS Usb51Read (IN PDEVICE_OBJECT fdo, IN PIRP Irp)
- 向设备写
NTSTATUS Usb51Write (IN PDEVICE_OBJECT fdo, IN PIRP Irp)
- 设备控制
NTSTATUS Usb51DeviceControl (IN PDEVICE_OBJECT fdo,
IN PIRP Irp)

支持函数主要包括：

- 重置设备状态
NTSTATUS UsbResetDevice (IN PUSB51_DEVICE_EXTENSION dx)
- 请求设备描述符
NTSTATUS UsbGetDeviceDescriptor (
IN PUSB51_DEVICE_EXTENSION dx,
OUT PUSB_DEVICE_DESCRIPTOR& deviceDescriptor,
OUT ULONG& Size)
- 请求配置描述符
NTSTATUS UsbGetConfigurationDescriptors (
IN PUSB51_DEVICE_EXTENSION dx, OUT

USB_CONFIGURATION_DESCRIPTOR& descriptors,
IN UCHAR ConfigIndex, OUT ULONG& DescriptorsSize)

- 选择配置
NTSTATUS UsbSelectConfiguration (
IN PUSB51_DEVICE_EXTENSION dx)
- 进行块传输
NTSTATUS UsbDoBulkTransfer (IN PUSB51_DEVICE_EXTENSION dx,
PVOID& UserBuffer, ULONG& UserBufferSize,
bool bDirection, bool bSoftReset)
- 进行控制传输
NTSTATUS UsbSendOutputReport (
IN PUSB51_DEVICE_EXTENSION dx,
IN UCHAR OutputData, unsigned long& iLen,
bool bDirection)

3.5 USB 设备安装文件

INF 文件包含安装一个 WDM 设备驱动程序需要的所有必需的信息, 包括要复制的文件列表, 要创建的注册表项等。

INF 文件是一个文本文件, 看起来象老的 INI 文件一样。由节组成, 每节中的一行定义了一项。接下来将对本次安装文件所用到的节作简单的介绍:

Version 节: 包括版本, 系统定义类名字, 匹配的 GUID 等;

Manufacturer 节: 制定厂商名和对应的 models 节名称;

Models 节: 指定产品的名称, install 节的名称, 对应的硬件 ID 等;

Install 节: 指定要复制的文件或是 filelist 节的名称等;

FileList 节: 要安装的文件列表。

当发现新的设备时 (在系统启动时, 在安装热插拔设备时, 或者从控制面板中安装新设备时), 系统就调用 Windows 的“添加设备向导”扫描所有可用的 INF 文件, 根据设备硬件 ID 和设备兼容 ID 选择需要装入的驱动程序。而这个硬件 ID 应该在相应驱动程序的 INF 文件中指定, 这样, Windows 就可以找到新设备的相应的驱动程序了。

3.6 USB 驱动程序的调试

驱动程序的常见的错误方式有崩溃, 内核转储, 驱动程序不启动, 挂起, 资源遗漏, 时间依赖性等。而这其中绝大部分如果不使用调试工具的话, 根本不会在系统中留下任何的记录, 所以即使机器不重启, 也看不到任何的结果。

因此, 为了在驱动程序执行过程中进行调试, 必须选择辅助的调试工具。在

本次程序中使用的是 DebugPrint 软件,它允许使用格式化打印语句跟踪驱动程序的执行。尽管它不是最好的调试解决方案,但它肯定为驱动程序开发人员提供了一个非常有用的工具,用以跟踪驱动程序运行的代码和变量中的数据。

要使用 DebugPrint,必须先安装 DebugPrint 驱动程序。然后使用 DebugPrint Monitor 程序与被测试的程序在同一台计算机上运行。启动后,DebugPrint Monitor 开始监视 DebugPrint 跟踪事件,如果出现任何事件,就读取并显示这些事件。

在驱动程序中使用 DebugPrint 必须首先把 DebugPrint.c 和 DebugPrint.h 两个标准的源文件复制到驱动程序项目中,然后在头文件中包括 DebugPrint.h。在驱动程序代码中运行对 DebugPrint 函数的调用。记住,只能在 DISPATCH_LEVEL 或更低的 DIRQL 级别调用这些函数。

DebugPrint 的调用应该在驱动程序执行中只引起很小的延迟,DebugPrint 调用的主要工作是在以低实时优先级在后台运行的系统线程中发生。

在附录中,可以通过 DebugPrint 的打印信息看到 USB 设备从插入到进行一次发送和接收的完整过程。

第四章 虚拟文件系统的设计与实现

4.1 虚拟文件系统设计概述

文件系统是操作系统的必不可少的部分,对于操作系统的整体性能起到至关重要的作用。在原型系统中的操作系统 TEOS 开发过程中,由于本身硬件没有提供对于外存的支持,因此为了简化整个系统的设计和开发难度,决定对 Windows 的文件系统进行必要的封装,基于底层 USB 对于块传输的支持,构建合适的通讯协议,从而完成系统调用的命令,参数以及最后的返回结果,为上层提供了一套比较完整的标准接口。

USB 驱动实际上是实现虚拟文件系统的底层工具,因此在 USB 驱动程序中没有提供过多的对于传输方面的支持。在虚拟文件系统中,将对 USB 驱动进行封装,同时建立起一套简单,完整的协议来完成对于文件系统调用的支持。

4.2 虚拟文件系统设计需求

本次完成的虚拟文件系统,是为了支持原型系统中的操作系统 TEOS 所设计的。同时这个虚拟文件系统构建于 USB 驱动程序之上,因此虚拟文件系统所要完成的基本功能包括两个部分:

- 面向底层,为了完成基本的传输,必须使用 USB 驱动程序,同时对于 USB 设备的读写等控制进一步封装,更加的适合于上层使用。简单的说,就是完全支持对于任意指定长度的缓冲区内容的传输。
- 面向上层,必须完成文件系统的基本原语支持,包括文件的 I/O 操作,目录处理函数。这些原语所需要的参数,数据等最终都将按照定义的协议转换为标准的字符串。虚拟文件系统支持的系统调用包括:
 - ◆ 文件操作: 打开文件, 读文件, 写文件, 移动文件指针;
新建文件, 删除文件, 打开文件, 关闭文件;
 - ◆ 目录操作: 新建目录, 删除目录;
查询当前目录, 改变当前目录;
查找目录第一个匹配文件, 查找目录下一个匹配文件,
查找结束

为了更好的支持对于操作系统等程序的调试,在程序的运行过程中加入了相应的事件记录的显示,这样便可以方便的查询整个操作过程中与文件系统的交互情况。

4.3 虚拟文件系统的定义

4.3.1 基本原理

整个原型系统中的操作系统的文件系统由原型系统端的文件系统调用和 PC 端的文件系统支持两部分组成，两端都是基于各自的 USB 驱动程序实现的。主要的运行过程如下：

1. 原型系统端的应用程序或是系统程序发出文件系统调用
2. 操作系统调用相应的文件系统函数
3. 文件系统函数使用由 USB 驱动程序实现的虚拟文件系统原语
4. 由 USB 驱动程序通过 IO 控制器控制 USB，向 PC 发出实际的文件系统请求
5. PC 端的 USB 驱动程序接收到原型系统端发过来的信息，并交给上层设备驱动程序
6. 设备驱动程序把得到的信息交给 PC Daemon
7. PC Daemon 具体分析得到的信息，调用 windows 文件系统命令完成相应的操作，并把结果通过上述的通路返回给原型系统端

这样，当返回后，原型系统端的应用程序或是系统程序就会得到所需要的结果。

4.3.2 层次定义

整个文件系统的结构实际上可以描述为三层的通讯协议栈的形式：

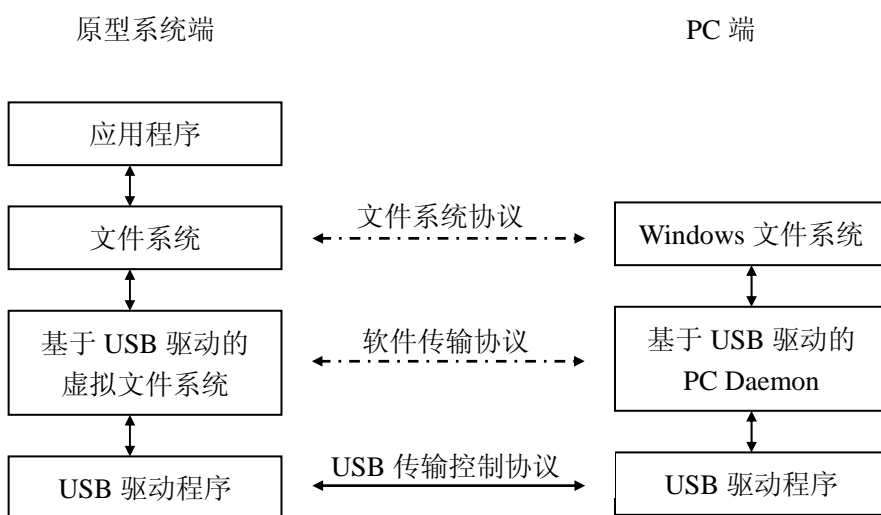


图 4-1 基于 USB 接口的文件系统结构

1. 文件系统协议：在原型系统端的文件系统函数使用用于与 windows 文件系统类似的参数；这样 PC 端调用对应函数，完成所需功能
2. 软件传输协议：对于文件系统调用的参数，使用自定义的软件协议，最后交给底层的格式是纯粹的字节流；在 PC 端，进行解码，分析命令格式，还原为上层文件系统可以使用的参数。
3. USB 传输控制协议：原型系统端控制 USB 控制器，完成与 PC 端 USB 主机控制器的交互，保证字节流以无错的方式在两端传输。

4.3.3 传输原语定义

虽然 USB 驱动程序已经提供了读写的传输功能，但是由于 Windows 底层缓冲区（4K）的限制，因此在 USB 驱动程序设计的时候，实际上每次可以对设备读写的缓冲区的大小不能超过这个限制。

这个缺陷为上层文件系统的开发增加了繁琐的考虑，因此在这里采用对字节流自动切分的方式，实现了对 USB 设备的传输功能进行简单的封装，从而真正提供了对于不限长度的字节流传输的支持。最终，为上层提供了两个简单的原语：向设备写入和从设备读出，参数为缓冲区指针和缓冲区长度。这样，高层协议的实现就不再受到驱动程序的限制了。

4.3.4 高层协议定义

由于底层的驱动传输是以字节流为基础的，因此在两端的文件系统对底层调用时，采用了把参数转化为字节流的方式。在两端传输实际数据前，先传送相应的命令，指定要传输的内容和长度，然后再发送参数的方式完成的。在函数返回值方面也采取了同样的软件通讯协议。

针对不同的类型具体的转换方式如下：

- 整数，作为 4 个字节的字符串直接拷贝到将要发送的字符串中，这样便可以在另一端很容易的恢复出来；
- 字符串，不作变化的拷贝到要发送的字符串后，在字符串的结尾加入一个分隔符，表示结束。
- 结构，这个复杂类型实际上只是以上两个简单类型的组合，因此只需要用上面的方式依次对结构中的每个域进行转换。

这样，有了以上的协议以后，对于系统调用的封装就可以很容易的实现了。系统调用的过程主要包括两个部分。

- 系统调用命令发送：第一个字节标志此次系统调用的类型，然后依次是系统调用的每个参数。

- 系统调用结果返回：结果分两次发送，先发送返回值，然后如果需要的话再发送数据。发送返回值也是第一个字节标志类型为返回值，后面才是要返回的具体值；发送的数据由于是紧跟着返回值，因此不需要标志类型，格式就是纯数据。

4.4 虚拟文件系统的实现

4.4.1 程序模块划分

整个程序按照功能可以划分为两个部分：

- USB 设备驱动程序封装模块。提供了对设备的打开，关闭功能的简单封装。同时屏蔽了所有的底层的传输细节，为上层的开发提供了简单的接收和发送原语。
- 虚拟文件系统模块。提供了对于 Windows 文件系统的系统调用的封装。利用上述模块的接收和发送原语，实现了系统调用的发送和返回的支持。为操作系统提供了基本的文件，目录操作的系统调用。同时实现了文件系统运行过程中的事件记录，为调试提供了进一步的支持。

4.4.2 主要模块的程序流程

在 USB 设备驱动程序封装模块中，主要是两个原语发送和接收，它们的流程比较的类似，因此下面以接收为例分析模块的流程，如图 4-2：

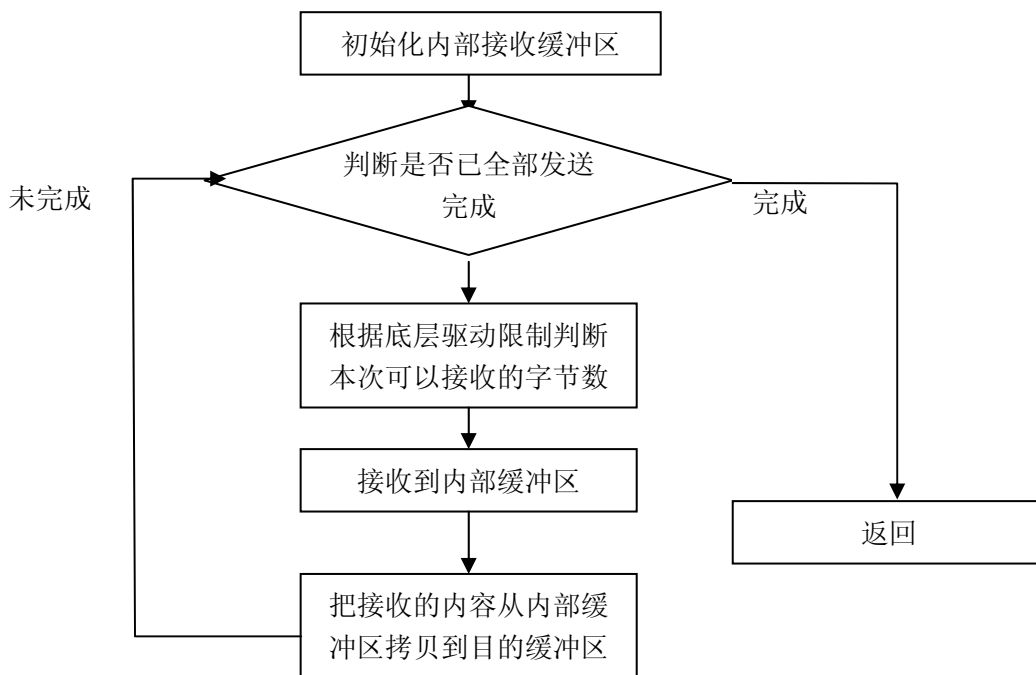


图 4-2 接收原语的实现流程

在虚拟文件系统模块中，主要是由一个主控中心负责具体处理整个接收，发送过程。并根据接收到的不同的命令调用相应的封装过的文件系统函数，最后再把结果通过 USB 返回给操作系统。具体流程如图 4-3：

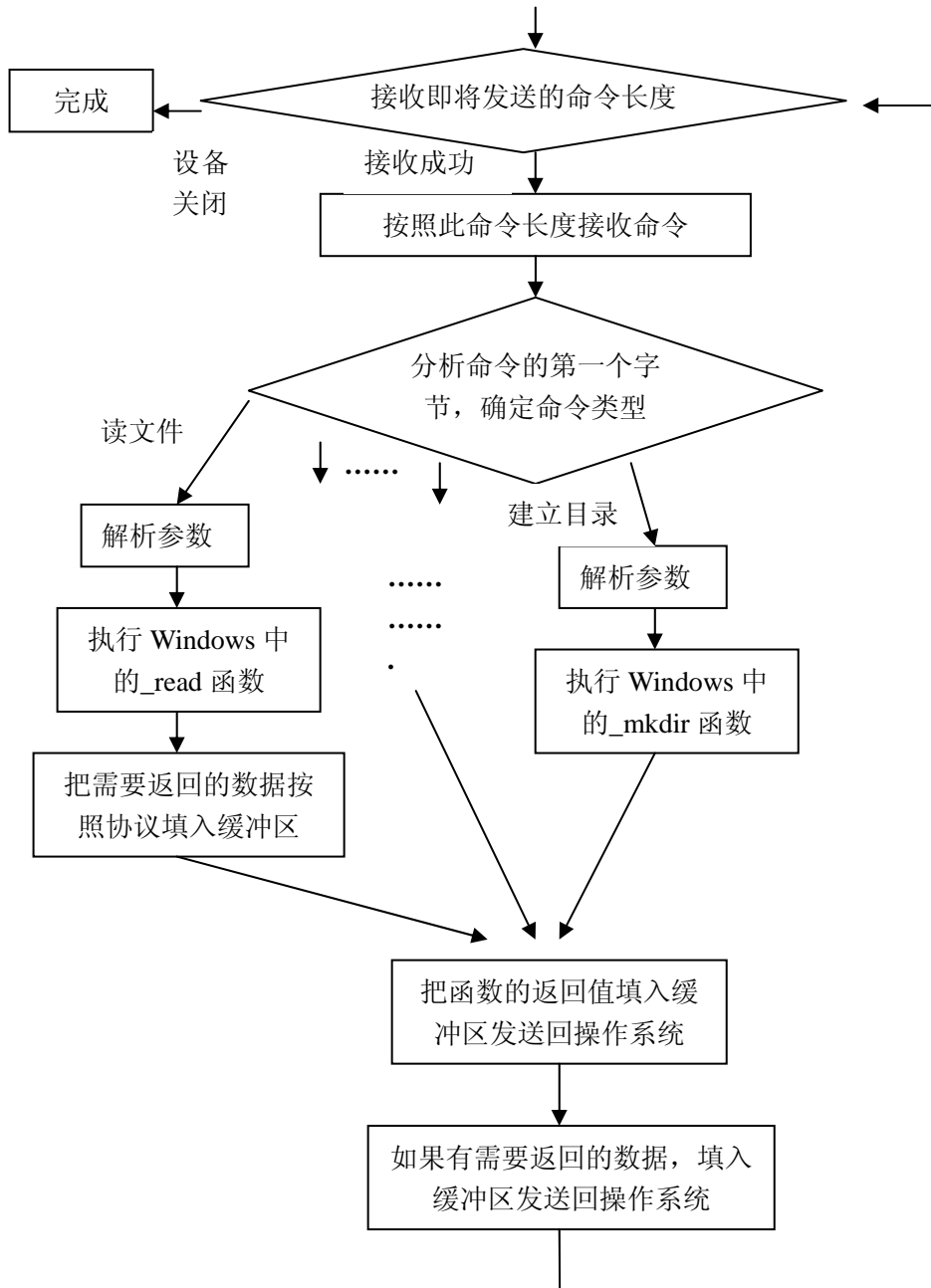


图 4-3 虚拟文件系统主控流程

4.4.3 主要数据结构和函数说明

对应于程序中的两层结构，因此主要的数据结构就是这两层所对应的类：

USB 设备封装类：

```
class CUSBDev
{
public:
    CUSBDev () ; //构造函数
    virtual ~CUSBDev () ; //析构函数
    BOOL CreateDevice () ; //打开设备
    void CloseDevice () ; //关闭设备
    BOOL Send (unsigned char* pBuffer, unsigned int uLen) ;
        //发送指定长度缓冲区内容
    BOOL Receive (unsigned char* pBuffer , unsigned int uLen) ;
        //接收指定长度内容到缓冲区中
protected:
    HANDLE GetDeviceViaInterface (GUID* pGuid,
        DWORD instance) ;
        //获得设备接口，打开设备时使用
    HANDLE m_hUSB ; //USB 设备的句柄
};
```

虚拟文件系统类：

```
class CPCDaemon
{
public:
    CPCDaemon () ; //构造函数
    virtual ~CPCDaemon () ; //析构函数
    CString m_pszCurDir ; //文件系统的当前路径
    CString m_pszRoot ; //文件系统的根目录
};
```

```
void StartDaemon () ;           //启动虚拟文件系统

void StopDaemon () ;           //停止虚拟文件系统

BOOL InitDaemon () ;           //初始化虚拟文件系统
protected:
CUSBDev m_USBDev;              //USB 设备类

int FileGetCwd (char *pResult) ;
                                //获得当前路径

int FileWrite (char *pLast) ; //写入文件

int FileRead (char *pLast, char *pResult, int& iBufLen) ;
                                //从文件读

int FileFindFirst (char *pLast, char *pResult) ;
                                //查找目录中第一个文件

int FileFindNext (char *pLast, char *pResult) ;
                                //查找目录中下一个文件

int FileFindClose (char *pLast) ;
                                //查找结束

int FileCreateDir (char *pLast) ;
                                //新建目录

int FileRemoveDir (char *pLast) ;
                                //删除目录

int FileRemoveFile (char *pLast) ;
                                //删除文件

int FileChangeDir (char *pLast) ;
                                //改变目录

int FileSeek (char *pLast) ; //移动文件指针

int FileClose (char *pLast) ; //关闭文件

int FileOpen (char* pLast) ; //打开文件
```

```
int FileCreate (char* pLast) ;//新建文件  
};
```

第五章 工作的总结与展望

5.1 毕业设计工作的总结

本次工作的主要内容是对 TEOS 虚拟文件系统的 Windows 一端的支持，在这次设计中主要处理的问题包括：

1. 针对 USBN9603 芯片所构成的 USB 设备编写驱动程序。完成设备从插入机器，直到最后进行传输的整个过程。为上层提供了简单的传输接口。
2. 在所编写的 USB 设备驱动程序的基础上，完成了对于虚拟文件系统的系统调用的支持。提供了文件，目录操作的标准接口。

本次工作的几点不足和有待改进的地方主要有以下两个方面：

1. 本次 USB 驱动程序实测的最高传输速度是 160KB/s，与 USB 总线协议的最大传输速度 1.5MB/s 相比还有较大的差距。
2. 在此次的虚拟文件系统系统调用仅仅是对 Windows 的文件系统系统调用进行了最简单的封装，也就是说参数基本上是一一对应的，没有进行较好的扩展。

5.2 对今后工作的展望

今后的工作重点可以放在开发 USB 设备传输的潜力。

- USBN9603 芯片提供 6 个管道，3 个输入和 3 个输出，因此如果可以完全利用起来，合理的调度，那么传输速度应该可以得到较大的提高。当然，由于设备的总线只有一个，因此再多的管道对于 USB 硬件上的传输来说仍然是串行的，但是基于以下几个考虑还是值得在这一方面进行尝试的：
 - ◆ Windows 驱动程序的陷入过程是非常耗时的
 - ◆ USB 设备端的驱动程序可以针对多个管道的传输进行更好的优化
- USBN9603 芯片提供了 DMA 方式的传输。使用 DMA 方式，可以保证在原型系统中几乎不占用 CPU 的情况下进行块传输，这样在原型系统中就节省了大量访问 I/O 端口的时间，使得基于 USB 传输的文件系统调用的执行速度大大提高。

致谢

本次设计工作的顺利完成，首先要感谢我的指导老师：程旭教授。程老师渊博的学识、严谨求实的治学精神、积极进取的生活态度都给我留下了深刻的印象，尤其是他对待事业、对待生活的热情，时刻感染着在他周围的每一个人。他的言传身教将会在我今后的学习、工作和生活中留下不可磨灭的影响，使我更加积极、认真的面对人生。

我要特别感谢同组的管雪涛师兄，他在我的工作过程中给了我很大的帮助和细致的指导，使我能够顺利的完成这次毕业设计。

我还要感谢体系结构实验室的每一位老师和同学，他们共同组成了这个团结互助、积极上进的集体，在这里，我感受到了家一样的温暖。很高兴能够成为这个集体中的一员。

最后，我要感谢我远在家乡的父母和哥哥，是他们始终鼓励着我，督促着我和支持着我。他们将永远是我精神上的最大支柱。

附录：从 DebugPrint 看 USB 设备驱动程序

下面是一个实际的例子，整个过程从设备插入，直到打开设备，进行传输。其中的调试信息就是在此过程中 DebugPrint 所即时打印出来的。从这个例子中可以看出在驱动程序的调试过程中 DebugPrint 所起到的不可忽视的作用。

这些调试信息被分为三个部分，其中前两个都是在设备插入后进行配置的过程，最后一个是打开设备，然后写入一个字符串，然后 USB 设备接到后，会把这个字符串原封不动的传回来，所以最后又接收到了相同的内容。最左边一列数字就是事件发生的时间，第二列文字则是发生的具体事件；黑体的解释文字是整个流程中比较关键的步骤。

插入设备，PNP 开始进行硬件配置

```
03:47:57 RegistryPath is \REGISTRY\MACHINE\SYSTEM\ControlSet001\Services\Usb51
03:47:57 DriverEntry completed
03:47:57 AddDevice
03:47:57 CreateDevice successful!
03:47:57 Device Extension initializing...
03:47:57 FDO is 81CCEDE0
03:47:57 Registering Device Interface
03:47:57 Add Device Successful!
03:47:57 PnpStartDeviceHandler
03:47:57 ForwardIrpAndWait
03:47:57 PnpStartDeviceHandler: post-processing
```

添加新设备

初始化设备扩展结构

成功添加

初始化配置设备

```
03:47:57 Starting Device...
03:47:57 Getting Device Descriptor...
03:47:57 CallUSBDI: waiting for URB completion
03:47:57 Get Device Descriptor successful!
03:47:57 Getting Configuration Descriptor...
03:47:57 Getting basic configuration descriptor
03:47:57 CallUSBDI: waiting for URB completion
03:47:57 Got basic config descr. MaxPower 0 units of 2mA
03:47:57 Basic Configuration Descriptor: total length is 0000003C; number of interface 00000001
03:47:57 configuration value 00000001;Attributes 000000C0
03:47:57 Getting full configuration descriptors
03:47:57 CallUSBDI: waiting for URB completion
03:47:57 status C000009C URB status 80000004
03:47:57 Getting basic configuration descriptor
03:47:57 CallUSBDI: waiting for URB completion
03:47:57 Got basic config descr. MaxPower 0 units of 2mA
```

请求设备描述符

请求配置描述符的基本信息

请求配置描述符的全部信息

失败，需要重新请求

```

03:47:57 Basic Configuration Descriptor: total length is 0000003C; number of interface 00000001
03:47:57 configuration value 00000001;Attributes 000000C0
03:47:57 Getting full configuration descriptors
03:47:57 CallUSBBDI: waiting for URB completion
03:47:57 Full Configuration Descriptor Transfer : 60 bytes
03:47:57 Got full config descr. MaxPower 0 units of 2mA
03:47:57 We will select this interface:number of endpoints 6;Interface number 0
03:47:57 Select configuration worked
03:47:57 interface Class 255 NumberOfPipes 6
03:47:57 Pipes[0] EndpointAddress 81 Interval 64ms PipeType 3 MaximumTransferSize 4096
03:47:57 Pipes[1] EndpointAddress 02 Interval 255ms PipeType 2 MaximumTransferSize 4096
03:47:57 Pipes[2] EndpointAddress 83 Interval 255ms PipeType 2 MaximumTransferSize 4096
03:47:57 Pipes[3] EndpointAddress 04 Interval 255ms PipeType 2 MaximumTransferSize 4096
03:47:57 Pipes[4] EndpointAddress 85 Interval 255ms PipeType 2 MaximumTransferSize 4096
03:47:57 Pipes[5] EndpointAddress 06 Interval 255ms PipeType 2 MaximumTransferSize 4096
03:47:57 Select Configuration Descriptor successful!

```

再次请求成功

描述符中的接口与端点属性

选择接口成功

使用设备进行传输

```

03:49:01 Create File is
03:49:01 USBBDI version 00000300 Supported version 00000100
03:49:01 Getting port status
03:49:01 Got port status 00000003; Get Port status successful!
03:49:01 Reset Device successful!
03:49:01 DeviceIoControl: Control code 00222020 InputLength 0 OutputLength 0
03:49:04 Start to Write 12 bytes to file pointer 0
03:49:04 the buffer is a r c h i t e c t u r e !
03:49:04 Starting Bulk Write
03:49:04 we are using pipe address is : 00000004
03:49:04 Write: 12 bytes written
03:49:06 Read 16 bytes from file pointer 0
03:49:06 we will read 16 bytes
03:49:06 we are using pipe address is : 00000083
03:49:06 Read: 00000000 16 bytes returned
03:49:06 Close

```

打开设备

重置设备

设置为块传输方式

向设备写入字符串

从设备读出字符串

关闭设备

参考文献

- 【1】 Chris Cant 著, 孙义, 马莉波, 国雪飞等译, *Windows WDM 设备驱动程序开发指南*, 机械工业出版社, 2000. 1
- 【2】 Compaq, Intel, Microsoft, NEC, *Universal Serial Bus Specification: Revision 1.1*, 1998. 9
- 【3】 National Semiconductor, *USBN9603 (Universal Serial Bus) Full Speed Function Controller With DMA Support*, 1998. 11
- 【4】 Walter Oney, *Programming the Microsoft Windows Driver Model*, Microsoft Press, 1999
- 【5】 Don Anderson 著, 精英科技译, *USB 系统体系*, 中国电力出版, 2001
- 【6】 张念淮, 江浩编著, *USB 总线接口开发指南*, 北京国防工业出版社, 2001. 3