5　USB 接口的步进电机控制板的驱动程序设计

对于运行于 PC 机上 Windows 操作系统平台下的应用程序而言，任何涉及到对硬件的访问和操作都需要 Windows 操作系统的允许。这是 Windows 操作系统为了保护自身和主机硬件不被恶意的应用程序破坏，所采取的必要措施。在此情况下，如果应用程序想要控制操作系统还未支持的某个硬件设备，就必须遵照操作系统的数据传输流程和规定的架构模型，编写一个运行于操作系统内核之中的程序模块，在此就是系统中的设备驱动程序。本章将在介绍必要的 Windows 操作系统中 USB 驱动程序结构的基础上，详细介绍开发 USB 接口的步进电机控制板驱动程序的方法和具体步骤。

5. 1　Windows 2000/XP 中的设备驱动程序模型 WDM

WDM(Windows Driver Model)是 Microsoft 公司力推的一种符合 Windows2000/XP 下的内核模式驱动程序的分层体系结构的驱动程序模型。它源于 Windows NT 的分层 32 位设备驱动程序模型，并支持更多的特性，如即插即用（ PnP ，Plug and Play ）、电源管理（ PM ，Power Management ）、Windows 管理诊断（ WMI ，Windows Management Instrumentation ）和 NT 事件。它为 Windows 操作系统的设备驱动程序提供了统一的框架，在 Windows 平台上，WDM 将成为主流的驱动模式。WDM 驱动程序在 Windows 系统的整体驱动结构中的位置如图 5-1 所示。

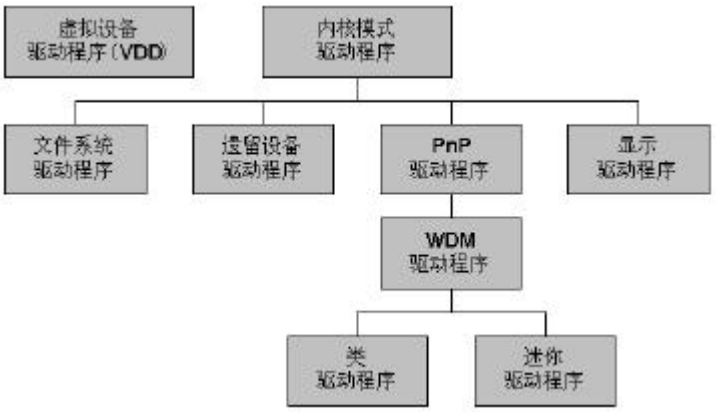

图 5-1 Windows 系统的整体驱动结构

在 WDM 驱动程序模型中，每个硬件设备至少有两个驱动程序。其中一个驱动程序为功能(Function)驱动程序，通常它就是硬件设备驱动程序。它了解使硬件工作的所有细节，负责初始化 I/O 操作，有责任处理 I/O 操作完成时所带来的中断事件，有责

任为用户提供一种设备适合的控制方式。另一个驱动程序我们称为总线(bus)驱动程序。它负责管理硬件与计算机的连接。例如，USB 总线驱动程序检测插入到 USB 总线上的设备插入时,它会自动确定设备的资源使用情况，并且还能控制设备所在 USB 总线的电流开关。

WDM 模型使用了如图 5-2 的层次结构。图中左边是一个设备对象堆栈。设备对象是系统为帮助软件管理硬件而创建的数据结构。一个物理硬件可以有多个这样的数据结构。处于堆栈最底层的设备对象称为物理设备对象(Physical Device Object)，或简称为 PDO。在设备对象堆栈的中间某处有一个对象称为功能设备对象(Functional Device Object)，或简称 FDO。在 FDO 的上面和下面还会有一些过滤器设备对象(Filter Device Object)。位于 FDO 上面的过滤器设备对象称为上层过滤器，位于 FDO 下面(但仍在 PDO 之上)的过滤器设备对象称为下层过滤器。一个 PDO 对应一个真实的硬件。一个硬件只允许有一个 PDO，却可以拥有多个 FDO，在驱动程序中直接操作的不是硬件，而是相应的 PDO 和 FDO。

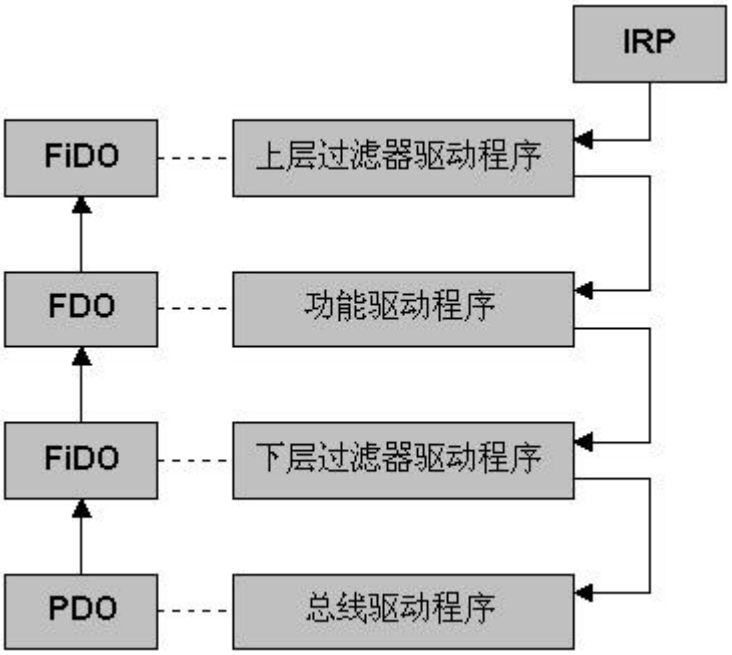

图 5-2 WDM 中设备对象和驱动程序的层次结构

可以把一个完整的驱动程序看作是一个容器，它包含许多例程，当操作系统遇到一个 IRP 时，它就调用这个容器中的例程来执行该 IRP 的各种操作。图 5-3 表现了这个概念。有些例程，例如 DriverEntry 和 AddDevice，还有与几种 IRP 对应的派遣函数将出现在每一个这样的容器中。需要对 IRP 排队的驱动程序一般都有一个 StartIo 例

程。执行 DMA 传输的驱动程序应有一个 AdapterControl 例程。大部分能生成硬件中断的设备，其驱动程序都有一个中断服务例程(ISR)和一个延迟过程调用(DPC)例程。驱动程序一般都有几个支持不同类型 IRP 的派遣函数，其中三个派遣函数是必须的。所以，WDM 驱动程序开发者的任务就是为这个容器选择并设计所需要的例程。

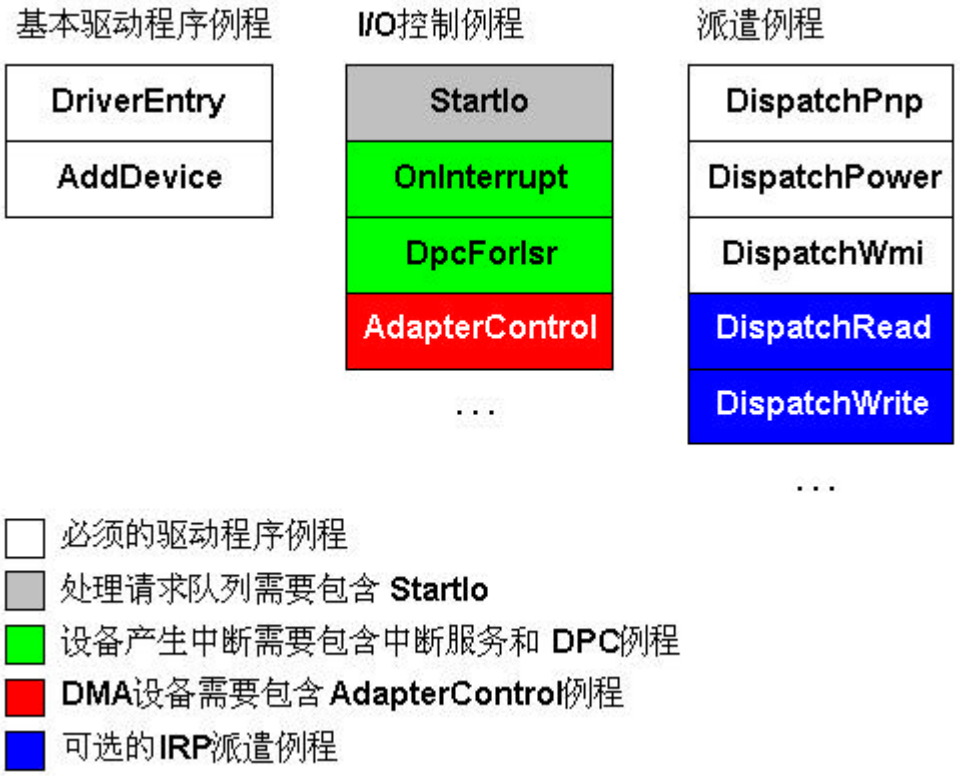

图 5-3 驱动程序的各种例程

WDM 是通过一个 128 位的全局唯一标识符(GUID)实现驱动程序的识别。应用程序与 WDM 驱动程序通信时，应用程序将每用户请求形成 I/O 请求包(IRP)发送到驱动程序。驱动程序识别出 IRP 请求后指挥硬件执行相应操作。

5. 2 WDM 型的 USB 驱动程序结构

对于 USB 设备来说，其 WDM 驱动程序分为 USB 底层（总线）驱动程序和 USB 功能(设备)驱动程序。USB 驱动程序符合 Windows 2000 下的内核模式驱动程序的分层体系结构，如图 5-4 所示：

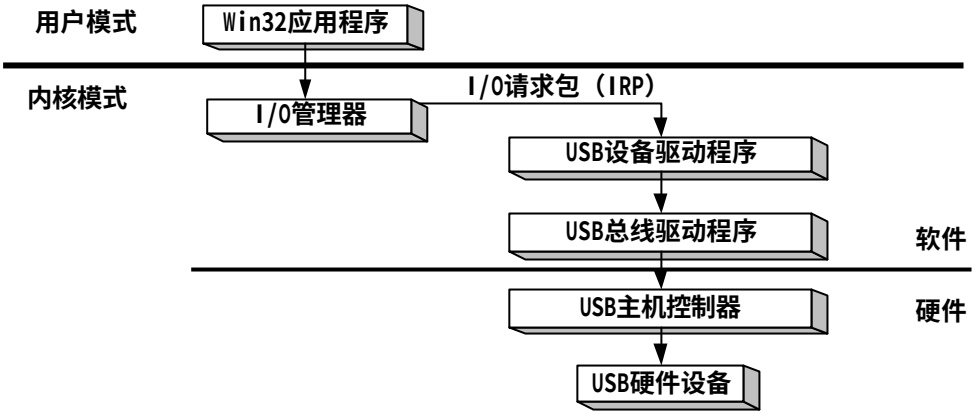

图 5-4　WDM 型的 USB 驱动程序体系结构

USB 底层驱动程序由操作系统提供，负责与实际的硬件打交道，实现繁琐的底层通信。USB 功能驱动程序由设备开发者编写，不对实际的硬件进行操作，而是通过向 USB 底层驱动程序发送包含 URB（USB Request Block，请求块）的 IRP，来实现对 USB 设备信息的发送和接收。采用这种分层驱动程序的设计方法有两个优点：(1)多个 USB 设备可以通过 USB 底层驱动程序来协调它们的工作。(2)编写分层驱动程序较之编写单一驱动程序相对简单，且可以节省内存和资源，不易出错。

USB 驱动程序工作简述如下：当应用程序想对 USB 设备进行 I/O 操作，它需调用 Windows API 函数，I/O 管理器将此请求构造成一个合适的 I/O 请求包（IRP）并把它传递给 USB 功能驱动程序。USB 功能驱动程序接收到这个 IRP 后，根据 IPR 中包含的具体操作代码构造相应 USB 请求块（URB），并把此 URB 放到一个新的 IRP 中，然后把它传递给 USB 底层驱动程序。USB 底层驱动程序根据 IRP 中所含的 URB 执行相应的操作，并把操作的结果返回给 USB 功能驱动程序。USB 功能驱动程序接收到此返回的 IRP 后，将操作结果通过 IRP 返还给 I/O 管理器，最后 I/O 管理器将此 IRP 操作结果传回给应用程序，至此应用程序对设备的一次 I/O 操作完成。

如图 5-5, 当将读/写请求发往一个 USB 设备时(USB 设备插入 USB hub，USB hub 再插入主机控制器)，USB 设备堆栈、hub 设备堆栈、主机控制器设备堆栈将加入 Windows 系统设备树。IRP_MJ_WRITE/IRP_MJ_READ 请求先穿过 FiDO 到达功能驱动程序，然后功能驱动程序生成一个或多个不同种类的 IRP 并发送到自己的 PDO。USB 设备的 PDO 驱动程序就是 USBHUB.SYS，该驱动程序把这些 IRP 送到主机控制器堆栈的最上层驱动程序，跳过中间的 USB 集线器堆栈(含有两个驱动程序)。

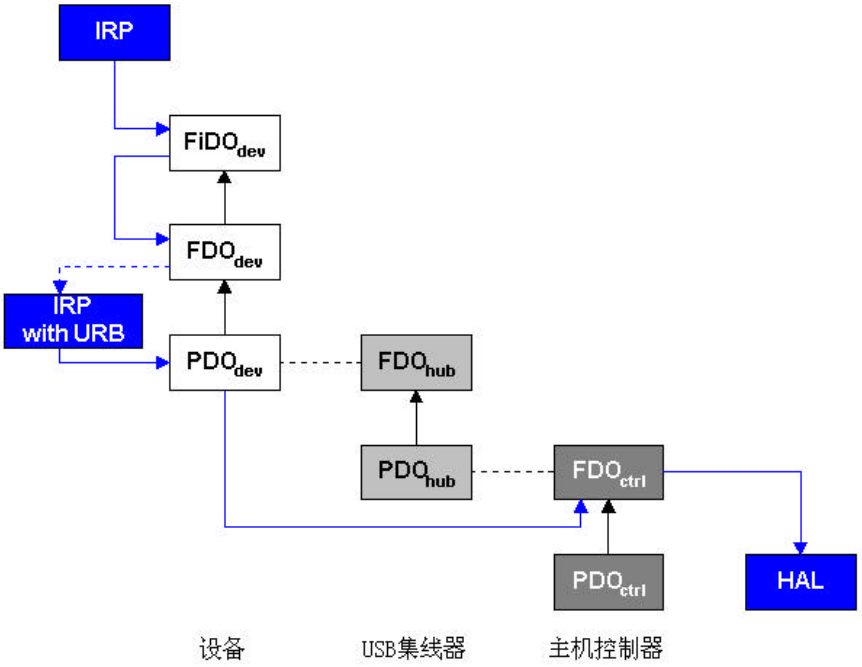

图 5-5　USB 驱动处理流程图

关于 USB 设备的总线驱动程序，具体请参考 2.3 节 “主机 USB 系统驱动调用层次”。该节对USB设备的总线驱动程序的层次结构及各个驱动模块有较详细的说明。

本系统中将要开发的设备驱动,便是设备的功能驱动程序(FDO)。因为本系统并不需要设备的过滤驱动程序，故无需开发 FiDO 部分。

5. 3　利用 DriverStudio 3.1 开发 WDM 型的 USB 设备驱动程序

目前开发 WDM 驱动程序的方法有三种：

(1) 使用 Microsoft 的 Windows2000/XP DDK 工具开发。

(2) 使用 KRFTech 公司的 WinDriver 。

(3) 使用 NuMega 公司的 DriverStudio 。

以上三种 WDM 驱动程序开发方法，笔者都曾尝试过。个人认为用 DriverStudio 开发工具包来开发 USB 驱动程序最有效。DriverStudio 中的 Driver Wizard 开发向导是创建 WDM 驱动程序框架的一个很好的工具，后面将利用它来创建 USB 设备驱动程序的基本框架。

5. 3. 1 建立开发平台

由于利用 DriverStudio 开发 WDM 驱动程序在搭建开发平台的过程中对软件的安装顺序要求颇高，否则在开发过程中会因为安装顺序的颠倒而失败。笔者在实践中总结了以下的安装步骤：

(1) 在已装了 Windows 2000 操作系统的主机上安装 Microsoft Visual C++6.0。

(2) 安装 Win2000 DDK 以及 Platform SDK。

(3) 安装 NuMega DriverStudio 3.1 驱动程序开发工具包。它包含 DriverWorks（用于开发内核模式 WDM 驱动程序）、SoftICE（用于调试 WDM 驱动程序）等开发工具。

(4) 设置 DDK 路径。

(5) 由于 DriverWorks 所用的类库是对 DDK 函数的封装，必须在 VC 中编译，创建自己的库文件。

5. 3. 2 利用 DriverStudio 的 DriverWorks 生成 USB 设备驱动程序框架

驱动程序开发平台搭建成功后，进入 Microsoft Visual C++6.0 中，可利用驱动程序生成向导 Driver Wizard，根据硬件设置生成 USB 设备驱动程序的整体框架。

本系统的具体配置步骤有 14 步，每步的具体配置如下：

(1) 在 New Project 菜单中选择 DriverWorks Project 选项。

(2) 填写工程名称及存放位置。Project:USB_Motor ; Location:E:\Mywork\USB_Motor

(3) 选择 WDM Driver 的驱动程序类型。

(4) 选择 WDM Function Driver 即功能驱动程序模型。

(5) 选择 USB 总线类型，系统选择的 USB 芯片是 Philip 公司的 PDIUSBD12，填写它的 VID（供应商 ID）和 PID（设备 ID），这些信息由芯片的供应商提供。但是，这两部分的内容、只要与步进电机控制板的固件代码中的对应部分相同就可以了。

(6) 添加端点 1 和端点 2，它们都分别具有 IN 和 OUT 属性。

具体对应设置，如表 5-1。

表 5-1 端点设置表

Endpoint Address		Endpoint Type	Max Transfer Size	Endpoint Name
1	In	Interrupt	16	Endpoint1In
	Out	Interrupt	16	Endpoint1Out
2	In	Bluck	64	Endpoint2In
	Out	Bluck	64	Endpoint2Out

(7) 填写 Driver Class: USB_Motor ；File Name:USB_Motor 。

(8) 根据需要选择对设备的操作有：Read、Write、Device Control 和 Cleanup。

(9) 选择给端点 2 产生 BULK 传输模式的 Read 和 Write 的代码，向导会自动产生一套对端点 2 进行 Read、Write 的代码。

(10) 在 Select Queueing Method 选项中选择 None-requests are not queued。这表示不对请求进行排队。

(11) 本步骤对 Windows 的注册表、进行设备相关的表项添加。这些表项是在设备启动时供设备读回的。本系统未用到这些表项，故可以不作添加而直接进入下一步。

(12) 本步中有三个选项卡，现在依次说明如下：

Interface 选卡: Device name: USB_MotorDevice ；在 Open device 选项中选 Interface,这样 DriverStudio 会自动为驱动程序产生 GUID；Exclude 选项可以不选，该选项表示是否只允许应用程序打开一个设备句柄。

Buffers 选卡: 本选卡主要是对 Read（）和 Write（）例程选择从用户缓冲区到系统缓冲区的数据传输方式。在选取读写方式时应遵循一条原则：需要快速传送大量数据时，用 Direct I/O ，反之用 Buffer I/O ,这里选择 BufferI/O（注意：这里选择的 I/O 模式仅仅是对 Read（）和 Write（）例程而言的，对 DeviceControl（）例程毫无影响。对 DeviceControl（）例程中的、各项 IOCTL 选项的相应部分设置，在(13)中进行。）。

Power 选卡: 由于无特殊的电源需求，故选用系统默认的 Manage Power For This Device。

(13) 添加 IOCTL 接口，在其生成的代码框架中加入自己的操作，以实现一个完整的 USB 设备驱动程序。

添加项的具体数据如下表 5-2:

表 5-2 控制命令设置表

Parameter Name	Ordinal	Method	Access
RUN	0X800	Buffered	Any
STOP	0X801	Buffered	Any
X_SELECT	0X802	Buffered	Any
Y_SELECT	0X803	Buffered	Any
OPPOSITE	0X804	Buffered	Any
NEGATIVE	0X805	Buffered	Any
RESET	0X806	Buffered	Any
SIN_WAITE	0X807	Buffered	Any
STEPNUMS	0X808	Buffered	Any
PLUSWIDTH	0X809	Buffered	Any

(14)本步骤中对驱动程序的附加选项进行选择：其中最重要的是 Debug Options 选项系列。Enable Break at Driver Entry 表示设备驱动是否能在启动之初就可以进行调试；Generate Trace Code 表示是否生成跟踪调试代码部分。上述这两个选项对驱动程序调试而言，实在是不可多得的利器，因此一定要选上。

至此，就生成了一个 WDM 型的 USB 设备驱动程序框架和一个测试该驱动程序的测试程序大体框架。然后可以在其中添加需要的功能代码。

5. 4 USB 设备驱动程序中的关键例程代码实现

在本节中，将在 5.3 节所生成的驱动代码的基础上，对代码中的相应部分做添加和修改。最终将生成本系统所需的完整的驱动程序代码。

5. 4. 1 USB_Motor.cpp 中的修改

在 AddDevice 例程中，在设备的特征标志位中，添加 DO_POWER_PAGABLE，否则，关机时将报告驱动程序状态错误。

具体修改如下：

```
USB_MotorDevice * pDevice = new (
static_cast<PCWSTR>(KUnitizedName(L"USB_MotorDevice", m_Unit)),
        FILE_DEVICE_UNKNOWN,
```

```
            NULL,
            0,
            DO_DIRECT_IO
                | DO_POWER_PAGABLE//必须在 PASSIVE_LEVEL 级上处理
            )                                   //IRP_MJ_PNP 请求
        USB_MotorDevice (Pdo, m_Unit);
```

5. 4. 2 USB_MotorDevice.h 中的修改

具体添加如下：

```
    protected:                                  // Added by IceFire on 2005.3.18
    ULONG                   m_Unit;   //设备的单元号
    KUsbLowerDevice         m_Lower;   //实例化 USB 下层驱动
    KUsbInterface       m_Interface;   //设备接口实例化
    KUsbPipe   m_Endpoint1IN; // Pipe for USB endpoint 1 IN (address 0x81)
    KUsbPipe   m_Endpoint1OUT // Pipe for USB endpoint 1 OUT (address 0x1)
    KUsbPipe   m_Endpoint2IN; // Pipe for USB endpoint 2 IN (address 0x82)
    KUsbPipe   m_Endpoint2OUT; // Pipe for USB endpoint 2 OUT (address 0x2)
    PUCHAR        m_buffer;     //设备缓冲区
    KIrp          m_kIrp;       //设备自己的 Irp
    PURB          m_pUrb;     //USB 请求块
    KIoWorkItem        m_pItem; //工作项队列，相当与 Linux 中的 BH
```

5. 4. 3 USB_MotorDevice.cpp 中的修改

对设备类实例的构造函数 USB_MotorDevice（）进行修改如下。该函数在设备接入主机的 USB 总线时，由 Windows 系统自动调用。

```
USB_MotorDevice:: USB_MotorDevice (PDEVICE_OBJECT Pdo, ULONG Unit) :
    KPnpDevice(Pdo, & USB_MotorDevice _Guid)
{
    t << "Entering USB_MotorDevice:: USB_MotorDevice (constructor)\n";
    if ( ! NT_SUCCESS(m_ConstructorStatus) )
    {
        return;
    }
```

```
    m_Unit = Unit; //  记录设备单元号
    m_Lower.Initialize(this, Pdo); //  初始化 USB 底层设备
    m_Interface.Initialize(        //初始化 USB 设备接口
                    m_Lower, //KUsbLowerDevice
                    0,          //选择 0 号接口
                    1,          //选择 1 号配置
                    0           //无可选接口配置
                    );

    m_Endpoint1IN.Initialize(m_Lower, 0x81, 16); //  初始化各个端点对象
    m_Endpoint1OUT.Initialize(m_Lower, 0x1, 16);
    m_Endpoint2IN.Initialize(m_Lower, 0x82, 64);
    m_Endpoint2OUT.Initialize(m_Lower, 0x2, 64);

    SetLowerDevice(&m_Lower); //  与底层的设备对象的基类建立联系
    SetPnpPolicy();//建立缺省的标准 PNP 策略
    SetPowerPolicy();//建立缺省的标准电源管理策略；
    m_pItem.Initialize(Pdo);//初始化工作项队列
}
```

在 OnStartDevice(KIrp I)中添加下列代码，以在设备启动时激活相关配置。

```
acStatus = m_Lower.ActivateConfiguration(
        1    //激活 1 号配置
        );
```

5. 4. 3 USB_MotorDevice.cpp 中的 DeviceControl( )例程编写

在本系统中最重要的函数莫过于 DeviceControl(KIrp I)了。它负责本系统全部的命令传递动作。现全部列出如下：

```
NTSTATUS USB_MotorDevice::DeviceControl(KIrp I)
{
    NTSTATUS status;
    t << "Entering USBINTDevice::Device Control, " << I << EOL;
    switch (I.IoctlCode())   //对应用程序传来的 dwIoControlCode 进行解析
    {                        //系统控制命令处理函数，对应见表 5-3。
```

```
  case RUN:
        status = RUN (I);
        break;
  case STOP:
        status = STOP (I);
        break;
  case X_SELECT:
        status = X_SELECT (I);
        break;
  case Y_SELECT:
        status = Y_SELECT (I);
        break;
case OPPOSITE:
        status = OPPOSITE (I);
        break;
case NEGATIVE:
        status = NEGATIVE (I);
        break;
case RESET:
        status = RESET (I);
        break;
case SIN_WAITE:
        status = SIN_WAITE (I);
        break;

case STEPNUMS:    //系统运动参数设置命令处理函数，对应见表 5-4。
        status = STEPNUMS (I);
        break;
case PLUSWIDTH:
        status = PLUSWIDTH (I);
        break;
  default:                // 无法识别的 IOCTL 请求
```

```
        status = STATUS_INVALID_PARAMETER;
                break;
        }
    if (status == STATUS_PENDING)
        {
            return status;
        }
        else
        {
            return I.PnpComplete(this, status);
        }
    }
```

上层应用程序通过打开驱动模块接口文件来获得系统核心的准入控制，此后就可以调用 DeviceIoControl,ReadFile,WriteFile 这三个函数与驱动模块通信了。在上层应用程序调用这三个函数时，可以指定驱动程序的控制命令类型，对于 DeviceIoControl（），这个参数为 dwIoControlCode。并且进一步的，可以将对步进电机的控制数据，放入指定的缓冲区中供 USB 模块读取，同样对于 DeviceIoControl（），这个缓冲区对应着 lpInBuffer 和 nInBufferSize 两个参数。在本系统中，我们选择将两者结合的方式与 USB 驱动模块通信，如表 5-3 和表 5-4。

表 5-3 为系统控制命令列表,其中 dwIoControlCode 及其含意如表,此处 lpInBuffer＝NULL，nInBufferSize＝0。

表 5-3 系统控制命令列表

dwIoControlCode	控制命令含意	驱动程序的 DeviceControl（）例程中的对应处理函数
RUN	启动	RUN（）
STOP	停止	STOP（）
X_SELECT	选择 X 轴驱动	X_SELECT（）
Y_SELECT	选择 Y 轴驱动	Y_SELECT（）
OPPOSITE	正方向运动	OPPOSITE（）
NEGATIVE	负方向运动	NEGATIVE（）
RESET	系统重启	RESET（）
SIN_WAITE	等待设备状态回复	SIN_WAITE（）

表 2 为系统运动参数设置命令列表，其中 dwIoControlCode ，lpInBuffer 和 nInBufferSize 的含意如表。

表 5-4 系统运动参数设置命令列表

dwIoControlCode	lpInBuffer	nInBufferSize	控制命令含意	DeviceControl（）例程中的对应处理函数
STEPNUMS	0 ～ 1000	4	步进电机步数设定	STEPNUMS（）
PLUSWIDTH	0 ～ 1000	4	驱动脉冲宽度微调	PLUSWIDTH（）

当应用程序调用 DeviceIoControl 时，系统会将我们赋值给 dwIoControlCode ，lpInBuffer 和 nInBufferSize 的数值封装在一个 IRP（系统中断请求包数据结构）的对应域中，然后传递给驱动程序。这个过程对应于图 5-5 中的 IRP 先传递到 FiDO，然后再传递到 FDO_{dev}(USB_Motor.SYS)的过程。

当 IRP 传递到 USB_Motor.SYS 后，系统自动调用其中的 DeviceControl（）例程。我们在 DeviceControl（）例程中，对 IRP 中的对应域进行解析：用 I.IoctlCode（）提取 dwIoControlCode ；用 I. IoctlBuffer（）提取 lpInBuffer；用 I.Information()提取 nInBufferSize。通过对以上三个参数的解析，选择调用对应的处理函数。其驱动程序的 DeviceControl（）例程中的对应处理函数，如表 5-3 和表 5-4 中的最后一列 。在这些对应的处理函数中，我们根据提取的参数，并对其值进行调整，然后将其作为，对 BuildVendorRequest()函数的输入参数，调用该函数构建 URB。创建完 URB 后，需要创建并发送一个内部 I/O 控制(IOCTL)请求的 IRP 到 USBD 驱动程序，即 USBD.SYS。USBD 驱动程序位于驱动程序层次结构的底端，也就是图 5-5 中的 PDO_{dev}。在大多数情况下，需要等待设备回应。驱动程序把 URB 包装成一个带有 IRP_MJ_INTERNAL_DEVICE_CONTROL 主功能码的普通 IRP。为了使上层过滤器驱动程序可以发送自己的 URB，每个 USB 设备驱动程序应有一个可以把 IRP 传递到下一层的派遣函数。当提交一个 URB 到 USB 总线驱动程序时，最终将收到一个描述该操作结果的 NTSTATUS 代码。上面这个过程对应于图 5-5 中 IRP 从 FDO_{dev} 传递到 PDO_{dev} 的过程。

现在分别列出具有代表性的，表 5-3 中的 RUN（）代码和表 5-4 中的 STEPNUMS（）代码，如下：

TSTATUS USB_MotorDevice:: RUN (KIrp I)

```
{
    PURB    pUrb;
    NTSTATUS    status;
    UCHAR    buffer[16];     //传输缓冲区 16 字节

    I.Information() = 0;
    memset(buffer,0,16);            //传输缓冲区清 0
    m_pUrb = m_Lower.BuildVendorRequest(
                buffer,             //传输缓冲区
                16,                 //传输缓冲区大小
                0,                  // 无请求保留位
                0,                  //厂商请求号，具体参见表 4-1，此处为 0。
                0,          // 请求数值，对应为 wValue，此处为 0，表示无数据。
               FALSE,            //  方向为输出
               TURE,            //传输数据可小于 16
               NULL,            //无链接 urb
               0,                //索引号对应于 wIndex。
               URB_FUNCTION_VENDOR_DEVICE    //默认的固定方程
                );

    if ( m_pUrb == NULL ) //若分配 urb 不成功，就返回资源不足状态码。
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }
     else
    {                   //向 USB 总线发送 USB 请求包
     status=m_Lower.SubmitUrb(pUrb);
     delete pUrb;    //发送完毕后，删除 pUrb 以避免无效引用和内存泄漏
    }
return status;          //返回发送状态
}
```

```
TSTATUS USB_MotorDevice:: STEPNUMS (KIrp I)
{
    PURB    pUrb;
    NTSTATUS    status;
    UCHAR    buffer[16];     //传输缓冲区 16 字节

    UCHAR    value[1];        //步进电机运动步数保存缓冲区
   strncpy( value ,(PCHAR)I.IoctlBuffer(),2)//拷贝应用程序下传的数据到 value

    I.Information() = 0;
    memset(buffer,0,16);            //传输缓冲区清 0
    m_pUrb = m_Lower.BuildVendorRequest(
                buffer,                 //传输缓冲区
                16,                     //传输缓冲区大小
                0,                      //  无请求保留位
                8,                      //厂商请求号，具体参见表 4-1，此处为 8
                Value[1],// 请求数值，此处为应用程序设置的步进电机运动步数
                FALSE,             //  方向为输出
                TURE,            //传输数据可小于 16
                NULL,            //无链接 urb
                0,                //索引号对应于 wIndex
                URB_FUNCTION_VENDOR_DEVICE    //默认的固定方程
                  );
    if ( m_pUrb == NULL ) //若分配 urb 不成功，就返回资源不足状态码
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }
     else
    {                   //向 USB 总线发送 USB 请求包
     status=m_Lower.SubmitUrb(pUrb);
     delete pUrb;    //发送完毕后，删除 pUrb 以避免无效引用和内存泄漏
    }
```

```
return status;          //返回发送状态
}
```

5.5 本章小结

本章首先介绍了 WDM 驱动程序模型的结构和驱动程序例程，随后对 WDM 型的 USB 驱动程序结构进行了具体分析。对利用 Driver Studio 3.1 工具包、开发本系统的 WDM 型 USB 设备驱动程序（USB_Motor.SYS）的步骤，进行了详细地介绍。最后，对初步生成的驱动程序 USB_Motor.SYS 的框架代码进行相应修改和添加，最终生成完整的步进电机控制板驱动程序（USB_Motor.SYS）。