第 1 章 从两个最简单的驱动谈起

Windows 驱动程序的编写，往往需要开发人员对 Windows 内核有深入了解和大量的内核调试技巧，稍有不慎，就会造成系统的崩溃。因此，初次涉及 Windows 驱动程序开发的程序员，即使拥有大量 Win32 程序的开发技巧，往往也很难入门。

本章向读者呈现两个最简单的 Windows 驱动程序，一个是 NT 式的驱动程序，另一个是 WDM 式的驱动程序。这两个驱动程序没有操作具体的硬件设备，只是在系统里创建了虚拟设备。在随后的章节中，它们会作为基本驱动程序框架，被本书其他章节的驱动程序开发所复用。笔者将带领读者编写代码、编译、安装和调试程序。相信对第一次编写驱动程序的读者来说，这将是非常激动和有趣的。代码的具体讲解将分散在后面的章节论述。现在请和笔者一起，开始 Windows 驱动编程之旅吧！

1.1 DDK 的安装

在编写第一个驱动之前，需要先安装微软公司提供的 Windows 驱动程序开发包 DDK（Driver Development Kit）。笔者计算机里安装的是 Windows XP 2462 版本的 DDK，建议读者安装同样版本或者更高版本的 DDK，如图 1-1 所示。

在安装的时候请选择完全安装，即安装 DDK 的所有部件，如图 1-2 所示。因为除了 DDK 的基本编译环境外，DDK 还提供了大量的源代码和实用工具，这对于 Windows 驱动程序的初学者进行学习和编写驱动程序将是非常有用的。

安装完毕后，会在开始菜单中出现相应的项目。其中，主要用到的是 Build Environment，如图 1-3 所示。该版本的 DDK 会同时安装上 Windows 2000 和 Windows XP 的编译环境。

图 1-1　DDK 的安装

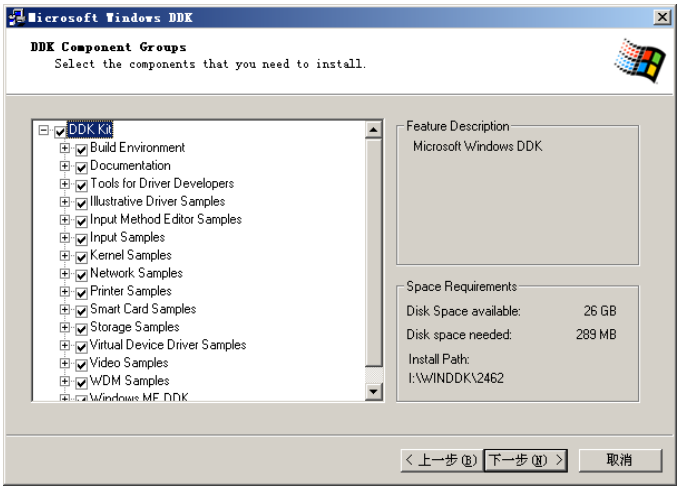


图 1-2　DDK 的安装

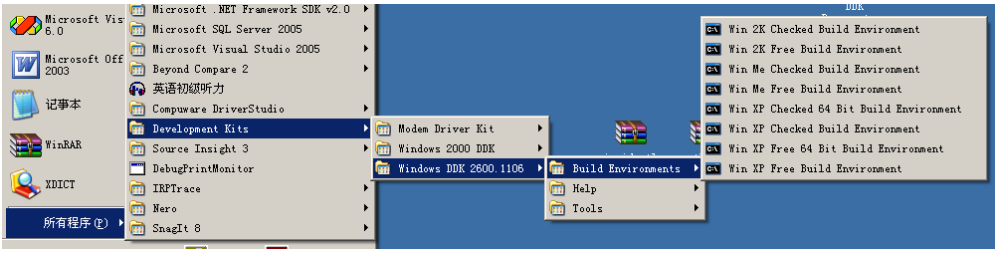


图 1-3　DDK 的编译环境

1.2　第一个驱动程序 HelloDDK 的代码分析

Windows 驱动程序分为两类，一类是不支持即插即用功能的 NT 式驱动程序，另一类

是支持即插即用功能的 WDM 驱动程序。本节介绍的 HelloDDK 是一个最简单的 NT 式驱动程序。在 1.4 节中将给出一个 WDM 式的驱动程序。

1.2.1 HelloDDK 的头文件

HelloDDK 的头文件主要是为了导入驱动程序开发所必需的 NTDDK.h 头文件，此头文件里包含了对 DDK 的所有导出函数的声明。NT 式的驱动程序要导入的头文件是 NTDDK.h，而 WDM 式的驱动程序要导入的头文件为 WDM.h。另外，此头文件中定义了几个标签，分别在程序中指明函数和变量分配在分页内存中或非分页内存中（分页和非分页内存的概念将在第 3 章中讲述）。最后，该头文件给出了此驱动的函数声明。

```
#001  /*************************************************************************
#002  * 文件名称:Driver.h
#003  * 作    者:张帆
#004  * 完成日期:2007-11-1
#005  *************************************************************************/
#006  #pragma once
#007
#008  #ifdef   cplusplus
#009  extern "C"
#010  {
#011  #endif
#012  #include <NTDDK.h>
#013  #ifdef __cplusplus
#014  }
#015  #endif
#016
#017  #define PAGEDCODE code_seg("PAGE")
#018  #define LOCKEDCODE code seg()
#019  #define INITCODE code_seg("INIT")
#020
#021  #define PAGEDDATA data seg("PAGE")
#022  #define LOCKEDDATA data_seg()
#023  #define INITDATA data_seg("INIT")
#024
#025  #define arraysize(p) (sizeof(p)/sizeof((p)[0]))
#026
#027  typedef struct  DEVICE EXTENSION {
#028    PDEVICE_OBJECT pDevice;
#029    UNICODE_STRING ustrDeviceName;      //设备名称
#030    UNICODE_STRING ustrSymLinkName;     //符号链接名
#031  } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
#032
#033  // 函数声明
#034
#035  NTSTATUS CreateDevice (IN PDRIVER_OBJECT pDriverObject);
#036  VOID HelloDDKUnload (IN PDRIVER_OBJECT pDriverObject);
#037  NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE_OBJECT pDevObj,
#038                                   IN PIRP pIrp);
#039
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 代码 6~15 行，包含 ddk.h 头文件，所有的 NT 式驱动程序都要包含此头文件。因为这里采用的是 C++语言编写，如果直接包含 ntddk.h，函数的符号表会导入错误，所以需要加入 extern "C"，这样可以保证符号表正确导入。关于 C++编写驱动需要注意的地方，将在第 3 章进行论述。
- 代码 17~23 行，定义分页标记、非分页标记和初始化内存块。在 Windows 驱动程序的开发中，所有程序的函数和变量要被指明被加载到分页内存中还是在非分页内存中。程序代码中加入这里定义的宏，就会被指明函数和变量是位于分页或非分页内存中。另外，有一个特殊的函数 DriverEntry 需要放在 INIT 标志的内存中。INIT 标志指明该函数只是在加载的时候需要载入内存，而当驱动程序成功加载后，该函数可以从内存中卸载掉。
- 代码 27~31 行，指定一个设备扩展结构体，这种结构体广泛应用于驱动程序中。根据不同驱动程序的需要，它负责补充定义设备的相关信息。
- 代码 33~38 行是函数的声明。

1.2.2　HelloDDK 的入口函数

和普通的应用程序不同，Windows 驱动程序的入口函数不是 main 函数，而是一个叫做 DriverEntry 的函数，代码将在下面列出。DriverEntry 函数由内核中的 I/O 管理器负责调用，其函数有两个参数：pDriverObject 和 pRegistryPath。其中，pDriverObject 是 I/O 管理器传递进来的驱动对象，pRegistryPath 是一个 Unicode 字符串，指向此驱动负责的注册表。

```
#001  /*************************************************************************
#002  * 文件名称:Driver.cpp
#003  * 作　　者:张帆
#004  * 完成日期:2007-11-1
#005  *************************************************************************/
#006
#007  #include "Driver.h"
#008
#009  /*************************************************************************
#010  * 函数名称:DriverEntry
#011  * 功能描述:初始化驱动程序，定位和申请硬件资源，创建内核对象
#012  * 参数列表:
#013      pDriverObject:从 I/O 管理器中传进来的驱动对象
#014      pRegistryPath:驱动程序在注册表的中的路径
#015  * 返回值:返回初始化驱动状态
#016  *************************************************************************/
#017  #pragma INITCODE
#018  extern "C" NTSTATUS DriverEntry (
#019              IN PDRIVER_OBJECT pDriverObject,
#020              IN PUNICODE_STRING pRegistryPath    )
#021  {
#022    NTSTATUS status;
```

```
#023     KdPrint(("Enter DriverEntry\n"));
#024
#025     //注册其他驱动调用函数入口
#026     pDriverObject->DriverUnload = HelloDDKUnload;
#027     pDriverObject->MajorFunction[IRP_MJ_CREATE] = HelloDDKDispatchRoutine;
#028     pDriverObject->MajorFunction[IRP_MJ_CLOSE] = HelloDDKDispatchRoutine;
#029     pDriverObject->MajorFunction[IRP_MJ_WRITE] = HelloDDKDispatchRoutine;
#030     pDriverObject->MajorFunction[IRP_MJ_READ] = HelloDDKDispatchRoutine;
#031
#032     //创建驱动设备对象
#033     status = CreateDevice(pDriverObject);
#034
#035     KdPrint(("DriverEntry end\n"));
#036     return status;
#037   }
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 代码 17 行，用#pragma 指明此函数是加载到 INIT 内存区域中，即成功卸载后，可以退出内存。
- 代码 18 行，标志 DriverEntry 函数的开始。注意此处在函数体的前面用 extern "C"修饰，这样在编译的时候会编译成_DriverEntry@8 的符号。如果不加入此修饰符号，编译器会自动按照 C++的符号名编译，导致错误链接。
- 代码 23 行，打印一行调试信息。KdPrint 其实是一个宏，在调试版本（Checked 版）中，会用 DbgPrint 代替。而在发行版（Free 版）中，则不执行任何操作，其功能类似于 MFC 中的 TRACE 宏。由于驱动程序是运行在 Windows 的核心态，没有用户界面，所以查看调试信息有别于 Win32 程序。关于查看调试信息的讲解将在第 3 章论述。
- 代码 26~30 行，驱动程序向 Windows 的 I/O 管理器注册一些回调函数。回调函数是由程序员定义的函数，这些函数不是由驱动程序本身负责调用，而是由操作系统负责调用。程序员将这些函数的入口地址告诉操作系统，操作系统会在适当的时候调用这些函数。在这个例子中，这几个回调函数基本是自解释型的，读者可以根据函数名分析出其作用。当驱动被卸载时，调用 HelloDDKUnload。当驱动程序处理创建、关闭和读写相关的 IRP 时，调用 HelloDDKDispatchRoutine（这里只是将处理函数简化为一个函数，实际情况要比这个复杂）。
- 代码第 33 行，调用 CreateDevice 函数，此函数的解释见下一节。
- 代码第 36 行，返回 CreateDevice 的执行结果。如果执行正确，驱动将被成功加载。

1.2.3 创建设备例程

CreateDevice 函数是一个帮助函数（Helper Function），辅助 DriverEntry 创建一个设备对象。其完全可以展开放在 DriverEntry 中，但为了代码的条理性，笔者将其构造成一个辅

助函数。

```
#001  /************************************************************************
#002  * 函数名称:CreateDevice
#003  * 功能描述:初始化设备对象
#004  * 参数列表:
#005      pDriverObject:从 I/O 管理器中传进来的驱动对象
#006  * 返回 值:返回初始化状态
#007  ************************************************************************/
#008  #pragma INITCODE
#009  NTSTATUS CreateDevice (
#010        IN PDRIVER_OBJECT pDriverObject)
#011  {
#012    NTSTATUS status;
#013    PDEVICE_OBJECT pDevObj;
#014    PDEVICE EXTENSION pDevExt;
#015
#016    //创建设备名称
#017    UNICODE_STRING devName;
#018    RtlInitUnicodeString(&devName,L"\\Device\\MyDDKDevice");
#019
#020    //创建设备
#021    status = IoCreateDevice( pDriverObject,
#022                          sizeof(DEVICE_EXTENSION),
#023                          &(UNICODE STRING)devName,
#024                          FILE_DEVICE_UNKNOWN,
#025                          0, TRUE,
#026                          &pDevObj );
#027    if (!NT_SUCCESS(status))
#028        return status;
#029
#030    pDevObj->Flags |= DO_BUFFERED_IO;
#031    pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
#032    pDevExt->pDevice = pDevObj;
#033    pDevExt->ustrDeviceName = devName;
#034    //创建符号链接
#035    UNICODE_STRING symLinkName;
#036    RtlInitUnicodeString(&symLinkName,L"\\??\\HelloDDK");
#037    pDevExt->ustrSymLinkName = symLinkName;
#038    status = IoCreateSymbolicLink( &symLinkName,&devName );
#039    if (!NT_SUCCESS(status))
#040    {
#041        IoDeleteDevice( pDevObj );
#042        return status;
#043    }
#044    return STATUS_SUCCESS;
#045  }
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 代码 16~18 行，构造一个 Unicode 字符串，此字符串用来存储此设备对象的名称。Unicode 字符串大量运用在驱动程序开发中，有关 Unicode 的讲解请参考第 3 章。
- 代码 21~28 行，用 IoCreateDevice 函数创建一个设备对象。其对象名称来自于上一步构造的 Unicode 字符串，设备类型为 FILE_DEVICE_UNKNOWN，且此种设备

为独占设备，即设备只能被一个应用程序所使用。

- 代码第 30 行，表明此种设备为 BUFFERED_IO 设备。设备对内存的操作分为两种，BUFFERED_IO 和 DO_DIRECT_IO，此部分讲解请参考第 3 章。
- 代码 31~33 行，填写设备的扩展结构体，在其他驱动程序的函数中，可以很方便地得到这个结构体，进而得到该设备的自定义信息。此结构体的定义在 Driver.h 中。
- 代码 34~38 行，创建符号链接。驱动程序虽然有了设备名称，但是这种设备名称只能在内核态可见，而对于应用程序是不可见的。因此，驱动需要暴露一个符号链接，该链接指向真正的设备名称。
- 代码 39~44 行，当设备创建成功后返回。如果不成功，则删除该设备。

1.2.4 卸载驱动例程

卸载驱动例程用来设备被卸载的情况，由 I/O 管理器负责调用此回调函数。此例程遍历系统中所有的此类设备对象。第一个设备对象的地址存在于驱动对象的 DeviceObject 域中，每个设备对象的 NextDevice 域记录着下一个设备对象的地址，这样就形成一个链表。卸载驱动例程的主要目的就是遍历系统中所有的此类设备对象，然后删除设备对象以及符号链接。

```
#001  /*************************************************************************
#002  * 函数名称:HelloDDKUnload
#003  * 功能描述:负责驱动程序的卸载操作
#004  * 参数列表:
#005      pDriverObject:驱动对象
#006  * 返回值:返回状态
#007  *************************************************************************/
#008  #pragma PAGEDCODE
#009  VOID HelloDDKUnload (IN PDRIVER_OBJECT pDriverObject)
#010  {
#011    PDEVICE_OBJECT    pNextObj;
#012    KdPrint(("Enter DriverUnload\n"));
#013    pNextObj = pDriverObject->DeviceObject;
#014    while (pNextObj != NULL)
#015    {
#016        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)
#017            pNextObj->DeviceExtension;
#018
#019        //删除符号链接
#020        UNICODE_STRING pLinkName = pDevExt->ustrSymLinkName;
#021        IoDeleteSymbolicLink(&pLinkName);
#022        pNextObj = pNextObj->NextDevice;
#023        IoDeleteDevice( pDevExt->pDevice );
#024    }
#025  }
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 代码第 13 行，由驱动对象得到设备对象。
- 代码 19~21 行，删除设备对象的符号链接。
- 代码 22~23 行，遍历设备对象，并删除。

1.2.5　默认派遣例程

对设备对象的创建、关闭和读写操作，都被指定到这个默认的派遣例程中。由于这是一个最简单的演示程序，故只是简单地将其成功返回。后面的章节中，笔者将会扩充该例程。

```
#001  /*************************************************************************
#002  * 函数名称:HelloDDKDispatchRoutine
#003  * 功能描述:对读 IRP 进行处理
#004  * 参数列表:
#005        pDevObj:功能设备对象
#006        pIrp:从 I/O 请求包
#007  * 返回值:返回状态
#008  *************************************************************************/
#009  #pragma PAGEDCODE
#010  NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE_OBJECT pDevObj,
#011                                   IN PIRP pIrp)
#012  {
#013    KdPrint(("Enter HelloDDKDispatchRoutine\n"));
#014    NTSTATUS status = STATUS_SUCCESS;
#015    // 完成 IRP
#016    pIrp->IoStatus.Status = status;
#017    pIrp->IoStatus.Information = 0;    // bytes xfered
#018    IoCompleteRequest( pIrp, IO_NO_INCREMENT );
#019    KdPrint(("Leave HelloDDKDispatchRoutine\n"));
#020    return status;
#021  }
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 代码 16 行，设置 IRP 的状态为成功。IRP 的讲解请参考第 4 章。
- 代码 17 行，设置操作的字节数为 0，这里无实际意义。
- 代码 18 行，指示完成此 IRP。
- 代码 20 行，成功返回。

1.3　HelloDDK 的编译和安装

本节会带领读者一步步地对 HelloDDK 进行编译和安装。编译和安装往往是初学者最先需要面对的问题，笔者将会从两个方面讲解编译过程。一是传统的用 DDK 编译环境编译，二是用 Visual C++（以下简称 VC）集成开发环境编译。

1.3.1 用 DDK 环境编译 HelloDDK

这种编译驱动的办法是 DDK 文档中所提倡的办法。此种方法需要编写一个编译脚本文件，在这个脚本中描述了 DDK 驱动程序的源文件、用到的 lib 文件和 inlcude 路径名、编译输出的目录和文件名等信息，具体介绍请参考第 3 章。编写此类脚本对于 Windows 程序员可能比较陌生，尤其是当源文件较多时，编写脚本文件可能显得更如麻烦。下一节将向读者介绍一种简单的用 Visual C++ 6.0 IDE（以下简称 VC IDE）环境编译驱动的方法。在源程序的相同目录下创建两个文件 makefile 和 Sources，这两个文件都是文本文件，内容如下。

Sources：

```
#001   TARGETNAME=HelloDDK
#002   TARGETTYPE=DRIVER
#003   TARGETPATH=OBJ
#004
#005   INCLUDES=$(BASEDIR)\inc;\
#006           $(BASEDIR)\inc\ddk;\
#007
#008   SOURCES=Driver.cpp\
```

此段代码可以在配套光盘中本章的 NT_Driver 目录下找到。

- 第 1 行说明此驱动的名称。
- 第 2 行指明此驱动的类型为 NT 型驱动。
- 第 3 行设置编译输出目录。
- 第 5~6 行设置 include 目录。
- 第 8 行指定源文件。

编写完这两个脚本后，在 Windows 的开始菜单中选择“Windows XP Checked Build Environment”编译环境。这里选择的是 Checked 版本，而不是 Free 版本。两者的区别类似于 Win32 程序开发的 Debug 版本和 Release 版本，具体的差别详见第 3 章。

选择版本后，进入的是一个命令行方式的窗口。用 cd 命令进入需要编译的目录，然后输入“build”DDK 的编译环境会自动调用编译器进行编译，笔者计算机中的编译结果如图 1-4 所示。

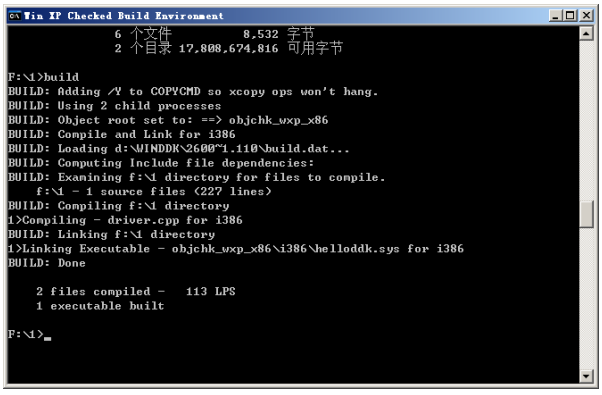


图 1-4　DDK 的编译环境

编译好的结果位于源码目录下的子目录 objchk_wxp_x86\i386（如果读者是用 Windows 2000 DDK 编译的，目录会稍有不同）中。编译出来的二进制文件为 HelloDDK.sys，它不像 exe 文件那样运行，而是必须通过特殊的加载方式加载，详见 1.3.3 节。

1.3.2　用 VC 集成开发环境编译 HelloDDK

初次学习编写 Windows 驱动程序的开发人员，大部分是熟悉 VC IDE 开发环境的 Windows 程序员。他们可能不喜欢用编辑脚本来描述一个工程，而是更希望在熟悉的 VC IDE 环境下编译，并且利用 VC IDE 可以方便快速地对代码进行交叉索引等操作。本节将向读者介绍此种方法。

（1）用 VC 建立一个新工程。在 VC IDE 环境中选择“File”|“New”，弹出“New”对话框。在该对话框中，选择“Project”选项卡。在“Project”选项卡中，选择 Win32 Application（因为 VC 并没有提供驱动程序的工程，所以在 Win32 工程的基础上进行修改）。工程名为“DriverDev”，如图 1-5 所示。单击“OK”按钮，进入下一个对话框。在该对话框中，选择一个空的工程，如图 1-6 所示。

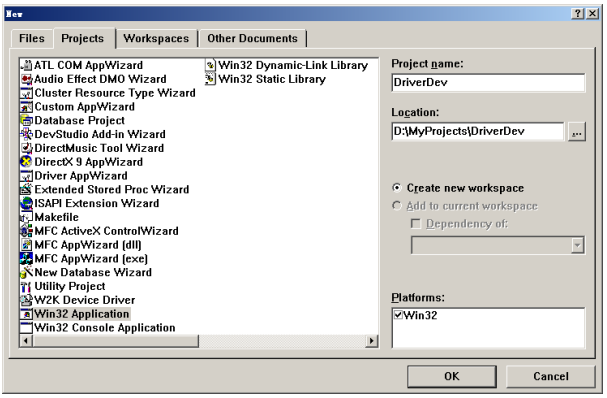


图 1-5　添加新工程

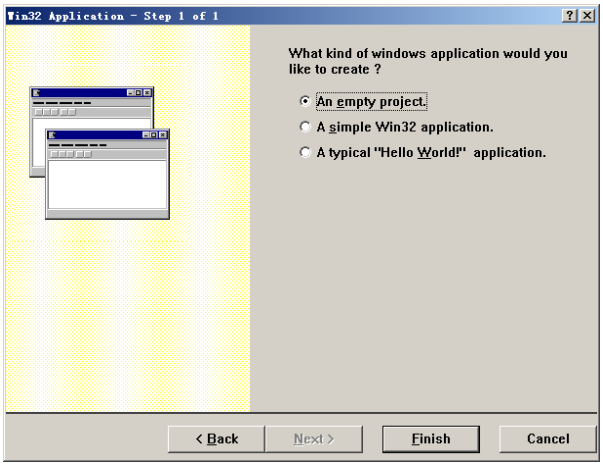


图 1-6　创建新工程

（2）将两个源文件 Driver.h 和 Driver.cpp 拷贝到工程目录中，并添加到工程中，如图 1-7 所示。

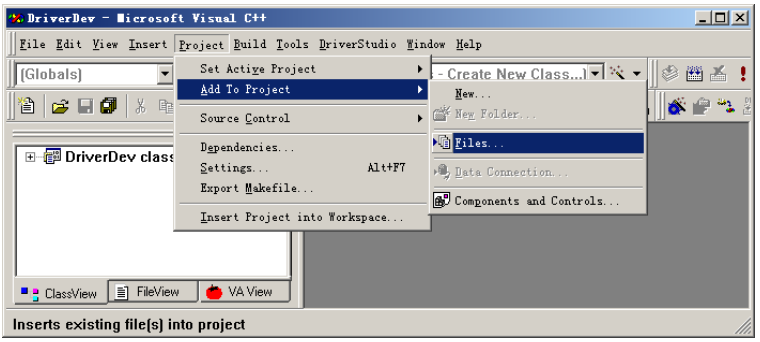


图 1-7　添加新文件到工程

（3）增加新的编译版本，去掉 Debug 和 Release 版本，如图 1-8 和图 1-9 所示。

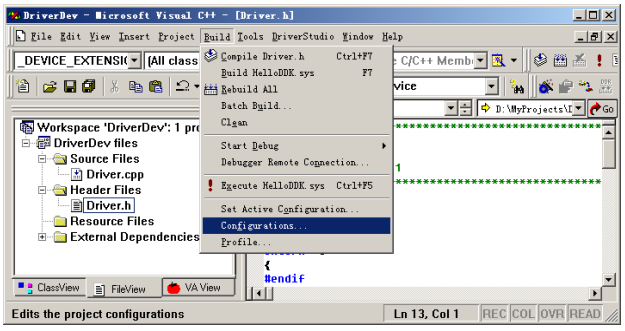


图 1-8　配置编译版本

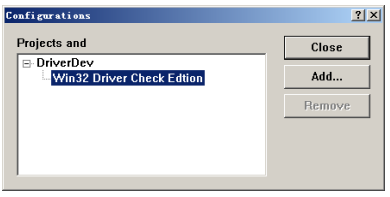


图 1-9　修改后的 check 版本

（4）修改工程属性。选择“Project”|“Setting”，或者直接按下 Alt+F7 键，弹出“Project Settings”对话框。在对话框中，选择“General”选项卡。将 Intermediate files 和 Output files 改为 MyDriver_Check，如图 1-10 所示。

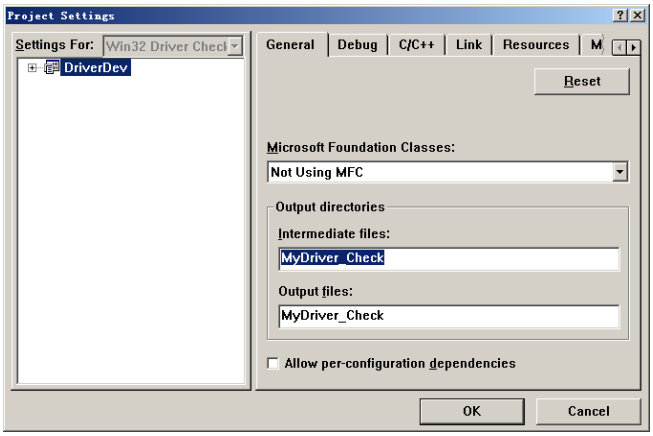


图 1-10　修改输出目录

选择 C/C++选项卡，将原有的 Project Options 内容全部删除，替换成如下内容，如图 1-11 所示。

```
    /nologo /Gz /MLd /W3 /WX /Z7 /Od /D WIN32=100 /D _X86_=1 /D WINVER=0x500 /D DBG=1
/Fo"MyDriver_Check/" /Fd"MyDriver_Check/" /FD /c
```

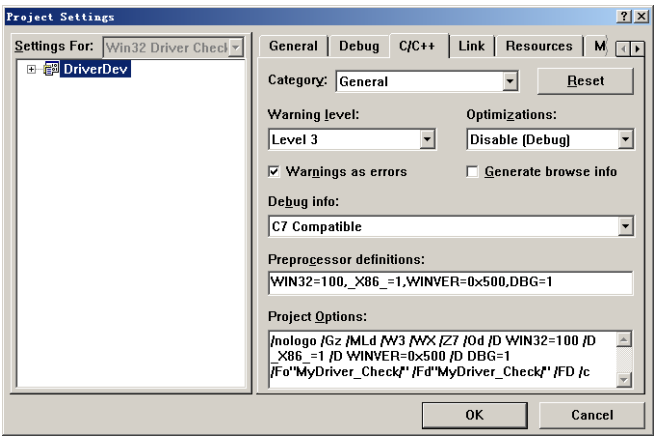


图 1-11　修改 C++选项卡

选择 Link 选项卡，将原有的 Project Options 内容全部删除，替换成如下内容，如图 1-12 所示。

```
    ntoskrnl.lib /nologo /base:"0x10000" /stack:0x400000,0x1000 /entry:"DriverEntry"
/subsystem:console /incremental:no /pdb:"MyDriver_Check/HelloDDK.pdb" /debug
/machine:I386 /nodefaultlib /out:"MyDriver_Check/HelloDDK.sys" /pdbtype:sept
/subsystem:native /driver /SECTION:INIT,D /RELEASE /IGNORE:4078
```

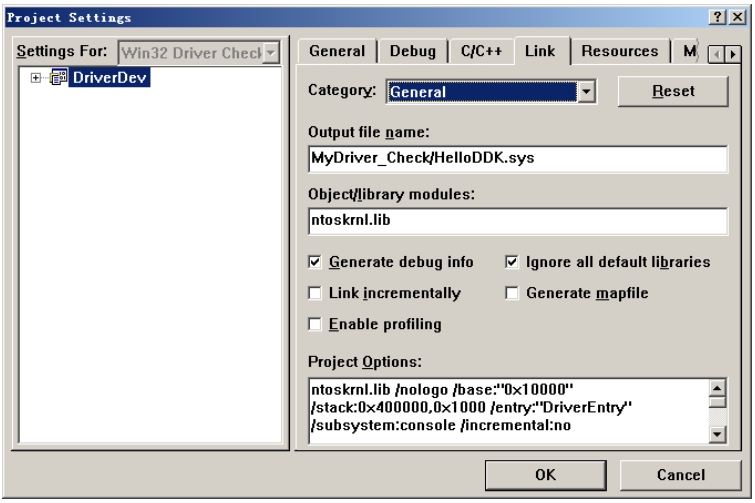
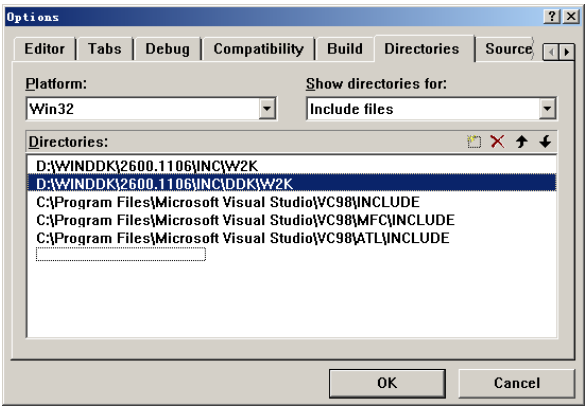
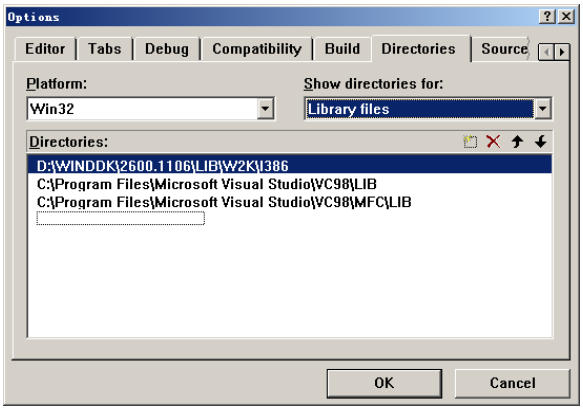


图 1-12　修改 link 选项卡

（5）修改 VC 的 lib 目录和 include 目录。在 VC 中选择“Tools”|“Options”，在弹出的对话框中选择“Directories”选项卡。在“Show directories for”下拉菜单中选择“Include files”菜单。添加“D:\WINDDK\2600.1106\INC\W2K”和“D:\WINDDK\2600.1106\INC\DDK\W2K”，并将这两个目录置于最上，如图 1-13（a）所示。读者可将“D:\WINDDK\ 2600.1106”替换成自己的 DDK 安装目录。这里应该选择 W2K 子目录，DDK 中还会有相应的 XP 子目录。因为 XP 驱动编译时候需要高版本的 VC 编译器，所以这里用的是 W2K 子目录，它编译的代码完全可以应用于 Windows 2000 和 Windows XP 操作系统下。



（a）



（b）

图 1-13　设置 include 目录和设置 lib 目录

在“Show directories for”下拉菜单中选择“Library files”菜单，添加目录“D:\WINDDK\2600.1106\LIB\W2K\I386”，并置于最上端，如图 1-13（b）所示。

（6）编译。按下 F7 键，和 1.3.2 节一样，同样会编译出一个 HelloDDK.sys 文件。

1.3.3　HelloDDK 的安装

NT 式驱动程序类似于 Windows 服务程序，以服务的方式加载在系统中。在第 3 章中，笔者将给出加载驱动的代码。为了简化步骤，这里利用一个叫做 DriverMonitor 的工具软件加载 HelloDDK。DriverMonitor 是 Compuware 公司开发的 DriverStudio 中的一个工具，笔者将在第 4 章对 DriverStudio 提供的一系列工具软件逐一介绍。笔者强烈推荐读者安装 DriverStudio，因为它提供的一系列工具对调试驱动非常有用。如果读者没有安装 DriverMonitor，请参阅第 3 章，笔者将介绍如何编写一个 NT 式驱动程序的加载器。

运行 DriverMonitor，选择“File”|“Open Driver”，将会弹出文件选择对话框，选择编译好的 HelloDDK.sys 文件。再次选择“File”|“Start Driver”。至此，NT 驱动加载成功，DriverMonitor 会报告加载情况，如图 1-14 所示。

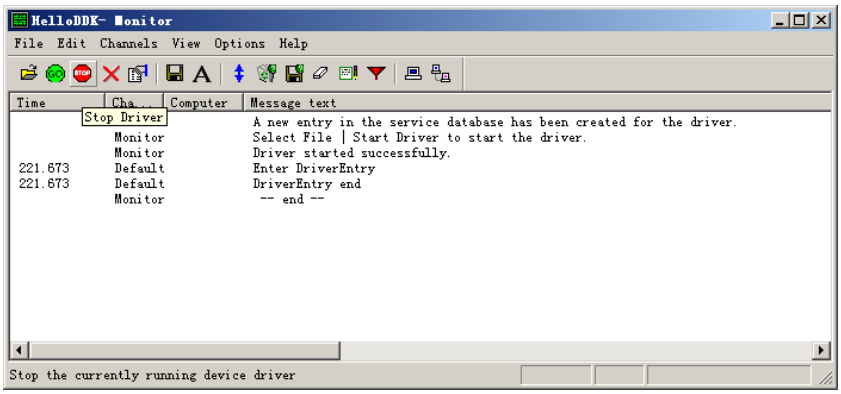


图 1-14　用 Driver Monitor 安装 HelloDDK

用 Driver Monitor 加载驱动时，默认是加载一次。重新启动电脑后，该驱动不会被加载。如果想在每次开机启动时自动加载，需要修改设置。选择“Edit”|“Properties”，弹出如图 1-15 所示的对话框。在“Start Type”选择组中，选择“Automatic”单选按钮。保存后，就可以在每次开机启动时自动加载该驱动。

成功加载的驱动，会出现在 Windows 的设备管理器中。默认情况下，NT 式的驱动程序是隐藏的，在设备管理器中选择“查看”|“显示隐藏设备”，如图 1-16 所示。

在“Not-Plug and Play Drivers”的列表中，会出现 HelloDDK 的驱动程序，如图 1-17 所示。

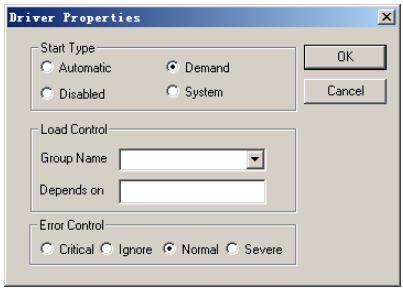


图 1-16　选择加载方式

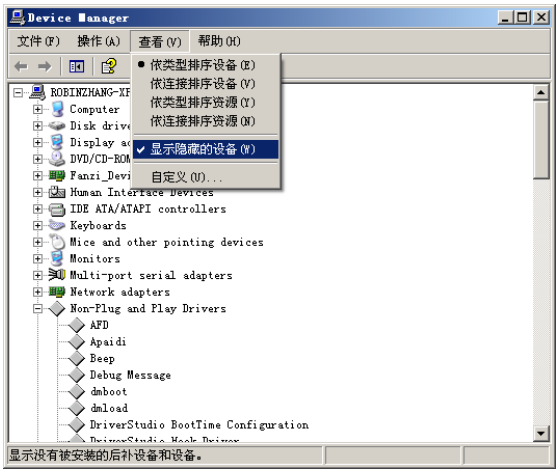


图 1-17　在设备管理器中显示 HelloDDK 设备

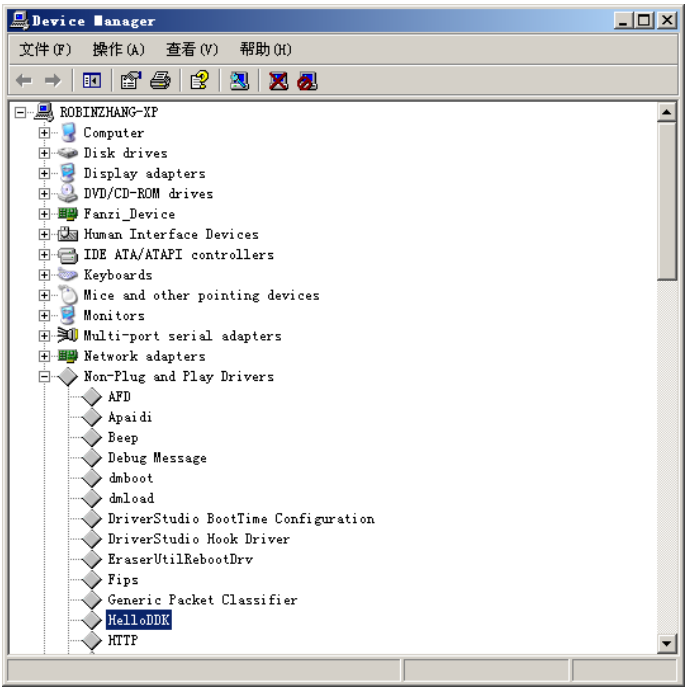


图 1-17　在设备管理器中显示设备

1.4　第二个驱动程序 HelloWDM 的代码分析

本节编写的 HelloWDM 是基于 WDM 的驱动程序，和前面介绍的 HelloDDK 非常相似，且多了对即插即用功能的支持。NT 式驱动和 WDM 驱动的异同将在第 3 章介绍。

1.4.1　HelloWDM 的头文件

HelloWDM 的头文件主要是为了导入驱动程序开发所必需的 WDM.h 头文件，此头文件里包含了对 DDK 所有导出函数的声明。NT 式的驱动程序要导入的是 NTDDK.h，而 WDM 驱动要导入的头文件为 WDM.h。另外，此头文件中定义了几个标签，分别在程序中指明函数和变量分配在分页内存中或非分页内存中。最后，该头文件给出了此驱动程序的函数声明。

```
#001  /*************************************************************************
#002  * 文件名称:HelloWDM.h
#003  * 作　　者:张帆
#004  * 完成日期:2007-11-1
#005  *************************************************************************/
#006
#007  #ifdef __cplusplus
#008
#009  extern "C"
```

```
#010  {
#011  #endif
#012  #include <wdm.h>
#013  #ifdef   cplusplus
#014  }
#015  #endif
#016
#017  typedef struct _DEVICE_EXTENSION
#018  {
#019      PDEVICE OBJECT fdo;
#020      PDEVICE_OBJECT NextStackDevice;
#021    UNICODE_STRING ustrDeviceName; // 设备名
#022    UNICODE_STRING ustrSymLinkName;     // 符号链接名
#023  } DEVICE EXTENSION, *PDEVICE EXTENSION;
#024
#025  #define PAGEDCODE code_seg("PAGE")
#026  #define LOCKEDCODE code seg()
#027  #define INITCODE code_seg("INIT")
#028
#029  #define PAGEDDATA data seg("PAGE")
#030  #define LOCKEDDATA data_seg()
#031  #define INITDATA data_seg("INIT")
#032
#033  #define arraysize(p) (sizeof(p)/sizeof((p)[0]))
#034
#035  NTSTATUS HelloWDMAddDevice(IN PDRIVER OBJECT DriverObject,
#036                             IN PDEVICE_OBJECT PhysicalDeviceObject);
#037  NTSTATUS HelloWDMPnp(IN PDEVICE_OBJECT fdo,
#038                       IN PIRP Irp);
#039  NTSTATUS HelloWDMDispatchRoutine(IN PDEVICE_OBJECT fdo,
#040                                   IN PIRP Irp);
#041  void HelloWDMUnload(IN PDRIVER OBJECT DriverObject);
#042
#043  extern "C"
#044  NTSTATUS DriverEntry(IN PDRIVER OBJECT DriverObject,
#045                       IN PUNICODE_STRING RegistryPath);
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

- 代码 7~15 行，包含头文件 wdm.h。和 HelloDDK 的头文件类似，在引用时也用 extern "C"进行修饰，为的是保持符号链接的正确性。注意，此处包含的是 wdm.h 而非 ntddk.h，这是 WDM 驱动程序和 NT 驱动程序最主要的区别。关于 WDM 驱动程序和 NT 驱动程序的不同，请参考第 3 章。
- 代码 17~23 行，定义了设备扩展结构体，此结构体为本设备记录相关信息。
- 代码 25~31 行，定义了分页内存、非分页内存和 INIT 段内存的标志，以便在下面的程序中声明。
- 代码 35~45 行为代码声明。

1.4.2 HelloWDM 的入口函数

和 NT 式驱动程序一样，WDM 的入口函数地址同样是 DriverEntry，且在 C++编译的

时候需要用 extern "C"修饰。

```
#001  /*************************************************************************
#002  * 函数名称:DriverEntry
#003  * 功能描述:初始化驱动程序，定位和申请硬件资源，创建内核对象
#004  * 参数列表:
#005      pDriverObject:从 I/O 管理器中传进来的驱动对象
#006      pRegistryPath:驱动程序在注册表的中的路径
#007  * 返回值:返回初始化驱动状态
#008  *************************************************************************/
#009  #pragma INITCODE
#010  extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
#011                                  IN PUNICODE_STRING pRegistryPath)
#012  {
#013    KdPrint(("Enter DriverEntry\n"));
#014
#015    pDriverObject->DriverExtension->AddDevice = HelloWDMAddDevice;
#016    pDriverObject->MajorFunction[IRP_MJ_PNP] = HelloWDMPnp;
#017    pDriverObject->MajorFunction[IRP MJ DEVICE CONTROL] =
#018    pDriverObject->MajorFunction[IRP_MJ_CREATE] =
#019    pDriverObject->MajorFunction[IRP_MJ_READ] =
#020    pDriverObject->MajorFunction[IRP MJ WRITE] = HelloWDMDispatchRoutine;
#021    pDriverObject->DriverUnload = HelloWDMUnload;
#022
#023    KdPrint(("Leave DriverEntry\n"));
#024    return STATUS_SUCCESS;
#025  }
```

- 代码第 9 行，将此函数放在 INIT 段中，当驱动加载结束后，此函数就可以从内存中卸载掉。
- 代码 10~11 行，定义驱动入口函数，用 extern "C"修饰。其中传进来两个参数，第一个参数 pDriverObject 为驱动对象，第二个参数 pRegistryPath 为此驱动的注册表路径。
- 代码第 15 行，设置 AddDevice 回调函数，此回调函数只出现在 WDM 驱动程序中，而在 NT 式的驱动中没有此回调函数。此回调函数的作用是创建设备对象并由 PNP（即插即用）管理器调用。
- 代码 16 行，设置对 IRP_MJ_PNP 的 IRP 的回调函数。对 PNP 的 IRP 处理，是 NT 式驱动和 WDM 驱动的重大区别之一。
- 代码 17~20 行，设置常用 IRP 的回调函数。这里只是简单地指向了同一个默认函数 HelloWDMDispatchRoutine。
- 代码 21 行，向系统注册卸载例程。在 WDM 程序中，大部分卸载工作已不在此处理，而是放在对 IRP_MN_REMOVE_DEVICE 的 IRP 的处理函数中处理。

1.4.3　HelloWDM 的 AddDevice 例程

在 WDM 的驱动程序中，创建设备对象的任务不再由 DriverEntry 承担，而需要驱动

程序向系统注册一个称做 AddDevice 的例程。此例程由 PNP 管理器负责调用，其函数主要职责是创建设备对象。HelloWDMAddDevice 例程有两个参数，DriverObject 和 PhysicalDeviceObject。DriverObject 是由 PNP 管理器传递进来的驱动对象，此对象是 DriverEntry 中的驱动对象。PhysicalDeviceObject 是 PNP 管理器传递进来的底层驱动设备对象，这个概念在 NT 式的驱动中是没有的。关于这个概念，请参考第 3 章。

```
#001  /*************************************************************************
#002  * 函数名称:HelloWDMAddDevice
#003  * 功能描述:添加新设备
#004  * 参数列表:
#005       DriverObject:从 I/O 管理器中传进来的驱动对象
#006       PhysicalDeviceObject:从 I/O 管理器中传进来的物理设备对象
#007  * 返回值:返回添加新设备状态
#008  *************************************************************************/
#009  #pragma PAGEDCODE
#010  NTSTATUS HelloWDMAddDevice(IN PDRIVER_OBJECT DriverObject,
#011                          IN PDEVICE_OBJECT PhysicalDeviceObject)
#012  {
#013    PAGED_CODE();
#014    KdPrint(("Enter HelloWDMAddDevice\n"));
#015
#016    NTSTATUS status;
#017    PDEVICE_OBJECT fdo;
#018    UNICODE_STRING devName;
#019    RtlInitUnicodeString(&devName,L"\\Device\\MyWDMDevice");
#020    status = IoCreateDevice(
#021         DriverObject,
#022         sizeof(DEVICE EXTENSION),
#023         &(UNICODE_STRING)devName,
#024         FILE_DEVICE_UNKNOWN,
#025         0,
#026         FALSE,
#027         &fdo);
#028    if( !NT SUCCESS(status))
#029         return status;
#030    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION)fdo->DeviceExtension;
#031    pdx->fdo = fdo;
#032    pdx->NextStackDevice = IoAttachDeviceToDeviceStack(fdo, Physical
DeviceObject);
#033    UNICODE STRING symLinkName;
#034    RtlInitUnicodeString(&symLinkName,L"\\DosDevices\\HelloWDM");
#035
#036    pdx->ustrDeviceName = devName;
#037    pdx->ustrSymLinkName = symLinkName;
#038    status = IoCreateSymbolicLink(&(UNICODE_STRING)symLinkName,&(UNICODE_
STRING)devName);
#039
#040    if( !NT_SUCCESS(status))
#041    {
#042         IoDeleteSymbolicLink(&pdx->ustrSymLinkName);
#043         status = IoCreateSymbolicLink(&symLinkName,&devName);
#044         if( !NT SUCCESS(status))
#045         {
#046              return status;
```

```
#047          }
#048      }
#049
#050      fdo->Flags |= DO_BUFFERED_IO | DO_POWER_PAGABLE;
#051      fdo->Flags &= ~DO_DEVICE_INITIALIZING;
#052
#053      KdPrint(("Leave HelloWDMAddDevice\n"));
#054      return STATUS_SUCCESS;
#055  }
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录中找到。

- 代码第 9 行，用#pragma 指明将此例程分派在分页内存中。
- 代码第 13 行，PAGED_CODE 是一个 DDK 提供的宏，只在 check 版中有效。当此例程所在的中断请求级超过 APC_LEVEL 时，会产生一个断言，断言会使程序终止，并报告出错地址。有关中断请求级的讲解，请参考第 3 章。
- 代码 16~27 行，创建设备对象，此处和 HelloDDK 的创建方法一样。
- 代码 30 行，得到设备对象扩展数据结构。
- 代码 31 行，记录设备扩展中的功能设备对象为其刚才创建的设备。
- 代码 32 行，用 IoAttachDeviceToDeviceStack 函数将此 fdo（功能设备对象）挂接在设备堆栈上，并将返回值（下层堆栈的位置），记录在设备扩展结构中。
- 代码 37~38 行，创建设备的符号链接，此处和 HelloDDK 方法一样。
- 代码 50~51 行，设置设备为 BUFFERED_IO 设备，并指明驱动初始化完成。
- 代码 54 行，成功返回。

1.4.4　HelloWDM 处理 PNP 的回调函数

WDM 式驱动程序主要区别在于对 IRP_MJ_PNP 的 IRP 的处理。其中，IRP_MJ_PNP 会细分为若干个子类。例如，IRP_MN_START_DEVICE、IRP_MN_REMOVE_DEVICE、IRP_MN_STOP_DEVICE 等。本例中除了对 IRP_MN_REMOVE_DEVICE 做特殊处理，其他 IRP 则做相同处理。

```
#001  /*************************************************************************
#002  * 函数名称:HelloWDMPnp
#003  * 功能描述:对即插即用 IRP 进行处理
#004  * 参数列表:
#005        fdo:功能设备对象
#006        Irp:从 I/O 请求包
#007  * 返回值:返回状态
#008  *************************************************************************/
#009  #pragma PAGEDCODE
#010  NTSTATUS HelloWDMPnp(IN PDEVICE_OBJECT fdo,
#011                       IN PIRP Irp)
#012  {
#013    PAGED_CODE();
#014
```

```
#015     KdPrint(("Enter HelloWDMPnp\n"));
#016     NTSTATUS status = STATUS_SUCCESS;
#017     PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
#018     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
#019     static NTSTATUS (*fcntab[])(PDEVICE_EXTENSION pdx, PIRP Irp) =
#020     {
#021         DefaultPnpHandler,          // IRP_MN_START_DEVICE
#022         DefaultPnpHandler,          // IRP_MN_QUERY_REMOVE_DEVICE
#023         HandleRemoveDevice,         // IRP_MN_REMOVE_DEVICE
#024         DefaultPnpHandler,          // IRP_MN_CANCEL_REMOVE_DEVICE
#025         DefaultPnpHandler,          // IRP_MN_STOP_DEVICE
#026         DefaultPnpHandler,          // IRP_MN_QUERY_STOP_DEVICE
#027         DefaultPnpHandler,          // IRP_MN_CANCEL_STOP_DEVICE
#028         DefaultPnpHandler,          // IRP_MN_QUERY_DEVICE_RELATIONS
#029         DefaultPnpHandler,          // IRP_MN_QUERY_INTERFACE
#030         DefaultPnpHandler,          // IRP_MN_QUERY_CAPABILITIES
#031         DefaultPnpHandler,          // IRP_MN_QUERY_RESOURCES
#032         DefaultPnpHandler,          // IRP_MN_QUERY_RESOURCE_REQUIREMENTS
#033         DefaultPnpHandler,          // IRP_MN_QUERY_DEVICE_TEXT
#034         DefaultPnpHandler,          // IRP_MN_FILTER_RESOURCE_REQUIREMENTS
#035         DefaultPnpHandler,          //
#036         DefaultPnpHandler,          // IRP_MN_READ_CONFIG
#037         DefaultPnpHandler,          // IRP_MN_WRITE_CONFIG
#038         DefaultPnpHandler,          // IRP_MN_EJECT
#039         DefaultPnpHandler,          // IRP_MN_SET_LOCK
#040         DefaultPnpHandler,          // IRP_MN_QUERY_ID
#041         DefaultPnpHandler,          // IRP_MN_QUERY_PNP_DEVICE_STATE
#042         DefaultPnpHandler,          // IRP_MN_QUERY_BUS_INFORMATION
#043         DefaultPnpHandler,          // IRP_MN_DEVICE_USAGE_NOTIFICATION
#044         DefaultPnpHandler,          // IRP_MN_SURPRISE_REMOVAL
#045     };
#046
#047     ULONG fcn = stack->MinorFunction;
#048     if (fcn >= arraysize(fcntab))
#049     {                               //未知的 IRP 类别
#050         status = DefaultPnpHandler(pdx, Irp); // 对于未知的 IRP 类别，我们让
                                                  // DefaultPnpHandler 函数处理
#051         return status;
#052     }                               //未知的 IRP 类别
#053
#054   #if DBG
#055     static char* fcnname[] =
#056     {
#057         "IRP_MN_START_DEVICE",
#058         "IRP_MN_QUERY_REMOVE_DEVICE",
#059         "IRP_MN_REMOVE_DEVICE",
#060         "IRP_MN_CANCEL_REMOVE_DEVICE",
#061         "IRP_MN_STOP_DEVICE",
#062         "IRP_MN_QUERY_STOP_DEVICE",
#063         "IRP_MN_CANCEL_STOP_DEVICE",
#064         "IRP_MN_QUERY_DEVICE_RELATIONS",
#065         "IRP_MN_QUERY_INTERFACE",
#066         "IRP_MN_QUERY_CAPABILITIES",
#067         "IRP_MN_QUERY_RESOURCES",
#068         "IRP_MN_QUERY_RESOURCE_REQUIREMENTS",
#069         "IRP_MN_QUERY_DEVICE_TEXT",
#070         "IRP_MN_FILTER_RESOURCE_REQUIREMENTS",
#071         "",
```

```
#072          "IRP_MN_READ_CONFIG",
#073          "IRP_MN_WRITE_CONFIG",
#074          "IRP_MN_EJECT",
#075          "IRP_MN_SET_LOCK",
#076          "IRP_MN_QUERY_ID",
#077          "IRP_MN_QUERY_PNP_DEVICE_STATE",
#078          "IRP_MN_QUERY_BUS_INFORMATION",
#079          "IRP_MN_DEVICE_USAGE_NOTIFICATION",
#080          "IRP_MN_SURPRISE_REMOVAL",
#081      };
#082
#083      KdPrint(("PNP Request (%s)\n", fcnname[fcn]));
#084  #endif // DBG
#085
#086      status = (*fcntab[fcn])(pdx, Irp);
#087      KdPrint(("Leave HelloWDMPnp\n"));
#088      return status;
#089  }
```

- 代码 13 行，用 PAGED_CODE 宏确保该例程运行在低于 APC_LEVEL 的中断优先级的级别上。
- 代码 17 行，得到设备扩展结构。
- 代码 18 行，得到当前 IRP 的堆栈。设备堆栈是一个很复杂的概念，笔者将在第 4 章进行论述。
- 代码 19~52 行，将对应类别的即插即用 IRP 调用做不同的处理，并打印出调试信息。其中，IRP_MN_REMOVE_DEVICE 由 HandleRemoveDevice 处理，而其他 IRP 则由 DefaultPnpHandler 处理。

1.4.5　HelloWDM 对 PNP 的默认处理

除了 IRP_MN_STOP_DEVICE 以外，HelloWDM 对其他 PNP 的 IRP 做同样的处理，即直接传递到底层驱动，并将底层驱动的结果返回。

```
#001  /*************************************************************************
#002  * 函数名称:DefaultPnpHandler
#003  * 功能描述:对 PNP IRP 进行默认处理
#004  * 参数列表:
#005      pdx:设备对象的扩展
#006      Irp:从 I/O 请求包
#007  * 返回值:返回状态
#008  *************************************************************************/
#009  #pragma PAGEDCODE
#010  NTSTATUS DefaultPnpHandler(PDEVICE_EXTENSION pdx, PIRP Irp)
#011  {
#012    PAGED_CODE();
#013    KdPrint(("Enter DefaultPnpHandler\n"));
#014    IoSkipCurrentIrpStackLocation(Irp);
#015    KdPrint(("Leave DefaultPnpHandler\n"));
#016    return IoCallDriver(pdx->NextStackDevice, Irp);
```

```
#017   }
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

- 代码 12 行，确保该例程处于 APC_LEVEL 之下。
- 代码 14 行，略过当前堆栈。
- 代码 16 行，用下层堆栈的驱动设备对象处理此 IRP。

1.4.6　HelloWDM 对 IRP_MN_REMOVE_DEVICE 的处理

对 IRP_MN_REMOVE_DEVICE 的处理类似于在 NT 式驱动中的卸载例程，而在 WDM 式的驱动中，卸载例程几乎不用做处理。

```
#001  /*************************************************************************
#002  * 函数名称:HandleRemoveDevice
#003  * 功能描述:对 IRP_MN_REMOVE_DEVICE IRP 进行处理
#004  * 参数列表:
#005        fdo:功能设备对象
#006        Irp:从 I/O 请求包
#007  * 返回值:返回状态
#008  *************************************************************************/
#009  #pragma PAGEDCODE
#010  NTSTATUS HandleRemoveDevice(PDEVICE EXTENSION pdx, PIRP Irp)
#011  {
#012    PAGED_CODE();
#013    KdPrint(("Enter HandleRemoveDevice\n"));
#014
#015    Irp->IoStatus.Status = STATUS SUCCESS;
#016    NTSTATUS status = DefaultPnpHandler(pdx, Irp);
#017    IoDeleteSymbolicLink(&(UNICODE_STRING)pdx->ustrSymLinkName);
#018
#019     //调用 IoDetachDevice()把 fdo 从设备栈中脱开
#020     if (pdx->NextStackDevice)
#021        IoDetachDevice(pdx->NextStackDevice);
#022
#023     //删除 fdo
#024     IoDeleteDevice(pdx->fdo);
#025    KdPrint(("Leave HandleRemoveDevice\n"));
#026    return status;
#027  }
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录中找到。

- 代码 15 行，设置此 IRP 的状态为顺利完成。
- 代码 16 行，调用默认的 PNP 的 IRP 的处理函数。
- 代码 17 行，删除此设备对象的符号链接。
- 代码 19~21 行，从设备堆栈中卸载此设备对象。
- 代码 24 行，删除设备对象。

1.4.7　HelloWDM 对其他 IRP 的回调函数

此处对创建、关闭、读写设备的默认处理同 HellloDDK 中，所以不重复说明。

```
#001  /*************************************************************************
#002  * 函数名称:HelloWDMDispatchRoutine
```

```
#003   * 功能描述:对默认 IRP 进行处理
#004   * 参数列表:
#005         fdo:功能设备对象
#006         Irp:从 I/O 请求包
#007   * 返回值:返回状态
#008   ******************************************************************************/
#009   #pragma PAGEDCODE
#010   NTSTATUS HelloWDMDispatchRoutine(IN PDEVICE OBJECT fdo,
#011                                    IN PIRP Irp)
#012   {
#013     PAGED_CODE();
#014     KdPrint(("Enter HelloWDMDispatchRoutine\n"));
#015     Irp->IoStatus.Status = STATUS_SUCCESS;
#016     Irp->IoStatus.Information = 0; // no bytes xfered
#017     IoCompleteRequest( Irp, IO_NO_INCREMENT );
#018     KdPrint(("Leave HelloWDMDispatchRoutine\n"));
#019     return STATUS_SUCCESS;
#020   }
```

1.4.8 HelloWDM 的卸载例程

由于 WDM 式的驱动程序将主要的卸载任务放在了对 IRP_MN_REMOVE_DEVICE 的处理函数中，在标准的卸载例程几乎没有什么需要做的。在这里，仅仅是打印几行调试信息。

```
#001   /******************************************************************************
#002   * 函数名称:HelloWDMUnload
#003   * 功能描述:负责驱动程序的卸载操作
#004   * 参数列表:
#005         DriverObject:驱动对象
#006   * 返回值:返回状态
#007   ******************************************************************************/
#008   #pragma PAGEDCODE
#009   void HelloWDMUnload(IN PDRIVER OBJECT DriverObject)
#010   {
#011     PAGED_CODE();
#012     KdPrint(("Enter HelloWDMUnload\n"));
#013     KdPrint(("Leave HelloWDMUnload\n"));
#014   }
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

1.5 HelloWDM 的编译和安装

HelloWDM 的编译和安装与 HelloDDK 的过程有一些不同，尤其是安装过程。这主要是因为 HelloWDM 是即插即用式的驱动程序，并且需要借助 INF 文件安装。

1.5.1　用 DDK 编译环境编译 HelloWDM

同 HelloDDK 一样，首先介绍用 DDK 编译环境编译，再介绍用 VC6 IDE 的编译方法。编译 HelloWDM 需要编写两个脚本文件，makefile 和 Sources。makefile 同 HelloDDK 中的一样，这里不再给出。Sources 稍有不同，如下：

```
TARGETNAME=HelloWDM
TARGETTYPE=DRIVER
DRIVERTYPE=WDM
TARGETPATH=OBJ

INCLUDES=$(BASEDIR)\inc;\
        $(BASEDIR)\inc\ddk;\

SOURCES=HelloWDM.cpp\
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

将两个文件和源文件放在同一目录下，进入“Windows XP Checked Build Environments”编译环境。用 cd 进入对应目录，运行 build，会编译出相应的 HelloWDM.sys 文件。

1.5.2　HelloWDM 的编译过程

HelloWDM 的编译过程和 HelloDDK 的方法类似

① 用 VC 建立一个新工程，步骤同 HelloDDK。

② 将两个源文件 HelloWDM.h 和 HelloWDM.cpp 复制到工程目录中，并添加到工程中，步骤同 HelloDDK。

③ 增加新的编译版本，去掉 Debug 和 Release 版本，步骤同 HelloDDK。

④ 修改工程属性。在 VC 中选择“Project”|“Setting”。

⑤ 选择 General 选项卡。将“Intermediate files”和“Output files”改为 MyDriver_Check。

⑥ 选择 C/C++选项卡，将原有的 Project Options 内容全部删除，替换成如下内容：

```
/nologo /Gz /MLd /W3 /WX /Z7 /Od /D WIN32=100 /D _X86_=1 /D WINVER=0x500 /D DBG=1
/Fo"MyDriver_Check/" /Fd"MyDriver_Check/" /FD /c
```

⑦ 选择 Link 选项卡，将将原有的 Project Options 内容全部删除，替换成如下内容：

```
wdm.lib /nologo /base:"0x10000" /stack:0x400000,0x1000 /entry:"DriverEntry"
/subsystem:console /incremental:no /pdb:"MyDriver Check/HelloWDM.pdb" /debug
/machine:I386 /nodefaultlib /out:"MyDriver_Check/HelloWDM.sys" /pdbtype:sept
/subsystem:native /driver /SECTION:INIT,D /RELEASE /IGNORE:4078
```

⑧ 修改 VC 的 lib 目录和 include 目录，同 HelloDDK。

⑨ 编译。按下 F7 键，编译生成 HelloWDM.sys。

1.5.3　安装 HelloWDM

WDM 式驱动程序和 NT 式驱动程序的安装有很大的出入，首先为了安装 HelloWDM

驱动程序，需要为驱动程序编写一个 inf 文件。inf 文件描述了 WDM 驱动程序的操作硬件设备的信息和驱动程序的一些信息。下面是这个 inf 文件的代码，请将这个文件和其他源文件放在同一个目录里。

HelloWDM.inf：

```
#001  ;; Win2K DDK 文档中有详细参考
#002
#003  ;--------- 版本区域 ---------------------------------------------------
#004
#005  [Version]
#006  Signature="$CHICAGO$"
#007  Provider=Zhangfan_Device
#008  DriverVer=11/1/2007,3.0.0.3
#009
#010  ; 如果设备是一个标准类别，使用标准类的名称和 GUID
#011  ; 否则创建一个自定义的类别名称，并自定义它的 GUID
#012
#013  Class=ZhangfanDevice
#014  ClassGUID={EF2962F0-0D55-4bff-B8AA-2221EE8A79B0}
#015
#016
#017  ;--------- 安装磁盘节-----------------------
#018
#019  ; 这些节确定安装盘和安装文件的路径
#020  ;读者可以按照自己的需要修改
#021
#022  [SourceDisksNames]
#023  1 = "HelloWDM",Disk1,,
#024
#025  [SourceDisksFiles]
#026  HelloWDM.sys = 1,MyDriver_Check,
#027
#028  ;--------- ClassInstall/ClassInstall32 Section -------------------------
#029
#030  ; 如果使用标准类别设备，下面的是不需要的
#031
#032  ; 9X Style
#033  [ClassInstall]
#034  Addreg=Class_AddReg
#035
#036  ; NT Style
#037  [ClassInstall32]
#038  Addreg=Class_AddReg
#039
#040  [Class_AddReg]
#041  HKR,,,,%DeviceClassName%
#042  HKR,,Icon,,"-5"
#043
#044  ;--------- 目标文件节 -----------------------------------------
#045
#046  [DestinationDirs]
```

```
#047  YouMark_Files_Driver = 10,System32\Drivers
#048
#049  ;--------- 制造商节 ----------------------------------
#050
#051  [Manufacturer]
#052  %MfgName%=Mfg0
#053
#054  [Mfg0]
#055
#056  ; 在这里描述 PCI 的 VendorID 和 ProductID
#057  ; PCI\VEN aaaa&DEV bbbb&SUBSYS ccccccccc&REV dd
#058  ;改成自己的 ID
#059  %DeviceDesc%=YouMark DDI, PCI\VEN 9999&DEV 9999
#060
#061  ;---------- DDInstall Sections ------------------------------------------
#062  ; --------- Windows 9X -----------------
#063
#064  ; 如果在 DDInstall 中的字符串超过 19，将会导致严重的问题
#065  ;
#066
#067  [YouMark_DDI]
#068  CopyFiles=YouMark_Files_Driver
#069  AddReg=YouMark_9X_AddReg
#070
#071  [YouMark 9X AddReg]
#072  HKR,,DevLoader,,*ntkern
#073  HKR,,NTMPDriver,,HelloWDM.sys
#074  HKR, "Parameters", "BreakOnEntry", 0x00010001, 0
#075
#076  ; --------- Windows NT -----------------
#077
#078  [YouMark_DDI.NT]
#079  CopyFiles=YouMark_Files_Driver
#080  AddReg=YouMark_NT_AddReg
#081
#082  [YouMark DDI.NT.Services]
#083  Addservice = HelloWDM, 0x00000002, YouMark_AddService
#084
#085  [YouMark_AddService]
#086  DisplayName = %SvcDesc%
#087  ServiceType = 1 ; SERVICE KERNEL DRIVER
#088  StartType = 3 ; SERVICE_DEMAND_START
#089  ErrorControl = 1 ; SERVICE_ERROR_NORMAL
#090  ServiceBinary = %10%\System32\Drivers\HelloWDM.sys
#091
#092  [YouMark NT AddReg]
#093  HKLM, "System\CurrentControlSet\Services\HelloWDM\Parameters",\
#094  "BreakOnEntry", 0x00010001, 0
#095
#096
#097  ; --------- 文件节(common) -------------
#098
#099  [YouMark Files Driver]
#100  HelloWDM.sys
```

```
#101
#102   ;--------- 字符串节-----------------------------------------------------
#103
#104   [Strings]
#105   ProviderName="Zhangfan."
#106   MfgName="Zhangfan Soft"
#107   DeviceDesc="Hello World WDM!"
#108   DeviceClassName="Zhangfan_Device"
#109   SvcDesc="Zhangfan"
```

此段代码可以在配套光盘中本章的 WDM_Driver 目录下找到。

HelloWDM 是个虚拟设备，安装需要如下方法。

① 首先，在第一次安装 HelloWDM 驱动程序时，进入控制面板，选择添加硬件。系统会自动检索是否有新设备插入，并弹出对话框询问是否将硬件连接到计算机，选择是，如图 1-18 所示。

② 在弹出的下一个对话框中选择“添加新的硬件设备”，并单击“下一步”按钮，如图 1-19 所示。

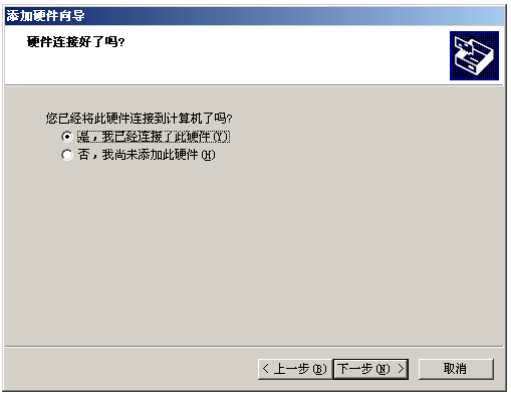


图 1-18　添加虚拟设备

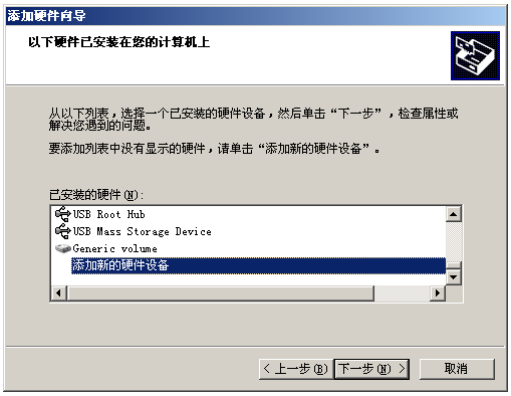


图 1-19　选择添加新硬件

③ 在弹出的下一个对话框中选择“安装我手动从列表选择的硬件（高级）”，并单击“下一步”按钮，如图 1-20 所示。

④ 在弹出的下一个对话框中选择显示所有设备，并单击“下一步”按钮，如图 1-21 所示。

⑤ 在弹出的下一对话框中选择“从磁盘安装”并选择 inf 文件，如图 1-22 所示。

⑥ 最后系统提示安装成功，如图 1-23 所示。

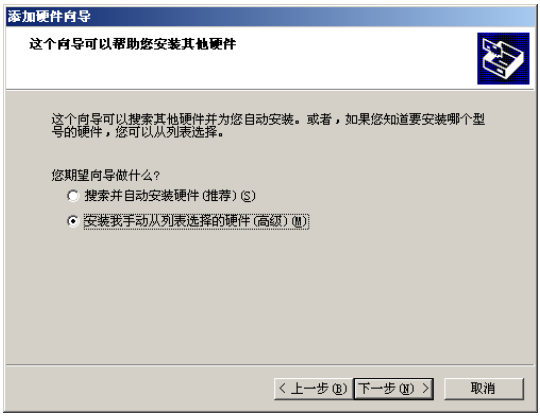


图 1-20　手动安装设备

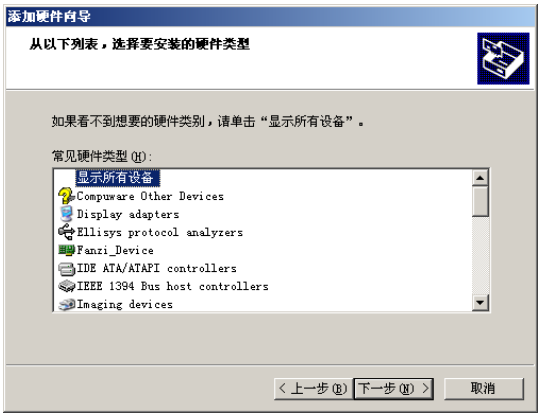


图 1-21　选择显示所有设备

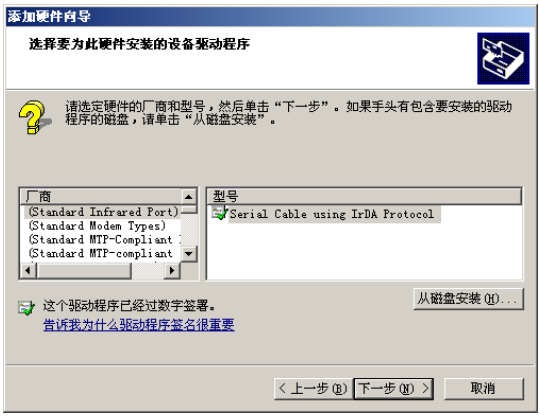


图 1-22　从磁盘安装设备驱动

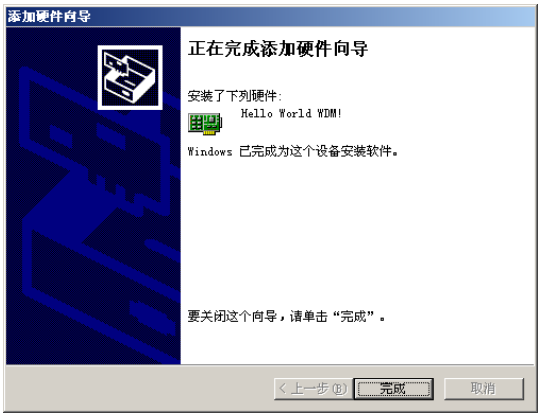


图 1-23　安装完毕

和 HelloDDK 一样，也可以在设备管理器中找到这个虚拟的设备，如图 1-24 所示。

这种安装 WDM 驱动程序的方法，过于烦琐，同时需要等待系统枚举设备，因此会等待很久的时间。在这里，笔者向读者介绍一个快速安装 WDM 程序的方法。使用一个叫做 EzDriverInstaller 的工具，这个工具也是 DriverStudio 自带的一个工具软件。打开 EzDriverInstaller，选择 “File”|“Open”，在弹出的对话框中，选择需要安装的 inf 文件，单击“Add New Device”按钮，如图 1-25 所示。很快地，驱动程序就被加载了，EzDriverInstaller 还提供了删除驱动程序功能、开启/关闭驱动功能、屏蔽驱动功能、重启驱动功能等。在调试驱动的时候，EzDriverInstaller 不失为一个很好的加载 WDM 驱动程序的工具。

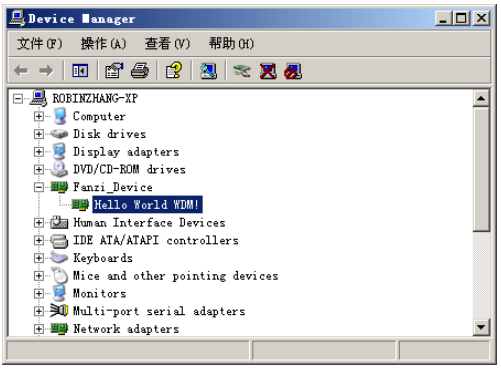


图 1-24　在设备管理器中显示 HelloWDM 设备

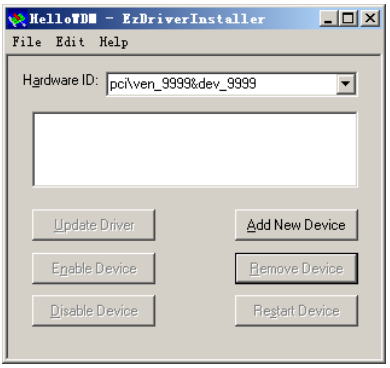


图 1-25　用 EzDriverInstaller 安装 HelloWDM 驱动

1.6　小结

本章笔者带领读者编写了两个非常简单的驱动程序——HelloDDK 和 HelloWDM，并详细说明了这两个驱动程序的编译过程、安装过程。笔者在初学 Windows 驱动程序开发的时候，对编译的方法摸索了很久，尤其是用 VC 编译驱动程序（其实用 VC 还是 DDK 环境编译本质完全一样，都是用 cl.exe 进行编译）。这方面的资料非常少，就连微软官方的 DDK 文档中都很少有介绍。通过本章的学习，读者能很快地将这两个驱动程序编译并且顺利安装。在以后的章节里，笔者会陆续对这两个驱动程序做深入的剖析，并且重复利用这两个驱动程序的框架。驱动程序的代码量往往都很小，在这两个驱动程序的基础上，就能编写出很多有意思的驱动程序来。相信读者已经进入 Windows 驱动开发的大门了。