第五章 DCS 数据采集技术 10 页 1.6 万字

目前各种 I/O 设备提供的对外数据接口可分为以下几类：

1、 数字通讯接口，包括串口类、以太网（TCP/IP 协议）类、现场总线类、仪器总线类通讯接口（如 GPIB 等）。

2、 模拟量通道输出，设备直接提供 4-20mA、1-5V 或继电器接点信号等。

力控®具有世界上大部分主流设备的 I/O 接口程序，对 GPIB 总线以及 Honeywell、Yokogawa、Foxboro、Fisher-Rosemount 等厂家的 DCS 也能够支持。

除通常意义上的数据采集外，力控®可以利用采集到的实时数据对装置进行实时建模，插入力控®自己的先进控制控件，实施先进控制。

5.1 对一个设备上的数据定义不同的采集周期

如果一台设备上有 1000 个实时数据需要采集，而在这 1000 个数据中只有 10 个是经常刷新且需要密切监视的，其余 990 个全部是辅助数据，但是也需要时常查看。如果把这 1000 个数据同等地对待，采用统一的扫描周期进行采集，就会严重影响 10 个重要数据的刷新速度。怎样既保证 1000 个数据都能够采集，又确保这 10 个重要数据的采 集速度呢？有两种办法：办法 1：为一个设备定义两个逻辑设备，使其具有不同的采集周期，如图 5-1 所示。但是这种方法定义的最长扫描周期为 10 分钟。

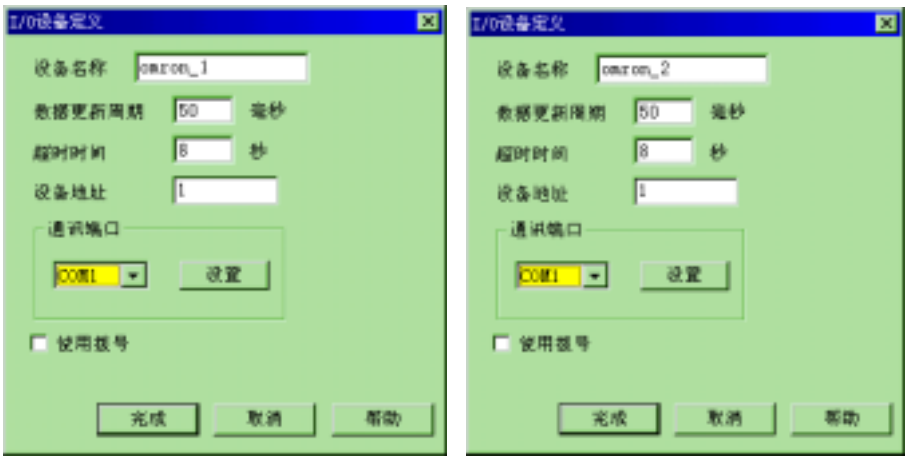

图 5-1

办法 2：不用上面的办法，一台设备只定义一个设备名称也可以达到要求。因为力控®的 I/O 驱动对画面中不显示而且没有组态历史趋势和报警的数据是不采集的，仅当画面中显示这个数据时才进行采集。因此将不常用的数据单独组态在一个或几个画面中，使用完毕马上关闭就不会影响整个采集速度。这种方法适用于存在有大量不需要快速更新的数据的情况。

5.2 合理设置扫描周期，避免引起设备死机

有些 I/O 设备内部只有一个 CPU，同时负责数据通讯和计算，如果在力控®上设置的数据扫描周期太快容易使设备死机，因此在设置这一参数时应该慎重，最好通过多次试验确定一个合适的扫描周期。一般的串口设备的扫描周期可设在 10-100 毫秒之间。

5.3 通过拨号方式与 I/O 设备通讯

力控®的所有串口 I/O 驱动程序都支持通过 MODEM 以拨号方式与设备通讯。只要正确设置电话号码即可，如图 5-2 所示。

5.4 通讯状态监视、设备状态数据的读取

力控®为每一个 I/O 设备自动定义了一个系统变量，假如系统中有一个设备 PLC1，则每当 PLC1 不能与力控®正常通讯时，系统变量$IO PLC1 的值就会被置为 1。I/O 设备故障属于系统报警。计算机通讯口故障、电缆、PLC 端通讯口的故障、PLC 通讯口与计算机通讯口的参数设置不一致都会造成这种结果。还有一种可能，就是数据连接项错误，如果计算机的命令发给 PLC 的只读参数，PLC 是不会予以理睬的。

5.5 怎样用 I/O 驱动程序调试 I/O 设备

力控®的 I/O 驱动程序有数百个，针对每一种设备都有一个独立的程序。当力控®实时数据库 DB 没有启动时，单独启动 I/O 驱动可以作为本地 I/O 设备调试工具使用。此时可以测试计算机与 I/O 设备的通讯情况，摸索最佳的扫描周期。

图 5-3

菜单“ 设置[S]/参数 ”用来规定 I/O 通讯过程中是否显示计算机发出和设备响应的通讯信息。如图 5-3 所示。

菜单“ 工具[T]/工具 ”用来在不启动实时数据库及其数据连接项的情况下执行与 I/O 设备的通讯。弹出对话框如图 5-5 上部所示。此时可以按“ 参数设置 ”按钮设置通讯参数，如图 5-6 和 5-7 所示，主要是设置串口的 DCB 参数、IP 地址等。

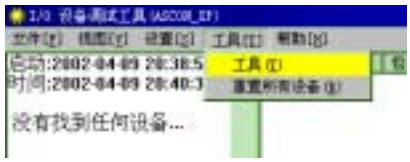

图 5-4

正确设置参数后，按“ 连接设备 ”按钮,如果成功的话,"连接设备"按钮的标题变成"断开连接",表示可以收发数据了。如果在"参数设置"中设置"周期性发送周期"不为 0,则在"连接设备"后会出现"周期发送"按钮,否则出现"手动发送"按钮。使用"周期发送"或"手动发送"按钮,可以周期性或一次性地发送编辑框中的数据了。

图 5-5

编辑框中数据的格式缺省是混合方式的,如果你想发送编辑框中的数据,这也是唯一的数据类型,它的形成规则如下:任何 ASCII 码(除'[')可以直接输入,'['可以使用"[[]"来输入;"[]"内是由' '(空格)分隔的转意字符,它们的意义为:

'[': 用来输入'[';

'*': 在发送数据是表示延时 1 毫秒
'#': 在发送数据是表示延时 10 毫秒
'$': 在发送数据是表示延时 100 毫秒
'!': 在发送数据是表示延时 1000 毫秒
'\': 表示它后面的数据是十进制的(缺省是 16 进制)
' '(空格): 作为分隔符,任何未定义的字符都可以作为分隔符,最好使用空格。
0~9: 可以用来输入 10 进制或 16 进制数据。
A~F/a~f: 可以用来输入 16 进制数据。
例子: abcv[[30 *#$!]345 对应的 16 进制数据串为: 61H,62H,63H,76H,5BH,30H,33H,34H,35H; 而且在 30H 之后有 1111 毫秒的延时.
当用在其它情况(不是用来发送数据)时,唯一的差别是没有了延时的概念.

图 5-6

图 5-7

"其它工具":校验使用混合格式的数据,计算常用的校验码，ASCII 码表显示使用 16 进制和 10 进制显示的 ASCII 码表，各种数据转换把混合格式,16/32 位整数,32 位浮点数等转换位十几种常用数据格式,除混合格式外,其它数据格式为直接用空格分隔的数据串

5.6 如何开发 I/O 设备驱动程序

在力控®中有一个 SDK 工具包，叫做 FIOS SDK，可以开发硬件设备与计算机的通讯接口程序。最简单的只需要编写几个函数就可以可以做自己的 I/O 驱动，现在支持的硬件类型有串口(RS485/232/422)，网络，板卡，硬件厂家提供 DLL 等多种方式的通讯类型。在该 SDK 中开发自己的通讯接口，不需要关心硬件和计算机通讯的细节，只需要了解通讯协议就可以了。

如果通讯协议很复杂，该 SDK 中提供了足够灵活的手段满足不同层次的需要。例如：在设备初始化时发什么命令或做其他处理，动态改变硬件通讯参数等等。

5.6.1 FIOS 开发包简介

FIOS 负责完成与各种 I/O 设备进行数据交换。一方面，它把从 I/O 设备采集到的实时过程数据发送给数据库 DB，另一方面，从 DB 发出的下置数据也通过 FIOS 发送给 I/O 设备。

根据监控 PC 与 I/O 设备之间通信机制的不同，FIOS 主要支持两种工作方式：同步方式与异步方式。

异步方式适用以下一类 I/O 设备：这类 I/O 设备一般可以独立运行，与监控计算机之间通过串口、网络或 MODEM 连接。与监控计算机之间通过明确的消息传送（文本或二进制消息）完成数据交换。数据交换过程为异步方式。

同步方式适用以下一类 I/O 设备：这类 I/O 设备或者依赖 PC 运行（如：插在 PC 插槽内），或者独立运行。但与监控计算机之间主要通过直接访问方式进行数据交换，具体形式包括：寄存器直接访问（如：板卡）、API 函数调用、ActiveX 控件访问等。数据交换过程为同步方式。

下面列举了 FIOS 可实现的一些基本功能：

底层通信功能：1)、串口通信，包括：RS232/422/485。2)、TCP/IP 网络通信。3)、MODEM

通信，通过模拟 MODEM 在电话网上通信。4)、寄存器访问，如：各种 DAS 板卡。5)、其它。

链路控制功能：用 M 代表 Master，即上位机（监控 PC 工作站）；S 代表 Slaver，即下位机（各种 I/O 设备）。对于异步方式，FIOS 支持多种链路控制方式。链路控制方式支持以下几种方式：1)、M 请求，S 应答方式。2)、M 请求，S 无应答方式。3)、S 主动发送，M 被动等待。另外，对一次完整数据处理（读或写）过程，支持以下方式：1)、1 次请求，0 次应答方式。2)、1 次请求，1 次应答方式。3)、1 次请求，多次应答方式。4)、0 次请求，1 次应答方式。5)、多次请求，多次应答方式。

冗余功能：FIOS 支持的冗余方式包括：1)、单监控站，双 I/O 冗余。2)、双监控站，单 I/O 冗余。3)、双监控站，双 I/O 冗余。4)、对于总线型设备（如 RS485），提供总线监测功能，可实现对冗余通信网络的保护和监测。

前端机功能：DB 与 IO Server 不在同一工作站上，IO Server 运行在前端机上，前端机与操作站之间通过串口、TCP/IP 网络或 MODEM 进行通信。

硬件测试与远程调试功能：使用 FIOS 可完成对 I/O 设备的简单测试功能。另外可实现远程调试。

故障诊断与恢复功能：FIOS 提供诊断机制，在较短的采集周期内报告故障的发生，诊断出下位机故障情况。当下位机更换或恢复后，不需要对 FIOS 及相关程序进行任何人工干预，而在较短时间自动恢复通信。当某一台、几台或部分通道发生故障，FIOS 要自动优化通信链，使其与其他下位机或通道之间的通信不受影响，保证通信效率。

界面显示功能：为测试、调试、运行维护方便，FIOS 提供显示界面，可显示包括：发送、应答、状态信息，启动时间、分包数、分包信息，成功通信次数（发送次数、成功应答次）、故障次数等信息。

历史数据处理功能：对于某些能保存历史数据的设备（如：无纸记录仪等），FIOS 能将采集到的历史数据恢复到数据库 DB 中。

5.6.2 FIOS SDK 编程方式

FIOS SDK 提供了一种简洁的、面向对象的编程方式以缩短开发时间，降低开发难度。

FIOS SDK 提供标准的开发接口和程序模板，程序员仅需要根据 I/O 设备的具体通信协议或驱动接口说明，填写几个扫描函数的实现代码，进行必要的调试与测试，即可完成一种 FIOS 的开发。

FIOS 提供的开发工具封装了大部分程序员不必关心的技术环节，如：底层通信功能（串口通信、网络通信等）、设备超时处理、设备故障诊断等。同时 FIOS 提供各种调试工具，方便程序员进行系统测试。

FIOS 开发环境完全基于 32 位 Windows 平台。它使用动态链接库（DLL）技术将程序员开发的代码整合到力控®系统中。FIOS 提供给程序员的开发接口为 API 函数和 C++类库。

5.6.3 FIOS SDK 组件及示例程序

FIOS SDK 主要由 4 部分组成：设备组态接口（Iodevui）、数据连接组态接口（Ioitemui）、编程接口 Ioapi 和描扫程序 Ioscan。Iodevui:负责管理设备组态过程。Ioitemui:负责管理数据连接组态过程。Ioapi:负责完成与 I/O 设备间的数据交换，包括：对通信协议的解析、数据格式的转换等。Ioscan:主要完成对 Ioapi 部分的 dll 代码进行周期性地扫描。同时完成与 I/O 设备的底层通信（串口通信、网络通信等），以及设备超时处理、设备故障诊断等。Ioscan 还负责与数据库 DB 之间的通信与协作。它把从 I/O 设备采集到的数据经 Ioapi 解析转换后提交给 DB，或将 DB 下置给 I/O 设备的数据经 Ioapi 解析转换后写入 I/O 设备。Ioscan 是 FIOS SDK 提供的一个标准软件工具。 程序员仅需要开发 Iodevui、Ioitemui、Ioapi 三部分的代码。

示例程序

FIOS SDK 提供了两个示例：DemoController 与 DemoModbus。

DemoController 是一个初级编程示例，它能引导初学者快速掌握开发 FIOS 的基本概念和方法。DemoModbus 是一个实用编程示例，它采用标准 MODBUS 通信协议，通过该示例，可以掌握在力控®平台上开发标准 MODBUS 设备 I/O 驱动程序的方法。

FIOS SDK 的全部内容都是在安装在力控®自动安装的,在力控®目录下的子目录 Fiossdk 中。FIOS SDK 主要包含以下几部分内容：Examples，程序示例、仿真程序。Include，头文件。Manual，文档说明。Utility，调试工具。

这 2 个示例具有一定的代表性，它们体现了 FIOS SDK 的主要功能。FIOS SDK 提供了这 2 个示例的全部源代码，在它们的基础上，稍做改动，就可以开发出新的 FIOS。我们把象这 2 个示例源程序一样具有模板作用的程序称为 I/O 模板程序。为了提高开发效率，我们建议尽量使用 I/O 模板程序，这在一定程度上，也减少、降低了编程错误的发生。

常用术语

我们把 FIO SDK 中经常涉及的一些概念给出定义，有些术语虽然是通用名词，但在 FIOSDK 中有特定含义。这些术语有一些在前文给出了解释，有一些会在后文中陆续给出解释。

FIOS	ForceControl I/O Server，即力控®I/O 驱动程序
FIOS SDK	FIOS 软件开发工具包
FCINSTDIR	力控®软件系统的安装目录
FCAPPINSTDIR	用力控®创建的工程应用的目录
IOID	唯一区别各个 I/O 驱动程序的 I/O 标志
Iodevui	设备组态接口
Ioitemui	数据连接组态接口
Ioapi	编程接口
Ioscan	扫描程序
I/O 模板程序	FIOS 工 SDK 附带示例的源程序
I/O 配置文件	设备组态时的缺省参数设置保存文件
连接项结构	保存数据连接信息的数据结构 IOITEMDEF
I/O 描述文件	定义设备的类别、厂商、型号、通信方式等参数的文本文件 Iodesc.txt
程序员	在本文档范围内专指用 FIOS SDK 进行开发的技术人员
扫描函数	包含在 Ioapi 中的 API 函数，它们由扫描程序周期扫描。扫描函数完成对设备数据解析及格式转换
IOC	Input Output Class（输入输出类库）的缩写。

5.6.4 设备组态接口 IODEVUI.DLL

I/O 描述文件

在使用力控®进行组态时，一般均涉及定义 I/O 设备的过程。在定义设备时，要选择设备的类别（PLC、智能仪表等)、厂商、设备型号或通信协议，然后根据设备通信方式（串口方式、网络方式、其它方式等）设置参数。以上关于一种设备的信息（类别、厂商、型号、通信方式等）完全是由 I/O 描述文件决定的。I/O 描述文件是一个标准文本文件，根据其规定的填写格式，由程序员根据具体设备自行填写。下面介绍 I/O 描述文件的填写格式。

I/O 描述文件的文件名为 IODESC.TXT，安装目录为：“FCINSTDIR\IO Servers\IOID\”。

IO 文件说明格式为：

类别;厂商或 IO 程序描述;执行文件名称<CR><LR>

子类型 1;类型号;资源标志;提供设备地址<CR><LR>
子类型 2;类型号;资源标志;提供设备地址<CR><LR>
.....

注意，子类型号不能重复。<CR><LR>表示回车换行.最上面一行是驱动程序的总体描述，包括三项。各项之间必须以分号“；”分隔。各项内容不能含有分号“；”。

各项含义如下：类别，驱动程序所属类别，现分为以下几类：PLC、智能仪表、智能模块、变频器。程序员也可以自行扩展。厂商或 IO 程序描述，I/O 设备生产厂商名称，协议名称，如西门子。执行文件名称，I/O 驱动程序（运行程序）的名称，如 opto_drv.exe
接下来几行为驱动程序所包含的设备类型的描述，如西门子包括 S5，S7 等，每一子类别一行，每行包括三项，各项之间必须以分号“；”分隔。各项内容不能含有分号“；”。各项含义如下：子类型，设备类型描述。如 S5。类型号，设备类型编号，类型号不能重复。合法的值为 0，1，2，3 等。使用计算机资源，使用计算机何种通信资源通信，合法的值为 0、1、2 等。含义如下：0，同步通信方式；1，串口通信方式；2，TCP/IP 网络通信方式； 3，MODEM 通信方式；4，板卡方式；5，并口通信方式。提供设备地址:1 表示需要指定设备地址，否则表示不需要设备地址。

管理程序会自动将相同厂商或 IO 程序描述相同的驱动程序归为同一树下。

开发 Iodevui

力控®组态环境 DRAW 中的设备管理功能提供了一个根据 I/O 描述文件可灵活配置的标准设备组态接口。这个组态接口提供了一些对常用设备参数进行设置的方法。如：设备名称、设备地址、通信端口、端口参数等。如下图所示：

对于很多设备，如果标准设备组态接口能够满足要求，就不再需要自己编写 Iodevui 接口程序了。比如示例 DemoController 采用的就是标准设备组态接口。而示例 DemoModbus 因为涉及一些特殊的参数设置，就需要自己编写 Iodevui 接口程序了。

因此，Iodevui 接口程序实际上就是对标准设备组态接口的一个补充和扩展，并可由程序员灵活控制。Iodevui 要以 DLL 形式提供。该 DLL 必须是 MFC 扩展 DLL。该 DLL 的缺省文件名称为 IODEVUI.DLL，该文件必须安装在目录“FCINSTDIR\IO Servers\IOID\”下。

在进行设备组态时，力控®的 I/O 设备管理程序会自动检查在目录“FCINSTDIR\IO Servers\IOID\”下是否存在 IODEVUI.DLL 文件。如果存在，则首先根据 I/O 描述文件的格式，调出标准设备组态接口界面，当用户确认后，再调出 Iodevui 组态接口界面；若不存在该文件，则只调出标准设备组态接口界面。

示例 DemoModbus 的 Iodevui 接口程序可以做为开发 Iodevui 的模板程序。我们结合示例 DemoModbus 的 Iodevui 模板程序具体解释实现过程。查看头文件 Iodevui.h 可以发现，Iodevui.dll 主要实现 3 个输出函数：

```
extern "C" AFX_EXT_API long AddIoDev(const char* szDeviceName, int nType);
extern "C" AFX_EXT_API long ModIoDev(const char* szDeviceName);
extern "C" AFX_EXT_API long DelIoDev (const char* szDeviceName);
```

在进行设备组态时，当增加一个设备时，力控®设备管理程序会自动调用 AddIoDev()函数；当修改一个已创建设备时会调用 ModIoDev()函数；当删除一个设备时会调用 DelIoDev ()函数。

其中，参数 szDeviceName 为 I/O 设备名称（输入值，组态时由用户指定）。nType 为设备子类型号，由程序员在 I/O 描述文件中指定。返回值为 0 表示操作成功；其它表示操作失败。为了较好地实现程序结构化，本模板程序提供了一个 CDevMan 类对设备及组态操作过程进行管理。Iodevui.dll 的 3 个输出函数 AddIoDev()、ModIoDev()DelIoDev ()的具体实现过程是在 CDevMan 的三个成员函数 Add()、Mod()和 Del()中实现的。

首先看一下 Add()的实现代码：

```
//*************************************************************************
// 添加 I/O 设备
// szDeviceName，设备名称(输入值)
// nType，设备子类型(用于一个驱动程序驱动多种类型设备)(输入值)
// 返回值说明：0，操作成功；其它，操作失败
//*************************************************************************
long CDevMan::Add(const char* szDeviceName, int nType)
{
    if(Find(szDeviceName))
    {
        AfxMessageBox("该数据源名已经存在！");
        return -1;
    }
    CDevice* pDev = new CDevice(szDeviceName,nType);
    if(CallDialog(pDev))
    {
        m_list.AddTail(pDev);
        Store();
        return 0;
    }
    else
        delete pDev;
    return -1;
}
```

程序的一开始，调用 Find()函数来查找是否已有相同的设备名存在，如果有给出提示并返回-1 表示操作失败，否则生成一个 CDevice 对象并调用 CallDialog 函数来显示一个对话框，让用户做进一步的选择，如果用户进行确认，操作成功，它把此 CDevice 对象加入设备链表中，并调用 Store 函数来保存设备信息。另外两个函数和它类似。

Store()函数如下：

```
void CDevMan::Store()
{
    CFile file;
```

```
    if(file.Open((const char*)("ddeacc.dat"),CFile::modeReadWrite|CFile::modeCreate))
    {
      CArchive ar(&file, CArchive::store);
      Serialize(ar);
      ar.Close();
      file.Close();
    }
}
```

该函数它先打开 ddeacc.dat 文件，如果不存在，就建立此文件。然后调用序列化函数对它进行保存，最后关闭此文件。再看一看序列化函数：

```
void CDevMan::Serialize(CArchive &ar)
{
    TRY
    {
      CObject::Serialize(ar);
      m_list.Serialize(ar);
    }
    CATCH(CFileException,e)
    {
      AfxMessageBox("文件版本不匹配！");
    }
    END_CATCH
}
```

该函数对 *m_list*（由 CDevice 类实例组成）进行序列化。在调用各个 CDevice 类实例的序列化函数时，如果是读取操作，会依次创建 CDevice 实例，并调用 CDevice 的序列化函数，随后把 CDevice 实例加入 *m_list* 链表。具体保存和读取的变量数据在 CDevice 类中控制，也就是说程序员针对不同的设备可以改写 CDevice 类，定义不同的成员变量，记录设备的不同的属性，对 CDevice 类重载 Serialize 即可实现设备的保存、加载、增加，删除和修改等功能。

我们再看一下 CDevice 类序列化的实现过程：

```
void CDevice::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
      ar << m_csName;          //设备名称
      ar << m_dwData;          //协议类型
    }
    else
    {
      ar >> m_csName;
      ar >> m_dwData;
    }
}
```

如果是保存操作，序列化函数会将参数自动存盘；如果是读取操作，序列化函数会从磁盘上读取参数值。

察看 CallDialog 函数可以发现，它生成了一个对话框，让用户做相应的选择，然后把用户选择的信息保存在 CDevice 类的成员函数中，以便于储存。

整个程序框架使用 CDevice 类来保存设备的信息。在 CallDialog 函数中使用一个对话框，来让用户进行选择设备的属性，并且在 CallDialog 函数中把它保存在 CDevice 类中。所以对于一个新的设备，程序员所要做的工作就是，分析设备的协议查看是否仅使用描述文件就可以完成设备的定义，如果不能，那么应该编制 IoDevUi.dll。这时应分析应该增加哪一些属性，定义哪一些 CDevice 类的成员变量，以及显示什么样的对话框，让用户做什么样的选择。所以程序员的工作重点在于修改 CDevice 类，增加成选变量，并重载它的 Serialize 函数，然后修改对话框，让用户做不同的选择，并把选择保存在 CDevice 类的成员变量中即可。

在该示例中，我们定义了 2 个设备参数：

CString m_csName; //设备的名称

DWORD m_dwData; //用于保存数据

这样只需在对话框中对 m_csName 和 m_dwData 赋值即可。

```
//*****************************************************************//
//调用对话框定义数据源
// pDev  数据源指针
//返回值  true  成功
//*****************************************************************//
bool CDevMan::CallDialog(CDevice* pDev)
{
    ASSERT(pDev);
    CDevDef dlg;
    dlg.m_name = pDev->m_csName;
    dlg.m_nProtocol = (pDev->m_dwData&0x01);
    dlg.m_inPackLong= ((pDev->m_dwData)>>8)&0xff;
    if(IDOK == dlg.DoModal())
    {
       pDev->m_csName = dlg.m_name ;
       pDev->m_dwData = (dlg.m_nProtocol&0x01);
          //m_dwData 的第 0 位为 1 表示是 RTU 方式  0  表示 ASCII 方式
       pDev->m_dwData = (pDev->m_dwData)|(dlg.m_inPackLong<<8);
          //m_dwData 的第 2 个字节储存包的最大长度
       return true;
    }
    return false;
}
```

不过到这里又产生了一个问题，那就是保存到文件中的数据在编写设备驱动程序时，怎么再读取出来呢？让我们先看一下在 Ioapi.dll 中的 OnLoadDevice 函数吧。

```
void OnLoadDevice(CManager* pManager)
{
    CString csPath = pManager->GetAppPath();//得到工程所在的目录
    csPath+="\\ddeacc.dat";//数据保存在了工程目录的 ddeacc.dat 中
    DWORD data;
    CString strtemp;
```

```
    short temp;
    CFile file;
    if(file.Open((const char*)csPath,CFile::modeRead))//打开该文件
    {
        CArchive ar(&file, CArchive::load);
//读取的第一个数据是定义的设备的个数，
        //但是因为可以通过 GetDeviceCount 函数得到设备的个数，
        //所以这里把读到的数据简单的丢掉。
        ar>>temp;
        int nDevCnt = pManager->GetDeviceCount();
        for (int i = 0; i < nDevCnt; i ++)
        {
            CDevice* pDevice = pManager->GetDevice(i);
            ar>>strtemp;//读取设备的名字
            ar>>data;//读取数据
    //这两句在讲到 Ioapi.dll 时再进行介绍
            pDevice->SetPrivateData(1,long(data&1));
            pDevice->SetPrivateData(2,long((data>>8)&0xff));
            DCB dcb;
            pDevice->GetDCB(dcb);
            dcb.fBinary = TRUE;
            dcb.fOutxCtsFlow = FALSE;
            dcb.fOutxDsrFlow = FALSE;
            dcb.fDtrControl = DTR_CONTROL_DISABLE;
            dcb.fNull = FALSE;
            dcb.fRtsControl = RTS_CONTROL_DISABLE;
            pDevice->SetDCB(dcb);
        }
        ar.Close();
        file.Close();//关闭文件
    }
    else
    {
        strtemp="对不起，没有找到";
        strtemp+=csPath;
        strtemp+="程序不能运行!!";
        AfxMessageBox(strtemp);
        PostQuitMessage(0);//没有找到文件，给出提示，并终止程序的运行。
    }
}
```

注意序列化的内容和顺序必须和 IoDevUi.dll 一致，否则会导致程序运行时产生错误。

5.6.5 Ioitemui 介绍及编程示例

在用力控®进行组态时，把数据库 DB 中的点参数与某种设备的具体通道建立连接的过

程被为数据连接过程。在进行数据连接时，一般还要指定数据转换格式、数据长度等参数。

数据连接过程对于不同的 I/O 设备，其形式和内容可能完全不同。因此必须针对不同的 I/O 设备，设计相应的数据连接形式，保存各种参数信息。

Ioitemui 接口主要完成的两部分功能，一是为用户进行数据连接组态时提供一个界面；另外就是将用户组态的设备参数信息用某种格式保存起来，以便在开发编程接口 Ioapi 时使用。我们定义了一个数据结构来保存设备参数信息，这就是数据连接项结构（下面简称连接项结构）IOITEMDEF。

IOITEMDEF 定义在 Ioitemui.h 中：

```
typedef struct IoItemDefStru
{
char str[64];
long n[8];
}IOITEMDEF;
```

这个结构是一个通用结构，由程序员自己赋值，自己解释。Ioitemui 要以 DLL 形式提供。该 DLL 必须是 MFC 扩展 DLL。该 DLL 的缺省文件名称为 IOITEMUI.DLL，该文件必须安装在目录“FCINSTDIR\IO Servers\IOID\”下。

Ioitemui 的工作过程如下：

在进行数据连接组态时，力控®的 DBMAN 管理程序会自动检查在目录“FCINSTDIR\IO Servers\IOID\”下是否存在 IOITEMUI.DLL 文件。如果存在，则调出数据连接组态接口界面。

下面介绍如何编写 Ioitemui 接口程序。

Ioitemui.dll 主要实现 1 个输出函数：extern "C" AFX_EXT_API long DoItemDlg(const char * szDeviceName, int nType, IOITEMDEF &item, char * szDesc, int nFlag);

其参数说明如下：

szDeviceName, 设备名称(输入值)。如果在力控®中定义了一个设备 Device1，那么在给该设备组点时，传给 DoItemDlg 的 szDeviceName 值就是字符串”Device1”。

nType, 设备子类型(用于一个驱动程序驱动多种类型设备)(输入值)。它的值在 IODESC.TXT 中指定（参见上一章对 I/O 描述文件的介绍）。

item, 数据连接项结构(返回值)。需要注意的是，item 除了是输出值外，也是输入值，DBMAN 管理程序每次调用 DoItemDlg()时，将上一次操作赋给 item 的值传递过来。

szDesc, 数据连接项描述，用于 DBMAN 程序显示的提示信息。

nFlag, 1 表示增加数据连接项,2 表示修改,0 表示删除(输入值)。其返回值 0 表示操作成功。其它, 操作失败。

Ioitemui.dll 的工作过程如下：

当用户打开数据组点连接对话框时，选中了一个点，并按下增加、修改或删除键，这时就会调用 Ioitemui.dll 的 DoItemDlg 函数。程序员应该在此函数中，弹出一个对话框让用户进行选择，在用户按下了 OK 键之后，把用户的选择保存在 item 中，以后编制 Ioapi.dll 时可以利用这些信息。

编程示例

我们先结合示例 DemoController 介绍如何开发 Ioitemui。

仿真器 SimController 的内部有数字区(DIO)和模拟区(AIO)。DIO 和 AIO 区通道范围为：0~255。每个 DIO 通道的数据的数值范围为 0 或 1。每个 AIO 通道数据的数值范围为 0~4095。所以我们应该在 DoItemDlg 函数中弹出一个对话框，用户可以在此对话框中选择输入通道和内存地址。输入通道有两个选项 DIO 通道和 AIO 通道供用户选择，内存地址可以让用户输入 0~255 之间的数据。

我们介绍一下如果不使用 I/O 模板，如何自己生成一个新的 Ioitemui 工程：

在 VC++环境下，选择菜单命令 new，选择新建工程，工程名为 Ioitemui，选择“ MFC Appwizard (dll) ”选项，在下一步 DLL 类型中选择“ MFC Extension DLL”型，然后按下“Finish ”键。即可创建一个新的 Ioitemui 工程。

打开 Ioitemui.cpp 文件，在文件的开头加入#include "Ioitemui.h"，把 Ioitemui.h 拷入本工程，然后在文件的最后键入：

```
long DoItemDlg(const char * strDataSour,int nType,IOITEMDEF &item,char * szDesc,int nFlag)
{
}
```

这就加入了 dll 的输出函数。

打开示例 DemoController 的 Ioitemui 模板程序，它的 DoItemDlg()函数实现过程如下：

```
long DoItemDlg(const char * szDeviceName,
            int nType,IOITEMDEF &item,char * szDesc,int nFlag)
    {
       CLinkDlg dlg;
       dlg.item_n[0] = item.n[0];
       dlg.item_n[1] = item.n[1];
       switch(nFlag)
       {
          case 0://删除操作
             return 0;
    //增加或修改操作
          case 1:
          case 2:
             if(dlg.DoModal()==IDOK)
             {
                item.n[0]=dlg.item_n[0];
                item.n[1]=dlg.item_n[1];
                sprintf(szDesc,"%s",(LPCSTR)dlg.m_desc);
                sprintf(item.str,"%s",(LPCSTR)dlg.m_desc);
                return 0;
             }
             break;
       }
       return 1;
    }
```

在这个模板程序里，还涉及一个对话框类 CLinkDlg 。这个对话框为用户进行数据连接组态时提供一个界面，其形式如下：

CLinkDlg 类有 2 个成员变量：

CString m_desc;//保存连接项描述

int item_n[2];//item_n[0]保存数据区类型，0 表示 DIO，1 表示 AIO；

//item_n[1]保存地址

在 CLinkDlg 的 WM_INITDIALOG 消息函数中进行如下处理：

```
BOOL CLinkDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    //在此处设置值使对话框的显示和是一次选择相同，以利于执行和上一次相近的操作
    m_CtrlChannel.SetCurSel(item_n[0]);    //设置操作选项为上一次的操作
    m_nAddr = item_n[1];                   //设置地址为上一次的值
    UpdateData(FALSE);
    return TRUE;
}
```

这些处理为了使对话框的显示和上一次选择相同，以利于执行和上一次相近的操作。在 ONOK 消息函数进行如下处理：

```
void CLinkDlg::OnOK()
{
    UpdateData(TRUE);//得到各个选项得值
    CString string;
    item_n[0] = m_CtrlChannel.GetCurSel();//保存操作选项
    m_CtrlChannel.GetWindowText(m_desc);
    item_n[1] = m_nAddr;//保存输入的地址
    m_desc+="  起始地址:";
    string.Format("%d",m_nAddr);
    m_desc+=string;
    CDialog::OnOK();
}
```

在这个函数里，把用户组态的内容（数据区的选择、地址的指定）保存在 item_n，并根据这些内容生成连接项描述。

5.6.6 扫描程序 IOSCAN

IOSCAN 是 FIOS 的一个主要程序模块。它负责完成对 IOAPI 部分的 DLL 代码进行周期性地扫描。同时完成与 I/O 设备的底层通信（串口通信、网络通信等），以及设备超时处理、设备故障诊断等。IOSCAN 还负责与数据库 DB 之间的通信与协作。它把从 I/O 设备采集到的数据经 IOAPI 解析转换后提交给 DB，或将 DB 下置给 I/O 设备的数据经 IOAPI 解析转换后写入 I/O 设备。

IOSCAN 是 FIOS SDK 提供的一个标准软件工具供程序员在调试和运行时直接使用。

FIOS 开发工具包里提供了 debug 和 release 版本的 IOSCAN 程序，在目录“FCINSTDIR\Fiossdk\Utility”下可以找到它们。Debug 版本的 IOSCAN 程序主要供程序员在调试时使用，它能提供更为丰富的调试信息。在使用时，需要把 IOSCAN.EXE 以及配套的几个 DLL 文件（即目录“FCINSTDIR\Fiossdk\Utility\Debug”下的 DLL 文件）拷贝到生成的 debug 版本的 IOAPI.DLL 文件的同一目录下（注意：debug 版本的 IOAPI.DLL 文件必须配合 debug 版本的 IOSCAN 程序，release 版本的 IOAPI.DLL 文件必须配合 release 版本的 IOSCAN 程序）。同时不要忘记将 IOSCAN.EXE 的文件名更改为要开发的 I/O 驱动的 IOID 名称。debug 版本的 IOSCAN 需要程序员手工启动或用 VC++调试启动。

5.6.7 编程接口 IOAPI.DLL

IOAPI 是 FIOS 提供的最主要的一个编程接口。程序员的主要工作就是开发 IOAPI 部分的程序代码。

IOAPI 提供了一组 API 函数和一些 C++类库。这组 API 函数规定了名称、参数及返回值，函数内容由程序员根据具备的 I/O 设备编程实现。C++类库则为程序员提供各种获取力控®I/O 组态信息、参数设置信息、与数据库 DB 进行数据交换等数据处理的方法。我们把这组 API 函数称为扫描函数，把这些 C++类库称为 IOC，IOC 是 Input Output Class（输入输出类库）的缩写。

程序员编写的 Ioapi 最后要形成 MFC 的扩展动态链接库（MFC Extension DLL），扫描函数是这个 DLL 的输出函数。当力控®系统运行时，力控®FIOS 的扫描程序 Ioscan 对 Ioapi 中扫描函数部分的 dll 代码进行周期性地扫描，它把从 I/O 设备采集到的数据经扫描函数解析转换后提交给 DB，或将 DB 下置给 I/O 设备的数据经扫描函数解析转换后写入 I/O 设备。

归结起来，开发 Ioapi 的主要内容就是用 IOC 编写扫描函数。

IOC 中的所有类库全部以纯虚类的形式提供，并且只有成员函数，没有成员变量。 目前 IOC 中主要包括 4 个类：CItem 、CPacket、CDevice、CManager。

CItem，数据项类。

CPacket，数据包类。

CDevice，设备类。

CManager，管理器类。

一个 FIOS 实例创建一个 CManager 实例。用户在组态时每定义一个设备，则创建一个 CDevice 实例。CManager 对所有的 CDevice 进行管理。一个 CDevice 实例，由一个或多个 CPacket 实例组成，而每个 CPacket 实例又由一个或多个 CItem 实例组成。每个 CItem 实例，对应数据库 DB 中的一个点参数，也就是对应 I/O 设备的一个“点”（如：设备的一个通道，一个参数等）。

IOC 提供的这 4 个类库，实际上就是对以上所述的这几种数据对象提供了一组操作方法，以供程序员更加灵活的控制程序。

Citem 类

CItem 类提供了对数据项对象的一组操作方法。一个数据项对象包含的是数据库 DB 中的一个点参数与 I/O 设备中一个物理通道的映射关系。CItem 使用的基本数据结构是 IOITEMDEF。一个 CItem 实例保存一个 IOITEMDEF 实例。IOITEMDEF 的定义如下:

```
typedef struct IoItemDefStru
{
    char str[64];
    long n[8];
}IOITEMDEF;
```

CItem 类的定义如下:

```
class CItem : public CObject
{
public:
    virtual IOITEMDEF* GetItemStru()=0; //取得数据连接项结构指针
    //设置连接项的可写属性，缺省时可写的，bAttribute 为 TRUE 设置为不可写。
    virtual void SetReadOnly(BOOL bAttribute = TRUE)=0;
    //设置连接项的可读属性，缺省时可读的，bAttribute 为 TRUE 设置为不可读。
    virtual void SetWriteOnly(BOOL bAttribute = TRUE)=0;
    virtual void SetData(short sData)=0; //按短整型格式设置采集数据
    virtual void SetData(long lData)=0; //按长整型格式设置采集数据
    virtual void SetData(double fData)=0; //按浮点型格式设置采集数据
    virtual void SetData(char* szData)=0; //按字符串格式设置采集数据
    //按字符串格式取得上一次用 SetData()设置的采集数据
    virtual void GetData(char* szData)=0;
    //设置私有数据，offset 范围：0~3
    virtual void SetPrivateData(unsigned short offset, long lPrivateData)=0;
    virtual long GetPrivateData(unsigned short offset)=0; //取得私有数据，offset 范围：0~3
    virtual CPacket* GetPacket()=0; //取得本连接项类所归属的数据包指针
    virtual CDevice* GetDevice()=0; //取得本连接项所归属的设备指针
    virtual CManager* GetManager()=0; //取得 IoScan 管理器指针
    //按浮点型格式设置历史数据
    virtual void SetHisData(HisInsDatStru* pHisInsDatStru,int nCount)=0;
};
```

各个函数的解释如下:

1. IOITEMDEF* GetItemStru()

功能：取得数据连接项结构指针。

参数：无。

返回值：数据项结构指针。

举例：

```
IOITEMDEF* pItemStru = pItem->GetItemStru();
long nCmdType = pItemStru->n[3];
```

2. void SetReadOnly(BOOL bAttribute = TRUE)

功能：设置连接项的写属性，缺省时连接项是可写的。

参数：TRUE：设置为不可写；FALSE：设置为可写。

返回值：无。

举例：

```
for (int i = 0; i < pPacket->GetItemCount(); i++)
{
    CItem* pItem = pPacket->GetItem(i);
    pItem->SetReadOnly();
}
```

3. void SetWriteOnly(BOOL bAttribute = TRUE)

功能：设置连接项的读属性，缺省时连接项是可读的。

参数：TRUE：设置为不可读；FALSE：设置为可读。

返回值：无。

举例：

```
for (int i = 0; i < pPacket->GetItemCount(); i++)
{
    CItem* pItem = pPacket->GetItem(i);
    pItem->SetWriteOnly();
}
```

4. void SetData(short sData)

功能：按短整型格式设置采集数据。

参数：短整形数据值。

返回值：无。

5. void SetData(long lData)

功能：按长整型格式设置采集数据。

参数：长整型数据值。

返回值：无。

6. void SetData(double fData)

功能：按浮点格式设置采集数据。

参数：浮点数据值。

返回值：无。

7. void SetData(char* szData)

功能：按字符串格式设置采集数据。

参数：字符串数据值。

返回值：无。

8. void GetData(char* szData)

功能：按字符串格式取得上一次用 SetData()设置的采集数据。

参数：存放最近一次设置的采集数据（字符串型数值）的字符串指针。缓冲区长度应不小于 32。

返回值：无。

举例：

```
char szData[32];
pItem->GetData(szData);
```

9. void SetPrivateData(unsigned short offset, long lPrivateData)；

功能：设置私有数据。

参数：offset，私有数据的偏置，0~3；lPrivateData，长整型私有数据。

返回值：无。

备注：Ioscan 自动为每个 CItem 实例分配了一块由 4 个整型数（32 位）组成的程序员私有数据空间供程序员使用。程序员除了用这部分私有数据区保存数值外，也可以分配新的内存空间，然后将内存指针保存在私有数据区内，但不要忘记，在程序退出前，正确释放新分配的内存空间。

举例：

```
char* pBuf = new char[MAXCMDLEN];
pItem->SetPrivateData(3, (long)pBuf);
```

10. long GetPrivateData(unsigned short offset)

功能：取得私有数据。

参数：offset，私有数据的偏置，0~3；lPrivateData，长整型私有数据。

返回值：整型私有数据。备注：Ioscan 自动为每个 CItem 实例分配了一块由 4 个整型数（32 位）组成的程序员私有数据空间供程序员使用。程序员除了用这部分私有数据区保存数值外，也可以分配新的内存空间，然后将内存指针保存在私有数据区内，但不要忘记，在程序退出前，正确释放新分配的内存空间。

11. CPacket* GetPacket()

功能：取得本连接项类所归属的数据包指针。

参数：无

返回值：本数据项所归属的数据包指针。

举例：CPacket* pPacket = pItem->GetPacket();

12. CDevice* GetDevice()

功能：取得本连接项所归属的设备指针。

参数：无

返回值：本连接项所归属的设备指针。

13. CManager* GetManager()

功能：取得 Ioscan 管理器指针。

参数：无

返回值：Ioscan 管理器指针。

14. void SetHisData(HisInsDatStru* pHisInsDatStru,int nCount)

功能：按浮点型格式设置历史数据

参数：无

返回值：无

Cpacket 类

CPacket 类提供了对数据包对象的一组操作方法。一个数据包对象包含一个或多个数据项对象。CPacket 类的声明如下：

```
class CPacket : public CObject
{
public:
    virtual int GetItemCount()=0; //取得包中数据连接项个数
    virtual int AddItem(CItem* pItem)=0; //按先后顺序加入数据连接项
    virtual void InsertItem(unsigned short nIndex, CItem* pItem)=0; //按索引插入连接项
    virtual CItem* GetItem(int nIndex)=0; //按索引取得数据连接项指针
    //设置包的可写属性，缺省时可写的，bAttribute 为 TRUE 设置为不可写。
    virtual void SetReadOnly(BOOL bAttribute = TRUE)=0;
    //设置包的可读属性，缺省时可读的，bAttribute 为 TRUE 设置为不可读。
    virtual void SetWriteOnly(BOOL bAttribute = TRUE)=0;
    //设置私有数据，offset 范围：0~15
    virtual void SetPrivateData(unsigned short offset, long lPrivateData)=0;
    virtual long GetPrivateData(unsigned short offset)=0;//取得私有数据，offset 范围 0~15};
    virtual void RepeatScan()=0; //重复扫描当前数据包
    virtual CDevice* GetDevice()=0; //取得本数据包所归属的设备指针
    virtual CManager* GetManager()=0; //取得 IoScan 管理器指针
};
```

下面详述 CPacket 类的各个函数：

1. int GetItemCount()

功能：取得本数据包内数据项的个数。

参数：无。

返回值：本数据包内数据项的个数。

举例：int nItemCnt = pPacket->GetItemCount();

2. int AddItem(CItem* pItem)；

功能：按先后顺序加入数据连接项。参数：加入数据包的数据项指针。

返回值：本数据包内数据项的个数。

举例：

```
if (pPacket->GetItemCount() == 0)
    pPacket->AddItem(pItem);
```

3. void InsertItem(CItem* pItem, unsigned short nIndex)

功能：按指定位置加入数据连接项。

参数：pItem，加入数据包的数据项指针。nIndex，加入的数据项在数据包内的序号（从 0 开始）。

返回值：本数据包内数据项的个数。

4. CItem* GetItem(int nIndex)；

功能：按序号取得数据连接项。

参数：nIndex，数据项在数据包内的序号（从 0 开始）。

返回值：数据项指针。

举例：

```
for (int i = 0; i < pPacket->GetItemCount(); i++)
{
    CItem* pItem = pPacket->GetItem(i);
    ......
}
```

5. . void SetReadOnly(BOOL bAttribute = TRUE)

功能：设置数据包的写属性，缺省时数据包是可写的。

参数：TRUE：设置为不可写；FALSE：设置为可写。

返回值：无。

举例：

```
int nPacketCnt = 0;
nPacketCnt = pDevice->GetPacketCount();
for (int i = 0; i < nPacketCnt; i++)
{
    CPacket* pPacket = pDevice->GetPacket(i);
    pPacket->SetReadOnly();
}
```

6. void SetWriteOnly(BOOL bAttribute = TRUE)

功能：设置数据包的读属性，缺省时数据包是可读的。

参数：TRUE：设置为不可读；FALSE：设置为可读。

返回值：无。

7. void SetPrivateData(unsigned short offset, long lPrivateData)

功能：设置私有数据。

参数：offset，私有数据的偏置，0~15；lPrivateData，长整型私有数据。

返回值：无。

备注：Ioscan 自动为每个 CPacket 实例分配了一块由 16 个整型数（32 位）组成的程序员私有数据空间供程序员使用。程序员除了用这部分私有数据区保存数值外，也可以分配新的内存空间，然后将内存指针保存在私有数据区内，但不要忘记，在程序退出前，正确释放新分配的内存空间。

举例：

```
char* pBuf = new char[MAXCMDLEN];
pPacket->SetPrivateData(3, (long)pBuf);
```

8. long GetPrivateData(unsigned short offset)

功能：取得私有数据。

参数：offset，私有数据的偏置，0~15；lPrivateData，长整型私有数据。

返回值：整型私有数据。

备注：Ioscan 自动为每个 CPacket 实例分配了一块由 16 个整型数（32 位）组成的程序员私有数据空间供程序员使用。程序员除了用这部分私有数据区保存数值外，也可以分配新的内存空间，然后将内存指针保存在私有数据区内，但不要忘记，在程序退出前，正确释放新分配的内存空间。

9. void RepeatScan()

功能：重复扫描当前数据包。

参数：无

返回值：无

备注：对于异步工作方式，在采集数据时，一个数据包的采集数据可能需要多帧传送，或者一次采集过程需要多次请求棗应答过程。在这种情况下，在解析数据时，执行该函数，Ioscan 将重复扫描当前数据包。

10. CDevice* GetDevice()；

功能：取得本数据包所归属的设备指针。

参数：无

返回值：本数据包所归属的设备指针。

举例：CDevice* pDevice = pPacket->GetDevice();

11. CManager* GetManager()

功能：取得 Ioscan 管理器指针。

参数：无

返回值：Ioscan 管理器指针。

举例：CManager* pManager = pPacket->GetManager();

Cdevice 类

CDevice 类提供了对设备对象进行操作的一组方法。一个设备包对象包含一个或多个数据包对象。下面是 CDevice 的定义：

```
class CDevice : public CObject
{
public:
    //取得设备名称
    virtual CString GetName()=0;
    virtual int GetType()=0;            //取得设备类型(如：串口类、网络类等)
```

```
    virtual int GetModel()=0;                  //取得设备型号
    virtual CString GetAddr()=0;        //取得设备地址
    virtual CString GetNetAddr()=0;          //取得 IP 地址或主机名称
    virtual CString GetPhoneNumber()=0;     //取得电话号码
    virtual int GetPort()=0;              //取得串口设备端口号或网络设备端口
    virtual int GetUpdateCycle()=0;          //取得数据更新周期常数(毫秒)
    virtual int GetTimeOut()=0;              //取得超时时间常数(毫秒)
    //取得通信空闲时间常数(毫秒),用于决定是否按字节发送和每个字节之间的延时
    virtual int GetIdleTime()=0;
    //设置通信空闲时间常数(毫秒),用于决定是否按字节发送和每个字节之间的延时
    virtual void SetIdleTime(unsigned short nTime)=0;
    virtual void GetDCB(DCB& dcb)=0;     //取得 DCB 参数
    virtual void SetDCB(DCB dcb) =0;     //设置 DCB 参数
    //设置向设备写入数据的结果状态
    virtual void SetWriteStatus(BOOL bSuccess = TRUE)=0;
    //设置私有数据，offset 范围：0~15
    virtual void SetPrivateData(unsigned short offset, long lPrivateData)=0;
    virtual long GetPrivateData(unsigned short offset)=0; //取得私有数据，offset 范围：0~15
    virtual void RepeatScan()=0; //重复扫描当前设备
    //按 LPCSTR 格式显示过程信息
    virtual void ShowProcessMessage(L PCSTR szMsg)=0;
    //按 CString 格式显示过程信息
    virtual void RecordProcessMessage(const CString& csMsg)=0;
    //按 LPCSTR 格式显示事件信息，bNotifyDB  决定是否提交给 DB
    virtual void ShowEventMessage(LPCSTR szEvent, BOOL bNotifyDB = FALSE)=0;
    //按 CString 格式显示事件信息，bNotifyDB  决定是否提交给 DB
    virtual void RecordEventMessage(const CString& csEvent)=0;
    virtual int GetPacketCount()=0; //取得设备包含的数据包个数
    //按索引取得数据包指针
    virtual CPacket* GetPacket(unsigned short nIndex)=0;
    virtual CManager* GetManager()=0; //取得 IoScan 管理器指针
    virtual int Send(char* chCommand, int nCommandLen)=0; //向设备写入数据
    virtual int GetItemCount()=0; //取得设备中数据连接项个数
    virtual CItem* GetItem(int nIndex)=0; //按索引取得数据连接项指针
    virtual void SetUnPublicized(BOOL bUnPublicized = TRUE) = 0; //设置协议为未公开
    virtual void SetDCBAfterSend(DCB dcb) =0; //设置 DCB 参数
    //设置接收缓冲区大小（字节）
    virtual int SetAcceptBufferLength(int nNewAcceptBufferLength = 1024) = 0;
    //设置发送缓冲区大小（字节）
    virtual int SetSendBufferLength(int nNewSendBufferLength = 1024) = 0;
    //设置一次从串口读取的字符串块大小（字节）
    virtual int SetReadDataBufferLength(int nNewReadDataBufferLength = 128) = 0;
};
```

下面详述 CDevice 类的各个函数：

1. CString GetName()

功能：取得设备的名称。

参数：无。

返回值：设备的名称。

备注：设备名即在“I/O 设备定义”对话框“设备名称”编辑框中输入的字符串。

举例：

CString csDeviceName = pDevice->GetName(););

2. int GetType()；

功能：取得设备类型(如：串口类、网络类等)。

参数：无。

返回值：设备类型。目前该函数返回值及含义包括如下几种：

_SYNCH，同步通信类型设备

_SERIAL_PORT ，串口通信类型设备

_NET，TCP/IP 网络通信类型设备

_MODEM，MODEM 通信类型设备

_BOARD，板卡类型设备

_PARELLEL，并口通信类型设备

对于同步通信类型设备、板卡类型设备，Ioscan 为同步工作方式。对于串口通信类型设备、网络通信类型设备、MODEM 通信类型设备、并口通信类型设备，扫描程序 Ioscan 为异步工作方式。

举例：int nDeviceType = pDevice-> GetType();

3. int GetModel()

功能：取得设备型号。

参数：无。

返回值：设备型号。

备注：设备型号在文件 iodesc.txt 中指定。

举例：int nDeviceModel = pDevice->GetModel();

4. CString GetAddr()

功能：取得设备地址。

参数：无。

返回值：设备地址。

备注：设备地址指在“I/O 设备定义”对话框中的“设备地址”编辑框中输入的内容，并不一定与物理设备地址相同。

举例：int nDeviceAddr = pDevice->GetAddr();

5. CString GetNetAddr()

功能：取得 TCP/IP 网络结点的地址（IP 地址或主机名称）。

参数：无。

返回值：TCP/IP 网络结点的地址（IP 地址或主机名称）。

备注：TCP/IP 网络结点的地址在设备组态时指定。

举例：CString csNetAddr = pDevice->GetNetAddr();

6. CString GetPhoneNumber()

功能：取得电话号码。

参数：无。

返回值：电话号码。

备注：设备的电话号码在设备组态时指定。

举例：CString csPhoneNumber = pDevice->GetPhoneNumber();

7. int GetPort()

功能：取得串口设备端口号或网络设备端口。

参数：无。

返回值：如果是串口类设备，该函数返回串口的端口号（1、2、3...）；如果是 TCP/IP 网络设备，则返回的是网络端口号。

备注：设备的串口端口号或 TCP/IP 网络端口号在设备组态时指定。

举例：int nPort = pDevice->GetPort();

8. int GetUpdateCycle()

功能：取得数据更新周期(毫秒)。

参数：无。

返回值：数据更新周期(毫秒)。

备注：数据更新周期在设备组态时指定。

举例：int nUpdateCycle = pDevice->GetUpdateCycle();

9. int GetTimeOut()

功能：取得设备超时时间（毫秒）。

参数：无。

返回值：设备超时时间（毫秒）。

备注：设备超时时间在设备组态时指定。

举例：int nTimeOut = pDevice->GetTimeOut();

10. int GetIdleTime()

功能：取得通信空闲时间(毫秒)。

参数：无。

返回值：通信空闲时间(毫秒)。

通信空闲时间由 SetIdleTime 函数设定，初始为 0。当通信空闲时间设置为 0 时，向 I/O 设备发送的每帧数据为一次性发送，若设置了通信空闲时间（大于 0），每帧数据按字节一个一个发送，每个字节之间让通信信道空闲等待，等待的时间长度则为通信空闲时间。

举例：int nIdleTime = pDevice->GetIdleTime();

11. void SetIdleTime(unsigned short nTime)

功能：设置通信空闲时间(毫秒)。

参数：通信空闲时间(毫秒)。

返回值：无

备注：

通信空闲时间由 SetIdleTime 函数设定，初始为 0。当通信空闲时间设置为 0 时，向 I/O 设备发送的每帧数据为一次性发送，若设置了通信空闲时间（大于 0），每帧数据按字节一个一个发送，每个字节之间让通信信道空闲等待，等待的时间长度则为通信空闲时间。

举例：pDevice->SetIdleTime(10);

12. void GetDCB(DCB& dcb)

功能：取得 DCB 参数。

参数：返回 DCB 参数的结构指针。

返回值：无

举例：

```
DCB dcb;
pDevice->GetDCB(dcb);
```

13. void SetDCB(DCB dcb)

功能：设置 DCB 参数。

参数：DCB 参数。

返回值：

备注：一般情况下，如果需要对系统缺省设置的 DCB 参数中的个别参数需要修改时，必须先通过 GetDCB()取得系统缺省设置的 DCB 参数，然后修改其中的参数后，再用本函数设置回去。

举例：

```
DCB dcb;
pDevice->GetDCB(dcb);
dcb.fBinary = TRUE;
dcb.fOutxCtsFlow = FALSE;
dcb.fOutxDsrFlow = FALSE;
dcb.fDtrControl = DTR_CONTROL_DISABLE;
dcb.fNull = FALSE;
dcb.fRtsControl = RTS_CONTROL_DISABLE;
pDevice->SetDCB(dcb);
```

14. void SetWriteStatus(BOOL bSuccess = TRUE)

功能：设置向设备写入数据的结果状态。

参数：

TRUE:写成功

FALSE:写失败

返回值：无

备注：在异步方式下，向设备发送下置数据命令，并且需要设备应答以明确下置过程是否成功时，在扫描函数 OnParseResponse()中必须调用本函数明确通知 Ioscan 下置过程是否成功，在调用本函数时同时相当于通知 Ioscan 本次下置过程结束（无论成功与否）。如果程序员没有正确调用本函数，在下置数据时，可能会导致 Ioscan 无限制地循环在数据下置过程。

举例：

```
if(byResponse[3] == 0x6b && byResponse[4] == 0x80)
   pDevice->SetWriteStatus();
else
   pDevice->SetWriteStatus(FALSE);
```

15. void SetPrivateData(unsigned short offset, long lPrivateData)

功能：设置私有数据。

参数：offset，私有数据的偏置，0~15；lPrivateData，长整型私有数据。

返回值：无。

备注：Ioscan 自动为每个 CDevice 实例分配了一块由 16 个整型数（32 位）组成的程序员私有数据空间供程序员使用。程序员除了用这部分私有数据区保存数值外，也可以分配新的内存空间，然后将内存指针保存在私有数据区内，但不要忘记，在程序退出前，正确释放新分配的内存空间。

举例：

```
char* pBuf = new char[MAXCMDLEN];
pDevice->SetPrivateData(3, (long)pBuf);
```

16. long GetPrivateData(unsigned short offset)

功能：取得私有数据。

参数：offset，私有数据的偏置，0~15；lPrivateData，长整型私有数据。

返回值：整型私有数据。

备注：Ioscan 自动为每个 CDevice 实例分配了一块由 16 个整型数（32 位）组成的程序员私有数据空间供程序员使用。程序员除了用这部分私有数据区保存数值外，也可以分配新的内存空间，然后将内存指针保存在私有数据区内，但不要忘记，在程序退出前，正确释放新分配的内存空间。

17. void RepeatScan()

功能：重复扫描当前设备。

参数：无

返回值：无

备注：对于异步工作方式，在创建与 I/O 设备的初始链接过程时，如果协商过程需要多次请求禀应答过程，在解析数据时，执行该函数，Ioscan 将重复扫描当前设备。

18. void ShowProcessMessage(LPCSTR szMsg)

功能：按 LPCSTR 格式显示过程信息

参数：LPCSTR 格式字符串

返回值：无

备注：调用本函数后，szMsg 的内容将被显示在 Ioscan 的过程显示窗口上。

举例：

```
char szMsg[100];
sprintf(szMsg, "Hello!");
pDevice->ShowProcessMessage(szMsg);
```

19. void ShowProcessMessage(const CString& csMsg)

功能：按 CString 格式显示过程信息。

参数：CString 格式信息

返回值：无

备注：调用本函数后，csMsg 的内容将被显示在 Ioscan 的过程显示窗口上。

20. void ShowEventMessage(LPCSTR szEvent, BOOL bNotifyDB = FALSE)

功能：按 LPCSTR 格式显示事件信息

参数：szEvent:LPCSTR 格式字符串。bNotifyDB:为 TRUE 时报告给 DB，否则不报告。

返回值：无

备注：调用本函数后，szEvent 的内容将被显示在 Ioscan 的事件信息对话条的列表框中。如果将该事件通知给 DB，该信息会在 View 运行时显示在系统报警窗口上。

举例：

```
char szMsg[100];
sprintf(szMsg, "Hello!");
pDevice->ShowEventMessage(szMsg);
```

21. void ShowEventMessage(const CString& csEvent, BOOL bNotifyDB = FALSE)

功能：按 CString 格式显示事件信息。

参数：csEvent:LPCSTR 格式字符串。bNotifyDB:为 TRUE 时报告给 DB，否则不报告。

返回值：无

备注：调用本函数后，csMsg 的内容将被显示在 Ioscan 的过程显示窗口上。如果将该事件通知给 DB，该信息会在 View 运行时显示在系统报警窗口上。

举例：void RecordEventMessage(LPCSTR szEvent)

22. int GetPacketCount()

功能：取得本设备内数据包的个数。

参数：无

返回值：本设备内数据包的个数。

举例：

```
int nPacketCnt = 0;
nPacketCnt = pDevice->GetPacketCount();
```

23. CPacket* GetPacket(unsigned short nIndex)

功能：按索引取得数据包。

参数：从 0 开始的索引号

返回值：数据包指针。

举例：

```
nPacketCnt = pDevice->GetPacketCount();
for (int i = 0; i < nPacketCnt; i++)
{
    CPacket* pPacket = pDevice->GetPacket(i);
    ......
}
```

24. CManager* GetManager()

功能：取得 Ioscan 管理器指针。

参数：无

返回值：Ioscan 管理器指针。

举例：CManager* pManager = pDevice->GetManager();

25. int Send(char* chCommand, int nCommandLen)

功能：直接发送命令字符串，可能通过串口，MODEM，网络等...

参数：chCommand:　命令字符串指针。nCommandLen:　命令字符串长度。

返回值：成功：TRUE;失败：FALSE。

26. int GetItemCount()

功能：取得设备上数据连接项的数量

参数：无

返回值：设备的所有连接项数量

备注：一般用来直接在设备上遍历连接项。

27. CItem* GetItem(int nIndex)

功能：按索引得到设备上的连接项。

参数：索引（从 0 开始）

返回值：连接项指针。

28. void SetUnPublicized(BOOL bUnPublicized = TRUE)

功能：设置设备的协议为未公开的。

参数：无

返回值：无

备注：如果设备的协议不设置为未公开的，可以在调度程序里通过“系统参数”对话框，设置“显示十六进制裸数据”或“显示混合裸数据”而直接显示通讯过程数据。设置为不公开协议的设备永远不能直接显示通讯过程数据。

29. void SetDCBAfterSend(DCB dcb)

功能：在发送数据后设置 DCB 参数为 dcb,某设备专用，一般不要理会这个函数。

参数：无

返回值：无

30. int SetAcceptBufferLength(int nNewAcceptBufferLength = 1024)

功能：设置接收缓冲区的大小，建议使用这个参数设置接收缓冲区为合适的值，以节省内存开销。

参数：新的接收缓冲区大小。

返回值：新的接收缓冲区大小。

31. int SetSendBufferLength(int nNewSendBufferLength = 1024)

功能：设置发送缓冲区大小。

参数：新的发送缓冲区大小。

返回值：新的发送缓冲区大小。

32. int SetReadDataBufferLength(int nNewReadDataBufferLength = 128)

功能：设置一次从串口读取的字符串块大小。

参数：新的一次从串口读取的字符串块大小。

返回值：新的一次从串口读取的字符串块大小。

Cmanager 类

CManager 类提供了对管理器对象进行操作的一组方法。一个管理器对象管理一个或多个设备对象。下面是 CManager 类的定义：

```
class CManager : public CObject
{
public:
    virtual void SetTitle(const CString& csTitle)=0;          //设置 IOSCAN 程序标题
    virtual CString GetModuleName()=0;                        //取得 IOSCAN 执行文件名称，
    virtual CString GetAppPath()=0;                           //取得当前工程应用的安装路径
    virtual CString GetInstallPath()=0;                       //取得 I/O 驱动的安装路径
    virtual int GetDeviceCount()=0;                           //取得设备个数
    virtual CDevice* GetDevice(unsigned short nIndex)=0;     //按索引取得设备指针
    virtual void SetSynch(BOOL bSynch = TRUE)=0;
    virtual void SetSlave(BOOL bSlave = TRUE)=0;              //设置 Ioscan 为从机方式
    //设置私有数据，offset 范围：0~15
    virtual void SetPrivateData(unsigned short offset, long lPrivateData)=0;
    //取得私有数据，offset 范围：0~15};
    virtual long GetPrivateData(unsigned short offset)=0;
    //设置是否进行超时处理，缺省如果不止一个设备，则进行超时处理。
    virtual void SetDoNothingWhenTimeOut(BOOL bDoNothing = TRUE) = 0;
    //是否需要动态分包，缺省不动态分包
    virtual BOOL SetDynamicAllocatePacket(BOOL bDynamicAllocatePacket = TRUE) = 0;
};
```

下面详述 CManager 类的各个函数：

1. void SetTitle(const CString& csTitle)

功能：设置 Ioscan 程序标题

参数：Ioscan 程序标题

返回值：无

举例：pManager->SetTitle("SMAR CD600 智能调节器");

2. CString GetModuleName()

功能：取得 Ioscan 的执行文件名称，即 IOID

参数：无

返回值：Ioscan 的执行文件名称，即 IOID

举例：CString csIoid = pManager->GetModuleName();

3. CString GetAppPath()

功能：取得当前工程应用的安装路径。

参数：无

返回值：当前工程应用的安装路径。

备注：

也就是在系统安装路径下存放该 I/O 驱动程序相关系统文件的子目录名称，以及在用户工程应用路径下存放该 I/O 驱动程序相关配置文件的子目录名称.

举例：CString csAppPath = pManager->GetAppPath();

4. CString GetInstallPath()

功能：取得 I/O 驱动的安装路径。

参数：无

返回值：I/O 驱动的安装路径。

5. int GetDeviceCount()

功能：取得设备个数。

参数：无

返回值：取得设备个数。

举例：

```
int nDeviceCnt = 0;
nDeviceCnt = pManager->GetDeviceCount();
```

6. CDevice* GetDevice(unsigned short nIndex)

功能：按索引取得设备指针。

参数：索引号（从 0 开始）

返回值：设备指针。

7. void SetSlave(BOOL bSlave = TRUE)

功能：设置 Ioscan 为从机方式。

参数：

TRUE:从机方式。

FALSE:主机方式。

返回值：无

备注：缺省为主机方式。

8. void SetPrivateData(unsigned short offset, long lPrivateData)

功能：设置私有数据。

参数：

offset，私有数据的偏置，0~15；

lPrivateData，长整型私有数据。

返回值：无。

备注：Ioscan 自动为 CManager 实例分配了一块由 16 个整型数（32 位）组成的程序员私有数据空间供程序员使用。程序员除了用这部分私有数据区保存数值外，也可以分配新的内存空间，然后将内存指针保存在私有数据区内，但不要忘记，在程序退出前，正确释放新分配的内存空间。

举例：

char* pBuf = new char[MAXCMDLEN];

pManager->SetPrivateData(3, (long)pBuf);

9. long GetPrivateData(unsigned short offset)

功能：取得私有数据。

参数：offset，私有数据的偏置，0~15；lPrivateData，长整型私有数据。

返回值：整型私有数据。

备注：Ioscan 自动为每个 CMnager 实例分配了一块由 16 个整型数（32 位）组成的程序员私有数据空间供程序员使用。程序员除了用这部分私有数据区保存数值外，也可以分配新的内存空间，然后将内存指针保存在私有数据区内，但不要忘记，在程序退出前，正确释放新分配的内存空间。

10. void SetDoNothingWhenTimeOut(BOOL bDoNothing = TRUE)

功能：超时时是否进行既定的处理。

参数：TRUE:不进行超时处理;FALSE:进行超时处理;

返回值：无

备注：缺省进行超时处理。

11. BOOL SetDynamicAllocatePacket(BOOL bDynamicAllocatePacket = TRUE)

功能：是否需要动态分包。

参数：

TRUE:设置为动态分包。

FALSE:不动态分包。

返回值：原来是否动态分包。

备注：缺省不动态分包。

扫描函数

扫描函数是程序员必须处理的部分。Ioapi 具体规定了扫描函数的名称、参数形式及返回类型。程序员根据具体 I/O 设备及通信协议编写函数内容。下面是 IOAPI.DLL 中扫描函数的声明（这些函数不必全部实现，没有实现的函数不调用，也不出错）

```
//*****************************************************************
// 扫描函数定义
//*****************************************************************
extern "C" AFX_EXT_API void OnCreate(CManager* pManager);
extern "C" AFX_EXT_API void OnLoadDevice(CManager* pManager);
extern "C" AFX_EXT_API void OnSortItem(CDevice* pDevice, int& nKey1, int& nKey2);
extern "C" AFX_EXT_API void OnItemToPacket(CPacket* pPacket, CItem* pItem);
extern "C" AFX_EXT_API void OnEndPacketLoad(CDevice* pDevice);
extern "C" AFX_EXT_API long OnCreateDeviceLink(CDevice* pDevice, char* szCommand, int& nCmdLen);
extern "C" AFX_EXT_API long OnCreatePacketLink(CPacket* pPacket, char* szCommand, int& nCmdLen);
```

```
extern "C" AFX_EXT_API long OnReadData(CPacket* pPacket, char* szCommand, int& nCmdLen);
extern "C" AFX_EXT_API long OnWriteData(CItem* pItem, LPCSTR szWriteData, char* szCommand, int& nCmdLen);
extern "C" AFX_EXT_API long OnIsResponseComplete(CDevice* pDevice, CPacket* pPacket, char* szResponse, int nResponLen, int& nStart, int& nLen);
extern "C" AFX_EXT_API void OnParseResponse(CDevice* pDevice, CPacket* pPacket, char* szResponse, int nResponLen);
extern "C" AFX_EXT_API long OnTimeout(CDevice* pDevice, CPacket* pPacket, CItem* pItem, char* szCommand, int& nCmdLen);
extern "C" AFX_EXT_API void OnUnloadDevice(CManager* pManager);
extern "C" AFX_EXT_API void OnClose(CManager* pManager);
extern "C" AFX_EXT_API void OnUnloadPacket(CPacket* pPacket);
```

1. void OnCreate(CManager* pManager)

当 IOSCAN 载入 IOAPI.DLL 后，立刻调用该函数。此时 IOSCAN 还没有活动设备列表，所以不能在这里调用关于设备，包，连接项等的任何函数。一般在这里设置调度程序的标题，设置进行动态分包，是否进行超时处理等。

举例：

```
void OnCreate(CManager* pManager)
{
    pManager->SetTitle("SMAR CD600 智能调节器");
}
```

2. void OnLoadDevice(CManager* pManager)

IOSCAN 形成设备列表后，立刻调用该函数，这时还取得数据连接项列表，更没有形成包，所以不要试图遍历包或数据连接项。

举例：

```
void OnLoadDevice(CManager* pManager)
{
    int nDevCnt = pManager->GetDeviceCount();
    for (int i = 0; i < nDevCnt; i ++)
    {
        CDevice* pDevice = pManager->GetDevice(i);
        DCB dcb;
        pDevice->GetDCB(dcb);
        dcb.fBinary = TRUE;
        dcb.fOutxCtsFlow = FALSE;
        dcb.fOutxDsrFlow = FALSE;
        dcb.fDtrControl = DTR_CONTROL_DISABLE;
        dcb.fNull = FALSE;
        dcb.fRtsControl = RTS_CONTROL_DISABLE;
        pDevice->SetDCB(dcb);
    }
}
```

3. void OnSortItem(CDevice* pDevice, int& nKey1, int& nKey2)

IOSCAN 得到连接项列表后，调用该函数，在该函数内部程序员可以给 nKey1(第一排序关键字),nKey2(第二排序关键字)赋值，程序返回后，IOSCAN 将根据 nKey1 和 nKey2 对所以连接项进行排序。（nKey1 和 nKey2 是连接项结构 n 数组的索引，有效值是 0~7）实际上我们一般不需要排序，这样的话我们可以在该函数内不写任何代码，或根本不实现该函数。

```
void OnSortItem(CDevice* pDevice, int& nKey1, int& nKey2)
{
   nKey1 = 4;(使用 n[4]进行排序。)
}
```

4. void OnItemToPacket(CPacket* pPacket, CItem* pItem);

IOSCAN 完成所以连接项的排序后，随即调用该函数，将所有连接项打入包中。这个函数同时也会遍历所有的连接项，所有可以用来对连接项设置各种标志。如果没有一个连接项被加入包中，则每个连接项形成一个包。

例子：

```
void OnItemToPacket(CPacket* pPacket, CItem* pItem)
{
   if (pPacket->GetItemCount() == 0)
      pPacket->AddItem(pItem);
   else
   {
      IOITEMDEF * pPackHeadItemStru = pPacket->GetItem(0)->GetItemStru();
      IOITEMDEF * pItemStru = pItem->GetItemStru();
      if (pItemStru->n[0] - pPackHeadItemStru->n[0] > 128)
         return;
      if (pItemStru->n[0] == pPackHeadItemStru->n[0] &&
         pItemStru->n[1] == pPackHeadItemStru->n[1] &&
         pItemStru->n[2] == pPackHeadItemStru->n[2] &&
         pItemStru->n[3] == pPackHeadItemStru->n[3])
         pPacket->AddItem(pItem);
   }
}
```

5. void OnEndPacketLoad(CDevice* pDevice);

当所有的包形成之后，调用一次该该函数。

例子：

```
void OnEndPacketLoad(CDevice* pDevice)
{
   int nPacketCnt = 0;
   nPacketCnt = pDevice->GetPacketCount();
   for (int i = 0; i < nPacketCnt; i++)
   {
      CPacket* pPacket = pDevice->GetPacket(i);
      char* pBuf = new char[MAXCMDLEN];
```

```
            pPacket->SetPrivateData(3, (long)pBuf);
        }
    }
```

6. void OnUnloadPacket(CPacket* pPacket)

当一个包即将失效时调用该函数，应该在这里释放动态分配的资源（包上的或包中连接项上的。）

7. void OnUnloadDevice(CManager* pManager)

当所有设备即将失效时调用该函数，应该在这里释放动态分配的资源。

8. void OnClose(CManager* pManager)

程序退出前调用该函数，调用它时所有设备已经失效。

9. long OnCreateDeviceLink(CDevice* pDevice, char* szCommand, int& nCmdLen)

IOSCAN 在进而周期性扫描后首先一次或多次扫描该函数，使用 RepeatScan 函数可以重复调用一个设备。

本函数用于完成与 I/O 设备建立链接、会话、初始化通信等过程，对于每个设备均要扫描本函数。在异步方式下，程序根据要建立链接的设备，形成正确的链接请求命令，由 Ioscan 将命令传送给设备，然后在扫描函数 OnIsResponseComplete()和 OnParseResponse()处理收到的设备应答信息。在同步方式下，在本函数中要完成全部链接的处理，并且不再扫描扫描函数 OnIsResponseComplete()和 OnParseResponse()。

返回值：无

用于异步方式的返回类型：

COMMAND_WAIT_FOR_RESPONSE//向设备发送命令，等待设备应答（需要在 OnIsResponseComplete()和 OnParseResponse()中处理）

COMMAND_NO_RESPONSE//向设备发送命令，设备没有应答或忽略设备应答（发送后即完成）。

NO_COMMAND_WAIT_FOR_RESPONSE//不向设备发送命令，等待设备主动应答（需要在 OnIsResponseComplete()和 OnParseResponse()中处理）

SKIP//直接跳过当前设备，调度下一个设备。

用于同步方式的返回类型：

NO_COMMAND_NO_RESPONSE//调度完该设备后，在下一个调度周期中调度下一个设备。

SKIP//直接跳过当前设备，调度下一个设备。

参数：

pDevice，设备指针

chCommand，在异步方式下，发送命令字符串指针。在同步方式下，该参数无效。

nCmdLen，在异步方式下，发送命令字符串的长度。在同步方式下，该参数无效。

10. long OnCreatePacketLink(CPacket* pPacket, char* szCommand, int& nCmdLen)

IOSCAN 对所有设备完成 OnCreateDeviceLink 的调度后调度该程序，每个包将被调度一次或多次，取决于是否调用 RepeatScan 函数。

本函数用于完成各个数据包与 I/O 设备建立链接、会话、初始化通信等过程，对于每个数据包均要扫描本函数。

参见(OnCreateDeviceLink)。

11. OnReadData(CPacket* pPacket, char* szCommand, int& nCmdLen);

IOSCAN 周期性调度该程序

参见(OnCreateDeviceLink)。

12. long OnWriteData(CItem* pItem, LPCSTR szWriteData, char* szCommand, int& nCmdLen)

当有数据下置时调用该程序。

参数：

pItem，数据项指针

szWriteData，下置数据（字符串格式）

chCommand，在异步方式下，发送命令字符串指针。在同步方式下，该参数无效。

nCmdLen，在异步方式下，发送命令字符串的长度。在同步方式下，该参数无效。

当数据下置时，必须调用 SetWriteStatus 函数，同步方式直接在 OnWriteData 函数中调用，异步方式在 OnParseResponse 中调用。否则将导致无限次的写同一个连接项。

13. long OnIsResponseComplete(CDevice* pDevice, CPacket* pPacket, char* szResponse, int nResponLen, int& nStart, int& nLen)

异步方式下当以等待应答的方式调用了 OnCreateDeviceLink，OnCreatePacketLink，OnReadData，OnWriteData 后，如果 IOSCAN 收到异步数据，将调用该函数询问 IOAPI.DLL 是否收到了完整的包。

IOAPI.DLL 需要在这里判断是否形成完整的包，如果形成完整包，则 nStart 代表有效数据的起始偏移量，nLen 代表数据串的长度，并且应该返回 COMPLETE.

返回值：

如果收到的设备应答信息完整、正确，返回 COMPLETE；如果收到的设备应答信息不完整，返回 INCOMPLETE（Ioscan 将继续等待接收剩余信息）；如果收到的设备应答信息完整，但内容发生错误（如：未经过校验）返回 BAD_DATA（建议不使用 BAD_DATA 返回值，而是在 OnParseResponse 函数中处理错误信息的显示）。

注：如果返回 INCOMPLETE 时将 nLen 置为-2,将导致 nStart 之前的数据被截除。

参数：

pDevice，设备指针

pPacket，数据包指针

chResponse，接收到的应答信息字符串指针

nResponLen，接收到的应答信息字符串的长度

nStart，接收到的应答信息中正确、有效部分的开始位置（从 0 开始）

nLen，接收到的应答信息中正确、有效部分的长度

14. void OnParseResponse(CDevice* pDevice, CPacket* pPacket, char* szResponse, int nResponLen)

如果 IOAPI.DLL 在 OnIsResponseComplete 时返回 COMPLETE，将调用该函数，主要用来设置写状态，向数据库下设数据等。

参数：

pDevice，设备指针

pPacket，数据包指针

chResponse，接收到的完整、正确应答信息的字符串指针

nResponLen，接收到的完整、正确应答信息字符串的长度

例子：

```
void OnParseResponse(CDevice* pDevice, CPacket* pPacket, char* chResponse, int nResponLen)
{
    long nRorW = pPacket->GetPrivateData(0);
```

```
        if (nRorW == _READ)
        {
            CItem* pItem = pPacket->GetItem(0);
            //因为这里一个包中只有一个点所以处理非常简单
            switch(chResponse[2])
            {
            case 1: //DIO
                pItem->SetData(long(chResponse[4]));
                break;
            case 3: //AIO
                {
                    long ldata;
                    ldata = BYTE(chResponse[4]);
                    ldata = (ldata<<8)|BYTE(chResponse[5]);
                    pItem->SetData(ldata);
                }
                break;
            }
            CString csMsg;
            csMsg.Format("本数据包采集成功!");
            pDevice->ShowProcessMessage(csMsg);
        }
    }
```

15. long OnTimeout(CDevice* pDevice, CPacket* pPacket, CItem* pItem, char* szCommand, int& nCmdLen);

当异步扫描发生超时时调用该函数，可以在这里设置各种标志，状态等。

参数：

pDevice,设备指针

pPacket,包指针

pItem,连接项指针

szCommand,没有意义

nCmdLen，没有意义