

PCI2310

Win95/98/NT/2000 驱动程序使用说明书

请您务必阅读《[使用纲要](#)》，他会使您事半功倍！

目 录

第一章	版权信息
第二章	绪 论
	第一节 使用纲要
	第二节 驱动程序功能概述
第三章	PCI 即插即用设备驱动程序安装
	第一节 Windows95、98、Me 环境下 PCI 设备驱动程序安装
	第二节 WindowsNT 环境下 PCI 设备及驱动程序安装
	第三节 Windows2000 环境下 PCI 设备及驱动程序安装
	第四节 PCI 接口程序及测试、示范程序的安装
	第五节 PCI 设备软件测试系统的介绍
	第六节 本驱动程序软件的关键文件
第四章	PCI 即插即用设备操作函数接口介绍
	第一节 接口函数列表
	第二节 设备对象管理函数原型说明
	第三节 简易的数字 IO 输入输出开关量操作函数原型说明
	第四节 硬件中断处理函数
	第五节 PCI 内存映射寄存器操作函数原型说明
第五章	共用函数介绍
	第一节 公用接口函数列表
	第二节 公用接口函数原型说明
	第三节 其他函数
第六章	硬件参数结构
	第一节 AD 硬件参数结构 (PCI2310_PARA_AD)
	第二节 用于数字 I/O 输出参数 (PCI2310_PARA_DO)
	第三节 用于数字 I/O 输入参数 (PCI2310_PARA_DI)
第七章	数据转换与排列规则
	第一节 如何将 AD 原始数据 LSB 转换电压值 Volt
	第二节 关于采集函数的 ADBuffer 缓冲区中的数据排放规则
	第三节 关于测试应用程序创建并形成的数据文件格式
第八章	上层用户函数接口应用实例
	第一节 怎样使用 ReadDeviceProAD_NotEmpty 函数直接取得 AD 数据
	第二节 怎样使用 ReadDeviceProAD_Half 函数直接取得 AD 数据
	第三节 怎样使用 ReadDeviceIntAD 函数直接取得 AD 数据
	第四节 怎样使用 SetDeviceDO 函数进行更便捷的数字开关量输出操作
	第五节 怎样使用 GetDeviceDI 函数进行更便捷的数字开关量输入操作
第九章	底层用户函数接口应用实例
	第一节 怎样使用映射寄存器读写函数直接编写数据采集程序？
	第二节 怎样使用映射寄存器读写函数直接编写开关量输入输出程序？
第十章	高速大容量、连续不间断数据采集及存盘技术详解
	第一节 使用程序查询方式实现该功能
	第二节 使用中断方式实现该功能
	附录 A LabView/CVI 图形语言专述
第一章	图形化编程语言 LabVIEW 环境及其开放性
第二章	LabView 驱动程序接口
	第一节 内嵌式驱动程序介绍

第二节 [内嵌式驱动器的原型说明](#)

第三节 [如何使用我公司的现有的驱动直接创建外挂式设备驱动器](#)

第四节 [如何使用我公司为用户已定制好的外挂式驱动器](#)

第五节 [如何在 LabView 中用上层函数实现 AD 采集](#)

第六节 [怎样用上层函数实现开关量输入输出操作](#)

第一章 版权信息

本软件产品及相关套件均属北京市阿尔泰科贸有限公司所有, 其产权受国家法律绝对保护, 除非本公司书面允许, 其他公司、单位、我公司授权的代理商及个人不得非法使用和拷贝, 否则将受到国家法律的严厉制裁。您若需要我公司产品及相关信息请及时与我们联系, 我们将热情接待。

第二章 绪 论

第一节、使用纲要

一、使用上层用户函数, 高效、简单

如果您只关心通道及频率等基本参数, 而不必了解复杂的硬件知识和控制细节, 便可如能所需, 那么我们强烈建议您使用上层用户函数, 它们就是几个简单的形如 Win32 API 的函数, 具有相当的灵活性、可靠性和高效性。诸如 SetDeviceDO、GetDeviceDI 等。而底层用户函数如 WritePortWord、ReadPortWord、WritePortByte、ReadPortByte……则是满足了解硬件知识和控制细节、且又需要特殊复杂控制的用户。但不管怎样, 我们强烈建议您使用上层函数(在这些函数中, 您见不到任何设备地址、寄存器端口、中断号等物理信息, 其复杂的控制细节完全封装在上层用户函数中。)对于上层用户函数的使用, 您基本上可以必参考硬件说明书, 除非您需要知道板上 D 型插座等管脚分配情况。

二、如何管理 PCI 设备

由于我们的驱动程序采用面向对象编程, 所以要使用设备的一切功能, 则必须首先用 CreateDevice 函数创建一个设备对象句柄 hDevice, 有了这个句柄, 您就拥有了对该设备的绝对控制权。然后将此句柄作为参数传递给其他函数, 如 SetDeviceDO 可以使用 hDevice 句柄以开关量的输出, 最后可以通过 ReleaseDevice 将 hDevice 释放掉。

三、如何实现开关量的简便操作

当您有了 hDevice 设备对象句柄后, 便可用 SetDeviceDO 函数实现开关量的输出操作, 其各路开关量的输出状态由其 pPara 中的成员变量 DO0-DO31 决定。由 GetDeviceDI 函数实现开关量的输入操作, 其各路开关量的输入状态由其 pPara 中的成员变量 DI0-DI31 决定。

四、如何实现 DA 的简便输出

当您有了 hDevice 设备对象句柄后, 首先用 InitDevProDA 函数实现 DA 的复位操作, 然后反复调用 WriteDevProDA 函数输出每一个 DA 数据。

五、哪些函数对您不是必须的?

当公共函数如 CreateFileObject, WriteFile, ReadFile 等一般来说都是辅助性函数, 除非您要使用存盘功能。如果您使用上层用户函数访问设备, 那么 WritePortByte, WritePortWord, WritePortULong, ReadPortByte, ReadPortWord, ReadPortULong 则对 PCI 用户来说, 可以说完全是辅助性, 它们只是对我公司驱动程序的一种功能补充, 对用户额外提供的, 它们可以帮助您在 NT、Win2000 等操作系统中实现对您原有传统设备如 ISA 卡、串口卡、并口卡的访问, 而没有这些函数, 您可能在新操作系统中无法继续使用您原有的老设备(除非您自己愿意去编写复杂的硬件驱动程序)。

第二节 驱动程序功能概述(不具体针某一种产品)

一、数据传输采集方式

我公司提供的驱动程序完全支持 *程序查询方式、硬件中断方式、直接内存存取 DMA* 方式。您从我公司所购买的硬件产品能支持的数据采集方式, 我们的驱动程序均予以满足。

二、数据传输与数据处理的独立性

为了提高数据吞吐率以及实现实时数据处理(如随时取数、随时暂停设备、随时开始传输、随时存盘、随时显示波形、随时设备控制输出等功能), 我们采用一种最新、最灵活的设计思想, 即数据采集传输和数据处理相独立的思想。即用我们所创建的设备对象在 Windows 系统空间管理一个一级强制性缓冲队列, 该缓冲队列可支持 128K 字(即 256K 字节)的系统内存空间 Buffer, 该队列采用先进先出策略和动态链表等技术来更高效地管理这个 Buffer。这个队列缓冲与用户数据缓冲区相独立, 设备对象在后台负责数据采集和传输, 将其数据映射到相应的队列缓冲单元, 且维持一个动态链表, 并向用户发送相应的通知消息。而用户则不必知道内部的任何复杂操作, 而只须在这个消息到来时, 使用 [ReadDeviceIntAD](#) 函数读一批 AD 数据或几批即可。重要的是, 在这个消息没有到来时, 用户代码不必

花任何 CPU 时间去轮询等待, 而用户正好利用这段空闲时间去处理更多的任务。即轻松实现了数据采集与数据处理的同步并发进行。这将是最高效的。这个队列缓冲跟先进先出存储器 FIFO 芯片功能基本一致, 只不过这个缓冲是一个被软件仿真的 FIFO 存储器。使用这项技术的最大优点就是完全解决了在多任务环境中实现高速连续采集数据难的问题。特别是整个系统突然繁忙的时候, 比如用户在高速采集数据或实时存盘时, 偶而移动窗口或改变窗口大小或弹出对话框时, 这项技术足以保证所采集的数据完整无缺。如果用户希望应用程序有更好的处理能力和克服操作系统的陡然忙碌对连续数据采集的影响, 可以考虑在用户模式中再使用二级缓冲队列和相应的缓冲区链表技术。具体细节请参考 NT 下的中断演示程序。(目前在 Window NT 中完全支持此项技术, 在以后的 Win2000 和 WinXP 版本中应该会进一步提供)。

三、连续不间断大容量采集存盘

在虚拟仪器、实验室数据分析、医疗设备、记录仪等诸多研究和应用领域中, 对数据的要求很高, 一方面数据容量较大, 如几百兆甚至几千兆, 另一方面采样速度都较高, 如 200KHz, 300KHz 等, 更重要是要求在高速长时间的采集数据过程中, 不能丢掉一个点, 必须全部存入硬盘, 同时还要进行一些点的抽样分析, 这在 DOS 环境中实现起来就有较大的难度, 就更别说在 Windows 这样的多任务环境中(对于 Windows 多任务机制请参阅有关 Windows 手册)。大家知道 Windows 的各应用程序总是不断地被任务调度器调度, 循环处在睡眠、排队、就绪、触发运行等状态中。Win95 任务之间的切换密度至少大于 1 毫秒, 那么如果要以 300KHz 频率采样(即每 3.3 微秒就得传输一个数据), 很显然有大量的数据在传输中由于任务之间的切换而被丢失掉。这就是基于 Windows 客户程序在传统模式下, 高速连续采集传输数据时所具有的局限性。为了突破这种局限性, 就得采用别的办法, 如非客户程序、内核程序、驱动程序(如 VxD、微代码)等, 再加上我们所掌握的新技术, 如内存映射、直接写盘技术以及独有的设计思想便可以很好的解决这些问题。从 1998 年 9 月开始, 已有部分用户实际使用, 反映良好。我们自己也经过全面测试, 比如在 Windows95 下使用无 FIFO 芯片的 BH5104 模板, 实际结果是: 以 200KHz 频率, 双通道采集正弦波且存盘, 写满整个硬盘近 4000 兆数据, 其时间长达 6 个小时左右, 随后再读盘回放磁盘数据, 整个波形没有发现任何串道、断点和畸形状。当然 PCI2005 等 PCI 设备同样具这样的性能。它不仅具有一级硬件缓冲 FIFO(其缓冲深度可调 1KB、2KB、4KB、8KB、16KB 等), 同样具有第二节中叙述的二级强制队列缓冲, 这个软件仿真的缓冲比一级缓冲要大几十倍。如果用户需要的话, 可以在应用程序中再建立循环式用户缓冲, 即可实现高速不间断大容量采集存盘功能。

四、后台工作方式

我们的驱动程序为用户提供了后台工作方式进行数据传输, 这样可以保证您的前台应用程序能实时高效的进行数据处理。后台方式的特点是在进行数据采集和传输过程中不占用客户程序的任何时间, 当采集的数据长度达到客户指定的值时便触发客户事件, 客户程序接受该事件便开始进行数据处理。在数据处理的同时, 驱动程序依然在进行下一批数据的传输, 即实现了并行操作, 极大的提高了数据的吞吐量和计算机系统的整体处理能力。

五、与设备无关性

通过总结各数据采集卡的的共同特点, 设计了基本一致的接口方式, 可以让您的应用程序不仅能适应您所购买的我公司第一种产品, 同时也能不经修改地适应我公司的其他同类产品(只有极少数设备需要极少的修改, 其修改的比例基本不超过 5%)。所以可以保证您的应用程序在我们的硬件产品基础上极为容易地进行功能和应用扩展, 节省您的大部分软件投资, 极大的缩短工程开发周期。

六、驱动程序的坚固性

我们的驱动程序都是经过严密彻底的测试和验证, 并经部分用户试用之后, 确认没有任何问题后才予以正式发行的, 所以当您使用起来应该有十足的安全感。

七、驱动程序特点

由于我们的驱动程序均采用动态虚拟技术(Windows 95), 微内核代码(Windows NT)因此可动态装载和卸载, 而且可以重入, 即可实现多道任务同时访问硬件设备的功能。这样可以保证您的软硬件资源可以被充分有效的利用。特别是在 Windows NT 下, 采用队列突发机制, 可以实现几十道线程程序同时访问一设备的功能。

八、高效与灵活兼备

如果您只是应用系统的上层用户, 您多半不愿意了解 PCI 硬件设备的各种复杂控制和操作协议, 而只需要设置好您最关心的硬件参数(比如采集的 AD 通道、采样频率等), 然后用一个读数的函数, 跟上 AD 数据缓冲区和请求采集的数据长度, 一执行程序便可以得到外部的数据, 试想, 这不是一种最高效、最简单易用的方式。这种方式我们优先提供。比如下面将介绍的接口函数: InitDeviceProAD、ReadDeviceProAD 等两三个函数便可以帮助上层用户实现数据采集; 但如果您是较为底层的用户, 对硬件设备很熟悉, 且有更特殊的编程和控制模式, 那么您可能需要对硬件进行直接编程, 就象传统的 ISA 总线设备, 您想用 C 语言的 outp、inp, 汇编语言的 out、in 的命令访问这些 ISA 设备的形式来访问 PCI 设备的各个寄存器, 那么我们为您提供了几类类似的方法, 只是函数名不一样, 如写寄存器函数名为: [WriteRegisterULong](#)(32 位方式), 读寄存器函数名为 [ReadRegisterULong](#) (32 位方式), 但是功能更强, 不仅有 32 位的, 更有 16 位的, 8 位的读写函数。它能访问硬件说明书里提供的任何一个寄存器。凡未注明, 本设备所列寄存器均以 32 位方式访问, 当然, 随着设备与驱动程序的升级, 我们还会提供 64 位的寄存器读写接口。

九、安装程序特点

关于驱动程序的安装方式我们采用大多数 Windows 应用程序所使用的标准模式,因而简捷、方便、直观。您只需执行安装盘上的 Setup.exe 启动文件即可进行驱动程序的安装工作。在安装过程中您设置好安装目标路径以及文件夹名称等信息后,安装程序便自动而又快捷地为您安装好驱动程序,随后您便可以用驱动程序接口编写应用程序或用我们提供的简易测试程序测试设备了。

十、多语言编程环境

本系统提供 Visual C++、C++ Builder、Visual Basic、Delphi、LabView、LabWindows/CVI 的函数接口,使您完全可以根据自己的需要和喜爱选择合适的编程语言。请记住,您得使用 32 位编程模式。但对于 LabWindows/CVI 接口,属于定制服务,如果用户需要,请另与我公司或指定代理商联系以便协商解决。

十一、为 Visual Basic 环境提供直接的多线程支持

在 VB 环境中进行各种实时控制和用户级后台操作,不用子线程,那简直是不可想象的事情。但是在通常情况下,要在 VB 环境中实现多线程操作并不象 VC 那么容易了。往往要相当复杂的对象操作,而且很不具有灵活性。但是有了我们的驱动程序支持,使这件事变得极为容易,甚至比 VC 还要容易。比如执行 CreateVBThread 函数,跟上 hThread 和 NewRoutine 两个参数,即可创建线程对象,并获得对象句柄,随后便可用 ResumeThread 函数启动子线程。在 VB 应用程序中,可以创建任意多个子线程。

十二、我公司动态库与其他公司动态库的比较

值得注意的是,我们的 DLL 库不同于其它许多公司所编写的那样,只是对动态库的简单直接地调用,其硬件控制、数据传输代码都放在 DLL 中,那么其代码的优先执行级别跟一般的用户程序是一样的,它总要定期地、不断地被系统级任务调度器调度,所以当这些代码在负责传输数据时往往被瞬时中断,有时这个时间还很长,故此,极有可能造成丢点的严重现象。且这种方案不可能提供硬件中断以及内存直接存取(DMA)方式来传输数据,这样难以满足用户的各种需求。为了解决这些问题,在 Win95、Win98 环境下,我们没有把硬件控制、数据传输代码简单地放在 DLL 中,而是通过动态虚拟技术以 VxD 的形式放在了 Windows 系统空间中,以 CPU 的 0 级环级别同系统代码协同工作,也就是说它可以获得与任务调用器一样的级别,且不受任务调度器的调度管理。在 NT 环境下,我们通过微内核技术把硬件控制、数据传输代码以微内核代码(简称微代码)形式放在 NT 的内核模式中,成为 NT 操作系统的一部分,并可根据代码的重要程度进一步迅速临时提升 CPU 的 IRQL 级别,使这些代码以高优先级、高速度工作,极大的提高了数据采集和传输的质量。而我们的 DLL 的主要任务不是采集数据,而是对驱动程序的全面封装,对用户负责简化所有复杂的繁琐的操作细节,特别是 Windows 底层管理,提供简洁一致的函数接口供用户使用。它具体表现在从用户空间到系统空间(Windows95,98)、从用户模式到内核模式(Windows NT)、从 CPU 的 3 级环到 0 级环(Windows)等相互间的转换以及设备 I/O 请求的来回传递。所以,我们的驱动程序不是 DLL,而是形如*.VxD(Win95)或*.SYS(NT)的代码文件。通过这样的技术便能实现设备所有功能,极大范围地满足用户需要。

十三、跨平台设计

至今,Windows95 与 Windows NT 是两大主流操作系统,它们各有其优点,但随着计算机的进一步网络化以及追求高可靠性和高稳定性,Windows NT 在不远的将来便会最终取代其它相关的 Windows 平台,成为更先进的视窗操作系统。为保护用户的软硬件投资,满足用户更长远的需要,我们不仅同时提供了基于 Windows95(98)和 Windows NT 操作平台的驱动程序,而且尽力做到了跨平台设计,使您的用户程序不用作怎么修改,一般只须在不同的平台重新编译、链接,便可运行在两种平台上。

十四、LabView/CVI 支持

LabView/CVI 是美国国家仪器公司(National Instrument)的虚拟仪器开发平台,特别是基于图形化编程的 LabView 语言,在测量、工控、虚拟仪器方面受到广大工程师和用户的青睐。其全球销售量仅次于 C++ 语言。我们自主开发的硬件(PCI、USB、ISA 总线系列)产品提供了基于 LabView 的驱动软件接口模块,与 LabView 软件平台完全兼容,让您轻松实现图形化编程。

十五、对传统总线设备的支持

由于某些用户可能除了使用现在流行的高级设备,如 PCI、USB 等总线的设备,但同时还要沿用以前购置的老设备(如基于 ISA 总线、并口、串口等设备)这些设备只能使用 I/O 地址,而不象 PCI 设备那样能使用内存映射地址。特别是对这些设备的访问,只能使用汇编语言的 in、out 指令和 C 语言的 outp、inp 等函数来实现读写。但是这些手段只能在 DOS、Win3.1、Win95 等操作系统中使用,如果您要在 Windows NT、Windows 2000、Windows XP 中使用,那是绝对不行的。因为这些指令已被操作系统作为权限控制,不允许用户在上层应用程序中使用这为了尽可能的保护您的前期投资,我们为您提供了能直接访问这些老设备的相应接口。如 WritePortByte、ReadPortByte 等。注意,为了提高速度,您应使用带后缀“Ex”函数访问设备,如 WritePortByteEx、ReadPortByteEx 等,因这些函数通过驱动程序底层的支持,突破了操作系统的某些限制,使它们能在用户直接访问 I/O 端口。

十六、自动卸载功能

在您已安装了本软件系统后,如果不再准备使用本系统,您可以通过我们为您提供的组件 unInstallShield 从 Windows 系统中自动卸载本软件系统。

十七、LabView/CVI 支持

如果您采用 Typical 安装选项，那么您一般可以得到我们为您提供的如下组件：

Hardware Help 硬件使用说明 Word 帮助文档；

ReadmeFile 安装目录等信息简介；

Software Help 软件使用说明 Word 帮助文档；

Test Application 基于 Microsoft Visual C++代码的硬件测试应用程序；

Visual C++ Sample Microsoft VC++演示程序（这个程序对驱动程序演示说明最全面）；

Visual Basic Microsoft VB 演示及接口程序文件(PCI2005.Bas)

C++ Builder Borland C++ Builder 演示程序；

Delphi Borland Delphi 演示及接口程序文件（PCI2005.Pas）；

LabView 美国国家仪器公司(National Instrument)的虚拟仪器开发平台的演示程序及接口模块程序

UnInstallShield 本软件卸载应用程序；

第三章 PCI 即插即用设备驱动程序安装

注意：均以 PCI2000 作为范例

第一节 Windows95、98、Me 环境下 PCI 设备驱动程序安装

一、安装步骤

第一步 将 PCI 设备按硬件要求插入计算机主板上的任意一个 PCI 插槽中，并将其固定好，连接好其外接设备后，打开计算机电源，启动 Windows95/98/Me 系统。

第二步 如果您正确地插好了 PCI 设备，Windows 系统在启动过程中便会发现这个新的 PCI 设备，并弹出“找到新硬件”的对话框，几秒钟后，便进入“添加新硬件向导”对话框的第一步，它告之所发现的新硬件的设备类型为“PCI Card”或“PCI Input Device”，在然后请单击“下一步”按钮。



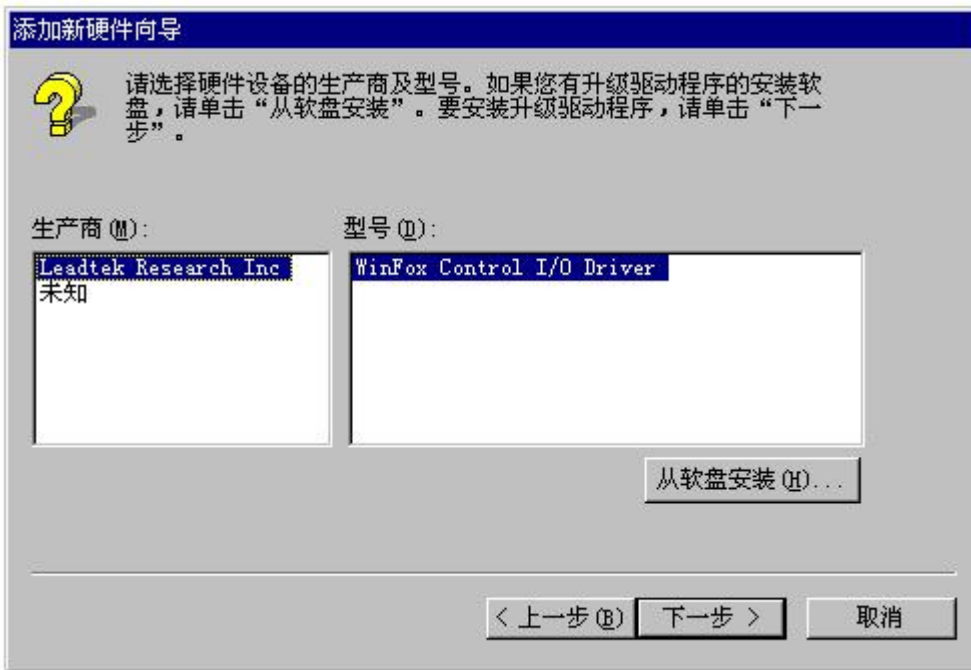
第三步 “添加新硬件向导”对话框的第二步，询问您是自动搜索 PCI 的驱动程序，还是手工从列表中选择。您最好单选第二个选项“显示指定位置的所有驱动程序列表，以便可从列表中选择所需的驱动程序”然后单击“下一步”按钮。



第四步“添加新硬件向导”对话框的第三步进入设备类型列表，您应滚动该列表，选择“其他设备”然后单击“下一步”按钮。



第五步“添加新硬件向导”对话框的第四步是在列表中选择具体的设备名，此处您应单击“从软盘安装”按钮，“从软盘安装”对话框，单击“浏览”，如果从光盘安装，将路径定位在光盘上的\PCI\PCI2310\INF\WIN9X路径下，选择 Art_PCI.INF 文件，其他设备类同。并单击“确定”，即出现我公司 PCI 产品型号列表，然后您根据您所购买的 PCI 产品型号选择相应项，然后单击“下一步”按钮。





第五步 “添加新硬件向导”对话框的第五步 告诉用户该设备的型号及驱动程序的 INF 文件所在位置，然后单击“下一步”。即刻弹出“复制驱动程序文件”对话框之后相继弹出“创建驱动程序信息库”对话框，稍等片刻，即进入最后一步。（这里以 PCI2310 为例，其他设备完全一样，只是设备名不一样而已）





第六步 在最后一步中，用户只须单击“完成”按钮，那么 PCI 硬件驱动程序即安装成功。



二、安装结果验证

进入 Windows95/98/Me “控制面板”窗口，双击“系统”图标，弹出“系统 属性”对话框，在对话框中单击“设备管理器”标签，然后在“计算机”树形列表中双击“阿尔泰科技”，检查此项目中是否有“Art PCIxxxx...”等字样。若有，表示 PCI 设备和其驱动程序已成功安装，否则，说明您的安装过程出现了问题，请试着再安装，或向硬件供应商求助。



三、疑难问题解答

如果当您安装了 PCI 设备后，在 Windows 启动过程中屏幕上没有任何反应，在控制面板的系统属性对话框中也没有出现“Art PCI...”等字样，有可能您的设备与计算机连接出现了问题。请关掉计算机，再试着拔插一次 PCI 设备或将其插入另外一个 PCI 插槽里，然后打开计算机电源，请注意在进入 Windows 启动画面以前的字符模式下显示的最后一屏信息，即“PCI Device Listing”列表，看其中是否有 Vendor ID 为 11e3, Device ID 为 2000，如果没有这些字符显示，则视为 PCI 设备与计算机连接不正常。

第二节、Windows NT 环境下 PCI 设备及驱动程序安装

一、安装步骤

第一步 将 PCI 设备按硬件要求插入计算机主板上的任意一个 PCI 插槽中，并将其固定好，连接好其外接设备后，打开计算机电源，启动 Windows NT 系统。

第二步 在 Windows NT 的资源管理器中，进入驱动程序安装盘，在\PCIPCI2310\App 目录下双击带有计算机小图标的 Setup.exe 可执行文件，按照第四节<PCI 接口程序及测试、示范程序的安装>的说明完成安装后，会提示您重新复位操作系统，然后用户选择“是<Y>”以重新启动计算机即可。

二、安装结果验证

从 Windows NT 的任务栏，进入“阿尔泰测控演示系统\PCIxxxx....”，单击“高级测试演示系统。”菜单项，启动测控演示程序，如果设备及驱动程序没有装好，则该应用程序会对话框告之没有装好设备及驱动程序。或者用户还可以在该应用程序的“设备管理”菜单中单击“列表该设备”菜单项，在弹出的对话框中看是否能列出 PCI2310 的硬件资源分配情况，如果有且正确，即可该设备已与系统正确连接。否则，说明您的安装过程出现了问题，请试着重新启动计算机，注意在正式进入 Windows 启动画面以前的字符模式下的最后一屏信息出现时按键盘右上方的 [Pause] 键，当这一屏画面停止后，仔细查看屏幕上的“PCI Device Listing”列表，看其中是否有 Vendor ID 为 11e3, Device ID 为 2000 的表项，如果没有这样的表项，则视为 PCI 设备与计算机连接不正常。请试着重新安装驱动程序或将 PCI 板从计算机插槽中拔出再插入另外一个 PCI 插槽中再试，或向硬件供应商求助。

第三节、Windows2000 环境下 PCI 设备及驱动程序安装

一、安装步骤

第一步 将 PCI 设备按硬件要求插入计算机主板上的任意一个 PCI 插槽中，并将其固定好，连接好其外接设备后，打开计算机电源，启动 Windows2000 系统。

第二步 如果您正确地插好了 PCI 设备，Windows 系统在启动过程中便会发现这个新的 PCI 设备，并弹出 [欢迎使用找到新的硬件向导] 对话框，单击 [下一步] 按钮



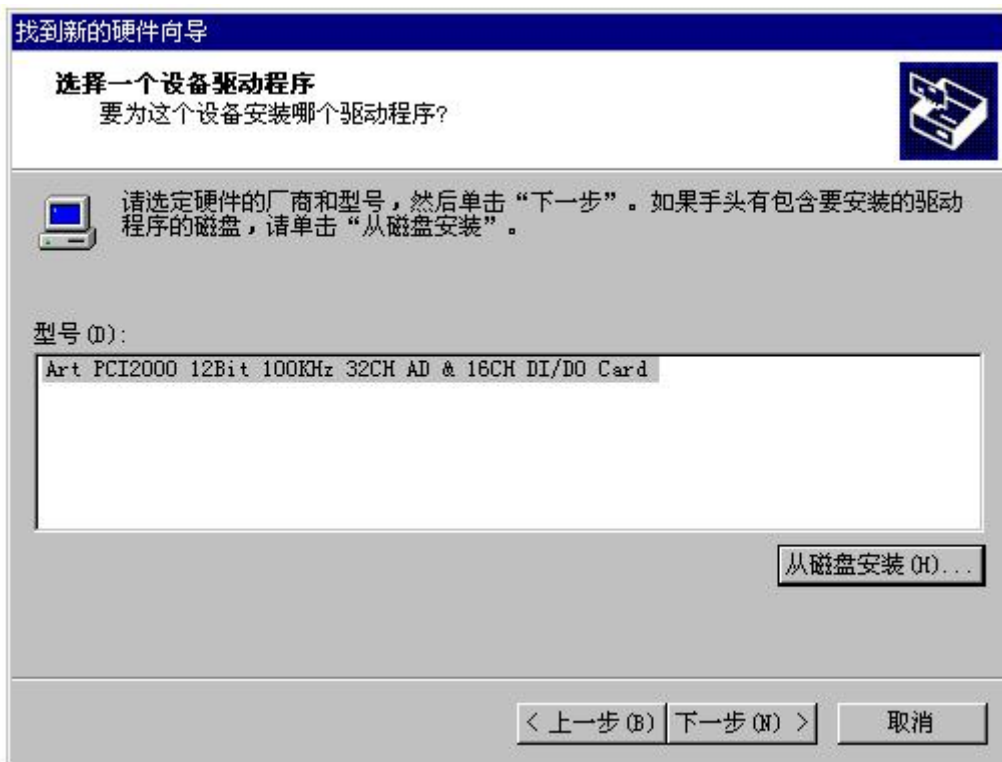
第三步 弹出[安装硬件设备驱动程序]对话框，在对话框中单击 [显示已知设备程序列表，从中选择特定驱动程序] 单选框，然后单击 [下一步] 按钮



第四步 弹出[选择一个设备驱动程序]对话框，在对话框中单击“从磁盘安装”，在“从磁盘安装”对话框中再单击 [浏览] 按钮，然后将文件目录定位在安装盘的...\\PCI2310\\INF \\Win2000 目录下，并选择 ART_PCI.INF 文件，最后单击 [打开] 按钮回到 [从磁盘安装] 对话框中，此时单击 [确定] 按钮再回到 [选择一个设备驱动程序] 对话框中，单击 [下一步]按钮。



第五步 [开始设备驱动程序安装], 用户确认一下计算机图标右边所列设备名是不是 PCI2003 的相关字样, 单击 [下一步]按钮。



第六步 此步骤可能会出现 Windows 安装驱动程序的进度状态窗口, 用户稍等片刻, 然后出现[完成找到新硬件向导]对话框, 单击 [完成]按钮。



第七步 可能会出现 [系统设置改变]对话框提示，要求用户重新启动计算机，单击 [是]按钮即可成功。

二、安装结果验证

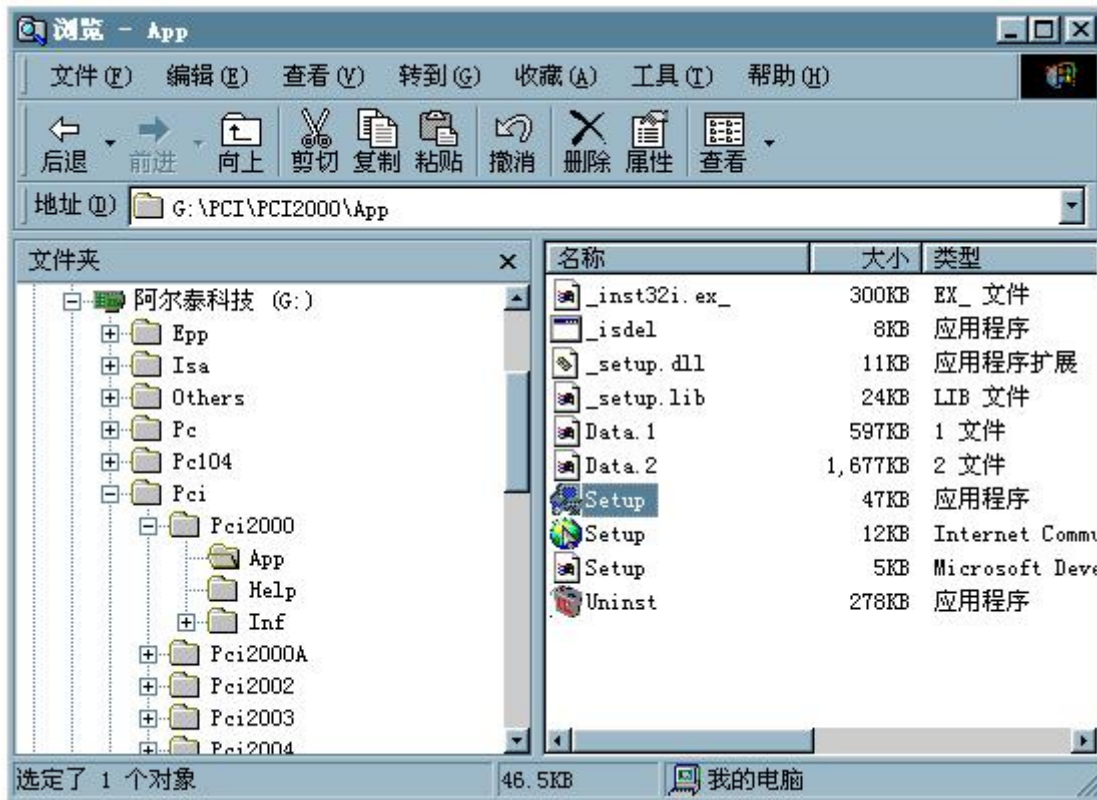
进入 Windows2000 [控制面板] 窗口，双击 [系统] 图标，弹出 [系统 特性] 对话框，在对话框中单击 [硬件] 标签页，然后单击 [设备管理器]按钮，进入 [设备管理器] 窗口，在 [本地计算机] 列表中单击 [系统设备] 在展开的子列表中检查是否有“Art PCIxxxx.....”等字样，若有，表示 PCI 设备和其驱动程序已成功安装，否则，说明您的安装过程出现了问题，请试着再安装，或向硬件供应商求助。

三、疑难问题解答

如果当您安装了 PCI 设备后，在 Windows 启动过程中屏幕上没有任何反应，在控制面板的 [系统 特性] 对话框中也没有出现“Art PCI...”等字样，有可能您的设备与计算机连接出现了问题。请关掉计算机，再试着拔插一次 PCI 设备或将其插入另外一个 PCI 插槽里，然后打开计算机电源，请注意在进入 Windows 启动画面以前的字符模式下显示的最后一屏信息，即“PCI Device Listing”列表，看其中是否有 Vendor ID 为 11e3, Device ID 为设备类型编号，如果没有这些字符显示，则视为 PCI 设备与计算机连接不正常。

第四节 PCI 接口程序及测试、示范程序的安装

这部分软件的安装也很简单，在光盘上\PCI\PCI2310 的 App 目录下，双击 setup.exe 应用程序图标，接受默认的设置，单击下一步，直到最后单击完成。当您成功安装了这部分软件后，您再单击系统任务条上的“开始”按钮，进入“程序”菜单中，便会发现有了“阿尔泰测控演示系统”主菜单，您再进入这个菜单项，便会找到您需要的应用程序。



第五节 PCI 设备软件测试系统的介绍

- 1.怎样进入测试系统：当您正确完成了第四节中的工作，您便可以在 Windows 的系统菜单中启动“Art VC Test Application……”即可进入设备测试系统。
- 2.怎样进行开关量测试,最好的办法是将板上的开关量输出端与开关量输入端一一对接起来，然后用户点击左边的按钮进行输出，则右边对应的输入则将发生相应变化，这种方法将开关量的输入和开关量输出同时进行测试，且也是一种最可靠的测试方法。

第六节 本驱动程序软件的关键文件

(WinDir 指 Windows 的系统根目录, UserDir 为本驱动软件的用户安装根目录)

文件名	文件类型及功能	适用的操作系统	文件位置
PCI2310.VxD	动态虚拟设备驱动程序库	Window95/98	WinDir\System
PCI2310.Sys	Win32 标准设备驱动 WDM 模式的设备驱动程序库	Windows NT/2000	WinDir\System32\Drivers
PCI2310.Dll	底层驱动程序库的用户级函数接口封装所用的动态库。	所有操作系统	WinDir\System
PCI2310.Lib	基于 Microsoft Visual C++工程开发环境的驱动程序函数接口输入库。	所有操作系统	UserDir\Include 或 UserDir\Samples\VC...
PCI2310.Lib	基于 Borland C++ Builder 工程开发环境的驱动程序函数接口输入库。	所有操作系统	UserDir\Samples\C_Builder
PCI2310.Bas	基于 Microsoft Visual Basic 工程开发环境的驱动程序函数接口输入模块文件	所有操作系统	UserDir\Samples\VB

PCI2310.Pas	基于 Borland Delphi 工程开发环境的驱动程序函数接口输入单元文件。	所有操作系统	UserDir\Samples\Delphi
PCI2310.VI	基于 National Instrument LabView 工程开发环境的驱动程序函数接口输入部件文件。(只是外挂驱动接口)	所有操作系统	UserDir\Samples\LabView

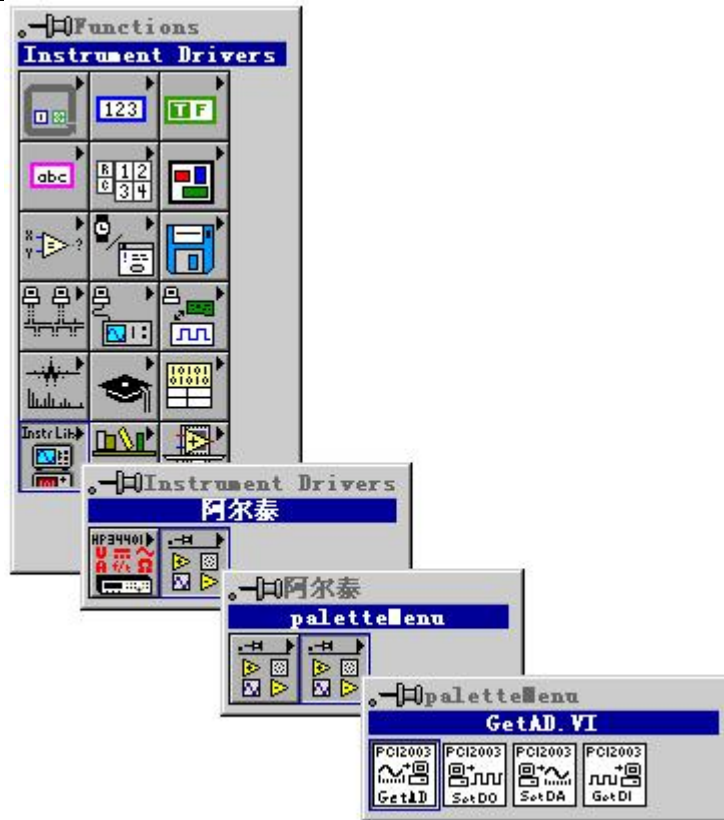
第四章 PCI 即插即用设备操作函数接口介绍

由于我公司的设备应用于各种不同的领域，有些用户可能根本不关心硬件设备的控制细节、只关心开关量的通道等，然后就能通过一两个简易的采集函数便能轻松得到所需要的开关量控制。这方面的用户我们称之为上层用户。那么还有一部分用户不仅对硬件控制熟悉，而且由于应用对象的特殊要求，则要直接控制设备的每一个端口，这是一种复杂的工作，但又是必须的工作，我们则把这一群需要直接跟设备端口打交道的用户称之为底层用户。因此总的看来，上层用户要求简单，快捷，他们最希望他们在软件操作上所要面对的全是他们最关心的问题，比如在正式开关量控制之前，只须用户简单赋值参数中的通道等，然后便可以用 SetDeviceDO（或 GetDeviceDI）函数即可实现。而关于设备的物理地址、端口分配及功能定义等复杂的硬件信息则与上层用户无任何关系。那么对于底层用户则不然。他们不仅要关心设备的物理地址，还要关心虚拟地址、端口寄存器的功能分配，甚至每个端口的 Bit 位都要了如指掌，看起来这是一项相当复杂、繁琐的工作。但是这些底层用户一旦使用我们提供的技术支持，则不仅可以让您不必熟悉 PCI 总线复杂的控制协议，同是还可以省掉您许多繁琐的工作，比如您不用去了解 PCI 的资源配置空间、PNP 即插即用管理，而只须用 GetDeviceAddr 函数便可以同时取得指定设备的物理基地址和虚拟线性基地址。这个时候您便可以用这个虚拟线性基地址，再根据硬件使用说明书中的各端口寄存器的功能说明，然后使用 ReadPortWord 和 WritePortWord 对这些端口寄存器进行 16 位模式的读写操作，即可实现设备的所有控制。

综上所述，用户使用我公司提供的驱动程序软件包极大的方便和满足您的各种需求。但为了您更省心，别忘了在您正式阅读下面的函数说明时，先得明白自己是上层用户还是底层用户，因为在《第一节 接口函数列表》中的备注栏里明确注明了适用对象。

另外需要申明的是，在本章和下一章中列出的关于 LabView 的接口，均属于外挂式驱动接口，他是通过 LabView 的 Call Library Function 功能模板实现的。它的特点是除了自身的语法略有不同以外，每一个基于 LabView 的驱动图标与 Visual C++、Visual Basic、Delphi 等语言中每个驱动函数是一一对应的，其调用流程和功能是完全相同。那么相对于外挂式驱动接口的另一种方式是内嵌式驱动。这种驱动是完全作为 LabView 编程环境中的紧密耦合的一部分，它可以直接从 LabView 的 Functions 模板中取得，如下图所示。此种方式更适合上层用户的需要，它的最大特点是方便、快捷、简单，而且可以取得它的在线帮助。**此功能由于 LabView 自身版本兼容的问题，我们不便提供内嵌式驱动，如果用户确有此要求，请与我们的代理商或公司总部联系，但我们不保证完全免费。**

关于 LabView 的外挂式驱动和内嵌式驱动更详细的叙述，请参考附录 A 的《[LabView 驱动程序接口](#)》章节。



LabView 内嵌式驱动接口的获取方法

第一节 接口函数列表（每个函数省略了前缀“PCI2310_”）

函数名	函数功能	备注
PCI 通用函数		
CreateDevice	创建 PCI 设备对象	上层及底层用户
GetDeviceCount	取得同一种 PCI 设备的总台数	上层及底层用户
ListDevice	列表所有同一种 PCI 设备的各种配置	上层及底层用户
ReleaseDevice	关闭设备，且释放 PCI 总线设备对象	上层及底层用户
开关量简易操作函数		
SetDeviceDO	开关输出函数	上层用户
GetDeviceDI	开关输入函数	上层用户
中断操作函数		
InitDeviceInt	初始化硬件中断函数	上层用户
GetDeviceIntCount	取得中断产生次数	上层用户
ReleaseDeviceInt	释放中断资源函数	上层用户
PCI 总线内存映射寄存器操作函数		
GetDeviceAddr	取得指定 PCI 设备寄存器操作基地址	底层用户
WritePortByte	以字节(8Bit)方式写 I/O 端口	底层用户
WritePortWord	以字(16Bit)方式写 I/O 端口	底层用户
WritePortULong	以无符号双字(32Bit)方式写 I/O 端口	底层用户
ReadPortByte	以字节(8Bit)方式读 I/O 端口	底层用户
ReadPortWord	以字(16Bit)方式读 I/O 端口	底层用户
ReadPortULong	以无符号双字(32Bit)方式读 I/O 端口	底层用户

使用需知：

Visual C++ & C++Builder:

要使用如下函数关键的问题是：

首先，必须在您的源程序中包含如下语句：

```
#include "C:\Art\PCI2310\INCLUDE\PCI2310.H"
```

注：以上语句采用默认路径和默认板号，应根据您的板号和安装情况确定 PCI2310.H 文件的正确路径，当然也

可以把此文件拷到您的源程序目录中。

其次,您还应该在 Visual C++ 编译环境软件包的 Project Setting 对话框的 Link 属性页中的 Object/Library Module 输入行中加入指令 C:\Art\PCI2310\PCI2310.LIB

或者:单击 Visual C++ 编译环境软件包的 Project 菜单中的 Add To Project 的菜单项,在此项中再单击 Files..., 在随后弹出的对话框中选择 PCI2310.Lib, 再单击“确定”, 即可完成。

注:以上语句采用默认路径和默认板号,应根据您的板号和安装情况确定 PCI2310.LIB 的路径,当然也可以把此文件拷到您的源程序目录中。

另外,在 Visual C++ 演示工程的目录下,也有相应的 PCI2310.h 和 PCI2310.Lib 文件。

为了驱动程序和相关接口尽量精炼快速,所以没有加任何调试代码,因此用户在使用 VC 接口的时候应使用发行版本进行源代码编译 (Win32 Release), 而不应该使用调试版本 (Win32 Debug)。具体方法是在源代码编译前,执行 Build 总菜单中的 Set Active Configuration 子菜单命令,便可实现其发行版的设置,然后再编译,即可生成发行版的应用程序。

C++ Builder:

要使用如下函数一个关键的问题是首先必须将我们提供的头文件

(PCI2310.H)写进您的源程序头部。如: #include “\Art\PCI2310\Include\PCI2310.h”

然后再将 PCI2310.Lib 库文件分别加入到您的 C++ Builder 工程中。其具体办法是选择 C++ Builder 集成开发环境中的工程(Project)菜单中的“添加”(Add to Project)命令,在弹出的对话框中分别选择文件类型: Library file (*.lib), 即可选择 PCI2310.Lib 文件。该文件的路径为用户安装驱动程序后其子目录 Samples\C_Builder 下

Visual Basic:

要使用如下函数一个关键的问题是首先必须将我们提供的模块文件(*.Bas)加入到您的 VB 工程中。其方法是选择 VB 编程环境中的工程(Project)菜单,执行其中的“添加模块”(Add Module)命令,在弹出的对话框中选择 PCI2310.Bas 模块文件,该文件的路径为用户安装驱动程序后其子目录 Samples\VB 下面。

请注意,因考虑 Visual C++ 和 Visual Basic 两种语言的兼容问题,在下列函数说明和示范程序中,所举的 Visual Basic 程序均是需编译后在独立环境中运行。所以用户若在解释环境中运行这些代码,我们不能保证完全顺利运行。

Delphi:

要使用如下函数一个关键的问题是首先必须将我们提供的单元模块文件(*.Pas)加入到您的 Delphi 工程中。其方法是选择 Delphi 编程环境中的 View 菜单,执行其中的“Project Manager”命令,在弹出的对话框中选择*.exe 项目,再单击鼠标右键,最后 Add 指令,即可将 PCI2310.Pas 单元模块文件加入到工程中。或者在 Delphi 的编程环境中的 Project 菜单中,执行 Add To Project 命令,然后选择*.Pas 文件类型也能实现单元模块文件的添加。该文件的路径为用户安装驱动程序后其子目录 Samples\Delphi 下面。最后请在使用驱动程序接口的源程序文件中的头部的 Uses 关键字后面的项目中加入:“PCI2310”。如:

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

PCI2310; // 注意: 在此加入驱动程序接口单元 PCI2310

LabView/CVI :

LabVIEW 是美国国家仪器公司(National Instrument)推出的一种基于图形开发、调试和运行程序的集成化环境,是目前国际上唯一的编译型的图形化编程语言。在以 PC 机为基础的测量和工控软件中,LabVIEW 的市场普及率仅次于 C++/C 语言。LabVIEW 开发环境具有一系列优点,从其流程图式的编程、不需预先编译就存在的语法检查、调试过程使用的数据探针,到其丰富的函数功能、数值分析、信号处理和设备驱动等功能,都令人称道。关于 LabView/CVI 的进一步介绍请见本文最后一部分关于 LabView 的专述。其驱动程序接口单元模块的使用方法如下:



一、在 LabView 中打开 PCI2310.VI 文件,用鼠标单击接口单元图标,比如 CreateDevice 图标

然后按 Ctrl+C 或选择 LabView 菜单 Edit 中的 Copy 命令,接着进入用户的应用程序 LabView 中,按 Ctrl+V 或选择 LabView 菜单 Edit 中的 Paste 命令,即可将接口单元加入到用户工程中,然后按以下函数原型说明或演示程序的说明连续该接口模块即可顺利使用。

二、在单元接口图标中,凡标有“I32”为有符号长整型 32 位数据类型,“U16”为无符号短整型 16 位数据类型,“[U16]”为无符号 16 位短整型数组或缓冲区或指针,“[U32]”与“[U16]”同理,只是位数不一样。

第二节、设备对象管理函数原型说明

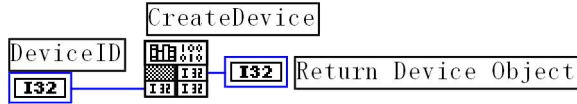
1、创建设备对象函数

Visual C++ & C++Builder:

HANDLE CreateDevice (int DeviceID)

Visual Basic:

Declare Function CreateDevice Lib "PCI2310"(ByVal DeviceID as long)as long

Delphi:Function CreateDevice(DeviceID:Integer):Integer;
StdCall; External 'PCI2310' Name 'CreateDevice';**LabView:**

功能: 该函数负责创建 PCI 设备对象，并返回其设备对象句柄。

参数:

DeviceID 设备 ID (Identifier) 标识号。当向同一个 Windows 系统中加入若干相同类型的 PCI 设备时，我们的驱动程序将以该设备的“基本名称”与 DeviceID 标识值为名称后缀的标识符来确认和管理该设备。比如若用户往 Windows 系统中加入第一个 PCI2310 模板时，驱动程序则以“PCI2310”作为基本名称，再以 DeviceID 的初值组合成该设备的标识符“PCI2310-0”来确认和管理这第一个设备，若用户接着再添加第二个 PCI2310 模板时，则系统将以“PCI2310-1”来确认和管理第二个设备，若再添加，则以此类推。所以当用户要创建设备句柄管理和操作第一个 PCI 设备时，DeviceID 应置 0，第二个应置 1，也以此类推。

返回值: 如果执行成功，则返回设备对象句柄；如果没有成功，则返回错误码 INVALID_HANDLE_VALUE。由于此函数已带容错处理，即若出错，它会自动弹出一个对话框告诉您出错的原因。您只需要对此函数的返回值作一个条件处理即可，别的任何事情您都不必做。

相关函数: [ReleaseDevice](#)

Visual C++ & C++Builder 程序举例

```

:
HANDLE hDevice; // 定义设备对象句柄
hDevice=CreateDevice ( 0 ); // 创建设备对象,并取得设备对象句柄
if(hDevice==INVALID_HANDLE_VALUE); // 判断设备对象句柄是否有效
{
    return; // 退出该函数
}
)
:

```

Visual Basic 程序举例

```

:
Dim hDevice As Long ' 定义设备对象句柄
hDevice = CreateDevice ( 0 ) ' 创建设备对象,并取得设备对象句柄
If hDevice = INVALID_HANDLE_VALUE Then ' 判断设备对象句柄是否有效

Else
    Exit Sub ' 退出该过程
End If
:

```

2、取得本计算机系统中 PCI2310 设备的总数量

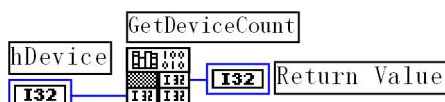
函数原型:

Visual C++ & C++Builder:

int GetDeviceCount (HANDLE hDevice)

Visual Basic:

Declare Function GetDeviceCount Lib "PCI2310" (ByVal hDevice As Long) As Long

Delphi:Function PCI2310_GetDeviceCount (hDevice : Integer):Integer;
StdCall; External 'PCI2310' Name 'GetDeviceCount';**LabView:**

函数原型:

功能: 取得 PCI2310 设备的数量。

参数: hDevice 设备对象句柄, 它应由 CreateDevice 创建。

返回值: 返回系统中 PCI2310 的数量。

相关函数: [CreateDevice](#) [ReleaseDevice](#)

3、用对话框控件列表计算机系统中所有 PCI2310 设备各种配置信息

函数原型:

Visual C++ & C++Builder:

BOOL ListDevice (HANDLE hDevice)

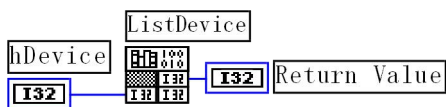
Visual Basic:

Declare Function ListDevice Lib "PCI2310" (ByVal hDevice As Long) As Boolean

Delphi:

Function ListDevice (hDevice : Integer):Boolean;
StdCall; External 'PCI2310' Name 'ListDevice';

LabView:



功能: 列表系统中 PCI2310 的硬件配置信息。

参数: hDevice 设备对象句柄, 它应由 CreateDevice 创建。

返回值: 若成功, 则返回列表设备。

相关函数: [CreateDevice](#) [ReleaseDevice](#)

4、释放设备对象所占的系统资源及设备对象

函数原型:

Visual C++ & C++Builder:

BOOL ReleaseDevice(HANDLE hDevice)

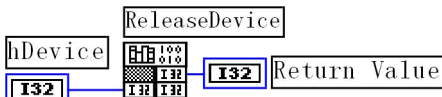
Visual Basic:

Declare Function ReleaseDevice Lib "PCI2310" (ByVal hDevice As Long) As Boolean

Delphi:

Function ReleaseDevice(hDevice : Integer):Boolean;
StdCall; External 'PCI2310' Name 'ReleaseDevice';

LabView:



功能: 释放设备对象所占用的系统资源及设备对象自身。

参数: hDevice 设备对象句柄, 它应由 CreateDevice 创建。

返回值: 若成功, 则返回 TRUE, 否则返回 FALSE, 用户可以用 GetLastError 捕获错误码。

相关函数: [CreateDevice](#)

应注意的是, CreateDevice 必须和 ReleaseDevice 函数一一对应, 即当您执行了一次 CreateDevice 后, 再一次执行这些函数前, 必须执行一次 ReleaseDevice 函数, 以释放由 CreateDevice 占用的系统软硬件资源, 如 DMA 控制器, 系统内存等。只有这样, 当您再次调用 CreateDevice 函数时, 那些软硬件资源才可被再次使用。

第三节、简易的数字 IO 输入输出开关量操作函数原型说明

三十二路开关量输出

函数原型:

Visual C++ & C++Builder:

BOOL SetDeviceDO (HANDLE hDevice, BYTE bDOSts[32])

Visual Basic:

Declare Function SetDeviceDO Lib "PCI2307" (ByVal hDevice As Long, _
ByVal bDOSts(31) As Byte) As Boolean

Delphi:

Function SetDeviceDO (hDevice : Integer; bDOSts[32]:Byte):Boolean;

StdCall; External 'PCI2307' Name ' SetDeviceDO';

LabView(包括相关演示):

功能: 此函数负责将 bDOSets[0]–bDOSets[31]共 32 路开关量输出置成相应的状态。

参数:

hDevice 设备对象句柄,它应由 CreateDevice 决定。

bDOSets 三十二路开关量输出状态的参数数组分别对应于 DO0-DO31 路开关量输出状态位。比如置 bDOSets[0]为“1”则使 0 通道处于“开”状态,若为“0”则置 0 通道为“关”状态。其他同理。请注意,在实际执行这个函数之前,必须分配 bDOSets 为 32 个元素且为 Byte 类型的数组,对每个元素赋初值,其值必须为“1”或“0”。

返回值: 若成功,返回 TRUE,否则返回 FALSE。

相关函数: [CreateDevice](#) [GetDeviceDI](#) [ReleaseDevice](#)

三十二路开关量输入

函数原型:

Visual C++ & C++Builder:

BOOL GetDeviceDI (HANDLE hDevice, BYTE bDISts[32])

Visual Basic:

Declare Function GetDeviceDI Lib "PCI2307" (ByVal hDevice As Long, _
ByVal bDISts(31) As Byte) As Boolean

Delphi:

Function GetDeviceDI (hDevice : Integer; bDISts[32]:Byte):Boolean;

StdCall; External 'PCI2307' Name ' GetDeviceDI ';

LabView(包括相关演示):

功能: 负责将 PCI 设备上的输入开关量状态读入内存。

参数:

hDevice 设备对象句柄,它应由 CreateDevice 决定。

bDISts 三十二路开关量输入状态的参数数组分别对应于 DI0-DI31 路开关量输入状态位。比如置 bDISts[0]为“1”则使 0 通道处于“开”状态,若为“0”则置 0 通道为“关”状态。其他同理。请注意,在实际执行这个函数之前,必须分配 bDOSets 为 32 个元素且为 Byte 类型的数组。

返回值: 若成功,返回 TRUE,其 bDISts 中的值有效;否则返回 FALSE,其 bDISts 中的值无效。

相关函数: [CreateDevice](#) [SetDeviceDO](#) [ReleaseDevice](#)

3、以上函数调用一般顺序

① CreateDevice

② SetDeviceDO (或 GetDeviceDI,当然这两个函数也可同时进行)

③ ReleaseDevice

用户可以反复执行第②步,以进行数字 I/O 的输入输出(数字 I/O 的输入输出及 AD 采样可以同时进行,互不影响)。

第四节、硬件中断处理函数(但是在 Win98 下暂不提供)

硬件中断初始化函数

函数原型:

Visual C++ & C++Builder:

BOOL InitDeviceInt (HANDLE hDevice, HANDLE hEventInt)

Visual Basic:

Declare Function InitDeviceInt Lib "PCI2307" (ByVal hDevice As Long, _
ByVal hEventInt As Long) As Boolean

Delphi:

Function InitDeviceInt (hDevice : Integer; hEventInt: Integer):Boolean;

StdCall; External 'PCI2307' Name ' InitDeviceInt ';

LabView(包括相关演示):

功能: 此函数负责挂接中断服务程序,并使其与 hEventInt 消息关联,且复位内核中断计数变量至 0。

参数:

hDevice 设备对象句柄,它应由 **CreateDevice** 决定。

hEventInt 中断事件, 它应由 **CreateSystemEvent** 函数创建。每当硬件中断产生时, 该事件被触发一次。

返回值: 若成功, 返回 TRUE, 否则返回 FALSE。

相关函数: [CreateDevice](#) [ReleaseDeviceInt](#)
[GetDeviceIntCount](#) [ReleaseDevice](#)

取得内核中断服务程序发生次数的函数

函数原型:

Visual C++ & C++Builder:

LONG GetDeviceIntCount (HANDLE hDevice, HANDLE hEventInt)

Visual Basic:

Declare Function GetDeviceIntCount Lib "PCI2307" (ByVal hDevice As Long, _
ByVal hEventInt As Long) As Long

Delphi:

Function GetDeviceIntCount (hDevice : Integer; hEventInt: Integer):Long;
StdCall; External 'PCI2307' Name 'GetDeviceIntCount';

LabView(包括相关演示):

功能: 当外界产生一个硬件中断信号, 即中断服务程序被执行一次, 每当中断服务程序被执行一次时, 即对内核计数变量作加“1”动作, 该函数就是返回内核计数变量的当前值, 该值反映了中断信号产生的次数。

参数:

hDevice 设备对象句柄,它应由 **CreateDevice** 决定。

返回值: 若成功, 返回 0 或 0 以上的数据, 否则返回-1, 以视失败。

相关函数: [CreateDevice](#) [ReleaseDeviceInt](#) [ReleaseDevice](#)

中断释放函数

函数原型:

Visual C++ & C++Builder:

BOOL ReleaseDeviceInt (HANDLE hDevice)

Visual Basic:

Declare Function ReleaseDeviceInt Lib "PCI2307" (ByVal hDevice As Long) As Boolean

Delphi:

Function ReleaseDeviceInt (hDevice : Integer):Boolean; StdCall; External 'PCI2307' Name 'ReleaseDeviceInt';

LabView(包括相关演示):

功能: 负责将 **InitDeviceInt** 函数加载的中断资源释放掉。

参数:

hDevice 设备对象句柄,它应由 **CreateDevice** 决定。

返回值: 若成功, 返回 TRUE, 否则返回 FALSE。

相关函数: [CreateDevice](#) [InitDeviceInt](#)
[GetDeviceIntCount](#) [ReleaseDevice](#)

3、以上函数调用一般顺序

- ① CreateDevice
- ② InitDeviceInt
- ③ WaitForSingleObject (Win32 API 函数, 它以睡眠方式等待中断事件 hEventInt)
- ④ ReleaseDevice

用户可以反复执行第③步, 以重复等待多次中断。

注意事项:

VC 高级演示程序中的中断外计数是指: 在 **hEventInt** 事件的同步下做中断计数动作, 而中断内计数是指在内核中断服务程序中做计数动作。在中断频率较快以及用户在 **WaitForSingleObject** 之后做的工作较多的情况下, 中断外计数往往会小于中断内计数, 也由此说明只有中断内计数才能真正反映硬件中断产生的次数。

第五节、PCI 寄存器操作函数**取得指定寄存器的线性地址和物理地址**

函数原型:

Visual C++ & C++ Builder:

```

BOOL GetDeviceAddr( HANDLE hDevice,
                   PULONG LinearAddr,
                   PULONG PhysAddr,
                   int RegisterID)

```

Visual Basic:

```

Declare Function GetDeviceAddr Lib "PCI2307" (ByVal hDevice as Long, _
                                             ByRef LinearAddr As Long, _
                                             ByRef PhysAddr As Long, _
                                             ByVal RegisterID As Long, _
                                             ) As Boolean

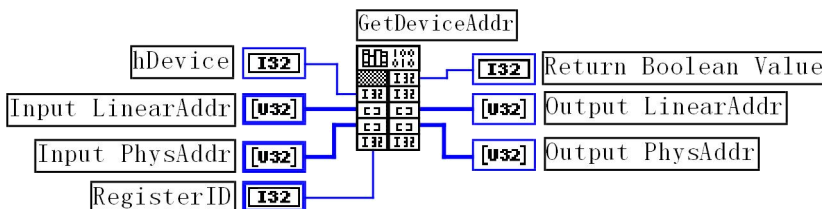
```

Delphi:

```

Function GetDeviceAddr(hDevice : Integer;
                      LinearAddr:Pointer;
                      PhysAddr:Pointer;
                      RegisterID:Integer):Boolean;
StdCall; External 'PCI2307' Name 'GetDeviceAddr';

```

Labview:

功能：取得 PCI 设备的指定的内存映射寄存器的线性地址。

参数：

hDevice 设备对象句柄,它应由 CreateDevice 创建。

LinearAddr 指针参数,用于返回所取得的映射寄存器指向的线性地址,它可用于 WriteRegisterX 或 ReadRegisterX (X 代表 Byte、ULong、Word) 等函数,以便于访问设备寄存器。它指明该设备位于系统空间的虚拟位置。

PhysAddr 所取得的映射寄存器指向的物理地址,它指明该设备位于系统空间的物理位置,它可用于 WritePortX,ReadPortX 等函数。

RegisterID 指定映射寄存器的 ID 号,其取值范围为[0, 5],通常情况下,用户应使用 0 号映射寄存器,特殊情况下,我们为用户加以申明。(但需要用户特别注意的是:本设备使用了三个地址映射寄存器 0、1、3,而 0、1 号寄存器是仅供 PCI 设备本身所专用的,用户只能使用 3 号地址寄存器的物理地址 PhysAddr 来操作 AD 和开关量)。

返回值：如果执行成功,则返回 TRUE,它表明由 RegisterID 指定的映射寄存器的无符号 32 位线性地址和物理地址被正确返回,否则会返回 FALSE,同时还要检查其 LinearAddr 和 PhysAddr 是否为 0,若为 0 则依然视为失败。用户可用 GetLastError 捕获当前错误码,并加以分析。

相关函数： [CreateDevice](#) [WritePortByte](#)
[WritePortWord](#) [WritePortULong](#)
[ReadPortByte](#) [ReadPortWord](#)
[ReleaseDevice](#)

Visual C++ & C++ Builder 程序举例:

```

:
HANDLE hDevice; ULONG LinearAddr, PhysAddr;
hDevice = CreateDevice(0);
if(!GetDeviceAddr(hDevice, &LinearAddr, &PhysAddr, 0))
{
    AfxMessageBox("取得设备地址失败...");
}
:

```

Visual Basic 程序举例:

```

:
Dim hDevice As Long
Dim LinearAddr, PhysAddr As Long
hDevice = CreateDevice(0)
if Not GetDeviceAddr(hDevice, LinearAddr, PhysAddr, 0) then
    MsgBox "取得设备地址失败..."
End If
:

```

1 以单字节(8Bit)方式写 I/O 端口

Visual C++ & C++ Builder:

BOOL WritePortByte (HANDLE hDevice, UINT nPort, BYTE Value)

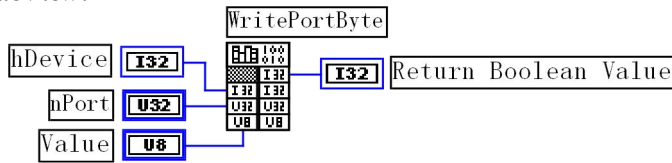
Visual Basic:

Declare Function WritePortByte Lib "PCI2307" (ByVal hDevice As Long, _
ByVal nPort As Long, _
ByVal Value As Byte) As Boolean

Delphi:

Function WritePortByte(hDevice : Integer; nPort:LongWord; Value:Byte):Boolean;
StdCall; External 'PCI2307' Name 'WritePortByte';

LabView:



功能：以单字节(8Bit)方式写 I/O 端口

参数：

hDevice 设备对象句柄,它应由 CreateDevice 或 CreateDeviceEx 创建。

nPort 设备的 I/O 端口号。(本设备应使用 GetDeviceAddr 函数返回的 PhysAddr 地址作为基地址)

Value 写入由 nPort 指定端口的值。

返回值：若成功，返回 TRUE，否则返回 FALSE，用户可用 GetLastError 捕获当前错误码。

相关函数： [CreateDevice](#) [WritePortByte](#)
[WritePortWord](#) [WritePortULong](#)
[ReadPortByte](#) [ReadPortWord](#)
[ReleaseDevice](#)

2 以双字(16Bit)方式写 I/O 端口

Visual C++ & C++ Builder:

BOOL WritePortWord (HANDLE hDevice, UINT nPort, WORD Value)

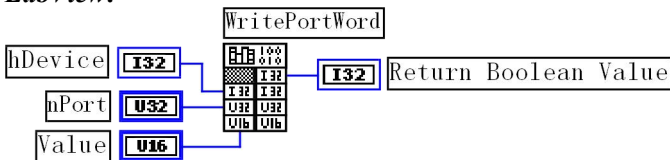
Visual Basic:

Declare Function WritePortWord Lib "PCI2307" (ByVal hDevice As Long, _
ByVal nPort As Long, _
ByVal Value As Integer) As Boolean

Delphi:

Function WritePortWord(hDevice : Integer; nPort:LongWord; Value:Word):Boolean;
StdCall; External 'PCI2307' Name 'WritePortWord';

LabView:



功能：以双字(16Bit)方式写 I/O 端口

参数：

hDevice 设备对象句柄,它应由 CreateDevice 创建。

nPort 设备的 I/O 端口号。(本设备应使用 GetDeviceAddr 函数返回的 PhysAddr 地址作为基地址)

Value 写入由 nPort 指定端口的值。

返回值: 若成功, 返回 TRUE, 否则返回 FALSE, 用户可用 GetLastError 捕获当前错误码。

相关函数: [CreateDevice](#) [WritePortByte](#)
[WritePortWord](#) [WritePortULong](#)
[ReadPortByte](#) [ReadPortWord](#)
[ReleaseDevice](#)

3 以四字节(32Bit)方式写 I/O 端口

Visual C++ & C++ Builder:

BOOL WritePortULong(HANDLE hDevice, UINT nPort, ULONG Value)

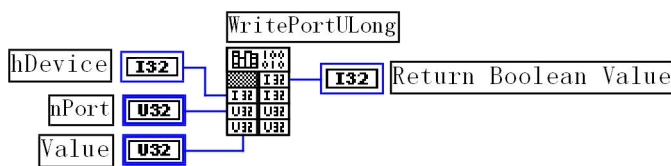
Visual Basic:

Declare Function WritePortULong Lib "PCI2307" (ByVal hDevice As Long, _
 ByVal nPort As Long, _
 ByVal Value As Long) As Boolean

Delphi:

Function WritePortULong(hDevice : Integer; nPort:LongWord; Value:LongWord):Boolean;
 StdCall; External 'PCI2307' Name 'WritePortULong';

LabView:



功能: 以四字节(32Bit)方式写 I/O 端口

参数:

hDevice 设备对象句柄,它应由 CreateDevice 或 CreateDeviceEx 创建。

nPort 设备的 I/O 端口号。(本设备应使用 GetDeviceAddr 函数返回的 PhysAddr 地址作为基地址)

Value 写入由 nPort 指定端口的值。

返回值: 若成功, 返回 TRUE, 否则返回 FALSE, 用户可用 GetLastError 捕获当前错误码。

相关函数: [CreateDevice](#) [WritePortByte](#)
[WritePortWord](#) [WritePortULong](#)
[ReadPortByte](#) [ReadPortWord](#)
[ReleaseDevice](#)

4 以单字节(8Bit)方式读 I/O 端口

Visual C++ & C++ Builder:

BYTE ReadPortByte(HANDLE hDevice, UINT nPort)

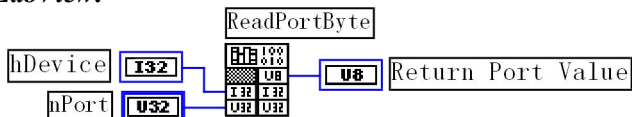
Visual Basic:

Declare Function ReadPortByte Lib "PCI2307" (ByVal hDevice As Long, _
 ByVal nPort As Long) As Byte

Delphi:

Function ReadPortByte(hDevice : Integer; nPort:LongWord):Byte;
 StdCall; External 'PCI2307' Name 'ReadPortByte';

LabView:



功能: 以单字节(8Bit)方式读 I/O 端口

参数:

hDevice 设备对象句柄,它应由 CreateDevice 或 CreateDeviceEx 创建。

nPort 设备的 I/O 端口号。(本设备应使用 GetDeviceAddr 函数返回的 PhysAddr 地址作为基地址)

返回值: 返回由 nPort 指定的端口的值

相关函数: [CreateDevice](#) [WritePortByte](#)

[WritePortWord](#)
[ReadPortByte](#)
[ReleaseDevice](#)

[WritePortULong](#)
[ReadPortWord](#)

5 以双字节(16Bit)方式读 I/O 端口

Visual C++ & C++ Builder:

WORD ReadPortWord(HANDLE hDevice, UINT nPort)

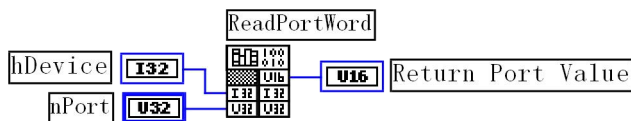
Visual Basic:

Declare Function ReadPortWord Lib "PCI2307" (ByVal hDevice As Long, _
 ByVal nPort As Long) As Integer

Delphi:

Function ReadPortWord(hDevice : Integer; nPort:LongWord):Word;
 StdCall; External 'PCI2307' Name 'ReadPortWord';

LabView:



功能：以双字节(16Bit)方式读 I/O 端口

参数:

hDevice 设备对象句柄,它应由 CreateDevice 或 CreateDeviceEx 创建。

nPort 设备的 I/O 端口号。(本设备应使用 GetDeviceAddr 函数返回的 PhysAddr 地址作为基地址)

返回值：返回由 nPort 指定的端口的值

相关函数：[CreateDevice](#) [WritePortByte](#)
[WritePortWord](#) [WritePortULong](#)
[ReadPortByte](#) [ReadPortWord](#)
[ReleaseDevice](#)

6 以四字节(32Bit)方式读 I/O 端口

Visual C++ & C++ Builder:

DWORD ReadPortULong(HANDLE hDevice, UINT nPort)

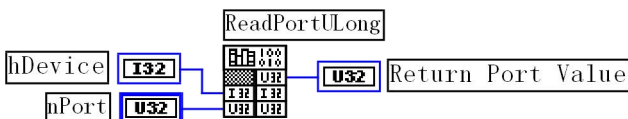
Visual Basic:

Declare Function ReadPortULong Lib "PCI2307" (ByVal hDevice As Long, _
 ByVal nPort As Long) As Long

Delphi:

Function ReadPortULong(hDevice : Integer; nPort:LongWord):LongWord;
 StdCall; External 'PCI2307' Name 'ReadPortULong';

LabView:



功能：以四字节(32Bit)方式读 I/O 端口

参数:

hDevice 设备对象句柄,它应由 CreateDevice 创建。

nPort 设备的 I/O 端口号。(本设备应使用 GetDeviceAddr 函数返回的 PhysAddr 地址作为基地址)

返回值：返回由 nPort 指定端口的值

相关函数：[CreateDevice](#) [WritePortByte](#)
[WritePortWord](#) [WritePortULong](#)
[ReadPortByte](#) [ReadPortWord](#)

第五章 共用函数介绍

这部分函数不参与本设备的实际操作，它只是为您编写数据采集与处理程序时的有力手段，使您编写应用程序更容易，使您的应用程序更高效。

第一节 公用接口函数列表

函数名	函数功能	备注
① 创建 Visual Basic 子线程，线程数量可达 32 个以上		
CreateVBThread	在 VB 环境中建立子线程对象	在 VB 中可实现多线程
TerminateVBThread	终止 VB 的子线程	
CreateSystemEvent	创建系统内核事件对象	用于线程同步或中断
② 文件对象操作函数		
CreatFileObject	初始设备文件对象	
WriteFile	请求文件对象写用户数据到磁盘文件	
ReadFile	请求文件对象读数据到用户空间	
ReleaseFile	释放已有的文件对象	
③ ISA 总线 I/O 端口操作函数		
WritePortByte	以字节(8Bit)方式写 I/O 端口	用户程序操作端口
WritePortWord	以字(16Bit)方式写 I/O 端口	用户程序操作端口
WritePortULong	以无符号双字(32Bit)方式写 I/O 端口	用户程序操作端口
ReadPortByte	以字节(8Bit)方式读 I/O 端口	用户程序操作端口
ReadPortWord	以字(16Bit)方式读 I/O 端口	用户程序操作端口
ReadPortULong	以无符号双字(32Bit)方式读 I/O 端口	用户程序操作端口
WritePortByteEx	以字节(8Bit)方式写 I/O 端口	NT 下直接操作端口
WritePortWordEx	以字(16Bit)方式写 I/O 端口	NT 下直接操作端口
WritePortULongEx	以无符号双字(32Bit)方式写 I/O 端口	NT 下直接操作端口
ReadPortByteEx	以字节(8Bit)方式读 I/O 端口	NT 下直接操作端口
ReadPortWordEx	以字(16Bit)方式读 I/O 端口	NT 下直接操作端口
ReadPortULongEx	以无符号双字(32Bit)方式读 I/O 端口	NT 下直接操作端口
④ 其他函数		
GetDiskFreeBytes	取得指定磁盘的可用空间(字节)	适用于所有设备
DelayTimeNs	高效高精度延时函数	不消耗 CPU 时间

第二节 公用接口函数原型说明

一、创建 VB 子线程

1、在 VB 环境中,创建子线程对象,以实现多线程操作

Visual Basic

Declare Function CreateVBThread Lib "PCI2310" (hThread As Long, _
ByVal RoutineAddr As Long _
) As Boolean

功能: 该函数在 VB 环境中解决了不能实现或不能很好地实现多线程的问题.通过该函数用户可以很轻松地实现多线程操作.

参数:

hThread 若成功创建子线程, 该参数将返回所创建的子线程的句柄, 当用户操作这个子线程时将用到这个句柄, 如启动线程, 暂停线程, 以及删除线程等。

RoutineAddr 作为子线程运行的函数的地址, 在实际使用时, 请用 AddressOf 关键字取得该子线程函数的地址, 再传递给 CreateVBThread 函数。

返回值: 当成功创建子线程时, 返回 TRUE, 且所创建的子线程为挂起状态, 用户需要用 ResumeThread 函数启动它. 若失败, 则返回 FALSE, 用户可用 GetLastError 捕获当前错误码。

相关函数: [CreateVBThread](#)
[TerminateVBThread](#)

注意: RoutineAddr 指向的函数或过程必须放在 VB 的模块文件中, 如 PCI2310.Bas 文件中。

Visual Basic 程序举例:

' 在模块文件中定义子线程函数(注意参考实例)

```
Function NewRoutine() As Long ' 定义子线程函数
: ' 线程运行代码
NewRoutine = 1 ' 返回成功码
End Function
```

' 在窗体文件中创建子线程

```
:
Dim hNewThread As Long
If Not CreateVBThread(hNewThread, AddressOf NewRoutine) Then ' 创建新子线程
MsgBox "创建子线程失败"
Exit Sub
End If
ResumeThread (hNewThread) ' 启动新线程
:
```

2 在 VB 中,删除子线程对象

Visual Basic:

Declare Function TerminateVBThread Lib "PCI2310" (ByVal hThread As Long) As Boolean

功能: 在 VB 中删除由 CreateVBThread 创建的子线程对象。

参数: hThread 指向需要删除的子线程对象的句柄, 它应由 CreateVBThread 创建。

返回值: 当成功删除子线程对象时, 返回 TRUE, 否则返回 FALSE, 用户可用 GetLastError 捕获当前错误码。

相关函数: [CreateVBThread](#)
[TerminateVBThread](#)

Visual Basic 程序举例:

```
:
If Not TerminateVBThread (hNewThread) ' 终止子线程
MsgBox "创建子线程失败"
Exit Sub
End If
:
```

二、创建内核系统事件

Visual C++:

HANDLE CreateSystemEvent(void);

Visual Basic:

Declare Function CreateSystemEvent Lib " PCI2310 " () As Long

Delphi:

Function CreateSystemEvent():Integer; StdCall; External 'PCI2310' Name 'CreateSystemEvent';

LabView:



功能: 创建系统内核事件对象,它将被用于中断事件响应或数据采集线程同步事件。

参数: 无任何参数

返回值: 若成功, 返回系统内核事件对象句柄, 否则返回-1(或 INVALID_HANDLE_VALUE)。

Visual C++ 程序举例:

```

:
HANDLE hEvent;
hEvent=CreateSystemEvent ()
if(hEvent==INVALID_HANDLE_VALUE)
    MessageBox("创建内核事件失败...");
    return;    // 退出该函数
}

```

Visual Basic 程序举例:

```

:
hEvent = CreateSystemEvent() ' 创建内核事件
If hEvent = INVALID_HANDLE_VALUE Then
    MsgBox "创建事件对象失败"
    Exit Sub
End If
:

```

三、文件对象操作函数

1 初始化设备文件对象

函数原型:

Visual C++:

```

Handle CreateFileObject (
    HANDLE hDevice,
    LPCTSTR NewFileName,
    int Mode)

```

Visual Basic:

```

Declare Function CreateFileObjectLib "PCI2310" (ByVal hDevice As Long, _
    ByVal NewFileName As String, _
    ByVal Mode As Long
    ) As Long

```

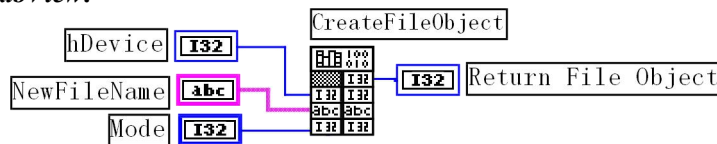
Delphi:

```

Function CreateFileObject (hDevice : Integer; NewFileName: string; Mode: Integer):LongInt;
    stdcall; external 'PCI2310' name CreateFileObject;

```

LabView:



功能: 初始化设备文件对象, 以期待 WriteFile 请求准备文件对象进行文件操作。

参数:

hDevice: 设备对象句柄,它应由 CreateDevice 创建。

NewFileName: 与新文件对象关联的磁盘文件名, 可以包括盘符和路径等信息。在 C 语言中, 其语法格式如: "C:\\PCI2310\\Data.Dat", 在 Basic 中, 其语法格式如: "C:\\PCI2310\\Data.Dat"

Mode 文件操作方式, 所用的文件操作方式控制字定义如下(可通过或指令实现多种方式并操作)

PCI2310_modeRead 只读文件方式

PCI2310_modeWrite 只写文件方式

PCI2310_modeReadWrite 既读又写文件方式

PCI2310_modeCreate 如果文件不存在, 可以创建该文件, 如果存在, 则重建此文件, 并清 0

返回值: 若成功, 则返回设备对象句柄。

相关函数：[CreateDevice](#) [CreateFileObject](#)
[WriteFile](#) [ReadFile](#)
[ReleaseFile](#)
[ReleaseDevice](#)

2 通过设备对象,往指定磁盘上写入用户空间的采样数据.

函数原型:

Visual C++:

```
BOOL WriteFile( HANDLE hDevice,
                PVOID pUserRegion,
                LONG nWriteSizeBytes)
```

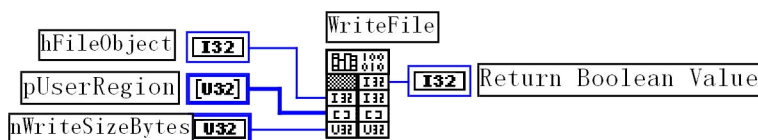
Visual Basic:

```
Declare Function WriteFile Lib "PCI2310" (ByRef hDevice As Long,
                                           ByVal pUserRegion As Integer,
                                           ByVal nWriteSizeBytes As Long,
                                           ) As Boolean
```

Delphi:

```
function WriteFile(hDevice : Integer;
                  pUserRegion:PWordArray;
                  nWriteSizeBytes: Long Word):Boolean;
  stdcall; external 'PCI2310' name 'WriteFile';
```

LabView:



功能: 通过向设备对象发送“写磁盘消息”，设备对象便会以最快的速度完成写操作。注意为了保证写入的数据是可用的，这个操作将与用户程序保持同步，但与设备对象中的环形内存池操作保持异步，以得到更高的数据吞吐量，其文件名及路径应由 CreateFileObject 函数中的 [NewFileName](#) 指定。

参数:

hDevice 设备对象句柄,它应由 CreateDevice 创建。

pUserRegion 用户数据空间地址。

nWriteSizeBytes 告诉设备对象往磁盘上一次写入数据的长度(以字节为单位)

返回值: 若成功，则返回 TRUE，否则返回 FALSE，用户可以用 GetLastError 捕获错误码。

相关函数：[CreateDevice](#) [CreateFileObject](#)
[WriteFile](#) [ReadFile](#)
[ReleaseFile](#)
[ReleaseDevice](#)

注意: 它写入磁盘中的数据将来自于 InitDeviceAD 中的 ADBuffer 用户内存区。

3 通过设备对象,从指定磁盘文件中读采样数据.

函数原型:

Visual C++:

```
BOOL ReadFile( HANDLE hDevice,
               PVOID pFileUserRegion,
               LONG OffsetBytes,
               LONG nReadSizeBytes)
```

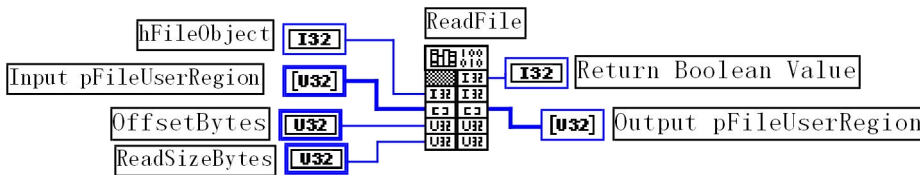
Visual Basic:

```
Declare Function ReadFile Lib "PCI2310" (ByVal hDevice As Long,
                                           ByRef pFileUserRegion As Integer,
                                           ByVal OffsetBytes As Long,
                                           ByVal ReadSizeBytes As Long,
                                           ) As Boolean
```

Delphi:

```
Function ReadFile(hDevice : Integer;
                  pUserRegion:PWordArray;
```

```
OffsetBytes:LongWord;
nReadSizeBytes:LongWord);Boolean;
stdcall; external 'PCI2310' name 'ReadFile';
```

LabView:

功能: 通过向设备对象发送写磁盘消息, 设备对象便会以最快的速度完成写操作。注意为了保证写入的数据是可用的, 这个操作将与用户程序保持同步, 但与设备对象中的环形内存池操作保持异步, 以得到更高的数据吞吐量, 其文件名及路径应由 CreateFileObject 函数中的 [NewFileName](#) 指定。

参数:

hDevice 设备对象句柄, 它应由 CreateDevice 创建。

pFileUserRegion 用于接受文件数据的用户缓冲区指针。

OffsetBytes 指定从文件始端起所偏移的读位置。

ReadSizeBytes 告诉设备对象从磁盘上一次读入数据的长度(以字为单位), 其取值范围为[1, 65535]。

返回值: 若成功, 则返回 TRUE, 否则返回 FALSE, 用户可以用 GetLastError 捕获错误码。

相关函数: [CreateDevice](#) [CreateFileObject](#)
[WriteFile](#) [ReadFile](#)
[ReleaseFile](#)
[ReleaseDevice](#)

注意: 它读出磁盘的数据将放置于 *InitDeviceAD* 中的 *ADBuffer* 用户内存区前端。

4 释放设备文件对象

函数原型:

Visual C++:

[BOOL ReleaseFile\(HANDLE hDevice\)](#)

Visual Basic:

[Declare Function ReleaseFile Lib "PCI2310" \(ByVal hDevice As Long\) as Boolean](#)

Delphi:

[Function ReleaseFile\(hDevice : Integer\):Boolean;](#)
[stdcall; external 'PCI2310' name 'ReleaseFile';](#)

LabView:



功能: 释放设备文件对象。

参数: **hDevice** 设备对象句柄, 它应由 CreateDevice 创建。

返回值: 若成功, 则返回 TRUE, 否则返回 FALSE, 用户可以用 GetLastError 捕获错误码。

相关函数: [CreateDevice](#) [CreateFileObject](#)
[WriteFile](#) [ReadFile](#)
[ReleaseFile](#)
[ReleaseDevice](#)

四、I/O 端口读写函数**1 以单字节(8Bit)方式写 I/O 端口**

Visual C++ & C++ Builder:

[BOOL WritePortByte \(HANDLE hDevice, UINT nPort, BYTE Value\)](#)

[BOOL WritePortByteEx \(UINT nPort, BYTE Value\)](#)

(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

Visual Basic:

[Declare Function WritePortByte Lib "PCI2310" \(ByVal hDevice As Long, _](#)

```

ByVal nPort As Long, _
ByVal Value As Byte) As Boolean
Declare Function WritePortByte Lib "PCI2310" (ByVal nPort As Long, _
ByVal Value As Byte) As Boolean
(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

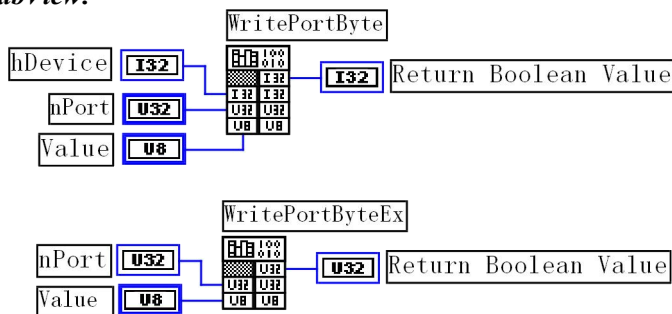
```

Delphi:

```

Function WritePortByte(hDevice : Integer; nPort:LongWord; Value:Byte):Boolean;
StdCall; External 'PCI2310' Name 'WritePortByte';
Function WritePortByteEx(nPort:LongWord; Value:Byte):Boolean;
StdCall; External 'PCI2310' Name 'WritePortByte';

```

LabView:

功能: 以单字节(8Bit)方式写 I/O 端口

参数:

hDevice 设备对象句柄,它应由 CreateDevice 或 CreateDeviceEx 创建。

nPort 设备的 I/O 端口号。

Value 写入由 nPort 指定端口的值。

返回值: 若成功, 返回 TRUE, 否则返回 FALSE, 用户可用 GetLastError 捕获当前错误码。

相关函数:

CreateDevice	WritePortByte
WritePortWord	WritePortULong
ReadPortByte	ReadPortWord
WritePortByteEx	WritePortWordEx
WritePortULongEx	ReadPortByteEx
ReadPortWordEx	ReadPortULongEx
ReadPortULongEx	ReleaseDevice

2 以双字(16Bit)方式写 I/O 端口**Visual C++ & C++ Builder:**

```

BOOL WritePortWord (HANDLE hDevice, UINT nPort, WORD Value)

```

```

BOOL WritePortWordEx (UINT nPort, WORD Value)

```

(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

Visual Basic:

```

Declare Function WritePortWord Lib "PCI2310" (ByVal hDevice As Long, _
ByVal nPort As Long, _
ByVal Value As Integer) As Boolean

```

```

Declare Function WritePortWordEx Lib "PCI2310" (ByVal nPort As Long, _
ByVal Value As Integer) As Boolean

```

(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

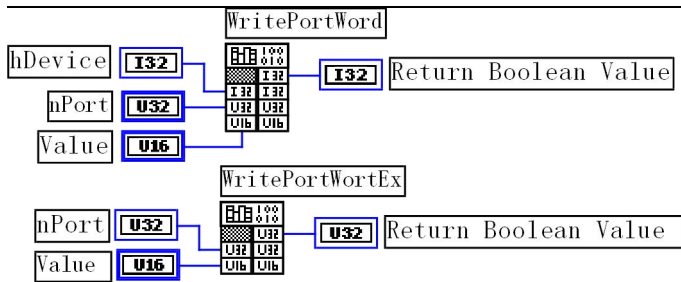
Delphi:

```

Function WritePortWord(hDevice : Integer; nPort:LongWord; Value:Word):Boolean;
StdCall; External 'PCI2310' Name 'WritePortWord';
Function WritePortWordEx(nPort:LongWord; Value:Word):Boolean;
StdCall; External 'PCI2310' Name 'WritePortWord';

```

LabView:



功能: 以双字(16Bit)方式写 I/O 端口

参数:

`hDevice` 设备对象句柄,它应由 `CreateDevice` 创建。

`nPort` 设备的 I/O 端口号。

`Value` 写入由 `nPort` 指定端口的值。

返回值: 若成功, 返回 TRUE, 否则返回 FALSE, 用户可用 `GetLastError` 捕获当前错误码。

相关函数: [CreateDevice](#) [WritePortByte](#)
[WritePortWord](#) [WritePortULong](#)
[ReadPortByte](#) [ReadPortWord](#)
[WritePortByteEx](#) [WritePortWordEx](#)
[WritePortULongEx](#) [ReadPortByteEx](#)
[ReadPortWordEx](#) [ReadPortULongEx](#)
[ReadPortULongEx](#) [ReleaseDevice](#)

3 以四字节(32Bit)方式写 I/O 端口

Visual C++ & C++ Builder:

`BOOL WritePortULong(HANDLE hDevice, UINT nPort, ULONG Value)`

`BOOL WritePortULongEx(UINT nPort, ULONG Value)`

(在 NT 用户模式程序中直接访问 I/O 端口)

Visual Basic:

`Declare Function WritePortULong Lib "PCI2310" (ByVal hDevice As Long, _
 ByVal nPort As Long, _
 ByVal Value As Long) As Boolean`

`Declare Function WritePortULongEx Lib "PCI2310"(ByVal nPort As Long, _
 ByVal Value As Long) As Boolean`

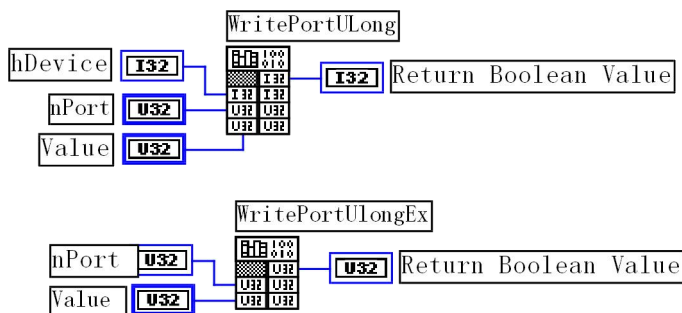
(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

Delphi:

`Function WritePortULong(hDevice : Integer; nPort:LongWord; Value:LongWord):Boolean;
 StdCall; External 'PCI2310' Name 'WritePortULong';`

`Function WritePortULongEx(nPort:LongWord; Value:LongWord):Boolean;
 StdCall; External 'PCI2310' Name 'WritePortULong';`

LabView:



功能: 以四字节(32Bit)方式写 I/O 端口

参数:

`hDevice` 设备对象句柄,它应由 `CreateDevice`。

`nPort` 设备的 I/O 端口号。

`Value` 写入由 `nPort` 指定端口的值。

返回值: 若成功, 返回 TRUE, 否则返回 FALSE, 用户可用 GetLastError 捕获当前错误码。

相关函数:

CreateDevice	WritePortByte
WritePortWord	WritePortULong
ReadPortByte	ReadPortWord
WritePortByteEx	WritePortWordEx
WritePortULongEx	ReadPortByteEx
ReadPortWordEx	ReadPortULongEx
ReadPortULongEx	ReleaseDevice

4 以单字节(8Bit)方式读 I/O 端口

Visual C++ & C++ Builder:

BYTE ReadPortByte(HANDLE hDevice, UINT nPort)

BYTE ReadPortByteEx(UINT nPort)

(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

Visual Basic:

Declare Function ReadPortByte Lib "PCI2310" (ByVal hDevice As Long, _
ByVal nPort As Long) As Byte

Declare Function ReadPortByteEx Lib "PCI2310" (ByVal nPort As Long) As Byte

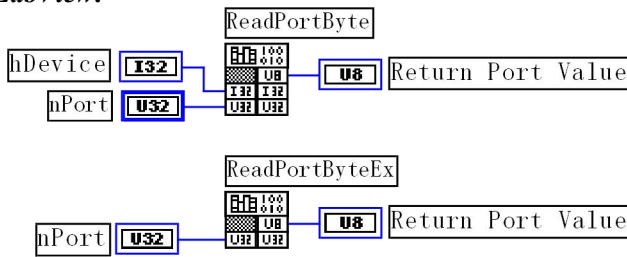
(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

Delphi:

Function ReadPortByte(hDevice : Integer; nPort:LongWord):Byte;
StdCall; External 'PCI2310' Name 'ReadPortByte';

Function ReadPortByteEx(nPort:LongWord):Byte;
StdCall; External 'PCI2310' Name 'ReadPortByte';

LabView:



功能: 以单字节(8Bit)方式读 I/O 端口

参数:

hDevice 设备对象句柄,它应由 CreateDevice 或 CreateDeviceEx 创建。

nPort 设备的 I/O 端口号。

返回值: 返回由 nPort 指定的端口的值

相关函数:

CreateDevice	WritePortByte
WritePortWord	WritePortULong
ReadPortByte	ReadPortWord
WritePortByteEx	WritePortWordEx
WritePortULongEx	ReadPortByteEx
ReadPortWordEx	ReadPortULongEx
ReadPortULongEx	ReleaseDevice

5 以双字节(16Bit)方式读 I/O 端口

Visual C++ & C++ Builder:

WORD ReadPortWord(HANDLE hDevice, UINT nPort)

WORD ReadPortWordEx(UINT nPort)

(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

Visual Basic:

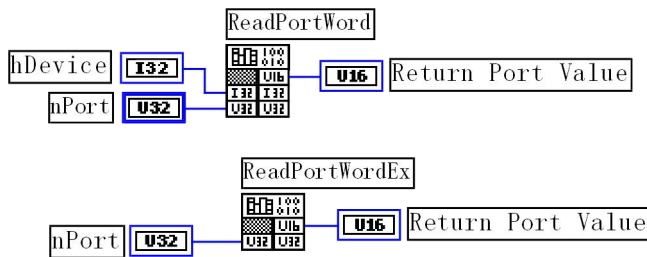
Declare Function ReadPortWord Lib "PCI2310" (ByVal hDevice As Long, _
ByVal nPort As Long) As Integer

Declare Function ReadPortWordEx Lib "PCI2310" (ByVal hDevice As Long, _
ByVal nPort As Long) As Integer

(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

Delphi:

```
Function ReadPortWord(hDevice : Integer; nPort:LongWord):Word;
    StdCall; External 'PCI2310' Name 'ReadPortWord';
Function ReadPortWordEx(nPort:LongWord):Word;
    StdCall; External 'PCI2310' Name 'ReadPortWord';
```

LabView:

功能: 以双字节(16Bit)方式读 I/O 端口

参数:

hDevice 设备对象句柄,它应由 CreateDevice 或 CreateDeviceEx 创建。

nPort 设备的 I/O 端口号。

返回值: 返回由 nPort 指定的端口的值

相关函数:

CreateDevice	WritePortByte
WritePortWord	WritePortULong
ReadPortByte	ReadPortWord
WritePortByteEx	WritePortWordEx
WritePortULongEx	ReadPortByteEx
ReadPortWordEx	ReadPortULongEx
ReadPortULongEx	ReleaseDevice

6 以四字节(32Bit)方式读 I/O 端口**Visual C++ & C++ Builder:**

```
WORD ReadPortULong(HANDLE hDevice, UINT nPort)
```

```
WORD ReadPortULongEx(UINT nPort)
```

(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

Visual Basic:

```
Declare Function ReadPortULong Lib "PCI2310" (ByVal hDevice As Long, _
    ByVal nPort As Long) As Long
```

```
Declare Function ReadPortULongEx Lib "PCI2310" (ByVal nPort As Long) As Long
```

(在 Windows NT/2000 用户模式程序中直接访问 I/O 端口)

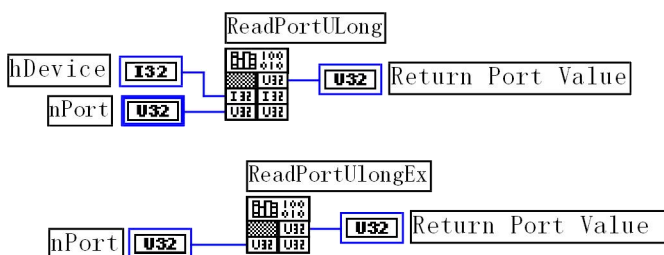
Delphi:

```
Function ReadPortULong(hDevice : Integer; nPort:LongWord):LongWord;
```

```
    StdCall; External 'PCI2310' Name 'ReadPortULong';
```

```
Function ReadPortULongEx(nPort:LongWord):LongWord;
```

```
    StdCall; External 'PCI2310' Name 'ReadPortULong';
```

LabView:

功能: 以四字节(32Bit)方式读 I/O 端口

参数:

hDevice 设备对象句柄,它应由 CreateDevice 创建。

nPort 设备的 I/O 端口号。

返回值: 返回由 nPort 指定端口的值

相关函数:

CreateDevice	WritePortByte
WritePortWord	WritePortULong
ReadPortByte	ReadPortWord
WritePortByteEx	WritePortWordEx
WritePortULongEx	ReadPortByteEx
ReadPortWordEx	ReadPortULongEx
ReadPortULongEx	ReleaseDevice

第三节 其他函数

一、取得指定磁盘的可用空间

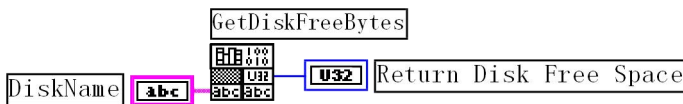
Visual C++:

LONG GetDiskFreeBytes (LPCTSTR DiskName)

Visual Basic:

Declare Function GetDiskFreeBytes Lib "PCI2310" (ByVal DiskName As String) As Boolean

LabView:



功能: 取得指定磁盘的可用剩余空间(以字为单位)。

参数: 需要访问的盘符,若为 C 盘为"C:\", D 盘为"D:\", 以此类推。

返回值: 若成功, 返回大于或等于 0 的长整型值,否则返回负值, 用户可用 GetLastError 捕获错误码

二、高效高精度延时

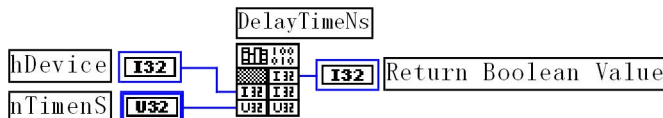
Visual C++:

BOOL DelayTimeNs(HANDLE hDevice, LONG nTimenS)

Visual Basic:

Declare Function DelayTimeNs Lib "PCI2310" (ByVal hDevice As Long, ByVal nTimenS As Long) As Boolean

LabView:



功能: 以不消耗 CPU 时间为前提, 提供精确到纳秒级的延时操作, 单位为 100 纳秒。如果应用在线程中, 它会使所在的线程自动睡眠指定时间, 且将这段时间抛给其他线程及进程, 再自动被唤醒。如果应用在进程中, 它会使所在进程自动睡眠指定时间, 且将这段时间抛给整个系统及其他进程。特别是在等待某些状态, 但状态未达到需要轮询时, 可以使用该函数延时等待状态, 而不必消耗大量 CPU 时间。因此, 使用此函数可能提高应用程序的整体性能。

参数:

hDevice 设备对象句柄, 它应由 CreateDevice 决定。

nTimenS 时间常数。单位为 100 纳秒。如果用户要求延时 100 微秒, 则用下列公式换算:

$$\text{时间常数} = \text{延时时间(纳秒)} / 100(\text{纳秒})$$

可得到时间常数为 1000, 即该参数应等于 1000。

返回值: 若成功, 返回 TRUE, 否则返回 FALSE, 用户可用 GetLastError 捕获错误码

注意事项:

此函数现只提供了 Windows NT、Windows 2000 版, 对于 Win95、Win98 等系统可能在以后会提供。

用户若要精确延时, 切莫以多次调用为累加。比如您若延时 100 微秒, 而用循环方式循环执行下面指令 100 次则是错的:

```
DelayTimeNs(hDevice, 10);
```

因为此函数本身就有时间消耗, 它在调用时, 首先有一个从用户模式到内核模式的切换, 还有压栈的过程, 然后延时后还有从内核模式到用户模式的切换及出栈过程, 这可能会花掉 1-3 微秒。

为正确的实现方法是一次调用此函数：

```
DelayTimeNs(hDevice, 1000);
```

第六章 硬件参数结构

第一节、AD 硬件参数结构 (PCI2310_PARA_AD)

Visual C++ & C++Builder:

```
typedef struct _PCI2310_PARA_AD // 板卡各参数值
{
    DWORD FirstChannel; // 首通道
    DWORD LastChannel; // 末通道
    DWORD Frequency; // 采集频率(Hz)
    DWORD Gains; // 采集增益
    DWORD TriggerSource; // AD 触发转换方式
} PCI2310_PARA_AD,* PPCI2310_PARA_AD;
```

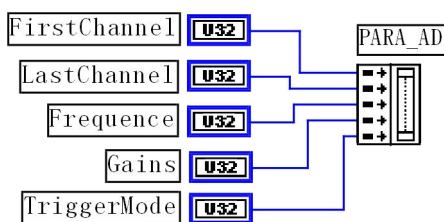
Visual Basic:

```
Private Type PCI2310_PARA_AD
    FirstChannel As Long ' 首通道号
    LastChannel As Long
    Frequency As Long
    Gains As Long
    TriggerSource As Long
End Type
```

Delphi:

```
type // 定义结构体数据类型
    PPCI2310_PARA_AD = ^PCI2310_PARA_AD; // 指针类型结构
    PCI2310_PARA_AD = record // 标记为记录型
        FirstChannel: DWORD;
        LastChannel: DWORD;
        Frequency: DWORD;
        Gains: DWORD;
        TriggerSource : DWORD
    end;
End;
```

LabView:



首先请您关注一下这个结构与前面 ISA 总线部分中的两个结构 PARA、PARAEX 比较，该结构实在太简短了。其原因就是 PCI 设备是系统全自动管理的设备，什么端口地址，中断号，DMA 等将与 PCI 设备的用户永远告别，一句话 PCI 设备是一种更易于管理和使用的设备。

硬件参数说明：此结构主要用于设定设备硬件参数值，用这个参数结构对设备进行硬件配置完全由 InitDeviceProAD 或 InitDeviceIntAD 函数完成。

FirstChannel: AD 采样首通道值，取值范围应根据设备的总通道数设定，本设备的通道取值为：0~31，要求首通道小于或等于末通道。

LastChannel: AD 采样末通道值，取值范围应根据设备的总通道数设定，本设备的通道取值为：0~31，要求末通道大于或等于首通道。

注：当首通道和末通道相等时，即为单通道采集。

Frequency AD 采样频率，单位 Hz，其范围应根据具体的设备而定，但其最小值为 1Hz。切忌不能等于 0，本设备的 AD 采样频率取值范围为[1, 100000]

Gains 硬件采样增益, 其取值分别为 1、2、3、4、5。如果 AD 转换器前使用的是 PGA202 放大器, 则这些值分别表示放大倍数为 1 倍、10 倍、100 倍、1000、10000 倍, 如果 AD 转换器前使用的是 PGA203 放大器, 则其值分别表示放大倍数为 1、2、4、8、16 倍。除了 1、2、3、4、5 四个值以外, 其它值均为非法值。

TriggerSource AD 采样触发方式, 若等于常量 PCI2310_IN_TRIGGER 则为内部定时触发, 若等于常量 PCI2310_OUT_TRIGGER 则为外触发。两种方式的主要区别是: 外触发是当设备被 InitDeviceProAD(InitDeviceIntAD) 函数初始化就绪后, 并没有立即启动 AD 采集, 仅当外接管脚 TR(在 37 芯 D 型头上)上有一个由低至高变化的上升沿(TTL 电平)时, AD 转换器便被启动, 且按用户预先设定的采样频率由板上的硬件定时器定时触发 AD 等间隔转换每一个 AD 数据, 此后, 除非用户重新初始化设备, 否则, TR 管脚上所产生的新的上升沿信号并不影响 AD 转换进程。因此如果用户不断的使下一个触发信号有效, 那么您必须在每一个外触发信号到来之前重新初始化设备。而对于内触发方式则与 TR 管脚上的信号无任何关系, 它是在用户调用 InitDeviceAD 函数初始化设备时, 由这个函数中的最后一条软件指令立即启动 AD 转换器, AD 转换器便以指定的频率由板上定时器等间隔定时触发 AD 转换。指两种触发方式的应用场合: 对于瞬间变化(持续时间短、变化频率较高)、或随机性较强的信号的测量和采样。或是需要精确定位所要采集的一批 AD 数据中的第一个点的时间轴, 那么您需要使用外触发方式。而对于持续变化时间较长, 不需要精确定位信号起点的信号, 则一般使用内触发方式。

相关函数: [InitDeviceProAD](#) [InitDeviceIntAD](#)
 [SaveParameter](#) [LoadParameter](#)

第二节、用于数字 I/O 输出参数 (PCI2310_PARA_DO)

Visual C++ & C++Builder:

```
typedef struct _PCI2310_PARA_DO        // 数字量输出参数
{
    BYTE DO0;            // 0 通道
    BYTE DO1;            // 1 通道
    BYTE DO2;            // 2 通道
    BYTE DO3;            // 3 通道
    BYTE DO4;            // 4 通道
    BYTE DO5;            // 5 通道
    BYTE DO6;            // 6 通道
    BYTE DO7;            // 7 通道
    BYTE DO8;            // 8 通道
    BYTE DO9;            // 9 通道
    BYTE DO10;           // 10 通道
    BYTE DO11;           // 11 通道
    BYTE DO12;           // 12 通道
    BYTE DO13;           // 13 通道
    BYTE DO14;           // 14 通道
    BYTE DO15;           // 15 通道
} PCI2310_PARA_DO,*PPCI2310_PARA_DO;
```

Visual Basic:

```
Type PCI2310_PARA_DO
    DO0 As Byte ' 0 通道
    DO1 As Byte ' 1 通道
    DO2 As Byte ' 2 通道
    DO3 As Byte ' 3 通道
    DO4 As Byte ' 4 通道
    DO5 As Byte ' 5 通道
    DO6 As Byte ' 6 通道
    DO7 As Byte ' 7 通道
    DO8 As Byte ' 8 通道
    DO9 As Byte ' 9 通道
    DO10 As Byte ' 10 通道
    DO11 As Byte ' 11 通道
    DO12 As Byte ' 12 通道
```

DO13 As Byte ‘ 13 通道
DO14 As Byte ‘ 14 通道
DO15 As Byte ‘ 15 通道

End Type

Delphi:

Type // 定义结构体数据类型

```
PPCI2310_PARA_DO = ^PCI2310_PARA_DO; // 指针类型结构
```

```
PCI2310_PARA_DO = record // 标记为记录型
```

```
DO0: Byte; // 0 通道
```

```
DO1: Byte; // 1 通道
```

```
DO2: Byte; // 2 通道
```

```
DO3: Byte; // 3 通道
```

```
DO4: Byte; // 4 通道
```

```
DO5: Byte; // 5 通道
```

```
DO6: Byte; // 6 通道
```

```
DO7: Byte; // 7 通道
```

```
DO8: Byte; // 8 通道
```

```
DO9: Byte; // 9 通道
```

```
DO10: Byte; // 10 通道
```

```
DO11: Byte; // 11 通道
```

```
DO12: Byte; // 12 通道
```

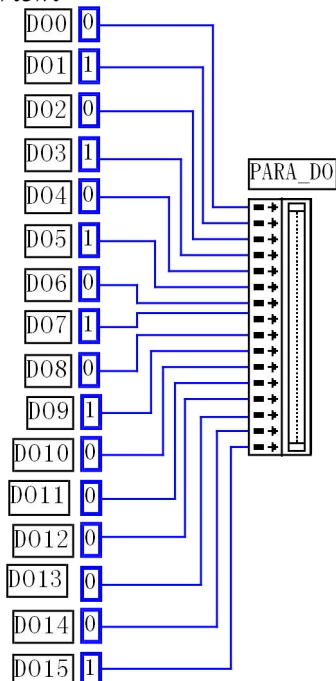
```
DO13: Byte; // 13 通道
```

```
DO14: Byte; // 14 通道
```

```
DO15: Byte; // 15 通道
```

End;

LabView:



该参数结构的使用极大的方便了不熟悉硬件端口控制和二进制位操作的用户。在这里您不需要了解技术细节，只需要象 Visual Basic 中的属性操作那样，只需要有简单的进行属性赋值，然后执行 SetDeviceDO 即可完成数字量输出。注意关于 LabView 的参数定义，他最主要表达了在 LabView 环境中怎样使用 SetDeviceDO 实现开关量输出操作的基本实现方法。在用户实际使用中，您可以将左边的常量图标换成开关控件图标等，以实现动态改变开关量输出状态。但需要注意的是开关控件图标 (xxx Switch) 输出的值是布尔变量，因此在开关控件图标与 PPCI2310_PARA_DO 之间，应使用 Boolean To (0,1) 逻辑转换控件，即先将布尔变量转换成 0 或 1 的整型值，再将这个整型值传递给 PPCI2310_PARA_DO，详见开关量输入输出 LabView 演示部分。

其每一个成员变量对应于相应的 DO 通道，即 DO0-DO15 分别对应于 DO 通道 0-15。且这些成员变量只能被赋值为“0”或“1”数值。“0”代表“关”状态或“低”状态，“1”代表“开”状态或“高”状态。

第三节、用于数字 I/O 输入参数 (PCI2310_PARA_DI)

```
typedef struct _PCI2310_PARA_DI      // 数字量输入参数
{
    BYTE DI0;           // 0 通道
    BYTE DI1;           // 1 通道
    BYTE DI2;           // 2 通道
    BYTE DI3;           // 3 通道
    BYTE DI4;           // 4 通道
    BYTE DI5;           // 5 通道
    BYTE DI6;           // 6 通道
    BYTE DI7;           // 7 通道
    BYTE DI8;           // 8 通道
    BYTE DI9;           // 9 通道
    BYTE DI10;          // 10 通道
    BYTE DI11;          // 11 通道
    BYTE DI12;          // 12 通道
    BYTE DI13;          // 13 通道
    BYTE DI14;          // 14 通道
    BYTE DI15;          // 15 通道
} PCI2310_PARA_DI,*PPCI2310_PARA_DI;
```

Visual Basic:

```
Type PCI2310_PARA_DI
    DI0 As Byte ' 0 通道
    DI1 As Byte ' 1 通道
    DI2 As Byte ' 2 通道
    DI3 As Byte ' 3 通道
    DI4 As Byte ' 4 通道
    DI5 As Byte ' 5 通道
    DI6 As Byte ' 6 通道
    DI7 As Byte ' 7 通道
    DI8 As Byte ' 8 通道
    DI9 As Byte ' 9 通道
    DI10 As Byte ' 10 通道
    DI11 As Byte ' 11 通道
    DI12 As Byte ' 12 通道
    DI13 As Byte ' 13 通道
    DI14 As Byte ' 14 通道
    DI15 As Byte ' 15 通道
```

End Type

Delphi:

```
Type // 定义结构体数据类型
PPCI2310_PARA_DI = ^PCI2310_PARA_DI; // 指针类型结构
PCI2310_PARA_DI = record // 标记为记录型
    DI0: Byte; // 0 通道
    DI1: Byte; // 1 通道
    DI2: Byte; // 2 通道
    DI3: Byte; // 3 通道
    DI4: Byte; // 4 通道
    DI5: Byte; // 5 通道
    DI6: Byte; // 6 通道
    DI7: Byte; // 7 通道
    DI8: Byte; // 8 通道
    DI9: Byte; // 9 通道
    DI10: Byte; // 10 通道
```

```

DI11: Byte; // 11 通道
DI12: Byte; // 12 通道
DI13: Byte; // 13 通道
DI14: Byte; // 14 通道
DI15: Byte; // 15 通道
End;

```

该参数结构的使用极大的方便了不熟悉硬件端口控制和二进制位操作的用户。在这里您不需要了解技术细节，只需要执行 GetDeviceDI 即可完成数字量输入操作。然后象 Visual Basic 中的属性操作那样，简单的进行属性成员分析即可确定各路状态。

关于 LabView 的参数，由于需要的是返回值，因此根据 LabView 的特点，应分配一个 16 字节的内存单元，每一个字节的内存单元对应相应位置上的开关量输入状态。要使用这些状态，则应在 GetDeviceDI 之后，将存放实际的当前开关量状态的内存单元用 Index Array 数组操作控件将其每一路开关量状态分离出来，即可确定每一路开关输入状态。详见开关量输入输出 LabView 演示部分。

其每一个成员变量对应于相应的 DI 通道，即 DI0-DI15 分别对应于 DI 通道 0-15。且这些成员变量只能是“0”或“1”数值。“0”代表“关”状态或“低”状态，“1”代表“开”状态或“高”状态。

第七章 数据格式转换与排列规则

第一节、如何将 AD 原始数据 LSB 转换电压值 Volt

在换算过程中弄清模板精度（即 Bit 位数）是很关键的，因为它决定了 LSB 数码的总宽度 CountLSB。比如 8 位的模板 CountLSB 为 256。而本设备的 AD 为 12 位，则为 4096。其他类型同理均按 $2^n = \text{LSB 总数}$ （n 为 Bit 位数）换算即可。

第一节 AD 原码 LSB 数据如何转换成电压值？

设从设备上读入的某个 AD 原码数据为变量 Lsb，其对应的电压为变量 Volt（单位 mV）

量程(伏)	计算机语言换算公式	Volt 取值范围 mV
±5000mV	$\text{Volt} = \text{Lsb} * (10000 / 4096) - 5000$ 或 $\text{Volt} = (\text{Lsb} - 2048) * (10000 / 4096)$	[-5000, +5000]

注意：以上所列 Lsb 必须是从设备上读入的 AD 数(设为 ADBuffer[0])经最高 4 位屏蔽后得到的，各种语言的求反指令如下：

Visual C++&C++Builder:

```
Lsb = ADBuffer[0]&0xFFF
```

Visual Basic:

```
Lsb = ADBuffer(0)And &HFFF
```

Delphi:

```
Lsb := ADBuffer[0]And $FFF
```

换算举例：（设量程为 ±5000mV，这里只转换第一个点）

Visual C++&C++Builder:

```
WORD Lsb; // 定义存放标准 LSB 原码的变量
```

```
float Volt; // 定义存放转换后的电压值的变量
```

```
Lsb = ADBuffer[0] & 0x0FFF; // 取得标准 LSB 原码
```

```
Volt = Lsb * (10000.0/4096) - 5000.0; // 用 LSB 原码与单位电压值相乘求得实际电压值
```

Visual Basic:

```
Dim Lsb As Integer ' 定义存放标准 LSB 原码的变量
```

```
Dim Volt As Single ' 定义存放转换后的电压值的变量
```

```
Lsb = ADBuffer(0) And &H0FFF ' 取得标准 LSB 原码
```

```
Volt = Lsb * (10000.0/4096) - 5000.0 ' 用 LSB 原码与单位电压值相乘求得实际电压值
```

Delphi:

```
Lsb : Integer; // 定义存放标准 LSB 原码的变量
```

```
Volt : Single; // 定义存放转换后的电压值的变量
```

```
Lsb := ADBuffer[0] And $0FFF; // 取得标准 LSB 原码
```

```
Volt := Lsb * (10000.0/4096) - 5000.0; // 用 LSB 原码与单位电压值相乘求得实际电压值
```


但建议用户在实际换算过程中，为了提高运算效率和数据处理能力，可以将“10000.0/4096”这部分运算提出先事先赋给一个变量或将最终值设计成常量（假如设为 PerLsbVolt），然后再用这个值去反复的乘以 AD 的不同 LSB 原码。但由于“10000.0/4096”的运算为浮点数运算，其结果 2.44140625...mV 也为浮点数，大家众所周知，浮点数运算一般是非常耗时的，所以我们可以用微伏做单位，即将 2.44140625...乘以 1000 最后取得的值为 2441.40625uV，再取整，即为 2441uV，最后再以这个 2441uV 单位电压值去乘以 AD 的 LSB 原码，即可以整型数据相乘的方式求得相应电压值。举例说明：以±5V 为例

Visual C++&C++Builder:

```
LONG PerLsbVolt; // 定义一个足够宽度的整型变量存放单位电压值(uV)
LONG Volt;      // 定义一个足够宽度的整型变量存放转换后的实际电压值(uV)
PerLsbVolt = (LONG)((10000.0/4096) * 1000); // 取得单位电压值，并以 uV 为单位将其转换成整型值
Volt = (ADBuffer[0] & 0x0FFF) * PerLsbVolt-5000*1000; // 用整型运算方式求得相应电压值(uV)
```

Visual Basic:

```
Dim PerLsbVolt As Long ' 定义一个足够宽度的整型变量存放单位电压值(uV)
Dim Volt As Long ' 定义一个足够宽度的整型变量存放转换后的实际电压值(uV)
PerLsbVolt = (Int)((10000.0/4096) * 1000) ' 取得单位电压值，并以 uV 为单位将其转换成整型值
Volt = (ADBuffer(0) And &H0FFF) * PerLsbVolt-5000*1000 '用整型运算方式求得相应电压值(uV)
```

Delphi:

```
PerLsbVolt:Integer; // 定义一个足够宽度的整型变量存放单位电压值(uV)
Volt:Integer;      // 定义一个足够宽度的整型变量存放转换后的实际电压值(uV)
PerLsbVolt := (10000.0/4096) * 1000; // 取得单位电压值，并以 uV 为单位将其转换成整型值
Volt := (ADBuffer [0] And $0FFF) * PerLsbVolt-5000*1000; // 用整型运算方式求得相应电压值(uV)
```

第二节、关于采集函数的 ADBuffer 缓冲区中的数据排放规则

当首末通道相等时，即为单通道采集，假如 FirstChannel=5, LastChannel=5,其排放规则如下

数据缓冲区索引号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
通道号	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	...

两通道采集(CH0 - CH2)

数据缓冲区索引号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
通道号	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	...

四通道采集(CH0 - CH3)

数据缓冲区索引号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
通道号	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	...

其他通道方式以此类推。

如果用户是进行连续不间断循环采集，即用户只进行一次初始化设备操作，然后不停的从设备上读取 AD 数据，那以需要用户特别注意的是应处理好各通道数据排列和对齐问题，尤其任意通道数采集时。否则，用户无法将规则排放在缓冲区中的各通道数据正确分离出来。怎样正确处理呢？我们建议的方法是，每次从设备上读取的点数应置为所选通道数量的整数倍长，这样便能保证每读取的这批数据在缓冲区中的相应位置始终固定对应于某一个通道的数据。比如用户要求对 1、2 两个 AD 通道的数据进行连续循环采集，则置每次读取长度为其 2 的整倍长 2n(n 为每个通道的点数)，这里设为 2048。试想，如此一来，每次读取的 2048 个点中的第一个点始终对应于 1 通道数据，第二个点始终对应于 2 通道，第三个点再应于 1 通道，第四个点再对应于 2 通道……以此类推。直到第 2047 个点对应于 1 通道数据，第 2048 个点对应 2 通道。这样一来，每次读取的段长正好包含了从首通道到末通道的完整轮回，如此一来，用户只须按通道排列规则，按正常的处理方法循环处理每一批数据。而对于其他情况也是如此，比如 3 个通道采集，则可以使用 3n(n 为每个通道的点数)的长度采集。为了更加详细地说明问题，请参考下表（演示的是采集 1、2、3 共三个通道的情况）。由于使用连续采样方式，所以表中的数据序列一行的数字变化说明了数据采样的连续性，即随着时间的延续，数据的点数连续递增，直至用户停止设备为止，从而形成了一个有相当长度的连续不间的多通道数据链。而通道序列一行则说明了随着连续采样的延续，其各通道数据在其整个数据链中的排放次序，这是一种非常规则而又绝对严格的顺序。但是这个相当长度的多通道数据链则不可能一次通过设备对象函数如 ReadDeviceProAD_X 函数读回，即便不考虑是否能一次读完的问题，但对用户的实时数据处理要求来说，一次性读取那么长的数据，则往往是相当矛盾的。因此我们就得分若干次分段读取。但怎样保证既方便处理，又不易出错，而且还高效。还是正如前面所说，采用通道数的整数倍长读取每一段数据。如表中列举的方法 1（为了说明问题，我们每读取一段数据只读取 2n 即 3*2=6 个数据）。从方法 1 不难看出，每一段缓冲区中的数据在相同缓冲区索引位置都对应于同一个通道。而在方

法 2 中由于每次读取的不是通道整数倍长, 则出现问题, 从表中可以看出, 第一段缓冲区中的 0 索引位置上的数据对应的是第 1 通道, 而第二段缓冲区中的 0 索引位置上的数据则对应于第 2 通道的数据, 而第三段缓冲区中的数据则对应于第 3 通道……, 这显然不利于循环有效处理数据。

在实际应用中, 我们在遵循以上原则时, 应尽可能地使每一段缓冲足够大, 这样, 可以一定程度上减少数据采集程序和数据处理程序的 CPU 开销量。

数据序列	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	...
通道序列	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	...
方法 1	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	...
缓冲区号	第一段缓冲					第二段缓冲区					第三段缓冲区					第 n 段缓冲						
方法 2	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	...
	第一段缓冲区			第二段缓冲区			第三段缓冲区			第四段缓冲区			第五段缓冲区			第 n 段缓						

第三节、关于测试应用程序创建并形成的数据文件格式

首先该数据文件从始端 0 字节位置开始往后至第 255 字节位置共 256 字节宽度属于文件头信息, 而从 256 开始才是真正的 AD 数据。文件头信息包含的内容如下结构体所示。

```
typedef struct _FILE_HEADER
{
    LONG BusType;           // 1:PCI, 2:USB, 3:ISA, 4:PC104
    LONG DeviceID;         // 0x2000
    LONG HeaderSizeBytes; // 文件头信息长度

    LONG VoltBottomRange;  // 量程下限(mV)
    LONG VoltTopRange;     // 量程上限(mV)

    LONG FirstChannel;     // 首通道
    LONG LastChannel;      // 末通道
    LONG Frequency;        // 采集频率(Hz)
    LONG Gains;            // 采集增益
    LONG TriggerSource;    // 触发方式
    LONG HeadEndFlag;      // 文件结束位(AA55AA55H)
} FILE_HEADER, *PFILE_HEADER;
```

AD 数据的格式为 16 位二进制格式, 它的排放规则与在 ADBuffer 缓冲区排放的规则一样, 即每 16 位二进制(字)数据对应一个 16 位 AD 数据。您只需要先开辟一个 16 位整型数组或缓冲区, 然后将磁盘数据从指定位置(即双字节对齐的某个位置)读入数组或缓冲区, 然后访问数组中的每个元素, 即是对相应 AD 数据的访问。

第八章 上层用户函数接口应用实例

第一节、怎样使用 ReadDeviceProAD_NotEmpty 函数直接取得 AD 数据

以下程序用非空读数据方式演示了 AD 采集的全部过程。

AD 采样的要求是: 共采集 32 通道[0, 31], 每个通道分别采集 1 个点, 共 32 个点, 采集频率为 50KHz,

Visual C++ & C++Builder:

```
int DeviceID = 0; // 假设在本台计算机系统中只有一台 PCI2310
HANDLE hDevice; // 设备对象句柄
PCI2310_PARA_AD Para; // 定义硬件参数
WORD ADBuffer[512]; // 分配数据缓冲区
PerLsbVolt = 10000.0/4096
hDevice = PCI2310_CreateDevice(DeviceID); // 创建设备对象
if(hDevice == INVALID_HANDLE_VALUE)
{
    AfxMessageBox("创建设备对象失败...", MB_ICONSTOP, 0);
}
```

```

    return 0;
}
// 对硬件 AD 参数进行预置
Para.FirstChannel = 0; // 置首通道为 0
Para.LastChannel = 31; // 置末通道为 31
Para.Frequency = 50000; // 置 AD 采样频率为 50KHz
Para.Gains = 1; // 置硬件放大增益为 1 倍
Para.TriggerSource = PCI2310_IN_TRIGGER;// 置 AD 启动模式为内部定时触发
// 用以上硬件参数初始化设备的 AD 对象
if(!PCI2310_InitDeviceProAD(hDevice, &Para))
{
    AfxMessageBox("不明确的初始化错误...",MB_ICONSTOP,0);
    return 0;
} // 注意: 此函数一旦返回 TRUE,设备即开始传输,客户程序必须能以
// 最快的速度读取数据状态,

// 从设备上读取 32 个字的数据
// 如果用户需要连续采集,即反复执行下面这个函数即可。
if(!PCI2310_ReadDeviceProAD_NotEmpty(hDevice, ADBuffer, 32))
{
    AfxMessageBox("读数据出错...",MB_ICONSTOP);
    return FALSE;
}
// 当上面的函数成功返回后,即所指定长度的数据已放在了 ADBuffer 中
// 用户即可以对这批数据进行处理。
Else
{
    int Channel; CString string; char str[500]; WORD LSB; float Volt;
    for(Channel=0; Channel<32; Channel++)
    {
        // 取得实际有效的 LSB
        // 将 12 位数据的最高求反,形成补码,即有效 LSB
        // 将 LSB 换算成电压值(单位 mV)
        Volt = (ADBuffer[Channel] & 0xFFF)*PerLsbVolt -5000.0;

        sprintf(str, "Channel : %d Volt=%f\n", Volt);
        string=string+str;
    }
    // 通过对话框显示各路电压值
    AfxMessageBox(string);
}
if(!PCI2310_ReleaseDeviceProAD(hDevice))
{
    AfxMessageBox("释放 AD 部件失败");
}
PCI2310_ReleaseDevice( hDevice );

```

Visual Basic:

```

Dim hDevice As Long ' 设备对象句柄
Dim Para As PCI2310_PARA_AD ' 定义硬件参数
Dim i As Long
Const PerLsbVolt =10000.0/4096
Dim bStatus As Boolean
Dim ReadIndex As Integer ' 级链缓冲区的索引号
Dim DigitString As String
Dim CurrentIndex As Integer ' 数据处理时使用的当前缓冲索引
Dim ADBuffer[512] As Long ' 分配数据缓冲区

```

```

Dim Device As Long
DeviceID = 0 ' 假设在本台计算机系统中只有一台 PCI2310
hDevice =PCI2310_CreateDevice(DeviceID) ' 创建设备对象
If hDevice = INVALID_HANDLE_VALUE Then
    MsgBox "创建设备对象失败..."
End If
Para.FirstChannel = 0 ' 置首通道为 0
Para.LastChannel =31 ' 置末通道为 31
Para.Frequency = 50000 '置采集频率为 50KHz
Para.Gains = 1 ' 置硬件增益为 1 倍
Para.TriggerSource = PCI2310_IN_TRIGGER ' 置触发方式为板上内部定时触发
'初始化设备对象
If Not PCI2310_InitDeviceProAD(hDevice, Para) Then
    MsgBox "不明确的初始化错误..."
Exit Function
End If
Do While (bDeviceRun) '循环采集 AD 数据
    If Not PCI2310_ReadDeviceProAD_NotEmpty(hDevice, ADBuffer(0), 32) Then
        MsgBox "读数据出错..."
        Exit Function
    Else
        For Channel = 0 To 31
            DigitString =( (Str$( ADBuffer(Channel)And &HFFF) * PerLsbVolt - 5000))
        End If
    Next Channel
    MsgBox "DigitString"
Loop
If Not PCI2310_ReleaseDeviceProAD(hDevice) Then '释放设备上的 AD 部件
    MsgBox "释放 AD 部件失败"
End If
If Not PCI2310_ReleaseDevice(hDevice) Then '关闭设备
    MsgBox "关闭设备失败..."
End If

```

LabView:

关于 LabView 的 AD 数据采集请参考附录 A 的第二章《[LabView 驱动程序接口](#)》

第二节、怎样使用 ReadDeviceProAD_Half 函数直接取得 AD 数据

以下程序用半满读数据方式演示了 AD 采集的全部过程。

设 FIFO 的长度为 4096 个点，即半满长度为 2048 个点，AD 采样的要求是：共采集两个通道 0、31，每个通道分别采集 1024 个点，共 2048 个点，正好是 FIFO 的半满长度（当然每次读数据时可以小于半满长度，但是绝对不应大于半满长度）采集频率为 50KHz。

Visual C++ & C++Builder:

```

int DeviceID = 0; // 假设在本台计算机系统中只有一台 PCI2310
HANDLE hDevice; // 设备对象句柄
PCI2310_PARA_AD Para; // 定义硬件参数
WORD ADBuffer[2048]; // 分配数据缓冲区
hDevice =PCI2310_CreateDevice(DeviceID); // 创建设备对象
PerLsbVolt =10000.0/4096
if(hDevice==INVALID_HANDLE_VALUE)
{
    AfxMessageBox("创建设备对象失败...",MB_ICONSTOP,0);
    return 0;
}
// 对硬件 AD 参数进行预置

```

```

Para.FirstChannel = 0; // 置首通道为 0
Para.LastChannel = 1; // 置末通道为 1
Para.Frequency = 50000; // 置 AD 采样频率为 50KHz
Para.Gains = 1; // 置硬件放大增益为 1 倍
Para.TriggerSource = PCI2310_IN_TRIGGER;// 置 AD 启动模式为内部定时触发
// 用以上硬件参数初始化设备的 AD 对象
if(!PCI2310_InitDeviceProAD(hDevice, &Para))
{
    AfxMessageBox("不明确的初始化错误...",MB_ICONSTOP,0);
    return 0;
} // 注意: 此函数一旦返回 TRUE,设备即开始传输,客户程序必须能以
// 最快的速度读取 AD 的半满状态,
if(PCI2310_GetDevStatusAD_Half(hDevice)) // 等待 FIFO 的半满
{
    break;
}
// 当半满信号有效时,即可从设备上读走半满或半满长度以下的数据
// 从设备上读取 2048 个字的数据
// 如果用户需要连续采集,即反复执行下面这个函数即可。
if(!PCI2310_ReadDeviceProAD_Half(hDevice, ADBuffer, 2048))
{
    AfxMessageBox("读数据出错...",MB_ICONSTOP);
    return FALSE;
}
// 当上面的函数返回后,即所指定长度的数据已放在了 ADBuffer 中
// 用户即可以对这批数据进行处理。
Else
{
    int Channel; Cstring string; char str[500]; WORD LSB; float Volt;
    // 此处只举例处理 32 个通道中的第一轮数据
    for(Channel=0; Channel<32; Channel++)
    {
        // 取得实际有效的 LSB
        // 将 12 位数据的最高求反,形成补码,即有效 LSB
        // 将 LSB 换算成电压值(单位 mV)
        Volt = (ADBuffer[Channel] &0xFFF)*PerLsbVolt -5000.0;

        sprintf(str, "Channel : %d Volt=%f\n", Volt);
        string=string+str;
    }
    // 通过对话框显示各路电压值
    AfxMessageBox(string);
}

if(!PCI2310_ReleaseDeviceProAD(hDevice))
{
    AfxMessageBox("释放 AD 部件失败");
}
PCI2310_ReleaseDevice( hDevice );

```

Visual Basic:

```

Dim hDevice As Long ' 设备对象句柄
Dim Para As PCI2310_PARA_AD ' 定义硬件参数
Dim i As Long
Dim bStatus As Boolean
Dim ReadIndex As Integer ' 级链缓冲区的索引号
Dim DigitString As String

```

```

Dim CurrentIndex As Integer      '数据处理时使用的当前缓冲索引
Dim ADBuffer[2048] As Long
'分配数据缓冲区
Dim Device As Long
Const PerLsbVolt = 10000/4096
DeviceID = 0      '假设在本台计算机系统中只有一台 PCI2310
hDevice = PCI2310_CreateDevice(DeviceID)      '创建设备对象
If hDevice = INVALID_HANDLE_VALUE Then
    MsgBox "创建设备对象失败..."
End If
Para.FirstChannel = 0      '置首通道为 0
Para.LastChannel = 31      '置末通道为 31
Para.Frequency = 50000      '置采集频率为 50KHz
Para.Gains = 1      '置硬件增益为 1 倍
Para.TriggerSource = PCI2310_IN_TRIGGER      '置触发方式为板上内部定时触发
If Not PCI2310_InitDeviceProAD(hDevice, Para) Then
    MsgBox "不明确的初始化错误..."
Exit Function
End If
Do While (bDeviceRun)      '循环采集 AD 数据
    If Not PCI2310_ReadDeviceProAD_Half(hDevice, ADBuffer(0), 2048) Then
        MsgBox "读数据出错..."
    Else
        For Channel = 0 To 31

            DigitString = ((Str$( ADBuffer(Channel) And &HFFF) *PerLsbVolt - 5000))
        End If
        Next Channel
        MsgBox "DigitString"
    Loop
    If Not PCI2310_ReleaseDeviceProAD(hDevice) Then      '释放设备上的 AD 部件
        MsgBox "释放 AD 部件失败"
    End If
    If Not PCI2310_ReleaseDevice(hDevice) Then      '关闭设备
        MsgBox "关闭设备失败..."
    End If
LabView:
关于 LabView 的 AD 数据采集请参考附录 A 的详细叙述

```

第三节、怎样使用 ReadDeviceIntAD 函数直接取得 AD 数据

以下程序用中断方式演示了 AD 采集的全部过程。

设 FIFO 的长度为 4096 个点，即半满长度为 2048 个点，AD 采样的要求是：共采集两个通道 0、1，每个通道分别采集 1024 个点，共 2048 个点，正好是 FIFO 的半满长度（当然每次读数据时可以小于半满长度，但是绝对不应大于半满长度）采集频率为 50KHz。

注意：如果用户要实现连续采集，则应将下列标注为红色的代码段置入一循环体内，反复执行即可。

Visual C++ & C++Builder:

```

Int DeviceID = 0;      // 假设在本台计算机系统中只有一台 PCI2310
HANDLE hDevice;      // 设备对象句柄
HANDLE hEventINT;      // 内核中断事件对象
PCI2310_PARA_AD Para; // 定义硬件参数
WORD ADBuffer[2048]; // 分配数据缓冲区
DWORD Remaining=0;      // 用于记录用户每次真正接受到中断事件后一
                        // 级缓冲链表的乘余单元
PerLsbVolt =(10000.0/4096)
hEventINT = PCI2310_CreateSystemEvent(); // 创建系统中断事件对象
if (hEvent == NULL)

```

```
{
    AfxMessageBox("创建系统事件对象失败");
    return 0;
}
hDevice =PCI2310_CreateDevice(DeviceID); // 创建设备对象
if(hDevice==INVALID_HANDLE_VALUE)
{
    AfxMessageBox("创建设备对象失败...",MB_ICONSTOP,0);
    return 0;
}
// 对硬件 AD 参数进行预置
Para.FirstChannel = 0; // 置首通道为 0
Para.LastChannel = 1; // 置末通道为 1
Para.Frequency = 50000; // 置 AD 采样频率为 50KHz
Para.Gains = 1; // 置硬件放大增益为 1 倍
Para.TriggerSource = PCI2310_IN_TRIGGER;// 置 AD 启动模式为内部定时触发
// 下面用以上硬件参数初始化设备的 AD 对象，且指定半满长度。
if(!PCI2310_InitDeviceIntAD(hDevice, hEventINT, 2048, &Para))
{
    AfxMessageBox("不明确的初始化错误...",MB_ICONSTOP,0);
    return 0;
} // 注意: 此函数一旦返回 TRUE,设备即开始传输,客户程序必须能以
// 最快的速度迎接 FIFO 半满中断事件

// 等待 FIFO 半满中断事件发生，一旦发生即刻读数
WaitForSingleObject (hEventINT, INFINITE);
// 当半满信号到达时，即可从设备上读走半满或半满长度以下的数据
// 从设备上读取 2048 个字的数据
// 如果用户需要连续采集，即反复执行下面这个函数即可。
Do{
    Remaining=PCI2310_ReadDeviceIntAD(hDevice, \
                                     ADBuffer[ReadIndex], \
                                     2048);

    // 判断一级缓冲区是否溢出或有其他错误
    if(Remaining!=0xe1000000)
    {
        // 如果返回值不等于 0xe1000000，则确定一级缓冲区链表中的数据有效
        // 当上面的函数返回后，即所指定长度的数据已放在了 ADBuffer 中
        // 用户即可以对这批数据进行处理。
        // 需要注意的是：ReadDeviceIntAD 中的 ReadSizeWords 参数必须和
        // InitDeviceIntAD 函数中的 nFifoHalfLength 相等，才会保证数据不会丢掉。
        int Channel; CString string; char str[500]; WORD LSB; float Volt;
        // 此处只举例处理 32 个通道中的第一轮数据
        for(Channel=0; Channel<32; Channel++)
        {
            // 取得实际有效的 LSB
            // 将 LSB 换算成电压值(单位 mV)
            Volt = (ADBuffer[Channel] &0xFFF)*PerLsbVolt -5000.0;
            sprintf(str, "Channel : %d Volt=%f\n", Volt);
            String=string+str;
        }
        // 通过对话框显示各路电压值
        AfxMessageBox(string);
    }
} while(Remaining); // 至到一级缓冲片段数为 0
if(!PCI2310_ReleaseDeviceProAD(hDevice))
```

```
{
AfxMessageBox("释放 AD 部件失败");
}
PCI2310_ReleaseDevice( hDevice );
```

Visual Basic:

```
Dim hEventINT As Long      '内核中断事件对象
Dim Para As PCI2310_PARA_AD '定义硬件参数
Dim ADBuffer(2048) As Long 分配数据缓冲区
Dim hEventAs Long
Dim bStatus As Boolean
Dim ReadIndex As Integer   '级链缓冲区的索引号
Dim hDevice As Long
Dim i As Long
Const PerLsbVolt = 10000.0/4096
DeviceID = 0 '设当前被操作的 PCI 设备只有一个
hEvent = INVALID_HANDLE_VALUE
hEvent = PCI2310_CreateSystemEvent() '创建系统内核事件对象，用于线程同步
If hEvent = INVALID_HANDLE_VALUE Then
    MsgBox "创建设备对象失败..."
    Exit Function
End If
hDevice = PCI2310_CreateDevice(DeviceID) '创建设备对象
If hDevice = INVALID_HANDLE_VALUE Then
    MsgBox "创建设备对象失败..."
    Exit Function
End If
Para.FirstChannel = 0 '置首通道为 0
Para.LastChannel = 1 '置末通道为 1
Para.Frequency = 50000 '置采集频率为 50KHz
Para.Gains = 1 '置硬件增益为 1 倍
Para.TriggerSource = PCI2310_IN_TRIGGER '置触发方式为板上内部定时触发
'初始化设备对象
If Not PCI2310_InitDeviceProAD(hDevice, Para) Then
    MsgBox "不明确的初始化错误..."
    Exit Function
End If
'等待 FIFO 半满中断事件发生，一旦发生即刻读数
WaitForSingleObject (hEventINT, INFINITE)
'当半满信号到达时，即可从设备上读走半满或半满长度以下的数据
'从设备上读取 2048 个字的数据
'如果用户需要连续采集，即反复执行下面这个函数即可。
Do While
    Remaining=PCI2310_ReadDeviceIntAD(hDevice
                                     ADBuffer(ReadIndex)
                                     2048)

    '判断一级缓冲区是否溢出或有其他错误
    if(Remaining<>0xe1000000) then

        '如果返回值不等于 0xe1000000，则确定一级缓冲区链表中的数据有效
        '当上面的函数返回后，即所指定长度的数据已放在了 ADBuffer 中
        '用户即可以对这批数据进行处理。
        '此处只举例处理 32 个通道中的第一轮数据
For Channel = 0 To 31
    DigitString = ((Str$( ADBuffer(Channel) And &HFFF) * PerLsbVolt - 5000))
End If
```


Next Channel MsgBox “DigitString” Loop

```
If Not PCI2310_ReleaseDeviceProAD(hDevice) Then '释放设备上的 AD 部件
    MsgBox "释放 AD 部件失败"
End If
If Not PCI2310_ReleaseDevice(hDevice) Then '关闭设备
    MsgBox "关闭设备失败..."
End If
```

LabView:

关于 LabView 的 AD 数据采集请参考附录 A 的第二章《[LabView 驱动程序接口](#)》

第四节、怎样使用 SetDeviceDO 函数进行更便捷的数字开关量输出操作

以下程序演示了如何使通道 0、2、4、6、8、10、12、14 为关状态“0”，其他通道为开状态“1”。

Visual C++ & C++Builder:

```
HANDLE hDevice;
PCI2310_PARA_DO Para; // 定义开关量参数结构
hDevice = PCI2310_CreateDevice(0); // 创建设备对象
Para.DO0 = 0; Para.DO8 = 0;
Para.DO1 = 1; Para.DO9 = 1;
Para.DO2 = 0; Para.DO10 = 0;
Para.DO3 = 1; Para.DO11 = 1;
Para.DO4 = 0; Para.DO12 = 0;
Para.DO5 = 1; Para.DO13 = 1;
Para.DO6 = 0; Para.DO14 = 0;
Para.DO7 = 1; Para.DO15 = 1;
// 输出各路开关量状态
PCI2310_SetDeviceDO( hDevice, &Para);
PCI2310_ReleaseDevice( hDevice );
```

Visual Basic:

```
Dim hDevice As Long
Dim Para As PCI2310_PARA_DO ' 定义开关量参数结构
Dim bStatus As Boolean
hDevice = PCI2310_CreateDevice(0) ' 创建设备对象
Para.DO0 = 0 Para.DO8 = 0
Para.DO1 = 1 Para.DO9 = 1
Para.DO2 = 0 Para.DO10 = 0
Para.DO3 = 1 Para.DO11 = 1
Para.DO4 = 0 Para.DO12 = 0
Para.DO5 = 1 Para.DO13 = 1
Para.DO6 = 0 Para.DO14 = 0
Para.DO7 = 1 Para.DO15 = 1
'输出各路开关量状态
bStatus =PCI2310_SetDeviceDO( hDevice, Para)
bStatus =PCI2310_ReleaseDevice( hDevice )
```

LabView:

关于 LabView 的开关量输出操作请参考附录 A 的详细叙述

第五节、怎样使用 GetDeviceDI 函数进行更便捷的数字开关量输入操作

以下程序演示了如何使通道 0、2、4、6、8、10、12、14 为关状态“0”，其他通道为开状态“1”。

Visual C++ & C++Builder:

```
HANDLE hDevice;
PCI2310_PARA_DI Para; // 定义开关量参数结构
HDevice = PCI2310_CreateDevice(0); // 创建设备对象
```

```

// 输入各路开关量状态
Para.DI0 = 0;      Para.DI8 = 0;
Para.DI1 = 1;      Para.DI9 = 1;
Para.DI2 = 0;      Para.DI10 = 0;
Para.DI3 = 1;      Para.DI11 = 1;
Para.DI4 = 0;      Para.DI12 = 0;
Para.DI5 = 1;      Para.DI13 = 1;
Para.DI6 = 0;      Para.DI14 = 0;
Para.DI7 = 1;      Para.DI15 = 1;
PCI2310_GetDeviceDI( hDevice, &Para);
// 当输入各路开关量状态后, 即可进行根据用户需要相应处理
PCI2310_ReleaseDevice( hDevice );
Visual Basic:
Dim hDevice As Long
Dim Para As PCI2310_PARA_DO ‘ 定义开关量参数结构
Dim bStatus As Boolean
hDevice = PCI2310_CreateDevice(0) ‘ 创建设备对象
Para.DI0 = 0      Para.DI8 = 0
Para.DI1 = 1      Para.DI9 = 1
Para.DI2 = 0      Para.DI10 = 0
Para.DI3 = 1      Para.DI11 = 1
Para.DI4 = 0      Para.DI12 = 0
Para.DI5 = 1      Para.DI13 = 1
Para.DI6 = 0      Para.DI14 = 0
Para.DI7 = 1      Para.DI15 = 1
‘输出各路开关量状态
bStatus =PCI2310_GetDeviceDI( hDevice, Para)
bStatus =PCI2310_ReleaseDevice( hDevice )

```

LabView:

关于 LabView 的开关量输入操作请参考附录 A 的第二章《[LabView 驱动程序接口](#)》

第九章 底层用户函数接口应用实例

请注意, 关于数据采集技术和数据处理技术的相关论坛请参考用户安装根目下的 CommonHelp 中的有关 Word

第一节、怎样使用映射寄存器读写函数直接编写数据采集程序?

对一般方式的数据采集, 如程序采集方式, 通常就是对设备各寄存器进行多次读写操作的过程。所以这个过程就显得很简单, 参照硬件说明书的各种寄存器定义情况反复使用 WriteRegisterX 和 ReadRegisterX 函数即可轻松实现数据采集。以下各种各板的 Visual C++ 演示程序(用户可以使用下面示范中的反复执行红色部分代码, 即可实现连续数据采集):

```

ULONG Index=0;   HANDLE hDevice;
ULONG LinearAddr, PhysAddr;
ULONG Control;
WORD TimerCount; // 8253 定时器的 16 位计数值
WORD TimerCountL; // 8253 定时器的低 8 位计数值
WORD TimerCountH; // 8253 定时器的高 8 位计数值
PPCI2310_PARA_AD Para;
Para.FirstChannel=0; // 预置首通道
Para.LastChannel=31; // 预置末通道
Para.Gains=1;        // 预置增益
Para.Frequency=20000; // 预置频率 Hz
Para.TriggerSource = PCI2310_IN_TRIGGER; // 预置触发方式
Control=(pPara->Gains-1)*32 + Para.FirstChannel; // 构造控制字

```

```

// 将频率换算成定时计数器的 16 位计数值
TimerCount=(WORD)((1000000*2)/Para.Frequency);
// 从 16 位定时计数值中提取低 8 位计数值
TimerCountL=(BYTE)(Frequency&0x00ff);
// 从 16 位定时计数值中提取高 8 位计数值
TimerCountH=(BYTE)((Frequency&0xff00)>>8);

hDevice = PCI2310_CreateDevice(0);
// 取得 0 号映射寄存器的线性基地址
PCI2310_GetDeviceAddr( hDevice, &LinearAddr, &PhysAddr,0);

PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x4, 0x162);
// 开始设备实际操作
// 设备控制字, D0=1:允许中断, D1=1:允许 AD 转换, D2=0:允许
// 开关量输出, 此处是不允许开关量输出
WriteRegisterULong( hDevice, LinearAddr, 0x280, 0x08); ///////////////////////////////////////////////////////////////////
//置首通道,末通道,增益(D7-D5)+首通道(D4-D0)
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x200, Control);
PCI2310_WriteRegisterULong( hDevice, LinearAddr, \
0x210, pPara->LastChannel); // 末通道
/////////////////////////////////////////////////////////////////
// 写入 8253 工作控制字, 方式 3, 计数器 0, (该端口低 8 位有效)
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x220, 0x34);
// 程序产生写控制字寄存器 0x230 的时序信号(写 8253 控制字)
// D3: WR53; D2: CS53; D1D0: A1A0
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x230, 0x0b);
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x230, 0x03);
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x230, 0x0f);

// 写计数器 0 的计数值(低 8 位)
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x220, TimerCountL);

// 程序产生写计数器 0 的计数值的时序信号
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x230, 0x08);
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x230, 0x00);
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x230, 0x0c);
// 写计数器 0 的计数值(高 8 位)
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x220, TimerCountH);
// 程序产生写计数器 0 的计数值的时序信号
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x230, 0x08);
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x230, 0x00);
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x230, 0x0c);
//清 FIFO 存储器
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x290, 0x07);
// 允许 8253 工作
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x280, \
pPara->TriggerSource); //触发方式

register ULONG i; register ULONG Value;
WORD ADBuffer[65536];
// 以下两段代码, 在某一时间只能选择其中一段
//+++++
// 以下读取数据代码属半满方式

while(TRUE)
{
Value = PCI2310_ReadRegisterULong(LinearAddr+0x260)&0x00000004;
If(Value!=4) break; // D2=1: 正在转换 D2=0: 完成
}

```

```

Else
// Win9x 环境下的延时操作
_asm{
    nop
}
// NT/WIN2000 环境下的纳秒级的高效延时操作
// PCI2310_DelayTimeNs(hDevice, 10); // 延时 1 微秒
}
for(i=0; i<2048; i++)
{
// 置总线时序
PCI2310_WriteRegisterULONG(hDevice, LinearAddr+0x2a0, 0x00);
ADBuffer[i]=(WORD) PCI2310_ReadRegisterULONG(hDevice,\
                                                LinearAddr+0x270);
// 置总线时序
PCI2310_WriteRegisterULONG(hDevice, LinearAddr+0x2a0, 0x01);
}
// 此处便可以对 ADBuffer 缓冲区中的数据进行相应处理

//+++++
// 以下读取数据代码属非空方式

for(i=0; i<4096; i++)
{
    while(TRUE)
    {
        Value = PCI2310_ReadRegisterULONG(LinearAddr+ \
                                           0x260)&0x00000001;
        if(Value==1) break; // D1=0: 正在转换 D2=1: 完成
        // NT/WIN2000 环境下的纳秒级的高效延时操作
        else
            PCI2310_DelayTimeNs(hDevice, 1); // 延时 100 纳秒
    }

// 当非空标志有效时，开始读数据，
// 置总线时序
PCI2310_WriteRegisterULONG(hDevice, LinearAddr+0x2a0, 0x00);
ADBuffer[i]=(WORD) PCI2310_ReadRegisterULONG(hDevice,\
                                                LinearAddr+0x270);
// 置总线时序
PCI2310_WriteRegisterULONG(hDevice, LinearAddr+0x2a0, 0x01);
}
// 此处便可以对 ADBuffer 缓冲区中的数据进行相应处理
//+++++

// 程序结束时，一定要释放设备对象
PCI2310_ReleaseDevice( hDevice )

```

第二节、怎样使用映射寄存器读写函数直接编写开关量输入输出程序？

对于开关量操作主要通过 WriteRegisterX 或 ReadRegisterX 对 PCI 设备映射寄存器的开关量单元进行读写实现的，所以编程也很容易。现就 PCI2310 开关量部分的操作程序（For Visual C++）举例如下：

// 以下是开关量输出操作

```
WORD m_Switch; // 用于临时存放 DO, DI 端口状态
BYTE m_DI[16]; // 存放 0-7, 8-15 等 16 路 DI 状态
BYTE m_DO[16]; // 用于 0-7, 8-15 等 16 路 DO 的状态
HANDLE hDevice;
ULONG LinearAddr, PhysAddr;
hDevice = PCI2310_CreateDevice(0); // 创建设备对象
// 取得 0 号内存映射寄存器的线性基地址和物理基地址
PCI2310_GetDeviceAddr(hDevice, &LinearAddr, &PhysAddr, 0);
// 置各路开关量输出状态
m_DO[0] = 1; // 置 DO0 为开“1”状态
m_DO[1] = 0; // 置 DO1 为开“0”状态
m_DO[2] = 1; // 置 DO2 为开“1”状态
m_DO[3] = 0; // 置 DO3 为开“0”状态
m_DO[4] = 1; // 置 DO4 为开“1”状态
m_DO[5] = 0; // 置 DO5 为开“0”状态
m_DO[6] = 1; // 置 DO6 为开“1”状态
m_DO[7] = 0; // 置 DO7 为开“0”状态
m_DO[8] = 1; // 置 DO8 为开“1”状态
m_DO[9] = 0; // 置 DO9 为开“0”状态
m_DO[10] = 1; // 置 DO10 为开“1”状态
m_DO[11] = 0; // 置 DO11 为开“0”状态
m_DO[12] = 1; // 置 DO12 为开“1”状态
m_DO[13] = 0; // 置 DO13 为开“0”状态
m_DO[14] = 1; // 置 DO14 为开“1”状态
m_DO[15] = 0; // 置 DO15 为开“0”状态
m_Switch = m_DO[0] | m_DO[1]<<1 | m_DO[2]<<2 | m_DO[3]<<3 | m_DO[4]<<4
           | m_DO[5]<<5 | m_DO[6]<<6 | m_DO[7]<<7; // 组合 0-7 路的端口开关量控制字
m_Switch = m_Switch | m_DO[8]<<8 | m_DO[9]<<9 | m_DO[10]<<10 | m_DO[11]<<11 | m_DO[12]<<12 |
m_DO[13]<<13 | m_DO[14]<<14 | m_DO[15]<<15; // 组合 0-15 路的端口开关量控制字
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x240, m_Switch); // DO0-DO15 开关量输出控制。
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x2F0, 0x00);
// 以下是开关量输入操作
PCI2310_WriteRegisterULong( hDevice, LinearAddr, 0x280, 0x02);
m_Switch = PCI2310_ReadRegisterULong( hDevice, LinearAddr, 0x250)&0x0000ffff; // 读入 15 路状态
// 分离出 DI0-DI5 等 16 路 DI 状态
m_DI[0] = m_Switch&0x0001; // 取得 DI0 的状态
m_DI[1] =(m_Switch&0x0002)>>1; // 取得 DI1 的状态
m_DI[2] =(m_Switch&0x0004)>>2; // 取得 DI2 的状态
m_DI[3] =(m_Switch&0x0008)>>3; // 取得 DI3 的状态
m_DI[4] =(m_Switch&0x0010)>>4; // 取得 DI4 的状态
m_DI[5] =(m_Switch&0x0020)>>5; // 取得 DI5 的状态
m_DI[6] =(m_Switch&0x0040)>>6; // 取得 DI6 的状态
m_DI[7] =(m_Switch&0x0080)>>7; // 取得 DI7 的状态
// 分离出 DI8-DI15 等 8 路 DI 状态
m_DI[8] =(m_Switch&0x0100)>>8; // 取得 DI8 的状态
m_DI[9] =(m_Switch&0x0200)>>9; // 取得 DI9 的状态
m_DI[10] =(m_Switch&0x0400)>>10; // 取得 DI10 的状态
m_DI[11] =(m_Switch&0x0800)>>11; // 取得 DI11 的状态
m_DI[12] =(m_Switch&0x1000)>>12; // 取得 DI12 的状态
m_DI[13] =(m_Switch&0x2000)>>13; // 取得 DI13 的状态
m_DI[14] =(m_Switch&0x4000)>>14; // 取得 DI14 的状态
m_DI[15] =(m_Switch&0x8000)>>15; // 取得 DI15 的状态
PCI2310_ReleaseDevice( hDevice );
}
```

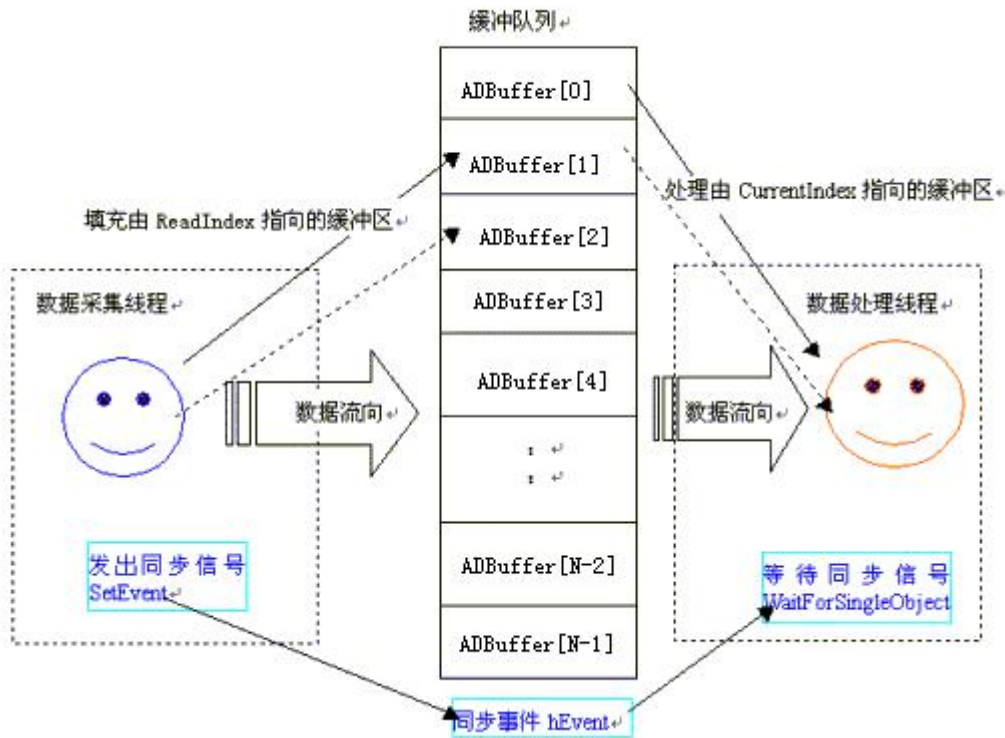
第十章 高速大容量、连续不间断数据采集及存盘技术详解

与 ISA、USB 设备同理，使用子线程跟踪 AD 转换进度，并进行数据采集是保持数据连续不间断的最佳方案。但是与 ISA 总线设备不同的是，PCI 设备在这里不使用动态指针去同步 AD 转换进度，因为 ISA 设备环形内存池的动态指针操作是一种软件化的同步，而 PCI 设备不再有软件化的同步，而完全由硬件和驱动程序自动完成。这样一来，用户要用程序方式实现连续数据采集，其软件实现就显得极为容易。每次用 ReadDeviceProAD_X(或 ReadDeviceIntAD) 函数读取 AD 数据时，那么设备驱动程序会按照 AD 转换进度将 AD 数据一一放进用户数据缓冲区，当完成该次所指定的点数时，它便会返回，当您再次用这个函数读取数据时，它会接着上一次的位置传递数据到用户数据缓冲区。只是要求每两次 ReadDeviceAD_X 之间的时间间隔越短越好。

但是由于我们的设备是通常工作在一个单 CPU 多任务的环境中，由于任务之间的调度切换非常平凡，特别是当用户移动窗口、或弹出对话框等，则会使当前线程猛地花掉大量的时间去处理这些图形操作，因此如果处理不当，则将无法实现高速连续不间断采集，那么如何更好的克服这些问题呢？用子线程则是必须的（在这里我们称之为数据采集线程），但这还不够，必须要求这个线程是绝对的工作者线程，即这个线程在正常采集中不能有任何窗口等图形操作。只有这样，当用户进行任何窗口操作时，这个线程才不会被堵塞，因此可以保证正常连续的数据采集。但是用户可能要问，不能进行任何窗口操作，那么我如何将采集的数据显示在屏幕上呢？其实很简单，再开辟一个子线程，我们称之为数据处理线程，也叫用户界面线程。最初，数据处理线程不做任何工作，而是在 Win32 API 函数 WaitForSingleObject 的作用下进入睡眠状态，此时它不消耗 CPU 任何时间，即可保证其他线程代码有充分的运行机会（这里当然主要指数据采集线程），当数据采集线程取得指定长度的数据到用户空间时，则再用 Win32 API 函数 SetEvent 将指定事件消息发送给数据处理线程，则数据处理线程即刻恢复运行状态，迅速对这批数据进行处理，如计算、在窗口绘制波形、存盘等操作。

可能用户还要问，既然数据处理线程是非工作者线程，那么如果用户移动窗口等操作堵塞了该线程，而数据采集线程则在不停地采集数据，那数据处理线程难道不会因此而丢失数据采集线程发来的某一段数据吗？如果不另加处理，这个情况肯定有发生的可能。但是，我们采用了一级缓冲队列和二级缓冲队列的设计方案，足以避免这个问题。即假设数据采集线程每一次从设备上取出 8K 数据，那么我们就创建一个缓冲队列，在用户程序中最简单的办法就是开辟一个二维数组如 ADBuffer[Count][DataLen]，我们将 DataLen 视为数据采集线程每次采集的数据长度，Count 则为缓冲队列的成员个数。您应根据您的计算机物理内存大小和总体使用情况来设定这个数。假如我们设成 32，则这个缓冲队列实际上就是数组 ADBuffer[32][8192] 的形式。那么如何使用这个缓冲队列呢？方法很简单，它跟一个普通的缓冲区如一维数组差不多，唯一不同是，两个线程首先要通过改变 Count 字段的值，即这个下标 Index 的值来填充和引用由 Index 下标指向某一段 DataLen 长度的数据缓冲区。需要注意的两个线程不共用一个 Index 下标变量。具体情况是当数据采集线程在 AD 部件被 InitDeviceProAD 或 InitDeviceIntAD 初始化之后，首次采集数据时，则将自己的 ReadIndex 下标置为 0，即用第一个缓冲区采集 AD 数据。当采集完后，则向数据处理线程发送消息，且两个线程的公共变量 SegmentCounts 加 1，（注意 SegmentCounts 变量是用于记录当前时刻缓冲队列中有多少个已被数据采集线程使用了，但是却没被数据处理线程处理掉的缓冲区数量。）然后再接着将 ReadIndex 偏移至 1，再用第二个缓冲区采集数据。再将 SegmentCounts 加 1，至到 ReadIndex 等于 31 为止，然后再回到 0 位置，重新开始。而数据处理线程则在每次接受到消息时判断有多少由于自己被堵塞而没有被处理的缓冲区个数，然后逐一进行处理，最后再从 SegmentCounts 变量中减去在所接受到的当前事件下所处理的缓冲区个数，具体处理哪个缓冲区由 CurrentIndex 指向。因此，即便应用程序突然很忙，使数据处理线程没有时间处理已到来的数据，但是由于缓冲区队列的缓冲作用，可以让数据采集线程先将数据连续缓存在这个区域中，由于这个缓冲区可以设计得比较大，因此可以缓冲很大的时间，这样即便是数据处理线程由于系统的偶而繁忙而被堵塞，也很难使数据丢失。而且通过这种方案，用户还可以在数据采集线程中对 SegmentCounts 加以判断，观察其值是否大小了 32，如果大于，则缓冲区队列肯定因数据处理采集的过度繁忙而被溢出，如果溢出即可报警。因此具有强大的容错处理。

下图便形象的演示了缓冲队列处理的方法。可以看出，最初设备启动时，数据采集线程在往 ADBuffer[0] 里面填充数据时，数据处理线程便在 WaitForSingleObject 的作用下睡眠等待有效数据。当 ADBuffer[0] 被数据采集线程填充后，立即给数据处理线程 SetEvent 发送通知 hEvent，便紧接着开始填充 ADBuffer[1]，数据处理线程接到事件后，便醒来开始数据 ADBuffer[0] 缓冲。它们就这样始终差一个节拍。如虚线箭头所示。



第一节、使用程序查询方式实现该功能

下面用 Visual C++ 程序举例说明。

一、使用 ReadDeviceProAD_NotEmpty 函数读取设备上的 AD 数据（它使用 FIFO 的非空标志）

①下面只是基于 C 语言的简要的策略说明，其详细应用实例请参考 Visual C++ 测试与演示系统，您先点击 Windows 系统的[开始]菜单，再按下列顺序点击，即可打开基于 VC 的 Sys 工程。

[程序] [阿尔泰测控演示系统] [Microsoft Visual C++] [高级代码演示] [高级测试演示源程序]

然后，您着重参考 ADDoc.cpp 和 ADThread.cpp 源文件中以下函数：

```
void CADDoc:: OnStartDeviceAD () // 采集线程和处理线程的启动函数
BOOL StartDeviceAD_NotEmpty () // 启动采集线程函数
UINT ReadDataThread_NotEmpty() // 采集线程函数
BOOL StopDeviceAD_NotEmpty() // 采集线程的终止函数
UINT DrawWindowProc () // 绘制数据线程
void CADDoc:: OnStopDeviceAD () // 终止采集函数
```

在该工程级源代码的可执行程序中，选择菜单中的[设备管理] [非空方式采集]后，开始采集即可使用此方式。

二、使用 ReadDeviceProAD_Half 函数读取设备上的 AD 数据（它使用 FIFO 的半满标志）

该方案的实现与 ReadDeviceProAD_NotEmpty 基本相同，只有数据采集代码中部分参数不一样而已。这里也列出部分代码：(假设该设备上配置的是 1K FIFO，即只有 1024 个字长，那么其半满字数为 512，注意代码中的黑体部分)

下面只是简要的策略说明，其详细应用实例请参考 Visual C++ 测试与演示系统，您先点击 Windows 系统的[开始]菜单，再按下列顺序点击，即可打开基于 VC 的 Sys 工程。

[程序] [阿尔泰测控演示系统] [Microsoft Visual C++] [高级代码演示] [高级测试演示源程序]

然后，您着重参考 ADDoc.cpp 和 ADThread.cpp 源文件中以下函数：

```
void CADDoc:: OnStartDeviceAD () // 采集线程和处理线程的启动函数
BOOL StartDeviceAD_Half () // 启动采集线程函数
UINT ReadDataThread_Half() // 采集线程函数
BOOL StopDeviceAD_Half() // 采集线程的终止函数
```

```
UINT DrawWindowProc () // 绘制数据线程
```

```
void CADDoc:: OnStopDeviceAD () // 终止采集函数
```

在该工程级源代码的可执行程序中, 选择菜单中的[设备管理] [半满方式采集]后, 开始采集即可使用此方式。

第二节、使用中断方式实现该功能

中断方式是利用 FIFO 的半满信号产生硬件中断申请, 强迫中央处理器以最快的速度去进行数据传输, 其执行级别要比程序方式优先得多, 所以利用中断方式采集数据, 其吞吐率要比程序方式高很多。

需要注意的是, 由于中断方式采用了多缓冲级链的方式, 因此每次接受到中断事件后, 一定要注意 `ReadDeviceIntAD` 函数的返回值, 如果返回值不为 0, 则必须该次事件之下, 再反复用 `ReadDeviceIntAD` 函数将其缓冲区读空为止, 即其返回值必须为 0, 才能允许程序再去接管下一次中断事件。(注意下列程序中 `do while` 语句)

下面只是简要的策略说明, 其详细应用实例请参考 Visual C++测试与演示系统, 您先点击 Windows 系统的[开始]菜单, 再按下列顺序点击, 即可打开基于 VC 的 Sys 工程。

[程序] [阿尔泰测控演示系统] [Microsoft Visual C++] [高级代码演示] [高级测试演示源程序]

然后, 您着重参考 `ADDoc.cpp` 源文件中以下函数:

```
void CADDoc:: OnStartDeviceAD () // 采集线程和处理线程的启动函数
```

```
BOOL StartDeviceAD_Int () // 启动采集线程函数
```

```
UINT ReadDataThread_Int() // 采集线程函数
```

```
BOOL StopDeviceAD_Int() // 采集线程的终止函数
```

```
UINT DrawWindowProc () // 绘制数据线程
```

```
void CADDoc:: OnStopDeviceAD () // 终止采集函数
```

在该工程级源代码的可执行程序中, 选择菜单中的[设备管理] [中断方式采集]后, 开始采集即可使用此方式。

附录 A LabVIEW/CVI 图形语言专述

第一章 图形化编程语言 LabVIEW 环境及其开放性

图形化编程语言 LabVIEW 是著名的虚拟仪器开发平台。LabVIEW 是美国国家仪器公司(National Instrument)推出的一种基于图形开发、调试和运行程序的集成化环境, 是目前国际上唯一的编译型的图形化编程语言。在以 PC 机为基础的测量和工控软件中, LabVIEW 的市场普及率仅次于 C++/C 语言。LabVIEW 开发环境具有一系列优点, 从其流程图式的编程、不需预先编译就存在的语法检查、调试过程使用的数据探针, 到其丰富的函数功能、数值分析、信号处理和设备驱动等功能, 都令人称道。本文对 LabVIEW 开发环境及其开放性作一简述。

第一节 LabVIEW 概述

LabVIEW 使用了一种称为 G 的数据流编程模式, 它有别于基于文本语言的线性结构。在 LabVIEW 中执行程序的顺序是由块之间的数据流决定的, 而不是传统文本语言的按命令行次序连续执行的方式。

LabVIEW 程序称为虚拟仪表(Vitual Instrument)程序, 简称 VI。VI 包括 3 个部分: 前面板、框图程序和图标/连接口。前面板用于输入数值和观察输出量。

输入量被称为 Controls, 输出量被称为 Indicators。用户可以使用许多图标, 如旋钮、开关、文本框和刻度盘等来使前面板易看易懂。如图 1 所示, 它是一个温度计程序(Thermometer VI)的前面板。



图 1 温度计的前面板

每一个前面板都伴有一个对应的框图(block diagram)程序。框图程序使用图形编程语言编写, 可以把它理解成传统程序的源代码。框图中的程序可以看成程序节点, 如循环控制、事件控制和算术功能等。这些部件用连线联接, 以定义框图内的数据流动方向。上述温度计程序的框图程序如图 2 所示, 框图程序的编写过程与人的思维过程非常接近。LabVIEW 提供的 3 类可移动的图形化工具模板用于创建和运行程序, 它们是工具(Tools Palette)、控制(Controls Palette)和功能(Functions Palette)等。工具模板用于创建、修改和调试程序(如连线、着色等); 控制模板用来设计仪器的前面板(如增加输入控制量和输出指示量等); 功能模板用来创建相当于源代码的 LabVIEW 框图程序(如循环、数值运算、文件 I/O 等)。LabVIEW 平台的特点可归结为以下几个方面:

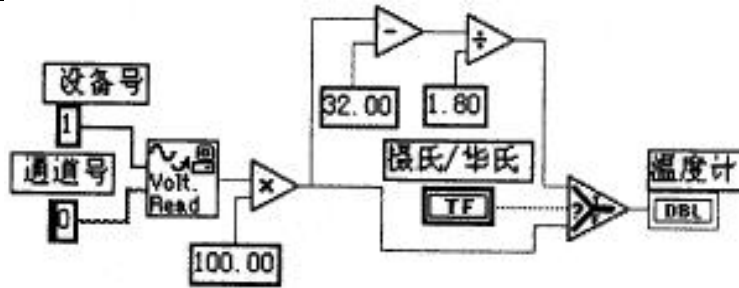


图 2 温度计的框图程序

- (1)图形编程方式: 使用直观形象的数据流程图式的语言书写程序源代码;
- (2)提供程序调试功能, 如设置断点或探针, 单步执行, 语法检查等;
- (3)拥有数据采集、仪器控制、分析、网络、ActiveX 等集成库;
- (4)继承传统编程语言结构化和模块化的优点, 这对于建立复杂应用和代码的可重用性来说是至关重要的;
- (5)提供 DLL 库接口、CIN 节点以及大量的仪器驱动器、网络通信 VIs 与其它应用程序或外部设备进行连接;
- (6)采用编译方式运行 32 位应用程序;
- (7)支持多种系统平台, 如 Macintosh、HP-UX、SUN SPARC 和 Windows 3.x/95/NT 等, LabVIEW 应用程序能在上述各平台之间跨平台进行移植;
- (8)提供大量的函数库及附加工具。如数学函数、字串处理函数、数组运算函数、文件 I/O、高级数字信号处理函数、数据分析函数、仪器驱动和通信函数等。

第二节 程序设计结构

(1)层次化结构

LabVIEW 是模块化程序设计语言, 用户可以把一个 VI 程序创建成自己的一个图标/接口(即 VI 子程序), 然后被其它 VI 程序所调用。用这种方法可设计出一个有层次关系的 VIs 或子 VIs, 而且调用阶数是无限制的。

(2)并行工作

LabVIEW 是一个多任务的软件系统, 当创建具有同步工作的程序块时, 就可交互地运行并行 VIs 程序。

(3)常规语法结构: While Loops, For Loop, Case 结构, 顺序结构等;

(4)基于文本的公式结(Formula Node)

公式结是一种用于书写数学公式的文本编辑框。

第三节 LabVIEW 的运算形式

(1)模块化图标运算

LabVIEW 中的图标/连接口表示一定的函数功能, 将若干个图标/连接口组合起来就可进行有关运算, 如算术、布尔逻辑、比较和数组运算、数值运算(三角函数、对数等)、字符串运算和文件 I/O 等;

(2)公式运算

使用公式结运行数学公式。公式结包含一个或多个公式表达式, 各公式之间用分号";"隔开。公式表达式使用了一种类似于大多数基于文本编程语言(如 BASIC 语言)的算术表达式的语法。如图 3 所示, 输入变量为 m、b 和 x, 经公式结运算后的输出变量为 y1 和 y2。公式结中使用的变量或公式的数量是无限制的。

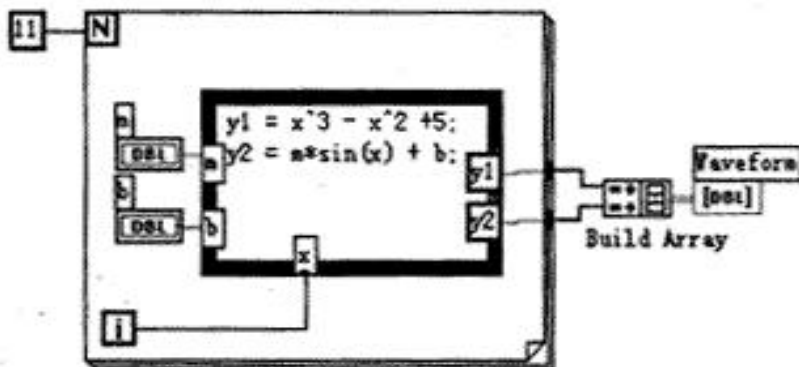


图 3 公式结运算例子

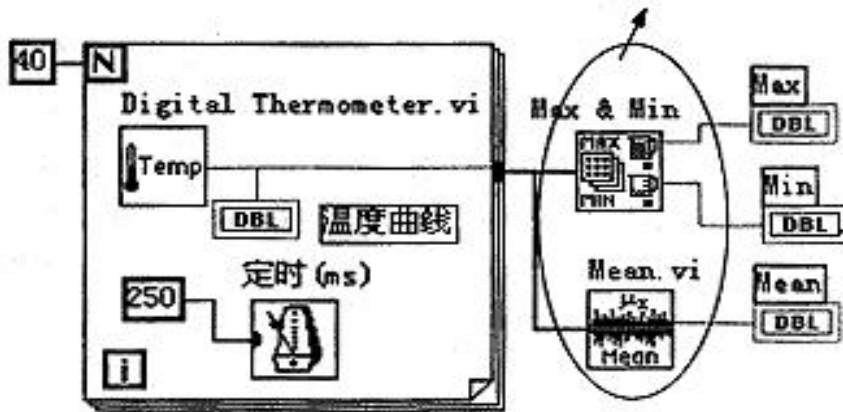


图 4 使用功能子模块进行温度曲线分析

(3) 使用集成库的功能子模板完成运算

LabVIEW 中集成了大量的生成图形界面的模板, 丰富实用的数值分析、数字信号处理功能, 以及多种硬件设备驱动器(包括 RS232、GPIB、VXI、DAQ 卡和网络等)。用户不需了解有关运算细节就能直接使用这些功能子模块, 这对于编程工作来说, 可节省了大量的时间开销。如图 4 所示, 使用两个功能子模块进行温度曲线分析, 以求出数组的最大值、最小值和平均值。

(4) 通过链接 DLL 形式的代码进行运算

LabVIEW 提供 DLL 库接口和 CIN 节点来使用户有能力在该平台上使用其它软件开发平台生成的模块。即用户可通过其它开发平台(如 BC++)建立一个子例程, 并生成动态链接库 DLL, 然后与 LabVIEW 框图程序进行链接。LabVIEW 的这一开放性, 为用户自行编写某些软件模块提供了方便。如用户可通过 C++/C 语言为某一新设备开发通信及驱动程序, 或编写一控制算法软件, 然后链入 LabVIEW 程序。

第四节 LabVIEW 的开放性

LabVIEW 是开放型的开发环境, 它拥有大量的与其它应用程序进行通信的 VI 库。因此, LabVIEW 可从众多的外部设备获取或传送数据, 这些设备包括 GPIB、VXI、PXI、串行设备、PLCs、和插件式 DAQ 板等; LabVIEW 甚至可以通过 Internet 取得外部数据源。

(1) DLLs

在 Windows 或其它平台下调用内部或外部的 DLL 形式的代码或分享其它平台(包括 Windows)中的库资源; 使用 CodeLink, 同样可自动分享在 LabWindows/CVI 中开发的 C 程序库;

(2) ActiveX, DDE, SQL

使用自动化 ActiveX、DDE 和 SQL, 与其它 Windows 应用程序一起集成用户的应用程序;

(3) 远程通信: Internet, TCP/IP

使用 TCP/IP 和 UDP 网络 VIs, 与远程应用程序进行通信; 在用户的应用程序中融入 e-mail、FTP 和浏览器等; 通过远程自动控制 VIs, 可远程操作其它机器上的分散 VIs 的执行。

第五节 调试工具

(1) 语法检查: 如果程序有错, 则无需编译, 工具栏的运行按钮就会出现一个折断的箭头。点击该箭头, 就会给出错误列表信息。

(2) 运行灯高亮: 运行灯高亮用于在单步模式下跟踪框图程序中的数据流动。

(3) 单步执行: 按顺序一个节点一个节点地执行程序。

(4) 探针: 探针工具用来查看程序流经某一根连线上的数据。

(5) 断点: 设置断点可在程序的某一地方终止程序的执行, 以观察调试部分的执行结果。

综上所述, 列出 LabVIEW 的开发环境表, 如表 1 所示。

第六节 工具软件包

NI 公司及其协作单位提供众多的软件工具箱和支持软件, 用于扩展支持 LabVIEW。这些工具软件包有:

(1) 常用工具箱

Application Builder: 创建可单独运行的应用程序;

控件与指示器

按钮/开关 LED, 滑块/数显, 计量器/刻度盘/旋钮, 水槽/温度表, 曲线图/图表, 表格/数组, 密度图, 菜单/列表/环, 文本框;

仪器控制

GPIB, VXI, Serial, CAMAC, PLC 等 600 多种仪器驱动器;

文件 I/O 电子表格, 二进制, ASCII 码, 日志;

开放性联接

Internet, SQL, TCP/IP, Activex, DLLs, DDE 等;

数据采集

DAQ, 单点输入/输出, 波形采集/发生, 图像采集, 信号调理, 触发/定时, TTL/CMOS 输入/输出, 数字图案发生, 数字握手, 脉冲发生, 事件计数, 边界检测, 周期和脉宽测量等;

程序设计结构

While Loops, For Loop, Case 结构, 顺序结构, 基于文本的公式结;

程序设计原则

算术运算, 布尔逻辑, 数组处理, 串函数, 时间/日期函数, 多数据类型结构, 用户子例程;

分析

信号发生, 信号处理, 图象处理, 曲线拟合, 窗体, 过滤, 线性, 统计等;

优化与应用程序管理

用于存储管理和执行时间跟踪的 Profiler, 在所有平台上的 TURE 编译性能, 源代码控制, 文档打印等;

调试

断点, 探针, 单步模式, 执行高亮, 帮助窗口, 在线帮助 Test Suite:包括 600 多个仪器驱动程序软件包、连接到 30 多个本地或远程数据库的数据库连接工具、程序性能测试和分析软件等;

Test Executive:

多用途的附加软件包。使用该软件包, 可以控制程序执行的次序, 生成应用程序。按照自己的特定要求和标准来设置应用程序。在保持扩展升级兼容性的前提下, 允许用户增强操作和人机接口;

SQL:用于与本地或远程数据库的直接访问;

SPC:过程控制中统计方法应用程序库;

Internet:把 VI 程序转换成可在 Internet 上执行的应用程序;

PID:给 LabVIEW 加入复杂的控制算法。该软件包带有许多误差反馈及外部复位的 PID 算法, 同时含有超前-滞后补偿和设置点斜率生成等功能;

Picture Control: 一个多功能的图形软件包, 用于生成前面板显示, 如特殊的棒形图、饼形图和 Smith 图表等。

7 (2)分析工具箱

HIQ: 一个交互式的工作环境, 可以对数学、科学计算和工程问题的数据进行组织、可视化处理。HIQ 集成了数学运算用户接口控制、数值分析、矩阵运算及二维、三维和四维图形处理;

Signal Processing Suite: 提供数据处理功能和高级信号处理工具。如数字滤波器、1/3 倍频程分析和动态信号分析等;

G Math:算术运算、数据分析和数据可视化。如常微分方程、最优化、变换和过程控制模拟等;

Image Processing:提供图象处理功能和机器视觉功能等。

第七节 总结

LabVIEW 是开放型模块化程序设计语言, 使用它可快速建立自己的仪器仪表系统, 而又不用担心程序的质量和运行速度。LabVIEW 既适合编程经验丰富的用户使用, 也适合编程经验不足的工程技术人员使用, 所以被誉为工程师和科学家的语言。

第二章 LabView 驱动程序接口


关于 LabView 的驱动程序, 我们以两种方式提供, 即内嵌式驱动程序和外挂式驱动程序。

外挂式驱动程序的提供主要是为了解决 LabView 版本的问题, 因为此种方式在我公司的帮助之下, 可以由用户自己直接创建 (当然我们已提供了 LabView5.0 环境下现成的接口定义文件), 这样就避免了 LabView 版本升级所带来的新问题。而内嵌式驱动程序*.llb 则很难做到上下版本的完全兼容, 但是此种驱动程序却有外挂式驱动程序所没有特点, 因为用内嵌式驱动程序, 您不必了解各功能模块间的控制时序和工作机理, 而以单一图标即实现了某一独立功能如 AD 采样。他体现的宗旨是方便、快捷、简单。

第一节 内嵌式驱动程序介绍

此功能由于 LabView 自身版本兼容的问题, 我们不便提供内嵌式驱动, 如果用户确有此要求, 请与我们的代理商或公司总部联系, 但我们不保证完全免费。

使用内嵌式驱动异常方便、快捷、简单。首先您能 LabView 的 Functions 模板中直接找到它, 如图 5。在这里您可以得到 “GetAD.VI”、“SetDA.VI”、“GetDI.VI”、“SetDO.VI” 等相应的设备驱动器, 比

如您要实现通过本设备实现 AD 采样，您只点击“GetAD.VI”驱动器，然后往 Diagram 窗口一放置，即可得到形如  的单个图标，它有许多诸如元器件或芯片的所具有的管脚，如图 6：

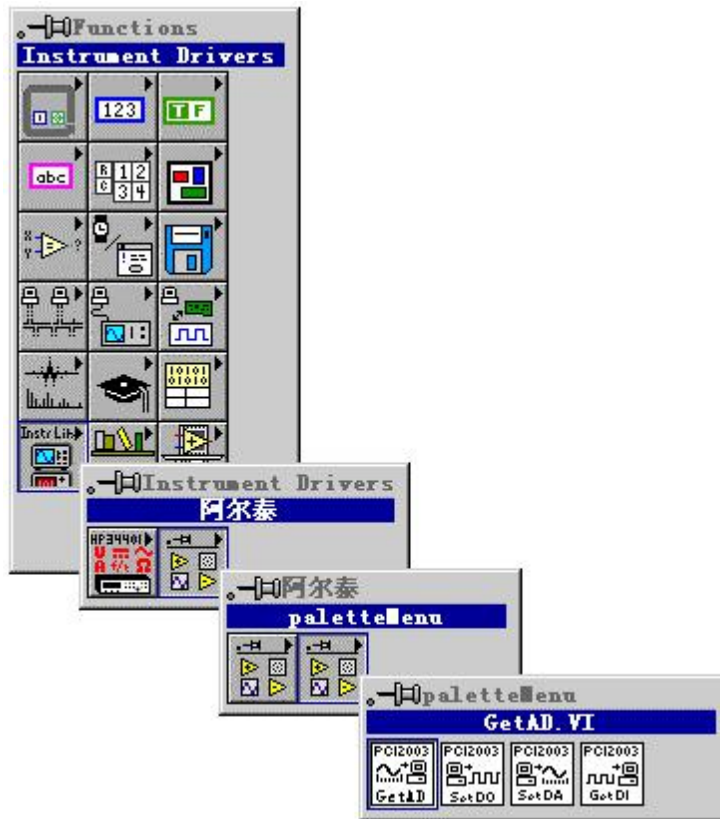


图 5 如何在 Functions 模板中获得 AD 采样驱动器

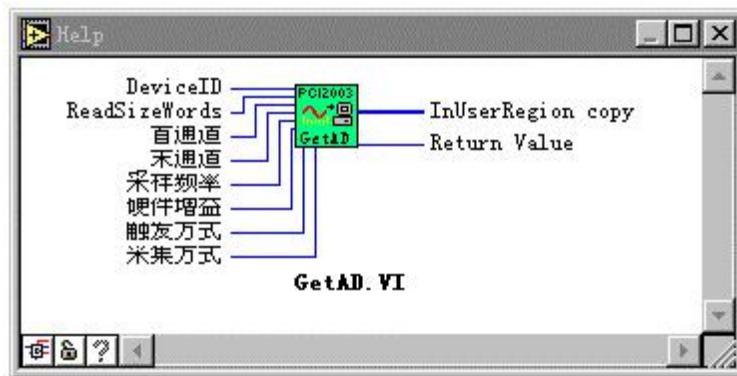


图 6 GetAD.VI 模拟数据采集驱动程序

当您取得此图标后，往 Diagram 窗口一放，然后将其输入输出端连接相应的控件或常量，即可开始 AD 数据采集。

需要注意的是，在我公司的所有此类设备驱动器图标中，其左边的管脚一般为输入端，右边的管脚一般为输出端。上下两边则根据实际需要合理分配。在实际连接时，您应将鼠标操作方式设为“Connect Wire”，如图 7，当您把鼠标移到这个驱动器图标具有连线管脚的一边时，即会出现该管脚名的自动提示，它指明了该管脚的功能。如图 8，当您把鼠标移动左上方第一个管脚时，即自动出现了



图 7 设置鼠标操作方式



图 8 自动提标

提标，指明该管脚用于指定设备的 ID 号即 DeviceID。值得用户高兴的是，每一根连续管脚不管是输入还是输出都有一个类型相匹配的部件与其对应，而且具有设备能正常运行的默认值。那么怎样使用这些默认部件和默认值呢？方法很简单，对于输入管脚（即带有小方块的管脚，如图 8 中，左边的管脚），只须用鼠标对准它，然后单击右键，如图 9 选择 Create Control 命令即可。

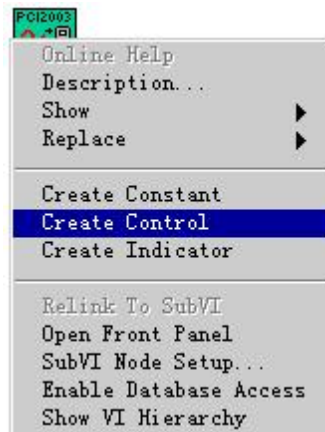


图 9 选择 Create Control

而对于输出端，则选择 Create Indicator 即可。当连接完所有管脚之后，即可直接运行以取得 AD 数据，且显示于波形窗口中。（当然既然每个管脚都有其默认值，所以某些管脚也可不予连接，但要求必须连接的管脚除外）。



图 10 选择 Create Indicator

对于每个驱动器都可以在 LabView 中取得在线帮助。其方法是选择 Help 菜单中的 Show Help 命令，如图 11。



图 11 准备获取驱动器图标的在线帮助

然后将鼠标指向需要在线帮助的图标上，即可弹出该图标的各个管脚的定义。比如图 6 中的内容就是这样得到的。

第二节 内嵌式驱动器的原型说明

- 1. 数据采集驱动器如图 6
- 2. 开关量输出驱动器如图 12
- 3. 开关量输入驱动器如图 13

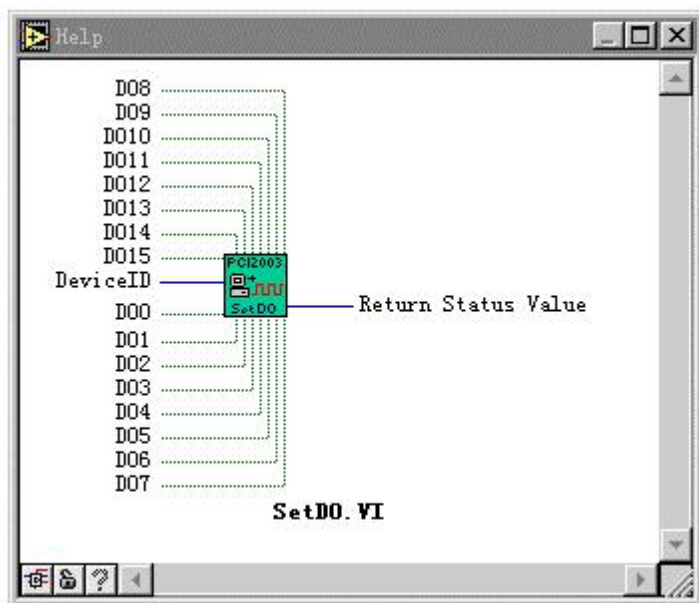


图 12 开关量输出驱动器

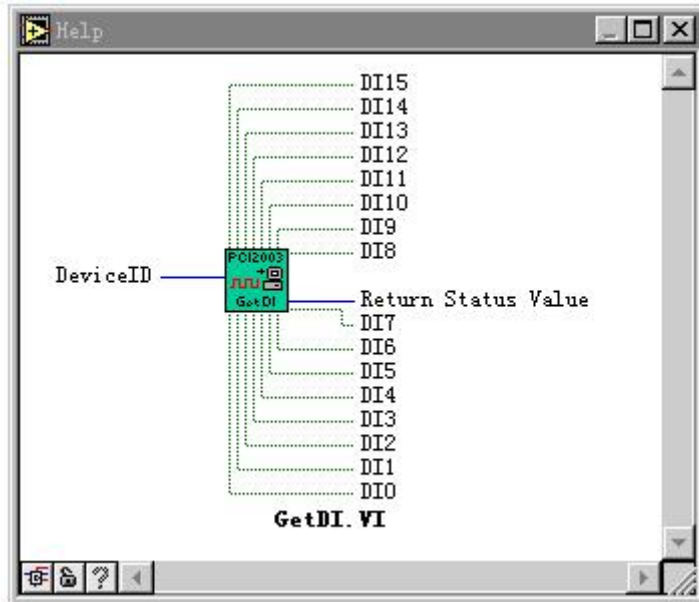


图 13 开关量输入驱动器

附注：我公司现只提供 LabView5.0 版本的内嵌驱动。具有完整内嵌驱动的设备是 PCI-2003.对于其他设备和其他 LabView 版本的内嵌驱动请需要的用户与我们协商解决。但我们不做最终承诺。而对于外挂式驱动则没有此限制，稍有 LabView 语言知识的用户，可以不必考虑使用内嵌驱动，而直接使用外挂式驱动，因为它更直接、更灵活。这不仅体现在使用上，更体现在创建上接口上，对外挂式驱动的使用，可以极大地避免版本不一致的问题。

第三节 如何使用我公司的现有的驱动接口在 LabView 中直接创建外挂式设备驱动器

要使用外挂式驱动，应知道如何创建每一个驱动接口图标。其方法很简单。下面以 PCI-2004 为范例加以说明，其他设备同理。

首先在 Functions 模板中选择 Call Library Function 子模板，如图 14。

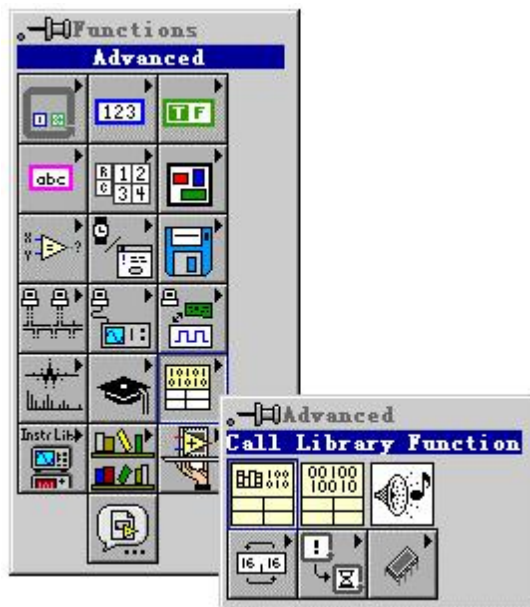


图 14 选择创建外挂式驱动的子模板

然后用户应先阅读第五章和第六章驱动接口原型定义，弄清相应接口的返回值和各参数类型，最后在 Diagram 窗口中单击鼠标左键，即出现某一外挂式驱动接口的图标 ，紧接着双击此图标，即弹出该驱动接口的配置窗口，

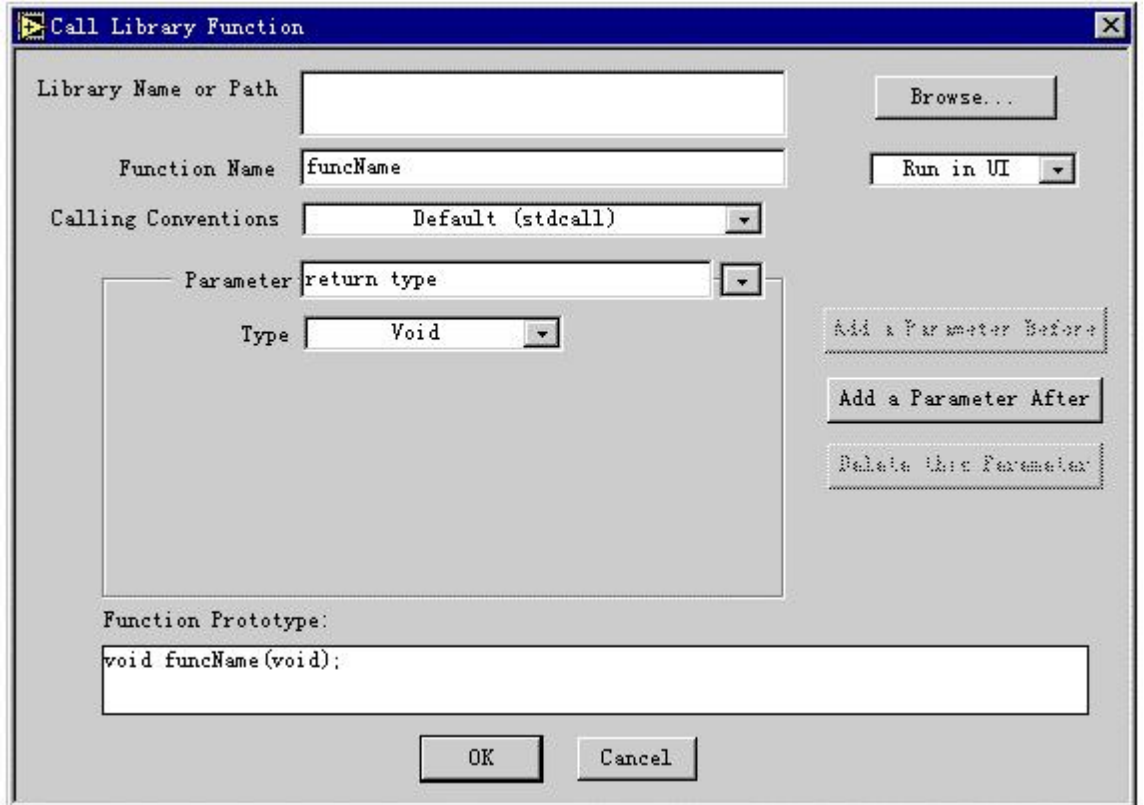


图 15 配置驱动接口的对话框

下面以创建 CreateDevice 为例，加以说明：

- 1.在 Library Name Function 栏中输入 PCI2310，以指明该接口使用的动态库名；
- 2.在 Function Name 中输入函数名，这里为 CreateDevice;但别忘了在函数前置前缀 PCI2310_
- 3.在 Parameter 的下拉列表框中选择 return type 字符串改名为 hDevice,以示返回值为设备句柄
- 4.在 Calling Conventions 中确保其为默认值 Default(stdcall)
- 5.在 Type 下拉列表框中选择以确定以上返回值的类型，此处选择 Numeric,如图 16

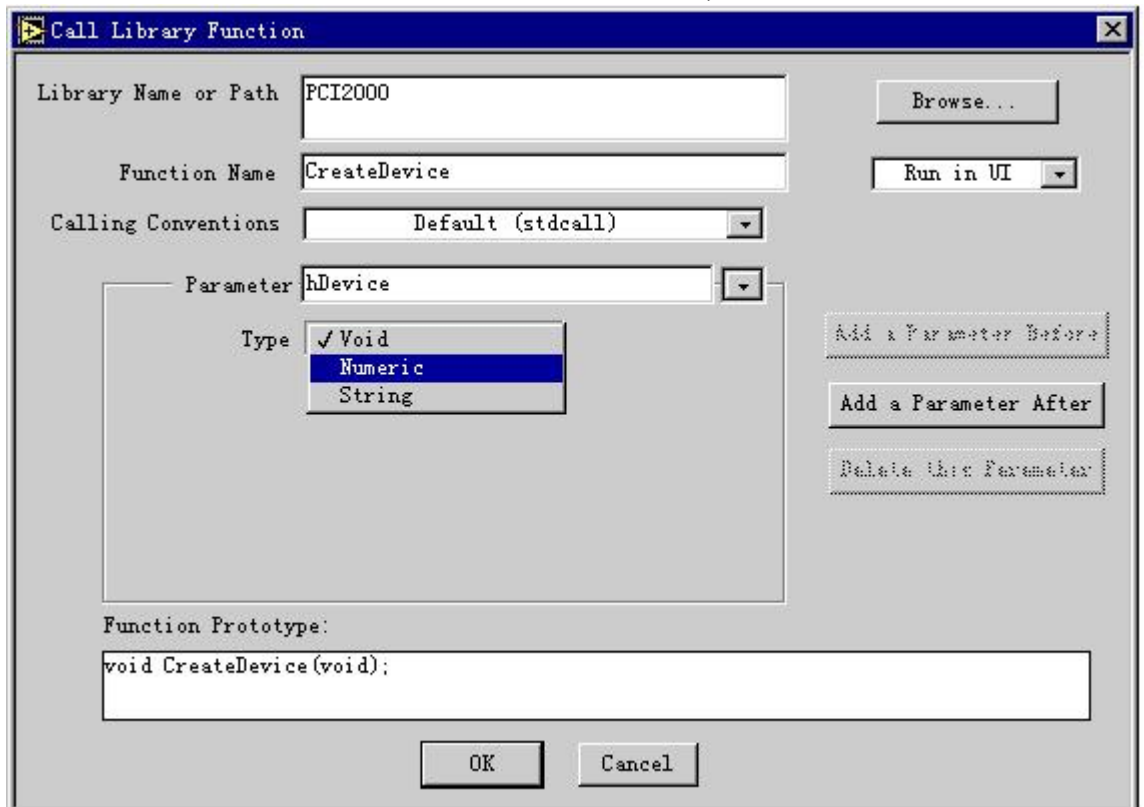


图 16 选择的参数确定基本类型

- 6.在 Data Type 中选择 Signed 32-Bit Integer 类型，如图 17。且确保 Pass 栏为 Value 选项

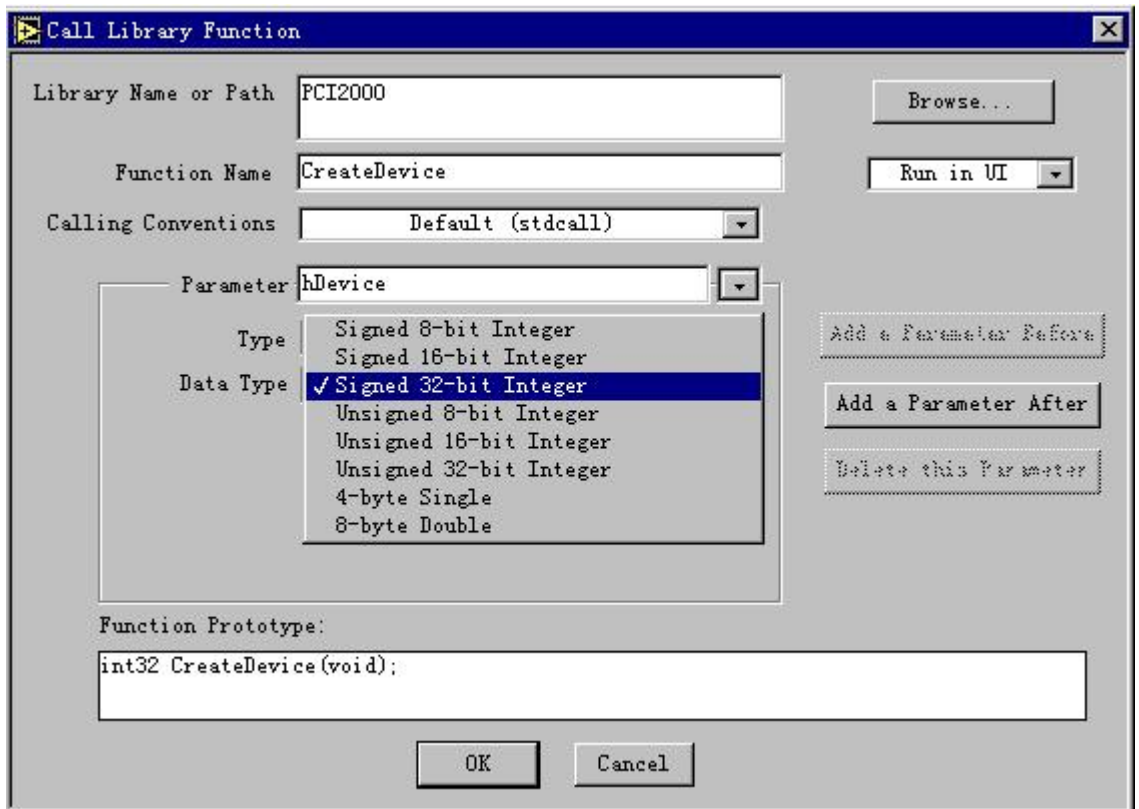


图 17 选择指定返回值的数据类型

7.继续在返回值的后面增添函数参数，用鼠标单击 Add a Parameter After 后，如图 18，然后将 Parameter 中的 arg1

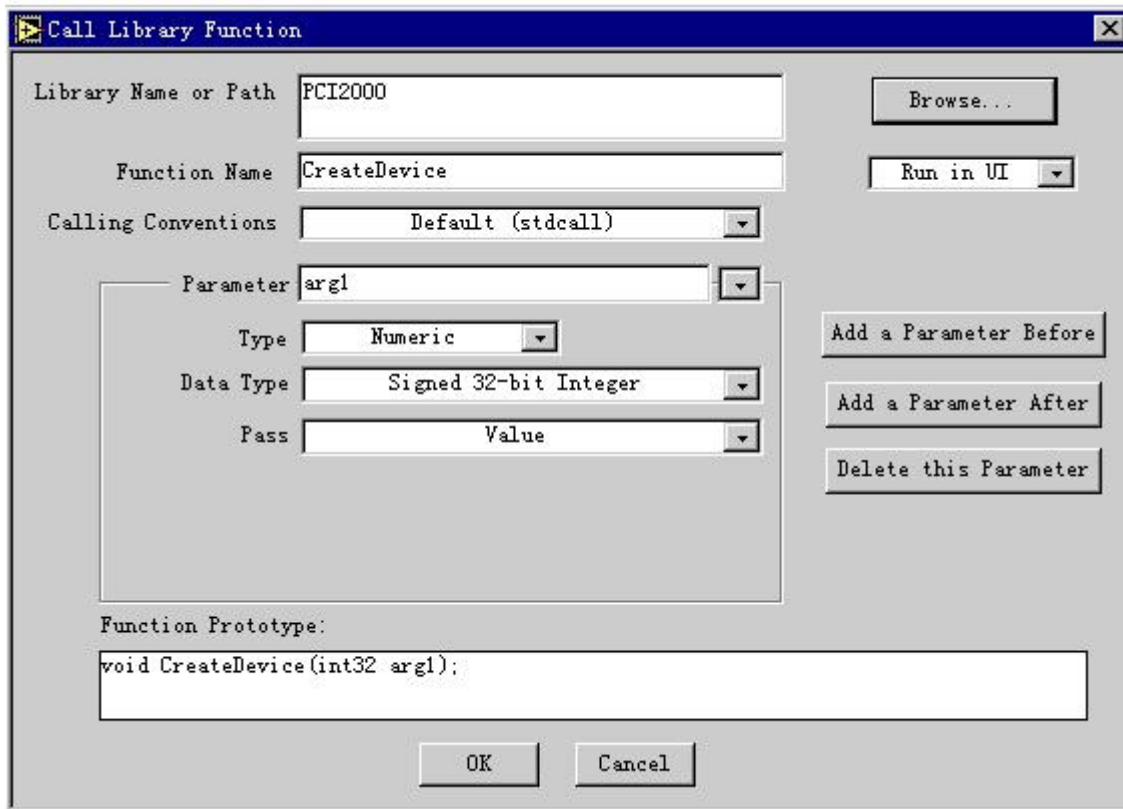


图 18 增添参数

成 DeviceID.

8.再确保 Type 为 Numeric, DataType 为 Signed 32-Bit Integer, Pass 为 Value。这样，CreateDevice 即创建成功，如图 19。

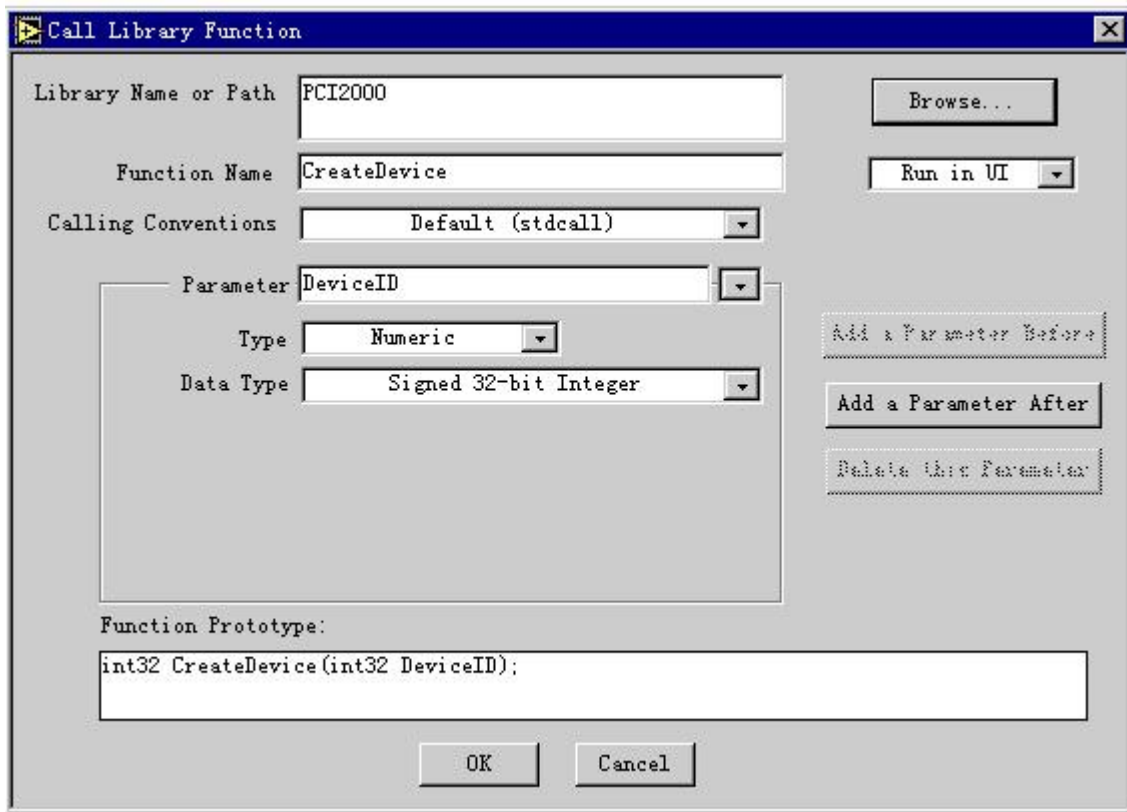


图 19 CreateDevice 创建成功后的配置情况

9.单击以上对话框的 OK 按钮,即可得到 CreateDevice 函数的接口图标如图 20



图 20 CreateDevice 函数的最终用户图标

下面再举一个例子说明如何创建一个具有指针或数组类型参数的驱动接口,如 ReadDeviceProAD_NotEmpty 由于其每个接口创建过程基本类似,这里只演示如何定义该函数的 ADBuffer 这个数组参数。

1.在 Parameter 下拉列表框中将当前参数位置定位在 hDevice 参数上,如图 21;

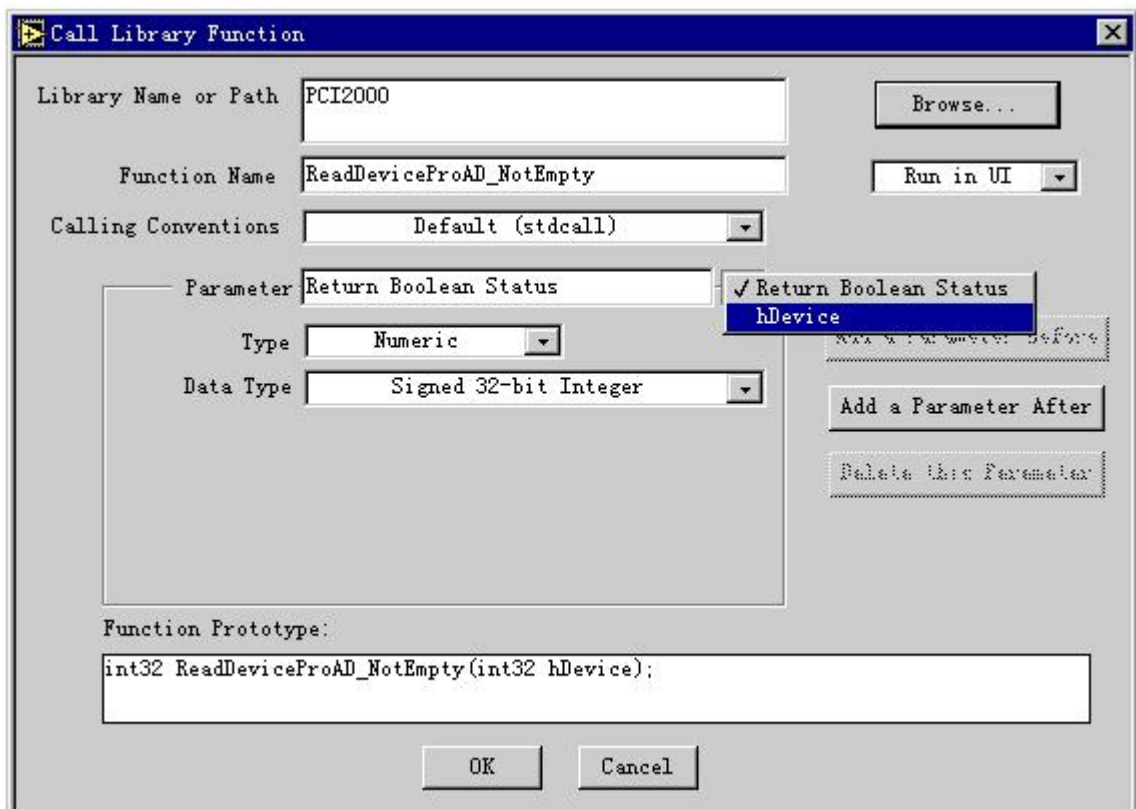


图 21 如何在某个参数这后添加另一个参数的一步

- 单击 Add a Parameter After, 即在当前参数之后添加一个参数, 然后将 Parameter 中出现的 arg2 改成实际的参数名 ADBuffer。
- 在 Type 下拉列表框中, 选择 Array 之后, 如图 22:

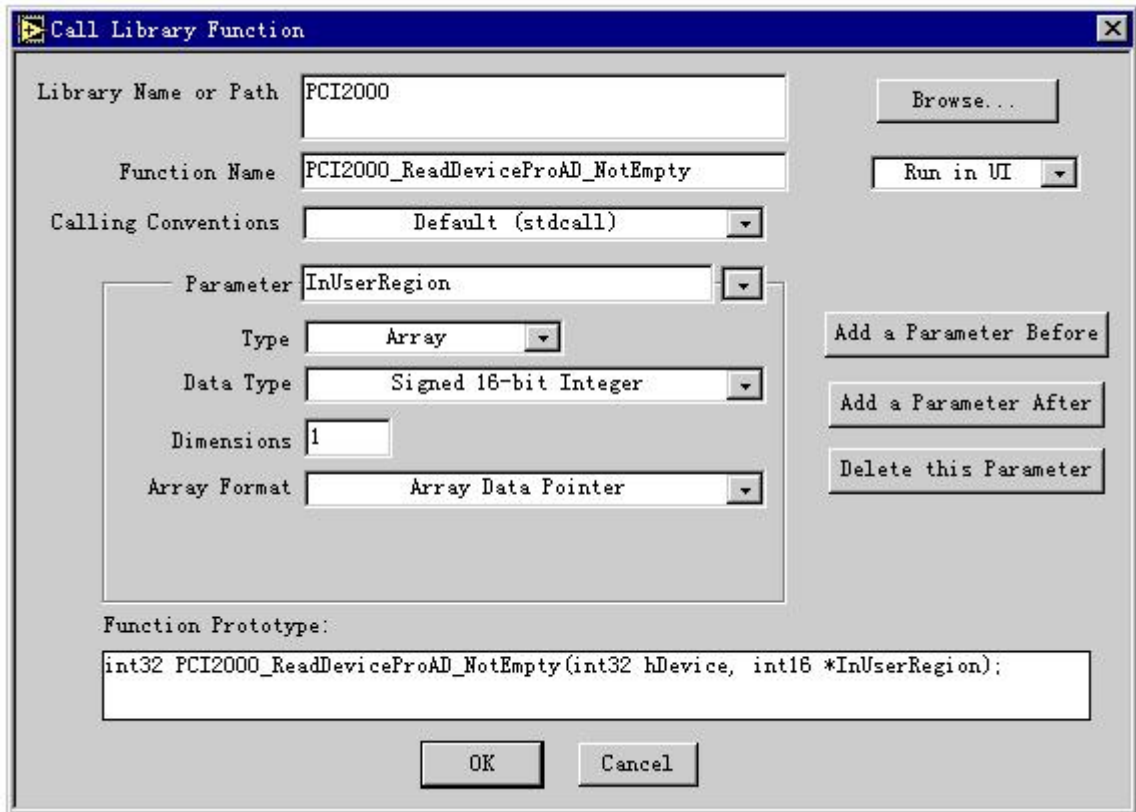


图 22 如何定义一个数组参数

- 在 Data Type 下拉列表中选择 Unsigned 16-Bit Integer, 以确保在参数为无符号 16 位数据类型。
- 在 Dimension 栏中置 1, 视该参数为 1 维数组。在 Array Format 栏中置该数组为数据类型指针。
- 在接着单击 Add a Parameter After 按钮, 添加该函数的最后一个参数 ReadSizeWords. 如图 23, 即该驱动接口的创建即告成功。

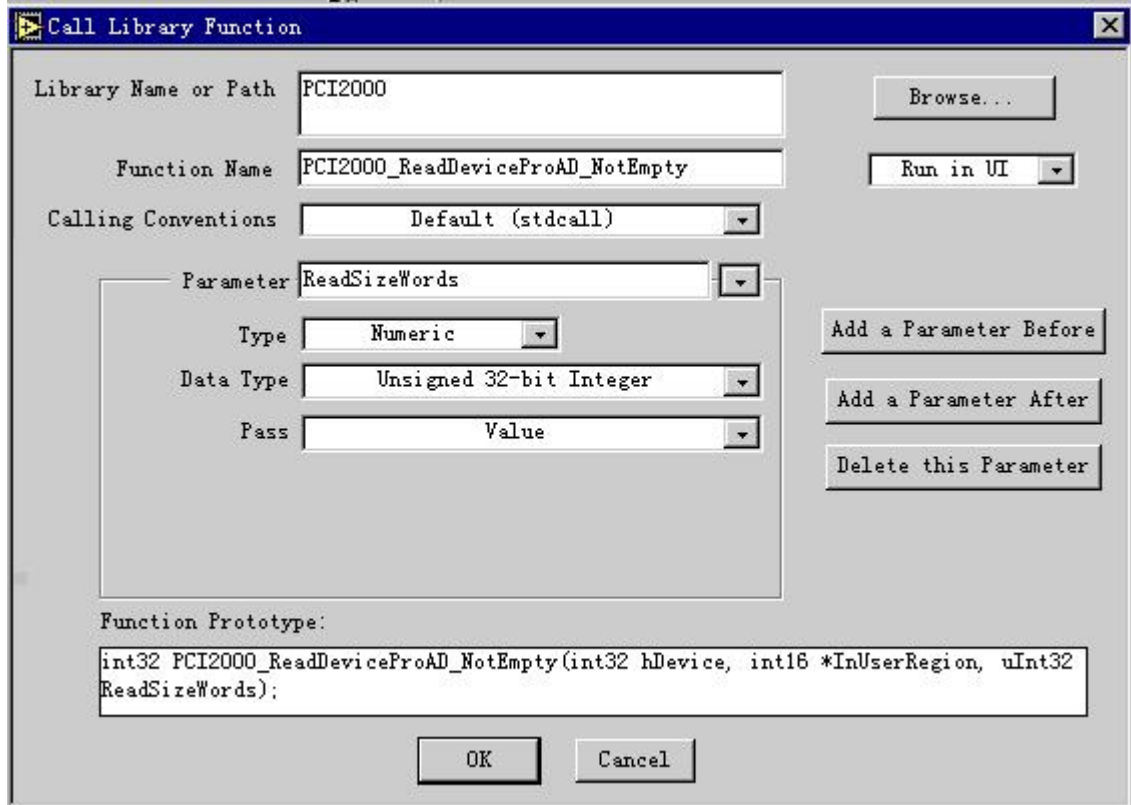


图 23 函数创建成功

7.单击 OK 按钮，即可在 Diagram 窗口形成驱动器图标，如图 24。

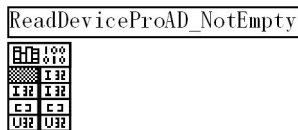


图 24 最终形成的驱动器图标

关于外挂式驱动图标都有许多共同的特点，它由许多小格子构成，最上方的格子（即带有 100010 图案的格子）只是一种标识，指明这类图标的执行代码来源于驱动程序库。除了这个格子以外，其他格子完全被中间的由上至下的较粗的黑色竖线分为左右两部分格子。每一横向上对齐的左右两个格子则共同对应于函数的返回值或某一个参数。第一行格子对应于函数的返回值（如果函数没有返回值，则这行格子全为模糊图案），第二行格子对应第一个参数，第三行格子对应于函数的第二个参数，以此类推。左边的格子均用于输入端，右边的格子均用于输出端。如果这个参数的值需要返回，那么它应该是指针类型，且可以从图标右边输出到某个部件如 Indicator，比如 ReadDeviceProAD 函数中的 ADBuffer 参数。如果这个参数不需要返回，则只须从左边输入，而右边则不必理会，如 CreateDevice 函数中的 DeviceID 参数。如果有返回，则象图 20 那样，右边的格子则出现表明返回值数据类型的标识。而左边输入端格子无论如何都是模糊的，因为任何函数都不能对返回值进行某种输入操作。

其格子里面的各种标识图案如文字等以及颜色则简要说明了该参数的数据类型。I32 表示有符号的 32 位整型参数，U32 则表示无符号的 32 位整型参数，其他位数的整型参数以此类推，[]则表示指针或数组参数，其他位数的整型指针或数组以此类推。而从颜色上看，蓝色的凡指整型数据类型，红色则指浮点数。

第四节 如何使用我公司为用户已定制好的外挂式驱动器

如果您使用的 LabView5.0,或者是在以上版本的 LabView 中能打开 Samples\LabView 目录下的 PCI2310.VI 文件，那么您可得所有驱动程序接口的最原始定义。之所以称之为“最原始”是指这个文件中的每一个接口单元与第四、五章中的函数原型说明完全一致，且这个 VI 文件不能用于运行方式。用上这些接口，用户完全可以经自己的组合设计即可实现本驱动的所有功能。可以在 LabView 开发环境中打开 PCI2310.VI 文件，将鼠标变成箭头状，单击您所需要的模块(形如下面的图标)，然后用键盘命令 Ctrl+C 将其复制到剪切板中，然后再回到您的系统开发环境中，最后在您要放置这个模块的位置上单击一下，再用键盘命令 Ctrl+V，那么这个模块即成为您开发系统的一个部件。您便可以直接连接使用。

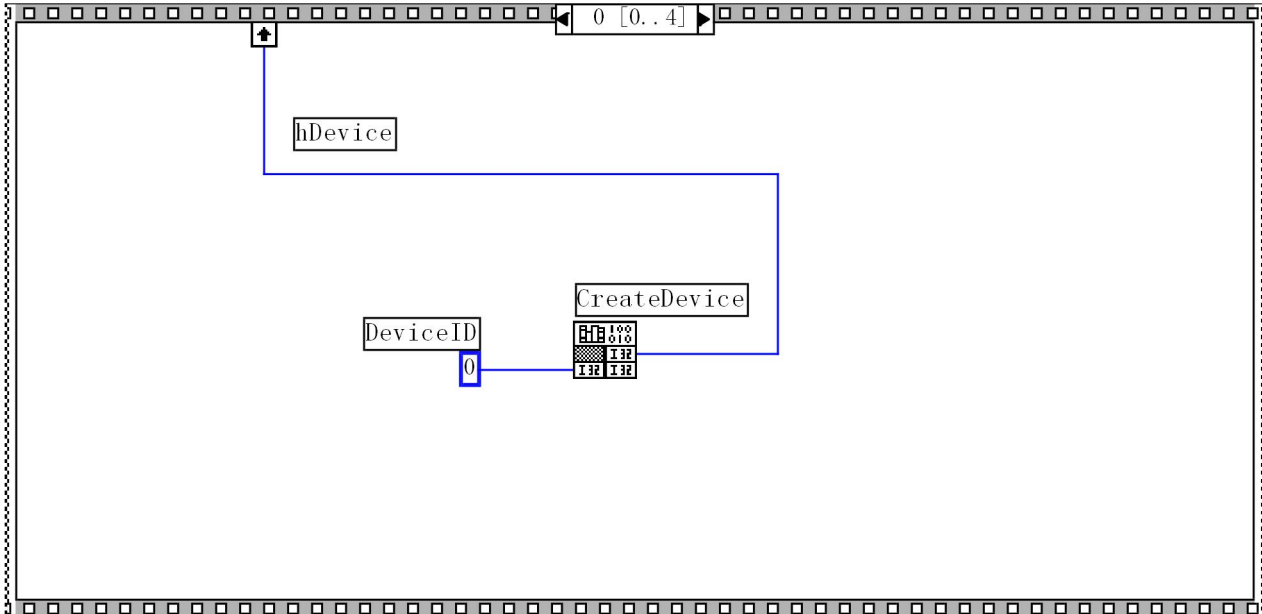


对于尚未在 PCI2310.VI 中例出的其他相关函数接口您可以按照本章第三节中的说明加以实现。不过需要注意的是您所新配置的接口应该是我公司底层驱动程序予以提供技术支持的接口，否则将会失败。但值得用户放心的只要您是我公司的合法用户，基本上所有的开放的接口都会得应有的支持。

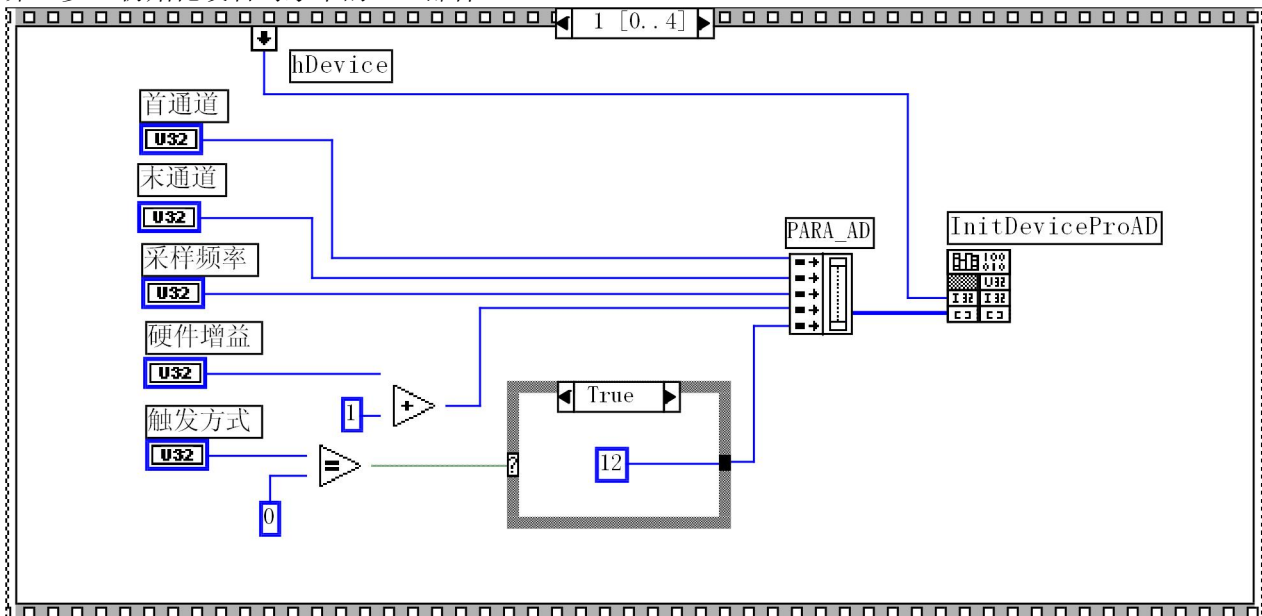
第五节 如何在 LabView 中用上层函数实现 AD 采集

此处只以程序轮询、且非空读方式加以说明，共分为五个步骤。详见 LabView\AD_Sample.VI 可执文件。

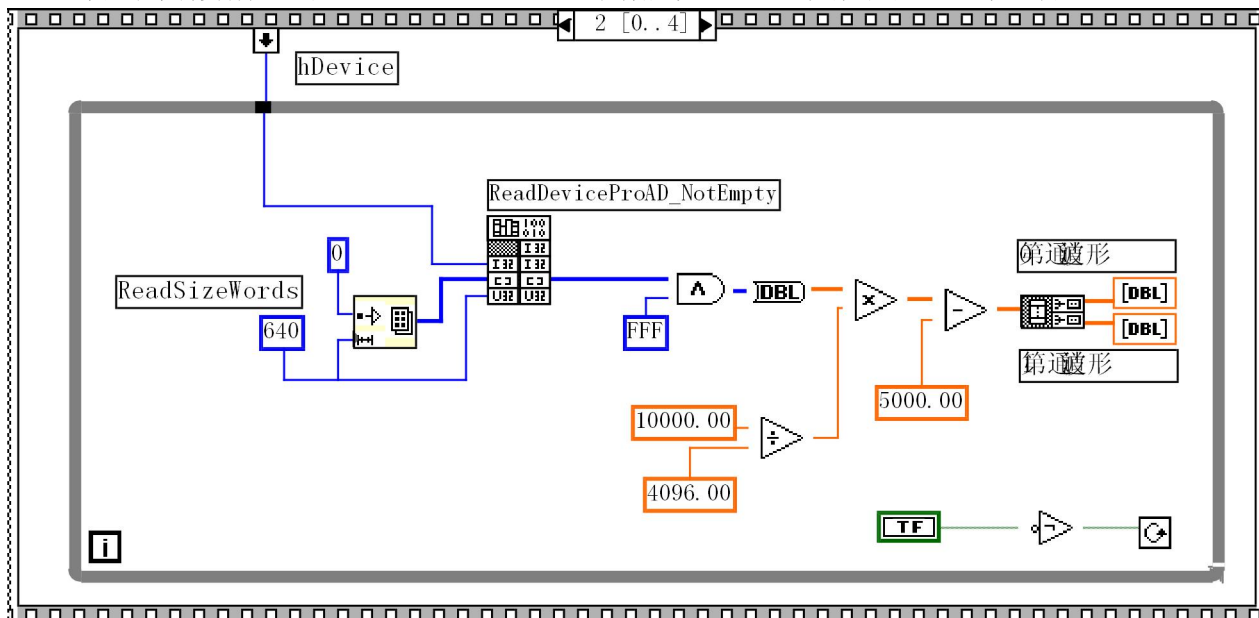
第一步 创建设备对象



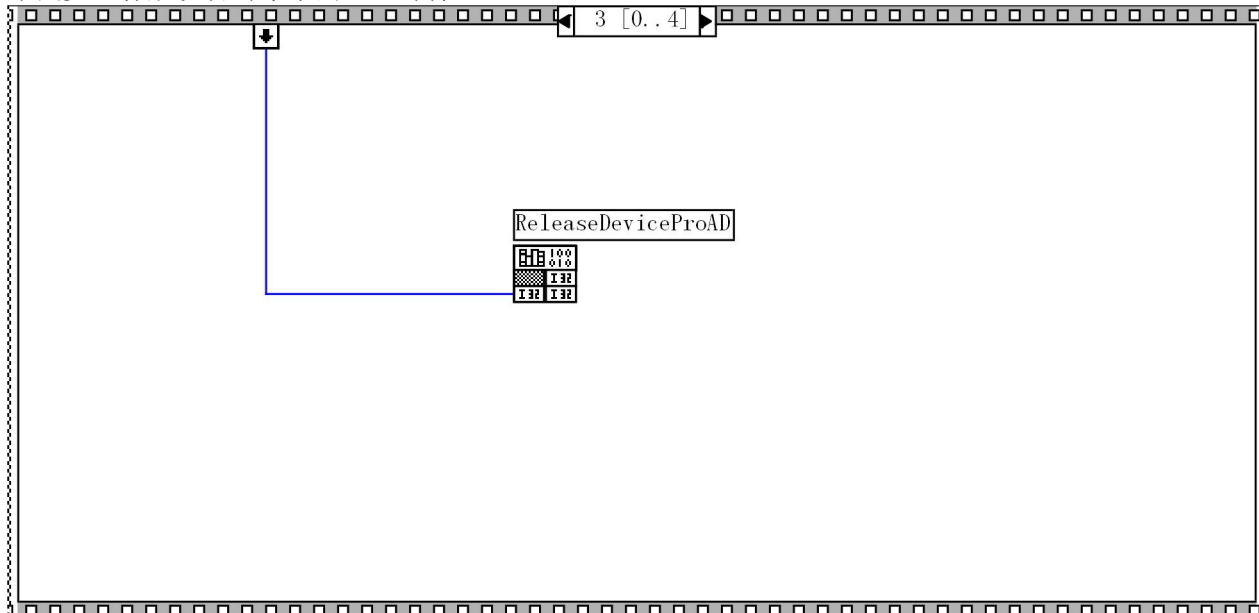
第二步 初始化设备对象中的 AD 部件



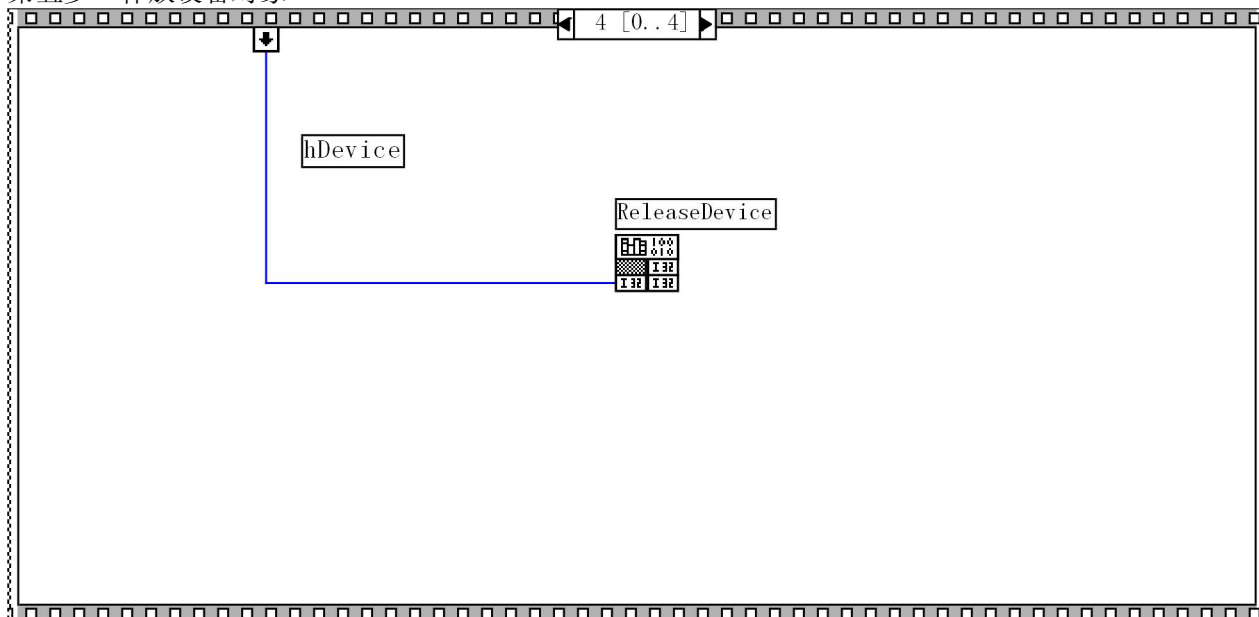
第三步 开始循环采集 AD 数据



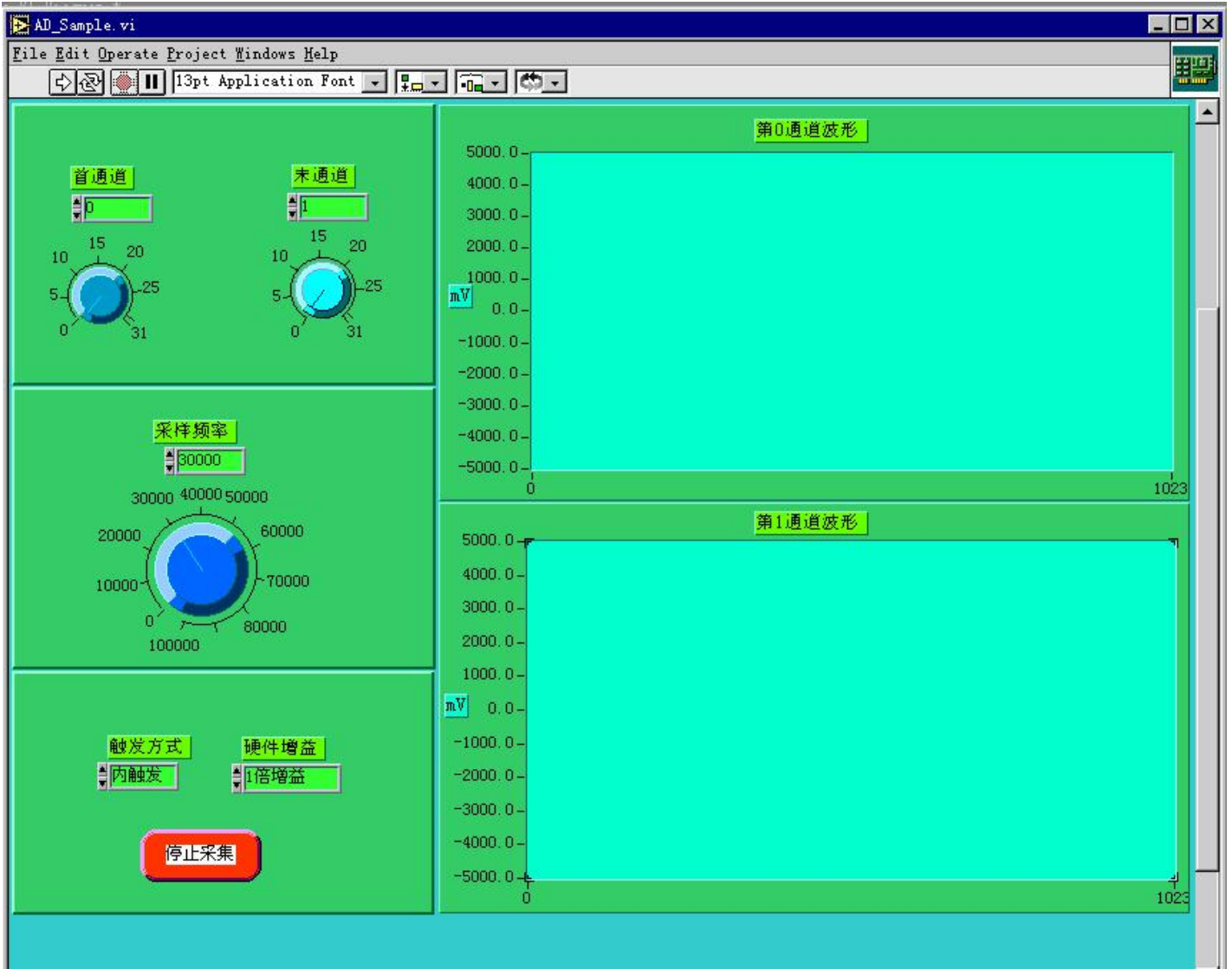
第四步 释放设备对象中的 AD 部件



第五步 释放设备对象



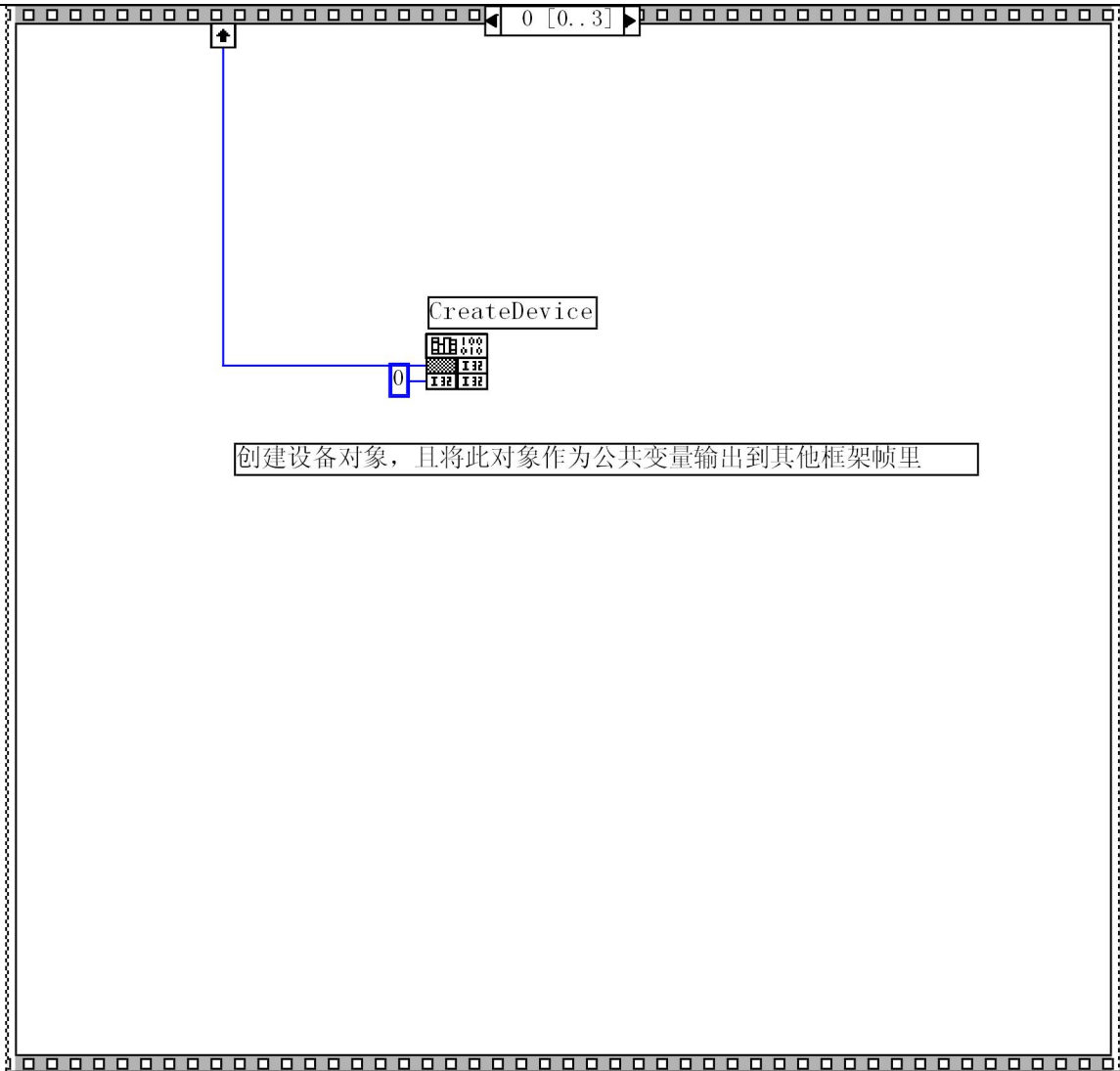
下面是 AD_Sample.VI 在采集两个通道数据时的前台控制面板:



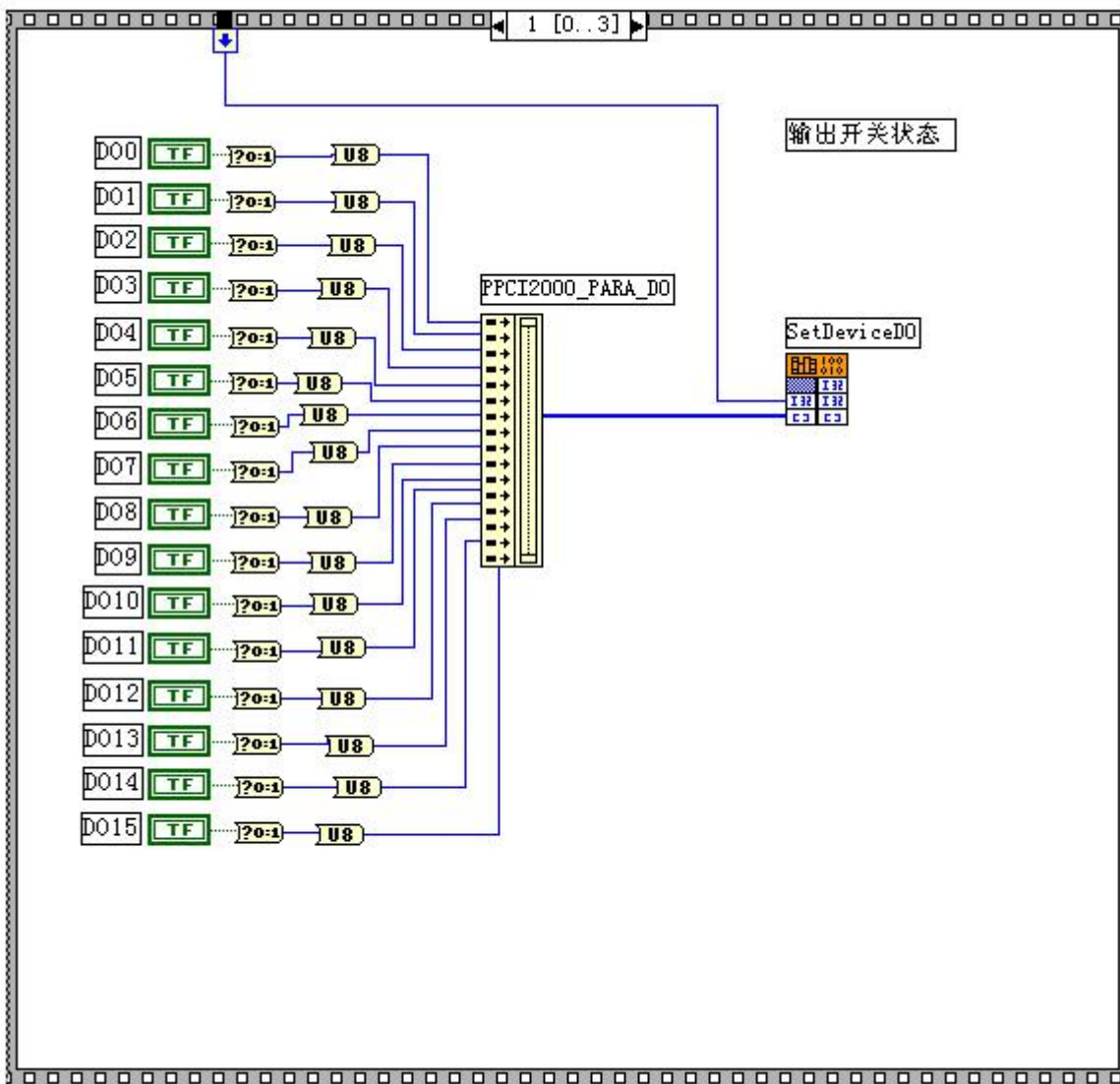
第六节 怎样用上层函数实现开关量输入输出操作

为了更好的演示开关量操作流程, 则把开关量的输出输入放在一起加以说明。详见 LabView\DIO_Sample.VI 文件, 在执行这个演示程序时, 为了能看到开关量的实际动作, 请用扁平电缆将 PCI2310 设备上的 XS2 和 XS3 的输入输出通道一一连接起来, 当用户点击左边的开关时, 右边的相对应的灯则会有变化。但是最便捷的方式就是使用 LabView\Drv.lib 这个目录下的子 VI 文件如 SetDO.VI 和 GetDI.VI 文件。使用方法是在后台流程控制面板 (Diagram) 中, 单击鼠标右键, 在弹出的浮动 Functions 菜单中选择 “Select a VI...”, 在弹出的对话框中选择 LabView\Drv.Lib 目录下的 SetDO.VI 或 GetDI.VI 文件, 即可实现象 NI 公司在 LabView 中提供的标准组件模块一样, 在您的流程图将出现一个图标, 就象一块封装得很好的芯片器件, 而各种开关量就象是这个芯片的各个管脚, 用户只须将这些管脚与控制对象或被控制对象相连接, 即可实现开关量的输出输入操作 (请详见 DIO_SubSample.VI)。

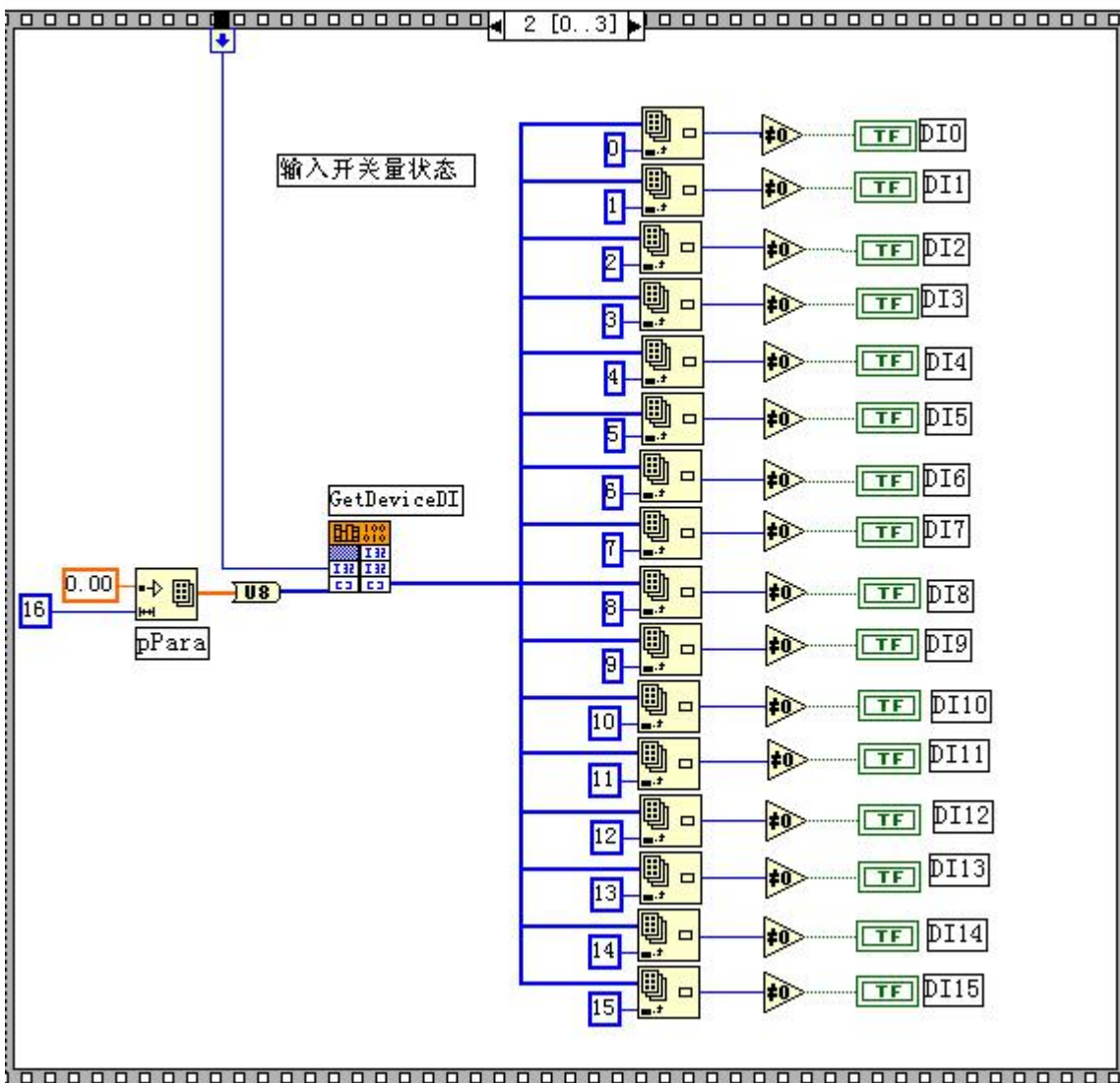
第一步 创建设备对象



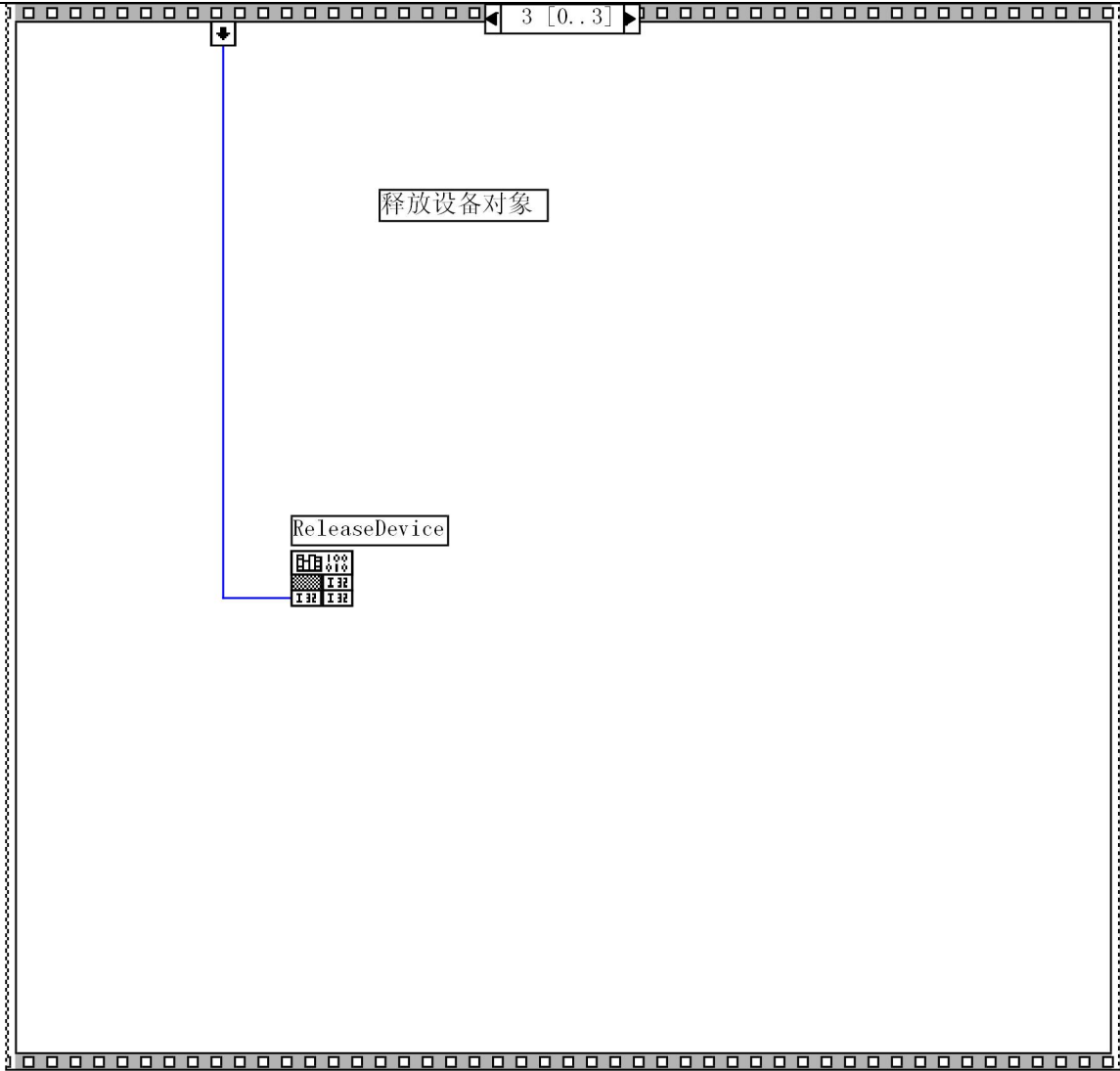
第二步 输出开关量



第三步 将输出的开关量状态通过输入端取出来



第四步 释放设备对象



下面是 DIO_Sample.VI 文件前台控制面板

