# PCI8025 数据采集卡

## WIN2000/XP 驱动程序使用说明书

阿尔泰科技发展有限公司
产品研发部修订

**请您务必阅读《使用纲要》，他会使您事半功倍!**

# 目　录

目　录 ........................................................................................................................................1
第一章 版权信息与命名约定.......................................................................................................2
第一节、版权信息 ..................................................................................................................2
第二节、命名约定 ..................................................................................................................2
第二章 使用纲要 ...........................................................................................................................2
第一节、使用上层用户函数，高效、简单..........................................................................2
第二节、如何管理PCI设备.....................................................................................................2
第三节、如何用查询方式取得AD数据................................................................................2
第四节、如何用DMA方式取得AD数据..............................................................................3
第五节、如何实现开关量的简便操作..................................................................................5
第六节、哪些函数对您不是必须的......................................................................................6
第三章 PCI即插即用设备操作函数接口介绍 ...........................................................................6
第一节、设备驱动接口函数总列表（每个函数省略了前缀“PCI8025_”） .....................7
第二节、设备对象管理函数原型说明..................................................................................8
第三节、AD程序查询方式采样操作函数原型说明............................................................10
第四节、AD硬件参数保存与读取函数原型说明................................................................16
第五节、AD模拟信号频率测量函数原型说明....................................................................16
第六节、DA模拟量输出操作函数原型说明........................................................................18
第七节、DIO数字量输入输出开关量操作函数原型说明...................................................19
第四章 硬件参数结构 .................................................................................................................20
第一节、AD硬件参数结构（PCI8025_PARA_AD） .........................................................20
第二节、AD状态参数结构（PCI8025_STATUS_AD） .....................................................22
第三节、数字量输入参数（PCI8025_PARA_DI） ............................................................23
第四节、数字量输出参数（PCI8025_PARA_DO） ...........................................................24
第五章 数据格式转换与排列规则.............................................................................................26
第一节、AD原码LSB数据转换成电压值的换算方法........................................................26
第二节、AD采集函数的ADBuffer缓冲区中的数据排放规则...........................................27
第三节、AD测试应用程序创建并形成的数据文件格式....................................................27
第四节、DA电压值转换成LSB原码数据的换算方法........................................................28
第六章 上层用户函数接口应用实例.........................................................................................28
第一节、怎样使用ReadDeviceProAD函数直接取得AD数据............................................28
第二节、怎样使用ReadDeviceDmaAD函数直接取得AD数据..........................................28
第三节、怎样使用WriteDeviceProDA函数取得DA数据...................................................28
第四节、怎样使用GetDeviceDI函数进行更便捷的数字开关量输入操作........................29
第五节、怎样使用SetDeviceDO函数进行更便捷的数字开关量输出操作.......................29
第七章 高速大容量、连续不间断数据采集及存盘技术详解.................................................29
第一节、使用程序查询方式实现该功能............................................................................30
第二节、使用DMA方式实现该功能...................................................................................31
第八章 共用函数介绍 .................................................................................................................31
第一节、公用接口函数总列表（每个函数省略了前缀“PCI8025_”） ............................31
第二节、PCI内存映射寄存器操作函数原型说明...............................................................31
第三节、IO端口读写函数原型说明.....................................................................................37
第四节、线程操作函数原型说明........................................................................................40

# 第一章 版权信息与命名约定

## 第一节、版权信息

本软件产品及相关套件均属北京阿尔泰科技发展有限公司所有，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当地代理商联系或直接与我们联系，我们将热情接待。

## 第二节、命名约定

一、为简化文字内容，突出重点，本文中提到的函数名通常为基本功能名部分，其前缀设备名如 PCIxxxx_则被省略。如 PCI8025_CreateDevice 则写为 CreateDevice。

二、函数名及参数中各种关键字缩写

| 缩写 | 全称 | 汉语意思 | 缩写 | 全称 | 汉语意思 |
|---|---|---|---|---|---|
| Dev | Device | 设备 | DI | Digital Input | 数字量输入 |
| Pro | Program | 程序 | DO | Digital Output | 数字量输出 |
| Int | Interrupt | 中断 | CNT | Counter | 计数器 |
| Dma | Direct Memory Access | 直接内存存取 | DA | Digital convert to Analog | 数模转换 |
| AD | Analog convert to Digital | 模数转换 | DI | Differential | (双端或差分) 注：在常量选项中 |
| Npt | Not Empty | 非空 | SE | Single end | 单端 |
| Para | Parameter | 参数 | DIR | Direction | 方向 |
| SRC | Source | 源 | ATR | Analog Trigger | 模拟量触发 |
| TRIG | Trigger | 触发 | DTR | Digital Trigger | 数字量触发 |
| CLK | Clock | 时钟 | Cur | Current | 当前的 |
| GND | Ground | 地 | OPT | Operate | 操作 |
| Lgc | Logical | 逻辑的 | ID | Identifier | 标识 |
| Phys | Physical | 物理的 | | | |

# 第二章 使用纲要

## 第一节、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，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InitDeviceAD、ReadDeviceProAD、SetDeviceDO等。而底层用户函数如 WriteRegisterULong、ReadRegisterULong、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，您基本上不必参考硬件说明书，除非您需要知道板上D型插座等管脚分配情况。

## 第二节、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相应的驱动函数，如 InitDeviceAD可以使用hDevice句柄以程序查询方式初始化设备的AD部件，ReadDeviceProAD (或 ReadDeviceDmaAD)函数可以用hDevice句柄实现对AD数据的采样读取，SetDeviceDO函数可用实现开关量的输出等。最后可以通过 ReleaseDevice将hDevice释放掉。

## 第三节、如何用查询方式取得 AD 数据

当您有了hDevice设备对象句柄后，便可用 InitDeviceAD函数初始化AD部件，关于采样通道、频率等参数的设置是由这个函数的pADPara参数结构体决定的。您只需要对这个pADPara参数结构体的各个成员简单赋值即可实现所有硬件参数和设备状态的初始化。然后用 StartDeviceAD即可启动AD部件，开始AD采样，然后便可用 ReadDeviceProAD反复读取AD数据以实现连续不间断采样。当您需要暂停设备时，执行 StopDeviceAD，当

您需要关闭AD设备时，ReleaseDeviceAD便可帮您实现（但设备对象hDevice依然存在）。（注：ReadDeviceProAD虽然主要面对批量读取、高速连续采集而设计，但亦可用它以单点或几点的方式读取AD数据，以满足慢速、高实时性采集需要）。具体执行流程请看下面的图 2.1.1。

## 第四节、如何用 DMA 方式取得 AD 数据

当您有了hDevice设备对象句柄后，便可用 InitDeviceAD函数初始化AD部件，关于采样通道、频率等的参数的设置是由这个函数的pADPara参数结构体决定的。您只需要对这个pADPara参数结构体的各个成员简单赋值即可实现所有硬件参数和设备状态的初始化。同时应调用 CreateSystemEvent函数创建一个内核事件对象句柄hDmaEvent赋给 ReadDeviceDmaAD的相应参数，它将作为Dma事件的变量。然后用 StartDeviceAD即可启动AD部件，开始AD采样，接着调用Win32 API函数WaitForSingleObject等待hDmaEvent事件的发生，当当前缓冲段没有被DMA完成时，自动使所在线程进入睡眠状态（不消耗CPU时间），反之，则立即唤醒所在线程，执行它下面的代码，此时您便可用 GetDevStatusAD来确定哪一段缓冲是新的数据，即刻处理该数据，至到所有的缓冲段变为旧数据段。然后再回到WaitForSingleObject，就这样反复读取AD数据即可实现连续不间断采样。当您需要暂停设备时，执行 StopDeviceAD，当您需要关闭AD设备时，ReleaseDeviceAD便可帮您实现（但设备对象hDevice依然存在）。具体执行流程请看图 2.1.2。

注意：图中较粗的虚线表示对称关系。如红色虚线表示 CreateDevice和 ReleaseDevice两个函数的关系是：最初执行一次 CreateDevice，在结束是就须执行一次 ReleaseDevice。

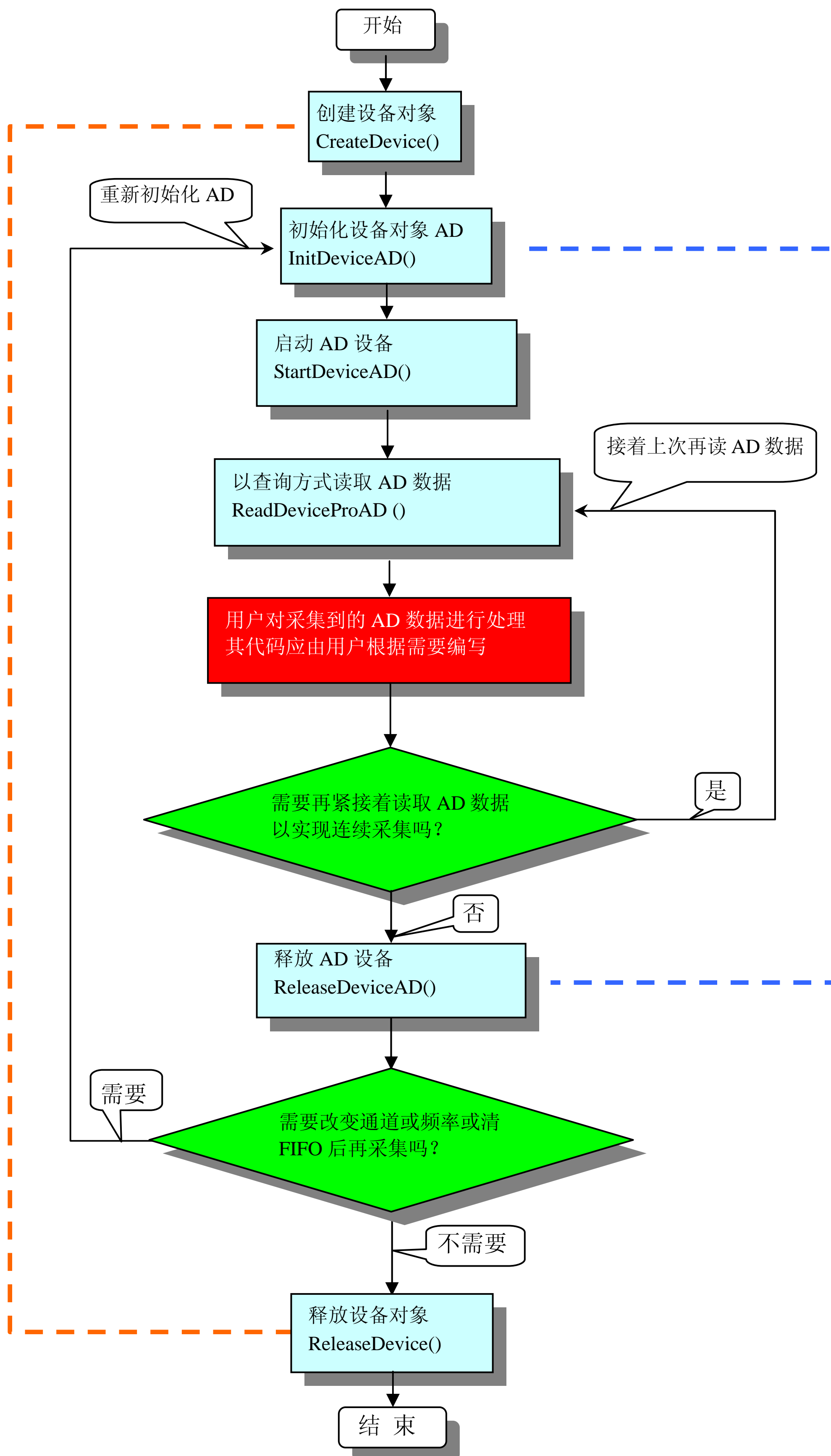



图 2.1.1 查询方式 AD 采集过程

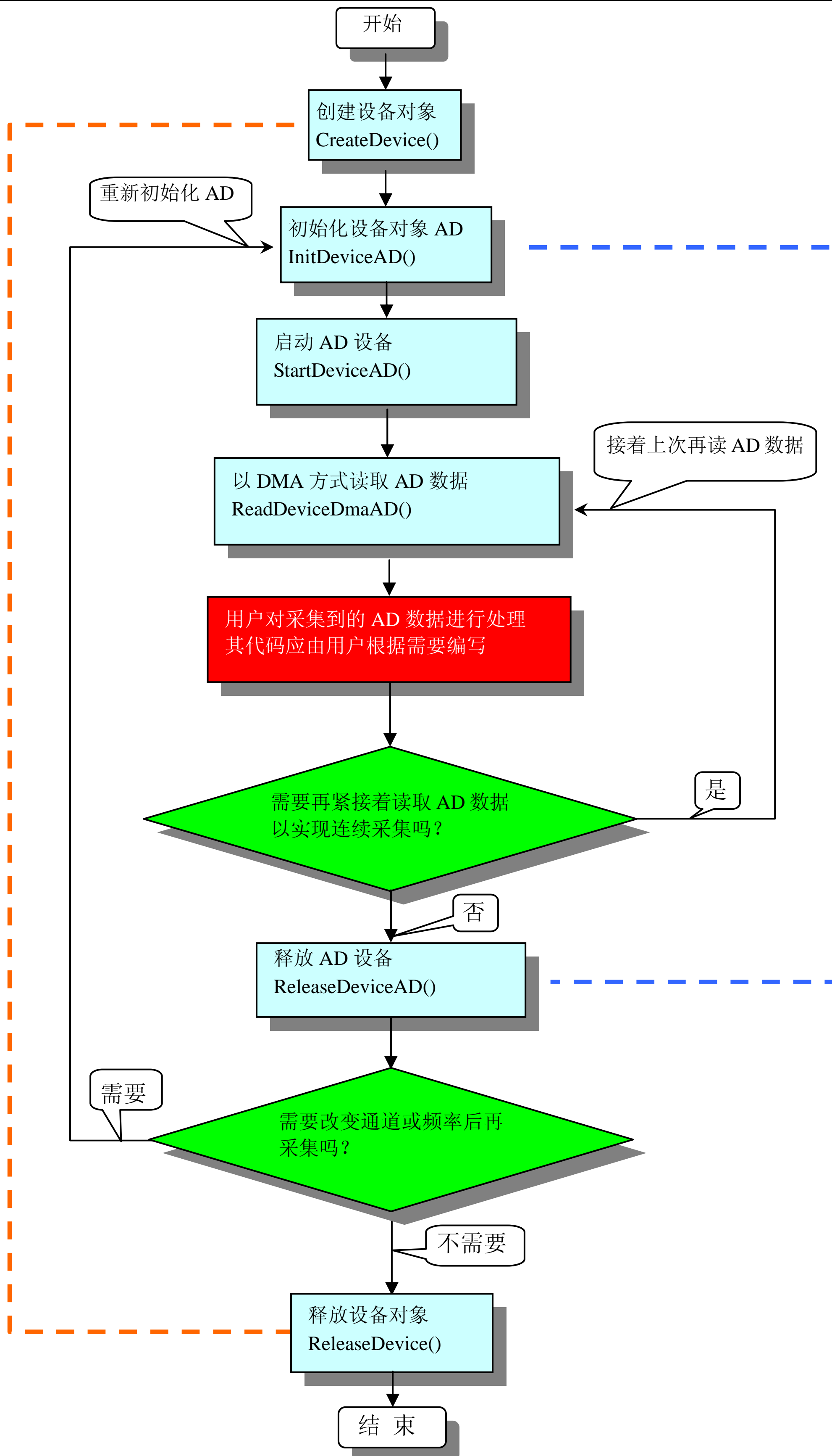



图 2.1.2 DMA 方式 AD 采集过程

## 第五节、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bDOSts[16]中的相应元素决定。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bDISts[16]中的相应元素决定。

### 第六节、哪些函数对您不是必须的

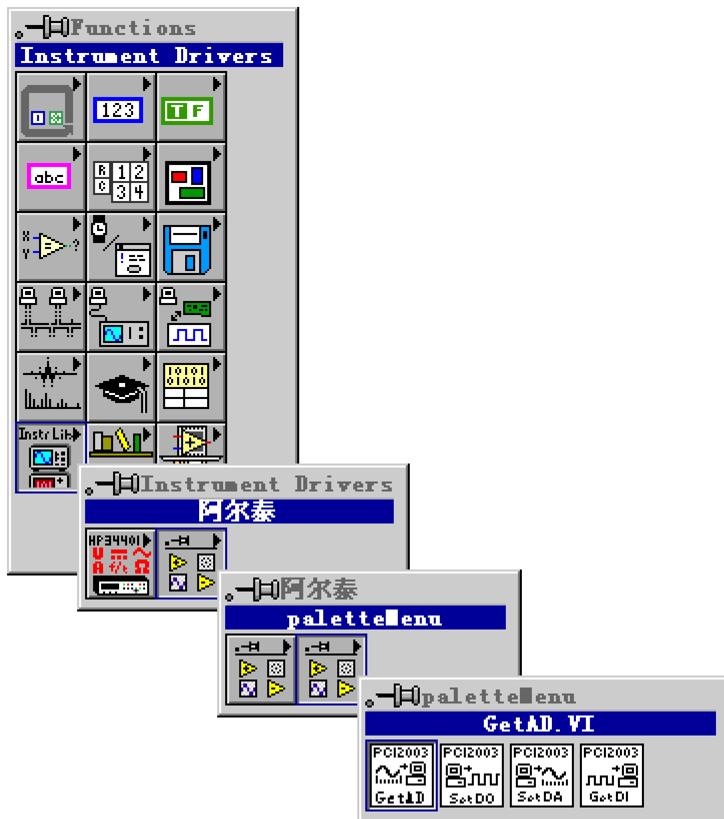
公共函数如 CreateFileObject， WriteFile，ReadFile等一般来说都是辅助性函数，除非您要使用存盘功能。如果您使用上层用户函数访问设备，那么 GetDeviceAddr，WriteRegisterByte， WriteRegisterWord，WriteRegisterULong， ReadRegisterByte， ReadRegisterWord， ReadRegisterULong等函数您可完全不必理会，除非您是作为底层用户管理设备。而 WritePortByte， WritePortWord， WritePortULong， ReadPortByte，ReadPortWord， ReadPortULong则对PCI用户来讲，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，您可能在基于Windows NT架构的操作系统中无法继续使用您原有的老设备。

# 第三章 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AD的首末通道、采样频率等，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AD数据。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，比如在正式采集数据之前，只须用户调用一个简易的初始化函数（如 InitDeviceAD）告诉设备我要使用多少个通道，采样频率是多少赫兹等，然后便可以用 ReadDeviceProAD（或 ReadDeviceDmaAD）函数指定每次采集的点数，即可实现数据连续不间断采样。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 ReadRegisterULong和 WriteRegisterULong对这些端口寄存器进行 32 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将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，先明白自己是上层用户还是底层用户，因为在《设备驱动接口函数总列表》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明的关于 LabView 的接口，均属于外挂式驱动接口，他是通过 LabView 的 Call Labrary 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的。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。关于 LabView 的外挂式驱动和内嵌式驱动更详细的叙述，请参考 LabView 的相关演示。




LabView 内嵌式驱动接口的获取方法

## 第一节、设备驱动接口函数总列表（每个函数省略了前缀“PCI8025_”）

| 函数名 | 函数功能 | 备注 |
|---|---|---|
| ① 设备对象操作函数 | | |
| CreateDevice | 创建 PCI 设备对象(用设备逻辑号) | 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 |
| ListDeviceDlg | 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 |
| ② AD 的程序方式读取函数 | | |
| SetDevFrequencyAD | 可动态改变 AD 采样频率 | 上层用户 |
| SetDevTrigSrcAD | 设置触发源 | 上层用户 |
| GetDevStatusAD | 取得设备的各种状态 | 上层用户 |
| GetDevTrigPosAD | 取得触发点位置 | 上层用户 |
| ClearDevStatusAD | 清除指定标志 | 上层用户 |
| InitDeviceAD | 初始化 AD 部件准备传输 | 上层用户 |
| StartDeviceAD | 启动 AD 设备，开始转换 | 上层用户 |
| ReadDeviceProAD | 程序读取当前 PCI 设备上的 AD 数据 | 上层用户 |
| ReadDeviceDmaAD | DMA 方式读取当前 PCI 设备上的 AD 数据 | 上层用户 |
| StopDeviceAD | 暂停 AD 设备 | 上层用户 |
| ReleaseDeviceAD | 释放设备上的 AD 部件 | 上层用户 |
| ③ AD 硬件参数系统保存、读取函数 | | |
| LoadParaAD | 从 Windows 系统中读入硬件参数 | 上层用户 |
| SaveParaAD | 往 Windows 系统写入设备硬件参数 | 上层用户 |
| ResetParaAD | | 上层用户 |
| ④ 模拟信号频率测量函数 | | |
| InitDevicePulse0 | 初始化(ATR) | 上层用户 |
| GetDevPulse0 | 取得模拟触发源的脉冲宽度 | 上层用户 |
| InitDevicePulse1 | 测定 PULSE 信号输入的频率 | 上层用户 |

| GetDevPulse1 | 取得状态(PULSE) | 上层用户 |
|---|---|---|
| **⑤ DA 模拟量输出操作函数** | | |
| WriteDeviceProDA | 写出 DA 数据 | 上层用户 |
| TransVoltToLsbDA | 将电压值转换成原码值 | 上层用户 |
| **⑥ DIO 开关量简易操作函数** | | |
| GetDeviceDI | 开关输入函数 | 上层用户 |
| SetDeviceDO | 开关输出函数 | 上层用户 |

使用需知：

***Visual C++：***

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\PCI8025\INCLUDE\PCI8025.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 PCI8025.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。然后加入如下语句：

#include “PCI8025.H”

**另外，要在 VB 环境中用子线程以实现高速、连续数据采集与存盘，请务必使用 VB5.0 版本。当然如果您有 VB6.0 的最新版，也可以实现子线程操作。**

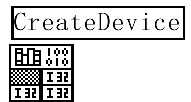
***Visual Basic:***

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，在弹出的对话中选择 PCI8025.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 程序均是需要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。

***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 PCI8025.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接该接口模块即可顺利使用。

二、根据LabView语言本身的规定，接口单元图标以黑色的较粗的中间线为中心，以左边的方格为数据输入端，右边的方格为数据的输出端，如 ReadDeviceProAD接口单元，设备对象句柄、用户分配的数据缓冲区、要求采集的数据长度等信息从接口单元左边输入端进入单元，待单元接口被执行后，需要返回给用户的数据从接口单元右边的输出端输出，其他接口完全同理。

三、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

◆ 创建设备对象函数（逻辑号）

函数原型：

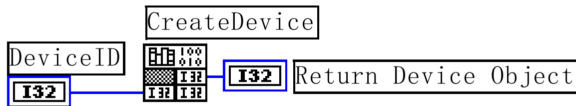
***Visual C++：***

HANDLE CreateDevice (int DeviceID = 0)

***Visual Basic*：**

Declare Function CreateDevice Lib "PCI8025" (ByVal DeviceID As Integer = 0) As Long

***LabVIEW*：**




功能：该函数使用逻辑号创建设备对象，并返回其设备对象句柄 hDevice。只有成功获取 hDevice，您才能实现对该设备所有功能的访问。

参数：

DeviceID 设备 ID 标识号。

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：　CreateDevice　　GetDeviceCount
　　　　　　ListDeviceDlg　　ReleaseDevice

***Visual C++程序举例*：**

```
    :
HANDLE hDevice;     // 定义设备对象句柄
int DeviceLgcID = 0;
hDevice = CreateDevice ( DeviceLgcID ); // 创建设备对象,并取得设备对象句柄
if(hDevice == INVALIDE_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
Dim DeviceLgcID As Long
DeviceLgcID = 0
hDevice = CreateDevice ( DeviceLgcID )  ' 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
        MsgBox "创建设备对象失败"
        Exit Sub     ' 退出该过程
End If
    :
```

◆ **取得本计算机系统中 PCI8025 设备的总数量**

函数原型：

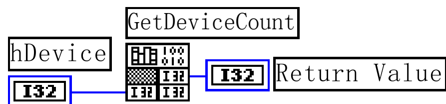
***Visual C++:***

int GetDeviceCount (HANDLE hDevice)

***Visual Basic:***

Declare Function GetDeviceCount Lib "PCI8025" (ByVal hDevice As Long ) As Integer

***LabVIEW*：**




功能：取得 PCI8025 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 PCI8025 的数量。

相关函数：　CreateDevice　　GetDeviceCount
　　　　　　ListDeviceDlg　　ReleaseDevice

◆ **用对话框控件列表计算机系统中所有 PCI8025 设备各种配置信息**

函数原型：

***Visual C++:***

BOOL ListDeviceDlg (HANDLE hDevice)

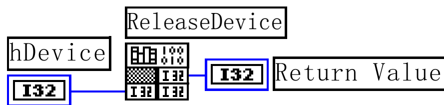
***Visual Basic*:**

Declare Function ListDeviceDlg Lib "PCI8025" (ByVal hDevice As Long ) As Boolean
***LabVIEW:***
请参考相关演示程序。

功能：列表系统中 PCI8025 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，则弹出对话框控件列表所有 PCI8025 设备的配置情况。
相关函数：CreateDevice ReleaseDevice

◆ 释放设备对象所占的系统资源及设备对象

函数原型：
***Visual C++:***
BOOL ReleaseDevice(HANDLE hDevice)
***Visual Basic:***
Declare Function ReleaseDevice Lib "PCI8025" (ByVal hDevice As Long ) As Boolean
***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 GetLastErrorEx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、AD 程序查询方式采样操作函数原型说明

◆ 动态改变采样频率(Set device AD frequency)

函数原型：
***Visual C++:***
BOOL SetDevFrequencyAD (HANDLE hDevice,
LONG nFrequency)
***Visual Basic:***
Declare Function SetDevFrequencyAD Lib "PCI8025" (ByVal hDevice As Long, _
ByVal nFrequency As Long) As Boolean
***LabVIEW:***
请参考相关演示程序。

功能：在AD采样过程中，可动态改变采样频率(在分组采样中只能改变组内频率 Frequency)。
参数：
hDevice设备对象句柄，它应由 CreateDevice创建。
nFrequency 指定 AD 的当前采样频率。本设备的同步采样频率最大为 400KHz，异步采样频率最大为1.6MHz。
返回值：如果调用成功，则返回TRUE，否则返回FALSE，用户可用 GetLastErrorEx捕获当前错误码，并加以分析。
相关函数： CreateDevice SetDevFrequencyAD SetDevTrigSrcAD
GetDevStatusAD GetDevTrigPosAD ClearDevStatusAD
InitDeviceAD StartDeviceAD ReadDeviceProAD
ReadDeviceDmaAD StopDeviceAD ReleaseDeviceAD
ReleaseDevice

◆ 设置触发源

函数原型：

***Visual C++:***

```
BOOL SetDevTrigSrcAD (HANDLE hDevice,
                      LONG TriggerSource)
```

***Visual Basic:***

```
Declare Function SetDevTrigSrcAD Lib "PCI8025" (ByVal hDevice As Long, _
                                        ByVal TriggerSource As Long) As Boolean
```

***LabVIEW：***

请参考相关演示程序。

功能：在 AD 采样过程中，设置触发源。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

TriggerSource 触发源选择。

| 常量名 | 常量值 | 功能定义 |
| --- | --- | --- |
| PCI8025_TRIGSRC_AI0 | 0x00 | 选择 AI0 作为触发源 |
| PCI8025_TRIGSRC_AI1 | 0x01 | 选择 AI1 作为触发源 |
| PCI8025_TRIGSRC_AI2 | 0x02 | 选择 AI2 作为触发源 |
| PCI8025_TRIGSRC_AI3 | 0x03 | 选择 AI3 作为触发源 |
| PCI8025_TRIGSRC_ATR | 0x04 | 选择 ATR 作为触发源 |

返回值：如果调用成功，则返回TRUE，否则返回FALSE，用户可用 GetLastErrorEx捕获当前错误码，并加以分析。

相关函数：
CreateDevice SetDevFrequencyAD SetDevTrigSrcAD
GetDevStatusAD GetDevTrigPosAD ClearDevStatusAD
InitDeviceAD StartDeviceAD ReadDeviceProAD
ReadDeviceDmaAD StopDeviceAD ReleaseDeviceAD
ReleaseDevice

- **在 AD 采样过程中取得设备的各种状态**

函数原型：

***Visual C++:***

```
BOOL GetDevStatusAD (HANDLE hDevice,
                     PPCI8025_STATUS_AD pADStatus)
```

***Visual Basic:***

```
Declare Function GetDevStatusAD Lib "PCI8025" (ByVal hDevice As Long, _
                                        ByVal pADStatus As PCI8025_STATUS_AD) As Boolean
```

***LabVIEW：***

请参考相关演示程序。

功能：在 AD 采样过程中取得设备的各种状态，返回值表示函数是否成功。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

pADStatus 获得 AD 的各种当前状态。它属于结构体，具体定义请参考《AD 状态参数结构（PCI8025_STATUS_AD）》章节。

返回值：如果调用成功，则返回TRUE，否则返回FALSE，用户可用 GetLastErrorEx捕获当前错误码，并加以分析。

相关函数：
CreateDevice SetDevFrequencyAD SetDevTrigSrcAD
GetDevStatusAD GetDevTrigPosAD ClearDevStatusAD
InitDeviceAD StartDeviceAD ReadDeviceProAD
ReadDeviceDmaAD StopDeviceAD ReleaseDeviceAD
ReleaseDevice

- 取得触发点位置(相对于当前采集的所有数据的起点)

函数原型：

***Visual C++:***

```
BOOL GetDevTrigPosAD (HANDLE hDevice,
```

PLONGLONG nTriggerPos)

***Visual Basic:***

Declare Function GetDevTrigPosAD Lib "PCI8025" (ByVal hDevice As Long, _
ByRef nTriggerPos As Currency) As Boolean

***LabVIEW:***

请参考相关演示程序。

功能：取得触发点位置(相对于当前采集的所有数据的起点)。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

nTriggerPos 返回触发点位置（即触发点前的数据总量）。

返回值：如果调用成功，则返回TRUE，否则返回FALSE，用户可用 GetLastErrorEx捕获当前错误码，并加以分析。

相关函数： CreateDevice SetDevFrequencyAD SetDevTrigSrcAD
GetDevStatusAD GetDevTrigPosAD ClearDevStatusAD
InitDeviceAD StartDeviceAD ReadDeviceProAD
ReadDeviceDmaAD StopDeviceAD ReleaseDeviceAD
ReleaseDevice

### ♦ 清除指定标志

函数原型：

***Visual C++:***

```
BOOL ClearDevStatusAD (HANDLE hDevice,
                       BOOL bClrRamSwitch = FALSE,
                       BOOL bClrOverflow = FALSE,
                       BOOL bClrTrigFlag = FALSE)
```

***Visual Basic:***

```
Declare Function ClearDevStatusAD Lib "PCI8025" (ByVal hDevice As Long, _
                       ByVal bClrRamSwitch As Boolean = FALSE,_
                       ByVal bClrOverflow As Boolean = FALSE,_
                       ByVal bClrTrigFlag As Boolean = FALSE) As Boolean
```

***LabVIEW:***

请参考相关演示程序。

功能：清除指定标志。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

bClrRamSwitch 是否清除触发标志，=TRUE 表示清除， =FALSE 表示不清除。

bClrOverflow 是否清除 RAM 重写标志， =TRUE 表示清除，=FALSE 表示不清除。

bClrTrigFlag 是否清除触发标志， =TRUE 表示清除，=FALSE 表示不清除。

返回值：如果调用成功，则返回 TRUE，否则返回 FALSE，用户可用 GetLastErrorEx 捕获当前错误码，并加以分析。

相关函数： CreateDevice SetDevFrequencyAD SetDevTrigSrcAD
GetDevStatusAD GetDevTrigPosAD ClearDevStatusAD
InitDeviceAD StartDeviceAD ReadDeviceProAD
ReadDeviceDmaAD StopDeviceAD ReleaseDeviceAD
ReleaseDevice

### ♦ 启动 AD 设备(Start device AD for program mode)

函数原型：

***Visual C++:***

```
BOOL StartDeviceAD (HANDLE hDevice)
```

***Visual Basic:***

```
Declare Function StartDeviceAD Lib "PCI8025" (ByVal hDevice As Long ) As Boolean
```

***LabVIEW:***

请参考相关演示程序。

功能：启动AD设备，它必须在调用 InitDeviceAD后才能调用此函数。该函数除了启动AD设备开始转换以外，不改变设备的其他任何状态。

参数：hDevice设备对象句柄，它应由 CreateDevice创建。

返回值：如果调用成功，则返回TRUE，且AD立刻开始转换，否则返回FALSE，用户可用 GetLastErrorEx捕获当前错误码，并加以分析。

相关函数：

| CreateDevice | SetDevFrequencyAD | SetDevTrigSrcAD |
| --- | --- | --- |
| GetDevStatusAD | GetDevTrigPosAD | ClearDevStatusAD |
| InitDeviceAD | StartDeviceAD | ReadDeviceProAD |
| ReadDeviceDmaAD | StopDeviceAD | ReleaseDeviceAD |
| ReleaseDevice | | |

♦ **程序方式读取 PCI 设备上的 AD 数据**

函数原型：

***Visual C++:***

```
BOOL ReadDeviceProAD ( HANDLE hDevice,
                       SHORT pADBuffer[],
                       LONG nReadOffsetWords,
                       LONG nReadSizeWords,
                       PLONG nRetSizeWords,
                       int nADChannel)
```

***Visual Basic:***

```
Declare Function ReadDeviceProAD Lib "PCI8025" (ByVal hDevice As Long, _
                                               ByRef ADBuffer As Integer,_
                                               ByVal nReadOffsetWords As Long,_
                                               ByVal nReadSizeWords As Long,_
                                               ByRef nRetSizeWords As Long,_
                                               ByVal nADChannel As Integer) As Boolean
```

***LabVIEW:***

请参考相关演示程序。

功能：当 AD 标志有效时，用此函数读取设备上的 AD 数据(程序方式)。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

ADBuffer接受AD数据的用户缓冲区，它可以是一个用户定义的数组。关于如何将这些AD数据转换成相应的电压值，请参考《数据格式转换与排列规则》。

nReadOffsetWords 相对于该通道物理 RAM 零位置的偏移点(字)。

nReadSizeWords 指定一次 ReadDeviceProAD操作应读取多少字数据到用户缓冲区。注意此参数的值不能大于用户缓冲区ADBuffer[]的最大空间。此参数值只与ADBuffer[]指定的缓冲区大小有效，而与FIFO存储器大小无效。

nRetSizeWords指定在读取AD数据的过程中，是否对溢出标志进行检测。默认值为FALSE。若指定为TRUE，表示对溢出标志进行监控，若发生溢出，则该函数立即返回，其返回值表示在溢出前已成功读取的AD数据点数，但这个返回值必将小于nReadSizeWords参数的值。若指定为FALSE，则表示不对FIFO存储器的溢出标志进行监控，即便溢出已发生，也始终返回由nReadSizeWords参数指定长度的数据，其返回值也必将等于nReadSizeWords参数值，除非用户在这个函数返回前，就提前调用了 ReleaseDeviceAD函数要释放AD设备，那么返回值也可能小于nReadSizeWords参数值。究竟是溢出还是提前释放AD引起的返回值小于nReadSizeWords参数值，用户可以在这种情况下，调用 GetLastErrorEx来判断。

nADChannel AD 通道号(0-3)。

返回值： 其返回值表示所成功读取的数据点数(字)，也表示当前读操作在ADBuffer[]缓冲区中的有效数据量。通常情况下其返回值应与ReadSizeWords参数指定量的数据长度(字)相等，除非用户在这个读操作以外的其他线程中执行了 ReleaseDeviceAD函数中断了读操作，否则设备可能有问题。对于返回值不等于nReadSizeWords参数值的，用户可用 GetLastErrorEx捕获当前错误码，并加以分析。

相关函数：

| CreateDevice | SetDevFrequencyAD | SetDevTrigSrcAD |
| --- | --- | --- |
| GetDevStatusAD | GetDevTrigPosAD | ClearDevStatusAD |
| InitDeviceAD | StartDeviceAD | ReadDeviceProAD |
| ReadDeviceDmaAD | StopDeviceAD | ReleaseDeviceAD |
| ReleaseDevice | | |

♦ **DMA 方式读取 PCI 设备上的 AD 数据**

函数原型：

***Visual C++:***

```
BOOL ReadDeviceDmaAD ( HANDLE hDevice,
                       SHORT pADBuffer[],
                       LONG nReadOffsetWords,
                       LONG nReadSizeWords,
                       PLONG nRetSizeWords,
                       int nADChannel)
```

***Visual Basic:***

```
Declare Function ReadDeviceDmaAD Lib "PCI8025" (ByVal hDevice As Long, _
                                    ByRef ADBuffer As Integer,_
                                    ByVal nReadOffsetWords As Long,_
                                    ByVal nReadSizeWords As Long,_
                                    ByRef nRetSizeWords As Long,_
                                    ByVal nADChannel As Integer) As Boolean
```

***LabVIEW:***

请参考相关演示程序。

功能：当 AD 标志有效时，用此函数读取设备上的 AD 数据(DMA 方式)。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

ADBuffer接受AD数据的用户缓冲区，它可以是一个用户定义的数组。关于如何将这些AD数据转换成相应的电压值，请参考《数据格式转换与排列规则》。

nReadOffsetWords 相对于该通道物理 RAM 零位置的偏移点(字)。

nReadSizeWords 指定一次 ReadDeviceDmaAD操作应读取多少字数据到用户缓冲区。注意此参数的值不能大于用户缓冲区ADBuffer[]的最大空间。此参数值只与ADBuffer[]指定的缓冲区大小有效，而与FIFO存储器大小无效。

nRetSizeWords指定在读取AD数据的过程中，是否对溢出标志进行检测。默认值为FALSE。若指定为TRUE，表示对溢出标志进行监控，若发生溢出，则该函数立即返回，其返回值表示在溢出前已成功读取的AD数据点数，但这个返回值必将小于nReadSizeWords参数的值。若指定为FALSE，则表示不对FIFO存储器的溢出标志进行监控，即便溢出已发生，也始终返回由nReadSizeWords参数指定长度的数据，其返回值也必将等于nReadSizeWords参数值，除非用户在这个函数返回前，就提前调用了 ReleaseDeviceAD函数要释放AD设备，那么返回值也可能小于nReadSizeWords参数值。究竟是溢出还是提前释放AD引起的返回值小于nReadSizeWords参数值，用户可以在这种情况下，调用 GetLastErrorEx来判断。

nADChannel AD 通道号(0-3)。

返回值： 其返回值表示所成功读取的数据点数(字)，也表示当前读操作在ADBuffer[]缓冲区中的有效数据量。通常情况下其返回值应与ReadSizeWords参数指定量的数据长度(字)相等，除非用户在这个读操作以外的其他线程中执行了 ReleaseDeviceAD函数中断了读操作，否则设备可能有问题。对于返回值不等于nReadSizeWords参数值的，用户可用 GetLastErrorEx捕获当前错误码，并加以分析。

相关函数：

| | | |
|---|---|---|
| CreateDevice | SetDevFrequencyAD | SetDevTrigSrcAD |
| GetDevStatusAD | GetDevTrigPosAD | ClearDevStatusAD |
| InitDeviceAD | StartDeviceAD | ReadDeviceProAD |
| ReadDeviceDmaAD | StopDeviceAD | ReleaseDeviceAD |
| ReleaseDevice | | |

♦ **暂停 AD 设备**

函数原型：

***Visual C++:***

```
BOOL StopDeviceAD (HANDLE hDevice)
```

***Visual Basic:***

```
Declare Function StopDeviceAD Lib "PCI8025" (ByVal hDevice As Long )As Boolean
```

***LabVIEW:***

请参考相关演示程序。

功能：暂停AD设备。它必须在调用 StartDeviceAD后才能调用此函数。该函数除了停止AD设备不再转换以外，不改变设备的其他任何状态。此后您可再调用 StartDeviceAD函数重新启动AD，此时AD会按照暂停以前的状态（如FIFO存储器位置、通道位置）开始转换。

参数：hDevice设备对象句柄，它应由 CreateDevice创建。

返回值：如果调用成功，则返回TRUE，且AD立刻停止转换，否则返回FALSE，用户可用 GetLastErrorEx捕获当前错误码，并加以分析。

相关函数：

| CreateDevice | SetDevFrequencyAD | SetDevTrigSrcAD |
| --- | --- | --- |
| GetDevStatusAD | GetDevTrigPosAD | ClearDevStatusAD |
| InitDeviceAD | StartDeviceAD | ReadDeviceProAD |
| ReadDeviceDmaAD | StopDeviceAD | ReleaseDeviceAD |
| ReleaseDevice | | |

- ♦ 释放设备上的 **AD** 部件

函数原型：

***Visual C++:***

BOOL ReleaseDeviceAD(HANDLE hDevice)

***Visual Basic:***

Declare Function ReleaseDeviceAD Lib "PCI8025" (ByVal hDevice As Long ) As Boolean

***LabVIEW:***

请参考相关演示程序。

功能：释放设备上的 AD 部件。

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 GetLastErrorEx捕获错误码。

应注意的是，InitDeviceAD必须和 ReleaseDeviceAD函数一一对应，即当您执行了一次 InitDeviceAD后，再一次执行这些函数前，必须执行一次 ReleaseDeviceAD函数，以释放由 InitDeviceAD占用的系统软硬件资源，如映射寄存器地址、系统内存等。只有这样，当您再次调用 InitDeviceAD函数时，那些软硬件资源才可被再次使用。

相关函数：

| CreateDevice | InitDeviceAD | ReleaseDeviceAD |
| --- | --- | --- |
| ReleaseDevice | | |

- ♦ 程序查询方式采样函数一般调用顺序

非空查询方式：

① CreateDevice
② InitDeviceAD
③ StartDeviceAD
④ ReadDeviceProAD
⑤ StopDeviceAD
⑥ ReleaseDeviceAD
⑦ ReleaseDevice

注明：用户可以反复执行第④步，以实现高速连续不间断大容量采集。

半满查询方式：

① CreateDevice
② InitDeviceAD
③ StartDeviceAD
④ GetDevStatusAD
⑤ ReadDeviceDmaAD
⑥ StopDeviceAD
⑦ ReleaseDeviceAD
⑧ ReleaseDevice

注明：用户可以反复执行第④、⑤步，以实现高速连续不间断大容量采集。

关于两个过程的图形说明请参考《使用纲要》。

## 第四节、AD 硬件参数保存与读取函数原型说明

◆ **从 Windows 系统中读入硬件参数函数**

函数原型：

***Visual C++:***

```
BOOL LoadParaAD(HANDLE hDevice,
                PPCI8025_PARA_AD pADPara)
```

***Visual Basic:***

```
Declare Function LoadParaAD Lib "PCI8025" (ByVal hDevice As Long, _
                        ByRef pADPara As PCI8025_PARA_AD) As Boolean
```

***LabVIEW:***

请参考相关演示程序。

功能：负责从 Windows 系统中读取设备的硬件参数。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

pADPara属于PPCI8025_PARA_AD的结构指针类型，它负责返回PCI硬件参数值，关于结构指针类型PPCI8025_PARA_AD请参考PCI8025.h或PCI8025.Bas或PCI8025.Pas函数原型定义文件，也可参考本文《硬件参数结构》关于该结构的有关说明。

返回值：若成功，返回 TRUE，否则返回 FALSE。

相关函数： CreateDevice LoadParaAD SaveParaAD ReleaseDevice

◆ **往 Windows 系统写入设备硬件参数函数**

函数原型：

***Visual C++:***

```
BOOL SaveParaAD (HANDLE hDevice,
                 PPCI8025_PARA_AD pADPara)
```

***Visual Basic:***

```
Declare Function SaveParaAD Lib "PCI8025" (ByVal hDevice As Long, _
                        ByRef pADPara As PCI8025_PARA_AD) As Boolean
```

***LabVIEW:***

请参考相关演示程序。

功能：负责把用户设置的硬件参数保存在 Windows 系统中，以供下次使用。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

pADPara设备硬件参数，关于PCI8025_PARA_AD的详细介绍请参考PCI8025.h或PCI8025.Bas或PCI8025.Pas函数原型定义文件，也可参考本文《硬件参数结构》关于该结构的有关说明。

返回值：若成功，返回 TRUE，否则返回 FALSE。

相关函数： CreateDevice LoadParaAD SaveParaAD ReleaseDevice

## 第五节、AD 模拟信号频率测量函数原型说明

◆ **初始化(ATR)**

函数原型：

***Visual C++:***

```
BOOL InitDevicePulse0 (HANDLE hDevice,
                       LONG PulseDir)
```

***Visual Basic:***

```
Declare Function InitDevicePulse0 Lib "PCI8025" (ByVal hDevice As Long, _
                        ByVal PulseDir As Long) As Boolean
```

***LabVIEW:***

请参考相关演示程序。

功能：初始化(ATR)。

参数：
hDevice设备对象句柄，它应由 CreateDevice创建。
PulseDir 脉冲计数方向。

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_PULSEDIR_NEGATIVE | 0x00 | 负脉冲开始测量 |
| PCI8025_PULSEDIR_POSITIVE | 0x01 | 正脉冲开始测量 |
| PCI8025_PULSEDIR_POSIT_NEGAT | 0x02 | 负/正脉冲开始测量 |

返回值：若成功，返回 TRUE，否则返回 FALSE。
相关函数： CreateDevice InitDevicePulse0 GetDevPulse0 InitDevicePulse1 GetDevPulse1 ReleaseDevice

◆ 取得模拟触发源的脉冲宽度

函数原型：
***Viusal C++:***

```
BOOL GetDevPulse0 (HANDLE hDevice,
                   PLONG nStep,
                   PBOOL bOverflowT0,
                   double* Time0,
                   PBOOL bOverflowT1,
                   double* Time1)
```

***Visual Basic:***

```
Declare Function GetDevPulse0 Lib "PCI8025" (ByVal hDevice As Long, _
                                    ByRef nStep As Long,_
                                    ByRef bOverflowT0 As Boolean,_
                                    ByRef Time0 As double,_
                                    ByRef bOverflowT1 As Boolean,_
                                    ByRef Time1 As double) As Boolean
```

***LabVIEW:***
请参考相关演示程序。

功能：取得模拟触发源的脉冲宽度。
参数：
hDevice设备对象句柄，它应由 CreateDevice创建。
nStep 返回当前测量步骤，0=未开始，1=测量 T0，2=测量 T1， 3=测量结束。
bOverflowT0 T0 溢出位，FALSE(0)=未溢出，TRUE(1)=溢出。
Time0 返回当前 T0 的时间宽度(单位：微秒)。
bOverflowT1 T1 溢出位，FALSE(0)=未溢出，TRUE(1)=溢出。
Time1 返回当前 T1 的时间宽度(单位：微秒)。
返回值：若成功，返回 TRUE，否则返回 FALSE。
相关函数： CreateDevice InitDevicePulse0 GetDevicePulse0 InitDevicePulse1 GetDevicePulse1 ReleaseDevice

◆ 测定 PULSE 信号输入的频率

函数原型：
***Viusal C++:***

```
BOOL InitDevicePulse1 (HANDLE hDevice,
                       LONG PulseDir,
                       double TimeUs)
```

***Visual Basic:***

```
Declare Function InitDevicePulse1 Lib "PCI8025" (ByVal hDevice As Long, _
                                    ByVal PulseDir As Long,_
                                    ByVal TimeUs As double) As Boolean
```

***LabVIEW:***
请参考相关演示程序。

功能：测定 PULSE 信号输入的频率。
参数：
hDevice设备对象句柄，它应由 CreateDevice创建。
PulseDir 脉冲计数方向。
TimeUs 定时计数的时间(微秒)，取值范围为[1, 53687091.1875]。
返回值：若成功，返回 TRUE，否则返回 FALSE。
相关函数： CreateDevice InitDevicePulse0 GetDevicePulse0
InitDevicePulse1 GetDevicePulse1 ReleaseDevice

◆ **取得状态(PULSE)**
函数原型：
***Visual C++:***
```
BOOL GetDevPulse1 (HANDLE hDevice,
                   PBOOL bComplete,
                   double* CurTimeUs,
                   ULONG* CNTVal)
```
***Visual Basic:***
```
Declare Function GetDevPulse1 Lib "PCI8025" (ByVal hDevice As Long, _
                                             ByRef bComplete As Boolean,_
                                             ByRef CurTimeUs As double,_
                                             ByRef CNTVal As Long) As Boolean
```
***LabVIEW:***
请参考相关演示程序。

功能：取得状态(PULSE)。
参数：
hDevice设备对象句柄，它应由 CreateDevice创建。
bComplete 返回完成标志，0=未完成，1=完成。
CurTimeUs 当前完成的时间(单位：微秒)，(只有当 bComplete=TRUE，才是最终换算频率的时间值)。
CNTVal 当前已被统计的脉冲个数（只有当 bComplete=TRUE，才是最终换算频率的计数值)。
返回值：若成功，返回 TRUE，否则返回 FALSE。
相关函数： CreateDevice InitDevicePulse0 GetDevicePulse0
InitDevicePulse1 GetDevicePulse1 ReleaseDevice

## 第六节、DA 模拟量输出操作函数原型说明

◆ **写出 DA 数据**
函数原型：
***Visual C++:***
```
BOOL WriteDeviceProDA (HANDLE hDevice,
                       SHORT nDALsb)
```
***Visual Basic:***
```
Declare Function WriteDeviceProDA Lib "PCI8025" ( ByVal hDevice As Long, _
                                                  ByVal nDALsb As Integer) As Boolean
```
***LabVIEW:***
请参考相关演示程序。

功能：写出 DA 数据。
参数：
hDevice设备对象句柄，它应由 CreateDevice创建。
nDALsb 输出的 DA 原码数据。
返回值：若成功，返回TRUE，写出DA数据；否则返回FALSE，您可以调用 GetLastErrorEx函数取得错误或错误字符信息。
相关函数： CreateDevice WriteDeviceProDA TransVoltToLsbDA
ReleaseDevice

◆ 将电压值转换成原码值(不操作实际硬件)

函数原型：

***Visual C++:***

```
SHORT TransVoltToLsbDA (float fDAVolt,
                        LONG OutputRange = 0)
```

***Visual Basic:***

```
Declare Function TransVoltToLsbDA Lib "PCI8025" (ByVal fDAVolt As Single, _
                                                 ByVal OutputRange As Long = 0) As Integer
```

***LabVIEW:***

请参考相关演示程序。

功能：将电压值转换成原码值(不操作实际硬件)。

参数：

fDAVolt 待转换的电压值(mV)。

OutputRange 电压值所对应的输出量程范围(本设备只有默认选择 0)。

返回值：若成功，返回TRUE；否则返回FALSE，您可以调用 GetLastErrorEx函数取得错误或错误字符信息。

相关函数： CreateDevice　　WriteDeviceProDA　　TransVoltToLsbDA　　ReleaseDevice

## 第七节、DIO 数字量输入输出开关量操作函数原型说明

◆ 开关量输入

函数原型：

***Visual C++:***

```
BOOL GetDeviceDI ( HANDLE hDevice,
                   PPCI8025_PARA_DI pDIPara)
```

***Visual Basic:***

```
Declare Function GetDeviceDI Lib "PCI8025" ( ByVal hDevice As Long, _
                                             ByVal pDIPara As PCI8025_PARA_DI) As Boolean
```

***LabVIEW:***

请参考相关演示程序。

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创建。

pDIPara 十六路开关量输入状态的参数结构，共有 16 个元素，分别对应于 DI0～DI15 路开关量输入状态位。如果 bDISts[0]等于“1”则表示 0 通道处于开状态，若为“0”则 0 通道为关状态。其他同理。

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

相关函数： CreateDevice　　SetDeviceDO　　ReleaseDevice

◆ 开关量输出

函数原型：

***Visual C++:***

```
BOOL SetDeviceDO (HANDLE hDevice,
                  PPCI8025_PARA_DO pDOPara)
```

***Visual Basic:***

```
Declare Function SetDeviceDO Lib "PCI8025" (ByVal hDevice As Long, _
                                            ByVal pDOPara As PCI8025_PARA_DO) As Boolean
```

***LabVIEW:***

请参考相关演示程序。

功能：负责将 PCI 设备上的输出开关量置成相应的状态。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

pDOPara 十六路开关量输出状态的参数结构，共有 16 个成员变量，分别对应于 DO0～DO15 路开关量输出状态位。比如置 pPara->DO0 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，必须对这个参数结构的 DO0 至 DO15 共 16 个成员变量赋初值，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

◆ 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）。

# 第四章 硬件参数结构

## 第一节、AD 硬件参数结构（PCI8025_PARA_AD）

***Visual C++：***

```
typedef struct _PCI8025_PARA_AD
{
    LONG InputRange[4];       // 模拟输入量程选择(分别控制四个通道),其取值请见下面选项值定义
    LONG SwitchLenCtrl;       // 存储器切换长度控制,其取值请见下面选项值定义
    LONG ADMode;              // AD 模式选择(同步/异步方式),其取值请见下面选项值定义
    LONG Frequency;           // 采集频率,单位为 Hz,(大于 0.01Hz)，为正数时，以 1Hz 为单位，反之以
0.001Hz 为单位
    LONG TriggerMode;         // 触发模式(软件内触发和硬件外触发),其取值请见下面选项值定义
    LONG TriggerSource;       // 触发源信号选择(AI0, AI1, AI2, AI3, ATR),其取值请见下面选项值定义
    LONG TriggerType;         // 触发类型选择(边沿触发/脉冲触发),其取值请见下面选项值定义
    LONG TriggerDir;          // 触发方向选择(正向/负向触发),其取值请见下面选项值定义
    LONG bSetTrigLevel;       // 是否置触发电平, =TRUE:表示在调用
    LONG TrigLevelVolt;       // 触发电平(-10000mV -- +10000mV)
    LONG ClockSource;         // 时钟源选择(内/外时钟源),其取值请见下面选项值定义
    LONG bClockOutput;        // 允许时钟输出
} PCI8025_PARA_AD, *PPCI8025_PARA_AD;
```

***Visual Basic：***

```
Private Type PCI8025_PARA_AD
    InputRange (4) As Long
    SwitchLenCtrl As Long
    ADMode As Long
    Frequency As Long
    TriggerMode As Long
    TriggerSource As Long
    TriggerType As Long
    TriggerDir As Long
    bSetTrigLevel As Long
    TrigLevelVolt As Long
    ClockSource As Long
    bClockOutput As Long
End Type
```

***LabVIEW:***

请参考相关演示程序。

该结构实在太简易了，其原因就是 PCI 设备是系统全自动管理的设备，再加上驱动程序的合理设计与封装，什么端口地址、中断号等将与 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。

此结构主要用于设定设备AD硬件参数值，用这个参数结构对设备进行硬件配置完全由 InitDeviceAD函数自动完成。用户只需要对这个结构体中的各成员简单赋值即可。

***硬件参数说明***:

InputRange 模拟输入量程选择(分别控制四个通道)。

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_INPUT_N10000_P10000 | 0x0000 | ±10000mV |
| PCI8025_INPUT_N5000_P5000 | 0x0001 | ±5000mV |
| PCI8025_INPUT_N2500_P2500 | 0x02 | ±2500mV |
| PCI8025_INPUT_N1250_P1250 | 0x03 | ±1250mV |

SwitchLenCtrl 存储器切换长度控制。

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_SWITCH_LEN_64K | 0x00 | 64K |
| PCI8025_SWITCH_LEN_32K | 0x01 | 32K |
| PCI8025_SWITCH_LEN_16K | 0x02 | 16K |
| PCI8025_SWITCH_LEN_8K | 0x03 | 8K |
| PCI8025_SWITCH_LEN_4K | 0x04 | 4K |
| PCI8025_SWITCH_LEN_2K | 0x05 | 2K |
| PCI8025_SWITCH_LEN_1K | 0x06 | 1K |

ADMode AD 采样模式。它的取值如下表：

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_ADMODE_SYNC | 0x00 | 同步采样 |
| PCI8025_ADMODE_ASYN | 0x01 | 异步采样 |

Frequency AD 采样频率，单位 Hz(大于 0.01Hz)，为正数时，以 1Hz 为单位，反之以 0.001Hz 为单位。

TriggerMode AD 触发模式。

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_TRIGMODE_SOFT | 0x00 | 软件触发(属于内触发) |
| PCI8025_TRIGMODE_POST | 0x01 | 硬件后触发(属于外触发) |

TriggerSource AD 转换触发源，其选项如下：

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_TRIGSRC_AI0 | 0x00 | 选择 AI0 作为触发源 |
| PCI8025_TRIGSRC_AI1 | 0x01 | 选择 AI1 作为触发源 |
| PCI8025_TRIGSRC_AI2 | 0x02 | 选择 AI2 作为触发源 |
| PCI8025_TRIGSRC_AI3 | 0x03 | 选择 AI3 作为触发源 |
| PCI8025_TRIGSRC_ATR | 0x04 | 选择 ATR 作为触发源 |

TriggerType AD 触发类型。

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_TRIGTYPE_EDGE | 0x00 | 边沿触发 |
| PCI8025_TRIGTYPE_PULSE | 0x01 | 脉冲触发(电平) |

TriggerDir AD 触发方向。它的选项值如下表：

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_TRIGDIR_NEGATIVE | 0x00 | 负向触发(低脉冲/下降沿触发) |
| PCI8025_TRIGDIR_POSITIVE | 0x01 | 正向触发(高脉冲/上升沿触发) |
| PCI8025_TRIGDIR_POSIT_NEGAT | 0x02 | 正负方向均有效 |

注明：PCI8025_TRIGDIR_POSIT_NEGAT 在边沿类型下，则都表示不管是上边沿还是下边沿均触发。而在电平类型下，无论正电平还是负电平均触发。

bSetTrigLevel 是否置触发电平， =TRUE：表示在调用。

TrigLevelVolt 触发电平(-10000mV -- +10000mV)。

ClockSource　AD 触发时钟源选择。它的选项值如下表：

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_CLOCKSRC_IN | 0x00 | 内部时钟定时触发 |
| PCI8025_CLOCKSRC_OUT | 0x01 | 外部时钟定时触发 |

当选择内时钟时，其AD定时触发时钟为板上时钟振荡器经分频得到。它的大小由 Frequency参数决定。

**当选择外时钟时：**

当选择连续采集时(即 ADMode = PCI8025_ADMODE_SEQUENCE)，其AD定时触发时钟为外界时钟输入CLKIN得到，而 Frequency参数则自动失效。

但是当选择分组采集时(即 ADMode = PCI8025_ADMODE_GROUP)，外时钟则是每一组的触发时钟信号，而组内的触发频率则由 Frequency参数决定，由此可见，此时外时钟触发周期必须大于每组总周期，否则紧跟其后的某一外时钟可能会被失效。

bClockOutput　AD 内部时钟输出使能控制。

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_CLOCKOUT_DISABLE | 0x0000 | 禁止本卡上的自带时钟向外输出 |
| PCI8025_CLOCKOUT_ENABLE | 0x0001 | 允许本卡上的自带时钟向外输出 |

**相关函数：**　CreateDevice　　LoadParaAD　　SaveParaAD
ReleaseDevice

InitDevicePulse　PulseDir 脉冲方向。它的选项值如下表：

| 常量名 | 常量值 | 功能定义 |
|---|---|---|
| PCI8025_PULSEDIR_NEGATIVE | 0x00 | 负脉冲开始测量 |
| PCI8025_PULSEDIR_POSITIVE | 0x01 | 正脉冲开始测量 |
| PCI8025_PULSEDIR_POSIT_NEGAT | 0x02 | 负/正脉冲开始测量 |

## 第二节、AD 状态参数结构（PCI8025_STATUS_AD）

***Visual C++：***

```
typedef struct _PCI8025_STATUS_AD
{
    LONG bRamSwitch;          // 板载 RAM 是否切换，=TRUE 表示切换，  =FALSE 表示未切换
    LONG bOverflow;           // 板载 RAM 是否发生重写，=TRUE 表示已重写，  =FALSE 表示未重写
    LONG bTriggerFlag;        // 触发标志是否有效，=TRUE 表示触点标有效，  = FALSE 表示无效（即触发点未到）
    LONGLONG nTriggerPos;     // 触发点位置
    LONG bConverting;         // AD 是否正在转换，  =TRUE:表示正在转换，  =FALS 表示转换完成
    LONG nCurRamNum;          // 当前可读取的 RAM 编号，取值为[0, 1]
} PCI8025_STATUS_AD, *PPCI8025_STATUS_AD;
```

***Visual Basic：***

```
Private Type  PCI8025_STATUS_AD
    bRamSwitch As Long
    bOverflow As Long
    bTriggerFlag As Long
    nTriggerPos As Currency
    bConverting As Long
    nCurRamNum As Long
End Type
```

***LabVIEW:***

请参考相关演示程序。

此结构体主要用于查询AD的各种状态，GetDevStatusAD函数使用此结构体来实时取得AD状态，以便同步各种数据采集和处理过程。

bRamSwitch AD 板载 RAM 是否切换，=TRUE 表示切换，  =FALSE 表示未切换。
bOverflow AD 板载 RAM 是否发生重写，=TRUE 表示已重写，  =FALSE 表示未重写。
bTriggerFlag AD 触发标志是否有效，=TRUE 表示触点标有效，  = FALSE 表示无效（即触发点未到）。
nTriggerPos AD 触发点位置。
bConverting AD 是否正在转换，  =TRUE：表示正在转换，  =FALS 表示转换完成。
nCurRamNum 当前可读取的 RAM 编号，取值为[0, 1]。
相关函数： CreateDevice  GetDevStatusAD  ClearDevStatusAD
ReleaseDevice

## 第三节、数字量输入参数（PCI8025_PARA_DI）

***Visual C++:***

```
typedef struct _PCI8025_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
} PCI8025_PARA_DI,*PPCI8025_PARA_DI;
```

***Visual Basic：***

```
Type PCI8025_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
  PPCI8025_PARA_DI = ^PCI8025_PARA_DI;   // 指针类型结构
  PCI8025_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。

相关函数： CreateDevice GetDeviceDI ReleaseDevice

## 第四节、数字量输出参数（PCI8025_PARA_DO）

***Visual C++:***

```
typedef struct _PCI8025_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
} PCI8025_PARA_DO,*PPCI8025_PARA_DO;
```

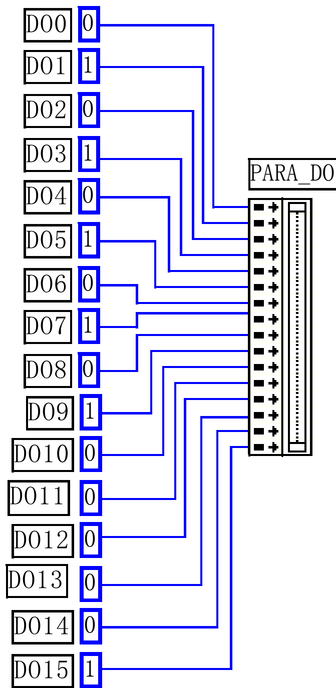
***Visual Basic：***

```
Type PCI8025_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
  PPCI8025_PARA_DO = ^PCI8025_PARA_DO;   // 指针类型结构
  PCI8025_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PPCI8025_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PPCI8025_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。

相关函数： CreateDevice SetDeviceDO ReleaseDevice

# 第五章 数据格式转换与排列规则

## 第一节、AD 原码 LSB 数据转换成电压值的换算方法

首先应根据设备实际位数屏蔽掉不用的高位，然后依其所选量程，按照下表公式进行换算即可。这里只以缓冲区 ADBuffer[]中的第 1 个点 ADBuffer[0]为例。

| 量程(mV) | 计算机语言换算公式(ANSI C 语法) | Volt 取值范围（mV） |
|---|---|---|
| ±10000mV | Volt = (20000.00/16384) * ((ADBuffer[0]^0x2000)&0x3FFF) – 10000.00 | [-10000, +9998.77] |
| ±5000mV | Volt = (10000.00/16384) * ((ADBuffer[0]^0x2000)&0x3FFF) – 5000.00 | [-5000, +4999.38] |
| ±2500mV | Volt = (5000.00/16384) * ((ADBuffer[0]^0x2000)&0x3FFF) – 2500.00 | [-2500, +2499.69] |
| ±1250mV | Volt = (2500.00/16384) * ((ADBuffer[0]^0x2000)&0x3FFF) – 1250.00 | [-1250, +1249.84] |

下面举例说明各种语言的换算过程（以±10000mV 量程为例）

***Visual C++：***

```
Lsb = (ADBuffer[0]^0x2000)&0x3FFF;
Volt = (20000.00/16384) * Lsb -10000.00;
```

***Visual Basic：***

```
Lsb = (ADBuffer [0] XOR &H2000) And &H3FFF
Volt = (20000.00/16384) * Lsb – 10000.00
```

***LabVIEW:***

请参考相关演示程序。

## 第二节、AD 采集函数的 ADBuffer 缓冲区中的数据排放规则

单通道采集，当通道总数首末通道相等时，假如此时首末通道=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：

| 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：

| 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，那么需要用户特别注意的是应处理好各通道数据排列和对齐的问题，尤其是在任意通道数采集时。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。那怎样正确处理呢？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断的多通道数据链。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，仅对于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，而且还高效呢？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、AD 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 HeadSizeBytes 字节位置宽度属于文件头信息，而从 HeadSizeBytes 开始才是真正的 AD 数据。HeadSizeBytes 的取值通常等于本头信息的字节数大小。文件头信息包含的内容如下结构体所示。对于更详细的内容请参考 Visual C++高级演示工程中的 UserDef.h 文件。

```
typedef struct _FILE_HEADER
{
	LONG HeadSizeBytes;		// 文件头信息长度
	LONG FileType;
	// 该设备数据文件共有的成员
	LONG BusType;			// 设备总线类型(DEFAULT_BUS_TYPE)
	LONG DeviceNum;			// 该设备的编号(DEFAULT_DEVICE_NUM)
	LONG HeadVersion;		// 头信息版本(D31-D16=Major   D15-D0=Minijor) = 1.0
	LONG VoltBottomRange;		// 量程下限(mV)
	LONG VoltTopRange;		// 量程上限(mV)
	PCI8025_PARA_AD ADPara;		// 保存硬件参数

	LONG ChannelNum;		// 通道号
	LONGLONG nTriggerPos;		// 触发点位置
	LONG BatCode;			// 同批文件识别码
	LONG HeadEndFlag;		// 文件结束位
} 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。

## 第四节、DA 电压值转换成 LSB 原码数据的换算方法

| 量程(伏) | 计算机语言换算公式(标准 C 语法) | Lsb 取值范围 |
|---|---|---|
| ±10000mV | Lsb = Volt / (20000.00 / 4096) + 2048 | [0，4095] |

# 第六章 上层用户函数接口应用实例

## 第一节、怎样使用 ReadDeviceProAD函数直接取得AD数据

***Visual C++:***

其详细应用实例及正确代码请参考 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]╯**[PCI8025 4 路 AD、DA 和开关量卡]**╯**[Microsoft Visual C++]**╯**[简易代码演示]**╯**[AD 程序查询方式]**

## 第二节、怎样使用 ReadDeviceDmaAD函数直接取得AD数据

***Visual C++:***

其详细应用实例及正确代码请参考 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]╯**[PCI8025 4 路 AD、DA 和开关量卡]**╯**[Microsoft Visual C++]**╯**[简易代码演示]**╯**[AD DMA 直接内存方式]**

## 第三节、怎样使用 WriteDeviceProDA函数取得DA数据

***Visual C++:***

其详细应用实例及正确代码请参考 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]╯**[PCI8025 4 路 AD、DA 和开关量卡]**╯**[Microsoft Visual C++]**╯**[简易代码演示]**╯**[DA 方式]**

### 第四节、怎样使用GetDeviceDI函数进行更便捷的数字开关量输入操作

***Visual C++:***

其详细应用实例及正确代码请参考 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]╜**[PCI8025 4 路 AD、DA 和开关量卡]**╜**[Microsoft Visual C++]**╜**[简易代码演示]**╜**[DIO…]**

### 第五节、怎样使用SetDeviceDO函数进行更便捷的数字开关量输出操作

***Visual C++:***

其详细应用实例及正确代码请参考 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]╜**[PCI8025 4 路 AD、DA 和开关量卡]**╜**[Microsoft Visual C++]**╜**[简易代码演示]**╜**[DIO…]**

## 第七章 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 ReadDeviceProAD(或者 ReadDeviceDmaAD)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，因此可以保证其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，此时它基本不消耗 CPU 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两维数组如ADBuffer [SegmentCount][SegmentSize]，我们将SegmentSize视为数据采集线程每次采集的数据长度，SegmentCount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SegmentCount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SegmentSize长度的数据缓冲区。需要注意的是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 InitDeviceAD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SegmentCount加 1，（注意SegmentCount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SegmentCount加 1，直到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已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SegmentCount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SegmentCount加以判断，观察其值是否大于了 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。

图 7.1 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满后，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，便醒来开始处理数据 ADBuffer[0]缓冲。它们就这样始终差一个节拍。如虚线箭头所示。

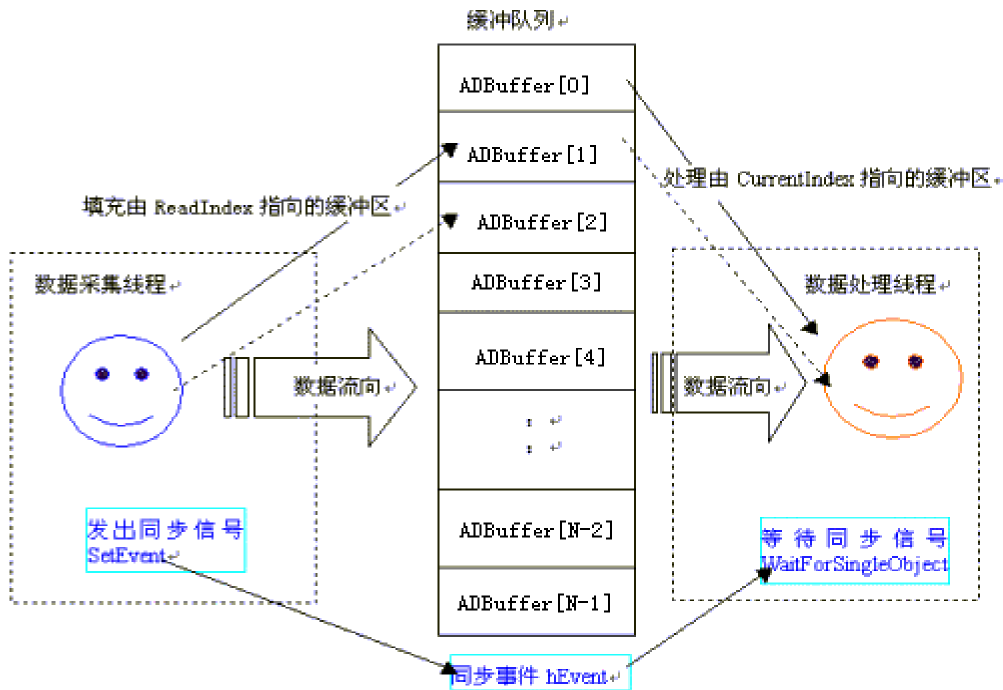



图 7.1

## 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。**

**使用 ReadDeviceProAD函数读取设备上的AD数据**

其详细应用实例及正确代码请参考 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(ADDoc.h 和 ADDoc.cpp， ADThread.h 和 ADThread.cpp)。

[程序] 丨[阿尔泰测控演示系统]丨 **[PCI8025 4 路 AD、DA 和开关量卡]**丨**[Microsoft Visual C++]**丨[高级代码演示]丨 [演示源程序]

然后，您着重参考 ADDoc.cpp 源文件中以下函数：

void CADDoc::StartDeviceAD() // 启动线程函数
BOOL MyStartDeviceAD(HANDLE hDevice); // 位于 ADThread.cpp
UINT ReadDataThread (PVOID pThreadPara) // 读数据线程，位于 ADThread.cpp
UINT ProcessDataThread(PVOID pThreadPara) // 绘制数据线程
BOOL MyStopDeviceAD(HANDLE hDevice); // 位于 ADThread.cpp
void CADDoc::StopDeviceAD() // 终止采集函数

当然用 FIFO 标志读取 AD 数据，能获得接近 FIFO 总容量的栈深度，这样用户在两批数据之间，便有更多的时间来处理某些数据。

## 第二节、使用 DMA 方式实现该功能

其详细应用实例及正确代码请参考 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(ADDoc.cpp 和 ADThread.cpp)。

[程序]╯[阿尔泰测控演示系统]╯**[PCI8025 4 路 AD、DA 和开关量卡]**╯**[Microsoft Visual C++]**╯[高级代码演示]╯[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 () // 启动采集线程函数
UINT ReadDataThread () // 采集线程函数
BOOL StopDeviceAD () // 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

# 第八章 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、公用接口函数总列表（每个函数省略了前缀“PCI8025_”）

| 函数名 | 函数功能 | 备注 |
|---|---|---|
| ① **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 |
| WriteRegisterByte | 以字节(8Bit)方式写寄存器端口 | 底层用户 |
| WriteRegisterWord | 以字(16Bit)方式写寄存器端口 | 底层用户 |
| WriteRegisterULong | 以双字(32Bit)方式写寄存器端口 | 底层用户 |
| ReadRegisterByte | 以字节(8Bit)方式读寄存器端口 | 底层用户 |
| ReadRegisterWord | 以字(16Bit)方式读寄存器端口 | 底层用户 |
| ReadRegisterULong | 以双字(32Bit)方式读寄存器端口 | 底层用户 |
| ② **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 |
| ③线程操作函数 | | |
| CreateSystemEvent | 创建系统内核事件对象 | 用于线程同步或中断 |
| ReleaseSystemEvent | 释放系统内核事件对象 | |

## 第二节、PCI 内存映射寄存器操作函数原型说明

♦ **以单字节（即 8 位）方式写 PCI 内存映射寄存器的某个单元**

函数原型：

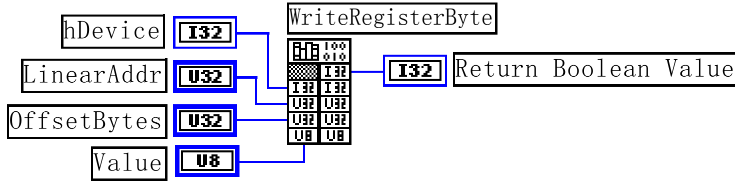
***Visual C++:***

```
BOOL WriteRegisterByte( HANDLE hDevice,
                        PUCHAR pbLinearAddr,
                        ULONG OffsetBytes,
                        BYTE Value)
```

***Visual Basic:***

```
Declare Function WriteRegisterByte Lib "PCI8025" (ByVal hDevice As Long, _
                                                  ByVal pbLinearAddr As Long, _
                                                  ByVal OffsetBytes As Long, _
                                                  ByVal Value As Byte ) As Boolean
```

***LabVIEW:***




功能：以单字节（即 8 位）方式写 PCI 内存映射寄存器。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

LinearAddr PCI设备内存映射寄存器的线性基地址，它的值应由 GetDeviceAddr确定。

OffsetBytes 相对于 LinearAddr 线性基地址的偏移字节数，它与 LinearAddr 两个参数共同确定 WriteRegisterByte函数所访问的映射寄存器的内存单元。

Value 输出 8 位整数。

返回值：若成功，返回 TRUE，否则返回 FALSE。

相关函数： CreateDevice WriteRegisterByte WriteRegisterWord WriteRegisterULong ReadRegisterByte ReleaseDevice ReadRegisterWord ReadRegisterULong

***Visual C++程序举例:***

```
    :
HANDLE hDevice;
ULONG LinearAddr, PhysAddr, OffsetBytes;
hDevice = CreateDevice(0)
if (!GetDeviceAddr(hDevice, &LinearAddr,    &PhysAddr, 0) )
{
        AfxMessageBox "取得设备地址失败…";
}
OffsetBytes = 100;      // 指定操作相对于线性基地址偏移 100 个字节数位置的单元
WriteRegisterByte(hDevice, LinearAddr, OffsetBytes, 0x20);    // 往指定映射寄存器单元写入 8 位的十六进制数据 20
ReleaseDevice( hDevice );    // 释放设备对象
    :
```

***Visual Basic 程序举例:***

```
    :
Dim hDevice As Long
Dim LinearAddr, PhysAddr, OffsetBytes As Long
hDevice = CreateDevice(0)
GetDeviceAddr( hDevice, LinearAddr, PhysAddr, 0)
OffsetBytes = 100
WriteRegisterByte( hDevice, LinearAddr, OffsetBytes, &H20)
ReleaseDevice(hDevice)
    :
```

♦ **以双字节（即 16 位）方式写 PCI 内存映射寄存器的某个单元**

函数原型：

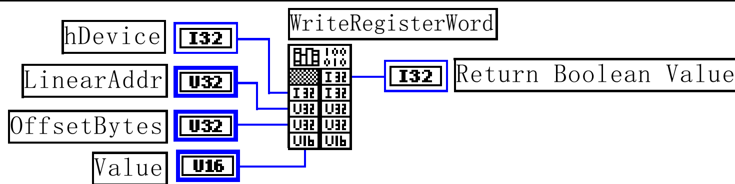
***Visual C++:***

```
BOOL WriteRegisterWord( HANDLE hDevice,
                        PUCHAR pbLinearAddr,
                        ULONG OffsetBytes,
                        WORD Value)
```

***Visual Basic:***

```
Declare Function WriteRegisterWord Lib "PCI8025" (ByVal hDevice As Long, _
                                                 ByVal pbLinearAddr As Long, _
                                                 ByVal OffsetBytes As Long, _
                                                 ByVal Value As Integer) As Boolean
                        StdCall; External 'PCI8025' Name ' WriteRegisterWord ';
```

***LabVIEW:***




功能：以双字节（即 16 位）方式写 PCI 内存映射寄存器。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

LinearAddr　PCI设备内存映射寄存器的线性基地址，它的值应由 GetDeviceAddr确定。

OffsetBytes　相对于LinearAddr线性基地址的偏移字节数，它与LinearAddr两个参数共同确定WriteRegisterWord函数所访问的映射寄存器的内存单元。

Value　输出 16 位整型值。

返回值：无。

相关函数：　CreateDevice　WriteRegisterByte　WriteRegisterWord　WriteRegisterULong　ReadRegisterByte　ReleaseDevice　ReadRegisterWord　ReadRegisterULong

***Visual C++程序举例:***

```
    :
HANDLE hDevice;
ULONG LinearAddr, PhysAddr, OffsetBytes;
hDevice = CreateDevice(0)
if (!GetDeviceAddr(hDevice, &LinearAddr,  &PhysAddr, 0) )
{
    AfxMessageBox "取得设备地址失败…";
}
OffsetBytes = 100;    // 指定操作相对于线性基地址偏移 100 个字节数位置的单元
WriteRegisterWord(hDevice, LinearAddr, OffsetBytes, 0x2000);  // 往指定映射寄存器单元写入 16 位的十六进制数据
ReleaseDevice( hDevice );  // 释放设备对象
    :
```

***Visual Basic 程序举例:***

```
    :
Dim hDevice As Long
Dim LinearAddr, PhysAddr, OffsetBytes As Long
hDevice = CreateDevice(0)
GetDeviceAddr( hDevice, LinearAddr, PhysAddr, 0)
OffsetBytes=100
WriteRegisterWord( hDevice, LinearAddr, OffsetBytes, &H2000)
ReleaseDevice(hDevice)
    :
```

- **以四字节（即 32 位）方式写 PCI 内存映射寄存器的某个单元**

函数原型：

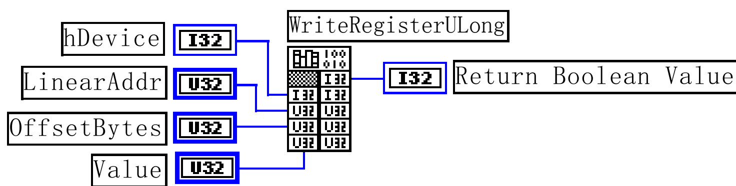
***Visual C++:***

```
BOOL WriteRegisterULong( HANDLE hDevice,
                          PUCHAR pbLinearAddr,
                          ULONG OffsetBytes,
                          ULONG Value)
```

***Visual Basic:***

```
Declare Function WriteRegisterULong Lib "PCI8025" (ByVal hDevice As Long, _
                                                   ByVal pbLinearAddr As Long, _
                                                   ByVal OffsetBytes As Long, _
                                                   ByVal Value As Long) As Boolean
```

***LabVIEW:***




功能：以四字节（即 32 位）方式写 PCI 内存映射寄存器。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

LinearAddr PCI设备内存映射寄存器的线性基地址，它的值应由 GetDeviceAddr确定。

OffsetBytes 相对于LinearAddr线性基地址的偏移字节数，它与LinearAddr两个参数共同确定WriteRegisterULong函数所访问的映射寄存器的内存单元。

Value 输出 32 位整型值。

返回值：若成功，返回 TRUE，否则返回 FALSE。

相关函数：CreateDevice WriteRegisterByte WriteRegisterWord WriteRegisterULong ReadRegisterByte ReleaseDevice ReadRegisterWord ReadRegisterULong

***Visual C++程序举例:***

```
:
HANDLE hDevice;
ULONG LinearAddr, PhysAddr, OffsetBytes;
hDevice = CreateDevice(0)
if (!GetDeviceAddr(hDevice, &LinearAddr,   &PhysAddr, 0) )
{
    AfxMessageBox "取得设备地址失败…";
}
OffsetBytes=100;// 指定操作相对于线性基地址偏移 100 个字节数位置的单元
WriteRegisterULong(hDevice, LinearAddr, OffsetBytes, 0x20000000);   // 往指定映射寄存器单元写入 32 位的十六进制数据
ReleaseDevice( hDevice );   // 释放设备对象
:
```

***Visual Basic 程序举例:***

```
:
Dim hDevice As Long
Dim LinearAddr, PhysAddr, OffsetBytes As Long
hDevice = CreateDevice(0)
GetDeviceAddr( hDevice, LinearAddr, PhysAddr, 0)
OffsetBytes = 100
WriteRegisterULong( hDevice, LinearAddr, OffsetBytes, &H20000000)
ReleaseDevice(hDevice)
:
```

◆ **以单字节（即 8 位）方式读 PCI 内存映射寄存器的某个单元**

函数原型：

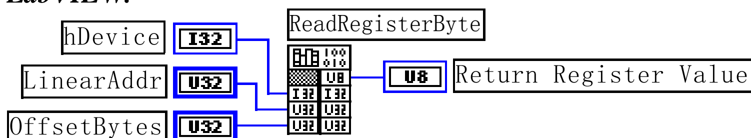
***Visual C++:***

```
BYTE ReadRegisterByte( HANDLE hDevice,
                       PUCHAR pbLinearAddr,
                       ULONG OffsetBytes)
```

***Visual Basic:***

```
Declare Function ReadRegisterByte Lib "PCI8025" (ByVal hDevice As Long, _
                                                  ByVal pbLinearAddr As Long, _
                                                  ByVal OffsetBytes As Long) As Byte
```

***LabVIEW:***




功能：以单字节（即 8 位）方式读 PCI 内存映射寄存器的指定单元。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

LinearAddr　PCI设备内存映射寄存器的线性基地址，它的值应由 GetDeviceAddr确定。

OffsetBytes　相对于LinearAddr线性基地址的偏移字节数，它与LinearAddr两个参数共同确定 ReadRegisterByte函数所访问的映射寄存器的内存单元。

返回值：返回从指定内存映射寄存器单元所读取的 8 位数据。

相关函数：　CreateDevice　WriteRegisterByte　WriteRegisterWord　WriteRegisterULong　ReadRegisterByte　ReadRegisterWord　ReadRegisterWord　ReadRegisterULong　ReleaseDevice

***Visual C++程序举例:***

```
    :
HANDLE hDevice;
ULONG LinearAddr,   PhysAddr, OffsetBytes;
BYTE    Value;
hDevice = CreateDevice(0);   // 创建设备对象
GetDeviceAddr(hDevice, &LinearAddr, &PhysAddr, 0);    // 取得 PCI 设备 0 号映射寄存器的线性基地址
OffsetBytes = 100;     // 指定操作相对于线性基地址偏移 100 个字节数位置的单元
Value = ReadRegisterByte(hDevice, LinearAddr, OffsetBytes);   // 从指定映射寄存器单元读入 8 位数据
ReleaseDevice( hDevice );   // 释放设备对象
    :
```

***Visual Basic 程序举例:***

```
    :
Dim hDevice As Long
Dim LinearAddr, PhysAddr, OffsetBytes As Long
Dim Value As Byte
hDevice = CreateDevice(0)
GetDeviceAddr( hDevice, Linearddr, PhysAddr, 0)
OffsetBytes = 100
Value = ReadRegisterByte( hDevice, LinearAddr, OffsetBytes)
ReleaseDevice(hDevice)
    :
```

♦ **以双字节（即 16 位）方式读 PCI 内存映射寄存器的某个单元**

函数原型：

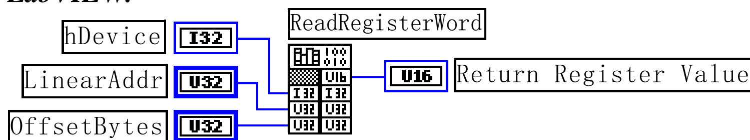
***Visual C++:***

```
WORD ReadRegisterWord( HANDLE hDevice,
                       PUCHAR pbLinearAddr,
                       ULONG OffsetBytes)
```

***Visual Basic:***

```
Declare Function ReadRegisterWord Lib "PCI8025" (  ByVal hDevice As Long, _
                                                   ByVal pbLinearAddr As Long, _
                                                   ByVal OffsetBytes As Long) As Integer
```

***LabVIEW:***




功能：以双字节（即 16 位）方式读 PCI 内存映射寄存器的指定单元。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

LinearAddr PCI设备内存映射寄存器的线性基地址，它的值应由 GetDeviceAddr确定。

OffsetBytes　相对于LinearAddr线性基地址的偏移字节数，它与LinearAddr两个参数共同确定 ReadRegisterWord函数所访问的映射寄存器的内存单元。

返回值：返回从指定内存映射寄存器单元所读取的 16 位数据。

相关函数：　CreateDevice　WriteRegisterByte　WriteRegisterWord　WriteRegisterULong　ReadRegisterByte　ReadRegisterWord　ReadRegisterWord　ReadRegisterULong　ReleaseDevice

***Visual C++程序举例:***

```
    :
HANDLE hDevice;
ULONG LinearAddr,    PhysAddr, OffsetBytes;
WORD Value;
hDevice = CreateDevice(0);    // 创建设备对象
GetDeviceAddr(hDevice, &LinearAddr, &PhysAddr, 0);    // 取得 PCI 设备 0 号映射寄存器的线性基地址
OffsetBytes = 100;    // 指定操作相对于线性基地址偏移 100 个字节数位置的单元
Value = ReadRegisterWord(hDevice, LinearAddr, OffsetBytes);    // 从指定映射寄存器单元读入 16 位数据
ReleaseDevice( hDevice );    // 释放设备对象
    :
```

***Visual Basic 程序举例:***

```
    :
Dim hDevice As Long
Dim LinearAddr, PhysAddr, OffsetBytes As Long
Dim Value As Word
hDevice = CreateDevice(0)
GetDeviceAddr( hDevice, Linearddr, PhysAddr, 0)
OffsetBytes = 100
Value = ReadRegisterWord( hDevice, LinearAddr, OffsetBytes)
ReleaseDevice(hDevice)
    :
```

◆ 以四字节（即 **32** 位）方式读 **PCI** 内存映射寄存器的某个单元

函数原型：

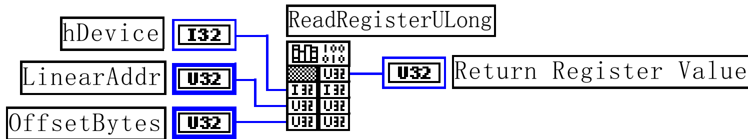
***Visual C++:***

ULONG ReadRegisterULong( HANDLE hDevice,
　　PUCHAR pbLinearAddr,
　　ULONG OffsetBytes)

***Visual Basic:***

Declare Function ReadRegisterULong Lib "PCI8025" (ByVal hDevice As Long, _
　　ByVal pbLinearAddr As Long, _
　　ByVal OffsetBytes As Long) As Long

***LabVIEW:***




功能：以四字节（即 32 位）方式读 PCI 内存映射寄存器的指定单元。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

LinearAddr　PCI设备内存映射寄存器的线性基地址，它的值应由 GetDeviceAddr确定。

OffsetBytes　相对与LinearAddr线性基地址的偏移字节数，它与LinearAddr两个参数共同确定WriteRegisterULong函数所访问的映射寄存器的内存单元。

返回值：返回从指定内存映射寄存器单元所读取的 32 位数据。

相关函数：　CreateDevice　WriteRegisterByte　WriteRegisterWord
WriteRegisterULong　ReadRegisterByte　ReleaseDevice
ReadRegisterWord　ReadRegisterULong

***Visual C++程序举例:***

```
    :
HANDLE hDevice;
ULONG LinearAddr,    PhysAddr, OffsetBytes;
ULONG    Value;
hDevice = CreateDevice(0);    // 创建设备对象
GetDeviceAddr(hDevice, &LinearAddr, &PhysAddr, 0);    // 取得 PCI 设备 0 号映射寄存器的线性基地址
OffsetBytes = 100;    // 指定操作相对于线性基地址偏移 100 个字节数位置的单元
Value = ReadRegisterULong(hDevice, LinearAddr, OffsetBytes);    // 从指定映射寄存器单元读入 32 位数据
```

ReleaseDevice( hDevice );　// 释放设备对象
:

***Visual Basic 程序举例:***

:
Dim hDevice As Long
Dim LinearAddr, PhysAddr, OffsetBytes As Long
Dim Value As Long
hDevice = CreateDevice(0)
GetDeviceAddr( hDevice, Linearddr, PhysAddr, 0)
OffsetBytes = 100
Value = ReadRegisterULong( hDevice, LinearAddr, OffsetBytes)
ReleaseDevice(hDevice)
:

## 第三节、IO 端口读写函数原型说明

注意：若您想在 **WIN2K** 系统的 **User** 模式中直接访问 **I/O** 端口，那么您可以安装光盘中 **ISA\CommUser** 目录下的公用驱动，然后调用其中的 **WritePortByteEx** 或 **ReadPortByteEx** 等有 **"Ex"** 后缀的函数即可。

◆ 以单字节(8Bit)方式写 I/O 端口

函数原型：

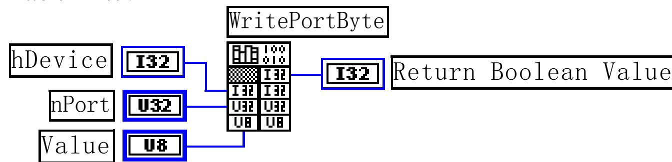
***Visual C++:***

```
BOOL WritePortByte (HANDLE hDevice,
                    PUCHAR pbPort,
                    ULONG offserBytes,
                    BYTE Value)
```

***Visual Basic:***

```
Declare Function WritePortByte Lib "PCI8025" ( ByVal hDevice As Long, _
                                              ByVal pbPort As Long, _
                                              ByVal offserBytes As Long, _
                                              ByVal Value As Byte) As Boolean
```

***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 GetLastErrorEx捕获当前错误码。

相关函数：　CreateDevice　WritePortByte　WritePortWord
　　　　　　WritePortULong　ReadPortByte　ReadPortWord

◆ 以双字(16Bit)方式写 I/O 端口

函数原型：

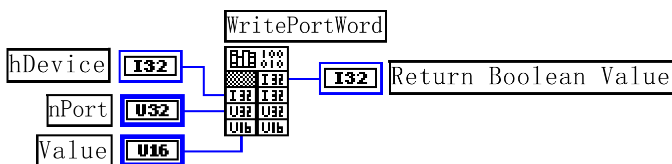
***Visual C++:***

```
BOOL WritePortWord (HANDLE hDevice,
                    PUCHAR pbPort,
                    ULONG offserBytes,
                    WORD Value)
```

***Visual Basic:***

```
Declare Function WritePortWord Lib "PCI8025" (ByVal hDevice As Long, _
                                             ByVal pbPort As Long, _
                                             ByVal offserBytes As Long, _
                                             ByVal Value As Integer) As Boolean
```

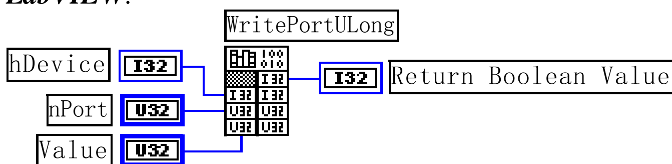
***LabVIEW:***




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 GetLastErrorEx捕获当前错误码。
相关函数： CreateDevice WritePortByte WritePortWord
WritePortULong ReadPortByte ReadPortWord

◆ **以四字节(32Bit)方式写 I/O 端口**

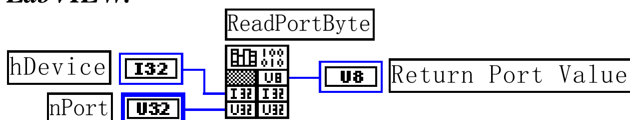
函数原型：
***Visual C++:***
```
BOOL WritePortULong(HANDLE hDevice,
                    PUCHAR pbPort,
                    ULONG offserBytes,
                    ULONG Value)
```
***Visual Basic:***
```
Declare Function WritePortULong Lib "PCI8025" (ByVal hDevice As Long, _
                                               ByVal pbPort As Long, _
                                                ByVal offserBytes As Long, _
                                               ByVal Value As Long ) As Boolean
```
***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 GetLastErrorEx捕获当前错误码。
相关函数： CreateDevice WritePortByte WritePortWord
WritePortULong ReadPortByte ReadPortWord

◆ **以单字节(8Bit)方式读 I/O 端口**

函数原型：
***Visual C++:***
```
BYTE ReadPortByte(HANDLE hDevice,
                  PUCHAR pbPort,
                  ULONG offserBytes)
```
***Visual Basic:***
```
Declare Function ReadPortByte Lib "PCI8025" (ByVal hDevice As Long, _
                                             ByVal offserBytes As Long ) As Byte
                                              ByVal pbPort As Long ) As Byte
```
***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

◆ **以双字节(16Bit)方式读 I/O 端口**

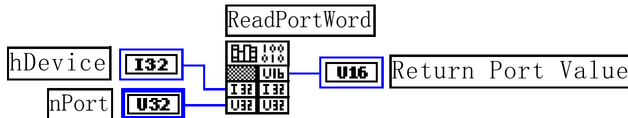
函数原型：
***Visual C++:***

```
WORD ReadPortWord(HANDLE hDevice,
                  PUCHAR pbPort,
                  ULONG offserBytes)
```

***Visual Basic:***

```
Declare Function ReadPortWord Lib "PCI8025" ( ByVal hDevice As Long, _
                                              ByVal offserBytes As Long, _
                                              ByVal pbPort As Long ) As Integer
```

***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

◆ **以四字节(32Bit)方式读 I/O 端口**

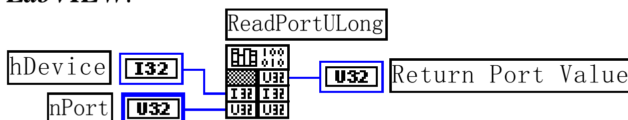
函数原型：
***Visual C++:***

```
ULONG ReadPortULong(HANDLE hDevice,
                    PUCHAR pbPort,
                    ULONG offserBytes)
```

***Visual Basic:***

```
Declare Function ReadPortULong Lib "PCI8025" ( ByVal hDevice As Long, _
                                               ByVal offserBytes As Long, _
                                               ByVal pbPort As Long ) As Long
```

***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

## 第四节、线程操作函数原型说明

（如果您的 VB6.0 中线程无法正常运行，可能是 VB6.0 语言本身的问题，请选用 VB5.0）

♦ 创建内核系统事件

函数原型：

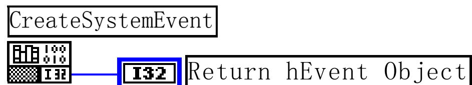
***Visual C++:***

HANDLE CreateSystemEvent(void)

***Visual Basic:***

Declare Function CreateSystemEvent Lib " PCI8025 " () As Long

***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)。

♦ 释放内核系统事件

函数原型：

***Visual C++:***

BOOL ReleaseSystemEvent(HANDLE hEvent)

***Visual Basic:***

Declare Function ReleaseSystemEvent Lib " PCI8025 " (ByVal hEvent As Long) As Boolean

***LabVIEW:***

请参见相关演示程序。

功能：释放系统内核事件对象。

参数：hEvent 被释放的内核事件对象。它应由 CreateSystemEvent成功创建的对象。

返回值：若成功，则返回 TRUE。