

PCI9252 任意波形发生器

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

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	3
第一节、使用上层用户函数，高效、简单	3
第二节、如何管理PCI设备	3
第三节、如何实现DA波形数据输出	3
第四节、哪些函数对您不是必须的	3
第三章 PCI即插即用设备操作函数接口介绍	4
第一节、设备驱动接口函数总列表（每个函数省略了前缀“PCI9252_”）	5
第二节、设备对象管理函数原型说明	5
第三节、DA数据采样操作函数原型说明	7
第四节、DA硬件参数系统保存与读取函数原型说明	12
第四章 硬件参数结构	13
第五章 数据格式转换与排列规则	14
第一节、DA电压值转换成LSB原码数据的换算方法	14
第二节、关于DA数据DABuffer缓冲区中的数据排放规则	14
第六章 上层用户函数接口应用实例	15
第一节、简易程序演示说明	15
第二节、高级程序演示说明	15
第七章 共用函数介绍	16
第一节、公用接口函数总列表（每个函数省略了前缀“PCI9252_”）	16
第二节、内存映射寄存器操作函数原型说明	16
第三节、IO端口读写函数原型说明	20

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 PCIxxx_ 则被省略。如 PCI9252_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[InitDeviceProDA](#)、[WriteDeviceProDA](#)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，您基本上可以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。因为上层函数的命名、参数的命名极其规范。

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其他函数，如[InitDeviceProDA](#)可以使用hDevice句柄以程序查询方式初始化设备的DA部件，[WriteDeviceProDA](#)函数可以用hDevice句柄实现对DA数据的采样读取。最后可以通过[ReleaseDevice](#)将hDevice释放掉。

第三节、如何实现 DA 波形数据输出

当您有了hDevice设备对象句柄后，便可用[InitDeviceProDA](#)函数初始化DA部件，关于频率等参数的设置是由这个函数的pDAPara参数结构体决定的。您只需要对这个pDAPara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。然后调用[WriteDeviceProDA](#)将准备好的DA数据写入板载RAM中，接着用[StartDeviceProDA](#)即可启动DA部件，开始DA输出。当您需要暂停设备时，执行[StopDeviceProDA](#)，当您需要关闭DA设备时，[ReleaseDeviceDA](#)便可帮您实现（但设备对象hDevice依然存在）。

第四节、哪些函数对您不是必须的

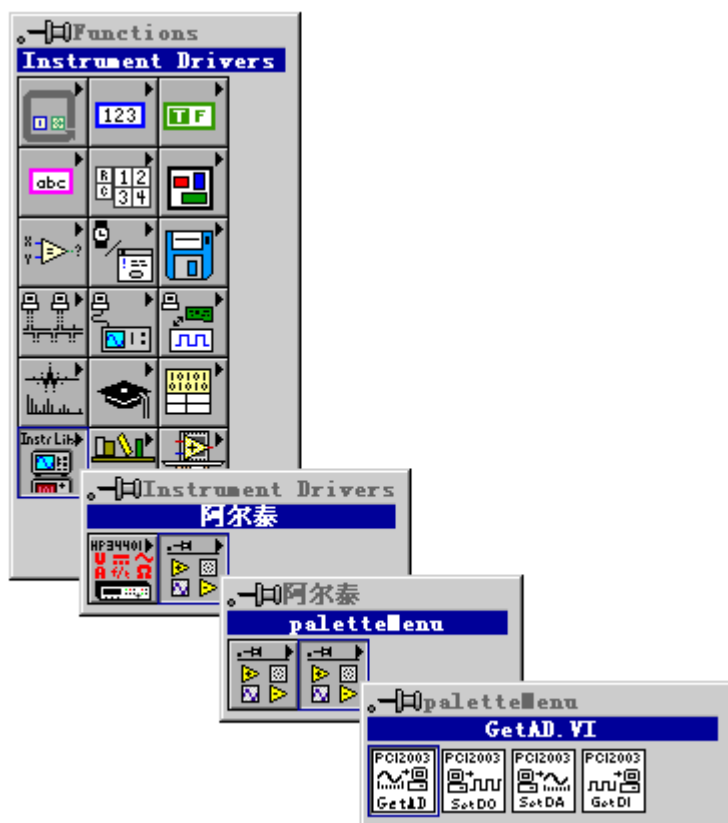
公共函数如[CreateFileObject](#)，[WriteFile](#)，[ReadFile](#)等一般来说都是辅助性函数，除非您要使用存盘功能。如果您使用上层用户函数访问设备，那么[GetDeviceAddr](#)，[WriteRegisterByte](#)，[WriteRegisterWord](#)，[WriteRegisterULong](#)，[ReadRegisterByte](#)，[ReadRegisterWord](#)，[ReadRegisterULong](#)等函数您可完全不必理会，除非您是作为底层用户管理设备。而[WritePortByte](#)，[WritePortWord](#)，[WritePortULong](#)，[ReadPortByte](#)，[ReadPortWord](#)，[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。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 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“PCI9252_”）

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	上层及底层用户
GetDeviceCount	取得同一种 PCI 设备的总台数	上层用户
GetDeviceCurrentID	取得指定设备句柄指向的设备 ID 号	上层用户
ListDeviceDlg	列表所有同一种 PCI 设备的各种配置	上层用户
ReleaseDevice	关闭设备，且释放 PCI 总线设备对象	上层及底层用户
② DA 数据采集操作函数		
ResetDeviceDA	复位整个 DA 设备状态	上层用户
SetDeviceFreqDA	可动态改变 DA 采样频率	上层用户
ClearFIFODA	清除 FIFO 中的数据	上层用户
SetOutputRangeDA	设置 DA 量程	上层用户
InitDeviceProDA	初始化 PCI 设备上的 DA 部件准备传输	上层用户
StartDeviceProDA	启动 DA 设备，开始转换	上层用户
StopDeviceProDA	暂停 DA 设备	上层用户
GetDeviceStatusProDA	取得 FIFO 的状态	上层用户
WriteDeviceProDA	向 DA 的 FIFO 中写入批量数据	
ReleaseDeviceProDA	释放 DA 设备	上层用户
StartCalibration	启动 DA 校准	上层用户
GetDACalibration	设备 DA 校准	上层用户
SetDACalibration	设备 DA 校准	上层用户
StopCalibration	停止 DA 校准	上层用户
③ DA 的硬件参数操作函数		
LoadParaDA	从 Windows 系统中读入硬件参数	上层用户
SaveParaDA	往 Windows 系统写入设备硬件参数	上层用户
ResetParaDA	将注册表中的 DA 参数恢复至出厂默认值	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:\PCI\PCI9252\INCLUDE\PCI9252.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 PCI9252.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 "PCI9252.H"
```

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 DeviceLgcID = 0)
```

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：

DeviceLgcID 逻辑设备 ID(Logic Device 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 DeviceLgcID 标识值为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 PCI9252 模板时，驱动程序逻辑号为“0”来确认和管理第一个设备，若用户接着再添加第二个 PCI9252 模板时，则系统将以逻辑号“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，DeviceLgcID 应置 0，第二个应置 1，也以此类推。但默认值为 0。该参数之所以称为逻辑设备号，是因为每个设备的逻辑号是不能事

先由用户硬性确定的,而是由 BIOS 和操作系统加载设备时,依据主板总线编号等信息进行这个设备 ID 号分配,说得简单点,就是加载设备的顺序编号,编号的递增顺序为 0、1、2、3……。所以用户无法直接固定某一个设备的在设备列表中的物理位置,若想固定,则必须使用物理 ID 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](#) [ReleaseDevice](#)

Visual C++ 程序举例

```

:
HANDLE hDevice; // 定义设备对象句柄
int DeviceLgclID = 0;
hDevice = CreateDevice ( DeviceLgclID ); // 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
}
:

```

◆ 取得本计算机系统中 PCI9252 设备的总数量

函数原型:

Visual C++:

int GetDeviceCount (HANDLE hDevice)

功能: 取得 PCI9252 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 PCI9252 的数量。

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

◆ 用对话框控件列表计算机系统中所有 PCI9252 设备各种配置信息

函数原型:

Visual C++:

BOOL ListDeviceDlg (HANDLE hDevice)

功能: 列表系统中 PCI9252 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 PCI9252 设备的配置情况。

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

◆ 取得该设备当前逻辑 ID 和物理 ID

函数原型:

Visual C++:

**BOOL GetDeviceCurrentID (HANDLE hDevice,
PLONG DeviceLgclID,
PLONG DevicePhysID)**

功能: 取得 PCI9252 设备的数量。

参数: hDevice 设备对象句柄,它应由[CreateDevice](#)创建。

返回值: 如果初始化设备对象成功,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](#) [GetDeviceCount](#) [ReleaseDevice](#)
[GetDeviceCurrentID](#) [ListDeviceDlg](#)

◆ 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)

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、DA 数据采集操作函数原型说明

◆ 复位整个 DA 设备状态

函数原型：

Visual C++:

**BOOL ResetDeviceDA (HANDLE hDevice,
PPCI9252_PARA_DA pDAPara)**

功能：复位整个 DA 设备状态。

参数：

hDevice 设备对象句柄，它应由[CreateDevice](#)创建。

pDAPara 设备对象参数结构，它决定了设备对象的各种状态及工作方式，如采样频率等。关于具体操作请参考《[DA硬件参数结构](#)》。

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

相关函数：

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeD	

◆ 动态改变采样频率

函数原型：

Visual C++:

**BOOL SetDeviceFreqDA (HANDLE hDevice,
ULONG Frequency,
int nTimerChannel)**

功能：在 DA 采样过程中，可动态改变采样频率。

参数：

hDevice 设备对象句柄，它应由[CreateDevice](#)创建。

Frequency 时钟输出频率，单位为 Hz。

nTimerChannel 时钟通道号，取值为(0 或 1)。

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

相关函数：

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 设置 DA 的输出范围

函数原型：

Visual C++:

**BOOL SetOutputRangeDA (HANDLE hDevice,
ULONG OutputRange[4])**

功能: 设置 DA 的输出范围(只适用于 PCI9252)。

参数:

hDevice 管理 DA 对象的设备对象。

OutputRange[4] 输出量程,分别控制四个通道。

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

相关函数:

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 清除 FIFO 中的数据

函数原型:

Visual C++:

**BOOL ClearFIFODA (HANDLE hDevice,
int nDAChannel)**

功能: 清除 FIFO 中的数据。

参数:

hDevice 设备对象句柄, 它应由[CreateDevice](#)创建。

nDAChannel DA 通道号, 取值为: 0-3。

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

相关函数:

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 初始化设备对象

函数原型:

Visual C++:

**BOOL InitDeviceProDA (HANDLE hDevice,
LONG ClockSource,
int nDAChannel)**

功能: 它负责初始化设备对象中的DA部件, 为设备操作就绪有关工作做准备, 如预置DA采集通道、采样频率等。但它并不启动DA设备, 若要启动DA设备, 须在调用此函数之后再调用[StartDeviceProDA](#) (但DA要实际输出波形, 则一般要等待某种触发事件的到来)。

参数:

hDevice 设备对象句柄, 它应由[CreateDevice](#)创建。

nDAChannel DA 通道号, 取值为[0, 3]。

返回值: 如果初始化设备对象成功,则返回 TRUE, 否则返回 FALSE, 用户可用 GetLastError 捕获当前错误码, 并加以分析。

相关函数:

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 启动 DA 设备

函数原型:

Visual C++:

**BOOL StartDeviceProDA (HANDLE hDevice,
int nDAChannel)**

功能：启动DA设备，它必须在调用[InitDeviceProDA](#)后才能调用此函数。调用该函数后它可能立即启动，这就要取决您选择的触发方式或触发源，详细请参考后面的《[触发功能详述](#)》。

参数：

hDevice 设备对象句柄，它应由[CreateDevice](#)创建。

nDAChannel 设备通道号，取值范围为：0-3。

返回值：如果调用成功，则返回 TRUE，且 DA 准备就绪，等待触发事件的到来就开始实际的 DA 输出，否则返回 FALSE，用户可用 `GetLastError` 捕获当前错误码，并加以分析。

相关函数：

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 暂停 DA 设备

函数原型：

Visual C++:

**BOOL StopDeviceProDA (HANDLE hDevice,
int nDAChannel)**

功能：暂停DA设备。它必须在调用[StartDeviceProDA](#)后才能调用此函数。该函数除了停止DA设备不再转换以外，不改变设备的其他任何工作参数。

参数：

hDevice 设备对象句柄，它应由[CreateDevice](#)创建。

nDAChannel 设备通道号，取值范围为：0-3。

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

相关函数：

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 取得 DA 的状态标志

函数原型：

Visual C++:

**BOOL GetDeviceStatusProDA (HANDLE hDevice,
PPCI9252_STATUS_DA pDAStatus)**

功能：在 AD 采样过程中取得 FIFO 的状态，通常用于半满方式读取。

参数：

hDevice 设备对象句柄，它应由[CreateDevice](#)创建。

pDAStatus 取得 FIFO 状态

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

相关函数：

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 向 DA 的 FIFO 中写入批量数据（通常为 FIFO 的半满长度）

函数原型：

Visual C++:

BOOL WriteDeviceProDA (HANDLE hDevice,

PLONG pDABuffer,
ULONG nWriteSizeWords,
int nDAChannel)

功能: 向 DA 的 FIFO 中写入批量数据 (通常为 FIFO 的半满长度)。

参数:

hDevice 设备对象句柄, 它应由[CreateDevice](#)创建。

pDABuffer DA 数据用户缓冲区。

nWriteSizeWords 写入的点数 (以字为单位)。

nDAChannel 设备通道号, 取值范围为: 0-3。

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

相关函数:

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 释放 DA 设备

函数原型:

Visual C++:

BOOL ReleaseDeviceProDA (HANDLE hDevice,
int nDAChannel)

功能: 释放DA设备。它必须在调用[InitDeviceProDA](#)后的某个时刻调用此函数。该函数除了停止DA设备, 还释放掉所占用的各种资源。

参数:

hDevice 设备对象句柄, 它应由[CreateDevice](#)创建。

nDAChannel DA 输出通道。

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

相关函数:

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 启动 DA 校准

函数原型:

Visual C++:

BOOL StartCalibration (HANDLE hDevice)

功能: 启动 DA 校准功能。

参数:

hDevice设备对象句柄, 它应由[CreateDevice](#)创建。

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	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 设备 DA 校准

函数原型:

Visual C++:

BOOL GetDACalibration (HANDLE hDevice,

LONG OutputRange,
LONG CalMode,
PLONG pCalData,
int nDAChannel)

功能：设备 DA 校准功能。

参数：

hDevice设备对象句柄，它应由[CreateDevice](#)创建。

OutputRange 指定通道的输出量程范围。

CalMode 当为 0 时是零点校准，为 1 时是满度校准。

pCalData 校准值。

nDAChannel 需要指定的模拟量通道号，其取值范围为：0-1。

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	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 设置设备 DA 校准

函数原型：

Visual C++:

BOOL SetDACalibration (HANDLE hDevice,
LONG OutputRange,
LONG CalMode,
LONG CalData,
int nDAChannel)

功能：设置设备的 DA 校准功能。

参数：

hDevice设备对象句柄，它应由[CreateDevice](#)创建。

OutputRange 指定通道的输出量程范围。

CalMode 当为 0 时是零点校准，为 1 时是满度校准。

CalData 校准值。

nDAChannel 需要指定的模拟量通道号，其取值范围为：0-1。

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	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA
ReleaseDeviceDA	SetOutputRangeDA	

◆ 停止 DA 校准

函数原型：

Visual C++:

BOOL StopCalibration (HANDLE hDevice)

功能：停止 DA 校准功能。

参数：

hDevice设备对象句柄，它应由[CreateDevice](#)创建。

返回值：若成功，返回 TRUE；否则返回 FALSE。

相关函数：

CreateDevice	ResetDeviceDA	SetDeviceFreqDA
ClearFIFODA	InitDeviceProDA	StartDeviceProDA
StopDeviceProDA	EnableDASYNC	SetDASYNCValue
GetDeviceStatusProDA	ReleaseDevice	WriteDeviceProDA

[ReleaseDeviceDA](#)[SetOutputRangeDA](#)

第四节、DA 硬件参数系统保存与读取函数原型说明

◆ 写设备硬件参数函数到 Windows 系统中

函数原型:

Visual C++:

**BOOL SaveParaDA (HANDLE hDevice,
PPCI9252_PARA_DA pDAPara)**

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](#)。

pDAPara设备硬件参数, 关于PCI9252_PARA_DA的详细介绍请参考PCI9252.h或PCI9252.Bas或PCI9252.Pas函数原型定义文件, 也可参考《[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](#) [LoadParaDA](#) [SaveParaDA](#)
[ReleaseDevice](#)

◆ 从 Windows 系统中读入硬件参数函数

函数原型:

Visual C++:

**BOOL LoadParaDA(HANDLE hDevice,
PPCI9252_PARA_DA pDAPara)**

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](#)。

pDAPara属于PCI9252_PARA_DA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PCI9252_PARA_DA请参考PCI9252.h或PCI9252.Bas或PCI9252.Pas函数原型定义文件, 也可参考《[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](#) [LoadParaDA](#) [SaveParaDA](#)
[ReleaseDevice](#)

◆ 将硬件参数结构体值复位为出厂默认值

函数原型:

Visual C++:

**BOOL ResetParaDA (HANDLE hDevice,
PPCI9252_PARA_DA pDAPara)**

功能: 负责将硬件参数的值复位至出厂默认值, 不仅会将 pDAPara 指向的结构体成员值更新为默认值, 同时会将系统中保存的参数更新为默认值。这些默认值在产品驱动第一次被安装时会出现。而且这些默认值的设定是充分的考虑到用户的实际情况, 确保用户不用外部任何条件, 只要开始采集数据, 即可获得相应的结果。

参数:

hDevice 设备对象句柄, 它应由[CreateDevice](#)。

pDAPara设备硬件参数, 关于PCI9252_PARA_DA的详细介绍请参考PCI9252.h或PCI9252.Bas或PCI9252.Pas函数原型定义文件, 也可参考《[硬件参数结构](#)》关于该结构的有关说明。调用此函数后, 该参数指向的结构体成员将被复位至默认值。

nDAChannel DA 通道号, 取值范围为[0, 1]。

返回值: 若成功, 返回 TRUE, 它表明已成功将系统中的 DA 参数复位至默认值, 同时更新了 pDAPara 指向的结构体。否则返回 FALSE。

相关函数: [CreateDevice](#) [LoadParaDA](#) [SaveParaDA](#)
[ResetParaDA](#) [ReleaseDevice](#)

第四章 硬件参数结构

DA 硬件参数结构 (PCI9252_PARA_DA)

Visual C++:

```
typedef struct _PCI9252_PARA_DA
{
    LONG bEnableTimer0;    // 1:允许时钟 0 工作; 0:禁止时钟 0 工作
    LONG Frequency0;       // 时钟 0 输出频率
    LONG bEnableTimer1;    // 1:允许时钟 1 工作; 0:禁止时钟 1 工作
    LONG Frequency1;       // 时钟 1 输出频率
    LONG TriggerSource;     // 内触发和外触发方式选择
    LONG OutTriggerEdge;   // 外触发上升沿和下降沿类型选择
    LONG bClockOutput;     // 是否允许时钟输出
    LONG bImmediateClkOut; // 是否立即启动定时器输出
    LONG ClockOutputNum;    // 选择哪一路时钟源作为输出(0:Timer0; 1:Timer1)
} PCI9252_PARA_DA, *PPCI9252_PARA_DA;
```

bEnableTimer0 时钟 0 工作选择。=1，允许时钟 0 工作；=0：禁止时钟 0 工作。

Frequency0 时钟 0 输出频率。

bEnableTimer1 时钟 1 工作选择。=1，允许时钟 1 工作；=0：禁止时钟 1 工作。

Frequency1 时钟 1 输出频率。

TriggerSource 内触发和外触发方式选择。

常量名	常量值	功能定义
PCI9252_IN_TRIGGER	0x0000	内触发启动方式
PCI9252_OUT_TRIGGER	0x0001	外触发启动方式

OutTriggerEdge 外触发上升沿和下降沿类型选择。

常量名	常量值	功能定义
PCI9252_FALLING_EDGE	0x0000	外触发下降沿触发方式
PCI9252_RISING_EDGE	0x0001	外触发上升沿触发方式

bClockOutput 是否允许时钟输出。

常量名	常量值	功能定义
PCI9252_ENABLE_CLOCK_OUT	0x0000	允许本卡上的自带时钟向外输出
PCI9252_DISABLE_CLOCK_OUT	0x0001	禁止本卡上的自带时钟向外输出

bImmediateClkOut 是否立即启动定时器输出。

ClockOutputNum 选择哪一路时钟源作为输出(=0: Timer0; =1: Timer1)。

ClockSource 参数使用的时钟源选项。

常量名	常量值	功能定义
PCI9252_IN_CLOCK0	0x0000	使用本设备上的内部时钟 0
PCI9252_IN_CLOCK1	0x0001	使用本设备上的内部时钟 1
PCI9252_FRONTENDINTERFACE_CLOCK	0x0002	使用前端接口上的外时钟
PCI9252_SyncClockLine_OUT_CLOCK	0x0003	使用同步时钟线上的外时钟

nDAChannel 参数。

常量名	常量值	功能定义
PCI9252_ALL_CHANNEL	0xFFFF	当需要对所有通道进行操作时

DA FIFO 状态参数 (PCI9252_STATUS_DA)**Visual C++:**

typedef struct _PCI9252_STATUS_DA

{

ULONG bOverflow0; // 板载 FIFO 存储器的 DA0 溢出标志, = TRUE 已发生溢出, = FALSE 未发生溢出

ULONG bHalf0; // 板载 FIFO 存储器的 DA0 半满标志, = TRUE 半满以上, = FALSE 半满以下

ULONG bNotEmpty0; // 板载 FIFO 存储器的 DA0 非空标志, = TRUE 非空, = FALSE 空

ULONG bOverflow1; // 板载 FIFO 存储器的 DA1 溢出标志, = TRUE 已发生溢出, = FALSE 未发生溢出

ULONG bHalf1; // 板载 FIFO 存储器的 DA1 半满标志, = TRUE 半满以上, = FALSE 半满以下

ULONG bNotEmpty1; // 板载 FIFO 存储器的 DA1 非空标志, = TRUE 非空, = FALSE 空

ULONG bOverflow2; // 板载 FIFO 存储器的 DA2 溢出标志, = TRUE 已发生溢出, = FALSE 未发生溢出

ULONG bHalf2; // 板载 FIFO 存储器的 DA2 半满标志, = TRUE 半满以上, = FALSE 半满以下

ULONG bNotEmpty2; // 板载 FIFO 存储器的 DA2 非空标志, = TRUE 非空, = FALSE 空

ULONG bOverflow3; // 板载 FIFO 存储器的 DA3 溢出标志, = TRUE 已发生溢出, = FALSE 未发生溢出

ULONG bHalf3; // 板载 FIFO 存储器的 DA3 半满标志, = TRUE 半满以上, = FALSE 半满以下

ULONG bNotEmpty3; // 板载 FIFO 存储器的 DA3 非空标志, = TRUE 非空, = FALSE 空

} PCI9252_STATUS_DA, *PPCI9252_STATUS_DA;

OutputRange 模拟量输入范围所使用的选项。

常量名	常量值	功能定义
PCI9252_OUTPUT_0_P5000	0x00	0~5000mV
PCI9252_OUTPUT_0_P10000	0x01	0~10000mV
PCI9252_OUTPUT_N5000_P5000	0x02	±5000mV
PCI9252_OUTPUT_N10000_P10000	0x03	±10000mV

第五章 数据格式转换与排列规则**第一节、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
0~5000mV	$Lsb = Volt / (5000.00/65536)$	[0, 65535]
0~10000mV	$Lsb = Volt / (10000.00/65536)$	[0, 65535]
±5000mV	$Lsb = Volt / (10000.00/65536) + 32768$	[0, 65535]
±10000mV	$Lsb = Volt / (20000.00/65536) + 32768$	[0, 65535]

第二节、关于 DA 数据 DABuffer 缓冲区中的数据排放规则

由于各个通道的段信息与波形数据均共享一个板载物理 RAM, 它们的排放顺序如图 5.1, 但各个通道所占 RAM 空间大小均可以通过函数 SetDevRamSizeDA()调整, 系统默认值为四个通道均分整个 RAM 空间, 即默认

每通道 RAM 空间为 256K 点。从下图可以看出，各个通道所占 RAM 空间不一定相等，可大可小，只是四个通道的总空间不能大于板载物理 RAM 空间即可。

通道 0	通道 1	通道 2	通道 3
0 至(256K-1)	256 至(512K-1)	512 至(768K-1)	768K 至(1024K-1)

图 5.1

关于每个通道 RAM 空间的内部分配是这样的，其空间首部存放的是所有段的段信息数据，其后才是各个段的波形数据，再其后可能还有未用空间。假如有三个分段，如图 5.2:

段 0 信息	段 1 信息	段 2 信息	段 0 波形数据	段 1 波形数据	段 2 波形数据	未用空间
-----------	-----------	-----------	----------	----------	----------	------

图 5.2

关于每个段的段信息包括的内容有：该段波形数据在 RAM 中的起始地址、终止地址、段循环次数，如表 5.2.1，注意其段起始地址和终止地址是由 PCI9252_PARA_DA 中的 SegmentInfo 决定的。

表 5.2.1

板载 RAM 内存单元(16Bit)	各单元定义	有效位
0	段 0 波形数据起始地址低 12 位	D11:D0
1	段 0 波形数据起始地址高 8 位	D7:D0
2	段 0 波形数据终止地址低 12 位	D11:D0
3	段 0 波形数据终止地址高 8 位	D7:D0
4	段 0 循环次数低 12 位	D11:D0
5	段 0 循环次数高 8 位	D7:D0
6	段 1 波形数据起始地址低 12 位	D11:D0
7	段 1 波形数据起始地址高 8 位	D7:D0
8	段 1 波形数据终止地址低 12 位	D11:D0
9	段 1 波形数据终止地址高 8 位	D7:D0
10	段 1 循环次数低 12 位	D11:D0
11	段 1 循环次数高 8 位	D7:D0
:	:	:
段信息结束后便是波形数据	段信息结束后便是波形数据	D11:D0

第六章 上层用户函数接口应用实例

第一节、简易程序演示说明

一、怎样使用 WriteDeviceProDA 函数进行批量 DA 数据输出

其详细应用实例及工程级代码请参考 Visual C++ 简易演示系统及源程序，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(主要参考 PCI9252.h 和 Sys.cpp)。

[程序] | [阿尔泰测控演示系统] | [Microsoft Visual C++] | [简易代码演示] | [DA 批量输出演示源程序]

其简易程序默认存放路径为：系统盘\PCI\PCI9252\SAMPLES\VC\SIMPLE\DA\BULK

其他语言的演示可以用上面类似的方法找到。

第二节、高级程序演示说明

高级程序演示了本设备的所有功能，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(主要参考 PCI9252.h 和 DADoc.cpp)。

[程序] | [阿尔泰测控演示系统] | [Microsoft Visual C++] | [高级代码演示]

其默认存放路径为：系统盘\PCI\PCI9252\SAMPLES\VC\ADVANCED

其他语言的演示可以用上面类似的方法找到。

第七章 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“PCI9252_”）

函数名	函数功能	备注
① PCI 总线内存映射寄存器操作函数		
GetDeviceBar	取得指定 PCI 设备寄存器操作基地址	底层用户
GetDeviceAddr	取得指定的指定设备寄存器组的线性基地址和物理地址	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

第二节、内存映射寄存器操作函数原型说明

◆取得指定的指定设备寄存器组 BAR 地址

函数原型:

Visual C++:

```
BOOL GetDeviceBar ( HANDLE hDevice,
                   PCHAR pbPCIBar[6])
```

功能: 取得指定的指定设备寄存器组 BAR 地址。

参数:

hDevice设备对象句柄，它应由[CreateDevice](#)创建。

pbPCIBar[6]) 返回 PCI BAR 所有地址,具体 PCI BAR 中有多少可用地址请看硬件说明书。

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GetLastErrorEx捕获当前错误码，并加以分析。

相关函数: [CreateDevice](#) [GetDeviceAddr](#) [WriteRegisterByte](#)
 [WriteRegisterWord](#) [WriteRegisterULong](#) [ReadRegisterByte](#)
 [ReadRegisterWord](#) [ReadRegisterULong](#) [ReleaseDevice](#)

Visual C++ 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...");
}
```

```
}  
:
```

◆ 取得指定的指定设备寄存器组的线性基地址和物理地址

函数原型:

Visual C++:

```
BOOL GetDeviceAddr ( HANDLE hDevice,  
                    PCHAR* LinearAddr,  
                    PCHAR* PhysAddr,  
                    int RegisterID = 0)
```

功能: 获取设备固件及程序版本。

参数:

hDevice 设备对象句柄, 它应由[CreateDevice](#)创建。

LinearAddr 返回指定寄存器组的线性地址。

PhysAddr 返回指定寄存器组的物理地址。

RegisterID = 0 设备寄存器组的 ID 号 (0-5)

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GetLastErrorEx捕获当前错误码, 并加以分析。

相关函数: [CreateDevice](#) [GetDeviceAddr](#) [WriteRegisterByte](#)
 [WriteRegisterWord](#) [WriteRegisterULong](#) [ReadRegisterByte](#)
 [ReadRegisterWord](#) [ReadRegisterULong](#) [ReleaseDevice](#)

◆ 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,  
                      PCHAR LinearAddr,  
                      ULONG OffsetBytes,  
                      BYTE Value)
```

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](#) [GetDeviceAddr](#) [WriteRegisterByte](#)
 [WriteRegisterWord](#) [WriteRegisterULong](#) [ReadRegisterByte](#)
 [ReadRegisterWord](#) [ReadRegisterULong](#) [ReleaseDevice](#)

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  
:
```

◆ 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,
                        PCHAR LinearAddr,
                        ULONG OffsetBytes,
                        WORD Value)
```

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	GetDeviceAddr	WriteRegisterByte
WriteRegisterWord	WriteRegisterULong	ReadRegisterByte
ReadRegisterWord	ReadRegisterULong	ReleaseDevice

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
:

```

◆ 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,
                        PCHAR LinearAddr,
                        ULONG OffsetBytes,
                        ULONG Value)
```

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	GetDeviceAddr	WriteRegisterByte
WriteRegisterWord	WriteRegisterULong	ReadRegisterByte
ReadRegisterWord	ReadRegisterULong	ReleaseDevice

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
:
```

◆ 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,
                       PCHAR LinearAddr,
                       ULONG OffsetBytes)
```

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](#) [GetDeviceAddr](#) [WriteRegisterByte](#)
[WriteRegisterWord](#) [WriteRegisterULong](#) [ReadRegisterByte](#)
[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
:
```

◆ 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,
                       PCHAR LinearAddr,
                       ULONG OffsetBytes)
```

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](#) [GetDeviceAddr](#) [WriteRegisterByte](#)
[WriteRegisterWord](#) [WriteRegisterULong](#) [ReadRegisterByte](#)

[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
:

```

◆ 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,
                        PCHAR LinearAddr,
                        ULONG OffsetBytes)

```

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](#) [GetDeviceAddr](#) [WriteRegisterByte](#)
[WriteRegisterWord](#) [WriteRegisterULong](#) [ReadRegisterByte](#)
[ReadRegisterWord](#) [ReadRegisterULong](#) [ReleaseDevice](#)

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
:

```

第三节、I/O 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,
                   PCHAR nPort,
                   BYTE Value)

```

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 写出的 8 位整型数据。

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 nPort,
 WORD Value)

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 nPort,
 ULONG Value)

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 nPort)

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 nPort)

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 nPort)

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](#)