PCI9757 WIN2000/XP 驱动程序使用说明书

请您务必阅读《使用纲要》，他会使您事半功倍!

目　　录

PCI9757 WIN2000/XP 驱动程序使用说明书 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、如何用非空查询方式取得AD数据 2
第四节、如何用半满查询方式取得AD数据 3
第五节、如何用Dma直接内存方式取得AD数据 3
第六节、哪些函数对您不是必须的 7
第三章 PCI即插即用设备操作函数接口介绍 7
第一节、设备驱动接口函数总列表（每个函数省略了前缀“PCI9757_”） 8
第二节、设备对象管理函数原型说明 9
第三节、AD程序查询方式采样操作函数原型说明 12
第四节、AD直接内存存取DMA方式采样操作函数原型说明 16
第五节、AD硬件参数保存与读取函数原型说明 20
第四章 硬件参数结构 22
第一节、AD硬件参数结构（PCI9757_PARA_AD） 22
第二节、AD状态参数结构（PCI9757_STATUS_AD） 23
第三节、DMA状态参数结构（PCI9757_STATUS_DMA） 24
第五章 各种功能的使用方法 25
第一节、AD触发功能的使用方法 25
第二节、AD内时钟与外时钟功能的使用方法 27
第三节、AD触发功能的变通与扩展 27
第六章 数据格式转换与排列规则 28
第一节、AD原码LSB数据转换成电压值的换算方法 28
第二节、AD采集函数的ADBuffer缓冲区中的数据排放规则 28
第三节、AD测试应用程序创建并形成的数据文件格式 29
第七章 上层用户函数接口应用实例 30
第一节、简易程序演示说明 30
第二节、高级程序演示说明 30
第八章 高速大容量、连续不间断数据采集及存盘技术详解 30
第一节、使用程序查询方式实现该功能 32
第二节、使用DMA方式实现该功能 33
第九章 共用函数介绍 33
第一节、公用接口函数总列表（每个函数省略了前缀“PCI9757_”） 33
第二节、PCI内存映射寄存器操作函数原型说明 33
第三节、IO端口读写函数原型说明 39
第四节、线程操作函数原型说明 42

第一章 版权信息与命名约定

第一节、版权信息

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 PCI9757_CreateDevice 则写为 CreateDevice。

二、函数名及参数中各种关键字缩写规则

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			

以上规则不局限于该产品。

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 InitDeviceProAD、InitDeviceDmaAD、ReadDeviceProAD_Npt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 InitDeviceProAD可以使用hDevice句柄以程序查询方式初始化设备的AD部件，ReadDeviceProAD_Npt (或 ReadDeviceProAD_Half)函数可以用hDevice句柄实现对AD数据的采样读取等。最后可以通过 ReleaseDevice将hDevice释放掉。

第三节、如何用非空查询方式取得 AD 数据

当您有了hDevice设备对象句柄后，便可用 InitDeviceProAD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 StartDeviceProAD即可启动AD部件，开始AD采样，然后便可用 ReadDeviceProAD_Npt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StopDeviceProAD，当您需要关闭AD设备时，ReleaseDeviceProAD便可帮您实现（但设备对象hDevice依然存在）。（注：ReadDeviceProAD_Npt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、如何用半满查询方式取得 AD 数据

当您有了hDevice设备对象句柄后，便可用 InitDeviceProAD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 StartDeviceProAD即可启动AD部件，开始AD采样，接着调用 GetDevStatusProAD函数以查询AD的存储器FIFO的半满状态，如果达到半满状态，即可用ReadDeviceProAD_Half函数读取一批半满长度（或半满以下）的AD数据，然后接着再查询FIFO的半满状态，若有效再读取，就这样反复查询状态反复读取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StopDeviceProAD，当您需要关闭AD设备时，ReleaseDeviceProAD便可帮您实现（但设备对象hDevice依然存在）。（注：ReadDeviceProAD_Half函数在半满状态有效时也可以单点或几点的方式读取AD数据，只是到下一次半满信号到来时的时间间隔会变得非常短，而不再是半满间隔。）具体执行流程请看下面的图 2.1.2。

第五节、如何用 Dma 直接内存方式取得 AD 数据

当您有了hDevice设备对象句柄后，便可用 InitDeviceDmaAD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 InitDeviceDmaAD的相应参数，它将作为Dma事件的变量。然后用 StartDeviceDmaAD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 GetDevStatusDmaAD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 StopDeviceDmaAD，当您需要关闭AD设备时，ReleaseDeviceDmaAD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 2.1.3。

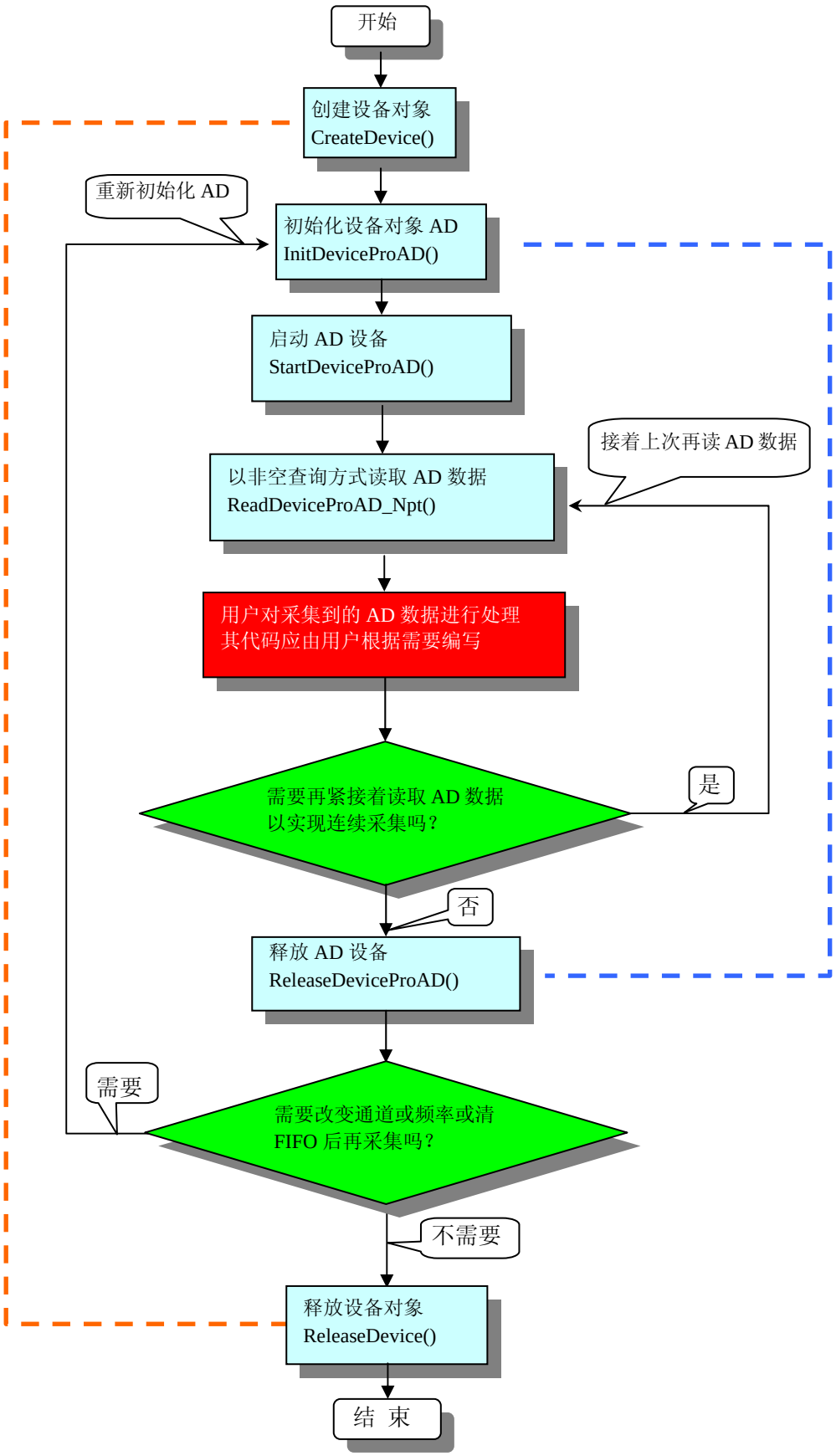


图 2.1.1 非空查询方式 AD 采集过程

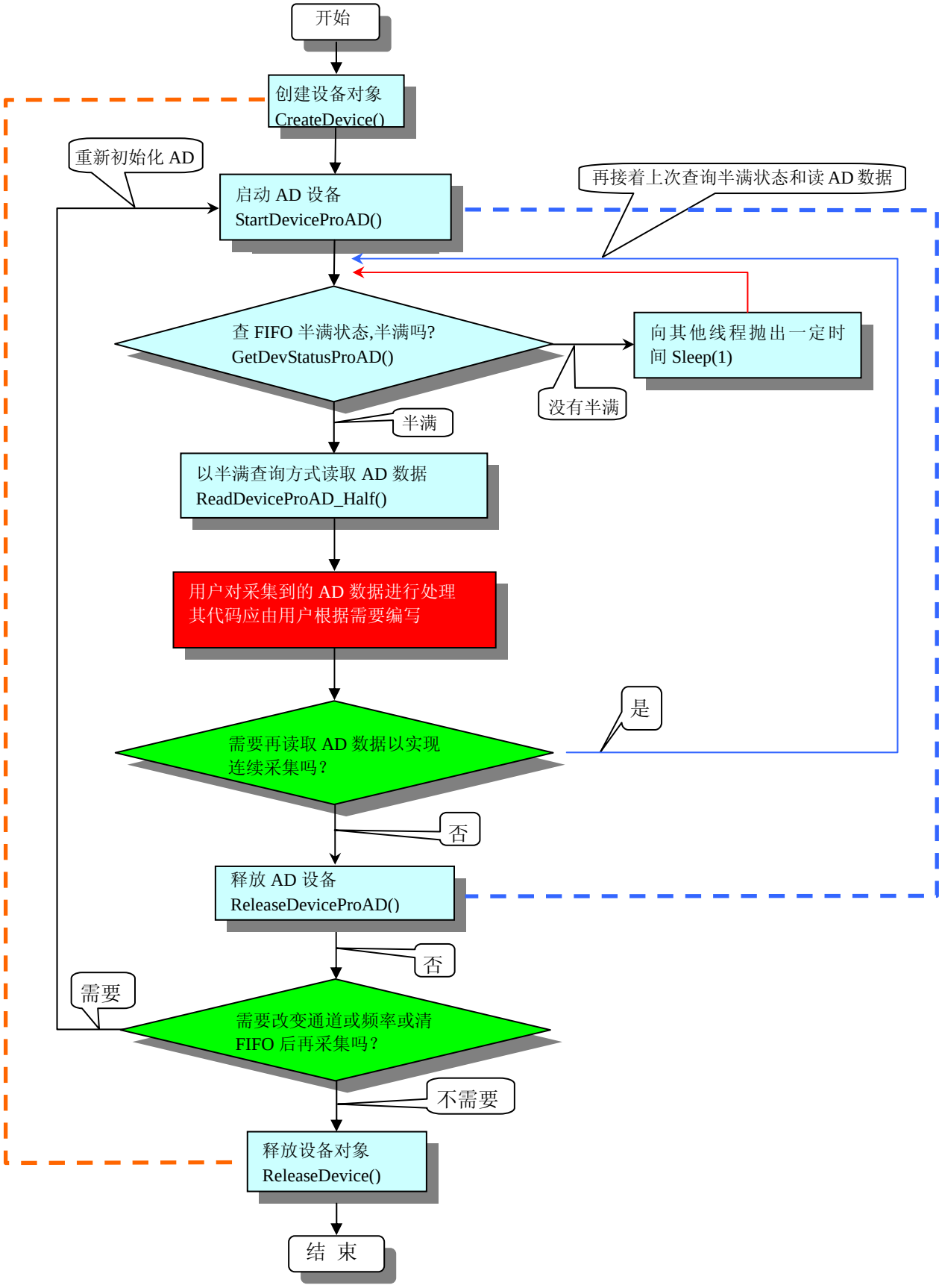


图 2.1.2 半满查询方式 AD 采集过程

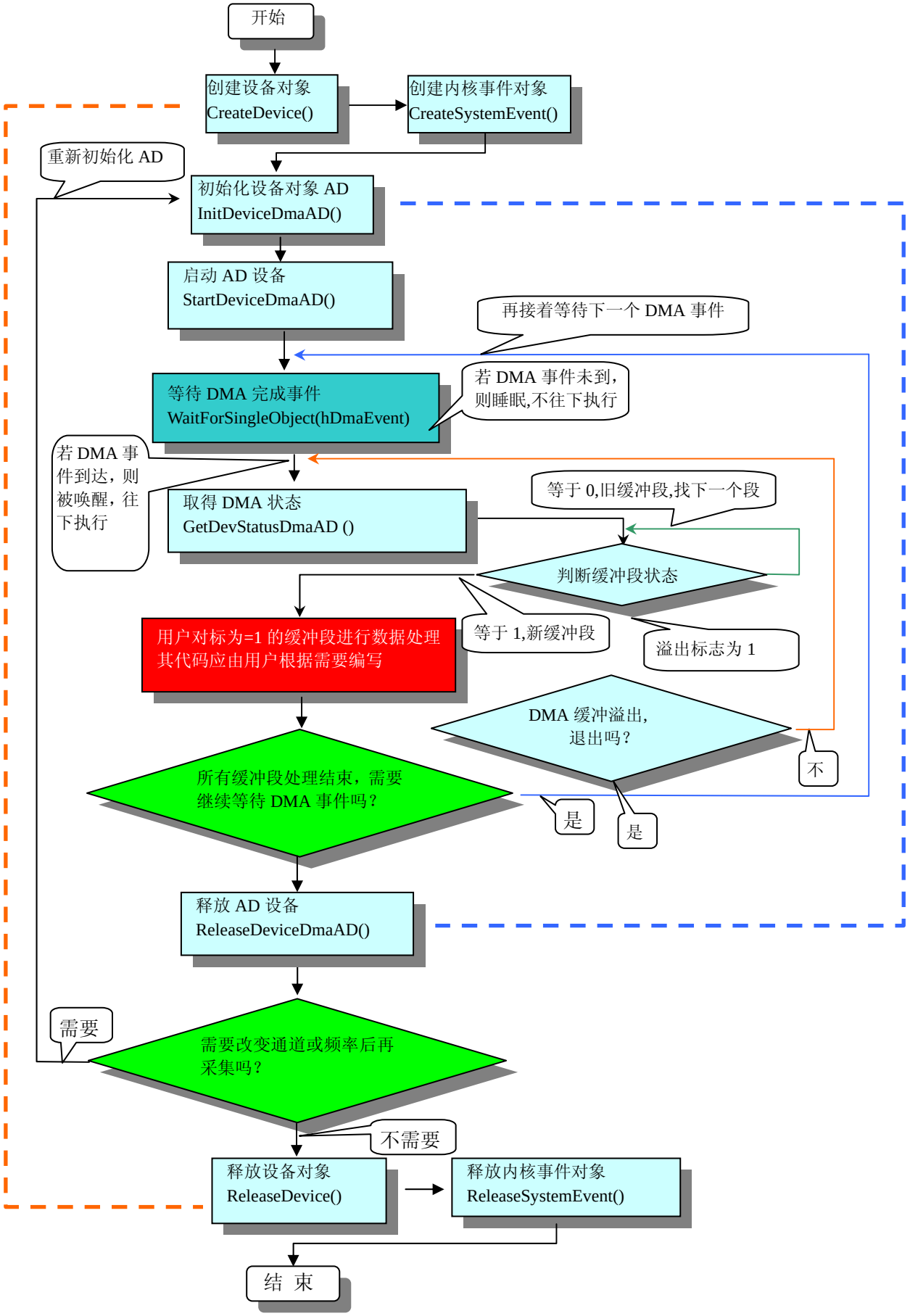


图 2.1.3 DMA 方式 AD 采集实现过程

第六节、哪些函数对您不是必须的

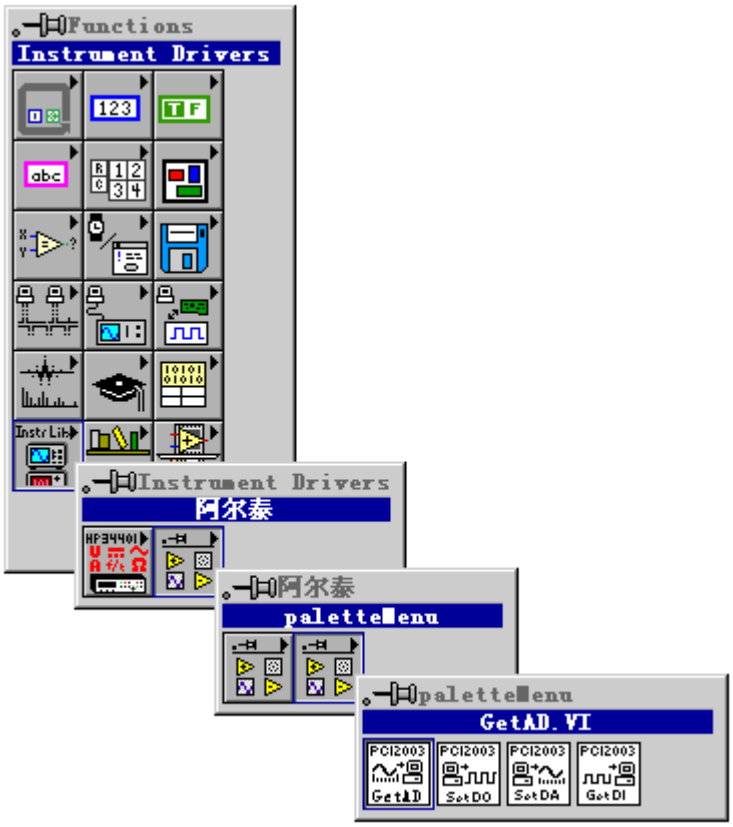
公共函数如 CreateFileObject， WriteFile，ReadFile等一般来说都是辅助性函数，除非您要使用存盘功能。如果您使用上层用户函数访问设备，那么 GetDeviceAddr， WriteRegisterByte， WriteRegisterWord， WriteRegisterULong， ReadRegisterByte， ReadRegisterWord， ReadRegisterULong等函数您可完全不必理会，除非您是作为底层用户管理设备。而 WritePortByte， WritePortWord， WritePortULong， ReadPortByte， ReadPortWord， 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 InitDeviceProAD）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_Npt（或 ReadDeviceProAD_Half）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“PCI9757_”）

函数名	函数功能	备注
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
GetDeviceCurrentID	取得指定设备的逻辑 ID 和物理 ID	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
程序方式 AD 读取函数		
Calibration	AD 校准	上层用户
InitDeviceProAD	初始化 AD 部件准备传输	上层用户
StartDeviceProAD	启动 AD 设备，开始转换	上层用户
ReadDeviceProAD_Npt	连续读取当前 PCI 设备上的 AD 数据	上层用户
GetDevStatusProAD	取得当前 PCI 设备 FIFO 半满状态	上层用户
ReadDeviceProAD_Half	连续批量读取 PCI 设备上的 AD 数据	上层用户
StopDeviceProAD	暂停 AD 设备	上层用户
ReleaseDeviceProAD	释放设备上的 AD 部件	上层用户
DMA 方式 AD 读取函数（唯有此种方式效率最高）		
InitDeviceDmaAD	初始化 AD 部件，如通道等	上层用户
StartDeviceDmaAD	启动 AD 采集	上层用户
GetDevStatusDmaAD	取得 DMA 的各种状态	上层用户
SetDevStatusDmaAD	清除 DMA 状态	
StopDeviceDmaAD	停止 AD 采集	上层用户
ReleaseDeviceDmaAD	释放设备上的 AD 部件	上层用户
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	将注册表中的 AD 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:\Art\PCI9757\INCLUDE\PCI9757.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 PCI9757.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。

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。

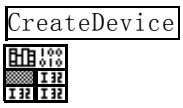
Visual Basic:

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 PCI9757.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 PCI9757.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_Npt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：

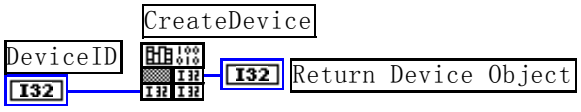
Visual C++：

HANDLE CreateDevice (int DeviceLgcID = 0)

Visual Basic：

Declare Function CreateDevice Lib "PCI9757_32" (Optional ByVal DeviceLgcID As Long = 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：

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PCI9757 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PCI9757 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，则必须须使用物理ID号，调用 CreateDeviceEx函数实现。

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 GetDeviceCurrentID ListDeviceDlg ReleaseDevice

Visual C++程序举例

```
    :
HANDLE hDevice;     // 定义设备对象句柄
int DeviceLgcID = 0;
hDevice = PCI9757_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 = PCI9757_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 PCI9757 设备的总数量**

函数原型：

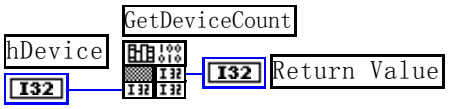
Visual C++:

```
int GetDeviceCount (HANDLE hDevice)
```

Visual Basic:

```
Declare Function GetDeviceCount Lib "PCI9757_32" (ByVal hDevice As Long) As Long
```

LabVIEW：



功能：取得 PCI9757 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 PCI9757 的数量。

相关函数： CreateDevice GetDeviceCount GetDeviceCurrentID ListDeviceDlg 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 DeviceLgcID，
                         PLONG DevicePhysID)
```

Visual Basic:

```
Declare Function GetDeviceCurrentID Lib "PCI9757_32" (ByVal hDevice As Long,_
                                                      ByRef DeviceLgcID As Long,_
                                                      ByRef DevicePhysID As Long) As Boolean
```

LabVIEW：

请参考相关演示程序。

功能：取得指定设备逻辑和物理 ID 号。

参数：

hDevice 设备对象句柄，它指向要取得逻辑和物理号的设备，它应由 CreateDevice创建。

DeviceLgcID 返回设备的逻辑 ID，它的取值范围为[0, 15]。

DevicePhysID 返回设备的物理 ID，它的取值范围为[0, 15]，它的具体值由卡上的拔码器 DID 决定。

返回值：如果初始化设备对象成功，则返回 TRUE， 否则返回 FALSE。

相关函数： CreateDevice GetDeviceCount GetDeviceCurrentID ListDeviceDlg ReleaseDevice

- **用对话框控件列表计算机系统中所有 PCI9757 设备各种配置信息**

函数原型：

Visual C++:

```
BOOL ListDeviceDlg (HANDLE hDevice)
```

Visual Basic:

```
Declare Function ListDeviceDlg Lib "PCI9757_32" (ByVal hDevice As Long) As Boolean
```

LabVIEW：

请参考相关演示程序。

功能：列表系统中 PCI9757 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 PCI9757 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：

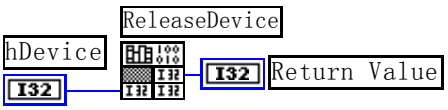
Visual C++:

```
BOOL ReleaseDevice(HANDLE hDevice)
```

Visual Basic:

```
Declare Function ReleaseDevice "PCI9757_32" (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。

返回值：若成功，则返回 TRUE， 否则返回 FALSE。

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

♦ AD 校准

函数原型：

Visual C++：

BOOL Calibration (HANDLE hDevice)

Visual Basic:

Declare Function Calibration Lib "PCI9757_32" (ByVal hDevice As Long) As Long

LabVIEW:

请参考相关演示程序。

功能：AD 功能的校准。

参数：

hDevice 设备对象句柄，它应由设备的 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。

相关函数： CreateDevice, InitDeviceAD, StartDeviceAD, SetDeviceTrigAD, GetDevStatusAD, ReadDeviceProAD, ReadDeviceDmaAD, StopDeviceAD, ReleaseDeviceAD, ReleaseDevice

♦ 初始化 AD 设备（ Initlize device AD for program mode）

函数原型：

Visual C++:

```
BOOL InitDeviceProAD( HANDLE hDevice,
                      PPCI9757_PARA_AD pADPara)
```

Visual Basic:

```
Declare Function InitDeviceProAD Lib "PCI9757_32" (ByVal hDevice As Long, _
                                                   ByRef pADPara As PCI9757_PARA_AD) As Boolean
```

LabVIEW：

请参考相关演示程序。

功能：它负责初始化设备对象中的AD部件，为设备的操作就绪做有关准备工作，如预置AD采集通道、采样频率等。但它并不启动AD设备，若要启动AD设备，须在调用此函数之后再调用 StartDeviceProAD。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

pADPara 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AD采样通道、采样频率等。关于PCI9757_PARA_AD具体定义请参考PCI9757.h(.Bas或.Pas或.VI)驱动接口文件及本文档中的《AD硬件参数结构》章节。

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, SetDevFrequencyAD, InitDeviceProAD, StartDeviceProAD, ReadDeviceProAD_Npt, GetDevStatusProAD, ReadDeviceProAD_Half, StopDeviceProAD, ReleaseDeviceProAD, ReleaseDevice

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

函数原型：

Visual C++:

BOOL StartDeviceProAD (HANDLE hDevice)

Visual Basic:

Declare Function StartDeviceProAD Lib "PCI9757_32" (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 InitDeviceProAD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　InitDeviceProAD
StartDeviceProAD　ReadDeviceProAD_Npt　GetDevStatusProAD
ReadDeviceProAD_Half　StopDeviceProAD　ReleaseDeviceProAD
ReleaseDevice

♦ **读取 PCI 设备上的 AD 数据**

① 使用 FIFO 的非空标志读取 AD 数据

函数原型:

Visual C++:

```
BOOL ReadDeviceProAD_Npt( HANDLE hDevice,
                          LONG ADBuffer[],
                          LONG nReadSizeWords,
                          PLONG nRetSizeWords)
```

Visual Basic:

```
Declare Function ReadDeviceProAD_Npt Lib "PCI9757_32" ( _
                                    ByVal hDevice As Long, _
                                    ByRef ADBuffer As Long, _
                                    ByVal nReadSizeWords As Long, _
                                    ByRef nRetSizeWords As Long) As Boolean
```

LabVIEW：

请参考相关演示程序。

功能：一旦用户使用 StartDeviceProAD后，应立即用此函数读取设备上的AD数据。此函数使用FIFO的非空标志进行读取AD数据。

参数：

hDevice设备对象句柄，它应由 CreateDevice。

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》。

nReadSizeWords 指定一次 ReadDeviceProAD_Npt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 返回实际读取的点数(或字数)。

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 ReleaseDeviceProAD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。

当前错误码	功能定义
0xE1000000	其他不可预知的错误
0xE2000000	表示用户提前终止读操作

注释：此函数也可用于单点读取和几个点的读取，只需要将nReadSizeWords设置成 1 或相应值即可。其使用方法请参考《高速大容量、连续不间断数据采集及存盘技术详解》章节。

相关函数： CreateDevice　SetDevFrequencyAD　InitDeviceProAD
StartDeviceProAD　ReadDeviceProAD_Npt　GetDevStatusProAD
ReadDeviceProAD_Half　StopDeviceProAD　ReleaseDeviceProAD
ReleaseDevice

② 使用 FIFO 的半满标志读取 AD 数据

♦ **取得 FIFO 的状态标志**

函数原型：

Visual C++:

```
BOOL GetDevStatusProAD ( HANDLE hDevice,
                         PPCI9757_STATUS_AD pADStatus)
```

Visual Basic:

```
Declare Function GetDevStatusProAD Lib"PCI9757_32" ( _
                                    ByVal hDevice As Long, _
                                    ByRef pADStatus As PCI9757_STATUS_AD) As Boolean
```

LabVIEW:

请参考相关演示程序。

功能：一旦用户使用 StartDeviceProAD后，应立即用此函数查询FIFO存储器的状态（半满标志、非空标志、溢出标志）。我们通常用半满标志去同步半满读操作。当半满标志有效时，再紧接着用 ReadDeviceProAD_Half读取FIFO中的半满有效AD数据。

参数：

hDevice设备对象句柄，它应由 CreateDevice或 CreateDeviceEx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（PCI9757_STATUS_AD）》章节。

返回值：若调用成功则返回TRUE，否则返回FALSE，用户可以调用 GetLastErrorEx函数取得当前错误码。若用户选择半满查询方式读取AD数据，则当 GetDevStatusProAD函数取得的 bHalf等于TRUE，应立即调用 ReadDeviceProAD_Half读取FIFO中的半满数据。否则用户应继续循环轮询FIFO半满状态，直到有效为止。注意在循环轮询期间，可以用Sleep函数抛出一定时间给其他应用程序(包括本应用程序的主程序和其他子线程)，以提高系统的整体数据处理效率。

其使用方法请参考本文档的《高速大容量、连续不间断数据采集及存盘技术详解》章节。

相关函数： CreateDevice, SetDevFrequencyAD, InitDeviceProAD, StartDeviceProAD, ReadDeviceProAD_Npt, GetDevStatusProAD, ReadDeviceProAD_Half, StopDeviceProAD, ReleaseDeviceProAD, ReleaseDevice

◆ **当 FIFO 半满信号有效时，批量读取 AD 数据**

函数原型:

Visual C++:

```
BOOL ReadDeviceProAD_Half( HANDLE hDevice,
                           LONG ADBuffer[],
                           LONG nReadSizeWords,
                           PLONG nRetSizeWords)
```

Visual Basic:

```
 Declare Function ReadDeviceProAD_Half Lib "PCI9757_32" ( _
                                    ByVal hDevice As Long, _
                                    ByRef ADBuffer As Long, _
                                    ByVal nReadSizeWords As Long, _
                                    ByRef nRetSizeWords As Long) As Boolean
```

LabVIEW:

请参考相关演示程序。

功能：一旦用户使用 GetDevStatusProAD后取得的FIFO状态 bHalf等于TRUE(即半满状态有效)时，应立即用此函数读取设备上FIFO中的半满AD数据。

参数：

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

ADBuffer 接受AD数据的用户缓冲区，通常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》。

nReadSizeWords 指定一次 ReadDeviceProAD_Half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，而且应等于FIFO总容量的二分之一(如果用户有特殊需要可以小于FIFO的二分之一长)。比如设备上配置了 1K FIFO，即 1024 字，那么这个参数应指定为 512 或小于 512。

nRetSizeWords 返回实际读取的点数(或字数)。

返回值：如果成功的读取由nReadSizeWords参数指定量的AD数据到用户缓冲区，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。

其使用方法请参考本部分第十章 《高速大容量、连续不间断数据采集及存盘技术详解》。

相关函数： CreateDevice SetDevFrequencyAD InitDeviceProAD StartDeviceProAD ReadDeviceProAD_Npt GetDevStatusProAD ReadDeviceProAD_Half StopDeviceProAD ReleaseDeviceProAD ReleaseDevice

♦ 暂停 AD 设备

函数原型：

Visual C++:

```
BOOL StopDeviceProAD ( HANDLE hDevice )
```

Visual Basic:

```
Declare Function StopDeviceProAD Lib "PCI9757_32" ( _
                                        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 StartDeviceProAD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 StartDeviceProAD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或 CreateDeviceEx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 InitDeviceProAD StartDeviceProAD ReadDeviceProAD_Npt GetDevStatusProAD ReadDeviceProAD_Half StopDeviceProAD ReleaseDeviceProAD ReleaseDevice

♦ 释放设备上的 AD 部件

函数原型：

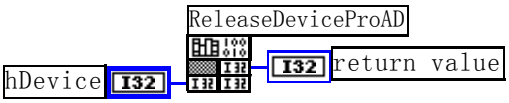
Visual C++:

```
BOOL ReleaseDeviceProAD(HANDLE hDevice)
```

Visual Basic:

```
Declare Function ReleaseDeviceProAD Lib "PCI9757_32" ( _
                                        ByVal hDevice As Long) As Boolean
```

LabVIEW:



功能：释放设备上的 AD 部件。

参数：hDevice设备对象句柄，它应由 CreateDevice或 CreateDeviceEx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，InitDeviceProAD 必须和 ReleaseDeviceProAD 函数一一对应，即当您执行了一次InitDeviceProAD后，再一次执行这些函数前，必须执行一次 ReleaseDeviceProAD函数，以释放由 InitDeviceProAD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 InitDeviceProAD函数时，那些软硬件资源才可被再次使用。

相关函数： CreateDevice InitDeviceProAD ReleaseDeviceProAD 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 InitDeviceProAD

③ StartDeviceProAD
④ ReadDeviceProAD_Npt
⑤ StopDeviceProAD
⑥ ReleaseDeviceProAD
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 InitDeviceProAD
③ StartDeviceProAD
④ GetDevStatusProAD
⑤ ReadDeviceProAD_Half
⑥ StopDeviceProAD
⑦ ReleaseDeviceProAD
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 直接内存存取 DMA 方式采样操作函数原型说明

（注：函数中的“Dma”字符是 Direct Memory Access 的缩写，标明以直接内存存取方式）

◆ **初始化设备上的 AD 对象**

函数原型:

Visual C++:

```
BOOL InitDeviceDmaAD( HANDLE hDevice,
                      HANDLE hDmaEvent,
                      LONG ADBuffer[ ],
                      LONG nReadSizeWords,
                      LONG nSegmentCount,
                      LONG nSegmentSizeWords,
                      PPCI9757_PARA_AD pADPara)
```

Visual Basic:

```
Declare Function InitDeviceDmaAD Lib"PCI9757_32" ( _
                                    ByVal hDevice As Long, _
                                    ByVal hDmaEvent As Long, _
                                    ByRef ADBuffer As Long, _
                                    ByVal nReadSizeWords As Long, _
                                    ByVal nSegmentCount As Long, _
                                    ByVal nSegmentSizeWords As Long, _
                                    ByRef pADPara As PCI9757_PARA_AD) As Boolean
```

LabVIEW:

请参考相关演示程序。

功能：它负责初始化设备对象中的AD部件，为设备操作及DMA传输就绪有关工作，如预置AD采集通道、采样频率等。且让设备上的AD部件以硬件DMA的方式工作，但它并不启动AD采样，而是需要在此函数被成功调用之后，再调用 StartDeviceDmaAD函数即可启动AD采样。

参数：

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

hDmaEvent DMA事件对象句柄，它应由 CreateSystemEvent函数创建。它被创建时是一个不发信号且自动复位的内核系统事件对象。当硬件每次DMA完一个指定段长(nSegmentSizeWords)的数据时这个内核系统事件被触发一次。用户应在数据采集子线程中使用WaitForSingleObject这个Win32 函数来接管这个内核系统事件。当该事件没有到来时，WaitForSingleObject将使所在线程进入睡眠状态，此时，它不同于程序轮询方式，因为它并不消耗CPU时间。当hDmaEvent事件被触发成发信号状态，那么WaitForSingleObject将复位该内核系统事件对象，使其处于不发信号状态，并立即唤醒所在线程，继而执行WaitForSingleObject其后的代码，比如移走ADBuffer中的数据、分析数据、显示数据等，待处理完数据后再循环调用WaitForSingleObject，让所在线程再

次进入睡眠状态，重复以上过程。所以利用DMA方式采集数据，不仅等待AD转换指定数据不需要消耗CPU时间，同时将AD数据从卡上传输到计算机主存更是不需要花消CPU时间，其效率是最高的。其具体实现方法请参考《高速大容量、连续不间断数据采集及存盘技术详解》。

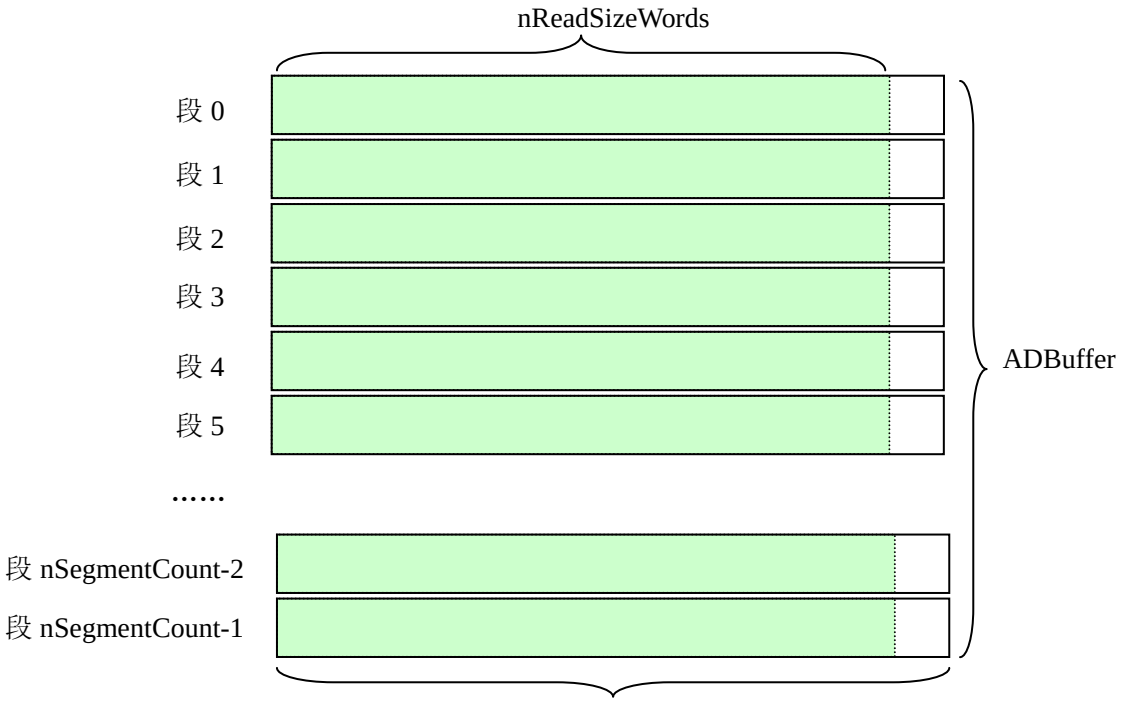
ADBuffer 接受AD数据的用户缓冲区，可以是一个相应类型的足够大的数组，也可以是用户使用内存分配函数分配的内存空间。关于如何将缓冲区中的这些AD数据转换成相应的电压值，请参考请第六章《数据格式转换与排列规则》。注意该缓冲区最好定义为两维缓冲或数组，以便DMA数据传输和缓冲区数据处理分时错开，以更好的达到AD转换、传输、处理等过程的并行工作。**注意：该缓冲区的生命周期必须跨越DMA的整个操作周期，我建议最好将期置为全局缓冲区，即整个应用程序的生命周期内存在。否则，可能会造成严重的存储区访问违反。**

nReadSizeWords 在每个段缓冲中应 DMA 填充和用户读走的数据点数。它的取值范围不应小于 1，同时，不能大于段长 nSegmentSizeWords，其具体取值应根据采样通道数来确定其大小，通常应在段长范围内，取用为采样通道数整数倍长，同时又最接近段长的读取长度来设置本参数。也就是说每当用户接受到 hDmaEvent 事件后，对相应段缓冲区作数据处理时只能从该段缓冲首单元开始往后共处理 nReadSizeWords 个数据采样点。

nSegmentCount 缓冲区段数。其取值范围为[2-128]。为了提高整体效率和性能，将用户缓冲区人为的划分为若干段，让 DMA 分段传输整个数据序列，以便用户能够实时并发的处理。而每段的长度由 nSegmentSizeWords 参数决定。

nSegmentSizeWords 缓冲区各段的长度(字或点)。其取值范围应等于或小于板载 FIFO 的半满空间。而段数由 nSegmentCount 决定。

pADPara 设备对象参数结构PCI9757_PARA_AD的指针， 它的各成员值决定了设备上的AD对象的各种状态及工作方式，如AD采样通道、采样频率等。具体定义请参考PCI9757.h(.Bas或.Pas或.VI)驱动接口文件和本文档中的《硬件参数结构》章节。



DMA 缓冲区结构图

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。

备注：DMA是直接内存存取的意思，其英文定义为：Direct Memory Access。它的技术含义可以顾名思义，就是数据传输在设备和内存之间直接进行，无需要CPU的参与。该项技术的使用大大提高了数据实时采集和处理的效率。但是为了更好的配合这样好的机制，我们需要将用户缓冲区分段，比如分为 32 段，每段的长度等于FIFO半满长度 4096，因此可以定义一个两维数组。如：SHORT ADBuffer[32][4096]，即nSegmentCount=32，nSegmentSizeWords=4096，然后开始启动设备后，ADBuffer[0]首先被DMA占用，当传输完成后，hDmaEvent即被触发，用户即可处理ADBuffer[0]，而DMA接着占用ADBuffer[1]，当传输完成后，hDmaEvent即再次被触发，用户即可处理ADBuffer[1]，而DMA接着占用ADBuffer[2]，就这样依次类推。至到ADBuffer[31]被传输完

后DMA再回到始端，占用ADBuffer[0]，就这样周而复始的进行下去。除了hDmaEvent事件对象可以通知用户何时处理数据外，其 GetDevStatusDmaAD函数也可以实时返回DMA各种状态，如DMA正在占用的缓冲段ID（iCurSegmentID），整个缓冲链各个段的更新状态(bSegmentSts[])，整个缓冲链是否溢出(bBufferOverflow)等，跟踪这些信息，可以使数据转换、传输和处理之间有更大的时间弹性，高度保证数据的连续性。

切记：在 InitDeviceDmaAD 函数被调用之后若想再调用它改变硬件的某些参数，那么必须在ReleaseDeviceDmaAD之后方可调用。即 InitDeviceDmaAD和 ReleaseDeviceDmaAD必须成对调用，且在应用程序被关闭前必须确保已调用 ReleaseDeviceDmaAD释放了各种DMA资源，否则可能会引起系统严重错误。

相关函数： CreateDevice InitDeviceDmaAD StartDeviceDmaAD
GetDevStatusDmaAD SetDevStatusDmaAD StopDeviceDmaAD
ReleaseDeviceDmaAD ReleaseDevice

◆ **启动设备上的 AD 部件**

函数原型：

Visual C++:

```
BOOL StartDeviceDmaAD(HANDLE hDevice)
```

Visual Basic:

```
Declare Function StartDeviceDmaAD Lib"PCI9757_32" ( _
                                        ByVal hDevice As Long) As Boolean
```

LabVIEW:

请参考相关演示程序。

功能：在 InitDeviceDmaAD被成功调用之后，调用此函数即可启动设备上的AD部件，让设备开始AD采样。

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

返回值：若成功，则返回TRUE，意味着AD被启动，否则返回FALSE，用户可以用 GetLastErrorEx捕获错误码。

相关函数： CreateDevice InitDeviceDmaAD StartDeviceDmaAD
GetDevStatusDmaAD SetDevStatusDmaAD StopDeviceDmaAD
ReleaseDeviceDmaAD ReleaseDevice

◆ **取得 DMA 的状态标志**

Visual C++:

```
BOOL GetDevStatusDmaAD ( HANDLE hDevice,
                                PPCI9757_STATUS_DMA pDMAStatus )
```

Visual Basic:

```
Declare Function GetDevStatusDmaAD Lib "PCI9757" (ByVal hDevice As Long,_
                                        ByRef pDMAStatus As PCI9757_STATUS_DMA)As Boolean
```

LabVIEW:

请参考相关演示程序。

功能：一旦用户使用 StartDeviceDmaAD后，应立即用此函数查询DMA的状态（当前段缓冲ID、缓冲段新旧标志、DMA缓冲溢出标志）。我们通常用缓冲段新旧标志bSegmentSts[x]去同步缓冲区数据处理操作。当bSegmentSts[x]标志为 1 时表示其该段为新数据段，则可以处理x段数据，然后再执行 SetDevStatusDmaAD函数将x段新旧标志置为 0，表示已处理完，该段变为旧数据。

参数：

hDevice设备对象句柄，它应由 CreateDevice创建。

pDMAStatus它属于PCI9757_STATUS_DMA的结构体指针。该参数实时返回DMA的当前状态。关于PCI9757_STATUS_DMA具体定义请参考PCI9757.h(.Bas或.Pas或.VI)驱动接口文件以及本文档中的《DMA状态参数结构（PCI9757_STATUS_DMA）》。

返回值：若调用成功则返回TRUE，否则返回FALSE，用户可以调用 GetLastErrorEx函数取得当前错误码。

相关函数： CreateDevice InitDeviceDmaAD StartDeviceDmaAD
GetDevStatusDmaAD SetDevStatusDmaAD StopDeviceDmaAD
ReleaseDeviceDmaAD ReleaseDevice

◆ **取得 DMA 的状态标志**

函数原型：

Visual C++:

```
BOOL SetDevStatusDmaAD ( HANDLE hDevice,
                         LONG iClrBufferID)
```

Visual Basic:

```
Declare Function SetDevStatusDmaAD Lib"PCI9757_32" ( _
                                   ByVal hDevice As Long, _
                                   ByVal iClrBufferID As Long) As Boolean
```

LabVIEW:

请参考相关演示程序。

功能：当处理完 DMA 缓冲链中的某一段数据后，应该立即调用此函数将其缓冲段状态标志清除，使其复位至 0，表示该数据已被处理过，已变成了旧数据，以便在下一个 DMA 事件响应下，不会重复自理某一缓冲段的数据。同时也避免产生 DMA 缓冲区溢出的可能。

参数：

hDevice设备对象句柄，它应由 CreateDevice。

iClrBufferID 要被清除标志的缓冲段ID。当指定的缓冲段状态标志清除后，则从 GetDevStatusDmaAD函数返回的bSegmentSts[x]则会为 0。只有待到DMA事件下，其相应的缓冲段状态标志才会被置 1。

返回值：若调用成功则返回TRUE，否则返回FALSE，用户可以调用 GetLastErrorEx函数取得当前错误码。

相关函数： CreateDevice　InitDeviceDmaAD　StartDeviceDmaAD　GetDevStatusDmaAD　SetDevStatusDmaAD　StopDeviceDmaAD　ReleaseDeviceDmaAD　ReleaseDevice

◆ **暂停设备上的 AD 采样工作**

函数原型：

Visual C++:

```
BOOL StopDeviceDmaAD(HANDLE hDevice)
```

Visual Basic:

```
Declare Function StopDeviceDmaAD Lib"PCI9757_32" ( _
                                 ByVal hDevice As Long) As Boolean
```

LabVIEW:

请参考相关演示程序。

功能：在 StartDeviceDmaAD被成功调用之后，用户可以在任何时候调用此函数停止AD采样(必须在ReleaseDeviceDmaAD 之间被调用)，注意它不改变设备的其它任何状态。如果过后用户再调用StartDeviceDmaAD，那么设备会接着停止前的状态(如通道位置)继续开始正常的AD数据转换。

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

返回值：若成功，则返回TRUE，意味着AD被停止，否则返回FALSE，用户可以用 GetLastErrorEx捕获错误码。

相关函数： CreateDevice　InitDeviceDmaAD　StartDeviceDmaAD　GetDevStatusDmaAD　SetDevStatusDmaAD　StopDeviceDmaAD　ReleaseDeviceDmaAD　ReleaseDevice

◆ **释放设备上的 AD 部件**

函数原型：

Visual C++:

```
BOOL ReleaseDeviceDmaAD(HANDLE hDevice)
```

Visual Basic:

```
Declare Function ReleaseDeviceDmaAD Lib "PCI9757_32" ( _
                                    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，如果AD没有被 StopDeviceDmaAD函数停止，则此函数在释放AD部件之前先停止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。
相关函数： CreateDevice InitDeviceDmaAD StartDeviceDmaAD
GetDevStatusDmaAD SetDevStatusDmaAD StopDeviceDmaAD
ReleaseDeviceDmaAD ReleaseDevice

应注意的是，InitDeviceDmaAD必须和 ReleaseDeviceDmaAD函数一一对应，即当您执行了一次InitDeviceDmaAD后，再一次执行这些函数前，必须执行一次 ReleaseDeviceDmaAD函数，以释放先前由InitDeviceDmaAD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InitDeviceDmaAD函数时，那些软硬件资源才可被再次使用。

◆ **函数一般调用顺序**
① CreateDevice
② CreateSystemEvent(公共函数)
③ InitDeviceDmaAD
④ StartDeviceDmaAD
⑤ WaitForSingleObject(WIN32 API 函数，详细说明请参考 MSDN 文档)
⑥ GetDevStatusDmaAD
⑦ SetDevStatusDmaAD
⑧ StopDeviceDmaAD
⑨ ReleaseDeviceDmaAD
⑩ ReleaseSystemEvent（公共函数）
⑩ ReleaseDevice
注明：用户可以反复执行第⑤⑥⑦步，以实现高速连续不间断大容量采集。
关于这个过程的图形说明请参考《使用纲要》。
注意：若成功初始化 DMA 后，要退出整个应用程序，切记应先释放 DMA 才能退出。

第五节、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,
                PPCI9757_PARA_AD pADPara)
```
Visual Basic:
```
Declare Function LoadParaAD Lib"PCI9757_32" (ByVal hDevice As Long,_
                                             ByRef pADPara As PCI9757_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PPCI9757_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PPCI9757_PARA_AD请参考PCI9757.h或PCI9757.Bas或PCI9757.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：
Viusal C++:
```
BOOL SaveParaAD (HANDLE hDevice,
                 PPCI9757_PARA_AD pADPara)
```
Visual Basic:

```
Declare Function SaveParaAD Lib"PCI9757_32" (ByVal hDevice As Long,_
                                    ByRef pADPara As PCI9757_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。
pADPara设备硬件参数，关于PCI9757_PARA_AD的详细介绍请参考PCI9757.h或PCI9757.Bas或PCI9757.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

♦ AD 采样参数复位至出厂默认值函数

函数原型：
Viusal C++ :

```
BOOL ResetParaAD (HANDLE hDevice,
                  PPCI9757_PARA_AD pADPara)
```

Visual Basic:

```
Declare Function ResetParaAD Lib"PCI9757_32" (ByVal hDevice As Long,_
                                    ByRef pADPara As PCI9757_PARA_AD) As Boolean
```

LabVIEW:
请参考相关演示程序。

功能：将系统中原来的 AD 参数值复位至出厂时的默认值。以防用户不小心将各参数设置错误造成一时无法确定错误原因的后果。
参数：
hDevice设备对象句柄，它应由 CreateDevice创建。
pADPara设备硬件参数，它负责在参数被复位后返回其复位后的值。关于PCI9757_PARA_AD的详细介绍请参考PCI9757.h或PCI9757.Bas或PCI9757.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
ResetParaAD　ReleaseDevice

注意：在您编写工程应用软件时，若要更方便的保存和读取您特有的软件参数，请不防使用我们为您提供的辅助函数：SaveParaInt、 LoadParaInt、SaveParaString、LoadParaString，详细说明请参考共用函数介绍章节中的《各种参数保存和读取函数原型说明》。

第四章 硬件参数结构

第一节、AD 硬件参数结构（PCI9757_PARA_AD）

Visual C++：

```
typedef struct _PCI9757_PARA_AD
{
    LONG bChannelArray[4];    // 采样通道选择阵列，分别控制 4 个通道，=TRUE 表示该通道采样，否则不采样
    LONG InputRange[4];       // 模拟量输入量程选择阵列，分别控制 4 个通道
    LONG Frequency;           // 采集频率，单位为 Hz, [3, 800000]
    LONG TriggerMode;         // 触发模式选择
    LONG TriggerSource;       // 触发源选择
    LONG TriggerType;         // 触发类型选择(边沿触发/脉冲触发)
    LONG TriggerDir;          // 触发方向选择(正向/负向触发)
    LONG TrigLevelVolt;       // 触发电平(0—10000mV)
    LONG TrigWindow;          // 触发灵敏窗[1, 65535]，单位 25 纳秒
    LONG ClockSource;         // 时钟源选择(内/外时钟源)
} PCI9757_PARA_AD, *PPCI9757_PARA_AD;
```

Visual Basic：

```
Type PCI9757_PARA_AD
    bChannelArray(0 To 3) As Long   ' 采样通道选择阵列，分别控制 8 个通道，=TRUE 表示该通道采样，否则不采样
    InputRange(0 To 3) As Long      ' 模拟量输入量程选择
    Frequency As Long               ' 采集频率，单位为 Hz
    TriggerMode As Long             ' 触发模式选择
    TriggerSource As Long           ' 触发源选择
    TriggerType As Long             ' 触发类型选择(边沿触发/脉冲触发)
    TriggerDir As Long              ' 触发方向选择(正向/负向触发)
    TrigLevelVolt  As Long          ' 触发电平(0—10000mV)
    TrigWindow  As Long             ' 触发灵敏窗[1, 65535]，单位 25 纳秒
    ClockSource As Long             ' 时钟源选择(内/外时钟源)
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、DMA 等将与 PCI 设备的用户永远告别，一句话 PCI 设备是一种更易于管理和使用的设备。

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 InitDeviceProAD或InitDeviceDmaAD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。

bChannelArray　AD 采样通道选择阵列，分别控制 4 个通道，=TRUE 表示该通道采样，否则不采样。

InputRange 模拟量输入量程选择阵列，分别控制 4 个通道。取值如下表：

常量名	常量值	功能定义
PCI9757_INPUT_N10000_P10000mV	0x00	±10000mV
PCI9757_INPUT_N5000_P5000mV	0x01	±5000mV
PCI9757_INPUT_N2500_P2500mV	0x02	±2500mV
PCI9757_INPUT_0_P10000mV	0x04	0～10000mV
PCI9757_INPUT_0_P5000mV	0x05	0～5000mV

关于各个量程下采集的数据ADBuffer[]如何换算成相应的电压值，请参考《AD原码LSB数据转换成电压值的换算方法》章节。

Frequency　AD 采样频率，本设备的频率取值范围为[3, 800KHz]。

TriggerMode　AD 触发模式。

常量名	常量值	功能定义
PCI9757_TRIGMODE_SOFT	0x00	软件触发(属于内触发)
PCI9757_TRIGMODE_POST	0x01	硬件后触发(属于外触发)

TriggerSource　AD 触发源。

常量名	常量值	功能定义
PCI9757_TRIGSRC_ATR	0x00	选择外部 ATR 作为触发源
PCI9757_TRIGSRC_DTR	0x01	选择外部 DTR 作为触发源
PCI9757_TRIGSRC_PCI_TRIG0	0x02	CONVST_IN 信号触发源

TriggerType　AD 触发类型。

常量名	常量值	功能定义
PCI9757_TRIGTYPE_EDGE	0x00	边沿触发
PCI9757_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
PCI9757_TRIGDIR_NEGATIVE	0x00	负向触发(低脉冲/下降沿触发)
PCI9757_TRIGDIR_POSITIVE	0x01	正向触发(高脉冲/上升沿触发)
PCI9757_TRIGDIR_POSIT_NEGAT	0x02	正负方向均有效

注明：PCI9757_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。

TrigLevelVolt 触发电平，取值范围为(0—10000mV)。
TrigWindow　触发灵敏窗时间值，取值范围为[1, 65535], 单位 25 纳秒。
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
PCI9757_CLOCKSRC_IN	0x00	内部时钟定时触发
PCI9757_CLOCKSRC_OUT	0x01	外部时钟定时触发

相关函数： CreateDevice　LoadParaAD　SaveParaAD　ReleaseDevice

设备地址获取函数 GetDeviceAddr 的参数 RegisterID 所使用的选项(有效部分)

常量名	常量值	功能定义
PCI9757_REG_MEM_PLXCHIP	0x00	0 号寄存器对应 PLX 芯片所使用的内存模式基地址(使用 LinearAddr)
PCI9757_REG_IO_PLXCHIP	0x01	1 号寄存器对应 PLX 芯片所使用的 IO 模式基地址(使用 PhysAddr)
PCI9757_REG_IO_CPLD	0x02	2 号寄存器对应板上控制单元所使用的 IO 模式基地址(使用 PhysAddr)

第二节、AD 状态参数结构（PCI9757_STATUS_AD）

Visual C++：

```
typedef struct _PCI9757_STATUS_AD
{
    LONG bNotEmpty;            // 板载 FIFO 存储器的非空标志，=TRUE 非空，  = FALSE 空
    LONG bHalf;                // 板载 FIFO 存储器的半满标志，=TRUE 半满以上，  = FALSE 半满以下
    LONG bDynamic_Overflow; // 板载 FIFO 存储器的动态溢出标志，= TRUE 已发生溢出，= FALSE  未发生溢出
    LONG bStatic_Overflow;  // 板载 FIFO 存储器的静态溢出标志，= TRUE 已发生溢出，  = FALSE  未发生溢出
```

```
    LONG bConverting;          // AD 是否正在转换， =TRUE:表示正在转换， =FALS 表示转换完成
    LONG bTriggerFlag;         // 触发标志, =TRUE 表示触发事件发生， =FALSE 表示触发事件未发生
    LONG nTriggerPos;          // 触发位置
} PCI9757_STATUS_AD, *PPCI9757_STATUS_AD;
```

Visual Basic：

```
Type PCI9757_STATUS_AD
    bNotEmpty As Long            ' 板载 FIFO 存储器的非空标志，=TRUE 非空， = FALSE 空
    bHalf As Long                ' 板载 FIFO 存储器的半满标志，=TRUE 半满以上， = FALSE 半满以下
    bDynamic_Overflow As Long ' 板载 FIFO 存储器的动态溢出标志,= TRUE 已发生溢出，= FALSE 未发生溢出
    bStatic_Overflow As Long   ' 板载 FIFO 存储器的静态溢出标志，= TRUE 已发生溢出， = FALSE 未发生溢出
    bConverting As Long          'AD 是否正在转换， =TRUE:表示正在转换， =FALS 表示转换完成
    bTriggerFlag As Long         ' 触发标志, =TRUE 表示触发事件发生， =FALSE 表示触发事件未发生
    nTriggerPos As Long          ' 触发位置

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，GetDevStatusProAD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。

bNotEmpty AD 板载存储器 FIFO 的非空标志，=TRUE 表示存储器处在非空状态，即有可读数据，否则表示空。

bHalf AD 板载存储器 FIFO 的半满标志，=TRUE 表示存储器处在半满状态，即有至少有半满以上数据可读，否则表示在半满以下，可能有小于半满的数据可读。

bDynamic_Overflow AD 板载存储器 FIFO 的溢出标志，=TRUE 表示存储器处在全满或溢出状态，即全满的数据可读数据，但此时的数据很有可能已有丢点现象。否则表示满以下状态。该状态处于动态溢出状态，即 FIFO 随时溢出，它随时=TRUE，而随时不溢出，则随时=FALSE。

bStatic_Overflow AD 板载存储器 FIFO 的溢出标志，=TRUE 表示存储器至少有过一次出现全满或溢出状态，然后永远为 TRUE，除非用户重新开始采集数据则会自动变为 FALSE。在启动采集过程中，只有一次全满或溢出状态都未发生过，则此标志恒等于 FALSE。所以用此标志可以确定在整过采集中是否有过溢出丢点现象。当然要避免丢点现象的发生，您需要考虑应用软件设计的合理性、效率性等各方因素，我们提供的高级演示程序（尤其是 VC）便很好的展示了此类思想。

bConverting AD 是否正在转换， =TRUE:表示正在转换， =FALS 表示转换完成。

bTriggerFlag AD 触发标志，=TRUE 表示已被触发(即产生触发事件)，=FALSE 表示未产生触发事件。

nTriggerPos AD 触发位置。

相关函数： CreateDevice GetDevStatusProAD ReleaseDevice

第三节、DMA 状态参数结构（PCI9757_STATUS_DMA）

```
const int MAX_SEGMENT_COUNT = 128;
```

Visual C++：

```
typedef struct _PCI9757_STATUS_DMA
{
    LONG iCurSegmentID;   // 当前段缓冲 ID,表示 DMA 正在传输的缓冲区段
    LONG bSegmentSts[MAX_SEGMENT_COUNT];
        // 各个缓冲区的新旧状态,=1 表示该相应缓冲区数据为新,否则为旧
    LONG bBufferOverflow;     // 返回溢出状态,=1 表示溢出，否则未溢出
} PCI9757_STATUS_DMA, *PPCI9757_STATUS_DMA;
```

Visual Basic：

```
    Type PCI9757_STATUS_DMA
    iCurSegmentID As Long     ' 当前段缓冲 ID,表示 DMA 正在传输的缓冲区段
    bSegmentSts(0 To MAX_SEGMENT_COUNT - 1) As Long
                              ' 各个缓冲区的新旧状态,=1 表示该相应缓冲区数据为新,否则为旧
    bBufferOverflow As Long   ' 返回溢出状态
End Type
```

LabVIEW:

请参考相关演示程序。

此结构体主要用于DMA传输时的状态监控，GetDevStatusDmaAD函数使用此结构体来实时取得DMA状态，以便同步各种数据处理过程。

iCurSegmentID　DMA正在传输的当前缓冲段ID号。该ID号返回值的最大范围为 0 至 63，但其具体的返回值范围为 InitDeviceDmaAD中的nSegmentCount参数决定，它的返回值为 0 至nSegmentCount-1。注意，每次调用 InitDeviceDmaAD初始化设备后，其值自动被复位至 0。

bSegmentSts[ ]　DMA缓冲区各段的状态。 如bSegmentSts[0]=0，表示缓冲区段 0 此时为旧数据段，若=1 则段 0 为新数据段，可以对其进行数据处理。同理，bSegmentSts[1]=0，表示缓冲区段 1 此时为旧数据段，若=1 则段 1 为新数据段，可以对其进行数据处理。注意，每次调用 InitDeviceDmaAD初始化设备后，其值自动被复位至 0。

bBufferOverflow　组缓冲区溢出标志。若等于 0，则表示整个DMA缓冲链未发生溢出，若等于 1，则表示整个DMA缓冲链已发生溢出。注意，每次调用 InitDeviceDmaAD初始化设备后，其值自动被复位至 0。

相关函数： CreateDevice　LoadParaAD　SaveParaAD
ResetParaAD　ReleaseDevice

第五章 各种功能的使用方法

第一节、AD 触发功能的使用方法

一、AD软件触发（亦叫内触发）

在初始化AD时，若AD硬件参数ADPara. TriggerSource = PCI9757_TRIGMODE_SOFT时，则可实现软件触发采集。在软件触发采集功能下，调用StartDeviceProAD (或StartDeviceDmaAD)函数启动AD时，AD即刻进入转换过程，不等待其他任何外部硬件条件。也可理解为内触发。

具体过程请参考以下图例，图中AD工作脉冲的周期由设定的采样频率(Frequency)决定。AD启动脉冲由软件接口函数StartDeviceProAD (或StartDeviceDmaAD)产生。

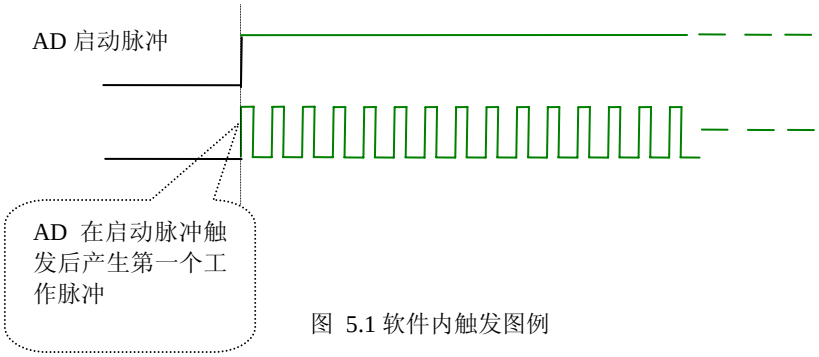


图 5.1 软件内触发图例

二、AD硬件后触发(亦叫外触发)

在初始化AD时，若AD硬件参数ADPara. TriggerSource = PCI9757_TRIGMODE_POST时，则可实现硬件后触发采集。在硬件后触发采集功能下，调用StartDeviceProAD (或StartDeviceDmaAD)函数启动AD时，AD并不立即进入转换过程，而是要等待外部硬件触发源信号符合指定条件后才开始转换AD数据，也可理解为硬件外触发。其外部硬件触发源信号由CN1中的DTR管脚输入提供。关于在什么条件下触发AD，由用户选择的触发类型(TriggerType)、 触发沿方向(TriggerDir)共同决定。

（1）、AD边沿触发功能

边沿触发就是捕获触发源信号变化特征来触发AD转换。

当TriggerType = PCI9757_TRIGTYPE_EDGE时，即为边沿触发。具体实现如下：

ADPara.TriggerDir = PCI9757_TRIGDIR_NEGATIVE时，即选择触发方向为下边沿触发。即当DTR触发源信号出现一瞬态的下边沿信号时产生触发事件，AD即刻进入触发工作状态，其后续变化对AD采集无影响。

ADPara.TriggerDir = PCI9757_TRIGDIR_POSITIVE时，即选择触发方向为上边沿触发。它与下边沿触发的方向相反以外，其他方面同理。

ADPara.TriggerDir = PCI9757_TRIGDIR_POSIT_NEGAT时，即选择触发方向为上边沿或下边沿触发。它的特点是只要DTR出现任何边沿的瞬态跳变时产生触发事件。AD即刻进入触发工作状态，其后续变化对AD采集无影响。此项功能可应用在只要外界的某一信号变化时就采集的场合。

具体过程请参考以下图例，图中AD工作脉冲的周期由设定的采样频率(Frequency)决定。AD启动脉冲由软件接口函数StartDeviceProAD (或StartDeviceDmaAD)产生。

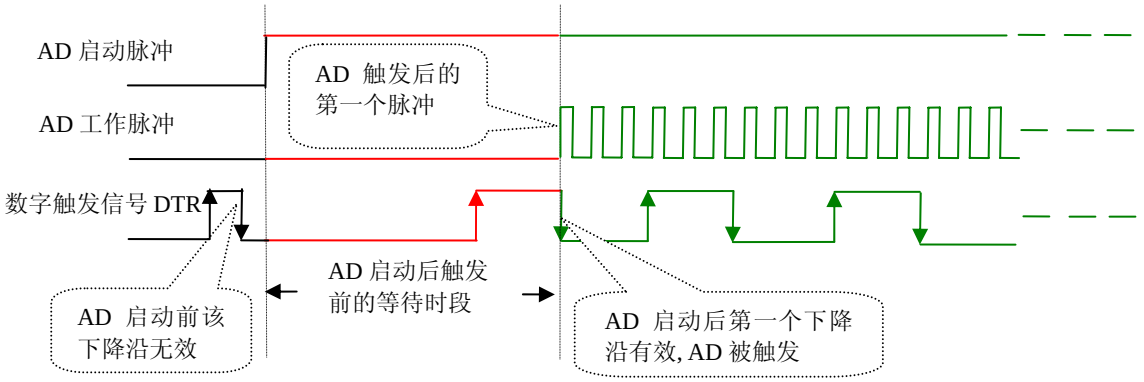


图 5.2 下降沿触发图例

（2）、脉冲电平触发功能

脉冲电平触发就是捕获触发源信号相对于触发电平的信号以上位置或以下位置作为条件来触发AD转换。说得简单点，就是利用模拟比较器的输出Result的正脉冲或负脉冲作为触发条件。该功能可以应用在地震波、馒头波等信号的有效部分采集。

当ADPara.TriggerType = PCI9757_TRIGTYPE_PULSE即选择了脉冲电平触发功能。

ADPara.TriggerDir = PCI9757_TRIGDIR_NEGATIVE（负向触发）时，若模拟触发源一旦小于触发电平时AD触发采集，一旦触发源大于触发电平时自动停止采集，当再小于时接着采集，即只采集位于触发电平下端的波形。如下图5.3。

ADPara.TriggerDir = PCI9757_TRIGDIR_POSITIVE（正向触发）时，若模拟触发源一旦大于触发电平时AD触发采集，一旦触发源小于触发电平时自动停止采集，当再大于时接着采集，即只采集位于触发电平上端的波形。

当ADPara.TriggerDir = PCI9757_TRIGDIR_POSIT_NEGAT时，即选择触发方向为正脉冲或负脉冲触发。它的特点是不管是正脉冲或负脉冲都触发。此时它与内部软件触发同理。

具体过程请参考以下图例，图中AD工作脉冲的周期由设定的采样频率(Frequency)决定。AD启动脉冲由软件接口函数StartDeviceProAD (或StartDeviceDmaAD)产生。

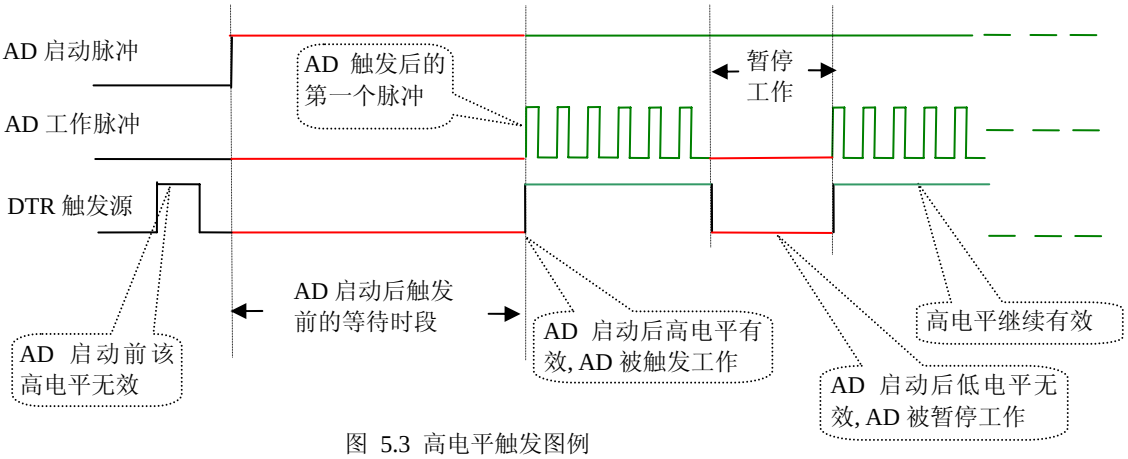


图 5.3 高电平触发图例

第二节、AD 内时钟与外时钟功能的使用方法

一、AD 内时钟功能

内时钟功能是指使用板载时钟振荡器经板载逻辑控制电路根据用户指定的分频数分频后产生的时钟信号去触发AD定时转换。要使用内时钟功能应在软件中置硬件参数ADPara.ClockSouce = PCI9757_CLOCKSRC_IN。该时钟的频率在软件中由硬件参数ADPara.Frequency决定。如Frequency = 100000，则表示AD以100000Hz的频率工作（即100KHz，10微秒/点）。

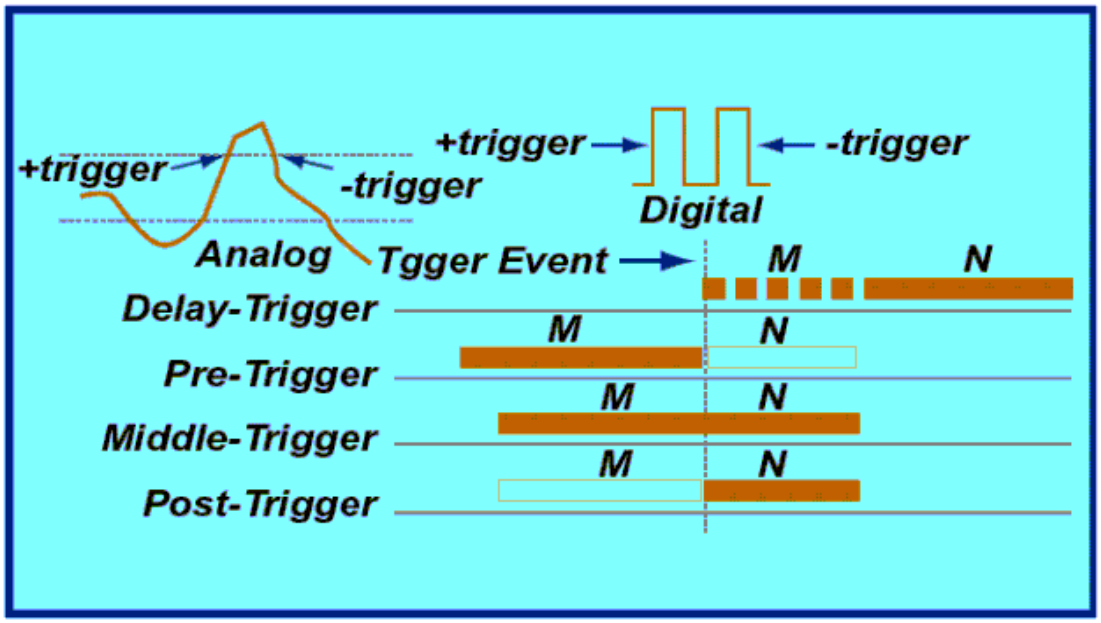
二、AD 外时钟功能

外时钟功能是指使用板外的时钟信号来定时触发AD进行转换。该时钟信号由连接器的CLKIN脚输入提供。板外的时钟可以是另外一块PCI9757的时钟输出由连接器的CLKOUT脚提供，也可以是其他设备如时钟频率发生器等。要使用外时钟功能应在软件中置硬件参数ADPara.ClockSouce = PCI9757_CLOCKSRC_OUT。该时钟的频率主要取决于外时钟的频率，而板内时钟的频率（即硬件参数ADPara.Frequency决定的频率）只有在分组采集模式下有一定作用外，其整个AD采样频率完全受控于外时钟频率。

第三节、AD 触发功能的变通与扩展

如下图所示，您可能需要观察触发事件(Trigger Event)发生时若干时间以后的某段数据 N（延迟触发 Delay-Trigger）；也可能需要观察触发事件发生以前的一段数据 M（预触发 Pre-Trigger）；还可能需要观察触发事件前一段数据 M 和后一段数据 N（中间触发 Middle-Trigger）；当然也可能只需要观察触发事件紧靠其后的一段数据 N（后触发 Post-Trigger）。而您面对的这些需求通常在示波器产品中才能很好的得以全面实现，但是触发点前后的数据通常要受到板载 RAM 空间的限制。而在一般基于 FIFO 的采样器中可能只是简单的实现的后触发功能(Post-Trigger)，而其他三种功能则无力实现，这是由硬件存储策略决定的，因为基于 FIFO 存储器，它无法像示波器 RAM 存储一样进行循环写入扫描操作。我们基于 FIFO 存储的产品当然也不例外。不过为了更好的满足您的其他三种触发功能的需求，我们想到了一个非常巧妙的变通办法对设备进行了功能扩展，很容易地实现了这些触发功能，而且还突破了板载存储空间的限制，使您在触发事件前后想看到多少数据就能看到多少数据（除触发事件提前以外）。

首先请您关注一下AD状态参数(PCI9757_STATUS_AD)中的 nTriggerPos成员，它返回的就是触发事件在所采集数据序列中的点位置。比如您选择软件内触发，置触发类型为边沿触发，同时置好触发方向、触发电平等参数，然后启动采集并保存数据，此时每采集一个点，nTriggerPos便会自动加 1，直至触发事件产生，此时bTriggerFlag也由 0 变成恒值 1，nTriggerPos也停止加 1，固定值某个值上，假如为 10000，而此时AD继续采集触发事件之后的数据，往后再采集多长数据完全由用户决定，比如总共采集保存了 50000 个点。然后您可以根据 bTriggerFlag是否等于 1 来确定是否产生了触发事件，若等于 1，则表示触发事件产生，nTriggerPos的值则表示触发位置在距起点 10000 的位置上，此时您便可以看到触发事件以前 10000 个点，触发事件之后 40000 个点。试想，这岂不是变通的实现了多种触发功能吗。



另外，对于12位或14位的基于FIFO的AD卡，其最高位D15位均用于表示触发事件是否产生，若等于0则表示未产生触发，若等于1则表示已产生触发事件。

第六章 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 / 65536) * (ADBuffer[0] & 0xFFFF) -10000.00	[-10000, +9999.69]
±5000mV	Volt = (10000.00 / 65536) * (ADBuffer[0] & 0xFFFF) -5000.00	[-5000, +4999.84]
±2500mV	Volt = (5000.00 / 65536) * (ADBuffer[0] & 0xFFFF) -2500.00	[-2500, +2499.92]
0~10000mV	Volt = (10000.00 / 65536) * (ADBuffer[0] & 0xFFFF)	[0，+9999.84]
0~5000mV	Volt = (5000.00 / 65536) * (ADBuffer[0] & 0xFFFF)	[0，+4999.92]

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])&0xFFFF;
Volt = (20000.00/65536) * Lsb -10000.00;
```

Visual Basic：

```
Lsb = (ADBuffer [0]) And &HFFFF
Volt = (20000.00/65536) * 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(假如 FirstChannel=0, LastChannel=1)：

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(假如 FirstChannel=0, LastChannel=3)：

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 VoltBottomRange;          // 量程下限(mV)
    LONG VoltTopRange;             // 量程上限(mV)
    PCI9757_PARA_AD ADPara;        // 保存硬件参数
    LONG CrystalFreq;              // 晶振频率
    LONG HeadEndFlag;              // 头信息结束位
} 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。

第七章 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 ReadDeviceProAD_Npt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]丨**[阿尔泰测控演示系统]**丨**[PCI9757 AD 卡]**丨**[Microsoft Visual C++]**丨**[简易代码演示]**丨**[AD 非空方式]**

二、怎样使用 ReadDeviceProAD_Half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]丨**[阿尔泰测控演示系统]**丨**[PCI9757 AD 卡]**丨**[Microsoft Visual C++]**丨**[简易代码演示]**丨**[AD 半满方式]**

三、怎样使用 DMA 方式取得 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]丨**[阿尔泰测控演示系统]**丨**[Microsoft Visual C++]**丨**[简易代码演示]**丨**[AD DMA 方式]**

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 PCI9757.h 和 DADoc.cpp)。

[程序]丨**[阿尔泰测控演示系统]**丨**[PCI9757 AD 卡]**丨**[Microsoft Visual C++]**丨**[高级代码演示]**

其默认存放路径为：系统盘\ART\PCI9757\SAMPLES\VC\ADVANCED

其他语言的演示可以用上面类似的方法找到。

第八章 高速大容量、连续不间断数据采集及存盘技术详解

与ISA、USB设备同理，使用子线程跟踪AD转换进度，并进行数据采集是保持数据连续不间断的最佳方案。但是与ISA总线设备不同的是，PCI设备在这里不使用动态指针去同步AD转换进度，因为ISA设备环形内存池的

动态指针操作是一种软件化的同步，而PCI设备不再有软件化的同步，而完全由硬件和驱动程序自动完成。这样一来，用户要用程序方式实现连续数据采集，其软件实现就显得极为容易。每次用ReadDeviceProAD_X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_Npt(或者 ReadDeviceProAD_Half)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 InitDeviceProAD或 InitDeviceDmaAD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己被堵塞而没有被处理的缓冲区个数，然后逐一进行处理，最后再从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 8.1 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。

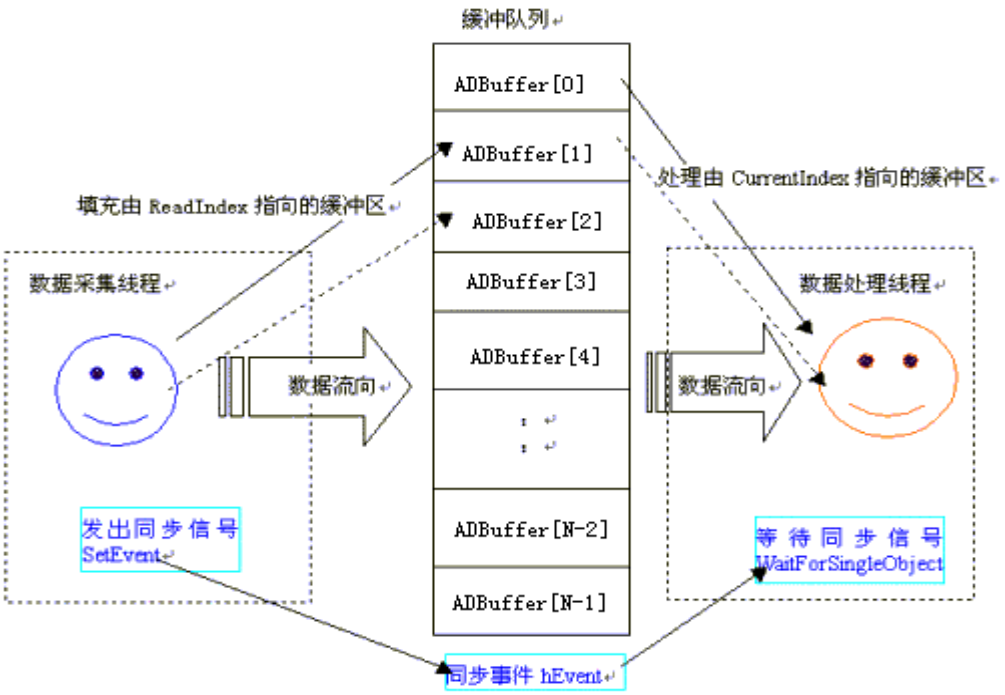


图 8.1

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。

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

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[PCI9757 64 路高速 AD、DIO 卡]丨[Microsoft Visual C++]丨[高级演示程序]

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_Npt (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
```

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

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[PCI9757 64 路高速 AD、DIO 卡]丨[Microsoft Visual C++]丨[高级演示程序]

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_Half (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 非空标志读取 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。而用半满标志，则最多只能达到 FIFO 总容量的二分之一的栈深度，那么用户在两批数据之间处理数据的时间会相对短些，但是半满读取时，查询 AD 转换标志的时间则最少。当然究意那种方案最好，还得看用户的实际需要。

第二节、使用 DMA 方式实现该功能

DMA 方式是利用直接内存存取技术实现的数据传输技术，它基本上不占用 CPU 时间就可能很快的将数据从设备读到用户缓冲区中。所以利用 DMA 方式采集数据，其吞吐率要比程序方式高很多。

需要注意的是，由于DMA方式采用了多缓冲级链的方式，因此每次接受到DMA事件后，一定要注意GetDevStatusDmaAD函数返回的缓冲区状态，必须在该次事件之下，探测所有缓冲区段状态是否为新标志 1，直至所有标志为旧标志 0 后才能允许程序再去接管下一次DMA事件。

其详细应用实例及完整代码请参考 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[PCI9757 64 路高速 AD、DIO 卡]丨[Microsoft Visual C++]丨[高级演示程序]

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_Dma (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
```

第九章 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“PCI9757_”）

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	取得指定的指定设备寄存器组 BAR 地址	
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 创建 Visual Basic 子线程，线程数量可达 32 个以上		
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

◆ **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,
                    PUCHAR pbPCIBar[6])
```

Visual Basic:

```
Declare Function GetDeviceBar Lib "PCI9757_32" ( _
                                    ByVal hDevice As Long, _
                                    ByRef pulPCIBar As Long) As Boolean
```

LabVIEW:

请参考相关演示程序。

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。

pulPCIBar 返回 PCI BAR 所有地址。

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

◆ **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:

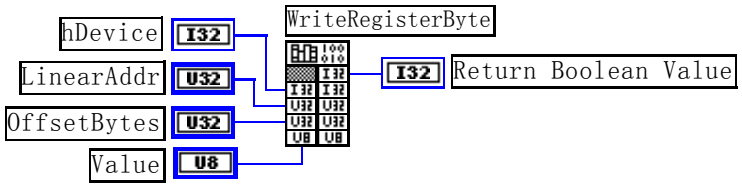
Visual C++:

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

Visual Basic:

```
Declare Function WriteRegisterByte Lib "PCI9757_32" ( _
                                    ByVal hDevice As Long, _
                                    ByVal LinearAddr 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。

pbLinearAddr 指定映射寄存器的线性基地址。

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
      :
```

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:

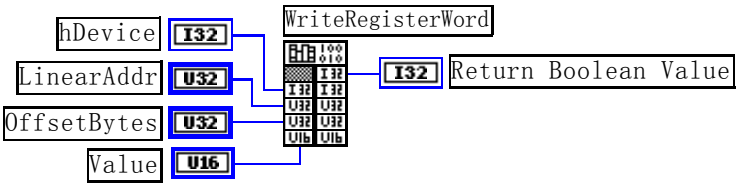
Visual C++:

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

Visual Basic:

```
Declare Function WriteRegisterWord Lib"PCI9757_32" ( _
                                   ByVal hDevice As Long, _
                                   ByVal LinearAddr As Long, _
                                   ByVal OffsetBytes As Long, _
                                   ByVal Value As Integer) As Boolean
```

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。

pbLinearAddr　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
        :
```

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:

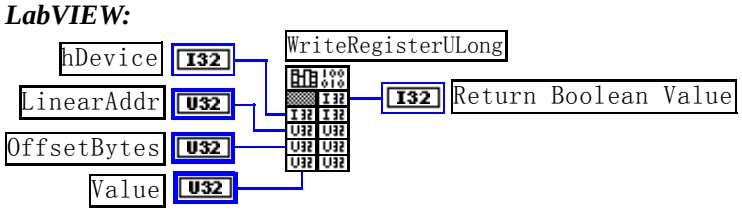
Visual C++:

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

Visual Basic:

```
Declare Function WriteRegisterULong Lib "PCI9757_32" ( _
                                    ByVal hDevice As Long, _
                                    ByVal LinearAddr 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。

pbLinearAddr 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
    :
```

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:

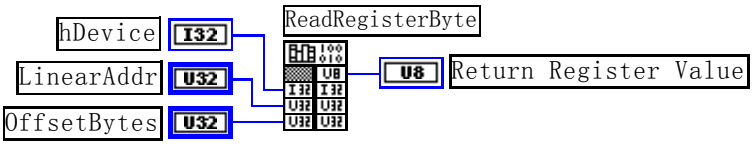
Visual C++:

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

Visual Basic:

```
Declare Function ReadRegisterByte Lib"PCI9757_32" ( _
                                        ByVal hDevice As Long, _
                                        ByVal LinearAddr 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。

pbLinearAddr　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
    :
```

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:

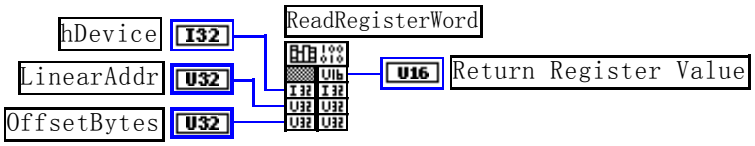
Visual C++:

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

Visual Basic:

```
Declare Function ReadRegisterWord Lib"PCI9757_32" ( _
                                   ByVal hDevice As Long, _
                                   ByVal LinearAddr 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或 CreateDeviceEx创建。

pbLinearAddr 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
    :
```

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++:

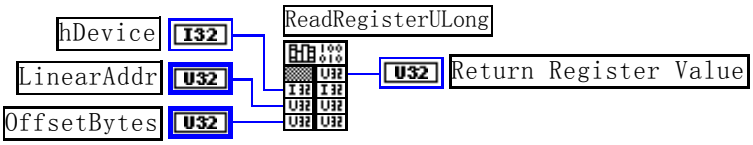
```
ULONG ReadRegisterULong (HANDLE hDevice,
```

PUCHAR pbLinearAddr,
ULONG OffsetBytes)

Visual Basic:

```
Declare Function ReadRegisterULong Lib"PCI9757_32" ( _
                                        ByVal hDevice As Long, _
                                        ByVal LinearAddr 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。

pbLinearAddr　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
    :
```

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、I0 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**

Visual C++:

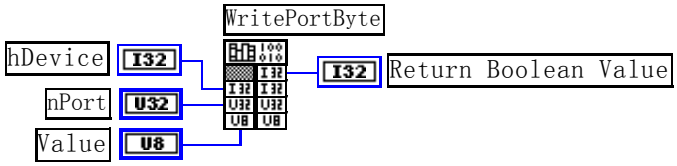
```
BOOL WritePortByte (HANDLE hDevice,
                    PUCHAR pbPort,
                    BYTE Value)
```

Visual Basic:

```
Declare Function WritePortByte Lib "PCI9757_32" ( _
                                    ByVal hDevice As Long, _
```

ByVal nPort 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。
pbPort 设备的 I/O 端口号。
Value 写入由 pbPort 指定端口的值。
返回值：若成功，返回 TRUE，否则返回 FALSE。
相关函数： CreateDevice WritePortByte WritePortWord
WritePortULong ReadPortByte ReadPortWord

- **以双字(16Bit)方式写 I/O 端口**

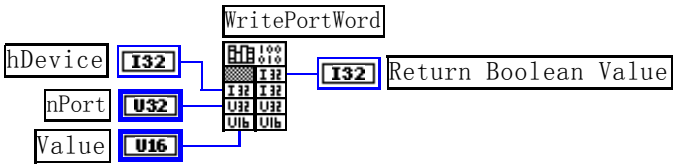
Visual C++:

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

Visual Basic:

```
Declare Function WritePortWord Lib"PCI9757_32" (ByVal hDevice As Long,_
                                                ByVal nPort 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。
pbPort 设备的 I/O 端口号。
Value 写入由 pbPort 指定端口的值。
返回值：若成功，返回 TRUE，否则返回 FALSE，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**

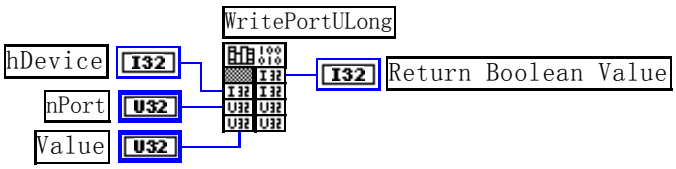
Visual C++:

```
BOOL WritePortULong(HANDLE hDevice,
                    PUCHAR pbPort,
                    ULONG Value)
```

Visual Basic:

```
Declare Function WritePortULong Lib "PCI9757_32" (ByVal hDevice As Long,_
                                                  ByVal nPort 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或 CreateDeviceEx创建。
pbPort 设备的 I/O 端口号。
Value 写入由 pbPort 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**

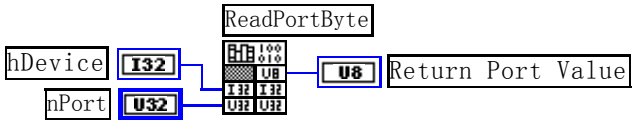
Visual C++:

```
BYTE ReadPortByte( HANDLE hDevice,
                   PUCHAR pbPort)
```

Visual Basic:

```
Declare Function ReadPortByte Lib "PCI9757_32" (ByVal hDevice As Long,_
                                                ByVal nPort 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。
pbPort 设备的 I/O 端口号。
返回值：返回由 pbPort 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**

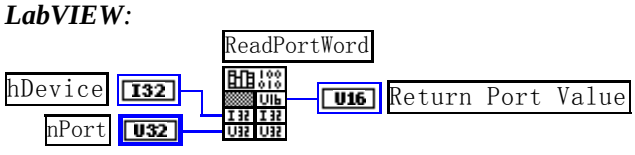
Visual C++:

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

Visual Basic:

```
Declare Function ReadPortWord Lib "PCI9757_32" (ByVal hDevice As Long,_
                                                ByVal nPort 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。
pbPort 设备的 I/O 端口号。
返回值：返回由 pbPort 指定的端口的值。
相关函数： CreateDevice WritePortByte WritePortWord
WritePortULong ReadPortByte ReadPortWord

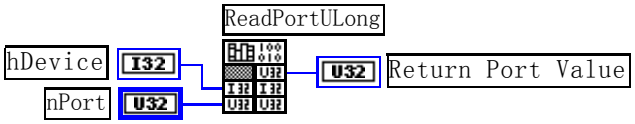
◆ **以四字节(32Bit)方式读 I/O 端口**

Visual C++:

ULONG ReadPortULong(HANDLE hDevice,
PUCHAR pbPort)

Visual Basic:

Declare Function ReadPortULong Lib "PCI9757_32" (ByVal hDevice As Long,_
ByVal nPort As Long) As Long



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。

pbPort　设备的 I/O 端口号。

返回值：返回由 pbPort 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

◆ **创建内核系统事件**

函数原型：

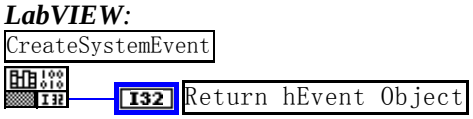
Visual C++:

HANDLE CreateSystemEvent(void)

Visual Basic:

Declare Function CreateSystemEvent Lib "PCI9757_32" () As Long

LabVIEW:



功能：创建系统内核事件对象，供 InitDeviceInt 和 VB 子线程等函数使用。

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 "PCI9757_32" (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。