Software Reference Manual

Table of Contents

Introduction

This manual provides a single reference guide to all of the generic driver interfaces and utilities for our hardware products. If a driver is designed for use with more than one device, it is described here. Please note, drivers specific to a model or model family are described in that model's manual.

You can use this document in a number of ways. The table of contents allows you to jump straight to a particular driver for easy reference use. The lists of drivers-by-type at the end of the manual allow you to browse for which specific drivers you should be using in your environment. Or, you could read the entire manual front-to-back; although this may seem inefficient, it may provide useful insight into other products you may be able to use in your system.

Not all of the information in this manual will apply to your specific hardware. We've provided some easy tools to determine if you should bother reading a particular section. For example, the beginning of each section will look similar to the following:

RISCTerm

APPLIES TO						
USB	*ISA, PC/104*	**SERIAL**	**DOS**	*WIN 95, 98*	*WIN NT4*	*Linux*
PCI, PCI/104	*A/D, D/A*	*NON-SERIAL*	**WIN3.x**	*WIN 98se, Me*	*WIN 2000, XP*	

There are three possible levels of compatibility shown by these tables. In the example above the SERIAL, DOS, and WIN3.x entries are shown in a normal style, black, easy to read. This indicates "optimum" compatibility. On the other hand, the *WIN95, 98* section is shown in an italic font. This indicates "works, but not optimally". All of the other sections are shown in a light color, and italicized. This section of the manual will not be useful if you're using one of these combinations of bus, card type, and operating system.

The table above is from the RISCTerm.EXE section of this manual. In addition to providing a quick reference table at the start of each section, certain pertinent facts will be presented below the table, pointing you towards finding a better solution than the current section. For example, the full table for RISCTerm.EXE actually looks like the following:

RISCTerm

APPLIES TO						
USB	*ISA, PC/104*	**SERIAL**	**DOS**	*WIN 95, 98*	*WIN NT4*	*Linux*
PCI, PCI/104	*A/D, D/A*	*NON-SERIAL*	**WIN3.x**	*WIN 98se, Me*	*WIN 2000, XP*	
WIN 95,98, WIN NT4, WIN98se, Me, WIN 2000, XP users should refer to WINRISC.EXE, most Linux distributions ship with minicomm*, a similar utility.*						

Please note: although no "bus types" (PCI, PCI/104, , ISA, USB, PC/104) are emphasized, this does not mean *none*; it indicates the lack of relevance. RISCTerm applies to SERIAL, regardless of architecture.

Utilities

FindBase

APPLIES TO						
USB	**ISA, PC/104**	*SERIAL*	**DOS**	*WIN 95, 98*	*WIN NT4*	*Linux*
PCI, PCI/104	*A/D, D/A*	*NON-SERIAL*	**WIN3.x**	*WIN 98se, Me*	*WIN 2000, XP*	
Users of PnP Operating Systems should refer to OS provided tools, such as Device Manager						

FindBase is a utility program designed to assist you in selecting an available address to use when installing ISA or PC/104 hardware for the first time. Hardware devices will malfunction if they are installed with the Base Address the same as or even merely overlapping any existing device. FindBase uses a complex search algorithm to read from the I/O ports on the system, looking for indications that some piece of hardware already exists at a given location in I/O memory.

It is important to keep in mind that no algorithm to determine used non-plug-and-play resources is perfect. Certain devices simply cannot be found. Therefore, when using FindBase to select an address for your new hardware installation, its very useful to have the documentation for your computer, its motherboard, and any other hardware you already have installed available for reference. These documents should be able to refine the search results returned by FindBase to eliminate as many problems as possible.

In addition, our hardware manuals contain a handy reference table of the most commonly occupied addresses. Consider carefully if you want to install your new hardware at any of the addresses listed in these tables even if recommended by FindBase – the next device you install may normally reside there, or something currently disabled in CMOS may normally occupy that address range.

To use FindBase you must be running DOS or Windows 3.x. Windows XP and similarly advanced operating systems will NOT work. In any plug-and-play operating system, you should use the tools provided by that operating system, instead of FindBase. For example, in all 32-bit Windows operating systems Device Manager has a similar functionality.

FindBase should be run before you have installed the new device into your system. Select the device from the presented list, or, if it is not present, choose one of the generic selections that matches your hardware best. For example, if your new device requires 30 bytes of registers, you would select “32-byte” from the list of presented options. When FindBase presents its recommendation consider carefully its suitability. Check the reference table presented in the hardware manual, and read the computer, motherboard, and already installed hardware manuals to ensure the FindBase recommended address is truly available. Once you’re satisfied, follow the instructions in your device’s Setup program or hardware manual to configure the Address jumpers and/or switches to the recommended setting.

If you have any difficulty using your newly installed device for the first time it may be related to a base address conflict. Running FindBase again and rejecting its first recommendation may provide an alternate address setting that can resolve such issues.

Note, FindBase creates a file in the same directory as its executable containing a list of addresses it has already scanned during its complex search algorithm. Certain devices do not react well to being read from during the search, and can even cause the system to lock up or hang. FindBase takes every step we could think of to eliminate this situation, and in fact no longer causes lock ups on any systems we have tested it with. However, just in case it does in the field, FindBase will detect the existence of this extra file when it is run, and learn not to check the address at which it locked up during the previous attempt. Therefore, if the system locks up during the FindBase scan, don't hesitate to simply run FindBase again – it will pick up where it left off, skipping the offending I/O address.

PCIFind

APPLIES TO						
USB	*ISA, PC/104*	*SERIAL*	**DOS**	**WIN 95, 98**	**WIN NT4**	*Linux*
PCI, PCI/104	*A/D, D/A*	*NON-SERIAL*	**WIN3.x**	**WIN 98se, Me**	**WIN 2000, XP**	
Linux users should refer to pcifind.pl, a similar utility						

PCIFind is used to determine the resources allocated to PCI-based devices by the plug-and-play manager in your computer system. These resources include Base Address and IRQs, both of which must be known to successfully interface to any hardware device.

PCIFind is a core component of all PCI software. This one executable contains a DOS compatible program, as well as a 32-bit windows compatible program. The windows portion of the executable automatically detects and loads an NT 4, 2000, and XP required kernel device driver (NTioPCI.SYS) when used on those operating systems. Under windows PCIFind places itself in the "run" key in the registry so it will be executed silently at each boot, ensuring critical information remains up-to-date.

When used in DOS:
PCIFind will display a text-mode interface listing all of the detected PCI I/O hardware installed in the system one at a time. Each will be presented with the list of all used Base Addresses and the IRQ associated with the device (if any). If you have more than one I/O device installed you can press any key to see the next device's information.

Write this information down! You will need it when running sample programs or calibration utilities, etc, if those programs do not automatically detect the information.

This information can change at every boot. The plug-and-play manager used by your system (generally the BIOS when running DOS) is allowed to reallocate resources as needed, and "need" is not defined. Generally, the only time BIOS will change resources is if the device is moved to a

different slot, or an additional device is installed. If you encounter difficulty using the address you have noted for your hardware, rerun PCIFind to confirm nothing has changed.

Because the information can change at every boot, using PCIFind's information in any embedded environment, or even any consumer environment where user-friendliness is a goal, is probably not the best idea. You should instead write the detection code into your application so it automatically detects the base address and IRQ every time it runs, and you don't need to run PCIFind at all. We provide the source code for the detection algorithm in the /PCI/source directory on our Software Master CD. Use this program as the basis for your own, and you'll have a clean user experience, regardless of changing resource allocation.

When used in Windows:
PCIFind copies hard-to-find resource information out of the low-level windows system into the normal registry. It also displays the most pertinent pieces of this information in an easy-to-read format to the user. The displayed information includes the Base Addresses and IRQs assigned by the Plug and Play manager (generally windows itself takes over this duty from the BIOS), as well as the name of the device, if known.

Write this information down! Not all of the programs you use may automatically load this information out of the registry; you may need to enter the address and/or IRQ manually for these programs.

When writing your own programs, INTEGRATE! Your users do not want to run PCIFind manually to determine resources, then type the information into your software every time! PCIFind runs automatically at every boot to keep the registry information up-to-date, all your program needs to do is read out this information each time your software runs. Sample programs demonstrating how to read this information from the registry are included with all of our hardware, but here's a brief overview:

The information is stored in the registry, under the HKLM/Software/PCIFind key. There's quite a bit of information tucked into this registry key, but the only entries your software needs are:

`HKLM/Software/PCIFind/NTioPCI/Parameters/NumDevices`
(this is a REG_DWORD)
and
`HKLM/Software/PCIFind/NTioPCI/Parameters/PCICommonConfig`
(this is a REG_BINARY)

NumDevices indicates how many of our PCI devices were detected, and PCICommonConfig is an array of PCI_COMMON_CONFIG structures, NumDevices in length, as shown conceptually by this "C" code fragment:

PCI_COMMON_CONFIG PCICommonConfig[NumDevices-1];

Once the data has been read from the registry into the array of structures, you simply iterate the array, checking for your card type. Each card has a known DeviceID and VendorID (provided in the hardware manual), and the structure has VendorID and DeviceID fields; when you find the array element containing the matching VendorID and DeviceID, you've found your device, and you simply read the IRQ and Base Address fields to determine your resources. Again, for details, refer to the sample programs provided on the Software Master CD.

Poly

APPLIES TO						
USB	*ISA, PC/104*	*SERIAL*	**DOS**	*WIN 95, 98*	*WIN NT4*	*Linux*
PCI, PCI/104	**A/D, D/A**	*NON-SERIAL*	**WIN3.x**	*WIN 98se, Me*	*WIN 2000, XP*	
32-bit Widows users should refer to WinPoly for a Windows specific solution						

Poly is a utility program designed to do some rather arcane arithmetic for you, simplifying and improving your application software.

Specifically, Poly performs a least squares curve fitting with partial rotation, for output polynomials of between first and fifteenth order.

Basically, you provide a table of x,y data pairs, and Poly tries its best to tell you a polynomial that produces that table. For example, given:

x	y
-2	4
0	0
2	4

Poly should be capable of producing the polynomial $y = x^2$, and it will indeed, if you've limited its search to second order polynomials.

There are several things to keep in mind when using least-squares curve fitting: the more points in your table, the better, and you *must* have more points in your table than you have order of polynomial selected for output. For example, if you choose a 15^{th} order polynomial, you must have *at least* 16 points in your table, or your results will be *too* good.

One very common use for a program like Poly is "Sensor Linearization." Many sensor types have non-linear output values versus their inputs. For example, thermocouples have complex temperature versus millivolt curves associated with them. It is possible to acquire a gamut of data, feed it into Poly, and produce a polynomial capable of interpolating (and to some extent extrapolating) to determine what temperature any given millivolt reading actually means.

Although thermocouples usually provide a polynomial for linearization purposes, so there's no reason to re-derive them using a program like this, other sources of nonlinearity are rarely accompanied by their polynomials: instead, they may have a small table, a little graph from which a table could be derived, or nothing at all! Poly can be very helpful when using these nonlinear data sources.

This old DOS utility is pretty handy for small tables, but it isn't capable of loading data from a spreadsheet, nor any other file for that matter; when possible, use WinPoly, a "full-featured" polynomial generation utility for Windows.

RISCTerm

APPLIES TO						
USB	*ISA, PC/104*	**SERIAL**	**DOS**	*WIN 95, 98*	*WIN NT4*	*Linux*
PCI, PCI/104	*A/D, D/A*	*NON-SERIAL*	**WIN3.x**	*WIN 98se, Me*	*WIN 2000, XP*	
WIN 95,98, WIN NT4, WIN98se, Me, WIN 2000, XP users should refer to WINRISC.EXE, most Linux distributions ship with minicomm*, a similar utility*						

RISCTerm is a short form of the full name "Really Incredibly Simple Communications Terminal." The utility is a terminal program designed to be as easy to use as possible, especially when used with our line of RS-485 8031-based data acquisition pods.

In order to use RISCTerm you need to supply the Base Address of the 16450 compatible UART you want to use. In addition, you can provide that port's IRQ if you want to use IRQ based serial communications (preferred), but the program will allow you to specify IRQ 0 to indicate "polling mode." Please note, higher baud rates on slower computers won't work well in polling mode – its best to use IRQ mode if you can.

The utility is fairly self-documenting. The most important thing to remember is "F1 for help". The help screen has a list of all the possible keystrokes associated with controlling the program. Also, the status bar along the bottom of the screen is mouse-sensitive. If you have a mouse driver loaded you can left-click or right-click on the various displayed items to toggle them, or to select the next or previous option, where applicable.

RISCTerm defaults to 9600 Baud with 7E1 communication parameters. This is the same as the shipping configuration of our 8031 based Data Acquisition Pods, so you can get up and running as fast as possible. The available Baud rates in RISCTerm is fairly limited; the list was chosen to match the Baud rates available to the Pods.

Although you cannot "Save" settings so RISCTerm will apply them as the default the next time you run it, you can specify various settings via command line parameters. Run RISCTerm and type "F1" for more information on options that can be set from the command line.

RISCTerm, although designed with RS-485 in mind, works with any 16450 compatible serial device, allowing you to very simply interface to a wide variety of devices. To further enhance its broad appeal, you can enter a binary display mode. Pressing ALT-D within RISCTerm will open a small "debug" style interface window, showing the actual ASCII coded hexadecimal values for the received serial data bytes. This greatly assists in debugging binary serial connections.

WinPoly

APPLIES TO						
USB	*ISA, PC/104*	*SERIAL*	*DOS*	**WIN 95, 98**	**WIN NT4**	*Linux*
PCI, PCI/104	**A/D, D/A**	*NON-SERIAL*	*WIN3.x*	**WIN 98se, Me**	**WIN 2000, XP**	

WinPoly is a 32-bit Windows version of the utility "Poly", described above.

It performs the same curve-fitting function, but with a little extra oomph: WinPoly allows you to load a CSV (comma delimited) text file to use as the table of data. WinPoly will draw graphs of the resultant polynomial curves, as well as your data set, so you can spot the "...your results will be *too* good..." kinds of errors right off the bat.

WinRISC

APPLIES TO						
USB	*ISA, PC/104*	**SERIAL**	*DOS*	**WIN 95, 98**	**WIN NT4**	*Linux*
PCI, PCI/104	*A/D, D/A*	*NON-SERIAL*	*WIN3.x*	**WIN 98se, Me**	**WIN 2000, XP**	

WinRISC provides an easy-to-use communication terminal for 32-bit windows environments.

This utility is designed to interact seamlessly with our RS-485 data acquisition pods, and has been configured for the 7E1, 9600 baud protocol that is the default for those pods. In addition to being tuned for use with our pods, WinRISC has a very full-featured set of flexible configuration options, including various error reporting options, to assist you in debugging any serial interconnect.

Both direct keyboard input (typing) and file transfer options are available, the program supports COM ports up to "COM255", and all features of RS232 communications signals are available for your control.

WinRISC is based on a Microsoft example program called MTTTY, whose source is available via the MSDN downloads site. MTTTY provides a very detailed and flexible example of programming multi-threaded serial communications routines in Windows; WinRISC adds support for COM ports higher than "COM4", some ease-of-use user interface features, additional Baud rate choices, and options are defaulted for use with our RS485 8031-based data acquisition Pods.

To use WinRISC, merely run the executable, select the COM port from the combo box, and click the CONNECT button (or choose File | Connect.) Once the connection has been established, click in the large open text area, and type. Data is sent across the serial link immediately (no need to press ENTER), and any received data will be displayed simultaneously.

You can select a different Baud rate from the droplist at any time; to choose a new COM port, disconnect first. If the COM port you wish to use is not in the list (the list only shows COM1 through COM24) you can type any COM number you'd like into the edit portion of the combobox.

Source

VB.NET Headers

APPLIES TO						
USB	**ISA, PC/104**	*SERIAL*	*DOS*	*WIN 95, 98*	*WIN NT4*	*Linux*
PCI, PCI/104	*A/D, D/A*	**NON-SERIAL**	*WIN3.x*	*WIN 98se, Me*	**WIN 2000, XP**	

While we don't provide full samples for Visual BASIC (VB) .NET, this document describes the process needed to work in VB.NET fully. First realize our existing VB samples are written for VB 5 and 6; there are some significant differences in the languages. To simplify your transition to .NET, we've created a set of VB.NET compatible "header" files. These are source code files used to interface between the source code in the sample programs, and the binary format .DLL files we provide as driver APIs. Versions of these header files already exist in our sample programs' directories as installed by the Software Master CD, but they are compatible with VB 5 and 6. To use our samples in VB.NET, you'll need to use the .NET environment to convert the project, then delete the VB 5/6 header (with a `.BAS` extension) and copy in the VB.NET header (with a `.VB` extension) of the same name. The VB.NET headers are installed to the subdirectory `Win32\HeadersVB.NET\` in the install directory.

For example, if you've installed the software for the USB-DIO-32 to the directory `C:\USBDIO32\`, the VB.NET header for the appropriate driver (in this case AIOUSB) was installed to `C:\USBDIO32\Win32\HeadersVB.NET\AIOUSB.vb`.

Drivers

ACCES32

APPLIES TO						
USB	**ISA, PC/104**	*SERIAL*	*DOS*	**WIN 95, 98**	**WIN NT4**	*Linux*
PCI, PCI/104	*A/D, D/A*	**NON-SERIAL**	*WIN3.x*	**WIN 98se, Me**	**WIN 2000, XP**	
WIN3.x is supported by VBACCES a very similar driver; Linux users should refer to ACCESLinux						

ACCES32 provides programmers with a fast, simple way to access I/O mapped devices and to control custom hardware (such as our plug-in data acquisition cards) by means of direct register access. ACCES32 is in the form of a 32-bit dynamic link library and makes no assumptions as to how the hardware is configured.

Note:
Our AIOWDM driver provides the same functions and more; see its section for details.

Windows tries to protect the system from runaway programs by removing the ability to access ports, making register-level access extremely difficult.

ACCES32.DLL provides direct register access (to any port not currently registered with Windows as being in use) faster and easier and works without wrappers using the standard 32-bit DLL interface. ACCES32.DLL provides fast, simple access to I/O mapped devices and to control hardware such as data acquisition cards.

Nine functions are exported by ACCES32.DLL:

- InPortB
- OutPortB
- INSB
- InPort
- OutPort
- INSW
- InPortL / InPortDWord (two names for the same function)
- OutPortL / OutPortDWord (two names for the same function)
- INSD

To use these functions in a program, they must first be imported into that program. The simplest way to import these functions in Pascal is to include the ACCES32 unit in your Uses clause, and in C to #include ACCES32.h. Additionally, in C, the library file CBACCES.lib (for C++ Builder) or VCACCES.lib (for Visual C++) must be linked to the program. This is accomplished by adding it to the project or editing the makefile. If you are not using the ACCES32 header file and wish to import the functions manually, the necessary lines in Pascal would be:

```
function InPortB(Port: DWord): Word; cdecl; external 'ACCES32.dll';
function InPort(Port: DWord): Word; cdecl; external 'ACCES32.dll';
function InPortL(Port: DWord): DWord; cdecl; external 'ACCES32.dll';
function InPortDWord(Port: DWord): DWord; cdecl; external 'ACCES32.dll';
function OutPortB(Port: DWord; Value: Byte): Word; cdecl; external 'ACCES32.dll';
function OutPort(Port: DWord; Value: Word): Word; cdecl; external 'ACCES32.dll';
function OutPortL(Port: DWord; Value: DWord): Word; cdecl; external 'ACCES32.dll';
function OutPortDWord(Port: DWord; Value: DWord): Word; cdecl; external 'ACCES32.dll';
function INSB(Port, Count: LongWord; var pBuffer: Byte): LongWord; cdecl; external
'ACCES32.dll';
function INSW(Port, Count: LongWord; var pBuffer: Word): LongWord; cdecl; external
'ACCES32.dll';
function INSD(Port, Count: LongWord; var pBuffer: LongWord): LongWord; cdecl; external
'ACCES32.dll';
```

To import the functions manually in C, the necessary lines would be:

```
extern "C" __declspec(dllimport) unsigned short InPort(unsigned long Port);
extern "C" __declspec(dllimport) unsigned short InPortB(unsigned long Port);
extern "C" __declspec(dllimport) unsigned long InPortL(unsigned long Port);
extern "C" __declspec(dllimport) unsigned long InPortDWord(unsigned long Port);
extern "C" __declspec(dllimport) unsigned short OutPort(unsigned long Port, unsigned short
Value);
extern "C" __declspec(dllimport) unsigned short OutPortB(unsigned long Port, unsigned char
Value);
extern "C" __declspec(dllimport) unsigned short OutPortL(unsigned long Port, unsigned long
Value);
extern "C" __declspec(dllimport) unsigned short OutPortDWord(unsigned long Port, unsigned
long Value);
extern "C" __declspec(dllimport) unsigned long function INSB(unsigned long Port, unsigned
long Count, unsigned char *pBuffer);
extern "C" __declspec(dllimport) unsigned long function INSW(unsigned long Port, unsigned
long Count, unsigned short *pBuffer);
extern "C" __declspec(dllimport) unsigned long function INSD(unsigned long Port, unsigned
long Count, unsigned long *pBuffer);
```

No header file is provided for Visual Basic, so the functions must be imported manually with the following declarations:

```
Private Declare Function InPortB Lib "ACCES32" Alias "VBInPortB" (ByVal Port As Long) As Integer
Private Declare Function InPort Lib "ACCES32" Alias "VBInPort" (ByVal Port As Long) As Integer
Private Declare Function InPortL Lib "ACCES32" Alias "VBInPortL" (ByVal Port As Long) As Long
Private Declare Function InPortDWord Lib "ACCES32" Alias "VBInPortDWord" (ByVal Port As Long) As Long
Private Declare Function OutPortB Lib "ACCES32" Alias "VBOutPortB" (ByVal Port As Long, ByVal Value As Byte) As Integer
Private Declare Function OutPort Lib "ACCES32" Alias "VBOutPort" (ByVal Port As Long, ByVal Value As Integer) As Integer
Private Declare Function OutPortL Lib "ACCES32" Alias "VBOutPortL" (ByVal Port As Long, ByVal Value As Long) As Integer
Private Declare Function OutPortDWord Lib "ACCES32" Alias "VBOutPortDWord" (ByVal Port As Long, ByVal Value As Long) As Integer
Private Declare Function INSB Lib "ACCES32" Alias "VBINSB" (ByVal Port As Long, ByVal Count As Long, ByRef pBuffer As Byte) As Integer
Private Declare Function INSW Lib "ACCES32" Alias "VBINSW" (ByVal Port As Long, ByVal Count As Long, ByRef pBuffer As Integer) As Integer
Private Declare Function INSD Lib "ACCES32" Alias "VBINSD" (ByVal Port As Long, ByVal Count As Long, ByRef pBuffer As Long) As Integer
```

The commands access the ports either 1, 2, or 4 bytes wide. InPortB will return 2 bytes of data but only the lowest byte is valid. With all six functions, a return value of hex AA55 signifies an error condition. ACCES32 comes with four sample applications for Windows written in Visual C++ 5.0, VisualBASIC 5.0, Delphi 5.0, and Borland C++ Builder 5.0.

ACCES32.DLL allows ANY windows language to access I/O (Port) memory resources as if the language was designed for it. This is a 32-bit DLL designed for 32-bit applications. For 16-bit applications under VisualBASIC, you should use our Windows 3.x driver, VBACCES.DLL.

In order to use ACCES32 in Windows NT 4.0, 2000, or XP, an additional file, ACCESNT.SYS, must be copied to the Drivers directory, which is usually `C:\Windows\System32\Drivers`.

AIOWDM

APPLIES TO						
USB	*ISA, PC/104*	*SERIAL*	*DOS*	*WIN 95, 98*	*WIN NT4*	*Linux*
PCI, PCI/104	*A/D, D/A*	**NON-SERIAL**	*WIN3.x*	**WIN 98se, Me**	**WIN 2000, XP**	

AIOWDM provides IRQ-handling support, generic watchdog functionality, and direct register access for user programs.

All functions are exported in three forms: underscored with the cdecl calling convention (the easiest for C++), undecorated with the stdcall calling convention (easiest for Visual BASIC), and undecorated with the cdecl calling convention (easiest for most other languages). Import declarations are given for Object Pascal / Delphi, C++ Builder / Visual C++, and Visual BASIC 6. Visual BASIC doesn't have true pointers, and thus pointer-using functions are declared in Visual BASIC to always have a non-null pointer.

When using AIOWDM your program should always begin by calling the first two functions listed below, GetNumCards, and QueryCardInfo. These two functions allow your software to acquire a "CardNum", used as a kind of "handle" into the list of cards installed in your system that AIOWDM has detected. All further accesses will use this CardNum to ensure you're controlling the correct device.

GetNumCards()
function GetNumCards: Integer; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) signed long GetNumCards(void);

Private Declare Function GetNumCards Lib "AIOWDM" Alias "VBGetNumCards" () As Long

This function returns the number of cards in the system using AIOWDM. The cards are numbered from zero to one less than the number returned by GetNumCards(); these card numbers are specified for the CardNum parameters of most other AIOWDM.dll functions. If an invalid card number is specified to another function, the function will fail.

QueryCardInfo()
function QueryCardInfo(CardNum: Integer; pDeviceID: PLongWord; pBase: PLongWord; pNameSize: pLongWord; pName: PChar): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long QueryCardInfo(long CardNum, unsigned long *pDeviceID, unsigned long *pBase, unsigned long *pNameSize, unsigned char *pName);

Private Declare Function QueryCardInfo Lib "AIOWDM" Alias "VBQueryCardInfo" (ByVal CardNum As Long, ByRef pDeviceID As Long, ByRef pBase As Long, ByRef pNameSize as Long, ByVal pName As String) As Long

This function retrieves information about the card indicated by CardNum. It returns nonzero if it succeeds, or zero if it fails (which probably means CardNum was bad). The information retrieved is:

* DeviceID. This is a DWord value that tells you what kind of card it is. (For PCI cards, this is equal to their PCI DeviceID.)
* Base Address. This is the card's location within I/O space. To control the card, take data from it, etc, use this base address.
* "Friendly Name". This Windows ANSI string is the name of the card. It corresponds directly to the DeviceID, but makes sense to a human. If the friendly name could not be retrieved, QueryCardInfo() will give an empty string.

Parameters are as follows:

* pDeviceID - This is a pointer to a place to put the DeviceID of the card (or a null pointer if you don't need this information).
* pBase - This is a pointer to a place to put the base address of the card (or a null pointer if you don't need this information).
* pNameSize - This is a pointer to the size of the buffer for the card's friendly name (or a null pointer if you don't need this information). This value will be changed to the actual size needed for the buffer, including the terminating null character.
* pName - This is a pointer to a buffer for the card's friendly name (or a null pointer if you don't need this information). If the value pointed to by pNameSize is initially zero, or if it's greater after the call to QueryCardInfo() (indicating that a larger buffer is required), this buffer will not be altered.

WaitForIRQ()

function WaitForIRQ(CardNum: Integer): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long WaitForIRQ(long CardNum);

Private Declare Function WaitForIRQ Lib "AIOWDM" Alias "VBWaitForIRQ" (ByVal CardNum As Long) As Long

This function deadlocks the calling thread until an IRQ occurs or the request is aborted. It's intended for multi-threaded IRQ handling, where the calling application spawns a separate thread to service IRQs quickly.

It returns nonzero if it returned because an IRQ occurred, or zero if it returned because the wait was aborted or there was an error. If the wait was aborted, GetLastError() will return ERROR_OPERATION_ABORTED, and if someone is already waiting for an IRQ on that card, it will return ERROR_INVALID_FUNCTION. (Other errors will return other error codes.)

AbortRequest()
function AbortRequest(CardNum: Integer): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long AbortRequest(long CardNum);

Private Declare Function AbortRequest Lib "AIOWDM" Alias "VBAbortRequest" (ByVal CardNum As Long) As Long

This function cancels an IRQ request begun by WaitForIRQ() or COSWaitForIRQ(), or the one begun internally by WDGHandleIRQ(). It returns zero if it fails or nonzero if it succeeds, though the return value is generally not useful since if it fails there was no pending IRQ request on that card to abort.

Make sure you call AbortRequest() or CloseCard() before closing the requesting thread or application; if you don't, the thread will never complete, and the driver will retain the request even though the thread or application has been unloaded from memory. If an IRQ then occurs, or the request is aborted, the wait will resume execution in the nonexistent thread, which will cause a Blue Screen Of Death.

If this does happen, perhaps due to the application crashing, ***do not*** then call AbortRequest(). Instead, use Device Manager to disable all devices using AIOWDM, then re-enable them. This will clear the faulty request without causing a Blue Screen Of Death, and allow you to resume working.

CloseCard()
function CloseCard(CardNum: Integer): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long CloseCard(long CardNum);

Private Declare Function CloseCard Lib "AIOWDM" Alias "VBCloseCard" (ByVal CardNum As Long) As Long

This function aborts any pending IRQ requests and closes the handle for the card. While it's necessary to abort all pending requests before closing the application, it isn't necessary to close the handles, as this is done automatically when AIOWDM.dll is unloaded. (When the .DLL is unloaded during an application close, IRQ-requesting threads have been closed already, and thus it's already too late to abort the pending requests - doing so would simply crash the computer.)

It returns zero if it fails or nonzero if it succeeds, though the return value is generally not useful since if it fails there was no open handle to close.

COSWaitForIRQ()
function COSWaitForIRQ(CardNum: Integer; PPIs: LongWord; pData: Pointer): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long COSWaitForIRQ(long CardNum, unsigned long PPIs, void *pData);

Private Declare Function COSWaitForIRQ Lib "AIOWDM" Alias "VBCOSWaitForIRQ" (ByVal CardNum As Long, ByVal PPIs As Long, ByRef pData As Byte) As Long

This function is similar to WaitForIRQ(), but it also reads data from the card's PPIs immediately after the IRQ. It's designed specifically for the PCI-DIO-24S, PCI-DIO-48S, IOD24S, and IOD48S, and can be dangerous with another card (as some cards take action based on incoming reads), so check the DeviceID of your card through QueryCardInfo() first. Parameters are as follows:

* PPIs - This is the number of PPIs on the card. The PCI-DIO-24S has 1 PPI, the PCI-DIO-48S has 2 PPIs. (You can also specify 1 PPI with the PCI-DIO-48S if you don't need the data from the second one.)
* pData - This is a pointer to a buffer for the data. It must be at least 3 bytes per PPI.

For Visual BASIC, pData should be the first byte in an array of at least 3 bytes per PPI. For example, if Data was declared with "Dim Data(0 To 5) As Byte", you might make the call "COSWaitForIRQ(CardNum, 2, Data(0))".

WDGInit()
function WDGInit(CardNum: Integer): LongWord; cdecl;

extern __declspec(dllimport) unsigned long WDGInit(long CardNum);

Private Declare Function WDGInit Lib "AIOWDM" Alias "VBWDGInit" (ByVal CardNum As Long) As Long

This function prepares a watchdog card driver for further WDG...() function calls, like WDGPet() and WDGHandleIRQ(). It's designed specifically for the PCI-WDG-CSM, and can be dangerous with another card, so check the DeviceID of your card through QueryCardInfo() first. If you call one of these functions on a card whose driver hasn't been prepared, it will fail. It returns nonzero if it succeeds, or zero if it fails.

WDGHandleIRQ()

function WDGHandleIRQ(CardNum: Integer; Action: LongWord): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long WDGHandleIRQ(long CardNum, unsigned long Action);

Private Declare Function WDGHandleIRQ Lib "AIOWDM" Alias "VBWDGHandleIRQ" (ByVal CardNum As Long, Action As Long) As Long

This function sets up an IRQ handler to handle the next watchdog IRQ from the card specified. (This card's driver must already be prepared for watchdog operation by the WDGInit() function.) It returns nonzero if it succeeds, or zero if it fails. Action determines how the IRQ will be handled, as follows:

0 = Ignore.
This will allow the watchdog circuit to trip when the timer times out, performing a "hard" reset (if connected to the computer's reset line or power supply). A "hard" reset doesn't give Windows the opportunity to save important data, but is as certain a reset as possible. Note that this is what will happen if WDGHandleIRQ() is never called.

1 = Disable.
The watchdog timer will be disabled when the card generates an IRQ. This is not generally useful by itself except in testing.

2 = "Soft" Shutdown And Restart.
When the card generates an IRQ the driver will set the watchdog timer to a 90sec timeout, then begin a system shutdown. This "soft" shutdown allows Windows and other programs to save important data, but possibly allows them to cancel it. When the watchdog timer times out, however, the reset circuit will trip, performing a "hard" restart. If the "soft" shutdown completed successfully, the computer will be reset at that time.

3 = Disable And "Soft" Restart.
When the card generates an IRQ the driver will disable the watchdog timer, then begin a system restart. This is similar to 2, but doesn't allow the watchdog timer to actually time out except in case of failures so dire the IRQ doesn't get handled.

4 = "Mostly Soft" Shutdown And Restart.
When the card generates an IRQ the driver will set the watchdog timer to a 90sec timeout, then begin a system shutdown. This "mostly soft" shutdown allows Windows and other programs to save important data within a limited timeframe, and doesn't allow the shutdown to be cancelled. When the watchdog timer actually times out, the reset circuit will trip, performing a "hard" restart. If the "mostly soft" shutdown completed successfully, the computer will be reset at that time.

5 = Disable And "Mostly Soft" Restart.
When the card generates an IRQ the driver will disable the watchdog timer, then begin a system restart. This is similar to 4, but doesn't allow the watchdog timer to actually time out except in case of failures so dire the IRQ doesn't get handled.

Actions greater than 5 are not supported, and will cause this function to fail.

WDGSetTimeout()
function WDGSetTimeout(CardNum: Integer; Milliseconds: Double; MHzClockRate: Double): Double; cdecl;

extern __declspec(dllimport) double WDGSetTimeout(long CardNum, double Milliseconds, double MHzClockRate);

Private Declare Function WDGSetTimeout Lib "AIOWDM" Alias "VBWDGSetTimeout" (ByVal CardNum As Long, ByVal Milliseconds As Double, ByVal MHzClockRate As Double) As Double

This function stops the watchdog's timer, sets the timer's timeout duration, and prepares it for a WDGStart(). (This card's driver must already be prepared for watchdog operation by the WDGInit() function.) If it fails, it returns zero. If it succeeds, it returns the actual timeout achieved, which will be as close to Milliseconds as allowed by the counter's resolution. Parameters are as follows:

* Milliseconds is the desired timeout duration in milliseconds.
* MHzClockRate is the clock rate of the card, in MHz; for the PCI-WDG-CSM, this is 2.08333, and for the WDG-CSM (ISA card), this is 0.894886. These constants are provided in the AIOWDM headers as PCI_WDG_CSM_RATE and ISA_WDG_CSM_RATE.

WDGSetResetDuration()
function WDGSetResetDuration(CardNum: Integer; Milliseconds: Double; MHzClockRate: Double): Double; cdecl;

extern __declspec(dllimport) double WDGSetResetDuration(long CardNum, double Milliseconds, double MHzClockRate);

Private Declare Function WDGSetResetDuration Lib "AIOWDM" Alias "VBWDGSetResetDuration" (ByVal CardNum As Long, ByVal Milliseconds As Double, ByVal MHzClockRate As Double) As Double

This function sets the duration of the watchdog circuit's reset for the PCI-WDG-CSM, or enables or disables the buzzer for the WDG-CSM (ISA card). (This card's driver must already be prepared for watchdog operation by the WDGInit() function.) Setting Milliseconds to 0.0 will

disable the WDG-CSM's buzzer, or prevent the PCI-WDG-CSM from triggering the reset circuit even if it times out. Setting Milliseconds to a value greater than 0.0 but less than or equal to 1.0 will enable the WDG-CSM's buzzer, or set an infinite reset duration for the PCI-WDG-CSM. Setting Milliseconds to a value greater than 1.0 will enable the WDG-CSM's buzzer, or set the PCI-WDG-CSM's reset duration as close to the value passed as possible. If it sets the reset duration to a finite value, it returns that value, otherwise (if it fails or merely controls the buzzer) it returns zero.

WDGPet()
function WDGPet(CardNum: Integer): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long WDGPet(long CardNum);

Private Declare Function WDGPet Lib "AIOWDM" Alias "VBWDGPet" (ByVal CardNum As Long) As Long

This function "pets" a watchdog card, resetting its timers to prevent it from timing out. (This card's driver must already be prepared for watchdog operation by the WDGInit() function.) It returns nonzero if it succeeds, or zero if it fails.

WDGReadTemp()
function WDGReadTemp(CardNum: Integer): Double; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) double WDGReadTemp(long CardNum);

Private Declare Function WDGReadTemp Lib "AIOWDM" Alias "VBWDGReadTemp" (ByVal CardNum As Long) As Double

This function returns the temperature sensor from a watchdog card, in degrees Farenheit. (This card's driver must already be prepared for watchdog operation by the WDGInit() function.) A watchdog card without a temperature sensor will report a temperature of 194.0 degrees F. An actual temperature will never be less than 7.0 degrees F (and even that should never be seen in practice, of course). If this function fails, it returns 0.0 degrees F.

WDGReadStatus()
function WDGReadStatus(CardNum: Integer): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long WDGReadStatus(long CardNum);

Private Declare Function WDGReadStatus Lib "AIOWDM" Alias "VBWDGReadStatus" (ByVal CardNum As Long) As Long

This function reads a watchdog card's status byte at Base + 4. (This card's driver must already be prepared for watchdog operation by the WDGInit() function.) Note that this action incidentally enables watchdog IRQs, but that IRQs will be ignored unless WDGHandleIRQ() has been called for that card. It returns the value of the status byte (which will be FF hex or less) if it succeeds, or a value greater than FF hex if it fails. The format of this byte is described in the card's manual.

WDGStart()
function WDGStart(CardNum: Integer): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long WDGStart(long CardNum);

Private Declare Function WDGStart Lib "AIOWDM" Alias "VBWDGStart" (ByVal CardNum As Long) As Long

This function starts watchdog timing. (This card's driver must already be prepared for watchdog operation by the WDGInit() function.) Note that the watchdog timer should be configured (as by WDGSetTimeout()) to start properly. It returns nonzero if it succeeds, or zero if it fails.

WDGStop()
function WDGStop(CardNum: Integer): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long WDGStop(long CardNum);

Private Declare Function WDGStop Lib "AIOWDM" Alias "VBWDGStop" (ByVal CardNum As Long) As Long

This function stops watchdog timing. (This card's driver must already be prepared for watchdog operation by the WDGInit() function.) It returns nonzero if it succeeds, or zero if it fails.

EmergencyReboot()
function EmergencyReboot(): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long EmergencyReboot(void);

Private Declare Function EmergencyReboot Lib "AIOWDM" Alias "VBEmergencyReboot" () As Long

This function begins a "mostly soft" restart, which allows Windows and other programs to save important data within a limited timeframe, but doesn't allow the restart to be cancelled. It returns nonzero if it succeeds, or zero if it fails(which probably means the calling process isn't allowed to restart the system).

InPortB()

function InPortB(Addr: LongWord): Word; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned short InPortB(unsigned long Addr);

Private Declare Function InPortB Lib "AIOWDM" Alias "VBInPortB" (ByVal Addr As Long) As Integer

This function reads a byte from a hardware register. It will normally return the byte read, but on an error will return AA55 hex. If this occurs, check Device Manager for an error on your device. Unlike the similar one from ACCES32, this function will not work without a device to load the driver.

InPort()
function InPort(Addr: LongWord): Word; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned short InPort(unsigned long Addr);

Private Declare Function InPort Lib "AIOWDM" Alias "VBInPort" (ByVal Addr As Long) As Integer

This function reads a word from a hardware register. It will normally return the word read, but on an error will return AA55 hex, similar to InPortB().

InPortL()
function InPortL(Addr: LongWord): LongWord; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned long InPortL(unsigned long Addr);

Private Declare Function InPortL Lib "AIOWDM" Alias "VBInPortL" (ByVal Addr As Long) As Long

This function reads a longword from a hardware register. It will normally return the longword read, but on an error will return AA55 hex, similar to InPortB().

OutPortB()
function OutPortB(Addr: LongWord; Value: Byte): Word; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned short OutPortB(unsigned long Addr, unsigned char Value);

Private Declare Function OutPortB Lib "AIOWDM" Alias "VBOutPortB" (ByVal Addr As Long, ByVal Value As Byte) As Integer

This function writes a byte to a hardware register. It will normally return zero, but on an error will return AA55 hex, similar to InPortB().

OutPort()
function OutPort(Addr: LongWord; Value: Word): Word; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned short OutPort(unsigned long Addr, unsigned short Value);

Private Declare Function OutPort Lib "AIOWDM.dll" Alias "VBOutPort" (ByVal Addr As Long, ByVal Value As Integer) As Integer

This function writes a word to a hardware register. It will normally return zero, but on an error will return AA55 hex, similar to InPortB().

OutPortL()
function OutPortL(Addr: LongWord; Value: LongWord): Word; cdecl; external 'AIOWDM.dll';

extern __declspec(dllimport) unsigned short OutPortL(unsigned long Addr, unsigned long Value);

Private Declare Function OutPortL Lib "AIOWDM.dll" Alias "VBOutPortL" (ByVal Addr As Long, ByVal Value As Long) As Integer

This function writes a longword to a hardware register. It will normally return zero, but on an error will return AA55 hex, similar to InPortB().

Win32IRQ

APPLIES TO						
USB	**ISA, PC/104**	*SERIAL*	*DOS*	**WIN 95, 98**	**WIN NT4**	*Linux*
PCI, PCI/104	*A/D, D/A*	**NON-SERIAL**	*WIN3.x*	**WIN 98se, Me**	**WIN 2000, XP**	

APPLIES TO						
USB	*ISA, PC/104*	**SERIAL**	*DOS*	**WIN 95, 98**	**WIN NT4**	*Linux*
PCI, PCI/104	*A/D, D/A*	*NON-SERIAL*	*WIN3.x*	*WIN 98se, Me*	*WIN 2000, XP*	

Win32IRQ, aka IRQGen, provides generic IRQ support in Windows.

IRQGen.vxd is a Virtual Device Driver, or VxD, for Windows 9x/ME. IRQGen.sys is a Device Driver for Windows NT(including 2000 and XP). Functionally, they are identical, and for the purposes of this manual the term IRQGen Driver will be used to refer to the appropriate file for your operating system. Together with Win32IRQ.dll, the IRQGen Driver allows a program to easily respond to hardware interrupt requests, or IRQs.

Warning: Since Windows 2000 and XP made subtly incompatible changes to how PCI IRQs are handled, IRQGen should not be used for PCI IRQs under these operating systems. AIOWDM should be used instead.

Win32IRQ.DLL

Win32IRQ.dll is a Dynamic Link Library, or DLL, for all 32-bit Windows versions (Windows 9x, ME, NT 4, NT 2000, and NT XP). It provides a simple interface to the IRQGen Driver, allowing a program to easily respond to hardware interrupts. Four functions are exported by Win32IRQ.dll:

* InitGenDriver
* DetectIRQ
* AbortRequest
* SendEOI

To use these functions in a program, they must first be imported into that program. The simplest way to import these functions in Pascal is to include the Win32IRQ unit in your Uses clause, and in C to #include Win32IRQ.h. Additionally, in C, the library file CBIRQ.LIB (for C++ Builder) or VCIRQ.LIB (for Visual C++) must be linked to the program. This is accomplished by adding it to the project or editing the makefile. If you are not using the Win32IRQ header file and wish to import the functions manually, the necessary lines in Pascal would be:

```
function InitGenDriver(BaseAddress: DWORD; IRQ: BYTE; BusType: SmallInt; BusNumber:
BYTE; ClearOffset: WORD; Operation: BYTE): ByteBool; cdecl; external 'Win32IRQ.dll';
function DetectIRQ: ByteBool; cdecl; external 'Win32IRQ.dll';
function SendEOI: ByteBool; cdecl; external 'Win32IRQ.dll';
function AbortRequest: ByteBool; cdecl; external 'Win32IRQ.dll';
```

To import the functions manually in C, the necessary lines would be:

```
extern "C" __declspec(dllimport) unsigned char InitGenDriver(unsigned long BaseAddress,
unsigned char IRQ, short BusType, unsigned char BusNumber, unsigned short ClearOffset,
unsigned char Operation);
extern "C" __declspec(dllimport) unsigned char DetectIRQ(void);
extern "C" __declspec(dllimport) unsigned char SendEOI(void);
extern "C" __declspec(dllimport) unsigned char AbortRequest(void);
```

InitGenDriver

This function performs the initialization required to detect IRQs. It must be called before DetectIRQ and SendEOI can be called, although calling either of those functions first will only cause them to return a value of FALSE. The function takes six parameters: the card's base address, the IRQ level to monitor, the card's bus type and bus number, the offset from the base address that contains the latch to clear an IRQ, and the operation that must be performed to clear

the latch, which can either be READ_TO_CLEAR or WRITE_TO_CLEAR. The parameters BusType and BusNumber are only used in Windows NT. The header files Win32IRQ.PAS (for Pascal) and Win32IRQ.H (for C) contain the enumerated constants that can be passed in the BusType parameter. The two possible values are PCIBus and Isa. For Windows 95/98, the IRQGEN Driver ignores these two parameters and their values can be set to zero. The return result is TRUE if initialization was completed successfully, FALSE if not.

DetectIRQ
This function suspends the thread that it was called from and waits for an IRQ to occur on the IRQ level that was passed to InitGenDriver. DetectIRQ will return immediately, however, if an IRQ occurred anytime before the DetectIRQ call, as long as an IRQ level was initialized with InitGenDriver. Otherwise, DetectIRQ will not return until an IRQ occurs. If the program needs to continue running while waiting for DetectIRQ, a separate thread should be created from which to call DetectIRQ. The return result is TRUE if an IRQ was successfully detected on the given IRQ level, FALSE if no IRQ was initialized before the call. There are no parameters.

AbortRequest
This function instructs the Driver to abort a request to detect an interrupt. The Interrupt Service Routine will remain connected, so a call to DetectIRQ will resume waiting for an IRQ that was specified in InitGenDriver.

Win32IRQM

APPLIES TO						
USB	*ISA, PC/104*	**SERIAL**	*DOS*	**WIN 95, 98**	**WIN NT4**	*Linux*
PCI, PCI/104	*A/D, D/A*	*NON-SERIAL*	*WIN3.x*	**WIN 98se, Me**	**WIN 2000, XP**	

IRQGenM.vxd is a Virtual Device Driver, or VxD, for Windows9x/ME. IRQGenM.sys is a Device Driver for Windows NT(including NT 2000 and NT XP). Functionally, they are identical, and for the purposes of this manual the term IRQGenM Driver will be used to refer to the appropriate file for your operating system. Together with W32IRQM.dll, the IRQGenM Driver allows a program to easily respond to multiple hardware interrupt requests, or IRQs.

This driver is very similar to the IRQGen Driver. Use of this driver is slightly more involved, as the flexibility is provided to support more than one IRQ source simultaneously. When using only a single IRQ source it is recommended that the IRQGen driver and Win32IRQ be used instead of this driver.

W32IRQM.DLL
W32IRQM.dll is a Dynamic Link Library, or DLL, for all 32-bit Windows versions (Windows 9x, ME, NT 4, NT 2000, and NT XP). It provides a simple interface to the IRQGenM Driver, allowing a program to easily respond to multiple hardware interrupts. Five functions are exported by W32IRQM.dll:

* InitGenDriver
* DetectIRQ
* AbortRequest
* SendEOI
* DisconnectIRQ

To use these functions in a program, they must first be imported into that program. The simplest way to import these functions in Pascal is to include the W32IRQM unit in your Uses clause, and in C to include W32IRQM.H. Additionally, in C, the library file CBIRQM.LIB (for C++ Builder) or VCIRQM.LIB (for Visual C++) must be linked to the program. This is accomplished by adding it to the project or editing the makefile. If you are not using the W32IRQM header file and wish to import the functions manually, the necessary lines in Pascal would be:

```
function InitGenDriver(BaseAddress: DWORD; IRQ: BYTE; BusType: SmallInt;
BusNumber: BYTE; ClearOffset: WORD; Operation: BYTE): ByteBool; cdecl; external
'W32IRQM.dll';
function DetectIRQ(IRQ: Byte): ByteBool; cdecl; external 'W32IRQM.dll';
function AbortRequest(IRQ: Byte): ByteBool; cdecl; external 'W32IRQM.dll';
function SendEOI(IRQ: Byte): ByteBool; cdecl; external 'W32IRQM.dll';
function DisconnectIRQ(IRQ: Byte): ByteBool; cdecl; external 'W32IRQM.dll';
```

To import the functions manually in C, the necessary lines would be:

```
extern "C" __declspec(dllimport) unsigned char InitGenDriver(unsigned long BaseAddress,
unsigned char IRQ, short BusType, unsigned char BusNumber, unsigned short ClearOffset,
unsigned char Operation);
extern "C" __declspec(dllimport) unsigned char DetectIRQ(unsigned char IRQ);
extern "C" __declspec(dllimport) unsigned char AbortRequest(unsigned char IRQ);
extern "C" __declspec(dllimport) unsigned char SendEOI(unsigned char IRQ);
extern "C" __declspec(dllimport) unsigned char DisconnectIRQ(unsigned char IRQ);
```

InitGenDriver
This function performs the initialization required to detect IRQs. It must be called before DetectIRQ, DisconnectIRQ, and SendEOI can be called, although calling either of those functions first will only cause them to return FALSE. The function takes six parameters: the card's base address, the IRQ level to monitor, the card's bus type and bus number, the offset from the base address that contains the latch to clear an IRQ, and the operation that must be performed to clear the latch, which can either be READ_TO_CLEAR or WRITE_TO_CLEAR. The parameters BusType and BusNumber are only used in Windows NT. The header files W32IRQM.PAS (for Pascal) and W32IRQM.H (for C) contain the enumerated constants that can be passed in the BusType parameter. The two possible values are PCIBus and Isa. For Windows 95/98, the IRQGENM Driver ignores these two parameters and their values can be set to zero. The return result is TRUE if initialization was completed successfully, FALSE if not.

DetectIRQ
This function suspends the thread that it was called from and waits for an IRQ to occur on the IRQ level specified by the parameter IRQ. DetectIRQ will return immediately, however, if an IRQ occurred anytime before the DetectIRQ call, as long as an IRQ level was initialized with InitGenDriver. Otherwise, DetectIRQ will not return until an IRQ occurs. If the program needs to continue running while waiting for DetectIRQ, a separate thread should be created from which to call DetectIRQ. The only parameter is the IRQ number to be monitored. The return result is TRUE if an IRQ was successfully detected on the given IRQ level, FALSE if no IRQ was initialized before the call.

AbortRequest
This function instructs the Driver to abort a request to detect an interrupt on the IRQ level specified by the parameter IRQ. The Interrupt Service Routine will remain connected, so a call to DetectIRQ will resume waiting for that IRQ. The only parameter is the IRQ number to abort.

DisconnectIRQ
This function disconnects the Interrupt Service Routine that was connected with InitGenDriver. It should be called when the IRQ number specified no longer needs to be monitored. The only parameter is the IRQ number to disconnect. The return result is TRUE if the ISR was successfully disconnected, FALSE if not.