Introduction

This manual provides a single reference guide to all of the functionality provided by the AIOWDM driver package. This driver supports several families APIs for using PCI-based plug and play devices, including direct register access, watchdog timer control, and change-of-state IRQ handling.

This document describes the API's functions in sections to better distinguish which ones apply to YOUR device.

All PCI-compatible devices, other than serial boards, use this AIOWDM driver. This includes PCI, PCI Express, PCI-104, PC/104+, and many others.

USB devices, including PICO, USBP, USB/104, etc., use the AIOUSB driver and documentation, located in the "USB Software Reference.pdf" manual.

ISA devices, including PC/104, use a variety of drivers and utilities as documented in the "non PnP Software Reference".

The top of each section will include a header, like the following.

General Information				
Change-of-State IRQs	***Generic IRQ***	*Any Register Access*	*Watchdog Boards*	*Board Discovery*

Please check the header to make sure you need to bother reading any given section, based on which card you're working with. Black, bold text means "the following section is applicable to these categories of boards or functionality." This driver has many functions, and it might seem a little overwhelming. Make sure you don't get bogged down trying to learn entire families of functions that have nothing to do with your product!

General Information				
Change-of-State IRQs	*Generic IRQ*	*Any Register Access*	*Watchdog Boards*	*Board Discovery*
Useful to All PCI Related Tasks				

General Information

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, Visual C++, and Visual BASIC. Visual BASIC doesn't have true pointers, and thus pointer-using functions are declared in Visual BASIC to always have a non-null pointer.

For function signatures or prototypes in any specific language, please refer to the language-specific source files (AIOWDM.h, AIOWDM.cs, AIOWDM.vb, etc.).

We also provide a managed code wrapper DLL for easy use of the driver in .NET languages. Consult the .NET section of this manual for more information.

There are two ways to use the functions in this driver: A simple way, and a flexible way. If you know at design time that you will never have more than one PCI or PCI-Express based card (any card that uses

AIOWDM as its driver) in your system, use the simple way. If you will ever need to support more than a single AIOWDM-using card in the system simultaneously, you need to implement the more flexible interface.

The difference lies in how you acquire the parameter "CardNum". CardNum is a handle into the list of AIOWDM-using cards detected in the system.

In a simple setup with only one AIOWDM-using card, CardNum will always be Zero, being the first and only card in the system.

If you have two or more AIOWDM-using cards you will need to determine which card is installed at which CardNum index, using the API's flexible GetNumCards() and QueryCardInfo() functions. Please see the "Flexible Access" section of the manual for detailed information on using multiple cards in a system.

Once you have your CardNum squared away, whether it be simply Zero (0) or the specific CardNums determined via QueryCardInfo(), the way you use the rest of the AIOWDM functions is the same.

Register Level Access				
Change-of-State IRQs	*Generic IRQ*	***Any Register Access***	*Watchdog Boards*	*Board Discovery*
Virtually every program will use one or more of these functions				

RelInPortB
RelInPort
RelInPortL
RelOutPortB
RelOutPort
RelOutPortL

All of these functions take a CardNum as the first parameter, and a register offset as the second parameter.

The "In" family reads from the specified device's specified register and returns the value (a Byte, a Word, or a 32-bit Unsigned value) requested. These functions perform an operation equivalent to the "IN AX, DX" assembler instruction, or the READ_PORT_UCHAR Windows' kernel macro.

The "Out" family takes a third parameter (a Byte, a Word, or a 32-bit Unsigned value) and writes it to the specified device's specified register. These functions perform an operation equivalent to the "OUT DX, AX" assembler instruction, or the WRITE_PORT_UCHAR Window's kernel macro.

So, to read a Word from the first card's register at offset 2:
```
value = RelInPort(0, 2);
```

To read a Byte from the second card's register at offset 12:
```
value = RelInPortB(1, 12);
```

To write a 32-bit value to the first card's register at offset zero:
```
RelOutPortL(0, 0, value);
```

Example Function Prototype:
UInt16 RelInPortB(Uint32 CardNum, UInt32 Register);

A note about error handling:
All of these functions can return an error value of "0xAA55", which means the underlying driver (AIOWDM.SYS) is not installed correctly. Because 0xAA55 is a valid return value from the Word and 32-bit functions, it is useful to issue a single Byte sized read from an innocuous address at the init-time of your program to check for this error: no byte read function can return a 16-bit value unless it is this error, and if any function returns this error, all functions will, every time, until the driver is installed correctly. This also means, if you perform this byte-read init-time error check, no further error handling is needed in your program for these functions.

Operating system migration: Moving legacy applications to Windows 7
The following register access functions are deprecated:

InPortB()
InPort()
InPortL()
OutPortB()
OutPort()
OutPortL()

These functions do not use CardNum as the first parameter, nor the register offset as the second. Instead, they are NON PLUG AND PLAY, and use an absolute address into the I/O memory space as their first parameter (which then eliminates the second).

For example, instead of
```
value = RelInPortL( CardNum, RegisterOffset );
```

these functions would use
```
value = InPortL( CardBaseAddress + RegisterOffset );
```

Because of this "CardBaseAddress" business, if the card's plug-and-play assigned base address changes, your code needs to know how to handle this.

To support this we implemented QueryCardInfo() in a way that allows your program to acquire the actual absolute address of the card, which older code used to do every time your program ran.

Before even the QueryCardInfo technique customers were using the non-plug-and-play ACCES32 driver which doesn't support QueryCardInfo() at all. In order to get the absolute base address of a card in programs written *that* Int32 ago the software would query a set of Windows Registry keys containing an array of structures called "PCI_COMMON_CONFIG" which then had to be parsed to find the base address. Additionally, this registry information was only updated by running a utility called PCIFind and its underlying kernel driver NTioPCI.SYS, none of which works in modern operating systems!

In summary, AIOWDM eliminates the need to know the address of your card. It eliminates the NTioPCI.SYS driver running at every start-up of your system. It eliminates the PCIFind.exe utility copying the information from NTioPCI.SYS into the software-readable Windows registry locations every time the computer boots. It eliminates the need for applications to query this array of structures from the registry and wade through the data to determine a base address of the card.

For systems using only a single PCI card through AIOWDM, all of this simplification is achieved just by changing from "`address+offset`" to "`0, offset`" in calls to the API.

If you have a single card system, you can migrate from ACCES32 or the QueryCardInfo() techniques to the modern plug-and-play API by making the following changes to your program:

1) Locate your init code. This code either calls QueryCardInfo() or accesses the "HKLM/Software/PCIFind/Parameters" registry key.

2) Delete all of this init code. Really.

3) Search and replace all instances of "InPort" and "OutPort" in your program with "RelInPort" and "RelOutPort" respecively. Note, this changes "InPortB" and "OutPortL" to "RelInPortB" and "RelOutPortL", etcetera, at the same time.

4) Search and replace all instances of your base address variable plus an offset with zero comma the offset. For example, if the orignal code had the following line:

```
u32bData = InPortL( BASE + 3 );
```
it will now be, after the search and replace mentioned in step 3:
```
u32bData = RelInPortL( BASE + 3 );
```
Now perform a replace of "(BASE + " with "(0, ". This will transform the line into:
```
u32bData = RelInPortL( 0, 3 );
```
which is perfect and correct for the new, plug-and-play, functions.

If you are forced to use the "flexible" method of determining your CardNums because you have more than one AIOWDM-using device in your system, use the variable you've chosen to hold your detected CardNum instead of the "0": replace "(BASE + " with "(CardNumVariable, ". This would yield:
```
u32bData = RelInPortL( CardNumVariable, 3 );
```

These few steps will completely convert from the older, absolute address based API to the new plug-and-play API. Just make sure you're careful with your replace operations not to impact calls to non AIOWDM functions!

WaitForIRQ()
UInt32 WaitForIRQ(Int32 CardNum);

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()
UInt32 AbortRequest(Int32 CardNum);

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.

Warning: 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()
UInt32 CloseCard(Int32 CardNum);

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()
UInt32 COSWaitForIRQ(Int32 CardNum, UInt32 PPIs, void *pData);

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))".

AbortRequest()
CloseCard()
Both of these functions, described in the “Generic IRQ Handling” section, also apply to COS IRQ Handling.

WDGInit()
UInt32 WDGInit(Int32 CardNum);

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()
UInt32 WDGHandleIRQ(Int32 CardNum, UInt32 Action);

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()
double WDGSetTimeout(Int32 CardNum, double Milliseconds, double MHzClockRate);

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()
double WDGSetResetDuration(Int32 CardNum, double Milliseconds, double MHzClockRate);

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()
UInt32 WDGPet(Int32 CardNum);

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()
double WDGReadTemp(Int32 CardNum);

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()
UInt32 WDGReadStatus(Int32 CardNum);

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()
UInt32 WDGStart(Int32 CardNum);

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()
UInt32 WDGStop(Int32 CardNum);

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.

AbortRequest()
CloseCard()
Both of these functions, described in the “Generic IRQ Handling” section, also apply to Watchdog IRQ Handling.

EmergencyReboot()
UInt32 EmergencyReboot(void);

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).

When using more than one AIOWDM-using card in your system your program should always begin by calling the two functions described 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()
Int32 GetNumCards(void);

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()
UInt32 QueryCardInfo(Int32 CardNum, UInt32 *pDeviceID, UInt32 *pBase, UInt32 *pNameSize, unsigned char *pName);

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.

.NET Programming Wrapper				
Change-of-State	*Generic IRQ*	*Any Register Access*	*Watchdog Boards*	*Board Discovery*
an easy way to use the driver from Managed code environments like C# or VB.NET				

Microsoft .NET languages attempt to lock down the programing environment to prevent certain types of security flaws from being introduced. This is called “Managed” programming, and really refers to the fact that these languages are very high-level scripting languages that perform much of the low-level programming chores for the developer. This can be a good thing (more secure, easier) but it can also be a drawback (larger code that executes slowly, difficulty integrating with hardware, quirks when integrating with other languages).

This “managed code” environment prevents .NET code from calling directly into certain types of drivers and DLLs, like the DLLs used to control data acquisition hardware in other languages, without violating the “managed” wrapper. To avoid this violation, we have provided a C# language wrapper for the driver DLLs.

The PCI Driver API, AIOWDM.DLL, is wrapped up in AIOWDMNet.DLL. This DLL is simply a little piece of code written in C# that marshals the parameters used into forms .NET is more comfortable with calling while “managed”. The full source is provided under your installation path’s /win32 directory, so you can take a look at it if you’d like.

This AIOUSBNet.DLL replaces certain file types used in other languages, things like “header files” “lib files” “interface files” etcetera. This type of .NET DLL is often referred to as a “Class Library” – every function from our AIOUSB.DLL is provided in the form of a .NET compatible “Class”.

The only provided support at this time is for 32-bit systems. So, the first tip in this guide:

1) Make sure your Application target development system is “x86", not “any”

> Please note: some versions of Visual Studio may not have a convenient way to set the x86 configuration (they are coded always to “any”). Here’s an article on how to modify your project file in those cases: http://social.msdn.microsoft.com/Forums/nl-BE/vblanguage/thread/d4fa83dc-eed1-4ead-96a1-78bbd9ba6d3a

These samples were built using Visual Studio 2010, but can be converted to compile in older versions.

2) Create a brand new project in your version, then copy and paste the source code into that new project to build our VS2010 code in your older version.

3) If you’re rebuilding one of our Class Libraries (AIOUSBNet.DLL for example), make sure you select a Class Library Project when you create your new project to get all the settings correct.

4) The Class Library DLL can be made much more convenient to use if you install it into the GAC (Global Assembly Cache). This process can be difficult, but we made it easy – Check out the sub-project in the AIOWDMNet.DLL solution which will create an installer .MSI file for you. By simply building this project and running the resultant .MSI file, the dll and settings will be properly installed into the GAC.

As always, check for the latest versions of our code at our website, and feel free to chat, email, or call for technical support. We’re here to help!