ACCES I/O PRODUCTS, INC. **USB Software Reference Manual**

V2.11

INTRODUCTION

This manual acts as a reference for the USB function driver, "AIOUSB", and other provided software drivers and utilities that apply to our *non-serial-port* USB products.

***Serial** products simply appear as "COM" ports and are operated using the built-in Windows serial interfaces, and thus programming for them is documented by Microsoft.*

Note: Although this document may be a useful reference when programming for Linux or OSX, please refer to the provided documentation trees in those distribution directories for complete information.

How to use this reference

First, read the entirety of this introduction.

Next, read the source code of one of our sample programs in the programming language of your choice, and refer to this manual for the description of each API used, if needed.

Because the AIOUSB driver is shared among all of our USB products, not all of the API information in this manual will apply to your specific hardware.

In addition to providing a quick reference table at the start of each API function, certain quick-reference or pertinent facts will be presented below the table, giving you valuable insight into the quirks or pitfalls you may encounter.

Here's a breakdown of what each category description in the table means:

DIGITAL INPUTS:	TTL / CMOS / LVTTL (DI), and Isolated Input (IDI, II) types all included here. AIOUSB treats all boolean inputs generically as "bits". Contrast with Buffered DI, DO, DIO, below.
DIGITAL OUTPUTS:	TTL / CMOS / LVTTL and Isolated Output (RO, IDO) types are all included here. AIOUSB treats all boolean outputs generically as "bits". Contrast with Buffered DI, DO, DIO, below.
ANALOG INPUTS:	All inputs using "Analog-to-Digital Converter" chips. These boards accept analog signals (voltage, current), and create data the computer can use. Note there are two sub-types of Analog Inputs, those belonging to analog input devices (the USB-AIx family), and those inputs provided as a convenience on otherwise non-analog-input products. Most API functions in this category only apply to the USB-AIx family of devices.
ANALOG OUTPUTS:	All outputs that produce analog signals, including voltage and current. Note there are two sub-types of Analog Outputs in our current USB product lineup: Waveform and DC. Most API functions in this category only apply to the "Waveform" capable USB-DA12-8A.
BUFFERED DI, DO, DIO:	"High-Speed" digital "Bus" cards. This category applies only to boards that use "bulk" USB to achieve speeds higher than the ~4000/second transaction rate would otherwise allow, namely the USB-DIO-16H family.
COUNTER TIMERS:	This category applies to counter-timer and other frequency devices, such as the ever popular 8254. Generally used to count, measure, or produce frequencies or pulse trains.

Note: many models are “Multi-Function” and use API functions from more than one of these categories, and some specific API functions apply to all devices.

How to use the AIOUSB Driver

AIOUSB provides a standard interface for all Data Acquisition USB devices. Each specific USB device will use a subset of the driver calls listed below, based on its specific capabilities and needs.

USB devices are plug-and-play. Every time the driver detects a new USB device it assigns it a new, unused, Device Index. This is a temporary id, and will be different each time any given hardware device is detected.

When using a single USB Device - ever:

You can choose to ignore the specific Device Index assigned by the driver, and instead refer to the device by the constant “diOnly” (FFFFFFFD hex). This value tells the driver “Whichever Device Index exists, that’s the one I want to talk with”. Most of our sample programs operate this way.

Please note: this is the *simplest* way to code, but it is not very future-proof. If you need to add a second device years later, you’ll need to change your code.

When you are using, or might someday use, more than one device:

In all other cases the best practice is to assign each device a unique BOARD ID by writing a known BYTE value into location “0” of the user-accessible “Custom EEPROM”.

Run the provided eWriter.exe utility to set this “BOARD ID” byte, or write your own application using the CustomEEPROMWrite() function.

Avoid using FF hex or 00 hex as your BOARD IDs, as units can ship from the factory with these values.

Using a BOARD ID allows you to perform field-replacement of units by configuring the replacement board’s ID to the same as the unit it is replacing. The software doesn’t need to be informed.

Once you’ve written the BOARD ID into each unit, you use GetDeviceByEEPROMByte() to resolve the unique-and-unchanging BOARD ID into a detection-order-variable Device Index.

Your application may have code such as the following:

```
#define BOARD_ID_UUT    (0x01) // value written into location zero in EEPROM for UUT
#define BOARD_ID_ATE_AO (0x40) // value written into location zero in EEPROM for AO board
#define BOARD_ID_ATE_AI (0x80) // value written into location zero in EEPROM for AI board

enum {UUT, ATEANALOGOUTPUTS, ATEANALOGINPUTS, SOMEOTHERDEVICE};

// deviceIndex is a global array of UInt32
deviceIndex[UUT] = GetDeviceByEEPROMByte(BOARD_ID_UUT);
deviceIndex[ATEANALOGOUTPUTS] = GetDeviceByEEPROMByte(BOARD_ID_ATE_AO);
deviceIndex[ATEANALOGINPUTS] = GetDeviceByEEPROMByte(BOARD_ID_ATE_AI);
```

or, to make your code more or less readable:

```
diUUT = GetDeviceByEEPROMByte(BOARD_ID_UUT);
diATE_AO = GetDeviceByEEPROMByte(BOARD_ID_ATE_AO);
diATE_AI = GetDeviceByEEPROMByte(BOARD_ID_ATE_AI);
```

For Device Index parameters to the AIOUSB API you would then pass the appropriate variable.

About Error/Status return values

Most return values are Microsoft defined “Win32” error codes.

ERROR_SUCCESS (equal to 0) means no error occurred.

If the USB device has been unplugged, functions will return ERROR_DEVICE_REMOVED (equal to 1617 decimal). Call ClearDevices() to clear this condition.

If this state is cleared and the board is not reconnected, the error returned is ERROR_FILE_NOT_FOUND (equal to 2). You can also get ERROR_FILE_NOT_FOUND if you pass invalid filenames or paths to the API.

The full list of Windows error codes is in <winerror.h> or similar Microsoft-provided reference and can also be found on the web.

The set of error codes the AIOUSB DLL can produce is the combined set called out in our AIOUSB.DLL source code (provided) plus anything Microsoft’s lower-level code decides to throw our way.

DEVICE INDEPENDENT FUNCTIONS

This group of API calls can be used on any device, returning useful information about the device, dealing with various status, warning, or error conditions, talking with the onboard EEPROM, and more.

FUNCTIONS YOU CALL AT INITIALIZATION

GetDeviceByEEPROMByte

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Use this function to determine the assigned Device Index of a device in a deterministic way, especially if you’re controlling more than one device in the same system. Finds the device with the specified byte value at address 0x000 in the custom EEPROM area.

Delphi: `function GetDeviceByEEPROMByte(Data: Byte): LongWord; cdecl;`

Visual C: unsigned long GetDeviceByEEPROMByte(unsigned char Data);

C#: UInt32 GetDeviceByEEPROMByte(Byte Data);

Argument	Description
Data	A single byte, generally treated as a “user board ID”, previously written into a device using CustomEEPROMWrite, eWriter.exe, or other application software.

Returns: The current DeviceIndex of the card with matching “user board ID”. If there aren't any matching devices, returns FFFFFFFF hex.

Note: This function only looks in the onboard EEPROM at location “0”. “Board ID” is arbitrary, but you should avoid 0x00 and 0xFF, since those can match uninitialized EEPROMs. If there are multiple matching devices, returns the **first** one's device index (0-31) and sets the last error code to ERROR_DUP_NAME. If there's one matching device, returns its device index and sets the last error code to ERROR_SUCCESS. (You can get the last error code with the Win32 API call GetLastError().)

GetDevices

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Optional if you know you'll only *ever* have one device in your system at a time					

Purpose: This function will provide a bit-map of which Device Indices were detected by the driver.

Delphi: function GetDevices: LongWord; cdecl;

Visual C: unsigned long GetDevices(void);

C#: UInt32 GetDevices();

Returns: Returns a 32-bit bit-mask. Each bit set to "1" indicates an AIOUSB device was detected at a device index corresponding to the set bit index. For example, if the return is 0x00000104, then device indices 2 and 8 are detected devices.
Returns 0 if no devices found (which may mean the device is not installed properly, or you are using an incorrect DLL version.)

Notes: This does not return one device index; it returns a pattern of bits indicating all valid device indices.

This also prevents detection of more than 32 AIOUSB devices on one computer simultaneously. Let us know if this is of any concern for your application.

Most applications using a single device can skip this call. Use diOnly (0xFFFFFFFD) as your DeviceIndex instead.

GetDeviceByEEPROMByte() is the preferred discovery mechanism, and GetDevices() is being phased out.

QueryDeviceInfo

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Optional if you know you'll only *ever* have one *model* in your system at a time					

Purpose: This function provides various data about the card at a given Device Index.

Delphi: function QueryDeviceInfo(DeviceIndex: LongWord; pPID: PLongWord; pNameSize: PLongWord; pName: PChar; pDIOBytes, pCounters: PLongWord): LongWord; cdecl;

Visual C: unsigned long QueryDeviceInfo(unsigned long DeviceIndex, unsigned long *pPID, unsigned long *pNameSize, char *pName, unsigned long *pDIOBytes, unsigned long *pCounters);

C#: UInt32 QueryDeviceInfo(UInt32 DeviceIndex, out UInt32 pPID, ref UInt32 pNameSize, out String Name, out UInt32 pDIOBytes, out UInt32 pCounters)

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.

pPID	Pointer to PID ("Product ID"), a number indicating which hardware device was found at the specified DeviceIndex. Each device has a model-specific Product ID, see the hardware manual for your devices. This variable is set by the function, and does not need to be initialized prior to use. You can pass "null" if you don't need to know the PID.
pNameSize	Pointer to NameSize. Pass in the length of your Name buffer (in bytes) to prevent buffer overruns, and the function will set this to the actual name length. If your buffer is too small, the Name data will be truncated. You can pass "null".
pName	Pointer to Name, a byte array variable. This is an array of characters, not a null-terminated string. The length of the string is passed back via pNameSize. Set by the driver to the model name of the specified device. You can pass "null".
pDIOBytes	Pointer to DIOBytes. Set by the driver to the number of digital input/output byte-groups. You can pass "null"
pCounters	Pointer to Counters. Set by the driver to the number of 8254-style counter/timer groups. You can pass "null".

Returns: Windows standard "Win32" error codes. 0 means "Success".

Notes: It is common to call this function with "null" for most parameters. Most applications need only the DeviceIndex and pPID, simply to verify which device type is present at the DeviceIndex given. The additional parameters allow a more user-friendly experience if your program supports multiple card types.

GetDeviceSerialNumber

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Provides the on-board nonvolatile serial number.

Delphi: function GetDeviceSerialNumber(DeviceIndex: LongWord; var pSerialNumber: Int64): LongWord; cdecl;

Visual C: unsigned long GetDeviceSerialNumber(unsigned long DeviceIndex, unsigned __int64 *pSerialNumber);

C#: UInt32 GetDeviceSerialNumber(UInt32 DeviceIndex, out UInt64 pSerialNumber);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; either diOnly or a specific device's Device Index.
pSerialNumber	Pointer to an 8-byte (64-bit) value to fill with the serial number.

Returns: Windows standard "Win32" error codes. 0 means "Success".

Notes: This serial number *is not related* to the serial number on the physical label of the device.

FUNCTIONS YOU CALL AT DEINITIALIZATION

AIOUSB_CloseDevice

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Applies to every device, but rarely useful					

Purpose: Explicitly closes handles to a device.

Delphi: function AIOUSB_CloseDevice(DeviceIndex: LongWord): LongWord; cdecl;

Visual C: unsigned long AIOUSB_CloseDevice(unsigned long DeviceIndex);

C#: UInt32 AIOUSB_CloseDevice(UInt32 DeviceIndex);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; either diOnly or a specific device's Device Index.

Returns: Windows standard "Win32" error codes. 0 means "Success".

Notes: Used in version 8.xx of the AIOUSB driver package (WinUSB) to support multi-process programming. Deprecated as of 9.xx (CyUSB) AIOUSB drivers.

ClearDevices

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Use when you get errors from the device being unplugged unexpectedly					

Purpose: Closes handles and clears records of unplugged devices.

Delphi: function ClearDevices: LongWord; cdecl;

Visual C: unsigned long ClearDevices(void);

C#: UInt32 ClearDevices();

Returns: Windows standard "Win32" error codes. 0 means "Success".

Notes: Use this function to clear out the "ghost" DeviceIndices. Typically called when a device removal has been detected, either by the "Windows Message" method, or because API functions are returning "ERROR_DEVICE_REMOVED" or "ERROR_FILE_NOT_FOUND".

ResolveDeviceIndex

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Returns a device index from 0-31 corresponding to the "predefined constant" passed in, or FFFFFFFF hex if it can't be resolved.

Delphi: function ResolveDeviceIndex(DeviceIndex: LongWord): LongWord; cdecl;

Visual C: unsigned long ResolveDeviceIndex(unsigned long DeviceIndex);

C#: UInt32 ResolveDeviceIndex(UInt32 DeviceIndex);

Argument **Description**

DeviceIndex	Pass in diOnly or diFirst.

Returns: Returns the actual, detected, DeviceIndex that the parameter is interpreted as

Notes: This function is provided primarily for use during debugging.

GetDeviceByEEPROMData

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Finds a device based on custom EEPROM data. Like GetDeviceByEEPROMByte(), except that it can look for more than a single byte, and at any location in the custom EEPROM area. Deterministically translates from non-volatile EEPROM-stored data into a DeviceIndex

Delphi: function GetDeviceByEEPROMData(StartAddress, DataSize: LongWord; pData: PByte): LongWord; cdecl;

Visual C: unsigned long GetDeviceByEEPROMData(unsigned long StartAddress, unsigned long DataSize, unsigned char *pData);

C#: UInt32 GetDeviceByEEPROMData(UInt32 StartAddress, UInt32 DataSize, [In, Out] Byte[] pData);

Argument	**Description**
StartAddress	The address of the beginning of the block to look for in the custom EEPROM area
DataSize	The number of consecutive bytes to look for in the EEPROM
pData	A pointer to the block of data to look for, generally treated as a "user board ID", previously written into a device using CustomEEPROMWrite, eWriter.exe, or other application software.

Returns: The current DeviceIndex of the card with matching "user board ID". If there aren't any matching devices, returns FFFFFFFF hex.

Notes: If there are multiple matching devices, returns the **first** one's device index (0-31) and sets the last error code to ERROR_DUP_NAME. If there's one matching device, returns its device index and sets the last error code to ERROR_SUCCESS. (You can get the last error code with the Windows API call GetLastError().)

OTHER COMMON FUNCTIONS

In general CustomEEPROMWrite() and CustomEEPROMRead() can be used at any time in your application, but most applications will not need to.

CustomEEPROMWrite

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: This function writes to the user-accessible EEPROM

Language	**Signature**
Delphi:	`function CustomEEPROMWrite(DeviceIndex: LongWord; StartAddress: LongWord; DataSize: LongWord; Data: Pointer): LongWord; cdecl;`
Visual C:	`unsigned long CustomEEPROMWrite(unsigned long DeviceIndex, unsigned long StartAddress, unsigned long DataSize, void *pData);`
C#:	`UInt32 CustomEEPROMWrite(UInt32 DeviceIndex, UInt32 StartAddress, UInt32 DataSize, [In, Out] Byte[] Data);`

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
StartAddress	number from 0x000 to 0x1FF of the first EEPROM byte you wish to write to.
DataSize	number of custom EEPROM bytes to write. The last custom EEPROM byte is 0x1FF, so StartAddress plus DataSize can't be greater than 0x200.
Data	pointer to the start of a block of bytes to write to the custom EEPROM area.

Returns: Windows standard "Win32" error codes. 0 means "Success".

Notes: If you write a one unique byte to StartAddress zero you can use GetDeviceByEEPROMByte() to robustly determine the DeviceIndex of your devices. Refer to GetDeviceByEEPROMByte() for more information.

EWriter.exe is provided to perform this and other EEPROM functions, so your code doesn't have to.

CustomEEPROMRead

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: This function reads data previously written by CustomEEPROMWrite() or application software.

Delphi:	function CustomEEPROMRead(DeviceIndex: LongWord; StartAddress: LongWord; var DataSize: LongWord; Data: Pointer): LongWord; cdecl;
Visual C:	unsigned long CustomEEPROMRead(unsigned long DeviceIndex, unsigned long StartAddress, unsigned long *DataSize, void *pData);
C#:	UInt32 CustomEEPROMRead(UInt32 DeviceIndex, UInt32 StartAddress, ref UInt32 DataSize, [In, Out] Byte[] Data);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
StartAddress	number from 0x000 to 0x1FF of the first EEPROM byte you wish to write to.
DataSize	number of custom EEPROM bytes to write. The last custom EEPROM byte is 0x1FF, so StartAddress plus DataSize can't be greater than that.
Data	pointer to the start of a block of bytes to write to the custom EEPROM area.

Returns: Windows standard "Win32" error codes. 0 means "Success".

Notes: Devices are shipped with 0x00 in all custom EEPROM bytes.

DIGITAL INPUT / OUTPUT

Our driver supports two broad classes of Digital I/O: "Fast" or "Buffered" DIO, and "Slow" or "Normal" DIO.

"Normal" DIO on the USB bus means not-more-than 4000 transactions per second. Buffered DIO is described in the "Buffered DIO" section, and refers to digital bits capable of 8 to 40 MHz - or even faster - operations. Buffered DIO boards have both kinds of DIO on them, so you will need to refer to both sections, and the DIO_Configure*x* family of functions is used with both kinds of boards.

I/O Groups

Digital I/O (DIO) generally consists of several groups of 8 TTL/CMOS digital pins capable of being configured as either input or output, but only as a group.

Many devices act very similar to the venerable 8255 device, wherein the unit provides a 50-pin IDC header with 24 digital I/O lines often called Port A, Port B, and Port C, compatible with OPTO-22 module racks. Other devices may only provide two 8-bit ports, called Port 0 and Port 1, or, each bit may be selectable on a bit-by-bit basis as either input or output.

Each collection of pins able to be configured for input or output, but only together, is referred to as an "I/O Group."

Most devices of this type provide TTL/CMOS compatible pins. Other devices may provide three-volt signals, called 3.0VDC, 3.3VDC, or LVTTL, via either jumper selection, software selection, or factory option.

Regardless of the number of I/O Groups or TTL / CMOS / LVTTL voltages of the pins, these types of digital bits are referred to as "TTL DIO," "standard DIO," or simply "DIO."

Relays as DIO

Alternately, Digital Outputs may refer to solid-state relay or electro-mechanical relay outputs, while Digital Inputs may refer to optically- or electrically- isolated digital input bits. In both cases the voltage and current ranges far exceed the standard TTL/CMOS / LVTTL capability of "standrd DIO," and may be referred to collectively as "isolated digital bits," "Isolated DIO," "IDIO," or "IIRO." Further, inputs of these types may be called "IDI", "II", or "Isolated Inputs", while outputs may be referred to as "IDO", "RO", or "Isolated DO".

The AIOUSB API provides a single, unified collection of functions, or Application Programming Interface (API) to talk to all of these flavors of digital inputs and outputs, designated the "DIO_" API.

One stark difference between DIO and IDIO is the contrast in "I/O Groups". IDIO pins do not belong to any I/O Groups. Each isolated digital bit, be it input or output, is fixed to that state, and cannot be changed from one to the other.

Another difference is the useful speed of the bits. DIO bits can be read or driven as fast as the USB bus allows; generally one transaction (input or output) every 250usec under optimum conditions. IDO and RO bits *could* be driven at the same speeds, but because of the high-current drive, high-voltage capability, and / or electro-mechanical nature of the outputs, the fastest an output can *usefully* be driven is limited to once every 5 milliseconds or so. Similarly, isolated inputs, because of the isolation and (optional) filtering, rarely respond to input changes lasting less than ten milliseconds.

This distinction becomes an important factor when using functions such as DIO_WriteAll(). On a typical DIO board you very clearly have *n* I/O Groups, each of which has a byte, which you pass to the DIO_WriteAll function in an array, with each byte being the next group's data.

On IDIO / IIRO boards you still have the array of bytes, but do not know which bytes are the inputs or outputs.

In all cases outputs come first in the array for IDIO / IIRO devices. Conceptually this is because outputs *usefully* have two functions: output and read back, while inputs only *usefully* read.

Regardless of DIO vs IDIO, the outputs are always safe to read from and will return the latest data written; the inputs are safe to write to, and will simply ignore any data written.

DIO_Configure

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
The structure format changes based on how many bytes of data the card family supports					

Purpose: Configure Digital Input/Output "I/O Groups" for use as input or as output digital bits, and set the default state of any output bits. On some products can globally enable/tristate all DIO bits.

Delphi: function DIO_Configure(DeviceIndex: LongWord; Tristate: ByteBool; pOutMask: Pointer; pData: Pointer): LongWord; cdecl;

Visual C: unsigned long DIO_Configure(unsigned long DeviceIndex, unsigned char bTristate, void *pOutMask, void *pData);

C#: UInt32 DIO_Configure(UInt32 DeviceIndex, Byte Tristate, ref UInt32 OutMask, ref UInt32 Data);
UInt32 DIO_Configure(UInt32 DeviceIndex, Byte Tristate, ref Int16 OutMask, Byte[] Data);
UInt32 DIO_Configure(UInt32 DeviceIndex, Byte Tristate, Byte[] OutMask, Byte[] Data);
UInt32 DIO_Configure(UInt32 DeviceIndex, Byte Tristate, ref Byte OutMask, ref Byte Data);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
bTristate	TRUE causes all bits on the device to enter tristate (high-impedance) mode. FALSE removes the tristate. The tristate is changed after the remainder of the configuration has occurred. All devices with this feature power-on in the "tristate" mode at this time.
pOutMask	A pointer to array of bits; one bit per "I/O Group". Each "1" bit in the array indicates that the corresponding "I/O Group" of the device is to be configured as Output.
pData	A pointer to an array of bytes. Each byte is copied to the digital output ports on the device before the ports are taken out of tristate. Any bytes in the array associated with ports configured as input are ignored.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: In this context "I/O Group" means "a group of one or more DIO bits for which a single direction control bit determines the input vs output state for all the bits in the group."

Most cards use 1 bit per 8 DIO pins; others use 1 bit per DIO pin. Other groupings are possible: the USB-DIO-16H family uses 4 bits; one for 8-bit group A, one for 4-bit group B, and one each for 3-bit groups C and D.

When more than one bit is in each group "I/O Groups" are often referred to as "Ports". In these cases, the length of the array pointed to by pData should be the same number of bytes as the number of "I/O Group" control bits, even if a specific "I/O Group" contains fewer than 8 bits - significant bits are LSB aligned, and extra bits are ignored.

However, in the "one bit per group" case (USB-DIO-32I, for example), pData is effectively an array of *bits*, not bytes, as each bit is copied onto the output bits in a 1-to-1 fashion.

The sizes of the out mask and data for specific, representative, DIO boards are as follows:

	USB-DIO-32	USB-DIO-32I	USB-IIRO-xx	USB-Dxx16A	USB-DIO-96	USB-AIx
Out Mask	1 byte	4 bytes	1 byte	1 byte	2 bytes	1 byte
Data	4 bytes	4 bytes	4 bytes	4 bytes	12 bytes	2 bytes

DIO_ConfigureEx

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
The USB-DIO-16A family has only two tristate groups, "A", and "all other digital ports"					

Purpose: Configure Digital Input/Output "I/O Groups" for use as input or as output digital bits, and set the default state of any output bits. On some products can globally enable/tristate all DIO bits.

Delphi: function DIO_ConfigureEx(DeviceIndex: LongWord; pOutMask: Pointer; pData: Pointer; pTristateMask: Pointer): LongWord; cdecl;

Visual C: unsigned long DIO_ConfigureEx(unsigned long DeviceIndex, void *pOutMask, void *pData, void *pTristateMask);

C#: UInt32 DIO_ConfigureEx(UInt32 DeviceIndex, Byte[] pOutMask, Byte[] pData, Byte[] pTristateMask);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
pOutMask	A pointer to array of bits; one bit per "I/O Group". Each "1" bit in the array indicates that the corresponding "I/O Group" of the device is Output.
pData	A pointer to an array of bytes. Each byte is copied to the digital output ports on the device before the ports are taken out of tristate. Any bytes in the array associated with ports configured as input are ignored.
pTristateMask	A pointer to array of bits; one bit per "Tristate Group". Each "1" bit in the array indicates that the corresponding "Tristate Group" of the device is to be tristated.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Please refer to DIO_Configure for additional information.

DIO_ConfigurationQuery

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Normally your program will simply "remember" what it last sent to "DIO_Configure..."					

Purpose: Determine DIO port direction and tristate-related information about the device found at a specific DeviceIndex

Delphi: function DIO_ConfigurationQuery(DeviceIndex: LongWord; pOutMask: Pointer; pTristateMask: Pointer): LongWord; cdecl;

Visual C: unsigned long DIO_ConfigurationQuery(unsigned long DeviceIndex, void *pOutMask, void *pTristateMask);

C#: `UInt32 DIO_ConfigurationQuery(UInt32 DeviceIndex, [In, Out] Byte[] pOutMask, [In, Out] Byte[] pTristateMask);`

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
pOutMask	a pointer to the first element of an array of bytes; one byte per 8 ports or fraction. Each bit in the array will be set to "1" if the corresponding port is an Output, or "0" if it's an Input
pTristateMask	a pointer to the first element of an array of bytes; one byte per 8 tristate groups or fraction. Each bit in the array will be set to "1" if the corresponding tristate group is in tristate (high-impedance) mode, or a "0" if not

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: This function is not yet implemented on all devices. Please consult with the factory if you need to use it.

DIO_WriteAll

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
The most efficient method of writing to digital outputs					

Purpose: Write to all digital outputs on a device

Delphi: function DIO_WriteAll(DeviceIndex: LongWord; pData: Pointer): LongWord; cdecl;

Visual C: unsigned long DIO_WriteAll(unsigned long DeviceIndex, void *pData);

C#: UInt32 DIO_WriteAll(UInt32 DeviceIndex, ref UInt32 Data);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
pData	pointer to the first element of an array of bytes. Each byte is copied to the corresponding output byte. Bytes written to ports configured as inputs are ignored

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The number of bytes in the pData array needs to be the same as described for "pData" under DIO_Configure. This operation consumes one USB transaction, regardless of the number of bits on the device. Bits written to ports configured as "input" will be ignored. The USB-IIRO-xx and USB-IDIO-xx families' outputs are bytes 0 and 1 in the array; the inputs are bytes 2 and 3.

DIO_Write8

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
This function will give unexpected results unless you call DIO_WriteAll or DIO_Configure... first.					

Purpose: Write to a single byte-worth of digital outputs on a device

Delphi: function DIO_Write8(DeviceIndex, ByteIndex: LongWord; Data: Byte): LongWord; cdecl;

Visual C: unsigned long DIO_Write8(unsigned long DeviceIndex, unsigned long ByteIndex, unsigned char Data);

C#: UInt32 DIO_Write8(UInt32 DeviceIndex, UInt32 ByteIndex, Byte Data);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
ByteIndex	Number of the byte you wish to change.
Data	One byte. The byte will be copied to the port outputs. Each set bit will cause the equivalent port bit to be set to "1"

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

*Not**es:*** Bytes written to ports configured as "input" are ignored.

The USB-IIRO-xx and USB-IDIO-xx families' outputs are bytes 0 and 1 in the array; the inputs are bytes 2 and 3. This is true even for 8-bit models, and models with only inputs. I.e., the USB-II-8's input byte is located at Data[2].

Warning: This function is usually implemented in the DLL: the DLL remembers the most recently written DIO data for all ports, and merges the byte sent to DIO_Write8, then sends the actual hardware a DIO_WriteAll() command. Therefore, do not use this function simultaneously to control digital bits on a single device from multiple processes.

DIO_Write1

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
This function will give unexpected results unless you call DIO_WriteAll or DIO_Configure first.					

Purpose: Determine information about the device found at a specific DeviceIndex

Delphi: function DIO_Write1(DeviceIndex, BitIndex: LongWord; Data: ByteBool): LongWord; cdecl;

Visual C: unsigned long DIO_Write1(unsigned long DeviceIndex, unsigned long BitIndex, unsigned char bData);

C#: UInt32 DIO_Write1(UInt32 DeviceIndex, UInt32 BitIndex, Byte Data);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
BitIndex	Number of the bit you wish to change.
bData	TRUE will set the bit to "1", FALSE will clear the bit to "0"

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Bits written to ports configured as "input" are ignored.

The USB-IIRO-xx and USB-IDIO-xx families' outputs are bits 0-15; the inputs are bits 16-31. This is true even for 8-bit models, and models with only inputs. I.e., the USB-II-8's first input bit is located at BitIndex 16.

Warning: This function is usually implemented in the DLL: the DLL remembers the most recently written DIO data for all ports, and merges the bit sent to DIO_Write1(), then sends the actual hardware a DIO_WriteAll() command. Therefore, do not use this function simultaneously to control digital bits on a single device from multiple processes.

Also, on boards that do not need a DIO_Configure() call before outputs are operational, make sure to call DIO_WriteAll() at least once, to initialize the DLL's memory. You can call DIO_Configure() instead of DIO_WriteAll(), and doing so helps future-proof your code for compatibility with other devices.

DIO_ReadAll

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
The most efficient method of reading digital inputs.					

Purpose: Read all digital bits on a device, including read-back of ports configured as "output"

Delphi: function DIO_ReadAll(DeviceIndex: LongWord; pData: Pointer): LongWord; cdecl;

Visual C: unsigned long DIO_ReadAll(unsigned long DeviceIndex, void *Buffer);

C#: UInt32 DIO_ReadAll(UInt32 DeviceIndex, out UInt32 Data);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
pBuffer	pointer to the first element of an array of bytes. Each port will be read, and the reading stored in the corresponding byte in the array.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The number of bytes in the pBuffer array needs to be the same as described for "pData" under DIO_Configure. This operation consumes one USB transaction, regardless of the number of bits on the device.
The USB-IIRO-xx and USB-IDIO-xx families' inputs are bytes 2 and 3; the outputs' readback are bytes 0 and 1 in the array. This is true even for 8-bit models, and models with only inputs. I.e., the USB-II-8's input byte is located at Data[2].

DIO_Read8

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
	The driver performs a ReadAll and returns the selected byte				

Purpose: Read one byte of digital data from the device

Delphi: function DIO_Read8(DeviceIndex, ByteIndex: LongWord; Buffer: PByte): LongWord; cdecl;

Visual C: unsigned long DIO_Read8(unsigned long DeviceIndex, unsigned long ByteIndex, unsigned char *pBuffer);

C#: UInt32 DIO_Read8(UInt32 DeviceIndex, UInt32 ByteIndex, out Byte Data);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
ByteIndex	Number of the byte you wish to read.
pBuffer	Pointer to one byte. The byte will contain the reading or readback from the port specified by ByteIndex. Each set bit indicates the equivalent port bit read as "1"

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The USB-IIRO-xx and USB-IDIO-xx families' inputs are bytes 2 and 3; the outputs are bytes 0 and 1. This is true even for 8-bit models, and models with only inputs. I.e., the USB-II-8's first input byte is located at ByteIndex 2.

DIO_Read1

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
	The driver performs a ReadAll and returns the selected bit				

Purpose: Read one bit of digital data from a device.

Delphi: function DIO_Read1(DeviceIndex, BitIndex: LongWord; Buffer: PByte): LongWord; cdecl;

Visual C: unsigned long DIO_Read1(unsigned long DeviceIndex, unsigned long BitIndex, unsigned char *Buffer);

C#: UInt32 DIO_Read1(UInt32 DeviceIndex, UInt32 BitIndex, out Byte Data);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.

BitIndex	Number of the bit you wish to read.
pBuffer	Pointer to one byte; will be set to "1" or "0" to indicate the state of the bit read (or readback).

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The USB-IIRO-xx and USB-IDIO-xx families' inputs are bits 16-31; the outputs are bits 0-15. This is true even for 8-bit models, and models with only inputs. I.e., the USB-II-8's first input bit is located at BitIndex 16.

BUFFERED DIO

Our driver supports two broad classes of Digital I/O: "Fast" or "Buffered" DIO, and "Slow" or "Normal" DIO.

This section refers exclusively to "Buffered" DIO. "Normal" DIO is described in the "Digital Input / Output" section, and refers to digital bits capable of DC to 4kHz transaction rate operation.

Buffered DIO boards have both kinds of DIO on them, so you will need to refer to both sections, and the DIO_Configure family of functions is used in either case.

Buffered Digital I/O is currently supported only on the USB-DIO-16H family of products.

This product family can input, output, or either, at continuous speeds of at least 8MHz per 16-bit word, and burst speeds in excess of 40MHz (for the depth of the onboard FIFO).

To use the fast digital bits, issue a DIO_Configure(), DIO_StreamOpen(), DIO_StreamSetClocks(), loop while calling DIO_StreamFrame(), and call DIO_StreamClose() when finished.

DIO_StreamOpen

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	**BUFFERED DI, DO, DIO**	COUNTER TIMERS

Purpose: Configure a USB-DIO-16H family device for use as input or output

Delphi: function DIO_StreamOpen(DeviceIndex: LongWord; bIsRead: LongBool): LongWord; cdecl;

Visual C: unsigned long DIO_StreamOpen(unsigned long DeviceIndex, unsigned long bIsRead);

C#: UInt32 DIO_StreamOpen(UInt32 DeviceIndex, UInt32 bIsRead);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
bIsRead	boolean. TRUE will open a stream for reading, FALSE will open a stream for writing.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Call DIO_ConfigureEx() and DIO_StreamSetClocks() as well, before sending/receiving data using DIO_StreamFrame()

DIO_StreamClose

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Terminate the input or output operation on a USB-DIO-16H family device.

Delphi: function DIO_StreamClose(DeviceIndex: LongWord): LongWord; cdecl;

Visual C: unsigned long DIO_StreamClose(unsigned long DeviceIndex);

C#: UInt32 DIO_StreamClose(UInt32 DeviceIndex);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: For best results, should be called before your application closes.

DIO_StreamSetClocks

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Configure the highspeed port for external or internal clock source, and, if internal, the frequency thereof.

Delphi: function DIO_StreamSetClocks(DeviceIndex: LongWord; var ReadClockHz, WriteClockHz: Double): LongWord; cdecl;

Visual C: unsigned long DIO_StreamSetClocks(unsigned long DeviceIndex, double *ReadClockHz, double *WriteClockHz);

C#: UInt32 DIO_StreamSetClocks(UInt32 DeviceIndex, ref Double ReadClockHz, ref Double WriteClockHz);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
pReadClockHz	a pointer to an IEEE double-precision value indicating the desired frequency of an internal read clock.
pWriteClockHz	a pointer to an IEEE double-precision value indicating the desired frequency of an internal write clock.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Use "0" to indicate "External clock mode"

The Read and Write clock variables will be modified to indicate the Hz rate to which the unit will actually be configured: not all frequencies you can specify with a IEEE double can be achieved by the frequency generation circuit. Our DLL calculates the closest achievable frequency. If you're interested, you can consult the LTC6904 chipspec for details, or the provided source for the DLL.

The slowest available frequency from the onboard generator is 1kHz; the fastest *usable* is 40MHz (the limit of the standard FIFO); the fastest *useful for non-burst operation* is ~8MHz (the streaming bandwidth limit of the USB->digital interface logic and code is 8MHz minimum, often as high as 12MHz if your computer is well optimzed.)

DIO_StreamFrame

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
If opened as input, the FramePoints must be a multiple of 256 or you will generate error 31 "ERROR_GEN_FAILURE"					

Purpose: Send or Receive "fast" data across the DIO bus.

Delphi: function DIO_StreamFrame(DeviceIndex, FramePoints: LongWord; pFrameData: PWord; var BytesTransferred: DWord): LongWord; cdecl;

Visual C: unsigned long DIO_StreamFrame(unsigned long DeviceIndex, unsigned long FramePoints, unsigned short *pFrameData, unsigned long *BytesTransferred);

C#: UInt32 DIO_StreamFrame(UInt32 DeviceIndex, UInt32 FramePoints, [In, Out] Byte[] FrameData, out UInt32 ByteTransferred);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
FramePoints	number of WORD-sized points you wish to stream
pFrameData	pointer to the beginning of the block of data you wish to stream
BytesTransferred	pointer to a DWORD that will receive the amount of data actually transferred, in BYTEs

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: This function is used for either input or output operation. "Stream" can be interpreted "upload," "write," "read," "download," "send," "receive," or any similar term.

COUNTER / TIMERS

An important note about the CTR_8254 family of functions:

Each of these functions is designed to operate in one of two addressing modes. The parameter "BlockIndex" refers to 8254 chips, each of which contains 3 "Counters". CounterIndex refers to the counters inside the 8254s. In the primary addressing mode you specify the block and the counter. In the secondary addressing mode, you specify zero (0) for the block, and consider the counters to be addressed sequentially. That is, BlockIndex 3, CounterIndex 1 can also be addressed as BlockIndex 0, CounterIndex 10. The equation to determine the secondary, or sequential, CounterIndex given the primary or block values is as follows (they simply count consecutively):

$\text{CounterIndex}_{\text{sequential}} = \text{BlockIndex} * 3 + \text{CounterIndex}_{\text{Primary}}$

CounterIndex values associated with BlockIndex 0 are compatible with either addressing mode, there is no need to tell the driver which addressing mode you wish to use.

Specific, common, counter / timer tasks:

Frequency Generation: To generate a frequency using an 8254 wired in the "standard" way, use CTR_8254StartOutputFreq (). Refer to that function for more information.

Event Counting: To count up to 65535 events per 8254 wired in the standard way, mode counter 1 in mode 1, mode counter 0 in mode 0, and load counter 0 with "0". Make sure to not load any count value into counter 1. Read counter 0 to determine the number of times it has decremented since the last time you read it.

CTR_8254Mode

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Configure the mode of operation for a specified counter.

Delphi: function CTR_8254Mode(DeviceIndex, BlockIndex, CounterIndex, Mode: LongWord): LongWord; cdecl;

Visual C: unsigned long CTR_8254Mode(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned long Mode);

C#: UInt32 CTR_8254Mode(UInt32 DeviceIndex, UInt32 BlockIndex, UInt32 CounterIndex, UInt32 Mode);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
BlockIndex	Index of the 8254 chip you wish to control. See note at the beginning of this section.
CounterIndex	Index of the 8254 counter you wish to control. See note at the beginning of this section.
Mode	A number, from 0 to 5, specifying to which 8254 mode you want the specified counter to be configured.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Calling CTR_8254Mode() will halt the counter specified until CTR_8254Load() or CTR_8254ModeLoad() is called.

Neat trick:

Configuring a counter for Mode 0 will set that counter's output LOW.
Configuring a counter for Mode 1 will set that counter's output HIGH.

This can be used to convert a counter into an additional, albeit slow, digital output bit. The output pin will remain as configured until/unless the counter is "loaded" with a count value.

CTR_8254Load

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Load a counter with a 16-bit count-down value.

Delphi: function CTR_8254Load(DeviceIndex, BlockIndex, CounterIndex: LongWord; LoadValue: Word): LongWord; cdecl;

Visual C: unsigned long CTR_8254Load(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned short LoadValue);

C#: UInt32 CTR_8254Load(UInt32 DeviceIndex, UInt32 BlockIndex, UInt32 CounterIndex, UInt16 LoadValue);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
BlockIndex	Index of the 8254 chip you wish to control. See note at the beginning of this section.
CounterIndex	Index of the 8254 counter you wish to control. See note at the beginning of this section.
LoadValue	A number, from 0 to 65535, specifying how many counts to load in the counter specified.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: A load value of "0" is indistinguishable from a (hypothetical) load value of 65536.

Some modes do not support a load value of "1". Other modes support neither "1" nor "2" as load values.

CTR_8254ModeLoad

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Configure the mode of operation for a specified counter, and load that counter with a 16-bit count-down value.

Delphi: function CTR_8254ModeLoad(DeviceIndex, BlockIndex, CounterIndex, Mode: LongWord; LoadValue: Word): LongWord; cdecl;

Visual C: unsigned long CTR_8254ModeLoad(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned long Mode, unsigned short LoadValue);

C#: UInt32 CTR_8254ModeLoad(UInt32 DeviceIndex, UInt32 BlockIndex, UInt32 CounterIndex, UInt32 Mode, UInt16 LoadValue);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.

BlockIndex	Index of the 8254 chip you wish to control. See note at the beginning of this section.
CounterIndex	Index of the 8254 counter you wish to control. See note at the beginning of this section.
Mode	A number, from 0 to 5, specifying to which 8254 mode you want the specified counter to be configured.
LoadValue	A number, from 0 to 65535, specifying how many counts to load in the counter specified.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: This operation takes a single USB transaction, making it at least 250usec faster than issuing the two operations independently.

See CTR_8254Mode() and CTR_8254Load() for more notes.

CTR_StartOutputFreq

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Determine information about the device found at a specific DeviceIndex

Delphi: function CTR_StartOutputFreq(DeviceIndex, BlockIndex: LongWord; pHz: PDouble): LongWord; cdecl;

Visual C: unsigned long CTR_StartOutputFreq(unsigned long DeviceIndex, unsigned long CounterIndex, double *pHz);

C#: UInt32 CTR_StartOutputFreq(UInt32 DeviceIndex, UInt32 BlockIndex, ref double Hz);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
BlockIndex	Index of the 8254 from which you wish to output a frequency. Most devices only contain one 8254, and therefore BlockIndex should be "0" for those units.
pHz	pointer to a double precision IEEE floating point number containing the desired output frequency. This value is set by the driver to the *actual* frequency that will be output, as limited by the device's capabilities.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: This function requires that the individual counters in the 8254 specified be wired up as follows: 10MHz -> IN1 and OUT1 -> IN2. If the 10MHz is replaced with other frequencies, the pHz calculation may scale predictably.
This wiring is provided by the USB-CTR-15's "Standard Configuration Adapter" and is the permanent wiring configuration of most 8254s on our USB product line.

The USB-CTR-15 can output as many as 15 frequencies, if you use CTR_8254ModeLoad() - but if you use CTR_StartOutputFreq(), you can only achieve 5, on CTR2 of each of the 5 blocks. (Counters 2, 5, 8, 11, and 14 under the secondary naming convention. See note at the beginning of this section.)

CTR_8254Read

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Determine information about the device found at a specific DeviceIndex

Delphi: function CTR_8254Read(DeviceIndex, BlockIndex, CounterIndex: LongWord; pReadValue: PWord): LongWord; cdecl;

Visual C: unsigned long CTR_8254Read(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned short *pReadValue);

C#: UInt32 CTR_8254Read(UInt32 DeviceIndex, UInt32 BlockIndex, UInt32 CounterIndex, out UInt16 ReadValue);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
BlockIndex	Index of the 8254 chip you wish to control. See note at the beginning of this section.
CounterIndex	Index of the 8254 counter you wish to control. See note at the beginning of this section.
pReadValue	a pointer to a WORD in which will be stored the value latched and read from the specified counter

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The counts loaded cannot be read back. Only the "current count" is readable, and the current count does not initialize with the "(re-)load count value" until the first input clock occurs. Use CTR_8254ReadStatus() to distinguish between "the data read has never been loaded from the load count" (called a "null count" by the 8254 chip specification) and "the current count has been loaded, and decremented at least once."

CTR_8254ReadAll

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Currently only supported by the USB-CTR-15. Call if you need support for this function on a different device.					

Purpose: Reads the current count values from all counters.

Delphi: function CTR_8254ReadAll(DeviceIndex: LongWord; pData: PWord): LongWord; cdecl;

Visual C: unsigned long CTR_8254ReadAll(unsigned long DeviceIndex, unsigned short *pData);

C#: UInt32 CTR_8254ReadAll(UInt32 DeviceIndex, [In, Out] UInt16[] pData);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.

pData	a pointer to an array of WORDs in which will be stored the value latched and read from each counter, in order. Counter 0 in pData[0], etc.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Currently only supported by the USB-CTR-15. Call if you need support for this function on a different device.

The counts loaded cannot be read back. Only the "current count" is readable, and the current count does not initialize with the "(re-)load count value" until the first input clock occurs. Let us know if you need CTR_8254ReadStatusAll() functionality in your application.

CTR_8254ReadStatus

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
The meaning of the status is best described in the 8254 chip spec. Consult the CD\ChipDocs directory, or search the internet.					

Purpose: Determine information about the device found at a specific DeviceIndex

Delphi: function CTR_8254ReadStatus(DeviceIndex, BlockIndex, CounterIndex: LongWord; pReadValue: PWord; pStatus: PByte): LongWord; cdecl;

Visual C: unsigned long CTR_8254ReadStatus(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned short *pReadValue, unsigned char *pStatus);

C#: UInt32 CTR_8254ReadStatus(UInt32 DeviceIndex, UInt32 BlockIndex, UInt32 CounterIndex, out UInt16 ReadValue, out byte Status);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
BlockIndex	Index of the 8254 chip you wish to control. See note at the beginning of this section.
CounterIndex	Index of the 8254 counter you wish to control. See note at the beginning of this section.
pReadValue	a pointer to a WORD in which will be stored the value latched and read from the specified counter
pStatus	a pointer to a BYTE in which will be stored the status latched and read from the specified counter

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The meaning of the individual bits in the status byte is best described in the 8254 chip spec. Consult the CD\ChipDocs directory, or search the internet.

CTR_8254ReadModeLoad

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
The read value is acquired before the mode or write happens					

Purpose: Read, then Mode, then Load, a specified counter.

Delphi: function CTR_8254ReadModeLoad(DeviceIndex, BlockIndex, CounterIndex, Mode: LongWord; LoadValue: Word; pReadValue: PWord): LongWord; cdecl;

Visual C: unsigned long CTR_8254ReadModeLoad(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned long Mode, unsigned short LoadValue, unsigned short *pReadValue);

C#: UInt32 CTR_8254ReadModeLoad(UInt32 DeviceIndex, UInt32 BlockIndex, UInt32 CounterIndex, UInt32 Mode, UInt16 LoadValue, out UInt16 ReadValue);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The reading is taken before the mode and load occur.

This operation takes a single USB transaction, making it at least 500usec faster than issuing the three operations independently.

See CTR_8254LoadRead(), CTR_8254Mode(), and CTR_8254Load() for additional notes.

Some important notes about the following CTR_8254 functions:

These functions only apply to the USB-CTR-15

CTR_8254SelectGate() and **CTR_8254ReadLatched**() are used in measuring frequency. To measure frequency one must count pulses for a known duration. In simplest terms, the number of pulses that occur during 1 second is the frequency, in Hertz. In the USB-CTR-15 you can create a known duration by configuring one counter output to act as a "gating" signal for any collection of other counters. The other "measurement" counters will only decrement during the "high" side of the gate signal, which we can control.

So, to measure frequency:

1. Create a gate signal of known duration by calling CTR_ModeLoad() on one or more concatenated counters. Use Mode 3, Square Wave Generation, for *at minimum* the last counter in the chain. (Mode 2 should be used for counters earlier in the chain. Due to how the counter interprets odd vs even load values in Mode 3, avoid loading odd numbers for the Mode 3 counter, or a ±1 "factor" in the gate duration intrudes.)
2. Connect this gating signal to the gate pins of the "measurement" counter(s)
3. Connect the frequency(-ies)-to-be-measured to the input pin(s) of the same counter(s)
4. Configure each of those counters with Mode 2 and a known load value (0 is best)
5. Call CTR_8254SelectGate() to tell the board which counter is generating that gate

6. Call CTR_8254ReadLatched() once in a while to read the latched count values from all the "measurement" counters

In practice, it may not be possible to generate a gating signal of sufficient duration from a single counter. Simply concatenate two or more counters into a series (daisy-chain them), and use the last counter's output as your gating signal. This last counter in the chain should be reported as the "gate source" using CTR_8254SelectGate().

Once a value has been read from a counter using the CTR_8254ReadLatched() call, it can be translated into actual Hz by dividing the count value returned by the high-side-duration of the gating signal, in seconds. For example, if your gate is configured for 10Hz, the high-side lasts 0.05 seconds; if you read a delta-counts of 1324 via the CTR_8254ReadLatched() call, the frequency would be "1324 / 0.05", or 26.48KHz.

To be clear: the counters are count-down-only. Subtract the reading from the load value to determine the delta-counts for use in the calculation. I.e., if you use "0" as your load value, and read 56001 counts from the CTR_8254_ReadLatched() call, the delta-counts is 65536 - 56001 = 9535 counts. If your gate is configured for 10Hz, the input frequency would be "9535 / 0.05", or 190.7kHz.

CTR_8254SelectGate

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Currently only supported by the USB-CTR-15					

Purpose: This function selects a counter for use as the gate in frequency measurement on other counters, and starts the frequency measurement process.

Delphi: function CTR_8254SelectGate(DeviceIndex, GateIndex: LongWord): LongWord; cdecl;

Visual C: unsigned long CTR_8254SelectGate(unsigned long DeviceIndex, unsigned long GateIndex);

C#: UInt32 CTR_8254SelectGate(UInt32 DeviceIndex, UInt32 GateIndex);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
GateIndex	Index of the counter output being used as the gating signal, from 0-14.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Currently only supported by the USB-CTR-15

CTR_8254ReadLatched

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Currently only supported by the USB-CTR-15					

Purpose: Read the latest values latched by the board during frequency measurement.

Delphi: function CTR_8254ReadLatched(DeviceIndex: LongWord; pData: PWord): LongWord; cdecl;

Visual C: unsigned long CTR_8254ReadLatched(unsigned long DeviceIndex, unsigned short *pData);

C#: UInt32 CTR_8254ReadLatched(UInt32 DeviceIndex, [In, Out] UInt16[] pData);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
pData	A pointer to the first of an array of 15 WORDs, and one BYTE, in which will be stored the most recent values latched and read from the counters by the board. After the array of WORDs is one additional BYTE. This byte contains useful information when optimizing polling rates. If the value of the byte is "0", you're looking at old data, and are reading faster than your Gate signal is running.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Currently only supported by the USB-CTR-15

For advanced users: The value of the last byte returned is the number of Gate events that have occurred since you last read the data. "0" therefore means "old data, already seen, reading too often" and any number higher than "1" means "I could be getting data more often by reading faster by this factor"

ANALOG TO DIGITAL

The Analog to Digital inputs on the USB product line fall into two board categories: Analog Inputs provided by any board in the USB-AIx family, and Analog Inputs provided by any other USB board.

The USB-AIx family analog input products are optimized for performing analog-to-digital conversions, with a wide variety of configurations and options available to tune the inputs to fit your exact needs, including resolution, samples per second, calibration and reference options, elaborate signal-conditioning, etc.

The other category of analog inputs, however, are provided to monitor simple DC voltage levels with little or no flexibility in hardware or software.

The USB-AIx analog inputs ("AIx") can use all of the API functions presented in this section.

The non AIx inputs, on the other hand, can only use ADC_GetChannelV(), ADC_GetScanV(), ADC_GetChannel(), and ADC_GetScan(). No other functions described in this chapter are useful for analog input products other than members of the USB-AIx family.

ADC_GetScanV

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
The easiest way to read your data, but often can't achieve more than several hundred Hz, slower depending on options.					

Purpose:	Read the current voltage level on all inputs within the scan limits configured
Delphi:	function ADC_GetScanV(DeviceIndex: LongWord; pBuf: PDouble): LongWord; cdecl;
Visual C:	unsigned long ADC_GetScanV(unsigned long DeviceIndex, double *pBuf);
C#:	UInt32 ADC_GetScanV(UInt32 DeviceIndex, [In, Out] double[] pBuf);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
pBuf	Pointer to the first of an array of double precision IEEE floating point numbers. Each element in the array will receive the value read from the corresponding A/D input channel. The array must be at least as large as the number of A/D input channels your product contains (16, 32, 64, 96, or 128) - but it is safe to always pass a pointer to an array of 128 IEEE doubles. Only elements in the array corresponding to A/D channels actually acquired during the scan will be updated: start-channel through end-channel, inclusive. Other values will remain unchanged.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: This function converts input counts to voltages based on the range previously configured with ADC_Init or ADC_SetConfig.

On boards with A/Ds that don't support ADC_SetConfig(), it scans all channels, without oversampling.

It will take data at the configured number of oversamples or more, average the readings from the channels, and convert the counts to voltage.

This function performs many housekeeping USB transactions to make it simple to use. Many of these steps could be performed one-time, during init or de-init of the program. Doing so would improve the performance of the ADC_GetScanV() concept.

For more information on a faster but less convenient API that moves these housekeeping functions out of the acquisition loop, please refer to the section on ADC_GetFastScanV().

This convenience function should readily achieve 0 to 100 Hz operation, on up to 128 channels.

ADC_GetChannelV

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Note: slow					

Purpose:	Read one voltage input's current value
Delphi:	function ADC_GetChannelV(DeviceIndex, ChannelIndex: LongWord; pBuf: PDouble): LongWord; cdecl;
Visual C:	unsigned long ADC_GetChannelV(unsigned long DeviceIndex, unsigned long ChannelIndex, double *pBuf);

C#:	UInt32 ADC_GetChannelV(UInt32 DeviceIndex, UInt32 ChannelIndex, out double pBuf);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
ChannelIndex	number indicating which channel's data you wish to get
pBuf	a pointer to a double precision IEEE floating point number which will receive the value read

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: This function performs an ADC_GetScanV() and returns the specified channel's data.

Reading two channels using ADC_GetChannelV() is therefore more-than-twice as slow as using ADC_GetScanV.

This function is provided only for ease-of-use.

ADC_GetScan

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Returns data in "counts", which expose the "digital" nature of the conversion. Also slow, ADC_GetScanV()

Purpose: This simple function takes one scan of A/D data, in counts.

Delphi:	function ADC_GetScan(DeviceIndex: LongWord; pBuf: PWord): LongWord; cdecl;
Visual C:	unsigned long ADC_GetScan(unsigned long DeviceIndex, unsigned short *pBuf);
C#:	UInt32 ADC_GetScanV(UInt32 DeviceIndex, [In, Out] double[] pBuf);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
pBuf	Pointer to an array of WORDs. Each element in the array will receive the value read from the corresponding A/D input channel. The array must be at least as large as the number of A/D input channels your product contains (16, 32, 64, 96, or 128) - but it is safe to always pass a pointer to an array of 128 WORDs. Only elements in the array corresponding to A/D channels actually acquired during the scan will be updated: start-channel through end-channel, inclusive. Other values will remain unchanged.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: All counts are offset-binary coded, and left-justified (12-bit boards read count values 0x0000 through 0xFFF0, with 0x0000 being "minimum input voltage" and 0xFFF0 being "maximum input voltage"

ADC_GetConfig

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Determine information about the device found at a specific DeviceIndex

Delphi: function ADC_GetConfig(DeviceIndex: LongWord; pConfigBuf: Pointer; var ConfigBufSize: LongWord): LongWord; cdecl;

Visual C: unsigned long ADC_GetConfig(unsigned long DeviceIndex, void *pConfigBuf, unsigned long *ConfigBufSize);

C#: UInt32 ADC_GetConfig(UInt32 DeviceIndex, [In, Out] byte[] pConfigBuf, ref UInt32 ConfigBufSize);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.
pConfigBuf	a pointer to the first of an array of bytes for configuration data
ConfigBufSize	a pointer to a variable holding the number of configuration bytes to read. Will be set to the number of configuration bytes read

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes:

ADC_Initialize

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
		USB-AI16-16A and USB-AI12-16A only.			

Purpose: Determine information about the device found at a specific DeviceIndex

Delphi: function ADC_Initialize(DeviceIndex: LongWord; pConfigBuf: Pointer; var ConfigBufSize: LongWord; CalFileName: PChar): LongWord; cdecl;

Visual C: unsigned long ADC_Initialize(unsigned long DeviceIndex, void *pConfigBuf, unsigned long *ConfigBufSize, char *CalFileName);

C#: UInt32 ADC_Initialize(UInt32 DeviceIndex, [In, Out] byte pConfigBuf, ref UInt32 pConfigBufSize, [In, Out] Char[] CalFileName);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
pConfigBuf	a pointer an array of configuration bytes, identical to that used in ADC_SetConfig()
ConfigBufSize	a pointer to a variable holding the number of configuration bytes to write.
CalFileName	the file name of a calibration file, or a command string. See ADC_SetCal() for details.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: ADC_Initialize just calls both ADC_SetConfig() and ADC_SetCal() conveniently. Refer to those two functions for details on the parameters.

ADC_SetConfig

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Perform extensive configuration of analog input circuits on the USB-AIx family of products.

Delphi: function ADC_SetConfig(DeviceIndex: LongWord; pConfigBuf: Pointer; var ConfigBufSize: LongWord): LongWord; cdecl;

Visual C: unsigned long ADC_SetConfig(unsigned long DeviceIndex, void *pConfigBuf, unsigned long *ConfigBufSize);

C#: UInt32 ADC_SetConfig(UInt32 DeviceIndex, [In, Out] byte[] pConfigBuf, ref UInt32 ConfigBufSize);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
pConfigBuf	a pointer to the first of an array of bytes for configuration data
ConfigBufSize	a pointer to a variable holding the number of configuration bytes to write (21 bytes, typically)

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The configuration of the USB-AIx analog inputs can be configured in one function call via ADC_SetConfig, but the array of configuration options can be a little confusing. Feel free to use the individual configuration calls, documented next.

Configuration bytes for analog input boards(the USB-AIx family) are as follows:

[00h]	…	[0Fh]	[10h]	[11h]	[12h]	[13h]	[14h]
Range Code 0	…	Range Code 15	Cal. Code	Trigger & Counter	Start & End Channel	Oversample	Extended Channel

A configuration of 21 zeros is close to an "ordinary" use; you'll likely want to set timer or external trigger, and start and end channels.

Config Bytes 00-0Fh:
Range codes correspond to ranges as follows:

Range Code	00	01	02	03	04	05	06	07
Range	0-10V	±10V	0-5V	±5V	0-2V	±2V	0-1V	±1V

Add 0x08 to the range code for any "lower" channel(s) to configure that channel as differential.

These range codes are the shown for the standard, un-signal-conditioned AI input ranges. If you're using a submultiplexer, the range chosen on those channels is further modified by the range selected using the table above.

The USB-AIx supports the use of sub-multiplexer boards to increase the number of channels, and/or to provide elaborate and flexible signal-conditioning options. The 16 "Range Code" entries in the

ADC_SetConfig (elements 00h-0Fh in the byte array) each control the input range/gain on one or more channels, as follows:

16-Channel Boards								
Config Byte	00h	01h	…	07h	08h	…	0Eh	0Fh
sets range on channels:	0	1	…	7	8	…	14	15
"64M" Boards (64-channel)								
Config Byte	00h	01h	…	07h	08h	…	0Eh	0Fh
sets range on channels:	0-3	4-7	…	28-31	32-35	…	56-59	60-63
Other Boards (32-, 64-, 96-, or 128-channel)								
Config Byte	00h	01h	…	07h	08h	…	0Eh	0Fh
sets range on channels:	0-7	8-15	…	56-63	64-71	…	112-119	120-127

Please note: When using differential mode, the ADC_SetConfig array elements 8-15 are unused.

Config Byte 10h:
Calibration Mode configuration. Typical applications will always write "0x00" to this config byte.

Calibration Code		Expected
00h	The ADC will acquire data from the external pins, as normally expected	user input
01h	The ADC will acquire the onboard Unipolar Ground reference	0V
03h	The ADC will acquire the onboard Unipolar Full-Scale voltage reference	9.9339V @10V unipolar range
05h	The ADC will acquire the onboard Bipolar Zero reference	0V
07h	The ADC will acquire the onboard Bipolar Full-Scale voltage reference	9.8678V @ 10V bipolar range

Config Byte 11h:
Trigger & counter 0 CLK configuration

Bit	d7	d6	d5	d4	d3	d2	d1	d0
Value	Reserved, use 0			CTR0 EXT	Falling Edge	Scan	External Trigger	Timer Trigger

- If CTR0 EXT is set, counter 0 is externally-clocked; otherwise, counter 0 is clocked by the onboard 10MHz clock.
- If Falling Edge is set, A/D acquisition is triggered by the falling edge of its trigger source; otherwise, A/D acquisition is triggered by the rising edge of its trigger source.
- If Scan is set, a single A/D trigger will acquire all channels from start channel to end channel, with oversampling (if so configured), at maximum speed. Otherwise, a single A/D trigger will cause a single conversion, iterating through oversamples and channels.
- If External Trigger is set, the external A/D trigger pin is an A/D trigger source. Otherwise, this pin is ignored. Please note: the trigger pin can trigger a Scan, or a single Conversion. It is not used to start a sequence of Scans or Conversions.
- If Timer Trigger is set an onboard pacing clock circuit is used as an A/D trigger source. Otherwise, the output of the pacing clock circuit is ignored.

Config Byte 12h and Config Byte 14h:
Start channel and end channel configuration (for "scans")

Bits	4-7	0-3
config byte 12h	End Channel bits 0-3	Start Channel bits 0-3
config byte 14h	End Channel bits 4-7	Start Channel bits 4-7

For example, to start at 0 and end at 63 (3Fh), set config byte 12h to F0h and config byte 14h to 30h. To start at 7 and end at 107 (6Bh), set config byte 12h to B7h and config byte 14h to 60h. In any case, if the end channel is less than the start channel, then the board's behavior is unspecified.

It may be more convenient to set these bytes by calling ADC_SetScanLimits.

Config Byte 13h:
Oversample is a number indicating how many **extra** samples should be acquired from each channel before moving on to the next. In a noisy environment, the samples can be averaged together by software to effectively reduce noise. The number set here is in addition to the single acquisition of every channel that happens normally. Therefore, passing an oversample of "zero" results in one conversion per channel, passing an oversample of "one" (1) results in two conversions per channel, and passing "255" will result in a total of 256 conversions per channel being taken.

ADC_RangeAll

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: This utility function performs a read-modify-write of the ADC_SetConfig structure's first 16 bytes (the gain/range codes).

Delphi: function ADC_RangeAll(DeviceIndex: LongWord; pGainCodes: PByte; bDifferential: LongBool): LongWord; cdecl;

Visual C: unsigned long ADC_RangeAll(unsigned long DeviceIndex, unsigned char *pGainCodes, unsigned long bDifferential);

C#: UInt32 ADC_RangeAll(UInt32 DeviceIndex, ref byte GainCodes, UInt32 bDifferential);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
pRangeCodes	a pointer to an array of 16 bytes, each of which contains a range code.
bDifferential	Use FALSE for single-ended mode, use TRUE for differential mode

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: See ADC_SetConfig() for details about valid Range Codes

This function cannot configure individual channels as differential, only all-or-none. To configure single-ended/differential on a per-channel basis, use ADC_Range1() or ADC_SetConfig()

ADC_Range1

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Determine information about the device found at a specific DeviceIndex

Delphi: function ADC_Range1(DeviceIndex, ADChannel: LongWord; GainCode: Byte; bDifferential: LongBool): LongWord; cdecl;

Visual C: unsigned long ADC_Range1(unsigned long DeviceIndex, unsigned long ADChannel, unsigned char GainCode, unsigned long bDifferential);

C#: UInt32 ADC_Range1(UInt32 DeviceIndex, UInt32 ADChannel, byte GainCode, UInt32 bDifferential);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
RangeEntry	number from 0-15 indicating an A/D range code entry on the device
RangeCode	a range code byte
bDifferential	For range code entries 0-7, use FALSE for single-ended mode, use TRUE to pair it with the respective channel 8-15 in differential mode. For channels 8-15, use FALSE

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: See ADC_SetConfig() for details about valid Range Codes.

Also, see ADC_SetConfig() for details about mapping A/D Channel numbers to Range Entries.

ADC_SetScanLimits

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose:

Delphi: function ADC_SetScanLimits(DeviceIndex, StartChannel, EndChannel: LongWord): LongWord; cdecl;

Visual C: unsigned long ADC_SetScanLimits(unsigned long DeviceIndex, unsigned long StartChannel, unsigned long EndChannel);

C#: UInt32 ADC_SetScanLimits(UInt32 DeviceIndex, UInt32 StartChannel, UInt32 EndChannel);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.

StartChannel	the number of the first channel you want in a scan
EndChannel	the number of the last channel you want in a scan

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: a "Scan" will consist of data from all channels from StartChannel through EndChannel.

ADC_ADMode

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Configure the A/D trigger source and Calibration Acquisition Mode

Delphi: function ADC_ADMode(DeviceIndex: LongWord; TriggerMode, CalMode: Byte): LongWord; cdecl;

Visual C: unsigned long ADC_ADMode(unsigned long DeviceIndex, unsigned char TriggerMode, unsigned char CalMode);

C#: UInt32 ADC_ADMode(UInt32 DeviceIndex, byte TriggerMode, byte CalMode);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
TriggerMode	configuration of A/D trigger mode and CTR0 input clock source.
CalMode	byte indicating which A/D input to acquire: external pins, or one of the onboard calibration references.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Refer to ADC_SetConfig() for details.

ADC_SetOversample

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Oversample can make your data quieter, but slows down the acquisition and adds skew.					

Purpose: This utility function provides read-modify-write access to the "Oversample" configuration byte in the ADC_SetConfig array.

Delphi: function ADC_SetOversample(DeviceIndex: LongWord; Oversample: Byte): LongWord; cdecl;

Visual C: unsigned long ADC_SetOversample(unsigned long DeviceIndex, unsigned char Oversample);

C#: UInt32 ADC_SetOversample(UInt32 DeviceIndex, byte Oversample);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.

Oversample	the number of extra samples to take from each channel in a scan.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The total number of conversions per channel is 1 + this setting.

ADC_SetCal

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-AI16-16A and USB-AI12-16A only.					

Purpose: Configure the analog input calibration of a USB-AIx board.

Delphi: function ADC_SetCal(DeviceIndex: LongWord; CalFileName: PChar): LongWord; cdecl;

Visual C: unsigned long ADC_SetCal(unsigned long DeviceIndex, char *CalFileName);

C#: UInt32 ADC_SetCal(UInt32 DeviceIndex, [In, Out] Char[] CalFileName);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
CalFileName	either the file name of a calibration file, or one of several reserved command strings. A file name can include the full path, or be relative to the current directory. A command string of ":AUTO:" causes this function to generate a calibration file from the calibration references and upload that. A command string of ":NONE:" causes this function to upload an "uncalibrated" calibration file

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: ":NONE:" can be written, and is identical to, ":1TO1:"

It is not valid to call ADC_SetCal() on boards that do not support calibration. Use ADC_QueryCal() to determine if your device supports calibration.

Note: Some revisions of the driver will return errors under all conditions if you attempt to ADC_SetCal() unsupported hardware; other driver revisions will return ERROR_SUCCESS if you explicitly request ":NONE:" (or ":1TO1:") on a board that does not support calibration, as the request, although skipped, was technically successful - the calibration table of unsupported hardware is *always* :NONE:

ADC_QueryCal

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Determine if a given device supports uploaded Calibration tables

Delphi: function ADC_QueryCal(DeviceIndex: LongWord): LongWord; cdecl;

Visual C: unsigned long ADC_QueryCal(unsigned long DeviceIndex);

C#: UInt32 ADC_QueryCal(UInt32 DeviceIndex);

Argument **Description**

DeviceIndex DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success". ERROR_NOT_SUPPORTED (50) will be returned for USB-AIx devices that do not support calibration table upload.

Notes: It is useful to call before any call to ADC_SetCal(), to avoid trying to issue calibration uploads on unsupported hardware.

A/D HIGH SPEED ACQUISITION

Our USB-AIx family of Analog Input boards support a variety of acquisition modes.

"ADC_GetScanV()", described above, is the simplest and most convenient of these. However, ADC_GetScanV is not synchronous to either an external trigger source, nor to the onboard pacer clock; rather, it simply takes data when commanded by software. Additionally, the overhead of the init and deinit steps performed limits the maximum scan rate to less than approximately 100Hz.

If you need to acquire data faster than 100Hz per channel, or need the data to be acquired periodically or synchronous to an external pulse-train, you will need to use one of the "high-speed acquisition modes" described below.

The set of high-speed acquisition modes is always increasing as we make improvements to our software suite.

At this time, two high-speed modes are published: Poll Mode, and Callback Mode.

Poll Mode
Poll mode is implemented using ADC_BulkAcquire(), and is the oldest of the high speed acquisition modes provided.

It:

- takes a fixed number of samples from the A/D then stops (up to 2 gigasamples),
- allows full speed acquisition of input channels (up to 500k readings/second),
- uses either the onboard pacer clock or the external trigger as scan or conversion trigger,
- requires you to to allocate and pass a buffer large enough to hold all the readings.
- uses ADC_BulkPoll() to determine how much data in the allocated buffer is "valid" - your program can use as much data as is valid immediately, while the remainder of the data is still being acquired.

Callback Mode:
Callback mode is implemented using ADC_BulkContinuousCallbackStart(), and is the newer of the high speed modes published.

It:

- acquires data until stopped,
- allows full-speed acquisition,
- uses either the onboard pacer clock or the external trigger pin,
- does not require the preallocation of a buffer large enough to take all the readings,
- does not require you to know in advance how many readings to take.
- uses callbacks to inform your application when new data is available. If your programming language supports callbacks, this can greatly reduce code complexity.

ADC High-Speed Polling Mode

"Polling" might seem like something of a misnomer: you don't actually acquire data by polling the card. Rather, the card and drivers cooperate silently in the background to collect the data and stuff it into a buffer for your use. However, you *do* poll "how much data is ready for me" information from the driver, using the ADC_BulkPoll() function, and this is how Polling mode got its name.

A true, slow, polling mode is available via the "ADC_Get" family of functions.

ADC_BulkAcquire

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Determine information about the device found at a specific DeviceIndex

Delphi: function ADC_BulkAcquire(DeviceIndex: LongWord; BufSize: LongWord; pBuf: Pointer): LongWord; cdecl;

Visual C: unsigned long ADC_BulkAcquire(unsigned long DeviceIndex, unsigned long BufSize, void *pBuf);

C#: UInt32 ADC_BulkAcquire(UInt32 DeviceIndex, UInt32 BufSize, [In, Out] UInt16[] Buf);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
BufSize	the size, in bytes, of the buffer to receive the data
pBuf	a pointer to the buffer in which to receive data

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: This function will return immediately. A return value of "ERROR_SUCCESS"(equal to 0) indicates that bulk data is being acquired in the background, and the buffer should not be deallocated or moved. Use ADC_BulkPoll() to query this background operation.

ADC_BulkPoll

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Check how much ADC data remains to be taken, and how much is available for use.

Delphi: function ADC_BulkPoll(DeviceIndex: LongWord; var BytesLeft: LongWord): LongWord; cdecl;

Visual C: unsigned long ADC_BulkPoll(unsigned long DeviceIndex, unsigned long *BytesLeft);

C#: UInt32 ADC_BulkPoll(UInt32 DeviceIndex, out UInt32 BytesLeft);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to query; generally either diOnly or a specific device's Device Index.

BytesLeft	a pointer to a variable which will be set to the number of bytes of A/D data remaining to be taken

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Any data that has been taken is available in the buffer, starting from the beginning. For example, if ADC_BulkAcquire() was called to take 1024 MB of data, and ADC_BulkPoll() indicates 768 MB is left to be taken, then the first 256 MB of data is available.

ADC Callback Mode

ADC_BulkContinuousCallbackStart

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Determine information about the device found at a specific DeviceIndex

Delphi: function ADC_BulkContinuousCallbackStart(DeviceIndex: LongWord; BufSize: LongWord; BaseBufCount: LongWord; Context: LongWord; pCallback: Pointer): LongWord; cdecl;

Visual C: unsigned long ADC_BulkContinuousCallbackStart(unsigned long DeviceIndex, unsigned long BufSize, unsigned long BaseBufCount, unsigned long Context, void *pCallback);

C#: UInt32 ADC_BulkContinuousCallbackStart(UInt32 DeviceIndex, UInt32 BufSize, UInt32 BaseBufCount, UInt32 Context, ADCallback pCallback);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
BufSize	number of bytes (a multiple of 512) for each buffer in the software FIFO
BaseBufCount	number of buffers in the software FIFO, for example 64. Minimum 2. This value is used to pre-allocate buffers. The code will allocate new buffers as needed, but allocating memory is relatively slow and could result in data loss. See Flags Bit 2 for more information.
Context	any value, will be passed to the callback
pCallback	pointer to an ADContCallback() function to receive buffers

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Starts a continuous bulk acquire process. A worker thread will acquire data, using the current configuration as set by ADC_SetConfig or its helper functions, a buffer at a time; another worker thread will pass a buffer at a time to the callback. The clock should be stopped while calling this function, like so:

```
double Hz = 0.0;
CTR_StartOutputFreq(DeviceIndex, 0, &Hz);
ADC_SetConfig(DeviceIndex, &Config[0], ConfigSize);
ADC_BulkContinuousCallbackStart(DeviceIndex, 16*1024, 32, 0, &ADCallback);
Hz = 30000;
CTR_StartOutputFreq(DeviceIndex, 0, &Hz);
```

The pCallback parameter is a pointer to a function with the following signature:

```
void ADContCallback(
        UInt16 *pBuf - pointer to the first of an array of WORD samples
        UInt32 BufSize - size, in bytes, of the array passed in pBuf; can be zero
        UInt32 Flags - a bitmask of flags, see table, below.
        UInt32 Context - a copy of the Context parameter passed to
                ADC_BulkContinuousCallbackStart()
)
```

Each callback the driver will fill in its parameters as indicated. Note that it will be called from an alternate thread context. Flags are as follows:

Bit	Mask	Meaning
0		*factory use only*
1	Flags & 2	End of stream; this is the last buffer. Typically one last zero-size buffer will be passed, in order to set this flag.
2	Flags & 4	The BaseBufCount was too small; this buffer was added to the FIFO, which may interrupt the data stream afterward. At sampling rates of a few Hz, a BaseBufCount of 2 is plenty. On a fast computer, a BaseBufCount of 64 can handle up to 500kHz sampling rate. High sampling rates on a slow computer may require higher BaseBufCount values.

ADC_BulkContinuousEnd

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Terminate continuous acquisition and cease callbacks.

Delphi: function ADC_BulkContinuousEnd(DeviceIndex: LongWord; pIOStatus: PLongWord): LongWord; cdecl;

Visual C: unsigned long ADC_BulkContinuousEnd(unsigned long DeviceIndex, unsigned long *pIOStatus);

C#: UInt32 ADC_BulkContinuousEnd(UInt32 DeviceIndex, out UInt32 pIOStatus);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
pIOStatus	pointer to a variable to receive I/O status of the continuous process. If you don't care about the I/O status, pass a null pointer

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes:

DIGITAL TO ANALOG

The AIOUSB Digital To Analog API operates in two distinct modes: direct mode, and streaming mode.

Only the USB-DA12-8A device' DACs support streaming mode operation at this time.

All DACs, including those provided by the USB-DA12-8A, support direct mode operation.

DAC Direct Mode

DACSetBoardRange

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Turn on the DAC voltage references. In some products also configures the DAC calibration table for bipolar or unipolar use. Some boards also use this function to set the output range.

Delphi: function DACSetBoardRange(DeviceIndex: LongWord; RangeCode: LongWord): LongWord; cdecl;

Visual C: unsigned long DACSetBoardRange(unsigned long DeviceIndex, unsigned long RangeCode);

C#: UInt32 DACSetBoardRange(UInt32 DeviceIndex, UInt32 RangeCode);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
RangeCode	the range code to set for the board; see the hardware manual for your device's range codes

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: Must be called before analog outputs will produce meaningful outputs.

DACDirect

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS

Purpose: Update the output of a single DAC

Delphi: function DACDirect(DeviceIndex: LongWord; Channel: Word; Value: Word): LongWord; cdecl;

Visual C: unsigned long DACDirect(unsigned long DeviceIndex, unsigned long Channel, unsigned short Counts);

C#: UInt32 DACDirect(UInt32 DeviceIndex, UInt16 Channel, UInt16 Counts);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.

Channel	number from 0-7 indicating which DAC you wish to set
Value	count value to which you wish to set the DAC

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: On the USB-DA12-8A family, a Value of 000h indicates the lowest DAC level and 0FFFh indicates the highest DAC level; other values are proportional.

On the USB-AO family, and all other USB analog outputs, Value varies from 0000h through FFFFh or FFF0h, for the 16- and 12-bit models, respectively.

DACMultiDirect

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS
The most efficient method to output voltages					

Purpose: Update the output(s) of a list of DACs

Delphi: function DACMultiDirect(DeviceIndex: LongWord; pDACData: PWord; DACDataCount: LongWord): LongWord; cdecl;

Visual C: unsigned long DACMultiDirect(unsigned long DeviceIndex, void *pDACData, unsigned long DACDataCount);

C#: UInt32 DACMultiDirect(UInt32 DeviceIndex, [In, Out] UInt16[] pDACData, UInt32 DACDataCount);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
pDACData	a pointer to an array of WORDs, consisting of channel/value pairs
DACDataCount	indicates how many channel/value pairs are in the pDACData array

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: to update a single channel with DACMultiDirect you could pass an array consisting of a single pair of WORDS, the channel number in the first, and the count value in the second. Normally, however, you pass an array containing more than one of these pairs.

Calling DACMultiDirect() a single time takes very slightly longer than calling DACDirect() a single time, but DACMultiDirect can update as many as 16 DAC channels with a single USB transaction. Therefore, if you are updating more than one DAC's value, DACMultiDirect() will always execute faster.

DAC Streaming Mode

DACOutputProcess

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
			USB-DA12-8A only.		

Purpose: This function begins a one-shot DAC output process. Rather than streaming DAC data continuously, it opens a connection, sends a single block of data, then closes. The DAC data will then be clocked out based on the EOP bit, see DACOutputFrameRaw() below for details

Delphi: function DACOutputProcess(DeviceIndex: LongWord; var ClockHz: Double; NumSamples: LongWord; pSampleData: PWord): LongWord; cdecl;

Visual C: unsigned long DACOutputProcess(unsigned long DeviceIndex, double *ClockHz, unsigned long NumSamples, unsigned short *SampleData);

C#: UInt32 DACOutputProcess(UInt32 DeviceIndex, ref double ClockHz, UInt32 NumSamples, [In, Out] UInt16 SampleData);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
pClockHz	a pointer to a double precision IEEE floating point number containing the desired output clock frequency. This value is set by the driver to the *actual* frequency at which DAC data will be clocked out, as limited by the device's capabilities.
NumSamples	the total number of samples to output. Notably, this is not a number of "points"
SampleData	a pointer to an array of WORDs; each DAC value is stored in a WORD, so it should contain (samples to output) WORDs. The features are controlled by the upper bits in the data array; for details on this format, refer to DACOutputFrameRaw()

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: **CAUTION: A Known Bug requires your waveform to be 65537 samples or larger.** If you intend to send waveforms shorter than this, build your shorter waveform as usual, then pad it out to 65537 using a pad value of 0x1nnn, where "nnn" is the count value in your built waveform's first point's first data sample. (The first count value for DAC #0.)

DACOutputOpen

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
			USB-DA12-8A only.		

Purpose: This function begins a DAC streaming process. The stream is divided into "points"; each point contains data for one or more DACs, and during the streaming process the onboard counter/timer clocks out points at a steady rate.

Delphi: function DACOutputOpen(DeviceIndex: LongWord; var ClockHz: Double): LongWord; cdecl;

Visual C: unsigned long DACOutputOpen(unsigned long DeviceIndex, double *ClockHz);

C#: UInt32 DACOutputOpen(UInt32 DeviceIndex, ref double ClockHZ);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
pClockHz	a pointer to a double precision IEEE floating point number containing the desired output clock frequency. This value is set by the driver to the *actual* frequency at which DAC data will be clocked out, as limited by the device's capabilities.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: All DACs in a single "point" will be updated simultaneously (on the same clock tick). The next point will be output on the subsequent clock tick.

DACOutputClose

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only. *Deprecated: DACOutputCloseNoEnd is preferred for new code.*					

Purpose: Deprecated

This function causes the USB-DA12-8A to cease streaming data

Delphi: function DACOutputClose(DeviceIndex: LongWord; bWait: LongBool): LongWord; cdecl;

Visual C: unsigned long DACOutputClose(unsigned long DeviceIndex, unsigned long bWait);

C#: UInt32 DACOutputClose(UInt32 DeviceIndex, UInt32 bWait);

Argument	Description
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
bWait	reserved for future expansion; currently, this function always waits for the streaming process to complete before returning to the caller

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: This function is deprecated because it will insert an EOM bit on the last data point under certain circumstances.

DACOutputCloseNoEnd

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.					

Purpose: This function closes a DAC streaming process *without* ending it. This is most useful when you've set LOOP or EOM via DACOutputFrameRaw().

Delphi: function DACOutputCloseNoEnd(DeviceIndex: LongWord; bWait: LongBool): LongWord; cdecl;

Visual C: unsigned long DACOutputCloseNoEnd(unsigned long DeviceIndex, unsigned long bWait);

C#: UInt32 DACOutputCloseNoEnd(UInt32 DeviceIndex, UInt32 bWait);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
bWait	reserved for future expansion; currently, this function always waits for the streaming process to complete before returning to the caller.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes:

DACOutputSetCount

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.					

Purpose: This function sets the number of DACs involved in each DAC streaming point henceforth.

Delphi: function DACOutputSetCount(DeviceIndex, NewCount: LongWord): LongWord; cdecl;

Visual C: unsigned long DACOutputSetCount(unsigned long DeviceIndex, unsigned long NewCount);

C#: UInt32 DACOutputClose(UInt32 DeviceIndex, UInt32 bWait);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
NewCount	number from 1-8 indicating the number of DACs in future points

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: When the driver connects to the device, this is initialized to 5 (for ILDA use).

You can set this freely between calls to DACOutputFrameRaw() if you wish.

This value is used to scale the "FramePoints" parameter in subsequent calls to DAOutputFrameRaw(), make sure it matches.

DACOutputFrame

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only. *Deprecated: DACOutputFrameRaw is preferred for new code.*					

Purpose: Deprecated

This function writes a collection of points (called "a frame") into the DAC stream.

Delphi: function DACOutputFrame(DeviceIndex, FramePoints: LongWord; FrameData: PWord): LongWord; cdecl;

Visual C: unsigned long DACOutputFrame(unsigned long DeviceIndex, unsigned long FramePoints, unsigned short *FrameData);

C#: UInt32 DACOutputFrame(UInt32 DeviceIndex, UInt32 FramePoints, [In, Out] UInt16 pFrameData);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
FramePoints	the number of points in the frame - NOT BYTES.
FrameData	a pointer to an array of WORDs; each DAC value is stored in a WORD, so it should contain (DAC count) × (points in the frame) WORDs

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: This function is deprecated because it causes the output to start playback automagically when 1¼ SRAMs worth of samples have been uploaded. It also inserts an EOM bit if DACOutputClose() is called, under certain circumstances.

All points in any given frame buffer must control the same number of DACs; if, for example, you wish to output one point with all 8 DACs, followed by 99 points with only 2 DACs, set the DAC count to 8, output a frame consisting of just the first point, then set the DAC count to 2, and output a frame of the next 99 points.

If the driver's internal buffer is full, the function will return "ERROR_NOT_READY" (equal to 21 decimal); try again in a moment, as the driver's buffer should drain some as soon as there's room in the on-card hardware buffer, and available time on the USB bus.

DACOutputFrameRaw

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.					

Purpose: This function writes a group of "points" (a group of points is referred to as a "frame") into the DAC stream.

Delphi: function DACOutputFrameRaw(DeviceIndex, FramePoints: LongWord; FrameData: PWord): LongWord; cdecl;

Visual C: unsigned long DACOutputFrameRaw(unsigned long DeviceIndex, unsigned long FramePoints, unsigned short *FrameData);

C#: UInt32 DACOutputFrameRaw(UInt32 DeviceIndex, UInt32 FramePoints, [In, Out] UInt16 FrameData);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
FramePoints	the number of points in the frame - NOT BYTES.
FrameData	a pointer to an array of WORDs; each DAC value is stored in a WORD, so it should contain (DAC count) × (points in the frame) WORDs Also note: all WORDs, in each point, contain a 12-bit count value AND four control

bits (in the highest nybble). See the discussion, below.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: All points in any given frame buffer must control the same number of DACs; if, for example, you wish to output one point with all 8 DACs, followed by 99 points with only DAC 0 and 1 changing, set the DAC count to 8, output a frame consisting of just the first point, then set the DAC count to 2, and output a frame of the next 99 points. (You set the DAC count using DACOutputSetCount()).

You *may* continue sending frames using DACOutputFrameRaw() after issuing a DACOutputStart().

You *must* continue sending frames to avoid the playback of uninitialized or recycled memory, if your already-uploaded waveform does not include a LOOP or EOM bit.

This is true streaming, and there is no upper limit to the amount of data in a waveform that can be written in this manner.

If the driver's internal buffer is full, the function will return "ERROR_NOT_READY" (equal to 21 decimal); try again in a moment, as the driver's buffer should drain some as soon as there's room in the on-card hardware buffer, and available time on the USB bus.

The meanings of the bits in each sample's WORD in each point are as follows:

Bit #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Meaning	EOM	EOF	EOP	LOOP	DAC Value, in counts											

- If EOM ("End Of Memory") is set on any sample the board will stop the waveform after outputting the sample. Do not set EOM if LOOP is set, and vice-versa.
- If EOF ("End Of Frame") is set, the frame pin will be pulsed. The bit is otherwise unused, making it available for your application, as desired.
- If EOP ("End Of Point") is set, the card will write the point samples to the hardware, and wait for the next tick. This also has the effect of telling the device that the next WORD in waveform memory is the beginning of the next point's data, and therefore is the sample for DAC #0.
- If LOOP is set, the board will "jump" to the beginning of its buffer after outputting the sample. This can be used to load a "repeating" waveform, like a sine wave or sawtooth, and then loop it without further attention from the host computer. Indeed, with external power, you can disconnect the USB cable without interrupting the loop. Do not set LOOP if EOM is set, and vice-versa.

Note that the EOM and LOOP bits are for mutually exclusive uses. Setting them both is reserved for future expansion.

DACOutputStart

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.					

Purpose: Starts your uploaded waveform data updating to the DAC output pins.

Delphi: function DACOutputStart(DeviceIndex: LongWord): LongWord; cdecl;

Visual C: unsigned long DACOutputStart(unsigned long DeviceIndex);

C#: UInt32 DACOutputStart(UInt32 DeviceIndex);

Argument Description

DeviceIndex DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: **CAUTION: A Known Bug requires your waveform to be 65537 samples or larger before starting.** If you intend to send waveforms shorter than this, build your shorter waveform as usual, then pad it out to 65537 using a pad value of 0x1nnn, where "nnn" is the count value in your built waveform's first point's first data sample. (The first count value for DAC #0.)

The following paragraph describes the intended operation:

Note that before starting DAC output you must send the lesser of one SRAM worth of data (128K bytes, i.e. 65536 samples) or your entire waveform, due to the use of bank-switched single-ported memory.

You may continue sending frames using DACOutputFrameRaw() after issuing a DACOutputStart(), and, in fact, if your already-uploaded waveform does not include a LOOP or EOM bit, must upload additional data to avoid the playback of uninitialized or recycled memory.

This is true streaming, and there is no upper limit to the amount of data in a waveform that can be written in this manner.

DACOutputSetInterlock

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.					

Purpose: Enables or Disables the "Interlock" feature

Delphi: function DACOutputSetInterlock(DeviceIndex: LongWord; bInterlock: LongBool): LongWord; cdecl;

Visual C: unsigned long DACOutputSetInterlock(unsigned long DeviceIndex, unsigned long bInterlock);

C#: UInt32 DACOutputSetInterlock(UInt32 DeviceIndex, UInt32 bInterlock);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
bInterlock	TRUE to enable interlock, FALSE to disable interlock. While interlock is enabled, DAC streaming is paused unless the interlock pin is grounded, usually through the cable. The interlock pin is pin 12 of the DB25 M connector (or, on the OEM version, pin 7 of the connector named J4)

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The device powers up with the interlock feature disabled.

GENERAL FUNCTIONS

AIOUSB_SetStreamingBlockSize

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS 🟩	ANALOG OUTPUTS	BUFFERED DI, DO, DIO 🟩	COUNTER TIMERS

Purpose: Reconfigure the size used for the low-level interface to the USB Bulk endpoints.

Delphi: function AIOUSB_SetStreamingBlockSize(DeviceIndex, BlockSize: LongWord): LongWord; cdecl;

Visual C: unsigned long AIOUSB_SetStreamingBlockSize(unsigned long DeviceIndex, unsigned long BlockSize);

C#: UInt32 AIOUSB_SetStreamingBlockSize(UInt32 DeviceIndex, UInt32 BlockSize);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
BlockSize	the new streaming block size you wish to set. For DIO streaming, this will get rounded up to the next multiple of 256. For A/D streaming, this will get rounded up to the next multiple of 512.

Returns: Standard "Win32" error codes. ERROR_SUCCESS (0) means "Success".

Notes: The larger the BlockSize, the more lag-tolerance.

The smaller the BlockSize, the less latency between action in software and action in hardware, or vice versa.

The driver defaults to 8192.

AIOUSB_ClearFIFO

APPLIES TO					
DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS 🟩	ANALOG OUTPUTS	BUFFERED DI, DO, DIO 🟩	COUNTER TIMERS

Purpose: Determine information about the device found at a specific DeviceIndex

Delphi: function AIOUSB_ClearFIFO(DeviceIndex: LongWord; TimeMethod: LongWord): LongWord; cdecl;

Visual C: unsigned long AIOUSB_ClearFIFO(unsigned long DeviceIndex, unsigned long TimeMethod);

C#: UInt32 AIOUSB_ClearFIFO(UInt32 DeviceIndex, UInt32 TimeMethod);

Argument	**Description**
DeviceIndex	DeviceIndex of the card you wish to control; generally either diOnly or a specific device's Device Index.
Method	0 to simply clear the FIFO right away, others per the table below

Returns: Standard “Win32" error codes. ERROR_SUCCESS (0) means “Success”.

Notes:

Clear FIFO Method Codes

Code (decimal)	Effect
0	Clear FIFO as soon as command received (and disable auto-clear)
1	Enable auto-clear FIFO every falling edge of DIO port D bit 1 (on digital boards, analog boards treat this method as identical to “method 0")
5	As 0, but also abort any streaming data flow currently in progress