ACCES I/O PRODUCTS, INC.
10623 Roselle Street, San Diego, CA 92121 • (858) 550-9559 • www.acces.io

USB Software Reference Manual

TABLE OF CONTENTS

INTRODUCTION

This manual provides a reference to the USB function driver, AIOUSB, and other provided software that applies to our USB products. ***Serial*** *products simply appear as "COM" ports and are operated using the built-in Windows serial interfaces*.

You can use this document in a number of ways. All users should read the entirety of this introduction, first. After that the most common is to 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. Or, you could read the entire manual front-to-back; although this may seem inefficient, it may provide useful insight into other APIs or devices you may be able to use in your system, which the admittedly rather simple sample programs may not demonstrate.

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. We've provided some easy tools to determine if you should bother reading about a particular API section. For example, the beginning of each section will look similar to the following:

DIO_Read1

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

Explanation

DIO_Read1 is the name of the API function this section of the manual is describing.

The list of product types has ***one valid, highlighted, I/O type***, in this case Digital Inputs. The other types are displayed ***in gray, indicating they do not apply***.

This indicates the **DIO_Read1** function will be used only with digital inputs on our USB products. This API's section of the manual will not be useful if you're using one of the other I/O types, such as Analog Outputs.

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. For example, the full table for **DIO_Read1** actually looks like the following:

DIO_Read1

APPLIES TO						
RS-232 -422 / -485	**DIGITAL INPUTS**	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

The bit is returned as “0" or “1" - be careful in your programming language if you are using true / false.

A breakdown of what each category description means follows:

RS-232, -422 / -485: Any serial device. Included only for completeness, as serial ports use the standard Windows Comm interfaces and are documented by Microsoft.

DIGITAL INPUTS: TTL, CMOS, LVTTL, and Isolated Input types are 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 types are all included here. AIOUSB treats all boolean inputs 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.

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. Some functions that are marked with this category only apply to the “Waveform” capable models.

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.

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.

Please note that many models are “Multi-Function” and contain I/O falling into more than one of these categories; digital input/output being the most obvious. Additionally, some specific API functions apply to all devices.

DRIVER REFERENCE

AIOUSB provides a standard interface to all our Data Acquisition USB modules. Each specific USB device will use a subset of the driver calls listed below, based on its specific capabilities and needs. The first two function calls listed (GetDevices and QueryDeviceInfo) are used by every device as part of the initialization process for your software code. For example source code, please refer to any of the numerous software sample programs provided.

The constants **diFirst** (equal to FFFFFFFE hex) and **diOnly** (equal to FFFFFFFD hex) can be passed for DeviceIndex in place of an actual device index. diFirst causes the function to operate on the first device, whatever its device index. diOnly causes the function to operate on the only device if is only one, or to return "ERROR_DUP_NAME" (equal to 52 decimal) if there's more than one device. Using these defined constants can greatly simplify programming for USB devices in situations where only a single USB device will be installed in the system.

For example, if you know that only one of our USB data acquisition devices at a time, of a known type, will ever be installed in the system you’re writing the program for, you can

skip GetDevices and QueryDeviceInfo entirely. Using diOnly instead guarantees you will find the correct device index. This can reduce the code required to operate the unit to a single call to a single API function in some cases. *Our sample programs demonstrate how to handle multiple boards, not the simple case.*

All DWORD return values other than GetDevices() and ResolveDeviceIndex() are Windows error codes, and will be "ERROR_SUCCESS" (equal to 0) if no error occurred. If the USB device has been removed during use, the error returned is "ERROR_DEVICE_REMOVED" (equal to 1617 decimal) and will persist until ClearDevices() has been called. If this state is cleared and the board was not reconnected, the error returned is "ERROR_FILE_NOT_FOUND" (equal to 2). Versions of AIOUSB.dll prior to 2.1 return "ERROR_DEV_NOT_EXIST" (equal to 55 decimal) under both conditions. The full list of Windows error codes is in WinError.h, which can also be found on the web.

GetDevices

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Applies to every device, but optional if you know you'll only *ever* have one at a time in your system						

unsigned long **GetDevices**(void)

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 number. For example, if the return is 0x00000104, then device indices 2 and 8 are USB devices that use this driver.
Returns 0 if no devices found (which may mean the driver is not installed properly).
Note, 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.

The use of WinUSB limits each device to one process at a time, and all available devices are opened in order to get a device index. To close devices so that other processes can open them, use AIOUSB_CloseDevice().

QueryDeviceInfo

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Applies to every device, but optional if you know you'll only *ever* have one *model* of card at a time in your system						

unsigned long **QueryDeviceInfo**(
unsigned long DeviceIndex - number from 0-31 of the device you want to query.
unsigned long *pPID - DWORD gets set to the ProductID of the device at DeviceIndex.
unsigned long *pNameSize - DWORD that specifies the size of the pName buffer before the call. After the call it's set to the size needed for the entire name. If your buffer is too small, the data will be truncated.
char *pName - pointer to char[] buffer. This is an array of characters, not a null-terminated string. Length of string is passed back via pNameSize
unsigned long *pDIOBytes - DWORD gets set to how many bytes of DIO the device supports.
unsigned long *pCounters - DWORD gets set to how many 8254-compatible counters are available.
)

AIOUSB_CloseDevice

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
Applies to every device, but only when using multiple simultaneous proccesses						

unsigned long **AIOUSB_CloseDevice**(
unsigned long DeviceIndex - number from 0-31 of the device you want to close.
)
Explicitly closes handles to a device, mainly so that it can be opened by another process. The standard use for that situation is to call GetDevices() to get the bitmask of devices, call QueryDeviceInfo() for devices as needed to find the one(s) you want to work with, then call AIOUSB_CloseDevice() for each other found device.

ClearDevices

APPLIES TO						
RS-232 -422 / -485	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						

unsigned long **ClearDevices**(void)
Closes handles and clears records of unplugged devices.

ResolveDeviceIndex

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ResolveDeviceIndex**(
unsigned long DeviceIndex - the device index you want to resolve, usually diFirst or diOnly
)
Returns a device index from 0-31 corresponding to the index passed in, or FFFFFFFF hex if it can't be resolved.

GetDeviceByEEPROMByte

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **GetDeviceByEEPROMByte**(
unsigned char Data - the byte at the beginning of the EEPROM of the device you want
)
Finds a device with the specified byte at address 0x000 in the custom EEPROM area.

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().) If there aren't any matching devices, returns FFFFFFFF hex.

Note: avoid 0x00 and 0xFF, since those can match uninitialized EEPROMs.

GetDeviceByEEPROMData

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **GetDeviceByEEPROMData**(
unsigned long StartAddress - the address of the beginning of the block to look for
unsigned long DataSize - the length of the block to look for
unsigned char *pData - a pointer to the block of data to look for
)
Finds a device based on custom EEPROM data, like GetDeviceByEEPROMByte(), except that it can look for a larger block at any position in the custom EEPROM area.

DIGITAL INPUT / OUTPUT

DIO_Configure

APPLIES TO						
RS-232 -422 / -485	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						

unsigned long **DIO_Configure**(
unsigned long DeviceIndex - number from 0-31 of the device you want to configure
unsigned char bTristate - boolean value. 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.
void *pOutMask - a pointer to the first element of an array of bits; one bit per I/O port. Each "1" bit in the array indicates that the corresponding port of the device is Output. In this context "port" 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."
void *pData - a pointer to the first element of 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.
)

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

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

While byte arrays are the most generic, most of these are small enough to use a more specific type. 12 bytes doesn't match such a type, however.

	Pascal	Visual Basic 6	Visual Basic .NET	C/C++
1 byte	Byte	Byte	Byte	unsigned char
2 bytes	Word	Integer	Short	unsigned short
4 bytes	LongWord	Long	Integer	unsigned long

DIO_ConfigureEx

APPLIES TO						
RS-232 -422 / -485	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”						

unsigned long **DIO_ConfigureEx**(
unsigned long DeviceIndex - number from 0-31 of the device you want to configure
void *pOutMask - a pointer to the first element of an array of bytes; one byte per 8 ports or fraction. Each "1" bit in the array indicates that the corresponding byte of the device is Output.
void *pData - a pointer to the first element of an array of bytes; one byte per port. 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.

void *pTristateMask - a pointer to the first element of an array of bytes; one byte per 8 tristate groups or fraction. Each "1" bit in the array causes the corresponding tristate group to enter tristate (high-impedance) mode. A “0" bit 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.

)

DIO_ConfigurationQuery

APPLIES TO						
RS-232 -422 / -485	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...”

unsigned long **DIO_ConfigurationQuery**(
unsigned long DeviceIndex - number from 0-31 of the device whose configuration you want to query
void *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
void *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
)

DIO_WriteAll

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

The most efficient method of writing to digital outputs

unsigned long **DIO_WriteAll**(
unsigned long DeviceIndex - number from 0-31 of the device to which you wish to write all output bits
void *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
)
Note that the size of "all" is the same as the size of the data given under DIO_Configure.

DIO_Write8

APPLIES TO						
RS-232 -422 / -485	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.

unsigned long **DIO_Write8**(
unsigned long DeviceIndex - number from 0-31 of the device to which you want to write an output byte
unsigned long ByteIndex - Number of the byte you wish to change. Writes to bytes configured as inputs are ignored
unsigned char Data - one byte. The byte will be copied to the port outputs. Each set bit will cause the same port bit to be set to "1"
)

DIO_Write1

APPLIES TO						
RS-232 -422 / -485	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.

unsigned long **DIO_Write1**(
unsigned long DeviceIndex - number from 0-31 of the device to which you want to write an output bit
unsigned long BitIndex - Number of the bit you wish to change. Writes to bits configured as inputs are ignored
unsigned char bData - boolean. TRUE will set the bit to "1", FALSE will clear the bit to "0"
)

DIO_ReadAll

APPLIES TO						
RS-232 -422 / -485	**DIGITAL INPUTS**	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

The most efficient method of reading digital inputs.

unsigned long **DIO_ReadAll**(
unsigned long DeviceIndex - number from 0-31 of the device from which you wish to read all digital bits
void *Buffer - 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.
)
Note that the size of "all" is the same as the size of the data given under DIO_Configure.

DIO_Read8

APPLIES TO						
RS-232 -422 / -485	**DIGITAL INPUTS**	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

The driver performs a ReadAll and returns the selected byte

unsigned long **DIO_Read8**(
unsigned long DeviceIndex - number from 0-31 of the device from which you wish to read a byte
unsigned long ByteIndex - Number of the byte you wish to read
unsigned char *pBuffer - pointer to a byte in which the input byte will be stored. Data read from ports configured as output results in a "read-back" of the output
)

DIO_Read1

APPLIES TO						
RS-232 -422 / -485	**DIGITAL INPUTS**	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

The driver performs a ReadAll and returns the selected bit

unsigned long **DIO_Read1**(
unsigned long DeviceIndex - number from 0-31 of the device from which you wish to read a bit
unsigned long BitIndex - Number of the bit you wish to read
unsigned char *pBuffer - pointer to a byte which will be set to zero or one based on the input bit. Data read from ports configured as output results in a "read-back" of the output
)

DIO_StreamOpen

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	**BUFFERED DI, DO, DIO**	COUNTER TIMERS

unsigned long **DIO_StreamOpen**(
unsigned long DeviceIndex - number from 0-31 of the device through which you wish to stream data
unsigned long bIsRead - boolean. TRUE will open a stream for reading, FALSE will open a stream for writing
)

DIO_StreamClose

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	**BUFFERED DI, DO, DIO**	COUNTER TIMERS

unsigned long **DIO_StreamClose**(
unsigned long DeviceIndex - number from 0-31 of the device whose stream you wish to close
)

DIO_StreamSetClocks

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	**BUFFERED DI, DO, DIO**	COUNTER TIMERS

unsigned long **DIO_StreamSetClocks**(
unsigned long DeviceIndex - number from 0-31 of the device for which you wish to set stream clocks
double *ReadClockHz - a pointer to an IEEE double-precision value indicating the desired frequency of an internal read clock; it will be changed to the actual frequency achieved. Use "0" for an external read clock
double *WriteClockHz - a pointer to an IEEE double-precision value indicating the desired frequency of an internal write clock; it will be changed to the actual frequency achieved. Use "0" for an external write clock
)

DIO_StreamFrame

APPLIES TO						
RS-232 -422 / -485	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”						

unsigned long **DIO_StreamFrame**(
unsigned long DeviceIndex - number from 0-31 of the device through which you wish to stream data
unsigned long FramePoints - number of WORD-sized points you wish to stream
unsigned short *pFrameData - pointer to the beginning of the block of data you wish to stream
unsigned long *BytesTransferred - pointer to a DWORD that will receive the amount of data actually transferred, in BYTEs
)

COUNTER / TIMERS

An important note about the following CTR_ 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):

$CounterIndex_{sequential} = BlockIndex * 3 + CounterIndex_{Primary}$

Please note, 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.

CTR_8254Mode

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **CTR_8254Mode**(
unsigned long DeviceIndex - number from 0-31 of the device on which you wish to configure an 8254 mode
unsigned long BlockIndex - number indicating which 8254 you wish to configure.
unsigned long CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to configure
unsigned long Mode - a number from 0-5 specifying which 8254 mode you want the specified counter to be.
)
Note: issuing a mode to an 8254 counter without also issuing a load causes the counter to cease counting. This allows you to use the counter as a digital output: mode 0 causes the counter output to immediately clear to zero, and mode 1 causes the counter output to immediately set to one.

CTR_8254Load

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **CTR_8254Load**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to load an 8254 counter
unsigned long BlockIndex - number indicating which 8254 you wish to load
unsigned long CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to load
unsigned short LoadValue - a number from 0 to 65535 which you wish loaded into the specified counter
)

CTR_8254ModeLoad

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **CTR_8254ModeLoad**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to mode and load an 8254 counter
unsigned long BlockIndex - number indicating which 8254 you wish to mode and load
unsigned long CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to mode and load
unsigned long Mode - a number from 0-5 specifying which 8254 mode you want the specified counter to be
unsigned short LoadValue - a number from 0 to 65535 which you wish loaded into the specified counter
)

CTR_8254ReadModeLoad

APPLIES TO						
RS-232 -422 / -485	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						

unsigned long **CTR_8254ReadModeLoad**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to read, mode, and load an 8254 counter
unsigned long BlockIndex - number indicating which 8254 you wish to read, mode, and load
unsigned long CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to read, mode, and load
unsigned long Mode - a number from 0-5 specifying which 8254 mode you want the specified counter to be
unsigned short LoadValue - a number from 0 to 65535 which you wish loaded into the specified counter
unsigned short *pReadValue - a pointer to a WORD in which will be stored the value latched and read from the specified counter. The reading is taken *before* the mode and load occur
)

CTR_8254Read

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **CTR_8254Read**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to read an 8254 counter
unsigned long BlockIndex - number indicating which 8254 you wish to read
unsigned long CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to read
unsigned short *pReadValue - a pointer to a WORD in which will be stored the value latched and read from the specified counter
)

CTR_8254ReadAll

APPLIES TO						
RS-232 -422 / -485	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.

unsigned long **CTR_8254ReadAll**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to read all 8254 counters
unsigned short *pData - a pointer to the first of an array of WORDs in which will be stored the values latched and read from the counters
)

This function is currently only supported by the USB-CTR-15.

CTR_8254ReadStatus

APPLIES TO						
RS-232 -422 / -485	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.

unsigned long **CTR_8254ReadStatus**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to read an 8254 counter
unsigned long BlockIndex - number indicating which 8254 you wish to read
unsigned long CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to read
unsigned short *pReadValue - a pointer to a WORD in which will be stored the value latched and read from the specified counter
unsigned char *pStatus - a pointer to a BYTE in which will be stored the status latched and read from the specified counter
)

CTR_StartOutputFreq

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **CTR_StartOutputFreq**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to output a frequency
unsigned long BlockIndex - number indicating which 8254 you wish to output a frequency from
double *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.
)

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 for 1 second translates directly to Hertz. In the USB-CTR-15 and other supported devices, you can create a known duration by configuring one counter to act as a "gating" signal for any collection of other counters. The other "measurement" counters will only count during the "high" side of the gate signal, which we can control.

So, to measure frequency you 1) create a gate signal of known duration; 2) connect this gating signal to the gate pins of all the "measurement" counters; 3) call CTR_8254SelectGate() to tell the board which counter is generating that gate; and 4) call CTR_8254ReadLatched() periodically 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, or daisy-chain, 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 0.05seconds; if you read 1324 counts via the CTR_8254ReadLatched() call, the frequency would be "1324 / 0.05", or 26.48KHz.

CTR_8254SelectGate

APPLIES TO						
RS-232 -422 / -485	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.

unsigned long **CTR_8254SelectGate**(- This function selects a counter for use as the gate in frequency measurement on other counters, and starts the frequency measurement process.
unsigned long DeviceIndex - number from 0-31 indicating which device you wish to select a gate for
unsigned long GateIndex - number from 0-14 indicating which counter you wish to select as a gate; this is in "blockless" addressing
)

CTR_8254ReadLatched

APPLIES TO						
RS-232 -422 / -485	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.

unsigned long **CTR_8254ReadLatched**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to read all 8254 counters
unsigned short *pData - a pointer to the first of an array of WORDs in which will be stored the values latched and read from the counters. After the array in the pointer buffer is an 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.
)

ANALOG TO DIGITAL

ADC_GetChannelV

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

The other easiest way, but often can't achieve more than 100Hz, slower depending on options.

unsigned long **ADC_GetChannelV**(- This simple function takes A/D data from one channel and converts it to voltage. It also averages oversamples for the channel.
unsigned long DeviceIndex - number from 0-31 indicating from which device you wish to get a channel's data
unsigned long ChannelIndex - number indicating which channel's data you wish to get
double *pBuf - a pointer to a double precision IEEE floating point number which will receive the value read
)

ADC_GetScanV

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

The easiest way, but often can't achieve more than several hundred Hz, slower depending on options.

unsigned long **ADC_GetScanV**(- This simple function takes one scan of A/D data and converts it to voltage. It also averages oversamples for each channel. The array must contain one entry per A/D channel on the board, though only entries [start channel] through [end channel] are altered. On boards with A/D that don't support ADC_SetConfig(), it scans all channels, without oversampling.
unsigned long DeviceIndex - number from 0-31 indicating from which device you wish to get a scan of data
double *pBuf - a pointer to the first of an array of double precision IEEE floating point numbers which will each receive the value read from one channel
)

ADC_GetScan

APPLIES TO						
RS-232 -422 / -485	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, see above.

unsigned long **ADC_GetScan**(- This simple function takes one scan of A/D data. It also averages oversamples for each channel. The array must contain one entry per A/D channel on the board, though only entries [start channel] through [end channel] are altered. On boards with A/D that don't support ADC_SetConfig(), it scans all channels, without oversampling.
unsigned long DeviceIndex - number from 0-31 indicating from which device you wish to get a scan of data
unsigned short *pBuf - a pointer to the first of an array of WORDs which will each receive the value from one channel
)

ADC_GetConfig

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ADC_GetConfig**(
unsigned long DeviceIndex - number from 0-31 indicating from which device you wish to get the A/D configuration
unsigned char *pConfigBuf - a pointer to the first of an array of bytes for configuration data
unsigned long *ConfigBufSize - a pointer to a variable holding the number of configuration bytes to read. Will be set to the number of configuration bytes read
)

ADC_SetConfig

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ADC_SetConfig**(
unsigned long DeviceIndex - number from 0-31 indicating to which device you wish to set the A/D configuration
unsigned char *pConfigBuf - a pointer to the first of an array of config bytes
unsigned long *ConfigBufSize - a pointer to a variable holding the number of config bytes to write. Will be set to the number of config bytes written
)

Configuration bytes for analog input boards(the USB-AI16-16 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 all zeroes is close to an "ordinary" use; you'll likely want to set timer or external trigger, and start and end channels. The extended channel byte only applies to boards with more than 16 channels.

Range codes (config bytes 00h-0Fh) map to channels as follows:

Config Byte	00h	01h	...	07h	08h	...	0Eh	0Fh
16-Channel Boards								
lower channel	0	1	...	7	8	...	14	15
upper channel	8	9	...	F	N/A, don't add 08			
"64M" Boards (64-channel)								
lower channels	0-3	4-7	...	28-31	32-35	...	56-59	60-63
upper channels	32-35	36-39	...	60-63	N/A, don't add 08			
Other Boards (32-, 64-, 96-, or 128-channel)								
lower channels	0-7	8-15	...	56-63	64-71	...	112-119	120-127
upper channels	always differential in hardware, don't add 08							

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 08 to the range code for any "lower" channel(s) to pair them with "upper" channel(s) in differential mode.

Calibration codes (config byte 10h) are as follows:

Cal. Code	00h	01h	03h	05h	07h
Effect	Acquire Normal Data	Acquire Cal. Unipolar Ground	Acquire Cal. Unipolar Reference	Acquire Cal. Bipolar Ground	Acquire Cal. Bipolar Reference
Target		0V @ 0-10V	9.9339V @ 0-10V	0V @ ±10V	9.8678V @ ±10V

Trigger & counter bits (config byte 11h) are as follows:

Bit	7	6	5	4	3	2	1	0
Value	Reserved, use 0			CTR0 EXT	Falling Edge	Scan	External Trigger	Timer Trigger

• If CTR0 EXT is set, counter 0 is externally-triggered; otherwise, counter 0 is triggered by the onboard 10MHz clock.
• If Falling Edge is set, A/D is triggered by the falling edge of its trigger source; otherwise, A/D 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 to end, oversampling if so configured, at maximum speed. Otherwise, a single A/D trigger will cause a single acquisition, "walking" through oversamples and channels.
• If External Trigger is set, the external A/D trigger pin is an A/D trigger source. Otherwise, it's ignored.
• If Timer Trigger is set, counter 2 is an A/D trigger source. Otherwise, it's ignored.

Start & end channel (config byte 12h) for 16-channel analog input boards are the start channel (0-F) in bits 0-3, and the end channel (0-F) in bits 4-7. For boards with more than 16 channels, the start & end channel are split among this config byte and the extended channel (config byte 14h), as follows:

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.

Oversample (config byte 13h) 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.

Extended channel (config byte 14h) is involved with the start & end channel, above.

ADC_RangeAll

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ADC_RangeAll**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to set A/D ranges
unsigned char *pRangeCodes - a pointer to the first of an array of 16 bytes, each of which contains a range code. This does not include single-ended/differential configuration; to configure single-ended/differential on a per-channel basis, use ADC_Range1() or ADC_SetConfig()
unsigned long bDifferential - boolean value. Use FALSE for 16-channel single-ended mode, use TRUE for 8-channel differential mode
)

ADC_Range1

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ADC_Range1**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to set an A/D channel range
unsigned long ADChannel - number from 0-15 indicating an A/D channel on the device
unsigned char RangeCode - a byte range code. See above for details
unsigned long bDifferential - boolean value. For channels 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
)

ADC_SetScanLimits

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ADC_SetScanLimits**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to set A/D scan limits
unsigned long StartChannel - the number of the first channel you want in a scan
unsigned long EndChannel - the number of the last channel you want in a scan
)

ADC_ADMode

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

The USB-AI12-16***E*** does not support calibration, and will use CalMode 00

unsigned long **ADC_ADMode**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to set overall A/D parameters
unsigned char TriggerMode - byte indicating which A/D trigger source to use, see the manual for details. Also sets the clock source for counter 0
unsigned char CalMode - byte indicating which A/D source to use - 00 hex for actual inputs, 01 hex for calibration unipolar ground reference, 03 hex for calibration unipolar high reference, 05 hex for calibration bipolar ground reference, 07 hex for calibration bipolar high reference. Other values cause the call to fail, returning "ERROR_INVALID_PARAMETER"(equal to 87 decimal).
)

ADC_SetOversample

APPLIES TO						
RS-232 -422 / -485	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 inter-channel delay.

unsigned long **ADC_SetOversample**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to set the A/D oversample
unsigned char Oversample - the number of extra samples to take from each channel in a scan
)

ADC_SetCal

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

USB-AI16-16A and USB-AI12-16A only.

unsigned long **ADC_SetCal**(
unsigned long DeviceIndex - number from 0-31 indicating to which device you wish to upload a calibration file
char *CalFileName - either the file name of a calibration file, or a command string. 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 generate an "uncalibrated" calibration file and upload that
)

ADC_QueryCal

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ADC_QueryCal**(- This function returns "ERROR_SUCCESS"(equal to 0) if the indicated device supports A/D calibration, or "ERROR_NOT_SUPPORTED"(equal to 50 decimal) if it has A/D but doesn't support calibration
unsigned long DeviceIndex - number from 0-31 indicating which device's ability you wish to query
)

ADC_Initialize

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-AI16-16A and USB-AI12-16A only.						

unsigned long **ADC_Initialize**(
unsigned long DeviceIndex - number from 0-31 indicating which device you wish to set A/D configuration on and upload a calibration file to
unsigned char *pConfigBuf - a pointer to the first of an array of configuration bytes
unsigned long *ConfigBufSize - a pointer to a variable holding the number of configuration bytes to write. Will be set to the number of configuration bytes written
char *CalFileName - the file name of a calibration file, or a command string. See ADC_SetCal() for details.
)

ADC_BulkAcquire

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ADC_BulkAcquire**(
unsigned long DeviceIndex - number from 0-31 indicating from which device you wish to acquire bulk data
unsigned long BufSize - the size, in bytes, of the buffer to receive the data
void *pBuf - a pointer to the beginning of the buffer to receive the data
)
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						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ADC_BulkPoll**(
unsigned long DeviceIndex - number from 0-31 indicating from which device you wish to query A/D status
unsigned long *BytesLeft - a pointer to a variable which will be set to the number of bytes of A/D data remaining to be taken
)
Note that 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_BulkContinuousCallbackStart

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ADC_BulkContinuousCallbackStart**(
unsigned long DeviceIndex - number from 0-31 indicating which device you wish to start
unsigned long BufSize - number of bytes (a multiple of 512) for each buffer in the software FIFO
unsigned long BaseBufCount - number of buffers in the software FIFO, for example 64. Minimum 2
unsigned long Context - any value, will be passed to the callback
void *pCallback - pointer to an ADContCallback() function to receive buffers
)
Starts a continuous bulk acquire process. A worker thread will acquire data, however the board is configured, 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:

```
Hz = 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);
```

void **ADContCallback**(
unsigned short *pBuf - pointer to the first of an array of WORD samples
unsigned long BufSize - size, in bytes, of the array passed in pBuf; can be zero
unsigned long Flags - a bitmask of flags, see table
unsigned long Context - a copy of the Context parameter to ADC_BulkContinuousCallbackStart()
)
This is a placeholder for the callback function passed to ADC_BulkContinuousCallbackStart(); 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	Flags & 1	Obsolete. (Previously, this was set on the first buffer of a terminal count. The current version doesn't use terminal counts.)
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						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	**ANALOG INPUTS**	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **ADC_BulkContinuousEnd**(
unsigned long DeviceIndex - number from 0-31 indicating which device you wish to end continuous acquisition on
unsigned long *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
)

DIGITAL TO ANALOG

DACDirect

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **DACDirect**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to set a DAC value
unsigned short Channel - number from 0-7 indicating which DAC you wish to set
unsigned short Value - number from 000h-FFFh indicating the count value to which you wish to set the DAC; 000h indicates the lowest DAC level, FFFh indicates the highest DAC level, other values are proportional
)

DACMultiDirect

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS
The most efficient method to output voltages						

unsigned long **DACMultiDirect**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to set a DAC value
unsigned short *pDACData - a pointer to the first of an array of WORDs, consisting of channel/value pairs; channels are from 0-7, values are from 000h-FFFh, as for DACDirect()
unsigned long DACDataCount - number indicating how many channel/value **pairs** are in the array referenced by pDACData
)

DACSetBoardRange

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **DACSetBoardRange**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to set the DAC range
unsigned long RangeCode - the range code to set for the board; see the manual for your device's range codes
)

DACOutputProcess

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.						

unsigned long **DACOutputProcess**(- 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 EOD bit, see DACOutputFrameRaw() below for details
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to begin DAC streaming

double *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.
unsigned long NumSamples - the total number of samples to output. Notably, this is not a number of "points"
unsigned short * SampleData - a pointer to the first of 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, see DACOutputFrameRaw() below
)

DACOutputOpen

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.						

unsigned long **DACOutputOpen**(- 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.
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to begin DAC streaming
double *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.
)

DACOutputOpenNoClear

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.						

unsigned long **DACOutputOpenNoClear**(- This function is now equivalent to DACOutputOpen(). The "clear" is no longer useful.
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to begin DAC streaming
double *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.
)

DACOutputClose

APPLIES TO						
RS-232 -422 / -485	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.*						

unsigned long **DACOutputClose**(- This function ends and closes a DAC streaming process.
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to end DAC streaming
unsigned long bWait - reserved for future expansion; currently, this function always waits for the streaming process to complete before returning to the caller
)

DACOutputCloseNoEnd

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.						

unsigned long **DACOutputCloseNoEnd**(- This function closes a DAC streaming process *without* ending it. This is most useful when you've set LOOP or EOM via DACOutputFrameRaw().
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to end DAC streaming
unsigned long bWait - reserved for future expansion; currently, this function always waits for the streaming process to complete before returning to the caller
)

DACOutputSetCount

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.						

unsigned long **DACOutputSetCount**(- This function sets the number of DACs involved in each DAC streaming point henceforth. When the driver connects to the device, this is initialized to 5 (for ILDA use). You can set this freely between calls to DACOutputFrame() and/or DACOutputFrameRaw() if you wish.
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to set the number of DACs in future points
unsigned long NewCount - number from 1-8 indicating the number of DACs in future points
)

DACOutputFrame

APPLIES TO						
RS-232 -422 / -485	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.*						

unsigned long **DACOutputFrame**(- This function writes a group of points(a “frame”) into the DAC stream.
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to stream a frame of DAC points
unsigned long FramePoints - the number of points in the frame
unsigned short * FrameData - a pointer to the first of an array of WORDs; each DAC value is stored in a WORD, so it should contain (DAC count) × (points in the frame) WORDs
)
All points in a frame 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 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 larger hardware buffer and available time on the USB bus.

DACOutputFrameRaw

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.						

unsigned long **DACOutputFrameRaw**(- This function is similar to DACOutputFrame(), except the features are controlled by the upper bits in the data array. This provides the greatest flexibility, at the cost of complexity.
unsigned long DeviceIndex - same as for DACOutputFrame()
unsigned long FramePoints - same as for DACOutputFrame()
unsigned short * FrameData - same as for DACOutputFrame(); notably, the DAC count determines the number of samples, even though you can place EOD bits(see below) as you wish
)
The meanings of the bits are as follows:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Meaning	EOM	EOF	EOD	LOOP	DAC Value											

• If EOM("End Of Movie") is set, the board will stop the waveform after outputting the sample. (Unless LOOP is also set, see below.)
• If EOF("End Of Frame") is set, the frame pin will be pulsed. This can be used for other things via DACOutputFrameRaw(), but is automatically set on the last sample of each frame by DACOutputFrame().
• If EOD("End Of DACs") is set, the next sample will go to the first DAC; otherwise, it will go to the next DAC in series. (If this sample goes to the last DAC, this bit isn't needed, but should be set anyway for future expansion.) Going to the first DAC also ends the point, which is significant because each tick clocks out a point.
• If LOOP is set, the board will "jump" to the beginning of its buffer after outputting the sample. (Unless EOM is also set, see below.) This can be used to load a "repeatable" waveform, like a sine wave, 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.

Note that the EOM and LOOP bits are for mutually exclusive uses. Setting them both issues extended commands instead of treating the sample normally. No extended commands are yet defined, but the feature is reserved for future expansion.

DACOutputStart

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	**ANALOG OUTPUTS**	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.						

unsigned long **DACOutputStart**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to start DAC streaming
)
Normally, DAC streaming will be automatically started by streaming 1¼ SRAMs worth of data (160K bytes, i.e. 81920 samples). It's only if you're using a smaller amount of data that you'd need to "manually" start DAC streaming with this function.

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.

DACOutputSetInterlock

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS
USB-DA12-8A only.						

unsigned long **DACOutputSetInterlock**(
unsigned long DeviceIndex - number from 0-31 indicating on which device you wish to enable or disable interlock
unsigned long 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)
)

GENERAL FUNCTIONS

GetDeviceSerialNumber

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **GetDeviceSerialNumber**(
unsigned long DeviceIndex - number from 0-31 of the device whose serial number you wish to read.
unsigned __int64 *pSerialNumber - pointer to an 8-byte (64-bit) value to fill with the serial number.
)

CustomEEPROMWrite

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **CustomEEPROMWrite**(- This function writes to the custom EEPROM area, so you can store data there for your own use.
unsigned long DeviceIndex - number from 0-31 of the device to which you wish to write custom EEPROM data.
unsigned long StartAddress - number from 0x000 to 0x1FF of the first custom EEPROM byte you wish to write to.
unsigned long 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.
void *Data - pointer to the start of a block of bytes to write to the custom EEPROM area.
)

CustomEEPROMRead

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **CustomEEPROMRead**(- This function reads data written by CustomEEPROMWrite.
unsigned long DeviceIndex - number from 0-31 of the device from which you wish to read custom EEPROM data.
unsigned long StartAddress - number from 0x000 to 0x1FF of the first custom EEPROM byte you wish to read from.
unsigned long *DataSize - pointer to a variable holding the number of custom EEPROM bytes to read. The last custom EEPROM byte is 0x1FF, so StartAddress plus *DataSize can't be greater than 0x200.
void *Data - pointer to the start of a block of bytes to fill with data read from the custom EEPROM area.
)

AIOUSB_SetStreamingBlockSize

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **AIOUSB_SetStreamingBlockSize**(
unsigned long DeviceIndex - number from 0-31 indicating which device's streaming block size you wish to set.
unsigned long 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.
)

AIOUSB_ClearFIFO

APPLIES TO						
RS-232 -422 / -485	DIGITAL INPUTS	DIGITAL OUTPUTS	ANALOG INPUTS	ANALOG OUTPUTS	BUFFERED DI, DO, DIO	COUNTER TIMERS

unsigned long **AIOUSB_ClearFIFO**(
unsigned long DeviceIndex - number from 0-31 of the device you wish to clear the streaming FIFO on.
unsigned long Method - 0 to simply clear the FIFO right away, others per the table below
)

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 as 0)
5	As 0, but also abort stream

.NET

APPLIES TO			
C#	VB.NET	Visual C CLI	and all other .net languages

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

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

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

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

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

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

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

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

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

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

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

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