Embedded Solutions

Overview

Synopsis

This document is a reference manual for the AIOUSB library. It describes all the API functions, as well as global variables and constants. It also provides some guidance as to how to use AIOUSB. This document is intended to serve both the Linux and Windows implementations of AIOUSB.

Document Organization and Conventions

The API functions are grouped by category, and within each group they are sorted in alphabetical order.

Everything is described in this frame, so the best way to navigate around the document is by using the menu at left.

Where appropriate, Linux- or Windows-specific notes are included.

Names of ***files***, ***program statements***, ***functions***, ***variables*** and ***constants*** are highlighted using a different text style than the regular body text.

Instead of replicating the same information dozens of times, commonly used variables and constants are described in a single place.

Sample Program

Below is an example of a minimalist C++ program for **Linux** that demonstrates how to properly initialize AIOUSB, query the bus for devices, query an individual device for its product ID and name and then terminate use of AIOUSB. If AIOUSB and ***libusb*** are properly installed, you should be able to copy this sample program from this document, paste it into a file named ***test.cpp*** and compile it using the command shown below. This program uses the first ADL device it finds on the bus. A "real" application would probably be looking for devices of a particular type, which can be determined from the product ID.

```
/*
 * compile with: g++ test.cpp -laiousbcpp -lusb-1.0 -o test
 */
#include <aiousb.h>
#include <stdio.h>
using namespace AIOUSB;
int main( int argc, char **argv ) {
  unsigned long result = AIOUSB_Init();
  if( result == AIOUSB_SUCCESS ) {
    const int MAX_NAME_SIZE = 20;
    char name[ MAX_NAME_SIZE + 2 ];
    unsigned long productID
      , nameSize = MAX_NAME_SIZE;
    result = QueryDeviceInfo( diFirst
      , &productID, &nameSize, name, NULL, NULL );
    if( result == AIOUSB_SUCCESS ) {
      name[ nameSize ] = 0;
      printf( "Found a device with product ID %#06x and name '%s'\n"
        , ( unsigned ) productID, name );
    } else
      printf( "Error '%s' querying device\n"
        , AIOUSB_GetResultCodeAsString( result ) );
    AIOUSB_Exit();
  }  // if( result ...
  return ( int ) result;
}  // main()
```

The above sample program can be ported to Windows by doing the following:

- Omitting the statement ***using namespace AIOUSB;***
- Changing the result code names from ***AIOUSB_ *** to ***ERROR_ ***
- Removing the calls to ***AIOUSB_Init()*** and ***AIOUSB_Exit()***
- Removing the call to ***AIOUSB_GetResultCodeAsString()*** and printing the result code as a number instead of a string
- Using a different command line - one appropriate for Windows - to compile the program

General Functions

unsigned long AIOUSB_ClearFIFO(unsigned long DeviceIndex, unsigned long Method)

Clears the streaming FIFO, using one of the method codes below.

Applies To

Analog input, buffered digital I/O; Linux, Windows

Parameters

DeviceIndex - a standard device index

Method - the method to use when clearing the FIFO. May be one of:

CLEAR_FIFO_METHOD_IMMEDIATE (0) - Clear FIFO as soon as command received (and disable auto-clear)

CLEAR_FIFO_METHOD_AUTO (1) - Enable auto-clear FIFO every falling edge of DIO port D bit 1 (on digital boards, analog boards treat as 0)

CLEAR_FIFO_METHOD_IMMEDIATE_AND_ABORT (5) - Clear FIFO as soon as command received (and disable auto-clear), and abort stream

CLEAR_FIFO_METHOD_WAIT (86) - Clear FIFO and wait for it to be emptied

Linux: Use the named constants listed above.

Windows: Use the numeric values shown in parenthesis.

Return Value

A standard result code

---

unsigned long AIOUSB_SetStreamingBlockSize(unsigned long DeviceIndex, unsigned long BlockSize)

Sets the streaming block size.

Applies To

Analog input, buffered digital I/O; Linux, Windows

Parameters

DeviceIndex - a standard device index

BlockSize - the 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.

Return Value

A standard result code

---

unsigned long ClearDevices(void)

Closes handles and clears records of unplugged devices.

Applies To

All products; Linux, Windows

Return Value

A standard result code

---

unsigned long CustomEEPROMRead(unsigned long DeviceIndex, unsigned long StartAddress, unsigned long *DataSize, void *Data)

Reads data from the custom programming area of the device EEPROM.

Applies To

All products; Linux, Windows

Parameters

DeviceIndex - a standard device index

StartAddress - number from 0x000 to 0x1FF of the first custom EEPROM byte you wish to read from.

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.

Data - pointer to the start of a block of bytes to fill with data read from the custom EEPROM area.

Return Value

A standard result code

---

unsigned long CustomEEPROMWrite(unsigned long DeviceIndex, unsigned long StartAddress, unsigned long DataSize, void *Data)

Writes data to the custom programming area of the device EEPROM.

Applies To

All products; Linux, Windows

Parameters

DeviceIndex - a standard device index

StartAddress - number from 0x000 to 0x1FF of the first custom 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.

Return Value

A standard result code

---

unsigned long GetDeviceBySerialNumber(const __uint64_t *pSerialNumber) (Linux)
unsigned long GetDeviceBySerialNumber(const unsigned __int64 *pSerialNumber) (Windows)

Searches the bus for an ADL device with the specified serial number. In the unlikely event that multiple devices have the same serial number, ***GetDeviceBySerialNumber()*** returns a device index to the first such device found.

Unlike ***GetDevices()***, which clears and rebuilds the internal list of devices, ***GetDeviceBySerialNumber()*** is "non-destructive" and simply searches the existing internal list.

Linux: also see ***AIOUSB_GetDeviceByProductID()***.

Applies To

All products; Linux, Windows

Parameters

pSerialNumber - pointer to an 8-byte (64-bit) value containing the serial number to search for

Return Value

A standard device index if the device was found, or ***diNone*** if no device was found with the specified serial number

---

unsigned long GetDevices(void)

Gets a "list" of all the ADL devices found on the USB bus. This "list" is returned in the form of a 32-bit device mask. Each bit set to a "1" indicates an ADL device was detected at the device index corresponding to the set bit number. For example, if the return value is 0x00000104 then ***DeviceIndex*** #2 and #8 correspond to devices that can be controlled by this driver. Returns 0 if no devices were found, which may mean the driver is not installed properly. This function does not return a single device index, but a pattern of bits indicating all the detected devices.

Also see ***GetDeviceBySerialNumber()***.

Linux: also see ***AIOUSB_GetDeviceByProductID()***.

Windows: A limitation in the current .SYS file prevents detection of more than 32 ADL devices connected to one computer simultaneously. Let us know if this is of any concern for your application.

Applies To

All products; Linux, Windows

Return Value

A 32-bit device mask (0 if no devices found).

---

unsigned long GetDeviceSerialNumber(unsigned long DeviceIndex, __uint64_t *pSerialNumber) (Linux)
unsigned long GetDeviceSerialNumber(unsigned long DeviceIndex, unsigned __int64 *pSerialNumber) (Windows)

Gets the device's unique serial number.

Applies To

All products; Linux, Windows

Parameters

DeviceIndex - a standard device index

pSerialNumber - pointer to an 8-byte (64-bit) value to fill with the serial number.

Return Value

A standard result code

---

unsigned long QueryDeviceInfo(unsigned long DeviceIndex, unsigned long *pPID, unsigned long *pNameSize, char *pName, unsigned long *pDIOBytes, unsigned long *pCounters)

Gets the device's specific properties.

Linux: also see ***AIOUSB_GetDeviceProperties()***.

Applies To

All products; Linux, Windows

Parameters

DeviceIndex - a standard device index

pPID - pointer to 32-bit variable in which to store the product ID of the device.

pNameSize - pointer to 32-bit value which specifies the size of the ***pName*** buffer. It gets set to the number of bytes actually required by the name. If ***pNameSize*** is too small to accommodate the entire name, the name will be truncated to fit within your buffer.

pName - pointer to ***char[]*** buffer in which to receive the device name. This is an array of characters, not a null-terminated string. Length of string is passed back via ***pNameSize***.

pDIOBytes - pointer to 32-bit value which gets set to how many bytes of DIO the card supports.

pCounters - pointer to 32-bit value which gets set to how many 8254-compatible counters are available.

Return Value

A standard result codeunsigned long ResolveDeviceIndex(unsigned long DeviceIndex)

Converts a possibly abstract device index into an absolute device index.

Applies To

All products; Linux, Windows

Parameters

DeviceIndex - the device index you want to resolve. This index may be a standard device index, in which case the same index will be returned, or one of these special constants:

diFirst (0xFFFFFFFEul) - obtains the index of the first ADL device found on the bus

diOnly (0xFFFFFFFDul) - obtains the index of the only ADL device found on the bus, meaning that there must be only one ADL device on the bus

Return Value

A standard device index or ***diNone (0xFFFFFFFFul)*** if a valid device index cannot be returned.

Extended General Functions

void AIOUSB_Exit()

Shuts down the AIOUSB API prior to exiting the program. **Must** be called if the call to ***AIOUSB_Init()*** was successful. No AIOUSB function calls may be made after ***AIOUSB_Exit()*** has been called.

Applies To

All products; Linux

---

unsigned AIOUSB_GetCommTimeout(unsigned long DeviceIndex)

Returns the current timeout setting (in milliseconds) for USB communications.

Applies To

All products; Linux

Parameters

DeviceIndex - a standard device index

Return Value

Current timeout setting (in milliseconds)

---

unsigned long AIOUSB_GetDeviceByProductID(int minProductID, int maxProductID, int maxDevices, int *deviceList)

Like ***GetDevices()*** and ***GetDeviceBySerialNumber()***, ***AIOUSB_GetDeviceByProductID()*** searches for devices on the bus. It returns an array of integers containing the list of devices found that match the search criteria. This array consists of <device index>-<product ID> pairs, making it easy to identify all the devices of interest.

Unlike ***GetDevices()***, which clears and rebuilds the internal list of devices, ***AIOUSB_GetDeviceByProductID()*** is "non-destructive" and simply searches the existing internal list.

If you are only interested in a single product ID, set ***minProductID*** and ***maxProductID*** to the same value.

If you want a list of all product IDs, set ***minProductID*** to 0 and ***maxProductID*** to 0xffff.

Applies To

All products; Linux

Parameters

minProductID - the minimum product ID to search for (0 - 0xffff)

maxProductID - the maximum product ID to search for (0 - 0xffff; must be greater than or equal to ***minProductID***)

maxDevices - the maximum number of devices to return in ***deviceList*** (1 - 127)

deviceList - pointer to an array of integers in which the found device indexes and product IDs will be returned. The size of this array must be equal to *1 + maxDevices * 2*. The array will contain <device index>-<product ID> *pairs*, which is why it must be twice the size of the number of devices requested. The actual number of devices found is returned in the first integer of the array, index [0]. The <device index>-<product ID> pairs follow, beginning with indexes [1] and [2], respectively.

Return Value

A standard result code

---

unsigned long AIOUSB_GetDeviceProperties(unsigned long DeviceIndex, DeviceProperties *properties)

Gets a richer set of device-specific properties. This function obtains a superset of the information obtained by ***QueryDeviceInfo()***.

Applies To

All products; Linux

Parameters

DeviceIndex - a standard device index

properties - pointer to a ***DeviceProperties*** structure in which the device properties will be returned.

Return Value

A standard result code

---

const char *AIOUSB_GetResultCodeAsString(unsigned long result)

Returns the string representation of an ***AIOUSB_ *** result code, useful mainly for debugging purposes.

Applies To

All products; Linux; **debugging**

Parameters

result - a standard result code

Return Value

Pointer to a null-terminated string representation of ***result***

---

const char *AIOUSB_GetVersion()

Returns the AIOUSB module version number as a string with the form, "1.78".

Applies To

All products; Linux

Return Value

Pointer to a null-terminated version number string

---

const char *AIOUSB_GetVersionDate()

Returns the AIOUSB module version date as a string with the form, "15 November 2009".

Applies To

All products; Linux

Return Value

Pointer to a null-terminated version date string

---

unsigned long AIOUSB_Init()

Initializes AIOUSB API. **Must** be called before using any other functions in the AIOUSB module. Automatically scans the USB bus and populates the list of ADL devices found. If this call is successful, then ***AIOUSB_Exit()*** must be called before the program exits.

Applies To

All products; Linux

Return Value

A standard result code

---

void AIOUSB_ListDevices()

A crude means to dump a list of all ADL devices found on the USB bus.

Applies To

All products; Linux; **debugging**

---

unsigned long AIOUSB_SetCommTimeout(unsigned long DeviceIndex, unsigned timeout)

Sets the timeout (in milliseconds) for USB communications. The default setting is 5,000 ms.

Applies To

All products; Linux

Parameters

DeviceIndex - a standard device index

timeout - the new timeout setting (in milliseconds)

Return Value

A standard result code

---

unsigned long AIOUSB_SetMiscClock(unsigned long DeviceIndex, double clockHz)

Specifies a clock frequency (in Hertz) for some of the timer-driven functions. The default setting is 1 Hz.

Applies To

All products; Linux

Parameters

DeviceIndex - a standard device index

clockHz - the new clock frequency in Hertz

Return Value

A standard result code

DIO Functions

unsigned long DIO_ConfigurationQuery(unsigned long DeviceIndex, void *pOutMask, void *pTristateMask)

Gets the configuration of the digital I/O ports.

Applies To

Digital inputs and outputs; Linux, Windows

Parameters

DeviceIndex - a standard 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.

Return Value

A standard result code

---

unsigned long DIO_Configure(unsigned long DeviceIndex, unsigned char bTristate, void *pOutMask, void *pData)

Configures the digital I/O ports.

Applies To

Digital inputs and outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

bTristate - ***AIOUSB_TRUE, TRUE*** causes all bits on the device to enter tristate (high-impedance) mode; ***AIOUSB_FALSE, 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 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 an output. The number of bytes required by the output mask is equal to the number of ports, divided by 8 (one bit controls the direction of an entire port) and rounded up to the next whole value. For example, if the device has 4 ports, then 4 divided by 8 and rounded up is 1; that is, one mask byte is required (actually only the least significant 4 bits of the mask are required).

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 number of bytes required to store all the data for the device is equal to the number or ports, or the number of channels divided by 8 channels/bits per port/byte. For example, if the device has 96 channels, the number of ports/bytes is 96 divided by 8, or 12.

Return Value

A standard result code

---

unsigned long DIO_ConfigureEx(unsigned long DeviceIndex, void *pOutMask, void *pData, void *pTristateMask)

Configures the digital I/O ports.

Applies To

Digital inputs and outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

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 an output. The number of bytes required by the output mask is equal to the number of ports, divided by 8 (one bit controls the direction of an entire port) and rounded up to the next whole value. For example, if the device has 4 ports, then 4 divided by 8 and rounded up is 1; that is, one mask byte is required (actually only the least significant 4 bits of the mask are required).

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 number of bytes required to store all the data for the device is equal to the number or ports, or the number of channels divided by 8 channels/bits per port/byte. For example, if the device has 96 channels, the number of ports/bytes is 96 divided by 8, or 12.

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 â€œ0" 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.

Return Value

A standard result code

---

unsigned long DIO_Read1(unsigned long DeviceIndex, unsigned long BitIndex, unsigned char *pBuffer)

Reads a single bit from a digital I/O port.

Applies To

Digital inputs and outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

BitIndex - Number of the bit you wish to read (0-based)

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.

Return Value

A standard result code

---

unsigned long DIO_Read8(unsigned long DeviceIndex, unsigned long ByteIndex, unsigned char *pBuffer)

Reads a single digital I/O port (8-bits).

Applies To

Digital inputs and outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

ByteIndex - Number of the byte you wish to read (0-based)

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.

Return Value

A standard result code

---

unsigned long DIO_ReadAll(unsigned long DeviceIndex, void *Buffer)

Reads all the digital I/O ports in the device.

Applies To

Digital inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

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. The number of bytes required to store all the data for the device is equal to the number or ports, or the number of channels divided by 8 channels/bits per port/byte. For example, if the device has 96 channels, the number of ports/bytes is 96 divided by 8, or 12.

Return Value

A standard result code

unsigned long DIO_Write1(unsigned long DeviceIndex, unsigned long BitIndex, unsigned char bData)

Writes a single bit to a digital output port.

Applies To

Digital outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

BitIndex - Number of the bit you wish to change (0-based). Writes to bits configured as inputs are ignored.

bData - ***AIOUSB_TRUE, TRUE*** will set the bit to "1"; ***AIOUSB_FALSE, FALSE*** will clear the bit to "0".

Return Value

A standard result code

---

unsigned long DIO_Write8(unsigned long DeviceIndex, unsigned long ByteIndex, unsigned char Data)

Writes a single digital output port (8-bits).

Applies To

Digital outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

ByteIndex - Number of the byte you wish to change (0-based). Writes to bytes configured as inputs are ignored.

Data - The byte will be copied to the port outputs. Each set bit will cause the same port bit to be set to "1".

Return Value

A standard result code

---

unsigned long DIO_WriteAll(unsigned long DeviceIndex, void *pData)

Writes all the digital output ports in the device.

Applies To

Digital outputs; Linux, Windows

Parameters

DeviceIndex - a standard 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. The number of bytes required to store all the data for the device is equal to the number or ports, or the number of channels divided by 8 channels/bits per port/byte. For example, if the device has 96 channels, the number of ports/bytes is 96 divided by 8, or 12.

Return Value

A standard result code

DIO Streaming Functions (Advanced)

unsigned long DIO_StreamClose(unsigned long DeviceIndex)

Closes a digital I/O stream opened by a call to ***DIO_StreamOpen()***.

Applies To

Buffered digital inputs and outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

Return Value

A standard result code

---

unsigned long DIO_StreamFrame(unsigned long DeviceIndex, unsigned long FramePoints, unsigned short *pFrameData, unsigned long *BytesTransferred)

Read from, or write to a digital I/O stream opened by a call to ***DIO_StreamOpen()***.

Applies To

Buffered digital inputs and outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

FramePoints - number of 16-bit points you wish to stream

pFrameData - pointer to the beginning of the block of data you wish to stream

BytesTransferred - pointer to a variable that will receive the amount of data actually transferred, in **bytes**

Return Value

A standard result code

---

unsigned long DIO_StreamOpen(unsigned long DeviceIndex, unsigned long bIsRead)

Opens a digital I/O stream. When you are done using the stream, you must close it by calling ***DIO_StreamClose()***.

Applies To

Buffered digital inputs and outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

bIsRead - ***AIOUSB_TRUE, TRUE*** will open a stream for reading; ***AIOUSB_FALSE, FALSE*** will open a stream for writing

Return Value

A standard result code

---

unsigned long DIO_StreamSetClocks(unsigned long DeviceIndex, double *ReadClockHz, double *WriteClockHz)

Sets the internal read/write clock speed of a digital I/O stream.

Applies To

Buffered digital inputs and outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

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.

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.

Return Value

A standard result code

CTR Functions

The 8254 programming details are not described in this manual. Programming details for the 8254 can be found in the document named ***CD/ChipDocs/8254.pdf***.

Counter Addressing

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 â€œCountersâ€□. ***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.

---

unsigned long CTR_8254Load(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned short LoadValue)

Loads a count value into a counter.

Applies To

Counter timers; Linux, Windows

Parameters

DeviceIndex - a standard device index

BlockIndex - number indicating which 8254 you wish to load

CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to load (see note about counter addressing)

LoadValue - a number from 0 to 65535 which you wish loaded into the specified counter

Return Value

A standard result code

---

unsigned long CTR_8254Mode(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned long Mode)

Sets a counter mode.

Applies To

Counter timers; Linux, Windows

Parameters

DeviceIndex - a standard device index

BlockIndex - number indicating which 8254 you wish to configure

CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to configure (see note about counter addressing)

Mode - a number from 0-5 specifying the 8254 counter mode

Return Value

A standard result code

---

unsigned long CTR_8254ModeLoad(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned long Mode, unsigned short LoadValue)

Sets a counter mode and loads a count value into the counter.

Applies To

Counter timers; Linux, Windows

Parameters

DeviceIndex - a standard device index

BlockIndex - number indicating which 8254 you wish to configure

CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to configure (see note about counter addressing)

Mode - a number from 0-5 specifying the 8254 counter mode

LoadValue - a number from 0 to 65535 which you wish loaded into the specified counter

Return Value

A standard result code

---

unsigned long CTR_8254Read(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned short *pReadValue)

Reads a counter's current count value.

Applies To

Counter timers; Linux, Windows

Parameters

DeviceIndex - a standard device index

BlockIndex - number indicating which 8254 you wish to read

CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to read (see note about counter addressing)

pReadValue - a pointer to a 16-bit integer in which will be stored the value latched and read from the specified counter

Return Value

A standard result code

---

unsigned long CTR_8254ReadAll(unsigned long DeviceIndex, unsigned short *pData)

Reads the current count values of all the counters.

Applies To

USB-CTR-15 only; Linux, Windows

Parameters

DeviceIndex - a standard device index

pData - a pointer to the first of an array of 16-bit integers in which will be stored the values latched and read from the counters

Return Value

A standard result code

---

unsigned long CTR_8254ReadLatched(unsigned long DeviceIndex, unsigned short *pData)

Reads the current count values of all the counters.

Applies To

USB-CTR-15 only; Linux, Windows

Parameters

DeviceIndex - a standard device index

pData - a pointer to the first of an array of 16-bit integers in which will be stored the values latched and read from the counters. After the array in the

pointer buffer is an additional 8-bit byte. This byte contains useful information when optimizing polling rates. If the value of the byte is â€œ0", youâ€™re looking at old data, and are reading faster than your gate signal is running.

Return Value

A standard result code

---

unsigned long CTR_8254ReadModeLoad(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned long Mode, unsigned short LoadValue, unsigned short *pReadValue)

Reads a counter's current count value, then sets a new mode and loads a new count value into the counter.

Applies To

Counter timers; Linux, Windows

Parameters

DeviceIndex - a standard device index

BlockIndex - number indicating which 8254 you wish to read, mode, and load

CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to read, mode, and load (see note about counter addressing)

Mode - a number from 0-5 specifying the 8254 counter mode

LoadValue - a number from 0 to 65535 which you wish loaded into the specified counter

pReadValue - a pointer to a 16-bit integer in which will be stored the value latched and read from the specified counter. The reading is taken *before* the mode and load occur.

Return Value

A standard result code

---

unsigned long CTR_8254ReadStatus(unsigned long DeviceIndex, unsigned long BlockIndex, unsigned long CounterIndex, unsigned short *pReadValue, unsigned char *pStatus)

Reads a counter's current count value and status.

Applies To

Counter timers; Linux, Windows

Parameters

DeviceIndex - a standard device index

BlockIndex - number indicating which 8254 you wish to read

CounterIndex - number from 0-2 indicating which counter on the specified 8254 you wish to read (see note about counter addressing)

pReadValue - a pointer to a 16-bit integer in which will be stored the value latched and read from the specified counter.

pStatus - a pointer to an 8-bit byte in which will be stored the status latched and read from the specified counter.

Return Value

A standard result code

---

unsigned long CTR_8254SelectGate(unsigned long DeviceIndex, unsigned long GateIndex)

Selects the counter to use as a gate in frequency measurement on other counters.

Applies To

USB-CTR-15 only; Linux, Windows

Parameters

DeviceIndex - a standard device index

GateIndex - number from 0-14 indicating which counter you wish to select as a gate (using sequential counter addressing)

Return Value

A standard result code

---

unsigned long CTR_StartOutputFreq(unsigned long DeviceIndex, unsigned long BlockIndex, double *pHz)

Selects an output frequency for a counter block and starts the counters.

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 â€œgatingâ€□ signal for any collection of other counters. The other â€œmeasurementâ€□ counters will only count during the â€œhighâ€□ 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 â€œmeasurementâ€□ counters
3. Call ***CTR_8254SelectGate()*** to tell the board which counter is generating that gate
4. Call ***CTR_8254ReadLatched()*** periodically to read the latched count values from all the â€œmeasurementâ€□ 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 counterâ€™s output as your gating signal. This last counter in the chain should be reported as the â€œgate sourceâ€ 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 â€œ1324 / 0.05", or 26.48KHz.

Applies To

Counter timers; Linux, Windows

Parameters

DeviceIndex - a standard device index

BlockIndex - number indicating which 8254 you wish to output a output a frequency

pHz - pointer to a double precision IEEE floating point number containing the desired output frequency (in Hertz). This variable is set by the driver to the actual frequency that will be output, as limited by the deviceâ€™s capabilities.

Return Value

A standard result code

ADC Functions

unsigned long ADC_ADMode(unsigned long DeviceIndex, unsigned char TriggerMode, unsigned char CalMode)

Sets the A/D trigger and calibration modes. The USB-AI12-16E does not support calibration, and will use CalMode 0.

Linux: there is a parallel set of convenience functions for configuring the A/D. Refer to ***AIOUSB_SetTriggerMode()*** and ***AIOUSB_SetCalMode()*** for more information.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

TriggerMode - byte indicating which A/D trigger source to use, see the manual for details. Also sets the clock source for counter 0.

Linux: You may use a bitwise OR of the ***AD_TRIGGER_**** values.

CalMode - byte indicating which A/D source to use. May be one of:

00 hex - actual inputs

01 hex - calibration ground reference

03 hex - calibration high reference

Other values will cause the function to fail and return an error result.

Linux: You may use one of the ***AD_CAL_MODE_**** values.

Return Value

A standard result code

---

unsigned long ADC_BulkAcquire(unsigned long DeviceIndex, unsigned long BufSize, void *pBuf)

Starts a large A/D acquisition operation and returns immediately. A result code of ***AIOUSB_SUCCESS / ERROR_SUCCESS*** indicates that A/D data is being acquired in the background, and the buffer should not be deallocated or moved. Use ***ADC_BulkPoll()*** to query the status of this background operation.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

BufSize - the size, in bytes, of the buffer to receive the data

pBuf - a pointer to the beginning of the buffer to receive the data

Return Value

A standard result code

---

unsigned long ADC_BulkPoll(unsigned long DeviceIndex, unsigned long *BytesLeft)

Queries the status of a background A/D acquisition operation initiated by a call to ***ADC_BulkAcquire()***, returning the number of bytes yet to be transferred.

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 remains to be taken, the first 256 MB of data is available.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard 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

Return Value

A standard result code

---

unsigned long ADC_GetChannelV(unsigned long DeviceIndex, unsigned long ChannelIndex, double *pBuf)

Reads a voltage from a single channel, averaging any oversamples, if so configured. This function is easy to use, but relatively slow and often canâ€™t achieve more than 100Hz sampling rates.

Linux: the channel scan range is temporarily changed to include *only* ***ChannelIndex***. This feature ensures that ***ADC_GetChannelV()*** will always return valid data. It also improves performance by reading just the channel requested instead of the entire channel scan range *(see **ADC_SetScanLimits()**)*.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

ChannelIndex - number indicating which channel to read

pBuf - a pointer to a double precision IEEE floating point number which will receive the value read (in Volts)

Return Value

A standard result code

---

unsigned long ADC_GetConfig(unsigned long DeviceIndex, unsigned char *pConfigBuf, unsigned long *ConfigBufSize)

Gets the current A/D configuration from the device. The format of the configuration data returned is described in ***ADC_SetConfig()***.

Linux: there is a parallel set of convenience functions for configuring the A/D. Refer to ***ADConfigBlock*** for more information.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

pConfigBuf - a pointer to the first of an array in which configuration bytes will be returned

ConfigBufSize - a pointer to a variable holding the number of configuration bytes to read. Will be set to the number of configuration bytes read.

Return Value

A standard result code

---

unsigned long ADC_GetScan(unsigned long DeviceIndex, unsigned short *pBuf)

Takes one scan of A/D data, from ***start-channel*** to ***end-channel***, averaging any oversamples for each channel, if so configured. The input array ***pBuf*** must contain one entry per A/D channel on the board, even though only entries ***start-channel*** through ***end-channel*** are altered. This function is easy to use, but relatively slow and often canâ€™t achieve more than several hundred Hertz sampling rates.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

pBuf - a pointer to the first of an array of 16-bit integers which will each receive the value from one channel (in A/D counts)

Return Value

A standard result code

---

unsigned long ADC_GetScanV(unsigned long DeviceIndex, double *pBuf)

Takes one scan of A/D data, from ***start-channel*** to ***end-channel***, averaging any oversamples for each channel, if so configured. The input array ***pBuf*** must contain one entry per A/D channel on the board, even though only entries ***start-channel*** through ***end-channel*** are altered. This function is easy to use, but relatively slow and often canâ€™t achieve more than several hundred Hertz sampling rates.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

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 (in Volts)

Return Value

A standard result code

---

unsigned long ADC_Initialize(unsigned long DeviceIndex, unsigned char *pConfigBuf, unsigned long *ConfigBufSize, const char *CalFileName)

Sets the A/D configuration in the device and/or loads a calibration table into the device. If ***pConfigBuf*** or ***ConfigBufSize*** are ***NULL (0)***, the A/D will not be configured. If ***CalFileName*** is ***NULL (0)***, the calibration table will not be loaded. If all three parameters are ***NULL (0)***, no action will be performed.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

pConfigBuf - a pointer to the first of an array of configuration bytes *(see **ADC_SetConfig()**)*

ConfigBufSize - a pointer to a variable holding the number of configuration bytes to write *(see **ADC_SetConfig()**)*

CalFileName - the name of a calibration file to load *(see **ADC_SetCal()**)*

Return Value

A standard result code

---

unsigned long ADC_QueryCal(unsigned long DeviceIndex)

Queries the device to determine if it supports A/D calibration. A result code of ***AIOUSB_SUCCESS / ERROR_SUCCESS*** indicates that the device does support A/D calibration.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

Return Value

A standard result code

---

unsigned long ADC_Range1(unsigned long DeviceIndex, unsigned long ADChannel, unsigned char GainCode, unsigned long bSingleEnded)

Sets the range and "ended" mode for an A/D channel.

Linux: there is a parallel set of convenience functions for configuring the A/D. Refer to ***AIOUSB_SetGainCode()*** and ***AIOUSB_SetDifferentialMode()*** for more information.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

ADChannel - number from 0-15 indicating an A/D channel on the device

GainCode - the gain code *(see **ADC_SetConfig()**)*

Linux: You may use one of the ***AD_GAIN_CODE_*** * values.

bSingleEnded - for channels 0-7, use ***AIOUSB_TRUE, TRUE*** for single-ended mode, or use ***AIOUSB_FALSE, FALSE*** to pair it with the respective channel 8-15 in differential mode. For channels 8-15, use ***AIOUSB_FALSE, FALSE***.

Return Value

A standard result code

---

unsigned long ADC_RangeAll(unsigned long DeviceIndex, unsigned char *pGainCodes, unsigned long bSingleEnded)

Sets the range and "ended" mode for all A/D channels.

Linux: there is a parallel set of convenience functions for configuring the A/D. Refer to ***AIOUSB_SetAllGainCodeAndDiffMode()*** for more information.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

pGainCodes - a pointer to the first of an array of 16 bytes, each of which contains a gain code. This does not include single-ended/differential configuration. To configure single-ended/differential mode on a per-channel basis, use ***ADC_Range1()*** or ***ADC_SetConfig()***.

Linux: You may use one of the ***AD_GAIN_CODE_*** * values for each of the gain codes.

bSingleEnded - Use ***AIOUSB_TRUE, TRUE*** for 16-channel single-ended mode, or use ***AIOUSB_FALSE, FALSE*** for 8-channel differential mode.

Return Value

A standard result code

---

unsigned long ADC_SetCal(unsigned long DeviceIndex, const char *CalFileName)

Loads a calibration table into the A/D. The calibration table can either be loaded from a file or generated automaticallly, depending on the file name passed to ***ADC_SetCal()***.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

CalFileName - this can be a full path to a calibration file or simply a file name in the current directory. To generate a calibration table automatically, one of these special command strings (case-sensitive) may be used instead of an actual file name:

":AUTO:" - automatically calibrates the A/D using internal ground and reference measurements and creates an appropriate calibration table, which is loaded into the A/D

":NONE:" - generates a default, uncalibrated table, which is loaded into the A/D

Return Value

A standard result code

---

unsigned long ADC_SetConfig(unsigned long DeviceIndex, unsigned char *pConfigBuf, unsigned long *ConfigBufSize)

Sets the A/D configuration in the device.

Linux: there is a parallel set of convenience functions for configuring the A/D. Refer to ***ADConfigBlock*** for more information.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

pConfigBuf - a pointer to the first of an array of configuration bytes *(described below)*

ConfigBufSize - a pointer to a variable holding the number of configuration bytes to write. Will be set to the number of configuration bytes written.

Embedded Solutions

Configuration bytes for analog input boards (such as USB-AI16-16 family)							
Byte 00	Bytes 01-0E	0F	10	11	12	13	14
Channel 0 Gain Code	Channels 1-14 Gain Code	Channel 15 Gain Code	Calibration Mode	Trigger & Counter Clock	Scan Start & End Channel	Oversample	MUX Scan Start & End Channel

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

Channel 0-15 Gain Codes (bytes 00-0F)								
Gain Code	00	01	02	03	04	05	06	07
Range	0-10V	Â±10V	0-5V	Â±5V	0-2V	Â±2V	0-1V	Â±1V

Add 08 to the gain code for channel 0-7 to pair it with the respective channel 8-15 in differential mode.

Calibration Mode (byte 10)			
Cal. Mode	00	01	03
Effect	Acquire Normal Data	Acquire Cal. Ground (0.0114V @ 0-10V)	Acquire Cal. Reference (9.9V @ 0-10V)

Trigger & Counter Clock (byte 11)								
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.

ADL Embedded Solutions Inc., 4411 Morena Blvd., Suite 101, San Diego, CA 92117-4345
P. +1 858 490-0597 **F.** +1 858 490-0599
e-mail: sales@adl-usa.com; **web**: www.adl-usa.com

- 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.

Scan Start & End Channel (byte 12)
The start channel (0-F) is specified in bits 0-3, and the end channel (0-F) is specified in bits 4-7. If the end channel is less than the start channel, then the board's behavior is unspecified.
Oversample (byte 13)
A number indicating how many *extra* samples should be acquired from each channel before moving on to the next channel. In a noisy environment, the samples can be averaged together by software to effectively reduce noise.
MUX Scan Start & End Channel (byte 14)
In devices with a MUX (i.e. more than 16 channels), the low 4-bits of the start and end channels are stored in byte 12, described above, and the high 4-bits of the start and end channels are stored here in byte 14. The high 4-bits of the start channel are specified in bits 0-3, and the high 4-bits of the end channel are specified in bits 4-7.

Return Value

A standard result code

---

unsigned long ADC_SetOversample(unsigned long DeviceIndex, unsigned char Oversample)

Sets the number of over-samples for all A/D channels.

Linux: there is a parallel set of convenience functions for configuring the A/D. Refer to ***AIOUSB_SetOversample()*** for more information.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

Oversample - the number of extra samples to take from each channel in a scan *(see **ADC_SetConfig()**)*

Return Value

A standard result code

---

unsigned long ADC_SetScanLimits(unsigned long DeviceIndex, unsigned long StartChannel, unsigned long EndChannel)

Sets the start and end channels for A/D scans.

Linux: there is a parallel set of convenience functions for configuring the A/D. Refer to ***AIOUSB_SetScanRange()*** for more information.

Applies To

Analog inputs; Linux, Windows

Parameters

DeviceIndex - a standard 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 *(must be greater than or equal to **StartChannel**)*

Return Value

A standard result code

Extended ADC Functions

unsigned long AIOUSB_ADC_ExternalCal(unsigned long DeviceIndex, const double points[], int numPoints, unsigned short returnCalTable[], const char *saveFileName)

Permits the A/D to be calibrated using an external voltage source. The proper way to use this function is to configure the A/D with a default calibration table (such as

by calling ***ADC_SetCal()*** or ***AIOUSB_ADC_InternalCal()***). Then inject a series of voltages into one of the A/D input channels, recording the count values reported by the A/D (by calling ***ADC_GetScan()***). It's also a good idea to enable oversampling while recording these values in order to obtain the most stable readings. Alternatively, since ***points*** is an array of ***double*** values, you can obtain individual A/D count measurements and average them yourself, producing a ***double*** average, and put that value into the ***points*** array.

The ***points*** array consists of *voltage-count* pairs; ***points[0]*** is the first input voltage; ***points[1]*** is the corresponding count value measured by the A/D; ***points[2]*** and ***points[3]*** contain the second pair of *voltage-count* values; and so on. You can provide any number of pairs, although more than a few dozen is probably overkill, not to mention would take a lot of effort to acquire.

This calibration procedure uses the current gain A/D setting for channel 0, so it must be the same as that used to collect the measured A/D counts. It's recommended that all the channels be set to the same gain, the one that will be used during normal operation. The calibration is gain dependent, so switching the gain after calibrating may introduce slight offset or gain changes. So for best results, the A/D should be calibrated on the same gain setting that will be used during normal operation. You can create any number of calibration tables. If your application needs to switch between ranges, you may wish to create a separate calibration table for each range your application will use. Then when switching to a different range, the application can load the appropriate calibration table.

Although calibrating in this manner does take some effort, it produces the best results, eliminating all sources of error from the input pins onward. Furthermore, the calibration table can be saved to a file and reloaded into the A/D, ensuring consistency.

Applies To

Analog inputs; Linux

Parameters

DeviceIndex - a standard device index

points - array of *voltage-count* pairs to calibrate the A/D with

numPoints - number of *voltage-count* pairs in ***points*** (minimum of 2)

returnCalTable - pointer to an array of 65,536 16-bit integers in which the generated calibration table will be returned (may be subsequently loaded into the A/D using ***AIOUSB_ADC_SetCalTable()***); if this parameter is 0, nothing is returned

saveFileName - the name of a file in which the generated calibration table will be saved (may be subsequently loaded into the A/D using ***ADC_SetCal()*** or ***AIOUSB_ADC_LoadCalTable()***); if this parameter is 0, the calibration table is not saved to a file

Return Value

A standard result code

---

unsigned long AIOUSB_ADC_InternalCal(unsigned long DeviceIndex, AIOUSB_BOOL autoCal, unsigned short returnCalTable[], const char *saveFileName)

Calibrates the A/D in the same manner as ***ADC_SetCal()***, generating either a default table or using the internal voltage references to generate a calibration table. The advantage of ***AIOUSB_ADC_InternalCal()*** over ***ADC_SetCal()*** is that ***AIOUSB_ADC_InternalCal()*** can return the calibration table or save it in a file.

Applies To

Analog inputs; Linux

Parameters

DeviceIndex - a standard device index

autoCal - ***AIOUSB_TRUE*** uses the internal voltage references to automatically calibrate the A/D; ***AIOUSB_FALSE*** generates a default (uncalibrated) table

returnCalTable - pointer to an array of 65,536 16-bit integers in which the generated calibration table will be returned (may be subsequently loaded into the A/D using ***AIOUSB_ADC_SetCalTable()***); if this parameter is 0, nothing is returned

saveFileName - the name of a file in which the generated calibration table will be saved (may be subsequently loaded into the A/D using ***ADC_SetCal()***or ***AIOUSB_ADC_LoadCalTable()***); if this parameter is 0, the calibration table is not saved to a file

Return Value

A standard result code

---

unsigned long AIOUSB_ADC_LoadCalTable(unsigned long DeviceIndex, const char *fileName)

Loads a calibration table from a file into the A/D in the same manner as ***ADC_SetCal()***.

Applies To

Analog inputs; Linux

Parameters

DeviceIndex - a standard device index

fileName - the name of a file containing the calibration table (such as created by ***AIOUSB_ADC_InternalCal()*** or ***AIOUSB_ADC_ExternalCal()***).

Return Value

A standard result code

---

unsigned long AIOUSB_ADC_SetCalTable(unsigned long DeviceIndex, const unsigned short calTable[])

Sets the calibration table in the A/D to the contents of ***calTable***.

Applies To

Analog inputs; Linux

Parameters

DeviceIndex - a standard device index

calTable - pointer to an array of 65,536 16-bit integers containing the calibration table (such as created by ***AIOUSB_ADC_InternalCal()*** or ***AIOUSB_ADC_ExternalCal()***).

Return Value

A standard result code

---

double AIOUSB_CountsToVolts(unsigned long DeviceIndex, unsigned channel, unsigned short counts)

Converts a single A/D count value to volts, based on the deviceâ€™s current gain setting for the specified channel. Be careful to ensure that the count value was actually obtained from the specified channel and that the gain hasnâ€™t changed since the count value was obtained.

Applies To

Analog inputs; Linux

Parameters

DeviceIndex - a standard device index

channel - the channel number to use for converting counts to volts

counts - the count value to convert to volts

Return Value

Volts

---

unsigned AIOUSB_GetCalMode(const ADConfigBlock *config)

Returns the current calibration mode. (Must call ***ADC_GetConfig()*** first to get configuration from device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

Return Value

Current calibration mode (one of the ***AD_CAL_MODE_**** values)

---

unsigned AIOUSB_GetEndChannel(const ADConfigBlock *config)

Returns the current end channel for A/D scans. (Must call ***ADC_GetConfig()*** first to get configuration from device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

Return Value

Current end channel for A/D scans

---

unsigned AIOUSB_GetGainCode(const ADConfigBlock *config, unsigned channel)

Returns the current gain code for ***channel***. (Must call ***ADC_GetConfig()*** first to get configuration from device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

channel - the channel for which to obtain the current gain code

Return Value

Current gain code for ***channel*** (one of the ***AD_GAIN_CODE_**** values)

---

unsigned AIOUSB_GetOversample(const ADConfigBlock *config)

Returns the current number of over-samples. (Must call ***ADC_GetConfig()*** first to get configuration from device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

Return Value

Current number of over-samples (0-255)

---

unsigned AIOUSB_GetStartChannel(const ADConfigBlock *config)

Returns the current start channel for A/D scans. (Must call ***ADC_GetConfig()*** first to get configuration from device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

Return Value

Current start channel for A/D scans

---

unsigned AIOUSB_GetTriggerMode(const ADConfigBlock *config)

Returns the current trigger mode. (Must call ***ADC_GetConfig()*** first to get configuration from device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

Return Value

Current trigger mode (a bitwise OR of the ***AD_TRIGGER_**** values)

---

void AIOUSB_InitConfigBlock(ADConfigBlock *config, unsigned long DeviceIndex, AIOUSB_BOOL defaults)

Initializes an ***ADConfigBlock*** instance for the specified device index. **Must** be called before passing ***config*** to any other A/D configuration block functions.

Applies To

Analog inputs; Linux

Parameters

config - pointer to an uninitialized ***ADConfigBlock*** structure

DeviceIndex - a standard device index

defaults - ***AIOUSB_TRUE*** will fully initialize the block to proper default values, suitable for passing to ***ADC_SetConfig()***; ***AIOUSB_FALSE*** performs a minimal initialization, preparing the configuration block for passing to ***ADC_GetConfig()***

AIOUSB_BOOL AIOUSB_IsDifferentialMode(const ADConfigBlock *config, unsigned channel)

Tells if ***channel*** is configured for single-ended or differential mode. (Must call ***ADC_GetConfig()*** first to get configuration from device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

channel - the channel for which to obtain the current differential mode

Return Value

AIOUSB_TRUE indicates differential mode; ***AIOUSB_FALSE*** indicates single-ended mode

---

AIOUSB_BOOL AIOUSB_IsDiscardFirstSample(unsigned long DeviceIndex)

Tells if the ADC functions will discard the first A/D sample taken. Discarding the first sample may be useful in cases in which voltage "residue" from reading a different channel affects the channel currently being read.

Applies To

Analog inputs; Linux

Parameters

DeviceIndex - a standard device index

Return Value

AIOUSB_FALSE indicates that no samples will be discarded;
AIOUSB_TRUE indicates that the first sample will be discarded

---

unsigned long AIOUSB_MultipleCountsToVolts(unsigned long DeviceIndex, unsigned startChannel, unsigned endChannel, const unsigned short counts[], double volts[])

Converts an array of A/D count values to volts, based on the deviceâ€™s current gain settings for the specified channels. Be careful to ensure that the count values were actually obtained from the specified channels and that the gains havenâ€™t changed since the count values were obtained. The arrays must be large enough to accommodate all the channels the device supports, even though only the count values for ***startChannel*** through ***endChannel*** are converted from counts to volts. An array of count values returned by ***ADC_GetScan()*** may be passed to ***AIOUSB_MultipleCountsToVolts()*** directly.

Applies To

Analog inputs; Linux

Parameters

DeviceIndex - a standard device index

startChannel - start channel to convert

endChannel - end channel to convert (must be greater than or equal to ***startChannel***)

counts - array of A/D counts, such as obtained from ***ADC_GetScan()***

volts - array in which the converted volts will be returned

Return Value

A standard result code

---

unsigned long AIOUSB_MultipleVoltsToCounts(unsigned long DeviceIndex, unsigned startChannel, unsigned endChannel, const double volts[], unsigned short counts[])

Converts an array of voltage values to A/D counts, based on the deviceâ€™s current gain settings for the specified channels. Be careful to ensure that the voltage values were actually obtained from the specified channels and that the gains havenâ€™t changed since the voltage values were obtained. The arrays must be large enough to accommodate all the channels the device supports, even though only the voltage values for ***startChannel*** through ***endChannel*** are converted from volts to counts. An array of voltage values returned by ***ADC_GetScanV()*** may be passed to ***AIOUSB_MultipleVoltsToCounts()*** directly.

Applies To

Analog inputs; Linux

Parameters

DeviceIndex - a standard device index

startChannel - start channel to convert

endChannel - end channel to convert (must be greater than or equal to ***startChannel***)

volts - array of voltage values, such as obtained from ***ADC_GetScanV()***

counts - array in which the converted A/D counts will be returned

Return Value

A standard result code

---

void AIOUSB_SetAllGainCodeAndDiffMode(ADConfigBlock *config, unsigned gainCode, AIOUSB_BOOL differentialMode)

Sets all the A/D channels to the same gain code and differential mode. (Must call ***ADC_SetConfig()*** to send configuration to device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

gainCode - one of the ***AD_GAIN_CODE_**** values

differentialMode - ***AIOUSB_TRUE*** selects differential mode; ***AIOUSB_FALSE*** selects single-ended mode

---

void AIOUSB_SetCalMode(ADConfigBlock *config, unsigned calMode)

Sets the A/D calibration mode. (Must call ***ADC_SetConfig()*** to send configuration to device.)

Linux: If ground or reference mode is selected, only one A/D sample may be taken at a time. That means, one channel and no oversampling. Attempting to read more than one channel or use an oversample setting of more than zero will result in a timeout error because the device will not send more than one sample. In order to protect users from accidentally falling into this trap, function ***ADC_GetScan()*** automatically and temporarily corrects the scan parameters, restoring them when it completes.

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

calMode - the calibration mode. May be one of:

AD_CAL_MODE_NORMAL (0) - normal measurement

AD_CAL_MODE_GROUND (1) - measure ground

AD_CAL_MODE_REFERENCE (3) - measure reference

---

void AIOUSB_SetDifferentialMode(ADConfigBlock *config, unsigned channel, AIOUSB_BOOL differentialMode)

Sets a single A/D channel to differential or single-ended mode. (Must call *ADC_SetConfig()* to send configuration to device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

channel - the channel for which to set differential or single-ended mode

differentialMode - ***AIOUSB_TRUE*** selects differential mode; ***AIOUSB_FALSE*** selects single-ended mode

---

unsigned long AIOUSB_SetDiscardFirstSample(unsigned long DeviceIndex, AIOUSB_BOOL discard)

Specifies whether the ADC functions will discard the first A/D sample taken. Discarding the first sample may be useful in cases in which voltage "residue" from reading a different channel affects the channel currently being read. (Must call *ADC_SetConfig()* to send configuration to device.)

Applies To

Analog inputs; Linux

Parameters

DeviceIndex - a standard device index

discard - ***AIOUSB_FALSE*** indicates that no samples will be discarded; ***AIOUSB_TRUE*** indicates that the first sample will be discarded

Return Value

A standard result code

---

void AIOUSB_SetGainCode(ADConfigBlock *config, unsigned channel, unsigned gainCode)

Sets the gain code for a single A/D channel. (Must call ***ADC_SetConfig()*** to send configuration to device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

channel - the channel for which to set the gain code

gainCode - the gain code (voltage range) for the channel. May be one of:

AD_GAIN_CODE_0_10V (0) - 0-10V

AD_GAIN_CODE_10V (1) - +/-10V

AD_GAIN_CODE_0_5V (2) - 0-5V

AD_GAIN_CODE_5V (3) - +/-5V

AD_GAIN_CODE_0_2V (4) - 0-2V

AD_GAIN_CODE_2V (5) - +/-2V

AD_GAIN_CODE_0_1V (6) - 0-1V

AD_GAIN_CODE_1V (7) - +/-1V

---

void AIOUSB_SetOversample(ADConfigBlock *config, unsigned overSample)

Sets the number of over-samples for all A/D channels. (Must call ***ADC_SetConfig()*** to send configuration to device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

overSample - number of over-samples (0-255)

---

void AIOUSB_SetScanRange(ADConfigBlock *config, unsigned startChannel, unsigned endChannel)

Sets the start and end channels for A/D scans. (Must call ***ADC_SetConfig()*** to send configuration to device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

startChannel - start channel

endChannel - end channel *(must be greater than or equal to **startChannel**)*

---

void AIOUSB_SetTriggerMode(ADConfigBlock *config, unsigned triggerMode)

Sets the trigger mode. (Must call ***ADC_SetConfig()*** to send configuration to device.)

Applies To

Analog inputs; Linux

Parameters

config - pointer to a properly initialized ***ADConfigBlock*** structure

triggerMode - a bitwise OR of these flags:

AD_TRIGGER_CTR0_EXT (0x10) - if set, counter 0 is externally triggered. Otherwise, counter 0 is triggered by the onboard 10MHz clock.

AD_TRIGGER_FALLING_EDGE (0x08) - if set, the A/D is triggered by the falling edge of its trigger source. Otherwise, the A/D is triggered by the rising edge of its trigger source.

AD_TRIGGER_SCAN (0x04) - if 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.

AD_TRIGGER_EXTERNAL (0x02) - if set, the external A/D trigger pin is an A/D trigger source. Otherwise, it's ignored.

AD_TRIGGER_TIMER (0x01) - if set, counter 2 is an A/D trigger source. Otherwise, it's ignored.

---

unsigned short AIOUSB_VoltsToCounts(unsigned long DeviceIndex, unsigned channel, double volts)

Converts a single voltage value to A/D counts, based on the deviceâ€™s current gain setting for the specified channel. Be careful to ensure that the voltage value was actually obtained from the specified channel and that the gain hasnâ€™t changed since the voltage value was obtained.

Applies To

Analog inputs; Linux

Parameters

DeviceIndex - a standard device index

channel - the channel number to use for converting volts to counts

volts - the voltage to convert to A/D counts

Return Value

A/D counts

DAC Functions

unsigned long DACDirect(unsigned long DeviceIndex, unsigned short Channel, unsigned short Value)

Writes a count value to a D/A channel.

Applies To

Analog outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

Channel - the channel number of the D/A you wish to set. The number of channels varies from model to model.

Value - the D/A count value to output. The number of bits of resolution for the D/A outputs varies from model to model, however it's usually 12- or 16-bits. Moreover, some of the 12-bit models actually accept a 16-bit value and simply truncate the least significant 4 bits. Consult the manual for the specific device to determine the range of D/A values the device will accept. In general, 12-bit devices accept a count range of 0-FFFh, and 16-bit devices accept a count range of 0-FFFFh.

Return Value

A standard result code

---

unsigned long DACMultiDirect(unsigned long DeviceIndex, unsigned short *pDACData, unsigned long DACDataCount)

Writes a block of count values to one or more D/A channels.

Applies To

Analog outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

pDACData - a pointer to an array of 16-bit integers representing channel/count *pairs*. The first integer of each pair is the D/A channel number and the second integer is the D/A count value to output to the specified channel. Refer to ***DACDirect()*** for an explanation of the channel addressing and count values.

DACDataCount - number indicating how many channel/count *pairs* are in the array referenced by ***pDACData***

Return Value

A standard result code

---

unsigned long DACOutputClose(unsigned long DeviceIndex, unsigned long bWait)

Ends and closes a DAC streaming process. *Deprecated:* ***DACOutputCloseNoEnd()*** *is preferred.*

Linux: Not yet implemented.

Applies To

USB-DA12-8A; Linux, Window

Parameters

DeviceIndex - a standard device index

bWait - reserved for future expansion; currently, this function always waits for the streaming process to complete before returning

Return Value

A standard result code

---

unsigned long DACOutputCloseNoEnd(unsigned long DeviceIndex, unsigned long bWait)

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

Linux: Not yet implemented.

Applies To

USB-DA12-8A; Linux, Windows

Parameters

DeviceIndex - a standard device index

bWait - reserved for future expansion; currently, this function always waits for the streaming process to complete before returning

Return Value

A standard result code

---

unsigned long DACOutputFrame(unsigned long DeviceIndex, unsigned long FramePoints, unsigned short *FrameData)

Writes a group of points (a â€œframeâ€□) into the DAC stream. *Deprecated:* ***DACOutputFrameRaw()*** *is preferred.*

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 driverâ€™s internal buffer is full, the function will return â€œERROR_NOT_READYâ€□ (equal to 21 decimal); try again in a moment, as the driverâ€™s buffer should drain some as soon as thereâ€™s room in the larger hardware buffer and available time on the USB bus.

Linux: Not yet implemented.

Applies To

USB-DA12-8A; Linux, Windows

Parameters

DeviceIndex - a standard device index

FramePoints - the number of points (16-bit integers) in the frame

FrameData - a pointer to the first of an array of 16-bit D/A count values to output

Return Value

A standard result code

---

unsigned long DACOutputFrameRaw(unsigned long DeviceIndex, unsigned long FramePoints, unsigned short *FrameData)

Writes a group of points (a â€œframeâ€□) into the DAC stream. 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.

Linux: Not yet implemented.

Applies To

USB-DA12-8A; Linux, Windows

Parameters

DeviceIndex - a standard device index

FramePoints - the number of points (16-bit integers) in the frame

FrameData - a pointer to the first of an array of 16-bit D/A count values to output. The DAC count determines the number of samples, even though you can place EOD bits (*see below*) as you wish.

FrameData Bit Fields															
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 stream 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.

Return Value

A standard result code

---

unsigned long DACOutputOpen(unsigned long DeviceIndex, double *pClockHz)

Begins a DAC streaming process. The stream is divided into â€œpointsâ€□. 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.

Linux: Not yet implemented.

Applies To

USB-DA12-8A; Linux, Windows

Parameters

DeviceIndex - a standard 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 deviceâ€™s capabilities.

Return Value

A standard result code

unsigned long DACOutputSetCount(unsigned long DeviceIndex, unsigned long NewCount)

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.

Linux: Not yet implemented.

Applies To

USB-DA12-8A; Linux, Windows

Parameters

DeviceIndex - a standard device index

NewCount - number from 1-8 indicating the number of DACs in future points

Return Value

A standard result code

---

unsigned long DACOutputSetInterlock(unsigned long DeviceIndex, unsigned long bInterlock)

Enables or disables 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).

Linux: Not yet implemented.

Applies To

USB-DA12-8A; Linux, Windows

Parameters

DeviceIndex - a standard device index

bInterlock - ***AIOUSB_TRUE, TRUE*** enables interlock; ***AIOUSB_FALSE, FALSE*** disables interlock

Return Value

A standard result code

---

unsigned long DACOutputStart(unsigned long DeviceIndex)

"Manually" starts a DAC streaming process. Normally, DAC streaming will be started automatically by streaming 1Â¼ 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.

Linux: Not yet implemented.

Applies To

USB-DA12-8A; Linux, Windows

Parameters

DeviceIndex - a standard device index

Return Value

A standard result code

---

unsigned long DACSetBoardRange(unsigned long DeviceIndex, unsigned long RangeCode)

Sets the voltage range of the D/A outputs.

Applies To

Analog outputs; Linux, Windows

Parameters

DeviceIndex - a standard device index

RangeCode - the voltage range to select. May be one of:

DAC_RANGE_0_5V (0) - 0-5V

DAC_RANGE_5V (1) - +/-5V

DAC_RANGE_0_10V (2) - 0-10V

DAC_RANGE_10V (3) - +/-10V

Linux: Use the named constants listed above.

Windows: Use the numeric values shown in parenthesis.

Return Value

A standard result code

Function Parameters

struct ADConfigBlock (Linux)

The ***ADConfigBlock*** structure is initialized by a call to ***AIOUSB_InitConfigBlock()*** and defined in ***aiousb.h*** as:

```
struct ADConfigBlock {
  void *device;
  unsigned long size;
  unsigned char registers[ AD_MAX_CONFIG_REGISTERS ];
}; // struct ADConfigBlock
```

This structure is intended to be somewhat opaque, its main purpose being to shield users from having to fill in the bytes and bit fields of an A/D configuration block which is used in calls to ***ADC_GetConfig()*** and ***ADC_SetConfig()***.

In addition to simplifying the getting and setting of the configuration, using ***ADConfigBlock*** and its associated functions permits you to change several configuration settings and then send them all to the device with a single call to ***ADC_SetConfig()***. The other API functions that manipulate individual configuration settings, such as ***ADC_Range1()***, send the modified configuration to the device immediately. There is nothing wrong with sending the configuration to the device after each setting change, but using ***ADConfigBlock*** and its associated functions gives users the option of deferring sending the configuration until it's fully set up. Of course, ***ADC_SetConfig()*** also lets users send the entire configuration in one step, but it requires them to manipulate the bytes and bit fields of the A/D configuration block.

The functions that manipulate ***ADConfigBlock*** are â€œobject-orientedâ€□ in the sense that the proper way to use them is to create an ***ADConfigBlock*** instance and pass it to all these functions. The ***ADConfigBlock*** instance should be initialized immediately after itâ€™s declared using ***AIOUSB_InitConfigBlock()***, which is equivalent to a â€œconstructorâ€□ in C++. ***AIOUSB_InitConfigBlock()*** makes sure that the configuration block is initialized properly for the device with which it will be used.

The following functions are the "methods" that can operate on ***ADConfigBlock***:

AIOUSB_GetCalMode()
AIOUSB_GetEndChannel()
AIOUSB_GetGainCode()
AIOUSB_GetOversample()
AIOUSB_GetStartChannel()
AIOUSB_GetTriggerMode()
AIOUSB_InitConfigBlock()
AIOUSB_IsDifferentialMode()
AIOUSB_SetAllGainCodeAndDiffMode()
AIOUSB_SetCalMode()
AIOUSB_SetDifferentialMode()
AIOUSB_SetGainCode()
AIOUSB_SetOversample()
AIOUSB_SetScanRange()
AIOUSB_SetTriggerMode()

Below is an example of the proper way to use ***ADConfigBlock***.

```
/*
 * procedure for setting A/D configuration
 */

// create "instance"
ADConfigBlock configBlock;

// call "constructor"
AIOUSB_InitConfigBlock( &configBlock, DeviceIndex, AIOUSB_TRUE );

//  ... set properties ...

// send configuration block to device
ADC_SetConfig( DeviceIndex, configBlock.registers, &configBlock.size );

/*
 * procedure for reading current A/D configuration
 */

// create "instance"
ADConfigBlock configBlock;

// call "constructor"
AIOUSB_InitConfigBlock( &configBlock, DeviceIndex, AIOUSB_FALSE );

// read configuration block from device
ADC_GetConfig( DeviceIndex, configBlock.registers, &configBlock.size );

//  ... use properties ...
```

---

unsigned long DeviceIndex

A "handle" to a device on the USB bus. Each device on the USB bus is assigned an index number ranging from 0-31. This index number must then be passed to most of the API functions in order to control a specific device.

For convenience when writing quick test programs and such, these two special constants may be used instead of an actual device index:

diFirst (0xFFFFFFFEul) - uses the first ADL device found on the bus

diOnly (0xFFFFFFFDul) - uses the only ADL device found on the bus, meaning that there must be only one ADL device on the bus

---

struct DeviceProperties (Linux)

The ***DeviceProperties*** structure is filled in by a call to ***AIOUSB_GetDeviceProperties()*** and defined in ***aiousb.h*** as:

```
struct DeviceProperties {
  const char *Name;          // null-terminated device name or 0
  __uint64_t SerialNumber;   // 64-bit serial number or 0
  unsigned ProductID;        // 16-bit product ID
  unsigned DIOPorts;         // number of digital I/O ports (bytes)
  unsigned Counters;         // number of 8254 counter blocks
  unsigned Tristates;        // number of tristates
  long RootClock;            // base clock frequency
  unsigned DACChannels;      // number of D/A channels
  unsigned ADCChannels;      // number of A/D channels
  unsigned ADCMUXChannels;   // number of MUXed A/D channels
}; // struct DeviceProperties
```

Constants

This section is a reference for all of the constants used in AIOUSB for Linux which aren't described elsewhere in this document. The values shown in parenthesis next to each constant name are the actual values.

AIOUSB_BOOL, AIOUSB_TRUE, AIOUSB_FALSE (Linux)

In C, the ***AIOUSB_BOOL*** type is declared as an ***enum***. In C++, it's declared as a ***bool***. Other libraries tend to declare the names ***BOOL***, ***TRUE*** and ***FALSE***, and worse, they declare them using *#defines*, which can be surprisingly difficult to work around. So we sidestep such potential conflicts by declaring the same names prefixed with ***AIOUSB_***. It's ugly, but if people want to use the shorter names and they are certain the shorter names won't conflict with anything else, they can define the ***ENABLE_BOOL_TYPE*** macro, which defines the constants ***BOOL***, ***TRUE*** and ***FALSE***.

AIOUSB_TRUE, TRUE (1, true) - boolean 'true' (in C, has the value '1'; in C++, the value 'true')

AIOUSB_FALSE, FALSE (0, false) - boolean 'false' (in C, has the value '0'; in C++, the value 'false')

Windows: use the names ***TRUE*** and ***FALSE***.

Result Codes

Implementation Notes

The AIOUSB function result codes are a bit confusing. The result codes used in the Windows implementation of the API are defined in a system file named ***winerror.h***. These result codes are generic and can apply to many applications. They are all prefixed with ***ERROR_***, the very first one, ***ERROR_SUCCESS***, sounding like an oxymoron. The result codes used in ***libusb*** (Linux), on the other hand, are more appealing: the result code for success is ***LIBUSB_SUCCESS***, while the result codes for errors are ***LIBUSB_ERROR_xxx***. Further complicating matters is that the AIOUSB result codes must be non-negative since all the functions return an unsigned result, whereas the ***libusb*** result codes are negative in the case of errors. Fortunately, both schemes use zero to indicate success. Finally, it's desirable to return the original ***libusb*** result code in cases where a ***libusb*** error causes an AIOUSB API function to fail. So to satisfy all these requirements, we've employed the following scheme:

- AIOUSB result codes in Linux are prefixed with ***AIOUSB_***. The result code for success is ***AIOUSB_SUCCESS***, which has a value of zero. The result codes for errors are ***AIOUSB_ERROR_xxx***, which have positive values, starting with one (1).

- In order to offer users the option of using result codes whose names are similar to those used in the Windows implementation, we define a second set of result codes with names similar to those used in the Windows implementation but which map to the same values as the ***AIOUSB_xxx*** result codes. These alternate result code names can be enabled by defining the macro ***ENABLE_WINDOWS_RESULT_CODES***, which is not defined by default.

- In order to preserve the original ***libusb*** result codes and pass them back from an AIOUSB API function, we translate the ***libusb*** result codes to a format that conforms to the one AIOUSB employs and provide macros for converting the AIOUSB result code back to a ***libusb*** result code. Macro ***LIBUSB_RESULT_TO_AIOUSB_RESULT(code)*** converts a ***libusb*** result code to an AIOUSB result code, and macro ***AIOUSB_RESULT_TO_LIBUSB_RESULT(code)*** does the reverse.

- In the Linux implementation we provide an extended AIOUSB API function named ***AIOUSB_GetResultCodeAsString()*** that returns a string representation of a result code, including those that encapsulate a ***libusb*** result code.

Linux Result Codes

Each of the result codes shown has two names: the "native" Linux name, and a Windows-style name, which can be enabled by defining the macro ***ENABLE_WINDOWS_RESULT_CODES***.

AIOUSB_SUCCESS, ERROR_SUCCESS (0) - The function completed successfully.

AIOUSB_ERROR_DEVICE_NOT_CONNECTED, ERROR_DEVICE_NOT_CONNECTED (1) - The internal handle to the device is not valid. This can occur if AIOUSB has not been properly initialized. On Linux this can occur because ***libusb*** fails to return a valid handle.

AIOUSB_ERROR_DUP_NAME, ERROR_DUP_NAME (2) - More than one device was found on the bus when ***diOnly*** was specified as a device index.

AIOUSB_ERROR_FILE_NOT_FOUND, ERROR_FILE_NOT_FOUND (3) - May mean multiple things. May mean that the attempt to load the A/D calibration table in ***ADC_SetCal()*** failed because the specified file does not exist or cannot be read. May also mean that the attempt to save the A/D calibration table to a file in ***ADC_SetCal()*** failed because the file could not be created. May also mean that no device was found on the bus when ***diFirst*** or ***diOnly*** were specified as a device index.

AIOUSB_ERROR_INVALID_DATA, ERROR_INVALID_DATA (4) - Data retrieved from the device is not valid. May mean that the data is corrupt or that an incorrect amount of data was transferred

AIOUSB_ERROR_INVALID_INDEX, ERROR_INVALID_INDEX (5) - The device index passed to an function is invalid.

AIOUSB_ERROR_INVALID_MUTEX, ERROR_INVALID_MUTEX (6) - An internal mutex is invalid. This usually means the library was unable to initialize the mutex.

AIOUSB_ERROR_INVALID_PARAMETER, ERROR_INVALID_PARAMETER (7) - One or more parameters passed to a function are invalid, such as outside the allowed range or null pointers.

AIOUSB_ERROR_INVALID_THREAD, ERROR_INVALID_THREAD (8) - An internal thread is invalid. This usually means the library was unable to initialize the thread.

AIOUSB_ERROR_NOT_ENOUGH_MEMORY, ERROR_NOT_ENOUGH_MEMORY (9) - A function was unable to allocate sufficient dynamic memory.

AIOUSB_ERROR_NOT_SUPPORTED, ERROR_NOT_SUPPORTED (10) - The function is not supported for the device specified, for example, trying to perform A/D operations on a device that doesn't have an A/D.

AIOUSB_ERROR_OPEN_FAILED, ERROR_OPEN_FAILED (11) - Typically means that a streaming device is already open for streaming and cannot be opened again until it's closed.

AIOUSB_ERROR_LIBUSB (100) - If the result code is greater than or equal to this value, then it is a ***libusb*** result code indicating an error. Use the ***LIBUSB_RESULT_TO_AIOUSB_RESULT(code)*** macro to extract the original ***libusb*** result code, or pass the result code to ***AIOUSB_GetResultCodeAsString()*** to obtain a string representation of it.

Windows Result Codes

Below is a list (in alphabetical order) of all the result codes returned by functions in the Windows implementation of the AIOUSB API. These result codes are defined in a system file named ***winerror.h***. Please consult the relevant Windows documentation for more information.

ERROR_ALREADY_EXISTS
ERROR_BAD_LENGTH
ERROR_BAD_TOKEN_TYPE
ERROR_DEVICE_NOT_CONNECTED
ERROR_DEVICE_REMOVED

ERROR_DUP_NAME
ERROR_FILE_EXISTS
ERROR_FILE_NOT_FOUND
ERROR_GEN_FAILURE
ERROR_HANDLE_EOF
ERROR_INSUFFICIENT_BUFFER
ERROR_INTERNAL_ERROR
ERROR_INVALID_ADDRESS
ERROR_INVALID_DATA
ERROR_INVALID_INDEX
ERROR_INVALID_PARAMETER
ERROR_NOACCESS
ERROR_NOT_READY
ERROR_NOT_SUPPORTED
ERROR_NO_DATA_DETECTED
ERROR_OPEN_FAILED
ERROR_OUTOFMEMORY
ERROR_SEEK
ERROR_SUCCESS
ERROR_TOO_MANY_OPEN_FILES

Linux-Specfic Notes

Language Support

The AIOUSB library for Linux is compiled for C and C++, residing within ***libaiousb.a*** and ***libaiousbcpp.a***, respectively. However, when using C++, all the public functions, variables, enums and so forth are wrapped up in a ***namespace AIOUSB*** declaration to prevent those names from conflicting with anything else. All of the C++ sample programs for Linux contain the statement ***using namespace AIOUSB;*** immediately after including the file ***aiousb.h***. Using the namespace in this manner is the simplest technique, but if you prefer to explicitly refer to public elements within this namespace, simply prefix all such names with ***AIOUSB::***. For example, instead of calling ***GetDevices()***, you would explicitly call ***AIOUSB::GetDevices()***. Similarly, constant names must be prefixed with ***AIOUSB::***. For example, ***AIOUSB_SUCCESS*** would instead be referred to as ***AIOUSB::AIOUSB_SUCCESS***.

Multithreading Support

The Linux implementation of ***AIOUSB*** is thread-safe to the extent that the internal data structures are protected by a mutex. Furthermore, the lower level ***libusb*** module is also thread safe in the same manner. However, neither ***AIOUSB*** nor ***libusb*** prevent multiple threads from communicating with the same device. A software design that permits multiple threads to communicate with the same device must provide its own mutual exclusion mechanism. ***AIOUSB*** for Linux is designed under the premise that one thread will communicate with one device, and optionally, another thread may communicate with a different device, and so on. That form of multithreading is supported by ***AIOUSB*** for Linux.

Also beware that certain functions, such as ***GetDevices()*** make substantial changes to the internal state of the AIOUSB module. It is not safe to call such functions while other threads are actively communicating with devices! While such internal state changes are made in the context of a locked mutex and will probably not cause the software to fail, in all likelihood any threads which are actively communicating with devices will subsequently experience communication failures with their respective devices. You should only call such functions when no threads are actively communicating with devices. After such changes, threads must reestablish a connection to their devices by obtaining a new device index "handle."

Document: $Revision: 1.2 $ $Date: 2009/12/02 16:58:53 $

ADL Embedded Solutions Inc., 4411 Morena Blvd., Suite 101 San Diego, CA 92117-4345
P. +1 858 490-0597 **F.** +1 858 490-0599
e-mail: sales@adl-usa.com; **web**: www.adl-usa.com