API User's Guide

for the

USB I2C/IO
Universal Serial Bus Interface Module

DeVaSys

http://www.devasys.com/

Revision 0.1.04
September 27th, 2002

Please send any comment to support@devasys.com

Table of contents

Introduction

Revision History

Rev. 0.1.04, September 27th, 2002 – Michael DeVault

Added the GetIoConfig function.
Added the I²C section.

Rev. 0.1.03, February 20th, 2001 – Michael DeVault

Fixed errors involving incorrect Port C bit mapping for the ReadIoPorts, WriteIoPorts, and ConfigIoPorts .

Rev. 0.1.02, February 2nd, 2001 – Michael DeVault

Fixed some formatting problems and typographical errors.

Rev. 0.1.01, February 1st, 2001 – Michael DeVault

Added the Introduction section.

Fixed two erroneous references to a constant named INVALID_FILE_HANDLE for OpenDeviceInstance and OpenDeviceBySerialId functions. The correct constant is INVALID_HANDLE_VALUE.

Revised the example code in the Tutorial section.

Rev. 0.1.00, January 25th, 2001 – Michael DeVault

Original Document

Summary

This document provides information to assist engineers using the DeVaSys UsbI2cIo USB interface hardware, as well as the corresponding applications programming interface (API) in their designs.

Scope

This is a technical document and is intended for use by engineers with experience in various aspects of electronic hardware design, the 'C' programming language, and the Windows 98 operating system. Only information specific to the use of the UsbI2cIo hardware and API software is provided.

Tutorial

Devices, device driver instances, and handles

Each USB I²C/Io board you attach to your host P.C. is a device. For each attached device, Windows will create an instance of the device driver in memory. In order to communicate with a device, you must first obtain a handle to that particular device's driver instance. Most of the functions provided in this API require a parameter called "HANDLE hDevInstance" which is a handle to the device driver instance for the particular device with which you wish to communicate.

An example Win32 console application

```
/*
  example.c – a simple application

  Note:
    This example uses implicit linking to the dll functions, and requires the UsbI2cIo.lib file.
    Explicit linking is also supported, and is recommended for most applications.
*/

#include <windows.h>
#include <stdio.h>
#include "UsbI2cIo.h"

void __cdecl main(void) {

      HANDLE hDevInstance = INVALID_HANDLE_VALUE;            // device instance handle

      // attempt to open device UsbI2cIo0 (instance 0 of the device with symbolic name UsbI2cIo)
      hDevInstance = DAPI_OpenDeviceInstance("UsbI2cIo", 0);

      if( hDevInstance != INVALID_HANDLE_VALUE) {
            // success - device instance handle opened
            printf("Opened device UsbI2cIo0\n");

            // configure all I/O ports for output
            DAPI_ConfigIoPorts(hDevInstance, 0x00000000);        // 0 = output, 1 = input
            printf("Configured all I/O ports as outputs\n");

            // clear all bits except A.0 which will be set
            DAPI_WriteIoPorts(hDevInstance, 0x00000001, 0xFFFFFFFF);
            printf("Cleared all port bits to 0, with the exception of A.0 which was set to 1\n");

            // now set bit B.7, and clear bit A.0, leave other bits unmodified
            DAPI_WriteIoPorts(hDevInstance, 0x00008000, 0x00008001);
            printf("Set bit B.7 and cleared bit A.0, left other bits unmodified\n");
      }
      else {
            printf("Failed to open a handle to device UsbI2cIo0\n");
      }


      if( hDevInstance != INVALID_HANDLE_VALUE) {
            // close handle to device instance
            DAPI_CloseDeviceInstance(hDevInstance);
      }
}
```

< Much more information will be added to the tutorial section in future revisions of this document >

API Functions

API Overview

The USB I²C/IO API provides a set of 'C' functions for discovering and communicating with attached USB I²C/IO devices. The API is implemented as a DLL (dynamically linked library) file which must be installed on your P.C. host.

The API provides support for the following tasks:

- Obtaining version information for the device firmware, device driver, and API dll
- Obtaining a list of attached USB I²C/IO devices.
- Opening and closing handles to devices
- Obtaining the Serial Id of a device (unique identifier for each device)
- Configuring, Reading, and Writing the 20 digital I/O ports of a device,
- Generating I²C bus transactions on a devices.
- Obtaining debug information from a device.

Comments

See the Tutorial Section for an example on how to use the API.

API Functions Summary

Version Checking

GetDllVersion	Returns the DLL version information.
GetDriverVersion	Returns the Device Driver version information.
GetFirmwareVersion	Returns the Firmware version for the specified device.

Discovery and Detection

GetDeviceCount	Returns the number of attached devices.
GetDeviceInfo	Generates a list of device information for attached devices.
GetSerialId	Obtains a device's Serial ID string (unique identifier).
DetectDevice	Return true if device handle is valid (device present).

Device Handle

OpenDeviceInstance	Opens a handle to a specified device instance.
CloseDeviceInstance	Closes a device instance handle
OpenDeviceBySerialId	Obtains a handle to a device specified by it Serial ID.

I/O Ports

ConfigIoPorts	Configures the I/O port bits as Inputs or Outputs
GetIoConfig	Reads the current configuration of the I/O pins.
ReadIoPorts	Reads from the I/O pins (both inputs and outputs).
WriteIoPorts	Writes to the I/O output pins

I²C Transactions

ReadI2c	Executes an I²C read transaction.
WriteI2c	Executes an I²C write transaction.

Debug Buffer

ReadDebugBuffer	Read the contents of the debug buffer.

GetDllVersion

The GetDllVersion function is used to obtain the revision information for the API dll file.

```
WORD _stdcall DAPI_GetDllVersion(void);
```

Parameters

none

Return Value

A WORD (unsigned long) containing the revision value formatted in binary coded decimal.
high byte = major revision, low byte = minor revision.
a returned value of 0x0201 would represent version 2.01.

Comments

This function provide a simple method for an application to obtain the version of the installed API dll file. This information can be useful for determining compatibility and for troubleshooting.

See Also

GetDriverVersion, GetFirmwareVersion

GetDriverVersion

Important Notice: This function is NOT CURRENTLY IMPLEMENTED.

The GetDriverVersion function is used to obtain the revision information for the USB I²C/IO device driver file.

```
WORD _stdcall DAPI_GetDriverVersion(void);
```

Parameters

none

Return Value

A WORD (unsigned long) containing the revision value formatted in binary coded decimal.
high byte = major revision, low byte = minor revision.
a returned value of 0x0201 would represent version 2.01.

Comments

This function provide a simple method for an application to obtain the version of the installed USB I²C/IO device driver file. This information can be useful for determining compatibility and for troubleshooting.

See Also

GetDllVersion, GetFirmwareVersion

GetFirmwareVersion

Important Notice: This function is NOT CURRENTLY IMPLEMENTED.

The GetFimwareVersion function is used to obtain the revision information for the firmware executing on a specific device.

```
WORD _stdcall DAPI_GetFimwareVersion(HANDLE hDevInstance);
```

Parameters

HANDLE hDevInstance
A valid device instance handle.

Return Value

A WORD (unsigned long) containing the revision value formatted in binary coded decimal.
high byte = major revision, low byte = minor revision.
a returned value of 0x0201 would represent version 2.01 of the API dll.

Comments

This function provide a simple method for an application to obtain the version of the USB I²C/IO device firmware that is currently loaded and executing on a specified device. This information can be useful for determining compatibility and for troubleshooting.

See Also

GetDllVersion, GetDriverVersion

GetDeviceCount

The GetDeviceCount function is used to determine how many USB I²C/IO devices are currently attached.

```
BYTE _stdcall DAPI_GetDeviceCount( LPSTR lpsDevName );
```

Parameters

LPSTR lpsDevName
A long pointer to a string, which contains the symbolic name of the device's driver. For the generic USB I²C/IO device driver the symbolic name is "UsbI2cIo".

Return Value

A BYTE (unsigned char) representing the number of attached devices.

Comments

Custom versions of the USB I²C/IO device driver will have different symbolic names.

See Also

GetDeviceInfo

GetDeviceInfo

The GetDeviceInfo function is used to obtain a list of all attached USB I²C/IO devices and their Serial IDs.

BYTE _stdcall DAPI_GetDeviceInfo(LPSTR lpsDevName, LPDEVINFO lpDevInfo);

Parameters

LPSTR lpsDevName

A long pointer to a string, which contains the symbolic name of the device's driver. For the generic USB I²C/IO device driver the symbolic name is "UsbI2cIo".

LPDEVINFO lpDevInfo

A long pointer to an array of device info structures. Information for each detected device will be written to the buffer, and indexed by the device count.
The device info structure is defined in the dll as:

```
typedef struct _DEVINFO {          // structure for device information
  BYTE byInstance;
  BYTE SerialId[9];
} DEVINFO, *LPDEVINFO;
```

Return Value

A BYTE (unsigned char) representing the number of attached devices, and the number of entries written to the array of device info structures pointed to by the lpDevInfo parameter.

Comments

Custom versions of the USB I²C/IO device driver will have different symbolic names.

See Also

GetDeviceCount

GetSerialId

The GetSerialId function is used to obtain a devices Serial ID string. The Serial ID uniquely identifies individual USB I²C/IO devices.

```
BOOL _stdcall DAPI_GetSerialId(HANDLE hDevInstance, LPSTR lpsSerialId);
```

Parameters

HANDLE hDevInstance
A HANDLE (long int) to a device instance. Specifies which USB I²C/IO device instance.

LPSTR lpsSerialId
A long pointer to a string, which will be used to store the Serial ID string obtained from the specified device. USB I²C/IO Serial ID strings are 9 bytes long, including the null termination.

Return Value

A BOOL indicating whether a Serial ID was successfully obtained from the specified device.

Comments

The Serial ID string provides a mechanism for applications to identify, remember, and communicate with uniquely identified USB I²C/IO devices.

See Also

GetDeviceInfo, OpenDeviceBySerialId

DetectDevice

The DetectDevice function is used to determine if a previously opened device is still attached.

```
BOOL _stdcall DAPI_DetectDevice(HANDLE hDevInstance);
```

Parameters

HANDLE hDevInstance
A HANDLE (long int) to a device instance. Specifies which USB I²C/IO device instance.

Return Value

A BOOL indicating whether the device is still attached (TRUE when device is attached).

Comments

This function provides a fast mechanism for monitoring device connection/disconnection. It generates a call to the device driver, but does not generate any USB traffic.

See Also

OpenDeviceInstance, OpenDeviceBySerialId

OpenDeviceInstance

The OpenDeviceInstance function is used to obtain a handle to a specific instance of a device driver.

```
HANDLE _stdcall DAPI_OpenDeviceInstance(LPSTR lpsDevName, BYTE byDevInstance);
```

Parameters

LPSTR lpsDevName
A long pointer to a string, which contains the symbolic name of the device's driver. For the generic USB I²C/IO device driver the symbolic name is "UsbI2cIo".

BYTE byDevInstance
A byte representing the device instance. The first device attached is usually device instance 0, the 2nd is usally 1, etc.

Return Value

A HANDLE (long int) to the specified device instance. The handle should always be compared to the value INVALID_HANDLE_VALUE to determine if a valid handle was obtained.

Comments

Custom versions of the USB I²C/IO device driver will have different symbolic names.

See Also

OpenDeviceBySerialId, CloseDeviceInstance, GetDeviceCount, GetDeviceInfo

CloseDeviceInstance

The CloseDeviceInstance function is used to close a handle to a device driver instance.

```
BOOL _stdcall DAPI_CloseDeviceInstance(HANDLE hDevInstance);
```

Parameters

HANDLE hDevInstance
A HANDLE (long int) to a device instance. Specifies which USB I²C/IO device instance.

Return Value

Returns a non-zero value to indicate success, zero to indicate failure.

Comments

Applications should close all open device handles before exiting.

See Also

OpenDeviceInstance, OpenDeviceBySerialId

OpenDeviceBySerialId

The OpenDeviceBySerialId function is used to obtain a handle to a device specified only by it's Serial ID string.

```
HANDLE _stdcall DAPI_OpenDeviceBySerialId(LPSTR lpsDevName, LPSTR lpsDevSerialId);
```

Parameters

LPSTR lpsDevName
A long pointer to a string, which contains the symbolic name of the device's driver.
For the generic USB I²C/IO device driver the symbolic name is "UsbI2cIo".

LPSTR lpsDevSerialId
A long pointer to a string, which contains the Serial ID string identifying the device to be opened.
USB I²C/IO Serial ID strings are 9 bytes long, including the null termination.

Return Value

A HANDLE (long int) to the device instance corresponding to the device identified by Serial ID. The handle should always be compared to the value INVALID_HANDLE_VALUE to determine if a valid handle was obtained.

Comments

Custom versions of the USB I²C/IO device driver will have different symbolic names.

See Also

OpenDeviceInstance, CloseDeviceInstance, GetDeviceCount, GetDeviceInfo

ConfigIoPorts

The ConfigIoPorts function is used to configure the I/O port bits as inputs or outputs.

```
BOOL _stdcall DAPI_ConfigIoPorts(HANDLE hDevInstance, ULONG ulIoPortConfig);
```

Parameters

HANDLE hDevInstance
A HANDLE (long int) to a device instance. Specifies which USB I²C/IO device instance.

ULONG ulIoPortConfig
An unsigned long specifying the desired configuration for the I/O port bits.

The bit mapping is a follows:
0x000CBBAA
Where C, B, and A correspond to the port bits
byte[0] bits 7..0 = Port A bits 7..0 configuration value
byte[1] bits 7..0 = Port B bits 7..0 configuration value
byte[2] bits 3..0 = Port C bits 7..4 configuration value (Port C bits 3..0 are reserved)
byte[2] bits 7..4 = reserved
byte[3] bits 7..0 = reserved

For all bits, a 1 indicates configuration as Input, a 0 indicates configuration as Output

Return Value

Returns a non-zero value to indicate success, zero to indicate failure.

Comments

A valid device instance handle must be obtained prior to calling ConfigIoPorts.

See Also

GetIoConfig, ReadIoPorts, WriteIoPorts

GetIoConfig

The GetIoConfig function is used to read the configuration of the I/O port bits.

```
BOOL _stdcall DAPI_GetIoConfig(HANDLE hDevInstance, LPLONG lpulIoPortConfig);
```

Parameters

HANDLE hDevInstance
A HANDLE (long int) to a device instance. Specifies which USB I²C/IO device instance.

LPLONG lpulIoPortConfig
A long pointer to an unsigned long specifying the location to store the I/O port configuration data read from the USB-I2CIO board.

The bit mapping is a follows:
0x000CBBAA
Where C, B, and A correspond to the port bits
byte[0] bits 7..0 = Port A bits 7..0 configuration value
byte[1] bits 7..0 = Port B bits 7..0 configuration value
byte[2] bits 3..0 = Port C bits 7..4 configuration value (Port C bits 3..0 are reserved)
byte[2] bits 7..4 = reserved
byte[3] bits 7..0 = reserved

For all bits, a 1 indicates configuration as Input, a 0 indicates configuration as an Output.

Return Value

Returns a non-zero value to indicate success, zero to indicate failure.

Comments

A valid device instance handle must be obtained prior to calling ConfigIoPorts.

See Also

ConfigIoPorts, ReadIoPorts, WriteIoPorts

ReadIoPorts

The ReadIoPorts function is used to read the I/O port pins. The state of all pins are read, regardless of whether the pins are configured as inputs or outputs.

BOOL _stdcall DAPI_ReadIoPorts(HANDLE hDevInstance, LPLONG lpulIoPortData);

Parameters

HANDLE hDevInstance
A HANDLE (long int) to a device instance. Specifies which USB I²C/IO device instance.

LPLONG lpulIoPortData
A long pointer to an unsigned long specifying the location to store the data read from the I/O ports.

The bit mapping for the read data is as follows:
0x000CBBAA
Where C, B, and A correspond to the port bits
byte[0] bits 7..0 = Port A bits 7..0 data value
byte[1] bits 7..0 = Port B bits 7..0 data value
byte[2] bits 3..0 = Port C bits 7..4 data value (Port C bits 3..0 are reserved)
byte[2] bits 7..4 = reserved
byte[3] bits 7..0 = reserved

Return Value

Returns a non-zero value to indicate success, zero to indicate failure.

Comments

A valid device instance handle must be obtained prior to calling ReadIoPorts.

When configured as inputs, the I/O pins "float" and will produce intermittent values unless they are driven high or low by external circuitry.

See Also

ConfigIoPorts, GetIoConfig, WriteIoPorts

WriteIoPorts

The WriteIoPorts function is used to write to the I/O port pins which are configured as outputs.

BOOL _stdcall DAPI_WriteIoPorts(HANDLE hDevInstance, ULONG ulIoPortData, ULONG ulIoPortMask);

Parameters

HANDLE hDevInstance
A HANDLE (long int) to a device instance. Specifies which USB I²C/IO device instance.

ULONG ulIoPortData
An unsigned long specifying the data to write to the I/O ports.

The bit mapping for the write data is as follows:
0x000CBBAA
Where C, B, and A correspond to the port bits
byte[0] bits 7..0 = Port A bits 7..0 data value
byte[1] bits 7..0 = Port B bits 7..0 data value
byte[2] bits 3..0 = Port C bits 7..4 data value (Port C bits 3..0 are reserved)
byte[2] bits 7..4 = reserved
byte[3] bits 7..0 = reserved

ULONG ulIoPortMask
An unsigned long specifying the data mask to use when modifying the I/O ports outputs. The mask value allows a Read-Modify-Write operation to occur at the firmware level, which frees the application software from having to maintain an image of the ports, and reduces USB traffic.

The bit mapping for the mask is as follows:
0x000CBBAA
Where C, B, and A correspond to the port bits
byte[0] bits 7..0 = Port A bits 7..0 mask value
byte[1] bits 7..0 = Port B bits 7..0 mask value
byte[2] bits 3..0 = Port C bits 7..4 mask value (Port C bits 3..0 are reserved)
byte[2] bits 7..4 = reserved
byte[3] bits 7..0 = reserved

Return Value

Returns a non-zero value to indicate success, zero to indicate failure.

Comments

A valid device instance handle must be obtained prior to calling WriteIoPorts.

Only port bits which were previously configured as outputs are written.

If you are confused by the "ulIoPortMASK" parameter, you can simply set it to 0xFFFFFFFF and always modify the value of all pins which are configured as outputs.

All of the I/O pins default to inputs upon power-up and "float". This makes it possible to set a "default state" for any pins that will eventually be used as outputs, by simply connecting a pull-up or pull-down resistor.

See Also

ConfigIoPorts, GetIoConfig, ReadIoPorts

ReadI2c

The ReadI2c function is used to execute an I²C read transaction.

LONG _stdcall DAPI_ReadI2c(HANDLE hDevInstance, I2C_TRANS * TransI2C);

Parameters

HANDLE hDevInstance
A HANDLE (long int) to a device instance. Specifies which USB I²C/IO device instance.

*I2C_TRANS * TransI2C*
A pointer to an I2C_TRANS structure. The I2C_TRANS structure is used to specify the details of an I²C transaction (device address, number of bytes, etc.).

The I2C_TRANS structure is defined in the dll header file as follows:

```
typedef struct _I2C_TRANS {
  BYTE byTransType;
  BYTE bySlvDevAddr;
  WORD wMemoryAddr;
  WORD wCount;
  BYTE Data[256];
} I2C_TRANS, *PI2C_TRANS;
```

Return Value

On success, returns the number of bytes successfully read from the specified I²C device.
On failure, returns a negative number.

Comments

A valid device instance handle must be obtained prior to calling the ReadI2c function.

The read data is written to the TransI2c.Data buffer.

The maximum number of bytes per transfer is currently limited to 64, not 256 as implied by the definition of the I2C_TRANS structure.

See Also

WriteI2c

WriteI2c

The WriteI2c function is used to execute an I²C write transaction.

LONG _stdcall DAPI_WriteI2c(HANDLE hDevInstance, I2C_TRANS * TransI2C);

Parameters

HANDLE hDevInstance
A HANDLE (long int) to a device instance. Specifies which USB I²C/IO device instance.

*I2C_TRANS * TransI2C*
A pointer to an I2C_TRANS structure. The I2C_TRANS structure is used to specify the details of an I²C transaction (device address, number of bytes, etc.).

The I2C_TRANS structure is defined in the dll header file as follows:

```
typedef struct _I2C_TRANS {
  BYTE byTransType;
  BYTE bySlvDevAddr;
  WORD wMemoryAddr;
  WORD wCount;
  BYTE Data[256];
} I2C_TRANS, *PI2C_TRANS;
```

Return Value

On success, returns the number of bytes successfully written to the specified I²C device.
On failure, returns a negative number.

Comments

A valid device instance handle must be obtained prior to calling the WriteI2c function.

The write data is obtained from the TransI2c.Data buffer.

The maximum number of bytes per transfer is currently limited to 64, not 256 as implied by the definition of the I2C_TRANS structure.

See Also

ReadI2c

ReadDebugBuffer

The ReadDebugBuffer function is used to read debug data from the device.

```
LONG _stdcall DAPI_ReadDebugBuffer(LPSTR lpsDebugInfo, HANDLE hDevInstance,  LONG lMaxBytes);
```

Parameters

LPSTR lpsDebugInfo
A long pointer to the string where the debug information will be written.

HANDLE hDevInstance
A HANDLE (long int) to a device instance. Specifies which USB I²C/IO device instance.

LONG lMaxBytes
A long indicating the maximum number of bytes that can be written to lpsDebugInfo.

Return Value

On success, returns the number of bytes successfully read from the debug buffer.
On Failure, returns –1.

Comments

A valid device instance handle must be obtained prior to calling the ReadDebugBuffer function.

See Also

None

I²C Transactions

Overview

The USB I²C/IO API provides a set of 'C' functions for communicating with devices that implement an I²C interface. The API is implemented at a "transaction" level in order to shield the user from having to deal with the details of I²C communications (start bits, stop bits, ack, nak, etc.).

The API provides support for the following tasks:

- Reading data from I2C slave devices.
- Writing data to I2C slave devices.

Comments

The USB-I²C/IO implements a "Single Master" I²C interface.
The USB- I²C /IO performs it's I²C signaling at 90Kbps.
The USB- I²C /IO cannot be configured as an I²C "Slave".
The USB- I²C /IO does not provide any I²C "bus monitoring" functionality.
The USB- I²C/IO board incorporates a 16KB I²C eeprom which must be accessible to the onboard micro-controller for normal board operation. The user should not attach conflicting or interfering devices to the I²C connector. The eeprom is at address I²C address 0xA2.

Implementation

Our approach to I2C is really very simple.

The user defines an I2C transaction by declaring an instance of a packed structure of type I2C_TRANS (defined in the file "usbi2cio.h").

```
typedef struct _I2C_TRANS {
  BYTE byTransType;
  BYTE bySlvDevAddr;
  WORD wMemoryAddr;
  WORD wCount;
  BYTE Data[256];
} I2C_TRANS, *PI2C_TRANS;
```

The I2C_TRANS structure consists of member variables that provide support for most I2C transactions.

There is a member variable for specifying the type of I2C transaction (byTransType). The value of byTransType can currently be set to one of the following enumerated values:

I2C_TRANS_NOADR
I2C_TRANS_8ADR
I2C_TRANS_16ADR.
I2C_TRANS_NOADR_NS.

Basic I2C devices will normally use the I2C_TRANS_NOADR value. Memory devices will use either the I2C_TRANS_8ADR (8 bit extended addressing, small memory devices), or I2C_TRANS_16ADR (16 bit extended addressing, larger memory devices). The I2C_TRANS_NOADR_NS transaction is identical to the I2C_TRANS_NOADR transaction with the exception that STOP signaling at the end of the transaction is suppressed, allowing custom transaction protocols to be built from two or more sequential transactions.

The member variable "bySlvDevAddr" is used for specifying the value of the "Control Byte" for the I2C transaction. The control byte of an I2C transaction contains several bit fields. These fields are used to identify the specific device for the transactions (bits 7-1), and whether the transaction is a read or write operation (bit 0). The value of the read/write bit is automatically set or cleared by the API software as required to perform the specified transaction. Your software should use a zero value at all times for the read/write bit. The device address is specified by setting/clearing bits 7 through 1. If you are trying to communicate with a device who's control byte is 0101000x (binary, where x is the read write bit), you would specify a value of 0x50 for bySlvDevAddr. No bit shifting is performed on the bySlvDevAddr value.

The member variable "wMemoryAddr" is used for specifying the value of the memory address *within* the device. When byTransType is set to I2C_TRANS_NOADR or I2C_TRANS_NOADR_NS, the member variable wMemoryAddr is unused. When either I2C_TRANS_8ADR or I2C_TRANS_16ADR is specified, wMemoryAddr is used to specify the 8 or 16 bit extended memory address for the transaction.

The member variable "wCount" is used for specifying the number of bytes to be read or written during the transaction. I^2C transactions are currently limited to 64 bytes of data.

The member variable "Data" is a 256 byte buffer, which stores the read/write data for the transaction. When performing an I2C write transaction, the data to be written is placed in the Data array (by the user application) before performing the write transaction. For read transactions, the read data is placed in the Data array by our software, and is available to the user application after performing the read transaction. Note: While the structure is currently defined with a 256 byte data array, transactions are currently limited to 64 bytes of data.

The I2C transactions are executed by calling one of two functions, DAPI_ReadI2c() or DAPI_WriteI2c(). These functions require two parameters, a handle to the USB I2C/IO board which will execute the transaction, and a pointer to the I2C_TRANS structure instance. Both functions return a LONG value (long integer) which should be the same value as what was specified for wCount prior to make the call (on a successful transaction). The return value will be negative for unsuccessful calls.

Typical I2C operation would require the following steps.

- Declare an instance of a board handle.
- Open a handle to the board.
- Declare an instance of an I2C_TRANS structure.
- Initialize the member variables of the structure as appropriate for the desired I2C transaction.
- Call either the ReadI2c() or WriteI2c() functions, passing the board handle and the address of the I2C_TRANS structure instance.
- Upon returning from the ReadI2c or WriteI2c call, compare the returned value to wCount, they should be equal.
- If a read was performed, the read data will be stored in the Data array of the I2C_TRAN structure instance.

