General Standards Corporation
8302A Whitesburg Drive
Huntsville, AL 36802
Phone (256) 880-8787
Fax (256) 880-8788

General Standards Corporation

PCI-16SDI Windows NT Device Driver User's Manual

Windows NT Device Driver Software for the
General Standards PCI-16SDI
hosted on x86 Processors

Document number:		Revision:	1.0	Date: 8/09/99
Engineering Approval:				Date:
Quality Representative Approval:				Date:

Acknowledgments

1. Scope

The purpose of this document is to describe how to interface with the PCI-16SDI Windows NT Device Driver developed by General Standards Corporation (GSC). This software provides the interface between "Application Software" and the PCI-16SDI Board. The interface to this board is at the device level. It requires no knowledge of the actual board addressing or device register contents.

The PCI-16SDI Driver Software executes under control of the Windows NT operating system. The PCI-16SDI is implemented as a standard Windows NT device driver written in the 'C' programming language. The PCI-16SDI Driver Software is designed to operate on CPU boards containing standard x86 processors.

2. Hardware Overview

The General Standards Corporation (GSC) PCI-16SDI board is a 16-Bit, 16-Channel Sigma-Delta, 220 KSPS Analog Input PCI Board. Each of the sigma-delta analog input channels can be controlled by any one of four independent sample clocks and multiple channels can be harmonically locked together. A/D conversions on multiple boards can be synchronized and phase-locked. Sample rates are adjustable from 5 KSPS to 220 KSPS, and the input range is software selectable as +/-1.25V, +/-2.5V, +/-5V or +/-10V. Internal autocalibration networks permit periodic calibration to be performed without removing the board from the system.

The PCI-16SDI board includes a DMA controller and 256K samples of FIFO buffering. The board also provides for interrupt generation for various states of the board, including operation complete, sample buffer threshold transition and sample buffer almost empty and full.

3. Referenced Documents

The following documents provide reference material for the PCI-16SDI Board:

- PCI-16SDI User's Manual - General Standards Corporation.
- PLX Technology, Inc. PCI 9080 PCI Bus Master Interface Chip data sheet.

4. Driver Interface

The PCI-16SDI Driver conforms to the device driver standards required by the Windows NT Operating System and contains the following standard driver entry points.

- CreateFile() - opens a driver interface to one PCI-16SDI Card
- CloseHandle() - closes a driver interface to one PCI-16SDI Card
- ReadFile() - reads A/D Samples received from a PCI-16SDI Card
- DeviceIoControl() - performs various control and setup functions on the PCI-16SDI Card

The PCI-16SDI Device Driver provides a standard driver interface to the GSC PCI-16SDI card for Windows NT applications which run on a x86 target processor. The device driver is installed and devices are created when the driver is started during boot up. The functions of the driver can then be used to access the board. Devices are created with the name "sdix" where 'x' is the device number. Device numbers start at 0 for Windows NT applications and 1 for DOS applications. For each PCI-16SDI board found, the device number increments.

Included in the device driver software package is a menu driven board application program and source code. This program is delivered undocumented and unsupported but may be used to exercise the PCI-16SDI card and device driver. It can also be used to break the learning curve somewhat for programming the PCI-16SDI device.

It is important to note that the PCI-16SDI device driver is target processor dependent. System calls are made within the driver which are only available on x86 processors.

When the driver is installed during the power up of the computer, certain default values are set by the driver. These values are not reset to the default values every time the user calls the CreateFile function. The following default values are set:

➢ Read Timeout	= 10 seconds
➢ DMA Enable	= Disabled

4.1. CreateFile()

The CreateFile() function is the standard NT entry point to open a connection to a PCI-16SDI Card. This function must be called before any other driver function may be called to control the device. The fdwAttrsAndFlags parameter needs to be set to FILE_FLAG_OVERLAPPED if overlapped I/O is to be performed. Using overlapped I/O for reads allows the calling task to continue executing while the driver is performing the I/O operation and making calls to GetOverlappedResult() to determine when the operation is complete. See the SDITest sample code for a example on how to perform overlapped I/O.

Multiple tasks may call CreateFile to access the driver for the same board. The programmer needs to be very careful if this is desirable. One task may set values that conflict with the other task.

PROTOTYPE:

```
HANDLE CreateFile(LPCTSTR                lpszName,
                  DWORD                  fdwAccess,
                  DWORD                  fdwShareMode,
                  LPSECURITY_ATTRIBUTES  lpsa,
                  DWORD                  fdwCreate,
                  DWORD                  fdwAttrsAndFlags,
                  HANDLE                 hTemplateFile);
```

Where:

lpszName - name of the device being opened which is "sdix" where x is the device number.
Device numbers begin with 0 for NT applications and 1 for DOS applications.
Each device is consecutively numbered after that.

fdwAccess – a logically or'ed combination of one or more of the following access modes:

GENERIC_ALL - Execute, Read and Write Access
GENERIC_EXECUTE - Execute Access
GENERIC_READ - Read Access
GENERIC_WRITE - Write Access

Use (GENERIC_WRITE | GENERIC_READ) for this parameter.

fdwShareMode – a logically or'ed combination of zero or more of the following share options:

FILE_SHARE_READ - Read Share Mode
FILE_SHARE_WRITE - Write Share Mode

This parameter is usually zero.

lpsa – a pointer to a security attributes structure.

This parameter is usually NULL.

fdwCreate – a logically or’ed combination of one or more of the following device creation options:

CREATE_NEW	- Create new file
CREATE_ALWAYS	- Always create file
OPEN_EXISTING	- Open existing file/device
OPEN_ALWAYS	- Always open file
TRUNCATE_EXISTING	- Truncate file

Use OPEN_EXISTING for this parameter.

fdwAttrsAndFlags – a logically or’ed combination of zero or more of the following attributes and flags:

FILE_ATTRIBUTE_READONLY	- Read-only file/device
FILE_ATTRIBUTE_HIDDEN	- Hidden file
FILE_ATTRIBUTE_SYSTEM	- System file
FILE_ATTRIBUTE_DIRECTORY	- Directory file
FILE_ATTRIBUTE_ARCHIVE	- Archive file
FILE_ATTRIBUTE_NORMAL	- Normal file/device
FILE_ATTRIBUTE_TEMPORARY	- Temporary file
FILE_FLAG_WRITE_THROUGH	- Write through access
FILE_FLAG_OVERLAPPED	- Overlapped access
FILE_FLAG_NO_BUFFERING	- No buffering
FILE_FLAG_RANDOM_ACCESS	- Random Access
FILE_FLAG_SEQUENTIAL_SCAN	- Sequential Scan
FILE_FLAG_DELETE_ON_CLOSE	- Delete file on Close
FILE_FLAG_BACKUP_SEMANTICS	- Backup semantics
FILE_FLAG_POSIX_SEMANTICS	- POSIX semantics
FILE_FLAG_OPEN_REPARSE_POINT	- Open reparse point
FILE_FLAG_OPEN_NO_RECALL	- Open no recall

Use either FILE_ATTRIBUTE_NORMAL or FILE_ATTRIBUTE_OVERLAPPED for this parameter. Use FILE_ATTRIBUTE_OVERLAPPED if you plan to use overlapped I/O.

hTemplateFile – handle to a template file.

Use NULL for this parameter.

Returns a handle to the device opened on success. This handle is then used as a parameter to all other device accesses. Returns a NULL when the create fails.

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwErrorCode;


/*   Open the PCI-16SDI device sdi1   */
hDevice = CreateFile(“\\\\.\\sdi1”, GENERIC_READ | GENERIC_WRITE, 0,
                     NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL);
if (hDevice == INVALID_HANDLE_VALUE)
{
   dwErrorCode = GetLastError();
   ErrorMessage(“CreateFile”, dwErrorCode);
   ExitProcess(dwErrorCode);
}

/* Access the device here. */

/* Close the device here. */
```

4.2. CloseHandle()

The CloseHandle() function is the driver entry point to close a connection to a PCI-16SDI Card. This function should only be called after the CreateFile() function has been successfully called for a PCI-16SDI Card. The CloseHandle() function doses an interface to a PCI-16SDI device.

If multiple tasks have created connections to the driver using the CreateFile function, the CloseHandle call will only close the connection for that particular task.

PROTOTYPE:

```
BOOL CloseHandle(HANDLE hObject);
```

Where:

hObject - handle to the device to close. The return parameter from a CreateFile() call.

Returns TRUE if successful or FALSE if unsuccessful.

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwErrorCode;

/* Open the device and get hDevice here. */

/* Access the device here. */

/*  Close the SDI Device.  */
if  (! CloseHandle(hDevice))
{
   dwErrorCode = GetLastError();
   ErrorMessage(“CloseHandle”, dwErrorCode);
}
hDevice = NULL;
```

4.3. ReadFile()

The ReadFile() function is the driver entry point to read A/D sample data from a PCI-16SDI Card. This function should only be called after the CreateFile() function has been successfully called for a PCI-16SDI Card. The ReadFile() function reads the number of bytes requested from the receive FIFO by the **NumberOfBytesToRead** parameter. Note that this parameter is the number of **bytes**, not the number of **words**. Therefore, the number of samples to be read should be multiplied by four for this parameter. If the device was selected for overlapped I/O in the CreateFile() call, this function may return without any words being read and a call to the GetOverlappedResult() needs to be made to determine when the operation completes.

If multiple tasks try to access the ReadFile function at the same time, the driver will process each request in the order they are received. This function will perform programmed I/O (PIO) or Direct Memory Access (DMA) transfers depending upon whether DMA is enabled using the IOCTL_SDI_SET_DMA_ENABLE ioctl() function call. DMA transfers allow the reading of the samples to be achieved without using CPU time allowing the application to perform operations concurrently.

If this function is called to read more samples than currently exist in the board buffer, the data read after the buffer is emptied will be invalid. It is up to the application to ensure that more data is not requested than exists in the buffer. This can be accomplished by using the interrupt notification ioctl() function to indicate to the application when the buffer threshold has been exceeded or the sample buffer is almost full.

The format of each sample word read is as follows:

Bits D19-D16	Bits D15-D0
Channel Tag (0-15)	16-bit A/D Sample Value in either Offset Binary or Two's Complement depending on the setting of the IOCTL_SDI_SET_DATA_FORMAT ioctl() function

PROTOTYPE:

```
BOOL ReadFile( HANDLE         hFile,
               LPVOID         lpBuffer,
               DWORD          NumberOfBytesToRead,
               LPDWORD        lpNumberOfBytesRead,
               LPOVERLAPPED   lpOverlapped);
```

Where:

hFile - handle to the device returned from CreateFile()

lpBuffer - pointer to a buffer to store the data read

NumberOfBytesToRead - number of bytes to read

lpNumberOfBytesRead - pointer to a location to return the number of bytes read

lpOverlapped - pointer to an overlapped structure. If device was opened using overlapped I/O, this structure is required. If device was not opened using overlapped I/O, this parameter should be NULL. See WIN32 documentation for a structure definition and the SDITest Program for an example of how to use it.

Returns TRUE on success. Returns FALSE on failure or if I/O is still pending on an overlapped I/O operation.

<u>EXAMPLE:</u>

```
#define NUM_SAMPLES 80
HANDLE   hDevice;
ULONG    ulBuffer[NUM_SAMPLES];
DWORD    dwBytesRead = 0;
DWORD    dwErrorCode;
BOOL     status;

/*  Read from the PCI-16SDI device  */
status = ReadFile(hDevice, ulBuffer, NUM_SAMPLES*4,
                  &dwBytesRead, NULL);
if (! status)
{
   dwErrorCode = GetLastError();
   ErrorMessage(“ReadFile”, dwErrorCode);
}
else
{
   if (dwBytesRead != (NUM_SAMPLES*4))
   {
      printf(“Only read %d bytes\n”, dwBytesRead);
   }
   else
   {
      /* Data read OK.  Use the data here. */
   }
}
```

4.4. DeviceIoControl()

The DeviceIoControl() function is the device entry point to perform control and setup operations on a PCI-16SDI Card. This function should only be called after the CreateFile() function has been successfully called for a PCI-16SDI Card. The DeviceIoControl() function will perform different functions based upon the dwIoControlCode parameter. These functions will be described in the following subparagraphs.

Certain DeviceIoControl function calls should not be used unless absolutely necessary. These routines are provided so that the driver is complete and does not limit the use of the PCI-16SDI board. Each of the subsections that follow will describe the limitations on their use.

PROTOTYPE:

```
BOOL DeviceIoControl(HANDLE         hDevice,
                     DWORD          dwIoControlCode,
                     LPVOID         lpInBuffer,
                     DWORD          nInBufferSize,
                     LPVOID         lpOutBuffer,
                     DWORD          nOutBufferSize,
                     LPDWORD        lpBytesReturned,
                     LPOVERLAPPED   lpOverlapped);
```

Where:

hDevice - handle to the device returned from CreateFile()

dwIoControlCode - control code for the operation to perform. One of the following constants from the include file "**SDIIoctl.h**":

- IOCTL_SDI_NO_COMMAND
- IOCTL_SDI_READ_REGISTER
- IOCTL_SDI_WRITE_REGISTER
- IOCTL_SDI_REQ_INT_NOTIFY
- IOCTL_SDI_SET_INPUT_RANGE
- IOCTL_SDI_SET_INPUT_MODE
- IOCTL_SDI_SET_SW_SYNC
- IOCTL_SDI_AUTO_CAL
- IOCTL_SDI_INITIALIZE
- IOCTL_SDI_SET_DATA_FORMAT
- IOCTL_SDI_SET_INITIATOR_MODE
- IOCTL_SDI_SET_BUFFER_THRESHOLD
- IOCTL_SDI_CLEAR_BUFFER
- IOCTL_SDI_SET_ACQUIRE_MODE
- IOCTL_SDI_SET_GEN_RATE

- IOCTL_SDI_ASSIGN_GEN_TO_GROUP
- IOCTL_SDI_SET_RATE_DIVISOR
- IOCTL_SDI_GET_DEVICE_ERROR
- IOCTL_SDI_READ_PCI_CONFIG
- IOCTL_SDI_READ_LOCAL_CONFIG
- IOCTL_SDI_WRITE_PCI_CONFIG_REG
- IOCTL_SDI_WRITE_LOCAL_CONFIG_REG
- IOCTL_SDI_SET_TIMEOUT
- IOCTL_SDI_DMA_ENABLE

lpInBuffer - pointer to a buffer that contains the data required to perform the operation. This parameter can be NULL if the dwIoControlCode parameter specifies an operation that does not require input data. See the individual subsections for a description of the structures required.

nInBufferSize - size, in bytes, of the buffer pointed to by lpInBuffer

lpOutBuffer - pointer to a buffer that receives the operation's output data. This parameter can be NULL if the dwIoControlCode parameter specifies an operation that does not produce output data. See the individual subsections for a description of the structures required.

nOutBufferSize - size, in bytes, of the buffer pointed to by lpOutBuffer

lpBytesReturned - pointer to a variable that receives the size, in bytes, of the data stored into the buffer pointed to by lpOutbuffer.

lpOverlapped - pointer to an overlapped structure. If device was opened using overlapped I/O, this structure is required. If device was not opened using overlapped I/O, this parameter should be NULL. See WIN32 documentation for a structure definition and the SDITest Program for an example of how to use it.

Returns TRUE if successful or FALSE on failure or if I/O is still pending on an overlapped I/O operation.

4.4.1. IOCTL_SDI_NO_COMMAND

This is an empty driver entry point. This command may be given to validate that the driver is correctly installed and that the PCI-16SDI Board Device has been successfully opened.

Input/Output Buffer:

not used

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwErrorCode;
DWORD    dwTransferSize;

if (! DeviceIoControl(hDevice, IOCTL_SDI_NO_COMMAND, NULL, 0,
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.2. IOCTL_SDI_READ_REGISTER

The IOCTL_SDI_READ_REGISTER function reads and returns the contents of one of the PCI-16SDI Registers.

Input/Output Buffer:

```
<from SDIIoctl.h>

typedef struct _SDI_REGISTER_PARAMS
{
    ULONG eSDIRegister;
    ULONG ulRegisterValue;
} SDI_REGISTER_PARAMS, *PSDI_REGISTER_PARAMS;

Where ulRegisterValue will store the value read from the register
and eSDIRegister is one of the following:

#define BOARD_CTRL_REG            0
#define RATE_CTRL_A_REG           1
#define RATE_CTRL_B_REG           2
#define RATE_CTRL_C_REG           3
#define RATE_CTRL_D_REG           4
#define RATE_ASSIGN_REG           5
#define RATE_DIVISOR_00_01_REG    6
#define RATE_DIVISOR_02_03_REG    7
#define RATE_DIVISOR_04_05_REG    8
#define RATE_DIVISOR_06_07_REG    9
#define RATE_DIVISOR_08_09_REG   10
#define RATE_DIVISOR_10_11_REG   11
#define RATE_DIVISOR_12_13_REG   12
#define RATE_DIVISOR_14_15_REG   13
#define BUFFER_THRESHOLD_REG     14
#define INPUT_DATA_BUFFER_REG    18
```

EXAMPLE:

```
HANDLE               hDevice;
DWORD                dwTransferSize;
DWORD                dwErrorCode;
SDI_REGISTER_PARAMS  InputRegData;
SDI_REGISTER_PARAMS  OutputRegData;

InputRegData.eSDIRegister     = BOARD_CTRL_REG;
OutputRegData.ulRegisterValue = 0xDEADBEEF;

if ((! DeviceIoControl(hDevice, IOCTL_SDI_READ_REGISTER,
                       &InputRegData, sizeof(SDI_REGISTER_PARAMS),
                       &OutputRegData, sizeof(SDI_REGISTER_PARAMS),
                       &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(SDI_REGISTER_PARAMS)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   printf("Board Control Register = %08lx\n”,
          OutputRegData.ulRegisterValue);
}
```

4.4.3. IOCTL_SDI_WRITE_REGISTER

The IOCTL_SDI_WRITE_REGISTER function writes a value to one of the PCI-16SDI Registers. The user should be very careful modifying values of certain registers. All SDI Registers may be manipulated using the driver's ioctl() functions. It is recommended that the ioctl() functions be used instead of the IOCTL_SDI_WRITE_REGISTER routine.

Input/Output Buffer:

```
<from SDIIoctl.h>

typedef struct _SDI_REGISTER_PARAMS
{
    ULONG eSDIRegister;
    ULONG ulRegisterValue;
} SDI_REGISTER_PARAMS, *PSDI_REGISTER_PARAMS;

Where ulRegisterValue contains the value to be written to the
register and eSDIRegister is one of the following:

#define BOARD_CTRL_REG            0
#define RATE_CTRL_A_REG           1
#define RATE_CTRL_B_REG           2
#define RATE_CTRL_C_REG           3
#define RATE_CTRL_D_REG           4
#define RATE_ASSIGN_REG           5
#define RATE_DIVISOR_00_01_REG    6
#define RATE_DIVISOR_02_03_REG    7
#define RATE_DIVISOR_04_05_REG    8
#define RATE_DIVISOR_06_07_REG    9
#define RATE_DIVISOR_08_09_REG   10
#define RATE_DIVISOR_10_11_REG   11
#define RATE_DIVISOR_12_13_REG   12
#define RATE_DIVISOR_14_15_REG   13
#define BUFFER_THRESHOLD_REG     14
#define INPUT_DATA_BUFFER_REG    18
```

EXAMPLE:

```
HANDLE               hDevice;
DWORD                dwTransferSize;
DWORD                dwErrorCode;
SDI_REGISTER_PARAMS  InputRegData;

InputRegData.eSDIRegister    = BUFFER_THRESHOLD_REG;
InputRegData.ulRegisterValue = 0x0003FF00;

if (! DeviceIoControl(hDevice, IOCTL_SDI_WRITE_REGISTER,
                      &InputRegData,  sizeof(SDI_REGISTER_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.4. IOCTL_SDI_REQ_INT_NOTIFY

The IOCTL_SDI_REQ_INT_NOTIFY function will request that the driver notify the application via an event when a specified interrupt occurs. The board only allows one interrupt condition to be enabled at once, therefore only one interrupt condition can be requested for notification. Notification of the following interrupt conditions can be requested:

- Initialization Complete
- Auto Calibration Complete
- Channels Ready
- Buffer Threshold Low To High Transition
- Buffer Threshold High To Low Transition
- Buffer Almost Empty
- Buffer Almost Full

The driver uses interrupts for certain operations in response to requests from the application. If the application requests an interrupt notification and then performs an operation that requires the driver to use the interrupts, the driver may miss the notification interrupt and not notify the application. The application should avoid the use of the following ioctl() operations that require the use of interrupts when waiting on an interrupt notification:

ioctl() Operation	**Interrupt Condition Used**
IOCTL_SDI_SET_INPUT_RANGE	Channels Ready
IOCTL_SDI_SET_INPUT_MODE	Channels Ready
IOCTL_SDI_AUTO_CAL	Auto Calibration Complete
IOCTL_SDI_INITIALIZE	Initialization Complete
IOCTL_SDI_SET_GEN_RATE	Channels Ready
IOCTL_SDI_ASSIGN_GEN_TO_GROUP	Channels Ready
IOCTL_SDI_SET_RATE_DIVISOR	Channels Ready

If the application requests notification of the same interrupt that the driver needs to use, the driver will use it and the application will get a notification.

Input/Output Buffer:

```
<from SDIIoctl.h>

typedef struct _SDI_REGISTER_PARAMS
{
    ULONG  eIntConditions;
    HANDLE hEvent;
} SDI_INT_NOTIFY_PARAMS, *PSDI_INT_NOTIFY_PARAMS;
```

Where hEvent contains a handle of the event to be signaled when the interrupt occurs and eIntConditions contains one of the following conditions to be notified of.

```
#define INIT_COMPLETE            0
#define AUTOCAL_COMPLETE         1
#define CHANNELS_READY           2
#define BUFFER_THRES_LOW_TO_HIGH 3
#define BUFFER_THRES_HIGH_TO_LOW 4
#define BUFFER_ALMOST_EMPTY      5
#define BUFFER_ALMOST_FULL       6
```

EXAMPLE:

```
HANDLE                 hDevice;
DWORD                  dwTransferSize;
DWORD                  dwErrorCode;
HANDLE                 hEvent;
SDI_INT_NOTIFY_PARAMS  IntNotify;

IntNotify.eIntConditions = AUTOCAL_COMPLETE;
hEvent                   = CreateEvent(NULL, FALSE, FALSE, NULL);
IntNotify.hEvent         = hEvent;

if (! DeviceIoControl(hDevice, IOCTL_SDI_REQ_INT_NOTIFY,
                      &IntNotify,  sizeof(SDI_INT_NOTIFY_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}

if (! DeviceIoControl(hDevice, IOCTL_SDI_AUTO_CAL,
                      NULL, 0, NULL, 0,
                      &dwTransferSize, &overlap))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}

// Wait ten seconds for Auto Calibration Interrupt.
if (WaitForSingleObject(hEvent, 10000) == WAIT_OBJECT_0)
{
   printf("Interrupt Occurred\n");
}
else
{
   printf("Timed Out Waiting for Interrupt \n");
}
```

4.4.5. IOCTL_SDI_SET_INPUT_RANGE

The IOCTL_SDI_SET_INPUT_RANGE function will set the SDI Analog Input Range to +/-1.25V, +/-2.5V, +/-5V or +/-10V. This function may be used in overlapped mode because the hardware needs time to settle. If the board is accessed before the settling time, results may be indeterminate. The user should not access this function while data sampling is in progress.

Input/Output Buffer:

```
<from SDIIoctl.h>

// Parameter = ULONG *pInputRange;
//     RANGE: 0-3

#define RANGE_1p25V 0
#define RANGE_2p5V  1
#define RANGE_5V    2
#define RANGE_10V   3
```

EXAMPLE:

```
HANDLE      hDevice;
DWORD       dwTransferSize;
DWORD       dwErrorCode;
ULONG       ulInputRange;
OVERLAPPED  overlap;

overlap.Offset     = 0;
overlap.OffsetHigh = 0;
overlap.hEvent     = CreateEvent(NULL, FALSE, FALSE, NULL);

ulInputRange = RANGE_5V;
if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_INPUT_RANGE,
                      &ulInputRange, sizeof(ULONG),
                      NULL, 0, &dwTransferSize, &overlap))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   status = GetOverlappedResult(hDevice, &overlap,
                                &dwTransferSize, TRUE);
   if (! status)
   {
      printf("GetOverlappedResult Failed\n");
      dwErrorCode = GetLastError();
      ErrorMessage("GetOverlappedResult", dwErrorCode);
   }
}
```

4.4.6. IOCTL_SDI_SET_INPUT_MODE

The IOCTL_SDI_SET_INPUT_MODE function will set the Analog Input Mode to either differential, single-ended, ZERO Test or VREF Test. This function may be used in overlapped mode because the hardware needs time to settle. If the board is accessed before the settling time, results may be indeterminate. The user should not access this function while data sampling is in progress.

Input/Output Buffer:

```
<from SDIIoctl.h>

// Parameter = ULONG *pInputMode;
//     RANGE: 0-3

#define MODE_DIFFERENTIAL 0
#define MODE_SINGLE_ENDED 1
#define MODE_ZERO_TEST    2
#define MODE_VREF_TEST    3
```

EXAMPLE:

```
HANDLE      hDevice;
DWORD       dwTransferSize;
DWORD       dwErrorCode;
ULONG       ulInputMode;
OVERLAPPED  overlap;

overlap.Offset     = 0;
overlap.OffsetHigh = 0;
overlap.hEvent     = CreateEvent(NULL, FALSE, FALSE, NULL);

ulInputMode = MODE_SINGLE_ENDED;
if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_INPUT_MODE,
                      &ulInputMode, sizeof(ULONG),
                      NULL, 0, &dwTransferSize, &overlap))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   status = GetOverlappedResult(hDevice, &overlap,
                                &dwTransferSize, TRUE);
   if (! status)
   {
      printf("GetOverlappedResult Failed\n");
      dwErrorCode = GetLastError();
      ErrorMessage("GetOverlappedResult", dwErrorCode);
   }
}
```

4.4.7. IOCTL_SDI_SET_SW_SYNC

The IOCTL_SDI_SET_SW_SYNC function will initiate a local ADC sync operation. It may also generate an external sync output if the Initiator Mode is selected.

Input/Output Buffer:

```
NONE
```

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;

if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_SW_SYNC,
                      NULL, 0, NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.8. IOCTL_SDI_AUTO_CAL

The IOCTL_SDI_AUTO_CAL function will command the SDI Board to perform an Auto Calibration. Auto Calibration will calibrate all input channels to a single internal voltage reference. Offset and gain error corrections for each channel are implemented with hardware DACs that retain the correction values until power is removed from the board or another calibration is performed. This function may be used in overlapped mode because the hardware needs time to complete the operation. If the board is accessed before the settling time, results may be indeterminate. The user should not access this function while data sampling is in progress.

Input/Output Buffer:

```
NONE
```

EXAMPLE:

```
HANDLE      hDevice;
DWORD       dwTransferSize;
DWORD       dwErrorCode;
OVERLAPPED  overlap;

overlap.Offset     = 0;
overlap.OffsetHigh = 0;
overlap.hEvent     = CreateEvent(NULL, FALSE, FALSE, NULL);

if (! DeviceIoControl(hDevice, IOCTL_SDI_AUTO_CAL,
                      NULL, 0, NULL, 0, &dwTransferSize, &overlap))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   status = GetOverlappedResult(hDevice, &overlap,
                                &dwTransferSize, TRUE);
   if (! status)
   {
      printf("GetOverlappedResult Failed\n");
      dwErrorCode = GetLastError();
      ErrorMessage("GetOverlappedResult", dwErrorCode);
   }
}
```

4.4.9. IOCTL_SDI_INITIALIZE

The IOCTL_SDI_INITIALIZE function will cause the internal logic to be initialized. The following is performed by the initialize command:

- Calibration D/A converters are initialized with midrange values
- Rate Generators adjusted to 125 KSPS
- Rate generator A controls all channels
- Divisor ratios are set to 5 (sample rates to 25 kHz)
- Analog Input Buffer Empty
- Buffer Threshold to 0x0003FFFE
- Input Range set to +/-10V
- Input Mode set to Differential
- Board Control Register Initialized
- Local Interrupt Request Asserted for Initialization Complete

This function may be used in overlapped mode because the hardware needs time to settle. If the board is accessed before the settling time, results may be indeterminate. The user should not access this function while data sampling is in progress.

Input/Output Buffer:

```
NONE
```

EXAMPLE:

```
HANDLE      hDevice;
DWORD       dwTransferSize;
DWORD       dwErrorCode;
OVERLAPPED  overlap;

overlap.Offset     = 0;
overlap.OffsetHigh = 0;
overlap.hEvent     = CreateEvent(NULL, FALSE, FALSE, NULL);

if (! DeviceIoControl(hDevice, IOCTL_SDI_INITIALIZE,
                      NULL, 0, NULL, 0, &dwTransferSize, &overlap))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   status = GetOverlappedResult(hDevice, &overlap,
                                &dwTransferSize, TRUE);
   if (! status)
   {
      printf("GetOverlappedResult Failed\n");
      dwErrorCode = GetLastError();
      ErrorMessage("GetOverlappedResult", dwErrorCode);
   }
}
```

4.4.10. IOCTL_SDI_SET_DATA_FORMAT

The IOCTL_SDI_SET_DATA_FORMAT function sets the data format to either Offset Binary or Two's Complement. The user should not access this function while data sampling is in progress.

ANALOG INPUT LEVEL	OFFSET BINARY	TWO'S COMPLEMENT
Positive Full Scale minus 1 LSB	0xFFFF	0x7FFF
Zero plus 1 LSB	0x8001	0x0001
Zero	0x8000	0x0000
Zero minus 1 LSB	0x7FFF	0xFFFF
Negative Full Scale plus 1 LSB	0x0001	0x8001
Negative Full Scale	0x0000	0x8000

Input/Output Buffer:

```
<from SDIIoctl.h>

// Parameter = ULONG *pDataFormat;
//     RANGE: 0-1

#define FORMAT_TWOS_COMPLEMENT 0
#define FORMAT_OFFSET_BINARY   1
```

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;
ULONG    ulDataFormat;

ulDataFormat = FORMAT_TWOS_COMPLEMENT;
if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_DATA_FORMAT,
                      &ulDataFormat, sizeof(ULONG),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.11. IOCTL_SDI_SET_INITIATOR_MODE

The IOCTL_SDI_SET_INITIATOR_MODE function allows selection of how this board will participate in multiple board synchronization. The board may be selected as an initiator or a target. Selecting the initiator mode will allow other boards to synchronize to this board's sampling clock and synchronization commands. Selecting target mode will allow this board to synchronize to an external sampling clock and synchronization commands. The external source may be another PCI-16SDI board.

Input/Output Buffer:

```
<from SDIIoctl.h>

// Parameter = ULONG *pInitTarget;
//     RANGE: 0-1

#define TARGET_MODE    0
#define INITIATOR_MODE 1
```

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;
ULONG    ulInitTarget;

ulInitTarget = INITIATOR_MODE;
if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_INITIATOR_MODE,
                      &ulInitTarget, sizeof(ULONG),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.12. IOCTL_SDI_SET_BUFFER_THRESHOLD

The IOCTL_SDI_SET_BUFFER_THRESHOLD function will set the threshold that will be used to indicate when a threshold interrupt should occur. The threshold interrupt may be used to determine how much sampling data is contained in the board data buffer. Interrupts may be generated based upon when the amount of data exceeds the threshold or based upon when the amount of data goes below the threshold.

Input/Output Buffer:

```
<from SDIIoctl.h>

// Parameter = ULONG *pulThreshold;
//     RANGE: 0x0 - 0x3FFFF
```

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;
ULONG    ulThreshold;

ulThreshold = 0x0003FF00;
if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_BUFFER_THRESHOLD,
                      &ulThreshold, sizeof(ULONG),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.13. IOCTL_SDI_CLEAR_BUFFER

The IOCTL_SDI_CLEAR_BUFFER function will empty the contents of the sample buffer.

Input/Output Buffer:

NONE

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;

if (! DeviceIoControl(hDevice, IOCTL_SDI_CLEAR_BUFFER,
                      NULL, 0, NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.14. IOCTL_SDI_SET_ACQUIRE_MODE

The IOCTL_SDI_SET_ACQUIRE_MODE function will enable or disable the SDI card from acquiring sample data and storing it in the buffer.

Input/Output Buffer:

```
<from SDIIoctl.h>

// Parameter = ULONG *pAcquireMode;
//     RANGE: 0-1

#define START_ACQUIRE 0
#define STOP_ACQUIRE  1
```

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;
ULONG    ulAcquireMode;

ulAcquireMode = START_ACQUIRE;
if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_ACQUIRE_MODE,
                      &ulAcquireMode, sizeof(ULONG),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.15. IOCTL_SDI_SET_GEN_RATE

The IOCTL_SDI_SET_GEN_RATE function will set the rate for one of the four generators on the SDI Board. This function may be used in overlapped mode because the hardware needs time to settle. If the board is accessed before the settling time, results may be indeterminate. The user should not access this function while data sampling is in progress.

Input/Output Buffer:

```
<from SDIIoctl.h>

// Parameter = GEN_RATE_PARAMS *pRateParams;

// Send in Generator Frequency (floating point in kHz)
// to get the Generator Rate to send as ulNrate in
// IOCTL_SDI_SET_GEN_RATE.
#define Fgen_To_Nrate(Fgen)           \
   ((Fgen < MIN_FGEN) ? MIN_NRATE :  \
    ((Fgen > MAX_FGEN) ? MAX_NRATE : \
     ROUND_TO_ULONG((Fgen * GEN_MULT) - GEN_OFFSET)))

#define GEN_A 0
#define GEN_B 1
#define GEN_C 2
#define GEN_D 3

typedef struct _GEN_RATE_PARAMS
{
   ULONG eGenerator; // RANGE: 0-3
   ULONG ulNrate;    // RANGE: 0-0x1FF
} GEN_RATE_PARAMS, *PGEN_RATE_PARAMS;
```

EXAMPLE:

```
HANDLE            hDevice;
DWORD             dwTransferSize;
DWORD             dwErrorCode;
OVERLAPPED        overlap;
GEN_RATE_PARAMS   GenRate;

overlap.Offset     = 0;
overlap.OffsetHigh = 0;
overlap.hEvent     = CreateEvent(NULL, FALSE, FALSE, NULL);

// Set parameters for 11.264 MHz
GenRate.eGenerator = GEN_A;
GenRate.ulNrate    = Fgen_To_Nrate(11264);  // in KHz

if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_GEN_RATE,
                      &GenRate, sizeof(GEN_RATE_PARAMS),
                      NULL, 0, &dwTransferSize, &overlap))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   status = GetOverlappedResult(hDevice, &overlap,
                                &dwTransferSize, TRUE);
   if (! status)
   {
      printf("GetOverlappedResult Failed\n");
      dwErrorCode = GetLastError();
      ErrorMessage("GetOverlappedResult", dwErrorCode);
   }
}
```

4.4.16. IOCTL_SDI_ASSIGN_GEN_TO_GROUP

The IOCTL_SDI_ASSIGN_GEN_TO_GROUP function will assign a generator to one of the four channel groups. The assigned generator may be one of the four generators (A-D) or an external sample clock. The channels in each group are different based upon how many channels are on the board as follows:

CHANNEL GROUP	16-CHANNEL BOARD	8-CHANNEL BOARD	4-CHANNEL BOARD
0	00, 01, 02, 03	00, 01	00
1	04, 05, 06, 07	02, 03	01
2	08, 09, 10, 11	04, 05	02
3	12, 13, 14, 15	06, 07	03

This function may be used in overlapped mode because the hardware needs time to settle. If the board is accessed before the settling time, results may be indeterminate. The user should not access this function while data sampling is in progress.

Input/Output Buffer:

```
<from SDIIoctl.h>

// Groups
#define GRP_0 0
#define GRP_1 1
#define GRP_2 2
#define GRP_3 3

// Generator Assignments
#define ASN_GEN_A    0
#define ASN_GEN_B    1
#define ASN_GEN_C    2
#define ASN_GEN_D    3
#define ASN_EXT_CLK  4
#define ASN_GEN_NONE 5

typedef struct _GEN_ASSIGN_PARAMS
{
   ULONG eGroup;     // RANGE: 0-3
   ULONG eGenAssign; // RANGE: 0-5
} GEN_ASSIGN_PARAMS, *PGEN_ASSIGN_PARAMS;
```

EXAMPLE:

```
HANDLE            hDevice;
DWORD             dwTransferSize;
DWORD             dwErrorCode;
OVERLAPPED        overlap;
GEN_ASSIGN_PARAMS GenAssign;

overlap.Offset     = 0;
overlap.OffsetHigh = 0;
overlap.hEvent     = CreateEvent(NULL, FALSE, FALSE, NULL);

GenAssign.eGroup     = GRP_0;
GenAssign.eGenAssign = ASN_GEN_A;

if (! DeviceIoControl(hDevice, IOCTL_SDI_ASSIGN_GEN_TO_GROUP,
                      &GenAssign, sizeof(GEN_ASSIGN_PARAMS),
                      NULL, 0, &dwTransferSize, &overlap))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   status = GetOverlappedResult(hDevice, &overlap,
                                &dwTransferSize, TRUE);
   if (! status)
   {
      printf("GetOverlappedResult Failed\n");
      dwErrorCode = GetLastError();
      ErrorMessage("GetOverlappedResult", dwErrorCode);
   }
}
```

4.4.17. IOCTL_SDI_SET_RATE_DIVISOR

The IOCTL_SDI_SET_RATE_DIVISOR function sets the value that divides the assigned rate generator frequency for a specified channel. This function may be used in overlapped mode because the hardware needs time to settle. If the board is accessed before the settling time, results may be indeterminate. The user should not access this function while data sampling is in progress.

Input/Output Buffer:

```
<from SDIIoctl.h>

// Send in Generator Frequency (floating point in kHz)
// and Sample Rate (floating point in kHz) to
// Fgen_and_Fsamp_To_Ndiv to get the Rate Divisor to
// send as ulDivisor in IOCTL_SDI_SET_RATE_DIVISOR.

#define Fgen_and_Fsamp_To_Ndiv(Fgen,Fsamp)       \
   ((Ndiv(Fgen,Fsamp) < MIN_NDIV) ? MIN_NDIV :  \
    ((Ndiv(Fgen,Fsamp) > MAX_NDIV) ? MAX_NDIV : \
   Ndiv(Fgen,Fsamp)))

typedef struct _RATE_DIVISOR_PARAMS
{
   ULONG ulChannel;  // RANGE 0-15
   ULONG ulDivisor;  // RANGE 1-32
} RATE_DIVISOR_PARAMS, *PRATE_DIVISOR_PARAMS;
```

EXAMPLE:

```
HANDLE               hDevice;
DWORD                dwTransferSize;
DWORD                dwErrorCode;
OVERLAPPED           overlap;
RATE_DIVISOR_PARAMS  RateDivisor;

overlap.Offset     = 0;
overlap.OffsetHigh = 0;
overlap.hEvent     = CreateEvent(NULL, FALSE, FALSE, NULL);

// Set Channel 5 divisor for generator frequency of 11.264 MHz and
// sample rate of 44 kHz.
RateDivisor.ulChannel = 5;
RateDivisor.ulDivisor = Fgen_and_Fsamp_To_Ndiv(11264.0, 44.0);

if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_RATE_DIVISOR,
                      &RateDivisor, sizeof(RATE_DIVISOR_PARAMS),
                      NULL, 0, &dwTransferSize, &overlap))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   status = GetOverlappedResult(hDevice, &overlap,
                                &dwTransferSize, TRUE);
   if (! status)
   {
      printf("GetOverlappedResult Failed\n");
      dwErrorCode = GetLastError();
      ErrorMessage("GetOverlappedResult", dwErrorCode);
   }
}
```

4.4.18. IOCTL_SDI_GET_DEVICE_ERROR

The IOCTL_SDI_GET_DEVICE_ERROR function will return the error that occurred on the last call to one of the PCI-16SDI Device Driver entry points. Whenever a driver function is called and it returns an error, this function may be called to determine the cause of the error.

Input/Output Buffer:

```
<from SDIIoctl.h>

// Parameter = ULONG *pulDeviceError;
//     RANGE: 0-8

#define SDI_SUCCESS                    0
#define SDI_INVALID_PARAMETER          1
#define SDI_INVALID_BUFFER_SIZE        2
#define SDI_PIO_TIMEOUT                3
#define SDI_DMA_TIMEOUT                4
#define SDI_IOCTL_TIMEOUT              5
#define SDI_OPERATION_CANCELLED        6
#define SDI_RESOURCE_ALLOCATION_ERROR  7
#define SDI_INVALID_REQUEST            8
```

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;
ULONG    DeviceError;

if ((! DeviceIoControl(hDevice, IOCTL_SDI_GET_DEVICE_ERROR,
                       NULL, 0, &DeviceError,
                       sizeof(ULONG), &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(ULONG)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.19. IOCTL_SDI_READ_PCI_CONFIG

The IOCTL_SDI_READ_PCI_CONFIG function will read all of the PCI Configuration Registers.

Input/Output Buffer:

```
<from SDIIoctl.h>

typedef struct _SDI_READ_PCI_CONFIG_PARAM
{
   ULONG ulDeviceVendorID;
   ULONG ulStatusCommand;
   ULONG ulClassCodeRevisionID;
   ULONG ulBISTHdrTypeLatTimerCacheLineSize;
   ULONG ulRuntimeRegAddr;
   ULONG ulConfigRegAddr;
   ULONG ulPCIBaseAddr2;
   ULONG ulPCIBaseAddr3;
   ULONG ulUnusedBaseAddr1;
   ULONG ulUnusedBaseAddr2;
   ULONG ulCardbusCISPtr;
   ULONG ulSubsystemVendorID;
   ULONG ulPCIRomAddr;
   ULONG ulReserved1;
   ULONG ulReserved2;
   ULONG ulMaxLatMinGntIntPinIntLine;
} SDI_READ_PCI_CONFIG_PARAM, *PSDI_READ_PCI_CONFIG_PARAM;
```

EXAMPLE:

```
HANDLE                     hDevice;
DWORD                      dwTransferSize;
DWORD                      dwErrorCode;
SDI_READ_PCI_CONFIG_PARAM  ConfigRegs;

if ((! DeviceIoControl(hDevice, IOCTL_SDI_READ_PCI_CONFIG, NULL, 0,
                       &ConfigRegs,
                       sizeof(SDI_READ_PCI_CONFIG_PARAM),
                       &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(SDI_READ_PCI_CONFIG_PARAM)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   printf("Device ID/Vendor ID Reg = %08lx\n",
          ConfigRegs.ulDeviceVendorID);
   printf("Status/Command Reg = %08lx\n",
          ConfigRegs.ulStatusCommand);
   printf("Class Code/Revision ID Reg = %08lx\n",
          ConfigRegs.ulClassCodeRevisionID);
   printf("BIST/Header Type/Lat Timer/Cache Line Size Reg = 08lx\n",
          ConfigRegs.ulBISTHdrTypeLatTimerCacheLineSize);
   printf("Runtime Register Address Reg = %08lx\n",
          ConfigRegs.ulRuntimeRegAddr);
   printf("Config Register Address Reg = %08lx\n",
          ConfigRegs.ulConfigRegAddr);
   printf("PCI Base Address 2 Reg = %08lx\n",
          ConfigRegs.ulPCIBaseAddr2);
   printf("PCI Base Address 3 Reg = %08lx\n",
          ConfigRegs.ulPCIBaseAddr3);
   printf("Unused Base Address 1 Reg = %08lx\n",
          ConfigRegs.ulUnusedBaseAddr1);
   printf("Unused Base Address 2 Reg = %08lx\n",
          ConfigRegs.ulUnusedBaseAddr2);
   printf("Cardbus CIS Pointer Reg = %08lx\n",
          ConfigRegs.ulCardbusCISPtr);
   printf("Subsystem ID/Vendor ID Reg = %08lx\n",
          ConfigRegs.ulSubsystemVendorID);
   printf("PCI Rom Address Reg = %08lx\n",
          ConfigRegs.ulPCIRomAddr);
   printf("Reserved 1 Reg = %08lx\n",
          ConfigRegs.ulReserved1);
   printf("Reserved 2 Reg = %08lx\n",
          ConfigRegs.ulReserved2);
   printf("Max Lat/Min Gnt/Int Pin/Int Line Reg = %08lx\n",
          ConfigRegs.ulMaxLatMinGntIntPinIntLine);
}
```

4.4.20. IOCTL_SDI_READ_LOCAL_CONFIG

The IOCTL_SDI_READ_LOCAL_CONFIG function will read and return the local configuration registers.

Input/Output Buffer:

```
<from SDIIoctl.h>

typedef struct _CONFIG_REGS_PARAMS
{
    /*** Local Configuration Registers ***/
    ULONG   ulPciLocRange0;
    ULONG   ulPciLocRemap0;
    ULONG   ulModeArb;
    ULONG   ulEndianDescr;
    ULONG   ulPciLERomRange;
    ULONG   ulPciLERomRemap;
    ULONG   ulPciLBRegDescr0;
    ULONG   ulLocPciRange;
    ULONG   ulLocPciMemBase;
    ULONG   ulLocPciIOBase;
    ULONG   ulLocPciRemap;
    ULONG   ulLocPciConfig;
    ULONG   ulOutPostQIntStatus;
    ULONG   ulOutPostQIntMask;
    UCHAR   uchReserved1[8];

    /*** Shared Run Time Registers ***/
    ULONG   ulMailbox[8];
    ULONG   ulPciLocDoorBell;
    ULONG   ulLocPciDoorBell;
    ULONG   ulIntCntrlStat;
    ULONG   ulRunTimeCntrl;
    ULONG   ulDeviceVendorID;
    ULONG   ulRevisionID;
    ULONG   ulMailboxReg0;
    ULONG   ulMailboxReg1;

    /*** Local DMA Registers ***/
    ULONG   ulDMAMode0;
    ULONG   ulDMAPCIAddress0;
    ULONG   ulDMALocalAddress0;
    ULONG   ulDMAByteCount0;
    ULONG   ulDMADescriptorPtr0;
    ULONG   ulDMAMode1;
    ULONG   ulDMAPCIAddress1;
    ULONG   ulDMALocalAddress1;
    ULONG   ulDMAByteCount1;
    ULONG   ulDMADescriptorPtr1;
    ULONG   ulDMACmdStatus;
    ULONG   ulDMAArbitration;
    ULONG   ulDMAThreshold;
    UCHAR   uchReserved3[12];

    /*** Messaging Queue Registers ***/
    ULONG   ulMsgUnitCfg;
    ULONG   ulQBaseAddr;
    ULONG   ulInFreeHeadPtr;
    ULONG   ulInFreeTailPtr;
    ULONG   ulInPostHeadPtr;
    ULONG   ulInPostTailPtr;
    ULONG   ulOutFreeHeadPtr;
    ULONG   ulOutFreeTailPtr;
    ULONG   ulOutPostHeadPtr;
    ULONG   ulOutPostTailPtr;
    ULONG   ulQStatusCtrl;
    UCHAR   uchReserved4[4];
    ULONG   ulPciLocRange1;
    ULONG   ulPciLocRemap1;
    ULONG   ulPciLBRegDescr1;
} CONFIG_REGS, *PCONFIG_REGS;
```

EXAMPLE:

```
HANDLE      hDevice;
DWORD       dwTransferSize;
DWORD       dwErrorCode;
CONFIG_REGS LocalConfigRegs;

if ((! DeviceIoControl(hDevice, IOCTL_SDI_READ_LOCAL_CONFIG, NULL,
                       0, &LocalConfigRegs, sizeof(CONFIG_REGS),
                       &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(CONFIG_REGS)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   printf("\n");
   printf("    LOCAL CONFIGURATION REGISTERS\n");
   printf("Range for PCI to Local 0 Reg    = %08lx\n",
          LocalConfigRegs.ulPciLocRange0);
   printf("Remap for PCI to Local 0 Reg    = %08lx\n",
          LocalConfigRegs.ulPciLocRemap0);
   printf("Mode Arbitration Reg            = %08lx\n",
          LocalConfigRegs.ulModeArb);
   printf("Big/Little Endian Descr. Reg    = %08lx\n",
          LocalConfigRegs.ulEndianDescr);
```

```
printf("Range for PCI to Local Reg      = %08lx\n",
       LocalConfigRegs.ulPciLERomRange);
printf("Remap for PCI to Local Reg      = %08lx\n",
       LocalConfigRegs.ulPciLERomRemap);
printf("Bus Region Descriptions for Reg = %08lx\n",
       LocalConfigRegs.ulPciLBRegDescr0);
printf("Range for Local to PCI Reg      = %08lx\n",
       LocalConfigRegs.ulLocPciRange);
printf("Base Addr for Local to PCI Reg  = %08lx\n",
       LocalConfigRegs.ulLocPciMemBase);
printf("Base Addr for Local to PCI Reg  = %08lx\n",
       LocalConfigRegs.ulLocPciIOBase);
printf("Remap for Local to PCI Reg      = %08lx\n",
       LocalConfigRegs.ulLocPciRemap);
printf("PCI Config Address Reg for Reg  = %08lx\n",
       LocalConfigRegs.ulLocPciConfig);
printf("Range for PCI to Local 1 Reg    = %08lx\n",
       LocalConfigRegs.ulPciLocRange1);
printf("Remap for PCI to Local 1 Reg    = %08lx\n",
       LocalConfigRegs.ulPciLocRemap1);
printf("Bus Region Descriptor Reg       = %08lx\n",
       LocalConfigRegs.ulPciLBRegDescr1);

printf("    RUNTIME REGISTERS\n");
printf("Mailbox Register 0          = %08lx\n",
       LocalConfigRegs.ulMailbox[0]);
printf("Mailbox Register 1          = %08lx\n",
       LocalConfigRegs.ulMailbox[1]);
printf("Mailbox Register 2          = %08lx\n",
       LocalConfigRegs.ulMailbox[2]);
printf("Mailbox Register 3          = %08lx\n",
       LocalConfigRegs.ulMailbox[3]);
printf("Mailbox Register 4          = %08lx\n",
       LocalConfigRegs.ulMailbox[4]);
printf("Mailbox Register 5          = %08lx\n",
       LocalConfigRegs.ulMailbox[5]);
printf("Mailbox Register 6          = %08lx\n",
       LocalConfigRegs.ulMailbox[6]);
printf("Mailbox Register 7          = %08lx\n",
       LocalConfigRegs.ulMailbox[7]);
printf("PCI to Local Doorbell Reg   = %08lx\n",
       LocalConfigRegs.ulPciLocDoorBell);
printf("Local to PCI Doorbell Reg   = %08lx\n",
       LocalConfigRegs.ulLocPciDoorBell);
printf("Interrupt Control/Status    = %08lx\n",
       LocalConfigRegs.ulIntCntrlStat);
printf("EEPROM Control, PCI Command = %08lx\n",
       LocalConfigRegs.ulRunTimeCntrl);
printf("Device ID                   = %08lx\n",
       LocalConfigRegs.ulDeviceVendorID);
printf("Revision ID                 = %08lx\n",
       LocalConfigRegs.ulRevisionID);
printf("Mailbox Register 0          = %08lx\n",
       LocalConfigRegs.ulMailboxReg0);
```

```
printf("Mailbox Register 1          = %08lx\n",
       LocalConfigRegs.ulMailboxReg1);

printf("    DMA REGISTERS\n");
printf("dma channel 0 mode Reg           = %08lx\n",
       LocalConfigRegs.ulDMAMode0);
printf("dma channel 0 pci address Reg    = %08lx\n",
       LocalConfigRegs.ulDMAPCIAddress0);
printf("dma channel 0 local address Reg  = %08lx\n",
       LocalConfigRegs.ulDMALocalAddress0);
printf("dma channel 0 transfer byte Reg  = %08lx\n",
       LocalConfigRegs.ulDMAByteCount0);
printf("dma channel 0 descriptor Reg     = %08lx\n",
       LocalConfigRegs.ulDMADescriptorPtr0);
printf("dma channel 1 mode Reg           = %08lx\n",
       LocalConfigRegs.ulDMAMode1);
printf("dma channel 1 pci address Reg    = %08lx\n",
       LocalConfigRegs.ulDMAPCIAddress1);
printf("dma channel 1 local address Reg  = %08lx\n",
       LocalConfigRegs.ulDMALocalAddress1);
printf("dma channel 1 transfer byte Reg  = %08lx\n",
       LocalConfigRegs.ulDMAByteCount1);
printf("dma channel 1 descriptor Reg     = %08lx\n",
       LocalConfigRegs.ulDMADescriptorPtr1);
printf("dma command/status registers Reg = %08lx\n",
       LocalConfigRegs.ulDMACmdStatus);
printf("dma arbitration register Reg     = %08lx\n",
       LocalConfigRegs.ulDMAArbitration);
printf("dma threshold register Reg       = %08lx\n",
       LocalConfigRegs.ulDMAThreshold);

printf("MESSAGING QUEUE REGISTERS\n");
printf("outbound post queue Int Status Reg = %08lx\n",
       LocalConfigRegs.ulOutPostQIntStatus);
printf("outbound post queue Int Mask Reg   = %08lx\n",
       LocalConfigRegs.ulOutPostQIntMask);
printf("Mailbox Reg 0                      = %08lx\n",
       LocalConfigRegs.ulMailbox[0]);
printf("Mailbox Reg 1                      = %08lx\n",
       LocalConfigRegs.ulMailbox[1]);
printf("messaging unit configuration Reg   = %08lx\n",
       LocalConfigRegs.ulMsgUnitCfg);
printf("queue base address register Reg    = %08lx\n",
       LocalConfigRegs.ulQBaseAddr);
printf("inbound free head pointer Reg      = %08lx\n",
       LocalConfigRegs.ulInFreeHeadPtr);
printf("inbound free tail pointer Reg      = %08lx\n",
       LocalConfigRegs.ulInFreeTailPtr);
printf("inbound post head pointer Reg      = %08lx\n",
       LocalConfigRegs.ulInPostHeadPtr);
printf("inbound post tail pointer Reg      = %08lx\n",
       LocalConfigRegs.ulInPostTailPtr);
printf("inbound free head pointer Reg      = %08lx\n",
       LocalConfigRegs.ulOutFreeHeadPtr);
```

```
      printf("inbound free tail pointer Reg      = %08lx\n",
             LocalConfigRegs.ulOutFreeTailPtr);
      printf("inbound post head pointer Reg      = %08lx\n",
             LocalConfigRegs.ulOutPostHeadPtr);
      printf("inbound post tail pointer Reg      = %08lx\n",
             LocalConfigRegs.ulOutPostTailPtr);
      printf("queue status/control Reg           = %08lx\n",
             LocalConfigRegs.ulQStatusCtrl);
   }
```

4.4.21. IOCTL_SDI_WRITE_PCI_CONFIG_REG

The IOCTL_SDI_WRITE_PCI_CONFIG_REG function will write a value to one of the PCI Configuration Registers. The user should be very careful modifying values of certain registers. The following registers should not be changed:

- PCI_MEM_BASE_ADDR
- PCI_IO_BASE_ADDR
- PCI_BASE_ADDR_0
- PCI_BASE_ADDR_1
- PCI_BASE_ADDR_LOC_ROM

Input/Output Buffer:

```
<from SDIIoctl.h>

typedef struct _SDI_REGISTER_PARAMS
{
    ULONG eSDIRegister;
    ULONG ulRegisterValue;
} SDI_REGISTER_PARAMS, *PSDI_REGISTER_PARAMS;

Where ulRegisterValue contains the value to be written to the
register and eSDIRegister is one of the following:

#define STATUS_COMMAND                1
#define BIST_HDR_TYPE_LAT_CACHE_SIZE  3
#define PCI_MEM_BASE_ADDR             4
#define PCI_IO_BASE_ADDR              5
#define PCI_BASE_ADDR_0               6
#define PCI_BASE_ADDR_1               7
#define PCI_BASE_ADDR_LOC_ROM        12
#define LAT_GNT_INT_PIN_LINE         15
```

EXAMPLE:

```
HANDLE               hDevice;
DWORD                dwTransferSize;
DWORD                dwErrorCode;
SDI_REGISTER_PARAMS  InputRegData;

InputRegData.eSDIRegister    = STATUS_COMMAND;
InputRegData.ulRegisterValue = 0x12345678;

if (! DeviceIoControl(hDevice, IOCTL_SDI_WRITE_PCI_CONFIG_REG,
                      &InputRegData,  sizeof(SDI_REGISTER_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.22. IOCTL_SDI_WRITE_LOCAL_CONFIG_REG

The IOCTL_SDI_WRITE_LOCAL_CONFIG_REG function will write a value to one of the Local Configuration Registers. The user should be very careful modifying values of certain registers. All of the DMA Registers should not be changed while a data transfer is in progress. The following registers should not be changed:

- All Local Configuration Registers

Input/Output Buffer:

```
<from SDIIoctl.h>

typedef struct _SDI_REGISTER_PARAMS
{
    ULONG eSDIRegister;
    ULONG ulRegisterValue;
} SDI_REGISTER_PARAMS, *PSDI_REGISTER_PARAMS;

Where ulRegisterValue contains the value to be written to the
register and eSDIRegister is one of the following:

/*** DMA Registers ***/
#define DMA_CH_0_MODE                 32
#define DMA_CH_0_PCI_ADDR             33
#define DMA_CH_0_LOCAL_ADDR           34
#define DMA_CH_0_TRANS_BYTE_CNT       35
#define DMA_CH_0_DESC_PTR             36
#define DMA_CH_1_MODE                 37
#define DMA_CH_1_PCI_ADDR             38
#define DMA_CH_1_LOCAL_ADDR           39
#define DMA_CH_1_TRANS_BYTE_CNT       40
#define DMA_CH_1_DESC_PTR             41
#define DMA_CMD_STATUS                42
#define DMA_MODE_ARB_REG              43
#define DMA_THRESHOLD_REG             44

/*** Local Configuration Registers. ***/
#define PCI_TO_LOC_ADDR_0_RNG          0
#define LOC_BASE_ADDR_REMAP_0          1
#define MODE_ARBITRATION               2
#define BIG_LITTLE_ENDIAN_DESC         3
#define PCI_TO_LOC_ROM_RNG             4
#define LOC_BASE_ADDR_REMAP_EXP_ROM    5
#define BUS_REG_DESC_0_FOR_PCI_LOC     6
#define DIR_MASTER_TO_PCI_RNG          7
#define LOC_ADDR_FOR_DIR_MASTER_MEM    8
#define LOC_ADDR_FOR_DIR_MASTER_IO     9
#define PCI_ADDR_REMAP_DIR_MASTER     10
```

```
#define PCI_CFG_ADDR_DIR_MASTER_IO   11
#define PCI_TO_LOC_ADDR_1_RNG        92
#define LOC_BASE_ADDR_REMAP_1        93
#define BUS_REG_DESC_1_FOR_PCI_LOC   94

/*** Run Time Registers ***/
#define MAILBOX_REGISTER_0           16
#define MAILBOX_REGISTER_1           17
#define MAILBOX_REGISTER_2           18
#define MAILBOX_REGISTER_3           19
#define MAILBOX_REGISTER_4           20
#define MAILBOX_REGISTER_5           21
#define MAILBOX_REGISTER_6           22
#define MAILBOX_REGISTER_7           23
#define PCI_TO_LOC_DOORBELL          24
#define LOC_TO_PCI_DOORBELL          25
#define INT_CTRL_STATUS              26
#define PROM_CTRL_CMD_CODES_CTRL     27
#define DEVICE_ID_VENDOR_ID          28
#define REVISION_ID                  29
#define MAILBOX_REG_0                30
#define MAILBOX_REG_1                31

/*** Messaging Queue Registers ***/
#define OUT_POST_Q_INT_STATUS        12
#define OUT_POST_Q_INT_MASK          13
#define IN_Q_PORT                    16
#define OUT_Q_PORT                   17
#define MSG_UNIT_CONFIG              48
#define Q_BASE_ADDR                  49
#define IN_FREE_HEAD_PTR             50
#define IN_FREE_TAIL_PTR             51
#define IN_POST_HEAD_PTR             52
#define IN_POST_TAIL_PTR             53
#define OUT_FREE_HEAD_PTR            54
#define OUT_FREE_TAIL_PTR            55
#define OUT_POST_HEAD_PTR            56
#define OUT_POST_TAIL_PTR            57
#define Q_STATUS_CTRL_REG            58
```

EXAMPLE:

```
HANDLE               hDevice;
DWORD                dwTransferSize;
DWORD                dwErrorCode;
SDI_REGISTER_PARAMS  InputRegData;

InputRegData.eSDIRegister    = MAILBOX_REGISTER_0;
InputRegData.ulRegisterValue = 0x99999999;

if (! DeviceIoControl(hDevice, IOCTL_SDI_WRITE_LOCAL_CONFIG_REG,
                      &InputRegData,  sizeof(SDI_REGISTER_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.23. IOCTL_SDI_SET_TIMEOUT

The IOCTL_SDI_SET_TIMEOUT function will set the timeout that the driver uses for ending read operations when enough data is not available. The time is specified in seconds. A –1 will indicate no timeout. The default time set when the driver is initialized is 10 seconds.

Input/Output Buffer:

```
<from SDIIoctl.h>

// Parameter = ULONG *pulTimeout;
//     RANGE: 0x0-0xFFFFFFFF, 0xFFFFFFFF=No Timeout
```

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;
ULONG    ulTimeout;

/* Set the time to never timeout. */
ulTimeout = -1L;
if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_TIMEOUT,
                      &ulTimeout, sizeof(ULONG),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.4.24. IOCTL_SDI_SET_DMA_ENABLE

The IOCTL_SDI_SET_DMA_ENABLE function will set the enable for DMA operations. If DMA is enabled the driver will perform DMA reads when read operations are requested. If DMA is not enabled the driver will just perform Programmed I/O transfers when read operations are requested. DMA Operations allow the CPU to be freed up for application use while the data is being transferred.

Input/Output Buffer:

```
// Parameter = BOOLEAN *pbDMAEnable;
//     RANGE: FALSE-TRUE
```

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;
BOOLEAN  bDMAEnable;

/* Enable DMA. */
bDMAEnable = TRUE;
if (! DeviceIoControl(hDevice, IOCTL_SDI_SET_DMA_ENABLE,
                      &bDMAEnable, sizeof(BOOLEAN),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

5. Driver Installation

This section will describe the procedure for installing the PCI-16SDI Windows NT Driver. The following is the installation procedure:

? Insert the installation floppy disk into a 3 ½" floppy drive
? Click on Run from the Start menu
? Type in "**A:\Setup.exe**" and Click the OK button in the Run Dialog Box
? Follow the instructions on the screen
? Either allow the install program to reboot the computer or reboot it manually so the driver will automatically be installed

The following files are installed in the selected directory by the install program:

? **SDIDriver.sys** – a copy of the driver file that is installed in the O/S **drivers** directory
? **SDIIoctl.h** – the 'C' header file that contains the driver access constants and structures. This file should be **#include**'d in application code where the driver is accessed.
? **SDITest.c** – a 'C' source file containing an example program that shows how to access each of the driver entry points
? **SDITest.exe** – compiled version of **SDITest.c** that will allow menu access to each of driver entry points
? **readme.txt** – a file containing the latest information on the driver
? **Uninst.isu** – a file containing information that allows the driver to be uninstalled

The driver is installed to be automatically started up when the computer is booted.

6. Test Program

This section will describe how to execute the test program installed with the PCI-16SDI driver. The following is the procedure for executing the test program:

? Start up a command prompt window
? Change to the directory where the driver was installed
? Type “**SDITest \\.\sdix**”, where x is the number of the PCI-16SDI board to access, starting with 1 for the first board