SENSORAY

Instruction Manual

Smart A/D Driver for Windows 95/98/NT

MANUAL-WIN95/98/NT SMART A/D DRIVER-10/07/1999

TABLE OF CONTENTS

1. Introduction

1.1 Scope

This document describes the installation and use of the Sensoray Windows 95/98/NT driver for Smart A/D™ boards.

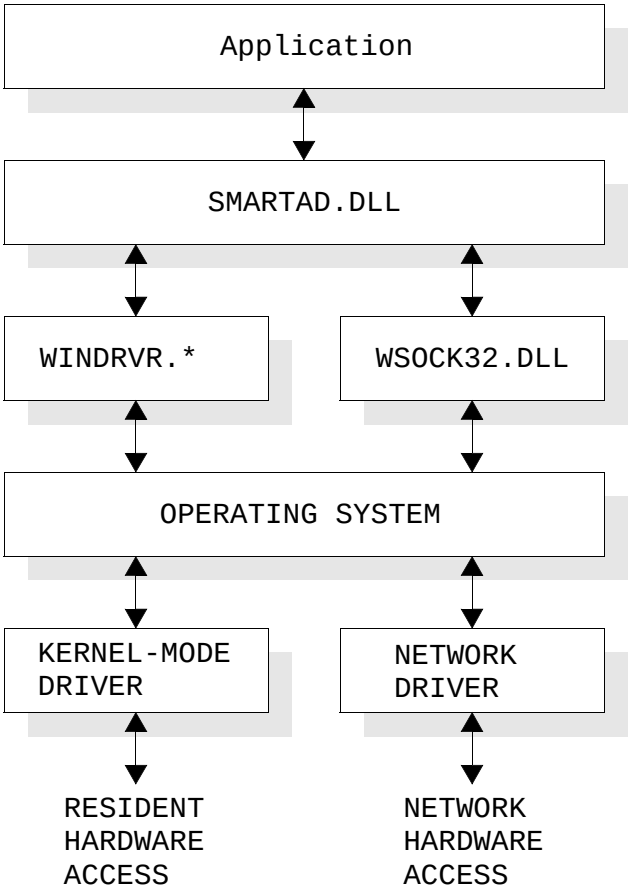
1.2 Description

The Smart A/D™ driver is a set of executable software modules that will interface one or more Sensoray Smart A/D™ boards to a Windows application program of your design. Application programs may be developed in any popular Windows development environment, including Visual C++, Visual Basic, Delphi, etc.

Up to sixteen Sensoray Smart A/D™ boards may be concurrently supported by the driver. The driver simultaneously supports any combination of 8-, 16- and 32-channel Smart A/D™ boards, as well as any combination of Network or Resident (non-network) Smart A/D™ product families.

1.3 Block Diagram

The Smart A/D™ driver consists of several software components that serve as an interface between the application program and the operating system. The following diagram illustrates the relationship between the various software components.



2. Installation

2.1 Software Components

Various software components must be installed on the target system to ensure the proper functioning of the Smart A/D™ driver. The components that are required, and their locations in your system, are a function of the operating system and Smart A/D™ product family that you will be using.

The following table specifies which components are required and where they must be located for various system configurations.

Operating System	Smart A/D Family	Component	
		Filename	Directory
NT	Network	SMARTAD.DLL	SYSTEM32
		WSOCK32.DLL	
	Resident	SMARTAD.DLL	
		WINDRVR.SYS	SYSTEM32\DRIVERS
95/98	Network	SMARTAD.DLL	SYSTEM
		WSOCK32.DLL	
	Resident	SMARTAD.DLL	
		WINDRVR.VXD	SYSTEM\VMM32

For example, if you are running an NT platform with a model 619 PCI Smart A/D™ board, you must install `SMARTAD.DLL` in the `SYSTEM32` directory, and `WINDRVR.SYS` in the `SYSTEM32\DRIVERS` directory.

2.2 Installation Procedure

1. Locate all required software components in their appropriate directories as detailed in Section 2.1.
2. If you are using any resident (non-network) Smart A/D™ product, perform the following steps:
   a. If you are running NT, make sure that you have administrative privileges before proceeding to the next step.
   b. Execute this command line (exclude quotes): "`WDREG.EXE INSTALL`"
   c. Shut down Windows and remove system power.
   d. Install the Smart A/D™ board or boards into the backplane.
   e. Apply power and restart your system.
   f. If you have installed a PCI or CompactPCI board, Windows may, during bootup, detect the board as a new "PCI Communication Device" and ask you if you wish to install it. If this happens, **do not allow Windows to choose the driver for the Smart A/D board**. If you permit Windows to automatically install the driver for your new board, it will not work properly and Windows may corrupt the configuration of other PNP devices in your system. Instead, click on "Have Disk" to manually select a driver, and specify `SmartAD.INI` as the driver to be used. This file is provided on the distribution diskette supplied with the Smart A/D™ board.
3. In some systems it may be necessary to reboot again before the system will "settle" into its new configuration.

3. Driver Functions

3.1 Board Handles

Each board is assigned a board number, which is referenced in this document as a "handle." A handle is the logical address of a board. Most driver functions include the board handle as a parameter so that driver calls will be directed to a specific board. Since the driver supports up to 16 Smart A/D™ boards, valid handles may have any numerical value in the range 0 to 15.

Board handles are not OS-allocated handles in the traditional Windows sense, but rather are integer values that are assigned by the application program. When a board is first declared to the driver by the application program, any unused valid handle may be specified. Once a handle has been assigned to a board, it must not be used by any other board.

3.2 Physical Addressing

In addition to the board handle, which is the "logical" address for a board, each board also has a physical address. In the case of direct I/O mapped products such as STDbus and ISA boards, the physical address is simply the base I/O address of the board. Networking Smart A/D™ products, on the other hand, use the IP address as a physical address.

In order to provide a common physical address format for all Smart A/D™ products, a board's physical address is always specified as a 32-bit, unsigned integer value, regardless of the Smart A/D™ product family. The following table describes the characteristics of the physical addresses employed by the driver for the different Smart A/D™ product families:

Product Family	Physical Address Value
PCI / CPCI	On PCI and CompactPCI models (Model numbers 61x and 71x), the address is a derived value that indicates both the slot number and bus number that the board resides in. The high word (16 bits) of the address value represents the bus number and the low word represents the slot number within that bus. For example, the address `0x0002000A` indicates that the board is located in bus number 2, slot 10.
Network	On Networking models (Model numbers 251x), the address value is the board's IP (Internet Protocol, v.4) address. The address is always specified in host byte order (versus network byte order). The IP address value is established as described in the hardware documentation for the specific Network Smart A/D™ product. Note that the software driver includes utility functions that will translate between this binary address form and the more user-friendly "dot address" format (i.e., "`124.105.73.116`").
All Others	On all other models—which are direct I/O mapped products—the address value is the board's base I/O address as established by programming shunts or switches on the board.

Four of the driver functions make use of physical board addresses:

- `SetBoardAttr()` declares a board to the driver in order to establish a connection to the board.
- `GetBoardAdrs()` returns the physical address of a connected board.
- `InetAtoN()` converts an IP address from "dot address" format to binary.
- `InetNtoA()` converts an IP address from binary to "dot address" format.

3.3 Required Function Calls

Some driver functions are used universally in all applications, while other driver functions may or may not be used depending on the sensor types in use and other considerations. All application programs must, as a minimum, perform the following steps for each Smart A/D™ board:

1. Call `SetBoardAttr()` to declare the Smart A/D™ board type and physical address.
2. In systems that have multiple PCI or CompactPCI Smart A/D™ boards, call `GetBoardAdrs()` to get the board's physical address.
3. Call `ResetBoard()` to initialize the board to a known state.

4. Call `GetFaultFlags()` to verify that the board is properly initialized, fault-free and registered with the driver.
5. For each channel, call `SetSensorType()` to declare the sensor type to be interfaced.

3.4 Network Utilities

This section is applicable if you are using Networking Smart A/D boards.

Network IP addresses are typically specified and manipulated at the user-interface level in *dotted decimal* notation. For example, the text string value "`135.135.135.127`" is much easier to use by a human operator than is `2273806207`, the equivalent numerical value. As evidence of this, consider the classic "Ping" utility, which accepts IP addresses only in dotted decimal notation for ease of use.

In order to provide a standardized format for all Smart A/D™ boards, physical addresses that are passed to or from the driver must be specified as integer values. To support the need for both the user-friendly dotted decimal address and the driver-required integer address formats, the driver includes two utilities that translate between these IP address representations, `InetAtoN()` and `InetNtoA()`, which are described in this section. These are the only driver functions that do not require a board handle parameter. A board handle is not required because these functions do not reference a specific board.

3.4.1 InetAtoN()

Function: Converts an IP "dot address" text string into a host byte-ordered integer IP address. This function can be used to convert an IP address from the "user-friendly" dot address form to the integer value that is required by many of the other driver functions.

Prototype: `long InetAtoN( char *dotadrs );`

Parameter	Type	Description
`dotadrs`	`char*`	Address of a buffer that contains an IP address expressed in the "dot address" format.
return value	`long`	Equivalent host byte-ordered integer IP address.

Example:

```
//////////////////////////////////////////////////////////////////////////
// Convert the IP Address 135.135.135.24 to its equivalent numerical value.
//////////////////////////////////////////////////////////////////////////

char dotadrs[] = "135.135.135.24";
long ipadrs = InetAtoN( dotadrs );
```

3.4.2 InetNtoA()

Function: Converts a host byte-ordered integer IP address to "dot address" form. This function can be used to convert an IP address from the integer form used by many driver functions to a "user-friendly" dot address string.

Prototype: `void InetNtoA( char *dotadrs, long ipadrs );`

Parameter	Type	Description
`dotadrs`	`char*`	Address of a buffer that will receive the "dot address" string.
`ipadrs`	`long`	Host byte-ordered binary IP address.

Example:

```
//////////////////////////////////////////////////////////////////////////
// Get the IP address of board number 0 in dot address form.
//////////////////////////////////////////////////////////////////////////

char dotadrs[16];
long ipadrs = InetNtoA( dotadrs, GetBoardAdrs(0) );
```

3.5 Board Configuration Functions

3.5.1 SetBoardAttr()

Function: Registers/unregisters (synonomous with "connects/disconnects") a Smart A/D™ board to/from the driver.

A board is registered by declaring to the driver the board's model number, and, in the case of non-PCI models, the board's physical address. **Each Smart A/D board must be registered by this function before calling any other driver functions that reference that board.**

Unregister a board by calling this function with the `modelnum` parameter set to zero. This releases the board handle and frees all resources that were used by the associated board. **All connected boards must be unregistered when the application terminates.** This can be accomplished as follows:

1. If the application is dynamically linked to the driver, call `FreeLibrary()` when the application terminates.
2. If the application is statically linked to the driver, Windows will automatically unregister all boards.

Prototype: `long SetBoardAttr( long hBD, long modelnum, long address );`

Parameter	Type	Description
`hBD`	`long`	Board handle. Use any value between 0 and 15, inclusive, but do not use a value that has already been used for another Smart A/D™ board.
`modelnum`	`long`	To register a Smart A/D™ board with the driver, specify the board's model number. A previously registered board may be unregistered by setting this to zero.
`address`	`long`	Physical address of the Smart A/D™ board. See "Physical Addressing" on page 3 for details of physical address formats. **PCI/CPCI models only:** If you know a board's address, you may register the board by specifying its address, just as you would with any other board type. Altermately, you may set the address to zero; this causes the driver to seek the next available unregistered PCI/CPCI Smart A/D™ board in your system and connect it to the specified board handle. You may then determine the physical address of the connected PCI/CPCI board by calling the `GetBoardAdrs()` function.
return value	`long`	A non-zero value is returned upon successful registration.

Example:
```
////////////////////////////////////////////////////////////////////////////
// Declare board number 0 to be a Model 619 PCI Smart A/D board, address unknown.
////////////////////////////////////////////////////////////////////////////

long successful = SetBoardAttr( 0, 619, 0 );
long board0adrs = GetBoardAdrs( 0 );  // Get board's bus/slot number.
```

Example:
```
////////////////////////////////////////////////////////////////////////////
// Declare board number 2 to be a Model 618 PCI Smart A/D board which resides at
// bus number 0, slot number 11.
////////////////////////////////////////////////////////////////////////////

long board2adrs = 0x0000000B;
long successful = SetBoardAttr( 0, 618, board2adrs );
```

Example:
```
////////////////////////////////////////////////////////////////////////
// Declare board number 3 to be a Sensoray Model 7419 STDbus Smart A/D board
// which resides at base I/O address F2B2 hex.
////////////////////////////////////////////////////////////////////////

long baseport = 0xF2B2L;
long successful = SetBoardAttr( 3, 7419, baseport );
```

Example:

```
////////////////////////////////////////////////////////////////////////
// Declare board number 15 to be a Sensoray Model 2518 Network Smart A/D board
// which resides at IP address 135.135.135.127.
////////////////////////////////////////////////////////////////////////

char dotadrs[] = "135.135.135.127";
long successful = SetBoardAttr( 15, 2518, InetAtoN( dotadrs ) );
```

3.5.2 ResetBoard()

Function: Invokes a soft reset on the specified Smart A/D™ board. The board is forced to its default condition and is then tested for a persistent fault condition. The board Fault indicator will momentarily light when this function is called. After calling `ResetBoard()`, your application should call `GetFaultFlags()` to verify that the board does not have a persistent fault.

Note: in the case of IoNet™ Smart A/D™ products, this function resets only the Smart A/D™ portion of the product; the IoNet™ core is reset only by power-up and network inactivity timeout events.

Prototype: `void ResetBoard( long hBD );`

Parameter	Type	Description
hBD	long	Board handle.

Example:

```
////////////////////////////////////////////////////////////////////////
// Reset board number 5 and determine if any faults are pending.
////////////////////////////////////////////////////////////////////////

#define BOARDNUM 5

ResetBoard( BOARDNUM );
BOOL FaultsArePending = ( GetFaultFlags( BOARDNUM ) != 0 );
```

3.5.3 SetIoNetTimeout()

This function is applicable if you are using Networking Smart A/D boards.

Function: Changes the Network Inactivity Timeout Interval (NITI) to the specified value.

All IoNet™ products are designed to execute a watchdog hardware reset if no directed network packets have been received within the NITI. Upon reset, the NITI assumes a default value as specified in your hardware documentation. This function can be used to change the NITI to accommodate applications which are not compatible with the default NITI value.

Note: the specified time interval is a guaranteed minimum value only; the actual time interval is typically about 1.6 seconds longer than the specified value.

Prototype: `void SetIoNetTimeout( long hBD, long msec );`

Parameter	Type	Description
hBD	long	Board handle.
msec	long	IoNet™ Network Inactivity Timeout Interval, specified in milliseconds. Values from 0x00000001L to 0x00FFFFFFL are legal, corresponding to minimum NITI time intervals ranging from 1ms to approximately 4.7 hours.

Example:

```
////////////////////////////////////////////////////////////////////////
// Set the Network Inactivity Timeout Interval of board number 2 to 60 seconds.
////////////////////////////////////////////////////////////////////////

SetIoNetTimeout( 2, 60000 );
```

3.5.4 SetHiSpeedMode()

Function: Switches the board to its High Scan Rate/ Low Resolution mode. This should be done before declaring sensor types, but after any board resets. After calling this function, the only way to return to the default Low Scan Rate/High Resolution mode, is to call the `ResetBoard()` function.

Prototype: `void SetHiSpeedMode( long hBD );`

Parameter	Type	Description
hBD	long	Board handle.

Example:

```
//////////////////////////////////////////////////////////////////////
// Switch board number 2 to the high speed, low resolution mode.
//////////////////////////////////////////////////////////////////////

SetHiSpeedMode( 2 );
```

3.5.5 SetReject50Hz()

Function: Switch the board to its 50 Hertz rejection mode. This should be done before declaring sensor types, but after any board resets. After calling this function, the only way to return to the default 60 Hertz rejection mode is to call the `ResetBoard()` function.

Prototype: `void SetReject50Hz( long hBD );`

Parameter	Type	Description
hBD	long	Board handle.

Example:

```
//////////////////////////////////////////////////////////////////////
// Switch board number 7 to the 50 Hertz rejection mode.
//////////////////////////////////////////////////////////////////////

SetReject50Hz( 7 );
```

3.6 Board Status Functions

3.6.1 GetFaultFlags()

Function: Returns a set of bit flags that indicate various fault conditions. This function may be called at any time after `SetBoardAttr()` has been called for the target board.

Prototype: `long GetFaultFlags( long hBD );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
return value	`long`	Zero or more active-high bit flags, specified by the following masks:

`0x00000001` **Driver failed to receive expected data from board.**
The onboard CPU has failed to return data in response to a command. In the case of non-PCI models, this is often caused by an address conflict with some other device.

`0x00000002` **Driver could not send command to board.**
The onboard CPU has failed to accept a command. In the case of non-PCI models, this is often caused by an address conflict with some other device.

`0x00000004` **Persistent fault condition.**
The board's Fault LED either never turns on or is turned on all the time. In the case of non-PCI models, this is sometimes the result of an address conflict with some other device.

`0x00000008` **Driver failed to open.**
This is usually caused by a missing or unregistered software component. Make sure all driver software components are properly installed and registered (see *Installation* instructions above).

`0x00000010` **Driver function `SetBoardAttr()` failed to register board.**
In the case of non-PCI models, this is usually caused by incorrect specification of the base I/O address or a hardware conflict with some other device in the system. In the case of PCI and CompactPCI models, this is usually an indication of a board that has not been properly detected by the BIOS. For PCI models, try terminating all applications, then performing a hard reset and reboot.

Example:

```
////////////////////////////////////////////////////////////////////////
// Fetch all fault flags from board number 2.
////////////////////////////////////////////////////////////////////////

long faults = GetFaultFlags( 2 );
```

3.6.2 GetBoardAdrs()

Function: Returns the physical address of a Smart A/D™ board. The address value is always returned as a 32-bit integer, but the interpretation of the address value depends upon the board type. See "Physical Addressing" on page 3 for details.

This function serves primarily as a utility to help determine the slot and bus numbering conventions of any PCI or CompactPCI system. In PCI and CPCI systems that have multiple Smart A/D™ boards, the ability to reference a specific board is critical. For example, one Smart A/D™ board might monitor a heater while a second board monitors a cooler. For obvious reasons an application must be able to distinguish between these boards, and the only way to do this in a PCI/CPCI system is to assign fixed slot locations to each board.

Prototype: `long GetBoardAdrs( long hBD );`

Parameter	Type	Description
hBD	long	Board handle.
return value	long	Board's physical address as described in section 3.2.

Example:

```
//////////////////////////////////////////////////////////////////////////
// Determine the physical addresses of all Model 619 PCI Smart A/D boards.
//////////////////////////////////////////////////////////////////////////

#define MAXBOARDS  16     // Maximum number of Smart A/D boards supported by driver.

short nboards;            // Number of detected Model 619 boards.
long  adrs[MAXBOARDS];    // Addresses of all detected Model 619 boards.

// Find and register all Model 619 boards.
for ( nboards = 0; nboards < MAXBOARDS; nboards++ )
{
    if ( !SetBoardAttr( nboards, 619, 0 ) )    // If no more boards,
        break;                                 //   terminate seek loop.
}

// Get the physical addresses of all Model 619 boards.
for ( short boardnum = 0; boardnum < nboards; boardnum++ )
    adrs[boardnum] = GetBoardAdrs( boardnum );
```

3.6.3 GetFWVersion()

Function: Returns the Smart A/D™ board firmware version number, times 100. For example, a call to this function that returns the value 223 indicates that the Smart A/D™ board is running firmware version number 2.23.

This function is useful when it is necessary to automatically determine whether a board feature—which may be available only in specific firmware releases—is accessible in an installed Smart A/D™ product.

Prototype: `long GetFWVersion( long hBD );`

Parameter	Type	Description
hBD	long	Board handle.
return value	long	Board's Smart A/D™ firmware version number, times 100.

Example:

```
//////////////////////////////////////////////////////////////////////
// Get the Smart A/D firmware version that board number 2 is running.
//////////////////////////////////////////////////////////////////////

double smartadversion = GetFWVersion( 2 ) / 100.0;
```

3.6.4 GetIoNetVersion()

This function is applicable if you are using Networking Smart A/D boards.

Function: Returns the IoNet™ firmware version number, times 100. For example, a call to this function that returns the value 200 indicates that the IoNet™ unit is running firmware version number 2.00.

This function is useful when it is necessary to automatically determine whether a board feature—which may be available only in specific firmware releases—is accessible in an installed IoNet™ Smart A/D™ product.

Prototype: `long GetIoNetVersion( long hBD );`

Parameter	Type	Description
hBD	long	Board handle.
return value	long	Board's IoNet™ firmware version number, times 100.

Example:

```
////////////////////////////////////////////////////////////////////////
// Get the IoNet firmware version that board number 2 is running.
////////////////////////////////////////////////////////////////////////

double netversion = GetIoNetVersion( 2 ) / 100.0;
```

3.6.5 ChkIoNetReset()

This function is applicable if you are using Networking Smart A/D boards.

Function: Returns a value that indicates whether the designated IoNet™ unit experienced a reset since the last time this function was called.

IoNet™ hardware resets are not usually caused by application programs. Consequently, there is often a need for the application to determine if a reset event has occurred. This function provides a mechanism for determining if such a reset event has occurred.

All IoNet™ products maintain a Reset Flag that is set upon hardware reset of the unit. This function returns the state of the Reset Flag and clears the flag so that future calls to this function will indicate that the IoNet™ has not been reset. Only a hardware reset of the IoNet™ unit will cause the Reset Flag to be set again.

Prototype: `long ChkIoNetReset( long hBD );`

Parameter	Type	Description
hBD	long	Board handle.
return value	long	Non-zero value indicates IoNet™ unit has been reset since the last call to this function.

Example:

```
////////////////////////////////////////////////////////////////////////
// Determine if board number 0 has been reset.
////////////////////////////////////////////////////////////////////////

if ( ChkIoNetReset(0) )
{
    // Insert code here to handle unexpected IoNet resets.
}
```

3.7 Channel Configuration Functions

All channel configuration functions specify a channel number parameter which designates the channel to be affected by the function. Channel numbers always begin at zero and extend upward to the maximum valid channel number belonging to the addressed board.

For example, models with eight channels use channel numbers ranging from 0 through 7. Models with sixteen channels use channel numbers ranging from 0 through 15, etc.

3.7.1 SetFailMode()

Function: Declares whether the specified channel's output data value should default to a large positive value or to a large negative value in the event an open sensor is detected.

Prototype: `void SetFailMode( long hBD, long channel, long failhigh );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`channel`	`long`	Sensor channel number to be affected.
`failhigh`	`long`	0 = fail low (large negative value). 1 = fail high (large positive value).

Example:

```
//////////////////////////////////////////////////////////////////////
// Program all channels on board number 5 to fail high.
//////////////////////////////////////////////////////////////////////

#define NCHANS 16    // Assume this is a 16-channel Smart A/D board

for ( chan = 0; chan < NCHANS; chan++ )
    SetFailMode( 5, chan, true );
```

3.7.2 SetFilter()

Function: Applies a software low-pass filter to the specified channel. The filter is implemented by the Smart A/D™ board as described in the Smart A/D™ hardware instruction manual.

Prototype: `void SetFilter( long hBD, long channel, long filtval );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`channel`	`long`	Sensor channel number to be affected.
`filtval`	`long`	Values may range from 0 (default, filter is disabled) to 255 (maximum filter). Filter values outside the legal range may cause unexpected filtering results.

Example:

```
//////////////////////////////////////////////////////////////////////
// Program board number 5, channel 11, to have a 50% filter.
//////////////////////////////////////////////////////////////////////

#define FILTER_PCT  50    // Desired filter percentage (0% to 99%).

SetFilter( 5, 11, FILTER_PCT * 2.56 );
```

3.7.3 SetSensorType()

Function: Declares the sensor type to be interfaced to the specified channel.

The sensor type is specified by means of a Logical Sensor Definition Code (LSDC). Not all LSDC values are supported by all Smart A/D™ products; refer to your hardware documentation for a complete list of supported sensor types and input rangesUse of illegal or unsupported LSDC values may cause unpredictable results.

Do not confuse the LSDC with the native SDC that may be discussed in your hardware documentation. The LSDC values are valid for all Smart A/D™ models, while the native SDC values, which are described in your hardware documentation, are specific to each product model and may deviate from one model to another.

Prototype: `void SetSensorType( long hBD, long channel, long LSDC );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`channel`	`long`	Sensor channel number to be affected.
`LSDC`	`long`	Logical Sensor Definition code from the following table:

Input Class	LSDC	Input Range / Sensor Type
Voltage Input	1	±10V
	3	±5V
	5	±500mV
	6	±100mV
Thermocouple	34	B
	33	C
	8	E
	10	J
	12	K
	32	N
	18	R
	16	S
	14	T
RTD	41	Copper 10 Ohm
	35	Nickel 120 Ohm
	36	Nickel 200 Ohm
	37	Nickel 1K Ohm (1353 Ohms @ 60 DegC)
	42	Nickel 1K Ohm (1285 Ohms @ 60 DegC)
	20	Platinum 100 Ohm 385 (0.1C res, full sensor range)
	38	Platinum 100 Ohm 385 (0.0125C res, range -200:+400C)
	22	Platinum 100 Ohm 392 (0.1C res, full sensor range)
	39	Platinum 100 Ohm 392 (0.0125C res, range -200:+400C)
	43	Platinum 1K Ohm (1422.9 Ohms @ 110 DegC)
Thermistor	24	10K with autorange
	40	10K on fixed 600K ohm range
Current Loop	25	4-20 mA
Gage	26	Strain/pressure gage

Input Class	LSDC	Input Range / Sensor Type
Resistance	27	400 ohm range
	28	3K ohm range
	29	600K ohm range
Other	31	Disabled: remove channel from scan list

Example:

```
////////////////////////////////////////////////////////////////////////
// Program board number 3, channel 7, to interface to a K Thermocouple.
////////////////////////////////////////////////////////////////////////

#define KTC 12  // Logical Sensor Definition Code for K Thermocouple.

SetSensorType( 3, 7, KTC );
```

3.8 Sensor Acquisition Functions

3.8.1 GetAmbientTemp()

Function: Returns the ambient temperature from a Sensoray termination board. The returned temperature value is from the thermocouple cold-junction compensation sensor that is present on all Smart A/D™ termination boards. This function is useful as a diagnostic tool as it can be used to acquire digitized data from the Smart A/D™ subsystem without depending on the presence of external sensors.

Prototype: `double GetAmbientTemp( long hBD, long tb );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`tb`	`long`	Specifies the termination board (TB) to be addressed: **8-channel Models:** always set to zero. **16-channel Models:** 0 = TB for channels 0 to 7, 1 = TB for channels 8 to 15. **32-channel Models:** 0 = TB for channels 0 to 15, 1 = TB for channels 16 to 31.
return value	`long`	TB temperature, in degrees C.

Example:

```
////////////////////////////////////////////////////////////////////////
// Fetch the ambient temperature from board 3 at the TB connected
// to the lowest channels.
////////////////////////////////////////////////////////////////////////

GetAmbientTemp( 3, 0 );
```

3.8.2 GetAllSensors()

Function: Returns sensor data from all channels on the specified board. The returned values are represented in engineering units that are appropriate for the sensor type. For example, channels configured for voltage measurement always return Volts. Channels configured for temperature sensors typically return Degrees Centigrade units.

Note: Visual Basic Version 5 has a bug that may cause unpredictable side-effects as a result of calling this function. If you are using VB5, we recommend that you do not use this function, but instead make multiple calls to the `GetSensorData()` function to obtain sensor data.

Prototype: `void GetAllSensors( long hBD, double *databuf );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`databuf`	`double*`	Address of an array of doubles that will be filled with sensor data from all channels on the specified board. To prevent system faults, make sure the array is large enough to contain data from all channels on the target board.

Example:

```
//////////////////////////////////////////////////////////////////////
// Fetch all sensor data from board 3, then extract the sensor data from
// channel 6, which has been previously configured for interface to a type
// K thermocouple.
//////////////////////////////////////////////////////////////////////

#define NCHANS 16               // Assume this is a 16-channel Smart A/D board

double SensorData[NCHANS];      // Allocate a large-enough buffer for all sensor chans

GetAllSensors( 3, SensorData );  // Copy all sensor data to buffer
double DegreesC = SensorData[6]; // Extract channel 6 sensor data
```

3.8.3 GetSensorData()

Function: Returns sensor data from one channel. The returned value is represented in the engineering units that are appropriate for the sensor type. For example, channels configured for voltage measurement always return Volts. Channels configured for any type of temperature sensor always return Degrees Centigrade units.

Prototype: `double GetSensorData( long hBD, long channel );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`channel`	`long`	Board handle.
return value	`long`	Board's physical address. as described in section 3.2.

Example:

```
//////////////////////////////////////////////////////////////////////
// Fetch the sensor data from board 3, channel 6, which has been
// previously configured for interface to a Platinum 1K Ohm RTD.
//////////////////////////////////////////////////////////////////////

double DegreesC = GetSensorData( 3, 6 );
```

3.9 Strain/Pressure Gauge Functions

This section is applicable if you are using strain or pressure gauges.

Special driver functions are provided to simplify the software interface to strain and pressure gauges. These functions apply only to Smart A/D™ models that support interface to gauges. Invoking these functions on models that do not support gauges may cause unpredictable behavior and should be avoided. Refer to your Smart A/D™ Instruction Manual to determine if your model supports gauges.

All of the gauge-related functions specify a channel number parameter which designates the channel to be affected by the function. Gauge functions assume that the referenced channel has been previously configured for interface to a gauge by means of the `SetSensorType()` function.

3.9.1 SetGageZero()

Function: Registers a "zero load" condition with the Smart A/D™ board. This function is invoked as the first step of a two-step physical gauge calibration procedure. Refer to your Smart A/D™ Instruction Manual for detailed information regarding gauge calibration.

Note: the applied load need not be a true "zero" load. It is only required that the physical load that is present when this command executes also is applied when the `SetGageSpan()` function executes.

Prototype: `void SetGageZero( long hBD, long channel );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`channel`	`long`	Sensor channel number to be affected.

Example:

```
//////////////////////////////////////////////////////////////////////
// Start a physical gauge calibration on board 3, channel 5.
// Assumes: (1) channel 5 is configured for a strain/pressure gauge, and
//          (2) the gauge on channel 5 has a Zero Load condition.
//////////////////////////////////////////////////////////////////////

SetGageZero( 3, 5 );
```

3.9.2 SetGageSpan()

Function: Registers a "full load" condition with the Smart A/D™ board. This function is called as the final step in a two-step physical gauge calibration procedure.

A gauge load parameter, `load`, must be specified when this function is called. This value represents the difference between the load that is applied when `SetGageZero()` was called and the load that is applied when `SetGageSpan()` is called. The `load` is always specified as an integer, as the Smart A/D™ board inherently produces an integer data value for any arbitrary load condition. The value of `load` is determined by dividing the applied physical load by the desired resolution. The resulting value must fall within the range -32768 to +32767.

For example, suppose the applied load is 2,000 pounds and a resolution of 0.1 pounds is desired. The `load` parameter should be set to `2,000 / 0.1 = 20,000`. After executing this command, the `GetSensorData()` function will return sensor data from this channel in units of 0.1 pounds. In this case, an applied load of 153.7 pounds, for example, would cause `GetSensorData()` to return the value `1537`.

Prototype: `void SetGageSpan( long hBD, long channel, long load );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`channel`	`long`	Sensor channel number to be affected.
`load`	`long`	Applied physical load, ranging from -32768 to +32767. This value is the number you should expect to measure at the applied load condition when calling `GetSensorData()`.

Example:

```
////////////////////////////////////////////////////////////////////////
// Complete the physical gauge calibration on board 3, channel 5, that
// began in the previous example.
// Assumes: (1) channel 5 is configured for a strain/pressure gauge, and
//          (2) the sensor on channel 5 has 3,000 pounds of additional
//              load since calling SetGageZero(), and
//          (3) a resolution of 0.5 pounds is desired.
////////////////////////////////////////////////////////////////////////

#define FULLLOAD    3000    // Applied load when SetGageSpan() is called.
#define RESOLUTION  0.5     // Desired resolution, in pounds.

SetGageSpan( 3, 5, FULLLOAD / RESOLUTION );
```

3.9.3 SetGageTare()

Function: Tares the gauge on the specified channel. Taring is accomplished by adjusting the data offset so that sensor data returned by `GetSensorData()` will equal zero at the current load condition..

Prototype: `void SetGageTare( long hBD, long channel );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`channel`	`long`	Sensor channel number to be affected.

Example:

```
////////////////////////////////////////////////////
// Tare the gauge on board number 1, channel 7.
////////////////////////////////////////////////////

SetGageTare( 1, 7 );
```

3.9.4 GetGageCal()

Function: Returns the calibration parameters from a previously calibrated gauge. The returned values can be used by the `SetGageCal()` function to restore a gauge calibration without having to perform a physical calibration.

Note: The returned values are represented in a proprietary form that is used internally by the Smart A/D™; they should not be construed as having any relationship to published gauge parameters, such as a mV/V or offset rating.

Prototype: `void GetGageCal( long hBD, double channel, double *gain, short *offset );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`channel`	`long`	Sensor channel number to be affected.
`gain`	`double*`	Pointer to a double that will receive the gauge “gain” parameter.
`offset`	`short*`	Pointer to a short that will receive the gauge “offset” parameter.

Example:

```
////////////////////////////////////////////////////////////////////////
// Fetch gauge calibration parameters from board number 0, channel 2.
// Assumed: (1) Channel 2 has been configured for strain/pressure gauge.
//          (2) Channel 2 previously calibrated either physically, or by
//              means of the SetGageCal() function.
////////////////////////////////////////////////////////////////////////

double gain2;      // Gain parameter will be stored here.
short  offset2;    // Offset parameter will be stored here.

GetGageCal( 0, 2, &gain2, &offset2 );
```

3.9.5 SetGageCal()

Function: Restores a gage calibration without having to perform a physical calibration.

Prototype: `void SetGageCal( long hBD, long channel, double gain, short offset );`

Parameter	Type	Description
`hBD`	`long`	Board handle.
`channel`	`long`	Sensor channel number to be affected.
`gain`	`double`	Gauge "gain" parameter, as obtained from `GetGageCal()`.
`offset`	`short`	Gauge "offset" parameter, as obtained from `GetGageCal()`.

Example:

```
////////////////////////////////////////////////////////////////////////
// Restore gauge calibration on board number 0, channel 2, by using the
// values obtained in the GetGageCal() example above.
// Assumed: Channel 2 has been configured for strain/pressure gauge.
////////////////////////////////////////////////////////////////////////

SetGageCal( 0, 2, gain2, offset2 );
```