CTI-HSIF-PCI Software Library

for

Acuity AR4000
Laser Distance Sensors
and PCI High-Speed Interface Card

Programmer's Guide

and

Reference Manual

Crandun Technologies Inc.

Manual V1.02

Printed in Canada

Table of Contents

Intended Audience

This document is intended for:

- End-users who will install and use the CTI Software Library to acquire data from Acuity AR4000 sensor systems equipped with the PCI version of the AccuRange High-Speed Interface card using one of the sample programs or spreadsheets shipped with the software.

- Software developers who will use the CTI Software Library to develop custom end-user application programs to acquire and manipulate data from AR4000 sensor systems.

- Software developers who will use the CTI Software Library to develop value-added application programs or libraries which will be redistributed to end-users or other downstream customers.

Note: This document is not a programming tutorial. Software developers using this library should be fully familiar with the programming language and development tools being used and be experienced in developing and debugging applications using that language. All users of this library should be fully familiar with the AR4000 and high speed interface hardware, and have read all appropriate Acuity documentation

Support Information

If you have questions or problems regarding the CTI-HSIF-PCI Software Library, please follow these steps:

1. Read this manual carefully, as most installation, usage and programming questions are answered in this document. The latest version of this manual, with correction of any errors that may be discovered after printing, is also available on our web site at www.crandun.com.

2. Ensure that the sensor, high speed interface, and cabling is properly installed and functioning. Consult the Acuity documentation for instructions on verifying the operation of the hardware.

3. Run the installation verification program (see page 3) to ensure that the software is properly installed.

4. Ensure that you are using the most recent version of the software. During installation you will be asked if the installation program should check the Crandun Technologies web site for an updated software version. It is strongly recommended that you do so, to ensure that the version you are using is the most up-to-date.

5. Consult the sample programs installed with the software. These working examples illustrate many common techniques for using the software library, and can be used as the building blocks for your application programs.

6. Consult our web site at www.crandun.com. Any errors discovered in this document are posted there, and the FAQ list on this site is frequently updated with answers to questions from our customers.

7. Technical support for this product is provided by Schmitt Measurement Systems Inc. Please contact them by telephone at 503-227-5178, or by email using the contact information given at www.acuityresearch.com.

8. If none of the above resolves your question, please contact Crandun Technologies Inc. by telephone at (905) 692-0012 or by email at support@crandun.com.

Feedback and Suggestions

Crandun Technologies welcomes your feedback and suggestions. If you should find an error or omission in this document, or the accompanying programs, or have suggestions on improving the clarity of the material, please contact us through our web site at www.crandun.com. Although this manual contains no known errors at the time of printing, an errata list on our web site will correct any errors that do come to our attention.

This page is intentionally blank.

Part I – Programmer's Guide

Product Overview

The Crandun Technologies Inc. CTI-HSIF-PCI software library is a set of kernel drivers, header files, object files, and linkable libraries (or DLLs on the Windows platform), that provide a high-performance, high-level interface to the functionality of the Acuity AR4000 series laser distance sensors, using the AccuRange High-Speed Interface card. Please note that this library supports the PCI version of the high-speed interface card. Customers with the ISA or PC104 version of the high-speed interface card should use the CTI-HSIF library instead. Consult our website at www.crandun.com for information on that library.

This library provides a "plug-and-play" interface to the sensor functionality, allowing the user to concentrate on their particular application, rather than on the details of data acquisition from the sensor. The library permits the easy and rapid development of application programs to acquire data from the Acuity AR4000 sensors using a variety of high-level languages on a standard PC platform.

In addition to the software library itself, the distribution package includes a number of C and C++ sample programs that demonstrate various features of the library, and illustrate techniques for making effective use of the library's capabilities. On Windows, Visual Basic and Microsoft Excel samples are also included. Please see the section "Sample Programs" on page 31 for more details.

Highlights

- High performance data acquisition using the AccuRange high-speed interface card.
- Usable from C, C++, and [on Windows] Visual Basic, and Visual Basic for Applications.
- Access to the complete functionality provided by the AR4000, High-Speed interface, and Line Scanner, including support for the High Speed Interface's half-full interrupt.
- Simple, intuitive, english-like application programming interface (API) to the AR4000 command set.
- User-configurable data buffering and callback notification.
- User-configurable filtering, based on amplitude, range, ambient light, or sensor temperature.
- Complete encoder support, including counter wrap-around handling [when using AccuRange Line Scanner].
- Encoder angle interpolation for increased sample precision.
- Fully configurable serial port baud rate and flow control.
- Multi-threaded and interrupt driven, for highest performance.
- Maintains state information, for easy determination of current sensor settings.

Features

The CTI-HSIF-PCI software library has been designed for ease of use and maximum flexibility and performance. No programming is needed to acquire data from the AR4000 sensors directly into a Microsoft® Excel spreadsheet. For more complex needs, the library's application programming interface (API) facilitates the rapid development of custom applications in C, C++ or Visual Basic®. The library provides an object-oriented class interface for C++ and Visual Basic®, and a call-level interface for C programmers.

A variety of features enhance the ease of use for an application programmer. These include:

Flexible Output Formats

The CTI-HSIF-PCI library allows the user to select a number of formats for the acquired data. English or metric output, calibrated or uncalibrated data, or both, may be selected. Additional information (signal

amplitude, ambient light, temperature) may optionally be returned. When using the Line Scanner, encoder count to angle conversion is done by the library, and data may be output in either polar or Cartesian coordinates.

Data Filtering

The library supports filtering of the raw data based on any combination of range, amplitude, ambient light and temperature. Only samples within each user-specified range are returned to the application. This can be used to discard spurious samples having high ambient light, or low amplitude readings, for example.

Configurable Data Buffering and Callback Notification

The High-Speed Interface card's onboard memory may be augmented by a user configurable buffer within the library, which will be filled as data arrives over the high-speed interface. When a predefined number of samples is available, the library sets a queryable status flag, or calls a user-defined callback function, enabling the application program to easily determine when a desired number of samples is ready for processing.

Multi-threaded and Interrupt Driven

Internally, the CTI-HSIF-PCI library is fully multi-threaded and interrupt driven for maximum performance.

Installing the Software

Please see "Appendix A - Installation & Components" for details of the installation procedure for your particular operating system.

Installation Verification Program

After installation, it is strongly recommended that you run the installation verification program to verify that the software has been correctly installed, and that communications with the AR4000 sensor and high-speed interface card can be successfully established. The installation verification program is installed along with the software, and may be re-run at any time.

Use of the verification program is self-explanatory, and its results will be displayed on the screen.

Should the verification program fail, one of the following common problems may exist:

- The license keycode has been entered incorrectly, or does not correspond to the particular sensor attached to the computer.
- The sensor's communications port is not configured for the factory default of 9600 baud. Please refer to the sensor hardware documentation for information on setting the sensor's baud rate.
- The sensor is not powered on, or the cabling between the computer and the sensor is faulty. Please refer to the sensor hardware documentation for information on confirming the cabling hookup and basic operation of the sensor.
- The serial port specified is incorrect.
- The serial port hardware is not functioning. This is particularly common on laptop computers, where the Power Management system may turn off the computer's serial port after a timeout interval.
- The serial port is already being used by another application (for example, a terminal program or an Internet connection).
- The high-speed interface card is not properly installed. Ensure that the card is firmly seated in the computer.

If after correcting these problems, the verification program still returns an error message, please contact us with a description of the error messages and codes displayed. (See "Support Information" page v).

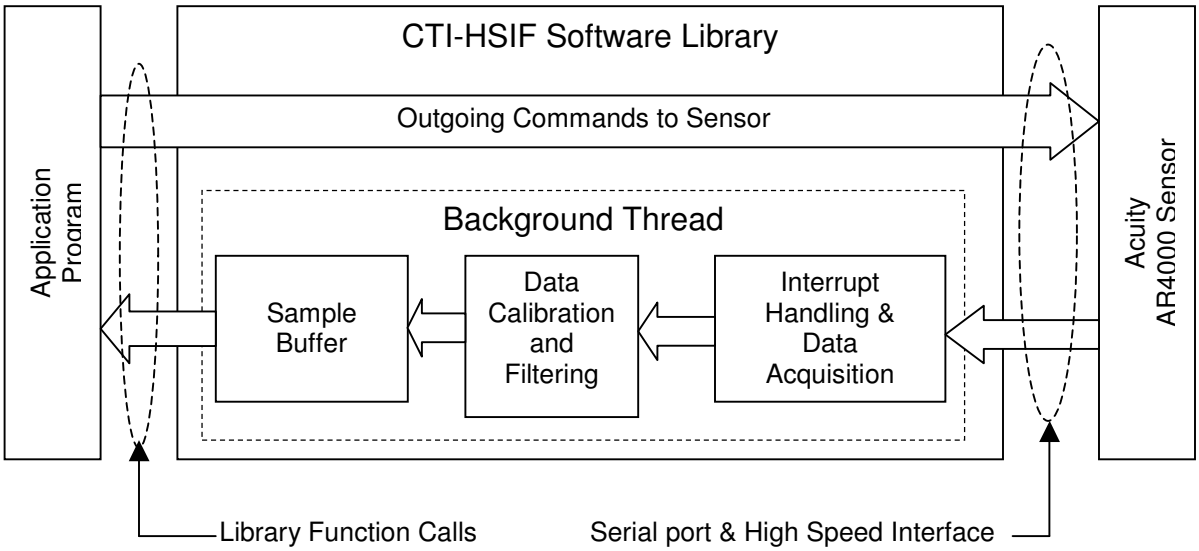
Library Components

Please see "Appendix A - Installation & Components" for details of the specific files installed on your particular operating system.

Library Conceptual Overview

This section gives an overview of the internal workings of the library. It is recommended that all users read this carefully, as an understanding of the information in this section will assist in making effective use of the library's features.

The CTI-HSIF-PCI library is an interface layer between the Acuity AR4000 sensor hardware and the application program(s). The library communicates with the sensor via the computer's serial port, and the high-speed interface card, while the application program communicates with the library via function calls. All commands sent to the sensor by the application program pass through the library, and all data sent by the sensor is processed by the library and made available for use by the application program.



One of the major functions of the library is to manage the flow of data to and from both the serial interface and the high-speed interface card, and ensure that both outbound data (commands sent to the sensor), and inbound data (range samples from the high-speed interface) are properly processed. To accomplish these tasks, the library consists of several distinct functional blocks. These are shown pictorially in the diagram above, and explained below.

Commands to Sensor

When an application program calls a library function that sends a command to the sensor or high-speed interface, the outbound data is passed directly to the computer's serial port for transmission to the sensor or is sent directly to the high-speed interface card. The library will not return control to the application program until the command has been sent. Inbound data however, goes through a more complex series of steps, as explained below.

Background Processing Thread

A key design concept of the CTI-HSIF-PCI library is a separation of responsibilities – the library is responsible for reading data from the sensor, and translating that data to range samples, while the application program is responsible for processing those samples, once read.

To accomplish its task, the library uses a background processing thread[1]. The background thread is started by the library when the `setCommOpen()` function (see the Reference Manual) is called, and runs continuously

[1] A *thread* is a separate stream of execution within a single program. The background thread runs *concurrently* with any other processing that may be done by the main part of the program. For more information on the concepts of threads, please consult a suitable programming text.

until the `setCommClosed()` function is called. *Without any interaction by the application programmer*, the thread reads data from the computer's serial port and the high-speed interface card, as it is received from the sensor and passes it to the other parts of the library for subsequent processing.

Since, during use, the AR4000 sensor is typically outputting a continuous stream of data samples via the high-speed interface, this ensures that those samples are captured and stored for use by the application program. Because the background thread runs independently of whatever else the application program may be doing, this design permits the application programmer to concentrate on processing samples already retrieved from the library, while the background thread assumes the responsibility of acquiring any new incoming data simultaneously.

Interrupt Handling and Data Acquisition

This functional block implements the low-level driver that handles hardware interrupts from the high-speed interface card, and reads samples from the card's hardware buffer into the library's internal memory.

Interrupt Handling

The high speed interface has the capability of generating a hardware interrupt whenever the card's onboard buffer is half-full. Through a kernel mode driver, the CTI-HSIF-PCI library recognizes when this interrupt occurs, and using a high-priority background thread reads the data from the buffer. Using the high speed interface's interrupt generation capability is the preferred mode of execution, since only when the interrupt occurs, indicating that data is actually available, does the library's background thread run.

Data Acquisition

Once the library has been notified that data is available (by an interrupt), it reads data from the high speed interface's onboard buffer into the library's internal memory. Samples which have the 'data lost' flag set may contain inaccurate data (see the sensor hardware documentation) and are discarded.

The remaining samples are passed to the data calibration and filtering block.

Data Calibration and Filtering

This functional block converts the raw encoder readings and range samples into angular measurements and calibrated samples.

The raw samples from the high-speed interface are calibrated using the data from the calibration file provided with the sensor hardware. Any filtering and transformation rules set by the application programmer are then applied to the data samples. For example, the application programmer may be interested in only those range samples that lie between specific bounds. This functional block sets to zero (or discards, if desired) any samples outside the specified range.

See the Reference Manual for more information on the Data Filtering functions of the library.

Sample Buffer

All samples that pass the data filtering stage are stored internally in the library's internal data buffer. When an application program requests samples, they are retrieved from the buffer and returned to the application. If sufficient samples are not currently available in the buffer, the library will wait until additional samples are inserted in the buffer by the background processing thread.

The size of the buffer is configurable by the application programmer, and the library provides functions to determine the number of samples currently available in the buffer, or to wait for a specific number of samples to be available.

Additionally, the library provides the ability for a user-specified callback function to be called when the buffer contains a desired number of samples. This callback function will be executed in a new thread of execution, independent of both the background processing thread mentioned above, and the main thread of the program.

Using the Library

Library Functions

All interaction with the AR4000 sensor or the high-speed interface card is done through one of the functions provided by the CTI-HSIF-PCI Library. These provide for configuring the library itself, and configuring and acquiring data from the sensor. Each function is explained in detail in the Reference Manual. The functions fall into one of the following groups:

Library Configuration Functions

These functions are used for configuring various aspects of the CTI-HSIF-PCI library itself, such as which serial port the library will use to communicate with the sensor, the library's internal buffer size, the callback threshold, etc..

Sensor and High Speed Interface Configuration Functions

These functions interact with the AR4000 sensor hardware and high-speed interface card to configure the sensor. Since the AR4000 sensor and high speed interface hardware do not provide means of determining their current settings, the CTI Library tracks the current state of each setting, for easy determination by an application programmer.

When using the AR4000 sensor and high-speed interface card, most sensor configuration commands are sent to the sensor via the serial port. However, some operations (such as setting the motor power or clearing the high-speed interface's buffer) require interacting directly with the high-speed interface board.

The application programmer need not be concerned with this distinction. The programmer simply calls the appropriate function, and the CTI library communicates with the serial port or high-speed interface card as needed.

Data Acquisition Functions

The data acquisition functions are used to turn the laser on or off, set the sensor sampling rate, acquire data from the sensor, turn on or off continuous sampling, etc..

Data Format Functions

These functions set the sample units to English units (inches) or metric units (millimetres), and the angle measurements to polar or Cartesian coordinates.

Data Filtering and Transformation Functions

The data filtering functions allow filtering the raw samples from the sensor based on any combination of range, amplitude, ambient light, or temperature reading. These functions are explained in detail in the section "Data Filtering" on page 25 of the Programmer's Guide, and the section "Data Filtering and Transformation Functions" starting on page 51 of the Reference Manual.

Error Handling and Miscellaneous Functions

These functions perform various miscellaneous operations, such as querying the library's version number, the sensor's firmware version, and retrieving error messages.

The "Hello World" Program

Many programming texts use the so-called "Hello World" example to introduce the syntax and features of a programming language. The "Hello World" example is the simplest possible program – one that simply prints the words "Hello World", then exits. This section presents the equivalent "Hello World" for the CTI-HSIF-PCI library – the simplest possible program that communicates with the sensor and high speed interface, acquires data and prints that data.

The "Hello World" program is presented below in Visual Basic, C++ and C, and an Excel spreadsheet using Visual Basic for Applications (VBA). The corresponding executable programs, as well as complete source code for each of these examples is installed in the "Samples" subdirectory during the installation.

Hello World In C++

The C++ "Hello World" sample opens communications to the sensor and high speed interface and acquires samples into an array. Each range sample is printed and the connection to the sensor is closed. See the source code listing and explanation below for more details. In this sample, the line scanner and encoder are not used.

```
    // "Hello World" example for the Crandun Technologies CTI-HSIF-PCI software library.
    // Copyright (c) 2006, Crandun Technologies Inc.
    //
    #include <iostream>
A   #include "CTI_HSIF_PCI.h"  // required header file

    using namespace std;
B   using namespace Crandun;   // all CTI library symbols are in this namespace

    int main()
    {
        char c;
        char errMsg[300];
C       long rc;
        const int maxSamples = 20;
        HSIF_DATA_PT rangeData[maxSamples];

D       CTI_HSIF_PCI my_Sensor;

        cout << "Crandun Technologies CTI-HSIF-PCI Library 'Hello World' example." <<
    endl;

        // Open driver
E       rc = my_Sensor.setDriverOpen();
F       if (rc != CTI_SUCCESS) {
            cerr << "ERROR: setDriverOpen returned error: " << rc << endl;
            cerr << "Enter any character to exit: ";
            cin >> c;
            return -1;
        }

        // Set calibration data file to use. MUST match the particular HSIF card
        rc = my_Sensor.setCalibrationFile("C:\\lookuphs");
G       if (rc != CTI_SUCCESS) {
            cerr << "ERROR: setCalibrationFile returned error: " << rc << endl;
            cerr << "Enter any character to exit: ";
            cin >> c;
            return -1;
        }

        // Open communications to the serial port and high-speed interface
        cout << "Opening the serial port..." << endl;
H       rc = my_Sensor.setCommOpen("COM1", 9600);
        if (rc != CTI_SUCCESS) {
            my_Sensor.setCommClosed();
```

```
        cerr << "ERROR: setCommOpen returned error: " << rc << endl;
        if (my_Sensor.getIsError()) {
            my_Sensor.getErrorMessage(errMsg, sizeof(errMsg));
            cerr << "Library error msg is: " << errMsg << endl;
        }
        cerr << "Enter any character to exit: ";
        cin >> c;
        return -1;
    }

    // Get samples from the sensor
    cout << "Reading samples from sensor ..." << endl;
I   rc = my_Sensor.getSamples(rangeData, maxSamples, 10, 1000);
J   if (rc < 0) {
        cerr << "ERROR: getSamples returned error: " << rc << endl;
        if (my_Sensor.getIsError()) {
            my_Sensor.getErrorMessage(errMsg, sizeof(errMsg));
            cerr << "Library error msg is: " << errMsg << endl;
        }
        my_Sensor.setCommClosed();
        cerr << "Enter any character to exit: ";
        cin >> c;
        return -1;
    }
K   my_Sensor.setCommClosed();


    cout << "Read " << rc << " range samples from the sensor." << endl;
L   for (int i= 0; i < rc; i++)
        cout << "Range " << i << " is " << rangeData[i].R_X << endl;

    cout << "Enter any character to exit: ";
    cin >> c;
    return 0;
}
```

The points below describe what each line of the program is doing. Please reference the corresponding line numbers shown to the left of the code listing above.

A. This line includes the header file that defines the classes, methods, functions and error codes of the CTI-HSIF-PCI library. This header file must be included by any program using the CTI-HSIF-PCI library.

B. All C++ classes and methods defined by the CTI-HSIF-PCI library reside in namespace "`Crandun`". The "`using namespace Crandun`" statement should be included by any C++ program using CTI-HSIF-PCI library. If not included, then each symbol defined by the CTI-HSIF-PCI library must be explicitly qualified when referenced (e.g. `Crandun::CTI_HSIF my_Sensor`). This can lead to a rather "messy" syntax, so we recommend use of the "`using namespace…`" statement.

C. The return codes of all methods supplied by the CTI-HSIF-PCI library must be checked for success or failure. The "`rc`" variable will be used to hold the return code from each method call in the sample program. The next two lines declare the array `rangeData`, used in the call to the `getSamples` method (see below).

D. This line instantiates a local object (variable) called "`my_Sensor`" of type "`CTI_HSIF`". Note that at this point, the object does not yet refer to any specific AR4000 sensor or high speed interface that may be attached to the computer.

E. This line opens communication to the CTI-HSIF-PCI library. It must be the first function called when using the library from C++.

F. This checks the return code from the `setDriverOpen()` call. It is *essential* that every method's return code be checked for success or failure

G. This line sets the calibration file to use to calibrate samples from the high speed interface.

H. This line uses the `setCommOpen()` method to open the communications port (serial port) named "`COM1`", using a baud rate of 9600 baud, and establish communications with the sensor attached to this port. If successful, the method's return code is CTI_SUCCESS, otherwise an error code will be returned.

I. This line tells the CTI-HSIF-PCI library to acquire at least 10, and at most "`maxSamples`" samples from the sensor, and return the results in the array `rangeData`. The library will wait a maximum of one second (1000 milliseconds) for the samples to be available. See the description of the `getSamples()` function in the Reference Manual for more details.

J. This line checks the return code from the `getSamples()` call. If the function succeeds, it will return the number of samples actually acquired. If an error occurs, then a negative number, representing an error code, will be returned. A list of error codes is defined in the "`CTI_HSIF_PCI.h`" header file. If an error does occur, the communications link to the sensor is closed, and the program exits.

K. This line terminates communications with the sensor. The library will close the serial communications port, shut down the background data acquisition threads, and free any resources used internally by the library. This function must be called by an application program to cleanly terminate use of the CTI-HSIF-PCI library. *Failure to call this function before the program exits may result in undesirable behaviour, including memory leaks, and difficulty re-establishing communications with the sensor during subsequent runs of the application program.*

L. These lines print out the samples returned by the `getSamples()` method.

Hello World In C

The C "Hello World" sample opens communications to the sensor and high-speed interface, and acquires data samples into an array. Each range sample is then printed out and the connection to the sensor is closed. Please see the source code listing below, and the explanation following the code listing for more details.

```
    /* "Hello World" example for the Crandun Technologies CTI-HSIF-PCI
        software library. Copyright (c) 2006, Crandun Technologies Inc. */

A   #include "CTI_HSIF_PCI.h"  /* required header */
    #include <stdio.h>
B   #define maxSamples 20

    int main()
    {
        char c;
        char errMsg[300];
C       long i, rc;
        HSIF_DATA_PT rangeData[maxSamples];
        long sensorHandle=-1;

        printf("Crandun Technologies CTI-HSIF-PCI Library 'Hello World' example.\n");

D       sensorHandle = getNewCTIHSIF_PCI();
        if (sensorHandle < 0) {
            printf("ERROR: getNewCTIHSIF_PCI returned error code %ld\n", sensorHandle);
            printf("Please enter any character to exit: ");
            scanf("%c", &c);
            return -1;
        }

        /* Open library */
E       rc = setDriverOpen(sensorHandle);
F       if (rc != CTI_SUCCESS) {
            printf("ERROR: setBoardParams returned error %d\n", rc);
            printf("Please enter any character to exit: ");
            scanf("%c", &c);
            return -1;
        }

        /* Set calibration data file to use. MUST match the actual sensor used */
G       rc = setCalibrationFile(sensorHandle, "C:\\lookuphs");
        if (rc != CTI_SUCCESS) {
            printf("ERROR: setCalibrationFile returned error %d\n", rc);
            printf("Please enter any character to exit: ");
            scanf("%c", &c);
            return -1;
        }

        /* Open communications to the serial port and high-speed interface */
        printf("Opening the serial port...\n");
H       rc = setCommOpen(sensorHandle, "COM1", 9600);
        if (rc != CTI_SUCCESS) {
            printf("ERROR: setCommOpen returned error %d\n", rc);
            return -1;
        }

        /* Get samples from the sensor */
        printf("Reading samples from sensor ...\n");
I       rc = getSamples(sensorHandle, rangeData, maxSamples, 10, 1000);
J       if (rc < 0) {
            printf("ERROR: getSamples returned error %d\n", rc);
            setCommClosed(sensorHandle);
```

```
        printf("Please enter any character to exit: ");
        scanf("%c", &c);
        return -1;
    }

K   setCommClosed(sensorHandle);
L   setReleaseHandle(sensorHandle);

    printf("Read %d range samples from the sensor.\n", rc);
M   for (i = 0; i < rc; i++)
        printf("Range %d is %8.3f inches\n", i, rangeData[i].R_X);

    printf("Please enter any character to exit: ");
    scanf("%c", &c);
    return 0;
}
```

The points below describe what each line of the program is doing. Please reference the corresponding line numbers shown to the left of the code listing above.

A. This line includes the header file that defines the functions and error codes of the CTI-HSIF-PCI library. This header file must be included by any program using the CTI-HSIF-PCI library.

B. We define a constant that will be used to dimension the array holding the range samples.

C. The return codes of all functions supplied by the CTI-HSIF-PCI library must be checked for success or failure. The "`rc`" variable will be used to hold the return code from each function call in the sample program. The next line declares the array `rangeData,` used in the call to the `getSamples` function (see below).

D. This line calls the `getNewCTIHSIF_PCI()` function to obtain a "handle" to a particular sensor. If successful, this function will return a non-negative value that must be passed as the first parameter to all other functions when using the library from C. Note that at this point, the handle does not yet refer to any specific AR4000 sensor or high-speed interface that may be attached to the computer.

E. This line opens communication to the CTI-HSIF-PCI library. It must be the first function called when using the library.

F. This line checks the return code from the `setDriverOpen()` call. It is *essential* that every method's return code be checked for success or failure

G. This line sets the calibration file to use to calibrate samples from the high speed interface.

H. This line uses the `setCommOpen()` method to open the communications port (serial port) named "`COM1`", using a baud rate of 9600 baud, and establish communications with the sensor attached to this port. If successful, the method's return code is CTI_SUCCESS, otherwise an error code will be returned.

I. This line tells the CTI-HSIF-PCI library to acquire at least 10, and at most "`maxSamples`" samples from the sensor, and return the results in the array `rangeData`. The library will wait a maximum of one second (1000 milliseconds) for the samples to be available.

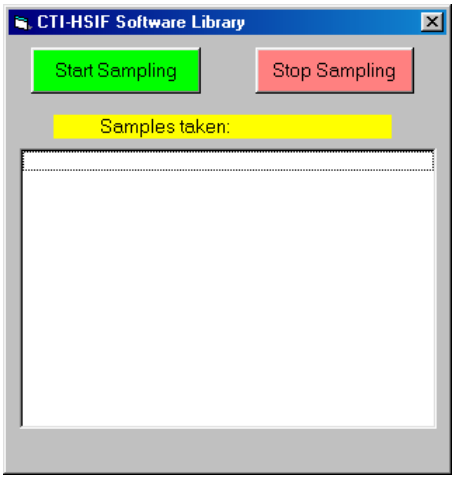
J. This line checks the return code from the `getSamples()` call. If the function succeeds, it will return the number of samples actually acquired. If an error occurs, then a negative number, representing an error code, will be returned. A list of error codes is defined in the "`CTI_ErrCodes.h`" header file. If an error does occur, the communications link to the sensor is closed, and the program exits.

K. This line terminates communications with the sensor. The library will close the serial communications port, shut down the background data acquisition threads, and free any resources used internally by the library. This function must be called by an application program to cleanly terminate use of the CTI-HSIF-PCI library. *Failure to call this function before the program exits may result in undesirable behaviour, including memory leaks, and difficulty re-establishing communications with the sensor during subsequent runs of the application program.*

L. The `setReleaseHandle()` function frees any internal library resources used by the sensor referenced by `sensorHandle`. Following this call, `sensorHandle` is invalid, and must not be used in any other library calls.

M. These lines print out the samples returned by the `getSamples()` function.

Hello World In Visual Basic



NOTE: The Visual Basic "Hello World" example is provided on the Windows operating system only.

The example program displays a window as shown at right.

When the "Start Sampling" button is pressed, the program opens communications to the sensor and high-speed interface, and acquires range samples at the sensor's factory default rate. Each range sample is added to the list box in the lower half of the window. The total number of data points taken is displayed in the text field above the list box. Pressing the "Stop Sampling" button turns off the laser and closes the communications to the sensor.

The program's source code, with further explanations, is shown below.

```
    Option Base 1
    Option Explicit
A   Private mySensor As CTI_HSIF
B   Private bExitFlag As Boolean
C   Private Sub cmdStart_Click()
D      Set mySensor = New CTI_HSIF_PCI
       Dim rc As Long

E      rc = mySensor.setDriverOpen()
F      If rc <> CTI_SUCCESS Then
          MsgBox "ERROR: setDriverOpen returned " + Str(rc), vbCritical
          Exit Sub
       End If

G      rc = mySensor.setCalibrationFile("C:\LOOKUPHS")
       If rc <> CTI_SUCCESS Then
          MsgBox "ERROR: setCalibrationFile returned " + Str(rc), vbCritical
          Exit Sub
       End If

H      rc = mySensor.setCommOpen("COM1", 9600)
       If rc <> CTI_SUCCESS Then
          MsgBox "ERROR: setCommOpen returned " + Str(rc), vbCritical
          Exit Sub
       End If
       MsgBox "Successfully opened the serial port. Starting sampling.", vbInformation

I      Dim rangeData(2000) As HSIF_DATA_PT
       Dim totSamples As Long
       Dim i As Integer

J      bExitFlag = False

       rc = mySensor.setClearBuffer()      ' Always returns CTI_SUCCESS


       totSamples = 0
K      Do
L         rc = mySensor.getSamples(rangeData, 10, 1000)
M         If rc < 0 Then
             MsgBox "ERROR: getSamples returned " + Str(rc), vbCritical
             bExitFlag = True
          Else
N            For i = 1 To rc
```

```
O              ListRange.AddItem (rangeData(i).R_X)
               ListRange.ListIndex = ListRange.ListCount - 1
               totSamples = totSamples + 1
P              lblTotSamples.Caption = Str(totSamples)
Q              DoEvents
               If (bExitFlag = True) Then Exit Do
            Next i
R        End If
      Loop While (bExitFlag = False)

S     rc = mySensor.setCommClosed()
      If rc <> CTI_SUCCESS Then
         MsgBox "ERROR: setCommClosed returned " + Str(rc), vbCritical
      Else
         MsgBox "Closed the serial connection.", vbInformation
      End If

   End Sub

   ------------------------------------------------------------------------
   Private Sub cmdStop_Click()
T     bExitFlag = True
   End Sub

   ------------------------------------------------------------------------
   Private Sub Form_Unload(Cancel As Integer)
U     bExitFlag = True
   End Sub
```

The points below describe what each line of the program is doing. Please reference the corresponding line numbers shown to the left of the code listing above.

A. This line, the two preceding lines, and the following line are in the Visual Basic form's "Declarations" section. This line declares a variable of type "CTI_HSIF_PCI". Since it is in the form's "Declarations" section, the variable is accessible to all methods of the form, or any of the controls on the form. Note that at this point, the variable does not yet refer to any specific AR4000 sensor or high-speed interface that may be attached to the computer.

B. This line declares a boolean flag used to determine when to stop taking samples from the sensor.

C. This is the "Click" method of the "Start Sampling" button. When the button is clicked this method is executed, and the code establishes communications with the sensor and collects range data.

D. This line creates a new instance of a CTI_HSIF_PCI object, and assigns it to the `mySensor` variable.

E. This line opens communication to the CTI-HSIF-PCI library. It must be the first function called when using the library.

F. This line checks the return code from the `setBoardParams()` call. It is *essential* that every method's return code be checked for success or failure.

G. This line sets the calibration file to use to calibrate samples from the high speed interface.

H. This line uses the `setCommOpen()` method to open the communications port (serial port) named "COM1", using a baud rate of 9600 baud, and establish communications with the sensor attached to this port. If successful, the method's return code is CTI_SUCCESS, otherwise an error code will be returned.

I. This line declares an array "`rangeData`" to hold the samples returned from the library. Each sample is of type 'HSIF_DATA_PT' (See the Reference Manual for a definition of the HSIF_DATA_PT type). The "`totSamples`" variable on the next line records the total number of samples acquired from the sensor.

J. This line sets the exit flag "`bExitFlag`" to false. Pressing the "Stop Sampling" button sets it to true.

K. This is the start of a continuous loop that retrieves range samples and displays them in the list box.

L. This line tells the CTI-HSIF-PCI library to acquire at most 100 samples (the size of the `rangeData` array), and at least 10 samples from the high speed interface, and return the results in the array `rangeData`. The library will wait a maximum of one second (1000 milliseconds) for the samples to be available. See the `getSamples()` function description in the Reference Manual for more details.

M. This line checks the return code from the `getSamples()` call. If the function succeeds, it will return the number of samples actually acquired. If an error occurs, then a negative number, representing an error code, will be returned. A list of error codes is defined in the "`CTI_ErrCodes.h`" header file.

N. This loop processes each sample returned from the `getSamples()` call.

O. This line adds each range value to the list box.

P. This line updates the form's total samples counter label.

Q. This line permits Visual Basic to process other events (such as the "Stop Sampling" button being pressed). When samples are acquired in a continuous loop, as shown here, Visual Basic's DoEvents method must be called within the loop, otherwise Visual Basic is unable to process other user interactions (button presses, etc.) with the application.

R. If `bExitFlag` becomes true (when the "Stop Sampling" button is pressed), the loop will terminate.

S. This line terminates communications with the sensor. The library will close the serial communications port, shut down the background data acquisition threads, and free any resources used internally by the library. This function must be called by an application to cleanly terminate use of the CTI-HSIF-PCI library. *Failure to call this function before the program exits may result in undesirable behaviour, including memory leaks, and difficulty re-establishing communications with the sensor during subsequent runs of the application program.*

T. This line in the "Stop Sampling" button's "Click" method sets the exit flag to true.

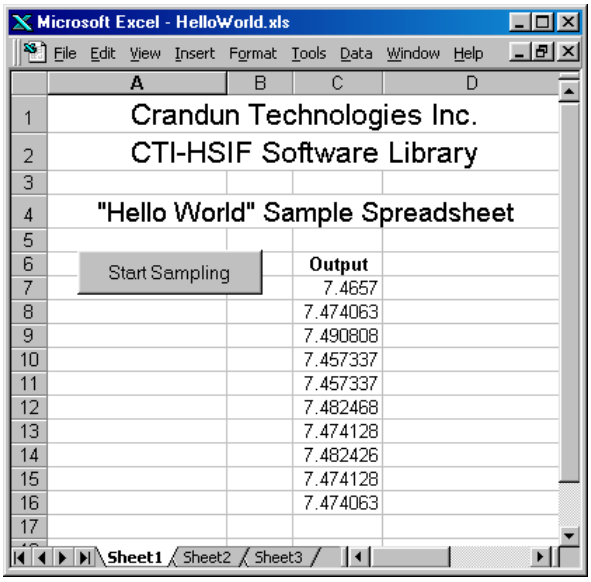
U. If the form is closed before the "Stop Sampling" button is clicked, this line will also set the exit flag to true.

Hello World In Excel

NOTE: The Excel "Hello World" spreadsheet is provided on the Windows operating system only.

The example spreadsheet is shown at right. When the "Start Sampling" button is pressed, the program opens communications to the sensor, and acquires 10 range samples that are displayed under the 'Output' heading.

The spreadsheet's VBA (Visual Basic for Applications) source code, with further explanations, is shown below. (Note: for clarity, some error handling code included in the installed example is removed from the listing below.)




```
A   Private mySensor As CTI_HSIF_PCI

B   Sub CommandStartSampling_Click()

C      Set mySensor = New CTI_HSIF_PCI    ' The sensor instance

       Dim rc As Long
       Dim numSamples As Long
       Dim outRange1 As range
       Dim c As range

       'Open driver
D      rc = mySensor.setDriverOpen()
E      If rc <> CTI_SUCCESS Then
          MsgBox "ERROR: setDriverOpen returned " + Str(rc), vbCritical
          Exit Sub
       End If

       'Define location of calibration data file
F      rc = mySensor.setCalibrationFile("C:\LOOKUPHS")
       If rc <> CTI_SUCCESS Then
          MsgBox "ERROR: setCalibrationFile returned " + Str(rc), vbCritical
          Exit Sub
       End If

       'Open the serial link to the sensor
G      rc = mySensor.setCommOpen("COM1", 9600)
       If rc <> 0 Then
          MsgBox "ERROR: setCommOpen returned " + Str(rc), vbCritical
          Exit Sub
       End If

       MsgBox "Successfully opened the serial port. Starting sampling.", vbInformation

H      Dim rangeData(100) As HSIF_DATA_PT
       Dim totSamples As Long
       Dim i As Integer

I      rc = mySensor.setClearBuffer()   'Always returns success
```

```
J       rc = mySensor.getSamples(rangeData, 10, 10000)   'Get 10 samples
K       If rc < 0 Then
           MsgBox "ERROR: getSamples returned " + Str(rc), vbCritical
        Else
           ' Successfully read samples - now display them
           MsgBox "Read " & Str(rc) & " samples", vbInformation
           Set outRange1 = Worksheets("Sheet1").range("Output_Range")
           i = 1
L          For Each c In outRange1
              c.Value = rangeData(i).R_X
              i = i + 1
              If i > rc Then Exit For
           Next c
           DoEvents
        End If

M       rc = mySensor.setCommClosed()
        If rc <> 0 Then
           MsgBox "ERROR: setCommClosed returned " + Str(rc), vbCritical
        Else
           MsgBox "Closed the serial connection.", vbInformation
        End If


    End Sub
```

A. This line declares a variable of type "CTI_HSIF_PCI". Note that at this point, the variable does not yet refer to any specific AR4000 sensor and high-speed interface that may be attached to the computer.

B. This is the "Click" method of the "Start Sampling" button. When the button is clicked this method is executed, and the code establishes communications with the sensor and collects range data.

C. This line creates a new instance of a CTI_HSIF_PCI object, and assigns it to the mySensor variable.

D. This opens communications to the library driver.

E. This line checks the return code from the `setDriverOpen()` call. It is *essential* that every method's return code be checked for success or failure.

F. This line sets the calibration file to use to calibrate samples from the high speed interface.

G. This line uses the `setCommOpen()` method to open the communications port (serial port) named "COM1", using a baud rate of 9600 baud, and establish communications with the sensor attached to this port. If successful, the method's return code is CTI_SUCCESS, otherwise an error code will be returned.

H. This line declares an array "`rangeData`" to hold the samples returned from the software library. Each sample is of type 'HSIF_DATA_PT' (See the Reference Manual for a definition of the HSIF_DATA_PT type).

I. This line clears the library's internal buffer of any samples that may be present.

J. This line tells the library to acquire at most 100 samples (the size of the `rangeData` array), and at least 10 samples from the sensor, and return the results in the array `rangeData`. The library will wait a maximum of ten seconds (10,000 milliseconds) for the samples to be available. See the `getSamples()` function description in the Reference Manual for more details.

K. This line checks the return code from the `getSamples()` call. If the function succeeds, it will return the number of samples actually acquired. If an error occurs, then a negative number, representing an error code, will be returned. A list of error codes is defined in the file "CTI_HSIF_PCI_Defs.bas".

L. This loop processes each sample returned from the `getSamples()` call, displaying the results.

M. This line terminates communications with the sensor. The library will close the serial communications port, shut down the background data acquisition threads, and free any resources used internally by the library. This function must be called by an application to cleanly terminate use of the CTI-HSIF-PCI library. *Failure to call this function before the program exits may result in undesirable behaviour, including memory leaks, and difficulty re-establishing communications with the sensor during subsequent runs of the application program.*

Reading Data

When acquiring data from a hardware device, such as the AR4000 sensors, the application program must have some means of determining when data is available to be read from the device. Broadly speaking, there are three ways in which this can be done:

1. The device can be polled (queried) to see if it has valid data available. If so, the application program reads the data, otherwise the application program can perform some other function (such as process already received data) and loop back at a later time to poll the device again. This is illustrated by the pseudo-code below:

```
LOOP FOREVER
   IF (device has data to read) THEN
      Read and process the data
   ELSE
      Do some other work
   ENDIF
END LOOP
```

   This method has the advantage of being simple to program. However, its disadvantages are:

   - The application must poll the device frequently in order to ensure that no data is lost while the 'other work' is being done.
   - It is typically inefficient, since the majority of the time it is likely that no data is yet available, and the frequent polling simply wastes CPU time that could be used by other processes.

2. The application program can perform a "blocking read" in which it starts a read operation that does not return control to the application program until the device has the desired amount of data ready. This is illustrated by the pseudo-code shown:

```
LOOP FOREVER
   WAIT for and READ data from device
   Process the data
END LOOP
```

   This method is also simple to program, and is typically much more efficient than the polling approach. However this method suffers from the disadvantage that the application cannot do other work while waiting for data to be available from the hardware device.

3. The application program can install a 'callback function'[2] that is called by the hardware device, or the device's driver software when data is available. This is illustrated by the pseudo-code shown:

```
INSTALL Callback function foo()
Tell device to call foo() when data is available
LOOP FOREVER
   Do some work
END LOOP

FUNCTION FOO
   READ data from device
   Process the data
END FUNCTION
```

This method can be more complex to program than the other two approaches, however it has the significant advantage that the device *tells* the application when it has data available, rather than the application program having to *ask* the device if data is ready.

For a given application, any of the above three approaches may be appropriate. The CTI-HSIF-PCI library supports all of the preceding methods of retrieving sensor data from the library's internal buffer, permitting the application developer to choose the method best suited to the particular application.

[2] For those familiar with hardware devices, a callback function is analogous to a hardware device's Interrupt Service Routine (ISR).

1. The `getNumSamples()` or the `getIsBufferAtThreshold()` functions may be used to 'poll' the library to determine if the desired number of samples are available to be read.

2. The `getSamples()` function may be used to perform a 'blocking read' and will only return control to the application program when the desired number of samples are available. This method is demonstrated in the "Hello World" example programs. (See page 8 and the "Sample Programs" section on page 31).

3. The `setCallbackFunction()` function may be used to have the CTI-HSIF-PCI library call a user defined callback function when a desired number of samples is available. This method is used in the "Callback_CPP" and "Callback_C" sample programs (see pages 31 and 32).

In general, the last two methods are preferred, since polling the library's sample counter is typically inefficient.

Note that when using the callback approach, the callback function is called within a *different thread of execution* than the program's main function. This is an important point, and is explained in more detail below.

Using Callback Functions

As explained in the "Library Conceptual Overview" section (page 4), the CTI-HSIF-PCI library continually collects samples from the high speed interface and fills its internal buffer. If a callback threshold and callback function have been set (using `setCallbackThreshold()` and `setCallbackFunction() )` then as soon as the library's buffer exceeds the callback threshold, the library will create a *new thread of execution* and call the callback function from that new thread. This permits the application program to do whatever processing is required within the callback function without affecting the internal operations of the library.

However, the programmer must be careful to take the appropriate precautions when accessing data that is shared between the callback function and other parts of the program. Mutexes, critical sections or other synchronization mechanisms *must* be used to ensure that any shared data is accessed by only one thread of execution at a time. The CTI-HSIF-PCI library itself is fully thread-safe, and any library function may be called from any thread of execution without user-supplied synchronization mechanisms.

For an in-depth explanation of multi-threaded programming considerations, it is recommended that any of the numerous suitable texts, such as "Win32 Multithreaded Programming" (ISBN 1-56592-296-5) or "Programming with POSIX® Threads" (ISBN 0-20163-392-2) be consulted.

When the callback function is called, pointers to the data within the library's internal buffer will be passed into the function. These may be used within the callback function to retrieve the sample data from the library, without making any additional calls to the library. The code sample below illustrates this technique. Please see the code listing, and the explanation following the listing for more details. (This program is also installed in the `Samples\C` directory as the "`Callback_C`" sample.) For clarity, some error checking code has been removed from the listing below. The full error handling code is included in the installed sample program.

```
    /*
     Crandun Technologies CTI-HSIF-PCI Software Library Callback Example
     Copyright (c) 2006, Crandun Technologies Inc.

     This sample program demonstrates how to use a callback function to
     retrieve sample data from the library, and write that data to a disk file

     Note that this program must be compiled and linked with the
     Visual C++ Multi-Threaded libraries, since the callback is called
     in a different thread from the main program.
    */

A   #include "CTI_HSIF_PCI.h"  /* required header */
    #include <stdio.h>
```

```
   /* File handle is global, so that both the callback and main have access */
B  FILE * my_DataFile;

   /* Declare the callback function that will be called by the library */
C  long myCallback(const HSIF_DATA_PT * pD1,
                  const long N1,
                  const HSIF_DATA_PT * pD2,
                  const long N2);

   int main()
   {
D     const char * outFileName = "C:\\CallbackTest.out";
      char c;
      long rc, sensorHandle=-1;

      printf("Crandun Technologies CTI-HSIF-PCI Library Callback example.\n");

E     sensorHandle = getNewCTIHSIF_PCI();

F     rc = setDriverOpen(sensorHandle);
      rc = setCalibrationFile(sensorHandle, "C:\\LOOKUPHS");

      printf("Opening the serial port...\n");
G     rc = setCommOpen(sensorHandle, "COM1", 9600);

      printf("Opening output data file %s\n", outFileName);
H     my_DataFile = fopen(outFileName, "w");

      printf("Testing callback function.\n");

      /* set the function "myCallback" as the callback function */
I     rc = setCallbackFunction(sensorHandle, myCallback);

      /* Tell library to call the callback when 500 samples are available */
J     rc = setCallbackThreshold(sensorHandle, 500);

      printf("Main program is sleeping for 3 seconds...\n");
K     DO_SLEEP(3);

      printf("Main program finished sleeping - closing the serial port.\n");

      /* Close the sensor serial port. This also ensures the callback is done */
L     rc = setCommClosed(sensorHandle);

M     fclose(my_DataFile);   /* close the data file */

      printf("Please enter any character to exit: ");
      scanf("%c", &c);

      return 0;
   }

   /* This is the callback function that will be called by the library */
N  long myCallback(const HSIF_DATA_PT* pD1,
                   const long N1,
                   const HSIF_DATA_PT* pD2,
                   const long N2)
   {
      long i,j;

      printf("In callback, reading %d samples. Writing to file\n", N1+N2);
```

```
O       for (i = 0, j = 0; i < N1; i++, j++, pD1++) {
           fprintf(my_DataFile, "%8.3f\n", pD1->R_X);
        }

        /* read the samples from the second data pointer, if any */
P       for (i = 0; i < N2; i++, j++, pD2++) {
           fprintf(my_DataFile, "%8.3f\n", pD2->R_X);
        }

        /* return non-zero to tell lib to remove samples from its buffer */
Q       return 1;
    }
```

The points below describe what each line of the program is doing. Please reference the corresponding line numbers shown to the left of the code listing above.

A. This line includes the header file that defines the functions and error codes of the CTI-HSIF-PCI library. This header file must be included by any program using the CTI-HSIF-PCI library.

B. This line declares the handle of the file that will be used to record the range samples from the sensor.

C. This line declares the callback function that will be called by the CTI library. The function is defined at the bottom of the program after `main()`.

D. This line defines the output file to which the range samples will be written.

E. This line calls the `getNewCTIHSIF_PCI()` function to obtain a "handle" to a particular sensor. If successful, this function will return a non-negative value that must be passed as the first parameter to all other functions when using the library from C. Note that at this point, the handle does not yet refer to any specific AR4000 sensor or high speed interface that may be attached to the computer.

F. The next two lines open communications to the library, and set the calibration file to use to calibrate samples. (The installed sample program also includes error checking code that has been omitted here for clarity.)

G. This line uses the `setCommOpen()` function to open the communications port named "`COM1`", using a baud rate of 9600 baud, and establish communications with the sensor attached to this port.

H. This line opens the output data file to which range samples will be written.

I. This line sets the function "`myCallback`" as the callback function.

J. This line sets the callback threshold at 500 samples. When the library has 500 samples in its internal buffer, it will call the callback function.

K. This line pauses the main program thread for 3 seconds (3000 milliseconds).

L. This line closes the serial communications link between the CTI library and the sensor. The library will wait up to 20 seconds to allow the callback function to complete before closing the serial port.

M. This line closes the data file before the main thread exits.

N. This line is the start of the callback function. Four parameters are passed to the callback by the library – two data pointers, and two data point counters. `pD1` points to a contiguous buffer holding `N1` range samples, and `pD2` points to a second contiguous buffer holding `N2` range samples.

O. This loop reads range samples from the first data buffer, and writes them to the data file.

P. This loop reads the samples, if any, from the second data buffer, and writes them to the file.

Q. If the callback function returns a non-zero value, the library will remove the data samples from its internal buffer. If the callback function returns zero, the samples will not be removed, and will remain in the library's internal buffer.

Key points to note are:

- If the callback function returns a non-zero value, immediately upon the callback function's return, the library will remove all samples referenced by the two data pointers from its internal buffer. These samples will be lost if the application has not already read them.

- If the callback function returns zero, the data samples will *not* automatically be removed by the library upon return from the callback function. The application programmer must be sure to remove these samples from the library buffer by some other method (such as calling `getSamples()`or `setClearBuffer()`), otherwise the library's buffer will eventually fill up and incoming sensor data will be lost.

- Since the library continues to read incoming samples from the high speed interface at the same time as the application program processes existing samples from the library's buffer, it is possible that the callback function will be called again immediately upon its return, if the library's internal buffer is still over the callback threshold. This is particularly likely if the callback function returns a zero value, so that the library does not automatically remove the samples from its buffer.

- The library guarantees that the callback function will never be called a second time while a callback is already active. Thus, the application programmer does not need to be concerned about re-entrancy issues within the callback function itself.

- The sample program shown above does not simultaneously access any data that is shared between the callback function and other parts of the program. (The data file is guaranteed never to be accessed simultaneously since it is opened before the callback is active, and closed after the callback is finished.) However, if an application program *does* access any shared data, then mutexes, critical sections or other synchronization mechanisms *must* be used to ensure that the shared data is accessed by only one thread of execution at a time.

Data Filtering

Once data samples have been read from the library, using one of the methods outlined above, the data acquisition program typically processes those samples according to the needs of the particular application.

In many data acquisition applications, determining which samples received from the measurement equipment are valid and which are not can be a major consideration. The physical constraints imposed by the measurement apparatus may result in extraneous data being received by the data collection application that must then be separated from the 'good' data before further processing is done.

For example, when using the AR4000 sensor, the requirements of a given application may dictate that only samples between 24 and 48 inches are of interest, while all others should be ignored. However, samples outside of the 24-48 inch range may also be returned by the sensor (perhaps due to objects in the background, beyond the 48-inch range), and these samples should be ignored in any further processing. Similarly, it may be desirable to ignore samples with a low amplitude or a high ambient light reading, as these may be inaccurate due to the weak return signal received by the sensor.

To facilitate this type of discrimination, the CTI-HSIF-PCI library permits filtering of the raw samples from the sensor, based on any combination of range, amplitude, ambient light, temperature reading, or angular measurement [if using the line scanner]. These functions are explained in detail in the section "Data Filtering and Transformation Functions" starting on page 51 of the Reference Manual. Use of these functions can significantly reduce the amount of processing that an application programmer must do on the raw sensor datastream.

For example, setting a minimum valid range of 24 inches using the `setMinValidRange()` function and a maximum valid range of 48 inches using the `setMaxValidRange()` function will cause the library to set any sample which does not fall between 24 and 48 inches to a zero range. Alternatively, `setDiscardInvalidOn()` tells the library to simply discard these out-of-range samples and not return them to the application program.

Similarly, when using the line scanner, only part of the full 360-degree field of view may be of interest. The `setMinValidAngle()` and `setMaxValidAngle()` functions can be used to have the library discard samples that are known to be outside of the field of interest.

During application development, some experimentation will typically be required to determine what filter criteria are appropriate. Once the appropriate filtering criteria are determined, the filtering functions and the `setDiscardInvalidOn()` function can be used to have the CTI library discard samples that are not of interest. Thus any samples actually returned to the application program will be known to be valid.

Whenever possible the filtering functionality of the CTI-HSIF-PCI library should be used to discard samples that are not of interest, as this will be much more efficient and less error-prone than doing the equivalent testing within the application program.

Encoder Support

The high speed interface provides two encoder counters that record the encoder position simultaneously with the range data measurement. (See the sensor hardware documentation for more details.) When using the AccuRange Line Scanner, the position of the rotating mirror is tracked by the first encoder reader, while the other encoder counter remains available for other uses.

The CTI-HSIF-PCI library converts the raw encoder count and returns to the application a standard angular measurement, using the encoder index pulse as the zero-angle point. From the index pulse point, the angular measurements will increase in the direction that the mirror is rotating.

The zero angle reference point may be changed from the position of the encoder index pulse by using the `setAngleOffset()` function to have the CTI library add an offset to the raw angle measured.

The encoder pulses from the second encoder (if so equipped) are tracked and returned to the application in a similar manner.

Using the Line Scanner

The AR4000 sensor system may optionally be equipped with the AccuRange Line Scanner (please see the sensor hardware documentation).

The line scanner provides a precision rotating mirror, whose speed may be controlled using the CTI library's `setMotorPower()` or `setMotorRPM()` functions. The angular position of the mirror is tracked by the attached encoder, and is returned to the application program, along with the range data, by the `getSamples()` function.

When using high sampling rates with relatively low line scanner rotation speeds, two or more data samples may be taken for the same encoder reading. When this situation occurs, the CTI-HSIF-PCI library will interpolate the angle between the data points to increase the apparent angular resolution of the data points. This will occur automatically, without any special action on the part of the user.

Performance Considerations

To achieve the best performance from the CTI-HSIF-PCI library and high-speed interface hardware, the user or application programmer should consider the following:

- Use a sample rate only as high as required to achieve the desired application results. A higher than necessary rate imposes additional overhead on the library, increasing the chance that the application program may not be able to process the data fast enough. Calculate the sample rate required carefully. For example, if the application is to scan items moving past the sensor at 10 meters/second, then a sample rate of 5,000 samples/second results in scanning the items at 500 samples per meter or 5 samples per centimeter. If this sample density is sufficient for the needs of the application, a higher sample is undesirable, for the reasons mentioned above.

- Whenever possible, use the library's filtering functions to discard unwanted or 'out of range' samples. The library immediately discards, and does not further process, any sample that fails a filter criterion. This can result in a performance improvement by avoiding relatively time-consuming operations (such as calibration of the raw range) that would otherwise be done on the raw data.

  This is particularly useful when only a small percentage of the total samples are required. For example, when using the line scanner, only a small portion of the full 360-degree field of view may be of interest.

- Use the `setBufferSize()` function to allocate as large a buffer as can be accommodated within the available memory of the computer used. A large library buffer gives the application programmer more leeway in processing samples from the library without being concerned with the library's buffer overflowing. (Note however that allocating too large a buffer may use up all the physical memory on the computer, and result in paging to disk as the virtual memory manager takes effect. This will cause a severe loss in performance.)

- In general, at high sample rates, application performance will be better from C or C++, rather than Visual Basic or Visual Basic for Applications (Excel). Although the internal performance of the CTI library is identical regardless of the language used, the overhead of the Visual Basic and VBA runtime systems can be avoided by using C or C++.

The CTI-HSIF-PCI Library has been written with performance as a primary objective. Careful attention to the above points will provide the best results for users of the library.

Recommended Usage

As documented in the Reference Manual, some of the functions of the CTI-HSIF-PCI library have side-effects, such as clearing the library's internal buffer, resetting sensor values, etc.. Thus, it is recommended that application programs using the CTI-HSIF-PCI Library adhere to the following sequence of steps in order to make most effective use of the library.

1. If using the library from a "C" language program, use `getNewCTIHSIF_PCI()` (see Reference Manual page 2) to obtain a sensor instance handle.

2. Open communications to the library driver using `setDriverOpen()` (Reference Manual, page 3).

3. Set the calibration file using `setCalibrationFile()` (Reference Manual, page 4).

4. Set the encoder counts per revolution, using `setEncoderCountsPerRev()` (Reference manual, page 6).

5. If using a non-Windows version of the library, call `setKeycodeFile()` (Reference manual, page 7) to specify the license keycode file.

6. Open the serial port using the `setCommOpen()` function (see page 9 of the Reference Manual).

7. Set the sensor's sampling interval using `setSampleInterval()` or `setSamplesPerSec()` (Reference Manual pages 38 and 39).

8. If desired, set the library's buffer size using the `setBufferSize()` function. (Reference Manual page 13).

9. If callbacks will be used, set the callback function and callback threshold using `setCallbackFunction()` and `setCallbackThreshold()` (Reference Manual pages 16 and 17).

10. Set the range offset and range scale factors, if any, using `setRangeOffset()` and `setRangeScaleFactor()` (Reference Manual pages 65 and 66).

11. Set any filtering parameters using `setMinAmplitude()`, `setMaxRange()`, etc. (See the "Data Filtering" section on page 25 of the Programmer's Guide and the "Data Filtering and Transformation Functions" section, starting on page 51 of the Reference Manual.)

12. Set continuous sampling on using the `setContinuousHSIFOn()` function (Reference Manual page 40). (By default continuous serial samples will be on, so this step may be superfluous.)

13. Clear the library's buffer using `setClearBuffer()` (Reference Manual, page 14).

14. Retrieve samples using `getSamples()` (Reference Manual page 42) or by reading from the data pointers passed to the callback function, and process those samples as required by the application.

Common Operations

Many of the common operations that an application programmer will want to do when using the CTI-HSIF-PCI library are illustrated by the sample programs shipped with the software, as indicated below.

Writing Data to a File

This technique is illustrated in the `Callback_C` and `Callback_CPP` example programs.

Changing the Sampling Rate

Changing the sensor's sampling rate is illustrated in the `TwoSensors_C` and `TwoSensors_CPP` programs.

Changing the Serial Port Baud Rate

Changing the serial communications link's baud rate is shown in the program `BaudRate_C` and `BaudRate_CPP` example programs. However, when using the AR4000 sensor with the high speed interface, all data from the sensor is returned via the high speed interface card, not the serial interface, so it is strongly recommended that the serial port baud rate be left at the factory default of 9600 baud.

Dealing with Spurious Samples

The `Filtering_C` and `Filtering_CPP` sample programs and `Filtering` spreadsheet demonstrate use of the filtering functionality to have the library discard spurious samples.

Do's and Don'ts

Below are some "do's and don'ts" recommendations for use of the CTI-HSIF-PCI software library. Following these guidelines is strongly recommended to simplify use of the library, maximize the performance of the resulting application, and minimize difficulties in using the library.

Do's

1. Call the `setDriverOpen()`, `setCalibrationFile()` and `setEncoderCountsPerRev()` functions before calling `setCommOpen()`. These functions supply the information needed to properly communicate with the sensor and high speed interface, and must be the first functions called in any program using the library. (C language programmers must call `getNewCTIHSIF_PCI()` first to get a valid sensor handle.)

2. Ensure that the `initialBaudRate` parameter of the `setCommOpen()` function matches the actual baud rate of the sensor's serial port. Since the library has no means of determining the current actual baud rate setting of the sensor, the `initialBaudRate` parameter specified ***must*** match the actual baud rate of the sensor, otherwise communications with the sensor will fail.

3. Call the `setCommClosed()` function before exiting the application program. This will ensure that the sensor's serial port is reset to 9600 baud, and that any resources used by the library are freed. *Failure to call this function before the program exits may result in undesirable behaviour, including memory leaks, and difficulty re-establishing communications with the sensor during subsequent runs of the application program.*

4. Open the serial communications port to the sensor only once in the program and keep it open throughout the program. Calling the `setCommOpen()` function is a relatively expensive operation, as this function starts up the background threads used to process incoming data from the sensor, then opens the serial port to the sensor, resets the sensor to factory defaults (changing the computer's serial port baud rate to 9600 baud, if necessary), and sets the communications with the sensor to binary mode.

   Although it is not wrong to open and close the serial port multiple times, the overhead of these relatively time-consuming initialization steps can be avoided if an application program opens the serial port only once.

5. Ensure that the correct calibration file is used. If the calibration file specified does not correspond to the actual sensor being used, then the calibrated range data returned by the library will be inaccurate.

6. Do use the data filtering functionality provided by the library, rather than perform this function in the application program. The library provides a wide range of highly optimized data filtering options, and use of these functions is much more efficient and less error-prone than writing equivalent functionality within the application.

7. Ensure that appropriate concurrency mechanisms (mutexes, semaphores, etc.) are used to protect any global data in your program that is accessed from within a callback function. Remember that the callback function executes in a different thread than the rest of the program.

8. Ensure that the callback function returns a non-zero value to have the library remove the samples from its internal buffer, or provide some other mechanism for removing the samples from the library's buffer.

Don'ts

1. In general, do not poll the library for the availability of data in the library's input buffer. Usually this will result in poor performance. Instead, either one of the following two techniques is recommended:

   a) Specify a minimum number of desired samples using the `getSamples()` functions. If the required number of samples is not immediately available in the library's buffer, the library will enter an efficient wait state until all samples are available, then return to the application program.

b) Use the functionality provided by `setCallbackFunction()` and `setCallbackThreshold()` to have the library call a function within the application program when the desired number of samples is ready. See the sample programs provided for an example of using the library's callback functionality.

2. Do not change the serial port baud rate from the factory default of 9600 baud, unless there is a compelling reason to do so.

   When using the AR4000 sensor with the high speed interface, all data is returned via the high speed interface card, not the serial interface. Increasing the serial port baud rate will not improve the sampling speed, so it is strongly recommended that the serial port baud rate be left at the 9600 baud factory default.

3. [Applicable to C language programmers only.] Do not repeatedly call `getNewCTIHSIF_PCI()` without making corresponding calls to `setReleaseHandle()`. If the maximum number of handles supported by the library (currently 100) have already been allocated by calling `getNewCTIHSIF_PCI()` without corresponding calls to `setReleaseHandle()`, then future calls to `getNewCTIHSIF_PCI()` will fail.

   In general, a C program should allocate all the sensor handles required at the start of execution, and deallocate them by calling `setReleaseHandle()` at the end of the program.

4. Do not assume that function calls to the library succeed. Check *each* function call's return code for success or failure.

Sample Programs

All sample programs are installed in the "Samples" subdirectory of the installation directory. A compiled copy of each program, along with complete source code for the program is included. We continually develop new sample programs in response to customer inquiries. Please check our web site at www.crandun.com for other sample programs in addition to those listed here.

Note that many of the sample programs assume that the AR4000 sensor's serial cable is attached to the computer's COM1 serial port, and that the sensor's serial communications rate is 9600.

If the sensor is configured differently, the compiled version of the program will not run successfully. In this case, the program should be recompiled from the source code provided to reflect the actual sensor configuration used.

C++ Language Examples

All C++ sample programs may be recompiled from the source code provided by using:

- for Linux: the Makefile provided. Type '`make program_name`' (without the quotes). For example `make Hello_World_CPP`
- for Windows: the Visual C++ project file included in the directory.

Hello_World_CPP

The C++ language "Hello World" program documented on page 8 is installed in the `Samples\CPP` subdirectory.

Callback_CPP

This program is installed in the `Samples\CPP` subdirectory, and demonstrates using the library's callback functionality to efficiently acquire samples and write the data to a disk file.

Filtering_CPP

This program is installed in the `Samples\CPP` subdirectory, and demonstrates using the library's filtering functions to discard samples outside of a specified range.

Motor_CPP

This program is installed in the `Samples\CPP` subdirectory. It demonstrates using the high-speed interface and the AccuRange Line Scanner to gather range samples with corresponding angle measurements.

TwoSensors_CPP

This program is installed in the `Samples\CPP` subdirectory, and demonstrates acquiring and displaying range data from two sensors, using two high speed interfaces simultaneously.

C Language Examples

All C++ sample programs may be recompiled from the source code provided by using:

- for Linux: the Makefile provided. Type '`make program_name`' (without the quotes). For example `make Hello_World_CPP`
- for Windows: the Visual C++ project file included in the directory.

Hello_World_C

The C language "Hello World" program documented on page 9 is installed in the `Samples\C` subdirectory.

Callback_C

This program is installed in the `Samples\C` subdirectory, and demonstrates using the library's callback functionality to efficiently acquire samples and write the data to a disk file.

Filtering_C

This program is installed in the `Samples\C` subdirectory and demonstrates using the library's filtering functions to discard samples outside of a specified range.

Motor_C

This program is installed in the `Samples\C` subdirectory. It demonstrates using the high-speed interface's motor control and encoder to gather range samples with corresponding angle measurements. You must have the Acuity AccuRange Line Scanner installed to use this example.

TwoSensors_C

This program is installed in the `Samples\C` subdirectory and demonstrates acquiring and displaying range data from two sensors, using two high speed interfaces simultaneously.

Visual Basic Language Examples

NOTE: Visual Basic examples are provided on the Windows operating system only.

HelloWorld

The Visual Basic language "Hello World" program documented on page 12 is installed in the `Samples\VB` subdirectory. The program may be run by double-clicking the executable file from the Windows Explorer and the program's project file `HelloWorld.vbp` may be opened directly in Microsoft Visual Basic.

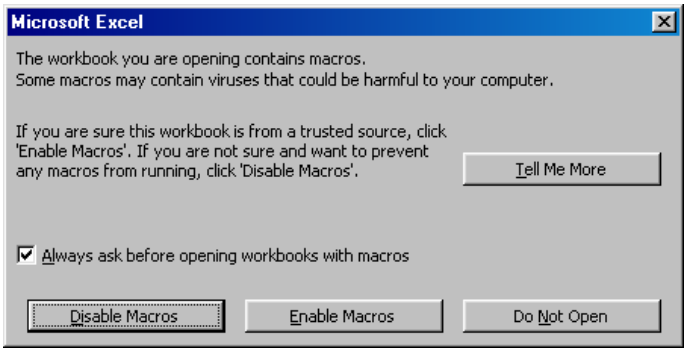
Filtering

This program is installed in the `Samples\VB` subdirectory. The program may be run from a Command Prompt window, or by double-clicking the executable file from the Windows Explorer. The program's project file `Filtering.vbp` may be opened directly in Microsoft Visual Basic. This program demonstrates using the library's filtering functions to discard samples outside of a specified range.

Microsoft Excel Examples

NOTE: Excel examples are provided on the Windows operating system only.

When opening any of the sample spreadsheets, Excel may display a window similar to that shown at right, warning that the spreadsheet contains macros. This is normal. Excel is referring to the Visual Basic for Applications (VBA) code that communicates with the sensor. The sample spreadsheets do not contain any harmful macros, and may be opened safely.



HelloWorld

The Microsoft Excel "Hello World" spreadsheet documented on page 17 is installed in the `Samples\Excel` subdirectory. The spreadsheet may be opened by clicking on the file "`HelloWorld.xls`" from a Windows Explorer window, or by opening the file from Excel's File Open menu.

Filtering

This spreadsheet is installed in the `Samples\Excel` subdirectory, and may be opened by clicking the file "`Filtering.xls`" from the Windows Explorer, or by opening the file from Excel's File Open menu. This spreadsheet demonstrates using the library's filtering functions to discard samples outside of a specified range.

Motor

This spreadsheet is installed in the `Samples\Excel` subdirectory, and may be opened by clicking the file "`Motor.xls`" from the Windows Explorer, or by opening the file from Excel's File Open menu. This spreadsheet demonstrates using the high-speed interface's motor control and encoder to gather range samples with corresponding angle measurements. You must have the Acuity AccuRange Line Scanner installed to use this example.

TimeSeries

This spreadsheet is installed in the `Samples\Excel` subdirectory, and may be opened by clicking the file "`TimeSeries.xls`" from the Windows Explorer, or by opening the file from Excel's File Open menu. This spreadsheet demonstrates using the library to collect time series data (i.e. a sequence of samples for a particular time period).

Building a Program

Building a C or C++ Program on Windows

This section gives a brief overview of the basic steps involved in building a program that uses the CTI-HSIF-PCI library from either C or C++. This is not intended to be an in-depth programming tutorial. It is assumed that application developers using the CTI-HSIF-PCI library are fully familiar with the programming language being used, and how to develop and debug programs using the editor, compiler, debugger and other tools required.

The steps required to develop a program that uses the CTI-HSIF-PCI library from either C or C++ are:

1. Create the source code of the program using Microsoft's Visual Studio, or other text editor. (If using Visual Studio, you may wish to use one of the project files in the Samples directory as a starting point.)

2. Use a `"#include"` statement to include the header file `CTI_HSIF_PCI.h` within any source file that calls a library function. `CTI_HSIF_PCI.h` may be found in the `Source\C_CPP` directory under the Library's installation directory. (The default installation directory is "`C:\Program Files\Crandun Technologies\CTI-HSIF-PCI`") You should not modify `CTI_HSIF_PCI.h` in any way.

3. When linking the program, include the file `CTI_HSIF_PCI_DLL.lib` in the list of files to link. This file is also found in the `Source\C_CPP` directory under the Library's installation directory. `CTI_HSIF_PCI_DLL.lib` contains definitions of all the functions exported by `CTI_HSIF.DLL` and must be linked with your project in order to call the functions in `CTI_HSIF.DLL.`

4. When running the program, the file `CTI_HSIF_PCI_DLL.DLL` must be on the DLL search path. By default, this file is installed in the `Windows\System` directory, and will be found there.

Building a Visual Basic Program

This section gives a brief overview of the steps involved in building a program that uses the CTI-HSIF-PCI library from Microsoft Visual Basic. This is not intended to be an in-depth programming tutorial. It is assumed that application developers using the CTI-HSIF-PCI library are fully familiar with Visual Basic, and how to develop and debug programs using the Visual Basic development environment. The steps required to develop a program that uses the CTI-HSIF-PCI library from Visual Basic are:

1. Create a new Visual Basic project, and create your forms and corresponding code as with any other project. Alternatively, you may open one of the sample Visual Basic projects from the `Samples` directory, and use it as a starting point.

2. Include the file `CTI_HSIF_PCI.cls` in the project as a Class Module. This is done using the "Project Add Class Module" menu option. The file `CTI_HSIF_PCI.cls` may be found in the `Source\VB` directory under the Library's installation directory. (The default installation directory is "`C:\Program Files\Crandun Technologies\CTI-HSIF-PCI`")

   Depending on your version of Visual Basic, the procedure for inserting Class Modules may vary. Please consult the appropriate Microsoft documentation.

   You should not modify `CTI_HSIF_PCI.cls` in any way.

3. Include the file `CTI_HSIF_PCI_Defs.bas` in the project as a Code Module. This is done using the "Project Add Module" menu option. The file `CTI_HSIF_PCI_Defs.bas` may be found in the `Source\VB` directory under the Library's installation directory.

Depending on your version of Visual Basic, the procedure may vary. Please consult the appropriate Microsoft documentation.

You should not modify `CTI_HSIF_PCI_Defs.bas` in any way.

4. When running the program, either from within the VB development environment, or as a compiled executable program, the file `CTI_HSIF_PCI_DLL.DLL` must be on the DLL search path. By default, this file is installed in the `Windows\System` directory, and will be found there.

Building a VBA Program using Excel

This section gives a brief overview of the steps involved in building a program that uses the CTI-HSIF-PCI library from Microsoft Excel using Visual Basic for Applications (VBA). This is not intended to be an in-depth programming tutorial. It is assumed that users developing custom spreadsheets using the CTI-HSIF-PCI library are fully familiar with Excel, Visual Basic for Applications programming from Excel, and how to develop and debug programs using the VBA development environment.

The steps required to develop a program that uses the CTI-HSIF-PCI library from Excel are:

1. Create a new Excel spreadsheet, and create your controls (buttons, etc.) and corresponding code. Alternatively, you may open one of the sample spreadsheets from the `Samples\Excel` directory, and use it as a starting point.

2. Include the file `CTI_HSIF_PCI.cls` in the VBA project as a Class Module, using the "File Import File" menu option. This will open a dialog box to select the file to insert. Select the file `CTI_HSIF_PCI.cls` which may be found in the `Source\VBA` directory under the Library's installation directory. (The default installation directory is "`C:\Program Files\Crandun Technologies\CTI-HSIF-PCI`").

   Depending on your version of Excel, the procedure for inserting Class Modules may vary. Please consult the appropriate Microsoft documentation.

   You should not modify `CTI_HSIF_PCI.cls` in any way.

3. Using the same procedure as the previous step, include the file `CTI_HSIF_PCI_Defs.bas` in the project as a standard (Code) Module. `CTI_HSIF_PCI_Defs.bas` may be found in the `Source\VB` directory under the Library's installation directory.

   You should not modify `CTI_HSIF_PCI_Defs.bas` in any way.

4. When running the program, either from within the VBA development environment, or from within Excel, the file `CTI_HSIF_PCI_DLL.DLL` must be on the DLL search path. By default, this file is installed in the `Windows\System` directory, and will be found there.

Distributing Software Created using the Library

Please note: The End-User License Agreement you agreed to during installation of this software defines the terms under which components distributed with the CTI-HSIF-PCI library may be redistributed. This section of the manual discusses specifically which components of the library may be redistributed.

However, at all times, the End-User License Agreement remains the definitive document for determining permissible uses of any component of the CTI-HSIF-PCI Library. Nothing in this manual should be interpreted as modifying the terms of the End-User License Agreement.

The components listed in the following table may be redistributed as part of a value-added application that uses the CTI-HSIF-PCI library. ***No other files or components of the CTI-HSIF-PCI Library package may be redistributed.***

If you distribute an application or library that incorporates the CTI-HSIF-PCI Software Library, your application must add significant value to the API provided by the CTI-HSIF-PCI library alone, and cannot be merely a simple "wrapper" around the CTI library functionality. Please refer to the End-User License Agreement for more details.

The list of files that may be redistributed depends on the particular operating sytem being used. Please see Appendix B for details of the specific files for each operating system.

Part II - Reference Manual

Introduction

This Reference Manual describes each of the functions provided by the CTI-HSIF-PCI library. Logically related functions are grouped together, and each function's prototype, purpose, parameters, and return value is provided. NOTE: Visual Basic and Visual Basic for Applications (VBA) functions are available on Windows only.

Function Parameters

In most cases, the parameters required by each function are identical, regardless of the language used. In the few cases where differences exist, the notation [C], [C++] and [VB] denotes the parameters required from C, C++, and Visual Basic respectively. Unless otherwise noted, all comments regarding Visual Basic also apply to VBA.

Function Return Values and Status Codes

As documented in the descriptions below, many library functions return a status code to indicate success or failure. These codes are defined in the files `CTI_ErrCodes.h` (for C and C++ programs), and in `CTI_HSIF_PCI_Defs.bas` (for Visual Basic programs). The appropriate file should be included in any programs using the library.

Library Configuration Functions

The configuration functions are used to initiate communications with the sensor and high speed interface, configure the library's internal buffer size, and control the library's callback functionality.

getNewCTIHSIF_PCI

Return a new handle to a particular sensor instance.

```
C:     long getNewCTIHSIF_PCI()

C++:   This function should not be used from C++.

VB:    This function should not be used from Visual Basic.
```

PARAMETERS

None.

RETURN VALUES

If successful, this function will return a non-negative value that is a "handle" to a particular sensor. The handle is valid for the duration of the process in which it was allocated, and must be passed as the first parameter to all other functions when using the library from C.

Sensor handles that are no longer needed should be released by calling `setReleaseHandle()(see` page 3). If the maximum number of sensors supported by the library (currently 100) are already in use by this process, or if sufficient memory could not be allocated, CTI_FAILURE is returned.

COMMENTS

This function *must* be the first function called when using the CTI libraries from C.

The handle value is valid across all threads in the process and may be used by any thread in calls to the library functions. The CTI-HSIF-PCI library is fully thread-safe, and any library function may be called from any thread of execution without user-supplied synchronization mechanisms.

setReleaseHandle

Release a sensor handle.

```
C:     long setReleaseHandle(const long sensorHandle)

C++:   This function should not be used from C++.

VB:    This function should not be used from Visual Basic.
```

PARAMETERS

`sensorHandle` – the sensor handle to release. This must be a valid sensor handle, previously obtained by calling `getNewCTIHSIF_PCI()`.

RETURN VALUES

CTI_BAD_PARAM if `sensorHandle` is not a valid handle previously obtained by calling `getNewCTIHSIF_PCI()`, or if the handle has already been released. CTI_SUCCESS otherwise.

COMMENTS

This function calls `setCommClosed()` for the specified sensor handle (see page 10 for a description of the effect of `setCommClosed()`), then releases any internal library resources used by the sensor referenced by `sensorHandle`. Following this call, `sensorHandle` is invalid, and must not be used in any other library calls.

When a sensor handle is no longer needed, it is a good practice to call `setReleaseHandle()`, to return the `sensorHandle` to the library's internal pool of available handles to be allocated. If repeated calls to `getNewCTIHSIF_PCI()` are made without corresponding calls to `setReleaseHandle()`, then the library's pool of sensor handles will eventually be exhausted, and future calls to `getNewCTIHSIF_PCI()` will fail.

setDriverOpen

Opens communications with the library's internal driver for the high speed interface card.

```
C:     long mySensor.setBoardParams(const long sensorHandle)

C++:   long mySensor.setBoardParams()

VB:    mySensor.setBoardParams() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if successful.

CTI_BAD_PARAM if either parameter is not one of the valid values supported by the high speed interface.

COMMENTS

This function must be called before `setCommOpen()` is called.

setCalibrationFile

Tell the CTI-HSIF-PCI library which file contains the high speed interface's calibration data.

```
C:     long mySensor.setCalibrationFile(const long sensorHandle,
                                        const char * filename)

C++:   long mySensor.setCalibrationFile(const char * filename)

VB:    mySensor.setCalibrationFile(ByVal filename As String) As Long
```

PARAMETERS

`filename` – the fully-qualified filename of the calibration data file corresponding to the particular high speed interface board being used. This data file is shipped by Schmitt Measurement Systems, with the high speed interface.

RETURN VALUES

CTI_SUCCESS if the specified calibration file can be successfully opened and read.

CTI_CANNOT_OPEN if the file cannot be opened. This may occur if the specified filename is incorrect (a fully qualified filename must be given), or if the calling process does not have permission to open the file. `getErrorMessage()` may be used to get an extended error message.

CTI_BAD_PARAM if the file contents cannot be read successfully. This may be caused by a corrupted file. Ensure that the file specified is the one supplied with the high speed interface hardware.

COMMENTS

This function will attempt to open and read the contents of the specified calibration data file.

Each high speed interface board has a *different* calibration data file. The file specified *must* match the particular high speed interface used, otherwise the calibrated sample data returned by the library will be inaccurate.

getCalibrationFile

Get the name of the high speed interface's calibration data file, as set by `setCalibrationFile()`.

```
C:     long mySensor.getCalibrationFile(const long sensorHandle,
                                        char * pFileName,
                                        const long buflen)

C++:   long mySensor.getCalibrationFile(char * pFileName,
                                        const long buflen)

VB:    mySensor.getCalibrationFile(filename As String) As Long
```

PARAMETERS

`pFilename` – [C, C++] Pointer to a buffer to hold the returned filename. In order to return the entire filename, the buffer should be at least as long as the filename set with the `getCalibrationFile()` function.

`filename` – [VB]. A String variable to hold filename returned from the library.

`buflen` – [Not required from Visual Basic]. The length of the buffer pointed to, in characters.

RETURN VALUES

CTI_SUCCESS if successful.

CTI_BAD_PARAM if `pFilename` is null, or `buflen` is negative.

CTI_BUFFER_TOO_SMALL if the buffer is too small to hold the entire length of the filename. In this case, as much of the filename as can fit in the actual buffer is returned.

COMMENTS

None.

setEncoderCountsPerRev

Set the number of counts per revolution of the encoder.

```
C:     long mySensor.setEncoderCountsPerRev(const long sensorHandle,
                                            const long encoderNum,
                                            const long encoderCountsPerRev)

C++:   long mySensor.setEncoderCountsPerRev(const long encoderNum,
                                            const long encoderCountsPerRev)

VB:    mySensor.setEncoderCountsPerRev(ByVal encoderNum As Long,
                                       ByVal encoderCountsPerRev As Long) As Long
```

PARAMETERS

`encoderCountsPerRev` – the number of counts per full revolution of the encoder specified by `encoderNum`. `encoderCountsPerRev` must be zero or greater.

`encoderNum` – the encoder number (1 or 2) to set the count for.

RETURN VALUES

CTI_BAD_PARAM if `encoderCountsPerRev` is less than zero, or if `encoderNum` is not 1 or 2. CTI_SUCCESS otherwise.

COMMENTS

If the encoder is not being used, this function should not be called. Alternatively, setting a value of zero for the counts per revolution disables use of the encoder.

If the encoder is to be used, the application programmer should be aware of the following considerations:

1. This function must be called before `setCommOpen()` is called.

2. The encoder index pulse is the zero-angle point. From the index pulse, angular measurements will increase in the direction of rotation of the motor.

3. When the library starts acquiring samples from the high speed interface, it will discard all samples until the first index pulse is found, thereby establishing the zero-angle reference point. The implication of this is that if `setEncoderCountsPerRev()` is called, but the motor is not started prior to `getSamples()` being called, no samples will be returned by `getSamples()` (since a sample with the index pulse will never be found). This applies to encoder number one only. If encoder number 2 is used, then all samples acquired before the encoder index pulse is found are returned with an angle reading of –FLT_MAX degrees (i.e. a very large negative number).

4. The number of counts per revolution of the encoder, as set by this function, must match the actual number of counts per revolution of the encoder being used, otherwise the angular measurements returned by the library will be inaccurate.

Normally, the angular resolution of the data points is limited by the number of encoder counts per revolution. For example, using a 4096 count encoder, the angular resolution of each sample is 1/4096 * 360 degrees (approximately 0.088 degrees). However, when using high sampling rates with relatively low line scanner rotation speeds, two or more data samples may be taken for the same encoder reading.

When this situation occurs, the CTI-HSIF-PCI library will interpolate the angle between the data points to increase the apparent angular resolution of the data points. This occurs automatically, without any special action on the part of the user.

getEncoderCountsPerRev

Return the number of counts per revolution of the encoder, as set by `setEncoderCountsPerRev()`.

```
C:     long mySensor.getEncoderCountsPerRev(const long sensorHandle,
                                            const long encoderNum)

C++:   long mySensor.getEncoderCountsPerRev(const long encoderNum)

VB:    mySensor.getEncoderCountsPerRev(ByVal encoderNum As Long) As Long
```

PARAMETERS

`encoderNum` – the encoder number (1 or 2) to get the count for.

RETURN VALUES

CTI_BAD_PARAM if `encoderNum` is not 1 or 2. Otherwise, returns the number of counts per one full revolution of the specified encoder, as set by `setEncoderCountsPerRev()`.

COMMENTS

This function returns the value set with the `setEncoderCountsPerRev()` function. This function does not query the high speed interface or the sensor.

setKeycodeFile

Specifies the file containing the license keycode(s) corresponding to the sensor(s) being used. This function applies to non-Windows versions of the CTI-HSIF-PCI library only.

```
C:     long mySensor.setKeycodeFile(const long sensorHandle,
                                    const char* filename)

C++:   long mySensor. setKeycodeFile (const char *filename)

VB:    This function is not available in Visual Basic.
```

PARAMETERS

`filename` – the fully-qualified filename of the keycode file. This file must contain the license keycode supplied with the CTI-HSIF-PCI library. If more than one sensor is being used, each keycode should be on a separate line in the file.

RETURN VALUES

CTI_SUCCESS if the specified file can be successfully opened and read.

CTI_CANNOT_OPEN if the file cannot be opened. This may occur if the specified filename is incorrect (a fully qualified filename must be given), or if the calling process does not have permission to open the file. `getErrorMessage()` may be used to get an extended error message.

CTI_BAD_PARAM if an invalid filename is specified.

CTI_LICENSE_ERR if the file is empty.

COMMENTS

This function applies to non-Windows versions of the library only. Windows versions of the CTI-HSIF-PCI library store license keycode information in the Windows registry. This function may be called on Windows versions of the library, but will have no effect.

getKeycodeFile

Get the name of the license keycode file, as set by `setKeycodeFile()`.

```
C:     long mySensor.getKeycodeFile(const long sensorHandle,
                                    char * pFileName, const long buflen)

C++:   long mySensor. getKeycodeFile (char * pFileName, const long buflen)

VB:    This function is not available in Visual Basic.
```

PARAMETERS

`pFilename` – Pointer to a buffer to hold the returned filename. In order to return the entire filename, the buffer should be at least as long as the filename set with the `setKeycodeFile()` function.

`buflen` – The length of the buffer pointed to, in characters.

RETURN VALUES

CTI_SUCCESS if successful.

CTI_BAD_PARAM if `pFilename` is null, or `buflen` is negative.

CTI_BUFFER_TOO_SMALL if the buffer is too small to hold the entire length of the filename. In this case, as much of the filename as can fit in the actual buffer is returned.

COMMENTS

This function applies to non-Windows versions of the library only. Windows versions of the CTI-HSIF-PCI library store license keycode information in the Windows registry.

setCommOpen

Opens the serial communications port to the sensor, and resets the sensor to factory defaults.

```
C:     long setCommOpen(const long sensorHandle,
                        const char * serialPortName,
                        const long initialBaudRate)

C++:   long mySensor.setCommOpen(const char * serialPortName,
                        const long initialBaudRate)

VB:    mySensor.setCommOpen(ByVal serialPortName As String,
                            ByVal initialBaudRate As Long) As Long
```

PARAMETERS

`serialPortName` – The name of the serial port used to communicate with the sensor. (e.g. "COM1")

`initialBaudRate` – The baud rate at which to open the port. This must be one of the valid baud rates supported by the sensor hardware. See the sensor hardware documentation for a list of valid baud rates.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_BOARDPARAMS_NOT_SET if the `setBoardParams()` function has not yet been called.
CTI_CALIBFILE_NOT_SET if the `setCalibrationFile()` function has not yet been called.
CTI_ILLEGAL_CALL if the serial port has already been opened by the library.
CTI_BAD_PARAM if the serial port name is empty, or invalid, or if the baud rate is invalid for this sensor.
CTI_COMM_OPEN_ERROR if the specified serial port could not be opened successfully.
CTI_HW_INIT_FAILURE or CTI_BUFFERSIZE_UNKNOWN if communications with the high-speed interface card cannot be established, or if the size of the high-speed interface's on-board hardware buffer cannot be determined. On Linux systems this may be because the kernel driver is not loaded.

COMMENTS

setCommOpen function *must* be called before any function that communicates with the sensor.

This function resets the library's internal error indicator flag and error message text, starts up the background threads used to process incoming data from the sensor and high speed interface, opens the serial port to the sensor, resets the sensor to factory defaults (changing the computer's serial port baud rate to 9600 baud, if necessary), and resets the high speed interface board.

If this function returns successfully, the computer's serial port will be set to 9600 baud.

WARNING: Since the library has no means of determining the current baud rate setting of the sensor, the `initialBaudRate` parameter specified *must* match the actual baud rate of the sensor, otherwise communications with the sensor will fail.

If the function fails, `getErrorMessage()` may be called to get an extended error message.

The user should be aware of the following considerations before calling `setCommOpen()`:

- The `setBoardParams()` function *must* be called prior to calling `setCommOpen()`.
- The `setCalibrationFile()` function must be called to specify the calibration file to be used.
- If the encoder is used, `setEncoderCountsPerRev()` must also be called before `setCommOpen()`.
- For non-Windows library versions, `setKeycodeFile()` must be called before `setCommOpen()`.

getIsCommOpen

Determine if the communications link to the sensor is open.

```
C:     long getIsCommOpen(const long sensorHandle)

C++:   long mySensor.getIsCommOpen ()

VB:    Long mySensor.getIsCommOpen ()
```

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the communications link to the sensor and high-speed interface card is currently open, and 0 (zero) otherwise.

COMMENTS

None.

setCommClosed

Turns laser off and closes the serial port to the sensor, resetting baud rate to 9600.

```
C:     long setCommClosed(const long sensorHandle)

C++:   long mySensor.setCommClosed()

VB:    Long mySensor.setCommClosed()
```

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_WAIT_TIMEOUT if the function timed out while waiting for an active callback to complete.
If any error occurs, `getErrorMessage()` may be called to get an extended error message.

COMMENTS

If this function is called while a callback is active, it will wait a maximum of 20 seconds for the callback to complete. If the application program does not return from the callback function within 20 seconds, then `setCommClosed()` will return CTI_WAIT_TIMEOUT, and no other action will be taken.

Otherwise, this function resets the callback threshold to zero (disabling callbacks), resets the sensor to its factory default settings and turns off the laser. It then closes the computer's serial communications port, shuts down the background data acquisition thread, resets the high speed interface, resets the library's error indicator, and frees any internal resources used by the library.

If the function fails, `getErrorMessage()` may be called to get an extended error message.

This function must be called by an application program to cleanly terminate use of the CTI-HSIF-PCI library. Failure to call this function may result in undesirable behaviour, including memory leaks, and difficulty re-establishing communications with the sensor during subsequent runs of the application program.

`setCommClosed()`should not be called from within a callback.

setBaudRate

Changes the baud rate used for serial communications to the sensor.

```
C:     long setBaudRate(const long sensorHandle, const long baudRate)

C++:   long mySensor.setBaudRate(const long baudRate)

VB:    mySensor.setBaudRate(ByVal baudRate As Long) As Long
```

PARAMETERS

`baudRate` – The new baud rate to use in communicating with the sensor.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_BAD_PARAM if the specified baud rate is invalid for the sensor.
CTI_FAILURE if any other error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This function will change the baud rate of both the sensor and the host computer's serial port to the new rate specified, which must be one of the valid rates for the sensor. Please consult the sensor hardware documentation for a list of valid baud rates.

Calling `setBaudRate()` will discard any data samples currently in the library's internal buffer.

When using the AR4000 sensor with the high speed interface, all data from the sensor is returned via the high speed interface card, not the serial interface. Increasing the serial port baud rate will not improve the sampling speed, so it is strongly recommended that the serial port baud rate be left at the factory default of 9600 baud.

This is equivalent to the sensor's 'B' command.

getBaudRate

Returns the serial communications link's current baud rate.

```
C:     long getBaudRate(const long sensorHandle)

C++:   long mySensor.getBaudRate()

VB:    mySensor.getBaudRate() As Long
```

PARAMETERS

None.

RETURN VALUES

The currently set serial port baud rate.

COMMENTS

This function returns the currently set baud rate for the computer's serial port. Note that this may differ from the sensor's current baud rate setting, if the sensor's baud rate has been changed by some means other than using the library functions.

setBufferSize

Sets the library's internal data buffer size, in samples.

```
C:     long setBufferSize(const long sensorHandle, const long nSamples)

C++:   long mySensor.setBufferSize(const long nSamples)

VB:    mySensor.setBufferSize(ByVal nSamples As Long) As Long
```

PARAMETERS

`nSamples` – The size of the library's internal data buffer, in samples. Any size smaller than the library's default of 5000 samples will result in the 5000 sample default size being used.

RETURN VALUES

The actual buffer size set (in number of samples) if successful.
CTI_BAD_PARAM if the specified size is zero or less.
CTI_FAILURE if any error other occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This function sets a new size for the internal buffer within the library that is used to hold data coming from the high speed interface before that data is read by the application program. Calling this function will discard any data currently in the library's buffer.

If the library cannot allocate enough memory to satisfy the requested buffer size, it will try successively smaller sizes, and will return the size, in samples, of the buffer actually allocated.

If the resultant buffer size is less than the callback threshold set with the `setCallbackThreshold()` function, the callback threshold will be set to zero (i.e. disabling callbacks).

getBufferSize

Returns the library's internal data buffer size, in samples.

```
C:     long getBufferSize(const long sensorHandle)

C++:   long mySensor.setBufferSize()

VB:    mySensor.setBufferSize() As Long
```

PARAMETERS

`None`

RETURN VALUES

The size of the library's internal data buffer, in samples.

COMMENTS

None.

setClearBuffer

Clears (empties) the library's internal data buffer, and the high speed interface's hardware buffer memory.

```
C:     long setClearBuffer(const long sensorHandle)

C++:   long mySensor.setClearBuffer()

VB:    mySensor.setClearBuffer() As Long
```

PARAMETERS

None.

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

This function discards any samples in the library's internal buffer, leaving the buffer empty. However, if continuous sampling is on (the default), the library's background thread will add samples to the buffer as new data arrives from the high speed interface. Use `setContinuousHSIFOff()` to turn off continuous sampling.

This function also clears the overflow flag for the library's internal buffer.

getDidBufferOverflow

Determine if the library's internal data buffer has overflowed.

```
C:     long getDidBufferOverflow(const long sensorHandle)

C++:   long mySensor.getDidBufferOverflow()

VB:    mySensor.getDidBufferOverflow() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the library's internal data buffer has overflowed, and 0 (zero) otherwise.

COMMENTS

If the application program does not remove samples from the library's internal buffer fast enough, then it will eventually overflow. This function determines if that has happened.

The overflow flag may be reset using the `setResetBufferOverflow()` function. The `setFactoryDefaults()` and `setClearBuffer()` functions also reset the overflow flag.

setResetBufferOverflow

Reset the library's internal data buffer overflow flag.

```
C:     long setResetBufferOverflow(const long sensorHandle)

C++:   long mySensor.setResetBufferOverflow()

VB:    mySensor.setResetBufferOverflow() As Long
```

PARAMETERS

None.

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

This function resets the library's internal data buffer overflow flag. If continuous sampling is on (see the `setContinuousHSIFOn()` function on page 40), the library's background data acquisition thread will attempt to add new samples to the library's internal buffer as raw data is available from the high speed interface. Therefore, unless samples are removed from the library's buffer, the overflow flag will be immediately reset once new samples arrive.

The `setFactoryDefaults()` and `setClearBuffer()` functions also reset the overflow flag.

Do not confuse this function with the `setResetHSIFBoard()` function, which sends a reset signal to the high speed interface board.

setCallbackFunction

Sets the function to be called when the number of samples in the library's internal data buffer reaches the callback threshold.

```
C:     long setHSIFCallbackFunction(const long sensorHandle,
                    long cbFn(const HSIF_DATA_PT * pD1,
                              const long N1,
                              const HSIF_DATA_PT * pD2,
                              const long N2))

C++:   long mySensor.setHSIFCallbackFunction(
                    long cbFn(const HSIF_DATA_PT * pD1,
                              const long N1,
                              const HSIF_DATA_PT * pD2,
                              const long N2))

VB:    This function is not available from Visual Basic.
```

PARAMETERS

`cbFn` – the function to be called when the number of samples in the library's internal data buffer reaches the threshold set by the `setCallbackThreshold()` function. The callback function must have the prototype:

```
long fn(const HSIF_DATA_PT * pD1, const long N1,
        const HSIF_DATA_PT * pD2, const long N2)
```

RETURN VALUES

CTI_BAD_PARAM if the function pointer is null. CTI_SUCCESS otherwise.

COMMENTS

When the number of samples in the library's internal buffer exceeds the threshold set by the `setCallbackThreshold()` function, the library will create a *new thread of execution* and call the callback function from that new thread. The library guarantees that the callback function will not be called a second time while it is already active. Thus, the application programmer does not need to be concerned about re-entrancy issues within the callback function itself.

When the callback is called the pD1 pointer points to a contiguous buffer holding N1 range samples. The pD2 parameter points to a second contiguous buffer holding N2 range samples. These pointers may be used to retrieve range samples from the library without making any additional calls to the library. Each sample is a `HSIF_DATA_PT` structure (see structure definition on page 43 of the Reference Manual)

See the Programmers Guide, and the sample programs, for an example of the use of the library's callback functionality.

The application programmer must be careful to take appropriate precautions when accessing data that is shared between the callback function and other parts of the program. Mutexes, critical sections or other synchronization mechanisms *must* be used to ensure that any shared data is accessed by only one thread of execution at a time. The CTI-HSIF-PCI library itself is fully thread-safe, and any library function may be called from any thread of execution without user-supplied synchronization mechanisms.

The callback functionality is not available from Visual Basic, since the "apartment model" threading used by Visual Basic is incompatible with the free-threading model used by C, C++ and the CTI library.

setCallbackThreshold

Sets the threshold, in samples, at which to call the callback function.

```
C:     long setCallbackThreshold(const long sensorHandle, const long nSamples)

C++:   long mySensor.setCallbackFunction(const long nSamples)

VB:    This function is not available from Visual Basic.
```

PARAMETERS

`nSamples` – the number of samples which must be present in the library's internal data buffer for the callback function specified by `setCallbackFunction()` to be called. `nSamples` must be less than or equal to the library's internal data buffer size.

Calling `setCallbackThreshold()` with a value of zero disables calling the callback function.

RETURN VALUES

CTI_BAD_PARAM if `nSamples` is negative. Otherwise, returns the actual threshold set, in samples.

COMMENTS

The `setCommClosed()` function resets the callback threshold to zero, so `setCallbackThreshold()` should be called to re-establish the callback threshold if the serial port is closed and subsequently reopened.

If the `setBufferSize()` function is called, and the resultant buffer size is less than the current callback threshold, the threshold will be set to zero (i.e. disabling callbacks).

getCallbackThreshold

Gets the threshold, in samples, at which the callback function.will be called.

```
C:     long getCallbackThreshold(const long sensorHandle)

C++:   long mySensor.getCallbackFunction()

VB:    This function is not available from Visual Basic.
```

PARAMETERS

None.

RETURN VALUES

The threshold, in samples, at which the callback function set by `setCallbackFunction()` will be called.

COMMENTS

None.

getIsBufferAtThreshold

Determine if the library's internal buffer has the number of samples specified by the callback threshold.

```
C:     long getIsBufferAtThreshold(const long sensorHandle)

C++:   long mySensor.getIsBufferAtThreshold()

VB:    This function is not available from Visual Basic.
```

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the library's internal buffer has at least as many samples as specified by the callback threshold value set with the `setCallbackThreshold()` function. Returns 0 (zero) otherwise.

COMMENTS

This function returns zero or one based on the number of samples available in the library's internal buffer at the particular instant in time that the function was called. If continuous sampling is on (see the `setContinuousHSIFOn()` function on page 40), the library's background data acquisition thread will continue to add samples to the library's internal buffer. Therefore, two consecutive calls to the `getIsBufferAtThreshold()` function may return different results.

setMotorMaxRPM

Tell the library what the speed of the motor is when the motor power setting is at its maximum.

```
C:     long mySensor.setMotorMaxRPM(const long sensorHandle,
                                    const long mNum,
                                    const long maxRPM)

C++:   long mySensor.setMotorMaxRPM(const long mNum, const long maxRPM)

VB:    mySensor.setMotorMaxRPM(ByVal mNum As Long, ByVal maxRPM As Long) As Long
```

PARAMETERS

`mNum` – the motor number (1 or 2) to set the maximum RPM for.

`maxRPM` – the speed of the specified motor, in revolutions per minute, when the motor power is set to its maximum value of 255. `maxRPM` must be greater than zero.

RETURN VALUES

CTI_BAD_PARAM if `mNum` is not 1 or 2, or if `maxRPM` is less than or equal to zero. CTI_SUCCESS otherwise.

COMMENTS

When using the high speed interface card, the speed of the motor is controlled by specifying a power setting from 0 to 255, using the `setMotorPower()` function (see page 27). For convenience, the function `setMotorRPM()` (see page 29) is also provided, which allows the approximate speed of the motor to be specified in revolutions per minute.

The actual speed of the motor at the maximum power setting depends on the particular motor model ordered with the high speed interface hardware, therefore before the `setMotorRPM()` function can be used, `setMotorMaxRPM()` must be called so that the library can calculate the motor power setting corresponding to the desired revolutions per minute.

This function does not access the high speed interface card.

Do not confuse this function with `setMotorRPM()` which actually turns on the motor by setting the high speed interface's motor power level.

getMotorMaxRPM

Return the speed of the motor at the maximum motor power setting.

```
C:     long mySensor.getMotorMaxRPM(const long sensorHandle,
                                    const long mNum)

C++:   long mySensor.getMotorMaxRPM(const long mNum)

VB:    mySensor.getMotorMaxRPM(ByVal mNum As Long) As Long
```

PARAMETERS

`mNum` – the motor number (1 or 2) for which to get the maximum RPM.

RETURN VALUES

CTI_BAD_PARAM if `mNum` is not 1 or 2. Otherwise returns the revolutions per minute of the specified motor when the motor power setting is at its maximum, as set by the `setMotorMaxRPM()` function.

COMMENTS

This function does not access the high speed interface card.

Do not confuse this function with `getMotorRPM()` which returns the currently set motor speed.

Sensor and High Speed Interface Configuration Functions

setFactoryDefaults

Resets all sensor settings to their factory default values, and resets the high speed interface.

```
C:     long setFactoryDefaults(const long sensorHandle)

C++:   long mySensor.setFactoryDefaults()

VB:    mySensor.setFactoryDefaults() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_BOARDPARAMS_NOT_SET if the `setBoardParams()` function has not yet been called.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_FAILURE if any error occurs. Call `getErrorMessage()` to get an extended error message.

COMMENTS

This function resets all sensor configuration parameters to their factory default settings, as documented in the sensor hardware documentation. In particular, both the sensor's and the computer's serial communications baud rate will be changed to 9600 baud after this function call.

This function also resets the high speed interface board, which empties the board's onboard buffer memory, resets the motor power controls to zero, and resets the encoder counters.

Calling this function will discard any data samples currently in the library's internal buffer, and reset the overflow flag for the library's internal buffer.

This function is equivalent to the sensor's 'I' command, followed by a reset to the high speed interface.

setResetHSIFBoard

Reset the high speed interface board.

```
C:     long mySensor.setResetHSIFBoard(const long sensorHandle)

C++:   long mySensor.setResetHSIFBoard()

VB:    mySensor.setResetHSIFBoard() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_BOARDPARAMS_NOT_SET if the `setBoardParams()` function has not yet been called.
CTI_SUCCESS if successful.

COMMENTS

This function sends a reset signal to the high speed interface board. This clears the card's onboard buffer memory, resets the hardware's buffer overflow flag, and resets the motor power and encoder counters. Please see the sensor hardware documentation for more details.

setAnalogOutputCalibrated

Selects the analog current loop to be based on calibrated distance measurements.

```
C:     long setAnalogOutputCalibrated(const long sensorHandle)

C++:   long mySensor.setAnalogOutputCalibrated()

VB:    mySensor.setAnalogOutputCalibrated() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if the function succeeds.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_FAILURE if any error occurs. Call `getErrorMessage()` to get an extended error message.

COMMENTS

This function is equivalent to the sensor's 'X1' command. Please see the sensor hardware documentation for information on the effect of this command on the sensor's zero point and analog span settings.

setAnalogOutputUnCalibrated

Selects the analog current loop to be based on the direct, uncalibrated sensor output.

```
C:     long setAnalogOutputUnCalibrated(const long sensorHandle)

C++:   long mySensor.setAnalogOutputUnCalibrated()

VB:    mySensor.setAnalogOutputUnCalibrated() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if the function succeeds.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_FAILURE if any error occurs. `getErrorMessage()` may be used to get an extended error message.

COMMENTS

This function is equivalent to the sensor's 'X2' command. Please see the sensor hardware documentation for information on the effect of this command on the sensor's zero point and analog span settings.

getIsAnalogOutputCalibrated

Returns 1 (one) if the analog output mode is calibrated output, and 0 (zero) otherwise.

```
C:     long getIsAnalogOutputCalibrated(const long sensorHandle)

C++:   long mySensor.getIsAnalogOutputCalibrated()

VB:    mySensor.getIsAnalogOutputCalibrated() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the analog output mode is calibrated output, and 0 (zero) otherwise.

COMMENTS

This function will return 1 (one) if the current analog output mode is calibrated output, even if the analog output is not currently on. The `getIsAnalogOutputOn()` function should be used to determine if the analog output is currently on.

setAnalogOutputOff

Turns off the current loop (analog) output.

```
C:     long setAnalogOutputOff(const long sensorHandle)

C++:   long mySensor.setAnalogOutputOff()

VB:    mySensor.setAnalogOutputOff() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if the function succeeds.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_FAILURE if any error occurs. `getErrorMessage()` may be used to get an extended error message.

COMMENTS

This function is equivalent to the sensor's 'X3' command.

getIsAnalogOutputOn

Determine if the analog (current loop) output is on.

```
C:     long getIsAnalogOutputOn(const long sensorHandle)

C++:   long mySensor.getIsAnalogOutputOn()

VB:    mySensor.getIsAnalogOutputOn() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the analog output is on, and 0 (zero) otherwise.

COMMENTS

None.

setAnalogZeroCurrent

Sets the output current delivered over the current loop interface at the sensor zero point setting.

```
C:     long setAnalogZeroCurrent(const long sensorHandle, const long microAmps)

C++:   long mySensor.setAnalogZeroCurrent(const long microAmps)

VB:    mySensor.setAnalogZeroCurrent(ByVal microAmps As Long) As Long
```

PARAMETERS

`microAmps` – the desired analog zero current in micro-amps. Must be greater than or equal to zero and less than or equal to 20,000 (i.e. 20mA).

RETURN VALUES

CTI_BAD_PARAM if the parameter is out of range.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if the function succeeds.
CTI_FAILURE if any error occurs. `getErrorMessage()` may be used to get an extended error message.

COMMENTS

This is equivalent to the sensor's 'J' command.

getAnalogZeroCurrent

Get the analog zero current setting.

```
C:     long getAnalogZeroCurrent(const long sensorHandle)

C++:   long mySensor.getAnalogZeroCurrent()

VB:    mySensor.getAnalogZeroCurrent() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns the analog zero current setting in micro-amps, as set by the `setAnalogZeroCurrent()` function.

COMMENTS

None.

getHSIFBufSizeBytes

Returns the size, in bytes, of the high speed interface card's onboard buffer memory.

```
C:     long mySensor.getHSIFBufSizeBytes(const long sensorHandle)

C++:   long mySensor.getHSIFBufSizeBytes()

VB:    mySensor.getHSIFBufSizeBytes() As Long
```

PARAMETERS

None.

RETURN VALUES

The size, in bytes, of the high speed interface card's onboard buffer memory if successful.

CTI_BOARDPARAMS_NOT_SET if the `setBoardParams()` function has not yet been called.

CTI_FAILURE if not successful at determining the high speed interface card's buffer size.

COMMENTS

None.

getHSIFBufSizeSamples

Returns the size, in number of samples, of the high speed interface card's onboard buffer memory.

```
C:     long mySensor.getHSIFBufSizeSamples(const long sensorHandle)

C++:   long mySensor.getHSIFBufSizeSamples()

VB:    mySensor.getHSIFBufSizeSamples() As Long
```

PARAMETERS

None.

RETURN VALUES

The size, in samples, of the high speed interface card's onboard buffer memory if successful.

CTI_BOARDPARAMS_NOT_SET if the `setBoardParams()` function has not yet been called.

CTI_FAILURE if not successful at determining the high speed interface card's buffer size.

COMMENTS

The value returned by this function is 1/16 that returned by the `getHSIFBufSizeBytes()` function, since each sample from the high speed interface consists of 16 bytes (see the sensor hardware documentation).

setMotorPower

Set the motor power level.

```
C:     long mySensor.setMotorPower(const long sensorHandle,
                                   const long mNum,
                                   const long power,
                                   const long dir)

C++:   long mySensor.setMotorPower(const long mNum,
                                   const long power,
                                   const long dir)

VB:    mySensor.setMotorPower(ByVal mNum As Long,
                              ByVal power As Long,
                              ByVal dir As Long) As Long
```

PARAMETERS

`mNum` – the motor number (1 or 2) for which to set the power.

`power` – the power to be applied to the specified motor. Valid range is 0 <= `power` <= 255.

`dir` – the motor direction (0 or 1). This parameter exists for backward compatibility, and has no effect when using the PCI high-speed interface card.

RETURN VALUES

CTI_BAD_PARAM if any of the parameters are out of range.

CTI_BOARDPARAMS_NOT_SET if the `setBoardParams()` function has not yet been called.

CTI_SUCCESS otherwise.

COMMENTS

Please read the comments for the `setMotorRPM()` function on page 29.

getMotorPower

Return the current motor power setting.

```
C:     long mySensor.getMotorPower(const long sensorHandle,
                                   const long mNum) const

C++:   long mySensor.getMotorPower(const long mNum)

VB:    mySensor.getMotorPower(ByVal mNum As Long) As Long
```

PARAMETERS

`mNum` – the motor number (1 or 2) for which to get the power setting.

RETURN VALUES

CTI_BAD_PARAM if `mNum` is not 1 or 2. Otherwise returns the current motor power setting, as set by the `setMotorPower()` function.

COMMENTS

None.

getMotorDirection

Return the current motor direction setting. This function exists for backward compatibility, as the motor direction setting has no effect when using the PCI high-speed interface card.

```
C:     long mySensor.getMotorDirection(const long sensorHandle,
                                       const long mNum) const

C++:   long mySensor.getMotorDirection(const long mNum)

VB:    mySensor.getMotorDirection(ByVal mNum As Long) As Long
```

PARAMETERS

`mNum` – the motor number (1 or 2) for which to get the direction setting.

RETURN VALUES

CTI_BAD_PARAM if `mNum` is not 1 or 2. Otherwise returns the current motor direction setting, as set by the `setMotorPower()` function. This function exists for backward compatibility, as the motor direction setting has no effect when using the PCI high-speed interface card.

COMMENTS

None.

setMotorRPM

Set the motor speed.

```
C:     long mySensor.setMotorRPM(const long sensorHandle,
                                 const long mNum,
                                 const long rpm,
                                 const long dir)

C++:   long mySensor.setMotorRPM(const long mNum,
                                 const long rpm,
                                 const long dir)

VB:    mySensor.setMotorRPM(ByVal mNum As Long,
                            ByVal rpm As Long,
                            ByVal dir As Long) As Long
```

PARAMETERS

`mNum` – the motor number (1 or 2) for which to set the speed.

`rpm` – the speed to set for the specified motor. rpm must be greater than or equal to zero, and less than or equal to the value set with the `setMotorMaxRPM()` function (see page 18).

`dir` – the motor direction (0 or 1). This parameter exists for backward compatibility, and has no effect when using the PCI high-speed interface card.

RETURN VALUES

CTI_BAD_PARAM if any of the parameters are out of range.

CTI_BOARDPARAMS_NOT_SET if the `setBoardParams()` function has not yet been called, or if the `setMotorMaxRPM()` function has not yet been called.

CTI_SUCCESS otherwise.

COMMENTS

Please read the sensor hardware documentation carefully for complete information on the high speed interface's motor and encoder features.

When using the high speed interface card, the speed of the motor is controlled by specifying a power setting from 0 to 255, using the `setMotorPower()` function (see page 27). For convenience, the `setMotorRPM()` function is also provided, which allows the approximate speed of the motor to be specified in revolutions per minute.

However, since the actual speed of the motor at the maximum power setting depends on the particular motor model ordered with the high speed interface hardware, before the `setMotorRPM()` function can be used, `setMotorMaxRPM()` (see page 19) must be called so that the library can calculate the motor power setting corresponding to the desired revolutions per minute.

Generally, a minimum value of at least 25% of full power (or 25% of the full speed RPMs) must be set in order to overcome the resting friction of the motor, and start the motor turning. The actual speed of the motor will depend on many factors, such as the frictional effects of the motor bearings, the linearity of the motor speed with respect to applied voltage, etc., therefore the speed set by this function should be considered an approximation only.

Do not confuse this function with `setMotorMaxRPM()`.

getMotorRPM

Returns the currently set motor speed, in revolutions per minute.

```
C:     long getMotorRPM(const long sensorHandle, const long mNum)

C++:   long mySensor.getMotorRPM(const long mNum)

VB:    mySensor.getMotorRPM(ByVal mNum As Long) As Long
```

PARAMETERS

`mNum` – the motor number (1 or 2) for which to get the speed setting.

RETURN VALUES

CTI_BAD_PARAM if `mNum` is not 1 or 2. Otherwise returns the current motor speed, in revolutions per minute, as set by the `setMotorRPM()` function.

COMMENTS

Note that the actual motor speed may differ from the value returned by this function, since the actual speed of the standard motors supplied with the high speed interface can only be approximately determined by the power setting applied to the motor.

setSensorMaxRange

Sets the maximum range the sensor is expected to measure, in inches.

```
C:     long setSensorMaxRange(const long sensorHandle, const long inches)

C++:   long mySensor.setSensorMaxRange(const long inches)

VB:    mySensor.setSensorMaxRange(ByVal inches As Long) As Long
```

PARAMETERS

`inches` – the maximum range the sensor is expected to measure, in inches. Valid range is 0 <= `distance` <= 99,999.

RETURN VALUES

CTI_BAD_PARAM if the parameter is out of range.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if the function succeeds.
CTI_FAILURE if any error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This function changes the sensor's hardware parameters to facilitate better resolution and higher sample rates, as explained in the sensor hardware documentation. (Do not confuse this with the `setMaxValidRange()` function, which sets a filter threshold for acquired samples.)

Please see the sensor hardware documentation for a complete discussion of the factors affecting sample rate and resolution.

This is equivalent to the sensor's 'F' command.

getSensorMaxRange

Returns the currently configured maximum sensor range setting.

```
C:     long getSensorMaxRange(const long sensorHandle)

C++:   long mySensor.getSensorMaxRange()

VB:    mySensor.getSensorMaxRange() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns the currently configured maximum sensor range setting, in inches, as set with the `setSensorMaxRange()` function.

COMMENTS

Do not confuse the value returned by this function with that returned by `getMaxValidRange()`.

setSpan

Sets the distance at which the current loop output is at the maximum value.

```
C:     long setSpan(const long sensorHandle, const long distance)

C++:   long mySensor.setSpan(const long distance)

VB:    mySensor.setSpan(ByVal distance As Long) As Long
```

PARAMETERS

`distance` – the distance to set as the span point. Units are in 1/100ths of an inch, if the current sensor output mode is English units, or in millimeters, if the current sensor output mode is metric units. Valid range is 0 <= `distance <=` 9,999,999.

RETURN VALUES

CTI_BAD_PARAM if the distance parameter is out of range.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if the function succeeds.
CTI_FAILURE if any error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This function is equivalent to the sensor's 'U' command. Please refer to the sensor hardware documentation for further information on the effect of the sensor's `setSpan()` command.

getSpan

Returns the span point set with the `setSpan` function.

```
C:     long getSpan(const long sensorHandle)

C++:   long mySensor.getSpan()

VB:    mySensor.getSpan() As Long
```

PARAMETERS

None.

RETURN VALUES

The value set with the `setSpan()` function.

COMMENTS

None.

setTempHoldLevel

Sets the sensor's internal temperature hold level, in degrees Fahrenheit.

```
C:     long setTempHoldLevel(const long sensorHandle, const long temp)

C++:   long mySensor.setTempHoldLevel(const long temp)

VB:    mySensor.setTempHoldLevel(ByVal temp As Long) As Long
```

PARAMETERS

`temp` – the temperature hold level in degrees Fahrenheit. Valid range is 0 <= `temp` <= 99.

RETURN VALUES

CTI_BAD_PARAM if the parameter is out of range.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if the function succeeds.
CTI_FAILURE if any error occurs. Call `getErrorMessage()` to get an extended error message.

COMMENTS

In general, the sensor temperature hold level should not be changed, since the sensor is calibrated at 95°F, and changing the temperature hold level from this value may result in inaccurate readings. Note that the sensor can only heat, not cool, so the actual sensor temperature may differ from the set value if high ambient temperatures are present.

This is equivalent to the sensor's 'C' command.

getTempHoldLevel

Returns the currently configured sensor temperature hold level.

```
C:     long getTempHoldLevel(const long sensorHandle)

C++:   long mySensor.getTempHoldLevel()

VB:    mySensor.getTempHoldLevel() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns the currently configured maximum sensor temperature hold level, in degrees Fahrenheit.

COMMENTS

Since the sensor can only heat, not cool, the *actual* sensor temperature may differ from this set value if high ambient temperatures are present. The sensor's actual temperature reading will be returned with the range data.

setZeroPt

Sets the zero point for calibrated output values.

```
C:     long setZeroPt(const long sensorHandle, const long distance)

C++:   long mySensor.setZeroPt(const long distance)

VB:    mySensor.setZeroPt(ByVal distance As Long) As Long
```

PARAMETERS

`distance` – the distance to set as the sensor zero point. Units are in 1/100ths of an inch, if the current sensor mode is English output, or in millimeters, if the current sensor output mode is metric units. This function will reset the uncalibrated zero point to zero. Valid range is 0 <= `distance` <= 99,999.

RETURN VALUES

CTI_BAD_PARAM if the distance parameter is out of range.
CTI_SUCCESS if the function succeeds.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_FAILURE if any error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This function is equivalent to the sensor's 'Z' command. Please refer to the sensor hardware documentation for further information on the effect of this command.

getZeroPt

Returns the zero point for calibrated output values.

```
C:     long getZeroPt(const long sensorHandle)

C++:   long mySensor.getZeroPt()

VB:    mySensor.getZeroPt() As Long
```

PARAMETERS

None.

RETURN VALUES

The zero point value set with the setZeroPt function. Units are in 1/100ths of an inch, if the sensor mode is English output, or in millimeters, if the sensor output mode is metric units.

COMMENTS

None.

setZeroPtUncalibrated

Sets the zero point for serial and current loop outputs.

```
C:     long setZeroPtUncalibrated(const long sensorHandle, const long distance)

C++:   long mySensor.setZeroPtUncalibrated(const long distance)

VB:    mySensor.setZeroPtUncalibrated(ByVal distance As Long) As Long
```

PARAMETERS

`distance` – the distance to set as the sensor's zero point for uncalibrated output data. The distance will be interpreted as being in the currently set sensor output units. Valid range is 0 <= `distance` <= 9,999,999.

RETURN VALUES

CTI_BAD_PARAM if the distance parameter is out of range.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if the function succeeds.
CTI_FAILURE if any error occurs. Call `getErrorMessage()` to get an extended error message.

COMFMENTS

This function is equivalent to the sensor's 'Y' command. This function will reset the calibrated zero point to zero.

getZeroPtUncalibrated

Returns the uncalibrated zero point set with the `setZeroPtUncalibrated()` function.

```
C:     long getZeroPtUncalibrated(const long sensorHandle)

C++:   long mySensor.getZeroPtUncalibrated()

VB:    mySensor.getZeroPtUncalibrated() As Long
```

PARAMETERS

None.

RETURN VALUES

The zero point value set with the `setZeroPtUncalibrated()` function.

COMMENTS

None.

Data Acquisition Functions

setLaserOn

Turns on the laser.

```
C:     long setLaserOn(const long sensorHandle)

C++:   long mySensor.setLaserOn()

VB:    mySensor.setLaserOn() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if the function succeeds.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
If any other error occurs, call `getErrorMessage() to` get an extended error message.

COMMENTS

This is equivalent to the sensor's 'H' command.

setLaserOff

Turns off the laser.

```
C:     long setLaserOff(const long sensorHandle)

C++:   long mySensor.setLaserOff()

VB:    mySensor.setLaserOff() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if the function succeeds.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
If any other error occurs, call `getErrorMessage() to` get an extended error message.

COMMENTS

This is equivalent to the sensor's 'L' command.

getIsLaserOn

Determine if the laser is currently on.

```
C:     long getIsLaserOn(const long sensorHandle)

C++:   long mySensor.getIsLaserOn()

VB:    mySensor.getIsLaserOn() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called, otherwise returns 1 (one) if the laser is currently on, and 0 (zero) otherwise.

COMMENTS

None.

setSampleInterval

Set the sensor sampling interval, in microseconds.

```
C:     long setSampleInterval(const long sensorHandle, const long interval)

C++:   long mySensor.setSampleInterval(const long interval)

VB:    mySensor.setSampleInterval(ByVal interval As Long) As Long
```

PARAMETERS

`interval` – the sampling interval, in microseconds. Valid range is 20 <= `interval` <= 9,999,999.

RETURN VALUES

The sampling interval, in microseconds, if the function succeeds.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_BAD_PARAM if the parameter is out of range.
CTI_FAILURE if any error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This is equivalent to the sensor's 'S' command.

Attempting to set a sample interval longer than approximately 2200 microseconds will not change the actual sample rate unless the sensor's maximum expected range is also changed, using the `setSensorMaxRange()` function. Please see the sensor hardware documentation for details.

This function will discard any samples currently in the library's internal buffer.

getSampleInterval

Returns the sensor's currently set sampling interval, in microseconds.

```
C:     long getSampleInterval(const long sensorHandle)

C++:   long mySensor.getSampleInterval()

VB:    mySensor.getSampleInterval() As Long
```

PARAMETERS

None.

RETURN VALUES

The sensor sampling interval, in microseconds.

COMMENTS

None.

setSamplesPerSec

Set the sensor sampling rate.

```
C:     long setSamplesPerSec(const long sensorHandle, const long samplesPerSec)

C++:   long mySensor.setSamplesPerSec(const long samplesPerSec)

VB:    mySensor.setSamplesPerSec(ByVal samplesPerSec As Long) As Long
```

PARAMETERS

`samplesPerSec` – the sampling rate to set, in samples per second. Valid range is 32 <= `samplesPerSec` <= 50,000.

Note that the actual rate set may differ from the value provided. See the comments section below.

RETURN VALUES

The actual sampling rate set, in samples per second, if the function succeeds.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_BAD_PARAM if the parameter is out of range.
CTI_FAILURE if any error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

The sensor's sampling interval may be set in discrete intervals of one microsecond. Due to the discrete sampling intervals possible, the actual sample rate set may differ from the supplied parameter value. For example, specifying a sample rate of 101 samples per second will result in a sample interval of 9900 microseconds, and an actual sample rate of approximately 101.01 samples/second.

The lowest possible sample rate when using the high speed interface is approximately 32 samples/second.

Attempting to set a sample rate slower than approximately 450 samples/second will not change the actual sample rate unless the sensor's maximum expected range is also changed, using the `setSensorMaxRange()` function. Please see the sensor hardware documentation for details.

This function will discard any samples currently in the library's internal buffer.

getSamplesPerSec

Get the sensor sampling rate.

```
C:     long getSamplesPerSec(const long sensorHandle)

C++:   long mySensor.getSamplesPerSec()

VB:    mySensor.getSamplesPerSec() As Long
```

PARAMETERS

None.

RETURN VALUES

The currently set sampling rate, in samples per second.

COMMENTS

None.

setContinuousHSIFOff

Stop continuously reading samples from the high speed interface.

```
C:     long setContinuousHSIFOff(const long sensorHandle)

C++:   long mySensor.setContinuousHSIFOff()

VB:    mySensor.setContinuousHSIFOff() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_BOARDPARAMS_NOT_SET if the `setBoardParams()` function has not yet been called.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if successful.

COMMENTS

This function tells the library to stop continuously reading samples from the high speed interface's onboard memory. Any samples already in the CTI library's internal buffer are not affected, and remain available for retrieval by the application program.

Note that if the library does not periodically read samples from the high speed interface's onboard memory, eventually that memory will fill up, and samples will be lost.

This function does not change any of the other sensor settings.

setContinuousHSIFOn

Start continuously reading samples from the high speed interface.

```
C:     long setContinuousHSIFOn(const long sensorHandle)

C++:   long mySensor.setContinuousHSIFOn()

VB:    mySensor.setContinuousHSIFOn() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_BOARDPARAMS_NOT_SET if the `setBoardParams()` function has not yet been called.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if successful.

COMMENTS

Whenever the AR4000 sensor is powered on, the high speed interface card is continuously receiving sample data from the sensor, and storing it in the card's onboard memory. This function tells the CTI library to begin continuously reading those samples from the high speed interface and storing them in the library's internal buffer for retrieval by the application program.

By default, continuous sampling is on.

getIsContinuousHSIFOn

Determine if the library is continuously reading samples from the high speed interface.

```
C:     long getIsContinuousHSIFOn(const long sensorHandle)

C++:   long mySensor.getIsContinuousHSIFOn()

VB:    mySensor.getIsContinuousHSIFOn() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the library is continuously reading samples from the high speed interface and 0 (zero) otherwise.

COMMENTS

None.

getNumSamples

Returns the minimum number of samples currently available in the library's internal buffer.

```
C:     long getNumSamples(const long sensorHandle)

C++:   long mySensor.getNumSamples()

VB:    mySensor.getNumSamples() As Long
```

PARAMETERS

None.

RETURN VALUES

The minimum number of samples currently available in the library's internal buffer.

COMMENTS

This function returns a "snapshot" of the number of samples available in the library's internal buffer at a particular instant in time. If continuous sampling is on (ON is the default setting – see the `setContinuousHSIFOn()` function on page 40), the actual number of samples available at any future time may be larger than the value returned by this function, since the library is continuously processing incoming samples from the high speed interface and adding them to the library's buffer.

This function does *not* determine the number of samples in the high speed interface's hardware buffer.

getSamples

Returns range sample data from the high speed interface.

```
C:     long getSamples(const long sensorHandle,
                       HSIF_DATA_PT* dataBuffer,
                       const long bufSizeInSamples,
                       const long minNumSamples,
                       const long msWait)

C++:   long mySensor.getSamples(HSIF_DATA_PT* dataBuffer,
                                const long bufSizeInSamples,
                                const long minNumSamples,
                                const long msWait)

VB:    mySensor.getSamples(dataBuffer() As HSIF_DATA_PT,
                           ByVal minNumSamples As Long,
                           ByVal msWait As Long) As Long
```

PARAMETERS

`databuffer` – the data buffer to hold the range samples returned. Each sample consists of a structure of type HSIF_DATA_PT (see definition on page 43). It is the responsibility of the calling program to ensure that this buffer is large enough to hold all the required samples.

`bufSizeInSamples` – [Not required from VB] The data buffer size, in samples. Must be zero or greater.

`minNumSamples` – the minimum number of samples to acquire before returning to the calling program. Must be zero or greater.

`msWait` – the maximum time, in milliseconds, to wait for samples to be available. Must be zero or greater.

RETURN VALUES

CTI_BAD_PARAM if any parameter is out of range.
CTI_BOARDPARAMS_NOT_SET if the `setBoardParams()` function has not yet been called.
CTI_COMM_NOT_OPEN if this function is called before the serial port is opened with `setCommOpen()`.
CTI_WAIT_TIMEOUT if `minNumSamples` are not available within `msWait` milliseconds.

If the return value is negative, an error has occurred. `getErrorMessage()` may be used to get an extended error message. Otherwise, the function returns the actual number of samples read. A minimum of `minNumSamples` and a maximum of `bufSizeInSamples` will be returned. Each sample returned is a structure of type HSIF_DATA_PT, as defined below.

COMMENTS

This function removes samples from the library's internal buffer and returns them to the application program. Unless an error occurs, or the timeout period expires, the function will always return at least `minNumSamples` samples. If fewer than `minNumSamples` samples are available in the library's internal buffer when the function is called, the library enters an efficient wait state until enough additional samples are returned from the high speed interface. The library will wait for a maximum of `msWait` milliseconds. If `minNumSamples` are not available during this time, the library returns CTI_WAIT_TIMEOUT. This ensures that the function call will never 'hang' indefinitely.

It is the responsibility of the application programmer to specify a suitable timeout period based on the sensor sample rate and the filtering criteria set. For example, if the sensor sample rate is 100 samples/second, and the filtering criteria results in half of all samples being discarded, then the effective sample rate is 50 samples/second. If 500 samples are required, the timeout period must be at least 10,000 milliseconds (10 seconds). It is prudent to specify a timeout period at least 2 to 3 times higher than the minimum, to account for timing variances in the data collection.

HSIF_DATA_PT Structure

The structure below is used to return range data to C and C++ programs. A similar structure is used for Visual Basic – please see the file "`CTI_HSIF_PCI_Defs.bas`" in the installation directory for details.

```
/* C and C++ structure for returning high-speed interface data */
typedef struct {
   float R_X;          /* calibrated range, or X value of range          */
   float A1_Y;         /* encoder #1 angle, or Y value of range          */
   float A2;           /* encoder #2 angle                               */
   long  rawE1;        /* raw count for encoder #1                       */
   long  rawE2;        /* raw count for encoder #2                       */
   long  amplitude;    /* signal amplitude reading (0-255)               */
   long  ambient;      /* ambient light reading (0-255)                  */
   float temperature;  /* sensor internal temperature in degrees F.      */
   long  rawRange;     /* raw range reading                              */
   long  inputs;       /* lower 3 bits = 0/1 depending on inputs 1,2,3   */
} HSIF_DATA_PT;
```

This structure is used to return the sample values from the CTI-HSIF-PCI library to the application program. Each field in the structure is explained below:

`R_X` — If the current angular output format is Polar (the default), this field returns the calibrated range value. If the current angular output format is Cartesian (see `setAngleOutputCartesian()` on page 48), this field is the X value of the range.

`A1_Y` — If the current angular output format is Polar (the default), this field returns the angle of encoder #1, in degrees, referenced from the encoder's index pulse. If the current output format is Cartesian (see `setAngleOutputCartesian()` on page 48), this field is the Y value of the range.

`A2` — The angle of encoder #2, in degrees, referenced from the encoder's index pulse.

`rawE1` — The raw count from encoder #1, as given by the HSIF card's hardware counter.

`rawE2` — The raw count from encoder #2, as given by the HSIF card's hardware counter.

`amplitude` — The amplitude of the range reading, from 0-255. See the sensor hardware documentation for more details.

`ambient` — The ambient light value of the current range reading, from 0-255. See the sensor hardware documentation for more details.

`temperature` — The internal temperature of the AR4000 sensor, in degrees Fahrenheit.

`rawRange` — The 19-bit uncalibrated range measurement from the high-speed interface.

`inputs` — The status of the high speed interface's three general purpose input bits. The status of inputs 1, 2, and 3 are encoded in bits 0, 1 and 2 respectively of the `inputs` field. If the input is set at the time of the sample, then the corresponding bit in `inputs` is 1 (one), otherwise it is 0 (zero). When using the Line Scanner, input bit #1 is normally used to latch the encoder index pulse, so this bit being set will correspond with an encoder #1 count of zero.

getSamples2

Returns range samples from the high speed interface with the individual fields of the HSIF_DATA_PT structure (see above) as elements in separate arrays. This is useful for languages such as Delphi® that cannot manipulate C/C++ structures easily.

```
C: long getSamples2(const long sensorHandle,
                    const long minSamplesToTake,
                    const long msWait,
                    const long bufSizeSamples,
                    float* p_R_X,
                    float* p_A1_Y,
                    float* p_A2,
                    long*  p_rawE1,
                    long*  p_rawE2,
                    long*  p_amplitude,
                    long*  p_ambient,
                    float* p_temperature,
                    unsigned long* p_rawRange,
                    long* p_inputs)

C++:   long mySensor.getSamples2(const long minSamplesToTake,
                                 const long msWait,
                                 const long bufSizeSamples,
                                 float* p_R_X,
                                 float* p_A1_Y,
                                 float* p_A2,
                                 long*  p_rawE1,
                                 long*  p_rawE2,
                                 long*  p_amplitude,
                                 long*  p_ambient,
                                 float* p_temperature,
                                 unsigned long* p_rawRange,
                                 long* p_inputs)

VB:    mySensor.getSamples2(ByVal minSamples As Long, ByVal msWait As Long, _
                p_R_X() As Single, _
                p_A1_Y() As Single, _
                p_A2() As Single, _
                p_rawE1() As Long, _
                p_rawE2() As Long, _
                p_amplitude() As Long, _
                p_ambient() As Long, _
                p_temperature() As Single, _
                p_rawRange() As Long, _
                p_inputs() As Long) As Long
```

PARAMETERS

`minNumSamples` – the minimum number of samples to acquire before returning to the calling program. Must be zero or greater.

`msWait` – the maximum time, in milliseconds, to wait for samples to be available. Must be zero or greater.

`bufSizeSamples` – [Not required from VB] The size of the data buffers, in samples. Must be zero or greater..

`p_R_X, p_A1_Y, p_A2, p_rawE1, p_rawE2, p_amplitude, p_ambient, p_temperature, p_rawRange, p_inputs`

Arrays to hold the range samples returned. Upon return each array will hold elements consisting of the corresponding field of the `HSIF_DATA_PT` structure (see `HSIF_DATA_PT` definition on page 43). Each array MUST be the same length, as given by the bufSizeSamples parameter.

RETURN VALUES

See the getSamples function on page 42.

COMMENTS

This function is useful for languages such as Delphi®, which cannot manipulate C/C++ structures easily. There is a slight performance penalty when using this function versus the getSamples function (see page 42), therefore it is preferable to use the getSamples function whenever possible.

The comments for the getSamples function on page 42 also apply to this function.

getSingleSample

Turns on the laser, returns a single sample, then turns the laser off.

```
C:     long getSingleSample(const long sensorHandle, float * dp)

C++:   long mySensor.getSingleSample(float * dp)

VB:    mySensor.getSingleSample(dp As Single) As Long
```

PARAMETERS

`dp` – the variable to hold the data point returned.

RETURN VALUES

CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if the function succeeds. If the return value is not CTI_SUCCESS, an error has occurred. `getErrorMessage()` may be used to get an extended error message.

COMMENTS

If continuous sampling is on, calling `getSingleSample()` will turn it off (equivalent to calling the `setContinuousHSIFOff()` function.)

This function will discard any samples currently in the library's internal buffer.

This function is equivalent to the sensor's 'E1' command.

getExtSingleSample

Turns on the laser, returns a single sample, with extended sample data, then turns the laser off.

```
C:     long getExtSingleSample(const long sensorHandle, HSIF_DATA_PT* dp)

C++:   long mySensor.getExtSingleSample(HSIF_DATA_PT* dp)

VB:    mySensor.getExtSingleSample(dp As HSIF_DATA_PT) As Long

VBA:   mySensor.getExtSingleSample(dp() As HSIF_DATA_PT) As Long
```

PARAMETERS

`dp` – [C, C++, VB] a structure of type HSIF_DATA_PT, as defined on page 43, to hold the data point returned. [VBA] an array of type HSIF_DATA_PT.

RETURN VALUES

CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if the function succeeds. If the return value is not CTI_SUCCESS, an error has occurred. `getErrorMessage`() may be used to get an extended error message.

COMMENTS

This function uses the serial interface to return one sample from the AR4000 sensor. The HSIF_DATA_PT structure returned will have the calibrated range in the R_X member, and zero for the angular measurements, since the serial interface does not return encoder angle data.

If continuous sampling is on, calling `getExtSingleSample()` will turn it off (equivalent to calling the `setContinuousHSIFOff()` function.)

This function will discard any samples currently in the library's internal buffer.

This function is equivalent to the sensor's 'E3' command.

getNumDataLost

Return the number of samples having the 'data lost' flag set, due the high speed interface's onboard buffer overflowing.

```
C:     long mySensor.getNumDataLost(const long sensorHandle)

C++:   long mySensor.getNumDataLost()

VB:    mySensor.getNumDataLost() As Long
```

PARAMETERS

None.

RETURN VALUES

The number of samples which have the 'data lost' flag set when read from the high speed interface. This indicates that samples have been lost due to the high speed interface's onboard buffer overflowing.

COMMENTS

If the high speed interface's onboard buffer memory fills up, the first sample inserted in the card's buffer *after* space in the card's buffer becomes available will have it's 'data lost' flag set. See the sensor hardware documentation for further information.

If this function returns a non-zero value, it is an indication that the CTI-HSIF-PCI library is not reading samples from the high speed interface fast enough. Typically, this is because either other tasks are running simultaneously with the data collection program, leaving too little processor time available for the library's background data acquisition thread to run, or because the computer being used is too slow for the current sample rate. See the comments for the `getMaxSampleRate()` function on page **Error! Bookmark not defined.**.

Note that this function *cannot determine how many samples were actually lost*, only how many have the 'data lost' flag set, indicating that one or more samples were lost prior to that sample.

setResetDataLost

Reset the library's 'data lost' samples counter.

```
C:     long mySensor.setResetDataLost(const long sensorHandle)

C++:   long mySensor.setResetDataLost()

VB:    mySensor.setResetDataLost() As Long
```

PARAMETERS

None.

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

This function resets the library's internal 'data lost' samples counter, as returned by the `getNumDataLost()` function.

Data Format Functions

setAngleOutputPolar

Set the format for encoder angle measurements to polar (range, angle). This is the library's default.

```
C:     long setAngleOutputPolar(const long sensorHandle)

C++:   long mySensor.setAngleOutputPolar()

VB:    mySensor.setAngleOutputPolar() As Long
```

PARAMETERS

None.

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

By default, the library returns data points with the encoder #1 angle readings in polar format. In this format, the R_X member of the HSIF_DATA_PT structure is the reading's calibrated range measurement, and the A1_Y member of the structure is the reading's angular measurement.

See page 43 for a definition of the HSIF_DATA_PT structure.

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

setAngleOutputCartesian

Set the format for encoder angle measurements to Cartesian (X, Y).

```
C:     long setAngleOutputCartesian(const long sensorHandle)

C++:   long mySensor.setAngleOutputCartesian()

VB:    mySensor.setAngleOutputCartesian() As Long
```

PARAMETERS

None.

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

When set to Cartesian format, the library returns encoder #1 angle readings as (X, Y) pairs, with the X value of the reading in the R_X member of the HSIF_DATA_PT structure, and the Y value in the A1_Y member of the structure.

See page 43 for a definition of the HSIF_DATA_PT structure.

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

getIsAngleOutputPolar

Determine if the format for encoder #1 angle measurements is Polar or Cartesian.

```
C:     long getIsAngleOutputPolar(const long sensorHandle)

C++:   long mySensor.getIsAngleOutputPolar()

VB:    mySensor.getIsAngleOutputPolar() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the current angle output format for encoder #1 is Polar, and 0 (zero) if Cartesian.

COMMENTS

None.

setOutputFormatEnglish

Sets the data output format to English units (i.e. inches).

```
C:     long setOutputFormatEnglish(const long sensorHandle)

C++:   long mySensor.setOutputFormatEnglish()

VB:    mySensor.setOutputFormatEnglish() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if the function succeeds.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_FAILURE if an error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This is equivalent to the sensor's 'A1' command.

setOutputFormatMetric

Sets the data output format to metric units (i.e. millimeters).

```
C:     long setOutputFormatMetric(const long sensorHandle)

C++:   long mySensor.setOutputFormatMetric()

VB:    mySensor.setOutputFormatMetric() As Long
```

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if the function succeeds.
CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_FAILURE if an error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This is equivalent to the sensor's 'A4' command.

getIsOutputFormatEnglish

Determine if the current output format is English units (inches).

```
C:     long getIsOutputFormatEnglish(const long sensorHandle)

C++:   long mySensor.getIsOutputFormatEnglish()

VB:    mySensor.getIsOutputFormatEnglish() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the current output format is English units (i.e. inches), and 0 (zero) otherwise.

COMMENTS

None.

Data Filtering and Transformation Functions

In many data acquisition applications it is common to transform the raw data by scaling or adding offsets to the raw values. The CTI-HSIF-PCI library data provides a number of functions that permit these transformations to be done within the library, before the data is returned to the application program.

In addition, the data filtering functions enable the programmer to specify which samples acquired by the library will be returned to the application program. These functions should be used to discard samples that are not of interest, rather than performing this function within the application program itself. The library routines are fully tested and highly optimized, and use of these functions will be much less error-prone and more efficient than writing equivalent functionality within the application.

setAngleOffset

Set an offset to be applied to angle readings from the specified encoder.

```
C:     long setAngleOffset(const long sensorHandle,
                           const long encoderNum,
                           const float offset)

C++:   long mySensor.setAngleOffset(const long encoderNum, const float offset)

VB:    mySensor.setAngleOffset(ByVal encoderNum As Long) As Long
```

PARAMETERS

`encoderNum` – the encoder to set the angle offset for. Must be 1 or 2.

`offset` – the offset, in degrees, to be added to the angle measurement of each subsequent sample. This value may be positive or negative. The offset must be greater than –360.0 and less than +360.0 degrees.

The offset is added before any conversion from Polar to Cartesian coordinates is done, and before any comparison with the limits set by `setMaxValidAngle()` and `setMinValidAngle()`.

RETURN VALUES

CTI_BAD_PARAM if `encoderNum` is not 1 or 2, or if `offset` is out of range. CTI_SUCCESS otherwise.

COMMENTS

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

By default, the zero angle of the samples is at the encoder index pulse position, which may be at any arbitrary point around the circle, depending on the alignment of the encoder on the motor shaft. This function is useful for moving the zero angle position of the samples from that point to any desired position.

Note that the library will normalize the angle measurement of the returned samples to between 0 and 360 degrees regardless of the angle offset applied by this function. Thus, all sample angles will be greater than or equal to zero degrees, and less than 360.0 degrees.

getAngleOffset

Return the encoder angle offset.

```
C:     float getAngleOffset(const long sensorHandle, const long encoderNum)

C++:   float mySensor.getAngleOffset(const long encoderNum)

VB:    mySensor.getAngleOffset(ByVal encoderNum As Long) As Single
```

PARAMETERS

`encoderNum` – the encoder to get the offset for. Must be 1 or 2.

RETURN VALUES

CTI_BAD_PARAM if `encoderNum` is not 1 or 2.

Otherwise returns the current angle offset for the specified encoder, in degrees, as set by `setAngleOffset()`.

COMMENTS

None.

setDiscardInvalidOn

Discard any samples that fail the filtering criteria, and not return them to the application program.

```
C:     long setDiscardInvalidOn(const long sensorHandle)

C++:   long mySensor.setDiscardInvalidOn()

VB:    mySensor.setDiscardInvalidOn() As Long
```

PARAMETERS

None.

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

By default, the library sets any samples that fail the filtering criteria to zero, and returns them to the application program. (This is consistent with the default behaviour of the sensor hardware.) This function tells the library to instead discard these samples, and not return them to the application program.

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

setDiscardInvalidOff

Set to zero any samples that fail the filtering criteria, and return them to the application program.

```
C:     long setDiscardInvalidOff(const long sensorHandle)

C++:   long mySensor.setDiscardInvalidOff()

VB:    mySensor.setDiscardInvalidOff() As Long
```

PARAMETERS

None.

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

Calling `setDiscardInvalidOff()` tells the library to set to zero any samples that do not pass the filtering criteria and return these samples to the application program. This is the library's default behaviour.

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

getIsDiscardInvalidOn

Determine if invalid samples are to be discarded by the library.

```
C:     long getIsDiscardInvalidOn(const long sensorHandle)

C++:   long mySensor.getIsDiscardInvalidOn()

VB:    mySensor.getIsDiscardInvalidOn() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if samples that fail the filtering criteria are discarded by the library, and 0 (zero) otherwise.

COMMENTS

None.

setMaxValidAmbient

Sets the maximum ambient light reading threshold for a sample to be considered valid.

```
C:     long setMaxValidAmbient(const long sensorHandle, const long amb)

C++:   long mySensor.setMaxValidAmbient(const long amb)

VB:    mySensor.setMaxValidAmbient(ByVal amb As Long) As Long
```

PARAMETERS

`amb` – The value of the ambient light reading threshold. Valid range is 0 <= `amb` <= 255. `amb` must be greater than or equal to the value set by the `setMinValidAmbient()` function.

RETURN VALUES

CTI_BAD_PARAM if the parameter supplied is out of range, or less than the value set by the `setMinValidAmbient()` function.
CTI_SUCCESS if successful.

COMMENTS

Data points with ambient light readings equal to or below the threshold value set are considered valid, and returned to the application program by the library, while data points with readings above the value specified are set to zero or discarded, depending on the setting of the `setDiscardInvalidOn()` function (see page 52).

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

This function may be used to have the library automatically discard spurious data samples having high ambient light readings for example.

getMaxValidAmbient

Gets the currently set maximum valid ambient setting.

```
C:     long getMaxValidAmbient(const long sensorHandle)

C++:   long mySensor.getMaxValidAmbient()

VB:    mySensor.getMaxValidAmbient() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns the currently set maximum ambient light reading value that will be considered valid when returning samples from the library to the calling program.

COMMENTS

None.

setMinValidAmbient

Sets the minimum ambient light reading threshold for a sample to be considered valid.

```
C:     long setMinValidAmbient(const long sensorHandle, const long amb)

C++:   long mySensor.setMinValidAmbient(const long amb)

VB:    mySensor.setMinValidAmbient(ByVal amb As Long) As Long
```

PARAMETERS

amb – The value of the ambient light reading threshold. Valid range is 0 <= `amb` <= 255. `amb` must be less than or equal to the value set by the `setMaxValidAmbient()` function.

RETURN VALUES

CTI_BAD_PARAM if the parameter supplied is out of range, or more than the value set by the `setMaxValidAmbient()` function.
CTI_SUCCESS if successful.

COMMENTS

Data points with ambient light readings equal to or above the threshold value set are considered valid, and returned to the application program by the library, while data points with less than the value specified are set to zero or discarded, depending on the setting of the `setDiscardInvalidOn()` function (see page 52).

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

getMinValidAmbient

Gets the currently set minimum valid ambient setting.

```
C:     long getMinValidAmbient(const long sensorHandle)

C++:   long mySensor.getMinValidAmbient()

VB:    mySensor.getMinValidAmbient() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns the currently set minimum ambient light reading value that will be considered valid when returning samples from the library to the calling program.

COMMENTS

None.

setMaxValidAmplitude

Sets the maximum amplitude reading that will be considered valid.

```
C:     long setMaxValidAmplitude(const long sensorHandle, const long amp)

C++:   long mySensor.setMaxValidAmplitude(const long amp)

VB:    mySensor.setMaxValidAmplitude(ByVal amp As Long) As Long
```

PARAMETERS

`amp` – The maximum amplitude value that will be considered valid when returning samples from the library to the calling program. Samples with more than the value specified are set to zero or discarded, depending on the setting of the `setDiscardInvalidOn()` function (see page 52). Valid range is 0 <= `amp` <= 255. `amp` must be greater than or equal to the value set by the `setMinValidAmplitude()` function.

RETURN VALUES

CTI_BAD_PARAM if the parameter supplied is out of range, or less than the value set with the `setMinValidAmplitude()` function.
CTI_SUCCESS if successful.

COMMENTS

This function may be used to have the library set to zero or discard what may be inaccurate samples having a high amplitude reading resulting from sensor overload.

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

This function is equivalent to the sensor's 'M' command.

getMaxValidAmplitude

Gets the currently set maximum valid amplitude setting.

```
C:     long getMaxValidAmplitude(const long sensorHandle)

C++:   long mySensor.getMaxValidAmplitude()

VB:    mySensor.getMaxValidAmplitude() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns the currently set maximum amplitude value that will be considered valid when returning samples from the library to the calling program.

COMMENTS

None.

setMinValidAmplitude

Sets the minimum amplitude reading that will be considered valid.

```
C:     long setMinValidAmplitude(const long sensorHandle, const long amp)

C++:   long mySensor.setMinValidAmplitude(const long amp)

VB:    mySensor.setMinValidAmplitude(ByVal amp As Long) As Long
```

PARAMETERS

`amp` – The minimum amplitude value that will be considered valid when returning samples from the library to the calling program. Samples with less than the value specified are set to zero or discarded, depending on the setting of the `setDiscardInvalidOn()` function (see page 52). Valid range is 0 <= `amp` <= 255. `amp` must be less than or equal to the value set by the `setMaxValidAmplitude()` function.

RETURN VALUES

CTI_BAD_PARAM if the parameter supplied is out of range, or greater than the value set with the `setMaxValidAmplitude()` function.
CTI_SUCCESS if successful.

COMMENTS

This function can be used to have the library set to zero or discard what may be inaccurate samples having a low amplitude reading, which can result when measuring a target that provides weak return signals for the sensor.

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

This function is equivalent to the sensor's 'P' command.

getMinValidAmplitude

Gets the currently set minimum valid amplitude setting.

```
C:     long getMinValidAmplitude(const long sensorHandle)

C++:   long mySensor.getMinValidAmplitude()

VB:    mySensor.getMinValidAmplitude() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns the currently set minimum amplitude value that will be considered valid when returning samples from the library to the calling program.

COMMENTS

None.

setMaxValidAngle

Sets the maximum angle for a sample to be considered valid.

```
C:     long setMaxValidAngle(const long sensorHandle,
                             const long encoderNum,
                             const float ang)

C++:   long mySensor.setMaxValidAngle(const long encoderNum, const float ang)

VB:    mySensor.setMaxValidAngle(ByVal encoderNum As Long,
                                 ByVal ang As Single) As Long
```

PARAMETERS

`ang` – The maximum angle threshold.

`encoderNum` – the encoder number (1 or 2) to set the angle for.

RETURN VALUES

CTI_SUCCESS if successful.

COMMENTS

Data points with angle readings between that set with `setMinValidAngle()` and `setMaxValidAngle()` are kept by the library. All others are set to zero or discarded, depending on the setting of the `setDiscardInvalidOn()` function. This is true even if the angle set by `setMinValidAngle()` is numerically larger than that set by `setMaxValidAngle()`.

For example, setting a minimum valid angle of 20 degrees, using `setMinValidAngle()`, and a maximum of 100 degrees using `setMaxValidAngle()`, will keep samples with angles greater than or equal to 20 degrees and less than or equal to 100 degrees, and discard samples with angles less than 20 degrees and greater than 100 degrees.

Setting a minimum valid angle of 340 degrees, and a maximum of 30 degrees will keep samples with angles greater than or equal to 340 degrees and less than or equal to 30 degrees. Angles greater than 30 degrees and less than 340 degrees will be discarded.

To turn off maximum angle filtering, call this function with a value greater than 360.

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

When using the line scanner, this function may be used to have the library automatically discard data samples outside of a specific field of view.

getMaxValidAngle

Gets the currently set maximum valid angle setting.

```
C:     float getMaxValidAngle(const long sensorHandle, const long encoderNum)

C++:   float mySensor.getMaxValidAngle(const long encoderNum)

VB:    mySensor.getMaxValidAngle(ByVal encoderNum As Long) As Single
```

PARAMETERS

`encoderNum` – the encoder number (1 or 2) to get the angle for.

RETURN VALUES

Returns the currently set maximum angle value that will be considered valid when returning samples from the library to the calling program.

COMMENTS

None.

setMinValidAngle

Sets the minimum angle for a sample to be considered valid.

```
C:     long setMinValidAngle(const long sensorHandle,
                             const long encoderNum,
                             const float ang)

C++:   long mySensor.setMinValidAngle(const long encoderNum, const float ang)

VB:    mySensor.setMinValidAngle(ByVal encoderNum As Long,
                                 ByVal ang As Single) As Long
```

PARAMETERS

`ang` – The minimum angle threshold in degrees.

`encoderNum` – the encoder number (1 or 2) to set the angle for.

RETURN VALUES

CTI_SUCCESS if successful.

COMMENTS

Data points with angle readings between that set with `setMinValidAngle()` and `setMaxValidAngle()` are kept by the library. All others are set to zero or discarded, depending on the setting of the `setDiscardInvalidOn()` function. This is true even if the angle set by `setMinValidAngle()` is numerically larger than that set by `setMaxValidAngle()`.

For example, setting a minimum valid angle of 20 degrees, using `setMinValidAngle()`, and a maximum of 100 degrees using `setMaxValidAngle()`, will keep samples with angles greater than or equal to 20 degrees and less than or equal to 100 degrees, and discard samples with angles less than 20 degrees and greater than 100 degrees.

Setting a minimum valid angle of 340 degrees, and a maximum of 30 degrees will keep samples with angles greater than or equal to 340 degrees and less than or equal to 30 degrees. Angles greater than 30 degrees and less than 340 degrees will be discarded.

To turn off minimum angle filtering, call this function with a value less than zero.

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

When using the line scanner, this function may be used to have the library automatically discard data samples outside of a specific field of view.

getMinValidAngle

Gets the currently set minimum valid angle setting.

```
C:     float getMaxValidAngle(const long sensorHandle, const long encoderNum)

C++:   float mySensor.getMaxValidAngle(const long encoderNum)

VB:    mySensor.getMaxValidAngle(ByVal encoderNum As Long) As Single
```

PARAMETERS

`encoderNum` – the encoder number (1 or 2) to get the angle for.

RETURN VALUES

Returns the currently set minimum angle value that will be considered valid when returning samples from the library to the calling program.

COMMENTS

None.

setMaxValidRange

Sets the maximum (calibrated) range reading that will be considered valid.

```
C:     long setMaxValidRange(const long sensorHandle, const float rng)

C++:   long mySensor.setMaxValidRange(const float rng)

VB:    mySensor.setMaxValidRange(ByVal rng As Single) As Long
```

PARAMETERS

`rng` – The maximum range value that will be considered valid when returning samples from the library to the calling program. The value will be interpreted as being in the currently set sensor output units. (i.e. in inches if the current sensor output mode is English units, and millimeters if the current output mode is metric units.) `rng` may be any value, including negative values, since the library will return negative range values if a negative range offset is set. `rng` must be greater than or equal to the value set with the `setMinValidRange()` function.

RETURN VALUES

CTI_BAD_PARAM if rng is less than the value set with the `setMinValidRange()` function.
CTI_SUCCESS if successful.

COMMENTS

This function is useful if it is known that the objects the sensor is intended to measure are within a certain range. Using this function, samples with more than the value specified are set to zero or discarded, depending on the setting of the `setDiscardInvalidOn()` function (see page 52).

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

The maximum valid range test is done *after* any offset specified with `setRangeOffset()`and any scale factor specified with `setRangeScaleFactor()` are applied to the original range value from the sensor.

Do not confuse this function with the `setSensorMaxRange()` function, which changes the sensor's hardware parameters to facilitate better resolution and higher sample rates. (See page 30 for details.)

getMaxValidRange

Gets the currently set maximum valid range setting.

```
C:     float getMaxValidRange(const long sensorHandle)

C++:   float mySensor.getMaxValidRange()

VB:    mySensor.getMaxValidRange() As Single
```

PARAMETERS

None.

RETURN VALUES

Returns the currently set maximum range value that will be considered valid when returning samples from the library to the calling program. This value should be considered as inches if the current sensor output format is English units, or millimeters if the current sensor output mode is metric units.

COMMENTS

Do not confuse this function with `getSensorMaxRange()`, which determines the maximum distance the sensor hardware is currently configured to measure.

setMinValidRange

Sets the minimum (calibrated) range reading that will be considered valid.

```
C:     long setMinValidRange(const long sensorHandle, const float rng)

C++:   long mySensor.setMinValidRange(const float rng)

VB:    mySensor.setMinValidRange(ByVal rng As Single) As Long
```

PARAMETERS

`rng` – The minimum range value that will be considered valid when returning samples from the library to the calling program. The value will be interpreted as being in the currently set sensor output units. (i.e. in inches if the current sensor output mode is English units, and millimeters if the current output mode is metric units.) `rng` may be any value, including negative values, since the library will return negative range values if a negative range offset is set. `rng` must be less than or equal to the value set with the `setMaxValidRange()` function.

RETURN VALUES

CTI_BAD_PARAM if rng is greater than the value set with the `setMaxValidRange()` function.
CTI_SUCCESS if successful.

COMMENTS

This function is useful if it is known that the objects the sensor is intended to measure are within a certain range. Using this function, samples with less than the value specified are set to zero or discarded, depending on the setting of the `setDiscardInvalidOn()` function (see page 52).

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

The minimum valid range test is done *after* any offset specified with `setRangeOffset()`and any scale factor specified with `setRangeScaleFactor()` are applied to the original range value from the sensor.

getMinValidRange

Gets the currently set minimum valid range setting.

```
C:     float getMinValidRange(const long sensorHandle)

C++:   float mySensor.getMinValidRange()

VB:    mySensor.getMinValidRange() As Single
```

PARAMETERS

None.

RETURN VALUES

Returns the currently set minimum range value that will be considered valid when returning samples from the library to the calling program. This value should be considered as inches if the current sensor output format is English units, or millimeters if the current sensor output mode is metric units.

COMMENTS

None.

setRangeOffset

Sets the range offset to be added to all subsequent samples.

```
C:     long setRangeOffset (const long sensorHandle, const float offset)

C++:   long mySensor.setRangeOffset(const float offset)

VB:    mySensor.setRangeOffset(ByVal offset As Single) As Long
```

PARAMETERS

`offset` – The offset to be added to each subsequent sample returned by the library to the application program. An amount may be subtracted from each sample by specifying a negative offset. The value will be interpreted as being in the currently set sensor output units. (i.e. in inches if the current sensor output mode is English units, and millimeters if the current output mode is metric units.)

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

This function may be used to have the library add a constant offset to each range sample. Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned to the application program by the library.

This function may be used change the sensor's zero range reference point from the hardware default to any desired position. For example, when using the line scanner, the reference point can be set to the mirror's centerline by adding the appropriate offset.

The range offset is added to the original value returned from the sensor *before* any comparison with the limits set with the `setMaxValidRange()` or `setMinValidRange()` functions is done, and before any scale factor specified with `setRangeScaleFactor()` is applied.

getRangeOffset

Gets the currently set range offset setting.

```
C:     float getRangeOffset(const long sensorHandle)

C++:   float mySensor.getRangeOffset()

VB:    mySensor.getRangeOffset() As Single
```

PARAMETERS

None.

RETURN VALUES

Returns the currently set range offset setting. This value should be considered as inches if the current sensor output format is English units, or millimeters if the current sensor output mode is metric units.

COMMENTS

None.

setRangeScaleFactor

Sets the range scale factor by which all subsequent samples will be multiplied.

```
C:     long setRangeScaleFactor(const long sensorHandle, const float scale)

C++:   long mySensor.setRangeScaleFactor(const float scale)

VB:    mySensor.setRangeScaleFactor(ByVal scale As Single) As Long
```

PARAMETERS

`scale` – The scale factor by which each subsequent sample will be multiplied.

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

This function may be used to have the library multiply each range sample by a constant scale factor. This setting takes effect immediately, and will apply to all future samples returned to the application program by the library.

Calling this function will discard any data samples currently in the library's internal buffer.

The range is multiplied by the specified scale factor *after* any offset specified with the `setRangeOffset()` function is added to the original value from the sensor and *before* any comparison with the limits set with the `setMaxValidRange()` or `setMinValidRange()` functions is done.

getRangeScaleFactor

Gets the currently set range scale factor.

```
C:     float getRangeScaleFactor(const long sensorHandle)

C++:   float mySensor.getRangeScaleFactor()

VB:    mySensor.getRangeScaleFactor() As Single
```

PARAMETERS

None.

RETURN VALUES

Returns the currently set range scale factor.

COMMENTS

None.

setMaxValidTemp

Sets the maximum sensor temperature value for a sample to be considered valid.

```
C:     long setMaxValidTemp(const long sensorHandle, const float temp)

C++:   long mySensor.setMaxValidTemp(const float temp)

VB:    mySensor.setMaxValidTemp(ByVal temp As Single) As Long
```

PARAMETERS

`temp` – The maximum temperature threshold in degrees Fahrenheit. Must be greater than or equal to the value set with the `setMinValidTemp()` function.

RETURN VALUES

CTI_BAD_PARAM if the parameter supplied is less than the value set with `setMinValidTemp()`.
CTI_SUCCESS if successful.

COMMENTS

Data points with sensor temperatures equal to or below the value set are considered valid, and returned to the application program by the library, while data points with readings above the value specified are set to zero or discarded, depending on the setting of the `setDiscardInvalidOn()` function (see page 52).

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

The sensor is factory calibrated at 95 degrees Fahrenheit. This function may be used to have the library automatically discard data samples that are collected when the sensor's internal temperature is significantly above the calibration temperature, and consequently may be inaccurate.

This function should not be confused with the `setTempHoldLevel()` function (see page 32), which sets the sensor's internal temperature hold level.

getMaxValidTemp

Gets the currently set maximum valid temperature setting.

```
C:     float getMaxValidTemp(const long sensorHandle)

C++:   float mySensor.getMaxValidTemp()

VB:    mySensor.getMaxValidTemp() As Single
```

PARAMETERS

None.

RETURN VALUES

Returns the currently set maximum sensor temperature that will be considered valid when returning samples from the library to the calling program.

COMMENTS

None.

setMinValidTemp

Sets the minimum sensor temperature value for a sample to be considered valid.

```
C:     long setMinValidTemp(const long sensorHandle, const float temp)

C++:   long mySensor.setMinValidTemp(const float temp)

VB:    mySensor.setMinValidTemp(ByVal temp As Single) As Long
```

PARAMETERS

`temp` – The minimum temperature threshold in degrees Fahrenheit. Must be less than or equal to the value set with the `setMaxValidTemp()` function.

RETURN VALUES

CTI_BAD_PARAM if the parameter supplied is more than the value set with `setMaxValidTemp()`.
CTI_SUCCESS if successful.

COMMENTS

Data points with sensor temperatures equal to or above the value set are considered valid, and returned to the application program by the library, while data points with readings below the value specified are set to zero or discarded, depending on the setting of the `setDiscardInvalidOn()` function (see page 52).

Calling this function will discard any data samples currently in the library's internal buffer. This setting takes effect immediately, and will apply to all future samples returned by the library.

The sensor is factory calibrated at 95 degrees Fahrenheit. This function may be used to have the library automatically discard data samples that are collected when the sensor's internal temperature is significantly below the calibration temperature, and consequently may be inaccurate.

This function should not be confused with the `setTempHoldLevel()` function (see page 32), which sets the sensor's internal temperature hold level.

getMinValidTemp

Gets the currently set minimum valid temperature setting.

```
C:     float getMinValidTemp(const long sensorHandle)

C++:   float mySensor.getMinValidTemp()

VB:    mySensor.getMinValidTemp() As Single
```

PARAMETERS

None.

RETURN VALUES

Returns the currently set minimum sensor temperature that will be considered valid when returning samples from the library to the calling program.

COMMENTS

None.

Error Handling and Miscellaneous Functions

getIsError

Determine if an error has been detected.

```
C:     long getIsError(const long sensorHandle)

C++:   long mySensor.getIsError()

VB:    mySensor.getIsError() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the library has detected an error, and 0 (zero) otherwise.

COMMENTS

This function may be used to determine if the software library has detected an error condition. If the function returns 1 (one), then the `getErrorMessage()` function may be used to get an extended error message describing the error that has occurred. The error indicator flag will remain set until reset with the `setClearError()` function.

getErrorMessage

Return extended error message.

```
C:     long getErrorMessage(const long sensorHandle,
                            char * pErrMsg,
                            const long bufLen)

C++:   long mySensor.getErrorMessage(char * pErrMsg, const long bufLen)

VB:    mySensor.getErrorMessage(ByRef errMsg As String) As Long
```

PARAMETERS

`pErrMsg` – [C, C++] Pointer to a buffer to hold the error message text. To ensure that the full error message is returned, this buffer should be capable of holding at least 256 characters.

`errMsg` – [VB]. A String variable to hold the error message returned from the library.

`bufLen` – [Not required from Visual Basic]. The length of the buffer pointed to, in characters.

RETURN VALUES

Returns CTI_BUFFER_TOO_SMALL if the buffer pointed to is too small to hold the full text of the error message. Returns CTI_SUCCESS otherwise.

If an error has been detected by the library, then the text of the error message will be copied into the buffer pointed to by `pErrMsg`, [C, C++] or the variable `errMsg` [VB]. If the buffer is too small to hold the full error message, then as many characters as will fit in the space provided will be copied. If multiple errors occur, only the text of the first error will be returned. If no errors have been detected by the library, an empty string will be returned.

COMMENTS

The library's error indicator flag and error message text may be reset using `setClearError()`.

setClearError

Resets the CTI software library's internal error flag, and clears the library's internal error message text.

```
C:     long setClearError(const long sensorHandle)

C++:   long mySensor.setClearError()

VB:    mySensor.setClearError() As Long
```

PARAMETERS

None.

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

If the software library has detected an error condition, this function will reset the library's internal error indicator flag and clear the error message text. The `setCommOpen()` function also resets the error indicator and message text.

getFirmwareVersion

Get the sensor firmware version number.

```
C:     long getFirmwareVersion(const long sensorHandle,
                               char * version,
                               const long versionStringLen)

C++:   long mySensor.getFirmwareVersion(char * version,
                               const long versionStringLen))

VB:    mySensor.getFirmwareVersion(ByRef version As String) As Long
```

PARAMETERS

`version` – [C, C++] A character buffer to hold the returned string. [VB] A String variable to hold the returned version information.

`versionStringLen` – [Not required from Visual Basic]. The length of the string buffer. In order to return the full version string from the sensor, this should be at least 10 characters long. If this buffer is too short to return the entire firmware version string, then only as many characters as can fit in the actual buffer space will be returned.

RETURN VALUES

CTI_SUCCESS if the function succeeds.

CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.

CTI_BUFFER_TOO_SMALL if the buffer is too small to hold the full length of the firmware version string.

CTI_FAILURE if any other error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This function returns the version of the Acuity AR4000 sensor's firmware. Use `getLibraryVersion()` (see page 74) to get the version of the Crandun Technologies library software.

This function is equivalent to the sensor's 'V' command.

getLibraryName

Get the CTI software library name and version as a string.

```
C:     long getLibraryName(const long sensorHandle,
                           char * libName,
                           const long stringLen)

C++:   long mySensor.getLibraryName(char * libName,
                           const long stringLen)

VB:    mySensor.getLibraryName(ByRef libName As String) As Long
```

PARAMETERS

`libName` – [C, C++] A character buffer to hold the returned string. [VB] A String variable to hold the returned library name.

`stringLen` – [Not required from Visual Basic]. The length of the string buffer. In order to return the full string, this should be at least 40 characters long. . If this buffer is too short to return the entire library name version string, then only as many characters as will fit in the actual buffer space will be returned.

RETURN VALUES

CTI_SUCCESS if the function succeeds.

CTI_BUFFER_TOO_SMALL if the buffer is too small to hold the full length of the version string. As much of the version string as will fit in the actual buffer space is returned.

CTI_FAILURE if an error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This function returns the name and version number of the Crandun Technologies library software.

Use `getFirmwareVersion()` (see page 72) to get the Acuity sensor hardware's firmware version.

getLibraryVersion

Get the version number of the CTI software library as a numeric value.

```
C:     long getLibraryVersion(const long sensorHandle)

C++:   long mySensor.getLibraryVersion()

VB:    mySensor.getLibraryVersion() As Long
```

PARAMETERS

None.

RETURN VALUES

Returns the version number of the CTI Library as a numeric value.

COMMENTS

This function returns the version of the Crandun Technologies library software as a numeric value. Use the `getLibraryName()` function to return the library's name and version as a string.

This function may be used by a programmer to ensure that the current CTI library version matches that required by the particular application.

Use `getFirmwareVersion()` (see page 72) to get the Acuity sensor hardware's firmware version.

Appendix A - Installation & Components

Microsoft Windows

Installation Directions

Before installing the Crandun Technologies CTI-HSIF-PCI Software library, the Acuity software shipped with the PCI HSIF card must be installed. Please follow the Acuity instructions to install the software shipped with the card.

To install the Crandun software library, insert the Software Library distribution CD into your computer's CDROM drive. After a few seconds, the installation should start automatically. If not, run the file "`CTI-HSIF-PCI-Setup.EXE`" located in the `Windows` subdirectory of the CD.

During installation you will be asked if you want the installation program to check the Crandun Technologies web site for an updated version of the library. It is strongly recommended that you do so, to ensure that the version you are using is the most up to date.

The installation will prompt for a directory into which to install the library components. The default directory is "`C:\Program Files\Crandun Technologies\CTI-HSIF-PCI`", although this may be changed if desired. All source code files, sample programs and manuals are installed in this directory. The files `CTI_HSIF.DLL` and `CTIDRV1.DLL` are always installed in the "`Windows\System`" directory regardless of the installation directory chosen. See the section "Library Components" below, for more details of the specific files installed.

Please enter your license keycode when prompted for it by the installation program. You must have a valid keycode for each licensed copy of the CTI-HSIF-PCI library.

At the end of the installation, you will be asked if you wish to run the installation verification program. It is strongly recommended that you do so to verify that the software has been correctly installed, and that communications with the AR4000 sensor and high-speed interface card can be successfully established.

Use of the verification program is self-explanatory, and its results will be displayed on the screen.

Should the verification program fail, please see the section "Installation Verification Program" in the Programmer's Guide for possible reasons.

Library Components

The installation program installs the components listed in the table below. Each of these components are required to develop programs that use the CTI-HSIF-PCI library. As shown, some files are copied into the `Windows\System` and `Windows\System32` directories. All others are copied into the `\Source` subdirectory of the installation directory (by default `C:\Program Files\Crandun Technologies\CTI-HSIF-PCI`).

The sample programs are installed in the `\Samples` subdirectory. (See "Sample Programs" in the Programmer's Guide).

Directory	Files	Description

`Windows\System`
 	
`Windows\System32\Drivers`
	`CTI_HSIF.DLL`
	`CTIDRV1.DLL`
	`CTIDRV1.VXD`
	
	`CTIDRV1.SYS`
		These files implement the majority of the functionality of the CTI-HSIF-PCI library. They are required for any application using the library. CTIDRV1.VXD is installed for Windows 9x/Me, and CTIDRV1.SYS is installed for all other Windows versions.

`instdir\Source\C_CPP`	`CTI_HSIF_PCI_DLL.lib`	This file defines the functions exported by CTI_HSIF.DLL. C and C++ applications must link with this file.
`instdir\Source\C_CPP`	`CTI_HSIF_PCI.h`	This header file defines all the functions provided by the library. It must be included in any C or C++ program using the library.
`instdir\Source\C_CPP`	`CTI_ErrCodes.h`	Defines the error codes returned by the library functions. It is included by CTI_HSIF_PCI.h
`instdir\Source\C_CPP`	`CTI_HSIFDataPt.h`	Defines the structure used to return low-level sensor information to the application. It is included by CTI_HSIF_PCI.h
`instdir\Source\VB`	`CTI_HSIF_PCI.cls`	Defines the Visual Basic CTI_HSIF class object. Must be included as a Class Module in any Visual Basic application using the library.
`instdir\Source\VB`	`CTI_HSIF_PCI_Defs.bas`	Defines the status codes and data structures returned by the library. It must be included as a Code Module in any Visual Basic or Visual Basic for Applications program using the library.
`instdir\Source\VBA`	`CTI_HSIF_PCI.cls`	Defines the Visual Basic for Applications CTI_HSIF class object. Must be included as a Class Module in any Visual Basic for Applications (VBA) program using the library.

Linux

The CTI-HSIF-PCI Software Library is currently not supported on Linux. Please contact Crandun Technologies Inc. at 905-692-0012, or www.crandun.com for availability details.

Appendix B - Distributing Software Created using the Library

Please note: The End-User License Agreement you agreed to during installation of this software defines the terms under which components distributed with the CTI-HSIF-PCI library may be redistributed. This section of the manual discusses specifically which components of the library may be redistributed.

However, at all times, the End-User License Agreement remains the definitive document for determining permissible uses of any component of the CTI-HSIF-PCI Library. Nothing in this manual should be interpreted as modifying the terms of the End-User License Agreement.

The components listed in the following table may be redistributed as part of a value-added application that uses the CTI-HSIF-PCI library. ***No other files or components of the CTI-HSIF-PCI Library package may be redistributed.***

If you distribute an application or library that incorporates the CTI-HSIF-PCI Software Library, the application must add significant value to the API provided by the CTI-HSIF-PCI library alone, and cannot be merely a simple "wrapper" around the library functionality. Please refer to the End-User License Agreement for details.

Microsoft Windows Version – Redistributable Files

Directory	File s	Description

`Windows\System`
 	
`Windows\System32\Drivers`
	`CTI_HSIF.DLL`
	`CTIDRV1.DLL`
	`CTIDRV1.VXD`
	
	`CTIDRV1.SYS`
		These DLLs implement the majority of the functionality of the CTI-HSIF-PCI library. They are required for any application using the library.

`instdir\Source\C_CPP`	`CTI_ErrCodes.h`	This header file defines the error codes returned by the library functions.
`instdir\Source\C_CPP`	`CTI_HSIFDataPt.h`	Defines the structure used to return range data and other sensor information to the application.
`instdir\Source\VB`	`CTI_HSIF_PCI.cls`	This file defines the Visual Basic CTI_HSIF class object. It must be included as a Class Module in any Visual Basic application using the library.
`instdir\Source\VB`	`CTI_HSIF_PCI_Defs.bas`	Defines the status codes and data structures returned by the CTI-HSIF-PCI library. It must be included as a Code Module in any Visual Basic or Visual Basic for Applications program using the library.
`instdir\Source\VBA`	`CTI_HSIF_PCI.cls`	This file defines the Visual Basic for Applications CTI_HSIF class object. It must be included as a Class Module in any Visual Basic for Applications (VBA) program using the library.