CTI-AR600 Software Library

for

Acuity AR600
Laser Distance Sensors

Programmer's Guide

and

Reference Manual

Crandun Technologies Inc.

V1.05A

Printed in Canada

Table of Contents

PART II – REFERENCE MANUAL

Intended Audience

This document is intended for:

- End-users who will install and use the CTI Software Library to acquire data from Acuity AR600 sensor systems 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 AR600 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. It is assumed that software developers using the CTI-AR600 library are fully familiar with the programming language and development tools being used and are experienced in developing and debugging applications using that language. All users of this library should be fully familiar with the AR600 sensor hardware, and have read all appropriate Acuity documentation.

Support Information

If you have questions or problems regarding the CTI-AR600 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 hardware and cabling is properly installed and functioning. Consult the Acuity hardware documentation for instructions on verifying the operation of the sensor 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-AR600 software library is a set of Dynamic Link Libraries (DLLs), object files and header files that provide a high-performance, high-level interface to the functionality of the Acuity AR600 series laser distance sensors. The libraries permit the easy and rapid development of application programs to acquire data from the Acuity AR600 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, C++ and Visual Basic sample programs and Microsoft Excel spreadsheets, which demonstrate various features of the library, and illustrate techniques for making effective use of the library's capabilities. Please see the section "Sample Programs" on page 30 for more details.

Highlights

- High performance data acquisition using the AR600 serial interface.
- Usable from C, C++, Visual Basic, and Visual Basic for Applications.
- Easy integration of data from other instrumentation along with range data from the AR600.
- Control multiple sensors from a single computer.
- Access to the complete functionality provided by the AR600.
- Simple, intuitive, english-like application programming interface (API) to the AR600 command set.
- User-configurable data buffering and callback notification.
- User-configurable filtering, based on amplitude, range, ambient light, or sensor temperature.
- Fully configurable serial port baud rate and flow control.
- Maintains state information, for easy determination of current sensor settings.
- Multi-threaded and interrupt driven, for highest performance.

Features

The CTI-AR600 software library has been designed for ease of use and maximum flexibility and performance. No programming is needed to acquire data from the AR600 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 Formatting and Filtering

The AR600's serial interface can transmit ASCII or binary data to the host computer. ASCII format, although more convenient, limits the maximum sample rate due to the additional data sent over the serial link. The library uses binary data transmission, with conversion on the host computer, to maximize sample rates. Either English or metric output units may be selected, a user specified offset and/or scale factor may be added to the samples and the library can optionally filter the raw data stream to exclude samples outside of a particular range of interest.

Configurable Data Buffering and Callback Notification

A user configurable buffer within the library may be defined, which will be filled as data arrives over the serial 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.

Easy Integration of Data from Other Instrumentation

The CTI-AR600 library provides a full set of functions to enable the easy acquisition and integration of data from other types of sensors simultaneously with the range data from the AR600.

Multi-threaded and Interrupt Driven

Internally, the CTI-AR600 library is fully multi-threaded and interrupt driven for maximum performance.

Installing the Software

To install the software, insert the distribution CD into your computer's CDROM drive. After a few seconds, the installation should start automatically. If not, run the file "`CTI-AR600-Setup.EXE`" located on 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-AR600`", although this may be changed if desired. All source code files, sample programs and manuals are installed in this directory. The file `CTI_AR600.DLL` is 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-AR600 library.

Installation Verification Program

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 AR600 sensor 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. If this occurs, the installation program will prompt you to re-enter the keycode.
- The sensor's communications port is not configured for the factory default of 9600 baud. Please refer to the Acuity 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 Acuity 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, HyperTerminal or an Internet connection).

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" on page v).

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-AR600 library. The `CTI_AR600.DLL` file is copied into the `Windows\System` directory. All other components are copied into the `\Source` subdirectory of the installation directory (by default `C:\Program Files\Crandun Technologies\CTI-AR600`).

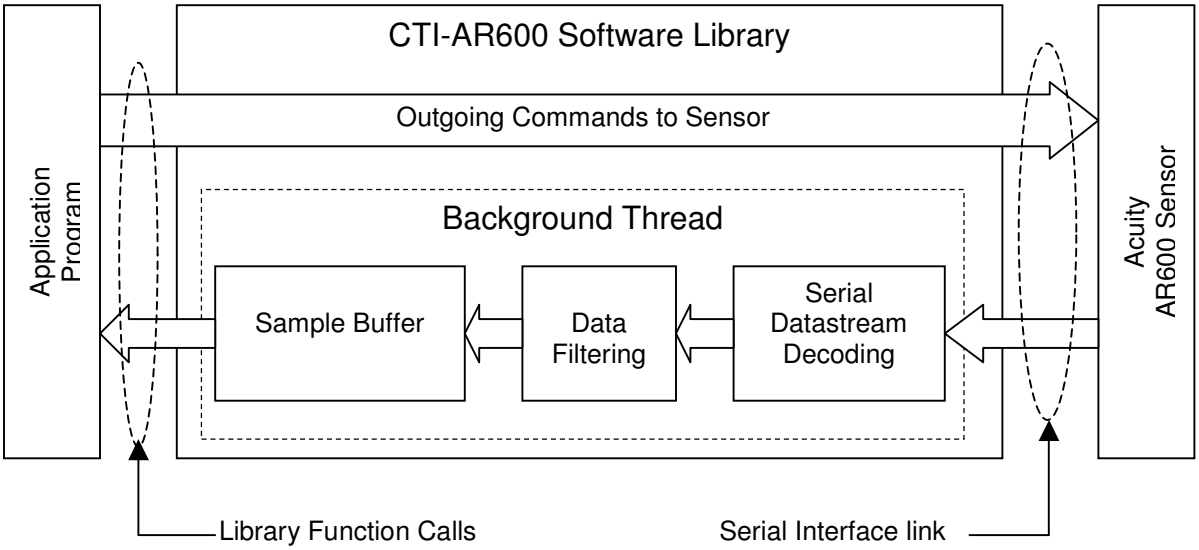
In addition, a number of demonstration and sample programs are installed in the `\Samples` subdirectory. (See "Sample Programs" – page 30).

Directory	Files	Description
`Windows\System`	`CTI_AR600.DLL`	This DLL implements the majority of the functionality of the CTI-AR600 library. It is required for any application using the library.
`instdir\Source\C_CPP`	`CTI_AR600.lib`	This file defines the functions exported by CTI_AR600.DLL. C and C++ applications must link with this file.
`instdir\Source\C_CPP`	`CTI_AR600.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_AR600.h
`instdir\Source\VB`	`CTI_AR600.cls`	Defines the Visual Basic CTI_AR600 class object. Must be included as a Class Module in any Visual Basic application using the library.
`instdir\Source\VB`	`CTI_AR600_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_AR600.cls`	Defines the Visual Basic for Applications CTI_AR600 class object. Must be included as a Class Module in any Visual Basic for Applications (VBA) program using the library.

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-AR600 library is an interface layer between the Acuity AR600 sensor hardware and the application program(s). The library communicates with the sensor via the computer's serial port, 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 over the computer's serial interface, and ensure that both outbound data (commands sent to the sensor), and inbound data (range samples from the sensor) 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, the outbound data is passed directly to the computer's serial port for transmission to the sensor. The library will not return control to the application program until the command has been sent to the sensor. Inbound data however, goes through a more complex series of steps, as explained below.

Background Processing Thread

A key design concept of the CTI-AR600 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 until the `setCommClosed()` function is called. *Without any interaction by the application programmer*, the

[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.

thread reads data from the computer's serial port as it is received from the sensor and passes it to the other parts of the library for subsequent processing.

Since, during use, the AR600 sensor is typically outputting a continuous stream of data samples over its serial 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.

Serial Datastream Decoding

This functional block decodes the raw serial datastream from the sensor into range samples. The library checks for the presence of the framing bytes in each incoming sample, and synchronizes the datastream accordingly. Each raw sample is decoded into a range value which is then passed on to the data filtering block.

Data Filtering

This functional block applies any filtering and transformation rules set by the application programmer 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 AR600 sensor is done through one of the functions provided by the CTI-AR600 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-AR600 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 Configuration Functions

These functions interact with the AR600 hardware to configure the sensor. The CTI Library tracks the current state of each sensor setting, for easy determination by an application programmer.

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..

Auxiliary Communications Port Functions

These functions permit the easy acquisition and integration of data received from a second serial port with the range data received from the AR600 laser. For example, another type of sensor can be attached to a second serial port, and its data measured at the same time as the AR600's data.

Data Format Functions

These functions set the sample units to English units (inches) or metric units (millimetres).

Data Filtering and Transformation Functions

These functions allow filtering the raw samples from the sensor to exclude samples outside of a desired range, add an offset or scale factor to each sample, etc.. These functions are explained in detail in the section "Data Filtering" on page 23 of the Programmer's Guide, and the section "Data Filtering and Transformation Functions" starting on page 44 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-AR600 library – the simplest possible program that communicates with the sensor, 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 the serial connection to the sensor and acquires data samples into an array. Each range sample is then printed out and then the serial connection to the sensor is closed. Please see the source code listing below, and the explanation following the listing for more details.

```
   #include <iostream>
A  #include "CTI_AR600.h"     // required header

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

   int main()
   {
      char c;
C     long rc;
      const int maxSamples = 20;
      float fullScaleSpan, rangeData[maxSamples];
D     CTI_AR600 my_AR600;

E     cout << "Crandun Technologies CTI-AR600 Library 'Hello World' example." << endl;
      cout << "Enter the Full Scale Span of the sensor you are using (e.g. 4.0): ";
      cin >> fullScaleSpan;

F     rc = my_AR600.setCommOpen("COM1", 9600, fullScaleSpan);
G     if (rc != CTI_SUCCESS) {
         cerr << "ERROR: setCommParams returned error: " << rc << endl;
         return -1;
      }

      cout << "Reading samples from sensor ..." << endl;
H     rc = my_AR600.getSamples(rangeData, maxSamples, 10);
I     if (rc < 0) {
         cerr << "ERROR: getSamples returned error: " << rc << endl;
         my_AR600.setCommClosed();
         return -1;
      }

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

K     cout << my_AR600.getNumBytesSkipped() << " bytes of data were skipped." << endl;

L     my_AR600.setCommClosed();
      cout << "Please 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-AR600 library. This header file must be included by any program using the CTI-AR600 library.

B. All C++ classes and methods defined by the CTI-AR600 library reside in namespace "`Crandun`". The "`using namespace Crandun`" statement should be included by any C++ program using CTI-AR600 library. If not included, then each symbol defined by the CTI-AR600 library must be explicitly qualified when referenced (e.g. `Crandun::CTI_AR600 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-AR600 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_AR600`". At this point, the object does not yet refer to any specific sensor that may be attached to the computer's serial ports.

E. These lines prompt the user to enter the full-scale span of the particular AR600 sensor being used. Each AR600 sensor model has a different full scale span, and this information must be passed to the library.

F. This line of the program 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. The third parameter to this function is the full scale span value that was entered. The `setCommOpen()` method must be the first function called when communicating with a sensor. If successful, the method's return code is CTI_SUCCESS, otherwise an error code is returned.

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

H. This line tells the CTI-AR600 library to acquire at least 10, and at most "`maxSamples`" samples from the sensor, and return the results in the array `rangeData`. See the description of the `getSamples()` function in the Reference Manual for more details.

I. 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_AR600.h`" header file. If an error does occur, the communications link to the sensor is closed, and the program exits.

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

K. This line displays the number of bytes that were skipped in the input data stream, in order to synchronize valid data samples, during the `getSamples()` call. See the Reference Manual for more information.

L. 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-AR600 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.*

Hello World In C

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

```
    /*  C-Language "Hello World" example for the
        Crandun Technologies CTI-AR600 software library. */

A   #include "CTI_AR600.h" /* required header */
    #include <stdio.h>
B   #define maxSamples 20

    int main()
    {
        char c;
C       long i, rc;
        float fullScaleSpan, rangeData[maxSamples];
        long numSkipped;
        long sensorHandle = -1;

D       printf("Crandun Technologies CTI-AR600 Library 'Hello World' example.\n");
        printf("Enter the Full Scale Span of the sensor you are using (e.g. 4.0): ");
        scanf("%f", &fullScaleSpan);

E       sensorHandle = getNewCTIAR600();
F       if (sensorHandle < 0) {
            printf("ERROR: getNewCTIAR600 returned error code %ld\n", sensorHandle);
            return -1;
        }

G       rc = setCommOpen(sensorHandle, "COM1", 9600, fullScaleSpan);
        if (rc != CTI_SUCCESS) {
            printf("ERROR: setCommOpen returned error: %ld\n", rc);
            return -1;
        }

        printf("Reading samples from sensor ...\n");
H       rc = getSamples(sensorHandle, rangeData, maxSamples, 10);
I       if (rc < 0) {
            printf("ERROR: getSamples returned error: %ld", rc);
            setCommClosed(sensorHandle);
            return -1;
        }

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

        numSkipped = getNumBytesSkipped(sensorHandle);
K       printf("%ld bytes of data were skipped.\n", numSkipped);

L       setCommClosed(sensorHandle);
M       setReleaseHandle(sensorHandle);

       printf("Please enter any character to exit: ");
        scanf("%c\n", &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-AR600 library. This header file must be included by any program using the CTI-AR600 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-AR600 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. These lines prompt the user to enter the full-scale span of the particular AR600 sensor being used. Each model of the AR600 sensor has a different full scale span, and this information must be passed into the CTI-AR600 library so that the sample data from the sensor can be properly calibrated.

E. This line calls the `getNewCTIAR600()` 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 AR600 sensor(s) that may be attached to the computer's serial ports.

F. This line checks the return code from the `getNewCTIAR600()` call. It is *essential* that every function's return code be checked for success or failure.

G. This line of the program uses the `setCommOpen()` function 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. The third parameter to this function is the full scale span value that was entered. The `setCommOpen()` function must be the first function called after obtaining a sensor handle. If successful, the function's return code is `CTI_SUCCESS`, otherwise an error code is returned.

H. This line tells the CTI-AR600 library to acquire at least 10, and at most "`maxSamples`" samples from the sensor, and return the results in the array `rangeData`.

I. 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_AR600.h`" header file. If an error does occur, the communications link to the sensor is closed, and the program exits.

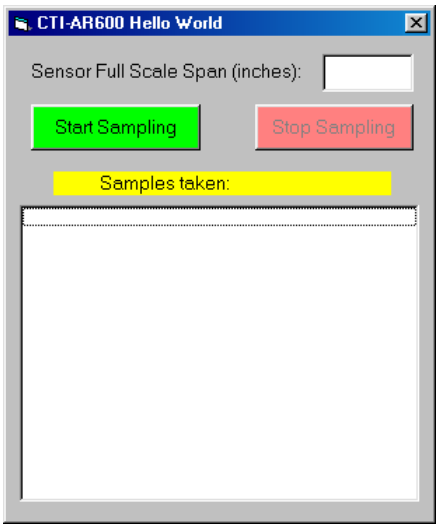
J. These lines print out the samples returned by the `getSamples()` function.

K. This line displays the number of bytes that were skipped in the input data stream, in order to synchronize valid data samples, during the `getSamples()` call. See the Reference Manual for more information.

L. 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-AR600 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.*

M. 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.

Hello World In Visual Basic



The Visual Basic "Hello World" program displays a window as shown at right.

The sensor's full scale span value must be entered in the text field at the top of the window.

When the "Start Sampling" button is pressed, the program opens the serial connection to the sensor, 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 serial port connection to the sensor.

The program's source code, with further explanations, is shown below. Note: for clarity, some error handling code has been omitted in the listing below. The full code is included in the installed sample program.

```
A   Private mySensor As CTI_AR600
B   Private bExitFlag As Boolean
    ------------------------------------------------------------------------
    Private Sub cmdStart_Click()
       Dim rc As Long
       Dim fullScaleSpan As Single

       On Error GoTo err

C      fullScaleSpan = CSng(txtSpan.Text)
D      Set mySensor = New CTI_AR600        'Create a new AR600 object

E      rc = mySensor.setCommOpen("COM1", 9600, fullScaleSpan)
F      If rc <> CTI_SUCCESS Then
          MsgBox "ERROR: setCommOpen returned " + Str(rc), vbCritical
          Exit Sub
       End If

       cmdStop.Enabled = True
       MsgBox "Successfully opened the serial port. Starting sampling.", vbInformation

G      Dim rangeData(100) As Single
       Dim totSamples As Long
       Dim i As Integer

H      bExitFlag = False

       totSamples = 0
I      Do
J         rc = mySensor.getSamples(rangeData, 1)
K         If rc < 0 Then
             MsgBox "ERROR: getSamples returned " + Str(rc), vbCritical
             bExitFlag = True
          Else
L            For i = 1 To rc
M               ListRange.AddItem (Str(rangeData(i)))
                totSamples = totSamples + 1
N               lblTotSamples.Caption = Str(totSamples)
O               DoEvents
             Next i
```

```
P          End If
        Loop While (bExitFlag = False)
Q
        rc = mySensor.setCommClosed()
        If rc <> CTI_SUCCESS Then
           MsgBox "ERROR: setCommClosed returned " + Str(rc), vbCritical
        Else
           MsgBox "Closed the serial connection.", vbInformation
        End If
     err:

     End Sub
```

```
R    Private Sub cmdStop_Click()
        bExitFlag = True
     End Sub
```

```
S    Private Sub Form_Unload(Cancel As Integer)
        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_AR600". 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 AR600 sensor(s) that may be attached to the computer's serial ports.

B. This declares a boolean flag that will be 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 code attempts to convert the value in the "txtSpan" text field to a number. If not successful (such as when a non-numeric value is entered in the text field), then the "On Error" is executed, and the method is exited. Otherwise, the following code is executed.

D. This line creates a new instance of a CTI_AR600 object, and assigns it to the mySensor variable.

E. 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. The third parameter to this function is the full scale span value that was entered. The `setCommOpen()` method must be the first function called when communicating with a sensor. If successful, the function's return code is CTI_SUCCESS, otherwise an error code is returned.

F. This line checks the return code from the `setCommOpen()`call. It is essential that every function's return code be checked for success or failure. A list of error codes is defined in the file "CTI_AR600_Defs.bas".

G. This line declares an array "`rangeData`" to hold the range samples returned from the software library. The "`totSamples`" variable on the next line records the total number of samples acquired from the sensor.

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

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

J. This line tells the CTI-AR600 library to acquire at most 100 samples (the size of the `rangeData` array), and at least 1 sample from the sensor, and return the results in the array `rangeData`. See the description of the `getSamples()` function 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_AR600_Defs.bas".

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

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

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

O. 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.

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

Q. 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-AR600 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.*

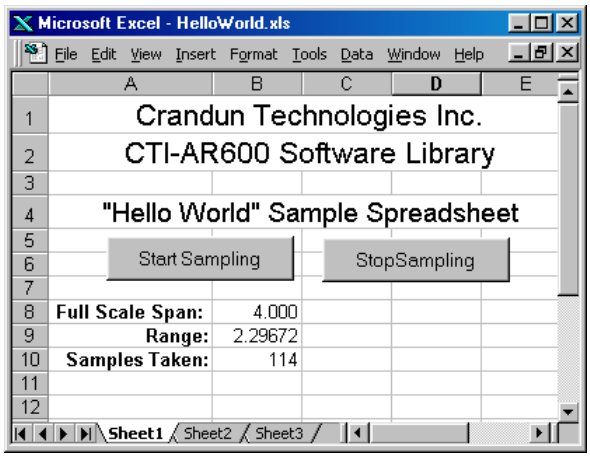
R. This line in the "Stop Sampling" button's "Click" method sets the exit flag to true.

S. 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

The Excel "Hello World" spreadsheet is shown at right. When the "Start Sampling" button is pressed, the program opens the serial connection to the sensor, and acquires range samples at the sensor's factory default rate. Each range sample is displayed in cell B9. The total number of data points taken is displayed in cell B10. Pressing the "Stop Sampling" button turns off the laser and closes the serial port connection to the sensor.

The spreadsheet's VBA (Visual Basic for Applications) source code, with further explanations, is shown below.




```
A   Private mySensor As CTI_AR600
B   Private bExitFlag As Boolean

C   Sub CommandStartSampling_Click()
D      Set mySensor = New CTI_AR600
       Dim rc As Long
       Dim fullScaleSpan As Single

E      fullScaleSpan = Val(Worksheets("sheet1").range("B8").Value)
       If (fullScaleSpan <= 0) Then
          MsgBox "ERROR: The sensor's full scale span must be positive", vbCritical
          Exit Sub
       End If

F      rc = mySensor.setCommOpen("COM1", 9600, fullScaleSpan)
       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

G      Dim rangeData(100) As Single
       Dim totSamples As Long
       Dim i As Integer

H      bExitFlag = False

       totSamples = 0
I      Do
J         rc = mySensor.getSamples(rangeData, 1)
K         If rc < 0 Then
             MsgBox "ERROR: getSamples returned " + Str(rc), vbCritical
             bExitFlag = True
          Else
L            For i = 1 To rc
M               Worksheets("Sheet1").range("B9").Value = rangeData(i)
                totSamples = totSamples + 1
N               Worksheets("Sheet1").range("B10").Value = totSamples
O               DoEvents
             Next i
          End If
```

```
P       Loop While (bExitFlag = False)

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


    End Sub
    ------------------------------------------------------------------------
    Sub CommandStopSampling_Click()
R      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, in the VBA module's "Declarations" section, declares a variable of type "CTI_AR600". Since it is in the "Declarations" section, the variable is accessible to all methods of the module. Note that at this point, the variable does not yet refer to any specific AR600 sensor(s) that may be attached to the computer's serial ports.

B. This line declares a boolean flag that will be 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_AR600 object, and assigns it to the mySensor variable.

E. When the "Start Sampling" button is clicked, this code attempts to convert the value in cell B8 to a number. If not successful (such as when a non-numeric value is entered), then an error message is displayed and the method is exited. Otherwise, the following code is executed.

F. 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. The third parameter to this function is the full scale span value that was entered. The `setCommOpen()` method must be the first function called when communicating with a sensor. If successful, the method's return code will be CTI_SUCCESS, otherwise an error code will be returned.

G. This line declares an array "`rangeData`" to hold the range samples returned from the software library. The "`totSamples`" variable declared in the next line records the total number of samples read from the sensor.

H. This line sets the exit flag "`bExitFlag`" to false. When the "Stop Sampling" button is pressed, it is set to true.

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

J. This line tells the CTI-AR600 library to acquire at most 100 samples (the size of the `rangeData` array), and at least 1 sample from the sensor, and return the results in the array `rangeData`. See the description of the `getSamples()` function 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_AR600.bas".

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

M. This line displays each range value in cell B9.

N. This line displays the total samples counter in cell B10.

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

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

Q. 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-AR600 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.*

R. This line in the "Stop Sampling" button's "Click" method sets the exit flag to true.

Reading Data

When acquiring data from a hardware device, such as the AR600 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-AR600 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 30).

3. The `setCallbackFunction()` function may be used to have the CTI-AR600 library call a user defined callback function when a desired number of samples is available. This method is demonstrated in the "Callback_CPP" and "Callback_C" sample programs (see pages 30 and 31).

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 5), the CTI-AR600 library continually collects samples from the sensor 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-AR600 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-AR600 Software Library Callback Example
    Copyright (c) 2001, 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_AR600.h" /* required header */
   #include <stdio.h>
   #include <windows.h>   /* needed for Sleep function */
```

```
   /* 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 float * pD1,
                  const long N1,
                  const float * pD2,
                  const long N2);

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

      printf("Crandun Technologies CTI-AR600 Library Callback example.\n");

      sensorHandle = -1;
E     sensorHandle = getNewCTIAR600();

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

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

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

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

      /* Tell library to call the callback when 5 samples are available
         Sensor is at factory default of 5 samples/sec
         so this should result in the callback being called 3 times
         during the 3 second period that the main thread is sleeping */
I     rc = setCallbackThreshold(sensorHandle, 5);

      printf("Main program is sleeping for 3 seconds...\n");
J     Sleep(3000);

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

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

L     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 */
M  long myCallback(const float * pD1,
                   const long N1,
                   const float * pD2,
                   const long N2)
   {
      long i,j;

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

```
N       for (i = 0, j = 0; i < N1; i++, j++, pD1++) {
           printf("Range %d is %8.3f inches\n", j, *pD1);
           fprintf(my_DataFile, "%8.3f\n", *pD1);
        }

        /* read the samples from the second data pointer, if any */
O       for (i = 0; i < N2; i++, j++, pD2++) {
           printf("Range %d is %8.3f inches\n", j, *pD2);
           fprintf(my_DataFile, "%8.3f\n", *pD2);
        }

        /* return non-zero to tell lib to remove samples from its buffer */
P       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-AR600 library. This header file must be included by any program using the CTI-AR600 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 `getNewCTIAR600()` 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 AR600 sensor(s) that may be attached to the computer's serial ports.

F. 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.

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

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

I. This line sets the callback threshold at 5 samples. When the library has 5 samples in its internal buffer, it will call the callback function. The sensor is at its factory default of 5 samples per second, so a 5 sample callback threshold should result in the callback function being called 3 times during the 3 second period that the main thread will sleep.

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

K. 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.

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

M. 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.

N. This loop reads range samples from the first data buffer, displays each sample, and writes it to the data file.

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

P. 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 sensor 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 and Transformation

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 AR600 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.

Furthermore, it is common to transform the range samples received by adding offset values or scaling the results. To facilitate these kinds of operations, the CTI-AR600 library permits filtering the raw samples from the sensor to discard samples outside of a specific range of interest, and adding any offset or scale factor to the samples.

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.

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 and transformation functionality of the CTI-AR600 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 work within the application program.

These functions are explained in detail in the section "Data Filtering and Transformation Functions" starting on page 44 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.

Using the Auxiliary Communications Port

The CTI-AR600 library provides the ability to easily integrate data from other sensors or external data sources with range data received from the AR600 sensor. Typical measurement situations where this is required might include:

- The AR600 is used to measure the height of material on a conveyor belt, while a RPM sensor measures the speed of the conveyor.
- The AR600 sensor is mounted on a jackscrew driven carriage, and scans the surface of a steel plate for flatness, while an encoder on the jackscrew determines the carriage position from a known starting point.
- The AR600 sensor measures deflection of a surface, while an analog to digital converter measures the force applied to the surface.

In these circumstances, it is desirable to collect range data from the AR600 simultaneously with data from the other sensors, so that the two measurements can be related to one another. The CTI-AR600 library makes it

easy to collect measurement data from other sensors attached to another serial port on the computer. These functions are detailed in the Reference Manual section "Auxiliary Communications Port Functions".

The library's auxiliary port functions provide facilities for reading data from a serial port at the same time as data is read from the AR600 sensor. The CTI-AR600 library takes measurements from each sensor simultaneously, so that the data can easily be correlated.

Typical Usage

To use a auxiliary sensor with the CTI-AR600 library, the sensor must have serial interface (RS232, RS422, RS485, etc.), and the computer must have an available serial interface that matches that of the sensor.

Broadly speaking, using a sensor involves three operations:

(1) Initialization. Before use, the sensor may need to be 'turned on', initialized or otherwise set to some known state.

(2) Sampling. During use, many (although not all) sensors need to be sent a 'sample' command, to tell the sensor to output data back to the receiving computer.

(3) Shutdown. After use, the sensor may need to be sent a shutdown command to turn it off, or otherwise set it to a known state.

The CTI-AR600 library provides for these operations as follows:

SENSOR INITIALIZATION

If the sensor needs to be sent a particular command prior to being used, then the CTI-AR600 library's `setAuxCommOpenCmd()` function should be used. Immediately after opening the auxiliary serial port, the CTI-AR600 library will send the command specified by this function to the sensor attached to the port.

SAMPLING

Generally speaking, sensors fall into two broad categories:

(1) The first type of sensor continuously outputs data at regular (or irregular) time intervals, without interaction. For example, an encoder may continuously output angular measurements every 5 seconds, or a temperature sensor may output a new measurement every time the temperature changes.

(2) The second type of sensor has to be 'triggered', or commanded, before it outputs data. For example, an analog-to-digital converter may output a new sample only when it receives a command to do so.

If a type (1) sensor is connected to the auxiliary communications port, then, once the `setAuxCommOpen()` function is called, the CTI-AR600 library will read any data that the sensor is outputting at the same time as the library reads range data from the AR600 laser.

If a type (2) sensor is used, then the `setAuxCommSampleCmd()` function should be used to specify the command that is to be sent to the sensor to trigger the taking of a 'sample'. The command will be sent at the frequency set with the `setAuxCmdFreq()` function (see page 37 of the Reference Manual).

The frequency is expressed in number of AR600 samples. For example, a frequency of 1 (one), tells the library to send the sampling command to the auxiliary communications port every time the library receives a sample from the AR600 sensor. A frequency of 5 tells the library to send the sampling command for every fifth sample received from the AR600 sensor.

With both type (1) and type (2) sensors, the library will read range data from the AR600 laser and data from the auxiliary sensor simultaneously, so that the two measurements are easily correlated.

How the CTI-AR600 library determines what a constitutes a 'sample' from the equipment is determined by the `setAuxCommSampleDelim()`and `setAuxCommSampleLen()` functions.

If individual data samples from the sensor are delimited by a specific character, then that character should be specified using the `setAuxCommSampleDelim()` function. The default delimiter character is Carriage Return (ASCII 13).

On the other hand, if individual data samples from the sensor are *not* delimited by a specific character, but instead are always a fixed length, then the `setAuxCommSampleLen()` function should be used to tell the library how many bytes constitute a 'sample'. The library will read this number of bytes from the auxiliary communications port, and return that data along with AR600 range data to the application program.

SHUTDOWN

If the sensor attached to the auxiliary communications port needs to be sent a particular command prior to shutdown, then the CTI-AR600 library's `setAuxCommCloseCmd()` function should be used. Before the CTI-AR600 library closes the auxiliary serial port, it will send the command specified by this function to the port.

Recommended Usage

As documented in the Reference Manual, some of the functions of the CTI-AR600 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-AR600 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 `getNewCTIAR600()` (see Reference Manual page 1) to obtain a sensor instance handle.

2. Open the serial port using the `setCommOpen()` function (see page 2 of the Reference Manual).

3. Set the sensor's sampling interval using `setSampleInterval()` or `setSamplesPerSec()` (Reference Manual pages 25 and 26).

4. If desired, set the library's buffer size using the setBufferSize`()` function. (Reference Manual page 7).

5. If desired, turn off the sensor's background light elimination feature, using `setBGLightElimOff()` (Reference manual, page 16).

6. If callbacks will be used, set the callback function and callback threshold using `setCallbackFunction()` and `setCallbackThreshold()` (Reference Manual page 8 and 11).

7. Set the range offset and range scale factors, if any, using `setRangeOffset()` and `setRangeScaleFactor()` (Reference Manual pages 46 and 48).

8. Set any filtering and transformation parameters using `setMaxRange()`, `setRangeOffset()` and `setRangeScaleFactor()`. (See the "Data Filtering and Transformation" section on page 23 of the Programmer's Guide and the "Data Filtering and Transformation Functions" section, starting on page 44 of the Reference Manual.)

9. Set continuous sampling on using the `setContinuousSerialOn()` function (Reference Manual page 27). (By default continuous serial samples will be on, so this step may be superfluous.)

10. Clear the library's buffer using `setClearBuffer()` (Reference Manual, page 8).

11. Retrieve samples using `getSamples()` (Reference Manual page 29) 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-AR600 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, and the `TwoSensors` spreadsheet.

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.

Controlling Multiple Sensors

Use of the library with multiple sensors is shown in the `TwoSensors_C` and `TwoSensors_CPP` programs and the `TwoSensors` spreadsheet.

Dealing with Spurious Samples

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

Integrating Data from Other Instrumentation

The `AuxComm_C` and `AuxComm_CPP` C and C++ sample programs, and the `AuxComm` Visual Basic program demonstrate simultaneously acquiring data from the AR600 laser, and an additional sensor attached to a second serial port.

Do's and Don'ts

Below are some "do's and don'ts" recommendations for use of the CTI-AR600 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 `setCommOpen()` function as the first function in any application program using the library. (C language programmers must call the `getNewCTIAR600()` function to get a valid sensor handle first.) This function performs all the initialization required to start using the library, and must be the first function called by any application.

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. Ensure that the `fullScaleSpan` parameter of the `setCommOpen()` function matches the actual full scale span of the AR600 sensor model being used. An incorrect setting will cause the samples returned by the library to be incorrectly calibrated.

4. 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.*

5. 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.

6. Ensure that the serial communications link baud rate is set sufficiently high to accommodate the currently set sample rate without losing data. In the binary communications mode used by the CTI library, the sensor transmits 3 bytes per sample, with each byte consisting of 8 data bits and 1 stop bit (see the Acuity hardware documentation). At 9600 baud, this gives a maximum sample rate of 355 samples/second. ([9600 bits per second/9 bits per byte]/3 bytes per sample is approximately 355 samples/sec.) Therefore, any rate higher than 355 samples/sec will require a faster baud rate (see the `setBaudRate()` function).

7. Do use the data filtering and transformation functionality provided by the library, rather than perform this work in the application program, as use of these functions is more efficient and less error-prone than writing equivalent functionality within the application.

8. 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.

9. 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()` function. 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. Applicable to C language programmers only. Do not repeatedly call `getNewCTIAR600()` without making corresponding calls to `setReleaseHandle()`. If the maximum number of handles supported by the library (currently 100) have already been allocated by calling `getNewCTIAR600()` without corresponding calls to `setReleaseHandle()`, then future calls to `getNewCTIAR600()` 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.

3. 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 the compiled version of each program assumes that the sensor is attached to the computer's COM1 serial port, and that the sensor's serial communications rate is 9600 baud (the factory default). If the sensor is configured differently, the compiled version of the program will not run successfully.

C++ Language Examples

Hello_World_CPP

The C++ language "Hello World" program documented on page 8 is installed in the `Samples\CPP` 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 `Hello_World_CPP.dsp` may be opened directly in Microsoft Visual C++.

AuxComm_CPP

This program is installed in the `Samples\CPP` 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 `AuxComm_CPP.dsp` may be opened directly in Microsoft Visual C++. This program demonstrates simultaneously reading data from the AR600 sensor and a sensor attached to the auxiliary communications port.

BaudRate_CPP

This program is installed in the `Samples\CPP` 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 `BaudRate_CPP.dsp` may be opened directly in Microsoft Visual C++. This program demonstrates querying the sensor's firmware version and the library's name. It also demonstrates changing the baud rate of the serial communications link, and the sensor's sample rate.

Callback_CPP

This program is installed in the `Samples\CPP` 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 `CallBack_CPP.dsp` may be opened directly in Microsoft Visual C++. This program 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. 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_CPP.dsp` may be opened directly in Microsoft Visual C++. This program demonstrates using the library's filtering functions to discard samples outside of a specified range.

TwoSensors_CPP

This program is installed in the `Samples\CPP` 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 `TwoSensors_CPP.dsp` may be opened directly in Microsoft Visual C++. This program demonstrates acquiring and displaying range data from two sensors simultaneously.

C Language Examples

Hello_World_C

The C language "Hello World" program documented on page 10 is installed in the `Samples\C` 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 `Hello_World_C.dsp` may be opened directly in Microsoft Visual C++.

AuxComm_C

This program is installed in the `Samples\C` 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 `AuxComm_C.dsp` may be opened directly in Microsoft Visual C++. This program demonstrates simultaneously reading data from the AR600 sensor and a sensor attached to the auxiliary communications port.

BaudRate_C

This program is installed in the `Samples\C` 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 `BaudRate_C.dsp` may be opened directly in Microsoft Visual C++. This program demonstrates querying the sensor's firmware version and the library's name. It also demonstrates changing the baud rate of the serial communications link, and the sensor's sample rate.

Callback_C

This program is installed in the `Samples\C` 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 `CallBack_C.dsp` may be opened directly in Microsoft Visual C++. This program 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. 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_C.dsp` may be opened directly in Microsoft Visual C++. This program demonstrates using the library's filtering functions to discard samples outside of a specified range.

TwoSensors_C

This program is installed in the `Samples\C` 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 `TwoSensors_C.dsp` may be opened directly in Microsoft Visual C++. This program demonstrates acquiring and displaying range data from two sensors simultaneously.

Visual Basic Language Examples

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.

AuxComm

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 `AuxComm.vbp` may be opened directly in Microsoft Visual Basic. This program demonstrates simultaneously reading data from the AR600 sensor and a sensor attached to the auxiliary communications port.

BaudRate

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 `BaudRate.vbp` may be opened directly in Microsoft Visual Basic. This program demonstrates querying the sensor's firmware version and the library's name. It also demonstrates changing the baud rate of the serial communications link, and the sensor's sample rate.

Filtering

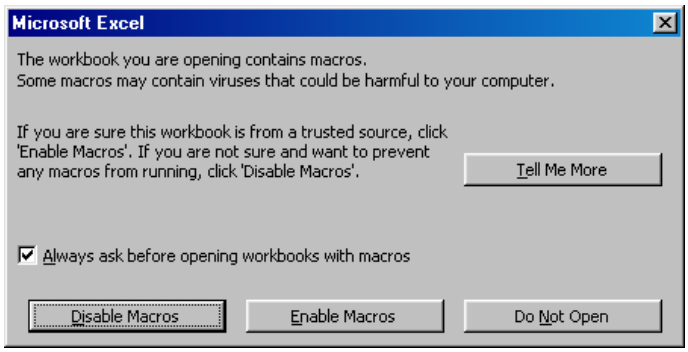
This program is installed in the `Samples\VB` subdirectory. The program may be run from a Command Prompt window, or by double-clicking the executable file from the Windows Explorer. The program's project file `Filtering.vbp` may be opened directly in Microsoft Visual Basic. This program demonstrates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

TwoSensors

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 `TwoSensors.vbp` may be opened directly in Microsoft Visual Basic. This program demonstrates acquiring and displaying range data from two sensors simultaneously.

Microsoft Excel Examples

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 15 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.

TwoSensors

This spreadsheet is installed in the `Samples\Excel` subdirectory, and may be opened by clicking on the file "`TwoSensors.xls`" from a Windows Explorer window, or by opening the file from Excel's File Open menu.

This spreadsheet demonstrates acquiring and displaying range data from two sensors simultaneously.

Building a Program

Building a C or C++ Program

This section gives a brief overview of the basic steps involved in building a program that uses the CTI-AR600 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-AR600 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-AR600 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_AR600.h` within any source file that calls a library function. `CTI_AR600.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-AR600`")

   You should not modify `CTI_AR600.h` in any way.

3. When linking the program, include the file `CTI_AR600.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_AR600.lib` contains definitions of all the functions exported by `CTI_AR600.DLL` and must be linked with your project in order to call the functions in `CTI_AR600.DLL.`

4. When running the program, the file `CTI_AR600.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-AR600 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-AR600 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-AR600 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_AR600.cls` in the project as a Class Module. In Visual Basic 6.0, this is done using the "Project Add Class Module" menu option. The file `CTI_AR600.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-AR600`")

   If you are using a different version of Visual Basic, the procedure for inserting Class Modules may vary. Please consult the appropriate Microsoft documentation.

   You should not modify `CTI_AR600.cls` in any way.

3. Include the file CTI_AR600_Defs.bas in the project as a Code Module. In Visual Basic 6.0, this is done using the "Project Add Module" menu option. The file CTI_AR600_Defs.bas may be found in the Source\VB directory under the Library's installation directory.

   If you are using a different version of Visual Basic, the procedure may vary. Please consult the appropriate Microsoft documentation.

   You should not modify CTI_AR600_Defs.bas in any way.

4. When running the program, either from within the Visual Basic development environment or as a compiled Visual Basic executable, the file CTI_AR600.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-AR600 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-AR600 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-AR600 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_AR600.cls in the VBA project as a Class Module. Using Microsoft Excel 97, this is done using the "File Import File" menu option. This will open a dialog box to select the file to insert. Select the file CTI_AR600.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-AR600").

   If you are using a different version of Excel, the procedure for inserting Class Modules may vary. Please consult the appropriate Microsoft documentation.

   You should not modify CTI_AR600.cls in any way.

3. Using the same procedure as the previous step, include the file CTI_AR600_Defs.bas in the project as a standard (Code) Module. CTI_AR600_Defs.bas may be found in the Source\VB directory under the Library's installation directory.

   You should not modify CTI_AR600_Defs.bas in any way.

4. When running the program, either from within the VBA development environment, or from within Excel, the file CTI_AR600.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-AR600 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-AR600 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-AR600 library. ***No other files or components of the CTI-AR600 Library package may be redistributed.***

If you distribute an application or library that incorporates the CTI-AR600 Software Library, your application must add significant value to the API provided by the CTI-AR600 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.

Redistributable Files

Directory	File s	Description
`Windows\System`	`CTI_AR600.DLL`	This DLL implements the majority of the functionality of the CTI-AR600 library. It is 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\VB`	`CTI_AR600.cls`	This file defines the Visual Basic CTI_AR600 class object. It must be included as a Class Module in any Visual Basic application using the library.
`instdir\Source\VB`	`CTI_AR600_defs.bas`	Defines the status codes and data structures returned by the CTI-AR600 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_AR600.cls`	This file defines the Visual Basic for Applications CTI_AR600 class object. It must be included as a Class Module in any Visual Basic for Applications (VBA) program using the library.

Part II – Reference Manual

This Reference Manual describes each of the functions provided by the CTI-AR600 library. Logically related functions are grouped together, and each function's prototype, purpose, parameters, and return values is provided.

Function Parameters

In most cases, the parameters required by each function are identical, regardless of whether C, C++ or Visual Basic is being used. In the few cases where parameters differ, the notation [C] denotes the parameters required from C, [C++] denotes those required from C++, and [VB] denotes those required from Visual Basic. Unless otherwise noted, all comments regarding Visual Basic also apply to Visual Basic for Applications programs.

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 status codes are defined in the files `CTI_ErrCodes.h` (for C and C++ programs), and in the file `CTI_AR600_Defs.bas` (for Visual Basic and VBA 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, configure the library's internal buffer size, and control the library's callback functionality.

getNewCTIAR600

Return a new handle to a particular sensor instance.

```
C:     long getNewCTIAR600()

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 value 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 2). 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-AR600 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 `getNewCTIAR600()`.

RETURN VALUES

CTI_BAD_PARAM if `sensorHandle` is not a valid handle previously obtained by calling `getNewCTIAR600()`, or if the handle has already been released. CTI_SUCCESS otherwise.

COMMENTS

This function calls `setCommClosed()` for the specified sensor handle (see page 4 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 `getNewCTIAR600()` are made without corresponding calls to `setReleaseHandle()`, then the library's pool of sensor handles will eventually be exhausted, and future calls to `getNewCTIAR600()` will fail.

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,
                        const float fullScaleSpan)

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

VB:    mySensor.setCommOpen(ByVal serialPortName As String,
                            ByVal initialBaudRate As Long,
                            ByVal fullScaleSpan as Single) 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 Acuity hardware documentation for a list of valid baud rates.
`fullScaleSpan` – The full scale span of the particular AR600 sensor being used. Must be greater than zero.

RETURN VALUES

CTI_SUCCESS if successful.
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, if the baud rate is invalid for this sensor, or if the full scale span value provided is zero or negative.
CTI_COMM_OPEN_ERROR if the specified serial port could not be opened successfully.

COMMENTS

Once a sensor instance has been allocated (by obtaining a handle from the getNewCTIAR600 function in C, or by instantiating an object instance in C++ or Visual Basic) the setCommOpen function *must* be the next function called.

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, 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. 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.

The full scale span value provided must match that of the actual AR600 sensor being used, otherwise all samples returned by the library will be incorrectly calibrated.

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

getIsCommOpen

Determine if the communications link to the sensor is open.

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

C++:   long mySensor.getIsCommOpen()

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

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the communications link to the sensor 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:    mySensor.setCommClosed() As Long
```

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 library's error indicator, and frees any internal resources used by the library.

This function must be called by an application program to cleanly terminate use of the CTI-AR600 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.

This function does *not* close the auxiliary communications port, if open. Use the `setAuxCommClosed()` function (see page 34) to close the auxiliary communications port.

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 Acuity hardware documentation for a list of valid baud rates.

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

It is the application programmer's responsibility to ensure that the serial communications link baud rate is set sufficiently high to accommodate the currently set sample rate without losing data. In the binary communications mode used by the CTI library, the sensor transmits 3 bytes per sample, with each byte consisting of 8 data bits and 1 stop bit (see the Acuity hardware documentation). At 9600 baud, this gives a maximum sample rate of 355 samples/second. ([9600 bits per second/9 bits per byte]/3 bytes per sample is approximately 355 samples per second.) Therefore, any data rate higher than 355 samples/second will require a faster baud rate to avoid loss of data.

This function 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 baud rate for the computer's serial port.

COMMENTS

None.

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 sensor 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.

```
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 sensor. Use `setContinuousSerialOff()` 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 a non-zero value 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 `setContinuousSerialOn()` function on page 27), 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 sensor. 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.

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 setCallbackFunction(const long sensorHandle,
                    long cbFn(const float * pD1,
                              const long N1,
                              const float * pD2,
                              const long N2))

C++:   long mySensor.setCallbackFunction(
                    long cbFn(const float * pD1,
                              const long N1,
                              const float * 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 float * pD1, const long N1, const float * 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.

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-AR600 library itself is fully thread-safe, and any library function may be called from any thread of execution without user-supplied synchronization mechanisms.

This function should be used when the auxiliary communications port is *not* being used. When the auxiliary port is being used, `setExtCallbackFunction()` (see page 11) should be used instead.

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.

setExtCallbackFunction

Sets the function to be called when the number of samples in the library's internal data buffer reaches the callback threshold, and the auxiliary communications port is being used.

```
C:     long setCallbackFunction(const long sensorHandle,
                      long cbFn(const AR600_DATA_PT * pD1,
                                const long N1,
                                const AR600_DATA_PT * pD2,
                                const long N2))

C++:   long mySensor.setCallbackFunction(
                      long cbFn(const AR600_DATA_PT * pD1,
                                const long N1,
                                const AR600_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 AR600_DATA_PT * pD1, const long N1,
          const AR600_DATA_PT * pD2, const long N2)
```

RETURN VALUES

CTI_BAD_PARAM if the function pointer is null. CTI_SUCCESS otherwise.

COMMENTS

See the comments for the setCallbackFunction on page 10.

This function must be used when the auxiliary communications port is being used (see the section "Auxiliary Communications Port Functions" for more details).

This function behaves identically to the `setCallbackFunction()`, except that samples returned to the application program are of type AR600_DATA_PT (see definition on page 30).

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 `setContinuousSerialOn()` function on page 27), 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.

getFullScaleSpan

Returns the sensor's full scale span setting.

```
C:     float getFullScaleSpan(const long sensorHandle)

C++:   float mySensor.getFullScaleSpan()

VB:    mySensor.getFullScaleSpan() As Single
```

PARAMETERS

None.

RETURN VALUES

CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.

Otherwise, returns the sensor's full scale span setting, as set with the `setCommOpen()` function.

COMMENTS

The full scale span setting, as specified in the call to the `setCommOpen()` function *must* match that of the actual AR600 sensor being used, otherwise all samples returned by the library will be incorrect.

Sensor Configuration Functions

setFactoryDefaults

Resets all sensor settings to their factory default values.

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

C++:   long mySensor.setFactoryDefaults()

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

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if successful.
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 Acuity 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.

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.

setAnalogOutputOn

Turns on the current loop (analog) output.

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

C++:   long mySensor.setAnalogOutputOn()

VB:    mySensor.setAnalogOutputOn() 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 will have no effect if the sensor does not have the current loop option installed.

This function is equivalent to the sensor's 'X1' command.

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 will have no effect if the sensor does not have the current loop option installed.

This function is equivalent to the sensor's 'X2' 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

The result of this function is meaningless if the sensor does not have the current loop option installed.

setBGLightElimOn

Turn on the sensor's background light elimination feature.

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

C++:   long mySensor.setBGLightElimOn()

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

PARAMETERS

None.

RETURN VALUES

CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if successful. 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 turns on the sensor's background light elimination feature, which allows the sensor to eliminate the effects of background light sources on the readings taken. By default, this feature is on. See the Acuity hardware documentation for more details.

This function does *not* affect any samples currently in the library's internal buffer.

This function is equivalent to the sensor's 'L1' command.

setBGLightElimOff

Turn off the sensor's background light elimination feature.

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

C++:   long mySensor.setBGLightElimOff()

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

PARAMETERS

None.

RETURN VALUES

CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_SUCCESS if successful. 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 turns off the sensor's background light elimination feature. See the Acuity hardware documentation for details on this feature.

This function is equivalent to the sensor's 'L2' command.

getIsBGLightElimOn

Determine if the sensor's background light elimination feature is on.

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

C++:   long mySensor.getIsBGLightElimOff()

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

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the sensor's background light elimination feature is on, and 0 (zero) otherwise.

COMMENTS

None.

setHWFlowControlOn

Turn on serial port hardware flow control (using the DTR line) for control of data sent from the sensor.

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

C++:   long mySensor.setHWFlowControlOn()

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

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if successful.
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 sets the sensor and computer serial port to use the DTR line for flow control of data sent from the sensor to the computer. For this setting to be effective, the serial cable used to connect the sensor to the computer must implement the DTR line. This setting has no effect for sensors using the RS-422 interface, since the wire pairs for DTR are not included. This function is equivalent to the sensor's 'T1' command.

setHWFlowControlOff

Turn off use of the DTR line for flow control of data sent from the sensor over the serial port.

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

C++:   long mySensor.setHWFlowControlOff()

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

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if successful.
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 sets the sensor and computer serial port to NOT use the DTR line for flow control of data sent from the sensor to the computer. This function is equivalent to the sensor's 'T2' command.

getIsHWFlowControlOn

Determine if hardware flow control (using the DTR line) is on.

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

C++:   long mySensor.getIsHWFlowControlOn()

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

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if hardware flow control is on, and 0 (zero) otherwise.

COMMENTS

None.

setSamplePriorityQuality

Set the sensor's sample priority mode to "Quality".

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

C++:   long mySensor.setSamplePriorityQuality()

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

PARAMETERS

None.

RETURN VALUES

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

COMMENTS

Sets the sensor's sample priority to "Quality". In this mode, the sensor adjusts the time to take a reading to maintain a high signal quality. See the Acuity hardware documentation for more details.

This is equivalent to the sensor's 'P1' command.

setSamplePriorityRate

Set the sensor's sample priority mode to "Rate".

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

C++:   long mySensor.setSamplePriorityRate()

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

PARAMETERS

None.

RETURN VALUES

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

COMMENTS

Sets the sensor's sample priority to "Rate" (this is the sensor's default setting). In this mode, the sensor sends samples at the rate specified, optimizing and averaging internal samples to the extent that the sample time allows. Please see the Acuity hardware documentation for more details.

This is equivalent to the sensor's 'P2' command.

getIsSamplePriorityQuality

Determine the sensor's current sample priority mode

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

C++:   long mySensor.getIsSamplePriorityQuality()

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

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the sensor's current sample priority mode is "Quality", and 0 (zero) if it is "Rate".

COMMENTS

None.

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. This is interpreted as the absolute distance from the start of the sensor's physical measurement range. Please see the Acuity hardware documentation for more details. Valid range is 0 <= `distance <=` 50,000.

RETURN VALUES

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

COMMENTS

If the sensor's zero point is subsequently changed with `setZeroPt()`, the span setting is also changed. Please refer to the Acuity hardware documentation for further information on the effect of this command.

This function is equivalent to the sensor's 'U' 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.

setZeroPt

Sets the zero point for both the serial and analog 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 the sensor physical measurement units. Valid range is 0 <= `distance` <= 50,000. Please see the Acuity hardware documentation for more details.

RETURN VALUES

CTI_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_BAD_PARAM if the distance parameter is out of range.
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 'Z' command. Please refer to the Acuity hardware documentation for further information on the effect of this command.

getZeroPt

Returns the zero point.

```
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.

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.
CTI_FAILURE if any error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This is equivalent to the sensor's 'H1' 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.
CTI_FAILURE if any error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This is equivalent to the sensor's 'H2' 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 800 <= `interval` <= 50,000.

RETURN VALUES

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

COMMENTS

The AR600 sensor's sampling interval may be set in increments of 100 microseconds. If the `interval` parameter is not an even multiple of 100 microseconds, the CTI library will round the parameter supplied to the nearest 100 microsecond interval before sending the command to the sensor hardware. If the function succeeds, the actual sampling interval (corresponding to the rounded value) set is returned.

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

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 1 <= `samplesPerSec` <= 1,250.

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 other error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

The AR600 sensor's sampling interval may be set in increments of 100 microseconds. Due to the discrete sampling intervals possible, the actual number of samples per second may differ from the value supplied as a parameter. Please see the `setSampleInterval()` function on page 25 for details on how the CTI-AR600 library handles sample intervals that are not an even multiple of the 100 microsecond increments.

A fractional number of samples per second cannot be set using this function. If less than one sample per second is desired, the `setSampleInterval()` function (see page 25) should be used.

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.

setContinuousSerialOn

Continuously output samples over the serial link at the currently configured sample interval.

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

C++:   long mySensor.setContinuousSerialOn()

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

PARAMETERS

None.

RETURN VALUES

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

COMMENTS

This function tells the sensor to begin continuously outputting samples over the serial link at the currently configured sample interval. These samples will be captured by the CTI library and stored in the library's internal buffer for retrieval by the application program.

By default, continuous serial sampling is on.

This is equivalent to the sensor's 'A1' or 'A2' command, depending on the currently selected output units (English or metric respectively).

setContinuousSerialOff

Stop continuously outputting samples over the serial link.

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

C++:   long mySensor.setContinuousSerialOff()

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

PARAMETERS

None.

RETURN VALUES

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

COMMENTS

This function tells the sensor to stop continuously outputting samples over the serial link at the currently configured sample interval. Any samples already in the CTI library's internal buffer are not affected, and remain available for retrieval by the application program.

This is equivalent to the sensor's 'A3' command.

getIsContinuousSerialOn

Determine if the sensor is configured to continuously output samples over the serial link.

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

C++:   long mySensor.getIsContinuousSerialOn()

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

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the sensor is currently configured to continuously output samples 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 `setContinuousSerialOn()` function on page 27), 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 sensor and adding them to the library's buffer.

getSamples

Returns range sample data from the sensor.

```
C:     long getSamples(const long sensorHandle, float * databuffer,
                  const long bufSizeInSamples,
                  const long minNumSamples,
                  const long msWait)

C++:   long mySensor.getSamples(float * databuffer,
                  const long bufSizeInSamples,
                  const long minNumSamples,
                  const long msWait)

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

PARAMETERS

`databuffer` – the data buffer to hold the range samples returned. Each sample is a floating point number in the currently selected output units (inches if English, millimeters if metric.) 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. 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_COMM_NOT_OPEN if `setCommOpen()` has not been called.
CTI_ILLEGAL_CALL if `setAuxCommOpen()` has been called.
CTI_BAD_PARAM if any parameter is out of range.
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.

COMMENTS

This function removes samples from the library's internal buffer returning them to the application program. Unless an error occurs, or the timeout period expires, the function always returns at least `minNumSamples` samples. If fewer than `minNumSamples` samples are available in the library's buffer when the function is called, the library will enter an efficient wait state, until enough samples are returned from the sensor. The library will wait for a maximum of `msWait` milliseconds. If `minNumSamples` are not available during this time, CTI_WAIT_TIMEOUT is returned, ensuring 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 50% higher than the minimum, to account for timing variances in the data collection. The programmer must also ensure that the serial port baud rate is sufficient to accommodate the sample rate specified.

If the library is reading data from a sensor attached to the auxiliary communications port, as well as the AR600 sensor, then the `getExtSamples()` function must be used instead of this function.

getExtSamples

Returns range data from the sensor, along with data from the auxiliary communications port.

```
C:     long getExtSamples(const long sensorHandle, AR600_DATA_PT * databuffer,
                  const long bufSizeInSamples, const long minNumSamples,
                  const long msWait)

C++:   long mySensor.getSamples(AR600_DATA_PT * databuffer,
                  const long bufSizeInSamples, const long minNumSamples,
                  const long msWait)

VB:    mySensor.getSamples(dataBuffer() as AR600_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 structure of type AR600_DATA_PT (see definition below). 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_ILLEGAL_CALL if `setAuxCommOpen()` has not been called.
CTI_BAD_PARAM if any parameter is out of range.
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.

COMMENTS

See also the comments for the `getSamples()` function on page 29, and the section "Using the Auxiliary Communications Port" in the Programmer's Guide.

This function should be called when the library is reading samples from both the AR600 laser, and a sensor attached to the auxiliary communications port (i.e. after `setAuxCommOpen()` has been called).

Each sample returned is a structure of type AR600_DATA_PT as defined below:

```
typedef struct {       /* C and C++ AR600_DATA_PT structure */
   float range;        /* calibrated range value */
   char auxData[80];   /* data read from auxiliary comm port */
} AR600_DATA_PT;

Public Type AR600_DATA_PT  '  VB data type for returning sensor data
   range As Single         '  calibrated range value from AR600
   auxData As Byte * 80    '  data from auxiliary comm port
End Type
```

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_WAIT_TIMEOUT if the raw sample returned from the sensor is discarded by the current filtering criteria.

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

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

getNumBytesSkipped

Returns the number of bytes skipped by the library while processing the serial data stream.

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

C++:   long mySensor.getNumBytesSkipped()

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

PARAMETERS

None.

RETURN VALUES

The number of bytes skipped in the serial data stream since the last time the `setResetBytesSkipped()` or `setFactoryDefaults()` function was called.

COMMENTS

This function returns the number of samples skipped by the library while processing incoming serial data from the sensor. As the library processes the serial data stream, it discards any samples that do not have framing bytes in the correct sequence. An excessively high number of skipped bytes may indicate a poor serial connection between the computer and the sensor.

setResetBytesSkipped

Resets the bytes skipped count.

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

C++:   long mySensor.setResetBytesSkipped()

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

PARAMETERS

None.

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

This function resets the library's internal bytes skipped counter, as returned by the `getNumBytesSkipped()` function. The `setFactoryDefaults()` function also resets this counter.

Auxiliary Communications Port Functions

setAuxCommOpen

Opens the auxiliary communications (serial) port.

```
C:     long setAuxCommOpen(const long sensorIndex,
              const char * serialPortName, const long initialBaudRate)

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

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

PARAMETERS

`serialPortName` – The name of the serial port to open. (e.g. "COM4")
`initialBaudRate` – The baud rate at which to open the port. This must match the baud rate of the equipment attached to the serial port.

RETURN VALUES

CTI_SUCCESS if successful.
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.
CTI_COMM_OPEN_ERROR if the specified serial port could not be opened successfully.

COMMENTS

This function opens the specified serial port, and tells the CTI-AR600 library to read data from this port, along with data from the AR600 sensor. See the section "Using the Auxiliary Communications Port" in the Programmer's Guide for more details.

If a command has been defined using `setAuxCommOpenCmd()`, (see page 34), then this command will be sent to the port immediately after it is opened.

Do not confuse this function with `setCommOpen()`, which establishes communications to the AR600 sensor. This function has no impact on communications with the AR600 sensor.

getIsAuxCommOpen

Determine if the auxiliary communications link is open.

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

C++:   long mySensor.getIsAuxCommOpen()

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

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if the auxiliary communications port is currently open, and 0 (zero) otherwise.

COMMENTS

None.

setAuxCommClosed

Closes the auxiliary communications (serial) port.

```
C:     long setAuxCommClosed(const long sensorIndex)

C++:   long mySensor.setAuxCommClosed()

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

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if successful.
If any error occurs, `getErrorMessage()` may be called to get an extended error message.

COMMENTS

This function closes the auxiliary communications port opened with setAuxCommOpen() and tells the CTI-AR600 library to no longer read data from the port.

If a command has been defined using `setAuxCommCloseCmd()`, (see page 35), then that command will be sent to the port immediately before it is actually closed.

Do not confuse this function with `setCommClosed()`, which terminates communications to the AR600 sensor. This function has no impact on communications with the AR600 sensor.

setAuxCommOpenCmd

Sets the command to send when the auxiliary communications port is opened

```
C:     long setAuxCommOpenCmd(const long sensorIndex,
              const char * command, const long cmdLen)

C++:   long mySensor.setAuxCommOpenCmd(const char * command, const long cmdLen)

VB:    mySensor.setAuxCommOpenCmd(ByVal command As String) As Long
```

PARAMETERS

`command` – The command to send to the auxiliary serial port when it is opened. The command is not assumed to be null terminated, so any arbitrary sequence of characters may be sent. The string should contain any terminating character (carriage return, etc.) required by the equipment attached to the auxiliary port.

cmdLen [not required from VB] – The length of the command. Valid range is 0 < cmdLen <= 80.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_BAD_PARAM if the `command` parameter is null, or if `cmdLen` is out of range.

COMMENTS

This function sets the command to be sent to the auxiliary communications port when the port is opened with the `setAuxCommOpen()` command (see page 33).

setAuxCommCloseCmd

Sets the command to send when the auxiliary communications port is closed.

```
C:     long setAuxCommCloseCmd(const long sensorIndex,
              const char * command, const long cmdLen)

C++:   long mySensor.setAuxCommCloseCmd(const char * command, const long cmdLen)

VB:    mySensor.setAuxCommCloseCmd(ByVal command As String) As Long
```

PARAMETERS

`command` – The command to send to the auxiliary serial port immediately before it is closed. Note that the command is not assumed to be null terminated, so any arbitrary sequence of characters may be sent. The string should contain any terminating character (carriage return, etc.) required by the equipment attached to the auxiliary communications port.

cmdLen [not required from VB] – The length of the command. Valid range is 0 < cmdLen <= 80.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_BAD_PARAM if the `command` parameter is null, or if `cmdLen` is out of range.

COMMENTS

This function sets the command to be sent to the auxiliary communications port immediately before the port is closed with the `setAuxCommClosed()` command (see page 34).

setAuxCommSampleCmd

Sets the command to send to tell the auxiliary communications port equipment to take a 'sample'.

```
C:     long setAuxCommSampleCmd(const long sensorIndex,
              const char * command,
              const long cmdLen)

C++:   long mySensor.setAuxCommSampleCmd(const char * command,
              const long cmdLen)

VB:    mySensor.setAuxCommSampleCmd(ByVal command As String) As Long
```

PARAMETERS

`command` – The command to send. Note that the command is not assumed to be null terminated, so any arbitrary sequence of characters may be sent. The string should contain any terminating character (carriage return, etc.) required by the equipment attached to the auxiliary communications port.

cmdLen [not required from VB] – The length of the command. Valid range is 0 < cmdLen <= 80.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_BAD_PARAM if the `command` parameter is null, or if `cmdLen` is out of range.

COMMENTS

This function sets the command to be sent to the auxiliary communications port to tell the equipment attached to the port to take a 'sample'. The command will be sent at the frequency specified by the `setAuxCmdFreq()` function (see page 37).

Note that a value must be set using this function, *and* a non-zero frequency must be set using setAuxCmdFreq() in order for the library to send the sampling command.

See the section "Using the Auxiliary Communications Port" in the Programmer's Guide for more information on using the auxiliary communications port.

setAuxCmdFreq

Sets the frequency at which the 'sample' command is sent to the auxiliary communications port.

```
C:     long setAuxCmdFreq(const long sensorIndex, const long freq)

C++:   long mySensor.setAuxCmdFreq(const long freq)

VB:    mySensor.setAuxCmdFreq(ByVal freq As Long) As Long
```

PARAMETERS

`freq` – The frequency at which to send the command to the auxiliary communications port.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_BAD_PARAM if `freq` is less than zero.

COMMENTS

This function sets the frequency at which to send the 'sampling' command (set with the `setAuxCommSampleCmd()` function – see page 36) to the auxiliary communications port.

The frequency is expressed in number of AR600 samples. For example, a frequency of 1 (one), tells the library to send the sampling command to the auxiliary communications port every time the library receives a sample from the AR600 sensor. A frequency of 5 tells the library to send the sampling command for every fifth sample received from the AR600 sensor.

Note that a value must be set using the `setAuxCommSampleCmd()` function, *and* a non-zero frequency must be set using setAuxCmdFreq`()` in order for the library to send the sampling command.

See the section "Using the Auxiliary Communications Port" in the Programmer's Guide for more information on using the auxiliary communications port.

getAuxCmdFreq

Returns the frequency at which the 'sample' command is sent to the auxiliary communications port.

```
C:     long getAuxCmdFreq(const long sensorIndex)

C++:   long mySensor.getAuxCmdFreq()

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

PARAMETERS

None.

RETURN VALUES

The frequency set with the `setAuxCmdFreq()` function.

COMMENTS

None.

setAuxCommSampleDelim

Sets the delimiter used to read data from the auxiliary communications port equipment.

```
C:     long setAuxCommSampleDelim(const long sensorIndex,
              const char delimChar)

C++:   long mySensor.setAuxCommSampleDelim(const char delimChar)

VB:    mySensor.setAuxCommSampleDelim(ByVal delimChar As Byte) As Long
```

PARAMETERS

`delimChar` – The character which delimits individual data samples from the equipment attached to the auxiliary communcations port. If not explicitly set, the default is Carriage Return (ASCII code 13).

RETURN VALUES

Always returns CTI_SUCCESS.

COMMENTS

When reading data from the auxiliary communications port, the CTI-AR600 library will assume the individual data samples are delimited by the character specified by this command. However, if a non-zero value has been set using the `setAuxCommSampleLen()` function (see page 39), then the library will always read the specified number of bytes, and will *not* use the delimiter character specified by the `setAuxCommSampleDelim()` function.

See the section “Using the Auxiliary Communications Port” in the Programmer’s Guide for more details.

setAuxCommSampleLen

Sets the number of bytes per sample to read from the auxiliary communications port equipment.

```
C:     long setAuxCommSampleLen(const long sensorIndex,
              const long bytesPerSample)

C++:   long mySensor.setAuxCommSampleLen(const long bytesPerSample)

VB:    mySensor.setAuxCommSampleLen(ByVal bytesPerSample As Long) As Long
```

PARAMETERS

`bytesPerSample` – The number of bytes which constitute a 'sample' from the equipment attached to the auxiliary communications port. Must be zero or greater. Setting a value of zero implies that the delimiter character set by the `setAuxCommSampleDelim()` function will be used when reading from the auxiliary port, instead of the number of bytes per sample.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_BAD_PARAM if the parameter is out of range.

COMMENTS

By default, when reading data from the auxiliary communications port, the CTI-AR600 library assumes that individual data samples are delimited by the character specified by the `setAuxCommSampleDelim()` function (see page 38). If this is not the case (for example, the equipment attached to the auxiliary port always sends 4 bytes per sample, with no particular delimiter character), then `setAuxCommSampleLen()` should be used to tell the library how many bytes to read for each 'sample'.

See the section "Using the Auxiliary Communications Port" in the Programmer's Guide for more information on using the auxiliary communications port.

writeAuxCommData

Writes data to the auxiliary communications port.

```
C:     long writeAuxCommData(const long sensorIndex,
              const char * data, const long dataLen)

C++:   long mySensor.writeAuxCommData(const char * data, const long dataLen)

VB:    mySensor.writeAuxCommData(ByVal data As String) As Long
```

PARAMETERS

`data` – The data to send to the auxiliary port. The data is not assumed to be null terminated, so any arbitrary sequence of characters may be sent. The data should contain any terminating character (carriage return, etc.) required by the equipment attached to the auxiliary communications port.

dataLen [not required from VB] – The length of the data string. Must not be negative.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_COMM_NOT_OPEN if the auxiliary communications port has not been opened.
CTI_BAD_PARAM if the `command` parameter is null, or if `cmdLen` is out of range.

COMMENTS

This function sends data to the auxiliary communications port. This function has no other interaction with the CTI-AR600 library.

See the section "Using the Auxiliary Communications Port" in the Programmer's Guide for more information on using the auxiliary communications port.

setAuxCommHWFlowControlOn

Turn on hardware flow control (using the DTR line) for the auxiliary communications port.

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

C++:   long mySensor.setAuxCommHWFlowControlOn()

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

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_COMM_NOT_OPEN if `setAuxCommOpen()` has not been called.
CTI_FAILURE if any error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This function sets the auxiliary serial port to use the CTS and DTR lines for flow control of data sent to and from the equipment attached to the serial port. For this setting to be effective, the serial cable used to connect the equipment must implement the CTS and DTR lines.

setAuxCommHWFlowControlOff

Turn off hardware flow control for the auxiliary communications port.

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

C++:   long mySensor.setAuxCommHWFlowControlOff()

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

PARAMETERS

None.

RETURN VALUES

CTI_SUCCESS if successful.
CTI_COMM_NOT_OPEN if `setAuxCommOpen()` has not been called.
CTI_FAILURE if any error occurs. Call `getErrorMessage() to` get an extended error message.

COMMENTS

This function sets the auxiliary serial port to NOT use the CTS and DTR lines for flow control of data sent to/from the computer.

getIsAuxCommHWFlowControlOn

Determine if hardware flow control, is on for the auxiliary communications port.

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

C++:   long mySensor.getIsAuxCommHWFlowControlOn()

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

PARAMETERS

None.

RETURN VALUES

Returns 1 (one) if hardware flow control is on, and 0 (zero) otherwise.

COMMENTS

None.

Data Format Functions

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

COMMENTS

This is equivalent to the sensor's 'A2' 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

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.

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 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.

setMaxValidRange

Sets the maximum 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 44).

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.

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

None.

setMinValidRange

Sets the minimum 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 44).

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 zero range reference point of the sensor from the hardware default to any desired position.

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.

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_COMM_NOT_OPEN if `setCommOpen()` has not been called.

CTI_SUCCESS if the function succeeds.

CTI_BUFFER_TOO_SMALL if the buffer is too small to hold the full length of the firmware version string. As much of the firmware version string as will fit in the actual buffer space is returned.

CTI_FAILURE if any other error occurs. Call `getErrorMessage()` to get an extended error message.

COMMENTS

This function returns the version of the Acuity AR600 sensor's firmware. Use `getLibraryVersion()` (see page 54) 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()` (page 52) 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()` (page 52) to get the Acuity sensor hardware's firmware version.