INSTRUCTION MANUAL

PC9000 Software Development Kit Reference Manual

Windows 32-bit Link Libraries for the
CR9000 and CR5000 Measurement and Control Systems

Revision: 10/04

SDK Version 1.0
(PC9000.DLL Version 5.0)

License for Use

This PC9000 Software Development Kit software, hereinafter referred to as the PC9000 SDK, is protected by both United States copyright law and international copyright treaty provisions. The installation and use of this software constitutes an agreement to abide by the provisions of this license agreement. The term "developer" herein refers to anyone using this PC9000 SDK.

The core operational files provided with this PC9000 SDK (hereinafter referred to as “PC9000 Binaries”) include the files PC9000.DLL and PLA100DD.SYS.

Developer may make a copy of this software on a second computer for the sole purpose of backing-up CAMPBELL SCIENTIFIC, INC. software and protecting the investment from loss. All copyright notices and labeling must be left intact. Developer may distribute or sell their software including the PC9000 Binaries subject to the terms hereafter set forth.

RELATIONSHIP

Campbell Scientific, Inc. hereby grants a license to use PC9000 Binaries in accordance with the license statement above. No ownership in Campbell Scientific, Inc. patents, copyrights, trade secrets, trademarks, or trade names is transferred by this Agreement. Developer may use these PC9000 Binaries to create as many applications as desired and freely distribute them. Campbell Scientific, Inc. expects no royalties or any other compensation outside of the PC9000 SDK purchase price. Developer is responsible for supporting applications created using the PC9000 Binaries.

RESPONSIBILITIES OF DEVELOPER

The Developer agrees:

To provide a competent programmer familiar with Campbell Scientific, Inc. datalogger programming to write the applications.

Not to sell or distribute documentation on use of PC9000 Binaries.

Not to sell or distribute the applications that are provided as examples in the PC9000 SDK. Developers may copy and paste portions of the code into their own applications, but their applications are expected to be unique creations.

This Agreement does not give Developer the right to sell or distribute any other Campbell Scientific, Inc. Software (e.g., PC9000, Program Generators, LoggerNet or any of their components, files, documentation, etc.) as part of Developer's application.

Not to develop applications that compete directly with any application developed by Campbell Scientific, Inc. or its affiliates.

To assure that each application developed with PC9000 Binaries clearly states the name of the person or entity that developed the application. This information should appear on the first window the user will see.

WARRANTY

There is no written or implied warranty provided with the PC9000 SDK software other than as stated herein. Developer agrees to bear all warranty responsibility of any derivative products distributed by Developer.

TERMINATION

Any license violation or breach of Agreement will result in immediate termination of the developer's rights herein and the return of all PC9000 SDK materials to Campbell Scientific, Inc.

MISCELLANEOUS

Notices required hereunder shall be in writing and shall be given by certified or registered mail, return receipt requested. Such notice shall be deemed given in the case of certified or registered mail on the date of receipt.

This Agreement shall be governed and construed in accordance with the laws of the State of Utah, USA. Any dispute resulting from this Agreement will be settled in arbitration.

This Agreement sets forth the entire understanding of the parties and supersedes all prior agreements, arrangements and communications, whether oral or written pertaining to the subject matter hereof. This Agreement shall not be modified or amended except by the mutual written agreement of the parties. The failure of either party to enforce any of the provisions of this Agreement shall not be construed as a waiver of such provisions or of the right of such party thereafter to enforce each and every provision contained herein. If any term, clause, or provision contained in this Agreement is declared or held invalid by a court of competent jurisdiction, such declaration or holding shall not affect the validity of any other term, clause, or provision herein contained. Neither the rights nor the obligations arising under this Agreement are assignable or transferable.

If within 30 days of receiving the PC9000 SDK product developer does not agree to the terms of license, developer shall return all materials without retaining any copies of the product and shall remove any use of the PC9000 Binaries in any applications developed or distributed by Developer. CSI shall refund 1/2 of the purchase price within 30 days of receipt of the materials. In the absence of such return, CSI shall consider developer in agreement with the herein stated license terms and conditions.

Limited Warranty

CAMPBELL SCIENTIFIC, INC. warrants that the installation media on which the accompanying computer software is recorded and the documentation provided with it are free from physical defects in materials and workmanship under normal use. CAMPBELL SCIENTIFIC, INC. warrants that the computer software itself will perform substantially in accordance with the specifications set forth in the instruction manual published by CAMPBELL SCIENTIFIC, INC.

CAMPBELL SCIENTIFIC, INC. will either replace or correct any software that does not perform substantially according to the specifications set forth in the instruction manual with a corrected copy of the software or corrective code. In the case of significant error in the installation media or documentation, CAMPBELL SCIENTIFIC, INC. will correct errors without charge by providing new media, addenda or substitute pages.

If CAMPBELL SCIENTIFIC, INC. is unable to replace defective media or documentation, or if CAMPBELL SCIENTIFIC, INC. is unable to provide corrected software or corrected documentation within a reasonable time, CAMPBELL SCIENTIFIC, INC. will either replace the software with a functionally similar program or refund the purchase price paid for the software.

The above warranties are made for ninety (90) days from the date of original shipment.

CAMPBELL SCIENTIFIC, INC. does not warrant that the software will meet licensee's requirements or that the software or documentation are error free or that the operation of the software will be uninterrupted. The warranty does not cover any media or documentation that has been damaged or abused. The software warranty does not cover any software that has been altered or changed in any way by anyone other than CAMPBELL SCIENTIFIC, INC. CAMPBELL SCIENTIFIC, INC. is not responsible for problems caused by computer hardware, computer operating systems or the use of CAMPBELL SCIENTIFIC, INC.'s software with non-CAMPBELL SCIENTIFIC, INC. software.

ALL WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED AND EXCLUDED. CAMPBELL SCIENTIFIC, INC. SHALL NOT IN ANY CASE BE LIABLE FOR SPECIAL, INCIDENTAL, CONSEQUENTIAL, INDIRECT, OR OTHER SIMILAR DAMAGES EVEN IF CAMPBELL SCIENTIFIC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

CAMPBELL SCIENTIFIC, INC. IS NOT RESPONSIBLE FOR ANY COSTS INCURRED AS A RESULT OF LOST PROFITS OR REVENUE, LOSS OF USE OF THE SOFTWARE, LOSS OF DATA, COST OF RE-CREATING LOST DATA, THE COST OF ANY SUBSTITUTE PROGRAM, CLAIMS BY ANY PARTY OTHER THAN LICENSEE, OR FOR OTHER SIMILAR COSTS.

LICENSEE'S SOLE AND EXCLUSIVE REMEDY IS SET FORTH IN THIS LIMITED WARRANTY. CAMPBELL SCIENTIFIC, INC.'S AGGREGATE LIABILITY ARISING FROM OR RELATING TO THIS AGREEMENT OR THE SOFTWARE OR DOCUMENTATION (REGARDLESS OF THE FORM OF ACTION; E.G., CONTRACT, TORT, COMPUTER MALPRACTICE, FRAUD AND/OR OTHERWISE) IS LIMITED TO THE PURCHASE PRICE PAID BY THE LICENSEE.

CAMPBELL SCIENTIFIC, INC.

815 W. 1800 N.
Logan, UT 84321-1784
USA
Phone (435) 753-2342
FAX (435) 750-9540
www.campbellsci.com

Campbell Scientific Canada Corp.
11564 -149th Street
Edmonton, Alberta T5M 1W7
CANADA
Phone (780) 454-2505
FAX (780) 454-2655

Campbell Scientific Ltd.
Campbell Park
80 Hathern Road
Shepshed, Loughborough
LE12 9GX, U.K.
Phone +44 (0) 1509 601141
FAX +44 (0) 1509 601091

PC9000 SDK Table of Contents

PDF viewers note: These page numbers refer to the printed version of this document. Use the Adobe Acrobat® bookmarks tab for links to specific sections.

Section 1. PC9000 SDK Overview

The PC9000 Software Development Kit (PC9000 SDK) is a programming interface designed for use with Microsoft Visual Basic. It provides direct, real-time communication capability with CR9000/CR5000 dataloggers, allowing a single open connection at a time. It is a standard, "call-level" API, meaning that at this time it does not support OLE automation, and does not need to be registered in the system registry.

The PC9000 SDK consists of the PC9000.DLL and associated drivers, and currently supports the following CSI Datalogger configurations:

- CR9000 connection via a BLC100 PC Bus Link Card (TLink) - Windows 95/98/ME only
- CR9000 connection via a PLA100 Parallel Port Link Adapter (LPT)
- CR9000 connection via a TL925 RS-232 Interface (COM)
- CR9000 connection via a NL105 Network Link Interface (NET)
- CR5000 connection via a 9-pin RS-232 serial cable (COM)
- CR5000 connection via a NL100/NL105 Network Link Interface (NET)

1.1 General Notes on DLL Usage

By default, the PC9000.DLL will be installed in the C:\Campbellsci\PC9000SDK\DLL\ directory. Before running the included example application, you must do one of the following:

- Place the DLL in the example's application directory
- Create a new PATH environmental variable describing the location of the PC9000.DLL
- Place the DLL in the "Windows System" directory. The Windows System directory is normally C:\WINDOWS\SYSTEM for Windows 95/98/Me systems, C:\WINNT\SYSTEM32 for Windows NT 4.0 and Windows 2000 systems, or C:\WINDOWS\SYSTEM32 for Windows XP systems.

If desired, PC9000.DLL can alternately be located in other directories; for example, some developers prefer to keep a copy of all key DLL files in the same subdirectory as the applications program that they create. By locating a DLL in the same directory as a custom program, multiple DLL versions can reside on the same machine at the same time. For custom, PC9000.DLL-based applications, this will not be of any concern unless the computer in question also runs a copy of PC9000 Version 5.X software. In that case, if the PC9000 V5.X installation uses an older version of PC9000.DLL than the one provided with this software development kit, and there is some specific concern about PC9000 DLL compatibility, then keeping a separate copy of the DLL in the custom application directory may be warranted.

If you feel that it is essential to install a dedicated copy of PC9000.DLL with your custom applications, you may also want to rename the DLL copy that you distribute. (The function declaration statements within your source code will need to be modified as well for this to work as intended.) If the names are not different, one can never be certain that all versions of Windows will always load the desired version of the DLL when a program starts – it may depend upon whether a version is already resident in memory due to some other current or previously running program that uses PC9000.DLL.

Generally, the version of PC9000.DLL distributed with PC9000 Version 5.X releases will be synchronized with current release versions of the PC9000 SDK. Therefore, before attempting to run multiple versions simultaneously, it is recommended that you upgrade the PC9000 program on the computer in question to be current with your custom applications.

When invoking a DLL function from within a Visual Basic program for the first time, Windows will look for the DLL in the local subdirectory, and then in the Windows System directory and in any other subdirectories included in the PATH environment variable. If it cannot locate the DLL in any of these directories, VB will raise an Error 53 on the line of code where the first DLL function is invoked.

1.2 Declaring and Calling DLL Functions from Visual Basic

Declaring and using PC9000.DLL functions within a Visual Basic program is no different than using standard Windows API functions. For general assistance in calling API functions from within Visual Basic, consult the applicable Microsoft Visual Basic and/or MSDN documentation. The required declare statements all appear together in the last section of this manual. They also appear in the documentation for each individual function.

For simplicity, the declare statements are often placed in a code (.BAS) module of the programmer's choice. They are then available for use in all form, code, and class modules within the application. If the DLL functionality is to be encapsulated within a VB form or class module, the declare statements can be located there, but the "Private" keyword will need to be added at the beginning of each Declare statement, else the code will not compile. In that case, the functions will only be usable from routines within the form/class module.

API-style DLL functions do not raise errors to Visual Basic. The success or failure of each function must be determined within the Visual Basic program by evaluating the return codes, as documented in this reference. On the other hand, if fundamental errors occur in linking the DLL function to the application, the Visual Basic runtime engine will not be able to properly call the DLL function in the first place. In such cases runtime errors will be raised to the application program by the runtime engine itself. These are explained in more detail below.

Declare statements are not checked for correctness against the DLL until the function is actually called by the program. Each invocation of a particular DLL function within the VB code must match perfectly with the function as described in the Declare statement, but there is no way to insure that the Declare statement is correct until the function is actually used. (This is what is

meant by “dynamic linking”.) Therefore the following Visual Basic errors may occur if the declare statements do not properly match the DLL:

Error 453: “Specified DLL function not found”, or
“Can't find DLL entry point [function name] in PC9000.DLL”

In this case, the DLL file itself was found, but the particular function name (as specified in the Declare Statement) could not be located.

Error 49: “Bad DLL calling convention”

In this case, the specified DLL function was found, but either the number of parameters or the parameter data types as specified in the Declare statement, don’t match what was found in the DLL.

Other times, mismatched data types may slip through Visual Basic’s runtime type checking, only to cause unexpected problems in the DLL, causing General Protection Faults (GPFs). Use care at all times when declaring and using DLL functions to insure successful results.

All PC9000.DLL functions run in the thread of the calling Visual Basic program, meaning that the VB program execution is effectively “blocked”, or suspended on the line of code which called the DLL function, until that function completes. None of the functions employ callbacks, requiring the use of the Visual Basic 'AddressOf' operator when calling the function.

During the time that Visual Basic code’s thread of execution is suspended, the program will appear to be hung, though it is not. This time can be noticeable for data retrieval calls returning large blocks of data. It is also noticeable for any call that encounters a communications interruption between the computer port and the datalogger: the function will not complete its operation until either communications is established, or it times out. While some control over timeouts is possible through the various arguments passed to the functions, the specific time delays are more influenced by the particular port in use, and also by the operating system version. At the time of this release, the behavior of the DLL functions in response to communications interruptions is noticeably slower under Windows NT 4.0 than on Windows 95/98 computers.

1.3 PC9000.DLL Function Arguments and Return Codes

While this DLL was designed for use with Visual Basic, no VB-specific data types (such as Variant, Boolean, Date or variable dimension arrays) have been used in the arguments or return codes. Therefore, it is possible to use the DLL with other languages such as C, C++ or Delphi, although no specific examples are provided in this document.

All arguments and function return codes are one of the following Visual Basic data types:

- Integer (16-bit Integer)
- Long (32-bit Integer)

- Single (IEEE 4-byte floating point)
- Single Array (IEEE 4-byte floating point)

1.3.1 Interpreting Single Data Types

All single values and single value arrays are returned to the calling routine as 4 byte IEEE floating point, regardless of how they are stored inside the datalogger.

In instances where the data are stored in the datalogger table as CSI 2-byte floating point, the conversion within the DLL, prior to returning the values, is transparent to the application program using the DLL. Remember that in those cases, however, the return value or values will contain meaningless extra digits of precision. It is usually recommended that FP2 data types NOT be used in datalogger programs, unless the datalogger storage requirements absolutely require that type of data storage economy. Further, the use of FP2 data types will actually slow down the throughput in real time data retrieval applications, as each value must be converted to 4 byte floating point instead of being simply passed through without conversion.

In instances where the "Time of Maximum" or "Time of Minimum" data has been sampled into an output table, this data is stored in the datalogger table as two 32-bit long values. Any fields defined as such will be coerced into a single floating point value in the return array without raising errors or faults, but information will be lost in the conversion and the effective values will be meaningless. At present there is no way using real-time function calls, to retrieve this data as part of an entire, contiguous datalogger table data record. If it is important to at least suppress data in these instances, the DataType argument of the **GetFieldName2( )** function can be used to detect the occurrence of non-floating point table data types on a field-by-field basis. As another alternative, the **LogTable( )** function will correctly write all field data types to a disk file, where the information can then be retrieved and used.

1.3.1.1 "Not A Number" Conditions

A special case occurs in situations where a value or values retrieved from the datalogger correspond to the condition "not a number". This condition will occur in situations such as a measurement channel over-range, or an overflow/divide by zero error caused by some CRBasic math instructions. The IEEE floating point standards recognize specific floating point codes for these numbers, but Visual Basic is limited in its ability to deal with these special codes.

The binary representation of the 4-byte IEEE code corresponding to "not a number" is all ones, or Hex FFFFFFFF. This is also the value used internally by the Campbell dataloggers for this condition. A Visual Basic Single typed variable cannot be set to this value directly through Visual Basic script, but variables typed as Single can be set to this value externally, through API functions written in C or other languages (such as the PC9000.DLL). If Visual Basic encounters such a value in a return argument, it interprets the variable as "not a number": outputting the variable's value to a message box or string variable will yields the expression "-1.#QNAN".

To some extent, Visual Basic can deal with this condition. The "not a number" value can be successfully assigned to other Single typed variables, and any

math involving a variable with this value will result in an output that is also not a number. Unfortunately, any attempts to do bit-wise logical tests or comparisons on this variable, while it is typed as a Single, will result in overflow errors. As a result, doing anything other than outputting the value to text-based display controls will cause errors, resulting in messy error handlers in order to test for this condition.

To address this situation, the DLL pre-converts any such "not a number" values retrieved to a specific value that does not cause overflow errors in Visual Basic. The hex code chosen is FF7FFFFF, which corresponds to the largest negative number that can be expressed in the IEEE floating point standard. Visual Basic receives this code and expresses the value in decimal as -3.402823E+38. Unfortunately, due to round-off error, the code is actually closer to –3.4028234E+38, but Visual Basic does not allow you to enter that precise value as a Single typed value in VB code! The internal hex code that VB stores for a value entered in VB source code as –3.402823E+38, is FF7FFFFD. What this all means is that performing an equivalence test in VB, between the DLL's "not a number" code and the closest value that can be defined in source code for a Single-typed variable, will not produce the desired result.

To remedy this, and define a 4-byte single-precision floating point value in VB that exactly corresponds to the code returned from the DLL, use the following syntax:

```
Dim fVBMaxNegValue as Single

fVBMaxNegValue = CSng(-3.4028234E+38)
```

If depending upon floating point round-off errors in that manner scares you, then as an alternative, define the equivalent HEX code as a VB Long Integer, and "cast" this variable into a Single data type using the Windows CopyMemory API function, or some similar function written in C. That exercise is beyond the scope of this manual to explain in detail: however, the Visual Basic example program NotANumber.vbp (included on the PC9000 SDK disk), shows how this is done, and can also be used as an aide to better understand the Visual Basic behaviors described above.

1.3.2 Interpreting Integer Data Types

Visual Basic Long data type values are fixed 32-bit signed integer, and Integer data types are fixed as 16-bit signed integers. Most Long and Integer arguments and return codes are only meaningful within a limited range of positive values, so unsigned-to-signed conversions are usually not an issue.

Table record numbers are one exception to the above statement. Record numbers are long values that will eventually become greater than $2^{31}-1$, and the record number is defined internally within the datalogger and the DLL as a 32-bit unsigned integer. In the event that a datalogger record number exceeds $2^{31}-1$, the signed representation in Visual Basic will appear to "wrap around" to -2^{31} and begin counting up toward zero. When the unsigned, real record number value reaches $2^{32}-1$, (FFFFFFFF Hex), the signed Visual Basic value will have reached –1. At that point, the unsigned record number value in the datalogger wraps around to zero, the signed and unsigned values are once again in agreement, and the process repeats.

This is identical to behaviors of many common Windows API functions used in Visual Basic, such as GetTickCount( ), which returns the number of integer milliseconds since the last time the computer was booted. This value will eventually wrap around in similar fashion, given enough continuous operating time. If tracking record number values is critical to a given application, this (possibly) unexpected discontinuity needs to be taken into account.

1.3.3 Handling String Data Types

Wherever values are returned via strings, the calling routine must provide enough space to accommodate the values that will be returned. Either of the following two code fragments will allocate a suitable string.

```
Dim str As String * 100
```

or

```
Dim str as String
str = String(100, vbNullChar)
```

Then use the VB Len( ) function to establish the string size. For example,

```
iRslt = GetTableName(str, Len(str))
```

You need to dimension or set the string length one character larger than the maximum length of the string you are working with, as the last character is reserved for the null termination character (ASCII 0)

When string values are passed to a function but their contents are not altered by the function, the use of fixed, padded, or variable length strings is acceptable. Be careful, however, not to send a Visual Basic empty string, without at least a null termination character or single padded space, unless the examples for a specific function indicate to do so. In most cases, doing so may trigger a General Protection Fault (GPF), causing the application to terminate without warning.

1.3.4 DLL Function Return Values

All PC9000.DLL procedures are functions, in that they return some value as a result, in addition to the arguments that may be passed back and forth.

For the majority of functions, this return value is an integer result code, representing the outcome of the function as good, bad, or otherwise. In some cases, the return code is always zero. The return codes are completely specific to a given function, and are fully documented in the detailed function reference section of this document.

Generally, in cases where a DLL function returns result codes, and a datalogger or port is being accessed, the value of zero is reserved for "OK", although there are exceptions, as documented in the function reference. Other non-zero return codes either indicate a failure, or they provide the status of an ongoing transaction. Be aware that a return code of "OK" does not guarantee the desired outcome was achieved. Rather, it indicates that the underlying datalogger protocol transactions were executed successfully by the DLL, and no protocol errors were indicated.

The use of the return code of 1 is always reserved for instances where either a port has not been successfully opened, or all attempts to communicate with the datalogger have timed out with no response. If this response is not applicable for a given function, the return code 1 will not be used. This applies to all DLL functions.

For data collection transactions, the validity checking in the DLL (and in the protocol transactions themselves) is more thorough, so that a return code of zero should always mean success. Codes 2 and 3 are utilized in a common fashion across all of the data collection functions, as explained below:

Return code 2, "No data", will always result if a datalogger response was obtained, but one of the following conditions exists:

- The table is currently empty (i.e., no records).
- A record-specific data collection call was made, but the specified record number is exactly one higher than the current record.
- A field-level data retrieval call was made, and the specified table name does not exist within the current table definitions. (Also will apply to bad field names in certain cases – see exception note below regarding return code 3).

Return code 3, "Bad table name / Bad field name" will normally only occur in the following situations:

- A field name must be specified in a data collection function, and the field name that was passed does not exist in the specified table.

  NOTE: Exceptions - for certain specific data retrieval functions (**GetVariable( ), GetPartialFieldArray( )**), a return code of 2 will occur in these cases.

- A table-level data retrieval call was made, and the specified table name does not exist within the current table definitions.

In an analogous fashion, all file management functions reserve the return code 3 for "Bad file name".

Return codes 2 or 3 will also often result if a datalogger response was obtained, but the internal table definition information within the DLL is no longer current. This occurs if either a new program has been compiled and set to run, or a different datalogger is being accessed (usually due to a port change). Refer to the section entitled "Order of operations for data retrieval", later in this overview, for a detailed explanation of the DLL and datalogger behavior that underlies this condition. In these instances it is very difficult to predict how the DLL will interpret the arguments in the data retrieval function call, since it is not operating with correct datalogger configuration information.

For data collection calls which retrieve arrays of data rather than single values, the other parameters will need to be evaluated carefully as well, to completely determine the precise outcome of the function call.

There are some situations in which the misuse of DLL functions may not be cleanly trapped by the DLL. In these cases, a somewhat misleading error code,

or no error code at all, may be returned. These situations would most likely involve improper naming syntaxes on tables, fields or file names, or improper settings for certain DLL function parameters. The datalogger protocol transactions may attempt to execute based upon the faulty information, with unexpected results in the datalogger. The DLL does some basic validation checks, but extended parameter checking is left out in the interest of simplicity and overall real time performance.

Use extreme care at all times in setting up DLL calls involving datalogger transactions, particularly those that have the effect of changing a state or changing the running configuration of the datalogger. As you become more familiar with the datalogger's basic internal organization, and the functions available here, you may desire to add your own, more extended, pre- and post-validation steps to your routines that execute critical datalogger configuration tasks and state changes.

In a few isolated cases, the DLL function return value is not a code, but is a value representing some quantity related to the performance of the function. These exceptions are:

- ClosePort ( ) – returns the protocol "best packet size" (Integer)
- FP2ToSingle ( ) – returns the conversion function's result (Single)
- LongFromString ( ) – returns the conversion function's result (Long)
- SingleFromString ( ) – returns the conversion function's result (Single)
- UserRd ( ) – returns the number of characters retrieved from the input buffer (Integer)

1.4 General Notes on the Use of PC9000.DLL Functions

1.4.1 Functions Not Directly Controlling Datalogger Communication

Most functions in the DLL directly invoke some communications transaction with the datalogger. The following functions known as "Miscellaneous Utility Functions", however, are exceptions to that rule and may be used at any time:

- GetCR9kApiVers( )
- FP2ToSingle( )
- LongFromString( )
- SingleFromString( )
- RdStatus( )

The following two functions control the low-level logging capabilities built into the DLL (rarely used in applications). While associated with datalogger

communication, they do not initiate or terminate datalogger communications in any way, and low-level logging runs independently of any particular I/O port. Therefore, these functions also may be invoked at any time:

- StartIOLog( )
- StopIOLog( )

The following functions, known as the "Port Control Functions" do not communicate with the datalogger, but do control the opening and closing of the ports through which communications occurs. Their uses are explained in detail in the section immediately following:

- OpenCom( )
- OpenCSICard( )
- OpenLpt( )
- OpenSock( )
- ClosePort( )

1.4.2 Open Port Instructions and Related Issues

All remaining functions not listed above perform real time datalogger communications and therefore assume a viable datalogger connection. **UserRd( )**, **UserWr( ),** and **GetModemStatus( )** are partial exceptions, in that they do not necessarily require a datalogger connection, depending upon their use, but always at least require a viable open port. Correspondingly, for all these communication functions, one of the "Open Port" port control functions must have been successfully executed, prior to their use, otherwise errors will be returned.

Open Port instructions only initialize the physical port's device driver. They will fail if the device or port is not available. These instructions do not communicate with the datalogger, however, so the datalogger does not have to be connected and active for these calls to succeed. Generally, one of the two functions, **GetLgrIdent( )** or **SetLgrClock( )** (used also to read the datalogger clock) are used if one desires to perform a simple, quick check to insure that a datalogger is connected to an open port and is on line.

ClosePort( ) is used to close the I/O port at the end of program execution or prior to changing to another port. **ClosePort( )** returns best packet size, which can be stored in the application program memory and passed to subsequent open port functions. Calling **ClosePort( )** with no port open does not by itself have any unwanted side effects, however please note carefully the paragraphs that follow.

The DLL operates within an application on a "single-port, single-device at a time" model. What this means is that, although the DLL can connect to different dataloggers through different ports from the same application, it must do so in sequence, not simultaneously. In other words, the DLL only keeps track of one open port connection to a datalogger at a time. All datalogger

communication functions simply assume that their communications is with the currently "open" port: there are no logical station numbers, port handles, etc. to pass to any of those functions.

NOTE

It is very important that, when communicating with multiple dataloggers, the **ClosePort( )** instruction be executed after opening each port, before opening any subsequent port. If **ClosePort( )** is not called, the subsequent port may actually open successfully, and DLL communications may indeed switch to the new port, as if everything was OK. The DLL has not preserved state information regarding the original port, however. As a result, after opening and closing a second port, without having first closed the original port, all communications functions will then return error codes when used, even though a port is still open.

Further, attempts to issue multiple **ClosePort( )** instructions at that point will not succeed in allowing the first port to be re-opened. The application will likely not re-allow communications through the original port until the application is entirely shut down and re-started, in which case the operating system releases the port's resources.

One last important programming point of note with regard to opening and closing ports, with special application to Visual Basic:

Any time that your application is running in non-compiled mode from within VB, your program is running in the same process space as the development environment. Since the PC9000.DLL API function calls operate entirely external to Visual Basic, the VB development environment knows nothing of the status of any open comm port resources when your application shuts down. Therefore, if your application code does not make sure to close out any open comm ports before shutting down, they will not be automatically released by VB. You will need to shutdown and re-start Visual Basic in order to get access to any such comm port that was left open. The above also applies to the conditions when, by using the "End" menu command in VB to abruptly terminate your application, the shutdown code (normally called from the main form's Form_Unload routine) is bypassed.

1.4.4 Port Timeout Issues

In general, once a communications port is opened, it remains open until such time that the application closes the port through the DLL. There is an important exception regarding the CR5000, however, as explained below.

The CR5000 incorporates some advanced power management features that can affect communications with an external computer. Of particular note is the “RS232 Timeout” setting, accessible from the front panel of the instrument through the “Configure” menu. When this setting is set to “Yes”, the CR5000 will itself terminate any open serial link after approximately 30 seconds of idle time (i.e., no communications occurring). This is used in situations in which the datalogger is running on limited remote power, where inadvertently leaving a communications interface open potentially represents a significant drain on available power.

In the event that this timeout occurs while communicating with the datalogger, the next attempt at communications will result in an error code, usually code 1 for “port not open or datalogger does not respond”. Once the link has been terminated in this fashion, the only recourse is to close and re-open the port, which will quickly re-establish the link.

This close/re-open cycle has an unacceptable side effect, however, when communicating through a modem link. Whenever the port is closed and re-opened (or the link is idled long enough that the CR5000 times out), any modem connections will be lost and will need to be re-established. Therefore the only sure ways to keep a CR5000 link active indefinitely are: 1) set the CR5000's RS232 Timeout to “No” or 2) maintain some periodic background communications with the CR5000, every 15-20 seconds, whenever the port is open. A timer-driven routine calling **GetLgrIdent( )** or **SetLgrClock( )** is usually the simplest way to accomplish this. Unfortunately, at this time there is no direct way to determine the state of the CR5000's RS232 timeout setting through the DLL communications interface.

One further note regarding low-power CR5000 applications. When the RS-232 timeout is set to “Yes”, executing the **CR9000Hangup( )** function before closing the port will immediately cause the CR5000 to terminate the remote link, not waiting for the idle timeout to occur. This may be useful in situations where the power is at a premium for every second of extra time spent running a remote communications link. If this is done when the timeout is not activated, there is no harm done, but the **CR9000Hangup( )** command has no effect.

1.4.5 Order of Operations for Data Retrieval

PC9000.DLL provides a layer between the low-level CSI datalogger communications protocols and a user application program. Part of its role in this regard is to handle the error and validity checking that is integral to, and required by, the datalogger protocols. These protocols insure that the real time data retrieved by the application will be reliable and free from errors or misrepresentations. Most of this validation checking is handled by the DLL in a fashion that is completely invisible to the user.

There is a very noteworthy exception, having to do with the handshaking that occurs between the datalogger and the computer, regarding the structure of the program's output data tables, as explained below.

When the DLL requests information from the datalogger regarding the current table definitions (in response to an initial **GetTableName( )** function call), it obtains detailed information about all datalogger internal table record structures, including the field names and data formats within each table. The DLL also tabulates a unique "signature" for each table, as defined by the datalogger's protocol rules. This signature relates to the precise structure of the table. Later, when data retrieval calls are made, the program requesting datalogger data must pass the signature. This validates that the program's understanding of a table's current record structure is precisely correct. By performing this handshaking, data can be transmitted in a very compact and efficient "block" manner, as the datalogger and the receiving program both agree how the data is to be interpreted. Failure to pass a correct signature will cause data retrieval requests to fail.

This table signature safeguards against changes to the structures of the user tables that might otherwise occur in a manner not detected by the application's

data retrieval routines. For instance, if the currently running program stopped and a new program started, the structure of the tables may change. It is imperative while performing high-speed data retrieval of complete table records, that the datalogger and the computer's application program remain in full agreement as to the data types, associated field names, and the ordering of the data contained therein.

Whenever **GetTableName( )** is invoked by an applications program, with an empty string passed for the TableName argument, the DLL will retrieve new definitions for all tables within the datalogger whose connection is currently active. All previous table definition information will be deleted, and the new information will be stored in the DLL's dynamic global memory, for access by all other DLL data retrieval functions.

This is analogous to the DLL behavior regarding communications state information. As discussed earlier, the DLL only keeps one set of communications state information, regardless of how many different ports are accessed during the running of a client application program. Datalogger table definition information also follows this rule, in that the most recently obtained set of datalogger table definitions and signatures is the only set kept in memory by the DLL at any given time.

None of the table definition information is saved to disk by the DLL, so it must be retrieved by the DLL every time that the application starts, whether or not there have been changes to a datalogger's currently active table definitions. This means that, even if your table names never change and you've hard coded them into your applications program, the DLL will still need to refresh its own internal information before retrieving data.

Further, whenever a new or modified program is compiled and made active, the safe approach is to always refresh table definition information, even if the table definitions themselves weren't changed in the new datalogger program.

Making data retrieval calls without first insuring that the DLL table definition information is current will have unpredictable results. Usually, some error code will be returned. In some instances, however, particularly if table definition information for an old program or connection exists in the DLL's memory, the DLL may erroneously interpret data that was returned.

How does this affect the programming of PC9000.DLL-based applications? The following points should be noted.

General rules (to be safe):

- Execute the **GetTableName( )** function, with an empty string for the TableName argument, at least once, whenever:
    - Any new datalogger communications port has been opened, before any record-level data retrieval occurs.
    - A new program has been compiled and run (i.e., **DownloadStart( )** was invoked with the Options argument set equal to 4).

Potential Exceptions (for reducing overhead and increasing performance on very slow connections):

- You may find that it is better to close the current port whenever there are no datalogger communication tasks running, even when you are only communicating with a single datalogger through a single port connection. The internal cache of table signatures is not automatically erased when a port is closed and the same port is opened. In fact, the previous signature cache should remain available until the **GetTableName( )** instruction is re-executed, or the program is shut down. Therefore, it is possible to re-open a port and run datalogger communications without having to refresh the information each time, if data retrieval calls are always used with the same port and datalogger.

  Keep in mind in this case, however, that the DLL does not provide information on the state of its internal table signature cache. Tracking this information is entirely up to the application program.

1.4.6 Wait States Between Commands

In general, PC9000.DLL (and the datalogger with which it is communicating) can process sequential communications requests as fast as they can be invoked through application code.

In cases where data is being served by the datalogger and no additional datalogger action is warranted, no delays between calls are needed. Care should be taken, however, not to request data more often than is indicated by the table update times for a given program. The extra undue burden on the datalogger to service rapid-fire repetitive communications processing may in some cases result in skipped scans within the datalogger program where there otherwise would be none. The best approach when high data throughput rates are needed is to experiment with different function call strategies to see which combination provides the best result. Take advantage, where possible, of retrieving values from multiple records in a single call; this will result in less communications overhead, and therefore more efficient processing.

Conversely, in cases where DLL functions cause the datalogger to take other internal actions, you will want to factor in some sort of programmatic delay. Examples of these types of functions would be: setting clocks, starting file downloads, changing file statuses, etc. Often, datalogger function calls occurring immediately after such functions will fail, as the DLL perceives that the datalogger is not available when in fact it is simply still busy responding to the previous transaction. "Sleep" statements have been inserted in all code examples (in the detailed function reference section of this document) where this type of behavior is likely to occur. Sleep is not a recognized Visual Basic command; rather, it is a Windows API call which effectively functions as a wait statement. The declaration statement for that function is as follows:

```
Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds as Long)
```

In general, this is the recommended way to insert delays between a series of sequential communications function calls in your VB code. It is much preferred over Visual Basic Do…Loop until Timer methods: those methods effectively cause the Visual Basic application to "hog" CPU time, greatly affecting the performance of any other applications, services, or drivers that are

also running. Keep in mind when using the Sleep statement, that your application (or thread) will be completely idle, and therefore unresponsive to all outside events, during the time of the Sleep statement. If this is unacceptable, better to use a combination of the Do … Loop and Sleep statements, breaking up the total wait time into chunks no smaller than 50 milliseconds. For instance, as shown in the following pseudo-code:

```
' Do two datalogger functions, separated by an ~2 sec. wait time

Dim iCnt as Integer

DoFirstLoggerTrans()

Do
   Sleep 200
   UpdateProgressWindow()
   DoEvents
   iCnt = iCnt + 1
Loop Until iCnt = 10

DoSecondLoggerTrans()
```

The duration of the wait times as shown in the examples are considered as recommended starting points, providing satisfactory pauses in most cases, and without causing undue application delays. Optimal times may vary, based upon the datalogger type being used, the particular program being run, and the speed of the computer on which the application is running. You may want to experiment with decreasing or increasing these times as warranted in your own applications, for best performance.

1.4.7 Selecting the Best-Suited Data Retrieval Functions

PC9000.DLL provides a dozen or so different function calls to be used for retrieving data from the dataloggers. Each one has its own unique purposes. While the function reference section explains in detail how each function is used, it is also be helpful for the uninitiated to have an overview of the various functions, so one is provided here.

The functions are first split into three major groups, as follows:

- Current value only, single data value functions. These functions return the current value only of a single field, and provide no other information. They are the easiest to set up and use when simply doing qualitative checks on one or a small handful of current data values, but have limited utility beyond those uses.

- Real-time bulk data retrieval. These functions return values in arrays. They also can retrieve values from many records in a single call, and provide more information about the data retrieved. As such, they are more complex to set up and use, but are essential in obtaining and trending datalogger table contents over specified periods of time.

- Logging to disk. These functions enable table data to be automatically logged by the DLL, without any processing required by the application.

Outline of functions according to these categories:

1. **Current value only, single data value:**
   GetVariable( ) (Floating point data only)
   GetCurrentValue( ) (All data types)
   GetStatusValue( ) (All data types – Status Table only)

2. **Real-time bulk data retrieval:**
   2.1. Entire Records:
   GetRecentRecords( ), GetRecentRecordsTS( )
   GetRecordsSinceLast( ), GetRecordsSinceLastTS( )
   2.2. Single Field Values:
   GetRecentValues( ), GetRecentValuesTS( )
   GetValuesSinceLast( ), GetValuesSinceLastTS( )
   2.3 Individual Arrays Within a Record:
   GetPartialFieldValues( )
   GetPartialFieldArray( )

3. **Real-time bulk data retrieval:**
   LogTable( )

Section 2. PC9000.DLL Function Reference

2.1 Port Control Functions

2.1.1 OpenCom( ) - Open RS-232 Serial Port

This applies to either the CR9000 (using the TL925 Serial Interface) or the CR5000 (using a direct serial cable).

Declaration:

```
Declare Function OpenCom Lib "PC9000.DLL" (ByVal
Port As String, curBaudRate As Long, ByVal
ExtraResp As Long, ByVal maxPktSize As Integer,
ByVal BestPktSize As Integer, ByVal ModemOn As
Integer) As Integer
```

Parameters:

Port is a string like "COM1" or "COM2" or any valid COM port name.

curBaudRate is the baud rate to be used with the COM port. If the specified baud rate can not be used, the port will be opened at a lower baud rate and the function will return the value actually used.

ExtraResp is extra response time in msec per packet, used by the datalogger communication protocol. Extra response time is a user entered value. It can be as low as zero, but may need to be as high as 15000 (15 seconds) or more on some remote connections.

maxPktSize is maximum packet size, used by the datalogger communication protocol. Maximum packet size is a user entered value and must be 32, 64, 128, 256, 512, 1024, or 2048. Start with 2048 and decrease only if communications problems are encountered.

BestPktSize specifies a best packet size to be used initially in communicating with the datalogger. Best packet size is then adjusted within the DLL each time communication takes place. It is reported back to the program through the **ClosePort( )** command and should be remembered for a given port and station between communications sessions. A good starting value is 512 or the maxPktSize, whichever is smaller.

ModemOn specifies whether the COM port connection to the datalogger will be through a phone modem. 0=False, 1=True. For timing purposes only; does not perform, enable or inhibit any modem functions.

Return Codes:

0 = Port was successfully opened.

1 = Port could not be opened: device failure, or already in use by this or other application.

Example:

```
Dim sPort as String
Dim lBaudRate as Long
Dim lExtraResp as Long
Dim iMaxPktSize as Integer
Dim iBestPktSize as Integer
Dim iModemOn as Integer
Dim iRslt as Integer

sPort = "COM2"
lBaudRate = 115200
lExtraResp = 500
iMaxPktSize = 2048
iBestPktSize = ClosePort()
iModemOn = 1
iRslt = OpenCom(sPort, lBaudRate, lExtraResp, iMaxPktSize, _
 iBestPktSize, iModemOn)

If iRslt = 1 Then
   MsgBox "Com Port not available"
End If
```

2.1.2 OpenLpt( ) - Open Parallel Port for use with PLA100 Parallel Link Adapter

This port may only be used by the CR9000, through the PLA100 interface. Does not apply to the CR5000.

NOTE This function, and all subsequent datalogger communications functions through this port, will not execute under Windows NT/2000/XP without the installation of the PLA100DD parallel port driver. This driver should be in place on these systems if the full installation program has been run for either the PC9000DLL Software SDK, or PC9000 V 5.X or greater. Simply copying the PC9000.DLL files and the PLA100DD.SYS files onto a computer will not activate the driver.

Declaration:

```
Declare Function OpenLpt Lib "PC9000.DLL" (ByVal
LptName As String, ByVal ExtraResp As Long, ByVal
maxPktSize As Integer, ByVal BestPktSize As
Integer) As Integer
```

Parameters:

LptName is either "LPT1" or "LPT2".

ExtraResp is extra response time in msec per packet, used by the datalogger communication protocol. Extra response time is a user entered value. It can be as low as zero, but may need to go as high as 1000-2000 on noisy port connections.

MaxPktSize is maximum packet size, used by the datalogger communication protocol. Maximum packet size is a user entered value and must be 32, 64, 128, 256, 512, 1024, or 2048. Start with 2048 and decrease only if communications problems are encountered.

BestPktSize specifies a best packet size to be used initially in communicating with the datalogger. Best packet size is then adjusted within the DLL each time communication takes place. It is reported back to the program through the **ClosePort( )** command and should be remembered for a given port and station between communications sessions. A good starting value is 512 or the maxPktSize, whichever is smaller.

Return Codes:

0 = Port was successfully opened.
1 = Port could not be opened: device failure.

Example:

```
Dim sPort as String
Dim lExtraResp as Long
Dim iMaxPktSize as Integer
Dim iBestPktSize as Integer
Dim iRslt as Integer

sPort = "LPT1"
lExtraResp = 500
iMaxPktSize = 2048
iBestPktSize = ClosePort()
iRslt = OpenLpt(sPort, lExtraResp, iMaxPktSize, iBestPktSize)

If iRslt = 1 Then
   MsgBox "Parallel Port not available"
End If
```

2.1.3 OpenCsiCard( ) - Open Port for BLC100 Bus Link Card

This port may only be used by the CR9000, through the BLC100 interface. Does not apply to the CR5000.

Declaration:

```
Declare Function OpenCSICard Lib "PC9000.DLL"
(ByVal PortNbr As Integer, ByVal ExtraResp As
Long, ByVal maxPktSize As Integer, ByVal
BestPktSize As Integer) As Integer
```

Parameters:

PortNbr is the I/O address assigned to the BLC100 card (default = &H150).

ExtraResp is extra response time in msec per packet, used by the datalogger communication protocol. Extra response time is a user entered value. It can be as low as zero, but may need to go as high as 1000-2000 on noisy port connections.

maxPktSize is maximum packet size, used by the datalogger communication protocol. Maximum packet size is a user entered value and must be 32, 64, 128, 256, 512, 1024, or 2048. Start with 2048 and decrease only if communications problems are encountered.

BestPktSize specifies a best packet size to be used initially in communicating with the datalogger. Best packet size is then adjusted within the DLL each time communication takes place. It is reported back to the program through the **ClosePort( )** command and should be remembered for a given port and station between communications sessions. A good starting value is 512 or the maxPktSize, whichever is smaller.

Return Codes:

0 = Port was successfully opened.
1 = Port could not be opened: device failure.

Example:

```
Dim sPortName as String
Dim iPortNbr as Integer
Dim lExtraResp as Long
Dim iMaxPktSize as Integer
Dim iBestPktSize as Integer
Dim iRslt as Integer

sPortName = "&H150"
iPortNbr = CInt(sPortName)
lExtraResp = 500
iMaxPktSize = 2048
iBestPktSize = ClosePort()

iRslt= OpenCsiCard(iPortNbr, lExtraResp, iMaxPktSize, iBestPktSize)
If iRslt = 1 Then
   MsgBox "Tlink Port " & sPortName & " not available"
End If
```

2.1.4 OpenSock( ) - Open TCP/IP Port for use with NL100/NL105 Network Link Interface

The NL100/105 devices expose up to three separate ports for a given unit (CS I/O, RS-232, and TLink). Through these separate ports, simultaneous connections of different dataloggers to different computers can be facilitated.

Only one computer may connect to a specific port on a specific interface at any time, however. Once a connection is made to an IP Address and port from one computer, all attempted subsequent connections from other computers to that port will fail.

For Open Port functions to succeed, the only requirement is that a specific port be available. In the case of **OpenSock( )**, that means the presence of an available NL100/105 at the specified IP address, having its RS232, CS I/O, or Tlink specific port configured to the IP and port number specified in the function call,. If the address and port cannot be accessed (through whatever IP routes are available) this function will fail even though the computer's Ethernet interface is functioning properly. The datalogger does not have to be actively connected to the applicable NL100/105 interface until actual datalogger communications functions are invoked.

Declaration:

```
Declare Function OpenSock Lib "PC9000.DLL" (ByVal
ipAddr As String, ByVal IPPort As String, ByVal
ExtraResp As Long, ByVal maxPktSize As Integer,
ByVal BestPktSize As Integer) As Integer
```

Parameters:

ipAddr is the IP Address assigned to the NL100/105 during device configuration.

IPPort is the Port Address assigned to the specific port off of the NL100/105 to which the datalogger is attached: either the TLink port(CR9000) or the RS-232 or CS I/O ports(CR5000). Specific port addresses for each port off of the NL100/105 are also user-assigned during device configuration.

ExtraResp is extra response time in msec per packet, used by the datalogger communication protocol. Extra response time is a user entered value. It can be as low as zero, but may need to be as high as 15000 (15 seconds) or more on some remote connections.

maxPktSize is maximum packet size, used by the datalogger communication protocol. Maximum packet size is a user entered value and must be 32, 64, 128, 256, 512, 1024, or 2048. Start with 2048 and decrease only if communications problems are encountered.

BestPktSize specifies a best packet size to be used initially in communicating with the datalogger. Best packet size is then adjusted within the DLL each time communication takes place. It is reported back to the program through the **ClosePort( )** command and should be remembered for a given port and station between communications sessions. A good starting value is 512 or the maxPktSize, whichever is smaller.

Return Codes:

0 = Port was successfully opened.
1 = Port could not be opened: device failure, or already in use by this or other application.

Example:

```
Dim sIPAddress as String
Dim sPortAddress as Integer
Dim lExtraResp as Long
Dim iMaxPktSize as Integer
Dim iBestPktSize as Integer
Dim iRslt as Integer

sIPAddress = "192.168.6.46"
sPortAddress = "3000"
lExtraResp = 500
iMaxPktSize = 2048
iBestPktSize = ClosePort()

iRslt = OpenSock(sIPAddress, sPortAddress, sExtraResp, iMaxPktSize,
 iBestPktSize)
If iRslt = 1 Then
   MsgBox "NET Address:Port " & sIPAddress & ":" & sPortAddress _
    & " not available"
End If
```

2.1.5 GetModemStatus( ) - Retrieves Modem Control Register Values

This function is used in conjunction with COM port connections wherein modems are used to facilitate communication with remote CR5000 or CR9000 dataloggers. The function is used to retrieve current modem connection state information. In these cases, the **UserRd( )** and **UserWr( )** functions are utilized to send and receive ASCII character strings in order to control the modem functions.

GetModemStatus( ) is a simple wrapper around the Microsoft Windows API GetCommModemStatus( ) function, passing the control register mask exactly as returned by the operating system. The only reason that this function is necessary is that PC9000.DLL does not return operating system comm port handles, such as are needed to call the Windows API function directly.

Support for modem communications programming is beyond the scope of this manual or SDK, other than providing the necessary port access. Refer to MSDN documentation and/or other general programmer's resources for additional support information.

Declaration:

```
Declare Function GetModemStatus Lib "PC9000.DLL"
(ByVal StatusWord as Long) As Integer
```

Parameters:

StatusWord bit mask containing information about modem status lines:

MS_CTS_ON	&H10& (Bit 4)	The CTS (clear-to-send) signal is on.
MS_DSR_ON	&H20& (Bit 5)	The DSR (data-set-ready) signal is on.

MS_RING_ON	&H40& (Bit 6)	The ring indicator signal is on.
MS_RLSD_ON	&H80& (Bit 7)	The RLSD (receive-line-signal-detect) signal is on.

Return Codes:

0 = Completed successfully.
1 = Function failed (port not open, modem not found, or invalid port).

Example:

```
Sub SerialPortStatus(ByRef bCTS As Boolean, ByRef bDSR As Boolean, _
  ByRef bRING As Boolean, ByRef bCD As Boolean)

Const MS_CTS_ON = &H10&
Const MS_DSR_ON = &H20&
Const MS_RING_ON = &H40&
Const MS_RLSD_ON = &H80&

Dim lStatWord As Long
Dim iRes As Integer

   iRes = GetModemStatus(lStatWord)

   If iRes <> 0 Then
      MsgBox "Port not responding. "
   Else
      bCTS = lStatWord And MS_CTS_ON
      bDSR = lStatWord And MS_DSR_ON
      bRING = lStatWord And MS_RING_ON
      bCD = lStatWord And MS_RLSD_ON
   End If

   Exit Sub
```

2.1.6 ClosePort( ) - Close an I/O Port Opened by any one of the Open Port Instructions

Declaration:

```
Declare Function ClosePort Lib "PC9000.DLL" ( )
As Integer
```

Return Codes:

best packet size (default=512 if no port is presently open).

Example:

```
Dim BestPktSize as Integer
BestPktSize = ClosePort( )
```

2.2 Datalogger Utility Functions

2.2.1 GetLgrIdent ( ) - Gets the Datalogger Protocol Version, Model, Serial #, and Name

This function is most commonly used as the basic test to validate that a datalogger is connected and communicating with the DLL on the currently open port. Given its reporting of Model, Serial Number and station name, it is also a good way to verify that the application program is in fact communicating with the datalogger that it expects to find over a particular connection.

Declaration:

```
Declare Function GetLgrIdent Lib "PC9000.DLL"
(BmpVer As Integer, Model As Integer, SerNbr As
Long, ByVal StnName As String, ByVal StnNameSize
As Integer) As Integer
```

Parameters:

BmpVer returns the version number of the protocol version being used by the datalogger operating system.

Model returns the model: 0 = CR9000, 1 = CR5000

SerNbr returns the datalogger serial number.

StnName returns the station name (as set by **SetLgrName( )**

StnNameSize declares the size of the station name string buffer set up by the calling routine.

Return Codes:

0 = Completed successfully.
1 = Port not open or datalogger does not respond.

Example:

```
Dim iBmpVer As Integer
Dim iModel As Integer
Dim lSerNbr As Long
Dim sStnName As String
Dim sLoggerModelName as String
Dim iRslt as Integer

sStnName = String(20, vbNullChar)
iRslt = GetLgrIdent(iBmpVer, iModel, lSerNbr, sStnName, Len(sStnName))
If iRslt = 0 Then
   If iModel = 0 Then
      sLoggerModelName = "CR9000"
   ElseIf Model = 1 Then
      sLoggerModelName = "CR5000"
   Else
      sLoggerModelName = "Unknown"
   End If
End If
```

2.2.2 SetLgrName ( ) - Sets the Station Name of a Datalogger

The station name is a user-assigned name for a physical datalogger station. Once assigned, this name is saved in datalogger memory and is not lost if the datalogger is turned off.

The name can be up to eight characters long: Longer names will be truncated to 8 characters. Spaces are allowed but not recommended.

Always add a null termination character when passing strings to this function. Attempts to set the datalogger name to an empty string will be ignored, and passing an empty string without a null termination character may trigger a general protection fault.

The currently assigned station name can be retrieved either by using the **GetLgrIdent( )** function, or by querying the "StationName" field in the datalogger "Status" table using the **GetCurrentValue( )** function.

Declaration:

```
Declare Function SetLgrName Lib "PC9000.DLL"
(ByVal StnName As String) As Integer
```

Parameters:

StnName sets the station name of the datalogger.

Return Code:

0 = OK.
1 = Port not open or datalogger does not respond.

Example:

```
Dim sStnName as String
Dim iRslt as Integer

sStnName = txtStnName.text & vbNullChar
iRslt = SetLgrName(sStnName)
If iRslt = 1 Then
   MsgBox "Datalogger does not respond."
End If
```

2.2.3 SetLgrClock( ) - Set or Check the Datalogger Clock

The name of this function, together with its series of time arguments, can be a bit misleading. The function does both set and check the datalogger clock, but in each case the time arguments are only return arguments. In other words, when the function is called in the "set" mode, the function sets the datalogger time to the value of the PC's real time clock, regardless of the values passed in to the function. The time arguments returned in that case correspond to the PC's time as it was set in the datalogger.

In "get" mode, the time arguments correspond to the time as obtained from the datalogger, as would be expected.

Declaration:

```
Declare Function SetLgrClock Lib "PC9000.DLL"
(ByVal SetIt As Integer, DYear As Integer, DMonth
As Integer, DDay As Integer, DHour As Integer,
DMinute As Integer, DSecond As Integer) As
Integer
```

Parameters:

SetIt	if zero then the clock will be checked only. Otherwise it will be set to match the PC clock.
DYear	returns the year, as set or as read.
DMonth	returns the month number within the year, as set or as read.
DDay	returns the day number within the month, as set or as read.
DHour	returns the hour within the day (0-23), as set or as read.
DMinute	returns the minutes as set or as read.
DSecond	returns the seconds as set or as read.

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.

Example:

```
Function LoggerTime (bSet as Boolean) as Date
' This function gets or sets the datalogger clock, and
' returns the value in Microsoft standard date/time format.

Dim iYear As Integer
Dim iMonth As Integer
Dim iDay As Integer
Dim iHour As Integer
Dim iMin As Integer
Dim iSec As Integer
Dim iSet as Integer
Dim iRslt as Integer

iSet = IIf(bSet, 1 ,0)   ' Convert from Boolean to Integer logic

iRslt = SetLgrClock(bSet, iYear, iMonth, iDay, iHour, iMin, iSec)
If iRslt = 0 then
   LoggerTime = DateSerial(iYear, iMonth, iDay) + _
    TimeSerial(iHour, iMin, iSec)
Else
   Err.Raise 32000       ' User-defined error
End If

End Function
```

2.2.4 CR9000Dial( ) – Executes a Link Dial Transaction on the Currently Open Port; CR9000Hangup( ) – Executes a Link Termination Transaction on the Currently Open Port

These functions support two lesser-used communication link management transactions. The responses to these commands from any dataloggers or other devices connected to a currently open port are completely dependent on the particular model of datalogger connected to the port, and on any other intermediate communications devices.

In the case of a direct or phone modem connection to the CR5000, the Dial transaction has no effect. The Hangup transaction will cause the CR5000 to immediately close the port from its end, but only if the RS232 Timeout configuration setting is set to "Yes". (This setting is accessible from the front panel menu of the CR5000.)

In the case a direct or phone modem connection to the CR9000 these transactions support a datalogger configuration which rarely occurs: that is, the connection of multiple CR9000 dataloggers in a series chain, using a combination of TLink and Fiber Optic connections. In this case, each invocation of **CR9000Dial( )** passes control from the currently active datalogger, further down to the next datalogger in the series chain. Each invocation of **CR9000Hangup( )** passes control one station back up the chain. It is entirely up to the applications program to verify that the connection was passed successfully by testing communications with the dataloggers at each step of the sequence. If for any reason, one of the dataloggers in the link is not responding, the status of the connection is difficult to determine.

The Campbell Scientific Block Mode Protocol (BMP) transactions that are associated with these commands also have other application when working with radio communications links. These applications are beyond the scope of this document, and do not typically apply to logging applications using the CR9000 or CR5000.

These commands are NOT used in controlling dial-up telephone modems for remote dial-up connections, as might be expected by their names. Telephone modem control using the PC9000.DLL is implemented using the **UserRd( )** and **UserWr( )** functions.

With the possible exception of CR5000 link termination in certain configurations, these functions will generally not be applicable to writing custom datalogger applications code.

Declaration:

```
Declare Function CR9000Dial Lib "PC9000.DLL"
(ByVal DialString As String) As Integer

Declare Function CR9000HangUp Lib "PC9000.DLL" ()
As Integer
```

Parameters:

DialString defines a sequence of operations which must be executed to dial through the current remote to a new remote. Usually does not pertain when used with the CR5000 and CR9000 as described here.

Return Codes:

0 = OK.
1 = Transaction Failed.

Example:

None given.

2.2.5 StartIOLog ( ) - Start Logging Low Level I/O to a File; StopIOLog ( ) - Stop Logging Low Level I/O

These two functions are used to enable and disable a low-level logging function that logs all binary information passing back and forth at the protocol level between the DLL and the datalogger. It is primarily intended for DLL and protocol level debugging and generally is not useful in debugging applications unless a low-level protocol error is occurring.

The log files created by this function will become extremely large in a relatively short period of time, so this function should not be used unless absolutely necessary. The burden of continuously writing to the log files can also significantly slow communications throughput.

The logging function is not tied to a particular port or station, and can be turned on and off at any time. The currently open port may be shut down, changed, etc., while logging is enabled.

The DLL creates and appends its contents to a separate file for each of the ports to which it may be connected. The file names are as follows:

- IO$COM1.LOG, IO$COM2.LOG… COM1, COM2…respectively
- LOG$.LOG NET ports – (single file for all NET connections)
- IO$PAR4.LOG LPT ports (low level logging not supported under Windows NT 4.0 / 2000)
- IO$TPTR.LOG TLink ports

Declarations:

```
Declare Function StartIOLog Lib "PC9000.DLL" ( )
As Integer

Declare Function StopIOLog Lib "PC9000.DLL" ( )
As Integer
```

Return Codes:

Always returns 0.

Example:

```
' Module-level variable
Private mbLowLevelLog as Boolean

Sub LowLevelLog (ByVal bLog as Boolean)

Dim iRslt as Integer

   If bLog <> mbLowLevelLog Then
      If bLog Then
         iRslt = StartIOLog()
         mbLowLevelLog = True
      Else
         iRslt = StopIOLog()
         mbLowLevelLog = False
      End If
   End If

End Sub
```

2.2.6 UserWr( ) - Write the Specified ASCII Character String Directly to the Port

UserWr( ) is intended to be used with its companion function, **UserRd( )**. Together these functions are used in one of two modes:

- To communicate directly with modems, typically on a RS-232 serial port previously opened by the **OpenCom( )** function, in order to manage remote dial-up connections. The details of this process are modem-specific and are beyond the scope of this document.

- To communicate with dataloggers in a mode known as "low-level I/O" or "terminal" mode, sending individual ASCII character commands, and receiving various status and diagnostic information in return, in ASCII carriage-return/line feed terminated format. This mode is not limited to RS-232 connections; it may be used for all port types.

  The details of terminal mode are not covered here in detail. In general, using this mode consists of sending 2-3 individual carriage return characters to "wake-up" the terminal mode connection, evidenced by the return of a CR9000> OR CR5000> prompt string retrieved by the **UserRd( )** function. After receiving the prompt, a single "h" character with a carriage return will cause a menu of command choices to be transmitted from the datalogger.

Be aware that this terminal mode ASCII communications protocol is not the protocol used by other DLL functions for executing datalogger transactions and receiving data. It is simply a way of monitoring datalogger behavior independently of the normal protocols. Further, this mode of communications with the datalogger should only be used when all normal datalogger communications routines are completely idle, else data transmission errors and timeouts may occur.

UserWr( ) and **UserRd( )** functions read and write all characters directly to and from ports, with no parsing or overhead. They can be used anywhere that one would use Windows HyperTerminal or similar programs to communicate with a serial device through ASCII command sequences. It is left entirely up to the application to be aware of the current state of the port connection (modem on or off-line) in order to send the right commands and properly interpret the responses.

The **UserWr( )** and **UserRd( )** functions require an open port. A valid on-line datalogger connection is not monitored or reported as with all other communications commands, however.

Declaration:

```
Declare Function UserWr Lib "PC9000.DLL" (ByVal
Buf As String, ByVal BufSize As Integer) As
Integer
```

Parameters:

Buf a string to write to the port. Any desired termination characters (carriage returns, line feed, EOF characters) must be included in the passed string as none are added by the function.

BufSize the length of Buf.

Return Codes:

0 = Bytes were sent
1 = No byes were sent.

Example:

```
Function WriteChars(byVal sChars as String) as Long

Dim sBuf as String

   sBuf = sChars & vbNullChar
   WriteChars = UserWr(sBuf, Len(sBuf))

End Function
```

2.2.7 UserRd ( ) - Returns up to the Allocated Number of Characters in a Port's Input Buffer

UserRd( ) is intended to be used with its companion function, **UserWr( )**. Together these functions are used in one of two modes:

- To communicate directly with modems, typically on a RS-232 serial port previously opened by the **OpenCom( )** function, in order to manage remote dial-up connections. The specifics of this process are modem-specific and are beyond the scope of this document and SDK.
- To communicate with dataloggers in a mode known as "low-level I/O" or "terminal" mode, sending individual ASCII character commands, and receiving various status and diagnostic information in return, in ASCII carriage-return/line feed terminated format. This mode is not limited to RS-232 connections: it may be used for all port types.

The details of terminal mode are not covered here in detail. In general, using this mode consists of sending 2-3 individual carriage return characters to "wake-up" the terminal mode connection, evidenced by the return of a CR9000> OR CR5000> prompt string retrieved by the **UserRd( )** function. After receiving the prompt, a single "h" character with a carriage return will cause a menu of command choices to be transmitted from the datalogger.

Be aware that this terminal mode ASCII communications protocol is not the protocol used by other DLL functions for executing datalogger transactions and receiving data. It is simply a way of monitoring datalogger behavior independently of the normal protocols. Further, this mode of communications with the datalogger should only be used when all normal datalogger communications routines are completely idle, else data transmission errors and timeouts will occur.

UserWr( ) and **UserRd( )** functions read and write all characters directly to and from ports, with no parsing or overhead. They can be used anywhere that one would use Windows HyperTerminal or similar programs to communicate with a serial device through ASCII command sequences. It is left entirely up to the application to be aware of the current state of the port connection (modem on or off-line) in order to send the right commands and properly interpret the responses.

The **UserWr( )** and **UserRd( )** functions require an open port. A valid on-line datalogger connection is not monitored or reported as with all other communications commands, however.

Declaration:

```
Declare Function UserRd Lib "PC9000.DLL" (ByVal
Buf As String, ByVal BufSize As Integer) As
Integer
```

Parameters:

Buf string buffer in which to place the current input characters received.

This buffer can be sized (within reason) to any size that is convenient to the application, however no bytes will be returned by this function until there are enough bytes to completely fill the port. All characters returned in the buffer will be automatically cleared from the input port.

If there are more characters available in the port's input buffer than will fit in Buf, the remaining characters may be picked up in subsequent function calls: they will not be cleared until they have been returned in a call.

BufSize the length of Buf.

Return Codes:

Number of characters read in and placed in string buffer Buf.

Example:

```
Function ReadChars( ) as String
' This function would normally be called by some periodic timer
' routine that is monitoring the port during terminal mode
' communications sequences

Dim sBuf as String
Dim iRslt as Integer

   Do
      sBuf = String(10, vbNullChar)
      iRslt = UserRd(sBuf, Len(sBuf))
      If iRslt > 0 Then ReadChars = ReadChars & Left$(sBuf, iRslt)

   Loop Until iRslt = 0   ' Or Len(ReadChars) > some limit

End Function
```

2.3 Datalogger File/Directory Functions

2.3.1 BootFromLinkStart( ) - Start the Cold Boot Datalogger Transaction

The boot from link transaction is executed by calling **BootFromLinkStart( ),** then **BootFromLinkMore( )** is repeated in a loop until the boot transaction is finished. This will reset the BLC100 card, if one is in use, before download. This may, depending on the position of a jumper in the CR9000, reset the CR9000.

Consult the PC9032 program and help files for descriptions of the full sequence of steps to take when booting a CR9000 or CR5000 from a link.

Declaration:

```
Declare Function BootFromLinkStart Lib
"PC9000.DLL" (ByVal OSName As String, NbrWr As
Long) As Integer
```

Parameters:

OSName operating system name.

NbrWr indicates how many bytes of the operating system have been transmitted to the datalogger.

Return Codes:

1 = Port not open or datalogger does not respond.
2 = Boot transaction started.
3 = Bad file name.

Example:

```
Dim iRslt as Integer
Dim sOsName As String
Dim lNbrWr As Long

sOsName = App.Path & "\OS.RUN"
iRslt = BootFromLinkStart(sOsName, lNbrWr)

Select Case iRslt
Case 1
   Err.Raise 32000, , "Datalogger does not respond."
Case 3
   Err.Raise 32001, , "Bad file name."
End Select

Do
   iRslt = BootFromLinkMore(NbrWr)
   If iRslt = 1 Err.Raise 32000, , "Datalogger does not respond."
   ‘ iRslt will be = 2 if the transaction is still in progress
Loop Until iRslt = 0
```

2.3.2 BootFromLinkMore( ) - Continue with Cold Boot from Link Transaction

The boot from link transaction is executed by calling **BootFromLinkStart( ),** then **BootFromLinkMore( )** is repeated in a loop until the boot transaction is finished. This will reset the BLC100 card, if one is in use, before download. This may, depending on the position of a jumper in the CR9000, reset the CR9000.

Consult the PC9032 program and help files for descriptions of the full sequence of steps to take when booting a CR9000 or CR5000 from a link.

Declaration:

```
Declare Function BootFromLinkMore Lib
"PC9000.DLL" (NbrWr As Long) As Integer
```

Parameters:

NbrWr indicates how many bytes of the operating system have been transmitted to the datalogger.

Return Codes:

0 = Operation complete.
1 = Port not open or datalogger does not respond.
2 = Not done yet.

Example:

```
Dim iRslt as Integer
Dim sOsName As String
Dim lNbrWr As Long

sOsName = App.Path & "\OS.RUN"
iRslt = BootFromLinkStart(sOsName, lNbrWr)

Select Case iRslt
Case 1
   Err.Raise 32000, , "Datalogger does not respond."
Case 3
   Err.Raise 32001, , "Bad file name."
End Select

Do
   iRslt = BootFromLinkMore(NbrWr)
   If iRslt = 1 Err.Raise 32000, , "Datalogger does not respond."
   ' iRslt will be = 2 if the transaction is still in progress
Loop Until iRslt = 0
```

2.3.3 GetDirectory( ) - Get a Directory of Files (programs) that are in Datalogger

GetDirectory( ) is designed to be used in a loop. Before entering the loop, set the FileName argument to an empty string buffer. Call **GetDirectory( )** to get the first file name, then pass the first name back in to the function to get the next name, and so on to get all file names. An empty string and a result code of 4 is returned when the all table names have been returned.

The file names are returned in the format
DeviceName:FileName

Where the device name will either be "CPU", "P4A", P4B" or "CRD", depending on the datalogger model and the auxiliary memory storage devices available. The file name will be the name of the file in standard format, such as "BRIDGE1.CR9".

For each existing device that is currently empty (i.e., contains no files), a single file directory entry will be returned to that effect, in one of the following formats:

CR9000:

P4A:NO_CARD	(Datalogger has a 9080 card but no memory card is in slot A)
P4B:NO FILES	(Datalogger has a memory card but it is empty)

CR5000:

CRD:	(Datalogger's card slot either has no card or has a card with no files)

Declaration:

```
Declare Function GetDirectory Lib "PC9000.DLL"
(ByVal FileName As String, ByVal FileNameSize As
Integer, Attrib As Integer) As Integer
```

Parameters:

FileName a string buffer in which the next file name is returned by the function.

FileNameSize the length of the FileName string buffer.

Attribute returns the file attributes in a bit-masked integer:
bit 0: not used.
bit 1: 1 = execute on power-up. (Integer Value 2)
bit 2: 1 = executing now. (Integer Value 4)
other bits not used.

Return Codes:

0 = Next directory entry returned.
1 = Port not open or datalogger does not respond.
4 = End of list.

Example:

```
Dim sFileName as String
Dim iAttr as Integer
Dim iRslt as Integer
Dim sText as String

‘lstDir is a VB List Box
lstDir.Clear
sFileName = String(40, vbNullChar)

Do
   iRslt = GetDirectory (sFileName, Len(sFileName), iAttr)
   If iRslt = 1 Then Err.Raise 32000   ‘User-defined error code
   If iRslt = 0 Then
      sText = Trim$(sFileName)
      If (iAttr And 4) > 0 Then
         sText = sText & vbTab & "Running"
      End If
      lstDir.AddItem sText
   End If
Loop Until (iRslt = 4) Or (Len(Trim$(sFileName)) = 0)
```

2.3.4 DownloadStart( ) - Initiate Downloading and Other File Management Options

DownloadStart( ) is used for a number of tasks including downloading files to the datalogger, starting and stopping programs, and operating on files that are already in the datalogger memory.

DownloadStart( ) is used for the following types of operations:

- Downloading a CRBasic file to the datalogger, and compiling and running the file
- Modifying the run attributes of a CRBasic program file already in datalogger memory
- Stopping the current program

- Deleting a file, a group of files, or all files, from a datalogger storage device

- Downloading a non-program file (data file, wiring diagram, etc.) to datalogger memory

- Downloading a new datalogger operating system file to replace the existing operating system

DownloadWait( ) is normally used in conjunction with **DownloadStart( )**, to monitor the progress of the specified transaction until it is complete.

The examples section of this function contains a set of procedures, one for each of the function's main operating modes described above. This is perhaps the most complex function in the DLL. There are many possible combinations of function arguments that are not allowable or nonsensical. Examples would be: attempting to compile a non-CRBasic file, downloading a file with the attribute set to 8 or 16, passing the name of a datalogger file when SendFile specifies a disk file (or vice versa). Also, the procedure to stop the currently running CRBasic program is not what one would intuitively expect.

Be careful to stay within the bounds described and illustrated here for this function. Study the PC9000Test program example code that is included on the PC9000.DLL developer kit disk for further clues about working with this function successfully. Failure to stay within the recommended bounds may result in Windows General Protection Faults, or may crash the datalogger operating system or leave it in an uncertain state. Not every possible error combination will be trapped by the function.

When **DownloadStart( )** is used to download files from the computer (as opposed to operating on files already in the datalogger), the datalogger will automatically overwrite any existing file already existing on the specified storage device having the same name as the file being downloaded.

Programs may be downloaded, compiled and activated, with one **DownloadStart( )/Wait( )** sequence.

Deleting the running CRBasic file from memory will not stop it from running, but once stopped, it will not be able to be re-started until the program file is downloaded again.

When **DownloadStart( )** is used to compile a CRBasic Program file (applies both to new downloads and existing datalogger files), the **DownloadWait( )** instruction is used to evaluate the compile results. See the reference documentation for that instruction, for more information.

In order to simplify the purging of data files from a device, the "*.DAT" file specification is recognized for the delete file function only (Option = 8). All other times, a single explicit file name is required.

Don’t assume that the function will always cleanly protect against the passing of a bad file or device name. The end result will often be an unsuccessful compile attempt, stopping the datalogger.

Downloading an operating system (OS) is similar to downloading other files. Key items of note are:

1) Any previously running program must have been previously stopped

2) The OS File's extension is always ".OBJ"

3) The attribute bit must be 0

4) The device name must be "CPU:"

5) The **DownloadWait()** instruction may return 7 instead of 0 when the download is complete, if the OS compile time exceeds communications timeout settings. If so, the downloading routine should fall into a "wait and retry" datalogger status check loop before proceeding.

6) Downloading a new OS should not cause datalogger files to be deleted, other than data files currently in use. These should be backed up prior to downloading a new operating system file.

Declaration:

```
Declare Function DownloadStart Lib "PC9000.DLL"
(ByVal FileName As String, ByVal DevName As
String, ByVal Options As Integer, ByVal SendFile
As Integer) As Integer
```

Parameters:

FileName the name of the file on which to operate:

If the SendFile argument is set to zero, (file exists in datalogger) this file name must exactly match the file name as reported by the **GetDirectory( )** command, minus the device specifier and delimiter (which is instead specified in the device name argument).

If the SendFile argument is set to 1, (new file download) this parameter must contain the explicit Drive:\path\file specification for the file as it is located on the computer.

IMPORTANT: The dataloggers only accept filenames in DOS standard 8.3 format. Programs that call this function must pre-convert any non-conforming file specification to this legacy "short path\8.3" format. (This applies not just to the file name, but to the entire path\file specification.) See the example code for illustration of how this can be accomplished.

DevName the name of the device in the datalogger where the file is located (existing datalogger file) or will be written to (new file download). These device names must be specified literally as shown below, <u>including</u> the delimiting colon.

Recognized Device Names for the CR9000:
"CPU:" - Main datalogger flash memory
"P4A:" - CR9080 memory card A
"P4B:" - CR9080 memory card B

Recognized Device Names for the CR5000:
"CPU:" - Main datalogger flash memory
"CRD:" - CR5000 memory card

Options the transaction options bit-masked flags (with corresponding integer values):

bit 0 (1): not used at this time.
bit 1 (2): 1 = make the named program the one to execute on power-up.
bit 2 (4): 1 = compile and execute the named program now.
bit 3 (8): 1 = delete the named file (or group of files, according to wildcard characters).
bit 4 (16): 1 = format the specified device (will delete all files).
Bits 5-15: not used at this time.

Note that, while the numerical values for the option settings comprise a mask, only bits 1 and 2 (Integer 2 + Integer 4 = 6) may be set together in one call to this function. All of the rest of the mask settings must be used by themselves in separate calls.

SendFile the send file option:

1 = send the specified file from the specified computer disk drive

0 = do not send the file (assumes file already exists in datalogger memory).

Return Codes:

1 = Port not open or datalogger does not respond.
2 = Download started.
3 = Bad file name: Typically this would mean a bad PC File name, if applicable to the current usage of the function. Bad datalogger file device names if detected, will not likely be reported until some time during the **DownloadWait( )** loop.

Examples:

The first example shows a function that will work on a file already existing in the datalogger, supporting attribute bits 1, 2 or 3, and all allowable combinations of those bits. It takes a complete datalogger file specification (DEV:FILE, as originally passed by the **GetDirectory( )** function) and parses it as required for the DLL **DownloadStart( )** function. Also recognizes the special case of stopping the current program (explained in more detail immediately following this example)

```
' Module level declarations
Private Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As
Long)

Private Function SetFileAttribute(ByVal sFileSpec As String, _
  ByVal iAttrib As Integer) As Long

Dim sResult As String
Dim iYear As Integer
Dim iMonth As Integer
Dim iDay As Integer
Dim iHour As Integer
Dim iMin As Integer
Dim iSec  As Integer

Dim iRslt As Integer
Dim sMsg As String
Dim bStopProg As Boolean
Dim sDevName As String
Dim sFileName As String
Dim lMrk As Long

On Error GoTo Err_FileAttr

   ' Only allows these specific combinations of attribute settings.
   ' (Attribute bit 4 is handled by a separate example function)
   If Not (iAttrib = 0 Or iAttrib = 2 Or iAttrib = 4 Or iAttrib = 6 Or
_ iAttrib = 8)  Then Err.Raise 32000

   ' Parse the device and file names from the file specification
   If Len(sFileSpec) > 0 Then
      lMrk = InStr(1, sFileSpec, ":")
      If lMrk > 0 And Len(sFileSpec) > 1 Then
         sDevName = Left$(sFileSpec, lMrk) & vbNullChar
         sFileName = Mid$(sFileSpec, lMrk + 1) & vbNullChar
      Else
         sDevName = "CPU:" & vbNullChar
         sFileName = sFileSpec & vbNullChar
      End If
```

(Example continues on next page)

SetFileAttribute Example (continued from previous page)

```
   Else
      sDevName = vbNullChar
      sFileName = vbNullChar
      If iAttrib = 4 Then bStopProg = True
   End If ' Begin file attribute set operation
   iRslt = DownloadStart(sFileName, sDevName, iAttrib, 0)
   If iRslt = 1 Then Err.Raise 32001
   If iRslt = 3 Then Err.Raise 32002  'Bad File Name

   ' Wait loop
   Sleep 200
   sResult = String(512, vbNullChar)
   Do
      iRslt = DownloadWait(sResult, Len(sResult), iYear, iMonth, iDay,
_
       iHour, iMin, iSec)
      If iRslt = 1 Then Err.Raise 32001
      If iRslt = 4 then Err.Raise 32003
      If iRslt = 5 then Err.Raise 32004
      If iRslt = 6 then Err.Raise 32002
   Loop Until iRslt <> 2

   ' Interpret results, in light of the requested operation
   If iRslt = 0 Then
' Operation commenced successfully
      SetFileAttribute = True
      If bStopProg Then
         sMsg = "Currently running program was stopped."
      Else
         ' Strip the null termination characters off of the device and
         ' file names, to use them in a message string.
         sDevName = Left$(sDevName, Len(sDevName) - 1)
         sFileName = Left$(sFileName, Len(sFileName) - 1)

         If iAttrib <> 8 Then
            sMsg = "Attribute for File " & sDevName & sFileName & _
             " was set to " & iAttrib & "."
         Else
            sMsg = "File " & sDevName & sFileName & " was deleted."
         End If
      End If
   Else
      sMsg = "Unknown error code."
   End If

   If iAttrib = 4 Or iAttrib = 6 Then
      ' A compile was just attempted.
      sMsg = sMsg & vbCrLf & "COMPILE RESULTS: "

& Trim$(sResult)
   End If
   lblStatus.Caption = sMsg
```

(Example continues on next page)

SetFileAttribute Example (continued from previous page)

```
Exit_FileAttr:
   Exit Function

Err_FileAttr:
   Select Case Err.Number
      Case 32000
         sMsg = "Invalid File Attribute."
      Case 32001
         sMsg = "Port not open or datalogger does not respond."
      Case 32002
         sMsg = "Invalid File Name."
      Case 32003
         sMsg = "Insufficient resources to complete file operation."
      Case 32004
         sMsg = "Access Denied."
      Case Else
         sMsg = Err.Description
   End Select
MsgBox sMsg
   Resume Exit_FileAttr

End Function
```

The currently running program cannot be stopped by setting the attribute for that file to zero, as one might presume. Rather, a "null" program must be started. Using the above SetFileAttribute function to accomplish this function, as follows:

```
Call SetFileAttribute "", 4
```

NOTES:
Stopping the currently running program as outlined above will effectively reset that program's bit 2 attribute (run now). It will do nothing to the bit 1 (run on power up attribute), however. Therefore, resetting both attributes of a file whose bit 1 and bit 2 attributes are set requires two operations: first, stopping the current program, and then setting the file attribute from 2 to zero.

Stopping the current program is only required when a new operating system is downloaded. A new file can be started or downloaded, or the currently running program deleted, without first stopping the current program from running. It is good practice, however, to stop the current program first when doing other file operations affecting the current program, to avoid unwanted side effects from sudden termination of the current program.

The next example shows a function which can download new files to disk, and also compile and run them as desired. It receives a complete PC Disk path/file specification in long file name format, and converts it to DOS short path name format as required for the DLL **DownloadStart( )** function.

```
' Module level declarations
Private Declare Function GetShortPathName Lib "kernel32" _
 Alias "GetShortPathNameA" (ByVal lpszLongPath As String, _
 ByVal lpszShortPath As String, ByVal cchBuffer As Long) As Long

Private Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

Private mbCancel As Boolean


Public Function DownloadFile(ByVal sFileSpec As String, _
  ByVal sDevName As String, ByVal iAttrib As Integer) As Long

Dim sResult As String
Dim iYear As Integer
Dim iMonth As Integer
Dim iDay As Integer
Dim iHour As Integer
Dim iMin As Integer
Dim iSec  As Integer

Dim iRslt As Integer
Dim sMsg As String
Dim sFileName As String
Dim lMrk As String
Dim lPath As Long

On Error GoTo Err_Download

   ' These are the only attribute settings allowed
   ' for a file download operation.
   If Not (iAttrib = 0 Or iAttrib = 2 Or iAttrib = 4 Or iAttrib = 6) _
    Then Err.Raise 32000

   ' This is a Windows API function call, to convert
   ' 32-bit long path/file names to the DOS 8.3 compatible format,
   ' required by the DownloadStart() function.
   sFileName = String(255, vbNullChar)
   lPath = GetShortPathName(sFileSpec, sFileName, Len(sFileName))
   sFileName = Left(sFileName, lPath)

   lblStatus.Caption = "Downloading File " & sFileName & _
    " into " & sDevName & " ..."

   ' This initiates the file transfer.  Normal result will be a code 2.
   iRslt = DownloadStart(sFileName, sDevName, iAttrib, 1)
   If iRslt = 1 Then Err.Raise 32001
   If iRslt = 3 Then Err.Raise 32002   'Bad File Name
' Wait loop
   ' mbCancel is a module-level flag, which would be set true by
   ' some other event procedure, designated by the programmer as
   ' as means to allow the cancel of large file downloads.
   mbCancel = False
```

(Example continues on next page)

DownloadFile() Example (continued from previous page)

```
Sleep 200
   sResult = String(512, vbNullChar)
   Do
      iRslt = DownloadWait(sResult, Len(sResult), iYear, iMonth, _
       iDay, iHour, iMin, iSec)
      If iRslt = 1 Then Err.Raise 32001
      If iRslt = 4 Then Err.Raise 32003
      If iRslt = 5 Then Err.Raise 32004
      If iRslt = 6 Then Err.Raise 32002   'Bad File Name

      ' This allows event procedures to run, in order to detect
      ' user cancel requests.
      DoEvents

      ' Return code of 2 means "not finished yet."
      ' Other return codes are processed immediately below.
   Loop Until (iRslt <> 2) Or mbCancel
   '
   If mbCancel Then
      sMsg = "Download cancelled by user."
   Else
      If iRslt = 0 Then
         DownloadFile = True
         sMsg = "File " & sDevName & sFileName & " was downloaded."
      Else
         sMsg = "Unknown Error Code."
      End If
      If iAttrib = 4 or iAttrib = 6 Then
         sMsg = sMsg & vbCrLf & "COMPILE RESULTS: " & Trim$(sResult)
      End If
   End If
   lblStatus.Caption = sMsg

Exit_Download:
   Exit Function
Err_Download:
   Select Case Err.Number
      Case 32000
         sMsg = "Invalid File Attribute."
      Case 32001
         sMsg = "Port not open or datalogger does not respond."
      Case 32002
         sMsg = "Invalid File Name."
      Case 32003
         sMsg = "Insufficient resources to complete file operation."
      Case 32004
         sMsg = "Access denied."
      Case Else
         sMsg = Err.Description
End Select
   MsgBox sMsg
   lblStatus.Caption = "File download was terminated due to errors."
   Resume Exit_Download

End Function
```

The next example shows a special case, used to delete all files from the specified storage device and format the device.

```
Private Function FormatLoggerDevice(ByVal sDevName As String)

Dim sResult As String
Dim iYear As Integer
Dim iMonth As Integer
Dim iDay As Integer
Dim iHour As Integer
Dim iMin As Integer
Dim iSec  As Integer

Dim iRslt As Integer
Dim sMsg As String
Dim sFileName As String

On Error GoTo Err_Format

   sFileName = vbNullChar
   sDevName = sDevName & vbNullChar

   ' Begin device format operation
   iRslt = DownloadStart(sFileName, sDevName, 16, 0)
   If iRslt = 1 Then Err.Raise 32001
   If iRslt = 3 Then Err.Raise 32002   ' Bad File Name (should never occur
                                       ' in this calling mode)
   Sleep 200
   sResult = String(512, vbNullChar)
   Do
      iRslt = DownloadWait(sResult, Len(sResult), iYear, iMonth, _
        iDay, iHour, iMin, iSec)
      If iRslt = 1 Then Err.Raise 32001
      If iRslt = 4 Then Err.Raise 32003
      If iRslt = 5 Then Err.Raise 32004
      If iRslt = 6 Then Err.Raise 32002
   Loop Until iRslt <> 2

   If iRslt = 0 Then
      FormatLoggerDevice = True
      sMsg = "Device " & sDevName & " was formatted."
   Else
      sMsg = "Unknown error code."
   End If
   lblStatus.Caption = sMsg

Exit_Format:
   Exit Function
```

(Example continues on next page)

FormatLoggerDevice() Example (continued from previous page)

```
Err_Format:
   Select Case Err.Number
      Case 32001
         sMsg = "Port not open or datalogger does not respond."
      Case 32002
         sMsg = "Invalid File Name."
      Case 32003
         sMsg = "Insufficient resources to complete file operation."
      Case 32004
         sMsg = "Access Denied."
      Case Else
         sMsg = Err.Description
   End Select
   MsgBox sMsg
   Resume Exit_Format

End Function
```

2.3.5 DownloadWait( ) - Monitor Status of Download Operation in Progress

DownloadWait( ) is only used in a loop, following a call to **DownloadStart( )**, to monitor the progress of the specified transaction until it is complete.

Depending upon the type of operation being performed by **DownloadStart( )/ DownloadWait( ),** bad file names may either be detected immediately during the **DownloadStart( )** call**,** or sometimes not until after **DownloadWait( )** is invoked. Well-written error-handling code will monitor both functions for bad file name conditions.

Declaration:

```
Declare Function DownloadWait Lib "PC9000.DLL"
(ByVal Result As String, ByVal ResultSize As
Integer, DYear As Integer, DMonth As Integer,
DDay As Integer, DHour As Integer, DMinute As
Integer, DSecond As Integer) As Integer
```

Parameters:

Result points to a string buffer where the compile result is returned when the download is done. The compile result will be a sequence of LF-delimited ASCII strings. Compile results will return empty if no program was compiled.

ResultSize length of the Result string buffer. Should be at least 256 bytes long.

DYear, DMonth, DDay, DHour, DMinute, DSecond
Time parameters indicate when the named program was compiled. These fields are returned with invalid values if no compile was done.

Return Codes:

0 = Operation complete.
1 = Port not open or datalogger does not respond.
2 = Not finished.
4 = Insufficient resources (will occur if file is larger than available space remaining on storage device)
5 = Permission denied.
6 = Bad file name or invalid file (reported primarily for operating system OS*.OBJ files)
7 = OS Download complete (all packets successfully sent), but function timed out waiting for a datalogger response, probably because the datalogger was busy compiling the new OS file.

Example:

See examples for **DownloadStart( )**.

2.3.6 UploadFile( ) - Upload Program or Data File from the Datalogger with One Command

UploadFile( ) is used to retrieve files from the datalogger. It performs the same function as the combination of **UploadStart( )** and **UploadWait( )**, in one instruction. It is therefore simpler to use but does not provide the means to program a wait loop which provides progress updates to the user screen and/or allows cancel options.

Declaration:

```
Declare Function UploadFile Lib "PC9000.DLL"
(ByVal FileName As String, ByVal DestFileName As
String, ByVal DevName As String) As Integer
```

Parameters:

FileName the name of the file to upload.

DestFileName the explicit Drive:\path\filename specification for the file as it is to be saved on the computer. Can be in long path/file name format if desired.

DevName the name of the device in the datalogger where file is located (existing datalogger file) or will be written to (new file download). These device names must be specified literally as shown below, including the delimiting colon.

Recognized Device Names for the CR9000:
"CPU:" - Main datalogger flash memory
"P4A:" - CR9080 memory card A
"P4B:" - CR9080 memory card B

Recognized Device Names for the CR5000:
"CPU:" - Main datalogger flash memory
"CRD:" - CR5000 memory card

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.
3 = Bad file name.

Example:

```
Private Function UploadLoggerFile(ByVal sLoggerFile As String, _
  ByVal sDestFile As String, ByVal sDevName As String) As Long

Dim iRslt As Integer
Dim sMsg As String

On Error GoTo Err_Upload

   lblStatus.Caption = "Retrieving File " & sDevName & sLoggerFile

   iRslt = UploadFile(sLoggerFile, sDestFile, sDevName)
   If iRslt = 1 Then Err.Raise 32001
   If iRslt = 3 Then Err.Raise 32002

   UploadLoggerFile = True
   lblStatus.Caption = "File upload complete."

Exit_Upload:
   Exit Function

Err_Upload:
   Select Case Err.Number
      Case 32001
         sMsg = "Port not open or datalogger does not respond."
      Case 32002
         sMsg = "Invalid file name."
   End Select
   MsgBox sMsg
   lblStatus.Caption = "File upload terminated due to errors."
   Resume Exit_Upload

End Function
```

2.3.7 UploadStart( ) - Start File Upload from the Datalogger Using a Progress Loop

UploadStart( ) is used to initiate file uploads in cases where a progress loop with cancel options is desired.

UploadWait( ) and **UploadStop( )** are normally used in conjunction with **UploadStart( )**, to monitor the progress of the specified transaction until it is complete.

UploadStart( ) has an identical parameter list to **UploadFile( ).** The return codes are slightly different, reflecting the differences in the ways that these two functions are used.

Declaration:

```
Declare Function UploadStart Lib "PC9000.DLL"
(ByVal FileName As String, ByVal DestFileName As
String, ByVal DevName As String) As Integer
```

Parameters:

FileName the name of the file to upload.

DestFileName the explicit Drive:\path\filename specification for the file as it is to be saved on the computer. Can be in long path/file name format if desired.

DevName the name of the device in the datalogger where file is located (existing datalogger file) or will be written to (new file download). These device names must be specified literally as shown below, including the delimiting colon.

Recognized Device Names for the CR9000:
"CPU:" - Main datalogger flash memory
"P4A:" - CR9080 memory card A
"P4B:" - CR9080 memory card B

Recognized Device Names for the CR5000:
"CPU:" - Main datalogger flash memory
"CRD:" - CR5000 memory card

Return Codes:

1 = Port not open or datalogger does not respond.
2 = Started.
3 = Bad destination file path name.

Example:

```
' Module level declarations
Private Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

Private mbCancel As Boolean

Private Function UploadFileWithLoop(ByVal sLoggerFile As String, _
  ByVal sDestFile As String, ByVal sDevName As String) As Long

Dim iRslt As Integer
Dim lBytes As Long
Dim sMsg As String

On Error GoTo Err_Upload

   mbCancel = False
   iRslt = UploadStart(sLoggerFile, sDestFile, sDevName)
   If iRslt = 1 Then Err.Raise 32001
   If iRslt = 3 Then Err.Raise 32002
```

(Example continues on next page)

UploadFileWithLoop() Example (continued from previous page)

```
   ' Wait Loop
   ' mbCancel is a module-level flag, which would be set true by
   ' some other event procedure, designated by the programmer as
   ' as means to allow the cancel of large file downloads.
   mbCancel = False

   Sleep 200
   Do
      DoEvents
      If mbCancel Then
         UploadStop
         Exit Do
      End If

      'iRslt = UploadWait(lBytes)
      If iRslt = 1 Then Err.Raise 32001

      lblStatus.Caption = "Retrieving File " & sDevName & sLoggerFile _
       & ":  " & lBytes & " bytes sent.  Press 'ESC' to cancel."

   Loop Until iRslt <> 2

   UploadFileWithLoop = True
   If mbCancel Then
      lblStatus.Caption = "File upload cancelled by user."
   Else
      lblStatus.Caption = "File upload complete."
   End If


Exit_Upload:
   mbCancel = False
   Exit Function

Err_Upload:
   Select Case Err.Number
      Case 32001
         sMsg = "Port not open or datalogger does not respond."
      Case 32002
         sMsg = "Invalid file name."
      Case Else
         sMsg = Err.Description
   End Select
   MsgBox sMsg
   lblStatus.Caption = "File upload terminated due to errors."
   Resume Exit_Upload

End Function
```

2.3.8 UploadWait( ) - Monitor Status of Upload Operation in Progress

UploadWait( ) is only used in a loop, following a call to **UploadStart( )**, to monitor the progress of the specified transaction until it is complete.

A bad destination file path specification will be determined immediately upon calling **UploadStart( )**, when it attempts to open the output file. Bad datalogger file names, on the other hand, will not be detected until after **UploadWait( )** is invoked. Well-written error-handling code will monitor both functions for bad file name conditions.

If a file upload is terminated prematurely by the application (usually due to user input) the **UploadStop( )** function should be invoked.

Declaration:

```
Declare Function UploadWait Lib "PC9000.DLL"
(BytesSent As Long) As Integer
```

Parameters:

BytesSent the number of bytes retrieved thus far.

Return Codes:

0 = Operation complete.
1 = Port not open or datalogger does not respond.
2 = Not finished.
3 = Bad datalogger file name

Example:

See examples for **UploadStart( )**.

2.3.9 UploadStop( ) - Terminate Upload Operation in Progress

UploadStop( ) is only used in conjunction with a loop involving **UploadStart( )**. It only needs to be used if the application determines that the current upload operation should be terminated. It does not need to be used when the DLL itself returns an error code and stops the upload operation.

Declaration:

```
Declare Function UploadStop Lib "PC9000.DLL" ()
As Integer
```

Return Codes:

Always returns 0.

Example:

See examples for **UploadStart( )**.

2.4 Datalogger Table Management Functions

2.4.1 GetTableName( ) - Retrieves the Names and Sizes of All Available Datalogger Tables

GetTableName( ) is designed to be used in a loop. Before entering the loop, set the TableName argument to an empty string buffer. Call **GetTableName( )** to get the first table name, then pass the first name back in to the function to get the next name, and so on to get all table names. An empty string and a result code of 4 is returned when the all table names have been returned.

The special datalogger system table, named "Status", is available at all times, and will be the only table returned by this call when no program is running. When programs are running, the "Status" table will always be the first table returned in the sequence of **GetTableName( )** calls.

If variables were declared as Public in the CRBasic program, a "Public" table will automatically be created; if present, this table will always appear as the last table in the sequence.

All other tables defined and named by the user within CRBasic DataTable statements will appear in sequence after the "Status" table, and before the "Public" table, if present.

Invocation of the **GetTableName( )** or **GetTableName2( )** function, with an empty string for the TableName argument, is essential after any new datalogger communications port has been opened, and before any record-level data retrieval occurs. This guarantees that the DLL has the necessary table definition information to use for data retrieval for that datalogger.

Declaration:

```
Declare Function GetTableName Lib "PC9000.DLL"
(ByVal TableName As String, ByVal TableNameSize
As Integer, TableSize As Long) As Integer
```

Parameters:

TableName a string buffer in which the next datalogger data table name is returned by the function.

TableNameSize the length of the TableName buffer.

TableSize returns the number or records allocated to the table.

Return Codes:

0 = Next table name returned.
1 = Port not open or datalogger does not respond.
4 = End of list.

Example:

```
' Fills a combo box with the current list of table names

Dim lTableSize As Long
Dim sTableName As String
Dim iRslt as Integer

   ' cboTable is a VB combo or list box control
   cboTable.Clear
   sTableName = String(40, vbNullChar)

   Do
      iRslt = GetTableName(sTableName, Len(sTableName), lTableSize)

      If iRslt = 1 Then Err.Raise 32000      'User defined error
code
      If iRslt = 0 Then cboTable.AddItem Trim$(sTableName)

   Loop Until (iRslt = 4) Or (Len(Trim$(sTableName)) = 0)
```

2.4.2 GetTableName2( ) - Retrieves the Names, Sizes and Times of All Datalogger Tables

GetTableName2( ) is designed to be used in a loop. Before entering the loop, set the TableName argument to an empty string buffer. Call **GetTableName2( )** to get the first table name, then pass the first name back in to the function to get the next name, and so on to get all table names. An empty string and a result code of 4 are returned when the all table names have been returned.

GetTableName2( ) is identical to **GetTableName( )** except for the two additional arguments associated with the data table interval.

The special datalogger system table, named "Status", is available at all times, and will be the only table returned by this call when no program is running. When programs are running, the "Status" table will always be the first table returned in the sequence of **GetTableName2( )** calls.

If variables were declared as Public in the CRBasic program, a "Public" table will automatically be created; if present, this table will always appear as the last table in the sequence.

All other tables defined and named by the user within CRBasic DataTable statements will appear in sequence after the "Status" table, and before the "Public" table, if present.

Invocation of the **GetTableName( )** or **GetTableName2( )** function, with an empty string for the TableName argument, is essential after any new datalogger communications port has been opened, and before any record-level data retrieval occurs. This guarantees that the DLL has the necessary table definition information to use for data retrieval for that datalogger.

The effective table interval (determined from the Seconds and Microseconds arguments combined) can be very useful in optimizing the frequency at which the data retrieval functions are called to acquire new records from the table. The interval will be set to zero, however, if the table is an "event-driven" table. An event-driven table is considered as one which does NOT contain a

DataInterval( ) statement in its CRBasic definition, thereby triggering data storage based on the value of a trigger variable rather than time. This definition holds, even when the CRBasic table definition also included conditional triggers and/or DataEvent( ) statements.

If the table is event-driven, there is no fool-proof means by which to determine its new record interval. An average interval can be determined by observing the record numbers from **GetRecentRecords( )** or **GetRecentValues( )** calls at two known instants of time, and dividing the elapsed time by the difference in the beginning and ending record numbers. The actual record intervals may not be constant, however, depending upon the program control instructions which determine when CRBasic CallTable statements are invoked.

Another means to determine a default refresh interval for an event-driven table is to monitor the value of the Status table variable "SecsPerRecord", using the **GetCurrentValue( )** function. This field is an array if there is more than one user-defined table in the program. The array is one-based, with the lowest index corresponding to the first user-defined table, and the highest index corresponding to the last user-defined table (i.e., there are no entries for the Public or Status tables in this array). The sequence of the tables is the same as that returned from this function, when called in a loop as described above. If the table is fixed-interval, the SecsPerRecord entry will be the same as the table interval returned by this function. If the table is event-driven, the entry will be equal to the shortest scan interval within the CRBasic program. This serves only as a best-guess suggested default update interval, from which to make more accurate observations of the actual interval.

GetTableName2( ) allows for programmers writing more advanced datalogger interfaces to obtain additional information not available in the base **GetTableName( )** call. The call interface to **GetTableName( )** will remain fixed, but **GetTableName2( )** may be modified in the future to return additional table-related information. At present, the only additional table parameters included here are the table interval.

Declaration:

```
Declare Function GetTableName2 Lib "PC9000.DLL"
(ByVal TableName As String, ByVal TableNameSize
As Integer, TableSize As Long, Seconds As Long,
MicroSeconds As Long, TableSig as Long) As
Integer
```

Parameters:

TableName	a string buffer in which the next datalogger data table name is returned by the function.
TableNameSize	the length of the TableName buffer.
TableSize	returns the number or records allocated to the table.
Seconds	returns the integer seconds portion of the table interval.
MicroSeconds	returns the integer microseconds portion of the table interval.

TableSig returns the table CRC signature, computed as defined in Campbell BMP protocols. Not used by other DLL functions at present, but can be used by application code to verify that a datalogger table structure has not changed since it was last retrieved.

Return Codes:

0 = Next table name returned.
1 = Port not open or datalogger does not respond.
4 = End of list.

Example:

```
' Fills a combo box with the current list of table names
' and fills two arrays with the table size and interval settings

Dim sTableName As String
Dim lSize as Long
Dim lSecs as Long
Dim lUSecs as Long
Dim lSig as Long
Dim iRslt as Integer
Dim iNumTables as Integer
Dim lTableSize( ) As Long
Dim lTableSig( ) As Long
Dim dTableTime( ) as Double

   ' cboTable is a VB combo or list box control
   cboTable.Clear
   sTableName = String(40, vbNullChar)

   Do
      iRslt = GetTableName2(sTableName, Len(sTableName), lSize, _
       lSecs, lUSecs, lTableSig)

      If iRslt = 1 Then Err.Raise 32000      ' User defined error code

      If iRslt = 0 Then
         cboTable.AddItem Trim$(sTableName)
         Redim Preserve lTableSize(0 to iNumTables)
         Redim Preserve lTableSig(0 to iNumTables)
         Redim Preserve dTableTime(0 to iNumTables)
         lTableSize(iNumTables) = lSize
         lTableSig(iNumTables) = lSig
         dTableTime(iNumTables) = lSecs + (lUSecs/1000000)
         iNumTables = iNumTables + 1
      End If
   Loop Until (iRslt = 4) Or (Len(Trim$(sTableName)) = 0)
```

2.4.3 GetFieldName( ) - Retrieves Field Names and Basic Associated Information

In similar fashion to the **GetTableName( )** function, **GetFieldName( )** is designed to be used in a loop in order to retrieve all of the fields in a table. Before entering the loop, specify the name of the Table in the TableName argument, and set the FieldName, Units, and Proc arguments to empty string buffers. Call **GetFieldName( )** to get the first field name, then pass the first

name back in to the function to get the next name, and so on to get all field names. An empty string and a result code of 4 are returned when the all field names have been returned for that table.

If a field is an array, one name will be returned for each element of the array. The field name for each individual element of an array will consist of three parts:

1. The base array name. If the array was not explicitly declared, but instead was created as part of a CRBasic output processing instruction (such as a Histogram or a CRBasic FFTFilt instruction), the base array name will consist of that particular output instruction's variable name, plus an extension that is unique to each CRBasic array output instruction.

   For example,
   The complete base array name of a variable TF used in a CRBasic Histogram instruction would be TF_Hst.
   The complete base array name of a variable TF used in a CRBasic Histogram4D instruction would be TF_H4D.

2. The array indices (one-based, single or multi-dimensional) as specified in the CRBasic program, enclosed in parentheses.

3. A one-based index value (always one-dimensional), corresponding to the absolute position of that particular array element in datalogger memory, enclosed in brackets.

Example for a one-dimensional array:

> if the array name was "TC" then the names returned would be TC(1)[1], TC(2)[2], TC(3)[3], etc.

Example for a multi-dimensional array:

> if a 2X2 array name was "SP" then the names returned would be SP(1,1)[1], SP(1,2)[2], SP(2,1)[3], SP(2,2)[4].

Field base names are potentially 16 characters long at a maximum, not including either of the two indexes. The maximum required length of the field name when arrays are involved will depend upon the number of indexes in the array, and the maximum number of digits in each.

The "Units" argument returns the units string name associated with a field, if units were assigned to that variable within the program using the CRBasic Units instruction.

The "Proc" argument returns processing information associated with a field, which in turn is set by the particular CRBasic output processing instruction used to generate the field in an output table. In the cases of non-array output processing, such as Samples, Averages, Min/Max or Standard Deviation instructions, the Proc code will be a simple code indicating the type of instruction that generated the field. In the cases of array output processing, such as that based upon Histograms and FFT instructions, the Proc code will consists of a list of comma-delimited items describing the precise characteristics of the output array. These processing fields are documented in the CR9000 and CR5000 instruction reference manuals.

Declaration:

```
Declare Function GetFieldName Lib "PC9000.DLL"
(ByVal TableName As String, ByVal FieldName As
String, ByVal FieldnameSize As Integer, ByVal
Units As String, ByVal UnitsSize As Integer,
ByVal Proc As String, ByVal ProcSize As Integer)
As Integer
```

Parameters:

TableName	the name of the datalogger data table from which the field names are to be retrieved.
FieldName	a string buffer in which the next data table field name is returned.
FieldNameSize	the length of the FieldName buffer.
Units	a string buffer in which the units associated with each field name are returned.
UnitsSize	the length of the Units buffer.
Proc	a string buffer in which the processing details associated with each field name are returned.
ProcSize	the length of the Proc buffer.

Return Codes:

0 = Next field name returned.
1 = Port not open or datalogger does not respond.
4 = End of list.

Example:

```
Sub ListPublicFields( )

Dim sTableName as String
Dim sNameBuf As String
Dim sUnitsBuf As String
Dim sProcBuf As String
Dim iRslt as Integer
'Dim iType as Integer

   ' lstPublic is a VB ListBox
   ' (Will need to be very wide to display the concatenated info
   ' as is programmed here)
   lstPublic.Clear

   sNameBuf = String(40, vbNullChar)
   sTableName = "Public"
   Do
      sUnitsBuf = String(40, vbNullChar)
      sProcBuf = String(60, vbNullChar)
      iRslt = GetFieldName(sTableName, sNameBuf, Len(sNameBuf), _
       sUnitsBuf, Len(sUnitsBuf), sProcBuf, Len(sProcBuf))

' Alternate call to get extended information
      ' iRslt = GetFieldName2(sTableName, sNameBuf, Len(sNameBuf), _
       sUnitsBuf, Len(sUnitsBuf), sProcBuf, Len(sProcBuf), iType)
      If iRslt = 1 Then Err.Raise 32000   'User-defined error

      lstPublic.AddItem Trim$(sNameBuf) & vbTab & Trim$(sUnitsBuf) _
       & Trim$(sProcBuf)
   Loop Until iRslt = 4

End Sub
```

2.4.4 GetFieldName2( ) - Retrieves Field Names and Extended Associated Information

In similar fashion to the **GetTableName( )** function, **GetFieldName2( )** is designed to be used in a loop in order to retrieve all of the fields in a table Before entering the loop, specify the name of the Table in the TableName argument, and set the FieldName, Units, and Proc arguments to empty string buffers. Call **GetFieldName2( )** to get the first field name, then pass the first name back in to the function to get the next name, and so on to get all field names. An empty string and a result code of 4 are returned when all field names have been returned for that table.

If a field is an array, one name will be returned for each element of the array. The field name for each individual element of an array will consist of three parts:

1. The base array name. If the array was not explicitly declared, but instead was created as part of a CRBasic output processing instruction (such as a Histogram) or a CRBasic FFTFilt instruction, the base array name will consist of the instructions variable argument plus an extension that is unique to each CRBasic array output instruction.

For example,
The complete base array name of a variable TF used in a CRBasic Histogram instruction would be TF_Hst.
The complete base array name of a variable TF used in a CRBasic Histogram4D instruction would be TF_H4D.

2. The array indices (one-based, single or multi-dimensional) as specified in the CRBasic program, enclosed in parentheses.

3. A one-based index value (always one-dimensional), corresponding to the absolute position of that particular array element in datalogger memory, enclosed in brackets.

Example for a one-dimensional array:

if the array name was "TC" then the names returned would be TC(1)[1], TC(2)[2], TC(3)[3], etc.

Example for a multi-dimensional array:

if a 2X2 array name was "SP" then the names returned would be SP(1,1)[1], SP(1,2)[2], SP(2,1)[3], SP(2,2)[4].

Field base names are potentially 16 characters long at a maximum, not including either of the two indexes. The maximum required length of the field name when arrays are involved will depend upon the number of indexes in the array, and the maximum number of digits in each.

The "Units" argument returns the units string name associated with a field, if units were assigned to that variable within the program using the CRBasic Units instruction.

The "Proc" argument returns processing information associated with a field, which in turn is set by the particular CRBasic output processing instruction used to generate the field in an output table. In the cases of non-array output processing, such as Samples, Averages, Min/Max or Standard Deviation instructions, the Proc code will be a simple code indicating the type of instruction that generated the field. In the cases of array output processing, such as that based upon Histograms and FFT instructions, the Proc code will consists of a list of comma-delimited items describing the precise characteristics of the output array. These processing fields are documented in the CR9000 and CR5000 instruction reference manuals.

GetFieldName2( ) allows for programmers writing more advanced datalogger interfaces to obtain additional information not available in the base **GetFieldName( )** call. The call interface to **GetFieldName( )** will remain fixed, but **GetFieldName2( )** may be modified in the future to return additional field-related information. At present, the only additional field parameters included here are the CSI field data type.

Declaration:

```
Declare Function GetFieldName2 Lib "PC9000.DLL"
(ByVal TableName As String, ByVal FieldName As
String, ByVal FieldnameSize As Integer, ByVal
Units As String, ByVal UnitsSize As Integer,
ByVal Proc As String, ByVal ProcSize As Integer,
ByVal DataType as Integer) As Integer
```

Parameters:

TableName	the name of the datalogger data table from which the field names are to be retrieved.
FieldName	a string buffer in which the next data table field name is returned.
FieldNameSize	the length of the FieldName buffer.
Units	a string buffer in which the units associated with each field name are returned.
UnitsSize	the length of the Units buffer.
Proc	a string buffer in which the processing details associated with each field name are returned.
ProcSize	the length of the Proc buffer.
DataType	the CSI data type of the field.

Return Codes:

0 = Next field name returned.
1 = Port not open or datalogger does not respond.
4 = End of list.

Example:

See examples for **GetFieldName( )**. A list of the recognized CSI data types if given here.

```
Private Const CsiAscii = 11
Private Const CsiAsciiZ = 16
Private Const CsiBool = 10
Private Const CsiBool8 = 17
Private Const CsiFp4 = 8
Private Const CsiFs2 = 7
Private Const CsiFs3 = 15
Private Const CsiFs4 = 26
Private Const CsiFsf = 27
Private Const CsiIeee4 = 9
Private Const CsiIeee4Lsf = 24
Private Const CsiIeee8 = 18
Private Const CsiIeee8Lsf = 25
Private Const CsiInt1 = 4
Private Const CsiInt2 = 5
Private Const CsiInt2Lsf = 19
Private Const CsiInt4 = 6
Private Const CsiInt4Lsf = 20
Private Const CsiInt8 = 29
Private Const CsiInt8Lsf = 28
Private Const CsiNSec = 14
Private Const CsiNSecLsf = 23
Private Const CsiSec = 12
Private Const CsiUInt1 = 1
Private Const CsiUInt2 = 2
Private Const CsiUInt2Lsf = 21
Private Const CsiUInt4 = 3
Private Const CsiUInt4Lsf = 22
Private Const CsiUSec = 13
```

2.4.5 TableCtrl( ) - Clear Logged Records in a Table, or Insert File Marks in File-based Table

Declaration:

```
Declare Function TableCtrl Lib "PC9000.DLL"
(ByVal ROption As Integer, ByVal TableName As
String) As Integer
```

Parameters:

ROption specifies the option to use:

1 = reset table
2 = "roll over" data into a new file. In essence, this means that a file mark will be written into the table before the next record is stored. The file mark is like one that would be entered automatically, according to the rules configured in a CRBasic DataEvent( ) instruction for the specified table. Only applies to tables that are maintained in files like those on a CR9000 PAM card.

TableName specifies the table on which to operate.

Return Codes:

0 = OK.
1 = datalogger does not respond.
2 = option not applicable.
3 = bad table name.

Example:

```
Dim sTableName as String
Dim iRslt as Integer

' cboTable is a VB combo or list box control
' containing the names of the tables for the currently
' running program.

If cboTable.ListIndex >=0 Then
   sTableName = cboTable.List(cboTable.ListIndex)
   iRslt = TableCtrl(1, sTableName)
   If iRslt <> 0 Then Err.Raise 32000   'Application-defined error
End If
```

2.5 Data Retrieval Functions

2.5.1 GetVariable( ) - Get the Current Value of a Floating Point Variable

Normally used to access the current variables in the "Public" table, but can be used to retrieve floating point values in any table as explained below.

A "Public" table will exist in the datalogger whenever at least one variable has been declared as Public in a CRBasic program.

The FieldName argument accepts either syntax **FIELDNAME** or **TABLENAME.FIELDNAME** . If the TableName portion of the argument is omitted, the Public table is assumed. The "." delimiter must only be present if the TableName is specified along with the field name.

When the field to be retrieved is an element of an array, the syntax rules are not completely intuitive. Refer to the documentation for **GetFieldName( )** for a complete description of array field name syntax. As described there, the array field syntax consists of a base name, the programmatically specified array indices, in parentheses (1,2, or 3-dimensional), and then a one-dimensional array index, in brackets. When fully specifying the array field element in this function, however, the following rules apply:

Consider a typical Flag field in the Public table. The field names returned by **GetFieldName( )** will be Flag(1)[1], Flag(2)[2], etc. When specifying a particular element of the Flag array in this function, only the second index is important. In other words:
Flag(2)[2]
Flag(100)[2]
Flag()[2]
All specify the same thing, that is, the second element of the Flag array. The characters within the parentheses are ignored, but the parentheses must be there if the field is in fact part of an array. Passing an out-of-bounds index value within the brackets will cause a "bad field name" error.
IMPORTANT: If the brackets are omitted for a field name which is an array, this FieldName syntax will not cause an error but will be interpreted as specifying the first element of the array.

For **GetVariable( )** to succeed, one initial call must have been made previously to **GetTableName( )** or **GetTableName2( )** (using an empty string for the TableName argument) so that the DLL can retrieve the necessary table definition information to properly process the data request.

Declaration:

```
Declare Function GetVariable Lib "PC9000.DLL"
(ByVal FieldName As String, Value As Single) As
Integer
```

Parameters:

FieldName the name of a data field in the datalogger.

Value the value of FieldName.

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.
2 = Bad table name or no data.
3 = Bad field name.

Example:

```
Public Sub UpdateFlagStates
' This sub populates a list box with the states of as many flags
' as it can find.

Dim sFlagField As String
Dim fVal As Single
Dim iFlag As Integer
Dim sVal As String
Dim iRslt As Integer

   'lstFlags is a list box showing the names and states
   'of all flags found in the public table
   lstFlags.Clear

   iFlag = 1
   Do
      sFlagField = "Flag(" & iFlag & ")[" & iFlag & "]"
      Rslt = GetVariable(sFlagField, fVal)

      If Rslt = 1 Then Err.Raise 32000   ' user defined error
      If Rslt = 2 Then Exit Do

      sVal = IIf(fVal <> 0, "HI", "LO")
      lstFlags.AddItem aFlagField & vbTab & fVal
      iFlag = iFlag + 1
   Loop

End Sub
```

2.5.2 SetVariable( ) - Set the Current Value of a Floating Point Variable

Normally used to set the current values of variables in the "Public" table, but can be used to set floating point values in any writable Status table field as well, as explained below. The values in output tables are never writable by this function.

A Public Table will exist whenever at least one variable has been declared as Public in a CRBasic program.

The FieldName argument accepts either syntax **FIELDNAME** or **TABLENAME.FIELDNAME** . If the TableName portion of the argument is omitted, the Public table is assumed. The "." delimiter must only be present if the TableName is specified along with the field name.

When the field to be retrieved is an element of an array, the syntax rules are not completely intuitive. Refer to the documentation for **GetFieldName( )** for a complete description of array field name syntax. As described there, the array field syntax consists of a base name, the programmatically specified array indices, in parentheses (1,2, or 3-dimensional), and then a one-dimensional array index, in brackets. When fully specifying the array field element in this function, however, the following rules apply:

Consider a typical Flag field in the Public table. The field names returned by **GetFieldName( )** will be Flag(1)[1], Flag(2)[2], etc. When specifying

a particular element of the Flag array in this function, only the second index is important. In other words:
Flag(2)[2]
Flag(100)[2]
Flag()[2]
All specify the same thing, that is, the second element of the Flag array. The characters within the parentheses are ignored, but the parentheses must be there if the field is in fact part of an array. Passing an out-of-bounds index value within the brackets will cause a "bad field name" error.
IMPORTANT: If the brackets are omitted for a field name which is an array, this FieldName syntax will not cause an error but will be interpreted as specifying the first element of the array. This is even more important here than with the "Get" functions, as improper syntax may cause the application to inadvertently change the value of the first field in an array, in error.

For **SetVariable( )** to succeed, one initial call must have been made previously to **GetTableName( )** or **GetTableName2( )** (using an empty string for the TableName argument) so that the DLL can retrieve the necessary table definition information to properly process the data request.

This function will not return error codes when attempting to set the value of a read-only field. For this and all of the above reasons, use extra care in validating the table and field names used with this function.

Declaration:

```
Declare Function SetVariable Lib "PC9000.DLL"
(ByVal FieldName As String, ByVal value As
Single) As Integer
```

Parameters:

FieldName the name of a data field in the datalogger.

Value the value of FieldName.

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.
3 = Bad table or field name.

Example:

```
Private Sub chkFlag_Click(Index As Integer)

Dim fVal As Single
Dim sFlagField As String
Dim iRslt As Integer

' chkFlag() is a control array of check boxes,
' one for each flag in the public table.

   fVal = IIf(chkFlag(Index).Value <> 0, True, False)
   sFlagField = "Flag(" & Index & ")[" & Index & "]"
   iRslt = SetVariable(sFlagField, fVal)
   If iRslt <> 0 Then
      MsgBox "Could not write to public table."
   End If

End Sub
```

2.5.3 GetCurrentValue( ) - Get the Most Recent Value of a Field, in ASCII Format

GetCurrentValue( ) allows individual data table field values to be retrieved, one at a time.

The function will work on any valid table and field, although it is most useful with the datalogger's system status table. The reason is that many of the Status table fields are non-numeric, and the other data retrieval functions assume data that is strictly numeric in value. As a result, those other functions will not return values for the non-numeric status table fields.

When the field to be retrieved is an element of an array, the syntax rules are not completely intuitive. Refer to the documentation for **GetFieldName( )** for a complete description of array field name syntax. As described there, the array field syntax consists of a base name, the programmatically specified array indices, in parentheses (1,2, or 3-dimensional), and then a one-dimensional array index, in brackets. When fully specifying the array field element in this function, however, the following rules apply:

Consider a typical Flag field in the Public table. The field names returned by **GetFieldName( )** will be Flag(1)[1], Flag(2)[2], etc. When specifying a particular element of the Flag array in this function, only the second index is important. In other words:
Flag(2)[2]
Flag(100)[2]
Flag()[2]

All specify the same thing, that is, the second element of the Flag array. The characters within the parentheses are ignored, but the parentheses must be there if the field is in fact part of an array. Passing an out-of-bounds index value within the brackets will cause a "bad field name" error.
IMPORTANT: If the brackets are omitted for a field name which is an array, this FieldName syntax will not cause an error but will be interpreted as specifying the first element of the array.

On tables other than the Status table, this function will not normally be used, as other data retrieval calls will tend to be much faster, particularly if all of the fields in a table are being retrieved. Another reason that **GetCurrentValue( )** is less useful beyond the Status table is that, for simplicity, this call returns no record number or timestamp information. That information is meaningless in the Status table, since the Status table is not a logging table; on the other hand, it is generally desired when retrieving data from user-defined logging tables.

Refer also to the **GetStatusValue( )** function for an alternative method of retrieving Status table information.

Declaration:

```
Declare Function GetCurrentValue Lib "PC9000.DLL"
(ByVal TableName As String, ByVal FieldName As
String, ByVal ValueBuf As String, ByVal
ValueBufSize As Integer) As Integer
```

Parameters:

TableName the name of a data table in the datalogger.

FieldName the name of a data field in the datalogger.

ValueBuf a string variable used to return the value of the specified field.

ValueBufSize the maximum string length of the returned value.

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.
2 = No data exists or bad table name.
3 = Bad field name.

Example:

```
' This function runs a loop to get all of the Names and values of the
' fields in the datalogger status table, and places them in a
' two dimensional array
'
Private Sub GetStatusFields(sArray() As String)

Dim sTblName As String
Dim sFldName As String
Dim sUnits As String
Dim sProc As String
Dim iRslt As Integer
Dim sBuf As String
Dim iFieldCnt As Integer

On Error GoTo Err_Status

   sTblName = "Status" & vbNullChar
   sFldName = String(40, vbNullChar)
   sUnits = String(40, vbNullChar)
   sProc = String(60, vbNullChar)

   ReDim sArray(0 To 1, 0 To 0)
   Do
      iRslt = GetFieldName(sTblName, sFldName, Len(sFldName), _
       sUnits, Len(sUnits), sProc, Len(sProc))
      If iRslt = 1 Then Err.Raise 32000      'no response
      If iRslt = 4 Then Exit Do          'end of list

      sBuf = String(40, vbNullChar)
      iRslt = GetCurrentValue(sTblName, sFldName, sBuf, _
       Len(sBuf))
      If iRslt = 1 Then Err.Raise 32000      'no response
```

(Example continues on next page)

GetRecords1() Example (continued from previous page)

```
ReDim Preserve sArray(0 To 1, 0 To iFieldCnt)
      sArray(0, iFieldCnt) = Trim$(sFldName)
      sBuf = Trim$(sBuf)
      If Asc(Left$(sBuf, 1)) = 34 Then _
       sBuf = Mid$(sBuf, 2, Len(sBuf) - 2)
      sArray(1, iFieldCnt) = Trim$(sBuf)
      iFieldCnt = iFieldCnt + 1

   Loop

   Exit Sub

Err_Status:
   If Err.Number = 32000 Then
      MsgBox "Port not open or datalogger does not respond."
   Else
      MsgBox Err.Description
   End If

End Sub
```

2.5.4 GetStatusValue( ) – Optimized Status Table Retrieval, in ASCII Format

GetStatusValue( ) provides an alternate method for retrieving the current values of Status table fields, particularly when a large list of values is needed all at once. It operates in a similar manner to **GetCurrentValue( )**, but is limited to, and optimized for, the Status table.

The datalogger transaction that is executed by PC9000.DLL when either the **GetCurrentValue( )** or **GetStatusValue( )** functions are called gets all of the field values, even though only one value at a time is returned to the calling routine. For tables having only a few fields, (or when retrieving only a short list of fields), this inefficiency is not of great concern. For the case of the status table, however, where there may be 100 or more fields, the overhead required to retrieve all values, one value at a time, can be significant, particularly over a slow data link.

GetStatusValue( ) overcomes this inefficiency through the use of the additional Refresh parameter. When using a loop to retrieve a list of values, setting this argument to 1 on the first pass, and to zero on every subsequent pass will cause the status table values to only be retrieved from the datalogger one time.

GetStatusValue( ) follows identical rules to **GetCurrentValue( )** in all other aspects (with the exception that the table name does not need to be specified). Refer to the documentation for **GetCurrentValue( )** and **GetFieldName( )** for a complete description of array field name syntax.

Declaration:

```
Declare Function GetStatusValue Lib "PC9000.DLL"
(ByVal FieldName As String, ByVal ValueBuf As
String, ByVal ValueBufSize As Integer, ByVal
Refresh As Integer) As Integer
```

Parameters:

FieldName the name of a data field in the datalogger.

ValueBuf a string variable used to return the value of the specified field.

ValueBufSize the maximum string length of the returned value.

Refresh 1 = retrieve new data from the datalogger on this pass, 0 = used cached data.

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.
2 = No data exists.
3 = Bad field name.

Example:

```
' This function runs a loop to get all of the Names and values of the
' fields in the datalogger status table, and places them in a
' two dimensional array
'
Private Sub GetStatusFields(sArray() As String)

Dim sTblName As String
Dim sFldName As String
Dim sUnits As String
Dim sProc As String
Dim iRslt As Integer
Dim sBuf As String
Dim iFieldCnt As Integer
Dim iRefresh as Integer

On Error GoTo Err_Status

   sTblName = "Status" & vbNullChar
   sFldName = String(40, vbNullChar)
   sUnits = String(40, vbNullChar)
   sProc = String(60, vbNullChar)

   ReDim sArray(0 To 1, 0 To 0)
   Do
      iRslt = GetFieldName(sTblName, sFldName, Len(sFldName), _
       sUnits, Len(sUnits), sProc, Len(sProc))
      If iRslt = 1 Then Err.Raise 32000      'no response
      If iRslt = 4 Then Exit Do          'end of list

      sBuf = String(40, vbNullChar)
      iRefresh = IIf(iFldCnt = 0, 1, 0)
      iRslt = GetCurrentValue(sFldName, sBuf, Len(sBuf), _
       iRefresh)
      If iRslt = 1 Then Err.Raise 32000      'no response
```

(Example continues on next page)

GetRecords1() Example (continued from previous page)

```
ReDim Preserve sArray(0 To 1, 0 To iFieldCnt)
      sArray(0, iFieldCnt) = Trim$(sFldName)
      sBuf = Trim$(sBuf)
      If Asc(Left$(sBuf, 1)) = 34 Then _
       sBuf = Mid$(sBuf, 2, Len(sBuf) - 2)
      sArray(1, iFieldCnt) = Trim$(sBuf)
      iFieldCnt = iFieldCnt + 1

   Loop

   Exit Sub

Err_Status:
   If Err.Number = 32000 Then
      MsgBox "Port not open or datalogger does not respond."
   Else
      MsgBox Err.Description
   End If

End Sub
```

2.5.5 GetRecentRecords( ) - Get All Data from Most Recent Records of the Specified Table; GetRecentRecordsTS( ) - Get Recent Records with Timestamps

GetRecentRecords( ) is the most frequently employed call to perform basic bulk data retrieval. It is also the initial call used in more complex bulk data retrieval strategies, as it enables an application to determine the current record number from which to start.

GetRecentRecordsTS( ) is identical to **GetRecentRecords( )** with the exception that an array of timestamp data is also computed and returned.

The return values are placed in the 4-byte floating point array pointed to by ArrayStart. Values returned will always start at the beginning of a record, and the values will be ordered according to the field names, in precisely the same order that those field names are retrieved using either the **GetFieldName( )** or **GetFieldName2( )** functions. If the ArraySize argument implies that data from more than one record is desired, subsequent records will appear immediately following the values for the first record. Data from multiple records will always be ordered starting with the earliest record to the most recent record.

The RecNum value returned by the function will indicate the last record from which data was retrieved. When retrieving multiple records, to determine the starting record number, count the number of records returned by the function (= to NbrGot / (number of fields in the table)) and compute the starting record number from there.

Record numbers are stored internally in the datalogger as 32 bit unsigned integers, but the Visual Basic long data type interprets these values as 32 bit signed integers. As a result, when the record number reaches the largest signed 32 bit value allowed ($2^{31} – 1$), it will appear to "wrap around" to the largest signed negative value allowed (-2^{31}) and begin counting up from there. This

needs to be taken into account when working with record numbers in various data retrieval algorithms.

The actual number of values returned in a data retrieval call is dependent upon data table sector boundaries within the datalogger, and will often be less be less than the number of values requested, if a large number of values was requested in a single call. It will of course also be limited by the total number of records available in the table. As a result, the maximum number of values that are actually retrievable will vary from table to table and from call to call, but will always stop at the end of a record boundary. In other words, if the DLL function (not the array size) limits the number of values returned, it will always return whole records.

Due to the nature of this function described above, programmers wishing to reliably recover all real time data, without any gaps between records, will want to use this function in conjunction with **GetRecordsSinceLast( ).** In those cases, use **GetRecentRecords( )** to establish a current record number baseline, and then move backward or forward from that starting point.

If the number of values requested is within the limits that are retrievable in a single function call, but the ArraySize is not evenly divisible by the number of fields in a record, the function will fill as much of the array as possible with partial record data.

GetRecentRecordsTS( ) will return one timestamp value for every field data value retrieved. This means that all the timestamp values for fields within a single record will be the same.

Declaration:

```
Declare Function GetRecentRecords Lib
"PC9000.DLL" (ByVal TableName As String,
ArrayStart As Single, ByVal ArraySize As Integer,
NbrGot As Integer, RecNum As Long) As Integer

Declare Function GetRecentRecordsTS Lib
"PC9000.DLL" (ByVal TableName As String,
ArrayStart As Single, ByVal ArraySize As Integer,
NbrGot As Integer, RecNum As Long, TSStart As
Double) As Integer
```

Parameters:

TableName name of the table from which the most recent records are read.

ArrayStart pointer to the first location within the 4-byte floating point array that has been pre-dimensioned to receive the data. Usually this will be the first element of the array.

ArraySize the number of data values that the function should attempt to retrieve and place in the array pointed to by ArrayStart. The array should be sized large enough to receive as many values as are indicated by the starting location in ArrayStart and the number of values specified here. The effective number of records to be collected is ArraySize divided by the number of fields in the table.

NbrGot indicates the number of values (not records) actually retrieved, and returned in the 4-byte floating point array. The values returned in this array do not include time stamps or record numbers.

RecNum indicates the record number from which data was retrieved, if all values are from the same record. If values from multiple records were retrieved, indicates the last record number from which values appear in the return arrays. If no data is retrieved, returns –1.

Applies to GetRecentRecordsTS( ) only:

TSStart pointer to the first location within the 8-byte floating point array that has been pre-dimensioned to receive timestamp information. Usually this will be the first element of the array; in any event this array should have the same number of elements and starting index location as the 4-byte floating point values array.

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.
2 = No data exists. Also returned if ArraySize is larger than the entire number of values (i.e., records x fields) currently residing in the table.
3 = Bad table name.

Example:

Basic examples are provided here. For more useful examples, refer to the PC9000Test program example code which is included on the PC9000.DLL developer kit disk.

```
Sub GetRecords1()

Dim iRslt As Integer
Dim sTblName As String
Dim fVals() As Single
Dim dTS() As Double
Dim iVals As Integer
Dim lRecNum As Long
Dim iNbrGot As Integer
Dim iVal As Integer
Dim bTS As Boolean

On Error GoTo Err_GetRecs

   lstRecords.Clear
   lstTS.Clear

   iVals = Int(Val(txtNumVals.Text))
   If iVals <= 0 Then
      Err.Raise 32000, Description:="Enter the number of values " _
        & "you would like to retrieve."
   End If
```

(Example continues on next page)

GetRecords1() Example (continued from previous page)

```
   sTblName = txtTableName.Text & vbNullChar
   ReDim fVals(0 To iVals - 1)

   bTS = (chkTS.Value <> 0)
   If bTS Then ReDim dTS(0 To iVals - 1)

   If bTS Then
      iRslt = GetRecentRecordsTS(sTblName, fVals(0), iVals, iNbrGot, _
lRecNum, dTS(0))
   Else
      iRslt = GetRecentRecords(sTblName, fVals(0), iVals, iNbrGot, lRecNum)
   End If
   Select Case iRslt
      Case 0
         lblStat.Caption = "Data retrieval successful."
      Case 1
         lblStat.Caption = "Port not open or datalogger does not respond."
      Case 2
         lblStat.Caption = "No data."
      Case 3
         lblStat.Caption = "Bad table name."
      Case 5
         lblStat.Caption = "Errors in data transmission."
      Case Else
         lblStat.Caption = "Unknown result code."
   End Select

   txtRecNum.Text = lRecNum

   For iVal = 1 To iNbrGot
      lstRecords.AddItem fVals(iVal - 1)
      If bTS Then lstTS.AddItem dTS(iVal – 1)
   Next iVal


Exit_GetRecs:
   Exit Sub

Err_GetRecs:
   MsgBox Err.Description
   Resume Exit_GetRecs

End Sub
```

2.5.6 GetRecordsSinceLast( ) - Get Data from Specified Table Starting at Specified Record and GetRecordsSinceLastTS( ) - Get Specified Record Data with Timestamps

Similar to **GetRecentRecords( )**, but provides more control over the precise records retrieved, facilitating "gapless" real time data retrieval, up to the bandwidth limits of the datalogger communications interface and performance limitations of the application program.

GetRecordsSinceLastTS( ) is identical to **GetRecordsSinceLast ( )** with the exception that an array of timestamp data is also computed and returned.

The return values are placed in the 4-byte floating point array pointed to by ArrayStart. Values returned will always start at the beginning of a record, and the values will be ordered according to the field names, in precisely the same order that those field names are retrieved using either the **GetFieldName( )** or **GetFieldName2( )** functions. If the ArraySize argument implies that data from more than one record is desired, subsequent records will appear immediately following the values for the first record. Data from multiple records will always be ordered starting with the earliest record to the most recent record.

The RecNbr value passed to the function tells the function the record number from which it should attempt to start retrieving data (if that record number is available). The RecNbr value returned by the function will typically indicate the last record from which data was actually retrieved. When retrieving multiple records, to determine the starting record number, count the number of records returned by the function (= to NbrGot / (number of fields in the table)) and compute the starting record number from there.

A very important concept to understand when using **GetRecordsSinceLast()** is that, if there is data available in the specified table, this function will almost always return some records, regardless of the RecNbr value specified. If the record specified by RecNbr currently exists in the table, all is as anticipated. If RecNbr is greater than the last record number written to the table, or is less than the first record number still available, then record retrieval will usually begin with the first record in the table, with one exception. A special case exists when the RecNbr asked for is not available, but is the number of the very next record that is to be written. In that case, the function returns an error code, "No data", and the RecNbr is not changed. This is very useful when employing continuous real time data collection strategies that are periodically polling for any new records that may have become available.

The method used to initialize the RecNbr argument before the first call to **GetRecordsSinceLast()**, will therefore depend upon the type of data collection that is desired:

- To collect all records in the table, initially set the RecNbr value to zero. The value returned will correspond to where in the table that the function found the earliest records still available.

- To collect records starting with the current record and moving forward as new records are stored, make a one-record call to **GetRecentRecords( )**, to determine the current record number.

On all subsequent calls, set RecNbr equal to the value returned by the previous call + 1 (unless no data was returned on the last call, in which case the RecNbr value should not be incremented).

Record numbers are stored internally in the datalogger as 32 bit unsigned integers, but the Visual Basic long data type interprets these values as 32 bit signed integers. As a result, when the record number reaches the largest signed 32 bit value allowed ($2^{31} – 1$), it will appear to "wrap around" to the largest signed negative value allowed (-2^{31}) and begin counting up from there. This needs to be taken into account when working with record numbers in various data retrieval algorithms.

The actual number of values returned in a data retrieval call is dependent upon data table sector boundaries within the datalogger, and will often be less be less than the number of values requested, if a large number of values was requested in a single call. It will of course also be limited by the total number of records available in the table. As a result, the maximum number of values that are actually retrievable will vary from table to table and from call to call, but will always stop at the end of a record boundary. In other words, if the DLL function (not the array size) limits the number of values returned, it will always return whole records.

If the number of values requested is within the limits that are retrievable in a single function call, but the ArraySize is not evenly divisible by the number of fields in a record, the function will fill as much of the array as possible with partial record data. Returning partial records is usually not a good strategy to use with this function, however.

Keep in mind that this DLL supports direct, real time data collection only. Datalogger records are not being retrieved and cached by a LoggerNet server. Therefore, depending upon table intervals, table sizes, and the frequency at which calls are made using this function, it is quite possible that sequential table records will be overwritten before this function call retrieves the next group of records. It is entirely up to the applications program to carefully monitor the value of the RecNbr record number pointer, and make intelligent determinations as to the status of datalogger tables in this regard.

GetRecordsSinceLastTS( ) will return one timestamp value for every field data value retrieved. This means that all the timestamp values for fields within a single record will be the same.

Declaration:

```
Declare Function GetRecordsSinceLast Lib
"PC9000.DLL" (ByVal TableName As String, RecNbr
As Long, ArrayStart As Single, ByVal ArraySize As
Integer, NbrGot As Integer) As Integer

Declare Function GetRecordsSinceLastTS Lib
"PC9000.DLL" (ByVal TableName As String, RecNbr
As Long, ArrayStart As Single, ByVal ArraySize As
Integer, NbrGot As Integer, TSStart As Double) As
Integer
```

Parameters:

TableName name of the table from which the desired records are to be read.

RecNbr specifies the record number from which to attempt to begin retrieving data. Returns with the last record number from which data was retrieved.

ArrayStart pointer to the first location within the 4-byte floating point array that has been pre-dimensioned to receive the data. Usually this will be the first element of the array.

ArraySize the number of data values that the function should attempt to retrieve and place in the array pointed to by ArrayStart. The array should be sized large enough to receive as many values as are indicated by the starting

location in ArrayStart and the number of values specified here. The effective number of records to be collected is ArraySize divided by the number of fields in the table.

NbrGot indicates the number of values (not records) actually retrieved, and returned in the 4-byte floating point array. The values returned in this array do not include time stamps or record numbers.

Applies to GetRecordsSinceLastTS( ) only:

TSStart pointer to the first location within the 8-byte floating point array that has been pre-dimensioned to receive timestamp information. Usually this will be the first element of the array; in any event this array should have the same number of elements and starting index location as the 4-byte floating point values array.

Return Codes:

0 = done.
1 = Port not open or datalogger does not respond.
2 = No data exists.
3 = Bad table name.

Example:

Basic examples are provided here. For more useful examples, refer to the PC9000Test program example code which is included on the PC9000.DLL developer kit disk.

```
Sub GetRecords2()

Dim iRslt As Integer
Dim sTblName As String
Dim fVals() As Single
Dim dTS() As Double
Dim iVals As Integer
Dim lRecNum As Long
Dim iNbrGot As Integer
Dim iVal As Integer
Dim bTS As Boolean

On Error GoTo Err_GetRecs

   lstRecords.Clear
   lstTS.Clear

   iVals = Int(Val(txtNumVals.Text))
   If iVals <= 0 Then
      Err.Raise 32000, Description:="Enter the number of values " _
        & "you would like to retrieve."
   End If

   sTblName = txtTableName.Text & vbNullChar
   ReDim fVals(0 To iVals - 1)
```

(Example continues on next page)

GetRecords2() Example (continued from previous page)

```
   bTS = (chkTS.Value <> 0)
   If bTS Then ReDim dTS(0 To iVals - 1)

   lRecNum = Int(Val(txtRecNum.Text))
   If bTS Then
      iRslt = GetRecordsSinceLastTS(sTblName, lRecNum, fVals(0), iVals, _
iNbrGot, dTS(0))
   Else
      iRslt = GetRecordsSinceLast(sTblName, lRecNum, fVals(0), iVals, _
 iNbrGot)
   End If
   Select Case iRslt
      Case 0
         lblStat.Caption = "Data retrieval successful."
      Case 1
         lblStat.Caption = "Port not open or datalogger does not respond."
      Case 2
         lblStat.Caption = "No data."
      Case 3
         lblStat.Caption = "Bad table name."
      Case 5
         lblStat.Caption = "Errors in data transmission."
      Case Else
         lblStat.Caption = "Unknown result code."
   End Select

   txtRecNum.Text = lRecNum

   For iVal = 1 To iNbrGot
      lstRecords.AddItem fVals(iVal - 1)
      If bTS Then lstTS.AddItem dTS(iVal – 1)
   Next iVal

Exit_GetRecs:
   Exit Sub

Err_GetRecs:
   MsgBox Err.Description
   Resume Exit_GetRecs

End Sub
```

2.5.7 GetRecentValues( ) - Get Most Recent Values of the Specified Table and Field; GetRecentValuesTS( ) - Get Recent Values with Timestamps

GetRecentValues( ) and **GetValuesSinceLast( )** are very similar in their behavior to **GetRecentRecords( )** and **GetRecordsSinceLast( )**, with the exception that these functions only return a single field value per record. These functions therefore facilitate the same data retrieval strategies as the corresponding full record function calls, with the potential for speed improvements on large tables with very fast data update intervals, since data

retrieval is specific to a particular field. There is a very significant qualifier to this potential speed improvement, however, that is described below.

GetRecentValuesTS( ) is identical to **GetRecentValues( )** with the exception that an array of timestamp data is also computed and returned.

When the field to be retrieved is an element of an array, the FieldName syntax rules are not completely intuitive. Refer to the documentation for **GetFieldName( )** for a complete description of array field name syntax. As described there, the array field syntax consists of a base name, the programmatically specified array indices, in parentheses (1,2, or 3-dimensional), and then a one-dimensional array index, in brackets. When fully specifying the array field element in this function, however, the following rules apply:

Consider a typical Flag field in the Public table. The field names returned by **GetFieldName( )** will be Flag(1)[1], Flag(2)[2], etc. When specifying a particular element of the Flag array in this function, only the second index is important. In other words:
Flag(2)[2]
Flag(100)[2]
Flag()[2]
All specify the same thing, that is, the second element of the Flag array. The characters within the parentheses are ignored, but the parentheses must be there if the field is in fact part of an array. Passing an out-of-bounds index value within the brackets will cause a "bad field name" error.
IMPORTANT: If the brackets are omitted for a field name which is an array, this FieldName syntax will not cause an error but will be interpreted as specifying the first element of the array.

The return values are placed in the 4-byte floating point array pointed to by ArrayStart. If the ArraySize argument implies that data from more than one record is desired, subsequent values will appear immediately in order by record. Values from multiple records will always be ordered starting with the earliest record to the most recent record.

Refer to the documentation for **GetRecentRecords( )** and **GetRecordsSinceLast( )**, for a detailed description of how to set and interpret record number pointers when performing continuous real time data collection. The behavior of the record number arguments and return codes in **GetRecentValues( )** and **GetValuesSinceLast( )** is identical to those functions. Due to the nature of these functions as detailed there, programmers wishing to reliably recover all real time data, without any gaps between values, will want to use this function in conjunction with **GetValuesSinceLast( ).** In those cases, use **GetRecentValues( )** to establish a current record number baseline, and then move backward or forward from that starting point.

The potential speed improvements to be realized from using **GetRecentValues( )/ GetValuesSinceLast( )** are greatly affected by one limitation of the underlying data retrieval protocol. Ideally, when requesting values from an individual field (as opposed to an entire record), only those field values requested will be transmitted by the datalogger, thereby enabling the most optimum bandwidth usage possible. The CSI protocols treat all elements from an array as a single field, however; if the field requested is part

of an array, all the values from that array will be transmitted. When dealing with very large arrays, the performance gains realized from using these functions will therefore be minimal compared to using the full-record retrieval functions. (If multiple individual values calls are made, the performance may actually be slower.)

As a result, if it is desired to obtain the fastest real time performance possible using this DLL, the associated CRBasic programs should set up the key values to be monitored into non-array field locations. On the other hand, if the data retrieval strategy in general involves the optimizing of high-speed data retrieval from very large arrays (such as is typical with FFT results), consider using the **GetPartialFieldValues( )** and **GetPartialFieldArray( )** data retrieval functions. These two functions were specifically written for those purposes.

GetRecentValuesTS( ) will return one timestamp value for every data value retrieved.

Declaration:

```
Declare Function GetRecentValues Lib "PC9000.DLL"
(ByVal TableName As String, ByVal FieldName As
String, ArrayStart As Single, ByVal ArraySize As
Integer, NbrGot As Integer, RecNum as Long) As
Integer

Declare Function GetRecentValuesTS Lib
"PC9000.DLL" (ByVal TableName As String, ByVal
FieldName As String, ArrayStart As Single, ByVal
ArraySize As Integer, NbrGot As Integer, RecNum
As Long, TSStart As Double) As Integer
```

Parameters:

TableName name of the table from which the most recent values are read.

FieldName name of a field in the specified table.

ArrayStart pointer to the first location within the 4-byte floating point array that has been pre-dimensioned to receive the data. Usually this will be the first element of the array.

ArraySize the number of data values that the function should attempt to retrieve and place in the array pointed to by ArrayStart. The array should be sized large enough to receive as many values as are indicated by the starting location in ArrayStart and the number of values specified here. The number of records from which to collect values is equal to the ArraySize.

NbrGot indicates the number of values actually retrieved, and returned in the 4-byte floating point array. The values returned in this array do not include time stamps or record numbers.

RecNum indicates the record number from which data was retrieved, if one value was retrieved. If multiple values were retrieved, indicates the last record number from

which a value appears in the return arrays. If no values were retrieved, returns –1.

Applies to GetRecentValuesTS( ) only:

TSStart pointer to the first location within the 8-byte floating point array that has been pre-dimensioned to receive timestamp information. Usually this will be the first element of the array; in any event this array should have the same number of elements and starting index location as the 4-byte floating point values array.

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.
2 = No data exists or bad table name. Also returned if ArraySize is larger than the entire number of records currently residing in the table.
3 = Bad field name.

Example:

Basic examples are provided here. For more useful examples, refer to the PC9000Test program example code which is included on the PC9000.DLL developer kit disk.

```
Sub GetValues1()

Dim iRslt As Integer
Dim sTblName As String
Dim sFldName As String
Dim fVals() As Single
Dim dTS() As Double
Dim iVals As Integer
Dim lRecNum As Long
Dim iNbrGot As Integer
Dim iVal As Integer
Dim bTS As Boolean

On Error GoTo Err_GetVals

   lstRecords.Clear
   lstTS.Clear

   iVals = Int(Val(txtNumVals.Text))
   If iVals <= 0 Then
      Err.Raise 32000, Description:="Enter the number of values " _
        & "you would like to retrieve."
   End If

   sTblName = txtTableName.Text & vbNullChar
   sFldName = txtFieldName.Text & vbNullChar
   ReDim fVals(0 To iVals - 1)
```

(Example continues on next page)

GetValues1() Example (continued from previous page)

```
   bTS = (chkTS.Value <> 0)
   If bTS Then ReDim dTS(0 To iVals - 1)

   If bTS Then
      iRslt = GetRecentValuesTS(sTblName, sFldName, fVals(0), iVals, _
iNbrGot, lRecNum, dTS(0))
   Else
      iRslt = GetRecentValues(sTblName, sFldName, fVals(0), iVals, _
iNbrGot, lRecNum)
   End If

Select Case iRslt
      Case 0
         lblStat.Caption = "Data retrieval successful."
      Case 1
         lblStat.Caption = "Port not open or datalogger does not respond."
      Case 2
         lblStat.Caption = "No data."
      Case 3
         lblStat.Caption = "Bad field name."
      Case 5
         lblStat.Caption = "Errors in data transmission."
      Case Else
         lblStat.Caption = "Unknown result code."
   End Select

   txtRecNum.Text = lRecNum

   For iVal = 1 To iNbrGot
      lstRecords.AddItem fVals(iVal - 1)
      If bTS Then lstTS.AddItem dTS(iVal - 1)
   Next iVal


Exit_GetVals:
Exit Sub

Err_GetVals:
   MsgBox Err.Description
   Resume Exit_GetVals
End Sub
```

2.5.8 GetValuesSinceLast( ) - Get Individual Field Values Beginning at a Specified Record; GetValuesSinceLastTS( ) - Get Field Values with Timestamps

Similar to **GetRecentValues( )**, but provides more control over the precise records retrieved, facilitating "gapless" real time data retrieval, up to the bandwidth limits of the datalogger communications interface and performance limitations of the application program.

GetValuesSinceLastTS( ) is identical to **GetValuesSinceLast ( )** with the exception that an array of timestamp data is also computed and returned.

When the field to be retrieved is an element of an array, the FieldName syntax rules are not completely intuitive. Refer to the documentation for **GetFieldName( )** for a complete description of array field name syntax. As described there, the array field syntax consists of a base name, the programmatically specified array indices, in parentheses (1, 2, or 3-dimensional), and then a one-dimensional array index, in brackets. When fully specifying the array field element in this function, however, the following rules apply:

> Consider a typical Flag field in the Public table. The field names returned by **GetFieldName( )** will be Flag(1)[1], Flag(2)[2], etc. When specifying a particular element of the Flag array in this function, only the second index is important. In other words:
> Flag(2)[2]
> Flag(100)[2]
> Flag()[2]
> All specify the same thing, that is, the second element of the Flag array. The characters within the parentheses are ignored, but the parentheses must be there if the field is in fact part of an array. Passing an out-of-bounds index value within the brackets will cause a "bad field name" error.
> **IMPORTANT:** If the brackets are omitted for a field name which is an array, this FieldName syntax will not cause an error but will be interpreted as specifying the first element of the array.

The return values are placed in the 4-byte floating point array pointed to by ArrayStart. If the ArraySize argument implies that data from more than one record is desired, subsequent records will appear immediately following the values for the first record. Values from multiple records will always be ordered starting with the earliest record to the most recent record.

Refer to the documentation for **GetRecentRecords( )** and **GetRecordsSinceLast( )**, for a detailed description of how to set and interpret record number pointers when performing continuous real time data collection. The behavior of the record number arguments and return codes in **GetRecentValues( )** and **GetValuesSinceLast( )** is identical to those functions.

The method used to initialize the RecNbr argument before the first call to **GetValuesSinceLast()**, will depend upon the type of data collection that is desired:

- To collect all records in the table, initially set the RecNbr value to zero. The value returned will correspond to where in the table that the function found the earliest records still available.

- To collect records starting with the current record and moving forward as new records are stored, make a one-record call to **GetRecentValues( )**, to determine the current record number.

On all subsequent calls, set RecNbr equal to the value returned by the previous call + 1 (unless no data was returned on the last call, in which case the RecNbr value should not be incremented).

The potential speed improvements to be realized from using **GetRecentValues( )/ GetValuesSinceLast( )** are greatly affected by one

limitation of the underlying data retrieval protocol. Ideally, when requesting values from an individual field (as opposed to an entire record), only those field values requested will be transmitted by the datalogger, thereby enabling the most optimum bandwidth usage possible. The CSI protocols treat all elements from an array as a single field, however; if the field requested is part of an array, all the values from that array will be transmitted. When dealing with very large arrays, the performance gains realized from using these functions will therefore be minimal compared to using the full-record retrieval functions. (If multiple individual values calls are made, the performance may actually be slower.)

As a result, if it is desired to obtain the fastest real time performance possible using this DLL, the associated CRBasic programs should set up the key values to be monitored into non-array field locations. On the other hand, if the data retrieval strategy in general involves the optimizing of high-speed data retrieval from very large arrays (such as is typical with FFT results), consider using the **GetPartialFieldValues( )** and **GetPartialFieldArray( )** data retrieval functions. These two functions were specifically written for those purposes.

GetValuesSinceLastTS( ) will return one timestamp value for every data value retrieved.

Declaration:

```
Declare Function GetValuesSinceLast Lib
"PC9000.DLL" (ByVal TableName As String, ByVal
FieldName As String, RecNbr As Long, ArrayStart
As Single, ByVal ArraySize As Integer, NbrGot As
Integer) As Integer

Declare Function GetValuesSinceLastTS Lib
"PC9000.DLL" (ByVal TableName As String, ByVal
FieldName As String, RecNbr As Long, ArrayStart
As Single, ByVal ArraySize As Integer, NbrGot As
Integer, TSStart As Double) As Integer
```

Parameters:

TableName name of the table from which the specified values are read.

FieldName name of a field in the specified table.

RecNbr specifies the record number from which to attempt to begin retrieving values. Returns with the last record number from which values were retrieved.

ArrayStart pointer to the first location within the 4-byte floating point array that has been pre-dimensioned to receive the data. Usually this will be the first element of the array.

ArraySize the number of data values that the function should attempt to retrieve and place in the array pointed to by ArrayStart. The array should be sized large enough to receive as many values as are indicated by the starting location in ArrayStart and the number of values specified here. The number of records from which to collect values is equal to the ArraySize.

NbrGot indicates the number of values actually retrieved, and returned in the 4-byte floating point array. The values returned in this array do not include time stamps or record numbers.

Applies to GetValuesSinceLastTS( ) only:

TSStart pointer to the first location within the 8-byte floating point array that has been pre-dimensioned to receive timestamp information. Usually this will be the first element of the array; in any event this array should have the same number of elements and starting index location as the 4-byte floating point values array.

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.
2 = No data exists or bad table name.
3 = Bad field name.

Example:

Basic examples are provided here. For more useful examples, refer to the PC9000Test program example code which is included on the PC9000.DLL developer kit disk.

```
Sub GetValues2()

Dim iRslt As Integer
Dim sTblName As String
Dim sFldName As String
Dim fVals() As Single
Dim dTS() As Double
Dim iVals As Integer
Dim lRecNum As Long
Dim iNbrGot As Integer
Dim iVal As Integer
Dim bTS As Boolean

On Error GoTo Err_GetVals

   lstRecords.Clear
   lstTS.Clear

   iVals = Int(Val(txtNumVals.Text))
   If iVals <= 0 Then
      Err.Raise 32000, Description:="Enter the number of values " _
        & "you would like to retrieve."
   End If

   sTblName = txtTableName.Text & vbNullChar
   sFldName = txtFieldName.Text & vbNullChar
   ReDim fVals(0 To iVals - 1)
```

(Example continues on next page)

GetValues2() Example (continued from previous page)

```
   bTS = (chkTS.Value <> 0)
   If bTS Then ReDim dTS(0 To iVals - 1)

   lRecNum = Int(Val(txtRecNum.Text))
   If bTS Then
      iRslt = GetValuesSinceLastTS(sTblName, sFldName, lRecNum, _
fVals(0), iVals, iNbrGot, dTS(0))
   Else
      iRslt = GetValuesSinceLast(sTblName, sFldName, lRecNum, _
fVals(0), iVals, iNbrGot)
   End If

Select Case iRslt
      Case 0
         lblStat.Caption = "Data retrieval successful."
      Case 1
         lblStat.Caption = "Port not open or datalogger does not respond."
      Case 2
         lblStat.Caption = "No data."
      Case 3
         lblStat.Caption = "Bad field name."
      Case 5
         lblStat.Caption = "Errors in data transmission."
      Case Else
         lblStat.Caption = "Unknown result code."
   End Select

txtRecNum.Text = lRecNum

   For iVal = 1 To iNbrGot
      lstRecords.AddItem fVals(iVal - 1)
      If bTS Then lstTS.AddItem dTS(iVal - 1)
   Next iVal


Exit_GetVals:
Exit Sub

Err_GetVals:
   MsgBox Err.Description
   Resume Exit_GetVals
End Sub
```

2.5.9 GetPartialFieldValues( ) - Get Part of an Array from the Specified Table and Field

GetPartialFieldValues( ) and **GetPartialFieldArray( )** are two similar functions, both provided to optimize the retrieval of large arrays. **GetPartialFieldArray( )** provides the absolute optimum obtainable speed, but uses a data retrieval strategy which does not provide record number information. **GetPartialFieldValues( )** is less optimized but does provide the record number. Both functions are limited to retrieving the current record only.

GetPartialFieldValues( ) is different from other data retrieval functions in that it requires the field number rather than a field name or sequential index value. A unique sequential field number is assigned by the dataloggers, starting at one, for each unique field or field array in the table: each array only counts for one number. This is best illustrated by the following example of a hypothetical Public Table:

Full Field Name:	Field Index:	Field Number:	Field Array Index:	Base Field Name:
BattVolt	1	1	1	BattVolt
Temp1	2	2	1	Temp1
Flag(1)[1]	3	3	1	Flag
Flag(2)[2]	4	3	2	Flag
Flag(3)[3]	5	3	3	Flag
Temp2	6	4	1	Temp2
Temp1_Hst(1)[1]	7	5	1	Temp1_Hst
Temp1_Hst(2)[2]	8	5	2	Temp1_Hst
Temp1_Hst(3)[3]	9	5	3	Temp1_Hst
Temp1_Hst(4)[4]	10	5	4	Temp1_Hst

Field numbers are not directly obtainable from DLL function calls. They must be derived, usually within a loop of **GetFieldName( )** calls, based upon detecting the occurrences of new base field names using standard Visual Basic string handling functions.

GetPartialFieldValues( ) optimizes performance by only retrieving data for the specified array, not the entire record. It returns only those values within the array, starting at the specified field array index value, for as many values as specified by ArraySize, provided that all the values are part of the same field array. The CSI protocol commands used to retrieve the data will always retrieve the entire array, however, even if only a part of the array is specified to be returned.

GetPartialFieldValues( ) only returns information from the current record. Any record number specified will be ignored and overwritten with the actual number obtained.

It is possible to sample a portion of a CRBasic internal array to a final storage table, wherein the first field array index that appears in the table for the array in question is a number greater than one. In that event, using this function to request array values for index values that are positive integers but are less than the first index value saved to the table, will not generate errors, but rather the values for the non-applicable index elements will be meaningless.

Declaration:

```
Declare Function GetPartialFieldValues Lib
"PC9000.DLL" (ByVal TableName As String, ByVal
FieldNbr As Integer, ByVal StartIndex As Integer,
ByVal ArraySize As Integer, ArrayStart As Single,
ByRef NbrGot As Integer, ByRef RecNbr As Long) As
Integer
```

Parameters:

TableName name of the table from which the field array values are read.

FieldNbr the number of the field in the current table. This is not the same as the field index: see discussion in section above.

StartIndex the field array index value, i.e., the index *within* the array. This is equivalent to the number in brackets within the field name, for the first element which is to be returned in the floating point values array.

ArraySize the number of data values that the function should attempt to retrieve and place in the array pointed to by ArrayStart. The array should be sized large enough to receive as many values as are indicated by the starting location in ArrayStart and the number of values specified here. Bear in mind that this function will not retrieve data across field or record boundaries, but will only return results up to the end of the array specified by the FieldNbr. Sizing the array in a manner that will cause the function to go beyond the end of the specified array will result in an error.

ArrayStart pointer to the first location within the 4-byte floating point array that has been pre-dimensioned to receive the data. Usually this will be the first element of the array.

NbrGot indicates the number of values actually retrieved, and returned in the 4-byte floating point array. The values returned in this array do not include time stamps or record numbers.

RecNbr indicates the record number from which data was retrieved. If no values were retrieved, returns –1.

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.
2 = No data exists or bad table name.
4 = Array index out of bounds (i.e., either the StartIndex argument itself is outside of array bounds, or, one or more of the requested array elements (starting from StartIndex and counting upward ArraySize # of elements), extends beyond the upper bound of the specified variable.

Example:

No examples provided in this document. The PC9000Test program code provides an example of the use of this function.

2.5.10 GetPartialFieldArray( ) - Get Part of an Array from the Specified Table and Field

GetPartialFieldValues( ) and **GetPartialFieldArray( )** are two similar functions, both provided to optimize the retrieval of large arrays. **GetPartialFieldArray( )** provides the absolute optimum obtainable speed, but uses a data retrieval strategy which does not provide record number information. **GetPartialFieldValues( )** is less optimized but does provide the record number. Both functions are limited to retrieving the current record only.

GetPartialFieldArray( ) only retrieves data for the specified array, and only those values requested, not the entire array. This is particularly useful when zooming in on a portion of a very large FFT array or histogram. "Very large" typically corresponds to more than about 100-200 records on a direct serial port connection and more than about 400-500 records on a parallel port or Ethernet connection. Performance will vary from computer to computer and from operating system to operating system.

It returns only those values within the array, starting at the specified field array index value, for as many values as specified by ArraySize, provided that all the values are part of the same field array.

It is possible to sample a portion of a CRBasic internal array to a final storage table, wherein the first field array index that appears in the table for the array in question is a number greater than one. In that event, using this function to request array values for index values that are positive integers, but are less than the first index value saved to the table, will generate array index out of bounds errors.

GetPartialFieldValues( ) only returns information from the current record. Due to the CSI protocol commands used to retrieve the data, the record number is not available using this function.

Declaration:

```
Declare Function GetPartialFieldArray Lib
"PC9000.DLL" (ByVal TableName As String, ByVal
FieldName As String, ByVal StartIndex As Long,
ByVal ArraySize As Long, ArrayStart As Single) As
Integer
```

Parameters:

TableName name of the table from which the field array values are read.

FieldName the base name of the field array, without any indices.

StartIndex the field array index value, i.e., the index *within* the array. This is equivalent to the number in brackets within the field name, for the first element which is to be returned in the floating point values array.

ArraySize the number of data values that the function should attempt to retrieve and place in the array pointed to by ArrayStart. The array should be sized large enough to receive as many values as are indicated by the starting

location in ArrayStart and the number of values specified here. Bear in mind that this function will not retrieve data across field or record boundaries, but will only return results up to the end of the array specified by the FieldNbr. Sizing the array in a manner that will cause the function to go beyond the end of the specified array will result in an error.

ArrayStart pointer to the first location within the 4-byte floating point array that has been pre-dimensioned to receive the data. Usually this will be the first element of the array.

Return Codes:

0 = OK.
1 = Port not open or datalogger does not respond.
2 = No data exists.
3 = Bad field name.
4 = Array index out of bounds (i.e., either the StartIndex argument itself is outside of array bounds, or, one or more of the requested array elements (starting from StartIndex and counting upward ArraySize # of elements), extends beyond the upper bound of the specified variable.

Example:

No examples provided in this document. The PC9000Test program code provides an example of the use of this function.

2.5.11 LogTable( ) - Log Table Contents to a PC Disk File

Writes records from a specified table to the specified file on the PC disk. File writes operate in append mode, to allow for continuous background data logging.

Files are recorded in CSI standard formats, either comma delimited ASCII or binary. All field values are properly logged using this function, regardless of data type.

To store all data as it is logged:

- On the first call to **LogTable( )**, set StartAt to either 1 or 2 to select a starting location. RecNbr and RecCnt values are don't care.
- On subsequent calls to **LogTable()**, set StartAt to 0 and pass in the same value in RecNbr that was passed back on the last call to **LogTable( ).** Set RecCnt to -1 so that it will not limit collection.

To store the most recent n records:

- Make a first call to **LogTable( )** with StartAt set to 3 and RecCnt to the number of records wanted. **LogTable( )** will back up from the current location as far as is specified by RecCnt and will begin collection from that point.
- **LogTable( )** returns the number of records collected via RecCnt. If less than wanted was collected, then call **LogTable( )** again with "StartAt" set to 0, RecNbr at the value that was passed back last time, and RecCnt set to a value that indicates how many more records are wanted.

Data discontinuity codes will be returned if the next records requested are not available. Even if there is a discontinuity in the record numbers, data is still logged.

Declaration:

```
Declare Function LogTable Lib "PC9000.DLL" (ByVal
FileName As String, ByVal TableName As String,
ByVal dstType As Integer, ByVal StartAt As
Integer, RecNbr As Long, RecCnt As Long) As
Integer
```

Parameters:

FileName specifies the name of the destination file.

TableName specifies the table from which the records are read.

DstType specifies the format for the file:
0 = binary with time.
1 = ASCII.
2 = binary values only.
3 = ASCII without time stamps.

StartAt specifies where to begin reading records:
0 = start at "RecNbr" and collect to end of table.
1 = start at oldest record in table and collect to end of table.
2 = start at current record and collect to end of table.
3 = start "RecCnt" records back from the current location.

RecNbr is the number of the record.

RecCnt is the quantity of records wanted.

Return Codes:

0 = done.
1 = datalogger does not respond.
3 = discontinuity in data.
5 = could not open the file or disk full.

Example:

There are numerous ways in which this function can be used. The example given here outlines a timer-driven routine, intended to run in the background, logging data in ASCII format. It periodically calls LogTable( ) to retrieve whatever new data is available, picking up from where it left off the last time the timer event fired. A series of comments in the declarations code should provide the necessary details associated with initializing the timer routine.

```
' Module-level declarations:

' This is explicit Drive:\path\filename of the file to which
' data is being logged.  The example code does not show how
' this file name was originally assigned: this is entirely
' up to the application to determine.
Private msWriteFile as String
' Same as above, also applies to the table name.
Private msTableName as String

' Current record number pointer, preserved between calls
Private mlRecNbr as Long

' Collection type.  Usually would be initialized outside of the timer
' routine to a value of 1 or 2 for the first collection, depending upon
' the type of collection to perform (initialization code not shown here.)
Private miCollectType as Integer

Private Sub tmrWrite_Timer()

' NOTE: may want to add re-entrance protection to this routine.
Dim iType As Integer
Dim lRecCnt As Long
Dim iRslt as Integer
Dim bPathTooLong As Boolean

   If Len(msWriteFiile) > 126 Then bPathTooLong = True
   iType = 1  'ascii data
   lRecCnt = -1
   iRslt = LogTable(msWriteFile, msTableName, iType, miCollectType, _
mlRecNbr, lRecCnt)

   miCollectType = 0   ' after first pass, collect from where it left off

   Select Case iRslt
Case 0
         lblLog.Caption = msWriteFile & " Record Number: " & mlRecNbr
      Case 1
         lblLog.Caption = " Datalogger does not respond."
      Case 3
         lblLog.Caption = " Discontinuity in " & WriteFileName$ & " data."
      Case 5
         If bPathTooLong Then
            lblLog.Caption = " File path too long: " & msWriteFile
         Else
            lblLog.Caption = " Could not open " & msWriteFile
         End If
  End Select
End Sub
```

2.6 Miscellaneous Utility Functions

2.6.1 GetCR9KApiVers( ) - Get Extended Version Information Regarding PC9000.DLL

This function can be very useful for tracking in the case where two different versions of the DLL are in use (such as one for PC9000, one for custom applications). The release date will always be unique for each released version.

Declaration:

```
Declare Function GetCR9KApiVers Lib "PC9000.DLL"
(ByVal ApiVer As String, ByVal ApiItem As String,
ByVal ApiDate As String) As Integer
```

Parameters:

ApiVer returns the version number of PC9000.DLL.

ApiItem returns the internal CSI product tracking number for the PC9000.DLL

ApiDate returns the release date of the PC9000.DLL version in use: the date is always a six-character string in YYMMDD format.

Return Codes:

Always returns 0.

Example:

```
Dim sDLLVers as String
Dim sDLLItem as String
Dim sDLLDate as String
Dim iRslt as Integer

sDLLVers = String(20, vbNullChar)
sDLLItem = String(20, vbNullChar)
sDLLDate = String(20, vbNullChar)

iRslt = GetCR9KApiVers(sDLLVers, sDLLItem, sDLLDate)
sDLLVers = Trim$(sDLLVers)
sDLLItem = Trim$(sDLLItem)
sDLLDate = Trim$(sDLLDate)
MsgBox "PC9000.DLL Version:  " & sDLLVers & vbCrLf _
 & "Product#  " & sDLLItem & vbCrLf & "Date:  " & sDLLDate
```

2.6.2 FP2ToSingle( ) – Converts a CSI 2-byte Floating Point Value to IEEE 4-byte Float

This function is useful primarily for performing conversions on CSI TOB1 or TOB2 binary files read in from disk, for such functions as plotting data. It is not needed for converting any information passed to the application by the DLL in real time data retrieval functions: any necessary conversions in those cases are done within the function.

Declaration:

```
Declare Function FP2toSingle Lib "PC9000.DLL"
(ByVal High As String, ByVal Low As String) As
Single
```

Parameters:

High The high byte of the two-byte binary FP2 value, passed as a single string character.

Low The low byte of the two-byte binary FP2 value, passed as a single string character

Return Codes:

Returns converted value.

Example:

Any example which used this function to convert information from within a binary file would be not only extremely complex but would also be totally specific to the organization of that particular file. The example provides a simple piece of user test code which illustrates how the function works.

```
Private Sub cmdFP2ToSingle_Click()
' txtFP2High and txtFP2Low are text boxes with
' their MaxLength property set=2.

Dim sCompare As String
Dim sHighByte As String * 1
Dim sLoByte As String * 1
Dim fResult As Single

sCompare = "[0-9ABCDEF][0-9ABCDEF]"
If (txtFP2High.Text Like sCompare) And _
 (txtFP2Low.Text Like sCompare) Then
   sHighByte = Chr$(CLng("&H" & txtFP2High.Text))
   sLoByte = Chr$(CLng("&H" & txtFP2Low.Text))
   fResult = FP2toSingle(sHighByte, sLoByte)
   txtFP2Result.Text = fResult
Else
   MsgBox "High and Low Bytes must each be " _
    & "valid two-character HEX numbers (00 - FF)"
End If

End Sub
```

2.6.3 LongFromString( ) – Loads a 4-byte Packed String into a Long Integer Variable

This function is useful primarily for performing conversions on CSI TOB1 or TOB2 binary files read in from disk, for such functions as plotting timestamps. It is not needed for converting any information passed to the application by the DLL in real time data retrieval functions: any necessary conversions in those cases are done within the function.

LongFromString( ) performs the same task as the "CopyMemory" or RTLMoveMemory Windows API function. It has a different call level

interface and is safe from the GPFs that automatically result when using the CopyMemory function incorrectly.

Declaration:

```
Declare Function LongFromString Lib "PC9000.DLL"
(ByVal strg As String) As Long
```

Parameters:

strg The 4-character packed binary string to convert. Any characters beyond 4 are ignored. The string is read low byte first.

Return Codes:

Returns converted value.

Example:

Any example which used this function to convert information from within a binary file would be not only extremely complex but would also be totally specific to the organization of that particular file. The example illustrates how the function works.

```
Dim lRslt as Long

' This statement returns a value of 0.
lRslt = LongfromString(Chr$(0)& Chr$(0)& Chr$(0)& Chr$(0))

' This statement returns the largest positive value (2^32 –1)
lRslt = LongFromString(Chr$(255)& Chr$(255)& Chr$(255)& Chr$(127))

' This statement returns the largest negative value (-2^32)
lRslt = LongFromString(Chr$(0)& Chr$(0)& Chr$(0)& Chr$(128))

' This statement returns a value of –1.
lRslt = LongFromString(Chr$(255)& Chr$(255)& Chr$(255)& Chr$(255))
```

2.6.4 SingleFromString( ) – Loads a 4-byte Packed String into a 4-byte Float Variable

This function is useful primarily for performing conversions on CSI TOB1 or TOB2 binary files read in from disk, for such functions as plotting timestamps. It is not needed for converting any information passed to the application by the DLL in real time data retrieval functions: any necessary conversions in those cases are done within the function.

SingleFromString( ) performs the same task as the "CopyMemory", the common alias for the RTLMoveMemory Windows API function. It has a different call level interface and is safe from the GPFs that automatically result when using the CopyMemory function incorrectly.

Declaration:

```
Declare Function SingleFromString Lib
"PC9000.DLL" (ByVal strg As String) As Single
```

Parameters:

strg The 4-character packed binary string to convert. Any characters beyond 4 are ignored. The string is read low byte first.

Return Codes:

Returns converted value.

Example:

No examples provided.

2.6.5 RdStatus ( ) - Read a String from the DLL's Internal Status Message Queue

Allows for the monitoring of internal status messages detected by DLL functions, which in turn are generated by the Windows API during various datalogger communications tasks. Messages may be useful for debugging purposes.

Declaration:

```
Declare Function RdStatus Lib "PC9000.DLL" (ByVal
StatusMsg As String, ByVal StatusMsgSize As
Integer) As Integer
```

Parameters:

StatusMsg String returned by the function, corresponding to the next message in the queue.

StatusMsgSize declares the size of the status message string buffer set up by the calling routine.

Return Codes:

0 = done.

1 = queue was empty so no string was returned.

Example:

```
Dim sBuf As String
Dim iRslt as Integer

' lstStatus is a VB ListBox control in which to display the
' status messages.

lstStatus.Clear
Do
   sBuf = String(80, vbNullChar)
   iRslt = RdStatus(sBuf, Len(sBuf))
   If iRslt = 0 Then lstStatus.AddItem Trim$(sBuf)
Loop Until iRslt <> 0
```

Section 3. Function Declarations

This section lists the declarations that must be made before making a call to that function in PC9000.DLL. These declarations are ordinarily placed in a separate module such as CR9000.BAS.

```
Declare Function OpenCom Lib "PC9000.DLL" (ByVal Port As String,
curBaudRate As Long, ByVal ExtraResp As Long, ByVal maxPktSize As
Integer, ByVal BestPktSize As Integer, ByVal ModemOn As Integer)
As Integer

Declare Function OpenLpt Lib "PC9000.DLL" (ByVal LptName As
String, ByVal ExtraResp As Long, ByVal maxPktSize As Integer,
ByVal BestPktSize As Integer) As Integer

Declare Function OpenCSICard Lib "PC9000.DLL" (ByVal PortNbr As
Integer, ByVal ExtraResp As Long, ByVal maxPktSize As Integer,
ByVal BestPktSize As Integer) As Integer

Declare Function OpenSock Lib "PC9000.DLL" (ByVal ipAddr As
String, ByVal IPPort As String, ByVal ExtraResp As Long, ByVal
maxPktSize As Integer, ByVal BestPktSize As Integer) As Integer

Declare Function GetModemStatus Lib "PC9000.DLL" (ByRef StatusWord
as Long) As Integer

Declare Function ClosePort Lib "PC9000.DLL" () As Integer

Declare Function GetLgrIdent Lib "PC9000.DLL" (BmpVer As Integer,
Model As Integer, SerNbr As Long, ByVal StnName As String, ByVal
StnNameSize As Integer) As Integer

Declare Function SetLgrName Lib "PC9000.DLL" (ByVal StnName As
String) As Integer

Declare Function SetLgrClock Lib "PC9000.DLL" (ByVal SetIt As
Integer, DYear As Integer, DMonth As Integer, DDay As Integer,
DHour As Integer, DMinute As Integer, DSecond As Integer) As
Integer

Declare Function CR9000Dial Lib "PC9000.DLL" (ByVal DialString As
String) As Integer

Declare Function CR9000HangUp Lib "PC9000.DLL" () As Integer

Declare Function StartIOLog Lib "PC9000.DLL" () As Integer

Declare Function StopIOLog Lib "PC9000.DLL" () As Integer

Declare Function UserRd Lib "PC9000.DLL" (ByVal Buf As String,
ByVal BufSize As Integer) As Integer

Declare Function UserWr Lib "PC9000.DLL" (ByVal Buf As String,
ByVal BufSize As Integer) As Integer

Declare Function BootFromLinkStart Lib "PC9000.DLL" (ByVal OsName
As String, NbrWr As Long) As Integer

Declare Function BootFromLinkMore Lib "PC9000.DLL" (NbrWr As Long)
As Integer

Declare Function GetDirectory Lib "PC9000.DLL" (ByVal FileName As
String, ByVal FileNameSize As Integer, Attrib As Integer) As
Integer

Declare Function DownloadStart Lib "PC9000.DLL" (ByVal FileName As
String, ByVal DevName As String, ByVal Options As Integer, ByVal
SendFile As Integer) As Integer

Declare Function DownloadWait Lib "PC9000.DLL" (ByVal Result As
String, ByVal ResultSize As Integer, DYear As Integer, DMonth As
Integer, DDay As Integer, DHour As Integer, DMinute As Integer,
DSecond As Integer) As Integer
```

```
Declare Function UploadFile Lib "PC9000.DLL" (ByVal FileName As
String, ByVal DestFileName As String, ByVal DevName As String) As
Integer

Declare Function UploadStart Lib "PC9000.DLL" (ByVal FileName As
String, ByVal DestFileName As String, ByVal DevName As String) As
Integer

Declare Function UploadWait Lib "PC9000.DLL" (BytesSent As Long)
As Integer

Declare Function UploadStop Lib "PC9000.DLL" () As Integer

Declare Function GetTableName Lib "PC9000.DLL" (ByVal TableName As
String, ByVal TableNameSize As Integer, TableSize As Long) As
Integer

Declare Function GetTableName2 Lib "PC9000.DLL" (ByVal TableName
As String, ByVal TableNameSize As Integer, TableSize As Long,
Seconds As Long, MicroSeconds As Long, TableSig As Long) As
Integer

Declare Function GetFieldName Lib "PC9000.DLL" (ByVal TableName As
String, ByVal FieldName As String, ByVal FieldnameSize As Integer,
ByVal Units As String, ByVal UnitsSize As Integer, ByVal Proc As
String, ByVal ProcSize As Integer) As Integer

Declare Function GetFieldName2 Lib "PC9000.DLL" (ByVal TableName
As String, ByVal FieldName As String, ByVal FieldnameSize As
Integer, ByVal Units As String, ByVal UnitsSize As Integer, ByVal
Proc As String, ByVal ProcSize As Integer, ByVal DataType as
Integer) As Integer

Declare Function TableCtrl Lib "PC9000.DLL" (ByVal ROption As
Integer, ByVal TableName As String) As Integer

Declare Function GetVariable Lib "PC9000.DLL" (ByVal FieldName As
String, Value As Single) As Integer

Declare Function SetVariable Lib "PC9000.DLL" (ByVal FieldName As
String, ByVal Value As Single) As Integer

Declare Function GetCurrentValue Lib "PC9000.DLL" (ByVal TableName
As String, ByVal FieldName As String, ByVal ValueBuf As String,
ByVal ValueBufSize As Integer) As Integer

Declare Function GetStatusValue Lib "PC9000.DLL" (ByVal FieldName
As String, ByVal ValueBuf As String, ByVal ValueBufSize As
Integer, ByVal Refresh as Integer) As Integer

Declare Function GetRecentRecords Lib "PC9000.DLL" (ByVal
TableName As String, ArrayStart As Single, ByVal ArraySize As
Integer, NbrGot As Integer, RecNum As Long) As Integer

Declare Function GetRecentRecordsTS Lib "PC9000.DLL" (ByVal
TableName As String, ArrayStart As Single, ByVal ArraySize As
Integer, NbrGot As Integer, RecNum As Long, TSStart As Double) As
Integer

Declare Function GetRecordsSinceLast Lib "PC9000.DLL" (ByVal
TableName As String, RecNbr As Long, ArrayStart As Single, ByVal
ArraySize As Integer, NbrGot As Integer) As Integer

Declare Function GetRecordsSinceLastTS Lib "PC9000.DLL" (ByVal
TableName As String, RecNbr As Long, ArrayStart As Single, ByVal
ArraySize As Integer, NbrGot As Integer, TSStart As Double) As
Integer

Declare Function GetRecentValues Lib "PC9000.DLL" (ByVal TableName
As String, ByVal FieldName As String, ArrayStart As Single, ByVal
ArraySize As Integer, NbrGot As Integer, RecNum as Long) As
Integer

Declare Function GetRecentValuesTS Lib "PC9000.DLL" (ByVal
TableName As String, ByVal FieldName As String, ArrayStart As
Single, ByVal ArraySize As Integer, NbrGot As Integer, RecNum As
Long, TSStart As Double) As Integer
```

```
Declare Function GetValuesSinceLast Lib "PC9000.DLL" (ByVal
TableName As String, ByVal FieldName As String, RecNbr As Long,
ArrayStart As Single, ByVal ArraySize As Integer, NbrGot As
Integer) As Integer

Declare Function GetValuesSinceLastTS Lib "PC9000.DLL" (ByVal
TableName As String, ByVal FieldName As String, RecNbr As Long,
ArrayStart As Single, ByVal ArraySize As Integer, NbrGot As
Integer, TSStart As Double) As Integer

Declare Function GetPartialFieldValues Lib "PC9000.DLL" (ByVal
TableName As String,  ByVal FieldNbr As Integer, ByVal StartIndex
As Integer, ByVal ArraySize As Integer,  ArrayStart As Single,
ByRef NbrGot As Integer, ByRef RecNbr As Long) As Integer

Declare Function GetPartialFieldArray Lib "PC9000.DLL" (ByVal
TableName As String,  ByVal FieldName As String, ByVal StartIndex
As Long, ByVal ArraySize As Long,  ArrayStart As Single) As
Integer

Declare Function LogTable Lib "PC9000.DLL" (ByVal FileName As
String, ByVal TableName As String, ByVal dstType As Integer, ByVal
StartAt As Integer, RecNbr As Long, RecCnt As Long) As Integer

Declare Function GetCR9KApiVers Lib "PC9000.DLL" (ByVal ApiVer As
String, ByVal ApiItem As String, ByVal ApiDate As String) As
Integer

Declare Function FP2toSingle Lib "PC9000.DLL" (ByVal High As
String, ByVal Low As String) As Single

Declare Function LongFromString Lib "PC9000.DLL" (ByVal strg As
String) As Long

Declare Function SingleFromString Lib "PC9000.DLL" (ByVal strg As
String) As Single

Declare Function RdStatus Lib "PC9000.DLL" (ByVal StatusMsg As
String, ByVal StatusMsgSize As Integer) As Integer
```

Index

Campbell Scientific Companies

Campbell Scientific, Inc. (CSI)
815 West 1800 North
Logan, Utah 84321
UNITED STATES
www.campbellsci.com
info@campbellsci.com

Campbell Scientific Africa Pty. Ltd. (CSAf)
PO Box 2450
Somerset West 7129
SOUTH AFRICA
www.csafrica.co.za
sales@csafrica.co.za

Campbell Scientific Australia Pty. Ltd. (CSA)
PO Box 444
Thuringowa Central
QLD 4812 AUSTRALIA
www.campbellsci.com.au
info@campbellsci.com.au

Campbell Scientific do Brazil Ltda. (CSB)
Rua Luisa Crapsi Orsi, 15 Butantã
CEP: 005543-000 São Paulo SP BRAZIL
www.campbellsci.com.br
suporte@campbellsci.com.br

Campbell Scientific Canada Corp. (CSC)
11564 - 149th Street NW
Edmonton, Alberta T5M 1W7
CANADA
www.campbellsci.ca
dataloggers@campbellsci.ca

Campbell Scientific Ltd. (CSL)
Campbell Park
80 Hathern Road
Shepshed, Loughborough LE12 9GX
UNITED KINGDOM
www.campbellsci.co.uk
sales@campbellsci.co.uk

Campbell Scientific Ltd. (France)
Miniparc du Verger - Bat. H
1, rue de Terre Neuve - Les Ulis
91967 COURTABOEUF CEDEX
FRANCE
www.campbellsci.fr
campbell.scientific@wanadoo.fr

Campbell Scientific Spain, S. L.
Psg. Font 14, local 8
08013 Barcelona
SPAIN
www.campbellsci.es
info@campbellsci.es

Please visit www.campbellsci.com to obtain contact information for your local US or International representative.