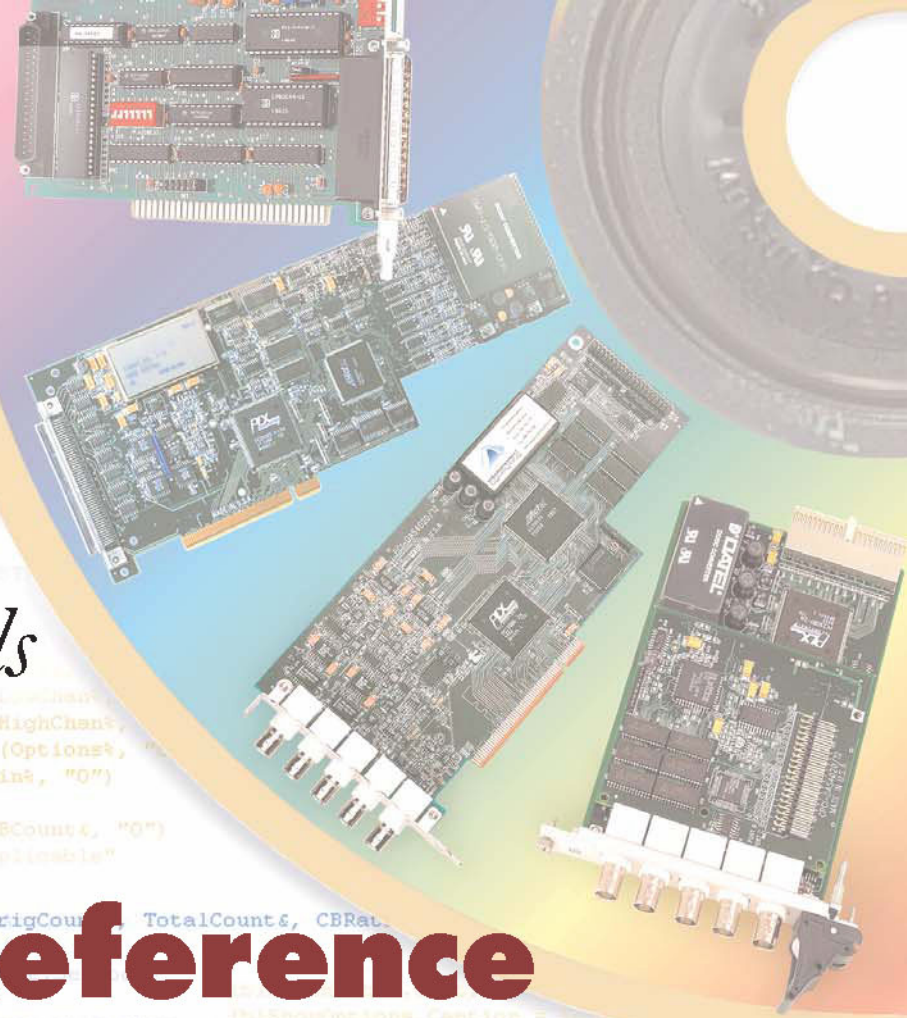
UNIVERSAL LIBRARY™

Data Acquisition and Control Programming Tools



Function Reference

MEASUREMENTCOMPUTING™ www.mccdaq.com

Universal Library™

Function Reference

MEASUREMENT COMPUTING™

Document Revision 6.5, July, 2005

Your new Measurement Computing product comes with a fantastic extra —

Management committed to your satisfaction!

Refer to www.mccdaq.com/execteam.html for the names, titles, and contact information of each key executive at Measurement Computing.

Thank you for choosing a Measurement Computing product—and congratulations! You own the finest, and you can now enjoy the protection of the most comprehensive warranties and unmatched phone tech support. It's the embodiment of our two missions:

- To offer the highest-quality, computer-based data acquisition, control, and GPIB hardware and software available—at the best possible price.
- To offer our customers superior post-sale support—FREE. Whether providing unrivaled telephone technical and sales support on our latest product offerings, or continuing that same first-rate support on older products and operating systems, we're committed to you!

30 Day Money Back Guarantee: You may return any Measurement Computing Corporation product within 30 days of purchase for a full refund of the price paid for the product being returned. If you are not satisfied, or chose the wrong product by mistake, you do not have to keep it. Please call for an RMA number first. No credits or returns accepted without a copy of the original invoice. Some software products are subject to a repackaging fee.

These warranties are in lieu of all other warranties, expressed or implied, including any implied warranty of merchantability or fitness for a particular application. The remedies provided herein are the buyer's sole and exclusive remedies. Neither Measurement Computing Corp., nor its employees shall be liable for any direct or indirect, special, incidental or consequential damage arising from the use of its products, even if Measurement Computing Corp. has been notified in advance of the possibility of such damages.

Licensing Information

Each original copy of Universal Library is licensed for development use on one CPU at a time. It is theft to make copies of this program for simultaneous program development. If a customer creates an application using the Universal Library, they may distribute the necessary runtime files (Universal Library driver files) with their application royalty free. They may not distribute any files that give their customer the ability to develop applications using the Universal Library.

Trademark, and Copyright Information

MEGA-FIFO, the CIO prefix to data acquisition board model numbers, the PCM prefix to data acquisition board model numbers, PCM-DAS08, PCM-DAC02, PCM-COM422, PCM-COM485, PCM-DAS16D/12, PCI-DAS6402/16, Universal Library*, InstaCal*, Measurement Computing Corporation, and the Measurement Computing logo are either trademarks or registered trademarks of Measurement Computing Corp.

SoftWIRE and the SoftWIRE logo are registered trademarks of SoftWIRE Technology.

Pentium is a trademark of Intel Corp.

PC is a trademark of International Business Machines Corp.

Microsoft, MS-DOS, Visual Basic, Visual C#, Visual Studio .NET, Windows, and Windows NT are trademarks of Microsoft Corp.

All other trademarks are the property of their respective owners.

Information furnished by Measurement Computing Corp. is believed to be accurate and reliable. However, no responsibility is assumed by Measurement Computing Corp. neither for its use; nor for any infringements of patents or other rights of third parties, which may result from its use. No license is granted by implication or otherwise under any patent or copyrights of Measurement Computing Corp.

Notice

Measurement Computing Corporation does not authorize any Measurement Computing Corporation product for use in life support systems and/or devices without the written approval of the CEO of Measurement Computing Corporation. Life support devices/systems are devices or systems which, a) are intended for surgical implantation into the body, or b) support or sustain life and whose failure to perform can be reasonably expected to result in injury. Measurement Computing Corp. products are not designed with the components required, and are not subject to the testing required to ensure a level of reliability suitable for the treatment and diagnosis of people.

Table of Contents

Universal Library Functions (16-bit and 32-bit)

Universal Library for .NET Classes, Methods, and Properties

Appendix

Universal Library Functions
(16-bit and 32-bit)

Overview – Universal Library (16-bit and 32-bit)

Introduction

This section contains a complete, detailed explanation of all Universal Library functions. This chapter briefly explains each function, and provides you with a general idea of the capability of the Universal Library. We highly recommend that you refer to one of the many example programs provided. These programs present a "hands-on" explanation of the various functions, as well as providing you with a starting point from which to write your own programs.

DOS vs. Windows libraries

The function prototypes shown in this manual are those used in a 32 bit Windows application. The form of these prototypes follows very closely to that of the DOS prototypes. The most noticeable difference is the use of the memory handle argument (MemHandle) in place of the array argument (ADData() for example) seen in the DOS prototypes.

If you are using a DOS platform, use the header files and example programs provided for the DOS language you are using as a guide for the library syntax.

16-bit vs. 32-bit libraries

Universal Library is available in 16- and 32-bit versions. Unless you have a specific reason for using the 16-bit library (such as required compatibility with Windows 3.x or DOS), use the 32-bit library. The two versions are nearly identical, but there are important differences. An explanation of the major differences between the two follows:

- The 32-bit library is compatible with the latest operating systems. Operating systems such as Windows NT and Windows 2000 require the 32-bit interface provided by the 32-bit version of the Universal Library.
- Although most UL functions are supported in both versions of the Universal Library, the 32-bit version has additional features not found in the 16-bit library. Those few functions that are not supported in the 16-bit version of UL are identified in this manual.
- Most UL functions reference a board number. This number is limited to 0 to 9 in the 16-bit version. The 32-bit version of UL supports board numbers from 0 to 99.
- There may be differences in the data types for the 16- and 32-bit versions of a function. For C++ programmers, the majority of the data type differences are handled by the programming environment and no action is required by the user. The differences are more pronounced using Visual Basic. If you are using the 16-bit version of the library, refer to the header files for the language you are using to determine the appropriate data types.

Either the 16-bit or the 32-bit version may be used in Windows 95 and 98 systems. However, Windows 3.x and DOS systems are limited to the 16-bit version.

Again, unless you have specific reasons for using the 16-bit version, we strongly recommend using the 32-bit version.

Analog I/O functions

These functions perform analog input or analog output.

Most PCI boards that support analog input and output scanning allow for simultaneous analog input and output scans (32-bit UL only). However, for most older boards, analog input scans (cbAInScan() and cbAPretrig()) cannot operate while an analog output scan (cbAOutScan()) is active.

- **cbAIn()** - Takes a single reading from an analog input channel (A/D).
- **cbAInScan()** - Repeatedly scans a range of analog input (A/D) channels. You can specify the channel range, the number of iterations, the sampling rate, and the A/D range. The data that is collected is stored in an array.
- **cbALoadQueue()** - Loads a series of chan/gain pairs into A/D board's queue. These chan/gains are used with all subsequent analog input functions.
- **cbAOut()** - Outputs a single value to an analog output (D/A).
- **cbAOutScan()** - Repeatedly scans a range of analog output (D/A) channels. You can specify the channel range, the number of iterations, and the rate. The data values from consecutive elements of an array are sent to each D/A channel in the scan.
- **cbAPretrig()** - Repeatedly scans a range of analog input (A/D) channels waiting for a trigger signal. When a trigger occurs, it returns the specified number of samples and points before the trigger occurred. You can specify the channel range, the sampling rate, and the A/D range. All of the data that is collected is stored in an array.
- **cbATrig()** - Reads the analog input and waits until it goes above or below a specified threshold. When the trigger condition is met, the current sample is returned.
- **cbAConvertData()** - Converts raw analog data into 12-bit A/D values. Each raw sample from analog input is a 16-bit value. For many 12-bit A/D boards, the raw data is a 16-bit value that contains a 12-bit A/D value and a 4-bit channel tag. This function is not intended for use with 16-bit A/D boards.

  This conversion is handled automatically by the `cbAIn()` function. It can also be done automatically by the `cbAInScan()` function with the CONVERTDATA option. In some cases though, it may be useful or necessary to collect the data and then do the conversion sometime later. The `cbAConvertData()` function takes a buffer full of unconverted data and converts it.
- **cbACalibrateData()** - Calibrates analog data. Each raw sample from a board with software calibration factors that must be applied to the sample may be acquired and calibrated, then passed to an array. Alternatively, they can be acquired then passed to the array without calibration. When this second method is used, `cbACalibrateData()` may be used to apply the calibration factors to an array of data after the acquisition is complete. The only case where you would withhold calibration until after the acquisition run was complete is on slower CPUs, or when the processing time is at a premium. Applying calibration factors in real time on a per sample basis does eat up machine cycles.

  To disable the automatic calibration so that you may apply the calibration later, specify the NOCALIBRATEDATA option when collecting data with `cbAInScan()`.
- **cbAConvertPretrigData()** - Converts and re-orders pre-trigger data from data plus channel tags to separate the data and channel tags.

  When data is collected with the `cbAPretrig()` function, the same data conversion needs to be done as is performed by the `cbAConvertData()` function. There is a further complication because `cbAPretrig()` collects analog data into an array. It treats the array like a circular buffer. While it is waiting for the trigger to occur, it fills the array. When it gets to the end it resets to the start and begins again. When the trigger signal occurs it continues collecting data into the circular buffer until the requested number of samples have been collected.

When the data acquisition is complete, all of the data is in the array but it is in the wrong order. The first element of the array does not contain the first data point. The data has to be rotated in the correct order.

This conversion can be done automatically by the `cbAPretrig()` function with the `CONVERTDATA` option. In some cases though, it may be useful or necessary to collect the data and then do the conversion sometime later. The `cbAConvertPretrigData()` function takes a buffer full of unconverted data, converts it, and arranges the data in the correct order.

Configuration functions

The configuration information for all boards is stored in the configuration file CB.CFG. This information is loaded from CB.CFG by all programs that use the library. The library includes the following functions to retrieve or change configuration options:

- **cbGetConfig()** - Returns the current value for a specified configuration option.
- **cbSetConfig()** - Sets the current value for a specified configuration option.
- **cbGetSignal()** - Retrieves the configured auxiliary or DAQ Sync connection and polarity for the specified timing and control signal. This function is intended for advanced users.
- **cbSelectSignal()** - Configures timing and control signals to use specific auxiliary or DAQ Sync connections as a source or destination. This function is intended for advanced users.
- **cbSetTrigger()** - Sets up trigger parameters used with the `EXTTRIGGER` option for `cbAInScan()`.

Counter functions

Counter functions load, read, and configure counters. There are five types of counter chips used in MCC counter boards: 8254's, 8536's, 7266's, 9513's, and generic event counters. Some of the counter commands only apply to one type of counter.

- **cbC7266Config()** - Selects the operating mode of an LS7266 counter. (Not available in 16 bit version of library.)
- **cbC8254Config()** - Selects the operating mode of the 8254 counter.
- **cbC8536Config()** - Selects the operating mode of the 8536 counter.
- **cbC8536Init()** - Initializes and selects all of the chip-level features for a 8536 counter board. The options set by this command are associated with each counter chip, not the individual counters within it.
- **cbC9513Config()** - Sets the operating mode of the 9513 counter. This function sets all of the programmable options that are associated with a 9513 counter. It is similar in purpose to cbC8254Config() except that it is used with a 9513 counter.
- **cbC9513Init()** - Initializes and selects all of the chip level features for a 9513 counter board. The options set by this command are associated with each counter chip, not the individual counters within it.
- **cbCFreqIn()** - Measures the frequency of a signal by counting it for a specified period of time (`GateInterval`), and then converting the count to count/sec (Hz). This function only works with 9513 counters.
- **cbCIn()** - Reads a counter's current value.
- **cbCIn32()** - Reads a counter's current value as a 32-bit integer. Used primarily with LS7266 counters.
- **cbCLoad()** - Loads a counter with an initial count value.
- **cbCLoad32()** -Loads a counter with a 32-bit integer initial value. Used primarily with LS7266 counters.

- **cbCStatus()** - Read the counter status of a counter. Returns various bits that indicate the current state of a counter. (Not available in 16 bit library - currently only applies to LS7266 counters).
- **cbCStoreOnInt()** - Installs an interrupt handler that stores the current count whenever an interrupt occurs. This function only works with 9513 counters.

Digital I/O functions

The digital I/O functions perform digital input and output operations on various types of digital I/O ports.

- **cbDBitIn()** - Reads a single bit from a digital input port.
- **cbDBitOut()** - Sets a single bit on a digital output port.
- **cbDConfigBit()** - Configures a specific digital bit as input or output.
- **cbDConfigPort()** - Selects whether a digital port is an input or an output.
- **cbDIn()** - Reads a specified digital input port.
- **cbDInScan()** - Reads a specified number of bytes or words from a digital input port at a specified rate.
- **cbDOut()** - Writes a byte to a digital output port.
- **cbDOutScan()** - Writes a series of bytes or words to a digital output port at a specified rate.

Error handling functions

All library functions return error codes. The Universal Library includes two functions for handling errors. The different methods built into the functions for handling errors include stopping the program when an error occurs, and printing error messages versus error codes.

- **cbErrHandling()** - Sets the method of reporting and handling errors for all function calls.
- **cbGetErrMsg()** - Returns the error message associated with a specific error code.

Memory board functions

The memory board functions read and write data to and from a memory board, and also set modes that control memory boards (MEGA-FIFO).

The most common use for the memory boards is to store large amounts of data from an A/D board via a DT-Connect cable between the two boards. To do this, use the `EXTMEMORY` option with `cbAInScan()` or `cbAPretrig()`.

Once the data has been transferred to the memory board, you can use the memory functions to retrieve it.

- **cbMemSetDTMode()** - Sets DT-Connect mode on a memory board. Memory boards have a DT-Connect interface which can be used to transfer data through a cable between two boards rather than through the PC's system memory. The DT-Connect port on the memory board can be configured as either an input (from an A/D) or as an output (to a D/A). This function configures the port to one of these settings.
- **cbMemReset()** - Resets the memory board address. The memory board is organized as a sequential device. When data is transferred to the memory board, it is automatically put in the next address location. This function resets the current address to the location 0.
- **cbMemRead()** - Reads a specified number of points from a memory board starting at a specified address.
- **cbMemWrite()** - Writes a specified number of points to a memory board starting at a specified address.

- **cbMemReadPretrig()** - Reads data collected with `cbAPretrig()`. The `cbAPretrig()` function writes the pre-triggered data to the memory board in a scrambled order. This function unscrambles the data and returns it in the correct order.

Revision control functions

As new revisions of the library are released, bugs from previous revisions are fixed and occasionally new functions are added. It is the manufacturers goal to preserve existing programs you have written and therefore to never change the order or number of arguments in a function. However, sometimes it is not possible to achieve this goal.

The revision control function initializes the DLL so that the functions are interpreted according to the format of the revision you wrote and compiled your program in.

- **cbDeclareRevision()** - Declares the revision # of the Universal Library that your program was written with.
- **cbGetRevision()** - Returns the version number of the installed Universal Library.

Streamer file functions

The streamer file functions explained below create, fill, and read streamer files.

- **cbFileAInScan()** - Transfer analog input data directly to file. Very similar to cbAInScan() except that the data is stored in a file instead of an array.
- **cbFilePretrig()** - Pre-triggered analog input to a file. Very similar to cbAPretrig() except that the data is stored in a file instead of an array.
- **cbFileGetInfo()** - Reads streamer file information on how much data is in the file, and the conditions under which it was collected (sampling rate, channels, etc.).
- **cbFileRead()** - Reads a selected number of data points from a streamer file into an array.

Temperature input functions

The temperature sensor functions convert a raw analog input from an EXP or other temperature sensor board to temperature.

- **cbTIn()** - Reads a channel from a digital input board, filters it (if specified), determines the cold junction compensation, linearizes and converts it to temperature.
- **cbTInScan()** - Scans a range of temperature inputs. Reads input temperatures from a range of channels, and returns the temperature values in an array.

Windows memory management functions

The Windows memory management functions are only available and needed in the Windows version of the library. These functions take care of allocating, freeing and copying to/from Windows global memory buffers. These functions are not used in VEE since VEE handles memory allocation. For customers wishing to customize memory management under VEE, the source code to CBV.DLL and CBV32.DLL is available. Please call technical support and request it.

- **cbWinBufAlloc()** - Allocates a Windows memory buffer.
- **cbWinBufFree()** - Frees a Windows buffer.
- **cbWinArrayToBuf()** - Copies data from an array to a Windows buffer.

- **cbWinBufToArray()** - Copies data from a Windows buffer to an array.

Miscellaneous functions

These functions do not as a group fit into a single category. They get and set board information, convert units, manage events and background operations, and perform serial communication operations.

- **cbDisableEvent()** - Disables one or more events set up with cbEnableEvent() and disconnects their user-defined handlers.
- **cbEnableEvent()** - Binds one or more event conditions to a user-defined callback function.
- **User Callback Function** – Defines the prototype for the user function for cbEnableEvent(). This defines the format for the user-defined handlers to be called when the events set up using cbEnableEvent() occurs.
- **cbFlashLED()** - Causes the LED on a USB to flash.
- **cbFromEngUnits()** - Converts a voltage (or current) to a D/A count value.
- **cbGetBoardName()** - Returns the name of a specified board.
- **cbGetStatus()** - Returns the status of a background operation. Once a background operation starts, your program needs to periodically check on its progress. This function returns the current status of the process.
- **cbInByte()** - Reads a byte from a hardware register on a board.
- **cbInWord()** - Reads a word from a hardware register on a board.
- **cbOutByte()** - Writes a byte to a hardware register on a board.
- **cbOutWord()** - Writes a word to a hardware register on a board.
- **cbRS485()** - Sets the transmit and receive buffers on an RS485 port.
- **cbStopBackground()** - Stop a background process. It is sometimes necessary to stop a background process even though the process has been set up to run continuously. This function stops a background process that is running. `cbStopBackground()` should be executed after normal termination of all background functions in order to clear variables and flags.
- **cbToEngUnits()** - Converts a count value from an A/D to voltage (or current).

Universal Library example programs

Universal Library contains many example programs to aid the user in learning and applying UL functions. We strongly recommend running appropriate example programs before attempting to use the functions.

Table 1-1 lists Universal Library example programs sorted by the program name. It includes their featured function calls, special aspects, and other function calls included in the program. All example programs include `cbDeclareRevision()` and `cbErrHandling()` functions. Table 1-2 lists the Universal Library example programs sorted by the function name.

CWIN sample programs
The CWIN sample program directory contains programs A101, A102 and A103 only.

Table 1-1. UL Example Programs – Sorted By Program Name

Program Name	Featured UL Function Call	Notes	Other UL Function Calls
ULAI01	`cbAIn`		`cbToEngUnits()`

ULAI02	`cbAInScan`	`FOREGROUND` mode	`cbWinBufToArray()`
			`cbWinBufFree()`
			`cbWinBufAlloc()`

ULAI03	`cbAInScan`	`BACKGROUND` mode	`cbGetStatus()`
			`cbStopBackground()`
			`cbWinBufToArray()`
			`cbWinBufFree()`
			`cbWinBufAlloc()`

ULAI04	`cbAConvertData`		`cbAInScan()`
			`cbGetStatus()`
			`cbStopBackground()`
			`cbWinBufToArray()`
			`cbWinBufFree()`
			`cbWinBufAlloc()`

ULAI05	`cbAInScan`	with manual data conversion	`cbGetStatus()`
			`cbStopBackground()`
			`cbWinBufToArray()`
			`cbWinBufFree()`
			`cbWinBufAlloc()()`

ULAI06	`cbAInScan`	`CONTINUOUS`
		`BACKGROUND` mode
			`cbAConvertData`
			`cbGetStatus()`
			`cbStopBackground()`
			`cbWinBufToArray()`
			`cbWinBufFree()`
			`cbWinBufAlloc()`

ULAI07	`cbATrig`		`cbFromEngUnits()`

ULAI08	`cbAPretrig`		`cbWinBufToArray()`
			`cbWinBufFree()`
			`cbWinBufAlloc()`

ULAI09	`cbAConvertPretrigData`	`BACKGROUND`	`cbAPretrig()`
			`cbGetStatus()`
			`cbStopBackground()`
			`cbWinBufToArray()`
			`cbWinBufFree()`
			`cbWinBufAlloc()`

ULAI10	`cbALoadQueue`		`cbAInScan()`
			`cbWinBufToArray()`
			`cbWinBufFree()`
			`cbWinBufAlloc()`

ULAI11	`cbToEngUnits`		`cbAIn()`

ULAI12	`cbAInScan`	`EXTCLOCK` mode	`cbWinBufToArray()`
			`cbWinBufFree()`
			`cbWinBufAlloc()`

ULAI13	`cbAInScan`	Various sampling mode options	`cbWinBufToArray()`
			`cbWinBufFree()`
			`cbWinBufAlloc()`

Program Name	Featured UL Function Call	Notes	Other UL Function Calls
ULAI14	cbSetTrigger	with EXTTRIGGER selected	cbAInScan() cbFromEngUnits() cbWinBufToArray() cbWinBufFree() cbWinBufAlloc()
ULAIO01	cbAInScan cbAOutScan	Concurrent analog input and analog output scans	cbGetStatus () cbStopBackground() cbWinArraytoBuf() cbWinBufToArray() cbWinBufFree() cbWinBuftoAlloc()
ULAO01	cbAOut		cbFromEngUnits()
ULAO02	cbAOutScan		cbWinBufToArray() cbWinBufFree() cbWinBufAlloc()
ULAO03	cbAOut cbSetConfig	Demonstrates the difference between BIDACUPDATEMODE settings of UPDATEIMMEDIATE and UPDATEONCOMMAND. Board 0 must support BIDACUPDATEMODE settings, such as the PCI-DAC6700 Series boards.	cbFromEngUnits()
ULCT01	cbC8254Config		cbCLoad() cbCIn()
ULCT02	cbC9513Init cbC9513Config		cbCLoad() cbCIn()()
ULCT03	cbCStoreOnInt		cbC9513Init cbC9513Config() cbCLoad() cbCIn()
ULCT04	cbCFreqIn		cbC9513Init()
ULCT05	cbC8536Init cbC8536Config		cbCLoad() cbCIn()
ULCT06	cbC7266Config		cbCLoad32 () cbCIn32() cbCStatus()
ULDI01	cbDIn		cbDConfigPort()
ULDI02	cbDBitIn		cbDConfigPort()
ULDI03	cbDInScan		cbDConfigPort() cbGetStatus() cbStopBackground() cbWinBufToArray() cbWinBufFree() cbWinBufAlloc()
ULDI04	cbDIn	using the AUXPORT	
ULDI05	cbDBitIn	using the AUXPORT	
ULDI06	cbDConfigBit		cbDBitIn()
ULDO01	cbDOut		cbDConfigPort()
ULDO02	cbDBitOut		cbDOut() cbDConfigPort()
ULDO04	cbDOut	using the AUXPORT	
ULDO05	cbDBitOut	using the AUXPORT	cbDOut()
ULEV01*	cbEnableEvent	using ONEXTERNALINTERRUPT	cbDisableEvent() cbDConfigPort() cbDIn()

Program Name	Featured UL Function Call	Notes	Other UL Function Calls
ULEV02*	cbEnableEvent	using ON_SCAN_ERROR, ON_DATA_AVAILABLE and ON_END_OF_AI_SCAN	cbAInScan() cbStopBackground() cbToEngUnits() cbWinBufAlloc() cbWinBufFree() cbWinBufToArray()
ULEV03*	cbEnableEvent	using ON_SCAN_ERROR,ON_PRETRIGGER, and ON_END_OF_AI_SCAN	cbAPretrig() cbAConvertPretrigData cbDConfigPort() cbDOut() cbStopBackground() cbToEngUnits() cbWinBufAlloc() cbWinBufFree() cbWinBufToArray()
ULEV04*	cbEnableEvent()	using ON_END_OF_AO_SCAN	cbAOutScan() cbDConfigPort() cbDOut() cbFromEngUnits() cbStopBackground() cbWinBufAlloc() cbWinBufFree() cbWinBufToArray()
ULFI01	cbFileAInScan()		cbFileGetInfo()
ULFI02	cbFileRead()		cbFileAInScan() cbFileGetInfo()
ULFI03	cbFilePretrig()		cbFileGetInfo() cbFileRead()
ULGT01	cbGetErrMsg()		cbAIn()
ULGT03	cbGetConfig()		cbGetBoardName()
ULGT04	cbGetBoardName()		cbGetConfig()
ULMBDI01	cbDIn()	Reads a digital input port on a MetraBus card	
ULMBDI02	cbDBitIn()	Reads the status of a single digital input bit from a MetraBus card	
ULMBDO01	cbDOut()	Writes a byte to a digital output port on a MetraBus card	
ULMBDO02	cbDBitOut()	Sets the state of a single digital output bit for a MetraBus card	
ULMM01	cbMemReadPretrig()		cbAPretrig()
ULMM02	cbMemRead() cbMemWrite()		
ULMM03	cbAInScan()	With the EXTMEMORY option	cbMemReset() cbMemRead()
ULTI01	cbTIn()		cbGetConfig()
ULTI02	cbTInScan()		cbGetConfig()
*Sample programs ULEV01, ULEV02, ULEV03 and ULEV04 are not available for the C Console.			

Table 1-2. UL Example Programs – Sorted By Function

UL Function Call	**UL Example Program Name**	**Special Features / Notes**
`cbAConvertData()`	ULAI04 ULA106	
`cbAConvertPretrigData()`	ULAI09 ULEV03*	
`cbACalibrateData()`	None	No example programs at this time
`cbAIn()`	ULAI01 ULGT01 ULAI11	
`cbAInScan()`	ULAI02 ULAI10 ULAI03 ULAI12 ULAI04 ULAI13 ULAI05 ULAI14 ULAI06 ULMM03 ULEV02*	`FOREGROUND`, `BACKGROUND` mode with manual data conversion `CONTINUOUS BACKGROUND` mode `EXTCLOCK` mode Various sampling mode options
`cbALoadQueue()`	ULAI10	
`cbAOut()`	ULAO01 ULAO03	ULAO03 demonstrates the difference between `BIDACUPDATEMODE settings of UPDATEIMMEDIATE` and `UPDATEONCOMMAND`. Board 0 must support `BIDACUPDATEMODE` settings, such as the PCI-DAC6700 Series.
`cbAOutScan()`	ULAO02 ULAIO01 ULEV04*	
`cbAPretrig()`	ULAI08 ULEV03* ULAI09 ULMM01 ULFI03	
`cbATrig()`	ULAI07 ULMM01	
`cbC7266Config()`	ULCT06	
`cbC8254Config()`	ULCT01	
`cbC8536Config()`	ULCT05	
`cbC8536Init()`	ULCT05	
`cbC9513Config()`	ULCT02 ULCT03	
`cbC9513Init()`	ULCT02 ULCT04 ULCT03	
`cbCFreqIn()`	ULCT04	
`cbCIn()`	ULCT01 ULCT05 ULCT02	
`cbCIn32()`	ULCT06	
`cbCLoad()`	ULCT01 ULCT03 ULCT02 ULCT05	
`cbCLoad32()`	ULCT06	
`cbCStoreOnInt()`	ULCT03	
`cbCStatus()`	ULCT06	
`cbDBitIn()`	ULDI02 ULDI06 ULDI05 ULMBDI02	
`cbDBitOut()`	ULDO02 ULDO05 ULMBDO02	

UL Function Call	UL Example Program Name	Special Features / Notes
`cbDConfigBit()`	ULDI06	
`cbDConfigPort()`	ULDI01 ULDO01 ULDI02 ULDO02 ULDI03 ULDO05 ULEV01* ULEV04* ULEV03*	
`cbDIn()`	ULDI01 ULDI04 ULDI03 ULMBDI01 ULEV04*	
`cbDInScan()`	ULDI03	
`cbDOut()`	ULDO01 ULDO05 ULDO02 ULMBDO01 ULDO04 ULMBDO02 ULEV03* ULEV04*	
`cbDOutScan()`	None	No example programs at this time

`cbEnableEvent()`
`cbDisableEvent()`
	ULEV01* ULEV03*
	ULEV02* ULEV04*
		`ON_EXTERNAL_INTERRUPT`
		`ON_DATA_AVAILABLE`
		`ON_PRETRIGGER`
		`ON_END_OF_AO_SCAN`
		`ON_SCAN_ERROR`
		`ON_END_OF_AI_SCAN`

`cbMemRead()`	ULMM01 ULMM03 ULMM02	
`cbMemReadPretrig()`	ULMM01	
`cbMemReset()`	ULMM03	
`cbMemSetDTMode()`	None	No example programs at this time
`cbMemWrite()`	ULMM02	
`cbRS485()`	None	No example programs at this time
`cbGetBoardName()`	ULGT03 ULGT04	
`cbErrHandling()`	All Samples	All example programs use this function
`cbGetErrMsg()`	ULGT01	
`cbGetStatus()`	ULAI03 ULAI06 ULAI04 ULAI09 ULAI05 ULCT03 ULAIO01 ULDI03	
`cbInByte()`	None	No example programs at this time
`cbInWord()`	None	No example programs at this time
`cbOutByte()`	None	No example programs at this time
`cbOutWord()`	None	No example programs at this time
`cbGetConfig()`	ULGT03 ULTI01 ULGT04 ULTI02	
`cbSetConfig()`	ULAO03	Demonstrates the difference between `BIDACUPDATEMODE` `settings of UPDATEIMMEDIATE` and `UPDATEONCOMMAND`. Board 0 must support `BIDACUPDATEMODE` settings, such as the PCI-DAC6700 Series boards.
`cbSetTrigger()`	ULAI14	

UL Function Call	**UL Example Program Name**	**Special Features / Notes**
`cbStopBackground()`	ULAI03 ULAI06 ULAI04 ULAI09 ULAI05 ULCT03 ULAIO01 ULDI03 ULEV02* ULEV03* ULEV04*	Concurrent `cbAInScan()` and `cbAOutScan()`
`cbToEngUnits()`	ULAI01 ULAI11 ULAI07 ULEV02* ULEV03*	
`cbFromEngUnits()`	ULAI01 ULAO03 ULAI07 ULEV04* ULAI14	
`cbDeclareRevision()`	All Samples	All example programs use this function
`cbGetRevision()`	None	No example programs at this time
`cbFileAInScan()`	ULFI01 ULFI02	
`cbFilePretrig()`	ULFI03	
`cbFileRead()`	ULFI02 ULFI03	
`cbTIn()`	ULTI01	
`cbTInScan()`	ULTI02	

`cbWinBufAlloc()`
`cbWinBufFree()`
`cbWinBufToArray()`
	ULAI01 ULAI10
	ULAI02 ULAI12
	ULAI03 ULAI13
	ULAI04 ULAI14
	ULAI05
	ULAI06 ULAO02
	ULAI08 ULCT03
	ULAI09 ULDI03
	ULEV02* ULEV03*
	ULEV04*
	(`cbWinBufAlloc()` and `cbWinBufFree()` only)	

`cbWinArrayToBuf()`	ULAI01 ULAO02 ULEV04*	
*Sample programs ULEV01, ULEV02, ULEV03 and ULEV04 are not available for the C Console.		

Analog I/O Functions

Introduction

The functions explained in this chapter handle analog input, analog output and analog data manipulation. To determine which of these functions are compatible with your hardware, refer to the *Universal Library User's Guide* (available in PDF format on our website at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Most of the functions in this section provide options that may not be compatible with your hardware. Again, you should refer to the *Universal Library User's Guide* to determine if the options you are considering using with a particular function are compatible with your hardware.

Table 2-1 below lists the constants you can use in the `Range` argument found in most of the functions explained in this chapter. These values are also used in the `cbALoadQueue()` function's `GainArray` argument. Valid ranges for your hardware are listed in the *Universal Library User's Guide*.

Table 2-1. `Range` constants

UL settings	Value	UL settings	Value
BIP20VOLTS	±20 volts (V)	UNI10VOLTS	0 to 10 V
BIP10VOLTS	±10 V	UNI5VOLTS	0 to 5 V
BIP5VOLTS	±5 V	UNI2PT5VOLTS	0 to 2.5 V
BIP4VOLTS	±4 V	UNI2VOLTS	0 to 2 V
BIP2PT5VOLTS	±2.5 V	UNI1PT25VOLTS	0 to 1.25 V
BIP2VOLTS	±2 V	UNI1PT67VOLTS	0 to 1.67 V
BIP1PT25VOLTS	±1.25 V	UNI1VOLTS	0 to 1 V
BIP1VOLTS	±1 V	UNIPT5VOLTS	0 to 0.5 V
BIP1PT67VOLTS	±1.67 V	UNIPT25VOLTS	0 to 0.25 V
BIPPT625VOLTS	±0.625 V	UNIPT2VOLTS	0 to 0.2 V
BIPPT5VOLTS	±0.5 V	UNIPT1VOLTS	0 to 0.1 V
BIPPT25VOLTS	±0.25 V	UNIPT01VOLTS	0 to 0.01 V
BIPPT2VOLTS	±0.2 V	UNIPT02VOLTS	0 to 0.02 V
BIPPT1VOLTS	±0.1 V	MA4TO20	4 to 20 milliamperes (mA)
BIPPT05VOLTS	±0.05 V	MA2TO10	2 to 10 mA
BIPPT01VOLTS	±0.01 V	MA1TO5	1 to 5 mA
BIPPT005VOLTS	±0.005 V	MAPT5TO2PT5	0.5 to 2.5 mA
		MA0TO20	0 to 20 mA

cbAConvertData()

Changed R3.3 RW

Converts the raw data collected by `cbAInScan()` into 12-bit A/D values. The `cbAInScan()` function can return either raw A/D data or converted data, depending on whether or not the `CONVERTDATA` option is used. For many 12-bit A/D boards, the raw data is a 16-bit value that contains a 12-bit A/D value and a 4 bit channel tag (refer to the board-specific information or the board's user manual). The converted data consists of just the 12-bit A/D value.

Function prototype:

C/C++:	`int cbAConvertData (int BoardNum, long NumPoints, unsigned short ADData[ ], unsigned short ChanTags[ ])`
Visual Basic:	`Function cbAConvertData( ByVal BoardNum&, ByVal NumPoints&, ADData%, ChanTags%) As Long`
Delphi:	`function cbAConvertData (BoardNum:Integer; NumPoints:Longint; var ADData:Word; var ChanTags:Word):Integer;`

Arguments:

`BoardNum`	The board number used to collect the data. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library). Refers to the number associated with the board used to collect the data when it was installed with the *Insta*Cal® configuration program.
`NumPoints`	Number of samples to convert
`ADData`	Pointer or reference to start of data array
`ChanTags`	Pointer or reference to start of channel tag array

Returns:

Error code or 0 if no errors.

`ADData` - converted data.

`ChanTags` - channel tags if available.

When collecting data using `cbAInScan()` without the `CONVERTDATA` option, use this function to convert the data after it has been collected. There are cases where the `CONVERTDATA` option is not allowed. For example - if you are using both the `DMAIO` and `BACKGROUND` option with `cbAInScan()`. In those cases this function should be used to convert the data after the data collection is complete.

For some boards, each raw data point consists of a 12-bit A/D value with a 4-bit channel number. This function pulls each data point apart and puts the A/D value into the `ADData` array and the channel number into the `ChanTags` array.

Notes:

12-bit A/D boards

- Name of the array must match that used in `cbAInScan()` or `cbWinBufToArray()`.
- Upon returning from `cbAConvertData()`, `ADData` array contains only 12-bit A/D data.

16-bit A/D boards

This function is not for use with 16-bit A/D boards because 16-bit boards do not have channel tags. The argument `BoardNum` was added in revision 3.3 to prevent applying this function to 16-bit data. If you wrote your program for a 12-bit board then later upgrade to a 16-bit board all you need change is the *Insta*Cal configuration file. If this function is called for a 16-bit board, it is simply ignored. No errors are generated.

cbAConvertPretrigData()

Changed R3.3 RW

Converts the raw data collected by `cbAPretrig()`. The `cbAPretrig()` function can return either raw A/D data or converted data, depending on whether or not the `CONVERTDATA` option was used. The raw data as it is collected is not in the correct order. After the data collection is completed it must be rearranged into the correct order. This function correctly orders the data also, starting with the first pretrigger data point and ending with the last post-trigger point.

Change at revision 3.3 is to support multiple background tasks. It is now possible to run two boards with DMA or REP-INSW background convert-and-transfer features active, therefore, the convert function must know which board the data came from. The data value assigned to `BoardNum` should be assigned in the header file so it will be easy to locate if a change is needed.

Function prototype:

C/C++:

```
int cbAConvertPretrigData( int BoardNum, long PretrigCount, long TotalCount, unsigned short ADData[], unsigned short ChanTags[] )
```

Visual Basic:

```
Function cbAConvertPretrigData( ByVal BoardNum&, ByVal PretrigCount&, ByVal TotalCount&, ADData%, ChanTags% ) As Long
```

Delphi:

```
function cbAConvertPretrigData (BoardNum:Integer; PretrigCount:Longint; TotalCount:Longint; var ADData:Word; var ChanTags:Word):Integer;
```

Arguments:

`BoardNum`	The board number used to collect the data. `BoardNum` may be 0 to 99 (0 to 9 for the 16-bit version of Universal Library). Refers to the number associated with the board used to collect the data when it was installed with the *InstaCal®* configuration program.
`PretrigCount`	Number of pre-trigger samples (this value must match the value returned by the `PretrigCount` argument in the `cbAPretrig()` function)
`TotalCount`	Total number of samples that were collected
`ADData`	Pointer to data array (must match array name used in `cbAPretrig()` function)
`ChanTags`	Pointer to channel tag array or a NULL pointer may be passed if using 16-bit boards or if channel tags are not desired (see the note regarding 16-bit boards on page 17).

Returns:

Error code or 0 if no errors.

`ADData` - converted data.

When you collect data with `cbAPretrig()` and you don't use the `CONVERTDATA` option, you must use this function to convert the data after it is collected. There are cases where the `CONVERTDATA` option is not allowed: for example, if you use the `BACKGROUND` option with `cbAPretrig()`. In those cases this function should be used to convert the data after the data collection is complete.

Notes:

12-bit A/D boards:

- On some 12-bit boards, each raw data point consists of a 12-bit A/D value with a 4-bit channel number. This function pulls each data point apart and puts the A/D value into the `ADData` and the channel number into the `ChanTags` array.

- Name of the ADData array must match that used in `cbAInScan()` or `cbWinBufToArray()`.
- Upon returning from `cbAConvertPretrigData()`, `ADData` array contains only 12-bit A/D data.

16-bit A/D boards:

This function is for use with 16-bit A/D boards only insofar as ordering the data. No channel tags are returned.

Visual Basic programmers:

After the data is collected with `cbAPretrig()` it must be copied to an array with `cbWinBufToArray().`

IMPORTANT

The entire array must be copied. This array includes the extra 512 samples needed by `cbAPretrig()`. Example code is given below.

```
Count& = 10000

Dim ADData% (Count& + 512)
Dim ChanTags% (Count& + 512)

cbAPretrig%(BoardNum, LowChan, HighChan, PretrigCount&, Count&...)
cbWinBufToArray%(MemHandle%, ADData%, Count& + 512)
cbAConvertPretrigData%(PretrigCount&, Count&, ADData%, ChanTags%)
```

cbACalibrateData()

New R3.3

Calibrates the raw data collected by `cbAInScan()` from boards with real time software calibration when the real time calibration has been turned off. The `cbAInScan()` function can return either raw A/D data or calibrated data, depending on whether or not the `NOCALIBRATEDATA` option was used.

Function prototype:

C/C++:
```
int cbACalibrateData (int BoardNum, long NumPoints, int Range, unsigned ADData[ ])
```

Visual Basic:
```
Function cbACalibrateData( ByVal BoardNum&, ByVal NumPoints&, ByVal Range&, ADData% ) As Long
```

Delphi:
```
function cbACalibrateData ( BoardNum:Integer; var NumPoints:Longint; Range:Integer; var ADData:Word ):Integer;
```

Arguments:

`BoardNum`	May be 0 to 99 (0 to 9 for 16-bit version of Universal Library). Number associated with the board when it was installed using *InstaCal®*.
`NumPoints`	Number of samples to convert
`Range`	The programmable gain/range used when the data was collected. See Table 2- on page 13 for valid values.
`ADData`	Pointer to data array.

Returns:

Error code or 0 if no errors.

`ADData` - converted data.

Notes:

When collecting data using `cbAInScan()` with the `NOCALIBRATEDATA` option, use this function to calibrate the data once collected.

- The name of the array must match that used in `cbAInScan()` or cbWinBufToArray().
- Applying software calibration factors in real time on a per sample basis eats up machine cycles. If your CPU is slow, or if processing time is at a premium, do not calibrate until the acquisition run finishes. Turn off real time software calibration to save CPU time during high speed acquisitions by using the `NOCALIBRATEDATA` option to a turn off real-time software calibration. After the acquisition is run, calibrate the data with `cbACalibrateData()`.

cbAIn()

Reads an A/D input channel. This function reads the specified A/D channel from the specified board. If the specified A/D board has programmable gain then it sets the gain to the specified range. The raw A/D value is converted to an A/D value and returned to `DataValue`.

Function prototype:

C/C++:
```
int cbAIn( int BoardNum, int Channel, int Range, unsigned short *DataValue );
```

Visual Basic:
```
Function cbAIn( ByVal BoardNum&, ByVal Channel&, ByVal Range&, DataValue% ) As Long
```

Delphi:
```
function cbAIn ( BoardNum:Integer; Channel:Integer; Range:Integer; var DataValue:Word ):Integer;
```

Arguments:

`BoardNum` — The board number used to collect the data. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library). Refers to the number associated with the board used to collect the data when it was installed with the *InstaCal*® configuration program. The specified board must have an A/D.

`Channel` — A/D channel number. The maximum allowable channel depends on which type of A/D board is being used. For boards with both single ended and differential inputs, the maximum allowable channel number also depends on how the board is configured. For example, a CIO-DAS1600 has 8 channels for differential, 16 for single ended. Expansion boards are also supported by this function, so this argument can contain values up to 272. See board specific information for EXP boards if you are using an expansion board.

`Range` — A/D range code. If the selected A/D board does not have a programmable gain feature, this argument is ignored. If the A/D board does have programmable gain, set the `Range` argument to the desired A/D range. See Table 2- on page 13 for valid values.

`DataValue` — Pointer or reference to the data value.

Returns:

Error code or 0 if no errors.

`DataValue` - Returns the value of the A/D sample.

cbAInScan()

Changed R3.3 ID

Scans a range of A/D channels and stores the samples in an array. `cbAInScan()` reads the specified number of A/D samples at the specified sampling rate from the specified range of A/D channels from the specified board. If the A/D board has programmable gain, then it sets the gain to the specified range. The collected data is returned to the data array.

Changes: Revision 3.3 added a *'no real time calibration'* option.

Function prototype:

C/C++:

```
int cbAInScan (int BoardNum, int LowChan, int HighChan, long Count,
long *Rate, int Range, int MemHandle, int Options)
```

Visual Basic:

```
Function cbAInScan(ByVal BoardNum&, ByVal LowChan&, ByVal HighChan&,
ByVal Count&, Rate&, ByVal Range&, ByVal MemHandle&, ByVal Options&)
As Long
```

Delphi:

```
function cbAInScan (BoardNum:Integer; LowChan:Integer;
HighChan:Integer; Count:Longint; var Rate:Longint; Range:Integer;
MemHandle:Integer; Options:Integer) : Integer;
```

Arguments:

`BoardNum`
The board number used to collect the data. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library). Refers to the number associated with the board used to collect the data when it was installed with the *InstaCal®* configuration program. The specified board must have an A/D.

`LowChan`
The first A/D channel of scan. When `cbALoadQueue()` is used, the channel count is determined by the total number of entries in the channel gain queue. `LowChan` is ignored.

`HighChan`
The last A/D channel of scan. When `cbALoadQueue()` is used, the channel count is determined by the total number of entries in the channel gain queue. `HighChan` is ignored.

Low / High Channel # - The maximum allowable channel depends on which type of A/D board is being used. For boards that have both single ended and differential inputs the maximum allowable channel number also depends on how the board is configured. For example, a CIO-DAS1600 has 8 channels for differential, 16 for single ended.

`Count`
Number of A/D samples to collect. Specifies the total number of A/D samples that will be collected. If more than one channel is being sampled then the number of samples collected per channel is equal to
`Count` / (`HighChan - LowChan` + 1).

`Rate`
The sample rate at which acquisitions are triggered, in samples per second per channel.

For example, if you sample four channels, 0-3, at a rate of 10,000 scans per second (10 kHz), the resulting A/D converter rate is 40 kHz: four channels at 10,000 samples per channel per second. This is different from some software where you specify the total A/D chip rate. In those systems, the per channel rate is equal to the A/D rate divided by the number of channels in a scan.

The channel count is determined by the LowChan and HighChan parameters. Channel Count = (`HighChan` - `LowChan` + 1).

When `cbALoadQueue` is used, the channel count is determined by the total number of entries in the channel gain queue. `LowChan` and `HighChan` are ignored.

	`Rate` also returns the value of the actual rate set, which may be different from the requested rate because of pacer limitations.
`Range`	A/D range code. If the selected A/D board does not have a programmable range feature, this argument is ignored. Otherwise, set the `Range` argument to any range that is supported by the selected A/D board. See Table 2- on page 13 for valid values.
`MemHandle`	Handle for Windows buffer to store data in (Windows). This buffer must have been previously allocated with the `cbWinBufAlloc()` function. In HP VEE this panel is called Data Array. Refer to HP VEE specific information for more details.
`Options`	Bit fields that control various options. This field may contain any combination of non-contradictory choices from the values listed in the "Options argument values" section below.

Returns:

Error code or 0 if no errors.

`Rate` - actual sampling rate used.

`MemHandle` - collected A/D data returned via the Windows buffer.

Options argument values:

Transfer method options: The following four options determine how data is transferred from the board to PC memory. If none of these four options are specified (recommended), the optimum sampling mode is automatically chosen based on board type and sampling speed.

SINGLEIO	A/D transfers to memory are initiated by an interrupt. One interrupt per conversion.
DMAIO	A/D transfers are initiated by a DMA request.
BLOCKIO	A/D transfers are handled in blocks (by REP-INSW for example). **BLOCKIO is not recommended for slow acquisition rates:** If the rate of acquisition is very slow (say less than 200 Hz) `BLOCKIO` is probably not the best choice for transfer mode. The reason for this is that status for the operation is not available until one packet of data has been collected (typically 512 samples). The implication is that if acquiring 100 samples at 100 Hz using `BLOCKIO`, the operation will not complete until 5.12 seconds has elapsed.
BURSTIO	Allows higher sampling rates for sample counts up to full FIFO. Data is collected into the local FIFO. Data transfers to the PC are held off until after the scan is complete. For `BACKGROUND` scans, the count and index returned by `cbGetStatus()` remain 0 and the status equals `RUNNING` until the scan finishes. When the scan is complete and the data is retrieved, the count and index are updated and the status equals IDLE. `BURSTIO` is the default mode for non-`CONTINUOUS` fast scans (aggregate sample rates above 1000 Hz) with sample counts up to full FIFO. To avoid the `BURSTIO` default, specify `BLOCKIO`. Non-`BURSTIO` scans are limited to a maximum of 1200 Hz.

BURSTMODE	Enables burst mode sampling. Scans from `LowChan` to `HighChan` are clocked at the maximum A/D rate in order to minimize channel to channel skew. Scans are initiated at the rate specified by Rate. `BURSTMODE` is not recommended for use with the `SINGLEIO` option. If this combination is used, the `Count` value should be set as low as possible, preferably to the number of channels in the scan. Otherwise, overruns may occur.

CONVERTDATA	If the CONVERTDATA option is used for 12-bit boards then the data that is returned to the buffer will automatically be converted to 12-bit A/D values. If CONVERTDATA is not used then the data from 12-bit A/D boards will be return unmodified (which, for some boards is 16-bit values that contain both a 12-bit A/D value and a 4 bit channel number). After the data collection is complete you can call cbAConvertData() to convert the data after the fact. CONVERTDATA may not be specified if you are using the BACKGROUND option and DMA transfers. This option is ignored for the 16-bit boards.
BACKGROUND	If the BACKGROUND option is not used then the cbAInScan() function will not return to your program until all of the requested data has been collected and returned to the buffer. When the BACKGROUND option is used, control will return immediately to the next line in your program and the data collection from the A/D into the buffer will continue in the background. Use cbGetStatus() to check on the status of the background operation. Alternatively, some boards support cbEnableEvent() for event notification of changes in status of BACKGROUND scans. Use cbStopBackground() to terminate the background process before it has completed. cbStopBackground() should be executed after normal termination of all background functions in order to clear variables and flags.
CONTINUOUS	This option puts the function in an endless loop. Once it collects the required number of samples, it resets to the start of the buffer and begins again. The only way to stop this operation is with cbStopBackground(). Normally this option should be used in combination with BACKGROUND so that your program will regain control. **Count argument settings in CONTINUOUS mode:** For some DAQ hardware, Count must be an integer multiple of the *packet size.* Packet size is the amount of data that a DAQ device transmits back to the PC's memory buffer during each data transfer. Packet size can differ among DAQ hardware, and can even differ on the same DAQ product depending on the transfer method. In some cases, the minimum value for the Count argument may change when the CONTINUOUS option is used. This can occur for several reasons; the most common is that in order to trigger an interrupt on boards with FIFOs, the circular buffer must occupy at least half the FIFO. Typical half-FIFO sizes are 256, 512 and 1024. Another reason for a minimum Count value is that the buffer in memory must be periodically transferred to the user buffer. If the buffer is too small, data will be overwritten during the transfer resulting in garbled data. Refer board-specific section of the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) for packet size information for your particular DAQ hardware.
EXTCLOCK	If this option is used then conversions will be controlled by the signal on the external clock input rather than by the internal pacer clock. Each conversion will be triggered on the appropriate edge of the clock input signal (see board specific info). When this option is used the Rate argument is ignored. The sampling rate is dependent on the clock signal. Options for the board will default to a transfer mode that will allow the maximum conversion rate to be attained unless otherwise specified.

SINGLEIO is recommended for slow external clock rates: If the rate of the external clock is very slow (say less than 200 Hz) and the board you are using supports `BLOCKIO`, you may want to include the `SINGLEIO` option. The reason for this is that the status for the operation is not available until one packet of data has been collected (typically 512 samples). The implication is that, if acquiring 100 samples at 100 Hz using `BLOCKIO` (the default for boards that support it if `EXTCLOCK` is used), the operation will not complete until 5.12 seconds has elapsed.

`EXTMEMORY`	Causes the command to send the data to a connected memory board via the DT-Connect interface rather than returning the data to the buffer. Data for each call to this function will be appended unless `cbMemReset()` is called. The data should be unloaded with the `cbMemRead()` function before collecting new data. When `EXTMEMORY` option is used, the `MemHandle` argument can be set to null or 0. `CONTINUOUS` option cannot be used with `EXTMEMORY`. Do not use `EXTMEMORY and DTCONNECT` together. The transfer modes `DMAIO`, `SINGLEIO`, `BLOCKIO and BURSTIO` have no meaning when used with this option.
`EXTTRIGGER`	If this option is specified the sampling will not begin until the trigger condition is met. On many boards, this trigger condition is programmable (see `cbSetTrigger()` on page 44 and board-specific information for details) and can be programmed for rising or falling edge or an analog level. On other boards, only 'polled gate' triggering is supported. In this case, assuming active high operation, data acquisition will commence immediately if the trigger input is high. If the trigger input is low, acquisition will be held off unit it goes high. Acquisition will then continue until `NumPoints`& samples have been taken regardless of the state of the trigger input. For 'polled gate' triggering, this option is most useful if the signal is a pulse with a very low duty cycle (trigger signal in TTL low state most of the time) so that triggering will be held off until the occurrence of the pulse.
`NOTODINTS`	If this option is specified, the system's time-of-day interrupts are disabled for the duration of the scan. These interrupts are used to update the systems real time clock and are also used by various other programs. These interrupts can limit the maximum sampling speed of some boards - particularly the PCM-DAS08. If the interrupts are turned off using this option then the real time clock will fall behind by the length of time that the scan takes.
`NOCALIBRATEDATA`	Turns off real-time software calibration for boards which are software calibrated, by applying calibration factors to the data on a sample by sample basis as it is acquired. Examples are the PCM-DAS16/330 and PCM-DAS16x/12. Turning off software calibration saves CPU time during a high speed acquisition run. This may be required if your processor is less than a 150 MHz Pentium and you desire an acquisition speed in excess of 200 kHz. These numbers may not apply to your system. Only trial will tell for sure. DO NOT use this option if you do not have to. If this option is used, the data must be calibrated after the acquisition run with the `cbACalibrateData()` function.
`DTCONNECT`	All A/D values will be sent to the A/D board's DT-Connect port. This option is incorporated into the `EXTMEMORY` option. Use `DTCONNECT` only if the external board is not supported by Universal Library.

Notes:

In HP VEE, this panel is called Data Array. See `cbvGetAInData()` information in the *Universal Library Help for HP VEE* for more details.

Caution! You will generate an error if you specify a total A/D rate beyond the capability of the board. For example; if you specify rate `LowChan` = 0, `HighChan` = 7 (8 channels total) and `Rate` = 20,000 and you are using a CIO-DAS16/JR, you will get an error. You have specified a total rate of 8*20,000 = 160,000. The CIO-DAS16/JR is capable of converting 120,000 samples per second. The maximum sampling rate depends on the A/D board that is being used. It is also dependent on the sampling mode options.

Important

In order to understand the functions, you must read the board-specific information found in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf). The example programs should be examined and run prior to attempting any programming of your own. Following this advice will save you hours of frustration, and possibly time wasted holding for technical support.

This note, which appears elsewhere, is especially applicable to this function. Now is the time to read the board specific information for your board (see the *Universal Library User's Guide*). We suggest that you make a copy of that page to refer to as you read this manual and examine the example programs.

cbALoadQueue()

Loads the A/D board's channel/gain queue. This function only works with A/D boards that have channel/gain queue hardware.

Some products do not support channel / gain queue, and some that do support it are limited on the order of elements, number of elements, and gain values that can be included, etc. Please refer to the device-specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) to find details for your particular product.

Function prototype:

C/C++:
```
int cbALoadQueue( int BoardNum, short ChanArray[], short
GainArray[], int Count )
```

Visual Basic:
```
Function cbALoadQueue( ByVal BoardNum&, ChanArray%, GainArray%,
ByVal Count& ) As Long
```

Delphi:
```
function cbALoadQueue (BoardNum:Integer; var ChanArray:SmallInt; var
GainArray:SmallInt; Count:LongInt):Integer;
```

Arguments:

`BoardNum`	The board number used to collect the data. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library). Refers to the number associated with the board used to collect the data when it was installed with the *InstaCal*® configuration program. The specified board must have an A/D and a channel/gain queue.
`ChanArray`	Array containing channel values. This array should contain all of the channels that will be loaded into the channel gain queue.
`GainArray`	Array containing A/D range values. This array should contain each of the A/D ranges that will be loaded into the channel gain queue.
`Count`	Number of elements in `ChanArray` and `GainArray` or 0 to disable channel/gain queue. Specifies the total number of channel/gain pairs that will be loaded into the queue. `ChanArray and GainArray` should contain at least `Count` elements. Set `Count` = 0 to disable the board's channel/gain queue. The maximum value is specific to the queue size of the A/D boards channel gain queue.

Returns:

Error code or 0 if no errors.

Notes:

Normally the `cbAInScan()` function scans a fixed range of channels (from `LowChan` to `HighChan`) at a fixed A/D range. If you load the channel gain queue with this function then all subsequent calls to `cbAInScan()` will cycle through the channel/range pairs that you have loaded into the queue.

cbAOut()

Sets the value of a D/A output.

Function prototype:

C/C++:
```
int cbAOut( int BoardNum, int Channel, int Range, unsigned short
DataValue )
```

Visual Basic:
```
Function cbAOut( ByVal BoardNum&, ByVal Channel&, ByVal Range&,
ByVal DataValue% ) As Long
```

Delphi:
```
function cbAOut( BoardNum:Integer; Channel:Integer; Range:Integer;
DataValue:Word ):Integer;
```

Arguments:

`BoardNum` The board number used to collect the data. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library). Refers to the number associated with the board used to collect the data when it was installed with the *InstaCal*® configuration program. The specified board must have a D/A.

`Channel` D/A channel number. The maximum allowable channel depends on which type of D/A board is being used.

`Range` D/A range code. The output range of the D/A channel can be set to any of those supported by the board. If the D/A board does not have programmable ranges then this argument will be ignored. See Table 2- on page 13 for valid values.

`DataValue` Value to set D/A to. Must be in the range 0 - N where N is the value $2^{Resolution}$ - 1 of the converter

Exception: Using 16-bit boards with Basic range is -32768 to 32767. Refer to the discussion of Basic signed integers in the "16-bit values using a signed integer data type" section in the "Universal Library Description & Use" chapter of the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Returns:

Error code or 0 if no errors

Notes:

"Simultaneous Update" or "Zero Power-Up" boards: If you set the simultaneous update jumper for simultaneous operation, use `cbAOutScan()` for simultaneous update of multiple channels. `cbAOut()` always writes the D/A data then reads the D/A, which causes the D/A output to be updated.

cbAOutScan()

Outputs values to a range of D/A channels. This function can be used for paced analog output on hardware that supports paced output. It can also be used to update all analog outputs at the same time when the `SIMULTANEOUS` option is used.

Function prototype:

C/C++:
```
int cbAOutScan (int BoardNum, int LowChan, int HighChan, long Count,
long *Rate, int Range, int MemHandle, int Options)
```

Visual Basic:
```
Function cbAOutScan(ByVal BoardNum&, ByVal LowChan&, ByVal
HighChan&, ByVal Count&, Rate&, ByVal Range&, ByVal MemHandle&,
ByVal Options&) As Long
```

Delphi:
```
function cbAOutScan (BoardNum:Integer; LowChan:Integer;
HighChan:Integer; Count:Longint; var Rate:Longint; Range:Integer;
MemHandle:Integer; Options:Integer):Integer;
```

Arguments:

`BoardNum` — Refers to the board number associated with the board when it was installed with the configuration program. The specified board must have a D/A. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`LowChan` — First D/A channel of scan.

`HighChan` — Last D/A channel of scan.

`LowChan`/`HighChan` - The maximum allowable channel depends on which type of D/A board is being used.

`Count` — Number of D/A values to output. Specifies the total number of D/A values that will be output. Most D/A boards do not support timed outputs. For these boards, set the count to the number of channels in the scan.

`Rate` — Sample rate in scans per second. For many D/A boards the `Rate` is ignored and can be set to `NOTUSED`. For D/A boards with trigger and transfer methods which allow fast output rates, such as the CIO-DAC04/12-HS, `Rate` should be set to the D/A output rate (in **scans/sec**). This argument also returns the value of the actual rate set. This value may be different from the user specified rate because of pacer limitations.

If supported, this is the rate at which scans are triggered. If you are updating 4 channels, 0-3, then specifying a rate of 10,000 scans per second (10 kHz) will result in the D/A converter rates of 10 kHz: (one D/A per channel). The data transfer rate will be 40,000 words per second; 4 channels * 10,000 updates per scan.

The maximum update rate depends on the D/A board that is being used. It is also dependent on the sampling mode options.

`Range` — D/A range code. The output range of the D/A channel can be set to any of those supported by the board. If the D/A board does not have a programmable then this argument will be ignored. See Table 2- on page 13 for valid values.

`MemHandle` — Handle for Windows buffer from which data will be output. This buffer must have been previously allocated with the `cbWinBufAlloc()` function and data values loaded (perhaps using `cbWinArrayToBuf()`.

`Options` — Bit fields that control various options. This field may contain any combination of non-contradictory choices from the values listed in the "Options argument values" section on page 28.

Returns:

Error code or 0 if no errors.

`Rate` - actual sampling rate used.

Options argument values:

CONTINUOUS	This option may only be used with boards which support interrupt, DMA or REP-INSW transfer methods. This option puts the function in an endless loop. Once it outputs the specified (by `Count`) number of D/A values, it resets to the start of the buffer and begins again. The only way to stop this operation is with `cbStopBackground()`. This option should only be used in combination with BACKGROUND so that your program can regain control.
BACKGROUND	This option may only be used with boards which support interrupt, DMA or REP-INSW transfer methods. When this option is used the D/A operations will begin running in the background and control will immediately return to the next line of your program. Use `cbGetStatus()` to check the status of background operation. Alternatively, some boards support `EnableEvent()` for event notification of changes in status of BACKGROUND scans. Use `cbStopBackground()` to terminate background operations before they are completed. `cbStopBackground()` should be executed after normal termination of all background functions in order to clear variables and flags.
SIMULTANEOUS	When this option is used (if the board supports it and the appropriate switches are set on the board) all of the D/A voltages will be updated simultaneously when the last D/A in the scan is updated. This generally means that all the D/A values will be written to the board, then a read of a D/A address causes all D/As to be updated with new values simultaneously.
EXTCLOCK	If this option is used then conversions will be paced by the signal on the external clock input rather than by the internal pacer clock. Each conversion will be triggered on the appropriate edge of the clock input signal (see board specific info). When this option is used the `Rate` argument is ignored. The sampling rate is dependent on the clock signal. `Options` for the board will default to transfer types that allow the maximum conversion rate to be attained unless otherwise specified.
EXTTRIGGER	If this option is specified, the sampling will not begin until the trigger condition is met. On many boards, this trigger condition is programmable (see `cbSetTrigger()` on page 44 and board-specific information for details).

Notes:

In VEE this panel is called Data Array. Refer to the `cbvAOutSetData()` function in the *Universal Library Help for HP VEE.* for more information.

Caution! You will generate an error if you specify a total D/A rate beyond the capability of the board. For example: If you specify `LowChan` = 0 and `HighChan` = 3 (4 channels total) and `Rate` = 100,000, and you are using a cSBX-DDA04, you will get an error. You have specified a total rate of 4*100,000 = 400,000. The cSBX-DDA04 is rated to 330,000 updates per second. The maximum update rate depends on the D/A board that is being used. It is also dependent on the sampling mode options.

cbAPretrig()

Waits for a trigger to occur and then returns a specified number of analog samples before and after the trigger occurred. If only 'polled gate' triggering is supported, the trigger input line (refer to the user's manual for the board) must be at TTL low before this function is called, or a TRIGSTATE error will occur. The trigger occurs when the trigger condition is met. Refer to `cbSetTrigger()` on page 44 for details.

Function prototype:

C/C++:

```
int cbAPretrig (int BoardNum, int LowChan, int HighChan, long
*PretrigCount, long *TotalCount, long *Rate, int Range, int
MemHandle, int Options)
```

Visual Basic:

```
Function cbAPretrig( ByVal BoardNum&, ByVal LowChan&, ByVal
HighChan&, PretrigCount&, TotalCount&, Rate&, ByVal Range&, ByVal
MemHandle&, ByVal Options& ) As Long
```

Delphi:

```
function cbAPretrig (BoardNum:Integer; LowChan:Integer;
HighChan:Integer; var PretrigCount:Longint; var TotalCount:Longint;
var Rate:Longint; Range:Integer; MemHandle:Integer;
Options:Integer):Integer;
```

Arguments:

`BoardNum` Refers to the board number associated with the board when it was installed with the configuration program. The specified board must have an A/D. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`LowChan` First A/D channel of scan.

`HighChan` Last A/D channel of scan.

LowChan/HighChan: The maximum allowable channel depends on which type of A/D board is being used. For boards with both single ended and differential inputs, the maximum allowable channel number also depends on how the board is configured (e.g., 8 channels for differential inputs, 16 for single ended inputs).

`PretrigCount` Number of pre-trigger A/D samples to collect. Specifies the number of samples to collect before the trigger occurs. `PretrigCount` must be less than (`TotalCount` - 512).

If the trigger occurs too early, fewer than the requested number of pre-trigger samples will be collected, and a `TOOFEW` error will occur. The `PretrigCount` will be set to indicate how many samples were actually collected. The post trigger samples will still be collected.

`TotalCount` Total number of A/D samples to collect. Specifies the total number of samples that will be collected and stored in the buffer. `TotalCount` must be greater than or equal to the `PretrigCount` + 512.

If the trigger occurs too early, fewer than the requested number of samples will be collected, and a `TOOFEW` error will occur. The `TotalCount` will be set to indicate how many samples were actually collected.

`TotalCount` must be evenly divisible by the number of channels being scanned. If it is not, this function will adjust the number (down) to the next valid value and return that value to the `TotalCount` argument.

`Rate` Sample rate in scans per second.

`Range` A/D `Range` code. If the selected A/D board does not have a programmable gain feature, this argument is ignored. Otherwise, set to any range that is supported by the selected A/D board. See Table 2- on page 13 for valid values.

`MemHandle`	Handle for Windows buffer to store data in (Windows). This buffer must have been previously allocated with the `cbWinBufAlloc()` function. See the "Notes" section on page 31.
`Options`	Bit fields that control various options. This field may contain any combination of non-contradictory choices from the values listed in the "`Options` argument values" section below.

Returns:

Error code or 0 if no errors

`PretrigCount` - Number of pre-trigger samples

`TotalCount` - Total number of samples collected

`Rate` - actual sampling rate

`MemHandle` - Collected A/D data returned via the Windows buffer

Options argument values:

CONVERTDATA	The data is collected into a "circular" buffer. When the data collection is complete, the data is in the wrong order. If you use the CONVERTDATA option, the data is automatically rotated into the correct order (and converted to 12-bit values if required) when the data acquisition is complete. Otherwise, call `cbAConvertPretrigData()` to rotate the data. You cannot use the CONVERTDATA option in combination with the BACKGROUND option for this function.
BACKGROUND	If the BACKGROUND option is not used, the `cbAPretrig()` function will not return to your program until all of the requested data has been collected and returned to the buffer. When the BACKGROUND option is used, control returns immediately to the next line in your program, and the data collection from the A/D into the buffer will continue in the background. Use `cbGetStatus()` to check on the status of the background operation. Alternatively, some boards support `cbEnableEvent()` for event notification of changes in status of BACKGROUND scans. Use `cbStopBackground()` to terminate the background process before it has completed. Call `cbStopBackground()` after normal termination of all background functions to clear variables and flags. You cannot use the CONVERTDATA option in combination with the BACKGROUND option for this function. To correctly order and parse the data, use `cbAConvertPretrigData()` after the function completes.
EXTCLOCK	This option is available only for boards that have separate inputs for external pacer and external trigger. See your hardware manual or board-specific information.
EXTMEMORY	Causes this function to send the data to a connected memory board via the DT-Connect interface rather than returning the data to the buffer. If you use this option to send the data to a MEGA-FIFO memory board, then you must use `cbMemReadPretrig()` to later read the pre-trigger data from the memory board. If you use `cbMemRead()`, the data will NOT be in the correct order. Every time this option is used, it overwrites any data already stored in the memory board. All data should be read from the board (with `cbMemReadPretrig()`) before collecting any new data. When this option is used, the `MemHandle` argument is ignored. The MEGA-FIFO memory must be fully populated in order to use the `cbAPretrig()` function with the EXTMEMORY option.

`DTCONNECT`	When `DTCONNECT` option is used with this function the data from ALL A/D conversions is sent out the DT-Connect interface. While this function is waiting for a trigger to occur, it will send data out the DT-Connect interface continuously. If you have a Measurement Computing memory board plugged into the DT-Connect interface then you should use `EXTMEMORY` option rather than this option.

Notes:

IMPORTANT

The buffer referenced by `MemHandle` must be big enough to hold at least `TotalCount` + 512 integers.

cbATrig()

Waits for a specified analog input channel to go above or below a specified value. `cbATrig` continuously reads the specified channel and compares its value to `TrigValue`. Depending on whether `TrigType` is set to `TRIGABOVE` or `TRIGBELOW`, it waits for the first A/D sample that is above or below `TrigValue`. The first sample that meets the trigger criteria is returned to `DataValue`.

Function prototype:

C/C++:

```
int cbATrig ( int BoardNum, int Channel, int TrigType, int
TrigValue, int Range, unsigned short *DataValue )
```

Visual Basic:

```
Function cbATrig( ByVal BoardNum&, ByVal Channel&, ByVal TrigType&,
ByVal TrigValue%, ByVal Range&, DataValue% ) As Long
```

Delphi:

```
function cbATrig ( BoardNum:Integer; Channel:Integer;
TrigType:Integer; TrigValue:Word; Range:Integer; var DataValue:Word
):Integer;
```

Arguments:

`BoardNum` — Refers to the board number associated with the board when it was installed with the configuration program. The specified board must have an A/D. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`Channel` — A/D channel number. The maximum allowable channel depends on which type of A/D board is being used. For boards with both single ended and differential inputs, the maximum allowable channel number also depends on how the board is configured. For example a CIO-DAS1600 has 8 channels for differential inputs and 16 channels for single ended inputs.

`TrigType` — `TRIGABOVE` or `TRIGBELOW`. Specifies whether to wait for the analog input to be ABOVE or BELOW the specified trigger value.

`TrigValue` — The threshold value that all A/D values are compared to. Must be in the range 0 - 4095 for 12-bit A/D boards, or 0-65,535 for 16-bit A/D boards. Refer to your BASIC manual for information on signed BASIC integer data types.

`Range` — Gain code. If the selected A/D board does not have a programmable gain feature, this argument is ignored. Otherwise, set to any range that is supported by the selected A/D board. See Table 2- on page 13 for valid values.

`DataValue` — Returns the value of the first A/D sample to meet the trigger criteria.

Returns:

Error code or 0 if no errors

`DataValue` - value of first A/D sample to match the trigger criteria.

Notes:

Pressing **Ctrl-C** will not terminate the wait for an analog trigger that meets the specified condition. There are only two ways to terminate this call: satisfy the trigger condition or reset the computer.

Caution! Use caution when using this function in Windows programs. All active windows will lock on the screen until the trigger condition is satisfied. The keyboard and mouse activity will also lock until the trigger condition is satisfied.

Configuration Functions

Introduction

This section covers Universal Library functions that retrieve or change configuration options on a board. The configuration information for all boards is stored in the configuration file CB.CFG. This information is loaded from CB.CFG by all programs that use the library.

To determine which of these functions are compatible with your hardware, refer to the *Universal Library User's Guide* (available in PDF format on our website at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

cbGetConfig()

Returns a configuration option for a board. The configuration information for all boards is stored in the CB.CFG file. This information is loaded from CB.CFG by all programs that use the library. You can change the current configuration within a running program with the `cbSetConfig()` function. The `cbGetConfig()` function returns the current configuration information.

Function prototype:

C/C++:	`int cbGetConfig (int InfoType, int BoardNum, int DevNum, int ConfigItem, int *ConfigVal)`
Visual Basic:	`Function cbGetConfig(ByVal InfoType&, ByVal BoardNum&, ByVal DevNum&, ByVal ConfigItem&, ConfigVal&) As Long`
Delphi:	`function cbGetConfig (InfoType:Integer; BoardNum:Integer; DevNum:Integer; ConfigItem:Integer; var ConfigVal:Integer):Integer;`

Arguments:

`InfoType`	The configuration information for each board is grouped into different categories. This argument specifies which category you want. Set it to one of the constants listed in the "`InfoType` argument values" section below.
`BoardNum`	Refers to the board number associated with a board when it was installed. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`DevNum`	Selects a particular device. If `InfoType = DIGITALINFO`, then `DevNum` specifies which of the board's digital devices you want information on. If `InfoType = COUNTERINFO`, then `DevNum` specifies which of the board's counter devices you want info on.
`ConfigItem`	Specifies which configuration item you wish to retrieve. Set it in conjunction with the `InfoType` argument using the table in the "`ConfigItem` argument values" section on page 35.
`ConfigVal`	The specified configuration item is returned to this variable.

Returns:

Error code or 0 if no errors.

`ConfigVal` - returns the value of the specified configuration item here.

`InfoType` argument values:

`GLOBALINFO`	Information about the configuration file.
`BOARDINFO`	General information about a board.
`DIGITALINFO`	Information about a digital device.
`COUNTERINFO`	Information about a counter device.
`EXPANSIONINFO`	Information about an expansion device.
`MISCINFO`	One of the miscellaneous options for the board.

`ConfigItem` argument values:

Valid `ConfigItem` constant settings for each `InfoType` constant are as follows:

InfoType	ConfigItem	Description
GLOBALINFO	GIVERSION	CB.CFG file format - used by the library to determine compatibility.
	GINUMBOARDS	Maximum number of installable boards
	GINUMEXPBOARDS	Maximum number of expansion boards allowed to be installed.
BOARDINFO	BIBASEADR	Base address of board
	BIBOARDTYPE	Returns a unique number in the range of 0 to 8000 Hex describing the board type installed.
	BIINTLEVEL	Interrupt level. 0 for none or 1 - 15
	BIDMACHAN	DMA channel. 0, 1 or 3
	BIINITIALIZED	True (non-zero) or False (0) (16-bit library only)
	BICLOCK	Clock frequency in MHz (40, 10, 8, 6, 5, 4, 3, 2, 1) or 0 for not supported.
	BIRANGE	Selected voltage range. For switch selectable gains only. If the selected A/D board does not have a programmable gain feature, this argument returns the range as defined by the installed InstaCal settings. If InstaCal and the board are installed correctly, the returned range will correspond to the input range as set via the switches on the board. Refer to board specific information for a list of the A/D ranges supported by each board.
	BINUMADCHANS	Number of A/D channels
	BIUSESEXPS	Supports expansion boards TRUE/FALSE
	BIDINUMDEVS	Number of digital devices
	BIDIDEVNUM	Index into digital information for the first device.
	BICINUMDEVS	Number of counter devices
	BICIDEVNUM	Index into counter information for the first device.
	BINUMDACHANS	Number of D/A channels
	BIWAITSTATE	Setting of Wait State jumper. 1 = enabled, 0 = disabled
	BINUMIOPORTS	Number of IO Ports used by board
	BIPARENTBOARD	Board number of parent board (16-bit library only)
	BIDTBOARD	Board number of connected DT board
	BIDACUPDATEMODE	Setting of the update mode for a digital-to-analog converter (DAC). Refer to the "Notes" section on page 36 for more information.
	BIDACSTARTUP	Returns the setting of a DAC board's configuration register STARTUP bit. Refer to the "Notes" section for the `cbSetConfig()` method on page 45 for more information.
	BISERIALNUM	Returns the user serial number assigned to a USB device by *Insta*Cal.
DIGITALINFO	DIBASEADR	Base address (16-bit library only)
	DIINITIALIZED	True (non-zero) or False (0) (16-bit library only)
	DIDEVTYPE	Device Type - `AUXPORT`, `FIRSTPORTA` etc.
	DIMASK	Bit mask for this port (16-bit library only)
	DIREADWRITE	Read required before True/False (16-bit library only)
	DICONFIG	Current configuration INPUT or OUTPUT
	DINUMBITS	Number of bits in port
	DICURVAL	Current value of outputs
	DIINMASK	Returns the bit configuration of the specified port. Any of the lower eight bits that return a value of 1 are configured for input. Each of the upper eight bits always return 0. Refer to the "Notes" section on page 36 for more information.
	DOUTMASK	Returns the bit configuration of the specified port. Any of the lower eight bits that return a value of 1 are configured for output. Each of the upper eight bits always return 0. Refer to the "Notes" section on page 36 for more information.

InfoType	ConfigItem	Description
`COUNTERINFO`	`CIBASEADR`	Base address (16-bit library only)
	`CIINITIALIZED`	True (non-zero) or False (0) (16-bit library only)
	`CICTRTYPE`	Counter chip type. where: 1 = 8254, 2 = 9513 , 3 = 8536, 4 = 7266 or 5 = event counter
	`CICTRNUM`	Which counter on chip (16-bit library only)
	`CICONFIGBYTE`	Configuration Byte (16-bit library only)
`EXPANSIONINFO`	`XIBOARDTYPE`	Board type (refer to the "BoardType Codes" topic in the *Universal Library User's Guide)*
	`XIMUXADCHAN1`	A/D channel EXP board is connected to
	`XIMUXADCHAN2`	2nd A/D channel EXP board is connected to
	`XIRANGE1`	`Range` (gain) of low 16 channels
	`XIRANGE2`	`Range` (gain) of high 16 channels
	`XICJCCHAN`	A/D channel that CJC is connected to
	`XITHERMTYPE`	Sensor type. Use one of the sensor types listed below: J = 1 K = 2 T = 3 E = 4 R = 5 S = 6 B = 7 Platinum .00392 = 257 Platinum .00391 = 258 Platinum .00385 = 259 Copper .00427 = 260 Nickel/Iron .00581 = 261 Nickel/Iron .00527 = 262
	`XINUMEXPCHANS`	Number of channels on expansion board
	`XIPARENTBOARD`	Board number of parent A/D board

Notes:

- Use the `DIINMASK` and `DIOUTMASK` options to determine if an `AUXPORT` is configurable. Execute `cbGetConfig()` twice to the same port—once using `DIINMASK` and once using `DIOUTMASK`. If both of the `ConfigVal` arguments returned have input and output bits that overlap, the port is not configurable.

  You can determine overlapping bits by *And*ing both arguments: For example, the PCI-DAS08 has seven bits of digital I/O (four outputs and three inputs). For this board, the `ConfigVal` returned by `DIINMASK` is always 7 (`0000 0111`), while the `ConfigVal` argument returned by `DIOUTMASK` is always 15 (`0000 1111`). When you *And* both `ConfigVal` arguments together, you get a non-zero number (7). Any non-zero number indicates that input and output bits overlap for the specified port, and the port is a non-configurable `AUXPORT`.

- Use the `BIDACUPDATEMODE` option to check the update mode for a DAC board.

  With `ConfigItem` set to `BIDACUPDATEMODE`, if `ConfigVal` returns 0, the DAC update mode is immediate. Values written with `cbAOut()` are automatically output by the DAC channels.
  With `ConfigItem` set to `BIDACUPDATEMODE`, if `ConfigVal` returns 1, the DAC update mode is set to *on command*. Values written with `cbAOut()` are not output by the DAC channels until a `cbSetConfig()` call is made with its `ConfigItem` argument set to `BIDACUPDATECMD`.

- Use the `BIDACSTARTUP` option (`ConfigItem` argument) Returns 0 is if startup bit is disabled, or 1 to if startup bit is enabled to determine if the DAC values before the board was last powered down are stored.

  Refer to the "Notes" section for `cbSetConfig()` on page 45 for more information.

To store the current DAC values as start-up values, call `cbSetConfig()` with a value of 1 for the `BIDACSTARTUP` value. Then, call `cbAOut()` or `cbAOutScan()` for each channel, and call `cbSetConfig()` again with a value of 0 for the `BIDACSTARTUP` value.

Example:

```
cbSetConfig(BOARDINFO, boardNumber, 0, BIDACSTARTUP, 1);
for (int i =1; i <8; i++)
{
cbAOut(boardNumber, i, BIP5VOLTS, DACValue[i]);
}
cbSetConfig(BOARDINFO, boardNumber, 0, BIDACSTARTUP, 0);
```

To store the DAC's last settings, call `cbSetConfig()` with a `BIDACSTARTUP` value of 1. Leave this bit turned on until the application exits. The next time the board is powered up, it restores the values last written to the DACs.

cbGetSignal()

Retrieves the configured Auxiliary or DAQ Sync connection and polarity for the specified timing and control signal.

This function is intended for advanced users. Except for the SYNC_CLK input, you can easily view the settings for the timing and control signals using *Insta*Cal.

Note: This function is not supported by all board types.

Function prototype:

C/C++:
```
int cbGetSignal (int BoardNum, int Direction, int Signal, int Index,
int* Connection, int* Polarity)
```

Visual Basic:
```
Function cbGetSignal (ByVal BoardNum&, ByVal Direction&, ByVal
Signal&, ByVal Index&, ByRef Connection, ByRef Polarity) As Long
```

Delphi:
```
function cbGetSignal (BoardNum:Integer; Direction:Integer;
Signal:Integer; Index:Integer; var Connection:Integer; var
Polarity:Integer):Integer;
```

Arguments:

`BoardNum`
Refers to the board number associated with the A/D board when it was installed. The specified board must have configurable signal inputs and outputs.

`Direction`
Specifies whether retrieving the source (`SIGNAL_IN`) or destination (`SIGNAL_OUT`) of the specified signal.

`Signal`
Signal type whose connection is to be retrieved. See `cbSelectSignal()` on page 40 for valid signal types.

`Index`
Used to indicate which connection to reference when there is more than one connection associated with the output `Signal` type. When querying output signals, increment this value until `BADINDEX` is returned or 0 is returned via the `Connection` parameter to determine all the output Connections for the specified output `Signal`. The first `Connection` is indexed by 0.

For input signals (`Direction=SIGNAL_IN`), this should always be set to 0.

`Connection`
The specified connection is returned through this variable. Note that this is set to 0 if no connection is associated with the `Signal`, or if the `Index` is set to an invalid value.

`Polarity`
Holds the polarity for the associated `Signal` and `Connection`.

For output Signals assigned an AUXOUT `Connection`, the return value is either `INVERTED` or `NONINVERTED`.

For `Signal` settings of `ADC_CONVERT`, `DAC_UPDATE`, `ADC_TB_SRC` and `DAC_TB_SRC` input signals, either `POSITIVEEDGE` or `NEGATIVEEDGE` are returned.

All other signals return 0.

Returns:

Error code or 0 if no errors.

Notes:

The above timing and control configuration information can also be viewed and edited inside *Insta*Cal. To do this:

1. Run *Insta*Cal.

2. Click on the board and press the **Configure**... button or menu item.

   If the board supports DAQ Sync and Auxiliary Input/Output signal connections, a button labeled **Advanced Timing & Control Configuration** displays.

3. Press this button to open a display for viewing and modifying the above timing and control signals.

cbSelectSignal()

Configures timing and control signals to use specific Auxiliary or DAQ Sync connections as a source or destination.

This function is intended for advanced users. Except for the SYNC_CLK input, you can easily configure all the timing and control signals using *Insta*Cal.

Note: This function is not supported by all board types. Please refer to board specific information for details.

Function prototype:

C/C++:

```
int cbSelectSignal (int BoardNum, int Direction, int Signal, int Connection, int Polarity);
```

Visual Basic:

```
Function cbSelectSignal (ByVal BoardNum&, ByVal Direction&, ByVal Signal&, ByVal Connection&, ByVal Polarity&) as Long
```

Delphi:

```
Function cbSelectSignal (BoardNum:Integer; Direction:Integer; Signal:Integer; Connection:Integer; Polarity:Integer) : Integer; StdCall;
```

Arguments:

`BoardNum`	Refers to the board number associated with the A/D board when it was installed. The specified board must have configurable signal inputs and outputs.
`Direction`	Direction of the specified signal type to be assigned a connector pin. For most signal types, this should be either SIGNAL_IN or SIGNAL_OUT. For the SYNC_CLK , ADC_TB_SRC and DAC_TB_SRC signals, the external source can also be disabled by specifying `DISABLED`(=0) such that it is neither input nor output. Set it in conjunction with the `Signal`, `Connection`, and `Polarity` arguments using the tables in the "Direction argument values" starting on page 41.
`Signal`	Signal type to be associated with a connector pin. Set it to one of the constants listed in the "Signal argument values" section below.
`Connection`	Designates the connector pin to associate the signal type and direction. Since individual pin selection is not allowed for the DAQ-Sync connectors, all DAQ-Sync pin connections are referred to as DS_CONNECTOR. The AUXIN and AUXOUT settings match their corresponding hardware pin names.
`Polarity`	`ADC_TB_SRC` and `DAC_TB_SRC` input signals (`SIGNAL_IN`) can be set for either rising edge (`POSITIVEEDGE`) or falling edge (`NEGATIVEEDGE`) signals. The `AUXOUT` connections can be set to `INVERTED` or `NONINVERTED` from their internal polarity.

Returns:

Error code or 0 if no errors.

`Signal` argument values:

`ADC_CONVERT`	A/D conversion pulse or clock.
`ADC_GATE`	External gate for A/D conversions.
`ADC_SCANCLK`	A/D channel scan signal.
`ADC_SCAN_STOP`	A/D scan completion signal.
`ADC_SSH`	A/D simultaneous sample and hold signal.
`ADC_STARTSCAN`	Start of A/D channel-scan sequence signal.
`ADC_START_TRIG`	A/D scan start trigger.

ADC_STOP_TRIG	A/D stop- or pre- trigger.
ADC_TB_SRC	A/D pacer timebase source.
CTR1_CLK	CTR1 clock source.
CTR2_CLK	CTR2 clock source.
DAC_START_TRIG	D/A start trigger.
DAC_TB_SRC	D/A pacer timebase source.
DAC_UPDATE	D/A update signal.
DGND	Digital ground.
SYNC_CLK	STC timebase signal.

Direction argument values:

Valid input (`Direction=SIGNAL_IN`) settings include:

Signal	Connection	Polarity
ADC_CONVERT	AUXIN0..AUXIN5 DS_CONNECTOR	POSITIVEEDGE or NEGATIVEEDGE
ADC_GATE	AUXIN0..AUXIN5	See cbSetTrigger().
ADC_START_TRIG	AUXIN0..AUXIN5 DS_CONNECTOR	See cbSetTrigger().
ADC_STOP_TRIG	AUXIN0..AUXIN5 DS_CONNECTOR	See cbSetTrigger()
ADC_TB_SRC	AUXIN0..AUXIN5	POSITIVEEDGE or NEGATIVEEDGE
DAC_START_TRIG	AUXIN0..AUXIN5 DS_CONNECTOR	Not assigned here.
DAC_TB_SRC	AUXIN0..AUXIN5	POSITIVEEDGE or NEGATIVEEDGE
DAC_UPDATE	AUXIN0..AUXIN5 DS_CONNECTOR	POSITIVEEDGE or NEGATIVEEDGE
SYNC_CLK	DS_CONNECTOR	Not assigned here.

Valid output (`Direction=SIGNAL_OUT`) settings include:

Signal	Connection	Polarity
ADC_CONVERT	AUXOUT0..AUXOUT2 DS_CONNECTOR	INVERTED* or NONINVERTED
ADC_SCANCLK	AUXOUT0..AUXOUT2	
ADC_SCAN_STOP	AUXOUT0..AUXOUT2	
ADC_SSH	AUXOUT0..AUXOUT2	
ADC_STARTSCAN	AUXOUT0_AUXOUT2	
ADC_START_TRIG	AUXOUT0..AUXOUT2 DS_CONNECTOR	
ADC_STOP_TRIG	AUXOUT0..AUXOUT2 DS_CONNECTOR	
CTR1_CLK	AUXOUT0_AUXOUT2	
CTR2_CLK	AUXOUT0_AUXOUT2	
DAC_START_TRIG	AUXOUT0..AUXOUT2 DS_CONNECTOR	
DAC_UPDATE	AUXOUT0..AUXOUT2 DS_CONNECTOR	
DGND	AUXOUT0_AUXOUT2	Not assigned here.
SYNC_CLK	DS_CONNECTOR	Not assigned here.

* `INVERTED` is only valid for Auxiliary Output (`AUXOUT`) connections.

Valid disabled settings (`Direction=DISABLED`):

Signal	Connection	Polarity
ADC_TB_SRC	Not assigned here.	Not assigned here.
DAC_TB_SRC		
SYNC_CLK		

Notes:

- You can view and edit the above timing and control configuration information from *Insta*Cal. Open *Insta*Cal, click on the board, and press the "Configure..." button or menu item. If the board supports DAQ Sync and Auxiliary Input/Output signal connections, a button labeled "Advanced Timing & Control Configuration" displays. Press that button to open a display for viewing and modifying the above timing and control signals.

- Except for the `ADC_TB_SRC`, `DAC_TB_SRC` and `SYNC_CLK` signals, selecting an input signal connection does not necessarily activate it. However, assigning an output signal to a connection does activate the signal upon performing the respective operation. For instance, when running an `EXTCLOCK cbAInScan()`, `ADC_CONVERT SIGNAL_IN` selects the connection to use as an external clock to pace the A/D conversions; if `cbAInScan()` is run without setting the `EXTCLOCK` option, however, the selected connection is not activated and the signal at that connection is ignored. In both cases, the `ADC_CONVERT` signal is output via the connection(s) selected for the `ADC_CONVERT SIGNAL_OUT`. Since there are no scan options for enabling the Timebase Source and the `SYNC_CLK`, selecting an input for the A/D or D/A Timebase Source, or `SYNC_CLK` does activate the input source for the next respective operations.

- Multiple input signals can be mapped to the same `AUXIN`*n* connection by successive calls to `cbSelectSignal`; however, only one connection can be mapped to each input signal. If another connection had already been assigned to an input signal, the former selection is de-assigned and the new connection is assigned.

- Only one output signal can be mapped to the same `AUXOUT`*n* connection; however, multiple connections can be mapped to the same output signal by successive calls to `cbSelectSignal`. If an output signal had already been assigned to a connection, then the former output signal is de-assigned and the new output signal is assigned to the connection. Note that there are at most `MAX_CONNECTIONS` (=4) connections that can be assigned to each output signal.

- When selecting `DS_CONNECTOR` for a signal, only one direction per signal type can be defined at a given time. Attempting to assign both directions of a signal to the `DS_CONNECTOR` results in only the latest selection being applied. If the signal type had formerly been assigned an input direction from the `DS_CONNECTOR`, assigning the output direction for that signal type results in the input signal being reassigned to its default connection.

Default Input Signal Connections	Input signal	Default connection
	ADC_ CONVERT	AUXIN0
	ADC_ GATE	AUXIN5
	ADC_START_TRIG	AUXIN1
	ADC_STOP_TRIG	AUXIN2
	DAC_ UPDATE	AUXIN3
	DAC_START_TRIG	AUXIN3

- `ADC_TB_SRC` and `DAC_TB_SRC` are intended to synchronize the timebase of the analog input and output pacers across two or more boards. Internal calculations of sampling and update rates assume that the external timebase has the same frequency as its internal clock. Adjust sample rates to compensate for differences in clock frequencies.

For instance, if the external timebase has a frequency of 10 MHz on a board that has a internal clock frequency of 40 MHz, the scan function samples or updates at a rate of about 1/4 the rate entered. However, while compensating for differences in external timebase and internal clock frequency, if the rate entered results in an invalid pacer count, the function returns a BADRATE error.

cbSetConfig()

Sets a configuration option for a board. The configuration information for all boards is stored in the CB.CFG file. All programs that use the library read this file. You can use this function to override the configuration information stored in the CB.CFG file.

Function prototype:

C/C++:	`int cbSetConfig(int InfoType, int BoardNum, int DevNum, int ConfigItem, int ConfigVal)`
Visual Basic:	`Function cbSetConfig(ByVal InfoType&, ByVal BoardNum&, ByVal DevNum&, ByVal ConfigItem&, ByVal ConfigVal&) As Long`
Delphi:	`function cbSetConfig(InfoType:Integer; BoardNum:Integer; DevNum:Integer; ConfigItem:Integer; ConfigVal:Integer):Integer;`

Arguments:

`InfoType`	The configuration information for each board is grouped into different categories. `InfoType` specifies which category you want. Set it to one of the constants listed in the "`InfoType` " section below.
`BoardNum`	Refers to the board number associated with a board when it was installed. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`DevNum`	Selects a particular device. If `InfoType` = `DIGITALINFO`, then `DevNum` specifies which of the board's digital devices you want to set information on. If `InfoType` = `COUNTERINFO` then `DevNum` specifies which of the board's counter devices you want to set information on.
`ConfigItem`	Specifies which configuration item you wish to set. Set it in conjunction with the `InfoType` argument using the table under "`ConfigItem` argument values" on page 45.
`ConfigVal`	The value to set the specified configuration item to.

Returns:

Error code or 0 if no errors.

`InfoType` argument values:

`BOARDINFO`	General information about a board.
`DIGITALINFO`	Information about a digital device.
`COUNTERINFO`	Information about a counter device.
`EXPANSIONINFO`	Information about an expansion device.
`MISCINFO`	One of the miscellaneous options for the board.

ConfigItem argument values:

InfoType	ConfigItem	Description
BOARDINFO	BIBASEADR	Base address of board
	BIINTLEVEL	Interrupt level
	BIDMACHAN	DMA channel
	BICLOCK	Clock frequency in MHz (1, 4, 6 or 10)
	BIRANGE	Selected voltage range
	BINUMADCHANS	Number of A/D channels
	BIWAITSTATE	Sets the Wait State jumper
	BIDACUPDATEMODE	Sets the update mode for a digital-to-analog converter (DAC). Use this setting in conjunction with one of these ConfigVal settings: UPDATEIMMEDIATE UPDATEONCOMMAND Refer to the "Notes" section below for more information.
	BIDACUPDATECMD	Updates all analog output channels. When ConfigItem is set to BIDACUPDATECMD, the DevNum and ConfigVal arguments are not used and can be set to 0. Refer to the "Notes" section below for more information.
	BIDACSTARTUP	Sets the board's configuration register STARTUP bit to 0 or 1 to enable/disable the storing of digital-to-analog converter (DAC) startup values. Each time the board is powered up, the stored values are written to the DACs. Refer to the "Notes" section below for more information.
	BICALOUTPUT	Sets the voltage for the CAL pin on supported USB devices.
	BISRCADPACER	Outputs the A/D pacer signal to the SYNC pin on supported USB devices.
EXPANSIONINFO	XIMUXADCHAN1	A/D channel board is connect to
	XIMUXADCHAN2	2nd A/D channel board is connected to
	XIRANGE1	Range (gain) of low 16 channels
	XIRANGE2	Range (gain) of high 16 channels
	XICJCCHAN	A/D channel that CJC is connected to
	XITHERMTYPE	Thermocouple type

Notes:

Use the BIDACSTARTUP option (ConfigItem argument) to store either the current DAC values, or the DAC values before the board was last powered down.

- To store the current DAC values as start-up values, call cbSetConfig() with a value of 1 for the BIDACSTARTUP value. Then, call cbAOut() or cbAOutScan() for each channel (), and call cbSetConfig() again with a value of 0 for the BIDACSTARTUP value.

  Example:

```
cbSetConfig(BOARDINFO, boardNumber, 0, BIDACSTARTUP, 1);
for (int i =1; i <8; i++)
{
cbAOut(boardNumber, i, BIP5VOLTS, DACValue[i]);

}
cbSetConfig(BOARDINFO, boardNumber, 0, BIDACSTARTUP, 0);
```

- To store the DAC's last settings, call cbSetConfig() with a BIDACSTARTUP value of 1. Leave this bit turned on until the application exits. The next time the board is powered up, it restores the values last written to the DACs.

Use the `BIDACUPDATEMODE` option (ConfigItem argument) to set the update mode for a DAC board.

- With `ConfigItem` set to `BIDACUPDATEMODE`, and `ConfigVal` set to 0, the DAC update mode is *immediate*. Values written with `cbAOut()` or `cbAOutScan()` are automatically output by the DAC channels
- With `ConfigItem` set to `BIDACUPDATEMODE` and `ConfigVal` set to 1, the DAC update mode is *on command*. Values written with `cbAOut()` or `cbAOutScan()` are not output by the DAC channels until another `cbSetConfig()` call is made with `ConfigItem` set to `BIDACUPDATECMD`.

cbSetTrigger()

Selects the trigger source and sets up its parameters. This trigger is used to initiate analog to digital conversions using the following Universal Library functions:

- `cbAInScan()`, if the `EXTTRIGGER` option is selected.
- `cbAPretrig()`
- `cbFilePretrig()`

Function prototype:

C/C++:
```
int cbSetTrigger (int BoardNum, int TrigType, unsigned short
LowThreshold, unsigned short HighThreshold);
```

Visual Basic:
```
Function cbSetTrigger (ByVal BoardNum&, ByVal TrigType&, ByVal
LowThreshold%, ByVal HighThreshold%) As Long
```

Delphi:
```
Function cbSetTrigger (BoardNum:Integer; TrigType:Integer;
LowThreshold:Word; HighThreshold:Word):Integer;
```

Arguments:

`BoardNum` Specifies the board number associated with the board when it was installed with the configuration program. The board must have the software selectable triggering source and/or options. `BoardNum` may be 0 to 99 (0 to 9 for the 16-bit version of the Universal Library).

`TrigType` Specifies the type of triggering based on the external trigger source. Set it to one of the constants in the "TrigType argument values" section on page 48.

`LowThreshold` Selects the low threshold used when the trigger input is analog. The range depends upon the resolution of the trigger circuitry. Must be 0 to 255 for 8-bit trigger circuits, 0 to 4095 for 12-bit trigger circuits, and 0 to 65535 for 16-bit trigger circuits. Refer to the "Notes" section on page 48.

`HighThreshold` Selects the high threshold used when the trigger input is analog. The range depends upon the resolution of the trigger circuitry. Must be 0 to 255 for 8-bit trigger circuits, 0 to 4095 for 12-bit trigger circuits, and 0 to 65535 for 16-bit trigger circuits. Refer to the "Notes" section on page 48.

Returns:

Error code or 0 if no errors.

`TrigType` argument values:

Trigger Source	TrigType	Explanation
Analog	`GATE_NEG_HYS`	AD conversions are enabled when the external analog trigger input is more positive than HighThreshold. AD conversions are disabled when the external analog trigger input more negative than Low/Threshold. Hysteresis is the level between Low/Threshold and HighThreshold.
	`GATE_POS_HYS`	AD conversions are enabled when the external analog trigger input is more negative than LowThreshold. AD conversions are disabled when the external analog trigger input is more positive than HighThreshold. Hysteresis is the level between LowThreshold and HighThreshold.
	`GATE_ABOVE`	AD conversions are enabled as long as the external analog trigger input is more positive than HighThreshold.
	`GATE_BELOW`	AD conversions are enabled as long as the external analog trigger input is more negative than LowThreshold.
	`TRIG_ABOVE`	AD conversions are enabled when the external analog trigger makes a transition from below HighThreshold to above. Once conversions are enabled, the external trigger is ignored.
	`TRIG_BELOW`	AD conversions are enabled when the external analog trigger input makes a transition from above LowThreshold to below. Once conversions are enabled, the external trigger is ignored.
	`GATE_IN_WINDOW`	AD conversions are enabled as long as the external analog trigger is inside the region defined by LowThreshold and HighThreshold.
	`GATE_OUT_WINDOW`	AD conversions are enabled as long as the external analog trigger is outside the region defined by LowThreshold and HighThreshold.
Digital	`GATE_HIGH`	AD conversions are enabled as long as the external digital trigger input is 5 V (logic HIGH or 1).
	`GATE_LOW`	AD conversions are enabled as long as the external digital trigger input is 0 V (logic LOW or 0).
	`TRIG_HIGH`	AD conversions are enabled when the external digital trigger is 5 V (logic HIGH or '1'). Once conversions are enabled, the external trigger is ignored.
	`TRIG_LOW`	AD conversions are enabled when the external digital trigger is 0 V (logic LOW or '0'). Once conversions are enabled, the external trigger is ignored.
	`TRIG_POS_EDGE`	AD conversions are enabled when the external digital trigger makes a transition from 0 V to 5 V (logic LOW to HIGH). Once conversions are enabled, the external trigger is ignored.
	`TRIG_NEG_EDGE`	AD conversions are enabled when the external digital trigger makes a transition from 5 V to 0 V (logic HIGH to LOW). Once conversions are enabled, the external trigger is ignored.

Notes:

The threshold value must be within the range of the analog trigger circuit associated with the board. Refer to the board-specific information in the *Universal Library User's Guide*. For example, on the PCI-DAS 1602/16, the analog trigger circuit handles ±10 V. A value of 0 corresponds to -10 V, whereas a value of 65535 corresponds to +10 V.

Since Visual Basic does not support unsigned integer types, the thresholds range from -32768 to 32767 for 16-bit boards, instead of from 0 to 65535. In this case, the unsigned value of 65535 corresponds to a value of -1, 65534 corresponds to -2, ..., 32768 corresponds to -32768.

For most boards that support analog triggering, you can pass the required trigger voltage level and the appropriate `Range` to `cbFromEngUnits/FromEngUnits` to calculate the `HighThreshold` and `LowThreshold` values.

For some boards (refer to the "Analog Input Boards" chapter in the *Universal Library User's Guide*), you must manually calculate the threshold by first calculating the least significant bit (LSB) for a particular range for the trigger resolution of your hardware. You then use the LSB to find the threshold in counts based on an analog voltage trigger threshold.

To calculate the threshold, do the following:

1. Calculate the LSB by dividing the full scale range (FSR) by $2^{resolution}$. FSR is the entire span from – FS to +FS of your hardware for a particular range. For example, the full scale range of ±10 V is 20 V.

2. Calculate how many times you need to add the LSB calculate in step 1 to the negative full scale (-FS) to reach the trigger threshold value.

The maximum threshold value is $2^{resolution}$ - 1. The formula is shown here:

Abs (-FS - threshold in volts) ÷ (LSB) = threshold in counts

Here are two examples that use this formula—one for 8-bit trigger resolution and one for 12-bit trigger resolution.

- 8-bit example using the ±10 V range with a -5 V threshold:

  Calculate LSB: LSB = 20 ÷ 2^8 = 20 ÷ 256 = .078125
  Calculate threshold: Abs(-10 - (-5)) ÷ .078125 = 5 ÷ .078125 = 64 (round this result if it is not an integer). A count of 64 translates to a voltage threshold of -5.0 V.

- 12-bit example using the ±10 V range with a +1 V threshold:

  Calculate LSB: LSB = 20 ÷ 2^{12} = 20 ÷ 4096 = .00488
  Calculate threshold: Abs(-10 - 1) ÷ .00488 = 11 ÷ .00488 = 2254 (rounded from 2254.1). A count of 2254 translates to a voltage threshold of 0.99952 V.

Counter Functions

Introduction

This section covers Universal Library functions that load, read, and configure counters. There are five types of counter chips used in MCC counter boards: 8254s, 8536s, 7266s, 9513s and generic event counters. Some of the counter commands only apply to one type of counter.

cbC7266Config() (32-bit UL only)

Configures 7266 counter for desired operation. This function can only be used with boards that contain a 7266 counter chip (Quadrature Encoder boards). For more information, refer to the LS7266R1 data sheet in the accompanying ls7266r1.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default).

This data sheet is also available on our web site at www.mccdaq.com/PDFmanuals/LS7266R1.pdf

Function prototype:

C/C++:

```
int cbC7266Config( int BoardNum, int CounterNum, int Quadrature, int
CountingMode, int DataEncoding, int IndexMode, int InvertIndex, int
FlagPins, int Gating )
```

Visual Basic:

```
Function cbC7266Config( ByVal BoardNum&, ByVal CounterNum&, ByVal
Quadrature&, ByVal CountingMode&, ByVal DataEncoding&, ByVal
IndexMode&, ByVal InvertIndex&, ByVal FlagPins&, ByVal Gating& ) As
Long
```

Delphi:

```
function cbC7266Config( BoardNum:Integer; CounterNum:Integer;
Quadrature:Integer; CountingMode:Integer; DataEncoding:Integer;
IndexMode:Integer; InvertIndex:Integer; FlagPins:Integer;
Gating:Integer ):Integer;
```

Arguments:

`BoardNum` — Refers to the board number associated with the board when it was installed with the configuration program. The specified board must have an LS7266 counter. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`CounterNum` — Counter Number (1 - n), where n is the number of counters on the board.

`Quadrature` — Selects the resolution multiplier for quadrature input, or disables quadrature input (`NO_QUAD`) so that the counters can be used as standard TTL counters. `NO_QUAD`, `X1_QUAD`, `X2_QAUD or X4_QUAD`.

`CountingMode` — Selects operating mode for the counter. `NORMAL_MODE`, `RANGE_LIMIT`, `NO_RECYCLE`, `MODULO_N`. Set it to one of the constants in the "`CountingMode` argument values" section on page 53.

`DataEncoding` — Selects the format of the data that is returned by the counter - either Binary or BCD format. `BCD_ENCODING`, `BINARY_ENCODING`.

`IndexMode` — Selects which action will be taken when the `Index` signal is received. The `IndexMode` must be set to `INDEX_DISABLED` whenever a `Quadrature` is set to `NON_QUAD` or when Gate is set to `ENABLED`. Set it to one of the constants in the "`IndexMode` argument values" section on page 53.

`InvertIndex` — Selects the polarity of the `Index` signal. If set to `DISABLED` the `Index` signal is assumed to be positive polarity. If set to `ENABLED` the `Index` signal is assumed to be negative polarity.

`FlagPins` — Selects which signals will be routed to the FLG1 and FLG2 pins. Set it to one of the constants in the "`FlagPins` argument values" section on page 53.

`Gating` — If gating is set to `ENABLED` then the RCNTR pin will be used as a gating signal for the counter. Whenever `Gating=ENABLED` the `IndexMode` must be set to `DISABLE_INDEX`.

Returns:

Error code or 0 if no error occurs

`CountingMode` argument values:

NORMAL_MODE	Each counter operates as a 24-bit counter that rolls over to 0 when the maximum count is reached.
RANGE_LIMIT	In range limit count mode, an upper an lower limit is set, mimicking limit switches in the mechanical counterpart. The upper limit is set by loading the PRESET register with the `cbCLoad()` function after the counter has been configured. The lower limit is always 0. When counting up, the counter freezes whenever the count reaches the value that was loaded into the PRESET register. When counting down, the counter freezes at 0. In either case the counting is resumed only when the count direction is reversed.
NO_RECYCLE	In non-recycle mode the counter is disabled whenever a count overflow or underflow takes place. The counter is re-enabled when a reset or load operation is performed on the counter.
MODULO_N	In modulo-n mode, an upper limit is set by loading the PRESET register with a maximum count. Whenever counting up, when the maximum count is reached, the counter will roll-over to 0 and continue counting up. Likewise when counting down, whenever the count reaches 0, it will roll over to the maximum count (in the PRESET register) and continue counting down.

`IndexMode` argument values:

INDEX_DISABLED	The `Index` signal is ignored.
LOAD_CTR	The counter is loaded whenever the `Index` signal ON the `LCNTR` pin occurs.
LOAD_OUT_LATCH	The current count is latched whenever the `Index` signal on the `LCNTR` pin occurs. When this mode is selected, the `cbCIn()` function will return the same count each time it is called until the `Index` signal occurs.
RESET_CTR	The counter is reset to 0 whenever the `Index` signal on the `RCNTR` pin occurs.

`FlagPins` argument values:

CARRY_BORROW	FLG1 pin is `CARRY` output, FLG2 is `BORROW` output.
COMPARE_BORROW	FLG1 pin is `COMPARE` output, FLG2 is `BORROW` output.
CARRYBORROW_UPDOWN	FLG1 pin is `CARRY/BORROW` output, FLG2 is `UP/DOWN` signal.
INDEX_ERROR	`FLG1` is `INDEX` output, FLG2 is error output.

cbC8254Config()

Configures 8254 counter for desired operation. This function can only be used with 8254 counters. For more information, refer to the 82C54 data sheet in the accompanying 82C54.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default).

This data sheet is also available on our web site at www.mccdaq.com/PDFmanuals/82C54.pdf

Function prototype:

C/C++:

```
int cbC8254Config( int BoardNum, int CounterNum, int Config )
```

Visual Basic:

```
Function cbC8254Config( ByVal BoardNum&, ByVal CounterNum&, ByVal
Config& ) As Long
```

Delphi:

```
function cbC8254Config(BoardNum:Integer; CounterNum:Integer;
Config:Integer ):Integer;
```

Arguments:

`BoardNum`	Refers to the number associated with the board when it was installed with the configuration program. Board must have an 82C54 installed. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`CounterNum`	Selects one of the counter channels. An 8254 has 3 counters. The value may be 1 - n, where n is the number of 8254 counters on the board (see board-specific information in the *Universal Library User's Guide* available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf, or in your hardware manual).
`Config`	Refer to the 8254 data sheet for a detailed description of each of the configurations. Set it to one of the constants in the "`Config` " section below.

Returns:

Error code or 0 if no errors

`Config` argument values:

`HIGHONLASTCOUNT`	Output of counter (OUT N) transitions from low to high on terminal count and remains high until reset. See `Mode` 0 on 8254 data sheet in accompanying 82C54.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default).
`ONESHOT`	Output of counter (OUT N) transitions from high to low on rising edge of GATE N, then back to high on terminal count. See mode 1 on 8254 data sheet in accompanying 82C54.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default).
`RATEGENERATOR`	Output of counter (OUT N) pulses low for one clock cycle on terminal count, reloads counter and recycles. See mode 2 on 8254 data sheet in the accompanying 82C54.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default).
`SQUAREWAVE`	Output of counter (OUT N) is high for count < 1/2 terminal count then low until terminal count, whereupon it recycles. This mode generates a square wave. See mode 3 on 8254 data sheet in accompanying 82C54.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default).
`SOFTWARESTROBE`	Output of counter (OUT N) pulses low for one clock cycle on terminal count. Count starts after counter is loaded. See mode 4 on 8254 data sheet in accompanying 82C54.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default).

`HARDWARESTROBE`	Output of counter (OUT N) pulses low for one clock cycle on terminal count. Count starts on rising edge at GATE N input. See mode 5 on 8254 data sheet in accompanying 82C54.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default).

cbC8536Config()

Configures 8536 counter for desired operation. This function can only be used with 8536 counters. Refer to the Zilog 8536 manual. (This manual is available from MCC, but is not available on our web site.)

Function prototype:

C/C++:	`int cbC8536Config( int BoardNum, int CounterNum, int OutputControl, int RecycleMode, int Retrigger)`
Visual Basic:	`Function cbC8536Config( ByVal BoardNum&, ByVal CounterNum&, ByVal OutputControl&, ByVal RecycleMode&, ByVal Retrigger& ) As Long`
Delphi:	`function cbC8536Config( BoardNum:Integer; CounterNum:Integer; OutputControl:Integer; RecycleMode:Integer; Retrigger:Integer ):Integer;`

Arguments:

`BoardNum`	Refers to the board number associated with the board when it was installed with the *InstaCal®* configuration program. The board must have an 8536. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`CounterNum`	Selects one of the counter channels. An 8536 has 3 counters. The value may be 1, 2 or 3.
`OutputControl`	Specifies the action of the output signal. Set it to one of the constants in the "OutputControl " section below.
`RecycleMode`	If set to `RECYCLE` (as opposed to `ONETIME`) then the counter will automatically reload to the starting count every time it reaches 0, then counting will continue.
`Retrigger`	If set to `ENABLED` (`CBENABLED` in Visual Basic and Delphi) then every trigger on the counter's trigger input will initiate loading of the initial count and counting will proceed from initial count.

Returns:

Error code or 0 if no errors

`OutputControl` argument values:

`HIGHPULSEONTC`	Output transitions from low to high for one clock pulse on terminal count.
`TOGGLEONTC`	Output will change state on terminal count.
`HIGHUNTILTC`	Output will transition to high at the start of counting then go low on terminal count.

cbC9513Config()

Sets all of the configurable options of a 9513 counter. For more information, refer to the AM9513A data sheet in the 9513A.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default).

The data sheet is also available on our web site at www.mccdaq.com/PDFmanuals/9513A.pdf

Function prototype:

C/C++:

```
int cbC9513Config( int BoardNum, int CounterNum, int GateControl,
int CounterEdge, int CountSource, int SpecialGate, int Reload, int
RecycleMode, int BCDMode, int CountDirection, int OutputControl );
```

Visual Basic:

```
Function cbC9513Config( ByVal BoardNum&, ByVal CounterNum&, ByVal
GateControl&, ByVal CounterEdge&, ByVal CountSource&, ByVal
SpecialGate&, ByVal Reload&, ByVal RecycleMode&, ByVal BCDMode&,
ByVal CountDirection&, ByVal OutputControl& ) As Long
```

Delphi:

```
function cbC9513Config( BoardNum:Integer; CounterNum:Integer;
GateControl:Integer; CounterEdge:Integer; CountSource:Integer;
SpecialGate:Integer; Reload:Integer; RecycleMode:Integer;
BCDMode:Integer; CountDirection:Integer;
OutputControl:Integer):Integer;
```

Arguments:

`BoardNum`	Refers to the board number associated with the board when it was installed with the configuration program. The specified board must have a 9513 counter. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`CounterNum`	Counter number (1 - n) where n is the number of counters on the board. For example, a CIO-CTR5 has 5, a CIO-CTR10 has 10, etc. See board specific info.
`GateControl`	Sets the gating response for level, edge, etc. Set it to one of the constants in the "GateControl argument values" section on page 58.
`CounterEdge`	Which edge to count. Referred to as "Source Edge" in 9513 data book. Can be set to `POSITIVEEDGE` (count on rising edge) or `NEGATIVEEDGE` (count on falling edge).
`CountSource`	Each counter may be set to count from one of 16 internal or external sources. Set it to one of the constants in the "CountSource argument values" section on page 58.
`SpecialGate`	Special gate may be enabled or disabled (`CBENABLED` or `CBDISABLED`: in Visual Basic or Delphi).
`Reload`	Reload the counter from the load register (Reload = `LOADREG`) or alternately load from the load register, then the hold register (Reload = `LOADANDHOLDREG`).
`RecycleMode`	Execute once (`RecycleMode` = `ONETIME`) or reload and recycle (`RecycleMode` = `RECYCLE`).
`BCDMode`	Counter may operate in *binary coded decimal* count (`ENABLED`) or *binary* count (`DISABLED`) (CBENABLED or CBDISABLED in Visual Basic or Delphi).
`CountDirection`	AM9513 may count up (`COUNTUP`) or down (`COUNTDOWN`).
`OutputControl`	The type of output desired. Set it to one of the constants in the "OutputControl argument values" section on page 58.

Returns:

Error code or 0 if no errors

GateControl argument values:

`NOGATE`	No gating
`AHLTCPREVCTR`	Active high TCN -1
`AHLNEXTGATE`	Active High Level GATE N + 1
`AHLPREVGATE`	Active High Level GATE N - 1
`AHLGATE`	Active High Level GATE N
`ALLGATE`	Active Low Level GATE N
`AHEGATE`	Active High Edge GATE N
`ALEGATE`	Active Low Edge GATE N

CountSource argument values:

`TCPREVCTR`	TCN - 1 (Terminal count of previous counter)
`CTRINPUT1`	SRC 1 (Counter Input 1)
`CTRINPUT2`	SRC 2 (Counter Input 2)
`CTRINPUT3`	SRC 3 (Counter Input 3)
`CTRINPUT4`	SRC 4 (Counter Input 4)
`CTRINPUT5`	SRC 5 (Counter Input 5)
`GATE1`	GATE1
`GATE2`	GATE2
`GATE3`	GATE3
`GATE4`	GATE4
`GATE5`	GATE 5
`FREQ1`	F1
`FREQ2`	F2
`FREQ3`	F3
`FREQ4`	F4
`FREQ5`	F5
`ALWAYSLOW`	Inactive, Output Low

OutputControl argument values:

`HIGHPULSEONTC`	High pulse on Terminal Count
`TOGGLEONTC`	TC Toggled
`DISCONNECTED`	Inactive, Output High Impedance
`LOWPULSEONTC`	Active Low Terminal Count Pulse
3, 6, 7	(numeric values) Illegal

Notes:

The information provided here and in the `cbC9513Init()` data sheet will only help you understand how Universal Library syntax corresponds to information in the 9513 data sheet. It is not a substitute for the data sheet. You cannot program and use a 9513 without this data sheet.

Refer to the accompanying 9513A.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default). The data sheet is also available on our web site at www.mccdaq.com/PDFmanuals/9513A.pdf.

cbC8536Init()

Initializes the counter linking features of an 8536 counter chip. Refer to the *Zilog 8536* manual, "Counter/Timer Link Controls" section, for a complete description of the hardware affected by this mode. (This manual is available from MCC, but is not available on our web site.) Counters 1 and 2 must be linked before enabling the counters.

Function prototype:

C/C++:	`int cbC8536Init( int BoardNum, int ChipNum, int CtrlOutput )`
Visual Basic:	`Function cbC8536Init( ByVal BoardNum&, ByVal ChipNum&, ByVal Ctr1Output& ) As Long`
Delphi:	`function cbC8536Init( BoardNum:Integer; ChipNum:Integer; Ctr1Output:Integer ):Integer;`

Arguments:

`BoardNum`	Refers to the board number associated with the board when it was installed with the *InstaCal*® configuration program. The specified board must have an 8536. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`ChipNum`	Selects one of the 8536 chips on the board, 1 to *n*.
`CtrlOutput`	Specifies how counter 1 is to be linked to counter 2, if at all. Set it to one of the constants in the "CtrlOutput argument values" section below.

Returns:

Error code or 0 if no errors.

CtrlOutput argument values:

`NOTLINKED`	Counter 1 is not connected to any other counters inputs.
`GATECTR2`	Output of counter 1 is connected to the GATE of counter #2.
`TRIGCTR2`	Output of counter 1 is connected to the trigger of counter #2.
`INCTR2`	Output of counter 1 is connected to counter #2 clock input.

cbC9513Init()

Initializes all of the chip level features of a 9513 counter chip. This function can only be used with 9513 counters. For more information, refer to the AM9513A data sheet in the 9513A.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default).

This data sheet is also available on our web site at www.mccdaq.com/PDFmanuals/9513A.pdf.

Function prototype:

C/C++:

```
int cbC9513Init( int BoardNum, int ChipNum, int FOutDivider, int
FOutSource, int Compare1, int Compare2, int TimeOfDay )
```

Visual Basic:

```
Function cbC9513Init( ByVal BoardNum&, ByVal ChipNum&, ByVal
FOutDivider&, ByVal FOutSource&, ByVal Compare1&, ByVal Compare2&,
ByVal TimeOfDay& ) As Long
```

Delphi:

```
function cbC9513Init( BoardNum:Integer; ChipNum:Integer;
FOutDivider:Integer; FOutSource:Integer; Compare1:Integer;
Compare2:Integer; TimeOfDay:Integer ):Integer;
```

Arguments:

`BoardNum`
Refers to the board number associated with the board when it was installed with the configuration program. The specified board must have a 9513 counter. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`ChipNum`
Specifies which 9513 chip is to be initialized. For a CTR05 board this should be set to 1. For a CTR10 board it should be either 1 or 2, and for a CTR20 it should be 1-4.

`FOutDivider`
F-Out divider (0-15). If set to 0, `FoutDivider` is the rate of `FoutSource` divided by 16. If set to a number between 1 ands 15, `FoutDivider` is the rate of `FoutSource` divided by `FoutDivider`.

`FOutSource`
Specifies source of the signal for F-Out signal. Set it to one of the constants in the "`FOutSource` argument values" section on page 62.

Compare1
Compare1 ENABLED or Compare1 DISABLED (CBENABLED or CBDISABLED in Visual Basic or Delphi).

Compare2
Compare2 ENABLED or Compare2 DISABLED. (CBENABLED or CBDISABLED in Visual Basic or Delphi).

`TimeOfDay`
TimeOfDay ENABLED or TimeOfDay DISABLED. (CBENABLED or CBDISABLED in Visual Basic or Delphi). The options for this argument are listed in the "`TimeOfDay` argument values" section on page 62.

Returns:

[Error code]() or 0 if no errors

`FOutSource` argument values:

FOutSource	9513 Data Sheet Equivalent
CTRINPUT1	SRC 1 (Counter Input 1)
CTRINPUT2	SRC 2 (Counter Input 2)
CTRINPUT3	SRC 3 (Counter Input 3)
CTRINPUT4	SRC 4 (Counter Input 4)
CTRINPUT5	SRC 5 (Counter Input 5)
GATE1	GATE1
GATE2	GATE2
GATE3	GATE3
GATE4	GATE4
GATE5	GATE5
FREQ1	F1
FREQ2	F2
FREQ3	F3
FREQ4	F4
FREQ5	F5

`TimeOfDay` argument values:

TimeOfDay	9513 Data Sheet Equivalent
CBDISABLED	TOD Disabled
1	TOD Enabled / 5 Input
2	TOD Enabled / 6 Input
3	TOD Enabled / 10 Input
No arguments for:	9513 data sheet equivalent
0 (FOUT on)	FOUT Gate
0 (Data bus matches board)	Data Bus Width
1 (Disable Increment)	Data Pointer Control
1 (BCD Scaling)	Scalar Control

Notes:

The information provided here and in `cbC9513Config()` will help you understand how the Universal Library syntax corresponds to the 9513 data sheet, but is not a substitute for the data sheet. You cannot program and use a 9513 without this data sheet.

Refer to the accompanying 9513A.pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default). The data sheet is also available on our web site at www.mccdaq.com/PDFmanuals/9513A.pdf

cbCFreqIn()

Measures the frequency of a signal. This function is only used with 9513 counters. This function uses internal counters #4 and #5.

Function prototype:

C/C++:	`int cbCFreqIn( int BoardNum, int SigSource, int GateInterval, unsigned short *Count, long *Freq )`
Visual Basic:	`Function cbCFreqIn(ByVal BoardNum&, ByVal SigSource&, ByVal GateInterval&, Count%, Freq&) As Long`
Delphi:	`function cbCFreqIn( BoardNum:Integer; SigSource:Integer; GateInterval:Integer; var Count:Word; var Freq:Longint ):Integer;`

Arguments:

`BoardNum` — The board number associated with the board when it was installed with the configuration program. The specified board must have a 9513 counter. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`SigSource` — Specifies the source of the signal from which the frequency is calculated. The signal to be measured is routed internally from the source specified by `SigSource` to the clock input of counter 5. On boards with more than one 9513 chip, there is more than one counter 5. Which counter 5 is used is also determined by `SigSource`. Set it to one of the constants in the "SigSource argument values" section on page 64.

The value of `SigSource` determines which chip will be used. `CTRINPUT6` through `CTRINPUT10`, `FREQ6` through `FREQ10` and `GATE6` through `GATE9` indicate chip two will be used. The signal to be measured must be present at the chip two input specified by `SigSource`. Also, the gating connection from counter 4 output to counter 5 gate must be made between counters 4 and 5 of this chip (see below). Refer to board-specific information to determine valid values for your board.

`GateInterval` — Gating interval in milliseconds (must be > 0). Specifies the time (in milliseconds) that the counter will be counting. The optimum `GateInterval` depends on the frequency of the measured signal. The counter can count up to 65535. If the gating interval is too low, the count will be too low and the resolution of the frequency measurement will be poor. For example, if the count changes from 1 to 2, the measured frequency doubles. If the gating interval is too long, then the counter overflows and a `FREQOVERFLOW` error occurs.

The `cbCFreqIn` function does not return until the `GateInterval` has expired. There is no background option. Under Windows, this means that window activity will stop for the duration of the call. Adjust the `GateInterval` so this does not pose a problem to your user interface.

`Count` — The raw count is returned here.

`Freq` — The measured frequency in Hz is returned here.

Returns:

Error code or 0 if no errors.

`Count` - Count that frequency calculation based on returned here.

`Freq` - Measured frequency in Hz returned here.

SigSource argument values:

One 9513 chip (Chip 1 used):

- `CTRINPUT1` through `CTRINPUT5`
- `GATE1` through `GATE4`
- `FREQ1` through `FREQ5`

Two 9513 chips (Chip 1 or Chip 2 used):

- `CTRINPUT1` through `CTRINPUT10`
- `GATE1` through `GATE9` (excluding gate 5)
- `FREQ1` through `FREQ10`

Four 9513 chips (Chips 1- 4 may be used):

- `CTRINPUT1` through `CTRINPUT20`
- `GATE1` through `GATE19` (excluding gates 5, 10 & 15)
- `FREQ1` through `FREQ20`

Notes:

- This function requires an electrical connection between counter 4 output and counter 5 gate. This connection must be made between counters 4 and 5 *on the chip determined by* `SigSource`.
- `cbC9513Init()` must be called for each `ChipNum` that will be used by this function. The values of `FOutDivider`, `FOutSource`, `Compare1`, `Compare2`, and `TimeOfDay` are irrelevant to this function and may be any value shown in the `cbC9513Init()` function description.
- If you select an external clock source for the counters, the `GateInterval`, `Count`, and `Freq` settings are only valid if the external source is 1 MHz. Otherwise, you need to scale the values according to the frequency of the external clock source. For example, for an external clock source of 2 MHz, increase your `GateInterval` setting by a factor of 2, and also double the `Count` and `Freq` values returned when analyzing your results.

cbCIn()

Reads the current count from a counter.

Function prototype:

C/C++:	`int cbCIn( int BoardNum, int CounterNum, unsigned short *Count )`
Visual Basic:	`Function cbCIn( ByVal BoardNum&, ByVal CounterNum&, Count% ) As Long`
Delphi:	`function cbCIn( BoardNum:Integer; CounterNum:Integer; var Count:Word ):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. The specified board must have a counter. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`CounterNum`	The counter to read the current count from. Valid values are 1 to 20, up to the number of counters on the board.
`Count`	Counter value returned here. See the "Notes" section below.

Returns:

Error code or 0 if no errors.

Notes:

`Count` - The range of counter values returned are: 0 to 65,535 for C or PASCAL languages. Refer to your BASIC manual for information on BASIC integer data types. -32,768 to 32,767 for BASIC languages. BASIC reads counters as:

- -1 reads as 65535
- -21768 reads as 32768
- 32767 reads as 32767
- 2 reads as 2
- 0 reads as 0

cbCIn() vs. cbCIn32(): Although the `cbCIn()` and `cbCIn32()` functions perform the same operation, `cbCIn32()` is the preferred function to use.

The only difference between the two is that `cbCIn()` returns a 16-bit count value and `cbCIn32()` returns a 32-bit value. Both `cbCIn()` and `cbCIn32()` can be used, but `cbCIn32()` is required whenever you need to read count values greater than 16 bits (counts > 65535).

cbCIn32() (32-bit UL Only)

Reads the current count from a counter and returns it as a 32-bit integer.

Function prototype:

C/C++: `int cbCIn32( int BoardNum, int CounterNum, unsigned long *Count )`

Visual Basic: `Function cbCIn32( ByVal BoardNum&, ByVal CounterNum&, Count& ) As Long`

Delphi: `function cbCIn32 (BoardNum:Integer; CounterNum:Integer; var Count:Longint):Integer;`

Arguments:

`BoardNum` — The board number associated with the board when it was installed with the configuration program. The specified board must have an LS7266 counter. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`CounterNum` — The counter to read current count from. Valid values are 1 to N, where N is the number of counters on the board.

`Count` — Current count value from selected counter is returned here.

Returns:

Error code or 0 if no error occurs.

Notes:

cbCIn() vs. cbCIn32(): Although the `cbCIn()` and `cbCIn32()` functions perform the same operation, `cbCIn32()` is the preferred function to use.

The only difference between the two is that `cbCIn()` returns a 16-bit count value and `cbCIn32()` returns a 32-bit value. Both `cbCIn()` and `cbCIn32()` can be used, but `cbCIn32()` is required whenever you need to read count values greater than 16 bits (counts > 65535).

cbCLoad()

Loads the specified counter's `LOAD`, `HOLD`, `ALARM`, `COUNT`, `PRESET` or `PRESCALER` register with a count. When loading a counter with a starting value, it is never loaded directly into the counter's count register. Rather, it is loaded into the load or hold register. From there, the counter, after being enabled, loads the count from the appropriate register, generally on the first valid pulse.

Function prototype:

C/C++:	`int cbCLoad( int BoardNum, int RegNum, unsigned LoadValue )`
Visual Basic:	`Function cbCLoad( ByVal BoardNum&, ByVal RegNum&, ByVal LoadValue& ) As Long`
Delphi:	`function cbCLoad( BoardNum:Integer; RegNum:Integer; LoadValue:Word ):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. The specified board must have a counter. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`RegNum`	The register to load the count to. Set it to one of the constants in the "`RegNum` argument values" section below.
`LoadValue`	The value to be loaded. Must be between 0 and $2^{resolution}$ - 1 of the counter. For example, a 16-bit counter is 2^{16} - 1, or 65,535. Refer to the discussion of Basic signed integers in the "16-bit values using a signed integer data type" section in the "Universal Library Description & Use" chapter of the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Returns:

Error code or 0 if no errors.

`RegNum` argument values:

`LOADREG1 .. 20`	Load registers 1 through 20. This may span several chips.
`HOLDREG1 .. 20`	Hold registers 1 through 20. This may span several chips. (9513 only)
`ALARM1CHIP1`	Alarm register 1 of the first counter chip. (9513 only)
`ALARM2CHIP1`	Alarm register 2 of the first counter chip. (9513 only)
`ALARM1CHIP2`	Alarm register 1 of the second counter chip. (9513 only)
`ALARM2CHIP2`	Alarm register 2 of the second counter chip. (9513 only)
`ALARM1CHIP3`	Alarm register 1 of the third counter chip. (9513 only)
`ALARM2CHIP3`	Alarm register 2 of the third counter chip. (9513 only)
`ALARM1CHIP4`	Alarm register 1 of the four counter chip. (9513 only)
`ALARM2CHIP4`	Alarm register 2 of the four counter chip. (9513 only)
`COUNT1` .. 4	Current Count (LS7266 only)
`PRESET1` .. 4	Preset register (LS7266 only)
`PRESCALER1` .. 4	Prescaler register (LS7266 only)

Notes:

You cannot load a count-down-only counter with less than 2.

Counter types: There are several counter types supported. Please refer to the counter chip's data sheet for the registers that are available.

cbCLoad() vs. cbCLoad32(): Although the `cbCLoad()` and `cbCLoad32()` functions perform the same operation, `cbCLoad32()` is the preferred function to use.

The only difference between the two is that `cbCLoad()` loads a 16-bit count value, and `cbCLoad32()` loads a 32-bit value. The only time you need to use `cbCLoad32()` is to load counts that are larger than 32-bits (counts > 65535).

cbCLoad32() (32-bit UL Only)

Loads the specified counter's `COUNT`, `PRESET`, or `PRESCALER` register with a count.

Function prototype:

C/C++:	`int cbCLoad32( int BoardNum, int RegNum, unsigned long LoadValue )`
Visual Basic:	`Function cbCLoad32( ByVal BoardNum&, ByVal RegNum&, ByVal LoadValue& ) As Long`
Delphi:	`function cbCLoad32 ( BoardNum:Integer; RegNum:Integer; LoadValue:Longint ):Integer;`

Arguments:

`BoardNum`	Refers to the board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`RegNum`	The register to load the value into. Set it to one of the constants in the "`RegNum` argument values" section below.

Returns:

Error code or 0 if no error occurs.

`RegNum` argument values:

`LOADREG1 .. 20`	Load registers 1 through 20. This may span several chips.
`HOLDREG1 .. 20`	Hold registers 1 through 20. This may span several chips. (9513 only)
`ALARM1CHIP1`	Alarm register 1 of the first counter chip. (9513 only)
`ALARM2CHIP1`	Alarm register 2 of the first counter chip. (9513 only)
`ALARM1CHIP2`	Alarm register 1 of the second counter chip. (9513 only)
`ALARM2CHIP2`	Alarm register 2 of the second counter chip. (9513 only)
`ALARM1CHIP3`	Alarm register 1 of the third counter chip. (9513 only)
`ALARM2CHIP3`	Alarm register 2 of the third counter chip. (9513 only)
`ALARM1CHIP4`	Alarm register 1 of the four counter chip. (9513 only)
`ALARM2CHIP4`	Alarm register 2 of the four counter chip. (9513 only)
`COUNT1` .. 4	Current Count (LS7266 only)
`PRESET1` .. 4	Preset register (LS7266 only)
`PRESCALER1` .. 4	Prescaler register (LS7266 only)

Notes:

cbCLoad() vs. cbCLoad32(): Although the `cbCLoad()` and `cbCLoad32()` functions perform the same operation, `cbCLoad32()` is the preferred function to use.

The only difference between the two is that `cbCLoad()` loads a 16-bit count value, and `cbCLoad32()` loads a 32-bit value. The only time you need to use `cbCLoad32()` is to load counts that are larger than 32-bits (counts > 65535).

cbCStatus() (32-bit UL Only)

Returns status information about the specified counter (7266 counters only). For more information, see the LS7261 data sheet in the LS7266R1pdf file located in the *Documents* subdirectory where you installed UL (C:\MCC by default). This data sheet is also available on our web site at www.mccdaq.com/PDFmanuals/LS7266R1.pdf.

Function prototype:

C/C++:	`int cbCStatus (int BoardNum, int CounterNum, unsigned long *StatusBits)`
Visual Basic:	`Function cbCStatus(ByVal BoardNum&, ByVal CounterNum&, StatusBits&) As Long`
Delphi:	`function cbCStatus (BoardNum:Integer; CounterNum:Integer; var StatusBits:Longint):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. The specified board must have an LS7266 counter. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`CounterNum`	The counter to read current count from. Valid values are 1 to N, where N is the number of counters on the board.
`StatusBits`	Current status from selected counter is returned here. The status consists of individual bits that indicate various conditions within the counter. Set it to one of the constants in the "StatusBits argument values" section below.

Returns:

Error code or 0 if no error occurs.

StatusBits argument values:

`C_UNDERFLOW`	Set to 1 whenever the count decrements past 0. Is cleared to 0 whenever `cbCStatus()` is called.
`C_OVERFLOW`	Set to 1 whenever the count increments past it's upper limit. Is cleared to 0 whenever `cbCStatus()` is called.
`C_COMPARE`	Set to 1 whenever the count matches the preset register. Is cleared to 0 whenever `cbCStatus()` is called.
`C_SIGN`	Set to 1 when the MSB of the count is 1. Is cleared to 0 whenever the MSB of the count is set to 0.
`C_ERROR`	Set to 1 whenever an error occurs due to excessive noise on the input. Is cleared to 0 by calling `cbC7266Config()` set to 1 when index is valid. Is cleared to 0 when index is not valid.
`C_UP_DOWN`	Set to 1 when counting up. Is cleared to 0 when counting down
`C_INDEX`	Set to 1 when index is valid. Is cleared to 0 when index is not valid.

cbCStoreOnInt()

Changed R4.0 RW

Installs an interrupt handler that will store the current count whenever an interrupt occurs. This function can only be used with 9513 counters. This function will continue to operate in the background until either `IntCount` has been satisfied or `cbStopBackground()` is called.

Function prototype:

C/C++:	`int cbCStoreOnInt (int BoardNum, int IntCount, int CntrControl[], int MemHandle)`
Visual Basic:	`Function cbCStoreOnInt ( ByVal BoardNum&, ByVal IntCount&, CntrControl%, ByVal MemHandle& ) As Long`
Delphi:	`function cbCStoreOnInt ( BoardNum:Integer; IntCount:Integer; var CntrControl:SmallInt; MemHandle:Integer ):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. The specified board must have a 9513 counter. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`IntCount`	The counters will be read every time an interrupt occurs until `IntCount` number of interrupts have occurred. If `IntCount` is = 0 then the function will run until `cbStopBackground()` is called. (refer to `MemHandle`).
`CntrControl`	The array should have an element for each counter on the board. (5 elements for CTR-05 board, 10 elements for a CTR-10, etc.). Each element corresponds to a counter channel. Each element should be set to either `CBDISABLED` or `CBENABLED`. All channels that are set to `CBENABLED` will be read when an interrupt occurs.
`MemHandle`	Handle for Windows buffer. Counts are stored in an array. The array should have an element for each counter on the board. (5 elements for CTR-05 board, 10 elements for a CTR-10, etc.). Each element corresponds to a counter channel. Each channel that is marked as `CBENABLED` in the `CntrControl` array will be read when an interrupt occurs. The count value will be stored in the `DataBuffer` element associated with that channel.

Returns:

Error code or 0 if no errors.

Notes:

New functionality: If the Library Revision is set to 4.0 or greater, the following code changes are required:

- If `IntCount` is non-zero, then the `CountData` array must be allocated to (`IntCount` * Number of Counters).

  For example, if you set `IntCount` to 100 for a CTR-05 board, then you must declare the `CountData` array with a size of (100 * 5) = 500. This new functionality keeps the user application from having to move the data out of the `CountData` buffer for every interrupt, before it is overwritten. Now, for each interrupt the counter values will be stored in adjacent memory locations in the `CountData` array.

Allocate the proper array size for non-zero `IntCount` settings

Specifying `IntCount` as a non-zero value and failing to allocate the proper sized array results in a runtime error. There is no way for the Universal Library to determine if the array has been allocated with the proper size.

- If `IntCount` = 0, the functionality is unchanged.

Digital I/O Functions

Introduction

Use the functions explained in this chapter to read and set digital values. Most digital ports are configurable, while some others are non-configurable. Some types of hardware allow readback of the values that output ports are set to on configurable port types. Devices using 8255 chips for digital I/O are one example. For these devices, input functions such as `cbDIn()` are valid for ports configured as output.

Use the tables below to determine the port number, bit number, and actual addresses being set by the digital I/O functions. Table 5-1 relates the port number (`PortNum`) to the port address and the 8255 port. Table 5-2 relates the bit number to the 8255 chip on the board.

Table 5-1. Port Numbers and Corresponding Port Address, 8255 Port Number

Mnemonic	**Bit No.**	**8255 Port No.**	**Port Address**	**8536 Port No.**	**Port Address**
FIRSTPORTA	0 - 7	1A	Base + 0	1A	Base + 0
FIRSTPORTB	8 - 15	1B		1B	
FIRSTPORTCL	16 - 19	1CL		1C	
FIRSTPORTCH	20 - 23	1CH		Not present	
SECONDPORTA	24 - 31	2A	Base + 4	2A	Base + 4
SECONDPORTB	32 - 39	2B		2B	
SECONDPORTCL	40 - 43	2CL		2C	
SECONDPORTCH	44 - 47	2CH		Not present	
and so on, to the last chip on the board as: THIRDPORT*x*, FOURTHPORT*x*, FIFTHPORT*x*, SIXTHPORT*x*, and SEVENTHPORT*x*					
EIGHTHPORTA	168 -175	8A	Base + 28		
EIGHTHPORTB	176 -183	8B			
EIGHTHPORTCL	184 -187	8CL			
EIGHTHPORTCH	188 -191	8CH			

Table 5-2 Bit Numbers and Corresponding 8255 Chip Number

82C55 Bit#	**Chip #**	**Address**	**8536 Bit#**	**Chip #**	**Address**
0 – 23	1	Base + 0	0 - 19	1	Base + 0
24 – 47	2	Base + 4	20 – 39	2	Base + 4
48 – 71	3	Base + 8			
72 – 95	4	Base + 12			
96 – 119	5	Base + 16			
120 – 143	6	Base + 20			
144 – 167	7	Base + 24			
168 – 191	8	Base + 28			

cbDBitIn()

Reads the state of a single digital input bit.

This function treats all of the DIO ports of a particular type on a board as a single port. It lets you read the state of any individual bit within this port.

Note that for some port types—such as 8255 ports—if the port is configured for `DIGITALOUT`, this function provides readback of the last output value.

Function prototype:

C/C++:
```
int cbDBitIn(int BoardNum, int PortType, int BitNum, unsigned short
*BitValue)
```

Visual Basic:
```
Function cbDBitIn Lib(ByVal BoardNum&, ByVal PortType&, ByVal
BitNum&, BitValue%) As Long
```

Delphi:
```
function cbDBitIn(BoardNum:Integer; PortType:Integer;
BitNum:Integer; var BitValue:Word):Integer;
```

Arguments:

`BoardNum` The number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit UL).

`PortType` There are three general types of digital ports—ports that are programmable as input or output, ports that are fixed input or output, and ports for which each bit may be programmed as input or output. For the first of these types, set `PortType` to `FIRSTPORTA`. For the latter two types, set `PortType` to `AUXPORT`. Some boards have both types of digital ports (DAS1600). Set `PortType` to either `FIRSTPORTA` or `AUXPORT`, depending on which digital inputs you wish to read.

`BitNum` This specifies the bit number within the single large port. Table 5-2 on page 73 shows which bit numbers are in which 82C55 and 8536 digital chips. The most 82C55 chips on a single board is eight (8), on the CIO-DIO196. The most (2) 8536 chips occur on the CIO-INT32.

`BitValue` Place holder for return value of bit. Value will be 0 or 1. A 0 indicates a logic low reading, a 1 indicates a logic high reading. Logic high does not necessarily mean 5V. See the board manual for chip input specifications.

Returns:

Error code or 0 if no errors.

`BitValue` - value (0 or 1) of specified bit returned here.

cbDBitOut()

Sets the state of a single digital output bit. This function treats all of the DIO ports of a particular type on a board as a single very large port. It lets you set the state of any individual bit within this large port. If the port type is not `AUXPORT`, you **must** use `cbDConfigPort()` to configure the port for output first. If the port type is `AUXPORT`, you **may** need to use `cbDConfigBit()` or `cbDConfigPort()` to configure the bit for output first. Refer to the board-specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) to determine if `AUXPORT` should be configured for your hardware.

Function prototype:

C/C++:	`int cbDBitOut (int BoardNum, int PortType, int BitNum, unsigned short BitValue)`
Visual Basic:	`Function cbDBitOut(ByVal BoardNum&, ByVal PortType&, ByVal BitNum&, ByVal BitValue%) As Long`
Delphi:	`function cbDBitOut (BoardNum:Integer; PortType:Integer; BitNum:Integer; BitValue:Word):Integer;`

Arguments:

`BoardNum`	The number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`PortType`	There are three general types of digital ports - ports that are programmable as input or output, ports that are fixed input or output and ports for which each bit may be programmed as input or output. For the first of these types, set `PortType` to `FIRSTPORTA`. For the latter two types, set `PortType` to `AUXPORT`. Some boards have both types of digital ports (DAS1600). Set `PortType` to either `FIRSTPORTA` or `AUXPORT` depending on which digital port you wish to write to.
`BitNum`	This specifies the bit number within the single large port. The specified bit must be in a port that is currently configured as an output. Table 5-2 on page 73 shows which bit numbers are in which 82C55 and 8536 digital chips. The most 82C55 chips on a single board is eight (8), on the CIO-DIO196. The most (2) 8536 chips occur on the CIO-INT32.
`BitValue`	The value to set the bit to. Value will be 0 or 1. A 0 indicates a logic low output, a 1 indicates a logic high output. Logic high does not necessarily mean 5V. See the board manual for chip specifications.

Returns:

Error code or 0 if no errors.

cbDConfigBit()

Configures a specific digital bit as Input or Output. This function treats all DIO ports of the AUXPORT type on a board as a single port. This function is NOT supported by 8255 type DIO ports. Refer to the board-specific information for details.

Function prototype:

C/C++:

```
int cbDConfigBit (int BoardNum, int PortType, int BitNum, int Direction)
```

Visual Basic:

```
Function cbDConfigBit (ByVal BoardNum&, ByVal PortType&, ByVal BitNum&, ByVal Direction&) As Long
```

Delphi:

```
function cbDConfigBit (Boardnum:Integer; PortType:Integer; BitNum:Integer; Direction:Integer) :Integer;
```

Arguments:

`BoardNum`	The number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`PortType`	The port (`AUXPORT`) whose bits are to be configured. The port specified must be bitwise configurable. See board specific information for details.
`BitNum`	The bit number to configure as input or output. See board specific information for details.
`Direction`	`DIGITALOUT` or `DIGITALIN` configures the specified bit for output or input, respectively.

Returns:

Error code or 0 if no errors.

cbDConfigPort()

Configures a digital port as input or output. This function is for use with ports that may be programmed as input or output, such as those on the 82C55 chips and 8536 chips. Refer to the Zilog 8536 manual for details of chip operation. Also refer to the 82C55 data sheet, which is available on our web site at www.mccdaq.com/PDFmanuals/82C55A.pdf.

Function prototype:

C/C++:	`int cbDConfigPort(int BoardNum, int PortNum, int Direction)`
Visual Basic:	`Function cbDConfigPort(ByVal BoardNum&, ByVal PortNum&, ByVal Direction&) As Long`
Delphi:	`function cbDConfigPort (Boardnum:Integer; PortNum:Integer; Direction:Integer) :Integer;`

Arguments:

`BoardNum`	The number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`PortNum`	The specified port must be configurable. For most boards, `AUXPORT` is not configurable; please consult board specific documentation. Table 5-1 on page 73 shows which ports and bit numbers are associated with which 82C55 and 8536 digital chips. The most 82C55 chips on a single board is eight (8), on the CIO-DIO196. The most (2) 8536 chips occur on the CIO-INT32.
`Direction`	`DIGITALOUT` or `DIGITALIN` configures entire eight or four bit port for output or input.

Returns:

Error code or 0 if no errors.

Notes:

When used on ports within an 8255 chip, this function will reset all ports on that chip configured for output to a zero state. This means that if you set an output value on `FIRSTPORTA` and then change the configuration on `FIRSTPORTB` from `OUTPUT` to `INPUT`, the output value at `FIRSTPORTA` will be all zeros. You can, however, set the configuration on `SECONDPORTX` without affecting the value at `FIRSTPORTA`. For this reason, this function is usually called at the beginning of the program for each port requiring configuration.

cbDIn()

Reads a digital input port. Note that for some port types, such as 8255 ports, if the port is configured for `DIGITALOUT`, this function will provide readback of the last output value.

Function prototype:

C/C++:	`int cbDIn (int BoardNum, int PortNum, unsigned short *DataValue)`
Visual Basic:	`Function cbDIn(ByVal BoardNum&, ByVal PortNum&, DataValue%) As Long`
Delphi:	`function cbDIn (BoardNum:Integer; PortNum:Integer; var DataValue:Word):Integer; StdCall;`

Arguments:

`BoardNum`	The number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`PortNum`	Specifies which digital I/O port to read. Some hardware does allow readback of the state of the output using this function. Check the board specific information in the *Universal Library User's Guide*. Table 5-1 on page 73 shows which ports are in which 82C55 and 8536 digital chips. The most 82C55 chips on a single board is eight, on a CIO-DIO192. The most 8536 chips on a board is two, on the CIO-INT32.
`DataValue`	Digital input value returned here.

Returns:

Error code or 0 if no errors.

`DataValue` - Digital input value returned here.

Notes:

The size of the ports vary. If it is an eight bit port then the returned value will be in the range 0 - 255. If it is a four bit port the value will be in the range 0 - 15.

Refer to the example programs and the board-specific information contained in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) for clarification of valid `PortNum` values.

cbDInScan()

Multiple reads of digital input port of a high speed digital port on a board with a pacer clock such as the CIO-PDMA16.

Function prototype:

C/C++:

```
int cbDInScan(int BoardNum, int PortNum, long Count, long *Rate, int MemHandle, int Options)
```

Visual Basic:

```
Function cbDInScan(ByVal BoardNum&, ByVal PortNum&, ByVal Count&, Rate&, ByVal MemHandle&, ByVal Options&) As Long
```

Delphi:

```
function cbDInScan(BoardNum:Integer; PortNum:Integer; Count:Longint; var Rate:Longint; MemHandle:Integer; Options:Integer):Integer;
```

Arguments:

`BoardNum` The number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`PortNum` Specifies which digital I/O port to read (usually, `FIRSTPORTA` or `FIRSTPORTB`). The specified port must be configured as an input.

`Count` The number of times to read digital input.

`Rate` Number of times per second (Hz) to read the port. The actual sampling rate in some cases will vary a small amount from the requested rate. The actual rate will be returned to the `Rate` argument.

`MemHandle` Handle for Windows buffer to store data in (Windows). This buffer must have been previously allocated with the `cbWinBufAlloc()` function.

`Options` Bit fields that control various options. Refer to the constants in the "Options argument values" section below.

Returns:

Error code or 0 if no errors.

`Rate` - actual sampling rate returned.

`MemHandle` - digital input value returned via allocated Windows buffer.

Options argument values:

`BACKGROUND` If the `BACKGROUND` option is not used then the `cbDInScan()` function will not return to your program until all of the requested data has been collected and returned to `DataBuffer`.

When the `BACKGROUND` option is used, control will return immediately to the next line in your program and the transfer from the digital input port to `DataBuffer` will continue in the background. Use `cbGetStatus()` to check on the status of the background operation. Use `cbStopBackground()` to terminate the background process before it has completed.

`CONTINUOUS` This option puts the function in an endless loop. Once it transfers the required number of bytes it resets to the start of `DataBuffer` and begins again. The only way to stop this operation is with `cbStopBackground()`. Normally this option should be used in combination with `BACKGROUND` so that your program will regain control.

`EXTCLOCK`	If this option is used then transfers will be controlled by the signal on the trigger input line rather than by the internal pacer clock. Each transfer will be triggered on the appropriate edge of the trigger input signal (see board specific info). When this option is used the `Rate` argument is ignored. The transfer rate is dependent on the trigger signal.
`WORDXFER`	Normally this function reads a single (byte) port. If `WORDXFER` is specified then it will read two adjacent ports on each read and store the value of both ports together as the low and high byte of a single array element in `DataBuffer`[].

Notes:

Transfer method - May not be specified. DMA is used.

cbDOut()

Writes a byte to a digital output port. If the port type is not `AUXPORT`, you **must** use [cbDConfigPort()]() to configure the port for output first. If the port type is `AUXPORT`, you **may** need to use [cbDConfigPort()]() to configure the port for output first. Check the board specific information in the *Universal Library User's Guide* (available on our web site at [www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf]()) to determine if `AUXPORT` should be configured for your hardware.

Function prototype:

C/C++:
```
int cbDOut (int BoardNum, int PortNum, unsigned short DataValue)
```

Visual Basic:
```
Function cbDOut(ByVal BoardNum&, ByVal PortNum&, ByVal DataValue%)
As Long
```

Delphi:
```
function cbDOut (BoardNum:Integer; PortNum:Integer;
DataValue:Word):Integer;
```

Arguments:

`BoardNum` The number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`PortNum` There are three general types of digital ports - ports that are programmable as input or output, ports that are fixed input or output and ports for which each bit may be programmed as input or output. For the first of these types, set `PortNum` to `FIRSTPORTA`. For the latter two types, set `PortNum` to `AUXPORT`. Some boards have both types of digital ports (DAS1600). Set `PortNum` to either `FIRSTPORTA` or `AUXPORT` depending on the digital port you want to set.

Table 5-1 on page 73 shows which ports are in which 82C55 and 8536 digital chips. The CIO-DIO196 has eight 82C55 chips—the most on a single board. The CIO-INT32 has two 8536 —the most on a single board.

`DataValue` Digital input value to be written.

Returns:

[Error code]() or 0 if no errors.

Notes:

The size of the ports vary. If it is an eight bit port then the output value should be in the range 0 - 255. If it is a four bit port the value should be in the range 0 - 15. Be sure to look at the example programs and the board specific information in the *Universal Library User's Guide* for clarification of valid `PortNum`.

cbDOutScan()

Performs multiple writes to a digital output port of a high speed digital port on a board with a pacer clock, such as the CIO-PDMA16 or CIO-PMA32.

Function prototype:

C/C++:
```
int cbDOutScan(int BoardNum, int PortNum, long Count, long *Rate,
int MemHandle, int Options)
```

Visual Basic:
```
Function cbDOutScan(ByVal BoardNum&, ByVal PortNum&, ByVal Count&,
Rate&, ByVal MemHandle&, ByVal Options&) As Long
```

Delphi:
```
function cbDOutScan (BoardNum:Integer; PortNum:Integer;
Count:Longint; var Rate:Longint; MemHandle:Integer;
Options:Integer):Integer;
```

Arguments:

`BoardNum`	The number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`PortNum`	Specifies which digital I/O port to write. The two choices are `FIRSTPORTA` or `FIRSTPORTB`. The specified port must be configured as an output.
`Count`	The number of times to write digital output.
`Rate`	Number of times per second (Hz) to write to the port. The actual update rate in some cases will vary a small amount from the requested rate. The actual rate will be returned to the `Rate` argument.
`MemHandle`	Handle for Windows buffer to store data in (Windows). This buffer must have been previously allocated with the `cbWinBufAlloc()` function.
`Options`	Bit fields that control various options. Refer to the constants in the "Options argument values" section below.

Returns:

Error code or 0 if no errors.

`Rate` - actual sampling rate returned.

Options argument values:

`BACKGROUND`	If the `BACKGROUND` option is not used then the `cbDOutScan()` function will not return to your program until all of the requested data has been output. When the `BACKGROUND` option is used, control returns immediately to the next line in your program and the transfer to the digital output port from `DataBuffer` will continue in the background. Use `cbGetStatus()` to check on the status of the background operation. Use `cbStopBackground()` to terminate the background process before it has completed.
`CONTINUOUS`	This option puts the function in an endless loop. Once it transfers the required number of bytes it resets to the start of the buffer and begins again. The only way to stop this operation is with `cbStopBackground()`. Normally this option should be used in combination with `BACKGROUND` so that your program will regain control.

`EXTCLOCK`	If this option is used then transfers will be controlled by the signal on the trigger input line rather than by the internal pacer clock. Each transfer will be triggered on the appropriate edge of the trigger input signal (see board specific information). When this option is used the `Rate` argument is ignored. The transfer rate is dependent on the trigger signal.
`WORDXFER`	Normally this function writes a single (byte) port. If `WORDXFER` is specified then it will write two adjacent ports as the low and high byte of a single array element in the buffer.

Notes:

- `BYTEXFER` is the default `option`. Make sure you are using an array when your data is arranged in bytes. Use the `WORDXFER` option for word array transfers.
- **Transfer method** - May not be specified. DMA is used.

6

Error Handling Functions

Introduction

Use the functions explained in this chapter to get information from error codes returned by other UL functions. Most library functions return error codes. The different methods built in to the functions for handling errors include stopping the program when an error occurs, and printing error messages versus error codes.

cbErrHandling()

Sets the error handling for all subsequent function calls. Most functions return error codes after each call. In addition, other error handling features are built into the library. This function controls those features. If the Universal Library cannot find the configuration file CB.CFG, it always terminates the program, regardless of the `cbErrHandling()` setting.

Function prototype:

C/C++:	`int cbErrHandling( int ErrReporting, int ErrHandling )`
Visual Basic:	`Function cbErrHandling( ByVal ErrReporting&, ByVal ErrHandling& ) As Long`
Delphi:	`function cbErrHandling( ErrReporting:Integer; ErrHandling:Integer ):Integer;`

Arguments:

`ErrReporting`	This argument controls when the library will print error messages on the screen. The default is `DONTPRINT`. Set it to one of the constants in the "ErrReporting argument values" section below.
`ErrHandling`	This argument specifies what class of error will cause the program to halt. The default is `DONTSTOP` Set it to one of the constants in the "ErrHandling argument values" section below.

Returns:

Always returns 0.

ErrReporting argument values:

`DONTPRINT`	Errors will not generate a message to the screen. In that case your program must always check the returned error code after each library call to determine if an error occurred.
`PRINTWARNINGS`	Only warning errors will generate a message to the screen. Your program will have to check for fatal errors.
`PRINTFATAL`	Only fatal errors will generate a message to the screen. Your program must check for warning errors.
`PRINTALL`	All errors will generate a message to the screen.

ErrHandling argument values:

`DONTSTOP`	The program will always continue executing when an error occurs.
`STOPFATAL`	The program will halt if a "fatal" error occurs.
`STOPALL`	Will stop whenever any error occurs. If you are running in an Integrated Development Environment (IDE) then when errors occur, the environment may be shut down along with the program. If your IDE behaves this way, (QuickBasic and VisualBasic do), then set `ErrHandling` to `DONTSTOP`. Refer to "Error Codes" on page 301 for a complete list of error codes and their associated messages.

Notes:

Warnings vs. Fatal Errors: All errors that can occur are classified as either "warnings" or "fatal":

- Errors that can occur in normal operation in a bug free program (disk is full, too few samples before trigger occurred) are classified as "warnings".
- All other errors indicate a more serious problem and are classified as "fatal".

`STOPALL` not intended for 32-bit C console programs: Do not use the `STOPALL` option in 32-bit C console applications. Instead, use other methods to end the program, such as checking the return value of the function.

cbGetErrMsg()

Returns the error message associated with an error code. Each function returns an error code. An error code that is not equal to 0 indicates that an error occurred. Call this function to convert the returned error code to a descriptive error message.

Function prototype:

C/C++:	`int cbGetErrMsg( int ErrCode, char ErrMsg[ERRSTRLEN] )`
Visual Basic:	`Function cbGetErrMsg(ByVal ErrCode&, ByVal ErrMsg$) As Long`
Delphi:	`function cbGetErrMsg (ErrCode:Integer; ErrMsg:PChar):Integer;`

Arguments:

`ErrCode`	Error code that is returned by any function in library.
`ErrMsg`	Error message returned here. The `ErrMsg` variable must be pre-allocated to be at least as large as `ERRSTRLEN`. This size is guaranteed to be large enough to hold the longest error message.

Returns:

Error code or 0 if no errors.

*`ErrMsg` - error message string is returned here.

Notes:

See also `cbErrHandling()` on page 86 for an alternate method of handling errors.

7

Memory Board Functions

Introduction

Use the functions explained in this chapter to read and write data to and from a memory board, and also set modes that control memory boards (MEGA-FIFO).

The most common use for the memory boards is to store large amounts of data from an A/D board via a DT-Connect cable to a memory board. To do this, use the EXTMEMORY option with cbAInScan() or cbAPretrig(). Once the data has been transferred to the memory board, use the memory functions to retrieve it.

cbMemRead()

Reads data from a memory board into an array.

Function prototype:

C/C++:	`int cbMemRead(int BoardNum, unsigned short DataBuffer[], long FirstPoint, long Count)`
Visual Basic:	`Function cbMemRead(ByVal BoardNum&, DataBuffer%, ByVal FirstPoint&, ByVal Count&) As Long`
Delphi:	`function cbMemRead(BoardNum:Integer; var DataBuffer:Word; FirstPoint:Longint; Count:Longint):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`DataBuffer`	Pointer to the data array
`FirstPoint`	Index of first point to read, or `FROMHERE`. Use `FirstPoint` to specify the first point to read. For example, to read data sample numbers 200 through 250, set `FirstPoint` = 200 and `Count` = 50.
`Count`	Number of data points (words) to read

Returns:

Error code or 0 if no errors.

`DataBuffer` - data read from memory board.

Notes:

When reading a large amount of data from the board in small chunks, set `FirstPoint` to `FROMHERE` to read each successive chunk. Using `FROMHERE` speeds up a `cbMemRead()` operation when working with large amounts of data.

For example, to read 300,000 points in 100,000 point chunks, the calls would look like this:

```
cbMemRead (0, DataBuffer, 0, 100000)
cbMemRead (0, DataBuffer, FROMHERE, 1000000)
cbMemRead (0, DataBuffer, FROMHERE, 1000000)
```

DT-Connect Conflicts - The `cbMemRead()` function can not be called while a DT-Connect transfer is in progress. For example, if you start collecting A/D data to the memory board in the background (by calling `cbAInScan()` with the `DTCONNECT` + `BACKGROUND` options) you can not call `cbMemRead()` until the `cbAInScan()` has completed. If you do you will get a `DTACTIVE` error.

cbMemReadPretrig()

Reads pre-trigger data collected with the `cbAPretrig()` function from a memory board, and re-arranges the data in the correct order (pre-trigger data first, then post-trigger data). This function can only be used to retrieve data that was collected with the `cbAPretrig()` function with `EXTMEMORY` set in the options argument. After each `cbAPretrig()` call, all data must be unloaded from the memory board with this function. If any more data is sent to the memory board then the pre-trigger data will be lost.

Function prototype:

C/C++:	`int cbMemReadPretrig(int BoardNum, unsigned short DataBuffer[], long FirstPoint, long Count)`
Visual Basic:	`Function cbMemReadPretrig(ByVal BoardNum&, DataBuffer%, ByVal FirstPoint&, ByVal Count&) As Long`
Delphi:	`function cbMemReadPretrig(BoardNum:Integer; var DataBuffer:Word; FirstPoint:Longint; Count:Longint):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`DataBuffer`	The pointer to the data array.
`FirstPoint`	Index of first point to read, or `FROMHERE`. Use `FirstPoint` to specify the first point to read. For example, to read data sample numbers 200 through 250, set `FirstPoint` = 200 and `Count` = 50.
`Count`	Number of data samples (words) to read

Returns:

Error code or 0 if no errors.

`DataBuffer` - data read from memory board.

Notes:

When reading a large amount of data from the board in small chunks, set `FirstPoint` to `FROMHERE` to read each successive chunk. Using `FROMHERE` speeds up a `cbMemRead()` operation when working with large amounts of data. For example, to read 300,000 points in 100,000 chunks the calls would look like this:

```
cbMemReadPretrig (0, DataBuffer, 0, 100000)
cbMemReadPretrig (0, DataBuffer, FROMHERE, 1000000)
cbMemReadPretrig (0, DataBuffer, FROMHERE, 1000000)
```

DT-Connect Conflicts - The `cbMemReadPretrig()` function can not be called while a DT-Connect transfer is in progress. For example, if you start collecting A/D data to the memory board in the background (by calling `cbAInScan()` with the `DTCONNECT + BACKGROUND` options), you can not call `cbMemReadPretrig()` until the `cbAInScan()` has completed. If you do you will get a `DTACTIVE` error.

cbMemReset()

Resets the memory board pointer to the start of the data. The memory boards are sequential devices. They contain a counter which points to the 'current' word in memory. Every time a word is read or written this counter increments to the next word.

Function prototype:

C/C++:	`int cbMemReset(int BoardNum)`
Visual Basic:	`Function cbMemReset(ByVal BoardNum&) As Long`
Delphi:	`function cbMemReset(BoardNum:Integer):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

Returns:

Error code or 0 if no errors.

Notes:

This function is used to reset the counter back to the start of the memory. Between successive calls to `cbAInScan()`, you should call this function so that the second `cbAInScan()` overwrites the data from the first call. Otherwise, the data from the first `cbAInScan()` will be followed by the data from the second `cbAInScan()` in the memory on the card.

Likewise, anytime you call `cbMemRead()` or `cbMemWrite()` it will leave the counter pointing to the next memory location after the data that you read or wrote. Call cbMemReset`()` to reset back to the start of the memory buffer before the next call to `cbAInScan()`.

cbMemSetDTMode()

Sets the DT-Connect Mode of a memory board.

Function prototype:

C/C++:	`int cbMemSetDTMode(int BoardNum, int Mode)`
Visual Basic:	`Function cbMemSetDTMode(ByVal BoardNum&, ByVal Mode&) As Long`
Delphi:	`function cbMemSetDTMode (BoardNum:Integer; Mode:Integer):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`Mode`	Must be set to either `DTIN` or `DTOUT`. Set the `Mode` on the memory board to `DTIN` to transfer data from an A/D board to the memory board. Set `Mode` = `DTOUT` to transfer data from a memory board to a D/A board.

Returns:

Error code or 0 if no errors.

Notes:

- This command only controls the direction of data transfer between the memory board and its parent board that is connected to it via a DT-Connect cable.
- If you are using the `EXTMEMORY` option, do not use `cbMemSetDTMode()`, as the memory board mode is already set with `EXTMEMORY`. Only use `cbMemSetDTMode()` when the parent board is not supported by the Universal Library.

cbMemWrite()

Writes data from an array to the memory card.

Function prototype:

C/C++:	`int cbMemWrite(int BoardNum, unsigned short DataBuffer[], long FirstPoint, long Count);`
Visual Basic:	`Function cbMemWrite(ByVal BoardNum&, DataBuffer%, ByVal FirstPoint&, ByVal Count&) As Long`
Delphi:	`function cbMemWrite(BoardNum:Integer; var DataBuffer:Word; FirstPoint:Longint; Count:Longint):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`DataBuffer`	Pointer to the data array.
`FirstPoint`	Index of first point to write, or `FROMHERE`. Use `FirstPoint` to specify the first point to write data to. For example, to write to location numbers 200 through 250, set `FirstPoint` = 200 and `Count` = 50.
`Count`	Number of data points (words) to write.

Returns:

Error code or 0 if no errors.

Notes:

To write a large amount of data to the board in small chunks, set `FirstPoint` to `FROMHERE` to write each successive chunk. For example, to write 300,000 points in 100,000 point chunks:

```
cbMemWrite (0, DataBuffer, 0, 100000)
cbMemWrite (0, DataBuffer, FROMHERE, 100000)
cbMemWrite (0, DataBuffer, FROMHERE, 100000)
```

DT-Connect Conflicts - The `cbMemWrite()` function cannot be called while a DT-Connect transfer is in progress. For example, if you start collecting A/D data to the memory board in the background (by calling `cbAInScan()` with the `DTCONNECT` + `BACKGROUND` options). You cannot call `cbMemWrite()` until the `cbAInScan()` is complete. Doing so will generate a `DTACTIVE` error.

Revision Control Functions

Introduction

Use the functions explained in this chapter to initialize the Universal Library DLL so that the functions are interpreted according to the format of the revision that you wrote and compiled your program in As new revisions of the library are released, bugs from previous revisions are fixed and occasionally new functions are added. It is Measurement Computing's goal to preserve the existing programs that you have written, and therefore to never change the order or number of arguments in a function. However, it is not always possible to achieve this goal.

cbDeclareRevision()

New R3.3 ID

Initializes the Universal Library with the revision number of the library used to write your program. Must be the first Universal Library function to be called by your program.

Function prototype:

C/C++:	`int cbDeclareRevision(float* RevNum);`
Visual Basic:	`Function cbDeclareRevision(RevNum!) As Long`
Delphi:	`Function cbDeclareRevision(var RevNum:single):Integer;`

Arguments:

`RevNum`	Revision number of the Universal Library to interpret function arguments. **Default setting**: Any program using the 32-bit library and not containing this line of code will be defaulted to revision 5.4 argument assignments.

Returns:

Error Code or 0 if no errors.

Notes:

Default: Any program using the 16-bit library that does not contain a call to this function will default to revision 3.2 argument assignments.

As new revisions of the library are released, bugs from previous revisions are fixed and occasionally new functions are added. It is Measurement Computing's goal to preserve existing programs you have written, and therefore to never change the order or number of arguments in a function. Sometimes this is not possible, as in the changes from revision 3.2 to 3.3. In revision 3.3, we added support for multiple background tasks, a feature that users have requested.

Allowing multiple background tasks required adding the argument `BoardNum` to several functions. Doing so would have meant that programs written for version 3.2 would not run with 3.3 if they called those functions. If not for the new `cbDeclareRevision()` function, the programs would have had to be rewritten in each line where the affected functions are used, and the program recompiled.

The revision control function initializes the DLL so that the functions are interpreted according to the format of the revision you wrote and used to compiled your program. This function is new in revision 3.3. To take advantage of it, the function must be added to your program and the program recompiled.

The function works by interpreting the UL function call from your program and filling in any arguments needed to run with the new revision. For example, the function `cbAConvertData()` which appears on the following pages had the argument `BoardNum` added in Revision 3.3.

The two revisions of the function look like this:

Rev 3.2

```
int cbAConvertData (long NumPoints, unsigned ADData[], int ChanTags[])
```

Rev 3.3

```
int cbAConvertData (int BoardNum, long NumPoints, unsigned ADData[], int ChanTags[])
```

If your program has declared you are running code written for revision 3.2, and you call this function, the argument `BoardNum` is ignored. If you want the benefits afforded by `BoardNum`, you must rewrite your program with the new argument and declare revision 3.3 (or higher) in `cbDeclareRevision()`.

If a revision less than 3.2 is declared, revision 3.2 is assumed.

cbGetRevision()

Gets the revision level of Universal Library DLL and the VXD.

Function prototype:

C/C++: `int cbGetRevision(float* DLLRevNum, float* VXDRevNum);`

Visual Basic: `Function cbGetRevision(DLLRevNum!, VXDRevNum!) As Long`

Delphi: `function cbGetRevision(var DLLRevNum:Single; var VXDRevNum: Single):Integer;`

Arguments:

`DLLRevNum` Place holder for the revision number of Library DLL.

`VXDRevNum` Place holder for the revision number of Library VXD.

Returns:

`DLLRevNum` - Revision number of the Library DLL

`VXDRevNum` - Revision number of the Library VXD

Error Code if revision levels of VXD and DLL are incompatible.

9

Streamer File Functions

Introduction

Use the streamer file functions explained in this chapter to create, fill, and read streamer files.

cbFileAInScan()

Scans a range of A/D channels and stores the samples in a disk file. `cbFileAInScan` reads the specified number of A/D samples at the specified sampling rate from the specified range of A/D channels from the specified board. If the A/D board has programmable gain, it sets the gain to the specified range. The collected data is returned to a file in binary format. Use `cbFileRead()` to load data from that file into an array. See board-specific information to determine if this function is supported on your board.

Function prototype:

C/C++:

```
int cbFileAInScan(int BoardNum, int LowChan, int HighChan, long
Count, long *Rate, int Range, char *FileName, unsigned Options)
```

Visual Basic:

```
Function cbFileAInScan(ByVal BoardNum&, ByVal LowChan&, ByVal
HighChan&, ByVal Count&, Rate&, ByVal Range&, ByVal FileName$, ByVal
Options&) As Long
```

Delphi:

```
function cbFileAInScan(BoardNum:Integer; LowChan:Integer;
HighChan:Integer; Count:Longint; var Rate:Longint; Range:Integer;
FileName:PChar; Options:Integer):Integer;
```

Arguments:

`BoardNum`	The board number associated with the board when it was installed with *Insta*Cal. The specified board must have an A/D. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`LowChan`	First A/D channel of scan
`HighChan`	Last A/D channel of scan The maximum allowable channel depends on which type of A/D board is being used. For boards with both single ended and differential inputs, the maximum allowable channel number also depends on how the board is configured (for example, eight channels for differential, 16 for single ended).
`Count`	Specifies the total number of A/D samples that will be collected. If more than one channel is being sampled, the number of samples collected per channel is equal to `Count` / (`HighChan-LowChan` + 1).
`Rate`	Sample rate in samples per second (Hz) per channel. The maximum sampling rate depends on the A/D board that is being used (see `Rate` explanation `cbAInScan()`).
`Range`	If the selected A/D board does not have a programmable range feature, this argument is ignored. Otherwise set the `Range` argument to any range that is supported by the selected A/D board. Refer to board specific information for a list of the supported A/D ranges of each board.
`FileName`	The name of the file in which to store the data. If the file doesn't exist, it will be created. (When using the 16 bit version of the Universal Library, the named file must already exist. It should have been previously created with the MAKESTRM.EXE program.)
`Options`	Bit fields that control various options. Refer to the constants in the "Options argument values" section on page 101.

Returns:

Error code or 0 if no errors.

`Rate` = actual sampling rate.

Options argument values:

EXTCLOCK	If this option is used, conversions are controlled by the signal on the trigger input line rather than by the internal pacer clock. Each conversion is triggered on the appropriate edge of the trigger input signal (see board specific info). Additionally, the `Rate` argument is ignored. The sampling rate is dependent on the trigger signal.
EXTTRIGGER	If this option is specified, the sampling does not begin until the trigger condition is met. On many boards, this trigger condition is programmable (refer to the `cbSetTrigger()` function and board-specific information for details) and can be programmed for rising or falling edge or an analog level. On other boards, only 'polled gate' triggering is supported. Assuming active high operation, data acquisition commences immediately if the trigger input is high. If the trigger input is low, acquisition is held off until it goes high. Acquisition continues until `NumPoints&` samples are taken, regardless of the state of the trigger input. For 'polled gate' triggering, this option is most useful if the signal is a pulse with a very low duty cycle (trigger signal in TTL low state most of the time) to hold off triggering until the pulse occurs.
DTCONNECT	Samples are sent to the DT-Connect port if the board is equipped with one.

Notes:

OVERRUN Error - (Error code 29) This error indicates that the data was not written to the file as fast as the data was sampled. Consequently some data was lost. The value returned from `cbFileGetInfo()` in `TotalCount` is the number of points that were successfully collected.

Important

In order to understand the functions, read the board-specific information contained in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

We also urge you to examine and run one or more of the example programs supplied prior to attempting any programming of your own. Following this advice may save you hours of frustration, and wasted time.

This note, which appears elsewhere, is especially applicable to this function. Now is the time to read the board specific information for your board. We suggest that you make a copy of that page to refer to as you read this manual and examine the example programs.

cbFileGetInfo()

Returns information about a streamer file. When `cbFileAInScan()` or `cbFilePretrig()` fills the streamer file, information is stored about how the data was collected (sample rate, channels sampled etc.). This function returns that information. Refer to board-specific information to determine if this function is supported on your board.

Function prototype:

C/C++:
```
int cbFileGetInfo(char *FileName, short *LowChan, short *HighChan,
long *PretrigCount, long *TotalCount, long *Rate, int *Range)
```

Visual Basic:
```
Function cbFileGetInfo(ByVal FileName$, LowChan%, HighChan%,
PretrigCount&, TotalCount&, Rate&, Range&) As Long
```

Delphi:
```
function cbFileGetInfo(FileName:PChar; var LowChan:SmallInt; var
HighChan:SmallInt; var PretrigCount:Longint; var TotalCount:Longint;
var Rate:Longint; var Range:LongInt):Integer;
```

Arguments:

`FileName`	Name of streamer file.
`LowChan`	Variable to return `LowChan` to.
`HighChan`	Variable to return `HighChan` to.
`PretrigCount`	Variable to return `PretrigCount` to.
`TotalCount`	Variable to return `TotalCount` to.
`Rate`	Variable to return sampling rate to.
`Range`	Variable to return A/D range code to.

Returns:

Error code or 0 if no errors.

`LowChan` - low A/D channel of scan.

`HighChan` - high A/D channel of scan.

`TotalCount` - total number of points collected.

`PretrigCount` - number of pre-trigger points collected.

`Rate` - sampling rate when data was collected.

`Range` - `Range` of A/D when data was collected .

cbFilePretrig()

Scan a range of channels continuously while waiting for a trigger. Once the trigger occurs, return the specified number of samples including the specified number of pre-trigger samples to a disk file. This function waits for a trigger signal to occur on the Trigger Input. Once the trigger occurs, it returns the specified number (`TotalCount`) of A/D samples including the specified number of pre-trigger points. It collects the data at the specified sampling rate (`Rate`) from the specified range (`LowChan`-`HighChan`) of A/D channels from the specified board. If the A/D board has programmable gain then it sets the gain to the specified range. The collected data is returned to a file. See board-specific info to determine if this function is supported by your board.

Function prototype:

C/C++:

```
int cbFilePretrig (int BoardNum, int LowChan, int HighChan, long
*PretrigCount, long *TotalCount, long *Rate, int Range, char
*FileName, unsigned Options)
```

Visual Basic:

```
Function cbFilePretrig(ByVal BoardNum&, ByVal LowChan&, ByVal
HighChan&, PretrigCount&, TotalCount&, Rate&, ByVal Range&, ByVal
FileName$, ByVal Options&) As Long
```

Delphi:

```
function cbFilePretrig (BoardNum:Integer; LowChan:Integer;
HighChan:Integer; var PretrigCount:Longint; var TotalCount:Longint;
var Rate:Longint; Range:Integer; FileName:PChar;
Options:Integer):Integer;
```

Arguments:

`BoardNum` — The board number associated with the board when it was installed with the configuration program. The specified board must have an A/D and pretrigger capability. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`LowChan` — First A/D channel of scan

`HighChan` — Last A/D channel of scan

The maximum allowable channel depends on which type of A/D board is being used. For boards that have both single ended and differential inputs the maximum allowable channel number also depends on how the board is configured. Refer to board-specific information for the maximum number of channels allowed in differential and single ended modes.

`PretrigCount` — Specifies the number of samples before the trigger that will be returned. `PretrigCount` must be less than 16000 and `PretrigCount` must also be less than `TotalCount` - 512.

If the trigger occurs too early, then fewer than the requested number of pre-trigger samples will be collected. In that case a TOOFEW error will occur. The `PretrigCount` will be set to indicate how many samples were collected and the post trigger samples will still be collected.

`TotalCount` — Specifies the total number of samples that will be collected and stored in the file. `TotalCount` must be greater than or equal to `PretrigCount` + 512. If the trigger occurs too early then fewer than the requested number of samples will be collected. In that case a TOOFEW error will occur. The `TotalCount` will be set to indicate how many samples were actually collected.

`Rate`	Sample rate in samples per second (Hz) per channel. The maximum sampling rate depends on the A/D board that is being used. This is the rate at which scans are triggered. If you are sampling 4 channels, 0 - 3, then specifying a rate of 10,000 scans per second (10 kHz) will result in the A/D converter rate of 40 kHz: 4 channels at 10,000 samples per channel per second. This is different from some software where you specify the total A/D chip rate. In those systems, the per channel rate is equal to the A/D rate divided by the number of channels in a scan. This argument also returns the value of the actual set. This may be different from the requested rate because of pacer limitations.
`Range`	If the selected A/D board does not have a programmable range feature, this argument is ignored. Otherwise, set the `Range` argument to any range that is supported by the selected A/D board. Refer to board specific information for a list of the supported A/D ranges of each board.
`FileName`	The name of the file in which to store the data. If the file doesn't exist, it will be created. (When using the 16 bit version of the Universal Library, the named file must already exist. It should have been previously created with the MAKESTRM.EXE program.)
`Options`	Bit fields that control various options. Refer to the constants in the "Options argument values" section below.

Returns:

Error code or 0 if no errors.

`PretrigCount` - actual number of pre-trigger samples collected.

`TotalCount` - actual number of samples collected.

`Rate` = actual sampling rate.

Options argument values:

`EXTCLOCK`	If this option is used then conversions will be controlled by the signal on the trigger input line rather than by the internal pacer clock. Each conversion will be triggered on the appropriate edge of the trigger input signal (see board specific info). When this option is used the `Rate` argument is ignored. The sampling rate is dependent on the trigger signal.
`DTCONNECT`	Samples are sent to the DT-Connect port if the board is equipped with one.

Notes:

OVERRUN Error - (Error code 29) This error indicates that the data was not written to the file as fast as the data was sampled. Consequently some data was lost. The value in `TotalCount` will be the number of points that were successfully collected.

cbFileRead()

Reads data from a streamer file. See board-specific info to determine if this function is supported on your board.

Function prototype:

C/C++:	`int cbFileRead(char *FileName, long FirstPoint, long *TotalCount, int *DataBuffer)`
Visual Basic:	`Function cbFileRead(ByVal FileName$, ByVal FirstPoint&, TotalCount&, DataBuffer%) As Long`
Delphi:	`function cbFileRead (FileName:PChar; FirstPoint:Longint; var NumPoints:Longint; var DataBuffer:Word):Integer;`

Arguments:

`FileName`	Name of streamer file
`FirstPoint`	Index of first point to read
`TotalCount`	Number of points to read from file
`DataBuffer`	Pointer to data buffer that data will be read into.

Returns:

Error code or 0 if no errors.

`DataBuffer` - data read from file.

`TotalCount` - number of points actually read.

`TotalCount` may be less than the requested number of points if an error occurs.

Notes:

Data format: The data is returned as 16-bits. The 16-bits may represent 12-bits of analog, 12-bits of analog plus 4 bits of channel, or 16-bits of analog. Use `cbAConvertData()` to correctly load the data into an array.

Loading portions of files: The file may contain much more data than can fit in `DataBuffer`. In those cases use `TotalCount` and `FirstPoint` to read a selected piece of the file into `DataBuffer`. Call `cbFileGetInfo()` first to find out how many points are in the file.

10

Temperature Input Functions

Introduction

Use the functions discussed in this chapter to convert a raw analog input from an EXP board, or other temperature sensor board, to temperature.

cbTIn()

Changed R3.3 ID

Reads an analog input channel, linearizes it according to the selected temperature sensor type, and returns the temperature in degrees. The CJC channel, the gain, and sensor type, are read from the *Insta*Cal configuration file. They should be set by running the *Insta*Cal configuration program.

Function prototype:

C/C++:	`int cbTIn(int BoardNum, int Chan, int Scale, float *TempVal, int Options)`
Visual Basic:	`Function cbTIn(ByVal BoardNum&, ByVal Chan&, ByVal Scale&, TempVal!, ByVal Options&) As Long`
Delphi:	`function cbTIn (BoardNum:Integer; Chan:Integer; Scale:Integer; var TempValue:Single; Options:Integer):Integer;`

Arguments:

`BoardNum`	`BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`Chan`	Input channel to read.
`Scale`	Specifies the temperature scale that the input will be converted to. Choices are `CELSIUS`, `FAHRENHEIT` and `KELVIN`.
`TempVal`	The temperature in degrees is returned here. Thermocouple resolution is approximately 0.25 °C, depending on scale, range and thermocouple type. RTD resolution is 0.1 °C.
`Options`	Bit fields that control various options. Refer to the constants in the "Options argument values" section below.

Returns:

Error code or 0 if no errors.

*`TempVal` - Temperature returned here.

Options argument values:

`FILTER`	When selected, a smoothing function is applied to temperature readings, very much like the electrical smoothing inherent in all hand held temperature sensor instruments. This is the default. When selected, 10 samples are read from the specified channel and averaged. The average is the reading returned. Averaging removes normally distributed signal line noise.
`NOFILTER`	If you use the `NOFILTER` option, then the readings will not be smoothed and you will see a scattering of readings around a mean.

Notes:

Using CIO-EXP boards: For CIO-EXP boards, the channel number is calculated using the following formula, where:

- ADChan is the A/D channel that is connected to the multiplexer.
- `MuxChan` is a number ranging from 0 to 15 that specifies the channel number on a particular bank of the multiplexer board

```
Chan = (ADChan * 16) + (16 + MuxChan)
```

For example, you have an EXP16 connected to a CIO-DAS08 via the CIO-DAS08 channel 0. (Remember that DAS08 channels are numbered 0, 1, 2, 3, 4, 5, 6 & 7). If you connect a thermocouple to channel 5 of the EXP16, the value for `chan` would be (0 * 16) + (16 + 5)= 0 + 21 = **21**.

Using 6K-EXP boards: For 6K-EXP boards, the channel number (Chan) is calculated using one of the following formulas, where:

- ADChan is the A/D channel that is connected to the multiplexer.
- `MuxChan` is a number ranging from 0 to 15 that specifies the channel number on a particular bank of the multiplexer board.
- If the A/D board has 16 or less single-ended channels:

  `Chan = (ADChan * 16) + (16 + MuxChan)`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6052 via the a PCI-DAS6052 channel 0. If you connect a thermocouple to channel 5 of the 6K-EXP16, the value for `Chan` would be (0 * 16) + (16 + 5)= 0 + 21 = **21**.
- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is less than or equal to 7:

  `Chan = (ADChan * 16) + (64 + MuxChan)`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6031 via the a PCI-DAS6031 channel 7. If you connect a thermocouple to channel 5 of the 6K-EXP16, the value for `Chan` would be (7 * 16) + (64 + 5) = 112 + 69 = **181**.
- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is greater than or equal to 31:

  `Chan = (ADChan * 16 – 320) + MuxChan`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6031 via the PCI-DAS6031 channel 32. If you connect a thermocouple to channel 5 of the 6K-EXP16, the value for `Chan` would be (32 * 16 – 320) + 5 = 192 + 5 = **197**.

CJC Channel: The CJC channel is set in the *Insta*Cal install program. If you have multiple EXP boards, Universal Library will apply the CJC reading to the linearization formula in the following manner:

3. If you have chosen a CJC channel for the EXP board that the channel you are reading is on, it will use the CJC temp reading from that channel.
4. If you left the CJC channel for the EXP board that the channel you are reading is on to `NOT SET`, the library will use the CJC reading from the next lower EXP board with a CJC channel selected.

For example: You have four CIO-EXP16 boards connected to a CIO-DAS08 on channel 0, 1, 2 and 3. You choose CIO-EXP16 #1 (connected to CIO-DAS08 channel 0) to have its CJC read on CIO-DAS08 channel 7, AND, you leave the CIO-EXP16's 2, 3 and 4 CJC channels to NOT SET. Result: The CIO-EXP boards will all use the CJC reading from CIO-EXP16 #1, connected to channel 7 for linearization. As you can see, it is important to keep the CIO-EXP boards in the same case and out of any breezes to ensure a clean CJC reading.

Important

For an EXP board connected to an A/D board that does not have programmable gain (DAS08, DAS16, DAS16F), the A/D board range is read from the configuration file (cb.cfg). In most cases, set hardware-selectable ranges to ±5 V for thermocouples, and to 0 to 10 V for RTDs. Refer to the board-specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) or in the user manual for your board. If the board has programmable RTDs gains, the cbTIn() function sets the appropriate A/D range.

Specific Errors: If an `OUTOFRANGE` or `OPENCONNECTION` error occurs, the value returned is -9999.0.

cbTInScan()

Changed R3.3 ID

Reads a range of channels from an analog input board, linearizes them according to temperature sensor type, and returns the temperatures to an array in degrees. The CJC channel, the gain, and temperature sensor type are read from the configuration file. Use the *Insta*Cal configuration program to change any of these options.

Function prototype:

C/C++:

```
int cbTInScan(int BoardNum, int LowChan, int HighChan, int Scale,
float DataBuffer[], int Options)
```

Visual Basic:

```
Function cbTInScan(ByVal BoardNum&, ByVal LowChan&, ByVal HighChan&,
ByVal Scale&, DataBuffer!, ByVal Options&) As Long
```

Delphi:

```
function cbTInScan(BoardNum:Integer; LowChan:Integer;
HighChan:Integer; Scale:Integer; var DataBuffer:Single;
Options:Integer):Integer;
```

Arguments:

`BoardNum`	`BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`LowChan`	Low mux channel of scan.
`HighChan`	High mux channel of scan.
`Scale`	Specifies the temperature scale that the input will be converted to. Choices are `CELSIUS`, `FAHRENHEIT` and `KELVIN`.
`DataBuffer`	The temperature is returned in degrees. Each element in the array corresponds to a channel in the scan. `DataBuffer` must be at least large enough to hold `HighChan` - `LowChan` + 1 temperature values. Thermocouple resolution is approximately 0.25° C, depending on scale, range and thermocouple type. RTD resolution is 0.1 °C.
`Options`	Bit fields that control various options. Refer to the constants in the "Options argument values" section below.

Returns:

Error code or 0 if no errors.

`DataBuffer`[] - Temperature values in degrees are returned here for each channel in scan.

Options argument values:

`FILTER`	When selected, a smoothing function is applied to temperature readings, very much like the electrical smoothing inherent in all hand held temperature sensor instruments. This is the default. When selected, 10 samples are read and averaged on each channel. The average is the reading returned. Averaging removes normally distributed signal line noise.
`NOFILTER`	If you use the `NOFILTER` option then the readings will not be smoothed, and you will see a scattering of readings around a mean.

Notes:

Using EXP boards: For EXP boards, these channel numbers are calculated using the following formula:

- ADChan = A/D channel that is connected to the multiplexer

- `MuxChan` is a number ranging from 0 to 15 that specifies the channel number on a particular bank of the multiplexer board

  `Chan = (ADChan *16) + (16 + MuxChan)`

For example, you have an EXP16 connected to a CIO-DAS08 via the CIO-DAS08 channel 0. (Remember, DAS08 channels are numbered 0, 1, 2, 3, 4, 5, 6 & 7). If you connect thermocouples to channels 5, 6, and 7 of the EXP16, the value for `LowChan` would be (0 * 16) + (16 + 5)= 0 + 21 = **21**, and the value for `HighChan` would be (0 * 16) + (16 + 7)= 0 + 21 = **23**.

> **Important**
> For an EXP board connected to an A/D board that does not have programmable gain (DAS08, DAS16, DAS16F), the A/D board range is read from the configuration file (cb.cfg). In most cases, set hardware-selectable ranges to ±5 V for thermocouples, and to 0 to 10 V for RTDs. Refer to the board-specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) or in the user manual for your board. If the board has programmable RTDs gains, the cbTIn() function sets the appropriate A/D range.

Using 6K-EXP boards: For 6K-EXP boards, the channel number is calculated using one of the following formulas, where:

- ADChan is the A/D channel that is connected to the multiplexer.
- `MuxChan` is a number ranging from 0 to 15 that specifies the channel number (Chan) on a particular bank of the multiplexer board.
- If the A/D board has 16 or less single-ended channels:

  `Chan = (ADChan  * 16) + (16 + MuxChan)`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6052 via the a PCI-DAS6052 channel 0. If you connect a thermocouple to channels 5, 6, and 7 of the 6K-EXP16, the value for `LowChan` would be (0 * 16) + (16 + 5)= 0 + 21 = **21**, and the value for `highChan` would be (0 * 16) + (16 + 5)= 0 + 231 = **23**.
- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is less than or equal to 7:

  `Chan = (ADChan * 16) + (64 + MuxChan)`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6031 via the a PCI-DAS6031 channel 7. If you connect a thermocouple to channels 5, 6, and 7 of the 6K-EXP16, the value for LowChan would be (7 * 16) + (64 + 5) = 112 + 69 = **181**, and the value for `HighChan` would be (7 * 16) + (64 + 7) = 112 + 71 = **183**.
- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is greater than or equal to 32:

  `Chan = (ADChan * 16 - 320) + MuxChan`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6031 via the PCI-DAS6031 channel 32. If you connect a thermocouple to channels 5, 6, and 7 of the 6K-EXP16, the value for LowChan would be (32 * 16 – 320) + 5 = 192 + 5 = **197**, and the value for HighChan would be (32 * 16 – 320) + 7 = 192 + 7 = **199**.

CJC Channel: The CJC channel is set in the *Insta*Cal install program. If you have multiple EXP boards, Universal Library will apply the CJC reading to the linearization formula in the following manner:

- First, if you have chosen a CJC channel for the EXP board that the channel you are reading is on, it will use the CJC temp reading from that channel.
- Second, if you have left the CJC channel for the EXP board that the channel you are reading is on to NOT SET, the library will use the CJC reading from the next lower EXP board with a CJC channel selected.

For example: You have four CIO-EXP16 boards connected to a CIO-DAS08 on channel 0, 1, 2 and 3. You choose CIO-EXP16 #1 (connected to CIO-DAS08 channel 0) to have its CJC read on CIO-DAS08 channel 7, AND, you leave the CIO-EXP16's 2, 3 and 4 CJC channels to NOT SET. Result: The CIO-EXP boards will all use the CJC reading from CIO-EXP16 #1, connected to channel 7 for linearization. As you can see, it is important to keep the CIO-EXP boards in the same case and out of any breezes to ensure a clean CJC reading.

Important

In order to understand the functions, refer to the board-specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) and also in the Readme files installed with the Universal Library. We also urge you to examine and run one or more of the example programs supplied prior to attempting any programming of your own. Following this advice may save you hours of frustration, and wasted time. This note, which appears elsewhere, is especially applicable to this function. Now is the time to read the board specific information for your board. We suggest that you make a copy of that page to refer to as you read this manual and examine the example programs.

Specific Errors: For most boards, if an `OUTOFRANGE` or `OPENCONNECTION` error occurs, the value in the array element associated with the channel causing the error returned will be -9999.0 (Refer to board specific information).

Windows Memory Management Functions

Introduction

Use the functions explained in this chapter when you run the Windows version of the library. These functions allocate, free and copy to/from Windows global memory buffers. These functions are not used in VEE, since VEE handles memory allocation. For customers wishing to customize memory management under VEE, the source code to CBV.DLL and CBV32.DLL is available. Please call technical support and request it.

cbWinBufAlloc()

Allocates a Windows global memory buffer which can be used with the scan functions and returns a memory handle for it.

Function prototype:

C/C++:	`int cbWinBufAlloc (long NumPoints)`
Visual Basic:	`Function cbWinBufAlloc(ByVal NumPoints&) As Long`
Delphi:	`function cbWinBufAlloc (NumPoints:Longint):Integer;`

Arguments:

`NumPoints`	Size of buffer to allocate. Specifies how many data points (16-bit integers, NOT bytes) can be stored in the buffer.

Returns:

0 if buffer could not be allocated or a non-zero integer handle to the buffer.

Notes:

Unlike most other functions in the library, this function does not return an error code. It returns a Windows global memory handle which can then be passed to the scan functions in the library. If an error occurs the handle will come back as 0 to indicate the error.

cbWinBufFree()

Frees a Windows global memory buffer which was previously allocated with the `cbWinBufAlloc()` function.

Function prototype:

C/C++:	`int cbWinBufFree(int MemHandle)`
Visual Basic:	`Function cbWinBufFree(ByVal MemHandle&) As Long`
Delphi:	`function cbWinBufFree(MemHandle:Integer):Integer;`

Arguments:

`MemHandle`	A Windows memory handle. This must be a memory handle that was returned by `cbWinBufAlloc()` when the buffer was allocated.

Returns:

Error code or zero if no errors.

cbWinArrayToBuf()

Copies data from an array into a Windows memory buffer.

Function prototype:

C/C++:	`int cbWinArrayToBuf(unsigned short *DataArray, int MemHandle, long FirstPoint, long Count)`
Visual Basic:	`Function cbWinArrayToBuf(DataArray%, ByVal MemHandle&, ByVal FirstPoint&, ByVal Count&) As Long`
Delphi:	`function cbWinArrayToBuf(var DataArray:Word; MemHandle:Integer; FirstPoint:Longint; Count:Longint):Integer;`

Arguments:

`DataArray`	The array containing the data to be copied.
`MemHandle`	This must be a memory handle that was returned by `cbWinBufAlloc()` when the buffer was allocated. The data will be copied into this buffer.
`FirstPoint`	Index of first point in memory buffer where data will be copied to.
`Count`	Number of data points to copy.

Returns:

Error code or zero if no errors.

Notes:

This function copies data from an array to a Windows global memory buffer. This would typically be used to initialize the buffer with data before doing an output scan. Using the `FirstPoint` and `Count` argument it is possible to fill a portion of the buffer. This can be useful if you want to send new data to the buffer after a `BACKGROUND`+`CONTINUOUS` scan command has sent the old data – for example, with circular buffering.

Although this function is available to both Windows C and Delphi programs, it is not necessary, since you can manipulate the memory buffer directly by casting the `MemHandle` returned from `cbWinBufAlloc()` to the appropriate type. This method avoids having to copy the data from an array to a memory buffer. The following example illustrates this method:

```
long Count= 1000;
unsigned short *DataArray=NULL;
int MemHandle = 0;

/*allocate the buffer and cast it to an unsigned short*/
MemHandle = cbWinBufAlloc(Count);
DataArray = (unsigned short*)MemHandle;

/*calculate and store the waveform*/
for(int i=0; i<Count; ++i)
  DataArray[i] = 2047*(1.0 + sin(6.2832*i/Count));

/*output the waveform*/
cbAOutScan (......,MemHandle,...);

/*free the buffer and NULL the pointer*/
cbWinBufFree(MemHandle);
DataArray = NULL;
```

cbWinBufToArray()

Copies data from a Windows memory buffer into an array.

Function prototype:

C/C++:	`int cbWinBufToArray(int MemHandle, unsigned short*DataArray, long FirstPoint, long Count)`
Visual Basic:	`Function cbWinBufToArray(ByVal MemHandle&, DataArray%, ByVal FirstPoint&, ByVal Count&) As Long`
Delphi:	`function cbWinBufToArray (MemHandle:Integer; var DataArray:Word; FirstPoint:Longint; Count:Longint):Integer;`

Arguments:

`MemHandle`	This must be a memory handle that was returned by `cbWinBufAlloc()` when the buffer was allocated. The buffer should contain the data that you want to copy.
`DataArray`	The array that the data will be copied to.
`FirstPoint`	Index of first point in memory buffer that data will be copied from.
`Count`	Number of data points to copy.

Returns:

Error code or zero if no errors.

Notes:

This function copies data from a Windows global memory buffer to an array. This would typically be used to retrieve data from the buffer after executing an input scan function.

Using the `FirstPoint` and `Count` argument it is possible to copy only a portion of the buffer to the array. This can be useful if you want foreground code to manipulate previously collected data while a `BACKGROUND` scan continues to collect new data.

Although this function is available to both Windows C and Delphi programs, it is not necessary, since it is possible to manipulate the memory buffer directly by casting the `MemHandle` returned from `cbWinBufAlloc()` to the appropriate type. This method avoids having to copy the data from the memory buffer to an array - Refer to the following example.

```
/*declare and initialize the variables*/
long Count=1000;
unsigned short *DataArray=NULL;
int MemHandle=0;

/*allocate the buffer and cast it to a pointer to an unsigned short*/
MemHandle = cbWinBufAlloc(Count);
DataArray = (unsigned short*)MemHandle;

/*output the waveform*/
cbAInScan (......,MemHandle,...);

/*print the results*/
for(int i=0; i<Count; ++i)
 printf("Data[%d]=%d\n", DataArray[i]);

/*free the buffer and NULL the pointer*/
cbWinBufFree(MemHandle);
DataArray = NULL;
```

12

Miscellaneous Functions

Introduction

The functions explained in this chapter do not as a group fit into a single category. They get and set board information, convert units, manage events and background operations, and perform serial communication operations.

cbDisableEvent() (32-bit UL Only)

Disables one or more event conditions and disconnects their user-defined handlers.

Function prototype:

C/C++:	`int cbDisableEvent (int BoardNum, unsigned EventType)`
Visual Basic:	`Function cbDisableEvent(ByVal BoardNum&, ByVal EventType&) as Long`
Delphi:	`Function cbDisableEvent(BoardNum:Integer; EventType:Integer):Integer;StdCall`

Arguments:

`BoardNum`	The board number used to indicate which device's event handling will be disabled. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library). Refers to the number associated with the board installed with the *Insta*Cal configuration program.
`EventType`	Specifies one or more event conditions to disable. More than one event type can be specified by bitwise OR'ing the event types. Note that specifying an event that has not been enabled is benign and will not cause any errors. Refer to "EventType argument values" on page 121 for valid `EventType` settings. To disable all events in a single call, use `ALL_EVENT_TYPES`.

Returns:

Error code or 0 if no errors.

Notes:

For most event types, this function cannot be called while any background operations (`cbAInScan()`, `cbAPretrig()`, or `cbAOutScan()`) are active. Perform a `cbStopBackground()` before calling `cbEnableEvent()`. However, for ON_EXTERNAL_INTERRUPT events, you can call `cbDisableEvent()` while the board is actively generating events.

Important

In order to understand the functions, refer to the board-specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) and also in the Readme files installed with the Universal Library. We also urge you to examine and run one or more of the example programs supplied prior to attempting any programming of your own. Following this advice may save you hours of frustration, and wasted time. This note, which appears elsewhere, is especially applicable to this function. Now is the time to read the board specific information for your board. We suggest that you make a copy of that page to refer to as you read this manual and examine the example programs.

cbEnableEvent() (32-bit UL Only)

Binds one or more event conditions to a user-defined callback function. Upon detection of an event condition, the user-defined function is invoked with board- and event-specific data. Detection of event conditions occurs in response to interrupts. Typically, this function is used in conjunction with interrupt driven processes such as `cbAInScan()`, `cbAPretrig()`, or `cbAOutScan()`.

Function prototype:

C/C++:
```
int cbEnableEvent (int BoardNum, unsigned EventType, unsigned
EventParam,  void* CallbackFunc, void* UserData)
```

Visual Basic:
```
Function cbEnableEvent (ByVal BoardNum&, ByVal EventType&, ByVal
EventParam&, ByVal CallbackFunc&, ByRef UserData as Any) as Long
```

Delphi:
```
Function cbEnableEvent(BoardNum:Integer; EventType:Integer;
EventParam:Integer; CallbackFunc:Pointer;
UserData:Pointer):Integer;StdCall
```

Arguments:

`BoardNum`	The board number used to indicate which device will generate the event conditions. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library). Refers to the number associated with the board installed with the *Insta*Cal configuration program.
`EventType`	Specifies one or more event conditions that will be bound to the user-defined callback function. More than one event type can be specified by bitwise OR'ing the event types Refer to the constants in the "EventType argument values" section below.
`EventParam`	Additional data required to specify some event conditions such as the `ON_DATA_AVAILABLE` event. For `ON_DATA_AVAILABLE` events, this is used to determine the minimum number of samples to acquire during an analog input scan before generating the event. Most event conditions ignore this value.
`CallbackFunc`	The address of or pointer to the user-defined callback function to handle the above event type(s). This function must be defined using the standard call (__stdcall) calling convention. Consequently, Visual Basic programs must define their callback functions in standard modules(.bas) and cannot be object methods. C++ programs can define this callback function as either a global function or as a `static` member function of a class (note that static members do NOT have access to instance specific data). Refer to the "User Callback function" on page 123 for proper function syntax.
`UserData`	The address of or pointer to user-defined data that will be passed to the user-defined callback function. This parameter is NOT dereferenced by the library or its drivers; as a consequence, a NULL pointer can be supplied.

Returns:

Error code or 0 if no errors.

EventType argument values:

`ON_SCAN_ERROR`	Generates an event upon detection of a driver error during `BACKGROUND` input and output scans. This includes `OVERRUN`, `UNDERRUN`, and `TOOFEW` errors.
`ON_EXTERNAL_INTERRUPT`	For some digital and counter boards, generates an event upon detection of a pulse at the External Interrupt pin.
`ON_PRETRIGGER`	For `cbAPretrig()`, generates an event upon detection of the first trigger.

`ON_DATA_AVAILABLE`	Generates an event whenever the number of samples acquired during an analog input scan increases by `EventParam` samples or more. Note that for `BLOCKIO` scans, events will be generated on packet transfers; for example, even if `EventParam` is set to 1, events will only be generated every packet-size worth of data (256 samples for the PCI-DAS1602) for aggregate rates greater than 1 kHz for the default `cbAInScan()` mode. For `cbAPretrig()`, the first event is not generated until a minimum of `EventParam` samples after the pretrigger.
`ON_END_OF_AI_SCAN`	Generates an event upon completion or fatal error of a `cbAInScan()` or `cbAPretrig()`. This event is NOT generated when scans are aborted using `cbStopBackground()`.
`ON_END_OF_AO_SCAN`	Generates an event upon completion or fatal error of a `cbAOutScan()`. This event is not generated when scans are aborted using `cbStopBackground()`.

Notes:

- This function cannot be called while any background operations (`cbAInScan()`, `cbAPretrig()`, or `cbAOutScan()`) are active. If a background operation is in progress when `cbEnableEvent()` is called, the function returns an `ALREADYACTIVE` error. Perform a `cbStopBackground()` before calling `cbEnableEvent()`.
- Events can be generated no faster than the user callback function can handle them. If an event type becomes multiply signaled before the event handler returns, events are merged. The event handler is called once per event type and is supplied with the event data corresponding to the latest event. In addition, if more than one event type becomes signaled, the event handler for each event type is called in the same order in which they are listed above.
- Events are generated while handling board-generated interrupts. Therefore, using `cbStopBackground()` to abort background operations *does not* generate `ON_END_OF_AI_SCAN` or `ON_END_OF_AO_SCAN` events. However, the event handlers can be called immediately after calling `cbStopBackground()`.
- `cbEnableEvent()` is intended for use with Windows applications. Use with console or DOS applications can produce unpredictable results.

User Callback function (32-bit UL only)

The User Callback function is called as an argument of the cbEnableEvent() function. You create the function using the prototype shown below. You call the function by passing either it's address or a pointer to the function to the `CallbackFunc` argument of the `cbEnableEvent()` function.

Callback function prototype:

C/C++:

```
void __stdcall CallbackFunc (int BoardNum, unsigned EventType,
unsigned EventData, void* UserData);
```

Visual Basic:

```
Sub CallbackFunc (ByVal BoardNum&, ByVal EventType&, ByVal
EventData&, ByRef UserData as UserDataType)
```

where `UserDataType` is the data type of the `UserData` argument passed in to `cbEnableEvent()` (refer to page 121).

Delphi:

```
procedure CallbackFunc (BoardNum:Integer; EventType:Integer;
EventData:Integer; UserData:Pointer);
```

Arguments:

`BoardNum`	Indicates which board caused the event.
`EventType`	Indicates which event occurred.
`EventData`	Board specific data associated with this event. Set it to one of the constants in the "EventData argument values" section below.
`UserData`	The pointer or reference to data supplied by the `UserData` parameter in `cbEnableEvent()` (refer to page 121). Note that before use, this parameter must be cast to the same data type as passed in to `cbEnableEvent()`.

EventData argument values:

`ON_SCAN_ERROR`	The Error code of the scan error.
`ON_EXTERNAL_INTERRUPT`	The number of interrupts generated since enabling the `ON_EXTERNAL_INTERRUPT` event.
`ON_PRETRIGGER`	The number of pretrigger samples available at time of pretrigger. This value is invalid for some boards when a `TOOFEW` error occurs. See board details.
`ON_DATA_AVAILABLE`	The number of samples acquired since the start of scan.
`ON_END_OF_AI_SCAN`	The total number of samples acquired upon scan completion or end.
`ON_END_OF_AO_SCAN`	The total number of samples output upon scan completion or end.

cbFlashLED()

Causes the LED on a USB device to flash.

Function prototype:

C/C++:	int `cbFlashLED` (int `BoardNum`);
Visual Basic:	Function `cbFlashLED` (ByVal `BoardNum`&) as Long
Delphi:	function `cbFlashLED` (`BoardNum`:Integer):Integer;

Arguments:

`BoardNum`	The board number of the USB device whose LED will flash.

cbFromEngUnits()

Converts a voltage (or current) in engineering units to a D/A count value for output to a D/A.

Function prototype:

C/C++:
```
int cbFromEngUnits(int BoardNum, int Range, float EngUnits, unsigned
short *DataVal)
```

Visual Basic:
```
Function cbFromEngUnits(ByVal BoardNum&, ByVal Range&, ByVal
EngUnits!, DataVal%) As Long
```

Delphi:
```
function cbFromEngUnits(BoardNum:Integer; Range:Integer;
EngUnits:Single; var DataVal:Word):Integer;
```

Arguments:

`BoardNum` The board number associated with the D/A board when it was installed. This function uses the board number to determine the range and resolution values to use in the conversion. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`Range` D/A voltage (or current) range. Some D/A boards have programmable voltage ranges, others set the voltage range via switches on the board. In either case, the selected range must be passed to this function. Each D/A board supports different voltage and/or current ranges. Refer to board specific information for the list of ranges supported by each board.

`EngUnits` The voltage (or current) value to set the D/A to. Set the value to be within the range specified by the `Range` argument.

`DataVal` The function returns a D/A count to this variable that is equivalent to the EngUnits argument.

Returns:

Error code or 0 if no errors.

`DataVal` – the binary counts equivalent to `EngUnits` is returned here.

cbGetBoardName()

Returns the board name of a specified board.

Function prototype:

C/C++:	`int cbGetBoardName(int BoardNum, char *BoardName)`
Visual Basic:	`Function cbGetBoardName(ByVal BoardNum&, ByVal BoardName$) As Long`
Delphi:	`function cbGetBoardName(BoardNum:Integer; BoardName:PChar):Integer;`

Arguments:

`BoardNum`	Refers either to the board number associated with a board when it was installed, or `GETFIRST` or `GETNEXT`. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library), `GETFIRST` or `GETNEXT`
`BoardName`	A null-terminated string variable that the board name will be returned to. This string variable must be pre-allocated to be at least as large as `BOARDNAMELEN`. This size is guaranteed to be large enough to hold the longest board name string. The "Appendix" in the *Universal Library User Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) lists the board names and associated device ID codes.

Returns:

Error code or 0 if no errors.

BoardName - return string containing the board name.

Notes:

There are two distinct ways of using this function:

- Pass a board number as the `BoardNum` argument. The string that is returned describes the board type of the installed board.
- Set `BoardNum` to `GETFIRST` or `GETNEXT` to get a list of all board types that are supported by the library. Set `BoardNum` to `GETFIRST` to get the first board type in the list of supported boards. Subsequent calls with `Board=GETNEXT` returns each of the other board types supported by the library. When you reach the end of the list, `BoardName` is set to an empty string. Refer to the `ulgt04` example program in the installation directory for more details.

cbGetStatus()

Returns the status about the background operation currently running.

Function prototype:

C/C++:	`int cbGetStatus (int BoardNum, int *Status, long *CurCount, long *CurIndex, int FunctionType)`
Visual Basic:	`Function cbGetStatus(ByVal BoardNum&, Status%, CurCount&, CurIndex&, FunctionType&) As Long`
Delphi:	`function cbGetStatus (BoardNum:Integer; var Status:SmallInt; var CurCount:Longint; var CurIndex:Longint; FunctionType:Integer):Integer;`

Arguments:

`BoardNum`
: The board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for the 16-bit version of Universal Library).

`Status`
: `Status` indicates whether or not a background process is currently executing.

`CurCount`
: Specifies how many points have been input or output. It can be used to gauge how far along the operation is towards completion. Generally the `CurCount` will return the total number of samples collected at the time of the call to `cbGetStatus()`.

  However, when `CONTINUOUS` and `BACKGROUND` options are both set, `CurCount` behavior depends on the board type and transfer mode. This value may recycle as the circular buffer recycles, or may continuously increment with the number of counts transferred. Also, `CurCount` may not update on each sample. For example, when running in `BLOCKIO` mode, `CurCount` updates after each packet of data has been transferred. The packet size is board-dependent. Refer to the *Universal Library User's Guide* for board-specific information.

`CurIndex`
: `CurIndex` is an index into the data buffer that points at the start of the last completed channel scan. It can be used to provide a real time display for a background operation. `DataBuffer[CurIndex]` points to the start of the last complete channel scan that was put in or taken out of the buffer. You should expect `CurIndex` to increment by the number of channels in the scan as well. If no points in the buffer have been accessed yet, `CurIndex` will equal -1. This value can also behave differently when `CONTINUOUS` and `BACKGROUND` options are both set (see `CurCount` description). Refer to board-specific information for details.

  If you use the `CONVERTDATA` option with either the `CONTINUOUS` option or with pre-triggering functions, `CurIndex` returns the index of the last A/D sample, rather than the start of the last completed channel scan.

  For many background operations `CurCount` = `CurIndex`. For Pre-Trigger inputs though, they are different. If the hardware allows background trigger operations, `CurCount` indicates how many points of the `TotalCount` have been collected. `CurCount` will rise to `PretrigCount`, stop until the trigger occurs then rise to `TotalCount`. `CurIndex`, though, will constantly increase and reset as it goes around and around the circular buffer while waiting for the trigger to occur.

`FunctionType`
: Specifies which scan to retrieve status information about. Set it to one of the constants in the "FunctionType argument values" section on page 128.

Returns:

Error code or 0 if no errors

`Status` -	`IDLE` - No background operation has been executed
	`RUNNING` - Background operation still underway

`CurCount` - current number of samples collected

`CurIndex` - Current sample index

FunctionType argument values:

`AIFUNCTION`	Specifies analog input scans started with `cbAInScan()` or `cbAPretrig()`.
`AOFUNCTION`	Specifies analog output scans started with `cbAOutScan()`.
`DIFUNCTION`	Specifies digital input scans started with `cbDInScan()`.
`DOFUNCTION`	Specifies digital output scans started with `cbDOutScan()`.
`CTRFUNCTION`	Specifies counter background operations started with `cbCStoreOnInt()`.

Notes:

VEE Programs Stopping Background Tasks Early: You must use the red STOP button on the `cbGetStatus()` panel to stop background processes before the scheduled completion. If you use the stop button on the VEE icon bar instead, the background process continues to run in the background. The result of this action and exiting VEE is undefined. Always use the `cbGetStatus()` STOP button. Refer to the example programs ULAI03.VEE through ULAI06.VEE in the installation directory for details.

cbInByte()

Reads a byte from a hardware register on a board.

Function prototype:

C/C++:	`int cbInByte(int BoardNum, int PortNum)`
Visual Basic:	`Function cbInByte(ByVal BoardNum&, ByVal PortNum&) As Long`
Delphi:	`function cbInByte(BoardNum:Integer; PortNum:Integer):Integer;`

Arguments:

`BoardNum`	Refers to the board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`PortNum`	Register within the board. Boards are set to a particular base address. The registers on the boards are at addresses that are offsets from the base address of the board (BaseAdr + 0, BaseAdr + 2, etc). Set this argument to the offset for the desired register. This function takes care of adding the base address to the offset, so that the board's address can be changed without changing the code.

Returns:

The current value of the specified register

Notes:

`cbInByte()` is used to read 8 bit ports. `cbInWord()` is used to read 16-bit ports.

This function was designed for use with ISA bus boards. Use with PCI bus boards is not recommended.

cbInWord()

Reads a word from a hardware register on a board.

Function prototype:

C/C++:	`int cbInWord (int BoardNum, int PortNum)`
Visual Basic:	`Function cbInWord(ByVal BoardNum&, ByVal PortNum&) As Long`
Delphi:	`function cbInWord(BoardNum:Integer; PortNum:Integer):Integer;`

Arguments:

`BoardNum` Refers to the board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`PortNum` Register within the board. Boards are set to a particular base address. The registers on the boards are at addresses that are offsets from the base address of the board (BaseAdr + 0, BaseAdr + 2, etc).

Set this argument to the offset for the desired register. This function takes care of adding the base address to the offset, so that the board's address can be changed without changing the code.

Returns:

The current value of the specified register.

Notes:

`cbInByte()` is used to read 8-bit ports. `cbInWord()` is used to read 16-bit ports.

This function was designed for use with ISA bus boards. Use with PCI bus boards is not recommended.

cbOutByte()

Writes a byte to a hardware register on a board.

Function prototype:

C/C++:	`int cbOutByte (int BoardNum, int PortNum, int PortVal)`
Visual Basic:	`Function cbOutByte(ByVal BoardNum&, ByVal PortNum&, ByVal PortVal%) As Long`
Delphi:	`function cbOutByte(BoardNum:Integer; PortNum:Integer; PortVal:Integer):Integer;`

Arguments:

`BoardNum`	Refers to the board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`PortNum`	register within the board. Boards are set to a particular base address. The registers on the boards are at addresses that are offsets from the base address of the board (BaseAdr + 0, BaseAdr + 2, etc). Set this argument to the offset for the desired register. This function takes care of adding the base address to the offset, so that the board's address can be changed without changing the code.
`PortVal`	The value that is written to the register.

Returns:

Error code or 0 if no errors

Notes:

`cbOutByte()` is used to write to 8-bit ports. `cbOutWord()` is used to write to 16-bit ports.

This function was designed for use with ISA bus boards, and is not recommended for use with PCI-bus boards.

cbOutWord()

Writes a word to a hardware register on a board.

Function prototype:

C/C++: `int cbOutWord(int BoardNum, int PortNum, int PortVal)`

Visual Basic: `Function cbOutByte(ByVal BoardNum&, ByVal PortNum&, ByVal PortVal%) As Long`

Delphi: `function cbOutWord (BoardNum:Integer; PortNum:Integer; PortVal:Integer):Integer;`

Arguments:

`BoardNum` Refers to the board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).

`PortNum` A register within the board. Boards are set to a particular base address. The registers on the boards are at addresses that are offsets from the base address of the board (BaseAdr + 0, BaseAdr + 2, etc).

Set this argument to the offset for the desired register. This function takes care of adding the base address to the offset, so that the board's address can be changed without changing the code.

`PortVal` The value that is written to the register.

Returns:

Error code or 0 if no errors

Notes:

`cbOutByte()` is used to write to 8-bit ports. `cbOutWord()` is used to write to 16-bit ports.

This function was designed for use with ISA bus boards, and is not recommended for use with PCI bus boards.

cbRS485()

Sets the direction of RS-485 communications port buffers.

Function prototype:

C/C++:	`int cbRS485(int BoardNum, int Transmit, int Receive)`
Visual Basic:	`Function cbRS485(ByVal BoardNum&, ByVal Transmit&, ByVal Receive&) As Long`
Delphi:	`function cbRS485(BoardNum:Integer; Transmit:Integer; Receive:Integer):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`Transmit`	Set to `ENABLED` or `DISABLED` (CBENABLED or CBDISABLED in Visual Basic or Delphi). The transmit RS-485 line driver is turned on. Data written to the RS-485 UART chip is transmitted to the cable connected to that port.
`Receive`	Set to `ENABLED` or `DISABLED` (CBENABLED or CBDISABLED in Visual Basic or Delphi). The receive RS-485 buffer is turned on. Data present on the cable connected to the RS-485 port is received by the UART chip.

Returns:

Error code or 0 if no errors

Notes:

You can simultaneously enable or disable the transmit and receive buffers. If both are enabled, data written to the port is also received by the port. For a complete discussion of RS485 network construction and communication, refer to the CIO-COM485 or PCM-COM485 hardware manual.

cbStopBackground()

Stops one or more subsystem background operations that are in progress for the specified board. use this function to stop any function that is running in the background. This includes any function that was started with the `BACKGROUND` option, as well as `cbCStoreOnInt()` (which always runs in the background).

Execute `cbStopBackground()` after normal termination of all background functions to clear variables and flags.

Function prototype:

C/C++:	`int cbStopBackground(int BoardNum, int FunctionType)`
Visual Basic:	`Function cbStopBackground(ByVal BoardNum&, ByVal FunctionType&) As Long`
Delphi:	`function cbStopBackground(BoardNum:Integer, FunctionType:Integer):Integer;`

Arguments:

`BoardNum`	The board number associated with the board when it was installed with the configuration program. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`FunctionType`	Specifies which background operation to stop. Set it to one of the constants in the "FunctionType argument values" section below.

Returns:

Error code or 0 if no errors

FunctionType argument values:

`AIFUNCTION`	Specifies analog input scans started with `cbAInScan()` or `cbAPretrig()`
`AOFUNCION`	Specifies analog output scans started with `cbAOutScan()`.
`DIFUNCTION`	Specifies digital input scans started with `cbDInScan()`.
`DOFUNCTION`	Specifies digital output scans started with `cbDOutScan()`.
`CTRFUNCTION`	Specifies counter background operations started with `cbCStoreOnInt()`.

cbToEngUnits()

Converts an A/D count value to an equivalent voltage value.

Function prototype:

C/C++:	`int cbToEngUnits (int BoardNum, int Range, unsigned short DataVal, float *EngUnits)`
Visual Basic:	`Function cbToEngUnits(ByVal BoardNum&, ByVal Range&, ByVal DataVal%, EngUnits!) As Long`
Delphi:	`function cbToEngUnits (BoardNum:Integer; Range:Integer; DataVal:Word; var EngUnits:Single):Integer;`

Arguments:

`BoardNum`	The board number associated with the A/D board when it was installed. This function uses the board number to determine the range and resolution values to use for the conversion. `BoardNum` may be 0 to 99 (0 to 9 for 16-bit version of Universal Library).
`Range`	A/D voltage (or current) range. Some A/D boards have programmable voltage ranges, others set the voltage range via switches on the board. In either case, the selected range must be passed to this function. Each A/D board supports different voltage and/or current ranges. Refer to board specific information for a list of the supported A/D ranges of each board.
`DataVal`	A/D count returned from an A/D board.
`EngUnits`	The voltage (or current) value that is equivalent to `DataVal` is returned to this variable. The value will be within the range specified by the `Range` argument.

Returns:

Error code or 0 if no errors.

`EngUnits` – the engineering units value equivalent to `DataVal` is returned to this variable.

Universal Library for .NET Classes, Methods, and Properties

UL for .NET Class Library Overview

The new Microsoft .NET platform provides a framework that allows for the development of Windows applications using a wide range of new programming languages. These languages include VB .NET:, C#, managed C++, JScript, and any other language that is compliant with the .NET Common Language Runtime (CLR). The CLR is a multi-language execution environment.

The interface to the Universal Library consists of standard "C" functions. These functions are not CLR-compliant. Therefore, the Universal Library for .NET was developed. This library enables the various .NET programming languages to call into the Universal Library.

The Universal Library for .NET consists of a set of classes. For the most part, the methods within each class have a corresponding function in the standard UL. Each UL for .NET method has virtually the same parameter set as their UL counterparts.

MccDaq namespace

The MccDaq namespace contains the classes and enumerated constants by which your UL for .NET applications can access the Universal Library data types and functions.

MccDaq classes

The MccDaq namespace contains four main classes:

- MccBoard class
- ErrorInfo class
- MccService class
- GlobalConfig class

The MccDaq namespace also contains the following four secondary classes:

cBoardConfig	Contains all of the members for setting and getting board-level configuration.
cCtrConfig	Contains all of the members for setting and getting the counter-level configuration of a board.
cDioConfig	Contains all of the members for getting the digital configuration of a board.
cExpansionConfig	Contains all of the members for setting and getting expansion board configuration.

These classes include methods that are accessible from properties of the MccBoard class (explained below).

MccBoard class

The MccBoard class provides access to all of the methods for data acquisition and properties providing board information and configuration for a particular board.

The MccBoard class is a member of the MccDaq namespace. Refer to the "MccDaq namespace" above for an explanation of the MccDaq namespace.

Class constructors:

The MccBoard class provides two constructors; one which accepts a board number argument and one with no arguments.

The following code examples demonstrate how to create a new instance of the MccBoard class using the latter version with a default board number of 0.

VB .NET:

```
Private DaqBoard As MccDaq.MccBoard
DaqBoard = New MccDaq.MccBoard()
```

C# .NET:

```
private MccDaq.MccBoard DaqBoard;
DaqBoard = new MccDaq.MccBoard();
```

The following code examples demonstrate how to create a new instance of the MccBoard class with the board number passed to it.

VB .NET:

```
Private DaqBoard As MccDaq.MccBoard
DaqBoard = New MccDaq.MccBoard(BoardNumber)
```

C# .NET:

```
private MccDaq.MccBoard DaqBoard;
DaqBoard = new MccDaq.MccBoard(BoardNumber);
```

Properties and methods

The MccBoard class includes close to 100 methods for data acquisition. The MccBoard class methods are equivalents of the function calls used in the standard Universal Library. The MccBoard class methods have virtually the same parameter set as their UL counterparts.

The MccBoard class also includes six properties that you can use to examine or change the configuration of your board. The configuration information for all boards is stored in the CB.CFG file, and is loaded from CB.CFG by all programs that use the library.

Each MccBoard property and method is explained briefly later in this chapter, and in detail in the remaining chapters of the reference manual.

ErrorInfo class

Contains all of the members for storing and reporting error codes and messages. This class also includes error code enumerated constants, which define the error number and associated message which can be returned when you call a method.

Most UL for .NET methods return ErrorInfo objects. Error information is stored internally on the return from calling the low-level UL function. The error is reported when the user calls the class library methods.

The ErrorInfo class is a member of the MccDaq namespace. Refer to the "MccDaq namespace" section on page 139 for an explanation of the MccDaq namespace.

Enumerated constants

`ErrorCode`	Lists the named constants for all error codes. For a full explanation of the error associated with each error code and error constant, refer to the "Error Codes" appendix on page 301.

Properties and methods

The ErrorInfo class also includes two properties and close to 100 methods that you can use to examine error information. Each property and method is explained briefly later in this chapter, and in detail in the remaining chapters of the reference manual.

MccService class

Contains all of the members for calling utility UL functions.

The MccService class is a member of the MccDaq namespace. Refer to the "MccDaq namespace" on page 139 for an explanation of the MccDaq namespace.

Methods

The MccService class contains nine static methods. You do not need to create an instance of the MccService class to call these methods.

GlobalConfig class

Contains all of the members for getting global board configuration information.

The GlobalConfig class is a member of the MccDaq namespace. Refer to the "MccDaq namespace" on page 139 for an explanation of the MccDaq namespace.

Properties and methods

The GlobalConfig class also includes three properties that you can use to examine global board configuration information. Each property is explained briefly later in this chapter, and in detail in the remaining chapters of the reference manual.

Analog I/O methods

The analog I/O methods available from the MccBoard class are explained below. These methods perform analog input and output and convert analog data.

- **MccBoard.AIn()** - Takes a single reading from an analog input channel (A/D).
- **MccBoard.AInScan()** - Repeatedly scans a range of analog input (A/D) channels. You can specify the channel range, the number of iterations, the sampling rate, and the A/D range. The data that is collected is stored in an array.
- **MccBoard.ALoadQueue()** - Loads a series of chan/gain pairs into an A/D board's queue. These chan/gains are used with all subsequent analog input methods.
- **MccBoard.AOut()** - Outputs a single value to an analog output (D/A).
- **MccBoard.AOutScan()** - Repeatedly scans a range of analog output (D/A) channels. You can specify the channel range, the number of iterations, and the rate. The data from consecutive elements of an array are sent to each D/A channel in the scan.
- **MccBoard.APretrig()** - Repeatedly scans a range of analog input (A/D) channels waiting for a trigger signal. When a trigger occurs, it returns the specified number of samples and points before the trigger occurred. You can specify the channel range, the sampling rate, and the A/D range. All of the data that is collected is stored in an array.
- **MccBoard.ATrig()** - Reads analog input and waits until it goes above or below a specified threshold. When the trigger condition is met, the current sample is returned.

- **MccBoard.AConvertData()** - Converts analog data from data plus channel tags to separate data and channel tags.

  Each raw sample from analog input is a 16-bit value. On some 12-bit A/D boards it consists of a 12-bit A/D value along with a four bit channel number. This method is not intended for use with 16-bit A/D boards.

  This conversion is done automatically by the `MccBoard.AIn()` method. It can also be done automatically by the `MccBoard.AInScan()` method with the `ConvertData` option. In some cases though, it may be useful or necessary to collect the data and then do the conversion sometime later. The `MccBoard.AConvertData()` method takes a buffer full of unconverted data and converts it.

- **MccBoard.ACalibrateData()** - Calibrates analog data.

  Each raw sample from a board with software calibration factors that must be applied to the sample may be acquired and calibrated, then passed to an array. Alternatively, they can be acquired then passed to the array without calibration. This technique applies the calibration factors to an array of data after the acquisition is complete. When this second technique is used, `ACalibrateData()` may be used to apply the calibration factors to an array of data after the acquisition is complete. The only case where you would withhold calibration until after the acquisition run was complete is on slower CPUs, or when the processing time is at a premium. Applying calibration factors in real time on a per sample basis does eat up machine cycles.

  To disable the automatic calibration so that you may apply the calibration later, specify the `NoCalibrateData` option when collecting data with the `MccBoard.AInScan()` method.

- **MccBoard.AConvertPretrigData()** - Converts and re-orders pre-trigger data from data plus channel tags to separate data and channel tags.

  When data is collected with the `MccBoard.APretrig()` method, the same data conversion needs to be done as is performed by the `MccBoard.AConvertData()` method. There is a further complication because MccBoard.APretrig() collects analog data into an array. It treats the array like a circular buffer. While it is waiting for the trigger to occur, it fills the array. When it gets to the end it resets to the start and begins again. When the trigger signal occurs it continues collecting data into the circular buffer until the requested number of samples have been collected.

  When the data acquisition is complete, all of the data is in the array but it is in the wrong order. The first element of the array does not contain the first data point. The data has to be rotated in the correct order.

  This conversion can be done automatically by the `MccBoard.APretrig()` method with the `ConvertData` option. In some cases though, it may be useful or necessary to collect the data and then do the conversion sometime later. The `MccBoard.AConvertPretrigData()` method takes a buffer full of unconverted data and converts it.

Configuration methods and properties

The configuration methods and properties available from the `MccBoard` class, `cBoardConfig` class, `cCtrConfig` class, `cDioConfig` class, and the `cExpansionConfig` class are explained below.

The configuration information for all boards is stored in the configuration file CB.CFG. This information is loaded from CB.CFG by all programs that use the library. The library includes the following classes and methods that retrieve or change configuration options.

- **MccBoard.BoardNum** property - Number of the board associated with an instance of the `MccBoard` class.

- **MccBoard.GetSignal()** - Retrieves the configured auxiliary or DAQ Sync connection and polarity for the specified timing and control signal. This method is intended for advanced users.

- **MccBoard.SelectSignal()** - Configures timing and control signals to use specific auxiliary or DAQ Sync connections as a source or destination. This method is intended for advanced users.
- **MccBoard.SetTrigger()** - Sets up trigger parameters used with the `ExtTrigger` option for `MccBoard.AInScan()`.
- **MccBoard.BoardConfig** property - Gets an instance of a cBoardConfig object.
- **MccBoard.BoardConfig.DACUpdate()** - Updates the voltage values on analog output channels.
- **MccBoard.BoardConfig.GetBaseAdr()** - Gets the base address of a board.
- **MccBoard.BoardConfig.GetBoardType()** - Gets the unique number (device ID) assigned to the board (between 0 and 8000h) indicating the type of board installed.
- **MccBoard.BoardConfig.GetCiNumDevs()** - Gets the number of counter devices on the board.
- **MccBoard.BoardConfig.GetDACStartup()** - Gets the board's configuration register STARTUP bit setting.
- **MccBoard.BoardConfig.GetDACUpdateMode()** - Returns the update mode for a digital-to-analog converter (DAC).
- **MccBoard.BoardConfig.GetClock()** - Gets the clock frequency in MHz (40, 10, 8, 6, 5, 4, 3, 2, 1), or 0 for not supported.
- **MccBoard.BoardConfig.GetDInMask()** - Determines the bits on a specified port that are configured for input.
- **MccBoard.BoardConfig.GetDiNumDevs()** - Gets the number of digital devices on the board.
- **MccBoard.BoardConfig.GetDmaChan()** - Gets the DMA channel (0, 1 or 3) set for the board.
- **MccBoard.BoardConfig.GetDOutMask()** - Determines the bits on a specified port that are configured for output.
- **MccBoard.BoardConfig.GetDtBoard()** - Gets the number of the board with the DT connector used to connect to external memory boards.
- **MccBoard.BoardConfig.GetIntLevel()** - Gets the interrupt level set for the board (0 for none, or 1 to 15).
- **MccBoard.BoardConfig.GetNumAdChans()** - Gets the number of A/D channels
- **MccBoard.BoardConfig.GetNumDaChans()** - Gets the number of D/A channels.
- **MccBoard.BoardConfig.GetNumExps()** - Gets the number of expansion boards.
- **MccBoard.BoardConfig.GetNumIoPorts()** - Gets the number of I/O ports used by the board.
- **MccBoard.BoardConfig.GetRange()** - Gets the selected voltage range.
- **MccBoard.BoardConfig.GetUsesExps()** - Gets the True/False value indicating support of expansion boards.
- **MccBoard.BoardConfig.GetWaitState()** - Gets the value of the Wait State jumper (1-enabled, 0-disabled).
- **MccBoard.BoardConfig.SetBaseAdr()** - Sets the base address of a board
- **MccBoard.BoardConfig.SetClock()** - .Sets the clock source by the frequency (40, 10, 8, 6, 5, 4, 3, 2, 1), or 0 for not supported.
- **MccBoard.BoardConfig.SetDACStartup()** - Sets the board's configuration register STARTUP bit to 0 or 1 to enable/disable the storing of digital-to-analog converter (DAC) startup values.

- **MccBoard.BoardConfig.SetDACUpdateMode()** - Sets the update mode for a digital-to-analog converter (DAC).
- **MccBoard.BoardConfig.SetDmaChan()** - Sets the DMA channel (0, 1 or 3).
- **MccBoard.BoardConfig.SetIntLevel()** - Sets the interrupt level: 0 for none, or 1 to 15.
- **MccBoard.BoardConfig.SetNumAdChans()** - Sets the number of A/D channels available on the board.
- **MccBoard.BoardConfig.SetRange()** - Sets the selected voltage range.
- **MccBoard.BoardConfig.SetWaitState()** - Sets the value of the Wait State jumper (1 = enabled, 0 = disabled).
- **MccBoard.CtrConfig** property - Gets an instance of a cCtrConfig object.
- **MccBoard.CtrConfig.GetCtrType()** - Gets the counter device number of counter type specified with the `configVal` parameter.
- **MccBoard.DioConfig** property - Gets an instance of a cDioConfig object.
- **MccBoard.DioConfig.GetConfig()** - Gets the configuration of a digital device (digital input or digital output).
- **MccBoard.DioConfig.GetCurVal()** - Gets the current value of digital outputs.
- **MccBoard.DioConfig.GetDevType()** - Gets the device type of the digital port (AUXPORT, FIRSTPORTA, etc.).
- **MccBoard.DioConfig.GetDInMask()** - Determines the bits on a specified port that are configured for input.
- **MccBoard.DioConfig.GetDOutMask()** - Determines the bits on a specified port that are configured for output.
- **MccBoard.DioConfig.GetNumBits()** - Gets the number of bits in the digital port value.
- **MccBoard.ExpansionConfig** property - Gets an instance of a cExpansionConfig object.
- **MccBoard.ExpansionConfig.GetBoardType()** - Gets the expansion board type.
- **MccBoard.ExpansionConfig.GetCjcChan()** - Gets the channel that the CJC is connected to.
- **MccBoard.ExpansionConfig.GetMuxAdChan1()** - Gets the first A/D channel that the board is connected to.
- **MccBoard.ExpansionConfig.GetMuxAdChan2()** - Gets the second A/D channel that the board is connected to.
- **MccBoard.ExpansionConfig.GetNumExpChans()** - Gets the number of expansion board channels.
- **MccBoard.ExpansionConfig.GetRange1()** - Gets the range/gain of the low 16 channels.
- **MccBoard.ExpansionConfig.GetRange2()** - Gets the range/gain of the high 16 channels.
- **MccBoard.ExpansionConfig.GetThermType()** - Gets the type of thermocouple configuration for the board (J, K, E, T, R, S, and B types).
- **MccBoard.ExpansionConfig.SetCjcChan()** - Sets the channel that the CJC is connected to.
- **MccBoard.ExpansionConfig.SetMuxAdChan1()** - Sets the first A/D channel that the board is connected to.
- **MccBoard.ExpansionConfig.SetMuxAdChan2()** - Sets the second A/D channel that the board is connected to.

- **MccBoard.ExpansionConfig.SetRange1()** - Sets the range/gain of the low 16 channels.
- **MccBoard.ExpansionConfig.SetRange2()** - Sets the range/gain of the high 16 channels.
- **MccBoard.ExpansionConfig.SetThermType()** - Sets the type of thermocouple configuration for the board (J, K, E, T, R, S, and B types).
- **GlobalConfig.NumBoards** property - Returns the maximum number of boards you can install at one time.
- **GlobalConfig.NumExpBoards** property- Returns the maximum number of expansion boards you can install on a board.
- **GlobalConfig.Version** property - Information used by the library to determine compatibility.

Counter methods

The counter functions available from the `MccBoard class` are explained below. These methods load, read, and configure counters. There are five types of counter chips used in MCC counter boards: 8254's, 8536's, 7266's, 9513's, and generic event counters. Some of the counter commands only apply to one type of counter.

- **MccBoard.C7266Config()** - Selects the basic operating mode of an LS7266 counter.
- **MccBoard.C8254Config()** - Selects the basic operating mode of an 8254 counter.
- **MccBoard.C8536Config()** - Selects the basic operating mode of an 8536 counter chip.
- **MccBoard.C8536Init()** - Initializes and selects all of the chip level features for a 8536 counter board. The options that are set by this command are associated with each counter chip, not the individual counters within it.
- **MccBoard.C9513Config()** - Sets the basic operating mode of a 9513 counter. This method sets all of the programmable options that are associated with a 9513 counter. It is similar in purpose to C8254Config() except that it is used with a 9513 counter.
- **MccBoard.C9513Init()** - Initializes and selects all of the chip level features for a 9513 counter board. The options that are set by this command are associated with each counter chip, not the individual counters within it.
- **MccBoard.CFreqIn()** - Measures the frequency of a signal by counting it for a specified period of time (`GatingInterval`), and then converting the count to count/sec (Hz). Works only with 9513 counters.
- **MccBoard.CIn()** - Reads a counter's current value.
- **MccBoard.CIn32()** - Reads a counter's current value as a 32-bit integer. Used primarily with LS7266 counters.
- **MccBoard.CLoad()** - Loads a counter with an initial count value.
- **MccBoard.CLoad32()** - Loads a counter with a 32-bit integer initial value. Used primarily with LS7266 counters.
- **MccBoard.CStatus()** - Read the counter status of a counter. Returns various bits that indicate the current state of a counter (currently only applies to LS7266 counters).
- **MccBoard.CStoreOnInt()** - Installs an interrupt handler that stores the current count whenever an interrupt occurs. This method only works with 9513 counters.

Digital I/O methods

The digital methods available from the MccBoard class are explained below. These methods perform digital input and output on various types of digital I/O ports.

- **MccBoard.DBitIn()** - Reads a single bit from a digital input port.
- **cbMccBoard.DBitOut()** - Sets a single bit on a digital output port.
- **MccBoard.DConfigBit()** - Configures a specific digital bit as input or output.
- **MccBoard.DConfigPort()** - Selects whether a digital port is an input or an output.
- **MccBoard.DIn()** - Reads a specified digital input port.
- **MccBoard.DInScan()** - Reads a set number of bytes or words from a digital input port at a specific rate.
- **MccBoard.DOut()** - Writes a byte to a digital output port.
- **MccBoard.DOutScan()** - Writes a series of bytes or words to a digital output port at a specified rate.

Error Handling methods and properties

Most UL for .NET methods return ErrorInfo objects. The `MccService` class includes one method that determines how errors are handled internally by the library. The `ErrorInfo` class includes two properties that provide information returned by the method called.

- **MccService.ErrHandling()** - Sets the manner of reporting and handling errors for all method calls.
- **ErrorInfo.Message** property - Gets the text of the error message associated with a specific error code.
- **ErrorInfo.Value** property - Gets the error constant associated with an `ErrorInfo` object.

Memory board methods

The memory board methods available from the `MccBoard class` read and write data to and from a memory board, and also set modes that control memory boards (MEGA-FIFO).

The most common use for memory boards is to store large amounts of data from an A/D board via a DT-Connect cable between the two boards. To do this, use the `ExtMemory` option with the `MccBoard.AInScan()` or `MccBoard.APretrig()` methods.

Once the data has been transferred to the memory board you can use the memory methods to retrieve the data.

- **MccBoard.MemSetDTMode()** - Set DT-Connect mode on a memory board. Memory boards have a DT-Connect interface which can be used to transfer data through a cable between two boards rather than through the PC's system memory. The DT-Connect port on the memory board can be configured as either an input (from an A/D) or as an output (to a D/A). This method configures the port.
- **MccBoard.MemReset()** - Resets the memory board address. The memory board is organized as a sequential device. When data is transferred to the memory board it is automatically put in the next address location. This method resets the current address to the location 0.
- **MccBoard.MemRead()** - Reads a specified number of points from a memory board starting at a specified address.
- **MccBoard.MemWrite()** - Writes a specified number of points to a memory board starting at a specified address.

- **MccBoard.MemReadPretrig()** - Reads data collected with MccBoard.APretrig(). The MccBoard.APretrig() method writes the pre-triggered data to the memory board in a scrambled order. This method unscrambles the data and returns it in the correct order.

Revision control methods and properties

The revision control methods and property explained below are available from the MccBoard class.

As new revisions of the library are released, bugs from previous revisions are fixed, and occasionally new functions are added. It is Measurement Computing's goal to preserve the programs you have written so that you never change the order or number of arguments in a method. However, sometimes it is not possible to achieve this goal.

The revision control methods initialize the DLL so that the functions are interpreted according to the format of the revision you wrote and compiled your program in.

- **MccBoard.DeclareRevision()** - Declares the revision number of the Universal Library for .NET that your program was written with.
- **MccBoard.GetRevision()** - Returns the version number of the installed Universal Library for .NET.

Streamer file methods

The streamer file methods available from the MccBoard class create, fill, and read streamer files.

- **MccBoard.FileAInScan()** - Transfer analog input data directly to file. Very similar to AInScan() except that the data is stored in a file instead of an array.
- **MccBoard.FilePretrig()** - Pre-triggered analog input to a file. Very similar to APretrig() except that the data is stored in a file instead of an array.
- **MccBoard.FileGetInfo()** - Reads streamer file information on how much data is in the file, and the conditions under which it was collected (sampling rate, channels, etc.).
- **MccBoard.FileRead()** - Reads a selected number of data points from a streamer file into an array.

Temperature input methods

The methods explained below convert a raw analog input from an EXP or other temperature sensor board to temperature.

- **MccBoard.TIn()** - Reads a channel from a digital input board, filters it (if specified), does the cold junction compensation, linearizes and converts it to temperature.
- **MccBoard.TInScan()** - Scans a range of temperature inputs. Reads temperatures from a range of channels and returns the temperature values to an array.

Windows memory management methods

The Windows memory management methods available from the MccService class take care of allocating, freeing, and copying to/from Windows global memory buffers.

- **MccService.WinBufAlloc()** - Allocate a Windows memory buffer.
- **MccService.WinBufFree()** - Free a Windows buffer.
- **MccService.WinArrayToBuf()** - Copies data from an array to a Windows buffer.

- **MccService.WinBufToArray()** - Copies data from a Windows buffer to an array.

Miscellaneous methods, properties, and delegates

The methods explained below are available from the MccBoard class. These functions do not as a group fit into a single category. They get and set board information, convert units, manage events and background operations, and perform serial communication operations.

- **MccBoard.GetStatus()** - Returns the status of a background operation.

  Once a background operation starts, your program must periodically check on its progress. This method returns the current status of the operation.

- **MccBoard.StopBackground()** - Stop a background process.

  It is sometimes necessary to stop a background process even though the process has been set up to run continuously. This method stops a background process that is running. `StopBackground()` should be executed after normal termination of all background functions in order to clear variables and flags.

- **MccBoard.EnableEvent()** - Binds one or more event conditions to a user-defined callback function.
- **MccBoard.DisableEvent()** - - Disables one or more events set up with EnableEvent() and disconnects their user-defined handlers.
- **EventCallback** delegate – Defines the prototype for the user function for `EnableEvent()`. This defines the format for the user-defined handlers to be called when the events set up using `EnableEvent()` occurs.
- **MccBoard.InByte()** - Reads a byte from a hardware register on a board.
- **MccBoard.InWord()** - Reads a word from a hardware register on a board.
- **MccBoard.OutByte()** - Writes a byte to a hardware register on a board.
- **MccBoard.OutWord()** - Writes a byte or word to a hardware register on a board.
- **MccBoard.GetBoardName()** - Returns the name of a specified board.
- **MccBoard.RS485()** - Sets the transmit and receive buffers on an RS485 port.
- **MccBoard.ToEngUnits()** - Converts a count value from an A/D to voltage (or current).
- **MccBoard.FromEngUnits()** - Converts a voltage (or current) to a D/A count value.
- **MccBoard.BoardName** property - Name of the board associated with an instance of the MccBoard class.

Universal Library for .NET example programs

The Universal Library for .NET contains many example programs to help you learn and apply UL for .NET methods. We strongly recommend running appropriate example programs before attempting to use the methods.

Table 13-1 lists the UL for .NET example programs sorted by program name. It includes their featured method calls, special aspects, and other method calls included in the program. All example programs include the `DeclareRevision()` and `ErrHandling()` methods. Table 13-2 lists the UL for .NET example programs sorted by the method name.

Table 13-1. UL for .NET Example Programs – Sorted by Program Name

Program name	Featured UL for .NET method call	Notes	Other UL for .NET method calls
ULAI01	`AIn()`		`ToEngUnits()`

ULAI02	`AInScan()`	`Default` mode	`WinBufToArray()`
			`WinBufFree()`
			`WinBufAlloc()`

ULAI03	`AInScan()`	`Background` mode	`GetStatus()`
			`StopBackground()`
			`WinBufToArray()`
			`WinBufFree()`
			`WinBufAlloc()`

ULAI04	`AConvertData()`		`AInScan()`
			`GetStatus()`
			`StopBackground()`
			`WinBufToArray()`
			`WinBufFree()`
			`WinBufAlloc()`

ULAI05	`AInScan()`	with manual data conversion	`GetStatus()`
			`StopBackground()`
			`WinBufToArray()`
			`WinBufFree()`
			`WinBufAlloc()`

ULAI06	`AInScan()`	`Continuous Background` mode	`AConvertData()`
			`GetStatus()`
			`StopBackground()`
			`WinBufToArray()`
			`WinBufFree()`
			`WinBufAlloc()`

ULAI07	`ATrig()`		`FromEngUnits()`

ULAI08	`APretrig()`		`WinBufToArray()`
			`WinBufFree()`
			`WinBufAlloc()`

ULAI09	`ConvertPretrigData()`	`Background`	`APretrig()`
			`GetStatus()`
			`StopBackground()`
			`WinBufToArray()`
			`WinBufFree()`
			`WinBufAlloc()`

ULAI10	`cbALoadQueue()`		`AInScan()`
			`WinBufToArray()`
			`WinBufFree()`
			`WinBufAlloc()`

ULAI11	`cbToEngUnits()`		`AIn()`

ULAI12	`cbAInScan()`	`ExtClock` mode	`WinBufToArray()`
			`WinBufFree()`
			`WinBufAlloc()`

ULAI13	`cbAInScan()`	Various sampling mode options	`WinBufToArray()`
			`WinBufFree()`
			`WinBufAlloc()`

Program name	Featured UL for .NET method call	Notes	Other UL for .NET method calls
ULAI14	SetTrigger()	With ExtTrigger selected	AInScan() FromEngUnits() WinBufToArray() WinBufFree() WinBufAlloc()
ULAIO01	AInScan() AOutScan()	Concurrent analog input and analog output scans	GetStatus() StopBackground() WinArrayToBuf() WinBufAlloc() WinBufFree() WinBufToArray()
ULAO01	AOut()		FromEngUnits() AOut()
ULAO02	AOutScan()		WinBufToArray() WinBufFree() WinBufAlloc()
ULAO03	AOut() DACUpdate() SetDACUpdateMode()	Demonstrates the difference between BoardConfig.DACUpdate.Immediate and BoardConfig.DACUpdate.OnCommand D/A update modes. Board 0 must support DAC update mode settings, such as the PCI-DAC6700 Series boards.	FromEngUnits()
ULCT01	C8254Config()		CLoad(), CIn()
ULCT02	C9513Init() C9513Config()		CLoad(), CIn()
ULCT03	CStoreOnInt()		C9513Init(), CLoad() C9513Config(), CIn()
ULCT04	CFreqIn()		C9513Init()
ULCT05	C8536Init() C8536Config()		CLoad() CIn()
ULCT06	C7266Config()		CLoad32(), CIn32() CStatus()
ULDI01	DIn()		DConfigPort()
ULDI02	DBitIn()		DConfigPort()
ULDI03	DInScan()		DConfigPort() GetStatus() StopBackground() WinBufToArray() WinBufFree() WinBufAlloc()
ULDI04	DIn()	Using the AuxPort	DioConfig() DConfigPort()
ULDI05	DBitIn()	Using the AuxPort	DioConfig() DConfigPort()
ULDI06	DConfigBit()		DBitIn() DioConfig() DConfigPort()
ULDO01	DOut()		DConfigPort()
ULDO02	DBitOut()		DOut(), DConfigPort()

Program name	Featured UL for .NET method call	Notes	Other UL for .NET method calls
ULDO04	`DOut()`	Using the `AuxPort`	`DioConfig()` `DConfigPort()`

ULDO05	`DBitOut()`	Using the `AuxPort`	`DOut()`
			`DioConfig()`
			`DConfigPort()`

ULEV01	`EnableEvent()` `DisableEvent()`	Using `OnExternalInterrupt`	`DConfigPort()` `DIn()`

ULEV02	`EnableEvent()`
	`DisableEvent()`
		Using `OnDataAvailable` and `OnEndOfAiScan`	`AInScan()`
			`StopBackground()`
			`ToEngUnits()`
			`WinBufAlloc()`
			`WinBufFree()`
			`WinBufToArray()`

ULEV03	`EnableEvent()`
	`DisableEvent()`
		Using `OnPretrig` and `OnEndOfAiScan`	`APretrig()`
			`AConvertPretrigData()`
			`DConfigPort()`
			`DOut()`
			`StopBackground()`
			`ToEngUnits()`
			`WinBufAlloc()`
			`WinBufFree()`
			`WinBufToArray()`

ULEV04	`EnableEvent()`
	`DisableEvent()`
		`Using OnEndOfAoScan`	`AOutScan()`
			`DConfigPort()`
			`DOut()`
			`FromEngUnits()`
			`StopBackground()`
			`WinArrayToBuf()`
			`WinBufAlloc()`
			`WinBufFree()`

ULFI01	`FileAInScan()`		`FileGetInfo()`
ULFI02	`FileRead()`		`FileAInScan()` `FileGetInfo()`
ULFI03	`FilePretrig()`		`FileGetInfo()` `FileRead()`
ULGT01	`GetErrMsg()`		`AIn()`
ULGT03	`MccDaq().MccBoard()` class() properties: BoardConfig, DioConfig and ExpansionConfig	Use the MccBoard class properties to get configuration information for a board.	`GetBoardName()`
ULGT04	`GetBoardName()`		`MccDaq.MccBoard.BoardName` property `MccDaq.GlobalConfig.NumBoards` property
ULMM01	`MemReadPretrig()`		`APretrig()`
ULMM02	`MemRead()` `MemWrite()`		
ULMM03	`AInScan()`	With `ExtMemory` option	`MemReset()` `MemRead()`
ULTI01	`TIn()`		

Program name	Featured UL for .NET method call	Notes	Other UL for .NET method calls
ULTI02	`TInScan()`		

Table 13-2. UL for .NET Example Programs – Sorted by Method Name

UL for .NET method call	UL for .NET example program Name	UL for .NET special features/notes
`AConvertData()`	ULAI04 ULAI06	
`AConvertPretrigData()`	ULAI09 ULEV03*	
`ACalibrateData()`	None	No example programs at this time
`AIn()`	ULAI01 ULGT01 ULAI11	
`AInScan()`	ULAI02 ULAI10 ULAI03 ULAI12 ULAI04 ULAI13 ULAI05 ULAI14 ULAI06 ULMM03 ULEV02*	`Default`, `Background` mode with manual data conversion `Continuous Background` mode `ExtClock` mode Various sampling mode options
`ALoadQueue()`	ULAI10	
`AOut()`	ULAO01 ULAO03	Demonstrates the difference between `BoardConfig.DACUpdate.Immediate` and `BoardConfig.DACUpdate.OnCommand` D/A update modes. Board 0 must support DAC update mode settings, such as the PCI-DAC6700 Series boards.
`AOutScan()`	ULAO02 ULAIO01 ULEV04*	Concurrent `AInScan()` and `AOutScan()`
`APretrig()`	ULAI08 ULFI03 ULAI09 ULMM01 ULEV03*	
`ATrig()`	ULAI07 ULMM01	
`C7266Config()`	ULCT06	
`C8254Config()`	ULCT01	
`C8536Config()`	ULCT05	
`C8536Init()`	ULCT05	
`C9513Config()`	ULCT02 ULCT03	
`C9513Init()`	ULCT02 ULCT04 ULCT03	
`CFreqIn()`	ULCT04	
`CIn()`	ULCT01 ULCT05 ULCT02	
`CIn32()`	ULCT06	
`CLoad()`	ULCT01 ULCT03 ULCT02 ULCT05	
`CLoad32()`	ULCT06	
`CStoreOnInt()`	ULCT03	
`CStatus()`	ULCT06	
`DBitIn()`	ULDI02 ULDI06 ULDI05	

UL for .NET method call	UL for .NET example program Name	UL for .NET special features/notes
`DBitOut()`	ULDO02 ULDO05	
`DConfigBit()`	ULDI06	
`DConfigPort()`	ULDI01 ULDO01 ULDI02 ULDO02 ULDI03 ULDO05 ULEV01* ULEV03* ULEV04*	
`DIn()`	ULDI01 ULDI04 ULDI03 ULEV01*	
`DInScan()`	ULDI03	
`DOut()`	ULDO01 ULDO05 ULDO02 ULDO04 ULEV03* ULEV04*	
`DOutScan()`	None	No example programs at this time

`EnableEvents()`
`DisableEvents()`
	ULEV01* ULEV03*
	ULEV02* ULEV04*
		`OnExternalInterrupt`
		`OnDataAvailable`
		`OnPretrigger`
		`OnEndOfAoScan`
		`OnScanError`
		`OnEndOfAiScan`

`ErrHandling()`	All Samples	All sample programs use this method
`FileAInScan()`	ULFI01 ULFI02	
`FilePretrig()`	ULFI03	ULFI01 ULFI02
`FileRead()`	ULFI02 ULFI03	
`FlashLED()`	ULFI01	Flashes the onboard LED for visual identification (board 0 must have an external LED, such as the miniLAB 1008 or the USB-1208LS.
`FromEngUnits`	ULAO01 ULAO03 ULAI07 ULAI14 ULEV04	
`GetBoardName`	ULGT03 ULGT04	
`GetDACStartup()`	None	No sample programs at this time
`GetDACUpdateMode()`	None	No sample programs at this time
`GetErrMsg()`	ULGT01	
`GetRevision()`	None	No sample programs at this time
`GetStatus()`	ULAI03 ULAI06 ULAI04 ULAI09 ULAI05 ULCT03 ULAIO01 ULDI0	
`InByte()`	None	No example programs at this time
`InWord()`	None	No example programs at this time

UL for .NET method call	UL for .NET example program Name	UL for .NET special features/notes
`MccDaq.MccBoard` class properties: BoardConfig, DioConfig, and ExpansionConfig	ULGT03 ULGT04	Use the MccBoard class properties to get configuration information for a board.
`MemRead()`	ULMM01 ULMM03 ULMM02	
`MemReadPretrig()`	ULMM01	
`MemReset()`	ULMM03	
`MemSetDTMode()`	None	No example programs at this time
`MemWrite()`	ULMM02	
`RS485()`	None	No example programs at this time
`SetTrigger()`	ULAI14	
`StopBackground()`	ULAI03 ULAI06 ULAI04 ULAI09 ULAI05 ULCT03 ULAIO01 ULDI03 ULEV02* ULEV03* ULEV04*	Concurrent `AInScan()` and `AOutScan()`
`TIn()`	ULTI01	
`TInScan()`	ULTI02	
`ToEngUnits()`	ULAI01 ULAI11 ULAI07 ULEV02* ULEV03*	
`WinArrayToBuf()`	ULAIO01 ULAIO02 ULEV04*	

`WinBufAlloc()`
`WinBufFree()`
`WinBufToArray()`
	ULAI01 ULAI10
	ULAI02 ULAI12
	ULAI03 ULAI13
	ULAI04 ULAI14
	ULAI05
	ULAI06 ULAO02
	ULAI08 ULCT03
	ULAI09 ULDI03
	ULEV02* ULEV03*
	
	ULEV04*
	(`WinBufAlloc` and `WinBufFree` only)	

Analog I/O Methods

Introduction

The methods explained in this chapter handle analog input, analog output and analog data manipulation. These methods are available from the `MccBoard class`.

Most analog I/O methods include options that may not be compatible with your hardware. To determine which of these functions are compatible with your hardware, refer to the *Universal Library User's Guide (*available in PDF format on our website at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Table 14-1 lists the MccDaq.Range enumerated constants you can use in the `range` parameter found in most of the methods explained in this chapter. These values are also used in the `ALoadQueue()` method's `gainArray` parameter. Valid ranges for your hardware are listed in the *Universal Library User's Guide*.

Table 14-1. `MccDaq.Range` Enumerated Constants

UL .NET settings	Value	UL .NET settings	Value
`MccDaq.Bip20Volts`	±20 volts(V)	`MccDaq.Uni5Volts`	0 to 5 V
`MccDaq.Bip10Volts`	±10 V	`MccDaq.Uni2Pt5Volts`	0 to 2.5 V
`MccDaq.Bip5Volts`	±5 V	`MccDaq.Uni2Volts`	0 to 2 V
`MccDaq.Bip4Volts`	±4 V	`MccDaq.Uni1Pt25Volts`	0 to 1.25 V
`MccDaq.Bip2Pt5Volts`	±2.5 V	`MccDaq.Uni1Pt67Volts`	0 to 1.67 V
`MccDaq.Bip2Volts`	±2 V	`MccDaq.Uni1Volts`	0 to 1 V
`MccDaq.Bip1Pt25Volts`	±1.25 V	`MccDaq.UniPt5Volts`	0 to 0.5 V
`MccDaq.Bip1Volts`	±1 V	`MccDaq.UniPt25Volts`	0 to 0.25 V
`MccDaq.Bip1Pt67Volts`	±1.67 V	`MccDaq.UniPt2Volts`	0 to 0.2 V
`MccDaq.BipPt625Volts`	±0.625 V	`MccDaq.UniPt1Volts`	0 to 0.1 V
`MccDaq.BipPt5Volts`	±0.5 V	`MccDaq.UniPt01Volts`	0 to 0.01 V
`MccDaq.BipPt25Volts`	±0.25 V	`MccDaq.UniPt02Volts`	0 to 0.02 V
`MccDaq.BipPt2Volts`	±0.2 V	`MccDaq.UniPt05Volts`	0 to 0.05 V
`MccDaq.BipPt1Volts`	±0.1 V	`MccDaq.Ma0To20`	0 to 20 milliamperes (mA)
`MccDaq.BipPt05Volts`	±0.05 V	`MccDaq.Ma4To20`	4 to 20 mA
`MccDaq.BipPt01Volts`	±0.01 V	`MccDaq.Ma2To10`	2 to 10 mA
`MccDaq.BipPt005Volts`	±0.005 V	`MccDaq.Ma1To5`	1 to 5 mA
`MccDaq.Uni10Volts`	0 to 10 V	`MccDaq.MaPt5To2Pt5`	0.5 to 2.5 mA

AConvertData()

Converts the raw data collected by `AInScan()` into 12-bit A/D values. The `AInScan()` method can return either raw A/D data or converted data, depending on whether or not the `ConvertData()` option is used. For many 12-bit A/D boards, the raw data is a 16-bit value that contains a 12-bit A/D value and a 4-bit channel tag (refer to board specific-information). The converted data consists of just the 12-bit A/D value.

Member of the `MccBoard class`.

Function prototype:

VB .NET:

```
Public Function AConvertData(ByVal numPoints As Integer, ByRef
adData As Short, ByRef chanTags As Short) As MccDaq.ErrorInfo
```

```
Public Function AConvertData(ByVal numPoints As Integer, ByRef
adData As System.UInt16, ByRef chanTags As System.UInt16) As
MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo AConvertData(int numPoints, ref ushort
adData, out ushort chanTags)
```

```
public MccDaq.ErrorInfo AConvertData(int numPoints, ref short
adData, out short chanTags)
```

Parameters:

`numPoints`	Number of samples to convert
`adData`	Reference to start of data array
`chanTags`	Reference to start of channel tag array

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`adData` - converted data

`chanTags` - channel tags if available.

When collecting data using `AInScan()` without the `ConvertData` option, use this method to convert the data after it has been collected. There are cases where the `ConvertData` option is not allowed. For example - if you are using both the `DmaIo` and `Background` option with `AInScan()`. In those cases, use `AConvertData()` to convert the data after the data collection is complete.

For some boards, each raw data point consists of a 12-bit A/D value with a 4-bit channel number. This method pulls each data point apart and puts the A/D value into the `adData` array and the channel number into the `chanTags` array.

Notes:

12-bit A/D boards: The name of the array must match that used in `AInScan()` or `WinBufToArray()`. Upon returning from `AConvertData`(), `adData` array contains only 12-bit A/D data.

AConvertPretrigData()

Converts the raw data collected by `APretrig()`. The `APretrig()` method can return either raw A/D data or converted data, depending on whether or not the `ConvertData` option was used. The raw data is not in the correct order as it is collected. After the data collection is completed, it must be rearranged into the correct order. This method also orders the data, starting with the first pretrigger data point and ending with the last post-trigger point.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function AConvertPretrigData(ByVal preTrigCount As Integer,
ByVal totalCount As Integer, ByRef adData As Short, ByRef chanTags
As Short) As MccDaq.ErrorInfo
```

```
Public Function AConvertPretrigData(ByVal preTrigCount As Integer,
ByVal totalCount As Integer, ByRef adData As System.UInt16, ByRef
chanTags As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo AConvertPretrigData(int preTrigCount, int
totalCount, ref ushort adData, out ushort chanTags)
```

```
public MccDaq.ErrorInfo AConvertPretrigData(int preTrigCount, int
totalCount, ref short adData, out short chanTags)
```

Parameters:

`preTrigCount`	Number of pre-trigger samples (this value must match the value returned by the `PretrigCount` parameter in the `APretrig()` method)
`totalCount`	Total number of samples that were collected
`adData`	Reference to data array (must match array name used in `APretrig()` method)
`chanTags`	Reference to channel tag array or a NULL reference may be passed if using 16-bit boards or if channel tags are not desired (see the note regarding 16-bit boards below).

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`adData` - converted data

When you collect data with `APretrig()` and you don't use the `ConvertData` option, you must use this method to convert the data after it is collected. There are cases where the `ConvertData` option is not allowed: for example, if you use the `Background` option with `APretrig()`. In those cases, this method should be used to convert the data after the data collection is complete.

Notes:

12-Bit A/D Boards: On some 12-bit boards, each raw data point consists of a 12-bit A/D value with a 4-bit channel number. This method pulls each data point apart and puts the A/D value into the `adData` and the channel number into the `chanTags` array.

Upon returning from `AConvertPretrigData()`, `adData` array contains only 12-bit A/D data.

16-Bit A/D Boards: This method is for use with 16-bit A/D boards only insofar as ordering the data. No channel tags are returned.

Name of the `ADData` array must match that used in `AInScan()` or `WinBufToArray()`.

Visual Basic programmers:

After the data is collected with `APretrig()`, it must be copied to a BASIC array with `WinBufToArray()`.

Important

The entire array must be copied, which includes the extra 512 samples needed by `APretrig()`. Example code is given below.

```
SampleCount& = 10000
Dim A_D_Data% (SampleCount& + 512)
Dim Chan_Tags% (SampleCount& + 512)
APretrig%(LowChan, HighChan, PretrigCount&, SampleCount&...)
WinBufToArray%(MemHandle%, A_D_Data%, SampleCount& + 512)
AConvertPretrigData%(Pretrig_Count&, SampleCount&, A_D_Data%, Chan_Tags%)
```

ACalibrateData()

Calibrates the raw data collected by `AInScan()` from boards with real time software calibration when the real time calibration has been turned off. The `AInScan()` method can return either raw A/D data or calibrated data, depending on whether or not the `NoCalibrateData` option was used.

Member of the `MccBoard class`.

Function prototype:

VB .NET:

```
Public Function ACalibrateData(ByVal numPoints As Integer, ByVal range As MccDaq.Range, ByRef adData As Short) As MccDaq.ErrorInfo
```

```
Public Function ACalibrateData(ByVal numPoints As Integer, ByVal range As MccDaq.Range, ByRef adData As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo ACalibrateData(int numPoints, MccDaq.Range, ref ushort adData)
```

```
public MccDaq.ErrorInfo ACalibrateData(int numPoints, MccDaq.Range range, ref short adData)
```

Parameters:

`numPoints`	Number of samples to convert
`range`	The programmable gain/range used when the data was collected. Refer to Table 14-1 on page 155 for a list of valid range settings.
`adData`	Reference to data array

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`adData` - converted data

Notes:

When collecting data using `AInScan()` with the `NoCalibrateData` option, use this method to calibrate the data after it is collected.

- The name of the array must match that used in `AInScan()` or `WinBufToArray()`.
- Applying software calibration factors in real time on a per sample basis eats up machine cycles. If your CPU is slow, or if processing time is at a premium, withhold calibration until after the acquisition run is complete. Turning off real time software calibration saves CPU time during a high speed acquisition run.

  Processor speed is a factor for DMA transfers and for real time software calibration. Processors of less than 150 MHz Pentium class may impose speed limits below the capability of the board (refer to specific board information.) If your processor is less than a 150 MHz Pentium, and you need an acquisition speed in excess of 200 kHz, use the `NoCalibrateData` option to a turn off real-time software calibration and save CPU time. After the acquisition is run, calibrate the data with `ACalibrateData()`.

AIn()

Reads an A/D input channel. This method reads the specified A/D channel from the specified board. If the specified A/D board has programmable gain then it sets the gain to the specified range. The raw A/D value is converted to an A/D value and returned to `DataValue`.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function AIn(ByVal channel As Integer, ByVal range As
MccDaq.Range , ByRef dataValue As Short) As MccDaq.ErrorInfo
```

```
Public Function AIn(ByVal channel As Integer, ByVal range As
MccDaq.Range, ByRef dataValue As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo AIn(int channel, MccDaq.Range range, out
ushort DataValue)
```

```
public MccDaq.ErrorInfo AIn(int channel, MccDaq.Range range, out
short DataValue)
```

Parameters:

`channel`	A/D channel number. The maximum allowable channel depends on which type of A/D board is being used. For boards with both single ended and differential inputs, the maximum allowable channel number also depends on how the board is configured. For example, a CIO-DAS1600 has 8 channels for differential, 16 for single ended. Expansion boards are also supported by this method, so this parameter can contain values up to 272. See board specific information for EXP boards if you are using an expansion board.
`range`	A/D Range code. If the selected A/D board does not have a programmable gain feature, this parameter is ignored. If the A/D board does have programmable gain, set the `range` parameter to the desired A/D range. Refer to board specific information for a list of the supported A/D ranges of each board. Refer to Table 14-1 on page 155 for a list of valid range settings.
`dataValue`	Reference to data value.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`dataValue` - Returns the value of the A/D sample.

AInScan()

Scans a range of A/D channels and stores the samples in an array. `AInScan()` reads the specified number of A/D samples at the specified sampling rate from the specified range of A/D channels from the specified board. If the A/D board has programmable gain, then it sets the gain to the specified range. The collected data is returned to the data array.

Member of the `MccBoard class`.

Function prototype:

VB .NET:
```
Public Function AInScan(ByVal lowChan As Integer, ByVal highChan As
Integer, ByVal numPoints As Integer, ByRef rate As Integer, ByVal
range As MccDaq.Range , ByVal memHandle As Integer, ByVal options As
MccDaq.ScanOptions ) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo AInScan(int lowChan , int highChan, int
numPoints, ref int rate, MccDaq.Range range, int memHandle,
MccDaq.ScanOptions options)
```

Parameters:

`lowChan` — First A/D channel of the scan. When `ALoadQueue()` is used, the channel count is determined by the total number of entries in the channel gain queue. `lowChan` is ignored.

`highChan` — Last A/D channel of the scan. When `ALoadQueue()` is used, the channel count is determined by the total number of entries in the channel gain queue. `highChan` is ignored.

low / high Channel # - The maximum allowable channel depends on which type of A/D board is being used. For boards that have both single ended and differential inputs the maximum allowable channel number also depends on how the board is configured. For example, a CIO-DAS1600 has 8 channels for differential, 16 for single ended.

`numPoints` — Number of A/D samples to collect. Specifies the total number of A/D samples that will be collected. If more than one channel is being sampled then the number of samples collected per channel is equal to `count` / (`highChan`- `lowChan`+1).

`rate` — The sample rate at which scans are triggered, in scans per second per channel.

For example, sampling four channels, 0-3, at a rate of 10,000 scans per second (10 kHz) results in an A/D converter rate of 40 kHz: four channels at 10,000 samples per channel per second. With other software, you specify the total A/D chip rate. In those systems, the per channel rate is equal to the A/D rate divided by the number of channels in a scan.

The channel count is determined by the `lowChan` and `highChan` parameters. Channel Count = (`highChan` - `lowChan` + 1).

When `ALoadQueue()` is used, the channel count is determined by the total number of entries in the channel gain queue. `lowChan` and `highChan` are ignored.

`rate` also returns the value of the actual rate set, which may be different from the requested rate because of pacer limitations.

`range` — A/D range code. If the selected A/D board does not have a programmable range feature, this parameter is ignored. Otherwise, set the `range` parameter to any range that is supported by the selected A/D board. Refer to board-specific information for a list of the supported A/D ranges of each board. Refer to Table 14-1 on page 155 for a list of valid range settings.

`memHandle`	Handle for Windows buffer to store data in (Windows). This buffer must have been previously allocated with the `WinBufAlloc()` method.
`options`	Bit fields that control various options . Refer to the constants in the "options parameter values" section below.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`rate` - actual sampling rate used.

`memHandle` - collected A/D data returned via the Windows buffer.

options parameter values:

All of the `options` settings are `MccDaq.ScanOptions` enumerated constants. To set a variable to one of these constants, you must refer to the MccDaq object and the `ScanOptions` enumeration (*variable* = `MccDaq.ScanOptions.SingleIo`, *variable* = `MccDaq.ScanOptions.DmaIo`, etc.).

Transfer method options: The following three options determine how data is transferred from the board to PC memory. If none of these options is specified (recommended), the optimum sampling mode will automatically be chosen based on board type and sampling speed.

`SingleIo`	A/D transfers to memory are initiated by an interrupt. One interrupt per conversion.
`DmaIo`	A/D transfers are initiated by a DMA request.
`BlockIo`	A/D transfers are handled in blocks (by REP-INSW for example). **BlockIo is not recommended for slow acquisition rates:** If the rate of acquisition is very slow (say less than 200 Hz) `BlockIo` is probably not the best choice for transfer mode. The reason for this is that status for the operation is not available until one packet of data has been collected (typically 512 samples). The implication is that if acquiring 100 samples at 100 Hz using `BlockIo`, the operation will not complete until 5.12 seconds has elapsed.
`BurstIo`	Allows higher sampling rates (up to 8000 Hz) for sample counts up to full FIFO. Data is collected into the local FIFO. Data transfers to the PC are held off until after the scan is complete. For `Background` scans, the count and index returned by `GetStatus` remain 0 and the status equals `Running` until the scan finishes. When the scan finishes and the data is retrieved, the count and index are updated and the status equals `Idle`. `BurstIo` is the default mode for non-Continuous fast scans (aggregate sample rates above 1000 Hz) with sample counts up to full-FIFO. To avoid the `BurstIo` default, specify `BlockIo`. Non-`BurstIo` scans are limited to a maximum of 1200 Hz. `BurstIo` is not a valid option for most boards. It is used mainly for USB products.

`BurstMode`	Enables burst mode sampling. Scans from `lowChan` to `highChan` are clocked at the maximum A/D rate between samples in order to minimize channel to channel skew. Scans are initiated at the rate specified by rate. `BurstMode` is not recommended for use with the `SingleIo` option. If this combination is used, the count value should be set as low as possible, preferably to the number of channels in the scan. Otherwise, overruns may occur.

`ConvertData`	If the `ConvertData` option is used for 12 bit boards then the data that is returned to the buffer will automatically be converted to 12 bit A/D values. If `ConvertData` is not used then the data from 12 bit A/D boards will be return unmodified (16 bit values that contain both a 12 bit A/D value and a 4 bit channel number). After the data collection is complete you can call `AConvertData()` to convert the data after the fact. `ConvertData` may not be specified if you are using the `Background` option and DMA transfers. This option is ignored for the 16 bit boards.
`Background`	If the `Background` option is not used, the `AInScan()` method will not return to your program until all of the requested data has been collected and returned to the buffer. When the `Background` option is used, control will return immediately to the next line in your program and the data collection from the A/D into the buffer will continue in the background. Use `GetStatus()` to check on the status of the background operation. Alternatively, some boards support `EnableEvent()` for event notification of changes in status of `Background` scans. Use `StopBackground()` to stop the background process before it has completed. StopBackground() should be executed after normal termination of all background functions in order to clear variables and flags.
`Continuous`	This option puts the method in an endless loop. Once it collects the required number of samples, it resets to the start of the buffer and begins again. The only way to stop this operation is with `StopBackground()`. Normally this option should be used in combination with `Background` so that your program will regain control. **numPoints parameter settings in CONTINUOUS mode:** For some DAQ hardware, `numPoints` must be an integer multiple of the *packet size.* Packet size is the amount of data that a DAQ device transmits back to the PC's memory buffer during each data transfer. Packet size can differ among DAQ hardware, and can even differ on the same DAQ product depending on the transfer method. In some cases, the minimum value for the `numPoints` parameter may change when the CONTINUOUS option is used. This can occur for several reasons; the most common is that in order to trigger an interrupt on boards with FIFOs, the circular buffer must occupy at least half the FIFO. Typical half-FIFO sizes are 256, 512 and 1024. Another reason for a minimum `numPoints` value is that the buffer in memory must be periodically transferred to the user buffer. If the buffer is too small, data will be overwritten during the transfer resulting in garbled data. Refer board-specific section of the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) for packet size information for your particular DAQ hardware.
`ExtClock`	If this option is used then conversions will be controlled by the signal on the external clock input rather than by the internal pacer clock. Each conversion will be triggered on the appropriate edge of the clock input signal (see board-specific info). When this option is used the `rate` parameter is ignored. The sampling rate is dependent on the clock signal. `Options` for the board will default to a transfer mode that will allow the maximum conversion rate to be attained unless otherwise specified. **SingleIo is recommended for slow external clock rates:** If the rate of the external clock is very slow (say less than 200 Hz) and the board you are using supports `BlockIo`, you may want to include the `SingleIo` option. This is because that the status for the operation is not available until one packet of data has been collected (typically 512 samples). The implication is that, if acquiring 100 samples at 100 Hz using `BlockIo` (the default for boards that support it if `ExtClock` is used), the operation will not complete until 5.12 seconds has elapsed

`ExtMemory`	Causes the command to send the data to a connected memory board via the DT-Connect interface rather than returning the data to the buffer. Data for each call to this method will be appended unless `MemReset()` is called. The data should be unloaded with the `MemRead()` method before collecting new data. When `ExtMemory` option is used, the reference to the buffer (memHandle) may be set to null or 0. `Continuous` option cannot be used with `ExtMemory`. Do not use `ExtMemory` and `DtConnect` together. The transfer modes `DmaIo`, `SingleIo` and `BlockIo` have no meaning when used with this option.
`ExtTrigger`	If this option is specified, the sampling will not begin until the trigger condition is met. On many boards, this trigger condition is programmable (refer to `SetTrigger()` and to board-specific info for details). On other boards, only 'polled gate' triggering is supported. In this case assuming active high operation, data acquisition will commence immediately if the trigger input is high. If the trigger input is low, acquisition will be held off unit until it goes high. Acquisition will then continue until `numPoints` samples have been taken regardless of the state of the trigger input. This option is most useful if the signal is a pulse with a very low duty cycle (trigger signal in TTL low state most of the time) so that triggering will be held off until the occurrence of the pulse.
`NoTodInts`	If this option is specified, the system's time-of-day interrupts are disabled for the duration of the scan. These interrupts are used to update the systems real time clock and are also used by various other programs. These interrupts can limit the maximum sampling speed of some boards - particularly the PCM-DAS08. If the interrupts are turned off using this option then the real-time clock will fall behind by the length of time that the scan takes.
`NoCalibrateData`	Turns off real-time software calibration for boards which are software calibrated, by applying calibration factors to the data on a sample by sample basis as it is acquired. Examples are the PCM-DAS16/330 and PCM-DAS16x/12. Turning off software calibration saves CPU time during a high speed acquisition run. This may be required if your processor is less than a 150 MHz Pentium and you desire an acquisition speed in excess of 200 kHz. These numbers may not apply to your system. Only trial will tell for sure. DO NOT use this option if you do not have to. If this option is used, the data must be calibrated after the acquisition run with the `ACalibrateData()` method.
`DTConnect`	All A/D values will be sent to the A/D board's DT-Connect port. This option is incorporated into the `ExtMemory` option. Use `DTConnect` only if the external board is not supported by Universal Library.

Caution! You will generate an error if you specify a total A/D rate beyond the capability of the board. For example; if you specify rate `LowChan` = 0, `HighChan` = 7 (8 channels total) and `Rate` = 20,000 and you are using a CIO-DAS16/JR, you will get an error. You have specified a total rate of 8*20,000 = 160,000. The CIO-DAS16/JR can convert up to 120,000 samples per second. The maximum sampling rate depends on the A/D board that is being used. It is also dependent on the sampling mode options.

Important

In order to understand the functions, read the board-specific information contained in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Review and run the example programs before attempting any programming of your own. Following this advice will save you hours of frustration, and possibly time wasted holding for technical support.

This note, which appears elsewhere, is especially applicable to this method. Now is the time to read board-specific information for your board (see the *Universal Library User's Guide*). We suggest that you make a copy of that page to refer to as you read this manual and examine the example programs.

ALoadQueue()

Loads the A/D board's channel/gain queue. This method only works with A/D boards that have channel/gain queue hardware.

Some products do not support channel / gain queue, and some that do support it are limited on the order of elements, number of elements, and gain values that can be included, etc. Please refer to the device-specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) to find details for your particular product.

Member of the `MccBoard` class.

Function prototype:

VB .NET:
```
Public Function ALoadQueue(ByVal chanArray As Short( ), ByVal gainArray As MccDaq.Range (), ByVal count As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo ALoadQueue(short[] chanArray, MccDaq.Range[] gainArray, int count)
```

Parameters:

`chanArray` Array containing channel values. This array should contain all of the channels that will be loaded into the channel gain queue.

`gainArray` Array containing A/D range values. This array should contain each of the A/D ranges that will be loaded into the channel gain queue. Refer to Table 14-1 on page 155 for a list of valid A/D range settings.

`count` Number of elements in chanArray and gainArray or 0 to disable chan/gain queue. Specifies the total number of chan/gain pairs that will be loaded into the queue.

`chanArray` and `gainArray` should contain at least count elements. Set count = 0 to disable the board's chan/gain queue. The maximum value is specific to the queue size of the A/D boards channel gain queue.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

Normally, the `AInScan()` method scans a fixed range of channels (from `lowChan` to `highChan`) at a fixed A/D range. If you load the channel gain queue with this method then all subsequent calls to `AInScan()` will cycle through the chan/range pairs that you have loaded into the queue.

AOut()

Sets the value of a D/A output.

Member of the MccBoard class.

Function prototype:

VB .NET:

```
Public Function AOut(ByVal channel As Integer, ByVal range As MccDaq.Range, ByVal dataValue As Short) As MccDaq.ErrorInfo
Public Function AOut(ByVal channel As Integer, ByVal range As MccDaq.Range, ByVal dataValue As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo AOut(int channel, MccDaq.Range range, ushort dataValue)
public MccDaq.ErrorInfo AOut(int channel, MccDaq.Range range, short dataValue)
```

Parameters:

`channel` — D/A channel number. The maximum allowable channel depends on which type of D/A board is being used.

`range` — D/A range code. The output range of the D/A channel can be set to any of those supported by the board. If the D/A board does not have programmable ranges then this parameter will be ignored. Refer to Table 14-1 on page 155 for a list of valid range settings.

`dataValue` — Value to set D/A to. Must be in the range 0 - N where N is the value $2^{Resolution}$ - 1 of the converter

Exception: using 16 bit boards with Basic range is -32768 to 32767. Refer to the discussion on Basic signed integers for more information.

Returns:

An ErrorInfo object that indicates the status of the operation.

Notes:

Simultaneous Update Boards: If you set the simultaneous update jumper for simultaneous operation, use AOutScan() for simultaneous update of multiple channels. AOut() always writes the D/A data then reads the D/A, which causes the D/A output to be updated.

AOutScan()

Outputs values to a range of D/A channels. This function can be used for paced analog output on hardware that supports paced output. It can also be used to update all analog outputs at the same time when the `Simultaneous` option is used.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function AOutScan(ByVal lowChan As Integer, ByVal highChan As
Integer, ByVal numPoints As Integer, ByRef rate As Integer, ByVal
range As MccDaq.Range , ByVal memHandle As Integer, ByVal options As
MccDaq.ScanOptions) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo AOutScan(int lowChan, int highChan, int
numPoints, ref int rate, MccDaq.Range range, int memHandle,
MccDaq.ScanOptions options)
```

Parameters:

`lowChan` First D/A channel of scan.

`highChan` Last D/A channel of scan.

`lowChan`/`highChan` - The maximum allowable channel depends on which type of D/A board is being used.

`numPoints` Number of D/A values to output. Specifies the total number of D/A values that will be output. Most D/A boards do not support timed outputs. For these boards, set the count to the number of channels in the scan.

`rate` Sample rate in scans per second. For many D/A boards the `rate` is ignored and can be set to `NotUsed`. For D/A boards with trigger and transfer methods which allow fast output rates, such as the CIO-DAC04/12-HS, `rate` should be set to the D/A output rate (in scans/sec). This parameter also returns the value of the actual rate set. This value may be different from the user specified rate because of pacer limitations.

If supported, this is the rate at which scans are triggered. If you are updating 4 channels, 0-3, then specifying a rate of 10,000 scans per second (10 kHz) will result in the D/A converter rates of 10 kHz — (one D/A per channel). The data transfer rate will be 40,000 words per second — 4 channels * 10,000 updates per scan.

The maximum update rate depends on the D/A board that is being used, and the sampling mode options.

`range` D/A range code. The output range of the D/A channel can be set to any of those supported by the board. If the D/A board does not have a programmable then this parameter will be ignored. Refer to Table 14-1 on page 155 for a list of valid range settings.

`memHandle` Handle for Windows buffer from which data will be output. This buffer must have been previously allocated with the `WinBufAlloc()` method and data values loaded (perhaps using `WinArrayToBuf()`.

`scanOptions` Bit fields that control various options . Refer to the constants in the "scanOptions" section on page 168.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`Rate` - actual sampling rate used.

scanOptions parameter values:

All of the `scanOptions` settings are `MccDaq.ScanOptions` enumerated constants. To set a variable to one of these constants, you must refer to the MccDaq object and the `ScanOptions` enumeration (*variable* = `MccDaq.ScanOptions.Continuous`, *variable* = `MccDaq.ScanOptions.Background`, etc.).

`Continuous`	This option may only be used with boards which support interrupt, DMA or REP-INSW transfer methods. This option puts the method in an endless loop. Once it outputs the specified (by Count) number of D/A values, it resets to the start of `the buffer` and begins again. The only way to stop this operation is with `StopBackground()`. This option should only be used in combination with `Background` so that your program can regain control.
`Background`	This option may only be used with boards which support interrupt, DMA or REP-INSW transfer methods. When this option is used the D/A operations will begin running in the background and control will immediately return to the next line of your program. Use `GetStatus()` to check the status of background operation. Alternatively, some boards support `EnableEvent()` for event notification of changes in status of `Background` scans. Use `StopBackground()` to terminate background operations before they are completed. `StopBackground()` should be executed after normal termination of all background functions in order to clear variables and flags.
`Simultaneous`	When this option is used (if the board supports it and the appropriate switches are set on the board) all of the D/A voltages will be updated simultaneously when the last D/A in the scan is updated. This generally means that all the D/A values will be written to the board, then a read of a D/A address causes all D/As to be updated with new values simultaneously.
`ExtClock`	If this option is used then conversions will be paced by the signal on the external clock input rather than by the internal pacer clock. Each conversion will be triggered on the appropriate edge of the clock input signal (see board-specific info). When this option is used the `Rate` parameter is ignored. The sampling rate is dependent on the clock signal. Options for the board will default to transfer types that allow the maximum conversion rate to be attained unless otherwise specified.
`ExtTrigger`	If this option is specified the sampling will not begin until the trigger condition is met. On many boards, this trigger condition is programmable (see `SetTrigger()` method and board-specific information for details).

Caution! You will generate an error if you specify a total D/A rate beyond the capability of the board. For example: If you specify `LowChan` = 0 and `HighChan` = 3 (4 channels total) and `Rate` = 100,000, and you are using a cSBX-DDA04, you will get an error. You have specified a total rate of 4*100,000 = 400,000. The cSBX-DDA04 is rated to 330,000 updates per second. The maximum update rate depends on the D/A board that is being used. It is also dependent on the sampling mode options.

APretrig()

Waits for a trigger to occur and then returns a specified number of analog samples before and after the trigger occurred. If only 'polled gate' triggering is supported, the trigger input line (refer to the user's manual for the board) must be at TTL low before this method is called, or a TrigState error will occur. The trigger occurs when the trigger condition is met. Refer to the `SetTrigger()` method for more details.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function APretrig(ByVal lowChan As Integer, ByVal highChan As
Integer, ByRef pretrigCount As Integer, ByRef totalCount As Integer,
ByRef rate As Integer, ByVal range As MccDaq.Range, ByVal memHandle
As Integer, ByVal options As MccDaq.ScanOptions) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo APretrig(int lowChan, int highChan, ref int
pretrigCount, ref int totalCount, ref int rate, MccDaq.Range range,
int memHandle, MccDaq.ScanOptions options)
```

Parameters:

`lowChan` First A/D channel of scan.

`highChan` Last A/D channel of scan.

lowChan/highChan - The maximum allowable channel depends on which type of A/D board is being used. For boards with both single ended and differential inputs, the maximum allowable channel number also depends on how the board is configured (e.g., 8 channels for differential inputs, 16 for single ended inputs).

`pretrigCount` Number of pre-trigger A/D samples to collect. Specifies the number of samples to collect before the trigger occurs. `PretrigCount` must be less than the (`totalCount` - 512).

If the trigger occurs too early, fewer than the requested number of pre-trigger samples will be collected, and a `TooFew` error will occur. The pretrigCount will be set to indicate how many samples were actually collected. The post trigger samples will still be collected.

`totalCount` Total number of A/D samples to collect. Specifies the total number of samples that will be collected and stored in the buffer. `TotalCount` must be greater than or equal to the `PretrigCount` + 512.

If the trigger occurs too early, fewer than the requested number of samples will be collected, and a `TooFew` error will occur. The `totalCount` will be set to indicate how many samples were actually collected.

`TotalCount` must be evenly divisible by the number of channels being scanned. If it is not, this method will adjust the number (down) to the next valid value and return that value to the totalCount parameter.

`rate` Sample rate in scans per second.

`range` A/D Range code. If the selected A/D board does not have a programmable gain feature, this parameter is ignored. Otherwise, set to any range that is supported by the selected A/D board. Refer to board specific information for a list of the supported A/D ranges of each board. Refer to Table 14-1 on page 155 for a list of valid range settings.

`memHandle` Handle for Windows buffer to store data in (Windows). This buffer must have been previously allocated with the `WinBufAlloc()` method.

`options`	Bit fields that control various options . Refer to the constants in the "options parameter values" section below.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`pretrigCount` - Number of pre-trigger samples

`totalCount` - Total number of samples collected

`rate` - actual sampling rate

`memHandle` - Collected A/D data returned via the Windows buffer

options parameter values:

All of the `options` settings are `MccDaq.ScanOptions` enumerated constants. To set a variable to one of these constants, you must refer to the MccDaq object and the `ScanOptions` enumeration (*variable* = `MccDaq.ScanOptions.DTConenct`, *variable* = `MccDaq.ScanOptions.ExtMemory`, etc.).

`ConvertData`	The data is collected into a "circular" buffer. When the data collection is complete, the data is in the wrong order. If you use the `ConvertData` option, the data is automatically rotated into the correct order (and converted to 12 bit values if required) when the data acquisition is complete. Otherwise, call `AConvertPretrigData()` to rotate the data. You cannot use the `ConvertData` option in combination with the `Background` option for this function.
`Background`	If the `Background` option is not used, the `APretrig()` method will not return to your program until all of the requested data has been collected and returned to the buffer. When the `Background` option is used, control returns immediately to the next line in your program, and the data collection from the A/D into the buffer will continue in the background. Use `GetStatus()` to check on the status of the background operation. Alternatively, some boards support `EnableEvent()` for event notification of changes in status of `Background` scans. Use `StopBackground()` to terminate the background process before it has completed. Call `StopBackground()` after normal termination of all background functions to clear variables and flags. You cannot use the `CONVERTDATA` option in combination with the `BACKGROUND` option for this function. To correctly order and parse the data, use `AConvertPretrigData()`after the function completes.
`ExtClock`	This option is available only for boards that have separate inputs for external pacer and external trigger. Refer to your hardware manual or board-specific information.
`ExtMemory`	Causes this method to send the data to a connected memory board via the DT-Connect interface rather than returning the data to the buffer. If you use this option to send the data to a MEGA-FIFO memory board, then you must use `MemReadPretrig()` to later read the pre-trigger data from the memory board. If you use `MemRead()`, the data will NOT be in the correct order. Every time this option is used, it overwrites any data already stored in the memory board. All data should be read from the board (with `MemReadPretrig()`) before collecting any new data. When this option is used, the memHandle parameter is ignored. The MEGA-FIFO memory must be fully populated in order to use the `APretrig()` method with the `ExtMemory` option.

`DTConnect`	When the `DtConnect` option is used with this method the data from ALL A/D conversions is sent out the DT-Connect interface. While this method is waiting for a trigger to occur, it will send data out the DT-Connect interface continuously. If you have a Measurement Computing memory board plugged into the DT-Connect interface then you should use the `ExtMemory` option rather than this option.

Important

The buffer referenced by `memHandle` must be big enough to hold at least `TotalCount` + 512 integers

ATrig()

Waits for a specified analog input channel to go above or below a specified value. `ATrig` continuously reads the specified channel and compares its value to `trigValue`. Depending on whether `trigType` is set to `TrigAbove` or `TrigBelow`, it waits for the first A/D sample that is above or below `trigValue`. The first sample that meets the trigger criteria is returned to `dataValue`.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function ATrig(ByVal chan As Integer, ByVal trigType As
MccDaq.TriggerType, ByVal trigValue As Short, ByVal range As
MccDaq.Range, ByRef dataValue As Short) As MccDaq.ErrorInfo
```

```
Public Function ATrig(ByVal chan As Integer, ByVal trigType As
MccDaq.TriggerType, ByVal trigValue As System.UInt16, ByVal range As
MccDaq.Range, ByRef dataValue As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo ATrig (int chan, MccDaq.TriggerType
trigType, short trigValue, MccDaq.Range range, out short dataValue)
```

```
public MccDaq.ErrorInfo ATrig(int chan, MccDaq.TriggerType trigType,
ushort trigValue, MccDaq.Range range, out ushort dataValue)
```

Parameters:

`chan`	A/D channel number. The maximum allowable channel depends on which type of A/D board is being used. For boards with both single ended and differential inputs, the maximum allowable channel number also depends on how the board is configured. For example a CIO-DAS1600 has eight channels for differential inputs and 16 channels for single-ended inputs.
`trigType`	`MccDaq.TriggerType.TrigAbove` or `MccDaq.TriggerType.TrigBelow`. Specifies whether to wait for the analog input to be above or below the specified trigger value.
`trigValue`	The threshold value that all A/D values are compared to. Must be in the range 0 - 4095 for 12 bit A/D boards, or 0-65,535 for 16-bit A/D boards. Refer to your BASIC manual for information on signed BASIC integer data types.
`range`	Gain code. If the selected A/D board does not have a programmable gain feature, this parameter is ignored. Otherwise, set to any range that is supported by the selected A/D board. Refer to Table 14-1 on page 155 for a list of valid range settings. Refer to board specific information for a list of the supported A/D ranges of each board.
`dataValue`	Returns the value of the first A/D sample to meet the trigger criteria.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`dataValue` - value of the first A/D sample to match the trigger criteria.

Notes:

Ctrl-C will not terminate the wait for an analog trigger that meets the specified condition. There are only two ways to terminate this call: satisfy the trigger condition or reset the computer.

Caution! Use caution when using this method in Windows programs. All active windows will lock on the screen until the trigger condition is satisfied. All keyboard and mouse activity will also lock until the trigger condition is satisfied.

Configuration Methods and Properties

Introduction

This section covers Universal Library for .NET methods and properties that retrieve or change configuration options on a board. The configuration information for all boards is stored in the configuration file CB.CFG. This information is loaded from CB.CFG by all programs that use the library.

To determine which of these methods are compatible with your hardware, refer to the board-specific information contained in the *Universal Library User's Guide (*available in PDF format on our website at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

BoardConfig property

Represents an instance of the cBoardConfig class. Use this property to call the board configuration methods.

Member of the MccBoard class.

Property prototype:

VB .NET:	`Public ReadOnly Property BoardConfig As MccDaq.cBoardConfig`
C# .NET	`public MccDaq.cBoardConfig BoardConfig [get]`

Methods:

Over 20 UL for .NET configuration methods are accessible only from the `BoardConfig` property. Before you call any of these methods, you need to create an instance of an MccBoard object.

```
Dim MyBoard As MccDaq.MccBoard
MyBoard = New MccDaq.MccBoard(MyBoardNum)
```

To call a method from the `BoardConfig` property, use the notation shown in the example below.

```
MyErrorInfo = MyBoard.BoardConfig.GetBoardType(MyBoardType)
```

Each method available from the `BoardConfig` property is explained below.

BoardConfig.DACUpdate()

Updates the voltage values on analog output channels. This method is usually called after a SetDACUpdateMode() method call with its `configVal` parameter set to 1 (on command).

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

VB .NET:	`Public Function DACUpdate() As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo DACUpdate()`

Returns:

An ErrorInfo object that indicates the status of the operation.

BoardConfig.GetBaseAdr()

Gets the base address of a board.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

VB .NET:	`Public Function GetBaseAdr(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetBaseAdr(int devNum, out int configVal)`

Parameters:

`devNum`	Number of the base address to return (PCI boards may have several address ranges).
`configVal`	Board's base address.

Returns:

An ErrorInfo object that indicates the status of the operation.

BoardConfig.GetBoardType()

Gets the unique number (device ID) assigned to the board (between 0 and 8000h) indicating the type of board installed.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

VB .NET:	`Public Function GetBoardType(ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetBoardType (out int configVal)`

Parameters:

`configVal`	Returns a number indicating the board type.

Returns:

An ErrorInfo object that indicates the status of the operation.

BoardConfig.GetCiNumDevs()

Gets the number of counter devices on the board.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

VB .NET:	`Public Function GetCiNumDevs(ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetCiNumDevs(out int configVal)`

Parameters:

`configVal`	Returns the number of counter devices.

Returns:

An ErrorInfo object that indicates the status of the operation.

BoardConfig.GetClock()

Gets the counter's clock frequency in MHz (40, 10, 8, 6, 5, 4, 3, 2, 1), or 0 for not supported.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

VB .NET:	`Public Function GetClock(ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetClock(out int configVal)`

Parameters:

`configVal`	Clock frequency in MHz.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetDACStartup()

Returns the board's configuration register STARTUP bit setting. Refer to the "Notes" section for the `SetDACStartup()` method on page 182 for more information.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function GetDACStartup(ByVal configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetDACStartup(out int configVal)`

Parameters:

`configVal`	Returns setting of startup bit (0 or 1).

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`configVal`	Returns 0 if startup bit is disabled, or 1 if startup bit is enabled.

BoardConfig.GetDACUpdateMode()

Returns the update mode for a digital-to-analog converter (DAC).

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function GetDACUpdateMode(ByVal devNum as Integer, ByVal configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetDACUpdateMode(int devNum, out int configVal)`

Parameters:

`devNum`	Number of the channel whose update mode you want set.
`configVal`	Returns a number indicating the DAC update mode (0 = *immediate*, 1 = *on command*).

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`configVal`	If `ConfigVal` returns 0, the DAC update mode is immediate. Values written with `AOut()` or `AOutScan()` are automatically output by the DAC channels. If `ConfigVal` returns 1, the DAC update mode is set to *on command*. Values written with `AOut()` or `AOutScan()` are not output by the DAC channels until a `DACUpdate()` method call is made.

BoardConfig.GetDiNumDevs()

Gets the number of digital devices on the board.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function GetDiNumDevs(ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetDiNumDevs(out int configVal)`

Parameters:

`configVal`	Returns the number of digital devices.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetDmaChan()

Gets the DMA channel (0, 1 or 3) set for the board.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function GetDmaChan(ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetDmaChan(out int configVal)`

Parameters:

`configVal`	Returns DMA channel. 0, 1 or 3

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetDtBoard()

Gets the number of the board with the DT-Connect interface used to connect to external memory boards.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function GetDtBoard(ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetDtBoard(out int configVal)`

Parameters:

`configVal`	Returns the board number of the board that the external memory board is connected to.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetIntLevel()

Gets the interrupt level set for the board (0 for none, or 1 to 15).

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET: `Public Function GetIntLevel(ByRef configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetIntLevel(out int configVal)`

Parameters:

`configVal` Returns the interrupt level (0 for none, or 1 – 15).

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetNumAdChans()

Gets the number of A/D channels.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET: `Public Function GetNumAdChans(ByRef configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetNumAdChans(out int configVal)`

Parameters:

`configVal` Returns the number of A/D channels.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetNumDaChans()

Gets the number of D/A channels.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET: `Public Function GetNumDaChans(ByRef configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetNumDaChans(out int configVal)`

Parameters:

`configVal` Returns the number of D/A channels.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetNumExps()

Gets the number of expansion boards.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function GetNumExps(ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetNumExps(out int configVal)`

Parameters:

`configVal`	Returns the number of expansion boards attached to the board.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetNumIoPorts()

Gets the number of I/O ports used by the board.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function GetNumIoPorts(ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetNumIoPorts(out int configVal)`

Parameters:

`configVal`	Returns the number of I/O ports used by the board.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetRange()

Gets the selected voltage range. For switch-selectable gains only.

If the selected A/D board does not have a programmable gain feature, this method returns the range as defined by the installed *Insta*Cal settings. If *Insta*Cal and the board are installed correctly, the range returned corresponds to the input range set by switches on the board. Refer to board-specific information for a list of the A/D ranges supported by each board.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function GetRange(ByRef configVal As MccDaq.Range) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetRange(out MccDaq.Range configVal)`

Parameters:

`configVal`	Returns the selected voltage range. Refer to Table 14-1 on page 155 for a list of valid `configVal` settings.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetUsesExps()

Gets the *True*/*False* value indicating support of expansion boards.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET: `Public Function GetUsesExps(ByRef configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetUsesExps(out int configVal)`

Parameters:

`configVal` Returns *True* if the board supports expansion boards, or *False* if the board does not support expansion boards.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.GetWaitState()

Gets the value of the Wait State jumper (1-enabled, 0-disabled).

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET: `Public Function GetWaitState(ByRef configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetWaitState(out int configVal)`

Parameters:

`configVal` Returns the wait state of the board.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.SetBaseAdr()

Sets the base address used by the Universal Library to communicate with a board. This is recommended for use only with ISA bus boards.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET: `Public Function SetBaseAdr(ByVal devNum As Integer, ByVal configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo SetBaseAdr(int devNum, int configVal)`

Parameters:

`devNum`	Number of the base address to configure (should always be 0 – can't configure PCI base addresses).
`configVal`	Sets the base address of the board.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.SetClock()

Sets the counter's clock source by the frequency (40, 10, 8, 6, 5, 4, 3, 2, 1), or 0 for not supported.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function SetClock(ByVal configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo SetClock(int configVal)`

Parameters:

`configVal`	Sets the clock frequency in MHz.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.SetDmaChan()

Sets the DMA channel (0, 1 or 3).

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function SetDmaChan(ByVal configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo SetDmaChan(int configVal)`

Parameters:

`configVal`	Sets the DMA channel to 0, 1 or 3.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.SetDACStartup()

Sets the board's configuration register STARTUP bit to 0 or 1 to enable/disable the storing of digital-to-analog converter (DAC) startup values. Each time the DAC board is powered up, the stored values are written to the DACs. New DAC start-up values are stored in memory by. Refer to the "Notes" section below for more information.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function SetDACStartup(ByVal configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo SetDACStartup(int configVal)`

Parameters:

`configVal`	Set to 0 to disable, or 1 to enable the storing of startup values for the channel.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

Use the `SetDACStartup()` method to store the DAC values you would like each DAC channel to be set to each time the board is powered up.

To store the current DAC values as start-up values, call `SetDACStartup()` with a `configVal` value of 1. Then, each time you call `AOut()`or `AOutScan()`,the value written for each channel is stored in NV RAM. The last value written to a particular channel while `SetDACStartup()` is set to 1 is the value that that channel will be set to at power up. Call `SetDACStartup()` again with a `configVal` value of 0 to stop storing values in NV RAM.

Example:

```
DacBoard.BoardConfig.SetDACStartup(1);
for (int i =1; i <8; i++)
{
DacBoard.AOut(i, BIP5VOLTS, DACValue[i]);
}
DacBoard.BoardConfig.SetDACStartup(0);
```

BoardConfig.SetDACUpdateMode()

Sets the update mode for a digital-to-analog converter (DAC).

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function SetDACUpdateMode(ByVal devNum as Integer, ByVal configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo SetDACUpdateMode(int devNum, int configVal)`

Parameters:

`devNum`	Number of the channel whose update mode you want set.
`configVal`	When set to 0, the DAC update mode is *immediate*. Values written with `AOut()` or `AOutScan()` are automatically output by the DAC channels. When set to 1, the DAC update mode is *on command*. Values written with `AOut()` or `AOutScan()` are not output by the DAC channel(s) until a `DACUpdate()` method call is made.

Returns:

An ErrorInfo object that indicates the status of the operation.

BoardConfig.SetIntLevel()

Sets the interrupt level: 0 for none, or 1 to 15. Recommended for use only with ISA bus boards.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

VB .NET:
```
Public Function SetIntLevel(ByVal configVal As Integer) As
MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo SetIntLevel(int configVal)
```

Parameters:

`configVal` Sets the interrupt level. Valid settings are 0 for none, or 1 – 15.

Returns:

An ErrorInfo object that indicates the status of the operation.

BoardConfig.SetNumAdChans()

Sets the number of A/D channels available on the board.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

VB .NET:
```
Public Function SetNumAdChans(ByVal configVal As Integer) As
MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo SetNumAdChans(int configVal)
```

Parameters:

`configVal` Sets the number of A/D channels on the board. Check board specific info for valid numbers. Note that this setting affects the single-ended/differential input mode of boards for which this setting is programmable.

Returns:

An ErrorInfo object that indicates the status of the operation.

BoardConfig.SetRange()

Sets the selected voltage range. For use with boards for which the range is manually selected.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

VB .NET:
```
Public Function SetRange(ByVal configVal As MccDaq.Range ) As
MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo SetRange(MccDaq.Range configVal)
```

Parameters:

`configVal` Range code.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardConfig.SetWaitState()

Sets the value of the Wait State jumper (1 = enabled, 0 = disabled).

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

VB .NET:	`Public Function SetWaitState(ByVal configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo SetWaitState(int configVal)`

Parameters:

`configVal`	Sets the wait state on the board.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

BoardNum property

Number of the board associated with an instance of the `MccBoard` class.

Member of the `MccBoard` class.

Property prototype:

VB .NET:	`Public ReadOnly Property BoardNum As Integer`
C# .NET:	`public int BoardNum [get]`

CtrConfig property

Represents an instance of the `cCtrConfig` class. Use this property to call counter chip configuration methods.

Member of the `MccBoard` class.

Property prototype:

VB .NET:	`Public ReadOnly Property CtrConfig As MccDaq.cCtrConfig`
C# .NET	`public MccDaq.cCtrConfig CtrConfig [get]`

Methods:

The `GetCtrType()` configuration method is accessible only from the `CtrConfig` property. Before you call this method, you need to create an instance of an MccBoard object.

```
Dim MyBoard As MccDaq.MccBoard
MyBoard = New MccDaq.MccBoard(MyBoardNum)
```

To call this method from the `CtrConfig` property, use the notation shown in the example below:

```
MyErrorInfo = MyBoard.CtrConfig.GetCtrType(MyCtrNum, MyCtrType)
```

This method is explained below.

CtrConfig.GetCtrType()

Gets the value that indicates the counter type.

Member of the `cCtrConfig` class. Accessible from the MccBoard.`CtrConfig` property.

Function prototype:

VB .NET:	`Public Function GetCtrType(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetCtrType(int devNum, out int configVal )`

Parameters:

`devNum`	Number of the counter device.
`configVal`	Returns the type of counter where: 1 = 8254, 2 = 9513 , 3 = 8536, 4 = 7266 or 5 = event counter

Returns:

An `ErrorInfo` object that indicates the status of the operation.

DioConfig property

Represents an instance of the `cDioConfig` class. Use this property to call various digital I/O configuration methods.

Member of the `MccBoard` class.

Property prototype:

VB .NET:	`Public ReadOnly Property DioConfig As MccDaq.cDioConfig`
C# .NET	`public MccDaq.cDioConfig DioConfig [get]`

Methods:

Six configuration methods are accessible only from the DioConfig property. Before you call any of these methods, you need to create an instance of an MccBoard object.

```
Dim MyBoard As MccDaq.MccBoard
MyBoard = New MccDaq.MccBoard(MyBoardNum)
```

To call these methods from the `DioConfig` property, use the notation shown in the example below.

```
MyErrorInfo = MyBoard.DioConfig.GetNumBits(MyDevNum, MyNumBits)
```

These methods are explained below.

DioConfig.GetDInMask()

Determines the bits on a specified port that are configured for input.

Member of the `cDioConfig` class. Accessible from the `MccBoard.DioConfig` property.

Function prototype:

VB .NET:	`Public Function GetDInMask(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetDInMask (int devNum, out int configVal)`

Parameters:

`devNum`	Number of the port whose input bit configuration you want to determine.
`configVal`	Returns a bit mask showing the bit configuration of the specified port. Any of the lower eight bits that return a value of 1 are configured for input. Each of the upper eight bits always return 0.

Returns:

An ErrorInfo object that indicates the status of the operation.

Notes:

Use `GetDInMask()` with the `GetDOutMask()` method to determine if an `AuxPort` is configurable. If you apply both methods to the same port, and both `configVal` parameters returned have input and output bits that overlap, the port is not configurable. You can determine overlapping bits by *And*ing both parameters.

For example, the PCI-DAS08 has seven bits of digital I/O (four outputs and three inputs). For this board, the `configVal` parameter returned by `GetDInMask()` is always 7 (`0000 0111`), while the `configVal` parameter returned by `GetDOutMask()` is always 15 (`0000 1111`). When you *And* both `configVal` parameters together, you get a non-zero number (7). Any non-zero number indicates that input and output bits overlap for the specified port, and that port is a non-configurable `AuxPort`.

DioConfig.GetDOutMask()

Determines the bits on a specified port that are configured for output.

Member of the `cDioConfig` class. Accessible from the `MccBoard.DioConfig` property.

Function prototype:

VB .NET: `Public Function GetDOutMask(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetDOutMask (int devNum, out int configVal)`

Parameters:

`devNum` Number of the port whose output bit configuration you want to determine.

`configVal` Returns a bit mask showing the bit configuration of the specified port. Any of the lower eight bits that return a value of 1 are configured for output. Each of the upper eight bits always return 0.

Returns:

An `ErrorInfo object` that indicates the status of the operation.

Notes:

Use `GetDInMask()` with the `GetDOutMask()` method to determine if an `AuxPort` is configurable. If you apply both methods to the same port, and both `configVal` parameters returned have input and output bits that overlap, the port is not configurable. You can determine overlapping bits by *And*ing both parameters.

For example, the PCI-DAS08 has seven bits of digital I/O (four outputs and three inputs). For this board, the `configVal` parameter returned by `GetDInMask()is` always 7 (`0000 0111`), while the `configVal` parameter returned by `GetDOutMask()` is always 15 (`0000 1111`). When you *And* both `configVal` parameters together, you get a non-zero number (7). Any non-zero number indicates that input and output bits overlap for the specified port, and that port is a non-configurable `AuxPort`.

DioConfig.GetConfig()

Gets the configuration of a digital device (digital input or digital output).

Member of the `cDioConfig` class. Accessible from the `MccBoard.DioConfig` property.

Function prototype:

VB .NET: `Public Function GetConfig(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetConfig(int devNum, out int configVal)`

Parameters:

`devNum` Number of the digital device.

`configVal` Current configuration (1 = `DigitalOut`, 2 = `DigitalIn`).

Returns:

An `ErrorInfo` object that indicates the status of the operation.

DioConfig.GetCurVal()

Gets the current value of digital outputs.

Member of the `cDioConfig` class. Accessible from the `MccBoard.DioConfig` property.

Function prototype:

VB .NET: `Public Function GetCurVal(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetCurVal(int devNum, out int configVal)`

Parameters:

`devNum`	Number of the digital device.
`configVal`	Current value of the digital output.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

DioConfig.GetDevType()

Gets the device type of the digital port (`AuxPort`, `FirstPortA`, etc.).

Member of the `cDioConfig` class. Accessible from the `MccBoard.DioConfig` property.

Function prototype:

VB .NET: `Public Function GetDevType(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetDevType(int devNum, out int configVal)`

Parameters:

`devNum`	Number of the digital device.
`configVal`	Constant that indicates the type of device (`AuxPort`, `FirstPortA`, etc.).

Returns:

An `ErrorInfo` object that indicates the status of the operation.

DioConfig.GetNumBits()

Gets the number of bits in the digital port.

Member of the `cDioConfig` class. Accessible from the `MccBoard.DioConfig` property.

Function prototype:

VB .NET: `Public Function GetNumBits(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetNumBits(int devNum, out int configVal)`

Parameters:

`devNum`	Number of the digital device.
`configVal`	Number of bits in the digital port.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig property

Represents an instance of the `cExpansionConfig` class. Use this property to call various expansion board configuration methods.

Member of the `MccBoard` class.

Property prototype:

VB .NET:	`Public ReadOnly Property ExpansionConfig As MccDaq.cExpansionConfig`
C# .NET	`public MccDaq.cExpansionConfig ExpansionConfig [get]`

Methods:

Over a dozen configuration methods are accessible only from the ExpansionConfig property. Before you call any of these methods, you need to create an instance of an MccBoard object.

```
Dim MyBoard As MccDaq.MccBoard
MyBoard = New MccDaq.MccBoard(MyBoardNum)
```

To call these methods from the `ExpansionConfig` property, use the notation shown in the example below.

```
MyErrorInfo = MyBoard.ExpansionConfig.GetBoardType(MyExpNum, MyExpType)
```

These methods are explained below.

ExpansionConfig.GetBoardType()

Gets the expansion board type.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:	`Public Function GetBoardType(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetBoardType(int devNum, out int configVal)`

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Returns a number indicating the expansion board type (refer to the "BoardType Codes" topic in the *Universal Library User's Guide).*

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.GetCjcChan()

Gets the channel that the CJC is connected to.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function GetCjcChan(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo GetCjcChan(int devNum, out int configVal)
```

Parameters:

`devNum` Number of the expansion board.

`configVal` Returns a number indicating the channel on the A/D board that the CJC is connected to.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.GetMuxAdChan1()

Gets the first A/D channel that the EXP board is connected to.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function GetMuxAdChan1(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo GetMuxAdChan1(int devNum, out int configVal)
```

Parameters:

`devNum` Number of the expansion board.

`configVal` Number indicating the first A/D channel that the EXP board is connected to.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.GetMuxAdChan2()

Gets the second A/D channel that the EXP board is connected to.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function GetMuxAdChan2(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo GetMuxAdChan2(int devNum, out int configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Number indicating the second A/D channel that the EXP board is connected to.

Returns:

An `ErrorInfo` `object` that indicates the status of the operation.

ExpansionConfig.GetNumExpChans()

Gets the number of expansion board channels.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function GetNumExpChans(ByVal devNum As Integer, ByRef
configVal As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo GetNumExpChans(int devNum, out int
configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Number of channels on the expansion board.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.GetRange1()

Gets the range/gain of the low 16 channels.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function GetRange1(ByVal devNum As Integer, ByRef configVal
As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo GetRange1(int devNum, out int configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Returns the range (gain) of the low 16 channels.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.GetRange2()

Gets the range/gain of the high 16 channels.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function GetRange2(ByVal devNum As Integer, ByRef configVal
As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo GetRange2(int devNum, out int configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Returns the range (gain) of the high 16 channels.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.GetThermType()

Gets the type of thermocouple or RTD configuration for the board (J, K, E, T, R, S, and B types).

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function GetThermType(ByVal devNum As Integer, ByRef
configVal As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo GetThermType(int devNum, out int configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Number indicating the type of thermocouple configured for the board. (J = 1, K = 2, T = 3, E = 4, R = 5, S = 6, B = 7, Platinum .00392 = 257, Platinum .00391 = 258, Platinum .00385 = 259, Copper .00427 = 260, Nickel/Iron .00581 = 261, Nickel/Iron .00527 = 262)

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.SetCjcChan()

Sets the channel that the CJC is connected to.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function SetCjcChan(ByVal devNum As Integer, ByVal configVal
As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo SetCjcChan(int devNum, int configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Sets the A/D channel to connect to the CJC.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.SetMuxAdChan1()

Sets the first A/D channel that the EXP board is connected to.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:

```
Public Function SetMuxAdChan1(ByVal devNum As Integer, ByVal
configVal As Integer) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo SetMuxAdChan1(int devNum, int configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Number indicating the first A/D channel that the EXP board is connected to.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.SetMuxAdChan2()

Sets the second A/D channel that the EXP board is connected to.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:

```
Public Function SetMuxAdChan2(ByVal devNum As Integer, ByVal
configVal As Integer) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo SetMuxAdChan2(int devNum, int configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Number indicating the second A/D channel that the EXP board is connected to.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.SetRange1()

Sets the range/gain of the low 16 channels.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function SetRange1(ByVal devNum As Integer, ByVal configVal
As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo SetRange1(int devNum, int configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Sets the range (gain) of the low 16 channels.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.SetRange2()

Sets the range/gain of the high 16 channels.

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function SetRange2(ByVal devNum As Integer, ByVal configVal
As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo SetRange2(int devNum, int configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Sets the range (gain) of the high 16 channels.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ExpansionConfig.SetThermType()

Sets the type of thermocouple or RTD configuration for the board (J, K, E, T, R, S, and B types).

Member of the `cExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

Function prototype:

VB .NET:
```
Public Function SetThermType(ByVal devNum As Integer, ByVal
configVal As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo SetThermType(int devNum, int configVal)
```

Parameters:

`devNum`	Number of the expansion board.
`configVal`	Number that sets the type of thermocouple configured for the board. (J = 1, K = 2, T = 3, E = 4, R = 5, S = 6, B = 7, Platinum .00392 = 257, Platinum .00391 = 258, Platinum .00385 = 259, Copper .00427 = 260, Nickel/Iron .00581 = 261, Nickel/Iron .00527 = 262)

Returns:

An `ErrorInfo` object that indicates the status of the operation.

GetSignal()

Retrieves the configured Auxiliary or DAQ Sync connection and polarity for the specified timing and control signal.

This method is intended for advanced users. Except for the `SYNC_CLK` input, you can easily view the settings for the timing and control signals using *Insta*Cal.

Member of the `MccBoard` class.

Note: This method is not supported by all board types. Refer to the board-specific information contained in the *Universal Library User's Guide* (available in PDF format on our website at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Function prototype:

VB .NET:

```
Public Function GetSignal(ByVal direction As MccDaq.SignalDirection
, ByVal signalType As MccDaq.SignalType , ByVal index As Integer,
ByRef connectionPin As MccDaq.ConnectionPin , ByRef signalPolarity
As MccDaq.SignalPolarity ) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo GetSignal(MccDaq.SignalDirection direction ,
MccDaq.SignalType signalType, int index, out MccDaq.ConnectionPin
connectionPin, out MccDaq.SignalPolarity signalPolarity )
```

Parameters:

`direction` — Specifies whether retrieving the source (`MccDaq.SignalDirection.SignalIn`) or destination (`MccDaq.SignalDirection.SignalOut`).

`signalType` — Signal type whose connection is to be retrieved. Refer to "signalType parameter values" under the `SelectSignal()` method section on page 202 for valid signal types.

`index` — Used to indicate which connection to reference when there is more than one connection associated with the output `Signal` type. When querying output signals, increment this value until `BadIndex` is returned or 0 is returned via the `connection` parameter to determine all the output `connectionPin`s for the specified output `Signal`. The first `connectionPin` is indexed by 0.

For input signals (`direction= MccDaq.SignalDirection.SignalIn`), always set `index` to 0.

`connectionPin` — The specified connection is returned through this variable. Note that this is set to 0 if no connection is associated with the `signalType`, or if the `index` is set to an invalid value. Refer to "direction, connectionPin, and polarity parameter values" under the `SelectSignal()` method section on page 202 for expected return values.

`signalPolarity` — Holds the polarity for the associated `signalType` and connectionPin.

For output signals assigned an `AuxOut connectionPin`, the return value is either `MccDaq.SignalPolarity.Inverted` or `MccDaq.SignalPolarity.NonInverted`.

For `AdcConvert, DacUpdate, AdcTbSrc` and `DacTbSrc`, input signals, either `MccDaq.SignalPolarity.PositiveEdge` or `MccDaq.SignalPolarity.NegativeEdge` are returned.

All other `signals` return 0.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

The above timing and control configuration information can also be viewed and edited inside *Insta*Cal: Open *Insta*Cal, click on the board, and press the **Configure**... button or menu item. If the board supports DAQ Sync and Auxiliary Input/Output signal connections, a button labeled **Advanced Timing & Control Configuration** displays. Press this button to open a display for viewing and modifying the above timing and control signals.

NumBoards property

Returns the maximum number of boards you can install at one time.

Member of the `GlobalConfig` class.

Property prototype:

VB .NET: `Public Shared ReadOnly Property NumBoards As Integer`

C# .NET: `public int NumBoards [get]`

NumExpBoards property

Returns the maximum total number of expansion boards you can install.

Member of the `GlobalConfig` class.

Property prototype:

VB .NET: `Public Shared ReadOnly Property NumExpBoards As Integer`

C# .NET: `public static int NumExpBoards [get]`

SelectSignal()

Configures timing and control signals to use specific Auxiliary or DAQ Sync connections as a source or destination.

This method is intended for advanced users. Except for the `SyncClk` input, you can easily configure all the timing and control signals using *Insta*Cal.

Member of the `MccBoard` class.

SelectSignal is not supported by all boards

This method is not supported by all board types. Refer to the board-specific information contained in the *Universal Library User's Guide* (available in PDF format on our website at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Function prototype:

VB .NET:

```
Public Function SelectSignal(ByVal direction As
MccDaq.SignalDirection , ByVal signalType As MccDaq.SignalType,
ByVal connectionPin As MccDaq.ConnectionPin , ByVal polarity As
MccDaq.SignalPolarity ) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo SelectSignal(MccDaq.SignalDirection
direction, MccDaq.SignalType signal, MccDaq.ConnectionPin
connectionPin, MccDaq.SignalPolarity polarity)
```

Parameters:

`direction`

Direction of the specified signal type to be assigned a connector pin. For most signal types, this should be either `MccDaq.SignalDirection.SignalIn` or `MccDaq.SignalDirection.SignalOut`.

For the `SyncClk`, `AdcTbSrc` and `DacTbSrc` signals, the external source can also be disabled by specifying `Disabled`(=0), such that it is neither input nor output. Set it in conjunction with the `signalType`, `connectionPin`, and `polarity` arguments using the tables in the "direction, connectionPin, and polarity parameter values" starting on page 202.

`signalType`

Signal type to be associated with a connector pin. Set it to one of the constants in the "signalType parameter values" section on page 202.

`connectionPin`

Designates the connector pin to associate the signal type and direction. Since individual pin selection is not allowed for the DAQ-Sync connectors, all DAQ-Sync pin connections are referred to as DsConnector. The `MccDaq.ConnectionPin.AuxIn` and `MccDaq.ConnectionPin.AuxOut` settings match their corresponding hardware pin names.

`polarity`

`AdcTbSrc` and `DacTbSrc` input signals (direction = `MccDaq.SignalDirection.SignalIn`) can be set for either rising edge (`MccDaq.SignalPolarity.PositiveEdge`) or falling edge (`MccDaq.SignalPolarity.NegativeEdge`) signals. The `AuxOut` connections can be set to `MccDaq.SignalPolarity.Inverted` or `MccDaq.SignalPolarity.NonInverted` from their internal polarity.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

signalType parameter values:

All of the `signalType` settings are `MccDaq.SignalType` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `SignalType` enumeration (*variable* = `MccDaq.SignalType.AdcConvert`, *variable* = `MccDaq.SignalType.AdcGate`, etc.).

`AdcConvert`	A/D conversion pulse or clock.
`AdcGate`	External gate for A/D conversions.
`AdcScanClk`	A/D channel scan signal.
`AdcScanStop`	A/D scan completion signal.
`ADC_SSH`	A/D simultaneous sample and hold signal.
`AdcStartScan`	Start of A/D channel-scan sequence signal.
`AdcStartTrig`	A/D scan start trigger.
`AdcStopTrig`	A/D stop- or pre- trigger.
`AdcTbSrc`	A/D pacer timebase source.
`Ctr1Clk`	CTR1 clock source.
`Ctr2Clk`	CTR2 clock source.
`DacStartTrig`	D/A start trigger.
`DacTbSrc`	D/A pacer timebase source.
`DacUpdate`	D/A update signal.
`DGnd`	Digital ground.
`SyncClk`	STC timebase signal.

direction, connectionPin, and polarity parameter values:

All of the `direction` settings are `MccDaq.SignalDirection` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `SignalDirection` enumeration (*variable* = `MccDaq.SignalDirection.SignalIn`, *variable* = `MccDaq. SignalDirection.SignalOut`, etc.).

All of the `connectionPin` settings are `MccDaq.ConnectionPin` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `ConnectionPin` enumeration (*variable* = `MccDaq.ConnectionPin.AuxIn0`, *variable* = `MccDaq.ConnectionPin.DsConnector`, etc.).

All of the `polarity` settings are `MccDaq.SignalPolarity` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `SignalPolarity` enumeration (*variable* = `MccDaq.SignalPolarity.PositiveEdge`, *variable* = `MccDaq.ConnectionPin.Negative`, etc.).

Valid input (`direction= MccDaq.SignalDirection.SignalIn`) settings include:		
signalType	connectionPin	polarity
AdcConvert	AuxIn0 to AuxIn5 DsConnector	PositiveEdge or NegativeEdge
AdcGate	AuxIn0 to AuxIn5 DsConnector	See SetTrigger.
AdcStartTrig	AuxIn0 to AuxIn5 DsConnector	
AdcStopTrig	AuxIn0 to AuxIn5 DsConnector	
AdcTbSrc	AuxIn0 to AuxIn5	PositiveEdge or NegativeEdge
DacStartTrig	AuxIn0 to AuxIn5 DsConnector	Not assigned here.
DscTbSrc	AuxIn0 to AuxIn5	PositiveEdge or NegativeEdge
DacUpdate	AuxIn0 to AuxIn5 DsConnector	PositiveEdge or NegativeEdge
SyncClk	DsConnector	Not assigned here.

Valid output (`direction= MccDaq.SignalDirection.SignalOut`) settings include:		
signalType	connectionPin	polarity
AdcConvert	AuxIn0 to AuxIn5 DsConnector	Inverted* or NonInverted
AdcScanClk	AuxOut0..AuxOut2	
AdcScanStop	AuxOut0..AuxOut2	
AdcSsh	AuxOut0..AuxOut2 DsConnector	
AdcStartScan	AuxOut0..AuxOut2 DsConnector	
AdcStartTrig	AuxOut0..AuxOut2 DsConnector	
AdcStopTrig	AuxOut0..AuxOut2 DsConnector	
Ctr1Clk	AuxOut0..AuxOut2	
Ctr2Clk	AuxOut0..AuxOut2	
DacStartTrig	AuxOut0..AuxOut2 DsConnector	
DacUpdate	AuxOut0..AuxOut2 DsConnector	
DGND	AuxOut0..AuxOut2	Not assigned here.
SyncClk	DsConnector	
* Inverted is only valid for Auxiliary Output (AuxOut) connections.		

Valid disabled settings (`direction = MccDaq.SignalDirection.Disabled`):		
signalType	connectionPin	polarity
AdcTbSrc	Not assigned here.	Not assigned here.
DacTbSrc		
SyncClk		

Notes:

- You can view and edit the above timing and control configuration information from *Insta*Cal. Open *Insta*Cal, click on the board, and press the **Configure**... button or menu item. If the board supports DAQ

Sync and Auxiliary Input/Output signal connections, an **Advanced Timing & Control Configuration** button displays. Press that button to open a display for viewing and modifying the above timing and control signals.

- Except for the `AdcTbSrc`, `DacTbsSrc` and `SyncClk` signals, selecting an input signal connection does not necessarily activate it. Alternately, assigning an output signal to a connection does activate the signal upon performing the respective operation. For instance, when running an `ExtClock AInScan()`, `AdcConvert SignalIn` selects the connection to use as an external clock to pace the A/D conversions; if `AInScan()` is run without setting the `ExtClock` option, however, the selected connection is not activated and the signal at that connection is ignored. In both cases, the `AdcConvert` signal is output the connection(s) selected for the `AdcConvert SignalOut`. Since there are no scan options for enabling the Timebase Source and the `SyncClk`, selecting an input for the A/D or D/A Timebase Source, or `SyncClk` does activate the input source for the next respective operations.
- Multiple input signals can be mapped to the same `AuxIn` connection by successive calls to `SelectSignal()`; however, only one connection can be mapped to each input signal. If another connection had already been assigned to an input signal, the former selection is de-assigned and the new connection is assigned.
- Only one output signal can be mapped to the same AuxOut*n* connection; however, multiple connections can be mapped to the same output signal by successive calls to `SelectSignal()`. If an output signal had already been assigned to a connection, then the former output signal is de-assigned and the new output signal is assigned to the connection.
- When selecting `DsConnector` for a signal, only one `direction` per signal type can be defined at a given time. Attempting to assign both Directions of a signal to the `DsConnector` results in only the latest selection being applied. If the signal type had formerly been assigned an input direction from the `DsConnector`, assigning the output direction for that signal type results in the input signal being reassigned to its default connection.
- `Adc_Tb_Src` and `Dac_Tb_Src` are intended to synchronize the timebase of the analog input and output pacers across two or more boards. Internal calculations of sampling and update rates assume that the external timebase has the same frequency as its internal clock. Adjust sample rates to compensate for differences in clock frequencies.

  For instance, if the external timebase has a frequency of 10 MHz on a board that has a internal clock frequency of 40 MHz, the scan function samples or updates at a rate of about 1/4 the rate entered. However, while compensating for differences in external timebase and internal clock frequency, if the rate entered results in an invalid pacer count, the method returns a `BADRATE` error.

SetTrigger()

Selects the trigger source and sets up its parameters. This trigger is used to initiate analog to digital conversions using the following Universal Library for .NET functions:

- `AInScan()`, if the `ExTrigger` option is selected.
- `APretrig()`
- `FilePretrig()`

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function SetTrigger(ByVal trigType As MccDaq.TriggerType , ByVal lowThreshold As Short, ByVal highThreshold As Short) As MccDaq.ErrorInfo
```

```
Public Function SetTrigger(ByVal trigType As MccDaq.TriggerType, ByVal lowThreshold As System.UInt16, ByVal highThreshold As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo SetTrigger(MccDaq.TriggerType trigType, short lowThreshold, short highThreshold)
```

```
public MccDaq.ErrorInfo SetTrigger(MccDaq.TriggerType trigType, ushort lowThreshold, ushort highThreshold)
```

Parameters:

`trigType` Specifies the type of triggering based on the external trigger source. Set it to one of the constants in the "trigType parameter values" section below.

`lowThreshold` Selects the low threshold used when the trigger input is analog. The range depends upon the resolution of the trigger circuitry. Must be 0 to 255 for 8-bit trigger circuits, 0 to 4095 for 12-bit trigger circuits, and 0 to 65535 for 16-bit trigger circuits. Refer to the "Notes" section on page 206.

`highThreshold` Selects the high threshold used when the trigger input is analog. The range depends upon the resolution of the trigger circuitry. Must be 0 to 255 for 8-bit trigger circuits, 0 to 4095 for 12-bit trigger circuits, and 0 to 65535 for 16-bit trigger circuits. Refer to the "Notes" section on page 206.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

trigType parameter values:

All of the `trigType` settings are `MccDaq.TriggerType` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `TriggerType` enumeration (*variable* = `MccDaq.TriggerType.GateNegHys`, *variable* = `MccDaq.TriggerType.GatePosHys`, etc.).

Trigger Source	trigType	Explanation
Analog	`GateNegHys`	AD conversions are enabled when the external analog trigger input is more positive than `highThreshold`. AD conversions are disabled when the external analog trigger input more negative than Low/Threshold. Hysteresis is the level between Low/Threshold and `highThreshold`.
	`GatePosHys`	AD conversions are enabled when the external analog trigger input is more negative than `lowThreshold`. AD conversions are disabled when the external analog trigger input is more positive than `highThreshold`. Hysteresis is the level between `lowThreshold` and `highThreshold`.
	`GateAbove`	AD conversions are enabled as long as the external analog trigger input is more positive than `highThreshold`.
	`GateBelow`	AD conversions are enabled as long as the external analog trigger input is more negative than `lowThreshold`.
Analog	`TrigAbove`	AD conversions are enabled when the external analog trigger makes a transition from below `highThreshold` to above. Once conversions are enabled, the external trigger is ignored.
	`TrigBelow`	AD conversions are enabled when the external analog trigger input makes a transition from above `lowThreshold` to below. Once conversions are enabled, the external trigger is ignored.
	`GateInWindow`	AD conversions are enabled as long as the external analog trigger is inside the region defined by `lowThreshold` and `highThreshold`.
	`GateOutWindow`	AD conversions are enabled as long as the external analog trigger is outside the region defined by `lowThreshold` and `highThreshold`.
Digital	`GateHigh`	AD conversions are enabled as long as the external digital trigger input is 5 V (logic HIGH or 1).
	`GateLow`	AD conversions are enabled as long as the external digital trigger input is 0 V (logic LOW or 0).
	`TrigHigh`	AD conversions are enabled when the external digital trigger is 5 V (logic HIGH or '1'). Once conversions are enabled, the external trigger is ignored.
	`TrigLow`	AD conversions are enabled when the external digital trigger is 0 V (logic LOW or '0'). Once conversions are enabled, the external trigger is ignored.
	`TrigPosEdge`	AD conversions are enabled when the external digital trigger makes a transition from 0 V to 5 V (logic LOW to HIGH). Once conversions are enabled, the external trigger is ignored.
	`TrigNegEdge`	AD conversions are enabled when the external digital trigger makes a transition from 5 V to 0 V (logic HIGH to LOW). Once conversions are enabled, the external trigger is ignored.

Notes:

The value of the threshold must be within the range of the analog trigger circuit associated with the board. Refer to the board-specific information in the *Universal Library User's Guide*. For example, on the PCI-DAS1602/16, the analog trigger circuit handles ±10 V. A value of 0 corresponds to -10 V, whereas a value of 65535 corresponds to +10 V.

If you are using signed integer types, the thresholds range from -32768 to 32767 for 16-bit boards, instead of from 0 to 65535. In this case, the unsigned value of 65535 corresponds to a value of -1, 65534 corresponds to -2, …, 32768 corresponds to -32768.

For most boards that support analog triggering, you can pass the required trigger voltage level and the appropriate `Range` to `cbFromEngUnits/FromEngUnits` to calculate the `HighThreshold` and `LowThreshold` values.

For some boards (refer to the "Analog Input Boards" chapter in the *Universal Library User's Guide*), you must manually calculate the threshold by first calculating the least significant bit (LSB) for a particular range for the trigger resolution of your hardware. You then use the LSB to find the threshold in counts based on an analog voltage trigger threshold.

To calculate the threshold, do the following:

1. Calculate the LSB by dividing the full scale range (FSR) by $2^{resolution}$. FSR is the entire span from – FS to +FS of your hardware for a particular range. For example, the full scale range of ±10 V is 20 V.
2. Calculate how many times you need to add the LSB calculated in step 1 to the negative full scale (-FS) to reach the trigger threshold value.

The maximum threshold value is $2^{resolution} - 1$. The formula is shown here:

$$\text{Abs}(-FS - \text{threshold in volts}) \div (LSB) = \text{threshold in counts}$$

Here are two examples that use this formula—one for 8-bit trigger resolution and one for 12-bit trigger resolution.

- 8-bit example using the ±10 V range with a -5 V threshold:

  Calculate LSB: LSB = $20 \div 2^8 = 20 \div 256 = 0.078125$
  Calculate threshold: Abs(-10 - (-5)) ÷ .078125 = 5 ÷ 0.078125 = 64 (round this result if it is not an integer). A count of 64 translates to a voltage threshold of -5.0 V.

- 12-bit example using the ±10 V range with a +1 V threshold:

  Calculate LSB: LSB = $20 \div 2^{12} = 20 \div 4096 = 0.00488$
  Calculate threshold: Abs(-10 - 1) ÷ .00488 = 11 ÷ 0.00488 = 2254 (rounded from 2254.1). A count of 2254 translates to a voltage threshold of 0.99952 V.

Version property

This information is used by the library to determine compatibility.

Member of the `GlobalConfig class`.

Property prototype:

VB .NET:	`Public Shared ReadOnly Property Version As Integer`
C# .NET:	`public int Version [get]`

Counter Methods

Introduction

This section covers Universal Library methods that load, read, and configure counters. There are five types of counter chips used in MCC counter boards: 8254's, 8536's, 7266's, 9513's and generic event counters.. Some of the counter methods apply to only one type of counter.

C7266Config()

Configures 7266 counter for desired operation. This method can only be used with boards that contain a 7266 counter chip (Quadrature Encoder boards). For more information, refer to the LS7266R1 data sheet in accompanying ls7266r1.pdf file located in the "*Documents*" subdirectory of the installation.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function C7266Config(ByVal counterNum As Integer, ByVal
quadrature As MccDaq.Quadrature , ByVal countingMode As
MccDaq.CountingMode , ByVal dataEncoding As MccDaq.DataEncoding ,
ByVal indexMode As MccDaq.IndexMode , ByVal invertIndex As
MccDaq.OptionState , ByVal flagPins As MccDaq.FlagPins , ByVal
gateState As MccDaq.OptionState ) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo C7266Config(int counterNum,
MccDaq.Quadrature quadrature, MccDaq.CountingMode countingMode,
MccDaq.DataEncoding dataEncoding, MccDaq.IndexMode indexMode,
MccDaq.OptionState invertIndex, MccDaq.FlagPins flagPins,
MccDaq.OptionState gateState)
```

Parameters:

`counterNum`	Counter Number (1 - n), where n is the number of counters on the board.
`quadrature`	Selects the resolution multiplier for quadrature input, or disables quadrature input (`NoQuad`) so that the counters can be used as standard TTL counters. `NoQuad`, `X1Quad`, `X2Quad` or `X4Quad`.
`countingMode`	Selects operating mode for the counter. NormalMode, RangeLimit, NoRecycle, ModuloN. Set it to one of the constants in the "countingMode" section below.
`dataEncoding`	Selects the format of the data that is returned by the counter - either Binary or BCD format. `BinaryCount` or `BCDCount`.
`indexMode`	Selects which action will be taken when the Index signal is received. The `IndexMode` must be set to `IndexDisabled` whenever a `Quadrature` is set to NOQuad or when `GateState` is set to `Enabled`. Set it to one of the constants in the "indexMode" section on page 211.
`invertIndex`	Selects the polarity of the Index signal. If set to Disabled, the Index signal is assumed to be positive polarity. If set to Enabled, the Index signal is assumed to be negative polarity.
`flagPins`	Selects which signals will be routed to the FLG1 and FLG2 pins. Set it to one of the constants in the "flagPins" section on page 211.
`gateState`	If `gateState` is set to `Enabled`, then the RCNTR pin will be used as a gating signal for the counter. Whenever `gateState` =`Enabled`, the `indexMode` must be set to `IndexDisabled`.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

countingMode parameter values:

All of the `countingMode` settings are `MccDaq.CountingMode` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `CountingMode` enumeration (*variable* = `MccDaq.CountingMode.NormalMode`, *variable* = `MccDaq.CountingMode.NormalMode`, `CountingMode.RangeLimit`, etc.).

`NormalMode`	Each counter operates as a 24-bit counter that rolls over to 0 when the maximum count is reached.
`RangeLimit`	In range limit count mode, an upper an lower limit is set, mimicking limit switches in the mechanical counterpart. The upper limit is set by loading the PRESET register with the `CLoad()` method after the counter has been configured. The lower limit is always 0. When counting up, the counter freezes whenever the count reaches the value that was loaded into the PRESET register. When counting down, the counter freezes at 0. In either case the counting is resumed only when the count direction is reversed.
`NoRecycle`	In non-recycle mode, the counter is disabled whenever a count overflow or underflow takes place. The counter is re-enabled when a reset or load operation is performed on the counter.
`ModuloN`	In `ModuloN` mode, an upper limit is set by loading the PRESET register with a maximum count. Whenever counting up, when the maximum count is reached, the counter will roll-over to 0 and continue counting up. Likewise when counting down, whenever the count reaches 0, it will roll over to the maximum count (in the PRESET register) and continue counting down.

indexMode parameter values:

All of the `indexMode` settings are `MccDaq.IndexMode` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `IndexMode` enumeration (*variable* = `MccDaq.IndexMode.IndexDisabled`, *variable* = `MccDaq.IndexMode.LoadCtr`, etc.).

`IndexDisabled`	The Index signal is ignored.
`LoadCtr`	The counter is loaded whenever the Index signal ON the LCNTR pin occurs.
`LoadOutLatch`	The current count is latched whenever the Index signal on the LCNTR pin occurs. When selected, the `CIn()` method returns the same count each time it is called until the Index signal occurs.
`ResetCtr`	The counter is reset to 0 whenever the Index signal on the RCNTR pin occurs.

flagPins parameter values:

All of the `flagPins` settings are `MccDaq.FlagPins` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `FlagPins` enumeration (*variable* = `MccDaq.FlagPins,CarryBorrow`, *variable* = `MccDaq.FlagPins.CompareBorrow`, etc.).

`CarryBorrow`	FLG1 pin is Carry output, FLG2 is Borrow output.
`CompareBorrow`	FLG1 pin is Compare output, FLG2 is Borrow output.
`CarryBorrowUpDown`	FLG1 pin is Carry/Borrow output, FLG2 is `Up/Down` signal.
`IndexError`	FLG1 is Index output, FLG2 is Error output.

C8254Config()

Configures 8254 counter for desired operation. This method can only be used with 8254 counters. For more information, see the 82C54 data sheet in accompanying 82C54.pdf file located in the "*Documents*" subdirectory of the installation.

Member of the `MccBoard` class.

Function prototype:

VB .NET:
```
Public Function C8254Config(ByVal counterNum As Integer, ByVal
config As MccDaq.C8254Mode ) As MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo C8254Config(int counterNum, MccDaq.C8254Mode
config)
```

Parameters:

`counterNum`	Selects one of the counter channels. An 8254 has 3 counters. The value may be 1 - n, where n is the number of 8254 counters on the board (refer to board-specific info in the).
`config`	Refer to the 8254 data sheet for a detailed description of each of the configurations. Set it to one of the constants in the "config" section below.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

config parameter values:

All of the `config` settings are `MccDaq.C8254Mode` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the C`8254Mode` enumeration (*variable* = `MccDaq.`C`8254Mode.HighOnLastCount`, *variable* = `MccDaq.C8254Mode.LastShot`, etc.).

`HighOnLastCount`	Output of counter (OUT N) transitions from low to high on terminal count and remains high until reset. See Mode 0 in the 8254 data sheet in accompanying 82C54.pdf file located in the *Documents* subdirectory of the installation.
`OneShot`	Output of counter (OUT N) transitions from high to low on rising edge of GATE N, then back to high on terminal count. See mode 1 in the 8254 data sheet in accompanying 82C54.pdf file located in the *Documents* subdirectory of the installation.
`RateGenerator`	Output of counter (OUT N) pulses low for one clock cycle on terminal count, reloads counter and recycles. See mode 2 in the 8254 data sheet in accompanying 82C54.pdf file located in the *Documents* subdirectory of the installation.
`SquareWave`	Output of counter (OUT N) is high for count < 1/2 terminal count then low until terminal count, whereupon it recycles. This mode generates a square wave. See mode 3 in the 8254 data sheet in the accompanying 82C54.pdf file located in the *Documents* subdirectory of the installation.
`SoftWareStrobe`	Output of counter (OUT N) pulses low for one clock cycle on terminal count. Count starts after counter is loaded. See mode 4 in the 8254 data sheet in the accompanying 82C54.pdf file located in the *Documents* subdirectory of the installation.
`HardwareStrobe`	Output of counter (OUT N) pulses low for one clock cycle on terminal count. Count starts on rising edge at GATE N input. See mode 5 in the 8254 data sheet in accompanying 82C54.pdf file located in the *Documents* subdirectory of the installation.

C8536Config()

Configures 8536 counter for desired operation. This method can only be used with 8536 counters.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function C8536Config(ByVal counterNum As Integer, ByVal
outputControl As MccDaq.C8536OutputControl , ByVal recycleMode As
MccDaq.RecycleMode , ByVal retrigger As MccDaq.OptionState ) As
MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo C8536Config(int counterNum,
MccDaq.C8536OutputControl outputControl, MccDaq.RecycleMode
recycleMode, MccDaq.OptionState retrigger)
```

Parameters:

`counterNum`	Selects one of the counter channels. An 8536 has three counters. The value may be 1, 2 or 3.
`outputControl`	Specifies the action of the output signal. Set it to one of the constants in the "outputControl" section below.
`recycleMode`	If set to `Recycle` (as opposed to `OneTime`), the counter automatically reloads to the starting count every time it reaches 0, and then counting continues.
`retrigger`	If set to `Enabled`, every trigger on the counter's trigger input will initiate loading of the initial count and counting will proceed from initial count.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

outputControl parameter values:

All of the `outputControl` settings are `MccDaq.C8536OutputControl` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `C8536OutputControl` enumeration (*variable* = `MccDaq.C8536OutputControl.HighPulseOnTc`, *variable* = `MccDaq.C8536OutputControl.ToggleOnTc`, etc.).

`HighPulseOnTc`	Output transitions from low to high for one clock pulse on terminal count.
`ToggleOnTc`	Output changes state on terminal count.
`HighUntilTc`	Output transitions to high at the start of counting, and then goes low on terminal count.

C8536Init()

Initializes the counter linking features of an 8536 counter chip. See the 8536 data sheet "Counter/Timer Link Controls" section for a complete description of the hardware affected by this mode. The linking of counters 1 and 2 must be accomplished prior to enabling the counters.

Member of the `MccBoard` class.

Function prototype:

VB .NET:	`Public Function C8536Init(ByVal chipNum As Integer, ByVal ctr1Output As MccDaq.CtrlOutput ) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo C8536Init(int chipNum, MccDaq.CtrlOutput ctr1Output)`

Parameters:

`chipNum`	Selects one of the 8536 chips on the board, 1 to *n*.
`ctrlOutput`	Specifies how the counter 1 is to be linked to counter 2, if at all. Set it to one of the constants in the "ctrlOutput" section below.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

ctrlOutput parameter values:

All of the `ctrlOutput` settings are `MccDaq.CtrlOutput` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `CtrlOutput` enumeration (*variable* = `MccDaq.CtrlOutput.NotLinked`, *variable* = `MccDaq.CtrlOutput.GateCtr2`, etc.).

`NotLinked`	Counter 1 is not connected to any other counter's inputs.
`GateCtr2`	Output of counter 1 is connected to the GATE of counter #2.
`TrigCtr2`	Output of counter 1 is connected to the trigger of counter #2.
`InCtr2`	Output of counter 1 is connected to counter #2 clock input.

C9513Config()

Sets all of the configurable options of a 9513 counter. For more information, see the AM9513A data sheet in accompanying 9513A.pdf file located in the *Documents* subdirectory of the installation.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function C9513Config(ByVal counterNum As Integer, ByVal
gateControl As MccDaq.GateControl , ByVal counterEdge As
MccDaq.CountEdge , ByVal counterSource As MccDaq.CounterSource ,
ByVal specialGate As MccDaq.OptionState , ByVal reload As
MccDaq.Reload , ByVal recycleMode As MccDaq.RecycleMode , ByVal
bcdMode As MccDaq.BCDMode , ByVal countDirection As
MccDaq.CountDirection , ByVal outputControl As
MccDaq.C9513OutputControl ) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo C9513Config(int counterNum,
MccDaq.GateControl gateControl, MccDaq.CountEdge counterEdge,
MccDaq.CounterSource counterSource, MccDaq.OptionState specialGate,
MccDaq.Reload reload, MccDaq.RecycleMode recycleMode, MccDaq.BCDMode
bcdMode, MccDaq.CountDirection countDirection,
MccDaq.C9513OutputControl outputControl)
```

Parameters:

`counterNum`	Counter number (1 - n) where n is the number of counters on the board. (For example, a CIO-CTR5 has 5, a CIO-CTR10 has 10, etc. See board specific info).
`gateControl`	Sets the gating response for level, edge, etc. Set it to one of the constants in the "gateControl parameter values" section on page 216.
`counterEdge`	Which edge to count. Referred to as "Source Edge" in 9513 data book. Can be set to `POSITIVEEDGE` (count on rising edge) or `NEGATIVEEDGE` (count on falling edge).
`counterSource`	Each counter may be set to count from one of 16 internal or external sources. Set it to one of the constants in the "counterSource parameter values" section on page 216.
`specialGate`	Special gate may be enabled (`MccDaq.OptionState.Enabled`) or disabled (`MccDaq.OptionState.Disabled`).
`reload`	Reload the counter from the load register (`reload` = `MccDaq.Reload.LoadReg`) or alternately load from the load register, then the hold register (`reload` = `MccDaq.Reload.LoadAndHoldReg`).
`recycleMode`	Execute once (`MccDaq.RecycleMode.OneTime`) or reload and recycle (`MccDaq.RecycleMode.Recycle`).
`bcdMode`	Counter may operate in *binary coded decimal* count (`MccDaq.BCDMode.BCDCount`) or *binary* count (`MccDaq.BCDMode.BinaryCount`).
`countDirection`	AM9513 may count up (`MccDaq.CountDirection.CountUp`) or down (`MccDaq.CountDirection.CountDown`).
`outputControl`	The type of output desired. Set it to one of the constants in the "

outputControl parameter values" section on page 217.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

gateControl parameter values:

All of the `gateControl` settings are `MccDaq.GateControl` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `GateControl` enumeration (*variable* = `MccDaq.GateControl.NoGate`, *variable* = `MccDaq.GateControl.AhlTcPrevCtr`, etc.).

`NoGate`	No gating
`AhlTcPrevCtr`	Active high TCN -1
`AhlNextGate`	Active High Level GATE N + 1
`AhlPrevGate`	Active High Level GATE N - 1
`AhlGate`	Active High Level GATE N
`AllGate`	Active Low Level GATE N
`AheGate`	Active High Edge GATE N
`Alegate`	Active Low Edge GATE N

counterSource parameter values:

All of the `counterSource` settings are `MccDaq.CounterSource` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `CounterSource` enumeration (*variable* = `MccDaq.CounterSource.TcPrevCtr`, *variable* = `MccDaq.CounterSource.CtrInput1`, etc.).

`TcPrevCtr`	TCN - 1 (Terminal count of previous counter)
`CtrInput1`	SRC 1 (Counter Input 1)
`CtrInput2`	SRC 2 (Counter Input 2)
`CtrInput3`	SRC 3 (Counter Input 3)
`CtrInput4`	SRC 4 (Counter Input 4)
`CtrInput5`	SRC 5 (Counter Input 5)
`Gate1`	GATE 1
`Gate2`	GATE 2
`Gate3`	GATE 3
`Gate4`	GATE4
`Gate5`	GATE 5
`Freq1`	F1
`Freq2`	F2
`Freq3`	F3
`Freq4`	F4
`Freq5`	F5

outputControl parameter values:

All of the `outputControl` settings are `MccDaq.9513OutputControl` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `9513OutputControl` enumeration (*variable* = `MccDaq.9513OutputControl.AlwaysLow`, *variable* = `MccDaq.9513OutputControl.HighPulseOnTc`, etc.).

`AlwaysLow`	Inactive, Output Low
`HighPulseOnTc`	High pulse on Terminal Count
`ToggleOnTc`	TC Toggled
`Disconnected`	Inactive, Output High Impedance
`LowPulseOnTc`	Active Low Terminal Count Pulse
`3, 6, 7`	(numeric values) Illegal

Notes:

The information provided here and in `C9513Init()` will only help you understand how Universal Library syntax corresponds to the 9513 data sheet (refer to the accompanying 9513A.pdf file located in the *Documents* subdirectory of the installation). It is not a substitute for the data sheet. You cannot program and use a 9513 counter/timer without the data sheet.

C9513Init()

Initializes all of the chip-level features of a 9513 counter chip. This method can only be used with 9513 counters. For more information, refer to the AM9513A data sheet in accompanying 9513A.pdf file located in the *Documents* subdirectory of the installation.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function C9513Init(ByVal chipNum As Integer, ByVal
foutDivider As Integer, ByVal foutSource As MccDaq.CounterSource,
ByVal compare1 As MccDaq.CompareValue , ByVal compare2 As
MccDaq.CompareValue , ByVal timeOfDay As MccDaq.TimeOfDay ) As
MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo C9513Init(int chipNum, int foutDivider,
MccDaq.CounterSource foutSource, MccDaq.CompareValue compare1,
MccDaq.CompareValue compare2, MccDaq.TimeOfDay timeOfDay)
```

Parameters:

`chipNum`	Specifies which 9513 chip is to be initialized. For a CTR05 board this should be set to 1. For a CTR10 board it should be either 1 or 2, and for a CTR20 it should be 1-4.
`foutDivider`	F-Out divider (0-15). If set to 0, `foutDivider` is the rate of `foutSource` divided by 16. If set to a number between 1 ands 15, `foutDivider` is the rate of `foutSource` divided by `foutDivider`.
`foutSource`	Specifies source of the signal for F-Out signal. Set it to one of the constants in the "foutSource parameter values" section on page 219.
`compare1`	`MccDaq.CompareValue.Enabled` or `MccDaq.CompareValue.Disabled`
`compare2`	`MccDaq.CompareValue.Enabled` or `MccDaq.CompareValue.Disabled`.
`timeOfDay`	MccDaq.TimeOfDay.`Disabled`, or three different enabled settings. Set it to one of the constants in the "timeOfDay " section on page 219.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

foutSource parameter values:

All of the `foutSource` settings are `MccDaq.CounterSource` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `CounterSource` enumeration (*variable* = `MccDaq.CounterSource.CtrInout1,` *variable* = `MccDaq.CounterSource.CtrInput2`, etc.).

foutSource	9513 Data Sheet Equivalent	foutSource	9513 Data Sheet Equivalent
CtrInput1	SRC 1 (Counter Input 1)	Gate3	GATE3
CtrInput2	SRC 2 (Counter Input 2)	Gate4	GATE4
CtrInput3	SRC 3 (Counter Input 3)	Gate5	GATE5
CtrInput4	SRC 4 (Counter Input 4)	Freq1	F1
CtrInput5	SRC 5 (Counter Input 5)	Freq2	F2
Gate1	GATE1	Freq3	F3
Gate2	GATE2	Freq4	F4

timeOfDay parameter values:

All of the `timeOfDay` settings are `MccDaq.TimeOfDay` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `TimeOfDay` enumeration (*variable* = `MccDaq.TimeOfDay.Disable,` *variable* = `MccDaq.TimeOfDay.One`, etc.).

timeOfDay	9513 Data Sheet Equivalent
`Disabled`	TOD Disabled
`One`	TOD Enabled/5 Input
`Two`	TOD Enabled/6 Input
`Three`	TOD Enabled/10 Input

No parameters for	9513 Data Sheet Equivalent
0 (FOUT on)	FOUT Gate
0 (Data bus matches board)	Data Bus Width
1 (Disable Increment)	Data Pointer Control
1 (BCD Scaling)	Scalar Control

Notes:

The information provided here and in `C9513Config()` will only help you understand how Universal Library for .NET syntax corresponds to the 9513 data sheet (refer to the accompanying 9513A.pdf file located in the *Documents* subdirectory of the installation). It is not a substitute for the data sheet. You cannot program and use a 9513 counter/timer without the data sheet.

CFreqIn()

Measures the frequency of a signal. This method can only be used with 9513 counters. This method uses internal counters #5 and #4.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function CFreqIn(ByVal signalSource As MccDaq.SignalSource , ByVal gateInterval As Integer, ByRef count As Short, ByRef freq As Integer) As MccDaq.ErrorInfo
```

```
Public Function CFreqIn(ByVal signalSource As MccDaq.SignalSource, ByVal gateInterval As Integer, ByRef count As System.UInt16, ByRef freq As Integer) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo CFreqIn(MccDaq.SignalSource signalSource, int gateInterval, out short count, out int freq)
```

```
public MccDaq.ErrorInfo CFreqIn(MccDaq.SignalSource signalSource, int gateInterval, out ushort count, out int freq
```

Parameters:

`signalSource`

Specifies the source of the signal to calculate the frequency from.

The signal to be measured is routed internally from the source specified by `signalSource` to the clock input of counter 5. On boards with more than one 9513 chip, there is more than one counter 5. Which counter 5 is used is also determined by SigSource. Set it to one of the constants in the "signalSource parameter values" section on page 221.

The value of `signalSource` determines which chip will be used. `CtrInput6` through `CtrInput10`, `Freq6` through `Freq10` and `Gate6` through `Gate9` indicate chip two will be used. The signal to be measured must be present at the chip two input specified by SigSource.

Note: The gating connection from counter 4 output to counter 5 gate must be made between counters 4 and 5 of *this chip* (see below). Refer to board-specific information to determine valid values for your board.

`gateInterval`

Gating interval in milliseconds (must be > 0). Specifies the time, in milliseconds, that the counter will count. The optimum `gateInterval` depends on the frequency of the measured signal. The counter can count up to 65535. If the gating interval is too low, then the count will be too low and the resolution of the frequency measurement will be poor. For example, if the count changes from 1 to 2 the measured frequency doubles.

If the gating interval is too long, the counter will overflow and a `FreqOverFlow` error will occur.

This method will not return until the `gateInterval` has expired. There is no background option. Under Windows, this means that window activity will stop for the duration of the call. Adjust the `gateInterval` so this does not pose a problem to your user interface.

`count`

The raw count.

`freq`

The measured frequency in Hz.

Returns:

An ErrorInfo object that indicates the status of the operation.

Count - Count that the frequency calculation is based on.

Freq - Measured frequency in Hz

signalSource parameter values:

All of the signalSource settings are MccDaq.SignalSource enumerated constants. To set a variable to one of these constants, you must refer to the MccDaq object and the SignalSource enumeration (*variable* = MccDaq.SignalSource.CtrInput1, *variable* = MccDaq.SignalSource.Gate1, etc.).

One 9513 chip (Chip 1 used):

- CtrInput1 through CtrInput5
- Gate1 through Gate4
- Freq1 through Freq5

Two 9513 chips (Chip 1 or Chip 2 used):

- CtrInput1 through CtrInput10
- Gate1 through Gate9 (excluding Gate5)
- Freq1 through Freq10

Four 9513 chips (Chips 1- 4 may be used):

- CtrInput1 through CtrInput20
- Gate1 through Gate19 (excluding gates 5, 10 & 15)
- Freq1 through Freq20

Notes:

- This method requires an electrical connection between counter 4 output and counter 5 gate. This connection must be made between counters 4 and 5 on the chip specified by signalSource.
- C9513Init() must be called for each chipNum that will be used by this method. The values of foutDivider, foutSource, compare1, compare2, and timeOfDay are irrelevant to this method and may be any value shown in the C9513Init() method description.
- If you select an external clock source for the counters, the gateInterval, count, and freq settings are only valid if the external source is 1 MHz. Otherwise, you need to scale the values according to the frequency of the external clock source.

  For example, for an external clock source of 2 MHz, increase your gateInterval setting by a factor of 2, and also double the count and freq values returned when analyzing your results.

CIn()

Reads the current count from a counter.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function CIn(ByVal counterNum As Integer, ByRef count As Short) As MccDaq.ErrorInfo
Public Function CIn(ByVal counterNum As Integer, ByRef count As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo CIn(int counterNum, out ushort count)
public MccDaq.ErrorInfo CIn(int counterNum, out short count)
```

Parameters:

`counterNum`	The counter to read current count from. Valid values are 1 to 20, up to the number of counters on the board.
`count`	Counter value returned here.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

`count`: Refer to your BASIC manual for information on BASIC integer data types. -32,768 to 32,767 for BASIC languages. BASIC reads counters as:

- -1 reads as 65535
- -21768 reads as 32768
- 32767 reads as 32767
- 2 reads as 2
- 0 reads as 0

CIn() vs. CIn32(): Although the `CIn()` and `CIn32()` methods perform the same operation, `CIn32()` is the preferred method to use.

The only difference between the two is that `CIn()` returns a 16-bit count value and `CIn32()` returns a 32-bit value. Both `CIn()` and `CIn32()` can be used, but `CIn32()` is required whenever you need to read count values greater than 16 bits (counts > 65535).

CIn32()

Reads the current count from a counter, and returns it as a 32 bit integer.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function CIn32(ByVal counterNum As Integer, ByRef count As Integer) As MccDaq.ErrorInfo
```

```
Public Function CIn32(ByVal counterNum As Integer, ByRef count As System.UInt32) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo CIn32(int counterNum, out uint count)
```

```
public MccDaq.ErrorInfo CIn32(int counterNum, out int count)
```

Parameters:

`counterNum`	The counter to read current count from. Valid values are 1 to *n*, where *n* is the number of counters on the board.
`count`	Current count value from selected counter.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

CIn() vs. CIn32(): Although the `CIn()` and `CIn32()` methods perform the same operation, `CIn32()` is the preferred method to use.

The only difference between the two is that `CIn()` returns a 16-bit count value and `CIn32()` returns a 32-bit value. Both `CIn()` and `CIn32()` can be used, but `CIn32()` is required whenever you need to read count values greater than 16 bits (counts > 65535).

CLoad()

Loads the specified counter's `Load`, `Hold`, `Alarm`, `QuadCount`, `QuadPreset` or `PreScaler` register with a count. When loading a counter with a starting value, it is never loaded directly into the counter's count register. Rather, it is loaded into the load or hold register. From there, the counter, after being enabled, loads the count from the appropriate register, generally on the first valid pulse.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function CLoad(ByVal regNum As MccDaq.CounterRegister, ByVal loadValue As Integer) As MccDaq.ErrorInfo
Public Function CLoad(ByVal regNum As MccDaq.CounterRegister, ByVal loadValue As System.UInt32) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo CLoad(MccDaq.CounterRegister regNum, uint loadValue)
public MccDaq.ErrorInfo CLoad(MccDaq.CounterRegister regNum, int loadValue)
```

Parameters:

`regNum`	The register to load the count to. Set it to one of the constants in the "regNum parameter values" section below.
`loadValue`	The value to be loaded. This value must be between 0 and $2^{resolution}$-1 of the counter. Refer to the discussion of Basic signed integers in the "16-bit values using a signed integer data type" section in the "Universal Library Description & Use" chapter of the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Returns:

An `ErrorInfo` object that indicates the status of the operation.

regNum parameter values:

All of the `regNum` settings are `MccDaq.CounterRegister` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `CounterRegister` enumeration (*variable* = `MccDaq.CounterRegister.LoadReg1`, *variable* = `MccDaq.CounterRegister.HoldReg1`, etc.).

`LoadReg1 … 20`	Load registers 1 to 20. Can span many chips.
`HoldReg1 … 20`	Hold registers 1 to 20. Can span several chips. (9513 only)
`Alarm1Chip1`	Alarm register 1 of the first counter chip. (9513 only)
`Alarm2Chip1`	Alarm register 2 of the first counter chip. (9513 only)
`Alarm1Chip2`	Alarm register 1 of the 2nd counter chip. (9513 only)
`Alarm2Chip2`	Alarm register 2 of the 2nd counter chip. (9513 only)
`Alarm1Chip3`	Alarm register 1 of the third counter chip. (9513 only)
`Alarm2Chip3`	Alarm register 2 of the third counter chip. (9513 only)
`Alarm1Chip4`	Alarm register 1 of the four counter chip. (9513 only)
`Alarm2Chip4`	Alarm register 2 of the four counter chip. (9513 only)
`QuadCount1` to `QuadCount4`	Current Count (LS7266 only)
`QuadPreset1` to `QuadPreset4`	Preset register (LS7266 only)

`QuadPrescaler1` to `QuadPrescaler4`	Prescaler register (LS7266 only)

Notes:

You cannot load a count-down-only counter with less than 2.

Counter types: There are several counter types supported. Please refer to the data sheet for the registers available for a counter type.

CLoad() vs. CLoad32(): The `CLoad()` and `CLoad32()` perform the same operation. These methods differ in that `CLoad()` loads a 16-bit count value, while `CLoad32()` loads a 32-bit value. The only time you need to use `CLoad32()` is to load counts that are larger than 32 bits (counts > 65535).

CLoad32()

Loads the specified counter's COUNT, PRESET or PRESCALER register with a count.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function CLoad32(ByVal regNum As MccDaq.CounterRegister , ByVal loadValue As Integer) As MccDaq.ErrorInfo
```

```
Public Function CLoad32(ByVal regNum As MccDaq.CounterRegister, ByVal loadValue As System.UInt32) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo CLoad32(MccDaq.CounterRegister regNum, uint loadValue)
```

```
public MccDaq.ErrorInfo CLoad32(MccDaq.CounterRegister regNum, int loadValue)
```

Parameters:

`regNum`	The register to load the value into. Set it to one of the constants in the "regNum parameter values" section below.
`loadValue`	The value to be loaded into `regNum`.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

regNum parameter values:

All of the `regNum` settings are `MccDaq.CounterRegister` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `CounterRegister` enumeration (*variable* = `MccDaq.CounterRegister.LoadReg1`, *variable* = `MccDaq.CounterRegister.HoldReg1`, etc.).

`LoadReg1 … 20`	Load registers 1 to 20. Can span many chips.
`HoldReg1 … 20`	Hold registers 1 to 20. Can span several chips. (9513 only)
`Alarm1Chip1`	Alarm register 1 of the first counter chip. (9513 only)
`Alarm2Chip1`	Alarm register 2 of the first counter chip. (9513 only)
`Alarm1Chip2`	Alarm register 1 of the 2nd counter chip. (9513 only)
`Alarm2Chip2`	Alarm register 2 of the 2nd counter chip. (9513 only)
`Alarm1Chip3`	Alarm register 1 of the third counter chip. (9513 only)
`Alarm2Chip3`	Alarm register 2 of the third counter chip. (9513 only)
`Alarm1Chip4`	Alarm register 1 of the four counter chip. (9513 only)
`Alarm2Chip4`	Alarm register 2 of the four counter chip. (9513 only)
`QuadCount1` to `QuadCount4`	Used to initialize the counter
`QuadPreset1` to `QuadPreset4`	Used to set upper limit of counter in some modes.
`QuadPrescaler1` to `QuadPrescaler4`	Used for clock filtering (valid values: 0 to 255).

Notes:

CLoad() vs. CLoad32(): Although the `CLoad()` and `CLoad32()` methods perform the same operation, `CLoad32()` is the preferred method to use.

The only difference between the two is that `CLoad()` loads a 16-bit count value, and `CLoad32()` loads a 32-bit value. The only time you need to use `CLoad32()` is to load counts that are larger than 32 bits (counts > 65535).

CStatus()

Returns status information about the specified counter (7266 counters only)

Function prototype:

VB .NET:	`Public Function CStatus(ByVal counterNum As Integer, ByRef statusBits As MccDaq.StatusBits ) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo CStatus(int counterNum, out MccDaq.StatusBits statusBits)`

Parameters:

`counterNum`	The counter to read current count from. Valid values are 1 to *n*, where *n* is the number of counters on the board.
`statusBits`	Current status from selected counter is returned here. The status consists of individual bits that indicate various conditions within the counter. Set it to one of the constants in the "statusBits parameter values" section below.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

All of the `statusBits` settings are `MccDaq.StatusBits` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `StatusBits` enumeration (*variable* = `MccDaq.StatusBits.UnderFlow`, *variable* = `MccDaq.StatusBits.Overflow`, etc.).

statusBits parameter values:

`Underflow`	set to 1 whenever the count decrements past 0. Is cleared to 0 whenever `CStatus()` is called.
`Overflow`	Set to 1 whenever the count increments past it's upper limit. Is cleared to 0 whenever `CStatus()` is called.
`Compare`	Set to 1 whenever the count matches the preset register. Is cleared to 0 whenever `CStatus()` is called.
`Sign`	Set to 1 when the MSB of the count is 1. Is cleared to 0 whenever the MSB of the count is set to 0.
`Error`	Set to 1 whenever an error occurs due to excessive noise on the input. Is cleared to 0 by calling `C7266Config()`.
`UpDown`	Set to 1 when counting up. Is cleared to 0 when counting down
`Index`	Set to 1 when index is valid. Is cleared to 0 when index is not valid.

CStoreOnInt()

Installs an interrupt handler that will store the current count whenever an interrupt occurs. This method can only be used with 9513 counters. This method will continue to operate in the background until either IntCount has been satisfied or `StopBackground()` is called.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function CStoreOnInt(ByVal intCount As Integer, ByRef
cntrControl As MccDaq.CounterControl , ByVal memHandle As Integer)
As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo CStoreOnInt(int intCount, ref
MccDaq.CounterControl cntrControl, int memHandle)
```

Parameters:

`intCount` — The counters will be read every time an interrupt occurs, until IntCount number of interrupts have occurred. If `intCount` = 0, the method will run until `StopBackground()` is called. (refer to `memHandle` below).

`cntrControl` — The array should have an element for each counter on the board. (5 elements for CTR-05 board, 10 elements for a CTR-10, etc.). Each element corresponds to a counter channel. Each element should be set to either `MccDaq.CounterControl.Disabled` or `MccDaq.CounterControl.Enabled`. All channels set to `MccDaq.CounterControl.Enabled` will be read when an interrupt occurs.

`memHandle` — The handle for the Windows buffer. Counts are stored in an array. The array should have an element for each counter on the board. (5 elements for CTR-05 board, 10 elements for a CTR-10, etc.). Each element corresponds to a counter channel. Each channel that is marked as Enabled in the CntrControl array will be read when an interrupt occurs. The count value will be stored in the DataBuffer element associated with that channel.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

If the library revision is set to 4.0 or greater, the following code changes are required:

- If `intCount` is non-zero, the `countData` array must be allocated to (`intCount` * Number of Counters).
- For example, if `intCount` is set to 100 for a CTR-05 board, then the `countData` array must be declared with a size of (100 * 5) = 500. This new functionality keeps the user application from having to move the data out of the countData buffer for every interrupt, before it is overwritten. Now, for each interrupt the counter values will be stored in adjacent memory locations in the `countData` array.

Allocate the proper array size for non-zero `IntCount` settings

Specifying `intCount` as a non-zero value and failing to allocate the proper sized array results in a runtime error. There is no way for the Universal Library to determine if the array has been allocated with the proper size.

- If `intCount` = 0, the functionality is unchanged.

Digital I/O Methods

Introduction

Use the methods explained in this chapter to read and set digital values. Most digital ports are configurable, while some others are non-configurable. Some types of hardware allow readback of the values that output ports are set to on configurable port types. Devices using 8255 chips for digital I/O are one example. For these devices, input methods such as `DIn()` are valid for ports configured as output.

Use the tables below to determine the port number, bit number, and actual addresses being set by the digital I/O methods. Table 17-1 relates the port number (`portNum`) to the port address and the 8255 port. Table 17-2 relates the bit number to the 8255 chip on the board.

Table 17-1. Port Numbers and Corresponding Port Address, 8255 Port Number

Mnemonic	Bit No.	8255 Port No.	Port Address	8536 Port No.	Port Address
`FirstPortA`	0 - 7	1A	Base + 0	1A	Base + 0
`FirstPortB`	8 - 15	1B		1B	
`FirstPortCL`	16 - 19	1CL		1C	
`FirstPortCH`	20 - 23	1CH		Not present	
`SecondPortA`	24 - 31	2A	Base + 4	2A	Base + 4
`SecondPortB`	32 - 39	2B		2B	
`SecondPortCL`	40 - 43	2CL		2C	
`SecondPortCH`	44 - 47	2CH		Not present	
and so on, to the last chip on the board as: `ThirdPort`*x*, `FourthPort`*x*, `FifthPort`*x*, `SixthPort`*x*, and `SeventhPort`*x*					
`EighthPortA`	168 -175	8A	Base + 28		
`EighthPortB`	176 -183	8B			
`EighthPortCL`	184 -187	8CL			
`EighthPortCH`	188 -191	8CH			

Table 17-2. Bit Numbers and Corresponding 8255 Chip Number

82C55 Bit#	Chip #	Address	8536 Bit#	Chip #	Address
0 – 23	1	Base + 0	0 - 19	1	Base + 0
24 – 47	2	Base + 4	20 – 39	2	Base + 4
48 – 71	3	Base + 8			
72 – 95	4	Base + 12			
96 – 119	5	Base + 16			
120 – 143	6	Base + 20			
144 – 167	7	Base + 24			
168 – 191	8	Base + 28			

DBitIn()

Reads the state of a single digital input bit. This method treats all of the DIO ports of a particular type on a board as a single port. It lets you read the state of any individual bit within this port. Note that for some port types, such as 8255 ports, if the port is configured for `DigitalOut`, this method provides readback of the last output value.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function DBitIn(ByVal portType As MccDaq.DigitalPortType ,
ByVal bitNum As Integer, ByRef bitValue As MccDaq.DigitalLogicState)
As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo DBitIn(MccDaq.DigitalPortType portType, int
bitNum, out MccDaq.DigitalLogicState bitValue )
```

Parameters:

`portType` There are three general types of digital ports - ports that are programmable as input or output, ports that are fixed input or output and ports for which each bit may be programmed as input or output. For the first of these types, set `PortType` to `FirstPortA`. For the latter two types, set `PortType` to `AuxPort`. Some boards have both types of digital ports (DAS1600). Set `PortType` to either `FirstPortA` or `AuxPort`, depending on which digital inputs you wish to read.

`bitNum` This specifies the bit number within the single large port. Table 17-2 on page 231 shows which bit numbers are in which 82C55 and 8536 digital chips. The most 82C55 chips on a single board is eight (8), on the CIO-DIO196. The most (2) 8536 chips occur on the CIO-INT32.

`bitValue` Place holder for return value of bit. Value will be 0 or 1. A 0 indicates a logic low reading, a 1 indicates a logic high reading. Logic high does not necessarily mean 5 V. See the board manual for chip input specifications.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`BitValue` - value (0 or 1) of specified bit returned here.

DBitOut()

Sets the state of a single digital output bit. This method treats all of the DIO chips of a particular type on a board as a single very large port. It lets you set the state of any individual bit within this large port. If the port type is not `AuxPort`, you **must** use `DConfigPort()` to configure the port for output first. If the port type is `AuxPort`, you **may** need to use `DConfigBit()` or `DConfigPort()` to configure the bit for output first. Check the board specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) to determine if `AuxPort` should be configured for your hardware.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function DBitOut(ByVal portType As MccDaq.DigitalPortType ,
ByVal bitNum As Integer, ByVal bitValue As MccDaq.DigitalLogicState
) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo DBitOut(MccDaq.DigitalPortType portType, int
bitNum, MccDaq.DigitalLogicState bitValue)
```

Parameters:

`portType` — There are three general types of digital ports - ports that are programmable as input or output, ports that are fixed input or output and ports for which each bit may be programmed as input or output. For the first of these types, set `PortType` to `FirstPortA`. For the latter two types, set `PortType` to `AuxPort`. Some boards have both types of digital ports (DAS1600). Set `PortType` to either `FirstPortA` or `AuxPort` depending on which digital port you wish to write to.

`bitNum` — This specifies the bit number within the single large port. The specified bit must be in a port that is currently configured as an output.

Table 17-2 on page 231 shows which bit numbers are in which 82C55 and 8536 digital chips. The most 82C55 chips on a single board is eight (8), on the CIO-DIO196. The most (2) 8536 chips occur on the CIO-INT32.

`bitValue` — The value to set the bit to. Value will be 0 or 1. A 0 indicates a logic low output, a 1 indicates a logic high output. Logic high does not necessarily mean 5V. Refer to the board's user's guide for chip specifications.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

DConfigBit()

Configures a specific digital bit as Input or Output. This method treats all DIO ports of the `AuxPort` type on a board as a single port. This method is NOT supported by 8255 type DIO ports. Please refer to board specific information for details.

Member of the [`MccBoard`] class.

Function prototype:

VB .NET:	`Public Function DConfigBit(ByVal portNum As MccDaq.DigitalPortType, ByVal bitNum As Integer, ByVal direction As MccDaq.DigitalPortDirection ) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo DConfigBit(MccDaq.DigitalPortType portNum, int bitNum, MccDaq.DigitalPortDirection direction)`

Parameters:

`portNum`	The port (`AuxPort`) whose bits are to be configured. The port specified must be bitwise configurable. See board specific information for details.
`bitNum`	The bit number to configure as input or output. See board specific information for details.
`direction`	`MccDaq.DigitalPortDirection DigitalOut` or `DigitalIn` configures the specified bit for output or input, respectively.

Returns:

An [`ErrorInfo`] object that indicates the status of the operation.

DConfigPort()

Configures a digital port as input or output. This method is for use with ports that may be programmed as input or output, such as those on the 82C55 chips and 8536 chips. See the board user's manual for details of chip operation.

Member of the `MccBoard` class.

Function prototype:

VB .NET:
```
Public Function DConfigPort(ByVal portNum As MccDaq.DigitalPortType
, ByVal direction As MccDaq.DigitalPortDirection ) As
MccDaq.ErrorInfo
```

C# .NET:
```
public MccDaq.ErrorInfo DConfigPort(MccDaq.DigitalPortType portNum,
MccDaq.DigitalPortDirection direction)
```

Parameters:

`portNum` — The specified port must be configurable. For most boards, `AuxPort` is not configurable; so please consult your board-specific documentation.

Table 17-2 on page 231 shows which ports and bit numbers are in which 82C55 and 8536 digital chips. The most 82C55 chips on a single board is eight (8), on the CIO-DIO196. The most (2) 8536 chips occur on the CIO-INT32.

`direction` — `MccDaq.DigitalPortDirection.DigitalOut` or `MccDaq.DigitalPortDirection.DigitalIn` configures the entire eight-bit or four-bit port for output or input.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

When used on ports within an 8255 chip, this method will reset all ports on that chip configured for output to a zero state. This means that if you set an output value on `FirstPortA` and then change the configuration on `FirstPortB` from `Output` to `Input`, the output value at `FirstPortA` will be all zeros. You can, however, set the configuration on `SecondPortX` without affecting the value at `FirstPortA`. For this reason, this method is usually called at the beginning of the program for each port requiring configuration.

DIn()

Reads a digital input port. Note that for some port types, such as 8255 ports, if the port is configured for `DigtalOut`, this method will provide readback of the last output value.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function DIn(ByVal portNum As MccDaq.DigitalPortType , ByRef dataValue As Short) As MccDaq.ErrorInfo
```

```
Public Function DIn(ByVal portNum As MccDaq.DigitalPortType, ByRef dataValue As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo DIn(MccDaq.DigitalPortType portNum, out ushort dataValue)
```

```
public MccDaq.ErrorInfo DIn(MccDaq.DigitalPortType portNum, out short dataValue)
```

Parameters:

`portNum` Specifies which digital I/O port to read. Some hardware does allow readback of the state of the output using this method. Check the board specific information in the *Universal Library User's Guide*.

Table 17-2 on page 231 shows which ports are in which 82C55 and 8536 digital chips. The most 82C55s on a single board is eight (8), on the CIO-DIO196. The most 8536s on a single board is two (2), on the CIO-INT32.

`dataValue` Digital input value returned here.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`dataValue` - Digital input value returned here

Notes:

The size of the ports vary. If it is an eight bit port, the returned value is in the 0 - 255 range. If it is a four bit port, the value is in the 0 - 15 range.

Refer to the board-specific information contained in the *Universal Library User's Guide* for clarification of valid `portNum` values (available in PDF format on our website at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf)

DInScan()

Multiple reads of digital input port of a high speed digital port on a board with a pacer clock such as the CIO-PDMA16.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function DInScan(ByVal portNum As MccDaq.DigitalPortType ,
ByVal numPoints As Integer, ByRef rate As Integer, ByVal memHandle
As Integer, ByVal options As MccDaq.ScanOptions ) As
MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo DInScan(MccDaq.DigitalPortType portNum, int
numPoints, ref int rate, int memHandle, MccDaq.ScanOptions options)
```

Parameters:

`portNum` — Specifies which digital I/O port to read (usually, `FirstPortA` or `FirstPortB`). The specified port must be configured as an input.

`numPoints` — The number of times to read digital input.

`rate` — Number of times per second (Hz) to read the port. The actual sampling rate in some cases will vary a small amount from the requested rate. The actual rate will be returned to the `rate` parameter.

`memHandle` — Handle for Windows buffer to store data in (Windows). This buffer must have been previously allocated with the `WinBufAlloc()` method.

`options` — Bit fields that control various options. Set it to one of the constants in the "options" section below.

Transfer method - May not be specified. DMA is used.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`rate` - actual sampling rate returned.

`memHandle` - digital input value returned via allocated Windows buffer.

options parameter values:

All of the `options` settings are `MccDaq.ScanOptions` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `ScanOptions` enumeration (*variable* = `MccDaq.ScanOptions.Background`, *variable* = `MccDaq.ScanOptions.Continuous`, etc.).

`Background` — If the `Background` option is not used, the `DInScan()` method will not return to your program until all of the requested data has been collected and returned to DataBuffer.

When the `Background` option is used, control will return immediately to the next line in your program and the transfer from the digital input port to DataBuffer will continue in the background. Use `GetStatus()` to check on the status of the background operation. Use `StopBackground()` to terminate the background process before it has completed.

`Continuous` — This option puts the method in an endless loop. Once it transfers the required number of bytes it resets to the start of `dataBuffer` and begins again. The only way to stop this operation is with `StopBackground()`.

	Normally this option should be used in combination with `Background` so that your program will regain control.
`ExtClock`	If this option is used then transfers will be controlled by the signal on the trigger input line rather than by the internal pacer clock. Each transfer will be triggered on the appropriate edge of the trigger input signal (refer to board-specific info). When this option is used, the `rate` parameter is ignored. The transfer rate is dependent on the trigger signal.
`WordXfer`	Normally this method reads a single (byte) port. If `WordXfer` is specified then it will read two adjacent ports on each read and store the value of both ports together as the low and high byte of a single array element in `dataBuffer`[].

Notes:

Transfer method - May not be specified. DMA is used.

DOut()

Writes a byte to a digital output port. If the port type is not `AuxPort`, you **must** use [`DConfigPort()`] to configure the port for output first. If the port type is `AuxPort`, you **may** need to use [`DConfigPort()`] to configure the port for output first. Check the board specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) to determine if `AuxPort` should be configured for your hardware.

Member of the [`MccBoard`] class.

Function prototype:

VB .NET:

```
Public Function DOut(ByVal portNum As MccDaq.DigitalPortType, ByVal dataValue As Short) As MccDaq.ErrorInfo
Public Function DOut(ByVal portNum As MccDaq.DigitalPortType, ByVal dataValue As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo DOut(MccDaq.DigitalPortType portNum, ushort dataValue)
public MccDaq.ErrorInfo DOut(MccDaq.DigitalPortType portNum, short dataValue)
```

Parameters:

`portNum` There are three general types of digital ports - ports that are programmable as input or output, ports that are fixed input or output, and ports for which each bit may be programmed as input or output. For the first of these types, set `portNum` to `FirstPortA`. For the latter two types, set `portNum` to `AuxPort`. Some boards have both types of digital ports (DAS1600). Set `portNum` to either `FirstPortA` or `AuxPort` depending on which digital port you wish to write to. Table 17-2 on page 231 shows which ports are in which 82C55 and 8536 digital chips. The most 82C55 chips on a single board is eight (8), on the CIO-DIO196. The most 8536 chips on a board is two (2) on the CIO-INT32.

`dataValue` Digital input value to be written.

Returns:

An [`ErrorInfo`] object that indicates the status of the operation.

Notes:

The size of the ports vary. If it is an eight bit port, the output value is in the 0 - 255 range. If it is a four bit port, the value is in the 0 - 15 range. Refer to the board-specific information in the *Universal Library User's Guide* for valid `portNum` values (available in PDF format on our website at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf)

DOutScan()

Performs multiple writes to a digital output port of a high speed digital port on a board with a pacer clock, such as the CIO-PDMA16 or CIO-PMA32.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function DOutScan(ByVal portNum As MccDaq.DigitalPortType ,
ByVal count As Integer, ByRef rate As Integer, ByVal memHandle As
Integer, ByVal options As MccDaq.ScanOptions ) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo DOutScan(MccDaq.DigitalPortType portNum, int
count, ref int rate, int memHandle, MccDaq.ScanOptions options)
```

Parameters:

`portNum`	Specifies which digital I/O port to write. The two choices are `FirstPortA` or `FirstPortB`. The specified port must be configured as an output.
`count`	The number of times to write digital output.
`*rate`	Number of times per second (Hz) to write to the port. The actual update rate in some cases will vary a small amount from the requested rate. The actual rate will be returned to the `rate` parameter.
`memHandle`	Handle for Windows buffer to store data in (Windows). This buffer must have been previously allocated with the `WinBufAlloc()` method.
`options`	Bit fields that control various options. Set it to one of the constants in the "options" section below.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`rate` - actual sampling rate returned.

options parameter values:

All of the `options` settings are `MccDaq.ScanOptions` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `ScanOptions` enumeration (*variable* = `MccDaq.ScanOptions.Background`, *variable* = `MccDaq.ScanOptions.Continuous`, etc.).

`Background`	If the `Background` option is not used then the `DOutScan()` method will not return to your program until all of the requested data has been output. When the `Background` option is used, control will return immediately to the next line in your program and the transfer to the digital output port from `dataBuffer` will continue in the background. Use `GetStatus()` to check on the status of the background operation. Use `StopBackground()` to terminate the background process before it has completed.
`Continuous`	This option puts the method in an endless loop. Once it transfers the required number of bytes it resets to the start of `the buffer` and begins again. The only way to stop this operation is with `StopBackground()`. Normally this option should be used in combination with `Background` so that your program will regain control.

`ExtClock`	If this option is used then transfers will be controlled by the signal on the trigger input line rather than by the internal pacer clock. Each transfer will be triggered on the appropriate edge of the trigger input signal (see board specific information). When this option is used the `rate` parameter is ignored. The transfer rate is dependent on the trigger signal.
`WordXfer`	Normally this method writes a single (byte) port. If `WordXfer` is specified then it will write two adjacent ports as the low and high byte of a single array element in `dataBuffer`.

Notes:

- `MccDaq.ScanOptions.ByteXfer` is the default `option`. Make sure you are using an array when your data is arranged in bytes. Use the `MccDaq.ScanOptions.WordXfer` option for word array transfers.
- **Transfer method** - May not be specified. DMA is used.

19

Error Handling Methods and Properties

Introduction

Use the methods and properties explained in this chapter to get information from error codes returned by other UL for .NET methods. Most library methods return ErrorInfo objects. These objects contain properties that provide information on the status of the method called. The different routines built into the methods for handling errors include stopping the program when an error occurs, and printing error messages versus error codes.

ErrHandling()

Sets the error handling for all subsequent method calls. Most methods return error codes after each call. In addition, other error handling features are built into the library. This method controls those features. If the Universal Library cannot find the configuration file CB.CFG, it always terminates the program, regardless of the `ErrHandling()` setting.

Member of the `MccService` class.

Function prototype:

VB .NET:
```
Public Shared Function ErrHandling(ByVal errorReporting As
MccDaq.ErrorReporting , ByVal errorHandling As MccDaq.ErrorHandling
) As MccDaq.ErrorInfo
```

C# .NET:
```
public static MccDaq.ErrorInfo ErrHandling(MccDaq.ErrorReporting
errorReporting, MccDaq.ErrorHandling errorHandling))
```

Parameters:

`errorReporting`	This parameter controls when the library will print error messages on the screen. The default is `DontPrint`. Set it to one of the constants in the "errorReporting parameter values" section below.
`errorHandling`	This parameter specifies what class of error will cause the program to halt. Set it to one of the constants in the "errorHandling parameter values" section below.

Returns:

Returns an `ErrorInfo` object that always has `ErrorInfo.Value` = `NOERRORS`.

`errorReporting` parameter values:

All of the `errorReporting` settings are `MccDaq.ErrorReporting` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `ErrorReporting` enumeration (*variable* = `MccDaq.ErrorReporting.DontPrint`, *variable* = `MccDaq.ErrorReporting.PrintWarnings`, etc.).

`DontPrint`	Errors will not generate a message to the screen. In that case your program must always check the returned error code after each library call to determine if an error occurred.
`PrintWarnings`	Only warning errors will generate a message to the screen. Your program will have to check for fatal errors.
`PrintFatal`	Only fatal errors will generate a message to the screen. Your program must check for warning errors.
`PrintAll`	All errors will generate a message to the screen.

`errorHandling` parameter values:

All of the `errorReporting` settings are `MccDaq.ErrorHandling` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `ErrorHandling` enumeration (*variable* = `MccDaq.ErrorHandling.DontStop`, *variable* = `MccDaq.ErrorHandling.StopFatal`, etc.).

`DontStop`	The program will always continue executing when an error occurs.
`StopFatal`	The program will halt if a "fatal" error occurs.
`StopAll`	Will stop whenever any error occurs. You can check error codes to determine the cause of the error.

Notes:

Warnings vs. fatal errors: All errors that can occur are classified as either "warnings" or "fatal."

- Errors that can occur in normal operation in a bug free program (disk is full, too few samples before trigger occurred) are classified as "warnings."
- All other errors indicate a more serious problem and are classified as "fatal."

Message property

Use the `ErrorInfo.Message` property to get the error message associated with an `ErrorInfo` object. Most UL for .NET methods return an `ErroInfo` object. If an error occurred, an `ErroInfo` object is returned with the `Message` property set to "No error has occurred".

Member of the `ErrorInfo` class.

Property prototype:

VB .NET:	`Public ReadOnly Property Message As String`
C# .NET:	`public string Message [get]`

Notes:

Refer to the `ErrHandling()` method for an alternate method of handling errors.

Value property

Use the `ErrorInfo.Value` property to get the error constant associated with an `ErrorInfo` object. Most UL for .NET methods return an `ErroInfo` object. If an error occurs, an `ErroInfo` object is returned with a non-zero value in the `Value` property.

Member of the `ErrorInfo` class.

Property prototype:

VB .NET:	`Public ReadOnly Property Value As MccDaq.ErrorInfo.ErrorCode`
C# .NET:	`public MccDaq.ErrorInfo.ErrorCode Value [get]`

Notes:

Refer to the `ErrHandling()` method for an alternate method of handling errors.

20

Memory Board Methods

Use the functions explained in this chapter to read and write data to and from a memory board, and also set modes that control memory boards (MEGA-FIFO).

The most common use for the memory boards is to store large amounts of data from an A/D board via a DT-Connect cable to a memory board. To do this, use the `ExtMemory` option with `AInScan()` or `APretrig()`.

Once the data is transferred to the memory board, you can use the memory functions to retrieve it.

MemRead()

Reads data from a memory board into an array. Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function MemRead(ByRef dataBuffer As Short, ByVal firstPoint
As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

```
Public Function MemRead(ByRef dataBuffer As System.UInt16, ByVal
firstPoint As Integer, ByVal numPoints As Integer) As
MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo MemRead(out short dataBuffer, int
firstPoint, int numPoints)
```

```
public MccDaq.ErrorInfo MemRead(out ushort dataBuffer, int
firstPoint, int numPoints)
```

Parameters:

`dataBuffer`	Reference to the data array.
`firstPoint`	Index of first point to read, or `FromHere`. Use the `firstPoint` parameter to specify the first point to be read. For example, to read data sample numbers 200 through 250, set `firstPoint`= 200 and `Count` = 50.
`numPoints`	Number of data points (words) to read.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`dataBuffer` - data read from the memory board.

Notes:

If you are going to read a large amount of data from the board in small chunks, set `firstPoint` to `FromHere` to read each successive chunk. Using `FromHere` speeds up the operation of `MemRead()` when working with large amounts of data.

For example, to read 300,000 points in 100,000 point chunks, the calls would look like this:

```
DaqBoard0.MemRead (DataBuffer, 0, 100000)
DaqBoard0.MemRead (DataBuffer, FROMHERE, 1000000)
DaqBoard0.MemRead (DataBuffer, FROMHERE, 1000000)
```

DT-Connect Conflicts - The `MemRead()` method can not be called while a DT-Connect transfer is in progress. For example, if you start collecting A/D data to the memory board in the background (by calling `AInScan()` with the `DTConnect` + `Background` options) you cannot call `MemRead()` until the `AInScan()` has completed. If you do you will get a `DtActive` error.

MemReadPretrig()

Reads pre-trigger data from a memory board that has been collected with the `APretrig()` method and re-arranges the data in the correct order (pre-trigger data first, then post-trigger data). This method can only be used to retrieve data that has been collected with the APretrig() method with `ExtMemory` set in the options parameter. After each APretrig() call, all data must be unloaded from the memory board with this method. If any more data is sent to the memory board then the pre-trigger data will be lost.

Member of the `MccBoard` class.

Function Prototype:

VB .NET:

```
Public Function MemReadPretrig(ByRef dataBuffer As Short, ByVal
firstPoint As Integer, ByVal numPoints As Integer) As
MccDaq.ErrorInfo
```

```
Public Function MemReadPretrig(ByRef dataBuffer As System.UInt16,
ByVal firstPoint As Integer, ByVal numPoints As Integer) As
MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo MemReadPretrig(out short dataBuffer, int
firstPoint, int numPoints)
```

```
public MccDaq.ErrorInfo MemReadPretrig(out ushort dataBuffer, int
firstPoint, int numPoints)
```

Parameters:

dataBuffer	Reference to the data array
firstPoint	Index of first point to read or `FromHere`. Use the `FirstPoint` parameter to specify the first point to be read. For example, to read data sample numbers 200 through 250, set `FirstPoint` = 200 and `Count` = 50.
numPoints	Number of data samples (words) to read

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`dataBuffer` - data read from memory board

Notes:

If you are going to read a large amount of data from the board in small chunks, set `FirstPoint` to `FromHere` to read each successive chunk. Using `FromHere` speeds up the operation of `MemRead()` when working with large amounts of data.

For example, to read 300,000 points in 100,000 chunks, the calls would look like this:

```
DaqBoard0.MemReadPretrig (0, DataBuffer, 0, 100000)
DaqBoard0.MemReadPretrig (0, DataBuffer, FROMHERE, 1000000)
DaqBoard0.MemReadPretrig (0, DataBuffer, FROMHERE, 1000000)
```

DT-Connect Conflicts - The `MemRead()` method can not be called while a DT-Connect transfer is in progress. For example, if you start collecting A/D data to the memory board in the background (by calling `AInScan()` with the `DTConnect` + `Background` options) you cannot call `MemRead()` until the `AInScan()` has completed. If you do you will get a `DTACTIVE` error.

MemReset()

Resets the memory board reference to the start of the data. The memory boards are sequential devices. They contain a counter which points to the 'current' word in memory. Every time a word is read or written this counter increments to the next word.

Member of the `MccBoard` class.

Function Prototype:

VB .NET: `Public Function MemReset() As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo MemReset()`

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

This method is used to reset the counter back to the start of the memory. Between successive calls to `AInScan()`, you should call this method so that the second `AInScan()` overwrites the data from the first call. Otherwise, the data from the first `AInScan()` will be followed by the data from the second `AInScan()` in the memory on the card.

Likewise, anytime you call `MemRead()` or `MemWrite()`, it will leave the counter pointing to the next memory location after the data that you read or wrote. Call `MemReset()` to reset back to the start of the memory buffer before the next call to `AInScan()`.

MemSetDTMode()

Sets the DT-Connect Mode of a memory board.

Member of the `MccBoard` class.

Function Prototype:

VB .NET: `Public Function MemSetDTMode(ByVal mode As MccDaq.DTMode ) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo MemSetDTMode(MccDaq.DTMode mode)`

Parameters:

`mode` Must be set to either `DTIn` or `DTOut`. Set the mode on the memory board to `DTIn` to transfer data from an A/D board to the memory board. Set `mode` = `DTOut` to transfer data from a memory board to a D/A board.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

This command only controls the direction of data transfer between the memory board and its parent board that is connected to it via a DT-Connect cable.

If using the `ExtMemory` option for `AInScan()`, etc., *this method should not be used*. The memory board mode is already set through the `ExtMemory` option.

Use this method only if the parent board is not supported by the Universal Library.

MemWrite()

Writes data from an array to the memory card.

Member of the `MccBoard` class.

Function prototype:

VB .NET:	`Public Function MemWrite(ByRef dataBuffer As Short, ByVal firstPoint As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo`
	
	`Public Function MemWrite(ByRef dataBuffer As System.UInt16, ByVal firstPoint As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo`

C# .NET:	`public MccDaq.ErrorInfo MemWrite(ref short dataBuffer, int firstPoint, int numPoints)`
	
	`public MccDaq.ErrorInfo MemWrite(ref ushort dataBuffer, int firstPoint, int numPoints)`

Parameters:

`dataBuffer`	Reference to the data array.
`firstPoint`	Index of first point to write or `FromHere`. Use the `firstPoint` parameter to specify where in the board's memory to write the first point. For example, to write to location numbers 200 through 250, set `firstPoint`= 200 and `Count` = 50.
`numPoints`	Number of data points (words) to write

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

To write large amount of data to the board in small chunks, set `firstPoint` to `FromHere` to write each successive chunk. Using `FromHere` speeds up the operation of `MemWrite()` when working with large amounts of data.

For example, to write 300,000 points in 100,000 point chunks, the calls would look like this:

```
DaqBoard1.MemWrite (0, DataBuffer, 0, 100000)
DaqBoard1.MemWrite (0, DataBuffer, FROMHERE, 100000)
DaqBoard1.MemWrite (0, DataBuffer, FROMHERE, 100000)
```

DT-Connect Conflicts - The `MemWrite()` method cannot be called while a DT-Connect transfer is in progress. For example, if you start collecting A/D data to the memory board in the background (by calling `AInScan()` with the `DTCONNECT + BACKGROUND` options). You cannot call `MemWrite()` until the `AInScan()` has completed. If you do, you will get a `DTACTIVE` error.

Revision Control Methods and Properties

Introduction

Use the methods and properties explained in this chapter to initialize the Universal Library DLL so that the underlying functions are interpreted according to the format of the revision you wrote and compiled your program in.

As new revisions of the library are released, bugs from previous revisions are fixed and occasionally new methods are added. It is our goal to preserve existing programs you have written and therefore to never change the order or number of arguments in a method. However, sometimes it is not possible to achieve this goal.

DeclareRevision()

Initializes the Universal Library with the revision number of the library used to write your program. Must be the first Universal Library for .NET method to be called by your program.

Member of the `MccService class`.

Function prototype:

VB .NET:	`Public Shared Function DeclareRevision(ByRef revNum As Single) As MccDaq.ErrorInfo`
C# .NET:	`public static MccDaq.ErrorInfo DeclareRevision(ref float revNum)`

Parameters:

`revNum`	Revision number of the Library to interpret method parameters.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

Default: Any program using the 32-bit library and not containing this line of code will be defaulted to revision 5.4 parameter assignments.

As new revisions of the library are released, bugs from previous revisions are fixed and occasionally new functions are added. It is Measurement Computing's goal to preserve existing programs you have written and therefore to never change the order or number of parameters in a method.

With the `DeclareRevision()` method, programs do not have to be rewritten in each line where new functions are used, and the program then recompiled. The revision control method initializes the DLL so that the functions are interpreted according to the format of the revision that you wrote and compiled your program in. The method works by interpreting the UL function call from your program and filling in any arguments needed to run with the new revision.

If your program has declared you are running code written for an earlier revision and you call a new method, you must rewrite your program to include the new parameter, and declare the current revision in `DeclareRevision()`.

GetRevision()

Gets the revision level of Universal Library DLL and the VXD.

Member of the `MccService class`.

Function prototype:

VB .NET:	`Public Shared Function GetRevision(ByRef revNum As Single, ByRef vxdRevNum As Single) As MccDaq.ErrorInfo`
C# .NET:	`public static MccDaq.ErrorInfo GetRevision(out float revNum, out float vxdRevNum)`

Parameters:

`revNum`	Place holder for the revision number of Library DLL.
`vxdRevNum`	Place holder for the revision number of Library VXD.

Returns:

`revNum` - Revision number of the Library DLL

`vxdRevNum` - Revision number of the Library VXD

An `ErrorInfo` object that indicates if the revision levels of VXD and DLL are incompatible.

Streamer File Methods

Introduction

Use the streamer file methods explained in the chapter to create, fill, and read streamer files.

FileAInScan()

Scans a range of A/D channels and stores the samples in a disk file. `FileAInScan()` reads the specified number of A/D samples at the specified sampling rate from the specified range of A/D channels from the specified board. If the A/D board has programmable gain, it sets the gain to the specified range. The collected data is returned to a file in binary format. Use `FileRead()` to load data from that file into an array. See board specific information to determine if this method is supported on your board.

Member of the `MccBoard class`.

Function Prototype:

VB .NET:

```
Public Function FileAInScan(ByVal lowChan As Integer, ByVal highChan As Integer, ByVal numPoints As Integer, ByRef rate As Integer, ByVal range As MccDaq.Range , ByVal fileName As String, ByVal options As MccDaq.ScanOptions) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo FileAInScan(int lowChan, int highChan, int numPoints, ref int rate, MccDaq.Range range, string fileName, MccDaq.ScanOptions options)
```

Parameters:

`lowChan` — First A/D channel of scan.

`highChan` — Last A/D channel of scan.

The maximum allowable channel depends on which type of A/D board is being used. For boards with both single ended and differential inputs, the maximum allowable channel number also depends on how the board is configured (for example, eight channels for differential, 16 for single ended).

`numPoints` — Specifies the total number of A/D samples that will be collected. If more than one channel is being sampled, the number of samples collected per channel is equal to Count / (HighChan-LowChan+1).

`rate` — Sample rate in samples per second (Hz) per channel. The maximum sampling rate depends on the A/D board that is being used (refer to the `rate` description in `AInScan()`).

`range` — If the selected A/D board does not have a programmable range feature, this parameter is ignored. Otherwise set the `range` parameter to any range that is supported by the selected A/D board. Refer to Table 14-1 on page 155 for a list of valid range settings. Refer to board specific information for a list of the supported A/D ranges of each board.

`filename` — The name of the file in which to store the data. If the file doesn't exist, it will be created. (When using the 16 bit version of the Universal Library, the named file must already exist. It should have been previously created with the MAKESTRM.EXE program.)

`options` — Bit fields that control various options. Set it to one of the constants in the "options" section on page 261.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`rate` = actual sampling rate

options :

All of the `options` settings are `MccDaq.ScanOptions` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `ScanOptions` enumeration (*variable* = `MccDaq.ScanOptions.ExtClock`, *variable* = `MccDaq.ScanOptions.ExtTrigger`, etc.).

`ExtClock`	If this option is used, conversions are controlled by the signal on the trigger input line rather than by the internal pacer clock. Each conversion is triggered on the appropriate edge of the trigger input signal (see board specific info). Additionally, the `rate` parameter is ignored. The sampling rate is dependent on the trigger signal.
`ExtTrigger`	If this option is specified, the sampling does not begin until the trigger condition is met. On many boards, this trigger condition is programmable (see `SetTrigger()` method and board specific info for details) and can be programmed for rising or falling edge or an analog level. On other boards, only "polled gate" triggering is supported. Assuming active high operation, data acquisition commences immediately if the trigger input is high. If the trigger input is low, acquisition is held off until it goes high. Acquisition continues until `numPoints&` samples are taken, regardless of the state of the trigger input. For 'polled gate' triggering, this option is most useful if the signal is a pulse with a very low duty cycle (trigger signal in TTL low state most of the time) to hold off triggering until the pulse occurs.
`DtConnect`	Samples are sent to the DT-Connect port if the board is equipped with one.

Notes:

Important

In order to understand the functions, you must read the board-specific information contained in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf). Review and run the example programs before attempting to program yourself. Following this advice will save you hours of frustration, and possibly time wasted holding for technical support.
This note, which appears elsewhere, is especially applicable to this method. Read the board-specific information for your board. We suggest that you make a copy of that page to refer to as you read this manual and examine the example programs.

`OverRun` error (Error code 29): This error indicates that the data was not written to the file as fast as the data was sampled. Consequently some data was lost. The value returned from `FileGetInfo()` in `*TotalCount` is the number of points that were successfully collected.

FileGetInfo()

This method returns information about a streamer file. When `FileAInScan()` or `FilePretrig()` fills the streamer file, information is stored about how the data was collected (sample rate, channels sampled etc.). This method returns that information. See board specific info to determine if this method is supported on your board.

Member of the `MccService` class.

Function prototype:

VB .NET:

```
Public Shared Function FileGetInfo(ByVal fileName As String, ByRef
lowChan As Short, ByRef highChan As Short, ByRef pretrigCount As
Integer, ByRef totalCount As Integer, ByRef rate As Integer, ByRef
range As MccDaq.Range ) As MccDaq.ErrorInfo
```

C# .NET:

```
public static MccDaq.ErrorInfo FileGetInfo(string fileName, out
short lowChan, out short highChan, out int pretrigCount, out int
totalCount, out int rate, out MccDaq.Range range)
```

Parameters:

`fileName`	Name of streamer file.
`lowChan`	Variable to return `lowChan` to.
`highChan`	Variable to return `highChan` to.
`pretrigCount`	Variable to return `pretrigCount` to.
`totalCount`	Variable to return `totalCount` to.
`rate`	Variable to return sampling `rate` to.
`range`	Variable to return A/D `range` code to. Refer to Table 14-1 on page 155 for a list of valid range settings.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`lowChan` - low A/D channel of scan

`highChan` - high A/D channel of scan

`totalCount` - total number of points collected

`pretrigCount` - number of pre-trigger points collected

`rate` - sampling rate when data was collected

`range` - Range of A/D when data was collected

FilePretrig()

Scan a range of channels continuously while waiting for a trigger.

Once the trigger occurs, `FilePretrig()` returns the specified number of samples, including the specified number of pre-trigger samples to a disk file. This method waits for a trigger signal to occur on the Trigger Input. Once the trigger occurs, it returns the specified number (`TotalCount`) of A/D samples, including the specified number of pre-trigger points. It collects the data at the specified sampling rate (`rate`) from the specified range (`lowChan-highChan`) of A/D channels from the specified board. If the A/D board has programmable gain then it sets the gain to the specified range. The collected data is returned to a file. See board specific info to determine if this method is supported by your board.

Member of the [`MccBoard`] class.

Function prototype:

VB .NET:

```
Public Function FilePretrig(ByVal lowChan As Integer, ByVal highChan
As Integer, ByRef pretrigCount As Integer, ByRef totalCount As
Integer, ByRef rate As Integer, ByVal range As MccDaq.Range , ByVal
fileName As String, ByVal options As MccDaq.ScanOptions ) As
MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo FilePretrig(int lowChan, int highChan, ref
int pretrigCount, ref int totalCount, ref int rate, MccDaq.Range
range, string fileName, MccDaq.ScanOptions options )
```

Parameters:

`lowChan`
First A/D channel of scan

`highChan`
Last A/D channel of scan

The maximum allowable channel depends on which type of A/D board is being used. For boards that have both single ended and differential inputs the maximum allowable channel number also depends on how the board is configured. Refer to board-specific information for the maximum number of channels allowed in differential and single ended modes.

`pretrigCount`
Specifies the number of samples before the trigger that will be returned. `PretrigCount` must be less than 16000, and `PretrigCount` must also be less than `TotalCount` - 512.

If the trigger occurs too early, then fewer than the requested number of pre-trigger samples will be collected. In that case a TooFew error will occur. The `PretrigCount` will be set to indicate how many samples were collected and the post trigger samples will still be collected.

`totalCount`
Sets the total number of samples to be collected and stored in the file. `TotalCount` must be greater than or equal to `PretrigCount` + 512.

If the trigger occurs too early, fewer than the requested number of samples will be collected and a TooFew error will occur. The `TotalCount` will be set to indicate how many samples were actually collected.

`rate`
Sample rate in samples per second (Hz) per channel. The maximum sampling rate depends on the A/D board that is being used. This is the rate at which scans are triggered.

If you are sampling 4 channels, 0 - 3, then specifying a rate of 10,000 scans per second (10 kHz) will result in the A/D converter rate of 40 kHz: 4 channels at 10,000 samples per channel per second. This is different from some software, where you specify the total A/D chip rate. In those systems, the per channel rate is equal to the A/D rate divided by the number of channels in a scan. This parameter also returns the value of the actual set. This may be different from the requested rate because of pacer limitations.

`range`	If the selected A/D board does not have a programmable range feature, this parameter is ignored. Otherwise, set the `range` parameter to any range that is supported by the selected A/D board. Refer to Table 14-1 on page 155 for a list of valid range settings. Refer to board specific information for a list of the supported A/D ranges of each board.
`filename`	The name of the file in which to store the data. If the file doesn't exist, it will be created. (When using the 16 bit version of the Universal Library, the named file must already exist. It should have been previously created with the MAKESTRM.EXE program.)
`options`	Bit fields that control various options. Set it to one of the constants in the "options" section below.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`preTrigCount` - actual number of pre-trigger samples collected

`totalCount` - actual number of samples collected

`rate` = actual sampling rate

options parameter values:

All of the `options` settings are `MccDaq.ScanOptions` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `ScanOptions` enumeration (*variable* = `MccDaq.ScanOptions.ExtClock` or *variable* = `MccDaq.ScanOptions.DtConnect`).

`ExtClock`	If this option is used then conversions will be controlled by the signal on the trigger input line rather than by the internal pacer clock. Each conversion will be triggered on the appropriate edge of the trigger input signal (see board specific info). When this option is used the `rate` parameter is ignored. The sampling rate is dependent on the trigger signal.
`DtConnect`	Samples are sent to the DT-Connect port if the board is equipped with one.

Notes:

OverRun error (Error code 29): This error indicates that the data was not written to the file as fast as the data was sampled. Consequently some data was lost. The value in `TotalCount` will be the number of points that were successfully collected.

FileRead()

This method reads data from a streamer file. Refer to board-specific information to determine if this method is supported on your board.

Member of the `MccService` class.

Function prototype:

VB .NET:

```
Public Shared Function FileRead(ByVal fileName As String, ByVal firstPoint As Integer, ByRef numPoints As Integer, ByRef dataBuffer As Short) As MccDaq.ErrorInfo
```

```
Public Shared Function FileRead(ByVal fileName As String, ByVal firstPoint As Integer, ByRef numPoints As Integer, ByRef dataBuffer As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public static MccDaq.ErrorInfo FileRead(string fileName, int firstPoint, ref int numPoints, out ushort dataBuffer)
```

```
public static MccDaq.ErrorInfo FileRead(string fileName, int firstPoint, ref int numPoints, out short dataBuffer)
```

Parameters:

`filename`	Name of streamer file.
`firstPoint`	Index of first point to read.
`totalCount`	Number of points to read from file.
`dataBuffer`	Reference to data buffer that data will be read into.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`dataBuffer` - data read from a file.

`totalCount` - number of points actually read.

`totalCount` may be less than the requested number of points if an error occurs.

Notes:

Data format: The data is returned as 16 bits. The 16 bits may represent 12 bits of analog, 12 bits of analog plus 4 bits of channel, or 16 bits of analog. Use `AConvertData()` to correctly load the data into an array.

Loading portions of files: The file may contain much more data than can fit in `dataBuffer`. In those cases, use `totalCount` and `firstPoint` to read a selected piece of the file into `dataBuffer`. Call `FileGetInfo()` first to find out how many points are in the file.

Temperature Input Methods

Introduction

Use the methods explained in this chapter to convert a raw analog input from an EXP or other temperature sensor board to temperature.

TIn()

Reads an analog input channel, linearizes it according to the selected temperature sensor type, and returns the temperature in degrees.

The CJC channel, the gain, and sensor type, are read from the *Insta*Cal configuration file. They should be set by running the *InstaCal*® configuration program.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function TIn(ByVal chan As Integer, ByVal scale As
MccDaq.TempScale , ByRef tempValue As Single, ByVal options As
MccDaq.ThermocoupleOptions ) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo TIn(int chan, MccDaq.TempScale scale, out
float tempValue, MccDaq.ThermocoupleOptions options)
```

Parameters:

`chan`	Input channel to read.
`scale`	Specifies the temperature scale that the input is converted to. Choices are `MccDaq.TempScale.Celsius`, `MccDaq.TempScale.Fahrenheit and MccDaq.TempScale.Kelvin`.
`tempValue`	The temperature in degrees is returned here. Thermocouple resolution is approximately 0.25 °C, depending on scale, range and thermocouple type. RTD resolution is 0.1 °C.
`options`	Bit fields that control various options. Set it to one of the constants in the "options parameter values" section below.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`tempValue` - Temperature returned here

options parameter values:

All of the `options` settings are `MccDaq.ThermocoupleOptions` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `ThermocoupleOptions` enumeration (*variable* = `MccDaq.ThermocoupleOptions.Filter` or *variable* = `MccDaq.ThermocoupleOptions.NoFilter`).

`Filter`	When selected, a smoothing function is applied to temperature readings, very much like the electrical smoothing inherent in all hand held temperature sensor instruments. This is the default. Ten samples are read from the specified channel and averaged. The average is the reading returned. Averaging removes normally distributed signal line noise.
`NoFilter`	When selected, the temperature readings are not smoothed, resulting in a scattering of readings around a mean.

Notes:

Using CIO-EXP boards: For CIO-EXP boards, the channel number is calculated using the following formula, where:

- ADChan is the A/D channel that is connected to the multiplexer

- `MuxChan` is a number ranging from 0 to 15 that specifies the channel number on a particular bank of the multiplexer board

  `Chan = (ADChan * 16) + (16 + MuxChan)`

For example, you have an EXP16 connected to a CIO-DAS08 via the CIO-DAS08 channel 0. (Remember that DAS08 channels are numbered 0, 1, 2, 3, 4, 5, 6 & 7). If you connect a thermocouple to channel 5 of the EXP16, the value for `chan` would be (0 * 16) + (16 + 5)= 0 + 21 = **21**.

Using 6K-EXP boards: For 6K-EXP boards, the channel number is calculated using one of the following formulas, where:

- ADChan is the A/D channel that is connected to the multiplexer.
- `MuxChan` is a number ranging from 0 to 15 that specifies the channel number on a particular bank of the multiplexer board.
- If the A/D board has 16 or less single-ended channels:

  `Chan = (ADChan * 16) + (16 + MuxChan)`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6052 via the a PCI-DAS6052 channel 0. If you connect a thermocouple to channel 5 of the 6K-EXP16, the value for `Chan` would be (0 * 16) + (16 + 5)= 0 + 21 = **21**.

- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is less than or equal to 7:

  `Chan = (ADChan * 16) + (64 + MuxChan)`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6031 via the a PCI-DAS6031 channel 7. If you connect a thermocouple to channel 5 of the 6K-EXP16, the value for `Chan` would be (7 * 16) + (64 + 5) = 112 + 69 = **181**.

- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is greater than or equal to 31:

  `Chan = (ADChan * 16 – 320) + MuxChan`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6031 via the PCI-DAS6031 channel 32. If you connect a thermocouple to channel 5 of the 6K-EXP16, the value for `Chan` would be (32 * 16 – 320) + 5 = 192 + 5 = **197**.

CJC Channel: The Cold Junction Compensation (CJC) channel is set in the *Insta*Cal install program. If you have multiple EXP boards, Universal Library will apply the CJC reading to the linearization formula in the following manner:

First, if you have chosen a CJC channel for the EXP board that the channel you are reading is on, it will use the CJC temp reading from that channel.

Second, if you left the CJC channel for the EXP board that the channel you are reading is on to NOT SET, the library will use the CJC reading from the next lower EXP board with a CJC channel selected.

For example: You have 4 CIO-EXP16 boards connected to a CIO-DAS08 on channel 0, 1, 2 and 3. You choose CIO-EXP16 #1 (connected to CIO-DAS08 channel 0) to have its CJC read on CIO-DAS08 channel 7, AND, you leave the CIO-EXP16's 2, 3 and 4 CJC channels to NOT SET. Result: The CIO-EXP boards all use the CJC reading from CIO-EXP16 #1, connected to channel 7 for linearization. As you can see, it is important to keep the CIO-EXP boards in the same case and out of any breezes to ensure a clean CJC reading.

A/D range (Important): If the EXP board is connected to an A/D that does not have programmable gain (DAS08, DAS16, DAS16F) then the A/D board range is read from the configuration file (cb.cfg). In most cases, hardware selectable ranges should be set to ±5 V for thermocouples and 0 to 10 V for RTDs. Refer to the board-specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) or in the user manual for your board. If the board does have programmable RTDs gains, the `TIn()` method will set the appropriate A/D range.

Specific Errors: If an `OutOfRange` or `OpenConnection` error occurs, the value returned is -9999.0.

TInScan()

Reads a range of channels from an analog input board, linearizes them according to temperature sensor type, and returns the temperatures to an array in degrees.

The CJC channel, the gain, and temperature sensor type are read from the configuration file. Use the *InstaCal®* configuration program to change any of these options.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function TInScan(ByVal lowChan As Integer, ByVal highChan As
Integer, ByVal scale As MccDaq.TempScale , ByVal dataBuffer As
Single( ), ByVal options As MccDaq.ThermocoupleOptions ) As
MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo TInScan(int lowChan, int highChan,
MccDaq.TempScale scale, out float dataBuffer,
MccDaq.ThermocoupleOptions options)
```

Parameters:

`lowChan`	Low mux channel of scan.
`highChan`	High mux channel of scan.
`scale`	Specifies the temperature scale that the input is converted to. Choices are `MccDaq.TempScale.Celsius`, `MccDaq.TempScale.Fahrenheit` and `MccDaq.TempScale.Kelvin.`
`dataBuffer`	The temperature is returned in degrees. Each element in the array corresponds to a channel in the scan. dataBuffer must be at least large enough to hold highChan - lowChan + 1 temperature values. Thermocouple resolution is approximately 0.25 °C, depending on scale, range and thermocouple type. RTD resolution is 0.1 °C.
options	Bit fields that control various options. Set it to one of the constants in the "options parameter values" section below.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`dataBuffer`[] - Temperature values in degrees are returned here for each channel in scan.

options parameter values:

All of the `options` settings are `MccDaq.ThermocoupleOptions` enumerated constants. To set a variable to one of these constants, you must refer to the `MccDaq` object and the `ThermocoupleOptions` enumeration (*variable* = `MccDaq.ThermocoupleOptions.Filter` or *variable* = `MccDaq.ThermocoupleOptions.NoFilter`).

`Filter`	When selected, a smoothing function is applied to temperature readings, very much like the electrical smoothing inherent in all hand held temperature sensor instruments. This is the default. Ten samples are read from the specified channel and averaged. The average is the reading returned. Averaging removes normally distributed signal line noise.
`NoFilter`	When selected, the temperature readings are not smoothed, resulting in a scattering of readings around a mean.

Notes:

Using EXP boards: For EXP boards, these channel numbers (Chan) are calculated using the following formula:

- ADChan = A/D channel that is connected to the multiplexer
- `MuxChan` is a number ranging from 0 to 15 that specifies the channel number on a particular bank of the multiplexer board

  `Chan = (ADChan *16) + (16 + MuxChan)`

For example, you have an EXP16 connected to a CIO-DAS08 via the CIO-DAS08 channel 0. (Remember, DAS08 channels are numbered 0, 1, 2, 3, 4, 5, 6 & 7). If you connect thermocouples to channels 5, 6, and 7 of the EXP16, the value for `lowChan` would be (0+1) * 16 + 5 = **21**, and the value for `highChan` would be (0+1) * 16 + 7 = **23**.

> **Important**
> For an EXP board connected to an A/D board that does not have programmable gain (DAS08, DAS16, DAS16F), the A/D board range is read from the configuration file (cb.cfg). In most cases, set hardware-selectable ranges to ±5 V for thermocouples, and to 0 to 10 V for RTDs. Refer to the board-specific information in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) or in the user manual for your board. If the board has programmable RTDs gains, the `TIn()` method sets the appropriate A/D range.

Using 6K-EXP boards: For 6K-EXP boards, the channel number (Chan) is calculated using one of the following formulas, where:

- ADChan is the A/D channel that is connected to the multiplexer.
- `MuxChan` is a number ranging from 0 to 15 that specifies the channel number on a particular bank of the multiplexer board.
- If the A/D board has 16 or less single-ended channels:

  `Chan = (ADChan * 16) + (16 + MuxChan)`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6052 via the a PCI-DAS6052 channel 0. If you connect a thermocouple to channels 5, 6, and 7 of the 6K-EXP16, the value for `lowChan` would be (0 * 16) + (16 + 5)= 0 + 21 = **21**, and the value for `highChan` would be (0 * 16) + (16 + 5)= 0 + 231 = **23**.
- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is less than or equal to 7:

  `Chan = (ADChan * 16) + (64 + MuxChan)`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6031 via the a PCI-DAS6031 channel 7. If you connect a thermocouple to channels 5, 6, and 7 of the 6K-EXP16, the value for low`Chan` would be (7 * 16) + (64 + 5) = 112 + 69 = **181**, and the value for `highChan` would be (7 * 16) + (64 + 7) = 112 + 71 = **183**.
- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is greater than or equal to 32:

  `Chan = (ADChan * 16 – 320) + MuxChan`

  For example, you have a 6K-EXP16 connected to a PCI-DAS6031 via the PCI-DAS6031 channel 32. If you connect a thermocouple to channels 5, 6, and 7 of the 6K-EXP16, the value for low`Chan` would be (32 * 16 – 320) + 5 = 192 + 5 = **197**, and the value for high`Chan` would be (32 * 16 – 320) + 7 = 192 + 7 = **199**.

CJC Channel: The Cold Junction Compensation (CJC) channel is set in the *Insta*Cal install program. If you have multiple EXP boards, Universal Library will apply the CJC reading to the linearization formula in the following manner:

- First, if you have chosen a CJC channel for the EXP board that the channel you are reading is on, it will use the CJC temp reading from that channel.

- Second, if you have left the CJC channel for the EXP board that the channel you are reading is on to NOT SET, the library will use the CJC reading from the next lower EXP board with a CJC channel selected.

For example: You have 4 CIO-EXP16 boards connected to a CIO-DAS08 on channel 0, 1, 2 and 3. You choose CIO-EXP16 #1 (connected to CIO-DAS08 channel 0) to have its CJC read on CIO-DAS08 channel 7, AND, you leave the CIO-EXP16's 2, 3 and 4 CJC channels to NOT SET. Result: The CIO-EXP boards all use the CJC reading from CIO-EXP16 #1, connected to channel 7 for linearization. As you can see, it is important to keep the CIO-EXP boards in the same case and out of any breezes to ensure a clean CJC reading.

Important

In order to understand the functions, you must read the board-specific information contained in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Review and run the example programs before attempting any programming of your own. Following this advice will save you hours of frustration, and possibly time wasted holding for technical support.

This note, which appears elsewhere, is especially applicable to this method. Read the board-specific information for your board (see the *Universal Library User's Guide*). We suggest that you make a copy of that page to refer to as you read this manual and examine the example programs.

Specific errors: For most boards, if an `OUTOFRANGE` or `OPENCONNECTION` error occurs, the value in the array element associated with the channel causing the error returned will be -9999.0 (Refer to board-specific information).

24

Windows Memory Management Methods

Introduction

Use the methods explained in this section to allocate, free, and copy to/from Windows global memory buffers.

WinBufAlloc()

Allocates a Windows global memory buffer which can be used with the scan functions and returns a memory handle for it.

Member of the `MccService` class.

Function prototype:

VB .NET:	`Public Shared Function WinBufAlloc(ByVal numPoints As Integer) As Integer`
C# .NET:	`public static int WinBufAlloc(int numPoints)`

Parameters:

`numPoints`	Size of buffer to allocate. Specifies how many data points (16-bit integers, NOT bytes) can be stored in the buffer.

Returns:

0 if buffer could not be allocated or a non-zero integer handle to the buffer.

Notes:

Unlike most other methods in the library, this method does not return an `ErrorInfo` object. It returns a Windows global memory handle, which can then be passed to the scan functions in the library. If an error occurs, the handle will come back as 0 to indicate the error.

WinBufFree()

Frees a Windows global memory buffer which was previously allocated with the `WinBufAlloc()` method.

Member of the `MccService` class.

Function prototype:

VB .NET:
```
Public Shared Function WinBufFree(ByVal memHandle As Integer) As
MccDaq.ErrorInfo
```

C# .NET:
```
public static MccDaq.ErrorInfo WinBufFree (int memHandle)
```

Parameters:

`memHandle`	A Windows memory handle. This must be a memory handle that was returned by `WinBufAlloc()` when the buffer was allocated.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

WinArrayToBuf()

Copies data from an array into a Windows memory buffer.

Member of the `MccService` class.

Function prototype:

VB .NET:

```
Public Shared Function WinArrayToBuf(ByRef dataArray As Short, ByVal
memHandle As Integer, ByVal firstPoint As Integer, ByVal numPoints
As Integer) As MccDaq.ErrorInfo
```

```
Public Shared Function WinArrayToBuf(ByRef dataArray As
System.UInt16, ByVal memHandle As Integer, ByVal firstPoint As
Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

C# .NET:

```
public static MccDaq.ErrorInfo WinArrayToBuf(ref ushort dataArray,
int memHandle, int firstPoint, int numPoints )
```

```
public static MccDaq.ErrorInfo WinArrayToBuf(ref short dataArray,
int memHandle, int firstPoint, int numPoints )
```

Parameters:

`dataArray`	The array containing the data to be copied.
`memHandle`	This must be a memory handle that was returned by `WinBufAlloc()` when the buffer was allocated. The data will be copied into this buffer.
`firstPoint`	Index of first point in memory buffer where data will be copied to.
`numPoints`	Number of data points to copy.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

This method copies data from an array to a Windows global memory buffer. This would typically be used to initialize the buffer with data before doing an output scan. Using the `firstPoint` and `count` parameter it is possible to fill a portion of the buffer. This can be useful if you want to send new data to the buffer after a `Background` + `Continuous` scan command has sent the old data - i.e. circular buffering.

WinBufToArray()

Copies data from a Windows memory buffer into an array.

Member of the `MccService` class.

Function prototype:

VB .NET:

```
Public Shared Function WinBufToArray(ByVal memHandle As Integer,
ByVal dataArray As System.UInt16( ), ByVal firstPoint As Integer,
ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

```
Public Shared Function WinBufToArray(ByVal memHandle As Integer,
ByRef dataArray As Short, ByVal firstPoint As Integer, ByVal
numPoints As Integer) As MccDaq.ErrorInfo
```

C# .NET:

```
public static MccDaq.ErrorInfo WinBufToArray (int memHandle, out
ushort dataArray, int firstPoint, int numPoints)
```

```
public static MccDaq.ErrorInfo WinBufToArray ( int memHandle, out
short dataArray, int firstPoint, int numPoints)
```

Parameters:

`memHandle`	This must be a memory handle that was returned by WinBufAlloc() when the buffer was allocated. The buffer should contain the data that you want to copy.
`dataArray`	The array that the data will be copied to.
`firstPoint`	Index of first point in memory buffer that data will be copied from.
`numPoints`	Number of data points to copy.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

This method copies data from a Windows global memory buffer to an array. This would typically be used to retrieve data from the buffer after executing an input scan method.

Using the `firstPoint` and `numPoints` parameters, it is possible to copy only a portion of the buffer to the array. This can be useful if you want foreground code to manipulate previously collected data while a `Background` scan continues to collect new data.

25

Miscellaneous Methods, Properties, and Delegates

Introduction

The methods and properties explained in this chapter do not as a group fit into a single category. They get and set board information, convert units, manage events and background operations, and perform serial communication operations.

BoardName property

Name of the board associated with an instance of the MccBoard class.

Member of the `MccBoard` class.

Function prototype:

VB .NET:	`Public ReadOnly Property BoardName As String`
C# .NET:	`public string BoardName [get]`

DisableEvent()

Disables one or more event conditions, and disconnects their user-defined handlers.

Member of the `MccBoard` class.

Function prototype:

VB .NET: `Public Function DisableEvent(ByVal eventType As MccDaq.EventType ) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo DisableEvent(MccDaq.EventType eventType)`

Parameters:

`eventType` Specifies one or more event conditions that will be disabled. More than one event type can be specified by bitwise OR'ing the event types. Note that specifying an event that has not been enabled is benign and will not cause any errors. Refer to "eventType" on page 284 for a list of valid event types.

To disable all events in a single call, use `AllEventTypes`.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

For most event types, this method cannot be called while any background operations (`AInScan()`, `APretrig()`, or `AOutScan()`) are active. Perform a `StopBackground()` before calling `EnableEvent()`. However, for `OnExternalInterrupt` events, you can call `DisableEvent()` while the board is actively generating events.

Important

In order to understand the functions, you must read the board-specific information contained in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Review and run the example programs before attempting any programming of your own. Following this advice will save you hours of frustration, and possibly time wasted holding for technical support.

This note, which appears elsewhere, is especially applicable to this method. Now is the time to read the board-specific information for your board (see the *Universal Library User's Guide*). We suggest that you make a copy of that page to refer to as you read this manual and examine the example programs.

EnableEvent()

This method binds one or more event conditions to a user-defined callback function. Upon detection of an event condition, the user-defined function is invoked with board- and event-specific data. Detection of event conditions occurs in response to interrupts. Typically, this method is used in conjunction with interrupt driven processes such as `AInScan`, `APretrig`, or `AOutScan`.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function EnableEvent(ByVal eventType As MccDaq.EventType ,
ByVal eventParameter As Integer, ByVal callbackFunc As
MccDaq.EventCallback, ByVal userData As IntPtr) As MccDaq.ErrorInfo
```

```
Public Function EnableEvent(ByVal eventType As MccDaq.EventType,
ByVal eventParameter As System.UInt32, ByVal callbackFunc As
MccDaq.EventCallback, ByVal userData As IntPtr) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo EnableEvent(MccDaq.EventType eventType, uint
eventParameter, MccDaq.EventCallback callbackFunc, System.IntPtr
userData)
```

```
public MccDaq.ErrorInfo EnableEvent(MccDaq.EventType eventType, int
eventParameter, MccDaq.EventCallback callbackFunc, System.IntPtr
userData)
```

Parameters:

`eventType`
Specifies one or more event conditions that will be bound to the user-defined callback function. More than one event type can be specified by bitwise OR'ing the event types. Set it to one of the constants in the "eventType" section below.

`eventParameter`
Additional data required to specify some event conditions such as the `OnDataAvailable` event. For `OnDataAvailable` events, this is used to determine the minimum number of samples to acquire during an analog input scan before generating the event.

Most event conditions ignore this value.

`callbackFunc`
A delegate type that is the user-defined callback function to handle the above event type(s). A *delegate* is a data structure that refers either to a static method, or to a class instance and an instance method of that class.

The callbackFunc needs the same parameters as the EventCallback delegate declaration. Refer to the "EventCallback delegate" section on page 286 for proper syntax and return values.

`userData`
Reference to user-defined data that is passed to the EventCallback delegate. This parameter is NOT de-referenced by the library or its drivers; as a consequence, a NULL pointer can be supplied.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

eventType parameter values:

`OnScanError`
Generates an event upon detection of a driver error during `Background` input and output scans. This includes `OverRun`, `UnderRun`, and `TooFew` errors.

`OnExternalInterrupt`
For some digital and counter boards, generates an event upon detection of a pulse at the External Interrupt pin.

`OnPretrigger`
For `APretrig()`, generates an event upon detection of the first trigger.

`OnDataAvailable`	Generates an event whenever the number of samples acquired during an analog input scan increases by EventParam samples or more. Note that for `BlockIo` scans, events will be generated on packet transfers; for example, even if EventParam is set to 1, events will only be generated every packet-size worth of data (256 samples for the PCI-DAS1602) for aggregate rates greater than 1 kHz for the default `AInScan()` mode. For `APretrig()`, the first event is not generated until a minimum of EventParam samples after the pretrigger.
`OnEndOfAiScan`	Generates an event upon completion or fatal error of a `AInScan()` or `APretrig()`. This event is NOT generated when scans are aborted using `StopBackground()`.
`OnEndOfAoScan`	Generates an event upon completion or fatal error of a `AOutScan()`. This event is not generated when scans are aborted using `StopBackground()`.

Notes:

`EnableEvent()` cannot be called while any background operations (`AInScan()`, `APretrig()`, or `AOutScan()`) are active. If a background operation is in progress when EnableEvent() is called, EnableEvent will return the AlreadyActive error. You should perform a StopBackground() before calling EnableEvent.

Events can be generated no faster than the user callback function can handle them. If an event type becomes multiply signaled before the event handler returns, events will be merged, such that the event handler is called once per event type, and the event handler is supplied with the event data corresponding to the latest event. In addition, if more than one event type becomes signaled, the event handler for each event type is called in the same order in which they are listed above.

Events are generated while handling board-generated interrupts. As a consequence, using `StopBackground()` to abort background operations will not generate `OnEndOfAoScan` or `OnEndOfAiScan` events. However, the event handlers can be called directly immediately after calling `StopBackground()`.

Important

In order to understand the functions, you must read the board -specific information section found in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Review and run the example programs prior to attempting any programming of your own. Following this advice will save you hours of frustration, and possibly time wasted holding for technical support.

This note, which appears elsewhere, is especially applicable to this method. Read the board-specific information for your board (see the *Universal Library User's Guide*). We suggest that you make a copy of that page to refer to as you read this manual and examine the example programs.

EventCallback delegate

The EventCallback delegate is called as a parameter of the `EnableEvent()` method. A delegate is a data structure that refers either to a static method, or to a class instance and an instance method of that class.

You create the data structure using the prototype shown below. You call the delegate by passing either it's address or a pointer to the delegate to the callbackFunc parameter of the `EnableEvent()` method.

Delegate prototype:

C# .NET:

```
public delegate void EventCallback( int BoardNum, MccDaq.EventType
EventType, uint EventData, IntPtr pUserData);
```

VB .NET:

```
Public Sub MyCallback(ByVal BoardNum As Integer, ByVal EventType As
MccDaq.EventType, ByVal EventData As UInt32, ByVal pUserData As
System.IntPtr)
```

Parameters:

`BoardNum`	Indicates which board caused the event.
`EventType`	Indicates which event occurred.
`EventData`	Board-specific data associated with this event. Set it to one of the constants in the "EventData parameter values" section below.
`pUserData`	Pointer to or reference of data supplied by the userData parameter in the `EnableEvent()` method. Note that before using this parameter value, it must be cast to the same data type as it was passed to `EnableEvent()`.

Returns:

`pUserData` – Returns value specified by the `userData` parameter in `EnableEvent()`.

EventData parameter values:

`OnScanError`	The Error code of the scan error.
`OnExternalInterrupt`	The number of interrupts generated since enabling the `ON_EXTERNAL_INTERRUPT` event.
`OnPretrigger`	The number of pretrigger samples available at time of pretrigger. Value is invalid for some boards when a `TOOFEW` error occurs. See board details.
`OnDataAvailable`	The number of samples acquired since the start of scan.
`OnEndOfAiScan`	The total number of samples acquired upon scan completion or end.
`OnEndOfAoScan`	The total number of samples output upon scan completion or end.

FlashLED()

Causes the LED on a USB device to flash.

Member of the `MccBoard` class.

Function prototype:

VB .NET: `Public Function FlashLED() As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo FlashLED()`

FromEngUnits()

Converts a voltage (or current) in engineering units to a D/A count value for output to a D/A.

Member of the MccBoard class.

Function prototype:

VB .NET:

```
Public Function FromEngUnits(ByVal range As MccDaq.Range , ByVal engUnits As Single, ByRef dataVal As Short) As MccDaq.ErrorInfo
```

```
Public Function FromEngUnits(ByVal range As MccDaq.Range, ByVal engUnits As Single, ByRef dataVal As System.UInt16) As MccDaq.ErrorInfo
```

C# .NET:

```
public MccDaq.ErrorInfo FromEngUnits(MccDaq.Range range, float engUnits, out ushort dataVal)
```

```
public MccDaq.ErrorInfo FromEngUnits(MccDaq.Range range, float engUnits, out short dataVal)
```

Parameters:

`range` D/A voltage (or current) range. Some D/A boards have programmable voltage ranges, others set the voltage range via switches on the board. In either case, the selected range must be passed to this method. Refer to Table 14-1 on page 155 for a list of valid range settings.

Each D/A board supports different voltage and/or current ranges. Refer to board specific information for the list of ranges supported by each board.

`engUnits` The voltage (or current) value to set the D/A to. Set the value to be within the range specified by the `range` parameter.

`dataVal` The method returns a D/A count to this variable that is equivalent to the `engUnits` parameter.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`dataVal` – the binary counts equivalent to `engUnits` is returned here

GetBoardName()

Returns the board name of a specified board.

Member of the `MccService` class.

Function prototype:

VB .NET: `Public Shared Function GetBoardName(ByVal boardNumber As Integer, ByRef boardName As String) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo GetBoardName(int boardNumber, ref string boardName)`

Parameters:

`boardNumber` Refers either to the board number associated with a board when it was installed, or `GETFIRST` or `GETNEXT`.

`boardName` A null-terminated string variable that the board name is returned to. Refer to the Appendix, "Board Type Codes," in the *Universal Library User's Guide* (available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

Returns:

An `ErrorInfo` object that indicates the status of the operation.

boardName - return string containing the board name.

Notes:

There are two distinct ways of using this function:

- Pass a board number as the `BoardNum` argument. The string that is returned describes the board type of the installed board.
- Set `BoardNum` to `GETFIRST` or `GETNEXT` to get a list of all board types that are supported by the library. Set `BoardNum` to `GETFIRST` to get the first board type in the list of supported boards. Subsequent calls with `Board=GETNEXT` returns each of the other board types supported by the library. When you reach the end of the list, `BoardName` is set to an empty string. Refer to the ulgt04 example program in the installation directory for more details.

GetStatus()

Returns the status about the background operation currently running.

Member of the `MccBoard` class.

Function prototype:

VB .NET:	`Public Function GetStatus(ByRef status As Short, ByRef curCount As Integer, ByRef curIndex As Integer, ByVal functionType As MccDaq.FunctionType ) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo GetStatus(out short status, out int curCount, out int curIndex, MccDaq.FunctionType functionType)`

Parameters:

`status` — Status indicates whether or not a background process is currently executing.

`curCount` — Specifies how many points have been input or output. It can be used to gauge how far along the operation is towards completion. Generally the `curCount` will return the total number of samples collected at the time of the call to `GetStatus()`.

However, when `Continuous` and `Background` options are both set, `curCount` behavior depends on the board type and transfer mode. This value may recycle as the circular buffer recycles, or may continuously increment with the number of counts transferred. Also, `curCount` may not update on each sample. For example, when running in `BlockIo` mode, `curCount` updates after each packet of data has been transferred. The packet size is board-dependent. Refer to the *Universal Library User's Guide* for board-specific information.

`curIndex` — `curIndex` is an index into the data buffer that points at the start of the last completed channel scan. It can be used to provide a real time display for a background operation. DataBuffer[`curIndex`] points to the start of the last complete channel scan that was put in or taken out of the buffer. You should expect `curIndex` to increment by the number of channels in the scan as well. If no points in the buffer have been accessed yet, `CurIndex` will equal -1. This value can also behave differently when `Continuous` and `Background` options are both set (see `CurCount` description). Refer to board specific information for details.

If you use the `ConvertData` option with either the `Continuous` option or with pre-triggering functions, `curCount` returns the index of the last A/D sample, rather than the start of the last completed channel scan.

For many background operations `curCount` = `curIndex`. For pre-trigger inputs though, they are different. If the hardware allows background trigger operations, `curCount` indicates how many points of the `totalCount` have been collected. `curCount` will rise to `pretrigCount`, stop until the trigger occurs then rise to `totalCount`. `curIndex`, though, will constantly increase and reset as it goes around and around the circular buffer while waiting for the trigger to occur.

`functionType` — Specifies which scan to retrieve status information about. Set it to one of the constants in the "functionType parameter values" section on page 291.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`Status` — `Idle` - No background operation has been executed

`Running` - Background operation still underway

`curCount` - current number of samples collected `curIndex` - Current sample index

functionType parameter values:

`AiFunction`	Specifies analog input scans started with `AInScan()` or `APretrig()`.
`AoFunction`	Specifies analog output scans started with `AOutScan()`.
`DiFunction`	Specifies digital input scans started with `DInScan()`.
`DoFunction`	Specifies digital output scans started with `DOutScan()`.
`CtrFunction`	Specifies counter background operations started with `CStoreOnInt()`.

InByte()

Reads a byte from a hardware register on a board.

Member of the `MccBoard` class.

Function prototype:

VB .NET:	`Public Function InByte(ByVal portNum As Integer) As Integer`
C# .NET:	`public int InByte(int portNum)`

Parameters:

`portNum`	Register within the board. Boards are set to a particular base address. The registers on the boards are at addresses that are offsets from the base address of the board (BaseAdr + 0, BaseAdr + 2, etc). Set this parameter to the offset for the desired register. This method takes care of adding the base address to the offset, so that the board's address can be changed without changing the code.

Returns:

The current value of the specified register

Notes:

`InByte()` is used to read 8 bit ports. `InWord()` is used to read 16-bit ports.

This method was designed for use with ISA bus boards. Use with PCI bus boards is not recommended.

InWord()

Reads a word from a hardware register on a board.

Member of the `MccBoard` class.

Function prototype:

VB .NET: `Public Function InWord(ByVal portNum As Integer) As Integer`

C# .NET: `public int InWord(int portNum)`

Parameters:

`portNum` Register within the board. Boards are set to a particular base address. The registers on the boards are at addresses that are offsets from the base address of the board (BaseAdr + 0, BaseAdr + 2, etc).

Set this parameter to the offset for the desired register. This method takes care of adding the base address to the offset, so that the board's address can be changed without changing the code.

Returns:

The current value of the specified register.

Notes:

`InByte()` is used to read 8-bit ports. InWord() is used to read 16 bit ports.

This method was designed for use with ISA bus boards. Use with PCI bus boards is not recommended.

OutByte()

Writes a byte to a hardware register on a board.

Member of the `MccBoard` class.

Function prototype:

VB .NET:	`Public Function OutByte(ByVal portNum As Integer, ByVal portVal As Integer) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo OutByte(int portNum, int portVal)`

Parameters:

`portNum`	Register within the board. Boards are set to a particular base address. The registers on the boards are at addresses that are offsets from the base address of the board (BaseAdr + 0, BaseAdr + 2, etc). Set this parameter to the offset for the desired register. This method takes care of adding the base address to the offset, so that the board's address can be changed without changing the code.
`portVal`	Value that is written to the register.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

`OutByte()` is used to write to 8-bit ports. `OutWord()` is used to write to 16-bit ports.

This method was designed for use with ISA bus boards. Use with PCI bus boards is not recommended.

OutWord()

Writes a word to a hardware register on a board.

Member of the `MccBoard` class.

Function Prototype:

VB .NET: `Public Function OutWord(ByVal portNum As Integer, ByVal portVal As Integer) As MccDaq.ErrorInfo`

C# .NET: `public MccDaq.ErrorInfo OutWord(int portNum, int portVal)`

Parameters:

`portNum` Register within the board. Boards are set to a particular base address. The registers on the boards are at addresses that are offsets from the base address of the board (BaseAdr + 0, BaseAdr + 2, etc).

Set this parameter to the offset for the desired register. This method takes care of adding the base address to the offset, so that the board's address can be changed without changing the code.

`PortVal` Value that is written to the register.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

`OutByte()` is used to write to 8-bit ports. `OutWord()` is used to write to 16-bit ports.

This method was designed for use with ISA bus boards. Use with PCI bus boards is not recommended.

RS485()

Sets the direction of RS-485 communications port buffers.

Member of the `MccBoard` class.

Function prototype:

VB .NET:	`Public Function RS485(ByVal transmit As MccDaq.OptionState , ByVal receive As MccDaq.OptionState ) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo RS485(MccDaq.OptionState transmit, MccDaq.OptionState receive)`

Parameters:

`transmit`	Set to `Enabled` or `Disabled`. The transmit RS-485 line driver is turned on. Data written to the RS-485 UART chip is transmitted to the cable connected to that port.
`receive`	Set to `MccDaq.OptionState.Enabled` or `MccDaq.OptionState.Disabled`. The receive RS-485 buffer is turned on. Data present on the cable connected to the RS-485 port is received by the UART chip.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

Notes:

You can simultaneously enable or disable the transmit and receive buffers. If both are enabled, data written to the port is also received by the port. For a complete discussion of RS485 network construction and communication, refer to the CIO-COM485 or PCM-COM485 hardware manual.

StopBackground()

Stops one or more subsystem background operations that are in progress for the specified board. Use this method to stop any method that is running in the background. This includes any method that was started with the Background option, as well as `CStoreOnInt()` (which always runs in the background).

Execute `StopBackground()` after normal termination of all background functions to clear variables and flags.

Member of the `MccBoard` class.

Function prototype:

VB .NET:	`Public Function StopBackground(ByVal funcType As MccDaq.FunctionType ) As MccDaq.ErrorInfo`
C# .NET:	`public MccDaq.ErrorInfo StopBackground(MccDaq.FunctionType funcType)`

Parameters:

`functionType`	Specifies which background operation to stop. Set it to one of the constants in the "functionType parameter values" section below.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

functionType parameter values:

`AiFunction`:	Specifies analog input scans started with `AInScan()` or `APretrig()`.
`AoFunction`	Specifies analog output scans started with `AOutScan()`.
`DiFunction`	Specifies digital input scans started with `DInScan()`.
`DoFunction`	Specifies digital output scans started with `DOutScan()`.
`CtrFunction`	Specifies counter background operations started with `CStoreOnInt().`

ToEngUnits()

Converts an A/D count value to an equivalent voltage value.

Member of the `MccBoard` class.

Function prototype:

VB .NET:

```
Public Function ToEngUnits(ByVal range As MccDaq.Range , ByVal dataVal As Short, ByRef engUnits As Single) As MccDaq.ErrorInfo
```

```
Public Function ToEngUnits(ByVal range As MccDaq.Range, ByVal dataVal As System.UInt16, ByRef engUnits As Single) As MccDaq.ErrorInfo
```

C# .NET:

```
Public MccDaq.ErrorInfo ToEngUnits(MccDaq.Range range, ushort dataVal, out float engUnits )
```

```
Public MccDaq.ErrorInfo ToEngUnits(MccDaq.Range range, short dataVal, out float engUnits )
```

Parameters:

`range`	A/D voltage (or current) range. Some A/D boards have programmable voltage ranges, others set the voltage range via switches on the board. In either case, the selected range must be passed to this method. Each A/D board supports different voltage and/or current ranges. Refer to Table 14-1 on page 155 for a list of valid range settings. Refer to board specific information for a list of the supported A/D ranges of each board.
`dataVal`	A/D count returned from an A/D board.
`engUnits`	The voltage (or current) value that is equivalent to `dataVal` is returned to this variable. The value will be within the range specified by the `range` parameter.

Returns:

An `ErrorInfo` object that indicates the status of the operation.

`engUnits` – the engineering units value equivalent to `dataVal` is returned to this variable.

Appendix

Error Codes

The following table lists error codes that are returned when running Universal Library or Universal Library for .NET.

Universal Library .NET errors can be referenced from the MccDaq.ErrorInfo.Message property.

Each entry in the list has four parts: the error code number, its symbolic name, its error message, and an explanation. Both the Universal Library function and its Universal Library .NET equivalent method are referred to when appropriate. Error code and error messages are identical for both programming libraries. The only difference in the error names used by each library is the case—the Universal Library error names are all uppercase (NOERRORS, etc.), while the Universal Library for .NET error names are mixed case (NoErrors, etc.).

Error number	Error name	Error message
0	**NOERRORS**	No error has occurred
	The function executed successfully.	
1	**BADBOARD**	Invalid board number
	The `BoardNum` argument that was specified does not match any of the boards that are listed in the configuration file. Run the configuration program to check which board numbers are configured.	
2	**DEADDIGITALDEV**	Digital device is not responding - is base address correct?
	The digital device on the specified board is not responding. Either the board was installed incorrectly or the board is defective. Run the configuration program and make sure that the correct board was installed.	
3	**DEADCOUNTERDEV**	Counter device is not responding - is base address correct?
	The counter device on the specified board is not responding. Either the board was installed incorrectly or the board is defective. Run the configuration program and make sure that the correct board was installed.	
4	**DEADDADEV**	D/A is not responding - is base address correct?
	The D/A device on the specified board is not responding. Either the board was installed incorrectly or the board is defective. Run the configuration program and make sure that the correct board was installed.	
5	**DEADADDEV**	A/D is not responding - is base address correct?
	The A/D device on the specified board is not responding. Either the board was installed incorrectly or the board is defective. Run the configuration program and make sure that the correct board was installed.	
6	**NOTDIGITALCONF**	Selected board does not have digital I/O
	A digital I/O function or method was called with a board number that referred to a board that does not support digital I/O. Run the configuration program to see which type of board that board number refers to.	
7	**NOTCOUNTERCONF**	Selected board does not have a counter
	A counter function or method was called with a board number that referred to a board that does not have a counter. Run the configuration program to see which type of board that board number refers to.	

Error number	Error name	Error message
8	**NOTDACONF**	Selected board does not have a D/A

An analog output function or method was called with a board number that referred to a board that does not have an analog output (D/A). Run the configuration program to see which type of board the board number refers to.

9	**NOTADCONF**	Selected board does not have an A/D

An analog input function or method was called with a board number that referred to a board that does not have an analog input (A/D). Run the configuration program to see which type of board that board number refers to.

10	**NOTMUXCONF**	Selected board does not have thermocouple inputs

A thermocouple input function or method was called with a board number that does not support thermocouple inputs or is not connected to an EXP board. Run the configuration program to view/change the board configuration.

11	**BADPORTNUM**	Invalid digital port number

The port number specified for a digital I/O function or method does not exist on the specified board.

12	**BADCOUNTERDEVNUM**	Invalid counter device

The `CounterNum` argument specified for a counter function or method references a counter that does not exist on the specified board.

13	**BADDADEVNUM**	Invalid D/A device

The D/A channel that was specified for an analog output function or method does not exist on the specified board.

14	**BADSAMPLEMODE**	Invalid sample mode

A sample mode that is not supported on this board (`SINGLEIO`, `DMAIO` or `BLOCKIO`) was specified in the `Options` argument. Try running the function or method without setting any of the Sample Mode options.

15	**BADINT**	Board configured for invalid interrupt level

No interrupt was selected in *Insta*Cal and one is required, or the board is set for "compatible mode" and the interrupt level selected is not supported in this mode. Interrupts above 7 are not valid in compatible mode. Either change the switch setting on the board to "enhanced mode", or change the interrupt level with the configuration program to something less than 8.

16	**BADADCHAN**	Invalid A/D channel number

An invalid channel argument was passed to an analog input function or method . The range of valid channel numbers depends on which A/D board you are using - refer to the board manual. For some boards it also depends on how the board is configured (with a switch). For those boards run the configuration program and check how many channels the board is configured for.

17	**BADCOUNT**	Invalid count

An invalid `Count` argument was specified to a function or method . If this error occurs during `cbAInScan()`/`AInScan()`, increasing the `Count` should correct the problem. For boards using DMAIO, adjust the data buffer and Count above (`HighChan-LowChan`+1)*Rate/100 for `CONTINUOUS` mode scans. However, those boards using `BLOCKIO`, require a user buffer and Count large enough to hold at least one half FIFO worth of samples (typically, 512 samples) for `CONTINUOUS` mode scans.

Error number	Error name	Error message
18	**BADCNTRCONFIG**	Invalid counter configuration specified

An invalid Config argument was passed to `cbC8254Config()/C8254Config()`. The only legal values are `HIGHONLASTCOUNT`, `ONESHOT`, `RATEGENERATOR`, `SQUAREWAVE`, `SOFTWARESTROBE` and `HARDWARESTROBE`.

19	**BADDAVAL**	Invalid D/A value

An invalid D/A value was passed as an argument to an analog output function or method . The only legal values are 0 to 4095 for 12-bit boards or 0 to 65,535 for 16-bit boards (see the "Note on Basic signed integers" at the beginning of the "Counter Boards" chapter in the *Universal Library User's Guide* available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).

20	**BADDACHAN**	Invalid D/A channel number

An invalid D/A channel was passed as an argument to an analog output function or method . The legal range of values depends on which D/A board you are using. Refer to the board manual to find how many D/A channels it has.

22	**ALREADYACTIVE**	Background operation already in progress

An attempt was made to start a second background process on the same board before the first one had completed. Background processes are started whenever the `BACKGROUND` option is used by `cbCStoreOnInt()/CStoreOnInt()`. To stop a background operation, call `cbStopBackground()/StopBackground()`. To wait for a background process to complete call `cbGetStatus()`/(`GetStatus()` and wait for `Status` = `IDLE`.

23	**PAGEOVERRUN**	DMA transfer crossed page boundary, may have gaps in data

When a DMA transfer crosses a 64K memory page boundary on boards without FIFO buffers, there may be a small gap (missing samples) in the data. For applications requiring high speed transfers of greater than 32K samples, please select a board with a FIFO buffer. For boards without, check the data for gaps and do not specify rates over that at which gapless data may be taken. This is system-specific so you must determine the rate by experimentation.

24	**BADRATE**	Invalid sampling rate

Invalid sampling rate argument was specified. The rate was either zero, a negative number or it was higher than the selected board supports. Refer to board-specific information for board maximum rates.

25	**COMPATMODE**	Board switches set for Compatible mode

An operation was attempted that is not possible when the board's switch is set for 'compatible' operation. The most likely causes are due to using the `BLOCKIO` option or the pre-triggering functions. Either turn off the 'compatible' mode switch on the board or don't use the `BLOCKIO` option or the pre-triggering functions.

26	**TRIGSTATE**	Incorrect initial trigger state - trigger must start at TTL low

Boards that use "polled gate" triggering require that the trigger be "off" when a pre-trigger function is first called. It then waits for the trigger signal. Make sure that the Trigger Input line (usually D0) is held at TTL low before calling the pre-trigger function.

27	**ADSTATUSHUNG**	A/D is not responding

The A/D board is not responding as it should. Usually indicates some kind of hardware problem - either defective hardware or more than one board at the same base address.

Error number	Error name	Error message
28	**TOOFEW**	Trigger occurred before the requested number of samples were collected

A pre-trigger function or method was called and the trigger signal occurred before the requested number of samples could be collected. This is only a warning message. The function or method continued anyway. The data that was returned to the array will contain fewer than the expected number of points. The function or method will return the actual number of pre-trigger points and the total number of points. You can use these two values to find your way around the data in the array.

29	**OVERRUN**	Data overrun - data was lost

Data was lost during an analog input because the computer could not keep up with the A/D sampling rate. This typically can only happen with the file input functions or methods, or by using `SINGLIO` mode. Possible solutions include lowering the sampling rate, defragmenting the "streamer" file, switching to a RAM disk, or lowering the count.

30	**BADRANGE**	Invalid voltage or current range

Invalid Range argument was specified to an analog input or output function or method . The board does not support the gain you specified. Refer to board-specific information for a list of allowable ranges.

31	**NOPROGGAIN**	This A/D board does not have programmable gain

Invalid `Range` argument was passed to an analog input function or method . The selected board does not support programmable gains so the only valid `Range` argument is 0. (This argument is ignored for these board types in later versions of the library.)

32	**BADFILENAME**	Specified file name is not valid

The `FileName` argument that was passed to a file function or method is not valid. It is either an empty string or a NULL pointer.

33	**DISKISFULL**	Disk is full, could not complete operation

A file operation failed before completing because the disk that it was writing to is full. Try erasing some files from the disk. If this error occurred during either `cbFileAInScan()/FileAInScan()` or `cbFilePretrig()/FilePretrig()`, it indicates another problem. The disk space for these commands should have been previously allocated with the MAKESTRM.EXE program. If this error is generated when data is being collected it indicates that you did not allocate a large enough file with MAKESTRM.EXE.

34	**COMPATWARN**	Board switch set to compatible mode - sampling speed may be limited

The board's switch is set for "compatible mode." When in "compatible mode," `BLOCKIO` transfers are not possible. `BLOCKIO` sampling was specified but it has automatically been changed to `DMAIO` transfers. The maximum sampling rate will be limited to the maximum rate for DMA transfers. Change the "compatible mode" switch on the board if you want to use `BLOCKIO` transfers.

35	**BADPOINTER**	Pointer is not valid

An invalid (NULL) pointer was passed as an argument/parameter to a function or method .

37	**RATEWARNING**	Sample rate may be too fast for `SINGLEIO` mode

The specified sampling rate MAY be too high. The maximum allowable sampling rate depends very much on the computer that the program is running on. This warning is generated based on the slowest CPU speed. Your computer may be able to sustain faster rates, but, you should expect the computer to lock up (fail to respond to keyboard input) if you do exceed the sampling rate your computer can sustain.

Error number	Error name	Error message
38	**CONVERTDMA**	CONVERTDATA cannot be used with DMAIO and BACKGROUND

The CONVERTDATA and BACKGROUND options can not be used together when the board is transferring data via DMA. Possible solutions include: Use cbAConvertData()/AConvertData() to convert the data after it is collected. Don't use BACKGROUND option. Use BLOCKIO option if your A/D board supports it. Use SINGLEIO option if your computer is fast enough to support the selected sampling rate.

39	**DTCONNECTERR**	Board does not support DTCONNECT option

The DTCONNECT Option was passed to an analog input function or method . The selected board does not support that option.

40	**FORECONTINUOUS**	CONTINUOUS can only be run with BACKGROUND

The CONTINUOUS option was passed to a function or method without also setting the BACKGROUND option. This is not allowed. Any time you set the CONTINUOUS option you must also set the BACKGROUND option.

41	**BADBOARDTYPE**	This function or method can not be used with this board

An attempt was made to call a function or method for a board that does not support that function or method .

42	**WRONGDIGCONFIG**	Digital port not configured correctly for requested operation

Some of the digital bits or ports (FIRSTPORTA - EIGHTHPORTCH) must be configured as inputs OR outputs but not both. An attempt was made to use a digital input function or method on a port or bit that was configured as an output or vice versa. Use cbDConfigPort()/DConfigPort() or cbDConfigBit()/DConfigBit() to switch a port's (or bit's) direction. If the board you are using contains configurable port types and you do not call cbDConfigPort()/DConfigPort() or cbDConfigBit()/DConfigBit() in your program, then all of the configurable ports will be in an unknown state (input or output).

43	**NOTCONFIGURABLE**	This digital port is not configurable (it's an In/Out port)

cbDConfigPort()/DConfigPort() or cbDConfigBit()/DConfigBit() was called for a port that is not configurable. Check the PortNum argument passed to cbDConfigPort() and make sure that it is in the range FIRSTPORTA - EIGHTHPORTCH. If PortNum is AUXPORT, make sure your hardware supports configuration of this port type. If not then there is no need to call this function or method .

44	**BADPORTCONFIG**	Invalid digital port configuration

The Direction argument passed to cbDConfigPort()/DConfigPort() or cbDConfigBit()/DConfigBit() is invalid. It must be set to either DIGITALIN or DIGITALOUT.

45	**BADFIRSTPOINT**	FirstPoint number is not valid

The FirstPoint argument to cbFileRead ()/FileRead() is invalid. It is either a negative number or it is larger then the number of points in the file.

46	**ENDOFFILE**	Attempted to read past the end of the file

cbFileRead()/FileRead() attempted to read beyond the end of the file. Check the file length with cbFileGetInfo()/FileGetInfo() and make sure that the FirstPoint and Count arguments to cbFileRead()/FileRead() are correct for that file length.

47	**NOT8254CTR**	This board does not have an 8254 counter

cbC8254Config()/C8254Config() was called for a board that has a counter but not an 8254 counter. This function or method can only be used with an 8254 counter.

Error number	Error name	Error message
48	**NOT9513CTR**	This board does not have a 9513 counter
	`cbC9513Config()/C9513Config()` was called for a board that has a counter but not a 9513 counter. This function or method can only be used with an 9513 counter.	
49	**BADTRIGTYPE**	Invalid `TrigType`
	`cbATrig()`/(`ATrig()`) was called with an invalid `TrigType` argument. It must be set to either `TRIGABOVE` or `TRIGBELOW`.	
50	**BADTRIGVALUE**	Invalid `TrigValue`
	`cbATrig()`/(`ATrig()`) was called with an invalid `TrigValue` argument. It must be in the range 0 to 4095 for 12-bit boards or 0 to 65535 for 16-bit boards (see the "Note on Basic signed integers" at the beginning of the "Counter Boards" chapter in the *Universal Library User's Guide,* available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf).	
52	**BADOPTION**	Invalid `Option` specified for this function or method
	The `Options` argument contains an option that is not valid for this function or method .	
53	**BADPRETRIGCOUNT**	Invalid `PretrigCount` specified
	Either `cbAPretrig()/APretrig()` or `cbFilePretrig()/FilePretrig()` was called with an invalid `PretrigCount` argument. The pre-trigger count must not be < 0 and must be less than `TotalCount`-512. It also must be less than 32k for `cbAPretrig()/APretrig()` and less than 16k for `cbFilePretrig()/FilePretrig()`.	
55	**BADDIVIDER**	Invalid `FOutDivider` value
	The `FOutDivider` argument to `cbC9513Init()` (`C9513Init()`) is not valid. It must be in the range 0 to 15.	
56	**BADSOURCE**	Invalid `FOutSource` value
	The `FOutSource` argument to `cbC9513Init()` (`C9513Init()`) is not valid. It must be one of the following values `CTRINPUT1, CTRINPUT2, CTRINPUT3, CTRINPUT4, CTRINPUT5, GATE1, GATE2, GATE3, GATE4, GATE5, FREQ1, FREQ2, FREQ3, FREQ4, FREQ5` (i.e. 0 to 15).	
57	**BADCOMPARE**	Invalid `Compare` value
	One or both of the compare arguments to `cbC9513Init()/C9513Init()` are not valid. They must be set to (`CB`)`ENABLED` or (`CB`)`DISABLED` (1 or 0).	
58	**BADTIMEOFDAY**	Invalid `TimeOfDay` value
	The `TimeOfDay` argument to `cbC9513Init()/C9513Init()` is not valid. It must be set to either (`CB`)`ENABLED` or (`CB`)`DISABLED` (1 or 0).	
59	**BADGATEINTERVAL**	Invalid `GateInterval` value
	The `GateInterval` argument to `cbCFreqIn()/CFreqIn()` is not valid. It must be greater than 0.	
60	**BADGATECNTRL**	Invalid `GateControl` value
	The `GateControl` argument to `cbC9513Config()/C9513Config()` is not valid. It must be in the range 0 to7.	
61	**BADCOUNTEREDGE**	Invalid `CounterEdge` value
	The `CounterEdge` argument to `cbC9513Config()/C9513Config()` is not valid. It must be set to either `POSITIVEEDGE` or `NEGATIVEEDGE`.	
62	**BADSPCLGATE**	Invalid `SpecialGate` value
	The `SpecialGate` argument to `cbC9513Config()/C9513Config()` is not valid. It must be set to either (`CB`)`ENABLED` or (`CB`)`DISABLED` (1 or 0).	

Error number	Error name	Error message
63	**BADRELOAD**	Invalid Reload value
	The Reload argument to cbC9513Config() (C9513Config()) is not valid. It must be set to either LOADREG or LOADANDHOLDREG	
64	**BADRECYCLEFLAG**	Invalid RecycleMode value
	The RecycleMode argument to cbC9513Config()/C9513Config() is not valid. It must be set to either (CB)ENABLED or (CB)DISABLED (1 or 0).	
65	**BADBCDFLAG**	Invalid BCDMode value
	The BCDMode argument to cbC9513Config()/C9513Config() is not valid. It must be set to either (CB)ENABLED or (CB)DISABLED (1 or 0).	
66	**BADDIRECTION**	Invalid CountDirection value
	The CountDirection argument to cbC9513Config() (C9513Config()) is not valid. It must be set to either COUNTUP or COUNTDOWN.	
67	**BADOUTCONTROL**	Invalid OutputControl value
	The OutputControl argument to cbC9513Config() (C9513Config()) is not valid. It must be set to either ALWAYSLOW, HIGHPULSEONTC, TOGGLEONTC, DISCONNECTED or LOWPULSEONTC.	
68	**BADBITNUMBER**	Invalid BitNum specified
	The BitNum argument to cbDBitIn() or cbDBitOut() (DBitIn() or DBitOut()) is not valid. The valid range of bit numbers depends on the selected board. If it is a DIO24 compatible board the maximum bit number is 23. If it's a DIO96, the maximum bit number is 95. (see board-specific information in the *Universal Library User's Guide,* available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf, or in your hardware manual)	
69	**NONEENABLED**	None of the counter channels were enabled
	None of the counter channels were marked as (CB)ENABLED in the CntrControl array that was passed to cbCStoreOnInt()/CStoreOnInt(). At least one of the counter channels must be enabled.	
70	**BADCTRCONTROL**	An element of CntrControl array not set to DISABLED or ENABLED
	One of the elements of the CntrControl array that was passed to cbCStoreOnInt()/(CStoreOnInt()) was set to something other then (CB)ENABLED or (CB)DISABLED. The array must have at least ten elements and the first ten elements must be set to either (CB)ENABLED or (CB)DISABLED.	
71	**BADEXPCHAN**	Invalid EXP channel specified
	An invalid channel was passed to one of the thermocouple input commands. The channel number when using an EXP board must be >= 16. The maximum allowable channel number depends on which EXP board is being used (and how many of them). Refer to the board manual to find the number of channels.	
72	**WRONGADRANGE**	Board set to wrong A/D range for reading thermocouples
	A thermocouple input function or method was called to read an EXP board input. The EXP board is connected to an A/D board with hardware selected gain that is set to the wrong range. When using EXP boards with thermocouples, the A/D must be set to the –5 to +5 volt range when available. When using RTD sensors, the range is 0 to 10 V when available.	

Error number	Error name	Error message
73	**OUTOFRANGE**	Temperature input is out of range
	A thermocouple input function or method returned an invalid temperature. This usually indicates an open connection in the thermocouple or its connection to the mux board.	
74	**BADTEMPSCALE**	Invalid temperature scale specified
	The `Scale` argument/parameter to a thermocouple input function or method is not valid. It must be set to either `CELSIUS`, `FAHRENHEIT`, `KELVIN`, or `VOLT`.	
76	**NOQUEUE**	Specified board does not have channel/gain queue
	The function or method that was called requires that the board has a channel/gain queue. The specified board does not have a queue.	
77	**CONTINUOUSCOUNT**	Count must be > packet size to use Continuous mode
	The `Count` argument is not valid for continuous mode. Using `BLOCKIO` mode, the `Count` argument must be large enough to cause at least one interrupt. This is usually half the size of the boards FIFO (typical sizes are 256, 512, and 1024). See board-specific information in the *Universal Library User's Guide,* available on our web site at www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf) or in your hardware manual.	
78	**UNDERRUN**	D/A FIFO went empty during output
	The specified D/A output rate could not be sustained. This error should not normally occur.	
79	**BADMEMMODE**	Invalid memory mode specified
	The memory mode that was selected with `cbMemSetDTMode()` (`MemSetDTMode()`) is not one of the valid modes.	
80	**FREQOVERRUN**	Measured frequency too high for selected gating interval
	The `GateInterval` argument used with `cbCFreqIn()` (`CFreqIn()`) is too large to measure the frequency of the signal connected to the counter. The counter is overflowing. Decrease the gating interval to eliminate the error.	
81	**NOCJCCHAN**	A CJC Channel must be configured to make temperature measurements
	When the board was installed (with the *Insta*Cal installation program) no Cold Junction Compression (CJC) channel was selected. To use the temperature measurement functions or methods with thermocouples, you must first select a CJC channel on the A/D board and then rerun the installation program.	
82	**BADCHIPNUM**	Invalid ChipNum specified
	An invalid `ChipNum` argument was used with `cbC9513Init()`/`C9513Init()`. If the board is CTR05, set `ChipNum` to 0. If the board is a CTR10, set `ChipNum` to either 0 or 1.	
83	**DIGNOTENABLED**	The digital I/O on this board is not enabled
	When the board was installed (with the *Insta*Cal installation program), the expansion digital I/O was set to `DISABLED`. To use these digital I/O lines, you must enable the digital I/O on the board (with a jumper) and then re-run the installation program and set the digital I/O to `ENABLED`.	
84	**CONVERT16BITS**	`CONVERTDATA` option can not be used with 16 bit A/D converters
	When using a 16-bit A/D (DAS1600/16), if you try to use the `CONVERTDATA` option with `cbAInScan()`/`AInScan()` or call `cbAConvertData()`/`AConvertData()`, this error is returned. (This has been updated so that it is ignored for boards for which it is inappropriate in later versions of the library.)	

Error number	Error name	Error message
85	**NOMEMBOARD**	The `EXTMEMORY` option requires that a MEGA-FIFO be attached

Attempt to use a `cbMem_()` function or `Mem_()` method without a MEGA-FIFO board installed. Install MEGA-FIFO through *Insta*Cal.

86	**DTACTIVE**	No memory read/write allowed while DT transfer in progress

A read or write to a memory board was attempted while data was being transferred via DT-Connect.

87	**NOTMEMCONF**	Specified board is not a memory board

The specified board is not a memory board. This function or method only works with memory boards.

88	**ODDCHAN**	The first channel in scan and number of channels must be even (0, 2, 4, etc)

Some boards use a channel/gain queue that require the first channel in the queue and the number of channels in the queue always be an even channel. This error can occur even when you are not in the process of loading the queue. Some boards use the queue automatically with `cbAInScan()/AInScan()`. On those boards, the low channel must be an even number.

89	**CTRNOINIT**	Counter was not configured or initialized

You attempted to use `cbCLoad()` or `cbCIn()` (`CLoad()` or `CIn()`) before initializing and configuring the counter.

90	**NOT8536CTR**	This board does not have an 8536 counter chip

Attempt to use 8536 initialization or configuration on board without 8536 chip.

91	**FREERUNNING**	Board doesn't time A/D sampling. Collecting at fastest possible speed

This board does not have an A/D pacer mechanism and you have called `cbAInScan()`/(`AInScan()`. The A/D will be sampled in a tight software loop as fast as the CPU can execute the instructions. The speed of sampling is dependent on the computer and the concurrent tasks.

92	**INTERRUPTED**	Operation interrupted with Ctrl-C key

A foreground operation was stopped before completion because either the Ctrl-C or Ctrl-Break keys were pressed.

93	**NOSELECTORS**	No selector could be allocated

A Windows selector required by the library could not be allocated. Close any open Windows applications that are nor required to be running and try again.

94	**NOBURSTMODE**	This board does not support burst mode

An attempt was made to use the `BURSTMODE` option on a board which does not support that option.

95	**NOTWINDOWSFUNC**	This function is not available in Windows library

The library function you called is not supported in the current revision of Universal Library for Windows Languages. It may be supported in the future. Contact us at 508 -946-5100, and follow the instructions for reaching Tech. Support.

96	**NOTSIMULCONF**	Board not configured for `SIMULTANEOUS` option

The configuration file of the D/A board in *Insta*Cal must be set for simultaneous update before you use the `SIMULTANEOUS` option of `cbAOutScan()/AOutScan()`. The jumpers on the D/A board must be set for simultaneous update before it will work.

Error number	Error name	Error message
97	**EVENODDMISMATCH**	An even channel is in an odd slot in the queue, or vice versa

The channel gain queue on some A/D boards has a restriction that the channel numbers must be in even queue positions and odd channel numbers must be in odd queue positions.

98	**M1RATEWARNING**	Sampling speed to system memory MAY be too fast

The A/D board sampling speed you have requested may be too fast for the computer system bus transfer to complete before the next packet is ready for transfer. If this is the case, data will overrun and sample data will be garbled. This warning is initiated whenever you request a sample rate over 625 kHz AND the sample set is larger than the FIFO buffer on the board AND an external memory board, such as a MEGA-FIFO is not being used. Your system may be able to handle the rate requested but only experimentation will bear this out. Your system may be capable of the full 1 MHz rate directly to system memory.

99	**NOTRS485**	Selected board is not a RS-485 board

An attempt was made to call `cbRS485()/RS485()` with a board that is not RS485 compatible.

100	**NOTDOSFUNC**	This function not available in DOS

The function that was called is not available in the DOS version of the Universal Library.

101	**RANGEMISMATCH**	Bipolar and unipolar ranges cannot be used together in A/D queue

The channel/gain queue should only be loaded (via `cbALoadQueue()/ALoadQueue()`) with all unipolar or bipolar ranges.

102	**CLOCKTOOSLOW**	Sampling rate is too high for clock speed; change clock jumper on board

The sampling rate that you requested is too fast. The A/D board pacer might be capable of running at a higher rate. Check the board for an XTAL jumper and, if it is not set for the highest rate, place the jumper in the position for the highest rate. After the jumper is set, re-run *Insta*Cal.

103	**BADCALFACTORS**	Calibration factors are invalid, disabling software calibration

The selected board uses software calibration and the stored calibration factors are invalid. Run *Insta*Cal and calibrate the board before using it.

104	**BADCONFIGTYPE**	Invalid configuration information type specified

An invalid `ConfigType` argument was passed to either `cbGetConfig()` or `cbSetConfig()`.

105	**BADCONFIGITEM**	Invalid configuration item specified

An invalid `ConfigItem` argument was passed to either `cbGetConfig()` or `cbSetConfig()`.

106	**NOPCMCIABOARD**	Cannot access the PCMCIA board

Cannot access the specified PCMCIA board. Make sure that the PCMCIA Card & Socket Services are installed correctly and that the board was installed in the system correctly via *Insta*Cal.

107	**NOBACKGROUND**	Board does not support background operation

The `BACKGROUND` option was used and the specified board does not support background operation.

Error number	Error name	Error message
108	**STRINGTOOSHORT**	The string argument is too short for the string being returned

The string passed to a library function or method is to small to contain the string that is being returned. Increase the size of the string to the minimum size specified for the function or method that you are using.

Error number	Error name	Error message
109	**CONVERTEXTMEM**	`CONVERTDATA` not allowed with `EXTMEMORY` option

You requested both the `CONVERTDATA` and `EXTMEMORY` option. These options cannot be used together. Collect the data without the `CONVERTDATA` option. After the data has been collected, read it back from the memory card (`cbMemRead()`/`MemRead()` or `cbMemReadPretrig()`/ `MemReadPretrig()`), and use `cbAConvertData()`/`AConvertData()`) to convert the data.

Error number	Error name	Error message
110	**BADEUADD**	Program error – bad values used in `cbFromEngUnits` or `cbToEngUnits()`

Invalid floating point data was used in `cbFromEngUnits()`/`FromEngUnits()` or `cbToEngUnits`/`ToEngUnits()`. Check the arguments passed to the relevant function or method .

Error number	Error name	Error message
111	**DAS16JRRATEWARNING**	Rates greater than 125 kHz must use on board 10 MHz clock

If a rate greater than 125 kHz is selected and the on board jumper is set for 1 MHz when using the CIO-DAS16/JR, this warning is generated. Place the jumper on the 10 MHz position and update your *Insta*Cal settings.

Error number	Error name	Error message
112	**DAS08TOOLOW_RATE**	The desired sample rate is below hardware minimum

Increase the value of the `Rate` argument in `cbAInScan()`/`AInScan()`. The lowest pacer frequency is the clock frequency (usually 8 MHz / 2) divided by 65535 for the CIO-, PC104 and PCM-DAS08.

Error number	Error name	Error message
114	**AMBIGSENSORONGP**	More than one temperature sensor type defined for EXP-GP

Thermocouple and RTD types are both defined for an EXP-GP. `cbTIn()`/(`TIn()` and `cbTInScan()`/`TInScan()`) require that only one be defined to operate. Set one of the sensor types to "Not Installed" within the appropriate *Insta*Cal menu.

Error number	Error name	Error message
115	**NOSENSORTYPEONGP**	No temperature sensor type defined for EXP-GP

Neither Thermocouple nor RTD types are defined for an EXP-GP. `cbTIn()`/(`TIn()` and `cbTInScan()`/`TInScan()`) require that one and only one be defined to operate. Set one of the sensor types to a predefined type within the appropriate *Insta*Cal menu.

Error number	Error name	Error message
116	**NOCONVERSIONNEEDED**	Selected 12 bit board already returns converted data

Some 12-bit boards do not need to have their data converted after a call to `cbAInScan()`/`AInScan()` with the `NOCONVERTDATA` option. These boards return no channel tags and therefore return data in its proper format. Calling `cbAConvertData()`/`AConvertData()` with data generated from these boards will generate this warning.

Error number	Error name	Error message
117	**NOEXTCONTINUOUS**	`CONTINUOUS` mode cannot be used with `EXTMEMORY`

`CONTINUOUS` mode is ignored when used with the `EXTMEMORY` option.

Error number	Error name	Error message
118	**INVALIDPRETRIGCONVERT**	`cbAConvertPretrigData` called after `cbAPretrig` failed

The data you are attempting to convert with `cbAConvertPretrigData()`/ `AConvertPretrigData()` can not be converted because `cbAPretrig()`/`APretrig()` did not return a complete data set, probably due to an early trigger.

Error number	Error name	Error message
119	**BADCTRREG**	Bad counter argument passed to cbCLoad()
	The RegNum argument passed to `cbCLoad()` (`CLoad()`) is not a valid register.	
120	**BADTRIGTHRESHOLD**	Low trigger threshold is greater than high threshold
	The LowThreshold arguments to `cbSetTrigger()/SetTrigger()` must be less than the HighThreshold.	
121	**BADPCMSLOTREF**	NO PCM Card was found in the specified slot
	This is usually caused by swapping PCMCIA cards and not re-running *Insta*Cal. Run *Insta*Cal.	
122	**AMBIGPCMSLOTREF**	Two identical PCM cards found. Please specify exact slot in InstaCal
	This error occurs in DOS mode only when *Insta*Cal is configured for a PCMCIA card in "any slot". To correct the problem, run *Insta*Cal. Go to the Install menu and pop up the board's menu. Highlight PCMCIA slot and choose either "0" or "1".	
123	**BADSENSORTYPE**	Invalid sensor type selected in *Insta*Cal
	The specified sensor type is not part of the allowed list of thermocouple/RTD types. Set the sensor type to a predefined type within the appropriate *Insta*Cal menu.	
126	**CFGFILENOTFOUND**	Cannot find CB.CFG file
	The CB.CFG file could not be found. This file should be located in the same directory that you installed the software in.	
127	**NOVDDINSTALLED**	The CBUL.386 virtual device driver is not installed
	The Windows device driver CBUL.386 is not installed on your system. Normally, it will be automatically installed when you run the standard installation program. The following line should be in your \windows\system.ini file in the [386Enh] section: `device=c:\cb\cbul.386`	
128	**NOWINDOWSMEMORY**	Requested amount of Windows page-locked memory is not available
	The Windows device driver could not allocate the required amount of physical memory. This error should not normally occur unless you are collecting very large amounts of data or your system is very memory constrained. If you are collecting a very large block of memory, try collecting a smaller amount. If this is not an option, than consider using `cbFileAInScan()/FileAInScan()` instead of `cbAInScan()/AInScan()`. Also, if you are running other programs, try shutting them down.	
129	**OUTOFDOSMEMORY**	Not enough DOS memory available
	Try closing down any unneeded programs that are running.	
130	**OBSOLETEOPTION**	Obsolete option specified for `cbSetConfig/cbGetConfig`
	The specified configuration item is no longer supported in the 32-bit version of the Universal Library.	
131	**NOPCMREGKEY**	No registry entry for this PCMCIA card
	When running under Windows/NT, there must be an entry in the system registry for each PCMCIA card that you will be using with the system. This is ordinarily taken care of automatically by the Universal Library installation program. If this error occurs, contact the technical support dept for assistance at 508-946-5100.	

Error number	Error name	Error message
132	**NOCBUL32SYS**	CBUL32.SYS device driver is not installed
	The Windows device driver CBUL.SYS is not installed on your system. Normally, it will be automatically installed when you run the MCC standard installation program. Contact the technical support dept for assistance at 508-946-5100.	
133	**NODMAMEMORY**	No DMA memory available to device driver
	The Windows device driver could not allocate the minimum required amount of memory for DMA. If you are sampling at slower speeds, you can specify `SINGLEIO` in the `Options` argument to `cbAInScan()`/(`AInScan()`. This will prevent the library from attempting to use DMA. In general though, this error should not ordinarily occur. Contact technical support at 508-946-5100 with the details.	
134	**IRQNOTAVAILABLE**	IRQ not available
	The Interrupt Level that was specified for the board (in *Insta*Cal) conflicts with another board in your computer. Try switching to a different interrupt level.	
135	**NOT7266CTR**	This board does not have an LS7266 counter
	This function or method can only be used with a board that contains an LS7266 chip. These chips are used on various quadrature encoder input boards.	
136	**BADQUADRATURE**	Invalid `Quadrature` argument passed to `cbC7266Config()`
	The `Quadrature` argument must be set to either `NO_QUAD, X1_QUAD, X2_QUAD, or X4_QUAD`.	
137	**BADCOUNTMODE**	Invalid `CountingMode` argument passed to `cbC7266Config()`
	The `CountingMode` argument must be set to either `NORMAL_MODE, RANGE_LIMIT, NO_RECYCLE,` or `MODULO_N`.	
138	**BADENCODING**	Invalid `DataEncoding` argument passed to `cbC7266Config()`
	The `DataEncoding` argument must be set to either `BCD_ENCODING` or `BINARY_ENCODING`.	
139	**BADINDEXMODE**	Invalid `IndexMode` argument passed to `cbC7266Config()`
	The `IndexMode` argument must be set to either `INDEX_DISABLED, LOAD_CTR, LOAD_OUT_LATCH`, or `RESET_CTR`.	
140	**BADINVERTINDEX**	Invalid `InvertIndex` argument passed to `cbC7266Config()`
	The `InvertIndex` argument must be set to either (`CB`)`ENABLED` or (`CB`)`DISABLED`.	
141	**BADFLAGPINS**	Invalid `FlagPins` argument passed to `cbC7266Config()`
	The FlagPins argument must be set to either `CARRY_BORROW`, `COMPARE_BORROW`, `CARRYBORROW_UPDOWN,` or `INDEX_ERROR`.	
142	**NOCTRSTATUS**	This board does not support `cbCStatus()`
	This board does not return any status information.	
143	**NOGATEALLOWED**	Gating can not be used when indexing is enabled
	Gating and indexing can not be used simultaneously. If `Gating` is set to (`CB`)`ENABLED`, then `IndexMode` must be set to `INDEX_DISABLED`.	
144	**NOINDEXALLOWED**	Indexing not allowed in non-quadrature mode
	Indexing is not supported when `Quadrature` argument is set to `NO_QUAD`.	

Error number	Error name	Error message
145	**OPENCONNECTION**	Temperature input has open connection
146	**BMCONTINUOUSCOUNT**	Count must be integer multiple of packet size for Continuous mode
147	**BADCALLBACKFUNC**	Invalid pointer to callback function or delegate passed as argument
148	**MBUSINUSE**	Metrabus in use
149	**MBUSNOCTLR**	Metrabus I/O card has no configured controller card
150	**BADEVENTTYPE**	Invalid `EventType` specified for this board
	Although this board does support `cbEnableEvent()`/`EnableEvent()`, it does not support one or more of the event types specified.	
151	**ALREADYENABLED**	Event handler already enabled for this event type
	There is already an event handler bound to one or more of the events specified. To attach the new handler to the event type, first disable and disconnect the current handler using `cbDisableEvent()/DisableEvent()`.	
152	**BADEVENTSIZE**	Invalid event count has been specified
	The `ON_DATA_AVAILABLE` event requires an event count greater than (0).	
153	**CANTINSTALLEVENT**	Unable to install event handler
	An internal error occurred while trying to setup the event handling.	
154	**BADBUFFERSIZE**	Buffer is too small for operation
	The memory allocated by `cbWinBufAlloc()/WinBufAlloc()` is too small to hold all the data specified in the operation.	
155	**BADAIMODE**	Invalid analog input mode
	Invalid analog input mode (`RSE`, `NRSE`, `DIFF`).	
156	**BADSIGNAL**	Invalid signal type specified
	The specified signal type does not exist, or is not valid for signal direction specified.	
157	**BADCONNECTION**	Invalid connection
	The specified connection does not exist, or is not valid for the signal type and direction specified.	
158	**BADINDEX**	Invalid index specified
	For `Index` > 0, indicates that the specified index is beyond the end of the internal list of output connections assigned to the specified signal type.	
159	**NOCONNECTION**	Invalid connection
	No connection is assigned to the specified signal.	
160	**BADBURSTIOCOUNT**	Count cannot be greater than the FIFO size for BURSTIO mode. Also, Count must be integer multiple of number of channels in the scan.
	When using `BURSTIO` mode, the count entered cannot be larger than the FIFO size.	
161	**DEADEV**	Device has stopped responding. Please check connections.
	Check cable connections to USB device and to your computer's USB port.	

Error number	Error name	Error message
163	**INVALIDACCESS**	Required access or privilege not acquired for specified operation. Please check for other users of device and restart application.

You are currently not the device owner and therefore cannot change the state or configuration of the Ethernet device with functions such as `cbAOut()/AOut()`, `cbDBitOut/DBitOut()`, `cbAInScan()/AInScan()`, `cbFlashLED()/FlashLED()`, and others. However, you can still read the state or configuration of the Ethernet device with functions such as `cbAIn()/AIn()`, `cbDBitIn()/DBitIn()`, and so on.

164	**UNAVAILABLE**	Device unavailable at time of request. Please repeat operation.

You requested an operation that conflicts with an operation in progress on the device. This error usually occurs in multithreaded applications or if you are running multiple applications that access the device. Both types of operations are not supported.

165	**NOTREADY**	Device is not ready to send data. Please repeat operation.

You requested an operation that conflicts with an operation in progress on the device. This error can occur during device initialization.

200-299	Internal 16-bit error	Internal error occurred in library: See details below
201	**CANT_LOCK_DMA_BUF**	DMA buffer could not be locked

There is not enough physical memory to lock down enough DMA memory for this operation. Try closing out other applications, or installing additional RAM.

202	**DMA_IN_USE**	DMA already controlled by another driver

The DMA controller is currently being used by another device, such as another DMA board or the floppy drive.

203	**BAD_MEM_HANDLE**	Invalid Windows memory handle

The memory handle supplied is invalid. Memory handles supplied to library functions and methods should be allocated using `cbWinBufAlloc()/WinBufAlloc()`, and should not be de-allocated until `BACKGROUND` operations using this buffer are complete or cancelled with `cbStopBackground()/StopBackground()`.

300-399	Internal 32-bit error	Error in 32-bit Windows library. See details below
304	**CFG_FILE_READ_FAILURE**	Error reading from configuration file

The program was unable to read configuration file cb.cfg. Confirm that cb.cfg was not deleted, moved, or renamed since the software installation.

305	**CFG_FILE_WRITE_FAILURE**	Error writing to configuration file

The program was unable to write to the configuration file cb.cfg. Confirm that cb.cfg is present and that its attributes are not set for Read-only. Also, check that not more than one application is trying to access this file.

308	**CFGFILE_CANT_OPEN**	Cannot open configuration file

The program was unable to open the configuration file `cb.cfg`. Confirm that `cb.cfg` was not deleted, moved, or renamed since the software installation.

325	**BAD_RTD_CONVERSION**	Overflow of RTD conversion

Either `cbTIn()/Tin()` or `cbTInScan()/ TInScan()` returned an invalid temperature conversion. Confirm that the configuration matches the RTD type, and physical EXP board settings; pay particular attention to gain settings and RTD base resistance. Also, check that the RTD leads are securely attached to the EXP terminals. Finally, confirm that the board is measuring reasonable voltages via `cbAIn()/AIn()`.

Error number	Error name	Error message
326	**`NO_PCI_BIOS`**	PCI BIOS not present on the PC
	Could not locate the BIOS for the PCI bus. Consult PC supplier for proper installation of the PCI BIOS.	
327	**`BAD_PCI_INDEX`**	Specified PCI board not detected
	The specified PCI board was not detected. Check that PCI board in securely installed into PCI slot. Also, run *Insta*Cal to locate/set valid base address and configuration.	
328	**`NO_PCI_BOARD`**	Specified PCI board not detected
	The specified PCI board was not detected. Check that PCI board in securely installed into PCI slot. Also, run *Insta*Cal to locate/set valid base address and configuration.	
334	**`CANT_INSTALL_INT`**	Cannot install interrupt handler. IRQ already in use
	The device driver could not enable requested interrupt. Check that the selected IRQ is not already in use by another device. This error can also occur if a `FOREGROUND` scan was aborted; in such cases, rebooting the PC will correct the problem.	
339	**`CANT_MAP_PCM_CIS`**	Unable to access Card Information Structure
	A resource conflict between the specified PCMCIA or PC-Card device and another device prevents the system from allocating sufficient resources to map the onboard CIS.	
400-499	PCMCIA error	Card & Socket Service error. Contact the manufacturer
500-599	Internal DOS error	Contact the manufacturer
600-699	Internal Windows error	See details below
603	**`WIN_CANNOT_ENABLE_INT`**	Cannot enable interrupt. IRQ already in use
	The device driver could not enable requested interrupt. Check that the selected IRQ is not already in use by another device. This error can also occur if a `FOREGROUND` scan was aborted; in such cases, rebooting the PC will correct the problem.	
605	**`WIN_CANNOT_DISABLE_INT`**	Cannot disable interrupts
	The device driver was unable to disable the IRQ. This can occur when interrupts are generated too fast for the PC to complete servicing. For example, sampling at high frequencies (above ~2 kHz) with scan mode set for `SINGLEIO` can lead to this error. Frequently, an `OVERRUN` error accompanies this condition.	
606	**`WIN_CANT_PAGE_LOCK_BUFFER`**	Insufficient memory to page lock data buffer
	There is not enough physical memory to lock down the entire data buffer. Try closing out other applications, selecting smaller data buffers, or installing additional RAM.	
630	**`NO_PCM_CARD`**	PCM card not detected
	The specified PCMCIA card was not detected. Confirm that the PCM card is securely plugged into PCMCIA slot. If the board continues to return this error, run *Insta*Cal to reset the configuration.	

Measurement Computing Corporation
16 Commerce Boulevard,
Middleboro, Massachusetts 02346
(508) 946-5100
Fax: (508) 946-9500
E-mail: info@mccdaq.com
www.mccdaq.com