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 mission:

- To provide PC-based data acquisition hardware and software that will save time and save money.

Simple installations minimize the time between setting up your system and actually making measurements. We offer quick and simple access to outstanding live FREE technical support to help integrate MCC products into a DAQ system.

**Limited Lifetime Warranty:** Most MCC products are covered by a limited lifetime warranty against defects in materials or workmanship for the life of the product, to the original purchaser, unless otherwise noted. Any products found to be defective in material or workmanship will be repaired, replaced with same or similar device, or refunded at MCC’s discretion. For specific information, please refer to the terms and conditions of sale.

**Harsh Environment Warranty® Program:** Any Measurement Computing product that is damaged due to misuse, or any reason, may be eligible for replacement with the same or similar device for 50% of the current list price. I/O boards face some harsh environments, some harsher than the boards are designed to withstand. Contact MCC to determine your product’s eligibility for this program.

**30 Day Money-Back Guarantee:** Any Measurement Computing Corporation product may be returned 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.

*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 Corporation, 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 Corporation 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**


Windows, Microsoft, and Visual Studio are either trademarks or registered trademarks of Microsoft Corporation.

LabVIEW is a trademark of National Instruments.

CompactFlash is a registered trademark of SanDisk Corporation.

XBees and XBee-PRO are trademarks of MaxStream, Inc.

All other trademarks are the property of their respective owners.

Information furnished by Measurement Computing Corporation is believed to be accurate and reliable. However, no responsibility is assumed by Measurement Computing Corporation 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 Corporation.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form by any means, electronic, mechanical, by photocopying, recording, or otherwise without the prior written permission of Measurement Computing Corporation.

**Notice**

Measurement Computing Corporation does not authorize any Measurement Computing Corporation product for use in life support systems and/or devices without prior written consent from 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 Corporation 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 (32-bit)

1 Overview – Universal Library .......................................................... 13
   Introduction .................................................................................. 13
   Analog I/O functions .................................................................... 13
   Configuration functions ............................................................... 14
   Counter functions ........................................................................ 14
   Data Logger functions .................................................................... 15
   Digital I/O functions ..................................................................... 16
   Error handling functions ............................................................... 16
   Memory board functions ............................................................... 16
   Revision control functions ......................................................... 16
   Streamer file functions .................................................................. 17
   Synchronous I/O functions ......................................................... 17
   Temperature input functions ....................................................... 17
   Windows memory management functions .................................. 17
   Miscellaneous functions .............................................................. 18
   Universal Library example programs ............................................. 18

2 Analog I/O Functions ................................................................. 28
   Introduction .................................................................................. 28
   cbAConvertData() ........................................................................ 29
   cbAConvertPretrigData() ............................................................. 31
   cbACalibrateData() ...................................................................... 33
   cbAIn() ...................................................................................... 34
   cbAInScan() ................................................................................ 35
   cbALoadQueue() ........................................................................... 40
   cbAOut() ..................................................................................... 41
   cbAOutScan() .............................................................................. 42
   cbAPretrig() ............................................................................... 44
   cbATrig() ..................................................................................... 47
   cbVIn() ...................................................................................... 48
   cbVOut() ..................................................................................... 49

3 Configuration Functions ............................................................... 50
   Introduction .................................................................................. 50
   cbGetConfig() ............................................................................ 51
   cbGetConfigString() ................................................................... 55
   cbGetSignal() .............................................................................. 56
   cbSelectSignal() .......................................................................... 57
   cbSetConfig() ............................................................................. 60
   cbSetConfigString() ................................................................. 63
   cbSetTrigger() ............................................................................ 64

4 Counter Functions ........................................................................ 67
   Introduction .................................................................................. 67
   cbC7266Config() .......................................................................... 68
   cbC8254Config() .......................................................................... 70
   cbC8536Config() .......................................................................... 71
   cbC9513Config() .......................................................................... 72
   cbC8536Init() ............................................................................... 74
   cbC9513Init() ............................................................................... 75
   cbCClear() ................................................................................... 77
   cbCConfigScan() .......................................................................... 78
   cbCFreqIn() ................................................................................ 81
   cbCIn() ....................................................................................... 83
   cbCIn32() .................................................................................... 84
5 Data Logger Functions .................................................................94
Introduction ..................................................................................94
cbLogConvertFile() ........................................................................95
cbLogGetAIChannelCount() ..............................................................96
cbLogGetAllInfo() ..........................................................................97
cbLogGetCJCInfo() ..........................................................................98
cbLogGetDIOMainInfo() .................................................................99
cbLogGetFileInfo() .......................................................................100
cbLogGetFileName() .....................................................................101
cbLogGetPreferences() ................................................................102
cbLogGetSampleInfo() .................................................................103
cbLogReadAIChannels() .................................................................104
cbLogReadCJCChannels() ...............................................................106
cbLogReadDIChannels() .................................................................108
cbLogReadTimeTags() ..................................................................109
cbLogSetPreferences() ..................................................................111

6 Digital I/O Functions ....................................................................112
Introduction ......................................................................................112
cbDIIn() .........................................................................................113
cbDIOut() .......................................................................................114
cbDConfigBit() ................................................................................115
cbDConfigPort() .............................................................................116
cbDIn() ..........................................................................................117
cbDInScan() ....................................................................................118
cbDOut() ........................................................................................120
cbDOutScan() ................................................................................121

7 Error Handling Functions .............................................................123
Introduction ......................................................................................123
cbErrHandling() .............................................................................124
cbErrMsg() .......................................................................................126

8 Memory Board Functions ................................................................127
Introduction ......................................................................................127
cbMemRead() ....................................................................................128
cbMemReadPretrig() .........................................................................129
cbMemReset() ...................................................................................130
cbMemSetDTMode() ..........................................................................131
cbMemWrite() ...................................................................................132

9 Revision Control Functions .........................................................133
Introduction ......................................................................................133
cbDeclareRevision() ........................................................................134
cbGetRevision() ................................................................................135

10 Streamer File Functions ..............................................................136
Introduction ......................................................................................136
cbFileAInScan() ...............................................................................137
cbFileGetInfo() ..............................................................................139
cbFilePretrig() ................................................................................140
cbFileRead() ....................................................................................142
# 11 Synchronous I/O Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbDaqInScan()</td>
<td>143</td>
</tr>
<tr>
<td>cbDaqOutScan()</td>
<td>144</td>
</tr>
<tr>
<td>cbDaqSetSetpoints()</td>
<td>147</td>
</tr>
<tr>
<td>cbDaqSetTrigger()</td>
<td>149</td>
</tr>
<tr>
<td>cbWinBufToOutByte()</td>
<td>151</td>
</tr>
</tbody>
</table>

# 12 Temperature Input Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbTin()</td>
<td>155</td>
</tr>
<tr>
<td>cbInScan()</td>
<td>156</td>
</tr>
</tbody>
</table>

# 13 Windows Memory Management Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbWinBufAlloc()</td>
<td>161</td>
</tr>
<tr>
<td>cbWinBufAlloc32()</td>
<td>163</td>
</tr>
<tr>
<td>cbWinBufFree()</td>
<td>164</td>
</tr>
<tr>
<td>cbWinArrayToBuf()</td>
<td>165</td>
</tr>
<tr>
<td>cbWinBufToArray()</td>
<td>166</td>
</tr>
<tr>
<td>cbWinBufToArray32()</td>
<td>167</td>
</tr>
</tbody>
</table>

# 14 Miscellaneous Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbDeviceLogin()</td>
<td>168</td>
</tr>
<tr>
<td>cbDeviceLogout()</td>
<td>169</td>
</tr>
<tr>
<td>cbDisableEvent()</td>
<td>170</td>
</tr>
<tr>
<td>cbEnableEvent()</td>
<td>171</td>
</tr>
<tr>
<td>cbFlashLED()</td>
<td>173</td>
</tr>
<tr>
<td>cbFromEngUnits()</td>
<td>174</td>
</tr>
<tr>
<td>cbGetBoardName()</td>
<td>175</td>
</tr>
<tr>
<td>cbGetStatus()</td>
<td>177</td>
</tr>
<tr>
<td>cbGetTCValues()</td>
<td>179</td>
</tr>
<tr>
<td>cbInByte()</td>
<td>180</td>
</tr>
<tr>
<td>cbInWord()</td>
<td>181</td>
</tr>
<tr>
<td>cbOutByte()</td>
<td>182</td>
</tr>
<tr>
<td>cbOutWord()</td>
<td>183</td>
</tr>
<tr>
<td>cbRS485()</td>
<td>184</td>
</tr>
<tr>
<td>cbStopBackground()</td>
<td>185</td>
</tr>
<tr>
<td>cbToEngUnits()</td>
<td>186</td>
</tr>
</tbody>
</table>

# Universal Library for .NET Classes, Methods, and Properties

## 15 UL for .NET Class Library Overview

<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MccDaq namespace</td>
<td>188</td>
</tr>
<tr>
<td>MccDaq classes</td>
<td>188</td>
</tr>
<tr>
<td>MccBoard class</td>
<td>188</td>
</tr>
<tr>
<td>ErrorInfo class</td>
<td>189</td>
</tr>
<tr>
<td>MccService class</td>
<td>190</td>
</tr>
<tr>
<td>GlobalConfig class</td>
<td>190</td>
</tr>
<tr>
<td>DataLogger class</td>
<td>190</td>
</tr>
<tr>
<td>Analog I/O methods</td>
<td>190</td>
</tr>
<tr>
<td>Configuration methods and properties</td>
<td>192</td>
</tr>
<tr>
<td>Counter methods</td>
<td>194</td>
</tr>
<tr>
<td>Data Logger methods and properties</td>
<td>195</td>
</tr>
<tr>
<td>Digital I/O methods</td>
<td>196</td>
</tr>
<tr>
<td>Error Handling method and properties</td>
<td>196</td>
</tr>
<tr>
<td>Memory board methods</td>
<td>196</td>
</tr>
<tr>
<td>Revision control methods and properties</td>
<td>197</td>
</tr>
<tr>
<td>Streamer file methods</td>
<td>197</td>
</tr>
<tr>
<td>Synchronous I/O methods</td>
<td>197</td>
</tr>
</tbody>
</table>
Temperature input methods .......................................................... 197
Windows memory management methods ........................................ 198
Miscellaneous methods, properties, and delegates ............................ 198
Universal Library for .NET example programs .................................. 199

16 Analog I/O Methods .................................................................. 208

Introduction .................................................................................. 208
AConvertData() ............................................................................. 209
AConvertPrerigData() ..................................................................... 210
ACalibrateData() ........................................................................... 212
An() ......................................................................................... 213
AnScan() .................................................................................... 214
ALoadQueue() ............................................................................... 219
AOut() ....................................................................................... 220
AOutScan() .................................................................................. 221
APretrig() .................................................................................... 223
ATrig() ....................................................................................... 226
Vin() ......................................................................................... 227
VOut() ....................................................................................... 228

17 Configuration Methods and Properties ...................................... 229

Introduction .................................................................................. 229
BoardConfig property ................................................................. 230
BoardConfig.DACUpdate() ............................................................ 230
BoardConfig.GetAdRetrigCount() .................................................. 230
BoardConfig.GetBaseAdr() ............................................................ 231
BoardConfig.GetBoardType() ......................................................... 231
BoardConfig.GetDINumDev() .......................................................... 231
BoardConfig.GetClock() ............................................................... 232
BoardConfig.GetDACStartup() ....................................................... 232
BoardConfig.GetDACUpdateMode() ............................................... 232
BoardConfig.GetDeviceID() ........................................................... 233
BoardConfig.GetDeviceNotes() ...................................................... 233
BoardConfig.GetDINumDev() .......................................................... 233
BoardConfig.GetDmaChan() ........................................................... 234
BoardConfig.GetDIBoard() ............................................................. 234
BoardConfig.GetIntLevel() ............................................................ 234
BoardConfig.GetIntAdChans() ....................................................... 235
BoardConfig.GetIntDaChans() ....................................................... 235
BoardConfig.GetIntExps() ............................................................. 235
BoardConfig.GetIntInPorts() ......................................................... 236
BoardConfig.GetPANID() ............................................................... 236
BoardConfig.GetRange() ............................................................... 236
BoardConfig.GetRFCannel() ........................................................... 237
BoardConfig.GetRSS() ................................................................. 237
BoardConfig.GetUsesExps() ........................................................... 237
BoardConfig.GetWaitState() .......................................................... 238
BoardConfig.SetAdRetrigCount() .................................................... 238
BoardConfig.SetBaseAdr() ............................................................. 238
BoardConfig.SetClock() ............................................................... 239
BoardConfig.SetDmaChan() ........................................................... 239
BoardConfig.SetDACStartup() ....................................................... 239
BoardConfig.SetDACUpdateMode() ............................................... 240
BoardConfig.SetDeviceID() ........................................................... 241
BoardConfig.SetDeviceNotes() ....................................................... 241
BoardConfig.SetIntLevel() ............................................................ 241
BoardConfig.SetIntAdChans() ....................................................... 242
BoardConfig.SetPANID() ............................................................. 242
BoardConfig.SetRange() ............................................................... 242
BoardConfig.SetRFCannel() ........................................................... 243
BoardConfig.SetWaitState() .......................................................... 243
BoardNum property ..................................................................... 244
CtrConfig property ................................................................... 245
# Universal Library Function Reference

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CtrConfig.GetCtrType()</td>
<td>245</td>
</tr>
<tr>
<td>DioConfig property</td>
<td>246</td>
</tr>
<tr>
<td>DioConfig.GetDInMask()</td>
<td>246</td>
</tr>
<tr>
<td>DioConfig.GetDOutMask()</td>
<td>247</td>
</tr>
<tr>
<td>DioConfig.GetConfig()</td>
<td>247</td>
</tr>
<tr>
<td>DioConfig.GetCurVal()</td>
<td>247</td>
</tr>
<tr>
<td>DioConfig.GetDevType()</td>
<td>248</td>
</tr>
<tr>
<td>DioConfig.GetDevType()</td>
<td>248</td>
</tr>
<tr>
<td>DioConfig.GetNumBits()</td>
<td>248</td>
</tr>
<tr>
<td>ExpansionConfig property</td>
<td>249</td>
</tr>
<tr>
<td>ExpansionConfig.GetBoardType()</td>
<td>249</td>
</tr>
<tr>
<td>ExpansionConfig.GetCjCChan()</td>
<td>249</td>
</tr>
<tr>
<td>ExpansionConfig.GetMuxAdChan1()</td>
<td>250</td>
</tr>
<tr>
<td>ExpansionConfig.GetMuxAdChan2()</td>
<td>250</td>
</tr>
<tr>
<td>ExpansionConfig.GetNumExpChans()</td>
<td>250</td>
</tr>
<tr>
<td>ExpansionConfig.GetRange1()</td>
<td>251</td>
</tr>
<tr>
<td>ExpansionConfig.GetRange2()</td>
<td>251</td>
</tr>
<tr>
<td>ExpansionConfig.GetThermType()</td>
<td>251</td>
</tr>
<tr>
<td>ExpansionConfig.SetCjCChan()</td>
<td>253</td>
</tr>
<tr>
<td>ExpansionConfig.SetMuxAdChan1()</td>
<td>253</td>
</tr>
<tr>
<td>ExpansionConfig.SetMuxAdChan2()</td>
<td>254</td>
</tr>
<tr>
<td>ExpansionConfig.SetRange1()</td>
<td>254</td>
</tr>
<tr>
<td>ExpansionConfig.SetRange2()</td>
<td>254</td>
</tr>
<tr>
<td>ExpansionConfig.SetThermType()</td>
<td>255</td>
</tr>
<tr>
<td>GetSignal()</td>
<td>256</td>
</tr>
<tr>
<td>NumBoards property</td>
<td>258</td>
</tr>
<tr>
<td>NumExpBoards property</td>
<td>258</td>
</tr>
<tr>
<td>SelectSignal()</td>
<td>259</td>
</tr>
<tr>
<td>SetTrigger()</td>
<td>263</td>
</tr>
<tr>
<td>Version property</td>
<td>266</td>
</tr>
</tbody>
</table>

## 18 Counter Methods .................................................. 267

- Introduction.................................................................. 267
- C7266Config().................................................. 268
- C8254Config().................................................. 270
- C8536Config().................................................. 271
- C8536Init().................................................. 273
- C9513Config().................................................. 274
- C9513Init().................................................. 277
- CClear()................................................................ 279
- CConfigScan().................................................. 280
- CFreqIn().................................................. 283
- CIn()................................................................ 285
- CIn32().................................................. 286
- CInScan().................................................. 287
- CLoad().................................................. 289
- CLoad32().................................................. 291
- CStatus().................................................. 292
- CStoreOnInit().............................................. 293
- TimerOutStart()............................................. 294
- TimerOutStop()............................................... 295

## 19 Data Logger Methods and Property .................................. 296

- Introduction................................................................ 296
- FileName property.................................................. 297
- ConvertFile().................................................. 298
- GetAChannelCount()............................................... 299
- GetAllInfo().................................................. 300
- GetCJCInfo().................................................. 301
- GetDIOInfo().................................................. 302
- GetFileName().................................................. 303
- GetPreferences().................................................. 304
- GetSampleInfo().................................................. 305
- GetSampleInfo().................................................. 306
Universal Library Function Reference

20 Digital I/O Methods .............................................................. 315
   Introduction .................................................................. 315
   DBitIn() ................................................................ 316
   DBitOut() ................................................................ 317
   DConfigBit() ............................................................ 318
   DConfigPort() ........................................................... 319
   DIn() .................................................................. 320
   DInScan() ................................................................ 321
   DOut() .................................................................. 323
   DOutScan() ................................................................ 324

21 Error Handling Methods and Properties ................................. 326
   Introduction ................................................................ 326
   ErrHandling() ............................................................ 327
   Message property ......................................................... 329

22 Memory Board Methods .......................................................... 330
   MemRead() ................................................................ 331
   MemReadPretrig() ......................................................... 332
   MemReset() ................................................................ 333
   MemSetDTMode() ........................................................ 334
   MemWrite() ................................................................ 335

23 Revision Control Methods ....................................................... 336
   Introduction ................................................................ 336
   DeclareRevision() ........................................................ 337
   GetRevision() .............................................................. 338

24 Streamer File Methods ............................................................ 339
   Introduction ................................................................ 339
   FileAInScan() .............................................................. 340
   FileGetInfo() .............................................................. 342
   FilePretrig() ............................................................... 343
   FileRead() ................................................................ 345

25 Synchronous I/O Methods ......................................................... 347
   Introduction ................................................................ 347
   DaqInScan() ................................................................ 348
   DaqOutScan() ............................................................. 351
   DaqSetSetpoints() ........................................................ 353
   DaqSetTrigger() .......................................................... 355

26 Temperature Input Methods ...................................................... 359
   Introduction ................................................................ 359
   TIn() .................................................................... 360
   TInScan() ................................................................ 363

27 Windows Memory Management Methods ................................. 366
   Introduction ................................................................ 366
   WinBufAlloc() ............................................................. 367
   WinBufAlloc32() .......................................................... 368
   WinBufFree() .............................................................. 369
   WinArrayToBuff() ........................................................ 370
   WinBufToArrary() ........................................................ 371
   WinBufToArray32() ......................................................... 372
28 Miscellaneous Methods, Properties, and Delegates ..................374
Introduction ................................................................................................. 374
BoardName property .................................................................................. 375
DeviceLogin() ............................................................................................ 376
DeviceLogout() ........................................................................................... 376
HideLoginDialog() ....................................................................................... 377
DisableEvent() ............................................................................................. 378
EnableEvent() ............................................................................................... 379
EngArrayToWinBuf() ..................................................................................... 381
EventCallback delegate ............................................................................... 383
FlashLED() .................................................................................................... 384
FromEngUnits() ............................................................................................ 385
GetBoardName() ........................................................................................... 386
GetStatus() ................................................................................................... 387
GetTCValues() .............................................................................................. 389
InByte() ........................................................................................................ 391
InWord() ....................................................................................................... 392
OutByte() ...................................................................................................... 393
OutWord() ..................................................................................................... 394
RS485() .......................................................................................................... 395
StopBackground() ......................................................................................... 396
ToEngUnits() ............................................................................................... 397
WinBufToEngArray() ..................................................................................... 398

Appendix

29 Error Codes .............................................................................................401
Universal Library Functions
(32-bit)
Overview – Universal Library

Introduction

This section contains a complete, detailed explanation of all 32-bit 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.

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. 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 channel/gain pairs into A/D board's queue. These channel/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...
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.

  For devices with a hardware implementation of pretrigger, 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.

- **cbAVIn()** - Reads an A/D input channel, and returns a voltage value.
- **cbVOut()** – Sets the value of a D/A output.

### 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.
- **cbGetConfigString()** - Retrieves configuration or device information as a null-terminated string.
- **cbSetConfig()** - Sets the current value for a specified configuration option.
- **cbSetConfigString()** - Sets the configuration or device information as a null-terminated string.
- **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.
- **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.
Data Logger functions

The data logger functions read and convert binary files logged by MCC hardware equipped with a data logger capability.

- `cbLogConvertFile()` - Converts a binary log file to a comma-separated values (CSV) text file or another text file format that you specify.
- `cbLogGetAIChannelCount()` - Retrieves the total number of analog input channels logged in a binary file.
- `cbLogGetAllInfo()` - Retrieves the channel number and unit value of each analog input channel logged in a binary file.
- `cbLogGetCJCInfo()` - Retrieves the number of CJC temperature channels logged in a binary file.
- `cbLogGetDIOInfo()` - Retrieves the number of digital I/O channels logged in a binary file.
- `cbLogGetFileInfo()` - Retrieves the version level and byte size of a binary file.
- `cbLogGetFileName()` - Retrieves the name of the nth file in the directory containing binary log files.
- `cbLogGetPreferences()` – Retrieves API preference settings for time stamped data, analog data, and CJC temperature data. Returns the default values unless changed using `cbLogSetPreferences()`.
- `cbLogGetSampleInfo()` - Retrieves the sample interval, sample count, and the date and time of the first data point contained in a binary file.
- `cbLogReadAIChannels()` - Retrieves analog input data from a binary file, and stores the values in an array.
- `cbLogReadCJCChannels()` - Retrieves CJC temperature data from a binary file, and stores the values in an array.
- `cbLogReadDIOChannels()` - Retrieves digital I/O channel data from a binary file, and stores the values in an array.
- `cbLogReadTimeTags()` - Retrieves date and time values logged in a binary file. This function stores date values in the DateTags array, and time values in the TimeTags array.
cbLogSetPreferences() - Sets preferences for returned time stamped data, analog temperature data, and CJC temperature data.

**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 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.

### Synchronous I/O functions

The synchronous I/O functions synchronously read, set, or write data from analog channels, counter channels, thermocouple channels, and digital ports.

- `cbDaqInScan()` – Scans analog, digital, temperature, and counter inputs synchronously, and stores the values in an array.
- `cbDaqOutScan()` – Outputs values synchronously to analog output channels and digital output ports.
- `cbDaqSetSetpoints()` – Configures up to 16 detection setpoints associated with the input channels within a scan group.
- `cbDaqSetTrigger()` – Selects a trigger source and sets up its parameters. This method starts or stops a synchronous data acquisition operation using `cbDaqInScan()` with the EXTRIGGER option.

### 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 take care of allocating, freeing and copying to/from Windows global memory buffers.

- `cbWinBufAlloc()` - Allocates a Windows memory buffer.
- `cbWinBufAlloc32()` - Allocates a Windows global memory buffer for use with 32-bit scan functions, and returns a memory handle for the buffer.
- `cbWinBufFree()` - Frees a Windows buffer.
Overview – Universal Library

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.

- cbDeviceLogin() – Opens a device session with a shared device.
- cbDeviceLogout() – Releases the device session with a shared device.
- 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.
- cbDisableEvent() – Disables one or more events set up with cbEnableEvent() and disconnects their user-defined handlers.
- cbFlashLED() – Causes the LED on a USB device to flash.
- cbFromEngUnits() – Converts a single precision voltage (or current) value in engineering units to an integer D/A count value for output to a D/A.
- 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.
- cbGetTCValues() – Converts raw thermocouple data gathered with cbDaqInScan() to Celsius, Fahrenheit, or Kelvin.
- 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() – Stops 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 an integer A/D count value to an equivalent single precision voltage (or current) value.

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 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 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>
<thead>
<tr>
<th>Program Name</th>
<th>Featured UL Function Call</th>
<th>Notes</th>
<th>Other UL Function Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>CInScan01</td>
<td>cbCInScan()</td>
<td>Scans a range of counter input channels, and writes the data to an array. Board 0 must support counter scans.</td>
<td>cbWinBuffAlloc32() cbWinBufToArray32() cbWinBufFree()</td>
</tr>
<tr>
<td>CInScan02</td>
<td>cbCInScan() cbCConfigScan()</td>
<td>Scans a counter input channel in decrement mode, and writes the data to an array. Board 0 must support counter scans.</td>
<td>cbWinBuffAlloc32() cbWinBufToArray32() cbWinBufFree()</td>
</tr>
<tr>
<td>DaqInScan01</td>
<td>cbDaqInScan()</td>
<td>Synchronously scans analog input channels, digital input ports and counter input channels in the foreground. Board 0 must support synchronous input.</td>
<td>cbDConfigPort() cbCConfigScan()</td>
</tr>
<tr>
<td>DaqInScan02</td>
<td>cbDaqInScan()</td>
<td>Synchronously scans analog input channels, digital input ports, and counter input channels in the background. Board 0 must support synchronous input.</td>
<td>cbDConfigPort() cbCConfigScan() cbGetStatus() cbStopBackground()</td>
</tr>
<tr>
<td>DaqInScan03</td>
<td>cbDaqInScan() cbGetTCValues()</td>
<td>Synchronously scans analog input channels, digital input ports, and thermocouple input channels in the foreground. Board 0 must support synchronous input.</td>
<td>cbDConfigPort() cbCConfigScan()</td>
</tr>
<tr>
<td>DaqOutScan01</td>
<td>cbDaqOutScan()</td>
<td>Synchronously writes to an analog output channel and a digital output port in the background. Board 0 must support synchronous output.</td>
<td>cbDConfigPort()</td>
</tr>
<tr>
<td>DaqSetSetpoints01</td>
<td>cbDaqSetSetpoints()</td>
<td>Configures setpoints, adds the setpoint status to the scanlist, and performs asynchronous reads of the setpoint status. Board 0 must support cbDaqInScan().</td>
<td>cbDaqInScan() cbDConfigPort() cbGetStatus() cbStopBackground()</td>
</tr>
<tr>
<td>DaqSetTrigger01</td>
<td>cbDaqSetTrigger()</td>
<td>Configures start and stop triggers. These triggers are used to initiate and terminate A/D conversion using cbDaqInScan() with the EXTTRIGGER option selected.</td>
<td>cbGetStatus() cbStopBackground()</td>
</tr>
<tr>
<td>Program Name</td>
<td>Featured UL Function Call</td>
<td>Notes</td>
<td>Other UL Function Calls</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------</td>
<td>-------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>TimerOut01</td>
<td>cbTimerOutputStart()</td>
<td>Sends a frequency output to an output timer channel. Board 0 must have a timer output.</td>
<td></td>
</tr>
<tr>
<td>ULAI01</td>
<td>cbAIn()</td>
<td></td>
<td>cbToEngUnits()</td>
</tr>
<tr>
<td>ULAI02</td>
<td>cbAInScan()</td>
<td></td>
<td>FOREGROUND mode</td>
</tr>
<tr>
<td>ULAI03</td>
<td>cbAInScan()</td>
<td></td>
<td>BACKGROUND mode</td>
</tr>
<tr>
<td>ULAI04</td>
<td>cbAConvertData()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULAI05</td>
<td>cbAInScan()</td>
<td>with manual data conversion</td>
<td></td>
</tr>
<tr>
<td>ULAI06</td>
<td>cbAInScan()</td>
<td>CONTINUOUS, BACKGROUND mode</td>
<td></td>
</tr>
<tr>
<td>ULAI07</td>
<td>cbATrig()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULAI08</td>
<td>cbAPretrig()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULAI09</td>
<td>cbAConvertPretrigData()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULAI10</td>
<td>cbALoadQueue()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULAI11</td>
<td>cbToEngUnits()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULAI12</td>
<td>cbAInScan()</td>
<td>EXT_CLOCK mode</td>
<td></td>
</tr>
<tr>
<td>ULAI13</td>
<td>cbAInScan()</td>
<td>Various sampling mode options</td>
<td></td>
</tr>
<tr>
<td>Program Name</td>
<td>Featured UL Function Call</td>
<td>Notes</td>
<td>Other UL Function Calls</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------</td>
<td>-------</td>
<td>------------------------</td>
</tr>
<tr>
<td>ULAI14</td>
<td>cbSetTrigger()</td>
<td></td>
<td>cbInScan()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbFromEngUnits()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufToArray()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufFree()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufAlloc()</td>
</tr>
<tr>
<td>ULAI001</td>
<td>cbAInScan() cbAOutScan()</td>
<td></td>
<td>cbGetStatus()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbStopBackground()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinArrayToBuf()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufToArray()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufFree()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufToAlloc()</td>
</tr>
<tr>
<td>ULAO01</td>
<td>cbAOut()</td>
<td></td>
<td>cbFromEngUnits()</td>
</tr>
<tr>
<td>ULAO02</td>
<td>cbAOutScan()</td>
<td></td>
<td>cbWinBufToArray()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufFree()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufAlloc()</td>
</tr>
<tr>
<td>ULAO03</td>
<td>cbAOut() cbSetConfig()</td>
<td></td>
<td>cbC9513Init()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbC9513Config()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbCLoad()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbCIn()</td>
</tr>
<tr>
<td>ULCT01</td>
<td>cbC8254Config()</td>
<td></td>
<td>cbCLoad()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbCIn()</td>
</tr>
<tr>
<td>ULCT02</td>
<td>cbC9513Init() cbC9513Config()</td>
<td></td>
<td>cbC9513Init()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbC9513Config()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbCLoad()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbCIn()</td>
</tr>
<tr>
<td>ULCT03</td>
<td>cbCStoreOnInt()</td>
<td></td>
<td>cbC9513Init()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbC9513Config()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbCLoad()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbCIn()</td>
</tr>
<tr>
<td>ULCT04</td>
<td>cbCFreqIn()</td>
<td></td>
<td>cbC9513Init()</td>
</tr>
<tr>
<td>ULCT05</td>
<td>cbC8536Init() cbC8536Config()</td>
<td></td>
<td>cbCLoad()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbCIn()</td>
</tr>
<tr>
<td>ULCT06</td>
<td>cbC7266Config()</td>
<td></td>
<td>cbCLoad32()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbCIn32()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbCStatus()</td>
</tr>
<tr>
<td>ULDI01</td>
<td>cbDIn()</td>
<td></td>
<td>cbDConfigPort()</td>
</tr>
<tr>
<td>ULDI02</td>
<td>cbDBitIn()</td>
<td></td>
<td>cbDConfigPort()</td>
</tr>
<tr>
<td>ULDI03</td>
<td>cbDInScan()</td>
<td></td>
<td>cbDConfigPort()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbGetStatus()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbStopBackground()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufToArray()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufFree()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbWinBufAlloc()</td>
</tr>
<tr>
<td>ULDI04</td>
<td>cbDIn()</td>
<td></td>
<td>using the AUXPORT</td>
</tr>
<tr>
<td>ULDI05</td>
<td>cbDBitIn()</td>
<td></td>
<td>using the AUXPORT</td>
</tr>
<tr>
<td>ULDI06</td>
<td>cbDConfigBit()</td>
<td></td>
<td>cbDBitIn()</td>
</tr>
<tr>
<td>ULD001</td>
<td>cbDOut()</td>
<td></td>
<td>cbDConfigPort()</td>
</tr>
<tr>
<td>ULD002</td>
<td>cbDBitOut()</td>
<td></td>
<td>cbDOut()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbDConfigPort()</td>
</tr>
<tr>
<td>Program Name</td>
<td>Featured UL Function Call</td>
<td>Notes</td>
<td>Other UL Function Calls</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------</td>
<td>-------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>ULDO04</td>
<td>cbDOut()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULDO05</td>
<td>cbDBitOut()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULEV01</td>
<td>cbEnableEvent()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULEV05</td>
<td>cbEnableEvent()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULEV02</td>
<td>cbEnableEvent()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULEV03</td>
<td>cbEnableEvent()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULEV04</td>
<td>cbEnableEvent()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULFI01</td>
<td>cbFileAInScan()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULFI02</td>
<td>cbFileRead()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULFI03</td>
<td>cbFilePretrig()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULGT01</td>
<td>cbGetErrMsg()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULGT03</td>
<td>cbGetConfig()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULGT04</td>
<td>cbGetBoardName()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULLLOG01</td>
<td>cbLogGetFileName()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULLLOG02</td>
<td>cbLogGetFileName()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULLLOG03</td>
<td>cbLogReadAIChannels()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Universal Library Example Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Featured UL Function Call</th>
<th>Notes</th>
<th>Other UL Function Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>ULOG04</td>
<td>cbLogConvertFile()</td>
<td>Converts a binary log file to a comma-separated values (.csv) text file or another text file format that you specify.</td>
<td>cbLogGetSampleInfo()</td>
</tr>
<tr>
<td>ULMBDI01</td>
<td>cbDIn()</td>
<td>Reads a digital input port on a MetraBus card</td>
<td></td>
</tr>
<tr>
<td>ULMBDI02</td>
<td>cbDBitIn()</td>
<td>Reads the status of a single digital input bit from a MetraBus card</td>
<td></td>
</tr>
<tr>
<td>ULMDO01</td>
<td>cbDOut()</td>
<td>Writes a byte to a digital output port on a MetraBus card</td>
<td></td>
</tr>
<tr>
<td>ULMDO02</td>
<td>cbDBitOut()</td>
<td>Sets the state of a single digital output bit for a MetraBus card</td>
<td></td>
</tr>
<tr>
<td>ULMM01</td>
<td>cbMemReadPretrig()</td>
<td></td>
<td>cbAPretrig()</td>
</tr>
<tr>
<td>ULMM02</td>
<td>cbMemRead()</td>
<td></td>
<td>cbAPretrig()</td>
</tr>
<tr>
<td></td>
<td>cbMemWrite()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULMM03</td>
<td>cbAInScan()</td>
<td>With the EXTMEMORY option</td>
<td>cbMemReset()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cbMemRead()</td>
</tr>
<tr>
<td>ULTI01</td>
<td>cbTIn()</td>
<td></td>
<td>cbGetConfig()</td>
</tr>
<tr>
<td>ULTI02</td>
<td>cbTInScan()</td>
<td></td>
<td>cbGetConfig()</td>
</tr>
<tr>
<td>VIn01</td>
<td>cbVin()</td>
<td>Reads an A/D input channel.</td>
<td></td>
</tr>
<tr>
<td>VOut01</td>
<td>cbVOut()</td>
<td>Writes to a D/A output channel.</td>
<td></td>
</tr>
</tbody>
</table>

*Sample programs ULEV01, ULEV02, ULEV03 and ULEV04 are not available for the C Console.

<table>
<thead>
<tr>
<th>UL Function Call</th>
<th>UL Example Program Name</th>
<th>Special Features / Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbAConvertData()</td>
<td>ULA104 ULA106</td>
<td></td>
</tr>
<tr>
<td>cbAConvertPretrigData()</td>
<td>ULA109 ULEV03</td>
<td></td>
</tr>
<tr>
<td>cbACalibrateData()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>cbAIn()</td>
<td>ULA101 ULA11</td>
<td></td>
</tr>
<tr>
<td>cbAInScan()</td>
<td>ULA102 ULA103 ULA112 ULA104 ULA113 ULA105 ULA14 ULA106 ULM03 ULEV02*</td>
<td>FOREGROUND, BACKGROUND mode with manual data conversion</td>
</tr>
<tr>
<td>cbALoadQueue()</td>
<td>ULA110</td>
<td></td>
</tr>
<tr>
<td>cbAOut()</td>
<td>ULA001 ULA003</td>
<td>ULA003 demonstrates the difference between BIDACUPDATEMODE settings of UPDATEIMMEDIATE and UPDATEONCOMMAND. Board 0 must support BIDACUPDATEMODE settings, such as the PCI-DAC6700 Series.</td>
</tr>
<tr>
<td>cbAOutScan()</td>
<td>ULA002 ULA001 ULEV04*</td>
<td></td>
</tr>
<tr>
<td>cbAPretrig()</td>
<td>ULA108 ULEV03 ULA109 ULF03</td>
<td></td>
</tr>
<tr>
<td>UL Function Call</td>
<td>UL Example Program Name</td>
<td>Special Features / Notes</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cbATrig()</td>
<td>ULA07 ULMM01</td>
<td></td>
</tr>
<tr>
<td>cbC7266Config()</td>
<td>ULCT06</td>
<td></td>
</tr>
<tr>
<td>cbC8254Config()</td>
<td>ULCT01</td>
<td></td>
</tr>
<tr>
<td>cbC8536Config()</td>
<td>ULCT05</td>
<td></td>
</tr>
<tr>
<td>cbC8536Init()</td>
<td>ULCT05</td>
<td></td>
</tr>
<tr>
<td>cbC9513Config()</td>
<td>ULCT02 ULCT03</td>
<td></td>
</tr>
<tr>
<td>cbC9513Init()</td>
<td>ULCT02 ULCT04 ULCT03</td>
<td></td>
</tr>
<tr>
<td>cbClear()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cbCConfigScan()</td>
<td>ClinScan02</td>
<td>Demonstrates how to scan a counter input channel in decrement mode, and then write the data to an array. Board 0 must support counter scans.</td>
</tr>
<tr>
<td>cbFreqIn()</td>
<td>ULCT04</td>
<td></td>
</tr>
<tr>
<td>cbCIn()</td>
<td>ULA01 ULA05 ULA02</td>
<td></td>
</tr>
<tr>
<td>cbCIn32()</td>
<td>ULCT06 ULCT07</td>
<td>For ULCT07, board 0 must have an event counter, such as the miniLAB 1008 or USB-1208LS.</td>
</tr>
<tr>
<td>cbCInScan()</td>
<td>ClinScan01 ClinScan02</td>
<td>Demonstrates how to scan one or more counter input channels and then write the data to an array. Board 0 must support counter scans.</td>
</tr>
<tr>
<td>cbCLoad()</td>
<td>ULCT01 ULCT03 ULCT02</td>
<td></td>
</tr>
<tr>
<td>cbCLoad32()</td>
<td>ULCT06</td>
<td></td>
</tr>
<tr>
<td>cbCStoreOnInt()</td>
<td>ULCT03</td>
<td></td>
</tr>
<tr>
<td>cbCStatus()</td>
<td>ULCT06</td>
<td></td>
</tr>
<tr>
<td>cbDaqInScan()</td>
<td>DaqInScan01 DaqInScan02 DaqInScan03</td>
<td>Demonstrates how to synchronously scan analog, counter, and thermocouple input channels, and digital input ports. Board 0 must support synchronous input.</td>
</tr>
<tr>
<td>cbDaqOutScan()</td>
<td>DaqOutScan01</td>
<td>Demonstrates how to synchronously write to an analog output channel and a digital output port in the background. Board 0 must support synchronous output.</td>
</tr>
<tr>
<td>cbDaqSetSetpoints()</td>
<td>DaqSetSetpoints01</td>
<td>Demonstrates how to configure and use setpoints, including how to add the setpoint status to the scanlist and perform asynchronous reads of the setpoint status. Board 0 must support cbDaqInScan().</td>
</tr>
<tr>
<td>cbDaqSetTrigger()</td>
<td>DaqSetTrigger01</td>
<td>Demonstrates how to set up start and stop trigger events and display input channel data.</td>
</tr>
<tr>
<td>cbDBitIn()</td>
<td>ULDI02 ULDI06 ULDI05 ULMBDI02</td>
<td></td>
</tr>
<tr>
<td>cbDBitOut()</td>
<td>ULDI02 ULDI05 ULMBDI02</td>
<td></td>
</tr>
<tr>
<td>cbDConfigBit()</td>
<td>ULDI06</td>
<td></td>
</tr>
<tr>
<td>cbDConfigPort()</td>
<td>ULDI01 ULDI02 ULDI03 ULDI05 ULEV01 ULEV04 ULEV03</td>
<td></td>
</tr>
<tr>
<td>UL Function Call</td>
<td>UL Example Program Name</td>
<td>Special Features / Notes</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cbDIn()</td>
<td>ULDI01 ULDI04 ULDI03 ULMBDI01 ULEV04*</td>
<td></td>
</tr>
<tr>
<td>cbDInScan()</td>
<td>ULDI03</td>
<td></td>
</tr>
<tr>
<td>cbDeclareRevision()</td>
<td>All samples</td>
<td>All example programs use this function</td>
</tr>
<tr>
<td>cbDOut()</td>
<td>ULD001 ULD005 ULD002 ULMBD001 ULD004 ULMBD002 ULEV03* ULEV04*</td>
<td></td>
</tr>
<tr>
<td>cbDOutScan()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>cbEnableEvent()</td>
<td>ULEV01* ULEV03*</td>
<td>ONEXTERNAL_INTERRUPT ONDATA_AVAILABLE ONPRETRIGGER ONENDOFAO_SCAN ONSCANEVENT ONENDOFAISCAN</td>
</tr>
<tr>
<td>cbDisableEvent()</td>
<td>ULEV02* ULEV04*</td>
<td></td>
</tr>
<tr>
<td>cbErrHandling()</td>
<td>All samples</td>
<td>All example programs use this function</td>
</tr>
<tr>
<td>cbFileAINScan()</td>
<td>ULFI01 ULFI02</td>
<td></td>
</tr>
<tr>
<td>cbFilePretrig()</td>
<td>ULFI03</td>
<td></td>
</tr>
<tr>
<td>cbFileRead()</td>
<td>ULFI02 ULFI03</td>
<td></td>
</tr>
<tr>
<td>cbFromEngUnits()</td>
<td>ULAI01 ULAI003 ULAI07 ULEV04* ULAI14</td>
<td></td>
</tr>
<tr>
<td>cbGetBoardName()</td>
<td>ULGT03 ULGT04</td>
<td></td>
</tr>
<tr>
<td>cbGetConfig()</td>
<td>ULGT03 ULTI01 ULGT04 ULTI02</td>
<td></td>
</tr>
<tr>
<td>cbGetErrMsg()</td>
<td>ULGT01</td>
<td></td>
</tr>
<tr>
<td>cbGetRevision()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>cbGetStatus()</td>
<td>ULAI03 – ULAI06 ULAIO01 ULAI09 ULCT03 ULDI03</td>
<td></td>
</tr>
<tr>
<td>cbGetTCValues()</td>
<td>DaqInScan03</td>
<td>Demonstrates how to retrieve analog, thermocouple, and digital data from a synchronous scan operation. Board 0 must support synchronous output.</td>
</tr>
<tr>
<td>cbInByte()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>cbInWord()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>cbLogConvertFile()</td>
<td>ULLLOG04</td>
<td>Demonstrates how to convert a binary log file to a .CSV file.</td>
</tr>
<tr>
<td>cbLogGetAINchannelCount()</td>
<td>ULLLOG02 ULLLOG03</td>
<td>Demonstrates how to retrieve the number of analog channels contained in a binary log file.</td>
</tr>
<tr>
<td>cbLogGetAINinfo()</td>
<td>ULLLOG03</td>
<td>Demonstrates how to retrieve information about the analog input data contained in a binary log file and then write the data to an array.</td>
</tr>
<tr>
<td>cbLogGetCJCinfo()</td>
<td>ULLLOG02 ULLLOG03</td>
<td>Demonstrates how to retrieve information about the CJC temperature data and digital I/O channel data contained in a binary log file.</td>
</tr>
<tr>
<td>cbLogGetDIOinfo()</td>
<td>ULLLOG02 ULLLOG03</td>
<td>Demonstrates how to retrieve the version level and byte size of a binary log file.</td>
</tr>
<tr>
<td>cbLogGetFileInfo()</td>
<td>ULLLOG01 – ULLLOG03</td>
<td>Demonstrates how to retrieve a binary log file name.</td>
</tr>
<tr>
<td>cbLogGetPreferences()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>UL Function Call</td>
<td>UL Example Program Name</td>
<td>Special Features / Notes</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cbLogGetSampleInfo()</td>
<td>ULLOG02 ULLOG03 ULLOG04</td>
<td>Demonstrates how to retrieve the sample interval, sample count, and the date and time of the first data point logged in a binary file.</td>
</tr>
<tr>
<td>cbLogReadAIClchannels()</td>
<td>ULLOG03</td>
<td>Demonstrates how to retrieve analog input data, CJC data, DIO channel data, and date/time values contained in a binary log file, and store the data in separate arrays.</td>
</tr>
<tr>
<td>cbLogSetPreferences()</td>
<td>ULLOG03</td>
<td>Demonstrates how to store preference settings for time stamped data, analog data, and CJC temperature data.</td>
</tr>
<tr>
<td>cbMemRead()</td>
<td>ULM01 – ULM03</td>
<td></td>
</tr>
<tr>
<td>cbMemReadPretrig()</td>
<td>ULM01</td>
<td></td>
</tr>
<tr>
<td>cbMemReset()</td>
<td>ULM03</td>
<td></td>
</tr>
<tr>
<td>cbMemSetDITMode()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>cbMemWrite()</td>
<td>ULM02</td>
<td></td>
</tr>
<tr>
<td>cbOutByte()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>cbOutWord()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>cbRS485()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>cbSetConfig()</td>
<td>ULA003</td>
<td>Demonstrates the difference between BIDACUPDATEMODE settings of UPDATEIMMEDIATE and UPDATEONCOMMAND. Board 0 must support BIDACUPDATEMODE settings, such as the PCI-DAC6700 Series boards.</td>
</tr>
<tr>
<td>cbSetTrigger()</td>
<td>ULA14</td>
<td></td>
</tr>
<tr>
<td>cbStopBackground()</td>
<td>ULA03 – ULA06 ULA09 ULA001 ULD03 ULEV02* – ULEV04*</td>
<td>Concurrent cbAInScan() and cbAOutScan()</td>
</tr>
<tr>
<td>cbTimerOutStart()</td>
<td>TimerOut01</td>
<td>Demonstrates how to send a frequency output to a timer output channel. Board 0 must have a timer output.</td>
</tr>
<tr>
<td>cbTimerOutStop()</td>
<td>TimerOut01</td>
<td></td>
</tr>
<tr>
<td>cbToEngUnits()</td>
<td>ULA01 ULA11 ULA07 ULEV02* ULEV03*</td>
<td></td>
</tr>
<tr>
<td>cbTIn()</td>
<td>ULI01</td>
<td></td>
</tr>
<tr>
<td>cbTInScan()</td>
<td>ULI02</td>
<td></td>
</tr>
<tr>
<td>cbVIn()</td>
<td>VIn01</td>
<td>Demonstrates how to read an A/D input channel.</td>
</tr>
<tr>
<td>cbVOut()</td>
<td>VOut01</td>
<td>Demonstrates how to write to a D/A output channel.</td>
</tr>
<tr>
<td>cbWinArrayToBuf()</td>
<td>ULA01 ULA002 ULEV04*</td>
<td></td>
</tr>
<tr>
<td>cbWinBufAlloc()</td>
<td>ULA01 – ULA06 ULA08 – ULA10 ULA12 – ULA14 ULA002 ULD03 ULEV02* – ULEV04* (ULEV04:WinBufAlloc and WinBufFree only)</td>
<td></td>
</tr>
<tr>
<td>cbWinBufFree()</td>
<td>ULA101</td>
<td></td>
</tr>
<tr>
<td>cbWinBufToArray()</td>
<td>ULA01 – ULA06 ULA08 – ULA10 ULA12 – ULA14 ULA002 ULD03 ULEV02* – ULEV04* (ULEV04:WinBufAlloc and WinBufFree only)</td>
<td></td>
</tr>
<tr>
<td>cbWinArrayToBuff()</td>
<td>ULA001 ULA002 ULEV04*</td>
<td></td>
</tr>
<tr>
<td>UL Function Call</td>
<td>UL Example Program Name</td>
<td>Special Features / Notes</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cbWinBufAlloc32()</td>
<td>CInScan01</td>
<td>Demonstrates how to allocate a Windows global memory buffer for use with 32-bit scan functions.</td>
</tr>
<tr>
<td></td>
<td>CInScan02</td>
<td></td>
</tr>
<tr>
<td>cbWinBufToArray32()</td>
<td>CInScan01</td>
<td>Demonstrates how to copy 32-bit data from a Windows memory buffer into an array.</td>
</tr>
<tr>
<td></td>
<td>CInScan02</td>
<td></td>
</tr>
</tbody>
</table>

*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 3 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>
<thead>
<tr>
<th>UL settings</th>
<th>Value</th>
<th>UL settings</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIP20VOLTS</td>
<td>±20 volts (V)</td>
<td>UNI10VOLTS</td>
<td>0 to 10 V</td>
</tr>
<tr>
<td>BIP10VOLTS</td>
<td>±10 V</td>
<td>UNI5VOLTS</td>
<td>0 to 5 V</td>
</tr>
<tr>
<td>BIP5VOLTS</td>
<td>±5 V</td>
<td>UNI2PT5VOLTS</td>
<td>0 to 2.5 V</td>
</tr>
<tr>
<td>BIP4VOLTS</td>
<td>±4 V</td>
<td>UNI2VOLTS</td>
<td>0 to 2 V</td>
</tr>
<tr>
<td>BIP2PT5VOLTS</td>
<td>±2.5 V</td>
<td>UNI1PT25VOLTS</td>
<td>0 to 1.25 V</td>
</tr>
<tr>
<td>BIP2VOLTS</td>
<td>±2 V</td>
<td>UNI1PT67VOLTS</td>
<td>0 to 1.67 V</td>
</tr>
<tr>
<td>BIP1PT25VOLTS</td>
<td>±1.25 V</td>
<td>UNI1VOLTS</td>
<td>0 to 1 V</td>
</tr>
<tr>
<td>BIP1VOLTS</td>
<td>±1 V</td>
<td>UNIPT5VOLTS</td>
<td>0 to 0.5 V</td>
</tr>
<tr>
<td>BIP1PT67VOLTS</td>
<td>±1.67 V</td>
<td>UNIPT25VOLTS</td>
<td>0 to 0.25 V</td>
</tr>
<tr>
<td>BIPPT625VOLTS</td>
<td>±0.625 V</td>
<td>UNIPT2VOLTS</td>
<td>0 to 0.2 V</td>
</tr>
<tr>
<td>BIPPT5VOLTS</td>
<td>±0.5 V</td>
<td>UNIPT1VOLTS</td>
<td>0 to 0.1 V</td>
</tr>
<tr>
<td>BIPPT25VOLTS</td>
<td>±0.25 V</td>
<td>UNIPT01VOLTS</td>
<td>0 to 0.01 V</td>
</tr>
<tr>
<td>BIPPT2VOLTS</td>
<td>±0.2 V</td>
<td>UNIPT02VOLTS</td>
<td>0 to 0.02 V</td>
</tr>
<tr>
<td>BIPPT1VOLTS</td>
<td>±0.1 V</td>
<td>MA4TO20</td>
<td>4 to 20 milliamperes (mA)</td>
</tr>
<tr>
<td>BIPPT05VOLTS</td>
<td>±0.05 V</td>
<td>MA2TO10</td>
<td>2 to 10 mA</td>
</tr>
<tr>
<td>BIPPT01VOLTS</td>
<td>±0.01 V</td>
<td>MA1TO5</td>
<td>1 to 5 mA</td>
</tr>
<tr>
<td>BIPPT005VOLTS</td>
<td>±0.005 V</td>
<td>MAPT5TO2PT5</td>
<td>0.5 to 2.5 mA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MA0TO20</td>
<td>0 to 20 mA</td>
</tr>
</tbody>
</table>
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 board-specific information in the Universal Library User's Guide). The data returned to ADData consists of just the 12-bit A/D value. The data returned to ChanTags consists of just the channel numbers.

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. Refers to the number associated with the board used to collect the data when it was installed with the InstaCal configuration program.

NumPoints Number of samples to convert

ADData Pointer or reference to the start of the data array

ChanTags Pointer or reference to the start of the 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() on some devices, the CONVERTDATA option is not allowed. 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

- The 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 InstaCal configuration file. If this function is called for a 16-bit board, it is simply ignored, and no errors are generated.
cbAConvertPretrigData()

Changed R3.3 RW

For products with pretrigger implemented in hardware (most products), this function converts and aligns 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 is 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. 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—must match the value returned by the PretrigCount argument in the cbAPretrig() function

TotalCount Total number of samples that were collected

ADData Pointer to the data array—must match the array name used in the cbAPretrig() function

ChanTags Pointer to the 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 32).

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() on some devices, the CONVERTDATA option is not allowed. In those cases this function should be used to convert the data after the data collection is complete.
Analog I/O Functions

cbAConvertPretrigData()

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.
- The 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++:
```c
int cbACalibrateData(int BoardNum, long NumPoints, int Range,
unsigned ADData[ ])
```

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

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

Arguments:

- **BoardNum**
  May be 0 to 99. Refers to the 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 3 on page 28 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. 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. Refer to board-specific information in the Universal Library User's Guide 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 3 on page 28 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++:
```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. 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, and 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, and HighChan is ignored.

Low / High Channel number: 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

The 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, the number of samples collected per channel is equal to Count/(HighChan - LowChan + 1).

Rate

The rate at which samples are acquired, 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.
Analog I/O Functions

cbAInScan()

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 3 on page 28 for valid values.

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. 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 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. Rates attainable using SINGLEIO are PC-dependent and generally less than 10 kHz. Use the default method unless you have a reason to select a specific transfer method.

- **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 (for example 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.

- **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.
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. On some devices, CONVERTDATA may not be specified if you are using the BACKGROUND option and DMA transfers. This option is ignored for the 16-bit boards.

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() with AIFUNCTION 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() with AIFUNCTION 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.

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 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) for packet size information for your particular DAQ hardware.

If this option is used, 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 (refer to the board-specific information in the Universal Library User's Guide). In most cases, 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.

In some cases, such as with the PCI-DAS4020/12, an approximation of the rate is used to determine the size of the packets to transfer from the board. Set the Rate argument to an approximate maximum value.
**Analog I/O Functions**

**cbAInScan()**

**SINGLEIO** is recommended for slow external clock rates: If the rate of the external clock is very slow (for example 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.**

**EXTRIGGER**

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 60 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 until it goes high. Acquisition will 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, 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. This is done 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.

**RETRIGMODE**

Re-arms the trigger after a trigger event is performed. With this mode, the scan begins when a trigger event occurs. When the scan completes, the trigger is re-armed to acquire the next the batch of data. You can specify the number of samples in the scan for each trigger event (described below). The RETrIGMODE option can be used with the CONTINUOUS option to continue arming the trigger until cbStopBackground() is called.
You can specify the number of samples to acquire with each trigger event. This is the trigger count. Use the ConfigItem option BIADTRIGCOUNT with cbSetConfig() to set the trigger count. If you specify a trigger count that is either zero or greater than the value of the cbAInScan() Count argument, the trigger count will be set to the value of the Count argument.

Specify the CONTINUOUS option with the trigger count set to zero to fill the buffer with Count samples, re-arm the trigger, and refill the buffer upon the next trigger.

**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 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, but the CIO-DAS16/JR is capable of converting only 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/smul-user-guide.pdf](http://www.mccdaq.com/PDFmanuals/smul-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 that is contained in the *Universal Library User's Guide*. We suggest that you make a copy of this information for reference 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. 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++:

```c
int cbAOut(int BoardNum, int Channel, int Range, unsigned short DataValue)
```

Visual Basic:

```vb
Function cbAOut(ByVal BoardNum&, ByVal Channel&, ByVal Range&, ByVal DataValue%) As Long
```

Delphi:

```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. 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 3 on page 28 for valid values.

- **DataValue**
  Value to set D/A to. Must be in the range 0 - N where N is the value $2^{\text{Resolution}} - 1$ 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++:  
```c
int cbAOutScan(int BoardNum, int LowChan, int HighChan, long NumPoints, long *Rate, int Range, int MemHandle, int Options)
```

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

Delphi:  
```delphi
function cbAOutScan(BoardNum:Integer; LowChan:Integer; HighChan:Integer; NumPoints: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 InstaCal configuration program. The specified board must have a D/A. BoardNum may be 0 to 99.

**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 argument returns the value of the actual rate set. This value may be different from the user specified rate due to 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 is 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 gain, this argument is ignored. See Table 3 on page 28 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 43.
Analog I/O Functions

cbAOutScan()

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 number (NumPoints) of D/A values, it resets to the start of the buffer and begins again. The only way to stop this operation is by calling cbStopBackground() with AOFUNCTION. 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() with AOFUNCTION to check the status of background operation. Alternatively, some boards support EnableEvent() for event notification of changes in status of BACKGROUND scans. Use cbStopBackground() with AOFUNCTION 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 specified, 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 (refer to board-specific information contained in the Universal Library Users Guide).

When this option is used the Rate argument is ignored. The sampling rate is dependent on the clock signal. Options for the board 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 60 and board-specific information for details).

ADCCLOCKTRIG
Triggers a data output operation when the ADC clock starts.

ADCCLOCK
Paces the data output operation using the ADC clock.

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 60 for details.

Function prototype:

C/C++:
```c
int cbAPretrig(int BoardNum, int LowChan, int HighChan, long *PretrigCount, long *TotalCount, long *Rate, int Range, int MemHandle, int Options)
```

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

Delphi:
```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 InstaCal configuration program. The specified board must have an A/D. BoardNum may be 0 to 99.

**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.

For products using a hardware implementation of pretrigger (most products), PretrigCount must be less than (TotalCount - 512). For these devices, 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.

For software implementations of pretrigger, PretrigCount must be less than TotalCount. For these devices, triggers that occur before the requested number of pre-trigger samples are collected are ignored. See board-specific information.

**TotalCount**

Total number of A/D samples to collect. Specifies the total number of samples that will be collected and stored in the buffer.

For products using a hardware implementation of pretrigger (most products), 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.

For software implementations of pretrigger, TotalCount must be greater than PretrigCount. For these devices, triggers that occur before the requested number of pre-trigger samples are collected are ignored. See board-specific information.
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.

PretrigCount must also be evenly divisible by the number of channels being scanned. If it is not, this function will adjust the number (up) to the next valid value and return that value to the PretrigCount 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 3 on page 28 for valid values.

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

For hardware trigger types, the buffer referenced by MemHandle must be big enough to hold at least TotalCount + 512 integers.

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  
For hardware trigger types, 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. The CONVERTDATA option is not required for software triggered types.

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() with AIFUNCTION 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() with AIFUNCTION to terminate the background process before it has completed.

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

For hardware trigger types, 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.

Important:
For hardware trigger types, 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 InstaCal configuration program. The specified board must have an A/D. BoardNum may be 0 to 99.

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 3 on page 28 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 the 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.
cbVIn()

Reads an A/D input channel, and returns a voltage value. If the specified A/D board has programmable gain, then this function sets the gain to the specified range. The voltage value is returned to `DataValue`.

**Function prototype:**

- **C/C++:**
  ```c
  int cbVIn(int BoardNum, int Channel, int Range, float *DataValue, int Options);
  ```

- **Visual Basic:**
  ```vb
  Function cbVIn(ByVal BoardNum&, ByVal Channel&, ByVal Range&, DataValue!, ByVal Options&) As Long
  ```

- **Delphi:**
  ```delphi
  function cbVIn(BoardNum:Integer; Channel:Integer; Range:Integer; Var DataValue:Single; Options:Integer):Integer;
  ```

**Arguments:**

- **BoardNum**
  The board number associated with the board used to collect the data when it was installed with the InstaCal configuration program. `BoardNum` may be 0 to 99. 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.

- **Range**
  A/D range code. If the board has a programmable gain, it will be set according to this argument value. Keep in mind that some A/D boards have a programmable gain feature, and others set the gain via switches on the board. In either case, the range that the board is configured for must be passed to this function. Refer to Table 3 on page 28 for a list of valid range settings.

- **DataValue**
  A pointer or reference to the data value.

- **Options**
  Reserved for future use.

**Returns:**

- **Error code** or 0 if no errors.

- **DataValue** - Returns the value in volts of the A/D sample.

**Options argument values:**

- **Default**
  Reserved for future use.
cbVOut()

Sets the value of a D/A output.

Function prototype:

C/C++: int cbVOut(int BoardNum, int Channel, int Range, float DataValue, int Options);

Visual Basic: Function cbVOut(ByVal BoardNum&, ByVal Channel&, ByVal Range&, ByVal DataValue!, ByVal Options&) As Long

Delphi: function cbVOut(BoardNum:Integer; Channel:Integer; Range:Integer; DataValue:Single; Options:Integer):Integer;

Arguments:

BoardNum The board number associated with the board used to collect the data when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99. The specified board must have an D/A.

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

Range The D/A range code. If the board has a programmable gain, it will be set according to this argument value. The output range of the D/A channel can be set to any of those supported by the board. Keep in mind that some D/A boards have programmable voltage ranges, and others set the voltage range via switches on the board. In either case, the range that the board is configured for must be passed to this function.

DataValue The voltage value to be written.

Options Reserved for future use.

Returns:

Error code or 0 if no errors.

Options argument values:

Default Reserved for future use.
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.

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 information from.

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 52.

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:

<table>
<thead>
<tr>
<th>InfoType</th>
<th>ConfigItem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBALINFO</td>
<td>GIVERSION</td>
<td>CB.CFG file format - used by the library to determine compatibility.</td>
</tr>
<tr>
<td></td>
<td>GINUMBOARDS</td>
<td>Maximum number of installable boards</td>
</tr>
<tr>
<td></td>
<td>GINUMEXPBOARDS</td>
<td>Maximum number of expansion boards allowed to be installed.</td>
</tr>
<tr>
<td>BOARDINFO</td>
<td>BIADCSETTLETIME</td>
<td>ADC settling time</td>
</tr>
<tr>
<td></td>
<td>BIBASEADR</td>
<td>Base address of the board</td>
</tr>
<tr>
<td></td>
<td>BIBOARDBASE</td>
<td>Returns a unique number in the range of 0 to 8000 Hex describing the board type installed.</td>
</tr>
<tr>
<td></td>
<td>BICIDDEVNUM</td>
<td>Index into counter information for the first device</td>
</tr>
<tr>
<td></td>
<td>BICINUMDEVS</td>
<td>Number of counter devices</td>
</tr>
<tr>
<td></td>
<td>BICLOCK</td>
<td>Clock frequency in MHz (40, 10, 8, 6, 5, 4, 3, 2, 1) or 0 for not supported.</td>
</tr>
<tr>
<td></td>
<td>BIDACSTARTUP</td>
<td>Returns the setting of a DAC board's configuration register STARTUP bit.</td>
</tr>
<tr>
<td></td>
<td>BIDACUPDATEMODE</td>
<td>Setting of the update mode for a digital-to-analog converter (DAC).</td>
</tr>
<tr>
<td></td>
<td>BIDACUPDATETIME</td>
<td>Refer to the &quot;Notes&quot; section for more information.</td>
</tr>
<tr>
<td></td>
<td>BIDIDEVNUM</td>
<td>Index into digital information for the first device</td>
</tr>
<tr>
<td></td>
<td>BIDNUMDEVS</td>
<td>Number of digital devices</td>
</tr>
<tr>
<td></td>
<td>BIDMACHAN</td>
<td>DMA channel. 0, 1 or 3</td>
</tr>
<tr>
<td></td>
<td>BIDBOARD</td>
<td>Board number of the connected DT board</td>
</tr>
<tr>
<td></td>
<td>BIFACTORYID</td>
<td>The factory serial number of a USB device, or the MAC address of a WEB device.</td>
</tr>
<tr>
<td></td>
<td>BIHIDELOGINDLG</td>
<td>Enables or disables the Device Login dialog. Set to a nonzero value to disable the dialog. When disabled, the cbDeviceLogin() function must be used to log in to a device session.</td>
</tr>
<tr>
<td></td>
<td>BIINTLEVEL</td>
<td>Interrupt level. 0 for none, or 1 - 15</td>
</tr>
<tr>
<td></td>
<td>BINETIOTIMEOUT</td>
<td>The amount of time (in milliseconds) to wait for a WEB device to acknowledge a command or query sent to the device over a network connection. If no acknowledgement is received in this time a timeout occurs.</td>
</tr>
<tr>
<td></td>
<td>BINUMADCHANS</td>
<td>Number of A/D channels</td>
</tr>
<tr>
<td></td>
<td>BINUMDACHANS</td>
<td>Number of D/A channels</td>
</tr>
<tr>
<td></td>
<td>BINUMIOPORTS</td>
<td>Number of IO ports used by board</td>
</tr>
<tr>
<td></td>
<td>BIPANID</td>
<td>Personal Area Network (PAN) identifier for a USB device that supports wireless communication.</td>
</tr>
<tr>
<td></td>
<td>BIRANGE</td>
<td>Selected voltage range. For switch-selectable gains only.</td>
</tr>
<tr>
<td></td>
<td>BIRFCHANNEL</td>
<td>Returns the RF channel number used to transmit/receive data by a USB device that supports wireless communication.</td>
</tr>
<tr>
<td></td>
<td>BIRSS</td>
<td>Returns the received signal strength in dBm of a remote device.</td>
</tr>
<tr>
<td></td>
<td>BISERIALNUM</td>
<td>Returns the serial number assigned by a user to a USB device in InstaCal.</td>
</tr>
<tr>
<td></td>
<td>BIWAITSTATE</td>
<td>Setting of Wait State jumper. 1 = enabled, 0 = disabled</td>
</tr>
<tr>
<td></td>
<td>BIUSESEXP</td>
<td>Supports expansion boards TRUE/FALSE</td>
</tr>
</tbody>
</table>
**InfoType** | **ConfigItem** | **Description**
---|---|---
DIGITALINFO | DIDEVTYPE | Device Type - AUXPORT, FIRSTPORTA etc.
 | DICONFIG | Current configuration INPUT or OUTPUT
 | DINUMBITS | Number of bits in the port
 | DICURVAL | Current value of outputs
 | DITINMASK | Returns the bit configuration of the specified port. Any bits that return a value of 1 are configured for input. Refer to the "Notes" section below for more information.
 | DIOUTMASK | Returns the bit configuration of the specified port. Any bits that return a value of 1 are configured for output. Refer to the "Notes" section below for more information.
COUNTERINFO | CICTRTYPE | Counter chip type, where 1 = 8254, 2 = 9513, 3 = 8536, 4 = 7266, 5 = event counter, 6 = scan counter, and 7 = timer counter.
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** DITINMASK **and** DIOUTMASK **options to determine if an AUXPORT is configurable. Execute** cbGetConfig() **twice to the same port—one using DITINMASK 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 **Anding** 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 DITINMASK 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 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 61 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:

```c
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.
cbGetConfigString()

Retrieves configuration or device information as a null-terminated string.

Function prototype:

C/C++: `int cbGetConfigString(int InfoType, int BoardNum, int ItemIndex, int ConfigItem, char *ConfigVal, int* maxConfigLen)`

Visual Basic: `Function cbGetConfigString(InfoType%, BoardNum%, ItemIndex%, ConfigItem%, ConfigVal$, maxConfigLen%) As Long`

Delphi: `function cbGetConfigString(InfoType: Integer; BoardNum: Integer; ItemIndex: Integer; ConfigItem: Integer; ConfigVal: PChar; var maxConfigLen: Integer): Integer;`

Arguments:

- **InfoType**: The configuration information for each board is grouped into different categories. This argument specifies which category you want. Always set this argument to `BOARDINFO`.
- **BoardNum**: Refers to the board number associated with a board when it was installed. BoardNum may be 0 to 99.
- **ItemIndex**: The location in the device memory (specified by ConfigItem) at which to start reading.
- **ConfigItem**: Specifies the type of information (or memory area) to read from the device. Set it to one of the constants listed in the "ConfigItem argument values" section below.
- **ConfigVal**: Pointer to a user-allocated buffer where the configuration string is copied.
- **maxConfigLen**: Pointer to the value holding the maximum number of bytes to be read from the device into ConfigVal.

Returns:

- **Error code or 0 if no errors.**
- **maxConfigLen**: The number of bytes read from the device into ConfigVal.
- **ConfigVal**: The string read from the device.

**ConfigItem argument values:**

- **BIDEVNOTES**: Reads up to `maxConfigLen` characters/bytes from the device notes memory, starting at the location defined by ItemIndex. Currently supported only for WLS Series devices.
- **BIFACTORYID**: Reads the MAC address of a WEB device.
- **BINODEID**: Reads up to `maxConfigLen` character/bytes from the string identifier memory. Note that ItemIndex is not used for this ConfigItem.
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 InstaCal.

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

**Function prototype:**

**C/C++:**

```c
int cbGetSignal(int BoardNum, int Direction, int Signal, int Index,
int* Connection, int* Polarity)
```

**Visual Basic:**

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

**Delphi:**

```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 57 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. 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:**

Timing and control configuration information can be viewed and edited inside InstaCal. Do the following:

1. Run InstaCal.
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 InstaCal.

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

**Function prototype:**

**C/C++:**

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

**Visual Basic:**

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

**Delphi:**

```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" section below.

- **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.
Configuration Functions

- **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:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Connection</th>
<th>Polarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_CONVERT</td>
<td>AUXIN0..AUXIN5 DS_CONNECTOR</td>
<td>POSITIVEEDGE or NEGATIVEEDGE</td>
</tr>
<tr>
<td>ADC_GATE</td>
<td>AUXIN0..AUXIN5 DS_CONNECTOR</td>
<td>See cbSetTrigger().</td>
</tr>
<tr>
<td>ADC_START_TRIG</td>
<td>AUXIN0..AUXIN5 DS_CONNECTOR</td>
<td>See cbSetTrigger().</td>
</tr>
<tr>
<td>ADC_STOP_TRIG</td>
<td>AUXIN0..AUXIN5 DS_CONNECTOR</td>
<td>See cbSetTrigger().</td>
</tr>
<tr>
<td>ADC_TB_SRC</td>
<td>AUXIN0..AUXIN5 DS_CONNECTOR</td>
<td>POSITIVEEDGE or NEGATIVEEDGE</td>
</tr>
<tr>
<td>DAC_START_TRIG</td>
<td>AUXIN0..AUXIN5 DS_CONNECTOR</td>
<td>Not assigned here.</td>
</tr>
<tr>
<td>DAC_TB_SRC</td>
<td>AUXIN0..AUXIN5 DS_CONNECTOR</td>
<td>POSITIVEEDGE or NEGATIVEEDGE</td>
</tr>
<tr>
<td>DAC_UPDATE</td>
<td>AUXIN0..AUXIN5 DS_CONNECTOR</td>
<td>POSITIVEEDGE or NEGATIVEEDGE</td>
</tr>
<tr>
<td>SYNC_CLK</td>
<td>DS_CONNECTOR</td>
<td>Not assigned here.</td>
</tr>
</tbody>
</table>

Valid output (Direction=SIGNAL_OUT) settings include:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Connection</th>
<th>Polarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_CONVERT</td>
<td>AUXOUT0..AUXOUT2 DS_CONNECTOR</td>
<td>INVERTED* or NONINVERTED</td>
</tr>
<tr>
<td>ADC_SCANCLK</td>
<td>AUXOUT0..AUXOUT2</td>
<td></td>
</tr>
<tr>
<td>ADC_SCAN_STOP</td>
<td>AUXOUT0..AUXOUT2</td>
<td></td>
</tr>
<tr>
<td>ADC_SSH</td>
<td>AUXOUT0..AUXOUT2</td>
<td></td>
</tr>
<tr>
<td>ADC_STARTSCAN</td>
<td>AUXOUT0_AUXOUT2</td>
<td></td>
</tr>
<tr>
<td>ADC_START_TRIG</td>
<td>AUXOUT0..AUXOUT2 DS_CONNECTOR</td>
<td></td>
</tr>
<tr>
<td>ADC_STOP_TRIG</td>
<td>AUXOUT0..AUXOUT2 DS_CONNECTOR</td>
<td></td>
</tr>
<tr>
<td>CTR1_CLK</td>
<td>AUXOUT0 AUXOUT2</td>
<td></td>
</tr>
<tr>
<td>CTR2_CLK</td>
<td>AUXOUT0 AUXOUT2</td>
<td></td>
</tr>
<tr>
<td>DAC_START_TRIG</td>
<td>AUXOUT0..AUXOUT2 DS_CONNECTOR</td>
<td></td>
</tr>
<tr>
<td>DAC_UPDATE</td>
<td>AUXOUT0..AUXOUT2 DS_CONNECTOR</td>
<td></td>
</tr>
<tr>
<td>DGND</td>
<td>AUXOUT0_AUXOUT2</td>
<td>Not assigned here.</td>
</tr>
<tr>
<td>SYNC_CLK</td>
<td>DS_CONNECTOR</td>
<td>Not assigned here.</td>
</tr>
</tbody>
</table>

*INVERTED* is only valid for Auxiliary Output (AUXOUT) connections.
Valid disabled settings (Direction=DISABLED):

<table>
<thead>
<tr>
<th>Signal</th>
<th>Connection</th>
<th>Polarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAC_TB_SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYNC_CLK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:

- You can view and edit the above timing and control configuration information from InstaCal. Open InstaCal, 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 EXTCLK 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 EXTCLK 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 SYNClK does activate the input source for the next respective operations.

- Multiple input signals can be mapped to the same AUXINn 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 AUXOUTn 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**

<table>
<thead>
<tr>
<th>Default connection</th>
<th>Input signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUXIN0</td>
<td>ADC_CONVERT</td>
</tr>
<tr>
<td>AUXIN5</td>
<td>ADC_GATE</td>
</tr>
<tr>
<td>AUXIN1</td>
<td>ADC_START_TRIG</td>
</tr>
<tr>
<td>AUXIN2</td>
<td>ADC_STOP_TRIG</td>
</tr>
<tr>
<td>AUXIN3</td>
<td>DAC_UPDATE</td>
</tr>
<tr>
<td>AUXIN3</td>
<td>DAC_START_TRIG</td>
</tr>
</tbody>
</table>

- 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 argument values" section below.

BoardNum
Refers to the board number associated with a board when it was installed. BoardNum may be 0 to 99.

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 61.

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.
### Configuration Functions

#### cbSetConfig()

**ConfigItem argument values:**

<table>
<thead>
<tr>
<th>InfoType</th>
<th>ConfigItem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOARDFINFO</td>
<td>BIDACSETTLETIME</td>
<td>ADC settling time</td>
</tr>
<tr>
<td></td>
<td>BIADTRIGCOUNT</td>
<td>Trigger count</td>
</tr>
<tr>
<td></td>
<td>BIBASEADR</td>
<td>Base address of the board</td>
</tr>
<tr>
<td></td>
<td>BICALOUTPUT</td>
<td>Sets the voltage for the CAL pin on supported USB devices.</td>
</tr>
<tr>
<td></td>
<td>BICLICK</td>
<td>Clock frequency in MHz (1, 4, 6 or 10)</td>
</tr>
<tr>
<td></td>
<td>BIDACSTARTUP</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>BIDACUPDATECMD</td>
<td>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 &quot;Notes&quot; section below for more information.</td>
</tr>
<tr>
<td></td>
<td>BIDACUPDATEMODE</td>
<td>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 &quot;Notes&quot; section below for more information.</td>
</tr>
<tr>
<td></td>
<td>BIDIDEBOUNCESTATE</td>
<td>The state of the digital inputs when debounce timing is set.</td>
</tr>
<tr>
<td></td>
<td>BIDIDEBOUNCETIME</td>
<td>Sets the debounce time of digital inputs</td>
</tr>
<tr>
<td></td>
<td>BIMACHAN</td>
<td>DMA channel</td>
</tr>
<tr>
<td></td>
<td>BINTLEVEL</td>
<td>Interrupt level</td>
</tr>
<tr>
<td></td>
<td>BIIHIDLOGINDLG</td>
<td>Enables or disables the Device Login dialog. Set to a nonzero value to disable the dialog. When disabled, the cbDeviceLogin() function must be used to log in to a device session.</td>
</tr>
<tr>
<td></td>
<td>BINETIOTIMEOUT</td>
<td>Sets the amount of time (in milliseconds) to wait for a WEB device to acknowledge a command or query sent to the device over a network connection. If no acknowledgement is received in this time a timeout occurs.</td>
</tr>
<tr>
<td></td>
<td>BINUMADCHANS</td>
<td>Number of A/D channels</td>
</tr>
<tr>
<td></td>
<td>BIFANID</td>
<td>Sets the Personal Area Network (PAN) identifier of a USB device that supports wireless communication.</td>
</tr>
<tr>
<td></td>
<td>BIRANGE</td>
<td>Selected voltage range</td>
</tr>
<tr>
<td></td>
<td>BIRFCHANNEL</td>
<td>Sets the RF channel number used to transmit/receive data by a USB device that supports wireless communication.</td>
</tr>
<tr>
<td></td>
<td>BIRSS</td>
<td>The received signal strength in dBm of a remote device.</td>
</tr>
<tr>
<td></td>
<td>BISRCADPACER</td>
<td>Outputs the A/D pacer signal to the SYNC pin on supported USB devices.</td>
</tr>
<tr>
<td></td>
<td>BINWAITSTATE</td>
<td>Sets the Wait State jumper</td>
</tr>
<tr>
<td>EXPANSIONINFO</td>
<td>XIMIXXADCHAN1</td>
<td>A/D channel board is connect to</td>
</tr>
<tr>
<td></td>
<td>XIMIXXADCHAN2</td>
<td>2nd A/D channel board is connected to</td>
</tr>
<tr>
<td></td>
<td>XIRANGE1</td>
<td>Range (gain) of low 16 channels</td>
</tr>
<tr>
<td></td>
<td>XIRANGE2</td>
<td>Range (gain) of high 16 channels</td>
</tr>
<tr>
<td></td>
<td>XICJCCHAN</td>
<td>A/D channel that CJC is connected to</td>
</tr>
<tr>
<td></td>
<td>XITHERMTYPE</td>
<td>Thermocouple type</td>
</tr>
</tbody>
</table>

#### 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`. 
cbSetConfigString()

Sets the configuration or device information as a null-terminated string.

Function prototype:

C/C++:  
```c
int cbSetConfigString(int InfoType, int BoardNum, int ItemIndex, int ConfigItem, char *ConfigVal, int* maxConfigLen)
```

Visual Basic:  
```vb
Function cbSetConfigString(ByVal InfoType, ByVal BoardNum&, ByVal ItemIndex&, ByVal ConfigItem&, ByVal ConfigVal$, ByRef maxConfigLen&) As Long
```

Delphi:  
```delphi
function cbSetConfigString(InfoType:Integer; BoardNum:Integer; ItemIndex:Integer; ConfigItem:Integer; ConfigVal:PChar; var maxConfigLen:Integer):Integer;
```

Arguments:

- **InfoType**  
  The configuration information for each board is grouped into different categories. This argument specifies which category you want. Always set this argument to BOARDINFO.

- **BoardNum**  
  Refers to the board number associated with a board when it was installed. BoardNum may be 0 to 99.

- **ItemIndex**  
  The location in the device memory (specified by ConfigItem) at which to start writing.

- **ConfigItem**  
  The type of information (or memory area) to write to the device. Set it to one of the constants listed in the "ConfigItem argument values" section below.

- **ConfigVal**  
  Pointer to the user-allocated buffer containing the string to copy to the device's memory.

- **maxConfigLen**  
  Pointer to the value specifying the number of bytes to be written to the device from ConfigVal.

Returns:

- **Error code** or 0 if no errors.

- **maxConfigLen**  
  The number of bytes written to the device.

**ConfigItem argument values:**

- **BIDEVNOTES**  
  Writes up to `maxConfigLen` characters/bytes from the ConfigVal buffer to the device notes memory, beginning at the location defined by ItemIndex. Currently supported only for WLS Series devices.

- **BINODEID**  
  Writes up to `maxConfigLen` characters/bytes from the ConfigVal buffer to the string identifier memory on the device. Note that ItemIndex is not used for this ConfigItem.
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.
- **cbDInScan()**, if the EXTTRIGGER option is selected.
- **cbCInScan()**, if the EXTTRIGGER option is selected.
- **cbAPretrig()**
- **cbFilePretrig()**

**Function prototype:**

- **C/C++:**
  ```c
  int cbSetTrigger(int BoardNum, int TrigType, unsigned short LowThreshold, unsigned short HighThreshold);
  ```
- **Visual Basic:**
  ```vb
  Function cbSetTrigger(ByVal BoardNum&, ByVal TrigType&, ByVal LowThreshold%, ByVal HighThreshold%) As Long
  ```
- **Delphi:**
  ```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.
- **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 65.
- **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 65.
- **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 65.

**Returns:**

- **Error code** or 0 if no errors.
**TrigType argument values:**

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

**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-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.

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 $2^{\text{resolution}}$. FSR is the entire span from $-\text{FS}$ to $+\text{FS}$ 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 $2^{\text{resolution}} - 1$. The formula is shown here:

$$\text{Abs} (-\text{FS} - \text{threshold in volts}) \div (\text{LSB}) = \text{threshold in counts}$$

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:** $\text{LSB} = 20 \div 2^8 = 20 \div 256 = .078125$
  
  **Calculate threshold:** $\text{Abs}(-10 - (-5)) \div .078125 = 5 \div .078125 = 64$ (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:** $\text{LSB} = 20 \div 2^{12} = 20 \div 4096 = .00488$
  
  **Calculate threshold:** $\text{Abs}(-10 - 1) \div .00488 = 11 \div .00488 = 2254$ (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()**

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:\Program files\Measurement Computing\DAQ by default).

This data sheet is also available on our web site at [www.mccdaq.com/PDFmanuals/LS7266R1.pdf](http://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 InstaCal configuration program. The specified board must have an LS7266 counter. BoardNum may be 0 to 99.

- **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_QUAD, 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 69.

- **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 69.

- **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 69.

- **Gating** If gating is set to ENABLED, then the channel INDEX input is routed to the RCNTR pin of the LS7266 chip, and is used as a gating signal for the counter. Whenever Gating = ENABLED the IndexMode must be set to INDEX_DISABLED.

**Returns:**

Error code or 0 if no error occurs
Counter Functions

Counter Functions

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 and 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 channel INDEX input is routed to the LCNTR pin of the LS7266 counter chip. The counter is loaded whenever the signal occurs.
- **LOAD_OUT_LATCH** The channel INDEX input is routed to the LCNTR pin of the LS7266 counter chip. The current count is latched whenever the signal occurs. When this mode is selected, the `cbCl()` function will return the same count value each time it is called until the Index signal occurs.
- **RESET_CTR** The channel INDEX input is routed to the RCNTR pin of the LS7266 counter chip. The counter is reset whenever the signal 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 pin is INDEX output, FLG2 is error output.
Counter Functions

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:\Program Files\Measurement Computing\DAQ 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 InstaCal configuration program. Board must have an 82C54 installed. BoardNum may be 0 to 99.

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).

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 argument values" 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:\Program Files\Measurement Computing\DAQ 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 the 82C54.pdf file located in the Documents subdirectory where you installed UL (C:\Program Files\Measurement Computing\DAQ 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 82C54.pdf file in the Documents subdirectory where you installed UL (C:\Program Files\Measurement Computing\DAQ 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 the 8254 data sheet in accompanying 82C54.pdf file located in the Documents subdirectory where you installed UL.

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 the 82C54.pdf file located in the Documents subdirectory where you installed UL (C:\Program Files\Measurement Computing\DAQ 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:\Program Files\Measurement Computing\DAQ by default).
cbC8536Config()

Configures 8536 counter for desired operation. This function can only be used with 8536 counters. For more information, refer to the Zilog 8536 product specification. The document is available on our web site at www.mccdaq.com/PDFmanuals/Z8536.pdf.

Function prototype:

C/C++:
```cpp
int cbC8536Config(int BoardNum, int CounterNum, int OutputControl,
                   int RecycleMode, int TrigType)
```

Visual Basic:
```vbnet
Function cbC8536Config(ByVal BoardNum&, ByVal CounterNum&, ByVal
                       OutputControl&, ByVal RecycleMode&, ByVal
                       TrigType&) As Long
```

Delphi:
```delphi
function cbC8536Config(BoardNum:Integer; CounterNum:Integer;
                        OutputControl:Integer; RecycleMode:Integer;
                        TrigType: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.

- **CounterNum**
  Selects one of the counter channels. An 8536 has 3 counters. The value may be 1, 2 or 3. INT32 Series boards have two chips installed, so the CounterNum value may be 1 to 6.

- **OutputControl**
  Specifies the action of the output signal. Set it to one of the constants in the "OutputControl argument values" section below.

- **RecycleMode**
  If set to RECYCLE (as opposed to ONETIME), the counter automatically reloads to the starting count every time it reaches 0, then counting continues.

- **TrigType**
  Specifies the trigger type. Set it to one of the constants in the "TrigType argument values" section below.

Returns:

- **Error code** or 0 if no errors

**OutputControl argument values:**

- **HIGHPULSEONTC**
  Output transitions from low to high for one clock pulse on the terminal count.

- **TOGGLEONTC**
  Output changes state on the terminal count.

- **HIGHUNTILTC**
  Output transition to high at the start of counting, then goes low on the terminal count.

**TrigType argument values:**

- **HW_START_TRIG**
  The first trigger on the counter's trigger input initiates loading of the initial count. Counting proceeds from the initial count.

- **HW_RETRIG**
  Every trigger on the counter's trigger input initiates loading of the initial count. Counting proceeds from the initial count.

- **SW_START_TRIG**
  The cbCLoad() function initiates loading of the initial count. Counting proceeds from the initial count.
Counter Functions

**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:\Program files\Measurement Computing\DAQ 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 InstaCal configuration program. The specified board must have a 9513 counter. BoardNum may be 0 to 99.

**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 73.

**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 73.

**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 73.

**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
- ALLEGATE: Active Low Edge GATE N
- ALEGATE: Active Low Level 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:\Program Files\Measurement Computing\DAQ by default). The data sheet is also available on our website at www.mcedaq.com/PDFmanuals/9513A.pdf.
cbC8536Init()

Initializes the counter linking features of an 8536 counter chip. Counters 1 and 2 must be linked before enabling the counters.

Refer to the Zilog 8536 product specification for a description of the hardware affected by this mode. The document is available on our web site at www.mccdaq.com/PDFmanuals/Z8536.pdf.

Function prototype:

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

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:\Program Files\Measurement Computing\DAQ 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 InstaCal configuration program. The specified board must have a 9513 counter. BoardNum may be 0 to 99.

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 and 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 76.

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 76.

**Returns:**

Error code or 0 if no errors
Counter Functions

FOutSource argument values:

<table>
<thead>
<tr>
<th>Argument</th>
<th>9513 Data Sheet Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOutSource</td>
<td>9513 Data Sheet Equivalent</td>
</tr>
<tr>
<td>CTRINPUT1</td>
<td>SRC 1 (Counter Input 1)</td>
</tr>
<tr>
<td>CTRINPUT2</td>
<td>SRC 2 (Counter Input 2)</td>
</tr>
<tr>
<td>CTRINPUT3</td>
<td>SRC 3 (Counter Input 3)</td>
</tr>
<tr>
<td>CTRINPUT4</td>
<td>SRC 4 (Counter Input 4)</td>
</tr>
<tr>
<td>CTRINPUT5</td>
<td>SRC 5 (Counter Input 5)</td>
</tr>
<tr>
<td>GATE1</td>
<td>GATE1</td>
</tr>
<tr>
<td>GATE2</td>
<td>GATE2</td>
</tr>
<tr>
<td>GATE3</td>
<td>GATE3</td>
</tr>
<tr>
<td>GATE4</td>
<td>GATE4</td>
</tr>
<tr>
<td>GATE5</td>
<td>GATE5</td>
</tr>
<tr>
<td>FREQ1</td>
<td>F1</td>
</tr>
<tr>
<td>FREQ2</td>
<td>F2</td>
</tr>
<tr>
<td>FREQ3</td>
<td>F3</td>
</tr>
<tr>
<td>FREQ4</td>
<td>F4</td>
</tr>
<tr>
<td>FREQ5</td>
<td>F5</td>
</tr>
</tbody>
</table>

TimeOfDay argument values:

<table>
<thead>
<tr>
<th>Argument</th>
<th>9513 Data Sheet Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeOfDay</td>
<td>9513 Data Sheet Equivalent</td>
</tr>
<tr>
<td>CBDDISABLED</td>
<td>TOD Disabled</td>
</tr>
<tr>
<td>1</td>
<td>TOD Enabled / 5 Input</td>
</tr>
<tr>
<td>2</td>
<td>TOD Enabled / 6 Input</td>
</tr>
<tr>
<td>3</td>
<td>TOD Enabled / 10 Input</td>
</tr>
<tr>
<td>No arguments for:</td>
<td>9513 data sheet equivalent</td>
</tr>
<tr>
<td>0 (FOUT on)</td>
<td>FOUT Gate</td>
</tr>
<tr>
<td>0 (Data bus matches board)</td>
<td>Data Bus Width</td>
</tr>
<tr>
<td>1 (Disable Increment)</td>
<td>Data Pointer Control</td>
</tr>
<tr>
<td>1 (BCD Scaling)</td>
<td>Scalar Control</td>
</tr>
</tbody>
</table>

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:\Program files\Measurement Computing\DAQ by default). The data sheet is also available on our website at www.mccdaq.com/PDFmanuals/9513A.pdf.
cbCClear()

Cleares a scan counter value (sets it to zero). This function only works with counter boards that have counter scan capability.

**Function prototype:**

**C/C++:**
```
int cbCClear(int BoardNum, int CounterNum)
```

**Visual Basic:**
```
Function cbCClear(ByVal BoardNum$, ByVal CounterNum$) As Long
```

**Delphi:**
```
function cbCClear(BoardNum:Integer; CounterNum:Integer):Integer;
```

**Arguments:**

- **BoardNum**
  The board number associated with the board when it was installed with the *InstaCal* configuration program. The specified board must have a counter. 
  BoardNum may be 0 to 99.

- **CounterNum**
  The counter to clear. **Note:** This argument is zero-based (the first counter number to clear is "0").

**Returns:**

- **Error code** or 0 if no errors
cbCConfigScan()

Configures a counter channel. This function only works with counter boards that have counter scan capability.

Function prototype:

C/C++:
```c
int cbCConfigScan(int BoardNum, short CounterNum, int Mode, int DebounceTime, int DebounceTrigger, int EdgeDetection, int TickSize, int MapCounter);
```

Visual Basic:
```vb
Function cbCConfigScan(ByVal BoardNum&, ByVal CounterNum& ..., ByVal MapCounter& ...) As Long
```

Delphi:
```delphi
function cbCConfigScan(BoardNum: Integer; CounterNum: SmallInt; ..., MapCounter: Integer): Integer;
```

Arguments:

**BoardNum**
The board number used to collect the data. BoardNum may be 0 to 99. 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 scan counter.

**CounterNum**
The counter to set up.

Note: This argument is zero-based (the first counter number to set up is "0").

**Mode**
Bit fields that control various options. This field may contain any combination of non-contradictory choices in the "Mode argument values" section on page 79.

**DebounceTime**
Used to bypass the debounce mode, or to set a channel’s comparator output to one of 16 debounce times. Debounce is used to eliminate switch-induced transients typically associated with electro-mechanical devices including relays, proximity switches, and encoders. The choices are:

- CTR_DEBOUNCE500ns
- CTR_DEBOUNCE1500ns
- CTR_DEBOUNCE3500ns
- CTR_DEBOUNCE7500ns
- CTR_DEBOUNCE15500ns
- CTR_DEBOUNCE31500ns
- CTR_DEBOUNCE63500ns
- CTR_DEBOUNCE127500ns
- CTR_DEBOUNCE100us
- CTR_DEBOUNCE300us
- CTR_DEBOUNCE700us
- CTR_DEBOUNCE1500us
- CTR_DEBOUNCE3100us
- CTR_DEBOUNCE6300us
- CTR_DEBOUNCE12700us
- CTR_DEBOUNCE25500us
- CTR_DEBOUNCE_NONE
DebounceMode

Sets the mode of the debounce module to **CTR_TRIGGER_AFTER_STABLE** or to **CTR_TRIGGER BEFORE_STABLE**.

**CTR_TRIGGER_AFTER_STABLE:** This mode rejects glitches, and only passes state transitions after a specified period of stability (the debounce time). This mode is used with electro-mechanical devices like encoders and mechanical switches to reject switch bounce and disturbances due to a vibrating encoder that is not otherwise moving. The debounce time should be set short enough to accept the desired input pulse but longer than the period of the undesired disturbance.

**CTR_TRIGGER BEFORE_STABLE:** Use this mode when the input signal has groups of glitches and each group is to be counted as one. The trigger before stable mode will recognize and count the first glitch within a group but reject the subsequent glitches within the group if the debounce time is set accordingly. In this case the debounce time should be set to encompass one entire group of glitches.

EdgeDetection

Selects whether to detect rising edge or falling edge. Choices are:

- **CTR_RISING_EDGE**
- **CTR_FALLING_EDGE**

If a counter is configured for **CTR_FALLING_EDGE**, calling **cbCIn()** or **cbCIn32()** for that counter will result in a **BADCOUNTERMODE** error.

TickSize

Reserved.

MapCounter

Used to select the mapped counter. A mapped counter is one of the counter input channels other than **CounterNum** that can participate with the input signal of the counter defined by **CounterNum** by gating the counter or decrementing the counter.

Returns:

**Error code** or 0 if no errors

**Mode argument values:**

- **CLEAR_ON_READ**
  The counter counts up and is cleared at the beginning of every sample. By default, the counter counts up and only clears the counter at the start of a new scan command.

- **STOP_AT_MAX**
  The counter will stop at the top of its count. For the **cbCIn32()** function, the top of the count depends on whether the **BIT_32** option is used. If it is, the top of the count is FFFFFFFF hex. If not, the top of the count is FFFF hex. By default, the counter counts upward and rolls over on the 32-bit boundary.

- **DECREMENT_ON**
  Allows the mapped channel to decrement the counter. With this option, the main counter will increment the counter, and the mapped counter can be used to decrement the counter. By default, the counter decrement option is set to "off."

  This mode is not compatible with **cbCIn()** or **cbCIn32()**. If a counter is configured for **DECREMENT_ON**, calling **cbCIn()** or **cbCIn32()** for that counter will result in a **BADCOUNTERMODE** error.

- **GATING_ON**
  Selects gating "on." When "on", the counter is enabled when the mapped channel to gate the counter is high. When the mapped counter is low, the counter is disabled but holds the count value. By default, the counter gating option is set to "off."

  This mode is not compatible with **cbCIn()** or **cbCIn32()**. If a counter is configured for **GATING_ON**, calling **cbCIn()** or **cbCIn32()** for that counter will result in a **BADCOUNTERMODE** error.
LATCH_ON_MAP  Causes the count to be latched by the signal on the mapped counter. By default, the count is latched by the internal "start of scan" signal, so the count is updated each time it’s read.

This mode is not compatible with cbCIn() or cbCIn32(). If a counter is configured for LATCH_ON_MAP, calling cbCIn() or cbCIn32() for that counter will result in a BADCOUNTERMODE error.

BIT_32  Selects a 32-bit counter. This mode affects only cbCIn32() and cbCIn(), and only when the counter is configured for STOP_AT_MAX. Recommended for use only with cbCIn32(). (Using the BIT_32 option with cbCIn() is not very useful, since the value returned by cbCIn() is only 16 bits. The effect is that the value returned by cbCIn() rolls over at 64k 65,535 times before stopping.)

ENCODER  Sets the specified counter to encoder mode.

ENCODER_MODE_X1  Sets the encoder measurement mode to X1.

ENCODER_MODE_X2  Sets the encoder measurement mode to X2.

ENCODER_MODE_X4  Sets the encoder measurement mode to X4.

LATCH_ON_Z  Selects the Encoder Z mapped signal to latch the counter outputs. This allows the user to know the exact counter value when an edge is present on another counter.

CLEAR_ON_Z_ON  Selects "clear on Z" on. The counter is cleared on the rising edge of the mapped (Z) counter. By default, the "clear on Z" option is off, and the counter is not cleared.
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 InstaCal configuration program. The specified board must have a 9513 counter. BoardNum may be 0 to 99.

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 82. 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 channel.

**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 *InstaCal* configuration program. The specified board must have a counter. BoardNum may be 0 to 99.
- **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:

- 65535 reads as -1
- 32768 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()

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 InstaCal configuration program. The specified board must have a counter. BoardNum may be 0 to 99.

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).
cbCInScan()

Scans a range of scan counter channels, and stores the samples in an array.

Function prototype:

C/C++: int cbCInScan(int BoardNum, int FirstCtr, int LastCtr, long Count, long *Rate, int MemHandle, int Options)

Visual Basic: Function cbCInScan(ByVal BoardNum&, ByVal FirstCtr&, ByVal LastCtr&, ByVal Count&, Rate&, ByVal MemHandle&, ByVal Options&) As Long

Delphi: function cbCInScan(BoardNum:Integer; FirstCtr:Integer; LastCtr:Integer; Count:Longint; var Rate:Longint; MemHandle:Integer; Options:Integer):Integer;

Arguments:

BoardNum The board number used to collect the data. BoardNum may be 0 to 99. 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 counter with scan capability.

FirstCtr First counter channel of the scan. This argument is zero-based, so the first counter number is "0".

LastCtr Last counter channel of the scan. This argument is zero-based, so the first counter number is "0".

The maximum allowable channel for both FirstCtr and LastCtr depends on how many scan counters are available on the Measurement Computing device in use.

Count The total number of counter samples to collect. If more than one channel is being sampled then the number of samples collected per channel is equal to Count / (LastCtr - FirstCtr + 1).

Rate The rate at which samples are taken in samples per second. Rate also returns the value of the actual rate set, which may be different from the requested rate because of pacer limitations.

MemHandle The handle for the Windows buffer to store data. This buffer must have been previously allocated with the cbWinBufAlloc32() function.

Options Bit fields that control various options. This field may contain any combination of non-contradictory choices in the "Options argument values" section below.

Returns:

Error code or 0 if no errors

Rate – the actual sampling rate used.

MemHandle – the collected counter data returned via the Windows buffer.

Options argument values:

BACKGROUND When the BACKGROUND option is used, control returns immediately to the next line in your program and the data collection from the counters into the buffer continues in the background. If the BACKGROUND option is not used, the cbCInScan() function does not return to your program until all of the requested data has been collected and returned to the buffer.
Use `cbGetStatus()` with `CTRFUNCTION` to check on the status of the background operation. Use `cbStopBackground()` with `CTRFUNCTION` to terminate the background process before it has completed. Execute `cbStopBackground()` 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 by using `cbStopBackground()` with `CTRFUNCTION`. Normally, you should use this option with `BACKGROUND` so that your program regains control.

**EXTTRIGGER**
If this option is specified, sampling does not begin until the trigger condition is met. You can set the trigger condition to rising edge, falling edge, or the level of the digital trigger input with the `cbSetTrigger()` function. Refer to board-specific information in the *UL User's Guide*.

**EXTCLOCK**
If this option is specified, 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 (refer to board-specific information in the *UL User's Guide*). 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.
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 InstaCal configuration program. The specified board must have a counter. BoardNum may be 0 to 99.

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()

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

Function prototype:

**C/C++:**
```c
int cbCLoad32(int BoardNum, int RegNum, unsigned long LoadValue)
```

**Visual Basic:**
```vb
Function cbCLoad32(ByVal BoardNum&, ByVal RegNum&, ByVal LoadValue&) As Long
```

**Delphi:**
```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 InstaCal configuration program. BoardNum may be 0 to 99.

- **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()

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:\Program files\Measurement Computing\DAQ by default). This data sheet is also available on our web site at www.mcedaq.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 InstaCal configuration program. The specified board must have an LS7266 counter. BoardNum may be 0 to 99.

CounterNum The number of the counter whose status bits you want to read. 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 its 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() with CTR FUNCTION is called.

Function prototype:

C/C++: int cbCStoreOnInt(int BoardNum, int IntCount, short 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 InstaCal configuration program. The specified board must have a 9513 counter. BoardNum may be 0 to 99.
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. If IntCount is non-zero, the buffer referenced by MemHandle must be of sufficient size to hold (IntCount * Number of Counters) points.

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, the buffer referenced by MemHandle must be able to hold (IntCount * Number of Counters) points.
  
  For example, if you set IntCount to 100 for a CTR-05 board, you must allocate the size of the buffer to be (100 * 5) = 500. This new functionality keeps the user application from having to move the data out of the buffer for every interrupt, before it is overwritten. Now, for each interrupt, the counter values will be stored in adjacent memory locations in the buffer.

Allocate the proper buffer size for non-zero IntCount settings

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

- If IntCount = 0, the functionality is unchanged.
cbTimerOutStart()

Starts a timer square wave output. Use cbTimerOutStop() to stop the output.

Function prototype:

C/C++: int cbTimerOutStart(int BoardNum, int TimerNum, double *Frequency)

Visual Basic: Function cbTimerOutStart(ByVal BoardNum&, ByVal TimerNum&, Frequency#) As Long

Delphi: function cbTimerOutStart(BoardNum:Integer; TimerNum:Integer; var Frequency:Double):Integer;

Arguments:

BoardNum The board number associated with the board when it was installed with the InstaCal configuration program. The specified board must have a timer-type counter. BoardNum may be 0 to 99.

TimerNum The timer to output the square wave from. Valid values are zero (0) up to the number of timers – 1 on the board.

Frequency The desired square wave frequency. The timers clock will be divided down by integer values to produce the frequency. The actual frequency output will be returned. Valid values are dependant on the timer’s clock and the timer resolution.

Returns:

Error code or 0 if no errors

Frequency – the actual frequency set.
cbTimerOutStop()

Stops a timer square wave output. Use cbTimerOutStart() to start the output.

Function prototype:

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

Arguments:

BoardNum The number associated with the board when it was installed with the InstaCal configuration program. The specified board must have a timer-type counter. BoardNum may be 0 to 99.

TimerNum The timer to stop. Valid values are zero up to the number of timers on the board – 1.

Returns:

Error code or 0 if no errors
Data Logger Functions

Introduction

This section covers Universal Library functions that read and convert data logged to a binary file from a data acquisition product equipped with data logging functionality. The data is typically logged to a CompactFlash® memory card which may then be inserted into a media reader for reading and conversion using these functions.

Data is stored in a binary file. The data may consist of analog data, digital I/O data, time stamped data and information about the device configuration. You can use the data logger functions to read this information, apply conversions to the data, and convert the files to a comma separated values (.CSV) text file or other specified text file formats.
cbLogConvertFile()

Converts a binary log file to a comma-separated values (.CSV) text file or another text file format that you specify.

Function prototype:

C/C++: int cbLogConvertFile(char* srcFile, char* destFile, int startSample, int count, int delimiter)

Arguments:

srcFile The name and path of the binary file to read.
destFile The name and destination path of the converted file. Use the file extension of the file type that you want to create.
startSample The index number of the first sample to read.
count The number of samples to read.
delimiter Specifies the character used between fields in the converted file. Set to one of the Delimiter constants. Choices are:
0 = DELIMITER_COMMA
1 = DELIMITER_SEMICOLON
2 = DELIMITER_SPACE
3 = DELIMITER_TAB

Returns: Error code or 0 if no errors.

Notes:

- Timestamp data is stored according to the TimeZone and TimeFormat arguments. Refer to cbLogSetPreferences() on page 111.
- Time stamps in the converted file may be in either 12-hour or 24-hour format based on the setting of the TimeFormat argument. Time stamps can optionally be converted to local time based on the setting of the TimeZone argument.
- AI temperature data is returned according to the Units preference. Refer to cbLogGetAIInfo() on page 97. This value is ignored if the AI data was logged as raw data.
- The Units preference is always applied to CJC data, since it is always logged as temperature data.
- If the destFile argument ends with a .CSV extension, the delimiter argument must be set to DELIMITER_COMMA. Otherwise, an INVALIDDELIMITER error is returned.
- You can open a comma-separated values text file (.CSV) directly in Microsoft Excel. Text files with extensions other than .CSV can only be imported into Excel.
cbLogGetAIChannelCount()

Retrieves the total number of analog input channels logged in a binary file.

**Function prototype:**

```
C/C++:  int cbLogGetAIChannelCount(char* Filename, int* AICount)
```

**Arguments:**

- **Filename**: The name of the file to retrieve the information from.
- **AICount**: The number of analog input channels logged in the binary file.

**Returns:**

- **Error code** or 0 if no errors.
- **AICount** – Returns the number of analog input channels logged in the binary file.
cbLogGetAIInfo()

Retrieves the channel number and unit value of each analog input channel logged in a binary file.

Function prototype:
C/C++:     int cbLogGetAIInfo(char* Filename, int* ChannelNumbers, int* Units)

Arguments:
Filename   The name of the file to retrieve the information.
ChannelNumbers An array that contains the analog input channel numbers logged in the file.
Units      An array that contains the unit values set for the device in InstaCal for each analog input channel logged in the file.

Returns:
Error code or 0 if no errors.

ChannelNumbers — Returns the analog input channel numbers logged in the binary file.

Units — Returns the unit values set for the device in InstaCal for each analog input channel logged in the binary file. Returned values include:

0 = UNITS_TEMPERATURE
1 = UNITS_RAW
cbLogGetCJCInfo()

Retrieves the number of CJC temperature channels logged in a binary file.

Function prototype:

C/C++:    int cbLogGetCJCInfo(char* Filename, int* CJCCount)

Arguments:

Filename    The name of the file to retrieve the information from.

CJCCount    The number of CJC temperature channels logged in the file.

Returns:

Error code or 0 if no errors.

CJCCount – Returns the number of CJC channels logged in the binary file.
cbLogGetDIOInfo()

Retrieves the number of digital I/O channels logged in a binary file.

**Function prototype:**

C/C++: `int cbLogGetDIOInfo(char* Filename, int* DIOCount)`

**Arguments:**

- **Filename**
  - The name of the file to retrieve the information from.
- **DIOCount**
  - The number of digital I/O channels logged in the binary file.

**Returns:**

- *Error code* or 0 if no errors.

DIOCount – Returns the number of digital I/O channels logged in the binary file.
cbLogGetFileInfo()

Retrieves the version level and byte size of a binary file.

Function prototype:

C/C++: int cbLogGetFileInfo(char* Filename, int* Version, int* Size)

Arguments:
- Filename The name of the file to retrieve the information from.
- Version The version level of the binary file.
- Size The size in bytes of the binary file.

Returns:
- Error code or 0 if no errors.
- Version – Returns the version level of the binary file.
- Size – Returns the size in bytes of the binary file.
cbLogGetFileName()

Retrieves the name of the \textit{n}th file in the directory containing binary log files.

**Function prototype:**

\begin{verbatim}
C/C++: int cbLogGetFileName(int FileNumber, char* Path, char*Filename)
\end{verbatim}

**Arguments:**

- **FileNumber**  
  Index of the file whose name you want to return. Specify one of the following:
  - The number \((n)\) that represents the location of the file in the directory (where \(n = 0, 1, 2, \) and so on), or
  - GETFIRST – get the first file in the directory, or
  - GETNEXT – get the next file in the directory, based on the current index.
  
  This parameter is the index of the file in the directory, and is not part of the filename.

- **Path**  
  The full path to the directory containing the binary file. The path must be NULL terminated and cannot be longer than 256 characters.

- **Filename**  
  A NULL terminated string containing the full path to the file.

**Returns:**

- Error code or 0 if no errors.
- **Filename** – Returns a NULL terminated string containing the full path to the file.

**Notes:**

Set \texttt{FileNumber} to \texttt{GETFIRST} to access the first binary file in a directory. Subsequent calls with \texttt{FileNumber = GETNEXT} returns each successive file in the directory. When you call the function after accessing the last file in the directory, the function returns the error code \texttt{NOMOREFILES}.  


cbLogGetPreferences()

Retrieves API preference settings for time stamped data, analog temperature data, and CJC temperature data. Returns the default values unless changed using cbLogSetPreferences().

Function prototype:

C/C++: int cbLogGetPreferences(int* TimeFormat, int* TimeZone, int* Units)

Arguments:

TimeFormat The time format to apply to time stamped data. Set to one of the TimeFormat constants. Choices are:
0 = TIMEFORMAT_12HOUR - for example 2:32:51PM.
1 = TIMEFORMAT_24HOUR - for example 14:32:51.
TimeZone The time zone to store time stamped data. Set to one of the TimeZone constants. Choices are:
0 = TIMEZONE_LOCAL. Converts time stamped data to the local time zone on your computer.
1 = TIMEZONE_GMT. Leaves time stamped data in Greenwich Mean Time.
Units The unit to use for temperature data. Set to one of the Units constants. Choices are:
0 = FAHRENHEIT
1 = CELSIUS
2 = KELVIN

Returns:

Error code or 0 if no errors.

TimeFormat – Returns the format to apply to time stamped data from API functions that return time data.
TimeZone – Returns the time zone to apply to time stamped data from API functions that return time data.
Units – Returns the unit to use when converting temperature data from API functions that return temperature data.
cbLogGetSampleInfo()

Retrieves the sample interval, sample count, and the date and time of the first data point contained in a binary file.

Function prototype:

```
C/C++: int cbLogGetSampleInfo(char* Filename, int* SampleInterval, int* SampleCount, int* StartDate, int* StartTime)
```

Arguments:

- **Filename**
  The name of the file to retrieve sample information from.

- **SampleInterval**
  The time interval, in seconds, between samples.

- **SampleCount**
  The number of samples contained in the file.

- **StartDate**
  The date when the first data point was logged in the file. Date values are packed in the following format:
  - Byte 0: day
  - Byte 1: month
  - Byte 2 - 3: year

- **StartTime**
  The time when the first data point was logged in the file. Time values are packed in the following format:
  - Byte 0: seconds
  - Byte 1: minutes
  - Byte 2: hours
  - Byte 3: 0xff = 24hour format
  - 0x0 = AM
  - 0x1 = PM

Returns:

- **Error code** or 0 if no errors.

  - **SampleInterval** – Returns the time interval, in seconds, between samples.
  - **SampleCount** – Returns the number of samples in the file.
  - **StartDate** – Returns the date when the first data point was logged in the file.
  - **StartTime** – Returns the time when the first data point was logged in the file.

Notes:

Time stamped data is returned according to the TimeZone and TimeFormat preferences. Refer to `cbLogSetPreferences()` on page 111.
cbLogReadAIChannels()

Retrieves analog input data from a binary file, and stores the values in an array.

**Function prototype:**

```c
C/C++: int cbLogReadAIChannels(char* Filename, int StartSample, int Count, float* AIChannels)
```

**Arguments:**

- `Filename` The name of the file to retrieve the information from.
- `StartSample` The first sample to read from the binary file.
- `Count` The number of samples to read from the binary file.
- `AIChannels` Receives the analog input values.

**Returns:**

- Error code or 0 if no errors.
- `AIChannels` – Returns the analog input values logged in the file.

**Notes:**

The units of the analog input data that is returned is set by the value of the Units preference. Refer to `cbLogSetPreferences()` on page 111.

The units preference is only applied if the logged data is temperature data. This value is ignored if the data logged is raw.

**Analog array:**

The user is responsible for allocating the size of the analog array, and ensuring that it is large enough to hold the data that will be returned. You can calculate the array allocation using the `SampleCount` value from `cbLogGetSampleInfo()`, and the `AICount` value from `cbLogGetAIChannelCount()`:

```c
float* aiChannels = new float[SampleCount * AICount];
```

The figure below shows the layout of the analog array, and how the elements should be indexed.

<table>
<thead>
<tr>
<th>CH0</th>
<th>CH1</th>
<th>...</th>
<th>CHn</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH0</td>
<td>CH1</td>
<td>...</td>
<td>CHn</td>
</tr>
<tr>
<td>CH0</td>
<td>CH1</td>
<td>...</td>
<td>CHn</td>
</tr>
<tr>
<td>CH0</td>
<td>CH1</td>
<td>...</td>
<td>CHn</td>
</tr>
</tbody>
</table>

Where `n` is `(AICount - 1)`. CH0 – CHn refer to the channels in the array, not the input channels of the device.

For example, assume that all of the even number input channels are logged. The analog array channels are mapped as shown here:

<table>
<thead>
<tr>
<th>Array Channel</th>
<th>Device Input Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Use the following code fragment to access the elements of the analog array:
for (i=0; i<numberOfSamples; i++)
{
    for (j=0; j<numberOfAIChannels; j++)
    {
        a = analogArray[(i *numberOfAIChannels) + j];
    }
}

where

the numberOfSamples is set by the SampleCount value from cbLogGetSampleInfo().

the numberOfAIChannels is set by the AICount value from cbLogGetAIChannelCount().
cbLogReadCJCChannels()

Retrieves CJC temperature data from a binary file, and stores the values in an array.

Function prototype:

```c
int cbLogReadCJCChannels(char* Filename, int StartSample, int Count, float* CJCChannels)
```

Arguments:

- **Filename**: The name of the file to retrieve the information from.
- **StartSample**: The first sample to read from the binary file.
- **Count**: The number of samples to read from the binary file.
- **CJCChannels**: Receives the CJC temperature values.

Returns:

- **Error code** or 0 if no errors.
- **CJCChannels** – Returns the CJC temperature values logged in the file.

Notes:

The temperature scale of the CJC data that is returned is set by the value of the **Units** preference. Refer to `cbLogSetPreferences()` on page 111.

**CJC array:**

The user is responsible for allocating the size of the CJC array, and ensuring that it is large enough to hold the data that will be returned. You can calculate the array allocation using the **SampleCount** value from `cbLogGetSampleInfo()`, and the **CJCCount** value from `cbLogGetCJCInfo()`:

```c
float* CJCChannels = new float[SampleCount * CJCCount];
```

The figure below shows the layout of the CJC array, and how the elements should be indexed.

Where \( n = (\text{CJCCount} - 1) \)
Use the following code fragment to access the elements of the CJC array.

```c
for (i=0; i<numberOfSamples; i++)
    {
    for (j=0; j<numberofCJCChannels; j++)
    {
        c = cjcArray[(i * numberofCJCChannels) + j];
    }
    }
```

where

- the `numberOfSamples` is set by the `SampleCount` value from `cbLogGetSampleInfo()`.
- the `numberofCJCChannels` is set by the `CJCCount` value from `cbLogGetCJCInfo()`.
cbLogReadDIOChannels()

Retrieves digital I/O channel data from a binary file, and stores the values in an array.

**Function prototype:**

\[
\text{int cbLogReadDIOChannels(char* Filename, int StartSample, int Count, int* DIOChannels)}
\]

**Arguments:**

- **Filename**
  - The name of the file to retrieve the information from.
- **StartSample**
  - The first sample to read from the binary file.
- **Count**
  - The number of samples to read from the binary file.
- **DIOChannels**
  - Receives the DIO input values.

**Returns:**

- **Error code** or 0 if no errors.

**DIOChannels** – Returns the DIO channel values logged in the file. Each element of the array contains the value of one bit from a digital channel.

**DIO array:**

The user is responsible for allocating the size of the DIO array, and ensuring that it is large enough to hold the data that will be returned. You can calculate the array allocation using the `SampleCount` value from `cbLogGetSampleInfo()` and the `DIOCount` value from `cbLogGetDIOInfo()`:

\[
\text{int* DIOChannels = new int[SampleCount * DIOCount];}
\]

The figure below shows the layout of the DIO array, and how the elements should be indexed.

- \(n\) is \((\text{DIOCount} - 1)\)

Use the following code fragment to access the elements of the DIO array:

\[
\text{for (i=0; i<numberOfSamples; i++)}
\]

\[
\text{\{ for (j=0; j<numberOfDIOChannels; j++)}
\]

\[
\text{\{ d = dioArray[(i * numberOfDIOChannels) + j];}
\]

\[
\text{\}}
\]

where

- `numberOfSamples` is set by the `SampleCount` value from `cbLogGetSampleInfo()`
- `numberOfDIOChannels` is set by the `DIOCount` value from `cbLogGetDIOInfo()`
cbLogReadTimeTags()

Retrieves date and time values logged in a binary file. This function stores date values in the DateTags array, and time values in the TimeTags array.

**Function prototype:**

```c
int cbLogReadTimeTags(char* Filename, int StartSample,
                      int Count, int* DateTags, int* TimeTags)
```

**Arguments:**

- **Filename**
  - The name of the file to retrieve the information from.
- **StartSample**
  - The first sample to read from the binary file.
- **Count**
  - The number of samples to read from the binary file.
- **DateTags**
  - Receives the date value for each sample logged in the file. The dates are packed in the following format:
    - Byte 0: day
    - Byte 1: month
    - Byte 2-3: year
- **TimeTags**
  - Receives the time value for each sample logged in the file. The times are packed in the following format:
    - Byte 0: seconds
    - Byte 1: minutes
    - Byte 2: hours
    - Byte 3: 0xff = 24-hour format, 0x0 = AM, 0x1 = PM

**Returns:**

- **Error code** or 0 if no errors.
- **DateTags** – Returns the date values for each sample logged in the file.
- **TimeTags** – Returns the time values for each sample logged in the file.

**Notes:**

Time stamped data is returned according to the TimeZone preference value and the TimeFormat preference value. Refer to `cbLogSetPreferences()` on page 111.

Time stamped data are logged in the file if InstaCal is configured to do so. If time stamps are not logged, the TimeTags and DateTags arrays are filled with values calculated from the file header information.

**Array size:**

The user is responsible for allocating the size of the DateTags and TimeTags arrays, and ensuring that they are large enough to hold the data that is returned. You can calculate the array allocation using the SampleCount value from `cbLogGetSampleInfo()` on page 102.

```c
int* dates = new int[SampleCount];
int* times = new int[SampleCount];
```
Data Logger Functions

cbLogReadTimeTags()

DateTags array

The figure below shows the layout of the DateTags array, and how the elements should be indexed.

| D0 | D1 | D2 | ... | Dn |

where: \( n = (\text{SampleCount} - 1) \)

Each sample has only one date. Use the following code fragment to access the elements of the DateTags array:

```c
for (i=0; i<\text{numberOfSamples}; i++)
{
    d = \text{DateTagsArray}[i];
}
```

TimeTags array

The figure below shows the layout of the TimeTags array, and how the elements should be indexed.

| T0 | T1 | T2 | ... | Tn |

where: \( n = (\text{SampleCount} - 1) \)

Each sample has only one time stamp. Use the following code fragment to access the elements of the TimeTags array:

```c
for (i=0; i<\text{numberOfSamples}; i++)
{
    t = \text{TimeTagsArray}[i];
}
```
cbLogSetPreferences()

Sets preferences for returned time stamped data, analog temperature data, and CJC temperature data.

Function prototype:

C/C++: int cbLogSetPreferences(int TimeFormat, int TimeZone, int Units)

Arguments:

- **TimeFormat**: Specifies the time format to apply when returning time stamped data (when using cbLogReadTimeTags() for example). Set to one of the TimeFormat constants. Choices are:
  - 0 = TIMEFORMAT_12HOUR - for example 2:32:51 PM (default).
  - 1 = TIMEFORMAT_24HOUR - for example 14:32:51.

- **TimeZone**: Specifies whether to convert time stamped data that is returned (when using cbLogReadTimeTags() for example) to the local time zone or to return the time stamps as they are stored in the file (in the GMT time zone). Set to one of the TimeZone constants. Choices are:
  - 0 = TIMEZONE_LOCAL. Converts timestamp data to the local time zone on your computer (default).
  - 1 = TIMEZONE_GMT. Leaves time stamped data in Greenwich Mean Time.

- **Units**: Specifies whether to convert temperature data returned (when using cbLogReadAIChannels() for example) to Fahrenheit or Kelvin, or return temperature data as they are stored in the file (in Celsius units). Set to one of the Units constants. Choices are:
  - 0 = FAHRENHEIT (Default)
  - 1 = CELSIUS
  - 2 = KELVIN

This value is ignored if raw data is logged.

Returns:

Error code or 0 if no errors.

Notes:

- The TimeFormat and TimeZone preferences are applied to all time data returned using API functions that return time data.
- The Units preference specifies the temperature scale that the API applies when reading and converting analog temperature and CJC data.
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.
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 InstaCal configuration program. BoardNum may be 0 to 99.

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  Specifies the bit number within the single large port.

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 InstaCal configuration program. BoardNum may be 0 to 99.

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 Specifies the bit number within the single large port. The specified bit must be in a port that is currently configured as an output.

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 InstaCal configuration program. BoardNum may be 0 to 99.

PortType The port (AUXPORT) whose bits are to be configured. The port specified must be bitwise configurable. Check the board-specific information in the Universal Library User's Guide 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 InstaCal configuration program. BoardNum may be 0 to 99.

PortNum The specified port must be configurable. For most boards, AUXPORT is not configurable. Check the board-specific information in the Universal Library User’s Guide for details.

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 InstaCal configuration program. BoardNum may be 0 to 99.

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.

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.mccdqa.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 InstaCal configuration program. BoardNum may be 0 to 99.

PortNum Specifies which digital I/O port to read (usually FIRSTPORTA or FIRSTPORTB).

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. 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 the 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 MemHandle.

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 MemHandle will continue in the background. Use cbGetStatus() with DIFUNCTION to check on the status of the background operation. Use cbStopBackground() with DIFUNCTION 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() with DIFUNCTION. 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 information contained in the *Universal Library Users Guide*). When this option is used the Rate argument is ignored. The transfer rate is dependent on the trigger signal.

**EXTTRIGGER**

If this option is used then the scan will not begin until the signal on the trigger input line meets the trigger criteria.

**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 the buffer. When WORDXFER is used, it is generally required to set PortNum to FIRSTPORTA.

**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 InstaCal configuration program. BoardNum may be 0 to 99.

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 (for example the DAS1600 Series). Set PortNum to either FIRSTPORTA or AUXPORT depending on the digital port you want to set.

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. Refer to the example programs and the board-specific information in the Universal Library User's Guide for clarification of valid PortNum values.
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++:
```c
int cbDOutScan(int BoardNum, int PortNum, long Count, long *Rate,
               int MemHandle, int Options)
```

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

Delphi:
```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 InstaCal configuration program. BoardNum may be 0 to 99.
PortNum  Specifies which digital I/O port to write (usually 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. 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 MemHandle will continue in the background. Use cbGetStatus() with DOFUNCTION to check on the status of the background operation. Use cbStopBackground() with DOFUNCTION 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() with DOFUNCTION. 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 information contained in the Universal Library Users Guide). 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. When WORDXFER is used, it is generally required to set PortNum to FIRSTPORTA.

NONSTREAMEDIO

When this option is used, you can output non-streamed data to a specific DAC output channel.

To load the data output buffer into the device’s internal output FIFO, the aggregate size of the data output buffer must be \( \leq \) the size of the internal data output FIFO in the device. Once the sample data are transferred or downloaded to the device, the device is responsible for outputting the data. You can’t make any changes to the output buffer once the output begins.

With NONSTREAMEDIO mode, you do not have to periodically feed output data through the program to the device for the data output to continue. However, the size of the buffer is limited.

ADCCLOCKTRIG

Triggers a data output operation when the ADC clock starts.

ADCCLOCK

Paces the data output operation using the ADC clock.

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.
- **NONSTREAMEDIO** can only be used with the number of samples (Count) set equal to the size of the FIFO or less.
- Transfer method may not be specified. DMA is used.
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 401 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 is 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 124 for an alternate method of handling errors.
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++:
```c
int cbMemRead(int BoardNum, unsigned short DataBuffer[], long FirstPoint, long Count)
```

Visual Basic:
```vb
Function cbMemRead(ByVal BoardNum&, DataBuffer%, ByVal FirstPoint&, ByVal Count&) As Long
```

Delphi:
```delphi
function cbMemRead(BoardNum:Integer; var DataBuffer:Word; FirstPoint:Longint; Count:Longint):Integer;
```

Arguments:

- **BoardNum**
  - The number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.

- **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:

```c
cbMemRead (0, DataBuffer, 0, 100000)
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 number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.
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) As Long
Delphi: function cbMemReset(BoardNum:Integer):Integer;

Arguments:

BoardNum The number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.

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++:**
  ```c
  int cbMemSetDTMode(int BoardNum, int Mode)
  ```

- **Visual Basic:**
  ```vb
  Function cbMemSetDTMode(ByVal BoardNum&, ByVal Mode&) As Long
  ```

- **Delphi:**
  ```delphi
  function cbMemSetDTMode(BoardNum:Integer; Mode:Integer):Integer;
  ```

**Arguments:**

- **BoardNum**
  The number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.

- **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 number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.

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 DTCOMNECT + 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
Initialize 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:
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() 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++:

```c
int cbGetRevision(float* DLLRevNum, float* VXDRevNum);
```

Visual Basic:

```vb
Function cbGetRevision(DLLRevNum!, VXDRevNum!) As Long
```

Delphi:

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

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

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

**Arguments:**

- **BoardNum**  
The number associated with the board when it was installed with the InstaCal configuration program. The specified board must have an A/D. BoardNum may be 0 to 99.

- **LowChan**  
The first A/D channel in the scan range.

- **HighChan**  
The last A/D channel in the scan range.

  The maximum allowable channel number 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**  
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**  
The 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 the data is stored. If the file does not exist, it will be created.

- **Options**  
Bit fields that control various options. Refer to the constants in the "Options argument values" section on page 138.

**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 (refer to board-specific information contained in the *Universal Library Users Guide*). 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.

---

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 in the *Universal Library User’s Guide* to determine if your board supports cbFileAInScan() and/or cbFilePretrig().

**Function prototype:**

**C/C++:**

```c
int cbFileGetInfo(char *FileName, short *LowChan, short *HighChan,
long *PretrigCount, long *TotalCount, long *Rate, int *Range)
```

**Visual Basic:**

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

**Delphi:**

```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 number associated with the board when it was installed with the *InstaCal* configuration program. The specified board must have an A/D and pretrigger capability. BoardNum may be 0 to 99.

**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.

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 (refer to board-specific information contained in the Universal Library Users Guide). 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. When cbFileAInScan() or cbFilePretrig() fills the streamer file, this function returns the content of that file. Refer to information on your board in the Universal Library User's Guide to determine if your board supports cbFileAInScan() and/or cbFilePreTrig().

Function prototype:

C/C++: int cbFileRead(char *FileName, long FirstPoint, long *NumPoints, int *DataBuffer)

Visual Basic: Function cbFileRead(ByVal FileName$, ByVal FirstPoint$, NumPoints $, 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
NumPoints 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.
NumPoints - number of points actually read.
NumPoints 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 NumPoints 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.
Synchronous I/O Functions

Introduction

Use the functions discussed in this chapter to synchronously read and write data from analog channels, counter channels, thermocouple channels, and digital ports. These functions can be used with hardware equipped with synchronous input and output capability.
cbDaqInScan()

Scans analog, digital, counter, and temperature input channels synchronously, and stores the samples in an array. This function only works with boards that support synchronous input.

**Function prototype:**

**C/C++:**

```c
int cbDaqInScan(int BoardNum, short ChanArray[], short ChanTypeArray[], short GainArray[], int ChanCount, long* Rate, long *PretrigCount, long *TotalCount, int MemHandle, int Options);
```

**Visual Basic:**

```vba
Function cbDaqInScan(ByVal BoardNum&, ChanArray%, ChanTypeArray%, GainArray%, ByVal ChanCount&), CBRate&, PretrigCount&, CBCount&, ByVal MemHandle&, ByVal Options&) As Long
```

**Delphi:**

```delphi
function cbDaqInScan(BoardNum:Integer; var ChanArray:SmallInt; var ChanTypeArray:SmallInt; var GainArray:SmallInt; ChanCount:Integer; var Rate:LongInt; var PretrigCount:LongInt; var TotalCount:LongInt; MemHandle:Integer; Options:Integer):Integer;
```

**Arguments:**

- **BoardNum**
  The board number used to collect the data. BoardNum may be 0 to 99. Refers to the number associated with the board used to collect the data when it was installed with the InstaCal configuration program. The board must support synchronous input.

- **ChanArray**
  Array containing channel values. Valid channel values are analog input channels, digital ports, counter input channels, and temperature input channels of the device.

- **ChanTypeArray**
  Array containing channel types. Each element of this array defines the type of the corresponding element in the ChanArray. Set to one of the constants in the "ChanTypeArray argument values" section on page 145.

- **GainArray**
  Array containing A/D range codes. If the corresponding element in the ChanArray is not an analog input channel, the range code for this channel is ignored.

- **ChanCount**
  Number of elements in each of the three arrays - ChanArray, ChanTypeArray and GainArray.

- **Rate**
  The sample rate at which samples are acquired, in samples per second per channel. Rate also returns the value of the actual rate set, which may be different from the requested rate because of pacer limitations.

- **PretrigCount**
  Sets the number of pre-trigger samples to collect. Specifies the number of samples to collect before the trigger occurs. This function won't run in pre-trigger mode if PreTrigCount is **set to zero**. PreTrigCount is **ignored** if the EXTRIGGER option is not specified.

- **PretrigCount** also returns the value of the actual pre-trigger count set, which may be different from the requested pre-trigger count because pre-trigger count must be a multiple of ChanCount.

- **PreTrigCount** must be evenly divisible by the number of channels being scanned (ChanCount). If it is not, this function adjusts the number (up) to the next valid value, and returns that value to the PreTrigCount argument.

- **TotalCount**
  Total number of samples to collect. Specifies the total number of samples to collect and store in the buffer. TotalCount **must be greater than PretrigCount**.

- **TotalCount** also returns the value of the actual total count set, which may be different from the requested total count, because total count must be a multiple of ChanCount.
cbDaqInScan()  

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

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

Options  
Bit fields that control various options. This field may contain any combination of non-contradictory choices in the "Options argument values" section below.

ChanTypeArray argument values:
- ANALOG  
  Analog input channel.
- DIGITAL8  
  8-bit digital input port.
- DIGITAL16  
  16-bit digital input port. (FIRSTPORTA only)
- CTR16  
  16-bit counter.
- CTR32LOW  
  Lower 16-bits of a 32-bit counter.
- CTR32HIGH  
  Upper 16-bits of a 32-bit counter.
- CJC  
  CJC channel.
- TC  
  Thermocouple channel.

The cbGetTCValues() function can be used to convert raw thermocouple data to data on a temperature scale (Celsius, Fahrenheit or Kelvin). Note: If at least one TC channel is listed in the channel array, and averaging is enabled for that channel, the averaging will be applied to all of the channels listed in the channel array.

SETPOINTSTATUS  
The setpoint status register. This is a bitfield indicating the state of each of the setpoints. A "1" indicates that the setpoint criteria has been met.

ChanTypeArray flag values:
- SETPOINT_ENABLE  
  Enables a setpoint. When this option is specified, it must be OR'ed with the ChanTypeArray argument values.
  You set the setpoint criteria with the cbDaqSetSetpoints() function. The number of channels set with the SETPOINT_ENABLE flag must match the number of setpoints set by the cbDaqSetSetpoints() function's SetpointCount argument.

Options argument values:
- BACKGROUND  
  When the BACKGROUND option is used, control returns immediately to the next line in your program and the data collection into the buffer continues in the background. If the BACKGROUND option is not used, the cbDaqInScan() function does not return control to your program until all of the requested data has been collected and returned to the buffer.
  Use cbGetStatus() with DAQIFUNCTION to check on the status of the background operation. Use cbStopBackground() with DAQIFUNCTION to terminate the background process before it has completed. Execute cbStopBackground() 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 by using cbStopBackground() with DAQIFUNCTION. Normally this option should be used in combination with BACKGROUND so that your program will regain control.
EXTCLOCK

If this option is used, 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. 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.

EXTTRIGGER

If this option is specified, the sampling will not begin until the trigger condition is met (refer to the cbDaqSetTrigger() function).

Returns:

Error code or 0 if no errors

Rate – Actual sampling rate used.

PreTrigCount – Actual pre-trigger count used.

TotalCount – Actual total count used.

MemHandle - Collected data returned via the Windows buffer.
cbDaqOutScan()

Outputs values synchronously to analog output channels and digital output ports. This function only works with boards that support synchronous output.

Function prototype:

C/C++:
```c
int cbDaqOutScan(int BoardNum, short ChanArray[], short ChanTypeArray[], short GainArray[], int ChanCount, long* Rate, long Count, int MemHandle, int Options);
```

Visual Basic:
```vb
Function cbDaqOutScan(ByVal BoardNum&, ChanArray%, ChanTypeArray%, GainArray%, ByVal ChanCount&, CBRate&), ByVal CBCount&, ByVal MemHandle&, ByVal Options&) As Long
```

Delphi:
```delphi
function cbDaqOutScan(BoardNum:Integer; var ChanArray:SmallInt; var ChanTypeArray:SmallInt; var GainArray:SmallInt; ChanCount:Integer; var Rate:LongInt; Count:LongInt; MemHandle:Integer; Options:Integer):Integer;
```

Arguments:

- **BoardNum**
  The board number used to collect the data. BoardNum may be 0 to 99. Refers to the number associated with the board used to collect the data when it was installed with the InstaCal configuration program. The board must support synchronous output.

- **ChanArray**
  Array containing channel values. Valid channel values are analog output channels and digital ports.

- **ChanTypeArray**
  Array containing channel types. Each element of this array defines the type of the corresponding element in the ChanArray. Choices are:
  - **ANALOG**: Analog output channel.
  - **DIGITAL16**: 16-bit digital output port. (FIRSTPORTA only)

- **GainArray**
  Array containing D/A range codes. If the corresponding element in the ChanArray is not an analog output channel, the range code for this channel is ignored. If the board does not have programmable gain, this parameter is ignored, and therefore can be set to null.

- **ChanCount**
  Number of elements in each of the three arrays - ChanArray, ChanTypeArray and GainArray.

- **Rate**
  Sample rate in scans per second. Rate also returns the value of the actual rate set, which may be different from the requested rate because of pacer limitations.

- **Count**
  Sets the total number of values to output. Count also returns the value of the actual count set, which may be different from the requested total count because count must be a multiple of the channel count.

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

- **Options**
  Bit fields that control various options. This field may contain any combination of non-contradictory choices in the "Options argument values" section on page 148.
Synchronous I/O Functions

Options argument values:

- **BACKGROUND**
  When this option is used the output operations will begin running in the background and control will immediately return to the next line of your program. Use `cbGetStatus()` with the DAQOFUNCTION option to check the status of background operation. Use `cbStopBackground()` with DAQOFUNCTION to terminate background operations before they are completed. Execute `cbStopBackground()` with DAQOFUNCTION 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 outputs the specified number (Count) of output values, it resets to the start of the buffer and begins again. The only way to stop this operation is by calling `cbStopBackground()` with DAQOFUNCTION. This option should only be used in combination with BACKGROUND so that your program can regain control.

- **EXTCLOCK**
  If this option is used, 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.

  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.

- **ADCCLOCKTRIG**
  If this option is used, the data output operation will be triggered upon the start of the ADC clock.

- **ADCCLOCK**
  When this option is specified, the data output operation will be paced by the ADC clock.

- **NONSTREAMEDIO**
  This option allows non-streamed data output to be generated to a specified output channel.

  In this mode, the aggregate size of data output buffer must be less than or equal to the size of the internal data output FIFO on the Measurement Computing device. This allows the data output buffer to be loaded into the device’s internal output FIFO.

  Once the sample updates are transferred (or downloaded) to the device, the device is responsible for outputting the data. While the size is limited, and the output buffer cannot be changed once the output is started, this mode has the advantage being able to continue data output without having to periodically feed output data through the program to the device.

Returns:

- **Error code** or 0 if no errors

- **Rate** – Actual sampling rate used.
cbDaqSetSetpoints()  

Configures up to 16 detection setpoints associated with the input channels within a scan group. This function only works with boards that support synchronous input.

Function Prototype:

C/C++:

```c
int cbDaqSetSetpoints(int BoardNum, float *LimitAArray, float *LimitBArray, float *reserved, int *SetpointFlagsArray, int *SetpointOutputArray, float *Output1Array, float *Output2Array, float *OutputMask1Array, float *OutputMask2Array, int SetpointCount);
```

Visual Basic:

```vb
Function cbDaqSetSetpoints(ByVal BoardNum&, LimitAArray!, LimitBArray!, Reserved!, SetpointFlagsArray!, SetpointOutputArray!, Output1Array!, Output2Array!, OutputMask1Array!, OutputMask2Array!, ByVal SetpointCount&) As Long
```

Delphi:

```delphi
function cbDaqSetSetpoints(BoardNum:Integer; var LimitAArray:Single; var LimitBArray:Single; var Reserved:Single; var SetpointFlagsArray:Integer; var SetpointOutputArray:Integer; var Output1Array:Single; var Output2Array:Single; var OutputMask1Array:Single; var OutputMask2Array:Single; SetpointCount:Integer):Integer;
```

Arguments:

- **BoardNum**
  - The board number used to collect the data. BoardNum may be 0 to 99. 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 support synchronous input.

- **LimitAArray**
  - Array containing the limit A values for the input channels used for the setpoint. Limit A specifies a value used to determine if the setpoint criteria are met.

- **LimitBArray**
  - Array containing the limit B values for the input channels used for the setpoint. Limit B specifies a value used to determine if the setpoint criteria are met.

- **Reserved**
  - Reserved for future use.

- **SetpointFlagsArray**
  - Array containing the setpoint flags. Set to one of the constants in the "SetpointFlagsArray argument values" section below.

- **SetpointOutputArray**
  - Array containing output sources. Set to one of the constants in the "SetpointOutputArray argument values" section on page 150.

- **Output1Array**
  - Array containing the values for the output channels used for the setpoint.

- **Output2Array**
  - Array containing the values for the output channels used for the setpoint.

- **OutputMask1Array**
  - Array containing the output masks for output value 1 – for FIRSTPORTC only.

- **OutputMask2Array**
  - Array containing the output masks for output value 2 – for FIRSTPORTC only.

- **SetpointCount**
  - Number of setpoints to configure (0 -16). Set to 0 to disable the setpoints.

**SetpointFlagsArray argument values:**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF_EQUAL_LIMITA</td>
<td>Setpoint criteria: The input channel = limit A.</td>
</tr>
<tr>
<td>SF_LESSTHAN_LIMITA</td>
<td>Setpoint criteria: The input channel &lt; limit A.</td>
</tr>
<tr>
<td>SF_GREATERTHAN_LIMITB</td>
<td>Setpoint criteria: The input channel &gt; limit B.</td>
</tr>
</tbody>
</table>
Synchronous I/O Functions

Synchronous I/O Functions

**cbDaqSetSetpoints()**

- **SF_INSIDE_LIMITS**
  - Setpoint criteria: The input channel > limit A and < limit B.

- **SF_OUTSIDE_LIMITS**
  - Setpoint criteria: The input channel < limit A and > limit B.

- **SF_HYSTERESIS**
  - Setpoint criteria: If the input channel > limit A then output value 1. If the input channel < limit B then output value 2.

- **SF_UPDATEON_TRUEONLY**
  - If the criteria is met then output value 1.

- **SF_UPDATEON_TRUEANDFALSE**
  - If the criteria is met then output value 1, else output value 2.

**SetpointOutputArray argument values:**

<table>
<thead>
<tr>
<th>Output Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SO_NONE</td>
<td>Perform no outputs.</td>
</tr>
<tr>
<td>SO_FIRSTPORTC</td>
<td>Output to FIRSTPORTC when the criteria is met.</td>
</tr>
<tr>
<td>SO_DAC0</td>
<td>Output to DAC0 when the criteria is met. You must have a device with DAC0.</td>
</tr>
<tr>
<td>SO_DAC1</td>
<td>Output to DAC1 when the criteria is met. You must have a device with DAC1.</td>
</tr>
<tr>
<td>SO_DAC2</td>
<td>Output to DAC2 when the criteria is met. You must have a device with DAC2.</td>
</tr>
<tr>
<td>SO_DAC3</td>
<td>Output to DAC3 when the criteria is met. You must have a device with DAC3.</td>
</tr>
<tr>
<td>SO_TMR0</td>
<td>Output to timer 0 when the criteria is met.</td>
</tr>
<tr>
<td>SO_TMR1</td>
<td>Output to timer 1 when the criteria is met.</td>
</tr>
</tbody>
</table>

**Returns:**

Error code or 0 if no errors
cbDaqSetTrigger()

Selects the trigger source and sets up its parameters. This trigger is used to initiate or terminate an acquisition using the cbDaqInScan() function if the EXTTRIGGER option is selected. This function only works with boards that support synchronous output.

Function prototype:

C/C++:
```c
int cbDaqSetTrigger(int BoardNum, int TrigSource, int TrigSense, int TrigChan, int ChanType, int Gain, float Level, float Variance, int TrigEvent);
```

Visual Basic:
```vb
Function cbDaqSetTrigger(ByVal BoardNum&, ByVal TrigSource&, ByVal TrigSense&, ByVal TrigChan&, ByVal ChanType&, ByVal Gain&, ByVal Level!, ByVal Variance!, ByVal TrigEvent&) As Long
```

Delphi:
```delphi
function cbDaqSetTrigger(BoardNum:Integer; TrigSource:Integer; TrigSense:Integer; TrigChan:Integer; ChanType:Integer; Gain:Integer; Level:Single; Variance:Single; TrigEvent:Integer):Integer;
```

Arguments:

`BoardNum` The board number used to collect the data. BoardNum may be 0 to 99. Refers to the number associated with the board used to collect the data when it was installed with the InstaCal configuration program. The board must support synchronous output.

`TrigSource` Specifies the type of triggering based on the external trigger source. Set to one of the constants specified in "TrigSource argument values" section on page 152.

`TrigSense` Specifies the trigger sensitivity. The trigger sensitivity normally defines the way in which a trigger event is detected based upon the characteristics of the trigger input signal. However, it often defines the way in which the trigger input signal(s) should be compared to the trigger level parameter value. Set to one of the constants specified in "TrigSense argument values" section on page 152.

`TrigChan` Specifies the trigger channel. The trigger channel must be a configured channel in the channel array (refer to cbDaqInScan()).

`ChanType` Specifies the channel type and should match the channel type setting for the trigger channel configured using the cbDaqInScan() function.

`Gain` Specifies the trigger channel gain code. If the device has programmable gain, this argument should match the gain code setting when the channel is configured using the cbDaqInScan() function. The Gain parameter is ignored if TrigChan is not an analog channel.

`Level` A single precision floating point value which represents, in engineering units, the level at or around which the trigger event should be detected.

This option is used for trigger types that depend on an input channel comparison to detect the start trigger or stop trigger event.

The actual level at which the trigger event is detected depends upon trigger sensing and variability. Refer to Trigger levels on page 153 for more information.

`Variance` A single-precision floating point value which represents, in engineering units, the amount that the trigger event can vary from the Level parameter.

While the TrigSense parameter indicates the direction of the input signal relative to the Level parameter, the Variance parameter specifies the degree to which the input signal can vary relative to the Level parameter.
Synchronous I/O Functions

cbDaqSetTrigger()

TrigEvent Specifies the trigger event type. Valid values indicate either a start trigger event (START_EVENT) or a stop trigger event (STOP_EVENT).

START_EVENT: The start trigger event defines the conditions under which post-trigger acquisition data collection should be initiated or triggered. The start trigger event can vary in complexity from starting immediately, to starting on complex channel value definitions.

STOP EVENT: The stop trigger event signals the current data acquisition process to terminate. The stop event can be as simple as that of a scan count, or as complex as involving a channel value level condition.

TrigSource argument values:

TRIG_IMMEDIATE Start trigger event only. Acquisition begins immediately upon invocation the cbDaqInScan function. No pre-trigger data acquisition is possible with this trigger type.

TRIG_EXTTTL Start trigger event only. Acquisition begins on the selectable edge of an external TTL signal. No pre-trigger data acquisition is possible with this trigger type.

TRIG_ANALOGHW Start trigger event only. Data acquisition begins upon a selectable criteria of the input signal (above level, below level, rising edge, etc.) TrigChan must be defined as the first channel in the channel scan group. No pre-trigger data acquisition is possible with this trigger type.

TRIG_ANALOGSW Post-trigger data acquisition begins upon a selectable criteria of the input signal (above level, below level, rising edge, etc.)

TRIG_DIGPATTERN Post-trigger data acquisition beings upon receiving a specified digital pattern on the specified digital port.

TRIG_COUNTER Post-trigger data acquisition begins upon detection of specified counter criteria.

TRIG_SCANCOUNT Stop trigger event only. Stops collecting post-trigger data when the specified number of post-trigger scans are completed.

TrigSense argument values:

RISING_EDGE: Triggers when the signal goes from low to high (TTL trigger) or rises through a specified level (hardware analog, software analog, and counter).

FALLING_EDGE: Triggers when the signal goes from high to low (TTL trigger) or falls through a specified level (hardware analog, software analog, and counter).

ABOVE_LEVEL: Triggers when the signal is above a specified level (hardware analog, software analog, counter, and digital pattern).

BELOW_LEVEL: Triggers when the signal is below a specified level (hardware analog, software analog, counter, and digital pattern).

EQ_LEVEL: Triggers when the signal equals a specified level (hardware analog, software analog, counter, and digital pattern).

NE_LEVEL: Triggers when the signal does not equal a specified level (hardware analog, software analog, counter, and digital pattern).

Returns:

Error code or 0 if no errors
Notes:

**Trigger levels:** The actual level at which the trigger event is detected depends upon trigger sensing and variability. The various ranges of possible values for the `Level` parameter based on the trigger source are listed here:

- **TRIG_ANALOG_HW:** The voltage used to define the trigger level. Trigger detection is performed in hardware.
- **TRIG_ANALOG_SW:** The voltage used to define the trigger level. Trigger detection is performed in software.
- **TRIG_DIGPATTERN:** Sets the bit pattern for the digital channel trigger. Choices are:
  - 0.0 (no bits set): 255.0 (all bits set) for 8-bit digital ports.
  - 0.0 (no bits set): 65,535.0 (all bits set) for 16-bit digital ports.
- **TRIG_COUNTER:** Selects either Pulse or Totalize counter values (0.0 – 65,535).
- **TRIG_IMMEDIATE:** Ignored
- **TRIG_SCANCOUNT:** Ignored

**Trigger start and stop criteria:** The table below lists the trigger start and stop criteria based on the selected trigger type and sensitivity.

<table>
<thead>
<tr>
<th>Trigger Start/Stop Source (TrigSource)</th>
<th>Trigger Sensitivity (TrigSense)</th>
<th>Trigger Start/Stop Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIG_ANALOG_HW (Start trigger event only)</td>
<td>RISING_EDGE</td>
<td>Triggers when the signal value &lt; (Level – Variance) Then, the signal value &gt; Level</td>
</tr>
<tr>
<td></td>
<td>FALLING_EDGE</td>
<td>Triggers when the signal value &gt; (Level + Variance) Then, the signal value &lt; Level</td>
</tr>
<tr>
<td></td>
<td>ABOVE_LEVEL</td>
<td>Triggers when the signal value &gt; (Level)</td>
</tr>
<tr>
<td></td>
<td>BELOW_LEVEL</td>
<td>Triggers when the signal value &lt; (Level)</td>
</tr>
<tr>
<td>TRIG_ANALOG_SW</td>
<td>RISING_EDGE</td>
<td>Triggers/Stops when the signal value &lt; (Level – Variance) Then, the signal value &gt; Level</td>
</tr>
<tr>
<td></td>
<td>FALLING_EDGE</td>
<td>Triggers/Stops when the signal value &gt; (Level + Variance) Then, the signal value &lt; Level</td>
</tr>
<tr>
<td></td>
<td>ABOVE_LEVEL</td>
<td>Triggers/Stops when the signal value &gt; (Level)</td>
</tr>
<tr>
<td></td>
<td>BELOW_LEVEL</td>
<td>Triggers/Stops when the signal value &lt; (Level)</td>
</tr>
<tr>
<td></td>
<td>EQ_LEVEL</td>
<td>Triggers/Stops when the (Level – Variance) &lt; signal value &lt; (Level + Variance)</td>
</tr>
<tr>
<td></td>
<td>NE_LEVEL</td>
<td>Triggers/Stops when the signal value &lt; (Level – Variance) OR when the signal value &gt; (Level + Variance)</td>
</tr>
<tr>
<td>TRIG_DIGPATTERN</td>
<td>ABOVE_LEVEL</td>
<td>Triggers/Stops when the (digital port value AND (bitwise) Variance) &gt; (Level AND (bitwise) Variance)</td>
</tr>
<tr>
<td></td>
<td>BELOW_LEVEL</td>
<td>Triggers/Stops when the (digital port value AND (bitwise) Variance) &lt; (Level AND (bitwise) Variance)</td>
</tr>
<tr>
<td></td>
<td>EQ_LEVEL</td>
<td>Triggers/Stops when the (digital port value AND (bitwise) Variance) = (Level AND (bitwise) Variance)</td>
</tr>
<tr>
<td></td>
<td>NE_LEVEL</td>
<td>Triggers/Stops when the (digital port value AND (bitwise) Variance) != (Level AND (bitwise) Variance)</td>
</tr>
<tr>
<td>Trigger Start/Stop Source (TrigSource)</td>
<td>Trigger Sensitivity (TrigSense)</td>
<td>Trigger Start/Stop Criteria</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>--------------------------------</td>
<td>-----------------------------</td>
</tr>
</tbody>
</table>
| TRIG_COUNTER                         | RISING_EDGE                    | Triggers/ Stops when the counter channel < (Level – Variance)  
|                                      |                                | Then, the counter channel > Level |
| FALLING_EDGE                         |                                | Triggers/ Stops when counter channel > (Level + Variance)  
|                                      |                                | Then, the counter channel < Level |
| ABOVE_LEVEL                          |                                | Triggers/ Stops when counter channel > (Level – Variance) |
| BELOW_LEVEL                          |                                | Triggers/ Stops when counter channel < (Level + Variance) |
| EQ_LEVEL                             |                                | Triggers/ Stops when the (Level – Variance) < counter channel < (Level + Variance) |
| NE_LEVEL                             |                                | Triggers/ Stops when the counter channel < (Level – Variance)  
|                                      |                                | OR when the counter channel > (Level + Variance) |
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 InstaCal configuration file. They should be set by running the InstaCal configuration program.

**Function prototype:**

- **C/C++:**
  ```c
  int cbTIn(int BoardNum, int Chan, int Scale, float *TempVal, int Options);
  ```

- **Visual Basic:**
  ```vbs
  Function cbTIn(ByVal BoardNum&, ByVal Chan&, ByVal Scale&, TempVal!, ByVal Options&) As Long
  ```

- **Delphi:**
  ```delphi
  function cbTIn(BoardNum:Integer; Chan:Integer; Scale:Integer; var TempValue:Single; Options:Integer):Integer;
  ```

**Arguments:**

- **BoardNum**
  The number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.

- **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 \( \text{Chan} \) would be \((0 * 16) + (16 + 5) = 0 + 21 = 21\).

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

- \( \text{ADChan} \) is the A/D channel that is connected to the multiplexer.
- \( \text{MuxChan} \) 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:**
\[
\text{Chan} = (\text{ADChan} \times 16) + (16 + \text{MuxChan})
\]
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 \( \text{Chan} \) 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:**
\[
\text{Chan} = (\text{ADChan} \times 16) + (64 + \text{MuxChan})
\]
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 \( \text{Chan} \) 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 \( \geq 31 \):**
\[
\text{Chan} = (\text{ADChan} \times 16 - 320) + \text{MuxChan}
\]
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 \( \text{Chan} \) would be \((32 * 16 - 320) + 5 = 192 + 5 = 197\).

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

- 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.
- 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 \( \pm 5 \) 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](http://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. If a NotReady error occurs, the value returned is -9000.
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 InstaCal 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** The number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.
- **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`
Temperature Input Functions

**cbTInScan()**

- MuxChan is a number ranging from 0 to 15 that specifies the channel number on a particular bank of the multiplexer board
  \[ \text{Chan} = (\text{ADChan} \times 16) + (16 + \text{MuxChan}) \]

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 \((0 \times 16) + (16 + 5) = 0 + 21 = 21\), and the value for HighChan would be \((0 \times 16) + (16 + 7) = 0 + 23 = 23\).

**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](http://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:
  \[ \text{Chan} = (\text{ADChan} \times 16) + (16 + \text{MuxChan}) \]
  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 \((0 \times 16) + (16 + 5) = 0 + 21 = 21\), and the value for HighChan would be \((0 \times 16) + (16 + 7) = 0 + 23 = 23\).

- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is less than or equal to 7:
  \[ \text{Chan} = (\text{ADChan} \times 16) + (64 + \text{MuxChan}) \]
  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 \((7 \times 16) + (64 + 5) = 112 + 69 = 181\), and the value for HighChan would be \((7 \times 16) + (64 + 7) = 112 + 71 = 183\).

- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is greater than or equal to 32:
  \[ \text{Chan} = (\text{ADChan} \times 16 - 320) + \text{MuxChan} \]
  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 \((32 \times 16 - 320) + 5 = 192 + 5 = 197\), and the value for HighChan would be \((32 \times 16 - 320) + 7 = 192 + 7 = 199\).

**CJC Channel:** The CJC channel is set in the InstaCal 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](http://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 the board-specific information in the *Universal Library User's Guide*).
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.
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   The size of the buffer to allocate. Specifies how many data points (16-bit integers, NOT bytes) can be stored in the buffer.

Returns:

0 if the 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 that the buffer was not allocated.
cbWinBufAlloc32()

Allocates a Windows global memory buffer for use with 32-bit scan functions, and returns a memory handle for the buffer.

Function prototype:

C/C++: int cbWinBufAlloc32(long NumPoints)

Visual Basic: Function cbWinBufAlloc32(ByVal NumPoints&) As Long

Delphi: function cbWinBufAlloc32(NumPoints:Longint):Integer;

Arguments:

NumPoints The size of the buffer to allocate. Specifies how many data points (32-bit integers, NOT bytes) can be stored in the buffer.

Returns:

0 if a 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 that the buffer was not allocated.
cbWinBufFree()

Frees a Windows global memory buffer which was previously allocated with the cbWinBufAlloc() or cbWinBufAlloc32() 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() or cbWinBufAlloc32() 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 arguments 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 output scan has been started – for example, during 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:

```c
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*/
cbWinBuffFree(MemHandle);
DataArray = NULL;
```
cbWinBufToArray()

Copies data from a Windows memory buffer into an array.

Function prototype:

**C/C++:**
```c
int cbWinBufToArray(int MemHandle, unsigned short*DataArray, long FirstPoint, long Count)
```

**Visual Basic:**
```vb
Function cbWinBufToArray(ByVal MemHandle&, DataArray%, ByVal FirstPoint&, ByVal Count&) As Long
```

**Delphi:**
```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:

```c
/*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;

/*Scan the waveform data*/
cbAInScan(......,MemHandle,....);

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

/*free the buffer and NULL the pointer*/
cbWinBufFree(MemHandle);
DataArray = NULL;
```
cbWinBufToArray32()

Copies 32-bit data from a Windows global memory buffer into an array. This function is typically used to retrieve data from the buffer after executing an input scan function.

Function prototype:
C/C++: int cbWinBufToArray32(int MemHandle, unsigned long* DataArray, long FirstPoint, long Count)
Visual Basic: Function cbWinBufToArray32(ByVal MemHandle&, DataArray%, ByVal FirstPoint&), ByVal Count& As Long
Delphi: function cbWinBufToArray32(MemHandle:Integer; var DataArray:Longint; FirstPoint:Longint; Count:Longint):Integer;

Arguments:
MemHandle The memory handle that was returned by cbWinBufAlloc32() when the buffer was allocated. The buffer should contain the data that you want to copy.
DataArray The array that the data is copied to.
FirstPoint The index of the first point in the memory buffer that data is copied from.
Count The number of data points to copy.

Returns:
Error code or 0 if no errors

Notes:
You can copy only a portion of the buffer to the array using the FirstPoint and Count argument. This is 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 you can manipulate the memory buffer directly by casting the MemHandle returned from cbWinBufAlloc32() to the appropriate type. This method avoids having to copy the data from the memory buffer to an array. Refer to the following example:

```c
/*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 long*/
MemHandle = cbWinBufAlloc32(Count);
DataArray = (unsigned long*)MemHandle;

/*scan in the data*/
cbCInScan(.......MemHandle,...);

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

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

Opens a device session with a shared device.

Function prototype:

C/C++:

```c
int cbDeviceLogin(int BoardNum, char* UserName, char* Password);
```

Visual Basic:

```vbs
Function cbDeviceLogin (ByVal BoardNum&, UserName$, Password$) As Long
```

Arguments:

- `BoardNum` The number associated with the board when it was installed with the InstaCal configuration program. `BoardNum` may be 0 to 99.
- `UserName` A null-terminated string that identifies the user name used to log in to a device session.
- `Password` A null-terminated string that identifies the password used to log in to a device session.

Returns:

Error code or 0 if no errors.

Notes:

If the user name or password is invalid, the function returns INVALIDLOGIN.

If the session is already opened by another user, the function returns SESSIONINUSE.

cbDeviceLogout()

Releases the device session with a shared device.

Function prototype:

C/C++:

```c
int cbDeviceLogout(int BoardNum);
```

Visual Basic:

```vbs
Function cbDeviceLogout (ByVal BoardNum) As Long
```

Arguments:

- `BoardNum` The number associated with the board when it was installed with the InstaCal configuration program. `BoardNum` may be 0 to 99.

Returns:

Error code or 0 if no errors.
cbDisableEvent()

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. Refers to the number associated with the board when it was installed with the InstaCal 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 171 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 cbDisableEvent(). 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()

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++:

```c
int cbEnableEvent(int BoardNum, unsigned EventType, unsigned EventParam, void* CallbackFunc, void* UserData)
```

Visual Basic:

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

Delphi:

```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. Refers to the number associated with the board when it was installed with the InstaCal 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 "Event Type argument values" section below.

- **EventParam**
  
  Additional data required to specify some event conditions such as an ON_DATA_AVAILABLE event or ON_EXTERNAL_INTERRUPT event.

  For ON_DATA_AVAILABLE events, EventParam is used to determine the minimum number of samples to acquire during an analog input scan before generating the event. For ON_EXTERNAL_INTERRUPT events, EventParam is used to latch digital bits on supported hardware by setting it to one of the constants in the "EventParam argument values" section on page 172.

  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 172 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().

**EventParam argument values:**

| LATCH_DI  | Returns the data that was latched in at the most recent interrupt edge. |
| LATCH_DO  | Latches out the data most recently written to the hardware. |

**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 multi-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 applications can produce unpredictable results.
User Callback function

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 its address or a pointer to the function to the `CallbAckFunc` argument of the `cbEnableEvent()` function.

Callback function prototype:

**C/C++:**
```cpp
void __stdcall CallbackFunc(int BoardNum, unsigned EventType, unsigned EventData, void* UserData);
```

**Visual Basic:**
```vbnet
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 171).
```

**Delphi:**
```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. Returns the value of the `EventType` as listed in the "EventData argument values" section below.

- **UserData**
  The pointer or reference to data supplied by the `UserData` parameter in `cbEnableEvent()` (refer to page 171). Note that before use, this parameter must be cast to the same data type as passed in to `cbEnableEvent()`.

EventData argument values:

<table>
<thead>
<tr>
<th>EventType</th>
<th>Value of EventData</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ON_SCAN_ERROR</strong></td>
<td>The Error code of the scan error.</td>
</tr>
<tr>
<td><strong>ONEXTERNAL_INTERRUPT</strong></td>
<td>The number of interrupts generated since enabling the ONEXTERNAL_INTERRUPT event.</td>
</tr>
<tr>
<td><strong>ONPRETRIGGER</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>ONDATA_AVAILABLE</strong></td>
<td>The number of samples acquired since the start of scan.</td>
</tr>
<tr>
<td><strong>ONEND_OF_AI_SCAN</strong></td>
<td>The total number of samples acquired upon scan completion or end.</td>
</tr>
<tr>
<td><strong>ONEND_OF_AO_SCAN</strong></td>
<td>The total number of samples output upon scan completion or end.</td>
</tr>
</tbody>
</table>
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 single precision voltage (or current) value in engineering units to an integer count value. This function is typically used to obtain a data value from a voltage value for output to a D/A with functions such as cbAOut().

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 board when it was installed with the InstaCal configuration program. This function uses the board number to determine the resolution and polarity values to use in the conversion. BoardNum may be 0 to 99.

Range The voltage (or current) range to use for the conversion to counts. When using this function to obtain a value to send to a D/A board, keep in mind that some D/A boards have programmable voltage ranges, and others set the voltage range via switches on the board. In either case, the desired range must be passed to this function. Refer to Table 3 on page 28 for a list of valid range settings.

EngUnits The single precision voltage (or current) value to use for the conversion to counts. Set the value to be within the range specified by the Range argument.

DataVal The function returns an integer count to this variable that is equivalent to the EngUnits argument using the resolution of the D/A on the board referenced by BoardNum (if any).

Returns:

Error code or 0 if no errors.

DataVal – the integer count equivalent to EngUnits is returned here.

Notes:

This function is not supported for hardware with resolution greater than 16 bits.

The default resolution of this function is 12 bits, so if the device referenced by BoardNum has neither analog input nor analog output, the result will be a 12 bit conversion.

If the device referenced by BoardNum has both analog input and analog output, the resolution and transfer function of the D/A converter on the device is used.
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, `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](http://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 `ulg04` example program in the installation directory for more details.
cbGetStatus()

Returns the status about the background operation currently running.

Function prototype:

C/C++:
```c
int cbGetStatus(int BoardNum, int *Status, long *CurCount, long *CurIndex, int FunctionType)
```

Visual Basic:
```vb
Function cbGetStatus(ByVal BoardNum&, Status%, CurCount&, CurIndex&, FunctionType&) As Long
```

Delphi:
```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 InstaCal configuration program. BoardNum may be 0 to 99.

Status

The Status argument indicates whether or not a background process is currently executing.

CurCount

The CurCount argument specifies how many points have been input or output since the Background process started. Use it to gauge how far along the operation is towards completion. Generally, CurCount returns the total number of samples transferred between the DAQ board and the Windows data buffer at the time cbGetStatus() was called.

When you set both the CONTINUOUS and BACKGROUND options, CurCount's behavior depends on the board model. Refer to the board-specific information in the Universal Library User's Guide for the behavior of your board.

With recent MCC DAQ designs, the CurCount argument continually increases in increments of the packet size as Windows' circular data buffer recycles, until it reaches $2^{31}$. Since the Count argument is a signed integer, at 2,147,483,647 + 1, the Count argument rolls back to a negative number (-2,147,483,647). The Count argument resumes incrementing, eventually reaching 0 and increasing back up to 2,147,483,647.

The CurIndex argument is usually more useful than the CurCount argument in managing data collected when you set both the CONTINUOUS and BACKGROUND options.

CurIndex

The CurIndex argument is an index into the Windows data buffer. This index points to the start of the last completed channel scan that was transferred between the DAQ board and the Windows data buffer. If a scan is running but no points in the buffer have been transferred, CurIndex equals -1 in most cases.

For CONTINUOUS operations, CurIndex rolls over when the Windows data buffer is full. This rollover indicates that "new" data is now overwriting "old" data. Your goal is to process the old data before it gets overwritten. You can keep ahead of the data flow by copying the old data out of the buffer before new data overwrites it.

The CurIndex argument can help you access the most recently transferred data. Your application does not have to process the data exactly when it becomes available in the buffer – in fact, you should avoid doing so unless absolutely necessary. The CurIndex argument generally increments by the packet size, but in some cases the CurIndex increment can vary within the same scan. One instance of a variable increment is when the packet size is not evenly divisible by the number of channels.
You should determine the best size of the "chunks" of data that your application can most efficiently process, and then periodically check on the CurIndex argument value to determine when that amount of additional data has been transferred.

Refer to the *Universal Library User's Guide* for information on your board, particularly when using Pre-Trigger.

**FunctionType**

Specifies which scan to retrieve status information about. Set it to one of the constants in the "FunctionType argument values" section below.

**Returns:**

- **Error code** or 0 if no errors

**Status** - Returns the status of the operation:

- **IDLE** - No background operation is running.
- **RUNNING** - Background operation is still executing.

**CurCount** - Returns the current number of samples collected.

**CurIndex** - Returns the 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()` or `cbCInScan()`.
- **DAQIFUNCTION** Specifies a synchronous input scan started with `cbDaqInScan()`.
- **DAQOFUNCTION** Specifies a synchronous output scan started with `cbDaqOutScan()`.
cbGetTCValues()

Converts raw thermocouple data collected using the cbDaqInScan() function to data on a temperature scale (Celsius, Fahrenheit or Kelvin).

Function prototype:
C/C++:

```c
int cbGetTCValues(int BoardNum, short *ChanArray, short *ChanTypeArray, int ChanCount, int MemHandle, int FirstPoint, long Count, int Scale, float *TempValArray)
```

Visual Basic:

```vb
Function cbGetTCValues(ByVal BoardNum&, ChanArray%, ChanTypeArray%, ByVal ChanCount&, ByVal MemHandle&, ByVal FirstPoint&, ByVal Count&, ByVal CBScale&, TempValArray!) As Long
```

Delphi:

```delphi
function cbGetTCValues(BoardNum:Integer;  var ChanArray:SmallInt; var ChanTypeArray:SmallInt; ChanCount:Integer; MemHandle:Integer; FirstPoint:Integer; CBCount:LongInt; Scale:Integer; var TempValArray:Single):Integer;
```

Arguments:

**BoardNum**
The board number used to collect the data. BoardNum may be 0 to 99. 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 support synchronous input.

**ChanArray**
Array containing channel values. Valid channel values are analog and temperature input channels and digital ports. ChanArray must match the channel array used with the cbDaqInScan() function.

**ChanTypeArray**
Array containing channel types. Each element of this array defines the type of the corresponding element in the ChanArray. ChanTypeArray must match the channel type settings used with the cbDaqInScan() function.

**ChanCount**
Number of elements in ChanArray.

**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 convert.

**FirstPoint**
The index into the raw data memory buffer that holds the first sample of the first channel to be converted. The index into the raw memory is (FirstPoint x ChanCount) so that converted data always starts with the first channel specified in the scan. For example, if firstPoint is 14 and the number of channels is 8, the index of the first converted sample is 112.

**Count**
The number of samples per channel to convert to engineering units. Count should not exceed Windows buffer size / ChanCount – FirstPoint.

**Scale**
Specifies the temperature scale that the input will be converted to. Choices are CELSIUS, FAHRENHEIT and KELVIN.

**TempValArray**
The array to hold the converted data. This array must be allocated by the user, and must be large enough to hold count samples x the number of temperature channels.

Returns:

**Error code** or 0 if no errors

**TempValArray** – Converted data.
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**
  The number associated with the board when it was installed with the InstaCal configuration program. `BoardNum` may be 0 to 99.

- **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++: \( \text{int cbInWord(int BoardNum, int PortNum)} \)

Visual Basic: Function cbInWord(ByVal BoardNum&, ByVal PortNum&) As Long

Delphi: function cbInWord(BoardNum:Integer; PortNum:Integer):Integer;

Arguments:

BoardNum The number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.

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++:**
  ```c
  int cbOutByte(int BoardNum, int PortNum, int PortVal)
  ```
- **Visual Basic:**
  ```vbs
  Function cbOutByte(ByVal BoardNum&, ByVal PortNum&, ByVal PortVal%)
  As Long
  ```
- **Delphi:**
  ```delphi
  function cbOutByte(BoardNum:Integer; PortNum:Integer;
  PortVal:Integer):Integer;
  ```

**Arguments:**

- **BoardNum**
  The number associated with the board when it was installed with the *InstaCal* configuration program. BoardNum may be 0 to 99.

- **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.
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**
  - The number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.

- **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 number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.

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 number associated with the board when it was installed with the InstaCal configuration program. BoardNum may be 0 to 99.

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()

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() or cbCInScan()

DAQFUNCTION Specifies a synchronous input scan started with cbDaqInScan()

DAQOFUNCTION Specifies a synchronous output scan started with cbDaqOutScan()
cbToEngUnits()

Converts an integer count value to an equivalent single precision voltage (or current) value. This function is typically used to obtain a voltage value from data received from an A/D with functions such as cbAIn().

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 board when it was installed with the InstaCal configuration program. This function uses the board number to determine the resolution and polarity values to use for the conversion. BoardNum may be 0 to 99.

Range: Voltage (or current) range to use for the conversion to engineering units. When using this function to obtain engineering units from a value received from an A/D board, keep in mind that some A/D boards have programmable voltage ranges, and others set the voltage range via switches on the board. In either case, the desired range must be passed to this function. Refer to Table 3 on page 28 for a list of valid range settings.

DataVal: An integer count value (typically, one returned from an A/D board).

EngUnits: The single precision 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 using the resolution of the A/D on the board referenced by BoardNum (if any).

Returns:

- Error code or 0 if no errors.

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

Notes:

This function is not supported for hardware with resolution greater than 16 bits.

The default resolution of this function is 12 bits, so if the device referenced by BoardNum has neither analog input nor analog output, the result will be a 12 bit conversion.

If the device referenced by BoardNum has both analog input and analog output, the resolution and transfer function of the A/D converter on the device is used.
Universal Library for .NET
Classes, Methods, and Properties
UL for .NET Class Library Overview

The 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 five main classes:

- MccBoard class
- ErrorInfo class
- MccService class
- GlobalConfig class
- DataLogger 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 188 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 401.

Properties and methods
The ErrorInfo class also includes two properties that you can use to examine error information. Each property 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 188 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 188 for an explanation of the MccDaq namespace.

**Properties and methods**
The GlobalConfig class 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.

**DataLogger class**
Contains all of the members for reading and converting binary log files.

The DataLogger class is a member of the MccDaq namespace. Refer to the "MccDaq namespace" on page 188 for an explanation of the MccDaq namespace.

**Property and methods**
The DataLogger class provides one property to get a reference to the file name associated with the current instance of the DataLogger.

The DataLogger class includes 14 methods used to read and convert the data contained in a binary log file. These methods are equivalents of the function calls used in the standard Universal Library. The methods have virtually the same parameter set as their UL counterparts.

The DataLogger property and methods are 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 channel/gain pairs into an A/D board's queue. These channel/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.

  For devices with a hardware implementation of pretrigger, 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.

- **VIn()** - Reads an A/D input channel, and returns a voltage value.

- **VOut()** - Sets the value of a D/A output.
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 - Retrieves the 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.GetAdRetrigCount()** - Gets the number of samples to acquire during a trigger event when ScanOptions.RetrigMode is set.
- **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.GetClock()** - Gets the clock frequency in MHz (40, 10, 8, 6, 5, 4, 3, 2, 1), or 0 for not supported.
- **MccBoard.BoardConfig.GetDACStartup()** - Gets the board’s configuration register STARTUP bit setting.
- **MccBoard.BoardConfig.GetDACUpdateMode()** - Gets the update mode for a digital-to-analog converter (DAC).
- **MccBoard.BoardConfig.GetDeviceId()** – Gets the name that identifies the instance of a device.
- **MccBoard.BoardConfig.GetDeviceNotes()** – Gets the device notes that are stored in the device's memory.
- **MccBoard.BoardConfig.GetDInMask()** - Determines the bits on a specified port that are configured for input.
- **MccBoard.BoardConfig.GetDInMask()** - 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.GetPANID()` - Gets the Personal Area Network (PAN) identifier for wireless communication.
- `MccBoard.BoardConfig.GetRange()` - Gets the selected voltage range.
- `MccBoard.BoardConfig.GetRFChannel()` - Gets the RF channel number that a wireless device uses to communicate.
- `MccBoard.BoardConfig.GetRSS()` - Gets the signal strength in dBm of a signal received by a remote device.
- `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.SetAdRetrigCount()` - Sets the number of samples to acquire during a trigger event when `ScanOptions.RetrigMode` is set.
- `MccBoard.BoardConfig.SetBaseAddr()` - 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.SetDeviceId()` - Sets the name that identifies the instance of a device.
- `MccBoard.BoardConfig.SetDeviceNotes()` - Sets the device notes to store in a device's memory.
- `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.SetPanID()` - Sets the Personal Area Network (PAN) identifier used for wireless communication.
- `MccBoard.BoardConfig.SetRange()` - Sets the selected voltage range.
- `MccBoard.BoardConfig.SetRFChannel()` - Sets the RF channel number used for wireless communications.
- `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.GetDevType()` - Gets the device type of the digital port (`AUXPORT`, `FIRSTPORTA`, etc.).
- `MccBoard.DioConfig.GetDlnMask()` - 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 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 methods 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: 8254s, 8536s, 7266s, 9513s, 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.CClear()` - Clears a scan counter value (sets it to zero).
- `MccBoard.CConfigScan()` - Configures a scan counter channel. This method only works with counter boards that have counter scan capability.
- `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). This method only works with 9513 counters.
- **MccBoard.CIn()** - Reads a counter's current value as a 16-bit integer. (**CIn32()** is the preferred counter read method.)
- **MccBoard.CIn32()** - Reads a counter's current value as a 32-bit integer.
- **MccBoard.CInScan()** - Scans a range of scan counter channels, and stores the samples in an array.
- **MccBoard.CLoad()** – Loads a counter with an initial count value as a 16-bit integer. (**CLoad32()** is the preferred counter loading method.)
- **MccBoard.CLoad32()** – Loads a counter with a 32-bit integer initial value.
- **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.
- **MccBoard.TimerOutStart()** - Starts a timer square wave output. This method only works with counter boards that have a timer-type counter.
- **MccBoard.TimerOutStop()** - Stops a timer square wave output. This method only works with counter boards that have a timer-type counter.

## Data Logger methods and properties

The methods and property available from the [DataLogger class](#) are explained below. These class members read and convert binary log files.

- **DataLogger.ConvertFile()** – Converts a binary log file to a comma-separated values (.CSV) text file or another text file format that you specify.
- **DataLogger.GetAIChannelCount()** – Retrieves the total number of analog channels that were logged in a binary file.
- **DataLogger.GetAllInfo()** – Retrieves the channel number and unit value of each analog input channel logged in a binary file.
- **DataLogger.GetCJCInfo()** - Retrieves the number of CJC temperature channels logged in a binary file.
- **DataLogger.GetDIOInfo()** – Retrieves the number of digital I/O channels logged in a binary file.
- **DataLogger.GetFileInfo()** – Retrieves the version level and byte size of a binary file.
- **DataLogger.GetFileName()** – Retrieves the name of the nth file in the directory containing binary log files.
- **DataLogger.GetPreferences()** - Retrieves API preference settings for time stamp data, analog temperature data, and CJC temperature data. Returns the default values unless changed using **SetPreferences()**.
- **DataLogger.GetSampleInfo()** – Retrieves the sample interval, sample count, and the date and time of the first data point in a binary file.
- **DataLogger.ReadAIChannels()** – Retrieves analog input data from a binary file, and stores the values in an array.
- **DataLogger.ReadCJCChannels()** – Retrieves CJC temperature data from a binary file, and stores the values in an array.
- **DataLogger.ReadDIOChannels()** – Retrieves digital I/O channel data from a binary file, and stores the values in an array.
- **DataLogger.ReadTimeTags()** – Retrieves date and time values logged in a binary file. This method stores date values in the dateTags array, and time values in the timeTags array.
- **DataLogger.SetPreferences()** – Sets preferences for returned time stamp data, analog temperature data, and CJC temperature data.
- **DataLogger.FileName** property – Returns the file name associated with an instance of the DataLogger class.
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.
- MccBoard.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 method 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 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 properties and methods 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 a one-dimensional or two-dimensional array.

Synchronous I/O methods

The synchronous I/O methods available from the `MccBoard` class synchronously read, set, or write data from analog channels, counter channels, thermocouple channels, and digital ports.

- `MccBoard.DaqInScan()` – Scans analog, digital, temperature, and counter inputs synchronously, and stores the values in an array.
- `MccBoard.DaqOutScan()` – Outputs values synchronously to analog output channels and digital output ports.
- `MccBoard.DaqSetSetpoints()` – Configures up to 16 detection setpoints associated with the input channels within a scan group.
- `MccBoard.DaqSetTrigger()` – Selects a trigger source and sets up its parameters. This method starts or stops a synchronous data acquisition operation using `MccBoard.DaqInScan()` with the ExtTrigger option.

Temperature input methods

The temperature input methods available from the `MccBoard` class 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.WinArrayToBuf() - Copies data from a one-dimensional or two-dimensional array into a Windows buffer.
- MccService.WinBufToArray() - Copies data from a Windows memory buffer into a one-dimensional or two-dimensional array.
- MccService.WinBufToArray32() - Copies 32-bit data from a Windows global memory buffer into an array. This method is typically used to retrieve data from the buffer after executing an input scan method.

Miscellaneous methods, properties, and delegates

The methods explained below are available from the MccBoard class. These methods do not as a group fit into a single category. They get and set board information, convert units, manage events and background operations, copy two-dimensional arrays to/from Windows global memory buffers, and perform serial communication operations.

- MccBoard.DeviceLogin() - Opens a device session with a shared device.
- MccBoard.DeviceLogout() - Releases the device session with a shared device.
- MccBoard.DisableEvent() - Disables one or more events set up with EnableEvent() and disconnects their user-defined handlers.
- MccBoard.EnableEvent() - Binds one or more event conditions to a user-defined callback function.
- 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.EngArrayToWinBuf() - Transfers a 2D array of engineering unit values to a Windows buffer as integer values.
- MccBoard.FlashLED() - Causes the LED on a USB device to flash.
- MccBoard.FromEngUnits() - Converts a single precision voltage (or current) value in engineering units to an integer D/A count value for output to a D/A.
- MccBoard.GetBoardName() - Returns the name of a specified board.
- 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.GetTCValues() - Converts raw thermocouple data gathered with DaqInScan() to Celsius, Fahrenheit, or Kelvin.
- MccBoard.HideLoginDialog() - Prevents the default login dialog from being shown when a protected function is called while not logged in.
- 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.RS485() - Sets the transmit and receive buffers on an RS485 port.
UL for .NET Class Library Overview

Universal Library for .NET example programs

- **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.ToEngUnits()** - Converts an integer A/D count value to an equivalent single precision voltage (or current) value.

- **MccBoard.WinBufToEngArray()** - Transfers integer values from a Windows buffer to a 2D array as engineering unit values.

- **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 5 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 6 on page 203 lists the UL for .NET example programs sorted by the method name.

### Table 5. UL for .NET Example Programs – sorted by program name

<table>
<thead>
<tr>
<th>Program name</th>
<th>Featured UL for .NET method call</th>
<th>Notes</th>
<th>Other UL for .NET method calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>CInScan01</td>
<td>cbCInScan()</td>
<td>Scans a range of counter input channels, and writes the data to an array. Board 0 must support counter scans.</td>
<td>WinBuffAlloc32() WinBufToArray32() WinBufFree()</td>
</tr>
<tr>
<td>CInScan02</td>
<td>CInScan() CConfigScan()</td>
<td>Scans a counter input channel in decrement mode, and writes the data to an array. Board 0 must support counter scans.</td>
<td>WinBuffAlloc32() WinBufToArray32() WinBufFree()</td>
</tr>
<tr>
<td>DaqInScan01</td>
<td>DaqInScan()</td>
<td>Synchronously scans analog input channels, digital input ports and counter input channels in the foreground. Board 0 must support synchronous input</td>
<td>DConfigPort() CConfigScan()</td>
</tr>
<tr>
<td>DaqInScan02</td>
<td>DaqInScan()</td>
<td>Synchronously scans analog input channels, digital input ports, and counter input channels in the background. Board 0 must support synchronous input</td>
<td>DConfigPort() CConfigScan() GetStatus() StopBackground()</td>
</tr>
<tr>
<td>DaqInScan03</td>
<td>DaqInScan() GetTCValues()</td>
<td>Synchronously scans analog input channels, digital input ports and thermocouple input channels in the foreground. Board 0 must support synchronous input.</td>
<td>DConfigPort()</td>
</tr>
<tr>
<td>DaqOutScan01</td>
<td>DaqOutScan()</td>
<td>Synchronously writes to an analog output channel and a digital output port in the background. Board 0 must support synchronous output</td>
<td>DConfigPort()</td>
</tr>
<tr>
<td>DaqSetSetpoints01</td>
<td>DaqSetSetpoints()</td>
<td>Configures setpoints, adds the setpoint status to the scanlist, and performs asynchronous reads of the setpoint status. Board 0 must support DaqInScan().</td>
<td>DaqInScan() DConfigPort() GetStatus() StopBackground()</td>
</tr>
<tr>
<td>Program name</td>
<td>Featured UL for .NET method call</td>
<td>Notes</td>
<td>Other UL for .NET method calls</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------</td>
<td>-------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>DaqSetTrigger01</td>
<td>DaqSetTrigger()</td>
<td>Configures start and stop triggers. These triggers are used to initiate and terminate A/D conversion using DaqInScan() with the ExtTrigger option selected. Board 0 must support synchronous output.</td>
<td>DConfigPort() GetStatus() StopBackground()</td>
</tr>
<tr>
<td>TimerOut01</td>
<td>TimerOutStart() TimerOutStop()</td>
<td>Sends a frequency output to an output timer channel. Board 0 must have a timer output.</td>
<td></td>
</tr>
<tr>
<td>ULA01</td>
<td>AIn()</td>
<td></td>
<td>ToEngUnits()</td>
</tr>
<tr>
<td>ULA02</td>
<td>AInScan()</td>
<td>Default mode</td>
<td>WinBufToArray() WinBufFree() WinBufAlloc()</td>
</tr>
<tr>
<td>ULA03</td>
<td>AInScan()</td>
<td>Background mode</td>
<td>GetStatus() StopBackground() WinBufToArray() WinBufFree() WinBufAlloc()</td>
</tr>
<tr>
<td>ULA04</td>
<td>AConvertData()</td>
<td></td>
<td>AInScan() GetStatus() StopBackground() WinBufToArray() WinBufFree() WinBufAlloc()</td>
</tr>
<tr>
<td>ULA05</td>
<td>AInScan()</td>
<td>With manual data conversion</td>
<td>GetStatus() StopBackground() WinBufToArray() WinBufFree() WinBufAlloc()</td>
</tr>
<tr>
<td>ULA06</td>
<td>AInScan()</td>
<td>Continuous Background mode</td>
<td>AConvertData() GetStatus() StopBackground() WinBufToArray() WinBufFree() WinBufAlloc()</td>
</tr>
<tr>
<td>ULA07</td>
<td>ATrig()</td>
<td></td>
<td>FromEngUnits()</td>
</tr>
<tr>
<td>ULA08</td>
<td>APretrig()</td>
<td></td>
<td>WinBufToArray() WinBufFree() WinBufAlloc()</td>
</tr>
<tr>
<td>ULA09</td>
<td>ConvertPretrigData()</td>
<td>Background</td>
<td>APretrig() GetStatus() StopBackground() WinBufToArray() WinBufFree() WinBufAlloc()</td>
</tr>
<tr>
<td>ULA10</td>
<td>cbALoadQueue()</td>
<td></td>
<td>AInScan() WinBufToArray() WinBufFree() WinBufAlloc()</td>
</tr>
<tr>
<td>ULA11</td>
<td>cbToEngUnits()</td>
<td></td>
<td>AIn()</td>
</tr>
<tr>
<td>Program name</td>
<td>Featured UL for .NET method call</td>
<td>Notes</td>
<td>Other UL for .NET method calls</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------</td>
<td>-------</td>
<td>-------------------------------</td>
</tr>
</tbody>
</table>
| ULA112      | cbAIlnScan()                    | ExtClock mode | WinBufToArray()  
|             |                                 |       | WinBufFree()  
|             |                                 |       | WinBufAlloc()  
| ULA113      | cbAIlnScan()                    | Various sampling mode options | WinBufToArray()  
|             |                                 |       | WinBufFree()  
|             |                                 |       | WinBufAlloc()  
| ULA114      | SetTrigger()                    | With ExtTrigger selected | AInScan()  
|             |                                 |       | FromEngUnits()  
|             |                                 |       | WinBufToArray()  
|             |                                 |       | WinBufFree()  
|             |                                 |       | WinBufAlloc()  
| ULA001      | AInScan()                        | Concurrent analog input and analog output scans | GetStatus()  
|             | AOutScan()                       |       | StopBackground()  
|             |                                 |       | WinArrayToArray()  
|             |                                 |       | WinBufFree()  
|             |                                 |       | WinBufAlloc()  
| ULA002      | AOutScan()                       | | WinBufToArray()  
|             |                                 |       | WinBufFree()  
|             |                                 |       | WinBufAlloc()  
| ULA003      | AOut()                           | 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()  
|             | DACUpdate()                      |       |  
|             | SetDACUpdateMode()               |       |  
|             |                                 |       |  
| ULCT01      | C8254Config()                   | CLoad(), Cin() |  
| ULCT02      | C9513Init()                      | CLoad(), Cin() |  
| ULCT03      | C9513Config()                   | CLoad()  
|             |                                 | Cin()  
| ULCT04      | C9513Init()                      | CLoad()  
|             |                                 | Cin()  
| ULCT05      | C8536Init()                      | CLoad32(), Cin32()  
|             | C8536Config()                    | CStatus()  
| ULCT06      | C7266Config()                   | CLoadConfigPort() |  
| ULDI01      | DIn()                            | DConfigPort() |  
| ULDI02      | DBitIn()                         | DConfigPort() |  
| ULDI03      | DInScan()                        | Using the AuxPort | DConfigPort()  
|             |                                 |       | DConfigPort()  
| ULDI04      | DIn()                            |       | DConfigPort()  
|             |                                 |       | DConfigPort()  


<table>
<thead>
<tr>
<th>Program name</th>
<th>Featured UL for .NET method call</th>
<th>Notes</th>
<th>Other UL for .NET method calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>ULDI05</td>
<td>DBitIn()</td>
<td>Using the AuxPort</td>
<td>DioConfig() DConfigPort()</td>
</tr>
<tr>
<td>ULDI06</td>
<td>DConfigBit()</td>
<td></td>
<td>DBitIn() DioConfig() DConfigPort()</td>
</tr>
<tr>
<td>ULDI07</td>
<td>DOut()</td>
<td></td>
<td>DOut() DConfigPort()</td>
</tr>
<tr>
<td>ULDI08</td>
<td>DBitOut()</td>
<td>Using the AuxPort</td>
<td>DioConfig() DConfigPort()</td>
</tr>
<tr>
<td>ULDI09</td>
<td>EnableEvent() DisableEvent()</td>
<td></td>
<td>DConfigPort() DIN()</td>
</tr>
<tr>
<td>ULDI10</td>
<td>EnableEvent() DisableEvent()</td>
<td>Using OnDataAvailable and OnEndOfAiScan</td>
<td>AInScan() StopBackground() ToEngUnits() WinBufAlloc() WinBufFree() WinBufToArray()</td>
</tr>
<tr>
<td>ULDI11</td>
<td>EnableEvent() DisableEvent()</td>
<td></td>
<td>APretrig() AConvertPretrigData() DConfigPort() DOut() StopBackground() ToEngUnits() WinBufAlloc() WinBufFree() WinBufToArray()</td>
</tr>
<tr>
<td>ULDI12</td>
<td>EnableEvent() DisableEvent()</td>
<td>Using OnEndOfAoScan</td>
<td>AOutScan() DConfigPort() DOut() FromEngUnits() StopBackground() WinArrayToBuf() WinBufAlloc() WinBufFree()</td>
</tr>
<tr>
<td>ULDI13</td>
<td>FileAInScan()</td>
<td></td>
<td>FileGetInfo()</td>
</tr>
<tr>
<td>ULDI14</td>
<td>FileRead()</td>
<td></td>
<td>FileAInScan() FileGetInfo()</td>
</tr>
<tr>
<td>ULDI15</td>
<td>FilePretrig()</td>
<td></td>
<td>FileGetInfo() FileRead()</td>
</tr>
<tr>
<td>ULDI16</td>
<td>GetErrMsg()</td>
<td></td>
<td>AIn()</td>
</tr>
<tr>
<td>ULDI17</td>
<td>MccDaq().MccBoard() class() properties: BoardConfig, DioConfig and ExpansionConfig</td>
<td>Use the MccBoard class properties to get configuration information for a board.</td>
<td>GetBoardName()</td>
</tr>
<tr>
<td>Program name</td>
<td>Featured UL for .NET method call</td>
<td>Notes</td>
<td>Other UL for .NET method calls</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------</td>
<td>-------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>ULGT04</td>
<td>GetBoardName()</td>
<td></td>
<td>MccDaq.MccBoard.BoardName property, MccDaq.GlobalConfig.NumBoards property</td>
</tr>
<tr>
<td>ULLOG01</td>
<td>GetFileName()</td>
<td>Retrieves the name of a binary log file.</td>
<td>GetFileName()</td>
</tr>
<tr>
<td>ULLOG02</td>
<td>GetFileInfo()</td>
<td>Retrieves information about the analog data, CJC data, and digital I/O channel data contained in a binary log file.</td>
<td>GetFileName()</td>
</tr>
<tr>
<td>ULLOG03</td>
<td>ReadAIChannels()</td>
<td>Retrieves the analog input data, CJC temperature data, digital I/O channel data, date values, and time values logged in a binary file, and writes the data to separate arrays.</td>
<td>GetFileName()</td>
</tr>
<tr>
<td>ULLOG04</td>
<td>ConvertFile()</td>
<td>Converts a binary log file to a comma-separated values (.csv) text file or another text file format that you specify.</td>
<td>GetSampleInfo()</td>
</tr>
<tr>
<td>ULMM01</td>
<td>MemReadPretrig()</td>
<td></td>
<td>APretrig()</td>
</tr>
<tr>
<td>ULMM02</td>
<td>MemRead()</td>
<td></td>
<td>MemReset() MemRead()</td>
</tr>
<tr>
<td>ULMM03</td>
<td>AInScan()</td>
<td>With ExtMemory option</td>
<td>MemReset() MemRead()</td>
</tr>
<tr>
<td>ULTI01</td>
<td>TIn()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULTI02</td>
<td>TInScan()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIn01</td>
<td>VIn()</td>
<td>Reads an A/D input channel.</td>
<td></td>
</tr>
<tr>
<td>VOut01</td>
<td>VOut()</td>
<td>Writes to a D/A output channel.</td>
<td></td>
</tr>
</tbody>
</table>

Table 6. UL for .NET Example Programs – sorted by method name

<table>
<thead>
<tr>
<th>UL for .NET method call</th>
<th>UL for .NET example program Name</th>
<th>UL for .NET special features/notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AConvertData()</td>
<td>ULA04</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ULA06</td>
<td></td>
</tr>
<tr>
<td>AConvertPretrigData()</td>
<td>ULA09</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ULEV03</td>
<td></td>
</tr>
<tr>
<td>ACalibrateData()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>AIn()</td>
<td>ULA01 ULGT01 ULA11</td>
<td></td>
</tr>
<tr>
<td>AInScan()</td>
<td>ULA02 ULA10 ULA03 ULA12 ULA04 ULA13 ULA05 ULA14 ULA06 ULM03 ULEV02</td>
<td>Default, Background mode with manual data conversion, Continuous Background mode, ExtClock mode, Various sampling mode options</td>
</tr>
<tr>
<td>ALoadQueue()</td>
<td>ULA10</td>
<td></td>
</tr>
<tr>
<td>AOut()</td>
<td>ULA001</td>
<td>Demonstrates the difference between BoardConfig.DACUpdate.Immediate and BoardConfig.DACUpdate.Command D/A update modes. Board 0 must support DAC update mode settings, such as the PCI-DAC6700 Series boards.</td>
</tr>
<tr>
<td>UL for .NET method call</td>
<td>UL for .NET example program Name</td>
<td>UL for .NET special features/notes</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>AOutScan()</td>
<td>ULA002 ULA001 ULEV04</td>
<td>Concurrent AInScan() and AOutScan()</td>
</tr>
<tr>
<td>APretrig()</td>
<td>ULA08 ULFI03 ULA09 ULMM01 ULEV03</td>
<td></td>
</tr>
<tr>
<td>ATrig()</td>
<td>ULA07 ULMM01</td>
<td></td>
</tr>
<tr>
<td>C7266Config()</td>
<td>ULCT06</td>
<td></td>
</tr>
<tr>
<td>C8254Config()</td>
<td>ULCT01</td>
<td></td>
</tr>
<tr>
<td>C8536Config()</td>
<td>ULCT05</td>
<td></td>
</tr>
<tr>
<td>C8536Init()</td>
<td>ULCT05</td>
<td></td>
</tr>
<tr>
<td>C9513Config()</td>
<td>ULCT02 ULCT03</td>
<td></td>
</tr>
<tr>
<td>C9513Init()</td>
<td>ULCT02 ULCT04 ULCT03</td>
<td></td>
</tr>
<tr>
<td>CClear()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CConfigScan()</td>
<td>ClnScan02</td>
<td>Demonstrates how to scan a counter input channel in decrement mode, and then write the data to an array. Board 0 must support counter scans.</td>
</tr>
<tr>
<td>CFreqIn()</td>
<td>ULCT04</td>
<td></td>
</tr>
<tr>
<td>CIn()</td>
<td>ULCT01 ULCT05 ULCT02</td>
<td></td>
</tr>
<tr>
<td>CIn32()</td>
<td>ULCT06</td>
<td></td>
</tr>
<tr>
<td>CInScan()</td>
<td>ClnScan01 ClnScan02</td>
<td>Demonstrates how to scan a range of counter channels and then write the data to an array. Board 0 must support counter scans.</td>
</tr>
<tr>
<td>CLoad()</td>
<td>ULCT01 ULCT03 ULCT02 ULCT07</td>
<td></td>
</tr>
<tr>
<td>CLoad32()</td>
<td>ULCT06</td>
<td></td>
</tr>
<tr>
<td>ConvertFile()</td>
<td>ULLLOG04</td>
<td>Demonstrates how to convert a binary log file to a different format. You enter the extension of the file type that you want to create.</td>
</tr>
<tr>
<td>CStoreOnInt()</td>
<td>ULCT03</td>
<td></td>
</tr>
<tr>
<td>CStatus()</td>
<td>ULCT06</td>
<td></td>
</tr>
<tr>
<td>DaqInScan()</td>
<td>DaqInScan01 DaqInScan02 DaqInScan03</td>
<td>Demonstrates how to synchronously scan analog, counter, and thermocouple input channels, and digital input ports. Board 0 must support synchronous input.</td>
</tr>
<tr>
<td>DaqOutScan()</td>
<td>DaqOutScan01</td>
<td>Demonstrates how to synchronously write to an analog output channel and digital output port in the background. Board 0 must support synchronous output.</td>
</tr>
<tr>
<td>DaqSetSetpoints()</td>
<td>DaqSetSetpoints01</td>
<td>Demonstrates how to configure and use setpoints, including how to add the setpoint status to the scanlist and perform asynchronous reads of the setpoint status. Board 0 must support DaqInScan().</td>
</tr>
<tr>
<td>DaqSetTrigger()</td>
<td>DaqSettrigger01</td>
<td>Demonstrates how to set up start and stop trigger events and display input channel data.</td>
</tr>
<tr>
<td>DBitIn()</td>
<td>ULDIO2 ULDIO6 ULDIO5</td>
<td></td>
</tr>
<tr>
<td>DBitOut()</td>
<td>ULDIO02 ULDIO05</td>
<td></td>
</tr>
<tr>
<td>DConfigBit()</td>
<td>ULDIO06</td>
<td></td>
</tr>
<tr>
<td>UL for .NET method call</td>
<td>UL for .NET example program Name</td>
<td>UL for .NET special features/notes</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>DConfigPort()</td>
<td>ULDI01  ULDI02  ULDI03  ULDI05</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ULEV01  ULEV03  ULEV04</td>
<td></td>
</tr>
<tr>
<td>DIN()</td>
<td>ULDI01  ULDI04  ULDI03  ULEV01</td>
<td></td>
</tr>
<tr>
<td>DINScan()</td>
<td>ULDI03</td>
<td></td>
</tr>
<tr>
<td>DOut()</td>
<td>ULDO01  ULDO05  ULDO02  ULDO04</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ULEV03  ULEV04</td>
<td></td>
</tr>
<tr>
<td>DOutScan()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>EnableEvents()</td>
<td>ULEV01  ULEV03</td>
<td>OnExternalInterrupt</td>
</tr>
<tr>
<td>DisableEvents()</td>
<td>ULEV02  ULEV04</td>
<td>OnDataAvailable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OnPretrigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OnEndOfAoScan</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OnScanError</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OnEndOfAiScan</td>
</tr>
<tr>
<td>ErrHandling()</td>
<td>All samples</td>
<td>All example programs use this method</td>
</tr>
<tr>
<td>FileAInScan()</td>
<td>ULFI01  ULFI02</td>
<td></td>
</tr>
<tr>
<td>FilePretrig()</td>
<td>ULFI03</td>
<td></td>
</tr>
<tr>
<td>FileRead()</td>
<td>ULFI02  ULFI03</td>
<td></td>
</tr>
<tr>
<td>FlashLED()</td>
<td>ULFI01</td>
<td>Flashes the onboard LED for visual identification (board 0 must have an external LED, such as the miniLAB 1008 or the USB-1208LS.</td>
</tr>
<tr>
<td>FromEngUnits</td>
<td>ULA001  ULA003</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ULA07  ULA14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ULEV04</td>
<td></td>
</tr>
<tr>
<td>GetAIChannelCount()</td>
<td>ULLLOG02  ULLLOG03</td>
<td>Demonstrates how to retrieve the number of analog channels contained in a binary log file.</td>
</tr>
<tr>
<td>GetAIInfo()</td>
<td>ULLLOG02</td>
<td>Demonstrates how to retrieve information about the analog input data contained in a binary log file.</td>
</tr>
<tr>
<td>GetBoardName</td>
<td>ULETTE03  ULETTE04</td>
<td></td>
</tr>
<tr>
<td>GetCJCInfo()</td>
<td>ULLLOG02  ULLLOG03</td>
<td>Demonstrates how to retrieve information about CJC temperature data contained in a binary log file.</td>
</tr>
<tr>
<td>GetDACStartup()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>GetDACUpdateMode()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>GetDIOInfo()</td>
<td>ULLLOG02  ULLLOG03</td>
<td>Demonstrates how to retrieve information about digital I/O channel data contained in a binary log file.</td>
</tr>
<tr>
<td>GetErrMsg()</td>
<td>Ulette01</td>
<td></td>
</tr>
<tr>
<td>GetFileInfo()</td>
<td>ULLLOG02</td>
<td>Demonstrates how to retrieve the version level and byte size of a binary log file.</td>
</tr>
<tr>
<td>GetFileName()</td>
<td>ULLLOG01  ULLLOG02  ULLLOG03</td>
<td>Demonstrates how to retrieve the name of a binary log file.</td>
</tr>
<tr>
<td>GetPreferences()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>GetRevision()</td>
<td>None</td>
<td>No example programs at this time</td>
</tr>
<tr>
<td>UL for .NET method call</td>
<td>UL for .NET example program Name</td>
<td>UL for .NET special features/notes</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>GetSampleInfo()</td>
<td>ULL02, ULL03</td>
<td>Demonstrates how to retrieve the sample interval, sample count, and the date and time of the first data point logged in a binary file.</td>
</tr>
<tr>
<td>GetStatus()</td>
<td>ULA03, ULA06, ULA04, ULA09, ULA05, ULA01, ULD01</td>
<td></td>
</tr>
<tr>
<td>GetTCValues()</td>
<td>DaqInScan03</td>
<td>Demonstrates how to retrieve analog, thermocouple, and digital data from a synchronous scan operation. Board 0 must support synchronous input.</td>
</tr>
<tr>
<td>InByte()</td>
<td>None</td>
<td>No example programs at this time.</td>
</tr>
<tr>
<td>MccDaq.MccBoard class properties:</td>
<td>ULG03, ULG04</td>
<td>Use the MccBoard class properties to get configuration information for a board.</td>
</tr>
<tr>
<td>MemRead()</td>
<td>ULMM01, ULMM03, ULMM02</td>
<td></td>
</tr>
<tr>
<td>MemReadPretrig()</td>
<td>ULMM01</td>
<td></td>
</tr>
<tr>
<td>MemSetDINMode()</td>
<td>None</td>
<td>No example programs at this time.</td>
</tr>
<tr>
<td>MemWrite()</td>
<td>ULMM02</td>
<td></td>
</tr>
<tr>
<td>ReadAIChannels()</td>
<td>ULL03</td>
<td>Demonstrates how to retrieve analog input data, CJC data, DIO port data, and date/time values contained in a binary log file, and then store the data in separate arrays.</td>
</tr>
<tr>
<td>ReadCJCChannels()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReadDIOChannels()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReadTimeTags()</td>
<td>ULLOG03</td>
<td></td>
</tr>
<tr>
<td>RS485()</td>
<td>None</td>
<td>No example programs at this time.</td>
</tr>
<tr>
<td>SetTrigger()</td>
<td>ULA14</td>
<td></td>
</tr>
<tr>
<td>SetPreferences()</td>
<td>ULL03</td>
<td>Demonstrates how to store preference settings for time stamped data, analog data, and CJC temperature data.</td>
</tr>
<tr>
<td>StopBackground()</td>
<td>ULA03, ULA06, ULA04, ULA09, ULA05, ULA01, ULD01, ULE02, ULE03, ULE04</td>
<td>Concurrent AInScan() and AOutScan()</td>
</tr>
<tr>
<td>TimerOutStart()</td>
<td>TimerOut01</td>
<td>Demonstrates how to start and stop a timer square wave output. The board must have a timer output.</td>
</tr>
<tr>
<td>TimerOutStop()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToEngUnits()</td>
<td>ULA01, ULA07, ULE02, ULE03</td>
<td></td>
</tr>
<tr>
<td>VIN()</td>
<td>VIN01</td>
<td>Demonstrates how to read an A/D input channel.</td>
</tr>
<tr>
<td>VOut()</td>
<td>VOut01</td>
<td>Demonstrates how to write to a D/A output channel.</td>
</tr>
<tr>
<td>WinArrayToBuf()</td>
<td>ULA001, ULA002, ULE04</td>
<td>Demonstrates how to allocate a Windows global memory buffer for use with 32-bit scan methods.</td>
</tr>
<tr>
<td>WinBufAlloc32()</td>
<td>ClnScan01, ClnScan02</td>
<td></td>
</tr>
<tr>
<td>WinBufToArray32()</td>
<td>ClnScan01, ClnScan02</td>
<td>Demonstrates how to copy 32-bit data from a Windows global memory buffer into an array.</td>
</tr>
<tr>
<td>UL for .NET method call</td>
<td>UL for .NET example program Name</td>
<td>UL for .NET special features/notes</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>WinBufAlloc()</td>
<td>ULAI01 – ULAI06</td>
<td></td>
</tr>
<tr>
<td>WinBufFree()</td>
<td>ULAI08 – ULAI10</td>
<td></td>
</tr>
<tr>
<td>WinBufToArrray()</td>
<td>ULAI12 – ULAI14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ULA002 – ULCT03</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UL0103</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ULEV02 – ULEV04</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(ULEV04: WinBufAlloc and WinBufFree only)</td>
<td></td>
</tr>
</tbody>
</table>
Analog I/O Methods

Introduction

This chapter covers Universal Library for .NET methods that 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 methods 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 7 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>
<thead>
<tr>
<th>UL .NET settings</th>
<th>Value</th>
<th>UL .NET settings</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MccDaq.Bip20Volts</td>
<td>±20 volts(V)</td>
<td>MccDaq.Uni10Volts</td>
<td>0 to 10 V</td>
</tr>
<tr>
<td>MccDaq.Bip10Volts</td>
<td>±10 V</td>
<td>MccDaq.Uni5Volts</td>
<td>0 to 5 V</td>
</tr>
<tr>
<td>MccDaq.Bip5Volts</td>
<td>±5 V</td>
<td>MccDaq.Uni2Pt5Volts</td>
<td>0 to 2.5 V</td>
</tr>
<tr>
<td>MccDaq.Bip4Volts</td>
<td>±4 V</td>
<td>MccDaq.Uni2Volts</td>
<td>0 to 2 V</td>
</tr>
<tr>
<td>MccDaq.Bip2Pt5Volts</td>
<td>±2.5 V</td>
<td>MccDaq.Uni1Pt25Volts</td>
<td>0 to 1.25 V</td>
</tr>
<tr>
<td>MccDaq.Bip2Volts</td>
<td>±2 V</td>
<td>MccDaq.Uni1Pt67Volts</td>
<td>0 to 1.67 V</td>
</tr>
<tr>
<td>MccDaq.Bip1Pt25Volts</td>
<td>±1.25 V</td>
<td>MccDaq.Uni1Volts</td>
<td>0 to 1 V</td>
</tr>
<tr>
<td>MccDaq.Bip1Volts</td>
<td>±1 V</td>
<td>MccDaq.UniPt5Volts</td>
<td>0 to 0.5 V</td>
</tr>
<tr>
<td>MccDaq.Bip1Pt67Volts</td>
<td>±1.67 V</td>
<td>MccDaq.UniPt25Volts</td>
<td>0 to 0.25 V</td>
</tr>
<tr>
<td>MccDaq.Bip625Volts</td>
<td>±0.625 V</td>
<td>MccDaq.UniPt2Volts</td>
<td>0 to 0.2 V</td>
</tr>
<tr>
<td>MccDaq.Bip5Volts</td>
<td>±0.5 V</td>
<td>MccDaq.UniPt1Volts</td>
<td>0 to 0.1 V</td>
</tr>
<tr>
<td>MccDaq.Bip2Pt5Volts</td>
<td>±0.25 V</td>
<td>MccDaq.UniPt01Volts</td>
<td>0 to 0.01 V</td>
</tr>
<tr>
<td>MccDaq.BipPt2Volts</td>
<td>±0.2 V</td>
<td>MccDaq.UniPt02Volts</td>
<td>0 to 0.02 V</td>
</tr>
<tr>
<td>MccDaq.BipPt1Volts</td>
<td>±0.1 V</td>
<td>MccDaq.UniPt05Volts</td>
<td>0 to 0.05 V</td>
</tr>
<tr>
<td>MccDaq.BipPt05Volts</td>
<td>±0.05 V</td>
<td>MccDaq.Ma0To20</td>
<td>0 to 20 milliamperes (mA)</td>
</tr>
<tr>
<td>MccDaq.BipPt01Volts</td>
<td>±0.01 V</td>
<td>MccDaq.Ma4To20</td>
<td>4 to 20 mA</td>
</tr>
<tr>
<td>MccDaq.BipPt005Volts</td>
<td>±0.005 V</td>
<td>MccDaq.Ma2To10</td>
<td>2 to 10 mA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MccDaq.Ma1To5</td>
<td>1 to 5 mA</td>
</tr>
<tr>
<td>NotUsed</td>
<td>-1</td>
<td>MccDaq.MaPt5To2Pt5</td>
<td>0.5 to 2.5 mA</td>
</tr>
</tbody>
</table>
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 in the Universal Library User's Guide). The data returned to adData consists of just the 12-bit A/D value. The data returned to chanTags consists of just the channel numbers.

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() on some devices, the ConvertData option is not allowed. In those cases this function should be used to convert the data after the data collection is complete.

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()

For products with pretrigger implemented in hardware (most products), this function 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)

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() on some devices, the ConvertData option is not allowed. 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 \texttt{APretrig()}, it must be copied to a BASIC array with \texttt{WinBufToArray()}. Example code is provided below.

```basic
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 7 on page 208 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 7 on page 208 for a list of valid range settings.

- **dataValue**
  - Reference to data value.

**Returns:**

- An `ErrorInfo` object that indicates the status of the operation.

- `dataValue` - 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 rate at which samples are acquired, in samples 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 7 on page 208 for a list of valid range settings.
Analog I/O Methods

\textit{AInScan()}

\begin{verbatim}
memHandle

Handle for Windows buffer to store data. This buffer must have been previously
allocated with the \texttt{WinBufAlloc()} method.

options

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

Returns:

An \texttt{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 \texttt{MccDaq.ScanOptions} enumerated constants. To set a variable to one of these
constants, you must refer to the \texttt{MccDaq} object and the \texttt{ScanOptions} enumeration (\texttt{variable =
MccDaq.ScanOptions.SingleIo, variable = MccDaq.ScanOptions.DmaIo, etc.}).

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

\begin{verbatim}
SingleIo
A/D transfers to memory are initiated by an interrupt. One interrupt per
conversion. Rates attainable using SingleIo are PC-dependent and
generally less than 10 kHz. Use the default method unless you have a
reason to select a specific transfer method.

DmaIo
A/D transfers are initiated by a DMA request.

BlockIo
A/D transfers are handled in blocks (by REP-INSW for example).

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 \texttt{GetStatus} 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. 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.
\end{verbatim}
\end{verbatim}

215
Analog I/O Methods

AInScan()

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. On some devices, ConvertData may not be specified if you are using the Background option and DMA transfers. This option is ignored for the 16-bit boards.

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() with AiFunction 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() with AiFunction 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.

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.


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 (refer to board-specific information contained in the Universal Library Users Guide). In most cases, 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.

In some cases, such as with the PCI-DAS4020/12, an approximation of the rate is used to determine the size of the packets to transfer from the board. Set the rate parameter to an approximate maximum value.
**Analog I/O Methods**

**AInScan()**

**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 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**

Disables the system’s time-of-day interrupts 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, 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 the Universal Library.

**RetrigMode**

Re-arms the trigger after a trigger event is performed. With this mode, the scan begins when a trigger event occurs. When the scan completes, the trigger is re-armed to acquire the next the batch of data. You can specify the number of samples in the scan for each trigger event (described below). The RetrigMode option can be used with the Continuous option to continue arming the trigger until StopBackground() is called.
You can specify the number of samples to acquire with each trigger event. This is the trigger count (`retrigCount`). Use `SetAdRetrigCount()` to set the trigger count. If you specify a trigger count that is either zero or greater than the value of the `AINScan` `numPoints` argument, the trigger count is set to the value of `numPoints`.

Specify the Continuous option with the trigger count set to zero to fill the buffer with `numPoints` samples, re-arm the trigger, and refill the buffer upon the next trigger.

**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 `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 $8 \times 20,000 = 160,000$, but the CIO-DAS16/JR is capable of converting only 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](http://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 7 on page 208 for a list of valid A/D range settings.

- **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:

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 channel/range pairs that you have loaded into the queue.
AOOut()

Sets the value of a D/A output.

Member of the MccBoard class.

Function prototype:

**VB .NET:**
```vbnet
Public Function AOut(ByVal channel As Integer, ByVal range As MccDaq.Range, ByVal dataValue As Short) As MccDaq.ErrorInfo
```
```vbnet
Public Function AOut(ByVal channel As Integer, ByVal range As MccDaq.Range, ByVal dataValue As System(UInt16)) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo AOut(int channel, MccDaq.Range range, ushort dataValue)
```
```csharp
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 7 on page 208 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 $2^{\text{Resolution}} - 1$ 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. AOOut() 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 method 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 gain this parameter is ignored. Refer to Table 7 on page 208 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()).

- **options**
  Bit fields that control various options. Refer to the constants in the "options parameter values" section on page 222.

Returns:

An ErrorInfo object that indicates the status of the operation.

**rate** - actual sampling rate used.
Analog I/O Methods

**AOutScan()**

**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.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 number (numPoints) of D/A values, it resets to the start of the buffer and begins again. The only way to stop this operation is by calling StopBackground() with AoFunction. 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() with AoFunction to check the status of background operation. Alternatively, some boards support EnableEvent() for event notification of changes in status of Background scans. Use StopBackground() with AoFunction 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 (refer to board-specific information contained in the Universal Library Users Guide). 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. Refer to the SetTrigger() method and to board-specific information contained in the Universal Library Users Guide for details.

- **ADCClockTrig**
  Triggers a data output operation when the ADC clock starts.

- **ADCClock**
  Paces the data output operation using the ADC clock.

**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.

For products using a hardware implementation of pretrigger (most products), pretrigCount must be less than the (totalCount - 512). For these devices, 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.

For software implementations of pretrigger, pretrigCount must be less than totalCount. For these devices, triggers that occur before the requested number of pre-trigger samples are collected are ignored. Refer to board-specific information contained in the Universal Library Users Guide.

totalCount
Total number of A/D samples to collect. Specifies the total number of samples that will be collected and stored in the buffer.

For products using a hardware implementation of pretrigger (most products), 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.

For software implementations of pretrigger, totalCount must be greater than pretrigCount. For these devices, triggers that occur before the requested number of pre-trigger samples are collected are ignored. Refer to board-specific information contained in the Universal Library Users Guide.

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.
pretrigCount must also be evenly divisible by the number of channels being
scanned. If it is not, this function will adjust the number (up) to the next valid value
and return that value to the pretrigCount 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 7 on page 208 for a list of
valid range settings.

memHandle
Handle for Windows buffer to store data. This buffer must have been previously
allocated with the \WinBufAlloc() method.

For hardware trigger types, the buffer referenced by memHandle must be big
enough to hold at least totalCount + 512 integers.

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.DtConnect, variable = MccDaq.ScanOptions.ExtMemory, etc.).

ConvertData
For hardware trigger types, 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 method.
The ConvertData option is not required for software triggered types.

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() with AiFunction 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() with AiFunction to terminate the background process before it
has completed.

Call StopBackground() after normal termination of all background functions to
clear variables and flags.

For hardware trigger types, you cannot use the ConvertData option in combination
with the Background option for this method. 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:**
For hardware trigger types, 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:**

```vbnet
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
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo ATrig(int chan, MccDaq.TriggerType trigType, short trigValue, MccDaq.Range range, out short 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 7 on page 208 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 match the trigger criteria.

**Returns:**

- A **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.
VIn()

Reads an A/D input channel, and returns a voltage value. If the specified A/D board has programmable gain, then this function sets the gain to the specified range. The voltage value is returned to dataValue.

Member of the MccBoard class.

**Function prototype:**

VB .NET: 
```vbnet
Public Function VIn(ByVal channel As Integer, ByVal range As MccDaq.Range, ByRef dataValue As Single, ByVal options As MccDaq.VInOptions) As MccDaq.ErrorInfo
```

C# .NET: 
```csharp
public MccDaq.ErrorInfo VIn (System.Int32 channel, MccDaq.Range range, System.Single dataValue, MccDaq.VInOptions options)
```

**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.

- **range** 
  A/D range code. If the board has a programmable gain, it will be set according to this parameter value. Keep in mind that some A/D boards have a programmable gain feature, and others set the gain via switches on the board. In either case, the range that the board is configured for must be passed to this method. Refer to Table 7 on page 208 for a list of valid range settings.

- **dataValue** 
  Reference to the data value.

- **options** 
  Reserved for future use.

**Returns:**

An ErrorInfo object that indicates the status of the operation.

- **dataValue** - The value in volts of the A/D sample.

**options parameter values:**

- **Default** 
  Reserved for future use.
**VOut()**

Sets the value of a D/A output.

Member of the [MccBoard](#) class.

**Function prototype:**

**VB .NET:**

```vb.net
Public Function VOut(ByVal channel As Integer, ByVal range As MccDaq.Range, ByVal dataValue As Single, ByVal options As MccDaq.VOutOptions) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo VOut(System.Int32 channel, MccDaq.Range range, System.Single dataValue, MccDaq.VOutOptions options)
```

**Parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>channel</td>
<td>The D/A channel number. The maximum allowable channel depends on which type of D/A board is being used.</td>
</tr>
<tr>
<td>range</td>
<td>The D/A range code. If the board has a programmable gain, it will be set according to this parameter value. The output range of the D/A channel can be set to any of those supported by the board. Keep in mind that some D/A boards have programmable voltage ranges, and others set the voltage range via switches on the board. In either case, the range that the board is configured for must be passed to this method.</td>
</tr>
<tr>
<td>dataValue</td>
<td>The voltage value to be written.</td>
</tr>
<tr>
<td>options</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>

**Returns:**

An [ErrorInfo](#) object that indicates the status of the operation.

**options parameter values:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>
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.GetAdRetrigCount()

Gets the number of samples to acquire during each trigger event when ScanOptions.RetrigMode is enabled.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

VB .NET: Public Function GetAdRetrigCount(ByRef retrigCount As Integer) As MccDaq.ErrorInfo
C# .NET: public MccDaq.ErrorInfo GetAdRetrigCount(System.Int32 retrigCount)

Parameters:

retrigCount Specifies the number of samples to acquire per trigger event when RetrigMode is set.

Returns:

An ErrorInfo object that indicates the status of the operation.
**BoardConfig.GetBaseAdr()**

Gets 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 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` The 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:**

```vbnet
Public Function GetClock(ByRef configVal As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
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 240 for more information.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

**Function prototype:**

**VB .NET:**

```vbnet
Public Function GetDACStartup(ByVal devNum As Integer, ByVal configVal As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo GetDACStartup(int devNum, out int configVal)
```

**Parameters:**

- `devNum` The number of the DAC channel whose startup bit setting you want to get.
- `configVal` Returns the setting of the 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:**

```vbnet
Public Function GetDACUpdateMode(ByVal configVal As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo GetDACUpdateMode(out int configVal)
```

**Parameters:**

- `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.GetDeviceID()**

Returns the name that identifies the instance of a device.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

**VB .NET:**

```vbnet
Public Function GetDeviceId(ByRef configVal As String, ByRef maxLen As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo GetDeviceId(System.String configVal, System.Int32 maxLen)
```

Parameters:

- `configVal` Returns a string containing the name that identifies the device.
- `maxLen` Specifies the maximum number of bytes to read, and returns the number of bytes that were actually read.

Returns:
An ErrorInfo object that indicates the status of the operation.

**BoardConfig.GetDeviceNotes()**

Returns the device notes that are stored in the device's memory.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

**VB .NET:**

```vbnet
Public Function GetDeviceNotes(ByVal start As Integer, ByRef configVal As String, ByRef maxLen As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
```

Parameters:

- `start` The start address of the device's memory to begin reading.
- `maxLen` The maximum number of bytes to read from the device's memory. Returns the number of bytes actually read.
- `configVal` Returns the text stored in the device's memory.

Returns:
An ErrorInfo object that indicates the status of the operation.

**BoardConfig.GetDiNumDevs()**

Gets the number of digital devices on the board.
Configuration Methods and Properties

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(ByVal 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(ByVal 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.
Configuration Methods and Properties

BoardConfig.GetNumAdChans()

Gets the number of A/D channels.

Member of the eBoardConfig 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 eBoardConfig 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 eBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:

- VB .NET: Public Function GetNumExps(ByRef configVal As Integer) As MccDaq.ErrorInfo

Returns:

An ErrorInfo object that indicates the status of the operation.
**Configuration Methods and Properties**

**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:**

```vbnet
Public Function GetNumIoPorts(ByRef configVal As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
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.GetPANID()**

Returns the Personal Area Network (PAN) identifier for wireless communication.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

Function prototype:

**VB .NET:**

```vbnet
Public Function GetPANID(ByRef configVal As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo GetPANID(System.Int32 configVal)
```

**Parameters:**

`configVal` Returns a number from 0 to 65534 that identifies the Personal Area Network used for wireless communication.

**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 `InstaCal` settings. If `InstaCal` 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 7 on page 208 for a list of valid configVal settings.

Returns:
An ErrorInfo object that indicates the status of the operation.

BoardConfig.GetRFChannel()
Returns the RF channel number that a wireless device uses to communicate.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:
VB .NET: Public Function GetRFChannel(ByRef configVal As Integer) As MccDaq.ErrorInfo
C# .NET: public MccDaq.ErrorInfo GetRFChannel(System.Int32 configVal)

Parameters:
configVal Returns the number of the RF channel selected for wireless communication. Valid channel numbers are 12 to 23.

Returns:
An ErrorInfo object that indicates the status of the operation.

BoardConfig.GetRSS()
Returns the signal strength in dBm of a signal received by a remote device.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:
VB .NET: Public Function GetRSS(ByRef configVal As Integer) As MccDaq.ErrorInfo
C# .NET: public MccDaq.ErrorInfo GetRSS(System.Int32 configVal)

Parameters:
configVal Returns the received signal strength in dBm of the remote device. In general, values are negative.

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:**

```vbnet
Public Function GetUsesExps(ByVal configVal As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
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:**

```vbnet
Public Function GetWaitState(ByVal configVal As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
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.SetAdRetrigCount()**

Sets the number of samples to acquire during each trigger event when `ScanOptions.RetrigMode` is enabled.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

**Function prototype:**

**VB .NET:**

```vbnet
Public Function SetAdRetrigCount(ByVal retrigCount As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo SetAdRetrigCount(System.Int32 retrigCount)
```

**Parameters:**

`retrigCount`

Specifies the number of samples to acquire per trigger event when `RetrigMode` is set. Set to zero to use the value of the `numPoints` argument of the scan function.

**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 calling AOut() or AOutScan() after calling SetDACStartup() with the argument set to 1. Refer to the "Notes" section below for more information.
Member of the `eBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

**Function prototype:**
- **VB .NET:**
  ```vbnet
  Public Function SetDACStartup(ByVal configVal As Integer) As MccDaq.ErrorInfo
  ```
- **C# .NET:**
  ```csharp
  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 `ErrorException` 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:**
```csharp
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 `eBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

**Function prototype:**
- **VB .NET:**
  ```vbnet
  Public Function SetDACUpdateMode(ByVal devNum as Integer, ByVal configVal As Integer) As MccDaq.ErrorInfo
  ```
- **C# .NET:**
  ```csharp
  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 `ErrorException` object that indicates the status of the operation.
BoardConfig.SetDeviceID()

Sets the name that identifies the instance of a device.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:
- **VB .NET:**  
  Public Function SetDeviceId(ByVal configVal As String, ByRef maxLen As Integer) As MccDaq.ErrorInfo
- **C# .NET:**  
  public MccDaq.ErrorInfo SetDeviceId(System.String configVal, System.Int32 maxLen)

**Parameters:**
- configVal: Sets the string that contains the name identifying a device.
- maxLen: Specifies the maximum number of bytes to write, and returns the number of bytes that were actually written. For WLS Series devices, the string can contain up to 20 characters.

**Returns:**
An ErrorInfo object that indicates the status of the operation.

BoardConfig.SetDeviceNotes()

Sets the device notes to store in the device's memory.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:
- **VB .NET:**  
  Public Function SetDeviceNotes(ByVal start As Integer, ByVal configVal As String, ByRef maxLen As Integer) As MccDaq.ErrorInfo
- **C# .NET:**  

**Parameters:**
- start: The start address of the device's memory to begin writing.
- maxLen: The maximum number of bytes to write to the device's memory. Returns the number of bytes actually written.
- configVal: The text to store in the device's memory.

**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.SetPANID()
Sets the Personal Area Network (PAN) identifier used for wireless communication.

Member of the cBoardConfig class. Accessible from the MccBoard.BoardConfig property.

Function prototype:
VB .NET:  Public Function SetPANID(ByVal configVal As Integer) As MccDaq.ErrorInfo
C# .NET: public MccDaq.ErrorInfo SetPANID(System.Int32 configVal)

Parameters:
configVal  Sets the number (from 0 to 65534) that identifies the Personal Area Network used for wireless communication.

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.SetRFChannel()**
Sets the RF channel number used for wireless communications.

Member of the `cBoardConfig` class. Accessible from the `MccBoard.BoardConfig` property.

**Function prototype:**
- **VB .NET:**
  ```vbscript```
  Public Function SetRFChannel(ByVal configVal As Integer) As MccDaq.ErrorInfo
  ```vbscript```
- **C# .NET:**
  ```csharp```
  public MccDaq.ErrorInfo SetRFChannel(System.Int32 configVal)
  ```csharp```

**Parameters:**
- `configVal` Sets the number of the RF channel to use for wireless communications. Valid channel numbers are 12 to 23.

**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:**
  ```vbscript```
  Public Function SetWaitState(ByVal configVal As Integer) As MccDaq.ErrorInfo
  ```vbscript```
- **C# .NET:**
  ```csharp```
  public MccDaq.ErrorInfo SetWaitState(System.Int32 configVal)
  ```csharp```

**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:**

```vbnet
Public Readonly Property CtrConfig As MccDaq.cCtrConfig
```

**C# .NET**

```csharp
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.

```vbnet
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:

```vbnet
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:**

```vbnet
Public Function GetCtrType(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
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, 5 = event counter, 6 = scan counter, and 7 = timer 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.

```csharp
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.

```csharp
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 MccDaq.ErrorInfo) 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:**
  ```vbnet
  Public Function GetDOutMask(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo
  ```

- **C# .NET:**
  ```csharp
  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:**
  ```vbnet
  Public Function GetConfig(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo
  ```

- **C# .NET:**
  ```csharp
  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.
Configuration Methods and Properties

DioConfig.GetDevType()

Member of the \texttt{cDioConfig} class. Accessible from the \texttt{MccBoard.DioConfig} property.

Function prototype:
\begin{itemize}
\item VB .NET: \texttt{Public Function GetCurVal(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo}
\item C# .NET: \texttt{public MccDaq.ErrorInfo GetCurVal(int devNum, out int configVal)}
\end{itemize}

Parameters:
\begin{itemize}
\item devNum \hspace{1cm} Number of the digital device.
\item configVal \hspace{1cm} Current value of the digital output.
\end{itemize}

Returns:
An \texttt{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 \texttt{cDioConfig} class. Accessible from the \texttt{MccBoard.DioConfig} property.

Function prototype:
\begin{itemize}
\item VB .NET: \texttt{Public Function GetDevType(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo}
\item C# .NET: \texttt{public MccDaq.ErrorInfo GetDevType(int devNum, out int configVal)}
\end{itemize}

Parameters:
\begin{itemize}
\item devNum \hspace{1cm} Number of the digital device.
\item configVal \hspace{1cm} Constant that indicates the type of device (AuxPort, FirstPortA, etc.).
\end{itemize}

Returns:
An \texttt{ErrorInfo} object that indicates the status of the operation.

DioConfig.GetNumBits()

Gets the number of bits in the digital port.

Member of the \texttt{cDioConfig} class. Accessible from the \texttt{MccBoard.DioConfig} property.

Function prototype:
\begin{itemize}
\item VB .NET: \texttt{Public Function GetNumBits(ByVal devNum As Integer, ByRef configVal As Integer) As MccDaq.ErrorInfo}
\item C# .NET: \texttt{public MccDaq.ErrorInfo GetNumBits(int devNum, out int configVal)}
\end{itemize}

Parameters:
\begin{itemize}
\item devNum \hspace{1cm} Number of the digital device.
\item configVal \hspace{1cm} Number of bits in the digital port.
\end{itemize}

Returns:
An \texttt{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.

```vba
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.

```vba
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 **eExpansionConfig** 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 **eExpansionConfig** 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 **eExpansionConfig** 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 eExpansionConfig 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 eExpansionConfig 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 eExpansionConfig 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 `ExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

**Function prototype:**

**VB .NET:**

```vbnet
Public Function SetCjcChan(ByVal devNum As Integer, ByVal configVal As Integer) As MccDaq>ErrorInfo
```

**C# .NET:**

```csharp
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 `ExpansionConfig` class. Accessible from the `MccBoard.ExpansionConfig` property.

**Function prototype:**

**VB .NET:**

```vbnet
Public Function SetMuxAdChan1(ByVal devNum As Integer, ByVal configVal As Integer) As MccDaq>ErrorInfo
```

**C# .NET:**

```csharp
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 InstaCal.

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 260 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 connectionPins 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 260 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 InstaCal: Open InstaCal, 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 InstaCal.

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:

```vb
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:

```csharp
```

**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. Refer to the "direction, connectionPin, and polarity parameter values” section starting on page 260.

- **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 260.

- **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**

**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.
- `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.
- `DGround`: 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.).

<table>
<thead>
<tr>
<th>signalType</th>
<th>connectionPin</th>
<th>polarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdcConvert</td>
<td>AuxIn0 to AuxIn5 DsConnector</td>
<td>PositiveEdge or NegativeEdge</td>
</tr>
<tr>
<td>AdcGate</td>
<td>AuxIn0 to AuxIn5 DsConnector</td>
<td>See <code>SetTrigger</code></td>
</tr>
<tr>
<td>AdcStartTrig</td>
<td>AuxIn0 to AuxIn5 DsConnector</td>
<td></td>
</tr>
<tr>
<td>AdcStopTrig</td>
<td>AuxIn0 to AuxIn5 DsConnector</td>
<td></td>
</tr>
<tr>
<td>signalType</td>
<td>connectionPin</td>
<td>polarity</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>AdcTbSrc</td>
<td>AuxIn0 to AuxIn5</td>
<td>PositiveEdge or NegativeEdge</td>
</tr>
<tr>
<td>DacStartTrig</td>
<td>AuxIn0 to AuxIn5</td>
<td>Not assigned here.</td>
</tr>
<tr>
<td>DscTbSrc</td>
<td>AuxIn0 to AuxIn5</td>
<td>PositiveEdge or NegativeEdge</td>
</tr>
<tr>
<td>DacUpdate</td>
<td>AuxIn0 to AuxIn5</td>
<td>PositiveEdge or NegativeEdge</td>
</tr>
<tr>
<td>SyncClk</td>
<td>DsConnector</td>
<td>Not assigned here.</td>
</tr>
</tbody>
</table>

**Valid output (direction= MccDaq.SignalDirection.SignalOut) settings include:**

<table>
<thead>
<tr>
<th>signalType</th>
<th>connectionPin</th>
<th>polarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdcConvert</td>
<td>AuxIn0 to AuxIn5</td>
<td>Inverted* or NonInverted</td>
</tr>
<tr>
<td>AdcScanClk</td>
<td>AuxOut0..AuxOut2</td>
<td></td>
</tr>
<tr>
<td>AdcScanStop</td>
<td>AuxOut0..AuxOut2</td>
<td></td>
</tr>
<tr>
<td>AdcSSh</td>
<td>AuxOut0..AuxOut2</td>
<td>DsConnector</td>
</tr>
<tr>
<td>AdcStartScan</td>
<td>AuxOut0..AuxOut2</td>
<td>DsConnector</td>
</tr>
<tr>
<td>AdcStartTrig</td>
<td>AuxOut0..AuxOut2</td>
<td>DsConnector</td>
</tr>
<tr>
<td>AdcStopTrig</td>
<td>AuxOut0..AuxOut2</td>
<td>DsConnector</td>
</tr>
<tr>
<td>CtrlClk</td>
<td>AuxOut0..AuxOut2</td>
<td></td>
</tr>
<tr>
<td>Ctrl2Clk</td>
<td>AuxOut0..AuxOut2</td>
<td></td>
</tr>
<tr>
<td>DacStartTrig</td>
<td>AuxOut0..AuxOut2</td>
<td>DsConnector</td>
</tr>
<tr>
<td>DacUpdate</td>
<td>AuxOut0..AuxOut2</td>
<td>DsConnector</td>
</tr>
<tr>
<td>DGND</td>
<td>AuxOut0..AuxOut2</td>
<td></td>
</tr>
<tr>
<td>SyncClk</td>
<td>DsConnector</td>
<td>Not assigned here.</td>
</tr>
</tbody>
</table>

* *Inverted is only valid for Auxiliary Output (AuxOut) connections.

**Valid disabled settings (direction = MccDaq.SignalDirection.Disabled):**

<table>
<thead>
<tr>
<th>signalType</th>
<th>connectionPin</th>
<th>polarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DacTbSrc</td>
<td></td>
<td>Not assigned here.</td>
</tr>
<tr>
<td>SyncClk</td>
<td></td>
<td>Not assigned here.</td>
</tr>
</tbody>
</table>

**Notes:**

- You can view and edit the above timing and control configuration information from InstaCal. Open InstaCal, 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, DacTbSrc 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 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 methods:

- **AInScan()**, if the ExTrigger option is selected.
- **CInScan()**, if the ExTrigger option is selected.
- **DInScan()**, if the ExTrigger option is selected.
- **APretrig()**
- **FilePretrig()**

Member of the **MccBoard** class.

**Function prototype:**

**VB .NET:**

```vb
Public Function SetTrigger(ByVal trigType As MccDaq.TriggerType,
                          ByVal lowThreshold As Short, ByVal highThreshold As Short) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo SetTrigger(MccDaq.TriggerType trigType,
                                 short lowThreshold, short 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 264.

- **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 264.

**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.).
## Configuration Methods and Properties

### SetTrigger()

<table>
<thead>
<tr>
<th>Trigger Source</th>
<th>trigType</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog</td>
<td>GateNegHys</td>
<td>AD conversions are enabled when the external analog trigger input is more positive than <code>highThreshold</code>. AD conversions are disabled when the external analog trigger input is more negative than <code>lowThreshold</code>. Hysteresis is the level between <code>lowThreshold</code> and <code>highThreshold</code>.</td>
</tr>
<tr>
<td>Source</td>
<td>GatePosHys</td>
<td>AD conversions are enabled when the external analog trigger input is more negative than <code>lowThreshold</code>. AD conversions are disabled when the external analog trigger input is more positive than <code>highThreshold</code>. Hysteresis is the level between <code>lowThreshold</code> and <code>highThreshold</code>.</td>
</tr>
<tr>
<td>Source</td>
<td>GateAbove</td>
<td>AD conversions are enabled as long as the external analog trigger input is more positive than <code>highThreshold</code>.</td>
</tr>
<tr>
<td>Source</td>
<td>GateBelow</td>
<td>AD conversions are enabled as long as the external analog trigger input is more negative than <code>lowThreshold</code>.</td>
</tr>
<tr>
<td>Analog</td>
<td>TrigAbove</td>
<td>AD conversions are enabled when the external analog trigger makes a transition from below <code>highThreshold</code> to above. Once conversions are enabled, the external trigger is ignored.</td>
</tr>
<tr>
<td>Source</td>
<td>TrigBelow</td>
<td>AD conversions are enabled when the external analog trigger input makes a transition from above <code>lowThreshold</code> to below. Once conversions are enabled, the external trigger is ignored.</td>
</tr>
<tr>
<td>Source</td>
<td>GateInWindow</td>
<td>AD conversions are enabled as long as the external analog trigger is inside the region defined by <code>lowThreshold</code> and <code>highThreshold</code>.</td>
</tr>
<tr>
<td>Source</td>
<td>GateOutWindow</td>
<td>AD conversions are enabled as long as the external analog trigger is outside the region defined by <code>lowThreshold</code> and <code>highThreshold</code>.</td>
</tr>
<tr>
<td>Digital</td>
<td>GateHigh</td>
<td>AD conversions are enabled as long as the external digital trigger input is 5 V (logic HIGH or 1).</td>
</tr>
<tr>
<td>Source</td>
<td>GateLow</td>
<td>AD conversions are enabled as long as the external digital trigger input is 0 V (logic LOW or 0).</td>
</tr>
<tr>
<td>Source</td>
<td>TrigHigh</td>
<td>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.</td>
</tr>
<tr>
<td>Source</td>
<td>TrigLow</td>
<td>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.</td>
</tr>
<tr>
<td>Source</td>
<td>TrigPosEdge</td>
<td>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.</td>
</tr>
<tr>
<td>Source</td>
<td>TrigNegEdge</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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 `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 $2^{\text{resolution}}$. FSR is the entire span from $-\text{FS}$ to $+\text{FS}$ 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 $2^{\text{resolution}} - 1$. The formula is shown here:

$$\text{Abs} (-\text{FS} - \text{threshold in volts}) ÷ (\text{LSB}) = \text{threshold in counts}$$

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:** $\text{LSB} = 20 ÷ 2^8 = 20 ÷ 256 = 0.078125$
  
  **Calculate threshold:** $\text{Abs}(-10 - (-5)) ÷ 0.078125 = 5 ÷ 0.078125 = 64$ (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:** $\text{LSB} = 20 ÷ 2^{12} = 20 ÷ 4096 = 0.00488$
  
  **Calculate threshold:** $\text{Abs}(-10 - 1) ÷ 0.00488 = 11 ÷ 0.00488 = 2254$ (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:**
  ```vbnet
  Public Shared ReadOnly Property Version As Integer
  ``

- **C#.NET:**
  ```csharp
  public int Version [get]
  ```
Counter Methods

Introduction

This section covers Universal Library for .NET 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 a 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 (ls7266r1.pdf) 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:


Parameters:

counterNum: Counter Number (1 - n), where n is the number of counters on the board.

quadrature: Selects the resolution multiplier for 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 parameter values" section on page 269.

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 parameter values" section on page 269.

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 parameter values" section on page 269.

gateState: If gateState is set to ENABLED, then the channel INDEX input is routed to the RCNTR pin of the LS7266 chip, and is 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** (\texttt{variable = MccDaq.CountingMode.NormalMode, variable = MccDaq.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 and 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** (\texttt{variable = MccDaq.IndexMode.IndexDisabled, variable = MccDaq.IndexMode.LoadCtr}, etc.).

- **IndexDisabled**
  - The Index signal is ignored.

- **LoadCtr**
  - The channel INDEX input is routed to the LCNTR pin of the LS7266 counter chip. The counter is loaded whenever the signal occurs.

- **LoadOutLatch**
  - The channel INDEX input is routed to the LCNTR pin of the LS7266 counter chip. The current count is latched whenever the signal occurs. When this mode is selected, the **CIn()** method will return the same count value each time it is called until the Index signal occurs.

- **ResetCtr**
  - The channel INDEX input is routed to the RCNTR pin of the LS7266 counter chip. The counter is reset whenever the signal 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** (\texttt{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 pin 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 ).

c 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 parameter values" 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 C8254Mode enumeration (variable = MccDaq.C8254Mode.HighOnLastCount, variable = MccDaq.C8254Mode.LastShot, etc.).

<table>
<thead>
<tr>
<th>config parameter values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HighOnLastCount</td>
<td>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.</td>
</tr>
<tr>
<td>OneShot</td>
<td>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.</td>
</tr>
<tr>
<td>RateGenerator</td>
<td>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.</td>
</tr>
<tr>
<td>SquareWave</td>
<td>Output of counter (OUT N) is high for count &lt; 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.</td>
</tr>
<tr>
<td>SoftWareStrobe</td>
<td>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.</td>
</tr>
<tr>
<td>HardwareStrobe</td>
<td>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.</td>
</tr>
</tbody>
</table>
C8536Config()

Configures 8536 counter for desired operation. This method can only be used with 8536 counters. For more information, refer to the Zilog 8536 product specification. The document is available on our web site at www.mccdaq.com/PDFmanuals/Z8536.pdf.

Member of the MccBoard class.

Function prototype:

VB .NET: Configure for software triggering:

Public Shared Function C8536Config(ByVal counterNum As Integer, ByVal outputControl As MccDaq.C8536OutputControl, ByVal recycleMode As MccDaq.RecycleMode, ByVal trigType As MccDaq.C8536TriggerType) As MccDaq.ErrorInfo

Configure for hardware triggering; use when existing code includes MccDaq.OptionState.

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: Configure for software triggering:

public MccDaq.ErrorInfo C8536Config(int counterNum, MccDaq.C8536OutputControl outputControl, MccDaq.RecycleMode recycleMode, MccDaq.C8536TriggerType trigType)

Configure for hardware triggering; use when existing code includes MccDaq.OptionState.

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. INT32 Series boards have two chips installed, so the counterNum value may be 1 to 6.

outputControl Specifies the action of the output signal. Set it to one of the constants in the "outputControl parameter values" section on page 272.

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 initiates loading of the initial count. Counting proceeds from the initial count.

trigType Specifies the trigger type. Set it to one of the constants in the "trigType parameter values" section on page 272.

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, 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.

trigType parameter values:

All of the trigType settings are MccDaq.C8536TriggerType enumerated constants. To set a variable to one of these constants, refer to the MccDaq object and the C8536TriggerType enumeration (variable = MccDaq.C8536OutputControl.HighPulseOnTc, variable = MccDaq.C8536OutputControl.ToggleOnTc, etc.).

- **HWStartTrig**: The first trigger on the counter's trigger input initiates loading of the initial count. Counting proceeds from the initial count.
- **HWRetrig**: Every trigger on the counter's trigger input initiates loading of the initial count. Counting proceeds from the initial count.
- **SWStartTrig**: The CLoad() method initiates loading of the initial count. Counting proceeds from the initial count.
C8536Init()

Initializes the counter linking features of an 8536 counter chip. The linking of counters 1 and 2 must be accomplished prior to enabling the counters.

Refer to the Zilog 8536 product specification for a description of the hardware affected by this mode. The document is available on our web site at www.mccdaq.com/PDFmanuals/Z8536.pdf.

Member of the MccBoard class.

Function prototype:

**VB .NET:**

```vbnet
Public Function C8536Init(ByVal chipNum As Integer, ByVal ctrlOutput As MccDaq.CtrlOutput) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo C8536Init(int chipNum, MccDaq.CtrlOutput ctrlOutput)
```

**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 parameter values" 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:**
```vbnet
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:**
```csharp
```

**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 275.

- **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 275.

- **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" on page 275.

**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 ($\text{variable} = \text{MccDaq.GateControl.NoGate}$, $\text{variable} = \text{MccDaq.GateControl.AhlTcPrevCtr}$, 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
- AhlEdge: 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 ($\text{variable} = \text{MccDaq.CounterSource.TcPrevCtr}$, $\text{variable} = \text{MccDaq.CounterSource.CtrInput1}$, 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: GATE 4
- 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 ($\text{variable} = \text{MccDaq.9513OutputControl.AlwaysLow}$, $\text{variable} = \text{MccDaq.9513OutputControl.HighPulseOnTc}$, etc.).

- AlwaysLow: Inactive, Output Low
Counter Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HighPulseOnTc</td>
<td>High pulse on Terminal Count</td>
</tr>
<tr>
<td>ToggleOnTc</td>
<td>TC Toggled</td>
</tr>
<tr>
<td>Disconnected</td>
<td>Inactive, Output High Impedance</td>
</tr>
<tr>
<td>LowPulseOnTc</td>
<td>Active Low Terminal Count Pulse</td>
</tr>
<tr>
<td>3, 6, 7</td>
<td>(numeric values) Illegal</td>
</tr>
</tbody>
</table>

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:

```vbnet
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:

```csharp
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, set to 1. For a CTR10 board, set to either 1 or 2, and for a CTR20 set to 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 278.

- **compare1**
  MccDaq.CompareValue.Enabled or MccDaq.CompareValue.Disabled

- **compare2**

- **timeOfDay**
  MccDaq.TimeOfDay.Disabled, or three different enabled settings. Set it to one of the constants in the "timeOfDay parameter values" section on page 278.

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.CtrInput1, variable = MccDaq.CounterSource.CtrInput2, etc.).

<table>
<thead>
<tr>
<th>foutSource</th>
<th>9513 Data Sheet Equivalent</th>
<th>foutSource</th>
<th>9513 Data Sheet Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>CtrInput1</td>
<td>SRC 1 (Counter Input 1)</td>
<td>Gate3</td>
<td>GATE3</td>
</tr>
<tr>
<td>CtrInput2</td>
<td>SRC 2 (Counter Input 2)</td>
<td>Gate4</td>
<td>GATE4</td>
</tr>
<tr>
<td>CtrInput3</td>
<td>SRC 3 (Counter Input 3)</td>
<td>Gate5</td>
<td>GATE5</td>
</tr>
<tr>
<td>CtrInput4</td>
<td>SRC 4 (Counter Input 4)</td>
<td>Freq1</td>
<td>F1</td>
</tr>
<tr>
<td>CtrInput5</td>
<td>SRC 5 (Counter Input 5)</td>
<td>Freq2</td>
<td>F2</td>
</tr>
<tr>
<td>Gate1</td>
<td>GATE1</td>
<td>Freq3</td>
<td>F3</td>
</tr>
<tr>
<td>Gate2</td>
<td>GATE2</td>
<td>Freq4</td>
<td>F4</td>
</tr>
</tbody>
</table>

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.).

<table>
<thead>
<tr>
<th>timeOfDay</th>
<th>9513 Data Sheet Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>TOD Disabled</td>
</tr>
<tr>
<td>One</td>
<td>TOD Enabled/5 Input</td>
</tr>
<tr>
<td>Two</td>
<td>TOD Enabled/6 Input</td>
</tr>
<tr>
<td>Three</td>
<td>TOD Enabled/10 Input</td>
</tr>
</tbody>
</table>

No parameters for 9513 Data Sheet Equivalent

<table>
<thead>
<tr>
<th>No parameters for</th>
<th>9513 Data Sheet Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (FOUT on)</td>
<td>FOUT Gate</td>
</tr>
<tr>
<td>0 (Data bus matches board)</td>
<td>Data Bus Width</td>
</tr>
<tr>
<td>1 (Disable Increment)</td>
<td>Data Pointer Control</td>
</tr>
<tr>
<td>1 (BCD Scaling)</td>
<td>Scalar Control</td>
</tr>
</tbody>
</table>

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.
**CClear()**

Clears a scan counter value (sets it to zero). This method only works with counter boards that have counter scan capability.

Member of the **MccBoard** class.

**Function prototype:**

- **VB .NET:**
  ```vbnet
  Public Function CClear(ByVal counterNum As Integer) As MccDaq.ErrorInfo
  ```

- **C# .NET:**
  ```csharp
  public MccDaq.ErrorInfo CClear(int counterNum)
  ```

**Parameters:**

- **counterNum**
  The counter to clear.

  **Note:** This parameter is zero-based (the first counter number to clear is "0").

**Returns:**

- **Error code** or 0 if no errors
CConfigScan()

Configures a counter channel. This method only works with counter boards that have counter scan capability.

Member of the MccBoard class.

Function prototype:

**VB .NET:**
```vbnet
Public Function CConfigScan(ByVal counterNum As Integer, ByVal mode As MccDaq.CounterModes, ByVal debounceTime As MccDaq.DebounceTimes, ByVal debounceMode As MccDaq.DebounceModes, ByVal edgeDetection As MccDaq.EdgeDetection, ByVal tickSize As Integer, ByVal mapCounter As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo CConfigScan(int counterNum, MccDaq.CounterModes mode, MccDaq.DebounceTimes debounceTime, MccDaq.DebounceModes debounceMode, MccDaq.EdgeDetection edgeDetection, int tickSize, int mapCounter)
```

Parameters:

- **counterNum**: The counter to set up.

  **Note**: This parameter is zero-based (the first counter number to set up is "0").

- **mode**: Bit fields that control various options. All of the mode settings are MccDaq.CounterModes enumerated constants. Set it to one of the constants in the "mode parameter values" section below.

- **debounceTime**: Used to bypass the debounce mode, or to set a channel’s comparator output to one of 16 debounce times. Debounce is used to eliminate switch-induced transients typically associated with electro-mechanical devices including relays, proximity switches, and encoders.

  All of the debounceTime settings are MccDaq.DebounceTimes enumerated constants. Set it to one of the constants in the "debounceTimes parameter values" section on page 281.

- **debounceMode**: Sets the mode of the debounce module. The debounceTrigger settings are MccDaq.DebounceModes enumerated constants. Set it to one of the constants in the "debounceMode parameter values" section on page 282.

- **edgeDetection**: Determines whether the rising edge or falling edge is to be detected. The edgeDetection settings are MccDaq.EdgeDetection enumerated constants. The choices are RisingEdge and FallingEdge.

- **tickSize**: Reserved for future use.

- **mapCounter**: Used to select the mapped counter. A mapped channel is one of the counter input channels other than counterNum that can participate with the input signal of the counter defined by counterNum by gating the counter or decrementing the counter.

Returns:

- **Error code** or 0 if no errors

**mode parameter values:**

- **ClearOnRead**: The counter is cleared at the beginning of every sample.

- **StopAtMax**: The counter will stop at the top of its count. For the CIn32() method, the top of the count depends on whether the Bit32 option is used. If it is, the top of the count is FFFFFFFF hex. If not, the top of the count is FFFF hex. By default, the counter counts upward and rolls over on the 32-bit boundary.
DecrementOn

Allows the mapped channel to decrement the counter. With this option, the main
counter will increment the counter, and the mapped counter can be used to
decrement the counter. By default, the counter decrement option is set to "off".

This mode is not compatible with CIn() or CIn32(). If a counter is configured for
DecrementOn, calling CIn() or CIn32() for that counter will result in a
BADCOUNTERMODE error.

GatingOn

Selects gating "on." When "on", the counter is enabled when the mapped channel
to gate the counter is high. When the mapped channel is low, the counter is
disabled but holds the count value.

This mode is not compatible with CIn() or CIn32(). If a counter is configured for
GatingOn, calling CIn() or CIn32() for that counter will result in a
BADCOUNTERMODE error.

LatchOnMap

Causes the count to be latched by the signal on the mapped counter. By default, the
count is latched by the internal "start of scan" signal, so the count is updated each
time it’s read.

This mode is not compatible with CIn() or CIn32(). If a counter is configured for
LatchOnMap, calling CIn() or CIn32() for that counter will result in a
BADCOUNTERMODE error.

Bit32

Selects a 32-bit counter. This mode affects only CIn32() and CIn() and only when
the counter is configured for StopAtMax. Recommended for use only with
CIn32(). (Using the Bit32 option with CIn() is not very useful, since the value
returned by CIn() is only 16 bits. The effect is that the value returned by CIn()
rolls over at 64k 65,535 times before stopping.)

Encoder

Sets the specified counter to encoder mode.

EncoderModeX1

Sets the encoder measurement mode to X1.

EncoderModeX2

Sets the encoder measurement mode to X2.

EncoderModeX4

Sets the encoder measurement mode to X4.

LatchOnZ

Selects the Encoder Z mapped signal to latch the counter outputs. This allows the
user to know the exact counter value when an edge is present on another counter.

ClearOnZOn

Selects "clear on Z" on. The counter is cleared on the rising edge of the mapped (Z)
counter. By default, the "ClearOnZ" option is off, and the counter is not cleared.

debounceTimes parameter values:

Debounce500ns  Sets the counter channel’s comparator output to 500 ns.
Debounce 1500ns  Sets the counter channel’s comparator output to 1500 ns.
Debounce 3500ns  Sets the counter channel’s comparator output to 3500 ns.
Debounce 7500ns  Sets the counter channel’s comparator output to 7500 ns.
Debounce 15500ns Sets the counter channel’s comparator output to 15500 ns.
Debounce 31500ns Sets the counter channel’s comparator output to 31500 ns.
Debounce 63500ns Sets the counter channel’s comparator output to 63500 ns.
Debounce 127500ns Sets the counter channel’s comparator output to 127500 ns.
Debounce 100us   Sets the counter channel’s comparator output to 100 us.
Debounce 300us   Sets the counter channel’s comparator output to 300 us.
Debounce 700us   Sets the counter channel’s comparator output to 700 us.
Debounce 1500us  Sets the counter channel’s comparator output to 1500 us.
Debounce 3100us  Sets the counter channel’s comparator output to 3100 us.
Debounce 6300us  Sets the counter channel’s comparator output to 6300 us.
Debounce 12700us Sets the counter channel’s comparator output to 12700 us.
Debounce 25500us Sets the counter channel’s comparator output to 25500 us.

defbounceMode parameter values:

TriggerAfterStable: This mode rejects glitches and only passes state transitions after a specified period of stability (the debounce time). This mode is used with electro-mechanical devices like encoders and mechanical switches to reject switch bounce and disturbances due to a vibrating encoder that is not otherwise moving. The debounce time should be set short enough to accept the desired input pulse but longer than the period of the undesired disturbance.

CtrTriggerBeforeStable: Use this mode when the input signal has groups of glitches and each group is to be counted as one. The trigger before stable mode will recognize and count the first glitch within a group but reject the subsequent glitches within the group if the debounce time is set accordingly. In this case the debounce time should be set to encompass one entire group of glitches.
**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:**
```vbnet
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:**
```csharp
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 284.

  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.
Counter Methods

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.CtrInp, 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 the 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).
Cln32()

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

Member of the MccBoard class.

Function prototype:

VB .NET:

```vbnet
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:

```csharp
public MccDaq.ErrorInfo CIn32(int counterNum, out uint count)
public MccDaq.ErrorInfo CIn32(int counterNum, out int count)
```

Parameters:

- `counterNum`: The counter to read the 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. Cln32(): Although the CIn() and Cln32() methods perform the same operation, Cln32() is the preferred method to use.

The only difference between the two is that CIn() returns a 16-bit count value and Cln32() returns a 32-bit value. Both CIn() and Cln32() can be used, but Cln32() is required whenever you need to read count values greater than 16-bits (counts > 65535).
CInScan()

Scans a range of scan counter channels, and stores the samples in an array. This method only works with counter boards that have counter scan capability.

Member of the MccBoard class.

Function prototype:

**VB .NET:**
```vbnet
Public Function CInScan(ByVal firstCtr As Integer, ByVal lastCtr As Integer, ByVal numPoints As Integer, ByVal rate As Integer, ByVal memHandle As Integer, ByVal options As MccDaq.ScanOptions) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo CInScan(int firstCtr, int lastCtr, int numPoints, int rate, int memHandle, MccDaq.ScanOptions options)
```

**Parameters:**

- **firstCtr**
  First counter channel of the scan.
  
  This parameter is zero-based, so the first counter number is "0".

- **lastCtr**
  Last counter channel of the scan.
  
  This parameter is zero-based, so the first counter number is "0".

- **numPoints**
  Number of counter samples to collect. Specifies the total number of counter 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 / (firstCtr - lastCtr + 1).

- **rate**
  The rate at which samples are taken – the counts are latched and saved in board memory, in samples per second.
  
  Rate also returns the value of the actual rate set, which may be different from the requested rate because of pacer limitations.

- **memHandle**
  The handle for the Windows buffer to store data (Windows). This buffer must have been previously allocated with the WinBufAlloc32() method.

- **options**
  Bit fields that control various options. Set it to one of the constants in the "options parameter values" section on page 288.

**Returns:**

- **Error code** or 0 if no errors

  - **rate** – the actual sampling rate used.

  - **memHandle** – the collected counter 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.Continuous, variable = MccDaq.ScanOptions.Background, etc.).

**Background**

When the Background option is used, control returns immediately to the next line in your program, and the data collection from the counters into the buffer continues in the background. If the Background option is not used, the CInScan() method does not return to your program until all of the requested data has been collected and returned to the buffer.

Use `GetStatus()` to check on the status of the background operation. Use `StopBackground()` with CtrFunction to terminate the background process before it has completed. Execute `StopBackground()` 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 StopBackground() with CtrFunction. Normally, you should use this option with Background so that your program regains control.

**ExtTrigger**

If this option is specified, sampling does not begin until the trigger condition is met. You can set the trigger condition to rising edge, falling edge, or the level of the digital trigger input with the `SetTrigger()` method. Refer to board-specific information in the UL User's Guide.

**ExtClock**

If this option is specified, 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 (refer to board-specific information in the UL User's Guide). 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.
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)
Counter Methods

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:**

```vbnet
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:**

```csharp
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 number of the counter whose status bits you want to read. 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 is satisfied or StopBackground() with CtrFunction 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     Handle for Windows buffer. If intCount is non-zero, the buffer referenced by memHandle must be of sufficient size to hold (intCount * Number of Counters) points.

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 buffer referenced by memHandle must be able to hold (intCount * Number of Counters) points.

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

Allocate the proper buffer size for non-zero IntCount settings

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

- If intCount = 0, the functionality is unchanged.
TimerOutStart()

Starts a timer square wave output. Use TimerOutStop() to stop the output. This method only works with counter boards that have a timer-type counter.

Member of the MccBoard class.

Function prototype:

VB .NET: 
Public Function TimerOutStart(ByVal timerNum As Integer, ByRef frequency As Double) As MccDaq.ErrorInfo

C# .NET: 
public MccDaq.ErrorInfo TimerOutStart(int timerNum, double frequency)

Parameters:

 timerNum The timer to output the square wave from. Valid values are zero up to the number of timers on the board – 1.
 frequency The desired square wave frequency. The timers clock will be divided down by integer values to produce the frequency. The actual frequency output will be returned. Valid values are dependent on the timer's clock and the timer resolution.

Returns:

 Error code or 0 if no errors

 frequency – the actual frequency set.
TimerOutStop()

Stops a timer square wave output. Use TimerOutStart() to start the output. This method only works with counter boards that have a timer-type counter.

Member of the MccBoard class.

Function prototype:

**VB .NET:**
```vbnet
Public Function TimerOutStop(ByVal timerNum As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo TimerOutStop(int timerNum)
```

**Parameters:**
- **timerNum**
  - The timer to stop. Valid values are zero up to the number of timers on the board – 1.

**Returns:**
- Error code or 0 if no errors
Data Logger Methods and Property

Introduction

This section covers Universal Library for .NET methods and a file name property used to read and convert data logged to a binary file from a data acquisition product equipped with data logging functionality. The data is typically logged to a CompactFlash® memory card, which may then be inserted into a media reader for reading and conversion using these methods.

Data is stored in a binary file. The data may consist of analog data, CJC temperature data, digital I/O data, time stamped data, and information about the device configuration. You can use the data logger methods and property to read this information, apply conversions to the data, and convert the files to a comma separated values (.CSV) text file or another specified text file format.
**FileName property**

Returns the file name associated with the current instance of the DataLogger class.

Member of the DataLogger class.

**Property prototype:**

- **VB .NET:**
  ```vbnet
  Public Shared ReadOnly Property DataLogger As String
  ```

- **C# .NET:**
  ```csharp
  public string FileName [get]
  ```
ConvertFile()

Converts a binary log file to a comma-separated values (.CSV) text file or another text file format that you specify.

Member of the DataLogger class.

Function prototype:

**VB .NET:**
```vbnet
Public Function ConvertFile(ByVal destFileName As String, ByVal startSample As Integer, ByVal count As Integer, ByVal delimiter As MccDaq.FieldDelimiter) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo ConvertFile(string destFileName, int startSample, int count, MccDaq.FieldDelimiter delimiter)
```

Parameters:

- **destFileName**
  The name and destination path of the converted file. Use the file extension of the file type that you want to create.

- **startSample**
  The first sample to read.

- **count**
  The number of samples to read.

- **delimiter**

Returns:

An ErrorInfo object that indicates the status of the operation.

Notes:

- Time stamp data is stored according to the timeZone preference and timeFormat preference. Refer to SetPreferences() on page 314.

- Time stamps in the converted file may be in either 12-hour or 24-hour format based on the value of the timeFormat preference. Time stamps can optionally be converted to local time based on the value of the timeZone preference.

- AI temperature data is returned according to the units preference. Refer to SetPreferences() on page 314.

- The units preference is only applied to the AI data if the data was logged as temperature data. Refer to GetAllInfo() on page 300. This value is ignored if the AI data was logged as raw data.

- The units preference is always applied to CJC data, since it is always logged as temperature data.

- If the destFileName argument ends with a .CSV extension, the delimiter parameter must be set to MccDaq.FieldDelimiter.Comma. Otherwise, an INVALIDDELIMITER error is returned.

- You can open a comma-separated values text file (.CSV) directly in Microsoft Excel. Text files with extensions other than .CSV can only be imported into Excel.
GetAIChannelCount()

Retrieves the total number of analog input channels logged in a binary file.

Member of the DataLogger class.

Function prototype:
- VB .NET: Public Function GetAIChannelCount(ByRef aiCount As Integer) As MccDaq.ErrorInfo
- C# .NET: public MccDaq.ErrorInfo GetAIChannelCount(ref int aiCount)

Parameters:
- aiCount: The number of analog input channels logged in the file.

Returns:
An ErrorInfo object that indicates the status of the operation.

aiCount – Returns the number of analog input channels logged in the binary file.
GetAllInfo()

Retrieves the channel number and unit value of each analog input channel logged in a binary file.

Member of the DataLogger class.

Function prototype:

**VB .NET:**

```vbnet
Public Function GetAllInfo(ByRef channelNumbers As Integer, ByRef units As MccDaq.LoggerUnits, As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo GetAllInfo(ref int channelNumbers, ref MccDaq.LoggerUnits units)
```

Parameters:

- **channelNumbers**
  An array that contains the analog input channel numbers logged in the file.
- **units**
  An array that contains the unit values set by the device in InstaCal for each analog input channel logged in the file.
  The units settings are MccDaq.LoggerUnits enumerated constants. Choices are MccDaq.LoggerUnits.Temperature and MccDaq.LoggerUnits.Raw.

Returns:

An ErrorInfo object that indicates the status of the operation.

channelNumbers - **Returns the analog input channel numbers logged in the binary file.**

units - **Returns the unit value set by the device in InstaCal for each analog input channel logged in the binary file (MccDaq.LoggerUnits.Temperature or MccDaq.LoggerUnits.Raw.)**
GetCJCInfo()

Retrieves the number of CJC temperature channels logged in a binary file.

Member of the DataLogger class.

Function prototype:

**VB .NET:**
```vbnet
Public Function GetCJCInfo(ByRef cjcCount As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo GetCJCInfo(ref int cjcCount)
```

**Parameters:**
- **cjcCount**
  - The number of CJC temperature channels logged in the file.

**Returns:**
- An ErrorInfo object that indicates the status of the operation.
- **cjcCount** – Returns the number of CJC temperature channels logged in the binary file.
GetDIOInfo()

Retrieves the number of digital I/O channels logged in a binary file.

Member of the DataLogger class.

Function prototype:

VB .NET:    Public Function GetDIOInfo(ByRef dioCount As Integer) As MccDaq.ErrorInfo
C# .NET:    public MccDaq.ErrorInfo GetDIOInfo(ref int dioCount)

Parameters:

dioCount    The number of digital I/O channels logged in the file.

Returns:

An ErrorInfo object that indicates the status of the operation.

dioCount – Returns the number of digital I/O channels logged in the binary file.
GetFileInfo()

Gets file information from the file associated with the current instance of the DataLogger.

Member of the DataLogger class.

Function prototype:

VB .NET: Public Function GetFileInfo(ByVal version As Integer, ByVal size As Integer) As MccDaq.ErrorInfo

C# .NET: public MccDaq.ErrorInfo GetFileInfo(ref int version, ref int size)

Parameters:

version The version level of the file.
size The size in bytes of the file.

Returns:

An ErrorInfo object that indicates the status of the operation.

version – Returns the version level of the binary file.
size – Returns the size in bytes of the binary file.
GetFileName()

Retrieves the name and path of the \( n \)th file in the directory containing binary log files.

Member of the DataLogger class.

Function prototype:

**VB .NET:**

```vbnet
Public Shared Function GetFileName(ByVal fileNumber As Integer, ByRef path As String, ByRef fileName As String) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public static MccDaq.ErrorInfo GetFileName(int fileNumber, ref string path, ref string fileName)
```

Parameters:

- **fileNumber**
  - Index of the file whose name you want to return. Specify one of the following:
    - The number \( n \) that represents the location of the file in the directory (where \( n = 0, 1, 2, \) and so on), or
    - `MccService.GetFirst` – get the first file in the directory, or
    - `MccService.GetNext` – get the next file in the directory, based on the current index.
  - This parameter is the index of the file in the directory, and is not part of the filename.

- **path**
  - The full path of the directory containing the log files.

- **fileName**
  - The full path and name of the binary file. The path must be NULL terminated and cannot be longer than 256 characters.

Returns:

- An ErrorInfo object that indicates the status of the operation.
- **fileName** – Returns the file name and path of the binary file.

Notes:

To access all of the files in a directory, first call `GetFileName()` with the `fileNumber` set to `MccService.GetFirst`, then again with the `fileNumber` set to `MccService.GetNext` until the method returns the error code `NOMOREFILES`. 
GetPreferences()

Retrieves API preference settings for time stamp data, analog temperature data, and CJC temperature data. Returns the default values unless changed using GetPreferences().

Member of the DataLogger class.

Function prototype:

VB .NET:
Public Shared Function GetPreferences(ByVal timeFormat As MccDaq.TimeFormat, ByVal timeZone As MccDaq.TimeZone, ByVal units As MccDaq.TempScale) As MccDaq.ErrorInfo

C# .NET:
public MccDaq.ErrorInfo GetPreferences(ref MccDaq.TimeFormat timeFormat, ref MccDaq.TimeZone timeZone, ref MccDaq.TempScale units)

Parameters:

timeFormat
Returns the format used to display time stamp data.

All of the timeFormat settings are MccDaq.TimeFormat enumerated constants. Choices are MccDaq.TimeFormat.12Hour (for example 2:32:51 PM) and MccDaq.TimeFormat.TwentyFourHour (for example 14:32:51).

timeZone
Returns the time zone to apply to time stamp data.

All of the timeZone settings are MccDaq.TimeZone enumerated constants. Choices are MccDaq.TimeZone.Local and MccDaq.TimeZone.GMT.

units
Returns the unit to use when converting temperature data. This value is ignored if raw data values are logged.

All of the units settings are MccDaq.TempScale enumerated constants. Choices are MccDaq.TempScale.Celsius, MccDaq.TempScale.Fahrenheit, and MccDaq.TempScale.Kelvin.

Returns:

An ErrorInfo object that indicates the status of the operation.

timeFormat – Returns the format to apply to time stamp data from API functions that return time data.

timeZone – Returns the time zone to apply to time stamp data from API functions that return time data.

units – Returns the unit to use when converting temperature data from API functions that return temperature data.
GetSampleInfo()

Retrieves the sample interval, sample count, and the date and time of the first data point in a binary file.

Member of the DataLogger class.

Function prototype:

VB .NET:
```vbnet
Public Function GetSampleInfo(ByRef sampleInterval As Integer, ByRef sampleCount As Integer, ByRef startDate As Integer, ByRef startTime As Integer) As MccDaq.ErrorInfo
```

C# .NET:
```csharp
public MccDaq.ErrorInfo GetFileInfo(ref int sampleInterval, ref int sampleCount, ref int startDate, ref int startTime)
```

Parameters:

- **sampleInterval**: The time, in seconds, between samples.
- **sampleCount**: The number of samples contained in the file.
- **startDate**: The date of the first data point logged in the file. Date values are packed in the following format:
  - Byte 0: day
  - Byte 1: month
  - Byte 2-3: year
- **startTime**: The time when the first data point was logged in the file. Time values are packed in the following format:
  - Byte 0: seconds
  - Byte 1: minutes
  - Byte 2: hours
  - Byte 3: 0xff = 24hour format
  - 0x0 = AM
  - 0x1 = PM

Returns:

- An ErrorInfo object that indicates the status of the operation.
- **sampleInterval**: Returns the time, in seconds, between samples.
- **sampleCount**: Returns the number of samples in the file.
- **startDate**: Returns the date of the first data point logged in the file.
- **startTime**: Returns the time when the first data point was logged in the file.

Notes:

Time stamped data is returned according to the timeZone and timeFormat preferences. Refer to SetPreferences() on page 314.
ReadAIChannels()

Retrieves analog input data from a binary file, and stores the values in an array.

Member of the DataLogger class.

Function prototype:

**VB .NET:**

```vbnet
Public Function ReadAIChannels(ByVal startSampleAs Integer, ByVal count Integer, ByRef aiChannels As Single) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo ReadAIChannels(int startSample, int count, ref float[] aiChannels)
```

**Parameters:**

- `startSample` The first sample to read from the binary file.
- `count` The number of samples to read from the binary file.
- `aiChannels` Receives the analog input values.

**Returns:**

- An ErrorInfo object that indicates the status of the operation.
- `aiChannels` - Returns the analog input values logged in the file.

**Notes:**

The units of the analog input data that is returned is set by the value of the Units preference. Refer to `SetPreferences()` on page 314.

The Units preference is only applied if the logged data is temperature data. This value is ignored if the data logged is raw.

**Analog array:**

The user is responsible for allocating the size of the analog data array, and ensuring that it is large enough to hold the data that will be returned. You can calculate the array allocation using the `SampleCount` value from `GetSampleInfo()`, and the `AICount` value from `GetAIChannelCount()`:

```plaintext
float* aiChannels = new float[sampleCount * aiCount];
```

The figure below shows the layout of the analog array, and how the elements should be indexed.

```
CH0  |  CH1  |  ...  |  CHn  |  CH0  |  CH1  |  ...  |  CHn  |  ...  |  CH0  |  CH1  |  ...  |  CHn  

Start of 1st sample  |  Start of 2nd sample  |  Start of last sample
```

Where \( n \) is \((\text{numberOfChannels} - 1)\). \( \text{CH0} - \text{CHn} \) refer to the channels in the array, not the input channels of the device.
For example, assume that all of the even number input channels are logged. The analog array channels are mapped as shown here:

<table>
<thead>
<tr>
<th>Array Channel</th>
<th>Device Input Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Use the following code fragment to access the elements of the analog array:

```c
for (i=0; i<numberOfSamples; i++)
{
    for (j=0; j<numberOfAIChannels; j++)
    {
        a = analogArray[(i *numberOfAIChannels) + j];
    }
}
```

where

- the `numberOfSamples` is set by the `sampleCount` value from `GetSampleInfo()`.
- the `numberOfAIChannels` is set by the `aiCount` value from `GetAIChannelCount()`.
ReadCJCChannels()

Retrieves CJC temperature data from a binary file, and stores the values in an array.

Member of the DataLogger class.

Function prototype:

VB .NET:

Public Function ReadCJCChannels(ByVal startSampleAs Integer, ByVal count Integer, ByRef cjcChannels As Single) As MccDaq.ErrorInfo

C# .NET:

public MccDaq.ErrorInfo ReadCJCChannels(int startSample, int count, ref float[] cjcChannels)

Parameters:

startSample  The first sample to read from the binary file.
count  The number of samples to read from the binary file.
cjcChannels  Receives the CJC temperature values.

Returns:

An ErrorInfo object that indicates the status of the operation.

cjcChannels - Returns the CJC temperature values logged in the file.

Notes:

The unit of the CJC temperature data that is returned is set by the value of the Units preference. Refer to SetPreferences() on page 314.

The Units preference is only valid if the logged data is temperature data. This value is ignored if the data logged is raw.

CJC array:

The user is responsible for allocating the size of the CJC array, and ensuring that it is large enough to hold the data that will be returned. You can calculate the array allocation using the SampleCount value from GetSampleInfo(), and the cjcCount value from GetCJCInfo():

```csharp
float* cjcChannels = new float[SampleCount * CJCCount];
```

The figure below shows the layout of the CJC array, and how the elements should be indexed.

The user is responsible for allocating the size of the CJC array, and ensuring that it is large enough to hold the data that will be returned. You can calculate the array allocation using the SampleCount value from GetSampleInfo(), and the cjcCount value from GetCJCInfo():

```csharp
float* cjcChannels = new float[SampleCount * CJCCount];
```

The figure below shows the layout of the CJC array, and how the elements should be indexed.

where \( n \) is \((CJCCount - 1)\)
Use the following code fragment to access the elements of the CJC array.

```c
for (i=0; i<numberOfSamples; i++)
{
    for (j=0; j<numberOfCJCChannels; j++)
    {
        c = cjcArray[(i * numberOfCJCChannels) + j];
    }
}
```

where

- `numberOfSamples` is set by the `sampleCount` value from `GetSampleInfo()`.
- `numberOfCJCChannels` is set by the `cjcCount` value from `GetCJCInfo()`.
ReadDIOChannels()

Retrieves digital I/O channel data from a binary file, and stores the values in an array.

Member of the DataLogger class.

Function prototype:

**VB .NET:**

```vbnet
Public Function ReadDIOChannels(ByVal startSampleAs Integer, ByVal count Integer, ByRef dioChannels As Single) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo ReadDIOChannels(int startSample, int count, ref float[] dioChannels)
```

**Parameters:**

- `startSample`: The first sample to read from the binary file.
- `count`: The number of samples to read from the binary file.
- `dioChannels`: Receives the DIO channel values.

**Returns:**

- An ErrorInfo object that indicates the status of the operation.
- `dioChannels`: Returns the DIO channel values logged in the file.

**DIO array:**

The user is responsible for allocating the size of the DIO array, and ensuring that it is large enough to hold the data that will be returned. You can calculate the array allocation using the `SampleCount` value from `GetSampleInfo()` and the `dioCount` value from `GetDIOInfo()`:

```csharp
float* dioChannels = new float[sampleCount * dioCount];
```

The figure below shows the layout of the DIO array, and how the elements should be indexed.

<table>
<thead>
<tr>
<th>D0</th>
<th>D1</th>
<th>...</th>
<th>Dn</th>
<th>E0</th>
<th>E1</th>
<th>...</th>
<th>En</th>
<th>CO</th>
<th>C1</th>
<th>...</th>
<th>Cn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start of 1st sample</td>
<td>Start of 2nd sample</td>
<td>Start of last sample</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where `n` is `(dioCount - 1)`

Use the following code fragment to access the elements of the DIO array:

```csharp
for (i=0; i<numberOfSamples; i++)
{
    for (j=0; j<numberOfDIOChannels; j++)
    {
        d = dioArray[(i * numberOfDIOChannels) + j];
    }
}
```

where

- `numberOfSamples` is set by the `sampleCount` value from `GetSampleInfo()`.
- `numberOfDIOChannels` is set by the `dioCount` value from `GetDIOInfo()`.
ReadTimeTags()

Retrieves date and time values logged in a binary file. This method stores date values in the dateTags array, and time values in the timeTags array.

Member of the DataLogger class.

Function prototype:

**VB .NET:**

```vbnet
Public Function ReadTimeTags(ByVal startSample As Integer, ByVal count Integer, ByRef dateTags As Integer, ByRef timeTags As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo ReadTimeTags(int startSample, int count, ref int[] dateTags, ref int[] timeTags)
```

Parameters:

- **startSample**
  The first sample to read from the binary file.

- **count**
  The number of samples to read from the binary file.

- **dateTags**
  Receives the date tag values. Dates are packed in the following format:
  - Byte 0: day
  - Byte 1: month
  - Byte 2 - 3: year

- **timeTags**
  Receives the time tag values. Times are packed in the following format:
  - Byte 0: seconds
  - Byte 1: minutes
  - Byte 2: hours
  - Byte 3: 0xff = 24-hour format
  - 0x0 = AM
  - 0x1 = PM

Returns:

- An ErrorInfo object that indicates the status of the operation.
- dateTags - Returns the date value for each sample logged in the file.
- timeTags - Returns the time value for each sample logged in the file.

Notes:

Time stamped data is stored according to the timeZone preference and the timeFormat preference. Refer to SetPreferences() on page 314.

Time stamped data is logged in the file if InstaCal is configured to do so. If time stamps are not logged, the time array is filled with values calculated from the file header information.

Array size:

The user is responsible for allocating the size of the date and time arrays, and ensuring that they are large enough to hold the data that is returned. You can calculate the array allocation using the sampleCount value from GetSampleInfo() on page 306.

```csharp
int* dates = new int[sampleCount];
int* times = new int[sampleCount];
```
**dateTags array**

The figure below shows the layout of the dateTags array, and how the elements should be indexed.

```
D0  D1  D2  ...  Dn
```

where: n is (numberOfSamples − 1)

Each sample has only one date. Use the following code fragment to access the elements of the dateTags array:

```java
for (i=0; i<numberOfSamples; i++)
{
    d = dateTagsArray[i];
}
```

**timeTags array**

The figure below shows the layout of the timeTags array, and how the elements should be indexed.

```
T0  T1  T2  ...  Tn
```

where: n is (numberOfSamples − 1)

Each sample has only one time stamp. Use the following code fragment to access the elements of the timeTags array:

```java
for (i=0; i<numberOfSamples; i++)
{
    t = timeTagsArray[i];
}
```
SetPreferences()

Sets preferences for returned time stamped data, analog temperature data, and CJC temperature data.

Member of the DataLogger class.

Function prototype:

VB .NET:  
Public Shared Function SetPreferences(ByVal timeFormat As MccDaq.TimeFormat, ByVal timeZone As MccDaq.TimeZone, ByVal units As MccDaq.TempScale) As MccDaq.ErrorInfo

C# .NET:  
public MccDaq.ErrorInfo SetPreferences(MccDaq.TimeFormat timeFormat, ref MccDaq.TimeZone timeZone, ref MccDaq.TempScale units)

Parameters:

timeFormat  
Specifies the time format to apply when returning time stamped data (when using ReadTimeTags() for example).

All of the timeFormat settings are MccDaq.TimeFormat enumerated constants. Choices are MccDaq.TimeFormat.12Hour (for example 2:32:51) and MccDaq.TimeFormat.TwentyFourHour (for example 14:32:51).

timeFormat defaults to MccDaq.TimeFormat.12Hour.

timeZone  
Specifies whether to convert time stamped data that is returned (when using ReadTimeTags() for example) to the local time zone or to return the time stamps as they are stored in the file (in the GMT time zone).

All of the timeZone settings are MccDaq.TimeZone enumerated constants. Choices are MccDaq.TimeZone.Local and MccDaq.TimeZone.GMT.

timeZone defaults to MccDaq.TimeZone.Local.

units  
Specifies the unit for analog data. This value is ignored if counts are logged.

All of the Units settings are MccDaq.TempScale enumerated constants. Choices are MccDaq.TempScale.Celsius, MccDaq.TempScale.Fahrenheit, and MccDaq.TempScale.Kelvin.

units defaults to MccDaq.TempScale.Fahrenheit.

Returns:

An ErrorInfo object that indicates the status of the operation.

Notes:

- The timeFormat and timeZone preferences are applied to all time data returned using API methods that return time data.
- The units preference specifies the temperature scale that the API applies when reading and converting analog, CJC, and time stamped data.
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.
**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:**

```vbnet
Public Function DBitIn(ByVal portType As MccDaq.DigitalPortType,
                        ByVal bitNum As Integer, ByRef bitValue As MccDaq.DigitalLogicState) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
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.

- **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.

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:**

```vbnet
Public Function DConfigBit(ByVal portNum As MccDaq.DigitalPortType, ByVal bitNum As Integer, ByVal direction As MccDaq.DigitalPortDirection) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
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. Refer to the board's hardware User Guide for details of chip operation.

Member of the MccBoard class.

Function prototype:

**VB .NET:**
```vbnet
Public Function DConfigPort(ByVal portNum As MccDaq.DigitalPortType, ByVal direction As MccDaq.DigitalPortDirection) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
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; consult your board-specific documentation.

- **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 
DigitalOut, this method will provide readback of the last output value.

Member of the MccBoard class.

Function prototype:

**VB .NET:**
```vbnet
Public Function DIn(ByVal portNum As MccDaq.DigitalPortType, ByRef dataValue As Short) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo DIn(MccDaq.DigitalPortType portNum, out ushort 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 
- **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- 
**DInScan()**

Performs multiple reads of a 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:**

```vbnet
Public Function DInScan(ByVal portNum As MccDaq.DigitalPortType , ByVal numPoints As Integer, ByVal rate As Integer, ByVal memHandle As Integer, ByVal options As MccDaq.ScanOptions ) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
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. This buffer must have been previously allocated with the **WinBufAlloc()** or **WinBufAlloc32()** method.
- **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.
- **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 memHandle.

  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 memHandle will continue in the background. Use GetStatus() with DiFunction to check on the status of the background operation. Use StopBackground() with DiFunction 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 by calling StopBackground() with DiFunction. 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 information in the *Universal Library User’s Guide*). When this option is used, the rate parameter is ignored. The transfer rate is dependent on the trigger signal.

ExtTrigger

If this option is used then the scan will not begin until the signal on the trigger input line meets the trigger criteria.

WordXfer

Normally this method reads a single (byte) port. If WordXfer is specified, 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 the buffer.

When WordXfer is used, it is generally required to set portNum to FirstPortA.

**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](http://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:**

```vbnet
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:**

```csharp
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.

- **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](http://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:**

```vbnet
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:**

```csharp
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 (usually 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 parameter values" 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, the DOutScan() method will not return control 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 memHandle will continue in the background. Use **GetStatus()** with DoFunction to check on the status of the background operation. Use **StopBackground()** with DoFunction 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()** with DoFunction. 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 information contained in the Universal Library Users Guide). 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, it will write two adjacent ports as the low and high byte of a single array element in dataBuffer.

When WordXfer is used, it is generally required to set portNum to FirstPortA.

NonStreamedIO

When this option is used, you can output non-streamed data to a specific DAC output channel.

To load the data output buffer into the device’s internal output FIFO, the aggregate size of the data output buffer must be \( \leq \) the size of the internal data output FIFO in the device. Once the sample data are transferred or downloaded to the device, the device is responsible for outputting the data. You can’t make any changes to the output buffer once the output begins.

With NonStreamedIO mode, you do not have to periodically feed output data through the program to the device for the data output to continue. However, the size of the buffer is limited.

ADCClockTrig

Triggers a data output operation when the ADC clock starts.

ADCClock

Paces the data output operation using the ADC clock.

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.
- NonStreamedIO can only be used with the number of samples (count) set equal to the size of the FIFO or less.
- Transfer method may not be specified. DMA is used.
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 errorHandling 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 ErrorInfo object. If no error occurred, an ErrorInfo 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 ErrorInfo object. If an error occurs, an ErrorInfo 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.
Memory Board Methods

Use the methods 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 methods to retrieve it.
**MemRead()**

Reads data from a memory board into an array. Member of the [MccBoard](#) class.

**Function prototype:**

**VB .NET:**

```vbnet
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:**

```csharp
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 `numPoints = 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:

```csharp
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

C# .NET: public MccDaq.ErrorInfo MemReadPretrig(out short 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 numPoints = 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(DataBuffer, 0, 100000)
    DaqBoard0.MemReadPretrig(DataBuffer, FROMHERE, 1000000)
    DaqBoard0.MemReadPretrig(DataBuffer, FROMHERE, 1000000)

DT-Connect Conflicts - The MemReadPretrig() 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 MemReadPretrig() 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:**
```vbnet
Public Function MemWrite(ByVal dataBuffer As Short, ByVal firstPoint As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo MemWrite(short dataBuffer, int firstPoint, int numPoints)
```

**Function prototype:**
```csharp
public MccDaq.ErrorInfo MemWrite(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 numPoints = 50.
- **numPoints**: Number of data points (words) to write.

Returns:

An ErrorInfo object that indicates the status of the operation.

Notes:

To write large amounts 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:

```csharp
DaqBoard1.MemWrite(DataBuffer, 0, 100000)
DaqBoard1.MemWrite(DataBuffer, FROMHERE, 100000)
DaqBoard1.MemWrite(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 DTCOMNECT + BACKGROUND options). You cannot call MemWrite() until the AInScan() has completed. If you do, you will get a DTACTIVE error.
Revision Control Methods

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:**
  ```vbnet
  Public Shared Function DeclareRevision(ByRef revNum As Single) As MccDaq.ErrorInfo
  ```

- **C# .NET:**
  ```csharp
  public static MccDaq.ErrorInfo DeclareRevision(ref float revNum)
  ```

**Parameters:**

- `revNum` Revision number of the Universal Library used 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 properties and methods 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.
- **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 7 on page 208 for a list of valid range settings. Refer to board-specific information in the Universal Library User’s Guide 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.
- **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.
- **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, 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 website at [www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf](http://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. Refer to board-specific information in the *Universal Library User’s Guide* to determine if your board supports **FileAInScan()** and/or **FilePretrig()**.

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 7 on page 208 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, ByVal pretrigCount As Integer, ByVal totalCount As Integer, ByVal 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 7 on page 208 for a list of valid range settings. Refer to board-specific information in the *Universal Library User's Guide* 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.

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.

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, and returns the data in a one-dimensional or two-dimensional array. When `FileAInScan()` or `FilePretrig()` fills the streamer file, this method returns the content of that file. Refer to information on your board in the *Universal Library User's Guide* to determine if your board supports `FileAInScan()` and/or `FilePreTrig()`.

Member of the [MccService](#) class.

**Function prototype:**

<table>
<thead>
<tr>
<th>VB .NET</th>
<th>C# .NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns a one-dimensional array of short values:</td>
<td>Returns a one-dimensional array of short values:</td>
</tr>
<tr>
<td><code>Public Shared Function FileRead(ByVal fileName As String, ByVal firstPoint As Integer, ByRef numPoints As Integer, ByRef dataBuffer As Short) As MccDaq.ErrorInfo</code></td>
<td><code>public static MccDaq.ErrorInfo FileRead(string fileName, int firstPoint, ref int numPoints, out short dataBuffer)</code></td>
</tr>
<tr>
<td>Returns a one-dimensional array of System.UInt16 values:</td>
<td>Returns a one-dimensional array of System.UInt16 values:</td>
</tr>
<tr>
<td><code>Public Shared Function FileRead(ByVal fileName As String, ByVal firstPoint As Integer, ByRef numPoints As Integer, ByRef dataBuffer As System.UInt16) As MccDaq.ErrorInfo</code></td>
<td><code>public static MccDaq.ErrorInfo FileRead(string fileName, int firstPoint, ref int numPoints, out ushort dataBuffer)</code></td>
</tr>
<tr>
<td>Returns a two-dimensional array of double values:</td>
<td>Returns a two-dimensional array of double values:</td>
</tr>
<tr>
<td><code>Public Shared Function FileRead(ByVal fileName As String, ByVal firstPoint As Integer, ByRef numPoints As Integer, ByRef dataBuffer As Double[,], ByVal numChannels As Integer) As MccDaq.ErrorInfo</code></td>
<td><code>public static MccDaq.ErrorInfo FileRead(string fileName, int firstPoint, ref int numPoints, out double dataBuffer[,], int numChannels)</code></td>
</tr>
</tbody>
</table>

**Parameters:**

- **filename**  
  Name of the streamer file.
- **firstPoint**  
  Index of the first point to read.
- **numPoints**  
  The number of points to read from the file.
- **dataBuffer**  
  Reference to the array in the data buffer that data is read into.
- **numChannels**  
  The number of channels to read into `dataBuffer`.

**Returns:**

An [ErrorInfo](#) object that indicates the status of the operation.

- `dataBuffer` - data read from a file.
- `numPoints` - number of points actually read.
- `numPoints` 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.

Loading portions of files: The file may contain much more data than can fit in dataBuffer. In those cases, use numPoints 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.
Synchronous I/O Methods

Introduction

Use the methods discussed in this chapter to synchronously acquire and output analog, digital, counter, and temperature data. These functions can be used with hardware equipped with synchronous input and output capability.
DaqInScan()

Scans analog, digital, counter, and temperature input channels synchronously, and stores the samples in an array. This method only works with boards that support synchronous input.

Member of the MccBoard class.

Function prototype:

VB .NET:  
Public Function DaqInScan(ByVal chanArray As Short(), ByVal chanTypeArray As MccDaq.ChannelType(), ByVal gainArray As MccDaq.Range(), ByVal chanCount As Integer, ByRef rate As Integer, ByRef pretrigCount As Integer, ByRef totalCount As Integer, ByVal memHandle As Integer, ByVal options MccDaq.ScanOptions) As MccDaq.ErrorInfo

Visual Basic:  
Public ErrorInfo DaqInScan(ByVal chanArray As Short(), ByVal chanTypeArray As MccDaq.ChannelType(), ByVal gainArray As MccDaq.Range(), ByVal chanCount As Integer, ByRef rate As Integer, ByRef pretrigCount As Integer, ByRef totalCount As Integer, ByVal memHandle As Integer, ByVal options MccDaq.ScanOptions)

Parameters:

chanArray  
Array containing channel values. Valid channel values are analog input channels, digital ports, counter input channels, and temperature input channels of the device.

canTypeArray  
Array containing channel types. Each element of this array defines the type of the corresponding element in the chanArray.

All of the chanTypeArray settings are Mccdaq.ChannelType enumerated constants. Set it to one of the constants in the "chanTypeArray parameter values" section on page 349.

gainArray  
Array containing A/D range codes. If the corresponding element in the chanArray is not an analog input channel, the range code for this channel is ignored.

All of the gainArray settings are MccDaq.Range enumerated constants. Set to any range that is supported by the selected A/D board. Refer to board-specific information in the Universal Library User's Guide for a list of the supported A/D ranges of each board. Refer to Table 7 on page 208 for a list of valid range settings.

chanCount  
Number of elements in each of the three arrays - chanArray, chanTypeArray and gainArray.

rate  
The sample rate at which samples are acquired, in samples per second per channel.

rate also returns the value of the actual rate set, which may be different from the requested rate because of pacer limitations.

pretrigCount  
Sets the number of pre-trigger samples to collect. Specifies the number of samples to collect before the trigger occurs. This method won’t run in pre-trigger mode if preTrigCount is set to zero. preTrigCount is ignored if the ExtTrigger option is not specified.

preTrigCount also returns the value of the actual pre-trigger count set, which may be different from the set pre-trigger count because pre-trigger count must be a multiple of the channel count (chanCount).

preTrigCount must be evenly divisible by the number of channels being scanned (chanCount). If it is not, this method adjusts the number (down) to the next valid value and returns that value to the preTrigCount parameter.
totalCount  Total number of samples to collect. Specifies the total number of samples to collect and store in the buffer. totalCount must be greater than pretrigCount.

totalCount also returns the value of the actual total count set, which may be different from the requested total count because total count must be a multiple of the channel count (chanCount).

totalCount must be evenly divisible by the number of channels being scanned (chanCount). If it is not, this method adjusts the number (down) to the next valid value and returns that value to the totalCount parameter.

memHandle  Handle for the Windows buffer to store data. This buffer must have been previously allocated with the WinBufAlloc() method.

options  Bit fields that control various options. All of the options settings are Mccdaq.ScanOptions enumerated constants. This field may contain any combination of non-contradictory choices in the "options parameter values" section below.

chanTypeArray parameter values:

- Analog  Analog input channel.
- Digital8  8-bit digital input port.
- Digital16  16-bit digital input port. (FIRSTPORTA only)
- Ctrl16  16-bit counter.
- Ctrl32Low  Lower 16-bits of a 32-bit counter.
- Ctrl32High  Upper 16-bits of a 32-bit counter.
- CJC  CJC channel.
- TC  Thermocouple channel.

The GetTCValues() method can be used to convert raw thermocouple data to data on a temperature scale (Celsius, Fahrenheit or Kelvin). Note: If at least one TC channel is listed in the channel array, and averaging is enabled for that channel, the averaging will be applied to all of the channels listed in the channel array.

SetpointStatus  The setpoint status register. This is a bitfield indicating the state of each of the setpoints. A "1" indicates that the setpoint criteria has been met.

chanTypeArray flag values:

- SetpointEnable  Enables a setpoint. When this option is specified, it must be OR'ed with the ChanTypeArray parameter values.

You set the setpoint criteria with the DaqSetSetpoints() method. The number of channels set with the SetpointEnable flag must match the number of setpoints set by the DaqSetSetpoints() method's setpointCount parameter.

options parameter values:

- Background  When the Background option is used, control returns immediately to the next line in your program and the data collection from the counters into the buffer continues in the background. If the Background option is not used, the DaqInScan() method does not return to your program until all of the requested data has been collected and returned to the buffer.
Use GetStatus() with DaqiFunction to check on the status of the background operation. Use StopBackground() with DaqiFunction to terminate the background process before it has completed. Execute StopBackground() after normal termination of all background functions in order to clear variables and flags.

If the Background option is not used, the DaqInScan() 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() with DaqiFunction to check on the status of the background operation. Use StopBackground() with DaqiFunction to terminate the background process before it has completed. Execute StopBackground() 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 to use StopBackground() with the DaqiFunction. Normally this option should be used in combination with Background so that your program will regain control.

ExtClock If this option is used, 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. 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.

ExtTrigger If this option is specified, the sampling will not begin until the trigger condition is met (refer to the DaqSetTrigger() method).

Returns:
An ErrorInfo object that indicates the status of the operation.
rate – Actual sampling rate used.
preTrigCount – Actual pre-trigger count used.
totalCount – Actual total count used.
memHandle - Collected data returned via the Windows buffer.
DaqOutScan()

Outputs values synchronously to analog output channels and digital output ports. This method only works with boards that support synchronous output.

Member of the MccBoard class.

Function prototype:

**VB .NET:**
```
Public Function DaqOutScan(ByVal chanArray As Short(), ByVal chanTypeArray As MccDaq.ChannelType(), ByVal gainArray As MccDaq.Range(), ByVal chanCount As Integer, ByVal rate As Integer, ByVal count As Integer, ByVal memHandle As Integer, ByVal options As MccDaq.ScanOptions) As MccDaq.ErrorInfo
```

**Visual Basic:**
```
Public ErrorInfo DaqOutScan(short[] chanArray, MccDaq.ChannelType[] chanTypeArray, MccDaq.Range[] gainArray, int chanCount, ref int rate, int count, int memHandle, MccDaq.ScanOptions options)
```

**Parameters:**

- **chanArray**
  - Array containing channel values. Valid channel values are analog output channels and digital ports.

- **chanTypeArray**
  - Array containing channel types. Each element of this array defines the type of the corresponding element in the chanArray. The chanTypeArray settings are Mccdaq.ChannelType enumerated constants. Choices are:
    - Analog
    - Digital16 (16-bit digital output port. (FirstPortA only)

- **gainArray**
  - Array containing D/A range codes. If the corresponding element in the ChanArray is not an analog output channel, the range code for this channel is ignored. If the board does not have programmable gain, this parameter is ignored, and therefore can be set to null.

- **chanCount**
  - Number of elements in each of the three arrays - chanArray, chanTypeArray and gainArray.

- **rate**
  - Sample rate in scans per second. rate also returns the value of the actual rate set, which may be different from the requested rate because of pacer limitations.

- **count**
  - Sets the total number of values to output. count must be a multiple of chanCount.

- **memHandle**
  - Handle for the Windows buffer from which data is output. This buffer must have been previously allocated with the WinBufAlloc() method and data values loaded (for example using WinArrayToBuff).

- **options**
  - Bit fields that control various options. All of the options settings are Mccdaq.ScanOptions enumerated constants. This field may contain any combination of non-contradictory choices in the "options parameter values" section below.

**options parameter values:**

- **Background**
  - When this option is used, the output operations begin running in the background, and control immediately returns to the next line of your program.

  Use GetStatus() with DaqFunction to check the status of background operation. Use StopBackground() with DaqFunction to terminate background operations before they are completed. Execute StopBackground() with DaqFunction 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 outputs the specified number (Count) of output values, it resets to the start of the buffer and begins again. The only way to stop this operation is by calling `StopBackground()` with `DaqFunction`. This option should only be used in combination with `Background` so that your program regains control.

**ExtClock**

If this option is used, conversions are paced by the signal on the external clock input rather than by the internal pacer clock. Each conversion is triggered on the appropriate edge of the clock input signal.

When this option is used, the `rate` parameter is ignored. The sampling rate is dependent on the clock signal. Options for the board default to transfer types that allow the maximum conversion rate to be attained unless otherwise specified.

**ADCClockTrig**

If this option is used, the data output operation is triggered upon the start of the ADC clock.

**ADCClock**

When this option is used, the data output operation is paced by the ADC clock.

**NonStreamedIO**

This option allows non-streamed data output to be generated to a specified output channel.

In this mode, the aggregate size of data output buffer must be less than or equal to the size of the internal data output FIFO on the Measurement Computing device. This allows the data output buffer to be loaded into the device’s internal output FIFO.

Once the sample updates are transferred (or downloaded) to the device, the device is responsible for outputting the data. While the size is limited, and the output buffer cannot be changed once the output is started, this mode has the advantage being able to continue data output without having to periodically feed output data through the program to the device.

**Returns:**

An `ErrorInfo` object that indicates the status of the operation.

`rate` – Actual sampling rate used.
DaqSetSetpoints()

Configures up to 16 detection setpoints associated with the input channels within a scan group. This method only works with boards that support synchronous input.

Member of the MccBoard class.

Function Prototype:

**VB .NET:**
```vbnet
Public Function DaqSetSetpoints(ByVal limitAArray As Single(), ByVal limitBArray As Single(), ByVal reserved As Single(), ByVal setpointFlagsArray As MccDaq.SetpointFlag(), ByVal setpointOutputArray As MccDaq.SetpointOutput(), ByVal output1Array As Single(), ByVal output2Array As Single(), ByVal outputMask1Array As Single(), ByVal outputMask2Array As Single(), ByVal setpointCount As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo DaqSetSetpoints(float[] limitAArray, float[] limitBArray, float[] reserved, MccDaq.SetpointFlag[] setpointFlagsArray, MccDaq.SetpointOutput[] setpointOutputArray, float[] output1Array, float[] output2Array, float[] outputMask1Array, float[] outputMask2Array, int setpointCount)
```

Parameters:

- **limitAArray**
  - Array containing the limit A values for the input channels used for the setpoint. Limit A specifies a value used to determine if the setpoint criteria are met.

- **limitBArray**
  - Array containing the limit B values for the input channels used for the setpoint. Limit B specifies a value used to determine if the setpoint criteria are met.

- **reserved**
  - Reserved for future use.

- **setpointFlagsArray**
  - Array containing the setpoint flags. All of the setpointFlagsArray settings are MccDaq.SetpointFlag enumerated constants. Set it to one of the constants in the "setpointFlagsArray parameter values" section below.

- **setpointOutputArray**
  - Array containing output sources. All of the setpointOutputArray settings are MccDaq.SetPointOutput enumerated constants. Set it to one of the constants in the "setpointOutputArray parameter values" section on page 354.

- **output1Array**
  - Array containing the values for the output channels used for the setpoint.

- **output2Array**
  - Array containing the values for the output channels used for the setpoint.

- **outputMask1Array**
  - Array containing the output masks for output value 1 – for FIRSTPORTC only.

- **outputMask2Array**
  - Array containing the output masks for output value 2 – for FIRSTPORTC only.

- **setpointCount**
  - Number of setpoints to configure (0 -16). Set to 0 to disable the setpoints.

**setpointFlagsArray parameter values:**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EqualLimitA</td>
<td>Setpoint criteria: The input channel = limit A.</td>
</tr>
<tr>
<td>LessThanLimitA</td>
<td>Setpoint criteria: The input channel &lt; limit A.</td>
</tr>
<tr>
<td>GreaterThanLimitB</td>
<td>Setpoint criteria: The input channel &gt; limit B.</td>
</tr>
<tr>
<td>OutsideLimits</td>
<td>Setpoint criteria: The input channel &lt; limit A and &gt; limit B.</td>
</tr>
<tr>
<td>InsideLimits</td>
<td>Setpoint criteria: The input channel &gt; limit A and &lt; limit B.</td>
</tr>
<tr>
<td>Hysteresis</td>
<td>Setpoint criteria: If the input channel &gt; limit A then output value 1. If the input channel &lt; limit B then output value 2.</td>
</tr>
</tbody>
</table>
**Synchronous I/O Methods**

`DaqSetSetpoints()` method:

- **UpdateOnTrueOnly**: If the criteria is met then output value 1.
- **UpdateOnTrueAndFalse**: If the criteria is met then output value 1, else output value 2.

**setpointOutputArray parameter values:**

<table>
<thead>
<tr>
<th>Output Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Perform no outputs.</td>
</tr>
<tr>
<td>FirstPortC</td>
<td>Output to FIRSTPORTC when the criteria is met.</td>
</tr>
<tr>
<td>DAC0</td>
<td>Output to DAC0 when the criteria is met. You must have a device with DAC0.</td>
</tr>
<tr>
<td>DAC1</td>
<td>Output to DAC1 when the criteria is met. You must have a device with DAC1.</td>
</tr>
<tr>
<td>DAC2</td>
<td>Output to DAC2 when the criteria is met. You must have a device with DAC2.</td>
</tr>
<tr>
<td>DAC3</td>
<td>Output to DAC3 when the criteria is met. You must have a device with DAC3.</td>
</tr>
<tr>
<td>TMR0</td>
<td>Output to timer 0 when the criteria is met.</td>
</tr>
<tr>
<td>TMR1</td>
<td>Output to timer 1 when the criteria is met.</td>
</tr>
</tbody>
</table>

**Returns:**

Error code or 0 if no errors
**DaqSetTrigger()**

Selects the trigger source and sets up its parameters. This trigger is used to initiate or terminate an acquisition using the **DaqInScan()** method if the ExtTrigger option is selected. This method only works with boards that support synchronous output.

Member of the **MccBoard** class.

**Function prototype:**

**VB .NET:**

```vbnet
Public Function DaqSetTrigger(ByVal trigSource As MccDaq.TriggerSource, ByVal trigSense MccDaq.TriggerSensitivity, ByVal trigChan As Integer, ByVal chanType As MccDaq.ChannelType, ByVal gain As MccDaq.Range, ByVal level As Single, ByVal variance As Single, ByVal trigEvent As MccDaq.TriggerEvent) As MccDaq.ErrorInfo
```

**Visual Basic:**

```vbnet
public ErrorInfo DaqSetTrigger(MccDaq.TriggerSource trigSource, MccDaq.TriggerSensitivity trigSense, int trigChan, MccDaq.ChannelType chanType, MccDaq.Range gain, float level, float variance, MccDaq.TriggerEvent trigEvent)
```

**Parameters:**

- **trigSource**
  Specifies the type of triggering based on the external trigger source. All of the trigSource settings are MccDaq.TriggerSource enumerated constants. Set it to one of the constants in the "trigSource parameter values" section on page 356.

- **trigSense**
  Specifies the trigger sensitivity. The trigger sensitivity normally defines the way in which a trigger event is detected based upon the characteristics of the trigger input signal. Often, it defines the way in which the trigger input signal(s) should be compared to the trigger level parameter value. All of the trigSense settings are MccDaq.TriggerSensitivity enumerated constants. Set it to one of the constants in the "trigSense parameter values" section on page 356.

- **trigChan**
  The trigger channel. This channel must be a configured channel in the channel array (refer to **DaqInScan()**).

- **chanType**
  The channel type. All of the chanType settings are MccDaq.ChannelType enumerated constants. chanType should match the channel type setting for the trigger channel configured using the **DaqInScan()** method.

- **gain**
  The trigger channel gain code. If the device has programmable gain, this parameter should match the gain code setting when the channel is configured using the **DaqInScan()** method. The gain parameter is ignored if trigChan is not an analog channel.

- **level**
  A single precision floating point value which represents, in engineering units, the level at or around which the trigger event should be detected. This option is used for trigger types that depend on an input channel comparison to detect the start trigger or stop trigger event.

  The actual level at which the trigger event is detected depends upon trigger sensing and variability. Refer to **Trigger levels** on page 357 for more information.

- **variance**
  A single-precision floating point value which represents, in engineering units, the amount that the trigger event can vary from the level parameter.

  While the TrigSense parameter indicates the direction of the input signal relative to the level parameter, the variance parameter specifies the degree to which the input signal can vary relative to the level parameter.
Synchronous I/O Methods

DaqSetTrigger()

trigEvent

Specifies the trigger event type. Valid values indicate either a start trigger event
(MccDaq.TriggerEvent.Start) or a stop trigger event
(MccDaq.TriggerEvent.Stop).

Start: The start trigger event defines the conditions under which post-trigger
acquisition data collection should be initiated or triggered. The start trigger event
can vary in complexity from starting immediately, to starting on complex channel
value definitions.

Stop: The stop trigger event signals the current data acquisition process to
terminate. The stop trigger event can be as simple as that of a scan count, or as
complex as involving a channel value level condition.

trigSource parameter values:

- **TrigImmediate**: Start trigger event only. Acquisition begins immediately upon invocation the
  DaqInScan() method. No pre-trigger data acquisition is possible with this
  trigger type.

- **TrigExtTTL**: Start trigger event only. Acquisition begins on the selectable edge of an external
  TTL signal. No pre-trigger data acquisition is possible with this trigger type.

- **TrigAnalogHW**: Start trigger event only. Acquisition begins upon a selectable criteria of the input
  signal (above level, below level, rising edge, etc.) trigChan must be defined as the
  first channel in the channel scan group. No pre-trigger data acquisition is possible
  with this trigger type.

- **TrigAnalogSW**: Post-trigger data acquisition begins upon a selectable criteria of the input signal
  (above level, below level, rising edge, etc.)

- **TrigDigPattern**: Post-trigger data acquisition begins upon receiving a specified digital pattern on the
  specified digital port.

- **TrigCounter**: Post-trigger data acquisition begins upon detection of specified counter criteria.

- **TrigScanCount**: Stop trigger event only. Stops collecting post-trigger data when the specified
  number of post-trigger scans are completed.

trigSense parameter values:

- **RisingEdge**: Triggers when the signal goes from low to high (TTL trigger) or rises through a
  specified level (hardware analog, software analog, and counter).

- **FallingEdge**: Triggers when the signal goes from high to low (TTL trigger) or falls through a
  specified level (hardware analog, software analog, and counter).

- **AboveLevel**: Triggers when the signal is above a specified level (hardware analog, software
  analog, counter, and digital pattern).

- **BelowLevel**: Triggers when the signal is below a specified level (hardware analog, software
  analog, counter, and digital pattern).

- **EqLevel**: Triggers when the signal equals a specified level (hardware analog, software
  analog, counter, and digital pattern).

- **NeLevel**: Triggers when the signal does not equal a specified level (hardware analog,
  software analog, counter, and digital pattern).

Returns:

An ErrorInfo object that indicates the status of the operation.
Notes:

**Trigger levels**: The actual level at which the trigger event is detected depends upon trigger sensing and variability. The various ranges of possible values for the level parameter based on the trigger source are:

- **TrigAnalogHW**: The voltage used to define the trigger level. Trigger detection is performed in hardware.
- **TrigAnalogSW**: The voltage used to define the trigger level. Trigger detection is performed in software.
- **TrigDigPattern**: Sets the bit pattern for the digital channel trigger. Choices are:
  - 0.0 (no bits set): 255.0 (all bits set) for 8-bit digital ports.
  - 0.0 (no bits set): 65,535.0 (all bits set) for 16-bit digital ports.
- **TrigCounter**: Selects either Pulse or Totalize counter values (0.0 – 65,535).
- **TrigImmediate**: Ignored
- **TrigScanCount**: Ignored

**Trigger start and stop criteria**: The table below lists the trigger start and stop criteria based on the selected trigger type and sensitivity.

<table>
<thead>
<tr>
<th>Trigger Start/Stop Source (TrigSource)</th>
<th>Trigger Sensitivity (TrigSense)</th>
<th>Trigger Start/Stop Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TrigAnalogHW</strong> (Start trigger event only)</td>
<td>RisingEdge</td>
<td>Triggers when the signal value &lt; (level – variance) Then, the signal value &gt; level</td>
</tr>
<tr>
<td></td>
<td>FallingEdge</td>
<td>Triggers when the signal value &gt; (level + variance) Then, the signal value &lt; level</td>
</tr>
<tr>
<td></td>
<td>AboveLevel</td>
<td>Triggers when the signal value &gt; (level)</td>
</tr>
<tr>
<td></td>
<td>BelowLevel</td>
<td>Triggers when the signal value &lt; (level)</td>
</tr>
<tr>
<td><strong>TrigAnalogSW</strong></td>
<td>RisingEdge</td>
<td>Triggers/stops when the signal value &lt; (level – variance) Then, the signal value &gt; level</td>
</tr>
<tr>
<td></td>
<td>FallingEdge</td>
<td>Triggers/stops when the signal value &gt; (level + variance) Then, the signal value &lt; level</td>
</tr>
<tr>
<td></td>
<td>AboveLevel</td>
<td>Triggers/stops when the signal value &gt; (level)</td>
</tr>
<tr>
<td></td>
<td>BelowLevel</td>
<td>Triggers/stops when the signal value &lt; (level)</td>
</tr>
<tr>
<td></td>
<td>EqLevel</td>
<td>Triggers/stops when the (level – Variance) &lt; signal value &lt; (level + variance)</td>
</tr>
<tr>
<td></td>
<td>NeLevel</td>
<td>Triggers/stops when the signal value &lt; (level – variance) OR when the signal value &gt; (level + variance)</td>
</tr>
<tr>
<td><strong>TrigDigPattern</strong></td>
<td>AboveLevel</td>
<td>Triggers/stops when (digital port value AND bitwise) variance &gt; (level AND bitwise variance)</td>
</tr>
<tr>
<td></td>
<td>BelowLevel</td>
<td>Triggers/stops when (digital port value AND bitwise) variance &lt; (level AND bitwise variance)</td>
</tr>
<tr>
<td></td>
<td>EqLevel</td>
<td>Triggers/stops when (digital port value AND bitwise) variance = (level AND bitwise variance)</td>
</tr>
<tr>
<td></td>
<td>NeLevel</td>
<td>Triggers/stops when (digital port value AND bitwise) variance != (level AND bitwise variance)</td>
</tr>
<tr>
<td>Trigger Start/Stop Source (TrigSource)</td>
<td>Trigger Sensitivity (TrigSense)</td>
<td>Trigger Start/Stop Criteria</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>TrigCounter</td>
<td>RisingEdge</td>
<td>Triggers/stops when the counter channel &lt; (level – variance) Then, the counter channel &gt; level</td>
</tr>
<tr>
<td></td>
<td>FallingEdge</td>
<td>Triggers/stops when the counter channel &gt; (level + variance) Then, the counter channel &lt; level</td>
</tr>
<tr>
<td></td>
<td>AboveLevel</td>
<td>Triggers/stops when the counter channel &gt; (level – variance)</td>
</tr>
<tr>
<td></td>
<td>BelowLevel</td>
<td>Triggers/stops when the counter channel &lt; (level + variance)</td>
</tr>
<tr>
<td></td>
<td>EqLevel</td>
<td>Triggers/stops when (level – variance) &lt; counter channel &lt; (level + variance)</td>
</tr>
<tr>
<td></td>
<td>NeLevel</td>
<td>Triggers/stops when the counter channel &lt; (level – variance) OR when the counter channel &gt; (level + variance)</td>
</tr>
</tbody>
</table>
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.
Temperature Input Methods

**Tln()**

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 InstaCal configuration file. Run the *InstaCal* configuration program to set these items.

Member of the **MccBoard** class.

**Function prototype:**

**VB .NET:**

```vbnet
Public Function Tln(ByVal chan As Integer, ByVal scale As MccDaq.TempScale, ByRef tempValue As Single, ByVal options As MccDaq.ThermocoupleOptions) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo Tln(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.

\[
\text{Chan} = (\text{ADChan} \times 16) + (16 + \text{MuxChan})
\]

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 \(\text{chan}\) would be \((0 \times 16) + (16 + 5) = 0 + 21 = 21\).

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:
  \[
  \text{Chan} = (\text{ADChan} \times 16) + (16 + \text{MuxChan})
  \]
  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 \(\text{Chan}\) would be \((0 \times 16) + (16 + 5) = 0 + 21 = 21\).

- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is less than or equal to 7:
  \[
  \text{Chan} = (\text{ADChan} \times 16) + (64 + \text{MuxChan})
  \]
  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 \(\text{Chan}\) would be \((7 \times 16) + (64 + 5) = 112 + 69 = 181\).

- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is greater than or equal to 31:
  \[
  \text{Chan} = (\text{ADChan} \times 16 - 320) + \text{MuxChan}
  \]
  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 \(\text{Chan}\) is \((32 \times 16 - 320) + 5 = 192 + 5 = 197\).

CJC Channel: The Cold Junction Compensation (CJC) channel is set in the InstaCal 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.
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. If a NotReady error occurs, the value returned is -9000.
Temperature Input Methods

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.

44
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
  \[ \text{Chan} = (\text{ADChan} \times 16) + (16 + \text{MuxChan}) \]

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 \((0+1) \times 16 + 5 = 21\), and the value for highChan would be \((0+1) \times 16 + 7 = 23\).

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:
  \[ \text{Chan} = (\text{ADChan} \times 16) + (16 + \text{MuxChan}) \]
  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 \((0 \times 16) + (16 + 5) = 0 + 21 = 21\), and the value for highChan would be \((0 \times 16) + (16 + 5) = 0 + 231 = 23\).

- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is less than or equal to 7:
  \[ \text{Chan} = (\text{ADChan} \times 16) + (64 + \text{MuxChan}) \]
  For example, you have a 6K-EXP16 connected to a PCI-DAS6031 via the PCI-DAS6031 channel 7. Connect a thermocouple to channels 5, 6, and 7 of the 6K-EXP16: the value for lowChan is \((7 \times 16) + (64 + 5) = 112 + 69 = 181\), and the value for highChan is \((7 \times 16) + (64 + 7) = 112 + 71 = 183\).

- If the A/D board has 64 single-ended channels and the A/D multiplexer channel is greater than or equal to 32:
  \[ \text{Chan} = (\text{ADChan} \times 16 - 320) + \text{MuxChan} \]
  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 is \((32 \times 16 - 320) + 5 = 192 + 5 = 197\), and the value for highChan is \((32 \times 16 - 320) + 7 = 192 + 7 = 199\).

CJC Channel: The Cold Junction Compensation (CJC) channel is set in the InstaCal 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](http://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 **OUTOF RANGE** or **OPEN CONNECTION** 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 in the *Universal Library User's Guide*).
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 methods and returns a memory handle for it.

Member of the MccService class.

Function prototype:

**VB .NET:**
```vbnet
Public Shared Function WinBufAlloc(ByVal numPoints As Integer) As Integer
```

**C# .NET:**
```csharp
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 methods in the library. If an error occurs, the handle will come back as 0 to indicate that the buffer was not allocated.
WinBufAlloc32()

Allocates a Windows global memory buffer for use with 32-bit scan methods, and returns a memory handle for the buffer.

Member of the MccService class.

Function prototype:

VB .NET: Public Shared Function WinBufAlloc32(ByVal numPoints As Integer) As Integer

C# .NET: public int WinBufAlloc32(int numPoints)

Parameters:
numPoints The size of buffer to allocate. Specifies how many data points (32-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 error code. It returns a Windows global memory handle which can then be passed to the scan methods in the library. If an error occurs, the handle will come back as 0 to indicate that the buffer was not allocated.
WinBufFree()

Frees a Windows global memory buffer which was previously allocated with the WinBufAlloc() or WinBufAlloc32() method.

Member of the MccService class.

Function prototype:

```vbnet
Public Shared Function WinBufFree(ByVal memHandle As Integer) As MccDaq.ErrorInfo
```

```csharp
public static MccDaq.ErrorInfo WinBufFree(int memHandle)
```

Parameters:

- `memHandle` A Windows memory handle. This must be a memory handle that was returned by WinBufAlloc() or WinBufAlloc32() when the buffer was allocated.

Returns:

An ErrorInfo object that indicates the status of the operation.
WinArrayToBuf()

Copies data from a one-dimensional or two-dimensional array into a Windows memory buffer.

Member of the MccService class.

Function prototype:

**VB .NET:**

```vbnet
Public Shared Function WinArrayToBuf(ByRef dataArray As Short, ByVal memHandle As Integer, ByVal firstPoint As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

Copies data from a one-dimensional array of short values:

```vbnet
Public Shared Function WinArrayToBuf(ByRef dataArray As System.UInt16, ByVal memHandle As Integer, ByVal firstPoint As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

Copies data from a one-dimensional array of System.UInt16 values:

```vbnet
Public Shared Function WinArrayToBuf(ByRef dataArray(,) As Double, ByVal memHandle As Integer, ByVal firstPoint As Integer, ByVal numPoints As Integer, ByVal numChannels As Integer) As MccDaq.ErrorInfo
```

Copies data from a two-dimensional array of double values:

**C# .NET:**

```csharp
public static MccDaq.ErrorInfo WinArrayToBuf(ref short dataArray, int memHandle, int firstPoint, int numPoints)
```

Copies data from a one-dimensional array of short values:

```csharp
public static MccDaq.ErrorInfo WinArrayToBuf(ref ushort dataArray, int memHandle, int firstPoint, int numPoints)
```

Copies data from a one-dimensional array of System.UInt16 values:

```csharp
public static MccDaq.ErrorInfo WinArrayToBuf(ref double[,] dataArray, int memHandle, int firstPoint, int numPoints, int numChannels)
```

Copies data from a two-dimensional array of double values:

**Parameters:**

- **dataArray**
  - The array containing the data to be copied. The first dimension should equal the number of channels. The second dimension should equal the number of points/channel.

- **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 the first point in the memory buffer where data will be copied to.

- **numPoints**
  - Number of data points to copy from dataArray.

- **numChannels**
  - Number of channels to copy from dataArray.

**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. You can use the firstPoint and numPoints parameters to fill a portion of the buffer. This is useful if you want to send new data to the buffer after a Background + Continuous output scan has been started, for example during circular buffering.
**WinBufToArray()**

Copies data from a Windows memory buffer into a one-dimensional or two-dimensional array.

Member of the `MccService` class.

**Function prototype:**

**VB .NET:**

Copies data to a one-dimensional array of short values:

```vbnet
Public Shared Function WinBufToArray(ByVal memHandle As Integer, ByVal dataArray As Short, ByVal firstPoint As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

Copies data to a one-dimensional array of System.UInt16 values:

```vbnet
Public Shared Function WinBufToArray(ByVal memHandle As Integer, ByVal dataArray As System.UInt16(), ByVal firstPoint As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

Copies data to a two-dimensional array of double values:

```vbnet
Public static ErrorInfo WinBufToArray(ByVal memHandle As Integer, ByVal dataArray As Double, ByVal firstPoint As Integer, ByVal numPoints As Integer, ByVal numChannels As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

Copies data to a one-dimensional array of short values:

```csharp
public static MccDaq.ErrorInfo WinBufToArray(int memHandle, out short dataArray, int firstPoint, int numPoints)
```

Copies data to a one-dimensional array of System.UInt16 values:

```csharp
public static MccDaq.ErrorInfo WinBufToArray(int memHandle, out ushort dataArray, int firstPoint, int numPoints)
```

Copies data to a two-dimensional array of double values:

```csharp
public static MccDaq.ErrorInfo WinBufToArray(int memHandle, out double[,] dataArray, int firstPoint, int numPoints, int numChannels)
```

**Parameters:**

- `memHandle`: This must be a memory handle that was returned by `WinBufAlloc()` when the buffer was allocated. The data will be copied from this buffer.
- `dataArray`: Array that the data will be copied to. The first dimension should equal the number of channels. The second dimension should equal the number of points/channel.
- `firstPoint`: Index of the first point in the memory buffer that the data will be copied from.
- `numPoints`: Number of data points to copy into `dataArray`.
- `numChannels`: Number of channels to copy into `dataArray`.

**Returns:**

An `ErrorInfo` object that indicates the status of the operation.

**Notes:**

This method copies data from a Windows global memory buffer to a single value or into an array of doubles. This would typically be used to retrieve data from the buffer after executing an input scan method. You can use the `firstPoint` and `numPoints` parameters 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.
WinBufToArray32()

Copies 32-bit data from a Windows global memory buffer into a one-dimensional or two-dimensional array. This method is typically used to retrieve data from the buffer after executing an input scan method.

Member of the MccService class.

Function prototype:

**VB .NET:**

```vbnet
Public Shared Function WinBufToArray32(ByVal memHandle As Integer, ByRef dataArray(,) As Double, ByVal firstPoint As Integer, ByVal numPoints As Integer, ByVal numChannels As Integer) As MccDaq.ErrorInfo
```

Copies data into a two-dimensional array of double values:

```vbnet
Public Shared Function WinBufToArray32(ByVal memHandle As Integer, ByRef dataArray(,) As Integer, ByVal firstPoint As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

Copies data into an array of integer values:

```vbnet
Public Shared Function WinBufToArray32(ByVal memHandle As Integer, ByRef dataArray As Integer, ByVal firstPoint As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

Copies data into an array of System.UInt32 values:

```vbnet
Public Shared Function WinBufToArray32(ByVal memHandle As Integer, ByRef dataArray As System.UInt32(), ByVal firstPoint As Integer, ByVal numPoints As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo WinBufToArray32 (int memHandle, out double[,] dataArray, int firstPoint, int numPoints, int numChannels)
```

Copies data into a two-dimensional array of double values:

```csharp
public MccDaq.ErrorInfo WinBufToArray32 (int memHandle, out int dataArray, int firstPoint, int numPoints)
```

Copies data into an array of integer values:

```csharp
public MccDaq.ErrorInfo WinBufToArray32 (int memHandle, out uint dataArray, int firstPoint, int numPoints)
```

Copies data into an array of System.UInt32 values:

```csharp
public MccDaq.ErrorInfo WinBufToArray32 (int memHandle, out uint dataArray, int firstPoint, int numPoints)
```

**Parameters:**

- **memHandle**
  - The memory handle that was returned by WinBufAlloc32() when the buffer was allocated. The buffer should contain the data that you want to copy.

- **dataArray**
  - The array where the data is copied.

- **firstPoint**
  - The index of first point in the memory buffer that data is copied from.

- **numPoints**
  - The number of data points to copy.

- **numChannels**
  - The number of channels to copy into dataArray.

**Returns:**

- **Error code** or 0 if no errors
Notes:
You can copy only a portion of the buffer to the array using the firstPoint and numPoints argument. This is useful if you want foreground code to manipulate previously collected data while a Background scan continues to collect new data.

Although this method 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 WinBufAlloc32() to the appropriate type. This method avoids having to copy the data from the memory buffer to an array. Refer to the following example:

```c
/*declare and initialize the variables*/
long numPoints = 1000;
unsigned short *dataArray = NULL;
int MemHandle = 0;

/*allocate the buffer and cast it to a pointer to an unsigned long*/
MemHandle = WinBufAlloc32(numPoints);
dataArray = (unsigned long*)MemHandle;

/*scan in the data*/
CInScan(......,MemHandle,...);

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

/*free the buffer and NULL the pointer*/
WinBufFree(MemHandle);
dataArray = NULL;
```
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]
DeviceLogin()

Opens a device session with a shared device.

Member of the MccBoard class.

Function prototype:
- **VB .NET:** Public Function DeviceLogin(ByVal userName As String, ByVal password As String) As MccDaq.ErrorInfo
- **C# .NET:** public MccDaq.ErrorInfo DeviceLogin(System.String userName, System.String password)

Parameters:
- **userName** A null-terminated string that identifies the user name used to log in to a device session.
- **password** A null-terminated string that identifies the password used to log in to a device session.

Returns:
- Error code or 0 if no errors.

Notes:
If the user name or password is invalid, the INVALIDLOGIN error is returned.

If the session is already opened by another user, the SESSIONINUSE error is returned.

DeviceLogout()

Releases the device session with a shared device.

Member of the MccBoard class.

Function prototype:
- **VB .NET** Public Function DeviceLogout() As MccDaq.ErrorInfo
- **C# .NET** public MccDaq.ErrorInfo DeviceLogout()
HideLoginDialog()

Prevents the default login dialog from being shown when a protected function is called while not logged in.

Member of the MccBoard class.

Function prototype:

**VB .NET**
```vbnet
Public Function HideLoginDialog(ByVal hide As Boolean) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo HideLoginDialog(System.Boolean hide)
```

Parameters:
- **hide**
  If true, default dialog will not be shown when a protected function is called while the user is not logged in.

Returns:
- Error code or 0 if no errors.

Notes:
- Overrides InstaCal Show Login Dialog prompt setting.
DisableEvent()

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

Member of the `MccBoard` class.

Function prototype:

**VB .NET:**

```vbnet
Public Function DisableEvent(ByVal eventType As MccDaq.EventType) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
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 parameter values" on page 380 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 `DisableEvent()`. 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](http://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:**

VBA.NET:

```vbnet
Public Function EnableEvent(ByVal eventType As MccDaq.EventType, ByVal eventParameter As Integer, ByVal callbackFunc As MccDaq.EventCallback, ByVal userData As IntPtr) As MccDaq.ErrorInfo
```

C# .NET:

```csharp
public MccDaq.ErrorInfo EnableEvent(MccDaq.EventType eventType, uint 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 parameter values**" section on page 380.

- `eventParameter` Additional data required to specify some event conditions, such as an `OnDataAvailable` event or `OnExternalInterrupt` event.

  For `OnDataAvailable` events, `eventParameter` is used to determine the minimum number of samples to acquire during an analog input scan before generating the event. For `OnExternalInterrupt` events, `eventParameter` is used to latch digital bits on supported hardware by setting it to one of the constants in the "**eventParameter parameter values**" section on page 380.

  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 381 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, latches digital input data, or latches digital output data 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 eventParameter samples or more. Note that for BlockIo scans, events will be generated on packet transfers; for example, even if EventParameter 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 EventParameter 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().

eventParameter parameter values:
- **LatchDI**: Returns the data that was latched in at the most recent interrupt edge.
- **LatchDO**: Latches out the data most recently written to the hardware.

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 multi-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 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.
EngArrayToWinBuf()

Transfers a 2D array of engineering unit values to a Windows buffer as integer values.

The conversion from engineering unit values to integer values uses the D/A resolution of the board associated with the MccBoard object.

This method is usually used to obtain values compatible with the AOutScan() method or the DaqOutScan() method from a 2D array of engineering unit values, such as those provided by Measurement Studio signal generation methods. The converted values are transferred to the buffer based on the gain, firstPoint, count, and numChannels parameters.

Member of the MccBoard class.

Function prototype:

**VB .NET:**
```vbnet
Public Function EngArrayToWinBuf(ByVal gain As MccDaq.Range, ByVal engUnits As Double(), ByVal memHandle As Integer, ByVal firstPoint As Integer, ByVal count As Integer, ByVal numChannels As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo EngArrayToWinBuf(MccDaq.Range gain, double[,] engUnits, int memHandle, int firstPoint, int count, int numChannels)
```

Parameters:

- **gain**
  
The range to use for converting the data. This range should be the same as the range specified for AOutScan() or DaqOutScan().

- **gainArray**
  
The array containing the D/A range values used during the analog output scan.
  
  If a gain queue was not used for the scan, this array should only contain 1 element whose value matches the gain used during the scan. If a gain queue was used during the scan, this array should match the gainArray value used in DaqOutScan().
  
  If the corresponding range in the gainArray is set to NotUsed (MccDaq.Range.NotUsed), engineering unit values are returned as integer values.

- **gainCount**
  
The number of array elements in gainArray. Set gainCount to 1 when no gain queue was used for the scan. If a gain queue was used for the scan, this number should match the number of gain queue pairs defined in DaqOutScan().

- **engUnits**
  
The array of data to convert to binary units and store in the windows memory buffer. With the engUnits array, the channel numbers are stored in the first dimension, and the number of points/channel is stored in the second dimension.

- **memHandle**
  
The handle to the windows memory buffer that holds the binary data that is output. This value should be large enough to hold (count x numChannels) samples.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>firstPoint</code></td>
<td>The index into the windows memory buffer that will hold the first sample of the converted first channel. The index into the raw memory is (<code>firstPoint</code> × <code>numChannels</code>) so that converted data always starts with the first channel specified in the scan. For example, if <code>firstPoint</code> is 14 and the number of channels is 8, the index of the first converted sample is 112.</td>
</tr>
<tr>
<td><code>count</code></td>
<td>The number of samples per channel to convert from engineering units. Count should not exceed Windows buffer size ( / (\text{numChannels} - \text{firstPoint}) ).</td>
</tr>
<tr>
<td><code>numChannels</code></td>
<td>The number of channels of data stored in the existing array to be transferred.</td>
</tr>
</tbody>
</table>

**Returns:**

An `ErrorInfo` object that indicates the status of the operation.

**Notes:**

This method stores the samples specified by `firstPoint` in the windows memory buffer. Each sample is converted using the ranges set by `gain`.

If the corresponding range in the `gainArray` is set to `NotUsed`, engineering unit values are returned as integer values.
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:

**VB .NET:**
```vbnet
Public Sub MyCallback(ByVal BoardNum As Integer, ByVal EventType As MccDaq.EventType, ByVal EventData As UInt32, ByVal pUserData As System.IntPtr)
```

**C# .NET:**
```csharp
public delegate void EventCallback(int BoardNum, MccDaq.EventType EventType, uint EventData, IntPtr pUserData);
```

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 the value specified by the userData parameter in EnableEvent().

EventData parameter values:

<table>
<thead>
<tr>
<th>EventType</th>
<th>Value of EventData</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnScanError</td>
<td>The <a href="#">Error code</a> of the scan error.</td>
</tr>
<tr>
<td>OnExternalInterrupt</td>
<td>The number of interrupts generated since enabling the ON_EXTERNAL_INTERRUPT event.</td>
</tr>
<tr>
<td>OnPretrigger</td>
<td>The number of pretrigger samples available at time of pretrigger. Value is invalid for some boards when a TOOFEW error occurs. See board details.</td>
</tr>
<tr>
<td>OnDataAvailable</td>
<td>The number of samples acquired since the start of scan.</td>
</tr>
<tr>
<td>OnEndOfAiScan</td>
<td>The total number of samples acquired upon scan completion or end.</td>
</tr>
<tr>
<td>OnEndOfAoScan</td>
<td>The total number of samples output upon scan completion or end.</td>
</tr>
</tbody>
</table>
FlashLED()

Causes the LED on a USB device to flash.

Member of the **MccBoard** class.

**Function prototype:**

- **VB .NET:**
  ```vbnet
  Public Function FlashLED() As MccDaq.ErrorInfo
  ```

- **C# .NET:**
  ```csharp
  public MccDaq.ErrorInfo FlashLED()
  ```
**FromEngUnits()**

Converts a single precision voltage (or current) value in engineering units to an integer count value. This method is typically used to obtain a data value from a voltage value for output to a D/A with methods such as AOut().

Member of the **MccBoard** class.

**Function prototype:**

**VB .NET:**

```vbnet
Public Function FromEngUnits(ByVal range As MccDaq.Range, ByVal engUnits As Single, ByRef dataVal As Short) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
public MccDaq.ErrorInfo FromEngUnits(MccDaq.Range range, float engUnits, out ushort dataVal)
```

**Parameters:**

- **range**
  
  The voltage (or current) range to use for the conversion to counts. When using this method to obtain a value to send to a D/A board, keep in mind that some D/A boards have programmable voltage ranges, and others set the voltage range via switches on the board. In either case, the desired range must be passed to this method. Refer to Table 7 on page 208 for a list of valid range settings.

- **engUnits**
  
  The single precision voltage (or current) value to use for the conversion to counts. Set the value to be within the range specified by the range parameter.

- **dataVal**
  
  Returns an integer count to this variable that is equivalent to the engUnits parameter using the resolution of the D/A on the board (if any).

**Returns:**

An **ErrorMessage** object that indicates the status of the operation.

- **dataVal** — the integer count equivalent to engUnits is returned here.

**Note:**

This method is not supported for hardware with resolution greater than 16 bits.

The default resolution of this method is 12 bits, so if the device has neither analog input nor analog output, the result will be a 12 bit conversion.

If the device has both analog input and analog output, the resolution and transfer function of the D/A converter on the device is used.
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, ByVal 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 online 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 method:

- **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(ByVal status As Short, ByVal curCount As Integer, ByVal 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

The curCount parameter specifies how many points have been input or output since the Background process started. Use it to gauge how far along the operation is towards completion. Generally, curCount returns the total number of samples transferred between the DAQ board and the Windows data buffer at the time GetStatus() was called.

When you set both the Continuous and Background options, curCount's behavior depends on the board model. Refer to the board-specific information in the Universal Library User's Guide for the behavior of your board.

With recent MCC DAQ designs, the curCount parameter continually increases in increments of the packet size as Windows' circular data buffer recycles, until it reaches $2^{31}$. Since the count parameter is a signed integer, at 2,147,483,647 + 1, the count parameter rolls back to a negative number (-2,147,483,647). The count parameter resumes incrementing, eventually reaching 0 and increasing back up to 2,147,483,647.

The curIndex parameter is usually more useful than the curCount parameter in managing data collected when you set both the Continuous and Background options.

curIndex

The curIndex parameter is an index into the Windows data buffer. This index points to the start of the last completed channel scan that was transferred between the DAQ board and the Windows data buffer. If a scan is running but no points in the buffer have been transferred, curIndex equals -1 in most cases.

For Continuous operations, curIndex rolls over when the Windows data buffer is full. This rollover indicates that "new" data is now overwriting "old" data. Your goal is to process the old data before it gets overwritten. You can keep ahead of the data flow by copying the old data out of the buffer before new data overwrites it.

The curIndex parameter can help you access the most recently transferred data. Your application does not have to process the data exactly when it becomes available in the buffer – in fact, you should avoid doing so unless absolutely necessary. The curIndex parameter generally increments by the packet size, but in some cases the curIndex parameter can vary within the same scan. One instance of a variable increment is when the packet size is not evenly divisible by the number of channels.

You should determine the best size of the "chunks" of data that your application can most efficiently process, and then periodically check on the curIndex parameter value to determine when that amount of additional data has been transferred.
Refer to the *Universal Library User's Guide* for information on your board, particularly when using Pre-Trigger.

**functionType**  
Specifies which scan to retrieve status information about. 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.
- **Status**
  - **Idle** - No background operation is running.
  - **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()** or **CInScan()**.
- **DaqiFunction**  
  Specifies a synchronous input scan started with **DaqInScan()**.
- **DaqoFunction**  
  Specifies a synchronous output scan started with **DaqOutScan()**.
GetTCValues()

Converts raw thermocouple data from a Windows global memory buffer collected using the DaqInScan() method to a one-dimensional or two-dimensional array of data on a temperature scale (Celsius, Fahrenheit or Kelvin).

Member of the MccBoard class.

Function prototype:

**VB .NET:**

Copies data to a one-dimensional array of single values:

```vbnet
Public Function GetTCValues(ByVal chanArray As Short(), ByVal chanTypeArray As MccDaq.ChannelType(), ByVal chanCount As Integer, ByVal memHandle As Integer, ByVal firstPoint As Integer, ByVal count As Integer, ByVal scale As MccDaq.TempScale, ByRef tempValArray As Single) As MccDaq.ErrorInfo
```

Copies data to a two-dimensional array of double values:

```vbnet
Public Function GetTCValues(ByVal chanArray As Short(), ByVal chanTypeArray As MccDaq.ChannelType(), ByVal chanCount As Integer, ByVal memHandle As Integer, ByVal firstPoint As Integer, ByVal count As Integer, ByVal scale As MccDaq.TempScale, ByRef tempValArray As Double()) As MccDaq.ErrorInfo
```

**C# .NET:**

Copies data to a one-dimensional array of single values:

```csharp
public ErrorInfo GetTCValues(short[] chanArray, MccDaq.ChannelType[] chanTypeArray, int chanCount, int memHandle, int firstPoint, int count, MccDaq.TempScale scale, out float tempValArray)
```

Copies data to a two-dimensional array of double values:

```csharp
public ErrorInfo GetTCValues(short[] chanArray, MccDaq.ChannelType[] chanTypeArray, int chanCount, int memHandle, int firstPoint, int count, MccDaq.TempScale scale, out double[,] tempValArray)
```

**Parameters:**

- `chanArray` - Array containing channel values. Valid channel values are analog and temperature input channels and digital ports. `chanArray` must match the channel array used with the DaqInScan() method.
- `chanTypeArray` - Array containing channel types. Each element of this array defines the type of the corresponding element in the `chanArray`. `chanTypeArray` must match the channel type settings used with the DaqInScan() method.
- `chanCount` - Number of elements in `chanArray`.
- `memHandle` - The memory handle that was returned by WinBufAlloc() when the buffer was allocated. The buffer should contain the data that you want to convert.
- `firstPoint` - The index into the raw data memory buffer that holds the first sample of the first channel to be converted. The index into the raw memory is `firstPoint x chanCount` so that converted data always starts with the first channel specified in the scan. For example, if `firstPoint` is 14 and the number of channels is 8, the index of the first converted sample is 112.
- `count` - The number of samples per channel to convert to engineering units. `count` should not exceed Windows buffer size / `chanCount - firstPoint`.
- `scale` - Specifies the temperature scale that the input will be converted to. Choices are Celsius, Fahrenheit and Kelvin.
tempValArray

The array to hold the converted data. This array must be allocated by the user, and must be large enough to hold \( \text{count} \) samples x the number of temperature channels.

**Returns:**

An [ErrorInfo](#) object that indicates the status of the operation

tempValArray – Converted data.
InByte()

Reads a byte from a hardware register on a board.

Member of the MccBoard class.

Function prototype:

**VB .NET:**

```vbnet
Public Function InByte(ByVal portNum As Integer) As Integer
```

**C# .NET:**

```csharp
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 MecBoard 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.ErrorInf
```

**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:**

```vbnet
Public Function OutWord(ByVal portNum As Integer, ByVal portVal As Integer) As MccDaq.ErrorInf
```

**C# .NET:**

```csharp
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:**

```vbnet
Public Function RS485(ByVal transmit As MccDaq.OptionState , ByVal receive As MccDaq.OptionState ) As MccDaq.ErrorInfo
```

**C# .NET:**

```csharp
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:
```vbnet
Public Function StopBackground(ByVal funcType As MccDaq.FunctionType) As MccDaq.ErrorInfo
End Function
```

C# .NET:
```csharp
public MccDaq.ErrorInfo StopBackground(MccDaq.FunctionType funcType)
```

Parameters:

- `funcType` 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() or CInScan().
- **DaqiFunction**: Specifies a synchronous input scan started with DaqInScan().
- **DaqoFunction**: Specifies a synchronous output scan started with DaqOutScan().
ToEngUnits()

Converts an integer count value to an equivalent single precision voltage (or current) value. This method is typically used to obtain a voltage value from data received from an A/D with methods such as AIn().

Member of the MccBoard class.

Function prototype:

**VB .NET:**
```vbnet
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:**
```csharp
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**
  Voltage (or current) range to use for the conversion to engineering units. When using this method to obtain engineering units from a value received from an A/D board, keep in mind that some A/D boards have programmable voltage ranges, and others set the voltage range via switches on the board. In either case, the desired range must be passed to this method. Refer to Table 7 on page 208 for a list of valid range settings.

- **dataVal**
  An integer count value (typically, one returned from an A/D board).

- **engUnits**
  The single precision 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.

Note:

This method is not supported for hardware with resolution greater than 16 bits.

The default resolution of this method is 12 bits, so if the device has neither analog input nor analog output, the result will be a 12 bit conversion.

If the device has both analog input and analog output, the resolution and transfer function of the D/A converter on the device is used.
WinBufToEngArray()

Transfers integer values from a Windows buffer to a 2D array as engineering unit values.

The conversion from integer values to engineering unit values uses the A/D resolution of the board associated with the MccBoard object.

This method is usually used to obtain values compatible with those required by Measurement Studio waveform display controls from a Windows buffer containing data from a method such as AInScan() or DaqInScan().

The converted values are transferred to the 2D array based on the gain, firstPoint, count, and numChannels parameters.

Member of the MccBoard class.

Function prototype:

**VB .NET:**
```vbnet
Public Function WinBufToEngArray(ByVal gain As MccDaq.Range, ByVal memHandle As Integer, ByVal engUnits As Double(,), ByVal firstPoint As Integer, ByVal count As Integer, ByVal numChannels As Integer) As MccDaq.ErrorInfo
```

**C# .NET:**
```csharp
public MccDaq.ErrorInfo WinBufToEngArray(MccDaq.Range gain, int memHandle, double[,] engUnits, int firstPoint, int count, int numChannels)
```

**Parameters:**

- **gain**
  
  The range to use for converting scan data. This range should be the same as the range specified for AInScan() or DaqInScan().

- **gainArray**
  
  The array containing the A/D range values used during the analog input scan.

  If a gain queue was not used for the scan, this array should only contain 1 element whose value matches the gain used during the scan. If a gain queue was used during the scan, this array should match the gainArray value used in ALoadQueue() or DaqInScan().

  If the corresponding range in the gainArray is set to NotUsed (MccDaq.Range.NotUsed), raw data is returned in engineering unit values.

- **gainCount**
  
  The number of array elements in gainArray. Set gainCount to 1 when no gain queue was used for the scan. If a gain queue was used for the scan, this number should match the number of gain queue pairs defined in ALoadQueue() or DaqInScan().

- **memHandle**
  
  The handle to the memory buffer holding the raw data to be converted to engineering units. This value should be large enough to hold (count x numChannels) samples.
engUnits: The array to hold the converted data. This array must be allocated by the user, and must be large enough to hold `count` samples. The first dimension should be the number of channels. The second dimension should equal the number of points/channel.

`firstPoint`: The index into the raw data memory buffer that holds the first sample of the first channel to be converted. The index into the raw memory is `(firstPoint x numChannels)` so that converted data always starts with the first channel specified in the scan. For example, if `firstPoint` is 14 and the number of channels is 8, the index of the first converted sample is 112.

`count`: The number of samples per channel to convert to engineering units. `count` should not exceed Windows buffer size / (numChannels - firstPoint).

`numChannels`: The number of channels of data stored in the existing array to be transferred.

**Returns:**

An `ErrorInfo` object that indicates the status of the operation.

**Notes:**

If `gainCount` is greater than one, the conversions cycle through the array until `count` samples have been converted. When only one gain is specified, that gain is applied to all conversions. Data is returned in engineering unit values as a two-dimensional array.

If the corresponding range in the `gainArray` is set to `NotUsed`, raw data is returned in engineering unit values.
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 (for example `NOERRORS`), while the Universal Library for .NET error names are mixed case (for example `NOERRORS`).

<table>
<thead>
<tr>
<th>Error number</th>
<th>Error name</th>
<th>Error message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOERRORS</td>
<td>No error has occurred</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The function executed successfully.</td>
</tr>
<tr>
<td>1</td>
<td>BADBOARD</td>
<td>Invalid board number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <code>BoardNum</code> argument that was specified does not match</td>
</tr>
<tr>
<td></td>
<td></td>
<td>any of the boards that are listed in the configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>file. Run the configuration program to check which</td>
</tr>
<tr>
<td></td>
<td></td>
<td>board numbers are configured.</td>
</tr>
<tr>
<td>2</td>
<td>DEADDIGITALDEV</td>
<td>Digital device is not responding - is base address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>correct?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The digital device on the specified board is not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>responding. Either the board was installed incorrectly</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or the board is defective. Run the configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>program and make sure that the correct board was</td>
</tr>
<tr>
<td></td>
<td></td>
<td>installed.</td>
</tr>
<tr>
<td>3</td>
<td>DEADCOUNTERDEV</td>
<td>Counter device is not responding - is base address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>correct?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The counter device on the specified board is not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>responding. Either the board was installed incorrectly</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or the board is defective. Run the configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>program and make sure that the correct board was</td>
</tr>
<tr>
<td></td>
<td></td>
<td>installed.</td>
</tr>
<tr>
<td>4</td>
<td>DEADDADDEV</td>
<td>D/A is not responding - is base address correct?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The D/A device on the specified board is not responding.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Either the board was installed incorrectly or the board</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is defective. Run the configuration program and make</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sure that the correct board was installed.</td>
</tr>
<tr>
<td>5</td>
<td>DEADADDEV</td>
<td>A/D is not responding - is base address correct?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The A/D device on the specified board is not responding.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Either the board was installed incorrectly or the board</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is defective. Run the configuration program and make</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sure that the correct board was installed.</td>
</tr>
<tr>
<td>6</td>
<td>NOTDIGITALCONF</td>
<td>Selected board does not have digital I/O</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A digital I/O function or method was called with a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>board number that referred to a board that does not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>support digital I/O. Run the configuration program to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>see which type of board that board number refers to.</td>
</tr>
<tr>
<td>7</td>
<td>NOTCOUNTERCONF</td>
<td>Selected board does not have a counter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A counter function or method was called with a board</td>
</tr>
<tr>
<td></td>
<td></td>
<td>number that referred to a board that does not have a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>counter. Run the configuration program to see which type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of board that board number refers to.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>NOTDACONF</td>
<td>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.</td>
</tr>
<tr>
<td>9</td>
<td>NOTADCONF</td>
<td>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.</td>
</tr>
<tr>
<td>10</td>
<td>NOTMUXCONF</td>
<td>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.</td>
</tr>
<tr>
<td>11</td>
<td>BADPORTNUM</td>
<td>Invalid digital port number The port number specified for a digital I/O function or method does not exist on the specified board.</td>
</tr>
<tr>
<td>12</td>
<td>BADCOUNTERDEVNUM</td>
<td>Invalid counter device The CounterNum argument specified for a counter function or method references a counter that does not exist on the specified board.</td>
</tr>
<tr>
<td>13</td>
<td>BADDADDEVNUM</td>
<td>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.</td>
</tr>
<tr>
<td>14</td>
<td>BADSAMPLEMODE</td>
<td>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.</td>
</tr>
<tr>
<td>15</td>
<td>BADINT</td>
<td>Board configured for invalid interrupt level No interrupt was selected in InstaCal and one is required, or the board is set for &quot;compatible mode&quot; 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 &quot;enhanced mode&quot;, or change the interrupt level with the configuration program to something less than 8.</td>
</tr>
<tr>
<td>16</td>
<td>BADADCHAN</td>
<td>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.</td>
</tr>
<tr>
<td>17</td>
<td>BADCOUNT</td>
<td>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.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>18</td>
<td>BadCNTRCONFIG</td>
<td>Invalid counter configuration specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An invalid Config argument was passed to cbC8254Config()/C8254Config(). The only legal values are HIGHONLASTCOUNT, ONESHOT, RATEGENERATOR, SQUAREWAVE, SOFTWARESTROBE and HARDWARESTROBE.</td>
</tr>
<tr>
<td>19</td>
<td>BadDAVAL</td>
<td>Invalid D/A value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 &quot;Note on Basic signed integers&quot; at the beginning of the &quot;Counter Boards&quot; chapter in the Universal Library User's Guide available on our web site at <a href="http://www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf">www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf</a>).</td>
</tr>
<tr>
<td>20</td>
<td>BadDACHAN</td>
<td>Invalid D/A channel number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>22</td>
<td>AlreadyActive</td>
<td>Background operation already in progress</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>23</td>
<td>PageOverrun</td>
<td>DMA transfer crossed page boundary, may have gaps in data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>24</td>
<td>BadRate</td>
<td>Invalid sampling rate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>25</td>
<td>CompATMODE</td>
<td>Board switches set for Compatible mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>26</td>
<td>TrigState</td>
<td>Incorrect initial trigger state - trigger must start at TTL low</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Boards that use &quot;polled gate&quot; triggering require that the trigger be &quot;off&quot; 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.</td>
</tr>
<tr>
<td>27</td>
<td>ADStatusHung</td>
<td>A/D is not responding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
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 or output 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 SINGLIO 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.
<table>
<thead>
<tr>
<th>Error number</th>
<th>Error name</th>
<th>Error message</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>CONVERTDMA</td>
<td>CONVERTDATA cannot be used with DMAIO and BACKGROUND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The CONVERTDATA and BACKGROUND options cannot be used together when the board is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>transferring data via DMA. Possible solutions include: Use</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cbAConvertData() / AConvertData() to convert the data after it is collected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Don't use BACKGROUND option. Use BLOCKIO option if your A/D board supports it.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use SINGLEIO option if your computer is fast enough to support the selected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sampling rate.</td>
</tr>
<tr>
<td>39</td>
<td>DTCONNECTERR</td>
<td>Board does not support DTCONNECT option</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The DTCONNECT Option was passed to an analog input function or method.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The selected board does not support that option.</td>
</tr>
<tr>
<td>40</td>
<td>FORECONTINUOUS</td>
<td>CONTINUOUS can only be run with BACKGROUND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The CONTINUOUS option was passed to a function or method without also setting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the BACKGROUND option. This is not allowed. Any time you set the CONTINUOUS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>option you must also set the BACKGROUND option.</td>
</tr>
<tr>
<td>41</td>
<td>BADBOARDTYPE</td>
<td>This function or method can not be used with this board</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An attempt was made to call a function or method for a board that does not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>support that function or method.</td>
</tr>
<tr>
<td>42</td>
<td>WRONGDIGCONFIG</td>
<td>Digital port not configured correctly for requested operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Some of the digital bits or ports (FIRSTPORTA - EIGHTHPORTCH) must be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>configured as inputs OR outputs but not both. An attempt was made to use a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>digital input function or method on a port or bit that was configured as an</td>
</tr>
<tr>
<td></td>
<td></td>
<td>output or vice versa. Use cbDConfigPort() / DConfigPort() or cbDConfigBit()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/ DConfigBit() to switch a port's (or bit's) direction. If the board you are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>using contains configurable port types and you do not call cbDConfigPort() /</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DConfigPort() or cbDConfigBit() / DConfigBit() in your program, then all of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the configurable ports will be in an unknown state (input or output).</td>
</tr>
<tr>
<td>43</td>
<td>NOTCONFIGURABLE</td>
<td>This digital port is not configurable (it's an In/Out port)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cbDConfigPort() / DConfigPort() or cbDConfigBit() / DConfigBit() was called</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for a port that is not configurable. Check the PortNum argument passed to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cbDConfigPort() and make sure that it is in the range FIRSTPORTA - EIGHTHPORTCH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If PortNum is AUXPORT, make sure your hardware supports configuration of this</td>
</tr>
<tr>
<td></td>
<td></td>
<td>port type. If not then there is no need to call this function or method.</td>
</tr>
<tr>
<td>44</td>
<td>BADPORTCONFIG</td>
<td>Invalid digital port configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Direction argument passed to cbDConfigPort() / DConfigPort() or cbDConfig</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit() / DConfigBit() is invalid. It must be set to either DIGITALIN or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIGITALOUT.</td>
</tr>
<tr>
<td>45</td>
<td>BADFIRSTPOINT</td>
<td>FirstPoint number is not valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The FirstPoint argument to cbFileRead() / FileRead() is invalid. It is either</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a negative number or it is larger then the number of points in the file.</td>
</tr>
<tr>
<td>46</td>
<td>ENDOFFILE</td>
<td>Attempted to read past the end of the file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cbFileRead() / FileRead() attempted to read beyond the end of the file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check the file length with cbFileGetInfo() / FileGetInfo() and make sure that</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the FirstPoint and Count arguments to cbFileRead() / FileRead() are correct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for that file length.</td>
</tr>
<tr>
<td>47</td>
<td>NOT8254CTR</td>
<td>This board does not have an 8254 counter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cbC8254Config() / C8254Config() was called for a board that has a counter but</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not an 8254 counter. This function or method can only be used with an 8254</td>
</tr>
<tr>
<td></td>
<td></td>
<td>counter.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>48</td>
<td>NOT9513CTR</td>
<td>This board does not have a 9513 counter. <code>cbC9513Config()</code> 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.</td>
</tr>
<tr>
<td>49</td>
<td>BADTRIGTYPE</td>
<td>Invalid <code>TrigType</code> <code>cbATrig()</code>/<code>ATrig()</code> was called with an invalid <code>TrigType</code> argument. It must be set to either TRIGABOVE or TRIGBELOW.</td>
</tr>
<tr>
<td>50</td>
<td>BADTRIGVALUE</td>
<td>Invalid <code>TrigValue</code> <code>cbATrig()</code>/<code>ATrig()</code> was called with an invalid <code>TrigValue</code> argument. It must be in the range 0 to 4095 for 12-bit boards or 0 to 65535 for 16-bit boards (see the &quot;Note on Basic signed integers&quot; at the beginning of the &quot;Counter Boards&quot; chapter in the Universal Library User's Guide, available on our web site at <a href="http://www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf">www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf</a>).</td>
</tr>
<tr>
<td>52</td>
<td>BADOPTION</td>
<td>Invalid Option specified for this function or method. The <code>Options</code> argument contains an option that is not valid for this function or method.</td>
</tr>
</tbody>
</table>
| 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()`.
<p>| 55           | BADDIVIDER      | Invalid <code>FOutDivider</code> value. The <code>FOutDivider</code> argument to <code>cbC9513Init()</code> (<code>C9513Init()</code>) is not valid. It must be in the range 0 to 15. |
| 56           | BADSOURCE       | Invalid <code>FOutSource</code> value. The <code>FOutSource</code> argument to <code>cbC9513Init()</code> (<code>C9513Init()</code> 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 (for example 0 to 15). |
| 57           | BADCOMPARE      | Invalid <code>Compare</code> value. One or both of the compare arguments to <code>cbC9513Init()</code> (<code>C9513Init()</code>) are not valid. They must be set to (CB)ENABLED or (CB)DISABLED (1 or 0). |
| 58           | BADTIMEOFDAY    | Invalid <code>TimeOfDay</code> value. The <code>TimeOfDay</code> argument to <code>cbC9513Init()</code> (<code>C9513Init()</code>) is not valid. It must be set to either (CB)ENABLED or (CB)DISABLED (1 or 0). |
| 59           | BADGATEINTERVAL| Invalid <code>GateInterval</code> value. The <code>GateInterval</code> argument to <code>cbCFreqIn()</code>/<code>CFreqIn()</code> is not valid. It must be greater than 0. |
| 60           | BADGATECNTRL    | Invalid <code>GateControl</code> value. The <code>GateControl</code> argument to <code>cbC9513Config()</code>/<code>C9513Config()</code> is not valid. It must be in the range 0 to 7. |
| 61           | BADCOUNTEREDGE  | Invalid <code>CounterEdge</code> value. The <code>CounterEdge</code> argument to <code>cbC9513Config()</code>/<code>C9513Config()</code> is not valid. It must be set to either POSITIVEEDGE or NEGATIVEEDGE. |
| 62           | BADSPCLGATE     | Invalid <code>SpecialGate</code> value. The <code>SpecialGate</code> argument to <code>cbC9513Config()</code>/<code>C9513Config()</code> is not valid. It must be set to either (CB)ENABLED or (CB)DISABLED (1 or 0). |</p>
<table>
<thead>
<tr>
<th>Error number</th>
<th>Error name</th>
<th>Error message</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>BADRELOAD</td>
<td>Invalid Reload value. The Reload argument to cbC9513Config() (C9513Config()) is not valid. It must be set to either LOADREG or LOADANDHOLDREG.</td>
</tr>
<tr>
<td>64</td>
<td>BADRECYCLEFLAG</td>
<td>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).</td>
</tr>
<tr>
<td>65</td>
<td>BADBCDFLAG</td>
<td>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).</td>
</tr>
<tr>
<td>66</td>
<td>BADDIRECTION</td>
<td>Invalid CountDirection value. The CountDirection argument to cbC9513Config() (C9513Config()) is not valid. It must be set to either COUNTUP or COUNTDOWN.</td>
</tr>
<tr>
<td>67</td>
<td>BADOUTCONTROL</td>
<td>Invalid OutputControl value. The OutputControl argument to cbC9513Config() (C9513Config()) is not valid. It must be set to either ALWAYSLOW, HIGHPULSEONTC, TOGGLEONTC, DISCONNECTED or LOWPULSEONTC.</td>
</tr>
<tr>
<td>68</td>
<td>BADBITNUMBER</td>
<td>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. Refer to board-specific information in the Universal Library User's Guide or in your hardware manual.</td>
</tr>
<tr>
<td>69</td>
<td>NONEENABLED</td>
<td>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.</td>
</tr>
<tr>
<td>70</td>
<td>BADCTRCONTROL</td>
<td>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.</td>
</tr>
<tr>
<td>71</td>
<td>BADEXPCHAN</td>
<td>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 &gt;= 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.</td>
</tr>
<tr>
<td>72</td>
<td>WRONGADRANGE</td>
<td>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.</td>
</tr>
<tr>
<td>73</td>
<td>OUTOFRANGE</td>
<td>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.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>74</td>
<td>BADTEMPSCALE</td>
<td>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.</td>
</tr>
<tr>
<td>76</td>
<td>NOQUEUE</td>
<td>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.</td>
</tr>
<tr>
<td>77</td>
<td>CONTINUOUCOUNT</td>
<td>Count must be &gt; 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). Refer to the board-specific information in the Universal Library User's Guide, available on our web site at <a href="http://www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf">www.mccdaq.com/PDFmanuals/sm-ul-user-guide.pdf</a> or in your hardware manual.</td>
</tr>
<tr>
<td>78</td>
<td>UNDERRUN</td>
<td>D/A FIFO went empty during output. The specified D/A output rate could not be sustained. Try increasing the size of the data buffer or reducing the update rate to eliminate the error.</td>
</tr>
<tr>
<td>79</td>
<td>BADMEMMODE</td>
<td>Invalid memory mode specified. The memory mode that was selected with cbMemSetDTMode() is not one of the valid modes.</td>
</tr>
<tr>
<td>80</td>
<td>FREQOVERRUN</td>
<td>Measured frequency too high for selected gating interval. The GateInterval argument used with cbCFreqIn() 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.</td>
</tr>
<tr>
<td>81</td>
<td>NOCJCCHAN</td>
<td>A CJC Channel must be configured to make temperature measurements. When the board was installed (with the InstaCal 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.</td>
</tr>
<tr>
<td>82</td>
<td>BADCHIPNUM</td>
<td>Invalid ChipNum specified. An invalid ChipNum argument was used with cbC9513Init(). If the board is CTR05, set ChipNum to 0. If the board is a CTR10, set ChipNum to either 0 or 1.</td>
</tr>
<tr>
<td>83</td>
<td>DIGNOTENABLED</td>
<td>The digital I/O on this board is not enabled. When the board was installed (with the InstaCal 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.</td>
</tr>
<tr>
<td>84</td>
<td>CONVERT16BITS</td>
<td>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() or call cbAConvertData(), 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.)</td>
</tr>
<tr>
<td>85</td>
<td>NOMEMBOARD</td>
<td>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 InstaCal.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>86</td>
<td>DTACTIVE</td>
<td>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.</td>
</tr>
<tr>
<td>87</td>
<td>NOTMEMCONF</td>
<td>Specified board is not a memory board. The specified board is not a memory board. This function or method only works with memory boards.</td>
</tr>
<tr>
<td>88</td>
<td>ODDCHAN</td>
<td>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.</td>
</tr>
<tr>
<td>89</td>
<td>CTRNOINIT</td>
<td>Counter was not configured or initialized. You attempted to use cbCLoad() or cbCIn() (CLoad() or CIn()) before initializing and configuring the counter.</td>
</tr>
<tr>
<td>90</td>
<td>NOT8536CTR</td>
<td>This board does not have an 8536 counter chip. Attempt to use 8536 initialization or configuration on board without 8536 chip.</td>
</tr>
<tr>
<td>91</td>
<td>FREERUNNING</td>
<td>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.</td>
</tr>
<tr>
<td>92</td>
<td>INTERRUPTED</td>
<td>Operation interrupted with Ctrl-C key. A foreground operation was stopped before completion because either the Ctrl-C or Ctrl-Break keys were pressed.</td>
</tr>
<tr>
<td>93</td>
<td>NOSELECTORS</td>
<td>No selector could be allocated. A Windows selector required by the library could not be allocated. Close any open Windows applications that are not required to be running and try again.</td>
</tr>
<tr>
<td>94</td>
<td>NOBURSTMODE</td>
<td>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.</td>
</tr>
<tr>
<td>95</td>
<td>NOTWINDOWSFUNC</td>
<td>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.</td>
</tr>
<tr>
<td>96</td>
<td>NOTSIMULCONF</td>
<td>Board not configured for SIMULTANEOUS option. The configuration file of the D/A board in InstaCal 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.</td>
</tr>
<tr>
<td>97</td>
<td>EVENODDMISMATCH</td>
<td>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.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>98</td>
<td>M1RATEWARNING</td>
<td>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.</td>
</tr>
<tr>
<td>99</td>
<td>NOTRS485</td>
<td>An attempt was made to call cbRS485() / RS485() with a board that is not RS485 compatible.</td>
</tr>
<tr>
<td>100</td>
<td>NOTDOSFUNCTION</td>
<td>The function that was called is not available in the DOS version of the Universal Library.</td>
</tr>
<tr>
<td>101</td>
<td>RANGEMISMATCH</td>
<td>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.</td>
</tr>
<tr>
<td>102</td>
<td>CLOCKTOOSLOW</td>
<td>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 InstaCal.</td>
</tr>
<tr>
<td>103</td>
<td>BADCALFACTORS</td>
<td>The selected board uses software calibration and the stored calibration factors are invalid. Run InstaCal and calibrate the board before using it.</td>
</tr>
<tr>
<td>104</td>
<td>BADCONFIGTYPE</td>
<td>Invalid configuration information type specified. An invalid ConfigType argument was passed to either cbGetConfig() or cbSetConfig().</td>
</tr>
<tr>
<td>105</td>
<td>BADCONFIGITEM</td>
<td>Invalid configuration item specified. An invalid ConfigItem argument was passed to either cbGetConfig() or cbSetConfig().</td>
</tr>
<tr>
<td>106</td>
<td>NOPCMCIABOARD</td>
<td>Cannot access the PCMCIA board. Cannot access the specified PCMCIA board. Make sure that the PCMCIA Card &amp; Socket Services are installed correctly and that the board was installed in the system correctly via InstaCal.</td>
</tr>
<tr>
<td>107</td>
<td>NOBACKGROUND</td>
<td>The BACKGROUND option was used and the specified board does not support background operation.</td>
</tr>
<tr>
<td>108</td>
<td>STRINGTOOSHORT</td>
<td>The string argument is too short for the string being returned. The string passed to a library function or method is too 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.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 109          | CONVERTTEXTMEM             | 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. |
| 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. |
| 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 InstaCal settings. |
| 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. |
| 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 InstaCal menu. |
| 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 InstaCal menu. |
| 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. |
| 117          | NOEXTCONTINUOUS            | CONTINUOUS mode cannot be used with EXTMEMORY  
CONTINUOUS mode is ignored when used with the EXTMEMORY option. |
| 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. |
| 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. |
<table>
<thead>
<tr>
<th>Error number</th>
<th>Error name</th>
<th>Error message</th>
</tr>
</thead>
<tbody>
<tr>
<td>121</td>
<td>BADPCMSLOTREF</td>
<td>NO PCM Card was found in the specified slot. This is usually caused by swapping PCMCIA cards and not re-running InstaCal. Run InstaCal.</td>
</tr>
<tr>
<td>122</td>
<td>AMBIGPCMSLOTREF</td>
<td>Two identical PCM cards found. Please specify exact slot in InstaCal. This error occurs in DOS mode only when InstaCal is configured for a PCMCIA card in &quot;any slot&quot;. To correct the problem, run InstaCal. Go to the Install menu and pop up the board's menu. Highlight PCMCIA slot and choose either &quot;0&quot; or &quot;1&quot;.</td>
</tr>
<tr>
<td>123</td>
<td>BADSENSORTYPE</td>
<td>Invalid sensor type selected in InstaCal. 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 InstaCal menu.</td>
</tr>
<tr>
<td>126</td>
<td>CFGFILENOTFOUND</td>
<td>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.</td>
</tr>
<tr>
<td>127</td>
<td>NOVDDINSTALLED</td>
<td>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</td>
</tr>
<tr>
<td>128</td>
<td>NOWINDOWSMEMORY</td>
<td>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.</td>
</tr>
<tr>
<td>129</td>
<td>OUTOFDOSMEMORY</td>
<td>Not enough DOS memory available. Try closing down any unneeded programs that are running.</td>
</tr>
<tr>
<td>130</td>
<td>OBSOLETEOPTION</td>
<td>Obsolete option specified for cbSetConfig / cbGetConfig. The specified configuration item is no longer supported in the 32-bit version of the Universal Library.</td>
</tr>
<tr>
<td>131</td>
<td>NOPCMREGKEY</td>
<td>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 technical support for assistance.</td>
</tr>
<tr>
<td>132</td>
<td>NOCBUL32SYS</td>
<td>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 technical support for assistance.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>133</td>
<td>NODMAMEMORY</td>
<td>No DMA memory available to device driver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Windows device driver could not allocate the minimum required amount of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>memory for DMA. If you are sampling at slower speeds, you can specify SINGLEIO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in the Options argument to cbAInScan() / AInScan(). This will prevent the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>library from attempting to use DMA. In general though, this error should</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not ordinarily occur. Contact technical support for assistance.</td>
</tr>
<tr>
<td>134</td>
<td>IRQNOTAVAILABLE</td>
<td>IRQ not available</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Interrupt Level that was specified for the board (in InstaCal) conflicts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>with another board in your computer. Try switching to a different interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>level.</td>
</tr>
<tr>
<td>135</td>
<td>NOT7266CTR</td>
<td>This board does not have an LS7266 counter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This function or method can only be used with a board that contains an LS7266</td>
</tr>
<tr>
<td></td>
<td></td>
<td>chip. These chips are used on various quadrature encoder input boards.</td>
</tr>
<tr>
<td>136</td>
<td>BADQUADRATURE</td>
<td>Invalid Quadrature argument passed to cbC7266Config()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Quadrature argument must be set to either NO_QUAD, X1_QUAD, X2_QUAD, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X4_QUAD.</td>
</tr>
<tr>
<td>137</td>
<td>BADCOUNTMODE</td>
<td>Invalid CountingMode argument passed to cbC7266Config()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The CountingMode argument must be set to either NORMAL_MODE, RANGE_LIMIT,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO_RECYCLE, or MODULO_N.</td>
</tr>
<tr>
<td>138</td>
<td>BADCOUNTMODE</td>
<td>Invalid DataEncoding argument passed to cbC7266Config()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The DataEncoding argument must be set to either BCD_ENCODING or BINARY_ENCODING</td>
</tr>
<tr>
<td>139</td>
<td>BADINDEXMODE</td>
<td>Invalid IndexMode argument passed to cbC7266Config()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The IndexMode argument must be set to either INDEX_DISABLED, LOAD_CTR, LOAD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OUT_LATCH, or RESET_CTR.</td>
</tr>
<tr>
<td>140</td>
<td>BADINDEXMODE</td>
<td>Invalid InvertIndex argument passed to cbC7266Config()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The InvertIndex argument must be set to either (CB)ENABLED or (CB)DISABLED.</td>
</tr>
<tr>
<td>141</td>
<td>BADFLAGPINS</td>
<td>Invalid FlagPins argument passed to cbC7266Config()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The FlagPins argument must be set to either CARRY_BORROW, COMPARE_BORROW,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CARRY_BORROW_UPDOWN, or INDEX_ERROR.</td>
</tr>
<tr>
<td>142</td>
<td>NOCTRSTATUS</td>
<td>This board does not support cbCStatus()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This board does not return any status information.</td>
</tr>
<tr>
<td>143</td>
<td>NOGATEALLOWED</td>
<td>Gating can not be used when indexing is enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gating and indexing can not be used simultaneously. If Gating is set to (CB)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENABLED, then IndexMode must be set to INDEX_DISABLED.</td>
</tr>
<tr>
<td>144</td>
<td>NOINDEXALLOWED</td>
<td>Indexing not allowed in non-quadrature mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Indexing is not supported when Quadrature argument is set to NO_QUAD.</td>
</tr>
<tr>
<td>145</td>
<td>OPENCONNECTION</td>
<td>Temperature input has open connection</td>
</tr>
<tr>
<td>146</td>
<td>BMCONTINUOUSCOUNT</td>
<td>Count must be integer multiple of packet size for Continuous mode</td>
</tr>
</tbody>
</table>
### Error Codes

<table>
<thead>
<tr>
<th>Error number</th>
<th>Error name</th>
<th>Error message</th>
</tr>
</thead>
<tbody>
<tr>
<td>147</td>
<td>BADCALLBACKFUNC</td>
<td>Invalid pointer to callback function or delegate passed as argument</td>
</tr>
<tr>
<td>148</td>
<td>MBUSINUSE</td>
<td>Metrabus in use</td>
</tr>
<tr>
<td>149</td>
<td>MBUSNOCTLR</td>
<td>Metrabus I/O card has no configured controller card</td>
</tr>
<tr>
<td>150</td>
<td>BADEVENTTYPE</td>
<td>Invalid EventType specified for this board&lt;br&gt;Although this board does support <code>cbEnableEvent()</code>/<code>EnableEvent()</code>, it does not support one or more of the event types specified.</td>
</tr>
<tr>
<td>151</td>
<td>ALREADYENABLED</td>
<td>Event handler already enabled for this event type&lt;br&gt;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 <code>cbDisableEvent()</code>/<code>DisableEvent()</code>.</td>
</tr>
<tr>
<td>152</td>
<td>BADEVENTSIZE</td>
<td>Invalid event count has been specified&lt;br&gt;The <code>ON_DATA_AVAILABLE</code> event requires an event count greater than (0).</td>
</tr>
<tr>
<td>153</td>
<td>CANTINSTALLEVENT</td>
<td>Unable to install event handler&lt;br&gt;An internal error occurred while trying to setup the event handling.</td>
</tr>
<tr>
<td>154</td>
<td>BADBUFFERSIZE</td>
<td>Buffer is too small for operation&lt;br&gt;The memory allocated by <code>cbWinBufAlloc()</code>/<code>WinBufAlloc()</code> is too small to hold all the data specified in the operation.</td>
</tr>
<tr>
<td>155</td>
<td>BADAIMODE</td>
<td>Invalid analog input mode&lt;br&gt;Invalid analog input mode (<code>RSE</code>, <code>NRSE</code>, <code>DIFF</code>).</td>
</tr>
<tr>
<td>156</td>
<td>BADSIGNAL</td>
<td>Invalid signal type specified&lt;br&gt;The specified signal type does not exist, or is not valid for signal direction specified.</td>
</tr>
<tr>
<td>157</td>
<td>BADCONNECTION</td>
<td>Invalid connection&lt;br&gt;The specified connection does not exist, or is not valid for the signal type and direction specified.</td>
</tr>
<tr>
<td>158</td>
<td>BADINDEX</td>
<td>Invalid index specified&lt;br&gt;For <code>Index &gt; 0</code>, indicates that the specified index is beyond the end of the internal list of output connections assigned to the specified signal type.</td>
</tr>
<tr>
<td>159</td>
<td>NOCONNECTION</td>
<td>Invalid connection&lt;br&gt;No connection is assigned to the specified signal.</td>
</tr>
<tr>
<td>160</td>
<td>BADBURSTIOCOUNT</td>
<td>Count cannot be greater than the FIFO size for BURSTIO mode. Also, Count must be integer multiple of number of channels in the scan.&lt;br&gt;When using BURSTIO mode, the count entered cannot be larger than the FIFO size.</td>
</tr>
<tr>
<td>161</td>
<td>DEADDEV</td>
<td>Device has stopped responding. Please check connections.&lt;br&gt;Check cable connections to USB device and to your computer's USB port.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>163</td>
<td>INVALIDACCESS</td>
<td>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 cbOut()/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.</td>
</tr>
<tr>
<td>164</td>
<td>UNAVAILABLE</td>
<td>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.</td>
</tr>
<tr>
<td>165</td>
<td>NOTREADY</td>
<td>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.</td>
</tr>
<tr>
<td>169</td>
<td>BITUSEDFORALARM</td>
<td>The specified bit is used for alarm. You attempted to set the state of a digital output bit that is configured as an alarm input.</td>
</tr>
<tr>
<td>170</td>
<td>PORTUSEDFORALARM</td>
<td>One or more bits on the specified port are used for alarm. You attempted to write to a digital output port that contains a bit configured as an alarm input.</td>
</tr>
<tr>
<td>171</td>
<td>PACEROVERRUN</td>
<td>Pacer overrun; external clock rate too fast. You set the external clock rate to a value that is higher than the rate supported by the board.</td>
</tr>
<tr>
<td>172</td>
<td>BADCHANATYPE</td>
<td>Invalid channel type specified. You set the channel type to a type that is not supported by the board.</td>
</tr>
<tr>
<td>173</td>
<td>BADTRIGSENSE</td>
<td>Invalid trigger sensitivity specified. You set the trigger sensitivity to a value that is not supported by the board.</td>
</tr>
<tr>
<td>174</td>
<td>BADTRIGCHAN</td>
<td>Invalid trigger channel specified. You set the trigger channel to a value that is not supported by the board.</td>
</tr>
<tr>
<td>175</td>
<td>BADTRIGLEVEL</td>
<td>Invalid trigger level specified. You set the trigger level to a value that is not supported by the board.</td>
</tr>
<tr>
<td>176</td>
<td>NOPRETRIGMODE</td>
<td>Pretrigger mode is not supported for the specified trigger type. You selected a trigger source that does not support pre-trigger data acquisitions.</td>
</tr>
<tr>
<td>177</td>
<td>BADDEBOUNCETIME</td>
<td>Invalid debounce timing specified. You set the debounce time to a value that is not supported by the board.</td>
</tr>
<tr>
<td>178</td>
<td>BADDEBOUNCETRIGMODE</td>
<td>Invalid debounce trigger mode specified. You set the debounce trigger mode to a value that is not supported by the board.</td>
</tr>
<tr>
<td>179</td>
<td>BADMAPPEDCOUNTER</td>
<td>Invalid mapped counter specified. You mapped to a counter input channel that is not supported by the board.</td>
</tr>
<tr>
<td>180</td>
<td>BADCOUNTERMODE</td>
<td>Invalid counter mode specified. This function cannot be used with the current mode of the specified counter.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>181</td>
<td>BADTCCHANMODE</td>
<td>Single-ended mode cannot be used for temperature input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You specified single-ended mode for use with a temperature input.</td>
</tr>
<tr>
<td>182</td>
<td>BADFREQUENCY</td>
<td>Invalid frequency specified.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You specified a frequency value that is not supported by the board.</td>
</tr>
<tr>
<td>183</td>
<td>BADEVENTPARAM</td>
<td>Invalid event parameter specified.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You specified an event parameter that is not supported by the board.</td>
</tr>
<tr>
<td>184</td>
<td>NONETIFC</td>
<td>No interface devices were found with the required PAN and channel.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No interface devices were detected whose PAN ID and RF channel number match those of a remote device.</td>
</tr>
<tr>
<td>185</td>
<td>DEADNETIFC</td>
<td>The interface device(s) with the required PAN and channel has failed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Please check the connection.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The interface device whose PAN ID and RF channel number match a remote device is not responding. Check that the USB connection to the computer.</td>
</tr>
<tr>
<td>186</td>
<td>NOREMOTENACK</td>
<td>The remote device is not responding to commands and queries.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Please check the device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The wireless remote device is not responding. Check that the device is powered, that its PAN ID and RF channel match the interface device, and that the LEDs are functioning.</td>
</tr>
<tr>
<td>187</td>
<td>INPUTTIMEOUT</td>
<td>The device acknowledged the operation, but has not completed before the timeout.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The operation was acknowledged but has timed out before it was completed.</td>
</tr>
<tr>
<td>188</td>
<td>MISMATCHSETPOINTCOUNT</td>
<td>Number of setpoints is not equal to number of channels with a setpoint flag set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set the number of setpoints equal to the number of channels with a setpoint flag set.</td>
</tr>
<tr>
<td>189</td>
<td>INVALIDSETPOINTLEVEL</td>
<td>Setpoint level is outside channel range.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You specified a setpoint level that is outside of the range supported by the board.</td>
</tr>
<tr>
<td>190</td>
<td>INVALIDSETPOINTOUTPUTTYPE</td>
<td>Setpoint Output Type is invalid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You specified a setpoint output type that is not supported by the board.</td>
</tr>
<tr>
<td>191</td>
<td>INVALIDSETPOINTOUTPUTVALUE</td>
<td>Setpoint Output Value is outside channel range.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You specified a setpoint output value that is outside of the range supported by the board.</td>
</tr>
<tr>
<td>192</td>
<td>INVALIDSETPOINTLIMITS</td>
<td>Setpoint Comparison limit B greater than Limit A.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set the setpoint comparison value for limit A to be larger than the value set for limit B.</td>
</tr>
<tr>
<td>193</td>
<td>STRINGTOOLONG</td>
<td>The string length entered is too long for this operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enter a string up to the maximum number of characters specified for the function or method that you are using.</td>
</tr>
<tr>
<td>194</td>
<td>INVALIDLOGIN</td>
<td>An invalid user name or password has been entered.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check that the password and user name entered were correct. If either has been lost, use the device reset button to reset the device to default values.</td>
</tr>
<tr>
<td>195</td>
<td>SESSIONINUSE</td>
<td>Device session is already in use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Another user is currently logged in to device session. Only one device session can be opened at a time.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>196</td>
<td>NOEXTPOWER</td>
<td>External power is not connected. External power is required. Connect the device to an external power supply.</td>
</tr>
<tr>
<td>200–299</td>
<td>Internal 16-bit error</td>
<td>Internal error occurred in library. See details below:</td>
</tr>
<tr>
<td>201</td>
<td>CANT_LOCK_DMA_BUF</td>
<td>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.</td>
</tr>
<tr>
<td>202</td>
<td>DMA_IN_USE</td>
<td>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.</td>
</tr>
<tr>
<td>203</td>
<td>BAD_MEM_HANDLE</td>
<td>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().</td>
</tr>
<tr>
<td>300–399</td>
<td>Internal 32-bit error</td>
<td>Error in 32-bit Windows library. See details below.</td>
</tr>
<tr>
<td>304</td>
<td>CFG_FILE_READ_FAILURE</td>
<td>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.</td>
</tr>
<tr>
<td>305</td>
<td>CFG_FILE_WRITE_FAILURE</td>
<td>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.</td>
</tr>
<tr>
<td>308</td>
<td>CFGFILE_CANT_OPEN</td>
<td>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.</td>
</tr>
<tr>
<td>325</td>
<td>BAD_RTD_CONVERSION</td>
<td>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().</td>
</tr>
<tr>
<td>326</td>
<td>NO_PCI_BIOS</td>
<td>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.</td>
</tr>
<tr>
<td>327</td>
<td>BAD_PCI_INDEX</td>
<td>Specified PCI board not detected. The specified PCI board was not detected. Check that PCI board in securely installed into PCI slot. Also, run InstaCal to locate/set valid base address and configuration.</td>
</tr>
<tr>
<td>328</td>
<td>NO_PCI_BOARD</td>
<td>Specified PCI board not detected. The specified PCI board was not detected. Check that PCI board in securely installed into PCI slot. Also, run InstaCal to locate/set valid base address and configuration.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td>334</td>
<td>CANT_INSTALL_INT</td>
<td>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.</td>
</tr>
<tr>
<td>339</td>
<td>CANT_MAP_PCM_CIS</td>
<td>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.</td>
</tr>
<tr>
<td>344</td>
<td>NOMOREFILES</td>
<td>No more files in the directory. The end of the log file was reached before the file header was read.</td>
</tr>
<tr>
<td>345</td>
<td>BADFILENAME</td>
<td>No file exists for the specified file number. The specified binary file number does not exist.</td>
</tr>
<tr>
<td>347</td>
<td>LOSSOFDATA</td>
<td>The file may not contain all of the data from the logging session because the logging session was not terminated properly. The log file may be incomplete if the logging session is not properly terminated. Always end a logging session by pressing the data logging button until the LED turns off. Possible data loss may occur if the end of the log file is reached before the file header is read.</td>
</tr>
<tr>
<td>348</td>
<td>INVALIDBINARYFILE</td>
<td>The file is not a valid MCC binary file. The binary file was not logged from an MCC USB device with data logging capability, or the binary file was logged during a data logging session that was not properly terminated and is missing information.</td>
</tr>
<tr>
<td>349</td>
<td>INVALIDDELEMITER</td>
<td>Invalid delimiter specified for CSV file extension. When converting a binary log file to a comma-separated values text file (.CSV), the delimiter character must be set to a comma.</td>
</tr>
<tr>
<td>400–499</td>
<td>PCMCIA error</td>
<td>Card &amp; Socket Service error. Contact the manufacturer.</td>
</tr>
<tr>
<td>500–599</td>
<td>Internal DOS error</td>
<td>Contact the manufacturer.</td>
</tr>
<tr>
<td>600–699</td>
<td>Internal Windows error</td>
<td>See details below.</td>
</tr>
<tr>
<td>603</td>
<td>WIN_CANNOT_ENABLE_INT</td>
<td>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.</td>
</tr>
<tr>
<td>605</td>
<td>WIN_CANNOT_DISABLE_INT</td>
<td>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.</td>
</tr>
<tr>
<td>606</td>
<td>WIN_CANT_PAGE_LOCK_BUFFER</td>
<td>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.</td>
</tr>
<tr>
<td>630</td>
<td>NO_PCM_CARD</td>
<td>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 InstaCal to reset the configuration.</td>
</tr>
<tr>
<td>Error number</td>
<td>Error name</td>
<td>Error message</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>801</td>
<td>INVALIDGAINARRAYLENGTH</td>
<td>The number of elements in the gain array must equal the number of channels in the scan. This error is generated when WinBufToEngArray() is called with the number of elements in gainArray not equal to the number of channels specified. Make sure that the number of elements in the array is the same as the number of channels in the scan.</td>
</tr>
<tr>
<td>802</td>
<td>INVALIDDIMENSION0LENGTH</td>
<td>The length of dimension 0 in the data array must equal the number of channels in the scan. This error is generated when WinBufToEngArray() is called with the length of dimension 0 of EngUnits not equal to the number of channels specified. Make sure that the length of dimension 0 in the array is the same as the number of channels in the scan.</td>
</tr>
</tbody>
</table>

**INVALIDGAINARRAYLENGTH and INVALIDDIMENSION0LENGTH errors only occur in the .NET class library**

The Universal Library will not print or stop if these errors occur, regardless of the error handling configuration specified by the call to MccService.ErrHandling. These errors must be checked by examining the ErrorInfo object returned from MccBoard.WinBufToEngArray.