AzeoTech® DAQFactory®

DAQFactory User's Guide

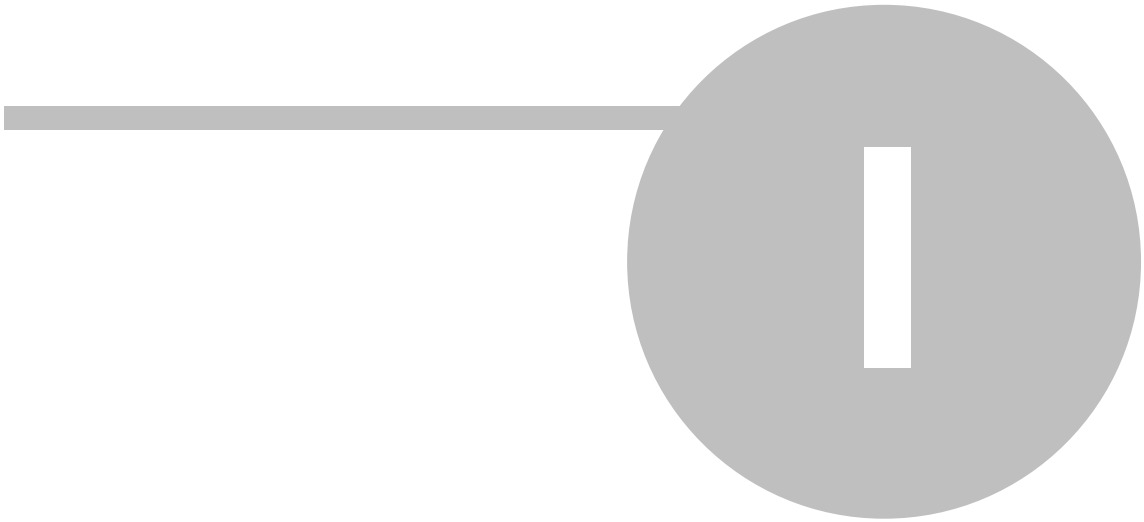
DAQFactory User's Guide

DAQFactory for Windows, Version 17.1, Friday, September 13, 2019:

Information in this document is subject to change without notice.

Table of Contents

1 Introducing DAQFactory



1 Introducing DAQFactory

1.1 End User License Agreement

AzeoTech, Inc., ("AzeoTech") licenses the accompanying software to you (referred to herein as "you" or the "end user") only upon the condition that you accept all of the terms contained within this Agreement relevant to the software. Please read the terms carefully before continuing the installation, as pressing the "Yes" button will indicate your assent to them. If you do not agree to these terms, please press the "No" button to exit the install, and return the full product with proof of purchase to AzeoTech within thirty (30) days of purchase.

I. LICENSE TERMS APPLICABLE TO ALL SOFTWARE

The following terms and conditions are applicable to any and all AzeoTech software products. The software which accompanies this Agreement is the property of AzeoTech and/or its licensors and is protected by U.S. Copyright law and state and federal trademark law, in addition to other intellectual property laws and treaties. This software is licensed to you, not sold. While AzeoTech continues to own the software, upon your acceptance of this Agreement you will have the following specifically defined rights and obligations arising from your license:

Once you have purchased a software license from AzeoTech, you may do the following:

(a) Use only one copy of the relevant software on a single computer;

(b) Make one copy of the software for archival purposes, or copy the software onto the hard disk of your computer and retain the original for archival purposes;

(c) Use the software on a network, provided that you have a licensed copy of the software for each computer that can access the software over that network;

(d) Upon written notice to AzeoTech and your receipt of AzeoTech's written approval, transfer the software on a permanent basis to another person or entity, provided that you retain no copies of the software, and that the transferee agrees to the terms of this Agreement.

(e) If you are an entity, you may designate one individual within your organization to have the right to use the software in the manner provided herein. The software is "in use" on a computer when it is loaded into temporary memory (RAM) or installed into permanent memory (hard disk, CD-ROM, or other storage device) of that computer.

The following are strictly prohibited by this Agreement:

(a) Copying the documentation which accompanies this software;

(b) The distribution of this software or copies of this software to third parties, except as provided in Section II(A) herein regarding the distribution of copies of Evaluation Software for evaluation purposes;

(c) Sublicensing, renting or leasing any portion of this software;

(d) Reverse engineering, decompiling, disassembling, modifying, or translating the software, attempting to discover the source code of the software, or creating derivative works of the software; and

(e) Using a previous version or copy of the software after you have received a disk replacement set or an upgraded version as a replacement of the prior version. Upon upgrading the software, all copies of the prior version must be immediately destroyed.

II. LICENSING TERMS RELEVANT TO SPECIFIC SOFTWARE PRODUCTS

A. DAQFactory and DAQFactory Runtime. The following provisions apply only to the use and license of all versions of DAQFactory and DAQFactory Runtime, but shall not apply to the use and license of DAQFactory Runtime in conjunction with DAQFactory-Developer, as provided in Section II(B) below, or to the use and license of DAQFactory Express, as provided in Section II(C) below.

1. AzeoTech provides a temporary version ("Evaluation Software") of DAQFactory and DAQFactory Runtime to all potential end users for a twenty-five (25) day evaluation period. At the end of the evaluation period, the Evaluation Software will convert into a DAQFactory Express license and its continued use without purchasing a license for a different version of DAQFactory is subject to Section II(C) below. Upon payment for a non-Express version of DAQFactory, AzeoTech provides the end user with a code that enables the appropriate features of the software.

2. You may distribute the Evaluation Software to third parties for evaluation purposes only. Such copies shall be

subject to the relevant terms and conditions of this Agreement in the same manner as if distributed directly by AzeoTech.

B. DAQFactory-Developer. The following provisions apply only to the use and license of DAQFactory-Developer and the use and license of DAQFactory Runtime in conjunction with DAQFactory-Developer and all documents and applications created with DAQFactory-Developer.

1. If you have purchased a license for DAQFactory-Developer, AzeoTech grants you permission to distribute your created documents and applications, together with DAQFactory Runtime, to third parties without a licensing fee. In exchange, you agree to be bound by all of the relevant terms and conditions set forth in this Agreement.

2. All third party users and recipients of documents or applications created with DAQFactory-Developer are bound by all of the terms and conditions of this agreement, and are strictly prohibited from distributing DAQFactory Runtime absent their own purchase of a license for DAQFactory-Developer. In addition, third party users and recipients are strictly prohibited from using DAQFactory Runtime to run applications other than those created with DAQFactory-Developer, except upon their purchase of a license for DAQFactory Runtime from AzeoTech.

3. Neither you nor third party users or recipients are permitted to create generic data acquisition applications using DAQFactory-Developer that would directly compete with DAQFactory or any other AzeoTech software product. This includes, but is not limited to, generic strip chart recorders, data loggers and PID loop controllers.

C. DAQFactory Express. The following provisions apply to the use and license of DAQFactory Express, a version of DAQFactory enabled once the evaluation period ends, or a separate software product available from AzeoTech and often provided to you in association with hardware from a hardware manufacturer.

1. DAQFactory Express is not "Evaluation Software" as defined in Section II(A) above, and accordingly, there is no 25 day time limit on its use and license. Despite the fact that you have not paid a licensing fee for this product, it is licensed software, subject to the relevant provisions of this Agreement. You are not permitted to distribute DAQFactory Express in any form to third parties, or upload this program onto the internet or a network which may be accessed by unlicensed third parties.

2. If you have received DAQFactory Express in conjunction with hardware from a hardware manufacturer, your license for DAQFactory Express is provided to you free of charge, by AzeoTech, through the hardware manufacturer. This license is strictly limited to your use of DAQFactory Express in conjunction with the accompanying hardware, and this program may not be used on any computer or device which does not contain the accompanying hardware. If you are a hardware manufacturer, you must enter into a separate agreement with AzeoTech to distribute DAQFactory Express licenses. DAQFactory Express cannot be distributed for free by simply including it with hardware.

D. DAQFactory Department Teaching License. The following provisions apply to the use of the DAQFactory Department Teaching License, a site license available for degree-granting educational institutions only. This license may only be used for instructional purposes and may not be used for research. This license applies to a single department within the institution at one physical location and does not apply across multiple campuses. This license may not be used for commercial or industrial purposes or used to create applications that will be used for commercial or industrial purposes. Subject to the terms of this license, the appropriate version of DAQFactory purchased (Base or Pro) may be install on an unlimited number of computers for an unlimited number of users within your department. This software is only licensed for use on institution owned computers, and home computers of faculty and staff for instructional purposes (preparing classwork for example). The software may not be installed on student-owned computers.

III. DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS

A. Support Services. AzeoTech may provide you with support services related to the software. Use of support services is governed by the AzeoTech policies and programs described in on-line documentation and other materials provided by AzeoTech. Any supplemental software code provided to you as part of the support services shall be considered part of the software and shall be subject to the relevant terms and conditions of this agreement. With respect to technical information provided to AzeoTech as a part of the support services, AzeoTech may use such information for its business purposes, including for product support and development.

B. Termination. Without prejudice to any other rights, AzeoTech may terminate this Agreement, in writing, upon your failure to comply with any of the terms and conditions of this Agreement, with such termination being effective upon your receipt of such notice. In such event, you shall immediately destroy all copies of the software and all of its component parts.

C. DISCLAIMER OF WARRANTIES. AZEOTECH EXPRESSLY DISCLAIMS ANY WARRANTY FOR THE SOFTWARE. THE SOFTWARE AND ANY RELATED DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OR CONDITION OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES AND CONDITIONS OF MERCHANTABILITY, FITNESS FOR A

PARTICULAR PURPOSE, AND NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE REMAINS WITH YOU.

AZEOTECH DOES NOT WARRANT THAT THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR FREE. AZEOTECH HAS TAKEN PRECAUTIONS TO GUARD AGAINST COMPUTER VIRUSES, BUT DOES NOT WARRANT THAT THE SOFTWARE PROVIDED WILL BE WITHOUT ANY COMPUTER VIRUSES. YOU ASSUME ALL RESPONSIBILITY FOR ACHIEVING YOUR INTENDED RESULTS, TAKING PROPER PRECAUTIONS TO GUARD AGAINST COMPUTER VIRUSES, AND FOR THE USE AND RESULTS OBTAINED FROM THE SOFTWARE.

D. No Liability for Damages. In no event shall AzeoTech or its suppliers be liable for any damages whatsoever, including, without limitation, any special, consequential, indirect or similar damages, including damages for loss of business profits, lost data arising out of the use or inability to use the software, business interruption, loss of business information, or any other pecuniary loss, arising from or out of the use of or inability to use the AzeoTech software, even if AzeoTech has been advised of the possibility of such damage. In any case, AzeoTech's entire liability under any provision of this Agreement shall be limited to the amount actually paid by you for the software. The disclaimers and limitations set forth herein will apply regardless of whether you accept the software. Because some states/jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

E. WARNING. AZEOTECH'S SOFTWARE IS NOT DESIGNED FOR COMPONENTS OR TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH ANY APPLICATION WHERE MALFUNCTION OF HARDWARE OR SOFTWARE COULD RESULT IN INJURY OR DEATH, OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY OR DEATH. THE SOFTWARE MUST NEVER BE USED FOR ANY PURPOSE THAT, IF THE SOFTWARE FAILED, COULD CAUSE INJURY OR DEATH TO ANY PERSON.

IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO THE FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER VIRUSES, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILER AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OF APPLICATIONS DESIGNER (THE ADVERSE FACTORS SUCH AS THE FOREGOING EXAMPLES ARE HEREAFTER TERMED "SYSTEM FAILURE"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY OR DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER IS CUSTOMIZED AND DIFFERS FROM AZEOTECH TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE AZEOTECH PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY AZEOTECH, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF AZEOTECH PRODUCTS WHENEVER AZEOTECH PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

IV. COPYRIGHT

All title and copyrights in and to AzeoTech software, including but not limited to any images, photographs, animations, video, audio, music, text, and "applets" incorporated into the software, any printed materials, and any copies of the software are owned by AzeoTech and its suppliers. The software is protected by U.S. Copyright laws and international treaty provisions. Therefore, you must treat the software like any other copyrighted material, subject to the terms of this agreement.

V. U.S. GOVERNMENT RESTRICTED RIGHTS

The software and documentation are provided with restricted rights. The software may constitute "commercial computer software" or "commercial computer software documentation" as those terms are used in 48 CFR 12.212. Unless otherwise agreed, the use, duplication or disclosure of such software and documentation by U.S. Government agencies is subject to the restrictions set forth in 48 CRR 52.227-14 (ALT III), 48 CFR 52.227-19 and the Defense Federal Acquisition Regulation Supplement (DFARS) 252.227.7013, as applicable, and the use, duplication or disclosure by the Department of Defense is subject to the restrictions set forth in 48 CFR 252.227-7013(c)(1)(ii) (Oct.1988). The manufacturer is AzeoTech, Inc., 2331 Morada Ln, Ashland, Oregon 97520.

VI. EXPORT RESTRICTIONS

You agree that you will not export or re-export the software, any part thereof, or any process or service that is the direct product of the software (the foregoing collectively referred to as the "Restricted Components"), to any country, person, entity, or end user subject to U.S. export restrictions. You specifically agree not to export or re-export any of the Restricted Components (i) to any country to which the U.S. has embargoed or restricted the export of goods or services, which currently include, but are not limited to Cuba, Iran, Iraq, Libya, North Korea, Sudan, and Syria, or to any national or any such country, wherever located, who intends to transmit or transport the products back to such country; (ii) to any end user who you know or have reason to know will utilize the Restricted Components in the design, development or production of nuclear, chemical or biological weapons; or (iii) to any end user who has been prohibited from participating in U.S. export transactions by any federal agency of the U.S. Government. You warrant and represent that neither the Bureau of Export Administration (BXA) nor any other U.S. Federal agency has suspended, revoked or denied your export privileges.

VII. GENERAL PROVISIONS

A. Entire Agreement. This Agreement contains the entire agreement of the parties, and may not be amended or modified except in writing signed by all of the parties. This Agreement supersedes any prior or contemporaneous understandings or agreements with respect to the subject matter hereof. Nothing in this Agreement shall be construed to limit any rights that AzeoTech may have under trade secret, U.S. and state trademark, copyright, patent, or other laws.

B. Non-Assignment. The end user may not assign any right or interest in this Agreement without the prior written consent of AzeoTech.

C. Divisibility. In the event that any provision of this Agreement shall be found to be invalid, unenforceable or prohibited by state or Federal law, the Agreement shall be considered divisible as to such part and all remaining provisions shall remain valid, enforceable and binding as though such part were not included in this Agreement.

D. Attorney Fees. In the event it becomes necessary for either party to file suit or instigate mediation or arbitration to enforce this Agreement or any provisions contained herein, and either party prevails in such action, then such prevailing party shall be entitled to recover, in addition to all other remedies or damages, reasonable attorney's fees and court costs incurred in the mediation, arbitration, at trial, and on appeal in such suit.

E. Choice of Law; Venue. This agreement shall be governed by the laws of the State of Oregon. Venue for all proceedings shall be in the Circuit or Federal Court for the State of Oregon, County of Jackson.

F. Contact Information. If you have any questions concerning this Agreement, or if you desire to contact AzeoTech for any reason, please email AzeoTech at support@azeotech.com.

1.2 Acknowledgements

We would like to thank the following companies for their excellent components and tools which are in use by DAQFactory® or were used to help create DAQFactory®.

- Dundas Software: For their Ultimate Grid MFC, Ultimate Toolbox, Ultimate Edit, and M++ math libraries.
- Chromium Embedded Framework and Google Chrome: for the embedded browser capabilities
- BCGSoft Ltd.: For their BCGControl application framework.
- Gigasoft, Inc.: For their ProEssentials graphing component.
- OptiCode - Dr. Martin Sander Software Development : For their curve fitting routine in the OptiVec math library.
- Software Toolbox, Inc: For their OPCData component and Symbol Factory image library.
- PJ Naughter: For his tray, single instance, web server, email, FTP and many other components.
- Concept Software, Inc: For their Protection Plus copy protection.
- eHelp Corporation: For their RoboHelp help authoring environment and tools.
- Red Hat, Inc.: For their foreign function interface code.
- Neil Hodgson: For the scintilla code editor

Here are the copyright notices for some of the above products.

- This software contains material that is © 1994-2000 DUNDAS SOFTWARE LTD., all rights reserved.
- Copyright (c) 2008-2013 Marshall A. Greenblatt. Portions Copyright (c) 2006-2009 Google Inc. All rights reserved.
- Copyright © BCGSoft Ltd. 1998-2001. All rights reserved
- Copyright (C) 2001 Gigasoft, Inc. All rights reserved
- Copyright © 1998-2001 OptiCode - Dr. Martin Sander Software Development

- Copyright Software Toolbox, Inc., 1996-2000, All Rights Reserved Worldwide
- Copyright (c) 1996 - 2001 by PJ Naughter
- Copyright © 1997-2000 Concept Software, Inc.
- Copyright (c) 1996-2003 Red Hat, Inc.
- Copyright 1998-2003 by Neil Hodgson <neilh@scintilla.org>

And do not forget that DAQFactory®, which includes DAQFactory® and associated files (including this one) are Copyright © 2001-2019 AzeoTech®, Inc. All rights reserved worldwide.

AzeoTech® and DAQFactory® are a registered trademark of AzeoTech, Inc.

The following notice is required for the Red Hat foreign function interface license:

libffi 2.00-beta - Copyright (c) 1996-2003 Red Hat, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software''), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL CYGNUS SOLUTIONS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The following notice is required for the Scintilla editor:

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

The following notice is required for the Chromium Embedded Framework browser:

Copyright (c) 2008-2013 Marshall A. Greenblatt. Portions Copyright (c) 2006-2009 Google Inc. All rights reserved.Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
* Neither the name of Google Inc. nor the name Chromium Embedded Framework nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The following notice is required for OpenSSL:

==========
Copyright (c) 1998-2007 The OpenSSL Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgment:

"This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)"

4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org.

5. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project.

6. Redistributions of any form whatsoever must retain the following acknowledgment:

"This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES

1.3 Welcome to DAQFactory

DAQFactory is the complete data acquisition, process control, and data analysis solution. With DAQFactory you can take data at precise time intervals, store your data to disk, broadcast it over a network, display it on your own custom designed screen, automate your process, and analyze your data.

DAQFactory was designed to be easy to use. Screens are created by simply dropping components onto the screen and using simple windows or tables to change their parameters. There is no wiring. Data analysis is as simple as typing in your formulas, or using our handy windows. All data is stored with its time stamp, so keeping track of time is a snap too.

1.4 System Requirements

First, it is important that you turn Windows Automatic Update off when using DAQFactory. If Automatic Update runs while DAQFactory is running it could cause problems as Automatic Update changes system files that DAQFactory uses.

For DAQFactory:

• Any computer capable of running Windows XP SP 3 or newer. We suggest at least a Pentium 90, and preferably much more. The program is designed to be responsive even under processor strain, however, the refresh rate of graphs and images will be greatly reduced on slower processors.

• 100 Meg of free hard disk space.

• Enough memory to run the operating system plus a minimum of 100 meg on top of that. Generally, the more you have, the more you will be able to take advantage of DAQFactory's capabilities. The exact memory requirements

depend on many factors, but in general, DAQFactory is not usually a memory hog.

• Video capable of 800x600 resolution. 1024 x 768 is strongly suggested. If DAQFactory is only going to be used as an HMI, it is possible to setup the DAQFactory screens on one computer and transfer the file to another computer that may not be able to achieve 800 x 600. In this case, DAQFactory will operate at almost any resolution. We suggest DAQFactory Runtime for this type of application.

• Windows XP SP 3 or newer. You cannot run the program on Windows 95, 98, 2000, Me, NT 4.00, NT 3.51, CE, or any other lesser version of Windows.

• Internet Explorer version 6 or greater must be installed for the help system.

For DAQFactory Runtime:

• The runtime version of DAQFactory is a part of the regular DAQFactory installation and has the same requirements. That said, you can skip installation of the image library and reduce the disk space requirements. It will run at any screen resolution, provided the DAQFactory document is designed for the new screen size.

1.5 DAQFactory Performance

Often DAQFactory users will use the Windows performance monitor to see how much CPU power DAQFactory is using. The performance monitor is a bit deceiving when using DAQFactory. DAQFactory graphs and symbols and the panel component are drawn in what's called the idle thread priority of Windows. This means that they are generated only when Windows is done doing everything else. This enables DAQFactory to remain responsive even if you try and draw a complicated graph that takes a few seconds to draw. This is also why when you change pages, everything but the graphs appear, and then any graphs appear.

To truly determine how much processor time is being used, go to the channel list or other configuration view. This will tell you better about how much processor power is being used to perform the acquisition and sequences.

Its also important to remember that as long as the CPU usage isn't hitting 100% you are fine. Even if it touches 100% occasionally, data acquisition and sequences have priority over the user interface and many other things in Windows, and so will run fine. With complex graphs on a page with a fast refresh rate you could even see it at 100% constantly. Also, the CPU will often hit 100% when you are manipulating components, turning a knob say, or when dragging windows around, or other such things. With the exception of the fastest computers, most systems will do this even without DAQFactory. To see this, just open the task manager and start dragging it around the screen.

The bottom line is that unless the user interface is sluggish, or your loops are not running in the times specified, or you are getting Timing Lag errors, you should not worry. But even if one of these things happen, the problem may not be with CPU power. For example Timing Lag errors are most common when you try and overdrive your DAQ hardware. For example, trying to read the LabJack U12 at more than 50hz. The most common reason for a sluggish user interface is having a looping sequence without a reasonable delay() and the sequence loop check turned off, or by using wait() improperly.

If you are really concerned, here are a few other things you can do to help performance of your application:

1) Reduce the resolution and/or color depth of the screen. This will decrease the amount of time required to draw the screen. It will also reduce DAQFactory's memory usage.

2) Reduce the refresh rate of the screen (right click on the page name and select Properties...). The default is 2 hz. There is no reason to have a refresh rate that is faster than your fastest data rate.

3) Keep your history lengths on your channels at smaller values. This is usually only an issue if you have histories above 20,000 or so on slow computers. On newer computers, history lengths in above 100,000 are easily possible. If you need larger histories or persistence lengths, make sure and subset all your data before manipulating it. For example, if you are doing a graph, don't just put the channel name, but rather subset to the x axis scaling.

4) Check your sequences for reasonable loop times, and whenever possible, use events instead of sequences with loops. For example when monitoring an input for a particular value, you should use that channel's event instead of creating a sequence loop.

1.6 Licensing DAQFactory

When you purchase DAQFactory, either before or after the demo expiration, you will need to get unlock codes to

eliminate the annoying dialogs that appears when you start DAQFactory and allow DAQFactory to be used past the expiration date. When you purchase DAQFactory you will be provided with the 15 or 16 letter keys to get the unlock code needed. There are two ways to use these keys depending on whether the computer running DAQFactory is connected to the Internet:

If the computer with DAQFactory is connected to the Internet:

1. At the Product Evaluation window, click the **Online Unlock** button.

   Make sure you are connected to the Internet.

2. When prompted, enter your 15 or 16 letter key (including the dashes).

   For example: **ABCDE-FGHIJ-KLMNO**

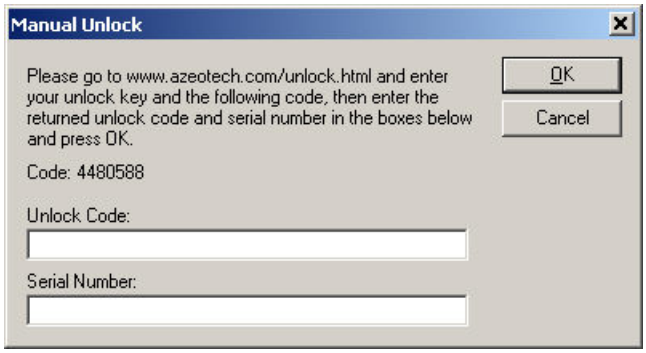
3. When you press **OK**, DAQFactory will connect to its unlock site, verify the code you entered, then if correct, unlock the product.

If the computer with DAQFactory is not connected to the Internet:

1. At the **Product Evaluation** window, click the **Manual Unlock** button.
2. A new dialog will appear requesting the first two letters of your key. Enter them and click **OK**.

   This is used to verify that you are using the correct code with the correct program.

3. Another dialog will appear.



4. Go to www.azeotech.com/unlock.html and enter in the code displayed in the window above (yours will be different than **4480588**) and your 15 letter key. **Make sure you enter the code correctly into the web site.** When you click **Get Unlock Code** on the web site, a numeric unlock code and your serial number will be provided. Enter these two numbers into the edit boxes in the **Manual Unlock** window then click **OK**.

5. Your product should now be unlocked.

If you have no Internet connection available, contact your vendor and they will be able to provide you with the manual unlock codes.

The unlock codes are specific to the machine you installed DAQFactory on. There is a different unlock code for each version of DAQFactory.

1.7 Moving a license

As just described, DAQFactory uses a 15 letter soft key to convert a trial installation to a licensed installation. At some point, however, you may need to move this license to another computer. As it states in the End User License Agreement, you have to have a license for each computer running DAQFactory simultaneously. To move a license and ensure that you are not violating the EULA:

1) On the old computer, within DAQFactory, go to Tools - Remove Development License... This will remove the license, reverting the installation to a trial.

2) On the new computer, use the unlock key provided to you to license the computer as described in the last section.

A few points:

• Make sure and remove the license before uninstalling DAQFactory. Uninstalling alone does not remove the license.

• This method does not work for runtime licenses, which cannot be moved.

• Removing the development license in step one above does not affect any existing runtime license. This allows you to install a development license on a customer computer along with the runtime and once the application is developed, you can remove the development license. If you expect to do this more than once you should consider the hardware key option.

• If you have a system crash or hard drive reformat you can skip the first step as the crash/reformat will erase the license.

• The unlock keys have a use limit, after which you will have to contact us for a new soft key. If you expect to move the license multiple times we recommend the hardware key option which makes this very easy and is described in the next section. We reserve the right to insist on the hardware key, at your cost, if we feel your soft key is being abused.

1.8 Using the hardware key option

As an alternative to using the software license mechanism, you can purchase a thumbnail sized key that connects to your USB port and contains your license. This key, and therefore your DAQFactory license, can easily be transferred from one computer to another. This is especially useful when you have runtime installations as you can simply insert the key to convert the runtime to a development version without stopping the application. You can then make any changes to your document. When finished, you can simply pull the hardware key and the system will revert to runtime mode, assuming of course a runtime license exists.

To use the hardware key, you first much purchase a key from AzeoTech® or your DAQFactory® reseller. These keys are not included with DAQFactory. You also must have a fully licensed and unlocked version of DAQFactory.

Note: DAQFactory Developer includes a hardware key. The key you receive will already have a license on it and is ready to use. You only need to perform the first step of this procedure to install the hardware key driver.

To move your license into the key:

1. Click on Start - Programs - DAQFactory - Tools - Install Hardware Key Driver - Silver Keys.

   This will install the USB driver for the hardware key. If you are using Windows XP or Vista, you may see a window saying the driver is not Logo tested. Click on **Continue Anyway**. The driver has been fully tested for XP and Vista.

2. Connect your USB key to your computer. Windows will finish installing the driver.

3. Start DAQFactory.

4. Within DAQFactory, select **Tools-Transfer License** to USB hardware key to copy your current license into the hardware key.

Once complete, you can move the key from computer to computer as needed. You must attach the key to your computer before starting DAQFactory, or the program will load in trial mode. Also, if you remove the key, DAQFactory will prompt you to reinsert or quit.

Note: you cannot transfer the license out of the key. In other words, once you transfer your license into the hardware key, your license will remain there and you will have to use the key to use DAQFactory.

Upgrading your license when using the hardware key:

To upgrade your DAQFactory license version, (for example from Base to Pro) simply insert your hardware key before starting DAQFactory, then proceed with the upgrade as you would with a regular software license. The license on the hardware key will upgrade automatically.

I have received the hardware key and can't get it to work:

If you have followed the above directions, but still can't get Windows to recognize the hardware key, then most likely you inserted the hardware key before performing the first 2 steps above and have marked the device as disabled. You will have to go into the Windows device manager and reenable the device.

1.9 Upgrading to a different version of DAQFactory

If you decide that you would like to take advantage of the features of a different version of DAQFactory, you can purchase an upgrade license. The cost of this license is typically slightly more than the difference between the normal cost of license types. Once purchased, you will receive an unlock key. This key is 16 characters long, and usually starts with either Y or Z. To access the unlock windows, start DAQFactory and then select **Tools-Enter Unlock Key** and select the type of unlock you would like to perform. From here the unlock method is identical to the method used when you first unlocked DAQFactory.

1.10 For users of earlier versions of DAQFactory

• As with most releases, a number of new functions have been added to the global namespace and may conflict if you have a sequence or variable with the same name. These include "using", "include", "pop3", "ftp", "clearglobals", "startthread", "getthreadstatus", "stopthread", and others.

• The strtodouble() function was updated to return NaN() if the string started with an invalid character (i.e. strtodouble("ab")) instead of 0. As always, if the string starts with a number but then contains an invalid character, the string is still properly converted up to that point: strtodouble("123ab") returns 123.

• On that same note, the Mean / Sum / Max / Min functions have been updated to handle NaNs properly which may cause existing calcs to change if you are using these functions and NaNs in your data.

• Another bug fix that may cause calcs to change is when subsetting by time. Older releases would incorrectly include one point past the end time. New releases properly subset the date range specified.

For users of DAQFactory release 5.15 and earlier

• Starting with Release 5.30, DAQFactory allows the naming of variables with the Var. or Private. prefix. Your previous documents will still work with the older Var. and Private. notation, but these variables will also be accessible without this notation. So, if you had a variable "Var.MyVar" you could also access this variable simply with "MyVar". This will not cause any problems with older documents unless you happened to name a variable the same as a channel. Since variable names have priority over channel names, referencing a channel with the same name as a variable will result in the contents of the variable, not the channel. So if you had a channel called "MyVar" and you do "MyVar[0]" in an expression, you will get the contents of Var.MyVar instead of the channel MyVar. In these cases, you'll need to rename either your variable or your channel.

• One bug fix in 5.30 may cause problems with earlier documents if you have a workaround in place. In previous versions, when doing the AddValue() function on a channel with a value that had more than one row, the rows would be flipped before being added, so:

after **`MyChannel.AddValue(Var.x)`**, Var.x[0] would not equal MyChannel[0]. This has been fixed so that the values are added in the right order.

• The format() function now accepts \ notation for carriage return, line feed and tab. This means that if you have a regular \ in a format() function, you'll need to add a second backslash: \\. This does not affect regular string assignment, so **`x = "c:\myfile.csv"`** is still valid, but **`x = Format("c:\myfile%d.csv",y)`** needs to be changed to **`x = Format("c:\\myfile%d.csv",y)`**

The Runtime version of DAQFactory is now a part of the development version and no longer a separate executable. This has a lot of advantages (see the section on Runtime), but will require some changes. First, you'll need to replace your DAQFactory Runtime installation with a regular DAQFactory installation (licensing will remain the same, so just install into the same directory and change your shortcuts). The Runtime also will no longer automatically load Runtime.ctl, so you will have to specify the file name in a Windows shortcut.

For users of DAQFactory release 4 and earlier

Release 5 of DAQFactory is a major new release with many changes. If you are a user of DAQFactory Release 4.14 or earlier, there are a few things that may cause your old documents not to function the same way:

• **DAQFactory.exe:** Since DAQFactory Acquire is now part of DAQFactory Control, we have simply named the program DAQFactory and changed the executable from Control.exe to DAQFactory.exe. Please make sure your shortcuts point to the correct executable and that you are running release 5. You can check your release by going to **Help - About DAQFactory**. The runtime has also been renamed DAQFactoryRuntime.exe.

• **Logging:** Data logging in DAQFactory release 5 has been completely revamped. User feedback on the difficulties in importing logged data into other programs was the impetus for these changes. All old forms of data logging including binary forms and local condition logging have been removed. Now all data logging is performed using logging sets. Please see the logging chapter for a complete description.

• **Sequences:** Sequences have been made into a complete scripting language, giving them much more power. Old sequences will load correctly and be correctly translated, however error handling in the new sequences is different. In old versions of DAQFactory, if an error occurred in a step, the step was simply skipped. This often made it difficult to figure out why a sequence was not performing the way it should, since errors were not apparent. The new sequences use a C++ type error handling mechanism. Errors must be caught and processed or the sequence will stop and display an alert. Because of this, all old sequences will load with the command **`ignore ("all")`** added. This will cause all errors to be ignored similar to the older releases.

• **Symbols (formally images):** To better handle 3rd party symbols and properly scale the symbol factory symbols, symbol scaling has been fixed to display the symbol within the component's rectangle. This will cause all of your current symbols to be scaled slightly differently. This can be fixed easily by resizing your symbols.

• **Point():** The Point() function now returns a 0 index array instead of 1 indexed array.

• **Relative time subsetting:** is no longer supported. Replace with SysTime() - x notation: **`MyChannel [SysTime()-5,SysTime()-2]`**

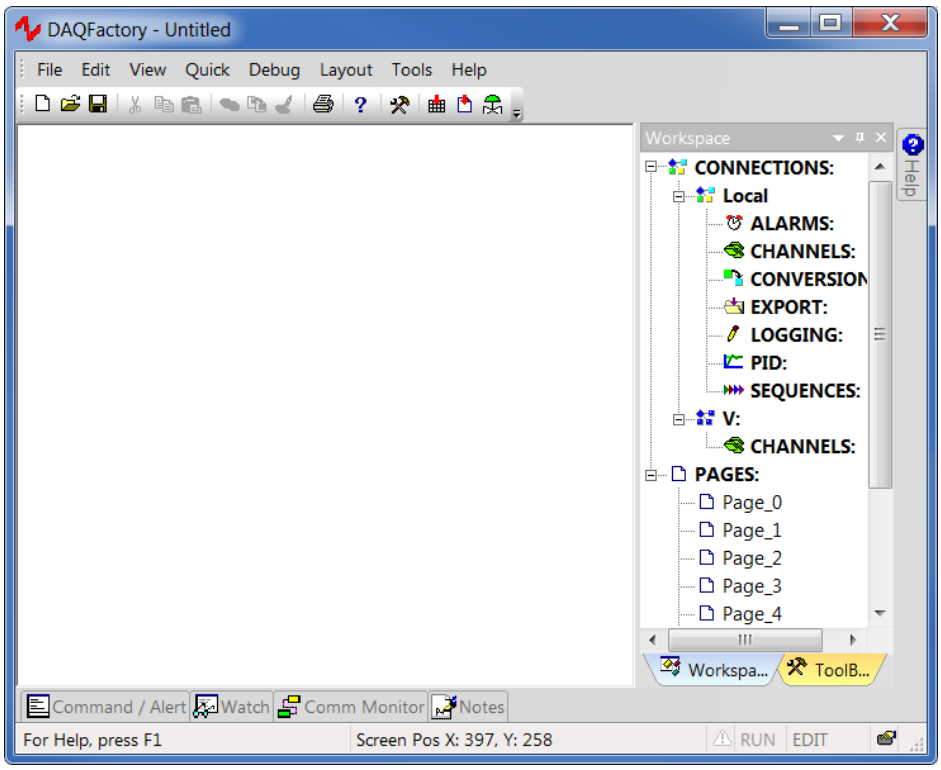
• **Case sensitivity:** DAQFactory is no longer case sensitive except when comparing string values. This will only affect older documents if you have two objects with the same name but spelled with different case: for example, MyChannel and mychannel. If so, rename one of the objects.

Also you should reset your menus and keyboard shortcuts. To do this, select **Tools-Customize...** from the main menu. Go to the **Toolbars**, **Keyboard** and **Menu** pages and click **Reset All** on each of them (**Reset** on the menu page).

Note: Be sure to keep a backup of your original document before upgrading to the new release.

1.11 DAQFactory's User Interface

When you start up DAQFactory for the first time, you will see a typical Window's window:



Title Bar: The area at the top of the window, common to most all Windows applications. This does nothing out of the ordinary, except that it will flash to get your attention if an alert event occurs. You can stop this behavior by going to File - Preferences.

Main menu: The menu area just below the title bar, also common to many Windows applications. Most of the functions of DAQFactory are available through these menus. Many of the options may appear grayed out until certain events have occurred.

Toolbars: These offer quick access to common features. One toolbars are initially displayed for the most common items. There are several other toolbars available as well. Click on **View** in the main menu to activate them. You can even customize the toolbars if desired. Go to **Tools-Customize...** to do so.

Pages: This is the main area of the screen, which is completely white when you start DAQFactory for the first time. A page is where you can drop in various components to display your data in numerous different ways, including schematics of your process, or complicated graphs. DAQFactory can have an infinite number of pages for displaying different parts of your data, or displaying your data in different ways. Switching between pages can be as simple as pressing a key. You can even overlay multiple pages to keep important data always on the screen, or popup pages in separate windows.

Status bar: The status bar is the area along the bottom of the window and contains quick help information on menu items and toolbar buttons. As you move the mouse over these items, a slightly longer explanation of the meaning of the menu item or button will be displayed. To the right side of the status bar are six other panes:

The first one displays the position on the screen in pixels. When your mouse cursor is over a graph, it will display the coordinates of your mouse cursor within the graph in graph coordinates instead.

Next is a panel that shows the amount of time it took to render the current page(s). This is useful for determining user interface performance in advanced applications.

Next is an alert panel. This, like the title bar, will flash when an alert event occurs.

Next there is the safe mode panel. This illuminates "SAFE" when you are running in safe mode, or displays "RUN" in gray when the system is running. Safe mode stops all data acquisition, sequences and prevents access to many

potentially damaging script functions. It is typically used when editing a document away from the actual data acquisition hardware. We use it all the time when customers email us their documents to review.

Next is the current edit mode. This shows "EDIT" when in edit mode, and "OPERATE" when in operate mode. When in edit mode, clicking or dragging on components will select or move the components. In operate mode, clicking or dragging will actually operate the screen control (clicking a button for example). You can hold the Ctrl key down to temporary switch modes, though the status bar will not change.

Finally is a button that lets you quickly switch between edit and operate modes. You can also switch modes from the main menu under Edit - Edit Mode.

Note: DAQFactory defaults to Operate mode, and we suggest you stay in Operate and use the Ctrl key to manipulate screen components. It takes a little getting used to but once you get it, it is much faster than switching between Operate and Edit modes.

The rest of the window contains what Windows calls Docking Control Bars. These are windows that can be "docked" to the various sides of the main DAQFactory window, or dragged elsewhere on the screen in detached mode. DAQFactory control bars also can be auto-hidden, and most are in this state by default. An auto-hidden control bar displays a tab most of the time, but if you click on the tab, or even just place you mouse cursor on the tab, the window will appear. You can then work with the window. Once you click outside the window, it will auto-hide back to the tab. Any of the control bars can be put in auto-hide mode, or switched out of it. To do so, click on the thumbtack at the top right corner of the control bar window.

Workspace: This is where most of the background parameters, such as connections, channels, conversions and sequences are accessed. These parameters are displayed in the workspace in tree form to quickly identify which channels correspond to which connections, etc. A list of the display pages are also displayed on the tree. This is a floating window that by default is docked to the right side of the screen, but can be dragged anywhere on the screen or docked to other sides. If you close this window, you can reopen it from **View** in the main menu.

Toolbox: This window contains all the components available for use on Pages. Simply drag the desired control from the toolbox to the Page. You can also add components by right clicking on the Page and selecting the desired component. This window is usually tabbed with the Workspace. If you have a large screen, you might consider docking it separately to the right so that you can see both the Workspace and Toolbox at the same time.

Help: This is another floating window that contains help information. By default, this bar is auto-hidden along the right side. Simply click on the tab to make it appear. If follow-me help is enabled, help will also appear under most popup windows. Follow-me help is enabled by default and can be turned off and on from **View-Help-Follow Me Help** in the main menu.

Command / Alert: DAQFactory contains a command line interface that allows you to execute script functions, or view your data. Use of this interface is optional, but often useful for those users who prefer this type of interface. In addition to the ability to enter commands and see the result, this window displays error messages. All messages remain in the window. To clear the alert window, right click on the window and select Clear Window. You can also copy out of the results part of the window by selecting the text, right clicking and selecting Copy.

Watch: This is another floating window that provides a table for displaying current values of any sort of expression. This is typically used to debug sequences. The table contains 100 lines. Enter an expression in the left **Watch** column and the result of the expression is displayed in the right **Value** column. This value is updated twice a second.

Comm Monitor: The Comm Monitor window allows you to easily monitor serial and Ethernet communications from a docking window. The window works almost identical to the monitor window described in the chapter on serial and Ethernet communications, except that to select the communications port to monitor, pause the monitor and adjust the other settings, you should right click for a popup menu. At the top of this window is a single line edit box where you can type text to be output to your communications port. Below this is a larger area showing the communications. Please see the section on serial and Ethernet communications for more details.

You can also copy out of the monitor part of the window by selecting the text, right clicking and selecting Copy.

Notes: This window displays quick notes. Quick notes provide a virtual lab notebook. You can enter any sort of information about your experiment or process and the note will be time stamped with the current computer time. The notes can be saved to disk as well with your data.

There are several other floating windows that are not initially displayed. They can be displayed by selecting the appropriate window from under **View** in the main menu.

On Screen Keyboard: If you are running on a touch screen or a computer without a keyboard, there is an on screen keyboard that can be displayed. This also enables the on screen keyboard in other areas of DAQFactory. The on screen keyboard cannot be used for creating your application as it does not function in the properties windows of page components. This window is normal not displayed. To display it, go to **View** in the main menu.

1.12 DAQFactory Objects

DAQFactory is made up of different objects that can be set up in many different ways to create your custom data acquisition solution:

Channels:

A channel typically is a single I/O location, be it analog to digital, digital to analog, digital out, digital in, counters, a spectrum, an image, or any other type of data. For those in the automation world, a channel is equivalent to a tag. There can also be channels that contain calculated values such as results from analysis functions, or even calculated formulas. These are discussed a bit more in virtual channels. To make life easier, the details of which I/O channel number and device correspond to what channel only need by entered once. Each channel is given a name, and after defining the I/O associated with that name, only this name is required throughout the rest of the program. There is no need to remember that ambient_temperature channel is attached to Acme I/O board, device #3, channel #2!

Channels have a history. This is a list of values with time tagged to each value. The most recent values are always the first values in the history progressing back in time.

Virtual Channels:

Virtual channels provide a location for storing values that do not have an I/O associated with them. This can be results from analysis calculations, captured values from graphs, calculated expressions, or values set from a sequence. They are called virtual channels because they can be used anywhere a channel is used within the DAQFactory program and contain a history like a channel. Virtual channel's usefulness has decreased as DAQFactory has moved forward. Typically you will want to use regular channels and the AddValue() function to store historical values, or variables to store scalar values.

Variables:

Variables are simply locations in memory for storing values. They are not associated with any I/O and do not have a history, though they can be an array of values. Variables are typically used in sequences, but can also be used as state flags or anything else you may come up with.

Conversions:

Most I/O devices output their values in rather abstract units, such as a number between 0 and 65535, 0 to 10 volts, or 4 to 20mA. Conversions provide a way to convert these arbitrary units into something more tangible and appropriate. For example, you may have a pressure transducer that has a range of 0 to 500 psi which corresponds to 0 to 10V output. Your A/D channel outputs 0 to 4096 corresponding to 0 to 10V. You could create a conversion to convert that 0 to 4096 value into a 0 to 500 psi value. The conversion would look something like this:

```
Value/4096*500
```

This is an example of a simple conversion. You can make much more complicated conversions to take into account calibration curves etc. Because you will often have multiple Channels reading similar devices (multiple identical pressure transducers in this example), the conversions are kept in a list by name, and when you need to apply a conversion to a channel, you simply select the name instead of having to retype the formula multiple times.

Conversions are especially useful on output channels. They allow you to specify your output value in useful units rather than the units required by the output channel. Here the conversion expression is actually reversed. The expression should convert useful units to I/O units.

Expressions:

Conversions are not the only place you can enter in calculations. In fact, conversions are typically only used to get things into proper units. While not really an object, expressions are calculations that you can use in various places throughout DAQFactory. In fact, the formula you enter in a conversion is actually an expression. Expressions can

be as simple as a constant or as complex as you need, acting on single data points or arrays of points.

When entering expressions, DAQFactory provides syntax coloring and an intelligent drop down box that displays channels, formulas and other things you may want use in your expression so you don't have to remember everything. Any edit box within DAQFactory that displays in color can take an expression. Expressions are used to display calculated values, to graph calculations, to set ranges, to control data logging and in many other locations.

Sequences:

Sequences provide a powerful scripting language to automate your project. Sequences have many uses including performing automatic calibrations, monitoring your system for certain conditions and performing actions based on those conditions, or even replacing programmable logic controllers or other hardware devices with software control. Sequences are easy to setup and maintain. Multiple sequences can be run at multiple times, and simultaneously. Sequences can also be used as user functions in expressions. Sequences can even be setup to start automatically when the DAQFactory starts, creating a completely automated system!

PID Loops:

PID is an effective algorithm for controlling closed loop systems. The algorithm attempts to keep a "Process Variable" (PV), such as temperature, at a particular "Set Point" (SP) by controlling an "Output Variable" (OV). The algorithm uses three parameters: proportion (P), integral (I), and derivitive (D). DAQFactory includes an autotuning mechanism to help you determine the best P, I and D parameters for your system. You can create as many PID loops as your computer will allow, all of which can run simultaneously.

Logging Sets:

Logging sets provide the mechanism for logging your data to disk. You can create multiple logging sets to log all or part of your data in several different formats with different options. Logging sets can be started and stopped at any time and can be run simultaneously. Currently, DAQFactory supports logging to delimited ASCII, ODBC database, and 3 binary modes. Delimited ASCII mode is easily read by most other programs such as Excel and the recommended format unless you have a back-end SQL database.

Export Sets:

Export sets are similar to logging sets but do not run continuously. While logging sets log channels directly (or calculated values through a manual mechanism), export sets log the results of expressions. The options for export sets are very similar to logging sets. Logging sets are typically used for continuous logging. Export sets are used when you want to conditionally log one line at a time, or you want to log data that has already been acquired.

Alarms:

Alarms provide a way for you to detect out of boundary conditions in your system even if your system returns to normal. For example, if you want to know if your system gets above 100°C, you can use an alarm to monitor the temperature and it will trigger if the system goes above 100°C even if it then immediately returns under 100°C. Alarms display until manually acknowledged and can be logged in an ASCII file or ODBC database. You can have multiple alarms watching for any number of events in your system.

Pages:

Within the DAQFactory program you can have multiple screens containing graphs, images, values, and other screen components that provide the user interface for your device. Each screen is in a basic way considered a page. Pages, however are more like overlays than a simple screen. You can easily display multiple pages overlaid on top of each other on a single screen. This is most useful when you have certain data you always want to be visible. You can place this data on a page and have that page overlaid on top of all the other pages. Pages can also be made to popup in a separate window. The window can either be modal, meaning the user cannot access any other window until the popup is closed, or modeless, a floating window that allows access to the rest of DAQFactory. Modeless popup pages are a useful alternative to overlaid pages.

One very powerful feature of DAQFactory is the ability to design your Pages while your application is running. No quitting your experiment or process halfway though when you realize that you need just one more graph. You can add, remove, or change any component at any time!

Components:

Components are the windows to your data. There are 44 components that allow you to create a completely custom user interface for your application. Components can be placed anywhere on pages enabling you to create any screen design you would like, from a simple summary of your data, to a complete virtual instrument or factory. All the Components have many options that control how the Component displays itself and what action is performed when you click on the Component.

Connections:

A Connection identifies where data is coming from. Data can come from various places:

- A local connection, namely from within DAQFactory itself.
- The virtual channel list, or V Connection containing calculated data, results from calculations, and captured data.

A single copy of DAQFactory can have multiple connections to any of the above sources, enabling the user to monitor multiple locations and data streams from a single spot. Remote connections are established in DAQFactory and consist of a name, a TCP/IP address or URL of the copy of DAQFactory it is connecting to, and some other miscellaneous information that is discussed later. With the exception of the V connection, each connection has its own alarm, channel, conversion, export, logging, PID and sequence list.

2 Guided Tour

2 Guided Tour

2.1 Tour Overview

In this tour we will show you how to use DAQFactory to perform data acquisition, display your data, both locally and on the web, and logging, and give you the base to proceed comfortably through the rest of the manual or explore DAQFactory on your own. We recommend performing all the steps on your computer as you read through this tour. To use this tour you do not have to have any data acquisition hardware. This tour uses the test device which generates sine waves.

2.2 Tour Assumptions

This guided tour, and the rest of the DAQFactory manual assumes you are a reasonably experienced Windows user. It also assumes that you have installed DAQFactory on your computer and are ready to use it. Installation instructions are provided by the installer.

2.3 Tour Terminology

We recommend you read the section on DAQFactory objects to get an overview of the terms used in this tour as well as the rest of this manual.

"Click" means left click.

"Ctrl-Click" means left click while holding down the Ctrl key.

"Right-Click" means right click.

"Double-Click" means two left clicks in rapid succession.

2.4 Starting DAQFactory

1. Click on the **Start** button, select **Programs**,**DAQFactory** then **DAQFactory**.

   A splash screen will display and then you will be left with a blank DAQFactory document. The DAQFactory screen is made up of several parts described under DAQFactory's User Interface. For this tour, we will mostly be using the workspace, which is the area along the right of the screen with an Explorer like tree, and pages, which is currently the blank area that occupies most of the screen.

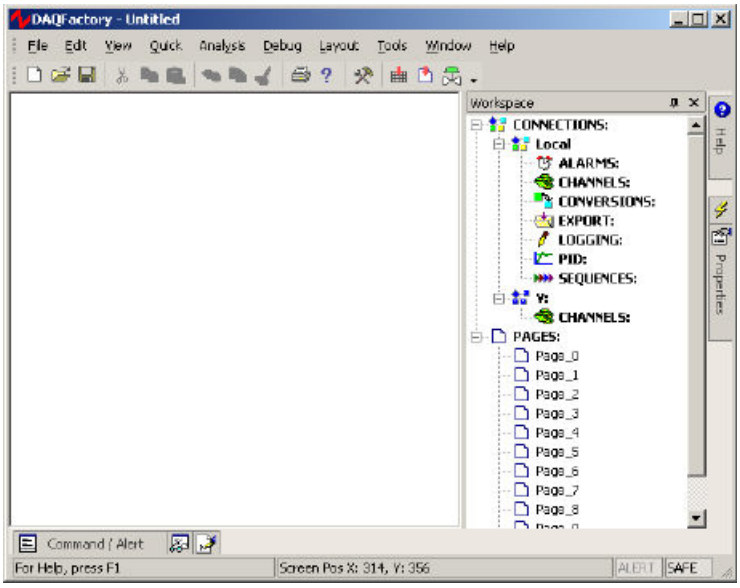
Note: If you are using a software firewall, such as the one included with Windows XP SP2, you may get a message saying that DAQFactory is trying to access the internet. This is simply DAQFactory's networking server being enabled. You can choose to allow or disallow this.

2.5 Creating Channels

In this part, we will add a channel to the document and start acquiring data.

1. In the workspace (the tree area along the right of the screen), click on **CHANNELS:** under **Local**.

   The workspace provides an organized presentation of the different configuration screens for setting up your DAQFactory document. Clicking on almost every item in the tree will bring up a new view related to the item selected. Clicking on **CHANNELS:** displays the channel table where you can manipulate all the channels in your document. Make sure and click the **CHANNELS:** under **Local** and not the one under **V:**



2. In the **Channel Table View** that appears, click on the **Add** button at the top.

   This creates a new row where we will enter the information for our first channel.

3. In the new row of the table, enter **Pressure** in the **Channel Name** column.

   All channels must have a name. You can assign any name you want to your channels as long as they are unique, start with a letter and only contain letters, numbers or the underscore "_".

4. In the second column, **Device Type**, select **Test** from the drop down list.

   The device type determines which type of device you wish to communicate with. This can be a specific manufacturer, a specific device, or a generic communications method such as OPC or serial.

5. In the third column, **D #**, leave the cell set at 0.

   The device # is only used by some devices and has different meanings depending on the device.

6. In the fourth column, **I/O Type**, select **A to D.**

   Most devices have several different types of data coming in or going out. The I/O type determines which type of data you desire. For example, many DAQ boards contain both A to D and D to A channels. The I/O Type determines which of these you wish to communicate with for this channel. I/O Types are not fixed types and different devices will have different choices available.

7. In the fifth column, **Chn #**, enter **1.**

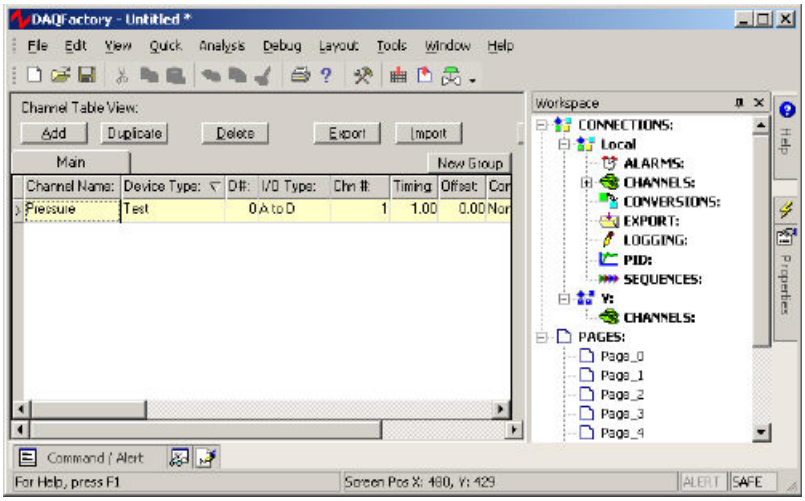
   Most devices have several I/O points per I/O type. The channel number determines which I/O point this channel refers to.

8. Leave the rest of the columns in their default setting.

   One important item to mention is the sixth column, **Timing**. For input channels this determines how often the input is read from the device. The interval provided here is in seconds. For now, we are going to leave it set at 1, or one new reading per second. In general you will never want to make this number smaller than 0.01. To acquire data at faster speeds requires hardware paced acquisition which, if supported by your device, is described in the section on your device.

9. Click on the **Apply** button at the top of the screen.

   The changes you make are not implemented by DAQFactory until you click **Apply**. If you do not want to keep your changes, you can click **Discard**. Once you click **Apply**, DAQFactory will start taking data from the test device, D#0, I/O Type A to D, Channel #0, and place that data in the "Pressure" channel.



10. Click on the **+** sign next to **CHANNELS** in the workspace.

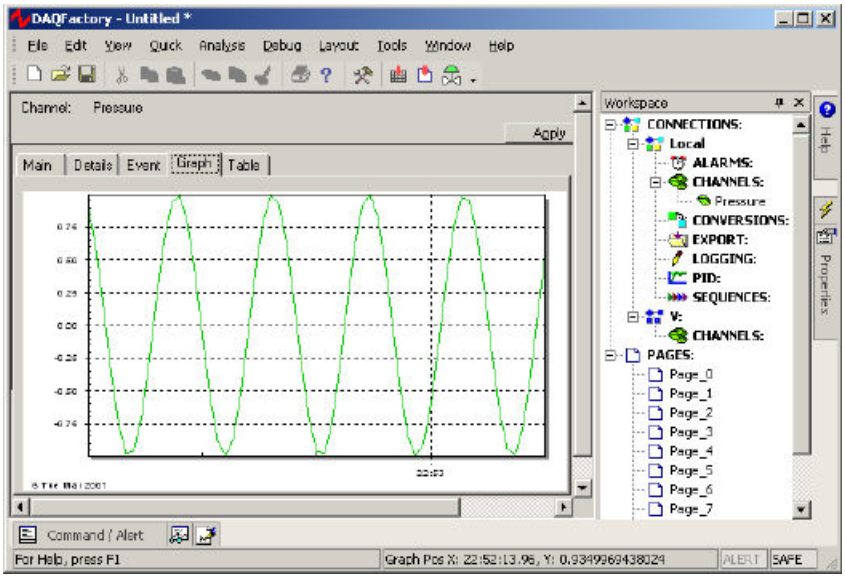
    The **+** sign appeared when you hit **Apply**. Clicking on it displays the channels you've created as separate items in the workspace.

11. Click on **Pressure** listed under **CHANNELS** in the workspace.

    This will display the channel view for the channel Pressure. The channel view displays the configuration information for a single channel. In general, it is quicker to create your channels in the channel table, but the channel view provides space for some extra details.

12. In the **Channel View** that appears, click on the **Graph** tab.

    The channel view also provides a quick graph and table to view your incoming data. This can be used to confirm that data is being acquired and that you are reading the correct data. You should see a sine wave displayed in the graph. You have just confirmed that you are acquiring data on your pressure channel.



13. If you'd like to see the table, click on the **Table** tab.

You are now taking data. In the next section we'll display it on our own custom screens.

For more information on channels and the various parameters available, see the chapter on Channels and Conversions.

2.6 Displaying the Data

The Channel View certainly provides a quick look at your data, but is not the most convenient or flexible way to display your data, especially once you have more than one channel. In this section we'll add a component some different components to a page that will display data from the pressure channel.

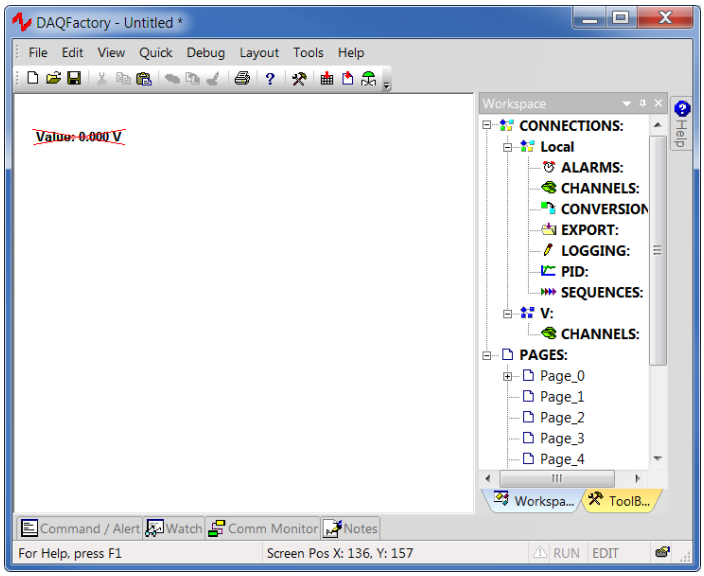
1. Click on **Page_0** in the workspace under **PAGES**.

   This will display a blank white page. When DAQFactory starts, 10 blank pages are generated automatically with default names. You are not limited to these 10 pages and can create as many pages as you want.

2. Right-click somewhere in the middle of the blank area and select **Displays-Variable Value**.

   This will place a new page component on the screen. There are many different components available that allow you to create custom screens. This particular component simply displays a changing value in text form.

   You can also drag the Variable Value component from the Toolbox.



NOTE: You must hold down the Ctrl key to select and move screen controls. If you do not hold the Ctrl key, DAQFactory will assume you want to interact with the control (i.e. press a button, drag a slider, etc...). There is an Edit mode, which allows you to select and move screen controls with holding the Ctrl key, but we generally do not recommend it as it is much faster to simply hold the Ctrl key than switch modes all the time.

3. Right click on the new component and select **Properties...**. If the component is not selected you may need to Ctrl-Click on the component to select it again.

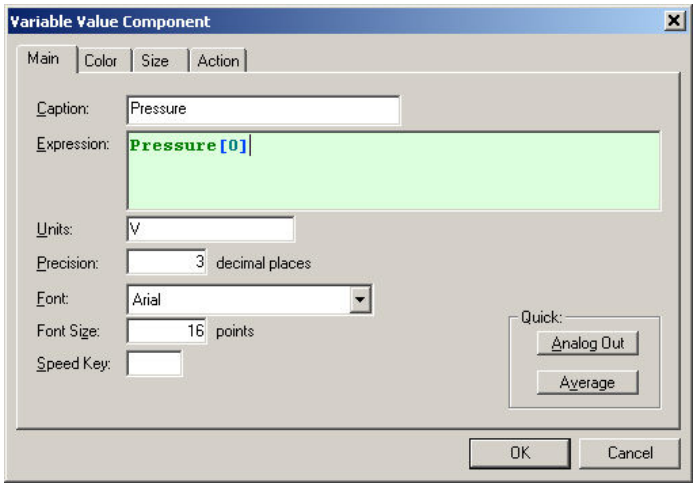
   All of the screen components have different properties windows for configuring the look of the component and telling the component what data to display.

4. Next to **Caption:**, enter `Pressure`.

   The caption is what is displayed in front of your value. A colon is automatically added when displayed.

5. Next to **Expression:**, enter `Pressure[0]`.

   Expressions are actually formulas that you can enter to display calculated data. In this case, the formula simply requests the most recent data point of the channel pressure. The [0] indicates the most recent data point. If you did [1] you'd get the next most recent data point and so forth back to the History length of the channel, which we left in the default of 3600. You could just as easily put `Pressure[0] * 10` if you wanted to display the channel's value times ten.



NOTE: A common error is to use () instead of [] for subsetting. Although this works in some cases, it does not work in all, nor do we guarantee that it will continue to work in newer versions.

6. Click on **OK**.

   Now the screen should display **Pressure: 0.324 V** with a changing number. This is the current value in the pressure channel we created earlier. Feel free to open the properties window again and play with the units, font and other settings.

7. Ctrl-Click on the first **VariableValue** component.

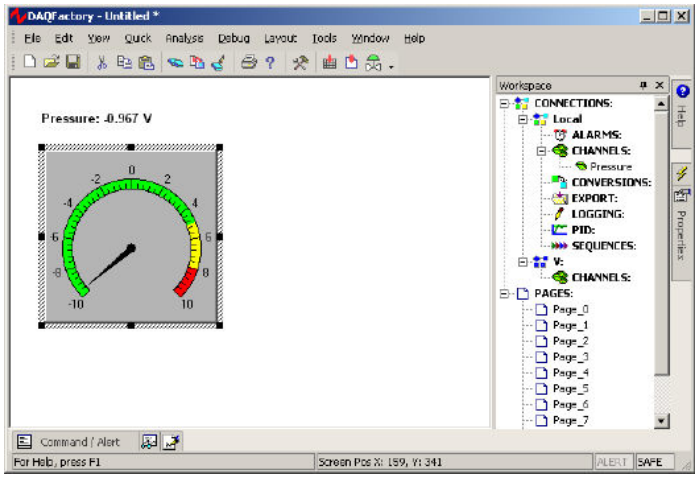
   This will select the component, as indicated by the shaded rectangle surrounding it. Remember: you must hold down the Ctrl key while clicking to select the component.

8. Ctrl-Click and drag the **VariableValue** component to a new location on the screen.

   This allows you to move the component around the screen. Again, you must hold down the Ctrl key to move a component around.

The next steps will not work in DAQFactory Express because it doesn't include gauges. Consider upgrading to any of the paid versions of DAQFactory to get gauges and thirty some other controls, among other things.

9. Right click somewhere else on the screen away from the displayed component and select **Gauges-Angular**.



10. Right click on the new component and select **Properties...** to open its properties window.

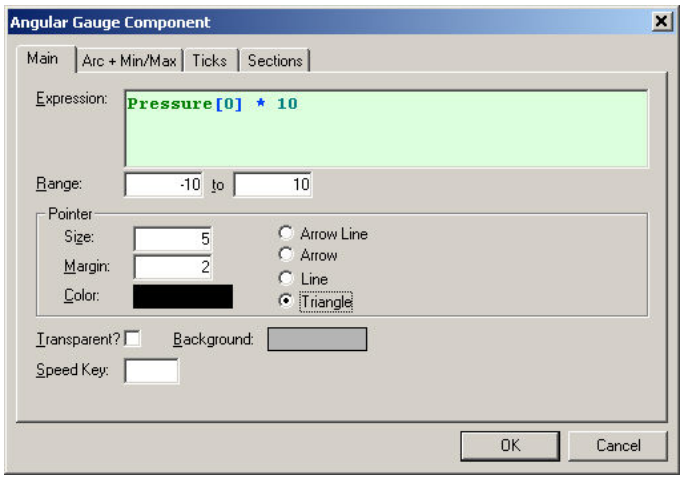
11. For the expression, enter `Pressure[0] * 10.`

    As we mentioned before, expressions are actually formula so wherever there is an expression we enter in

calculations. This particular one simply takes the most recent value of the pressure channel and multiplies it by 10.

12. For the Range, enter **-10 to 10.**

    Since pressure is simply a sine wave of amplitude one, and we're multiplying this by 10, the range this component will see is -10 to +10.



13. Click **OK**.

    The gauge will now be scaled from -10 to 10 and the gauge arrow will move with the value of pressure. Notice how the first screen component still displays values from -1 to 1. Both of these components are independent of each other.

14. Ctrl-Click on the first **Variable Value** component.

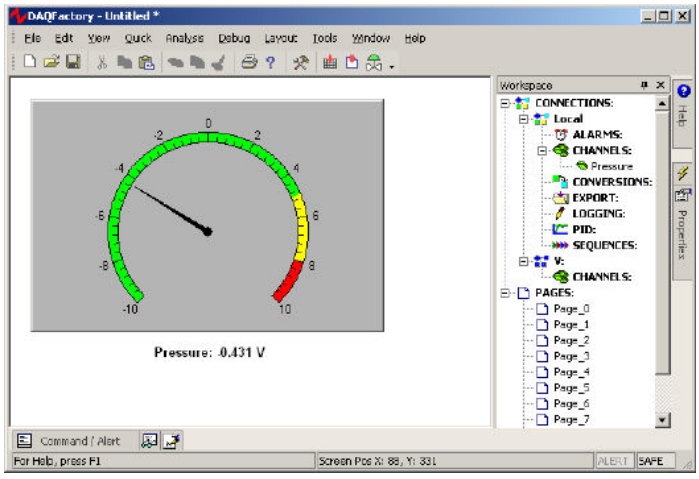
    This will select the component, as indicated by the shaded rectangle surrounding it.

15. Ctrl-Click and drag the **Variable Value** component to a new location on the screen.

    This allows you to move the component around the screen. Try the same thing with the gauge.

16. Ctrl-Click on the gauge. Square dots will appear at the corners of the gauge. Click and drag these dots to resize the graph.

    The gauge is a resizable component and displays square black dots around the selection rectangle for resizing. The **Variable Value** component automatically sizes to its contents so does not display the dots.



You now know how to display your data using several different screen components. Next we

will learn how to use the graph to display historical data.

In the mean time, feel free to play with other similar components if you'd like. When done, you can delete them by selecting the component and then right clicking and selecting **Delete Component**.

For more information on page components and creating your user interface, see the chapter entitled Pages and Components.

For more information on expressions and the functions available for use in expressions, see the chapter entitled Expressions.

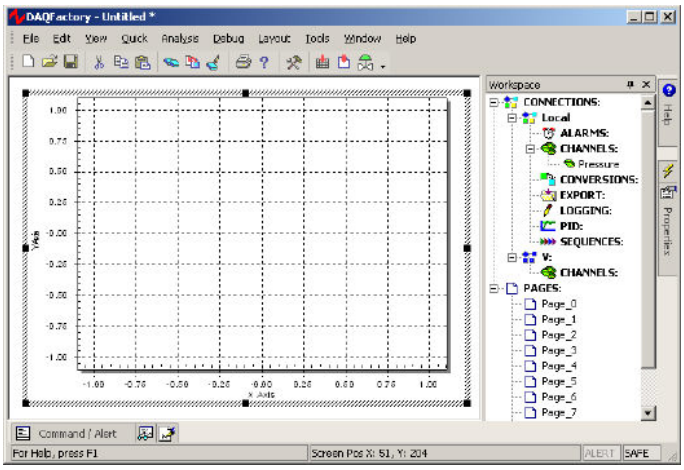
2.7 Graphing the data

Displaying scalar values in text or gauge form is certainly useful, but nothing is better than a graph to give you a handle on what is happening with your system. DAQFactory offers many advanced graphing capabilities to help you better understand your system. In this section we will make a simple Y vs time or trend graph.

1. If you are still displaying the page with the gaugevalue on it, hit the **1** key to switch to Page_1. If you are in a different view, click on **Page_1** in the workspace.

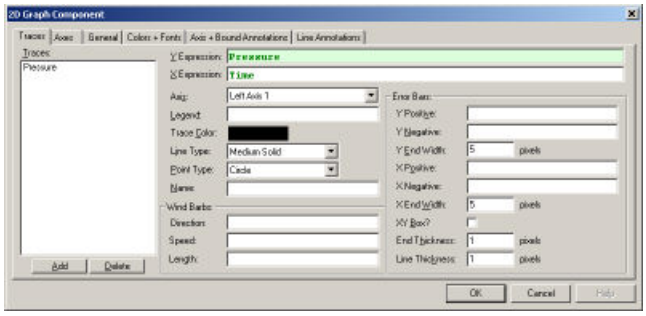
> The workspace is only one way to switch among pages. Another is using speed keys which can be assigned to a page. By default, the initial 10 pages are assigned the keys 1 through 0.

2. Right click somewhere on the blank page and select **Graphs - 2D Graph**. Move and resize the graph so it takes up most of the screen. To move the graph, simply ctrl-click and drag. To resize, ctrl-click and drag one of the 8 dark squares around the outside of the graph. Next, open the properties window for the graph.



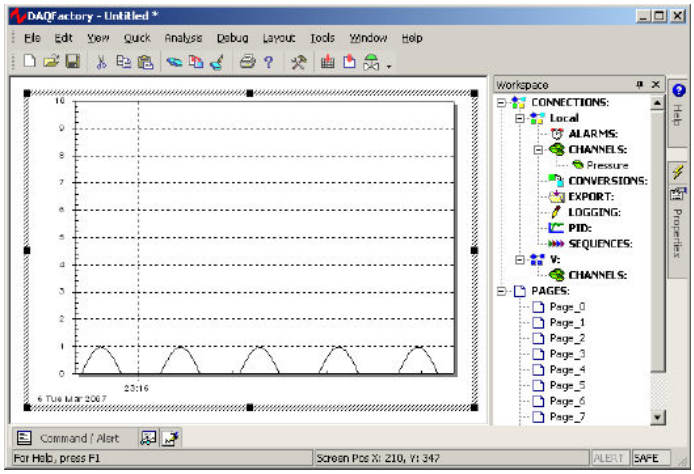
3. Next to **Y Expression:** type **Pressure**.

> The Y expression is an expression just like the others we have seen so far. The difference is that a graph expects a list (or array) of values to plot, where in the previous components we have only wanted the most recent value. By simply naming the channel in the Y expression and not putting a [0] after it, we are telling the graph we want the entire history of values stored in memory for the channel. The history length, which determines how many values are kept in memory and therefore how far back in time we can plot is one of the parameters of each channel that we left in its default setting of 3600.



4. Leave all the rest in their default settings and click **OK**.

   You should now see the top half of a sine wave displayed in the graph. The sine wave will move from right to left as more data is acquired for the pressure channel. To display the entire sine wave, we need to change the Y axis scaling.

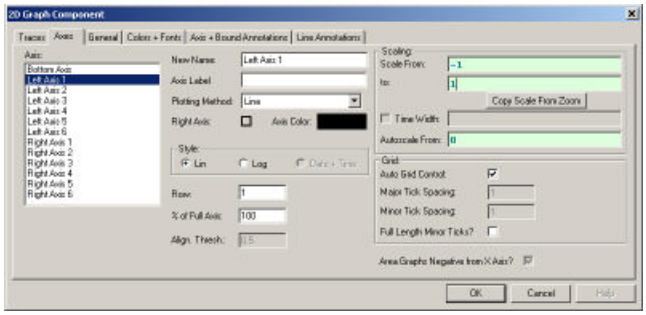


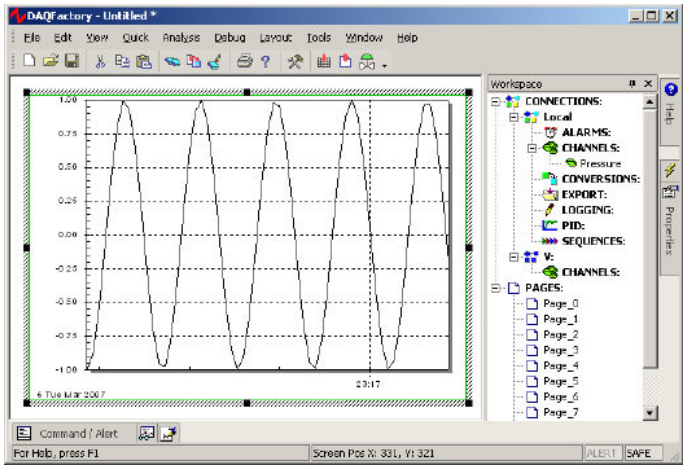
5. Double click on the left axis of the graph. In this case you do not need to hold the Ctrl key.

   Unlike other components, you can double click on different areas of the graph to open the properties window for the graph to different pages. Double clicking on the left axis brings up the axis page with the left axis selected.

6. Next to **Scale From** enter **-1**, next to **Scale To** enter **1**, then click **OK**.

   This will scale the graph from -1 to 1 which will show the entire graph.





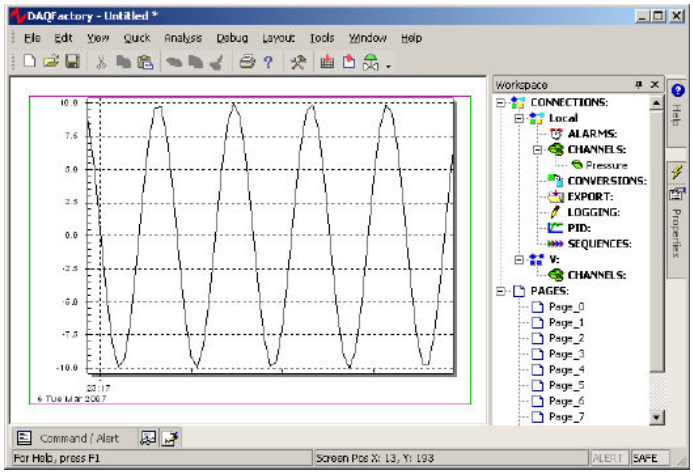
Now we'll try another method to scale the graph.

7. Open the graph properties box again and go to the **Traces** page. Change the Y Expression to **`Pressure * 10`** and hit **OK**.

> Like the other screen components, the expressions in graphs can be calculations as well as simple channels. In this case, we are multiplying each of the values of Pressure's history by 10 and plotting them. Since our range is -1 to 1, the graph will only display part of the sine wave.

8. Deselect the graph by clicking on the page outside of the graph. The shaded rectangle should disappear. Next, right click on the graph and select **AutoScale - Y Axis**

> The graph component has two different right click popup menus. When selected, it displays the same popup menu as the rest of the components. When it is unselected, it displays a completely different popup for manipulating the special features of the graph.
>
> After autoscaling, you should see the graph properly scaled from -10 to 10. Notice the rectangle around the graph. The left and right sides are green, while the top and bottom are purple. The purple lines indicate that the Y axis of the graph is "frozen". A frozen axis ignores the scaling parameters set in the properties window (like we did in step 6 above) and uses the scaling from an autoscale, pan, or zoom.



9. To "thaw" the frozen axis, right click on the graph and select **Freeze/Thaw - Thaw Y Axis**

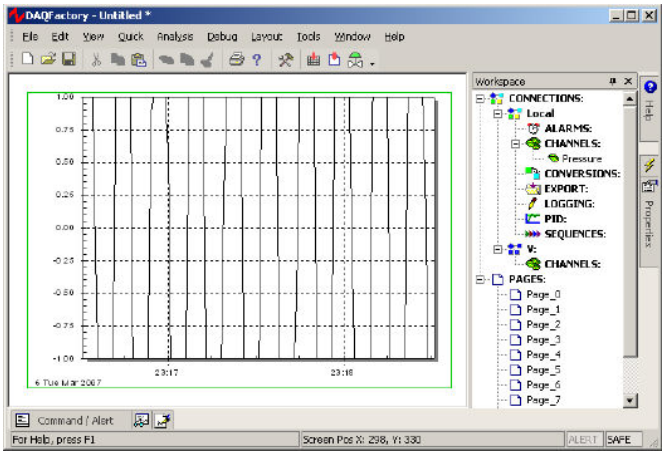
> Once thawed, the graph will revert to the -1 to 1 scaling indicated in the properties box and the box surrounding the graph will be drawn entirely in green.

10. Double click on the bottom axis to open the properties to the axis page with the bottom axis selected. Next to **Time Width:**, enter **`120`** and hit **OK**.

If the double click method does not work, you can always open the properties window for the graph using normal methods, select the **Axes** page and click on **Bottom Axis**.

In a vs. time graph, the bottom axis does not have a fixed scale. It changes as new data comes in so that the new data always appears at the very right of the graph. The time width parameter determines how far back in time from the most recent data point is displayed. By changing it from the default 60 to 120 we have told the graph to plot the last 2 minutes of data.

Once again, if you zoom, autoscale, or pan the graph along the x axis, it will freeze the x axis and the graph will no longer update with newly acquired values. You must then thaw the axis to have the graph display a marching trace as normal.



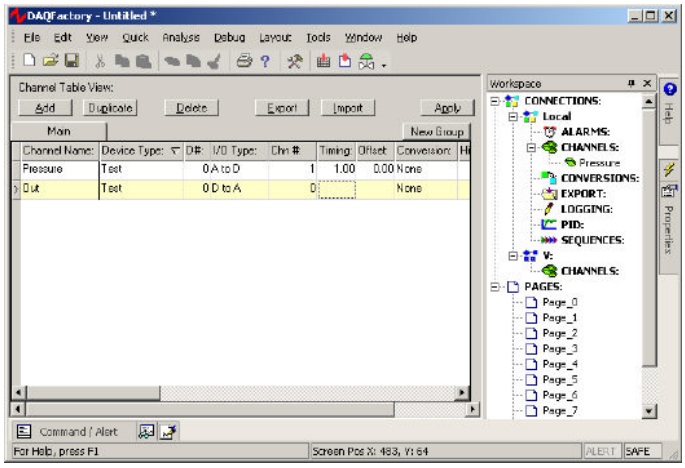
For more information on graphs and the many ways you can display your data in graph form, go to the chapter entitled Graphing and Trending.

2.8 Controlling an output

In addition to taking data, most systems also control outputs. Here DAQFactory offers many advanced automated control features such as PID loop control and sequences, but for now we will just create an output channel and control it manually from a page component.

1. Click on **CHANNELS:** in the workspace to go to the channel table.
2. Click on **Add** to create a new row in the table.
3. In the new row, name the channel **`Out,`** with an I/O type of **`D to A`** and a Channel # of **`0`**. Leave the rest in their defaults.
4. Click on **Apply** to save your changes.

The output channel has been created, but it hasn't been set to anything.



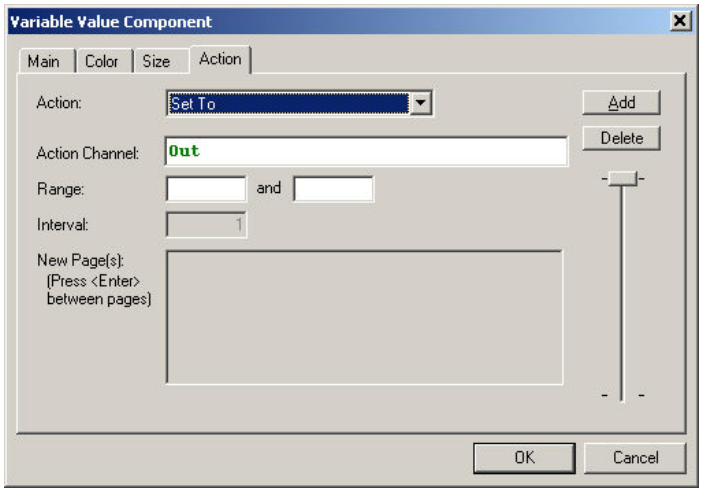
5. Click on **Page_0** to go to our first page.
6. Right-click and select **Displays-Variable Value** to create another variable value component.
7. Right click on the new component and select **Properties…** to open the properties window.
8. For the expression, enter **`Out[0]`**.

   Feel free to set the caption as well. Like before, this will simply display the most recent value of the out channel.
9. Click on **Action** tab.

   This tab exists on several different components including the static ones and works the same way with all of them.
10. From the **Action** drop down, select **Set To.**

    There are many different options for the **Action** described later in the manual. The **Set To** action will prompt the user for a new value when the component is clicked.
11. Next go to **Action Channel:** type **`Out`**.
12. Leave the **Range** blank and click **OK**.

    The range allows you to constrain the inputs to particular values. By leaving these properties blank, we are indicating that we do not want a range limitation. The page will now display your caption and 0 with a big red X through it. This indicates that out does not have a valid value yet. This is because we haven't set it to anything.



13. Click on the component. Don't hold Ctrl because we want to interact with the control, not move it around. A new window will appear requesting a new value. Enter any number and hit **OK**.

    Out will now be set to the value you entered. The component will now display your new value without the big red X.

The next steps will not work in DAQFactory Express because it doesn't include knobs. Consider upgrading to any of the paid versions of DAQFactory to get knobs and thirty some other controls, among other things.

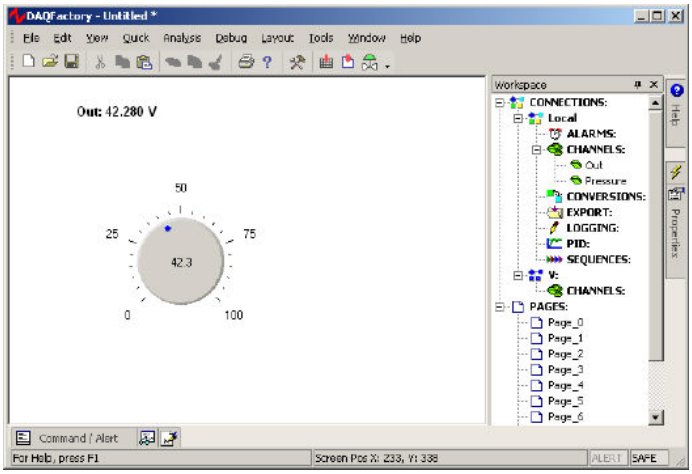
14. To try a different control, right click somewhere else on the screen and select **Sliders & Knobs - Knob**. Right click on the new component and select **Properties...** to open the properties window.

15. Next to **Set Channel**, type **Out**. Leave the rest of the properties at their default values and click **OK**.

    The knob is now set to control the out channel.

16. Click on the knob and drag in a circle as if turning the knob.

    You may have to click once beforehand to deselect the knob. The number inside will change indicating the present setting and out will be changed to this value. If you look at the variable value component we just created you will see it update as well. Likewise, if you click on the **variable value** component and set it to a different value (between the current knob's range of 0 to 100) you will see the knob update to the new setting.



You now know how to set an output to value using several different screen components. Next we will log the incoming data to disk.

For more information on the components that manually control outputs, see the chapter entitled Pages and Components. Most of the components can be used to control outputs.

For more information on PID loop control see the chapter on PID loops.

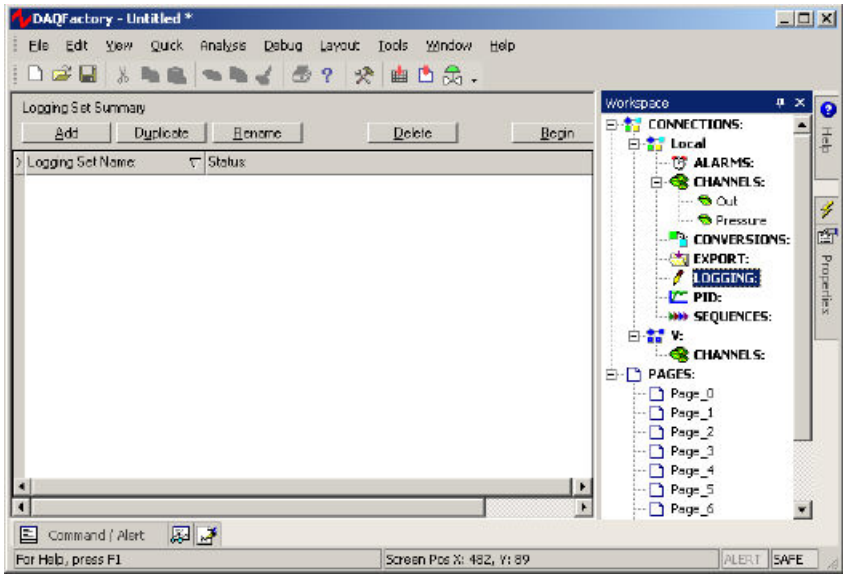
For more information on sequences and automating your system, see the chapter on Sequences.

2.9 Logging Data

So far we have seen how to take data and display it on the screen. Next we will learn how to store this data to disk so that it can be opened with another program for analysis. In this example we will create a simple comma delimited file with our pressure data.

1. Click on **LOGGING:** in the workspace.

   This will display the logging set summary view. Here you can see all your logging sets displayed and their running status. A logging set is a group of data values being logged in a particular way to a particular file. You can have as many logging sets as you need, and they can run concurrently.



2. Click on the **Add** button in the logging set and when prompted for the new name, type in **Log** and click **OK**.

   Like channels and all other DAQFactory names, the logging set name must start with a letter and only contain letters, numbers or the underscore. Once you click **OK**, the logging set view will be displayed for the new logging set. You can also get to this view by clicking on the logging set name listed under **LOGGING:** in the workspace. You may have to click on the **+** sign next to **LOGGING** to see your logging sets listed in the workspace.

3. Next to **Logging Method**, select **ASCII Delimited**

   ASCII Delimited is probably the most common method for data logging as it can be read by most other programs such as Excel. Unfortunately, it is not as space efficient or fast as the binary methods. But unless you have strict space constraints or you are logging more than about 10,000 points per second (depending also on your computer / hard drive speed), we suggest ASCII since it can easily be read by other programs like Excel.

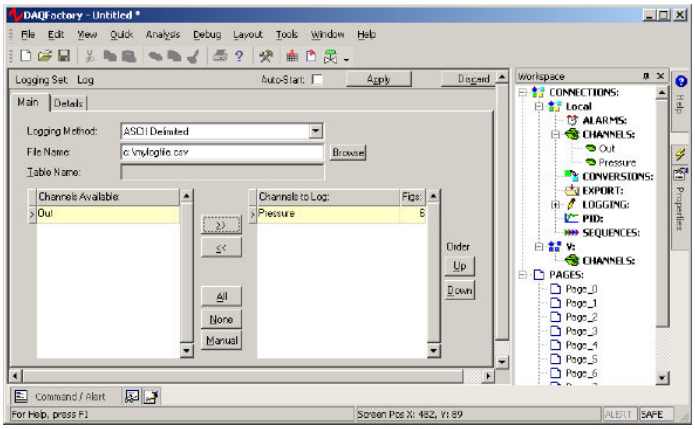
4. Next to **File Name** enter **`c:\DAQFactory\mylogfile.csv`**

   It is usually best to fully specify the path to your file, otherwise the data will be stored in your DAQFactory directory. The **`.csv`** is a windows standard designation for comma delimited values. Note the c:\DAQFactory directory will need to exist (and will if you installed DAQFactory in its normal place)

5. In the **Channels Available** table, click on the row with **Pressure**, then click on the **>>** button to move it to the **Channels to Log** table.

   Each logging set can log any combination of channels. In this case, we will just log the input channel.

6. Click on **Apply** to save the changes.



7. To start logging, click on the **+** next to **LOGGING** to display the new logging set, then right click on the logging set **Log** and select **Begin Logging Set**.

   Once started, the icon next to **Log** will change and the red stop sign will be removed to indicate that this logging set is running.

8. Wait at least 10 or 15 seconds to log some data and then right click on the logging set again and select **End Logging Set** to stop logging.

   There are other ways to start and stop logging sets, including the action page of some components such as the variable value component that we used earlier.

9. Now start Excel or Notepad and open the file **c:\DAQFactory\mylogfile.csv**.

   You will see two columns, time and data. By default the time is given in a format specific for your locale. DAQFactory can format this however you like, as well as log in a more computer friendly seconds since 1970 format.

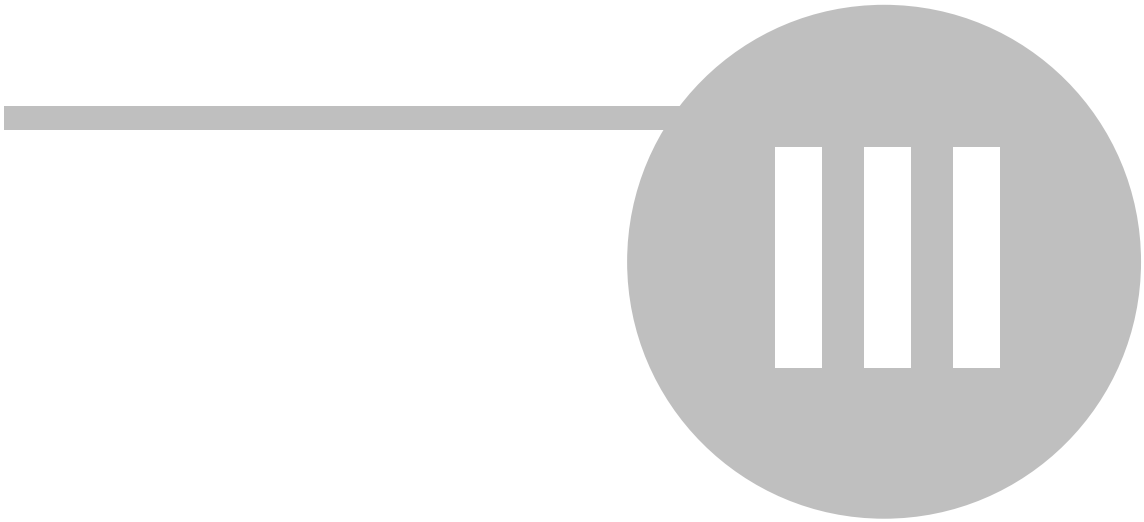
   Hint: if you cannot find the mylogfile.csv file, check to make sure that you selected Pressure to log and not Out. Since Out is an output channel, it only gets new values to log when you actually set it to something.

You have now acquired, displayed and logged your data.

For simpler applications, it is simply a matter of repeating most of these steps for additional channels. But even for simpler applications there are many adjustments available for fine tuning your application to work the way you want. These, along with many more walk-throughs and examples are provided in the appropriate sections of this manual. At this point we suggest playing with DAQFactory a bit, jumping to the appropriate sections of the manual for more detail on the areas of DAQFactory you may be interested in utilizing.

For more information on logging including the various file formats you can use, please see the chapter entitled Data Logging and Exporting.

3 The DAQFactory Document



3 The DAQFactory Document

3.1 Document Overview

All the settings for a particular application, channels, pages, components, sequences etc. are all kept in a DAQFactory document with the extension .ctl. You can open and close documents at will while keeping different groups of settings in different documents for use in different applications. Whenever you open another document, the acquisition and automation running in the current document is shut down and the new document is loaded and its acquisition started. Because all the settings are stored in this single file, you can move the .ctl document file from one computer to another running DAQFactory and load your application there (provided of course you have the same hardware available). The only exception are the user devices and protocols discussed in the Extending DAQFactory and Serial Communications chapters, preferences and customization discussed in the Other Features chapter.

This chapter discusses some of the document specific tasks you may perform and document wide settings.

3.2 Starting a new document

When DAQFactory starts, a new document is created, ready for you to start creating your application. If you have a document already loaded and you want to start from scratch, select **File - New** from the main menu. This will stop all acquisition, PID loops, sequences, and logging sets, clear all channels, conversions, pages, components and other objects and leave you with the same blank document you are presented with when starting DAQFactory. From here you have a clean slate to create a new masterpiece.

Note: Depending on the complexity of the currently running document, it may take a short bit, even 10 or 20 seconds or more to properly shut down the application. Please be patient.

3.3 Saving a document

Once you have started creating your application you will want to save what you have done so you do not lose anything. This is smart with any program, and we suggest saving often. To save your document, simply select **File - Save** or **File - Save As** from the main menu. This is just like any other Windows application. It is usually best to be in page view, or at least to have recently applied changes to your displayed view before saving to ensure that all your changes are saved to disk. Actually, unless you are in the sequence view, DAQFactory will automatically revert you to the page view, prompting to apply any changes, before saving. From the sequence view, saving a document automatically applies the changes to the sequence you are working on and saves the document without switching you to the page view.

Saving your document should not affect your acquisition or any automation processes that are running.

3.4 Saving with History

This feature is now depreciated. We recommend using channel persistence to maintain a history between restarts of DAQFactory. Currently channel persistence does not support spectral, image or string channels, so the Save with History feature provides an alternative and will remain until those channel types are supported.

The regular save methods do not save historical data of your channels. When you reload your document, the history for all your channels will be cleared out until new data is acquired. Since graphs use this historic data for display, the graphs will also clear out and will generate as new data arrives. If you wish to save historic data with your document, select **File - Save with History** or **File - Save as with History**. This will save all the settings that a normal **Save** would, plus the historic data for all the channels. Depending on your history sizes and the number of channels, this can make for a rather large file. Documents saved with history open the same way as normal documents. Once reopened, you will need to **Save with History** again if you wish to continue to save the document with newly acquired

data.

Note: Both Save and Save with History will save non-calculated V. channel data to disk. See V.Channels for more detail.

Note: Opening a file saved with History does not update any output channels. The channels will have the same history from when they were saved, but the actual output I/O on the device is not updated with the last value in the history. You would need to manually do this from a start up sequence.

3.5 Opening a document

To load a document and start acquiring data using the settings of that document, select **File - Open** from the main menu and select the desired document file. This will stop any existing document running, load the new document, and start acquisition under the new parameters.

Documents created in different versions (Pro vs. Base vs. Express) are compatible, however, when downgrading, not all the features of the document may be available.

Note: Depending on the complexity of an already running document, it may take a short amount of time to shut down that document, load the new document and start it up.

3.6 Safe mode

DAQFactory is a very powerful program with many automated features that can start as soon as a document is loaded. Unfortunately, with this automation it is possible to set DAQFactory in a mode where it does not function properly when the document loads. The most common occurrence of this is when you load a document and hardware that the document communicates with does not exist or is malfunctioning. To provide an escape, there is a 'Safe Mode'. Safe mode does not acquire any data, and does not start any auto-starting sequences, PID loops, or logging sets. This allows you to load the document and make modifications without running into problems or generating lots of missing hardware type errors. To open a document in safe mode, select **File - Open in Safe Mode...** instead of **File - Open**. Once open, the **Safe** indicator in the bottom right corner of the DAQFactory window will display red. When not in safe mode, this indicator will show "RUN" in grey.

Safe mode also does not allow you to execute any File., or DB. functions, nor execute the extern() or system. shellExecute() functions. This is to offer additional security when opening 3rd party applications.

Once you have the document set the way you want, you can exit safe mode by selecting **File - Leave Safe Mode** from the main menu. This will start acquisition and any auto-starting sequences, PID loops, or logging sets.

If you run into problems after already opening a document, you can also switch into safe mode by selecting **File - Switch to Safe Mode**. This will stop acquisition and all sequences, PID loops, logging and export sets that may be running while leaving the document fully loaded.

Application: Safe mode could also be used as a way to start and stop experimental runs. You could set up a complete experiment that acquires data and performs automation tasks automatically. You could then load your document in safe mode, then when ready to start a run, leave safe mode. Data will begin to acquire, get logged, and automation will start. When the run is complete, you could just switch to safe mode to stop all of this. When ready for the next run, simply leave safe mode again and repeat. Two things may help with this application: Load in Safe Mode which is a document setting that causes your document to always load in safe mode, even when **File-Open** is selected, and System.SafeMode, which is a system variable that can be used in a sequence to automatically end your run.

This isn't the most flexible way to achieve this end, but is probably the easiest. The better way is to avoid using the Timing parameter of your channels and set them all to 0. Then use a sequence to trigger the reading of your I/O and simply start and stop your sequences as needed.

3.7 Autoloading a document

DAQFactory normally starts with a blank document, at which point you can go to **File-Open** or use the most recent document list to open your document.

To have DAQFactory automatically load your document you need to create a shortcut to DAQFactory and provide the name of your document as the command line parameter:

1. Create a shortcut to DAQFactory. The easiest way is to right click on **DAQFactory.exe** in Explorer and select **Create Shortcut**. Name it something more useful than **Shortcut to DAQFactory** if you'd like.

2. Now, open the properties for the shortcut by right clicking on the shortcut and select **Properties**.

3. The target box will have the path to DAQFactory, more than likely it will be **c:\DAQFactory\DAQFactory.exe**. Add a space and then the name of your document:

```
c:\DAQFactory\DAQFactory.exe MyDocument.ctl
```

If the document is not in your DAQFactory directory, you'll need to specify the full path to the document, usually in quotes.

4. Make sure the "Start in" specifies the path in which DAQFactory is installed, for example: **c:\DAQFactory**

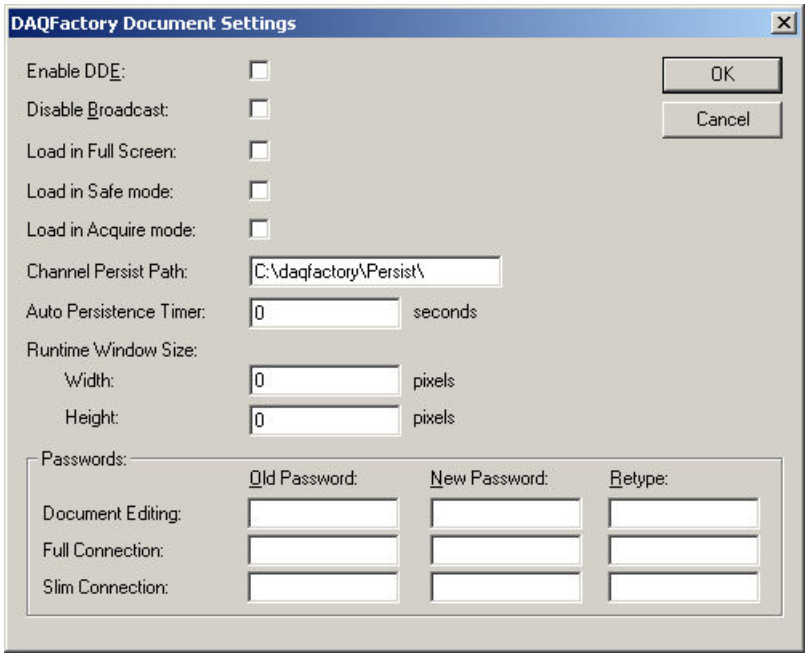
Now starting the shortcut will start DAQFactory and load the document.

Note: if you are using Express, you will need to include -runAsExpress in the command line. We recommend copying the Express shortcut and adding your file path to it.

As an alternative, and assuming you haven't done a weird or double installation you can simply double click on the document and both DAQFactory and the document will load, but this does not work for Express and can throw your scripts off if they make start up folder assumptions. You can also make a shortcut to the document if you'd like to put it in your start menu.

3.8 Document settings

Various document settings can be changed by selecting **File-Document Settings** from the main menu.



Enable DDE: DDE is a Windows mechanism for sharing data with another application running on your computer. You could use DDE for example, to fill cells in an Excel spreadsheet in real time. Unfortunately, acting as a DDE server requires more processor power and overhead. Because of this, by default, DDE is disabled. To enable DDE for this document, check the enable DDE box. DDE is described in further detail in the networking chapter.

Disable Broadcast: Broadcasting refers to the mechanism DAQFactory uses to transfer data in real time to other copies of DAQFactory running remotely. By checking this box, the internal network data server is disabled and no remote copies of DAQFactory can connect to this system. For security reasons, this is checked by default, so to use DAQFactory Networking, you must specifically uncheck this box. You will then need to save your document and restart to have the change take effect. We strongly recommend setting both a Full Connection and Slim Connection password as described below. This ensures that the data encryption used for communication uses a key that is unique to your application. If the password is not specified, the communication is encrypted with the same key as everyone else who leaves their password blank. The only time this would be safe would be when the system is on a completely stand alone network.

Application: You could also disable broadcasting to absolute ensure that no one could have access to your real time data. This is why it is disabled by default. Broadcasting and networking is discussed in the networking chapter.

Disable Reboot / Restart: if you have broadcasting enabled and are connected over the Full data stream, you can remotely reboot or shutdown the remote system. This is useful in headless applications. Since these applications are rare, this is checked (i.e. disabled) by default. To enable this feature you will need to uncheck this box, save your document then restart.

Load in Full Screen: Checking this will cause this document to load in full screen mode instead of the standard mode with a window frame, menus, etc. You can manually switch between full screen and non full screen by clicking View - Full Screen from the main menu.

Load in Safe Mode: By checking this box, you can force this document to always open in safe mode. You can then start the acquisition and automation by selecting **File-Leave Safe Mode**, or use a system function. This allows you to create an application where an experiment might be started and stopped. See Safe mode for more info on this application.

Channel Persist Path: This allows you to change the directory where the channel persist files are stored. If you are only working on one document, then you probably don't need to change this. If, however, you are working with multiple documents with the possibility of the same channel names amongst them, you will probably want to set each document with its own persist path so they don't conflict.

Auto Persistence Timer: Checking this box causes DAQFactory to perform a 'Save with History' automatically in the background at the interval specified (in seconds). This feature has been depreciated. We strongly recommend using channel persistence instead. If the computer power is lost in the middle of doing the save with this method, your document could be corrupted.

Application: The auto persistence timer is designed for users where power outages are common and they want to avoid having their graphs clear out every time the computer resets. Channel persistence achieves the same thing with more flexibility and more robustly.

Runtime Window Size: The runtime version of DAQFactory does not have a menu, toolbar, workspace or other docking windows. It only has the page view. When the runtime engine loads this document, it will automatically size its window to the size given here in pixels. This keeps the runtime window from having a huge amount of white space when displayed on larger monitors.

Passwords: There are three different passwords available. Each one is set by entering the current password (which defaults to blank) and then entering the new password twice. If you do not enter anything in these boxes, the password setting remains the same. Note that these passwords are not encrypted and are designed only as a deterrent. In cases where security is of grave concern, other methods should be employed. If you do not wish to require a password for a given event, set the password to blank.

Document Editing password: If specified, this password is required to open the document in any version of DAQFactory except in runtime mode. Note that the password cannot end with spaces.

Application: The document editing password allows you to create a document that you can distribute to anyone for use, but that cannot be edited even if they purchase a development copy of DAQFactory.

Full / Slim Connection: These passwords are required by a remote copy of DAQFactory to connect to the full or slim data stream of this document. See Networking for more information.

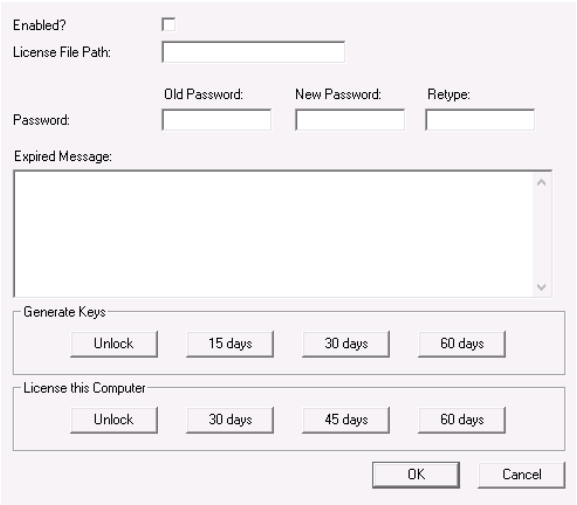
Application: If you do not want anyone to have access to your system remotely, but you still wish to have access, provide a password for both the slim and full stream. You can use the same password if you would like.

Application: If you want to share some data with other users, but keep them from seeing all the data and making changes to channels, sequences and the like, put a password on the full connection, but leave the slim connection open, or better yet, make your slim connection password something simple and tell everyone it. This at least causes the data encryption to be unique to your application and protect it from hackers.

3.9 Document Licensing

To protect a document you have created from being edited, you can use the document editing password described in 3.8 Document Settings. However, this does not prevent someone from purchasing a Runtime licence of DAQFactory (or using the embedded license from your Developer created document) and running your application on multiple computers. There are also occasions where you are providing a complete system with hardware and software and you want to make sure that you get paid. For both these cases, and probably others, DAQFactory supports a document licensing feature. Once enabled, whenever your document is opened on a new computer, it will not function until a key is provided to either make your document fully functioning forever, or to have it expire after a preset amount of time. This mechanism is document specific and completely separate from the licensing mechanism used by DAQFactory itself.

To use this feature, select File - Document Licensing... from the main menu. A popup will appear with the licensing administration options:



First you will need to Enable the feature, but in addition to selected Enabled, you will need to provide a full path and file name for the license file. This file is where licensing information is stored. It will need to be located in a location where DAQFactory can write to it, but otherwise can be any file name. We recommend using something that isn't obvious, for example, a random name with a .dll extension. It is up to you, but do not create a file in the DAQFactory folder with .dds or .ddp as all of these are loaded by DAQFactory on startup. The folder MUST exist where the file is to be created. DAQFactory will not create the folder for you and the licensing won't stick if the folder does not exist.

Finally you must provide a password of at least 4 characters. This password is separate from any document editing password you may have implemented. That way, you can provide the document editing password to your employees to allow them to enhance the document, but without enabling them to license your customers. Once you specify a password, you can only get to the setup popup by entering the password. For extra security, if an incorrect password is entered trying to get to the setup popup, DAQFactory will assume someone is trying to break your lock and will quit. This makes it much harder for someone to try a bunch of passwords in hopes of figuring it out.

Do not lose your password! We cannot recover it for you! If you lose the password you will not be able to use your application on systems that aren't otherwise licensed and you won't be able to install your application on any new computers. Once again, while we can recover document editing passwords for a fee and subject to proof that you created the document, we CANNOT recover the licensing password for you as there is no way for you to prove to us that you are the owner of the document other than knowing this password. That all said, we do offer a

document registration service. You provide us with your document before you enable the license feature, or at least before you have forgotten the licensing password, along with the emails of people authorized to retrieve the password, and in the case of a lost password, we can retrieve it for these individuals or the president / owner of the company it is registered to. There is a fee for this service and for any password recovery. We have to have your document ahead of time to retrieve an internal unique identifier. This identifier stays with your document even after you edit it and is what gets registered with the recovery email addresses. Please contact us to set this up.

You will likely also want to provide an Expired Message. When your application is expired, a popup will appear with a key, and requesting a code. It will also display any message you provide. This message should in general explain what happened and provide contact information so your customer can get a code. Please note that this application licensing feature does not support unattended licensing the way DAQFactory is licensed. Your customer must contact you with a key, and then you have to generate a code from that key and give it them. The key changes every time the popup appears, so make sure you customer knows not to close the popup before contacting you.

Please be sure to include in the message that your customers should NOT contact AzeoTech for unlock codes! We cannot generate them even if we wanted to as we do not know your password. If we get a lot of support calls from our customer's customers asking for unlock codes we will be forced to add the message ourself, and would rather let our customer determine what the message should say.

Once enabled, whenever the document is loaded it will look for the license file you specified. If it does not exist, which would happen on a new install, DAQFactory will create it, marking the application as expired. The application will not run, and the customer will be prompted with your message and a key and asked for a code. If the file does exist, DAQFactory will check several things to make sure it is valid. First it will check that the file was created for this computer. If not, it will mark it as expired. This is to prevent someone who discovers which file you are using for licensing from taking a licensed file and copying it to another computer. Next, DAQFactory will check the times in the license file. It will first check to make sure that the customer didn't turn back their clock (other than DST), and then it will see if the expiration date has past. Any of these events will cause the expired popup to appear requesting a code.

Of course when you first enable the document licensing feature, the development computer you are using won't be licensed. To avoid problems, there are three things to help you:

1) once you go into the license setup popup, all license checking for the machine you are on is disabled until DAQFactory restarts or the application is reloaded.
2) in the license setup popup there are 4 buttons along the bottom that allow you to license the computer you are using without using the key/code mechanism. So once you enable it on your development machine, you can simply click the Unlock button under License this Computer so that your development computer doesn't prompt you for a code. These buttons are also useful if you are shipping a PC complete with DAQFactory to your customer. You can thus install your application and go to the license setup popup and select one of the 3 expiration times and the application will run until the specified number of days has occurred.
3) if you are presented with the expired popup requesting a code, you can type in "admin" without the quotes and get to the license setup popup. You will, of course, be prompted for your password first, and if you enter the wrong one, like before the application will quit. This works even on runtime installations, though it is important to note that on Runtime installations, any changes to the license settings won't be saved unless you provided a button to save your document. In general you only want to use the License this Computer or Generate Keys buttons when accessing with a runtime license.

Unlocking a customer computer:

When your application expires on your customer's computer, the unlock dialog will appear with your message and a key. The customer will be prompted for a code. They will need to contact you to get this code. Again, make sure your customer understands they cannot contact AzeoTech for unlock codes for your application. Since your password is used in generating the codes, and we don't know your password, we cannot generate the codes. When they contact you, they will provide you with the key displayed. This key changes every time the popup appears, so make sure your customer doesn't close the popup before the process is complete. You will then open your application (in Safe mode if you want) on your computer and go to the licensing setup popup (File - Document Licensing...) and select one of the four buttons under Generate Keys depending on what you want to have happen on the customer computer. You will be prompted for their key, and then a code will be provided which you will give to your customer who will enter it in their popup. The three options for keys with days are based on the current time on the customer computer, so 15 days will cause the customer application to expire 15 days from now. It does not extend whatever time they might have already had left.

Script access:

The expired popup will appear pretty much immediately once the expiration time has past. Since a process may be running, there is a system event that is called "OnExpiry" that is run right before the expiration popup is displayed.

Use this to safely shutdown the system. Simply create a sequence called "OnExpiry" and put your desired script in there. No matter whether the customer enters a valid unlock code or an invalid one, DAQFactory will quit once the popup is closed. If they entered a valid code, then the license is changed the next time DAQFactory loads the application.

In addition to this event, there are several functions available. They all start with AppLicense., so for example: AppLicense.GetExpiry()

GetExpiry(): returns the time the document will expire, or 0 if its licensed fully and won't expire. Use this to display warnings so your customer knows that their application will expire soon.

SetExpiry(seconds from now): sets the expiry time. This is calculated by adding the parameter passed to the current time. So, to set the expiry for 24 hours from now, you would do: AppLicense.SetExpiry(86400). Note that you cannot set the expiry date past the current expiry date. You can only make the expiration date sooner. This is to prevent someone who has the document editing password from continually extending the expiration date.

DisplayUnlockDialog(): displays the unlock popup. We strongly recommend providing a button to allow access to the unlock dialog before the application expires. You should make sure your application is in a safe state before allowing this function to be called.

DisplaySetupDialog(): displays the licensing setup popup. The password of course needs to be entered, and if it is entered incorrectly DAQFactory will quit. OnExpiry() is not called, so you should make sure your application is in a safe state before allowing this function to be called.

4 Expressions

4 Expressions

4.1 Expressions Overview

One of the powerful features of DAQFactory is expressions. Expressions are calculations that you can use in various places throughout DAQFactory, just about any place you need a number and many places you need a string. Expressions can be as simple as a constant or as complex as you need, acting on single data points or arrays of points.

When entering expressions, DAQFactory provides syntax coloring and an intelligent drop down box that displays channels, formulas and other things you may want to use in your expression so you do not have to remember everything. Any edit box within DAQFactory that displays in color and some that do not can take an expression. Expressions are used to display calculated values, to graph calculations, to set ranges, to control data logging, and in many other locations.

4.2 Using expressions

Expressions are essentially values calculated in real-time based on a formula you enter. Expressions are used for simple things, such as determining what value or trace to display in a component, or for more complex things, such as what to evaluate in a sequence 'if' statement.

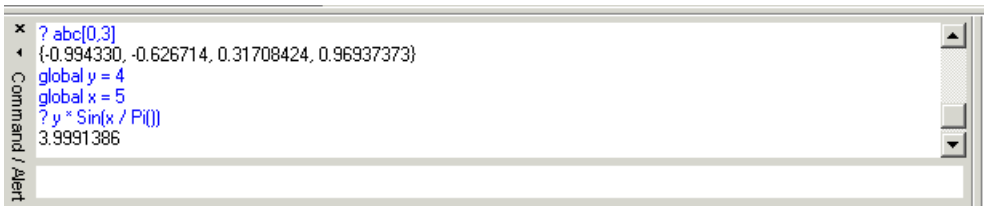
Expression edit boxes:

The expression edit boxes use syntax coloring, which colors the background of the edit box depending on the validity of the expression. By default, the background displays green for a numeric result, blue for a string result, and red for an invalid expression or one that evaluates to a null value. This is actually an easy way to tell if you can enter an expression formula in for a parameter. Only expressions use syntax coloring, so if the text you type is colored, then you can enter a formula. There are some exceptions, such as the watch, that allow expressions but do not syntax color.

Expression edit boxes have some other useful features too. As you are typing, a drop down list may appear with a list of available channels, connections, sequences, pages, functions, and other DAQFactory objects. You can either select from the list using the mouse, or keep typing and the list will track your typing. When the desired item is selected in the box, you can simply type an operator, space, or Enter key to automatically complete your selection. Once selected, a tooltip will briefly appear above your typing displaying either the quick note of the channel or the format of a function if available. If you want to see the drop down list of without typing a name, you can also hit Ctrl-Space.

Spaces are ignored in expressions, so use them as needed to make your expressions readable.

4.3 The command line interface



The Command / Alert serves two purposes. It displays DAQFactory messages known as alerts, and it allows you to enter commands and expressions. The use of the command line interface is not required to use DAQFactory, but can be useful. If you cannot see the Command / Alert window, go to View - Command/Alert in the main menu.

Most of this docking window contains a read-only area that displays DAQFactory alerts (in orange and red),

commands you have typed in (in blue) and the results of those commands if any (in black or blue depending on whether it is a number or string). At the bottom is a single line editing area where you can type in commands. Commands can be any sequence script command that runs in one line. For example:

```
global y = 4

read(MyChannel)

Execute(strMyFunc)
```

It cannot be script flow functions like **while** or **for**. When you hit Enter, the command is executed and added at the bottom of the display area. None of these commands will display anything else unless there is an error, in which case it will be displayed below the command.

If instead of a command, you wish to see the result of an expression or function call, you can use the print command, denoted by the question mark. The print command can also be used in sequence script as described later. By putting a question mark first, the result of the expression after the question mark is displayed. This means what is after the question mark must be an expression and not a command. So, you can do:

```
? MyFunction()

? 5 + 3
```

etc.

but you cannot do:

```
? global y = 4

? read(MyChannel)
```

etc.

A few points:

1) you can use the up and down arrow keys to scroll through a history of the last 100 commands you've executed.

2) if you are debugging a sequence, the command line interface runs in the scope of the sequence being debugged to allow you to view private variables. The watch works the same way. As a bonus, though, you can actually change privates from the command line interface when a sequence is being debugged. The watch only allows you to view values.

3) when you print a large array, the entire array is actually displayed in the display area. So:

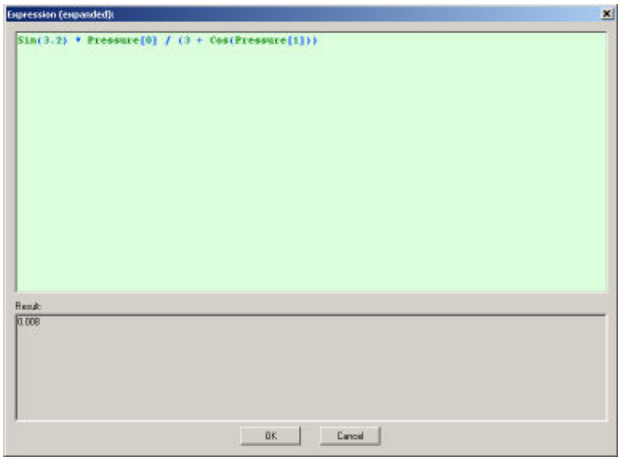
```
? MyChannel
```

will actually display the entire history of MyChannel (at least the part in memory, see Persistence for more detail on that...) up to 10,000 values in each dimension. With variable value components, the watch, and big expression window only the first 20 values in each dimension are displayed. This is actually done so you don't hang your computer trying to print a million points of history every 1/2 second as the watch refreshes.

4) you can use the F2 key to quickly put your cursor in the command line from most views, even if the command/ alert window is hidden.

4.4 The big expression window

Expressions are often entered on windows with many other parameters. Because they all have to fit on one screen, the expression boxes are often not very big. Although the entered expression will scroll, it can be difficult to read exactly what expression you have entered. Because of this, with just about any expression edit box you can expand the box and display a big expression window. To display the expanded expression window, either right click on the normal small expression edit box and select **Expand Edit Box** or hit F6 while editing the expression.



The expanded expression window contains two parts. The top is a large expression edit box. Here you should have plenty of room to enter the most complex of expressions. Feel free to hit the Enter key to split your expression into multiple lines. The linefeeds do not affect the expression. The bottom part of the window displays the result of the expression you've entered so far. This is especially handy when you are learning the syntax of expressions as it will give you immediate feedback as to the result of your expression.

When you are done with the big expression window, hit the OK button to save your changes, or Cancel to forget them. Saved changes are inserted back into the small expression edit box.

Note: For efficiency and responsiveness reasons, expressions that result in an array with more than 20 values will only display the first 20 values in the results area.

4.5 Arrays

All the data in DAQFactory is stored in memory in the form of arrays. DAQFactory supports scalars (0 dimension arrays), 1 dimensional arrays, 2 dimensional arrays, and 3 dimensional arrays. Because most of the data DAQFactory will see has a time associated with it, the arrays in DAQFactory have the ability to have a time associated with it. The time is automatically kept with the data making the data easier to manage.

Non-time associated data is stored in simple arrays. This type of data is created in DAQFactory from the results of data analysis, calculations, or manually. A manual scalar is simply a number and is entered as such anywhere in the program. A 1 dimensional array can be entered in any expression using {} notation. For example **{1,2,3,4}** will make a 4 element array containing the numbers 1, 2, 3, and 4. Only numbers and commas are allowed between the {}, but these numbers can be in hex or binary format using the appropriate delimiter described a little later. To create a 2 dimensional array, you must nest the {} notation. For example, {{1,2},{3,4}} creates a 2x2 array. {1,2} are in one row, two columns, and {3,4} are in the second row, in two columns. Three dimensional arrays follow accordingly: {{{1,2},{3,4}},{{5,6},{7,8}}} This is a 2x2x2 array.

Most of the data in DAQFactory has time associated with it. This is true for fresh data coming in and is stored in a channel. The array in a channel is called a history. A singular value read from your data acquisition device such as an A to D or Counter reading is appended along with its associated time to the 1 dimension array of values and times associated with that channel called the history. The new values are always inserted at the front of the array (giving the most recent value an index of zero). If the array grows larger than the given history length, the oldest values are removed. And so DAQFactory maintains a list of the most recent values of any given channel. With this list you can do any sort of data processing you may wish! For spectral data, which comes into DAQFactory as an 1 dimensional array of values with a single time stamp, data is appended to a 2 dimensional array, much like the history of singular values. A single row of a 2 dimensional array corresponds to a spectra. Likewise, Image data, which is 2 dimensional with a single time, is appended to a 3 dimensional array, the 3rd dimension being time. A single image takes one row of the 3 dimensional array. Within DAQFactory, time always exists along the rows or first indexed dimension.

Example: Lets assume we are reading a channel once a second.

After the first read the channel would have something like:

[0] Time 3:02:10 Data: 1.293

This is a scalar value with time.

A second later it will have:

[0] Time 3:02:11 Data: 1.382
[1] Time 3:02:10 Data: 1.293

This is an one dimensional array with two rows.

And in one more second:

[0] Time 3:02:12 Data: 1.483
[1] Time 3:02:11 Data: 1.382
[2] Time 3:02:10 Data: 1.293

Note: When doing array math, if you try and perform an operation or function on two arrays of different size when arrays of the same size are required, the longer array is truncated to the size of the shorter array before the operation is performed. If you perform an operation between an array and a scalar value, the operator will be applied to all the elements of the array. For example, the expression: `{5,3,2} * 2` will return: {10,6,4}, but `{5,3,2} * {2,3}`, will return {10,9}.

Note: Many properties that use expressions require a one dimensional value. If the result of the expression is an array, then the array is converted to a single dimensional value. If the array is one dimensional, then the most recent value is used. If the array is two or three dimensional, then the mean of the most recent row is used. It is recommended, however, that you explicitly perform this conversion in your expression using subsetting to avoid confusion.

4.6 Subsetting

If you specify a channel name in an expression, the entire channel history array will be returned. In some cases this may be exactly what you want, especially in graphs. In some cases, you will just want the most recent data point or some subset of your data. To get a subset of your data, put the desired range of points in brackets after the channel name. For example, **`ChannelName[0,4]`** will return the last five data points. You can get a single point as well: **`ChannelName[0]`** will return the most recent data point. Subsetting can actually occur at any point, for example: **`Sin(chan * otherchan[0])[0,49]`** will return the first 50 data points of the sine of the entire history of chan multiplied by the most recent value of otherchan.

NOTE: A common error is to use () instead of [] for subsetting. Although this works in some cases, it does not work in all, nor do we guarantee that it will continue to work in newer versions.

NOTE: Whenever possible, put the subset notation immediately after the channel or variable name. This allows DAQFactory to optimize the data retrieval. Otherwise, DAQFactory is forced to copy the entire channel history or variable array before subsetting. So, while you could do Sin(chan * otherchan[0])[0,49] it is much better to do sin (chan[0,49] * otherchan[0]). There are of course cases where you can't do this, which is why DAQFactory supports subsetting pretty much anywhere. Subsetting immediately after the channel name is especially important when channel persistence is used.

Advanced Subsetting

Some advanced subsetting is possible. For example you can specify time ranges in your subsetting, and instead of using the data point number, DAQFactory will use the times of your data points to determine which points get returned.

`chan[5h,6h]` will return all the values between 5 and 6 o'clock today. Of course this only works if the array has time associated with it, so **`({1,2,3})[5h,6h]`** won't work, though **`({1,2,3})[0,1]`** will return **`{1,2}`**

For even more advanced subsetting, you can put calculations in your subsets.

`channel[Abs(otherchannel[0])]`. There are many creative uses for this feature.

For two and three dimensional arrays you can subset on multiple dimensions by repeating the [] notation.

`channel[3][2][1]` returns the data point in the 4th row, 3rd column, 2nd depth.

By not specifying a dimension, the entire dimension is returned.

For example, given channel is 3 dimensional:

channel[3][2] returns a 1 dimensional array of data points corresponding to the 4th row, 3rd column.

channel[][2][1] returns a 1 dimensional array of data points corresponding to the 3rd column, 2nd depth.

You can also specify ranges:

channel[0,2][4][2] returns 3 values on the first three rows, 5th column, 3rd depth.

The first specifier is always rows. Rows always corresponds to time, and therefore you can only subset in time along rows. So, **channel[0][5h][2]** is not valid, but **channel[5h][2][4]** is.

4.7 Strings

The DAQFactory expression parser can also handle strings. This can be used to parse or calculate the string I/O types, or for other uses. String channels and variables are used just like regular channels and support subsetting and other expression features. You have to remember that the value is a string however, and must be converted before using math functions. For example: **"89" * 3** is invalid, but **StrToDouble("89") * 3** is valid. String constants can be entered by enclosing the desired string in quotes (single or double quotes). String constant arrays are created just like numeric arrays: **{"string a","string b"}**. You can concatenate two strings using the add operator to make one string: "**string a" + "string b"**. There are many other functions available for string processing. Please see the section on String Functions for the complete list.

Note: While DAQFactory supports the storage of two and three dimensional string arrays, most of the DAQFactory functions do not work with string arrays with more than one dimension.

4.8 Entering hex and binary constants

If it is more convenient to enter constants in expressions as hexadecimal or binary you can do so in DAQFactory using the following notation:

For hexadecimal, proceed the number with **0x**. For example **0x1f** is the same as decimal 31.

For binary, proceed the number with **0b**. For example **0b10011** is the same as decimal 19.

Note: The x and b must be in lowercase and preceded by a zero.

Note: Only integer values between 0 and 2^31 are supported. To do larger values, split your number into words and add them.

4.9 Entering in constant time values

Time is an important data element. Time is stored throughout DAQFactory as the number of seconds (with decimals) since January 1st, 1970. This is one of many standard ways of storing time, and provides the necessary precision (down to the microsecond) for anything DAQFactory may do. This number is also very easy for a computer to deal with. However, this number is somewhat meaningless to us humans. There are many times when an absolute date and time would be nice to use in an expression, and DAQFactory provides a simple way of entering these values without having to calculate the number of seconds since 1970, and in a non-location specific form. The form is:

xxYxxMxxD xxHxxMxx.xxxS

The YMDHMS are not case sensitive and are actually easier to read in lowercase when real numbers are used. The x's correspond to the desired year, month, day, hour, minute or second. Decimal seconds are completely valid. The S and the space are also completely optional. Since you probably don't want to have to always fully specify your date, you can skip some parameters and let DAQFactory fill them in for you. For example if you do not enter in the year, the current year will be used. Likewise for month, day and hour. If you don't specify minutes or seconds, 0 will be used. For example, given today is October 1, 2001, 1:31:23.345pm

Fully qualified time: **01y10m01d 13h31m23.345**

No date specified: **13h31m23.345**

No year specified: **10m01d 13h31m23.345**

No hour specified: **31m23.345**

No minute or seconds specified: **13h** this will actually give October 1,2001, 1:00:00.000pm

No seconds specified: **13h31m** this will give October 1,2001 1:31:00.000pm

Just seconds specified: **365s** this will give October 1,2001 1:06:05.000pm. Note that the s is required here, otherwise DAQFactory will just think you mean the number 365.

4.10 Object variables and functions

A DAQFactory document is made up of many different objects: channels, sequences, components, pages, connections as well as internal objects like variables, email, file and system. Most of these objects have variables that can be read or set, and functions that can be called to perform different tasks.

The most common object variable is a channel. By simply specifying a channel name, the current history of that channel is retrieved. But channels have functions as well, such as ClearHistory. To call the ClearHistory function of a particular channel, simply put a period **(.)** after the name of the channel, then enter **ClearHistory():**

```
MyChannel.ClearHistory()
```

Channels, however are actually objects contained in a connection. Because of the default connection the connection name does not have to be specified, but this:

```
Local.MyChannel.ClearHistory()
```

is also completely valid. Here we've specified the connection ("Local"), then put a period, then selected an object in Local ("MyChannel"), then a period, then a function of MyChannel ("ClearHistory()").

Most of the details of an objects variables and functions are described in the sections on the particular object.

If you are going to call one object's functions and access its variables often and don't want to have to type in the full name, you can use the using() function to put the object's functions and variables in the global namespace. This will make it accessible without the prefixes. The best use for this is when you are working with a particular device and need to call its functions. So instead of:

```
Device.LabJackUD.AddRequest(...)
Device.LabJackUD.AddRequest(...)
Device.LabJackUD.GoOne(1)
Device.LabJackUD.GetResult(...)
Device.LabJackUD.GetResult(...)
```

you could do:

```
using("device.labjack")
AddRequest(...)
AddRequest(...)
GoOne(1)
GetResult(...)
GetResult(...)
```

Once you call using() with a prefix like above, it is added to a list and kept in the list as long as the document remains loaded. You do not have to keep recalling this function and can simply put this in an auto-start sequence. You can call using() multiple times with different prefixes to bring other functions and variables into the global namespace. Just be careful as there may be collisions with existing function and variable names.

Note: if you do using("device.labjack"), DAQFactory will bring all the functions from both the LabJackUD device and the LabJackM device into the global namespace. All LabjackM function start with "LJM_", so are easy to identify.

4.11 Specifying Connections in Expressions

Because DAQFactory can have multiple connections, and two different connections could have some of the same object names, you must specify the connection name for the desired object as well. The format is **ConnectionName.ObjectName**. You can specify one connection to be your default connection. When you create a new document, this is the **Local** connection. You do not need to specify the connection name in front of the object name if the object is on the default connection.

To change the default connection:

1. In the workspace, click on the name of the connection you would like to make the default.

   This will display the connection view for that connection.

2. Check the box next to **Default?**.

   You do not need to hit apply, the change is immediate.

Remember, **V**, even though it only has channels and no other objects, is also considered a connection.

Application: In addition to making it easier to enter object names in expressions, this method also makes it so you can toggle a display between different connections simply by changing the default connection. For example, if you had several different locations taking data from the same sensors with the same channel names, and you were in a central location connected to both remote computers, you could create a single set of screens referencing the channels by name without specifying the connection, and then simply switch the default connection between the two remote computers and all the screens will update with the appropriate data.

4.12 Expression Operator / Function Reference

4.12.1 Order of operations

All expressions are evaluated from left to right, using standard notation. The order of operations, from lowest priority to highest is:

,

&& || | & #

< > <= =>

!= == << >>

+ -

* / %

^

! ~ ~~ ~~~ any function

(

This means that 1 + 2 * 3 equals the same thing as 3 * 2 + 1.

The meaning of these operators is described in the next few sections.

4.12.2 Boolean operators

Expressions support the following Boolean operators (which return either 0 or 1):

&&, || And, Or

<, >	Less Than, Greater Than
<=, >=	Less Than Equal, Greater Than Equal
==, !=	Equal (that's two equal signs), Not Equal
!	Not
iif()	Inline if

When evaluating Boolean, any non-zero value is considered true. Therefore **!5** equals 0. Like the math operators, the Boolean operators can be applied to arrays, so **{2,5,0,2} >= {1,6,3,1}** will return **{1,0,0,1}**.

It is recommended when combining Boolean operators to use parentheses liberally. For example:

```
if ((MyChannel[0] > 3) && (MyOtherChannel[0] < 2))
```

This will ensure you get the expression you expect and also makes it easier to read.

The inline if, or iif() function is a function, not an operator, but this seems an appropriate place to put it. The function takes three parameters. The first is an expression. If the expression evaluates to a non-zero value, the function returns the second parameter. Otherwise it returns the third parameter. So:

iif(5 < 10, "less than", "greater than") will return **"greater than"** while

iif(15 < 10, "less than", "greater than") will return **"less than"**

4.12.3 Math operators

Expressions support the standard mathematical operators:

+, -, *, /:	Add, Subtract, Multiply, Divide
%, ^	Modulus, Power
()	Parenthesis

These functions work both on scalar values and arrays of values.

Examples:

3 + 2 returns **5**

{3,5} + 2 returns **{5,7}** 2 is added to both elements of the array

{3,5} + {2,1} returns **{5,6}** each element of the array is added to the matching element of the other

{3,5,6} + {2,1} returns **{5,6}** the longer array is truncated to the size of the shorter array before adding

8 % 3 returns **2** 2 is the remainder of 8/3.

2 ^ 4 returns **16** 2 raised to the 4th power is 16.

2 ^ 0.5 returns **1.414** 2 raised to the 0.5 power is the same as the square root of 2.

4.12.4 Bitwise operators and functions

Expressions support the following bitwise operators:

&, |, #	Bitwise AND, Bitwise OR, Bitwise XOR
~	Bitwise NOT (8 bit)
~~	Bitwise NOT (16 bit)
~~~	Bitwise NOT (32 bit)
<<, >>	Shift Left, Shift Right
TestBit(x, #)	Returns 0 or 1 status of bit # of x (0 indexed)
SetBit(x, #)	Sets bit # of x to 1 (0 indexed)
ClearBit(x, #)	Clears bit # of x to 0 (0 indexed)

When evaluating bitwise operators and functions, all non-integer values are rounded down. For some advanced bitwise conversion, please see the section on byte conversions.

Examples: given using binary notation:

0b0100110 & 0b0101001 returns **0b0100000**

0b0100110 | 0b0101001 returns **0b0101111**

~0b0100110 returns **0b11011001** Note how the input value is filled out to 8 bits before performing the NOT.

~~0b0100110 returns **0b11111111 11011001** Note how the input value is filled out to 16 bits before performing the NOT.

0b0100110 << 1 returns **0b1001100**

0b0100110 >> 1 returns **0b0010011**

TestBit(0b0100110, 2) returns **1**

SetBit(0b0100110, 3) returns **0b0101110**

ClearBit(0b0100110, 2) returns **0b0100010**

The bitwise operators, but not the Test, Set, ClearBit functions, can be applied to arrays:

{0b1010, 0b0111} << 2 returns **{0b10100,0b1110}**

4.12.5 Math functions

Many mathematical functions are supported:

Abs (x)	Absolute Value

Ceil (x) First integer above the current value (round up)

Floor(x) First integer below the current value (round down)

Exp (x) Exponential

Ln (x) Natural Log

Log (x) Log base 10

Examples:

`Abs(-1)` returns `1`

`Abs(1)` returns `1`

`Ceil(5.3)` returns `6`

`Ceil(5)` returns `5`

`Ceil(-1.3)` returns `-1`

`Floor(5.3)` returns `5`

`Floor(5)` returns `5`

`Floor(-1.3)` returns `-2`

`Exp(3)` returns `20.86`

`Log(3)` returns `0.477`

`Ln(3)` returns `1.099`

Note that you can pass arrays into these functions as well:

`Ceil({5.3,5,-1.3})` returns `{6,5,-1}`

4.12.6 Trigometric functions

The standard trig functions are supported:

Normal:

Sin(x)
Cos(x)
Tan(x)

Inverse:

ASin(x)
ACos(x)
ATan(x)

Hyperbolic:

SinH(x)
CosH(x)
TanH(x)

X is assumed to be in Radians. To convert degrees to radians, just multiply by Pi()/180: `Sin(30 * Pi()/180)` returns `0.500`

As with most functions, you can pass scalar or array values to any of these functions.

Examples:

`Sin(1)` returns `0.841`

`Sin({1,2})` returns `{0.841,0.909}`

4.12.7 Statistical functions

Data analysis requires statistical functions:

Mean(Array) provides the mean of the given array along the rows dimension. Examples:
`Mean({1,2,3})` returns `2`
`Mean({{1,2,3},{4,5,6})` returns `{2.5,3.5,4.5}`

Sum(Array) provides the sum of the given array along the rows dimension. Examples:
`Sum({1,2,3})` returns `6`
`Sum({{1,2,3},{4,5,6})` returns `{5,7,9}`

Min(Array) provides the minimum value of the given array along the rows dimension. Examples:
`Min({1,2,3})` returns `1`
`Min({{1,2,7},{4,5,6})` returns `{1,2,6}`

Max(Array) provides the maximum value of the given array along the rows dimension. Examples:
`Max({1,2,3})` returns `2`
`Max({{1,7,3},{2,5,3})` returns `{2,7,3}`

Variance(Array) provides the variance of the given array along the rows dimension. Examples:
`Variance({1,2,4})` returns `2.333`
`Variance({{1,2,4},{4,5,6})` returns `{4.5,4.5,2}`

StdDev(Array) provides the standard deviation of the given array along the rows dimension. Examples:
`StdDev({1,2,4})` returns `1.528`
`StdDev({{1,2,4},{4,5,6})` returns `{2.121,2.121,1.414}`

All of the above functions have equivalents for other array dimensions:

MeanCols()
MeanDepth()
MaxCols()
MaxDepth()
MinCols()
MinDepth()

SumCols()
SumDepth()
VarianceCols()
VarianceDepth()
StdDevCols()
StdDevDepth()

Examples:
`Mean({{1,2,3},{4,5,6})` returns `{2.5,3.5,4.5}`
2.5 is the mean of 1 and 4
3.5 is the mean of 2 and 5
4.5 is the mean of 3 and 6

`MeanCols({{1,2,3},{4,5,6})` returns `{2,5}`
2 is the mean of 1, 2 and 3
5 is the mean of 4, 5 and 6

4.12.8 Boxcar averaging and smoothing functions

Boxcar functions:

Several boxcar averaging functions are available:

Boxcar(Value, Boxcar Size)
BoxcarSD(Value, Boxcar Size)
BoxcarMax(Value, Boxcar Size)
BoxcarMin(Value, Boxcar Size)
BoxcarSum(Value, Boxcar Size)

All do essentially the same thing. They take a subset of the data in Boxcar Size sized chunks from value starting at index 0, and apply the appropriate function: Boxcar returns the mean of each of these chunks, SD the standard dev, etc. The returned array has a size equal to the size of the original array divided by the boxcar size rounded up. If the array size of value is not a multiple of boxcar size, then the last boxcar will have less then boxcar size elements used in its calculation.

Example:

`Boxcar({1,4,2,6,7},2)` returns `{2.5,4,7}`

2.5 is the average of 1 and 4

4 is the average of 2 and 6

7 is the average of 7.

For arrays with time, the result of the boxcar has the time of the last point in each car stored. For BoxcarMax and BoxcarMin the time of the point in each car that is the max or min is stored instead. This is very useful for plotting.

For example, assuming the MyChannel contained {1,4,2,6,7} with time associated with each data point then:

`GetTime(BoxcarMin({1,4,2,6,7},2))` would return a three element array with the time of the 1, 2 and 7 data points.

Smoothing:

Smooth(Value, Amount):

Performs a running average. The returned array is the same size as the value array. Each element contains the average of the element and the next Amount-1 elements, unless the end of the array is reached. As the end is approached and amount elements are not available, only the available elements are used in the average. This results in a bit of roughness near the end of the array.

Example:

`Smooth({1,4,2,6,7},2)` returns `{2.5,3,4,6.5,7}`

2.5 is the average of 1 and 4.

3 is the average of 4 and 2.

4 is the average of 2 and 6.

6.5 is the average of 6 and 7.

7 is the average of 7.

4.12.9 String functions

DAQFactory supports many functions for string manipulation.

Asc(Character):

Converts the given character into its ASCII value. If you pass a string, only the first character is converted. For example:

`Asc({“ABC”,"DEF"})` returns `{65,68}`

AscA(String):

Same as Asc, but works on a single string and returns the ASCII value for each character in that string in an array. Only works with the first string. For example:

`AscA({“ABC”,”DEF”})` returns `{65,66,67}`

Chr(Value):

Converts the given value into the ASCII representation of that value. For example:

`Chr({65,66,67})` returns `{“A”,"B","C"}`

ChrA(Array of Values):

Same as Chr, but takes an array of values and creates a single string out of them. For example:

`ChrA({65,66,67})` returns `“ABC”`

Compare(String1,String2):

Compares two strings, case sensitive. Returns 0 if they match, -1 if String1 is less than String2, +1 if String1 is greater than String2. In most cases, it is easier to use the ==, < and > operators. Examples of Compare():

`Compare("abcdef","abcdef")` returns `0` but

`Compare("abcdef","abCDef")` returns `1`.

CompareNoCase(String1,String2):

Compares two strings case insensitive. Returns 0 if they match, -1 if String1 is less than String2, +1 if String1 is greater than String2. For example:

`CompareNoCase("abcdef","abCDef")` returns `1` but

`CompareNoCase("abcdef","bdfsef")` returns `-1`

Delete(String,Index,# Chars):

Deletes x number of chars starting at index in string. For example:

`Delete("abcdef",3,2)` returns `"abcf"`

DoubleToStr(Value):

This converts the given number to a string with up to 15 sig figs (the max for a double precision floating point value). This is essentially the same as Format("%f",Value)

Evaluate(String):

Evaluates the string as if it was an expression. Returns a value. For example:

`Evaluate("1+1")` returns `2`.

Find(String,String to find,Start point):

Finds the given string within the other given string starting at the given start point. Returns the index of the given search string or -1 if not found. For example:

`Find("abcdef","cd",0)` returns `2`.

FindExpr(String, Regular Expression):

Finds the first instance that matches the regular expression. Regular expressions are a somewhat standard method of specifying more complicated search strings. They are a bit more complicated, but rather powerful when used properly. DAQFactory follows pretty closely to the regular expressions used by Microsoft Visual C++ 6.0. The details about using regular expressions can found online. Some beginning information is provided at the end of this section. The result of this function is an array with two values. The first value is the first character that matches the expression. The second character is the length of the matching string. {-1,-1} is returned if nothing is found.

`FindExpr("abcdef","[^a]cd")` returns `{1,3}`.

Unlike most regular expression search engines, FindExpr can search for the ASCII character 0. So:

`FindExpr("abc" + chr(0) + "def", "[^a]c"+Chr(0)+"d")` returns `{1,4}`

FindOneOf(String,Chars to find):

Finds the first instance of one of the given chars. -1 if not found. Returns an index. For example:

`FindOneOf("abcdef","ce",0)` returns `2`.

Format(FormatString,Value,[Value2],...):

Performs a standard C printf style format. You can have up to 19 values in a single call. To format a time value, see the FormatDateTime() function in the section on Time. The following types are supported (case sensitive):

c or C: single byte character

d or i: signed decimal integer

o : unsigned octal integer

u : unsigned decimal integer

x : unsigned hexadecimal integer, using lowercase "abcdef"

X : unsigned hexadecimal integer, using uppercase "ABCDEF"

e : signed floating point value using exponential notation with a lower case "e"

E : same as e but with an upper case "E"

f : signed floating point value without exponent.

g : signed floating point value with optional exponent depending on size of resultant string. In other words, 3.0 would be simply 3, but 3 million would be 3e+006

G : same as g but with an upper case "E" for the exponent.

s or S : a string

For the details of using the flags, width and precision arguments in printf, we refer you to any C programming text.

For example:

`Format("My Value: %.3f",MyChannel[0])` returns `"My Value 34392.492"`

You can also use several \ modifiers to generate special characters:

\n : new line character, a line feed, ASCII 10

\r : return character, a carriage return, ASCII 13

\t : tab character, ASCII 9

\yyy: the ASCII character with the given code. All three digits are required. So, \065 will generate an A.

\xzz: same as above, but the code is given in hex. The x is required along with two hexadecimal digits. So, \x0a is the same as \n or ASCII 10.

\\ : a backslash

GetLength(String):

Returns the length of the string. For example:

`GetLength("abcdef")` returns `6`

HideHidden(String)

Takes all non-printing characters (ASCII values less than 32 or greater than 126) and removes them from the string. For example:

`HideHidden("abc" + Chr(9) + "def" + Chr(13))` returns `"abcdef"`

Insert(String,Index,Insert String):

Inserts the given string into the other given string starting at index. For example:

`Insert("abcdef",3,"abc")` returns `"abcabcdef"`

Left(String,# Chars):

Returns x number of characters starting from the left of the string. For example:

`Left("abcdef",4)` returns `"abcd"`

LTrim(String), RTrim(String):

Trims leading or trailing spaces. For example:

`LTrim("  abcdef")` returns `"abcdef"`

MakeLower(String):

Returns the string converted to all lowercase. For example:

`MakeLower("Abcdef")` returns `"abcdef"`

MakeReverse(String):

Flips the string. For example:

`Reverse("abcdef")` **returns** `"fedcba"`

MakeUpper(String):

Returns the string converted to all uppercase. For example:

`MakeUpper("abcdef")` returns `"ABCDEF"`

Mid(String,Start,# Chars):

Returns a subset from the center of the string. Start is zero indexed. For example:

Mid("abcdef",2,3) returns **"cde"**

Parse(String,Index,Parse By):

Assumes String is a list of values separated by Parse By. Returns the given index item into this list. For example:

Parse("this,is,a,test",2,",") returns **"a"**. Note 0 indexing (as always)

If you specify a negative Index, then parse will completely parse the String and return an array of the separated strings. For example:

Parse("this,is,a,test",-1,",") returns **{"this","is","a","test"}**

If you only need one element out of your delimited string, use the first, indexed method. If, however, you are going to work with all the elements, it is much faster to use the second method and work through the array then to repetitively call Parse() with incrementing indices.

Remove(String,Remove Char):

Removes all instances of char in string. For example:

Remove("abcdef","a") returns **"bcdef"**

Replace(String,Old,New):

Replaces all instances of old with new within string. For example:

Replace("abcabc","a","b") returns **"bbcbbc"**

ReverseFind(String,Char to find):

Just like find, but starts from the back. Returns an index (from the front). For example:

ReverseFind("abcdef","cd") returns **2**.

Right(String,# Chars):

Same as Left() but from the right. For Example:

Right("abcdef",4) returns **"cdef"**

ShowHidden(String)

Takes all non-printing characters (ASCII values less than 32 or greater than 126) and displays their ASCII code with a backslash in front. For example:

ShowHidden("abc" + Chr(9) + "def" + Chr(13)) returns **"abc\009def\013"**

StrToDouble(String):

Converts a string into a double for use in standard math functions. Converts up to the first invalid character. Returns a value. If the first character is invalid, it will return NaN. For example:

StrToDouble("534.27") returns **534.27**.

StrToDouble("534sd27") returns **534**.

StrToDouble("sd27") returns **NaN**.

You can also specify hex or binary values by proceeding the number with "0x" or "0b":

StrToDouble("0xff") returns **255**.

Although the function is called StrToDouble, if you use the hex or binary notation an integer is returned. Also, as when entering hex and binary constants, only the values between 0 and 2^31 are supported.

To convert date and time strings to DAQFactory times, use the StrToTime() function shown in the section on Time.

Supported operands:

= (same as compare and is case sensitive)

+ (concatenates the strings)

>

<

>=

<=

Regular Expression Summary (advanced topic for FindExpr()):

Regular expressions are made up of regular ASCII characters (including control codes generated with functions like Chr()) and tags. A tag is that part of a regular expression which consists of any of the symbols, or characters, listed in the table below. For example, the regular expression "s+.{2,4}" has two (2) tags: the "s+" which specifies that the "s" will be searched for one or more times; and ".{2,4}", which specifies that there must be two (2) to four (4) characters (except newline characters) following the occurrence of the "s". In essence tags allow you to break up a rule into separate search components.

\	Marks the next character as special. Precede any special character that you would actually like to search for with this symbol. For example, to search for '^' you would put '\^'.
^	A match occurs only if the character following this symbol is found at the beginning of a line or input.
$	A match occurs only if the character following this symbol is found at the end of the input, or line.
*	Searches for the preceding character zero or more times. For this implementation you must define more than one character. If only one character is specified in the regular expression then no matches will be found. That means that /zo*/ matches z and zoo, but /z*/ will match nothing because only one character has been specified.
+	Searches for the preceding character one or more times.
?	Searches for the preceding character either one time or not at all. It cannot be defined if only one character is specified in the regular expression.
.	Matches any single character except Chr(10).
(pattern)	Matches the specified pattern and remembers each match via unique indexes for each match found. Only characters enclosed within parentheses will be considered patterns. The found substring can then be retrieved by using '\0'-'\9' in the regular expression, where '0'-'9' is the identifying index number of that particular pattern. For example: '(re).*\0s+ion' will match 'regular expression' because it first finds the pattern 're' and remembers the pattern with an index of 0. '.*' will then match 'gular exp' in 'regular expression'. Then we look for the same pattern again by inserting '\0' into the regular expresson, which matches the second occurrence of 're' in 'regular expression'. Finally, 's+ion' matches 'ssion'.
x|y	Matches either character 'x' or 'y'. You can combine more than two characters (e.g. 'x|y|z').
{n}	The preceding character must match exactly 'n' times (non-negative values only).
{n,}	The preceding character must match at least 'n' times (non-negative values only).
{n,m}	The preceding character must match at least 'n' times and at most 'm' times. (n,m - non-negative numbers only).

[xyz]	A character set. A match occurs if any one of the enclosed characters is found.
[^xyz]	A non-matching character set. A match occurs if any character that is not in the set is found.
\b	Matches a word boundary. A boundary occurs between two non-space characters. Also, the ascii format characters, Chr(10) through Chr(13) do not define a word boundary. For example, the expression "me" + chr(10) only has one word boundary which occurs between the "m" and the "e".
\B	Searches for a non-word boundary. The exact opposite of the previous symbol ("\b"). A match occurs for any boundary between space characters or between a non-space character and a space character. For example, the expression " me"+chr(10)+" " has three (3) non-word boundaries: between the first space and the "m"; between the "e" and the newline character; and between the newline character and the last space.
\d	A match occurs for any digit 0-9.
\D	Matches any non-digit character.
\f	Matches a form feed (same as chr(12))
\n	Matches a new-line character. (same as chr(10))
\r	Matches a carriage return character. (same as chr(13))
\s	Matches any white space character.
\S	Matches any non-white space character.
\t	Matches a tab character. (same as chr(9))
\v	Matches any vertical tab character. (same as chr(11))
\w	Matches any alphabetic character including underscores. [A-Z a-z 0-9 _].
\W	Matches any non-alphabetic character.
\num	Matches any characters defined as a pattern with a unique index between 0 and 9. A match occurs if the pattern identified by 'num' is found (see the pattern description for an example).
/n/	A match occurs if a character is located with an ascii code of 'n'. 'n' must be between 1 and 255. You'll typically just want to use Chr() in creating your regular expression instead: FindExpr(MyVar, "abc" + chr(0) + "def")

4.12.10 Byte conversions

Byte arrays to values with the 'To' functions:

When retrieving data from instrumentation, especially serial devices, the data often arrives as an array of bytes.

These bytes however, represent numbers in different forms. For example, the two bytes, 10 and 15 could represent two 8 bit numbers, or they could represent a single 16 bit number. Converting these two numbers is relatively straight forward using bitwise operators, but gets much more complicated when the bytes represent floating point values. To help with either of these cases, DAQFactory offers a collection of functions for converting an array of bytes into a single value. All these functions start with "To.". Each function takes an array of values. The size of the resultant type determines the number of columns required in the array. You can have one or more rows to convert multiple values. The functions return an array of values with one column. For example, if you did **`To.Word ({{10,20},{30,40}})`** you'd get **`{5130,10270}.`**

Byte(): converts the given number into a signed byte (8 bit) sized value.

`To.Byte(253)` returns **`-3`**

uByte(): converts the given number into an unsigned byte sized value. This is only marginally faster than just doing % 256, but possibly a bit clearer.

`To.uByte(259)` returns **`3`**

Word(): converts the given number into a signed word (16 bit) value.

`To.Word({{0,128}})` returns **`-32768`**

uWord(): converts the given number into an unsigned word value.

`To.uWord({{0,128}})` returns **`32768`**

rWord(): converts the given number into a signed word value by reversing the bytes to Big Endian (most significant byte first)

`To.rWord({{0,128}})` returns **`128`**

urWord(): converts the given number into a unsigned word value by reversing the bytes to Big Endian (most significant byte first)

`To.urWord({{0,128}})` returns **`128`**

Long(): converts the given number into a signed double word (32 bit) value.

`To.Long({{0,128,0,128}})` returns **`-2147450880`**

uLong(): converts the given number into an unsigned double word value.

`To.uLong({{0,128,0,128}})` returns **`2147516416`**

rbLong(): converts the given number into a signed double word value by reversing the bytes in each word to Big Endian (most significant byte first). The words are left as Little Endian (least significant word first).

`To.rbLong({{0,128,0,128}})` returns **`8388736`**

urbLong(): converts the given number into a unsigned double word value by reversing the bytes in each word to Big Endian (most significant byte first). The words are left as Little Endian (least significant word first).

`To.urbLong({{0,128,0,128}})` returns **`8388736`**

rwLong(): converts the given number into a signed double word value by reversing the words to Big Endian (most significant word first). Bytes in each word are left as Little Endian (least significant byte first).

`To.rwLong({{0,128,0,128}})` returns **`-2147450880`**

urwLong(): converts the given number into a unsigned double word value by reversing the words to Big Endian (most significant word first). Bytes in each word are left as Little Endian (least significant byte first).

`To.urwLong({{0,128,0,128}})` returns **`2147516416`**

Float(): converts the given number into a 32 bit floating point value (IEEE format).

`To.Float({{3,4,66,69}})` returns **`3104.251`**

rwFloat(): converts the given number into a 32 bit floating point value, reversing the words

`To.rwFloat({{3,69,66,69}})` returns `2100.329`

rbFloat(): converts the given number into a 32 bit floating point value, reversing the bytes

Double(): converts the given number into a 64 bit floating point value (IEEE format).

`To.Double({{3,69,66,69,3,4,28,65}})` returns `459008.818`

Converting to bits:

In addition to the above To functions there is one additional function for converting an integer into its bit array:

Bit(): converts the given number to an array of bits. If you provide a single number to this function, it will return a 32 element array of the bits that make up that number. If you provide an array of numbers, a 2 dimensional array is returned. In both cases, the 32 elements are in the second dimension, with the least significant bit first. So:

`To.Bit(3)` returns `{{1,1,0,0,0,0,0,0,....,0}}`

`To.Bit({3,4})` returns `{{1,1,0,0,0,0...},{0,0,1,0,0,0,0,....}}`

The above results are truncated for clarity, but in fact there will be 32 columns in each result. The default is to convert to 32 bits. If you want less bits, specify it as the second parameter of the function:

`To.Bit({3,4},8)` returns `{{1,1,0,0,0,0,0,0},{0,0,1,0,0,0,0,0}}`

Either way, you can then access individual bits using standard subsetting:

```
Private bitarray = To.Bit(MyChannel[0])
if (bitarray[0][4])
  .. do something
endif
bitarray[0][6] = 1  // set a bit
Out = From.Bit(bitarray)
```

Notice that we have to subset in the second dimension, not the first, to get a particular bit. If you are only converting a single value and want to avoid this you could always use the Transpose function:

```
Private bitarray = Transpose(To.Bit(MyChannel[0]),0)
if (bitarray[4])
  .. do something
endif
```

At the end of the first example, we set a single bit and then used the From.Bit() function described below to convert the bit array back into its numeric representation. Note that if you do this in the second example after transposing, you will need to transpose back before calling From.Bit().

HexString(array): converts an array of numbers (each from 0 - 255) into a hex string. The hex string is formatted for use in setting blob data in SQL statements.

`To.HexString({1,2,3})` returns the string `010203`

To.HexString supports multicolumn data allowing you to convert numbers larger than byte size into a hex string. You have to convert the numbers into byte arrays (using the appropriate from. function), but even though the result of this will be a two dimensional array of bytes, To.HexString will convert this into a single hexstring. For example:

`To.HexString(from.urWord({65535,32767,3}))` returns the string `FFFF7FFF0003`

Values to a byte array with the 'From' functions:

Performing the direct opposite function as the To functions, the From functions allow you to convert a single value into its byte pattern based on a particular format. For example, the number 515 could be represented in a two byte word as 3 and 2 or a floating point value as the 4 bytes, 0, 0, 23 and 67. Each From functions take a value and return an array of bytes. All the functions start with "From.". Like the To functions, you can pass an array of values to convert multiple values at once. The resultant array will have the same number of rows as the original array, but as many columns as the size you are converting from. So, `From.Word({5130,10270})` will give you `{{10,20},`

{30,40}. Note that these functions are designed for you to pass in values that are within the range of the given type you will get . For example, the range of a signed byte value is -127 to 127.

Byte(): converts the given signed byte value into an unsigned byte format.

From.Byte(-3) returns **253**

uByte(): converts the given unsigned byte value into an unsigned byte format. This actually does the exact same thing as Byte() and is a pretty useless function, but we've provided it when you want clarity in your code.

From.uByte(3) returns **3**

Word(): converts the given signed word value into an array of unsigned bytes putting the least significant byte first.

From.Word(-32768) returns **{{0,128}}**

uWord(): converts the given unsigned signed word value into an array of unsigned bytes putting the least significant byte first.

From.uWord(32768) returns **{{0,128}}**

rWord(): converts the given signed word value into an array of unsigned bytes putting the most significant byte first.

From.rWord(128) returns **{{0,128}}**

urWord(): converts the given unsigned word value into an array of unsigned bytes putting the most significant byte first.

From.urWord(128) returns **{{0,128}}**

Long(): converts the given signed double word value into an array of unsigned bytes with the least significant byte first.

From.Long(-2147450880) returns **{{0,128,0,128}}**

uLong(): converts the given unsigned double word value into an array of unsigned bytes with the least significant byte first.

From.uLong(2147516413) returns **{{0,128,0,128}}**

rbLong(): converts the given signed double word value into an array of unsigned bytes with the bytes in each word reversed from standard Little Endian format of From.Long().

From.rbLong(-2147450880) returns **{{128,0,128,0}}**

urbLong(): converts the given unsigned double word value into an array of unsigned bytes with the bytes in each word reversed from standard Little Endian format of From.uLong().

From.urbLong(2147516413) returns **{{128,0,128,0}}**

rwLong(): converts the given signed double word value into an array of unsigned bytes with the words reversed from standard Little Endian format of From.Long().

From.rwLong(-2147450880) returns **{{0,128,0,128}}**

urwLong(): converts the given signed double word value into an array of unsigned bytes with the words reversed from standard Little Endian format of From.uLong().

From.urwLong(2147517413) returns **{{0,128,0,128}}**

Float(): converts the given IEEE floating point value into an array of unsigned bytes.

From.Float(3104.251) returns **{{4,4,66,69}}**

rwFloat(): converts the given floating point value into an array of unsigned bytes with the words reversed from

IEEE format.

From.rwFloat(3104.251) returns **{{66,69,4,4}}**

rbFloat(): converts the given floating point value into an array of unsigned bytes with the bytes reversed from IEEE format.

Double(): converts the given IEEE double precision (64 bit) floating point value into an array of unsigned bytes.

From.Double(459008.818) returns **{{193,202,161,69,3,4,28,65}}**

Converting from bits:

In addition to the above From functions there is one additional function for converting an array of bits to integers:

Bit(): converts the given array of bits to its numeric representation. Unlike To.Bit(), you do not have to provide all 32 bits. DAQFactory will assume any bits not provided are 0. The least significant bit is always first in the array. Here are some examples:

From.Bit({{1,1,0,1}}) returns **11**. Note how the array of bits is in the second dimension.

From.Bit({1,1,0,1},{0,1,1,0}}) returns **{11,6}**

In a more programmatic way:

```
Private bitarray
bitarray[0][3] = HeaterOn
bitarray[0][1] = ValveState
bitarray[0][0] = SwitchState
Out = From.Bit(bitarray)
```

Notice that we have to subset in the second dimension, not the first, to set a particular bit. This is not required if you are only converting one value at a time, but you can convert a full matrix of bits to multiple values this way.

Both these examples also show a trick when doing array creation. By setting the largest array element ([3]) first, we preallocate the array to that size, putting 0's in all the other elements. If we were to set the elements in order, [0], [1], then [3], internally DAQFactory would be forced to reallocate three times which is much slower, especially when the arrays get large.

HexString(string, element size): converts a string in hexstring format to an array of numbers. Every two characters in the hexstring correspond to one byte. Element size determines the number of bytes in each number (i.e. 1 = byte, 2 = word, 4 = double word / long integer or float, etc). However, the numbers are not reassembled. Instead, if you specify an element size other than one, you will be returned a multidimensional array. You can then use one of the To. functions to convert that array into an array of numbers. The hex string is typically for use in setting/reading blob data in SQL statements.

From.HexString("FFFF7FFF0003", 2) returns the **{{255, 255}, {127, 255}, {0, 3}}**

We can then convert that into the proper numbers using to.urWord():

To.urWord(From.HexString("FFFF7FFF0003", 2)) returns the **{65535, 32767, 3}**

Comment: now you may wonder why the To and From functions take and return values in the second dimension. This is because the first dimension in DAQFactory always refers to values in time, and individual bytes or bits are all part of a single value at a single time. This becomes handy when you want to do something like plot the status of a certain bit of a channel over time. Lets say you want to plot bit 2:

Y Expression: **(To.Bit(MyChannel,3))[][2]**

Notice also in the above expression that we do **,3**. This means that only up to the desired bit is converted saving memory. Of course the better way to retrieve a single bit is to use the TestBit function:

Y Expression: **TestBit(MyChannel,3)**

But if you wanted to sum the second and third bits for some reason...:

Y Expression: **SumCols(To.Bit(MyChannel,3)[][1,2])**

That all said, if you only want to convert to a single value with a To function (other then bit), or From.Bit, you can specify an array in the first dimension. So, **`To.Word({{0,128}})`** is the same as **`To.Word({0,128})`**. Of course, when you are dealing with time, you have to follow the convention just described.

4.12.11 Time functions

Time is an important part of data acquisition, and an important part of DAQFactory. We have provided some functions to make dealing with time a bit easier for you.

Align(Source, Align To, Threshold):

One of the powers of DAQFactory is the ability to take data at different data rates, or simply different times. This makes it difficult to perform many calculations such as correlation and the like. The **`Align()`** function will align the time associated array specified in source to the array given by **AlignTo**. Data that is outside of +/- threshold seconds from the times in **AlignTo** are set to **NaN()**. The value returned by this function is the aligned array. Note that many of the built in tools of DAQFactory automatically do an alignment when necessary. This includes X/Y graphs, error bars, trace colorization, logging and exporting, and correlation calculations.

FormatDate(time array)

This function converts a DAQFactory time value into a string representation of the date using your current windows locale setting. For example, with a US locale:

`FormatDate(04y2m18d)` returns **`"02/18/04"`**

FormatDateTime(format specifier string, time array)

This function converts a DAQFactory time value into a string representation using the format specifier provided to the function. The format specifier works a lot like the Format() function, but with different codes. Here are the codes:

`%a` The abbreviated weekday name (Mon, Tue, etc)

`%A` The full weekday name (Monday, Tuesday, etc)

`%b` The abbreviated month name (Jan, Feb etc.)

`%B` The full month name (January, February, etc.)

`%c` Date and time representation using the Window's locale settings

`%d` The day of month as decimal number (01 – 31)

`%H` The hour in 24-hour format (00 – 23)

`%I` The hour in 12-hour format (01 – 12)

`%j` The day of year as decimal number (001 – 366)

`%m` The month as decimal number (01 – 12)

`%M` The minute as decimal number (00 – 59)

`%p` The A.M./P.M. indicator for 12-hour clock as determined by the Window's locale settings

`%S` The second as decimal number (00 – 59)

`%U` The week of year as decimal number, with Sunday as first day of week (00 – 53)

`%w` The weekday as decimal number (0 – 6; Sunday is 0)

`%W` The week of year as decimal number, with Monday as first day of week (00 – 53)

%x Date representation using the Window's current locale settings

%X Time representation using the Window's current locale settings

%y The year without century, as decimal number (00 – 99)

%Y The year with century, as decimal number

%z, %Z Either the time-zone name or time zone abbreviation, depending on registry settings; nothing if time zone is unknown

%% Percent sign

Examples:

FormatDateTime("The current hour is: %H",12h45m) returns **"The current hour is: 12"**

FormatDateTime("%c", SysTime()) returns something like **"02/20/04 12:14:07"** depending on the current time and your locale settings.

FormatDateTime("%B %d, %Y",SysTime()) returns **"February 20, 2004"** for the above time.

FormatTime(time array)

This function converts a DAQFactory time value (seconds since 1970) into a string representation of the time using your current windows locale setting. For example:

FormatTime(1h2m3) returns **"01:02:03"** on a system set to display 24hr time.

GetTime(array):

Many of the values in DAQFactory have time arrays or values associated with them. Often these time values are used automatically, but sometimes you may wish to get the time array associated with a value array. The **GetTime()** function does just that. Alternatively with channels and variables, you can use .Time notation. So **GetTime(MyChannel)** is the same as **MyChannel.Time.** However, this does not work for the results of expressions, so **(Max(MyChannel[0])).Time** won't work, but **GetTime(Max(MyChannel[0]))** will. This is especially useful with the max/min functions as DAQFactory will actually return the time of the maximum or minimum point using GetTime().

InsertTime(Value, Start, Increment):

If you have an array that does not have time associated with it, you can use the **InsertTime()** to inject your own time into the array. This function returns the array given by value with a time value inserted for each element in the rows dimension. The time inserted is **Start** for the first element, then incremented by **Increment** for each element following. Because most arrays are set up with time backwards (i.e. point 0 has the latest time), you typically will want to use negative numbers for the **Increment** parameter.

ShiftTime(Value, Offset):

Often there are different latencies in instrumentation which makes intercomparison difficult without the **ShiftTime(Value, Offset)** function. This function returns Value with the time associated with it shifted by Offset.

SortTime(array):

DAQFactory typically takes care of keeping your data points sorted by time. However, when you perform the **Concat** function with two arrays with time associated with them, the resulting array will have time out of order unless all the points in the second array have times later than those in the first array. Often times this doesn't matter, but there are many parts of DAQFactory that, for optimization reasons, expect their arrays to be sorted by time. Graphs are the prime example of this. The **SortTime(array)** will sort an array for you based on time. We didn't automatically sort the array after a concat because often times you don't need to sort and sorting tends to be a relatively slow process.

StrToTime(string array, mask string):

This function allows you to convert a string or an array of strings specify date and time (i.e. "10/22/06 3:12:05") and convert them into DAQFactory time (number of seconds since 1970). If an array of strings is specified, then an array of times is returned. This function is most appropriate when reading in data logged in human readable time format. The mask string determines the order of the year, month, day, hour, minute and second fields. Use the

characters "ymdhms" to specify the ordering. For example, for a standard US date / time string, you would put a mask of "mdyhms". For European time, you would put "dmyhms". For example:

`StrToTime("10/22/06 3:12:05","mdyhms")` returns **`1161486725`**

If any of the date fields are not specified, the current date is assumed, so ("3:12:05","hms") would return the time of 3:12:05 today. If any time fields are missing, then they default to 0, so ("10/22/06","mdy") returns 10/22/06 at midnight. Note that you must specify the hour if you want to specify the minute, and it must be before the minute and after the month (if specified). This is because the "h" in the mask tells DAQFactory that the "m" following is minute and not month.

This function will interpret decimal seconds, so:

`StrToTime("10/22/06 3:12:05.123","mdyhms")` returns **`1161486725.123`**

The delimiters used to separate the fields of the date and time does not matter and any non-numeric value will work except ".", which is used to specify decimal seconds. There does have to be delimiters, even if they are spaces. This function does not work with fixed field time/date specifications (such as "102206 031205"). For these, you would have to manually insert delimiters using the mid() function, before using this function.

SysTime():

To get the current computer time use the **`SysTime()`** function. This returns the number of seconds since January 1st, 1970 (with decimal seconds).

Note: DAQFactory uses the system clock only to retrieve the initial time. Once DAQFactory starts, it uses the high precision counter to determine the current time. In addition to providing much more precise time measurements, this also means that changes to the system clock will NOT affect the DAQFactory time. You have to restart DAQFactory after a change in the system clock.

4.12.12 Array creation and manipulation functions

Concat():

`Concat(array1,array2,[array3,...])` will combine the arrays listed together and return the resulting array. This function can take up to twenty different arrays. This function is useful for things such as **`Max(Concat(chanX, chanY))`** which will return the maximum value of both chanX and chanY. Time is also combined in the concat, so **`GetTime(Max(Concat(chanX,chanY)))`** will return the time of the maximum point. The concatenation is a straight concatenation, i.e. **`Concat({1,2},{3,4,5})`** results in {1,2,3,4,5}. Time is concatenated the same way. This is not a problem when doing something like the GetTime(..) expression we used previously, but many parts of DAQFactory, graphs especially, expect an array to be sorted in time. In this case, you will need to use the SortTime() function to put the points in the proper time order. Note also that the arrays must be conformable. This means that they must have the same 2nd and 3rd dimensions. Concatenation always occurs along the rows (1st) dimension.

Fill(Value, Array Size)

Returns an array of Array Size where all elements contain Value.

`Fill(5,10)` returns **`{5,5,5,5,5,5,5,5,5,5}`**

Filter(Array, Criteria Array)

Returns an array of Array's elements where the corresponding row index of Criteria Array is non zero. This is perhaps best explained with an example. Let's say you have two arrays, one "Temperature" and one "Pressure" and you want to get all the pressure measurements where temperature is > 100. To do this, you would do **`Filter (Pressure,Temperature > 100)`**. Note that this only makes sense if the temperature and pressure measurements line up. The Align() function can help with this if these two inputs are taken at different data rates. If the Criteria Array is shorter than the Array, then Array is first truncated to the length of Criteria array. Here is another, more simple example:

`Filter({1,2,3,4,5},{0,1,1,0,0})` returns **`{2,3}`**

Flatten(Array)

Takes an array of multiple dimensions and converts it into an array with just one dimension. For example:

Flatten({{1,2,3},{4,5,6}}) returns **{1,2,3,4,5,6}**

This is especially useful when using the From. functions, which take numbers and convert them into arrays of bytes. The From. functions return an array in the second dimension (i.e. from.uWord(5) = {{5,0}} not {5,0}). This is to allow you to convert an array of values (i.e. from.uWord({5, 10}) = {{5,0},{10,0}}) But chra() does not work on 2 dimensional arrays, so you would use flatten to convert it first, then chra() to convert it to a string to send out a serial or Ethernet port.

IsEmpty(Array)

Returns 1 if the given array is empty (no values). This can be used to determine if a particular channel has a valid value, whether a variable has been initialized, or to look at the return value for functions. Note that "{}" does not generate an empty array. Use NULL instead.

NumCols(Array)

Returns the number of elements in the Column (2nd dimension) direction of the given array

NumCols({1,2,3}) returns **1**.

NumCols({{1,2,3},{2,3,4}}) returns **3**.

NumCols({{{1,2,3},{2,3,4}},{{4,5,6},{4,5,6}}} returns **2**.

NumDepth(Array)

Returns the number of elements in the Depth (3rd dimension) direction of the given array.

NumDepth({1,2,3}) returns **1**.

NumDepth({{1,2,3},{2,3,4}}) returns **1**.

NumDepth({{{1,2,3},{2,3,4}},{{4,5,6},{4,5,6}}} returns **3**.

NumRows(Array)

Returns the number of elements in the Row (1st dimension) direction of the given array.

NumRows({1,2,3}) returns **3**.

NumRows({{1,2,3},{2,3,4}}) returns **2**.

NumRows({{{1,2,3},{2,3,4}},{{4,5,6},{4,5,6}}} returns **2**.

Point():

Point(Value) returns the index of each element in the rows direction in place of the value of the element. For example, **Point({4,2,5,3,8})** will return {0,1,2,3,4}

Search(array, [starting row]):

Searches the array starting at the specified row, or the first row if not specified, and returns the row of the first non-zero value, or -1 if not found. This sounds like a less than useful function, but if your remember, you can apply boolean math to an array and it will return an array of 0's and 1's. This means you can search for the first row to match a particular boolean expression. So, if you declared: **global y = {1,2,5,2,3,7}**

Search(y == 2,0) returns **1** (the row of the first 2)

Search(y == 2,2) returns **3** (the row of the second 2, since we started past the first 2)

Note that this function only works along the rows dimension.

SeqAdd(Start, Increment, Array Size)

Returns an array of Array Size where the first element contains Start and each subsequent element contains the value of the previous element plus Increment.

SeqAdd(0,1,10) returns **{0,1,2,3,4,5,6,7,8,9}**

SeqMul(Start, Multiplier, Array Size)

Returns an array of Array Size where the first element contains Start and each subsequent element contains the value of the previous element times Multiplier.

`SeqMul(1,2,5)` returns `{1,2,4,8,16}`

Transpose(Array, Dimension)

Takes a one dimensional array and changes the direction to the given dimension. This only works on one dimensional arrays. For dimension, use 0 to create an array with multiple rows, 1 for multiple columns, and 2 for multiple depth.

`Transpose({1,2,3},1)` returns `{{1,2,3}}`. This took an array with three rows, and returned an array with three columns.

4.12.13 Random Number functions

Rand(Array Size) returns an array of Array Size filled with random numbers between 0 and 32767.

RandNormal(Center, Sigma, Array Size) returns an array of Array Size filled with random numbers normally distributed around Center with the given sigma value.

RandUniform(Bottom, Top, Array Size) returns an array of Array Size filled with random numbers evenly distributed between Bottom and Top.

4.12.14 Thermocouple conversion functions

We have included a set of functions to help you convert voltage measurements from a thermocouple into Celsius temperature measurements. All have the same format, the only difference is the thermocouple type. The generic format is `TypeX(voltage, cold junction temperature)`, where the voltage is an array of voltage measurements taken from your thermocouple, and the cold junction temperature is your CJC temperature in degrees Celsius. The CJC can either be a scalar or an array of values. The available functions are:

TypeB
TypeJ
TypeK
TypeN
TypeT
TypeE
TypeR
TypeS

4.12.15 Misc functions

NULL:

This is not so much a function as a constant with no value. This can be used to clear out a variable, but is really designed for use with external DLL calls to pass NULL values.

```
X = NULL
```

Note that you cannot do a comparison to NULL. Use the IsEmpty() function instead.

RGB(Red, Green, Blue):

This function creates a singular RGB color number out of the three color components. The three components each can range from 0 to 255.

Examples:

RGB(0,0,0) is black

RGB(255,255,255) is white

RGB(255,0,0) is red

RGB(0,255,0) is green.

To provide the user with a color selection dialog, see System.ColorSelectDialog()

4.12.16 Constants

For your convenience, we have provided quite a few constants:

Pi()	3.141592653589793
Amu()	1.660566e-27 kg
MassN()	1.674954e-27 kg
MassE()	1.672649e-27 kg
MassP()	9.109534e-31 kg
ChargeE()	1.602189e-19 C
BohrRadius()	5.29177e-11 m
Planck()	6.626176e-34 Js
GConstant()	6.6720e-11 Nm^2kg^{-2}
GAcceleration()	9.80665 m/s^2
SpeedLight()	2.9979246e8 m/s
Boltzmann()	1.38066e-23 J/K
Rydberg()	1.0973732e7 1/m
StefanBoltzmann()	5.670e-8 $Wm^{-2}K^{-4}$
Wein()	2.898e-3 mK
Avogadro()	6.022045e23 1/mol

GasConstant()	8.31441 J/Kmol
Faraday()	9.64846e4 C/mol
Permittivity()	8.854e-12 $AsV^{-1}m^{-1}$
Permeability()	1.25664e-6 $VsA^{-1}m^{-1}$
Euler()	0.577215664901532
Golden()	1.618033988749894

4.12.17 Advanced Analysis functions

Many of the advanced analysis capabilities of DAQFactory can be used directly in expressions for real-time analysis.

These functions are not available in all versions of DAQFactory.

Convolution and Correlation:

Convolve(Value1, Value2)
ConvolveFFT(Value1, Value2)
Correlate(Value1, Value2)
CorrelateFFT(Value1, Value2)

These functions perform convolution or correlation of Value1 and Value2. The FFT version simply uses an FFT to aid in the calculation.

FFT Functions:

All the FFT functions perform an FFT on the given array. The difference between them is how the resulting imaginary array is converted into real values:

FFTReal(Value)	Returns the real part of the result of the FFT
FFTPower(Value)	Performs a power function on the result of the FFT
FFTModulus (Value)	Performs a modulus function on the result of the FFT
FFTNorm(Value)	Performs a normalization function on the result of the FFT

FFT Windows:

The following windows are available to be used before performing an FFT operation. Each simply returns the Value after the window has been applied.

Hamming(Value)
Hanning(Value)
Blackman(Value)

BlackmanHarris3(Value)
BlackmanHarris4(Value)
TriangleWindow(Value)
CosineWindow(Value)
PoissonWindow(Value, Alpha)

Histogram:

Histogram(Value, Bin Array): returns value binned into bins as determined by Bin Array. Bin Array holds the upper limit of each bin. The resulting array is one element larger than Bin Array. All elements in Value above the last bin in Bin Array are put in this extra bin. Note that Bin Array must be in increasing numeric order to work properly.

Interpolation:

InterpolatePoly(X Array, Y Array, Interpolate To Array X)
InterpolateRational(X Array, Y Array, Interpolate To Array X)

Both of these functions interpolate the function described by X vs Y array to the X values given in Interpolate To Array X. The first function uses a polynomial function to perform the interpolation, the second, the rational function.

4.13 Questions, Answers and Examples

4.13.1 Converting a string to DAQFactory time

Question:

I need to read a string from a file (such as "01/02/2004" or " 20-12-1999") and convert this into the DAQFactory time format (i.e. number of seconds since Jan 1st 1970). Is there a function available that can do this? If not, any idea where I can find an algorithm that will allow me to write my own function to do this?

Answer:

As of 5.70 we've added a function called StrToTime() to do this. This function is described in the section on Time functions. Since the old method is useful as an example of parsing and evaluate(), we've left it here:

There isn't a direct function, but there is a pretty easy way to do it using the Evaluate function. The general idea is that DAQFactory expressions will accept time and date as constants if specified using our YMD hms format. So, "20-12-1999 13:15:02" (European format) would be "99Y12M20D 13H15M02"

So, what we need to do is get your string into that format. To do this, we'll create a sequence function that takes two arguments: strArg0 = "20-12-1999" and strArg1 = "13:15:02":

```
Private string strY = Parse(strArg0,2,"-")
Private string strM = Parse(strArg0,1,"-")
Private string strD = Parse(strArg0,0,"-")
Private string strH = Parse(strArg1,0,":")
Private string strMin = Parse(strArg1,1,":")
Private string strS = Parse(strArg1,2,":")
Private result = Evaluate(strY + "Y" + strM + "M" + strD + "D" + strH + "h" + strMin + "m" + strS)
return (result)
```

You could of course put all the Parse() statements in the evaluate and skip all the variables, but this is a bit easier to read.

Now, you can call your function from anywhere. Lets say we called this sequence StringToDate. So:

```
Var.date = StringToDate("20-12-1999","13:15:02")
```

To confirm it is working, go to the command line and put:

```
? FormatDateTime("%c",StringToDate("20-12-1999","13:15:02"))
```

The FormatDateTime() will take the DAQFactory time you created in your sequence and convert it back to a string. They should therefore match.

4.13.2 Calcing the max for a period of the day

Question:

I need my customer to be able to enter on a system set up screen, a time from (PeakTimeStart) and a time to (PeakTimeStop). This then needs to set the energy cost (EnergyCost) to a peak value say from 6:00 am to 6:00pm and an off peak cost from 6:01pm to 5:59am using the variables (PeakCost) and (OffPeakCost). The energy cost needs to change from Peak to Off Peak and back dependant on system time, each day and can be set by the customer in a runtime version using an edit box of some sort.

What would be my best approach for this?

Answer:

First you need to come up with a way to enter the time. You can use the date/time edit box but it returns the full date and we just want the time, but we can get around that using modulus. So you put those times in PeakTimeStart and PeakTimeStop. We'll assume that peak time is always in the middle of the day and never across midnight.

OK, now that we have those values, you should put an event on your channel that determines the energy cost. Lets say the channel is called "Usage" and cost = usage * 10. Here's the event code:

```
CurrentCost = Usage[0] * 10
switch
 case ((SysTime() % 86400) < (PeakTimeStart % 86400))
    // non-peak
    if (OffPeakCost < CurrentCost)
       OffPeakCost = CurrentCost // this also sets the time too!
    endif
 case  (((SysTime() % 86400) >= (PeakTimeStart % 86400)) && ((SysTime() % 86400) <  (PeakTimeStop %
86400)))
    // peak
    if (PeakCost < CurrentCost)
       PeakCost = CurrentCost // this also sets the time too!
    endif
 case ((SysTime() % 86400) >= (PeakTimeStop % 86400))
    // non-peak
    if (OffPeakCost < CurrentCost)
       OffPeakCost = CurrentCost // this also sets the time too!
    endif
endcase
```

A few quick points:

1) % is modulus, which in this case strips the date from DAQFactory time and simply returns the time of day in seconds since midnight. Since the date/time component returns the full date and time even if you only display the time in the component, we have to strip it for the comparison.

2) you could do this in a sequence with an infinite loop and a delay() instead of an event. By using an event, this always get called every time CurrentCost is calculated and you don't have a second loop running.

3) we used a switch/case statement here since only one of the three options can exist at a time. Using switch runs a little faster than a bunch of ifs.

4) the code could be made more efficient by pre-calcing the modulus:

```
Private time = SysTime() % 86400
PeakTimeStart = PeakTimeStart % 86400
PeakTimeStop = PeakTimeStop % 86400
```

and then use these variables in the case statements.

5) as the comments indicate, the time of the peak is also stored. When the first line executes, CurrentCost = Usage [0] * 10, the time of Usage[0] is stored with CurrentCost. When OffPeakCost = CurrentCost is executed, once again

the time carries through. This means you can display both OffPeakCost and OffPeakCost.Time to display the peak cost and the time of that peak cost.

6) this code does not reset the Peak and OffPeak costs each day. You'd have to do this manually by setting the variables to 0. You can do this automatically with a flag. Lets assume you've initialized it somewhere else and called it PeakFlag:

```
CurrentCost = Usage[0] * 10
Private time = SysTime() % 86400
PeakTimeStart = PeakTimeStart % 86400
PeakTimeStop = PeakTimeStop % 86400
switch
case (time < PeakTimeStart)
   // non-peak
   if (PeakFlag)
      OffPeakCost = 0
      PeakFlag = 0
   endif
   if (OffPeakCost < CurrentCost)
      OffPeakCost = CurrentCost // this also sets the time too!
   endif
case  ((time >= PeakTimeStart) && (time < PeakTimeStop))
   // peak
   if (!PeakFlag)
     PeakCost = 0
     PeakFlag = 1
  endif
  if (PeakCost < CurrentCost)
      PeakCost = CurrentCost // this also sets the time too!
   endif
case (time >= PeakTimeStop)
   // non-peak
   if (PeakFlag)
     OffPeakCost = 0
     PeakFlag = 0
  endif
  if (OffPeakCost < CurrentCost)
      OffPeakCost = CurrentCost // this also sets the time too!
   endif
endcase
```

4.13.3 A trick to avoid divide by 0 and other boolean tricks

Question:

I am using V channels to calculate the coefficient of performance. The expression for this V channel is **`Trane_kwRef / TraneKWE`**. The problem is that when we turn our chiller off, TraneKWE goes to 0 and so we get NaN because we are dividing by 0. We'd like the value to read 0 in this situation instead of NaN.

Answer:

The trick to solving this involves some boolean math. Change your V channel expression to:

```
Trane_kwRef * (TraneKWE != 0) / (TraneKWE + (TraneKWE == 0))
```

Here we have two boolean sections. The first, in the numerator, set the value to 0 when TraneKWE equals 0. This is because **`(TraneKWE != 0)`** evaluates to 0 when TraneKWE does equal 0, and 1 when it does not.

The boolean in the denominator prevents a divide by 0 error by adding one to the denominator when TraneKWE equals 0. This is again because **`(TraneKWE == 0)`** evaluates to 0 or 1.

Continuing on, let's say you wanted the value to read -999 instead of 0 when TraneKWE equals 0. The denominator will stay the same, but the numerator will be changed:

```
(Trane_kwRef * (TraneKWE != 0) - 999 * (TraneKWE == 0)) / (TraneKWE + (TraneKWE == 0))
```

The trick here in the numerator is that only one of the two boolean expressions will be true at a time since they are opposites. So we'll either get:

```
Trane_kwRef * 1 - 999 * 0 = Trane_kwRef
```

or

```
Trane_kwRef * 0 - 999 * 1 = -999
```

This numerator expression actually shows how to do an inline iff() statement that exists in many other languages. This structure and the iff() allows you to select one of two values for use in the rest of an expression based on a boolean expression similar to an if. iff() is not supported by DAQFactory. Using this method, though, you can actually extend it to multiple options:

```
1000 * (x > 10) + 500 * ((x <= 10) && (x > 0)) + 50 * (x <= 0)
```

This will return 1000 if x is greater than 10, 500 if x is between 0 and 10, and 50 if it is less than 0. Notice how we cover every possible value of x using a combination of > and <=. This can of course be extended to more values.

Note that all these expressions could be used anywhere, not just in a V channel. In many situations, however, you may want to use [0] notation on all the channels so you only get the most recent reading. By not using [0] here this creates a V channel with a full history based on the histories of Trane_kwRef and TraneKWE.

Finally, and hopefully not to confuse you, any computer scientist will tell you that + and * can actually be thought of as OR and AND in these sort of situations. So:

```
Trane_kwRef * (TraneKWE != 0) - 999 * (TraneKWE == 0)
```

actually could be read as:

```
Trane_kwRef AND (TraneKWe != 0) OR -999 AND (TraneKWE == 0)
```

(remember the minus sign in the original expression is really just + -999 ...)

4.13.5 Counting digital transitions

Question:

We are recording once a second the up time of a machine on a digital channel. This results in an history with 0's and 1's. We then simply do Sum(UpChannel) / 3600 and get an uptime ratio. Now we'd like to see how many times the machine gets switched on. Is there a way to count the transitions from 0 to 1 in this history array?

Answer:

Yes. Let's say you are storing the 0's and 1's in a channel called "State" with a history of 3600.

```
Var.Trans = (State[0,3600] != State[1,3600])
Var.TotalTrans = Sum(Trans)
```

The first statement creates an array of 0 and 1, putting a 1 everywhere there is a transition from 0 to 1 or 1 to 0. The second then sums those 1's. Of course to get just transitions from 0 to 1, divide Var.TotalTrans by 2, or better yet:

```
Var.Trans = ((State[0,3600] != State[1,3600]) && State[1,3600]
Var.TotalTrans = Sum(Trans)
```

Now the first line puts 1's where there is a transition and the second value in the transition is 1 (i.e. 0 to 1).

4.13.6 Calculating the current second / minute / hour

Question:

How do I determine what minute it is in the current hour?

Answer:

There are two ways. The slow way is to use FormatDateTime() to format a string with just the minute and convert the result into a number. This is actually the best way to get the current day, month or year number. However, to calculate what hour, minute or second it currently is, you can use simple math because there are always 60 seconds in a minute, 3600 seconds in an hour and 86400 seconds in a day. Math is always faster then any string function, and FormatDateTime() is particularly slow.

Current second:

```
SysTime()%60
```

Current Minute:

```
SysTime()%3600/60
```

Current Hour:

```
SysTime()%86400/3600
```

The above formulas will give you a value with decimals. If you want an integer, just use floor():

```
Floor(SysTime()%86400/3600)
```

5 Sequences & Scripting

5 Sequences & Scripting

5.1 Sequence Overview

DAQFactory provides many features and many options to perform most of the operations you need for your application. But we have learned over the years that we cannot come close to predicting the many different ways our customers use DAQFactory. To enable you to get the most out of DAQFactory there are sequences. Sequences are essentially a type of scripting language for performing different tasks. This can be anything from simply setting a few outputs to preset values all at once, to performing certain actions at fixed times, to complex programs for complete automation of complex processes.

For many applications you will not need to use sequences at all. For this reason, we probably could have put this chapter towards the end of the manual. However, many of the examples throughout the manual showing more advanced DAQFactory use sequences, so we thought it best to introduce sequences early on.

You can create as many sequences as you need. Sequences are typically run in separate threads. A thread is like a separate program that runs concurrently with other threads within DAQFactory. Because of this, you can create sequences that perform different tasks and run them at the same time. Sequences can also call other sequences as functions allowing you to perform common tasks with ease. Sequence code is also used as events for other objects, for example when a channel value is changed, or a page component is clicked.

5.2 Creating a New Sequence

Sequence code is used in different places within DAQFactory. For example, you can add sequence code that executes when a user clicks on a page component. These sort of events or quick sequences are part of the object they apply to and are simply a parameter of that object. Most sequences, however, are standalone objects to perform different tasks in your application. Each sequence is listed in the workspace under **SEQUENCES:**. You can also click on **SEQUENCES:** to display the sequence summary view that will display all your standalone sequences and their current status. To create a new sequence, you can either right click on **SEQUENCES:** in the workspace and select **Add Sequence**, or select **Add Sequence** from the sequence summary view. You will be prompted for a name, which like the rest of DAQFactory object names, must start with a letter and only contain letters, numbers and the underscore. After this you will be placed in the sequence view for your new sequence.

Sequences look a lot like most scripting languages, and the sequence view is simply a text editor with some advanced features. Sequences are made up of multiple steps. Each step is entered on a separate line in the sequence view. You are free to add blank lines to make things easier to read. You can also add comments, which are not used by the sequence itself, but will help you remember what your sequences does. Comments can be added to the end of any line or on their own line by typing two slashes "//". For example:

```
declare X = 32  // set the X variable to 32
// and now set it to 40:
X = 40
```

You can also comment a block of code using "/*" and "*/". For example:

```
declare X = 32  /* this is a comment
this is still a comment */
X = 40
```

Indentation Conventions:

To make sequences easier to read, we strongly recommend using indenting to organize your code. Block operations are typically indented three spaces to clearly display the block. For example:

```
if (condition)
  do something
  and something else
else
  do something
endif
```

You can manually indent or outdent blocks of code by selecting the block and pressing tab or shift-tab.

Breaking up Long Lines:

You can break up long lines of sequence code using the backslash. The next line of code is assumed to start where the backslash is placed:

```
if ((x < 5) && (y > 6) && \
  (z > 3))
  … do something
endif
```

The backslash cannot be used inside of a quoted string:

NO:

```
strMyString = “The quick brown fox jumped over \
  the lazy dogs tail”
```

Yes:

```
strMyString = “The quick brown fox jumped over “ + \
  “the lazy dogs tail”
```

Saving Changes:

To save the changes made to a sequence, click the **Apply** button. Alternatively, if you want to check for possible syntax errors, you can click **Apply & Compile** and the sequence will be saved and compiled. If you do not compile, the sequence will automatically be compiled when you try and run it.

Note: If you apply changes to a sequence that is running, you must stop and restart the sequence before the changes will take effect. Likewise, if you apply changes to a sequence that is called from another running sequence or expression, the changes will not apply until the sequence has returned.

To quickly save your sequence, compile it and run it (or restart it if it is already running), select **Debug - Start Sequence**. Note that this menu item will change depending on whether the sequence is compiled and whether it is already running. You can also use F5 to do the same thing.

Separate Editor Window:

In many cases it is more handy to edit your sequences in a separate window instead of the normal DAQFactory window. To do so, right click on the sequence name in the workspace and select **Edit in Separate Window**. This will display a new window with the sequence editor. This window has its own menu, duplicating the appropriate functionality of the regular DAQFactory menu. Initially this window will appear on top of the main area of DAQFactory, but it is a standard window and can be moved about as needed. You can leave this window open while working with the rest of DAQFactory. This can be very helpful when you need to test a sequence and need to see your pages to see the results. To switch between windows you can do several things:

1) Each window will appear as a separate item in the Window's Taskbar. The icon is slightly different then the standard DAQFactory icon, and the sequence name is indicated.

2) You can use **Alt-Tab** and **Shift-Alt-Tab** to switch among windows as they appear as separate tasks inside of windows.

3) In the DAQFactory menu is a submenu named Window. From this menu you can select from among the open windows. Note that the list automatically puts the current window as the first in the list, and the rest in order based on their last appearance. This makes it handy as item 2 is always the last window you had open, so the key combination: **Alt-W 2** will always toggle you between two windows.

4) Once a separate window is open for a particular sequence, clicking on the sequence in the workspace will make the window appear. Note that if you close the separate window, you will have to reopen with **Edit in Separate Window**.

Note that you can only have one window open for a particular sequence.

5.3 Starting and Stopping a Sequence

There are many ways to start or stop a sequence. When we say start or stop a sequence we mean running the sequence in its own thread. A thread is like a separate program that executes concurrently with other threads within DAQFactory. Sequences can also be called as functions or used as events. In these cases, they are not started, but are instead called. Called sequences do not run in their own thread, but instead run in the thread of the caller.

The easiest way to start or stop a sequence is to right click on the sequence name in the workspace and select **Begin Sequence** or **End Sequence**. You can also select the sequence in the sequence summary view and click the **Begin** or **End** button. You can get to the sequence summary view by clicking on **SEQUENCES:** in the workspace.

These two methods are not the most convenient methods however. A more convenient method is to create a screen component, such as a button, text component, panel, or descriptive text component and assign the Start/Stop Sequence action to it. The descriptive text component provides the quickest way to do this:

1) On a page, right click and select **Displays-Descriptive Text** to create a new Descriptive Text component.

2) Hit the **Enter** key, or right click on the new component and select **Properties...** to open the properties window for the new component.

3) At the bottom right of the window there is a button labelled **Sequence Trig**. Click on this button.

4) This will display a new window. Enter in the name of the sequence you would like to start/stop and hit **OK**.

5) Click **OK**.

This procedure creates a descriptive text component that will display either **Stopped** or **RUNNING** depending on the state of your sequence and when clicked will toggle the sequence between running and stopped. Take a look at the **Action** page of the component to see the settings. Other components, such as the button, static text, panel, have the same page and can also be set to trigger the sequence without displaying its status.

5.4 The most basic sequence and assignment

The most basic useful sequence is probably one that sets a few output channels to different values. Assignment is done by simply specifying the object to assign a value to, an equal sign, and the value to assign. For example:

```
MyOutputChannel = 3
AnotherOutput = 4 * 3 / 2
YetAnotherOutput = 5 / InputChannel[0]
```

The part to the left of the equal sign must be something that can take a value and cannot be an expression. This can typically be a channel or a variable. The part to the right of the equal sign can be any valid expression. If a sequence was created with just these three lines and run, the sequence would set MyOutputChannel to 3, AnotherOutput to 6 and YetAnotherOutput to 5 divided by the most recent value of InputChannel. The sequence would then stop.

5.5 More advanced assignment operators and the arraystep statement

Sequences have several more advanced assignment statements. The first two are called increment and decrement:

```
MyOutputChannel++
X--
```

The first line sets MyOutputChannel to its current value plus 1. The second, sets X to its current value minus 1. These are the same as:

```
MyOutputChannel = MyOutputChannel[0] + 1
X = X + 1
```

If you need to increment or decrement by larger amounts, you can use the following notation:

```
MyOutputChannel+=2
```

This will set MyOutputChannel to its current value plus 2 and is the same as:

```
MyOutputChannel = MyOutputChannel[0] + 2
```

This notation can be used for more advanced assignment as well:

```
MyOutputChannel+= InputChannel[0] / 2
```

This will set MyOutputChannel to its current value plus the latest value of InputChannel[0] divided by 2. This is the same as:

```
MyOutputChannel = MyOutputChannel[0] + InputChannel[0] / 2
```

There are equivalent operators for subtraction, multiplication, and division:

```
X -= 3
Y *= InputChannel[0]
Z /= 10
```

which is the same as:

```
X = X - 3
Y = Y * InputChannel[0]
Z = Z / 10
```

ArrayStep()

The arraystep statement also does assignment, but works a little differently. It is best explained with an example:

```
while(1)
  arraystep(Out,{1,3,4})
  delay(1)
endwhile
```

This example will set the Out channel to 1, wait one second, then set Out to 3, wait another second, set Out to 4, wait another second and then repeat. The format of the arraystep statement is: **`arraystep(channel/variable, array)`**. The array can be fixed, like the example, or an array from a channel or expression.

5.6 Reading inputs with the read statement and process control loops

For many applications, data is acquired from devices at constant intervals. These are the applications that the Timing parameter of a channel is designed for. For some applications you may want to read data at varying intervals and Timing will work for that as well. Some applications you will only want to read data when you need it. For these types of applications there is the read() statement. In order to use the read statement though you must still create a channel to identify what input you wish to read. Since you are using the read() statement for acquiring the data you should set the Timing parameter for your channel to 0.

Now, assuming we've created a channel called Input with a Timing of 0, we can use something like the following to read the channel at a varying time interval:

```
while(1)
  read(Input)
  delay(readinterval)
endwhile
```

This sequence will read Input at an interval determined by readinterval. readinterval could then be set using a page component allowing the user to specify the acquisition rate without changing the channel parameters.

Another use for the read statement is for process control loops. One way to implement process control loops is to read your inputs at fixed rate using the Timing parameter and use the channel's event. The event sequence code is executed every time a new value arrives on the channel, so you can evaluate the input and perform an output based on that input value. Alternatively, you can combine the polling and the control into a single loop. For example:

```
while(1)
  read(Pressure)
  if (Pressure[0] > 50) // if pressure is too high
```

```
    VacuumPump = 1  // start pump
    while(Pressure[0] > 50)  // wait until pressure drops
      delay(1)
      read(Pressure)  // read pressure again to check current value
    endwhile
    VacuumPump = 0  // then stop pump
  endif
  read(Temperature)
  if (Temperature[0] < 30) // if temperature too low
  Heater = 1 // turn on heater
    while (Temperature[0] < 30) // wait until warmer
      delay(1)
      read(Temperature)
    endwhile
    Heater = 0 // turn off heater
  endif
  delay(1)
endwhile
```

Although a simple example, it is pretty straight forward when things occur relative to each other. The pressure is read in from the device, then it is checked, then the temperature is read, then it is checked, then we wait a second and repeat. We could continue this and make a more complex control system. When doing process control, and many other sequence tasks it is usually much easier to split things into separate sequences. In our example above, we are monitoring temperature and pressure. If the pressure gets too high, we turn the pump on until it drops. If the temperature is too low, we turn on the heater until it warms up. The problem here is two fold. First, if both the temperature drops and the pressure rises at the same time, the system will have to pump down the system until the pressure drops below 50 before it can start heating the system up. This may be desired, but if not, this is not the fastest way to get the system under limits. Second, this makes for a more complicated sequence. Instead we can make two sequences:

Sequence MonitorPressure:

```
while(1)
  read(Pressure)
  if (Pressure[0] > 50) // if pressure is too high
    VacuumPump = 1  // start pump
    while(Pressure[0] > 50)  // wait until pressure drops
      delay(1)
      read(Pressure)  // read pressure again to check current value
    endwhile
    VacuumPump = 0  // then stop pump
  endif
  delay(1)
endwhile
```

Sequence MonitorTemperature:

```
while(1)
  read(Temperature)
  if (Temperature[0] < 30) // if temperature too low
    Heater = 1 // turn on heater
    while (Temperature[0] < 30) // wait until warmer
      delay(1)
      read(Temperature)
    endwhile
    Heater = 0 // turn off heater
  endif
  delay(1)
endwhile
```

These two sequence can run concurrently within DAQFactory and so both temperature and pressure can be monitored and adjusted simultaneously. If the pressure rises too high and the temperature drops at the same time, the system can heat and pump down at the same time, being monitored by two separate sequences.

In general, you would probably stick with channel events for simple control like this. To see why, here is the event code for Pressure, assuming you had a Timing of 1 on the Pressure channel to poll it. The code to control the temperature would be very similar:

```
if (Pressure[0] > 50)
  VacuumPump = 1
```

```
else
  VacuumPump = 0
endif
```

or even more succinctly:

```
VacuumPump = (Pressure[0] > 50)
```

or for hysteresis, which is not implemented in any of the above code:

```
if (Pressure[0] > 50)
 VacuumPump = 1
endif
if (Pressure[0] < 40)
 VacuumPump = 0
endif
```

5.7 Programmatically starting and stopping things

Sequences, PID loops, logging and export sets all have to be started to do anything. They often need to be stopped as well. You can do this manually with a page component, automatically when the document loads by setting Auto-Start, or programmatically with this group of begin/end functions:

beginseq / endseq: These two statements will start or stop a sequence:

```
beginseq(MySequence)
```

With beginseq(), The sequence is started in its own thread. This is different from a sequence function call which runs the sequence within the current thread. Because beginseq() starts the sequence in its own thread, the statement will return immediately, and the newly started sequence will run concurrently with the calling sequence. If the sequence is already running, the statement is ignored and the sequence continues running. It is not restarted.

endseq() will stop a running sequence. If the sequence is not running the statement is ignored. The base sequence should be referenced in the statement. In other words, if sequence A is running and it happens to call sequence B as a function (not using beginseq) and you want to stop it, you should do "endseq(A)" and not endseq(B). Unless sequence B happens to be also running on its own in a separate thread, endseq(B) won't do anything.

You can also use endseq to end the current sequence. Just reference the current sequence by name inside the endseq() statement. That said, it is better to use the return statement for this otherwise you can't copy your code or rename the sequence without changing these lines.

beginpid / endpid: These two statements will start or stop a PID loop. PID loops also run in separate threads inside of DAQFactory and therefore run concurrently with sequences. These statements will return immediately.

```
beginpid(MyPIDLoop)
```

beginlogging / endlogging: These two statements will start or stop a logging set. Like PID loops and most sequences, logging sets run in their own thread concurently with sequences and PID loops. These statements will return immediately. Note that logging sets have a little time latency built in so additional data may be logged after the logging set is complete. Logging sets are really designed to be left running. If you want to conditionally log data, use an export set instead to log a single line of data, and then trigger the export set every time you want to log a line of data.

```
beginlogging(MyLoggingSet)
```

beginexport / endexport: These two statements will start or stop an export set. Like PID loops and most sequences, export sets run in their own thread concurrently with sequences and PID loops. These statements will return immediately.

```
beginexport(MyExportSet)
```

5.8 Timed sequences and the goto statement

Note: goto statements are usually considered bad form in programming as they make the flow of the sequence difficult to follow. For that reason, the time/goto loop method should only be used in the most basic of loops, and in our opinion, not even then. Support for time/goto is only provided for backwards compatibility. while() loops and the delay() function should be used instead.

A common use of sequences is to simply perform a different action at preset intervals continuously. For example, switching an output on and off to run a calibration. There are two ways to perform this sort of loop. The first uses a while loop and delay statements. This is usually considered the cleaner way to write sequences, but for simpler timed loops, the second method is often easier. This method uses time and goto statements. Please make sure and read the notes at the end of this section before using this method.

The time statement determines when the code following it will be executed relative to the start of a sequence. For example:

```
time 0
  OutputChannel = 1
time 3
  OutputChannel = 0
```

The above example will set OutputChannel to 1 as soon as the sequence starts, wait three seconds and then set OutputChannel to 0. At this point there is nothing else to do, so the sequence will stop.

To make a loop with time statements, use the goto statement:

```
time 0
  OutputChannel = 1
time 3
  OutputChannel = 0
time 10
  goto (0)
```

This sequence will do the same thing as the first sequence, but after the OutputChannel is set to 0, it will wait an additional 7 seconds, then branch back to the beginning and repeat the whole loop.

One of the nice parts about the time statement (and the wait statement discussed later) is that it is non-propogating. This means that if it takes a second to set OutputChannel to 1 at time 0, the statement at time 3 will still execute 3 seconds after the beginning of the sequence and not 3 seconds after OutputChannel is set to 1.

You can of course put multiple statements between time statements to perform multiple actions at the same time. You can even put if and other block statements:

```
time 0
  OutputChannel = 1
  if (Input[0] > 2)
    OtherOut = 1
  endif
time 5m
  OutputChannel = 0
  OtherOut = 0
time 1h1m1
  goto (0)
```

This sequences also shows time being specified using hms format. This allows you to specify times in minutes or hours or a combination. Note that these are relative times not absolute times and the date is not used.

This sequence also shows the block if statement correctly used at a single time. If you are going to put block statements like **if** in a sequence using time, make sure you do not have a time statement inside the block.

NO:

```
time 0
  if (Input[0] > 2)
time 1
    OtherOut = 1
  endif
time 2
  goto (0)
```

The goto statement can go to other times besides 0 too:

```
time 0
  OutputChannel = 1
time 5
  OtherOut = 1
time 10
  OtherOut = 0
time 15
  goto (5)
```

This sequence will initialize OutputChannel to 1, wait 5 seconds, then alternate OtherOut between 0 and 1 every 5 seconds.

As you can see the time / goto type sequence makes things pretty easy for creating timed loops. It is important to remember though that for more complicated loops that may branch in different ways or require different delays, this method is not the best. A while loop and wait or delay statements are definitely the preferred method for anything other than the most basic timed loops.

Note: The time statements are non-propagating and therefore work like a wait() function. This is no problem if the things you are doing between the time statements will always take less time then the difference:

```
time 0
read(MyChannel)
time 1
goto 0
```

In the above example, as long as reading MyChannel will always take less than 1 second, then there will be no problem. This is usually the case for PCI DAQ cards and some external devices, such as the LabJack USB devices. If, however, it is possible for it to take longer than one second then this can cause a backlog and hang DAQFactory. The most common case where this occurs is when MyChannel refers to a serial channel, the timeout is set to one second or more, and the serial device is timing out. Sequences do not generate Timing Lag errors and do not reset their queue. So, if your device timed out for 60 seconds and then began functioning again, there would be 60 reads queue up and overdue. These would execute in rapid succession temporarily hanging DAQFactory.

5.9 Waiting in a sequence

The time statement provides one way to control the timing of a sequence, but it becomes very difficult to follow for anything but the most simple sequences. There are several different statements to help you control the timing of sequences.

delay() and wait()

The simplest of these statements are the delay() and wait() statements. These are both explained below, but in general you should always use the delay() statement as the wait() statement can hang DAQFactory if used improperly.

Both the delay() and wait() statements take a time in seconds to wait:

```
Output = 1
delay(60)
Output = 0
```

This will set Output to 1, wait 1 minute, then set Output to 0. This sequence will too:

```
Output = 1
wait(60)
Output = 0
```

The difference between the two is that the wait() statement is non-propagating. This means that if it takes one second for Output to be set to 1, then Output will still be set to 0 one minute after the sequence starts. The delay() statement is propagating and will set Output to 0 one minute and one second after the sequence starts. Again assuming it takes one second to set Output:

```
// this occurs as soon as the sequence starts:
Output = 1
delay(60)
// this occurs 1 minute 1 second after the sequence starts:
Output = 0
delay(30)
// this occurs 1 minute 32 seconds after the sequence starts:
Output = 2
```

Alternatively:

```
// this occurs as soon as the sequence starts:
Output = 1
wait(60)
// this occurs 1 minute after the sequence starts:
Output = 0
wait(30)
// this occurs 1 minute 30 seconds after the sequence starts:
Output = 2
```

You can also combine the two if needed:

```
// this occurs as soon as the sequence starts:
Output = 1
delay(60)
// this occurs 1 minute 1 second after the sequence starts:
Output = 0
wait(30)
// this occurs 1 minute 31 seconds after the sequence starts:
Output = 2
```

Choosing which function to use simply depends on how critical it is that events occur at fixed intervals. While it usually does not take a second to set an output, it does take a finite amount of time. This may or may not be an issue for your application. If you don't know or care, use the delay() statement. The wait() statement can cause DAQFactory to hang if used incorrectly. This next example shows one reason for not using the wait statement. Again, assume Output takes a second to set:

```
Output = 1 // this occurs as soon as the sequence starts
wait(0.5)
Output = 0 // this occurs 1 second after the sequence starts, as soon as Output = 1 is complete
wait(0.5)
Output = 2 // this occurs 2 seconds after the sequence starts, as soon as Output = 2 is complete
```

In this case, the wait statements don't do anything. Since the previous statement took longer than the wait time, there was no waiting. If this was inside a loop, the loop would run without any pausing to allow DAQFactory to do other things which could hang DAQFactory. To wait a half second between setting Output, the delay statement is the correct choice:

```
Output = 1 // this occurs as soon as the sequence starts
delay(0.5)
Output = 0 // this occurs 1.5 second after the sequence starts
delay(0.5)
Output = 2 // this occurs 3 seconds after the sequence starts
```

waituntil()

The waituntil command will wait until an absolute time:

```
Output = 1
waituntil(16h)
Output = 2
```

The above sequence will set Output to 1, then wait until 4pm today and set Output to 2. If it is already past 4pm, the sequence will set Output to 1 then immediately set Output to 2.

Application: Repeating something at the same time every day:

The most common use of waituntil() is to repeat something at the same time every hour, day or week. Here's an example that uses waituntil() to repeat something every day:

```
while (1)
  WaitUntil(16h + 86400)
  do something...
endwhile
```

Each time 16h is evaluated, it will automatically add in today's date, whatever that may be when the waituntil() line is executed. Depending on when the sequence started, it may already be past 4pm and we don't want the "do something" to execute until tomorrow. So we simply add 86400 to 4pm today which gives us 4pm tomorrow because there are 86400 seconds in a day. Thus the "do something" will execute at 4pm every day starting tomorrow at 4pm.

If you want the "do something" to start at 4pm today if sequence is started before 4pm, then you have to add some steps:

```
if (SysTime() < 16h)   // if its not 4pm yet
  WaitUntil(16h)   // wait until 4pm
else
  WaitUntil(16h+86400)  // otherwise, wait until 4pm tomorrow
endif
while (1)
  do something...
  WaitUntil(16h + 86400)  // wait until 4pm tomorrow.
endwhile
```

By reorganizing the loop to do something first, then wait until 4pm tomorrow we can add an if statement in front of the loop to determine when we want "do something" to execute first. If its not 4pm yet, then we simply wait until 4pm today, otherwise we wait until 4pm tomorrow.

A key to these samples is that while today's date is added to 16h each time the steps with 16h are executed, it only does so once. So while waiting for 4pm tomorrow (16h + 86400), the 4pm is not reevaluated at midnight thus pushing it back to 4pm the next day, and the next, forever in the future.

waitfor()

waitfor() is a bit different than the other wait statements. waitfor() will wait for a particular expression to evaluate to true:

```
output = 1
waitfor(Input[0] > 10,1)
output = 2
```

This sequence sets output to 1, then waits for Input to go above 10, then when that occurs sets output to 2. This of course assumes that you are reading Input somewhere else, either by setting the Timing parameter in the Input channel, or reading the channel through another sequence. The second parameter of the WaitFor statement determines how often the expression is evaluated. In this case, "Input[0] > 10" is evaluated once a second. To conserve processor power, this value should be set as high as possible. There is no practical use for setting it smaller then the rate the values in the expression are being updated, and it should never be smaller than 0.01 unless you have a multiprocessor computer and have planned for it. In general it should be set to maximum acceptable delay between the time the expression would evaluate to true and the time the next statement needs to execute. So in our example, if it is acceptable for Output to be set to 2 up to five seconds after Input goes above 10, then we should do "waitfor(Input[0] > 10,5)"

The above example is the same as:

```
output = 1
while (Input[0] <= 10)
  delay(1)
endwhile
output = 2
```

The waitfor() version is just a touch cleaner.

Note that the second parameter of waitfor(), the interval, must be a scalar number and cannot be a variable. If you need to use a variable loop time, use the normal while() loop syntax.

5.10 Variables

In addition to channels and V Channels for storing data, DAQFactory also supports several different types of variables: public, registry, private and static. Private and static variables are used in sequences and discussed in the next section. Public variables exist throughout DAQFactory and can be used in expressions or sequences. They can be set using the same mechanisms that channels can be set, even with knobs, switches and other components.

Public, global Variables

In order to use a variable, you must first declare it. To declare a public variable visible from everywhere within DAQFactory, use the global sequence command:

```
global MyVariable
```

You can also do an assignment at the same time:

```
global MyVariable = 3
```

String variables are created by adding "string" to the declaration:

```
global string MyStringVariable
MyStringVariable = "some string"
```

To determine if a variable exists yet, use the IsEmpty() function:

`IsEmpty(MyVariable)` returns 1 if MyVariable is not defined yet.

Note, however, that IsEmpty() will return 1 if MyVariable is declared, but contains NULL. NULL can be assigned: x = NULL, or is returned by several system functions.

ClearGlobals():

If you need to clear out your globals (and defines decribed below), use the ClearGlobals() function. This clears all globals and defines and basically resets the global variable namespace. Note that the "false" and "true" defines cannot be cleared out, but instead will reset to 0 and 1. You can, however, redefine them if you really wanted to.

You can also declare, assign and reference a variable by prefixing it with **Var.**:

```
Var.MyVariable = 3
```

In this case, MyVariable is set to 3. If it did not exist, it is declared first. This is most useful in for loops as we'll see later.

Note: Var.MyVariable notation is largely provided for backwards compatibility, but is useful in for() loops. One important difference is that to declare and define a string variable using Var. notation, you must name your variable with a name prefixed with "str". In other words: Var.strMyVariable = "some string". There is no way to create a string variable that does not start with "str" using Var. notation.

Note: Channels in the default connection also have global scope, so you should be careful to choose different names for your variables and channels to avoid any conflict. If there is a conflict, the variable will take precedence over the channel.

Variables vs. Channels

Variables can be used just like channels. Unlike channels, they do not have a history, but they can be created as arrays. Here are some differences:

`MyChannel = 5` will set the output channel MyChannel to a value of 5 and move all previous values in MyChannel's history back one spot, setting the most recent value to 5. So if MyChannel had a history of {3,2,6} before the above statement, it would have a history of {5,3,2,6} afterwards.

`MyVariable = 5` will set the variable MyVariable to a value of 5. Any data already in MyVariable is cleared out.

OK, we lied, if you are using DAQFactory Lite or higher there are several variable functions that will allow you to create a history using a variable, much like a channel. The big advantage of this is that variables tend to be faster than channels because they don't go through broadcasting, logging or alarming or any drivers. The functions are similar to their analogies for channels. All will start with the variable name, for example `MyVariable.AddValue(x)` :

AddValue(Value):

Calling this function adds a value to the history of the variable, moving the current value(s) in the variable back in the array. Value can be a scalar value or an array of values. If Value has time associated with it, the time of each row is inserted as well. If Value does not have time associated with it, the current time is inserted before the data is added to the channel. For array data with multiple rows, you must have time in the array beforehand. Use the InsertTime() function for this. Data will build up until the history length for the variable is reached. The default history length for a variable is 3600. To change, set the HistoryLength parameter of the variable to the new value, fox example: `MyVariable.HistoryLength = 100`. History only applies in a variable if you use the AddValue() function.

ClearHistory(len):

Calling this function will clear the variables history of all values except for the number of points specified in the function. Leaving this parameter blank clears all the points in the history, which is the same as `ClearHistory(0)`. After clearing, the history will rebuild itself as you call AddValue().

There are several other useful variable functions for inserting and removing array elements. The history length of the variable has no affect on these functions, so you'll have to limit the length of the arrays yourself if needed. These functions only work in the row dimension and do not work in DAQFactory-Express:

AliasFor(ChannelName):

Causes the variable to become an alias for a particular channel. The channel name should be passed as a string (i.e. myVariable.aliasFor("myChannel")). This function is especially useful when you have objects and want member variables of the objects to be connected to channels. Once a variable is set as an alias, reading the variable will instead give you the value in the channel, and likewise, setting the variable will send the value to the channel instead. To clear the alias setting of a variable, pass a blank string: myVariable.aliasFor("")

Append(Value):

Adds the given value to the end of the array, increasing the size of the array. This is the same as `myvar[numrows (myvar)] = Value`.

InsertAt(index, Value):

Inserts the given value at the given index of the array. So, if the variable contains `{1,2,3}` and you do `MyVar.InsertAt(1,4)`, MyVar will become `{1,4,2,3}`.

MakeNan(StartTime, EndTime):

Replaces all elements in the array with a time stamp between start time and end time with NaN(). Usefully for marking bad data so it is not used in mean, min, max and other statistical functions or included in trend graphs.

RemoveAt(index, [count]):

Removes count elements starting at index. So, if the variable contains `{1,4,2,3}` and you do `MyVar.RemoveAt(1,2)`, MyVar will become `{1,3}`. Count is optional and defaults to 1.

Push(Value):

Pushes the given value into the top of the array. This is almost identical to AddValue() except the history length is not looked at. It is largely used in conjunction with Pop() to create a stack.

Pop():

Removes the top element of the array (the last Push() usually) and returns that element. y=x.pop() is basically the same as doing y=x[0] followed by x.removeAt(0).

Arrayed Data

Arrayed data can be stored in variables though using either array notation:

`MyVariable = {5,3,2,5}` or by using subset notation to set individual elements-for example `MyVariable[3] = 5`

When using the subset notation in an assignment like this, existing values are not cleared out. So the following:

```
MyVariable = {5,3,2,5}
MyVariable[3] = 2
```

results in {5,3,2,2}

If you are initializing an array using the subset notation, it is recommended that you start with the last value in the array and work towards 0. When you set an array element past the end of an existing array, or in a new array, all intermediary values are set to 0. So, assuming MyVariable was empty, **`MyVariable[3] = 5`** will set MyVariable to {0,0,0,5}. By starting at the end of the array, you allocate the memory for the variable once, rather than reallocating multiple times which is much slower.

You can also make 3 and 4 dimensional arrays using similar notation:

`MyVariable[0][3][1] = 2` will, assuming MyVariable is empty, set MyVariable to {{{0,0},{0,0},{0,0},{0,2}}}. If MyVariable wasn't empty, it would simply set the array element in the 1st row, 4th column, 2nd depth to 2.

Like channels, variables can also have time associated with them. When you set a variable to a scalar value (**`MyVariable = 5`**), the time when the variable was set is also stored. This does not occur when setting array values. You can access this time using the GetTime() function, or you can use the following notation:

```
MyVariable.Time
```

or for an array:

`MyVariable.Time[5]` returns the time of the 6th row of the MyVariable array if time is available.

This notation also allows you to manually assign a time to a variable:

```
MyVariable.Time = 1h3m
```

To set the time of array values, just use subset notation:

```
MyVariable.Time[3] = 1h3m
```

Remember that only the Rows dimension has time, so the following:

```
MyVariable.Time[3][2] = 1h3m
```

and

```
MyVariable.Time[3][3] = 1h3m
```

will do the same thing because the row subset is the same.

As an alternative, you can use the InsertTime() function to quickly insert evenly spaced time values into an array.

Note: Graphs, and some other functions require time to be sorted in the array from most recent time back. Element 0 should have the most recent (largest) time. Use the SortTime() function to sort an array based on time into this order. If you are using InsertTime(), use a negative interval to go back in time.

Channels also support the .Time notation, but only for reading the time:

`MyChannel.Time[3]` returns the time of the 4th most recent data point of MyChannel.

`MyChannel.Time[3] = SysTime()` is invalid.

Defined constants:

Defined constants are used when you have "magic number", a constant that has some meaning. For example, in DAQFactory, False is equivalent to the constant 0, while True is equivalent to the constant 1. These are predefined constants. Defined constants work a little differently then variables. For one thing they are read only, and once you initialize them, you cannot change them. Also, because they are implemented at a compiler level for speed, the define statement is evaluated when the sequence compiles, not when it runs. Defines have global scope and will override pretty any other names you may have created in DAQFactory, including global variables and channels. The declaration of a defined constant is similar to a variables. Since the define is constant and read only, you have to initialize it in the declaration. For example:

```
Define False = 0
False = 3 // this does nothing
```

Like regular variables, add "string" to create a string define:

```
Define string CompanyName = "AzeoTech"
```

You can use defined constants just like regular global variables, except of course you cannot assign them a new value once declared except by redeclaring them. So if you do:

```
Define x = 3
Define x = 4
x = 5
```

then x will end up being 4.

You can clear defines and global variables by using the ClearGlobals() function. This function erases all defines and globals from memory.

Pulling constants out of C style .h files with Include():

If you are using an external DLL (using extern()) or a device that uses constants (i.e. LabJack), there are often a number of important constants defined in a header file, typically with device prototypes as well. While DAQFactory can't use the device prototypes directly, it can import a C style .h file and pull out most of the constants and create defines for them. It will pull out all non-array, non-string constants declared with either const or #define. To do so, use the include() function:

```
include("c:\program files\labjack\drivers\labjackud.h")
```

this will create a define for each valid constant DAQFactory can find in the header file. The variable type is largely ignored, as long as its a number, as DAQFactory only uses the double numeric type.

Registry variables:

Registry variables are also a global variable, but they are stored in the Window's registry and therefore persist between different sessions of DAQFactory. There is no declaration of registry variables. Registry variables are accessed by prefixing the variable name with **Registry**:

```
Registry.MyNumber = 5
```

```
Registry.strMyString = "data"
```

Registry variables have some limitations:

- In newer versions of windows (i.e. 7 and 8), access to the registry is limited for regular users. You will likely need to run DAQFactory as an Admin, or be logged in as an Admin or a similar user with the proper rights to write to the registry for registry variables to work.
- String registry variables must start with "str".
- Registry variables are a bit slower to access then regular variables.
- Registry variables do not support arrays.
- Registry variables only support integers and strings. Integers are double words and therefore must be between 0 and 2^32.
- Registry variables do not have time associated with them.

Uninitialized string registry variables will appear empty, i.e. `IsEmpty(Registry.strMyString)` will return 1. Uninitialized numeric registry variables return 0.

Once initialized, the only way to uninitilize the registry variable is to use the Windows Registry editor and manually remove the variable. Registry variables are stored in HKEY_LOCAL_MACHINE/Software/DAQFactory/Control/Variables. Setting a string registry variable to an empty string ("") will not uninitilize the variable, but instead will simply store an empty string.

5.11 Private sequence variables

In addition to the global variable types, sequences have two variable types of their own. They are Private and Static. Where Global and Registry variables exist throughout DAQFactory and are useable in any expression or local sequence, Private and Static variables only exist within the sequence in which they were created. This makes it so

you can use the same variable names to refer to different data in two different sequences. In general, if you are creating a variable in a sequence and you only plan on using within the sequence itself you should use a Private or Static variable.

Private variables are declared using the private statement.

```
private X = 1
private string MyString = "string"
private Y = {1,2,3}
```

Like regular global variables, you can assign arrays or strings to private variables. Also like global variables, you can set the private's initial value in the declaration statement. The initial value is only assigned if the variable does not already exist.

You can also declare, assign and reference a private variable by prefixing it with **Private.** Again, like "Var." notation, to create a string variable with "Private.", you must prefix the name of your variable with "str":

```
Private.MyVariable = 3
Private.strMyVar = "string"
```

In this case, MyVariable is set to 3 whether it exists or not. If it did not exist, it is declared first. This method is mostly provided for backwards compatability, but is also useful in for loops as we'll see later.

```
while (1)
  Private X = 3
  X++
  delay(1)
endwhile
```

So in the above sequence, X will be assigned to 3 only once and then incremented from there. The only difference between a global and Private is that the Private can only be seen by the sequence and once the sequence ends, the data in the Private is lost. Because of this, the private must be redeclared the next time the sequence is run. So, looking at the above sequence again, if the sequence is stopped and then restarted, X will be cleared when it is stopped, and will be redeclared and set to 3 again when restarted.

Privates are most commonly used for loops:

```
Private X = 0
while (X < 10)
  Output = X * 5
  X++
endwhile
```

This sequence simply sets Output to 0 through 45 in steps of 5.

Statics are almost the same as Privates, but unlike privates, they do not lose their data when the sequence ends. Instead, when the sequence is run again, the Static will still have the value it was last set to. This type of variable has its particular uses, but is not very common. Declaring statics is just like private and global variables:

```
Static Y = 0
Static String MyCompany = "AzeoTech"
```

The assignment is only made if Y does not already exist. So if this command is in a sequence and you call the sequence twice, the second time through it will not assign 0 to Y, but will simply ignore the statement as Y was declared the first time through. This allows you to initialize the static.

Remember that since statics persist even when a sequence stops and restarts that the above lines will only set the variables to the indicated values once. So:

```
Static Y = 0
Y++
? Y
```

will print 1 to the command window the first time it is run, and then 2 the next time, and so forth because Y only gets initialized to 0 when it is declared and doesn't exist already. If you used a private, 1 would be printed each time because the variable would fall out of scope and be reinitialized the next time the sequence is run.

Note: Although Statics retain their data after a sequence stops, they do not retain their value when DAQFactory quits. For this you must use a Registry variable.

Note: For backward compatibility, you can still use the Static.MyVar notation to create a static variable, but you will find that static declaration is easier because of the way initialization works with the new declaration methods.

5.12 Conditional branching with the if statement

DAQFactory can easily do different things depending on the state of your system or anything else. The if/else/endif statements provide the mechanism for this:

```
if (Input[0] > 5)
  do something when the latest value of Input is greater than 5
else
  do something when the latest value of Input is less than or equal to 5
endif
```

The else is actually optional. You can of course put multiple steps inside the if:

```
if (Input[0] > 5)
  Output = 3
  AnotherOut = 2
endif
```

You can also nest ifs inside of one another:

```
if (Input[0] > 5)
  if (AnotherInput[0] > 3)
    Output = 3
  else
    Output = 2
  endif
  AnotherOut = 2
endif
```

For that matter, you can nest while and for loops or other block structures inside of each other. Just make sure everything lines up correctly:

OK:

```
if (Input[0] > 5)
  for (Private.Count = 0, Private.Count < 10, Private.Count++)
    Output = Private.Count
  endfor
endif
```

NO:

```
if (Input[0] > 5)
  for (Private.Count = 0, Private.Count < 10, Private.Count++)
    Output = Private.Count
  endif
endfor
```

5.13 Replacing multiple ifs with a switch statement

If one you want to perform a different action for one of many possible options, you can use the switch/case/endswitch statements:

```
switch
  case (Input[0] > 5)
    Output = 2
  case (Input[0] > 3)
    Output = 1
  case (Input[0] > 1)
    Output = 0
  default
    Output = 5
```

```
endcase
```

This example sets Output to 2 if Input is greater than 5, to 1 if Input is greater than 3, but less than 5, to 0 if Input is greater than 1 but less than 3 and to 5 for any other value of Input. The case statements execute in order. Once a case statements expression evaluates to true, the steps following it until the next case or endcase are executed then the sequence jumps to the endcase statement. Therefore only one case statement executes. In this example, if Input[0] = 4 the second case "Input[0] > 3" will be the first to evaluate to true so its steps will execute even though the third case, "Input[0] > 1" is also true. The default statement's steps execute if none of the other case statements evaluate to true. The default statement is optional.

The example above can also be written with if/else statements:

```
if (Input[0] > 5)
  Output = 2
else
  if (Input[0] > 3)
    Output = 1
  else
    if (Input[0] > 1)
      Output = 0
    else
      Output = 5
    endif
  endif
endif
```

As you can see, the case statement is a quite a bit clearer.

5.14 Sequence looping with the while statement

The simplest way to make sequence steps execute repeatedly is to use a while loop. For example:

```
while (Input[0] <= 10)
  Output = 10 - Input[0]
  delay(1)
endwhile
```

The steps between the while and the endwhile will execute continuously until Input becomes greater than 10. In this particular case, Output is set to the difference between Input and 10 once a second until Input is greater than 10. The expression in the while can be more complicated then this example, and can also be very simple:

```
while (1)
  read (Input)
  delay(1)
endwhile
```

This sequence will read the Input channel once a second until the sequence is manually stopped. Because 1 is the same as true, this loop is an infinite loop, but because we put a delay statement inside the loop, this will not be a problem. If we forgot the delay statement, the loop would execute very rapidly and probably hang DAQFactory. With some rare exceptions, you should always put either a delay or wait statement inside of any loop.

Inside of the while loop, you can add a break statement to force the sequence to leave the loop:

```
while (1)
  if (Input[0] > 10)
    break
  endif
  Output = 10 - Input[0]
  delay(1)
endwhile
```

This sequence will actually do the exact same thing as our first while example. The while statement itself always evaluates to true, but inside the while we have an if statement that looks at Input and breaks out of the while loop if Input goes greater than 10.

Another useful statement inside of a while loop is continue. Normally all the steps between the while and the endwhile execute each loop. The continue statement causes the sequence to jump back to the while statement, reevaluate the condition and either start the loop over, or break out of it:

```
while (Input[0] <= 10)
  if (Input[0] > 9)
    delay(1)
    continue
  endif
  Output = 10 - Input[0]
  delay(1)
endwhile
```

This sequence works just like the one above it, except that the Output is not adjusted once Input goes above 9. The if statement looks for Input to go above 9, and when it does so, calls the continue statement which jumps the sequence back to the while statement. Notice the delay statement inside the if as well. If we did not put this in we'd have a fast infinite loop occur once Input raised above 9.

5.15 Looping while counting with the for statement

If instead of looping until a condition occurs you want to loop a certain number of times then the for statement is a better choice. For example:

```
for (Private.Counter = 0, Counter < 10, Counter++)
  do something...
endfor
```

This is the same as:

```
Private Counter = 0
while (Counter < 10)
  do something...
  Counter++
endwhile
```

The for statement is made up of three parts. The first part, **Private.Counter=0** is executed the first time the for statement is reached. In this case, we initialize the Counter variable to 0. By using the **Private.** in front of the counter variable, we declare (if necessary) and assign the initial value in one step. This saves us from having to do:

```
Private Counter
for (Counter = 0, Counter < 10, Counter++)
  do something...
endfor
```

The code between the for and the endfor statements will then execute repeatedly until the second term **Counter<10** no longer evaluates to true. The final term of the for statement gets executed every time the for loops. The first and last term are both statements and can perform anything you can normally do with a single sequence step. The middle term is an expression and should evaluate to a number.

For statements can also utilize the break statement to prematurely break out of a for loop, and the continue statement, to prematurely loop back to the for statement:

```
for (Private.Counter = 0, Counter < 10, Counter++)
  if (Input[0] > 5)
    break
  endif
  if (Input[0] < 3)
    delay(1)
    continue
  endif
  Output = Private.Counter
  delay(1)
endfor
```

The above example loops up to ten times. If Input ever goes above 5, the loop stops and jumps to any steps after the endfor. If Input is less than 3 than the loop does nothing, but if it is between 3 and 5, Output is set to the current value of Counter. Notice how we include delay() statements in the for loop. Delay or wait statements are needed to prevent the loop from running to quickly and stalling DAQFactory.

5.16 Avoiding being hanged using the infinite loop check

It is really easy to write an infinite loop in DAQFactory:

```
while(1)
  read(Input)
endwhile
```

On a first look, this sequence seems fine. It will continuously read the Input channel. The problem is that the sequence does not release the CPU for other tasks. Instead as soon as Input channel is read, the sequence loops and reads it again. This would be better:

```
while(1)
  read(Input)
  delay(0.1)
endwhile
```

Now we have a delay statement and the sequence will release the CPU for 0.1 seconds every time it reads Input. This will also cause Input to be read 10 times a second rather than as fast as electronically possible.

The worse part about the first example without the delay() statement is that, when run, it will hang DAQFactory and possibly your computer as it uses all the CPU time. Because of this, DAQFactory has an optional infinite loop check. The infinite loop check ensures that no more than 100 sequence steps execute in any one second. If more than 100 steps do execute in a second, the sequence throws an error and stops. So if you were to run the first sequence, the infinite loop check would very quickly detect the problem and stop the sequence.

The loop check is turned off by default as it can stop valid loops. To turn it on or adjust the settings, go to **File-Preferences** in the main menu. To turn on the infinite loop check, simply check the box labelled **Sequence Loop Check?**. If you want to increase the number of allowable steps, adjust the **Loop Check Count** to your desired number of steps. Remember though that, depending on your computer, sequences can execute at least 10,000 steps in a second, so setting the Loop Check Count to a huge number like 100,000 is the same as turning the loop check off.

If you are a beginning programmer we recommend turning the loop check on until valid sequences start triggering it, then try increasing the loop check count first before finally disabling the loop check. If you keep the loop check disabled, make sure and save your document before starting any new sequences, just in case you created an infinite loop and end up hanging your computer.

5.17 Using sequences as functions

Sequences typically run standalone in their own thread. But sequences can also act as functions, being called from another sequence or even an expression. In this case, the sequence runs in the thread of the calling sequence and the calling sequence waits until the function sequence is complete. To call a sequence from another sequence simply name the sequence followed by a list of optional arguments in parenthesis:

```
SequenceName(Input,3)
```

or

```
SequenceName()
```

The arguments appear in the called sequence as Private variables. By default, the arguments are named arg0, arg1, etc.. but you can tell DAQFactory to name them something different by putting a function declaration line at the top of the sequence. For example:

"AddTwo" sequence:

```
function AddTwo(valuein)
  return(valuein + 2)
```

This is a simple function that takes an argument named "valuein", adds to this value and returns the result. We can then call it from another sequence:

"Main" sequence

```
private X = 1
X = AddTwo(X)
? X   // displays 3
```

This sequence creates a variable named X, passes it as a parameter to our function, and sets the result back in X. Finally it prints the value of X to the command window.

The function declaration includes the reserved word "function", followed by the name of the sequence, and then a comma delimited list of argument names in parenthesis. If the argument is a string argument you should add the word string before the argument name:

```
function MyFunc(NumVar, string MyStringVar)
```

You can only have one function per sequence.

If you do not provide a function declaration, or you pass more arguments to a function then specified in the function declaration, the default names of arg0, arg1, etc. are used. So, our AddTwo sequence above without a function declaration becomes:

"AddTwo" sequence:

```
return(arg0 + 2)
```

It is recommended that all functions have a function declaration so that your arguments have names that make sense. The exception would be when you want to create a function that takes a variable number of arguments. There are several different situations where this occurs:

1) You want to create a function with optional arguments. In this case, you will want to put all the arguments in the function declaration, and then examine any optional ones to see if they were passed by using the isempty() function and setting the default value if not. Note that all the optional arguments must be at the end of the argument list:

```
function IncrementBy(x, by)
if (isempty(by))
  private by = 1
endif
return(x+by)
```

In this example, IncrementBy(4) will return 5 because "by" is optional and defaults to 1. IncrementBy(4,2), however, will return 6 because "by" was specified.

2) You want to create a function with a variable number of arguments. For this an additional private is also created inside of every function called "argc" which contains the number of arguments passed. So:

```
function SumVars()
  switch
    case (argc == 2)
      return(arg0 + arg1)
    case (argc == 3)
      return(arg0 + arg1 + arg2)
    default
      System.ErrorMessage("Invalid # of parameters")
    endcase
```

This sample looks at the number of arguments and returns the sum of 2 or 3 values or an error. Notice how we avoided the name "Sum" as a function. This is a reserved DAQFactory function and cannot be used as a sequence name.

3) You want to create a function with several defined arguments and a variable number of arguments in addition to the defined arguments.

```
function PrintVars(string Prefix)
  switch
    case (argc == 2)
      return(Prefix + DoubleToStr(arg1 + arg2))
    case (argc == 3)
      return(Prefix + DoubleToStr(arg1 + arg2 + arg3))
    default
      System.ErrorMessage("Invalid # of parameters")
    endcase
```

Notice that are arg notation starts at 1 because Prefix takes the 0 spot. However, the argc argument total still includes Prefix.

Note: All arguments are passed by value and not by reference. For example, calling our first example above: AddTwo(ChannelX[0]), the value of ChannelX[0] is passed to the AddTwo function, not ChannelX[0] itself. Changes

to valuein / arg0 in the AddTwo function do not affect ChannelX. Only by assigning the return value of AddTwo to ChannelX are we able to change ChannelX as seen in the Main sequence above.

Note: You can pass a maximum of 20 arguments to a sequence.

5.18 Running commands in a string with the execute function

There are many cases where you may need to be able to dynamically create code from within a sequence and execute it. The Evaluate() function allows you do this in an expression, but evaluate is an expression and cannot execute commands like x = 3. For that, there is the Execute() function. The execute function treats the string passed to it as sequence script and runs it. So, for example:

```
execute("MyChannel = 3")
```

will set the value of MyChannel to 3. Of course you could have just done MyChannel = 3 directly. Where the advantage comes is when you need to assemble the script while running. For example, lets say you have 50 channels named Ch1, Ch2, Ch3,... and you want to set the first n channels to 4:

```
for (private.x = 0, x < n, x++)
 execute("Ch"+DoubleToStr(x)+" = 4")
endfor
```

Since you don't know what n is when you wrote the script, you couldn't do this without execute.

Execute can actually run more than one line separating each line with a Chr(10). For example:

```
private strScript
strScript = "global x = 3" + chr(10)
strScript += "global y = 4" + chr(10)

execute(strScript)
```

Execute allows a lot of flexibility, but really should only be used when necessary for two reasons: 1) it runs slower then regular script because each time the execute() function runs, it has to compile the sequence, and 2) it can make your code harder to read.

Just a reminder: the Evaluate() function and the Execute() function are nearly the same, except that Evaluate() must be on the right side of an equal sign or in a place that expects an expression, while Execute() can be anywhere (though really should not be in places that expect an expression). If you put Execute in a place that expects an expression like a screen component, it will always return an empty value. Evaluate returns the result of the expression in the string parameter.

5.19 Error handling with try/catch

Unfortunately, unless your sequences are rather simple, errors will probably occur in your sequences. Something as simple as a device getting unplugged could generate an error. Older versions of DAQFactory sequences would simply ignore any errors and continue with the sequence. Unfortunately, this often made sequences unpredictable and difficult to debug. For this reason, sequences will now generate an error when an error occurs. The error must then either be caught by your sequence code, or the sequence will stop and display an alert. The try and catch statements are used to catch errors:

```
try
  do something here...
catch()
  do something if an error occurs
endcatch
```

The try statement marks the beginning of a block of steps that may throw an error. If an error occurs after the try statement, DAQFactory will jump to the catch statement and execute the steps immediately following it, if any, and then proceed with the steps after the endcatch.

Catch statements can also be set to catch particular errors. Since all errors within DAQFactory start with a code, you can specify the code of the error to catch:

```
try
  nonexistantchannel = 5
catch("C1001")
  do something if error code C1001 occurs
endcatch
catch()
  do something if any other error occurs
endcatch
```

In this case we can do two different things depending on the error that actually occurs. Only the first catch that matches will execute. The string specified in the catch statement must simply match the same number of characters as the error itself. So 'catch("C")' would catch all errors that start with the letter C.

Try/Catch statements can be nested as needed. They will also carry through to called functions. If an error occurs in a called function and that function does not handle it, the function will return and the calling sequence will have to handle the error.

You can access the text of the caught error with the private variable strLastError. For example, you could generate an error message:

```
catch()
  System.ErrorMessage(strLastError)
endcatch
```

You can also generate your own errors with the throw statement. This can be used as a way to branch, or as a way of creating your own codes:

```
try
...
  if (Input[0] > 5)
    throw("U1001 Input went out of bounds")
  endif
  ...
catch("U1001")
endcatch
```

If the error is not caught it will be treated like any other error and stop the sequence and generate an alert.

The error string inside the throw is optional. If the throw statement is inside a catch, it will simply rethrow the same error and is the same as **`throw(strLastError)`**. If it is outside the catch it will generate a default "User Specified Error".

Note: If you have multiple catch statements with a single try and you throw in the first catch, the other catch statements may catch the error you just generated:

```
try
  ...
catch("C1001")
  throw("my error")
endcatch
catch()
endcatch
```

In the above example, if error code "C1001" occurs inside the try, the first catch statement will handle it. This will then execute the throw which will throw another error. But, there is another catch left that catches all errors and this will then catch our new "my error".

Ignoring errors

You can optionally turn off the error catching mechanism by using the ignore statement. You can ignore certain types of errors or all errors. To ignore all errors, simply put **`ignore("all")`** at the point you want to start ignoring errors. For example:

```
...
ignore("all")
...  // at this point any step that generates an error is simply ignored and execution proceeds to the next step.
```

If you just want to ignore certain errors, you can specify the error code, much like the catch statement:

```
...
ignore("C1001")
... // now only the error C1001 is ignored, all other errors require catching.
```

Again, like the catch statement, only as many characters as specified in the ignore statement are compared to the error. For example, to ignore all standard errors which start with the letter C:

```
...
ignore("C")
... // all errors that begin with the letter C will be ignored.
```

The ignore statement does not carry into sequence function calls.

```
...
ignore("all")
...
MyOtherSequence()  // any errors in the MyOtherSequence will throw an error causing execution to
return to this sequence, at which point the error is ignored.
...
```

You can ignore multiple errors by passing an array of strings:

```
...
ignore({"C1001","C1002"})
... // errors C1001 and C1002 are ignored
```

To clear the ignore statement, simply pass an empty string:

```
...
ignore("all")
... // all errors ignored
ignore("")
... // no errors ignored
```

5.21 Sequence Threads and Thread Priority

When sequences are started they run in their own thread. A thread is like a separate program running concurrently with other threads inside of DAQFactory. Because each sequence runs in its own thread, they can all perform different tasks at the same time. Sequences called as functions either from another sequence or expression run in the thread of the calling sequence or expression. Events for channels and PID loops run in the thread of the loop or acquisition. The quick sequence action of many components always runs in the one and only main application thread. The main application thread is the thread that the user interface of DAQFactory runs in. This is where the mouse gets moved, components get displayed, and menu items get processed. The reason this distinction is important is two fold:

1) sequences running in the main application thread should be fast otherwise they will cause the user interface to be sluggish. This is because the sequence steps have to be executed before the rest of the user interface can respond. If you have wait or delay statements in your sequence then you can give the user interface time to respond. Even a wait(0) will help. Wait(0) tells windows to go do something else for a bit and then come back when other things are done.

2) some functions, such as System.NoteDialog(), only work in the main application thread. These functions will generate an error if they are called from a secondary thread.

Threads within DAQFactory and Windows have priority levels. This determines which threads get access to the CPU first when requested. By default, sequences run at the highest priority within DAQFactory along side acquisition loops. This means that if you write an infinite loop in a sequence or anything else that would use a lot of processor power, it will hang everything in DAQFactory but the data acquisition loops. The data acquisition loops have the same priority so share in the CPU usage. The other parts of DAQFactory run at lower priority and are starved out by your sequence. A sequence that is waiting does not require the CPU and allows other lower priority threads to run, so adding delay or wait statements inside your sequence will avoid this problem.

Alternatively, you can lower the priority level of your sequence. The thread priority is an option at the top of the Sequence view. There are 6 different levels, 0 through 5. The level of different parts of DAQFactory are displayed when selecting the thread priority. In general, leave the priority at its default value. But if you want to create a sequence that uses a lot of processor power, for example a big while loop analyzing a large data set, put the thread priority at 0 and your sequence will run after DAQFactory is finished with all other tasks. The following loop will

actually work fine if the sequence thread priority is set to 0:

```
while(1)
endwhile
```

This is an infinite loop that does nothing but use up the CPU. But, if the thread priority is set to 0, it only uses left over CPU power after Windows and DAQFactory are done using the CPU.

Tech Note: The 0 priority level is the windows idle priority and runs after everything else in windows. Other levels are intermingled with other applications, though the highest couple of levels will be above most other applications.

5.22 Auto-Start

If you want a sequence to start automatically when your document loads, you can check the **Auto-Start** box in the sequence view. Any sequences with the auto-start selected will start as soon as the document completes loading. If you have multiple sequences with auto-start selected, you cannot determine the order in which the sequences will start. If you need to know the order, set one sequence to auto-start and then have that sequence start the other sequences.

5.23 Sequence debugging

The sequence text editor contains an integrated debugger. The debugger allows you to set breakpoints which will temporarily stop the sequence at a particular point so you can examine the state at that time. Breakpoints can be set by selecting **Debug-Toggle Breakpoint** from the main menu when you have the cursor on the desired line. You can set as many breakpoints as you want and execution of a sequence will stop at that point. Once stopped, there are several step options to run through the sequence one step at a time:

Step In: executes the current step, and if that step is a sequence function, steps to the first step of that function

Step Out: executes all the steps left in the current sequence function and then stops once control is returned to the calling sequence

Step Over: executes the current step, and if that step is a sequence function, executes entire function without stopping, returning control when that function returns (unless of course you have a breakpoint in that function)

There is also:

Continue: starts the sequence back up from the current position until another breakpoint is reached or the sequence stops.

Stop: stops the sequence without executing any more steps

Restart: starts the sequence over from the top

Display Current Line: this will cause DAQFactory to display the current line about to be executed. DAQFactory does not automatically do this to allow you to view what is happening on your pages as the sequence is being stepped through.

Clear all breakpoints: this clears all breakpoints on the displayed sequence.

Note: sequences can only be debugged when they are running in their own thread (they were started with a Begin Sequence). A sequence used in an expression in Page Component, a quick action, an event, or elsewhere in DAQFactory will not break at any breakpoints and cannot be debugged.

5.24 Printing to the Command / Alert window

Although you can use the integrated debugger to find problems in your code, there are many cases where this is not possible or is inconvenient, but it would still be nice to know what the value of a particular variable is at particular point in code, or some other information. For that, you can use the print command to display the result of an

expression to the Command / Alert window. The format is the exact same as you would enter in the command line in this window using the ?. For example:

```
? MyVariable

? "Reached point y in sequence"

? "Value of MyVar is " + DoubleToStr(MyVar)
```

In general, you will probably only want to use the ? statement for debugging as it is somewhat slow. For this reason, DAQFactory provides a simple way to completely disable all the ? statements in all your sequences. The variable, "System.SequencePrint", when set to 0, will cause all the ? statements to be skipped. When set to 1, the default, the ? statements work as described and display to the Command / Alert window. This allows you to liter your code with ? statements to help you debug, and then when everything works, you can simply set `System.SequencePrint = 0` in a start up sequence to prevent the ? statements from executing and slowing your code.

Note: some sequence problems are caused by timing issues. Having ? statements in your code could change the timing of the execution of various steps of your code resulting in code that works with ? statements enabled, but stops working when disabled. If this happens you should immediately assume that your problem has to do with the timing of your loops.

5.25 Watch window

In addition to the integrated debugger, DAQFactory has a watch window that allows you to quickly view the results of any expressions. Of course this same thing can be done using display variable value components on pages, but the watch window is often much more convenient, and as a docking window, remains visible no matter which page is displayed. Also, the watch window allows you to view private variables when a sequence is being debugged. The privates of the sequence function currently being debugged are only the only privates visible.

The watch window contains a table with 100 rows. In any row, enter an expression in the **Watch** column. The result will be displayed in the **Value** column of the same row. This table updates about twice a second with the result of all the expressions. To clear out a watch, simply erase the **Watch** expression.

Note: For efficiency and responsiveness reasons, expressions that result in an array with more than 20 values will only display the first 20 values.

5.26 Sequence Functions

Sequences have functions and variables for programmatic access. "SequenceName" below refers to a particular sequence you wish access to.

Functions:

Sequence.StopAllSequences(): a quick function to stop all running sequences. If you want to stop all sequences, logging, PID loops and acquisition, use System.SafeMode to switch DAQFactory into safe mode.

Variables:

Sequence.SequenceName.Running: 0 or 1 on whether the sequence is running. A sequence called from another sequence is not considered running. Only a sequence that has been started with Begin Sequence in its own thread is considered running. Read only.

Sequence.SequenceName.CurrentLine: a numeric with the current line being executed. Read only.

Sequence.SequenceName.strCurrentSequence: a string with the name of the sequence currently running. If a sequence calls another sequence, this variable will contain the name of that sequence. Read only.

5.27 System Functions

There are quite a few useful system functions and variables available from within sequences to help you develop your application.

SendMessage(Message, [Param1], [Param2]): this function sends a message to all the screen components that are listening for the given message. Please see the section in Pages and Components that discusses user components and component functions for more information.

StartThread(Function, [Name = ""], [Priority = 3]): this function will start the given function (specified as a string) in a new thread. Its similar to beginseq() except that you can create multiple threads for a single sequence function, and you can pass parameters when starting the thread. The disadvantage of using this method is that you have no feedback or control over the thread except through the following two functions. There is no indication in the workspace that your thread is running and you cannot debug the thread. Function should be a string with the function call with parameters. Name can be any valid DAQFactory name and is used for stopping or getting the status of the thread. You should make sure your threads have unique names. You do not have to name your thread, but you won't be able to stop or get the status of the thread. The function will either need to reach its end, which kills the thread, or you'll need to quit DAQFactory or load another document. Priority matches the numbers listed in the drop down when editing a sequence, where 0 is the lowest, and 5 the highest priority. Example: `StartThread("MyFunction(x,3)",MyThread,5)` Note that the function call MUST be a string.

StopThread(Name): Stops the given thread if it is running. Throws an error if it can't find the thread.

GetThreadStatus(Name): Returns a string with the current status of the given thread, or a message indicating that the thread doesn't exist. The current status string is similar to the string in the sequence table and includes the current function and line being executed.

Note: when creating a user component, you can also use the StartLocalThread(), StopLocalThread() and GetLocalThreadStatus() functions. These functions work the exact same way, but the thread runs in the context of the component and will automatically terminate if the component is deleted. You can also use the same thread name in different components. User components can use the above global functions, but the created thread will run in the global context and won't be encapsulated in the user component.

The following functions all start with "System.". Some of these functions must be called from the main application thread, which means they really can only be called from the Quick Sequence action of many components.

Variables:

System.AppVersion: set or read the version number of your application. You can use this to do versioning of your application. By default, a new application is assigned version number 1.0 and every time you save, it will increment this number by 0.01. You can alternatively set this variable to any version number you want. The first time you save after assigning this variable a value, the version number won't be incremented and the value you specified will be used. After that, the number will be automatically incremented.

System.DisplayCursor: set to display or hide the Windows cursor. This is useful for touch screen applications where a cursor is not desired. Set to 1 (the default) to display the cursor. Set to 0 to hide it. Note that this does not affect cursors in other applications, just DAQFactory.

System.FullScreen: set to 1 to switch to full screen, and to 0 to get out of full screen. This is a write-only variable.

System.OnScreenKeyboard: set to display or hide the onscreen keyboard. There are two different keyboards, one that displays as a docking window and works on pages, and another that appears with entry dialogs. Set the low bit of this variable to display the page keyboard, and bit #1 to display the dialog keyboard. In other words, 0 displays no keyboard, 1 displays the page keyboard, 2 the dialog keyboard, and 3, both keyboards.

System.SafeMode: set to 1 or 0 to switch into and out of safe mode. Can be read or written to.

System.SequencePrint: set to 1 (the default) to enable the ? statement in a sequence. Set to 0 to have all ? statements in all sequences ignored.

Functions:

System.Quit(): call this function to quit DAQFactory.

System.Reboot(): call this function to reboot the local computer. There is way to stop the reboot, and a 30 second delay before it occurs.

System.Shutdown(): call this function to shutdown the local computer. There is way to stop the shutdown, and a 30 second delay before it occurs.

System.IsDST(): call this function to retrieve the current Daylight Savings Time setting of the system. Returns 1 if DST would be in effect for the current DAQFactory time, and 0 if not. Please note: DAQFactory does not adjust for DST no matter what the system settings. DAQFactory picks up the Windows system time at startup and then after that uses a high precision counter to determine the time. Any changes to the Windows system time, including DST, will not affect DAQFactory time. If DST is in effect when DAQFactory starts, all time in DAQFactory will be DST until you restart. The IsDST() function simply looks at the current DAQFactory time and determines from it if it is the time of year when DST would be in effect or not.

System.IsRuntime(): call this function to determine if we are in runtime mode or not. Returns 1 if in runtime, 0 if in development.

System.SaveDocument(): calling this function does the same thing as hitting File - Save from the main menu. It is designed to be used in runtime mode to save user component properties, and thus make user runtime configurable components. This function is not supported in DAQFactory Express.

System.SaveWithData(): calling this function does the same thing as hitting File - Save with Data from the main menu. This function is not supported in DAQFactory Express.

System.HideSystemMenu(): call this function to hide the system menu from the title bar of DAQFactory. This would mostly be used in Runtime mode to keep a user from exiting the application. Make sure you provide an on-screen button or other method to quit your application.

System.Maximize(): call this function to expand the DAQFactory window to take up the entire screen, similar to clicking the Maximize button on the window. This is not the same as Full Screen mode which doesn't display the Windows title bar.

System.Minimize(): call this function to minimize your DAQFactory window.

System.DeviceConfigDialog(): call this function to display the Device Configuration dialog. This must be done from the main application thread.

System.Connection.Add(Name,Address,[Password],[Fullstream]): the last two parameters are optional and default to blank and 0. Creates a new connection (provided one with the same name does not already exist) with the given name, address, password and fullstream setting (0 or 1)

System.Connection.Reset(Name): Resets the given connection. Closes the connection then attempts to reconnect. This does the same thing as clicking the Reset Connection button in the connection view.

System.Connection.Delete(Name): deletes the named connection if it exists. You can't delete the Local or V connection. If you want to create a runtime that connects to a computer with a varying IP address, call Delete followed by Add with the new IP address and same connection name.

Application: The above three functions can be used in the runtime version of DAQFactory to connect to an unknown remote computer.

System.Connection.Default(Name): sets the default connection to the given Name. For example:
```
System.Connection.Default("Local")
```

System.ShellExecute(File, [Operation], [Parameter], [Path], [Show]): Performs a system shell command allowing you to start other programs from within DAQFactory. This calls the Windows API ShellExecute command.

File can be an executable file, folder, or a document provided the document extension is assigned to an application (i.e. MyDoc.txt would probably run Notepad). File is the only required parameter.

Operation is a string and can be:
"edit": launches an editor and opens the file for editing. File must be a document and not an executable.
"explore": opens the folder specified in File in Explorer.

"open": opens the specified File. In this case, the file can be an executable (which will be run), a document or folder. This is the default.
"print": prints the document specified by File. File must be a document.

Parameter is used if File is an executable and determines which command line parameters are passed to the executable. It is a string and defaults empty.

Path is the default directory used. It is a string and defaults empty.

Show determines how the shell will display itself. It is a string and can be:
"hide": does not display the executable at all, not even in the Windows task bar. If the executable does not automatically terminate itself, the only way to stop it is using the Processes tab of the Windows Task Manager. This setting is best used when printing documents.
"minimize": opens File, but minimizes it.
"maximize": opens File and maximizes the window.
"normal": opens File in whatever way it normally opens. This is the default setting.
"noactivate": same as normal, but DAQFactory remains on top and in focus.

System.Sound.Play(.wav file name): plays the given wave file. If a file is currently playing, that sound will stop and the new one will start

System.Sound.Stop(): stops any playing sound

System.DelayUpdate(): causes the system to stop redrawing the current page. This is useful if you want to move a number of components around at once without having the user see each individual component move. Use ResumeUpdate to continue drawing the screen.

System.ResumeUpdate(): allows the system to continue updating the current page. We strongly recommend using a try/catch block around a DelayUpdate / ResumeUpdate pair to ensure that pages are updated if there is an error:

```
try
System.DelayUpdate()
... do a bunch of stuff
System.ResumeUpdate()
catch()
System.ResumeUpdate()
endcatch
```

System.SetFocus(): forces the system focus back to the Page view of DAQFactory. It does not switch the view, so if you are not viewing a Page, this function doesn't do much. This function is primarily designed for use with the browser component. There are instances when the browser component will take focus away from the page it is on which will cause mouse and keyboard events that you might be trying to capture to be directed to the browser instead of to DAQFactory. Calling SetFocus will restore focus to the DAQFactory page and allow those events to be capture by your script.

System.ColorSelectDialog() : displays the standard color selection dialog and returns the selected color or an empty value if the user hits cancel.

System.EntryDialog(Prompt, [RangeMin], [RangeMax], [DefaultValue], [HelpFile]): displays the DAQFactory SetTo action dialog to request a value from the user. All parameters except Prompt are optional. The prompt is what is displayed in the box. RangeMax/Min are optional checks on the user input. Default value is the value initially displayed in the edit box. HelpFile is the name of an HTML help file to display below the dialog. This can either be a fully specified path, or relative to the DAQFactory installation. The resulting string value is returned, or an empty value if the user hits cancel.

System.HelpWindow(HTML file, [x1], [y1], [x2], [y2]): displays a window with the specified HTML file. X1, Y1, X2, Y2 determine the coordinates of the window. The default is centered.

System.NoteDialog(): this displays the quick note dialog just as if the user selected Quick-Note from the main menu. Nothing is returned. The result is added to the Note history. This must be called from the main application thread.

System.MessageBox(Message, [Type], [Help]): this displays a standard windows message box, or a DAQFactory style message box with help. The first parameter is required, the others are optional. The message is what is displayed in the box. The type can be one of the following:

"Help", "AbortRetryIgnore", "OKCancel", "RetryCancel","YesNo", or "YesNoCancel"

These determine what buttons are displayed. The default and "Help" just display an OK button. If the "Help" type is selected, the Help parameter should contain either the DAQFactory installation path relative or absolute path to an HTML help file to display. This function returns one of the following depending on which button the user presses:

"OK", "Cancel", "Ignore", "No", "Yes", "Abort", or "Retry"

System.ErrorMessage(Message): this generates an Alert within DAQFactory with the given message. This does not generate an error within a sequence, but just sends the given message to the Alert window. To generate a catchable error in sequence code, use throw().

Note: although you can call most of the above functions from a sequence, for the ones that require a user response (MessageBox, HelpWindow, EntryDialog and ColorSelectDialog) you must be very careful. First of all, having windows popup without the user clicking on something can be annoying to the user, but there are other issues:

Two big things to watch for are loops that repetitively call these functions, and using the wait() function. The best way to show the potential problem is with examples:

```
while (1)
  page.page_0.backcolor = System.ColorSelectDialog()
.. some other quick stuff
delay(1)
endwhile
```

In the above loop, the user will essentially be prompted for a color every second. Because the delay is so short, and the color select box is modal, meaning the user can't access the reset of DAQFactory while it is displayed, the user won't be able to stop the sequence because the color box will redisplay every second.

This is compounded more if you use a wait() function instead of delay because the wait() will go to 0 because the user interface is slow, and the user will be continually prompted and will not be able to escape. So, in general NEVER use wait() in a sequence that contains one of these user interface functions.

A few other things to think about:

- If another sequence stops a sequence that is waiting for a user response, the sequence will stop, but the window requesting the user response will remain. When the user closes this window, any return value is discarded

- If another sequence calls one of these functions while a window is already displayed, the new window will appear on top and will have to be dismissed before the user can get back to the first one.

System.MessagePump(): this function services the windows message pump. This is a somewhat advanced function. The windows message pump is what handles things like mouse clicks and the like. If you have a quick sequence in a screen component, or you call a sequence as a function from a screen component action, then the script runs in the primary thread of the application. This is the same thread that handles the message pump for DAQFactory, so while the script is running, DAQFactory will appear "hung" because it won't respond to mouse clicks or anything else. It isn't hung, its just busy running your script. You can avoid this by servicing the message pump rather often (say at least every 0.1 seconds) simply by calling this function. If your script is really fast and finishes quickly, than this function is not needed, but if you really want to do something that takes a while from the primary thread, this function is available.

5.28 System Events

For advanced users, DAQFactory exposes several system events. Most of these are somewhat advanced, and misuse can result in DAQFactory crashing, or other problems. Fortunately, the crash or other problem will probably occur immediately, so as long as you save often you should be OK even if you are not familiar with Windows event handling.

To create code that runs when the event occurs, simply create a sequence in the Local connection with the event name. Because most of these events run in the primary thread of DAQFactory, it is very important that your code be fast. Otherwise, the system will become sluggish, possibly even hanging DAQFactory. This is especially important for the OnMouseMove event, which is called around 50 times a second. The exceptions are probably the OnShutDown event as shut down is slow anyway, and the OnPreDraw event which will only slow graph and image rendering. For many of the events, you can avoid the default DAQFactory processing of the event by putting return (1) at the end of the sequence code.

OnAlert(): called when an alert occurs. An alert is anything that would appear in the command / alert window as an alert. This does not include things printed there using ?, nor does it include errors that are caught in code using try/catch. The private variable strAlert is passed into the event which is the exact text of the alert as it appears in the Command / Alert window, but without the timestamp.

OnHelp(): called when the user presses the F1 key invoking help or when you select Help-Help Topics. Windows often captures the F1 key before it calls OnChar() or OnKeyUp(), so this is a way to use the F1 key for another purpose. To avoid displaying the DAQFactory help, put return(1) at the end of the sequence to indicate that default handling should not occur.

OnShutDown(): called when DAQFactory exits, or when a document is closed. Because of where this occurs in the Windows event structure, you cannot perform any windows based tasks here. For example, you cannot display a message box. You can, however, do things like adjust outputs and other similar clean up. Any returned value is ignored. There is no way to stop the shut down procedure.

These events only apply when the page view is displayed and in focus. For applications running in Runtime mode, this is pretty much all the time.

OnChar(Char, Shift, Ctrl): called when the a key is pressed. The differences between this and the OnKeyUp () event are subtle. The OnKeyUp() event won't detect a repeated character when a key is held down, but it will better detect special keys like function keys. The Char parameter is the numeric key code generated; Shift is 1 if the shift key is down; Ctrl is 1 if the control key is down. Returning 1 from this event will cause DAQFactory character events to be skipped. Note that which events DAQFactory does with OnChar and which with OnKeyUp are not documented because they may change at any point. If you want to process all keystrokes, make sure and put return(1) at the end of both the OnChar() and OnKeyUp() events.

OnContextMenu(Loc, Shift, Ctrl): called when the context menu is triggered. Typically, unless the user has reassigned their mouse buttons, this is when the user right clicks. It also occurs when the popup button is pressed on newer keyboards. The Loc parameter is an array with two columns containing the current X and Y position of the mouse. Shift is 1 if the Shift key is pressed; Ctrl is 1 if the control key is pressed. Returning 1 from this event will cause DAQFactory processing of this event to be skipped.

OnKeyDown(Key, Shift, Ctrl): called when a key is pressed. The Key parameter is the keyboard scan code for the key; Shift is 1 if the shift key is down; Ctrl is 1 if the control key is down. Most of the normal keys match their ASCII equivalent. For other keys, you can search the internet for Windows key codes, or simply create an event that sets a global variable to Key, display that global variable in the watch, and simply try different keys. Returning 1 from this event will cause DAQFactory processing of this event to be skipped.

OnKeyUp(Key, Shift, Ctrl): called when a key is released. Everything else is identical to OnKeyDown()

OnLButtonDblClk(Loc, Shift, Ctrl): called when the left mouse button is double clicked. What constitutes a double click is determined by windows. The Loc parameter is an array with two columns containing the X and Y position of the mouse. Shift is 1 if the Shift key is pressed; Ctrl is 1 if the control key is pressed. Returning 1 from this event will cause DAQFactory processing of this event to be skipped.

OnLButtonDown(Loc, Shift, Ctrl): called when the left mouse button is pressed. The Loc parameter is an array with two columns containing the X and Y position of the mouse. Shift is 1 if the Shift key is pressed; Ctrl is 1 if the control key is pressed. Returning 1 from this event will cause DAQFactory processing of this event to be skipped.

OnLButtonUp(Loc, Shift, Ctrl): called when the left mouse button is released. The Loc parameter is an array with two columns containing the X and Y position of the mouse. Shift is 1 if the Shift key is pressed; Ctrl is 1 if the control key is pressed. Returning 1 from this event will cause DAQFactory processing of this event to be skipped.

OnMouseMove(Loc): called when the mouse moves on a page. The Loc parameter is an array with two columns containing the current X and Y position of the mouse. Returning 1 from this event will cause DAQFactory processing of this event to be skipped. It will not, however, cause Windows handling of this event to be skipped.

OnMouseWheel(Loc, Delta): called when the mouse wheel is moved while on a page. The Loc parameter is an array with two columns containing the current X and Y position of the mouse. Delta is typically either -120 or +120 depending on which direction the wheel is spun. Some mice with finer wheels may return smaller values. This event is typically triggered for each click of the wheel, but again, with finer wheels, the event may be triggered more often. Returning 1 from this event will cause DAQFactory processing of this event to be skipped. It will not, however, cause Windows handling of this event to be skipped.

OnPostDraw(DC): called after DAQFactory renders the current page. This event passes in a private variable named "DC" which you can use with the drawing functions as described in the Canvas Component.

OnPreDraw(DC): called before DAQFactory renders the current page. This event passes in a private variable named "DC" which you can use with the drawing functions as described in the Canvas Component. This is the one event in this list that runs in a secondary thread along with the drawing of graphs and images. If your routine is slow, it will not slow DAQFactory, but will slow the rendering of graphs and could cause other rendering abnormalities.

OnRButtonDblClk(Loc, Shift, Ctrl):
OnRButtonDown(Loc, Shift, Ctrl):
OnRButtonUp(Loc, Shift, Ctrl): these three events are identical to their LButton counterparts except they apply to the right mouse button.

5.29 Questions, Answers and Examples

5.29.1 Misc sequence comments

Here is some sample code that has a few errors, both real and stylistic. A list of issues follows it.

```
while (1)
    try

 Private r = {2,3,4,6,8,10,12,14,16}
 Private x = 0
 while (x<10)
rGEO[x] = sqrt(r[x]/2*r[x+1]/2)
 x++
 endwhile
 Private r2=rGEO^2
 Private r3=rGEO^3
 Private r4=rGEO^4

 Private spp200r = {.09,.1,.11,.12,.13,.14,.15,.16,.17}
 x = 0
 while (x<10)
 spprGEO[x] = sqrt(spp200r[x]/2*spp200r[x+1]/2)
 x++
 endwhile

 Private spp200r2=spprGEO^2
 Private spp200r3=spprGEO^3
 Private spp200r4=spprGEO^4

 LWC_calculation.AddValue(CIP_Hotwire[0]/r2)
 CDP_NumbConc.AddValue(CDP_100_Spectrum[0]/r1)
 // and more calcs based on the above private variables snipped

 catch()
   endcatch
   delay(1)
endwhile
```

1) Style: the indentation was done poorly, which makes the code hard to follow.

2) Error: both internal while() loops will end up indexing the arrays past their ends. There are only 9 values in both arrays, so r[9] is invalid. To avoid this error, which makes for better programming form, use: `while (x < NumRows (r))`

3) Style: for readability, a 0 should be placed in front of all decimal numbers: `0.09` not `.09` Without the 0 in front, the decimal gets lost. This is a scientific standard that for some reason is not taught in other fields, but is quite important.

4) Style: almost every value in this sequence is a constant or calculated from a constant. The only exceptions are

the two channels, **LWC_calculation** and **CDP_NumbConc**. For example, **r4** is calced from **rGEO** which is calced from **r** which is a constant array. It is terribly inefficient to be calculating the same values over and over again. Instead, a separate sequence should be created to pre-calc these values and that sequence called once when the application loads. These values could then be stored in global variables and used anywhere.

5) A for() loop is usually preferred to a while() loop when you are simply looping a certain number of times. Although they result in the same thing, using a for() loop keeps you from accidentally forgetting to increment the counter variable.

5.29.2 Array manipulation

The follow code snippets show how to use sequences to perform several different types of array operations within DAQFactory. The sequences shown here can be used in any application that involves the use of arrays. Be sure to take advantage of the comments located within each sequence for a better understanding of how the sequence code works.

AppendValue(array, value): this sequence function will simply add the given value to the end of the array:

```
// Gets the index value for the end of the array
Private Index = NumRows(arg0)
// Adds the new value at the end of the array
arg0[Index] = arg1
// Returns the modified array
return(arg0)
```

Of course its easier (and much faster) just to do it in place in one line then use this function:

```
array[numrows(array)] = newvalue
```

AddHistory(array, value): this does almost the same thing as AppendValue() except adds the value at the beginning of the array, moving all the other elements back. As of 5.7 you can use the AddValue() function of a variable to do this for you.

```
// move everything out of the way
arg0[1] = arg0
// and put new value in place
arg0[0] = arg1
return(arg0)
```

DeleteValue(array, index): removes the given item out of the array.

```
// Gets the index value for the end of the array
Private ArrayEnd = NumRows(arg0) - 1
switch
case ((arg0 > 0) && (arg0 < ArrayEnd))
 // Checks to see if the value to be deleted is not at the beginning or the end.
 // Deletes the value located at the specified index, by concatenating the array indexed
 // from 0 to the delete value minus one with the array indexed from the delete value plus
 // one to the end of the array.
 arg0 = Concat(arg0[0,(arg1 - 1)], arg0[(arg1 + 1),ArrayEnd])
case (arg1 == ArrayEnd)
  // Checks to see if value to be deleted is located at the end
  // Deletes the last value of the array by setting the array
  // equal to the array minus the last index value.
  arg0 =  arg0[0,arg1 - 1]
case (arg1 == 0)
  // Checks to see if the value to be deleted is at the beginning of the array.
  // Deletes the first value of the array be setting the array
  // equal to the array minus the first index value.
  arg0 = arg0[1,ArrayEnd]
endcase
// Returns the modified array
return(arg0)
```

FindValue(array, value): searches through the array for the value and returns the index of the first occurrence.

As of 5.70 we now have a search() function that does this for us. All we do is **`Search(array == value)`**. That said, the old method shows us some nice tricks with arrays that may apply to other cases, so definitely read on if you are interested:

There are two ways to do this, the slow way...:

```
// The for loop searches through the array until it finds the specified value.
Private a
Private FoundAt = -1
for(a = 0, a < NumRows(arg0), a++)
// If the value is found perform the following code.
if (arg1 == arg0[a])
  // Sets the variable FoundAt to the index of the array where the value was found.
  FoundAt = a
  // Since the value was found break out of the loop
  break
endif
endfor
return(FoundAt)
```

and faster way:

```
Private FoundAt = GetTime(Max(InsertTime(arg0 == arg1,0,1)))
if ((FoundAt == 0) && (arg0[0] != arg1))
FoundAt = -1
endif
return(FoundAt)
```

For example, if the array had 100 values and the value you were searching for was the last value in the array, on our test computer the first slow way takes 16 milliseconds. The second way takes 0.8 milliseconds. And it gets worse with bigger arrays. If the array is 1000 values, the first method takes 183 milliseconds, while the second way takes 1.9. So while the first method took slightly more than 10 times as long to search 10 times as many values, the second fast way took only slightly more than twice as long. Go up to 10,000 values and its 4.5 seconds vs 5 milliseconds, a factor of almost 1000!

How the fast way is done requires a bit more explanation. If possible you always want to avoid doing loops in DAQFactory. DAQFactory is a scripted language, and therefore is not particularly fast. However, the built in functions provided are all written in C and are therefore VERY fast. The trick is figuring out how to use the functions to get what you want. In this case we want to find a value in an array and return the index. So:

1) arg0 == arg1 : this returns an array of 1's and 0's the same size as arg0 (remember arg0 is the array, and arg1 the value). There will be a 1 everywhere the value in arg0 equaled arg1. For example: {1,4,6,2,6} == 6 would return {0,0,1,0,1}.

2) InsertTime(...,0,1) : the insert time function is normally used to insert time into a value. Variables in DAQFactory are actually made up of two arrays, one with the data, and one with the time. In this example we don't need the time, so we are going to use that second array to store the index into the array. The 0 means start at 0, and the 1 means increment by one for each index in the array. We'll use this in a minute...

3) Max(...) : the max function finds the maximum value in an array. In this case, our array is 0's and 1's so it will return 1 if the desired value is in the array, and 0 if it is not. Where Max becomes handy is in the next step...

4) GetTime(...) : the Max() function, in addition to returning the maximum value in an array, also returns the time of the first instance of that maximum value. In this case, we've inserted the index of each point into the time part of our array, so the Max() function is actually returning the index as well. This index is, however, in the time part of the result, so we have to use the GetTime() function to retrieve it. At this point, we have the index of the data point in the array that equals our desired value. But we're not quite done...

5) What happens if the value doesn't exist in the array? In this case, arg0==arg1 would return an array of 0's, Max would return 0 and GetTime() would also return 0. So, if our first line sets FoundAt to 0 one of two things happened. Either the value actually exists at index 0, or it was not found. So, we just do an if to see. If FoundAt is 0 and the first point in the array does not equal the desired value then the value doesn't exist, so we set FoundAt to -1 to indicate no find. Now we are done.

There is actually another way to do step 5: **`if (Max(arg0 == arg1) == 0)`** This is probably slower though, at least when the arrays get bigger since the original method only examines two values, while the == and Max both would have to examine every value in the array.

One note: this function does not work on two or three dimensional arrays because time is only tracked along the first

dimension. That said, there are still ways to search quickly without using a for() loop. How this is done depends on whether you want to return the row with desired value or the column, or if both, which gets priority. The concepts are the same, except you'd probably use the MaxCols() or MaxDepth() function and then use the Transpose() function to get the array into the Rows dimension to use the Time indexing trick.

That all said, there are certainly plenty of cases where a loop is required, so don't think that you can't use loops in DAQFactory. Just try and use DAQFactory's built in functions whenever possible.

InsertValue(Array, Index, Value): Inserts the given value into the array at the given index, pushing anything above it out of the way

```
// Gets the index value for the end of the array
Private ArrayEnd = NumRows(arg0) - 1
// Shifts down the values to create space for the new value
arg0[arg1 + 1] = arg0[arg1,ArrayEnd]
// Inserts the value into the array before the insert value index
arg0[arg1] = arg2
return(arg0)
```

5.29.3 Start a sequence on the second

Question:

How do I get a sequence to start on the x.000 point of the second rather than whenever I happen to start the sequence?

Answer:

To start a sequence at the second, just add this to the first line of the sequence:

```
waituntil(floor(systime()) + 1)
```

waituntil() waits until the specified time. By taking the floor() of systime() we get the .000 of the current second. We then add one to wait until the beginning of the next second.

6 Channels and Conversions

6 Channels and Conversions

6.1 Channel and Conversion Overview

Channels provide an identifier for your data. A Channel typically is a single I/O point, be it input, output, string, spectral data or even an image. To make life easier, the details of which I/O channel number and device correspond to what channel only need be entered once. Each channel is given a name, and after defining the I/O associated with that name, only this name is required throughout the rest of the program. For instance, there is no need to remember that ambient_temperature channel is attached to Acme I/O board, device #3, channel #2! There can also be channels that contain calculated values: latched values from sequences, results from analysis functions, even calculated formulas. These are discussed a bit more in the section entitled Virtual Channels.

Conversions provide a simple way to put your data into useful units. You can, of course, use calculations throughout DAQFactory to do just about anything with your data. Conversions provide the first step, taking data that may be in volts, amps, counts, or some other units and converting them into something a bit more useful, like degrees, psi, etc.

This chapter will provide most of the detail needed to create channels and conversions and how to adjust them for the best performance.

6.2 Channels and the Channel Table

Channels contain all the information required to take the data and keep a part of that data in memory for display and analysis. Fortunately, once you setup the parameters of a channel, you only need refer to the channel by name throughout DAQFactory. Channel parameters are stored in a Connection, typically **Local**.

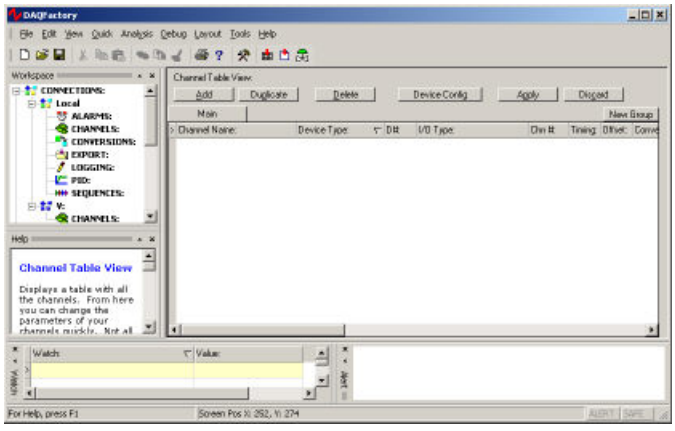
• To display the channel table, simply click on the word **CHANNELS:** under the connection **Local** in the workspace. The view will change and display the channel table.

• To add a new channel, simply click the **Add** button. This will create a new row in the table. Here you can fill in the details for the new channel.

• To enter multiple channels quickly, use the **Duplicate** button. The **Duplicate** button will also create a new row, but copies all the data from the current row (the one marked with the > and highlighted in yellow). The channel number is automatically incremented by one each time the **Duplicate** button is pressed. The channel name column is not copied unless the channel name ends in a number, in which case this number is also automatically incremented with each **Duplicate** press. The Quick Note column, which is often used for addressing special devices is duplicated, and if it ends in a number, that number is incremented as well. This makes it very easy to enter multiple channels quickly.

• To delete a channel, select the row with the channel by clicking on it. The selected row displays in yellow with a **>** mark along the left of the table. Once the desired channel is selected, click the **Delete** button to remove it.

None of your changes are permanent until you click on the **Apply** button. If you don't want to keep your changes, click the **Discard** button instead.



The channel table rows:

Each row of the table corresponds to a single Channel and has several columns.

Channel Name: This contains the name of this channel which will be referenced through DAQFactory for displaying and analyzing data. Like all names in DAQFactory, the channel name must start with a letter, and contain only letters, numbers and the underscore. DAQFactory will prompt you when applying your changes if you have any invalid channel names. We suggest using a descriptive name. Consider either separating multiple word names with an underscore like: `Inlet_Temperature`, or using upper lower format like: `InletTemperature`.

Device Type: Here you will select the type of hardware device. The list of possible choices depends on what device drivers are installed on your system. Typically, these refer to a hardware manufacturer, or special type of I/O (like Serial).

D#: This refers to the device number. Only certain device types use this parameter. Refer to the documentation on your particular device type to determine the appropriate values.

I/O Type: This helps define the class of data being received and has different meanings depending on your device. For example, most DAQ boards will have "A to D", "D to A", etc.

Channel #: This specifies the exact I/O channel number. Again this is device type specific, but typically refers directly to the channel described on your hardware's pinout. Some devices, OPC for example, do not use this parameter.

Timing: The timing value determines how often an input channel is read. It has no meaning and cannot be entered for output I/O types. The units of timing are in seconds. You can enter a value of 0 for timing, in which case the channel will only be read when a Sequence read channel command, read(), occurs on that channel or through other device specific methods.

Offset: There are cases when your hardware I/O might be somewhat slow. In this case, you probably do not want to hold up your other I/O for this slow hardware. By providing an offset value for your slow hardware you can have all your data read at the same timing interval, but stagger the reads in a predictable manner to allow the slow hardware to finish without holding up the fast I/O. For example, you could set all your fast I/O channels to a timing value of 1.0 and an offset of 0.0. Then you would set your one slow channel to a timing of 1.0 and an offset of 0.1. This way your fast I/O will always occur right on the second, and your slow I/O will not start until 100 ms later, which should give you fast I/O time to complete. The units for this column are also in seconds.

Conversion: Conversions allow you to put your data into more useful units. Conversions are discussed a little bit later on. In this column you will select from the list of available conversions, or select **None** to leave your data in raw form. Remember, you can always put your data in just about any units from within DAQFactory as well.

History: To allow you to display graphs and do data analysis, DAQFactory keeps a history of the data from all your channels in memory. Unfortunately, computers do not have an unlimited memory, and rather than assume some arbitrary limit to the amount of data that is remembered, the history column allows you to specify the number of data points that are kept in memory. You can therefore set the history amount to a large number for your important channels and a smaller number for the less important channels. As a reference, each data point within DAQFactory uses 16 bytes of memory. So a history of 86400, or a days worth of 1 hz data, will take just under 1.4 meg of memory. The same is true for image and spectral data, except that you have to multiply the history by the n umber of data points per spectrum or image. All memory is allocated as soon as the first data point is added to the channel. Use this in combination with Persist to extend the amount of data available to your graphs and calculations.

Persist: In addition to the history and separate from data logging, you can opt to keep a larger amount of historic data from a channel on your hard disk for easy retrieval. The Persist field specifies how many data points are kept on disk. Since hard disks are significantly larger than RAM, you can make this number very large and thus have quick access to lots of data. Please see the next section on channel persistence for a more thorough discussion of this parameter and how it works with History. This parameter is optional and defaults to 0, which means only the channels history in memory is used for graphs. Note that channel persistence does not work with string, spectral or image type data.

Avg?: If this is checked, then the data being stored to disk will be averaged first. This is a simple way to reduce the amount of data being stored without ruining the precision of your measurements. The amount of averaging is determined by the next column.

 # Avg: This column determines the number of data points that will be averaged before the average is stored. The average is a boxcar average, not a running average. This column can only be edited when the previous column is

checked.

Quick Note / Special / OPC: This column can have different meanings depending on the device type and I/O type specified. For most channels, the column simply contains the Quick Note where you can specify short details about the channel. For some I/O types on certain legacy devices, an ellipsis (...) button will be displayed. If you click on the ellipsis, a dialog will appear allowing you to set certain parameters for the channel. The Command I/O type will almost always have parameters. See the documentation on the device driver for more details on exact commands. For OPC devices, an ellipsis will also appear. Clicking on this will open a dialog that allows you to browse for your OPC tags. Once selected, the tag will be displayed in this column in the format: server;host;tag. You can always manually enter these values in the table. Please see the section on OPC for more information. Other devices may use this column for other channel specific details.

DAQConn?, determines if the channel data is being sent to DAQConnect web service. Check this box to mark the channel. Please see the section on DAQConnect and www.daqconnect.com for more information.

DC Hst: determines how many data points are kept on the DAQConnect server for historical viewing. This number is limited by the plan you select. You can enter -1 to have DAQConnect automatically apply an even amount to all of your tags based on your plan, or you can specify exact amounts for each channel allowing you to have more data retained for certain channels.

DC Intvl: determines how often the channel data is sent up to DAQConnect. This is in data points. So, if you are taking data once a second and you put a value of 10 in this column, every 10th data point is sent. This is to allow you to take data at a faster rate than it is sent to DAQConnect. Please note that DAQConnect may limit the spacing of data points being sent to it. If you need to send high speed data such as streamed data, you should contact DAQConnect to arrange for a special plan.

Mod Slave #: If you are using the Modbus Slave protocol, this assigns a particular Modbus address to the channel. Please review the section on the Modbus Slave protocol for more information.

 # Hst: If your memory is limited but you still need to be able to keep a long history, you can set this column to a number larger than 1. This column tells DAQFactory how often to update the history of this channel. If this column contains 10 on a 1 hz channel, then the history will contain a bunch of points separated by ten seconds. The latest data point is always kept in the history (i.e. at subset [0]).

Brd?: This determines if the channel data is broadcast on DDE. DDE must be enabled through Document-Settings first. See the chapter on networking for more information. This determines whether the channel data is broadcast to remote copies of DAQFactory that are connected to this copy of DAQFactory using the slim data stream. This is discussed more in the chapter on networking.

 # Brd: In low bandwidth applications, or any application where you want to reduce your network load, you can set the interval in which data is broadcast to other copies of DAQFactory. A value of **1** for this column results in all data being transmitted. A value of **0** results in no data being transmitted even if connected via the full data stream. A value of **2** results in every other data point being transmitted, and so forth. No averaging occurs, and this has no affect on the data storage. This affects both the full and slim data streams.

Group: This determines which group the channel belongs to. Channel grouping only affects how the channel table and workspace are organized and have no effect on the rest of DAQFactory. Using channel groups is discussed in the section on Channel Groups.

Additional channel parameters exist that can only be edited in the Channel View.

Once you have entered all your channels, or have made any changes to existing channels, make sure you press the **Apply** button to start taking data. Press the **Discard** button to ignore any changes and revert back to the page view. If you forget to click **Apply** and try and switch to a different view, you will be prompted to apply you changes.

6.3 Channel Import and Export

To allow for editing of channel information outside DAQFactory, you can export and import the channel table information to a tab delimited file that can be loaded into Excel or other similar application. To export your current channels, go to the channel table and click on the Export button. You will be prompted for a file name and then the export will begin. Once complete you can edit this file with another application and then import it back into DAQFactory. To import, click on the import button and select your file. Here are a few things to keep in mind:

Exporting:

- If you have multiple channel groups, only the group currently displayed in the channel table will be exported. To export all your channels, make sure and select the "All" tab first.
- If you are going to create a channel list from scratch from an external application, we recommend creating at least one channel in DAQFactory and then exporting it to create a starter file. This will create the headers and insure that you get the right information in the right columns.
- Columns with checks like Avg? and Brd? will appear as 1 for checked and 0 for not checked in the exported file.

Importing:

- All currently displayed channels are erased before importing. If you have multiple channel groups, only those channels in the group currently displayed are erased.
- **DAQFactory does not verify the imported data**, so make sure everything is correct before hitting Apply. You can always Discard your changes and the imported information will be forgotten. Things to watch for are improper Device names and I/O Types which ARE case sensitive, duplicate channel names, importing over only one channel group when you meant to replace all channels, and general typos.
- If you want to import without replacing existing channels, create a new junk channel in its own group and select that group tab to import over the junk channel.

We strongly recommend practicing a little with importing and exporting on a blank document with a few channels before creating a giant channel list. This will ensure that you are creating the file correctly.

6.4 Channel History and Persistence

Most of your data will be stored in channels. Channels have a history which is the data acquired with a time associated with each data point. The amount of data available from a particular channel for graphing and calculations is determined by the channels History Length and its Persistence Length. This is independent of data logging and has no bearing on logging or export sets. In older versions of DAQFactory, a channel only maintained its history in memory. This was its History Length, or the number of points kept in memory for graphs and other calculations. As described earlier, you can retrieve data from a channel by entering its name in an expression, often using subsetting to specify which part of the history you wish to use. The data available was limited to the channels History Length which, while user definable, is somewhat limited by your system's memory. Now channels can also save historic data to disk in an optimized binary format for quick and easy access by graphs and other calculations. The amount of disk space used for a particular channel is its Persistence Length, which is only limited by your available disk space. The file is preallocated and optimized so that the Persistance Length is completely scalable without a performance hit. This means you could keep 10 million data points on hand for your graphs and calculations without having to worry about loading your data into DAQFactory from a logging file.

Using channel persistence is as easy as setting the Persistence Length for each channel. This can be any size up to the limit of your hard-drive. Each data point takes up 16 bytes, so a Persistence Length of 100,000 will take 1.6 meg of disk space. The files are created in a subdirectory called Persist under your DAQFactory installation directory. The file is preallocated, meaning a Persistence Length of 100,000 will create a 1.6 meg file immediately, even though you haven't accumulated 100,000 data points yet.

Accessing persistence data is done completely transparently. When you request some channel data, for example by doing `MyChannel[0,4000]`, DAQFactory will first check the channel's history in memory as determined by its History Length and retrieve it from there if it is all available. If not, it will automatically go to the disk and retrieve it from the disk. So, if your History Length is set to the default 3600, `MyChannel[0,4000]` will retrieve the data from the persistence file, but `MyChannel[0,3000]` will retrieve it from memory. The only exception is when you do not subset at all, i.e. `MyChannel`. This will retrieve the history as determined by the History Length. This is largely to keep you from accidentally retrieving a huge number of data points from the persist file inadvertently, which would occur because of DAQFactory's expression colorization as you typed your expressions.

This last paragraph is an important point when determining what to set History and Persistence length to. If you don't need to keep a lot of historical data for a particular channel and you don't care if the historical data is reset the next time you start DAQFactory (separate to what is logged of course), then just use History Length and set Persistence Length to 0. If, however, you do want to keep a lot of historical data, or you want your data to automatically be available if you restart DAQFactory, then set your Persistence Length to the desired amount. Even if you use channel persistence, you will want to set your History Length to a non-zero value as well, otherwise DAQFactory will be forced to go to the disk every time it needs data from the channel. At a minimum, your History Length should be 1, so that all [0] requests for the most recent value pull from memory, but remember it only takes 16 bytes per data point, so if you have a display of the mean of the last 100 points, you probably should set your History Length to at least 100. What exactly to set it to depends on the situation. Just remember that data retrieved from memory is much faster than data retrieved from the persistence file. At the same time, we

recommend keeping History Length below 50,000 to 100,000.

Some important points and tips on channel persistence:

• Channel persistence is NOT data logging. Data logging is completely separate within DAQFactory. Channel persistence should NOT be used for long term archiving of data. You should always use a logging or export set to log your data into a ASCII file or to an ODBC database so you can retrieve it from other programs. This can easily be done simultaneously to channel persistence. The persistence files are in an optimized binary format and are not readable from other programs. They are stored in a rolling buffer, meaning that once the Persistence Length is reached, the oldest data is overwritten. Channel persistence is simply a way to allow quick access to long term data from within DAQFactory.

• Channel persistence does not work with string, spectral or image data. It only works with normal scalar values.

• Changing the Persistence Length for a channel will clear any data in the persistence file.

• Clearing the history of a channel, for example with the ClearHistory() function will clear the persistence file as well (though it won't change its apparent file size).

• All subsetting functions work with persisted data transparently. This includes subsetting by time, which makes it really easy to query for data from a particular time frame. As stated above, the exception is when you specify a channel name without any subsetting. In this case, only the history determined by the History Length is retrieved. If no data has been acquired, this means it could be from the persistence file.

• The Persistence Length is limited to less than 100 million data points. This is because when subsetting, a number larger than 100 million tells DAQFactory that you are subsetting by time, not data point. Presumably, you could set the Persistence Length larger than 100 million, but in order to access any points past 100 million you'd have subset by time and not point number.

• If you find DAQFactory is bogging down as it runs, check your expressions. You may be inadvertently retrieving a large amount of data from the persistence file. Data retrieval from the persistence file is very fast, but remember that a page component will repetitively retrieve data with each page refresh, so also watch your page refresh rates.

• Because specifying just a channel name with no subsetting only returns a maximum of History Length points, doing **`NumRows(MyChannel)`** will not return the full number of data points available from the persistence file, but instead the number of points available up to the History Length. To retrieve the full number of data points in the persistence file, use the GetHistoryCount() function of your channel: **`Len = MyChannel.GetHistoryCount()`**

• If you want to analyze a very large subset of your data, say over a million data points (or perhaps less), you should avoid doing it directly in a page component. For example, if you want to display the maximum value read over the last year at 1 hz, you should not do: **`Max(MyChannel[0,31536000])`** in a Display Value Component as the component will force those 31 million data points to be loaded into memory every time it refreshes. Instead, you should use a sequence and calculate it once there and store the result in a variable that is displayed. Of course this will still require DAQFactory to load 31 million data points requiring 504meg of memory, so you should think about alternative ways to do this. For example, load and analyze the data in smaller chunks:

```
Private x
Private maxs
for (x = 0, x < 100, x++)
  try
    maxs[x] = max(MyChannel[x * 100000+1,(x+1)*100000])
  catch()
  endcatch
endfor
return (max(maxs))
```

In this example, we determine the max of chunks of 100,000 data points and put it in the maxs array, then find the max of that array. This keeps our maximum memory load to 1.6 meg instead of 160 meg required if we did it in one step. Notice that we use a try/catch block as we may subset past the end of the persistence file and don't want an error, just an empty array item. You could also use **`ignore("all")`** if you didn't need any other error handling. A better way would be to simply use GetHistoryCount() to avoid subsetting past the end.

There are other ways of doing this. For example, you could precalc the max on startup and then use the channel event to update the max if needed. With channel persistence, DAQFactory makes very large data sets easy to deal with, but you still have to think about what you are doing and what DAQFactory is doing for you. Because systems are different, we cannot make it all completely automatic.

• A common use for channel persistence is to allow the user to scroll the graph back a long distance in time. One

way to do this is to create two variables for the graph start and stop time and use one of the many components available to change these variables. When using this method, the variables should be used both in the bottom axis scaling and in subsetting the channel as described above. So, if your variables were STime and ETime, you'd do MyChannel[STime,ETime] vs Time. If you don't subset like this you will waste a lot of processor power loading data from the disk that is never used, so try and only request the data that is needed. This is actually good practice even if you are retrieving data from memory.

• Another use of channel persistence is to allow you to restart DAQFactory without losing the data used for graphs and calculations. Since the persistence files remain when DAQFactory stops, they are available when it restarts so even though no data may have been acquired, your graphs can automatically retrieve the older data from the persist files.

• The channel table in the channel view will only display up to the History Length data points. Again, this is so you don't inadvertently retrieve a large number of data points.

6.5 Channel Groups

Channels can be placed into groups. Channel groups make it easier to use the channel table and workspace when you have many channels, but have no affect on addressing the channels in expressions. By default, all channels are in the main group. The channel table displays tabs above the table for selecting the group. At the very right is a button labeled **New Group** that can be used to create a new group. Once you have multiple groups, a new tab labeled **All** will also appear. This tab lists all channels irrespective of group. There is also a column in the channel table labeled **Group**. You can use this column to quickly change the group of a channel and even create new groups by entering in new group names here.

6.6 Device Configurator

The device configurator is used to set various settings of your devices using dialog windows. Some devices, such as serial based devices, require the device configurator to properly set up the device. Others simply use the configurator as a more user friendly way to set default parameters. To open the device configurator, either click **Quick - Device Configurator** from the main menu, or click on the toolbar icon. Once selected, you will be prompted to select which device configurator you would like to use. The options will depend on which devices are installed, and which installed devices have configurators. Once selected the given configurator will be displayed.

Once closed, settings in the configurator are saved with the document. These settings are actually just ASCII strings with the necessary parameters. The format of these strings varies slightly depending on the device, but in general are readable. Because of this, there is also a **View Raw** option when selecting which device configurator to use. If you select a configurator, then hit **View Raw** instead of **Select**, another generic window will appear with a text editor and the current configuration string for the given device. You can view and edit this at will. Click **OK** to save your changes, or **Cancel** to ignore them.

6.7 The Channel View

The channel view provides another way to enter or edit your channel information. The channel view also gives you access to additional parameters not available in the channel table.

To get to the channel view for a channel, simply click on the channel's name displayed in the workspace. You may have to expand the **CHANNELS:** list by clicking on the **+** sign to its left to see your channels listed.

The channel view only displays one channel at a time so can be slow to add a lot of channels. The channel view has some nice extras though. In addition to all the information that can be entered for a channel in the table:

• You can enter more detailed notes about the channel.

• You can quickly view your data in a table. This shows you the exact data coming in and is also helpful for determining if you have entered the correct channel information.

• You can quickly view your data in a graph. This gives you a quick look at the quality of your data. This is not the primary graphing functionality of DAQFactory though, and really only designed for a quick glimpse.

• You can create an event. The event is sequence type code that executes every time the channel gets a new data point. The event is called after the data in the channel is updated, so you can retrieve the latest data by simply referencing the channel. For high speed streaming data, the event is only triggered for each block of data. For performance reasons it is best to keep these events short and quick. If you need to do something that takes some time, consider starting a sequence from within the event using the beginseq() function.

To create new channels without using the channel table, you can right click on any of the channels listed in the workspace and select **Add Channel**. You can also delete channels by right clicking on the channel name in the workspace and selecting **Delete Channel**. If individual channel names are not shown in the workspace, you must click on the plus sign next to the **CHANNELS:** label. If there is no plus sign then there are no channels. If this is a remote connection and you think there should be channels listed, you can try resetting the connection to force another download of channel information. To do this, click on the connection name to display the connection view and press the **Reset Connection** button.

6.8 Fine Tuning Data Acquisition

Here are a few miscellaneous things that will help in your data acquisition.

Time interval limitation

The smallest timing value that you can use largely depends on your processor speed, and whether you have any other programs running besides DAQFactory. Due to limitations with Windows, a timing interval of less than 10 ms is not suggested, no matter how fast your computer. If you need faster acquisition, we suggest you use hardware paced acquisition, usually done using device functions or channel commands. The availability of hardware paced acquisition depends on your hardware. The other alternative is to use a computer with two processors, or a dual-core processor which are getting rather common. With this setup you can have fast acquisition run on one processor, while the other processor handles the rest of Window's tasks. This is a bit more tricky. If you elect to go this route, feel free to contact us for assistance.

Timing in sequences is a different issue and is discussed in the chapter entitled Sequences.

Determining timing latency

To determine how well your data acquisition is performing in terms of reading data at expected intervals, you can make a graph showing the difference between consecutive data points by doing the following:

1. Create a blank graph.

2. For the trace's Y expression enter **`GetTime(channelname) - GetTime(channelname[1,3600])`**, where channelname is the name of the channel you would like to check.

3. Leave the trace's X expression as **`Time`**.

The graph will display the time of each point minus the time of the previous point.

Reading data at non-uniform intervals

You can setup a sequence to read data at any time given. For example, the following sequence reads data once, waits 9 seconds, reads it again, waits another second, then loops around and reads again.

```
while (1)
  read(MyChannelName)
  delay(9)
  read(MyChannelName)
  delay(1)
endwhile
```

The data from the sequence gets stored just like normal data. Note that we use the **`delay()`** function instead of

wait(). The **wait()** function is non-propogating, so any delays in actually reading **MyChannelName** will not cause the data acquisition times to vary. However, this only works with devices that have short and constant read times otherwise the reads may get back-logged and probably hang DAQFactory. You should always use **delay()** instead of **wait()** if doing any sort of serial or Ethernet communications. Either way, you will want to set the timing for your channel to 0.

Dealing with slow hardware

To help you deal with slow hardware, we have provided an offset parameter for your channels that allows you to predictably stagger your acquisition. Offset is used in conjunction with the timing information for a channel to determine when exactly a channel is read. It is best illustrated with some examples:

Timing = 1, Offset = 0	Data is read every second on the second.
Timing = 10, Offset = 0	Data is read every ten seconds, on the second.
Timing = 1, Offset = 0.5	Data is read every second on the half second.
Timing = 10, Offset = 5	Data is read every ten seconds starting five seconds after a Timing 10, Offset 0 would read.

All channels with the same timing and offset information are grouped together internally, and each time the timing interval comes up, DAQFactory will read each of the channels in the group in an unpredictable order (although it will read together all channels from a single device). If you have a channel connected to a piece of hardware that is slow to read, the reading of all channels afterwards will be delayed. Offset provides the mechanism to prevent this. If, for example, you have a group of channels you'd like to read at 1 second interval and one of those channels takes half a second to read, you could give all the fast channels a timing of 1 and an offset of 0, then give the slow channel a timing of 1 and an offset of 0.1. This way all your fast channels will read right on the second. The slow channel would begin reading 100ms later, which gives the fast channels plenty of time to finish reading.

Technical Detail: Because each timing / offset setting runs in a different thread of execution of the computer, you can actually simply put a slow channel in any offset, even say 0.01, and provided they aren't on the same device driver (unless the driver is designed for multithreaded access), the reads will occur concurrently. If they are on the same device, you can set the offset at around 40 ms and the slow channel will read when the other channels are done, or 40 ms after they start, whichever is greater. The 40 ms comes from the maximum observed latency in timing plus some additional cushioning.

Reducing your memory usage

There are two ways to reduce your memory load.

1. Drop your screen's resolution or color depth. A screen at 1600 x 1200 x 32 bit color requires almost 50 meg of memory within DAQFactory because of DAQFactory's triple background painting algorithm. By contrast, a 1024 x 768 x 256 color screen requires only 3 meg of memory.

2. Shorten the history amounts of your channels, and consider using channel Persist. Alternatively, by using the **History Interval** (or **# Hst** column) parameter of the Channel, you can increase your history length in time in memory, while reducing your total memory load. This parameter determines the interval at which a data point is stored into the history. The default is 1, which corresponds to every point. A value of 10, for example, would mean every 10th data point is stored in the history. If your Channel is being read at 1 hz, you could store a full day's worth of data with a history length of 8640 with a History Interval of 10, instead of 86400 required with an History Interval of 1. Even though every 10th point is saved, the most recent data point is always the first data point in the history array (i.e. channelname[0] always contains the most recent data point.).

Reducing network load

To reduce your network load, connect using the slim data stream, and/or adjust the broadcast interval of your channels. The broadcast interval controls how often a channel's data is sent over the network (broadcasted). A value of 1 means all data is broadcast, a value of 0 means no data is broadcast, a value of 2 means every other data

point is broadcast, etc. You can still see the entire channel list and control outputs, but not all channels' data is sent over the network (this does not affect storage). You can even set the broadcast interval to 0 which will stop broadcasting that particular channel. The slim data stream only passes data for the channels marked **Broadcast?** With the slim data stream only channels that are broadcast are visible, and you can not set any outputs or start / stop any sequences (much less see them). This is designed more for sharing data with a colleague that does not need all the details of your experiment and certainly does not need to fiddle with the experiment.

Network security

When using DAQFactory networking (meaning adding a new Connection), data is encrypted to the level allowed by your location and your Windows installation. The encryption key is based off of the passwords you provide for networking, so it is important to put at least a minimal password in. You can protect yourself further using a hardware or software firewall. DAQFactory uses ports 2345 and 2346 to communicate over the full data stream, and ports 2347 and 2348 to communicate over the slim data stream. If you block traffic on ports 2345 and 2346, users on the other side of the firewall will not be able to connect using the full data stream, but can still connect using the slim data stream. Personal software firewalls are cheap (or even free), readily available, and pretty easy to setup. Just make sure you can block individual ports with your firewall.

Network passwords are discussed in the section on document settings.

6.9 Channel Functions and Variables

Functions:

Channels offer four different functions. These function are accessed by following the channel name with a period, then the function:

```
MyChannel.ClearHistory()
```

These functions can be used anywhere sequence code is used: sequences, events, and component actions.

Here are the four functions:

AddValue(Value):

Calling this function manually adds a value to the history of the channel. Value can be a scalar value or an array of values. If Value has time associated with it, the time of each row is inserted as well. If Value does not have time associated with it, the current time is inserted before the data is added to the channel. For array data with multiple rows, all rows are assigned the same time. Use the InsertTime() function on value if you want to insert a different time.

Application: The AddValue() function is typically used in sequences that acquire data using unusual methods. For example, many of the serial devices have device functions which will read a batch of points across multiple channels with a single call. Once the data is acquired, you can use the AddValue() function to add the data to the appropriate channels. For example:

```
// read three consecutive tags from a Modbus device:
Private.In = Device.MyModbus.ReadHoldingFloat(1,600,3)
// ReadHoldingFloat returns a 2 dimensional array, but with 1 row, 3 columns.

// now add the three values to three different channels:
CoreTemperature.AddValue(Private.In[0][0])
WaterTemperature.AddValue(Private.In[0][1])
WaterPressure.AddValue(Private.In[0][2])
```

AddValue() does not convert the value from the channel's conversion. It will, however, cause the data point to be logged to any logging sets, broadcast to any remote systems, and alarms to be calculated.

ClearHistory([# of points to leave]):

Calling this function will clear the channels history of all values except for the number of points specified in the function. Leaving this parameter blank clears all the points in the history, which is the same as `ClearHistory(0)`. After clearing, the history will rebuild itself as new data arrives.

`MyChannel.ClearHistory(5)` clears all but the last five data points.

MyChannel.ClearHistory() clears the entire history.

Application: The most common application of the ClearHistory() function is when creating X-Y graphs. Since graphs utilize the entire history unless specified otherwise, the X-Y graph may display undesired points. For example, if you are running batch experiments and need a new X-Y graph with each batch, you could clear the history of the channels used in your graph at the beginning of each batch.

GetHistoryLength():

Returns the number of historical data points in the channel, essentially the number of rows. This function differs from NumRows() in two ways: NumRows() works on channels, variables and any other array, but for channels will return the number of historical data points, but only up to the channel's history length. GetHistoryLength() only works with channels, but will return the length including any persisted data as well and is much more efficient then using **NumRows(MyChannel[0,1e7])** which will achieve the same thing but require the entire MyChannel persisted history to be loaded into memory. Bottom line: use GetHistoryLength() to retrieve the number of data points in a channel, and NumRows() for all other arrays.

Time:

This is not quite a function, but allows you to request the Time associated with the channel instead of the data. This is the same as GetTime(), but offers more when used with variables (as explained in the section on variables under Sequences & Scripting):

MyChannel.Time returns the time of all data points in the history of MyChannel (the same as **GetTime(MyChannel)**)

MyChannel.Time[0] returns the time of the most recent data point in MyChannel (the same as **GetTime(MyChannel[0])**)

Variables:

All the properties of a channel except the channel name are editable from script with the following variables. To edit the channel name you must delete the channel and recreate it. Changes to some of the properties, namely device, device number, i/o type, channel, strSpecifier, timing and offset, will not take effect until you call the Channel.Restart() function. This function, described in the next section, stops the acquisition and restarts it with the new parameters. The values may appear to be changed, but the changes do not affect acquisition until the Restart() call.

Like the functions above, all these variables are preceded with the channel name and a period.

strDevice: this is the name of the device as a string. If you change this value, make sure and change the IOType as well. Although two different devices may have the same IOType string name like "A to D", internally they may be coded differently.

DeviceNumber

strIOType: the I/O type name as a string.

Channel

Timing

Offset

HistoryLength

PersistAmount: remember that changing this value will clear the persistence file of any existing data.

ModbusRegister

HistoryInterval

strConversion: this should be set to "None" when no conversion is desired.

Broadcast: 0 or 1

BroadcastInterval

Average: this is 0 if averaging is off, or the number of data points to average.

strSpecifer: this is the quick note / special / opc column often used by the device.

strQuickNote2: another space to store some info about a channel. This is typically this is one liner information.

strNote: yet another space for random info about a channel. This is typically more in depth info about the channel.

strGroup: the group the channel belongs. Note that channel groups are created on the fly, so you simply assign a channel a new group name to create a new group. Changes to strGroup may not update in the workspace as dynamic, scripted changes to strGroup are typically used in fully scripted apps in conjunction with the Channel.ListAll () function described in the next section.

strEvent: the sequence script that gets executed when the channel receives a new data point.

6.10 Channel List Functions

In addition to channel specific functions there are several function that affect the list of channels and allow dynamic creation and removal of channels. All these functions start with "Channel.", i.e. `strList = Channel.ListAll()`

Before we get into the functions a few quick important points:

1) These function ONLY work on the Local connection. Attempting to use them on any other connection will have unpredictable results.

2) These functions will not update the workspace with any added or removed channels. The only way to get the workspace to update is to go to the channel table, make a minor modification and apply your changes. These functions are designed for dynamic applications that typically run in the runtime environment that doesn't have a workspace. To make them quicker, especially when dynamically adding many channels, the workspace is not refreshed.

3) If you add or remove any channel with timing information you must call Channel.Restart() to get DAQFactory to start or stop acquiring data for that channel.

Add(channel name, [device = "Test"], [device number = 0], [i/o type = "A to D"], [channel = 0], [timing = 0], [offset = 0], [specifier = ""], [group = "Main"]): adds a new channel to the local connection with the given name. If the channel already exists another is not created, but instead the current channel is updated with any of the information provided. Only the channel name is a required parameter. In cases where you are dynamically creating channels to use solely with AddValue(), this is all you should need to set, unless you want to group your channels (see ListAll()). For the device and I/O type parameters, you should pass a string with the desired device name or I/O type. Remember if timing equals something other than 0, you must call Restart() to start the timing loop.

Once you have a channel created, you can change parameters dynamically using the execute command:

```
Execute(strMyChannel + ".HistoryLength = NewHistoryLength")
```

Where both strMyChannel and NewHistoryLength are your variables. Of course, if you know the channel name at design time, you should just do:

```
MyChannel.HistoryLength = NewHistoryLength
```

AddValue(device name, device number, io type name, channel, value) This works just like the channel version of AddValue() described in the last section, except you don't have to know the channel name for it to work. Instead, you should provide the name of the device type, the device number, the name of the I/O type and the channel to specify which channel you wish to add the value to. For example:

```
Channel.AddValue("Test",0,"A to D",0,5)
```

This function will return an error if the specified channel does not exist. This function is most commonly used within user and communication devices when you wish to create a generic driver.

Delete(channel name): removes the channel with the given name if it exists. Remember, if the removed channel has a timing other than 0, you must call Restart() to remove it from the timing loops. Failure to do so will create excess overhead.

ClearAll(): removes all the channels in the Local connection. Remember if any of the channels have a timing other than 0 you must call Restart() to stop all the timing loops.

Read(string Array): takes an array of strings containing channel names and reads those channels. For example:

```
Channel.Read({"myChan", "myOtherChan"})
```

This is similar to doing read(mychan) and read(myOtherChan), except that by doing it in a single call, drivers are able to optimize reads, for example Modbus reads can be optimized into a single call.

ReadGroup(groupName): takes a string containing the name of one your channel groups and reads all the channels in that group. For example:

```
Channel.ReadGroup("myGroup")
```

This works just like Channel.Read() except uses the channel's group setting instead of taking a list of channels. Make sure you do not have any output channels in the Group!

Restart(): this stops all the timing loops and restarts them with any changes from the above functions or variable changes with channels described in the previous section. Restart() is not terribly fast, especially when dealing with slow devices as it has to wait for the current loop to complete. For this reason, the loops are not restarted automatically when a parameter changes or a new channel is added.

ListAll([group]): returns an array of strings containing the names of all the channels. If group is provided, only those channels in the given group are returned: `Channel.ListAll("Main")`. Used in conjunction with Execute () or Evaluate() you could easily apply things across groups of channels.

6.11 Conversions and the Conversion Table

Conversions provide a way to quickly convert your data from the sometimes arbitrary units that your hardware device generates into more useful units, such as psi, torr, K, etc. Conversions can be used for complex calculations, but we suggest keeping it simple and save the complex calculations for DAQFactory. Conversions work for both input and output channels.

Entering Conversions:

To create conversions you will use the conversion table. To access the conversion table, click on the **CONVERSIONS:** label under the desired connection. The table only has two columns, the conversion name and the expression for the formula of the conversion. To add a new conversion, click on the **Add** button. Enter in a name for the conversion. Next type in the formula. The formula cell is an expression type cell. This type of cell works the same as any expression box used in DAQFactory. However, it should be noted that for all Connections except local, only channels for the given connection will be listed.

You can print the conversion table by selecting **File - Print** from the main menu.

Using Conversions:

A single Conversion can be used on multiple channels. This is especially useful if, for example, you have ten pressure transducers that are all 0 to 500 psi corresponding to a voltage of 0 to 10V. If your A to D channels were 16 bit and output 0 to 65535 corresponding to 0 to 10V, you would create a conversion with this formula to convert from the raw A to D number to psi:

```
Value / 65535 * 500
```

The word `Value` is used wherever you want to substitute the raw channel value in the formula. You can use channels by name as well, as long as they exist on the same connection. You can also use almost all the available functions. The only ones that won't work are the ones that process an array (unless the conversion is for a spectrum) and the ones that modify time.

Conversions can also be used to convert output channels. As an example, lets make the last example an output. If you have a pressure controller that takes 0 to 10V for a pressure setting of 0 to 500 psi, and your 16 bit D to A takes a value from 0 to 65535 corresponding to 0 to 10V, you would enter the following formula for your conversion so that you could enter psi values for set points:

```
Value / 500 * 65535
```

Notice how this is the reverse of the previous formula. For outputs, we want to take useful units and convert them into the arbitrary units of your hardware. For inputs we do the reverse.

Applying Conversions:

To apply a conversion to a channel, open the channel in the channel view, or open the channel table and find the desired channel and select the conversion from the drop down list.

Note: Your data is sacred, so logging sets give you the option of storing your data before converting or afterwards. See the chapter entitled Logging Sets for more information.

6.12 V Channels

Virtual or V channels provide a location for storing values that do not have any I/O associated with them. This can be results from analysis calculations, captured values from graphs, calculated expressions, or used as variables with a history. They are called channels because they can be used pretty much anywhere a channel is used within the DAQFactory program and typically have a history. Virtual channels are associated with a static connection name **V**. This Connection is referenced just like other connections, and can be made the default connection if desired. This means that typically you will have to put `V.` in front of any the name of your Virtual Channels when accessing them from the rest of DAQFactory, such as a screen component.

To manually create a V channel, right click on the **V:** in the Workspace, or the **CHANNELS:** under the **V:** and select **Add VChannel**. Enter in the desired name of the new channel remembering the limitations on channel names. The name must start with a letter and can only contain letters, numbers, or the underscore. Click **OK** to create the V channel. To open the V channel view for the channel and edit its parameters, expand the **CHANNELS:** label by clicking on the **+** sign to the left of it, then click on the name of your new V channel. There is no V channel table like there is for channels.

The V channel view has some similarities to the normal channel view. V channels have quick notes and notes just like regular channels. The views both have a graph displaying the data, and an available table for viewing data.

There are three types of V channels:

Variable V channels:

A new V channel starts out as a variable V channel. Using the same methods you would use to set a real output channel, you can set the V channel to any value. The channel maintains a history of values and thus acts very much like a regular channel.

Application: Use this type of V channel like you would a variable when you want to track historical values. If you do not need a history and just need to track a singular value, use a variable, as variables are faster to access.

Calculated V channels:

The next type is a calculated V channel. You can assign an expression to the channel, and the values in the V channel will be calculated from this expression every time the V channel is referenced. A variable V channel becomes a calculated V channel as soon as you enter an expression.

Application: You can use calculated V channels to store complex calculations that you may use in multiple locations. Put the expression for the calculation in a V channel and then simply reference the V channel elsewhere in DAQFactory. This saves you from having to retype the expression multiple times.

Application: It is perfectly legal to create constant expressions. For example, this expression will create a simple array: `{3,4,7,1,3}`. Use the InsertTime function to add time to a constant expression: `InsertTime ({3,4,7,1,3},5h32m12,1)`. One example of this application is creating a lookup table of values.

Static V channels:

Static V channels are simply static arrays of data. The data can have time associated, or simply be data values. Captured values from graphs, and results from analysis calculations are stored in this type of V channel.

Note: the data in both variable and static V channels is saved with your document along with the rest of the information about the V channel. The expression in a calculated V channel is of course saved as well.

6.13 Disabling a Device with Bypass

There are times when you may be testing your software and not have all your hardware, or a piece of hardware is simply out for repair and you'd like to run your program without the constant alerts telling you DAQFactory couldn't communicate with the device. There are also times when you'd like to run a simulation of inputs instead of using the real inputs. For these cases, there is the Bypass feature. The bypass feature allows you, on a device by device basis, to skip calling the device or communication drivers and instead call script.

The bypass feature consists of two member functions of device types and a system event. To bypass a device use the SetBypass function:

device.myDevice.setBypass([Device Number])

where myDevice is the name of the device as you see in the Device Type column of the Channel table. Device Number is an optional value that allows you to disable just one particular piece of hardware for a device type. For example, let's say you have created a Modbus device connection called "MyMod" and it is an RS485 multidrop with 3 PLC's with Modbus ID's 1, 2 and 3. ID #2 died and you want to bypass it. So you would put:

```
device.myMod.setBypass(2)
```

If you don't specify the device number, then all device numbers for that device type are bypassed.

You can call the function multiple times to disable multiple devices for a particular device type or across different device types. So, continuing the last example, if ID #3 also died and you wanted to only read ID 1, you would do:

```
device.myMod.setBypass(2)
device.myMod.setBypass(3)
```

To cancel bypassing for a particular device type, use ClearBypass():

```
device.myMod.clearBypass()
```

This would clear all bypass instructions for the device type. This function does not take any parameters and affects all device numbers on the specified device type.

Note: the Bypass feature only affects channels. It will not affect device functions that read / write to the device. So, device.myMod.readHoldingU16(2, 0, 1) would not be bypassed in the above example.

If all you needed to do was disable a device, the above functions are all you need. But for more control there is the OnBypass system event. If you create a sequence called OnBypass, the sequence will be called every time DAQFactory needs to read or write a channel that would otherwise be bypassed. The function is passed the following parameters:

DeviceName, DeviceNumber, ChannelName, SetValue

SetValue is only passed if the channel is being written to, and it receives the unconverted output value.

With this information you can then manually insert values into your channels and thus create a simulation. You also have finer control over what gets bypassed. If you return(0) in the OnBypass sequence, then the channel will not be bypassed. So, continuing our example, if we had a channel called "MyInput" one device number 2 of our myMod device that we wanted to read even if the rest of the device was bypassed, we could put this in OnBypass:

```
if (ChannelName == "MyInput")
   return(0)
endif
```

Returning any non-zero value, or simply not calling return() in OnBypass results in the channel being bypassed.

Creating a simulation:

As mentioned, you can use the bypass feature to create a simulation mode for your application for when there is no hardware available. This is great for testing, or for demonstrations. How this is implemented depends quite a bit on your application and there is a lot of ways to do it, but here are a few tips:

1) use startup flags, or a persistent setting to determine if you are in Sim mode or not. Don't hard code it, otherwise you'll forget to set it back to non-sim mode when you ship

2) At the least, you probably want this script in OnBypass:

```
if (!isEmpty(setValue))
   execute(channelName + ".addValue(setValue)")
endif
return(1)
```

This will cause any output commands to put the output value into the channel for you.

3) You can also put your simulation code for inputs in OnBypass, but its usually better to create a separate sequence that runs when you are in your sim mode that loops at some interval and stuffs new values into your input channels based on the values of output channels. Use addValue() to stuff the values into the channels.

6.14 Questions, Answers and Examples

6.14.1 Oversampling a channel (averaging)

Question:

What's the quickest way to tell DAQFactory to oversample and average a channel before on see it on my Page_0 display? I saw an Avg column in the channel setup, but the docs made it sound like this was only for writing to disk.

Answer:

Depends. If you always want to use the average, then use the check box in the channel and enter the number of points to average. If you want to have the normal data and then display an average on the screen as well enter:

```
Mean(MyChannel[0,9])
```

in the expression for the component to display the average of the last 10 points. You can also graph averages using either the boxcar or smooth functions depending on what you want.

6.14.2 Creating Conversions for input channels

This sample demonstrates how to use conversions on an input channel and shows that a single conversion can be used on multiple input channels.

Step 1: Go to **CONVERSIONS:** in the Workspace and Add a new conversion.

Conversion Name: MyConversionName
Formula: Whatever conversion you desire to make
Example: Value / 65535 * 500

In the example, Value / 65535 * 500 would take the value of the input and divide it by 65535 and then multiply that by 500, storing the result in the channel. By using the keyword Value instead of the name of the channel, we can apply this conversion to multiple channels.

Step 2: Go to **CHANNELS:** in the Workspace and Add a two new channels.

Channel Name: MychannelName
Device Type: Test
I/O Type: A to D
Chn#: 0
Timing: 0

Channel Name: MyOtherName
Device Type: Test
I/O Type: A to D
Chn#: 10
Timing: 0

Leave all else to defaults. This will generate two sin waves with different periods.

Step 3: Click on **CHANNELS:** in the Workspace and then on **MyChannelName** under the CHANNELS: directory.

Step 4: Click on the **Detail** tab at the top of the page and use the drop down for **Conversion** to select `MyConversionName`

Step 5: Click on the **Table** tab and verify that the data conversion is working. It will be a small number since the original Test A to D value varied from –1 to 1.

Step 6: Repeat steps 3 through 5 with the **MyOtherName** channel. Once the conversion is applied, both channels will show small numbers, but even though they both have the same conversion, they will show different data.

6.14.3 AddValue() and Conversions

Question:

When I use the AddValue() function, it does not use the conversion that I have applied to the channel I am adding the value to.

Answer:

This is normal. Calling this function manually adds a value directly to the history of the channel. Conversions are done on incoming data before they are inserted into the history. Your solution would be to apply the conversion manually to the value before the AddValue() function call.

AddValue() will, however, cause the data point to be logged to any logging sets, broadcast to any remote systems, and alarms to be calculated

6.14.4 Applying a deadband to a channel

Question:

How can I add a sequence or condition that will consider a very small electrical current as '0' ? Is it possible to incorporate this sequence within the channel of that current sensor itself? If not possible, what is the best simple alternative?

Answer:

The easiest way to do this is with a conversion that takes advantage of simple boolean math. Lets assume the threshold below which the channel should be set to 0 is 1.5. Create a conversion with the following expression and apply it to your channels:

```
Value * (Value > 1.5)
```

The trick here is that the boolean part of this expression, Value > 1.5, evaluates to either 1, if it is true, or 0 if it is false. So, if the value is greater than 1.5, we multiple our channel value times 1 which simply results in the original value. If the value is less than or equal to 1.5, then the boolean evaluates to 0, which when multiplied by our channel results in 0.

7 Pages and Components

7 Pages and Components

7.1 Page and Component Overview

Once you have set up your channels, pages are where you will do most of your work. A page is a canvas for creating your own interface for controlling your experiment or process and displaying and processing your data. You can create simple pages that display your data, or you can create complex diagrams or pictures that allow you to control your process by simply clicking on different images. You can have an infinite number of pages, and can quickly jump between pages using the keyboard, or by clicking on components that you place on pages. You can even overlay pages to keep important information visible on multiple pages without having to duplicate the information or popup a page to display it in a separate window.

The items you place on your pages to display your data and control your process are called components. There are many different types of components. With the exception of graphs which are discussed in the next section, a description of each component and how to manipulate them is discussed here.

7.2 Adding Components to Pages

To switch to the page view, you can either click on a page name as listed under **PAGES:** in the workspace, or select **Quick - Goto Page View** in the main menu. If you are just starting, all of your pages will look the same: they will all be blank white. To add components to your pages, right-click where you would like to place a component and select the desired component type from the popup menu that appears, or drag the desired component from the Toolbox. There will be some times when you will see a big red X through a component. This typically occurs when the component does not have any valid data to display, which is often the case when you first add the component because you haven't configured it yet. It can also occur if you load a document created in a full version of DAQFactory in DAQFactory Express and the document has components that aren't supported in Express.

Whenever you add a new component to a page, the component will be listed in the Workspace under the page it was added. You can then click on the component in the Workspace and it will be selected. This works even for components inside of groups, allowing you to edit components inside groups without ungrouping the components. Components are given a default name corresponding to their type (prefixed by an underscore). As you will see, you can give your components other names and the new name will be reflected in the Workspace.

Once you have placed a component on your page, you will want to change its properties so it does something useful. By default, when you first add a new component, it is selected for you. If not, you must hold down the **Ctrl** key while clicking on a component to select it. A selected component has a shaded rectangle around it. Once you have selected the desired component, you can either press the **Enter** key, right-click on the component and select **Properties...** from the popup menu, or select **Edit - Properties...** from the main menu. Whichever one you choose will cause the properties window to appear. The properties windows often has several different sheets containing the various parameters of the selected component. Some sheets are the same between different types of components. You can switch from sheet to sheet by selecting the desired tab at the top of the box.

Alternatively, you can use the Properties control bar, which is typically docked along the right side of the screen. If not, go to View - Properties from the main menu. This is probably not the preferred method, but it offers some advantages. First, if you select multiple components, any shared properties can be edited at once and changed on all the selected components. Second, user component properties, which are discussed a bit later on in the chapter, are only accessible from this window. Third, you can easily edit the component's name from this box, which is usually not accessible from the regular properties window. The disadvantage of using the Properties control bar over the regular properties window, is that not all properties are available, and for many components, especially the graph components, the primary properties are completely inaccessible.

7.3 Arranging Components on the Page

Once you have created new components, DAQFactory offers quite a few tools for moving, resizing, and arranging the components on your pages. You can even make copies of your components to display your data in more than one place or to use a component's properties as a template for new components. Most all the mouse actions for arranging components require you to hold down the **Ctrl** key while clicking, or dragging. This is because a single

click on a component in Operate mode without the **Ctrl** key is used to trigger an action such as changing an output value. You can switch to Edit mode by clicking on "OPERATE" in the status bar in the bottom right corner and then not have to hold down Ctrl, but we find this much slower.

For this discussion we will assume you are in Operate mode

To select a single component, ctrl-click on the desired component. To select multiple components, hold down the **Shift** key and ctrl-click on all the desired components. You can also drag a box for selecting multiple components by ctrl-clicking in a blank area of the screen, and dragging, touching all the desired components with the newly created box.

Once you have a component or components selected, you can move them by simply ctrl-clicking and dragging inside the selection rectangle. The selection rectangle is the hatched rectangle that surrounds the selected component(s). Many components can be resized as well. When you select one of these components, the selection rectangle will have solid black handles at eight spots around the rectangle. You can ctrl-click and drag any of these handles to resize the component. To constrain the size of the component to a fixed width/height ratio, also hold down the shift key while you drag. You can only resize one component at a time. If you have more than one component selected, the handles will not appear. That said, you can resize multiple component using the keyboard shortcuts described below.

If you have overlapping components, each subsequent ctrl-click on an overlapping area will select the next component in that area, making it easier to select components under other components. When you are trying to move a component that is entirely within the area of another component, you may have to ctrl-click and drag the hatched area of the component instead of the center of the component, since clicking the center will select the other component.

You can also use the keyboard to move, or even just nudge components around on the page. Just press the arrow keys while components are selected (no **Ctrl** key necessary in Operate mode) and the selected components will move one pixel in the desired direction. You can hold down the **Shift** key while pressing the arrow keys to move ten pixels at a time. For resizable components, you can use the arrow keys to resize as well, just hold down the **Ctrl** key while pressing the arrow keys to move the bottom right corner one pixel in the desired direction. You can hold down both the **Shift** and **Ctrl** keys to move the bottom right corner ten pixels at a time.

When you have multiple components selected, you can use the tools in the **Layout** menu to quickly arrange your components. The **Layout** menu is available from the main menu. The tools in the **Layout** menu are also available by right clicking inside the selection rectangle.

Align: Aligns the designated edge of all the components to the component that was selected first among the block. Does not change the size of the components.

Space Evenly: Arranges the components evenly along the designated axis within, approximately, the width or height of the selection rectangle. Does not change the location of the components in the other axis.

Make Same Size: Makes the selected components the same size in the given axes based on the size of the first component selected.

Order: Changes the Z order of the component. This affects which components get displayed in front of other components. Note that since graphs and symbols are painted in the background, they are always displayed behind other components. So while moving a graph forward and back will cause it to be displayed in front of or behind other graphs and symbols, the graph will always be behind any other components. Think of it as two separate windows that you are looking through, one with all the graphs and symbols, which is always behind the other with all the other components.

Symbol: There are four options here that allow you to quickly rotate or flip a symbol component. This is especially fast when using the keyboard shortcuts displayed next to the menu item. Rotation of bitmap images is not supported.

DAQFactory supports the standard Windows clipboard for cut, copy and paste functions on components. You can use this to duplicate components, or move components on the same page, to another page, or even to another control document. For quick duplication, the **Edit** menu and the popup that appears when you right click inside the selection rectangle has a **Duplicate Component** menu item, which makes a copy of the component and pastes it onto the same Page, offset down the screen by 16 pixels. The **Edit** and popup menu also have a **Change Component's Page** option for moving a component to a different page (multiple pages are discussed in the next section). They also have a **Delete Component** menu item for permanently deleting the component.

7.4 Edit / Operate mode

DAQFactory has the rather unique feature of allowing complete edits while running. However, when editing pages, there are two modes available, Edit and Operate. In Edit mode, you can manipulate (move, select, and resize) components with simple clicks. In Operate mode, simple clicks will instead operate the components, for example, clicking one of your buttons, or turning a knob. Switching modes doesn't affect anything else in the system. Channels will still acquire data, sequences, logging sets and PID loops will continue to run, etc. The mode only affects how screen components are manipulated.

Different program modes are typically not recommended when designing new software, so you can also optionally use DAQFactory while remaining solely in Operate mode by simply holding down the Ctrl key when you want to move, select or resize components. We personally find this mode much faster to use than switching modes so strongly suggest remaining in Operate mode.

Note: DAQFactory will always start in Operate mode.

Note: Edit mode does not stop the system from running, but simply changes the way the Ctrl key works. If you want to stop your system from running, use Safe Mode.

7.5 Grouping Components

Like many object drawing applications, you can group several components into a single component. By doing this, you can move all the components in the group as one. To group components, select multiple components and go to Edit - Group Components. The resulting group will be selected. A selected group of components displays a slightly different box around the group to show you that you have a group selected. To ungroup a group, select just the group and go to Edit - Ungroup Components. To move individual components within a group, select the component from the workspace under the group, then you can drag the component or use the keyboard to nudge it.

A group of components is actually a component itself, a container really. This means you can group groups of components as well. As we will discuss more later, this also means that the group itself has events, local variables and all the features of regular components. This allows you to assemble several components into a group, add some code for advanced features, and thus create essentially a new component. This is discussed a bit later in this chapter. Normally, if you group several components, and then subsequently ungroup the components, the group "container" itself is deleted, and you are basically left exactly how you started. If, however, you add event code or user functions to your group container and then ungroup the components, the group container will not be deleted and instead will remain on the screen, painting itself as a green X. You can delete this group "component" if you want, or if you want to reassemble the group and keep the code, simply include the original green X group component when selecting the components to group. All the other components will be placed inside this already existing group and the code will remain.

Note: you cannot include two standalone group components in a single group. This will only take the code of one of the two group components.

7.6 Creating User Components

If you have certain settings for components that you like to use, for example, a text component with black background, white text and centered alignment, you can create your own component so that you can easily create new components with these default attributes. You can also take several components that are grouped and make a single user component. To do either of these, right click on the selected component or group and select **Create User Component...** This will popup a window requesting the name of the new component and the category. The name is the displayed name in the component popup menu (the right-click menu that allows you to select new components). The category is the submenu under which this name will appear. You can leave the category empty, in which case the component will appear at the bottom of the popup. If the category does not already exist, it is created at the bottom of the popup, otherwise the component is added to the existing category. Categories are case sensitive. You can use the same names as one of the predefined categories, but this will simply create a second category at the bottom with the same name.

When a new user component is created, it is saved to disk separate from the document as a .dfc file, in the form **name_category.dfc**. Then, when you create a new document, your user component will still appear as an option in the popup. You can share component by simply copying this file into your friend's DAQFactory directory.

User components become even more powerful when you use component events and functions to create powerful component objects, as described a bit later in this chapter.

Note: if you specify the same name and category for a new component as an existing component, the existing component will be replaced with the new one. Use this to modify existing user components.

Note: when a user component is placed on a page, a copy of the component is made. This means that the component on the page is saved with the document, and that any changes to the user component as just described will not change any components already placed.

Note: user components do not appear in the toolbox. You can only add user components to a page by right clicking and selecting it from the component popup menu.

7.7 Using the Grid and Movement Lock

Grid:

To help you arrange your components, you can activate a grid. To use the grid, select **Edit-Grid** from the main menu when you have a page displayed and select the desired number of pixels. As you move, resize, or place components, the position will be locked to integer multiples of the grid spacing. This does not affect nudging with the keyboard.

Lock Movement:

There is also a **Global Lock Component Movement** option under the **Edit** menu. Having this enabled will keep you from accidentally moving a component that you select using the mouse. Use this option when you are finished arranging your components. This does not affect nudging with the keyboard.

If you want to lock an individual component, you can select the component, right click and select Lock Component Movement. This will lock just the selected component. The component will remain locked in place until you unlock it using the same steps. You can lock / unlock groups of components by selecting them all. The locked status of a component is saved with the document, so the component will remain locked the next time you start DAQFactory. The lock does not affect nudging with the keyboard.

Move Undo:

Under the **Edit** menu you can also undo your last component move or resize. Only one move/resize is stored and repetitive undo's will simply alternate between the two locations/sizes.

Lock Against Mouse Click Selection:

Each component also has a lock against mouse click selection. When enabled, you cannot select the component with the mouse. This is useful if you have a large background that you almost never edit. By enabling the lock against mouse click selection, you won't accidently select the background when trying to work with components you have placed on top of it. To enable the lock, select the component(s) and right click and select Lock Component(s) Against Mouse Click Selection. Once you have done this and deselected the component, you will not be able to select the component using a direct mouse click. Instead you will have to find the component in the Workspace under it's page, and select it there. Once you have selected it there, you can then right click on it and undo the lock if desired. For this reason we strongly recommend giving your component a name (right click -> Component Name...) so that you can find it easier in the Workspace.

7.9 Multiple Pages and Page Properties

DAQFactory supports an infinite number of pages for placing components for viewing your data in many different ways. This is of course, computer infinite, which means as many as you want as long as you do not run out of memory! When you create a new DAQFactory document, eleven blank pages are automatically created for you. They are named Page0 through Page9 and Scratch. You can switch between the pages by pressing one of the number keys, or the **S** key for the scratch page. These are the default speed keys for the pages. You can rename

these pages, create more pages, or delete these pages. The scratch page is designed for graph capturing and for graphing the results of some of the analysis functions, and therefore probably should not be used for general display purposes.

The workspace provides an alternative way of switching among pages. If you click on the page's name in the workspace, the page will automatically be displayed. The workspace also provides the means to create new pages and delete old ones. To add a new page simply right click on any page or the **PAGES:** label and select **Add Page**. The properties box for your new page will appear. Enter in a name for your new page and click **OK**. Click on your new page's name in the workspace to display it and you are ready to drop components onto it.

You can switch among these pages using the keyboard or by clicking on the page name in the workspace. The page's property box is also available by right clicking on the desired page's name in the workspace and selecting **Page Properties**. There are only five properties available for pages:

Page Name: You can rename your page simply by typing your new page name in here. Page names have the same criteria as other DAQFactory names: they must start with a letter and only contain letters, numbers, and the underscore.

Speed Key: Speed keys for pages work like they do for components. Enter in any key in this field, and when you press that key the page will be displayed. You can enter more than one key per page and the same key on more than one page. This results in overlaid pages. One thing to note, though, is that the page view must have focus for speed keys to work. If you just clicked somewhere in the workspace or output, then that window will have focus and they will not process speed keys. To give the page view focus, simply click somewhere on the displayed page.

Refresh Rate: Your pages will not be redrawn the instant that a new data point arrives. The rate at which your pages are redrawn is determined by the refresh rate. Typically a refresh rate of 0.5 second is sufficient. If you would like quicker response, you can drop this to quarter second or less. Faster (smaller) update intervals chew up large amounts of your processor time, though, so be careful. Dropping the update interval too far can actually result in a decrease in performance since your computer's processor doesn't have any room to perform the calculations it needs to in the time you desire. Pages are always redrawn from scratch with the latest data when you switch pages. If you are analyzing static data, you may wish to set your update interval to zero. This means the page will only redraw when you switch pages or when you do something during your analysis to cause a redraw such as rescaling a graph, adding a trace, or changing a component's properties. The minimum update interval is 30 milliseconds.

Background Color: This is the color the background is drawn in before any other components are drawn.

Initial Page? If checked, this page is displayed when the document is first loaded. If multiple pages have this property selected, the pages will be overlaid.

In more advanced applications, some components offer actions to change the displayed page. This is especially appropriate for runtime touchscreen applications when a keyboard is not available and the workspace is not visible.

7.10 Creating Popups

You can take any page in DAQFactory and display it in a popup window. The popup window can be either modal, meaning they stay on top until closed and you cannot access anything else within DAQFactory (like the properties boxes), or modeless, meaning they appear, but you can work with the rest of DAQFactory.

You can create a popup out of any page you have created. You do not need to do anything special to the page. You can display or close a popup either from a component action or from a system variable. From a component, select one of the three Actions: **Popup Modal**, **Popup Modeless**, and **Close Popup**. From sequence code you can also use a page function: **.PopupModal**, **.PopupModeless**, or **.ClosePopup**. (i.e. `Page.Page_1.PopupModal()`)

The popup that appears is autosized based on where the components of the page are unless you provide location information through a function as described below. You cannot edit/add components from within a popup, but you can trigger component actions. If you edit the page directly outside the popup, the popup will update with these changes.

If you use the .PopupModal or .PopupModeless page functions you can control the location and the ability to close the popup with the X in the title bar. These functions have the following prototype:

PopupModal([No Close],[Top],[Left],[Bottom],[Right])

PopupModeless([No Close],[Top],[Left],[Bottom],[Right])

No Close is a flag. If set, then the X in the title bar of the popup will not close the window. This is useful for validation. Just make sure if you set this flag that you provide another way to close the popup using the . ClosePopup() function. For example, create a Close button on your page. If you don't and you display a modal popup you will be stuck with that popup and unable to do anything else with DAQFactory.

Top, Left, Bottom, and **Right** define the screen location of the popup. If any of these values are negative or not included, the popup is autosized and centered on the screen. Make sure if you have the No Close flag set that you do not make your popup too small and end up hiding your component that closes the window.

While you can have multiple popups at once, a particular page can only be displayed as a popup one at a time. In other words:

```
Page.PasswordEntry.PopupModal()
Page.PasswordEntry.PopupModeless()
```

will only popup PasswordEntry once as a modal window. The second line will be ignored.

However, if you wish to move a popup that is already displayed, or change the No Close state of the popup, you can simply call popup again with new parameters. Note that calling PopupModal() or PopupModeless with no position parameters on a popup that is already displayed will not autosize the displayed popup.

Application: The modal popup is very useful for requesting information from the user. The popup must be closed before any other part of DAQFactory can be accessed. For example, if we want to create a system where a user must provide a password to access certain pages:

1. Create a page that requests the password using the edit box component and whatever other components, such as panel and text components to make it look nice. Call the page "PasswordEntry". Store the result of the password entry into a variable, say Var.strPassword.

2. Add a button to the page labeled Enter. Assign a quick sequence action to the button:

```
if (Var.strPassword == "ThePassword")
  Page.PasswordEntry.ClosePopup()
  Page.CurrentPage = "NewPage"
else
  System.MessageBox("Invalid Password!")
endif
```

3. Create a page called "Main" with a welcome screen and a button labeled Login. Assign a quick sequence action to the button (or use the PopupModal action):

```
Page.PasswordEntry.PopupModal()
```

4. Clear out the Speed Keys for all your pages.

You now can put things on NewPage that can only be accessed by entering ThePassword into the PasswordEntry page.

Application: The modeless popup is very useful for displaying important data that you want to be able to view at any time. This window floats on top of DAQFactory while still allowing you to work with the rest of DAQFactory. This could also be used to allow DAQFactory to spread across multiple monitors. Windows considers 0,0 to be the top left corner of the main application screen. Depending on how you have your screens arranged, the coordinates will be in reference to that point. So, if your second screen is to the left of your main screen, the x coordinates will be negative.

7.11 Full screen mode

DAQFactory can display in full screen mode (no menu, borders, workspace, etc). when in page view. This gives you more room to create your pages. This works in both the development version of DAQFactory and the Runtime. This is accessed either from the **View** menu, or by hitting F4 to toggle between full screen and the normal window. In the runtime version, there is no accelerator key (no F4), and the menu selection for full screen is in the system menu (top left corner of the window, or Alt-Space). You can access the menu using the keyboard even in full screen mode by hitting **Alt** key and using the arrow keys.

If you want your document to always load in full screen mode, go to File - Document Settings and check the box labelled "Load in Full Screen", then save your document, or alternatively, use the -F startup flag as described in the startup flag section.

7.12 Creating multi-lingual applications

DAQFactory has the ability to create multi-lingual applications. Alas, it does not do the translation for you, but it does provide you with the tools to quickly and easily create translations of pages that can be changed at runtime by setting a system variable. To take advantage of the translation features, you should prefix any string that will end up in an on screen display with @ followed by the default language form or, if you want, a simple name. For example, if you had a variable value component, you might put **`@Pressure`** for the caption. Without adding any languages to the documents available language, the variable value component will simply display with the caption **Pressure**. But if you have other languages and you've enabled one as the current language, it will replace Pressure with new language equivalent. This applies to captions, labels, units, and any other text that appears in a component.

Translation also applies to string constants. For example, if you did:

```
global string x = {"@Pressure", "@Temperature"}
```

the string would fill with "Pressure" and "Temperature" if no language is enabled, or the translated strings if one is enabled. It is important to understand, though, that the substitution is done immediately. If you are assigning a string constant to a property of a page component that would normally be translated there are issues as shown in this example:

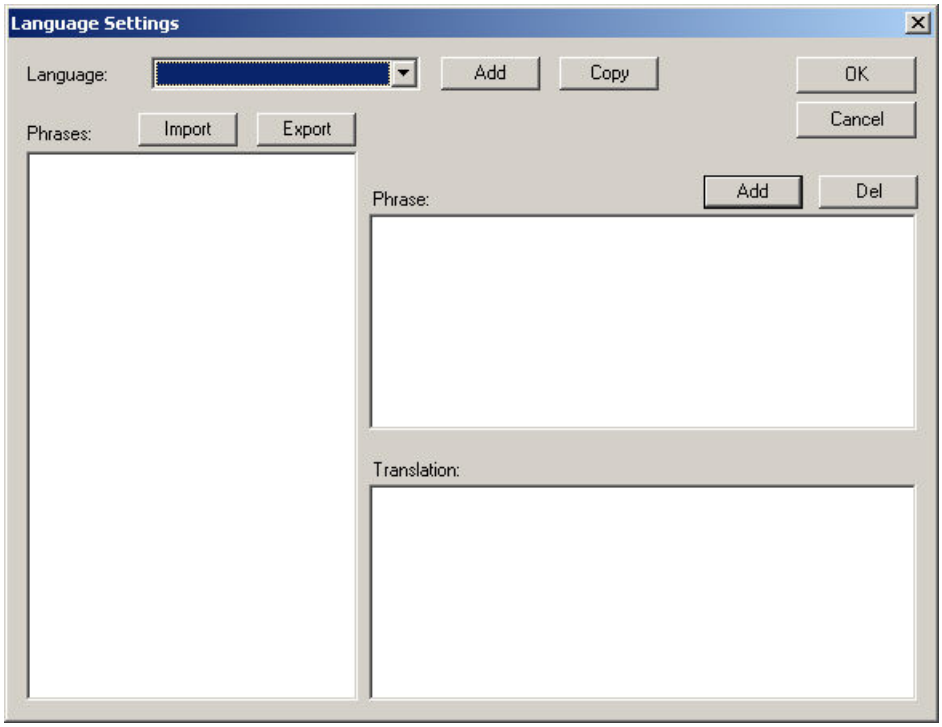
```
component.MyTextComponent.strText = "@Once upon a time..."
```

This will assign the strText property of MyTextComponent with the translated value of "@Once upon a time..." which won't include the "@" in the front. If you then change the language after doing this, MyTextComponent's strText property won't translate since its not marked with the @ sign. To avoid this, you have to use two @ signs:

```
component.MyTextComponent.strText = "@@Once upon a time..."
```

In this case, DAQFactory will try and find a translation for "@Once upon a time..." when the assignment occurs, which presumably it won't. The component now contains "@Once upon a time..." and when the component is drawn it will see the @ sign and try and translate "Once upon a time...". Since the component contains the @ sign still, a change in language will still have effect.

Of course none of this works if you don't have language translations setup for your strings. To create new languages and setup translations, go to Tools - Languages.... This brings up the Language Settings window:



At the top is a drop down list of the current languages in this document. To start, you'll need to click the Add button to add a new language. You will be prompted for the language name. This must be unique among languages, but is otherwise not subject to the normal limitations of DAQFactory names. Once you have added a language you can start adding phrases. Phrases are the strings in the default language you placed in the various components and script in your application. Phrases do not include the opening "@" which is implied. To add a phrase, type in the phrase in the box to the right, and then the translation of that phrase in the currently selected language. Then hit **Add** to add the phrase to the list. Continue as needed. To delete a phrase, select it and hit the **Del** key.

There are several features to make things easier. You may find it easier to keep a spreadsheet of all the phrases you use in your document. If so, you can import and export from comma delimited (CSV) files a complete language phrase list. The CSV file should contain two columns, the phrase and the translation. The first row in the file should be a header and is ignored by the import routine. The best way to ensure that your file is in the correct format is to export a small language file and then add your phrases to this file using a program like Excel. If using Excel or another program that supports all the features of a CSV file, then you should be able to use carriage returns, commas, and quotes within your phrases and translations. DAQFactory supports the proper escaping of these characters in the standard CSV format. If using Excel, simply save your worksheet as a .csv file and Excel will do the rest.

Once you have created one language, instead of hitting **Add** to add a new language, you can press **Copy** to add a new language with an exact copy of the currently selected language. You can then go through and edit the phrases of the new language as needed.

Languages are saved with your document. When you start DAQFactory and load your document, DAQFactory will use the phrase specified in the components and strings (without the @) until you specify a different language. To specify the working language use the system.language variable:

```
System.Language = "Klingon"
```

This setting is not case sensitive. The change will occur immediately with the next screen refresh. To reset back to the default phrases, simply specify a language that does not exist in this document. There is also a function to list all the languages in the document:

System.GetLanguageList(): returns a string array with all the languages in this document.

This function could be used to populate a combobox to allow the user to select their language at runtime. Then, if you added a language to your document, this combobox would automatically update.

Note: the @ marker only applies if it appears as the first character in a string. If it appears elsewhere in the string,

it will appear as a normal @ sign. If you actually want a @ as the first character of a string, simply put two @'s.

Note: the tree list component does not translate. The component must be pre-populated with translated strings. This means that if you change the language and want the tree list to update, you will have to clear the tree list and repopulate it. If you use strings marked for translation in a routine that populates the tree list, this should be easy.

7.13 Printing Pages

To print the currently display page(s), select **File - Print** from DAQFactory's main menu. You can use the **Print Setup** and **Print Preview** menu items to help you control your printout.

The page(s) will automatically zoom to fit on the printed page based on the components contained on all the pages. If you only have a component in the top left corner of your pages, the zoom will be very large. If you wish to keep consistent zooming, you may want to add a small component in the bottom right corner to force the scaling.

7.14 Page Functions

There are several functions for performing actions and accessing parameters with pages programmatically. All start with "Page.".

Variables:

Page.strCurrentPage:

Sets or retrieves the currently displayed page. This is a string.

```
Page.strCurrentPage = "Page_2"
```

Page.ScreenWidth / Page.ScreenHeight:

Read-only. Returns the width or height of the page area of the screen in pixels. These values change with both the overall DAQFactory window size, as well as with movement or resizing of any docking menus or windows. These functions are usually used with the Scaling variable to make it so that pages automatically scale.

Page.Scaling:

Sets or retrieves the current page scaling. By default this is 1. A smaller value results in an overall smaller page. You can have DAQFactory automatically scale your pages by using a sequence similar to this:

```
while(1)
   // handle page scaling, screens designed for 1280x1024
   private yrat = page.ScreenHeight / 1024
   private xrat = page.ScreenWidth / 1280
   private theMin = min(concat(yrat, xrat))
   page.scaling = theMin
   delay(0.2)
endwhile
```

To reduce CPU load, run the above code at a lower thread priority, say 2.

Functions:

Page.Capture(specifier, filename, width, height):

Captures the specified page to the jpeg file specified in filename. Don't forget .jpg in your file name. Width and height are optional and specify the size of the jpeg. The specifier is the name of the page to capture.

Note that although you can capture pages in the background, you MUST view the page at least once before capturing it. This can be done in a startup sequence that simply changes the page (using page.strCurrentPage), with a delay

(0.1) between each.

To specify overlaid pages, simply add them together:

```
"Barrels+Diagram"
```

would create a jpeg of Barrels and Diagram overlaid on top of each other. The pages are drawn in the order listed, putting the last listed page on top.

You can also crop your pages to only the important parts:

```
http://www.mydomain.com/Barrels-120,134,348,322
```

would use the area of the page Barrels with an upper left corner of 120,134 (x,y) and a lower right corner of 348,322. The coordinates on a page are displayed in the status bar as you move your mouse, unless of course you are over a graph, in which case the graph coordinates are displayed. Cropping in this way overrides any width/ height values provided in the function call.

Only supported in versions that support the networking.

Page.PrintPage(name):

Prints the given page. Will switch to the page first and then print it.

Page.PrintPageDirect(name, [file name]):

Prints the given page. Will switch to the page first and then print it. Unlike PrintPage, which will prompt the user for the printer settings, PrintPageDirect uses the current printer and does not display any window. File name is used when your default printer is the Adobe Distiller. If you specify the file name, the distiller will not ask the user for a file name, but instead will automatically generate the given file. This parameter only applies if the default printer is set to the Adobe Distiller.

Page.PrintSetup():

Displays the standard windows print setup window

Page.UpdateEdits():

This function causes all the edit and combo box components to replace their contents with the current value for their Set To Channel, basically displaying what the current value is. This applies to all pages.

Individual pages also have variables and functions. They are accessed by entering "Page.PageName." where PageName is the name of the desired page you'd like to work with.

Variables:

Page.PageName.Interval:

Sets or retrieves the refresh rate of the page

Page.PageName.strSpeedKey:

Sets or retrieves the speed key associated with the page

Page.PageName.BackColor:

Sets or retrieves the background color of the page

Functions:

Page.PageName.ClosePopup():

Closes the popup for the given page.

Page.PageName.PopupModal():

Displays the given page in a modal popup window. A modal popup window displays on top of DAQFactory and must

be closed before the user can work with any other part of DAQFactory. This is most useful for requesting information from the user.

Page.PageName.PopupModeless():

Displays the given page in a modeless popup window. A modeless popup window floats with the DAQFactory window and can stay open while the user works with other parts of DAQFactory. This is useful for displaying important data and keeping it visible even when you switch to other views.

Note: only one popup can exist at one time for each page in your document.

Page.PageName.UpdateEdits():

This function causes all the edit and combo box components to replace their contents with the current value for their Set To Channel, basically displaying what the current value is. Unlike Page.UpdateEdits(), this function only applies to components on the given page.

Page.PageName.WaitForClosePopup():

This function causes the current script to wait until the page's popup window is closed. This works even in the component actions and events in the main application thread. This is useful when you want to prompt the user for information and then process that information from the same script.

7.15 Component Names

DAQFactory provides a powerful scripting language called sequences which allows you to create more advanced applications. As part of this, DAQFactory also provides variable and function access to most all the properties of the components you use on your pages.

A new component is given a default name based on its type, prefixed by an underscore. Since DAQFactory won't let you give your components a name that starts with an underscore, this will keep all unnamed components at the top of the list in the workspace.

In order to access the properties of a particular component on one of your pages, you must give the component a name. You can do so four different ways:

1) find the component in the Workspace under the appropriate Page, right click and select Rename. You can then edit the component's name right in the Workspace.

2) select the component and enter the component name in the edit box at the top of the Properties control bar.

3) select the component and enter the component name in the edit box at the top of the Event/Function control bar.

4) select the component, then right click on the component and select **Component Name...** This will then display a window where you can specify the component's name.

As always, the name must start with a letter and contain only letters, numbers, or the underscore. If you put a number at the end of the component name, and then Duplicate the component, the name will auto-increment. In other words, if you have a component named "Edit1" and you duplicate the component, the new component will be named "Edit2". If your name doesn't end in a number, the new component will have the same name as the old one.

Once you have specified the component name, you can access its properties and functions using the following format:

```
Component.name.variable
```

So, if you had a simple Text component and you wanted to be able to change the text displayed dynamically, you could name the component "MyTextComponent" and then set the text using:

```
Component.MyTextComponent.Text = "new text"
```

When setting a variable, all components with the same name will be set to this value. That is, provided the specified variable exists for all the components. When retrieving variables or calling functions, only the first component with the given name is used.

If you use the same component name for multiple components and those components are on multiple pages, you can specify the components on a single page by simply prefixing with "Page." and the page name and a period:

```
Page.Page_1.Component.MyTextComponent.Text = "new text"
```

This will set all the components named "MyTextComponent" on Page_1. If you have "MyTextComponent" on other pages, it will not affect them.

Inside a component's Action or Event, you can reference its own variables and functions without the Component. Name.

At the end of each component's section in this help is a list of variables and functions available for that component. Most are self explanatory and match the properties available. For those that are less obvious an explanation is provided as well.

All components have the following functions and variables:

- **AddListener(Name)**: this function adds the specified message name to the list of messages this component will generate an OnMessage event for. All messages are forced lowercase. See the next topic for more information.
- **ComponentName**: (read-only) this variable returns the name of the component. This is most useful inside of a component actions or events.
  - **CreateProperty(Name, Initial Value, [Group = "Misc"], [Description], [Edit Type])**
  - **CreateStringProperty(Name, Initial Value, [Group = "Misc"], [Description = ""], [Edit Type = "string"], [Options], [OptionData])**: these two functions create a user property for the component. The only difference is the data type of the resulting property, number or string. A user property is a property that like the built in ones, will persist inside the DAQFactory .ctl document, so will be saved when you do File - Save (or call System.SaveDocument()). The property acts just like a local variable with the exception of the persistence and the ability to edit the property from the Properties control bar. Name must be a valid variable name. Initial Value is the value assigned to the property when it is first created. Once the property is saved with the document, the initial value, and for that matter, this whole function call, is ignored, as the document loading will create the variable and initialize it. Group determines which group the property will appear under in the Properties control bar. Description is currently unused, but in the future will offer a description of the property in the Properties control bar. Edit Type is one of the following strings and determines what sort of editor will be presented to the user in the Properties bar:
  - "color" - a color
  - "double" - any floating point value
  - "integer" - any integer
  - "pinteger" - a positive integer
  - "pzinteger" - a positive or 0 integer
  - "string" - a string
  - "bool" - true or false
  - "date" - date only
  - "time" - time of day only
  - "datetime" - date and time
  - "font" - a font from the system. A drop down list is presented. Returns a string with the name of the font.
  - "file" - a file path. A button with an ellipses is presented that opens a file selection window.
  - "options" - allows you to specify a list of options that the user can select from. The options should be a semicolon list as the Options parameter of the CreateProperty() function. You can also optionally present a semicolon list of data to be returned when the corresponding option is selected. If you do not specify the OptionData, the selected Option itself is returned.
- **GetFocus()**: returns true (1) if the component has focus. This usually only applies to components like the edit box.
- **MoveBy(OffsetX, OffsetY)**: this moves the component by the given pixel offsets in the X and Y directions. X is left and right with + values going to the right. Y is up and down, with + values going down. The size of the component is not changed.
- **MoveTo(X, Y)**: this moves the component to a particular location on the screen without changing its size. 0,0 is the top left corner of the page.
- **Position**: this is an array of 4 values (either a 1 dimensional array of 4 values, or a 2 x 2 array) in the order left, top, right, bottom and define the position of the component on the screen. Some components ignore the bottom and right coordinates.
- **PositionLeft, PositionRight, PositionTop, PositionBottom**: these determine the position of the component on the screen as well and exist mostly for backwards compatibility, but can also be used to easily

change just one of the position coordinates. Some component ignore the bottom and right coordinates.

- **SetFocus()**: this sets the focus to the current component. This usually only applies to components like the edit box the take user focus. This, along with getFocus() can be used to allow navigation among components using keys such as tab. See the setFocus.ctl sample in the samples folder.
- **strSpeedKey**
- **Visible**

The following three functions are identical to their global counterparts described in the sequence chapter, except they run in the locale of the component and will automatically terminate if the component is deleted:

- **StartLocalThread(Function, [Name = ""], [Priority = 3])**
- **StopLocalThread(Name)**
- **GetLocalThreadStatus(Name)**

7.16 Component Properties and Events

Components can even be extended to the point of almost becoming like objects. They have properties and events, can handle messages, and have member functions. By taking advantage of these, you can create your own powerful screen components out of the existing screen components. Since a group is technically a component containing other components, you can create powerful combinations as well. Components contained in a group have local scope to the group, so you can reference them directly by their name from the group scripts. Individual components within a group do not have access to the other components in the group except by using messaging. In general, then, you should place most of your script in the group.

Properties:

Components of course have their own predefined properties like BackColor, strText, etc. Each component type has their own set. You can add additional properties to be used by your components scripts, editable from the properties window, and persisting to the document file. These are just like local variables, except they persist to file. To create a new property, use the CreateProperty() and CreateStringProperty() functions described in the last section in detail. Of course you can also have just regular local variables by declaring them as such, but these variables won't persist to disk unless you manually do so. The most common place to put your CreateProperty() calls and local declarations is in the OnLoad() event.

Events:

There are a number of predefined events that get called at various times. These events can have sequence script to perform special actions. The script runs in the context of the component (meaning, for example, that you can access the properties of the component without putting component.myname. in front). To edit the events, use the Event/Function docking window. If it isn't visible, or tabbed along the right side, select View-Event/Function from the main menu. The Events available are:

OnChange: this event only applies to the following components: Edit box, Multiline Edit, Password Edit, DateTime Edit, Knob, Slider, and Scrollbar. Called whenever the value in the component is changed. The parameter "NewValue" is passed to the event. This allows you to do other things besides simply setting a channel or variable from these components.

OnContextMenu: called when the user right clicks on the component. The parameter "Loc" is passed in which contains the x/y coordinate where the click occurred relative to the top left corner of the component. Loc[0][0] contains x, loc[0][1] contains y.

OnLButtonDblClk: called when the user double clicks on the component. Like OnContextMenu, loc is passed in.

OnLButtonDown: called when the user clicks down on the mouse on the component. Like OnContextMenu, loc is passed in.

OnLButtonUp: called when the user releases the mouse on the component. Like OnContextMenu, loc is passed in.

OnLoad: called when the component is created or loaded from disk. This is the best place to put local declarations and CreateProperty() function calls.

OnMessage: this function is called when a message that this component is listening for is received. To listen for a particular message, use the AddListener function described in the last section. When a known message is received, this event passes in strMessage containing the message name, as well as Param1 and Param2, containing the parameters passed when the message was sent. To actually send a message use the SendMessage() function. This function has the prototype: `SendMessage(Message, Param1, Param2)`. The message will be sent to all listening components. This function is of global scope.

OnMouseMove: called when the user moves the mouse pointer over the component. Like OnContextMenu, loc is passed in.

OnPaint: called before the component is painted to the screen. This is a good place to adjust standard properties like background color or text before the component is drawn. There is no facility to draw from this event. If you want to do scripted drawing, use the Canvas component.

If you use the OnMouseMove event, make sure it is quite fast, otherwise the mouse will become sluggish when you move it over the component. Really, all the event code must be fast as it runs in the primary thread of the application and would cause DAQFactory to appear sluggish.

Functions:

In addition to events, you can create your own member functions. These functions are local to the component and can be called from any other member function or event. They are public, so they can also be called using `Component.MyComponent.` notation from anywhere else in DAQFactory if you named your component. To add a new function, select the component and in the Event / Function window, click on the **Add** button. You will be prompted for the function name. Once you click **OK**, you will see your new function in the Event drop down and you can enter script in the area below. If you decide you don't need the function anymore, select it from the drop down and hit the **Del** button.

7.17 Component Actions

Many components have a separate property page in their properties window called **Action.** This page can perform one or more actions when the component is clicked. Components with actions associated with them will display a hand when the mouse is moved over them.

The particular action is selected by the drop down box at the top of the page:

Nothing: This is the default and means the component will not respond to clicks.

Toggle Between: When clicked, the given **Action Channel** will be toggled between the two values given. The **Action Channel** can be a channel or variable. The toggle between can be numeric or string depending on the type of channel or variable. If you want to set a given channel or variable to a single value when the component is clicked, you can simply put the same value for both **Toggle Between** parameters.

Set To: When clicked, a window will appear prompting the user for a value. If the user clicks **OK**, the **Action Channel** is set to the entered value. You can optionally specify a range to constrain the users input. Leaving the range blank allows any value.

Increment: When clicked, the **Action Channel** is changed by the interval provided. The interval can be positive or negative.

Set To Advanced: This works just like **Set To,** except the user is prompted for an expression instead of a value. When the user clicks **OK**, the expression is evaluated, then the **Action Channel** is set to the result. There is no range validation available.

Start/Stop Sequence: When clicked, the given sequence is either started or stopped depending on its current state.

Start/Stop PID Loop: When clicked, the given PID Loop is either started or stopped depending on its current state.

Start/Stop Logging set: When clicked, the given **Logging Set** is either started or stopped depending on its current state.

Quick Sequence: When clicked, the given sequence code is executed. This allows you to do just about anything

when a user clicks the component. Note that the sequence code is run in the main application thread. This means two things: 1) if the code you entered takes a long time, DAQFactory will not respond until complete and 2) several System functions must be called from the main application thread. This would be where to call them from.

When Quick Sequence is selected, you will see a button labelled **Expand**. Pressing this button will display a new, bigger window with only a sequence editor. This is simply to give you more room to create your sequence.

The quick sequence action has a property called **On Mouse Down** that causes the quick sequence to execute as the user is pressing the mouse, instead of when the user releases the mouse.

Application: The On Mouse Down option is typically used to create a jog style button that increments as long as the button is pressed. To do this, create a sequence that loops and increments the desired value:

```
while(1)
  MyOutput++
  delay(0.1)
endwhile
```

Then create two quick sequence actions for the component. The first is simply beginseq() to start this sequence and has the On Mouse Down checked, and the second is simply endseq() to stop the sequence from incremented.

Submit: When clicked, all the edit boxes on the current page with the **On Submit** option selected will set their values. Use this to create a form.

Clear History: When clicked, the history of the given **Action Channel** is cleared to 1 value. This is provided for backwards compatibility. We suggest using a **Quick Sequence** and the **ClearHistory()** function which will allow you to clear the history leaving any number of values.

Display Alarms: When clicked, the view will switch to the alarm summary view displaying the status of all the alarms on the given connection. If you are not doing any networking, select **Local** for the connection.

Change Page: When clicked, the currently displayed page is changed to the given page(s). Multiple pages can be overlayed by listing multiple pages with the action.

Popup Page Modal: When clicked, the given page is displayed in a separate popup window. The window is on top of all other DAQFactory windows and must be closed before the rest of the DAQFactory windows can be accessed. This window is useful for requesting information from the user.

Popup Page Modeless: When clicked, the given page is displayed in a separate popup window. The window is a floating window. The rest of DAQFactory can be accessed while this window is displayed. This window is useful for displaying information that you would like to keep visible at all times.

Close Popup: When clicked, the given page's popup is closed.

Print Page: When clicked, the given pages are printed. If multiple pages are specified, the pages are overlayed before printing.

Print Preview Page: When clicked, the given pages are displayed in a print preview window. If multiple pages are specified, the pages are overlayed before previewing.

Print settings: When clicked, the standard windows **Print Setup** window is displayed.

Exit: When clicked, DAQFactory will stop all acquisition and terminate.

You can assign multiple actions to a single component. Simply click on the **Add** button to add more actions, or **Delete** to delete an action. Use the up/down arrows to select which action you would like to edit. In general you may find it easier to simply use a **Quick Sequence** and sequence code to perform multiple actions as most of the actions available are accessible through sequence code. The exceptions would be Submit and Display Alarms.

7.18 The Components

7.18.1 Static

7.18.1.1 Text Component

The text component displays a text label. The label can be a single line or multiple lines. The text is clipped to the size of the component, so you may need to resize the component to display multiple lines. Words will be wrapped to

the size of the component as well. If the background color is set at pure white, the background is left transparent. This component can be setup to perform an action when clicked.

Application: The obvious use of the text component is for labels, but the text component can also be used to provide longer explanations. The word wrap makes this easy.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- strAlignment: this contains a string and should be either "left", "right" or "center" (case sensitive)
- BackColor
- strBlink: this contains an expression that if evaluates to true causes the component to blink. This is not available from the properties window and is only available programmatically.
- strFontName
- ForeColor
- strText
- strVisible: this contains an expression that if evaluates to false causes the component to not display. If the expression is blank, then the component will display. This is not available from the properties window and is only available programmatically.
- strXSize: this is the same as the font size, but can be an expression similar to the Variable Value component.
- XSizeMax: this constrains the XSize. This is not available from the properties window and is only available programmatically.
- XSizeMin: this constrains the XSize. This is not available from the properties window and is only available programmatically.

7.18.1.2 Panel Component

The panel displays a simple block of color with an optional edge. You can also assign an action if the user clicks on the panel. The panel will draw behind most other components no matter what the order.

Application: Use the panel component to group other components on a page.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- DrawEdge
- RaisedEdge

7.18.2 Displays

7.18.2.1 Variable Value Component

The variable value component will display your data in simple numeric form. Like most components it can display a single channel or the result of a complex calculation. This component can also display an array of values as well.

The variable value component's property box has four sheets in it. The last three are common to the variable value, descriptive text and symbol components and are discussed a little later. The first sheet, the **Main** sheet is unique to this component. Here is a list of the fields on this sheet and their function:

Caption: This is the text that is displayed in front of the numeric value. A colon is placed between the caption and the value automatically, unless no caption is specified.

Expression: Enter the formula you would like to evaluate. This can be as simple as a single channel, or more complex formula. The result can either be a single dimensional value, or a two dimensional array, in which case only the first 20 values will be displayed.

Units: This is the text that is displayed after the numeric value.

Precision: Specifies the number of digits after the decimal point that will be displayed.

Font: Select the desired font for the display.

Font Size: This specifies the size, in points, of the font used to display the value. This will be overridden by any expression that changes the size of the display (see the size property page).

Speed Key: Because a mouse can sometimes be difficult to control, especially on a bumpy plane or ship, or when you need to quickly toggle an output or the like, DAQFactory provides speed keys. You can assign a speed key, which can be any character you can generate with the keyboard, to any component, and when you press that key and the page that the component is on is visible on the screen, the component will automatically be selected. Once the component is selected, you can quickly change an output value by pressing the space bar, or open the properties box by pressing the enter key.

• If you assign the same key to more than one component and they are on pages that are not displayed at the same time, then only the component being displayed will be selected.

• If you assign the same key to more than one component on the same page, or pages that are overlaid, then all the components with that key will be selected.

• If you assign a speed key to a component that is also used by a page, then the component speed key will take priority over the page speed key when the component is on a page being displayed.

• You can assign multiple keys to a single component. Just list the keys right after each other (i.e. **ABC**). Then pressing of those keys will select the component.

• The page view must have focus for speed keys to work. If you just clicked somewhere in the workspace or output, then that window will have focus and they will not process speed keys. To give the page view focus, simply click somewhere on the displayed page, or select **Quick - Goto Page View** from the main menu.

Quick - Analog Out: Pressing this button will automatically set up the parameters for this component to trigger an analog output or command channel. When you press the button, a dialog box will appear allowing you to select which output channel you would like assigned to this component. Simply select the desired channel, and DAQFactory will do the rest. You may want to adjust some of the settings, such as the caption, afterwards. This button only really works well with a newly created component as it only sets the minimal amount of settings, and does not reset other properties.

Quick - Average: Pressing this button automatically generates an expression to display a ten second average of a channel. When you press the button, a dialog box will appear allowing you to select your desired Channel. Once complete, you can make the average time longer by changing the **9** in the expression to your desired length minus one. Again, this button works best on a newly created component.

Application: The variable value component can be used to display both numeric as well as string data. Of course with string data, the precision property is not used.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor: this will reset the table of colors to a single background color.
- strBlink
- strCaption
- DisplayOutline
- strExpression
- strFontName
- ForeColor: this will reset the table of colors to a single foreground color.
- Precision
- strUnits
- strVisible
- strXSize
- XSizeMax
- XSizeMin

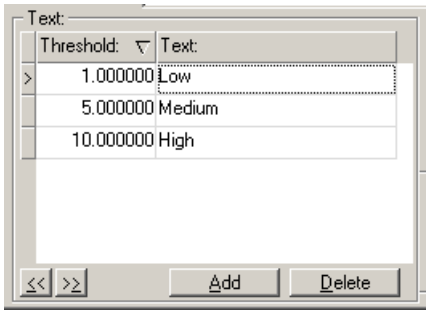
7.18.2.2 Descriptive Text Component

The descriptive text component is very similar to the variable value component, except that after the expression is evaluated, the result is compared to a table of text labels to determine which gets displayed. No numbers are displayed, only labels. This is most useful for digital signals that only have two values, for example On / Off , Open / Closed, etc. The descriptive text component can also be used to display simple text labels, which is what is displayed when a descriptive text component is first created, but the static - text component is probably better for this use.

To have the descriptive text component change with the result of the expression requires an expression and the entry of the various labels in the text table. The expression typically should evaluate to a one dimensional value, but will be converted to a one dimensional value if necessary.

The text table works like all the other tables in DAQFactory. Click on the **Add** button to add new rows, the **Delete** button to remove rows, and double click on a cell to edit its contents. The table contains two columns, the threshold and text label. The threshold of each row indicates the highest evaluated value that will result in the given text label being displayed. If the evaluated value is above the highest threshold in the table, the text label of the highest threshold is used.

An example of the table will explain this concept better:



The text component with this table would display the word **Low** for all values up to and including 1.00, **Medium** for all values up to and including 5.00, and **High** for all values above 5.00. You could have as many or few rows as necessary.

Note that the table is automatically sorted by threshold within DAQFactory, so you do not have to worry about the order of the rows.

Most of the rest of the fields in the text component's main property sheet are the same as display value's property sheet and we refer you there. Here is a description of the quick buttons which have different functions with text components:

Quick - Digital In: Pressing this button automatically creates an expression and text table for displaying Dig In type On / Off values. The text is automatically set to ON / Off for a value of 1 and 0, and is set to display the ON in green and Off in red.

Quick - Digital Out: Pressing this button does the same as the **Quick - Digital In** except it also sets up the component to toggle the output value of the channel when the component is clicked.

Quick - Sequence Trig: This button creates a component that will start and stop a sequence when clicked and display either RUNNING or Stopped in green or red depending on the current status of the sequence.

Quick - PID Loop Trig: This button creates a component that will start and stop a PID loop when clicked and display either RUNNING or Stopped in green or red depending on the current status of the loop.

Application: The typical uses for this component is for controlling digital outputs, and starting and stopping sequences, PID loops, export and logging sets. But as shown in the example above, this component can also be used to display values in more fuzzy terms like Low, Medium and High.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor: this will reset the table of colors to a single background color.
- strBlink
- strCaption
- DisplayOutline
- strExpression
- strFontName
- ForeColor: this will reset the table of colors to a single foreground color.
- strText: this will reset the table of text values to a single text
- strVisible
- strXSize
- XSizeMax
- XSizeMin

7.18.2.3 Symbol Component

Adding symbols to your pages is as simple as pasting them where you would like them. We've even provided a library of over 3800 symbols that you can use in your projects (DAQFactory Pro and higher). You can open this library by selecting **Tools - Symbol Factory**. This will open the symbol factory browser application which is an easy to use product for viewing all the symbols. You can flip or rotate the symbols, change the coloring, and even import your own symbols into the library. To get a symbol out of symbol factory and into DAQFactory, simply press the **Copy** button in **Symbol Factory** to copy the currently selected symbol onto the Windows clipboard, then right click on your page in DAQFactory and select **Paste**. You can actually put any symbol that can be converted to either a WMF or bitmap format into DAQFactory by simply copying it out of another application and pasting it into DAQFactory.

You can also put symbols into DAQFactory using the drag and drop method. From the **Symbol Factory**, simply click on a symbol and drag the symbol onto the **Control Page.** The cursor will change when you are over the page. Release the mouse where you want the symbol dropped. This method is very quick for placing multiple symbols from **Symbol Factory**.

When you paste or drop a symbol directly onto a page in DAQFactory, you are simply creating a symbol component and pasting the symbol into the component. The symbol component provides many more features, some of which have been discussed already. You can select any symbol you have pasted onto a page, or create a new symbol component, and open the properties box for the symbol component to change these features.

In addition to a static symbol that does not change, symbol components can display different symbols depending on the result of an expression, or even animate symbols. On the main sheet of the symbol component's property box is where the expression and symbols are entered. The system for selecting which symbol gets displayed works very similar to the text component and its selection of a text label.

Like the text table, use the **Add** and **Delete** buttons to add or remove rows from the table. Enter a threshold value in the same way that you would in the text table. To insert a symbol in the symbol cell, you can click Load From File to load a BMP, JPG, PNG, GIF or TIF file, or you can click Paste From Clipboard to paste a symbol or image from the clipboard. As a shortcut, double clicking on the symbol cell, the symbol from the clipboard will be pasted into the cell. Once pasted, you can adjust a vector symbol by setting the rotation, **HFlip** and **VFlip** columns.

Most symbols from the Symbol Factory are vector and can be flipped and rotated. Bitmap, Jpeg, PNG and other raster graphic formats can only be resized by resizing the image component.

The symbol table has some additional features for animation. The **Rotate** column allows you to rotate the desired symbol when it is displayed (as determined by the threshold). The **Rotate** column determines the amount the symbol is rotated, in degrees, each time the page is redrawn. Bitmap / raster images cannot be rotated.

To make an animation, simply create a separate row for each symbol in the animation and assign the same threshold to all the symbols. The order column determines the order in which the symbols are displayed in the animation. If you don't care about the order, just leave the column blank.

Note: As soon as you add more than one symbol to a symbol component, you must have an expression specified to determine which image to display. For a simple animation where the threshold is the same, simply put that threshold value as the expression. By default, the threshold is 0. If this is left as is, you can simply put 0 in the expression.

Note: Bitmap / raster symbols cannot be rotated from within DAQFactory. Some bitmaps however, can be flipped.

Symbol components also have an additional property sheet called **Location**. This sheet allows you to enter expressions that will change the position and rotation of the symbol on the screen. All three options, **X Location, Y Location,** and **Rotation**, work the same. Enter an expression, that when evaluated will be used as either an offset in the X or Y axis in pixels from the original location of the symbol, or a rotation, in degrees, relative to the original orientation of the symbol. The range parameters will constrain the total change in position and rotation. If you need to select a symbol that is moving, or is no longer at its initial position because of these features, you must click on its original location, not on the current location of the symbol. This way you don't have to chase down any fast moving symbols! If you don't want this, and need to be able to move a symbol and click on its location as displayed, use the Position variables of the component to move it around.

Note: While symbols are supported in all versions of DAQFactory, the Symbol Factory is not. You can create your own symbols in other applications, but you will not have access to the symbols in the symbol factory without the appropriate DAQFactory version.

Symbol components containing metafile / vector symbols (i.e. most Symbol Factory symbols) can also be partially filled with solid colors based on an expression. This is a great way to do a cutaway tank animation, but can be used on other symbols as well. Most of the symbols included in the Symbol Factory can be filled this way. Bitmap symbols cannot be partially filled. By default, symbols are displayed in their native colors. To fill a symbol with

partial colors, select the Split Colors tab from the Symbol component's properties and enter a valid expression that determines how much to fill the symbol. The result of the expression is compared against the Scaling factors specified on this page to determine what percentage of the symbol is filled with what color. The two colors can also be specified here. So, if your expression evaluated to 70 and the scaling was 0 to 100, then 70% of the component would be filled in the Active Color, while the other 30% would be filled in the Inactive Color. By default, the bottom 70% would be filled with the Active Color, but you can flip this by selecting Reverse, or run it horizontally instead of vertically by checking Horizontal.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- LoadImage(path name): this function will replace any existing symbols in the component with an image file loaded from disk. Supported formats are BMP, JPG, PNG, GIF, and TIF. Transparency in formats that support it, is supported in DAQFactory. Animated GIFs will only display the first frame.
- BackColor: this will reset the table of colors to a single background color.
- strBlink
- strExpression
- OutlineColor: this will reset the table of colors to a single outline color.
- strRotation
- RotationMax
- RotationMin
- SplitColorActive
- SplitColorInactive
- strSplitExpression
- SplitHoriz
- SplitMaxScale
- SplitMinScale
- SplitReverse
- strVisible
- strXLoc
- XLocMax
- XLocMin
- strXSize
- XSizeMax
- XSizeMin
- strYLoc
- YLocMax
- YLocMin
- strYSize
- YSizeMax
- YSizeMin

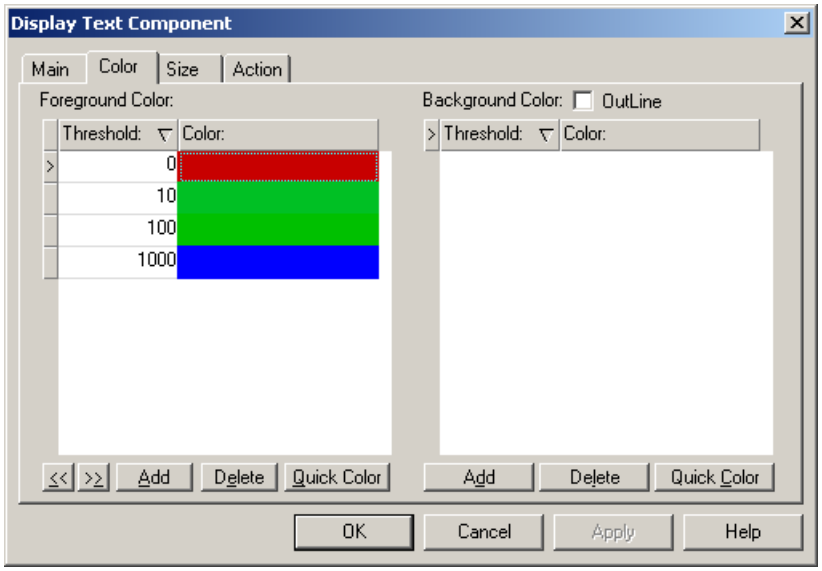
7.18.2.4 The Color Property Page

You can change a descriptive text or variable value component's foreground and background colors. You can even have the foreground or background color change with the value being displayed. The same is true for symbol components, but since symbols already have foreground coloring, you can create a box around the image and change the box's color instead.

To change the component's color, open the component's property box and select the **Color** sheet. The color sheet contains two tables, one for foreground color and one for background color. For image components, the foreground color table is replaced with an outline color table. If you only want to display your component in a single color that doesn't change with the value, press the **QuickColor** button under the desired table to quickly set a single color. If you would like to create an outline around a text or value Component, click on the outline checkbox above the background color table.

To have the fore or background color change with the result of the main expression, you will need to add more rows to the desired table(s). These tables work very similar to the text table on the main page of the descriptive text component. Please see this section for a description of how thresholding works. To select a color for a row in a table, either double click on the color currently displayed in the desired row, or move the cursor to that cell and press the space bar. A color selection dialog will appear allowing you to select your desired color.

The difference between color tables and the text table is that colors can be crossfaded. The following sample table will demonstrate this:



If the value of the main expression is less than 0 then the color will be red. As the value approaches 10, the color will cross fade from red to green. The color will remain solid green up to 100, then will cross fade from green to blue as the value approaches 1000. Above 1000, the color will be solid blue.

The default foreground color for text and value components is black. If nothing is entered in the foreground color table, black will be used.

The default background color for text, value, and symbol components, and outline color for symbol components is transparent. If nothing is entered in the background color table, no background is painted, and the component will appear transparent.

7.18.2.5 The Size Property Page

For variable value, descriptive text, and symbol components, you can enter an expression that, when evaluated, will determine the size of the component, if the component is visible, or whether the component should blink to get the users attention. All these are available from the **Size** sheet of the component's properties box.

Size: For descriptive text and variable value components, only the X Size fields will be available. This is because the result of the size affects the total font size in both axes. If you enter in an expression for the X size field, then the result of the expression will be used to determine the font size in points. This will override the value entered on the main sheet. You can limit the possible values by setting the minimum and maximum point size in the range fields.

For symbol components, both the X and Y size can be adjusted independently. The result of each expression determines the size, in pixels, along the given axis with the constraints given. If an expression is not provided for an axis, then the size of the component as originally drawn is used.

Visible: If a visible expression exists and the result is zero, then the component will not be displayed. Any non-zero result and the component will be visible (the default). Note that once the component is invisible, it may be hard to find to adjust any parameters!

Blink: If a blink expression exists and the result is non-zero, then the component will flash between visible and nonvisible. The flash rate is determined by the update interval of the page it is on.

7.18.2.6 Table Component

The table component will display multiple values in column format similar to the tables used within DAQFactory for adding channels, conversions and the like. Each column in the table will display the result of an expression. In the properties for the table component is a table for specifying the details of each column in the component. Each row in the properties table corresponds to a column in the component. The order of columns is determined by the order

of the rows in the properties table. You can move rows up and down by selecting the row and clicking either **Move Up** or **Move Down**.

Expression: the result of the expression determines what will be displayed in the column. The result can be either an array of numbers or strings.

Title: the title displayed at the top of the column. You can optionally turn of title display using the Display Titles? option described below.

Width: the width of the column in pixels.

Prec: the precision of the numbers displayed in the column. This is ignored if the expression results in string data.

Align: the alignment of the data within the cell. This can be left, right, or center. The title is aligned accordingly.

F Color: the foreground color of the text displayed.

B Color: the background color of the column.

Row Height: the height of each row. The columns will word wrap if the row height is large enough to allow multiple lines of text per row.

Show Grid: if checked, a grid is drawn around each cell in the table.

Grid Color: the color of the grid, if drawn.

Font: the font used to display all the text in the table.

Font Size: the font size in points used to display all the text in the table.

Ellipsis?: if checked, then an ellipsis (...) is displayed at the end of any text that does not fit inside its cell in the table.

Display Titles?: if checked, then a row is displayed with the titles for each column.

Title Fore Color: the color of the text used to display the titles.

Title Back Color: the color of the background of the title row.

Speed Key: a key that when pressed will select this component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- AddColumn(expression, [title], [width], [precision], [alignment], [color], [bkcolor]): Adds a column to the table with the given parameters. Only the expression parameter is required. Alignment should be "left", "right" or "center". To create a new table dynamically, call ClearColumns() then AddColumn() for each column in your new table.
- ClearColumns(): this function will clear all the columns, emptying the table. Typically you will use this before the AddColumn() function to create a table dynamically.
- DisplayEllipsis
- DisplayTitles
- strFontName
- FontSize
- GridLinesColor
- LeftCol: read only: the left most displayed column numbered from 0.
- RowHeight
- ShowGrid
- TitleBackColor
- TitleForeColor
- TopRow: the row number currently displayed at the top of the table. As you scroll down, this number will change to indicate the top row displayed, numbered from 0. Unlike LeftCol, you can write to this variable and adjust the vertical scrolling

7.18.2.7 Canvas Component

The canvas component allows you to use sequence type scripting to draw on an area of the screen. This is useful for doing complex animations and other types of drawings that might not be available using the other DAQFactory components. This is an advanced component and should only be used by those experienced with DAQFactory scripting.

The canvas component executes sequence script every time it is to be drawn. From within the sequence you can call a variety of drawing functions to actually draw to the canvas. Because all the drawing is handled by the sequence script, the script is the only property of the canvas component.

Two arguments are passed to the sequence each time the canvas component is to draw itself. The first, DC, is the device context. It is not terribly important to understand the purpose of this argument, but it is required for all the drawing functions. When you call any of the drawing functions, you should pass the DC argument. If you pass an invalid DC, nothing will happen. The other argument is Position. This is a 2x2 array with the top left and bottom right coordinates of the component. For example, {{10,20},{80,90}}.

Within the canvas component's sequence, you can perform any of the normal sequence functions. In addition to these, a number of other functions are available designed especially for drawing to the screen. Before delving into them, a note about coordinates. Pretty much all the drawing functions use screen coordinate pairs. Each pair consists of an X value and a Y value. The X value is the coordinate across the screen from the left side of the page. The Y value is the coordinate down the screen starting at the top of the page. In general the X,Y pair should be along the second dimension of the array. In other words, {{x,y}} and not {x,y}. Some functions will accept {x,y}, but we recommend against it. The reason for putting each {x,y} pair in the second dimension is to allow you to create an array of coordinates. Most drawing consists of polygons which a group of lines running between coordinate pairs. So, {{10,10},{100,10},{100,100},{10,100}} would define a square.

All the canvas component functions used for drawing start with **Draw.**. Please see the end for an example code.

One of the most important functions you will use with the canvas component is not for drawing at all, but rather for processing coordinates. Lets say you have the coordinate array that defines the square, {{10,10},{-10,10},{-10,-10},{10,-10}}, and you wish to rotate it, make it twice the size and offset it to the location of the component. To do so requires some rather basic matrix math. Fortunately, DAQFactory isolates you from even this with the Transform() function:

Transform(Coord array {{x,y},{x,y}...}, offset {{dx,dy}}, scale {{x,y}}, rotation (deg)): This function takes a coordinate array, for example our square, and will offset, scale and/or rotate them, returning the new coordinate array. The rotation is done around the {{0,0}} location. So:

```
Private Shape = {{10,10},{-10,10},{-10,-10},{10,-10}}
TShape = Draw.Transform(Shape,{{10,10}},2,45) results in {{38.284,10},{10,38.284},{-18.284,10},
{10,-18.284}}
```

Offset(Coord array, offset): This is simply a simpler version of the Transform function with scaling set to 1 and rotation set to 0. Use this to move your drawing to position of the component if you aren't doing any other transformations:

```
Private Shape = {{10,10},{-10,10},{-10,-10},{10,-10}}
TShape = Draw.Offset(Shape,Position[0])
```

Position is a 2x2 array. Position[0] has the x,y coordinate pair of the top left corner of the component, while Position [1] has the bottom right coordinate.

Unlike most of the rest of the functions described here, the above two functions can be called from anywhere in DAQFactory and not just in the Canvas Component sequence.

Once you have the proper coordinates, you can then use one of the following functions to actually draw. Note that all these functions take the DC argument provided when the sequence is called.

Clip(DC, Rectangular coord array): This is the same as ClipRgn, except it clips to a rectangle. Like ClipRgn(), you must clear the clipping by passing 0 for the coordinate array.

ClipRgn(DC, Coord array): This will limit all future drawing commands to the shape provided by the coordinate array. To clear out the clipping region, simply set Coord array to 0: Draw.ClipRgn(DC, 0). Make sure to clear the clipping region before your sequence ends. Unless you are clearing the clipping region, Coord array must have three coordinate pairs.

FillEllipse(DC, Rectangular coord array): This will fill an ellipse or circle that just fits inside the rectangle formed by the coordinate array provided. The current brush is used to fill the ellipse.

FillPolygon(DC, Coord array): This will fill a polygon given by the coordinate array with the current brush color. Use SetBrush() to adjust the brush color. Use FramePolygon() to draw a border.

FillRect(DC, Rectangular coord array): This is similar to FillPolygon() except designed for rectangles.

FrameEllipse(DC, Rectangular coord array): This will draw an ellipse or circle using the current pen that just fits inside the rectangle formed by the coordinate array provided.

FramePolygon(DC, Coord array, offset): This works just like PolyLine(), except a line is also drawn from the last coordinate in the array to the first, closing the shape.

FrameRect(DC, Rectangular coord array): This is similar to FramePolygon() except designed for rectangles.

GetTextExtent(DC, Text): returns the size of the given string in pixels if it were plotted with the current font. Use this for centering or other text manipulation.

Image(DC, Data {{x11,x12,x13},{x21,x22,x23},{x31,x32,x33},...}, Position, Scale, Colors, Thresholds): This will draw an image much like the Image component. Data should be in the same format. Position determines the top left corner. Scale determines the size in pixels of a unit of data. Colors and Threshold are arrays that determine the color range exactly like the Image Component.

LinearLegend(DC, Rectangular coord array, Colors, Thresholds, Precision): This will draw a linear legend with the legend colors stacked on top of each other. The coordinate array determines where the legend is displayed. Colors is an array of color values for each item in the legend. This is matched up with Thresholds, which is an equal sized array of numbers that label the colors. Precision determines how many figures past the decimal point are displayed when labelling. You could of course create your own legend using the other drawing functions, but this is much faster.

LineTo(DC, Point): This will move the current pen position to the given Point drawing a line from the previous point. Use SetPen() to change how the line is drawn.

MoveTo(DC, Point): This will move the current pen position to the given Point. Use in combination with LineTo() to create line drawings.

PolyLine(DC, Coord array): This will draw line segments between the coordinates provided using the current pen. Unlike FramePolygon(), this function does not connect a line between the last coordinate and first coordinate, closing the shape.

TextOut(DC, Point, String): This will output the given string using Point as the top left corner of the text. You can change the text color and font using the SetTextColor() and SetFont() functions.

For performance reasons you should use the rectangular versions of the above functions instead of the polygon versions if you are drawing rectangles. Likewise, Polyline() will be much faster than a series of LineTo() calls.

The following functions set the current pen, brush, font, and text color. They can be called from anywhere in DAQFactory.

SetBrush(Color): This sets the current brush color and therefore the color used for all the fill drawing functions.

SetFont(Style, [Height], [Bold], [Italic], [Underline]): This sets the font that TextOut() will use to draw the text. Only the Style parameter is required. Style is the name of the font. The font must exist in your system. Height is the height of the font in pixels (default is 16). Bold, Italic, and Underline should be 1 or 0 to enable the given characteristic (default is none). If you do not use SetFont(), TextOut will use Arial 16 pixels in bold.

SetPen(Color, Width, Style): This will set the current pen for any function drawing a line. The color is the RGB color, width is the width in pixels. Style is a string and can be one of the following: solid, dash, dot, dashdot, dashdotdot, null, insideframe.

SetTextColor(Color): This sets the color that TextOut() will use to draw the text. The background is always

transparent.

GetPixel(s):

These two functions retrieve the pixel values of part of the page. These function will get whatever the pixel color is at the time the component is drawn to the screen, so if other components are underneath this component and haven't been drawn over top of yet, you will get the color from that component. The primary use case for these functions is to capture pixel colors from embedded webcam or other images. Simply place this component on top of the area and use one of the two functions to retrieve the pixels. Then put the result in a global variable for evaluation elsewhere (or evaluate it in the component that evaluation is quick).

GetPixel(DC, Point): This returns the RGB color of the pixel at the specified Point. Like the other functions, Point is relative to the top left corner of the page, not the component.

GetPixels(DC, Rectangular coord array, [RGB]): This returns the RGB colors of the pixels in the rectangle specified. So, for example, if you do GetPixels(DC, Position), you will get an array with pixel color values for the entire component. The optional RGB parameter is a string which can be either "R", "B" or "G". If you specify this parameter it will only return that particular color component of the pixel. So, GetPixels(DC, Position, "G") will return the green part of pixel under the component. In this case, the result is always 0-255 no matter which color you select. Leaving the RGB parameter blank will get you the full 24 bit color for the pixels.

Note: because the component may be drawn often, make sure the canvas component's sequence is as short and fast as possible. Do not use any delay(), wait(), or similar functions inside the sequence.

Very Advanced: For those who are MFC programmers, the DC parameter is simply a pointer to an MFC 6.0 CPaintDC object. Provided you create an extension DLL so object pointers can be passed, you can create your own drawing routines in C for use in the canvas component. Just type cast the DC parameter back to a CPaintDC pointer. Use the extern() function to make your routines accessible from DAQFactory. Note, however, that you will have no way of verifying the DC parameter is valid.

Here is a sample that creates a rotating compass rose similar to the dual compass component. This is basically a circle with ticks every 5 degrees and labels every 30 degrees that rotates based on a variable. You can simply drop a Canvas component onto a page, and paste this code into the components property window and you will see it draw and rotate with every screen refresh:

```
// the next three lines just make the dial rotate 1 degree every refresh.
global rot = 0
rot += 10
rot = rot % 360

// the coords for a tick (at 0 degrees) {x,y}
private tick = {{0,0},{10,0}}
// the coordinate of the 0 degree text assuming 0 radius circle
private textloc = {{30,0}}

// force location to a circle even if user has dragged a rectangle:
// to keep simple, we'll use x coord only, so it ends up as:
// y2 = y1 + (x2-x1)
Position[1][1] = position[0][1] + (Position[1][0] - Position[0][0])

// calc radius
private rad = (Position[1][0] - Position[0][0]) / 2
rad[0][1] = rad[0][0] // must set y coord too

// calc center coordinate of circle
private centerpos = Draw.Offset(Position[0],rad)

// move tick based on radius, but only in x
tick[0][0] +=  rad[0][0]
tick[1][0] += rad[0][0]
textloc[0][0] += rad[0][0]
private temp

// first, set our pen:
Draw.SetPen(RGB(0,0,0),1,"solid")
// next, draw a circle at our screen loc:
```

```
Draw.FrameEllipse(DC,Position)

// now ticks, every 5 degrees:
for (private.x = 0, x < 360, x += 5)
  // rotate
  temp = Draw.Transform(tick,centerpos,1,x + rot)
  // and draw
  Draw.PolyLine(DC,temp)
endfor

// and finally text, every 30 degrees
for (private.x = 0, x < 360, x += 30)
  // rotate the text location
  temp = Draw.Transform(textloc,centerpos,1,x + rot)
  // now calc size of text
  rad = Draw.GetTextExtent(DC,DoubleToStr(360-x))
  // and offset location to center it
  temp = Draw.Offset(temp,rad/-2)
  // finally, draw!
  Draw.TextOut(DC,temp,DoubleToStr(360 - x))
endfor

// for kicks, draw current rot in center of circle
rad = Draw.GetTextExtent(DC,DoubleToStr(rot))
temp = Draw.Offset(Centerpos,rad/-2)
Draw.TextOut(DC,temp,DoubleToStr(rot))
```

Here is another sample to do a floating scale gauge:

```
// vertical version
// the next two lines just make the value increment 10 units each refresh
global alt = 0
alt += 10

// the coords for a tick at center {x,y}
private tick = {{10,0},{20,0}}
// the coordinate of the center text
private textloc = {{21,0}}
// the number of ticks to display:
private tickcount = 10 // should probably be even
// tick spacing
private tickspacing = 100
// scale factor from alt units to pixels:
private scale = (Position[1][1] - Position[0][1]) / (tickspacing * tickcount)

// top position (in pixels)
private top
top[0][1] = ((Position[1][1] - Position[0][1]) / 2) - \ // the middle
        (tickcount / 2 * tickspacing) * scale - \  // size of 5 ticks
        (alt % tickspacing) * scale               // the partial tick
top[0][0] = 0

private starttick = floor(alt / tickspacing)*tickspacing - tickcount / 2 * tickspacing

// move top based on component position
top = Draw.Offset(top,Position[0])
// in this case we are going to clip to the size of the component:
Draw.Clip(DC,Position)

private temp

// first, set our pen:
Draw.SetPen(RGB(0,0,0),1,"solid")
// next, draw a box at our screen loc:
Draw.FrameRect(DC,Position)

// calc size of text.  We only care about height, and only halfsize
private texthalfsize = (Draw.GetTextExtent(DC,"1") / 2)[0][1]

// draw center tick:
temp[0][1] = Position[0][1] + (Position[1][1] - Position[0][1]) / 2
```

```
temp[1][1] = temp[0][1]
temp[0][0] = Position[0][0]
temp[1][0] = Position[0][0] + 10
Draw.PolyLine(DC,temp)

// now ticks.  Draw an extra tick to ensure we make it out of the box.  Clipping
// will take care of the rest
for (private.x = 0, x < tickcount+1, x++)
  temp = Draw.Transform(tick,top,1,0)
  Draw.PolyLine(DC,temp)
  top[0][1] -= texthalfsize
  temp = Draw.Transform(textloc,top,1,0)
  Draw.TextOut(DC,temp,DoubleToStr(starttick + x * tickspacing))
  top[0][1] += tickspacing * scale + texthalfsize
endfor
```

7.18.2.8 LED Components

DAQFactory has four different LED components that are almost identical except for the shape of the drawn LED. With the exception of the arrow LED, which has radio buttons for selecting the direction of the arrow, the properties for these components are identical. The expression entered will be evaluated, and if the result is non-zero, the LED is drawn in the active color. Otherwise it is drawn in the inactive color.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- ActiveColor
- BackColor
- Bevel
- strExpression
- InactiveColor
- Transparent

The arrow LED adds the following:

- Style: 0 = right, 1 = left, 2 = up, 3 = down, 4 = left/right, 5 = up/down

7.18.2.9 Progress Bar Component

The progress bar is a standard windows progress bar displaying a colored bar of length determined by an expression and a range. The bar can be smooth or segmented, horizontal or vertical and you can change the bar and background color.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- BarColor
- strExpression
- RangeMax
- RangeMin
- Smooth
- Vertical

7.18.2.10 Browser Component

The browser component allows you to embed a browser window right on your DAQFactory pages. This has a wide variety of uses, including the ability to display video from IP cameras, display PDF documents right on a page, and more. This component has but one property, the starting URL used when the component loads. If you name the component, there is a function called SetURL() which allows you to change the URL from script as well. This does not change the starting URL, just the current URL.

The Browser component is a bit different then other components. The browser is actually a windowed component, meaning it runs on top of the page. This causes several things:

- A browser component will always be on top of other components. Two browser components on top of each other will have unpredictable ordering.
- When you are in Edit mode, the browser doesn't display. Instead a box with an X displays, showing you where the browser will be located.
- You must be in Edit mode to select, move or resize a browser component. This is because the browser itself will capture any clicks in Operate mode, even if you have the Ctrl key down.
- If the browser is larger than the Page, it may overlap your control bars when in Operate mode. It will clip to the DAQFactory window, but does not get clipped by your control bars (i.e. command / alert, workspace, etc).
- The browser component will not print.

The browser itself is an embedded form of the Chrome browser. In general you do not need to install Chrome to use this component. Plugins like Flash and Acrobat will work, however you will likely need to install the plugins outside of DAQFactory. This may require you to install Chrome to get the plugins to install. You may be able to uninstall Chrome after the plugins are installed, however you will need to experiment with this.

Browser Debugging: you can optionally enable debugging of your web pages using the Chrome debugger. To do so, you need to start DAQFactory with the startup flag remoteBrowserPort (case sensitive), and assign a port between 1024 and 65535, for example:

DAQFactory.exe -remoteBrowserPort=8080

Once you do this, you can point a full installation of Chrome to: http://localhost:8080 and a list of your browser components will appear. Click on one and the standard Chrome debugger will appear for that page. Since the Chrome debugger is just a web page, you can, technically, run the debugger in another Browser component! This would be handy if you needed debugging but can't or don't want to install the full Chrome browser.

Note: we do not recommend using the browser component for general browsing. It should only be used to view pages that can be maintained and tested before going into production, preferably ones that are hosted locally. While it is not possible for a website to manipulate DAQFactory, it is possible for a website to crash the browser and this crash could very likely also crash DAQFactory.

7.18.3 Gauges

7.18.3.1 Linear Gauge and Log Gauge Component

The linear and log gauge components draw a simple straight horizontal or vertical gauge. The orientation is determined by the size of the component. If the size is wider than it is tall, then the gauge is drawn horizontal, otherwise it is drawn vertical. The two components have almost identical properties except for their tick settings.

Main:

Expression: The result of the expression is used to determine the position of the gauge pointer.

Range: Determines the total range of the gauge. For the Log Gauge, both the min and max values must be greater than zero.

Pointer:

Size: The overall size of the pointer.

Margin: How far the pointer is from the main gauge line.

Color: The color of the pointer

Style: The type of pointer displayed

Orientation: Where the pointer is positioned relative to the main gauge line.

Transparent: Determines if the background is drawn or not.

Background Color: The color to draw the background if Transparent is not selected.

Speed Key: A key that when pressed will select this component.

Min/Max:

Min and Max Expressions: The results of these expressions are used to determine the location of the min and max pointers. Both are optional. The color button to the right of these edit boxes determine the color of the particular pointers.

Size: The overall size of the min/max pointers

Margin: How far the min/max pointers are from the main gauge line.

Ticks:

Label:

Show: Determines if the gauge labels are drawn. Only available on the linear gauge.

Precision: The number of digits after the decimal place displayed.

Margin: How far the labels are drawn from the main gauge line.

Color: The color of the text labels.

Style: Determines if the labels are drawn with scientific notation or regular notation. Only available on the log gauge.

Major:

Show: Determines if the major ticks are drawn. Only available on the linear gauge.

Length: The length, in pixels, of the major ticks.

Color: The color of the major ticks.

Minor:

Show: Determines if the minor ticks are drawn. Only available on the linear gauge.

Count: The number of minor ticks per major tick.

Length: The length, in pixels, of the minor ticks.

Color: The color of the minor ticks.

Alignment: How the minor ticks are drawn relative to the main gauge line.

Sections:

The sections determine the background shading of the gauge. Specifying a count of 0 displays no gauge shading. Any other count, and the gauge is shaded depending on the range, as determined by the End parameters.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- strExpression

- LabelColor
- LabelMargin
- LabelPrecision
- MajorTickColor
- MajorTickLength
- strMax: the expression for the max indicator
- MaxColor
- strMin: the expression for the min indicator
- MinColor
- MinorAlignment: 0 = inside, 1 = center, 2 = outside
- MinorTickColor
- MinorTickLength
- MMMargin: the margin for the max/min indicators
- MMSize: the size of the max/min indicators
- Orientation: 0 = top/left, 1 = bottom/right
- PointerColor
- PointerMargin
- PointerSize
- PointerType: 0 = arrow/line, 1 = arrow, 2 = line, 3 = triangle
- RangeMax
- RangeMin
- SectionColor1 - SectionColor5
- SectionCount
- SectionEnd1 - SectionEnd4
- TickMargin
- Transparent

The following are only available in the linear gauge:

- ShowLabels
- ShowMajor
- ShowMinor

The following are only available in the log gauge:

- LabelStyle: 0 = decimal, 1 = scientific

7.18.3.2 Angular Gauge and Angular Log Gauge Component

These two components create a circular gauge with an arrow that rotates depending on the result of an expression. They both have almost identical properties except for the tick settings.

Main Page:

Expression: The result of the entered expression is used to determine the position of the main pointer.

Range: These two numbers determine the total range of the gauge. Note, for the log version, both the range min and max must be above 0.

Pointer Size: A number that determines the overall size of the main pointer. The actual affect depends on the pointer style.

Pointer Margin: The number of pixels between the end of the pointer and the gauge ticks.

Pointer Color: The color of the main pointer, not including the hub.

Pointer style: Select on of four different types of pointers.

Transparent: If checked, then no background is drawn.

Background color: This only applies if transparent is not checked.

Speed Key: If a character is provided here, pressing that key while viewing the page this component is on will select the component.

Arc Page:

Arc:

Range Degrees: Determines the total arc length in degrees. 360 would be a complete circle.

Start Degrees: Determines the location of the lowest range in degrees. 0 degrees is directly to the right, 90 degrees is straight up.

Show Inner / Outer Arc: Determines if the line of the arc is displayed.

Hub:

Show?: Determines if the hub is drawn.

Size: Determines the size of the hub in pixels.

Color: Determines the color of the hub.

Min/Max Pointers:

Max/Min: These expressions determine the position of the max or min pointer. The color box to the right of these boxes determines the color of the pointer.

Size: The size of the max/min pointers.

Margin: The number of pixels between the end of the max/min pointers and the inner arc.

Style: Select from one of four pointer styles.

Ticks:

Margin: This determines the total margin for all pointers. The maximum of the this parameter and the individual pointer margins is used.

Label:

Show: Available only in the regular gauge.

Precision: The number of decimal places to display

Margin: The number of pixels from the outside arc to the labels

Color: The color of the tick labels.

Style: determines whether values are displayed in scientific notation or regular notation. Only available in the log gauge.

Major:

Show: Available only in the regular gauge.

Length: The size of the major ticks in pixels.

Color: The color of the major ticks.

Minor:

Show: Available only in the regular gauge.

Count: The number of minor ticks per major tick. Has no useful effect in log gauge.

Length: The size of the minor ticks in pixels.

Color: The color of the minor ticks.

Alignment: Determines where the minor ticks are plotted relative to the inner and outer arc.

Sections:

The sections determine the background shading of the arc. Specifying a count of 0 displays no arc shading. Any other count, and the arc is shaded depending on the range, as determined by the end parameters.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- strExpression
- HubColor
- HubSize
- LabelColor
- LabelMargin
- LabelPrecision
- MajorTickColor
- MajorTickLength
- strMax: the expression for the max indicator
- MaxColor
- strMin: the expression for the min indicator
- MinColor
- MinorAlignment: 0 = inside, 1 = center, 2 = outside
- MinorTickColor
- MinorTickLength
- MMMargin: the margin for the max/min indicators
- MMSize: the size of the max/min indicators
- MMType: the type of max/min pointer: 0 = arrow/line, 1 = arrow, 2 = line, 3 = triangle
- PointerColor
- PointerMargin
- PointerSize
- PointerType: 0 = arrow/line, 1 = arrow, 2 = line, 3 = triangle
- RangeDegrees
- RangeMax
- RangeMin
- SectionColor1 - SectionColor5
- SectionCount
- SectionEnd1 - SectionEnd4
- ShowHub
- ShowInnerArc
- ShowOuterArc
- StartDegrees
- TickMargin
- Transparent

The following are only available in the linear gauge:

- ShowLabels
- ShowMajor
- ShowMinor

The following are only available in the log gauge:

- LabelStyle: 0 = decimal, 1 = scientific

7.18.3.3 LED Bar Gauge Component

The LED Bar Gauge Component draws a gauge comprising of multiple LED segments similar to those used on some audio mixers.

Main:

Expression: The result of the expression is used to determine the number of LED segments that will be illuminated.

Range: The total range of values displayed by the LED segments.

Show Off Segments: Determines if segments that are not illuminated will be displayed.

Size: The size of the segments along the gauge axis in pixels.

Spacing: The spacing between individual segments in pixels.

Margin: The spacing between the segments and the outside of the component in pixels.

Direction: Determines the direction of increasing values.

Style: The style of the segments.

Transparent: If checked, then the background is not drawn.

Background Color: If transparent is not checked, this determines the color of the background.

Speed Key: A key, when pressed, will select this component.

Sections: The sections determine the colors of the segments. The count must be between 1 and 3.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- DisplayOffSegs
- strExpression
- Margin
- Orientation: up = 0, down = 1, right = 2, left = 3
- RangeMax
- RangeMin
- SectionColor1 - SectionColor3
- SectionCount
- SectionEnd1, SectionEnd2
- Size
- Spacing
- Style: 0 = rectangle, 1 = circle, 2 = triangle
- Transparent

7.18.4 Compasses

7.18.4.1 Compass Component

Displays a simple compass with optional textual display of the bearing.

Expression: The result of the expression determines the bearing displayed.

Pointer:

Size: The overall size of the bearing pointer.

Margin: The distance from the main hub to the pointer in pixels.

Color: The color of the pointer.

Style: The type of pointer displayed.

Ticks:

Margin: The distance from the main hub to the ticks in pixels.

Length: The overall length of the ticks in pixels

Color: The color of the ticks.

Width: The width of the ticks in pixels.

Direction Display:

Show: Determines if the textual display of the bearing is displayed in the center of the hub.

Precision: The number of digits after the decimal point displayed.

Color: The color of the textual bearing display.

Labels:

Inner Margin: Determines the distance from the main hub to the bearing labels in pixels.

Outer Margin: Determines the distance from the outside of the compass to the labels in pixels

Color: The color of the labels.

Transparent: If checked, then no background is drawn.

Background Color: If transparent is not checked, this determines the color of the background.

Scale Background: Determines the background color behind the ticks, labels, and pointer.

Speed Key: A key that when pressed selects this component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- DirectionColor
- DirectionPrecision
- strExpression
- LabelColor
- LabelInnerMargin
- LabelOuterMargin
- PointerColor
- PointerMargin
- PointerSize
- PointerType: 0 = triangle, 1 = circle, 2 = line
- ScaleBkColor
- ShowDirection
- TickColor
- TickLength
- TickMargin
- TickWidth
- Transparent

7.18.4.2 Dual Compass Component

Displays a compass that can display three different headings. The first is displayed as a textual label in the center, the second is displayed by moving a pointer on top of a scale, and the third rotates the scale under a fixed pointer.

Center Display:

Expression: The result of the expression is displayed in textual form in the center of the component in the color determined by the color button to the right of the edit box.

Precision: Determines the number of digits displayed after the decimal point.

Height/Width: Determines the size background box behind the textual display. The units are in characters.

Background color: Determines the color of the background box displayed behind the textual display in the center of the component.

Pointer 1:

Expression: The result of the expression determines the position of pointer 1. Pointer 1 is the fixed line pointer at the top of the control. The result is displayed by rotating the scale so the proper value is displayed behind the pointer. The color of the pointer is determined by the color button to the right of the edit box.

Width: The width of the fixed pointer in pixels.

Pointer 2:

Expression: The result of the expression determines the position of pointer 2. Pointer 2 is the rotating triangular pointer. The position is determined after the scale is rotated for Pointer 1.

Height/Width: Determines the size of the triangular pointer. Units are in characters.

Scale:

Margin: The distance from the outside of the bezel to the scale labels in characters.

Background / Font Color: Determines the colors used to draw the scale.

Labels: Determine the labels displayed above and below the textual display and their colors.

Border Style: The type of border displayed when drawing the background. Only used when transparent is not selected.

Transparent: Determines if a square background is drawn under the component.

Background Color: Determines the color of the background if transparent is not selected.

Inner Color: The background color drawn inside of the scale circle.

Speed Key: A key, if pressed, which will select the current component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- strBottomLabel
- BottomLabelColor
- BorderType: 0 = none, 1 = raised, 2 = lowered
- DisplayBkColor
- DisplayColor
- strDisplayExpression
- DisplayHeight
- DisplayPrecision
- DisplayWidth
- InnerColor
- Pointer1Color
- strPointer1Expression
- Pointer1Width
- strPointer2Expression
- Pointer2Height
- Pointer2Width
- ScaleBkColor
- ScaleFontColor
- ScaleMargin
- strTopLabel
- TopLabelColor

- Transparent

7.18.5 Percent

7.18.5.1 Percent Bar and Pie Chart Component

These two components display a shaded bar or pie chart based on multiple values. Both have similar properties with a few exceptions that adjust how the components are drawn.

Items: This table determines the different items that will be used to fill the bar or pie. You can add as many items as you need with the "Add" button, or remove items with the "Delete" button. Each row has the same three properties:

Expression: The result of the expression is used to determine the value used in the bar or pie. The percentage displayed is this value divided by the sum of the values for all items.

Text: The text to be displayed in the legend.

Color: The color used to shade the percentage used by the item.

Title: A title label displayed at the bottom of the component.

Title Margin: The distance from the bottom of the component to the title in pixels.

Outer Margin: The distance from the sides of the component to the bar or pie and legend in pixels.

Bar Width: The width of the bar in a percent bar component in pixels.

Start Degrees: The direction, with 0 being to the right, and 90 being up, where the first item will start to be drawn. Items are drawn in counter-clockwise order.

Legend:

Margin: The distance between the legend and the bar or pie in pixels.

Show Percent: If checked, the actual percentage of each item is displayed next to its legend.

Show Value: If checked, the actual value determined by each item's expression is displayed next to its legend.

Precision: Both the Percent and Value have a Precision parameter that determines the number of digits displayed after the decimal.

Transparent: If checked, then the background is not drawn.

Background Color: If transparent is not checked, this determines the color the background is drawn in.

Speed Key: A key, that when pressed, will select the current component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- AddSection(expression, [color], [label]): adds another section to the display. Only the expression parameter is required. Typically you will call ClearSections() first then call AddSection() multiple times to create a new display.
- BackColor
- strCaption
- ClearSections(): clears all the values from the display
- LegendMargin
- OuterMargin
- PercentPrecision
- ShowPercent
- ShowValue
- TitleMargin
- Transparent

- ValuePrecision

The percent bar component adds:

- BarWidth

The pie chart component adds:

- StartDegrees

7.18.6 Graphs

7.18.6.1 2D Graph Component

This component displays a two dimensional graph. This can be either a Y vs. Time trend graph, or an X/Y graph. There are many options available and so the graph component is discussed in its own chapter.

7.18.6.2 3D Graph Component

The 3D graph component is for displaying 3D data. 3D data is an array with rows and cols, or cols and depth. A channel with history only has 2 dimension, rows and value. A spectrum channel with history is 3D, where the rows are the historic values, and cols are a particular spectrum. An image channel with history is 4D, where each row corresponds to a single image with cols and depth. It is these last two types of data which can be displayed with this component. This display is 3 dimensional view, either as a contour or as a waterfall type graph.

Main:

Graph Style: one of three styles, 3D Surface, 3D Bar, and Scatter/Waterfall. Within each style there are several plotting method choices. Do not use the 3D bar type with large amounts of data as it takes much longer to plot then the other types.

Plotting Method: determines how to plot the selected style. The options vary depending on the style. Experiment with the different methods and styles to see how the 3D graph might be able to plot your data.

Expression: the result of the expression is used to determine what is drawn. The result should be three dimensional, i.e. it will have rows and cols, or cols and depth.

Scale From / To: Determines the Y (vertical) axis scaling. This is an expression.

Viewing Height: the height above the XZ plane in degrees to view the graph. Valid values are 0 to 45.

Rotation: the rotation around the y axis to view the graph. Valid values are 0 to 359.

XZ axis scaling: the x and z axis values are normally just the point number in the given dimension (i.e. rows, cols or depth). If you prefer to scale either axis, provide an expression the results in an array of values with the appropriate number of points. You can then scale within those units using the Scale From and Scale To parameters.

Details:

Y Color Shading: The color table determines the color of each pixel. The threshold parameter determines the upper level at which a given color is used. This only works with certain Graph Styles and Plotting Methods.

Shade between colors: If checked, the color of each pixel will be a cross between the two neighboring colors in the color table. Otherwise, the color is simply the color with the smallest threshold above the actual data point value.

Show Contour?: If checked, a flat contour graph is displayed. There are four options which determine how and where the contour is plotted.

Add skirts: If checked, skirts are added. Skirts are solid areas extending from the contour down to the XZ axis plane.

7.18.6.3 Image Component

The image component is for displaying 2D data. 2D data is an array with rows and cols, or cols and depth. A channel with history only has 1 dimension, rows and value. A spectrum channel with history is 2D, where the rows are the historic values, and cols are a particular spectrum. An image channel with history is 3D, where each row corresponds to a single image with cols and depth. It is these last two types of data which can be displayed with this component. This display is an Indian blanket or contour type where each value is represented by a pixel or block of pixels of a particular color.

Image components can also be used to view images stored in a binary string encoded as either JPEG or RGB32. The symbol component can also load JPEGs from files, but this component allows you to store multiple JPEG's or RGB32 images in memory as strings and display them. The JPEG functionality is primarily used with the webcam driver.

Expression: the result of the expression is used to determine what is drawn. The result should be three dimensional, i.e. it will have rows and cols, or cols and depth, or a string containing JPEG or RGB32 data.

Colors: If Data that is RGB is not checked, the color table determines the color of each pixel. The threshold parameter determines the upper level at which a given color is used. This parameter is not used for JPEG or RGB32 data.

Shade between colors: If checked, the color of each pixel will be a cross between the two neighboring colors in the color table. Otherwise, the color is simply the color with the smallest threshold above the actual data point value.

Data is RGB: If checked then the color table is not used and the actual data point is assumed to be an RGB color value. This parameter is not used for JPEG or RGB32 data. Note that Data is RGB indicates that you are providing a 2D array of numbers and is different from a single string containing a full RGB32 data stream.

Data is B & W (32 bit packed): If checked then the data is assumed to be 1 bit color (black and white) packed into 32 bit numbers. Each value in the array therefore represents 32 pixels in the X direction. The colors of the 0's and 1's are determined by the first two colors in the color table. This parameter is not used for JPEG or RGB32 data.

X / Y Scaling: The result of these expressions determines the number of pixels used to draw each data point in the X and Y directions. Use this to zoom in on the image. This parameter is not used for JPEG or RGB32 data.

X / Y Offset: Data is drawn from the top left corner, where the first row, col or depth is the top left. The offset makes the top left corner a different location in the data. For RGB32 data, the X offset specifies the width of the RGB32 image. The height is then calculated based on the length of the string (length of string / (width * 4)). For both RGB32 and JPEG data, Y Offset corresponds to the Alpha value used for rendering the image and can range from 0 to 1.

Speed Key: A key, when pressed, that will select the current component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BW
- strExpression
- RGBColor
- Shaded
- strXOffset
- strXScale
- strYOffset
- strYScale

7.18.6.4 Spectrum Display Component

Displays a multiple bar graph similar to a stereo equalizer with peak values. Although the scaling is quantitative, there are no ticks or labels on this component. If the peak markers are displayed, clicking on the component will reset them.

Mode: The component can take data in one of two methods:

Array: The array expression is evaluated and each array element of the result is displayed as a separate bar. The

total number of bars displayed is determined by the size of the resultant array.

Individual Expressions: A separate expression, which should evaluate to a scalar value, is used to determine the height of each bar. The total number of bars is determined by the Bar Count parameter, and the expressions are entered into the table.

Bar Width: The width of each bar in pixels.

Spacing: The distance between consecutive bars in pixels.

Color: The color of the bars.

Range: The range of valid values for the bars.

Show Peaks: If checked, then a horizontal line is displayed with the peak value of each bar.

Color: The color of the peak lines if drawn.

Transparent: If checked then only the bars and peak lines are drawn.

Background color: The color of the background displayed behind the bars if Transparent is not checked.

Speed Key: A key that when pressed will select the component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- BarColor
- Max
- Min
- PeakColor
- ShowPeak
- Transparent

7.18.7 Buttons_&_Switches

7.18.7.1 Button Component

Displays a standard windows button. Many different actions can be assigned when the user clicks the button. These are the same actions available on many other components.

Text: The caption that is displayed on the button.

Text Color: The color of the caption displayed on the button.

Background Color: The color of the button.

Font / Font Size: The font information used to draw the caption.

Speed Key: A key that when pressed will select this component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- strCaption
- strFontName
- FontSize
- ForeColor

7.18.7.2 Check Box Component

The check box is another standard windows control for selecting a Boolean value.

Caption: The text displayed to the right of the box.

Set Channel: The channel or variable that will be set with either a 1 or 0 depending on the state of the check box. This also determines whether the check is displayed in the box.

Text Color, Font, and font size: Determines the attributes of the caption. The check box size is determined by the caption font size as well.

Speed Key: A key that will select this component.

You can also set the component up to display an array of check boxes. The array can have multiple columns and rows. The Set Channel is set to an array result corresponding to each check box. So if you have an array size of 3 by 2, you would get an array with 3 columns and 2 rows in your set channel.

Array? Check this to enable array mode. When in array mode, the caption is not displayed.

Array Size: Specifies the number of columns and rows.

Array Margin: Specifies the number of pixels to put between each column or row. The size of each individual check box is determined by the size of the component itself and the array size and margins.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- strCaption
- strFontName
- FontSize
- ForeColor
- strSetChannel

7.18.7.3 Spin Button Component

This control displays a standard windows spin button. A spin button is actually two buttons that change a particular channel or variable by a particular interval up or down when clicked.

Action Channel: The output channel or variable that is changed when the component is clicked.

Interval: The amount the channel or variable is changed when the up or right button is clicked. The negative of this value is used when the down or left button is clicked.

Vertical: If checked then the spin button is displayed as up and down buttons, otherwise it is displayed as left and right buttons.

Arrow Color: The color of the triangular arrows drawn in each button.

Button Color: The background color of each button.

Speed Key: A key that will select this component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- strCaption
- strFontName
- FontSize
- ForeColor
- Interval
- strSetChannel
- Vertical

7.18.7.4 Lever Switch Component

Displays a paddle type lever switch that can control an output channel or variable.

Set Channel: The output channel or variable that determines the position of the switch and that will be toggled between 0 and 1 when the switch is clicked.

Enabled: If not checked, then the component will not respond to a mouse click.

Transparent: If checked, then the background is not drawn.

Background: Determines the color the background is drawn in if **Transparent** is not checked.

Speed Key: A key that will select this component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- Enabled
- strSetChannel
- Transparent

7.18.7.5 Rocker Switch Component

Displays a rocker type switch that can control an output channel or variable.

Set Channel: The output channel or variable that determines the position of the switch and that will be toggled between 0 and 1 when the switch is clicked.

Enabled: If not checked, then the component will not respond to a mouse click.

Transparent: If checked, then the background is not drawn. The background is a thin area around the actual switch.

LED Color: Determines the color of the LED displayed on the switch. The color selected is the color when the switch is active. The off color is automatically determined from the on color.

Speed Key: A key that will select this component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- Enabled
- LEDColor
- strSetChannel
- Transparent

7.18.7.6 Toggle Switch Component

Displays a slider type toggle switch that will set an output channel or variable to 1 or 0.

Set Channel: The output channel or variable that determines the position of the switch and that will be changed when the switch is clicked.

Enabled: If unchecked, then the component will not respond to mouse clicks.

Orientation: Determines the direction the switch is drawn.

Height: Determines the height, in pixels, of the sliding part of the switch.

Transparent: If unchecked, then the background of the switch is drawn in grey.

Speed Key: A key that will select the current component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- Enabled
- Horizontal
- strSetChannel
- SwitchHeight
- Transparent

7.18.7.7 Valve Component

Displays a standard valve symbol in a color determined by a channel or variable. Clicking on the component will toggle the channel or variable between 0 and 1.

Set Channel: The variable or channel that determines the color of the valve and will be changed when the component is clicked.

Enabled: If unchecked, then the component will not respond to mouse clicks.

Orientation: Determines whether the valve is drawn horizontally or vertically.

Open/Close Color: Determines the color of the valve when it is in the Open (1) or closed (0) state. The state is determined by the **Set Channel.**

Change Body Color: If checked then the entire valve color is set to the Open or Closed color. Otherwise, only the valve top is changed and the rest of the valve is drawn in grey.

Transparent: Determines if the background is drawn behind the valve.

Background: Determines the color of the background if Transparent is not checked.

Speed Key: A key that will select this component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- ChangeBodyColor
- CloseColor
- Enabled
- Horizontal
- OpenColor
- strSetChannel
- Transparent

7.18.7.8 Three Way Rocker Component

Displays a two directional switch that changes an output channel or variable by a specified interval up or down.

Set Channel: The channel or variable to change with every click. If the channel has not been initialized before clicking, it is assumed to have a 0 value.

Interval: The amount to change the set channel by with each click. Specifying a negative number causes the component to operate in reverse: a click on the upper part of the switch decrements the channel or variable.

Repeat Delay: The switch can be held for rapid adjustment. The repeat delay is the number of milliseconds after the switch is first pressed and not released before the rapid adjustment begins. This works just like holding down a key on the computer.

Repeat Rate: Once the rapid adjustment begins, the output channel will be changed by the interval once every repeat rate (in milliseconds) until the switch is released by releasing the mouse button.

Enabled: If checked, then the switch is active, otherwise the switch will not respond to mouse clicks.

Transparent: If checked then the background is not drawn. The background is a small border area around the switch.

Background color: If transparent is not checked, this determines the color of the background.

Speed Key: A key that selects the current component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- Enabled
- Interval

- RepeatDelay
- RepeatRate
- strSetChannel
- Transparent

7.18.7.9 Four Way Switch Component

Displays a four directional switch that changes two output channels or variables by a specified interval up or down. This works just like the 3 way rocker, but in two dimensions.

Set Channel X/Y: The channel or variable to change with every click. If the channel has not been initialized before clicking, it is assumed to have a 0 value. The X version is set by the left and right arrows, while the Y version is set by the up and down arrows.

Interval X/Y: The amount to change the set channel by with each click. Specifying a negative number causes the component to operate in reverse: a click on the upper part of the switch decrements the channel or variable.

Repeat Delay: The switch can be held for rapid adjustment. The repeat delay is the number of milliseconds after the switch is first pressed and not released before the rapid adjustment begins. This works just like holding down a key on the computer.

Repeat Rate: Once the rapid adjustment begins, the output channel will be changed by the interval once every repeat rate (in milliseconds) until the switch is released by releasing the mouse button.

Enabled: If checked, then the switch is active, otherwise the switch will not respond to mouse clicks.

Transparent: If checked then the background is not drawn. The background is a small border area around the switch.

Background color: If transparent is not checked, this determines the color of the background.

Speed Key: A key when pressed that selects the current component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- Enabled
- IntervalX
- IntervalY
- RepeatDelay
- RepeatRate
- strSetChannelX
- strSetChannelY
- Transparent

7.18.8 Edit_Controls

7.18.8.1 Edit Box, Password Edit, and Multiline Edit Box Component

These three components display a standard edit box which is an area on the screen where a user can enter in data. The normal edit box and password edit box allows only one line of text but has the option to submit changes when the Enter key is pressed. The only difference between these two components is that the password edit displays *'s in place of the characters typed. The multiline edit box allows multiple lines of data, but moves to the next line when the Enter key is pressed rather than submitting the changes. All three have pretty much the same properties:

Caption: An optional caption displayed to the left of the box.

Set Channel: An output channel or variable that will be set to the contents of the edit box. Unlike most other components, this component will not display the current contents of the channel or variable, but instead displays the user entry. If the output channel or variable is a numeric data type the entry will be converted to a number before setting the channel or variable.

Submit only if changed: if selected then the set channel will only be set to the entered value if that value is different than the current contents of the set channel.

Set on Exit: If checked then the set channel is set to the contents when the user clicks outside the edit box. This is only available in the normal edit box and password edit component.

Set on Exit: If checked then the set channel is set to the contents when the user clicks outside the edit box. This is only available in the multiline edit box component.

Set on Set Button press: If checked, a button is displayed to the right of the box. When this button is clicked, the set channel is set to the contents of the box.

Set button caption: The caption for the set button.

Set on Submit button press: If checked, the set channel is set to the contents of the box when another component on the page has a submit action and that component is clicked.

Read-only when: if the expression entered here evaluates to a non-zero value, then the edit box becomes read-only. If this is left empty, then the box is never read-only.

Color Expression: An expression that is used to determine the color of the control. Used in conjunction with the **Active Color** and **Inactive Color** pages.

Units: An optional text that displays to the right of the box.

Font/Font Size: Sets the font and font size for all the text in the component.

Speed Key: A key that selects this component.

Active/Inactive color: These two pages determine the colors for the control. If left blank, the control always draws in black on white. The active color tables are used when the control is editable. The inactive tables are used when the control is read-only, as determined by the "Read-only when" expression. The tables and coloring works just like the common color tables.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor: this will reset the table of colors to a single background color.
- strBlink
- strCaption
- strColorExpression
- strContents: sets the contents displayed in the edit box. Setting this will overwrite anything the user has entered into the edit box.
- DoOnCR
- DoOnSet
- DoOnSubmit
- strFontName
- ForeColor: this will reset the table of colors to a single foreground color.
- InvalidBackColor: this will reset the table of colors to a single invalid background color.
- InvalidForeColor: this will reset the table of colors to a single invalid foreground color.
- OnlyIfChanged
- ReadOnly
- strSetButtonCaption
- strSetChannel
- strUnits
- strVisible
- strXSize
- XSizeMax
- XSizeMin

7.18.8.2 Date & Time Edit Component

This control displays a standard windows date/time control. This control allows the selection of date, time or both with a drop down month calendar.

Caption: Optional text that will be displayed to the left of the date.

Set Channel: The output channel or variable that will be set with the date / time. The date and time is formatted using the standard DAQFactory seconds since 1970 format.

Set on Set Button Press: If checked, the Set Channel is set to date/time selected when the button displayed to the right of the date/time is clicked.

Set button caption: The caption for the Set button displayed when Set on Set Button press is selected.

Set on Submit button press: If checked, the Set Channel is set to the date/time selected when another component on the page has a Submit action and that component is clicked.

Display Date / Time: You can optionally display either the date or time or both.

Date/Time ratio: if you are displaying both date and time, this is the ratio of the size of the date part of the control to the size of the time part.

Font / Font Size: sets the font and font size for all the text used by this component.

Speed Key: A key that will select this component.

Date/Time Format: this determines how the date is displayed. Since this is a Windows control it uses a code determined by Windows:

"d" The one- or two-digit day.

"dd" The two-digit day. Single-digit day values are preceded by a zero.

"ddd" The three-character weekday abbreviation.

"dddd" The full weekday name.

"h" The one- or two-digit hour in 12-hour format.

"hh" The two-digit hour in 12-hour format. Single-digit values are preceded by a zero.

"H" The one- or two-digit hour in 24-hour format.

"HH" The two-digit hour in 24-hour format. Single-digit values are preceded by a zero.

"m" The one- or two-digit minute.

"mm" The two-digit minute. Single-digit values are preceded by a zero.

"M" The one- or two-digit month number.

"MM" The two-digit month number. Single-digit values are preceded by a zero.

"MMM" The three-character month abbreviation.

"MMMM" The full month name.

"t" The one-letter AM/PM abbreviation (that is, AM is displayed as "A").

"tt" The two-letter AM/PM abbreviation (that is, AM is displayed as "AM").

"y" The one-digit year (that is, 1996 would be displayed as "6").

"yy" The last two digits of the year (that is, 1996 would be displayed as "96").

"yyy" The full year (that is, 1996 would be displayed as "1996").

The quotes are not entered. If you want non-date or time characters (other than space), enclose them in quotes.

Example:

```
dddd MMMM d',' yyyy
```

displays

Tuesday January 20, 2004

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- strBlink
- strDateFormat
- DisplayDate
- DisplayTime
- DoOnSet
- DoOnSubmit
- Ratio
- strSetButtonCaption
- strSetChannel
- strTimeFormat
- strVisible
- strXSize
- XSizeMax
- XSizeMin

7.18.9 Selection

7.18.9.1 Combo Box and Radio Buttons Components

Both of these controls allow the user to select from a list of options. The difference is how they are presented. The ComboBox is a standard Windows control that displays the selection with a down arrow next to it. If the user clicks the down arrow, a box appears with the possible selections. The radio buttons displays a radio button with prompt for each option, with the selection having a filled in button.

Set Channel: The channel or variable that will be set to the selected value.

Buttons: This table holds the available options. Use the Add button to add new options, or the Delete button to remove old ones. It has three columns:

Prompt: This is how the option will display on the screen. This is what the user sees.

Value: This is what the Set Channel gets set to when the user selects this option.

Order: This determines the order that the options get listed. It is alphabetical by this column.

Speed Key: A key that will select this component.

The radio buttons component adds the following properties:

Text color, Font and Font Size: These determine the font information used to draw all the text for the buttons.

Note: If you can't get your combo box to open, it is usually because you have made the component too short. The size of the combo box component determines the *open* size of the combo box, not the closed size. If you make the component the same size as the closed box, or only slightly taller, then the box won't be able to open.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

Both Components:
- AddChoice(option, value): adds a new choice to the component. Both parameters are required. Option is a string that is displayed to the user. Value is what the SetChannel is set to if the user selects the option.
- ClearChoices(): removes all the choices from the component
- strSetChannel

Combo Box only:
- strContents

Radio Component only:
- BackColor
- strCaption
- strFontName
- FontSize
- ForeColor

7.18.9.2 Tree List Component

This is an advanced component and only usable with scripting. Because of this, you must name the component. The component name can be assigned using the normal method or through the properties window for the component.

The properties window also allows you to enter an event sequence that is executed, just like the Quick Sequence action on other components, when the user double clicks on a tree item. The event provides two private variables, Private.strItemText which is the actual text that was clicked on, and Private.ItemData which is a numeric value that you assigned to each item. But of course you first must fill the tree list. This can only be done with scripting with the following five functions:

InsertItem(Text, Parent, Insert After, Data): Inserts a new item with the given text into the tree. Returns a handle to the item which can be used for subsequent commands. Parent is the parent item or 0 for the root. Insert after is the item to insert this item after or 0 for the last item. Data is a user definable value that is returned to you in the double click event.

DeleteItem(Item): Deletes the given tree item. Item is the handle of the desired item returned when you did InsertItem.

Expand(Item): Expands the given item. This is the same as the user clicking on the + to the left of the item. Has no effect if the item does not have any children.

EnsureVisible(Item): Scrolls the item list so that the given item is visible.

DeleteAllItems(): Clears the tree list.

7.18.10 Sliders_&_Knobs

7.18.10.1 Knob Component

Provide a variable output value based on the position of the knob.

Main:

Set Channel: The specified channel or variable will be set to the value corresponding to the position of the knob.

Range: The total range of the knob.

Update Rate: The maximum rate the Set Channel will be set by the knob. By default this is 0, which means as fast as the knob is moved. With serial and other slow devices, this can often be too fast and can bog down the system. To avoid this, you can control the maximum update rate of the channel. Note that when the knob is released the Set Channel is set to the knob's final value.

Start Degrees: The position in a circle of the minimum point on the knob. 0 is to the right, 90 is up.

Arc Length: The number of degrees the knob can turn. Positive values are clockwise, negative are counter clockwise. You can make multiturn knobs by specify a multiple of 360, for example 720 for a 2-turn knob. You will probably want to turn off automatic labels however with multiturn knobs. For a jog wheel / knob, see below.

Jog Wheel: if checked then the knob will act like a jog wheel. A jog wheel is a knob with no limits. The range max determines the number of units to increase or decrease with each revolution. The range min is not used. Neither is start degrees, arc length, or labels. You can then select clockwise or counter clockwise rotation for positive values.

Update Interval: Knobs and sliders will update the output very rapidly as the knob is turned. For slower devices, especially serial and Ethernet based devices, this can cause problems and bog down the communications. To prevent this, you can set the update interval to a non-zero value and this will specify the maximum interval the output will be updated.

Knob Style: Determines how the knob is drawn.

Position Display:

Show?: If checked, the actual position of the knob is displayed in textual form in the center of the knob.

Precision: The number of digits to the right of the decimal for the position display.

Units: If entered, this will be displayed to the right of the position display.

Indicator:

Size: The overall size of the indicator.

Margin: The distance from the outside of the knob to the indicator in pixels.

Color: The color of the indicator bar / spot.

Indicator style: Determines the type of bar displayed.

Transparent: If checked, the background is not drawn.

Background: If Transparent is not checked, the color of the background.

Speed Key: A key that when pressed selects this component.

Ticks:

Margin: The distance between the ticks and the outside of the knob.

Ticks Label:

Show: If checked, labels are drawn next to the major ticks.

Precision: The number of digits displayed after the decimal.

Margin: The distance between the ticks and the labels in pixels.

Color: The color of the labels.

Major Ticks:

Show: If checked, major ticks are drawn.

Count: The number of major ticks to display over the entire range.

Length: The size of the tick marks in pixels.

Color: The color of the tick marks.

Minor Ticks:

Show: If checked, minor ticks are drawn.

Count: The number of minor ticks per major ticks to display.

Length: The size of the tick marks in pixels.

Color: The color of the tick marks.

Alignment: The position of the minor ticks relative to the major ticks.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- IndicatorColor
- IndicatorMargin
- IndicatorSize
- IndicatorStyle: 0 = lowered dot, 1 = raised dot, 2 = dot, 3 = line center, 4 = line, 5 = triangle
- KnobStyle: 0 = raised edge, 1 = raised, 2 = sunken, 3 = sunken edge

- MajorTickColor
- MajorTickCount
- MajorTickLength
- ArcLength
- MinorTickAlign: 0 = inside, 1 = center, 2 = outside
- MinorTickColor
- MinorTickCount
- MinorTickLength
- PositionPrecision
- strPositionUnits
- RangeMax
- RangeMin
- strSetChannel
- ShowMajorTicks
- ShowMinorTicks
- ShowPosition
- ShowTickLabels
- StartDegrees
- TickLabelColor
- TickLabelMargin
- TickLabelPrecision
- TicksMargin
- Transparent
- UpdateRate

7.18.10.2 Slider Component

Provides a variable output value based on the position of the slider.

Main:

Set Channel: The specified channel or variable will be set to the value corresponding to the position of the slider.

Range: The total range of the slider.

Update Interval: The maximum rate the Set Channel will be set by the slider. By default this is 0, which means as fast as the slider is moved. With serial and other slow devices, this can often be too fast and can bog down the system. To avoid this, you can control the minimum update interval of the channel. Note that when the slider is released the Set Channel is set to the slider's final value.

Track Style: Determines the type of track the slider slides on.

Track Color: The color of the track.

Ends Margin: The distance from the track to the edge of the component.

Reverse Scale: If checked, the scale will go in the opposite direction.

Orientation: Determines whether the track runs horizontally or vertically

Indicator:

Height / Width: The size of the indicator bar in pixels.

Color: The color of the indicator bar.

Bar style: Determines the type of bar displayed.

Transparent: If checked, the background is not drawn.

Background: If Transparent is not checked, the color of the background.

Speed Key: A key that when pressed selects this component.

Ticks:

Margin: The distance between the ticks and the slider track.

Tick orientation: Determines the position of the ticks relative to the slider track.

Ticks Label:

Show: If checked, labels are drawn next to the major ticks.

Precision: The number of digits displayed after the decimal.

Margin: The distance between the ticks and the labels in pixels.

Color: The color of the labels.

Major Ticks:

Show: If checked, major ticks are drawn.

Count: The number of major ticks to display over the entire range.

Length: The size of the tick marks in pixels.

Color: The color of the tick marks.

Minor Ticks:

Show: If checked, minor ticks are drawn.

Count: The number of minor ticks per major ticks to display.

Length: The size of the tick marks in pixels.

Color: The color of the tick marks.

Alignment: The position of the minor ticks relative to the major ticks.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- EndsMargin
- Horizontal
- IndicatorColor
- MajorTickColor
- MajorTickCount
- MajorTickLength
- MaxDegrees
- MinorTickAlign: 0 = inside, 1 = center, 2 = outside
- MinorTickColor
- MinorTickCount
- MinorTickLength
- PointerMargin
- PointerSize
- PointerStyle: 0 = LED, 1 = Arrow, 2 = Bar, 3 = Bar/Line, 4 = Colored Bar
- PositionPrecision
- PositionUnits
- RangeMax
- RangeMin
- ReverseScale

- strSetChannel
- ShowMajorTicks
- ShowMinorTicks
- ShowPosition
- ShowTickLabels
- StartDegrees
- TickLabelColor
- TickLabelMargin
- TickLabelPrecision
- TickOrientation: 0 = bottom/right, 1 = top/left
- TicksMargin
- TrackStyle: 0 = box, 1 = line, 2 = bevel lowered, 3 = bevel raised
- TrackColor
- Transparent

7.18.10.3 Scroll Bar Component

Displays a standard windows scroll bar.

Action Channel: The channel or variable that will be set to the position of the scroll bar.

Range: The values for the extreme ends of the scroll bar.

Step Size: The amount the position is adjusted when the user clicks on the arrows at the ends of the scroll bar.

Vertical: If checked, the scroll bar is displayed vertically.

Arrow Color: The color of the arrows at the ends of the scroll bar track.

Button Color: The background color of the arrow buttons at the ends of the scroll bar track.

Track Color: The background color of the track.

Tracker Color: The color of the moving box running in the track.

Speed Key: A key that when pressed will select this component.

Component Variables and Functions (advanced): please see the section on Component Names for a description on how to use these.

- BackColor
- strCaption
- strFontName
- FontSize
- ForeColor
- RangeMax
- RangeMin
- strSetChannel
- StepSize
- TrackColor
- TrackerColor

7.19 Questions, Answers and Examples

7.19.1 Displaying a different symbol based on a value

This sample shows how to change a symbol based on the value of a channel or a variable. When either button is depressed, the channel or variable is toggled. This is done in the action page of each symbol component.

In this sample we use several symbols in the symbol factory image library. The DAQFactory trial and some other versions of DAQFactory do not include the symbol factory library. In this case, you can either use the first image from two different categories of symbol factory, or you can create your own symbols in a program such as Paint. In Paint you would still select Copy to put the symbol on the Windows Clipboard.

Step 1: Go to **CHANNELS:** in the workspace, and add a new channel:

Channel Name: MyChannel
Device Type: Test
I/O Type: Dig Out
Chn #: 0
Timing: 0

Leave the rest to defaults.

Click **Apply**

Step 2: Under the application menu select Tools => Symbol Factory. Click on the 3-D green button (or any other symbol) and select **Copy**. Go to **PAGES:** right click on Page_0 and select **Paste**.

Step 3: Go back to the symbol factory and copy the 3-D red button. Right click on the 3-D green button and select **Properties...**.

Main Tab-

Select **Add** and double click in the blank Symbol: window between Thres: and Rotation. This should insert the image. Double click in the Thres: window for the 3-D red button and type 1. Double click in the Thres: window for the 3-D green button and type 0.

Expression: MyChannel[0]

Action Tab-
Action: Toggle Between
Action Channel: MyChannel
Toggle Between: 0 and 1

Click **OK**

Step 4: Click on the button to toggle and set their initial values. When you click on the button, they should change symbols going between the red and green 3-D symbols.

Step 5: You can add Variable Display components and Descriptive Text components to enhance the look of you sample.

7.19.2 Using a password to protect pages

This example shows how to use the MessageBox() function to prompt the user for a password before giving them access to the page "Protected". Password protection like this only works in the runtime environment where there is no workspace. Make sure and remember to clear out the speed key associated with the protected page.

The two methods shown here do almost the same thing. The first will return if the user enters an invalid password, thus requiring the user to click the Login button again. The second version continually reprompts the user until they either get the correct password or they hit cancel. The valid password is "password".

Step 1: In the Workspace right click on Page_0 and right click on the page to the right and select Buttons & Switches => Button.

Step 2: Select the button you just created and give it the follow properties.

Main Tab-
Text: Login

Action Tab-
Action: Quick Sequence
Sequence: Copy the code below

```
Private string strEntered = System.EntryDialog("Enter Password:")
if (!IsEmpty(strEntered))
 if (strEntered == "password")
    Page.CurrentPage = "Protected"
    return
 endif
```

```
endif
System.MessageBox("Invalid Password!")
```

Click **OK.**

Step 3: In the Workspace click on Page_0 and right click on a blank area of the page to the right and create a second button. Select Buttons & Switches => Button.

Step 4: Select the button you just created and give it the follow properties.

Main Tab-
Text: Login

Action Tab-
Action: Quick Sequence
Sequence: Copy the code below

```
private string strEntered
while (1)
 strEntered = System.EntryDialog("Enter Password:")
 if (IsEmpty(strEntered)) // if nothing is entered, assume Cancel was pressed
    return
 endif
 if (strEntered == "password")
    Page.CurrentPage = "Protected"
    return
 endif
 System.MessageBox("Invalid Password!")
endwhile
```

Click **OK**

Note: The second button will continually prompt the user for the password until they get it right where the first button will only ask for it once.

Step 5: Go to PAGES: located in the workspace and right click on a new page and select Page Properties. Give it the page name **Protected** as this is the page we want it to switch to when the password is correct.

Step 6: Click on your buttons and verify that the first button delivers a message valid or invalid. The second button should continue to prompt the user until a valid password is entered or cancel is selected.

7.19.3 Create a popup for user name and password

This example shows how to use a modal popup to prompt the user for a username and password before allowing them access to the "Protected" page. Password protection like this only works in the runtime environment where there is no workspace. Make sure and remember to clear out the speed key associated with the protected pages.

The valid username in this example is "user", password is "password".

Step 1: In the Workspace click on Page_0 and right click anywhere on the page to the right and select Buttons & Switches => Button.

Step 2: Select the button you just created and give it the follow properties.

Main Tab-
Text: Login

Action Tab-
Action: Quick Sequence
Sequence: Copy the code below

```
// first initialize variables:
global string strUser
global string strPassword
strUser = ""
strPassword = ""
// and clear out edit boxes
Component.PasswordPrompt.Contents = ""
```

```
Component.UserNamePrompt.Contents = ""
// then popup window:
Page.Login.PopupModal()
```

Click **OK**

This sequence clears out any previous user / password that may have been entered and then displays the Login page in a modal popup. A modal popup is one that requires the users attention until it is closed. The two lines, Component.PasswordPrompt... and Component.UserNamePrompt... both clear the contents of the edit box.

Step 3: In the Workspace right click on Page_1 and select Page Properties. Give it the page name **Login**.

Step 4: In the page you just called "Login" right click anywhere on the page and select Edit Controls => Edit Box. While this component is still highlighted hit Ctrl-D to Duplicate it. Select the first edit box and go to its properties.

Main Tab-
Caption: User Name
Set Channel: strUser

Click **OK**

While the component is still selected, right click on it and select **Component Name...** Give this component the name **UserNamePrompt**. We have to name the component so we can access it from the sequence script we wrote above, **Component.UserNamePrompt.Contents = ""**.

Select your second edit box and go to its properties.

Main Tab-
Caption: Password
Set Channel: strPassword

Click **OK**

While the component is still selected, right click on it and select **Component Name...** Give this component the name **PasswordPrompt**.

Step 5: In the same page right click in a blank area of the page and select Buttons & Switches => Button. Right click on the button to go to its Properties.

Main Tab-
Text: Login

Action Tab-
Action: Submit

Click Add to add a new action:

Action: Quick Sequence
Sequence: Copy the code below

```
// note that we have a submit action
// as the first action, then this
// quick sequence.  This is to
// force the edit boxes to update before
// we check for proper user+password



if ((strPassword == "password") && (strUser == "user"))
 // if correct, close popup
 Page.Login.ClosePopup()
 // and change page:
 Page.CurrentPage = "Protected"
 return
endif

// incorrect, so prompt.  You
// could optionally close the
// popup too.

System.MessageBox("Unknown username / password")
```

Click **OK**

Highlight all the components on this page by holding the Ctrl key and left clicking and drag a box around all three components. Move all of them to the top left corner of the page. When a page is displayed in a popup, the window is automatically sized to the components on the page starting from the top left corner of the screen. If we don't move our components to the top left corner, we'll have a bigger window than necessary with a lot of white space to the top and left of the components.

Step 6: Create a third page. This is the page that will be password protected. Right click on the page in the Workspace you wish to be protected and select Page Properties. Give it the page name **`Protected`**. This is the page that the user will only be able to access by going through the login.

Step 7: Click on your buttons and verify that the proper components for user name and password are present. Insert the proper name and password.

7.19.4 Create a popup numeric keypad

This sample shows how you can create your own custom pop-up screen for touch screen applications. Click on the button "Keypad" and a pop-up keypad will open. Enter the correct code and it will direct you to protected pages. Enter the incorrect code and a box will prompt you with the message "Invalid Password". Note that password protection like this only works in the runtime environment where there is no Workspace. Also, you must clear the SpeedKey for all protected pages or the user can bypass the keypad entry.

This example works with all versions of DAQFactory. Note for Express users, a pop-up requires its own page leaving you with one other page for the rest of your file. The attached example has three pages, the first is just a title page to introduce the file, the second page is where the pop-up is created and the third page illustrates the protected page. For Express users you will not see the third page because of the page limitation.

Step 1: Go to **PAGES:** in the Workspace and select a blank page to work in. Right click on the blank page and select Buttons & Switches => Button. For this sample create 13 buttons. The quick way to do this is to hit Ctrl-D with the first component still selected to duplicate the component.

Step 2: Go to each of the buttons Properties by right clicking on the button and selecting **Properties...**

Main Tab-
Text: 1

Action Tab-
Action: Quick Sequence
Sequence:

```
Component.EditBox.strContents += "1"
```

Leave all else to defaults

Click **OK**

Repeat this step for nine other buttons. Set each button up for the appropriate number until you have a button for 0-9.

Step 3: Go to the properties of one of your remaining undefined buttons. We will set this button as a clearing button.

Main Tab-
Text: Clear

Action Tab-
Action: Quick Sequence
Sequence:

```
Component.EditBox.strContents = ""
```

Leave all else to defaults

Click **OK**

Step 4: Right click on a blank area of the page and select Edit Controls => Edit Box. While your new Edit Box is

still highlighted right click on it and select **Component Name...**. Give it the name **EditBox**. Next, right click on the edit box and select **Properties...**

Set Channel: Var.strTotal
Check "Set on Set Button Press"
Set Button Caption: #

Leave all else to defaults

Click **OK**

Step 5: Go to the Properties of one of two remaining undefined buttons.

Main Tab-
Text: Enter

Action Tab-
Action: Quick Sequence
Sequence:

```
// evaluate if statement for correct passcode
if (Component.EditBox.strContents == "2468")
 // correct code, close popup go to new page
 Page.Keypad.ClosePopup()
 Page.CurrentPage = "Protected"
else
 // incorrect passcode
 System.MessageBox("Invalid Password!")
 // clear EditBox for new entry
 Component.EditBox.strContents = ""
endif
```

Leave all else to defaults

Click **OK**

This quick sequence checks the value entered into the edit box. If it is correct it takes you to the protected page, if not it prompts the user with an Invalid Password message. For the Enter button to work create a page in the Workspace called Protected. To do this, right click on PAGES: in the Workspace and select Add Page, then name your new page Protected.

Step 6: Go to the Properties of your last remaining undefined button.

Main Tab-
Text: Backspace

Action Tab-
Action: Quick Sequence
Sequence:

```
// set string variable to contents of edit box
private string strTotal = Component.EditBox.strContents
// evaluate the length of string minus one for backspace
private length = GetLength(Var.strTotal) - 1
// Delete the last character of the string
// Delete (String, Index #, # of Characters to delete)
strTotal = Delete (strTotal, length, 1)
// Reset EditBox to value with backspace
Component.EditBox.strContents = strTotal
```

Leave all else to defaults

Click **OK**

Step 7: Organize your components in the top left corner your page. The manner in how you organize them is how they will be displayed in the pop-up window. Note your buttons can be resized by and arranged by using the Layout option in DAQFactory.

Step 8: Go to another blank page and right click anywhere and select Buttons & Switches => Button. This is the button we will use to initiate the pop-up. Right click your button and select Properties.

Main Tab-
Text: Keypad

Action Tab-
Action: Quick Sequence
Sequence:

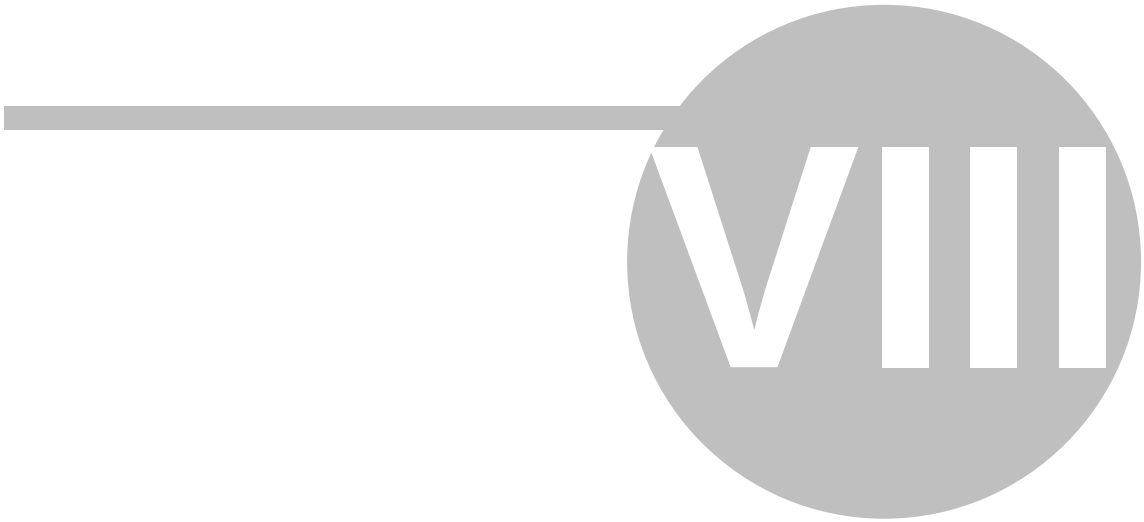
```
Component.EditBox.strContents = ""
Page.Keypad.PopupModal()
```

Leave all else to defaults

Click **OK**

Note: For this component to work you need to rename the page with your buttons and edit box to Keypad. To do this right click on this page in the Workspace and select Page Properties. Also, uncheck the box that says **Initial Page?** . Check this box on the page where you created the Keypad button in Step 8.

8 Graphing and Trending



8 Graphing and Trending

8.1 Graph Overview

While other Components are very useful for displaying the current state of your process, and for controlling it, graphs provide a much better way of watching your process over time, or for determining trends in your data. DAQFactory currently supports two dimensional graphs of values vs. time and X vs Y graphs. You can have as many traces as you need on up to 12 different Y axes, stacked and/or overlapping. You can add error bars to any of the traces, and can colorize traces based on an expression. You can add line annotations for marking important values, and axis annotations for marking important events. DAQFactory provides many features for scaling and zooming your graphs, markers for quick analysis, as well as more in depth analysis tools such as curve fitting. DAQFactory graphs have many attributes that control the look of the graph, enabling you to create publish quality graphs right in DAQFactory. You can export these graphs to bitmap or metafile format for inclusion in your next paper or report, or print them directly to paper.

To make DAQFactory responsive when you want to trigger events, switch views, etc. the package uses a special triple buffer background drawing algorithm. With this algorithm, graphs and images are drawn completely in the background whenever the processor gets free time to draw them. This means that even though it may take a few seconds to draw a very complicated graph, you can still trigger an output value while it is doing that. The other result of this is that when you switch Pages, the Text and Value Components will be displayed first, then the graphs and images.

8.2 Creating a Value vs Time (Trend) graph

A graph in DAQFactory is just another component that can be added to any page. The graph component can be moved, resized, copied, and pasted just like the other components we've discussed so far. Graphs have a completely different set of properties however, and offer quite a few advanced features that are not available on the other components.

To create a new graph, simply create a graph component on a page the same way you would a text, or value component. Right click in the blank area of the page where you would like to put the graph and select **2D Graph** from the popup menu. A blank graph will then be displayed. Depending on your screen resolution, you will probably want to resize the graph. This is done just like an image component. Select the graph and drag one of the eight black box handles on the selection rectangle. Since the graph is drawn in the background, the graph will not resize as quickly as the selection rectangle, so don't worry if it the graph seems to lag behind the rectangle. The size of the selection rectangle when you release the mouse button is what determines the final size of the graph.

You may notice that if you click on a graph (without holding the Ctrl key down), a green single-line rectangle will appear around the graph. This is the focus rectangle for a graph. This is used to indicate which graph is the target of many of the graph functions when you have more than one graph on a page.

You can add a trace using the graph's property box. Select the graph and open the properties box using the normal component methods, or simply right click on the graph and select **Properties...**. Enter in an expression to be used for the trace under **Y Expression**. This can be any expression you would like, from a simple channel name to a complex calculation. The result should evaluate to an array of numbers otherwise there won't be anything to graph. By default, a new graph is created as a value vs time graph. This means that the X axis will be time and that it will scroll as new data appears. If the **X Expression** is listed as **Time** then the time value associated with the result of the Y expression is used for the x location of the data points.

The graph properties box has many sheets, and you can quickly jump to the appropriate sheet by double clicking on an area of the graph to open the properties box to the appropriate sheet. For example, if you double click on an axis, the graph properties box will open to the **Axes** sheet with the correct axis selected.

When a graph is first created, a trace with a Y Expression of **New Trace** is created. On the **Traces** sheet of the graph properties box you will see this trace listed in the list of traces along the left side. The Y expression field will also display **New Trace**. To create a real trace, simply replace the **New Trace** with the desired expression in the Y expression field. When you leave that field, the trace list will update itself with the new Y expression for your trace.

Traces have several parameters that control how they are displayed. Here are a few of the basic ones. The more advanced features are discussed a little later:

Legend: If you specify some text for a legend for a trace, a legend will be displayed with your graph. The location of the legend is specified on the **General** sheet.

Trace Color: You can make a trace any color, or even colorize it based on an expression. To change the color, simply click on the current color of the trace and the MutliColor Picker dialog will be displayed. We'll discuss how to set this up for displaying a multicolor trace a little later. To display the trace in a single color, simply double click on the current color displayed in the first row of the table and select the desired color from the color dialog that appears.

Line Type: Select the desired line type to be used to display the trace. This is only applicable when the axis is set to display a line.

Point Type: Select the desired point type to be used to display the trace. This is only applicable when the axis is set to display points.

8.3 Graph scaling, zooming, and panning

More than likely, when you create your graph, the scaling will not be correct for the expression you entered. Fortunately, DAQFactory provides quite a few ways of setting the scaling of your graph.

The scaling of an axis is initially determined by the settings in the properties box. If you open the properties box for the graph and select the **Axes** sheet, you will see a list of the thirteen possible axes along the left side. You can select any of the axes and the scaling parameters for that axis, along with the rest of the parameters for that axis, will be displayed in the rest of the box. Although all the axes parameters are available, only the axes that have traces assigned to them are displayed. Multiple axes are explained a little later.

For all but the bottom axis, there are two scaling parameters, **Scale From** and **Scale To**. These determine the minimum and maximum values for the axis. You may notice that these parameters are actually expressions, so you can enter in fixed values if you would like, or you can enter in a more complex expression such as `Max(channel)`. If the bottom axis is displaying time, you can also have DAQFactory automatically set your scaling parameters so that the most recent data point is always displayed, and the traces scroll across the graph. Instead of using the Scale From and To parameters, you will use the **Time Width** parameter. Check the box to the left of **Time Width** then enter the width of the bottom axis in seconds in the field to the right. This field, like the Scale From and To, is an expression, so you can use a fixed value or a complex expression.

Even though you can quickly access the scaling parameters by double clicking on the desired axis, changing the scaling by always going to the properties box can be somewhat cumbersome. Because of this, DAQFactory provides several features for zooming and autoscaling.

To autoscale a graph, simply right click on the graph, select **Autoscale** from the popup, then choose the desired axis to autoscale from the submenu. This will scale the axis to completely display the traces on the axis. With stacked axes, which axes get autoscaled depends on where you right click. You can also use **Autoscale From** which does the same thing as autoscale, except that it replaces the minimum axis scaling with the value from the **Autoscale From** property for the axis, available on the **Axes** sheet of graph property page. If the maximum axis scaling value is less than this value, then it is replaced instead.

DAQFactory provides several zooming options. To zoom in a graph, click and drag a box around the area you would like to zoom in to in your graph. Right click inside the zoom rectangle and select the desired zoom axes. You can also zoom out, which adjusts the desired axes proportional to the zoom rectangle size. Once you are zoomed in, you can restore the previous zoom by selecting **Zoom Previous** from the regular graph popup which appears when you right click on your graph anywhere but inside the zoom rectangle. You can also pan the graph by holding down the **Shift** key and click and dragging inside the graph.

The autoscaling, zooming, and panning options discussed so far will freeze the scaling of the graph on the axes in which they apply. In this case when we are referring to an axis we are referring to all the Y axes, or the X axis, not just a single Y axis. A frozen axis does not use the **Scale From**, **Scale To**, or **Time Width** parameters from the graph parameters box. Therefore changing these parameters will not have any affect on the graph scaling if the axis is frozen. Likewise, a frozen X axis will not automatically scroll. The easy way to tell if an axis is frozen is to look at the focal rectangle around the graph. A non-frozen, or thawed graph will display the rectangle in green. If the X axis is frozen, then the left and right sides of the rectangle will be drawn in purple. If the Y axes are frozen, then the top and bottom sides of the rectangle are drawn in purple. You can freeze an axis manually, or restore / thaw an axis by right clicking on the graph and selecting Freeze / Thaw then selecting the desired axes. Thawing a particular axis will cause the scaling to revert back to the settings in the properties box. Freezing the X axis on a time graph manually can often be handy when you want to stop the constant march of your data across the graph,

and is especially useful in this situation when using markers. If you like your new scaling and would like to set the scaling properties of an axis to what is displayed, you can open the axis sheet and press the **Copy from Zoom** button. This will make the scaling match the current max and min values of the axis. Note that this will overwrite any complex expressions you may have entered for these parameters.

DAQFactory provides one other zooming feature that does not affect the freeze / thaw status of the graph. This is **Zoom In Time** and **Zoom Out Time**. You can access these options from the same menu as the other zooming functions. These work just like Zoom In / Out Horiz, except that on a time graph, these functions do not freeze the graph, but instead change the **Time Width** parameter directly.

8.4 Adding more traces to the graph

You can place as many traces as you would like on a graph. Adding addition traces can be done using the **graphs properties** box. In the property box, simply select the **Trace** sheet, and press the **Add** button below the list of traces. A new trace with a Y expression of **New Trace** will be created. Replace the New Trace with your desired expression and set any other parameters.

Once you have multiple traces, you can edit the parameters of a particular trace by simply clicking on the trace's Y expression as listed in the trace list of the properties box. When you click on it, all the fields in the rest of the box will be reset to display the parameters for the selected trace. If you change the Y expression of the trace, it will be updated automatically in the trace list. Be careful not to create two traces with the exact same Y expression. If you do so, you may have a difficult time selecting both traces. If you must have two identical traces, we suggest enclosing one of them in parenthesis or something similar. This will allow the trace list to distinguish between the two traces without changing the result of the Expression.

8.5 Multiple Axes

Your traces can be places on as many as six left axes and six right axes. These axes can be stacked or overlapping or both. DAQFactory creates all the possible axes when the graph is created, but only displays the axes that have traces assigned to them. All the parameters for the axes are available from the **Axes** sheet of the graph's properties box. To the left side of the box is the list of axes. Click on the desired axis you would like to edit the parameters for, and the fields to the right will be filled in with these parameters.

New Name: If you would like to rename your axis, simply replace the default name with your new name. The axis list and any traces assigned to this axis will automatically be adjusted.

Axis Label: This is the label that is displayed along the axis. Depending on the size of your graph, you may find it easier to read axis labels if you put a space between each character.

Plotting Method: Each axis can have one of a variety of plotting methods. All traces assigned to an axis will use the same plotting method. Some plotting methods are only available on left axes. The various methods are pretty self-explanatory, so we invite you to simply try them. Some methods may not work well with all data sets. Again, experiment.

Right Axis: This is a read only field. It indicates whether the axis will be displayed on the left or right side of the graph. This may seem extremely obvious considering the default names of the axes, but if you rename all your axes, things may get less obvious.

Axis Color: This determines the color of the axis and scaling labels. Click on the current color to open the color selection dialog box and select a new color.

Style: You can display any axis in either linear or logarithmic scaling. The bottom axis can also be displayed in Date + Time mode, which will automatically display the time values in a more readable form instead of seconds since 1970. Note that if you try and set an axis into logarithmic scaling and the axis contains traces with illegal log values (i.e. <= 0) then the scaling will revert back to linear.

Row: To create a stacked axis, simply assign a different row to each axis. If you would like to have stacked and overlapped axes, assign the same row number to the overlapping axes.

% of Full Axis: Stacked axes do not have to have an even share of the vertical space of the graph. If you leave this field at 100 for all the axes, then this occur. You can change this field to an appropriate percentage to

make one stacked axis use more or less of the total space. If you change this field, all the values for this field for the axes in use should add up to 100.

Align. Thresh: For X vs Y graphs, you may often be plotting data taken at different time scales. DAQFactory will perform an automatic alignment of your data in time just like the Align function. This parameter is passed to the Align function and determines how close two data points have to be in seconds to be considered to have the same time. If you want to disable this feature, set the alignment threshold to a negative number.

Auto Grid Control: If checked, DAQFactory will automatically set the grid line and tick spacing for you. If not, you will need to specify spacing values in the next two fields.

Major Tick Spacing: The spacing of the major grid lines and ticks in the units of the axis.

Minor Tick Spacing: The spacing of the minor ticks in the units of the axis.

Full Length Minor Ticks? If checked, grid lines are displayed for minor ticks as well as major ticks.

Area Graphs Negative from X Axis: Applies only if your **Plotting Method** is set to **Area**. If this is checked, then the area graph will shade to 0. If not, then the graph will shade to the bottom of the graph.

8.6 Colorizing Traces

In addition to assigning a single color to a trace, you can colorize a trace based on the result of an expression. This is a very effective way to display three dimensional data on a two dimensional graph. To do this, simply click on the **Trace Color** field for the desired trace in the graph's property box. This will open up the **MultiColor Picker** box. The table in this box works exactly like the colorization of the other component types. Simply enter threshold and color values in the table as needed. Unlike these other components, though, you can colorize the trace based on an Expression that is independent of the trace expressions. Simply type in the expression to use in the **Color based on:** field. This expression will automatically be aligned to the trace if the time scaling is different.

Note: Some plotting methods do not support colorization of the trace based on an expression and can only take single colors. These methods will plot in black if colorization is attempted.

8.7 Error Bars

Adding error bars to a trace in a graph is easy. On the **Traces** sheet of the graphs property box in the bottom right is a section dedicated to error bars. Select the trace you would like to add error bars to from the **Traces** list, then fill in the necessary error bar parameters.

Y Positive: An Expression that, when evaluated, determines the length of the positive going error bar in the Y axis. The result should be in the units of axis itself. You can use a constant, or a complex expression. The result should be positive, or the error bar will be upside down.

Y Negative: Identical to Y positive, but applies to the negative going error bar in the Y axis. The result should be positive, even though the error bar is going in the negative direction.

Y End Width: The width in pixels of the cap at the end of the Y error bars.

X Positive: Identical to Y positive but in the X axis.

X Negative: Identical to Y negative but in the X axis.

X End Width: Identical to Y end width but for X error bars.

XY Box: All error bar expressions must be specified and valid for this feature to work. If checked, a box that would outline the error bars with its center on the data point is displayed instead of the error bars.

End Thickness: The thickness in pixels of the end cap of all the error bars.

Line Thickness: The thickness in pixels of the error bar lines (not including the end caps).

8.8 Wind barbs

Graphs can display windbarbs on any trace, similar to errorbars. These are standard northern hemisphere (flag on the clockwise side) windbarbs. Speed can be displayed using either flags, length or both. The flags are also colored based on NOAA color standards. To create flags, enter expressions for Direction, Speed and Length in the new boxes on the trace property sheet of the 2D graph property box. Any of these expressions can be scalar if you want fixed values. For example, to display flags of fixed length, enter a scalar value for the length expression (in pixels). Speed is in whatever units you want. The flags display based on unitless values, short flag for 5, long flag for 10, 20, etc, and triangle flag for 50. The flag used is determined by the speed rounded to the nearest 5. To hide all flags, enter 0 for the speed expression. Color is based on actual speed value. Direction is in degrees. Flags are displayed right on the trace. To display them elsewhere, make an additional trace to tag them to.

8.9 Axis and Line annotations

DAQFactory currently provides three different ways of marking special events and values on your graph: axis annotations, line annotations, and bounds. Axis annotations appear as flags along the X axis and mark special events. They are typically used on time graphs. Line annotations can be set up to mark special events as well, but can also mark special values. Line annotations can also be used on any axis. Bounds are shaded areas of the graph to mark upper and lower limits in your data. For example, you could use a lower bound to display your noise floor. Bounds only work when you have a single left axis or right axis, or an unstacked left-right axis combination.

Axis Annotations:

Axis annotations appear as flags along the X axis to mark special events. They are designed for graphs displaying time along the X axis. The flags can have labels on them describing their purpose, and can be displayed in different colors. To create an axis annotation, open the **Axis+Bound Annotations** sheet of the graphs property box. The bulk of this sheet is taken up by the axis annotation table. Just like any other table in DAQFactory, use the **Add** and **Delete** buttons to add new rows and remove old ones. You can have as many different annotations as you need. Here is a description of the columns of the table:

Expression: The Expression for an axis annotation is a little unusual. Anywhere the result of the expression is non-zero, the time of that point is flagged. For example, if your data looked like this {0,1,4,3,4,2} and you entered the Expression `data = 4`, then a flag will be placed at the time of the 3rd and 5th data points (where the data equals 4).

Text: The text that will be displayed with the flag. If either the expression or the text column are left empty, the row will be deleted. If you don't want to display any text, enter a single space in the text column.

Color: The color of the flag and the text.

Line Annotations:

Line annotations can be used to mark any value(s) on any axis. You can also use these to mark special events as well, but the expression format is a bit different. To create line annotations, open the **Line Annotations** sheet of the graphs property box. This contains the line annotations table. You can have as many different annotations as needed.

Expression: The result of the Expression is used as the value along the desired axis for displaying the line. If the result is an array of values, a line will be drawn at each of the values. If you want to mark special events along a time X axis similar to axis annotations, you will need to add a GetTime function to your expression (for example `GetTime(data = 4)`).

Text: The text that will be displayed along the line. If either the Expression or the text column are left empty, the row will be deleted. If you don't want to display any text, enter a single space in the text column.

Axis: Select the desired axis that this line will be displayed along. The axis must have at least one trace assigned to it. Line annotations assigned to the bottom axis are vertical lines, all others are horizontal.

Type: Select the type of line to display.

Color: Select the color to display the line and the text.

Bound Annotations:

Bound annotations display a shaded area along the top and / or bottom of the graph to denote areas such as noise floors or total dynamic range. These annotations can only be used if you have a single left or right axis, or you have a non-stacked left-right axis pair. To create a bound annotation, open the **Axis+Bound Annotations** sheet of the graphs property box. At the bottom, below the axis annotation table are the fields for creating bound annotations. Simply enter the desired text for whichever bounds you wish to display and the value to use. The lower bound will shade from the value down, and the upper bound will shade from the value up. To remove a bound annotation, simply erase the associated text. To display a bound annotation without any text, enter a single space in the text field.

8.10 Other graph properties

DAQFactory provides a whole slew of properties for controlling the display of your graph. These are available from the **General** and **Colors+Fonts** sheets of the graph's properties box. Many of these are pretty self explanatory, but nevertheless, here they all are:

General Sheet:

Main Title: The title to be displayed along the top of the graph.

Subtitle: A smaller title to be display along the top of the graph under the main title (if any).

Legend Location: Where the legend should be displayed, if there is any to be displayed. A legend will only be displayed if one of the traces has the **Legend** field specified.

Shadows: For a more fancy graph you can add shadows to your data points. This will slow down the generation of the graph a bit, so you may want to save this feature for special occasions.

Point Size: The overall size of the points used with traces drawn with points.

Graph Data Labels: If checked, then the X / Y coordinates of all the data points will be drawn next to the data point. We suggest only using this if you have less than 50 data points or so. If you have more, you probably won't be able to read the data labels, and the graph will take a very long time to draw.

Line Gap Threshold: This specifies the distance two points need to be separated (in X) for a gap to be drawn. Any less and a line will be drawn connecting the points. This obviously does not apply to traces drawn with Points only.

Display Gaps?: Determines whether gaps should be drawn between points separated by NaN's.

Max Data Precision: The maximum precision used to display the axis scaling labels.

Grid Axes: Which axes grid lines should be displayed.

Grid Linetype: The type of line that should be used to display the grid lines.

Grid in Front? Whether to draw the grid in front of the traces or behind them.

No 1, 2, 5: Typically, DAQFactory will use an algorithm to display axis labels and grid lines at nice intervals. This field allows you to turn off this feature.

Log Scale E Notations: Determines whether standard E notation should be used when labelling a log axis.

Long Log Tick Threshold: Determines the point at which long log ticks should be displayed.

Colors + Fonts:

Most of these parameters are quite self explanatory. Here are the few that are a bit less obvious. In general you will probably just want to play with these to get the desired results, since their effect on the graph depends on the size of the graph and the text that is being displayed.

Overall Font Size: Since the graphs can be resized at will, font sizes are not absolute, but relative to the overall size of the graph. This parameter determines an overall scaling factor for all the fonts used to display the graph.

Overall Font Scaling: A more precise way to adjust the font scaling.

Title Font Scaling: A separate scaling parameter for the title and subtitle.

Legend Font Scaling: A separate scaling parameter for the legend.

Axis Annotation Size: A separate scaling parameter for axis annotations. Unlike the other scalings, which are relative to 1 and a change of 0.1 will result in a large change, this parameter is more coarse. A change of 10 will only be marginally noticeable. The default value of 50 will work for most cases.

8.11 X vs Y graphs

In DAQFactory you are not limited to value vs. time trend graphs. You can plot just about anything vs. anything else. Doing so is as simple as providing both a Y and X expression for the trace. Either can be a simple channel name, or a complex calculation. When both expressions evaluate to a value with time associated with it, DAQFactory will automatically align the Y data to the X data's time. In this way you can easily generate XY graphs on data taken at different time scales. If you wish to turn off alignment, set the Align Threshold to a negative number. The Align Threshold only applies to the bottom axis.

Here are some things to think about when creating XY graphs

• If your data is randomly distributed in X then you will probably want to use the point plotting method, otherwise you will get a spaghetti of lines.

• If you provide a blank X expression for a trace, then the point number of the Y expression will be used for the X values.

• Even though there is only one X axis, each trace in your graph can have a completely different X expression. This is true on time graphs too.

• Because the Traces property tab uses the Y Expression to select which trace to edit, you must make sure each Y Expression is different. If you need to have the same Y Expression for multiple traces, presumably with different X Expressions, add something to make the expression different without changing the result. This can easily be done by adding an additional parenthesis pair, or doing +0, or something similar.

• Axis annotations only work if the X axis is displaying time, since they use the time of the data points to determine the flag location (though there are some ways around this using the InsertTime function).

• Markers don't work terribly well in XY graphs when the X data is randomly distributed. Sometimes when you move the markers with the keyboard, the markers can jump around unexpectedly. Since the graphs may be displaying constantly changing data, it is sometimes hard for the markers to get a lock on a data point.

• When using markers for ranging functions, especially Make NaN, remember that the range is determined by the X value of the two markers, not the data point number, but when making each point a NaN, it scans by data point number. Make NaN may work a little weird because of this, since it will start at the first point and scan through the data by point number until it finds a data point out of range.

8.12 Advanced graphing

DAQFactory provides some handy features from the graph popup that haven't been discussed yet. To get to the graph popup, simply right click somewhere on the graph (other than inside a zoom rectangle).

Markers:

There exist 4 different markers for each graph within DAQFactory. The two standard markers are used for determining data point values and delta values between the two markers and can be used for ranging functions such as Make NaN and most analysis tools. The two peak markers enable you to quickly do a trapezoidal integration of a peak and also use a center-of-mass algorithm to take a quick guess at the peak location.

To add any of the markers to a graph, right click close to the data point you would like to place the marker on, select the desired marker from the popup menu and select **Add**. from its submenu. To remove a marker, simply right click anywhere on the graph and select **Clear** from the marker's submenu. Once a marker is placed, data about the markers position will be displayed along the bottom of the graph. Peak markers only display their data once both markers are placed.

You can move a marker once its placed using the keyboard. Make sure the graph with the markers is in focus. The left and right arrow keys will move Marker A, up and down arrow keys will move Marker B. Hold down the Ctrl key while pressing the arrow keys to move the peak markers.

You can change which trace the marker is on using the + and - keys on the keypad. This will affect Marker A. To change marker B, hold down Shift with + and -. For the peak markers, the same applies, but you should hold Ctrl as well.

Make NaN:

NaN stands for Not a Number and designates a data point with invalid data. The **Make NaN** function makes it easy for you to mark data as invalid while viewing that data on a graph. You can mark single data points or entire ranges, though for the range form to work properly, your X data must be sequential. X data is always sequential with time graphs. Make NaN even works on live data, effectively changing the history. Make NaN will not, however, affect the data being stored to disk by DAQFactory normally, but could affect local conditional logging.

To make a single point NaN, simply put the Marker A on the desired data point and select **Make NaN - Marker A** from the graph popup. To make a range of points NaN, mark the first and last points with Marker A and B and select **Make NaN - Range** from the graph popup. The points the markers are on will be converted to NaN as well.

If the trace you wish to apply the Make NaN contains a Y expression that references more than one channel, a dialog box will appear where you can select which channel you wish to apply the Make NaN to.

Analysis:

The graph popup offers direct access to the analysis boxes for analyzing traces. If a marker is on a trace, or if there is only one trace in the graph, the trace's expressions will automatically be inserted into the analysis parameters. If both Marker A and B are specified, then their locations can be used to specify the range of data to analyze. The data analysis tools are described in a later chapter.

Capture:

The capture function will create virtual channels with the data from the traces. If you choose to, this function can also make a new graph that references the new virtual channels. The new graph will essentially look like the old graph, but the data will be a static copy of the original data. Note that channels used for error bars, scaling parameters, or annotations are not captured.

To capture a graph, simply select **Capture** from the graph popup. The capture box will appear with a list of all the traces in a table. Select the traces you would like to capture using the second column of the capture table. A default name is specified in the third column of the table. You are welcome to use this name, or you can simply change the name. At the top of the capture dialog box is the **Create Graph?** option. If checked, a copy of your original graph with the new virtual channels in it will be created on the scratch page.

Note that the result of the expressions used to create the traces is captured, not the individual channels used in the expressions.

Export / Print:

Use this function to either export your graph to a Windows MetaFile, BMP file, or export the data that makes up the graph in ASCII text form. You can also use this function to print the graph. The options for these functions are fairly self explanatory.

Note: Error bars and markers will not be printed or exported. We suggest using a screen capture utility if you need to print or export these, or just print the entire Page. We are hoping in a future version to incorporate error bars into the exporting function.

8.13 Graph variables and functions

Please see the section on Component Names for a description on how to use these.

Like all the other components, graphs have variables and functions that allow you to adjust the graph's properties without using the properties window, usually from a sequence or component action. Because graphs are a more complicated component, there are many more variables and functions available:

General:

- PrintGraph(): Displays the print/export graph window to allow you to print or export the graph.
- SaveJPEG(filename, x size, y size): saves the graph as a JPEG of the given size and to the given file. The user is not prompted.
- SaveToFile(filename): saves all the data necessary to reproduce the graph to a file. Use with LoadFromFile():
- LoadFromFile(filename): loads all the parameters and data stored from a previous call to SaveToFile() into the graph.
- pauseDataUpdate: a variable that defaults to 0. If 0, the graph draws as normal. If 1, the graph does not recalculate the traces, but draws with whatever has already been calculated. If 2, the graph recalculates the trace data once, then sets pauseDataUpdate to 1. This variable is used for graphs that are used to display non-realtime data that doesn't need to be updated with every screen refresh. One can set the graph scaling, then set this variable to 2 and the graph will calculate once and draw, but then won't waste CPU time redrawing with each refresh.

Note that Load/Save do not save the parameters of the graph as they are saved in the document, but rather save a snapshot of the data and markers in a binary format to recreate the graph exactly. You really should only use LoadFromFile() on an empty graph component, or one that has had LoadFromFile() called on it previously. Do not use it on a graph component that has any traces specified, otherwise those traces will replace the data from LoadFromFile() on the next screen refresh.

Traces:

- AddTrace(tracename): adds a blank trace to the graph with the given name. Use the trace name to set the new trace's properties.
- DeleteTrace(tracename): removes the given trace from the graph
- DeleteAllTraces(): removes all the traces from the graph

All the trace properties require providing a unique name to your traces. The graph properties window has a name property for this purpose, or use the name you provided when using the AddTrace() function. The format would be: `Component.graphname.tracename.property`

- AddColor(threshold, color): adds a color to the color table with the given threshold. Usually used with ClearColors()
- ClearColors(): removes all colors from the color table for the trace. Usually used before using AddColor() to build a fresh table
- Color: setting this variable will reset the color table to a single value
- strLabel
- LineType: 0 = thin solid, 1 = dash, 2 = dot, 3 = dash-dot, 4 = dash-dot-dot, 5 = medium solid, 6 = thick solid, 9 = medium thin solid, 10 = medium thick solid, 11 = extra thick solid
- PointType: 0 = plus, 1 = cross, 2 = dot, 3 = dot solid, 4 = square, 5 = square solid, 6 = diamond, 7 = diamond solid, 8 = up triangle, 9 = up triangle solid, 10 = down triangle, 11 = down triangle solid
- strYAxis: the name of the Y axis as displayed in the properties window. This is not the same as the name used to programmatically access the axis as described below. For example, by default, traces are put on the YAxis "Left Axis 1".
- strXExpression
- strYExpression
- Visible: 0 = don't draw the trace, 1 (default) = draw the trace. This parameter can, for example, be connected to a checkbox to allow traces to be added / removed from the display by the end user.

ErrorBars:

- EndThickness
- LineThickness
- XEndWidth
- strXMinus
- strXPlus
- XYBox
- YEndWidth
- strYMinus

- strYPlus

Windbarbs:
- strWindBarbDir
- strWindBarbLength
- strWindBarbSpeed

Axes:

There are 13 axes in every graph. To programmatically access their properties you have to reference their internal name which is fixed. This is different from their displayed name in the graph properties which is variable. The fixed names are LeftAxis1 through LeftAxis6, RightAxis1 through RightAxis6 and BottomAxis. To change an axis property, use the following notation: `Component.graphname.axisname.property`l
- Alignment
- AutoGrid
- strAutoScaleFrom
- AxisColor
- AxisTick
- GridDensity
- Inverted: this property is not available from the properties window. It inverts the scaling too, so it may take some fiddling with Scale From and Scale To to get the graph to display the way you want.
- strLabel
- LongTick
- NegativeFromX
- PlottingMethod: 0 = line, 1 = bar, 2 = stick, 3 = point, 4 = area, 8 = point + best fit line, 10 = histogram, 12 = bubble, 13 = point + best fit curve, 15 = point + best fit spline, 16 = spline, 17 = point + line, 21 = step, 22 = ribbon. Many of these are not available from the properties box. Other valid values exist up to 29, but may cause unpredictable results.
- PlottingStyle: 0 = linear, 1 = log, 2 = date/time
- Proportion: for stacked axes
- Row: for stacked axes
- strScaleFrom
- strScaleTo

The bottom axis also has:
- strTimeWidth
- UseTimeWidth

General Settings:
- BarWidth: for bar graphs
- DataPrecision
- DataShadows
- GraphDataLabels
- LegendLocation: 0 = top, 1 = bottom, 2 = left, 3 = right
- LineGapThreshold
- LogScaleExpLabels
- LogTickThreshold
- strMainTitle
- NULLGaps
- PointSize
- strSubTitle

Grids:
- GridInFront
- GridLineControl
- GridStyle: 0 = thin solid line, 1 = thick solid line, 2 = dotted line, 3 = dashed line
- NoGridLineMultiples

General Colors:
- DeskColor
- GraphBackColor
- GraphForeColor
- ShadowColor
- TextColor

Font Sizes:
- AxesAnnotSize
- FontSize
- FontSizeGlobal
- FontSizeTitle

Bounds:
- strLowerBoundText
- LowerBoundValue
- strUpperBoundText
- UpperBoundValue

Markers and user interaction:
- BoxTracking: true when the zoom box is being dragged
- BoxX1, BoxX2, BoxY1, BoxY2: if a scale box is being displayed, these variables contain the coordinates of that box.
- BoxValid
- MarkerA_Valid
- MarkerA_X
- MarkerA_Y
- MarkerB_Valid
- MarkerB_X
- MarkerB_Y
- PeakArea: this is read only and is the calculated peak area as determined by the peak markers
- PeakLocation: this is read only and is the calculated value of the peak location as determined by the peak markers
- PeakMarkerA_Valid
- PeakMarkerA_X
- PeakMarkerA_Y
- PeakMarkerB_Valid
- PeakMarkerB_X
- PeakMarkerB_Y
- XAxisFrozen
- YAxisFrozen

9 Data Logging and Exporting

9 Data Logging and Exporting

9.1 Data Logging Overview

Data acquired in DAQFactory can be logged using one of two methods:

A logging set: will log acquired data continuously as it is acquired. Logging sets only logged acquired data and cannot log variables or virtual channels unless performed manually. Logging sets run continuously until stopped, adding more data to the file as more data is acquired.

An export set: will write the results of multiple expressions to disk. Because each column in the data set is the result of an expression, anything can be exported. Export sets typically are run occasionally to create complex data sets, and usually pull their data from the history. Export sets write their data and then stop. They do not monitor for new data, but simply use the results of the expressions once to write the file.

Typically, logging sets will be used to log your raw data, and then if desired, export sets can be created to log calculated data or regions of interest in your data.

9.2 Creating a logging set

You can easily manage your logging sets through the logging set summary view. To get to this view, simply click on **LOGGING:** in the workspace. This view will list all of the current logging sets and their status. To add a new logging set, simply click **Add**. You will be prompted for a name for your logging set. Enter a name and click **OK**. You will then be switched to the logging set view for your new logging set. Once a logging set is created you can get to the logging set view for it by clicking on its name under **LOGGING:** in the workspace.

Once created, you can start or stop a logging set either by clicking on the **Begin** and **End** buttons in the logging set view, or by right clicking on the logging set name and selecting **Begin Logging Set** or **End Logging Set**. Logging sets can also be started and stopped using components or in sequences.

Note: when taking and logging data faster than 20hz you'll need to set the Alignment Threshold parameter of your logging set to 0 to avoid missing data points in your logging set.

Logging Set Properties:

Auto-Start: If checked, the logging set is started as soon as the document is loaded, unless it is loaded in safe mode. If checked, the logging set will also be started when switching out of safe mode.

Main:

Logging Method: Determines how the data is logged to disk. The choices are:

ASCII: This is the default format that logs to a delimited ASCII files. Delimited ASCII files can be read by most every windows data analysis package such as Excel and so is a very flexible format. It is also easily opened and edited in other software packages, even notepad. Because of this, the file format can easily be repaired if some sort of damage was to occur to part of the file. This is the recommended format for logging your data, but has two disadvantages: 1) it consumes more disk space than the binary modes, and 2) it is reasonable slow. On our systems we can only log about 10,000 points per second in ASCII mode. Depending on your computer you may see better or worse performance. ASCII is also slow to load. If you have large data sets and are using a program that can load binary data, we suggest binary data format instead.

ODBC Database: This format will log to an ODBC compliant database. How to set up an ODBC database is described in a separate section. ODBC is convenient if you ultimately want your data in a database, but it is very slow, especially when logging directly to Microsoft formats such as Excel and Access. We see 20 rows per second maximum update rates for these formats. You may see better performance, especially if you are writing to a native SQL database.

Binary: There are three different binary modes. Each uses a different binary representation for the data. If you are going to use your data in an application that can read a binary format we suggest this method over the other methods as it is very fast both to log and to load into your analysis software, and it is the most compact. The three

different modes allow further fine tuning of the data set for most compactness and are discussed in a separate section.

File Name: The path and filename you would like to log data to. Click on the **Browse** button if you want to use the standard Window's file selection window to select a file name. If you specified auto split files, a number or date stamp will be appended to your name (in front of the file extension).

You can optionally use date/time format specifiers in your file names. This allows you to create new files every hour, day, week, etc. and include the time stamp in the file. To see what format specifiers are available, please see the section on the FormatDateTime() function, in 4.12.11. For example, if you wanted to create a new date stamped file every day, you might put something like: c:\myfolder\myFile%y%m%d

Data Source: This displays in place of file name when you select ODBC data base. This should be the name of the ODBC data source as defined in the ODBC manager. Note: This is NOT the name of the database! This has been a very common mistake despite our documentation so here it is again: THIS IS NOT THE NAME OF THE DATABASE SO IT SHOULD NOT BE A PATH TO A FILE! It should be the name you assigned in the ODBC manager. See the section on setting up an ODBC connection for more details.

SQL: This becomes availble when you select ODBC database. Clicking this displays a window that allows you to adjust the SQL commands used. This is sometimes necessary as different database engines use different dialects of SQL.

Table Name: This becomes available when you select ODBC database. This is the name of the table within the database where the data will be written.

Channels Available / Channels To Log: These two tables determine what data will be logged. The channels available table lists all the channels in your channel table. Since logging sets do not support direct logging of variables or virtual channels, these will not appear in this list. Select the channels you would like to log either individually, or by holding the Shift or Ctrl key and selecting multiple channels, then move them to the channels to log table using the **>>** button. The All, None, Up, Down, and << buttons will help you get the exact list you want and in the proper order. In the channels to log table, there is an additional column called **Figs:**. This determines the number of significant figures that will be used when logging the data. This has no effect on binary formats, and may be limited in ODBC formats depending on your database.

Manual: The manual button allows you to add additional columns to your logging set. This requires you to manually add data to the logging set through a sequence. Using this method you can log any sort of data in a logging set. Please see the section on logging set functions for more info. Two built in manual columns that can be added without writing a sequence is "Note" and "Alert" which will log any quick notes or alerts in with your data stream. Since notes and alerts are both strings, you cannot log these to a logging set using the binary logging method.

Details:

Mode: There are two different modes of logging. These are independent of the logging method and determine how data is collected and aligned to rows.

All Data Points (aligned): This mode will log all data that comes in. The data is aligned to place data with the same or similar times on the same row. The spacing of the rows may be variable depending on when your data is acquired.

Align Threshold: How close in time (in seconds) two data points have to be to be placed in the same row.

Align Mismatch: If a particular column does not align with the current row's time, DAQFactory can either leave the value blank (or 0 for binary modes), or copy the last value written.

Application: The All Data Points mode is useful when you want to make sure that every data point you acquire on the given channels is actually logged and no data massaging occurs before logging.

Fixed Interval: This mode writes a row of data at a constant interval. The data written is either the most recent value for the particular column, or the average of the values acquired since the last row was written.

Interval: Determines how often in seconds a row of data is written.

Type: Determines what data is written. If average is selected, then the average of the data points that have arrived since the last row was written is used. If snapshot, then the most recent value of each column is used.

Application: The fixed interval mode is useful when you are acquiring data at different rates but you want the final logged data to be evenly spaced in time.

Time Format: Internally, DAQFactory records time as seconds since 1970. This yields the best precision possible, and is the internal Windows standard, but is not the format used by Windows' Office products like Excel, nor is it easily read by humans. Windows' Office products use decimal days since 1900. As such, we give you a choice. You can log using DAQFactory time, which is best if you are going to import the data into a data analysis package, or if you are doing high speed data. You can use Excel Time if you are going to take the data into Excel and are doing high speed data. Or, if you want the time column to be in what we like to call "human readable" format, you can select Custom, and specify your own time format specifiers in the following property.

Time Sig Figs: Determines how many significant figures are used to write time values. A value of 9 yields time precise to the second, 12 to the millisecond and 15 to the microsecond. The maximum value is 15. Does not apply to the Custom time format.

Custom Time Formatting: If Custom time format is selected, this parameter takes a date / time format specifier to determine the format of the output. This follows the FormatDateTime() function exactly. To see what format specifiers are available, please see the section on the FormatDateTime() function, in 4.12.11. Typically you can use %c which just uses whatever Windows thinks is the appropriate format (i.e. MM/DD/YY HH:MM:SS in the United States).

Log Unconverted Data: If checked, the data logged will not have any conversions applied to it.

Application: This is a preference thing. It is certainly more convenient to log converted data. It saves you from having to convert the data after the fact. The problem is that many conversions are irreversible especially if you forgot what conversion you applied. By saving unconverted data, you always have the data in the rawest form and therefore should know what units it is in.

Continue on Error: If checked and a file error occurs, the logging set will continue to retry logging data to that file. If not checked, then the logging set will stop if a file error occurs.

Delimited By: Determines the delimiter used to separate values in the ASCII log mode. Enter "Tab" to use a tab delimiter.

Include Headers: If checked then a single row of channel names properly delimited is written at the top of each file. This only applies for ASCII mode. For ODBC, the channel names are used for the field names.

Header File: If checked, a separate file, with the same file name as the logging file, but with a .head appended is written describing the logging set. This is especially useful for binary logging methods which are completely useless if you forget what column is what or even how many columns you have.

Auto Split Files: If checked, the logging files will be closed at a preset interval and a new file opened. A number is appended to each file name to properly organize all the files. Auto splitting files is most useful for long term acquisition to avoid large files and prevent data loss in the case of file corruption.

Data File Size: Used if auto split files is checked. Determines the number of rows written per file.

Use Date in File Name: Used if auto split files is checked. If this is checked, the date and time of file creation is used in determining the file number instead of simply numbering the files.

Application: The auto split files option is designed for users who will be taking data continuously for long periods of time. By enabling this option, DAQFactory will create multiple sequential data files. This will keep you from ending up with a giant 20 gig data file. Auto split is also designed to help alleviate possible data loss from corrupted files. Very often, when a file gets corrupted, all the data in that file is lost. If you split your data into separate files, file corruption may be limited to a small chunk of your data.

9.3 Logging Notes and Alerts

To log alerts or quick notes in a data set you have to create a manual log column. For example, to log your quick notes in a logging set:

1. Open the logging set view for the desired logging set.

2. On the main page, click on the **Manual** button.

3. Enter **`Note`** and hit **OK**. This will add note to the channels to log table.

Now notes will be logged with the file. Alerts can also be logged by specifying **`Alert`** instead of **`Note`**. Both of these are strings so can not be used in binary mode logging.

9.4 ODBC Database logging

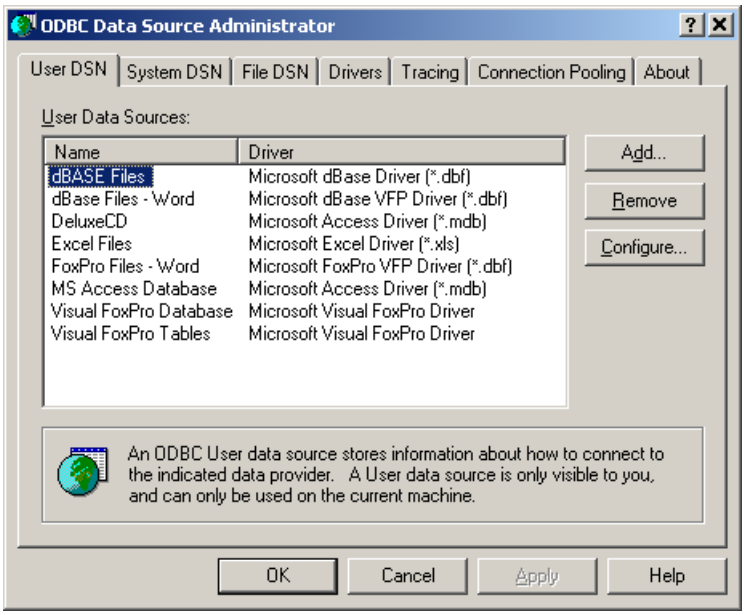
DAQFactory supports logging of data directly into an ODBC compliant database. In addition to enterprise type databases, this can be used to log data directly into Access, Excel, FoxPro and other local databases. To use this method, you must create a blank database and a data source. The blank database can be exactly that, blank. There is no need to create any tables (or worksheets in Excel). The database must simply exist. DAQFactory will take care of adding tables to the database as needed. A data source is used by ODBC to tell DAQFactory what database to use and what database engine to use with it. The database engine essentially acts like a translator, allowing DAQFactory to communicate with any database you have an engine for. Windows typically installs with a database engine for Access, Excel, dBase, FoxPro, Paradox, Text files, and a few others.

Creating a Data Source

We'll assume you know how to create a blank database (if necessary), and quickly give you instruction on creating a data source. This is all done using Microsoft tools external to DAQFactory and the exact method may be different for your version of Windows.

1. Open up the **Microsoft ODBC Administrator**.

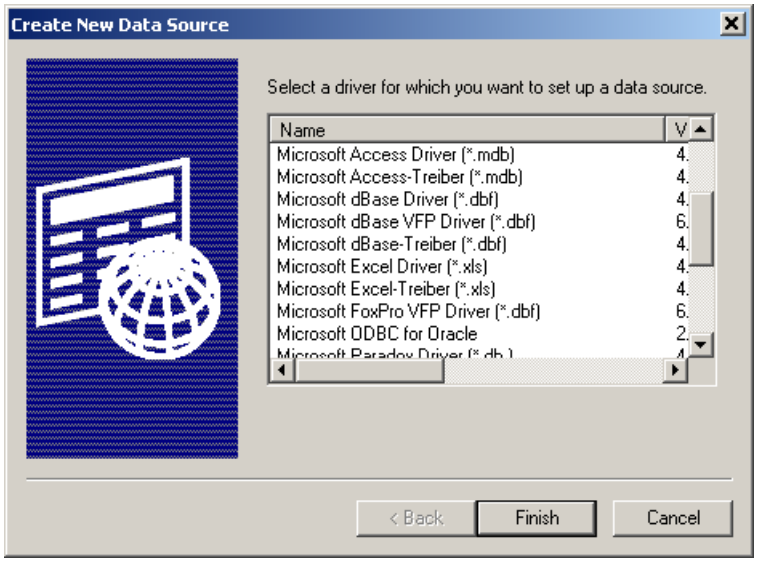
   This will either be in your **Administrative Tools** folder in your **Start Menu**, in your **Control Panel**, or in your **System** or **System32** directory (file name: ODBCAD32.exe). It may also be called **Data Sources (ODBC)**.



2. Once loaded, click on the **System DSN** tab.

   This looks just like the **User DSN** and **File DSN** tab. System DSN's are available to all users on your system. User DSN's are only available to the user that created them. Creating the Data Source is identical for both.

3. Click on the **Add** button. You will then be prompted for a driver. This is the desired engine driver. Click on **Finish** when you find the one you need.

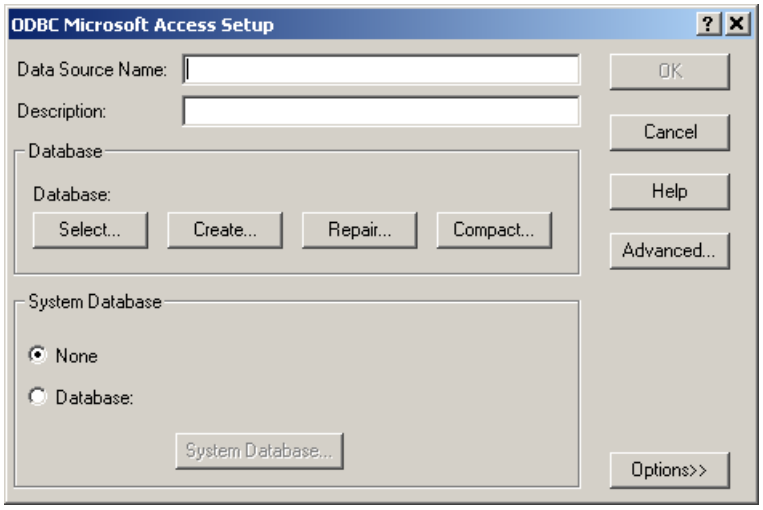


4. The next screen will differ slightly depending on the engine driver you selected. The first item is the **Data Source Name**. This is how you will reference the database within DAQFactory. Use some unique name.

   The rest of the items differ as mentioned, but typically there is a part that allows you to select the database file itself. There also may be some options that don't initially display.

5. Click on the **Options>>** button to expand them.

   For example, with the Excel engine, one of these options is **Read Only** which must not be checked otherwise DAQFactory won't be able to add any data!



6. When done, click **OK** and the data source is created. You can edit your data source by highlighting it in the list and clicking **Configure**. You are now ready to use ODBC within DAQFactory.

To log to the database, select **ODBC Database** as the logging method in the logging set. Under **Data Source** enter the name of the data source (DSN) you created in the ODBC manager. Next, enter the name of the table within the database you would like to place the data. The table does not need to exist yet, and in general should not exist unless it was created by DAQFactory earlier.

Important Note: A very common mistake is to put the path name to your database (c:\databases\mydatabase.mdb for example) in the box for Data Source. This is incorrect and will not work. DAQFactory does not need or want to know the name of your database file. All it wants is the name of the data source created in the ODBC administrator.

9.5 Reading from an ODBC Database

DAQFactory also supports querying from an ODBC database. To allow for the most flexibility, this is done entirely with scripting, similar to the direct file access functions. All functions begin with **DB.**:

Open(DSN,[username],[password]): opens the given data source using the given credentials (or none if not supplied). All parameters should be strings. Returns a handle which is used for functions described below.

OpenEx(ConnectString): same as above, but allows you to provide your own connection string. This gives more flexibility but requires the user to understand the format of connection strings, which is often database specific. For example, we have found that MSSQL often requires openex() because the normal Open() connection string includes "ODBC;". So you would probably want `db.OpenEx("DSN=mysdn;UID=myuser;PWD=mypass")`

Close(dshandle): closes the data source with the given handle. DAQFactory will clean up if you forget, but you can run into memory leaks if you repetitively do Open() on the same data source without closing, as each open creates a new internal object.

Execute(dshandle,SQL String): use this function to perform any SQL command except on that returns records like SELECT. So, this function can be used for adding records, updating records, and performing all other database tasks other than queries. This function returns the number of records affected.

QueryToClass(dshandle,SQL String,[class instance]): this function performs the given query on the dataset opened with Open() or OpenEx(). The SQL string should be a SELECT command. If a class instance is not provided, the function returns a custom object containing the complete query results. In the class member variables are created for each field name in the query results, plus two extra variables: **FieldNames**, which contains an array of strings listing all the fields of the query, and **RecordCount**, which contains the number of records returned from the query, which could be 0. So, for example, if you did:

```
global myset = QueryToClass(dshandle, "SELECT Name, Address FROM myTable")
```

then `myset.FieldNames` would equal `{"Name","Address"}`, `myset.Name` would contain an array containing the all the rows for the field Name, etc.

For more advanced users using classes, you can provide an existing class instance (object) instead of having QueryToClass() create one for you. This allows you to use classes with member functions that presumably could work on the fields queried. So:

```
myObject = new(MyClassType)
QueryToClass(dshandle, "SELECT Name, Address FROM myTable", myObject)
```

will create the same member variables (if necessary) as the other version of QueryToClass(), but will put them into myObject. Note that we provided the myObject instance and not the MyClassType class definition. The object must be instantiated already.

RecordSet Based Querying:

Query(dshandle,SQL String): performs the given SQL command on the given data source. No semi-colon is required at the end of the SQL string. SELECT statements work best and is what we have tested, though you may be able to use other SQL statements as well. You will have to experiment. This function returns a recordset handle used for the functions below. In most cases you should consider using QueryToClass() instead.

CloseQuery(rshandle): closes the given recordset created by the query. Again, DAQFactory will clean up, but repetitive queries without close will use up your memory.

Field(rshandle,field): retrieves the given field value from the given record set handle. The "field" parameter can either be a string with the field name, or an index of the desired field within the record.

MoveFirst,MoveLast,MovePrevious,MoveNext,IsBOF,IsEOF(rshandle): these are standard record scanning functions. Use this to move among the records returned by the query and to determine if you are at the beginning or end of the table.

Note that there are two different handles, dshandle, which is the data source handle returned by open and rshandle which is a record set handle returned by a query.

Here is some sample code. It opens a table, pulls out 4 fields where Operator_ID = "kjljkn" and fills arrays with these values. Obviously your script will be different, but this should give you a good template:

```
global dbase
global qr
global count
global thetime
global batch
global start
global strTime
dbase = db.Open("daqtest")
qr = db.Query(dbase,"select * from logfile1 where Operator_ID = 'kjljkn'")
global counter
counter = 0
while (!db.IsEOF(qr))
  thetime[counter] = db.Field(qr,"TheTime")
  batch[counter] = db.Field(qr,"Batch_Number")
  start[counter] = db.Field(qr,"Start_Date")
  strTime[counter] = db.Field(qr,"Start_Time")
  counter++
  db.MoveNext(qr)
endwhile
db.CloseQuery(qr)
db.Close(dbase)
```

9.6 Binary logging

The binary logging methods are the most efficient logging methods in both time and space. Unfortunately, not all analysis tools will import this format so you may be forced to use ASCII instead. Also, the binary logging methods only log numeric data and therefore cannot be used to log strings.

There are three binary modes. Select the desired mode based on the amount of precision required in your data.

Binary 64 bit floating point: This method maintains the complete precision used internally by DAQFactory: 15 to 16 significant figures. Even time can be recorded to the microsecond in a single column under this format. However, most data does not require 15 significant figures, so you may want to consider the other options. The format is simply an array of 64 bit floating point values, one for each column.

Binary 32 bit floating point: This method uses 32 bit floating point values, which take half the space of their 64 bit cousins. However, they only achieve 6 to 7 significant figures. Because of this the time columns require two to three values depending on the requested time significant figures. If the desired time significant figures is greater then 13 (precision better than 1 millisecond), then three columns are required for each time column. The three columns are the high, middle and low sections of the time value. To get the actual time value you have to add the three values together after rounding off the least significant figure. Here's some sample code (in C):

```
double hi,med,lo;
// the next three lines convert the floating point values into double floating point::
hi = (double)Point0;
med = (double)Point1;
lo = (double)Point2;
// round the highest part of time to the nearest 100,000
hi = floor(hi / 100000 + 0.5) * 100000;
// round the middle part to the nearest integer
med = floor(med + 0.5);
// the lo part does not need to be rounded...
// add the results to get the actual time value
ActualTime =  hi+med+lo;
```

If the desired time significant figures is less than 14, then only two columns are required for each time column. Again, simply add the two values after rounding:

```
double hi,lo;
// the next two lines convert the floating point values into double floating point::
hi = (double)Point0;
lo = (double)Point1;
// round the highest part of time to the nearest 1,000
```

```
hi = floor(hi / 1000 + 0.5) * 1000;
// the lo part does not need to be rounded...
// add the results to get the actual time value
ActualTime =  hi+lo;
```

Unless you need more than 6 significant figures in your data, or you are only logging a couple of channels, you probably should use the 32 bit version over the 64 bit version.

Binary 32 bit unsigned integers: This method uses simple 32 bit unsigned integers for each column. 32 bit integers range from 0 to about 4 billion. This mode is designed to efficiently store counter data or any other integer type data. If time significant figures is greater than 10 (i.e. need precision better than 1 seconds), then two integer columns are required to store time. To recreate the actual time value from these two columns add the second column divided by 1 billion (1e9) to the first column.

9.7 Export Sets

Export sets are similar to logging except that export sets run once with the current history and then stop, while logging sets take acquired data as it is accumulated and logs it. One big advantage of export sets is that you specify expressions instead of simple channel names. In this way you can subset data, or export calculated data.

Creating and configuring an export set is almost the same as a logging set. There is only one table for specifying what gets exported. In this table, create a row for each field you would like to export. In each row, specify a name for the field (also used for the header in the ASCII logging method), an expression to log (i.e. `MyChannel`, or `MyChannel / OtherChannel`), and the number of significant figures to use. Note that the expression should ideally have a history as the length of the history determines the number of rows generated in your file. So, `MyChannel[0]` is probably not a good idea unless you just want one row.

The export set details are very similar to logging sets. The difference is that export sets do not have auto splitting and have an option for whether an existing file should be overwritten or appended to. Logging sets always append to existing files.

Application: One interesting way to use exports is to actually specify scalar values for the expressions (i.e. `MyChannel[0]`) and then use a sequence to repetitively start the export set. This will allow you to log any type of data, converted, calculated or otherwise. To do this:

1. Create a new export set. In this example we'll call it "WriteOne".

2. Add columns for each data point you would like to log. Each expression should result in a scalar value without history. Go to the details page and make sure its set to Append.

3. Create a new sequence:

```
while(1)
  beginexport(WriteOne)
  wait(1)
endwhile
```

4. Now when you run this new sequence, it will start your export set once a second, each time writing a single line of data.

Application: You can extend the above application to log lines of data at particular events. Just call beginexport() whenever you want to write a line of data.

Application: When doing batch runs where you start and stop logging with each new batch, there are two ways to log. One is to use logging sets of course, and simply start and stop the set at the beginning and end of the batch. But this does not give you the option to throw away the batch (and not log it). If you want to be able to run a batch and then optionally save to disk, create an export set for your data instead of a logging set. You'll want to clear the history of your channels at the beginning of the batch so you are not logging older data.

9.8 Logging / Export Set Functions

Logging and export sets have variables and functions that allow programmatic access. Most of the variables match directly with parameters in the logging set and export set views. All are accessed using either `Logging.LoggingSetName.` or `Export.ExportSetName.` . You can optionally specify a connection name in front of this,

ConnectionName.Logging.LoggingSetName. If the logging or export set is running, changing most parameters will take immediate effect.

Variables:

.strLoggingMethod: a string, either "ASCII Delimited", "ODBC Database", "Binary - 64 bit floating point", "Binary - 32 bit floating point", or "Binary - 32 bit unsigned integers". This should not be changed while the set is running or unpredictable results will occur.

.strFileName: changing the file name while the logging or export set is running will close the current file and open the new file.

Application: if you are logging batch data and wish to start logging to a different file, simply set this variable to your new file name. You can use an edit box component or any of the components with actions, specifying **Logging.LoggingSetName.FileName** as the action/set channel or use a sequence with assignment: **Logging.LoggingSetName.FileName = "c:\newfile"**

.strTableName: changing the table name while the logging or export set is running will switch logging to the new table, creating it if necessary.

.AlignMismatchEmpty: 0 or 1

.AlignThreshold: numeric in seconds

.AutoSplit: 0 or 1

.DataFileSize: numeric in rows

.ExcelTime: 0 or 1

.IncludeAllTime: 0 or 1

.IncludeHeaders: 0 or 1

.strDelimiter: a string

.LoggingAllValues: 0 or 1

.OverwriteExisting: 0 or 1. Not an option in the logging set view, but can be forced to 1 programmatically for logging sets.

.RefreshRate: numeric in seconds. Determines how often the set writes buffered data. This should be 0 for export sets otherwise the export set will repeat the export at the given rate. For logging sets this defaults to 1.

.Snapshot: 0 or 1

.TimeOffset: this value is added to the time columns in the logging set. This is used when working with MS SQL databases. DAQFactory can use the time standard of Excel and Access, and for some reason, when using this format with MS SQL the time is offset by two days. Apparently a bug in MS SQL. You can therefore use this parameter to correct this bug.

.TimeSigFigs: numeric

.CreateHeaderFile: 0 or 1

.LogUnconverted: 0 or 1

.AutoStart: 0 or 1

.Running: 0 or 1, read only.

Functions:

.Start(): starts the set. If the set is already started, nothing happens.

.StartStop(): if the set is stopped, the set is started. If the set is started, the set is stopped.

.Stop(): stops the set. If the set is already stopped, nothing happens

These functions are not applicable or available for export sets, and only apply to logging sets:

.ClearChannelList(): clears the list of channels to be logged. Usually you will call this and then the AddChannel function to programmatically create a list of channels on the fly.

.AddChannel(name, [sigfigs]): adds a channel to the list of channels to be logged. Name is a string. Sig figs is optional and defaults to 6.

.RemoveChannel(name): removes the given channel from the list of channels to be logged.

Application: The above three functions can be used to programmatically create a logging set. You could pretty easily create a way for the user to enter in channels to log at runtime. We do not recommend using these functions while the logging set is running.

These functions are not applicable or available for logging sets, and only apply to export sets:

.ClearExpressionList(): clears the list of expressions to be exported. Usually you will call this and then the AddExpression function to programmatically create a list of channels on the fly.

.AddExpression(Name, Expression, [sigfigs]): adds an expression to the list of expressions to be logged. Name and Expression are strings. Sig figs is optional and defaults to 6.

.RemoveExpression(Name): removes the given expression from the list of expressions to be logged.

Application: The above three functions can be used to programmatically create a logging set. You could pretty easily create a way for the user to enter in channels to log at runtime. We do not recommend using these functions while the logging set is running.

.AddValue(name, value): this function is used to manually add data to the logging set. First you must manually add the column using the functions below or by clicking the Manual button in the logging set view. Then you can add data to this column by calling this function with that column name and the desired data. The data must have time associated with it as the time is used to determine which row the new data applies to. The time of the data points must be within the RefreshRate (1 second by default) of the current time or the data will appear in your file out of order.

Application: The addvalue() function is used to add calculated data to your logging set. For example, let us assume that you have a channel X and a channel Y and you want to log X/Y in a column called Ratio in your logging set. To do this:

1. In your logging set, hit the Manual button, and enter **Ratio**.

2. Create a sequence:

```
while (1)
  Logging.MyLoggingSet.AddValue("Ratio", X[0] / Y[0])
  wait(1)
endwhile
```

3. Now, when you run the sequence and the logging set, the ratio of X/Y will also be logged.

Application: The problem with the above technique is that the time of the logging of the ratio is determined by the sequence, not the time X or Y was acquired. To make things more event driven, replace your sequence in step 2 above with an event assigned to either the X or Y channel:

```
Logging.MyLoggingSet.AddValue("Ratio", X[0] / Y[0])
```

Now, the logging set gets the new value every time the channel that you applied the event to gets new data.

9.9 Direct file access

If you need more flexibility than the built in logging methods offer, you can utilize the lower level file functions. These functions allow you to directly control the reading to and writing from files as well as many other features.

Here's a list of the functions. All functions start with "**File.**". Any errors while executing the functions will throw an error that starts with "F".

File.CopyFile(oldpath, newpath, don't overwrite existing): copies the oldpath file to newpath. If overwrite is 1 and newpath exists, that file will be overwritten. If don't overwrite is 1 and newpath exists, an error will occur (C1122).

File.Delete(filename): permanently deletes filename. Caution: You are not prompted to confirm!

File.GetDiskFreeSpace(drive path): returns the amount of disk space remaining on the given disk. "drive path" can be any valid path on the desired drive.

File.GetFileExists(filename): return 1 if filename exists, otherwise it returns 0

File.MakeDirectory(directoryname): creates the given directory.

File.MoveFile(oldpath, newpath): moves the oldpath file to newpath. newpath cannot exist already or an error will occur (C1121).

File.RemoveDirectory(directoryname): removes the given directory if no files exist in it

File.Rename(oldname,newname): renames the oldname file to newname

Application: All the above functions allow you to manipulate your files and monitor the file system.

File.GetFileNameList(path): returns an array of strings containing the names of all the files matching the given path and wildcard. The name does not include the path. For c:\mydir\myfile.txt, the name is myfile.txt.

File.GetFilePathList(path): same as above, but returns the whole path (c:\mydir\myfile.txt) for each file

File.GetFileTitleList(path): same as above, but only returns the file title (myfile) for each file.

> **Application:** The above three functions are useful for retrieving a list of files. For example, if you wanted to open all the .csv files in a particular directory:
>
> ```
> Private.strPaths = File.GetFilePathList("c:\data*.csv")
> if (IsEmpty(Private.strPaths))
>   return    // if nothing found, we have to return
> for (Private.count = 0, Private.count < NumRows(Private.strPaths), Private.count++)
>   Private.handle = File.Open(Private.strPaths[count],1,0,0,1)
>   ...
> endfor
> ```

File.FileOpenDialog([default file], [initial dir], [filter], [title]): all parameters are optional. This displays a standard windows dialog prompting the user for a file and returns the selected file path or an empty value (use IsEmpty() to determine) if the user cancels out of the dialog. This function can only be called by sequences running in the main application thread (a quick sequence action)

File.FileSaveDialog([default file], [initial dir], [filter], [title]): same as above, but the user is prompted to save instead of open.

> **Application:** These two functions are great for giving the user a way to specify a logging file. Apply them to a button labelled "Set Logging File". Here is a sample Quick Sequence action that requests a file name and then sets the logging set "Log" to log to that name:
>
> ```
> Private.strFileName = FileOpenDialog()
> if (!IsEmpty(Private.strFileName))
>   Logging.test.FileName = Private.strFileName
> endif
> ```

All the rest of the functions are for working on a single file, either reading or writing, and require a file handle which is returned from the File.Open() function. This handle should be kept and passed to all functions that are going to work on the file opened. This allows you to have multiple files open at the same time. The handle returned provides a way to track which file you are working on. For example:

```
Var.FileHandle = File.Open("myfile.txt",1,0,0,0)
Var.strIn = File.Read(Var.FileHandle,10)
File.Close(Var.FileHandle)
```

This function opens myfile.txt and reads 10 characters from it. It is good practice to always close your files when you are done with them, but if you forget, DAQFactory will close all open files when you quit the program.

File.Open(filename,read,write,append,text): all parameters are required. This opens the given filename for manipulation and returns a handle to the file. Set the read, write, and append parameters to 1 or 0 depending on if you want to open the file for reading, writing, and if writing, if you want to append to or overwrite an existing file. If the file does not exist, the append parameter is ignored. The last parameter determines if the file is an ASCII text file or not. The Read and Write functions work differently for text files. If you have write = 1 and append = 1 then the file pointer is automatically put at the end of the file. Otherwise it is placed at the beginning.

File.Close(handle): closes the given file. The handle is a number returned by the FileOpen() function.

File.CloseAll(): closes all files currently open. This is most useful if you lose track of a file handle (for example, stored in a private variable in a sequence that stopped before it closed the file).

File.Abort(handle): closes the given file ignoring any potential errors.

File.Read(handle,number of characters): If the file is opened with the text parameter set to 0, this function reads the given number of characters from the given file. Note that less characters may actually be read if the end of file is reached. If the file is opened with the text parameter set to 1, then the number of characters parameter is ignored and the function reads until it finds a newline character. The newline character is not returned. This function always returns a string. The file must be opened with the read parameter set to 1.

File.ReadDelim(handle, index, parse by, eol character, number of lines, [string]): This function allows the rapid reading of a delimited file such as the common files generated by DAQFactory logging sets. Parsing begins at the current file location, so if you have a header line, simply do File.Read(handle) to read the first line, putting the file location at the beginning of the second line. This function only works if the file is opened with the text parameter set to 1. This function will read the item specified by index, where 0 is the first item on each line, in a list parsed by parse by, and a line ended with eol character. This will be repeated for the specified number of lines, or if -1 is passed for number of lines, until the end of file is reached. If string is specified and is 1, the result will be an array of strings. Otherwise, each item will be converted into a number and an array of numbers is returned, with NaN's returned whenever an item cannot be converted properly to a number. You can specify a negative index to retrieve all items in one read, returning a 2d array. With larger files, it is actually slower to read the entire file at once than to read individual items one at a time.

Example: if you had the file:

1,2,3,4

5,6,7,8

ReadDelim(handle,2,",",chr(10),0) would return {3,7}

ReadDelim(handle,-1,",",chr(10),0) would return {{1,2,3,4},{5,6,7,8}}

File.ReadInter(handle, offset, interval, number of characters, number of intervals): This function allows the rapid reading of a binary file containing interleaved data. It will jump to the offset byte of the file, then read the specified number of characters. It will then jump by interval bytes (from the beginning of the block) and read the specified number of characters again, repeating for the specified number of intervals. If the number of intervals is set to 0, it will read until the end of file is reached. Typically this function will be used in combination with the To. and From. function to convert a binary data file into actual numbers. The file must be opened with the text parameter set to 0. For example, let's say your file contains the bytes 0 1 2 3 4 5 6 7 8 9. If you did ReadInter (handle, 2, 4, 2, 2) you would get {{2,3},{6,7}}. You could then use, for example, the To.Word() function on the result to yield {770,1798}

File.Write(handle,string to write): writes the given string to the file. Note that you must add any desired CR/LF characters unless the file is opened with the text parameter set to 1 in which case a newline character is added automatically. For a file opened with the text parameter set to 0, the string is written verbatim with nothing added. The file must be opened with the write parameter set to 1.

File.WriteDelim(handle,{data}, delimiter, eol character): writes the given array of data to a delimited file with the specified delimiter and end of line character. This is basically the opposite of File.ReadDelim(). The file must be opened with the write parameter set to 1 and the text parameter set to 1.

Example:

File.WriteDelim(handle,{{1,2,3,4},{5,6,7,8}},",",chr(10)) would write this to the file:

1,2,3,4

5,6,7,8

File.Flush(handle): writes any cached data to disk. For performance reasons, windows keeps some data in

memory until a large enough block can be written to disk. This forces windows to write whatever it has kept in memory to disk.

File.Seek(handle,offset): jumps to the offset position from the beginning of the given file. Offset is in bytes and must be positive or zero.

File.SeekRelative(handle,offset): jumps the given offset from the current location in the file. Offset is in bytes and can be positive or negative or zero.

File.SeekToBegin(handle): jumps to the beginning of the file. This is the same as File.Seek(handle,0).

File.SeekToEnd(handle): jumps to the end of the file. This is the same as File.Seek(handle,File.GetLength (handle))

File.GetLength(handle): returns the number of bytes in the given file

File.SetLength(handle,new length): sets the given file to have a given length, truncating if necessary.

File.GetPosition(handle): returns the current position in the file that data will be read or written.

File.GetFileName(handle): returns the file name of the current file

File.GetFileTitle(handle): returns the file title of the current file

File.GetFilePath(handle): returns the full file path of the current file

9.10 Questions, Answers and Examples

9.10.1 Write data to a file from a sequence

This sample shows how to open a file for writing and then writes to that file a line of data. There are actually two ways of doing this. One using low level file I/O which is shown first. The other is to use an export set which is shown afterwards.

Step 1: Go to Channels and Add a new channel that will be used to provide data

Channel Name: MyChannelName
Device Type: Test
I/O Type: A to D
Chn #0
Timing: 1

This simply creates a channel with some sample data for us to use.

Step 2: Go to SEQUENCES and create a new sequence called **WriteFile**. Insert the code below.

```
// Open File and get handle
Private FileHandle = File.Open("C:\myfile.txt",0,1,1,1)
// Format string
Private string strText = FormatTime(MyChannelName.Time[0]) + "," + DoubletoStr(MyChannelName[0])
// Write data to text file
File.Write(FileHandle,strText)
// Close the file
File.Close(FileHandle)
```

Step 3: In the Workspace, go to Page_0 and add a button by right clicking on the page and selecting Buttons&Switches => Button.

Right click on the button and select Properties. Give the button the following properties:

Main Tab-
Text: Write File
Action Tab-
Action: Start/Stop Sequence

Sequence: WriteFile

Click **OK**

This creates a button to start the sequence and therefore write a line to the log.

Step 4: Go to PAGES: Page_0 and click on the button once a second for several seconds and then go to the root directory and verify the data written.

The file written to, and the directory, was set by this function: File.Open("C:\myfile.txt",0,1,1,1) from Step 3. The 0,1,1,1 indicates how the file was opened. The 0,1 at the beginning means it was opened for writing. The 2nd "1" means any existing file will be overwritten, and the last "1" indicates the file is opened in text mode, meaning a carriage return / line feed is appended to each line written using Write().

The above method gives the most flexibility in how and when your data is logged. But, it does not allow you to log in binary format or to a database. For this, you will need to use a logging or export set. A logging set is designed to log data continuously, which isn't the goal of this sample, so an export set is the other choice. Continuing this example we'll show how to do the same thing with an export set:

Step 5: Go to Export: and add a new export set called **`ExportLine`**. Set the following properties:

Main Tab-
File Name: C:\myfile.txt
Details Tab-
Mode: Fixed Interval
Interval: 1
Type: Snapshot

There are other settings you can use to tweak how the data is written. For example, the default in an export set is to put a header line at the top, to append to an existing file, and to use Excel time, which is decimal days since 1900. The sequence we created above does not create a header line, it overwrites any existing file, and it uses DAQFactory time, which is seconds since 1970.

Step 6: On the main tab of the export set, hit Add to add a new row. Enter **`MyData`** for the name, **`MyChannelName[0]`** for the expression and **`6`** for Figs.

This is a key step. You must specify [0] after MyChannelName. This indicates that you just want to export the most recent value and keeps the export set from exporting more than one line of data. Setting the export set in Fixed Interval mode is also required to generate a single line of data.

Step 7: In the Workspace, go to Page_0 and add a button by right clicking on the page and selecting Buttons&Switches => Button.

Right click on the button and select Properties. Give the button the following properties:

Main Tab-
Text: Export File
Action Tab-
Action: Quick Sequence
beginexport(ExportLine)

Click **OK**

This creates a button to start the export set and therefore write a line to the log.

Step 8: Go to PAGES: Page_0 and click on the Export File button once a second for several seconds and then go to the root directory and verify the data written.

Which method to use, file I/O vs Export Set really depends on whether you need to log to a database or in binary. If you do, then you have to use an export set, otherwise file I/O gives more flexibility. Even if you are logging in ASCII mode, you may find export sets easier as they have a number of nice automatic features, like header lines, excel time, easy control of significant figures, etc.

9.10.2 Read a file and set an output channel from its data

This sample shows how to write data to a file one line at a time delimited by commas. It then shows how you can

read that file and set independent outputs with the delimited data every 1 second.

Step 1: In the Workspace go to **CHANNELS:** and add the following channels:

Two input channels:

Channel Name: MyChannelNameIn_0, MyChannelNameIn_1
Device type: Test
I/O Type: A to D
Chn#: 0,1
Timing: 1

Three output channels:

Channel Name: MyChannelNameOut_0, MyChannelNameOut_1
Device Type: Test
I/O Type: D to A
Chn#: 0,1

In your actual application, you would of course use your own input and output channels.

Step 2: In the Workspace right click on **SEQUENCES** and Add a sequence called **CreateFile**. Copy the following code into the sequence window.

```
// Opens the file sets the variable FileHandle
Private FileHandle = File.Open("C:\mydata.txt",0,1,0,1)
Private Count
Private string strString
for (Count = 0, Count < 10, Count++)
  //Concatenates the inputs into a comma delimited string
  strString = DoubletoStr(SysTime()) + "," + DoubletoStr(MyChannelNameIn_0[0]) + "," + DoubletoStr
(MyChannelNameIn_1[1])
  // Writes the string to the file
  File.Write(FileHandle, strString)
  // Writes every 1 second
  delay(1)
endfor
File.Close(FileHandle)
```

This sequence will open the file c:\mydata.txt and then every second for ten seconds write the current values of MyChannelNameIn_0 and 1. This does almost the exact same thing as a logging or export set running in Fixed Interval / Snapshot mode, except that it runs for only ten intervals. The file I/O part is straightforward. The file is opened and the handle saved in FileHandle. The 0,1,0,1 in the Open() function indicates that we are writing to the file and that we are in text mode. Once open, we use the Write() function to write each string to the file. Since we opened the file in text mode, a carriage return / line feed is automatically written with each line as well. Finally we close the file once we are done writing.

Step 3: Add another **SEQUENCES** called **ReadFile**. Copy the following code into the sequence window.

```
// Opens the file sets the variable FileHandle
Private FileHandle = File.Open("C:\mydata.txt",1,0,0,1)
Private string strIn
while (1)
  // read a line
  strIn = File.Read(FileHandle)
  // This checks for an empty string
  if(strIn == "")
     // Breaks out if empty string
     break
  endif
  // This code Parses the String and sets the output channel
  MyChannelNameOut_0 = StrToDouble(Parse(strIn,1,","))
  MychannelNameOut_1 = StrToDouble(Parse(strIn,2,","))
  // Reads 1 second at a time
  delay (1)
endwhile
// Closes the file
File.Close(FileHandle)
```

In this sequence, we read the file back in, one line at a time, one line each second. With each line, we separate the line and set the outputs. The separation is done by the Parse() function. So if the strIn was "102932435.324,2.423,5.392", then Parse(strIn,1,",") would return "2.423". Since this is still a string, we call StrToDouble() to convert it to a number, and then set our output by simply setting the channel to the value we read. The file I/O part is very similar to the write sequence. The file is opened, but this time we open it in read mode (by using 1,0,0,1 instead of 0,1,0,1). Since we open it in text mode, the Read() function will read a line up to the CR/LF. Each read() returns a new line, or an empty string if the end of the file is reached. At the end, we close the file for proper cleanup.

Step 4: Add another **SEQUENCES** called **StartUp**. Copy the following code into the sequence window.

```
MyChannelNameOut_0 = 0
MyChannelNameOut_1 = 0
MyChannelNameOut_2 = 0
```

Check the **Auto-Start** box

Click **Apply & Compile**

This sequence simply sets the outputs to a default value of 0. This is not required for the functionality of this sample, but initializing outputs is a good habit to be in.

Step 5: In the Workspace select Page_0. Add a button to the page by right clicking on the page and selecting Buttons & Switches => Buttons. While the button is still selected, right click on it and select Properties. Give the button the following properties:

Main Tab-
Text: Make File
Action Tab-
Action: Start/Stop Sequence
Sequence: CreateFile

Click **OK**

Step 6: In the Workspace select Page_0. Add a button to the page by right clicking on the page and selecting Buttons&Switches => Buttons. While the button is still selected, right click on it and select Properties. Give the button the following properties:

Main Tab-
Text: Set Outputs
Action Tab-
Action: Start/Stop Sequence
Sequence: ReadFile

Click **OK**

In the above two steps, we created new screen components to trigger the sequences we created. Buttons aren't really the best choice for triggering sequences since they don't tell us the current state of the sequence. For a sequence that would run quickly and stop, this is not an issue, but our sequences here will take 10 seconds to run. This does, however, show that you can use many different components to do different actions. In this case, a Variable Text component would be a better choice. When using the Variable Text component, all you have to do is click on the Quick - Sequence Trig button on the main page and select your sequence and all the other properties will be set for you.

9.10.3 DAQFactory / Excel Time

Question:

I wanted to check on your non-excel formatted time. Is it the number of seconds since 12:00 am, Friday, January 1, 1904, UTC?

Answer:

No:

Excel Time: Decimal days since January 1, 1900.
DAQFactory Time: Seconds since January 1, 1970.

These are both local time, not UTC.

To convert:
ExcelTime = (DAQFactoryTime / 86400) + 365*70 + 19
DAQFactoryTime = (ExcelTime - 365*70 - 19) * 86400

As a side note, windows uses DAQFactory time internally for most things. Also, Excel time does not offer the same microsecond precision that DAQFactory time does.

9.10.4 Logging on an event

Question:

I want to log a line of data whenever my channel has a value greater than 3. How would I do that?

Answer:

There are two ways. The first, and more flexible is to use Direct File Access functions to do all the logging, but with the flexibility comes additional work, so we'll explain the easier way which is to use an export set. In your export set, put the names of the channels you want to log and make sure to follow them with [0]. This will cause the export set to only write the most recent values for your channels. Then, on the Details page of the export set, select "Fixed Interval" and Type = "Snapshot". Now, whenever you start the export set a single line of data will be written.

So, assuming you've created this export set, to trigger when your channel is greater than 3 you should use the event of the channel. Click on the name of the channel in the workspace to display the channel view, and then click on the Event tab. The Event is sequence code that gets executed whenever a new value is read or set on this channel. Put in something like this:

```
if (MyChannel[0] > 3)
  beginexport(MyExport)
endif
```

This assumes, of course, that the channel name is MyChannel, and the export set is MyExport.

9.10.5 Changing the logging file name every month

Question:

I want to create a new data file every month with the month/year in the logging file name. How do I do this?

Answer:

Simply use date / time specifiers in the file name for the logging set. These specifiers are the same as those used in the FormatDateTime() function, section 4.12.11. So, to do every month, you'd do something like:

c:\myfolder\myFile%y%m

9.10.6 Changing the logging file name every day

Question:

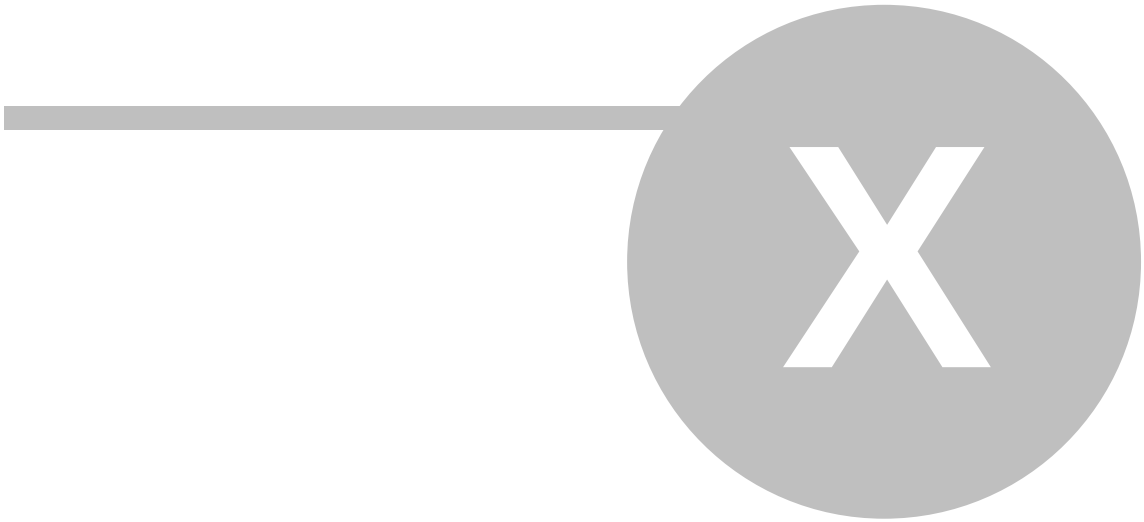
I want to create a new data file every day with the day/month/year in the logging file name. How do I do this?

Answer:

This is the same as the last, just with a different specifier:

```
c:\myfolder\myFile%y%m%d
```

10 PID Loops



10 PID Loops

10.1 PID Overview

PID is a common algorithm for process control. It is used to keep a process variable, such as temperature, set at a certain set point by controlling an output variable. PID is made up of three parts, proportion (P), integral (I), and derivative (D). There are many good books on PID and recommend reading them for more information on PID control. This chapter describes PID control within DAQFactory and assumes a basic understanding of the algorithm, at least its uses and the meaning and effect of the three terms.

Within DAQFactory you can create as many PID loops as your computer will handle. Each loop runs in a separate thread within DAQFactory. A thread is like a separate program running concurrently with other threads running within DAQFactory. You can stop and start loops as necessary, independent of each other. Each PID is a standalone object and so whether using one PID loop or many, the process is the same.

10.2 Creating a new PID loop

To create a PID loop, click on **PID:** in the workspace. This will display the PID summary view which will list any PID loops and their current status. Click **Add**. Enter a name for the loop. The PID view will appear. This allows you to set all the parameters of the PID, do autotuning, and set an event to execute with each PID loop iteration.

PID parameters:

Process Variable: an expression that when evaluated is used in the loop. This should be a singular value, i.e. `PV[0]` DAQFactory automatically does a 4 point weighted average to reduce the effects of noise. You can add to this by doing a mean: `Mean(PV[0,9])`

Set Point: an expression that when evaluated is used as the target for the process variable. This should also be a singular value. If you want this adjustable from a screen, use a variable, v channel or test channel.

Output Channel: a channel or variable name to set the result of the PID.

P, I and D: These take numeric values for the three parameters that make up a PID loop. I and D are in minutes. A value of 0 for I and/or D disables this part of the calculation, even though the I term is used in the denominator.

PID Type: There are two PID algorithms available. The standard algorithm applies the Gain to all three parts of the calculation, thus Out = P * (e + i/I - D*dPV). The alternate, "separate" algorithm, keeps the gain solely in its own section, thus Out = P*e + i/I + D*dPV. This is a less standard algorithm, but tends to be much easier to tune because each term is independent.

SP Range: holds the possible range of set points and is used to help scale the output of the PID calculation.

Out Range: limits the output to a certain range of values

Loop Interval: how often the PID is calculated and the output channel is set. This should not be below 0.01 without a multi-processor system. Smaller values use up larger amounts of processor time.

Integral Limit: limits the accumulated integral to +/- this amount. This is an easy way to avoid integral wind-up.

Reverse Acting: if checked, then a negative output results in a positive moving process variable.

Reset on SP change: if checked, then the accumulated integral and internal process variable and derivative smoothing is reset whenever the set point changes.

Reset on Start: if checked, then the accumulated integral and internal process variable and derivative smoothing is reset whenever a loop is restarted. Otherwise, the integral will continue to accumulate while the PID is stopped. The integral will not start to accumulate until the PID is started for the first time.

Running a PID Loop:

Once the PID is set up you can start and stop loops the same way you would start/stop a sequence. This includes component actions and sequence commands. The Settings tab of the PID View has some useful diagnostic information for a running sequence. To the right of the process variable, set point, and output channel fields numbers will appear showing the current value for each of these terms. The process variable is smoothed, so the value displayed will not be the exact current value of your expression, but it will be the value fed into the PID calculation. To the right of the P, I, and D terms numbers will appear showing the current effect related to the term. For P, this displays P*e, for I, P*i/I or i/I depending on the PID type, for D either P*D*dPV or D*dPV. Unless you have an PID event returning a different value, the sum of these three terms will equal the output value. Finally to the right of the integral limit field, a number will appear showing the current accumulated integral. This is useful to determine an appropriate integral limit.

In addition, a graph of the process variable, set point and output is displayed along the right. This is a standard DAQFactory graph, so you can double click on it to change graph scaling and other parameters.

Note: Unlike sequences, you do not need to stop and restart your PID loop every time you change a parameter. The option will apply immediately to a running PID loop.

Example: As an example, we will create a simulation temperature system and use a PID loop to control it.

1. Create a new sequence, call it whatever you would like. To create a sequence, right click on **SEQUENCES:** in the workspace and select **Add Sequence**. Enter a name for the sequence and the sequence view will appear. Enter the following text into the sequence:

```
global PV = 1
global Out = 0
global SP = 50
while(1)
  PV += Out - 0.25
  delay(1)
endwhile
```

The first three lines initialize the three variables we are going to use in our simulation. The rest does the actual simulation, increasing the process variable (PV) by the output of the PID loop, but also subtracting 0.25 from it every loop to simulate cooling. Click **Apply & Compile** to save your changes.

2. Create a new PID loop, again call it whatever you would like. Only three parameters need to be set:

Process Variable: `PV`

Set Point: `SP`

Output Channel: `Out`

3. Click **Apply** to save your changes.

4. Start your sequence. To do this, click on the + sign next to **SEQUENCES:** to expand the tree, then right click on your sequence and select **Begin Sequence**.

5. Start your PID loop. To see a different way to start loops, click on **PID:** in the workspace to display the PID summary view, then click **Begin** to start the highlighted PID loop in the view. There will be only one and it should be highlighted.

6. At this point, both your sequence and PID loop will be running. Go to the PID view for your PID loop by clicking on the PID loop name in the workspace. You may need to expand the PID tree by clicking the + next to **PID:**. Next, click on the **Graph** page to display a graph of our three parameters.

7. Depending on how long it took for you to get between step 5 and step 6, the system may have stabilized already. So, to create a disturbance we need to change the set point. To do this, go to the Command / Alert window and type: `SP = 20` and hit Enter. This will change the set point to 20 and the graph should show the change. The blue trace is your set point, the red trace is the output from the PID loop, and the green trace is the process variable.

At this point feel free to play around a little with some of the parameters. You do not have to stop the loop to adjust the parameters. Remember though that the simulation is very basic, so autotuning will not really work on it. As you gain experience with sequences, you may want to play with the simulation and try and improve it.

10.3 Autotuning

PID support includes autotuning, which helps determine the best P, I, and D parameters for your particular process. To use, you must have your system in a stable state, with the process variable sitting close to the set point. Once there, select a relay height, which is the amount the output will be adjusted depending on whether the process variable is above or below the set point. Unlike PID control, the output just alternates between these two values. Press the start key to start the autotune. The system will need to go through at least two cycles, at which point it can calculate P, I, and D parameters. Ideally you should wait until the period and height of the process variable stabilizes. At this point (or any other point) you can use the **Use Results** button to copy the results of the autotune into the P, I and D fields. You will then need to apply the changes to keep the new parameters.

Note: the big advantage of relay autotuning over other methods is that your system does not need to be taken far out of range to autotune. You can and should specify a very small relay height to keep your system within acceptable parameters. It is best to start with a very small and relay height and work up to ensure that you do not accidentally drive your system into an unacceptable range.

Example: If you want to see autotuning in action, though perhaps not showing its true power, create the simulation system example described in the Creating a PID loop section. Then:

1. Make sure both the sequence and the PID loop are running. Go to the PID view for the PID loop and select the AutoTune page.

2. The autotune page also shows a graph. If you recently changed the setpoint, wait until the process variable stabilizes close to the setpoint.

3. Enter a relay height of 2, then click **Start** to start the autotune procedure.

What you will see at this point is the output red trace alternating between +2 and -2 relative to your setpoint as the process variable in green itself alternates above and below the setpoint. After two such oscillations, the Calced P, I and D values will be displayed. With every cycle, these numbers will change slightly as the autotuning zones in on the best number. When you are satisfied with the values, you can stop the autotuning and click **Use Results** to copy the values into the loop parameters. You will also have to click **Apply** to save these new values into the loop. Because the simulation is so basic, the autotune probably will not improve the loop much and may even make it worse. In a real system however, autotune can really help get you close to the ideal parameters.

10.4 PID Event

You can optionally provide sequence code that is executed every time the PID loop calculates its setting. The following Privates are updated each time as well and available for use within the event:

effectI - calculated I for PID

effectD - calculated D for PID

effectP - calculated P for PID

- the normal output is the sum of these three parameters

calcedSP - actual SP value (result of SP expression)

calcedPV - actual PV value (result of PV expression)

You can optionally return an output value to use in place of the normal output value for this loop iteration. If you do so, the output range parameters of the PID are ignored. If you need to limit your outputs while using the event you'll need to do so in the event. For example:

```
Private out = (effectP + effectI + effectD) + 2.5
if (out > 5)
  out = 5
endif
if (out < 0)
  out = 0
endif
DAC0 = out
```

10.5 PID Variables and Functions

PID loops have variables and functions that allow programmatic access to the loop. Most of the variables match up with parameters available from the PID loop. All variables and functions start with **`PID.PIDName.`** or **`ConnectionName.PID.PIDName.`** where PIDName is the name of the PID loop you wish to work with.

Variables:

.P : numeric

.I : numeric

.D : numeric

.SPMax : numeric

.SPMin : numeric

.OutMax : numeric

.OutMin : numeric

.ReverseActing : 0 or 1

.LoopInterval : numeric in seconds

.HistoryLength : numeric. This determines the number of points kept in the PV, SP, and OV histories. The default is 3600. If you don't need these histories, you may want to set this parameter to 1. In addition to programmatic access, the histories are used to generate the PID view graphs.

.AutoTune_P: read only numeric result of the autotune

.AutoTune_D: read only numeric result of the autotune

.AutoTune_I: read only numeric result of the autotune

.StartAutoTune(height): starts up the autotune procedure with the given relay height. Height must be non-zero.

.StopAutoTune(): stops the current autotune procedure if running. Throws an error if the autotune is not running.

.PVHistory: an array with the history of process variable values with time. Use subset notation to pull parts of this array: **`.PVHistory[0]`**

.SPHistory: same for set point

.OVHistory: same for the output variable

10.6 Questions, Answers and Examples

10.6.1 PID Algorithm?

Question:

Would you have any additional information that you could provide me on the type of PID algorithm used? I noticed that if I set the proportional gain to 0, the PID output never changes even though I still have a valid Integral and

Derivative term. This implies to me that the proportional term might be used as a multiplier for the Integral and Derivative terms as well. I know that there are different PID algorithms; Series (interacting) PID, Parallel PID, and several different equations, but I don't know which one you use. Any additional information that you could provide would help me tune our system correctly.

Answer:

There are two choices:

With the standard algorithm, the gain applies to all three terms. It boils down to:

effectP = P * error

effectI = P / I * integral

effectD = -1 * P * D * dPV

There is some additional stuff to smooth out the error, derivative and other such items.

If you want an ID only loop, just set P to 1, then do `Out = effectI + effectD`

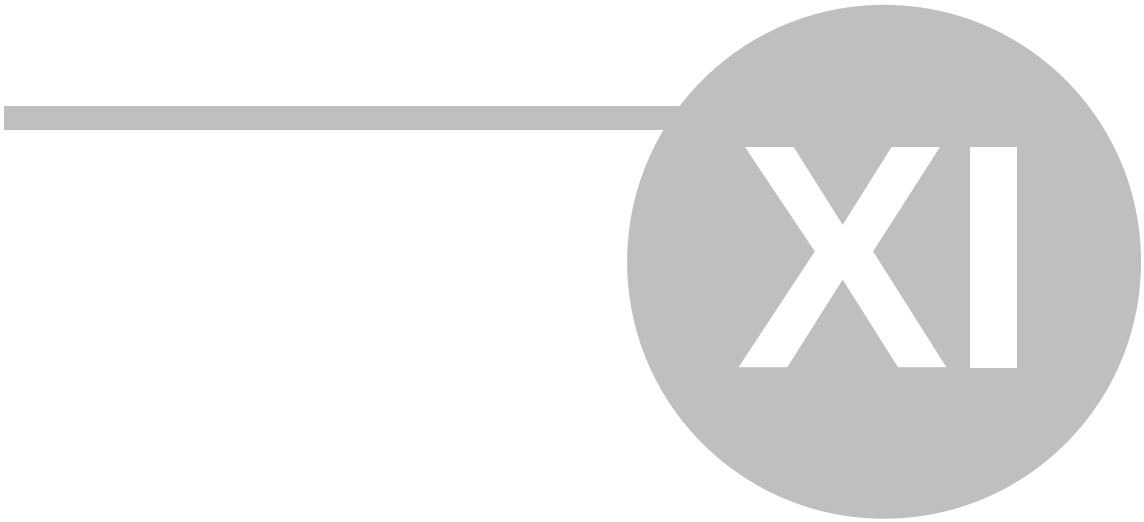
With the separate algorithm, the gain only applies to the first three terms:

effectP = P * error

effectI = integral / I

effectD = -1 * D * dPV

11 Alarming



11 Alarming

11.1 Alarming Overview

The alarming features in DAQFactory allow you to watch for certain alarm events. When an alarm event occurs, the alarm is said to have fired. Once fired, it will show up as an active alarm in the alarm summary view. Alarms remain active until a reset condition occurs at which time they are reset. Even when an alarm resets it still must be acknowledged for it to stop displaying in the alarm summary view. In this way if a part of your system goes out of bounds for a short time when you weren't watching, you will still see this as an alarm.

Alarm events are logged to disk either as an ASCII file or in an ODBC database. Alarm events can also trigger sequence events to perform other actions when an event occurs.

Alarming is only available in the pro version of DAQFactory.

11.2 Creating a new alarm

To create an alarm, right click on **ALARMS:** in the workspace and select **Add Alarm**. You will be prompted for a name for your alarm. Just like other names in DAQFactory, alarm names must start with a letter and only contain letters, numbers or the underscore. Once you've entered your alarm name, the alarm view will appear allowing you to edit the parameters of your alarm. To return to the alarm view for an alarm later, simply click on the alarm's name in the workspace under **ALARMS:**

Alarms in DAQFactory are a bit different than many other SCADA applications in that the alarm condition is not channel dependant. There are no High and High-High alarms for each channel, but rather expressions that determine the alarm and can be based on values from several channels at once. Because of this, all alarms are checked every time a new data point arrives on any channel. If you have a large quantity of alarms and a large quantity of channels, this can cause your acquisition loops to slow as each alarm is evaluated with each individual channel read. If you have this situation, you will want to use the Alarm.Paused and Alarm.CheckAlarms() variable and function to manually control when DAQFactory checks alarms. This is described in the section on Alarm Variables and Functions.

Alarm Parameters:

Condition: when this expression evaluates to a non-zero number then the alarm fires. If an alarm sound file is specified, it will be played. This is evaluated for every incoming data point, but does not evaluate streaming, or image data. Data added to a channel using the AddValue() function will also cause alarms to be evaluated.

Reset Condition: when this expression evaluates to a non-zero number and an alarm has fired, the alarm is considered reset. The alarm will still sound until acknowledged.

Description: description displayed in the alarm table view

Priority: determines the color and order alarms are displayed. Log only alarms do not display in the alarm table and are simply logged. They also automatically acknowledge.

Alarm Sound: a wave file that is played continuously until the alarm is reset and acknowledged or another alarm with a sound fires.

Enabled: determines if the alarm is evaluated

Help: extra information about the alarm

Events: Alarms also have three events, sequence steps that will be executed when a particular event occurs. There is one for when the alarm fires, when the alarm resets, and when the alarm is acknowledged. Each has its own tab in the alarm view. If you return(0) from the event, then the alarm fire, reset or ack will be ignored.

Application: One possible use for the alarm event is preventing a user who needs to see a list of alarms from acknowledging them. The alarm summary view has an Acknowledge All button, which cannot be disabled. You can however, still prevent the user from acknowledging the alarms. To do this, simply create an ack event for each

alarm. In that event do the following:

```
if (Var.AlarmRights == 1)
  return (1)
else
  return (0)
endif
```

Then, create a way for the user to login, and in the process set Var.AlarmRights to either 0 or 1 depending on whether they are allowed to acknowledge the alarms. This can of course be applied on an alarm by alarm basis, allowing different users the ability to acknowledge only certain alarms.

Hint: You could also code the above event as:

```
return (Var.AlarmRights == 1)
```

11.3 Alarm Summary View

The alarm summary view lists all non-LogOnly alarms and their current state, with fired alarms at the top. The different alarms will display in different colors depending on their priority and will flash when fired. From this view you can acknowledge all the alarms by clicking on the **Acknowledge All** button.

To get to the alarm summary view, click on **ALARMS:** in the workspace. You can also use a component action on many components to switch to the alarm summary view. The alarm summary view is the only view available in the runtime version (the **Add** and **Set Logging File** buttons do not appear in the runtime version).

From the Alarm Summary View you can also export and import alarms to a delimited file. This is provided to allow rapid editing of alarms using a program like Excel and works just like the Channel export and import. Like the Channel version the Alarm events are not exported or imported. Priorities run from 0 to 3, with 0 being Log Only and 3 being Critical. All other columns are strings.

11.4 Alarm Logging

Alarm events are always logged. The default file and format is "Alarms.dat" in ASCII format. You can change the logging file and optional log to an ODBC database. To do so:

1) Click on **ALARMS:** in the workspace to display the alarm summary view.

2) Click on the **Set Logging File** button.

3) Enter in the appropriate values, or check the **Disable Logging?** box to turn the built in logging off. If **Log to database?** is not checked, then the file name for the ASCII file should be specified. You can specify the full path if you do not want the data being stored in the DAQFactory directory. If **Log to database?** is checked, then you should instead specify the ODBC data source name. Please see the chapter on Logging to see how to setup an ODBC database in windows. Please note that the ODBC data source name IS NOT the file name of your database.

For ASCII files, data is logged in comma delimited format:

```
Time, Alarm Name, Description, Priority, State
```

For databases, the same data is logged in separate fields. The priority is a number from 1 (log only) to 4 (critical). State is either "F" for fired, "R" for reset, or "A" for acknowledged.

11.5 Alarm Variables and Functions

Alarms have variables and functions that allow programmatic access.

First variables and functions that are global to all alarms. These all begin with **`Alarm.`**

Variables:

Alarm.FiredCountAdvisory : returns the total count of Advisory alarms fired and not acknowledged.

Alarm.FiredCountCritical : returns the total count of Critical alarms fired and not acknowledged.

Alarm.FiredCountLog : returns the total count of Log alarms fired and not acknowledged.

Alarm.FiredCountTotal : returns the total count of all alarms fired and not acknowledged.

Alarm.FiredCountWarning : returns the total count of Warning alarms fired and not acknowledged.

Alarm.strLatestAlarm : returns the name of the last alarm to fire.

Alarm.LatestAlarmColor : returns the default color, based on the priority, of the last alarm to fire.

Alarm.LatestAlarmAcked : return 0 or 1 on the acknowledge state of the last alarm to fire.

Alarm.strLatestLogAlarm : returns the name of the last alarm with LogOnly priority level to fire.

Alarm.LatestLogAlarmAcked : returns 0 or 1 on the acknowledge state of the last LogOnly priority alarm to fire.

Alarm.strLatestAdvisoryAlarm : returns the name of the last alarm with Advisory priority level to fire.

Alarm.LatestAdvisoryAlarmAcked : returns 0 or 1 on the acknowledge state of the last Advisory priority alarm to fire.

Alarm.strLatestWarningAlarm : returns the name of the last alarm with Warning priority level to fire.

Alarm.LatestWarningAlarmAcked : returns 0 or 1 on the acknowledge state of the last Warning priority alarm to fire.

Alarm.strLatestCriticalAlarm : returns the name of the last alarm with Critical priority level to fire.

Alarm.LatestCriticalAlarmAcked : returns 0 or 1 on the acknowledge state of the last Critical priority alarm to fire.

Alarm.Paused: this will pause the checking of alarms. If you have a lot of alarms, your acquisition loops can be slowed because all alarms are normally checked every time a new value comes in. To give you control, you can pause the alarm checking while you acquire data in your loop, and then resume it when done. Use this in conjunction with the Alarm.CheckAlarms() function below. There are typically two ways you can implement this:

1) You can pause alarm checking in an auto-start sequence, leaving it paused, and then manually check alarms. This is probably the preferred way. You can either create a separate sequence to check alarms at a fixed interval, or put the CheckAlarms() at the end of your polling loop.

2) You can pause alarm checking while acquiring data in a polling loop and then unpause and check alarms after completing the loop. This is typically done with acquisition done with function calls and the AddValue() function:

```
try
Alarm.Paused = 1
... Acquire data and call addvalue() to put it into channels...
Alarm.Paused = 0
catch()
Alarm.Paused = 0
endcatch
Alarm.CheckAlarms()
```

You'll want to use the try/catch block to ensure that Alarm.Pause gets set back to 0 in case of error.

Alarm.ResetCountAdvisory : returns the total count of Advisory alarms reset and not acknowledged.

Alarm.ResetCountCritical : returns the total count of Critical alarms reset and not acknowledged.

Alarm.ResetCountLog : returns the total count of Log alarms reset and not acknowledged.

Alarm.ResetCountTotal : returns the total count of all alarms reset and not acknowledged.

Alarm.ResetCountWarning : returns the total count of Warning alarms reset and not acknowledged.

Functions:

Alarm.AckAllAlarms() : acknowledges all alarms. This is the same as clicking the **Acknowledge All** button in the alarm summary view.

Alarm.Ack(Alarm Name) : acknowledges the given alarm only. For example: `Alarm.Ack("MyAlarm")`

Alarm.CheckAlarms(): manually checks all alarms. Alarms are normally checked whenever a new data point arrives on any channel. You can also trigger the checking of alarms manually using this function. This is typically used with the Alarm.Paused variable.

Individual alarms also have variables. Most correspond directly to the parameters of the alarm. All of them start with `Alarm.AlarmName.` where AlarmName is the name of the alarm.

Alarm.ListAll(): returns an array of strings with the names of each of your alarms.

Variables:

.strCondition : a string containing the expression

.strResetCondition : a string containing the expression

.strDescription : a string with the description

.Priority : a number corresponding to the priority of the alarm: 1 = Log Only, 2 = Advisory, 3 = Warning and 4 = Critical

.strHelp : a string with the help information

.Enabled : 0 or 1

.strSound : a string with the path to the .wav file

.TimeFired : the last time the alarm fired.

.TimeReset : the last time the alarm was reset. This resets to 0 when the alarm fires.

.TimeAcked : the last time the alarm was acked. This resets to 0 when the alarm fires.

.Fired : 0 or 1. If 1, then the alarm has fired, but hasn't been reset.

.Acked : 0 or 1. If 1, then the alarm has fired and been acked.

For the above two parameters:

Fired = 0, Acked = 1: alarm is enabled, but off.

Fired = 0, Acked = 0: alarm has fired and been reset, but not acked.

Fired = 1, Acked = 0: alarm has fired but not reset nor acked.

Fired = 1, Acked = 1: alarm has fired and been acked, but not reset.

11.6 Questions, Answers and Examples

11.6.1 Basic Alarming

This sample shows a single alarm in action on a test channel. The state of the alarm is displayed with a symbol component. The time of the alarm (Time Fired), time of alarm reset, and acknowledge time are also displayed.

Buttons are provided on the page for acknowledgement and resetting. This particular example shows a manual alarm reset mechanism. Typically, alarms are automatically reset by providing another expression that includes hysteresis. For example, if the alarm trigger is MyChannel[0] > 5, the reset expression might be MyChannel[0] < 4. This keeps the alarm from retriggering multiple times from a noisy MyChannel signal.

Step 1: Create a new test channel. Go to **CHANNELS:** located in the Workspace and Add a new channel with the following Properties.

Channel Name: Input
Device Type: Test
D#: 0
I/O Type: A to D
Chn #: 0
Timing: 2
Offset: 0

Click **Apply**

This just creates some generic data to trigger the alarm. Test A to D channels create a sine wave.

Step 2: Create the Alarm: Go to **ALARMS:** located in the Workspace and Add a new Alarm called **GreaterThen** with the following Properties.

Main Tab-
Condition: Input[0] > 0.98
Reset Condition: AlarmReset

Reset Event tab: add this sequence script:

```
AlarmReset = 0
```

Click **Apply**

This creates an alarm that will trigger when the Input channel is greater than 0.98. It will remained Fired until the variable AlarmReset is set to a non-zero value. The Reset Event is script that executes when the alarm is reset. In this case, we set the AlarmReset variable back to 0 to keep the alarm from resetting immediately the next time it fires.

Step 3: Create a Sequence. Go to **SEQUENCES:** located in the Workspace and Add a new Sequence called **StartUp** . Copy and paste the following code into your sequence view.

```
global AlarmReset = 0
// This code allow us to reset the variables only once.
if(Alarm.GreaterThen.Fired)
  AlarmReset = 1
endif
// Call the function to acknowledge all alarms
Alarm.AckAllAlarms()
```

This sequence simply initializes the AlarmReset variable and acknowledges any alarms. This gives us a good starting point.

Step 4: Go to a blank page and right click anywhere and select Buttons & Switches => Button. While the button you just created is still highlighted hit Ctrl-D to Duplicate it for a second button. Right click the buttons and give them the following Properties.

Main Tab-
Text: Reset

Action Tab-
Action: Quick Sequence
Sequence: Copy and paste following code

```
if(Alarm.GreaterThen.Fired)
  AlarmReset = 1
endif
```

Click **OK**

This component will reset the alarm by setting the AlarmReset variable to 1. The if() statement keeps us from resetting the alarm before it is fired. Alarm.GreaterThen.Fired will return 1 (true) if the alarm has triggered.

Main Tab-
Text: Acknowledge

Action Tab-
Action: Quick Sequence
Sequence: Copy and paste following code

```
Alarm.Ack("GreaterThen")
```

Click **OK**

This component simply acknowledges the GreaterThen alarm.

Step 5: Right click on a blank area of your page and select Displays => Variable Value. Hit Ctrl-D to Duplicate this component three times for a total of four components. Go to the Properties of each and enter as follows.

Main Tab-
Caption: Channel Input
Expression: Input[0]

Click **OK**

Next component:

Main Tab-
Caption: Time Fired
Expression: FormatDateTime("%c",Alarm.GreaterThen.TimeFired)

Click **OK**

Next component:

Main Tab-
Caption: Time Reset
Expression: FormatDateTime("%c",Alarm.GreaterThen.TimeReset)

Click **OK**

Next component:

Main Tab-
Caption: Time Acknowledged
Expression: FormatDateTime("%c",Alarm.GreaterThen.TimeAck)

Click **OK**

These components simply display the current value and the times of various alarm events. FormatDateTime("%c",...) will format a DAQFactory date/time into a string using the date/time format for your locale (i.e. month / day / year for the U.S.). Alarm.GreaterThen.TimeFired and the others are simply variables of the GreaterThen alarm that hold the DAQFactory date/time of that particular event, or 0 if it has not occurred yet.

Step 6: Right click on a blank area of the page and select Displays => Symbol. Go to the Properties and enter as follows.

Main Tab-
Expression: Alarm.GreaterThen.Fired

Color Tab-
Background Color
Threshold: 0, Color: Green
Threshold: 1, Color: Red

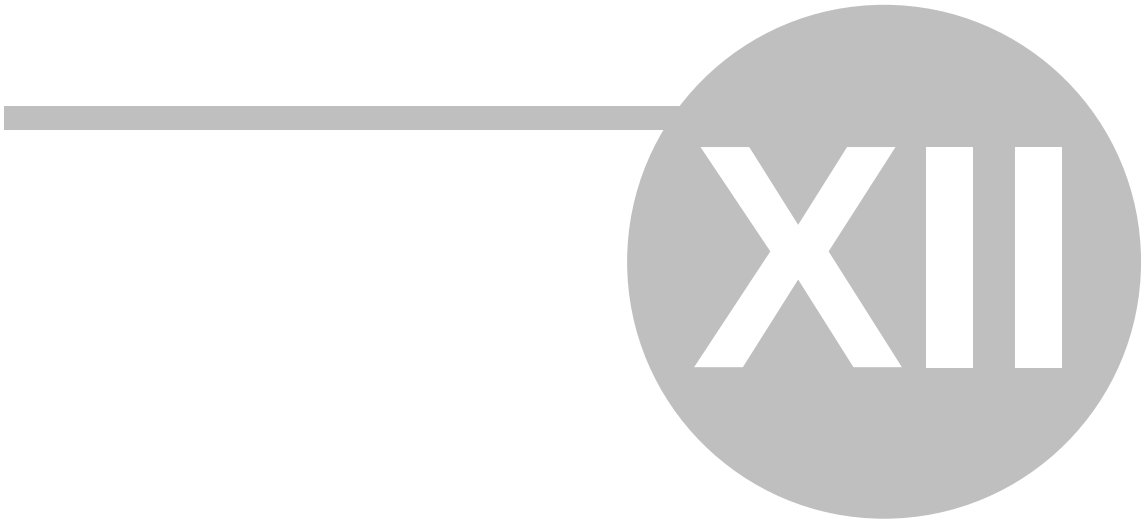
Size Tab-
Blink:
Expression: !Alarm.GreaterThen.Acked

Click **OK**

This component displays the fired state of the alarm by using the Fired variable of the alarm. As mentioned before, the Fired variable will return 1 if the alarm has triggered, and 0 if not. The background color table will therefore show green when the alarm is not triggered (0), and red when it is (1). Likewise, it will blink when the alarm has not been Acked. Acked gets set to 1 when you acknowledge the alarm, and is reset to 0 when the alarm fires.

Step 7: Right click on your sequence StartUp and select Begin Sequence. Verify that your program works. Use the Reset and Acknowledge buttons to reset the alarm.

12 Networking / Connectivity



12 Networking / Connectivity

12.1 Networking overview

DAQFactory can directly and easily connect to the DAQConnect service (see www.daqconnect.com) allowing you to view your data in real time (at least Internet real time) from a browser any where in the world without having to put the DAQFactory system on the Internet or open up your firewall or router settings. This provides a much more secure solution than allowing external Internet traffic to your DAQFactory system and eliminates the need to be an IT expert or get your IT personnel involved. It also limits the processor load on the DAQFactory system when you have multiple users viewing your data remotely. If you do not wish to use DAQConnect you can use the Page. Capture() functions and an external web server.

DAQFactory also has the ability to send emails, optionally with attachments. Emails can be sent out at any time with any message. This can be used to email you a daily status, or email when an alarm occurs. You can even have DAQFactory email you your data every day. DAQFactory can also retrieve emails from a POP3 server.

DAQFactory uses the Internet standard TCP/IP to communicate between copies of DAQFactory. Networking in DAQFactory is as simple as pointing DAQFactory to another copy of DAQFactory. The connection between DAQFactory and other copies of DAQFactory are called, appropriately enough, a connection. You can have as many connections as you need within DAQFactory to communicate with multiple copies of DAQFactory at the same time.

DAQFactory can perform Web service calls to retrieve data from the web.

DAQFactory can send and receive files using FTP.

DAQFactory supports the use of TAPI compliant modems for outgoing and incoming calls. When used with the SAPI speech engine, you can read or interpret speech over the phone line to announce alarms, settings, etc. or process commands. Detection of tones is also possible.

Finally, DAQFactory also can act as a DDE server allowing you to pass data in real time to a DDE client program such as Excel.

12.2 Real time Web with DAQConnect

DAQFactory supports direct, easy connectivity to the DAQConnect service to allow you to view your data live from a web browser without having to worry about your firewall or router settings, static IPs or anything else related to IT really. You don't need an external website either, as would be required using FTP. To learn more about the service and sign up for a free account, please visit www.daqconnect.com.

Sending data to DAQConnect:

You can easily start connecting to DAQConnect by going to **Real-time Web -> Connect...** in the DAQFactory main menu. The first time you do this in a session, you will be prompted to login or create an account. If you've already created a DAQConnect account, click login, otherwise, you can create a free DAQConnect account right from this window. If you haven't created any data sources yet, after logging in or creating an account you will go directly to the page for creating new data sources. A data source is a group of channels within DAQConnect. Typically you'll create a data source for each location sending data to DAQConnect, but you could also use separate data sources for separate applications running on a single computer. Give your data source a name, which like all DAQFactory names must start with a letter and contain only letters, numbers or the underscore. Once the data source is created, your DAQFactory document will be linked to it.

Note: if you are using a proxy to get to the Internet or a dedicated instance of DAQConnect (i.e. you access DAQConnect through a URL other than www.daqconnect.com, please see the Settings section below before you you do Connect...

To then send data to DAQConnect, you simply mark the desired channels for sending:

1) Go to the channel table by clicking **CHANNELS** in the workspace. Towards the right of the table you'll see three columns related to DAQConnect:

DAQConn?, determines if the channel data is being sent to DAQConnect. Check this box to mark the channel.

DC Hst:, determines how many data points are kept on the DAQConnect server for historical viewing. This number is limited by the plan you select. You can enter -1 to have DAQConnect automatically apply an even amount to all of your tags based on your plan, or you can specify exact amounts for each channel allowing you to have more data retained for certain channels.

DC Intvl: determines how often the channel data is sent up to DAQConnect. This is in data points. So, if you are taking data once a second and you put a value of 10 in this column, every 10th data point is sent. This is to allow you to take data at a faster rate than it is sent to DAQConnect. Please note that DAQConnect may limit the spacing of data points being sent to it. If you need to send high speed data such as streamed data, you should contact DAQConnect to arrange for a special plan.

Once you click **Apply** in the channel table, DAQFactory will start sending the data to DAQConnect. You can go to DAQConnect, and in the Basics menu see your tags appear. You can then start creating your screens within DAQConnect to view your data from any browser.

To change the data source you are sending data to, you can once again go to **Real-time Web -> Connect...**. You will be taken to a screen listing all your data sources on your account, highlighting the one you are currently connected to. You can then select a different data source.

Settings:

From Real-time Web -> Settings... you can adjust a couple settings for your DAQConnect connection:

Display Detailed Status: checking this box will display detailed status information about your DAQConnect connection in the Command / Alert window. This is quite useful when you are first establishing a connection. This setting is not saved with the document as its designed primarily for debugging.

Host: this setting should only be changed from www.daqconnect.com if you are using a proxy server or a dedicated DAQConnect instance. If using a proxy, set this value to the IP of the proxy. If using a dedicated instance (and no proxy), set this to the URL of your dedicated instance.

Host Name: this setting should only be changed from www.daqconnect.com if you are using a dedicated DAQConnect instance. If so, you should set this to the URL of the dedicated instance.

Control from DAQConnect:

DAQConnect can also send commands back to your DAQFactory application. To ensure that the commands are processed the way YOU want them to be, there is no automatic command processing. Instead, when a command comes in from DAQConnect, the system event "OnDCSet" is triggered, passing in the two values from DAQConnect: the key and the value.

To allow a remote user of DAQConnect to change values in your DAQFactory installation:

1) in DAQConnect, you will use the system.sendCommand() function (or other possible output functions) in an Click or other event. These functions will take a key and a value. The key is a string and often contains the name of the connector with a dot after it, such as "MyConnector.MyKey". The value can either be a string or number, and may be part of the function. Please review the DAQConnect help for more information.

2) in DAQFactory, create a Sequence called "OnDCSet". This sequence will be run whenever a new command comes in from DAQConnect. Commands from DAQConnect only come in when DAQFactory sends data to DAQConnect. So, if your channels are updating every 60 seconds, you can only possibly get commands from DAQConnect every 60 seconds. In these situations, you might consider using a Test channel or similar to ping the server more rapidly.

3) When the sequence is called, two local variables are created: "key" and "val". What these mean is largely up to your design. If, for example, you were directly controlling a digital output, "key" might be the name of a channel, and val either 0 or 1. In this case, you might use this script:

```
if (key == "mydigout")
  mydigout = val
endif
```

If you wanted to do more channels, you'd just repeat, or use a switch/case instead.

Now some important details:

- key is always lowercase, no matter how you specify it in DAQConnect, and will NOT include the data source name. This is stripped by DAQConnect.
- val is a string, but will automatically get converted to a number when setting numeric output channels. For other uses, you may need to use strtodouble() to convert it to a number
- in the above example we did no constraining or safety checks. Unless the output being controlled doesn't

connect to anything dangerous, you should always do a safety check or ensure proper safeties are in place in hardware. Remember that the output could be controlled remotely, and you should assume it could change at any time, even if you think you are the only one with access.

- here's an example of a constrained output:

```
private nVal = strtodouble(val) // convert to a number
if (key == "myout")
  if ((nVal > 20) && (nVal < 40))
     myout = nVal
  endif
endif
```

Note that we converted the string "val" to a numeric "nVal" before doing comparisons. > and < and other operators will work on strings, but work in a different way ("2" is actually > "10" for example, even though 2 < 10)

- the OnDCSet event should run fast. Do not put loops or delay()'s in this sequence. If you need to do a loop or delay, do it in another sequence started with beginseq()

NOTE: if you lose your Internet connection or otherwise have troubles sending data to DAQConnect, the data will queue up until a connection can be reestablished. There is a hard limit of 10,000 data points per packet across all tags to ensure you don't run out of RAM. If more than 10,000 data points queue up while waiting for a connection, the packet will be cleared and all accumulated data will be lost. New data will then reaccumulate until the connection is established or the process repeats.

12.3 Email - SMTP outgoing

With DAQFactory you can compose and send emails automatically. Use this to email a daily update, or to email when an alarm occurs. Email is implemented using an internal object. That object has a number of member variables and a single member function. To send an email, you should instantiate the CEmail class, set the variables, then call send(). For example:

```
private semail = new(CEMail)
semail.strhost = "smtp.gmail.com"
semail.port = 587
semail.strUserName = "sample@gmail.com"
semail.strPassword = "xxxx"
semail.strAuthenticate = "Auto"
semail.strBody = "this is a test"
semail.strReplyAddress = "sample@gmail.com"
semail.strReplyName = "sample@gmail.com"
semail.strSubject = "test subject"
semail.strTo = "myFriend@gmail.com"
semail.strConnectionType = "STARTTLS"
semail.strSSLProtocol = "TLSv1_2"
semail.Send()
```

You can keep the object around (say in a global variable), change any of the parameters and call Send() again to send another email. If you want to send emails from multiple sequences you should instead instantiate separate objects for each sequence (i.e. the "private semail = new(CEmail)" line) so they do not conflict with each other.

Note that for backwards compatability, there is a pre-instantiated object called "Email" with the exact same variables and function. When sending emails using this object, the send() function is asynchronous and will return immediately which makes it difficult to do error handling. For this reason and the fact the internal object can only work on one email at a time, we recommend migrating to the CEmail object.

strHost: the name of your smtp host. For example: `semail.strHost = "smtp.gmail.com"`

strUserName: your smtp username

strPassword: your smtp password (optional)

strEncoding: the encoding of your message. Default is "iso-8859-1"

strTo: the email address you'd like to send the message to

strCC: the CC for the email

strSubject: the subject of your email

strFile: the full path to a file you would like to attach to the email

strReplyAddress: the email that shows up on as the reply. This is required.

strReplyName: the name that shows up as the reply. Note that many servers require this as well, even if its just the same as the Reply Address.

strBCC: the BCC for the email

strBody: the body text of the email

AutoDial: if you are on a dialup connection, set this to 1 to have DAQFactory dial your ISP when the email is sent. Defaults to 0.

Mime: set to 1 for Mime type attachment. Defaults to 0.

HTML: set to 1 for an HTML body. Defaults to 0.

Port: set to the port of your SMTP server. Defaults to 25. For SSL it is usually 465 or 587. For gmail, we used STARTTLS on 587.

strAuthenticate: can be one of three strings to determine the authentication method required for your SMTP server: "NoLogin", "AuthLogin", "LoginPlain". If you do not require a password, use the first (the default). Usually if you do require a password you will use AuthLogin, but if that does not work you can try LoginPlain or Auto. For gmail we use "Auto"

strConnectionType: can be one of four strings to determine whether SSL is used and the type of SSL: "PlainText", "SSL_TLS", "STARTTLS", "AutoUpgradeToSTARTTLS". If you do not require SSL, use PlainText (the default). If you do, which one to select will depend on the server. For gmail, we use "STARTTLS".

strSSLProtocol: can be one of five strings to determine which SSL protocol to use: "SSLv2orv3", "TLSv1", "TLSv1_1", "TLSv1_2", "DTLSv1". This setting only applies if the connection type is not PlainText. Typically you will use SSLv2orv3 for normal SSL, and TLSv1_2 for TLS. For gmail, we use "TLSv1_2".

Send(): Once you have set the desired variables, call the Send() function to actually send the email. You can send multiple emails by simply changing the desired variables and calling send again, or instantiate a new CEmail object. For example if you wanted to send yourself an email with some values every 12 hours, you could set all the initial variables once and then set the body only and call send every 12 hours. This works great under a single sequence. If you are sending emails from multiple sequences, you should definitely instantiate a new CEmail object for each sequence.

Note that Send() is now a synchronous call, meaning it will block and not return until the email is sent or there is an error. If there is an error, it will be thrown. You can catch the error using try/catch blocks (see section 5.19). If you don't catch the error, the sequence will stop, and the error displayed in the command/alert window.

12.4 Email - POP3 incoming

With DAQFactory you can also write script to retrieve emails. This could be used to allow you to change parameters in your system remotely through email. That said, it should be noted that email is not terribly secure, so you should never allow remote control of critical functions through email. Note that SSL is not supported (required by gmail). The POP3 features require more functions then variables. First the variables. You will typically set these once:

POP3.strServer: the name of your POP3 server. For example: `POP3.strServer = "mail.myISP.com"`

POP3.strUserName: your POP3 username

POP3.strPassword: your POP3 password (optional)

POP3.Port: set to the port of your POP3 server. Defaults to 110.

POP3.Timeout: the connection timeout in milliseconds. This defaults to 60000, or 60 seconds.

Once these variables are set, you can use these functions to actually retrieve your emails. In general you will want to Connect(), retrieve emails, then Disconnect().

POP3.Connect(): call this function first once your variables are set for the proper server. This function will connect to your server and retrieve the number of messages waiting. The function returns the number of messages waiting.

POP3.Disconnect(): when done, call this function to disconnect from the server.

Once you have connected and checked to see that their are messages, there four functions you can call on each individual message. Which message these functions apply is determined by the message number, which starts at 0 for the first message, and goes to 1 less than the number of messages returned by Connect().

POP3.RetrieveMessage(message number): retrieves the given message in its entirety. Use the Get...() functions listed below to retrieve the various parts of the message.

POP3.DeleteMessage(message number): deletes the specified message from the server.

POP3.RetrieveHeader(message number): retrieves just the header for the message. You can then use the Get...() functions listed below to retrieve the details. Obviously the GetMessageText() and GetBody() functions won't return anything since the body is not retrieved by this function.

POP3.RetrieveMessageSize(message number): returns the size, in bytes, of the specified message. Does not actually retrieve the message.

Once you have retrieved the message using RetrieveMessage() or RetrieveHeader() you can get the various parts of the message using these Get...() functions. They are pretty self explanatory:

POP3.GetBody()

POP3.GetCC()

POP3.GetDate()

POP3.GetFrom()

POP3.ReplyTo()

POP3.GetSubject()

POP3.GetTo()

These two functions are a little less obvious:

POP3.GetHeaderText(): gets the raw header text

POP3.GetMessageText(): gets the raw message text

POP3 Example:

Here's an example script that connects to a POP3 server, retrieves each message if the body size is less than 100,000 bytes, and deletes the message.

```
   pop3.strServer = "mail.myserver.com"
pop3.strUserName = "me@myserver.com"
pop3.strPassword = "mypassword"
private count
private size
count = pop3.Connect()
? "found "+doubletostr(count)+ " messages..."
while (count > 0)
   count--
   size = pop3.RetrieveMessageSize(count)
```

```
   if (size < 100000)
      pop3.RetrieveMessage(count)
      ? pop3.GetBody()
   endif
   pop3.DeleteMessage(count)
endwhile
pop3.Disconnect()
```

12.5 FTP

With DAQFactory you can FTP to remote servers and upload and download files. Use this to upload data files to a server to share, or even generate dynamic HTML with the File. functions and then upload them to your webserver. FTP is implemented like Email, using an object and a collection of member variables and functions. The object is CFTP and must be instantiated first. For example:

```
private sFTP = new(CFTP)
FTP.strServer = "ftp.myISP.com"
FTP.strLocalFile = "c:\myData\data.csv"
FTP.strRemoteFile = "data.csv"
...
FTP.Upload()
```

If you want to FTP to multiple sites simultaneously, or send multiple files, or send from multiple sequences, you should instantiate a new CFTP object for each communications link.

Note that there is an internal object called "FTP" that has the same variables and functions provided for backwards compatibility. Using this object is less flexible as it only supports one communications link at a time and you have to be sure its done with its upload / download before doing another file.

strServer: the name of your FTP server. For example: `sFTP.strServer = "ftp.myISP.com"`

strUserName: your FTP username

strPassword: your FTP password (optional)

strLocalFile: the full path to the file you are uploading, or the place to put the downloaded file.

strRemoteFile: the path on the remote server where you'd like the uploaded file placed, or the path to the file you'd like to download.

Port: the port to use for FTP. This defaults to 21.

Timeout: the connection timeout in milliseconds. This defaults to 3000, or 3 seconds.

BytesTransfered: during the transfer, you can read this value to determine the progress of the transfer.

BytesToTransfer: during the transfer, if available, this tells you how big the file to transfer is.

Upload(): Once you have set the desired variables, call the FTP.Upload() function to actually start the FTP Upload.

FTP.Download(): Once you have set the desired variables, call the FTP.Download() function to actually start the FTP Download.

Upload() and Download() run the FTP transfer asynchronously. This means the functions will return immediately. To determine the progress of the FTP transfer, use the BytesTransfered / BytesToTransfer variables. Note that you can only run one FTP transfer at a time on a single CFTP and will receive an error if you try to do otherwise. If you want to run multiple transfers, simply instantiate new CFTP objects. Note that some servers may limit the number of simultaneous connections.

Abort(): Stops an FTP transfer that is running.

12.6 Voice Modem / AutoDialer (TAPI)

DAQFactory supports the use of TAPI compliant modems for outgoing and incoming calls. TAPI is a generic driver provided by Windows for use with telephony devices. Once a call is established, DAQFactory can play a wave file, use the Microsoft Text to Speech engine to speak announcements, and detect voice and tone commands. This can be used to announce alarm conditions, provide acknowledgment, and telephone based remote control of your system.

TAPI is not supported in all versions of DAQFactory.

The use of these features requires a TAPI compliant modem.

Most of the TAPI functionality requires SAPI 5.1 as well. SAPI is a generic driver provided by Microsoft to implement text to speech and speech recognition and unlike TAPI is not included with the Windows installation. This driver is currently available from the Microsoft website: http://www.microsoft.com/speech/download/sdk51/ . You should download and install the Speech SDK 5.1.

All of TAPI runs asynchronously and use events to tell you when a call is received, a text to speech is complete, etc. These events are simply sequences with a particular name in your local connection. The names of the various events are described below. One important general point is that these events run in the thread of TAPI, which is the main application thread, and not their own thread. This means that any event code should be fast or your user interface will become sluggish. If you need to do any slower processing, use beginseq() to start up another sequence in its own thread.

All functions for communicating with a TAPI modem start with TAPI. With the exception of GetDeviceList() all functions run asynchronously and will always succeed. Any errors will display in the alert window and cannot be caught using try / catch. You will want to use the events described below heavily to determine what state the modem is currently in and when to execute subsequent commands. You can also use delay() to give a command time to execute, but this is less reliable.

Before you can use almost all the functions you must call Init():

```
TAPI.Init("ModemName")
```

This function takes the name of the TAPI device you wish to use. You can retrieve a list of available TAPI devices by calling GetDeviceList():

```
private string devlist = TAPI.GetDeviceList()
```

This function returns an array of strings of valid TAPI devices. Note that these devices may be TAPI compliant but aren't guaranteed to work with the DAQFactory TAPI functionallity.

You only need to call Init() once. We recommend delaying at least 1 second after Init() before calling any other TAPI functions.

Once you have initialized TAPI, you can use the modem. At this point you will need to establish a call. You can either initiate the call from DAQFactory using the Dial() function or set the modem up to answer an incoming call (or both).

Dialing out:

To dial out, use the Dial() function:

```
TAPI.Dial("555-1212")
```

This function will return immediately and the dialing will occur asynchronously. You can just put a delay statement after the Dial() to give DAQFactory time to establish the connection:

```
TAPI.Dial("555-1212")
delay(30)
TAPI.Speak("Alarm on channel 3")
TAPI.Drop()
```

However, this method is both inefficient and doesn't handle a busy or no answer condition. Instead you should use the TAPIConnect event. When a connection is established, DAQFactory will try and run a sequence called TAPIConnect in the local connection. It is here that you can put the code you wish to perform when someone actually answers your call. In addition to TAPIConnect, there is TAPIBusy which indicates that a busy signal was detected.

Accepting an incoming call:

To accept an incoming call, set the AnswerOnRing variable to the desired number of rings to answer on:

```
TAPI.AnswerOnRing = 2
```

This variable defaults to 999, which essential means never answer. To tell you when an incoming connection occurs, DAQFactory will attempt to start a sequence named "TAPIAnswer" in the Local connection. DAQFactory will also attempt to start the TAPIConnect sequence as well since a new connection is established.

Playing a wave file:

Once a connection is established, you can perform several different actions on the line. The simplest is playing a .wav file stored to disk. Use the PlayFile() function for this:

```
TAPI.PlayFile("c:\windows\media\Chimes.wav")
```

Like most everything in TAPI, the file is played asynchronously, so the function will return immediately. When the playback is complete, DAQFactory will attempt to start a sequence named TAPIPlaybackComplete. You can use this sequence to trigger the next TAPI function on the connection. Of course, since you probably know the playback length, you could just delay:

```
TAPI.PlayFile("c:\windows\media\Chimes.wav")
delay(2)
TAPI.Drop()
```

Text to Speech:

One of the most useful parts of TAPI, when combined with SAPI is the ability to do text to speech, or speech synthesis. This allows you to read strings over the phone line that are generated dynamically. So, when an alarm occurs, you could announce the alarm and the current channel value. This is all done with the Speak() function:

```
TAPI.Speak("Alarm, temperature overtemp at " + DoubleToStr(TempChannel[0]) + " degrees")
```

Again, the Speak function runs asynchronously, so DAQFactory will attempt to start a sequence named TAPISpeakComplete when it is done.

Microsoft SAPI supports a wide variety of XML commands that can be embedded in your string to control the voice. These are described on the Microsoft website, currently at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/SAPI51sr/Whitepapers/WP_XML_TTS_Tutorial.asp. If you can't find it there, search for "XML TTS Tutorial". Here is a summary of the common commands. Note that since double quotes (") are used in the XML, you will have to use single quotes (') to specify your string:

Volume: **<volume level = "80">** The specified value is a percentage of the max voice volume.

Speed: **<rate speed="5">** The specified value is relative to the default rate of the voice, with negative numbers being slower.

Pitch: **<pitch middle = "5">** The specified value is relative to the default picth of the voice, with negative numbers being lower pitch.

Emphasis: **<emph> </emph>** The text between the two tags is spoken with emphasis. What this does varies with the voice.

Spell: **<spell> </spell>** The text between the two tags is spelled out rather than pronounced.

Silence: **<silence msec = "500">** A pause with the length specified in milliseconds is generated.

For example:

```
TAPI.Speak('<volume level = "80"><rate speed = "-3"><emph>Alarm</emph> temperature overtemp on
<spell>TR392</spell>')
```

Tone recognition:

DAQFactory can recognize tones generated on the line and perform actions based on these tones. This could be used to remotely acknowledge an alarm, or even provide remote control. Tones are generated by the remote phone by pressing the appropriate key on the phone. When a tone is detected, DAQFactory will attempt to start a

sequence called TAPIDTMF. It will pass the parameter "Key" to this event which will contain the character of the tone detected. So, if you wanted to ack your alarms if the user pressed 5, you would put this in the TAPIDTMF sequence:

```
if (Key == "5")
  Alarm.AckAllAlarms()
endif
```

This sequence is called with each tone press, so to detect a series of tones, you will need to use a global or static variable to accumulate the tones pressed.

Speech recognition:

DAQFactory with the help of SAPI can also detect spoken words on the phone line. This could also be used to acknowledge alarms, or provide voice remote control. To use this feature, you will first need to tell DAQFactory what words it should look for. This should be provided as an array of strings using the SetVoiceCommands() function:

```
TAPI.SetVoiceCommands({"Yes","No","Ack"})
```

Once this function is called, if DAQFactory detects any of the words in the list, it will attempt to run a sequence called TAPISpeechRecognized. It will pass the string of the command recognized in the "Command" private variable. So, like DTMF, you could put the following code to ack your alarms in the TAPISpeechRecognized sequence:

```
if (Command == "Ack")
  Alarm.AckAllAlarms()
endif
```

Caller ID:

If you have Caller ID service and the incoming call can be identified, DAQFactory will attempt to start a sequence called TAPICallerID passing the detected phone number in "Number". So, if you only wanted to accept calls from 303-555-1212, you could put in the TAPICallerID sequence:

```
if (Number != "3035551212")
  TAPI.Drop()
endif
```

You should probably verify the exact format your Caller ID service provides its phone numbers. In this case, the numbers are stripped of punctuation and always include the area code.

Ending the call:

To end the call from DAQFactory, use the Drop() function:

```
TAPI.Drop()
```

If the call ends by other means, either by the person hanging up, or a line disconnection, DAQFactory will attempt to start a sequence called TAPIDisconnect to tell you that this occurred. It will also automatically release the call to allow you to start or accept a new call. You do not need to call Drop() from within this sequence. Note that TAPIDisconnect is only started when the call is dropped remotely and not when you call TAPI.Drop().

Note: It is very important that your TAPI event sequences execute quickly. If they are running when a new event occurs, the second event may be missed. So, for example, if your TAPIDTMF sequence is still running when the next tone arrives, that tone may be missed. If you need to perform a longer action, you should probably start another sequence to perform that action, thus allowing the TAPI event sequence to complete and detect a new event.

12.7 Connecting to a remote copy of DAQFactory

If you have DAQFactory running on multiple computers connected by a network or modem connection, you can

connect them together and share data in real time between them. You can even add or remove channels, sequences, and other objects over the network, allowing you to make changes to remote systems. All of this is handled through connections. You have seen the Local connection, which contains the channels, sequences and other objects that run locally. Remote connections look the exact same, with channels, sequences and the rest of the objects of the Local connection. When you establish the connection with a remote copy of DAQFactory, the channels, sequences and other objects are download to your computer. You do not need to recreate the channel list of the remote computer and can very quickly start displaying data.

In order to use this feature, the DAQFactory instance you are connecting too must have Broadcasting enabled. This is disabled by default. You can enable it through File-Document Settings from the main menu. We strongly recommend implementing a password for both the slim and full streams, also from this menu. Data between DAQFactory instances is encrypted and the password is used for part of the encryption key. If it is not included, then a default encryption key is used, which is the same across all DAQFactory installations that do not have passwords set.

Creating a new Connection

When you open a new document in DAQFactory, a connection called **Local** is created which "connects" internally within DAQFactory to the data acquisition being done in DAQFactory. In many applications this is all you will have to worry about connections. If you want to view data in real time coming from a remote copy of DAQFactory then you will need to create new connections. To create a new connection, simply right click on the label **CONNECTIONS:** in the Workspace and select **Add Connection**. The New Connection dialog box will appear:

Connection Name: Enter in a name for your connection. Like all names in DAQFactory, the name must start with a letter and only contain letters, numbers or the underscore. Make sure you don't use a connection or other object name that you have used before.

Address: Enter in the IP address of the computer that you would like to connect to.

Password: If the remote DAQFactory has passwords assigned to it (through **File-Document Settings** from the main menu), you will need to provide the correct password or the connection will not establish.

Full Stream: There are two data streams that are sent out of DAQFactory, the full data stream and the slim data stream. The full data stream outputs all the data, and allows you to control outputs. A slim data stream will only output data for channels that have the Broadcast field checked (see The Channel Table). In this mode this is all you will see. You cannot set any outputs. This setting is designed for when you have other people connecting to your system and you do not want them to accidentally change any of your process parameters.

Once you are finished, press the **OK** button and the Connection will be created. Please note, though, that these parameters cannot be changed once the Connection is created. If you need to change any of them, you will need to delete the connection and create a new one.

To check to see if your new connection was successful, click on the name of the new connection in the Workspace under **CONNECTIONS:** This will bring up the connection view. Below all the parameters and buttons is displayed the connection status. This will display either **Connected** or **NOT Connected**. If the connection is established, then some statistics are also displayed below the **Connected** label.

Once a new connection is created and the connection established, DAQFactory will download its channels and other objects from the remote site. You can then use the Channels as you would on a local connection, displaying their values, or setting outputs. You can also view sequences and other objects in the workspace, but you cannot make any edits over the connection. The same goes for the channels. Sequences and other objects only appear under Full Stream.

If for some reason you get disconnected from a remote copy of DAQFactory, the icon next to the connection will have a red line through it. The channels and other objects are still maintained in memory, along with the history for all the channels, so you can still work with the data. Of course no new data will arrive until the connection is reestablished. While the connection is lost, you cannot change any objects, nor can you set outputs or start / stop sequences or other objects.

Since DAQFactory can have many different connections, and it is easily possible that two connections could have some channel or other objects with the same name, you will need to specify which connection to use when specifying channels in any expression To do this, simply put the connection name in front of the channel name with a period in between: `ConnectionName.ChannelName`

Typically, most of your data will come from one connection. Because of this, and to keep the expressions easier to read, you can mark one connection as the default connection. When referencing a channel or other object on the

default connection you do not need to specify the connection in front of the object name. You can change the default connection at will by opening up the connection view for the desired new default connection and selecting **Default?** Just be careful, all references to objects in expressions without a specified connection will now refer to different data. This can actually be a benefit if used correctly, but can really throw you if not.

The connection view has an additional parameter called **Max History Length**. Since memory on one machine may be less then another and history is determined by the master machine, you can set a max history length to limit the history length of any channel on the connection to the given value.

There are several functions to allow you to programmatically manipulate connections:

System.Connection.Add(Name,Address,[Password],[Fullstream]): the last two parameters are optional and default to blank and 0. Creates a new connection (provided one with the same name does not already exist) with the given name, address, password and fullstream setting (0 or 1)

System.Connection.Reset(Name): Resets the given connection. Closes the connection then attempts to reconnect. This does the same thing as clicking the Reset Connection button in the connection view.

System.Connection.Delete(Name): deletes the named connection if it exists. You can't delete the Local or V connection. If you want to create a runtime that connects to a computer with a varying IP address, call Delete followed by Add with the new IP address and same connection name.

System.Connection.Default(Name): sets the default connection to the given Name. For example:
```
System.Connection.Default("Local")
```

12.8 Internal Web Server (deprecated)

The built in web server is considered deprecated. The security issues of hosting a web server on the same computer as your SCADA system should be enough to keep you from doing so with ANY product. There are a number of other options, described in this chapter, that will provide the same, or better features in a more secure manner, so to ensure a secure DAQFactory we have removed easy access to the built in web server. For customers that still need to use the web server in existing applications, you can access the configuration by typing:

system.editWebSettings()

in the Command/Alert window. However:

1) the web server should only be used for internal, isolated (non-Internet connected) networks
2) the web server should not be used for ANY new applications
3) the web server will most likely be completely removed from the DAQFactory product in the coming releases, so you should migrate to one of the other features described in this chapter as soon as possible.

12.10 Using DDE to transfer data to other windows applications

DAQFactory has a built in DDE server that allows you to share your data in real-time with your favorite DDE client such as Excel. Within Excel for example you can use this to easily populate cells with values that update in real-time. Examples Excel spreadsheets filling cells with data are provided in the Samples subdirectory.

To use the DDE server you must enable it. Since there is some overhead involved in the server, the default is to have the server disabled. To enable the server go **File-Document Settings...** from the main menu and check the **Enable DDE** checkbox near the bottom. You will need to restart DAQFactory for the change to take effect.

Once the server is active, it is pretty much transparent. The server will automatically broadcast almost any Channel marked with the **Broadcast?** (or **Brd?**) checked to any DDE client that connects to it. The server does not broadcast high speed Stream data nor Spectral data. The server broadcasts at a maximum update rate of 10hz.

To your DDE client, the DAQFactory DDE server is called appropriately enough **`DAQFactory`**. There are two topics on the server, **`Data`** and **`Time`**. The Data topic has the data point values; the Time topic has the time of data point in seconds since 1970. Within each topic there will be an item for each channel that has received data. The items available in the Data topic are identical to the ones in the Time topic.

Excel makes an easy DDE client. To connect to the DAQFactory DDE server from Excel, you simply enter in the cell

you would like the value:

=ServerName|Topic!Item

So, if we had a channel called **MyChannel**:

=DAQFactory|Data!MyChannel or

=DAQFactory|Time!MyChannel

You can also combine this into a formula:

=(DAQFactory|Time!MyChannel) / 86400 + DATEVALUE("1/1/70")

The above formula, for example, converts the time from seconds since 1970 to days since 1900 which is how Excel stores times. You could then set the formatting of the cell to date/time and Excel would display it in a more human readable form. Just make sure you put parenthesis around the DDE reference to keep things clear.

Examples Excel spreadsheets filling cells with data are provided in the Samples subdirectory. These show macros for filling cells down with the same value for creating graphs in real-time in Excel.

12.11 Web Service Calls

You can programmatically perform a web service call to your favorite web service provider to retrieve information such as weather, stock prices, or whatever data you may need. This is only available through script and only available on the versions of DAQFactory that support networking. The command is simply:

HTTP.Get(URL, Path, [Port = 80], [AdditionalHeaders = ""], [Secure = 0], [GetHeaders = 0)

URL is the url for the server, such as www.azeotech.com. It does not include the path. Path is the path to the desired file on the server, say "/index.html". For example, to retrieve the weather for Winnecmucca, NV you would do something like:

```
global string datain
datain = HTTP.Get("www.weather.gov", "/xml/current_obs/KWMC.xml")
```

At this point, DAQFactory does not have an XML parser, so you will need to parse the response yourself. We have included a sample, HTTPGet.ctl in the samples folder that shows a simple XML parser in script to parse some of the weather data from weather.gov.

The other parameters are optional.

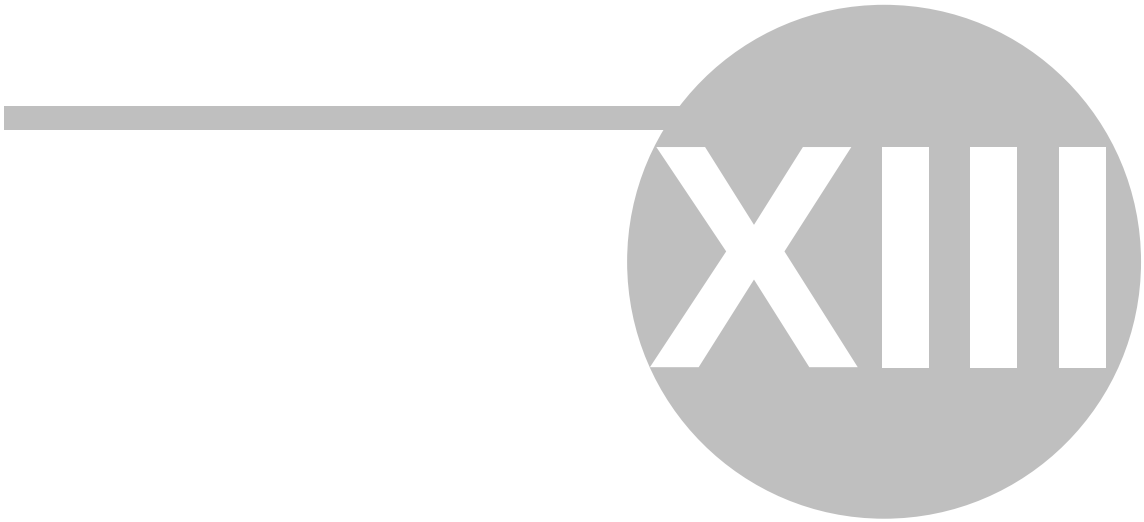
Port is the desired port to connect to on the server. HTTP is typically at port 80, while HTTPS is typically at port 443.

Additional headers is a string that is added to the headers sent to the server. This could be login information for a website with basic HTTP login.

Secure = 1 will cause the request to be done over SSL. Typically you will want to specify a port of 443 as well.

GetHeaders = 1 will cause the request to only return the headers and not the content. There are some server requests that require a GetHeaders to retrieve information about the main response before the main response is requested.

13 Analysis



13 Analysis

13.1 Analysis overview

DAQFactory provides a significant amount of data processing and analysis tools to complete your data acquisition tasks and convert your data into something useful. Since DAQFactory integrates the data analysis with the data acquisition and data display, you can perform your data analysis on your data as you are taking it.

13.2 Analysis Tools

DAQFactory's toolkit includes quite a few analysis tools. Most of these are available as functions for use in expressions. Their use in expressions is discussed in the expression chapter. All the tools also have a dialog box for setting the parameters for performing the analysis. The analysis tools are available either from the **Analysis** submenu of the main menu, or from a graph's popup menu. If a graph popup is used to get to these tools, the expressions will be filled in if there is only one trace or markers on the graph. The range will also be filled in if both markers are displayed on the graph.

All the analysis tools provide a range option. This essentially subsets the expression after the expression is evaluated. If the range is left blank, the entire range is used. If the range has time values then the time of the result of the expression is used to determine the range. If the range has normal constants, then the data point number is used. This is identical to [] style subsetting in expressions.

The results of analysis tools are placed in the Results window. If the Results window is not visible, select **View - Results** from the main menu. Many analysis tools also generate virtual channels to store their results. In many cases, these channels must be specified for the calculation to be performed. Graphs can also be created from the results of most of the functions. The graphs are created on the Scratch page and can be moved to any other page as you see fit.

13.3 Capturing Channels and Graphs

Since live data is constantly updating, it can sometimes be difficult to analyze. DAQFactory provides several ways to capture live data into virtual channels so you can analyze parts of your data easier.

Capturing Channels:

If you just want to save the history of a single channel, you can capture it to a virtual channel. Use this when something interesting occurs and you want to take a further look without worrying about running out of history length. To capture a channel, simply click on the channel name in the workspace to display the channel view. Click on the button labeled **Capture** at the top of the view. Next enter in a virtual channel name to capture the history to and press **OK**. A new static virtual channel with the history of the original channel will be created. The original channel will continue to take data, unaffected by the capture.

Capturing Graphs:

Very often while graphing live data, you will see something interesting and want to spend a little time investigating it. To avoid losing the interesting plot as new data comes in you can always freeze the graph, but eventually the history length of the traces' channels will run out and you will be left with an empty graph. By capturing a graph you can spend as long as you want working with the graph. When you capture a graph, each expression that makes up the traces of the graph are evaluated and stored in virtual channels. A new graph can then be created plotting the virtual channels instead of the real channels and expressions.

To capture a graph, simply right click on the desired graph and select **Capture** from the graph's popup. A capture dialog box will appear which allows you to select which traces to capture and assign virtual channel names to the traces.

Create Graph: If checked, a new graph is created on the scratch page. All the parameters of the new graph will be identical to the original graph, except the expressions for the traces will be replaced with the virtual channel

names the results of the original trace expressions were captured to.

Trace Table: This table will display all the traces and allows you to select which traces get captured and assign virtual channel names to the results of the capture.

Trace: The Y Expression for the trace.

Capture: If checked, then the trace will be captured to a virtual channel.

New Virtual Channel Name: Enter the desired virtual channel name to use for this trace. The default is the trace expression converted to a valid channel name (by converting all invalid values to an underscore).

Select All / Clear All: Either selects or clears the Capture? column.

Please note that error bar, scaling, and annotation Expressions are not captured, though they are copied to the new graph if the Create Graph option is selected.

13.4 General Statistics

Calculates some general statistics on the result of the expression. This includes the mean, standard deviation, min, max and variance. The results of this calculation are displayed in the Results sheet.

13.5 Curve Fitting

Using linear and non-linear least squares fitting routines, this tool will attempt to fit a chosen function to your data. The curve fitting dialog has two sheets, the first has the main parameters for the curve fit, the second allows you to set initial parameters, allowable ranges, and weighting.

Main Tab:

Function: Select the desired function to fit from the list. Be aware that some functions will not work with negative X values.

Order: If you selected a polynomial fit, enter the desired order of the polynomial you would like to fit.

Y Expression / X Expression: The curve fit routine requires both X and Y data values to fit to. These Expressions can be identical to the Expressions used to create a trace of your data in a graph. Entering in `Time`, or leaving the X Expression blank is interpreted the same way it is in generating graphs.

Coefficients: Enter your desired V channel name to store the coefficients. The resulting V channel will be an array with enough elements to hold all the coefficients of your chosen function. The coefficients are stored in order from left to right as you read the function.

Fitted Curve: Enter a V channel name to store a set of Y values along the fitted curve for the X values of the X Expression.

Residuals: Enter a V channel name to store the residuals (the difference between the fitted curve and the actual Y data).

Goodness of Fit: Enter a V channel name to store the Goodness of Fit or Chi-squared value. This is a single number.

Figure of Merit: Two different methods of determining how good the current coefficients are for the fitted curve. Linear least squares is the default.

Method: Determines the method to move across the goodness of fit space. Both is the best: Downhill simplex is used first, then the method is switched to Levenberg-Marquardt once the fit gets close.

Local Min Breakout? If you think that the curve fitting routine is finding a local minimum in fit space instead of the real minimum you can either try and change the initial values or check this parameter. When checked, a

special algorithm is used to try and break out of any local minimums. While effective, this method requires a bit more processor time to complete the fit.

If curve fitting was initiated from the graph popup, you can select **Create Graph** to create a new graph on the scratch page with the results of the curve fit. You can also select **Add to Graph** to add the result of the curve fit to the current graph.

Coefficients Tab:

Weights: Enter an expression that evaluates to an array of weight values corresponding to the result of the Y Expression. The resulting array should be the same size as the result of the Y expression array. If no weight expression is given, all values will be given an equal weighting of 1.

Coefficient table: This table allows you to specify starting values and other parameters relating to your coefficients. Setting starting values will make your curve fit faster and less likely to end up in a local minimum. Here is a list of the columns and there meanings:

Coefficient: A read only column displaying the corresponding coefficient for the row. The formula is displayed above the table for your reference.

Initial Value: The starting values to be used for the curve fit. The default is 1 for all parameters. Make sure you don't specify an illegal value, such as a negative coefficient inside of a log function.

Hold: If checked, then this coefficient will be held at its initial value and not changed during the curve fit.

Lower / Upper Bound: Use these two columns to constrain the possible values for a coefficient. If left empty then there are no constraints.

13.6 Correlation and convolution

These two tools have the same parameters but perform different functions.

1st Expression / 2nd Expression: Enter the two expressions to perform the correlation or convolution function on.

Type: Determines whether an FFT will be performed to aid in the function. Typically this is not necessary.

Output Channel: Enter a V channel where the results will be stored.

Create Graph: If checked, a new graph will be created on the scratch page with the results displayed. Correlations typically return 1 dimensional values (the correlation coefficient) and therefore cannot be graphed.

13.7 Histogram

Creates a histogram out from an expression. Bins can come from an expression or be manually generated. This function outputs both X and Y values.

Expression: Enter an expression to create the histogram out of.

Bins Expression: Enter an expression that evaluates to a sorted array of bin end points. Make sure the resulting array is sorted. Typically this is used for variable size bins and you will enter the bins expression as a constant array (i.e. {1,4,7,14,20}) or from the results of a previous histogram.

Bins Manual: Generates the bin end points from three parameters, the starting point, the ending point and the size of each bin. The bins are of uniform size.

Output Histogram: Enter a V channel name to store the results of the histogram.

Output Bins: Enter a V channel name to store the bins. A plot of the histogram channel vs. the bins channel yields the histogram.

Create Graph: If checked, a graph displaying the histogram channel vs the bins channel will be displayed on the scratch page.

13.8 Interpolation

Use this tool to take a set of X-Y values and interpolate them to a different set of X values.

X / Y Expression: Enter an expression for both the X and Y values. This works just like a graph. You can enter `Time` or leave the **X Expression** blank to use the time or point number associated with the Y values.

Type: Determines which function should be used to perform the interpolation.

Interpolate to (X): Enter an expression that results in X values that the X-Y values will be interpolated to.

Interpolated Y: Enter a V channel name to store the results of the interpolation.

X Values: Enter a V channel name to store the result of the Interpolate To expression.

Create Graph: If checked, creates a new graph on the scratch page with the interpolated values displayed.

13.9 Percentile

Creates a percentile graph from the given expression. Like a histogram, this function outputs both X and Y values.

Expression: Enter an expression to perform the percentile calculation on.

Values: Enter a V channel to store the Y values of the result of the percentile calculation.

Percentile: Enter a V channel to store the X values of the result. A plot of Values vs. Percentile yields the percentile graph.

Create Graph: Creates a new graph on the scratch value of values channel vs percentile channel.

13.10 FFT

Performs a fast-fourier transform of the given expression.

Expression: Enter an expression to perform the FFT on.

FFT Type: Determines how the resulting imaginary array is converted into real numbers.

Windowing: Determines which windowing function will be applied to the result of the expression before performing the FFT.

Output Channel: Enter a V channel name where the results of the FFT will be stored.

Create Graph: If checked, a graph will be created on the scratch page displaying the results of the FFT.

14 Other Features

14 Other Features

14.1 Startup Flags

In addition to being able to specify the file to open in the command line when starting DAQFactory, there are a couple other predefined flags. These are all case:

-R: forces DAQFactory into Runtime mode

-F: forces DAQFactory into full screen mode

-runAsExpress: forces DAQFactory to run in Express mode

-remoteBrowserPort: enables remote debugging of your browser components. See the section on the Browser component for more detail. Unlike the others, this one is case sensitive.

You can create your own flags as well. To retrieve your flags in script, call system.GetFlags():

System.GetFlags(): returns an array of strings containing all the flags entered in the command line, with the exception of a document name, the -R and the -F flags, if specified. So, if you did: DAQFactory.exe MyDocument -F -A -DoSomething to start DAQFactory, and then called System.GetFlags(), you'd get: {"A","DoSomething"}

14.2 Preferences

There are several adjustments that can be made to the DAQFactory environment. These are all available from **File-Preferences...** from the main menu. These preferences are saved in the registry and apply to DAQFactory as a whole, independent of which document may be loaded. For the colors, you will need to restart DAQFactory for the changes to take effect.

Disable 2nd Instance Check: Normally, DAQFactory will look to see if its already running and warn you so you don't accidently end up with two instances trying to communicate with your hardware and interfering with each other. However, there are many cases where two instances is actually useful, including when you want to copy pages from one document to another. If you do this a lot, you can disable the check for a second instance of DAQFactory here.

Table Font and Font Size: On some systems, the default font for DAQFactory's tables does not appear correctly. If this occurs on your system, you can select a different font here to make the tables appear properly. This also affects the watch. You must restart for the changes to take effect.

Flash Window on Alert? If checked, the DAQFactory window will flash whenever an alert arrives. If not, only the alert area of the status bar will flash.

Show all Alerts in Alert box only: This is identical to the option presented when an alert is displayed and eliminates any alert popups.

Sequence Editor Font and Size: This setting allows you to change the sequence editor font and size. This is required for some languages to ensure that the editor font supports characters in your language. We strongly recommend using a mono-spaced font. This also affects the command / alert window. You must restart for the changes to take effect.

Expression / Sequence editor colors: This gives you the ability to change the colors used to highlight certain types of entries in expression boxes and the sequence/event editors. It has no effect on the execution of your documents, just how the editors look. You must restart for the changes to take effect.

Guru Style: This is a special editor color settings that our developers prefer. Perhaps we are just showing our age, but we prefer white text with a black background, with other colors for comments, strings, etc. This mode also

does not display sequence line numbers since these are displayed in the status bar as well. If you want to be like the DAQFactory Gurus or you just like text on a black background like we do, then try Guru Style. You must restart for the changes to take effect.

Expression / Sequence editor background colors:

You must restart for these changes to take effect:

Default: this is the default background color before any syntax coloring is applied. This is normally white, and is changed by Syntax Coloring if enabled for expression edit boxes.

Syntax Coloring: Syntax coloring displays the background of an expression edit box in a different color if the entered expression results in a valid value. A different color is displayed for a numeric result, string result, or invalid result. Syntax coloring actually evaluates the expression. When working with large histories, spectral or image data this can sometimes result in slow response from DAQFactory as it does this. To avoid this, simply disable the syntax coloring. Sometimes syntax coloring will display invalid even when there are no problems. For example, if you are entering an expression that includes a channel or variable that hasn't been initialized, the syntax coloring will show invalid, but once the channel or variable is initialized, the expression will be perfectly valid.

14.3 Alerts

When an error occurs in DAQFactory, an alert is generated. Alerts appear in the docking window at the bottom of the screen labelled Command / Alert. This window maintains a list of alerts along with the command line.

Alerts are programmatically available with history like a channel for use in sequences or expressions. `strAlert[0]` will retrieve the most recent alert message. "strAlert" has a 100 entry history. Note that even though your commands appear in the command/alert window, only alert messages are saved in this history.

Alerts are not automatically logged. You must specify them in a logging set by hitting Manual and entering “Alert”. Alerts are strings so can’t be used in binary mode logging sets.

Note: to avoid an overflow of alerts, only one alert of a particular type is displayed in any given second.

Alert Preferences:

There are two preferences relating to alerts. These are available from the preferences window by select **File - Preferences** from the main menu.

Flash Window on Alert? If checked, the DAQFactory window will flash whenever an alert arrives. If not, only the alert area of the status bar will flash.

Show all Alerts in Alert box only: This is identical to the option presented when an alert is displayed and eliminates any alert popups.

14.4 Quick Notes

Quick notes allows you to enter in time stamped notes. The time is recorded at the moment you start the command, so you can take all the time you need to enter the note. To create a quick note, simply select **Quick - Note...** A dialog box will appear with a large editing area where you can enter your note. Press **Enter** at the end of each line. Click **OK** when done. The note will appear in the Notes window if displayed. To display the notes window, select **View - Notes** from the main menu.

Notes are accessible with history from sequences and expressions like a channel. `strNote[0]` will return the most recent note entered. "strNote" has a 100 entry history.

Notes are not automatically logged. You must specify them in a logging set by hitting Manual and entering “Note”. Notes are strings so can’t be used in binary mode logging sets.

You can programmatically display the notes dialog by calling System.NoteDialog(). This function must be called from the main application thread, so only really works with the Quick Sequence action of many components.

14.5 Customizing

You can customize the menus, toolbars, and keyboard accelerators in DAQFactory Control to fit the way you work. To view a list of the current keyboard accelerators, select **Help - Keyboard Shortcuts...** from the main menu. A dialog box will appear listing all the shortcuts. Make sure and select your desired category, or All Commands to display all the shortcuts. You can use the print or copy icon at the top left corner of the box to print a list of the shortcuts, or copy the list to the clipboard.

To edit the shortcuts and other features of Control, select **Tools - Customize...** from the main menu. The customize dialog box that appears has 5 sheets. Here is a quick description on how to use each sheet. We suggest you experiment to achieve the results that work for you.

Commands: Use to add / remove features from your toolbars or menus. Select from the categories on the left, then click and drag a command from the list on the right and drop it in a toolbar or menu on the screen. You can also edit the toolbars and menus displayed by clicking and dragging any toolbar button to a new location, or drag it to the main part of the screen to remove it from the toolbar.

Toolbars: Allows you to select which toolbars are displayed (you can also use the View menu from the main menu). You can also create new toolbars here too. If you completely mess up your toolbars, you can press the **Reset** or **Reset All** buttons to restore the toolbar or toolbars to their default settings.

Keyboard: Use to assign keyboard shortcuts to DAQFactory commands. Select the Category and Command from the drop down and list to the left of the box. To assign a new shortcut, click in the **Press New Shortcut Key** box, then press the desired key combination. The keys you pressed will appear in the box. Press the **Assign** button to assign it to the selected command. To remove a shortcut, select the key combination in the Current Keys list and press the **Remove** button. Press **Reset All** to restore all the shortcuts to their default values.

Menu: Use this sheet to display the popup menus on the screen. Once displayed, you can edit them by going back to the Commands sheet and using the functions available there. You can also control menu animations and shadowing from this sheet.

Options: Some additional customization options.

14.6 DAQFactory Runtime

DAQFactory is also available in a runtime version. The runtime version of DAQFactory is just the Pro version of DAQFactory without the editing capabilities. There is no menu, no workspace, no watch or other docking windows, and no status bar. You can't change component's properties, nor use the graph popup menu (or any other menu for that matter). What the runtime will do is load your DAQFactory document and display all your process screens. You can take data, display it, run sequences, change pages, log data, etc. Typically you would use the runtime version when you don't want the end user to be able to modify your settings, but would like to allow them to operate your process.

With release 5.30, DAQFactory Runtime is now a part of the same program as the development version of DAQFactory. You can now easily switch between development and runtime mode. There are three ways we foresee this being used:

1) You have a single installation with inexperienced users that should not be making document changes but once deployed you may need to occasionally make changes to your application. This also applies if you have multiple installations and put a development license on each.

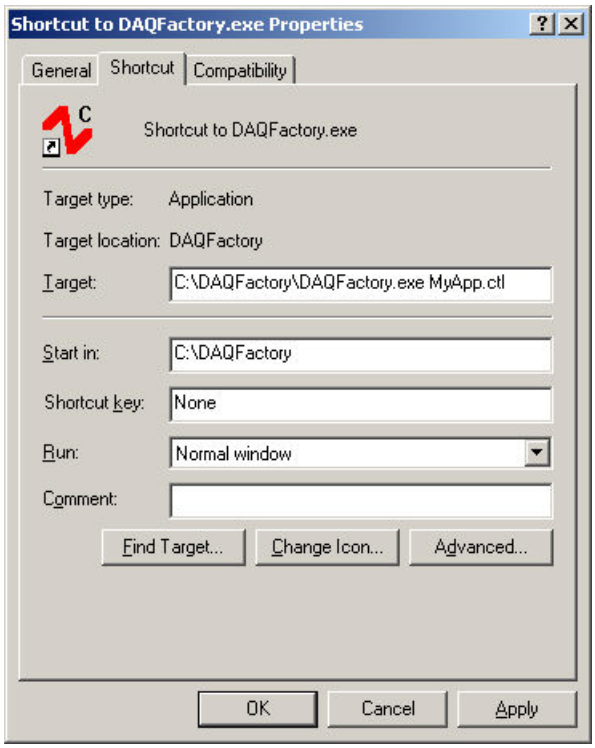
2) You have multiple installations with inexperienced users and wish to occasionally make modifications and you own the hardware key option.

3) You have multiple installations with inexperienced users and you will make modifications only in your office.

Single Installation:

For this, you will only need the development license. Put the development license on the computer that the end user will be using along with your application. Go to File - Document Settings... and put a password on your document to prevent the end user from switching to development mode. This will also cause DAQFactory to load the document in runtime mode. Finally, create a shortcut to DAQFactory.exe and add your document name in the target. To do this, right click on the shortcut and select properties. Then in the section under Target, add a space and your

document name after DAQFactory.exe:



Now, when DAQFactory loads, it will run your specified document in runtime mode. When you wish to make modifications to the document, you can click on the system menu (the DAQFactory icon at the top left corner of the window) and select "Switch to Development". You will be prompted for the password before switching. Then you can modify your document at will, making sure to save you changes, and then select File - Switch to Runtime mode to revert back. To force a development license into runtime mode at startup, use the -R startup flag as described in the section on startup flags.

Multiple Installations with Hardware key

If you expect to make multiple runtime installations and need to be able to occasionally modify the documents on-site you should consider the hardware key option. The hardware key is a thumb sized USB key that contains your development license. It is included with DAQFactory Developer and available as an option for the other versions of DAQFactory. On the installation computers you would install DAQFactory and license it only with a runtime license. Install your document and create the shortcut as described above. Install the hardware key driver as explained in the section of the hardware key option. At this point, your system should run properly in runtime mode. When you wish to make modifications to the document, insert the hardware key, then click on the system menu (the DAQFactory icon at the top left corner of the window) and select "Switch to Development". If you put a password in, you will be prompted for the password before switching. Then you can modify your document at will, making sure to save you changes. When you are done, you can select File - Switch to Runtime mode to revert back, or simply pull the hardware key out and the system will automatically switch back to Runtime mode. Please note that with a fresh DAQFactory installation, the end user will be able to switch to development mode for the first 25 days (the default trial period) if a password is not required.

Multiple Installations without Hardware key

If you will only be making document changes in your office, then you can simply install DAQFactory on your customer sites as explained above, creating the appropriate shortcut, password, license them for runtime and you are done. On your office computer, you would keep your development license where you could modify your DAQFactory documents. When complete, simply save your document and install it on the remote system over top of the existing file. Then restart DAQFactory using your shortcut and the new document will load into runtime. Since there is no development license on the customer sites, the customer will be unable to switch to development mode. Please note that with a fresh DAQFactory installation, the end user will be able to switch to development mode for the first 25 days (the default trial period) if a password is not required.

DAQFactory in Runtime mode does have small menu available for a few options. This menu is accessed by clicking

on the DAQFactory icon at the top left corner of the window.

Full Screen: This will put the runtime in full screen mode. You'll have to hit Alt-Space to open the menu up to switch out of full screen mode. Use Document Settings under File when creating your document to set the document in Full Screen mode on load. The F4 key can also be used to switch in and out of full screen mode.

On Screen Keyboard: This will display the on screen keyboard for touch screen applications.

The rest of the options are for licensing DAQFactory and are discussed in the section on licensing.

Here are a few pointers and extra features for runtimes:

• When designed runtime documents, remember that DAQFactory features accessed through menus are not accessible. Examples include changing pages (if running on a touch-screen), printing pages, and exiting the program. Use component actions for these. One exception is graph scaling and zooming. A user of DAQFactory Runtime can still right click on a graph and perform the zooming functions.

• You can set the size of a runtime document's window by selecting **File - Document Settings...** from the main menu. This information gets saved with the document. Note that this is the size of the entire window including the title bar.

• To have DAQFactory automatically start in Runtime mode, you must have a document editing password on your document.

• Runtime documents ignore the Sequence Loop Check setting.

Make sure and install the necessary device drivers into the runtime directory when installing the runtime.

Note: Documents created with DAQFactory Express and trial versions of DAQFactory cannot be used with DAQFactory Runtime.

14.8 User defined time reference

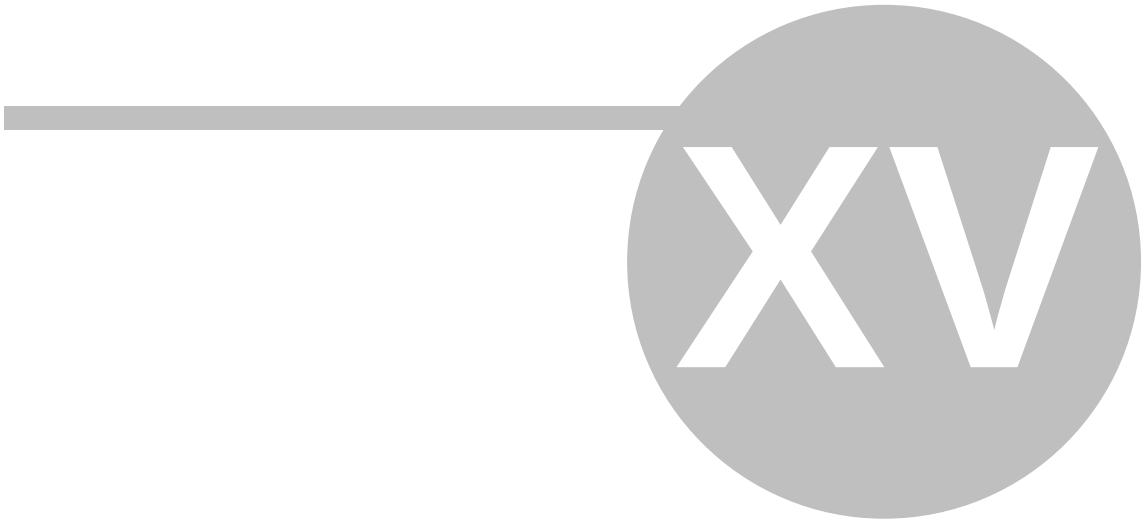
DAQFactory internally uses the high precision clock to measure time in sub microsecond precision. However, since computers often use lower quality crystals, this time can tend to drift over time. Many users use IRIG, GPS, or stable time source cards to solve this problem. You can write a simple C driver that provides the current time to DAQFactory. This driver can then pull the time from the IRIG, GPS or other hardware for a much more accurate and stable time. DAQFactory will then use this time for everything it does internally and externally. The use of the fixed hardware then becomes completely transparent to the user.

To create a time DLL, create a DLL called UserTime.dll in any programming language that exposes the following function:

double GetTime()

This function will get called whenever DAQFactory needs time. You should return the current time in seconds since 1970, in as high a precision as you can (ideally to the microsecond). A double can only store time precise to the microsecond, so this is the best precision that can be achieved.

15 Extending DAQFactory



15 Extending DAQFactory

15.1 Calling an External DLL

15.1.1 External DLL Overview

DAQFactory has the ability to call functions from external DLLs. These DLLs could be libraries provided by a device manufacturer, or simply a DLL with some C functions you created. Either way, DAQFactory can load the DLL and present the function as any other function in DAQFactory.

Note: calling of external DLLs, while not terribly difficult, is a bit of an advanced topic. Because the DLLs are loaded dynamically, there is no type checking which can result in a DAQFactory crash if you incorrectly specify the function prototypes. Because of this, it is strongly recommended that you save often to avoid losing any work, both in DAQFactory and your other open applications as an improper call could crash your computer as well. Of course once you correctly specify the prototype you should not have any problems.

15.1.2 Loading the DLL and declaring the functions

To use an external DLL in DAQFactory you must first specify the DLL file name and the various function prototypes you wish to use. You then assign each function prototype to a DAQFactory function name which you will use in the rest of DAQFactory. Typically you will want to do all this in an auto-start sequence so the DLLs are loaded immediately. Loading the DLL and specifying the prototypes is all handled by the **extern** function:

extern(DLL Name, Prototype, DAQFactory Function Name, Call Type, [Tip])

DLL Name: The name of the DLL you wish to load. This can just be the name of the DLL, in which case the default windows DLL search path is used, or you can fully specify the DLL path.

Prototype: The desired function prototype you wish to use. You can use multiple functions from a single DLL by simply calling **extern** multiple times. The proper format of the prototype is described in the next section.

DAQFactory Function Name: The name assigned to external function for use within DAQFactory.

Call Type: Also known as the calling convention. How the parameters are passed to the external function. This can either be "stdcall" or "cdecl".

Tip: The tool-tip displayed when you type in the function name in a sequence. If not specified, the prototype is displayed.

Here's an example of loading the ListAll function from the LabJackUD DLL:

```
extern("labjackud.dll","long ListAll(long, long, long[1], long[128],long[128],double
[128])","LJListAll","stdcall")
```

And calling the new function from within DAQFactory:

```
global numfound
global sn
global id
global address
global err
err = LJListAll(9,1,@numfound,@sn,@id,@address)
```

Note that the DAQFactory function name does not have to be the same as the DLL function name.

This example is explained in detail in the next sections.

Note: Do not use the extern function to load .lib files. .lib files are used by linkers to identify entry points and do not actually contain code that DAQFactory can use.

15.1.3 The Function Prototype

The second parameter of the extern function is the function prototype of the function you wish to import from a DLL. As you can see by this example, the function prototype looks a bit like a C function prototype:

```
long ListAll(long, long, long[1], long[128],long[128],double[128])
```

First there is the return type, then the name of the function, then the arguments provided in parenthesis. Although the data types may look familier, they require a little explaining. The following are valid data types:

ubyte or **uchar** - an 8 bit unsigned integer
byte or **char** - an 8 bit signed integer
uword or **ushort** or **unsigned short** - a 16 bit unsigned integer
word or **short** - a 16 bit signed integer
ulong or **unsigned long** - a 32 bit unsigned integer
long or **dword** - a 32 bit signed integer
float - a 32 bit floating point value
double - a 64 bit floating point value
string - a string of characters
void - used only as a return value, indicates nothing is returned.

They are not case sensitive. In addition, there are pointer versions of each, except void. To specify a pointer in the prototype, use array notation:

long[12] - a pointer to an array of 12 long integer.

For a simple pointer, use [1]:

long[1] - a pointer to a long integer.

A **string** carries through to a string value in DAQFactory. If you simply specify **string**, then you are specifying a constant string which cannot be changed by the external function. This is the same as the C **const char***. If you specify a string array, **string[x]**, then you are specifying a pointer to a buffer of characters of x length. Make sure to include a byte for the NULL at the end of the string. A NULL will automatically be placed at the last space of the buffer no matter what your external function does. So:

string - a pointer to a const null terminated string (**const char***)

string[10] - a pointer to a buffer of 10 bytes that will be translated to a string by DAQFactory on completion (**char***).

string[x] and **byte[x]** are essential the same thing as far as your external DLL is concerned as both pass a pointer to a buffer of characters. The difference is how they are treated by DAQFactory. A **string[x]** is converted into a single string on completion, while **byte[x]** is converted into an array of numbers:

```
extern("labjackud.dll","void ErrorToString(long, string[256])","ErrToString","stdcall")

global strError
ErrToString(1001,@strError)
```

strError now contains a single string with the error.

```
extern("labjackud.dll","void ErrorToString(long, byte[256])","ErrToString","stdcall")

global Error
ErrToString(1001,@Error)
```

Error now contains an array of 256 values. So if in the first example, strError was "ABC", Error would be {65,66,67}.

Note: DAQFactory has a general 20 argument limit which applies to external DLL functions as well.

Note: If you call extern with the same DLL and the same prototype function name but with different arguments, the old prototype is removed from the system and replaced by your new prototype.

15.1.4 Calling the External Function

Now that you have the external function declared in DAQFactory using extern:

```
extern("labjackud.dll","long ListAll(long, long, long[1], long[128],long[128],double
[128]","LJListAll","stdcall")
```

you can now call the function. Calling the function is as simple as calling any other function in DAQFactory. In fact, if you do not have any pointers in your external function, then it is identical. In this example and in many real cases, however, you have to provide pointers. To specify a pointer, you should put the at symbol (@) in front of the name of the variable you wish to point to in the function call. For example:

```
global numfound
global sn
global id
global address
global err
err = ListAll(9,1,@numfound,@sn,@id,@address)
```

As you can see, we declared several global variables, and then pointed to them in the function call to LJListAll by putting their names in quotes in the function call. If you've not guessed it already, the only thing you can provide a pointer to is a variable. This can be a global or a private variable. It cannot be a channel or system variable. Also you cannot subset the variable, so @numfound[0] is invalid.

As you may be able to tell also from the above example, you do not need to preallocate your arrays. DAQFactory will take care of this automatically for you. The same goes for strings:

```
extern("labjackud.dll","void ErrorToString(long, string[256])","ErrToString","stdcall")

global strError
ErrorToString(1001,@strError)
```

Notice how we didn't predefine strError to 256 characters.

To pass a null pointer, simply use the NULL constant:

```
ErrorToString(1001,NULL)
```

Of course you wouldn't want to pass a NULL to this particular function since it would do nothing, but you get the idea.

15.1.5 Allocating memory

In the DLL prototypes and functions we've described so far, the pointer is used only within a single function call and you include [] notation to indicate the size of the buffer and pass a variable in when you call the function. Internally, DAQFactory creates the buffer of the given size, fills it with whatever happens to be in the variable, if anything, calls the function, then fills the variable with the contents of the buffer after the function returns. The memory is then deallocated, so the DLL can no longer use that pointer without crashing the system.

Some externally DLL functions, however, require passing a pointer to a buffer that the DLL retains and then uses for subsequent functions, usually background streaming functions. For this situation, DAQFactory has a DMemory class which allows you to allocate and deallocate memory, and access that memory from script. DMemory is an internal class that you need to instantiate every time you need to allocate a chunk of memory:

global myMemory = new(DMemory)

Once you've created the object, you can allocate memory by calling its alloc function:

myMemory.alloc(65535)

passing in the number of bytes you'd like to allocate. When you create your prototype for your DLL call, you should specify that you want an unsigned long for the data type for the parameter for the buffer. This is because pointers in 32 bit memory maps are stored as 32 bit numbers. Do not use [] notation in this case. So a C function like this:

char myDLLFunction(short *buffer, unsigned long size);

would get a prototype like this for extern():

"char myDLLFunction(unsigned long, unsigned long)"

Remember, this is only for when the DLL is going to keep the pointer internally. In most cases, it won't do this and you can use the standard [] prototype notation described in 15.1.3.

When you then call the function, use the GetPointer() member function of your object to get the pointer and pass it:

private result = myDLLFunction(myMemory.GetPointer(), 65535)

Finally, if you want to get the values in the buffer in a form DAQFactory can use, you'll want to use one of the GetAs functions of the object. At present there are 6:

GetAsByte()
GetAsUByte()
GetAsWord()
GetAsUWord()
GetAsRWord()
GetAsURWord()

They function essentially the same, except for how they combine the bytes in the buffer to create actual values. In our example, since its a short *, we would use one of the AsWord functions. Which one depends on the byte ordering, but most likely it will be GetAsWord() since in most cases a DLL will use the same byte ordering as DAQFactory since they are both built for Windows.

To deallocating memory, either let the object destroy itself by removing all references to it (i.e. myMemory = 0), or call the free() function on the object:

myMemory.free()

Once the memory is freed, you can reallocate on the same object if you want by calling alloc(). Just be careful about freeing memory before your DLL is done with it.

Here are all the member functions of DMemory:

Alloc(size): allocates a buffer with the given size in bytes. Frees any memory that may have been previous allocated on this object.

Free(): explicitly frees the memory previously allocated in this object. The object will automatically free any memory it allocates when it is destroyed.

GetPointer(): returns the pointer to the memory buffer. Its up to you to properly manage this. For example, make sure you do not try and access past the end of the memory buffer.

NoAutoDelete(): if called on the object, the object will not free the memory when it is destroyed. This means that the memory must be freed elsewhere or it will be leaked. You would call this function when the DLL takes over managing the memory. Note that once you call this function, you will need to explicitly free the memory yourself using free() or in your DLL. You cannot undo this setting on an object.

GetAsByte()
GetAsUByte()
GetAsWord()
GetAsUWord()
GetAsRWord()
GetAsURWord(): these functions return an array of values of the appropriate size based on byte ordering. The word functions will return an error if you did not allocate the buffer size as a multiple of 2.

15.2 User Devices

15.2.1 User Device Overview

DAQFactory supports a wide variety of devices with the included drivers and protocols. Unfortunately it is impossible for us to support every device available, so DAQFactory provides you with the ability to create your own drivers

using sequence scripting. When combined with DAQFactory's ability to call external DLL's, you should be able to create your own DAQFactory devices for just about any device you may want. Creating a user device is rather straightforward provided you have a basic understanding of DAQFactory sequences.

Since user devices are saved outside your document, a user device can also be used as a general function library. You do not have to call device code from within a user device.

15.2.2 Creating your own device

To create your own device, go to the Device Configuration window by selecting Quick - Device Configuration from the main menu. In addition to the standard DAQFactory devices, you will see New User Device listed. Click on this and hit Select. This will open the User Device Configuration window. The first thing to do is name your device. Like all DAQFactory names, your device name needs to start with a letter and contain only letters, numbers or the underscore. You'll also need to specify a filename to save your device to. User devices are not stored with your document, but rather are written to a text file. This way, you can use your device in multiple documents and even share it with colleagues. In order for DAQFactory to recognize is as a device, you'll need to make sure it is in your DAQFactory directory and that the file name ends in ".dds".

A device consists of collection of I/O Types and functions. You can use either or both. I/O types apply to channels, while functions are accessible from sequences. Both consist of sequence script. We recommend coding your device using regular sequences so you can take advantage of the debugger and then transferring tested code into your device.

I/O Types:

Each I/O type that you create will have a sequence script associated with it. This script will take the device number, channel number and other information and return a value (for inputs) or set a value (for outputs). In the User Device Configuration window you can add new I/O Types to your device by clicking on the Add I/O type button at the bottom. A new window will appear asking about your I/O type. You'll first need to specify the I/O type name. The I/O Type is one of the few places where you can use spaces and other special characters in the name. You'll also need to provide a unique number for your I/O Type. Internally, I/O types are tracked by number. It doesn't matter what this number is, as long as it is a positive integer less than 2^32. Finally you'll need to specify whether the I/O type is an input or output, and whether the data will be numeric or a string.

Once you've added a new I/O type it will appear in the list along the left. If you click on the I/O type, the script for this type will appear to the right. Initially, of course, it will be blank. Here you can put in whatever sequence code you need to read or write your values. Remember, though, that this code executes often, so long loops or delays are not recommended. The following private variables are available to you:

DeviceNumber - the device number of the channel
Channel - the channel number of the channel
Specifier - the quick note / opc specifier for the channel
SetToValue - the value to set to (output I/O types)
strSetToValue - same as above, but the string version
OriginalValue - the pre-converted value to set to. Return this for output I/O types.
strOriginalValue - same as above, but the string version
ChannelName - the name of the channel

For input I/O types, you'll want to return the value read at the end of your script. If you don't have a value to return, for example in the case of an error, just return nothing: "return". For output I/O types you will typically want to return OriginalValue to confirm that the output was set, or again, nothing if the value could not be set.

Functions:

Device functions are typically called from sequences. To create a new function, click on the Add Function button at the bottom of the window. You will then be prompted for a function name, which like all other names in DAQFactory, must start with a letter, and contain only letters, numbers or the underscore. Like I/O types, a function is simply sequence code. The arguments passed to the function are just like a sequence used as a function and named arg0, arg1, etc. for each subsequent argument, and prefixed with 'str', i.e. strarg0, when a string is passed. You can optionally return a value as well.

There are two types of functions, Public and Private, which is selected at the top right above the code. A public function is visible from anywhere in DAQFactory and typically accessed by Device.MyFunction() notation. A private function is actually available from anywhere, but does not appear in any drop downs when entering code, so is not

really visible. Private functions are typically used solely by other functions or I/O types of this device.

OnLoad / OnUnload:

In addition to the I/O types that you add, there are two events that exist with every user device and will appear in the I/O type list from the beginning. On Load is called once before any of your I/O type code and allows you to put initialization code. This is a great place to put extern() statements and other initialization. On Unload is called when you quit DAQFactory or when you start a new document or open an existing document. Use this event for any shutdown code. Note that for every On Load called, On Unload will be called once as well. To edit these events, simply click on them in the I/O type list.

Note: you cannot remove an I/O type or function from within DAQFactory. To do this, you will need to quit DAQFactory, open the .dds file you specified and remove the I/O type or function and associated code from there.

15.2.3 Local Variables

For user devices, in addition to private variables, which are only visible in a particular I/O type or function, and global variables that are visible everywhere in DAQFactory, there are local variables. Local variables are kept with each device and accessible through the device. They are declared using the local keyword:

```
local MyVariable
```

Once declared, a local variable can be accessed from any of the I/O types or functions in the device by simply specifying its name:

```
MyVariable = 3
```

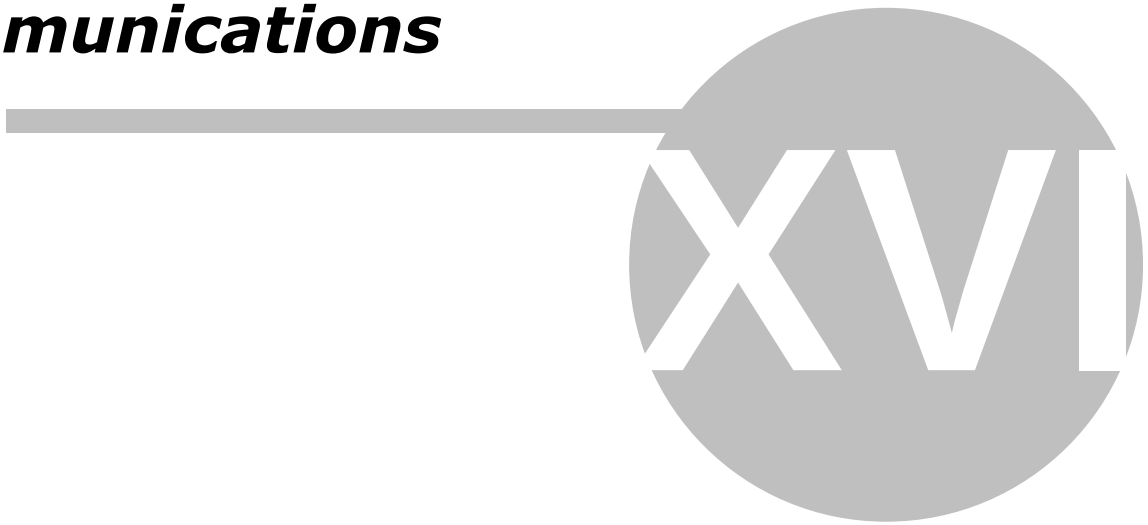
Typically you will want to declare and initialize your locals in the OnLoad function of the device.

In pretty much every case, we recommend using local variables in place of global variables in devices. You can still access these variables from elsewhere, but using them keeps your protocol code self-sufficient and not dependent on outside globals. To access a local variable from outside the protocol's I/O type and function code, use Device. notation:

```
Device.MyDevice.MyVariable = 3
```

Doing simply `MyVariable = 3` from outside the protocol's code will not work as DAQFactory will look for a global variable (or channel) named MyVariable and not the local variable. You also cannot declare a local variable from outside the protocol.

16 Serial and Ethernet Communications



16 Serial and Ethernet Communications

16.1 DAQFactory Communications Overview

A large number of devices communicate with the PC using either the serial (RS232 / 422/ 485) port, or through Ethernet. Internally, the communications is almost identical except one is over serial and the other over Ethernet. Rather than making separate drivers for every device, each including its own serial or Ethernet communications, DAQFactory allows you to create your own devices by combining the communications port, whether serial or Ethernet with a protocol. The protocol is the language that the device talks, while the port is the media that carries that communications. You can think of it like telephone communications. You can talk to someone over a landline phone or a cell phone. In both cases you are talking the same language (the protocol), but you are using a different technology (the port). To communicate with someone you must choose which language to speak and which method to contact the person with whom you'd like to speak. When communicating with an external device from your PC you do the same thing. You must choose whether to communicate over the serial port or Ethernet, and which protocol to use. Fortunately, DAQFactory makes all this very easy.

Note: While USB is technically just another technology for carrying serialized data, it is not handled the same way as normal serial port and Ethernet communications. USB has a number of extra layers that make it more difficult and thus the reason all USB devices include some sort of library for communications. If you are looking to communicate with a USB device, other than a serial to USB converter, that we do not have a driver for, you might want to consider using the User Device and external DLL features of DAQFactory to create your own device, or contact us and we can create a driver for you.

16.2 Creating a Comm Device

Before we start, some clarification of terminology for this section:

Port: either a serial (RS 232/422/485) port, or an Ethernet IP address / port.
Protocol: the language used to communicate with your device. For example, ModbusRTU.
Comm device: a DAQFactory device that you create by selecting a port and a protocol.

In order to do any sort of communications with external devices connected over the serial port or Ethernet, you have to create a comm device. A comm device is like any other device within DAQFactory and can be selected from the channel table. To create a new comm device, go to the device configuration window by selecting Quick - Device Configuration from the DAQFactory main menu. A list of devices will appear along with **New Serial (RS232/485) / Ethernet (TCP/IP) device**. Click on this and hit **Select** to add a new comm device.

First you'll need to name your device. The name, like most DAQFactory names, must begin with a letter and only contain letters, numbers, or the underscore. You'll see below the name are two lists, one with the ports, which will likely be empty, and the other with protocols. Next you'll need to create a new port. To do so, click either on **New Serial, New Ethernet Client (TCP),** or **New Ethernet Server** depending on the type of port you need. Each of these buttons will open a new window to allow you to specify the port parameters. For all port types, you will need to name your port as well. This is so you can use the port for more than one protocol.

Serial Configuration:

Connection Name: a valid DAQFactory name for your port. This can be as simple as COM1 or whatever you prefer.

Serial Port #: the comm port number. This should be a valid comm port on your computer. For example "1" for COM1. Make sure another program is not using the port. To create a placeholder port, use serial port # 0. This will create the port so you can assign it to a device, but will not initialize the port. Then when you are ready, you can change the port # and parameters from a sequence to start using it.

Baud: the baud rate for communications on the comm port. This and the next three items are the standard serial port parameters. Check the documentation for the device you are connecting to for the proper settings. If these parameters are incorrect you will either get no communications or garbled communications. The improper setting of these parameters is the most common difficulty with serial communications. The baud can be selected from a list of

standard bauds, or in the rare case that you are using a non-standard baud rate, you can manually enter your desired baud rate in the field.

Byte Size: the number of bits per byte used in communications.

Parity: the type of parity bit used in communications.

Stop Bits: the number of stop bits used in communications.

Timeout: the number of milliseconds to wait for a read / write request to complete. This will depend on the response time of your device. We recommend keeping this between 20 and 2000 milliseconds. Longer timeouts are appropriate when working with slow devices or low baud rates, but will result in a latency in a comm failure condition.

Note: if using the Timing parameter of a channel or the wait() function in the polling loop of your serial communications, make sure the timeout is set smaller than the loop time or a backlog of requests will occur in a comm failure condition. For this reason we typically recommend using the delay() function over wait() in serial polling loops. With channel Timing, you will eventually get a Timing Lag error, which is not serious, but an indicator that the things are running slower than you'd hoped.

Flow Control Type: determines the type of flow control used for communications. Flow control is the use of extra serial lines or codes to allow the device and the PC to control the speed of communications. Flow control has become less and less common as the speed of external devices has improved. Check the documentation for your device for the proper setting. Hardware uses the standard hardware flow control methods. None does not use any flow control. If only three wires of your serial device are used, then this is the correct setting unless your device uses Xon/Xoff. The three wires would then be Tx, Rx, and ground. Manual uses the settings of the other 5 parameters listed next.

Flow Control Parameters: for the proper setting of these parameters, please review the documentation of your remote device. In general these settings are not required. They are only used when Type is set to Manual.

Ethernet Configuration:

Connection Name: a valid DAQFactory name for your port.

IP Address: (applies to Ethernet Clients only) the IP address of the remote device. Depending on your network setup, you may be able to use an URL as well. To create a placeholder port, set this to a blank string. This will create the port so you can assign it to a device, but will not initialize the connection. Then when you are ready, you can change the IP address and parameters from a sequence to start using it.

Port: the IP port to connect to. For example, you would use 80 to connect to a web server. For the Ethernet Server, this is the port to listen on for new connections. Note that DAQFactory only supports one connection per port for the server.

Timeout: the number of milliseconds to wait for a read / write request to complete. This will depend on the response time of your device. We recommend keeping this between 20 and 2000 milliseconds. Longer timeouts are appropriate when working with slow devices or modem Ethernet, but will result in a latency in a comm failure condition. Please see the note above concerning polling loops and timeout.

Note: In most cases you will use Ethernet Client and not Ethernet Server, and typically the TCP version as only a few devices (such as SixNet) use UDP. Ethernet Server would typically be used for ModbusTCP slave, or to create your own networking within DAQFactory.

Note: Ethernet Server is only available in DAQFactory Standard and higher.

16.3 The Comm Protocol

Once you have created a port to communicate on, you can then select the desired protocol to communicate. The protocol is essentially the language used to communicate with your device and will be specific to your device. Many devices use a standard protocol, and we've provided protocol drivers for these protocols. For example, Modbus, which is used in a wide variety of devices. With the protocol drivers provided with DAQFactory, all you need to do is select the protocol and the port for your Comm Device and you are done. You can hit **OK** and begin using your device as described in Using Comm Devices. If your device uses a more unusual protocol that we do not currently include with DAQFactory, you have a couple choices. Your first choice is to contact us and see what would be involved for us to develop a protocol driver for your device. This can often be very simple and affordable. The other choice is to use DAQFactory scripting to write your own protocol driver. For many devices this is actually not a

difficult task and is the subject the section, User Comm Protocols.

NULL Protocol:

In addition to the regular DAQFactory protocols and user protocols, there is the NULL Protocol. The null protocol is just that, a nothing protocol. It does nothing. Use this protocol when you wish to use the low level communication functions of the comm port from a sequence or elsewhere in your application. One use for this is when you wish to receive data from a device that does not require polling and simply streams data to you without prompting. Using the OnReceive event of a user protocol is probably better for this type of device, but this serves as an example: In this case you could create a Comm Device with the null protocol, then create a simple sequence to read from the port. For example, to read from a streaming input where each line ends in a carriage return (ASCII 13) you might do:

```
Private string strIn
while (1)
  try
    strIn = Device.MyComm.ReadUntil(13)
    ... parse strIn ...
  catch() // catch any comm errors and ignore
  endcatch
endwhile
```

Note that we don't need a delay() in the loop as long as the timeout on the port is set to at least 20 milliseconds. The loop will run as fast as the data is coming in, but will pause whenever it is waiting for more data.

16.4 Using Comm Devices

With the exception of comm device using the NULL protocol, most comm devices work much like a regular DAQFactory device. Once you have created a new comm device you will see your device name listed as an option for your channels. To perform communications, create new channels specifying your comm device name as the device in the channel table or view. You will then be presented with I/O Types depending on which protocol you are using with your device. The meaning of D# (device number) and Channel # will also depend on the protocol. For example, in ModbusRTU, the device number is the module address, while channel number is the Modbus tag (without the 30,000 or 40,000...).

You will also see that your new comm device's name will appear in the list of devices in the device configuration window. If you wish to change the protocol or tweak the communications settings, you should click on your device name and not **New Serial (RS232/485) / Ethernet (TCP/IP) device** as this would create a new comm device.

Note: If using channel timing, you will usually want to set the timing / offset of all channels that communicate on a single serial / ethernet port to the same value. This will put them on the same thread and cause the requests to execute in turn. Likewise, if you have channels that communicate on multiple ports, you will want to use a different timing/offset combination for each group on a single port so that different ports end up on different threads. In this way, while DAQFactory is waiting for a response on one port, it can be communicating on the other port.

Deleting a Comm Device

If you create a comm device that you no longer need, you can easily delete it. You will have to be in Safe mode to do so however. So, first select **File - Switch To Safe Mode**. Then to delete the comm device, select **Quick - Delete Comm Device**. This will display a list of comm devices in your application. Click on the desired device and hit **DELETE**. This will remove the device.

16.5 Monitoring and Debugging Communications

To help you check your communications or debug communications problems, there are two different types of monitor windows. Both are similar, but one is an auto-hiding docking control bar similar to the workspace and command / alert, while the other is what is called a modeless window and remains on top DAQFactory whilst letting you manipulate the rest of DAQFactory. In general, you will probably want to use the docking window, activated through the View section of DAQFactory's main menu. You can quickly jump to this window by pressing Ctrl-F2. Once displayed, you can select a communications port from those you created by right clicking in the window and

selecting the port. Until you select a port, this window won't do anything. If you want to stop monitoring after selecting a port, you can select "None". Since monitoring takes a little CPU power, this will also make DAQFactory run slightly faster. Once a port is selected, you should see traffic, if there is any in the larger area at the bottom, and you can output a string as described below in the edit box at the top of the window and pressing Enter.

The only disadvantage of the docking window is that it will only display one communications port at a time. Each port also has a monitor window. To view the port's monitor window, simply open the configuration window for the comm device using the port and click on the Monitor button. This will bring up the monitor window. If you have already created channels to trigger the polling of your device, or your device simply streams data, you should see the communications in the main window. If not, you probably will need to send some data to your device to trigger a response. You can simply enter your output string at the top of the monitor window and hit send. Note that you do not need to put quotes around your string. You can also enter binary data here by preceding the ASCII code for the desired binary value with a backslash. For example:

```
req\013\010
```

would output 5 characters, r, e, q, and a carriage return, line feed. You can use \x notation to use hex values instead. Here is the same in hex notation:

```
req\x0d\x0a
```

Note that you have to specify three characters after the slash, so `req\xd\xa` is invalid. The exception are these control characters:

`\n` : new line character, a line feed, ASCII 10

`\r` : return character, a carriage return, ASCII 13

`\t` : tab character, ASCII 9

`\\` : a backslash

so, `req\013\010` is the same as `req\r\n`

Also note that a carriage return or line feed is not automatically outputted, so you must add it your self like the examples above if needed. The reason DAQFactory does not automatically add a carriage return or linefeed is that most binary protocols do not use these characters as delimiters.

The main window displays the data going in and out of the port. Data going out is preceded with **Tx:**, while incoming data is preceded with **Rx:** and displayed in a different color. By default, the data is displayed as characters when they are displayable characters, and in the \ notation when they are binary characters. You can force the display to always be in \ notation using the **Display all chars as ASCII codes** checkbox. The **Display codes in Hex** will force \x notation. If your data is all ASCII readable characters, you will probably also want to check the **New Line on CR or LF**. This will cause the monitor to move to a new line whenever it sees a carriage return (ASCII 13) or LF (ASCII 10), or a CR/LF pair, making it much more readable.

Since data often comes very quickly, you can click the **Pause** button to stop the update of the monitor window. Clicking the button again will resume the update.

Finally, the comm port monitor window is a modeless window. This means that while it is displayed you can still manipulate the rest of DAQFactory. So you can display the monitor window, and then while leaving it open, close your device configuration window and use DAQFactory while being able to monitor the details of the communication. Its much more convenient to use the docking Comm Monitor window instead of the port's monitor window, but if you need to monitor more than one port at once, you should use the port's modeless window.

16.6 Low Level Comm

In addition to using protocols to perform all the necessary communications, there are a variety of functions and variables you can call or set from sequence scripts to access the ports. These can be used in user protocols, or in regular sequences. All the functions and variables listed below start with Device.CommDeviceName. where CommDeviceName is the name you gave you Comm Device unless they are called from a user protocol. Some of these functions may fail. For example, if you try and read from a port and no data arrives within the timeout period. When these functions fail they will throw an error. The error will need to be caught in your sequence or the sequence will stop.

Functions:

For Both:

InitComm(): Initializes the serial or ethernet port. This is done automatically when you first try communications on the port.

ConfigureComm(): Displays the configuration window for the port. Use this to allow the user to switch the communications settings.

Monitor(): Displays the monitor window for the port. Use this to allow the user to view the low level communications information.

Write(string): Sends the given string to the port. No carriage return or line feed is sent, so make sure and include it in your string if needed.

Read(Num Bytes): Reads the given number of bytes and returns the result in a string. If a timeout occurs during the read then an error is generated. You will not receive a partial string. If you wish to simply read whatever is currently in the input buffer without waiting, do Read(0). This is useful for polling applications when the size of the response is unknown and there is no end of line character.

Purge(): Clears out the input and output buffers.

ReadUntil(End Character): Reads from the port until the given byte is received. End Character should be the ASCII code of the byte. For example, 13 for a carriage return. If a timeout occurs before the end of character is found, an error is generated.

ReadUntil(Regular Expression): Reads from the port until the regular expression is matched. Regular expression should be a string. If a timeout occurs before the end of character is found, an error is generated. For example, the Allen Bradley DF1 protocol generates a Chr(10) and Chr(3) to mark the end of a frame, but only if there isn't another Chr (10) ahead of the combination. To read a whole frame then, you would do:

```
ReadUntil("[^"+Chr(10)+"]" + Chr(10) + Chr(3))
```

For Serial Ports:

SetCommBreak(): Sets the comm port into a break state. This is required for some protocols (SDI-12 for example). While the port is set in break state, you cannot transmit or receive any traffic.

ClearCommBreak(): Clears the comm break state and allows communication again.

Variables:

There are different variables available for serial and ethernet type ports. For this reason, these variables should not be accessed from a user protocol as it will fix the protocol to serial or ethernet. When changing these variables, you must call InitComm() on the port to have them take effect.

For Both:

PortName: a string containing the name of the communications port assigned to this device. You can also set this variable to an existing port to change the port on the fly. Just make sure you spell the port exactly how you named it.

ProtocolName: a string containing the name of the protocol assigned to this device. You can also set this variable to an existing protocol to change the protocol on the fly. Just make sure you spell the protocol exactly how it appears in the serial/ethernet configuration list.

Timeout: the timeout value of the port in milliseconds

For Serial Ports:

Baud: the baud rate on the port. The following rates are supported: 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 56000, 57600, 115200, 128000, 256000

Byte Size: the byte size on the port,

CTS: 0 or 1 depending on whether CTS is used. Only applies if FlowControl is set to Manual.

DSR: 0 or 1 depending on whether DTR is used. Only applies if FlowControl is set to Manual.

DSRSens: 0 or 1 depending on whether the DSR Sense is needed. Only applies if FlowControl is set to Manual.

DTRControl: 0 = disable, 1 = enable, 2 = handshake. Only applies if FlowControl is set to Manual.

FlowControl: a string: "None", "Hardware" or "Manual"

Parity: 0 = no parity, 1 = odd parity, 2 = even parity, 3 = mark parity, 4 = space parity

Port: the comm port number on your computer

RTSControl: 0 = disable, 1 = enable, 2 = handshake, 3 = toggle. Only applies if FlowControl is set to Manual.

StopBits: 0 = one stop bit, 1 = 1.5 stop bits, 2 = two stop bits

For Ethernet Ports:

Address: a string containing the ethernet address

Port: the ethernet port to use

16.7 Low Level Comm Syncronization

There are many times when you may want to communicate on a single comm port from multiple threads. For example, with a user protocol you may have several different I/O types that you might want to trigger at different intervals. If the protocol is a poll request, it is possible for a two requests to be sent out the serial port from different threads before the device can respond. This can generate problems. To solve this problem we use something called synchronization. To help with this, comm devices have two functions, Lock() and Unlock():

Using Lock() and Unlock():

A synchronization flag (usually called a Mutex, or Critical Section) controls access to particular parts of code. Only one thread can have access to any of the protected code segments at one time. If you place all your low level comm calls inside protected code sections, then you can control the order they get executed and prevent overlap. In a polling protocol this usually means you put a Write() and a Read() inside the same code segment. To mark the beginning of a code segment, you should use the Lock() function of the device. This function returns 1 if it is safe to execute the subsequent code, or 0 if it is not. The function will return 0 if another thread is using the port and your code can't acquire the lock within the port's current timeout period. You should never proceed if it returns 0 and should probably generate an error message instead:

```
Private strDataIn
if (!Device.MyDevice.Lock())
throw("Port in use!")
endif
Device.MyDevice.Write("GiveMeData"+Chr(13)
strDataIn = ReadUntil(13)
Device.MyDevice.Unlock()
```

To mark then end of the protected segment, you should use the Unlock() function. This function always succeeds. Its actually relatively straight forward if you follow this format, and not nearly as dangerous as normal Mutexes. There are several points though:

1) If an error occurs, Unlock() never gets called, leaving the port blocked. This can be solved with a try / catch block:

```
Private strDataIn
if (!Device.MyDevice.Lock())
  throw("Port in use!")
  return
```

```
endif
try
  Device.MyDevice.Write("GiveMeData"+Chr(13)
  strDataIn = ReadUntil(13)
  Device.MyDevice.Unlock()
catch()
  Device.MyDevice.Unlock()
  throw()
endcatch
```

2) You could inadvertently stop the sequence midstream and therefore leave the device locked and the port blocked. Fortunately, unlike Mutexes, you can call Unlock() on a device from any sequence and it will release the port. This should be used as a last resort though, and proper try/catch coding is the best solution.

3) The Lock() and Unlock() are port specific, not device specific. This allows you to apply multiple protocols to the same serial port without interference.

For advanced users experienced with Mutexes:

The synchronization used here is a combination of a mutex and a flag. The flag indicates whether the synchronization object is signaled. The mutex and the flag are specific to each port. The flag itself is protected by a mutex.

The big difference between this method and a standard mutex is that the DAQFactory method allows you to reset the synchronization object from any thread. It also does not have the requirement that for each Lock() there must be an equal number of Unlocks. Instead, a single Unlock() call will fully release the sync object. There are good reasons why mutexes are not normally done this way, but DAQFactory is different from pretty much all other languages in that the user can start and stop threads at will and therefore could easily lockout their synchronization object and must have a way to reset it from another thread.

16.8 User Comm Protocols

If your device uses an unusual protocol then you can use DAQFactory sequence scripting to create your own protocol driver. To do so, create a new comm device as described earlier and click on the **New Protocol** button at the bottom of the comm device configuration window. Creating a protocol is almost identical to creating a user device with a few differences. To learn about creating a user protocol, please review the section on Creating a User Device and Local Variables keeping in mind the following points:

1) The File Name should end in .ddp instead of .dds. It still must be placed in your DAQFactory installation directory.

2) You should use the low level comm functions to communicate with your device. They are all local to your protocol code. Since you don't know what comm device your protocol might be assigned to, you should not use the Device.CommDevice. notation described in the section on low level comm, but instead simply reference the functions directly. For example:

```
Write("GetSomeData" + Chr(13))
strResult = ReadUntil(13)
```

This will insure that your protocol will work with whichever port it gets assigned to.

3) Since you will probably be using the low level comm functions, you can ignore the parts about extern DLL calls. Instead, you probably should do a Purge() or perhaps an Init() in your OnLoad event, and maybe a Purge() in the OnUnload event.

4) In addition to the OnLoad() and OnUnload() events, there are additional events called OnReceive(strIn) and OnSend(strOut). OnReceive() is called with every character received on the port and OnSend() is called with every string sent out the port. Note the difference, OnReceive() is with every individual character, while OnSend() is with each complete string (i.e. with each Write() call) not each character in the string. OnReceive() passes in the variable strIn that contains the character received, while OnSend() passes in the variable strOut containing the string sent out.

Note: You cannot Write() from within the OnSend() event. This will create an infinite loop as Write() will trigger the OnSend() event. If you do so, you are likely to hang the communications and have to restart DAQFactory.

You can, however do Read() and ReadUntil() from within the OnReceive, which is often very handy. For example, if you have a device that simply outputs a value every second followed by a carriage return, you might put:

```
if (strIn == chr(13)) // we've received the EOL character
  private string DataIn = ReadUntil(13) // read what we've accumulated
  Channel.AddValue(strDevice,0,"Input",0,StrToDouble(DataIn))
endif
```

This routine also shows the use of the AddValue() command that doesn't require a channel name and the strDevice variable of your device. You would want to create an I/O type called "Input", but with no code. The code above will take the value and put it in our device, D# = 0, I/O Type "Input" and channel 0.

5) When you create a new protocol, a standard Poll() function is created for you. This function performs a simple poll and wait for response. Its format is **`Poll(string out, until)`** where out is the string to send out the port, and until is the ASCII code that marks the end of the incoming line. This function will generate an error if there is a timeout, otherwise it will return the string received. The script for the code is completely viewable and even editable. The function is a very commonly used function and is provided to make creating new protocols easier.

6) Even though you may be sending your value out as a string using Write() or reading it as a string with Read(), you will probably want your I/O types to be numeric. The deciding factor is how DAQFactory stores the value internally for calculations and not how the communication with the actual device is done.

7) Local variables are specific to the device the protocol is assigned to, not the protocol itself. This means if you use the same user protocol on two different ports and therefore two different comm devices, you will have two sets of local variables.

8) Within a function or I/O type, you can access the device name that the user assigned your protocol two through the strDevice variable. This is most often used with the Channel.AddValue() function as shown above.

16.9 Predefined Protocols

16.9.1 Allen Bradley DF1 protocol

This protocol allows you to use full-duplex DF1 protocol (System point-to-point) to communicate with ALLEN-BRADLEY SLC-500 PLCs and any other PLC supporting DF1.

There are two ways within DAQFactory to communicate with your DF1 device. You can use channels as you would other devices, or you can use device functions. You can use a combination of methods as well. Both methods require you to setup a serial / ethernet device first. This is done with the device configurator which was described just previously. The DF1 protocol uses a slightly different addressing method than most PLCs.

Please note: ENQ retry should be disabled on your PLC. It is not implemented by DAQFactory and should not be needed.

Addressing:

The general format of item names for data from DF1 controllers matches the naming convention used by the programming software. The format is shown below. (The parts of the name shown in square brackets ([ ]) are optional.)

```
X [FILE] : ELEMENT [.FIELD] [/BIT]
```

X: Identifies the file type. The list below summarizes the valid file types and the default file number for each type:

X	File Type	Default File
O	*Output	0
I	*Input	1
S	Status	2
B	Binary	3

T	Timer	4
C	Counter	5
R	Control	6
N	Integer	7
F	**Floating P	8

* Output and Input file types may be Read Only depending on the SLC-500 model.

** Available only on certain SLC500 models. Check the Processor Manual for the model being used. If the Floating Point file type is not supported, file 8 is reserved and unusable.

FILE: File number must be 0-255 decimal. File 0 must be Output, file 1 must be Input, file 2 must be Status and so on for all the default file types. If this number is not specified, the default file number will be used.

ELEMENT: Element number within the file. For Input and Output files it must be between 0 and 30 decimal. All other file types, it must be between 0 and 255 decimal.

FIELD: Refers to the subelement number. If the file type is Timer, Counter or Control, then the subelement field must be replaced by its corresponding number. In some cases there is no reference needed for a subelement number, and if a subelement is included in the command, it will be ignored.

BIT: Valid for all file types except Floating Point. Must be 0-15 decimal. When setting discrete outputs in groups, you cannot specify a group that passes a word boundary. In other words, if you have a BIT value of 14, you can only set two bits at a time. It will not roll over to the next word.

File Items:

OUTPUT FILE ITEMS:

```
O[n]:e.s[/b]
```

- "n" represents the file number and is optional. If not specified, it is assumed to be zero.
- "e" indicates the element number in the file.
- "s" indicates the sub-element number (0 - 255).
- "b" specifies the bit (0 - 15 decimal.) "/b" may be omitted if necessary to treat the I/O group as a numeric value.

Examples:

```
O0:0/0
O:2/15
O:3
```

INPUT FILE ITEMS:

```
I[n]:e.s[/b]
```

- "n" represents the file number and is optional. If not specified, it is assumed to be one.
- "e" indicates the element number in the file.
- "s" indicates the sub-element number (0 - 255).
- "b" specifies the bit (0 - 15 decimal.) "/b" may be omitted if necessary to treat the I/O group as a numeric value.

Examples:

```
I1:0/0
I:2/15
I:3
```

ADDRESSING I/O MODULES:

The elements (words) in I/O modules are mapped into a memory table. If the Analog I/O modules are being used, then the point naming will differ from the point naming in the programming software. The Item Name must be computed from the sum total of words used by the previous input or output blocks regardless their relative position

in the module rack. The operator can use the programming software Data Monitor to look at the memory map of the I file or O file to verify your address. If the address is unsure, or if the PLC configuration is likely to change, copy the points in question to the N table or B table and access the data from there.

LABEL I/O MODULES WITH "WORD COUNTS":

The address of any point within the I/O datatable space, in an SLC processor, is the sum of the words occupied by previous modules (to the left in the rack) of the same type. Therefore, to determine the correct address for any particular point in the I/O datatable, one must know the number of words each module will consume. Refer to the list below:

N. of Words Module

0 1747-L524 SLC 5/02 Module Processor
1 1746-IA8 8 point 120VAC input module
1 1746-OA16 16 Point 120VAC output module
1 1746-IA16 16 point 120VAC input module
4 1746-NI4 4 point 20mA analog input module
4 1746-NO4I 4 point 20mA analog output module
1 1746-0A8 8 point 120VAC input module
2 1746-IB32 32 point DC input module

NOTE: In the table above, the minimum amount of words which can be consumed by a module is 1 (16 bits). This is due to the memory scheme of all Allen-Bradley processors.

STATUS FILE ITEMS:

S[n]:e[/b]

- "n" represents the file number and is optional. If not specified, it is assumed to be two.
- "e" indicates the element number in the file (0 - 255 decimal).
- "b" is optional. If specified, it indicates the bit (0 - 15 decimal).

NOTE: Refer to the SLC-500 Family Processor Manual (Allen-Bradley Publication) for a complete description of Status file information.

Examples:

S2:6 (major error fault)
S2:13 (math register)
S:1/5 (forces enabled)

BINARY FILE ITEMS:

B[n]:e/b or **B[n]/m**

- "n" represents the file number and is optional. If not specified, it is assumed to be three. If specified, the file number must be between 9 and 255 decimal.
- "e" specifies the element (word) number within the Binary file. It must be between 0 and 255 decimal.
- "b" specifies the bit number within the word. In the first form (where ":e" is present,) the bit number must be between 0 and 15 decimal.
- "m" also represents the bit number. However, in the second form, no word number is specified and the bit number may be between 0 and 4095.

Examples:

B3/4095 (same bit as B:255/15)
B:6/4 (same bit as B/100)
B3

TIMER FILE ITEMS:

T[n]:e[.f][/b]

- "n" represents the file number and is optional. If not specified, it is assumed to be four. If specified, the file number must be between 9 and 255 decimal.
- "e" specifies the element number (three words per element) within the Timer file. It must be between 0 and 255 decimal.
- "f" identifies one of the valid Timer fields. The valid fields for Timer Files are listed in the table below (use the numeric order as reference). If "f" is omitted, it is assumed to be the word 0.

- "b" is optional and is normally not used. All of the fields of a timer can be accessed by specifying the ".f" fields. However, it is possible to use "/b" to single out a bit in the .PRE or .ACC fields (which are words).

Order number Timer Fields

0 .PRE
1 .ACC
2 .EN
3 .TT
4 .DN

Examples:

T4:0.1 (means: .ACC)
T4:3.4 (means: .DN)
T4:1.0 (means: .PRE)

COUNTER FILE ITEMS:

C[n]:e[.f][/b]

- "n" represents the file number and is optional. If not specified, it is assumed to be five. If specified, the file number must be between 9 and 255 decimal.
- "e" specifies the element number (three words per element) within the Counter file. It must be between 0 and 255 decimal.
- "f" identifies one of the valid Counter fields. The valid fields for the Counter Files are listed in the table below (use the numeric order as reference). If "f" is omitted, it is assumed to be the word 0.
- "b" is optional and is normally not used. All of the fields of a counter can be accessed by specifying the ".f" fields. However, it is possible to use "/b" to single out a bit in the .PRE or .ACC fields (which are words).

Order number Counter Fields

0 .PRE
1 .ACC
2 .CU
3 .CD
4 .DN
5 .OV
6 .UN
7 .UA

Examples:

C5:0.1 (means: .ACC)
C5:3.5 (means: .OV)
C5:1.0 (means: .PRE)

CONTROL FILE ITEMS:

R[n]:e[.f][/b]

- "n" represents the file number and is optional. If not specified, it is assumed to be six. If specified, the file number must be between 9 and 255 decimal.
- "e" specifies the element number (three words per element) within the Control file. It must be between 0 and 255 decimal.
- "f" identifies one of the valid Control fields. The valid fields for the Control files are listed in the table below (use the numeric order as reference). If "f" is omitted, it is assumed to be the word 0.
- "b" is optional and is normally not used. All of the fields of a Control file can be accessed by specifying the ".f" fields. However, it is possible to use "/b" to single out a bit in the .LEN or .POS fields (which are words).

Order number Control Fields

0 .LEN
1 .POS
2 .EN
3 .DN
4 .ER
5 .UL
6 .IN
7 .FD

Examples:

R6:0.0 (means: .LEN)
R6:3.2 (means: .EN)
R6:1.1 (means: .POS)

INTEGER FILE ITEMS:

N[n]:e[/b]

- "n" represents the file number and is optional. If not specified, it is assumed to be seven. If specified, the file number must be between 9 and 255 decimal.
- "e" specifies the element number within the Integer file. It must be between 0 and 255 decimal.
- "b" is optional. If specified, it indicates the bit (0 - 15 decimal).

Examples:

N7:0
N7:0/15
N7:3

FLOATING POINT FILE ITEMS:

F[n]:e

- "n" represents the file number (optional). If not specified, it is assumed to be eight. If specified, the file number must be between 9 and 255 decimal.
- "e" specifies the element number within the Floating Point file. It must be between 0 and 255 decimal.

Examples:

F8:0
F8:2

{If you are using 1747-KE}

Note: Remember to configure the 1747-KE using the following settings:

- Use the DF1 FULL DUPLEX Protocol
- Use the checksum mode (BCC or CRC) as stated in P5 parameter.
- Use the baud-rate, parity, stop bits and data bits that are currently used in the driver settings.

Note: You should verify these settings by connecting your PC to the 1747-KE CONFIG port using an ASCII terminal (i.e. PROCOMM or XTALK) and then follow the steps described in the 1747-KE User Manual.

Note: Be careful to include the 1747-KE node address as the source node if it is other than 0.

DF1 Channels:

If you are reading a small number of inputs, the channel method of acquiring data from your SLC-500 device is the easier method. It is not the most efficient however, because each tag that is read from your device requires a separate request. After creating a new communications device as described previously, create a new channel, select your device name for the device type, enter the ID of the unit you are trying to communicate with as the device #, select the particular DF1 command from the I/O type, and enter the SLC-500 memory address as the Quick Note / Special / OPC column. The channel # parameter can be anything and is ignored.

I/O Types:

All of these types will read or output to a single memory location:

Read Analog BCC
Read Analog CRC
Read Binary BCC
Read Binary CRC
Write Analog BCC
Write Analog CRC
Write Binary BCC
Write Binary CRC

DF1 Functions:

This device includes functions accessible from sequences for more advanced control over your device. The biggest benefit of using device functions over creating individual channels is that you can read groups of memory locations with a single call which is more efficient. For smaller applications this may not be a problem, but if you need to read large blocks of consecutive tags, you will probably want to consider using device functions.

There are two basic formats for the device functions, input and output. Input functions take the ID of the device you wish to communicate with, the starting address, the number of points to read, and whether to use CRC or BCC. Input functions return an array of values corresponding to the data read. Output functions also take the same parameters, but take an array of values to set your memory locations to instead of the number of values. Output functions do not return anything.

You will still need to create a new communications device as described previously. For these examples, we'll assume you called it MyDevice.

Example - reading 8 holding register values: `data = Device.MyDevice.ReadAnalog(10,"F8:0",8,0)` This will read 8 values from device ID #10 starting memory location F8:0 in BCC mode and store the result in the data memory variable (which was declared previously).

Example - using the device function to pull blocks of data into channels:

1. Starting from a blank document, create your communications device using the Device Configurator.

2. Click on **CHANNELS:** in the workspace to go to the channel table. Click on the **Add** button to add a new channel row. Give the row the name **Input10**. You can of course use more appropriate names if you would like. Set the Device Type to MyDevice or whatever you named your communication device. Set the Timing column to 0. This last step is very important otherwise DAQFactory will start to poll the device directly for this channel. With the exception of the channel name, which must be unique, and the Timing, which must be set to 0, you can set the other channel parameters to anything. We still recommend selecting a device type that matches your device.

3. Click on the **Duplicate** button 7 times to create 7 new channels based off of the one you just created. DAQFactory will automatically give each the name: **Input11**, **Input12**, etc. Click **Apply** to save your changes.

4. Right click on **SEQUENCES:** in the workspace and select **Add Sequence** to create a new sequence. Call the new sequence **GroupRead** and click **OK**. You will be placed in the sequence view ready to edit your sequence. Here is the code to enter:

```
private data
while (1)
  data = Device.MyDevice.ReadAnalog(10,"F8:0",8,0)
  Input10.AddValue(Private.data[0][0])
  Input11.AddValue(Private.data[0][1])
  Input12.AddValue(Private.data[0][2])
  Input13.AddValue(Private.data[0][3])
  Input14.AddValue(Private.data[0][4])
  Input15.AddValue(Private.data[0][5])
  Input16.AddValue(Private.data[0][6])
  Input17.AddValue(Private.data[0][7])
  delay(1)
endwhile
```

5. Click **Apply** to save your sequence. Now click on the + sign next to **SEQUENCES** in the workspace to expand the tree, then right click on the **GroupRead** sequence and select **Begin Sequence**.

6. Now go to the channel view for any of your register channels and look at the table of the data coming in. (click on the + next to **CHANNELS:**, then click on the channel name and select the **Table** tab). The sequence loops once a second and reads 8 addresses in a single call to your device. The 8 Input statements in the middle add the newly read values to the channels you created earlier.

For those wondering why the data always has [0] after it, then [x] where x is the tag number: the first dimension of arrays in DAQFactory is always in time. Since all these values were read at once they all have the same time so all are on the same row (first dimension). Besides consistency, this allows you to create 2 dimensional arrays of data from multiple tags very easily.

Example - setting 8 outputs:

```
Private outdata = {{1,0,0,1,0,0,0,1}}
```

```
Device.MyDevice.WriteBinary(32,"N7:0/3",Private.outdata,1)
```

This will set 8 addresses on device ID #32 starting at memory location N7:0/3 in CRC mode. The values will be on, off, off, on, off, off, off, and on and will start at the 3rd bit of N7:0.

Note that you cannot set bits across a word boundary in a single call.

Functions:

ReadAnalog
ReadBinary
WriteAnalog
WriteBinary

16.9.3 Mitsubishi A/Q protocol

This protocol allows you to connect to Mitsubishi A and Q series PLC's through a communications card (AJ71C24 or QJ71C24 for example) using Mode 1 or Mode 4 protocols. You may have to configure these cards to communicate on mode 1 or mode 4. The QJ card will communicate on Mode 5 by default which is currently not supported so you will need to use GX Developer to set the card into Mode 1 or Mode 4. Depending on your 485 to 232 converter, you may have to set the flow control setting of the comm port in DAQFactory to Hardware, or even Manual with RTS Enabled, DTS Disabled. This varies with converter so you will have to experiment until you get the proper response.

There are two ways within DAQFactory to communicate with your device. You can use channels as you would other devices, or you can use device functions. You can use a combination of methods as well. Both methods require you to setup a serial / Ethernet device first. This is done with the device configurator which was described just previously. The A/Q protocol uses a slightly different addressing method than most PLCs. The addressing consists of a letter describing the data type followed by the address. For example D123 or T010. Because a channel's channel number only supports numbers, there is a separate I/O type for each of the possible data types. The number after the letters is specified as the channel. So, D123 would be I/O type "Read D" and channel # 123. The D#, or Device Number is the station ID of the device. If you need the NetStation parameter in your network, put this number in the Quick Note / Special column of the channel. Otherwise this defaults to 255.

This driver allows you to either create a separate channel for read and write tags for a single address, or you can create a single read tag for each address and simply write to it to make it into a read/write tag.

The channels will automatically optimize calls to use the minimal amount of calls required. For best efficiency, you should try and place memory locations consecutively within the PLC, and make sure all channels in a group of consecutive values have the same timing and offset.

FX Functions:

This device includes functions accessible from sequences for more advanced control over your device. There are three functions used to set configuration information of the driver, and a separate function for each I/O type. The advantage of using these functions over channels is that you can do error handling and have a little finer control over the polling.

SetCPUType(type): use to specify whether you have an A or AnA / AnU PLC. Specify 0 for A (or Q) and 1 for AnA/AnU.

SetMessageFormat(type): use to specify mode 1 or mode 4 protocol. Simply specify 1 or 4. The default is mode 1.

SetMessageWait(delay): specifies the message wait time. The default is 0. This is not the timeout, but instead used in the protocol.

There are two basic formats for the rest of the device functions, input and output. Input functions take the station ID, starting address and the number of points to read. You can also optionally specify the NetStation, otherwise it defaults to 255. Input functions return an array of values corresponding to the data read. Output functions also take the same parameters, but take an array of values to set your memory locations to instead of the number of values. Output functions do not return anything.

You will still need to create a new communications device as described previously. For these examples, we'll assume you called it MyDevice.

Example - reading 8 TN register values starting at TN10: `data = Device.MyDevice.ReadTN(1,10,8)` This will read 8 values starting memory location TN10 and store the result in the data memory variable (which was declared previously).

Example - setting 8 outputs:

```
Private outdata = {{1,0,0,1,0,0,0,1}}
Device.MyDevice.WriteTN(1,10,Private.outdata)
```

This will set 8 addresses starting at memory location TN10. The values will be on, off, off, on, off, off, off, and on.

All input functions have the parameter list (Station, Starting Address, Num Points, [NetStation]). All outputs have the parameter list (Station, Starting Address, {Data}, [NetStation]).

16.9.4 Mitsubishi FX direct serial protocol

This protocol allows you to connect to Mitsubishi FX series PLC's through the base unit programming connector. This is the PS/2 or DIN style connector on the PLC itself. This port always communicates under 9600,7,E,1. Depending on your 485 to 232 converter, you may have to set the flow control setting of the comm port in DAQFactory to Hardware, or even Manual with RTS Enabled, DTS Disabled. This varies with converter so you will have to experiment until you get the proper response. The SC-09 cable will not work for Mitsubishi FX using this protocol.

This protocol is often called the direct serial protocol and is not the same as the multidrop protocol. This protocol will not work in multidrop configurations.

There are two ways within DAQFactory to communicate with your FX device. You can use channels as you would other devices, or you can use device functions. You can use a combination of methods as well. Both methods require you to setup a serial / ethernet device first. This is done with the device configurator which was described just previously. The FX protocol uses a slightly different addressing method than most PLCs. The addressing consists of a letter describing the data type followed by the address. For example D123 or T010. Because a channel's channel number only supports numbers, the addressing for FX channels is placed in the Quick Note / Special / OPC column. The channel # is not used and can be any value. Since it is not a multidrop configuration, the Device # of a channel is also not used.

FX Channels:

If you are reading a small number of inputs, the channel method of acquiring data from your FX device is the easier method. It is not the most efficient however, because each tag that is read from your device requires a separate request. After creating a new communications device as described previously, create a new channel, select your device name for the device type, select the particular FX command from the I/O type, and enter the FX memory address as the Quick Note / Special / OPC column. The device # and channel # parameters can be anything and is ignored.

I/O Types:

All of these types will read or output to a single memory location:

Read Bit
Read Unsigned Word
Read Signed Word
Read Unsigned DWord
Read Signed DWord
Read Float
Write Bit
Write Unsigned Word
Write Signed Word
Write Unsigned DWord
Write Signed DWord
Write Float

Note that the 32 bit reads and writes (DWord and Float) use two memory locations in the PLC.

FX Functions:

This device includes functions accessible from sequences for more advanced control over your device. The biggest benefit of using device functions over creating individual channels is that you can read groups of memory locations with a single call which is more efficient. For smaller applications this may not be a problem, but if you need to read large blocks of consecutive tags, you will probably want to consider using device functions.

There are two basic formats for the device functions, input and output. Input functions take the starting address and the number of points to read. Input functions return an array of values corresponding to the data read. Output functions also take the same parameters, but take an array of values to set your memory locations to instead of the number of values. Output functions do not return anything.

You will still need to create a new communications device as described previously. For these examples, we'll assume you called it MyDevice.

Example - reading 8 holding register values: `data = Device.MyDevice.ReadUnsignedWord("D10",8)` This will read 8 values starting memory location D10 and store the result in the data memory variable (which was declared previously).

Example - using the device function to pull blocks of data into channels:

1. Starting from a blank document, create your communications device using the Device Configurator.

2. Click on **CHANNELS:** in the workspace to go to the channel table. Click on the **Add** button to add a new channel row. Give the row the name **Input10**. You can of course use more appropriate names if you would like. Set the Device Type to MyDevice or whatever you named your communication device. Set the Timing column to 0. This last step is very important otherwise DAQFactory will start to poll the device directly for this channel. With the exception of the channel name, which must be unique, and the Timing, which must be set to 0, you can set the other channel parameters to anything. We still recommend selecting a device type that matches your device.

3. Click on the **Duplicate** button 7 times to create 7 new channels based off of the one you just created. DAQFactory will automatically give each the name: **Input11**, **Input12**, etc. Click **Apply** to save your changes.

4. Right click on **SEQUENCES:** in the workspace and select **Add Sequence** to create a new sequence. Call the new sequence **GroupRead** and click **OK**. You will be placed in the sequence view ready to edit your sequence. Here is the code to enter:

```
private data
while (1)
  data = Device.MyDevice.ReadUnsignedWord("D0",8)
  Input10.AddValue(Private.data[0][0])
  Input11.AddValue(Private.data[0][1])
  Input12.AddValue(Private.data[0][2])
  Input13.AddValue(Private.data[0][3])
  Input14.AddValue(Private.data[0][4])
  Input15.AddValue(Private.data[0][5])
  Input16.AddValue(Private.data[0][6])
  Input17.AddValue(Private.data[0][7])
  delay(1)
endwhile
```

5. Click **Apply** to save your sequence. Now click on the + sign next to **SEQUENCES** in the workspace to expand the tree, then right click on the **GroupRead** sequence and select **Begin Sequence**.

6. Now go to the channel view for any of your register channels and look at the table of the data coming in. (click on the + next to **CHANNELS:**, then click on the channel name and select the **Table** tab). The sequence loops once a second and reads 8 addresses in a single call to your device. The 8 Input statements in the middle add the newly read values to the channels you created earlier.

For those wondering why the data always has [0] after it, then [x] where x is the tag number: the first dimension of arrays in DAQFactory is always in time. Since all these values were read at once they all have the same time so all are on the same row (first dimension). Besides consistency, this allows you to create 2 dimensional arrays of data from multiple tags very easily.

Example - setting 8 outputs:

```
Private outdata = {{1,0,0,1,0,0,0,1}}
Device.MyDevice.WriteBit("D0",Private.outdata)
```

This will set 8 addresses starting at memory location D0. The values will be on, off, off, on, off, off, off, and on.

Note that you cannot set bits across a word boundary in a single call.

Functions:

ReadBit
ReadU16
ReadS16
ReadU32
ReadS32
ReadFloat
WriteBit
WriteU16
WriteS16
WriteU32
WriteS32
WriteFloat

16.9.5 ModbusTCP/RTU/ASCII Master protocols

The Modbus protocol is actually made up of two similar protocols, ModbusASCII and ModbusRTU. These protocols are almost identical except for the method used to communicate and the functions available. ModbusASCII uses ASCII to pass the Modbus commands. ModbusRTU uses binary. Check your device to determine which mode it prefers. Modbus TCP is sometimes ModbusRTU on Ethernet, but can also be true ModbusTCP, which is a slightly different protocol. Since DAQFactory allows you to apply any protocol to both serial and Ethernet, you can do ModbusRTU (or ModbusASCII) protocol on an Ethernet port, but depending on your device, this may not be the ModbusTCP it needs. For this you will need to use the ModbusTCP protocol. The difference in the protocol is only a few header and tail bytes, so the two protocols appear the same in DAQFactory. Simply select which one you need and all the rest of the information described below applies to both, though the examples describe ModbusRTU.

In general, devices that have a serial port out and then go to a serial to Ethernet adapter will use RTU, while devices with an Ethernet port built in will use TCP. Only a few older devices still use ModbusASCII.

One difference between the TCP and RTU drivers is that the RTU driver adds the SetDelay() function. Some devices require a little downtime between requests and can't handle a continual barrage of queries. This is especially true in a multidrop configuration. To make this easy to implement, you can call SetDelay() with the number of milliseconds to delay after each request before allowing any communications on the port. For example, Device.mydevice.SetDelay(50) would cause the driver to pause 50 milliseconds after every query before allowing any other communications on the port. This was originally designed for the Watlow SD type temperature controllers, but we've found it useful for some other devices. The typical indicator that a delay may be required is if you are reading non-sequential channels and the first channel reads without a problem, but the second channel times out. A third channel would then read correctly because the timeout would create the necessary delay.

There are two ways within DAQFactory to communicate with your Modbus device. You can use channels as you would other devices, or you can use device functions. You can use a combination of methods as well. Both methods require you to create a Communications Device as described earlier in this chapter.

Modbus Channels:

The channel method of acquiring data from your Modbus device is the easier method. To create a Modbus channel, select your communications device, enter the ID of the unit you are trying to communicate with as the device #, select the particular Modbus command from the I/O type, and select the Modbus memory address as the channel #.

Detail: Unfortunately some manufacturers 0 index their addresses and others use 40,001 type notation. Internally, the Modbus packet itself uses 0 indexing. The Modbus protocol driver will assume you are using 0 indexed addresses until you try and read from an address greater than 30,000, but less than 50,000. Once you do this, it will assume you are using 40,001 type notation and subtract the appropriate value to send the correct request. This will apply to all the devices using the protocol.

In general you don't have to worry about this, unless you are only setting coils and output registers and still want to use the 1 indexed, 40,001 type notation. In this case you will either need to read a single holding or input register with an address greater then 30,000 or manually adjust your tag numbers to 0 indexed.

Alas, some devices use 0 index notation, but number their registers up into the 30,000 and 40,000. To force the

driver into 0 index mode, read an address greater than 50,000. Then the driver will remain in 0 indexed mode even if you read a 30,000 or 40,000 address.

In the I/O type, the number in parenthesis refers to the Modbus command number used. Since even the terminology of Modbus varies some between manufacturers, we have provided this to help keep things clear.

This driver allows you to either create a separate channel for read and write tags for a single address, or you can create a single read tag for each address and simply write to it to make it into a read/write tag. Obviously, when writing to a read tag the Modbus command number indicated will be substituted for the appropriate output command number. Also, you can only write to Holding registers and coils and not to input registers. You also cannot write to the U16 data type. Use S16 instead.

DAQFactory will automatically combine sequential channels with the same Timing and Offset into a single request to optimize the traffic on your comm port. When doing floating or 32 bit values, it assumes sequential channel numbers are in steps of 2 (40,001, 40,003, etc.). For the "skip 2" I/O types, it assumes sequential channel numbers are in steps of 3.

Example - reading a holding register once a second:

1. From a blank document, create a new Communication Device selecting ModbusRTU as the protocol. Read the section on the creating a Comm Device for more information. We'll assume for all these examples that you called your device MyDevice.

2. Click on **CHANNELS:** in the workspace to go to the channel table. Click **Add** to add a new channel row. In that new row, enter a channel name such as **`Register1`**. Select **`MyDevice`** for the device type, then enter the ID of your device under D#. For the I/O type, select **`Read Holding U16 (3)`**. You may have to expand the column to read it. The U16 means "Unsigned 16 bit integer" and is the same as an unsigned word. (3) means we are using Modbus command 3. Under Chn #, enter the memory location for the register you would like to read. Holding registers are normally numbered from 30000. However the 3 only identifies the address as a holding register and is not required. Therefore if your memory address is 30063 you should enter a channel number of 63.

3. Click **Apply** to save your changes. Since the channel you just created has a Timing parameter of 1.00, DAQFactory will immediately start polling that holding register once a second.

4. To quickly see the data, click on the **+** next to **CHANNELS:** in the workspace to expand the tree. Then click on your new channel name. This will display the channel view for this channel. Click on the **Table** tab to see the data in tabular form. You can now either add more channels, or go to the page and use page components to display your acquired data.

Example - forcing a coil:

1. From a blank document, create a new Communication Device selecting ModbusRTU as the protocol. Read the section on the creating a Comm Device for more information. We'll assume for all these examples that you called your device MyDevice.

2. Click on **CHANNELS:** in the workspace to go to the channel table. Click **Add** to add a new channel row. In that new row, enter a channel name such as **`Register1`**. Select **`MyDevice`** for the device type, then enter the ID of your device under D#. For the I/O type, select **`Force Coil (5)`**. You may have to expand the column to read it. Under Chn #, enter the memory location for the coil you would like to set.

3. Click **Apply** to save your changes. To quickly toggle the state of the coil, right click on the channel row and select **Toggle**. You should see your coil toggling between on and off. You can now either add more channels, or go to the page and use page components to control your output coil.

I/O Types:

All of the types will read or output to a single memory location. The number in parenthesis is the Modbus command used. Many of the commands are identical and the difference is how the data is treated. Here is a description of the codes used:

U16 : unsigned 16 bit word
S16 : signed 16 bit word
U32 : unsigned 32 bit double-word (long unsigned integer)
S32 : signed 32 bit double-word (long integer)
U32 R Words : unsigned 32 bit double-word with words reversed
S32 R Words : signed 32 bit double-word with words reversed
Float : 32 bit floating point value
Float R Bytes : 32 bit floating point value with bytes reversed
Float R Words : 32 bit floating point value with words reversed

Skip 2 : skip 2 bytes before reading the value

Modbus Functions:

This protocol includes functions accessible from sequences for more advanced control over your device. This is especially useful when you want to optimize the writing of multiple consecutive values to your device as the channels themselves cannot optimize outputs, only inputs.

There are two basic formats for the device functions, input and output. Input functions take the ID of the device you wish to communicate with, the starting address, and the number of points to read. Input functions return an array of values corresponding to the data read. Output functions also take the ID and starting address, but then take an array of values to set your memory locations to. Output functions do not return anything.

To use these functions you must create a new Communication Device with the ModbusRTU as its protocol. Read the section on the creating a Comm Device for more information. We'll assume for all these examples that you called your device MyDevice.

Example - reading 8 holding register values: `data = Device.MyDevice.ReadHoldingU16(32,10,8)` This will read 8 values from device ID #32 starting memory location 10 and store the result in the data memory variable.

Example - using the device function to pull blocks of data into channels. This can also be done by simply setting all your channels to the same Timing / Offset. The driver will automatically optimize the calls to your device. But, as an example, you can also:

1. From a blank document, create a new Communication Device selecting ModbusRTU as the protocol. Read the section on the creating a Comm Device for more information. We'll assume for all these examples that you called your device MyDevice.

2. Click on **CHANNELS:** in the workspace to go to the channel table. Click on the **Add** button to add a new channel row. Give the row the name **Register10**. You can of course use more appropriate names if you would like. Set the Device Type to **MyDevice**. Set the Timing column to 0. This last step is very important otherwise DAQFactory will start to poll the device directly for this channel. With the exception of the channel name, which must be unique, and the Timing, which must be set to 0, you can set the other channel parameters to anything. We still recommend selecting a device type that matches your device.

3. Click on the **Duplicate** button 7 times to create 7 new channels based off of the one you just created. DAQFactory will automatically give each the names: **Register11**, **Register12**, etc. Click **Apply** to save your changes.

4. Right click on **SEQUENCES:** in the workspace and select **Add Sequence** to create a new sequence. Call the new sequence **GroupRead** and click **OK**. You will be placed in the sequence view ready to edit your sequence. Here is the code to enter:

```
private data
while (1)
  data = Device.MyDevice.ReadHoldingU16(32,10,8)
  Register10.AddValue(Private.data[0][0])
  Register11.AddValue(Private.data[0][1])
  Register12.AddValue(Private.data[0][2])
  Register13.AddValue(Private.data[0][3])
  Register14.AddValue(Private.data[0][4])
  Register15.AddValue(Private.data[0][5])
  Register16.AddValue(Private.data[0][6])`
  Register17.AddValue(Private.data[0][7])
  delay(1)
endwhile
```

5. Click **Apply** to save your sequence. Now click on the + sign next to **SEQUENCES** in the workspace to expand the tree, then right click on the **GroupRead** sequence and select **Begin Sequence**.

6. Now go to the channel view for any of your register channels and look at the table of the data coming in. (click on the + next to **CHANNELS:**, then click on the channel name and select the **Table** tab). The sequence loops once a second and reads 8 registers in a single call to your device. The 8 Register statements in the middle add the newly read values to the channels you created earlier.

For those wondering why the data always has [0] after it, then [x] where x is the tag number: the first dimension of arrays in DAQFactory is always in time. Since all these values were read at once they all have the same time so all are on the same row (first dimension). Besides consistency, this allows you to create 2 dimensional arrays of data from multiple tags very easily.

Example - setting 8 coils:

```
Private.outdata = {{1,0,0,1,0,0,0,1}}
Device.MyDevice.ForceMultipleCoils(32,20,Private.outdata)
```

This will set 8 coils on device ID #32 starting at memory location 20. The values will be on, off, off, on, off, off, off, and on.

Functions:

The functions available start with either ReadInput, ReadHolding, ReadCoil, ForceCoil or SetRegister and follow the same coding as I/O types described above.

Note: Once the data is within DAQFactory, it is treated as a 64 bit floating point value.

16.9.6 ModbusTCP/RTU/ASCII Slave protocol

The ModbusTCP, ModbusRTU and ModbusASCII Slave protocols allows you to set DAQFactory up as a Modbus slave. A modbus slave receives commands from a master and performs those commands. Typically, PLC's and other devices are the slave and SCADA applications such as DAQFactory are the master. DAQFactory can act as the Modbus master but it can also act as a slave. This is useful if you want to integrate DAQFactory into a larger SCADA system that acts as a Modbus master, or you wish to network DAQFactory using Modbus instead of the built in networking.

Setting up the slave protocol is similar to other protocols. Please review the section on Creating a Comm Device for more information about working with protocols. Once you have created a device with the ModbusTCP, ModbusRTU or ModbusASCII Slave protocol, you will need to assign Modbus register addresses to any channels you wish to have accessible through the slave. This is done right in the channel configuration, either in the channel table or the channel view in the "Mod Slave #" or "Mod Slave Tag" area, NOT the "Channel #". The numbers you assign are rather arbitrary, but should be unique in your system. Register addresses under 10,000 (but greater than 0) will result in a signed 16 bit word, the default Modbus type, to be returned. Registry values 10,000 and over are treated as 32 bit floating point values. Since this takes 2 Modbus words, the register address should increment by 2 (i.e. 10000, 10002, 10004, etc). The duplicate button in the channel table will do this for you automatically.

Once the register addresses are assigned, a Modbus master can connect in and retrieve and set values. The Modbus Slave protocols supports Modbus function codes 3 and 4 for reading values and treats these codes the same way. It also supports function code 16 to set values. All other codes are ignored. The default slave address of DAQFactory is 1. This is a variable of the protocol, so you can change it using the simple script: `Device.mydevice.ModbusAddress = newvalue` where mydevice is the name you gave the slave device, and newvalue is the new address.

Note: With ModbusTCP slave you will typically use the Ethernet Server and not the Ethernet Client you would use for ModbusTCP master. You certainly can apply any of these protocol to any of the port types, however, this arrangement is the most common.

Note: The Modbus Slave protocols requires DAQFactory Standard or higher.

17 Devices

17 Devices

17.1 Device Overview

As a data acquisition package, DAQFactory must communicate with many different pieces of hardware. To enable DAQFactory to do so, it uses what are called *Devices*. These are libraries of code that act as translators between DAQFactory and the particular piece of hardware you would like to use. AzeoTech offers a wide variety of devices to allow you to communicate with an even larger variety of data acquisition and process control hardware and sensors. This chapter provides specific information for each device currently shipped with DAQFactory. Additional and updated devices are made available from our website at www.azeotech.com. We can also provide you with custom or enhanced devices for a modest cost if you would like.

Installing a device:

When DAQFactory ships, it installs a wide selection of devices. However, we are constantly creating new devices and updating old ones. Typically, installing these new devices is simply a matter of pointing the install program to the directory where DAQFactory is installed. Almost all devices are simply made up of a **`.dll`** file with the code for the actual device and a **`.dds`** text file with some information about the device.

Note: Most hardware specific devices require drivers from the manufacturer. The drivers must be installed for the corresponding DAQFactory device drivers to function. If you are getting the C1033 error, it may be because your hardware manufacturer driver is not installed, or because a required DLL from the manufacturer cannot be found.

Tip: If you would like to remove a device from the list of possible devices displayed when creating Channels, simply rename (or delete) the **`.dds`** file associated with the device(s) you would like to remove to something without a **`.dds`** extension. DAQFactory only loads devices with a **`.dds`** file associated with them.

17.2 Device Functions

Some devices have their own functions that can be called from sequences or expressions. These functions start with Device.DeviceName. For example: **`Device.MyDevice.ReadHoldingU16(1,600,3).`** Only some devices have functions.

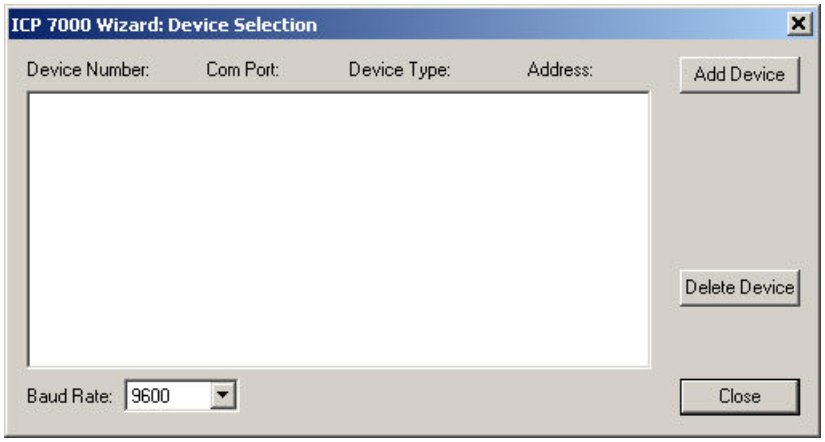
17.6 ICP-DAS i7000

This device driver supports the ICP series RS485 remote DAQ devices. These are manufactured by ICPDAS (www.icpdas-usa.com) and are brand labeled by Measurement Computing. These devices are identical to the ADAM and other similar devices marketed by several different companies. At present only the 7017 + 7018 devices are supported (more will be added on customer request).

To use this device, you must first configure using the ICP7000 device configurator:

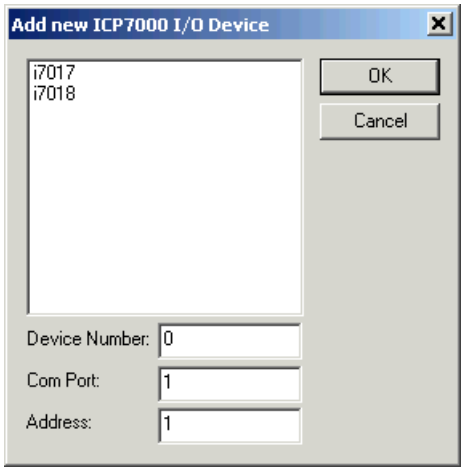
1. Open the device configurator: the easiest way is to select **Quick - Device Configuration...** from the main menu.
2. Click the **ICP7000** configurator and click **Select**.

The ICP7000 Device Selection window will be displayed:



3. Select the baud rate for communication. Although you can communicate with multiple devices on different COM ports, you can only communicate at one baud rate on all ports.

4. Click **Add Device** to add your first device. A new window will display:



5. Select your device and then fill in the three parameters. The device # is a arbitrary number that you will use in DAQFactory to identify this unit. It should be a unique number for each unit. The Com port determines which com port the unit is connect to. The address is the ID address assigned to the unit by the ICP configuration utility. You will probably have to run that program first to determine the addresses of your units.

6. Click **OK** to add your unit. It will now appear in the Device Selection window. If you have more units, click **Add Device** again and enter the appropriate information. Remember that with multiple units, you must specify a unique device number.

7. When done, click **Close**. This configuration is saved with your document, so you will need to perform the above steps with every new document you create. Alternatively, you can save your document now before you have done anything and use it as a template for future documents.

8. At this point, you can create channels for your board using the channel table and the standard DAQFactory methods.

Both the 7017 and 7018 are analog input modules. Both of these units support the Analog Input I/O type. In addition to the analog input, there is a single channel command for setting the range called Set Range.

Example: setting the gain.

1. Create a new channel in the channel table: ICP7000 device type with a device number matching the device number you assigned to the unit you would like to set the range for. I/O type is Command. Channel # is 0. If you have multiple channel commands on this unit, make sure each has a unique channel number.

2. Under Quick Note / Special / OPC for the channel a button should appear along the right side of the cell with three

dots (...). Click on this button.

3. The Channel Parameters window will appear. From the drop down at the top of the window, select **Set Range**.

4. One parameters will appear in the table: New Range is the desired gain to set the unit to. Select from the list of possible gains. Choose your settings and click **OK**. The all ranges displayed may not be supported by your unit. The 7018 supports more ranges than the 7017, but both will list the same options.

5. Click Apply to save your new channel. Simply creating the channel does not actually set the gain. To do this you must set the channel to something. It does not matter what. The simple way to do this is to right click on the new channel row and select Set Value, enter any value and click OK. You can also use components or sequences to set this channel.

17.7 LabJack UE9 / U6 / U3

This device driver communicates with the LabJack UE9 over either USB or Ethernet and the LabJack U3 and U6 over USB. This driver supports all modes of operation. This driver appears as "LabJackUD" within DAQFactory. The LabJack U12 driver appears as "LabJack_U12".

To learn how to use DAQFactory with the UE9, U6, or U3, please refer to the **Using DAQFactory with the LabJack** guide included in the DAQFactory installation. Go to **Start - Programs - DAQFactory - Using DAQFactory with the LabJack**. Included here is a function summary:

Initial function calls to setup driver:

```
include("c:\program files\labjack\drivers\labjackud.h")
```

This should go in a sequence, and probably one with AutoStart checked. You should change the directory specified if you installed the LabJack CD to a different directory. Once this function runs, all the constants in the .h file will appear as defined constants inside of DAQFactory and can be referenced directly by name, just like a variable, but read only.

```
using("device.labjack")
```

This function will move all the DAQFactory LabJack functions described below, as well as the ones for the LJM, into the global name space so you don't have to put device.labjackUD. or device.labjackM in front. Without this, all the functions listed below must have device.labjackUD. infront (i.e. "device.LabjackUD.AddRequest(...)") Again, this should go in a sequence, probably right after the include() function.

Once these functions are called, DAQFactory script for communicating with the LabJack will be almost identical to C and psuedocode examples provided by LabJack. The exception is that DAQFactory uses the @ sign to specify a variable by reference. Also, DAQFactory doesn't support callback functions, so it handles streaming data internally as described below.

Here are the functions. Almost all return an error code:

AddRequest(D#, IOType, Channel, Value, x1, UserData): This function works very much like the AddRequest() function in the UE9 programming library. The D# is the device number you assigned in the configurator or was returned by OpenLabJack. IOType and Channel can be the constants for a particular task. Since DAQFactory does not support indirect pointers, the Value, and x1 parameters only hold values used for setting outputs or parameters. Results are returned by GetResult(). Only the first three parameters are required.

DoubleToStringAddress(Number, @String, HexDot): This function is identical to the UD equivilent.

eAIN(D#, ChannelP, ChannelN, @Value, Range, Resolution, Settling, Binary, R1, R2): This function is identical to the UD equivilent.

eDAC(D#, Channel, Voltage, Binary, R1, R2): This function is identical to the UD equivilent.

eDI(D#, Channel, @State): This function is identical to the UD equivilent.

eDO(D#, Channel, State): This function is identical to the UD equivilent.

eGet(D#, IOType, Channel, @Value, x1): This function combines AddRequest, GoOne, and GetResult into one call. It obviously can only perform one task at a time, but for simple tasks can be easier to deal with. Like AddRequest(), only the first three parameters are required. So you could do this to read an analog value:

```
eGet(1,LJ_ioANALOG_INPUT, 1, @result)
```

ePut(D#, IOType, Channel, Value, x1): This function combines AddRequest, GoOne, and GetResult into one call, but designed for setting a parameter. It obviously can only perform one task at a time, but for simple tasks can be easier to deal with.

ErrorToString(ErrorCode, @String): This function is identical to the UD equivilent.

eTCConfig(D#, EnableTimers, EnableCounters, TCPinOffset, TimerClockBaseIndex, TimerClockDivisor, TimerModes, TimerValues, R1, R2): This function is identical to the UD equivilent.

eTCValues(D#, ReadTimers, UpdateResetTimers, ReadCounters, ResetCounters, @TimerValues, @CounterValues, R1, R2)",Tip): This function is identical to the UD equivilent.

GetDriverVersion(): This is the simplest function and simply returns the version number of the driver (i.e. 2.04)

GetFirstResult(D#, @IOType, @Channel, @Value, @x1, @UserData): This function is identical to the UD equivilent.

GetLastError(D#): returns the error code of the last error on the given device. The error codes are described in the labjackud.h file of the UE9 programming library.

GetLastStreamError(D#): returns the error code of the last error generated by the stream retrieval routine. Since stream data retrieval is done internally, this is the only way to see any errors occurring in stream. The error codes are described in the labjackud.h file of the UE9 programming library.

GetNextError(D#, @IOType, @Channel, [@Error]): Gets the next error from the list of errors from the previous GoOne() on the given D#. Repetitive calls will cycle through the list. Repeat until LJE_NOERROR is returned. This function returns the error, or LJE_NOERROR, but also allows you to optionally provide a reference to a variable to also receive the error. This allows you to put this function inside the while() expression and assign the error to a variable for further processing at the same time.

GetNextResult(D#, @IOType, @Channel, @Value, @x1, @UserData): This function is identical to the UD equivilent.

GetResult(D#, IOType, Channel, @Value): Gets the result for the given operation. Like AddRequest, IOType and Channel can be constants or numbers.

GoOne(D#): Performs all the AddRequests that have queued up on the given device since the last call to GoOne().

OpenLabJack(Device Type, ConnectionType, Address, FirstFound, @D#): this function opens the specified LabJack if not already open and returns a device number in @D# that is used by all other functions and channels. This is not quite the same as the handle returned by the regular C function, but should essentially be treated the same. Use this function if you need to dynamically open LabJack's at runtime without using the device configurator. Device Type and Connection Type take constants such as LJ_dtUE9 and LJ_ctUSB. Address is a string. First found is either 0 or 1, where 1 means find first found and ignore the specified Address.

ResetLabJack(D#): This function is identical to the UD equivilent.

StringToConstant(String): This function is identical to the UD equivilent.

StringToDoubleAddress(String, @Number, HexDot): This function is identical to the UD equivilent.

17.8 LabJack U12 device

This device driver communicates with the LabJack U12 USB data acquisition device. It supports all modes of operation.

To perform the basic functions of the LabJack U12, you simply have to create some channels for communication. You should pass the ID number of the desired LabJack U12 as the Device # parameter of your channel. A device number of 0 will select the first found LabJack U12 device and is the easiest method if you only have one LabJack U12. You cannot use device number 0 if you have more than one LabJack U12.

Device Details:

The LabJack U12 requires 20ms to perform any acquisition block. An acquisition block is either reading of up to 4 A to D channels, setting either or both D to A's, setting / reading the digital outputs, and reading the counter.

Therefore, do not make your Channel Timing values less than 0.02 seconds, as the LabJack will not be able to keep up. If you are using more than one acquisition block, your Timing values will have to be corresponding larger. DAQFactory, is however, multithreaded, so while it takes 20ms to communicate with a single LabJack, if you have two LabJacks, you can communicate with the second LabJack at the same time, thus increasing the maximum throughput. This follows for 3, 4, or more LabJacks. In order to do this however, you must give the Channels on the different LabJacks a different Timing and Offset parameter. The difference can be as small as 0.001 if you would like. This is actually one of the uses for the Offset parameter.

Analog Input: There are two types of analog inputs with the LabJack, single ended and differential. All are 12 bit and return measurements in volts. Single ended inputs are wired into AI0 through AI7 and GND and provide 8 inputs. Differential inputs are wired in pairs, AI0-AI1, AI2-AI3, AI4-AI5, AI6-AI7. The GND terminal is not used. Because two lines are used for each differential input, the LabJack only provides 4 differential inputs. Differential inputs have two advantages over single ended. First, they cancel possible noise in your wiring, and second they have programmable range. If you do not need to more than 4 inputs, we recommend using the differential inputs.

Example - Reading an analog input at 1 hz. See under Streaming Analog Input below for rates faster than 50hz.

1. Starting from a blank document with a single LabJack plugged into your computer, click on **CHANNELS:** in the workspace to display the channel table. Click on **Add** to add a blank row to the table.

2. Enter a simply channel name like Input0 for the channel name column. For device type, select LabJack_U12. For device number enter 0. Select A to D for I/O type.

3. If you want to read one of the eight single ended inputs, enter 0 through 7 for channel number, corresponding to AI0 through AI7. For differential inputs, use 8 through 11, corresponding to AI0-AI1, AI2-AI3, AI4-AI5, and AI6-AI7. In this example, select one of the differential inputs, say 8.

4. The next column is timing which determines how often the analog channel is read. The LabJack has a maximum software polled rate of 50hz, so the timing column should never have a value less than 0.02. As you use more and more inputs and outputs, the minimum timing value will grow. If you are getting C1038 Timing Lag errors then your timing values are probably too small. For now, leave it at 1.00.

5. Click **Apply** to save your new channel. DAQFactory will immediately start reading from the LabJack. The LED on the LabJack unit should flash indicating DAQFactory is communicating with it.

6. To quickly see the data, click on the + sign next to **CHANNELS** in the workspace, then click on your new channel that will appear when the tree expands. This will display the channel view. Click on the **Graph** tab. You should see a graph of your data. Alternatively, click on the **Table** tab to see the actual values in tabular form.

Example - Setting the range on a differential channel. Range is not an option with single ended inputs. Range is set using a channel command.

1. Create a new channel in the channel table: LabJack_12 device type with a device number of 0 (assuming you have only one LabJack). I/O type is Command. Channel # is 0. If you have multiple channel commands on this LabJack, make sure each has a unique channel number.

2. Under Quick Note / Special / OPC for the channel a button should appear along the right side of the cell with three dots (...). Click on this button.

3. The Channel Parameters window will appear. From the drop down at the top of the window, select **Gain Settings**.

4. There are two parameters that will appear in the table. Channels takes a comma delimited list of channels to set the gain for. Since single ended channels do not have gain, valid channels are 8 through 11. For example **`8,9,10,11`** would set all the channels to the same gain. The second parameter is the Gain. Select from the drop down list the desired range. Click **OK**.

5. Click **Apply** to save your new channel. Simply creating the channel does not actually set the gain. To do this you must set the channel to something. It does not matter what. The simple way to do this is to right click on the new channel row and select Set Value, enter any value and click OK. You can also use components or sequences to set this channel.

Analog Output: The U12 supports two 10 bit analog output channels. These are both 0 to 5 volt outputs.

Example - Creating and setting an analog output channel.

1. Starting from a blank document with a single LabJack plugged into your computer, click on **CHANNELS:** in

the workspace to display the channel table. Click on **Add** to add a blank row to the table.

2. Enter a simply channel name like Output0 for the channel name column. For device type, select LabJack_U12. For device number enter 0. Select D to A for I/O type.

3. The two analog output channels of the LabJack are numbered 0 and 1 corresponding to the AO0 and AO1 pins on the device. Enter the desired number in the channel column.

4. Click **Apply** to create your new channel. Once you have the channel created you can now set it to a value. The quick way to do this is to right click on the new channel in the channel table and select Set Value. This will display a window where you can type in your voltage setting. When you click OK, the analog output will be set to that voltage, provided it is between 0 and 5. You can also set this value using various page components or in an automated sequence. This is described in the rest of the DAQFactory documentation.

Digital I/O: The U12 supports 20 channels of Digital I/O numbered 0 to 19. 0 through 3 are the front panel IO0 though 3 pins. 4 through 19 are the D0 through D15 pins on the DB25 on the side of the unit. Each can be configured for either input or output by simply assigning it to a DigIn or DigOut Channel. The current state of a DigOut can be read back using the same DigIn channel once the DigOut is set. Until a DigOut is set, the channel is assumed to be an input. Use the Set I/O to Dig In command to revert a channel assigned as an output back to an input.

Example - To read a digital input, simply follow the example provided for the analog input, but use the Dig In I/O type instead.

Example - To create a digital output, simply follow the example provided for the analog output, but use the Dig Out I/O type instead. When you get to step 4, the Toggle option will be used in place of Set Value, as digital outputs only have two valid values, 0 and 1.

Example - All the I/O pins on the LabJack can be input or output. When the LabJack is first plugged in, all the pins are set as inputs. Once you set a pin to a value using the Dig Out I/O type, it is considered an output. To switch it back to a digital input:

1. Create a new channel in the channel table: LabJack_12 device type with a device number of 0 (assuming you have only one LabJack). I/O type is Command. Channel # is 0. If you have multiple channel commands on this LabJack, make sure each has a unique channel number.

2. Under Quick Note / Special / OPC for the channel a button should appear along the right side of the cell with three dots (...). Click on this button.

3. The Channel Parameters window will appear. From the drop down at the top of the window, select **Set I/O to Dig In**.

4. There is one parameter that will appear in the table. The channels parameter takes a comma delimited list of channels to set to digital input, for example **0,1,2,3** would refer to the front IO0 through 3 pins. Select your channels and click **OK**.

5. Click **Apply** to save your new channel. Simply creating the channel does not actually set the digital channel. To do this you must set the channel to something. It does not matter what. The simple way to do this is to right click on the new channel row and select Set Value, enter any value and click OK. You can also use components or sequences to set this channel.

Counter: The U12 supports a single 1 Mhz counter numbered 0. Since there is only one counter, all data comes back on channel number 0. By default, the counter is reset every time it is read. There is a channel command associated with this I/O type that allows you to keep the counter from resetting.

Example - To read the counter, simply follow the example provided for the analog input, but use the Counter I/O type instead. Only channel 0 is supported.

Example - Setting the counter to not reset:

1. On the row of the counter channel that you created, under Quick Note / Special / OPC for the channel a button should appear along the right side of the cell with three dots (...). Click on this button.

2. The Channel Parameters window will appear. From the drop down at the top of the window, select **Reset**.

3. The only parameter for this command is **Reset?.** Select Yes or No then click **OK**.

4. Click **Apply** to save your changes to the channel. The effect of the change is immediate.

Temperature / Humidity Probe (EI-1050): The EI-1050 is a digital temperature / humidity probe. To query this

device, create two channels, both with I/O type "Special", one with channel #0, and one with channel #1. Channel #0 is the temperature in degrees C, while channel #1 is the humidity. You must put a timing on channel #0 as this triggers the read of both sensors.

Streaming Analog Input: The U12 has the ability to stream its inputs instead of being polled for data. When streaming, the internal clock of the LabJack determines when data is acquired and the data is passed back to DAQFactory in blocks. This allows much faster data rates than would be possible using software polled mode described above. Unfortunately, when the LabJack is streaming you cannot do anything else with that LabJack. If you have two LabJacks you can work with the channels on the second LabJack, but on the streaming LabJack you should have the Timing parameter for all its channels set to 0. There are three different modes of streaming:

Continuous Streaming: This mode causes the LabJack to continuously stream the requested inputs until manually stopped. It is initiated with the Start Stream channel command, and stopped with the Stop Stream channel command. The maximum data rate is 1200 samples per second.

Example:

1. Starting from a new document, go to the channel table by clicking on **CHANNELS:** in the workspace.

2. Click the **Add** button to add a new row. Call the new channel **StartStream**, device type **LabJack_U12**, device number **0**, I/O type **Command**, channel #**0**.

3. Under Quick Note / Special / OPC for the channel a button should appear along the right side of the cell with three dots (...). Click on this button.

4. When the parameter window appears, select **Start Stream** from the drop down at the top.

5. Four parameters will appear in the table:

a. **Stream Channels:** this takes a comma delimited list of channels to stream. You can enter up to four channels. For this example, enter **8,9,10,11**. This will stream the 4 differential channels.

b. **Stream Gains:** this takes a comma delimited list of gain numbers (0-7) for each channel listed. You can leave this blank to use the default gains. This does not apply to single ended channels. For this example, leave it blank.

c. **Stream Scan Rate:** enter the desired scan rate. A scan is the reading of all channels listed under Stream Channels. Since the maximum rate is 1200 samples per second, the maximum scan rate is 1200 / number of channels read. So if you are streaming four channels, the maximum scan rate is 300. For this example, enter **100**.

d. **Stream Counter:** you can optionally stream the counter reading. This uses three of the four slots available for channels, so you can only stream one other A to D channel when streaming the counter. Data comes in under the Counter I/O type, channel 0. For this example, select **No**.

6. Click **OK** to close the parameters box, then click Duplicate to duplicate the channel you just created. Give the new row the name **StopStream**. Leave all the parameters as they are, but click on the Quick Note / Special / OPC button again to display the parameters window.

7. This time, select **Stop Stream**. There are no parameters for this command, so hit **OK**, then **Apply** to save your two new channels.

8. Simply creating these channels does not actually perform streaming. To do this you must set the channels to something. It does not matter what. The simple way to do this is to go to the Command/Alert window and in the small box type in **StartStream = 0**. This should start the streaming. The LED on your LabJack will flash rapidly. You can also use components or sequences to set this channel and start streaming.

9. Even though the streaming is running, the data is going no where. We have to create channels to accept the data. In the channel table, click **Add**. In the new row, call the new channel **AI8**, device type **LabJack_U12**, device number **0**, I/O type **A to D**, channel #**8**. Set the Timing to **0**. This is important or DAQFactory will try and read the channel in software polled mode and this will stop the streaming.

10. You can optionally create channels for 9, 10 and 11 which are also streaming. When done, click **Apply**.

11. To quickly see the data coming in, click on the + next to **CHANNELS:** in the workspace, then click on the AI8 channel. This will display the channel view for that channel. Click on the **Graph** page to view a graph of the incoming data, or optionally, click on **Table** to see the readings in tabular form. You can now use the rest of the DAQFactory page components, such as the 2D graph, to better display your data.

Burst Streaming: This mode causes the LabJack to stream for a preset amount of time. This mode allows for faster data rates, up to 8192 samples per second, but can only take 4096 samples at a time. It is initiated with the Burst Mode channel command.

Example:

1. Starting from a new document, go to the channel table by clicking on **CHANNELS:** in the workspace.

2. Click the **Add** button to add a new row. Call the new channel **StartBurst**, device type **LabJack_U12**, device number **0**, I/O type **Command**, channel #**0**.

3. Under Quick Note / Special / OPC for the channel a button should appear along the right side of the cell with three dots (...). Click on this button.

4. When the parameter window appears, select **Burst Mode** from the drop down at the top.

5. Six parameters will appear in the table:

a. **Burst Channels:** this takes a comma delimited list of channels to burst stream. You can enter up to four channels. For this example, enter **8,9,10,11**. This will stream the 4 differential channels.

b. **Burst Gains:** this takes a comma delimited list of gain numbers (0-7) for each channel listed. You can leave this blank to use the default gains. This does not apply to single ended channels. For this example, leave it blank.

c. **Burst Scan Rate:** enter the desired scan rate. A scan is the reading of all channels listed under Stream Channels. Since the maximum rate is 8192 samples per second, the maximum scan rate is 8192 / number of channels read. So if you are streaming four channels, the maximum scan rate is 2048. For this example, enter **1000**.

d. **Burst Number of Scans**: enter the desired number of scans to perform. The maximum number of samples is 4096, so if you are sampling 4 channels per scan, the maximum number of scans is 1024. For this example, enter **100**.

e. **Burst Trigger:** You can optionally have the LabJack use one of the IO pins as a trigger for the burst. In this row, you can select which IO pin to use, or select None or leave blank for no triggering. For this example, just leave this field blank.

f. **Burst Trigger State:** If you specified a burst trigger, you will need to specify whether you want to trigger when the indicated IO pin goes high (5V), or low (0V). For this example, leave the field blank.

6. Click **OK** to close the parameters box. Now we need to create channels to receive the data. In the channel table, click **Add**. In the new row, call the new channel **AI8**, device type **LabJack_U12**, device number **0**, I/O type **A to D**, channel #**8**. Set the Timing to **0**. This is important or DAQFactory will try and read the channel in software polled mode and this will stop the streaming.

7. You can optionally create channels for 9, 10 and 11 which will also burst stream. When done, click **Apply**.

8. In this example, we will create some screen components for triggering the burst and graphing the data on AI8. Go to the page view by clicking on **Page_0** in the workspace. The page view will be a blank white screen.

9. Right click in the upper left corner of the white space and select **Buttons-Button**. Right click on the new component and select **Properties...**

10. In the Text property, enter **Burst**. Then go to the **Action** page and select **Toggle Between**. Next to **Action Channel** enter **StartBurst**. For the two toggle between parameters, enter **0** and **0**. It does not matter what we set the StartBurst channel too, so we are just going to set it to 0 every time the button is clicked. Click **OK**.

11. Right click in another area of the screen and select **Graphs - 2D Graph**. You may want to resize the graph that appears by holding down the Ctrl key and dragging one of the black squares surrounding the graph.

12. Right click on the graph and select **Properties...**

13. Under Y Expression enter **AI8** and then click **OK**.

14. Now click on the button you created to start burst mode. The burst mode only lasts 0.1 seconds, so it will go quick. The graph should update, but might be out of range. Right click on the graph and select **AutoScale - Y axis** to fix this.

PreTrigger Streaming: This allows you to run in stream mode, but only log data around a certain event, either a threshold crossing for an analog channel, or a digital channel going high or low (or both). It is initiated with the PreTrigger Stream channel command and can be stopped with the Stop Stream channel command. Since it uses continuous streaming, the maximum data rate is 1200 samples per second. The setup is very similar to burst and stream. Here are the parameters of the channel command that are different than the Start Stream:

Trigger Channels: takes a comma delimited list of channels to monitor. These channels must be among the channels listed in Scan Channels.

Trigger Thresholds: takes a comma delimited list of voltage readings above or below which the trigger will have said to occur. There should be one threshold for each Trigger Channel listed.

Reset Threshold: the value above or below which the trigger is reset and another trigger can occur. For no historesis, set these values to the same values as the Trigger Thresholds.

Trigger Types: determines whether the trigger occurs above or below the threshold.

Digital Trigger Channels: takes a comma delimited list of digital channels to use as triggers. Only channels 0-3 can be used (IO0-IO3).

Digital Trigger States (trigger on 1 or 0): takes a comma delimited list, one item for each digital trigger. If the item is 1, then the digital trigger occurs when the signal goes high. If the item is 0, then the digital trigger occurs when the signal goes low.

Pretrig Samples: the number of samples before the trigger event occurs to log. Can be 0.

PostTrig Samples: the number of samples after the trigger event occurs to log (including the trigger value). Must be >= 2.

Immediate Retrigger: If yes, then a reset followed by a trigger while the PostTrig samples are being logged results in the posttrig sample counter being reset to Posttrig Samples. If no, then a retrigger during posttrig acquisition does nothing. Digital triggers do not need to be reset, a digital signal that remains in its trigger state will continually trigger.

Display Incoming?: If yes, then data from the stream will be output to channels of I/O type Special throughout the PreTriggering. This allows you to view the data being taken at all times. Simply create new Channels with I/O type Special, and channel numbers matching the A/D channels.

Some notes about PreTriggering:

• Data is not logged until all the PostTrig samples are taken.

• If the type is **Above**, the trigger occurs when reading > threshold and resetting occurs when reading < reset threshold. (and vice-versa)

• You can trigger off of any, or all of the channels being streamed. Each trigger must be independently reset before that particular trigger will refire. However, if a trigger on one channel fires, but is not reset, then a trigger on another channel fires, the second trigger will be recognized.

• Digital triggers do not need to be reset. They will continually refire if they remain in the trigger state. This allows you to use digital lines to start and stop logging.

NOTE: while doing high speed acquisition, you cannot do any other I/O on the LabJack. Any attempt will stop the high speed acquisition. If you have multiple LabJacks, you can perform high speed acquisition on only one LabJack. You can however, perform I/O on the other LabJacks while the high speed acquisition is occurring.

Note: when taking and logging data faster than 20hz you'll need to set the Alignment Threshold parameter of your logging set to 0 to avoid missing data points in your logging set.

Sequence optimization channel commands:

The following channel commands are typically used in sequences to utilize the LabJack in the most efficient manner:

Read Digital Block: Reads a block of digital channels in one call to the LabJack. Put a comma delimited list of desired digital channels (0-19) in the channels parameter.

Write Digital Block: Writes a block of digital channels in one call to the LabJack. Put a comma delimited list of desired digital channels (0-19) in the channels parameter, and a corresponding comma delimited list of 0's or 1's to output to each channel in the Values parameter.

Write Digital Block Single Value: Writes a block of digital channels in one call to the LabJack. Put a

comma delimited list of desired digital channels (0-19) in the Channels parameter. The Value parameter takes a single value that is bitwise parsed to each digital output channel. The first channel in the list is the LSB, the last, the MSB. So, if you put 0,3,5,7 in the Channels parameter, and 9 in the Value parameter, you would get a 1 output to channels 0 and 7, and a 0 output to channels 3 and 5. This command is very useful if you put the word "Value" in for the Value parameter. Then the outputs are set to whatever you set this channel command to.

Read Analog Block: Reads a block of analog channels in one call to the LabJack. Put a comma delimited list of desired analog channels (0-11) in the Channels parameter.

Write Analog Block: Writes a block of analog channels in one call to the LabJack. Put a comma delimited list with two values in the Values parameter. The first value in the list is output to AO0, the second to AO1.

17.9 LabJack M (T7)

This device driver communicates with the LabJack devices powered by their LJM driver. Please refer to the LJM library documentation on the LabJack website for details.

To use with channels, select LabJackM from the Device Type. The D# should correspond to the identifier for your device. You can use 0 or ANY in place of LJM_idANY. Unlike other devices, you can put IP addresses directly in the D# column for the LabJackM. DAQFactory opens the device using the LJM_Open() function (see LJM docs) with a device type of LJM_dtANY and connection type of LJM_ctANY.

The I/O types are pulled from the LJM driver table provided by LabJack and will vary depending on the version of the LJM you have installed. Only core I/O types are displayed. Please refer to their documentation for the meaning of each type.

For I/O types with # in them, you will need to provide a channel number as well. Usually the range of valid channel #'s is displayed in parenthesis after the # in the I/O type. For example, AIN#(0:254) would require you to specify a number from 0 - 254 depending on which analog input you wanted to read.

The I/O type list is limited to what LabJack identifies as core type. If you want to create a channel with a type that isn't in the list, you can select the "Special" I/O type, and then type in the name in the Quick Note / Special / OPC column. If the type you want has a # in it, you can either substitute the desired channel right there in the Quick Note (for example: AIN1_EF_READ_A), in which case the channel # column is not used. You can also leave the # and the channel # column is substituted for the #. So if you put AIN#_EF_READ_A, and then a Channel # of 4, the LabJack driver would use AIN4_EF_READ_A. This is better if you want to read the same type with different channel numbers, as you can just duplicate the channel and change the Channel #.

Note that you need to be consistent with your IDs and should not use a different ID to address the same device. This would include accessing a device with 0 for LJM_idANY and then accessing it explicitly using its ID. If you want to change how you address a device, you will likely need to restart DAQFactory after the change is made.

NOTE: DAQFactory currently does not distinguish between I/O types that are traditionally output only and inputs, for example DAC's, and will default the Timing to 1 for all I/O types. Make sure and set the Timing to 0 for any output channels or DAQFactory will send a command to read from that I/O type at the Timing interval.

DAQFactory also supports some LabJack M functions, all start with device.LabJackM. All will throw an error if an error occurs, which you can then catch using try/catch blocks.

LJM_eReadName(ID [string], Name [string]): reads a particular input specified by the Name. ID is the ID of the device (which would include the IP for Ethernet connections). DAQFactory will automatically open the device with the given ID if it is not open already. You do not need to call any sort of open function to open the device. Name follows the standard LJM nomenclature for the LJM_eReadName() function (see the LabJack M documentation from LabJack)

LJM_eWriteName(ID [string], Name [string], Value [number]): this is almost identical to LJM_eReadName, except writes the given value to the specified named addressed. This correlates with LJM_eWriteName(), though like LJM_eReadName, DAQFactory takes care of opening the device and managing handles.

LJM_eStreamStart(ID [string], Name(s) [string or array of strings], scanRate [number], scansPerRead [number], 1) : starts a stream on the given device. Name(s) is either a string with the desired value to stream, or an array of strings containing the names of the values to stream. Note the last parameter must be 1. This is to

allow for a future version of this streaming that automatically reads the data like the LabJackUD.

LJM_eStreamRead(ID [string]): returns data from a stream that is running on the specified device. The stream must be started using LJM_eStreamStart. This function returns an object. That object contains arrays of data for each name specified in LJM_eStreamStart, plus member variables for ljmScanBacklog and deviceScanBackLog. So, for example, if you were streaming AIN0:

private dataIn = device.LabJackM.LJM_eStreamRead("ANY")

dataIn.AIN0 would have an array of data (based on what the LabJack returns)
dataIn.ljmScanBacklog would have the scan backlog
dataIn.deviceScanBacklog would have the device scan backlog

LJM_eStopStream(ID [string]): stops a stream running on the specified device.

Here's some sample script for setting up and running a stream:

```
private scanRate = 10000
private scansPerRead = 500
device.labjackM.LJM_eStreamStart("ANY", {"AIN0", "AIN1"}, scanRate, scansPerRead, 1)
private dataIn
private data
private st = systime()
global backlog
while(1)
   dataIn = device.labjackM.LJM_eStreamRead("ANY")
   data = insertTime(dataIn.data.AIN0, st, 1 / scanRate)
   channelA.AddValue(data)
   data = insertTime(dataIn.data.AIN1, st, 1 / scanRate)
   channelB.addValue(data)
   st += scansPerRead / scanRate
   backlog = dataIn.ljmscanBacklog
endwhile
```

There is a bit here, but only a little of it needs to be changed to stream other inputs. But first, you'll need to create channels to receive the data. In this case, they are named ChannelA and ChannelB. We recommend setting them up as standard input channels for the LabJackM (in this case, D# "ANY", I/O Type "AIN#(0:254)", Chn #'s 0 and 1. Make sure the Timing on the channels are 0 though since you can't poll and stream a channel at the same time.

Here's a line by line breakdown:

```
private scanRate = 10000
```

Sets the scanRate. We store this in a variable because we need it in several locations to calculate times.

```
private scansPerRead = 500
```

Sets the scans to acquire with each read. Please see the LJM documentation for the best choice of value for this parameter based on scan rate.

```
device.labjackM.LJM_eStreamStart("ANY", {"AIN0", "AIN1"}, scanRate, scansPerRead, 1)
```

Starts the stream on the first found device "ANY", for names "AIN0" and "AIN1".

```
private dataIn
private data
private st = systime()
global backlog
```

Set up some variables to receive data. "st" is used to time stamp the data. "backlog" is global so you can view the backlog from a page.

```
while(1)
```

This sequence will stay running while the device is streaming to collect data and put it into the channels. While(1) means loop forever.

```
dataIn = device.labjackM.LJM_eStreamRead("ANY")
```

Read a block of data and store it in our private variable.

```
data = insertTime(dataIn.data.AIN0, st, 1 / scanRate)
channelA.AddValue(data)
```

Insert time into the array of values starting with st, and incrementing by 1/scanrate. Then take the result and stuff it into channelA. Repeat for ChannelB.

```
st += scansPerRead / scanRate
```

Increment st for the next round. Each round returns scansPerRead values, so the time increment between loop iterations is simply scansPerRead / scanRate.

```
backlog = dataIn.ljmscanBacklog
```

Save the backlog in a global variable so it can be accessed elsewhere.

```
endwhile
```

Loop around and repeat.

The sequence will remain running while streaming is running. To stop streaming, stop the sequence and call LJM_eStreamStop:

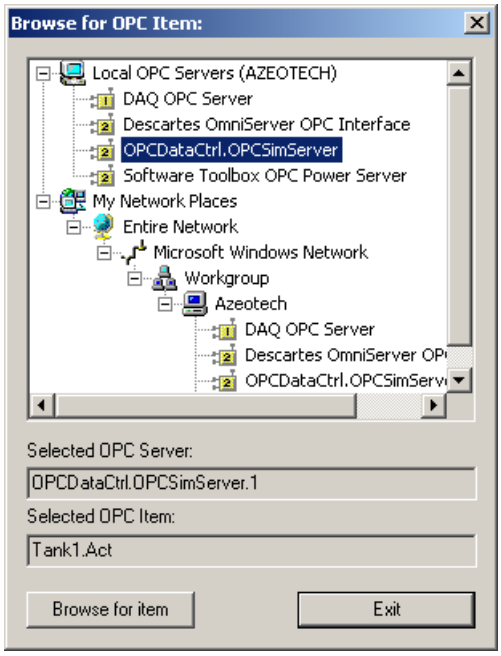
```
endSeq(startStream)
device.labjackM.LJM_eStreamStop("ANY")
```

17.12 OPC device

The OPC device allows you to communicate with any OPC server either locally or on a remote computer. An OPC server is a piece of software that translates the proprietary protocols used by PLC's and other devices into a standard language called OPC that DAQFactory can communicate. OPC servers are not included with DAQFactory, but a large selection is available separately from AzeoTech. DAQFactory also has some native support for PLC's and other devices included in the Connectivity pack. This includes the generic Modbus protocols and many specific manufacturer protocols.

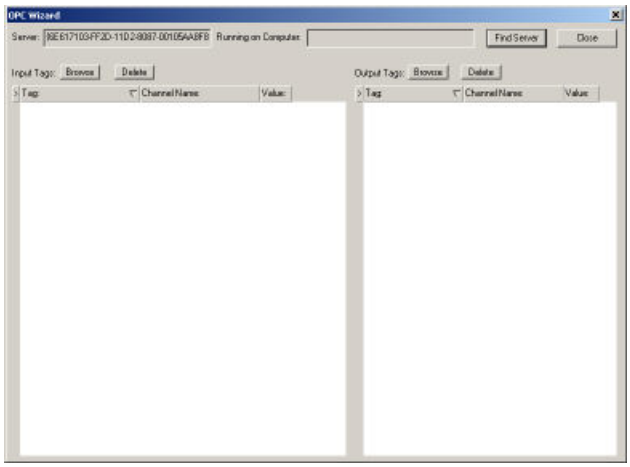
Device Configurator:

DAQFactory offers a configurator for very quickly selecting OPC tags and creating channels to access these tags from within DAQFactory. You can access the configurator by selecting **Quick – Device Configuration...** from the main menu. If you do not have any OPC channels yet, you will be prompted with a window to find your OPC server:



If you are trying to find a remote server, Windows sometimes does not keep an accurate listing of the available networked computers. If the remote computer you need does not appear, you can click on the Add Host button to manually enter the name of the remote host.

After selecting, the configurator window will appear:



At top of the wizard dialog is displayed the currently selected server. You can change the selected server by clicking the **Find Server** button to the right. In order to support remote connections, the server is indicated using a GUID. Although not that human friendly, this unique identifier works well over a network.

The rest of the window is split into two tables that display the input tags and output tags for the currently selected server. Above each table is the **Browse** button that allows you to browse for tags on the currently selected server. This will open up another dialog box with the tags on the server. Simply double click on the desired tags and they are added to the list. You do not need to close the browse dialog box with each tag. You can add multiple tags very quickly this way. When a new tag is added a channel name is produced from the tag name by removing invalid characters (such as the period that is typically in OPC tags). You can use this default channel name, or enter your own in the table.

For your convenience, the most recently polled value for each Channel is displayed in the Value column of the Input Tags table. You can also set output tags by entering the output value in the Value column of the Output Tags table.

Device Details:

It is strongly recommended that you use the device configurator for specify OPC tags. There are a few details that need to be mentioned though about the channels that are created:

• By default, all input tags are setup in asynchronous mode. This means that the OPC server only sends a new value to DAQFactory when that value changes up to a maximum rate of 10 updates per second. If the value does not change, no new data is created. This may result in a blank space in the data log. To avoid this, specify Duplicate Last Value in your logging set.

• Async tags also can act as output tags. Simply set the channel to a value and the value will be sent to the OPC server as an output.

• Async mode is definitely the preferred mode as it is more efficient. For fast changing systems where you do not need 10 hz update speeds, you may want to set your channels into synchronous mode. In this mode, the Timing parameter of each channel determines when the value is read. The value is read whether it has changed or not. You can switch any channels to sync mode by changing the I/O type of the channels. Remember to set the Timing parameter as well to the desired polling interval.

• For string tags, you will need to change the I/O type of the channel to String Read. String reading is async only.

• For array tags, you will need to change the I/O type of the channel to Array Read. Array reading is async only.

17.15 Test device

The Test device does not communicate with any real hardware. It is provided to allow you to try out DAQFactory on computers without any real hardware. It can also be used as a place to store variables or flags for sequences that need to be controlled over a network.

The A to D, Counter, Special, and Dig In I/O types for this device generate sine waves with a period proportional to the channel number.

The Spectrum I/O type generates a 500 point spectrum with 3 different sized gaussian peaks. The amount of noise that is placed on top of the spectrum is determined by the channel number.

The output I/O types and String I/O type simply bounce back their set points. Because of this, you can use the output Channels of the Test device as a place to store variables or flags for use in sequences that need to be adjustable from a remote copy of DAQFactory. Simply create a Test device output channel and set its value like you would set a real D to A or Digital Out.

The A to D channel also has a channel parameter that allows you to try high speed mode. To access the parameter, set up an A to D channel as you normally would, then go to the last column of the table (or what is normally the Quick Note field in the Channel view, just above the Notes). Click on the button to the right of this field. This will open a dialog window where you can select the Channel parameter and enter in the parameters.

For the A to D channel, the Channel parameter sets the number of samples that will be generated. The driver will use the timing of the channel to spread these samples out. So, for example, if you specified 1000 samples and your timing was 1 second, you would get 1000 samples per second, each separated by a millisecond.

18 Error Code Reference

18 Error Code Reference

C1000

The name provided was not found:

1) Check your spelling, check for typos.

2) If you are not specifying the connection, make sure the default connection is correct.

C1001

The channel name provided was not found:

1) Check your spelling, check for typos.

2) If you are not specifying the connection, make sure the default connection is correct.

C1002

The name provided was not found:

Check your spelling, check for typos.

C1003

The objects you have edited are on a connection that has become disconnected. Since DAQFactory cannot communicate with the remote site, your changes cannot be saved yet. Check the remote system and cabling and retry. More info...

C1004

An error occurred while trying to set the value. This is a general catch all error for something unexpected and may be due to hardware failure or system instability. You should probably save your work just in case and proceed with caution.

C1005

The variable name specified was not found and the data cannot be retrieved.

1) Check your spelling, check for typos.

2) Did you mean Private. instead of Var.? Static. instead of Registry.? etc.

More info...

C1006

A conversion was specified for a channel that does not exist. Most likely, the conversion was deleted without updating the channels to which it applied. Check your channels for deleted conversions. More info...

C1007

A problem occurred trying to connect to your email server. The error varies and is provided at the end of the message. Check your email server setting, and your internet connection. Also make sure your username and password are correct and that you've selected the correct authentication method. More info...

C1008

A problem occurred trying to send the email message. The response from the server is provided. The action to fix this error depends on the response, but typically is either related to the settings (username / password / authentication method) or a problem with the server. More info...

C1009

Most of the System. functions that display windows can only be called from the main application thread. The main application thread is the thread that the DAQFactory user interface runs in. Sequences started with Begin Sequence run in their own thread and cannot open any windows. Events for channels and PID loops also usually run in their own threads and cannot open windows. The only reliable place to call the System. functions that open windows is from the Quick Sequence action of a component. More info...

C1010

There was a problem trying to open the data source for ODBC data logging. The most common mistake is to provide the file name of your database for the data source and not the ODBC data source name defined in the Windows ODBC manager. More info...

C1011

A problem occurred trying to create the table for data logging in your ODBC database. The possible causes are database specific. The most common is that the database is opened exclusive in another application and DAQFactory cannot open it for modification.

C1012

A problem occurred trying to access the table for data logging in your ODBC database. The possible causes are database specific. The most common is that the database is opened exclusive in another application and DAQFactory cannot open it for modification.

C1013

An error occurred while trying to log. The exact error message is also provided. Common problems include trying to log to a file that is open exclusively in another application, logging to a ODBC table that already exists with the wrong field names, or the disk being full.

C1014

An error occurred trying to start up the thread used for the PID loop. This is usually caused by lack of memory or resources and is a good sign that you are overdriving your system. Try running DAQFactory with fewer other applications running.

C1015

Most of the System. variables that affect the display of windows can only be called from the main application thread. The main application thread is the thread that the DAQFactory user interface runs in. Sequences started with Begin Sequence run in their own thread and cannot open any windows. Events for channels and PID loops also usually run in their own threads and cannot open windows. The only reliable place to call the System. variables that affect the display of windows is from the Quick Sequence action of a component. More info...

C1016

The System.MessageBox function requires one of the following types:

"Help", "AbortRetryIgnore", "OKCancel", "RetryCancel","YesNo", or "YesNoCancel"

These types are case sensitive. More info...

C1017

There was insufficient resources to display the message box. Try running DAQFactory with fewer other applications running.

C1018

Must be either NoLogin, AuthLogin, or LoginPlain

The specified authentication type for the email server must be one of the three options listed. More info...

C1019

An internal error occurred that was caught by DAQFactory. Unfortunately, what the error was cannot be determined, but it must be something rare or another message would be displayed. You should save your work under a different file name just in case the file is corrupted and restart DAQFactory.

C1020

Alarm logging will not occur.

There was a problem trying to open the data source for ODBC alarm logging. The most common mistake is to provide the file name of your database for the data source and not the ODBC data source name defined in the Windows ODBC manager. More info...

C1021

A problem occurred trying to create the table for alarm logging in your ODBC database. The possible causes are database specific. The most common is that the database is opened exclusive in another application and DAQFactory cannot open it for modification.

C1022

A problem occurred trying to access the table for alarm logging in your ODBC database. The possible causes are

database specific. The most common is that the database is opened exclusive in another application and DAQFactory cannot open it for modification.

C1023

An error occurred while trying to open the alarm log file. The exact error message is also provided. Common problems include trying to log to a file that is open exclusively in another application, logging to a ODBC table that already exists with the wrong field names, or the disk being full.

C1024

This is a generic error that occurs when DAQFactory is unable to broadcast a command over the network. This can be caused by network failure or by internal errors. If the message persists, check you network connection, then save and restart DAQFactory.

C1025

An error occurred in the thread that sends commands over the network. The thread will be restarted but this may fail. If you find that commands are not being received remotely, you should restart DAQFactory. You may want to consider restarting anywhere.

C1026

(does not affect proper running of DAQFactory, but may indicate a further problem)

The heartbeat thread simply sends a small packet over the network so remote copies of DAQFactory can tell that the connection is still established. There was an error in the thread that runs the heartbeat and the heartbeat had to be stopped. Remote copies of DAQFactory will probably still receive data. This probably won't affect anything, but may be an indication of other problems, such as lack of memory, resources, or network problems.

C1027

The was a problem averaging channel data. Most likely this comes from an invalid data point, such as infinity that can't be averaged. Your data at this point in time is invalid. You should check you device and make sure it is working properly.

C1028

The license you have purchased only supports a single device driver at a time. Check your channel table and make sure the Device Type column lists only one device for all your channels.

C1029

The license you have purchased does not support the particular channel parameters you specified. Check your channel table.

C1030

There was a problem with a particular device driver file. Try downloading the latest version of the device driver and reinstalling.

C1031

The license you purchased only supports certain devices. At least one of your channels specifies a device that is not supported. Check your channel table and any sequences that may reference devices.

C1032

Device drivers are made up of two files, one with a .dll extension and one with a .dds. DAQFactory generates this error if you have different versions of these two files. This usually does not cause any problems, but if you find expected features missing you should download and install the device driver again.

C1033

This occurs in one of three situations:

1) You loaded a document from another computer that references devices that are not installed on this system.

2) The .dds file for the device driver exists, but the .dll file is missing.

3) Dependency .dll files are missing. This usually occurs when you install the DAQFactory device driver but have forgotten to install the manufacturers drivers. DAQFactory relies on these drivers and therefore requires them. We cannot distribute them with DAQFactory because of copyright issues as well as for version control.

C1034

With any out of memory error, you should try running DAQFactory with less applications running. You can also try lowering the resolution and/or color depth of your screen and restarting DAQFactory. Shortening your channel's history lengths will also free up memory.

C1035

Sequences typically run in their own thread. For some reason DAQFactory was unable to start a thread for a sequence. This is typically caused by lack of memory or resources. Try running DAQFactory with fewer other applications running. You can also try lowering the resolution and/or color depth of your screen and restarting DAQFactory. Shortening your channel's history lengths will also free up memory.

C1036

This is a generic catch-all error for any error that may occur in a sequence that does not have its own error. This is most likely caused by problems with the device or device driver, but could be caused by lack of memory or resources.

C1037

Please save work under a different name and restart DAQFactory.

If this error occurs, something very unusual occurred and rather than crashing, DAQFactory has caught the error and generated this message. You should save your work under a different name just in case there is file corruption and restart DAQFactory. Rebooting probably is not a bad idea either.

C1038

If you specify Timing parameters for your channels that are too small and your computer cannot keep up, DAQFactory will detect the backlog and reset. If this just occurs once, then there was probably just a temporary increase in CPU usage caused by starting another application or something similar. In this case you are near the limit, but not over it. If this recurs at a relatively constant interval then the device you are communicating with cannot maintain the data rate specified. For example, the LabJack U12 takes about 16ms to acquire a data point. If you specify a timing interval of 0.01 for a LabJack channel, you will receive this error every 10 seconds or so. You should adjust your Timing values accordingly.

C1039

An unknown error occurred while trying to access and subset channel data. Check your subsetting.

C1040

The channel that you have requested data from does not have any values yet. This typically occurs in sequences that start when the document loads and reference input channels. The sequence starts before data has been acquired on the referenced channels and the sequence throws this error. To fix this, try adding a little delay to the beginning of the sequence to ensure acquisition has time to start, or use the IsEmpty() function to check for valid data before actually trying to use it.

C1041

If this error occurs while loading a new document, or starting a new document then things may simply not have shut down properly. You may want to restart just in case, but this is probably not required.

If this error occurs during normal operation of your document, then there is a memory issue with your system. Unfortunately your document is probably corrupted and you will have to revert to a saved document when you restart. This is an indication of a fundamental operating system problem.

C1042

Some sort of uncaught error occurred while trying to compile your sequence. This is probably an uncaught problem with the compiler. You should email support@azeotech.com with a copy of the sequence you were trying to compile and will determine the problem rapidly.

C1043

In your sequence an extra endwhile was found without a matching while() statement.

C1044

In your sequence a while() was found without a matching endwhile.

C1045

In your sequence an extra endfor was found without a matching for()

C1046

In your sequence, a for() statement was found without a matching endfor

C1047

In your sequence, an endif was found without a matching if() statement.

C1048

In your sequence, an if() statement was found without a matching endif.

C1049

In switch / case constructs, you must list all your case statements first before listing the default.

C1050

In the switch construct, you can have many case statements, but you can only have one default statement.

C1051

An endcase statement was found without a matching switch statement.

C1052

In your sequence, a switch statement was found without a matching endcase.

C1053

In your sequence, an endcatch statement was found without a matching catch statement.

C1054

In your sequence a catch statement was found without a matching endcatch.

C1055

There was a problem with the formatting of a line in your sequence.

C1056

A statement or function was found without the expected open parenthesis. `Function()`

C1057

A comment ("//") was found in the middle of a parenthesis, subset, or array notation.

C1058

An open parenthesis was found without the matching close parenthesis.

C1059

An open quote was found without the matching close quotes.

C1060

Most statements require a certain number of paremeters.

C1061

An open bracket was found without the matching close bracket.

C1062

This error occurs when a line in a sequence starts with something other than a letter. All statements start with a letter as do all channels and functions, so with the exception of a comment, all sequence lines must start with a letter.

C1063

An expression was found after a function call.

C1064

An expression was found after the decrement operator: For example: `MyChannel-- + 3` is invalid

C1065

An expression was found after the increment operator: For example: **MyChannel++ * 3** is invalid

C1066

An operator was specified that doesn't exist. For example **MyChannel ^* 3**

C1067

The expression parser was called with an empty expression. An empty expression is valid in many places, but not in all places.

C1068

'Value' can only be used in conversion or graph trace expressions

The term **Value** is a special term that is used in conversion where a channel should be substituted, or in graph trace expressions where the X value should be substituted. This error occurs when you try an use this term in other expressions.

C1069

You tried to subset an array with no values in it.

C1070

Some operators require one parameter, for example: **MyChannel++**. Others require two, for example: **MyChannel + 3**. This error occurs when the second parameter is missing: **MyChannel +**

This can also occur with internal functions.

C1071

The given function was provided more parameters than it needs.

C1072

This occurs when the number of open { do not match the number of close }.

C1073

Example: **MyChannel[3+(2*3))]**

C1074

For example: **MyChannel[3,5}]**

C1075

For example: **MyChannel[3+(2*5]**

C1076

Example: **MyChannel[3,{3,2]**

C1077

Example: **Sin(MyChannel[0]])**

C1078

For example: **Sin({2,5})**

C1079

For example: **Sin(MyChannel[0)**

C1080

For example: **Sin({2,5)**

C1081

For example: **Var.x = MyChannel[3]]**

C1082

For example: **Var.x = 3 + (2 * 3))**

C1085

For example: **Var.x = MyFunction + 2 * 3**

C1086

The proper number of parameters was provided, but one of the parameters had an empty value.

C1087

In older releases of DAQFactory you could subset relative to the current time by specifying negative subset values. This feature has been removed. Instead, use SysTime()-x. So before if you had **`MyChannel[-2,-5]`**, replace this with **`MyChannel[SysTime()-2,SysTime()-5]`**

C1088

There was an error while performing a mathematical function.

C1089

An error occurred that does not fit into any other category. Check the syntax.

C1090

The ShiftTime() function requires an array that has time values to shift.

C1091

The FormatDateTime() function requires a numeric date/time in seconds since 1970 which is the DAQFactory standard way to store time.

C1092

You used the Remove() function on a string with nothing in it and therefore nothing to remove.

C1093

You cannot search an empty string.

C1094

Fill(Value, Array Size). The second parameter, which determines how many elements to create must be greater than 0.

C1095

For example you cannot Concat() an array with 5 rows and 10 columns to an array with 7 rows and 12 columns. The Concat() function combines arrays along the rows dimension. The total rows in each row can be different, but the column and depth count must be the same.

C1096

A generic error occurred while trying to calculate the temperature. Most likely an invalid voltage or CJC was provided.

C1097

Many functions require numbers. Most will take a string operator and convert it to a number if necessary. Other will generate this error instead.

C1098

This occurs when DAQFactory has an array with a different size from its reported size. If this persists, contact tech support with the actions taken.

C1099

The given operator or function does not exist.

C1100

AscA requires a valid string.

C1101

ChrA() requires an array of values

C1102

SortTime() sorts an array based on the time of each data point. This requires an array with time values.

C1103

The boxcar functions only work on numeric values. More info on Boxcar functions...

C1104

The boxcar functions only work on single dimensional arrays, or an array with multiple rows, but only one column and one depth. Use subsetting and the Transpose() function to boxcar other dimensions. More info on Boxcar functions...

C1105

More info on Boxcar functions...

C1106

This is an internal error. BoxcarMinMax is a function used to optimize graphing and should only be called when values have time associated.

C1107

More info on the Smooth() function...

C1108

The Smooth() function only works on single dimensional arrays, or an array with multiple rows, but only one column and one depth. Use subsetting and the Transpose() function to Smooth other dimensions. More info on the Smooth() function...

C1109

More info on the Smooth() function...

C1110

More info on Histogram()...

C1111

More info on Histogram()...

C1112

GetTime() returns an array with the time associated with a value.

C1113

More info on Percentile()...

C1114

More info on Percentile()...

C1115

More info on Percentile()...

C1116

More info on curve fitting...

C1117

An error occurred trying to perform the curve fit. More info on curve fitting...

C1118

By the vary nature of the binary logging format, you cannot use it to log string data. You should use ASCII or ODBC database to log string data instead. More info...

C1119

In Safe Mode, no acquisition, PID loops, logging or sequences will start. You must leave safe mode (File-Leave Safe Mode) before you can start any sequences, PID loops, or logging sets.

C1120

Logging will still continue.

There was an error trying to create the logging header file. This is most likely because the file is read-only, is open in another application, or the disk is full. More info...

C1121

There was a generic error while performing the File.Move() function. More than likely the destination already exists, or the source file does not exist. More info...

C1122

A generic error occurred while trying to do a File.Copy() function. More than likely, the source file does not exist. More info...

C1123

There was a generic error trying to retrieve the disk space. Most likely you tried to retrieve the disk space on a removable disk that has been removed. More info...

C1124

More than likely the directory still has files in it. More info...

C1125

More than likely, the directory already exists. More info...

C1126

The logarithm of a value <= 0 does not exist and therefore is invalid.

C1127

To help avoid hanging your computer when you accidentally write an infinite loop, DAQFactory by default limits the maximum number of sequence steps that can be executed in a second. If this is exceeded, this error occurs. This feature can be adjusted or disabled from **File-Preferences...** More info...