

DataAcq SDK

User's Manual

DASDK-900-01 Rev. A / January 2005

WARRANTY

Keithley Instruments, Inc. warrants this product to be free from defects in material and workmanship for a period of 3 years from date of shipment.

Keithley Instruments, Inc. warrants the following items for 90 days from the date of shipment: probes, cables, rechargeable batteries, diskettes, and documentation.

During the warranty period, we will, at our option, either repair or replace any product that proves to be defective.

To exercise this warranty, write or call your local Keithley representative, or contact Keithley headquarters in Cleveland, Ohio. You will be given prompt assistance and return instructions. Send the product, transportation prepaid, to the indicated service facility. Repairs will be made and the product returned, transportation prepaid. Repaired or replaced products are warranted for the balance of the original warranty period, or at least 90 days.

LIMITATION OF WARRANTY

This warranty does not apply to defects resulting from product modification without Keithley's express written consent, or misuse of any product or part. This warranty also does not apply to fuses, software, non-rechargeable batteries, damage from battery leakage, or problems arising from normal wear or failure to follow instructions.

THIS WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR USE. THE REMEDIES PROVIDED HEREIN ARE BUYER'S SOLE AND EXCLUSIVE REMEDIES.

NEITHER KEITHLEY INSTRUMENTS, INC. NOR ANY OF ITS EMPLOYEES SHALL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF ITS INSTRUMENTS AND SOFTWARE EVEN IF KEITHLEY INSTRUMENTS, INC., HAS BEEN ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH DAMAGES. SUCH EXCLUDED DAMAGES SHALL INCLUDE, BUT ARE NOT LIMITED TO: COSTS OF REMOVAL AND INSTALLATION, LOSSES SUSTAINED AS THE RESULT OF INJURY TO ANY PERSON, OR DAMAGE TO PROPERTY.

KEITHLEY

A GREATER MEASURE OF CONFIDENCE

Keithley Instruments, Inc.

Corporate Headquarters • 28775 Aurora Road • Cleveland, Ohio 44139
440-248-0400 • Fax: 440-248-6168 • 1-888-KEITHLEY (534-8453) • www.keithley.com

DataAcq SDK User's Manual

©2005, Keithley Instruments, Inc.
All rights reserved.
First Printing, January 2005
Cleveland, Ohio, U.S.A.
Document Number: DASDK-900-01 Rev. A

Manual Print History

The print history shown below lists the printing dates of all Revisions and Addenda created for this manual. The Revision Level letter increases alphabetically as the manual undergoes subsequent updates. Addenda, which are released between Revisions, contain important change information that the user should incorporate immediately into the manual. Addenda are numbered sequentially. When a new Revision is created, all Addenda associated with the previous Revision of the manual are incorporated into the new Revision of the manual. Each new Revision includes a revised copy of this print history page.

Revision A (Document Number DASDK-900-01A)..... January 2005

The following safety precautions should be observed before using this product and any associated instrumentation. Although some instruments and accessories would normally be used with non-hazardous voltages, there are situations where hazardous conditions may be present.

This product is intended for use by qualified personnel who recognize shock hazards and are familiar with the safety precautions required to avoid possible injury. Read and follow all installation, operation, and maintenance information carefully before using the product. Refer to the manual for complete product specifications.

If the product is used in a manner not specified, the protection provided by the product may be impaired.

The types of product users are:

Responsible body is the individual or group responsible for the use and maintenance of equipment, for ensuring that the equipment is operated within its specifications and operating limits, and for ensuring that operators are adequately trained.

Operators use the product for its intended function. They must be trained in electrical safety procedures and proper use of the instrument. They must be protected from electric shock and contact with hazardous live circuits.

Maintenance personnel perform routine procedures on the product to keep it operating properly, for example, setting the line voltage or replacing consumable materials. Maintenance procedures are described in the manual. The procedures explicitly state if the operator may perform them. Otherwise, they should be performed only by service personnel.

Service personnel are trained to work on live circuits, and perform safe installations and repairs of products. Only properly trained service personnel may perform installation and service procedures.

Keithley products are designed for use with electrical signals that are rated Measurement Category I and Measurement Category II, as described in the International Electrotechnical Commission (IEC) Standard IEC 60664. Most measurement, control, and data I/O signals are Measurement Category I and must not be directly connected to mains voltage or to voltage sources with high transient over-voltages. Measurement Category II connections require protection for high transient over-voltages often associated with local AC mains connections. Assume all measurement, control, and data I/O connections are for connection to Category I sources unless otherwise marked or described in the Manual.

Exercise extreme caution when a shock hazard is present. Lethal voltage may be present on cable connector jacks or test fixtures. The American National Standards Institute (ANSI) states that a shock hazard exists when voltage levels greater than 30V RMS, 42.4V peak, or 60VDC are present. **A good safety practice is to expect that hazardous voltage is present in any unknown circuit before measuring.**

Operators of this product must be protected from electric shock at all times. The responsible body must ensure that operators are prevented access and/or insulated from every connection point. In some cases, connections must be exposed to potential human contact. Product operators in these circumstances must be trained to protect themselves from the risk of electric shock. If the circuit is capable of operating at or above 1000 volts, **no conductive part of the circuit may be exposed.**

Do not connect switching cards directly to unlimited power circuits. They are intended to be used with impedance limited sources. NEVER connect switching cards directly to AC mains. When connecting sources to switching cards, install protective devices to limit fault current and voltage to the card.

Before operating an instrument, make sure the line cord is connected to a properly grounded power receptacle. Inspect the connecting cables, test leads, and jumpers for possible wear, cracks, or breaks before each use.

When installing equipment where access to the main power cord is restricted, such as rack mounting, a separate main input power disconnect device must be provided, in close proximity to the equipment and within easy reach of the operator.

For maximum safety, do not touch the product, test cables, or any other instruments while power is applied to the circuit under test. ALWAYS remove power from the entire test system and discharge any capacitors before: connecting or disconnecting cables or jumpers, installing or removing switching cards, or making internal changes, such as installing or removing jumpers.

Do not touch any object that could provide a current path to the common side of the circuit under test or power line (earth) ground. Always make measurements with dry hands while standing on a dry, insulated surface capable of withstanding the voltage being measured.


The instrument and accessories must be used in accordance with its specifications and operating instructions or the safety of the equipment may be impaired.


Do not exceed the maximum signal levels of the instruments and accessories, as defined in the specifications and operating information, and as shown on the instrument or test fixture panels, or switching card.


When fuses are used in a product, replace with same type and rating for continued protection against fire hazard.


Chassis connections must only be used as shield connections for measuring circuits, NOT as safety earth ground connections.

If you are using a test fixture, keep the lid closed while power is applied to the device under test. Safe operation requires the use of a lid interlock.

If a  screw is present, connect it to safety earth ground using the wire recommended in the user documentation.

The  symbol on an instrument indicates that the user should refer to the operating instructions located in the manual.

The  symbol on an instrument shows that it can source or measure 1000 volts or more, including the combined effect of normal and common mode voltages. Use standard safety precautions to avoid personal contact with these voltages.

The  symbol indicates a connection terminal to the equipment frame.

The **WARNING** heading in a manual explains dangers that might result in personal injury or death. Always read the associated information very carefully before performing the indicated procedure.

The **CAUTION** heading in a manual explains hazards that could damage the instrument. Such damage may invalidate the warranty.

Instrumentation and accessories shall not be connected to humans. Before performing any maintenance, disconnect the line cord and all test cables.

To maintain protection from electric shock and fire, replacement components in mains circuits, including the power transformer, test leads, and input jacks, must be purchased from Keithley Instruments. Standard fuses, with applicable national safety approvals, may be used if the rating and type are the same. Other components that are not safety related may be purchased from other suppliers as long as they are equivalent to the original component. (Note that selected parts should be purchased only through Keithley Instruments to maintain accuracy and functionality of the product.) If you are unsure about the applicability of a replacement component, call a Keithley Instruments office for information.

To clean an instrument, use a damp cloth or mild, water based cleaner. Clean the exterior of the instrument only. Do not apply cleaner directly to the instrument or allow liquids to enter or spill on the instrument. Products that consist of a circuit board with no case or chassis (e.g., data acquisition board for installation into a computer) should never require cleaning if handled according to instructions. If the board becomes contaminated and operation is affected, the board should be returned to the factory for proper cleaning/servicing.

Table of Contents

About this Manual	xiii
Intended Audience.....	xiii
What You Should Learn from this Manual.....	xiii
Organization of this Manual.....	xiv
Conventions Used in this Manual	xiv
Related Information.....	xv
Where to Get Help	xv
Chapter 1: Overview	1
What is the DataAcq SDK?.....	2
Installation.....	3
Using the DataAcq SDK Online Help	4
About the Examples Programs.....	5
About the Library Function Calling Conventions.....	7
Chapter 2: Function Summary	9
Data Acquisition Functions.....	10
Information Functions	10
Initialization and Termination Functions	21
Configuration Functions	22
Operation Functions	27
Data Conversion Functions	30
Data Management Functions	31
Buffer Management Functions	31
Buffer List Management Functions	33

Chapter 3: Using the DataAcq SDK	35
System Operations	37
Initializing a Device	37
Specifying a Subsystem	38
Configuring a Subsystem.....	39
Handling Errors.....	40
Handling Messages.....	40
Releasing the Subsystem and the Driver	40
Analog and Digital I/O Operations	41
Data Encoding	41
Resolution.....	42
Channels.....	42
Specifying the Channel Type	43
Specifying a Single Channel	44
Specifying One or More Channels	44
Specifying the Channel List Size	45
Specifying the Channels in the Channel List	46
Inhibiting Channels in the Channel List	47
Specifying Synchronous Digital I/O Values in the Channel List	48
Ranges.....	50
Gains	51
Specifying the Gain for a Single Channel	51
Specifying the Gain for One or More Channels	51
Filters.....	53
Data Flow Modes	54
Single-Value Operations	54
Continuous Operations	55
Continuous Post-Trigger Mode	57
Continuous Pre-Trigger Mode	58

Continuous About-Trigger Mode	59
Triggered Scan Mode	61
Scan-Per-Trigger Mode	62
Internal Retrigger Mode	63
Retrigger Extra Mode	64
Clock Sources	65
Internal Clock Source	65
External Clock Source	66
Extra Clock Source	67
Trigger Sources	67
Software (Internal) Trigger Source	68
External Digital (TTL) Trigger Source	68
External Analog Threshold (Positive) Trigger Source	69
External Analog Threshold (Negative) Trigger Source	69
Analog Event Trigger Source	70
Digital Event Trigger Source	70
Timer Event Trigger Source	70
Extra Trigger Source	70
Buffers	71
Ready Queue	71
Inprocess Queue	72
Done Queue	74
Buffer and Queue Management	76
Buffer Wrap Modes	78
DMA and Interrupt Resources	79
Counter/Timer Operations	81
Counter/Timer Operation Mode	82
Event Counting	82
Up/Down Counting	84

Frequency Measurement	86
Using the Windows Timer	86
Using a Pulse of a Known Duration	87
Edge-to-Edge Measurement	90
Rate Generation	93
One-Shot	96
Repetitive One-Shot	99
C/T Clock Sources	101
Internal C/T Clock	102
External C/T Clock	102
Internally Cascaded Clock	103
Extra C/T Clock Source	104
Gate Types	104
Software Gate Type	105
High-Level Gate Type	105
Low-Level Gate Type	105
Low-Edge Gate Type	106
High-Edge Gate Type	106
Any Level Gate Type	106
High-Level, Debounced Gate Type	107
Low-Level, Debounced Gate Type	107
High-Edge, Debounced Gate Type	107
Low-Edge, Debounced Gate Type	108
Level, Debounced Gate Type	108
Pulse Output Types and Duty Cycles	108
Simultaneous Operations	110
Chapter 4: Programming Flowcharts.	113
Single-Value Operations	115
Continuous Buffered Input Operations	117

Continuous Buffered Output Operations	119
Event Counting Operations	121
Up/Down Counting Operations	123
Frequency Measurement Operations	125
Edge-to-Edge Measurement Operations.	127
Pulse Output Operations.	129
Simultaneous Operations	131
Chapter 5: Product Support	149
Appendix A: Sample Code	153
Single-Value Analog Input	154
Declare Variables and User Functions	154
Initialize the Driver	155
Get a Handle to the Subsystem.	156
Set the DataFlow to Single Value	156
Configure the Subsystem	156
Acquire a Single Value	157
Convert the Value to Voltage	157
Release the Subsystem and Terminate the Session.	158
Handle Errors	158
Continuous Analog Input	159
Declare Variables and User Functions	159
Initialize the Driver	161
Get a Handle to the Subsystem.	162
Set the DataFlow to Continuous.	162
Specify the Channel List and Channel Parameters	162
Specify the Clocks	163
Specify DMA Usage	164
Set Up Window Handle and Buffering	164
Configure the Subsystem	165

Start the Continuous Analog Input Operation	165
Deal with Messages and Buffers	165
Convert Values to Voltage	168
Clean Up	169
Handle Errors	169
Index	171

About this Manual

This manual describes how to get started using the DataAcq SDK™ (Software Development Kit) to develop application programs for data acquisition devices that conform to the DT-Open Layers™ standard.

Intended Audience

This document is intended for engineers, scientists, technicians, OEMs, system integrators, or others responsible for developing application programs using Microsoft® Developer's Studio (version 6.0 and higher) to perform data acquisition operations.

It is assumed that you are a proficient programmer, that you are experienced programming in the Windows® 2000 or Windows XP operating environment on the IBM PC or compatible computer platform, and that you have familiarity with data acquisition principles and the requirements of your application.

What You Should Learn from this Manual

This manual provides installation instructions for Windows 2000 and Windows XP, summarizes the functions provided by the DataAcq SDK, and describes how to use the functions to develop a data acquisition program. Using this manual, you should be able to successfully install the DataAcq SDK and get started writing an application program for data acquisition.

This manual is intended to be used with the online help for the DataAcq SDK, which you can find in the same program group as the DataAcq SDK software. The online help for the DataAcq SDK contains all of the specific reference information for each of the functions, error codes, and Windows messages.

Organization of this Manual

This manual is organized as follows:

- [Chapter 1, “Overview,”](#) tells how to install the DataAcq SDK in Windows 2000 and Windows XP.
- [Chapter 2, “Function Summary,”](#) summarizes the functions provided in the DataAcq SDK.
- [Chapter 3, “Using the DataAcq SDK,”](#) describes the operations that you can perform using the DataAcq SDK.
- [Chapter 4, “Programming Flowcharts,”](#) provides programming flowcharts for using the functions provided in the DataAcq SDK.
- [Chapter 5, “Product Support,”](#) describes how to get help if you have trouble using the DataAcq SDK.
- [Appendix A, “Sample Code,”](#) provides code fragments that illustrate the use of the functions in the DataAcq SDK.
- An index completes this manual.

Conventions Used in this Manual

The following conventions are used in this manual:

- Notes provide useful information that requires special emphasis, cautions provide information to help you avoid losing data or damaging your equipment, and warnings provide information to help you avoid catastrophic damage to yourself or your equipment.
- Items that you select or type are shown in **bold**. Function names also appear in bold.
- Code fragments are shown in `courier font`.

Related Information

Refer to the following documentation for more information on using the DataAcq SDK:

- DataAcq SDK Online Help. This Windows help file is located in the same program group as the DataAcq SDK software and contains all of the specific reference information for each of the functions, error codes, and Windows messages provided by the DataAcq SDK. Refer to [page 4](#) for information on how to open this help file.
- Device-specific documentation, which consists of a getting started manual and a user's manual. The getting started manual describes how to install the data acquisition device, how to install the device driver for the device, and how to get started using the device. The user's manual describes the features of the device and the capabilities supported by the device driver for the device. These manuals are on the Keithley CD™.
- Windows 2000 or Windows XP documentation.
- For C programmers, refer to *Microsoft C Reference*, Document Number LN06515-1189, Microsoft Corporation, and *The C Programming Language*, Brian W. Kernighan and Dennis Ritchie, Prentice Hall, 1988, 1987 Bell Telephone Laboratories, Inc, ISBN 0-13-109950-7.

Where to Get Help

Should you run into problems installing or using the DataAcq SDK, the Keithley Technical Support Department is available to provide technical assistance.



Overview

What is the DataAcq SDK?	2
Installation	3
Using the DataAcq SDK Online Help	4
About the Examples Programs	5
About the Library Function Calling Conventions	7

What is the DataAcq SDK?

The DataAcq SDK is a programmer's DLL (Dynamic Linked Library) that supports Keithley's data acquisition devices under Microsoft Windows 2000 and Windows XP.

The DataAcq SDK functions are fully compatible with DT-Open Layers™, which is a set of standards for developing integrated, modular application programs under Windows.

Because DT-Open Layers is modular and uses Windows DLLs, you can add support for a new data acquisition device at any time. Just add the new DT-Open Layers device driver, modify your code to incorporate the features of the new device, and then recompile the code. All calls to DataAcq SDK functions currently in your application program can remain untouched.

Installation

1

The DataAcq SDK is installed automatically when you install the device driver for the module. Refer to your getting started manual for more information.

Using the DataAcq SDK Online Help

The *DataAcq SDK User's Manual* is intended to be used with the online help for the DataAcq SDK. The online help contains all of the specific reference information for each of the functions, error codes, and Windows messages not included in this manual.

To open the online help file, select **DataAcq SDK/DataAcq SDK Help** from the Windows Start menu.

About the Examples Programs

To help you understand more about using the functions of the DataAcq SDK in an actual program, the DataAcq SDK provides a C example program (CEXMPL32.EXE). This example program allows you to configure any of the subsystems on the data acquisition device. The source code is located in the \Examples\CExample directory. Resource files are also provided.

In addition to CEXMPL.EXE, the following simple example programs are also provided. These programs are designed to use minimum Windows user interface code, while demonstrating the functionality of the DataAcq SDK. Source code and resource files are provided for each of these programs:

- **ContAdc** –Opens the first available DT-Open Layers device, opens and configures an A/D subsystem, and performs continuous operations. The results are displayed in a dialog box.
- **ContDac** –Opens the first available DT-Open Layers device, opens and configures a D/A subsystem, and performs continuous operations outputting a square wave. The results are displayed in a dialog box.
- **DtConsole** –Opens the first available DT-Open Layers device, opens and configures an A/D subsystem, and performs a continuous A/D operation on a console screen.
- **GenerateFreq** –Opens the first available DT-Open Layers device, opens and configures a C/T subsystem, and continuously outputs a pulse. The results are displayed in a dialog box.
- **MeasureFreq** –Opens the first available DT-Open Layers device, opens and configures a C/T subsystem, and continuously measures a pulse. The results are displayed in a dialog box.
- **SvAdc** –Opens the first available DT-Open Layers device, opens and configures an A/D subsystem, and performs a single-value operation. The results are displayed in a message box.

- **SvDac** –Opens the first available DT-Open Layers device, opens and configures a D/A subsystem, and performs a single-value operation with maximum positive and maximum negative range. The results are displayed in a message box.
- **SvDin** –Opens the first available DT-Open Layers device, opens and configures a DIN subsystem, and performs a single-value operation. The results are displayed in a message box.
- **SvDout** –Opens the first available DT-Open Layers device, opens and configures a DOUT subsystem, and performs a single-value operation. The results are displayed in a message box.
- **ThermoAdc** –Opens the first available DT-Open Layers device, opens and configures an A/D subsystem, and performs a thermocouple measurement. The results are displayed in a dialog box.

Each example program provided in the DataAcq SDK comes with the MSVC.DSW (workspace) and MSVC.DSP (project) files for use in the integrated development environment provided by the Microsoft Developer's Studio. No special switches are necessary beyond instructing the IDE to create a Windows EXE or DLL.

Note: The DataAcq SDK installation program automatically includes an environment variable (DA_SDK). All the example programs use this environment variable; therefore, you can build the example programs without adding any include or library files to your projects.

About the Library Function Calling Conventions

The DataAcq SDK functions adhere to the Microsoft Pascal calling conventions. You can find prototypes for these functions in the include files OLDAAPI.H and OLMEM.H. It is recommended that you follow these calling conventions for proper operation.

DataAcq SDK functions return an ECODE value, which is an unsigned long value indicating the status of the requested function. It is recommended that you check the return status value for an error condition using the symbolic constants defined in the include files. This practice is illustrated in the C example program (CEXMPL32.EXE).

Note: For detailed information on the error codes, refer to the DataAcq SDK online help.



Function Summary

Data Acquisition Functions.....	10
Data Management Functions	31

Data Acquisition Functions

The following groups of data acquisition functions are available:

- Information functions,
- Initialization and Termination functions,
- Configuration functions,
- Operation functions, and
- Data Conversion functions.

These functions are briefly described in the following subsections.

Note: For specific information about each of these functions, refer to the DataAcq SDK online help. See [page 4](#) for information on opening the online help file.

Information Functions

To determine the capabilities of your installed devices, subsystems on each device, and software, use the Information functions listed in [Table 1](#).

Table 1: Information Functions

Query about	Function	Description
Devices	oIDaEnumBoards	Lists all currently installed DT-Open Layers data acquisition devices, drivers, and driver parameters.
	oIGetBoardInfo	Gets the driver name, model name, and instance number of the specified board, based on its board name.

Table 1: Information Functions (cont.)

Query about	Function	Description
Devices (cont.)	oIdaGetDeviceName	Gets the full name of the specified device (this name is set by the driver as part of the installation procedure).
Subsystems	oIdaEnumSubSystems	Lists the names, types, and element number for each subsystem supported by the specified device.
	oIdaGetDevCaps	Returns the number of elements available for the specified subsystem on the specified device.
	oIdaGetSSCaps	Returns information about whether the specified subsystem capability is supported and/or the number of capabilities supported. Refer to Table 2 for a list of possible capabilities and return values.
	oIdaGetSSCapsEx	Returns information about extended subsystem capabilities. Refer to Table 3 for a list of possible capabilities and return values.
	oIdaEnumBoardsEx	Lists all currently-installed DT-Open Layers DataAcq drivers and returns some registry information for each.
	oIdaEnumSSCaps	Lists the possible settings for the specified subsystem capabilities, including filters, ranges, gains, and resolution.
	oIdaGetDASSInfo	Returns the subsystem type and element number of the specified subsystem with the specified device handle.

Table 1: Information Functions (cont.)

Query about	Function	Description
Subsystems (cont.)	oIDaGetQueueSize	Returns the size of the specified queue (ready, done or inprocess) for the specified subsystem. The size indicates the number of buffers on the specified queue.
	oIDaEnumSSList	Lists all subsystems on the simultaneous start list.
Software	oIDaGetDriverVersion	Returns the device driver version number.
	oIDaGetVersion	Returns the software version of the DataAcq SDK.
	oIDaGetErrorString	Returns the string that corresponds to a device error code value.

[Table 2](#) lists the subsystem capabilities that you can query using the **oIDaGetSSCaps** function; this function returns values as integers. [Table 3](#) lists the subsystem capabilities that you can query using the **oIDaGetSSCapsEx** function; this function returns values as floating-point numbers. Note that capabilities may be added as new devices are developed; for the most recent set of capabilities, refer to the DataAcq SDK online help.

Table 2: Capabilities to Query with olDaGetSSCaps

Query about	Capability	Function Returns
Data Flow Mode	OLSSC_SUP_SINGLEVALUE	Nonzero if subsystem supports single-value operations.
	OLSSC_SUP_CONTINUOUS	Nonzero if subsystem supports continuous post-trigger operations.
	OLSSC_SUP_CONTINUOUS_PRETRIG	Nonzero if subsystem supports continuous pre-trigger operations.
	OLSSC_SUP_CONTINUOUS_ABOUTTRIG	Nonzero if subsystem supports continuous about-trigger (both pre- and post-trigger) operations.
Simultaneous Operations	OLSSC_SUP_SIMULTANEOUS_START	Nonzero if subsystem can be started simultaneously with another subsystem on the device.
Pausing Operations	OLSSC_SUP_PAUSE	Nonzero if subsystem supports pausing during continuous operation.
Windows Messaging	OLSSC_SUP_POSTMESSAGE	Nonzero if subsystem supports asynchronous operations.
Buffering	OLSSC_SUP_BUFFERING	Nonzero if subsystem supports buffering.
	OLSSC_SUP_WRPSINGLE	Nonzero if subsystem supports single-buffer wrap mode.
	OLSSC_SUP_WRPMULTIPLE	Nonzero if subsystem supports multiple-buffer wrap mode.
	OLSSC_SUP_INPROCESSFLUSH	Nonzero if subsystem supports the transferring of data from a buffer on a subsystem's inprocess queue.

Table 2: Capabilities to Query with oIDaGetSSCaps (cont.)

Query about	Capability	Function Returns
DMA	OLSSC_NUMDMACHANS	Number of DMA channels supported.
	OLSSC_SUP_GAPFREE_NODMA	Nonzero if subsystem supports gap-free continuous operation with no DMA.
	OLSSC_SUP_GAPFREE_SINGLEDMA	Nonzero if subsystem supports gap-free continuous operation with a single DMA channel.
	OLSSC_SUP_GAPFREE_DUALDMA	Nonzero if subsystem supports gap-free continuous operation with two DMA channels.
Triggered Scan Mode	OLSSC_SUP_TRIGSCAN	Nonzero if subsystem supports triggered scans.
	OLSSC_MAXMULTISCAN	Maximum number of scans per trigger or retrigger supported by the subsystem.
	OLSSC_SUP_RETRIGGER_SCAN_PER_TRIGGER	Nonzero if subsystem supports scan-per-trigger triggered scan mode (retrigger is the same as the initial trigger source).
	OLSSC_SUP_RETRIGGER_INTERNAL	Nonzero if subsystem supports internal retriggered scan mode. (retrigger source is on the device; initial trigger is any available trigger source).
	OLSSC_SUP_RETRIGGER_EXTRA	Nonzero if subsystem supports retrigger-extra triggered scan mode (retrigger can be any supported trigger source; initial trigger is any available trigger source).

Table 2: Capabilities to Query with oIDaGetSSCaps (cont.)

Query about	Capability	Function Returns
Channel-Gain List	OLSSC_CGLDEPTH	Number of entries in channel-gain list.
	OLSSC_SUP_RANDOM_CGL	Nonzero if subsystem supports random channel-gain list setup.
	OLSSC_SUP_SEQUENTIAL_CGL	Nonzero if subsystem supports sequential channel-gain list setup.
	OLSSC_SUP_ZEROSEQUENTIAL_CGL	Nonzero if subsystem supports sequential channel-gain list setup starting with channel zero.
Channel-Gain List (cont.)	OLSSC_SUP_SIMULTANEOUS_SH	Nonzero if subsystem supports simultaneous sample-and-hold operations. The channel-gain list must be set up with both a sample channel and a hold channel.
	OLSSC_SUP_CHANNELLIST_INHIBIT	Nonzero if subsystem supports channel-gain list entry inhibition.
Gain	OLSSC_SUP_PROGRAMGAIN	Nonzero if subsystem supports programmable gain.
	OLSSC_NUMGAINS	Number of gain selections.
	OLSSC_SUP_SINGLEVALUE_AUTORANGE	Nonzero if subsystem supports autoranging operations.
Synchronous Digital I/O	OLSSC_SUP_SYNCHRONOUS_DIGITALIO	Nonzero if subsystem supports synchronous digital output operations.
	OLSSC_MAXDIGITALIOLIST_VALUE	Maximum value for synchronous digital output channel list entry.
I/O Channels	OLSSC_NUMCHANNELS	Number of I/O channels.

Table 2: Capabilities to Query with oIDaGetSSCaps (cont.)

Query about	Capability	Function Returns
Channel Type	OLSSC_SUP_SINGLEENDED	Nonzero if subsystem supports single-ended inputs.
	OLSSC_MAXSECHANS	Number of single-ended channels.
	OLSSC_SUP_DIFFERENTIAL	Nonzero if subsystem supports differential inputs.
	OLSSC_MAXDICHANS	Number of differential channels.
Filters	OLSSC_SUP_FILTERPERCHAN	Nonzero if subsystem supports filtering per channel.
	OLSSC_NUMFILTERS	Number of filter selections.
Ranges	OLSSC_NUMRANGES	Number of range selections.
	OLSSC_SUP_RANGEPERCHANNEL	Nonzero if subsystem supports different range settings for each channel.
Resolution	OLSSC_SUP_SWRESOLUTION	Nonzero if subsystem supports software-programmable resolution.
	OLSSC_NUMRESOLUTIONS	Number of different resolutions that you can program for the subsystem.
Data Encoding	OLSSC_SUP_BINARY	Nonzero if subsystem supports binary encoding.
	OLSSC_SUP_2SCOMP	Nonzero if subsystem supports twos complement encoding.

Table 2: Capabilities to Query with oIDaGetSSCaps (cont.)

Query about	Capability	Function Returns
Triggers	OLSSC_SUP_SOFTTRIG	Nonzero if subsystem supports internal software trigger.
	OLSSC_SUP_EXTERNTRIG	Nonzero if subsystem supports external digital (TTL) trigger.
	OLSSC_SUP_THRESHTRIGPOS	Nonzero if subsystem supports positive analog threshold trigger.
	OLSSC_SUP_THRESHTRIGNEG	Nonzero if subsystem supports negative analog threshold trigger.
	OLSSC_SUP_ANALOGEVENTTRIG	Nonzero if subsystem supports analog event trigger.
	OLSSC_SUP_DIGITALEVENTTRIG	Nonzero if subsystem supports digital event trigger.
	OLSSC_SUP_TIMEREVENTTRIG	Nonzero if subsystem supports timer event trigger.
	OLSSC_NUMEXTRATRIGGERS	Number of extra trigger sources supported.
Clocks	OLSSC_SUP_INTCLOCK	Nonzero if subsystem supports internal clock.
	OLSSC_SUP_EXTCLOCK	Nonzero if subsystem supports external clock.
	OLSSC_NUMEXTRACLOCKS	Number of extra clock sources.
	OLSSC_SUP_SIMULTANEOUS_CLOCKING	Non-zero if subsystem supports simultaneous clocking of all channels.

Table 2: Capabilities to Query with oIDaGetSSCaps (cont.)

Query about	Capability	Function Returns
Counter/Timer Modes	OLSSC_SUP_CASCADING	Nonzero if subsystem supports cascading.
	OLSSC_SUP_CTMODE_COUNT	Nonzero if subsystem supports event counting mode.
	OLSSC_SUP_CTMODE_RATE	Nonzero if subsystem supports rate generation (continuous pulse output) mode.
	OLSSC_SUP_CTMODE_ONESHOT	Nonzero if subsystem supports (single) one-shot mode.
	OLSSC_SUP_CTMODE_ONESHOT_RPT	Nonzero if subsystem supports repetitive one-shot mode.
	OLSSC_SUP_CTMODE_UP_DOWN	Nonzero if subsystem supports up/down counting mode.
	OLSSC_SUP_CTMODE_MEASURE	Returns a value indicating how edge-to-edge measurement mode is supported (see page 90 for more information).
	OLSSC_SUP_CTMODE_CONT_MEASURE	Returns a value indicating how edge-to-edge measurement mode is supported (see page 90 for more information).
Counter/Timer Pulse Output Types	OLSSC_SUP_PLS_HIGH2LOW	Nonzero if subsystem supports high-to-low output pulses.
	OLSSC_SUP_PLS_LOW2HIGH	Nonzero if subsystem supports low-to-high output pulses

Table 2: Capabilities to Query with oIDaGetSSCaps (cont.)

Query about	Capability	Function Returns
Counter/Timer Gates	OLSSC_SUP_GATE_NONE	Nonzero if subsystem supports an internal (software) gate type.
	OLSSC_SUP_GATE_HIGH_LEVEL	Nonzero if subsystem supports high-level gate type.
	OLSSC_SUP_GATE_LOW_LEVEL	Nonzero if subsystem supports low-level gate type.
	OLSSC_SUP_GATE_HIGH_EDGE	Nonzero if subsystem supports high-edge gate type.
	OLSSC_SUP_GATE_LOW_EDGE	Nonzero if subsystem supports low-edge gate type.
	OLSSC_SUP_GATE_LEVEL	Nonzero if subsystem supports level change gate type.
	OLSSC_SUP_GATE_HIGH_LEVEL_DEBOUNCE	Nonzero if subsystem supports high-level gate type with input debounce.
	OLSSC_SUP_GATE_LOW_LEVEL_DEBOUNCE	Nonzero if subsystem supports low-level gate type with input debounce.
	OLSSC_SUP_GATE_HIGH_EDGE_DEBOUNCE	Nonzero if subsystem supports high-edge gate type with input debounce.
	OLSSC_SUP_GATE_LOW_EDGE_DEBOUNCE	Nonzero if subsystem supports low-edge gate type with input debounce.
OLSSC_SUP_GATE_LEVEL_DEBOUNCE	Nonzero if subsystem supports level change gate type with input debounce.	
Interrupt	OLSSC_SUP_INTERRUPT	Nonzero if subsystem supports interrupt-driven I/O.

Table 2: Capabilities to Query with oIDaGetSSCaps (cont.)

Query about	Capability	Function Returns
FIFOs	OLSSC_SUP_FIFO	Nonzero if subsystem has a FIFO in the data path.
	OLSSC_FIFO_SIZE_IN_K	Size of the output FIFO, in kilobytes.
Processors	OLSSC_SUP_PROCESSOR	Nonzero if subsystem has a processor on device.
Software Calibration	OLSSC_SUP_SWCAL	Nonzero if subsystem supports software calibration.

Table 3: Capabilities to Query with oIDaGetSSCapsEx

Query about	Capability	Function Returns
Triggered Scan Mode	OLSSCE_MAXRETRIGGER	Maximum retrigger frequency supported by the subsystem.
	OLSSCE_MINRETRIGGER	Minimum retrigger frequency supported by the subsystem.
Clocks	OLSSCE_BASECLOCK	Base clock frequency supported by the subsystem.
	OLSSCE_MAXCLOCKDIVIDER	Maximum external clock divider supported by the subsystem.
	OLSSCE_MINCLOCKDIVIDER	Minimum external clock divider supported by the subsystem.
	OLSSCE_MAXTHROUGHPUT	Maximum throughput supported by the subsystem.
	OLSSCE_MINTHROUGHPUT	Minimum throughput supported by the subsystem.

Initialization and Termination Functions

Once you have identified the available devices, use the Initialization functions described in [Table 4](#).

Table 4: Initialization Functions

Function	Description
oIDaInitialize	Provides the means for the software to associate specific requests with a particular device; it must be called before any other function. This function loads a specified device's software support and provides a "device handle" value. This value is used to identify the device, and must be supplied as an argument in all subsequent function calls that reference the device.
oIDaGetDASS	Allocates a subsystem for use by returning a handle to the subsystem.

When you are finished with your program, use the Termination functions listed in [Table 5](#).

Table 5: Termination Functions

Function	Description
oIDaReleaseDASS	Releases the specified subsystem and relinquishes all resources associated with it.
oIDaTerminate	Ends a session between your application and the specified device. The device is returned to an inactive state and all resources are returned to the system.

Configuration Functions

Once you have initialized a subsystem and determined what its capabilities are, set or get the value of the subsystem's parameters by calling the Configuration functions listed in [Table 6](#).

Table 6: Configuration Functions

Feature	Function	Description
Data Flow Mode	oiDaSetDataFlow	Sets the data flow mode.
	oiDaGetDataFlow	Gets the data flow mode.
Windows Messaging	oiDaSetNotificationProcedure	Specifies the notification procedure to call for information messages from the subsystem.
	oiDaGetNotificationProcedure	Gets the address of the notification procedure.
	oiDaSetWndHandle	Sets the window to which information messages are sent.
	oiDaGetWndHandle	Gets the window handle.
Buffer Wrap Mode	oiDaSetWrapMode	Sets the buffer processing wrap mode.
	oiDaGetWrapMode	Gets the buffer processing wrap mode.
DMA	oiDaSetDmaUsage	Sets the number of DMA channels to be used.
	oiDaGetDmaUsage	Gets the number of DMA channels to be used.

Table 6: Configuration Functions (cont.)

Feature	Function	Description
Triggered Scans	oiDaSetTriggeredScanUsage	Enables or disables triggered scan mode.
	oiDaGetTriggeredScanUsage	Gets the triggered scan mode setting.
	oiDaSetMultiscanCount	Sets the number of times to scan per trigger/retrigger.
	oiDaGetMultiscanCount	Gets the number of times to scan per trigger/retrigger.
	oiDaSetRetriggerMode	Sets the retrigger mode.
	oiDaGetRetriggerMode	Gets the retrigger mode.
	oiDaSetRetriggerFrequency	Sets the frequency of the internal retrigger when using internal retrigger mode.
	oiDaGetRetriggerFrequency	Gets the frequency of the internal retrigger when using internal retrigger mode.
Channel-Gain List	oiDaSetChannelListSize	Sets the size of the channel-gain list.
	oiDaGetChannelListSize	Gets the size of the channel-gain list.
	oiDaSetChannelListEntry	Sets the channel number of a channel-gain list entry.
	oiDaGetChannelListEntry	Gets the channel number of a channel-gain list entry.
	oiDaSetGainListEntry	Sets a gain value for a channel-gain list entry.
	oiDaGetGainListEntry	Gets the gain value of a channel-gain list entry.

Table 6: Configuration Functions (cont.)

Feature	Function	Description
Channel-Gain List (cont.)	oiDaSetChannelListEntryInhibit	Enables/disables channel entry inhibition for a channel-gain list entry.
	oiDaGetChannelListEntryInhibit	Gets the channel entry inhibition setting of a channel-gain list entry.
	oiDaSetDigitalIOListEntry	Sets the digital value to output for the channel-gain list entry.
	oiDaGetDigitalIOListEntry	Gets the digital value to output for the channel-gain list entry.
Synchronous Digital I/O	oiDaSetSynchronousDigitalIOUsage	Enables or disables synchronous digital I/O operations.
	oiDaGetSynchronousDigitalIOUsage	Gets the synchronous digital I/O setting.
Channel Type	oiDaSetChannelType	Sets the channel configuration type of a channel.
	oiDaGetChannelType	Gets the channel configuration type of a channel.
Filters	oiDaSetChannelFilter	Sets the filter cut-off frequency for a channel.
	oiDaGetChannelFilter	Gets the filter cut-off frequency for a channel.

Table 6: Configuration Functions (cont.)

Feature	Function	Description
Ranges	oiDaSetRange	Sets the voltage range for a subsystem.
	oiDaGetRange	Gets the voltage range for a subsystem.
	oiDaSetChannelRange	Sets the voltage range for a channel.
	oiDaGetChannelRange	Gets the voltage range for a channel.
Resolution	oiDaSetResolution	Sets the number of bits of resolution.
	oiDaGetResolution	Gets the number of bits of resolution.
Data Encoding	oiDaSetEncoding	Sets the data encoding type.
	oiDaGetEncoding	Gets the data encoding type.
Triggers	oiDaSetTrigger	Sets the post-trigger source.
	oiDaGetTrigger	Gets the post-trigger source.
	oiDaSetPretriggerSource	Sets the pre-trigger source.
	oiDaGetPretriggerSource	Gets the pre-trigger source.
	oiDaSetRetrigger	Sets the retrigger source for retrigger-extra retrigger mode.
	oiDaGetRetrigger	Gets the retrigger source for retrigger-extra retrigger mode.

Table 6: Configuration Functions (cont.)

Feature	Function	Description
Clocks	oiDaSetClockSource	Sets the clock source.
	oiDaGetClockSource	Gets the clock source.
	oiDaSetClockFrequency	Sets the frequency of the internal clock or a counter/timer's output frequency.
	oiDaGetClockFrequency	Gets the frequency of the internal clock or a counter/timer's output frequency.
	oiDaSetExternalClockDivider	Sets the input divider value of the external clock.
	oiDaGetExternalClockDivider	Gets the input divider value of the external clock.
Counter/ Timers	oiDaSetCTMode	Sets the counter/timer mode.
	oiDaGetCTMode	Gets the counter/timer mode.
	oiDaSetCascadeMode	Sets the counter/timer cascade mode.
	oiDaGetCascadeMode	Gets the counter/timer cascade mode.
	oiDaSetGateType	Sets the gate type for the counter/timer mode.
	oiDaGetGateType	Gets the gate type for the counter/timer mode.
	oiDaSetPulseType	Sets the pulse type for the counter/timer mode.
	oiDaGetPulseType	Gets the pulse type for the counter/timer mode.

Table 6: Configuration Functions (cont.)

Feature	Function	Description
Counter/ Timers (cont.)	oiDaSetPulseWidth	Sets the pulse output width for the counter/timer mode.
	oiDaGetPulseWidth	Gets the pulse width for the counter/timer mode.
	oiDaGetMeasureStartEdge	Gets the start edge for edge-to-edge measurement operations.
	oiDaSetMeasureStartEdge	Sets the start edge for edge-to-edge measurement operations.
	oiDaGetMeasureStopEdge	Gets the stop edge for edge-to-edge measurement operations.
	oiDaSetMeasureStopEdge	Sets the stop edge for edge-to-edge measurement operations.

Operation Functions

Once you have set the parameters of a subsystem, use the Operation functions listed in [Table 7](#).

Table 7: Operation Functions

Operation	Function	Description
Single-Value Operations	oIDaGetSingleValue	Reads a single input value from the specified subsystem channel.
	oIDaGetSingleValueEx	Determines the appropriate gain for the range (called autoranging), if desired, reads a single input value from the specified subsystem channel, and returns the value as both a code and a voltage.
	oIDaPutSingleValue	Writes a single output value to the specified subsystem channel.
Configure Operation	oIDaConfig	After setting up a specified subsystem using the configuration functions, configures the subsystem with new parameter values.
Start/Stop Operations	oIDaStart	Starts the operation for which the subsystem has been configured.
	oIDaPause	Pauses a continuous operation on the subsystem.
	oIDaContinue	Continues the previously paused operation on the subsystem.
	oIDaStop	Stops the operation and returns the subsystem to the ready state.
	oIDaAbort	Stops the subsystem's operation immediately.
	oIDaReset	Causes the operation to terminate immediately, and re-initializes the subsystem.

Table 7: Operation Functions (cont.)

Operation	Function	Description
Buffer Operations	oIDaGetBuffer	Gets a completed buffer from the done queue of the specified subsystem.
	oIDaPutBuffer	Assigns a data buffer for the subsystem to the ready queue.
	oIDaFlushBuffers	Transfers all data buffers held by the subsystem to the done queue.
	oIDaFlushFromBufferInprocess	Copies all valid samples, up to a given number of samples, from the inprocess buffer to a specified buffer. It also sets the logical size of the buffer with flushed data to the number of samples copied and places the inprocess buffer on the done queue when it has been filled with the remaining samples.
Counter/ Timer Operations	oIDaReadEvents	Gets the number of events that have been counted since the subsystem was started with oIDaStart .
	oIDaMeasureFrequency	Measures the frequency of the input clock source for the selected counter/timer.

Table 7: Operation Functions (cont.)

Operation	Function	Description
Simultaneous Operations	oIDaGetSSLList	Gets a handle to a simultaneous start list.
	oIDaPutDassToSSLList	Puts the specified subsystem on the simultaneous start list.
	oIDaSimultaneousPreStart	Simultaneously prestarts (performs setup operations on) all subsystems on the specified simultaneous start list.
	oIDaSimultaneousStart	Simultaneously starts all subsystems on the specified simultaneous start list.
	oIDaReleaseSSLList	Releases the specified simultaneous start list and relinquishes all resources associated with it.

Data Conversion Functions

Once you have acquired data, you can use the functions listed in [Table 8](#) to convert the data, if desired.

Table 8: Data Conversion Functions

Function	Description
oIDaCodeToVolts	Converts a code value to voltage value, using the range, gain, resolution, and encoding you specify.
oIDaVoltsToCode	Converts a voltage value to code value, using the range, gain, resolution, and encoding you specify.

Data Management Functions

Data management functions the various layers of the DT-Open Layers architecture together. The fundamental data object in the DataAcq SDK is a buffer. All functions that create, manipulate, and delete buffers are encapsulated in the data management portion of the DataAcq SDK.

The following groups of data management functions are available:

- Buffer management functions, and
- List management functions.

The following subsections summarize these functions.

Note: For specific information about each of these functions, refer to the DataAcq SDK online help. See [page 4](#) for information on launching the online help file.

Buffer Management Functions

The Buffer Management functions, listed in [Table 9](#), are a set of object-oriented tools intended for both application and system programmers. When a buffer object is created, a buffer handle (HBUF) is returned. This handle is used in all subsequent buffer manipulation.

Table 9: Buffer Management Functions

Function	Description
oIDmAllocBuffer	Creates a buffer object of a specified number of samples, where each sample is 2 bytes.
oIDmCallocBuffer	Creates a buffer object of a specified number of samples of a specified size.
oIDmCopyBuffer	Copies data from the buffer to the specified array.
oIDmCopyFromBuffer	Copies data from the buffer to the specified array.
oIDmCopyToBuffer	Copies data from the specified array to the buffer.
oIDmFreeBuffer	Deletes a buffer object.
oIDmGetBufferPtr	Gets a pointer to the buffer data.
oIDmGetBufferSize	Gets the physical buffer size (in bytes).
oIDmGetDataBits	Gets the number of valid data bits.
oIDmSetDataBits	Sets the number of valid data bits.
oIDmSetDataWidth	Sets the width of each data sample.
oIDmGetDataWidth	Gets the width of each data sample.
oIDmGetErrorString	Gets the string corresponding to a data management error code value.
oIDmGetMaxSamples	Gets the physical size of the buffer (in samples).
oIDmGetTimeDateStamp	Gets the time and date of the buffer's data.
oIDmSetValidSamples	Sets the number of valid samples in the buffer.
oIDmGetValidSamples	Gets the number of valid samples.
oIDmGetVersion	Gets the version of the data management library.
oIDmMallocBuffer	Creates a buffer object of a specified number of bytes.
oIDmReAllocBuffer	Reallocates a buffer object (alloc() interface).

Table 9: Buffer Management Functions (cont.)

Function	Description
oIDmReCallocBuffer	Reallocates a buffer object (calloc() interface).
oIDmReMallocBuffer	Reallocates a buffer object (malloc() interface).

Buffer List Management Functions

Buffer List Management functions, described in [Table 10](#), provide a straightforward mechanism for handling buffer lists, called *queues*, that the software creates internally as well as other lists that you might want to create. You are not required to use these functions; however, you may find them helpful in your application. Buffer List Management functions are particularly useful when dealing with a device that acquires or outputs continuous data. Refer to Chapter 5 for more information on queues and other lists.

Table 10: Buffer List Management Functions

Function	Description
oIDmCreateList	Creates a user-defined list object.
oIDmEnumBuffers	Enumerates all buffers on a queue or on a list you created.
oIDmEnumLists	Enumerates all queues or lists.
oIDmFreeList	Deletes a user-defined list.
oIDmGetBufferFromList	Removes a buffer from the start of a queue or user-defined list.
oIDmGetListCount	Gets the number of buffers on a queue or user-defined list.
oIDmGetListHandle	Finds the queue or user-defined list that a buffer is on.

Table 10: Buffer List Management Functions (cont.)

Function	Description
oidmGetListIds	Gets a description of the queue or list.
oidmPeekBufferFromList	Gets the handle of the first buffer in the queue or list but does not remove the buffer from the queue or list.
oidmPutBufferToList	Adds a buffer to the end of a queue or list.



Using the DataAcq SDK

System Operations	37
Analog and Digital I/O Operations	41
Counter/Timer Operations.....	81
Simultaneous Operations	110

This chapter provides conceptual information to describe the following operations provided by the DataAcq SDK:

- System operations, described starting on [page 37](#);
- Analog and digital I/O operations, described starting on [page 41](#);
- Counter/timer operations described starting on [page 81](#); and
- Simultaneous operations starting on [page 110](#).

Use this information with the reference information provided in the DataAcq SDK online help when programming your data acquisition devices; refer to [page 4](#) for more information on launching this help file.

System Operations

This DataAcq SDK provides functions to perform the following general system operations:

- Initializing a device,
- Specifying a subsystem,
- Configuring a subsystem,
- Handling errors,
- Handling messages, and
- Releasing a subsystem and driver.

The following subsections describe these operations in more detail.

Initializing a Device

To perform a data acquisition operation, your application program must initialize the device driver for the device you are using with the **olDaInitialize** function. This function returns a device handle, called HDEV. You need one device handle for each device. Device handles allow you to access more than one device in your system.

If you are unsure of the DT-Open Layers devices in your system, use the **olDaEnumBoards** function, which lists the device name, device driver name, and system resources used by each DT-Open Layers device in your system, or the **olDaGetBoardInfo** function, which returns the driver name, model name, and instance number of the specified board, based on its board name.

Once you have initialized a device, you can specify a subsystem, as described in the next section.

Specifying a Subsystem

The DataAcq SDK allows you to define the following subsystems:

- Analog input (A/D subsystem),
- Analog output (D/A subsystem),
- Digital input (DIN subsystem),
- Digital output (DOUT subsystem),
- Counter/timer (C/T subsystem), and
- Serial port (SRL subsystem).

Note: The SRL subsystem is provided for future use. It is not currently used by any DT-Open Layers compatible data acquisition device.

A device can have multiple elements of the same subsystem type. Each of these elements is a subsystem of its own and is identified by a subsystem type and element number. Element numbering is zero-based; that is, the first instance of the subsystem is called element 0, the second instance of the subsystem is called element 1, and so on. For example if two digital I/O ports are on your device, two DIN or DOUT subsystems are available, differentiated as element 0 and element 1.

Once you have initialized the device driver for the specified device, you must specify the subsystem/element on the specified device using the **oIDaGetDASS** function. This function returns a subsystem handle, called HDASS. To access a subsystem, you need one subsystem handle for each subsystem. Subsystem handles allow you to access more than one subsystem on a device.

If you are unsure of the subsystems on a device, use the **oldaEnumSubSystems** or **oldaGetDevCaps** function. **oldaEnumSubSystems** lists the names, types, and number of elements for all subsystems supported by the specified device. **oldaGetDevCaps** returns the number of elements for a specified subsystem type on a specified device.

Note: You can call any function that contains HDASS as a parameter for any subsystem. In some cases, however, the subsystem may not support the particular capability. If this occurs, the subsystem returns an error code indicating that it does not support that function.

3

Once you have specified a subsystem/element, you can configure the subsystem and perform a data acquisition operation, as described in the following section.

Configuring a Subsystem

You configure a subsystem by setting its parameters or capabilities. For more information on the capabilities you can query and specify, refer to the following:

- For analog and digital I/O operations, refer to [page 41](#);
- For the counter/timer operations, refer to [page 81](#), and
- For simultaneous operations, refer to [page 110](#).

Once you have set up the parameters appropriately for the operation you want to perform, call the **oldaConfig** function to configure the parameters before performing the operation.

Handling Errors

An error code is returned by each function in the DataAcq SDK. An error code of 0 indicates that the function executed successfully (no error). Any other error code indicates that an error occurred. Your application program should check the value returned by each function and perform the appropriate action if an error occurs.

Refer to the DataAcq SDK online help for detailed information on the returned error codes and how to proceed should they occur.

Handling Messages

The data acquisition device notifies your application of buffer movement and other events by generating messages.

To determine if the subsystem can post messages, use the **oIDaGetSSCaps** function, specifying the capability **OLSSC_SUP_POSTMESSAGE**. If this function returns a nonzero value, the capability is supported.

Specify the window to receive messages using the **oIDaSetWndHandle** function or the procedure to handle these messages using the **oIDaSetNotificationProcedure** function.

Refer to the DataAcq SDK online help for more information on the messages that can be generated and how to proceed should they occur.

Releasing the Subsystem and the Driver

When you are finished performing data acquisition operations, release the simultaneous start list, if used, using the **oIDaReleaseSSList** function. Then, release each subsystem using the **oIDaReleaseDASS** function. Release the driver and terminate the session using the **oIDaTerminate** function.

Analog and Digital I/O Operations

The DataAcq SDK defines the following capabilities that you can query and/or specify for analog and/or digital I/O operations:

- Data encoding,
- Resolution,
- Channels (including channel type, channel list, channel inhibit list, and synchronous digital I/O list),
- Ranges,
- Gains,
- Filters,
- Data flow modes,
- Triggered scan mode,
- Clock sources,
- Trigger sources,
- Buffers, and
- DMA and interrupt resources.

The following subsections describe these capabilities in more detail.

Data Encoding

For A/D and D/A subsystems only, the DataAcq SDK defines two data encoding types: binary and twos complement.

To determine the data encoding types supported by the subsystem, use the `olDaGetSSCaps` function, specifying the capability `OLSSC_SUP_BINARY` for binary data encoding or `OLSSC_SUP_2SCOMP` for twos complement data encoding. If this function returns a nonzero value, the capability is supported. Use the `olDaSetEncoding` function to specify the data encoding type.

Resolution

To determine if the subsystem supports software-programmable resolution, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_SWRESOLUTION`. If this function returns a nonzero value, the capability is supported.

To determine the number of resolution settings supported by the subsystem, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_NUMRESOLUTION`. To list the actual bits of resolution supported, use the **olDaEnumSSCaps** function, specifying the `OL_ENUM_RESOLUTION` capability.

Use the **olDaSetResolution** function to specify the number of bits of resolution to use for the subsystem.

Channels

Each subsystem (or element of a subsystem type) can have multiple channels. To determine how many channels the subsystem supports, use the **olDaGetSSCaps** function, specifying the `OLSSC_NUMCHANNELS` capability.

Specifying the Channel Type

The DataAcq SDK supports the following channel types:

- **Single-ended** - Use this configuration when you want to measure high-level signals, noise is insignificant, the source of the input is close to the device, and all the input signals are referred to the same common ground.

To determine if the subsystem supports the single-ended channel type, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_SINGLEENDED` capability. If this function returns a nonzero value, the capability is supported.

To determine how many single-ended channels are supported by the subsystem, use the **olDaGetSSCaps** function, specifying the `OLSSC_MAXSECHANS` capability.

Specify the channel type as single-ended for each channel using the **olDaSetChannelType** function.

- **Differential** - Use this configuration when you want to measure low-level signals (less than 1 V), you are using an A/D converter with high resolution (greater than 12 bits), noise is a significant part of the signal, or common-mode voltage exists.

To determine if the subsystem supports the differential channel type, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_DIFFERENTIAL` capability. If this function returns a nonzero value, the capability is supported.

To determine how many differential channels are supported by the subsystem, use the **olDaGetSSCaps** function, specifying the `OLSSC_MAXDICHANS` capability.

Specify the channel type as differential for each channel using the **olDaSetChannelType** function.

Note: For pseudo-differential analog inputs, specify the single-ended channel type; in this case, how you wire these signals determines the configuration. This option provides less noise rejection than the differential configuration, but twice as many analog input channels.

For older model devices, this setting is jumper-selectable and must be specified in the driver configuration dialog.

The channel list is not used to set the channel type.

The following subsections describe how to specify channels.

Specifying a Single Channel

The simplest way to acquire data from or output data to a single channel is to specify the channel for a single-value operation; refer to [page 54](#) for more information on single-value operations.

You can also specify a single channel using a channel list, described in the next section.

Specifying One or More Channels

You acquire data from or output data to one or more channels using a channel list.

The DataAcq SDK provides features that allow you to group the channels in the list sequentially (either starting with 0 or with any other analog input channel) or randomly. In addition, the DataAcq SDK allows you to specify a single channel or the same channel more than once in the list. Your device, however, may limit the order in which you can enter a channel in the channel list.

To determine how the channels can be ordered in the channel list for your subsystem, use the **oIDaGetSSCaps** function, specifying the `OLSSC_RANDOM_CGL` capability. If this function returns a nonzero value, the capability is supported; you can order the channels in the channel list in any order, starting with any channel. If this capability is not supported, use the **oIDaGetSSCaps** function, specifying the `OLSSC_SUP_SEQUENTIAL_CGL` capability. If this function returns a nonzero value, the capability is supported; you must order the channels in the channel list in sequential order, starting with any channel. If this capability is not supported, use the **oIDaGetSSCaps** function, specifying the `OLSSC_SUP_ZEROSEQUENTIAL_CGL` capability. If this function returns a nonzero value, the capability is supported; you must order the channels in the channel list in sequential order, starting with channel 0.

To determine if the subsystem supports simultaneous sample-and-hold mode use the **oIDaGetSSCaps** function, specifying the `OLSSC_SUP_SIMULTANEOUS_SH` capability. If this function returns a nonzero value, the capability is supported. You must enter at least two channels in the channel list. Generally, the first channel is the sample channel and the remaining channels are the hold channels.

The following subsections describe how to specify channels in a channel list.

Specifying the Channel List Size

To determine the maximum size of the channel list for the subsystem, use the **oIDaGetSSCaps** function, specifying the `OLSSC_CGLDEPTH` capability.

Use the **oIDaSetChannelListSize** function to specify the size of the channel list.

Note: The OLSSC_CGLDEPTH capability specifies the maximum size of the channel list, channel inhibit list, synchronous digital I/O list, and gain list.

Specifying the Channels in the Channel List

Use the `oIdaSetChannelListEntry` function to specify the channels in the channel list in the order you want to sample them or output data from them.

The channels are sampled or output in order from the first entry to the last entry in the channel list. Channel numbering is zero-based; that is, the first entry in the channel list is entry 0, the second entry is entry 1, and so on.

For example, if you want to sample channel 4 twice as frequently as channels 5 and 6, you could program the channel list as follows:

Channel-List Entry	Channel	Description
0	4	Sample channel 4.
1	5	Sample channel 5.
2	4	Sample channel 4 again.
3	6	Sample channel 6.

In this example, channel 4 is sampled first, followed by channel 5, channel 4 again, then channel 6.

Inhibiting Channels in the Channel List

If supported, you can set up a channel-inhibit list; this feature is useful if you want to discard values acquired from specific channels, as is typical in simultaneous sample-and-hold applications.

To determine if a subsystem supports a channel-inhibit list, use the **olDaGetSSCaps** function, specifying the **OLSSC_SUP_CHANNELLIST_INHIBIT** capability. If this function returns a nonzero value, the capability is supported.

Using the **olDaSetChannelListEntryInhibit** function, you can enable or disable inhibition for each entry in the channel list. If enabled, the acquired value is discarded after the channel entry is sampled; if disabled, the acquired value is stored after the channel entry is sampled.

In the following example, the values acquired from channels 11 and 9 are discarded and the values acquired from channels 10 and 8 are stored.

Channel-List Entry	Channel	Channel Inhibit Value	Description
0	11	True	Sample channel 11 and discard the value.
1	10	False	Sample channel 10 and store the value.
2	9	True	Sample channel 9 and discard the value.
3	8	False	Sample channel 8 and store the value.

Specifying Synchronous Digital I/O Values in the Channel List

If supported, you can set up a synchronous digital I/O list; this feature is useful if you want to write a digital output value to dynamic digital output channels when an analog input channel is sampled.

To determine if the subsystem supports synchronous (dynamic) digital output operations, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_SYNCHRONOUSDIGITALIO` capability. If this function returns a nonzero value, the capability is supported.

Use the **olDaSetSynchronousDigitalIOUsage** function to enable or disable synchronous (dynamic) digital output operation for a specified subsystem.

Once you enable a synchronous digital output operation, specify the values to write to the synchronous (dynamic) digital output channels using the **olDaSetDigitalIOListEntry** function for each entry in the channel list.

To determine the maximum digital output value that you can specify, use the **olDaGetSSCaps** function, specifying the `OLSSC_MAXDIGITALIOLIST_VALUE` capability.

As each entry in the channel list is scanned, the corresponding value in the synchronous digital I/O list is output to the dynamic digital output channels.

In the following example, when channel 7 is sampled, a value of 1 is output to the dynamic digital output channels. When channel 5 is sampled, a value of 1 is output to the dynamic digital output channels. When channels 6 and 4 are sampled, a value of 0 is output to the dynamic digital output channels.

Channel-List Entry	Channel	Synchronous Digital I/O Value	Description
0	7	1	Sample channel 7 and output a value of 1 to the dynamic digital output channels.
1	5	1	Sample channel 5 and output a value of 1 to the dynamic digital output channels.
2	6	0	Sample channel 6 and output a value of 0 to the dynamic digital output channels.
3	4	0	Sample channel 4 and output a value of 0 to the dynamic digital output channels.

If your device had two dynamic digital output channels and a value of 1 is output (01 in binary format), a value of 1 is written to dynamic digital output channel 0 and a value of 0 is written to dynamic digital output channel 1. Similarly, if a value of 2 is output (10 in binary format), a value of 0 is written to dynamic digital output channel 0 and a value of 1 is written to dynamic digital output channel 1.

Note: If you are controlling sample-and-hold devices with these channels, you may need to program the first channel at the sample logic level and the following channels at the hold logic level; see your device/device driver documentation for details.

Ranges

The range capability applies to A/D and D/A subsystems only.

Depending on your subsystem, you can set the range for the entire subsystem or the range for each channel.

To determine if the subsystem supports the range-per-channel capability, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_RANGEPERCHANNEL` capability. If this function returns a nonzero value, the capability is supported.

To determine how many ranges the subsystem supports, use the **olDaGetSSCaps** function, specifying the `OLSSC_NUMRANGES` capability.

To list the minimum and maximum ranges supported by the subsystem, use the **olDaEnumSSCaps** function, specifying the `OL_ENUM_RANGES` capability.

Use **olDaSetRange** to specify the range for a subsystem. If your subsystem supports the range-per-channel capability, use **olDaSetChannelRange** to specify the range for each channel.

Notes: The channel list is not used to set the range for a channel.

For older device models, the range is jumper-selectable and must be specified in the driver configuration dialog.

Gains

The range divided by the gain determines the effective range for the entry in the channel list. For example, if your device provides a range of ± 10 V and you want to measure a ± 1.5 V signal, specify a range of ± 10 V and a gain of 4; the effective input range for this channel is then ± 2.5 V ($10/4$), which provides the best sampling accuracy for that channel.

The way you specify gain depends on how you specified the channels, as described in the following subsections.

3

Note: If your device supports autoranging for single-value operations, the device can determine the appropriate gain for your range rather than you having to specify it. Refer to [page 54](#) for more information on autoranging.

Specifying the Gain for a Single Channel

The simplest way to specify gain for a single channel is to specify the gain in a single-value operation; refer to [page 54](#) for more information on single-value operations.

You can also specify the gain for a single channel using a gain list, described in the next section.

Specifying the Gain for One or More Channels

You can specify the gain for one or more channels using a gain list. The gain list parallels the channel list. (The two lists together are often referred to as the channel-gain list or CGL.)

To determine if the subsystem supports programmable gain, use the **olDaGetSSCaps** function, specifying the **OLSSC_SUP_PROGRAMGAIN** capability. If this function returns a nonzero value, the capability is supported.

To determine how many gains the subsystem supports, use the **olDaGetSSCaps** function, specifying the **OLSSC_NUMGAINS** capability.

To list the gains supported by the subsystem, use the **olDaEnumSSCaps** function, specifying the **OL_ENUM_GAINS** capability.

Specify the gain for each entry in the channel list using the **olDaSetGainListEntry** function.

In the following example, a gain of 2 is applied to channel 5, a gain of 4 is applied to channel 6, and a gain of 1 is applied to channel 7.

Channel-List Entry	Channel	Gain	Description
0	5	2	Sample channel 5 using a gain of 2.
1	6	4	Sample channel 6 using a gain of 4.
2	7	1	Sample channel 7 using a gain of 1.

Note: If your subsystem does not support programmable gain, enter a value of 1 for all entries.

If your subsystem does not support the gain-per-channel capability, set all entries in the gain list to the same value.

Filters

This capability applies to A/D and D/A subsystems only.

Depending on your subsystem, you can specify a filter for each channel. To determine if the subsystem supports a filter for each channel, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_FILTERPERCHAN` capability. If this function returns a nonzero value, the capability is supported.

To determine how many filters the subsystem supports, use the **olDaGetSSCaps** function, specifying the `OLSSC_NUMFILTERS` capability.

To list the cut-off frequency of all filters supported by the subsystem, use the **olDaEnumSSCaps** function, specifying the `OL_ENUM_FILTERS` capability.

If the subsystem supports filtering per channel, specify the filter for each channel using the **olDaSetChannelFilter** function. The filter is equal to or greater than a cut-off frequency that you supply.

Notes: The channel list is not used to set the filter for a channel.

If the subsystem supports more than one filter but does not support a filter per channel, the filter specified for channel 0 is used for all channels.

Data Flow Modes

The DataAcq SDK defines the following data flow modes for A/D, D/A, DIN, and DOUT subsystems:

- Single value, and
- Continuous (post-trigger, pre-trigger, and about-trigger).

The following subsections describe these data flow modes in detail.

Single-Value Operations

Single-value operations are the simplest to use but offer the least flexibility and efficiency. In a single-value operation, a single data value is read or written at a time. The result is returned immediately.

To determine if the subsystem supports single-value operations, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_SINGLEVALUE`. If this function returns a nonzero value, the capability is supported.

Specify the operation mode as `OL_DF_SINGLEVALUE` using the **olDaSetDataFlow** function.

Some devices also support autoranging for single-value analog input operations, where the device determines the best gain for the specified range. To determine if the subsystem supports autoranging for single-value operations, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_SINGLEVALUE_AUTORANGE`. If this function returns a nonzero value, the capability is supported.

If autoranging is supported, use the **olDaGetSingleValueEx** function to specify the range and analog input channel and to have the software determine the best gain for the range. The device then acquires the data from the specified channel and returns the result immediately in both counts and engineering units (such as voltage).

If autoranging is not supported, use the **olDaGetSingleValue** function to acquire a single value from an analog or digital input channel. You specify the channel and gain, then the device acquires the data from the specified channel and returns the result immediately, in counts. If you later want to convert the count value to engineering units, you can use the **olDaCodeToVolts** function. Similarly, if you want to convert the engineering units to counts, you can use the **olDaVoltsToCode** function.

To output a single value to an analog or digital output channel, use the **olDaPutSingleValue** function. You specify the channel, gain, and value, and the device outputs the single value to the specified analog or digital channel immediately.

For a single-value operation, you cannot specify a channel-gain list, clock source, trigger source, DMA channel, or buffer.

Single-value operations stop automatically when finished; you cannot stop a single-value operation manually.

Continuous Operations

For a continuous operation, you can specify any supported subsystem capability, including a channel-gain list, clock source, trigger source, pre-trigger source, retrigger source, DMA channel, and buffer.

Call the **olDaStart** function to start a continuous operation.

To stop a continuous operation, perform either an orderly stop using the **olDaStop** function or an abrupt stop using the **olDaAbort** or **olDaReset** function.

In an orderly stop (**olDaStop**), the device finishes acquiring the specified number of samples, stops all subsequent acquisition, and transfers the acquired data to a buffer on the done queue; all subsequent triggers or retriggers are ignored. (Refer to [page 71](#) for more information on buffers and queues.)

In an abrupt stop (**olDaAbort**), the device stops acquiring samples immediately; the acquired data is transferred to a buffer and put on the done queue; however, the buffer may not be completely filled. All subsequent triggers or retriggers are ignored.

The **olDaReset** function reinitializes the subsystem after stopping it abruptly.

Note: For analog output operations, you can also stop the operation by not sending new data to the device. The operation stops when no more data is available.

Some subsystems also allow you to pause the operation using the **olDaPause** function and to resume the paused operation using the **olDaContinue** function. To determine if pausing is supported, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_PAUSE` capability. If this function returns a nonzero value, the capability is supported.

The following continuous modes are supported by the DataAcq SDK: continuous (post-trigger), continuous pre-trigger, and continuous about-trigger. These modes are described in the following subsections.

Continuous Post-Trigger Mode

Use continuous post-trigger when you want to acquire or output data continuously when a trigger occurs.

To determine if the subsystem supports continuous (post-trigger) operations, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_CONTINUOUS`. If this function returns a nonzero value, the capability is supported.

For continuous (post-trigger) mode, specify the operation mode as `OL_DF_CONTINUOUS` using the **olDaSetDataFlow** function.

Use the **olDaSetTrigger** function to specify the trigger source that starts the operation. Refer to [page 67](#) for more information on supported trigger sources.

When the post-trigger event is detected, the device cycles through the channel list, acquiring and/or outputting the value for each entry in the channel list; this process is defined as a scan. The device then wraps to the start of the channel list and repeats the process continuously until either the allocated buffers are filled or you stop the operation. Refer to [page 44](#) for more information on channel lists; refer to [page 71](#) for more information on buffers.

[Figure 1](#) illustrates continuous post-trigger mode using a channel list of three entries: channel 0, channel 1, and channel 2. In this example, post-trigger analog input data is acquired on each clock pulse of the A/D sample clock; refer to [page 65](#) for more information on clock sources. The device wraps to the beginning of the channel list and repeats continuously.

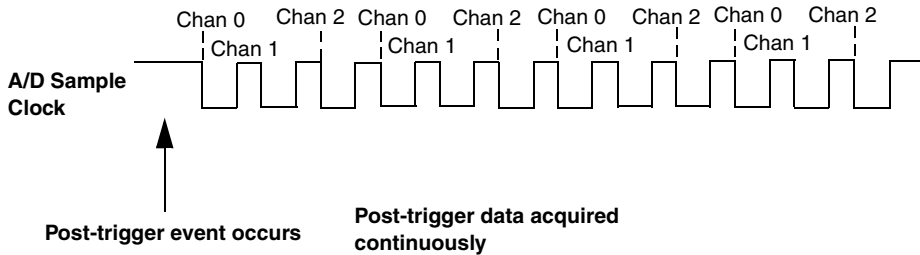


Figure 1: Continuous Post-Trigger Mode

Continuous Pre-Trigger Mode

Use continuous pre-trigger mode when you want to acquire data before a specific external event occurs.

To determine if the subsystem supports continuous pre-trigger mode, use the `olDaGetSSCaps` function, specifying the `_SUP_CONTINUOUS_PRETRIG` capability. If this function returns a nonzero value, the capability is supported.

Specify the operation mode as `OL_DF_CONTINUOUS_PRETRIG` using the `olDaSetDataFlow` function.

Pre-trigger acquisition starts when the device detects the pre-trigger source and stops when the device detects an external post-trigger source, indicating that the first post-trigger sample was acquired (this sample is ignored).

Use the `olDaSetPretriggerSource` function to specify the trigger source that starts the pre-trigger operation (generally this is a software trigger). Specify the post-trigger source that stops the operation using `olDaSetTrigger`. Refer to [page 67](#) and to your device/driver documentation for supported pre-trigger and post-trigger sources.

Figure 2 illustrates continuous pre-trigger mode using a channel list of three entries: channel 0, channel 1, and channel 2. In this example, pre-trigger analog input data is acquired on each clock pulse of the A/D sample clock; refer to [page 65](#) for more information on clock sources. The device wraps to the beginning of the channel list and the acquisition repeats continuously until the post-trigger event occurs. When the post-trigger event occurs, acquisition stops.

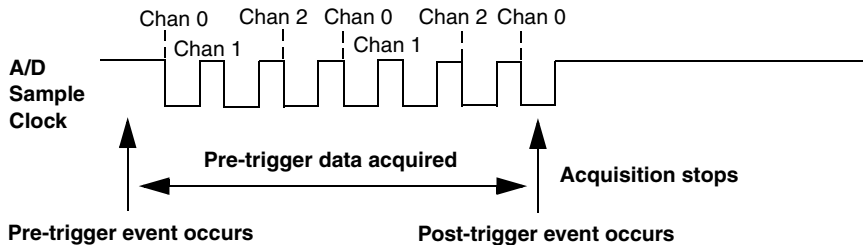


Figure 2: Continuous Pre-Trigger Mode

Continuous About-Trigger Mode

Use continuous about-trigger mode when you want to acquire data both before and after a specific external event occurs. This operation is equivalent to doing both a pre-trigger and a post-trigger acquisition.

To determine if the subsystem supports continuous about-trigger mode, use the `olDaGetSSCaps` function, specifying the `OLSSC_SUP_CONTINUOUS_ABOUTTRIG` capability. If this function returns a nonzero value, the capability is supported.

Specify the operation mode as `OL_DF_CONTINUOUS_ABOUTTRIG` using the `olDaSetDataFlow` function.

The about-trigger acquisition starts when the device detects the pre-trigger source. When it detects an external post-trigger source, the device stops acquiring pre-trigger data and starts acquiring post-trigger data.

Use the **oIDaSetPretriggerSource** function to specify the pre-trigger source that starts the pre-trigger operation (this is generally a software trigger) and **oIDaSetTrigger** to specify the trigger source that stops the pre-trigger acquisition and starts the post-trigger acquisition. Refer to [page 67](#) and to your device/driver documentation for supported pre-trigger and post-trigger sources.

The about-trigger operation stops when the specified number of post-trigger samples has been acquired or when you stop the operation.

[Figure 3](#) illustrates continuous about-trigger mode using a channel list of three entries: channel 0, channel 1, and channel 2. In this example, pre-trigger analog input data is acquired on each clock pulse of the A/D sample clock. The device wraps to the beginning of the channel list and the acquisition repeats continuously until the post-trigger event occurs. When the post-trigger event occurs, post-trigger acquisition begins on each clock pulse of the A/D sample clock; refer to [page 65](#) for more information on clock sources. The device wraps to the beginning of the channel list and acquires post-trigger data continuously.

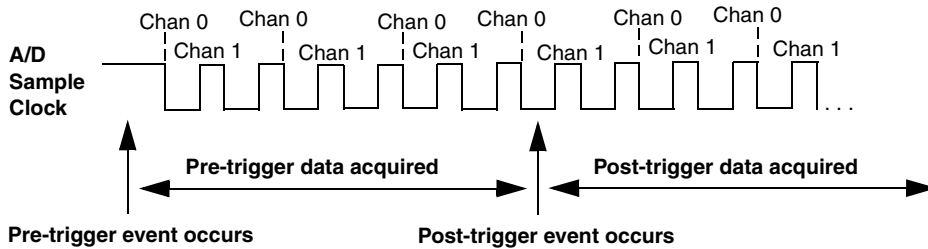


Figure 3: Continuous About-Trigger Mode

3

Triggered Scan Mode

In triggered scan mode, the device scans the entries in a channel-gain list a specified number of times when it detects the specified trigger source, acquiring the data for each entry that is scanned.

To determine if the subsystem supports triggered scan mode, use the **olDaGetSSCaps** function, specifying the **OLSSC_SUP_TRIGSCAN** capability. If this function returns a nonzero value, the capability is supported. Note that you cannot use triggered scan mode with single-value operations.

To enable (or disable) triggered scan mode, use the **olDaSetTriggeredScanUsage** function.

To determine the maximum number of times that the device can scan the channel-gain list per trigger, use the **olDaGetSSCaps** function, specifying the **OLSSC_MAXMULTISCAN** capability.

Use the **olDaSetMultiscanCount** function to specify the number of times to scan the channel-gain list per trigger.

The DataAcq SDK defines the following retrigger modes for a triggered scan; these retrigger modes are described in the following subsections:

- Scan-per-trigger,
- Internal retrigger, and
- Retrigger extra.

Note: If your device driver supports it, retrigger extra is the preferred triggered scan mode.

Scan-Per-Trigger Mode

Use scan-per-trigger mode if you want to accurately control the period between conversions of individual channels and retrigger the scan based on an internal or external event. In this mode, the retrigger source is the same as the initial trigger source.

To determine if the subsystem supports scan-per-trigger mode, use the **olDaGetSSCaps** function, specifying the **OLSSC_SUP_RETRIGGER_SCAN_PER_TRIGGER** capability. If this function returns a nonzero value, the capability is supported.

Specify the retrigger mode as scan-per-trigger using the **olDaSetRetriggerMode** function.

When it detects an initial trigger (post-trigger mode only), the device scans the channel-gain list a specified number of times (determined by the **olDaSetMultiscanCount** function), then stops. When the external retrigger occurs, the process repeats.

The conversion rate of each channel in the scan is determined by the frequency of the A/D sample clock; refer to [page 65](#) for more information on clock sources. The conversion rate of each scan is determined by the period between retriggers; therefore, it cannot be accurately controlled. The device ignores external triggers that occur while it is acquiring data. Only retrigger events that occur when the device is waiting for a trigger are detected and acted on. Some devices may generate an OLDA_WM_TRIGGER_ERROR message.

Internal Retrigger Mode

Use internal retrigger mode if you want to accurately control both the period between conversions of individual channels in a scan and the period between each scan.

To determine if the subsystem supports internal retrigger mode, use the **olDaGetSSCaps** function, specifying the OLSSC_SUP_RETRIGGER_INTERNAL capability. If this function returns a nonzero value, the capability is supported.

Specify the retrigger mode as internal using the **olDaSetRetriggerMode** function.

The conversion rate of each channel in the scan is determined by the frequency of the A/D sample clock; refer to [page 65](#) for more information on clock sources. The conversion rate between scans is determined by the frequency of the internal retrigger clock on the device. You specify the frequency on the internal retrigger clock using the **olDaSetRetriggerFrequency** function.

When it detects an initial trigger (pre-trigger source or post-trigger source), the device scans the channel-gain list a specified number of times (determined by the **olDaSetMultiscanCount** function), then stops. When the internal retrigger occurs, determined by the frequency of the internal retrigger clock, the process repeats.

It is recommended that you set the retrigger frequency as follows:

$$\text{Min. Retrigger Period} = \frac{\# \text{ of CGL entries} \times \# \text{ of CGLs per trigger} + 2 \mu\text{s}}{\text{A/D sample clock frequency}}$$

$$\text{Max. Retrigger Frequency} = \frac{1}{\text{Min. Retrigger Period}}$$

For example, if you are using 512 channels in the channel-gain list (CGL), scanning the channel-gain list 256 times every trigger or retrigger, and using an A/D sample clock with a frequency of 1 MHz, set the maximum retrigger frequency to 7.62 Hz, since

$$7.62 \text{ Hz} = \frac{1}{\frac{(512 * 256) + 2 \mu\text{s}}{1 \text{ MHz}}}$$

Retrigger Extra Mode

Use retrigger extra mode if you want to accurately control the period between conversions of individual channels and retrigger the scan on a specified retrigger source; the retrigger source can be any of the supported trigger sources.

To determine if the subsystem supports retrigger extra mode, use the **olDaGetSSCaps** function, specifying the **OLSSC_SUP_RETRIGGER_EXTRA** capability. If this function returns a nonzero value, the capability is supported.

Specify the retrigger mode as retrigger extra using the **olDaSetRetriggerMode** function.

Use the **olDaSetRetrigger** function to specify the retrigger source. Refer to [page 67](#) and to your device/device driver documentation for supported retrigger sources.

The conversion rate of each channel in the scan is determined by the frequency of the A/D sample clock; refer to [page 65](#) for more information on clock sources. The conversion rate of each scan is determined by the period between retriggers.

If you are using an internal retrigger, specify the period between retriggers using `olDaSetRetriggerFrequency` (see [page 63](#)). If you are using an external retrigger, the period between retriggers cannot be accurately controlled. The device ignores external triggers that occur while it is acquiring data. Only retrigger events that occur when the device is waiting for a trigger are detected and acted on. Some devices may generate an `OLDA_WM_TRIGGER_ERROR` message.

Clock Sources

The DataAcq SDK defines internal, external, and extra clock sources, described in the following subsections. Note that you cannot specify a clock source for single-value operations.

Internal Clock Source

The internal clock is the clock source on the device that paces data acquisition or output for each entry in the channel-gain list.

To determine if the subsystem supports an internal clock, use the `olDaGetSSCaps` function, specifying the `OLSSC_SUP_INTCLOCK` capability. If this function returns a nonzero value, the capability is supported.

Specify the clock source as internal using the `olDaSetClockSource` function. Then, use the `olDaSetClockFrequency` function to specify the frequency at which to pace the operation.

To determine the maximum frequency that the subsystem supports, use the **olDaGetSSCapsEx** function, specifying the **OLSSCE_MAXTHROUGHPUT** capability. To determine the minimum frequency that the subsystem supports, use the **olDaGetSSCapsEx** function, specifying the **OLSSCE_MINTHROUGHPUT** capability.

Note: According to sampling theory (Nyquist Theorem), you should specify a frequency for an A/D signal that is at least twice as fast as the input's highest frequency component. For example, to accurately sample a 20 kHz signal, specify a sampling frequency of at least 40 kHz. Doing so avoids an error condition called *aliasing*, in which high frequency input components erroneously appear as lower frequencies after sampling.

External Clock Source

The external clock is a clock source attached to the device that paces data acquisition or output for each entry in the channel-gain list. This clock source is useful when you want to pace at rates not available with the internal clock or if you want to pace at uneven intervals.

To determine if the subsystem supports an external clock, use the **olDaGetSSCaps** function, specifying the **OLSSC_SUP_EXTCLOCK** capability. If this function returns a nonzero value, the capability is supported.

Specify the clock source as external using the **olDaSetClockSource** function. Then, use the **olDaSetExternalClockDivider** to specify the clock divider used to determine the frequency at which to pace the operation; the clock input source divided by the clock divider determines the frequency of the clock signal.

To determine the maximum clock divider that the subsystem supports, use the **oldaGetSSCapsEx** function, specifying the **OLSSCE_MAXCLOCKDIVIDER** capability. To determine the minimum clock divider that the subsystem supports, use the **oldaGetSSCapsEx** function, specifying the **OLSSCE_MINCLOCKDIVIDER** capability.

Extra Clock Source

Your device driver may define extra clock sources that you can use to pace acquisition or output operations.

To determine how many extra clock sources are supported by your subsystem, use the **oldaGetSSCaps** function, specifying the **OLSSC_NUMEXTRACLOCKS** capability. Refer to your device/driver documentation for a description of the extra clock sources.

The extra clock sources may be internal or external. Refer to the previous sections for information on how to specify internal and external clocks and their frequencies or clock dividers.

Trigger Sources

The DataAcq SDK defines the following trigger sources:

- Software (internal) trigger,
- External digital (TTL) trigger,
- External analog threshold (positive) trigger,
- External analog threshold (negative) trigger,
- Analog event trigger,
- Digital event trigger,
- Timer event trigger, and
- Extra trigger.

To specify a post-trigger source, use the **olDaSetTrigger** function; refer to [page 57](#) for more information. To specify a pre-trigger source, use the **olDaSetPretriggerSource** function; see [page 58](#) for more information. To specify a retrigger source, use the **olDaSetRetrigger** function; see [page 64](#) for more information.

The following subsections describe these trigger sources. Note that you cannot specify a trigger source for single-value operations.

Software (Internal) Trigger Source

A software trigger occurs when you start the operation; internally, the computer writes to the device to begin the operation.

To determine if the subsystem supports a software trigger, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_SOFTTRIG`. If this function returns a nonzero value, the capability is supported.

External Digital (TTL) Trigger Source

An external digital trigger is a digital (TTL) signal attached to the device.

To determine if the subsystem supports an external digital trigger, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_EXTERNTRIG`. If this function returns a nonzero value, the capability is supported.

External Analog Threshold (Positive) Trigger Source

An external analog threshold (positive) trigger is generally either an analog signal from an analog input channel or an external analog signal attached to the device. An analog trigger occurs when the device detects a transition from a negative to positive value that crosses a threshold value. The threshold level is generally set using a D/A subsystem on the device.

To determine if the subsystem supports analog threshold triggering (positive polarity), use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_THRESHTRIGPOS`. If this function returns a nonzero value, the capability is supported.

Refer to your device/device driver documentation for a description of this trigger source.

External Analog Threshold (Negative) Trigger Source

An external analog threshold (negative) trigger is generally either an analog signal from an analog input channel or an external analog signal attached to the device. An analog trigger event occurs when the device detects a transition from a positive to negative value that crosses a threshold value. The threshold level is generally set using a D/A subsystem on the device.

To determine if the subsystem supports analog threshold triggering (negative polarity), use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_THRESHTRIGNEG`. If this function returns a nonzero value, the capability is supported.

Refer to your device/device driver documentation for a description of this trigger source.

Analog Event Trigger Source

For this trigger source, a trigger is generated when an analog event occurs. To determine if the subsystem supports an analog event trigger, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_ANALOGEVENTTRIG`. If this function returns a nonzero value, the capability is supported.

Digital Event Trigger Source

For this trigger source, a trigger is generated when a digital event occurs. To determine if the subsystem supports a digital event trigger, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_DIGITALEVENTTRIG`. If this function returns a nonzero value, the capability is supported.

Timer Event Trigger Source

For this trigger source, a trigger is generated when a counter/timer event occurs. To determine if the subsystem supports a timer event trigger, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_TIMEREVENTTRIG`. If this function returns a nonzero value, the capability is supported.

Extra Trigger Source

Extra trigger sources may be defined by your device driver. To determine how many extra triggers are supported by the subsystem, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_NUMEXTRATRIGGERS`. Refer to your device/driver documentation for a description of these triggers.

The extra trigger sources may be internal or external. Refer to the previous sections for information on how to specify internal and external triggers.

Buffers

The buffering capability usually applies to A/D and D/A subsystems only. Note that you cannot use a buffer with single-value operations.

A data buffer is a memory location that you allocate in host memory. This memory location is used to store data for continuous input and output operations.

To determine if the subsystem supports buffers, use the **oIDaGetSSCaps** function, specifying the capability **OLSSC_SUP_BUFFERING**. If this function returns a nonzero value, the capability is supported.

Buffers are stored on one of three queues: the ready queue, the inprocess queue, or the done queue. These queues are described in more detail in the following subsections.

Ready Queue

For input operations, the ready queue holds buffers that are empty and ready for input. For output operations, the ready queue holds buffers that you have filled with data and that are ready for output.

Allocate the buffers using the **oIDmMallocBuffer**, **oIDmAllocBuffer**, or **oIDmCallocBuffer** function. **oIDmAllocBuffer** allocates a buffer of samples, where each sample is 2 bytes; **oIDmCallocBuffer** allocates a buffer of samples of a specified size; **oIDmMallocBuffer** allocates a buffer in bytes.

For analog input operations, it is recommended that you allocate a minimum of three buffers; for analog output operations, you can allocate one or more buffers. The size of the buffers should be at least as large as the sampling or output rate; for example, if you are using a sampling rate of 100 ksamples/s (100 kHz), specify a buffer size of 100,000 samples.

Once you have allocated the buffers (and, for output operations, filled them with data), put the buffers on the ready queue using the **oIDaPutBuffer** function.

For example, assume that you are performing an analog input operation, that you allocated three buffers, and that you put these buffers on the ready queue. The queues appear on the ready queue as shown in [Figure 4](#).

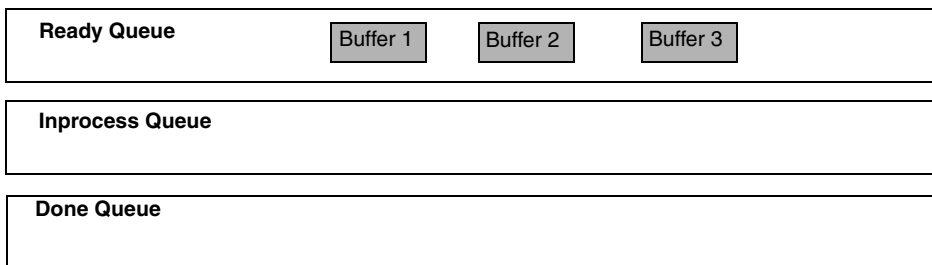


Figure 4: Example of the Ready Queue

Inprocess Queue

When you start a continuous (post-trigger, pre-trigger, or about-trigger) operation, the data acquisition device takes the first available buffer from the ready queue and places it on the inprocess queue.

The inprocess queue holds the buffer that the specified data acquisition device is currently filling (for input operations) or outputting (for output operations). The buffer is filled or emptied at the specified clock rate.

Continuing with the previous example, when you start the analog input operation, the driver takes the first available buffer (Buffer 1, in this case), puts it on the inprocess queue, and starts filling it with data. The queues appear as shown in [Figure 5](#).

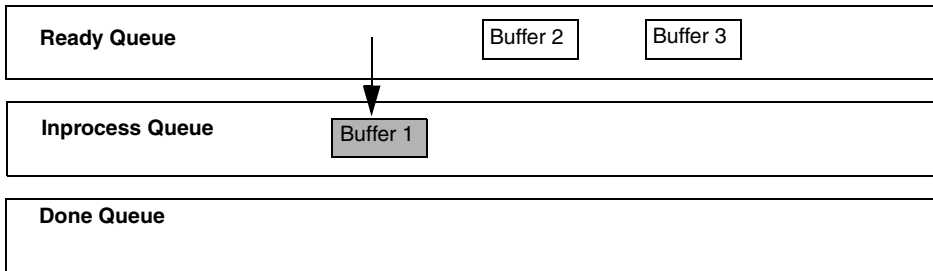


Figure 5: Example of the Inprocess Queue

If required, you can use the **olDaFlushFromBufferInprocess** function to transfer data from a partially-filled buffer on an inprocess queue to a buffer you create (if this capability is supported). Typically, you would use this function when your data acquisition operation is running slowly.

To determine if the subsystem supports transferring data from a buffer on the inprocess queue, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_INPROCESSFLUSH` capability. If this function returns a nonzero value, this capability is supported.

Note: Some devices transfer data to the host in segments instead of one sample at a time. For example, data from some boards is transferred to the host in 64 byte segments; the number of valid samples is always a multiple of 64 depending on the number of samples transferred to the host when `oldaFlushFromBufferInProcess` was called. It is up to your application to take this into account when flushing an inprocess buffer. Refer to your device documentation for more information.

Done Queue

Once the data acquisition device has filled the buffer (for input operations) or emptied the buffer (for output operations), the buffer is moved from the inprocess queue to the done queue. Then, either the `OLDA_WM_BUFFER_DONE` message is generated when the buffer contains post-trigger data, or in the case of pre-trigger and about-trigger acquisitions, an `OLDA_WM_PRETRIGGER_BUFFER_DONE` message is generated when the buffer contains pre-trigger data.

Note: For pre-trigger acquisitions only, when the operation completes or you stop a pre-trigger acquisition, the `OLDA_WM_QUEUE_STOPPED` message is also generated.

Continuing with the previous example, the queues appear as shown in [Figure 6](#) when you get the first `OLDA_WM_BUFFER_DONE` message.

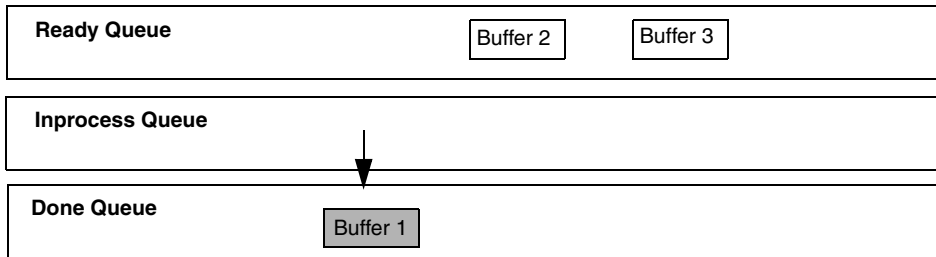


Figure 6: Example of the Done Queue

Then, the driver moves Buffer 2 from the ready queue to the inprocess queue and starts filling it with data. When Buffer 2 is filled, Buffer 2 is moved to the done queue and another `OLDA_WM_BUFFER_DONE` message is generated.

The driver then moves Buffer 3 from the ready queue to the inprocess queue and starts filling it with data. When Buffer 3 is filled, Buffer 3 is moved to the done queue and another `OLDA_WM_BUFFER_DONE` message is generated. [Figure 7](#) shows how the buffers are moved.

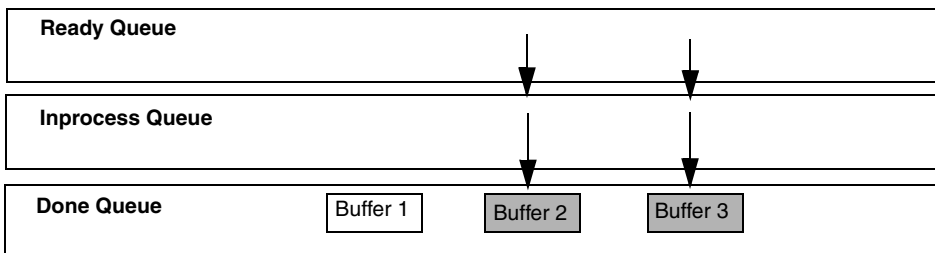


Figure 7: How Buffers are Moved to the Done Queue

If you transferred data from an inprocess queue to a new buffer using **olDaFlushFromBufferInprocess**, the new buffer is put on the done queue for your application to process. When the buffer on the inprocess queue finishes being filled, this buffer is also put on the done queue; the buffer contains only the samples that were not previously transferred.

Buffer and Queue Management

Each time it gets an OLDA_WM_BUFFER_DONE message, your application program should remove the buffers from the done queue using the **olDaGetBuffer** buffer management function.

Your application program can then process the data in the buffer. For an input operation, you can copy the data from the buffer to an array in your application program using the **olDmGetBufferPtr** function. For continuously paced analog output operations, you can fill the buffer with new output data using the **olDaGetBufferPtr** function.

If you want to convert the count value to engineering units, you can use the **olDaCodeToVolts** function. Similarly, if you want to convert the engineering units to counts, you can use the **olDaVoltsToCode** function.

When you are finished processing the data, you can put the buffer back on the ready queue using the **olDaPutBuffer** function if you want your operation to continue.

For example, assume that you processed the data from Buffer 1 and put it back on the ready queue. The queues would appear as shown in [Figure 8](#).

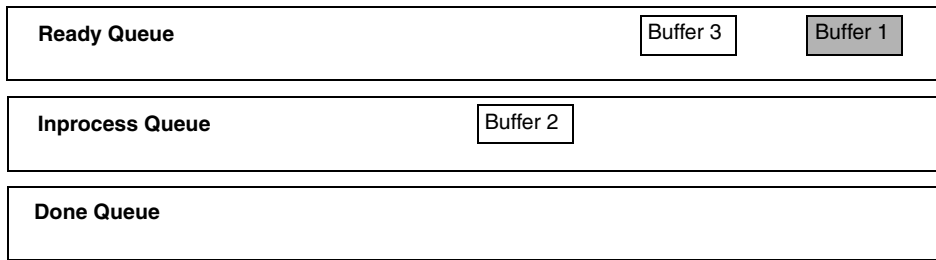


Figure 8: Putting Buffers Back on the Ready Queue

When the data acquisition operation is finished, use the **oldaFlushBuffers** function to transfer any data buffers left on the subsystem's ready queue to the done queue.

Once you have processed the data in the buffers, remove the buffers from the done queue using the **oldaFreeBuffer** function.

Note: For analog output operations only, the `OLDA_WM_IO_COMPLETE` message is generated when the last data point has been output from the analog output channel. In some cases, this message is generated well after the data is transferred from the buffer (when the `OLDA_WM_BUFFER_DONE` and `OLDA_WM_QUEUE_DONE` messages are generated).

Buffer Wrap Modes

Most Keithley data acquisition devices can provide gap-free data, meaning no samples are missed when data is acquired or output. You can acquire gap-free data by manipulating data buffers so that no gaps exist between the last sample of the current buffer and the first sample of the next buffer.

Note: The number of DMA channels, number of buffers, and buffer size are critical to the device's ability to provide gap-free data. It is also critical that the application process the data in a timely fashion.

If you want to acquire gap-free input data, it is recommended that you specify a buffer wrap mode of *none* using the **oldaSetWrapMode** buffer management function. When a buffer wrap mode of *none* is selected, if you process the buffers and put them back on the ready queue in a timely manner, the operation continues indefinitely. When no buffers are available on the ready queue, the operation stops, and an `OLDA_WM_QUEUE_DONE` message is generated.

If you want to continuously reuse the buffers in the queues and you are not concerned with gap-free data, specify *multiple* buffer wrap mode using **oldaSetWrapMode**. When multiple wrap mode is selected and no buffers are available on the ready queue, the driver moves the oldest buffer from the done queue to the inprocess queue (regardless of whether you have processed its data), and overwrites the data in the buffer. This process continues indefinitely unless you stop it. When it reuses a buffer on the done queue, the driver generates an `OLDA_WM_BUFFER_REUSED` message.

If you want to perform gap-free analog output operations, specify *single* wrap mode using **olDaSetWrapMode**. When single wrap mode is specified, a single buffer is reused continuously. In this case, the driver moves the buffer from the ready queue to the inprocess queue and outputs the data from the buffer. However, when the buffer is emptied, the driver (or device) reuses the data and continuously outputs it. This process repeats indefinitely until you stop it. When you stop the operation, the buffer is moved to the done queue. Typically, no messages are posted in this mode until you stop the operation.

To determine the buffer wrap modes available for the subsystem, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_WRPSINGLE` (for single wrap mode) or `OLSSC_SUP_WRPMULTIPLE` (for multiple wrap mode). If this function returns a nonzero value, the capability is supported.

DMA and Interrupt Resources

You cannot use DMA or interrupt resources for single-value operations.

To determine if your subsystem supports interrupt resources, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_INTERRUPT`. If this function returns a nonzero value, the capability is supported.

Generally, you specify interrupt resources on the device itself or in the driver configuration dialog.

To determine if gap-free data acquisition is supported, use the **olDaGetSSCaps** function, specifying `OLSSC_SUP_GAPFREE_NODMA` (for gap free data using no DMA channels), `OLSSC_SUP_GAPFREE_SINGLEDMA` (for gap free data using one DMA channel), or `OLSSC_SUP_GAPFREE_DUALDMA` (for gap free data using two DMA channels). If this function returns a nonzero value, the capability is supported.

To determine how many DMA channels are supported, use the **oIDaGetSSCaps** function, specifying the capability `OLSSC_NUMDMACHANS`.

Use the **oIDaSetDmaUsage** function to specify the number of DMA channels to use. These channels must also be specified in the driver configuration dialog.

Note: DMA channels are a limited resource and the request may not be honored if the requested number of channels is unavailable. For example, suppose that a device that supports both A/D and D/A subsystems provides hardware for two DMA channels, and that one DMA channel is currently allocated to the A/D subsystem. In this case, a request to the D/A subsystem to use two DMA channels will fail.

Counter/Timer Operations

Each user counter/timer channel accepts a clock input signal and gate input signal and outputs a clock output signal (also called a pulse output signal), as shown in [Figure 9](#).

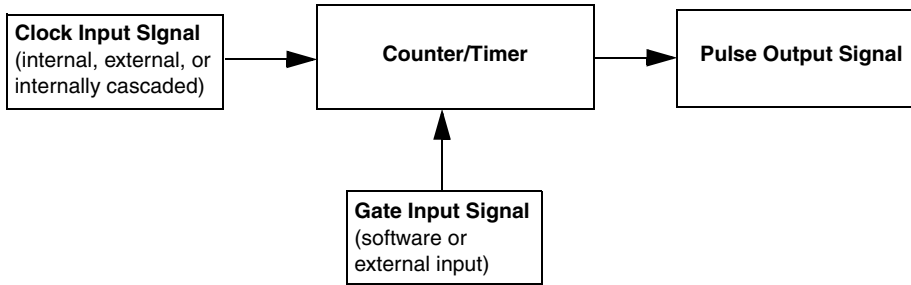


Figure 9: Counter/Timer Channel

Each counter/timer channel corresponds to a counter/timer (C/T) subsystem. To specify the counter to use in software, specify the appropriate C/T subsystem. For example, counter 0 corresponds to C/T subsystem element 0; counter 3 corresponds to C/T subsystem element 3.

The DataAcq SDK defines the following capabilities that you can query and/or configure for counter/timer operations:

- Counter/timer operation mode,
- Clock source,
- Gate source,

- Pulse output type, and
- Pulse output duty cycle.

The following subsections describe these capabilities in more detail.

Counter/Timer Operation Mode

The DataAcq SDK supports the following counter/timer operations:

- Event counting,
- Up/down counting,
- Frequency measurement,
- Edge-to-edge measurement,
- Rate generation (continuous pulse output),
- One-shot, and
- Repetitive one-shot.

The following subsections describe these counter/timer operations.

Event Counting

Use event counting mode to count events from the counter's associated clock input source.

To determine if the subsystem supports event counting, use the **olDaGetSSCaps** function, specifying the capability **OLSSC_SUP_CTMODE_COUNT**. If this function returns a nonzero value, the capability is supported.

To specify an event counting operation, use the **olDaSetCTMode** function, specifying the **OL_CTMODE_COUNT** parameter.

Specify the C/T clock source for the operation. In event counting mode, an external C/T clock source is more useful than the internal C/T clock source; refer to [page 101](#) for more information on specifying the C/T clock source.

Also specify the gate type that enables the operation; refer to [page 104](#) for more information on specifying the gate type.

Start an event counting operation using the **olDaStart** function. To read the current number of events counted, use the **olDaReadEvents** function.

To stop the event counting operation, call **olDaStop**, **olDaAbort**, or **olDaReset**; **olDaReset** function stops the operation and reinitializes the subsystem after stopping it.

Some subsystems also allow you to pause the operation using the **olDaPause** function and then resume the paused operation using the **olDaContinue** function. To determine if pausing is supported, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_PAUSE` capability. If this function returns a nonzero value, the capability is supported.

[Figure 10](#) shows an example of an event counting operation. In this example the gate type is low level.

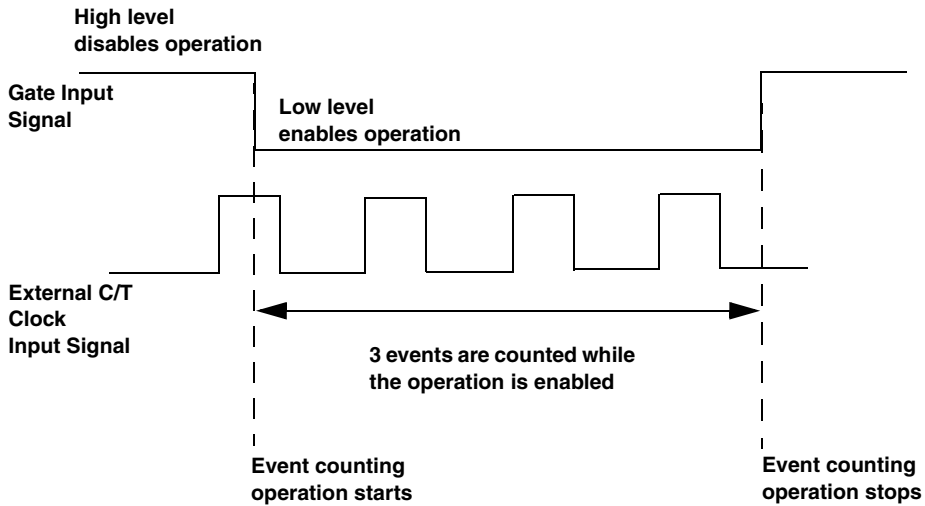


Figure 10: Example of Event Counting

Up/Down Counting

Use up/down counting mode to increment or decrement the number of rising edges that occur on the counter's associated clock input, depending on the level of the counter's associated gate signal. If the gate signal is high, the C/T increments; if the gate signal is low, the C/T decrements.

To determine if the subsystem supports up/down counting, use the **oIDaGetSSCaps** function, specifying the capability **OLSSC_SUP_CTMODE_UP_DOWN**. If this function returns a nonzero value, the capability is supported.

To specify an up/down counting operation, use the **oIDaSetCTMode** function, specifying the **OL_CTMODE_UP_DOWN** parameter.

Specify the C/T clock source for the operation as external. Note that you do not specify the gate type in software.

Start an up/down counting operation using the **olDaStart** function. To read the current number of rising edges counted, use the **olDaReadEvents** function.

To stop the event counting operation, call **olDaStop**, **olDaAbort**, or **olDaReset**; **olDaReset** function stops the operation and reinitializes the subsystem after stopping it.

Some subsystems also allow you to pause the operation using the **olDaPause** function and then resume the paused operation using the **olDaContinue** function. To determine if pausing is supported, use the **olDaGetSSCaps** function, specifying the **OLSSC_SUP_PAUSE** capability. If this function returns a nonzero value, the capability is supported.

Figure 11 shows an example of an up/down counting operation. The counter increments when the gate signal is high and decrements when the gate signal is low.

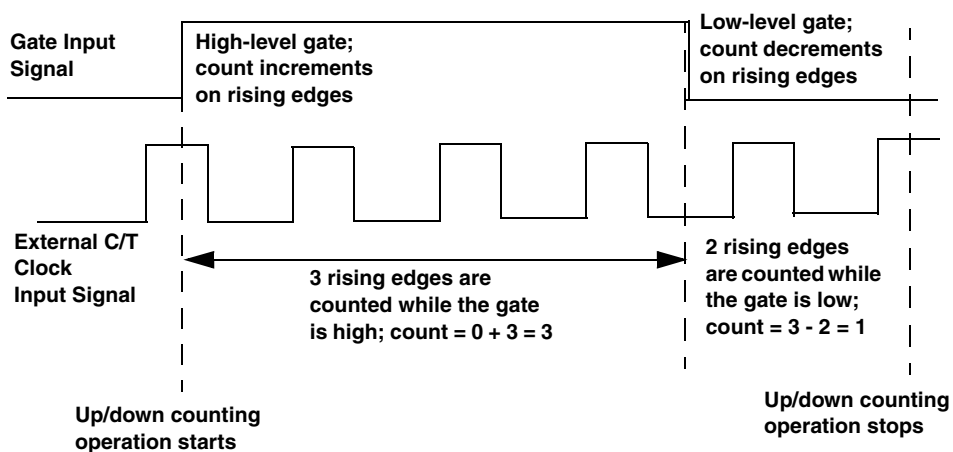


Figure 11: Example of Up/Down Counting

Frequency Measurement

You can also use event counting mode to measure the frequency of the clock input signal for the counter, since frequency is the number of events divided by a specified duration.

To determine if the subsystem supports event counting (and therefore, frequency measurement), use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_CTMODE_COUNT`. If this function returns a nonzero value, the capability is supported.

You can perform a frequency measurement operation in one of two ways: using the Windows timer to specify the duration or using a pulse of a known duration as the gate input signal to a counter/timer configured for event counting mode. The following subsections describe these ways of measuring frequency.

Using the Windows Timer

To perform a frequency measurement operation on a single C/T subsystem using the Windows timer to specify the duration, perform the following steps:

1. Use the **olDaSetCTMode** function, specifying the `OL_CTMODE_COUNT` parameter.
2. Specify the input clock source using **olDaSetClockSource**. In frequency measurement mode, an external C/T clock source is more useful than the internal C/T clock source; refer to [page 101](#) for more information on the external C/T clock source.
3. Use the **olDaSetGateType** function, specifying the `OL_GATE_NONE` parameter, to set the gate type to software.
4. Use the **olDaMeasureFrequency** function to specify the duration of the Windows timer (which has a resolution of 1 ms) and to start the frequency measurement operation.

Frequency is determined using the following equation:

$$\text{Frequency} = \frac{\text{Number of Events}}{\text{Duration of the Windows Timer}}$$

When the operation is complete, the OLDA_WM_MEASURE_DONE message is generated. Use the LongtoFreq (IParam) macro, described in the DataAcq SDK online help, to return the measured frequency value.

Figure 12 shows an example of a frequency measurement operation. Three events are counted from the clock input signals during a duration of 300 ms. The frequency is 10 Hz (3/.3).

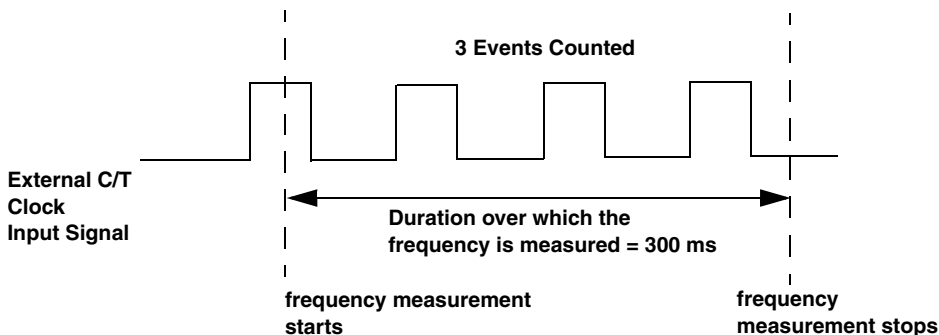


Figure 12: Example of Frequency Measurement

Using a Pulse of a Known Duration

If you need more accuracy than the Windows timer provides, you can connect a pulse of a known duration to the external gate input of a counter/timer configured for event counting; refer to the devices' user manuals for wiring details.

The following example describes how to use the DataAcq SDK to measure frequency using two C/T subsystems: one that generates a variable-width one-shot pulse as the gate input to a second C/T subsystem configured for event counting mode:

1. Set up one C/T subsystem for one-shot mode as follows:
 - a. Use the **olDaSetCTMode** function, specifying the **OL_CTMODE_ONESHOT** parameter.
 - b. For this C/T subsystem, specify the clock source (with **olDaSetClockSource**), the clock frequency (with **olDaSetClockFrequency** if using an internal clock source, or **olDaSetExternalClockDivider** if using an external clock source), the gate type (with **olDaSetGateType**), the type of output pulse (with **olDaSetPulseType**), and the pulse width (with **olDaSetPulseWidth**). The pulse width and period are used to determine the time that the gate is active.
 - c. Configure this C/T subsystem with **olDaConfig**.
 - d. Get the actual clock frequency used by this C/T subsystem with **olDaGetClockFrequency** or **olDaGetExternalClockDivider**. You will use this value in the measurement period calculation.
 - e. Get the actual pulse width used by this C/T subsystem with **olDaGetPulseWidth**. You will use this value in the measurement period calculation.
2. Set up another C/T subsystem for event counting mode:
 - a. Use the **olDaSetCTMode** function, specifying the **OL_CTMODE_COUNT** parameter, to set up this C/T subsystem for event counting mode (and, therefore, a frequency measurement operation).
 - b. For this C/T subsystem, use **olDaSetClockSource** to specify the clock source you want to measure. For frequency measurement operations, an external C/T clock source is more useful than the internal C/T clock source; refer to [page 101](#) for more information on the external C/T clock source.

- c. For this C/T subsystem, use the **olDaSetGateType** function to specify the gate type; ensure that the gate type for this C/T subsystem matches the active period of the output pulse from the C/T subsystem configured for one-shot mode.
 - d. Configure this C/T subsystem with **olDaConfig**.
3. Start the counter/timer configured for event counting mode with **olDaStart**.
 4. Start the counter/timer configured for one-shot mode with **olDaStart**.
 5. Allow a delay approximately equal to the measurement period to allow the one-shot to finish; events are counted only during the active period of the one-shot pulse.
 6. For the event-counting C/T subsystem, read the number of events counted with **olDaReadEvents**.
 7. Determine the measurement period using the following equation:

$$\text{Measurement Period} = \frac{1}{\text{Actual Clock Frequency}} * \text{Active Pulse Width of One-Shot C/T}$$

8. Determine the frequency of the clock input signal using the following equation:

$$\text{Frequency Measurement} = \frac{\text{Number of Events}}{\text{Measurement Period}}$$

Edge-to-Edge Measurement

Use edge-to-edge measurement to measure the time interval between a specified start edge and a specified stop edge. The start edge and the stop edge can occur on the rising edge of the counter's associated gate input, the falling edge of the counter's associated gate input, the rising edge of the counter's associated clock input, or the falling edge of the counter's associated clock input. When the start edge is detected, the counter starts incrementing, and continues incrementing until the stop edge is detected.

To determine if the subsystem supports edge-to-edge measurement, use the `olDaGetSSCaps` function, specifying the capability `OLSSC_SUP_CTMODE_MEASURE`. This function returns a bit value indicating how edge-to-edge measurement mode is supported for the specified device. For example, if edge-to-edge measurements are supported on the gate signal only (using both rising and falling edges), a bit value of 3 is returned. [Table 5](#) lists the possible bit values.

Table 5: Values for OLSSC_SUP_CTMODE_MEASURE

Value	Name	Description
0x00	–	Edge-to-edge measurements are not supported.
0x01	SUP_GATE_RISING_BIT	Supports edge-to-edge measurements based on the rising edge of the gate signal.
0x02	SUP_GATE_FALLING_BIT	Supports edge-to-edge measurements based on the falling edge of the gate signal.
0x04	SUP_CLOCK_RISING_BIT	Supports edge-to-edge measurements based on the rising edge of the clock signal.
0x08	SUP_CLOCK_FALLING_BIT	Supports edge-to-edge measurements based on the falling edge of the clock signal.

To specify an edge-to-edge measurement operation, use the **olDaSetCTMode** function, specifying the `OL_CTMODE_MEASURE` parameter.

Specify the C/T clock source for the operation as internal.

Start an edge-to-edge measurement operation using the **olDaStart** function. To read the current counter value, use the **olDaReadEvents** function.

To stop the event counting operation, call **olDaStop**, **olDaAbort**, or **olDaReset**; **olDaReset** function stops the operation and reinitializes the subsystem after stopping it.

Some subsystems also allow you to pause the operation using the **olDaPause** function and then resume the paused operation using the **olDaContinue** function. To determine if pausing is supported, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_PAUSE` capability. If this function returns a nonzero value, the capability is supported.

[Figure 13](#) shows an example of an edge-to-edge measurement operation. The start edge is a rising edge on the gate signal; the stop edge is a falling edge on the gate signal.

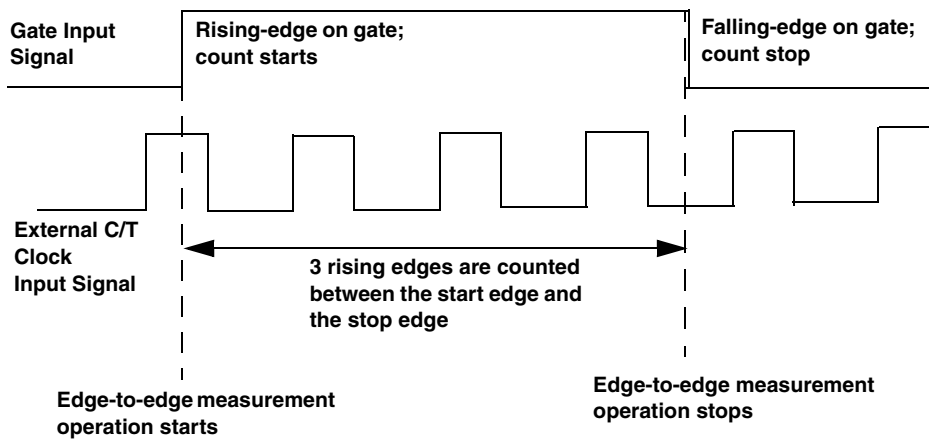


Figure 13: Example of Edge-to-Edge Measurement

You can use edge-to-edge measurement to measure the following:

- Pulse width of a signal pulse (the amount of time that a signal pulse is in a high or a low state, or the amount of time between a rising edge and a falling edge or between a falling edge and a rising edge). You can calculate the pulse width as follows:
 - Pulse width = Number of counts/18 MHz
- Period of a signal pulse (the time between two occurrences of the same edge - rising edge to rising edge or falling edge to falling edge). You can calculate the period as follows:
 - Period = 1/Frequency
 - Period = Number of counts/18 MHz
- Frequency of a signal pulse (the number of periods per second). You can calculate the frequency as follows:
 - Frequency = 18 MHz/Number of Counts

Rate Generation

Use rate generation mode to generate a continuous pulse output signal from the counter; this mode is sometimes referred to as continuous pulse output or pulse train output. You can use this pulse output signal as an external clock to pace analog input, analog output, or other counter/timer operations.

To determine if the subsystem supports rate generation, use the **oldaGetSSCaps** function, specifying the capability `OLSSC_SUP_CTMODE_RATE`. If this function returns a nonzero value, the capability is supported.

To specify a rate generation mode, use the **oldaSetCTMode** function, specifying the `OL_CTMODE_RATE` parameter.

Specify the C/T clock source for the operation. In rate generation mode, either the internal or external C/T clock input source is appropriate depending on your application; refer to [page 101](#) for information on specifying the C/T clock source.

Specify the frequency of the C/T clock output signal. For an internal C/T, the **oldaSetClockFrequency** function determines the frequency of the output pulse. For an external C/T clock source, the frequency of the clock input source divided by the clock divider (specified with the **oldaSetExternalClockDivider** function) determines the frequency of the output pulse.

Specify the polarity of the output pulses (high-to-low transitions or low-to-high transitions) and the duty cycle of the output pulses; refer to [page 108](#) for more information.

Also specify the gate type that enables the operation; refer to [page 104](#) for more information on specifying the gate type.

Start rate generation mode using the **olDaStart** function. While rate generation mode is enabled, the counter outputs a pulse of the specified type and frequency continuously. As soon as the operation is disabled, the pulse output operation stops.

To stop rate generation if it is in progress, call **olDaStop**, **olDaAbort**, or **olDaReset**; **olDaReset** stops the operation and reinitializes the subsystem after stopping it.

Some subsystems also allow you to pause the operation using the **olDaPause** function and resume the paused operation using the **olDaContinue** function. To determine if pausing is supported, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_PAUSE` capability. If this function returns a nonzero value, the capability is supported.

[Figure 14](#) shows an example of an enabled rate generation operation using an external C/T clock source with an input frequency of 4 kHz, a clock divider of 4, a low-to-high pulse type, and a duty cycle of 50%. (The gate type does not matter for this example.) A 1 kHz square wave is the generated output.

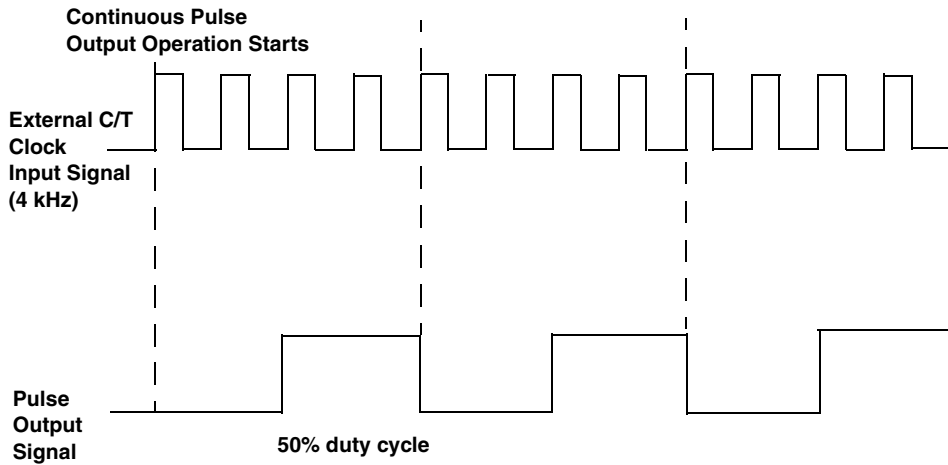


Figure 14: Example of Rate Generation Mode with a 50% Duty Cycle

Figure 15 shows the same example using a duty cycle of 75%.

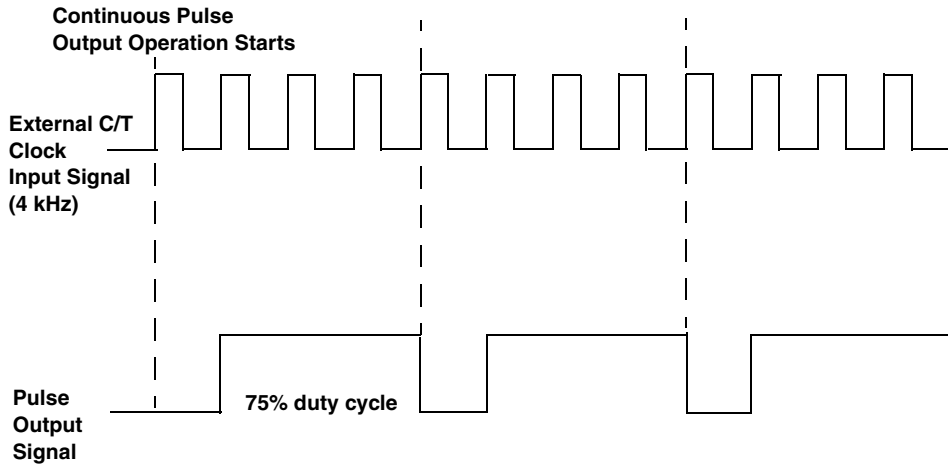


Figure 15: Example of Rate Generation Mode with a 75% Duty Cycle

Figure 16 shows the same example using a duty cycle of 25%.

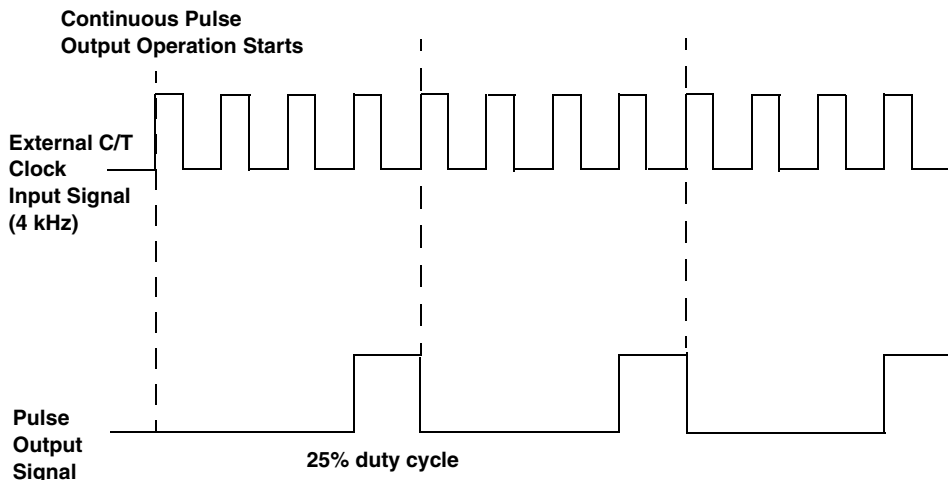


Figure 16: Example of Rate Generation Mode with a 25% Duty Cycle

One-Shot

Use one-shot mode to generate a single pulse output signal from the counter when the operation is triggered (determined by the gate input signal). You can use this pulse output signal as an external digital (TTL) trigger to start analog input, analog output, or other operations.

To determine if the subsystem supports one-shot mode, use the **oIDaGetSSCaps** function, specifying the capability `OLSSC_SUP_CTMODE_ONESHOT`. If this function returns a nonzero value, the capability is supported.

To specify a one-shot operation, use the **oIDaSetCTMode** function, specifying the `OL_CTMODE_ONESHOT` parameter.

Specify the C/T clock source for the operation. In one-shot mode, the internal C/T clock source is more useful than an external C/T clock source; refer to [page 101](#) for more information on specifying the C/T clock source.

Specify the polarity of the output pulse (high-to-low transition or low-to-high transition) and the duty cycle of the output pulse; refer to [page 108](#) for more information.

Note: In the case of a one-shot operation, use a duty cycle as close to 100% as possible to output a pulse immediately. Using a duty cycle less than 100% acts as a pulse output delay.

Also specify the gate type that triggers the operation; refer to [page 104](#) for more information.

To start a one-shot pulse output operation, use the **oldaStart** function. When the one-shot operation is triggered (determined by the gate input signal), a single pulse is output; then, the one-shot operation stops. All subsequent clock input signals and gate input signals are ignored.

Use software to specify the counter/timer mode as one-shot and wire the signals appropriately.

[Figure 17](#) shows an example of a one-shot operation using an external gate input (rising edge), a clock output frequency of 1 kHz (one pulse every 1 ms), a low-to-high pulse type, and a duty cycle of 99.99%.

[Figure 18](#) shows the same example using a duty cycle of less than or equal to 1%.

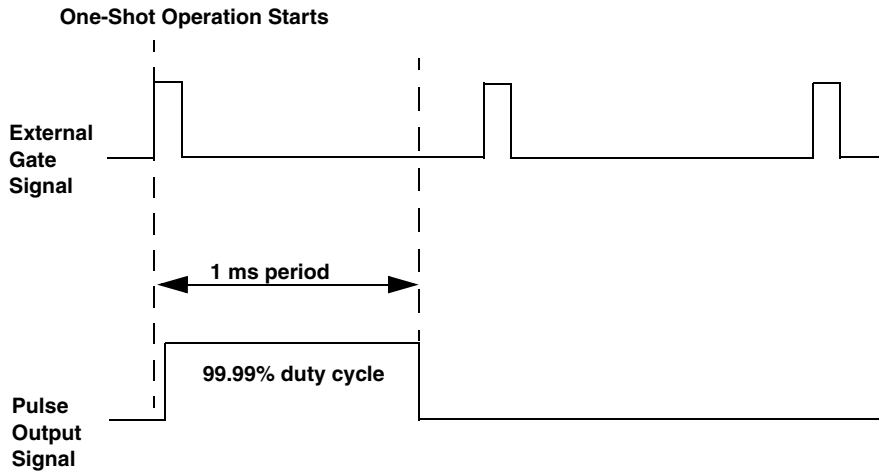


Figure 17: Example of One-Shot Mode Using a 99.99% Duty Cycle

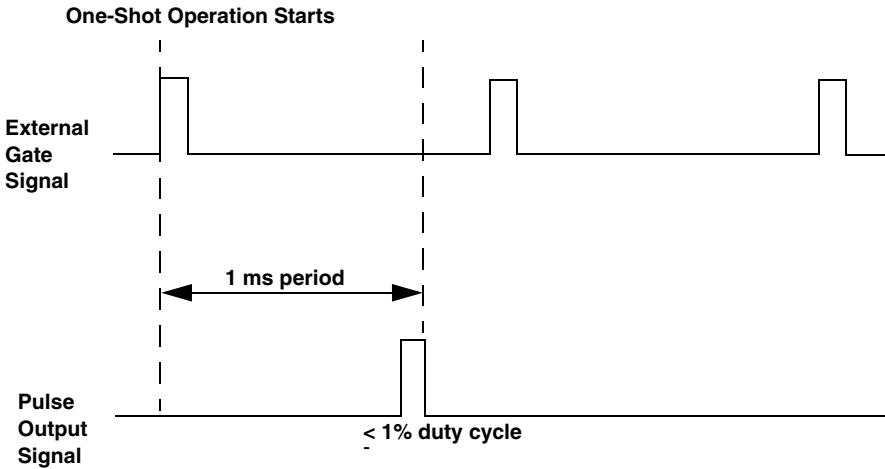


Figure 18: Example of One-Shot Mode Using a Duty Cycle Less Than or Equal to 1%

Repetitive One-Shot

Use repetitive one-shot mode to generate a pulse output signal each time the device detects a trigger (determined by the gate input signal). You can use this mode to clean up a poor clock input signal by changing its pulse width, then outputting it.

To determine if the subsystem supports repetitive one-shot mode, use the **olDaGetSSCaps** function, specifying the capability `OLSSC_SUP_CTMODE_ONESHOT_RPT`. If this function returns a nonzero value, the capability is supported.

To specify a repetitive one-shot operation, use the **olDaSetCTMode** function, specifying the `OL_CTMODE_ONESHOT_RPT` parameter.

Specify the C/T clock source for the operation. In repetitive one-shot mode, the internal C/T clock source is more useful than an external C/T clock source; refer to [page 101](#) for more information on specifying the C/T clock source.

Specify the polarity of the output pulses (high-to-low transitions or low-to-high transitions) and the duty cycle of the output pulses; refer to [page 108](#) for more information. Also specify the gate type that triggers the operation; refer to [page 104](#) for more information.

To start a repetitive one-shot pulse output operation, use the **olDaStart** function. When the one-shot operation is triggered (determined by the gate input signal), a pulse is output. When the device detects the next trigger, another pulse is output.

This operation continues until you stop the operation using **olDaStop**, **olDaAbort**, or **olDaReset**; **olDaReset** stops the operation and reinitializes the subsystem after stopping it.

Some subsystems also allow you to pause the operation using the **oIDaPause** function and resume the paused operation using the **oIDaContinue** function. To determine if pausing is supported, use the **oIDaGetSSCaps** function, specifying the **OLSSC_SUP_PAUSE** capability. If this function returns a nonzero value, the capability is supported.

Note: Triggers that occur while the pulse is being output are not detected by the device.

Figure 19 shows an example of a repetitive one-shot operation using an external gate (rising edge); a clock output frequency of 1 kHz (one pulse every 1 ms), a low-to-high pulse type, and a duty cycle of 99.99%.

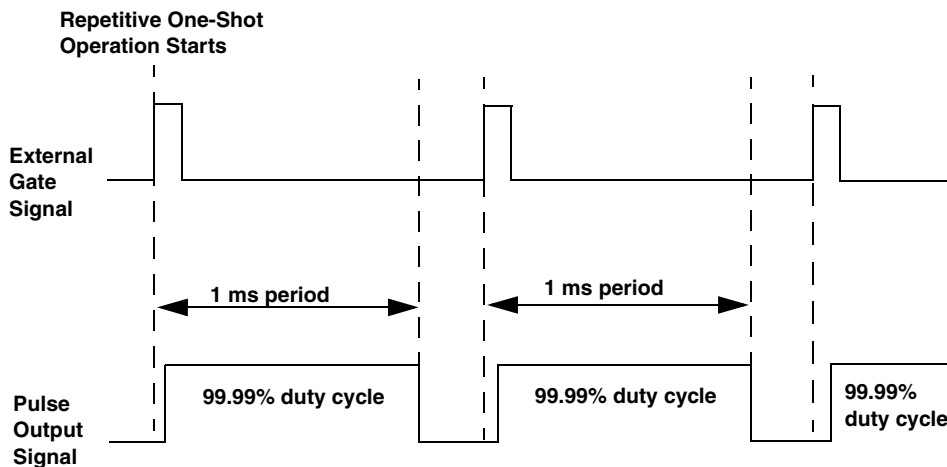


Figure 19: Example of Repetitive One-Shot Mode Using a 99.99% Duty Cycle

Figure 20 shows the same example using a duty cycle of 50%.

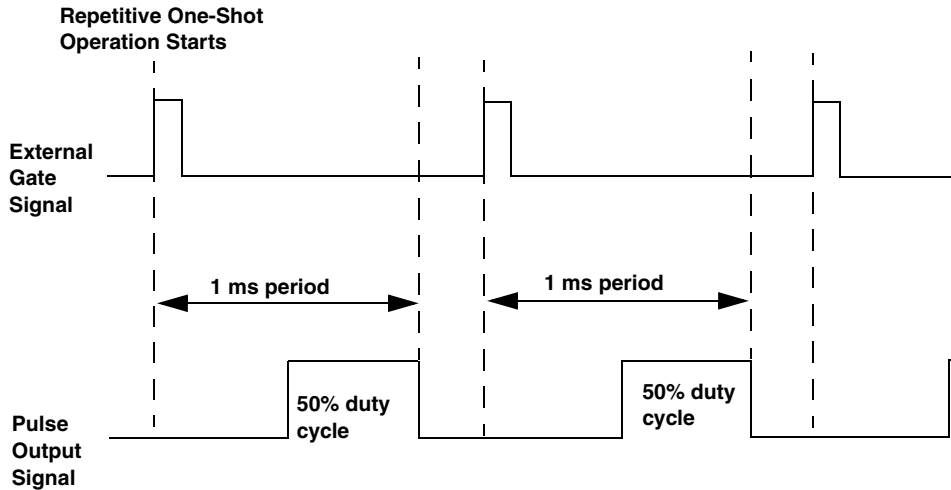


Figure 20: Example of Repetitive One-Shot Mode Using a 50% Duty Cycle

C/T Clock Sources

The DataAcq SDK defines the following clock sources for counter/timers:

- Internal C/T clock,
- External C/T clock,
- Internally cascaded clock, and
- Extra C/T clocks.

The following subsections describe these clock sources.

Internal C/T Clock

The internal C/T clock is the clock source on the device that paces a counter/timer operation for a C/T subsystem.

To determine if the subsystem supports an internal C/T clock, use the **olDaGetSSCaps** function, specifying the **OLSSC_SUP_INTCLOCK** capability. If this function returns a nonzero value, the capability is supported.

To specify the clock source, use the **olDaSetClockSource** function.

Using the **olDaSetClockFrequency** function, specify the frequency of the clock output signal.

To determine the maximum frequency that the subsystem supports, use the **olDaGetSSCapsEx** function, specifying the **OLSSCE_MAXTHROUGHPUT** capability. To determine the minimum frequency that the subsystem supports, use the **olDaGetSSCapsEx** function, specifying the **OLSSCE_MINTHROUGHPUT** capability.

External C/T Clock

The external C/T clock is a clock source attached to the device that paces counter/timer operations for a C/T subsystem. The external C/T clock is useful when you want to pace at rates not available with the internal clock or if you want to pace at uneven intervals.

To determine if the subsystem supports an external C/T clock, use the **olDaGetSSCaps** function, specifying the **OLSSC_SUP_EXTCLOCK** capability. If this function returns a nonzero value, the capability is supported.

Specify the clock source using the **olDaSetClockSource** function. Specify the clock divider using the **olDaSetExternalClockDivider** function; the clock input signal divided by the clock divider determines the frequency of the clock output signal.

To determine the maximum clock divider that the subsystem supports, use the **olDaGetSSCapsEx** function, specifying the `OLSSCE_MAXCLOCKDIVIDER` capability. To determine the minimum clock divider that the subsystem supports, use the **olDaGetSSCapsEx** function, specifying the `OLSSCE_MINCLOCKDIVIDER` capability

Internally Cascaded Clock

You can also internally connect or cascade the clock output signal from one counter/timer to the clock input signal of the next counter/timer in software. In this way, you can create a 32-bit counter out of two 16-bit counters.

To determine if the subsystem supports internal cascading, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_CASCADING` capability. If this function returns a nonzero value, the capability is supported.

Specify whether the subsystem is internally cascaded or not (single) using the **olDaSetCascadeMode** function.

Note: If a counter/timer is cascaded, you specify the clock input and gate input for the first counter in the cascaded pair. For example, if counters 1 and 2 are cascaded, specify the clock input and gate input for counter 1.

Extra C/T Clock Source

Extra C/T clock sources may be defined by your device driver.

To determine how many extra clock sources are supported by your subsystem, use the **olDaGetSSCaps** function, specifying the **OLSSC_NUMEXTRACLOCKS** capability. Refer to your device/driver documentation for a description of these clocks.

To specify internal or external extra clock sources and their frequencies and/or clock dividers, refer to the previous subsections.

Gate Types

The active edge or level of the gate input to the counter enables or triggers counter/timer operations. The DataAcq SDK defines the following gate input types:

- Software,
- High level,
- Low level,
- High edge,
- Low edge,
- Any level,
- High level debounced,
- Low level debounced,
- High edge debounced,
- Low edge debounced, and
- Any level debounced.

To specify the gate type, use the **olDaSetGateType** function. The following subsections describe these gate types.

Software Gate Type

A software gate type enables any specified counter/timer operation immediately when the **olDaSetGateType** function is executed.

To determine if the subsystem supports a software gate, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_GATE_NONE` capability. If this function returns a nonzero value, the capability is supported.

High-Level Gate Type

A high-level external gate type enables a counter/timer operation when the external gate signal is high, and disables a counter/timer operation when the external gate signal is low. Note that this gate type is used only for event counting, frequency measurement, and rate generation; refer to [page 82](#) for more information on these modes.

To determine if the subsystem supports a high-level external gate input, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_GATE_HIGH_LEVEL` capability. If this function returns a nonzero value, the capability is supported.

Low-Level Gate Type

A low-level external gate type enables a counter/timer operation when the external gate signal is low, and disables the counter/timer operation when the external gate signal is high. Note that this gate type is used only for event counting, frequency measurement, and rate generation; refer to [page 82](#) for more information on these modes.

To determine if the subsystem supports a low-level external gate input, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_GATE_LOW_LEVEL` capability. If this function returns a nonzero value, the capability is supported.

Low-Edge Gate Type

A low-edge external gate type triggers a counter/timer operation on the transition from the high edge to the low edge (falling edge). Note that this gate type is used only for one-shot and repetitive one-shot mode; refer to [page 99](#) for more information on these modes.

To determine if the subsystem supports a low-edge external gate input, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_GATE_LOW_EDGE` capability. If this function returns a nonzero value, the capability is supported.

High-Edge Gate Type

A high-edge external gate type triggers a counter/timer operation on the transition from the low edge to the high edge (rising edge). Note that this gate type is used only for one-shot and repetitive one-shot mode; refer to [page 82](#) for more information on these modes.

To determine if the subsystem supports a high-edge external gate input, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_GATE_HIGH_EDGE` capability. If this function returns a nonzero value, the capability is supported.

Any Level Gate Type

A level gate type enables a counter/timer operation on the transition from any level. Note that this gate type is used only for event counting, frequency measurement, and rate generation; refer to [page 82](#) for more information on these modes.

To determine if the subsystem supports a level external gate input, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_GATE_LEVEL` capability. If this function returns a nonzero value, the capability is supported.

High-Level, Debounced Gate Type

A high-level, debounced gate type enables a counter/timer operation when the external gate signal is high and debounced. Note that this gate type is used only for event counting, frequency measurement, and rate generation; refer to [page 82](#) for more information on these modes.

To determine if the subsystem supports a high-level debounced external gate input, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_GATE_HIGH_LEVEL_DEBOUNCE` capability. If this function returns a nonzero value, the capability is supported.

3

Low-Level, Debounced Gate Type

A low-level, debounced gate type enables a counter/timer operation when the external gate signal is low and debounced. Note that this gate type is used only for event counting, frequency measurement, and rate generation; refer to [page 82](#) for more information on these modes.

To determine if the subsystem supports a low-level debounced external gate input, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_GATE_LOW_LEVEL_DEBOUNCE` capability. If this function returns a nonzero value, the capability is supported.

High-Edge, Debounced Gate Type

A high-edge, debounced gate type triggers a counter/timer operation on the rising edge of the external gate signal; the signal is debounced. Note that this gate type is used only for one-shot and repetitive one-shot mode; refer to [page 82](#) for more information on these modes.

To determine if the subsystem supports a high-edge debounced external gate input, use the **olDaGetSSCaps** function, specifying the `OLSSC_SUP_GATE_HIGH_EDGE_DEBOUNCE` capability. If this function returns a nonzero value, the capability is supported.

Low-Edge, Debounced Gate Type

A low-edge, debounced gate type triggers a counter/timer operation on the falling edge of the external gate signal; the signal is debounced. Note that this gate type is used only for one-shot and repetitive one-shot mode; refer to [page 82](#) for more information on these modes.

To determine if the subsystem supports a low-edge debounced external gate input, use the `olDaGetSSCaps` function, specifying the `OLSSC_SUP_GATE_LOW_EDGE_DEBOUNCE` capability. If this function returns a nonzero value, the capability is supported.

Level, Debounced Gate Type

A level, debounced gate type enables a counter/timer operation on the transition of any level of the external gate signal; the signal is debounced. Note that this gate type is used only for event counting, frequency measurement, and rate generation; refer to [page 82](#) for more information on these modes.

To determine if the subsystem supports a high-edge debounced external gate input, use the `olDaGetSSCaps` function, specifying the `OLSSC_SUP_GATE_LEVEL_DEBOUNCE` capability. If this function returns a nonzero value, the capability is supported.

Pulse Output Types and Duty Cycles

The DataAcq SDK defines the following pulse output types:

- **High-to-low transitions** - The low portion of the total pulse output period is the active portion of the counter/timer clock output signal.

To determine if the subsystem supports high-to-low transitions on the pulse output signal, use the `olDaGetSSCaps` function, specifying the `OLSSC_SUP_PLS_HIGH2LOW` capability. If this function returns a nonzero value, the capability is supported.

- **Low-to-high transitions** - The high portion of the total pulse output period is the active portion of the counter/timer pulse output signal.

To determine if the subsystem supports low-to-high transitions on the pulse output signal, use the `olDaGetSSCaps` function, specifying the `OLSSC_SUP_PLS_LOW2HIGH` capability. If this function returns a nonzero value, the capability is supported.

Specify the pulse output type using the `olDaSetPulseType` function.

The duty cycle (or pulse width) indicates the percentage of the total pulse output period that is active. A duty cycle of 50, then, indicates that half of the total pulse is low and half of the total pulse output is high. Specify the pulse width using the `olDaSetPulseWidth` function.

[Figure 21](#) illustrates a low-to-high pulse with a duty cycle of approximately 30%.

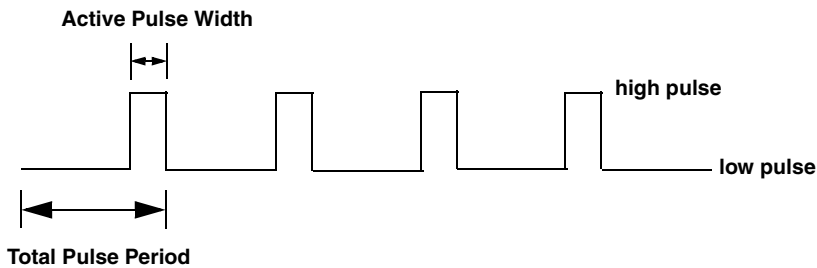


Figure 21: Example of a Low-to-High Pulse Output Type

Simultaneous Operations

If supported, you can synchronize subsystems to perform simultaneous operations. Note that you cannot perform simultaneous operations on subsystems configured for single-value operations.

To determine if the subsystems support simultaneous operations, use the **olDaGetSSCaps** function for each subsystem, specifying the `OLSSC_SUP_SIMULTANEOUS_START` capability. If this function returns a nonzero value, the capability is supported.

You can synchronize the triggers of subsystems by specifying the same trigger source for each of the subsystems that you want to start simultaneously and wiring them to the device, if appropriate.

Use the **olDaGetSSLList** function to allocate a simultaneous start list. Then, use the **olDaPutDassToSSLList** function to put the subsystems that you want to start simultaneously on the start list.

To determine the device handles given to each subsystem on the simultaneous start list, use the **olDaEnumSSLList** function.

Pre-start the subsystems using the **olDaSimultaneousPreStart** function. Pre-starting a subsystem ensures a minimal delay once the subsystems are started. Once you call the **olDaSimultaneousPreStart** function, do not alter the settings of the subsystems on the simultaneous start list.

Start the subsystems using the **olDaSimultaneousStart** function. When started, both subsystems are triggered simultaneously.

Note: Do not call **oIDaStart** when using simultaneous start lists, since the subsystems are already started.

When you are finished with the operations, call the **oIDaReleaseSSList** function to free the simultaneous start list. Then, call the **oIDaReleaseDASS** function for each subsystem to free it before calling **oIDaTerminate**.

To stop the simultaneous operations, call **oIDaStop** (for an orderly stop), **oIDaAbort** (for an abrupt stop) or **oIDaReset** (for an abrupt stop that reinitializes the subsystem).



Programming Flowcharts

Single-Value Operations	115
Continuous Buffered Input Operations	117
Continuous Buffered Output Operations.....	119
Event Counting Operations	121
Up/Down Counting Operations	123
Frequency Measurement Operations	125
Edge-to-Edge Measurement Operations.....	127
Pulse Output Operations.....	129
Simultaneous Operations	131

If you are unfamiliar with the capabilities of your device and/or subsystem, query the device as follows:

- To determine the number and types of DT-Open Layers devices and drivers installed, use the **olDaEnumBoards** function.
- To determine the subsystems supported by the device, use the **olDaEnumSubSystems** or **olDaGetDevCaps** function.
- To determine the capabilities of a subsystem, use the **olDaGetSSCaps** or **olDaGetSSCapsEx** function, specifying one of the capabilities listed in [Table 2 on page 13](#).
- To determine the gains, filters, ranges, and resolutions if more than one is available, use the **olDaEnumSSCaps** function.

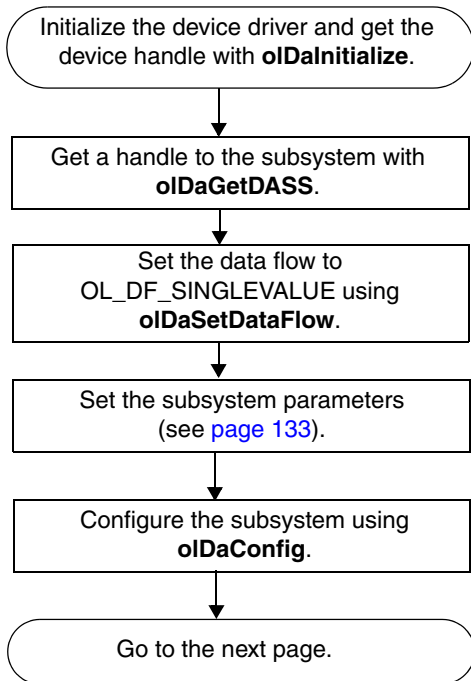
Then, follow the flowcharts presented in the remainder of this chapter to perform the desired operation.

Notes: Depending on your device, some of the settings may not be programmable. Refer to your device driver documentation for details.

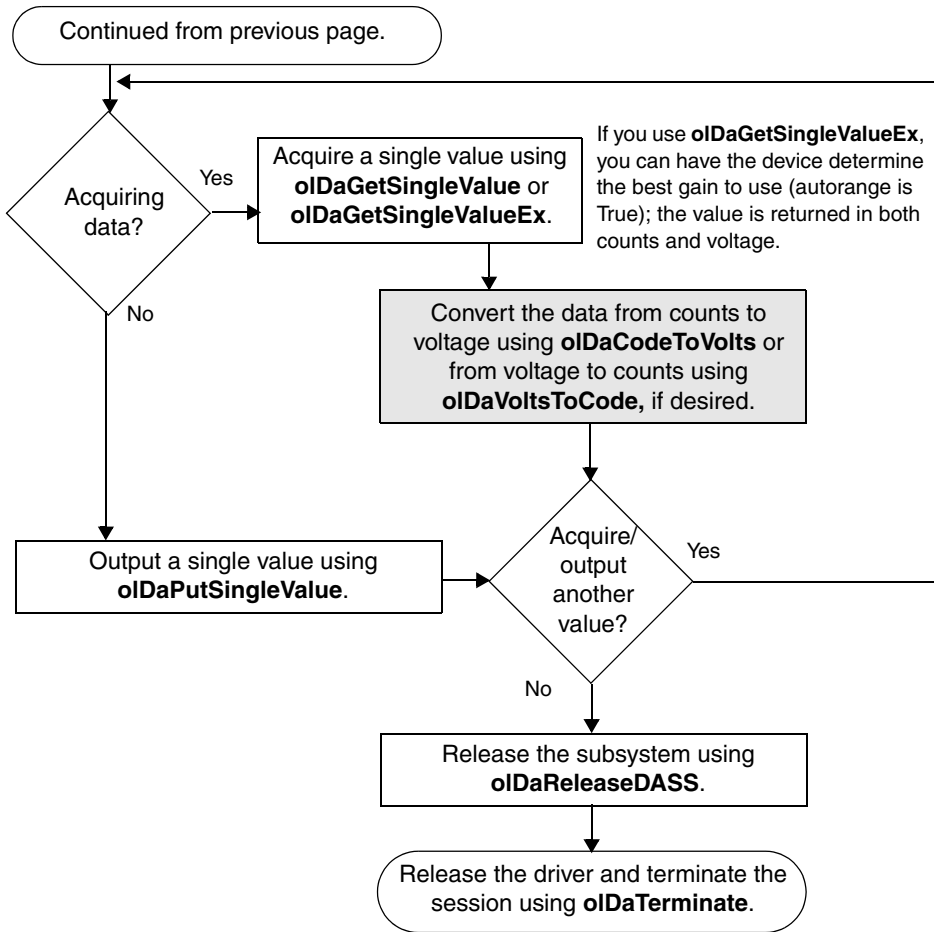
Although the flowcharts do not show error checking, it is recommended that you check for errors after each function call.

Some steps represent several substeps; if you are unfamiliar with the detailed operations involved with any one step, refer to the indicated page for detailed information. Optional steps appear in shaded boxes.

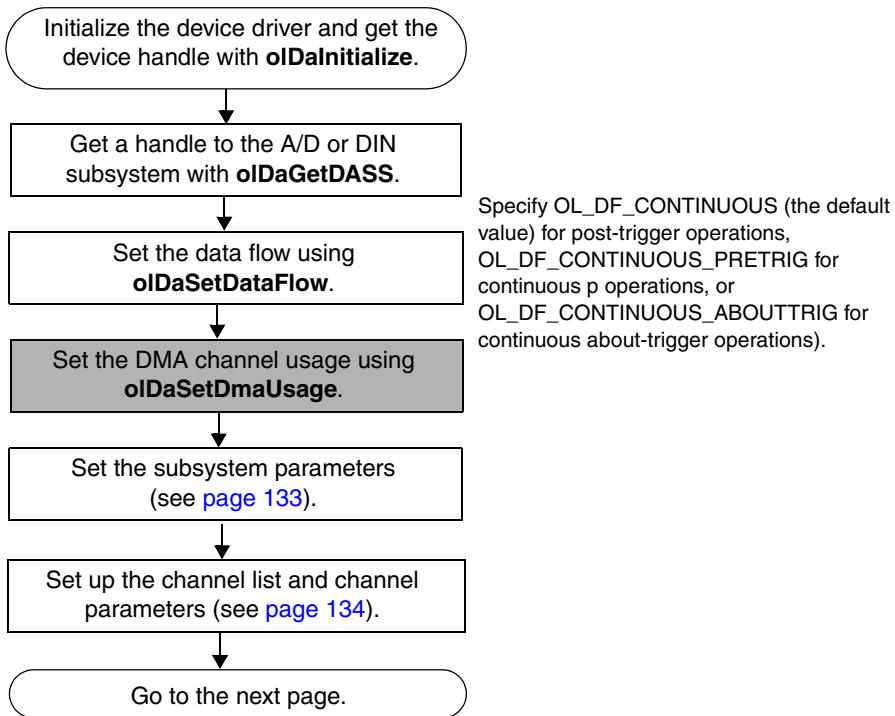
Single-Value Operations

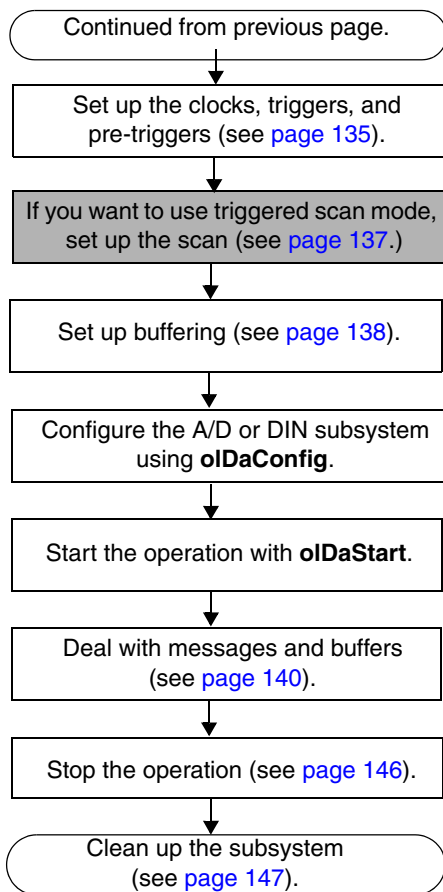


Specify A/D for an analog input subsystem, D/A for an analog output subsystem, DIN for a digital input subsystem, or DOUT for a digital output subsystem.



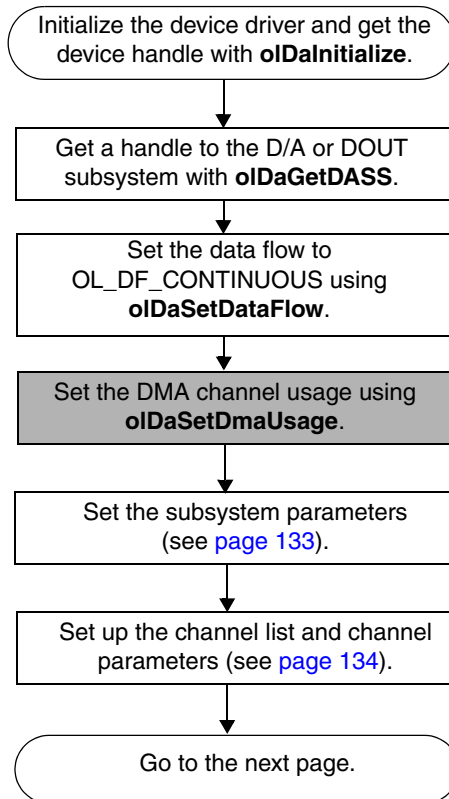
Continuous Buffered Input Operations

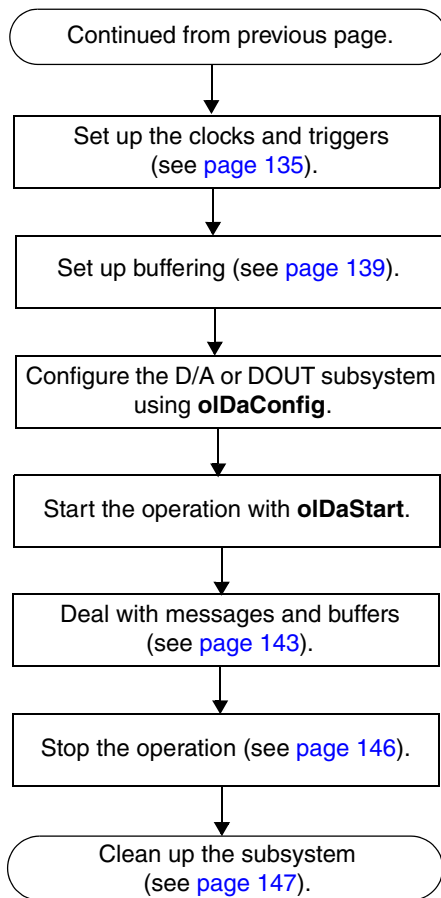




After configuration, if using an internal clock, you can use **oIDaGetClockFrequency** to get the actual frequency that the internal sample clock can achieve; if using an external clock, you can use **oIDaGetExternalClockDivider** to get the actual clock divider that the device can achieve; if using internal retrigger mode, you can use **oIDaGetRetriggerFrequency** to get the actual frequency that the internal retrigger clock can achieve.

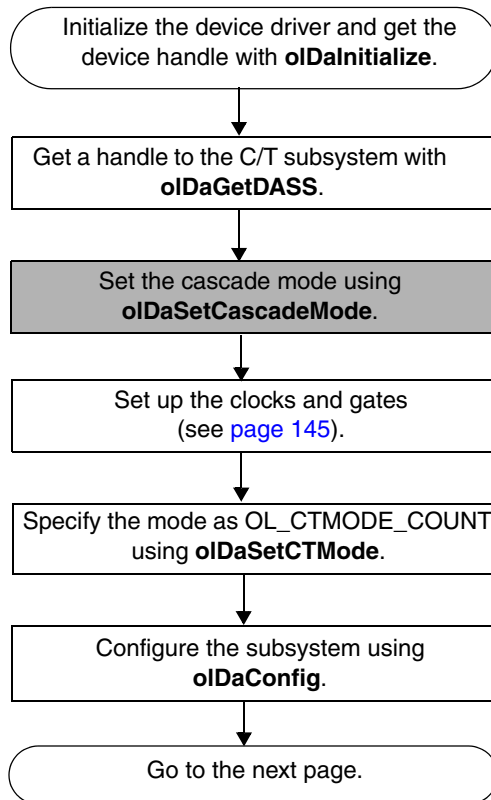
Continuous Buffered Output Operations

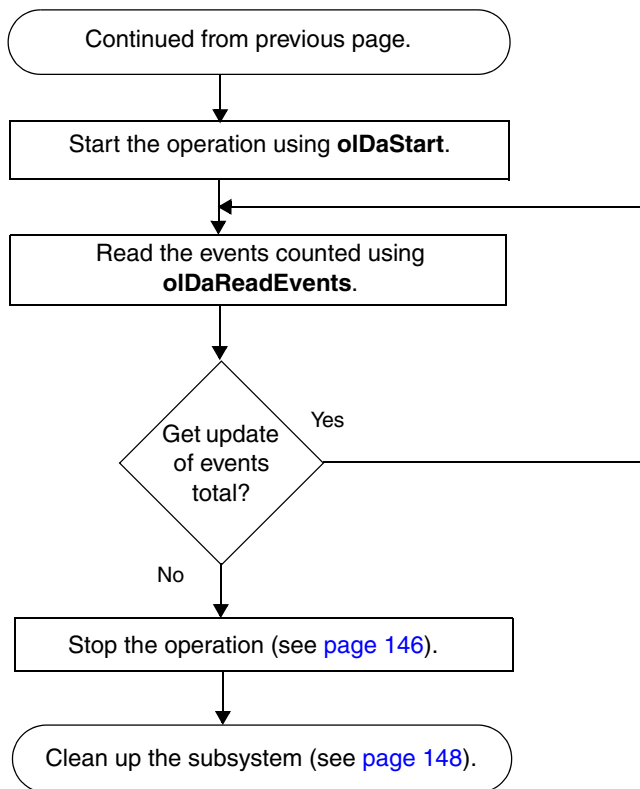




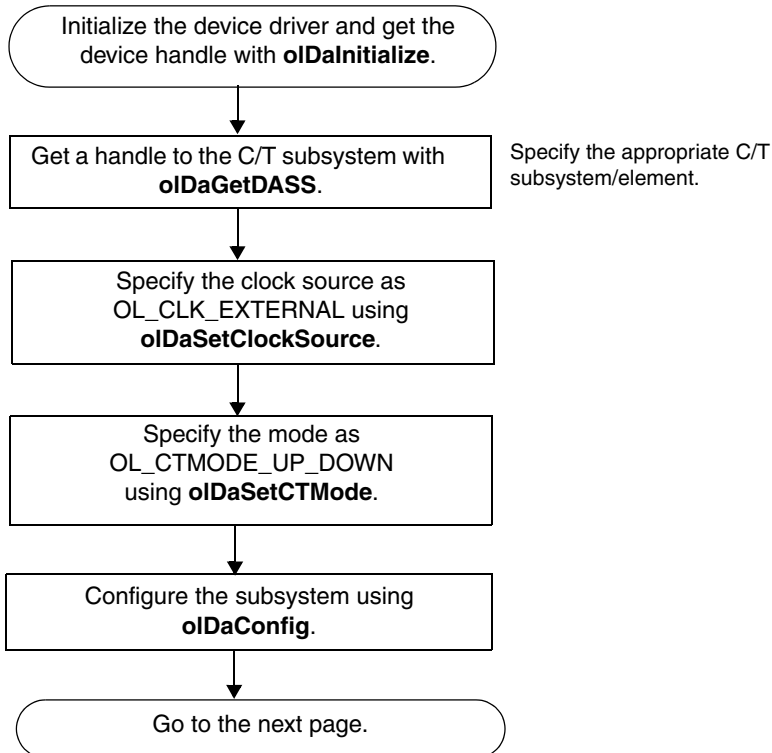
After configuration, if using an internal clock, you can use **oIDaGetClockFrequency** to get the actual frequency that the internal output clock can achieve; or if using an external clock, you can use **oIDaGetExternalClockDivider** to get the actual clock divider that the device can achieve.

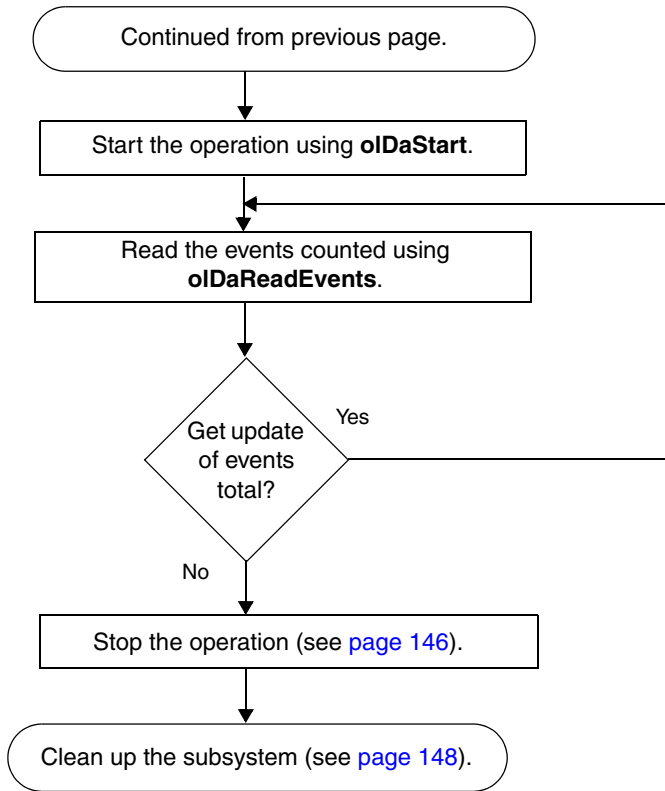
Event Counting Operations





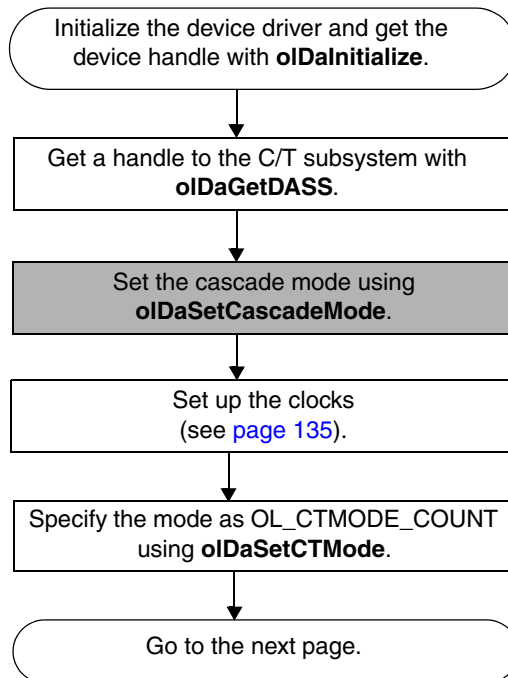
Up/Down Counting Operations

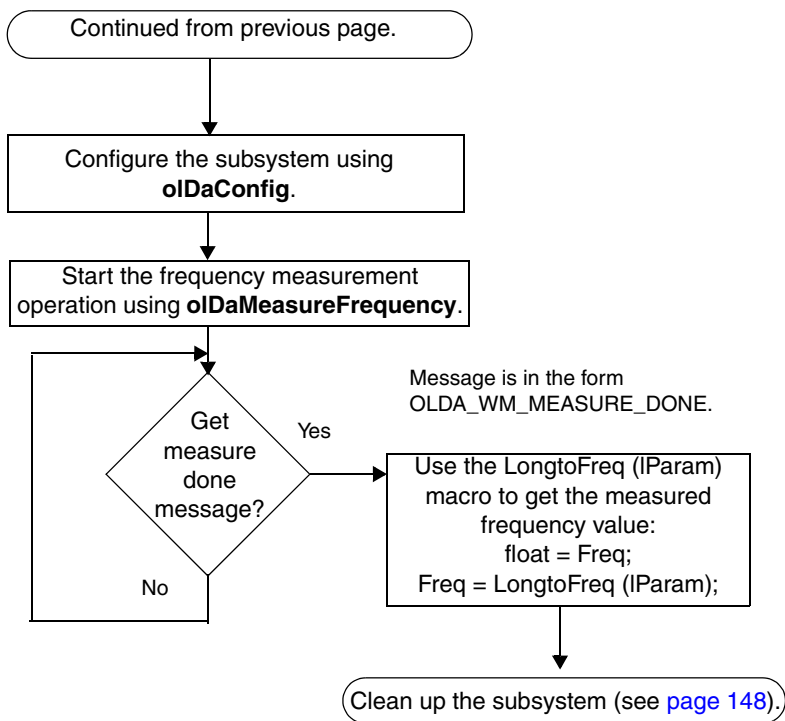




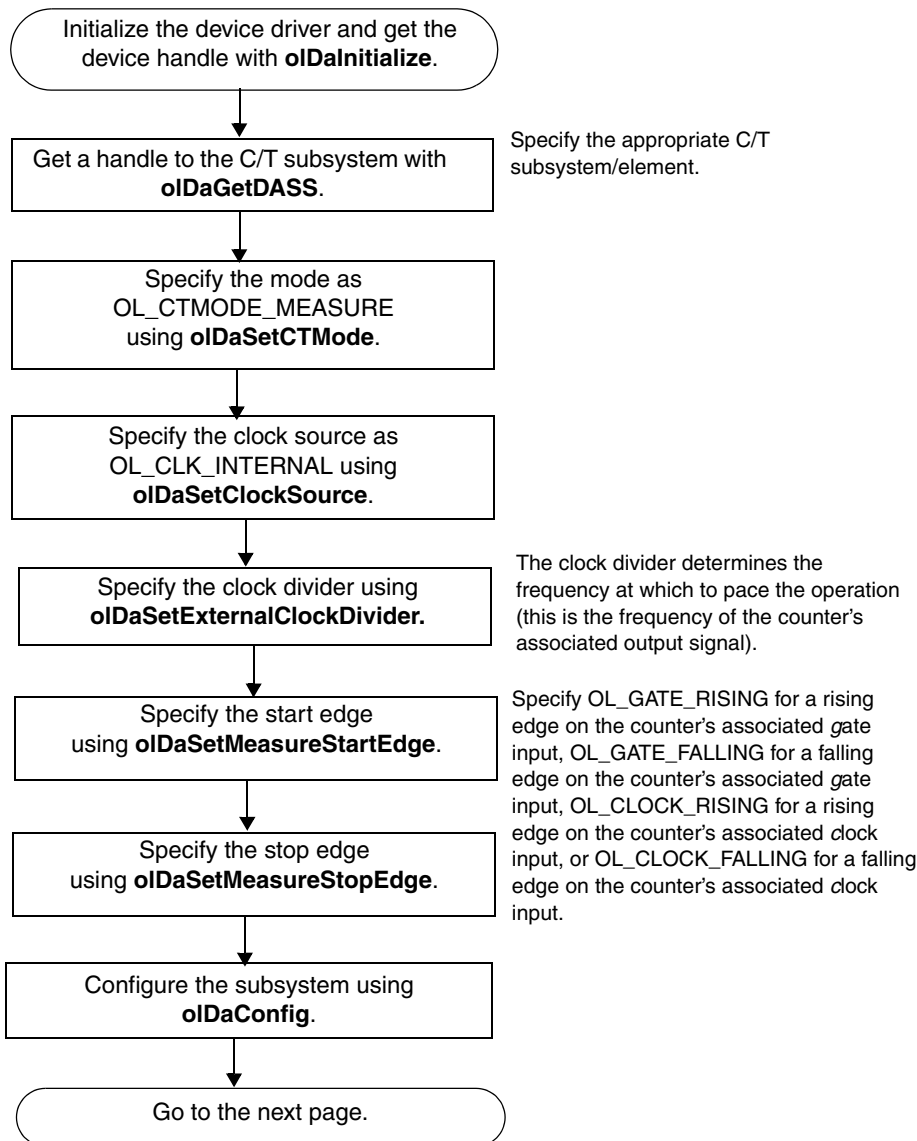
Frequency Measurement Operations

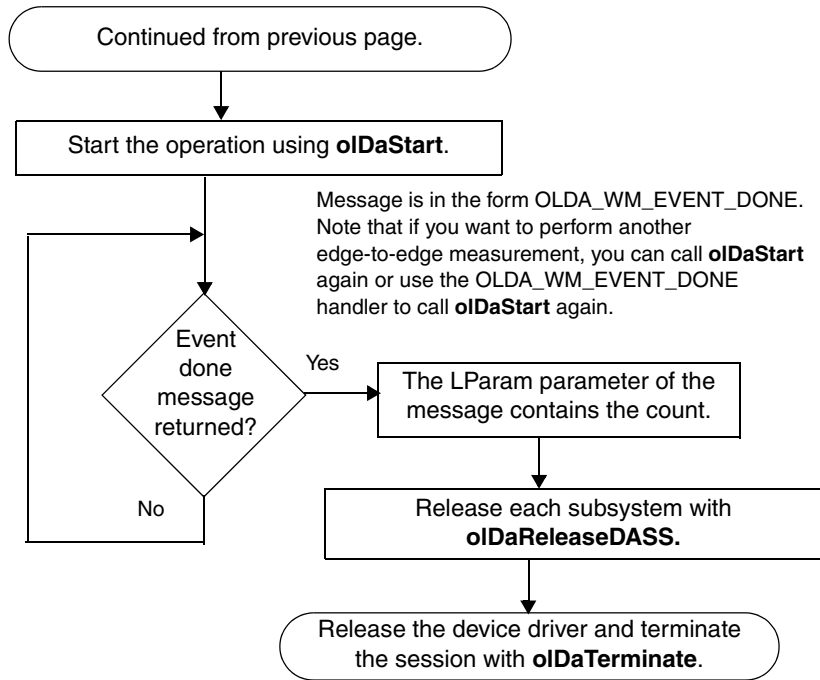
Note: If you need more accuracy than the system timer provides, refer to [page 86](#).



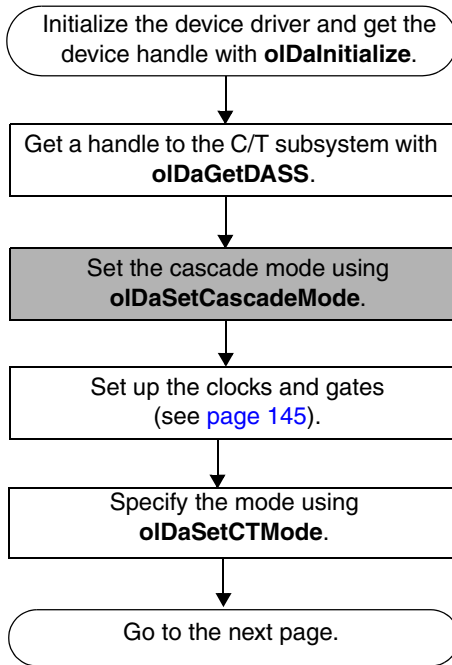


Edge-to-Edge Measurement Operations

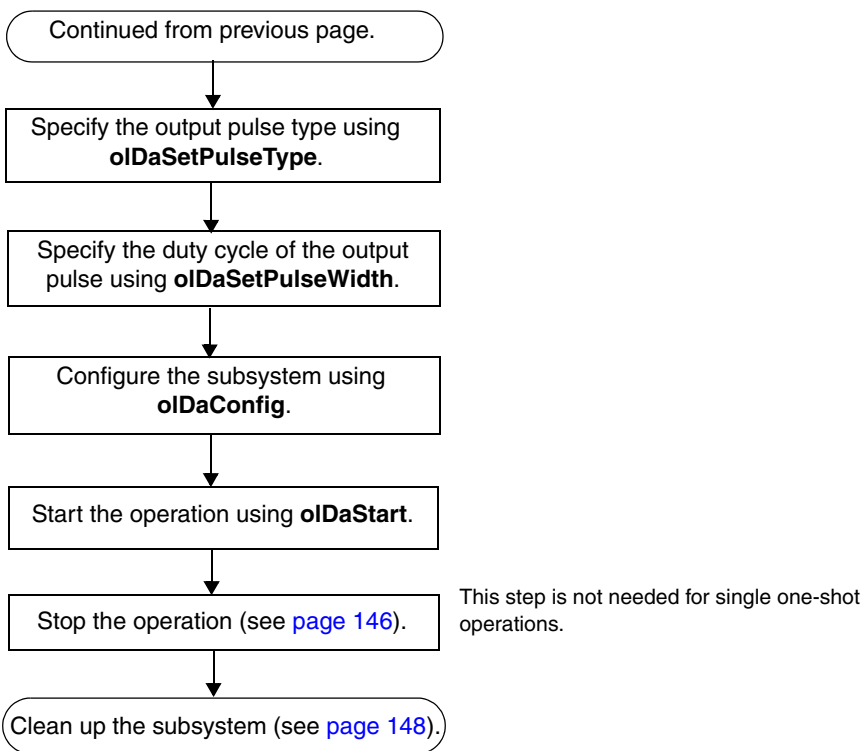




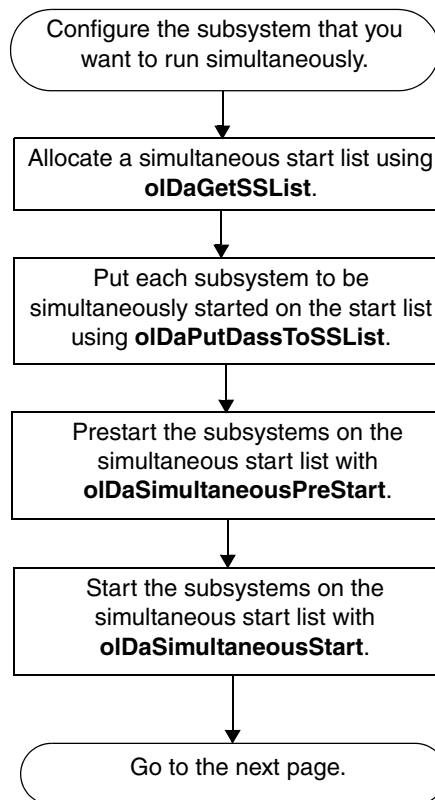
Pulse Output Operations



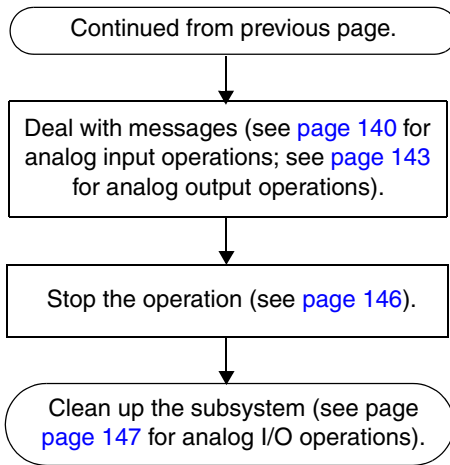
Specify `OL_CTMODE_RATE` for rate generation (continuous pulse output), `OL_CTMODE_ONESHOT` for single one-shot, or `OL_CTMODE_ONESHOT_RPT` for repetitive one-shot.



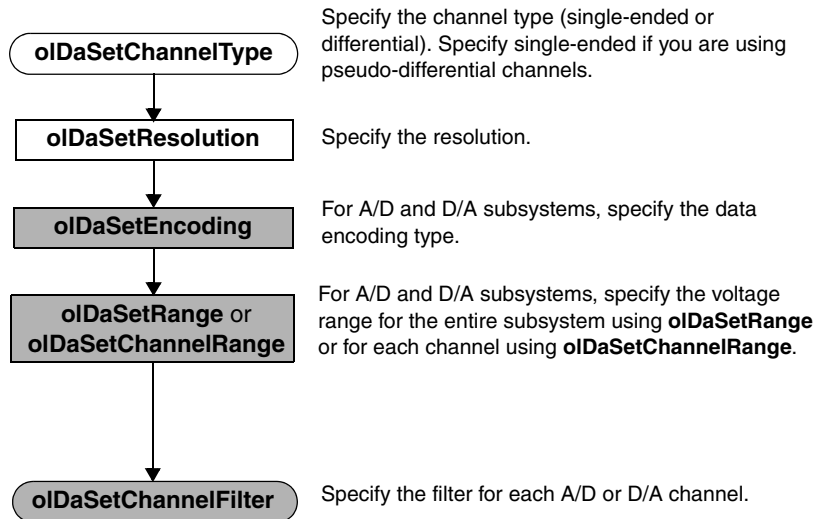
Simultaneous Operations



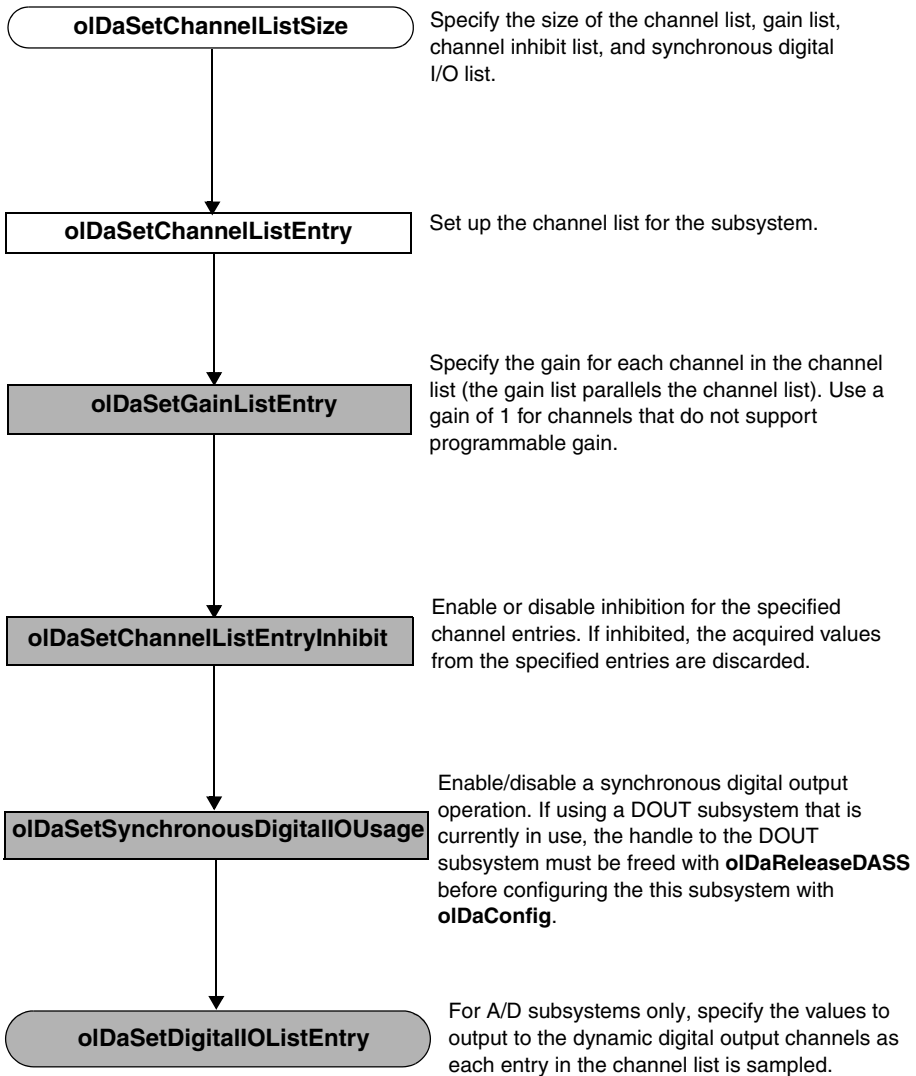
See the previous flow diagrams in this chapter; you cannot perform single-value operations simultaneously.



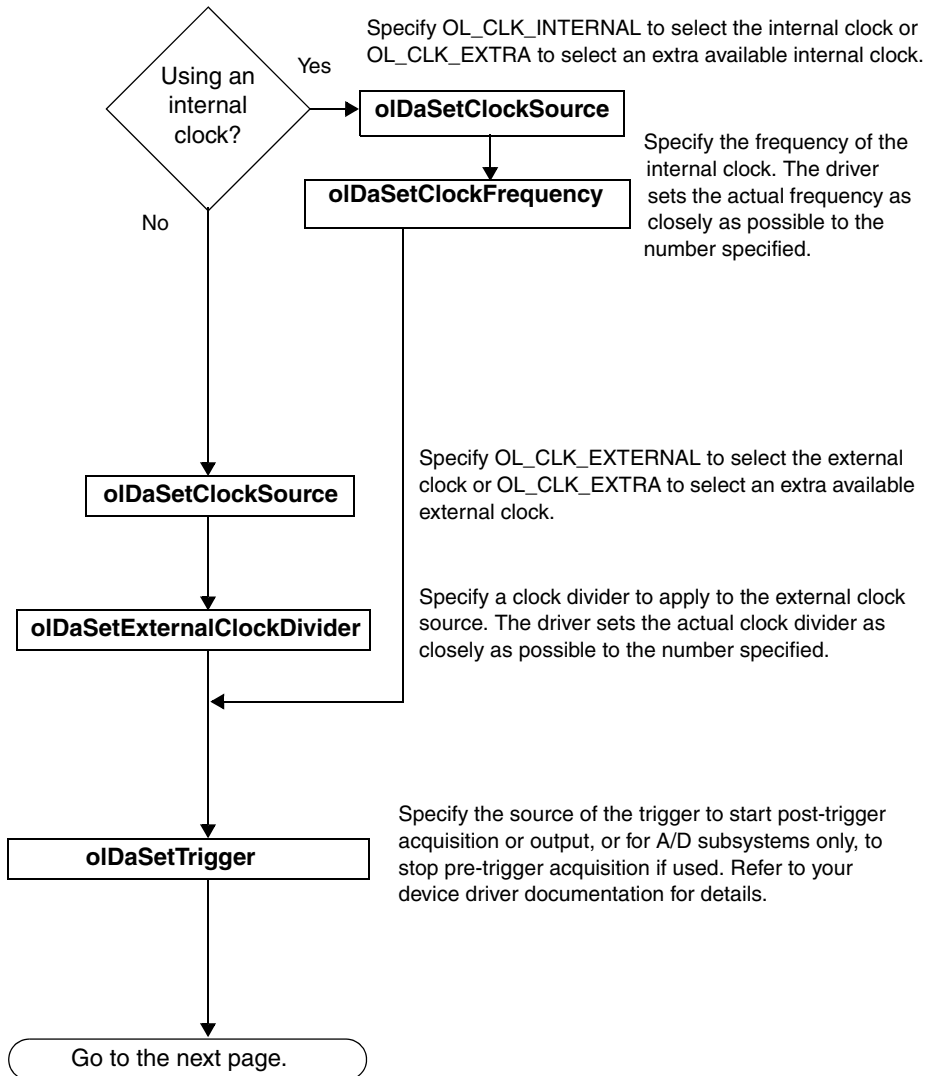
Set Subsystem Parameters

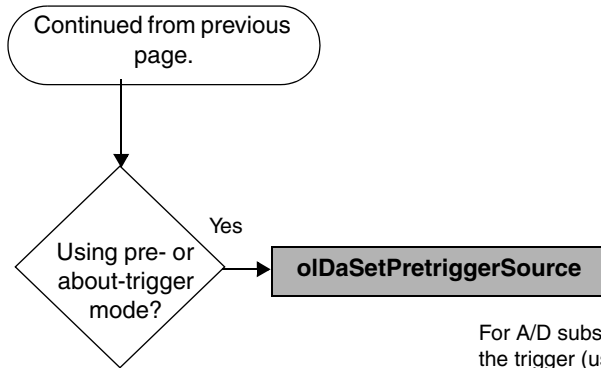


Set Up Channel List and Channel Parameters



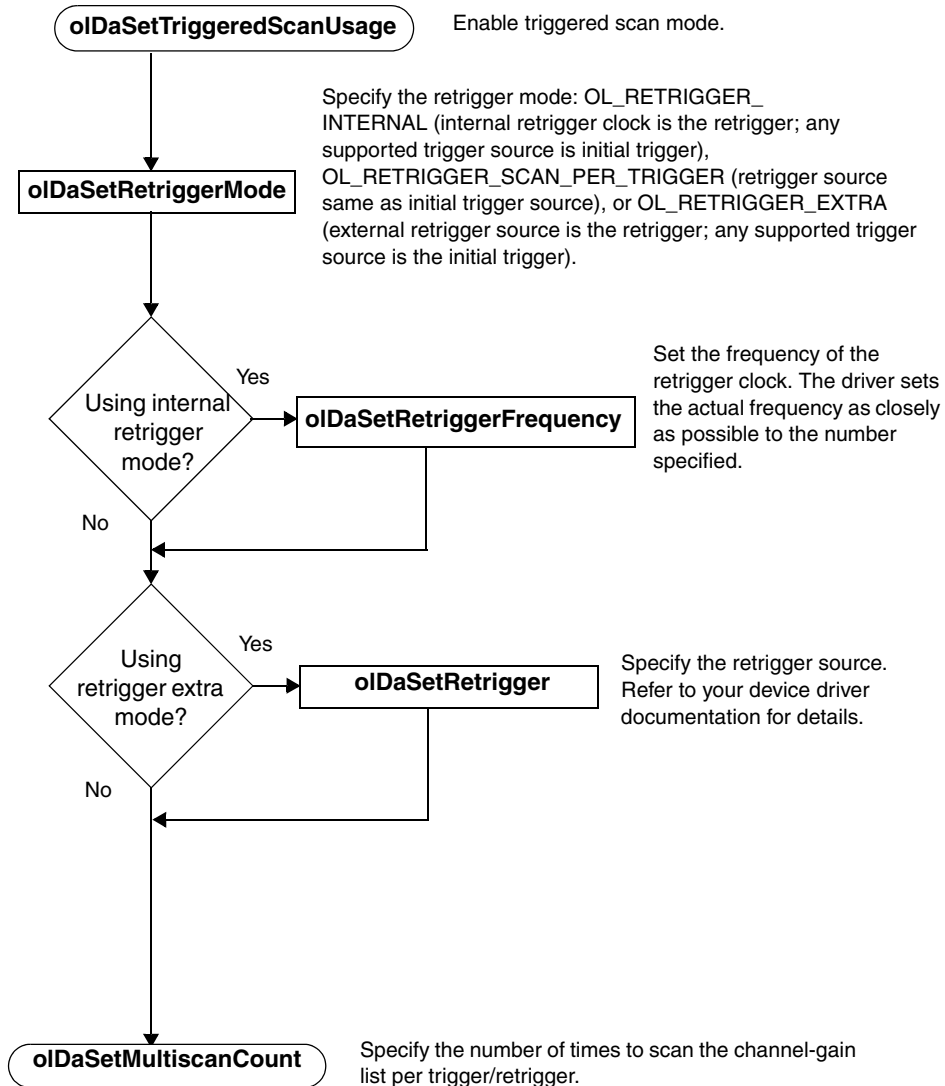
Set Clocks, Triggers, and Pre-Triggers



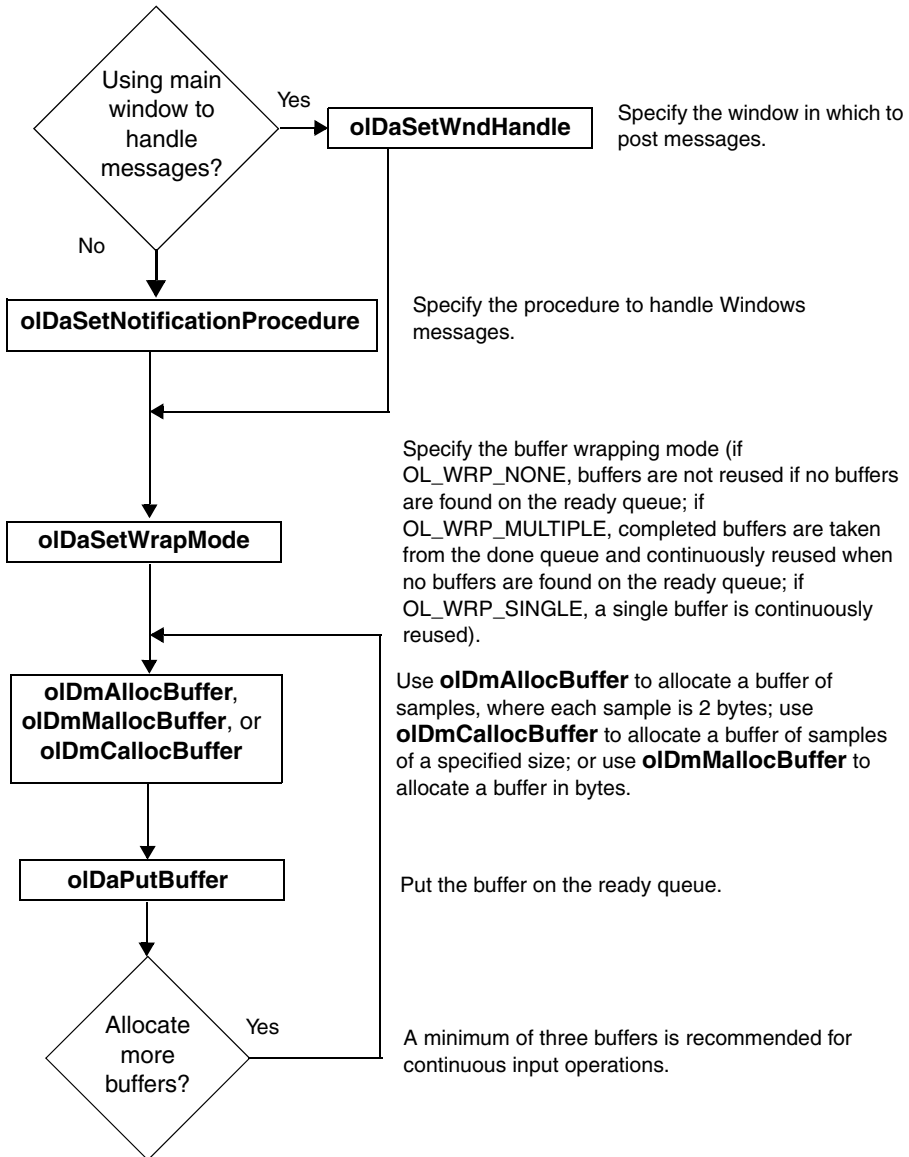


For A/D subsystems only, specify the trigger (usually software) to start the pre-trigger or about-trigger acquisition.

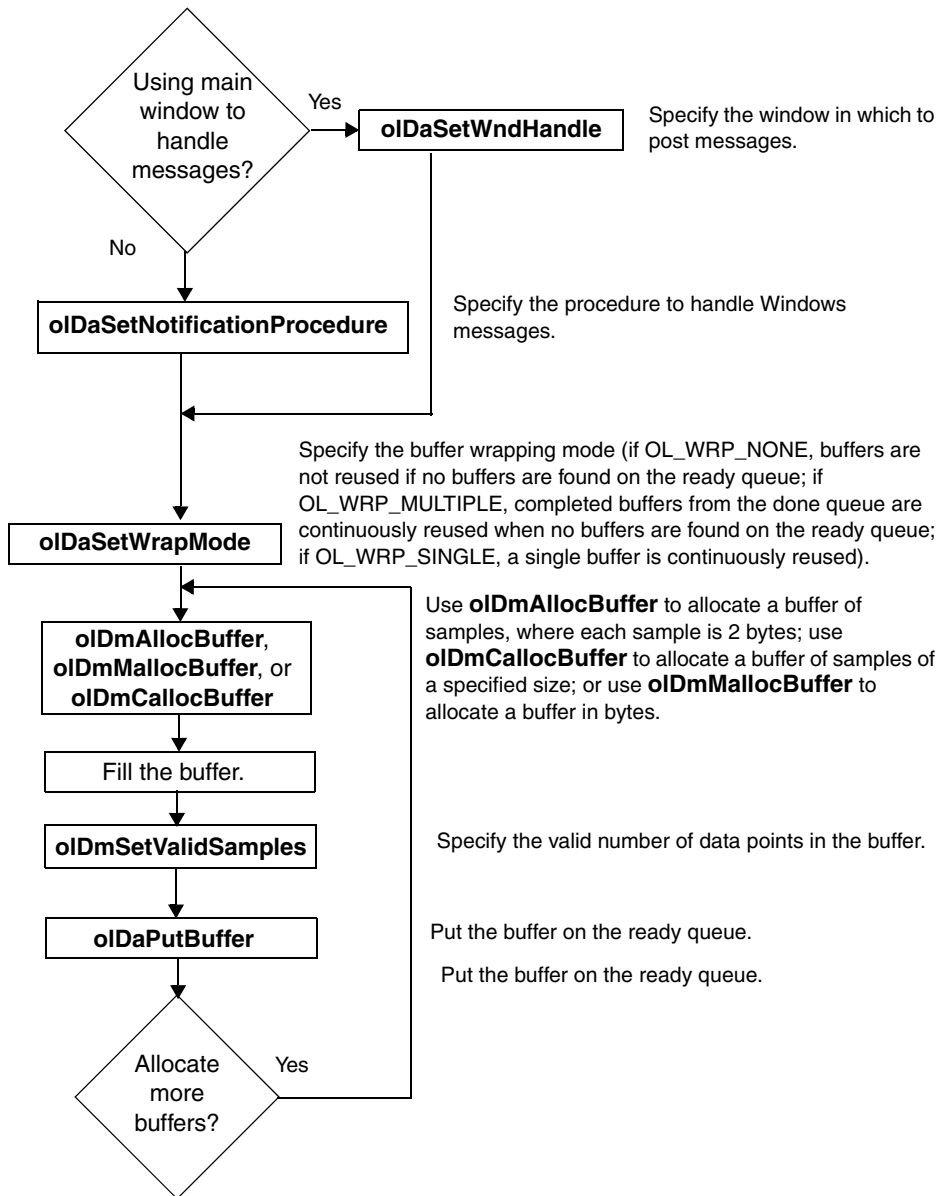
Set Up Triggered Scan



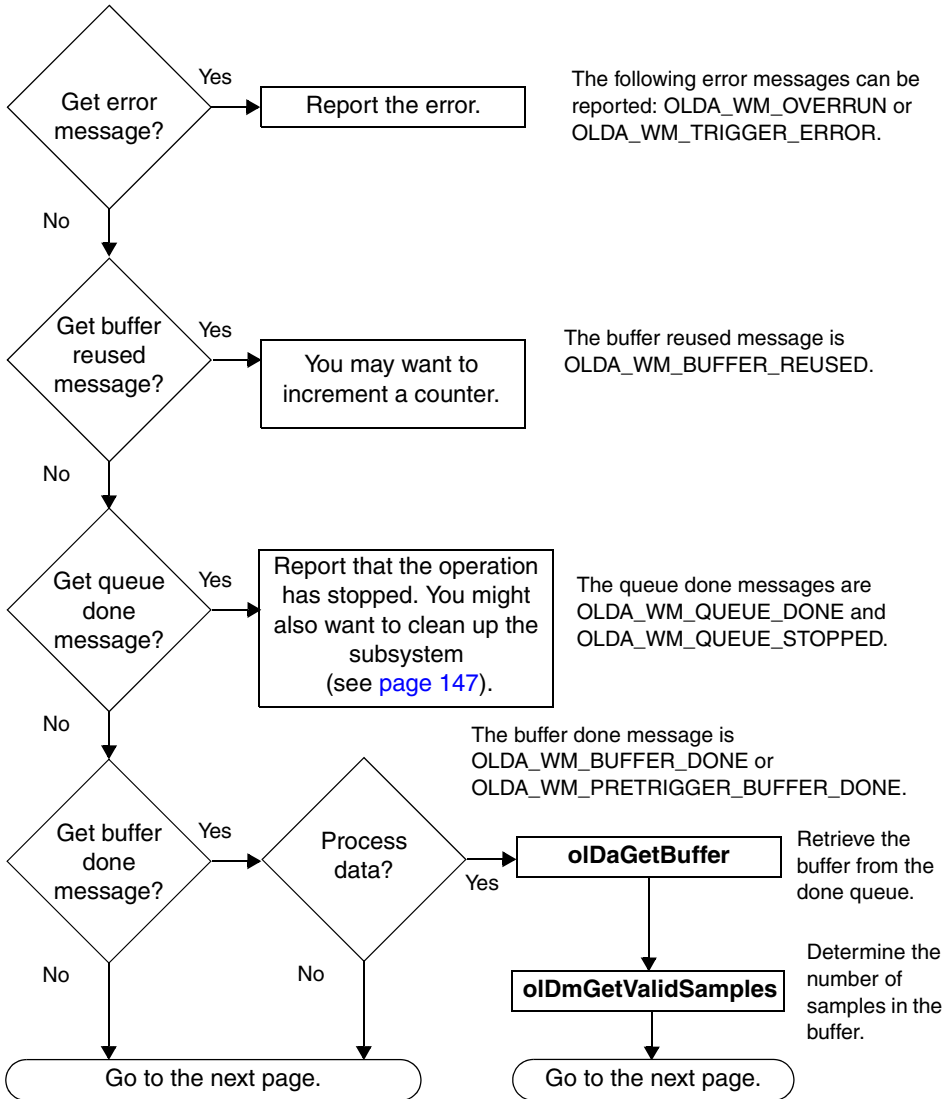
Set Up Input Buffering

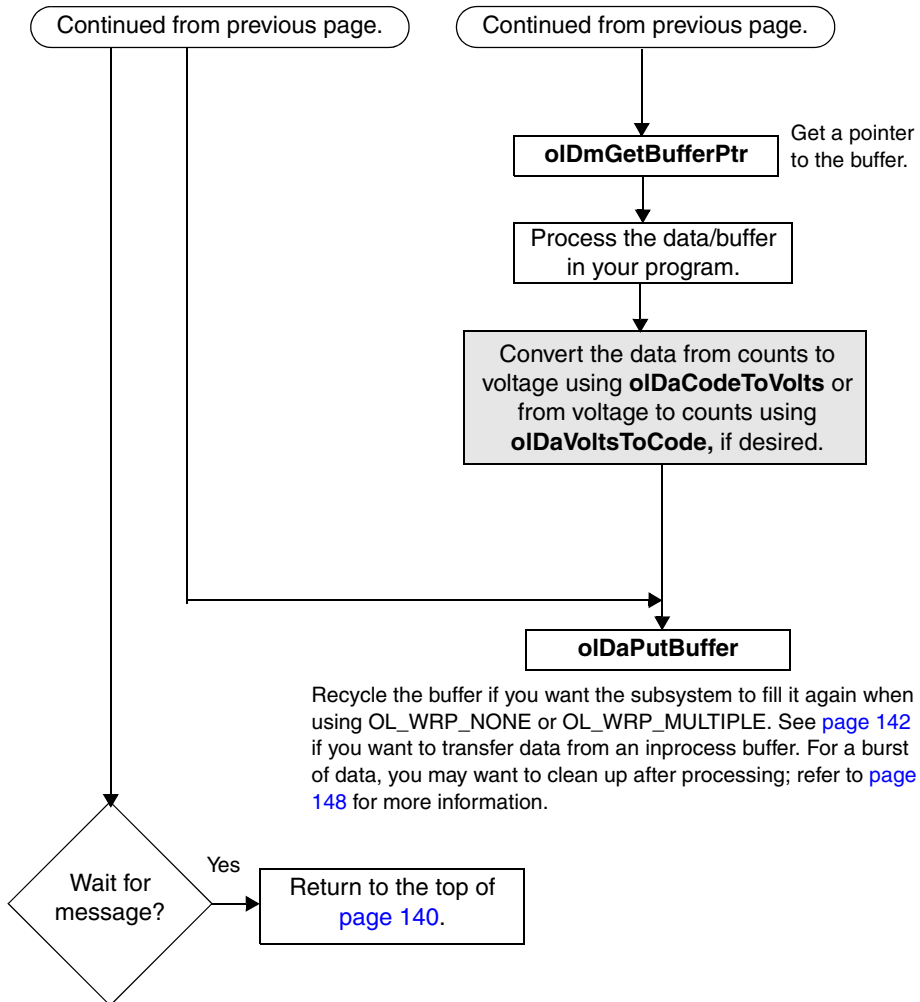


Set Up Output Buffering

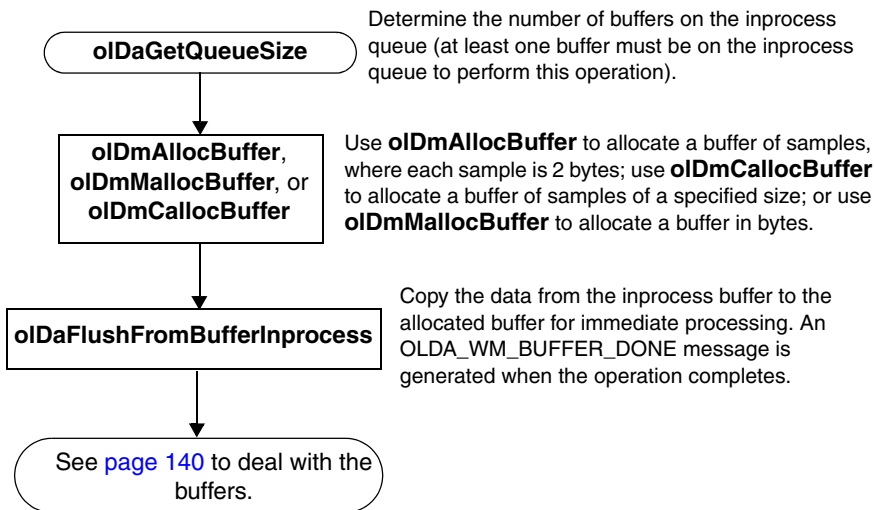


Deal with Messages and Buffers for Input Operations

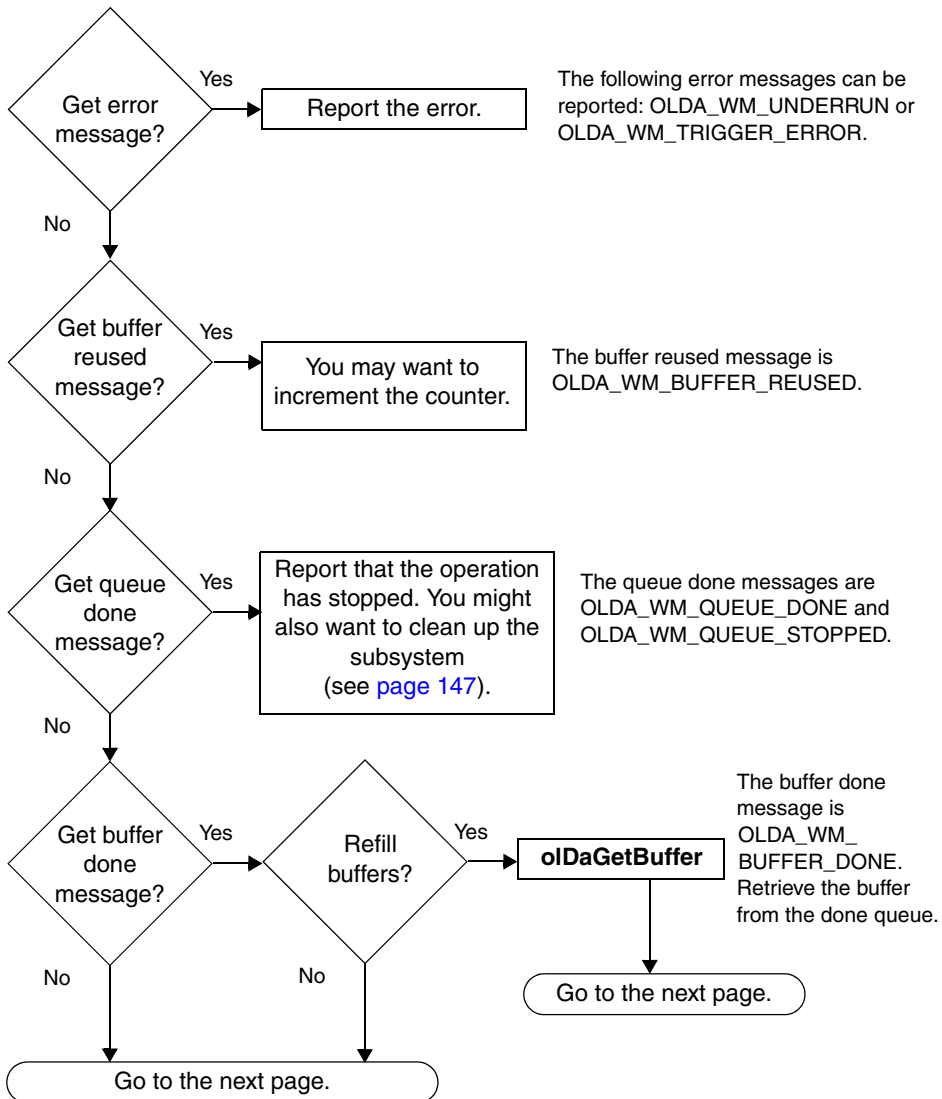


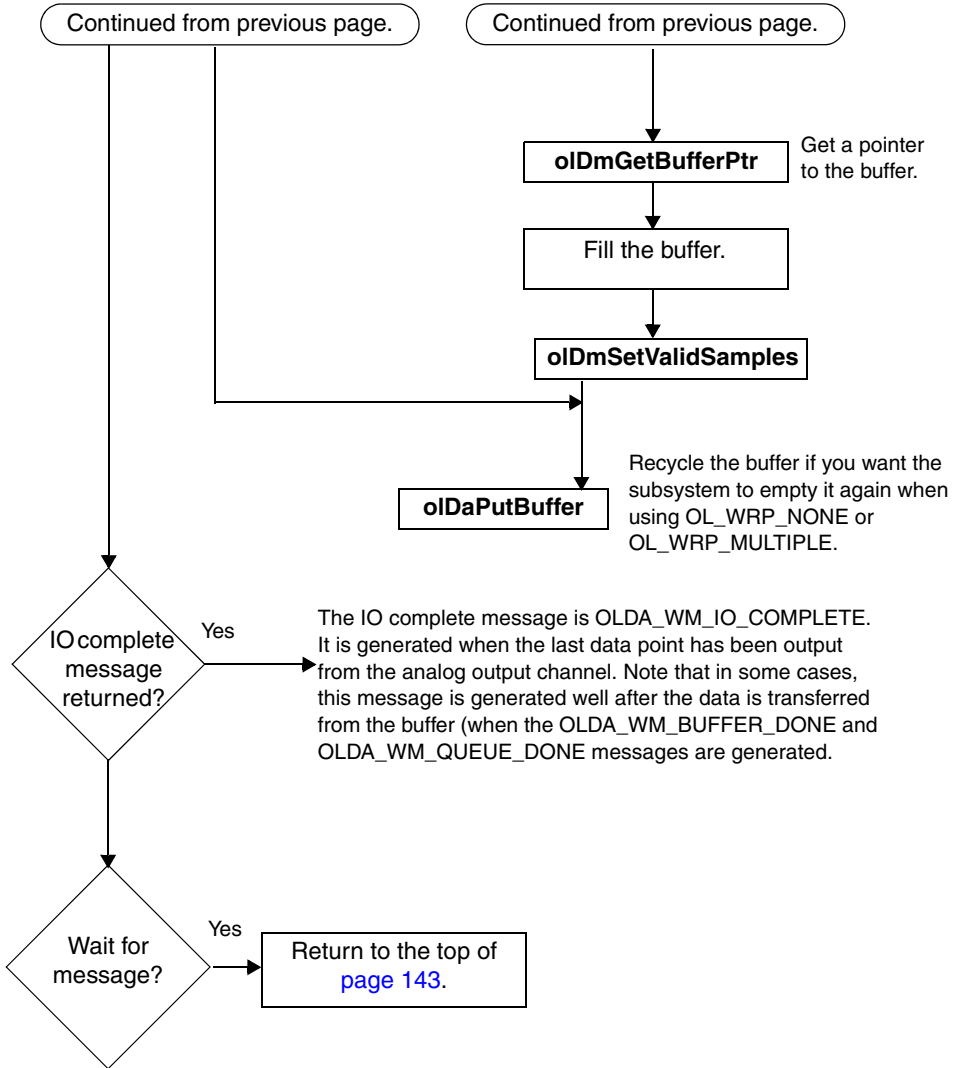


Transfer Data from an Inprocess Buffer

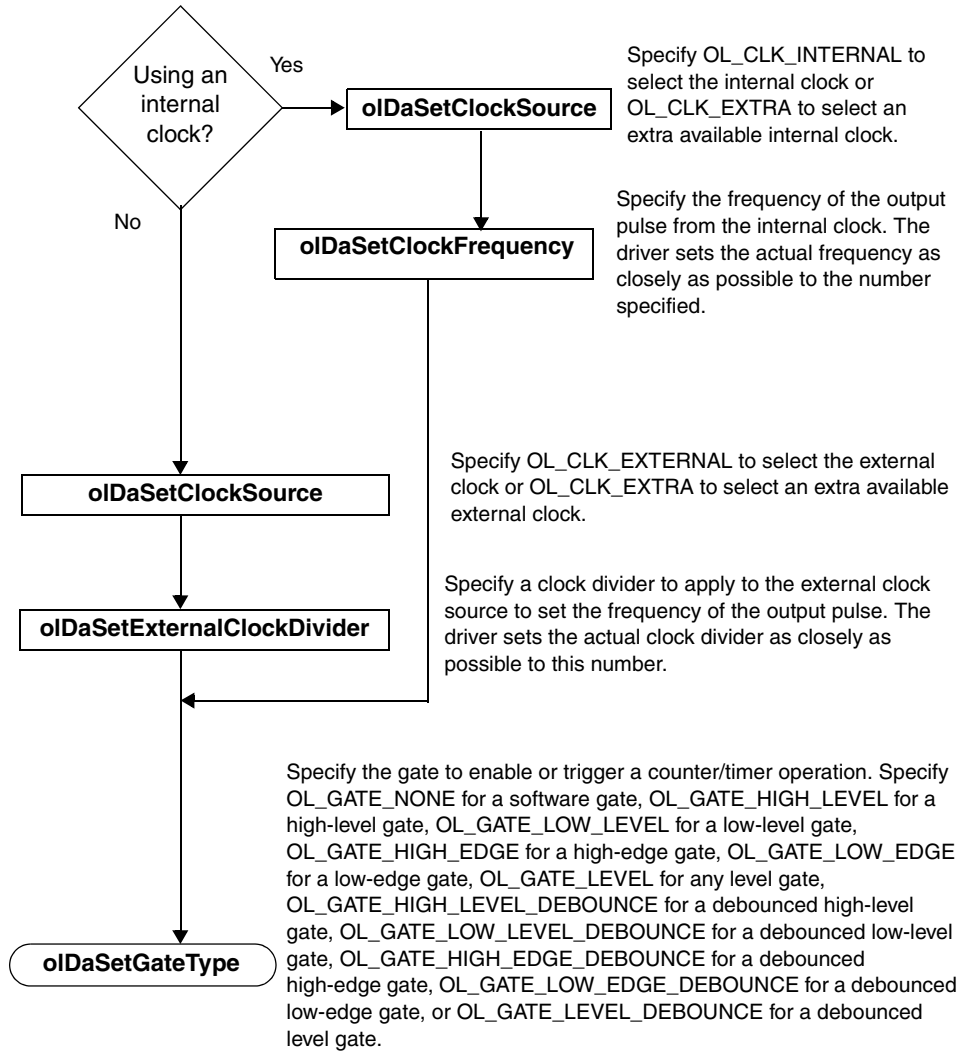


Deal with Messages and Buffers for Output Operations

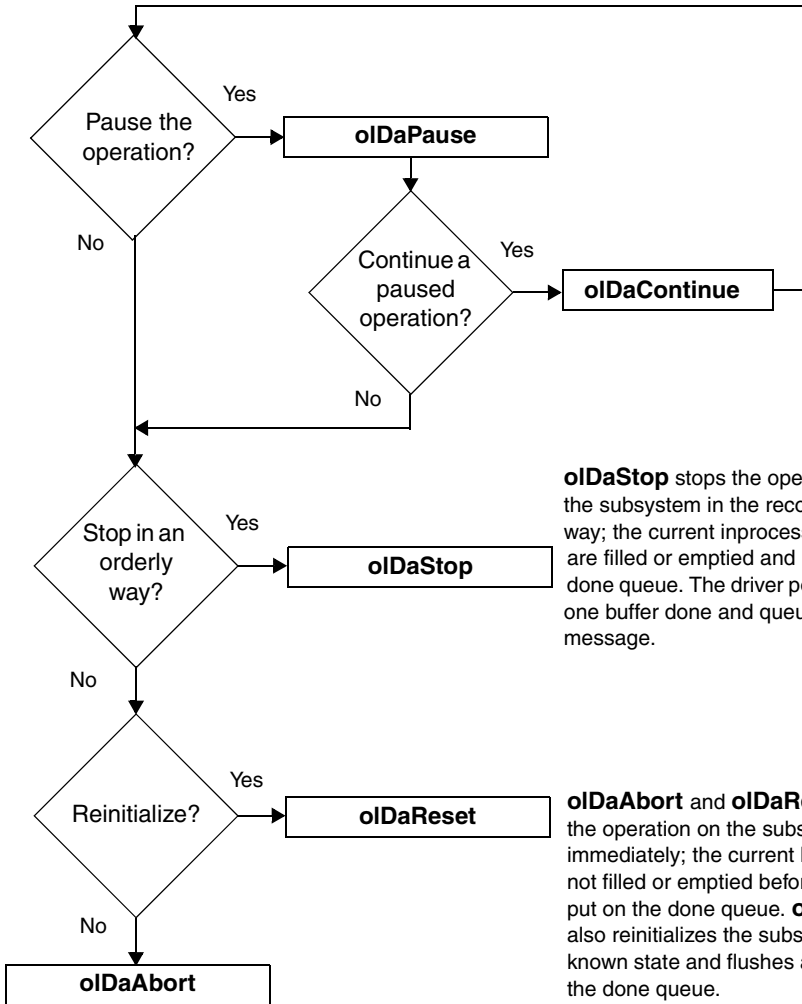




Set Clocks and Gates for Counter/Timer Operations



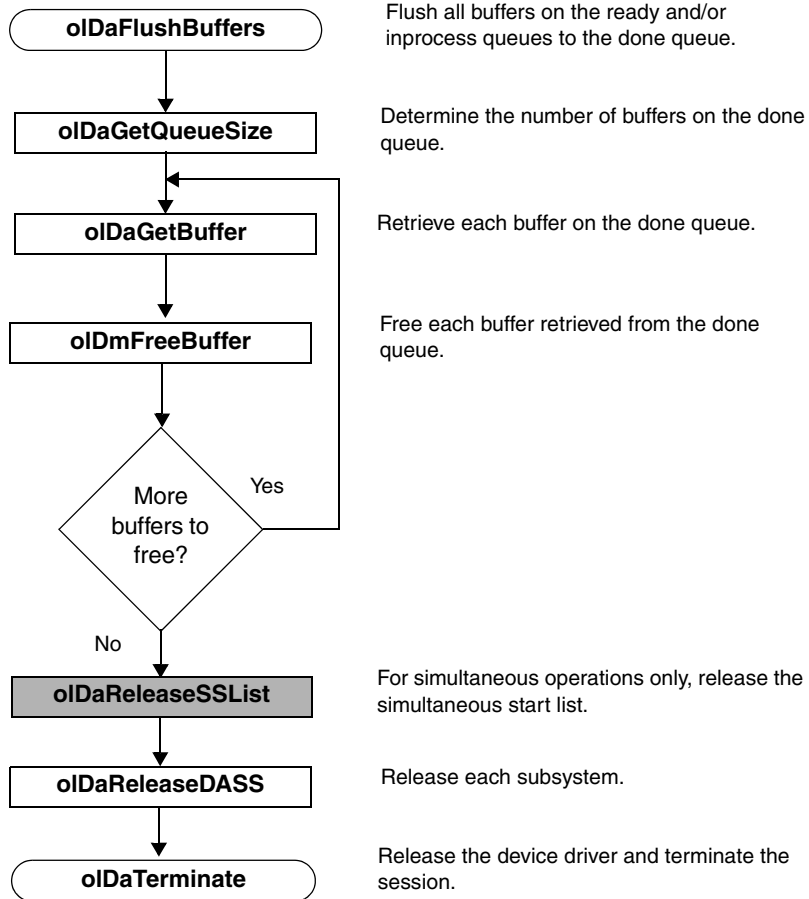
Stop the Operation



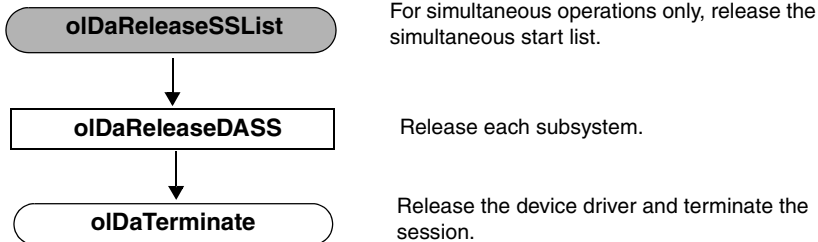
oIDaStop stops the operation on the subsystem in the recommended way; the current inprocess buffers are filled or emptied and put on the done queue. The driver posts at least one buffer done and queue stopped message.

oIDaAbort and **oIDaReset** stop the operation on the subsystem immediately; the current buffers are not filled or emptied before they are put on the done queue. **oIDaReset** also reinitializes the subsystem to a known state and flushes all buffers to the done queue.

Clean Up Buffered I/O Operations



Clean Up Counter/Timer Operations





Product Support

For the latest tips, software fixes, and other product information, you can always access our World-Wide Web site at the following address: <http://www.keithley.com>

Should you experience problems using the DataAcq SDK, follow these steps:

1. Read all the appropriate sections of this manual. Make sure that you have added any “Read This First” information to your manual and that you have used this information.
2. Check for a README file on the Keithley CD. If present, read this file for the latest installation and usage information.
3. Check that you have installed your hardware devices properly. For information, refer to the documentation supplied with your devices.
4. Check that you have installed the device drivers for your hardware devices properly. For information, refer to the documentation supplied with your devices.
5. Check that you have installed your software properly.

If you are still having problems, the Keithley Technical Support Department is available to provide technical assistance.

For the most efficient service, complete the form on [page 151](#) and be at your computer when you call for technical support. This information helps to identify specific system and configuration-related problems and to replicate the problem in house, if necessary.

Information Required for Technical Support

Name: _____ Phone _____

Contract Number: _____

Address: _____

Hardware product(s): _____

serial number: _____

configuration: _____

Device driver: _____

_____ version: _____

Software: _____

serial number: _____ version: _____

PC make/model: _____

operating system: _____ version: _____

Windows version: _____

processor: _____ speed: _____

RAM: _____ hard disk space: _____

network/number of users: _____ disk cache: _____

graphics adapter: _____ data bus: _____

I have the following boards and applications installed in my system: _____

I am encountering the following problem(s): _____

and have received the following error messages/codes: _____

I have run the board diagnostics with the following results: _____

You can reproduce the problem by performing these steps:

1. _____

2. _____

3. _____





Sample Code

Single-Value Analog Input	154
Continuous Analog Input	159

Single-Value Analog Input

The following code fragments illustrate the steps required to perform a single-value analog input operation. Refer to the example program `svadc.c` in the directory `C:\da_sdk\Examples\` for the entire program.

This program calls a user-defined function called `GetDriver()`, which enumerates the devices installed in the system.

Declare Variables and User Functions

This code fragment defines the variables used and the user-defined `GetDriver()` function; note that this program uses the device's default values for channel type, resolution, data encoding, range, and channel filter.

```
typedef struct tag_board {
    HDRVR  hdrvr;          /* driver handle          */
    HDASS  hdass;         /* subsystem handle      */
    ECODE  status;       /* board error status    */
    char  name[STRLEN];  /* string for board name */
    char  entry[STRLEN]; /* string for board name */
} BOARD;

typedef BOARD FAR* LPBOARD;

static BOARD board;

BOOL __export FAR PASCAL
GetDriver(lpszName, lpszEntry, lParam)

    LPSTR  lpszName;     /* board name            */
    LPSTR  lpszEntry;    /* system.ini entry      */
    LPARAM lParam;      /* optional user data    */
    UINT  channel = 0;
    DBL  gain = 1.0;
```



```
DBL min,max;
float volts;
long value;
UINT encoding,resolution;
```

Initialize the Driver

The following code fragment, in **WinMain ()**, calls the **CHECKERROR** error handler macro and the **oldaEnumBoards** function, which initializes the first available DT-Open Layers device. **oldaEnumBoards** calls **GetDriver()**, which lists the name of the device:

```
board.hdrvvr = NULL;
CHECKERROR(oldaEnumBoards(GetDriver,
    (LPARAM) (LPBOARD) &board));
```

This code fragment is in **GetDriver()** and gets the device name:

```
{
LPBOARD lpboard = (LPBOARD) (LPVOID) lParam;
/* fill in board strings */
lstrcpy(lpboard->name, lpszName, STRLEN);
lstrcpy(lpboard->entry, lpszEntry, STRLEN);
```

This code is in **WinMain()** and checks for errors within the callback function:

```
CHECKERROR (board.status);
/* check for NULL driver handle - means no boards */
if (board.hdrvvr == NULL){
    MessageBox(HWND_DESKTOP, "No DT-Open Layer
        boards!!!", "Error",
        MB_ICONEXCLAMATION | MB_OK);
return ((UINT) NULL);
}
```

This code fragment is in **WinMain()** and initializes the device:

```
lpboard->status = olDaInitialize(lpszName,
    &lpboard->hdrvr);
if (lpboard->hdrvr != NULL)
    return FALSE;
/* false to stop enumerating */
    else
        return TRUE;          /* true to continue */
}
```

Get a Handle to the Subsystem

The following code fragment gets a handle to the A/D subsystem and checks for errors:

```
CHECKERROR (olDaGetDASS (board.hdrvr, OLSS_AD, 0,
    &board.hdass));
```

Set the DataFlow to Single Value

The following code fragment sets the dataflow mode of the A/D subsystem to single value and checks for errors.

```
CHECKERROR (olDaSetDataFlow(board.hdass,
    OL_DF_SINGLEVALUE));
```

Configure the Subsystem

The following code fragment configures the A/D subsystem and checks for errors.

```
CHECKERROR (olDaConfig(board.hdass));
```

Acquire a Single Value

The following code fragment acquires a single analog input value from channel 0 of the A/D subsystem (using a gain of 1) and checks for errors.

```
CHECKERROR (olDaGetSingleValue(board.hdass, &value,  
                                channel, gain));
```

Convert the Value to Voltage

The following code fragment uses the default range, encoding, and resolution of the A/D subsystem to convert the acquired value into voltage and to check for errors. Note that this step is optional.

```
CHECKERROR (olDaGetRange (board.hdass, &max, &min));  
CHECKERROR (olDaGetEncoding (board.hdass,  
                              &encoding));  
CHECKERROR (olDaGetResolution (board.hdass,  
                               &resolution));  
  
/* Convert value to volts */  
if (encoding != OL_ENC_BINARY)  
{  
    /* convert to offset binary by inverting the */  
    /* sign bit */  
    value ^= 1L << (resolution-1);  
  
    /* zero upper bits */  
    value &= (1L << resolution) - 1;  
}  
volts=(float)max-(float)min)/(1L<<resolution)*  
    value+float)min;  
  
/* display value with message box */
```

```
    sprintf(str, "Single Value AD Op.\nADC Input =
             %.3f V", volts);
    MessageBox(HWND_DESKTOP, str, board.name,
              MB_ICONINFORMATION | MB_OK);
```

Release the Subsystem and Terminate the Session

The following code fragment releases the A/D subsystem, terminates the session, and checks for errors:

```
CHECKERROR (oldaReleaseDASS(board.hdass));
CHECKERROR (oldaTerminate(board.hdrv));
```

Handle Errors

The following code fragment handles the errors from the DataAcq SDK and displays the error codes. Note that this step is optional but recommended.

```
#define STRLEN 80 /* String size for general text*/
                /* manipulation. */
char str[STRLEN]; /* Global string for general */
                /* text manipulation      */

#define SHOW_ERROR(ecode)
MessageBox(HWND_DESKTOP, oldaGetErrorString(ecode,
      str, STRLEN), "Error",
          MB_ICONEXCLAMATION|MB_OK);

#define CHECKERROR(ecode) \
if ((board.status = (ecode)) != OLNOERROR)\
{\
    SHOW_ERROR(board.status);\
    oldaReleaseDASS(board.hdass);\
    oldaTerminate(board.hdrv);\
    return ((UINT)NULL);
}
```

Continuous Analog Input

The following code fragments illustrate the steps required to perform a continuous (post-trigger) analog input operation. Refer to the example program `contadc.c` in the directory `C:\da_sdk\Examples\` for the entire program.

This program calls two user-defined functions: **GetDriver()**, which enumerates the devices installed in the system, and **OutputBox()**, which creates a dialog box to handle information and error window messages from the A/D subsystem.

Declare Variables and User Functions

This code fragment defines the variables used and the user-defined **GetDriver()** function; note that this program uses the device's default values for channel type, resolution, data encoding, range, and channel filter.

```
typedef struct tag_board {
    HDRVR  hdrvr;          /* driver handle          */
    HDASS  hdass;         /* subsystem handle      */
    ECODE  status;       /* board error status    */
    HBUF   hbuf;         /* subsystem buffer handle */
    WORD FAR* lpbuf;     /* buffer pointer        */
    char  name[STRLEN];  /* string for board name  */
    char  entry[STRLEN]; /* string for board name  */
} BOARD;

typedef BOARD FAR* LPBOARD;

static BOARD board;
static ULNG count = 0;
BOOL __export FAR PASCAL GetDriver(lpszName,
                                   pszEntry, lParam)
```

```
LPSTR lpszName;          /* board name          */
LPSTR lpszEntry;        /* system.ini entry    */
LPARAM lParam;          /* optional user data  */

BOOL __export FAR PASCAL InputBox(hDlg, message,
    wParam, lParam)

HWND hDlg;
/* window handle of the dialog box */
UINT message;
/* type of message */
WPARAM wParam;
/* message-specific information */
LPARAM lParam;

DBL min,max;
float volts;
long value;
ULNG samples;
UINT encoding,resolution;

DBL freq;
UINT size,dma,gainsup;
int i;

UINT channel = 0;
DBL gain = 1.0;
```




Initialize the Driver

The following code fragment, in **WinMain()**, calls the **CHECKERROR** error handler macro and the **oldaEnumBoards** function, which initializes the first available DT-Open Layers device.

oldaEnumBoards calls **GetDriver()**, which lists the name of the device:

```
board.hdrvvr = NULL;
CHECKERROR(oldaEnumBoards(GetDriver,
    (LPARAM)(LPBOARD)&board));
CHECKERROR(board.status);

/* check for NULL driver handle - means no boards */
if (board.hdrvvr == NULL){
    MessageBox(HWND_DESKTOP, " No Open Layer
        boards!!!",
        "Error", MB_ICONEXCLAMATION | MB_OK);
    return ((UINT)NULL);
}
```

This code is in **GetDriver()** and gets the device name:

```
{
LPBOARD lpboard = (LPBOARD)(LPVOID)lParam;

/* fill in board strings */
lstrcpy(lpboard->name, lpszName, STRLEN);
lstrcpy(lpboard->entry, lpszEntry, STRLEN);
}
```

This code is in **WinMain()** and initializes the device:

```
lpboard->status = olDaInitialize(lpszName,
    &lpboard->hdrvr);
if (lpboard->hdrvr != NULL)
    return FALSE; /* false to stop enumerating */
    else
    return TRUE; /* true to continue */
}
```

Get a Handle to the Subsystem

The following code fragment gets a handle to the A/D subsystem and checks for errors:

```
CHECKERROR (olDaGetDASS(board.hdrvr, OLSS_AD, 0,
    &board.hdass));
```

Set the DataFlow to Continuous

The following code fragment sets the dataflow mode of the A/D subsystem to single value and checks for errors:

```
CHECKERROR (olDaSetDataFlow(board.hdass,
    OL_DF_CONTINUOUS));
```

Specify the Channel List and Channel Parameters

The following code fragment specifies a channel-gain list size of 1; specifies channel 0 in the channel-gain list; if the subsystem supports programmable gain, specifies a gain of 1 for this entry; and checks for errors:

```
/* Specify a channel-list size of 1.*/
CHECKERROR (olDaSetChannelListSize(board.hdass,1));

/* Specify a channel 0 in the channel list.*/
```



```
CHECKERROR (oldaSetChannelListEntry(board.hdass,0,
    channel));

/* Check if the subsystem supports programmable */
/* gain. */
CHECKERROR (oldaGetSSCaps(board.hdass,
    OLSSC_SUP_PROGRAMGAIN, gainsup));
/* Set the gain for entry 0 in the channel list */
/* if the board supports it. */
if (gainsup)
CHECKERROR (oldaSetGainListEntry(board.hdass,
    0,gain));
```

Specify the Clocks

The following code fragment specifies the frequency of the internal A/D sample clock and checks for errors:

```
/* Check the maximum frequency for the internal */
/* clock */
CHECKERROR(oldaGetSSCapsEx(board.hdass,
    OLSSCE_MAXTHROUGHPUT, &freq));

/* set 1000 Hz frequency */
freq = min (1000.0, freq);
CHECKERROR (oldaSetClockFrequency(board.hdass,
    freq));
```

Specify DMA Usage

The following code fragment specifies one DMA channel for the A/D subsystem and checks for errors:

```
/* Check the number of DMA channels supported. */
CHECKERROR (oldaGetSSCaps (board.hdass,
    OLSSC_NUMDMACHANS, &dma));

/* Set one dma channel.*/
dma = min (1, dma);
CHECKERROR (oldaSetDmaUsage (board.hdass, dma));
```

Set Up Window Handle and Buffering

The following code fragment specifies a handle to the message window, set up buffers, and check for errors:

```
/* Specify window handle. */
CLOSEONERROR (oldaSetWndHandle (board.hdass, hDlg,
    (UINT) NULL));

/*Specify the buffer wrap mode as multiple */
CHECKERROR (oldaSetWrapMode (board.hdass,
    OL_WRP_MULTIPLE));

/*Specify the buffers and put them on the ready */
/*queue. */
size = (UINT) freq/10;
/* Specify the buffer size. */

/* Allocate three input buffers and */
/* put the buffers on the ready queue. */
```



```
for (i=0;i<3;i++)
{
    CHECKERROR (oldmCallocBuffer(0,0,(ULONG) size,
        2,&board.hbuf));
    CHECKERROR (oldmGetBufferPtr(board.hbuf,
        (LPVOID FAR*) &board.lpbuf));
    CHECKERROR (oldaPutBuffer(board.hdass,
        board.hbuf));
}
```

Configure the Subsystem

The following code fragment configures the A/D subsystem and checks for errors.

```
CHECKERROR (oldaConfig(board.hdass));
```

Start the Continuous Analog Input Operation

The following code fragment acquires a single analog input value from channel 0 of the A/D subsystem (using a gain of 1) and checks for errors:

```
CLOSEONERROR (oldaStart(board.hdass));
```

Deal with Messages and Buffers

The following code fragment deals with messages and buffers for the A/D subsystem:

```
/* Use a dialog box to collect information */
/* and error messages from the subsystem. */
DialogBox(hInstance, (LPCSTR)INPUTBOX,
    HWND_DESKTOP, InputBox);

/* This function processes messages for the */
/* input dialog box. */
```

```
switch (message) {

    case WM_INITDIALOG:
        /* message: initialize dialog box*/
        /* set the title to the board name */
        SetWindowText(hDlg,board.name);
        return (TRUE); /* A message was returned. */

    case OLDA_WM_BUFFER_REUSED:
        /* message: buffer reused*/
        break;
    case OLDA_WM_BUFFER_DONE:
        /* message: buffer done*/

        /* Get buffer off the done queue. */
        CHECKERROR (oldaGetBuffer(board.hdass,
            &board.hbuf));
        /*If there is a buffer, get subsystem */
        /*information for code to volts conversion */
        if (board.hbuf != NULL){
            CLOSEONERROR (oldaGetRange(board.hdass,
                &max,&min));
            CLOSEONERROR (oldaGetEncoding(board.hdass,
                &encoding));
            CLOSEONERROR (oldaGetResolution (board.hdass,
                &resolution));

            /* get max samples in input buffer */
            CLOSEONERROR (oldmGetMaxSamples(board.hbuf,
                &samples));

            /* get last sample in buffer */
            value = board.lpbuf[samples-1];

            /* Get pointer to buffer data */
            CHECKERROR (oldmGetBufferPtr (board.hbuf,
                (LPVOID FAR*) &board.lpbuf));
```



```
/* Put buffer back on the ready queue. */
CHECKERROR (oldaPutBuffer(board.hdass,
    board.hbuf));

case OLDA_WM_QUEUE_DONE:
/* using wrap multiple or none */
/* if this message is received, */
/* acquisition has stopped. */
EndDialog(hDlg, TRUE);

case OLDA_WM_QUEUE_STOPPED:
/* using wrap multiple or none */
/* if this message is received, */
/* acquisition has stopped. */
EndDialog(hDlg, TRUE);

case OLDA_WM_TRIGGER_ERROR:

/* Process trigger error message */
MessageBox(hDlg, "Trigger error: acquisition
    stopped", "Error", MB_ICONEXCLAMATION |
    MB_OK);
EndDialog(hDlg, TRUE);

case OLDA_WM_OVERRUN_ERROR:
/* Process underrun error message */
MessageBox(hDlg, "Input overrun error:
    acquisition stopped", "Error",
    MB_ICONEXCLAMATION | MB_OK);
EndDialog(hDlg, TRUE);

case WM_COMMAND:
/* message: received a command */
#ifdef WIN32
switch ( LOWORD(wParam) ) {
#else
switch (wParam) {
```

```
        #endif
        case IDOK:
        case IDCANCEL:
            CLOSEONERROR (oldAbort(board.hdass));
            EndDialog(hDlg, TRUE);
            return (TRUE); /* Did process a message. */
    }
    break;
    }
    return (FALSE);      /* Didn't get a message */
}
```

Convert Values to Voltage

The following code fragment uses the default range, encoding, and resolution of the A/D subsystem to convert the acquired values into voltages and to check for errors. Note that this step is optional.

```
/* convert value to volts */
if (encoding != OL_ENC_BINARY) {

/* convert to offset binary by inverting */
/* sign bit */
value ^= 1L << (resolution-1);

/* zero upper bits */
value &= (1L << resolution) - 1;
}
volts = ((float)max-(float)min)/(1L<<resolution) *
        value + (float)min;

/* display value */
sprintf(str, "%.3f Volts", volts);
SetDlgItemText (hDlg, IDD_VALUE, str);
}
```


Clean Up

The following code fragment flushes the buffers, releases the subsystem, and terminates the program when the continuous A/D operation is complete:

```
/* Get the input buffers from the done queue and */
/* free the input buffers */
CHECKERROR (oldaFlushBuffers(board.hdass));

for (i=0;i<3;i++)
{
    CHECKERROR (oldaGetBuffer(board.hdass,
        &board.hbuf));
    CHECKERROR (oldmFreeBuffer(board.hbuf));
}

/* release the subsystem and terminate */
/* the session */
CHECKERROR (oldaReleaseDASS(board.hdass));
CHECKERROR (oldaTerminate(board.hdrv));
```

Handle Errors

The following code fragment handles the errors from the DataAcq SDK and displays the error codes. Note that this step is optional but recommended.

```
/* Error handling macros */

#define STRLEN 80 /* String size for general */
                /* text manipulation */
char str[STRLEN]; /* Global string for general */
                /* text manipulation. */
```

```
#define SHOW_ERROR(ecode)
MessageBox(HWND_DESKTOP,oldaGetErrorString(ecode,
    (str,STRLEN),"Error", MB_ICONEXCLAMATION |
    MB_OK);
#define CHECKERROR(ecode)
    if ((board.status = (ecode))!= OLNOERROR)\
    {\
        SHOW_ERROR(board.status);\
        oldaReleaseDASS(board.hdass);\
        oldaTerminate(board.hdrv);
    return ((UINT)NULL);}
#define CLOSEONERROR(ecode)
    if ((board.status = (ecode)) != OLNOERROR)\
    {\
        SHOW_ERROR(board.status);\
        oldaReleaseDASS(board.hdass);\
        oldaTerminate(board.hdrv);\
        EndDialog(hDlg, TRUE);\
    return (TRUE);}
```

Index

A

- A/D subsystem 38
- aborting an operation
 - A/D 55
 - D/A 55
 - digital input 55
 - digital output 55
 - edge-to-edge measurement 91
 - event counting 83
 - rate generation 94
 - repetitive one-shot 99
 - simultaneous 111
 - up/down counting 85
- about the example programs 5
- about-trigger mode 59
- aliasing 66
- analog event trigger 70
- analog input channel configuration
 - differential 43
 - pseudo-differential 44
 - single-ended 43
- analog input operations 41
 - inhibiting channels in the channel list 47
 - specifying a single channel 44
 - specifying buffers 71
 - specifying clock sources 65
 - specifying DMA channels 79
 - specifying filters 53
 - specifying gain 51
 - specifying ranges 50
 - specifying synchronous digital I/O values in the channel list 48
 - specifying the channel list size 45
 - specifying the channel type 43
 - specifying the channels in the channel list 46
 - specifying the data encoding 41
 - specifying the data flow 54
 - specifying the resolution 42
 - specifying trigger sources 67
 - specifying triggered scan mode 61
- analog output operations 41
 - specifying a single channel 44
 - specifying buffers 71
 - specifying clock sources 65
 - specifying DMA channels 79
 - specifying filters 53
 - specifying gain 51
 - specifying ranges 50
 - specifying the channel list size 45
 - specifying the channel type 43
 - specifying the channels in the channel list 46
 - specifying the data encoding 41
 - specifying the data flow 54
 - specifying the resolution 42
 - specifying trigger sources 67
- analog threshold (negative) trigger 69
- analog threshold (positive) trigger 69
- architecture 35
- autoranging 28, 54

B

- binary data format 41
- buffer list management functions 33
- buffer management functions 31
- buffers 71, 140, 143
 - done queue 74
 - inprocess queue 72
 - ready queue 71
 - transferring data from an inprocess
 - buffer 142
 - wrap modes 78

C

- C/T clock sources 101
 - external C/T clock 102
 - extra C/T clock 104
 - internal C/T clock 102
 - internally cascaded C/T clock 103
- C/T subsystem 38
- calling conventions 7
- capabilities, used with
 - olDaGetSSCaps** 13
- capabilities, used with
 - olDaGetSSCapsEx** 20
- cascading counters 103
- channel list inhibition 47
- channel list size 45
- channel list specification 46
- channel list, specifying synchronous
 - digital I/O values 48
- channel type 43
- cleaning up operations
 - A/D 147
 - counter/timer 148
 - D/A 147
 - digital I/O 147

- clock divider 127
- clock input signal 81
- clock sources
 - clock divider 127
 - external 66
 - external C/T clock 102
 - extra 67
 - extra C/T clock 104
 - internal 65, 123, 127
 - internal C/T clock 102
 - internally cascaded C/T clock 103
- configuration functions 22
- configuring a subsystem 39
- continuing an operation
 - A/D 56
 - D/A 56
 - digital input 56
 - digital output 56
 - edge-to-edge measurement 91
 - event counting 83
 - rate generation 94
 - repetitive one-shot 100
 - up/down counting 85
- continuous operations 55
 - continuous (post-trigger) mode 57
 - continuous about-trigger mode 59
 - continuous pre-trigger mode 58
 - flowcharts for A/D 117
 - flowcharts for D/A 119
 - flowcharts for digital input 117
 - flowcharts for digital output 119
 - sample code for continuous A/D 159
- continuous pulse output 93
- conventions used **xiv**
- conventions, calling 7
- conversion rate 63, 65

- counter/timer operations 81
 - C/T clock sources 101
 - channels 81
 - duty cycle 108
 - gate types 104
 - operation modes 82
 - pulse output types 108
- counting events 82, 84, 90
- D**
- D/A subsystem 38
- data acquisition functions 10
 - configuration 22
 - conversion 30
 - information 10
 - initialization and termination 21
 - operation 27
- data buffers 71
- data conversion functions 30
- data encoding 41
- data flow modes 54
- data management functions 31
 - buffer list management 33
 - buffer management 31
- data transfer, from an inprocess buffer 142
- DataAcq SDK
 - architecture 35
 - overview 2
- dealing with messages and buffers
 - input operations 140
 - output operations 143
- debounced gate type
 - any level 108
 - high-edge 107
 - high-level 107
 - low-edge 108
 - low-level 107
- device, initialization 37
- differential inputs 43
- digital event trigger 70
- digital input operations 41
 - specifying a single channel 44
 - specifying buffers 71
 - specifying clock sources 65
 - specifying DMA channels 79
 - specifying gain 51
 - specifying the channel list size 45
 - specifying the channel type 43
 - specifying the channels in the channel list 46
 - specifying the data flow 54
 - specifying the resolution 42
 - specifying trigger sources 67
- digital output operations 41
 - specifying a single channel 44
 - specifying buffers 71
 - specifying clock sources 65
 - specifying DMA channels 79
 - specifying gain 51
 - specifying the channel list size 45
 - specifying the channel type 43
 - specifying the channels in the channel list 46
 - specifying the data flow 54
 - specifying the resolution 42
 - specifying trigger sources 67
- DIN subsystem 38
- DMA channels 79
- done queue 74
- DOUT subsystem 38
- DT-Open Layers 2
- duty cycle 108

E

- edge-to-edge measurement operations
 - 90
 - flowcharts 127
- encoding 41
- error checking 114
- error codes 40
- event counting operations 82
 - flowcharts 121
- event trigger
 - analog 70
 - digital 70
 - timer 70
- example programs 5
- external analog threshold (negative) trigger 69
- external analog threshold (positive) trigger 69
- external C/T clock 102
- external clock source 66
- external digital (TTL) trigger 68
- external retrigger mode 62
- extra C/T clock 104
- extra clock sources 67
- extra retrigger mode 64
- extra trigger sources 70

F

- filters 53
- flowcharts 113
 - cleaning up A/D operations 147
 - cleaning up counter/timer operations 148
 - cleaning up D/A operations 147
 - cleaning up digital I/O operations 147

- continuous A/D operations 117
- continuous D/A operations 119
- continuous digital input operations 117
- continuous digital output operations 119
- dealing with messages and buffers 140, 143
- edge-to-edge measurement operations 127
- event counting operations 121
- frequency measurement operations 125
- pulse output operations 129
- setting up buffers for A/D operations 138
- setting up buffers for D/A operations 139
- setting up buffers for digital input operations 138
- setting up buffers for digital output operations 139
- setting up channel lists 134
- setting up clocks 135, 145
- setting up gates 145
- setting up pre-triggers 135
- setting up subsystem parameters 133
- setting up triggered scans 137
- setting up triggers 135
- simultaneous operations 131
- single-value operations 115
- stopping operations 146
- transferring data from an inprocess buffer 142
- up/down counting operations 123

frequency measurement operations [86](#)
 flowcharts [125](#)
 using a pulse of a known duration [87](#)
 using the Windows timer [86](#)
 functions
 buffer list management [33](#)
 buffer management [31](#)
 configuration [22](#)
 data conversion [30](#)
 information [10](#)
 initialization and termination [21](#)
 operation [27](#)
 summary [9](#)

G

gain list [51](#)
 gains [51](#)
 gate input signal [81](#), [104](#)
 gate types [104](#)
 any level [106](#)
 high-edge [106](#)
 high-edge, debounced [107](#)
 high-level [105](#)
 high-level, debounced [107](#)
 level, debounced [108](#)
 low-edge [106](#)
 low-edge, debounced [108](#)
 low-level [105](#)
 low-level, debounced [107](#)
 software [105](#)

H

handling errors [40](#)
 handling messages [40](#)
 help, launching online [4](#)

high-edge gate type [106](#)
 high-edge, debounced gate type [107](#)
 high-level debounced gate type [107](#)
 high-level gate type [105](#)
 high-to-low pulse output [108](#)

I

information functions [10](#)
 inhibiting channels in the channel list
[47](#)
 initialization functions [21](#)
 initializing a device [37](#)
 inprocess buffer, transferring data [142](#)
 inprocess queue [72](#)
 input configuration
 differential analog [43](#)
 single-ended analog [43](#)
 internal
 clock [123](#), [127](#)
 internal C/T clock [102](#)
 internal clock sources [65](#)
 internal retrigger mode [63](#)
 internal trigger [68](#)
 interrupts [79](#)

L

launching the online help [4](#)
 level gate type [106](#)
 level, debounced gate type [108](#)
 list management functions [33](#)
 LongtoFreq macro [126](#)
 low-edge gate type [106](#)
 low-edge, debounced gate type [108](#)
 low-level gate type [105](#)
 low-level, debounced gate type [107](#)

low-to-high pulse output 109

M

macro 126

measuring frequency 86

message handling 40

flowcharts for A/D 140

flowcharts for D/A 143

flowcharts for digital input 140

flowcharts for digital output 143

messages

OLDA_WM_BUFFER_DONE 74

OLDA_WM_BUFFER_REUSED 78

OLDA_WM_MEASURE_DONE 87

OLDA_WM_PRETRIGGER_BUFFER_DONE 74

OLDA_WM_QUEUE_DONE 78

OLDA_WM_QUEUE_STOPPED 74

multiple wrap mode 78

N

no wrap mode 78

Nyquist Theorem 66

O

OL_ENUM_FILTERS 53

OL_ENUM_GAINS 52

OL_ENUM_RANGES 50

OLDA_WM_BUFFER_DONE 74, 140, 143

OLDA_WM_BUFFER_REUSED 78, 140, 143

OLDA_WM_IO_COMPLETE 77, 144

OLDA_WM_MEASURE_DONE 87

OLDA_WM_OVERRUN 140

OLDA_WM_PRETRIGGER_BUFFER_DONE 74, 140

OLDA_WM_QUEUE_DONE 78, 140, 143

OLDA_WM_QUEUE_STOPPED 74, 140, 143

OLDA_WM_TRIGGER_ERROR 63, 65, 140, 143

OLDA_WM_UNDERRUN 143

olDaAbort 28, 146

in A/D operations 55

in D/A operations 55

in digital input operations 55

in digital output operations 55

in edge-to-edge measurement operations 91

in event counting operations 83

in rate generation operations 94

in repetitive one-shot operations 99

in simultaneous operations 111

in up/down counting operations 85

olDaCodeToVolts 30, 55, 76, 116, 141

olDaConfig 28, 39

in continuous A/D operations 118

in continuous D/A operations 120

in continuous digital input operations 118

in continuous digital output operations 120

in edge-to-edge measurement operations 127

in event counting operations 121

in frequency measurement operations 126

in pulse output operations 130

in single-value operations 115

- in up/down counting operations 123
- olDaContinue** 28, 146
 - in A/D operations 56
 - in D/A operations 56
 - in digital input operations 56
 - in digital output operations 56
 - in edge-to-edge measurement operations 91
 - in event counting operations 83
 - in rate generation operations 94
 - in repetitive one-shot operations 100
 - in up/down counting operations 85
- olDaEnumBoards** 10, 37, 114
- olDaEnumBoardsEx** 11
- olDaEnumLists** 33
- olDaEnumSSCaps** 11, 114
- olDaEnumSSList** 12
- olDaEnumSubSystems** 11, 39, 114
- olDaFlushBuffers** 29, 77, 147
- olDaFlushFromBufferInProcess** 29, 73, 76, 142
- olDaFreeBuffer** 77
- olDaGetBoardInfo** 10, 37
- olDaGetBuffer** 29, 76, 147
 - in continuous A/D operations 140
 - in continuous D/A operations 143
 - in continuous digital input operations 140
 - in continuous digital output operations 143
- olDaGetCascadeMode** 26
- olDaGetChannelFilter** 24
- olDaGetChannelListEntry** 23
- olDaGetChannelListEntryInhibit** 24
- olDaGetChannelListSize** 23
- olDaGetChannelRange** 25
- olDaGetChannelType** 24
- olDaGetClockFrequency** 26
- olDaGetClockSource** 26
- olDaGetCTMode** 26
- olDaGetDASS** 21, 38
 - in continuous A/D operations 117
 - in continuous D/A operations 119
 - in continuous digital input operations 117
 - in continuous digital output operations 119
 - in edge-to-edge measurement operations 127
 - in event counting operations 121
 - in frequency measurement operations 125
 - in pulse output operations 129
 - in single-value operations 115
 - in up/down counting operations 123
- olDaGetDASSInfo** 11
- olDaGetDataFlow** 22
- olDaGetDevCaps** 11, 39, 114
- olDaGetDeviceName** 11
- olDaGetDigitalIOListEntry** 24
- olDaGetDmaUsage** 22
- olDaGetDriverVersion** 12
- olDaGetEncoding** 25
- olDaGetErrorMessage** 12
- olDaGetExternalClockDivider** 26
- olDaGetGainListEntry** 23
- olDaGetGateType** 26
- olDaGetMeasureStartEdge** 27
- olDaGetMeasureStopEdge** 27
- olDaGetMultiscanCount** 23
- olDaGetNotificationProcedure** 22
- olDaGetPretriggerSource** 25
- olDaGetPulseType** 26
- olDaGetPulseWidth** 27

- olDaGetQueueSize** 12, 142, 147
- olDaGetRange** 25
- olDaGetResolution** 25
- olDaGetRetrigger** 25
- olDaGetRetriggerFrequency** 23
- olDaGetRetriggerMode** 23
- olDaGetSingleValue** 28, 55, 116
- olDaGetSingleValueEx** 28, 54, 116
- olDaGetSSCaps** 11, 114
- olDaGetSSCapsEx** 11, 114
- olDaGetSSList** 30, 110, 131
- olDaGetSynchronousDigitalIOUsage** 24
- olDaGetTrigger** 25
- olDaGetTriggeredScanUsage** 23
- olDaGetVersion** 12
- olDaGetWndHandle** 22
- olDaGetWrapMode** 22
- olDaInitialize** 21, 37
 - in continuous A/D operations 117
 - in continuous D/A operations 119
 - in continuous digital input operations 117
 - in continuous digital output operations 119
 - in edge-to-edge measurement operations 127
 - in event counting operations 121
 - in frequency measurement operations 125
 - in pulse output operations 129
 - in single-value operations 115
 - in up/down counting operations 123
- olDaMeasureFrequency** 29, 86, 126
- olDaPause** 28, 146
 - in A/D operations 56
 - in D/A operations 56
- in digital input operations 56
- in digital output operations 56
- in edge-to-edge measurement operations 91
- in event counting operations 83
- in rate generation operations 94
- in repetitive one-shot operations 100
- in up/down counting operations 85
- olDaPutBuffer** 29, 72, 76
 - in continuous A/D operations 138, 141
 - in continuous D/A operations 139, 144
 - in continuous digital input operations 141
 - in continuous digital output operations 139, 144
- olDaPutDassToSSList** 30, 110, 131
- olDaPutSingleValue** 28, 55, 116
- olDaReadEvents** 29, 83, 85, 91, 122, 124
- olDaReleaseDASS** 21, 40, 111
 - in A/D operations 147
 - in counter/timer operations 148
 - in D/A operations 147
 - in digital input operations 147
 - in digital output operations 147
 - in event counting operations 128
 - in single-value operations 116
- olDaReleaseSSList** 30, 111, 147, 148
- olDaReset** 28, 146
 - in A/D operations 55
 - in D/A operations 55
 - in digital input operations 55
 - in digital output operations 55
 - in edge-to-edge measurement operations 91

- in event counting operations 83
- in rate generation operations 94
- in repetitive one-shot operations 99
- in simultaneous operations 111
- in up/down counting operations 85
- olDataSetCascadeMode** 26, 103
 - in event counting operations 121
 - in frequency measurement operations 125
 - in pulse output operations 129
- olDataSetChannelFilter** 24, 53, 133
- olDataSetChannelListEntry** 23, 46, 134
- olDataSetChannelListEntryInhibit** 24, 47, 134
- olDataSetChannelListSize** 23, 45, 134
- olDataSetChannelRange** 25, 50, 133
- olDataSetChannelType** 24, 43, 133
- olDataSetClockFrequency** 26, 65, 93, 102, 135
 - in C/T operations 145
- olDataSetClockSource** 26, 65, 66, 102, 103, 123, 127, 135
 - in C/T operations 145
- olDataSetCTMode** 26
 - in edge-to-edge measurement operations 91, 127
 - in event counting operations 82, 88, 121
 - in frequency measurement operations 125
 - in one-shot mode 88
 - in one-shot operations 86, 96
 - in pulse output operations 129
 - in rate generation operations 93
 - in repetitive one-shot operations 99
 - in up/down counting operations 84, 123
- olDataSetDataFlow** 22
 - in continuous (post-trigger) operations 57
 - in continuous A/D operations 117
 - in continuous about-trigger operations 59
 - in continuous D/A operations 119
 - in continuous digital input operations 117
 - in continuous digital output operations 119
 - in continuous pre-trigger operations 58
 - in single-value operations 54, 115
- olDataSetDigitalIOListEntry** 24, 48, 134
- olDataSetDmaUsage** 22, 80
 - in continuous A/D operations 117
 - in continuous D/A operations 119
 - in continuous digital input operations 117
 - in continuous digital output operations 119
- olDataSetEncoding** 25, 41, 133
- olDataSetExternalClockDivider** 26, 66, 93, 103, 127, 135
 - in C/T operations 145
- olDataSetGainListEntry** 23, 52, 134
- olDataSetGateType** 26, 104, 105, 145
- olDataSetMeasureStartEdge** 27, 127
- olDataSetMeasureStopEdge** 27, 127
- olDataSetMultiscanCount** 23, 61, 137
- olDataSetNotificationProcedure** 22, 40
 - in continuous A/D operations 138
 - in continuous D/A operations 139
 - in continuous digital input operations 138

- in continuous digital output operations 139
- olDataSetPretriggerSource** 25, 58, 60, 68, 136
- olDataSetPulseType** 26, 109, 130
- olDataSetPulseWidth** 27, 109, 130
- olDataSetRange** 25, 50, 133
- olDataSetResolution** 25, 42, 133
- olDataSetRetrigger** 25, 64, 68, 137
- olDataSetRetriggerFrequency** 23, 63, 137
- olDataSetRetriggerMode** 23, 62, 63, 64, 137
- olDataSetSynchronousDigitalIOUsage** 24, 48, 134
- olDataSetTrigger** 25, 57, 68, 135
- olDataSetTriggeredScanUsage** 23, 61, 137
- olDataSetWndHandle** 22, 40
 - in continuous A/D operations 138
 - in continuous D/A operations 139
 - in continuous digital input operations 138
 - in continuous digital output operations 139
- olDataSetWrapMode** 22, 78
 - in continuous A/D operations 138
 - in continuous D/A operations 139
 - in continuous digital input operations 138
 - in continuous digital output operations 139
- olDaSimultaneousPreStart** 30, 110, 131
- olDaSimultaneousStart** 30, 110, 131
- olDaStart** 28
 - in A/D operations 55, 118
 - in D/A operations 55, 120
 - in digital input operations 55, 118
 - in digital output operations 55, 120
 - in edge-to-edge measurement operations 91, 128
 - in event counting operations 83, 122
 - in one-shot operations 97
 - in pulse output operations 130
 - in rate generation operations 94
 - in repetitive one-shot operations 99
 - in simultaneous operations 111
 - in up/down counting operations 85, 124
- olDaStop** 28, 146
 - in A/D operations 55
 - in D/A operations 55
 - in digital input operations 55
 - in digital output operations 55
 - in edge-to-edge measurement operations 91
 - in event counting operations 83
 - in rate generation operations 94
 - in repetitive one-shot operations 99
 - in simultaneous operations 111
 - in up/down counting operations 85
- olDaTerminate** 21, 40, 111, 147
 - in C/T operations 148
 - in digital input operations 147
 - in digital output operations 147
 - in event counting operations 128
 - in single-value operations 116
- olDaVoltsToCode** 30, 55, 76, 116, 141
- olDmAllocBuffer** 32, 71
 - in A/D operations 138
 - in D/A operations 139
 - in digital input operations 138
 - in digital output operations 139

- in inprocess buffer operations 142
- olDmCallocBuffer** 32, 71
 - in A/D operations 138
 - in D/A operations 139
 - in digital input operations 138
 - in digital output operations 139
 - in inprocess buffer operations 142
- olDmCopyBuffer** 32
- olDmCopyFromBuffer** 32
- olDmCopyToBuffer** 32
- olDmCreateList** 33
- olDmEnumBuffers** 33
- olDmFreeBuffer** 32, 147
- olDmFreeList** 33
- olDmGetBufferFromList** 33
- olDmGetBufferPtr** 32, 76
 - in continuous A/D operations 141
 - in continuous D/A operations 144
 - in continuous digital input operations 141
 - in continuous digital output operations 144
- olDmGetBufferSize** 32
- olDmGetDataBits** 32
- olDmGetDataWidth** 32
- olDmGetErrorString** 32
- olDmGetListCount** 33
- olDmGetListHandle** 33
- olDmGetListIDs** 34
- olDmGetMaxSamples** 32
- olDmGetTimeDateStamp** 32
- olDmGetValidSamples** 32
 - in continuous A/D operations 140
 - in continuous digital input operations 140
- olDmGetVersion** 32
- olDmMallocBuffer** 32, 71
 - in A/D operations 138
 - in D/A operations 139
 - in digital input operations 138
 - in digital output operations 139
 - in inprocess buffer operations 142
- olDmPeekBufferFromList** 34
- olDmPutBufferToList** 34
- olDmReAllocBuffer** 32
- olDmReCallocBuffer** 33
- olDmReMallocBuffer** 33
- olDmSetDataWidth** 32
- olDmSetValidSamples** 32, 139
 - in continuous D/A operations 144
 - in continuous digital output operations 144
- OLSSC_SUP_SEQUENTIAL_CGL 45
- OLSS_SUP_EXTCLOCK 17
- OLSSC_CGLDEPTH 15, 45
- OLSSC_FIFO_SIZE_IN_K 20
- OLSSC_MAXDICHANS 16
- OLSSC_MAXDIGITALIOLIST_VALUE 15, 48
- OLSSC_MAXMULTISCAN 14, 61
- OLSSC_MAXSECHANS 43
- OLSSC_NUM_DMACHANS 80
- OLSSC_NUM_RANGES 50
- OLSSC_NUM_RESOLUTION 42
- OLSSC_NUMCHANNELS 16, 42
- OLSSC_NUMDMACHANS 14
- OLSSC_NUMEXTRACLOCKS 17, 67, 104
- OLSSC_NUMEXTRATRIGGERS 17, 70
- OLSSC_NUMFILTERS 16, 53
- OLSSC_NUMGAINS 15, 52
- OLSSC_NUMRANGES 16

OLSSC_NUMRESOLUTIONS 16
OLSSC_RANDOM_CGL 45
OLSSC_SUP_2SCOMP 16, 41
OLSSC_SUP_ANALOGEVENTTRIG
17, 70
OLSSC_SUP_BINARY 16, 41
OLSSC_SUP_BUFFERING 13, 71
OLSSC_SUP_CASCADING 18, 103
OLSSC_SUP_CHANNELLIST_
INHIBIT 15, 47
OLSSC_SUP_CONTINUOUS 13, 57
OLSSC_SUP_CONTINUOUS_
ABOUTTRIG 13, 59
OLSSC_SUP_CONTINUOUS_
PRETRIG 13, 58
OLSSC_SUP_CTMODE_COUNT 18,
82, 86
OLSSC_SUP_CTMODE_MEASURE
18, 90
OLSSC_SUP_CTMODE_ONESHOT
18, 96
OLSSC_SUP_CTMODE_ONESHOT_
RPT 18, 99
OLSSC_SUP_CTMODE_RATE 18, 93
OLSSC_SUP_CTMODE_UP_DOWN
18, 84
OLSSC_SUP_DIFFERENTIAL 16, 43
OLSSC_SUP_DIGITALEVENTTRIG
17, 70
OLSSC_SUP_EXP2896 16, 42
OLSSC_SUP_EXP727 16, 42
OLSSC_SUP_EXTCLOCK 66, 102
OLSSC_SUP_EXTERNTRIG 17, 68
OLSSC_SUP_FIFO 20
OLSSC_SUP_FILTERPERCHAN 16,
53
OLSSC_SUP_GAPFREE_DUALDMA
14, 79
OLSSC_SUP_GAPFREE_NODMA 14,
79
OLSSC_SUP_GAPFREE_
SINGLEDMA 14, 79
OLSSC_SUP_GATE_GATE_LEVEL 19
OLSSC_SUP_GATE_HIGH_EDGE 19,
106
OLSSC_SUP_GATE_HIGH_EDGE_
DEBOUNCE 19, 107, 108
OLSSC_SUP_GATE_HIGH_LEVEL
19, 105
OLSSC_SUP_GATE_HIGH_LEVEL_
DEBOUNCE 19, 107
OLSSC_SUP_GATE_LEVEL 106
OLSSC_SUP_GATE_LEVEL_
DEBOUNCE 19
OLSSC_SUP_GATE_LOW_EDGE 19,
106
OLSSC_SUP_GATE_LOW_EDGE_
DEBOUNCE 19
OLSSC_SUP_GATE_LOW_LEVEL 19,
105
OLSSC_SUP_GATE_LOW_LEVEL_
DEBOUNCE 19
OLSSC_SUP_GATE_NONE 19, 105
OLSSC_SUP_INPROCESSFLUSH 13,
73
OLSSC_SUP_INTCLOCK 17, 65, 102
OLSSC_SUP_INTERRUPT 19, 79
OLSSC_SUP_MAXRETRIGGER 20
OLSSC_SUP_MAXSECHANS 16
OLSSC_SUP_MINRETRIGGER 20
OLSSC_SUP_PAUSE 13, 56
OLSSC_SUP_PLS_HIGH2LOW 18,
109

- OLSSC_SUP_PLS_LOW2HIGH 18, 109
- OLSSC_SUP_POSTMESSAGE 13, 40
- OLSSC_SUP_PROCESSOR 20
- OLSSC_SUP_PROGRAMGAIN 15, 52
- OLSSC_SUP_RANDOM_CGL 15
- OLSSC_SUP_RANGEPERCHANNEL 16, 50
- OLSSC_SUP_RETRIGGER_EXTRA 14, 64
- OLSSC_SUP_RETRIGGER_INTERNAL 14, 63
- OLSSC_SUP_RETRIGGER_SCAN_PER_TRIGGER 14, 62
- OLSSC_SUP_SEQUENTIAL_CGL 15
- OLSSC_SUP_SIMULTANEOUS_CLOCKING 17
- OLSSC_SUP_SIMULTANEOUS_SH 15, 45
- OLSSC_SUP_SIMULTANEOUS_START 13, 110
- OLSSC_SUP_SINGLEENDED 16, 43
- OLSSC_SUP_SINGLEVALUE 13, 54
- OLSSC_SUP_SINGLEVALUE_AUTORANGE 15, 54
- OLSSC_SUP_SOFTTRIG 17, 68
- OLSSC_SUP_SWCAL 20
- OLSSC_SUP_SWRESOLUTION 16, 42
- OLSSC_SUP_SYNCHRONOUS_DIGITALIO 15
- OLSSC_SUP_SYNCHRONOUS_DIGITALIO 48
- OLSSC_SUP_THRESHTRIGNEG 17, 69
- OLSSC_SUP_THRESHTRIGPOS 17, 69
- OLSSC_SUP_TIMEREVENTTRIG 17, 70
- OLSSC_SUP_TRIGSCAN 14, 61
- OLSSC_SUP_WRPMULTIPLE 13, 79
- OLSSC_SUP_WRPSINGLE 13, 79
- OLSSC_SUP_ZEROSEQUENTIAL_CGL 15, 45
- OLSSCE_BASECLOCK 20
- OLSSCE_MAXCLOCKDIVIDER 20, 67, 103
- OLSSCE_MAXTHROUGHPUT 20, 66, 102
- OLSSCE_MINCLOCKDIVIDER 20, 67, 103
- OLSSCE_MINTHROUGHPUT 20, 66, 102
- one-shot mode 96
- online help, launching 4
- operation functions 27
- operation modes, counter/timer 82
 - edge-to-edge measurement 90
 - event counting 82
 - frequency measurement 86
 - one-shot pulse output 96
 - rate generation 93
 - repetitive one-shot pulse output 99
 - up/down counting 84
- operations
 - analog input 41
 - analog output 41
 - counter/timer 81
 - digital input 41
 - digital output 41
 - simultaneous 110
- outputting pulses
 - continuously 93
 - one-shot 96

repetitive one-shot 99
overview of the DataAcq SDK 2

P

parameters
 setting up buffers for A/D operations 138
 setting up buffers for D/A operations 139
 setting up buffers for digital input operations 138
 setting up buffers for digital output operations 139
 setting up channel lists and channel parameters 134
 setting up clocks 135
 setting up clocks for counter/timer operations 145
 setting up gates 145
 setting up pre-triggers 135
 setting up the subsystem 133
 setting up triggered scans 137
 setting up triggers 135
pausing an operation
 A/D 56
 D/A 56
 digital input 56
 digital output 56
 edge-to-edge measurement 91
 event counting 83
 rate generation 94
 repetitive one-shot 100
 up/down counting 85
post-trigger mode 57
pre-trigger mode 58
programmable gain 52

programming flowcharts 113
pseudo-differential channels 44
pulse output 81
 duty cycle 108
 flowcharts 129
 one-shot 96
 output types 108
 rate generation (continuous) 93
 repetitive one-shot 99
pulse train output 93
pulse types
 high-to-low 108
 low-to-high 109
pulse width 109

Q

queues
 done 74
 inprocess 72
 ready 71

R

ranges 50
rate generation mode 93
ready queue 71
releasing the subsystem and driver 40
repetitive one-shot mode 99
resetting an operation
 A/D 55
 D/A 55
 digital input 55
 digital output 55
 edge-to-edge measurement 91
 event counting 83
 rate generation 94

- repetitive one-shot 99
 - simultaneous 111
 - up/down counting 85
- resolution 42
- retrigger modes
- internal retrigger 63
 - retrigger extra 64
 - scan-per-trigger 62
- ## S
- sample code
- for continuous A/D 159
 - for single-value A/D 154
- scan-per-trigger retrigger mode 62
- service and support procedure 150
- setting up buffers
- for A/D operations 138
 - for D/A operations 139
 - for digital input operations 138
 - for digital output operations 139
- setting up channel lists 134
- setting up channel parameters 134
- setting up clocks 135
- setting up clocks for counter/timer operations 145
- setting up gates 145
- setting up pretriggers 135
- setting up subsystem parameters 133
- setting up triggered scans 137
- setting up triggers 135
- simultaneous operations 110
- flowcharts 131
- single wrap mode 79
- single-ended inputs 43
- single-value operations 54
- flowcharts 115
 - sample code for analog input 154
- software architecture 35
- software gate type 105
- software trigger 68
- specifying a single channel 44
- specifying a subsystem 38
- specifying buffers 71
- specifying channels in the channel list 46
- specifying clock sources 65
- specifying DMA channels 79
- specifying filters 53
- specifying gains 51
- specifying ranges 50
- specifying single-value operations 54
- specifying synchronous digital I/O values in the channel list 48
- specifying the data flow 54
- specifying trigger sources 67
- specifying triggered scan mode 61
- SRL subsystem 38
- starting an operation
- A/D 55
 - D/A 55
 - digital input 55
 - digital output 55
 - edge-to-edge measurement 91
 - event counting 83
 - frequency measurement 86
 - one-shot 97
 - rate generation 94
 - repetitive one-shot 99
 - up/down counting 85
- stopping an operation
- A/D 55
 - D/A 55
 - digital input 55

- digital output 55
- edge-to-edge measurement 91
- event counting 83
- flowchart 146
- rate generation 94
- repetitive one-shot 99
- simultaneous 111
- up/down counting 85
- subsystem 38
 - setting up parameters 133
- system operations 37
 - configuring a subsystem 39
 - handling errors 40
 - handling messages 40
 - initializing a device 37
 - releasing the subsystem and driver 40
 - specifying a subsystem 38

T

- technical support 150
- terminating the session 40
- termination functions 21
- threshold (negative) trigger 69
- threshold (positive) trigger 69
- timer event trigger 70
- transferring data from an inprocess
 - buffer 142
- trigger source 67
 - analog event 70
 - digital event trigger 70
 - external analog threshold (negative) 69
 - external analog threshold (positive) 69
 - external digital (TTL) 68

- extra trigger 70
 - software (internal) 68
 - timer event trigger 70
- triggered scan mode 61
- troubleshooting checklist 150
- troubleshooting procedure 150
- twos complement data format 41
- type, channel 43

U

- up/down counting operations 84
 - flowcharts 123
- using the online help 4

W

- wrap modes 78
 - multiple 78
 - none 78
 - single 79

Specifications are subject to change without notice.
All Keithley trademarks and trade names are the property of Keithley Instruments, Inc.
All other trademarks and trade names are the property of their respective companies.

KEITHLEY

A G R E A T E R M E A S U R E O F C O N F I D E N C E

Keithley Instruments, Inc.

Corporate Headquarters • 28775 Aurora Road • Cleveland, Ohio 44139 • 440-248-0400 • Fax: 440-248-6168 • 1-888-KEITHLEY (534-8453) • www.keithley.com