DATA TRANSLATION®
A MEASUREMENT COMPUTING COMPANY

UM-22161-U

DT-Open Layers for .NET Class Library User's Manual

Trademark and Copyright Information

Table of Contents

About this Manual

This manual describes how to get started using the DT-Open Layers for .NET Class Library 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® Visual Studio .NET 2003 to 2012 to perform data acquisition operations.

It is assumed that you are a proficient programmer in Visual C# or Visual Basic .NET, 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, summarizes the classes and members of the DT-Open Layers for .NET Class Library, and describes how to develop a data acquisition program using these classes. Using this manual, you should be able to successfully install the DT-Open Layers for .NET Class Library and get started writing a data acquisition application.

This manual is intended to be used with the online help for the DT-Open Layers for .NET Class Library, which is an integrated part of the software.

Note: This manual describes only those class members that are specific to the DT-Open Layers for .NET Class Library. Refer to your Microsoft Visual Studio .NET documentation for information about inherited classes and class members.

Organization of this Manual

This manual is organized as follows:

- Chapter 1, "Getting Started," describes what you need to use the DT-Open Layers for .NET Class Library, how to install the software, how to access the online help, and how to use the example programs.
- Chapter 2, "Library Summary," summarizes the classes, delegates, enumerations, and structures provided in the DT-Open Layers for .NET Class Library.
- Chapter 3, "Using the OpenLayers.Base Namespace," describes how to use the OpenLayers.Base namespace to perform data acquisition operations on a DT-Open Layers-compliant device.

- Chapter 4, "Using the OpenLayers.DeviceCollection Namespace," describes how to use the OpenLayers.DeviceCollection namespace to perform data acquisition operations on a DT-Open Layers-compliant device collection.
- Chapter 5, "Programming Flowcharts for the OpenLayers.Base Namespace," provides programming flowcharts for using the properties, methods, and events that are provided in the OpenLayers.Base namespace of the DT-Open Layers for .NET Class Library.
- Chapter 6, "Programming Flowcharts for the OpenLayers.DeviceCollection Namespace," provides programming flowcharts for using the properties, methods, and events that are provided in the OpenLayers.DeviceCollection namespace of the DT-Open Layers for .NET Class Library.
- Chapter 7, "Product Support," describes how to get help if you have trouble using the DT-Open Layers for .NET Class Library.
- Appendix A, "Error Codes and Messages," provides a list of error codes and descriptions that can be returned by the DT-Open Layers for .NET Class Library.
- 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`.
- Methods and properties may be shown to indicate the class to which they belong, as follows: **DeviceMgr.Get** method means that this is the **Get** method of the DeviceMgr class. Similarly, **SubsystemBase.AnalogSubsystem.AnalogInputSubsystem.Start** means that this is the Start method of the AnalogInputSubsystem class, which is derived from the AnalogSubsystem class, which, in turn, is derived from the SubsystemBase class.

Related Information

Refer to the following documentation for more information on using the DT-Open Layers for .NET Class Library:

- DT-Open Layers for .NET Class Library Online Help. This help file is integrated as part of the software. Refer to page 23 for information on how to open this help file.
- Device-specific documentation. These manuals are provided on your Data Acquisition OMNI CD™ CD.
- Microsoft Visual Studio .NET documentation.

Where to Get Help

Should you run into problems installing or using the DT-Open Layers for .NET Class Library, our Technical Support Department is available to provide prompt, technical assistance. Refer to Chapter 7 for more information. If you are outside the U.S. or Canada, call your local distributor; see our web site (www.mccdaq.com) for the name and telephone number of your nearest distributor.

1

Getting Started

What's Included

The following software is provided on the Data Acquisition OMNI CD for programming DT-Open Layers-compliant devices in Visual C# and Visual Basic .NET:

- DT-Open Layers for .NET Class Library – Provides properties, methods, and events for performing data acquisition operations. This library includes the OpenLayers.Base and OpenLayers.DeviceCollection namespaces.
- DT-Display for .NET Control – Provides a control for plotting data at high speed. You can use this control to plot data that was acquired from the DT-Open Layers for .NET Class Library. This control includes the OpenLayers.Controls and OpenLayers.Signals namespaces.

These assemblies are supported under Windows XP (32-bit), Windows Vista (32-bit and 64-bit), Windows 7 (32-bit and 64-bit), and Windows 8 (32-bit and 64-bit).

This document describes the DT-Open Layers for .NET Class Library. For more information on the DT-Display control, refer to the *DT-Display for .NET User's Manual* on the CD.

What is the DT-Open Layers for .NET Class Library

The DT-Open Layers for .NET Class Library is a native .NET set of object-oriented classes for programming Data Translation's data acquisition devices in Visual C# and Visual Basic .NET.

The DT-Open Layers for .NET Class Library allows you to access the capabilities of your device programmatically. The library is fully compatible with the DT-Open Layers™ standard for developing integrated, modular application programs under Windows. Therefore, 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. Any existing code remains unchanged.

Note: This library is not compatible with the DT-Open Layers Software Development Kit (SDK). Therefore, any existing programs written using the SDK must be modified to work with the DT-Open Layers for .NET Class Library.

The list of supported data acquisition devices is constantly expanding. For the most up-to-date information, refer to the Data Translation web site (www.mccdaq.com).

Device Collection Support in Open Layers

Some devices, such as the VIBbox system, are collection of other devices and subsystems that are connected together through the Sync Bus. For devices and subsystems that support expansion through the Sync Bus, you can use the DT Device Collection Manager application to combine the devices to appear as one collection.

Once a device collection is defined, you can use the OpenLayers.DeviceCollection namespace in the DT-Open Layers for .NET Class Library to perform analog input and/or analog output operations on the collection.

Note: Only subsystems that support expansion through the Sync Bus can be added to a collection. For most devices, this applies to the analog input subsystem only. However, some devices, such as the VIBbox and DT9857E module, support expansion of the analog input and analog output subsystems through the Sync Bus. The OpenLayers.DeviceCollection namespace supports only those subsystems that are added to the collection.

What You Need

To use the DT-Open Layers for .NET Class Library, ensure that your system meets the following minimum requirements:

- PC with a Pentium II 450 MHz minimum processor (Pentium II 600 MHz recommended)
- Microsoft® Windows® XP Professional, Windows XP Home Edition (does not support creating Web applications or XML Web servers in .NET Professional), Windows Vista®, Windows 7®, Windows 8, or Windows Server™ 2003
- Minimum RAM requirements depend on the operating system you are using; consult your operating system documentation for details
- Visual Studio .NET 2005 to 2012 and .NET Framework 2.0 to 4.5 for developing Windows 32-bit and 64-bit applications
- CD-ROM or DVD drive
- Super VGA (1024 x 768) or higher resolution display with 256 colors
- Microsoft mouse or compatible pointing device
- One or more of the supported Data Translation data acquisition devices

---

Note: You can deploy applications on the following operating systems: Windows XP Professional, Windows XP Home Edition, Windows Vista, Windows 7, Windows 8, and Windows Server 2003.

Installing the Software

Note: Ensure that you install Microsoft Visual Studio .NET before installing the DT-Open Layers for .NET Class Library.

The DT-Open Layers for .NET Class Library is installed automatically when you install the device driver for your device. Refer to your documentation for your device for more information.

Building Applications Using DT-Open Layers for .NET

When building applications using DT-Open Layers for .NET, you must reference the OpenLayers.Base.dll assembly.

If you are building 32-bit .NET applications (supported in Visual Studio .NET 2005 and higher), these assemblies are located in the following directory:

Program Files\Data Translation\DotNet\ OLClassLib\Framework 2.0 Assemblies

If you are building 64-bit .NET applications (supported in Visual Studio .NET 2005 and later), these assemblies are located in this directory:

Program Files (x86)\Data Translation\DotNet\OLClassLib\Framework 2.0 Assemblies

You can determine how your application will run by configuring the build settings. For example, if you build your application with the "any CPU" build setting, your application will run as a 32-bit application on 32-bit systems or as a 64-bit application on 64-bit systems.

Using the Online Help

The online help for the DT-Open Layers for .NET Class Library is an integrated part of the software.

You can access the help file in one of the following ways:

- From the Task Bar, select **Start | Programs | Data Translation, Inc | DT-Open Layers for .NET | DT-Open Layers Class Library | DT-Open Layers for .NET API Help**.
  The stand-alone HTML help file is displayed. Click on OpenLayers.Base for help on the class library.
- Press F1 on any property or method of the class library for context-sensitive help.

The online help contains all of the specific reference information for each of the properties, methods, events, error codes, and so on, included in the DT-Open Layers for .NET Class Library.

Using the Example Programs

To help you understand more about using the classes included in the DT-Open Layers for .NET Class Library, the example programs, listed in Table 1, are provided.

For detailed information on the example programs, refer to the SamplesHelp help file provided with the DT-Open Layers for .NET Class Library.

Table 1: Example Programs

Example Type	Example Name	Example Description
Analog Input	ReadSingleValueAsVolts	Uses single-value mode to acquire a single value from an analog input channel and return the data in voltage, given the physical channel and input signal gain.
	ReadSingleValueAsRaw	Uses single-value mode to acquire a single value from an analog input channel and return the data in raw counts, given the physical channel and input signal gain.
	ReadSingleValueAsSensor	Uses single-value mode to acquire a single value from an analog input channel and return the data as a sensor value, given the physical channel number, input signal gain, sensor gain, and sensor offset.
	ReadSingleValueAs Temperature	Uses single-value mode to acquire a single value from an analog input channel and return the data as a temperature value, based on a specified thermocouple or RTD type.
	GetOneBuffer	Configures the analog input subsystem for a sensor, and uses an internal clock to acquire one buffer of data from the specified analog input channel. The data is returned as sensor values.
	ReadBufferedDataAsRaw	Uses continuous (post-trigger) mode and an internal clock to acquire multiple samples from an analog input channel and return the data in raw counts. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).
	ReadBufferedDataAsRaw DigTrigger	Uses continuous (post-trigger) mode, an internal clock, and either a software or external digital trigger to acquire multiple samples from an analog input channel and return the data in raw counts. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).
	ReadBufferedDataAsVolts	Uses continuous (post-trigger) mode and an internal clock to acquire multiple samples from an analog input channel and return the data in voltage. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).

Table 1: Example Programs (cont.)

Example Type	Example Name	Example Description
Analog Input (cont.)	ReadBufferedDataAsVolts SimStart	Uses continuous (post-trigger) mode, a simultaneous start list, and an internal clock to acquire multiple samples from analog input subsystems 0 and 1 simultaneously, and returns the data in voltage. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).
	ReadBufferedDataAsSensor	Uses continuous (post-trigger) mode and an internal clock to acquire multiple samples from an analog input channel and return the data as sensor values based on the specified sensor gain and offset. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).
	ReadBufferedDataAs Temperature	Uses continuous (post-trigger) mode and an internal clock to acquire multiple samples from an RTD or thermocouple input that is connected to an analog input channel, and then returns the data in temperature based on the specified RTD type or thermocouple type and CJC source. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).
	ReadBufferedIepeDataAsRaw	Uses continuous (post-trigger) mode and an internal clock to acquire multiple samples from an IEPE input that is connected to an analog input channel, and then returns the data in raw counts. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).
	ReadBufferedDataInto DtDisplay	Uses continuous (post-trigger) mode and an internal clock to acquire samples from an analog input channel continuously. When each buffer is completed, the data is converted to voltage and plotted to a form using the DT-Display control.
	ReadBufferedDataAsBridge BasedSensor	Configures the subsystem for bridge-based measurements, and uses continuous (post-trigger) mode and an internal clock to acquire multiple samples from an analog input channel. The data is returned in the engineering units of the sensor. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).
	ReadBufferedDataAsStrain	Configures the subsystem for strain measurements, and uses continuous (post-trigger) mode and an internal clock to acquire multiple samples from an analog input channel. The data is returned in microstrain values. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).

Table 1: Example Programs (cont.)

Example Type	Example Name	Example Description
Analog Input (cont.)	ReadBufferedDataFromMulti Sensor	For devices, such as the DT9829 module, that support multiple sensor types, configures the selected analog input channel for the appropriate sensor type, and uses continuous (post-trigger) mode and an internal clock to acquire multiple samples from an analog input channel. The data is returned in the engineering units for the specified sensor type. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).
Analog Output	WriteSingleValueAsVolts	Writes a single voltage value to a single analog output channel.
	WriteSingleValueAsRaw	Writes a single raw count value to a single analog output channel.
	WriteSingleValuesAsVolts	For subsystems that support simultaneous operations, simultaneously writes single voltage values to four analog output channels.
	WriteSingleValueAsRaw_ ProgRanges	Writes a single value, represented as a raw count, to a single analog output channel, and demonstrates how to specify the voltage range for the subsystem.
	WriteBufferedDataAsVolts	Uses continuous mode and an internal clock to write multiple values, represented as voltages, to the analog output channels. You can output three buffers of data and then stop (finite), or output data from the three buffers continuously (continuous).
Digital Input	ReadSingleValue	Reads a single value from a digital input port.
	InterruptOnChange	Reads data continuously from a digital input port, interrupting when a value of a digital input line changes state. The current value of the digital input port and the digital input lines that changed state are displayed.
Digital Output	WriteSingleValue	Writes a single value to a digital output port.
	DT9871TempNET	Supported on the DT9871 instrument only, configures the TEMPpoint instrument, acquires and displays data from up to 48 RTD or thermocouple input channels, optionally logs the acquired data to disk, reads the value of the digital input port, and updates the value of the digital output port. This application also allows you to set minimum and maximum threshold values for the analog input channels, and update the value of a digital output line when the threshold condition occurs.

Table 1: Example Programs (cont.)

Example Type	Example Name	Example Description
Counter/ Timer	EventCounting	Demonstrates how to use event counting and up/down counting mode to count events from an external clock connected to a counter/timer.
	MeasureEdgeToEdge	Uses edge-to-edge measurement mode to measures the time interval between a specified start edge and a specified stop edge of a gate or clock signal connected to a counter/timer.
	PulseOut_RateGeneration	Demonstrates how to use rate generation mode, one-shot mode, and repetitive one-shot mode to generate pulse output signals from a counter/timer.
Quadrature Decoder	ReadCounts	Demonstrates how to read the count of a quadrature decoder.
Simultaneous Input	BufferedInputAnalog_Counter	For devices that allow you to stream counter/timer data through the analog input subsystem, acquires continuous values for a specified analog input and counter/timer channel. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).
	BufferedInputAnalog_Digital	For devices that allow you to stream digital input data through the analog input subsystem, acquires continuous values for a specified analog input channel and digital input port. You can acquire data for a specified number of buffers and then stop (finite), or requeue the buffers to acquire data continuously (continuous).
Utilities	ConvertData	Converts a voltage value to a raw count, or converts a raw count to a voltage value, based on the data encoding, voltage range, and resolution of the subsystem.

To open and run these examples, do the following:

1. Start Microsoft Visual Studio .NET.
2. Click **File**, click **Open**, and then click **Project**.
3. Select the example you want to open from C:\Program Files\mccdaq\DotNet\OLClassLib\Examples folder (if you installed the software using the default destination location).
4. From the main menu of Microsoft Visual Studio .NET, click **Build**, and then click **Build Solution** to build the project.

Note: These examples are provided as 32-bit applications. You can rebuild them as 64-bit applications, if desired, by referencing the x64 DT-Open Layers for .NET assembly (OpenLayers.Base.dll) located in Program Files (x86)\Data Translation\DotNet\OLClassLib\Framework 2.0 Assemblies (64-bit). Refer to page 23 for more information.

---

5. To run the example, click **Debug** from the main menu, and then click **Start**.
   The example program is now running.
6. Use the capabilities of the example program to see how it operates.
7. When you are finished using the example program, click **Debug** from the main menu, then click **Stop Debugging**.
8. View the user interface of the example program by clicking the appropriate [Design] tab on the main window.
9. View the source code for the example program by clicking the appropriate tab (such as example.cs) on the main window.
10. Repeat steps 2 through 9 for each example program you want to open, run, and view.

Creating Your Own Program

To create your own application program, do the following:

1. Start Microsoft Visual Studio .NET.
2. Click **File**, click **New**, and then click **Project**.
3. Select the language you want to develop in (Visual Basic Projects or Visual C# Projects), and the template that you want to use (such as, Windows Application).
4. Enter the project name and where you want to save your project, and then click **OK**.
   The design environment is shown.
5. From the Solution Explorer window, right-click **References**, and then click **Add Reference (**or from the Project Menu, select **Add Reference)**.
   The Add Reference dialog box appears.
6. From the .NET tab, click **OpenLayers for .NET**, click **Select**, and then click **OK**.
   The OpenLayers.Base assembly is now referenced in your application.
7. If desired, click **Object Browser** from the main window, double-click **openlayers.base**, and then double-click **OpenLayers.Base**.
   All the classes included in the library are listed.
8. Select any of the classes to see its methods and properties.
9. Press F1 to access the context-sensitive online help for the library.
10. Develop your code, as appropriate, using the example programs and the information in this manual.
11. To build your program, click **Build** from the main menu, and then click **Build Solution**.
12. To run your program, click **Debug** from the main menu, and then click **Start**.
13. To save your program, click **File**, and then click **Save All**.
14. When you are finished, click **File**, and the click **Close Solution** before exiting from Microsoft Visual Studio .NET.

Distributing Your Program

When you distribute your program, ensure that you also distribute the version of the OpenLayers.Base assembly that was used to create your program. In addition, ensure that any Data Translation devices that are used by your program (along with their device drivers) are installed on the target machine.

One of the best ways to distribute your program is to create a Windows installation project that includes all the necessary files required to run your program. To create an installation project, do the following:

1. Open the program (solution) that you want to distribute within Visual Studio .NET.
2. From the File menu, select **Add Project**, and then select **New Project**.
   The Add New Project dialog box appears.
3. Select **Setup and Deployment Projects**, and select the **Setup Wizard** template.
4. Specify a name, such as **Setup**, for the installation project, specify the location for the installation project on your development system, and then click **OK**.
   The Welcome screen of the Setup wizard appears.
5. Click **Next**.
6. Select "**Create a setup for a Windows application**," and then click **Next**.
7. Select "**Primary output from <the project to be distributed>**," and then click **Finish**.
   Your program and the files OpenLayers.Base.dll and OpenLayers.Personality.dll are added to the Application Folder of the installation project automatically.
8. From the Solutions Explorer, right click on the installation project, and click **Build**.
9. To test that the setup program works properly, right click on the installation project, click **Install**, and follow the prompts.
10. Verify that your program and the files OpenLayers.Base.dll and OpenLayers.Personality.dll are installed in the directory that was specified by the setup program.
11. Distribute this setup program to your end users.

2

Library Summary

Overview

The DT-Open Layers for .NET Class Library consists of the OpenLayers.Base and OpenLayers.DeviceCollection namespaces. The OpenLayers.Base namespace provides the programming interface for all DT-Open Layers-compatible devices except device collections, which are programmed using the OpenLayers.DeviceCollection namespace.

The following elements comprise each namespace:

- Classes – Symbolic representations of objects. They define the operations that objects can perform using properties, methods, and events. In the DT-Open Layers for .NET Class Library, classes are used to define the I/O operations that can be performed on DT-Open Layers-compliant devices.
- Delegates – Data structures that refer to a static method. In the DT-Open Layers for .NET Class Library, delegates are used to call user-specified methods when specific events occur.
- Enumerations – Value types that associate names with specific values. In the DT-Open Layers for .NET Class Library, enumerations are used to define the values of properties and arguments used in methods.
- Structures – Value types that contain data members and functions like classes, but do not require heap allocation. In the DT-Open Layers for .NET Class Library, a structure is used to return specific information about DT-Open Layers devices.

This chapter summarizes the elements of the OpenLayers.Base and OpenLayers.DeviceCollection namespaces in the DT-Open Layers for .NET Class Library.

OpenLayers.Base Namespace

The OpenLayers.Base namespace provides the programming interface for DT-Open Layers-compatible hardware devices. This is the interface to use for all DT-Open Layers-compatible devices, except those devices that are defined as collections (such as the VIBbox system or a user-defined collection created using the DT Device Collection Manager application).

This section describes the elements of the OpenLayers.Base namespace. Refer to Chapter 3 for more information on how to use the OpenLayers.Base namespace.

Classes

The OpenLayers.Base namespace contains the classes listed in Table 2. Each class contains properties, methods, and/or events that allow you to perform specific operations. This section describes the classes and their members.

Table 2: Classes Included in the OpenLayers.Base Namespace

Operation Type	Class Name	Description
Device Management	DeviceMgr	Manages DT-Open Layers devices in the system and assigns Device objects.
	Device	Encapsulates an DT-Open Layers device and manages and distributes subsystems for the device.
	SimultaneousStart	Provides the properties for simultaneously starting multiple subsystems.
Analog Input Operations	AnalogInputSubsystem	Provides the properties, methods, and events for performing analog input operations. This class inherits members from the AnalogSubsystem[a] and SubsystemBase[b] classes.
Analog Output Operations	AnalogOutputSubsystem	Provides the properties, methods, and events for performing analog output operations. This class inherits members from the AnalogSubsystem[a] and SubsystemBase[b] classes.
Digital Input Operations	DigitalInputSubsystem	Provides the properties, methods, and events for performing digital input operations. This class inherits members from the SubsystemBase class[b].
Digital Output Operations	DigitalOutputSubsystem	Provides the properties, methods, and events for performing digital output operations. This class inherits members from the SubsystemBase class[b].

Table 2: Classes Included in the OpenLayers.Base Namespace (cont.)

Operation Type	Class Name	Description
Counter/Timer Operations	CounterTimerSubsystem	Provides the properties, methods, and events for performing counter/timer operations. This class inherits members from the SubsystemBase class[b].
Tachometer Operations	TachSubsystem	Provides the properties, methods, and events for performing operations. This class inherits members from the SubsystemBase class[b].
Quadrature Decoder Operations	QuadratureDecoderSubsystem	Provides the properties, methods, and events for performing quadrature decoder operations. This class inherits members from the SubsystemBase class[b].
Channels	SupportedChannelInfo	Contains information that describes a channel that is associated with a specific subsystem.
	SupportedChannels	A collection of SupportedChannelInfo objects.
	ChannelListEntry	Encapsulates a channel entry for the channel list of a specified subsystem.
	ChannelList	Specifies a collection of ChannelListEntry objects for use in a continuous I/O operation.
	StrainGageTeds	Provides the properties for a strain gage sensor that uses TEDS (Transducer Electronic Data Sheet). This class inherits members from the TedsBase class.
	BridgeSensorTeds	Provides the properties for a strain gage sensor that uses TEDS (Transducer Electronic Data Sheet). This class inherits members from the TedsBase class.
Clocks	Clock	Provides an interface for controlling the clock of a subsystem.
Triggers	Trigger	Provides an interface for controlling the trigger of a subsystem. For device that support a start trigger and a reference trigger, this class controls the start trigger.
	ReferenceTrigger	Provides an interface for controlling the reference trigger of a subsystem.
	TriggeredScan	Provides support for scanning the entries in a ChannelList a specified number of times when the device detects a specified retrigger source.
Ranges	Range	Specifies the upper and lower limits of a voltage range for an analog subsystem.

Table 2: Classes Included in the OpenLayers.Base Namespace (cont.)

Operation Type	Class Name	Description
Buffer Management	OlBuffer	Encapsulates a data buffer that is used in a continuous I/O operation.
	BufferQueue	Provides an interface for queuing OlBuffer objects to a device's subsystem for continuous I/O operations.
Event Handling	BufferDoneEventArgs	Contains data related to the event BufferDoneEvent. This class inherits members from the GeneralEventArgs class.[c]
	DriverRunTimeErrorEventArgs	Contains the data related to the event DriverRunTimeErrorEvent. This class inherits members from the GeneralEventArgs class.[c]
	EventDoneEventArgs	Contains the data related to the event EventDoneEvent. This class inherits members from the GeneralEventArgs class.[c]
	InterruptOnChangeEventArgs	Contains the data related to the event InterruptOnChangeEvent. This class inherits members from the GeneralEventArgs class.[c]
	IOCompleteEventArgs	Contains the data related to the event IOCompleteEvent. This class inherits members from the GeneralEventArgs class.[c]
	MeasureDoneEventArgs	Contains the data related to the event MeasureDoneEvent. This class inherits members from the GeneralEventArgs class.[c]
Error Handling	OlException	DT-Open Layers exception class. Exceptions are raised in response to error conditions within the DT-Open Layers for .NET Class Library.
	OlError	Encapsulates an DT-Open Layers error code.
Services	Utility	Provides properties and methods for getting information about assemblies and for converting data from raw counts to voltage and voltage to raw counts.

a. The AnalogSubsystem class provides the common properties, methods, and events for performing analog I/O operations. This is the base class for the analog input and analog output subsystems. This class inherits many of its capabilities from the SubsystemBase class. You cannot instantiate this object.

b. The SubsystemBase class provides the common properties, methods, and events that are inherited by the subsystems. This is the base class for all subsystems; you cannot instantiate this object.

c. The GeneralEventArgs class contains data that is returned by all DT-Open Layers events that are sent to the user.

Device Management

The OpenLayers.Base namespace provides the following classes for managing devices:

- DeviceMgr, described below
- Device, described starting on page 36
- SimultaneousStart, described starting on page 38

DeviceMgr Class

The DeviceMgr class provides methods for managing DT-Open Layers devices in the system and for assigning a Device object to each DT-Open Layers device that you want to use. Table 3 lists the methods in the DeviceMgr class.

Note: This class exposes the Device object.

Table 3: Methods of the DeviceMgr Class

Member Type	Member Name	Description
Methods	Get	Returns a DeviceMgr object.
	GetDevice	Returns a Device object for the specified device.
	GetDeviceNames	Returns a list of all DT-Open Layers-compatible devices plugged into the system.
	HardwareAvailable	Returns True if an DT-Open Layers-compliant device is plugged into the system; otherwise, returns False.

Device Class

The Device class provides a constructor, properties, and methods for encapsulating an DT-Open Layers device and managing and distributing subsystems for the device.

To access a Device object, it is recommended that you use the **DeviceMgr.GetDevice** method. If you prefer, you can also get a Device object using the Device constructor of the Device class.

Note: This class exposes the following objects: SimultaneousStart, AnalogInputSubsystem, AnalogOutputSubsystem, DigitalInputSubsystem, DigitalOutputSubsystem, CounterTimerSubsystem, TachometerSubsystem, and QuadratureDecoderSubsystem.

Table 4 lists the members of the Device class.

Table 4: Members of the Device Class

Member Type	Member Name	Description
Constructor	Device Constructor	Returns a Device object.
Read-Only Properties	BoardModelName	Returns the model name of the device.
	DeviceName	Returns the user-defined name of the device. This name can be modified in the DT-Open Layers Control Panel applet.
	DriverName	Returns the name of the driver for this device.
	DriverVersion	Returns the version of the driver for this device.
	PowerSource	Returns whether the device is powered by internal or external power.
	SupportsInternalAndExternal Power	Returns True if the device is capable of using an internal and external power source; otherwise, returns False.
Properties that Provide Interfaces	SimultaneousStart	Provides an interface to the SimultaneousStart object.
Methods	AnalogInputSubsystem	Returns an AnalogInputSubsystem object.
	AnalogOutputSubsystem	Returns an AnalogOutputSubsystem object.
	DigitalInputSubsystem	Returns a DigitalInputSubsystem object.
	DigitalOutputSubsystem	Returns a DigitalOutputSubsystem object.
	CounterTimerSubsystem	Returns a CounterTimerSubsystem object.
	TachSubsystem	Returns a TachSubsystem object.
	QuadratureDecoderSubsystem	Returns a QuadratureDecoderSubsystem object.
	Dispose	Terminates the connection to the device.
	GetHardwareInfo	Returns hardware specific-information about the current device.
	SetHardwareInfo	Writes hardware specific-information about the current device.
	GetNumSubsystemElements	Returns the number of available subsystem elements for a given subsystem type.
	DiagReadReg	Returns the value of a specified register on the device. This is an advanced method and is not normally used.
	DiagWriteReg	Writes a value to the specified register on the device. This is an advanced method and is not normally used.
	DiagReadCalPot	Returns the value of the specified calibration pot register. This is an advanced method and is not normally used.
	DiagWriteCalPot	Writes to the specified calibration pot. This is an advanced method and is not normally used.

SimultaneousStart Class

The SimultaneousStart class allows you to start multiple subsystems simultaneously using the properties listed in Table 5.

You access the SimultaneousStart object through the Device object.

Table 5: Additional Members of the SimultaneousStart Class

Member Type	Member Name	Description
Methods	AddSubsystem	Adds a subsystem to the list of subsystems to simultaneous start.
	RemoveSubsystem	Removes a subsystem from the list of subsystems to simultaneous start.
	Clear	Removes all subsystems from the simultaneous start list.
	GetSubsystemList	Returns an array of subsystems that are currently on the simultaneous start list.
	PreStart	Simultaneously prestarts all subsystems on the simultaneous start list.
	Start	Simultaneously starts all subsystems on the simultaneous start list.

Subsystem Operations

The following major classes are provided within the OpenLayers.Base namespace for performing subsystem operations:

- AnalogInputSubsystem, described below
- AnalogOutputSubsystem, described starting on page 47
- DigitalInputSubsystem, described starting on page 52
- DigitalOutputSubsystem, described starting on page 56
- CounterTimerSubsystem, described starting on page 60
- TachSubsystem, described starting on page 65
- QuadratureDecoderSubsystem, described starting on page 69

AnalogInputSubsystem Class

The AnalogInputSubsystem class encapsulates all methods, properties, and events that are specific to analog input operations. Table 6 lists the members of the AnalogInputSubsystem class.

To create an instance of this class, use the **Device.AnalogInputSubsystem** method (recommended) or the AnalogInputSubsystem constructor.

Note: This class provides interfaces to the following objects: BufferQueue, ChannelList, Clock, SupportedChannels, Trigger, and TriggeredScan.

This class inherits the members of the AnalogSubsystem and SubsystemBase classes.

Table 6: Members of the AnalogInputSubsystem Class

Member Type	Member Name	Description
Constructor	AnalogInputSubsystem Constructor	Gets an analog input subsystem.
Read/Write Properties	AsynchronousStop	Gets and sets the stop behavior (synchronous or asynchronous) of the subsystem.
	ChannelType	Gets and sets the channel type (SingleEnded or Differential) for the subsystem.
	DataFilterType	For devices, like the TEMPpoint and VOLTpoint instruments, that support programmable filter types, gets and sets the filter type.
	DataFlow	Gets and sets the data flow mode (Continuous, SingleValue, ContinuousPreTrigger ContinuousPrePostTrigger) for the subsystem.
	Encoding	Gets and sets the data encoding (Binary or TwosComplement) for the subsystem.
	ExcitationVoltageSource	Gets and sets the excitation voltage source (internal, external, or disabled) to apply to the subsystem.
	ExcitationVoltageValue	Gets and sets the value of the internal excitation voltage source to apply across the bridge for each channel of the subsystem.
	ReturnCjcTemperaturesInStream	Enables or disables the subsystem from returning CJC values in the data stream.
	StopOnError	Gets and sets the stop-on-error condition (stop if overrun occurs, or continue if overrun occurs) for the subsystem.
	SynchronizationMode	For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, gets and sets the synchronization mode (None, Master, or Slave).
	SynchronousBufferDone	Gets and sets the way Buffer Done events are executed (asynchronously or synchronously).
	TemperatureFIlterType	Deprecated property; replaced with the DataFilterType property.
	VoltageRange	Gets and sets the current voltage range for the subsystem.

Table 6: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (General)	Device	Returns the Device object that is associated with the subsystem.
	Element	Returns the element number of the subsystem.
	FifoSize	Returns the size of the FIFO on the device that is associated with the subsystem.
	IsRunning	Returns True if the subsystem is currently running; otherwise, returns False.
	ReturnsFloats	Returns True if the subsystem returns floating-point values; otherwise, returns False indicating that the subsystem returns integer values.
	State	Returns the current state of the subsystem (Initialized, ConfiguredForSingleValue, ConfiguredForContinuous, PreStarted, Running, Stopping, Aborting, or IoComplete).
	SubsystemType	Returns the subsystem type (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder).
	SupportsAutoCalibrate	Returns True if the subsystem supports self-calibration, where an auto-zero function is performed through software; otherwise, returns False.
	SupportsDataFilters	Returns True if the subsystem supports programmable filter types; otherwise, returns False.
	SupportsSetSingleValues	Returns True if the subsystem supports updating multiple channels simultaneously with a single value (using **SetSingleValuesAsRaw** or **SetSingleValuesAsVolts**); otherwise, returns False.
	SupportsSimultaneousSampleHold	Returns True if the subsystem supports acquisition on all channels simultaneously; otherwise, returns False.
	SupportsSimultaneousStart	Returns True if the subsystem supports starting multiple subsystems simultaneously; otherwise, returns False.
	SupportsSynchronization	Returns True if the subsystem supports synchronization with other devices; otherwise, returns False.

Table 6: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Data flow-related)	SupportsContinuous	Returns True if the subsystem supports continuous data flow mode; otherwise, returns False.
	SupportsContinuousPrePostTrigger	Returns True if the subsystem supports continuous about-trigger data flow mode; otherwise, returns False.
	SupportsContinuousPreTrigger	Returns True if the subsystem supports continuous pre-trigger data flow mode; otherwise, returns False.
	SupportsSingleValue	Returns True if the subsystem supports single-value data flow mode; otherwise, returns False.
	SupportsTriggeredScan	Returns True if the subsystem supports triggered scan operations; otherwise, returns False.
	SupportsWaveformModeOnly	Returns True if the subsystem supports waveform-based operations using the onboard FIFO only; otherwise, returns False. If this property is True, the buffer wrap mode must be set to WrapSingleBuffer. In addition, the buffer size must be less than or equal to the FifoSize.
Read-Only Properties (Channel-related)	MaxDifferentialChannels	Returns the number of differential channels that are supported by the subsystem.
	MaxSingleEndedChannels	Returns the number of single-ended channels that are supported by the subsystem.
	NumberOfChannels	Returns the total number of channels that are supported by the subsystem.
	SupportsChannelListInhibit	Returns True if the subsystem supports inhibition of a ChannelList entry; otherwise, returns False.
	SupportsDifferential	Returns True if the subsystem supports differential channels; otherwise, returns False.
	SupportsSingleEnded	Returns True if the subsystem supports single-ended channels; otherwise, returns False.
Read-Only Properties (Gain-related)	NumberOfSupportedGains	Returns the number of available gains for this subsystem.
	SupportedGains	Returns an array of available gains for the subsystem.
	SupportsProgrammableGain	Returns True if the subsystem supports programmable gain for ChannelListEntry objects; otherwise, returns False.
Read-Only Properties (Range-related)	NumberOfRanges	Returns the number of available voltage ranges for the subsystem.
	SupportedVoltageRanges	Returns an array of available voltage ranges supported by the subsystem.

Table 6: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Resolution-related)	NumberOfResolutions	Returns the number of resolutions that are supported by the subsystem.
	Resolution	Returns the current resolution of the subsystem.
	SupportedResolutions	Returns an array containing the available resolutions that are supported by the subsystem.
	SupportsSoftwareResolution	Returns True if the subsystem supports software programmable resolution; otherwise, returns False.
Read-Only Properties (Data encoding-related)	SupportsBinaryEncoding	Returns True if the subsystem supports Binary encoding; otherwise, returns False.
	SupportsTwosCompEncoding	Returns True if the subsystem supports TwosComplement encoding; otherwise, returns False.
Read-Only Properties (Buffer-related)	QueuedBufferDones	Returns the number of Buffer Done Events queued to be sent when **SynchronousBufferDone** is True.
	SupportsBuffering	Returns True if the subsystem supports continuous acquisition to or from OlBuffer objects; otherwise, returns False.
	SupportsInProcessFlush	Returns True if the subsystem allows you to move data from the current OlBuffer object while it is being filled; otherwise, returns False.
Read-Only Properties (Temperature-related)	SupportsCjcSourceChannel	Returns True if the subsystem provides channels that are used for cold junction compensation (CJC); otherwise, returns False.
	SupportsCjcSourceInternal	Returns True if the subsystem supports a CJC (cold junction compensation) source that is internal to the hardware; otherwise, returns False.
	SupportsInterleavedCjc TemperaturesInStream	(Has meaning only if SupportsTemperatureDataInStream is True.) Returns True if the device can optionally interleave CJC temperature data with A/D data (either voltage or temperature depending on the thermocouple type) in the data stream; otherwise, returns False.
	SupportsRTD	Returns True if the subsystem supports RTD inputs; otherwise, returns False.
	SupportsTemperatureDataInStream	Returns True if the subsystem supports temperature conversions in hardware, returning temperature data in the stream; otherwise, returns False.
	SupportsTemperatureFilters	Deprecated property; replaced by the SupportsDataFilters property, described on page 40.
	SupportsThermistor	Returns True if the subsystem supports thermistor inputs; otherwise, returns False.
	SupportsThermocouple	Returns True if the subsystem supports thermocouple inputs; otherwise, returns False.

Table 6: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Accelerometer-related)	SupportsACCoupling	Returns True if the subsystem supports AC coupling, where the DC offset is removed; otherwise, returns False.
	SupportsDCCoupling	Returns True if the subsystem supports DC coupling, where the DC offset is included; otherwise, returns False.
	SupportedExcitationCurrentValues	Returns an array containing the available values for the internal excitation current source.
	SupportsExternalExcitationCurrent Src	Returns True if the subsystem supports an external excitation current source; otherwise, returns False.
	SupportsInternalExcitationCurrent Src	Returns True if the subsystem supports an internal excitation current source; otherwise, returns False.
Read-Only Properties (Bridge and Strain Gage-related)	MinExcitationVoltageValue	Returns the minimum allowable excitation voltage that is supported by the subsystem if the **ExcitationVoltageSource** property is set to Internal.
	MaxExcitationVoltageValue	Returns the maximum allowable excitation voltage that is supported by the subsystem if the **ExcitationVoltageSource** property is set to Internal.
	SupportsBridge	Returns True if the subsystem supports bridge-based and/or general-purpose bridges; otherwise, returns False.
	SupportsExternalExcitationVoltage Src	Returns True if the subsystem supports an external excitation voltage source; otherwise, returns False.
	SupportsInternalExcitationVoltage Src	Returns True if the subsystem supports an internal excitation voltage source; otherwise, returns False.
	SupportsPerChannelVoltage Excitation	Returns True if the device supports setting the voltage excitation source and/or value per channel; otherwise, returns False if the voltage excitation source/value must be set for the subsystem.
	SupportsShuntCalibration	Returns True if the subsystem supports shunt calibration; otherwise, returns False.
	SupportsStrainGage	Returns True if the subsystem supports strain gage measurements; otherwise, returns False.
Read-Only Property (Current-Related)	SupportsCurrent	Returns True if the subsystem supports current input measurements; otherwise, returns False.
	SupportsCurrentOutput	Returns True if the subsystem supports current outputs; otherwise, returns False.

Table 6: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Property (Resistance-Related)	SupportsInternalExcitationCurrent Src	Returns True if the subsystem supports an internal excitation current source; otherwise, returns False.
	SupportedExcitationCurrentValues	Returns an array containing the available values for the internal excitation current source.
	SupportsExternalExcitationCurrent Src	Returns True if the subsystem supports an external excitation current source; otherwise, returns False.
	SupportsResistance	Returns True if the subsystem can return resistance measurements; otherwise returns False.
Properties that Provide Interfaces	BufferQueue	Provides an interface to a BufferQueue object.
	ChannelList	Provides an interface to a ChannelList object.
	Clock	Provides an interface to a Clock object.
	ReferenceTrigger	Provides an interface to a ReferenceTrigger object.
	SupportedChannels	Provides an interface to a SupportedChannels object.
	Trigger	Provides an interface to a Trigger object.
	TriggeredScan	Provides an interface to the TriggeredScan object.
Methods	Abort	Stops a continuous operation on the subsystem immediately without waiting for the current buffer to be filled.
	AutoCalibrate	Calibrates the subsystem in software, performing an auto-zero function.
	Config	Configures the subsystem based on the current property settings.
	Dispose	Releases the analog input subsystem's connection to the DT-Open Layers device.
	GetOneBuffer	Using continuous acquisition, acquires one buffer of data from the specified channel. This method uses the specified clock frequency, trigger, and so on, for the acquisition. This method is synchronous and returns only when the requested data has been acquired or a calculated timeout value is exceeded.
	GetSingleCjcValueAsTemperature	For subsystems that support thermocouples and the ability to return floating-point values, acquires a single CJC temperature for an input channel and returns the temperature in the units you specify.
	GetSingleCjcValuesAsTemperature	For subsystems that support simultaneous operations, thermocouples, and the ability to return floating-point values, simultaneously acquires a single CJC temperature value for each input channel and returns the temperature values in the units you specify.

Table 6: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	GetSingleValueAsBridgeBased Sensor	For subsystems that support strain gages, acquires a single value from a full-bridge-based transducer and returns the value in the engineering units of the transducer.
	GetSingleValueAsCurrent	For subsystems that support current measurement, acquires a single value from a current channel and returns the value in Amperes.
	GetSingleValueAsNormalizedBridge Output	For subsystems that support bridges, acquires a single value from a general-purpose bridge or bridge-based sensor and returns the value in the volts.
	GetSingleValueAsRaw	Acquires a single value from an input channel and returns it in raw counts.
	GetSingleValueAsResistance	Acquires a single value from a resistance measurement channel and returns the resistance value in ohms.
	GetSingleValueAsSensor	Acquires a single value from an input channel and returns it in the engineering units for the specified sensor.
	GetSingleValueAsStrain	For subsystems that support strain gages, acquires a single value from an input channel and returns the value in microstrain.
	GetSingleValueAsTemperature	Overloaded method. Acquires a single value from a input channel and returns it as a temperature value based on the specified thermocouple or RTD type and temperature units.
	GetSingleValueAsVolts	Acquires a single value from an input channel and returns the data in voltage.
	GetSingleValuesAsCurrent	For subsystems that support current measurement and simultaneous operations, simultaneously acquires a single value from a each current channel and returns the value in Amperes.
	GetSingleValuesAsRaw	For subsystems that support simultaneous operations, simultaneously acquires a single value from each input channel and returns the data in raw counts.
	GetSingleValuesAsSensor	For subsystems that support simultaneous operations, simultaneously acquires a single value from each input channel and returns the values in the engineering units of the specified sensor.
	GetSingleValuesAsTemperature	For subsystems that support simultaneous operations, thermocouples or RTDs, and the ability to return floating-point values, simultaneously acquires a single temperature value from each input channel and returns the data, in the units you specify, as an array of floating-point values.
	GetSingleValuesAsVolts	For subsystems that support simultaneous operations, simultaneously acquires a single value from each input channel and returns the data in voltages.

Table 6: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	MoveFromBufferInprocess	Moves samples from the OlBuffer object that is currently being filled into a new OlBuffer object.
	RawValueToSensor	Overloaded method that converts a raw count to a sensor value in engineering units.
	RawValueToVolts	Overloaded method that converts a raw count into a voltage value.
	Reset	Stops a continuous operation on a subsystem immediately without waiting for the current buffer to be filled, and reinitializes the subsystem to the default configuration.
	Start	Starts a continuous operation on the analog input subsystem.
	Stop	Stops a continuous operation on the analog input subsystem after the current buffer has been filled.
	ToString	Returns a string that describes the analog input subsystem and element.
	VoltsToRawValue	Converts a voltage value into a raw count.
Events	BufferDoneEvent	Occurs when the current OlBuffer object has been filled with post-trigger data, and if the operation is stopped, occurs for each of up to 8 inprocess buffers.
	DeviceRemovedEvent	Occurs when a device is removed from the system.
	DriverRunTimeErrorEventEvent	Occurs when the device driver detects one of the following error conditions during runtime: FifoOverflow, FifoUnderflow, DeviceOverClocked, TriggerError, or DeviceError.
	GeneralFailureEvent	Occurs when a when a general library failure occurs.
	IOCompleteEvent	For analog input operations that use a reference trigger whose trigger type is something other than software (none), occurs when the last post-trigger sample is copied into the user buffer. Devices that do not support a reference trigger will never receive this event for analog input operations.
	PreTriggerBufferDoneEvent	Occurs when the OlBuffer object is filled with pre-trigger data (for an input operation only).
	QueueDoneEvent	Occurs when no OlBuffer objects are available on the queue and the operation stops.
	QueueStoppedEvent	Occurs when a pre- or post-trigger acquisition operation completes or when you stop a continuous analog input operation.

AnalogOutputSubsystem Class

The AnalogOutputSubsystem class encapsulates all methods, properties, and events that are specific to analog output operations. Table 7 lists the members of the AnalogOutputSubsystem class.

To create an instance of this class, use the **Device.AnalogOutputSubsystem** method (recommended) or the AnalogOutputSubsystem constructor.

Note: This class provides interfaces to the following objects: BufferQueue, ChannelList, Clock, SupportedChannels, and Trigger.

This class inherits the members of the AnalogSubsystem and SubsystemBase classes.

Table 7: Members Added with the AnalogOutputSubsystem Class

Member Type	Member Name	Description
Constructor	AnalogOutputSubsystem Constructor	Gets an analog output subsystem.
Read/Write Properties	AsynchronousStop	Gets and sets the stop behavior (synchronous or asynchronous) of the subsystem.
	ChannelType	Gets and sets the channel type (SingleEnded or Differential) for the subsystem.
	DataFlow	Gets and sets the data flow mode (Continuous or SingleValue) for the subsystem.
	Encoding	Gets and sets the data encoding (Binary or TwosComplement) for the subsystem.
	StopOnError	Gets and sets the stop-on-error condition (stop if underrun occurs, or continue if underrun occurs) for the subsystem.
	SynchronizationMode	For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, gets and sets the synchronization mode (None, Master, or Slave).
	SynchronousBufferDone	Gets and sets the way Buffer Done events are executed (asynchronously or synchronously).
	VoltageRange	Gets and sets the current voltage range for the subsystem.
	WrapSingleBuffer	Gets and sets the wrap mode. If True, the device driver continuously reuses the first buffer queued to the subsystem. If False, the device driver uses all the buffers queued to the subsystem (this is the default mode).

Table 7: Members Added with the AnalogOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (General)	Device	Returns the Device object that is associated with the subsystem.
	Element	Returns the element number of the subsystem.
	FifoSize	Returns the size of the FIFO on the device that is associated with the subsystem.
	IsRunning	Returns True if the subsystem is currently running; otherwise, returns False.
	ReturnsFloats	Returns True if the subsystem returns floating-point values; otherwise, returns False indicating that the subsystem returns integer values.
	State	Returns the current state of the subsystem (Initialized, ConfiguredForSingleValue, ConfiguredForContinuous, PreStarted, Running, Stopping, Aborting, or IoComplete).
	SubsystemType	Returns the subsystem type (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder).
	SupportsCurrentOutput	Returns True if the subsystem supports current outputs; otherwise, returns False.
	SupportsMute	Returns True if the subsystem supports the ability to mute and/or unmute the output voltage.
	SupportsSetSingleValues	Returns True if the subsystem supports updating multiple channels simultaneously with a single value (using **SetSingleValuesAsRaw** or **SetSingleValuesAsVolts**); otherwise, returns False.
	SupportsSimultaneousStart	Returns True if the subsystem supports starting multiple subsystems simultaneously; otherwise, returns False.
	SupportsSynchronization	Returns True if the subsystem supports synchronization with other devices; otherwise, returns False.
Read-Only Properties (Data flow-related)	SupportsContinuous	Returns True if the subsystem supports continuous data flow mode; otherwise, returns False.
	SupportsContinuousPrePostTrigger	Returns True if the subsystem supports continuous about-trigger data flow mode; otherwise, returns False.
	SupportsContinuousPreTrigger	Returns True if the subsystem supports continuous pre-trigger data flow mode; otherwise, returns False.
	SupportsSingleValue	Returns True if the subsystem supports single-value data flow mode; otherwise, returns False.

Table 7: Members Added with the AnalogOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Data flow-related, cont.)	SupportsWaveformModeOnly	Returns True if the subsystem supports waveform-based operations using the onboard FIFO only; otherwise, returns False. If this property is True, the buffer wrap mode must be set to WrapSingleBuffer. In addition, the buffer size must be less than or equal to the FifoSize.
Read-Only Properties (Channel-related)	MaxDifferentialChannels	Returns the number of differential channels that are supported by the subsystem.
	MaxSingleEndedChannels	Returns the number of single-ended channels that are supported by the subsystem.
	NumberOfChannels	Returns the total number of channels that are supported by the subsystem.
	SupportsChannelListInhibit	Returns True if the subsystem supports inhibition of a ChannelList entry; otherwise, returns False.
	SupportsDifferential	Returns True if the subsystem supports differential channels; otherwise, returns False.
	SupportsSingleEnded	Returns True if the subsystem supports single-ended channels; otherwise, returns False.
Read-Only Properties (Gain-related)	NumberOfSupportedGains	Returns the number of available gains for this subsystem.
	SupportedGains	Returns an array of available gains for the subsystem.
	SupportsProgrammableGain	Returns True if the subsystem supports programmable gain for ChannelListEntry objects; otherwise, returns False.
Read-Only Properties (Range-related)	NumberOfRanges	Returns the number of available voltage ranges for the subsystem.
	SupportedVoltageRanges	Returns an array of available voltage ranges supported by the subsystem.
Read-Only Properties (Resolution-related)	NumberOfResolutions	Returns the number of resolutions that are supported by the subsystem.
	Resolution	Returns the current resolution of the subsystem.
	SupportedResolutions	Returns an array containing the available resolutions that are supported by the subsystem.
	SupportsSoftwareResolution	Returns True if the subsystem supports software programmable resolution; otherwise, returns False.
Read-Only Properties (Data encoding-related)	SupportsBinaryEncoding	Returns True if the subsystem supports Binary encoding; otherwise, returns False.
	SupportsTwosCompEncoding	Returns True if the subsystem supports TwosComplement encoding; otherwise, returns False.

Table 7: Members Added with the AnalogOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Buffer-related)	QueuedBufferDones	Returns the number of Buffer Done Events queued to be sent when **SynchronousBufferDone** is True.
	SupportsBuffering	Returns True if the subsystem supports continuous acquisition to or from OlBuffer objects; otherwise, returns False.
	SupportsWrapSingle	Returns True if the subsystem supports reusing a single buffer for continuous operations; otherwise, returns False.
Properties that Provide Interfaces	BufferQueue	Provides an interface to a BufferQueue object.
	ChannelList	Provides an interface to a ChannelList object.
	Clock	Provides an interface to a Clock object.
	ReferenceTrigger	Provides an interface to a ReferenceTrigger object.
	SupportedChannels	Provides an interface to a SupportedChannels object.
	Trigger	Provides an interface to a Trigger object.
Methods	Abort	Stops a continuous operation on the subsystem immediately without waiting for the data in current buffer to be output.
	Config	Configures the subsystem based on the current property settings.
	Dispose	Overloaded method that releases the analog output subsystem's connection to the DT-Open Layers device.
	Reset	Stops a continuous operation on a subsystem immediately without waiting for the data in the current buffer to be output, and reinitializes the subsystem to the default configuration.
	Mute	Attenuates the output voltage of the subsystem to 0 V over a hardware-dependent number of samples.
	RawValueToSensor	Overloaded method that converts a raw count to a sensor value in engineering units.
	RawValueToVolts	Overloaded method that converts a raw count into a voltage value.
	SetSingleValueAsRaw	Writes a single raw count to an analog output channel.
	SetSingleValueAsVolts	Writes a single voltage value to an analog output channel.
	SetSingleValuesAsRaw	For subsystems that support simultaneous operations, simultaneously updates the specified analog output channels with a single raw count value. You specify the channels to update and the value to output on each channel.

Table 7: Members Added with the AnalogOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	SetSingleValuesAsVolts	For subsystems that support simultaneous operations, simultaneously updates the specified analog output channels with a single voltage value. You specify the channels to update and the value to output on each channel.
	Start	Starts a continuous operation on the analog output subsystem.
	Stop	Stops a continuous operation on the analog output subsystem after the data in the current buffer has been output.
	ToString	Returns a string that describes the analog output subsystem and element.
	UnMute	If the subsystem is muted, returns the output voltage of the subsystem to its current level over a hardware-dependent number of samples.
	VoltsToRawValue	Converts a voltage value into a raw count.
Events	BufferDoneEvent	Occurs when all the data in the OlBuffer object has been output.
	DeviceRemovedEvent	Occurs when a device is removed from the system.
	DriverRunTimeErrorEventEvent	Occurs when the device driver detects one of the following error conditions during runtime: FifoOverflow, FifoUnderflow, DeviceOverClocked, TriggerError, or DeviceError.
	GeneralFailureEvent	Occurs when a when a general library failure occurs.
	IOCompleteEvent	For analog output operations, occurs when the when the last data point has been output from an analog output channel. In some cases, this event is raised well after the data is transferred from the buffer (and, therefore, well after BufferDoneEvent and QueueDoneEvents occur).
	QueueDoneEvent	Occurs when no OlBuffer objects are available on the queue and the operation stops.
	QueueStoppedEvent	Occurs when a continuous analog output operation is stopped and the queue is emptied.

DigitalInputSubsystem Class

The DigitalInputSubsystem class encapsulates all methods, properties, and events that are specific to digital input operations. Table 8 lists the members of the DigitalInputSubsystem class.

To create an instance of this class, use the **Device.DigitalInputSubsystem** method (recommended) or the DigitalInputSubsystem constructor.

Note: This class provides interfaces to the BufferQueue, ChannelList, Clock, SupportedChannels, and Trigger objects; however, for most DT-Open Layers devices, only SupportedChannels objects are supported for digital input operations.

This class inherits the members of the SubsystemBase class.

Table 8: Members Added with the DigitalInputSubsystem Class

Member Type	Member Name	Description
Constructor	DigitalInputSubsystem Constructor	Gets a digital input subsystem.
Read/Write Properties	AsynchronousStop[a]	Gets and sets the stop behavior (synchronous or asynchronous) of the subsystem.
	ChannelType[a]	Gets and sets the channel type (SingleEnded or Differential) for the subsystem.
	DataFlow	Gets and sets the data flow mode (Continuous or SingleValue) for the subsystem.
	Encoding[a]	Gets and sets the data encoding (Binary or TwosComplement) for the subsystem.
	Resolution	Gets and sets the resolution of the subsystem.
	StopOnError[a]	Gets and sets the stop-on-error condition (stop if overrun or underrun occurs, or continue if overrun or underrun occurs) for the subsystem.
	SynchronizationMode[a]	For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, gets and sets the synchronization mode (None, Master, or Slave).
Read-Only Properties (General)	Device	Returns the Device object that is associated with the subsystem.
	Element	Returns the element number of the subsystem.
	FifoSize	Returns the size of the FIFO on the device that is associated with the subsystem.
	IsRunning	Returns True if the subsystem is currently running; otherwise, returns False.

Table 8: Members Added with the DigitalInputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (General, cont.)	ReturnsFloats	Returns True if the subsystem returns floating-point values; otherwise, returns False indicating that the subsystem returns integer values.
	State	Returns the current state of the subsystem (Initialized, ConfiguredForSingleValue, ConfiguredForContinuous, PreStarted, Running, Stopping, Aborting, or IoComplete).
	SubsystemType	Returns the subsystem type (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder).
	SupportsCurrentOutput	Returns True if the subsystem supports current outputs; otherwise, returns False.
	SupportsSetSingleValues	Returns True if the subsystem supports updating multiple channels simultaneously with a single value (using **SetSingleValuesAsRaw** or **SetSingleValuesAsVolts**); otherwise, returns False.otherwise, returns False.
	SupportsSimultaneousStart	Returns True if the subsystem supports starting multiple subsystems simultaneously; otherwise, returns False.
	SupportsSynchronization	Returns True if the subsystem supports synchronization with other devices; otherwise, returns False.
Read-Only Properties (Data flow-related)	SupportsContinuous	Returns True if the subsystem supports continuous data flow mode; otherwise, returns False.
	SupportsContinuousPrePostTrigger	Returns True if the subsystem supports continuous about-trigger data flow mode; otherwise, returns False.
	SupportsContinuousPreTrigger	Returns True if the subsystem supports continuous pre-trigger data flow mode; otherwise, returns False.
	SupportsInterruptOnChange	Returns True if the subsystem supports interrupt-on-change; otherwise, returns False.
	SupportsSingleValue	Returns True if the subsystem supports single-value data flow mode; otherwise, returns False.
	SupportsWaveformModeOnly	Returns True if the subsystem supports waveform-based operations using the onboard FIFO only; otherwise, returns False. If this property is True, the buffer wrap mode must be set to WrapSingleBuffer. In addition, the buffer size must be less than or equal to the FifoSize.

Table 8: Members Added with the DigitalInputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Channel-related)	MaxDifferentialChannels	Returns the number of differential channels that are supported by the subsystem.
	MaxSingleEndedChannels	Returns the number of single-ended channels that are supported by the subsystem.
	NumberOfChannels	Returns the total number of channels that are supported by the subsystem.
	SupportsChannelListInhibit	Returns True if the subsystem supports inhibition of a ChannelList entry; otherwise, returns False.
	SupportsDifferential	Returns True if the subsystem supports differential channels; otherwise, returns False.
	SupportsProgrammableGain	Returns True if the subsystem supports programmable gain for ChannelListEntry objects; otherwise, returns False.
	SupportsSingleEnded	Returns True if the subsystem supports single-ended channels; otherwise, returns False.
Read-Only Properties (Resolution-related)	NumberOfResolutions	Returns the number of resolutions that are supported by the subsystem.
	SupportedResolutions	Returns an array containing the available resolutions that are supported by the subsystem.
	SupportsSoftwareResolution	Returns True if the subsystem supports software programmable resolution; otherwise, returns False.
Read-Only Properties (Data encoding-related)	SupportsBinaryEncoding	Returns True if the subsystem supports Binary encoding; otherwise, returns False.
	SupportsTwosCompEncoding	Returns True if the subsystem supports TwosComplement encoding; otherwise, returns False.
Read-Only Properties (Buffer-related)	SupportsBuffering	Returns True if the subsystem supports continuous acquisition to or from OlBuffer objects; otherwise, returns False.
Properties that Provide Interfaces	BufferQueue[a]	Provides an interface to a BufferQueue object.
	ChannelList[a]	Provides an interface to a ChannelList object.
	Clock[a]	Provides an interface to a Clock object.
	ReferenceTrigger[a]	Provides an interface to a ReferenceTrigger object.
	SupportedChannels	Provides an interface to a SupportedChannels object.
	Trigger[a]	Provides an interface to a Trigger object.
Methods	Abort	Stops a continuous operation on the subsystem immediately without waiting for the current operation to complete.
	Config	Configures the subsystem based on the current property settings.
	Dispose	Overloaded method that releases the subsystem's connection to the DT-Open Layers device.

Table 8: Members Added with the DigitalInputSubsystem Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	GetSingleValue	Acquires a single value from the digital input subsystem.
	ReadInterruptOnChangeMask	Returns a bit mask that indicates which lines within a digital input port will generate interrupt-on-change events when they change state.
	Reset	Stops a continuous operation on a subsystem immediately without waiting for the current buffer to be completed, and reinitializes the subsystem to the default configuration.
	Start	Starts a continuous operation on the subsystem.
	Stop	Stops a continuous operation on the subsystem.
	ToString	Returns a string that describes the digital input subsystem and element.
	WriteInterruptOnChangeMask	Selects a set of digital input lines to perform interrupt-on-change operations. When any of the specified lines changes state, the event InterruptOnChangeEvent gets raised.
Events	BufferDoneEvent[a]	Occurs when the current OlBuffer object has been filled with post-trigger data, and if the operation is stopped, occurs for each of up to 8 inprocess buffers.
	DeviceRemovedEvent	Occurs when a device is removed from the system.
	GeneralFailureEvent	Occurs when a when a general library failure occurs.
	InterruptOnChangeEvent	Occurs when a digital input bit changes state.
	QueueDoneEvent[a]	Occurs when no OlBuffer objects are available on the queue and the operation stops.
	QueueStoppedEvent[a]	Occurs when a continuous analog I/O operation is stopped.

a. Currently, no DT-Open Layers devices support this property/method for the digital input subsystem; it is provided for future compatibility.

DigitalOutputSubsystem Class

The DigitalOutputSubsystem class encapsulates all methods, properties, and events that are specific to digital output operations. Table 9 lists the members of the DigitalOutputSubsystem class.

To create an instance of this class, use the **Device.DigitalOutputSubsystem** method (recommended) or the DigitalOutputSubsystem constructor.

Note: This class provides interfaces to the BufferQueue, ChannelList, Clock, SupportedChannels, and Trigger objects; for most DT-Open Layers devices, only SupportedChannels objects are supported for digital output operations.

This class inherits the members of the SubsystemBase class.

Table 9: Members Added with the DigitalOutputSubsystem Class

Member Type	Member Name	Description
Constructor	DigitalOutputSubsystem Constructor	Gets a digital output subsystem.
Read/Write Properties	AsynchronousStop[a]	Gets and sets the stop behavior (synchronous or asynchronous) of the subsystem.
	ChannelType[a]	Gets and sets the channel type (SingleEnded or Differential) for the subsystem.
	DataFlow	Gets and sets the data flow mode (Continuous or SingleValue) for the subsystem.
	Encoding[a]	Gets and sets the data encoding (Binary or TwosComplement) for the subsystem.
	Resolution	Gets and sets the resolution of the subsystem.
	StopOnError[a]	Gets and sets the stop-on-error condition (stop if overrun or underrun occurs, or continue if overrun or underrun occurs) for the subsystem.
	SynchronizationMode[a]	For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, gets and sets the synchronization mode (None, Master, or Slave).
Read-Only Properties (General)	Device	Returns the Device object that is associated with the subsystem.
	Element	Returns the element number of the subsystem.
	FifoSize	Returns the size of the FIFO on the device that is associated with the subsystem.
	IsRunning	Returns True if the subsystem is currently running; otherwise, returns False.

Table 9: Members Added with the DigitalOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (General, cont.)	ReturnsFloats	Returns True if the subsystem returns floating-point values; otherwise, returns False indicating that the subsystem returns integer values.
	State	Returns the current state of the subsystem (Initialized, ConfiguredForSingleValue, ConfiguredForContinuous, PreStarted, Running, Stopping, Aborting, or IoComplete).
	SubsystemType	Returns the subsystem type (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder).
	SupportsCurrentOutput	Returns True if the subsystem supports current outputs; otherwise, returns False.
	SupportsSetSingleValues	Returns True if the subsystem supports updating multiple channels simultaneously with a single value (using **SetSingleValuesAsRaw** or **SetSingleValuesAsVolts**); otherwise, returns False.
	SupportsSimultaneousStart	Returns True if the subsystem supports starting multiple subsystems simultaneously; otherwise, returns False.
	SupportsSynchronization	Returns True if the subsystem supports synchronization with other devices; otherwise, returns False.
Read-Only Properties (Data flow-related)	SupportsContinuous	Returns True if the subsystem supports continuous data flow mode; otherwise, returns False.
	SupportsContinuousPrePost Trigger	Returns True if the subsystem supports continuous about-trigger data flow mode; otherwise, returns False.
	SupportsContinuousPreTrigger	Returns True if the subsystem supports continuous pre-trigger data flow mode; otherwise, returns False.
	SupportsSingleValue	Returns True if the subsystem supports single-value data flow mode; otherwise, returns False.
	SupportsWaveformModeOnly	Returns True if the subsystem supports waveform-based operations using the onboard FIFO only; otherwise, returns False. If this property is True, the buffer wrap mode must be set to WrapSingleBuffer. In addition, the buffer size must be less than or equal to the FifoSize.
	MaxDifferentialChannels	Returns the number of differential channels that are supported by the subsystem.
	MaxSingleEndedChannels	Returns the number of single-ended channels that are supported by the subsystem.

Table 9: Members Added with the DigitalOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Channel-related)	NumberOfChannels	Returns the total number of channels that are supported by the subsystem.
	SupportsChannelListInhibit	Returns True if the subsystem supports inhibition of a ChannelList entry; otherwise, returns False.
	SupportsDifferential	Returns True if the subsystem supports differential channels; otherwise, returns False.
	SupportsProgrammableGain	Returns True if the subsystem supports programmable gain for ChannelListEntry objects; otherwise, returns False.
	SupportsSingleEnded	Returns True if the subsystem supports single-ended channels; otherwise, returns False.
Read-Only Properties (Resolution-related)	NumberOfResolutions	Returns the number of resolutions that are supported by the subsystem.
	SupportedResolutions	Returns an array containing the available resolutions that are supported by the subsystem.
	SupportsSoftwareResolution	Returns True if the subsystem supports software programmable resolution; otherwise, returns False.
Read-Only Properties (Data encoding-related)	SupportsBinaryEncoding	Returns True if the subsystem supports Binary encoding; otherwise, returns False.
	SupportsTwosCompEncoding	Returns True if the subsystem supports TwosComplement encoding; otherwise, returns False.
Read-Only Properties (Buffer-related)	SupportsBuffering	Returns True if the subsystem supports continuous acquisition to or from OlBuffer objects; otherwise, returns False.
Properties that Provide Interfaces	BufferQueue[a]	Provides an interface to a BufferQueue object.
	ChannelList[a]	Provides an interface to a ChannelList object.
	Clock[a]	Provides an interface to a Clock object.
	ReferenceTrigger[a]	Provides an interface to a ReferenceTrigger object.
	SupportedChannels	Provides an interface to a SupportedChannels object.
	Trigger[a]	Provides an interface to a Trigger object.

Table 9: Members Added with the DigitalOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Methods	Abort	Stops a continuous operation on the subsystem immediately without waiting for the current operation to complete.
	Config	Configures the subsystem based on the current property settings.
	Dispose	Overloaded method that releases the subsystem's connection to the DT-Open Layers device.
	Reset	Stops a continuous operation on a subsystem immediately without waiting for the current buffer to be completed, and reinitializes the subsystem to the default configuration.
	SetSingleValue	Writes a single output value to the digital output subsystem.
	Start	Starts a continuous operation on the subsystem.
	Stop	Stops a continuous operation on the subsystem.
	ToString	Returns a string that describes the digital output subsystem and element.
Events	BufferDoneEvent[a]	Occurs when the data in the OlBuffer object has been output.
	DeviceRemovedEvent	Occurs when a device is removed from the system.
	GeneralFailureEvent	Occurs when a when a general library failure occurs.
	QueueDoneEvent[a]	Occurs when no OlBuffer objects are available on the queue and the operation stops.
	QueueStoppedEvent[a]	Occurs when a continuous analog I/O operation is stopped.

a. Currently, no DT-Open Layers devices support this property/method for the digital output subsystem; it is provided for future compatibility.

CounterTimerSubsystem Class

The CounterTimerSubsystem class encapsulates all methods, properties, and events that are specific to counter/timer operations. Table 10 lists the members of the CounterTimerSubsystem class.

To create an instance of this class, use the **Device.CounterTimerSubsystem** method (recommended) or the CounterTimerSubsystem constructor.

Note: This class provides interfaces to the BufferQueue, ChannelList, Clock, SupportedChannels, and Trigger objects; for most DT-Open Layers devices, only the Clock and SupportedChannel objects are supported for counter/timer operations.

This class inherits the members of the SubsystemBase class.

Table 10: Members Added with the CounterTimerSubsystem Class

Member Type	Member Name	Description
Constructor	CounterTimerSubsystem Constructor	Gets a counter/timer subsystem.
Read/Write Properties (General)	AsynchronousStop[a]	Gets and sets the stop behavior (synchronous or asynchronous) of the subsystem.
	ChannelType[a]	Gets and sets the channel type (SingleEnded or Differential) for the subsystem.
	DataFlow	Gets and sets the data flow mode (Continuous, SingleValue) for the subsystem.
	Encoding[a]	Gets and sets the data encoding (Binary or TwosComplement) for the subsystem.
	Resolution	Gets and sets the resolution of the subsystem.
	StopOnError[a]	Gets and sets the stop-on-error condition (stop if overrun or underrun occurs, or continue if overrun or underrun occurs) for the subsystem.
	SynchronizationMode[a]	For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, gets and sets the synchronization mode (None, Master, or Slave).

Table 10: Members Added with the CounterTimerSubsystem Class (cont.)

Member Type	Member Name	Description
Read/Write Properties (C/T-related)	CascadeMode	Gets and sets the cascade mode (Cascade or Single) for the subsystem.
	CounterMode	Gets and sets the counter/timer mode (Count, RateGenerator, OneShot, OneShotRepeat, UpDown, Measure, or ContinuousMeasure) for the subsystem.
	GateType	Gets and sets the gate type (None, HighLevel, LowLevel, HighEdge, LowEdge, or Level) for the subsystem.
	PulseType	Gets and sets the pulse type (HighToLow or LowToHigh) for the subsystem.
	PulseWidth	Gets and sets the width of the output pulse for the subsystem.
	StartEdge	Gets and sets the start edge (GateRising, GateFalling, ClockRising, or ClockFalling) for a Measure or ContinuousMeasure operation on the subsystem.
	StopEdge	Gets and sets the stop edge (GateRising, GateFalling, ClockRising, or ClockFalling) for an edge-to-edge measurement operation on the subsystem.
Read-Only Properties (General)	Device	Returns the Device object that is associated with the subsystem.
	Element	Returns the element number of the subsystem.
	FifoSize	Returns the size of the FIFO on the device that is associated with the subsystem.
	IsRunning	Returns True if the subsystem is currently running; otherwise, returns False.
	ReturnsFloats	Returns True if the subsystem returns floating-point values; otherwise, returns False indicating that the subsystem returns integer values.
	State	Returns the current state of the subsystem (Initialized, ConfiguredForSingleValue, ConfiguredForContinuous, PreStarted, Running, Stopping, Aborting, or IoComplete).
	SubsystemType	Returns the subsystem type (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder).
	SupportsCurrentOutput	Returns True if the subsystem supports current outputs; otherwise, returns False.
	SupportsSetSingleValues	Returns True if the subsystem supports updating multiple channels simultaneously with a single value (using **SetSingleValuesAsRaw** or **SetSingleValuesAsVolts**); otherwise, returns False.
	SupportsSimultaneousStart	Returns True if the subsystem supports starting multiple subsystems simultaneously; otherwise, returns False.
	SupportsSynchronization	Returns True if the subsystem supports synchronization with other devices; otherwise, returns False.

Table 10: Members Added with the CounterTimerSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Counter-mode related)	SupportsCascading	Returns True if the subsystem supports cascading of counter/timers; otherwise, returns False.
	SupportsContinuousMeasure	Returns True if the counter/timer subsystem supports continuous edge-to-edge measurement operations; otherwise, returns False.
	SupportsCount	Returns True if the counter/timer subsystem supports event counting operations; otherwise, returns False.
	SupportsMeasure	Returns True if the counter/timer subsystem supports edge-to-edge measurement mode; otherwise, returns False.
	SupportsOneShot	Returns True if the counter/timer subsystem supports one-shot operations; otherwise, returns False.
	SupportsOneShotRepeat	Returns True if the counter/timer subsystem supports repetitive one-shot operations; otherwise, returns False.
	SupportsRateGenerate	Returns True if the counter/timer subsystem supports continuous pulse output (rate generation) operations; otherwise, returns False.
	SupportsUpDown	Returns True if the counter/timer subsystem supports up/down counting operations; otherwise, returns False.
Read-Only Properties (Edge-related)	SupportedEdgeTypes	Returns an array containing the available edge types that are supported by the subsystem.
	SupportsClockFalling	Returns True if the falling edge of the clock signal can be used in an edge-to-edge measurement operation; otherwise, returns False.
	SupportsClockRising	Returns True if the rising edge of the clock signal can be used in an edge-to-edge measurement operation; otherwise, returns False.
	SupportsGateFalling	Returns True if the falling edge of the gate signal can be used in a continuous edge-to-edge measurement operation.
	SupportsGateRising	Returns True if the rising edge of the gate signal can be used in an edge-to-edge measurement operation; otherwise, returns False.
Read-Only Properties (Gate-related)	SupportsGateHighEdge	Returns True if the counter/timer subsystem supports a HighEdge gate type; otherwise, returns False.
	SupportsGateHighLevel	Returns True if the counter/timer subsystem supports a HighLevel gate type; otherwise, returns False.
	SupportsGateLevel	Returns True if the counter/timer subsystem supports a Level gate type; otherwise, returns False.
	SupportsGateLowEdge	Returns True if the counter/timer subsystem supports a LowEdge gate type; otherwise, returns False.
	SupportsGateLowLevel	Returns True if the counter/timer subsystem supports a LowLevel gate type; otherwise, returns False.

Table 10: Members Added with the CounterTimerSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Gate-related, cont.)	SupportsGateNone	Returns True if the counter/timer subsystem supports a software (None) gate type; otherwise, returns False.
Read-Only Properties (Pulse output-related)	SupportsHighToLowPulse	Returns True if the counter/timer subsystem supports high-to-low pulse output types; otherwise, returns False.
	SupportsLowToHighPulse	Returns True if the counter/timer subsystem supports low-to-high pulse output types; otherwise, returns False.
	SupportsVariablePulseWidth	Returns True if the counter/timer subsystem supports programmable pulse widths; otherwise, returns False.
Read-Only Properties (Data flow-related)	SupportsContinuous	Returns True if the subsystem supports continuous data flow mode; otherwise, returns False.
	SupportsContinuousPrePost Trigger	Returns True if the subsystem supports continuous about-trigger data flow mode; otherwise, returns False.
	SupportsContinuousPreTrigger	Returns True if the subsystem supports continuous pre-trigger data flow mode; otherwise, returns False.
	SupportsInterrupt	Returns True if the subsystem supports interrrupt-driven I/O; otherwise, returns False.
	SupportsSingleValue	Returns True if the subsystem supports single-value data flow mode; otherwise, returns False.
	SupportsWaveformModeOnly	Returns True if the subsystem supports waveform-based operations using the onboard FIFO only; otherwise, returns False. If this property is True, the buffer wrap mode must be set to WrapSingleBuffer. In addition, the buffer size must be less than or equal to the FifoSize.
Read-Only Properties (Channel-related)	MaxDifferentialChannels	Returns the number of differential channels that are supported by the subsystem.
	MaxSingleEndedChannels	Returns the number of single-ended channels that are supported by the subsystem.
	NumberOfChannels	Returns the total number of channels that are supported by the subsystem.
	SupportsChannelListInhibit	Returns True if the subsystem supports inhibition of a ChannelList entry; otherwise, returns False.
	SupportsDifferential	Returns True if the subsystem supports differential channels; otherwise, returns False.
	SupportsProgrammableGain	Returns True if the subsystem supports programmable gain for ChannelListEntry objects; otherwise, returns False.
	SupportsSingleEnded	Returns True if the subsystem supports single-ended channels; otherwise, returns False.

Table 10: Members Added with the CounterTimerSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Resolution-related)	NumberOfResolutions	Returns the number of resolutions that are supported by the subsystem.
	SupportedResolutions	Returns an array containing the available resolutions that are supported by the subsystem.
	SupportsSoftwareResolution	Returns True if the subsystem supports software programmable resolution; otherwise, returns False.
Read-Only Properties (Data encoding-related)	SupportsBinaryEncoding	Returns True if the subsystem supports Binary encoding; otherwise, returns False.
	SupportsTwosCompEncoding	Returns True if the subsystem supports TwosComplement encoding; otherwise, returns False.
Read-Only Properties (Buffer-related)	SupportsBuffering	Returns True if the subsystem supports continuous acquisition to or from OlBuffer objects; otherwise, returns False.
Properties that Provide Interfaces	BufferQueue[a]	Provides an interface to a BufferQueue object.
	ChannelList[a]	Provides an interface to a ChannelList object.
	Clock	Provides an interface to a Clock object.
	ReferenceTrigger[a]	Provides an interface to a ReferenceTrigger object.
	SupportedChannels	Provides an interface to a SupportedChannels object.
	Trigger[a]	Provides an interface to a Trigger object.
Methods	Abort	Stops a continuous operation on the counter/timer subsystem. For this subsystem type, behaves like Stop.
	Config	Configures the subsystem based on the current property settings.
	Dispose	Overloaded method that releases the counter/timer subsystem's connection to the DT-Open Layers device.
	ReadCount	Returns the current count of a counter/timer subsystem. This call is typically meaningful only for counter/timer subsystems that are set up for event counting, up/down counting, or continuous measure mode.
	Reset	Stops a continuous operation on a subsystem immediately without waiting for the current buffer to be completed, and reinitializes the subsystem to the default configuration.
	Start	Starts an operation on the counter/timer subsystem.
	Stop	Stops a continuous operation on the counter/timer subsystem.
	ToString	Returns a string that describes the counter/timer subsystem and element.

Table 10: Members Added with the CounterTimerSubsystem Class (cont.)

Member Type	Member Name	Description
Events	BufferDoneEvent[a]	Occurs when the current OlBuffer object has been filled with post-trigger data, and if the operation is stopped, occurs for each of up to 8 inprocess buffers. For output operations, occurs when the data in the OlBuffer object has been output.
	DeviceRemovedEvent	Occurs when a device is removed from the system.
	EventDoneEvent	Occurs on some devices, such as the DT340, when a digital input line changes state or when an interval timer operation is complete.
	GeneralFailureEvent	Occurs when a when a general library failure occurs.
	MeasureDoneEvent	Occurs when an edge-to-edge measurement (Measure) operation is complete.
	QueueDoneEvent[a]	Occurs when no OlBuffer objects are available on the queue and the operation stops.
	QueueStoppedEvent[a]	Occurs when a continuous analog I/O operation is stopped.

a. Currently, no DT-Open Layers devices support this property/method for the counter/timer subsystem; it is provided for future compatibility.

TachSubsystem Class

The TachSubsystem class encapsulates all methods, properties, and events that are specific to tachometer operations. Table 10 lists the members of the TachSubsystem class.

To create an instance of this class, use the **Device.TachSubsystem** method (recommended) or the TachSubsystem constructor.

Note: This class provides interfaces to the BufferQueue, ChannelList, Clock, SupportedChannels, and Trigger objects; for most DT-Open Layers devices, these objects are not supported for tachometer operations.

This class inherits the members of the SubsystemBase class.

Table 11: Members Added with the TachSubsystem Class

Member Type	Member Name	Description
Constructor	TachSubsystem Constructor	Gets a tachometer subsystem.
Read/Write Properties (General)	AsynchronousStop[a]	Gets and sets the stop behavior (synchronous or asynchronous) of the subsystem.
	ChannelType[a]	Gets and sets the channel type (SingleEnded or Differential) for the subsystem.
	DataFlow[a]	Gets and sets the data flow mode (Continuous, SingleValue, ContinuousPreTrigger, ContinuousPrePostTrigger) for the subsystem.
	Encoding[a]	Gets and sets the data encoding (Binary or TwosComplement) for the subsystem.
	Resolution	Gets and sets the resolution of the subsystem.
	StopOnError[a]	Gets and sets the stop-on-error condition (stop if overrun or underrun occurs, or continue if overrun or underrun occurs) for the subsystem.
	SynchronizationMode[a]	For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, gets and sets the synchronization mode (None, Master, or Slave).
Read/Write Properties (Tach-related)	EdgeType	Gets and sets the edge type (Falling or Rising) for the tachometer subsystem.
	StaleDataFlagEnabled	Gets and sets the flag indicating whether or not the value of the tachometer is new. If StaleDataFlagEnabled is True, the most significant bit (MSB) of the value is set to 0 to indicate new data; reading the value before the measurement is complete returns an MSB of 1. If the StaleDataFlagEnabled is False, the MSB is always set to 0.
Read-Only Properties (General)	Device	Returns the Device object that is associated with the tachometer subsystem.
	Element	Returns the element number of the subsystem.
	FifoSize	Returns the size of the FIFO on the device that is associated with the subsystem.
	IsRunning	Returns True if the subsystem is currently running; otherwise, returns False.
	ReturnsFloats	Returns True if the subsystem returns floating-point values; otherwise, returns False indicating that the subsystem returns integer values.
	State	Returns the current state of the subsystem (Initialized, ConfiguredForSingleValue, ConfiguredForContinuous, PreStarted, Running, Stopping, Aborting, or IoComplete).

Table 11: Members Added with the TachSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (General, cont.)	SubsystemType	Returns the subsystem type (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder).
	SupportsCurrentOutput	Returns True if the subsystem supports current outputs; otherwise, returns False.
	SupportsSetSingleValues	Returns True if the subsystem supports updating multiple channels simultaneously with a single value (using **SetSingleValuesAsRaw** or **SetSingleValuesAsVolts**); otherwise, returns False.
	SupportsSimultaneousStart	Returns True if the subsystem supports starting multiple subsystems simultaneously; otherwise, returns False.
	SupportsSynchronization	Returns True if the subsystem supports synchronization with other devices; otherwise, returns False.
Read-Only Properties (Tachometer-related)	Count	Returns the current count between two consecutive edges (rising to rising or falling to falling) of the tachometer signal.
	SupportsFallingEdge	Returns True if the subsystem supports falling edges of the tachometer signal; otherwise, returns False.
	SupportsRisingEdge	Returns True if the subsystem supports rising edges of the tachometer signal; otherwise, returns False.
	SupportsStaleDataFlag	Returns True if the subsystem supports the Stale Data flag.
Read-Only Properties (Data flow-related)	SupportsContinuous	Returns True if the subsystem supports continuous data flow mode; otherwise, returns False.
	SupportsContinuousPrePost Trigger	Returns True if the subsystem supports continuous about-trigger data flow mode; otherwise, returns False.
	SupportsContinuousPreTrigger	Returns True if the subsystem supports continuous pre-trigger data flow mode; otherwise, returns False.
	SupportsSingleValue	Returns True if the subsystem supports single-value data flow mode; otherwise, returns False.
	SupportsWaveformModeOnly	Returns True if the subsystem supports waveform-based operations using the onboard FIFO only; otherwise, returns False. If this property is True, the buffer wrap mode must be set to WrapSingleBuffer. In addition, the buffer size must be less than or equal to the FifoSize.
Read-Only Properties (Channel-related)	MaxDifferentialChannels	Returns the number of differential channels that are supported by the subsystem.
	MaxSingleEndedChannels	Returns the number of single-ended channels that are supported by the subsystem.
	NumberOfChannels	Returns the total number of channels that are supported by the subsystem.
	SupportsChannelListInhibit	Returns True if the subsystem supports inhibition of a ChannelList entry; otherwise, returns False.

Table 11: Members Added with the TachSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Channel-related, cont.)	SupportsDifferential	Returns True if the subsystem supports differential channels; otherwise, returns False.
	SupportsProgrammableGain	Returns True if the subsystem supports programmable gain for ChannelListEntry objects; otherwise, returns False.
	SupportsSingleEnded	Returns True if the subsystem supports single-ended channels; otherwise, returns False.
Read-Only Properties (Resolution-related)	NumberOfResolutions	Returns the number of resolutions that are supported by the subsystem.
	SupportsSoftwareResolution	Returns True if the subsystem supports software programmable resolution; otherwise, returns False.
	SupportedResolutions	Returns an array containing the available resolutions that are supported by the subsystem.
Read-Only Properties (Data encoding-related)	SupportsBinaryEncoding	Returns True if the subsystem supports Binary encoding; otherwise, returns False.
	SupportsTwosCompEncoding	Returns True if the subsystem supports TwosComplement encoding; otherwise, returns False.
Read-Only Properties (Buffer-related)	SupportsBuffering	Returns True if the subsystem supports continuous acquisition to or from OlBuffer objects; otherwise, returns False.
Properties that Provide Interfaces	BufferQueue[a]	Provides an interface to a BufferQueue object.
	ChannelList[a]	Provides an interface to a ChannelList object.
	Clock[a]	Provides an interface to a Clock object.
	ReferenceTrigger[a]	Provides an interface to a ReferenceTrigger object.
	SupportedChannels[a]	Provides an interface to a SupportedChannels object.
	Trigger[a]	Provides an interface to a Trigger object.
Methods	Abort	Stops a continuous operation on the subsystem. For this subsystem type, behaves like Stop.
	Config	Configures the subsystem based on the current property settings.
	Dispose	Overloaded method that releases the subsystem's connection to the DT-Open Layers device.
	Reset	Stops a continuous operation on a subsystem immediately without waiting for the current buffer to be completed, and reinitializes the subsystem to the default configuration.
	Start	Starts an operation on the subsystem.
	Stop	Stops a continuous operation on the subsystem.
	ToString	Returns a string that describes the subsystem and element.

Table 11: Members Added with the TachSubsystem Class (cont.)

Member Type	Member Name	Description
Events	BufferDoneEvent[a]	For input operations, occurs when the OlBuffer object has been filled with post-trigger data. For output operations, occurs when all the data in the OlBuffer object has been output. If you stop an analog I/O operation, the event BufferDoneEvent is generated for the current buffer and for up to eight inprocess buffers before a QueueStoppedEvent event occurs.
	DeviceRemovedEvent	Occurs when a device is removed from the system.
	GeneralFailureEvent	Occurs when a when a general library failure occurs.
	QueueDoneEvent[a]	Occurs when no OlBuffer objects are available on the queue and the operation stops.
	QueueStoppedEvent[a]	Occurs when a continuous analog I/O operation is stopped.

a. Currently, no DT-Open Layers devices support this property/method for the tachometer subsystem; it is provided for future compatibility.

QuadratureDecoderSubsystem Class

The QuadratureDecoderSubsystem class encapsulates all methods, properties, and events that are specific to quadrature decoder operations. Table 12 lists the members of the QuadratureDecoderSubsystem class.

To create an instance of this class, use the **Device.QuadratureDecoderSubsystem** method (recommended) or the QuadratureDecoderSubsystem constructor.

Note: This class provides interfaces to the BufferQueue, ChannelList, Clock, SupportedChannels, and Trigger objects. For most DT-Open Layers devices, only SupportedChannel objects are supported for quadrature decoder operations.

This class inherits the members of the SubsystemBase class.

Table 12: Members Added with the QuadratureDecoderSubsystem Class

Member Type	Member Name	Description
Constructor	QuadratureDecoder Subsystem Constructor	Gets a quadrature decoder subsystem.
Read/Write Properties (General)	AsynchronousStop	Gets and sets the stop behavior (synchronous or asynchronous) of the subsystem.
	ChannelType[a]	Gets and sets the channel type (SingleEnded or Differential) for the subsystem.

Table 12: Members Added with the QuadratureDecoderSubsystem Class (cont.)

Member Type	Member Name	Description
Read/Write Properties (General, cont.)	DataFlow[a]	Gets and sets the data flow mode (Continuous or SingleValue) for the subsystem.
	Encoding[a]	Gets and sets the data encoding (Binary or TwosComplement) for the subsystem.
	Resolution[a]	Gets and sets the resolution of the subsystem.
	StopOnError[a]	Gets and sets the stop-on-error condition (stop if overrun or underrun occurs, or continue if overrun or underrun occurs) for the subsystem.
	SynchronizationMode[a]	For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, gets and sets the synchronization mode (None, Master, or Slave).
Read/Write Properties (Quadrature Decoder -related)	ClockPreScale	Gets and sets the clock prescale value for the quadrature decoder subsystem.
	IndexMode	Gets and sets the index mode (Disabled, Low, High) for the quadrature decoder subsystem.
	X4Scaling	Gets and sets the quadrature decoder scaling mode (X1 or X4).
Read-Only Properties (General)	Device	Returns the Device object that is associated with the subsystem.
	Element	Returns the element number of the subsystem.
	FifoSize	Returns the size of the FIFO on the device that is associated with the subsystem.
	IsRunning	Returns True if the subsystem is currently running; otherwise, returns False.
	ReturnsFloats	Returns True if the subsystem returns floating-point values; otherwise, returns False indicating that the subsystem returns integer values.
	State	Returns the current state of the subsystem (Initialized, ConfiguredForSingleValue, ConfiguredForContinuous, PreStarted, Running, Stopping, Aborting, or IoComplete).
	SubsystemType	Returns the subsystem type (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder).
	SupportsCurrentOutput	Returns True if the subsystem supports current outputs; otherwise, returns False.
	SupportsSetSingleValues	Returns True if the subsystem supports updating multiple channels simultaneously with a single value (using **SetSingleValuesAsRaw** or **SetSingleValuesAsVolts**); otherwise, returns False.
	SupportsSimultaneousStart	Returns True if the subsystem supports starting multiple subsystems simultaneously; otherwise, returns False.

Table 12: Members Added with the QuadratureDecoderSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (General, cont.)	SupportsSynchronization	Returns True if the subsystem supports synchronization with other devices; otherwise, returns False.
Read-Only Properties (Data flow-related)	SupportsContinuous	Returns True if the subsystem supports continuous data flow mode; otherwise, returns False.
	SupportsContinuousPrePost Trigger	Returns True if the subsystem supports continuous about-trigger data flow mode; otherwise, returns False.
	SupportsContinuousPreTrigger	Returns True if the subsystem supports continuous pre-trigger data flow mode; otherwise, returns False.
	SupportsSingleValue	Returns True if the subsystem supports single-value data flow mode; otherwise, returns False.
	SupportsWaveformModeOnly	Returns True if the subsystem supports waveform-based operations using the onboard FIFO only; otherwise, returns False. If this property is True, the buffer wrap mode must be set to WrapSingleBuffer. In addition, the buffer size must be less than or equal to the FifoSize.
Read-Only Properties (Channel-related)	MaxDifferentialChannels	Returns the number of differential channels that are supported by the subsystem.
	MaxSingleEndedChannels	Returns the number of single-ended channels that are supported by the subsystem.
	NumberOfChannels	Returns the total number of channels that are supported by the subsystem.
	SupportsChannelListInhibit	Returns True if the subsystem supports inhibition of a ChannelList entry; otherwise, returns False.
	SupportsDifferential	Returns True if the subsystem supports differential channels; otherwise, returns False.
	SupportsProgrammableGain	Returns True if the subsystem supports programmable gain for ChannelListEntry objects; otherwise, returns False.
	SupportsSingleEnded	Returns True if the subsystem supports single-ended channels; otherwise, returns False.
Read-Only Properties (Resolution-related)	NumberOfResolutions	Returns the number of resolutions that are supported by the subsystem.
	SupportedResolutions	Returns an array containing the available resolutions that are supported by the subsystem.
	SupportsSoftwareResolution	Returns True if the subsystem supports software programmable resolution; otherwise, returns False.
Read-Only Properties (Data encoding-related)	SupportsBinaryEncoding	Returns True if the subsystem supports Binary encoding; otherwise, returns False.
	SupportsTwosCompEncoding	Returns True if the subsystem supports TwosComplement encoding; otherwise, returns False.

Table 12: Members Added with the QuadratureDecoderSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Buffer-related)	SupportsBuffering	Returns True if the subsystem supports continuous acquisition to or from OlBuffer objects; otherwise, returns False.
Properties that Provide Interfaces	BufferQueue[a]	Provides an interface to a BufferQueue object.
	ChannelList[a]	Provides an interface to a ChannelList object.
	Clock[a]	Provides an interface to a Clock object.
	ReferenceTrigger[a]	Provides an interface to a ReferenceTrigger object.
	SupportedChannels	Provides an interface to a SupportedChannels object.
	Trigger[a]	Provides an interface to a Trigger object.
Methods	Abort	Stops an operation on the quadrature decoder subsystem immediately without waiting for the current operation to complete.
	Config	Configures the subsystem based on the current property settings.
	Dispose	Overloaded method that releases the quadrature decoder subsystem's connection to the DT-Open Layers device.
	ReadCount	Returns the current count of the quadrature decoder subsystem.
	Reset	Stops a continuous operation on a subsystem immediately without waiting for the current buffer to be completed, and reinitializes the subsystem to the default configuration.
	Start	Starts an operation on the quadrature decoder subsystem.
	Stop	Stops an operation on the quadrature decoder subsystem.
	ToString	Returns a string that describes the quadrature decoder subsystem and element.
Events	BufferDoneEvent[a]	Occurs when the current OlBuffer object has been filled with post-trigger data, and if the operation is stopped, occurs for each of up to 8 inprocess buffers. For output operations, occurs when the data in the OlBuffer object has been output.
	DeviceRemovedEvent	Occurs when a device is removed from the system.
	GeneralFailureEvent	Occurs when a when a general library failure occurs.
	QueueDoneEvent[a]	Occurs when no OlBuffer objects are available on the queue and the operation stops.
	QueueStoppedEvent[a]	Occurs when a continuous analog I/O operation is stopped.

a. Currently, no DT-Open Layers devices support this property/method for the quadrature decoder subsystem; it is provided for future compatibility.

Channels

The following classes are provided within the OpenLayers.Base namespace for dealing with channels in a continuous I/O operation:

- SupportedChannelInfo, described below
- SupportedChannels, described starting on page 76
- ChannelListEntry, described starting on page 76
- ChannelList, described starting on page 77
- StrainGageTeds class, described starting on page 78
- BridgeSensorTeds class, described starting on page 80

SupportedChannelInfo Class

When you get a subsystem of a specified type, the software automatically populates the properties of the SupportedChannelInfo class, listed in Table 13, for each channel.

To access a SupportedChannelInfo object, use the SupportedChannels class, described on page 76.

Table 13: Members of the SupportedChannelInfo Class

Member Type	Member Name	Description
Read/Write Properties (General)	Name	Gets and sets the name for a channel.
	ExcitationCurrentSource	Gets and sets the excitation current source (internal, external, or disabled) to apply to the channel.
	ExcitationCurrentValue	Gets and sets the value of the internal excitation current source to apply to the channel.
	InputTerminationEnabled	Specifies whether input termination is enabled (termination resistor is switched in) or disabled (not used) for the analog input channel.
	LogicalChannelNumber	Returns the zero-based logical channel number for the specified physical channel and subsystem type.
	LogicalChannelWord	For channels with multi-word data (such as a 32-bit counter), returns the zero-based word number. For channels without multi-word data, returns -1.
	MultiSensorType	For subsystems that support multiple sensor types for a channel, gets and sets the sensor type to use for the channel, such as voltage input, current, resistance, thermocouple, RTD, strain gage, bridge, and so on.
	PhysicalChannelNumber	Returns the physical channel number that maps to the subsystem type, logical channel number, and the logical channel word.
	SensorWiringConfiguration	Gets and sets the wiring configuration (two-wire, three-wire, or four-wire) for the channel.

Table 13: Members of the SupportedChannelInfo Class (cont.)

Member Type	Member Name	Description
Read/Write Properties (General, cont.)	Subsystem	Returns the subsystem object (AnalogInputSubsystem, AnalogOutputSubsystem, DigitalInputSubsystem, DigitalOutputSubsystem, CounterTimerSubsystem, TachSubystem, or QuadratureDecoderSubsystem) with which the logical channel is associated.
Read/Write Properties (Generic Sensor-Related)	SensorGain	Gets and sets the gain specific to the sensor that is connected to the channel.
	SensorOffset	Gets and sets the offset specific to the sensor that is connected to the channel.
Read/Write Properties (Accelerometer-Related)	Coupling	Gets and sets the coupling type to apply to the channel.
Read/Write Properties (Thermocouple-Related)	ThermocoupleType	Gets and sets the thermocouple type that is connected to this channel.
Read/Write Properties (RTD-Related)	RTDType	Gets and sets the RTD type that is connected to this channel.
	RtdACoefficient	Gets and sets the A coefficient used in the Callendar-Van Dusen transfer function for the RTD that is connected to this channel.
	RtdBCoefficient	Gets and sets the B coefficient used in the Callendar-Van Dusen transfer function for the RTD that is connected to this channel.
	RtdCCoefficient	Gets and sets the C coefficient used in the Callendar-Van Dusen transfer function for the RTD that is connected to this channel.
	RtdR0	Gets and sets the resistance of the RTD that is connected to this channel.
Read/Write Properties (Thermistor-Related)	ThermistorACoefficient	Gets and sets the A coefficient used in the Callendar-Van Dusen transfer function for the RTD that is connected to this channel.
	ThermistorBCoefficient	Gets and sets the B coefficient used in the Callendar-Van Dusen transfer function for the RTD that is connected to this channel.
	ThermistorCCoefficient	Gets and sets the C coefficient used in the Callendar-Van Dusen transfer function for the RTD that is connected to this channel.

Table 13: Members of the SupportedChannelInfo Class (cont.)

Member Type	Member Name	Description
Read/Write Properties (Bridge- or Strain Gage-Related)	BridgeConfiguration	Gets and sets the configuration of the bridge-based sensor or general-purpose bridge that is connected to the channel.
	GageFactor	Gets and sets the gage factor, or sensitivity, of the strain gage.
	StrainGageBridgeConfiguration	Gets and sets the configuration of the strain gage that is connected to the channel.
	StrainGageLeadWireResistance	Gets and sets the value of the lead wire resistance, in ohms.
	StrainGageNominalResistance	Gets and sets the resistance, in ohms, of the bridge while it is not under strain/load.
	StrainGageOffsetNullingValue InVolts	Gets and sets the value of the bridge output (in volts) in the unstrained/unloaded condition. Internally, this value is subtracted from any measurements before the voltage is converted to strain.
	StrainGagePoissonRatio	Gets and sets the Poisson ratio for the bridge.
	StrainGageShuntCalibration ResistorEnabled	Specifies whether the internal shunt calibration resistor is enabled (switched in) or disabled (not used).
	StrainGageShuntCalibration Value	Gets and sets the shunt calibration value for the bridge. Internally, the software multiplies the channel measurement with this value to adjust the gain of the device.
	TransducerRatedOutputInMv	Gets and sets the rated output of a full-bridge-based transducer, such as a load cell, in terms of mV/V excitation.
	TransducerCapacity	Gets and sets the full-scale range of a full-bridge-based transducer, such as a load cell, in its native engineering units.
Read-Only Properties	CjcChannel	Gets the CJC (cold junction compensation) channel that is associated with this input channel.
	IOType	Returns the type of measurement that is supported by the channel.
	SubsystemType	Returns the type of subsystem (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder) with which the logical channel is associated.
	SupportsInputTermination	Returns True if the channel supports bias return termination resistor; otherwise, returns False.
Properties that Provide Interfaces	BridgeSensorTeds	Provides an interface to the BridgeSensorTeds object.
	StrainGageTeds	Provides an interface to the StrainGageTeds object.

SupportedChannels Class

The SupportedChannels class provides the properties and methods listed in Table 13 to access a SupportedChannelInfo object.

Table 14: Members of the SupportedChannels Class

Member Type	Member Name	Description
Read-Only Properties	Count	Returns the number of SupportedChannelInfo objects in the SupportedChannels collection.
	Item ([])	Returns the SupportedChannelInfo object at the specified index ([index]) of the SupportedChannels object.
Methods	GetChannelInfo	Overloaded method that returns a SupportedChannelInfo object for the specified channel. You can specify the channel by physical channel number, by name, by subsystem type and logical channel, or by subsystem type, logical channel, and logical channel word.

You can access a SupportedChannels object through the following classes:

- AnalogInputSubsystem, described on page 38
- AnalogOutputSubsystem, described on page 47
- DigitalInputSubsystem, described on page 52
- DigitalOutputSubsystem, described on page 56
- CounterTimerSubsystem, described on page 60
- TachSubsystem, described on page 65
- QuadratureDecoderSubsystem, described on page 69

ChannelListEntry Class

The ChannelListEntry class provides the constructor and properties listed in Table 15 to encapsulate a channel entry for a channel list of a specified subsystem.

Table 15: Members of the ChannelListEntry Class

Member Type	Member Name	Description
Constructor	ChannelListEntry Constructor	Returns a ChannelListEntry object.
Read/Write Properties	Gain	Gets and sets the gain to apply to the input signal of the associated ChannelListEntry object. The default value is 1.
	Inhibit	Gets and sets the inhibit state for the ChannelListEntry object. If True, the ChannelListEntry object takes up an entry in the ChannelList and is factored into the conversion time, but data is not returned for the ChannelListEntry object. If False (the default value), data is returned for the ChannelListEntry object.

Table 15: Members of the ChannelListEntry Class (cont.)

Member Type	Member Name	Description
Read-Only Properties	Name	Returns the name for the channel associated with the ChannelListEntry object.
	PhysicalChannelNumber	Returns the physical channel number that maps to the subsystem type, logical channel number, and the logical channel word.
	SubsystemType	Returns the type of subsystem (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder) with which the logical channel is associated.

ChannelList Class

The ChannelList class provides the properties and methods listed in Table 16 to create and manage a channel list, which is a collection of ChannelListEntry objects, for use in a continuous I/O operation.

Table 16: Members of the ChannelList Class

Member Type	Member Name	Description
Read/Write Property	Item ([])	Returns or replaces the ChannelListEntry object at the specified index.
Read-Only Property	CGLDepth	Returns the maximum number of ChannelListEntry objects that the ChannelList can contain.
Methods	Add	Overloaded method that adds a channel to the end of the ChannelList.
	Contains	Returns True if the ChannelList object contains a specific ChannelListEntry object; otherwise, returns False.
	IndexOf	Overloaded method that searches for the specified channel in the ChannelList and returns the zero-based index of the first occurrence of the channel within the ChannelList.
	Insert	Overloaded method that inserts a channel into the ChannelList object at the specified index.
	Remove	Removes the first occurrence of a specific ChannelListEntry object from the ChannelList object.

A ChannelList object is accessible using any subsystem object whose **SupportsContinuous** property returns True. The following classes expose an interface to the ChannelList object:

- AnalogInputSubsystem, described on page 38
- AnalogOutputSubsystem, described on page 47
- DigitalInputSubsystem, described on page 52
- DigitalOutputSubsystem, described on page 56
- CounterTimerSubsystem, described on page 60

- TachSubsystem, described on page 65
- QuadratureDecoderSubsystem, described on page 69

StrainGageTeds Class

The StrainGageTeds class encapsulates all properties and methods that are specific to analog input channels that support TEDS (Transducer Electronic Data Sheet) for strain gages. Table 6 lists the members of the StrainGageTeds class.

Note: This class inherits the members of the TedsBase class.

Table 17: Members of the StrainGageTeds Class

Member Type	Member Name	Description
Read-Only Properties	BridgeType	Gets the type of bridge (Full Bridge, Half Bridge, or Quarter Bridge) that was specified in the TEDS data for the channel.
	CalDate	Gets the calibration date that was specified in the TEDS data for the channel.
	CalibrationPeriod	Gets the calibration period that was specified in the TEDS data for the channel.
	CalInitials	Gets the calibration initials that were specified in the TEDS data for the channel.
	ElectricalSignalType	Gets the electrical signal type that was specified in the TEDS data for the channel.
	GageArea	Gets the area of each gage element, in mm^2, that was specified in the TEDS data for the channel.
	GageFactor	Gets the gage factor, or sensitivity, of the strain gage that was specified in the TEDS data for the channel.
	GageResistance	Gets the initial (unstrained) gage resistance, in ohms, that was specified in the TEDS data for the channel.
	GageType	Gets the type of gage that was specified in the TEDS data for the channel. Refer to page 95 for more information on the values that are defined for GageType:
	IsTedsConfigured	Inherited from the TedsBase class, returns True if the TEDS data stream is read successfully; otherwise, returns False.
	ManufacturerId	Inherited from the TedsBase class, gets identifying information about the manufacturer of the sensor from the TEDS data for the channel.
	MaxElectricalValue	Gets the maximum electrical output, in V/V, that was specified in the TEDS data for the channel.

Table 17: Members of the StrainGageTeds Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (cont.)	MaximumExcitationVoltage	Gets the maximum excitation voltage that was specified in the TEDS data for the channel.
	MaxPhysicalValue	Gets the positive full-scale value, in strain, that was specified in the TEDS data for the channel.
	MeasID	Gets the measurement location ID that was specified in the TEDS data for the channel.
	MinElectricalValue	Gets the minimum electrical output, in V/V, that was specified in the TEDS data for the channel.
	MinPhysicalValue	Gets the negative full-scale value, in strain, that was specified in the TEDS data for the channel.
	ModelNumber	Inherited from the TedsBase class, gets the model number of the sensor from the TEDS data for the channel.
	NominalExcitationVoltage	Gets the nominal excitation voltage that was specified in the TEDS data for the channel.
	PoissonCoefficient	Gets the Poisson coefficient after installation that was specified in the TEDS data for the channel.
	ResponseTime	Gets the response time, in seconds, that was specified in the TEDS data for the channel.
	SerialNumber	Inherited from the TedsBase class, gets the serial number of the sensor from the TEDS data for the channel.
	TransverseSensitivity	Gets the transverse sensitivity, in percentage, that was specified in the TEDS data for the channel.
	VersionLetter	Inherited from the TedsBase class, gets the version letter of the sensor from the TEDS data for the channel.
	VersionNumber	Inherited from the TedsBase class, gets the version number of the sensor from the TEDS data for the channel.
	YoungModulus	Gets the Young's modulus, or measure of the stiffness of the material, in MPa, that was specified in the TEDS data for the channel.
	ZeroOffset	Gets the zero offset value after installation, in V/V, that was specified in the TEDS data for the channel.
Methods	ReadHardwareTeds	Reads data from a TEDS-compatible sensor that is associated with the connected channel.
	ReadVirtualTeds	Reads TEDS data from a virtual TEDS file.

BridgeSensorTeds Class

The BridgeSensorTeds class encapsulates all properties and methods that are specific to analog input channels that support TEDS (Transducer Electronic Data Sheet) for bridge-based transducers, such as load cells, with a linear output. Table 6 lists the members of the BridgeSensorTeds class.

Note: This class inherits the members of the TedsBase class.

Table 18: Members of the BridgeSensorTeds Class

Member Type	Member Name	Description
Read-Only Properties	BridgeResistance	Gets the initial (unstrained) gage resistance, in ohms, that was specified in the TEDS data for the channel.
	BridgeType	Gets the type of bridge (Full Bridge, Half Bridge, or Quarter Bridge) that was specified in the TEDS data for the channel.
	CalDate	Gets the calibration date that was specified in the TEDS data for the channel.
	CalibrationPeriod	Gets the calibration period that was specified in the TEDS data for the channel.
	CalInitials	Gets the calibration initials that were specified in the TEDS data for the channel.
	ElectricalSignalType	Gets the electrical signal type that was specified in the TEDS data for the channel.
	IsTedsConfigured	Inherited from the TedsBase class, returns True if the TEDS data stream is read successfully; otherwise, returns False.
	ManufacturerId	Inherited from the TedsBase class, gets identifying information about the manufacturer of the sensor from the TEDS data for the channel.
	MaxElectricalValue	Gets the maximum electrical output, in V/V, that was specified in the TEDS data for the channel.
	MaximumExcitationVoltage	Gets the maximum excitation voltage that was specified in the TEDS data for the channel.
	MaxPhysicalValue	Gets the positive full-scale value, in strain, that was specified in the TEDS data for the channel.
	MeasID	Gets the measurement location ID that was specified in the TEDS data for the channel.
	MinElectricalValue	Gets the minimum electrical output, in V/V, that was specified in the TEDS data for the channel.
	MinimumExcitationVoltage	Gets the minimum excitation voltage that was specified in the TEDS data for the channel.

Table 18: Members of the BridgeSensorTeds Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (cont.)	MinPhysicalValue	Gets the negative full-scale value, in strain, that was specified in the TEDS data for the channel.
	ModelNumber	Inherited from the TedsBase class, gets the model number of the sensor from the TEDS data for the channel.
	NominalExcitationVoltage	Gets the nominal excitation voltage that was specified in the TEDS data for the channel.
	PhysicalMeasurand	Gets the physical Measureand units, described on page 100, that were specified in the TEDS data for the channel.
	ResponseTime	Gets the response time, in seconds, that was specified in the TEDS data for the channel.
	SerialNumber	Inherited from the TedsBase class, gets the serial number of the sensor from the TEDS data for the channel.
	VersionLetter	Inherited from the TedsBase class, gets the version letter of the sensor from the TEDS data for the channel.
	VersionNumber	Inherited from the TedsBase class, gets the version number of the sensor from the TEDS data for the channel.
Methods	ReadHardwareTeds	Reads data from a TEDS-compatible sensor that is associated with the connected channel.
	ReadVirtualTeds	Reads TEDS data from a virtual TEDS file.

Clock Class

The Clock class provides the properties and methods listed in Table 19 for controlling the clock of a specified subsystem.

Table 19: Members of the Clock Class

Member Type	Member Name	Description
Read/Write Properties	ExtClockDivider	Gets and sets the current value of the external clock divider, which is used to set the frequency of an external clock source.
	Frequency	Gets and sets the frequency of the internal clock source.
	Source	Gets and sets the current clock source (Internal or External).
Read-Only Properties (General)	BaseClockFrequency	Returns the frequency of the base clock for the subsystem.
	SupportsSimultaneousClocking	Returns True if the subsystem supports simultaneous clocking; otherwise, returns False.

Table 19: Members of the Clock Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Internal clock-related)	MaxFrequency	Returns the maximum allowable internal clock frequency supported by the subsystem.
	MinFrequency	Returns the minimum allowable internal clock frequency supported by the subsystem.
	SupportsInternalClock	Returns True if the subsystem supports an internal clock source; otherwise, returns False.
Read-Only Properties (External clock-related)	MaxExtClockDivider	Returns the maximum allowable clock divider value supported by the subsystem.
	MinExtClockDivider	Returns the minimum allowable clock divider value supported by the subsystem.
	SupportsExternalClock	Returns True if the subsystem supports an external clock source; otherwise, returns False.

You can access a Clock object through the following classes:

- AnalogInputSubsystem, described on page 38
- AnalogOutputSubsystem, described on page 47
- DigitalInputSubsystem, described on page 52
- DigitalOutputSubsystem, described on page 56
- CounterTimerSubsystem, described on page 60
- TachSubsystem, described on page 65
- QuadratureDecoderSubsystem, described on page 69

Triggers

The following classes are provided for controlling how a subsystem is triggered:

- Trigger, described below
- Reference trigger, described on page 84
- TriggeredScan, described on page 85

Trigger Class

The Trigger class provides the properties listed in Table 20 for controlling the trigger of a subsystem. For devices that support a start trigger and a reference trigger, the Trigger class is used to set up the start trigger, which starts pre-trigger data acquisition.

Table 20: Members of the Trigger Class

Member Type	Member Name	Description
Read/Write Properties	Level	Gets and sets the trigger threshold value. By default, the trigger threshold value is in voltage unless specified otherwise for the device; see the user's manual for your device for valid threshold value settings.
	PreTriggerSource	Gets and sets the trigger type for the pre-trigger source of a subsystem when using one of the following data flow modes: DataFlow.ContinuousPrePostTrigger or DataFlow.ContinuousPreTrigger.
	ThresholdTriggerChannel	Gets and sets the number of the channel that the device monitors for the ThresholdPos or ThresholdNeg trigger event. This property is valid only if the trigger type is ThresholdPos or ThresholdNeg.
	TriggerType	Gets and sets the trigger type (Software, TTLPos External TTL, DigitalEvent, TTLNeg External TTL, ThresholdPos, or ThresholdNeg) for the subsystem.
Read-Only Properties	SupportedThresholdTrigger Channels	Returns an array containing the channels that can be used for ThresholdPos or ThresholdNeg trigger types.
	SupportsDigitalEventTrigger	Returns True if the subsystem supports a digital event trigger type; otherwise, returns False.
	SupportsNegExternalTTLTrigger	Returns True if the subsystem supports an external, falling-edge, TLL trigger; otherwise, returns False.
	SupportsNegThresholdTrigger	Returns True if the subsystem supports an negative-going analog threshold trigger; otherwise, returns False.
	SupportsPosExternalTTLTrigger	Returns True if the subsystem supports an external, rising-edge, TLL trigger; otherwise, returns False.
	SupportsPosThresholdTrigger	Returns True if the subsystem supports a positive-going analog threshold trigger; otherwise, returns False.
	SupportsSoftwareTrigger	Returns True if the subsystem supports a software (internal) trigger; otherwise, returns False.
	SupportsSvPosExternalTTLTrigger	Returns True if the subsystem supports an external, rising-edge, TLL trigger for single-value operations; otherwise, returns False.
	SupportsSvNegExternalTTLTrigger	Returns True if the subsystem supports an external, falling-edge, TLL trigger for single-value operations; otherwise, returns False.

You can access a Trigger object through the following classes:

- AnalogInputSubsystem, described on page 38
- AnalogOutputSubsystem, described on page 47
- DigitalInputSubsystem, described on page 52
- DigitalOutputSubsystem, described on page 56

- CounterTimerSubsystem, described on page 60
- TachSubsystem, described on page 65
- QuadratureDecoderSubsystem, described on page 69

ReferenceTrigger Class

The ReferenceTrigger class provides the properties listed in Table 21 for controlling the reference trigger of a subsystem. For devices that support a reference trigger, pre-trigger data acquisition stops and post-trigger acquisition starts when the reference trigger event occurs. Post-trigger acquisition stops when the number of samples you specify for the post-trigger scan count has been reached.

Table 21: Members of the ReferenceTrigger Class

Member Type	Member Name	Description
Read/Write Properties	Level	Gets and sets the threshold value for the reference trigger. By default, the threshold value is in voltage unless specified otherwise for the device; see the user's manual for your device for valid threshold value settings for the reference trigger.
	PostTriggerScanCount	Gets and sets the samples per channel to acquire after the reference trigger occurs. This property is valid only for the ReferenceTrigger object.
	ThresholdTriggerChannel	Gets and sets the number of the channel that the device monitors for the ThresholdPos or ThresholdNeg trigger event. This property is valid only if the reference trigger type is ThresholdPos or ThresholdNeg.
	TriggerType	Gets and sets the reference trigger type (Software, TTLPos External TTL, DigitalEvent, TTLNeg External TTL, ThresholdPos, or ThresholdNeg) for the subsystem.
Read-Only Properties	SupportsDigitalEventTrigger	Returns True if the subsystem supports a digital event reference trigger; otherwise, returns False.
	SupportsNegExternalTTLTrigger	Returns True if the subsystem supports an external, falling-edge, TLL reference trigger; otherwise, returns False.
	SupportsNegThresholdTrigger	Returns True if the subsystem supports an negative-going analog threshold trigger for the reference trigger; otherwise, returns False.
	SupportsPosThresholdTrigger	Returns True if the subsystem supports a positive-going analog threshold trigger for the reference trigger; otherwise, returns False.
	SupportsPosExternalTTLTrigger	Returns True if the subsystem supports an external, rising-edge, TLL reference trigger; otherwise, returns False.

Table 21: Members of the ReferenceTrigger Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (cont.)	SupportsPostTriggerScanCount	Returns True if the subsystem supports acquiring a specified number of samples after the reference trigger occurs; otherwise, returns False.
	SupportsSyncBusTrigger	Returns True if the subsystem supports a Sync Bus trigger; otherwise, returns False.
	SupportedThresholdTrigger Channels	Returns an array containing the channels that can be used for ThresholdPos or ThresholdNeg reference trigger types.

You can access a ReferenceTrigger object through the following classes:

- AnalogInputSubsystem, described on page 38
- AnalogOutputSubsystem, described on page 47
- DigitalInputSubsystem, described on page 52
- DigitalOutputSubsystem, described on page 56
- CounterTimerSubsystem, described on page 60
- TachSubsystem, described on page 65
- QuadratureDecoderSubsystem, described on page 69

TriggeredScan Class

The TriggeredScan class allows you to scan the entries in a ChannelList object a specified number of times when the device detects a specified retrigger source by using the properties listed in Table 22.

You can access the TriggeredScan object through the AnalogInputSubsystem class.

Table 22: Members of the TriggeredScan Class

Member Type	Member Name	Description
Read/Write Properties	Enabled	Gets and sets whether triggered scan mode is enabled for the subsystem.
	MultiScanCount	Gets and sets the number of times to scan the ChannelList object per retrigger.
	RetriggerFrequency	Gets and sets the current frequency of the retrigger source.
	RetriggerSource	Gets and sets the trigger type (Software, TTLPos External, DigitalEvent, TTLNeg External, ThresholdPos, or ThresholdNeg) that retriggers the scan of the ChannelList object.

Table 22: Members of the TriggeredScan Class (cont.)

Member Type	Member Name	Description
Read-Only Properties	MaxMultiScanCount	Returns the maximum number of scans per retrigger that are supported by the subsystem.
	MaxRetriggerFreq	Returns the maximum retrigger frequency that is supported by the subsystem.
	MinRetriggerFreq	Returns the minimum retrigger frequency that is supported by the subsystem.

Range Class

The Range class is used by the **VoltageRange** and **SupportedVoltageRanges** methods to return the lower and upper limits of the voltage range for an analog subsystem.

Table 23: Members of the Range Class

Member Type	Member Name	Description
Constructor	Range Constructor	Initializes a new instance of a Range object with the specified lower and upper limits of the voltage range.
Read/Write Properties	High	Gets and sets the upper limit of the voltage range.
	Low	Gets and sets the lower limit of the voltage range.

Buffer Management

The following classes are provided for managing buffers in continuous I/O operations:

- OlBuffer, described below
- BufferQueue, described on page 89

OlBuffer Class

The OlBuffer class provides the constructor, properties, and methods listed in Table 24 for encapsulating a data buffer that is used in a continuous I/O operation.

Table 24: Members of the OlBuffer Class

Member Type	Member Name	Description
Constructor	OlBuffer Constructor	Creates and returns an OlBuffer object that will hold a specified number of samples.
Read/Write Property	Tag	Gets or sets a user-defined value.

Table 24: Members of the OlBuffer Class (cont.)

Member Type	Member Name	Description
Read-Only Properties	BufferSizeInBytes	Returns the size of the internal data buffer, in bytes.
	BufferSizeInSamples	Returns the size of the internal data buffer, in samples.
	ChannelListOffset	Returns the index into the ChannelList that corresponds to the first sample in the buffer.
	Encoding	Returns the data encoding for the raw data (Binary or TwosComplement).
	Item ([])	Returns the raw data value at the specified index of the buffer.
	RawDataFormat	Returns the format of the raw data (Int16, Uint16, Int32, Float, or Double).
	Resolution	Returns the resolution of the associated subsystem.
	SampleSizeInBytes	Returns the size, in bytes, of the samples in the buffer.
	State	Gets the current state (Idle, Queued, InProcess, Completed, or Released) of the OlBuffer object.
	ValidSamples	Gets the number of valid samples in the OlBuffer object.
	VoltageRange	Returns the current upper and lower limits of the voltage range for the associated subsystem.
Methods	Dispose	Overloaded method that deallocates the OlBuffer object.
	GetDataAsBridgeBasedSensor	For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into the native engineering units of the full-bridge-based transducer, and then copies these values into a user-declared array of 64-bit floating-point (double) values.
	GetDataAsCurrent	For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into current values, in Amps, and then copies these values into a user-declared array of floating-point values.
	GetDataAsNormalizedBridge Output	For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into normalized voltage values, in mV/Vexc, for the bridge-based sensor, and then copies these values into a user-declared array of floating-point values.
	GetDataAsRawByte	Overloaded method. Copies the data, as raw counts, from an OlBuffer object into a user-declared array of bytes.
	GetDataAsRawInt16	Overloaded method. Used when the resolution of the subsystem is 16 bits or less and when the data encoding is twos complement, copies the data, as raw counts, from an OlBuffer object into a user-declared array of signed, 16-bit integers.

Table 24: Members of the OlBuffer Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	GetDataAsRawUInt16	Overloaded method. Used when the resolution of the subsystem is 16 bits or less and when the data encoding is binary, copies the data, as raw counts, from an OlBuffer object into a user-declared array of unsigned, 16-bit integers.
	GetDataAsRawUInt32	Overloaded method. Used when the resolution of the subsystem is greater than 16 bits, copies the data, as raw counts, from an OlBuffer object into a user-declared array of unsigned 32-bit integers.
	GetDataAsResistance	For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into resistance values, in ohms, and then copies these resistance values into a user-declared array of 64-bit floating-point (double) values.
	GetDataAsRpm	For a specified ChannelListEntry, converts the tachometer data from the internal buffer of an OlBuffer object into RPM values, and then copies these values into a user-declared array of 64-bit floating-point (double) values.
	GetDataAsSensor	Overloaded method. Converts the data from an OlBuffer object into sensor values using the SupportedChannelInfo.SensorGain and SupportedChannelInfo. SensorOffset values and copies this data into a user-declared array of floating-point (double) values.
	GetDataAsStrain	For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into microstrain values, and then copies these microstrain values into a user-declared array of 64-bit floating-point (double) values.
	GetDataAsTemperatureByte	For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into temperature and then copies these temperature values into a user-declared array of bytes.
	GetDataAsTemperatureDouble	For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into temperature and then copies these temperature values into a user-declared array of 64-bit floating-point (double) values.
	GetDataAsVolts	Overloaded method. Converts the data from an OlBuffer object into voltages, and copies this data into a user-declared array of floating-point values.
	GetDataAsVoltsByte	For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into voltage values, and then copies these voltage values into a user-declared array of bytes. Each voltage value is stored as an Int32, and takes 4 bytes.

Table 24: Members of the OlBuffer Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	PutDataAsRaw	Overloaded method that copies raw counts from a user-specified array into an OlBuffer object.
	PutDataAsVolts	Overloaded method that copies voltage values from a user-specified array into an OlBuffer object.
	Reallocate	Reallocates the OlBuffer object to the specified number of samples. The existing internal data buffer is deallocated and any data that it contained is lost.

BufferQueue Class

The BufferQueue class provides the properties and methods listed in Table 25 for managing a queue of OlBuffer objects for a continuous I/O operation.

Table 25: Members of the BufferQueue Class

Member Type	Member Name	Description
Read-Only Properties	InProcessCount	Returns the number of OlBuffer objects that have been taken from the queue and sent to the device for processing.
	QueuedCount	Returns the number of olBuffer objects that are on the subsystem queue (OlBuffer objects are in the queued state).
Methods	DequeueBuffer	Removes the OlBuffer object at the front of the queue, and returns it to the user.
	FreeAllQueuedBuffers	Removes all OlBuffer objects from the subsystem queue and deallocates them.
	QueueBuffer	Adds an OlBuffer object to the queue for the subsystem.

You can access a BufferQueue object through the following classes:

- AnalogInputSubsystem, described on page 38
- AnalogOutputSubsystem, described on page 47
- DigitalInputSubsystem, described on page 52
- DigitalOutputSubsystem, described on page 56
- CounterTimerSubsystem, described on page 60
- TachSubsystem, described on page 65
- QuadratureDecoderSubsystem, described on page 69

Event Handling

The following classes are provided for handling events raised by the OpenLayers.Base namespace:

- GeneralEventArgs, described on page 90
- BufferDoneEventArgs, described on page 90
- DriverRunTimeErrorEventArgs, described on page 90
- EventDoneEventArgs, described on page 91
- InterruptOnChangeEventArgs, described on page 91
- IOCompleteEventArgs, described on page 91
- MeasureDoneEventArgs, described on page 92

GeneralEventArgs

The GeneralEventArgs class provides the properties listed in Table 26 to return information about DT-Open Layers events. This object is generated internally and is returned to event delegates. Refer to page 94 for more information on delegates.

Table 26: Members of the GeneralEventArgs Class

Member Type	Member Name	Description
Read-Only Properties	DateTime	Returns the time stamp of when the associated event occurred.
	Subsystem	Returns the subsystem (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder) that raised the event.

BufferDoneEventArgs

The BufferDoneEventArgs class inherits the members of the GeneralEventArgs class and adds the **OlBuffer** property. When the BufferDoneEvent event is raised, the completed buffer is returned in the **OlBuffer** property.

This object is generated internally and returned to the **BufferDoneHandler** delegate. Refer to page 94 for more information on delegates.

DriverRunTimeErrorEventArgs

The DriverRunTimeErrorEventArgs class inherits the members of the GeneralEventArgs class and adds the properties listed in Table 27 to return data related to the event DriverRunTimeErrorEvent.

This object is generated internally and returned to the **DriverRunTimeErrorEventHandler** delegate. Refer to page 94 for more information on delegates.

Table 27: Members of the DriverRunTimeErrorEventArgs Class

Member Type	Member Name	Description
Read-Only Properties	ErrorCode	Returns the error code that is associated with the driver error. Refer to Appendix A for more information.
	Message	Returns a descriptive string associated with the error code. Refer to Appendix A for more information.

EventDoneEventArgs

The EventDoneEventArgs class inherits the members of the GeneralEventArgs class and adds the **Data** property. When the EventDoneEvent event is raised, the **Data** property returns the data associated with the event. The meaning of the data depends on the device and subsystem used. Refer to your device documentation for details.

This object is generated internally and returned to the **EventDoneHandler** delegate. Refer to page 94 for more information on delegates.

InterruptOnChangeEventArgs

The InterruptOnChangeEventArgs class inherits the members of the GeneralEventArgs class and adds the properties listed in Table 28 to return data related to the event InterruptOnChangeEvent.

This object is generated internally and returned to the **InterruptOnChangeHandler** delegate. Refer to page 94 for more information on delegates.

Table 28: Members of the InterruptOnChangeEventArgs Class

Member Type	Member Name	Description
Read-Only Properties	ChangedBits	Returns the digital input bits that changed.
	NewValue	Returns the new value of the digital input port.

IOCompleteEventArgs

The IOCompleteEventArgs class inherits the members of the GeneralEventArgs class and adds the properties listed in Table 29 to return data related to the event IOCompleteEvent.

This object is generated internally and returned to the **IOCompleteHandler** delegate. Refer to page 94 for more information on delegates.

Table 29: Members of the IOCompleteEventArgs Class

Member Type	Member Name	Description
Read-Only Properties	LastSampleNumber	For analog input operations only, returns the total number of samples per channel that were acquired from the time acquisition was started (with the start trigger) to the last post-trigger sample. For example, a value of 100 indicates that a total of 100 samples (samples 0 to 99) were acquired. You can subtract the value of the **AnalogInputSubsystem.ReferenceTrigger. PostTriggerScanCount** property, described on page 217, from the value of this property to determine when the reference trigger occurred and the number of pre-trigger samples that were acquired. For example, if the value of this property is 100, and you specified a value of 75 for the post-trigger scan count, you can determine that the reference trigger occurred at sample count 25 (100-75) of the last buffer; samples 25 through 99 are post-trigger samples and samples 0 to 24 are pre-trigger samples.

MeasureDoneEventArgs

The MeasureDoneEventArgs class inherits the members of the GeneralEventArgs class and adds the **Count** property to return the data related to the event MeasureDoneEvent. The **Count** is the number of internal clock ticks that were counted during the measurement period.

This object is generated internally and returned to the **MeasureDoneHandler** delegate. Refer to page 94 for more information on delegates.

Error Handling

The following classes are provided for handling errors that may occur in the OpenLayers.Base namespace:

- OlException, described below
- OlError, described on page 93

OlException

The OlException class provides the properties listed in Table 30 for dealing with errors that can be generated by the library.

Table 30: Members of the OlException Class

Member Type	Member Name	Description
Read-Only Properties	ErrorCode	Returns the error code from the DT-Open Layers for .NET Class Library that is associated with this exception.
	Message	Returns the descriptive string for the exception.
	Subsystem	Returns the subsystem (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder) that raised the exception. If the exception is not related to a specific subsystem, returns null.

OlError

The OlError class provides the OlError constructor for encapsulating an DT-Open Layers error code. The OlError class provides the methods listed in Table 31 for getting information about errors returned by the DT-Open Layers for .NET Class Library. Refer to Appendix A for a list of errors that may be returned by the DT-Open Layers for .NET Class Library.

Table 31: Members of the OlError Class

Member Type	Member Name	Description
Methods	GetErrorCode	Returns the error code that is associated with a specified error message in the DT-Open Layers for .NET Class Library.
	GetErrorString	Returns a description for the specified error code in the DT-Open Layers for .NET Class Library.

Services

The Utility class provides the properties and methods listed in Table 32 for getting information about assemblies and for converting data from raw counts to voltage and voltage to raw counts.

Table 32: Members of the Utility Class

Member Type	Member Name	Description
Read-Only Property	AssemblyVersion	Gets the major, minor, revision, and build numbers of the assembly.
Methods	ConvertTemperatureToVolts	For a given thermocouple type and temperature value, converts the temperature value into voltage
	ConvertVoltsToTemperature	For a given thermocouple type and voltage value, converts the voltage value into temperature.
	ComputeRectangularRosette	For a rectangular rosette, calculates the minimum and maximum principal strain values and their associated angles (in degrees).

Table 32: Members of the Utility Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	ComputeDeltaRosette	For a delta rosette, calculates the minimum and maximum principal strain values and their associated angles (in degrees).
	GetThermocoupleRange	Returns the temperature range for a given thermocouple type.
	RawValueToVolts	Converts a data value from a raw count to a voltage.
	VoltsToRawValue	Converts a voltage value into a raw count.

Delegates

DT-Open Layers events are reported to user-specified callback routines using the .NET delegates listed in Table 33.

Table 33: Delegates Included in the OpenLayers.Base Namespace

Delegate Name	Description
BufferDoneHandler	When the event BufferDoneEvent occurs, returns the subsystem that generated the event and the BufferDoneEventArgs object that is associated with the event.
BufferReusedHandler	The BufferReusedHandler delegate is called when the event BufferReusedEvent occurs.
DeviceRemovedHandler	When the event DeviceRemovedEvent occurs, returns the subsystem that generated the event and the GeneralEventArgs object that is associated with the event.
DriverRunTimeErrorEventHandler	When the event DriverRunTimeErrorEvent occurs, returns the subsystem that generated the event and the DriverRunTimeErrorEventArgs object that is associated with the event.
EventDoneHandler	When the event EventDoneEvent occurs, returns the subsystem that generated the event and the EventDoneEventArgs object that is associated with the event.
GeneralFailureHandler	When the event GeneralFailureEvent occurs, returns the subsystem that generated the event and the GeneralEventArgs object that is associated with the event.
InterruptOnChangeHandler	When the event InterruptOnChangeEvent occurs, returns the subsystem that generated the event and the InterruptOnChangeEventArgs object that is associated with the event.
IOCompleteHandler	When the event IOCompleteEvent occurs, returns the subsystem that generated the event and the IOCompleteEventArgs object that is associated with the event.

Table 33: Delegates Included in the OpenLayers.Base Namespace (cont.)

Delegate Name	Description
MeasureDoneHandler	When the event MeasureDoneEvent occurs, returns the subsystem that generated the event and the MeasureDoneEventArgs object that is associated with the event.
PreTriggerBufferDoneHandler	When the event PreTriggerBufferDoneEvent occurs, returns the subsystem that generated the event and the BufferDoneEventArgs object that is associated with the event.
QueueDoneHandler	When the event QueueDoneEvent occurs, returns the subsystem that generated the event and the GeneralEventArgs object that is associated with the event.
QueueStoppedHandler	When the event QueueStoppedEvent occurs, returns the subsystem that generated the event and the GeneralEventArgs object that is associated with the event.

Enumerations

Table 34 lists the enumerations that are used by the properties and/or methods in the OpenLayers.Base namespace.

Table 34: Enumerations Included in the OpenLayers.Base Namespace

Enumeration Name	Values	Description
BridgeConfiguration	FullBridge	Full-bridge-based sensor, such as a load cell, or a general-purpose bridge that uses four active gages.
	HalfBridge	General-purpose bridge that uses two active gages.
	QuarterBridge	General-purpose bridge that uses one active gage. You must supply an external resistor that matches the nominal resistance of the bridge to complete the bridge externally.
CascadeMode	Cascade	Two counter/timers connected.
	Single	Counter/timer is not cascaded.
ChannelDataType	Int16	Signed, 16-bit values.
	Uint16	Unsigned, 16-bit values.
	Int32	Signed, 32-bit values.
	Float	32-bit floating-point values.
	Double	64-bit, floating-point (double-bit) values.
ChannelType	SingleEnded	Channel is configured for single-ended connections.
	Differential	Channel is configured for differential connections.
ClockSource	Internal	Internal clock source.
	External	External clock source.

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
CounterMode	Count	Event counting mode.
	RateGenerator	Continuous pulse output (rate generation) mode.
	OneShot	Single output pulse (one-shot) mode.
	OneShotRepeat	Repetitive single output pulse (repetitive one-shot) mode.
	UpDown	Up/down counting mode.
	Measure	Edge-to-edge measurement mode.
	ContinuousMeasure	Continuous edge-to-edge measurement mode.
CouplingType	DC	DC coupling, where the DC offset is included.
	AC	AC coupling, where the DC offset is removed.
DataFilterType	Raw	No filter. Provides fast response times, but the data may be difficult to interpret. Use when you want to filter the data yourself. The Raw filter type returns the data exactly as it comes out of the Delta-Sigma A/D converters. Note that Delta-Sigma converters provide substantial digital filtering above the Nyquist frequency. Generally, the only time it is desirable to use the Raw filter type is if you are using fast responding inputs, sampling them at higher speeds (> 1 Hz), and need as much response speed as possible.
	MovingAverage	Provides a compromise of filter functionality and response time. This filter can be used in any application. This low-pass filter takes the previous 16 samples, adds them together, and divides by 16.
DataFlow	Continuous	Continuous I/O operation.
	SingleValue	Single-value I/O operation.
	ContinuousPreTrigger	Continuous pre-trigger input operation.
	ContinuousPrePost Trigger	Continuous about-trigger operation.

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
EdgeSelect	GateRising	The specified start or stop edge occurs on the rising edge of the gate signal.
	GateFalling	The specified start or stop edge occurs on the falling edge of the gate signal.
	ClockRising	The specified start or stop edge occurs on the rising edge of the clock signal.
	ClockFalling	The specified start or stop edge occurs on the falling edge of the clock signal.
	ADCConversionComplete	The specified start or stop edge occurs when the A/D conversion is complete.
	TachometerInputFalling	The specified start or stop edge occurs on the falling edge of the tachometer input signal.
	TachometerInputRising	The specified start or stop edge occurs on the rising edge of the tachometer input signal.
	DigitalInput0Falling	The specified start or stop edge occurs on the falling edge of digital input signal 0.
	DigitalInput0Rising	The specified start or stop edge occurs on the rising edge of digital input signal 0.
	DigitalInput1Falling	The specified start or stop edge occurs on the falling edge of digital input signal 1.
	DigitalInput1Rising	The specified start or stop edge occurs on the rising edge of digital input signal 1.
	DigitalInput2Falling	The specified start or stop edge occurs on the falling edge of digital input signal 2.
	DigitalInput2Rising	The specified start or stop edge occurs on the rising edge of digital input signal 2.
	DigitalInput3Falling	The specified start or stop edge occurs on the falling edge of digital input signal 3.
	DigitalInput3Rising	The specified start or stop edge occurs on the rising edge of digital input signal 3.
	DigitalInput4Falling	The specified start or stop edge occurs on the falling edge of digital input signal 4.
	DigitalInput4Rising	The specified start or stop edge occurs on the rising edge of digital input signal 4.
	DigitalInput5Falling	The specified start or stop edge occurs on the falling edge of digital input signal 5.

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
EdgeSelect (cont.)	DigitalInput5Rising	The specified start or stop edge occurs on the rising edge of digital input signal 5.
	DigitalInput6Falling	The specified start or stop edge occurs on the falling edge of digital input signal 6.
	DigitalInput6Rising	The specified start or stop edge occurs on the rising edge of digital input signal 6.
	DigitalInput7Falling	The specified start or stop edge occurs on the falling edge of digital input signal 7.
	DigitalInput7Rising	The specified start or stop edge occurs on the rising edge of digital input signal 7.
	CT0ClockInputFalling	The specified start or stop edge occurs on the falling edge of the clock input signal associated with counter/timer 0.
	CT0ClockInputRising	The specified start or stop edge occurs on the rising edge of the clock input signal associated with counter/timer 0.
	CT0GateInputFalling	The specified start or stop edge occurs on the falling edge of the gate input signal associated with counter/timer 0.
	CT0GateInputRising	The specified start or stop edge occurs on the rising edge of the gate input signal associated with counter/timer 0.
EdgeType	Falling	Falling edge of the tachometer signal.
	Rising	Rising edge of the tachometer signal.
Encoding	Binary	Binary data encoding.
	TwosComplement	Twos complement data encoding.
ErrorCode	See Appendix A.	The error codes that can be returned by the library.
ExcitationCurrentSource	Internal	Internal excitation current source.
	External	External excitation current source.
	Disabled	Excitation current source is disabled (no excitation is applied).
ExcitationVoltageSource	Internal	Internal excitation voltage source.
	External	External excitation voltage source.
	Disabled	Excitation voltage source is disabled (no excitation is applied).

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
GageType	SingleElement	Single element gage.
	TwoPoissonElements	Two elements with a Poisson arrangement.
	TwoOppositeSigned Elements	Two elements, opposite sign (adjacent arms).
	TwoSameSigned Elements	Two elements, same sign (opposite arms).
	TwoElementChevron	Two elements, 45° Chevron (torque or shear) arrangement.
	FourSameSign ElementsPoisson	Four elements, Poisson strains of same sign in opposite arms.
	FourOppositeSigned Elements	Four elements, Poisson strains of opposite sign in adjacent arms.
	FourUniaxialElements	Four elements, equal strains of opposite sign in adjacent arms.
	FourElementDual Chevron	Four elements, 45° Chevron (torque or shear) arrangement.
	TeeRosetteGrid1_0 Degrees	Tee Rosette grid 1 or a (0°).
	TeeRosetteGrid2_90 Degrees	Tee Rosette grid 2 or b (90°).
	DeltaRosetteGrid1_0 Degrees	Delta Rosette grid 1 or a (0°).
	DeltaRosetteGrid2_60Deg rees	Delta Rosette grid 2 or b (60°).
	DeltaRosetteGrid3_ 120Degrees	Delta Rosette grid 3 or c (120°).
	RectangularRosette Grid1_0Degrees	Rectangular Rosette grid 1 or a (0°).
	RectangularRosette Grid2_45Degrees	Rectangular Rosette grid 2 or a (45°).
	RectangularRosette Grid3_90Degrees	Rectangular Rosette grid 3 or a (90°).
	None	Software gate.
	HighLevel	Enables a C/T operation when the gate signal is high.
	LowLevel	Enables a C/T operation when the gate signal is low.
	HighEdge	Enables a C/T operation on the rising edge of the gate signal.
	LowEdge	Enables a C/T operation on the falling edge of the gate signal.

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
GateType (cont.)	Level	Enables a C/T operation on the transition from any level on the gate signal.
IOType	VoltageIn	The channel supports a voltage input.
	VoltageOut	The channels supports a voltage output.
	DigitalInput	The channel supports a digital input.
	DigitalOutput	The channel supports a digital output.
	QuadratureDecoder	The channel supports quadrature decoder operations.
	CounterTimer	The channel supports counter/timer operations.
	Tachometer	The channel supports tachometer input.
	Current	The channel supports a current input.
	Thermocouple	The channel supports a thermocouple input.
	Rtd	The channel supports an RTD input.
	StrainGage	The channel supports a strain gage input.
	Accelerometer	The channel supports an IEPE (accelerometer) input.
	Bridge	The channel supports a bridge-based sensor or general-purpose bridge input.
	Thermistor	The channel supports a thermistor input.
	Resistance	The channel supports a resistance measurement input.
	MultiSensor	The channel supports more than one sensor type. Use the **SupportedChannelInfo.MultiSensorType** property or the **SupportedChannelInfo. SupportedMutliSensorTypes** property to determine which sensor types are supported for the channel.
OlBuffer.BufferState	Idle	Buffer is allocated but not queued to a subsystem.
	Queued	Buffer is queued to a subsystem.
	InProcess	Buffer is queued to a device driver.
	Completed	Buffer has been completed by the driver and is not queued to a subsystem.
	Released	Buffer has been released.
PhysicalMeasurandUnits	Temperature_Kelvin	Temperature (Kelvin).
	Temperature_Celsius	Temperature (Celsius).
	Strain	Strain.
	Microstrain	Microstrain.
	Newton	Force/Weight (Newton).
	pounds	Force/Weight (pounds).

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
PhysicalMeasurandUnits (cont.)	kilogramForcePer Kilopound	Force/Weight (kilogram-force/kilopound).
	Acceleration_m_ss	Acceleration (m/s^2).
	Acceleration_g	Acceleration (g).
	Torque_Nm_Radian	Torque (Nm/radian).
	Torque_Nm	Torque (Nm).
	Torque_oz_in	Torque (oz-in).
	Pressure_Pascal	Pressure (Pascal).
	Pressure_PSI	Pressure (PSI).
	Mass_Kg	Mass (kg).
	Mass_g	Mass (g).
	Distance_m	Distance (m).
	Distance_mm	Distance (mm).
	Distance_inches	Distance (inches).
	Velocity_m_s	Velocity (m/s).
	Velocity_mph	Velocity (mph).
	Velocity_fps	Velocity (fps).
	AngularPosition_radian	Angular Position (radian).
	AngularPosition_ degrees	Angular Position (degrees).
	RotationalVelocity_ radian_s	Rotational Velocity (radian/s).
	RotationalVelocity_rpm	Rotational Velocity (rpm).
	Frequency	Frequency.
	Concentration_gram_ liter	Concentration (gram/liter).
	Concentration_kg_liter	Concentration (kg/liter).
	MolarConcentration_ mole_m3	Molar Concentration (mole/m^3).
	MolarConcentration_ mole_l	Molar Concentration (mole/l).
	Volumetric Concentration_m3_m3	Volumetric Concentration (m^3/m^3).
	Volumetric Concentration_l_l	Volumetric Concentration (l/l).
	MassFlow	Mass Flow.

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
PhysicalMeasurandUnits (cont.)	VolumetricFlow_m3_s	Volumetric Flow (m^3/s).
	VolumetricFlow_m3_hr	Volumetric Flow (m^3/hr).
	VolumetricFlow_gpm	Volumetric Flow (gpm).
	VolumetricFlow_cfm	Volumetric Flow (cfm).
	VolumetricFlow_l_min	Volumetric Flow (l/min).
	RelativeHumidity	Relative Humidity.
	Ratio_percent	Ratio (percent).
	Voltage	Voltage.
	RmsVoltage	RMS Voltage.
	Current	Current.
	RmsCurrent	RMS Current.
	Power_Watts	Power (Watts).
PowerSource	Internal	The device is powered by the internal system power.
	External	The device is powered by an external power source.
PulseType	HighToLow	Low part of pulse is active.
	LowToHigh	High part of pulse is active.
QuadratureIndexMode	Disabled	Indexing disabled.
	Low	Reset quadrature decoder to 0 on falling edge of Index signal.
	High	Reset quadrature decoder to 0 on rising edge of Index signal.
ReferenceTriggerType	None	Triggering is disabled.
	TTLPos	An external digital (TTL) signal attached to the device. The trigger occurs when the device detects a transition on the rising edge of the digital TTL signal.
	DigitalEvent	A trigger is generated when an external digital event occurs.
	TTLNeg	An external digital (TTL) signal attached to the device. The trigger occurs when the device detects a transition on the falling edge of the digital TTL signal.
	ThresholdPos	Either an analog signal from an analog input channel or an external analog signal attached to the device. A positive analog threshold trigger occurs when the device detects a positive-going signal that crosses a threshold value. The threshold level is generally set using an analog output subsystem on the device.

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
ReferenceTriggerType (cont.)	ThresholdNeg	Either an analog signal from an analog input channel or an external analog signal attached to the device. A negative analog threshold trigger occurs when the device detects a negative-going signal that crosses a threshold value. The threshold level is generally set using an analog output subsystem on the device.
	SyncBus	An external Sync Bus signal attached to the device. For devices that support connecting multiple devices together in a master/slave relationship using Sync Bus (RJ45) connectors, the Sync Bus trigger occurs when the slave device detects a transition on the SyncBus trigger input of the Sync Bus connector.
RTDType	Pt3750	Temperature Coefficient of Resistance value of 0.003750 Ω / Ω /° C used in the Callendar-Van Dusen transfer function for Platinum 1000 Ω RTDs. This value is specified in the Low Cost standard.
	Pt3850	Temperature Coefficient of Resistance value of 0.003850 Ω / Ω /° C used in the Callendar-Van Dusen transfer function for Platinum 100, 500, and 1000 Ω RTDs. This value is specified in the DIN/IEC 60751 and ASTM-E1137 standards.
	Pt3911	Temperature Coefficient of Resistance value of 0.003911 Ω / Ω /° C used in the Callendar-Van Dusen transfer function for Platinum 100 Ω RTDs. This value is specified in the US Industrial Standard.
	Pt3916	Temperature Coefficient of Resistance value of 0.003916 Ω / Ω /° C used in the Callendar-Van Dusen transfer function for Platinum 100 Ω RTDs. This value is specified in the Japanese JISC 1604-1989 standard.
	Pt3920	Temperature Coefficient of Resistance value of 0.003920 Ω / Ω /° C used in the Callendar-Van Dusen transfer function for Platinum 100 Ω RTDs. This value is specified in the SAMA RC21-4-1966 standard.
	Pt3928	Temperature Coefficient of Resistance value of 0.003928 Ω / Ω /° C used in the Callendar-Van Dusen transfer function for the RTD.
	Custom	A user-defined value for the Temperature Coefficient of Resistance in the Callendar-Van Dusen transfer function for the RTD.
SensorWiringConfiguration	TwoWire	The sensor type (typically, an RTD, thermistor, or resistance measurement) uses two wires to connect to the device.
	ThreeWire	The sensor type (typically, an RTD, thermistor, or resistance measurement) uses three wires to connect to the device.
	FourWire	The sensor type (typically, an RTD, thermistor, or resistance measurement) uses four wires to connect to the device.

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
StrainGageBridge Configuration	FullBridgeBending	This configurations uses four active gages to measure bending strain. This configuration rejects axial strain, compensates for temperature and compensates for lead resistance.
	FullBridgeBending Poisson	This configuration uses four active gages to measure bending strain. This configuration also rejects axial strain, compensates for temperature, compensates for lead resistance, and compensates for the aggregate effect on the principle strain measurement due to the Poisson ratio of the specimen material.
	FullBridgeAxialPoisson	This configuration uses four active gages to measure axial strain. This configuration also compensates for temperature, rejects bending strain, compensates for lead resistance, and compensates for the aggregate effect on the principle strain measurement due to the Poisson ratio of the specimen material.
	HalfBridgePoisson	This configuration uses two active gages to measure either axial or bending strain. This configuration compensates for temperature, and compensates for the aggregate effect on the principle strain measurement due to the Poisson ratio of the specimen material.
	HalfBridgeBending	This configuration uses two active gages to measure bending strain. This configuration rejects axial strain and compensates for temperature.
	QuarterBridge	This configuration uses a single active gage to measure axial or bending strain. You must supply an external resistor that matches the nominal resistance of the bridge to complete the bridge externally.
	QuarterBridgeTemp Compensation	This configuration uses one active gage and one dummy gage to measure axial and bending strain while compensating for temperature.
SubsystemBase.States	Initialized	The subsystem has been initialized but has not been configured.
	ConfiguredForSingle Value	The subsystem has been configured for a single-value operation.
	ConfiguredFor Continuous	The subsystem has been configured for a continuous operation.
	PreStarted	The subsystem has been prestarted for a simultaneous operation.
	Running	The operation on the subsystem is running.
	Stopping	The operation on the subsystem is being stopped.
	Aborting	The operation on the subsystem is being aborted.
	IoComplete	The I/O operation on the subsystem is done.

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
SubsystemType	AnalogInput	Analog input subsystem
	AnalogOutput	Analog output subsystem
	DigitalInput	Digital input subsystem
	DigitalOutput	Digital output subsystem
	QuadratureDecoder	Quadrature decoder subsystem
	CounterTimer	Counter/timer subsystem
	Tachometer	Tachometer subsystem
SynchronizationModes	None	No synchronization
	Master	Device is the master
	Slave	Device is a slave
TemperatureFilterType	Deprecated enumeration; replaced by the DataFilterType enumeration, described on page 96.	
TedsBridgeType	Quarter Bridge	Quarter-bridge configuration.
	Half Bridge	Half-bridge configuration.
	Full Bridge	Full-bridge configuration.
TedsTemplateId	NotDefined	No TEDS template associated with the channel.
	BridgeSensors	TEDS template for a bridge sensor that is associated with the channel.
	StrainGage	TEDS template for a strain gage that is associated with the channel.
TemperatureUnit	Celsius	Temperature specified in Celsius.
	Fahrenheit	Temperature specified in Fahrenheit.
	Kelvin	Temperature specified in Kelvin.
ThermocoupleType	None	No thermocouple; voltage input.
	J	Type J thermocouple.
	K	Type K thermocouple.
	B	Type B thermocouple.
	E	Type E thermocouple.
	N	Type N thermocouple.
	R	Type R thermocouple.
	S	Type S thermocouple.
	T	Type T thermocouple.

Table 34: Enumerations Included in the OpenLayers.Base Namespace (cont.)

Enumeration Name	Values	Description
TriggerType	Software	Trigger is generated when the operation is started in software.
	TTLPos	Trigger is generated on a rising edge of an external, digital (TTL) signal.
	DigitalEvent	Trigger is generated when an external digital event occurs.
	TTLNeg	Trigger is generated on a falling edge of an external, digital (TTL) signal.
	ThresholdPos	Trigger is generated when a positive-going analog signal crosses a threshold value.
	ThresholdNeg	Trigger is generated when a negative-going analog signal crosses a threshold value.

Structures

The OpenLayers.Base namespace provides the following structures:

- **HardwareInfo** structure – This structure is used by the **Device.GetHardwareInfo** method to return information about a DT-Open Layers-compliant device.

  Table 35 lists the fields that are contained in the **HardwareInfo** structure.

Table 35: Fields of the HardwareInfo Structure in the OpenLayers.Base Namespace

Field	Description
VendorId	The identification number of the vendor. For most devices, this will be 0x087 hexadecimal, which is the vendor id for Data Translation devices.
ProductId	The product identification number, such as DT9832.
DeviceId	The version of the product. If only one version of the product exists, this number is 1. If two versions of the product exist, this number could be 1 or 2.
BoardId	This field contains the year (1 or 2 digits), week (1 or 2 digits), test station (1 digit), and sequence number (3 digits) of the device when it was tested in Manufacturing. For example, if BoardId contains the value 5469419, this device was tested in 2005, week 46, on test station 9, and is unit number 419.

- **SingleValuesInfoRaw** structure – Used with the **SetSingleValuesAsRaw** method, specifies the analog output channel to update and the raw count to output on that channel.

Table 36 lists the fields that are contained in the **SingleValuesInfoRaw** structure.

Table 36: Fields of the SingleValuesInfoRaw Structure

Field	Description
PhysicalChannel	The number of the physical analog output channel to update.
RawValue	The raw count value to output on the specified analog output channel.

- **SingleValuesInfoVolts** structure – Used with the **SetSingleValuesAsVolts** method, specifies the analog output channel to update and the voltage value to output on that channel.

  Table 37 lists the fields that are contained in the **SingleValuesInfoVolts** structure.

Table 37: Fields of the SingleValuesInfoVolts Structure

Field	Description
PhysicalChannel	The number of the physical analog output channel to update.
Voltage	The voltage value to output on the specified analog output channel.

OpenLayers.DeviceCollection Namespace

The OpenLayers.DeviceCollection namespace provides the programming interface for DT-Open Layers-compatible device collections. This is the interface to use for devices that are defined as collections, such as the VIBbox system or a user-defined collection created using the DT Device Collection Manager application. (For all other DT-Open Layers-compatible devices, use the OpenLayers.Base namespace instead.)

Devices in a collection are connected together through the Sync Bus. Only subsystems, such as the analog input and possibly the analog output subsystem, are supported in the collection as these are the only subsystems that may provide expansion through the Sync Bus. Check your hardware documentation to determine which subsystems are supported in the collection.

This section describes the elements of the OpenLayers.DeviceCollection namespace. Refer to Chapter 4 for more information on how to use the OpenLayers.DeviceCollection namespace.

Classes

The OpenLayers.DeviceCollection namespace contains the classes listed in Table 38. Each class contains properties, methods, and/or events that allow you to perform specific operations. This section describes the classes and their members.

Table 38: Classes Included in the OpenLayers.DeviceCollection Namespace

Operation Type	Class Name	Description
Device Management	DeviceMgr	Manages DT-Open Layers devices in the system and assigns Device objects.
	Device	Encapsulates an DT-Open Layers device and manages and distributes subsystems for the device.
	SimultaneousStart	Provides the properties for simultaneously starting multiple subsystems.
Analog Input Operations	AnalogInputSubsystem	Provides the properties, methods, and events for performing analog input operations. This class inherits members from the AnalogSubsystem[a] and SubsystemBase[b] classes.
Analog Output Operations	AnalogOutputSubsystem	Provides the properties, methods, and events for performing analog output operations. This class inherits members from the AnalogSubsystem[a] and SubsystemBase[b] classes.

Table 38: Classes Included in the OpenLayers.DeviceCollection Namespace (cont.)

Operation Type	Class Name	Description
Channels	SupportedChannelInfo	Contains information that describes a channel that is associated with a specific subsystem.
	SupportedChannels	A collection of SupportedChannelInfo objects.
	ChannelListEntry	Encapsulates a channel entry for the channel list of a specified subsystem.
	ChannelList	Specifies a collection of ChannelListEntry objects for use in a continuous I/O operation.
Clocks	Clock	Provides an interface for controlling the clock of a subsystem.
Triggers	Trigger	Provides an interface for controlling the trigger of a subsystem. For device that support a start trigger and a reference trigger, this class controls the start trigger.
	ReferenceTrigger	Provides an interface for controlling the reference trigger of a subsystem.
Ranges	Range	Specifies the upper and lower limits of a voltage range for an analog subsystem.
Buffer Management	OlBuffer	Encapsulates a data buffer that is used in a continuous I/O operation.
	BufferQueue	Provides an interface for queuing OlBuffer objects to a device's subsystem for continuous I/O operations.
Event Handling	BufferDoneEventArgs	Contains data related to the event BufferDoneEvent. This class inherits members from the GeneralEventArgs class.[c]
	DriverRunTimeErrorEventArgs	Contains the data related to the event DriverRunTimeErrorEvent. This class inherits members from the GeneralEventArgs class.[c]
	IOCompleteEventArgs	Contains the data related to the event IOCompleteEvent. This class inherits members from the GeneralEventArgs class.[c]
Error Handling	OlException	DT-Open Layers exception class. Exceptions are raised in response to error conditions within the DT-Open Layers for .NET Class Library.
	OlError	Encapsulates an DT-Open Layers error code.

a. The AnalogSubsystem class provides the common properties, methods, and events for performing analog I/O operations. This is the base class for the analog input and analog output subsystems. This class inherits many of its capabilities from the SubsystemBase class. You cannot instantiate this object.

b. The SubsystemBase class provides the common properties, methods, and events that are inherited by the subsystems. This is the base class for all subsystems; you cannot instantiate this object.

c. The GeneralEventArgs class contains data that is returned by all DT-Open Layers events that are sent to the user.

Device Management

The OpenLayers.DeviceCollection namespace provides the following classes for managing devices:

- DeviceMgr, described below
- Device, described starting on page 110
- SimultaneousStart, described starting on page 111

DeviceMgr Class

The DeviceMgr class provides methods for managing DT-Open Layers devices in the system and for assigning a Device object to each DT-Open Layers device that you want to use. Table 39 lists the methods in the DeviceMgr class.

Note: This class exposes the Device object.

Table 39: Methods of the DeviceMgr Class

Member Type	Member Name	Description
Methods	Get	Returns a DeviceMgr object.
	GetDevice	Returns a Device object for the specified device.
	GetDeviceNames	Returns a list of all DT-Open Layers-compatible devices plugged into the system.
	HardwareAvailable	Returns True if an DT-Open Layers-compliant device is plugged into the system; otherwise, returns False.

Device Class

The Device class provides a constructor, properties, and methods for encapsulating an DT-Open Layers device and managing and distributing subsystems for the device.

To access a Device object, it is recommended that you use the **DeviceMgr.GetDevice** method. If you prefer, you can also get a Device object using the Device constructor of the Device class.

Note: This class exposes the AnalogInputSubsystem, AnalogOutputSubsystem, and SimultaneousStart objects.

Table 40 lists the members of the Device class.

Table 40: Members of the Device Class

Member Type	Member Name	Description
Constructor	Device Constructor	Returns a Device object.
Read-Only Properties	CollectionDevices	Returns an array of Device objects for each device in the collection.
	DeviceName	Returns the user-defined name of the device. This name can be modified in the DT-Open Layers Control Panel applet.
	MasterIndex	Returns the index of the master Device object in the CollectionDevices array.
Properties that Provide Interfaces	SimultaneousStart	Provides an interface to the SimultaneousStart object.
Methods	AnalogInputSubsystem	Returns an AnalogInputSubsystem object.
	AnalogOutputSubsystem	Returns an AnalogOutputSubsystem object.
	Dispose	Terminates the connection to the device.
	GetHardwareInfo	Returns hardware specific-information about the current device collection.
	GetNumSubsystemElements	Returns the number of available subsystem elements for a given subsystem type.

SimultaneousStart Class

The SimultaneousStart class allows you to start multiple subsystems simultaneously using the properties listed in Table 41. You access the SimultaneousStart object through the Device object.

Table 41: Additional Members of the SimultaneousStart Class

Member Type	Member Name	Description
Methods	AddSubsystem	Adds a subsystem to the list of subsystems to simultaneous start.
	RemoveSubsystem	Removes a subsystem from the list of subsystems to simultaneous start.
	Clear	Removes all subsystems from the simultaneous start list.
	GetSubsystemList	Returns an array of subsystems that are currently on the simultaneous start list.
	PreStart	Simultaneously prestarts all subsystems on the simultaneous start list.
	Start	Simultaneously starts all subsystems on the simultaneous start list.

Subsystem Operations

The following major classes are provided within the OpenLayers.DeviceCollection namespace for performing subsystem operations:

- AnalogInputSubsystem, described below
- AnalogOutputSubsystem, described starting on page 117

AnalogInputSubsystem Class

The AnalogInputSubsystem class encapsulates all methods, properties, and events that are specific to analog input operations. Table 42 lists the members of the AnalogInputSubsystem class.

To create an instance of this class, use the **Device.AnalogInputSubsystem** method (recommended) or the AnalogInputSubsystem constructor.

Note: This class provides interfaces to the following objects: BufferQueue, ChannelList, Clock, SupportedChannels, and Trigger.

This class inherits the members of the AnalogSubsystem and SubsystemBase classes.

Table 42: Members of the AnalogInputSubsystem Class

Member Type	Member Name	Description
Constructor	AnalogInputSubsystem Constructor	Gets an analog input subsystem.
Read/Write Properties	AsynchronousStop	Gets and sets the stop behavior (synchronous or asynchronous) of the subsystem.
	ChannelType	Gets and sets the channel type (SingleEnded or Differential) for the subsystem.
	DataFlow	Gets and sets the data flow mode (Continuous, SingleValue, ContinuousPreTrigger ContinuousPrePostTrigger) for the subsystem.
	Encoding	Gets and sets the data encoding (Binary or TwosComplement) for the subsystem.
	StopOnError	Gets and sets the stop-on-error condition (stop if overrun occurs, or continue if overrun occurs) for the subsystem.
	SynchronousBufferDone	Gets and sets the way Buffer Done events are executed (asynchronously or synchronously).
	VoltageRange	Gets and sets the current voltage range for the subsystem.

Table 42: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (General)	Device	Returns the Device object that is associated with the subsystem.
	Element	Returns the element number of the subsystem.
	FifoSize	Returns the size of the FIFO on the device that is associated with the subsystem.
	IsRunning	Returns True if the subsystem is currently running; otherwise, returns False.
	ReturnsFloats	Returns True if the subsystem returns floating-point values; otherwise, returns False indicating that the subsystem returns integer values.
	State	Returns the current state of the subsystem (Initialized, ConfiguredForSingleValue, ConfiguredForContinuous, PreStarted, Running, Stopping, Aborting, or IoComplete).
	SubsystemType	Returns the subsystem type (AnalogInput or AnalogOutput).
	SupportsAutoCalibrate	Returns True if the subsystem supports self-calibration, where an auto-zero function is performed through software; otherwise, returns False.
	SupportsSetSingleValues	Returns True if the subsystem supports updating multiple channels simultaneously with a single value (using **SetSingleValuesAsRaw** or **SetSingleValuesAsVolts**); otherwise, returns False.
	SupportsSimultaneousStart	Returns True if the subsystem supports starting multiple subsystems simultaneously; otherwise, returns False.
Read-Only Properties (Data flow-related)	SupportsContinuous	Returns True if the subsystem supports continuous data flow mode; otherwise, returns False.
	SupportsContinuousPrePostTrigger	Returns True if the subsystem supports continuous about-trigger data flow mode; otherwise, returns False.
	SupportsContinuousPreTrigger	Returns True if the subsystem supports continuous pre-trigger data flow mode; otherwise, returns False.
	SupportsSingleValue	Returns True if the subsystem supports single-value data flow mode; otherwise, returns False.
	SupportsWaveformModeOnly	Returns True if the subsystem supports waveform-based operations using the onboard FIFO only; otherwise, returns False. If this property is True, the buffer wrap mode must be set to WrapSingleBuffer. In addition, the buffer size must be less than or equal to the FifoSize.

Table 42: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Channel-related)	MaxDifferentialChannels	Returns the number of differential channels that are supported by the subsystem.
	MaxSingleEndedChannels	Returns the number of single-ended channels that are supported by the subsystem.
	NumberOfChannels	Returns the total number of channels that are supported by the subsystem.
	SupportsChannelListInhibit	Returns True if the subsystem supports inhibition of a ChannelList entry; otherwise, returns False.
	SupportsDifferential	Returns True if the subsystem supports differential channels; otherwise, returns False.
	SupportsSingleEnded	Returns True if the subsystem supports single-ended channels; otherwise, returns False.
Read-Only Properties (Gain-related)	NumberOfSupportedGains	Returns the number of available gains for this subsystem.
	SupportedGains	Returns an array of available gains for the subsystem.
	SupportsProgrammableGain	Returns True if the subsystem supports programmable gain for ChannelListEntry objects; otherwise, returns False.
Read-Only Properties (Range-related)	NumberOfRanges	Returns the number of available voltage ranges for the subsystem.
	SupportedVoltageRanges	Returns an array of available voltage ranges supported by the subsystem.
Read-Only Properties (Resolution-related)	NumberOfResolutions	Returns the number of resolutions that are supported by the subsystem.
	Resolution	Returns the current resolution of the subsystem.
	SupportedResolutions	Returns an array containing the available resolutions that are supported by the subsystem.
	SupportsSoftwareResolution	Returns True if the subsystem supports software programmable resolution; otherwise, returns False.
Read-Only Properties (Data encoding-related)	SupportsBinaryEncoding	Returns True if the subsystem supports Binary encoding; otherwise, returns False.
	SupportsTwosCompEncoding	Returns True if the subsystem supports TwosComplement encoding; otherwise, returns False.
Read-Only Properties (Buffer-related)	QueuedBufferDones	Returns the number of Buffer Done Events queued to be sent when **SynchronousBufferDone** is True.
	SupportsBuffering	Returns True if the subsystem supports continuous acquisition to or from OlBuffer objects; otherwise, returns False.

Table 42: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Accelerometer-related)	SupportsIepe	Returns True if the subsystem supports IEPE (accelerometer) inputs; otherwise, returns False.
	SupportsACCoupling	Returns True if the subsystem supports AC coupling, where the DC offset is removed; otherwise, returns False.
	SupportsDCCoupling	Returns True if the subsystem supports DC coupling, where the DC offset is included; otherwise, returns False.
	SupportedExcitationCurrentValues	Returns an array containing the available values for the internal excitation current source.
	SupportsExternalExcitationCurrent Src	Returns True if the subsystem supports an external excitation current source; otherwise, returns False.
	SupportsInternalExcitationCurrent Src	Returns True if the subsystem supports an internal excitation current source; otherwise, returns False.
Read-Only Property (Current-Related)	SupportsCurrentOutput	Returns True if the subsystem supports current outputs; otherwise, returns False.
Properties that Provide Interfaces	BufferQueue	Provides an interface to a BufferQueue object.
	ChannelList	Provides an interface to a ChannelList object.
	Clock	Provides an interface to a Clock object.
	ReferenceTrigger	Provides an interface to a ReferenceTrigger object.
	SupportedChannels	Provides an interface to a SupportedChannels object.
	Trigger	Provides an interface to a Trigger object.
Methods	Abort	Stops a continuous operation on the subsystem immediately without waiting for the current buffer to be filled.
	AutoCalibrate	Calibrates the subsystem in software, performing an auto-zero function.
	Config	Configures the subsystem based on the current property settings.
	Dispose	Releases the analog input subsystem's connection to the DT-Open Layers device.
	GetOneBuffer	Using continuous acquisition, acquires one buffer of data from the specified channel. This method uses the specified clock frequency, trigger, and so on, for the acquisition. This method is synchronous and returns only when the requested data has been acquired or a calculated timeout value is exceeded.
	GetSingleValueAsRaw	Acquires a single value from an input channel and returns it in raw counts.

Table 42: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	GetSingleValueAsSensor	Acquires a single value from an input channel and returns it in the engineering units for the specified sensor.
	GetSingleValueAsVolts	Acquires a single value from an input channel and returns the data in voltage.
	MoveFromBufferInprocess	Moves samples from the OlBuffer object that is currently being filled into a new OlBuffer object.
	RawValueToSensor	Overloaded method that converts a raw count to a sensor value in engineering units.
	RawValueToVolts	Overloaded method that converts a raw count into a voltage value.
	Reset	Stops a continuous operation on a subsystem immediately without waiting for the current buffer to be filled, and reinitializes the subsystem to the default configuration.
	Start	Starts a continuous operation on the analog input subsystem.
	Stop	Stops a continuous operation on the analog input subsystem after the current buffer has been filled.
	ToString	Returns a string that describes the analog input subsystem and element.
	VoltsToRawValue	Converts a voltage value into a raw count.
Events	BufferDoneEvent	Occurs when the current OlBuffer object has been filled with post-trigger data, and if the operation is stopped, occurs for each of up to 8 inprocess buffers.
	DeviceRemovedEvent	Occurs when a device is removed from the system.
	DriverRunTimeErrorEventEvent	Occurs when the device driver detects one of the following error conditions during runtime: FifoOverflow, FifoUnderflow, DeviceOverClocked, TriggerError, or DeviceError.
	GeneralFailureEvent	Occurs when a when a general library failure occurs.

Table 42: Members of the AnalogInputSubsystem Class (cont.)

Member Type	Member Name	Description
Events (cont.)	IOCompleteEvent	For analog input operations that use a reference trigger whose trigger type is something other than software (none), occurs when the last post-trigger sample is copied into the user buffer. Devices that do not support a reference trigger will never receive this event for analog input operations.
	PreTriggerBufferDoneEvent	Occurs when the OlBuffer object is filled with pre-trigger data (for an input operation only).
	QueueDoneEvent	Occurs when no OlBuffer objects are available on the queue and the operation stops.
	QueueStoppedEvent	Occurs when a pre- or post-trigger acquisition operation completes or when you stop a continuous analog input operation.

AnalogOutputSubsystem Class

The AnalogOutputSubsystem class encapsulates all methods, properties, and events that are specific to analog output operations. Table 43 lists the members of the AnalogOutputSubsystem class.

To create an instance of this class, use the **Device.AnalogOutputSubsystem** method (recommended) or the AnalogOutputSubsystem constructor.

Note: This class provides interfaces to the following objects: BufferQueue, ChannelList, Clock, SupportedChannels, and Trigger.

This class inherits the members of the AnalogSubsystem and SubsystemBase classes.

Table 43: Members Added with the AnalogOutputSubsystem Class

Member Type	Member Name	Description
Constructor	AnalogOutputSubsystem Constructor	Gets an analog output subsystem.
Read/Write Properties	AsynchronousStop	Gets and sets the stop behavior (synchronous or asynchronous) of the subsystem.
	ChannelType	Gets and sets the channel type (SingleEnded or Differential) for the subsystem.
	DataFlow	Gets and sets the data flow mode (Continuous or SingleValue) for the subsystem.
	Encoding	Gets and sets the data encoding (Binary or TwosComplement) for the subsystem.

Table 43: Members Added with the AnalogOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Read/Write Properties	StopOnError	Gets and sets the stop-on-error condition (stop if underrun occurs, or continue if underrun occurs) for the subsystem.
	SynchronousBufferDone	Gets and sets the way Buffer Done events are executed (asynchronously or synchronously).
	VoltageRange	Gets and sets the current voltage range for the subsystem.
	WrapSingleBuffer	Gets and sets the wrap mode. If True, the device driver continuously reuses the first buffer queued to the subsystem. If False, the device driver uses all the buffers queued to the subsystem (this is the default mode).
Read-Only Properties (General)	Device	Returns the Device object that is associated with the subsystem.
	Element	Returns the element number of the subsystem.
	FifoSize	Returns the size of the FIFO on the device that is associated with the subsystem.
	IsRunning	Returns True if the subsystem is currently running; otherwise, returns False.
	ReturnsFloats	Returns True if the subsystem returns floating-point values; otherwise, returns False indicating that the subsystem returns integer values.
	State	Returns the current state of the subsystem (Initialized, ConfiguredForSingleValue, ConfiguredForContinuous, PreStarted, Running, Stopping, Aborting, or IoComplete).
	SubsystemType	Returns the subsystem type (AnalogInput or AnalogOutput).
	SupportsCurrentOutput	Returns True if the subsystem supports current outputs; otherwise, returns False.
	SupportsMute	Returns True if the subsystem supports the ability to mute and/or unmute the output voltage.
	SupportsSetSingleValues	Returns True if the subsystem supports updating multiple channels simultaneously with a single value (using **SetSingleValuesAsRaw** or **SetSingleValuesAsVolts**); otherwise, returns False.
	SupportsSimultaneousStart	Returns True if the subsystem supports starting multiple subsystems simultaneously; otherwise, returns False.

Table 43: Members Added with the AnalogOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Data flow-related)	SupportsContinuous	Returns True if the subsystem supports continuous data flow mode; otherwise, returns False.
	SupportsContinuousPrePostTrigger	Returns True if the subsystem supports continuous about-trigger data flow mode; otherwise, returns False.
	SupportsContinuousPreTrigger	Returns True if the subsystem supports continuous pre-trigger data flow mode; otherwise, returns False.
	SupportsSingleValue	Returns True if the subsystem supports single-value data flow mode; otherwise, returns False.
	SupportsWaveformModeOnly	Returns True if the subsystem supports waveform-based operations using the onboard FIFO only; otherwise, returns False. If this property is True, the buffer wrap mode must be set to WrapSingleBuffer. In addition, the buffer size must be less than or equal to the FifoSize.
Read-Only Properties (Channel-related)	MaxDifferentialChannels	Returns the number of differential channels that are supported by the subsystem.
	MaxSingleEndedChannels	Returns the number of single-ended channels that are supported by the subsystem.
	NumberOfChannels	Returns the total number of channels that are supported by the subsystem.
	SupportsChannelListInhibit	Returns True if the subsystem supports inhibition of a ChannelList entry; otherwise, returns False.
	SupportsDifferential	Returns True if the subsystem supports differential channels; otherwise, returns False.
	SupportsSingleEnded	Returns True if the subsystem supports single-ended channels; otherwise, returns False.
Read-Only Properties (Gain-related)	NumberOfSupportedGains	Returns the number of available gains for this subsystem.
	SupportedGains	Returns an array of available gains for the subsystem.
	SupportsProgrammableGain	Returns True if the subsystem supports programmable gain for ChannelListEntry objects; otherwise, returns False.
Read-Only Properties (Range-related)	NumberOfRanges	Returns the number of available voltage ranges for the subsystem.
	SupportedVoltageRanges	Returns an array of available voltage ranges supported by the subsystem.

Table 43: Members Added with the AnalogOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (Resolution-related)	NumberOfResolutions	Returns the number of resolutions that are supported by the subsystem.
	Resolution	Returns the current resolution of the subsystem.
	SupportedResolutions	Returns an array containing the available resolutions that are supported by the subsystem.
	SupportsSoftwareResolution	Returns True if the subsystem supports software programmable resolution; otherwise, returns False.
Read-Only Properties (Data encoding-related)	SupportsBinaryEncoding	Returns True if the subsystem supports Binary encoding; otherwise, returns False.
	SupportsTwosCompEncoding	Returns True if the subsystem supports TwosComplement encoding; otherwise, returns False.
Read-Only Properties (Buffer-related)	QueuedBufferDones	Returns the number of Buffer Done Events queued to be sent when **SynchronousBufferDone** is True.
	SupportsBuffering	Returns True if the subsystem supports continuous acquisition to or from OlBuffer objects; otherwise, returns False.
	SupportsWrapSingle	Returns True if the subsystem supports reusing a single buffer for continuous operations; otherwise, returns False.
Properties that Provide Interfaces	BufferQueue	Provides an interface to a BufferQueue object.
	ChannelList	Provides an interface to a ChannelList object.
	Clock	Provides an interface to a Clock object.
	ReferenceTrigger	Provides an interface to a ReferenceTrigger object.
	SupportedChannels	Provides an interface to a SupportedChannels object.
	Trigger	Provides an interface to a Trigger object.
Methods	Abort	Stops a continuous operation on the subsystem immediately without waiting for the data in current buffer to be output.
	Config	Configures the subsystem based on the current property settings.
	Dispose	Overloaded method that releases the analog output subsystem's connection to the DT-Open Layers device.
	Reset	Stops a continuous operation on a subsystem immediately without waiting for the data in the current buffer to be output, and reinitializes the subsystem to the default configuration.
	Mute	Attenuates the output voltage of the subsystem to 0 V over a hardware-dependent number of samples.
	RawValueToSensor	Overloaded method that converts a raw count to a sensor value in engineering units.

Table 43: Members Added with the AnalogOutputSubsystem Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	RawValueToVolts	Overloaded method that converts a raw count into a voltage value.
	SetSingleValueAsRaw	Writes a single raw count to an analog output channel.
	SetSingleValueAsVolts	Writes a single voltage value to an analog output channel.
	SetSingleValuesAsRaw	For subsystems that support simultaneous operations, simultaneously updates the specified analog output channels with a single raw count value. You specify the channels to update and the value to output on each channel.
	SetSingleValuesAsVolts	For subsystems that support simultaneous operations, simultaneously updates the specified analog output channels with a single voltage value. You specify the channels to update and the value to output on each channel.
	Start	Starts a continuous operation on the analog output subsystem.
	Stop	Stops a continuous operation on the analog output subsystem after the data in the current buffer has been output.
	ToString	Returns a string that describes the analog output subsystem and element.
	UnMute	If the subsystem is muted, returns the output voltage of the subsystem to its current level over a hardware-dependent number of samples.
	VoltsToRawValue	Converts a voltage value into a raw count.
Events	BufferDoneEvent	Occurs when all the data in the OlBuffer object has been output.
	DeviceRemovedEvent	Occurs when a device is removed from the system.
	DriverRunTimeErrorEventEvent	Occurs when the device driver detects one of the following error conditions during runtime: FifoOverflow, FifoUnderflow, DeviceOverClocked, TriggerError, or DeviceError.
	GeneralFailureEvent	Occurs when a when a general library failure occurs.
	IOCompleteEvent	For analog output operations, occurs when the when the last data point has been output from an analog output channel. In some cases, this event is raised well after the data is transferred from the buffer (and, therefore, well after BufferDoneEvent and QueueDoneEvents occur).
	QueueDoneEvent	Occurs when no OlBuffer objects are available on the queue and the operation stops.
	QueueStoppedEvent	Occurs when a continuous analog output operation is stopped and the queue is emptied.

Channels

The following classes are provided within the OpenLayers.DeviceCollection namespace for dealing with channels in a continuous I/O operation:

- SupportedChannelInfo, described below
- SupportedChannels, described starting on page 123
- ChannelListEntry, described starting on page 124
- ChannelList, described starting on page 124

SupportedChannelInfo Class

When you get a subsystem of a specified type, the software automatically populates the properties of the SupportedChannelInfo class, listed in Table 44, for each channel.

To access a SupportedChannelInfo object, use the SupportedChannels class, described on page 123.

Table 44: Members of the SupportedChannelInfo Class

Member Type	Member Name	Description
Read/Write Properties (General)	Name	Gets and sets the name for a channel.
	ExcitationCurrentSource	Gets and sets the excitation current source (internal, external, or disabled) to apply to the channel.
	ExcitationCurrentValue	Gets and sets the value of the internal excitation current source to apply to the channel.
	LogicalChannelNumber	Returns the zero-based logical channel number for the specified physical channel and subsystem type.
	LogicalChannelWord	For channels with multi-word data (such as a 32-bit counter), returns the zero-based word number. For channels without multi-word data, returns -1.
	PhysicalChannelNumber	Returns the physical channel number that maps to the subsystem type, logical channel number, and the logical channel word.
	Subsystem	Returns the subsystem object (AnalogInputSubsystem or AnalogOutputSubsystem) with which the logical channel is associated.
Read/Write Properties (Generic Sensor-Related)	SensorGain	Gets and sets the gain specific to the sensor that is connected to the channel.
	SensorOffset	Gets and sets the offset specific to the sensor that is connected to the channel.

Table 44: Members of the SupportedChannelInfo Class (cont.)

Member Type	Member Name	Description
Read/Write Properties (Accelerometer-Related)	Coupling	Gets and sets the coupling type to apply to the channel.
Read-Only Properties	IOType	Returns the type of measurement that is supported by the channel.
	SubsystemType	Returns the type of subsystem (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder) with which the logical channel is associated.

SupportedChannels Class

The SupportedChannels class provides the properties and methods listed in Table 13 to access a SupportedChannelInfo object.

Table 45: Members of the SupportedChannels Class

Member Type	Member Name	Description
Read-Only Properties	Count	Returns the number of SupportedChannelInfo objects in the SupportedChannels collection.
	Item ([])	Returns the SupportedChannelInfo object at the specified index ([index]) of the SupportedChannels object.
Methods	GetChannelInfo	Overloaded method that returns a SupportedChannelInfo object for the specified channel. You can specify the channel by physical channel number, by name, by subsystem type and logical channel, or by subsystem type, logical channel, and logical channel word.

You can access a SupportedChannels object through the following classes:

- AnalogInputSubsystem, described on page 112
- AnalogOutputSubsystem, described on page 117

ChannelListEntry Class

The ChannelListEntry class provides the constructor and properties listed in Table 46 to encapsulate a channel entry for a channel list of a specified subsystem.

Table 46: Members of the ChannelListEntry Class

Member Type	Member Name	Description
Constructor	ChannelListEntry Constructor	Returns a ChannelListEntry object.
Read/Write Properties	Gain	Gets and sets the gain to apply to the input signal of the associated ChannelListEntry object. The default value is 1.
	Inhibit	Gets and sets the inhibit state for the ChannelListEntry object. If True, the ChannelListEntry object takes up an entry in the ChannelList and is factored into the conversion time, but data is not returned for the ChannelListEntry object. If False (the default value), data is returned for the ChannelListEntry object.
Read-Only Properties	Name	Returns the name for the channel associated with the ChannelListEntry object.
	PhysicalChannelNumber	Returns the physical channel number that maps to the subsystem type, logical channel number, and the logical channel word.
	SubsystemType	Returns the type of subsystem (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder) with which the logical channel is associated.

ChannelList Class

The ChannelList class provides the properties and methods listed in Table 47 to create and manage a channel list, which is a collection of ChannelListEntry objects, for use in a continuous I/O operation.

Table 47: Members of the ChannelList Class

Member Type	Member Name	Description
Read/Write Property	Item ([])	Returns or replaces the ChannelListEntry object at the specified index.
Read-Only Property	CGLDepth	Returns the maximum number of ChannelListEntry objects that the ChannelList can contain.
Methods	Add	Overloaded method that adds a channel to the end of the ChannelList.
	Contains	Returns True if the ChannelList object contains a specific ChannelListEntry object; otherwise, returns False.
	IndexOf	Overloaded method that searches for the specified channel in the ChannelList and returns the zero-based index of the first occurrence of the channel within the ChannelList.

Table 47: Members of the ChannelList Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	Insert	Overloaded method that inserts a channel into the ChannelList object at the specified index.
	Remove	Removes the first occurrence of a specific ChannelListEntry object from the ChannelList object.

A ChannelList object is accessible using any subsystem object whose **SupportsContinuous** property returns True. The following classes expose an interface to the ChannelList object:

- AnalogInputSubsystem, described on page 112
- AnalogOutputSubsystem, described on page 117

Clock Class

The Clock class provides the properties and methods listed in Table 48 for controlling the clock of a specified subsystem.

Table 48: Members of the Clock Class

Member Type	Member Name	Description
Read/Write Properties	ExtClockDivider	Gets and sets the current value of the external clock divider, which is used to set the frequency of an external clock source.
	Frequency	Gets and sets the frequency of the internal clock source.
	Source	Gets and sets the current clock source (Internal or External).
Read-Only Properties (General)	BaseClockFrequency	Returns the frequency of the base clock for the subsystem.
	SupportsSimultaneousClocking	Returns True if the subsystem supports simultaneous clocking; otherwise, returns False.
Read-Only Properties (Internal clock-related)	MaxFrequency	Returns the maximum allowable internal clock frequency supported by the subsystem.
	MinFrequency	Returns the minimum allowable internal clock frequency supported by the subsystem.
	SupportsInternalClock	Returns True if the subsystem supports an internal clock source; otherwise, returns False.
Read-Only Properties (External clock-related)	MaxExtClockDivider	Returns the maximum allowable clock divider value supported by the subsystem.
	MinExtClockDivider	Returns the minimum allowable clock divider value supported by the subsystem.
	SupportsExternalClock	Returns True if the subsystem supports an external clock source; otherwise, returns False.

You can access a Clock object through the following classes:

- AnalogInputSubsystem, described on page 112
- AnalogOutputSubsystem, described on page 117

Triggers

The following classes are provided for controlling how a subsystem is triggered:

- Trigger, described below
- Reference trigger, described on page 84

Trigger Class

The Trigger class provides the properties listed in Table 49 for controlling the trigger of a subsystem. For devices that support a start trigger and a reference trigger, the Trigger class is used to set up the start trigger, which starts pre-trigger data acquisition.

Table 49: Members of the Trigger Class

Member Type	Member Name	Description
Read/Write Properties	Level	Gets and sets the trigger threshold value. By default, the trigger threshold value is in voltage unless specified otherwise for the device; see the user's manual for your device for valid threshold value settings.
	PreTriggerSource	Gets and sets the trigger type for the pre-trigger source of a subsystem when using one of the following data flow modes: DataFlow.ContinuousPrePostTrigger or DataFlow.ContinuousPreTrigger.
	ThresholdTriggerChannel	Gets and sets the number of the channel that the device monitors for the ThresholdPos or ThresholdNeg trigger event. This property is valid only if the trigger type is ThresholdPos or ThresholdNeg.
	TriggerType	Gets and sets the trigger type (Software, TTLPos External TTL, DigitalEvent, TTLNeg External TTL, ThresholdPos, or ThresholdNeg) for the subsystem.
Read-Only Properties	SupportedThresholdTrigger Channels	Returns an array containing the channels that can be used for ThresholdPos or ThresholdNeg trigger types.
	SupportsDigitalEventTrigger	Returns True if the subsystem supports a digital event trigger type; otherwise, returns False.
	SupportsNegExternalTTLTrigger	Returns True if the subsystem supports an external, falling-edge, TLL trigger; otherwise, returns False.
	SupportsNegThresholdTrigger	Returns True if the subsystem supports an negative-going analog threshold trigger; otherwise, returns False.
	SupportsPosExternalTTLTrigger	Returns True if the subsystem supports an external, rising-edge, TLL trigger; otherwise, returns False.
	SupportsPosThresholdTrigger	Returns True if the subsystem supports a positive-going analog threshold trigger; otherwise, returns False.

Table 49: Members of the Trigger Class (cont.)

Member Type	Member Name	Description
Read-Only Properties (cont.)	SupportsSoftwareTrigger	Returns True if the subsystem supports a software (internal) trigger; otherwise, returns False.
	SupportsSvPosExternalTTLTrigger	Returns True if the subsystem supports an external, rising-edge, TLL trigger for single-value operations; otherwise, returns False.
	SupportsSvNegExternalTTLTrigger	Returns True if the subsystem supports an external, falling-edge, TLL trigger for single-value operations; otherwise, returns False.

You can access a Trigger object through the following classes:

- AnalogInputSubsystem, described on page 112
- AnalogOutputSubsystem, described on page 117

ReferenceTrigger Class

The ReferenceTrigger class provides the properties listed in Table 50 for controlling the reference trigger of a subsystem. For devices that support a reference trigger, pre-trigger data acquisition stops and post-trigger acquisition starts when the reference trigger event occurs. Post-trigger acquisition stops when the number of samples you specify for the post-trigger scan count has been reached.

Table 50: Members of the ReferenceTrigger Class

Member Type	Member Name	Description
Read/Write Properties	Level	Gets and sets the threshold value for the reference trigger. By default, the threshold value is in voltage unless specified otherwise for the device; see the user's manual for your device for valid threshold value settings for the reference trigger.
	PostTriggerScanCount	Gets and sets the samples per channel to acquire after the reference trigger occurs. This property is valid only for the ReferenceTrigger object.
	ThresholdTriggerChannel	Gets and sets the number of the channel that the device monitors for the ThresholdPos or ThresholdNeg trigger event. This property is valid only if the reference trigger type is ThresholdPos or ThresholdNeg.
	TriggerType	Gets and sets the reference trigger type (Software, TTLPos External TTL, DigitalEvent, TTLNeg External TTL, ThresholdPos, or ThresholdNeg) for the subsystem.

Table 50: Members of the ReferenceTrigger Class (cont.)

Member Type	Member Name	Description
Read-Only Properties	SupportsDigitalEventTrigger	Returns True if the subsystem supports a digital event reference trigger; otherwise, returns False.
	SupportsNegExternalTTLTrigger	Returns True if the subsystem supports an external, falling-edge, TLL reference trigger; otherwise, returns False.
	SupportsNegThresholdTrigger	Returns True if the subsystem supports an negative-going analog threshold trigger for the reference trigger; otherwise, returns False.
	SupportsPosThresholdTrigger	Returns True if the subsystem supports a positive-going analog threshold trigger for the reference trigger; otherwise, returns False.
	SupportsPosExternalTTLTrigger	Returns True if the subsystem supports an external, rising-edge, TLL reference trigger; otherwise, returns False.
	SupportsPostTriggerScanCount	Returns True if the subsystem supports acquiring a specified number of samples after the reference trigger occurs; otherwise, returns False.
	SupportedThresholdTrigger Channels	Returns an array containing the channels that can be used for ThresholdPos or ThresholdNeg reference trigger types.

You can access a ReferenceTrigger object through the following classes:

- AnalogInputSubsystem, described on page 112.
- AnalogOutputSubsystem, described on page 117.

Range Class

The Range class is used by the **VoltageRange** and **SupportedVoltageRanges** methods to return the lower and upper limits of the voltage range for an analog subsystem.

Table 51: Members of the Range Class

Member Type	Member Name	Description
Constructor	Range Constructor	Initializes a new instance of a Range object with the specified lower and upper limits of the voltage range.
Read/Write Properties	High	Gets and sets the upper limit of the voltage range.
	Low	Gets and sets the lower limit of the voltage range.

Buffer Management

The following classes are provided for managing buffers in continuous I/O operations:

- OlBuffer, described below
- BufferQueue, described on page 131

OlBuffer Class

The OlBuffer class provides the constructor, properties, and methods listed in Table 52 for encapsulating a data buffer that is used in a continuous I/O operation.

Table 52: Members of the OlBuffer Class

Member Type	Member Name	Description
Constructor	OlBuffer Constructor	Creates and returns an OlBuffer object that will hold a specified number of samples.
Read/Write Property	Tag	Gets or sets a user-defined value.
Read-Only Properties	BufferSizeInBytes	Returns the size of the internal data buffer, in bytes.
	BufferSizeInSamples	Returns the size of the internal data buffer, in samples.
	ChannelListOffset	Returns the index into the ChannelList that corresponds to the first sample in the buffer.
	Encoding	Returns the data encoding for the raw data (Binary or TwosComplement).
	Item ([])	Returns the raw data value at the specified index of the buffer.
	RawDataFormat	Returns the format of the raw data (Int16, Uint16, Int32, Float, or Double).
	Resolution	Returns the resolution of the associated subsystem.
	SampleSizeInBytes	Returns the size, in bytes, of the samples in the buffer.
	State	Gets the current state (Idle, Queued, InProcess, Completed, or Released) of the OlBuffer object.
	ValidSamples	Gets the number of valid samples in the OlBuffer object.
	VoltageRange	Returns the current upper and lower limits of the voltage range for the associated subsystem.
Methods	Dispose	Overloaded method that deallocates the OlBuffer object.
	GetDataAsRawByte	Overloaded method. Copies the data, as raw counts, from an OlBuffer object into a user-declared array of bytes.
	GetDataAsRawInt16	Overloaded method. Used when the resolution of the subsystem is 16 bits or less and when the data encoding is twos complement, copies the data, as raw counts, from an OlBuffer object into a user-declared array of signed, 16-bit integers.

Table 52: Members of the OlBuffer Class (cont.)

Member Type	Member Name	Description
Methods (cont.)	GetDataAsRawUInt16	Overloaded method. Used when the resolution of the subsystem is 16 bits or less and when the data encoding is binary, copies the data, as raw counts, from an OlBuffer object into a user-declared array of unsigned, 16-bit integers.
	GetDataAsRawUInt32	Overloaded method. Used when the resolution of the subsystem is greater than 16 bits, copies the data, as raw counts, from an OlBuffer object into a user-declared array of unsigned 32-bit integers.
	GetDataAsRpm	For a specified ChannelListEntry, converts the tachometer data from the internal buffer of an OlBuffer object into RPM values, and then copies these values into a user-declared array of 64-bit floating-point (double) values.
	GetDataAsSensor	Overloaded method. Converts the data from an OlBuffer object into sensor values using the SupportedChannelInfo.SensorGain and SupportedChannelInfo. SensorOffset values and copies this data into a user-declared array of floating-point (double) values.
	GetDataAsVolts	Overloaded method. Converts the data from an OlBuffer object into voltages, and copies this data into a user-declared array of floating-point values.
	GetDataAsVoltsByte	For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into voltage values, and then copies these voltage values into a user-declared array of bytes. Each voltage value is stored as an Int32, and takes 4 bytes.
	PutDataAsRaw	Overloaded method that copies raw counts from a user-specified array into an OlBuffer object.
	PutDataAsVolts	Overloaded method that copies voltage values from a user-specified array into an OlBuffer object.
	Reallocate	Reallocates the OlBuffer object to the specified number of samples. The existing internal data buffer is deallocated and any data that it contained is lost.

BufferQueue Class

The BufferQueue class provides the properties and methods listed in Table 53 for managing a queue of OlBuffer objects for a continuous I/O operation.

Table 53: Members of the BufferQueue Class

Member Type	Member Name	Description
Read-Only Properties	InProcessCount	Returns the number of OlBuffer objects that have been taken from the queue and sent to the device for processing.
	QueuedCount	Returns the number of olBuffer objects that are on the subsystem queue (OlBuffer objects are in the queued state).
Methods	DequeueBuffer	Removes the OlBuffer object at the front of the queue, and returns it to the user.
	FreeAllQueuedBuffers	Removes all OlBuffer objects from the subsystem queue and deallocates them.
	QueueBuffer	Adds an OlBuffer object to the queue for the subsystem.

You can access a BufferQueue object through the following classes:

- AnalogInputSubsystem, described on page 112
- AnalogOutputSubsystem, described on page 117

Event Handling

The following classes are provided for handling events raised by the OpenLayers.DeviceCollection namespace:

- GeneralEventArgs, described on page 131
- BufferDoneEventArgs, described on page 132
- DriverRunTimeErrorEventArgs, described on page 132
- IOCompleteEventArgs, described on page 132

GeneralEventArgs

The GeneralEventArgs class provides the properties listed in Table 54 to return information about DT-Open Layers events. This object is generated internally and is returned to event delegates. Refer to page 134 for more information on delegates.

Table 54: Members of the GeneralEventArgs Class

Member Type	Member Name	Description
Read-Only Properties	DateTime	Returns the time stamp of when the associated event occurred.
	Subsystem	Returns the subsystem (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder) that raised the event.

BufferDoneEventArgs

The BufferDoneEventArgs class inherits the members of the GeneralEventArgs class and adds the **OlBuffer** property. When the BufferDoneEvent event is raised, the completed buffer is returned in the **OlBuffer** property.

This object is generated internally and returned to the **BufferDoneHandler** delegate. Refer to page 134 for more information on delegates.

DriverRunTimeErrorEventArgs

The DriverRunTimeErrorEventArgs class inherits the members of the GeneralEventArgs class and adds the properties listed in Table 55 to return data related to the event DriverRunTimeErrorEvent.

This object is generated internally and returned to the **DriverRunTimeErrorEventHandler** delegate. Refer to page 134 for more information on delegates.

Table 55: Members of the DriverRunTimeErrorEventArgs Class

Member Type	Member Name	Description
Read-Only Properties	ErrorCode	Returns the error code that is associated with the driver error. Refer to Appendix A for more information.
	Message	Returns a descriptive string associated with the error code. Refer to Appendix A for more information.

IOCompleteEventArgs

The IOCompleteEventArgs class inherits the members of the GeneralEventArgs class and adds the properties listed in Table 56 to return data related to the event IOCompleteEvent.

This object is generated internally and returned to the **IOCompleteHandler** delegate. Refer to page 134 for more information on delegates.

Table 56: Members of the IOCompleteEventArgs Class

Member Type	Member Name	Description
Read-Only Properties	LastSampleNumber	For analog input operations only, returns the total number of samples per channel that were acquired from the time acquisition was started (with the start trigger) to the last post-trigger sample. For example, a value of 100 indicates that a total of 100 samples (samples 0 to 99) were acquired. You can subtract the value of the **AnalogInputSubsystem.ReferenceTrigger. PostTriggerScanCount** property, described on page 322, from the value of this property to determine when the reference trigger occurred and the number of pre-trigger samples that were acquired. For example, if the value of this property is 100, and you specified a value of 75 for the post-trigger scan count, you can determine that the reference trigger occurred at sample count 25 (100-75) of the last buffer; samples 25 through 99 are post-trigger samples and samples 0 to 24 are pre-trigger samples.

Error Handling

The following classes are provided for handling errors that may occur in the OpenLayers.Base namespace:

- OlException, described below
- OlError, described on page 134

OlException

The OlException class provides the properties listed in Table 57 for dealing with errors that can be generated by the OpenLayers.DeviceCollection namespace.

Table 57: Members of the OlException Class

Member Type	Member Name	Description
Read-Only Properties	ErrorCode	Returns the error code from the DT-Open Layers for .NET Class Library that is associated with this exception.
	Message	Returns the descriptive string for the exception.
	Subsystem	Returns the subsystem (AnalogInput, AnalogOutput, DigitalInput, DigitalOutput, CounterTimer, Tachometer, or QuadratureDecoder) that raised the exception. If the exception is not related to a specific subsystem, returns null.

OlError

The OlError class provides the OlError constructor for encapsulating an DT-Open Layers error code. The OlError class provides the methods listed in Table 58 for getting information about errors returned by the DT-Open Layers for .NET Class Library. Refer to Appendix A for a list of errors that may be returned by the DT-Open Layers for .NET Class Library.

Table 58: Members of the OlError Class

Member Type	Member Name	Description
Methods	GetErrorCode	Returns the error code that is associated with a specified error message in the DT-Open Layers for .NET Class Library.
	GetErrorString	Returns a description for the specified error code in the DT-Open Layers for .NET Class Library.

Delegates

DT-Open Layers events are reported to user-specified callback routines using the .NET delegates listed in Table 33.

Table 59: Delegates Included in the OpenLayers.DeviceCollection Namespace

Delegate Name	Description
BufferDoneHandler	When the event BufferDoneEvent occurs, returns the subsystem that generated the event and the BufferDoneEventArgs object that is associated with the event.
DeviceRemovedHandler	When the event DeviceRemovedEvent occurs, returns the subsystem that generated the event and the GeneralEventArgs object that is associated with the event.
DriverRunTimeErrorEventHandler	When the event DriverRunTimeErrorEvent occurs, returns the subsystem that generated the event and the DriverRunTimeErrorEventArgs object that is associated with the event.
GeneralFailureHandler	When the event GeneralFailureEvent occurs, returns the subsystem that generated the event and the GeneralEventArgs object that is associated with the event.
IOCompleteHandler	When the event IOCompleteEvent occurs, returns the subsystem that generated the event and the IOCompleteEventArgs object that is associated with the event.
PreTriggerBufferDoneHandler	When the event PreTriggerBufferDoneEvent occurs, returns the subsystem that generated the event and the BufferDoneEventArgs object that is associated with the event.

Table 59: Delegates Included in the OpenLayers.DeviceCollection Namespace (cont.)

Delegate Name	Description
QueueDoneHandler	When the event QueueDoneEvent occurs, returns the subsystem that generated the event and the GeneralEventArgs object that is associated with the event.
QueueStoppedHandler	When the event QueueStoppedEvent occurs, returns the subsystem that generated the event and the GeneralEventArgs object that is associated with the event.

Enumerations

Table 60 lists the enumerations that are used by the properties and/or methods in the OpenLayers.Base namespace.

Table 60: Enumerations Included in the OpenLayers.Device Collection Namespace

Enumeration Name	Values	Description
ChannelDataType	Int16	Signed, 16-bit values.
	Uint16	Unsigned, 16-bit values.
	Int32	Signed, 32-bit values.
	Float	32-bit floating-point values.
	Double	64-bit, floating-point (double-bit) values.
ChannelType	SingleEnded	Channel is configured for single-ended connections.
	Differential	Channel is configured for differential connections.
ClockSource	Internal	Internal clock source.
	External	External clock source.
CouplingType	DC	DC coupling, where the DC offset is included.
	AC	AC coupling, where the DC offset is removed.
DataFlow	Continuous	Continuous I/O operation.
	SingleValue	Single-value I/O operation.
	ContinuousPreTrigger	Continuous pre-trigger input operation.
	ContinuousPrePost Trigger	Continuous about-trigger operation.
Encoding	Binary	Binary data encoding.
	TwosComplement	Twos complement data encoding.
ErrorCode	See Appendix A.	The error codes that can be returned by the library.

Table 60: Enumerations Included in the OpenLayers.Device Collection Namespace (cont.)

Enumeration Name	Values	Description
ExcitationCurrentSource	Internal	Internal excitation current source.
	External	External excitation current source.
	Disabled	Excitation current source is disabled (no excitation is applied).
IOType	VoltageIn	The channel supports a voltage input.
	VoltageOut	The channels supports a voltage output.
	DigitalInput	The channel supports a digital input.
	DigitalOutput	The channel supports a digital output.
	QuadratureDecoder	The channel supports quadrature decoder operations.
	CounterTimer	The channel supports counter/timer operations.
	Tachometer	The channel supports tachometer input.
	Current	The channel supports a current input.
	Thermocouple	The channel supports a thermocouple input.
	Rtd	The channel supports an RTD input.
	StrainGage	The channel supports a strain gage input.
	Accelerometer	The channel supports an IEPE (accelerometer) input.
	Bridge	The channel supports a bridge-based sensor or general-purpose bridge input.
	Thermistor	The channel supports a thermistor input.
	Resistance	The channel supports a resistance measurement input.
	MultiSensor	The channel supports more than one sensor type. Use the **SupportedChannelInfo.MultiSensorType** property or the **SupportedChannelInfo. SupportedMutliSensorTypes** property to determine which sensor types are supported for the channel.
OlBuffer.BufferState	Idle	Buffer is allocated but not queued to a subsystem.
	Queued	Buffer is queued to a subsystem.
	InProcess	Buffer is queued to a device driver.
	Completed	Buffer has been completed by the driver and is not queued to a subsystem.
	Released	Buffer has been released.
ReferenceTriggerType	None	Triggering is disabled.
	TTLPos	An external digital (TTL) signal attached to the device. The trigger occurs when the device detects a transition on the rising edge of the digital TTL signal.
	DigitalEvent	A trigger is generated when an external digital event occurs.

Table 60: Enumerations Included in the OpenLayers.Device Collection Namespace (cont.)

Enumeration Name	Values	Description
ReferenceTriggerType (cont.)	TTLNeg	An external digital (TTL) signal attached to the device. The trigger occurs when the device detects a transition on the falling edge of the digital TTL signal.
	ThresholdPos	Either an analog signal from an analog input channel or an external analog signal attached to the device. A positive analog threshold trigger occurs when the device detects a positive-going signal that crosses a threshold value. The threshold level is generally set using an analog output subsystem on the device.
	ThresholdNeg	Either an analog signal from an analog input channel or an external analog signal attached to the device. A negative analog threshold trigger occurs when the device detects a negative-going signal that crosses a threshold value. The threshold level is generally set using an analog output subsystem on the device.
SubsystemBase.States	Initialized	The subsystem has been initialized but has not been configured.
	ConfiguredForSingle Value	The subsystem has been configured for a single-value operation.
	ConfiguredFor Continuous	The subsystem has been configured for a continuous operation.
	PreStarted	The subsystem has been prestarted for a simultaneous operation.
	Running	The operation on the subsystem is running.
	Stopping	The operation on the subsystem is being stopped.
	Aborting	The operation on the subsystem is being aborted.
	IoComplete	The I/O operation on the subsystem is done.
SubsystemType	AnalogInput	Analog input subsystem
	AnalogOutput	Analog output subsystem
	DigitalInput	Digital input subsystem
	DigitalOutput	Digital output subsystem
	QuadratureDecoder	Quadrature decoder subsystem
	CounterTimer	Counter/timer subsystem
	Tachometer	Tachometer subsystem

Table 60: Enumerations Included in the OpenLayers.Device Collection Namespace (cont.)

Enumeration Name	Values	Description
TriggerType	Software	Trigger is generated when the operation is started in software.
	TTLPos	Trigger is generated on a rising edge of an external, digital (TTL) signal.
	DigitalEvent	Trigger is generated when an external digital event occurs.
	TTLNeg	Trigger is generated on a falling edge of an external, digital (TTL) signal.
	ThresholdPos	Trigger is generated when a positive-going analog signal crosses a threshold value.
	ThresholdNeg	Trigger is generated when a negative-going analog signal crosses a threshold value.

Structures

The OpenLayers.DeviceCollection namespace provides the following structures:

- **HardwareInfo** structure – This structure is used by the **Device.GetHardwareInfo** method to return information about a DT-Open Layers-compliant device.

  Table 35 lists the fields that are contained in the **HardwareInfo** structure.

Table 61: Fields of the HardwareInfo Structure in the OpenLayers.DeviceCollection Namespace

Field	Description
CollectionId	This field contains a unique collection ID value that is generated using the addition of the last two digits of the year multiplied by 10e7 (0 - 990000000) and the number of minutes into the current year (0 - 527040). Valid values for this field are 0 - 990527040.
NumberofDevices	This field contains the number of devices in the collection. .
VendorId	The identification number of the vendor. For most devices, this is 0x087 hexadecimal.

- **SingleValuesInfoRaw** structure – Used with the **SetSingleValuesAsRaw** method, specifies the analog output channel to update and the raw count to output on that channel.

  Table 62 lists the fields that are contained in the **SingleValuesInfoRaw** structure.

Table 62: Fields of the SingleValuesInfoRaw Structure

Field	Description
PhysicalChannel	The number of the physical analog output channel to update.
RawValue	The raw count value to output on the specified analog output channel.

- **SingleValuesInfoVolts** structure – Used with the **SetSingleValuesAsVolts** method, specifies the analog output channel to update and the voltage value to output on that channel.

  Table 63 lists the fields that are contained in the **SingleValuesInfoVolts** structure.

Table 63: Fields of the SingleValuesInfoVolts Structure

Field	Description
PhysicalChannel	The number of the physical analog output channel to update.
Voltage	The voltage value to output on the specified analog output channel.

3

Using the OpenLayers.Base Namespace

Overview

To perform a data acquisition operation on a DT-Open Layers-compliant device, you need to do the following:

1. Import the namespace into your program.
2. Get a DeviceMgr object to manage DT-Open Layers devices.
3. Get a Device object for each DT-Open Layers device that you want to use.
4. Get a subsystem of each type that you want to use.
5. Determine what channels are supported on each subsystem, and set up channel parameters.
6. Set up and configure the subsystem.
7. Perform the I/O operations.
8. Start subsystems simultaneously, if supported.
9. Auto-calibrate the subsystem, if supported.
10. Handle events.
11. Handle errors.
12. When finished, clean up the memory and resources used by the operations.

The remaining sections in this chapter describe these steps in detail.

Importing the Namespace for the Library

To use any of the classes in the OpenLayers.Base namespace, you first need to import the namespace into your program, as follows:

Visual C#

```
using OpenLayers.Base
```

Visual Basic

```
Imports OpenLayers.Base
```

Getting a DeviceMgr Object

Before performing any operation using the OpenLayers.Base namespace, you must first use the **DeviceMgr.Get** method to return a DeviceMgr object. The DeviceMgr object is responsible for managing all DT-Open Layers devices in your system.

The following examples shows how to get a DeviceMgr object:

Visual C#

```
DeviceMgr deviceMgr = DeviceMgr.Get();
```

Visual Basic

```
deviceMgr As DeviceMgr = DeviceMgr.Get()
```

Getting a Device Object

Once you have a DeviceMgr object, use the **DeviceMgr.GetDevice** method to return a Device object for each DT-Open Layers device that you want to use.

Note: If you wish, you can also create a Device object using the **Device** constructor instead of using the **GetDevice** method.

The following examples shows how to get a Device object for the device named *deviceName*:

Visual C#
```
Device device = deviceMgr.GetDevice (deviceName);
```

Visual Basic
```
device As Device = deviceMgr.GetDevice(deviceName)
```

You can determine if a DT-Open Layers-compatible device is plugged into your system by using the **DeviceMgr.HardwareAvailable** method. If this method returns True, at least one DT-Open Layers-compatible device is plugged into your system.

To determine the names of all DT-Open Layers-compatible devices plugged into your system, use the **DeviceMgr.GetDeviceNames** method.

You can also use the use the following properties and/or methods to return information about the specified Device object:

- **Device.BoardModelName** property – Returns the model name of the device.
- **Device.DeviceName** property – Returns the user-defined name for the device. You can modify this name using the DT-Open Layers Control Panel applet.
- **Device.DriverName** property – Returns the name of the Windows device driver for the device.
- **Device.DriverVersion** property – Returns the version of the Windows device driver for the device.
- **Device.GetHardwareinfo** method – Returns the driver id, product id, board id, and vendor id for the specified device. See page 106 for more information on these fields.

Getting a Subsystem

The following subsystem types are defined in the OpenLayers.Base namespace:

- AnalogInputSubsystem – This subsystem type represents the analog input channels of your device, if supported. Use this subsystem type if you want to acquire data from the analog input channels.

  If your device supports streaming digital input, counter/timer, and or quadrature decoder data through the analog input subsystem, use AnalogInputSubsystem to read this data.
- AnalogOutputSubsystem – This subsystem type represents the analog output channels of your device, if supported. Use this subsystem type if you want to update the values of the analog output channels.

  If your device supports streaming digital output data through the analog output subsystem, use AnalogOutputSubsystem to update the data on the digital output ports.
- DigitalInputSubsystem – This subsystem type represents the digital input lines of your device, if supported. Use this subsystem type if you want to read the values of the digital input lines on your device.

  If your device supports it, you can also use DigitalInputSubsystem to generate an interrupt when a digital input line changes state.
- DigitalOutputSubsystem – This subsystem type represents the digital output lines of your device, if supported. Use this subsystem type if you want to update the values the digital output lines.
- CounterTimerSubsystem – This subsystem type represents the counter/timer channels of your device, if supported. Use this subsystem type if you want to read the value of counter or output pulses from the counter under various conditions.
- TachSubsystem – This subsystem type represents the tachometer input channels of your device, if supported. Use this subsystem type if you want to read the value of a tachometer measurement.
- QuadratureDecoderSubsystem – This subsystem type represents the quadrature decoder channels of your device, if supported. Use this subsystem type if you want to perform quadrature decoder operations.

Your device may support all or a subset of these functions or subsystem types. In addition, your device may support multiple instances, called elements, of the same subsystem type. 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 your device has two digital input ports, two subsystems of type DigitalInputSubsystem are available, differentiated as elements 0 and 1.

Once you have a Device object, you need to get a subsystem of the appropriate type for each subsystem element that you want to use. While you can do this using the constructor provided in each subsystem class, it is recommended that you use one of the following methods of the Device class:

- **Device.AnalogInputSubsystem** method – Returns an analog input subsystem for a specified element and Device object. Most DT-Open Layers devices group all the analog input channels into one analog input subsystem element (0). However, some devices, like the DT9820 Series, provide one element per A/D converter.

  The following example shows how to get an AnalogInputSubsystem object for element 0:

  Visual C#
  ```
  AnalogInputSubsystem ainSS = device.AnalogInputSubsystem (0);
  ```
  Visual Basic
  ```
  ainSS As AnalogInputSubsystem = device.AnalogInputSubsystem(0)
  ```
- **Device.AnalogOutputSubsystem** method – Returns an analog output subsystem for a specified element and Device object. Most DT-Open Layers devices group all the analog output channels into one analog output subsystem element (0). The following example shows how to get an AnalogOutputSubsystem object for element 0:

  Visual C#
  ```
  AnalogOutputSubsystem aoutSS = device.AnalogOutputSubsystem (0);
  ```
  Visual Basic
  ```
  aoutSS As AnalogOutputSubsystem = device.AnalogOutputSubsystem(0)
  ```
- **Device.DigitalInputSubsystem** method – Returns a digital input subsystem for a specified element and Device object. Most DT-Open Layers devices provide one digital input subsystem element for each digital input port. The following example shows how to get a DigitalInputSubsystem object for element 0:

  Visual C#
  ```
  DigitalInputSubsystem dinSS = device.DigitalInputSubsystem (0);
  ```
  Visual Basic
  ```
  dinSS As DigitalInputSubsystem = device.DigitalInputSubsystem(0)
  ```
- **Device.DigitalOutputSubsystem** method – Returns a digital output subsystem for a specified element and Device object. Most DT-Open Layers devices provide one digital output subsystem element for each digital output port. The following example shows how to get a DigitalOutputSubsystem object for element 0:

  Visual C#
  ```
  DigitalOutputSubsystem doutSS = device.DigitalOutputSubsystem (0);
  ```
  Visual Basic
  ```
  doutSS As DigitalOutputSubsystem =
    device.DigitalOutputSubsystem(0)
  ```

- **Device.CounterTimerSubsystem** method – Returns a counter/timer subsystem for a specified element and Device object. Most DT-Open Layers devices provide one counter/timer subsystem element for each counter/timer channel. The following example shows how to get a CounterTimerSubsystem object for element 0:

  Visual C#
  ```
  CounterTimerSubsystem ctSS = device.CounterTimerSubsystem (0);
  ```
  Visual Basic
  ```
  ctSS As CounterTimerSubsystem = device.CounterTimerSubsystem(0)
  ```
- **Device.TachSubsystem** method – Returns a tachometer subsystem for a specified element and Device object. Most DT-Open Layers devices provide one tachometer subsystem element for each tachometer input channel. The following example shows how to get a TachSubsystem object for element 0:

  Visual C#
  ```
  TachSubsystem tachSS = device.TachSubsystem (0);
  ```
  Visual Basic
  ```
  tachSS As TachSubsystem = device.TachSubsystem(0)
  ```
- **Device.QuadratureDecoderSubsystem** method – Returns a quadrature decoder subsystem for a specified element and Device object. Most DT-Open Layers devices provide one quadrature decoder subsystem element for each quadrature decoder channel. The following example shows how to get a QuadratureDecoderSubsystem object for element 0:

  Visual C#
  ```
  QuadratureDecoderSubsystem quadSS =
   device.QuadratureDecoderSubsystem (0);
  ```
  Visual Basic
  ```
  quadSS As QuadratureDecoderSubsystem =
    device.QuadratureDecoderSubsystem(0)
  ```

You can determine the type of a specified subsystem by using the **SubsystemTyp**e property within the appropriate subsystem class.

To return the number of elements supported by a specified subsystem type on a specified device, use the **Device.GetNumSubsystemElements** method.

You can determine the state of a subsystem using the **State** property within the appropriate subsystem class. The following states have been defined:

- Initialized – The subsystem has been initialized, but not configured.
- ConfiguredForSingleValue – The subsystem has been configured for a single-value operation.
- ConfiguredForContinuous – The subsystem has been configured for a continuous operation.

- Running – The subsystem is running.

Note: You can also use the **IsRunning** property within the appropriate subsystem class to determine if the subsystem is running.

- Stopping – The operation on the subsystem is in the process of stopping.
- Aborting – The operation on the subsystem is in the process of being aborted.
- Prestarted – The subsystem has been prestarted for a continuous simultaneous operation.
- IOComplete – For analog input subsystems, the final post-trigger samples has been copied to the user buffer. For analog output subsystems, the final analog output sample has been written from the FIFO on the device; this is a transient state, which may not be seen, but does occur.

Determining the Available Channels and Setting up Channel Parameters

When you get a subsystem of a specified type, the software automatically determines the number of available channels for the subsystem and creates a SupportedChannelInfo object for each channel. The SupportedChannelInfo object contains the following information:

- physical channel number
- logical channel number
- logical channel word
- channel name
- I/O type
- Information that pertains to voltage input channels:
  - termination resistor
  - sensor gain
  - sensor offset
- Information that pertains to thermocouple channels:
  - thermocouple type
  - CJC channel
- Information that pertains to RTD channels:
  - RTD type
  - resistor value (R0) for an RTD
  - coefficient A value for an RTD
  - coefficient B value for an RTD
  - coefficient C value for an RTD
  - sensor wiring configuration
- Information that pertains to thermistor channels:
  - coefficient A value for a thermistor
  - coefficient B value for a thermistor
  - coefficient C value for a thermistor
  - sensor wiring configuration
- Information that pertains to resistance measurement channels:
  - sensor wiring configuration
  - excitation current source
  - value for the internal excitation current source

- Information that pertains to accelerometer (IEPE) channels:
  - coupling
  - excitation current source
  - value for the internal excitation current source
- Information that pertains to bridge-based sensors:
  - bridge configuration
  - bridge transducer capacity
  - bridge transducer rated output
  - strain gage lead wire resistance
  - strain gage nominal resistance
  - strain gage offset nulling value in volts
  - strain gage shunt calibration resistor (enabled or disabled)
  - strain gage shunt calibration value
- Information that pertains to strain gage channels:
  - strain gage bridge configuration
  - strain gage poisson ratio
  - strain gage lead wire resistance
  - strain gage gage factor
  - strain gage nominal resistance
  - strain gage offset nulling value in volts
  - strain gage shunt calibration resistor (enabled or disabled)
  - strain gage shunt calibration value

To get a collection of SupportedChannelInfo objects, use the SupportedChannels class.

You can get the SupportedChannelInfo object for a specific channel using the **SupportedChannels.GetChannelInfo** method and any one of the following arguments:

- The physical channel number.
- The user-defined name of the channel.
- The subsystem type and logical channel number.
- The subsystem type, logical channel number, and logical channel word.

You can also use the **SupportedChannels.Item** ([]) property to return the SupportedChannelsInfo object at a specific index.

The following subsections describe the elements of the SupportedChannelsInfo class in more detail.

Physical and Logical Channels

The logical channel number, which is zero-based, maps the physical channel to the channel's subsystem type. If the channels are native to the AnalogInputSubsystem, the logical channel number is the same as the physical channel number. If channels from other subsystem types are accessible through the AnalogInputSubsystem, the logical channel numbers are not the same as the physical channel numbers for the non-native channels.

For example, in Table 64, the SupportedChannels object for the analog input subsystem contains 8 analog input channels, four digital input ports, and two 32-bit counter/timers. As you can see, physical channels 0 to 7 map to logical channels 0 to 7 of the analog input subsystem, physical channels 8 to 11 map to logical channels 0 to 3 of the digital input subsystem, and physical channels 12 to 15 map to logical channels 0 and 1 of the counter/timer subsystem (in this case, since each counter is 32-bits, one logical channel maps to two 16-bit physical channels).

Table 64: Example of Logical and Physical Channels in a SupportedChannels Object for an Analog Input Subsystem

Subsystem Type	Logical Channel Number	Physical Channel Number
Analog Input	0	0
	1	1
	2	2
	3	3
	4	4
	5	5
	6	6
	7	7
Digital Input	0	8
	1	9
	2	10
	3	11
Counter/Timer	0	12
	0	13
	1	14
	1	15

You can determine the number of a physical channel for a given subsystem using the **SupportedChannelInfo.PhysicalChannelNumber** property.

You can determine the number of a logical channels for a given subsystem using the **SupportedChannelInfo.LogicalChannelNumber** property.

To reference a channel by number, specify either the physical channel number or the subsystem type and logical channel number.

Logical Channel Word

For channels like 32-bit counter/timers that return multi-word data, the logical channel word, which is zero-based, maps the physical channel to the data word that it returns. For example, looking at the counter/timer subsystem type in Table 65, physical channel 12 has a logical channel word of 0, indicating that this channel returns the first 16-bits of data. Physical channel 13 has a logical channel word of 1, indicating that this channel returns the second 16-bits of data.

For channels that do not return multi-word data, the value of the logical channel word is -1.

Table 65: Example of Logical and Physical Channels in a SupportedChannels Object or an Analog Input Subsystem

Subsystem Type	Logical Channel Number	Physical Channel Number	Logical Channel Word
Analog Input	0	0	–1
	1	1	–1
	2	2	–1
	3	3	–1
	4	4	–1
	5	5	–1
	6	6	–1
	7	7	–1
Digital Input	0	8	–1
	1	9	–1
	2	10	–1
	3	11	–1
Counter/Timer	0	12	0
	0	13	1
	1	14	0
	1	15	1

You can determine the value of the logical channel word for a given channel using the **SupportedChannelInfo.LogicalChannelWord** property.

To reference a channel by logical channel word, specify the subsystem type, logical channel number, and logical channel word.

Channel Name

By default, each channel that is listed in the SupportedChannelInfo class has a name that describes the subsystem type and includes the logical channel number and logical channel word, if applicable. Examples of default names include Ain0 for analog input channel 0, Aout1 for analog output channel 1, Din0 for digital input channel 0, Dout2 for digital output channel 2, CT0 Word 1 for counter/timer channel 0 (word 1), and Quad1 Word 0 for quadrature decoder channel 1 (word 0).

You can specify your own name for a channel using the **SupportedChannelInfo.Name** property.

To reference a channel by name, specify the name of the channel.

IOType

You can determine what kind of I/O operation is supported for a particular channel of a given subsystem using the **SupportedChannelInfo.IOType** property.

This property returns one of the following I/O types:

- VoltageIn – Refer to page 155 for information on setting up additional parameters for this channel I/O type.
- VoltageOut
- DigitalInput
- DigitalOutput
- QuadratureDecoder
- CounterTimer
- Tachometer
- Current – Refer to page 157 for information on setting up additional parameters for this channel I/O type.
- Thermocouple – Refer to page 157 for information on setting up additional parameters for this channel I/O type.
- Rtd – Refer to page 160 for information on setting up additional parameters for this channel I/O type.
- StrainGage – Refer to page 162 for information on setting up additional parameters for this channel I/O type.
- Accelerometer – Refer to page 167 for information on setting up additional parameters for this channel I/O type.

- Bridge – Refer to page 168 for information on setting up additional parameters for this channel I/O type.
- Thermistor – Refer to page 172 for information on setting up additional parameters for this channel I/O type.
- Resistance – Refer to page 173 for information on setting up additional parameters for this channel I/O type.
- MultiSensor – A MultiSensor I/O type means that the channel supports multiple sensors types. You must specify the sensor type that is connected to the channel using the **SupportedChannelInfo.MultiSensorType** property, as described in the following subsections.

Setting Up Voltage Input Channels

To determine whether a specific channel supports voltage inputs or multiple sensor types, use the **SupportedChannelInfo.IOType** property. If the value of **IOType** is MultiSensor, you must set the multisensor type to VoltageIn using the **SupportedChannelInfo.MultiSensorType** property to use the channel for voltage measurements.

Note: You can read a single voltage value from one channel using the **AnalogInputSubsystem. GetSingleValueAsVolts** method. If the analog input subsystem supports simultaneous operations (**AnalogInputSubsystem. SupportsSimultaneousSampleHold** is True), you can read a single voltage value from all channels using the **AnalogInputSubsystem.GetSingleValuesAsVolts** method. Refer to page 176 for more information.

If you are acquiring data to a buffer, you can read the voltage value from the specified channels using the **OlBuffer.GetDataAsVolts** method. Refer to page 223 for more information.

Termination Resistor

Some voltage input channels support a bias return termination resistor. To determine if the channel supports input termination, use the **SupportedChannelInfo.SupportsInputTermination** property.

The bias return termination resistor is typically enabled for floating and grounded voltage sources. It is typically disabled for voltage sources with grounded references. Refer to the documentation for your device for wiring information.

You can enable or disable the bias return termination resistor using the **SupportedChannelInfo.InputTerminationEnabled** property. If this property is True, the termination resistor is enabled. If this property is False, the termination resistor is not used.

Sensor Gain and Offset

If you want to read a value from a channel in engineering units, like pressure, and your channel supports voltage measurements only, you can specify the gain and offset for the sensor using the **SupportedChannelInfo.SensorGain** and **SupportedChannelInfo.SensorOffset** properties.

Note: If the channel supports an I/O type other than voltage, such as thermocouple, RTD, thermistor, resistance, current, strain gage, or bridge, use the properties specific to these I/O types instead of the sensor gain and offset. For example, if you want to read a temperature value from a thermocouple input, use the **ThermocoupleType** property, described on page 158, instead of the sensor gain and offset.

The sensor gain and offset are used to scale a sample from raw counts to a sensor format. The scaling occurs in two steps. First, the raw count value is converted to prescaled voltage using the gain applied to the input signal. Then, the prescaled voltage is scaled using the following equation:

```
y = mx + b
```

where *y* is the scaled sensor value, *m* is the sensor gain, *x* is the prescaled value in voltage, and *b* is the sensor offset.

The following example shows how to set the sensor gain and offset of channel 0 of the analog input subsystem using the SupportedChannels object:

Visual C#

```
SupportedChannelInfo Ch0Info =
  ainSS.SupportedChannels.GetChannelInfo(
    SubsystemType.AnalogInput,0);
.
.
// Set the sensor gain and offset
Ch0Info.SensorGain = 2;
Ch0Info.SensorOffset = 10;
```

Visual Basic

```
Dim Ch0Info As SupportedChannelInfo =
  ainSS.SupportedChannels.GetChannelInfo(
    SubsystemType.AnalogInput, 0)
.
.
' Set the sensor gain and offset
Ch0Info.SensorGain = 2
Ch0Info.SensorOffset = 10
```

Setting Up Current Input Channels

To determine if the analog input subsystem supports current inputs, use the **AnalogInputSubsystem. SupportsCurrent** property.

If this value is True, determine whether the specific channel supports current or multiple sensor types using the **SupportedChannelInfo.IOType** property. If the value of **IOType** is MultiSensor, you must set the multisensor type to Current using the **SupportedChannelInfo.MultiSensorType** property to use the channel for current measurements.

Some current channels support a bias return termination resistor. To determine if the channel supports input termination, use the **SupportedChannelInfo.SupportsInputTermination** property.

The bias return termination resistor is typically enabled for floating and grounded current sources. It is typically disabled for current sources with grounded references. Refer to the documentation for your device for wiring information.

You can enable or disable the bias return termination resistor using the **SupportedChannelInfo.InputTerminationEnabled** property. If this property is True, the termination resistor is enabled. If this property is False, the termination resistor is not used.

Note: You can read a single current value from one channel using the **AnalogInputSubsystem.GetSingleValueAsCurrent** method. Refer to page 176 for more information.

If you are acquiring data to a buffer, you can read the current value from the specified channels using the **OlBuffer.GetDataAsCurrent** method. Refer to page 222 for more information.

Setting Up Thermocouple Input Channels

To determine if the analog input subsystem supports thermocouple inputs, use the **AnalogInputSubsystem.SupportsThermocouple** property.

If this value is True, determine whether the specific channel supports thermocouple inputs or multiple sensor types using the **SupportedChannelInfo.IOType** property. If the value of **IOType** is MultiSensor, you must set the multisensor type to Thermocouple using the **SupportedChannelInfo.MultiSensorType** property to use the channel for thermocouple measurements.

For channels that support thermocouples, you can set the following properties:

- Thermocouple input type, described on this page
- CJC source, described on page 159

Note: You can read a single temperature value from one channel using the **AnalogInputSubsystem. GetSingleValueAsTemperature** method. If the analog input subsystem supports simultaneous operations (**AnalogInputSubsystem.SupportsSimultaneousSampleHold** is True), you can read a single temperature value from all channels using the **AnalogInputSubsystem.GetSingleValuesAsTemperature** method. Refer to page 176 for more information.

If you are acquiring data to a buffer, you can read the temperature from the specified channels using the **OlBuffer.GetDataAsTemperatureByte** or **OlBuffer.GetDataAsTemperatureDouble** method, depending on whether your device returns temperature values as integer or floating-point (4 byte) values. To determine if your subsystem returns floating-point values, use the **AnalogInputSubsystem. ReturnsFloats** property. Refer to page 223 for more information.

Thermocouple Input Types

If the subsystem supports thermocouple inputs, specify the type of thermocouple that is connected to the input channel using the **SupportedChannelInfo.ThermocoupleType** property. The following thermocouple types are defined:

- None – Specifies voltage rather than temperature
- J – Specifies a J thermocouple type
- K – Specifies a K thermocouple type
- B – Specifies a B thermocouple type
- E – Specifies a E thermocouple type
- N – Specifies a N thermocouple type
- R – Specifies a R thermocouple type
- S – Specifies a S thermocouple type
- T – Specifies a T thermocouple type

If the thermocouple type is set to None, data is returned in voltage rather than temperature. If the thermocouple type is set for any of the other defined thermocouple types, the data is returned in temperature; you can specify the units as degrees C, F, or K.

Note: If the **AnalogInputSubsystem. SupportsTemperatureDataInStream** is True, each channel's **ThermocoupleType** property is set to the value that was stored on the device when the subsystem was last configured. If **AnalogInputSubsystem.SupportsTemperature DataInStream** is False, the default thermocouple type is J.

You can get the temperature range for a specified thermocouple type using the **Utility.GetThermocoupleRange** method.

CJC Sources

Some devices do temperature conversion hardware based on the value of an internal CJC (cold junction compensation) channel. Every sample in the data stream corresponds to a single (typically, floating-point) value that represents either the temperature (in degrees C) or the voltage of the input channel, based on its thermocouple type.

Other devices return A/D input values as raw counts and the DT-Open Layers for .NET Class Library converts these values into temperatures or voltage based on the thermocouple input type and the value of the CJC channel.

To determine if your subsystem does temperature conversion in hardware, use the **AnalogInputSubsystem. SupportsTemperatureDataInStream** property. If this property returns True, temperature conversion is done by the DT-Open Layers for .NET Class Library.

To determine if the analog input subsystem supports a CJC (cold junction compensation) source that is internal to the hardware, use the **AnalogInputSubsystem. SupportsCjcSourceInternal** property. To determine if the analog input subsystem supports channels that are used for CJC, use the **AnalogInputSubsystem. SupportsCjcSourceChannel** property.

If the analog input subsystem supports one or more channels that are used for CJC, you can determine which CJC channel is associated with a specific input channel using the **SupportedChannelInfo.CjcChannel** property. This property applies only to devices that support a channel (not an internal source) as the CJC source. By default, channel 0 is used as the CJC source.

Note: Some devices that support temperature conversion in hardware also provide the option of returning CJC values in the data stream. This option is seldom used, but is provided if you want to implement your own temperature conversion algorithms in software when using continuous operations.

To determine if the subsystem supports interleaving CJC temperature values with A/D values (either voltage or temperature depending on the thermocouple type) in the data stream, use the **AnalogInputSubsystem.SupportsInterleavedCjcTemperatures InStream** property.

By default, the subsystem is disabled from returning CJC values in the data stream. To enable the subsystem to return CJC values in the data stream, use the **AnalogInputSubsystem. ReturnCjcTemperaturesInStream** property. When enabled, two (typically floating-point) values are returned in the data stream for each channel: the first value represents the temperature or voltage of the input channel (based on the thermocouple type of the input), and the second value represents the CJC temperature, in degrees C. Generally, in this configuration, a thermocouple type of None is specified for each channel; use **GetDataAsVolts**, described on page 222 to read the data). If you return CJC values in the data stream, ensure that you allocate a buffer that is twice as large to accommodate the CJC values (number of samples x 2).

Refer to page 176 and page 223 for more information on reading CJC values.

Setting Up RTD Input Channels

To determine if the analog input subsystem supports RTD inputs, use the **AnalogInputSubsystem.SupportsRTD** property.

If this value is True, determine whether the specific channel supports RTDs or multiple sensor types using the **SupportedChannelInfo.IOType** property. If the value of **IOType** is MultiSensor, you must set the multisensor type to Rtd using the **SupportedChannelInfo.MultiSensorType** property to use the channel for RTD measurements.

In an RTD measurement, the measurement device reads the voltage drop across the RTD as the resistance changes and converts the voltage to the appropriate temperature using the Callendar-Van Dusen transfer function:

$$R_T = R_0[1 + AT + BT^2 + CT^3(T - 100)]$$

where,

- R_T is the resistance at temperature.
- R_0 is the resistance at 0° C.
- A, B, and C are the Callendar-Van Dusen coefficients for a particular RTD type. (The value of C is 0 for temperatures above 0° C.)

For channels that support RTD inputs, you must specify the type of RTD that is connected to the input channel using the **SupportedChannelInfo.RTDType** property. To specify the R0 coefficient, use the **SupportedChannelInfo.RtdR0** property. To specify the A coefficient, use the **SupportedChannelInfo.RtdACoefficient** property. To specify the B coefficient, use the **SupportedChannelInfo.RtdBCoefficient** property. To specify the C coefficient, use the **SupportedChannelInfo.RtdCCoefficient** property.

Table 66 lists the values that are supported for these properties:

Table 66: Values Supported for RTD Properties

Values for the RTDType Property	Values for the RtdR0 Property (Ω)	Values for the RtdACoefficient Property	Values for the RtdBCoefficient Property	Values for the RtdCCoefficient Property
Pt3850[a] (the default)	100 (the default), 500, or 1000	3.9083×10^{-3}	-5.775×10^{-7}	-4.183×10^{-12}
Pt3920[b]	98.129	3.9787×10^{-3}	-5.869×10^{-7}	-4.167×10^{-12}
Pt3911[c]	100	3.9692×10^{-3}	-5.8495×10^{-7}	-4.233×10^{-12}
Pt3750[d]	1000	3.81×10^{-3}	-6.02×10^{-7}	-6.0×10^{-12}
Pt3916[e]	100	3.9739×10^{-3}	-5.870×10^{-7}	-4.4×10^{-12}
Pt3928[f]	100	3.9888×10^{-3}	-5.915×10^{-7}	-3.85×10^{-12}
Custom	User-defined	User-defined	User-defined	User-defined

a. Uses a Temperature Coefficient of Resistance (TCR) value of 0.003850 Ω / Ω /° C as specified in the DIN/IEC 60751 ASTM-E1137 standard.
b. Uses a TCR value of 0.003920 Ω / Ω /° C as specified in the SAMA RC21-4-1966 standard.
c. Uses a TCR value of 0.003911 Ω / Ω /° C as specified in the US Industrial Standard standard.
d. Uses a TCR value of 0.003750 Ω / Ω /° C as specified in the Low Cost standard.
e. Uses a TCR value of 0.003916 Ω / Ω /° C as specified in the Japanese JISC 1604-1989 standard.
f. Uses a TCR value of 0.003928 Ω / Ω /° C as specified in the ITS-90 standard.

If you specify a value of Pt3850 for **SupportedChannelInfo.RTDType**, you must also specify **SupportedChannelInfo.RtdR0**, unless you are using a 100 Ω RTD (the default value).

If you specify a value of Custom for **SupportedChannelInfo.RTDType**, you must specify the values for **SupportedChannelInfo.RtdR0**, **SupportedChannelInfo.RtdACoefficient, SupportedChannelInfo.RtdBCoefficient**, and **SupportedChannelInfo.RtdCCoefficient**. Otherwise, the software automatically sets the appropriate value for **SupportedChannelInfo.RtdR0**, **SupportedChannelInfo.RtdACoefficient, SupportedChannelInfo.RtdBCoefficient**, and **SupportedChannelInfo.RtdCCoefficient** based on the selected RTD type**.**

Note: If the **AnalogInputSubsystem. SupportsTemperatureDataInStream** property is True, each channel's **RTDType** property is set to the value that was stored on the device when the subsystem was last configured.

Use the **SupportedChannelInfo.SensorWiringConfiguration** property to specify the wiring configuration (two-wire, three-wire, or four-wire) for the RTD.

RTD data on most devices is represented as floating-point values (4 bytes). To determine if your subsystem returns floating-point values, use the **AnalogInputSubsystem. ReturnsFloats** property.

Note: You can read a single temperature value from one channel using the **AnalogInputSubsystem. GetSingleValueAsTemperature** method. If the analog input subsystem supports simultaneous operations (**AnalogInputSubsystem. SupportsSimultaneousSampleHold** is True), you can read a single temperature value from all channels using the **AnalogInputSubsystem. GetSingleValuesAsTemperature** method. Refer to page 176 for more information.

If you are acquiring data to a buffer, you can read the temperature from the specified channels using the **OlBuffer.GetDataAsTemperatureByte** or **OlBuffer.GetDataAsTemperatureDouble** method. Refer to page 223 for more information.

Setting Up Strain Gage Input Channels

To determine if the analog input subsystem supports strain gage measurements, use the **AnalogInputSubsystem.SupportsStrainGage** property.

If this value is True, determine whether the specific channel supports strain gage measurements or multiple sensor types using the **SupportedChannelInfo.IOType** property. If the value of **IOType** is MultiSensor, you must set the multisensor type to StrainGage using the **SupportedChannelInfo.MultiSensorType** property to use the channel for strain gage measurements.

For channels that support strain gages, you can set the following properties for the channel:

- TEDS information, if supported, as described on page 163
- Strain Gage configuration, described on page 165
- Poisson ratio, described on page 166
- Lead wire resistance, described on page 166
- Gage factor, described on page 166
- Nominal resistance, described on page 166
- Offset nulling value, described on page 166
- Shunt calibration resistor and value, described on page 167

Note: You must set the excitation source and value for the subsystem, as described on page 203.

You can read a single microstrain value from a strain gage channel using the **AnalogInputSubsystem.GetSingleValueAsStrain** method. For a device that supports simultaneous A/Ds, you can acquire a single value from each analog input channel and return the data as an array of values in microstrain using the **AnalogInputSubsystem.GetSingleValuesAsStrain** method. Refer to page 176 for more information.

If you are acquiring data to a buffer, you can read data from each strain gage, in microstrain, using the **OlBuffer.GetDataAsStrain** method. Refer to page 224 for more information.

In some cases, you may wish to use a rosette, which is arrangement of two or more closely positioned strain gage grids that are oriented to measure the normal strains along different directions in the underlying surface of a test material. The DT-Open Layers for .NET Class Library supports rectangular and delta rosettes; tee rosettes are not supported.

A rectangular rosette is an arrangement of three strain gage grids where the second grid is angularly displaced from the first grid by 45 degrees and the third grid is angularly displaced from the first grid by 90 degrees. In this arrangement, each strain gage grid (configured as a quarter bridge strain gage) corresponds to an analog input channel. You can read the strain value from each analog input channel individually, and if desired, use the utility method **Utility.ComputeRectangularRosette** to calculate the minimum and maximum principal strain values and their associated angles (in degrees).

A delta rosette is an arrangement of three strain gage grids where the second grid is angularly displaced from the first grid by 60 degrees and the third grid is angularly displaced from the first grid by 120 degrees. In this arrangement, each strain gage grid (configured as a quarter bridge strain gage) corresponds to an analog input channel. You can read the strain value from each analog input channel individually, and if desired, use the utility method **Utility.ComputeDeltaRosette** to calculate the minimum and maximum principal strain values and their associated angles (in degrees).

TEDS for Strain Gages

If your strain gage supports a TEDS interface, you can read the TEDS data from the strain gage directly using the **SupportedChannelInfo.StrainGageTeds.ReadHardwareTeds** method, or from a TEDS data file using the **SupportedChannelInfo.StrainGageTeds.ReadVirtualTeds** method.

Note: The properties in the StrainGageTeds class are read-only. It is up the application to read the value of these properties and to apply the appropriate values to the SupportedChannelInfo strain gage properties.

Table 67 lists the properties that you can read to access the TEDS data for a strain gage.

Table 67: Read-Only Properties for Accessing TEDS Data for Strain Gages

Property	Description
BridgeType	Gets the type of bridge (Full Bridge, Half Bridge, or Quarter Bridge) that was specified in the TEDS data for the channel.
CalDate	Gets the calibration date that was specified in the TEDS data for the channel.
CalibrationPeriod	Gets the calibration period that was specified in the TEDS data for the channel.
CalInitials	Gets the calibration initials that were specified in the TEDS data for the channel.
ElectricalSignalType	Gets the electrical signal type that was specified in the TEDS data for the channel.
GageArea	Gets the area of each gage element, in mm^2, that was specified in the TEDS data for the channel.
GageFactor	Gets the gage factor, or sensitivity of the strain gage, that was specified in the TEDS data for the channel.
GageResistance	Gets the initial (unstrained) gage resistance, in ohms, that was specified in the TEDS data for the channel.
GageType	Gets the type of gage that was specified in the TEDS data for the channel. Refer to page 95 for more information on the values that are defined for GageType:
IsTedsConfigured	Inherited from the TedsBase class, returns True if the TEDS data stream is read successfully; otherwise, returns False.
ManufacturerId	Inherited from the TedsBase class, gets identifying information about the manufacturer of the sensor from the TEDS data for the channel.
MaxElectricalValue	Gets the maximum electrical output, in V/V, that was specified in the TEDS data for the channel.
MaximumExcitationVoltage	Gets the maximum excitation voltage that was specified in the TEDS data for the channel.
MaxPhysicalValue	Gets the positive full-scale value, in strain, that was specified in the TEDS data for the channel.
MeasID	Gets the measurement location ID that was specified in the TEDS data for the channel.
MinElectricalValue	Gets the minimum electrical output, in V/V, that was specified in the TEDS data for the channel.
MinPhysicalValue	Gets the negative full-scale value, in strain, that was specified in the TEDS data for the channel.
ModelNumber	Inherited from the TedsBase class, gets the model number of the sensor from the TEDS data for the channel.
NominalExcitationVoltage	Gets the nominal excitation voltage that was specified in the TEDS data for the channel.

Table 67: Read-Only Properties for Accessing TEDS Data for Strain Gages (cont.)

Property	Description
PoissonCoefficient	Gets the Poisson coefficient after installation that was specified in the TEDS data for the channel.
ResponseTime	Gets the response time, in seconds, that was specified in the TEDS data for the channel.
SerialNumber	Inherited from the TedsBase class, gets the serial number of the sensor from the TEDS data for the channel.
TransverseSensitivity	Gets the transverse sensitivity, in percentage, that was specified in the TEDS data for the channel.
VersionLetter	Inherited from the TedsBase class, gets the version letter of the sensor from the TEDS data for the channel.
VersionNumber	Inherited from the TedsBase class, gets the version number of the sensor from the TEDS data for the channel.
YoungModulus	Gets the Young's modulus, or measure of the stiffness of the material, in MPa, that was specified in the TEDS data for the channel.
ZeroOffset	Gets the zero offset value after installation, in V/V, that was specified in the TEDS data for the channel.

Strain Gage Configuration

For an analog input channel that supports a strain gage input, you can specify one of the following configurations using the **SupportedChannelInfo.StrainGageBridgeConfiguration** property:

- **FullBridgeBending** – This configurations uses four active gages to measure bending strain. This configuration rejects axial strain, compensates for temperature, and compensates for lead resistance.
- **FullBridgeBendingPoisson –** This configuration uses four active gages to measure bending strain. This configuration also rejects axial strain, compensates for temperature, compensates for lead resistance, and compensates for the aggregate effect on the principle strain measurement due to the Poisson ratio of the specimen material.
- **FullBridgeAxialPoisson** – This configuration uses four active gages to measure axial strain. This configuration also compensates for temperature, rejects bending strain, compensates for lead resistance, and compensates for the aggregate effect on the principle strain measurement due to the Poisson ratio of the specimen material.
- **HalfBridgePoisson** – This configuration uses two active gages to measure either axial or bending strain. This configuration compensates for temperature, and compensates for the aggregate effect on the principle strain measurement due to the Poisson ratio of the specimen material.
- **HalfBridgeBending** – This configuration uses two active gages to measure bending strain. This configuration rejects axial strain and compensates for temperature.

- **QuarterBridge –** This configuration uses a single active gage to measure axial or bending strain. You must supply an external resistor that matches the nominal resistance of the bridge to complete the bridge externally.

Note: If you are using a rectangular or delta rosette, configure each channel to use the QuarterBridge configuration.

- **QuarterBridgeTempComp –** This configuration uses one active gage and one dummy gage to measure axial and bending strain while compensating for temperature.

Strain Gage Poisson Ratio

For an analog input channel that uses the FullBridgeBendingPoisson, FullBridgeAxialPoisson, and HalfBridgePoisson bridge configuration, you can specify the Poisson ratio of the specimen material using the **SupportedChannelInfo.StrainGagePoissonRatio** property. The Poisson ratio is a material-dependent constant that is the ratio of transverse (perpendicular) contraction to axial strain.

Strain Gage Lead Wire Resistance

For an analog input channel that supports a strain gage input and does not use remote sensing, you can specify the lead wire resistance of the bridge, in ohms, using the **SupportedChannelInfo.StrainGageLeadWireResistance** property.

If remote sensing is used, specify 0 for this property.

Gage Factor

For an analog input channel that supports a strain gage input, you can specify the gage factor of the strain gage using the **SupportedChannelInfo.StrainGageGageFactor** property.

Strain Gage Nominal Resistance

For an analog input channel that supports a strain gage input, you can specify the nominal resistance of the bridge (R_g), in ohms, when it is not under strain or load using the **SupportedChannelInfo.StrainGageNominalResistance** property.

Strain Gage Offset Nulling

For an analog input channel that supports a strain gage input, a balanced bridge produces zero volts under ideal conditions with zero strain applied. In practice, however, the output of a bridge in an unstrained condition is offset from zero slightly due to imperfect matching of bridge resistances.

You can adjust the offset of the channel by performing offset nulling on the channel. To perform offset nulling, read the value of the bridge in an unstrained condition using the **AnalogInputSubsystem.GetSingleValueAsVolts** method. Then, specify the value that you read using the **SupportedChannelInfo.StrainGageOffsetNullingValueInVolts** property. Internally, this value is subtracted from all subsequent measurements before the voltage is converted to strain.

Strain Gage Shunt Calibration

To determine if the analog input subsystem supports shunt calibration, use the **AnalogInputSubsystem. SupportsShuntCalibration** property.

You can use shunt calibration to correct span errors in the measurement path. You can also use shunt calibration to verify the integrity of the setup by turning on the shunt resistor before you acquire data.

If you want to use the internal shunt calibration resistor provided by the device, ensure that the internal RSHUNT+ and RSHUNT– lines are connected across the gage and that no strain is applied to the specimen, and then enable the resistor by setting the **SupportedChannelInfo.StrainGageShuntCalibrationResistorEnabled** property to True. (Be sure to set this value back to False when the shunt calibration procedure is complete.)

Once the internal shunt resistor is enabled or you have connected your own shunt resistor to the bridge, read the value of the bridge using the **AnalogInputSubsystem.GetSingleValueAsStrain** method. Then, divide the expected value of the bridge by the actual value that you read, and specify the result, in microstrain, using the **SupportedChannelInfo.StrainGageShuntCalibrationValue** property. Internally, the software multiplies the channel measurement with this value to adjust the gain of the device.

Refer to the user's manual for your device for more information on shunt calibration.

Setting Up Accelerometer (IEPE) Input Channels

To determine if the analog input subsystem supports IEPE inputs, use the **AnalogInputSubsystem.SupportsIepe** property.

Determine whether the specific channel supports IEPE inputs or multiple sensor types using the **SupportedChannelInfo.IOType** property. If the value of **IOType** is MultiSensor, you must set the multisensor type to Accelerometer using the **SupportedChannelInfo.MultiSensorType** property to use the channel for accelerometer measurements.

For channels that support accelerometers (IEPE inputs), you can set the following properties:

- Coupling
- Excitation current source

Coupling

To determine if the analog input subsystem supports DC coupling (where DC offset is included), use the **AnalogInputSubsystem. SupportsDCCoupling** property. To determine if the analog input subsystem supports AC coupling (where the DC offset is removed), use the **AnalogInputSubsystem.SupportsACCoupling** property.

You can specify one of the coupling type using the **SupportedChannelInfo.Coupling** property. By default, DC coupling is used.

Excitation Current Source Values

To determine if the analog input subsystem supports an internal excitation current source, use the **AnalogInputSubsystem.SupportsInternalExcitationCurrentSrc** property. To determine if the analog input subsystem supports an external excitation current source, use the **AnalogInputSubsystem.SupportsExternalExcitationCurrentSrc** property.

You can specify the excitation current source (Internal, External, or Disabled) using the **SupportedChannelInfo.ExcitationCurrentSource** property. By default, the excitation current source is disabled.

If you set the excitation current source to Internal, you can also set the value of the excitation current source using the **SupportedChannelInfo.ExcitationCurrentValue** property. To determine what current source values are supported by the subsystem, use the **AnalogInputSubsystem.SupportedExcitationCurrentValues** property. By default, the first value in the list of supported values is used.

Setting Up Bridge-Based Sensors

To determine if the analog input subsystem supports bridge-based sensors or general-purpose bridges, use the **AnalogInputSubsystem.SupportsBridge** property.

If this value is True, determine whether the specific channel supports bridge measurements or multiple sensor types using the **SupportedChannelInfo.IOType** property. If the value of **IOType** is MultiSensor, you must set the multisensor type to Bridge using the **SupportedChannelInfo.MultiSensorType** property.

For full-bridge-based sensors, you can set the following properties for the channel:

- TEDS information, if available, as described on page 169
- Bridge configuration, described on
- Transducer capacity, described on page 171
- Transducer rated output, described on page 171
- Nominal resistance, described on page 171
- Lead wire resistance, described on page 171
- Offset nulling value, described on page 171
- Shunt calibration resistor and value, described on page 172

Note: You must set the excitation source and value for the subsystem, as described on page 203.

For full-bridge-based sensors, you can read a single value in the native engineering units of the sensor using the **AnalogInputSubsystem.GetSingleValueAsBridgeBasedSensor** method. For a device that supports simultaneous A/Ds, you can acquire a single value from each bridge-based sensor and return the data as an array of values (in the native engineering units of the sensor) using the **AnalogInputSubsystem. GetSingleValuesAsBridgeBasedSensor** method. Refer to page 176 for more information.

If you are acquiring data to a buffer, you can read data from each bridge-based sensor (in the native engineering units of the sensor) using the **OlBuffer.GetDataAsBridgeBasedSensor** method. Refer to page 224 for more information.

For general-purpose bridges, you can read a single normalized bridge output value, in volts, from the bridge using the **AnalogInputSubsystem. GetSingleValueAsNominalBridgeOutput** method. If you are acquiring data to a buffer, you can read the normalized bridge output value from the specified channels, in mV/Vexe, using the **OlBuffer.GetDataAsNormalizedBridgeOutput** method. Refer to page 176 and page 224 for more information.

TEDS for Bridge-Based Sensors

If your bridge-based sensor or transducer supports a TEDS interface, you can read the TEDS data from the sensor directly using the **SupportedChannelInfo.BridgeSensorTeds.ReadHardwareTeds** method, or from a TEDS data file using the **SupportedChannelInfo.BridgeSensorTeds.ReadVirtualTeds** method.

Note: The properties in the BridgeSensorTeds class are read-only. It is up the application to read the value of these properties and to apply the appropriate values to the SupportedChannelInfo strain gage properties.

Table 68 lists the properties that you can read to access the TEDS data for a bridge-based sensor.

Table 68: Read-Only Properties for Accessing TEDS Data for Bridge-Based Sensor

Property	Description
BridgeResistance	Gets the initial (unstrained) gage resistance, in ohms, that was specified in the TEDS data for the channel.
BridgeType	Gets the type of bridge (Full Bridge, Half Bridge, or Quarter Bridge) that was specified in the TEDS data for the channel.
CalDate	Gets the calibration date that was specified in the TEDS data for the channel.

Table 68: Read-Only Properties for Accessing TEDS Data for Bridge-Based Sensor (cont.)

Property	Description
CalibrationPeriod	Gets the calibration period that was specified in the TEDS data for the channel.
CalInitials	Gets the calibration initials that were specified in the TEDS data for the channel.
ElectricalSignalType	Gets the electrical signal type that was specified in the TEDS data for the channel.
IsTedsConfigured	Inherited from the TedsBase class, returns True if the TEDS data stream is read successfully; otherwise, returns False.
ManufacturerId	Inherited from the TedsBase class, gets identifying information about the manufacturer of the sensor from the TEDS data for the channel.
MaxElectricalValue	Gets the maximum electrical output, in V/V, that was specified in the TEDS data for the channel.
MaximumExcitationVoltage	Gets the maximum excitation voltage that was specified in the TEDS data for the channel.
MaxPhysicalValue	Gets the positive full-scale value, in strain, that was specified in the TEDS data for the channel.
MeasID	Gets the measurement location ID that was specified in the TEDS data for the channel.
MinElectricalValue	Gets the minimum electrical output, in V/V, that was specified in the TEDS data for the channel.
MinimumExcitationVoltage	Gets the minimum excitation voltage that was specified in the TEDS data for the channel.
MinPhysicalValue	Gets the negative full-scale value, in strain, that was specified in the TEDS data for the channel.
ModelNumber	Inherited from the TedsBase class, gets the model number of the sensor from the TEDS data for the channel.
NominalExcitationVoltage	Gets the nominal excitation voltage that was specified in the TEDS data for the channel.
PhysicalMeasurand	Gets the physical Measurand (units), described on page 100, that were specified in the TEDS data for the channel.
ResponseTime	Gets the response time, in seconds, that was specified in the TEDS data for the channel.
SerialNumber	Inherited from the TedsBase class, gets the serial number of the sensor from the TEDS data for the channel.
VersionLetter	Inherited from the TedsBase class, gets the version letter of the sensor from the TEDS data for the channel.
VersionNumber	Inherited from the TedsBase class, gets the version number of the sensor from the TEDS data for the channel.

Bridge Configuration

For an analog input channel that supports a bridge input, you can specify one of the following configurations using the **SupportedChannelInfo.BridgeConfiguration** property:

- **FullBridge –** Use this configuration for bridge-based sensors or transducers, such as load cells, or general-purpose bridges that use four active gages.
- **HalfBridge –** Use this configuration for general-purpose bridges that use two active gages.
- **QuarterBridge –** Use this configuration for general-purpose bridges that use one active gages.

Transducer Capacity

Use the **SupportedChannelInfo.TransducerCapacity** property to specify or return the full-scale range of the bridge-based sensor or transducer in its native engineering units. This value is supplied by the manufacturer of the bridge-based sensor or transducer.

Transducer Rated Output

Use the **SupportedChannelInfo.TransducerRatedOutputInMv** property to specify or return the rated output of the transducer in terms of mV/V excitation. This value is supplied by the manufacturer of the bridge-based sensor or transducer.

Nominal Resistance

For an analog input channel that supports a full-bridge-based transducer that does not use remote sensing, you can specify the nominal resistance of the bridge (R_g), in ohms, when it is not under strain or load using the **SupportedChannelInfo.StrainGageNominalResistance** property.

Lead Wire Resistance

For an analog input channel that supports a full-bridge-based transducer that does not use remote sensing, you can specify the lead wire resistance of the bridge, in ohms, using the **SupportedChannelInfo.StrainGageLeadWireResistance** property.

If remote sensing is used, specify 0 for this property.

Offset Nulling

For an analog input channel that supports a strain gage input, a balanced bridge produces zero volts under ideal conditions with zero strain applied. In practice, however, the output of a bridge in an unstrained condition is offset from zero slightly due to imperfect matching of bridge resistances.

You can adjust the offset of the channel by performing offset nulling on the channel. To perform offset nulling, read the value of the bridge in an unstrained condition using the **AnalogInputSubsystem.GetSingleValueAsVolts** method. Then, specify the value that you read using the **SupportedChannelInfo.StrainGageOffsetNullingValueInVolts** property.

Internally, this value is subtracted from all subsequent measurements before the voltage is converted to strain.

Shunt Calibration

To determine if the analog input subsystem supports shunt calibration, use the **AnalogInputSubsystem.SupportsShuntCalibration** property.

You can use shunt calibration to correct span errors in the measurement path. You can also use shunt calibration to verify the integrity of the setup by turning on the shunt resistor before you acquire data.

If you want to use the internal shunt calibration resistor provided by the device, ensure that the internal RSHUNT+ and RSHUNT– lines are connected across the gage and that no strain is applied to the specimen, and then enable the resistor by setting the **SupportedChannelInfo.StrainGageShuntCalibrationResistorEnabled** property to True. (Be sure to set this value back to False when the shunt calibration procedure is complete.)

Once the internal shunt resistor is enabled or you have connected your own shunt resistor to the bridge, read the value of the bridge using the **AnalogInputSubsystem.GetSingleValueAsStrain** method. Then, divide the expected value of the bridge by the actual value that you read, and specify the result, in microstrain, using the **SupportedChannelInfo.StrainGageShuntCalibrationValue** property. Internally, the software multiplies the channel measurement with this value to adjust the gain of the device. Refer to the user's manual for your device for more information on shunt calibration.

Setting up Thermistor Input Channels

To determine if the analog input subsystem supports thermistor inputs, use the **AnalogInputSubsystem.SupportsThermistor** property.

If this value is True, determine whether the specific channel supports thermistors or multiple sensor types using the **SupportedChannelInfo.IOType** property. If the value of **IOType** is MultiSensor, you must set the multisensor type to Thermistor using the **SupportedChannelInfo.MultiSensorType** property to use the channel for thermistor measurements.

The resistance of NTC thermistors increases with decreasing temperature. The resistance to temperature relationship is characterized by the Steinhart-Hart equation:

$$\frac{1}{T} = A + BlnR + Cln(R)^3$$

where,

- T is the temperature, in degrees Kelvin.
- R is the resistance at T, in Ohms.
- A, B, and C are the Steinhart-Hart coefficients for a particular thermistor type and value, and are supplied by the thermistor manufacturer.

The value of the A, B, and C coefficients depend on the thermistor type and value that you are using. Specify the A coefficient using the **SupportedChannelInfo.ThermistorACoefficient** property. Specify the B coefficient using the **SupportedChannelInfo.ThermistorBCoefficient** property. Specify the C coefficient using the **SupportedChannelInfo.ThermistorCCoefficient** property.

Use the **SupportedChannelInfo.SensorWiringConfiguration** property to specify the wiring configuration (two-wire, three-wire, or four-wire) for the thermistor.

Note: You can read a single temperature value from one channel using the **AnalogInputSubsystem. GetSingleValueAsTemperature** method. Refer to page 176 for more information.

If you are acquiring data to a buffer, you can read the temperature from the specified channels using the **OlBuffer.GetDataAsTemperatureByte** or **OlBuffer.GetDataAsTemperatureDouble** method. Refer to page 223 for more information.

Setting Up Resistance Measurement Channels

To determine if the analog input subsystem supports resistance measurements, use the **AnalogInputSubsystem.SupportsResistance** property.

If this value is True, determine whether the specific channel supports resistance or multiple sensor types using the **SupportedChannelInfo.IOType** property. If the value of **IOType** is MultiSensor, you must set the multisensor type to Resistance using the **SupportedChannelInfo.MultiSensorType** property to use the channel for resistance measurements.

To determine if the analog input subsystem supports an internal excitation current source, use the **AnalogInputSubsystem.SupportsInternalExcitationCurrentSrc** property. To determine if the analog input subsystem supports an external excitation current source, use the **AnalogInputSubsystem.SupportsExternalExcitationCurrentSrc** property.

If the analog input channel supports a programmable current source, you can specify one of the excitation current source (Internal, External, or Disabled (the default value)) using the **SupportedChannelInfo.ExcitationCurrentSource** property.

If you set the excitation current source to Internal, you can also set the value of the excitation current source using the **SupportedChannelInfo.ExcitationCurrentValue** property. To determine what current source values are supported by the subsystem, use the **AnalogInputSubsystem.SupportedExcitationCurrentValues** property. By default, the first value in the list of supported values is used.

Note: You can read a single resistance value from one channel using the **AnalogInputSubsystem.GetSingleValueAsResistance**method. Refer to page 176 for more information.

If you are acquiring data to a buffer, you can read resistance values from the specified channels using the **OlBuffer.GetDataAsResistance** method. Refer to page 223 for more information.

Use the **SupportedChannelInfo.SensorWiringConfiguration** property to specify the wiring configuration (two-wire, three-wire, or four-wire) for the resistance measurement.

Setting Up and Configuring a Subsystem

Once you have gotten a subsystem and know about its supported channels, you can set up the subsystem for the I/O operation that you want to perform, and then configure it.

The way you set up the subsystem depends on the operation that you want to perform. Refer to the following sections for specific information on setting up I/O operations:

- For analog I/O operations, refer to page 176.
- For digital I/O operations, refer to page 229.
- For counter/timer operations, refer to page 232.
- For tachometer operations, refer to page 251.
- For quadrature decoder operations, refer to page 253.
- For simultaneous operations, refer to page 255.

Call the **Config** method within the appropriate subsystem class to configure the subsystem before performing the I/O operation.

Performing Analog I/O Operations

Using the DT-Open Layers for .NET Class Library, you can perform the following types of analog I/O operations.

- Single value analog input, described below
- Single value analog output, described on page 180
- Continuous pre- and post-trigger analog input using a start and reference trigger, described on page 185
- Continuous post-trigger analog input, described on page 187
- Continuous pre-trigger analog input, described on page 190
- Continuous about-trigger analog input, described on page 193
- Continuously paced analog output, described on page 196
- Continuous waveform generation analog output, described on page 198

Note: On some devices, an AnalogOutputSubsystem element is used to set an analog threshold trigger; these elements support single-value analog output operations only.

Single-Value Analog Input Operations

Single-value operations are the simplest to use but offer the least flexibility and efficiency. In a single-value analog input operation, a single data value is read from a single channel. The operation occurs immediately.

To determine if the subsystem supports single-value operations, use the **AnalogInputSubsystem.SupportsSingleValue** property. If this property returns a value of True, the subsystem supports single-value operations.

Once you have an AnalogInputSubsystem object, as described on page 146, and set up the channels as described on page 154, set up the AnalogInputSubsystem object for a single value operation as follows:

1. Set the **AnalogInputSubsystem.DataFlow** property to SingleValue.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogInputSubsystem.ChannelType** property. See page 201 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogInputSubsystem.Encoding** property. See page 202 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogInputSubsystem.VoltageRange** property. See page 202 for more information on voltage ranges.

5. (Optional) For measurements that require an excitation source (such as resistance, accelerometers, strain gages, or bridges), set the excitation voltage source for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property, and if using an internal excitation source, set the value of the internal excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltageValue** property. See page 203 for more information on excitation voltage sources.
6. (Optional) For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, set the synchronization mode of the analog input subsystem on each device using the **AnalogInputSubsystem.SynchronizationMode** property. See page 203 for more information on synchronization.
7. (Optional) For subsystems that support programmable filter, set the filter type using the **AnalogInputSubsystem.DataFilterType** property. See page 204 for more information.
8. Configure the subsystem using the **AnalogInputSubsystem.Config** method.
9. Acquire a single value using one of the following methods:

   For Devices with Multiplexed A/D architectures:

   – **AnalogInputSubsystem.GetSingleValueAsRaw** – Acquires a single value from a specified analog input channel using a specified gain, and returns the value as a raw count.

   – **AnalogInputSubsystem.GetSingleValueAsVolts** – Acquires a single value from a specified analog input channel using a specified gain, and returns the data as a voltage.

     For subsystems that support temperature conversions in hardware (**SupportsTemperatureDataInStream** is True), a voltage value is returned only if the specified channel is configured for a **ThermocoupleType** of None. If the channel is configured for any other **ThermocoupleType**, an exception is raised. Refer to page 158 for more information on thermocouples.

   – **AnalogInputSubsystem.GetSingleValueAsSensor** – Acquires a single value from a specified analog input channel at a specified gain, and returns the data as a sensor value.

   – **AnalogInputSubsystem.GetSingleValueAsCurrent** – For analog input subsystems that support current measurements, acquires a single current value from a specified analog input channel, and returns the data, in Amperes, as a floating-point value.

   – **AnalogInputSubsystem.GetSingleValueAsTemperature** – Overloaded method. Acquires a single temperature value from a specified analog input channel and returns the temperature data in the units you specify.

     For subsystems that support temperature conversions in hardware (**SupportsTemperatureDataInStream** is True), a temperature value is returned only if the specified channel is configured for a **ThermocoupleType** other than None. Otherwise, an exception is raised. Refer to page 158 for more information on thermocouples.

– **AnalogInputSubsystem.GetSingleCjcValueAsTemperature** – For analog input subsystems that support thermocouples, acquires a single CJC temperature value for a specified input channel and returns the temperature in the units you specify.

  This function is seldom needed. It is provided in the rare case when you want the application program, instead of the device, to correct and linearize temperature values based on the CJC temperature. To use this method, your device must support thermocouples and the ability to return floating-point values. Refer to page 158 for more information on thermocouples.

– **AnalogInputSubsystem.GetSingleValueAsResistance** – For analog input subsystems that support resistance measurements, acquires a single resistance value from a specified analog input channel, and returns the data, in ohms, as a floating-point value.

– **AnalogInputSubsystem.GetSingleValueAsStrain** – For analog input subsystems that support strain gages, acquires a single value from a specified analog input channel, and returns the data in microstrain.

– **AnalogInputSubsystem.GetSingleValueAsBridgeBasedSensor** – For analog input subsystems that support bridge-based sensors, acquires a single value from a full-bridge-based sensor and returns the data in the native engineering units of the sensor.

– **AnalogInputSubsystem.GetSingleValueAsNormalizedBridgeOutput** – For analog input subsystems that support bridge measurements, acquires a single normalized output value from the bridge and returns the data in volts.

For Devices with Simultaneous A/D architectures:

– **AnalogInputSubsystem.GetSingleValuesAsRaw** – Simultaneously acquires a single value from each input channel and returns the values as an array of raw counts.

  If your device supports streaming digital inputs, counter/timers, and/or quadrature decoder inputs through the analog input subsystem, **GetSingleValuesAsRaw** returns the data for all the analog input channels, digital input ports, counter/timer channels, and/or quadrature decoder channels.

  For meaningful digital input, counter/timer, and/or quadrature decoder data, ensure that you configure and/or start an operation on these subsystems before calling **GetSingleValuesAsRaw**. Refer to page 230 for information on configuring a continuous digital input operation, page 232, page 234, and page 237 for information on configuring counter/timer operations, and page 253 for information on configuring quadrature decoder operations.

– **AnalogInputSubsystem.GetSingleValuesAsVolts** – Simultaneously acquires a single value from each analog input channel and returns the data as an array of voltage values.

  For subsystems that support temperature conversions in hardware (**SupportsTemperatureDataInStream** is True), a voltage value is returned only if the specified channels are configured for a **ThermocoupleType** of None. If the channel is configured for any other **ThermocoupleType**, an exception is raised. Refer to page 158 for more information on thermocouples.

  - **AnalogInputSubsystem.GetSingleValuesAsSensor –** Simultaneously acquires a single value from each analog input channel and returns the data as an array of sensor values.
  - **AnalogInputSubsystem.GetSingleValuesAsTemperature –** For analog input subsystems that support temperature measurements, simultaneously acquires a single temperature value from each analog input channel and returns the data as an array of temperature values, in the units you specify.

    For subsystems that support temperature conversions in hardware (**SupportsTemperatureDataInStream** is True), a temperature value is returned only if the specified channels are configured for a **ThermocoupleType** other than None. Otherwise, an exception is raised. Refer to page 158 for more information on thermocouples.
  - **AnalogInputSubsystem.GetSingleCjcValuesAsTemperature –** For analog input subsystems that support thermocouples, simultaneously acquires a single CJC temperature for each input channel and returns the data as an array of temperature values, in the units you specify.

    This function is seldom needed. It is provided in the rare case when you want the application program, instead of the device, to correct and linearize temperature values based on the CJC temperature. To use this method, your device must support simultaneous operations, thermocouples, and the ability to return floating-point values.
  - **AnalogInputSubsystem.GetSingleValuesAsStrain –** For analog input subsystems that support strain gages, simultaneously acquires a single value from each analog input channel and returns the data as an array of values in microstrain.
  - **AnalogInputSubsystem.GetSingleValuesAsBridgeBasedSensor –** For analog input subsystems that support bridge-based sensors, simultaneously acquires a single value from each full-bridge-based sensor and returns the data as an array of values in the native engineering units of the sensor.

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

Refer to the example programs ReadSingleValueAsRaw, ReadSingleValueAsVolts, ReadSingleValueAsSensor, and ReadSingleValueAsTemperature to see how to perform a single-value analog input operation.

Note: After the acquisition is complete, you can convert a raw count value to voltage using the **AnalogInputSubsystem.RawValueToVolts** method or to a sensor value using the **AnalogInputSubsystem.RawToSensorValues** method. You can also convert a voltage to a temperature value using the **Utility.ConvertVoltsToTemperature** method. If you want to convert voltage to raw counts, you can use the **AnalogInputSubsystem.VoltsToRawValue** method.

Single-Value Analog Output Operations

In a single-value analog output operation, a single data value is output from a single analog output channel. The operation occurs immediately.

To determine if the subsystem supports single-value operations, use the **AnalogOutputSubsystem.SupportsSingleValue** property. If this property returns a value of True, the subsystem supports single-value operations.

Once you have an AnalogOutputSubsystem object, as described on page 146, set up the AnalogOutputSubsystem object for a single value operation as follows:

1. Set the **AnalogOutputSubsystem.DataFlow** property to SingleValue.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogOutputSubsystem.ChannelType** property. See page 201 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogOutputSubsystem.Encoding** property. See page 202 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogOutputSubsystem.VoltageRange** property. See page 202 for more information on voltage ranges.
5. Configure the subsystem using the **AnalogOutputSubsystem.Config** method.
6. Output a single value using one of the following methods:

   <u>**For Devices with Multiplexed D/A architectures:**</u>

   – **AnalogOutputSubsystem.SetSingleValueAsRaw** – Outputs a single raw count on the specified analog output channel.

   – **AnalogOutputSubsystem.SetSingleValueAsVolts** – Outputs a single voltage value on a specified analog output channel.

   <u>**For Devices with Simultaneous D/A architectures (SupportsSetSingleValues is True):**</u>

   – **AnalogOutputSubsystem.SetSingleValuesAsRaw** – Outputs a single raw count on each specified analog output channel. If an analog output channel is not specified, the value of the output channel will not change; the output channel maintains the last value that was written to it.

   – **AnalogOutputSubsystem.SetSingleValuesAsVolts** – Outputs a single voltage value on each specified analog output channel. If an analog output channel is not specified, the value of the output channel will not change; the output channel maintains the last value that was written to it.

Note: You can convert a raw count value to voltage using the **AnalogOutputSubsystem.RawValueToVolts** method. You can also convert a temperature value to a voltage using the **Utility.ConvertTemperatureToVolts** method. If you want to convert voltage to raw counts, you can use the **AnalogOutputSubsystem.VoltsToRawValue** method.

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

Refer to the example programs WriteSingleValueAsRaw, WriteSingleValueAsVolts, and WriteSingleValueAsRaw_ProgRanges to see how to perform a single-value analog output operation.

Continuous, Pre- and Post-Trigger Analog Input Operations Using a Start and Reference Trigger

Note: This mode requires use of an **AnalogInputSubsystem.Trigger** object and **AnalogInputSubsystem.ReferenceTrigger** object. Some devices may not support this mode.

Use this mode when you want to acquire pre-trigger data from multiple analog input channels continuously when a specified trigger occurs and, when a reference trigger occurs, acquire a specified number of post-trigger samples.

Once you have an AnalogInputSubsystem object, as described on page 146, and set up the channels as described on page 154, set up the AnalogInputSubsystem object for a continuous operation as follows:

1. Set the **AnalogInputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogInputSubsystem.ChannelType** property. See page 201 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogInputSubsystem.Encoding** property. See page 202 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogInputSubsystem.VoltageRange** property. See page 202 for more information on voltage ranges.
5. (Optional) For measurements that require an excitation source (such as resistance, accelerometers, strain gages, or bridges), set the excitation voltage source for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property, and if using an internal excitation source, set the value of the internal excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltageValue** property. See page 203 for more information on excitation voltage sources.
6. (Optional) For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, set the synchronization mode of the analog input subsystem on each device using the **AnalogInputSubsystem.SynchronizationMode** property. See page 203 for more information on synchronization.
7. (Optional) For subsystems that support programmable filter types for measuring temperature, set the filter type using the **AnalogInputSubsystem.DataFilterType** property. See page 204 for more information.
8. Set up the channel list (including setting the gain and inhibit value for each entry), as described on page 204.

Note: If you want to continuously acquire data from the digital input, counter/timer, tachometer, and/or quadrature decoder channels as part of the analog input stream, you must set up the channel list to include these channels. For counter/timer and quadrature decoder channels, you must also configure and start these subsystems before starting the analog input operation. For digital input ports, you must configure the digital input subsystem for a single-value operation before starting the analog input operation. Refer to page 230 for information on continuous digital input operations, page 232, page 234, and page 237 for information on continuous counter/timer operations, page 251 for information on tachometer operations, and page 253 for information on quadrature decoder operations.

9. Set up the clock, as described on page 212.
10. Use the **AnalogInputSubsystem.Trigger.TriggerType** property to specify the trigger type that starts pre-trigger acquisition. Refer to page 213 for more information on supported trigger sources.
11. Use the **AnalogInputSubsystem.ReferenceTrigger.TriggerType** property to specify the trigger type that stops pre-trigger acquisition and starts post-trigger acquisition. Refer to page 213 for more information on supported trigger sources.
12. If the start or reference trigger type is a threshold trigger, do the following:
    a. Specify the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger.ThresholdTriggerChannel** or **AnalogInputSubsystem.ReferenceTrigger.ThresholdTriggerChannel** property. Refer to page 215 for more information.
    b. Specify a voltage value for the threshold level using the **AnalogInputSubsystem.Trigger.Level** or **AnalogInputSubsystem.ReferenceTrigger.Level** property. Refer to page 215 for more information.
13. Specify the number of samples to acquire after the reference trigger occurs using the **AnalogInputSubsystem.ReferenceTrigger.PostTriggerScanCount** property. Refer to page 217 for more information on the post-trigger scan count.
14. If supported by your device, set up triggered scan mode, as described on page 227.
15. Set up the input buffers, as described on page 218.
16. If your program is running under a heavy CPU load, it is recommended that you set the **AnalogInputSubsystem.SynchronousBufferDone** property to True for synchronous execution of each BufferDoneEvent event in a single worker thread.
17. Configure the subsystem using the **AnalogInputSubsystem.Config** method.
18. Call the **AnalogInputSubsystem.Start** method to start the operation.

Pre-trigger acquisition begins when the start trigger is detected. When the reference trigger occurs, pre-trigger acquisition stops and post-trigger acquisition begins until the number of samples specified by **PostTriggerScanCount** has been acquired. At that point, you will get the last buffer that has valid samples; the remainder of the buffers are cancelled.

Figure 1 illustrates this mode using a channel list of two entries: channel 0 and channel 1. In this example, pre-trigger analog input data is acquired when the start trigger is detected. When the reference trigger occurs, the specified number of post-trigger samples (3, in this example) are acquired.

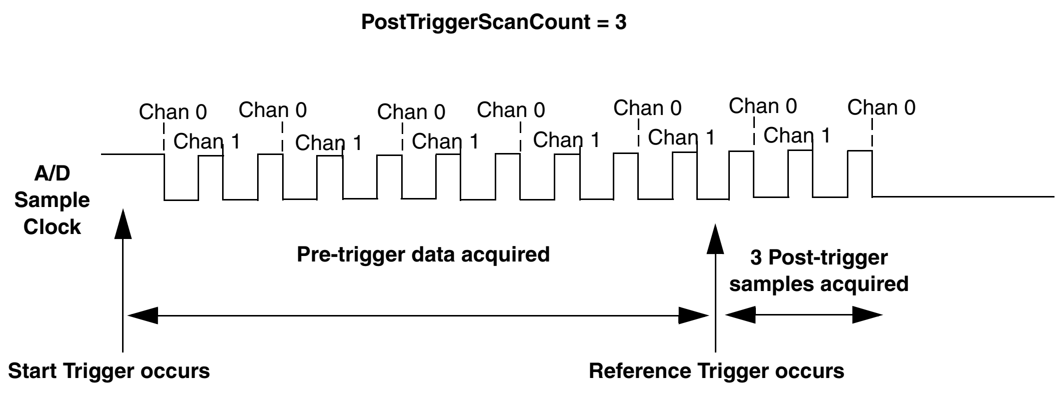


Figure 1: Continuous Pre- and Post-Trigger Operations Using a Start and Reference Trigger

If desired, you can also stop a continuous pre- and post-trigger operation using one of the following methods:

- **AnalogInputSubsystem.Stop** – Stops the operation after the current buffer has been filled. The driver raises a BufferDoneEvent event for the completed buffer and sets the **OlBuffer.ValidSamples** property to the number of samples in the completed buffer. It then raises a BufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, before raising a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored. Refer to page 218 for more information on buffers, and to page 257 for information on dealing with events.
- **AnalogInputSubsystem.Abort** – Stops the operation immediately without waiting for the current buffer to be filled and sets the **OlBuffer.ValidSamples** property to the number of samples in the buffer. The driver raises a BufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, and then raises a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored.
- **AnalogInputSubsystem.Reset** – Stops the operation immediately without waiting for the current buffer to be filled, and reinitializes the subsystem to the default configuration.

Note: If you set the **AnalogInputSubsystem.AsynchronousStop** property to True, control returns to your program after **Stop** is called. If you set the **AsynchronousStop** property to False (the default setting) control does not return to your program after **Stop** is called until the buffer completes or 20 seconds elapses (if the buffer takes longer than 20 seconds to fill). If you try to perform another operation while the stop is in progress, an exception is raised with the error code "SubsystemStopping" and the exception message "The subsystem is in the process of stopping or aborting".

Continuous Post-Trigger Analog Input Operations Using One Channel and One Buffer

Use this mode when you want to acquire one buffer of post-trigger data from one analog input channel.

Once you have an AnalogInputSubsystem object, as described on page 146, and set up the channels as described on page 154, perform the following steps:

1. Set the **AnalogInputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogInputSubsystem.ChannelType** property. See page 201 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogInputSubsystem.Encoding** property. See page 202 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogInputSubsystem.VoltageRange** property. See page 202 for more information on voltage ranges.
5. (Optional) For measurements that require an excitation source (such as resistance, accelerometers, strain gages, or bridges), set the excitation voltage source for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property, and if using an internal excitation source, set the value of the internal excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltageValue** property. See page 203 for more information on excitation voltage sources.
6. (Optional) For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, set the synchronization mode of the analog input subsystem on each device using the **AnalogInputSubsystem.SynchronizationMode** property. See page 203 for more information on synchronization.
7. (Optional) For subsystems that support programmable filter types for measuring temperature, set the filter type using the **AnalogInputSubsystem.DataFilterType** property. See page 204 for more information.
8. Set up the channel list (including setting the gain and inhibit value for the channel, and adding the channel to the channel list), as described on page 204.
9. Set up the clock, as described on page 212.
10. Use the **AnalogInputSubsystem.Trigger.TriggerType** property to specify the post-trigger source that starts the operation. Refer to page 213 for more information on supported trigger sources.
11. If the trigger type is a threshold trigger, do the following:
    a. Specify the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger.ThresholdTriggerChannel** property. Refer to page 215 for more information.
    b. Specify a voltage value for the threshold level using the **AnalogInputSubsystem.Trigger.Level** property. Refer to page 215 for more information.

12. Call the **AnalogInputSubsystem.GetOneBuffer** method to acquire one buffer of post-trigger data from the specified channel in the channel list. You specify the number of samples to acquire in the call.

    This method is synchronous and returns only after the requested data has been acquired or the specified timeout value, in milliseconds, has been exceeded. If the buffer is not filled before the specified timeout value is exceeded, **AnalogInputSubsystem.Abort** is called and a TimeoutException is raised. If a GeneralFailureEvent or DriverRuntimeErrorEvent occurs during acquisition, an OlException with the appropriate error code is raised.

13. Handle the input buffer, as described on page 221.

When the trigger occurs, post-trigger acquisition begins. When the number of samples have been acquired or the specified timeout value is exceeded, the OlBuffer object is returned.

Refer to the example program GetOneBuffer to see how to perform a continuous (post-trigger) analog input operation using one buffer.

Continuous, Post-Trigger Analog Input Operations Using Multiple Buffers

Note: This mode does not support use of the **AnalogInputSubsystem.ReferenceTrigger** object. To use a ReferenceTrigger object, refer to page 185.

Use continuous post-trigger mode when you want to acquire data from multiple analog input channel continuously when a specified start trigger occurs.

To determine if the subsystem supports continuous, post-trigger analog input operations, use the **AnalogInputSubsystem.SupportsContinuous** property. If this property returns a value of True, the subsystem supports continuous post-trigger analog input operations.

Once you have an AnalogInputSubsystem object, as described on page 146, and set up the channels as described on page 154, set up the AnalogInputSubsystem object for a continuous operation as follows:

1. Set the **AnalogInputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogInputSubsystem.ChannelType** property. See page 201 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogInputSubsystem.Encoding** property. See page 202 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogInputSubsystem.VoltageRange** property. See page 202 for more information on voltage ranges.
5. (Optional) For measurements that require an excitation source (such as resistance, accelerometers, strain gages, or bridges), set the excitation voltage source for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property, and if using an internal excitation source, set the value of the internal excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltageValue** property. See page 203 for more information on excitation voltage sources.
6. (Optional) For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, set the synchronization mode of the analog input subsystem on each device using the **AnalogInputSubsystem.SynchronizationMode** property. See page 203 for more information on synchronization.
7. (Optional) For subsystems that support programmable filter types for measuring temperature, set the filter type using the **AnalogInputSubsystem.DataFilterType** property. See page 204 for more information.
8. Set up the channel list (including setting the gain and inhibit value for each entry), as described on page 204.

Note: If you want to continuously acquire data from the digital input, counter/timer, tachometer, and/or quadrature decoder channels as part of the analog input stream, you must set up the channel list to include these channels. For counter/timer and quadrature decoder channels, you must also configure and start these subsystems before starting the analog input operation. For digital input ports, you must configure the digital input subsystem for a single-value operation before starting the analog input operation. Refer to page 230 for information on continuous digital input operations, page 232, page 234, and page 237 for information on continuous counter/timer operations, page 251 for information on tachometer operations, and page 253 for information on quadrature decoder operations.

9. Set up the clock, as described on page 212.
10. Use the **AnalogInputSubsystem.Trigger.TriggerType** property to specify the post-trigger source that starts the operation. Refer to page 213 for more information on supported trigger sources.
11. If the trigger type is a threshold trigger, do the following:
    a. Specify the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger.ThresholdTriggerChannel** property. Refer to page 215 for more information.
    b. Specify a voltage value for the threshold level using the **AnalogInputSubsystem.Trigger.Level** property. Refer to page 215 for more information.
12. If supported by your device, set up triggered scan mode, as described on page 227.
13. Set up the input buffers, as described on page 218.
14. If your program is running under a heavy CPU load, it is recommended that you set the **AnalogInputSubsystem.SynchronousBufferDone** property to True for synchronous execution of each BufferDoneEvent event in a single worker thread.
15. Configure the subsystem using the **AnalogInputSubsystem.Config** method.
16. Call the **AnalogInputSubsystem.Start** method to start the continuous post-trigger operation.

When the post-trigger is detected, the device cycles through the channel list, acquiring the value for each ChannelListEntry object 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. The event BufferDoneEvent is generated as each buffer is filled with analog input data; refer to page 257 for information on dealing with events and reading the data in the buffer.

Figure 2 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. The device wraps to the beginning of the channel list and repeats continuously.

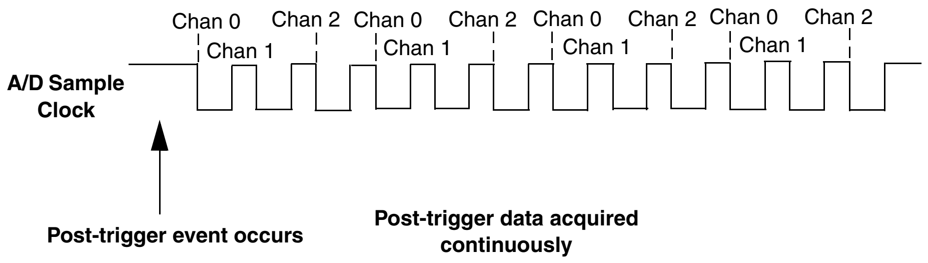


Figure 2: Continuous Post-Trigger Mode

To stop a continuous post-trigger operation, use one of the following methods:

- **AnalogInputSubsystem.Stop** – Stops the operation after the current buffer has been filled. The driver raises a BufferDoneEvent event for the completed buffer and sets the **OlBuffer.ValidSamples** property to the number of samples in the completed buffer. It then raises a BufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, before raising a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored. Refer to page 218 for more information on buffers, and to page 257 for information on dealing with events.
- **AnalogInputSubsystem.Abort** – Stops the operation immediately without waiting for the current buffer to be filled and sets the **OlBuffer.ValidSamples** property to the number of samples in the buffer. The driver raises a BufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, and then raises a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored.
- **AnalogInputSubsystem.Reset** – Stops the operation immediately without waiting for the current buffer to be filled, and reinitializes the subsystem to the default configuration.

Notes: If you set the **AnalogInputSubsystem.AsynchronousStop** property to True, control returns to your program after **Stop** is called. If you set the **AsynchronousStop** property to False (the default setting) control does not return to your program after **Stop** is called until the buffer completes or 20 seconds elapses (if the buffer takes longer than 20 seconds to fill). If you try to perform another operation while the stop is in progress, an exception is raised with the error code "SubsystemStopping" and the exception message "The subsystem is in the process of stopping or aborting".

Refer to the example programs ReadBufferedDataAsRaw, ReadBufferedDataAsRawDigTrigger, ReadBufferedDataAsVolts, ReadBufferedDataAsSensor, ReadBufferedDataAsTemperature, and ReadBufferedDataIntoOscilloscope to see how to perform a continuous (post-trigger) analog input operation.

Continuous, Pre-Trigger Analog Input Operations (Legacy Devices)

Note: This mode does not support use of the **AnalogInputSubsystem.ReferenceTrigger** object. To use a ReferenceTrigger object, see page 185.

Some older, legacy devices support continuous pre-trigger analog input operations. Use continuous pre-trigger mode when you want to continuously acquire data before a specific external trigger occurs.

To determine if the subsystem supports continuous pre-trigger operations, use the **AnalogInputSubsystem.SupportsContinuousPreTrigger** property. If this property returns a value of True, continuous pre-trigger mode is supported.

Once you have an AnalogInputSubsystem object, as described on page 146, and set up the channels as described on page 154, set up the AnalogInputSubsystem object for a continuous pre-trigger operation as follows:

1. Set the **AnalogInputSubsystem.DataFlow** property to ContinuousPreTrigger.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogInputSubsystem.ChannelType** property. See page 201 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogInputSubsystem.Encoding** property. See page 202 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogInputSubsystem.VoltageRange** property. See page 202 for more information on voltage ranges.
5. (Optional) For measurements that require an excitation source (such as resistance, accelerometers, strain gages, or bridges), set the excitation voltage source for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property, and if using an internal excitation source, set the value of the internal excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltageValue** property. See page 203 for more information on excitation voltage sources.
6. (Optional) For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, set the synchronization mode of the analog input subsystem on each device using the **AnalogInputSubsystem.SynchronizationMode** property. See page 203 for more information on synchronization.
7. (Optional) For subsystems that support programmable filter types for measuring temperature, set the filter type using the **AnalogInputSubsystem.DataFilterType** property. See page 204 for more information.
8. Set up the channel list (including setting the gain and the inhibit value for each entry), as described on page 204.

Note: If you want to continuously acquire data from the digital input, counter/timer, tachometer, and/or quadrature decoder channels as part of the analog input stream, you must set up the channel list to include these channels. For counter/timer and quadrature decoder channels, you must also configure and start these subsystems before starting the analog input operation. For digital input ports, you must also configure the digital input subsystem for a single-value operation before starting the analog input operation. Refer to page 230 for information on continuous digital input operations, page 232, page 234, and page 237 for information on continuous counter/timer operations, page 251 for information on tachometer operations, and page 253 for information on quadrature decoder operations.

9. (Optional) Set up the clock, as described on page 212.
10. Use the **AnalogInputSubsystem.Trigger.PretriggerSource** property to specify the trigger source that starts the pre-trigger operation (generally this is a software trigger).
11. Use the **AnalogInputSubsystem.Trigger.TriggerType** property to specify the external post-trigger source that stops the pre-trigger operation. Refer to page 213 for more information on supported trigger sources.
12. If the trigger type is a threshold trigger, do the following:
    a. Specify the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger.ThresholdTriggerChannel** property. Refer to page 215 for more information.
    b. Specify a voltage value for the threshold level using the **AnalogInputSubsystem.Trigger.Level** property. Refer to page 215 for more information.
13. If supported by your device, set up triggered scan mode, as described on page 227.
14. Set up the input buffers, as described on page 218.
15. Configure the subsystem using the **AnalogInputSubsystem.Config** method.
16. Call the **AnalogInputSubsystem.Start** method to start the continuous pre-trigger operation.

Pre-trigger acquisition begins 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). The event PreTriggerBufferDoneEvent is generated as each buffer is filled with pre-trigger analog input data; refer to page 257 for information on dealing with events and reading data from the buffers.

Figure 3 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. 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.

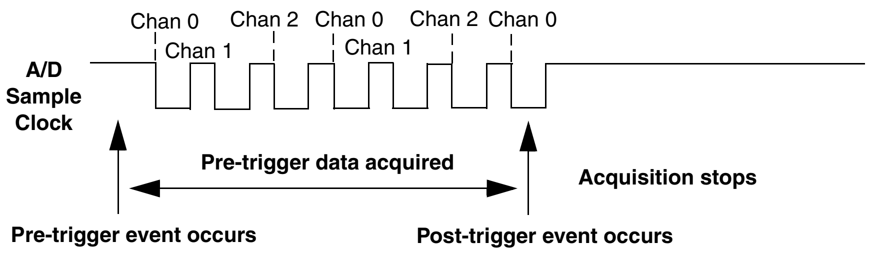


Figure 3: Continuous Pre-Trigger Mode

To stop a continuous pre-trigger operation, use one of the following methods:

- **AnalogInputSubsystem.Stop** – Stops the operation after the current buffer has been filled. The driver raises a PreTriggerBufferDoneEvent event for the completed buffer and sets the **OlBuffer.ValidSamples** property to the number of samples in the completed buffer. It then raises a PreTriggerBufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, before raising a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored. Refer to page 218 for more information on buffers, and to page 257 for information on dealing with events.
- **AnalogInputSubsystem.Abort** – Stops the operation immediately without waiting for the current buffer to be filled and sets the **OlBuffer.ValidSamples** property to the number of samples in the buffer. The driver raises a PreTriggerBufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, and then raises a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored.
- **AnalogInputSubsystem.Reset** – Stops the operation immediately without waiting for the current buffer to be filled, and reinitializes the subsystem to the default configuration.

Notes: If you set the **AnalogInputSubsystem.AsynchronousStop** property to True, control returns to your program after **Stop** is called.

If you set the **AsynchronousStop** property to False (the default setting) control does not return to your program after **Stop** is called until the buffer completes or 20 seconds elapses (if the buffer takes longer than 20 seconds to fill). If you try to perform another operation while the stop is in progress, an exception is raised with the error code "SubsystemStopping" and the exception message "The subsystem is in the process of stopping or aborting".

Continuous, About-Trigger Analog Input Operations (Legacy Devices)

Note: This mode does not support use of the **AnalogInputSubsystem.ReferenceTrigger** object. To use the ReferenceTrigger object, see page 185.

Some older, legacy devices support continuous about-trigger analog input operations. Use continuous about-trigger mode when you want to continuously acquire data both before and after a specific external trigger 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 operations, use the **AnalogInputSubsystem.SupportsContinuousPrePostTrigger** property. If this property returns a value of True, continuous about-trigger mode is supported.

Once you have an AnalogInputSubsystem object, as described on page 146, and set up the channels as described on page 154, set up the AnalogInputSubsystem object for a continuous about-trigger operation as follows:

1. Set the **AnalogInputSubsystem.DataFlow** property to ContinuousPrePostTrigger.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogInputSubsystem.ChannelType** property. See page 201 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogInputSubsystem.Encoding** property. See page 202 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogInputSubsystem.VoltageRange** property. See page 202 for more information on voltage ranges.
5. (Optional) For measurements that require an excitation source (such as resistance, accelerometers, strain gages, or bridges), set the excitation voltage source for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property, and if using an internal excitation source, set the value of the internal excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltageValue** property. See page 203 for more information on excitation voltage sources.
6. (Optional) For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, set the synchronization mode of the analog input subsystem on each device using the **AnalogInputSubsystem.SynchronizationMode** property. See page 203 for more information on synchronization.
7. (Optional) For subsystems that support programmable filter types for measuring temperature, set the filter type using the **AnalogInputSubsystem.DataFilterType** property. See page 204 for more information.
8. Set up the channel list (including setting the gain and inhibit value for each entry), as described on page 204.

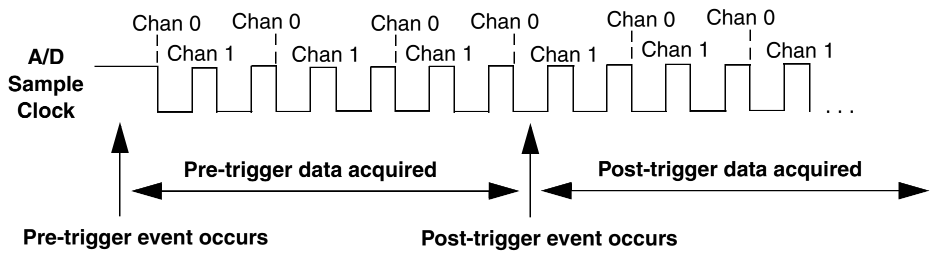
Note: If you want to continuously acquire data from the digital input, counter/timer, tachometer, and/or quadrature decoder channels as part of the analog input stream, you must set up the channel list to include these channels. For counter/timer and quadrature decoder channels, you must also configure and start these subsystems before starting the analog input operation. For digital input ports, you must also configure the digital input subsystem for a single-value operation before starting the analog input operation. Refer to page 230 for information on continuous digital input operations, page 232, page 234, and page 237 for information on continuous counter/timer operations, page 251 for information on tachometer operations, and page 253 for information on quadrature decoder operations.

9. (Optional) Set up the clock, as described on page 212.
10. Use the **AnalogInputSubsystem.Trigger.PreTriggerSource** property to specify the trigger source that starts the pre-trigger operation (generally this is a software trigger).
11. Use the **AnalogInputSubsystem.Trigger.TriggerType** property to specify the external post-trigger source that stops the pre-trigger operation and starts the post-trigger operation. Refer to page 213 for more information on supported trigger sources.
12. If the trigger type is a threshold trigger, do the following:
    - **a.** Specify the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger.ThresholdTriggerChannel** property. Refer to page 215 for more information.
    - **b.** Specify a voltage value for the threshold level using the **AnalogInputSubsystem.Trigger.Level** property. Refer to page 215 for more information.
13. If supported by your device, set up triggered scan mode, as described on page 227.
14. Set up the input buffers, as described on page 218.
15. If your program is running under a heavy CPU load, it is recommended that you set the **AnalogInputSubsystem.SynchronousBufferDone** property to True for synchronous execution of each BufferDoneEvent event in a single worker thread.
16. Configure the subsystem using the **AnalogInputSubsystem.Config** method.
17. Call the **AnalogInputSubsystem.Start** method to start the continuous about-trigger operation.

The about-trigger acquisition begins when the device detects the pre-trigger source. The event PreTriggerBufferDoneEvent is generated as each buffer is filled with pre-trigger analog input data; refer to page 257 for information on dealing with events.

When it detects an external post-trigger source, the device stops acquiring pre-trigger data and starts acquiring post-trigger data. The event BufferDoneEvent is generated as each buffer is filled with post-trigger analog input data. The about-trigger operation continues until either the allocated buffers are filled or you stop the operation.

Figure 4 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 212 for more information on clock sources. The device wraps to the beginning of the channel list and acquires post-trigger data continuously.



Chan 0
Chan 1
A/D Sample Clock
. . .
Pre-trigger data acquired
Post-trigger data acquired
Pre-trigger event occurs
Post-trigger event occurs

Figure 4: Continuous About-Trigger Mode

To stop a continuous about-trigger operation, use one of the following methods:

- **AnalogInputSubsystem.Stop** – Stops the operation after the current buffer has been filled. Depending on when the operation was stopped, the driver raises either a PreTriggerBufferDoneEvent or a BufferDoneEvent for the completed buffer and sets the **OlBuffer.ValidSamples** property to the number of samples in the completed buffer. It then raises either a PreTriggerBufferDoneEvent or BufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, before raising a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored. Refer to page 218 for more information on buffers, and to page 257 for information on dealing with events and reading data from the buffers.
- **AnalogInputSubsystem.Abort** – Stops the operation immediately without waiting for the current buffer to be filled and sets the **OlBuffer.ValidSamples** property to the number of samples in the buffer. Depending on when the operation was aborted, the driver raises either a PreTriggerBufferDoneEvent or BufferDoneEvent for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, and then raises a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored.
- **AnalogInputSubsystem.Reset** – Stops the operation immediately without waiting for the current buffer to be filled, and reinitializes the subsystem to the default configuration.

Notes: If you set the **AnalogInputSubsystem.AsynchronousStop** property to True, control returns to your program after **Stop** is called. If you set the **AsynchronousStop** property to False (the default setting) control does not return to your program after **Stop** is called until the buffer completes or 20 seconds elapses (if the buffer takes longer than 20 seconds to fill).

If you try to perform another operation while the stop is in progress, an exception is raised with the error code "SubsystemStopping" and the exception message "The subsystem is in the process of stopping or aborting".

Continuously Paced Analog Output Operations

Use continuously paced output mode if you want to accurately control the period between conversions of individual analog output channels in the channel list.

To determine if the subsystem supports continuous analog output operations, use the **AnalogOutputSubsystem.SupportsContinuous** property. If this property returns a value of True, the subsystem supports continuously paced analog output operations.

Once you have an AnalogOutputSubsystem object, as described on page 146, set up the AnalogOutputSubsystem object for a continuous operation as follows:

1. Set the **AnalogOutputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogOutputSubsystem.ChannelType** property. See page 201 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogOutputSubsystem.Encoding** property. See page 202 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogOutputSubsystem.VoltageRange** property. See page 202 for more information on voltage ranges.
5. Set up the channel list, as described on page 204.

Note: If you want to continuously update the digital output channels as part of the analog output stream, you must set up the channel list to include the digital output port. In addition, you must configure the digital output subsystem for a single-value operation, as described on page 230, before starting the analog output operation.

6. (Optional) Set up the clock, as described on page 212.
7. (Optional) Use the **AnalogOutputSubsystem.Trigger.TriggerType** property to specify the trigger source that starts the operation. Refer to page 213 for more information on supported trigger sources.

8. If the trigger type is a threshold trigger, do the following:
    - **a.** Specify the channel to use for the threshold trigger using the **AnalogOutputSubsystem.Trigger.ThresholdTriggerChannel** property. Refer to page 215 for more information.
    - **b.** Specify a voltage value for the threshold level using the **AnalogOutputSubsystem.Trigger.Level** property. Refer to page 215 for more information.
9. Set the **AnalogOutputSubsystem.WrapSingleBuffer** property to False (the default value) to specify a buffer wrap mode of none. In this mode, the operation continues indefinitely as long as you process the buffers ad put them back on the queue in a timely manner.
10. Use software to fill the output buffer with the values that you want to write to the analog output channels and to the digital output port, if applicable. Refer to page 218 for more information on output buffers.
11. (Optional) For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, set the synchronization mode of the analog output subsystem on each device using the **AnalogOutputSubsystem.SynchronizationMode** property. See page 203 for more information on synchronization.
12. Configure the subsystem using the **AnalogOutputSubsystem.Config** method.
13. Call the **AnalogOutputSubsystem.Start** method to start the continuous analog output operation.

When it detects the appropriate trigger, the device starts writing output values to the channels, as determined by the channel list. The operation repeats continuously until either all the data is output from the buffers or you stop the operation. The event BufferDoneEvent occurs as each OlBuffer object is completed. If no buffers are available on the queue, the operation stops, and the event QueueDoneEvent is raised. Refer to page 218 for more information about buffers.

Make sure that the host computer transfers data to the output channel list fast enough so that the list always has data to output; otherwise, the event DriverRunTimeErrorEvent is raised. Refer to page 266 for more information on this event.

If your device supports it, you can mute the output, which attenuates the output voltage to 0 V by calling **AnalogOutputSubsystem.Mute**. This does not stop the analog output operation; instead, the analog output voltage is reduced to 0 V over a hardware-dependent number of samples. You can unmute the output voltage to its current level by calling **AnalogOutputSubsystem.UnMute**. To determine if muting and unmuting are supported by your device, read the value of the **AnalogOutputSubsystem.SupportsMute** property. If this value is True, muting and unmuting are supported.

To stop a continuous analog output operation, do not send new data to the device or use one of the following methods:

- **AnalogOutputSubsystem.Stop** – Stops the operation after all the data in the current buffer has been output. The driver raises a BufferDoneEvent event for the completed buffer and up to eight inprocess buffers, before raising a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored. Refer to page 218 for more information on buffers.

- **AnalogOutputSubsystem.Abort** – Stops the operation immediately without waiting for the data in the current buffer to be output. The driver raises a BufferDoneEvent event for the partially completed buffer and up to eight inprocess buffers, before raising a QueueStoppedEvent event. All subsequent triggers are ignored.
- **AnalogOutputSubsystem.Reset** – Stops the operation immediately without waiting for the data in the current buffer to be output, and reinitializes the subsystem to the default configuration.

Notes: If you set the **AnalogOutputSubsystem.AsynchronousStop** property to True, control returns to your program after **Stop** is called. If you set the **AsynchronousStop** property to False (the default setting) control does not return to your program after **Stop** is called until the buffer completes or 20 seconds elapses (if the buffer takes longer than 20 seconds to be output).

If you try to perform another operation while the stop is in progress, an exception is raised with the error code "SubsystemStopping" and the exception message "The subsystem is in the process of stopping or aborting".

Refer to the example program WriteBufferedDataAsVolts to see how to perform a continuously paced analog output operation.

Continuous Waveform Generation Operations

Use waveform generation mode if you want to output a waveform repetitively to analog output channels and, if supported, digital output ports, as specified in the ChannelList object.

To determine if the subsystem supports waveform generation operations, use the following properties:

- **AnalogOutputSubsystem.SupportsContinuous** property – If this property returns a value of True, continuous output operations are supported. This is a requirement for waveform generation operations.
- **AnalogOutputSubsystem.SupportsWrapSingle** property – If this property returns a value of True, the device driver will output data continuously from the first buffer queued to the analog output subsystem. This is a requirement for waveform generation operations. Refer to page 218 for more information on buffers.
- **AnalogOutputSubsystem.SupportsWaveformModeOnly** property – If this property returns a value of True, the device driver will output a waveform continuously from the onboard FIFO only. Set the **AnalogOutputSubsystem.WrapSingleBuffer** property to True. In addition, set the buffer size to be less than or equal to the FIFO size specified by the **AnalogOutputSubsystem.FifoSize** property. Refer to page 218 for more information on buffers.

Once you have an AnalogOutputSubsystem object, as described on page 146, set up the AnalogOutputSubsystem object for a continuous operation as follows:

1. Set the **AnalogOutputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogOutputSubsystem.ChannelType** property. See page 201 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogOutputSubsystem.Encoding** property. See page 202 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogOutputSubsystem.VoltageRange** property. See page 202 for more information on voltage ranges.
5. Set up the channel list, as described on page 204.

Note: If you want to continuously update the digital output channels as part of the analog output stream, you must set up the channel list to include the digital output port. In addition, you must configure the digital output subsystem for a single-value operation, as described on page 230, before starting the analog output operation.

---

6. (Optional) Set up the clock, as described on page 212.
7. (Optional) Use the **AnalogOutputSubsystem.Trigger.TriggerType** property to specify the trigger source that starts the operation. Refer to page 213 for more information on supported trigger sources.
8. If the trigger type is a threshold trigger, do the following:
   a. Specify the channel to use for the threshold trigger using the **AnalogOutputSubsystem.Trigger.ThresholdTriggerChannel** property. Refer to page 215 for more information.
   b. Specify a voltage value for the threshold level using the **AnalogOutputSubsystem.Trigger.Level** property. Refer to page 215 for more information.
9. Set the **AnalogOutputSubsystem.WrapSingleBuffer** property to True, so that a single buffer is reused.
10. Use software to fill the output buffer with the values that you want to write to the analog output channels and to the digital output port, if applicable. Refer to your device documentation for details on the waveform pattern that you can specify and to page 218 for more information on output buffers.

Note: For devices that have a FIFO onboard for waveform generation operations, the device driver downloads the buffer into the FIFO on the device if the size of the buffer is less than or equal to the FIFO size. The driver (or device) outputs the data starting from the first location in the FIFO. When it reaches the end of the FIFO, the driver (or device) continues outputting data from the first location of the FIFO and the process continues indefinitely until you stop it.

You can determine the size of the FIFO on the device using the **AnalogOutputSubsystem.FifoSize** property. This property returns the actual FIFO size in kilobytes.

11. (Optional) For subsystems that allow you to synchronize operations on multiple devices using a synchronization connector, set the synchronization mode of the analog output subsystem on each device using the **AnalogOutputSubsystem.SynchronizationMode** property. See page 203 for more information on synchronization.
12. Configure the subsystem using the **AnalogOutputSubsystem.Config** method.
13. Call the **AnalogOutputSubsystem.Start** method to start the continuous analog output operation.

When it detects a trigger, the host computer writes the pattern in the buffer to specified output channels, as determined by the channel list.

If your device supports it, you can mute the output, which attenuates the output voltage to 0 V by calling **AnalogOutputSubsystem.Mute**. This does not stop the analog output operation; instead, the analog output voltage is reduced to 0 V over a hardware-dependent number of samples. You can unmute the output voltage to its current level by calling **AnalogOutputSubsystem.UnMute**. To determine if muting and unmuting are supported by your device, read the value of the **AnalogOutputSubsystem.SupportsMute** property. If this value is True, muting and unmuting are supported.

To stop a continuous analog output operation, do not send new data to the device or use one of the following methods:

- **AnalogOutputSubsystem.Stop** – Stops the operation after all the data in the current buffer has been output. The driver raises a BufferDoneEvent event for the completed buffer and up to eight inprocess buffers, before raising a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored. Refer to page 218 for more information on buffers.
- **AnalogOutputSubsystem.Abort** – Stops the operation immediately without waiting for the data in the current buffer to be output. The driver raises a BufferDoneEvent event for the partially completed buffer and up to eight inprocess buffers, before raising a QueueStoppedEvent event. All subsequent triggers are ignored.
- **AnalogOutputSubsystem.Reset** – Stops the operation immediately without waiting for the data in the current buffer to be output, and reinitializes the subsystem to the default configuration.

Notes: If you set the **AnalogOutputSubsystem.AsynchronousStop** property to True, control returns to your program after **Stop** is called. If you set the **AsynchronousStop** property to False (the default setting) control does not return to your program after **Stop** is called until the buffer completes or 20 seconds elapses (if the buffer takes longer than 20 seconds to be output).

If you try to perform another operation while the stop is in progress, an exception is raised with the error code "SubsystemStopping" and the exception message "The subsystem is in the process of stopping or aborting".

Setting the Channel Type

The DT-Open Layers for .NET Class Library supports the following channel types for a specified analog I/O subsystem:

- **SingleEnded –** 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 **SupportsSingleEnded** property of the appropriate subsystem. If this property returns a value of True, the subsystem supports single-ended inputs.

  To determine how many single-ended channels are supported by the subsystem, use the **MaxSingleEndedChannels** property of the appropriate subsystem.
- **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 **SupportsDifferential** property of the appropriate subsystem. If this property returns a value of True, the subsystem supports differential inputs.

  To determine how many differential channels are supported by the subsystem, use the **MaxDifferentialChannels** property of the appropriate subsystem.

Set and/or return the channel type using the **ChannelType** property of the appropriate subsystem.

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.

Setting the Data Encoding

Two data encoding types are available: binary and twos complement.

To determine if your subsystem supports binary data encoding, use the **SupportsBinaryEncoding** property of the appropriate subsystem. If this property returns a value of True, the subsystem supports binary data encoding.

To determine if your subsystem supports twos complement data encoding, use the **SupportsTwosCompEncoding** property of the appropriate subsystem. If this property returns a value of True, the subsystem supports twos complement data encoding.

Use the **Encoding** property of the appropriate subsystem to specify the data encoding type.

Setting the Voltage Range

To determine how many ranges the subsystem supports, use the **NumberOfRanges** property of the appropriate subsystem.

To determine all the available voltage ranges for your subsystem, use the **SupportedVoltageRanges** property of the appropriate subsystem.

Some analog output subsystems support both voltage and current output channels. To determine if the subsystem supports current outputs, use the **AnalogOutputSubsystem.SupportsCurrentOutput** property.

Use the **VoltageRange** property of the appropriate subsystem to set or return the voltage range for the subsystem.

Note: If you are using a current output channel, determine how the voltage range maps to your current output range and write the appropriate voltage to the output channel.

The following example shows how to set the voltage range for an analog input subsystem to the first range in the list of supported voltage ranges:

Visual C#
```
ainSS.VoltageRange = ainSS.SupportedVoltageRanges[0];
```

Visual Basic
```
ainSS.VoltageRange = ainSS.SupportedVoltageRanges(0)
```

Setting the Excitation Voltage Source and Value

To determine if the analog input subsystem supports an internal excitation voltage source, use the **AnalogInputSubsystem.SupportsInternalExcitationVoltageSrc** property. To determine if the analog input subsystem supports an external excitation voltage source, use the **AnalogInputSubsystem.SupportsExternalExcitationVoltageSrc** property.

You specify the excitation voltage source to use (Internal, External, or Disabled) for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property. By default, the excitation voltage source is disabled.

If you set the excitation voltage source to Internal, you can also set the value of the excitation voltage source using the **SupportedChannelInfo.ExcitationVoltageValue** property.

You can determine the minimum allowable value for the internal excitation voltage source using the **AnalogInputSubsystem.MinExcitationVoltageValue** property. Similarly, you can determine the maximum allowable value for the internal excitation voltage source using the **AnalogInputSubsystem.MaxExcitationVoltageValue** property.

Setting the Synchronization Mode

Some devices provide one or more synchronization connectors (such as an LVDS RJ45 or Sync Bus connector) that allows you to synchronize operations on multiple devices. In this configuration, the subsystem on one device is configured as the master and the subsystem on the other device is configured as a slave. When the subsystem on the master module is triggered, the specified subsystem on both the master device and the slave device start operating at the same time.

To determine if your subsystem supports the ability to program the synchronization mode, use the **SupportsSynchronization** property of the appropriate subsystem.

If the subsystem supports programmable synchronization modes, use the **SynchronizationMode** property to set or get the current synchronization mode; the following values are supported:

- None – The subsystem is configured to ignore the synchronization circuit.
- Master – Sets the subsystem as a master; the synchronization connector on the device is configured to output a synchronization signal.
- Slave – Sets the subsystem as a slave; the synchronization connector on the device is configured to accept a synchronization signal as an input.

Refer to your hardware documentation for more information on how synchronization works for your device.

Setting the Filter Type

For some devices, like the TEMPpoint and VOLTpoint instruments, that support programmable filter types for measuring data, you can set the filter type.

To determine if your subsystem supports the ability to program the filter type, use the **SupportsDataFilters** property of the appropriate subsystem.

If the subsystem supports programmable filter types, use the **DataFilterType** property to set or get the current filter type; the following values are supported:

- **Raw** – No filter. Provides fast response times, but the data may be difficult to interpret. Use when you want to filter the data yourself.

  The Raw filter type returns the data exactly as it comes out of the Delta-Sigma A/D converters. Note that Delta-Sigma converters provide substantial digital filtering above the Nyquist frequency.

  Generally, the only time it is desirable to use the Raw filter type is if you are using fast responding inputs, sampling them at higher speeds (> 1 Hz), and need as much response speed as possible.
- **MovingAverage** – Provides a compromise of filter functionality and response time. This filter can be used in any application. This low-pass filter takes the previous 16 samples, adds them together, and divides by 16.

Note: The properties **SupportsTemperatureFilters** and**TemperatureFilterType** are deprecated properties that have been replaced by **SupportsDataFilters** and **DataFilterType**, respectively.

Setting up the Channel List

Note: Single-value operations do not use a channel list.

If you want to acquire data from or update multiple channels, you need to use a continuous operation mode and specify the channels that you want to sample (and the order in which to sample them) in a ChannelList object.

Channels are sampled or updated in order from the first entry to the last entry in the ChannelList object. Channel numbering is zero-based; that is, the first entry in the ChannelList is at index 0, the second entry is at index 1, and so on.

The **ChannelList** property is accessible using any subsystem class whose **SupportsContinuous** property is True. Typically, a ChannelList is used with the AnalogInputSubsystem and AnalogOutputSubsystem classes.

For an analog input subsystem, you can specify analog input channels, as well as digital inputs, counter/timers, and/or quadrature decoders in the ChannelList object, if your device supports it. Similarly, for an analog output subsystem, you can specify analog output channels as well as digital outputs in the ChannelList object, if your device supports it. Refer to page 150 for more information on available channels.

You can add sequential channels (such as channels 0, 1, 2, 3) or random channels (such as channels 2, 9, 7) to the ChannelList object, and can specify a channel more than once in the list (such as channels 1, 2, 1), if your device supports it.

For devices that support simultaneous sample-and-hold mode, the channel numbers must typically be in ascending order (such as 3, 6, 8, and so on), and cannot be repeated. To determine if the subsystem supports simultaneous sample-and-hold mode, use the **AnalogInputSubsystem.SupportsSimultaneousSampleHold** property. If this property returns a value of True, the subsystem supports simultaneous sample-and-hold mode.

Other devices may limit the order in which you can enter a channel in the channel list. See the user's manual for your device to determine any channel ordering limitations.

The following example shows a ChannelList that contains four channels. Channel 1 is sampled first, followed by channel 2, channel 1 again, and then channel 0:

Table 69: Example of a ChannelList Object

Channel-List Index	Channel	Description
0	1	Sample channel 1.
1	2	Sample channel 2.
2	1	Sample channel 1 again.
3	0	Sample channel 0.

Adding Channels to a Channel List

The **ChannelList.Add** method adds a channel to the end of the ChannelList object, and returns the index of the added channel. You can specify the channel to add in one of the following ways:

- By physical channel number
- By channel name
- By ChannelListEntry object

The following sections describe these methods.

Adding Channels By Physical Channel Number

This method is the simplest way to add channels into the ChannelList object, particularly if you are adding channels that are native to the subsystem type (such as analog input channels on an analog input subsystem).

For native channels, the physical channel number always equals the logical channel number. While non-native channels, such as digital inputs that are streamed through the analog input subsystem, can also be added this way, the physical channel number is not the same as the logical channel number, so you may find it easier to add the channel by name or by ChannelListEntry object instead.

A new ChannelListEntry object is returned for each physical channel that is added this way. Refer to page 208 for more information on ChannelListEntry objects.

The following example shows how to use the **Add** method to add physical channel 0 to the end of a ChannelList for an analog input subsystem:

Visual C#
```
ch = AinSS.ChannelList.Add(0);
```

Visual Basic
```
ch = AinSS.ChannelList.Add(0)
```

Adding Channels By Channel Name

The channel name is the name that you assigned to the channel using the SupportedChannelInfo class, described on page 150. A new ChannelListEntry object is returned for each channel that is added this way. Refer to page 208 for more information on ChannelListEntry objects.

The following example shows how to use the **Add** method to add a channel named Sensor to the end of a ChannelList for an analog input subsystem:

Visual C#
```
//Specify the name Sensor for the first
//analog input channel.
ainSS.SupportedChannels[0].Name = "Sensor";
//Add the channel named Sensor to the ChannelList
ch = ainSS.ChannelList.Add("Sensor");
```

Visual Basic
```
'Specify the name Sensor for the first
'analog input channel.
ainSS.SupportedChannels(0).Name = "Sensor"
ch = AinSS.ChannelList.Add("Sensor")
```

Adding Channels By ChannelListEntry Object

This method is useful if you want a more generic approach to adding channels. This approach frees you from keeping track of physical channel numbers and their names.

To get a ChannelListEntry object, use the **ChannelListEntry** constructor within the ChannelListEntry class, specifying the SupportedChannelInfo object for the channel that you want to sample or update. See page 150 for more information on SupportedChannelInfo objects.

This example creates a ChannelListEntry called Ch0 for physical channel 0 of the analog input subsystem, using all the information contained in SupportedChannelInfo for that channel.

Visual C#
```
ChannelListEntry Ch0 = new ChannelListEntry (
  ainSS.SupportedChannels.GetChannelInfo
    (SubsystemType.AnalogInput, 0 ));
```

Visual Basic
```
Dim Ch0 As New ChannelListEntry (
  ainSS.SupportedChannels.GetChannelInfo
    (SubsystemType.AnalogInput, 0 ))
```

Note: It is recommended that you set the gain (see page 210) and inhibition value (page 211) for each ChannelListEntry object after you create it. However, it is possible to set or change these values after the ChannelListEntry object is added to the ChannelList.

The following example shows how to use the **Add** method to add ChannelListEntry object Ch0 to the end of a ChannelList:

Visual C#
```
AinSS.ChannelList.Add(Ch0);
```

Visual Basic
```
AinSS.ChannelList.Add(Ch0)
```

Inserting Channels in the Channel List

The **ChannelList.Insert** method inserts a channel at the specified index of a ChannelList object, incrementing all higher index entries by 1, and returns the index of the added channel. You can specify the channel to insert in one of the following ways:

- By physical channel number
- By channel name
- By ChannelListEntry object

The following sections describe these methods.

Inserting a Channel By Physical Channel Number

This method is the simplest way to insert channels into the ChannelList object, particularly if you are inserting channels that are native to the subsystem type (such as analog input channels on an analog input subsystem).

For native channels, the physical channel number always equals the logical channel number. While non-native channels, such as digital inputs that are streamed through the analog input subsystem, can also be inserted this way, the physical channel number is not the same as the logical channel number, so you may find it easier to insert the channel by name or by ChannelListEntry object instead.

A new ChannelListEntry object is returned for each physical channel that is inserted this way. Refer to page 208 for more information on ChannelListEntry objects.

The following example shows how to use the **Insert** method to insert physical channel 3 at index 0 of the ChannelList for an analog input subsystem. The channel that was formally at index 0 is now at index 1.

Visual C#
```
ch = AinSS.ChannelList.Insert(0, 3);
```

Visual Basic
```
ch = AinSS.ChannelList.Insert(0, 3)
```

Inserting a Channel By Channel Name

The channel name is the name that you assigned to the channel using the SupportedChannelInfo class, described on page 150. A new ChannelListEntry object is returned for each channel that is inserted this way. Refer to page 208 for more information on ChannelListEntry objects.

The following example shows how to use the **Insert** method to insert a channel named Ain3 at index 0 of the ChannelList for an analog input subsystem. The channel that was formally at index 0 is now at index 1.

Visual C#
```
ch = AinSS.ChannelList.Insert(0, "Ain3");
```

Visual Basic
```
ch = AinSS.ChannelList.Insert(0, "Ain3")
```

Inserting a Channel By ChannelListEntry Object

This method is useful if you want a more generic approach to inserting channels. This approach frees you from keeping track of physical channel numbers and their names.

To get a ChannelListEntry object, use the **ChannelListEntry** constructor within the ChannelListEntry class, specifying the SupportedChannelInfo object for each channel that you want to sample or update. See page 150 for more information on SupportedChannelInfo objects.

This example creates a ChannelListEntry called Ch3 for physical channel 3 of the analog input subsystem, using all the information contained in SupportedChannelInfo for that channel.

Visual C#
```
ChannelListEntry Ch3 = new ChannelListEntry (
  ainSS.SupportedChannels.GetChannelInfo
    (SubsystemType.AnalogInput, 3 ));
```

Visual Basic
```
Dim Ch3 As New ChannelListEntry (
  ainSS.SupportedChannels.GetChannelInfo
    (SubsystemType.AnalogInput, 3 ))
```

Note: It is recommended that you set the gain (see page 210) and inhibition value (page 211) for each ChannelListEntry object after you create it. However, it is possible to set or change these values after the ChannelListEntry object is added to the ChannelList.

The following example shows how to use the **Insert** method to insert ChannelListEntry object Ch3 at index 0 of the ChannelList. The channel that was formally at index 0 is now at index 1.

Visual C#
```
AinSS.ChannelList.Insert(0, Ch3);
```

Visual Basic
```
AinSS.ChannelList.Insert(0, Ch3)
```

Replacing Channels in the ChannelList

The ChannelList.Item ([]) property replaces a ChannelListEntry object at the specified index of the ChannelList. An exception is raised if an entry does not exist at the specified index.

The following example shows how to use the Item ([]) property to replace the ChannelListEntry object at index 1 of the ChannelList with ChannelListEntry object Ch3:

Visual C#
```
AinSS.ChannelList[1] = Ch3;
```

Visual Basic
```
AinSS.ChannelList(1) = Ch3
```

Removing Channels from the Channel List

To remove a ChannelListEntry from the ChannelList object, use the **ChannelList.Remove** method. This method removes the first instance of the specified ChannelListEntry object from the ChannelList object, decrementing all higher index entries by 1.

The following example shows how to remove the first instance of ChannelListEntry object Ch0 from the ChannelList object using the **Remove** method:

Visual C#

```
AinSS.ChannelList.Remove(Ch0);
```

Visual Basic

```
AinSS.ChannelList.Remove(Ch0)
```

Setting the Gain of a ChannelListEntry

The voltage range divided by the gain determines the effective range for a channel. For example, if your device provides a voltage 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.

To determine if the subsystem supports programmable gain, use the **SupportsProgrammableGain** property of the appropriate subsystem. If this property returns a value of True, programmable gain is supported.

To determine the number of gains the subsystem supports, use the **NumberofSupportedGains** property of the appropriate subsystem. To list all of the gain values supported by the subsystem, use the **SupportedGains** property.

The simplest way to specify the gain for a channel is by using a single-value operation. (In this case, a ChannelListEntry object is not used.) Refer to page 176 for more information on single-value analog input operations; refer to page 180 for more information on single-value analog output operations.

If you are using a ChannelListEntry object, specify or return the gain for each ChannelListEntry object using the **ChannelListEntry.Gain** property.

This example shows how to apply a gain of 2 to a ChannelListEntry called Ch0.

Visual C#

```
Ch0.Gain = 2;
```

Visual Basic

```
Ch0.Gain = 2
```

You can also apply gain to a ChannelListEntry in the ChannelList, as shown below; this example applies a gain of 2 to the ChannelListEntry at index 0 of the ChannelList:

Visual C#

```
AinSS.ChannelList[0].Gain = 2;
```

Visual Basic

```
AinSS.ChannelList(0).Gain = 2
```

Note: The driver sets the actual gain as closely as possible to the number specified. You can read back the exact gain after configuring the subsystem using the **Gain** property. If your subsystem does not support programmable gain, enter a value of 1 (the default value) for the gain.

Inhibiting Channels in a Channel List

If supported by your subsystem, you can inhibit data from being returned by the ChannelListEntry object. This feature is useful if you want to discard values that are acquired by specific channels.

To determine if a subsystem supports inhibition, use the **SupportsChannelListInhibit** property inherited from the SubsystemBase class. If this property returns a value of True, the subsystem supports channel inhibition.

Using the **Inhibit** property of the ChannelListEntry class, you can enable or disable inhibition for each ChannelListEntry object. If you set this property to True, the acquired value is discarded after the channel entry is sampled. If you set this property to False (the default value), the acquired value is stored after the channel entry is sampled.

This example shows how to set the channel inhibit value of the ChannelListEntry called Ch0 to True:

Visual C#

```
Ch0.Inhibit = 1;
```

Visual Basic

```
Ch0.Inhibit = 1
```

You can also set the inhibit value of a ChannelListEntry in the ChannelList, as shown below; this example sets the inhibit value to True for the ChannelListEntry at index 3 of the ChannelList:

Visual C#

```
AinSS.ChannelList[3].Inhibit = 1;
```

Visual Basic

```
AinSS.ChannelList(3).Inhibit = 1
```

Getting Information about Channels in the ChannelList Object

You can get information about the contents of a ChannelList object using the following methods:

- **ChannelList.Contains** method – Determines whether a specified ChannelListEntry object is contained in the ChannelList.
- **ChannelList.IndexOf** method – Searches for a specified channel (specified by physical channel or ChannelListEntry object) in the ChannelList and returns the zero-based index of the first occurrence within the ChannelList.
- **ChannelList.CGLDepth** property – Returns the maximum number of channels or ChannelListEntry objects that the ChannelList can contain.

Setting up a Clock Source

The DT-Open Layers for .NET Class Library defines internal and external clock sources, described in the following subsections. Note that single-value operations do not use clocks.

Note: Some subsystems allow you to read or update multiple channels on a single clock pulse. You can determine whether multiple channels are read or updated on a single clock pulse by using the **Clock.SupportsSimultaneousClocking** property.

In addition, some subsystems support different clock frequencies depending on whether the device is powered by an internal power source or an external power source. To determine if your device supports an internal power source or an external power source, use the **PowerSource** property inherited from the SubsystemBase class.

Internal Clock Source

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

To determine if the subsystem supports an internal clock, use the **Clock.SupportsInternalClock** property. If this property returns a value of True, an internal clock is supported.

To determine the maximum frequency supported by the internal clock, use the **Clock.MaxFrequency** property. To determine the minimum frequency supported by the internal clock, use the **Clock.MinFrequency** property.

Specify the clock source as internal using the **Clock.Source** property. Then, use the **Clock.Frequency** property to specify the frequency at which to pace the operation.

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.

The driver sets the frequency of the internal clock as close as possible to the value that you specified in the **Frequency** property. You can determine the actual frequency that was set on the hardware by reading the value of the **Frequency** property after the subsystem has been configured (using the **Config** method).

External Clock Source

The external clock is a clock source attached to the device that paces data acquisition or output for each channel in the channel 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 **Clock.SupportsExternalClock** property. If this property returns a value of True, an external clock is supported.

To determine the maximum external clock divider that the subsystem supports, use the **Clock.MaxExtClockDivider** property. To determine the minimum external clock divider that the subsystem supports, use the **Clock.MinExtClockDivider** property.

Specify the clock source as external using the **Clock.Source** property. Then, use the **Clock.ExtClockDivider** property to set or get the clock divider that is used to determine the frequency of the external clock source. The frequency of the external clock input divided by the external clock divider determines the frequency at which to pace the operation.

Setting Up a Trigger Type

Note: Single-value operations do not use triggers.

The DT-Open Layers for .NET Class Library provides the Trigger class that can be used to set up a start trigger, and the ReferenceTrigger class that can be used to set up a reference trigger, if supported by your device. The following trigger types are available for the start and reference triggers:

- Software
- TTLPos
- TTLNeg
- ThresholdPos

- ThresholdNeg
- DigitalEvent

For devices that support a start trigger and reference trigger for performing continuous pre-and post-trigger analog input operations, specify the start trigger type using the **AnalogInputSubsystem.Trigger.TriggerType** property and specify the reference trigger type using the **AnalogInputSubsystem.ReferenceTrigger.TriggerType** property; refer to page 185 for more information on pre- and post-trigger operations using a start and reference trigger.

For devices that support continuous post-trigger and about-trigger operations without using a reference trigger, specify the post-trigger source using the **AnalogInputSubsystem.Trigger.TriggerType** property; refer to page 187 for more information on post-trigger operations and page 193 for more information on about-trigger operations.

For legacy devices that support a pre-trigger source without using a reference trigger, use the **AnalogInputSubsystem.Trigger.PreTriggerSource** property of the Trigger class; see page 190 for more information on pre-trigger operations. To specify a retrigger source, use the **AnalogInputSubsystem.TriggeredScan.RetriggerSource** property; see page 228 for more information on retriggers.

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

Software 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 for the start trigger, use the **Trigger.SupportsSoftwareTrigger** property. If this property returns a value of True, a software trigger is supported.

To determine if the subsystem supports a software trigger for the reference trigger, use the **ReferenceTrigger.SupportsSoftwareTrigger** property. If this property returns a value of True, a software trigger is supported.

TTLPos Trigger Source

The TTLPos trigger source is an external digital (TTL) signal attached to the device. The trigger occurs when the device detects a transition on the rising edge of the digital TTL signal.

To determine if the subsystem supports a TTLPos trigger for a start trigger, use the **Trigger.SupportsPosExternalTTLTrigger** property. If this property returns a value of True, a TTLPos trigger is supported.

To determine if the subsystem supports a TTLPos trigger for a reference trigger, use the **ReferenceTrigger.SupportsPosExternalTTLTrigger** property. If this property returns a value of True, a TTLPos trigger is supported.

To determine if the subsystem supports a TTLPos trigger for a single-value operation, use the **Trigger.SupportsSvPosExternalTTLTrigger** property. If this property returns a value of True, a TTLPos trigger is supported.

TTLNeg Trigger Source

The TTLNeg trigger source is an external digital (TTL) signal attached to the device. The trigger occurs when the device detects a transition on the falling edge of the digital TTL signal.

To determine if the subsystem supports a TTLNeg trigger for a start trigger, use the **Trigger.SupportsNegExternalTTLTrigger** property. If this property returns a value of True, a TTLNeg trigger is supported.

To determine if the subsystem supports a TTLNeg trigger for a reference trigger, use the **ReferenceTrigger.SupportsNegExternalTTLTrigger** property. If this property returns a value of True, a TTLNeg trigger is supported.

To determine if the subsystem supports a TTLNeg trigger for a single-value operation, use the **Trigger.SupportsSvNegExternalTTLTrigger** property. If this property returns a value of True, a TTLNeg trigger is supported.

ThresholdPos Trigger Source

A threshold trigger is generally either an analog signal from an analog input channel or an external analog signal attached to the device. A positive analog threshold (ThresholdPos) trigger occurs when the device detects a positive-going signal that crosses a threshold value.

To determine if the subsystem supports a ThresholdPos trigger for the start trigger, use the **Trigger.SupportsPosThresholdTrigger** property. If this property returns a value of True, a ThresholdPos trigger is supported.

To determine if the subsystem supports a ThresholdPos trigger for the reference trigger, use the **ReferenceTrigger.SupportsPosThresholdTrigger** property. If this property returns a value of True, a ThresholdPos trigger is supported.

To determine which channels support a threshold trigger for the start trigger, use the **Trigger.SupportedThresholdTriggerChannels** property. To set the channel that you want to use for the threshold start trigger, use the **Trigger.ThresholdTriggerChannel** property.

To determine which channels support a threshold trigger for the reference trigger, use the **ReferenceTrigger.SupportedThresholdTriggerChannels** property. To set the channel that you want to use for the threshold reference trigger, use the **ReferenceTrigger.ThresholdTriggerChannel** property.

On some devices, the threshold level is set using an analog output subsystem on the device. On other devices, you set the threshold level using the **Trigger.Level** property (for the start trigger) or **ReferenceTrigger.Level** property (for the reference trigger). By default, the trigger threshold value is in voltage unless specified otherwise for the device; see the user's manual for your device for valid threshold value settings.

Note: The threshold level set by the **Trigger.Level** or **ReferenceTrigger.Level** property depends on the voltage range and gain of the subsystem. For example, if the voltage range of the analog input subsystem is ±10 V, and the specified gain is 1, specify a threshold voltage level within ±10 V. Likewise, if the voltage range of the analog input subsystem is ±10 V, and the specified gain is 10, specify a threshold voltage level within ±1 V. Refer to your device documentation for details on how to specify the threshold value for your device.

ThresholdNeg Trigger Source

A threshold trigger is generally either an analog signal from an analog input channel or an external analog signal attached to the device. A negative analog threshold trigger (ThresholdNeg) occurs when the device detects a negative-going signal that crosses a threshold value.

To determine if the subsystem supports a ThresholdNeg trigger for the start trigger, use the **Trigger.SupportsNegThresholdTrigger** property. If this property returns a value of True, a ThresholdNeg trigger is supported.

To determine if the subsystem supports a ThresholdNeg trigger for the reference trigger, use the **ReferenceTrigger.SupportsNegThresholdTrigger** property. If this property returns a value of True, a ThresholdNeg trigger is supported.

To determine which channels support a threshold trigger for the start trigger, use the **Trigger.SupportedThresholdTriggerChannels** property. To set the channel that you want to use for the threshold start trigger, use the **Trigger.ThresholdTriggerChannel** property.

To determine which channels support a threshold trigger for the reference trigger, use the **ReferenceTrigger.SupportedThresholdTriggerChannels** property. To set the channel that you want to use for the threshold reference trigger, use the **ReferenceTrigger.ThresholdTriggerChannel** property.

On some devices, the threshold level is set using an analog output subsystem on the device. On other devices, you set the threshold level using the **Trigger.Level** property (for the start trigger) or the **ReferenceTrigger.Level** property (for the reference trigger). By default, the trigger threshold value is in voltage unless specified otherwise for the device; see the user's manual for your device for valid threshold value settings.

Note: The threshold level set by the **Trigger.Level** or **ReferenceTrigger.Level** property depends on the voltage range and gain of the subsystem. For example, if the voltage range of the analog input subsystem is ±10 V, and the specified gain is 1, specify a threshold voltage level within ±10 V. Likewise, if the voltage range of the analog input subsystem is ±10 V, and the specified gain is 10, specify a threshold voltage level within ±1 V. Refer to your device documentation for details on how to specify the threshold value for your device.

DigitalEvent Trigger Source

For a DigitalEvent trigger source, a trigger is generated when an external digital event occurs.

To determine if the subsystem supports a DigitalEvent trigger for the start trigger, use the **Trigger.SupportsDigitalEventTrigger** property. If this property returns a value of True, a DigitalEvent trigger is supported.

To determine if the subsystem supports a DigitalEvent trigger for the reference trigger, use the **ReferenceTrigger.SupportsDigitalEventTrigger** property. If this property returns a value of True, a DigitalEvent trigger is supported.

Sync Bus Trigger Source

For devices that support connecting multiple devices together in a master/slave relationship using Sync Bus (RJ45) connectors, the slave device may support the ability to configure a Sync Bus trigger source as the reference trigger.

To determine if the subsystem supports a Sync Bus trigger source as the reference trigger, use **ReferenceTrigger.SupportsSyncBusTrigger** property. If this property returns a value of True, a Sync Bus trigger is supported.

Use the Sync Bus trigger source as the reference trigger if you want the slave device to receive a Sync Bus trigger from one of the other devices to stop pre-trigger acquisition and start post-trigger acquisition.

If you want to set the slave module to receive a Sync Bus trigger as the start trigger source, set the synchronization mode of the device to Slave using the **SynchronizationMode** property of for the subsystem, described on page 203; the Sync Bus trigger is used by the slave module as the start trigger source by default.

Setting up a Post-Trigger Scan Count

On devices that support a reference trigger for performing continuous pre- and post-trigger analog input operations, you can specify how many samples to acquire after the reference trigger occurs using the **AnalogInputSubsystem.ReferenceTrigger.PostTriggerScanCount** property.

To determine if your device supports the ability to specify the number of post-trigger samples to acquire, use the **AnalogInputSubsystem.ReferenceTrigger. SupportsPostTriggerScanCount** property.

Setting up Buffers

Note: Single-value operations do not use buffers.

Continuous analog input and analog output operations require buffers in which to store data. For input operations, a queue exists to hold the buffers that are empty and ready for input. For output operations, the queue holds buffers that you have filled with data and are ready for output.

To determine if the subsystem supports buffering, use the **SupportsBuffering** property within the appropriate subsystem class. If this property returns a value of True, buffering is supported.

If you want to acquire one buffer of data from one channel using a continuous analog input operation, use the **AnalogInputSubsystem.GetOneBuffer** method; this method allocates an OlBuffer object of the size you specify and acquires one buffer of data for you.

For all other operations, use the **OlBuffer** constructor within the OlBuffer class to create an OlBuffer object for use with an analog input or analog output subsystem. The library automatically allocates an internal data buffer, which is encapsulated by the OlBuffer object. You specify the subsystem with which to associate the OlBuffer object as well the size (in samples) of the internal buffer to allocate.

If desired, you can use the **OlBuffer.Tag** property, if desired, to name the buffer with the contents that are contained in the buffer.

Note: If you use the **ReturnCjcTemperaturesInStream** property, described on page 159, to return CJC data in the data stream, ensure that you set the size of the internal buffer to be twice as large to accommodate the returned CJC values for each channel (number of samples per channel x 2).

If you set the size of the internal buffer that is encapsulated by an OlBuffer object and later you want to change the size, call the **OlBuffer.Reallocate** method. This method reallocates the internal buffer to the specified number of samples; the initial internal buffer is deallocated and any data that it contained is lost.

The **AnalogInputSubsystem.GetOneBuffer** method uses one buffer. Other continuous analog input operations require a minimum of two OlBuffer objects. Continuous analog output operations require a minimum of two OlBuffer objects if **WrapSingleBuffer** is False; if **WrapSingleBuffer** is True, one OlBuffer object is required.

Once you have created the OlBuffer objects for multiple buffer operations (and, for output operations, filled the corresponding internal buffers with data), put the OlBuffer objects on the queue using the **BufferQueue.QueueBuffer** method of the appropriate subsystem.

The following example shows how to create multiple OlBuffer objects for a continuous analog input operation and put them on the queue for the analog input subsystem. In this example, an internal buffer of 1024 samples is allocated when the OlBuffer object is created:

Visual C#

```
// Create the buffers
for (int i=0; i<4; ++i)
{
  AinBuffer[i] = new OlBuffer (1024, ainSS);
  // Put the buffers on the queue
  ainSS.BufferQueue.QueueBuffer (AinBuffer[i]);
}
```

Visual Basic

```
While i < 4
  ' Create the buffers
  AinBuffers(i) = New OlBuffer(1024, ainSS)
  ' Put the buffers on the queue
  ainSS.BufferQueue.QueueBuffer(AinBuffers(i))
  i += 1
End While
```

When you start a continuous operation, the device takes up to eight OlBuffer objects from the subsystem queue and begins filling them (for input operations) or outputting data from them (for output operations) at the specified clock rate. The state of these objects changes from queued to inprocess.

About QueuedCount and InProcessCount

You can determine the number of OlBuffer objects that are on the subsystem queue by using the **BufferQueue.QueuedCount** property. You can determine the number of OlBuffer objects that are inprocess by using the **BufferQueue.InProcessCount** property.

Every time an OlBuffer object transitions from the queued state to the inprocess state, the value of the **QueuedCount** property decreases by 1 and the value of the **InProcessCount** property increases by 1. For example, assume that you call **QueueBuffer** for 10 OlBuffer objects; the **QueuedCount** is 10 and the **InProcessCount** is 0. Once you call **Start** for the subsystem, up to 8 OlBuffer objects are moved from the queued state to the inprocess state. **QueuedCount** is now 2 and **InProcessCount** is 8.

If you do not put the OlBuffer objects back on the queue as they are completed, the **QueuedCount** decreases while the **InProcessCount** remains the same (as a new inprocess buffer replaces a completed buffer) until the **QueuedCount** gets to 0, then the **InProcessCount** starts decreasing until all the OlBuffer objects are completed, as shown below:

Table 70: InProcessCount Example

Completed Buffers	QueueCount	InProcessCount
0	10	0
0	2	8
1	1	8
2	0	8
3	0	7
4	0	6
5	0	5
6	0	4
7	0	3
8	0	2
9	0	1
10	0	0

Buffer Completion Events

Note: Buffer completion events are not generated if you use the **AnalogInputSubsystem.GetOneBuffer** method. This is a synchronous method that does not return until the buffer has been acquired or the timeout value has expired.

One or more of the following events is generated when a buffer is completed:

- **BufferDoneEvent** – For input operations, this event is generated when the internal buffer of the OlBuffer object has been filled with post-trigger data. For output operations, this event is generated when all the data in the internal buffer of the OlBuffer object has been output. Refer to page 259 for more information on this event.
- **PreTriggerBufferDoneEvent** – For input operations only, this event is generated when the internal buffer of the OlBuffer object has been filled with pre-trigger data. Refer to page 261 for more information on this event.
- **QueueStoppedEvent** – This event occurs when you stop a continuous analog I/O operation with **Stop** or **Abort**. Refer to page 262 for more information on this event.

- **IOCompleteEvent –** For analog input operations that use a reference trigger whose trigger type is something other than software (none), this event occurs when the last post-trigger sample is copied into the user buffer; devices that do not support a reference trigger will never receive this event for analog input operations.

  For analog output operations, this event is generated when the last data point has been output from the analog output channel. Refer to page 263 for more information on this event.
- **QueueDoneEvent –** This event is generated when no OlBuffer objects are available on the queue and the operation stops. Refer to page 265 for more information on this event.

Handling Input Buffers

Each time a BufferDoneEvent or PreTriggerBufferDoneEvent event is raised, your application program must handle the event or you will lose the data in the internal buffer of the OlBuffer object. Refer to page 257 for more information about handling events and buffers.

You can post-process OlBuffer objects, if you wish. One technique for doing this is to allocate an array that will hold the OlBuffer objects as they are completed. When the BufferDoneEvent or PreTriggerBufferDoneEvent event occurs, move the OlBuffer object into a array. When the operation is complete, process the OlBuffer objects in your array.

For continuous analog input operations, use one of the following methods to copy the data from the internal buffer of an OlBuffer object into a user-declared array/variable (the data type of this array/variable is dictated by the method/property you choose):

Note: For ease of use, all of these methods allocate the returned array to the correct size. Simply declare an array of the appropriate type for use with one these methods.

- **OlBuffer.GetDataAsRawByte –** Copies the data, as raw counts, from the internal buffer of the OlBuffer object into a user-declared array of bytes. You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of bytes.

  Note: This method is useful when writing binary data to a file. Since each sample takes more than one array entry, other uses may be limited.
- **OlBuffer.GetDataAsRawInt16 –** Used when the resolution of the subsystem is 16 bits or less and when the data encoding is twos complement, copies the data, as raw counts, from the internal buffer of the OlBuffer object into a user-declared array of signed, 16-bit integers (short). You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

To use this method, first declare an array of signed, 16-bit integers (short).

- **OlBuffer.GetDataAsRawUInt16 –** Used when the resolution of the subsystem is 16 bits or less and when the data encoding is binary, copies the data, as raw counts, from the internal buffer of the OlBuffer object into a user-declared array of unsigned, 16-bit integers (ushort). You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of unsigned, 16-bit integers (ushort).

- **OlBuffer.GetDataAsRawUInt32 –** Used when the resolution of the subsystem is greater than 16 bits, copies the data, as raw counts, from the internal buffer of the OlBuffer object into a user-declared array of unsigned 32-bit integers (uint). You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of unsigned, 32-bit integers (uint).

- **OlBuffer.GetDataAsSensor –** Converts the data from the internal buffer of the OlBuffer object into sensor values using the specified sensor gain and offset (described on page 156), and copies this data into a user-declared array of 64-bit floating-point values (double). You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of 64-bit floating-point (double) values.

- **OlBuffer.GetDataAsVolts –** Converts the data from the internal buffer of the OlBuffer object into voltages, and copies this data into a user-declared array of 64-bit floating-point (double) values. You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of 64-bit floating-point (double) values.

Note: If the **AnalogInputSubsystem.ReturnCjcTemperaturesInStream** property is set to True, the CJC values, in temperature, are interleaved with the channel data; therefore, the returned array will be twice the number of valid samples (**OlBuffer.ValidSamples**). Refer to page 159 for more information on the **ReturnCjcTemperaturesInStream** property.

- **OlBuffer.GetDataAsVoltsByte –** For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into voltage values, and then copies these voltage values into a user-declared array of bytes. Each temperature value is stored as an Int32, and takes 4 bytes.

  To use this method, first declare an array of bytes.

- **OlBuffer.GetDataAsCurrent –** For a specified ChannelListEntry, converts the data from the internal buffer of the OlBuffer object into current values, in Amperes, and copies this data into a user-declared array of 64-bit floating-point (double) values. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

To use this method, first declare an array of 64-bit floating-point (double) values.

- **OlBuffer.GetDataAsResistance –** For a specified ChannelListEntry, converts the data from the internal buffer of the OlBuffer object into resistance values, in ohms, and copies this data into a user-declared array of 64-bit floating-point (double) values. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of 64-bit floating-point (double) values.

- **OlBuffer.GetDataAsTemperatureByte –** For a specified ChannelListEntry, converts the data from the internal buffer of the OlBuffer object into temperatures based on the specified thermocouple, RTD, or thermistor (described on page 157, page 160, and page 172), and copies this data into a user-declared array of bytes. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of bytes.

  Note: If the **AnalogInputSubsystem.SupportsTemperatureInDataStream** property is True, this method raises an exception if the **SupportedChannelInfo.ThermocoupleType** is None. If the **AnalogInputSubsystem.ReturnCjcTemperaturesInStream** property is set to True, the CJC values are interleaved with the channel data; therefore, the returned array will be twice the number of valid samples (**OlBuffer.ValidSamples**). Refer to page 159 for more information on the **ReturnCjcTemperaturesInStream** property.

- **OlBuffer.GetDataAsTemperatureDouble –** For a specified ChannelListEntry, converts the data from the internal buffer of the OlBuffer object into temperatures based on the specified thermocouple, RTD, or thermistor (described on page 157, page 160, and page 172), and copies this data into a user-declared array of 64-bit floating-point (double) values. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of 64-bit floating-point (double) values.

  Note: If the **AnalogInputSubsystem.SupportsTemperatureInDataStream** property is True, this method raises an exception if the **SupportedChannelInfo.ThermocoupleType** is None.

  If the **AnalogInputSubsystem.ReturnCjcTemperaturesInStream** property is set to True, the CJC values are interleaved with the channel data; therefore, the returned array will be twice the number of valid samples (**OlBuffer.ValidSamples**). Refer to page 159 for more information on the **ReturnCjcTemperaturesInStream** property.

- **OlBuffer.GetDataAsRpm –** For a specified ChannelListEntry, converts the tachometer data from the internal buffer of an OlBuffer object into RPM (rotations per minute) values, and then copies these values into a user-declared array of 64-bit floating-point (double) values. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of 64-bit floating-point (double) values.

- **OlBuffer.GetDataAsStrain** – Converts the data from the internal buffer of an OlBuffer object into microstrain values, and then copies these microstrain values into a user-declared array of 64-bit floating-point (double) values. You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer.

  To use this method, first declare an array of 64-bit floating-point (double) values.
- **OlBuffer.GetDataAsBridgeBasedSensor** – Converts the data from the internal buffer of an OlBuffer object into the native engineering units of the full-bridge-based sensor, and then copies these values into a user-declared array of 64-bit floating-point (double) values. You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer.

  To use this method, first declare an array of 64-bit floating-point (double) values.
- **OlBuffer.GetDataAsNormalizedBridgeOutput** – Converts the data from the internal buffer of an OlBuffer object into the normalized output value of the bridge, in volts, and then copies these values into a user-declared array of 64-bit floating-point (double) values. You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer.

  To use this method, first declare an array of 64-bit floating-point (double) values.
- **OlBuffer.Item** property ([]) – Copies the raw count value at the specified index of the buffer specified by the OlBuffer object into a user-declared signed, 32-bit integer variable (int).

When you have finished copying the data from the internal buffer of the OlBuffer object, you can put the OlBuffer object back on the queue for the analog input subsystem using the **AnalogInputSubsystem.BufferQueue.QueueBuffer** method.

See the example for the event BufferDoneEvent starting on page 259 for an example of using the **GetDataAsSensor** method to handle input buffers.

Handling Output Buffers

For continuous analog output operations, you need to create an array and fill it with data, then copy this data from the array to the internal buffer of the OlBuffer object using one of the following methods:

- **OlBuffer.PutDataAsRaw** – Copies raw counts from a user-specified array into the internal buffer of the OlBuffer object. This is an overloaded method that allows you to copy all the data from the array into the buffer or only the data for a specific ChannelListEntry in the array into the buffer. Note that if the ChannelListEntry occurs more than once in the array, the data for each occurrence of the ChannelListEntry is copied.

  If your subsystem supports a resolution of 16-bits or less, declare an array of unsigned, 16-bit integers (ushort) for use with this method.

  If your subsystem supports a resolution greater than 16 bits, declare an array of unsigned, 32-bit integers (uint) for use with this method.

- **OlBuffer.PutDataAsVolts –** Copies voltages from a user-specified array into the internal buffer of the OlBuffer object. This is an overloaded method that allows you to copy all the data from the array into the buffer or only the data for a specific ChannelListEntry in the array into the buffer. Note that if the ChannelListEntry occurs more than once in the array, the data for each occurrence of the ChannelListEntry is copied.

  Declare an array of 64-bit floating-point values (double) for use with this method.

When you have finished copying the data into the internal buffer of the OlBuffer object, put the OlBuffer object back on the queue for the analog output subsystem using the **AnalogOutputSubsystem.BufferQueue.QueueBuffer** method.

The following example shows how to create an OlBuffer object, fill the internal buffer of this OlBuffer object with 100 samples, and put the OlBuffer object on the analog output subsystem queue:

Visual C#
```
// Allocate a buffer of 100 samples
DacBuffer = new OlBuffer (100, aoutSS);
//Create an array of data
for (int i = 0; i < 100; i++)
   {
      data[i] = i;
   }
// Copy the raw data to the buffer
DacBuffer.PutDataAsRaw (data);
// Queue the buffer for output
aoutSS.BufferQueue.QueueBuffer (DacBuffer);
```

Visual Basic
```
' Allocate a buffer of 100 samples
DacBuffer = New OlBuffer(100, aoutSS)
' Create an array of data
Dim i As Integer
   For i = 0 To 99
      data(i) = i
   Next i
' Copy the raw data to the buffer
DacBuffer.PutDataAsRaw(data)
' Queue the buffer for output
aoutSS.BufferQueue.QueueBuffer(DacBuffer)
```

Moving Data from an Inprocess OlBuffer Object

Some devices allow you to transfer data from the internal buffer of an OlBuffer object while it is being filled. Typically, you would use this method when a continuous analog input operation is running slowly.

To determine if the subsystem supports this capability, use the **AnalogInputSubsystem.SupportsInProcessFlush** property. If this property returns a value of True, your subsystem supports moving data from the internal buffer as it is being filled.

Use the **AnalogInputSubsystem.MoveFromBufferInprocess** method to move data from the internal buffer of an OlBuffer object that is in the process of being filled to the internal buffer of a new OlBuffer object, which has not been put on the queue.

Note: Some devices transfer data to the host in segments instead of one sample at a time. For example, data from the DT3010 device 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 **MoveFromBufferInprocess** 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.

Getting Information about a Buffer

The DT-Open Layers for .NET Class Library provides the following additional properties for getting information about buffers:

- **OlBuffer.BufferSizeInBytes** – Returns the size, in bytes, of the internal data buffer that is encapsulated by the OlBuffer object.
- **OlBuffer.BufferSizeInSamples** – Returns the size, in samples, of the internal data buffer that is encapsulated by the OlBuffer object.
- **OlBuffer.ChannelListOffset** – Returns the index into the ChannelList that corresponds to the first sample in the internal buffer of the OlBuffer object.
- **OlBuffer.Encoding** – Returns the data encoding for the raw data (binary or twos complement) in the internal buffer of the OlBuffer object.
- **OlBuffer.RawDataFormat** – Returns the format of the raw data (Int16, Uint16, Int32, Float (32-bit float), or Double (64-bit float)) in the internal buffer of the OlBuffer object.
- **OlBuffer.Resolution** – Returns the resolution of the subsystem that is associated with the OlBuffer object.
- **OlBuffer.SampleSizeInBytes** – Returns the size of a sample, in bytes. Typically, each sample requires 2 bytes.
- **OlBuffer.State** property – Returns the state of the OlBuffer object. Valid states are as follows:
  - Idle – The OlBuffer object has been created, but has not been queued to a subsystem.
  - Queued – The OlBuffer object has been queued to a subsystem with **OlBuffer.QueueBuffer**.
  - InProcess – The OlBuffer object has been sent to the device driver for processing. A maximum of eight OlBuffer objects can be inprocess at one time.
  - Completed – For an input operation, the internal buffer of the OlBuffer object has been filled, and the OlBuffer object has not been put back on queue for the subsystem. For an output operation, all the data in the internal buffer of the OlBuffer object has been output, and the OlBuffer object has not been put back on the queue for the subsystem.
  - Released – The internal data buffer of the OlBuffer object has been deallocated by calling **OlBuffer.Dispose**.

- **OlBuffer.ValidSamples** – Returns the number of valid samples in the internal buffer of the OlBuffer object.

  For analog input operations, the **ValidSamples** property is set to the number of samples in the completed buffer under normal circumstances. However, in some cases, like if **Abort** is called in the middle of an operation, **ValidSamples** reflects the number of samples in the buffer when **Abort** was called. In addition, if **Abort** or **Stop** is called, any OlBuffer object whose state is Inprocess will have a **ValidSamples** of 0.

  Note: If the **AnalogInputSubsystem.ReturnCjcTemperaturesInStream** property is set to True, the CJC values will be interleaved with the channel data; therefore, the returned array will be twice the number of valid samples (**OlBuffer.ValidSamples**). Refer to page 159 for more information on the **ReturnCjcTemperaturesInStream** property.

  For analog output operations, **ValidSamples** is always equal to the maximum number of samples that the buffer was allocated to hold.

- **OlBuffer.VoltageRange** – Returns the upper limit and lower limit of the voltage range for the associated subsystem.

Cleaning up Buffers

When you are finished performing continuous analog I/O operations, use can use one of the following methods to clean up the OlBuffer objects:

- **BufferQueue.DequeueBuffer** – Removes and returns the OlBuffer object at the front of the queue.
- **BufferQueue.FreeAllQueuedBuffers** – Removes all OlBuffer objects from the queue and deallocates the internal data buffers that are encapsulated by them.

Setting Triggered Scan Mode

Note: Single-value operations do not support triggered scan mode.

On some devices, the analog input subsystem supports triggered scan mode. In triggered scan mode, the device scans the channel list a specified number of times when it detects the specified retrigger source, acquiring the data for each channel in the channel list. The conversion rate of each channel in the scan is determined by the frequency of the A/D sample clock; refer to page 212 for more information on clock sources. The conversion rate of each scan is determined by the period between retriggers.

To determine if the subsystem supports triggered scan mode, use the **AnalogInputSubsystem.SupportsTriggeredScan** property. If this property returns a value of True, triggered scan mode is supported.

To enable (or disable) triggered scan mode, use the **TriggeredScan.Enabled** property.

To determine the maximum number of times that the device can scan the channel list per retrigger, use the **TriggeredScan.MaxMultiScanCount** property.

Use the **TriggeredScan.MultiScanCount** property to set or get the number of times to scan the channel list per retrigger.

Use the **TriggeredScan.RetriggerSource** property to specify the retrigger source; the retrigger source can be any of the supported trigger sources. Refer to page 213 for more information on the supported trigger sources. The following subsections describe considerations when using a software or external retrigger source.

Using a Software Retrigger Source

If you are using a software retrigger source, specify the period between retriggers using the **TriggeredScan.RetriggerFrequency** property.

You can determine the maximum retrigger frequency supported by the subsystem using the **TriggeredScan.MaxRetriggerFreq** property. You can determine the minimum retrigger frequency supported by the subsystem using the **TriggeredScan.MinRetriggerFreq** property.

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

We recommend that you set the retrigger frequency as follows:

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

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

For example, if you have 512 ChannelListEntry objects in the ChannelList object, are scanning the channel list 256 times every trigger or retrigger, and are using an A/D sample clock with a frequency of 1 MHz, set the maximum retrigger frequency to 7.62 Hz, since

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

Using an External Retrigger Source

If you are using an external retrigger source, 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 the event DriverRunTimeErrorEvent. Refer to page 266 for more information on this event.

Refer to page 213 and to your device/device driver documentation for supported external retrigger sources.

Performing Digital I/O Operations

Using the DT-Open Layers for .NET Class Library, you can perform the following types of digital input operations:

- Single-value digital input, described below
- Single-value digital output, described on page 230
- Continuous digital input (interrupt-on-change), described on page 230

Single-Value Digital Input Operations

In a single-value digital input operation, a single data value is read from a single digital input port. The operation occurs immediately.

To determine if the subsystem supports single-value operations, use the **DigitalInputSubsystem.SupportsSingleValue** property. If this property returns a value of True, the subsystem supports single-value operations.

Once you have an DigitalInputSubsystem object, as described on page 146, set up the DigitalInputSubsystem object for a single value operation as follows:

1. Set the **DigitalInputSubsystem.DataFlow** property to SingleValue.
2. (Optional) Set the resolution of the subsystem using the **DigitalInputSubsystem.Resolution** property. Refer to page 231 for more information on resolution.
3. Configure the subsystem using the **DigitalInputSubsystem.Config** method.

Then, to acquire a single value from the digital input port, use the **DigitalInputSubsystem.GetSingleValue** method. You specify the digital input port to read and a gain of 1.

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

Note: If your device supports it, you can read a digital input port as part of the analog input stream. Ensure that you set up and configure the digital input subsystem, as described in this section, before starting the analog input operation. Refer to page 204 for more information on specifying a ChannelList object for an analog input subsystem.

Refer to the example program ReadSingleValue to see how to perform a single-value digital input operation.

Single-Value Digital Output Operations

In a single-value digital output operation, a single data value is output from a single digital output port. The operation occurs immediately.

To determine if the subsystem supports single-value operations, use the **DigitalOutputSubsystem.SupportsSingleValue** property. If this property returns a value of True, the subsystem supports single-value operations.

Once you have an DigitalInputSubsystem object, as described on page 146, set up the DigitalInputSubsystem object for a single value operation as follows:

1. Set the **DigitalOutputSubsystem.DataFlow** property to SingleValue.
2. (Optional) Set the resolution of the subsystem using the **DigitalOutputSubsystem.Resolution** property. Refer to page 231 for more information on resolution.
3. Configure the subsystem using the **DigitalOutputSubsystem.Config** method.

Then, to output a single value from the digital output port, use the **DigitalOutputSubsystem.SetSingleValue** method. You specify the digital output port to update and a gain of 1.

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

Note: If your device supports it, you can update a digital output port as part of the analog output stream. Ensure that you set up and configure the digital output subsystem, as described in this section, before starting the analog output operation. Refer to page 204 for more information on specifying a ChannelList object for an analog output subsystem.

Refer to the example program WriteSingleValue to see how to perform a single-value digital output operation.

Continuous, Interrupt-On-Change Operations

Use continuous digital input operation when you want to continuously monitor the state of the digital input lines, generating an interrupt when a digital input line changes state.

To determine if the digital input subsystem supports continuous operations, use the **DigitalInputSubsytem.SupportsContinuous** property. If this property returns a value of True, continuous mode is supported.

To determine if the digital input subsystem supports interrupt-on-change operations, use the **DigitalInputSubsytem.SupportsInterruptOnChange** property. If this property returns a value of True, interrupt-on-change operations are supported.

Once you have an DigitalInputSubsystem object, as described on page 146, set up the DigitalInputSubsystem object for a continuous operation as follows:

1. Set the **DigitalInputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the resolution of the subsystem using the **DigitalInputSubsystem.Resolution** property. Refer to page 231 for more information on resolution.
3. Select the digital input lines that you want to monitor for change of state using the **WriteInterruptOnChangeMask** method. (Note that you can read the current mask setting by using the **DigitalInputSubsystem.ReadInterruptOnChangeMask** method.)
4. Configure the subsystem using the **DigitalInputSubsystem.Config** method.

Once the subsystem is configured, call the **DigitalInputSubsystem.Start** method to start the interrupt-on-change operation.

An InterruptOnChangeEvent event is raised whenever one of the selected digital input lines (specified by the **WriteInterruptOnChangeMask** method) changes state. Use the InterruptOnChangeHandler, described on page 268, to deal with InterruptOnChangeEvent events.

To stop an continuous digital input operation, use one of the following methods:

- **DigitalInputSubsystem.Stop** – Stops the digital input operation.
- **DigitalInputSubsystem.Abort** – For this subsystem type, behaves like **Stop**.
- **Reset** – Stops the operation immediately, and reinitializes the subsystem to the default configuration.

Refer to the example program InterruptOnChange to see how to perform a continuous, interrupt-on-change operation on a digital input port.

Setting the Resolution

To determine if the subsystem supports software-programmable resolution, use the **SupportsSoftwareResolution** property of the appropriate subsystem. If this property returns a value of True, the subsystem supports software-programmable resolution.

To determine the number of resolution settings supported by the subsystem, use the **NumberOfResolutions** property of the appropriate subsystem. To list all the resolution settings supported by the subsystem, use the **SupportedResolutions** property of the appropriate subsystem.

Use the **Resolution** property of the appropriate subsystem to set and/or return the number of bits of resolution for the subsystem. Typically, you can set this property for digital I/O operations only.

Performing Counter/Timer Operations

The counter/timer subsystem supports general-purpose user counter/timers and measure counters. This section describes the operation of general-purpose counter/timers. Refer to page 249 for information on measure counters.

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 5.

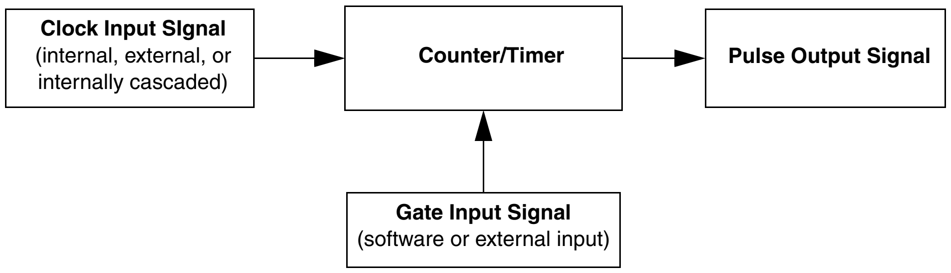

Clock Input SIgnal
(internal, external, or internally cascaded)
Counter/Timer
Pulse Output Signal
Gate Input Signal
(software or external input)

Figure 5: 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.

Using the DT-Open Layers for .NET Class Library, you can perform the following types of counter/timer operations.

- Event counting, described on page 232
- Up/down counting, described on page 234
- Edge-to-edge measurement, described on page 235
- Continuous edge-to-edge measurement, described on page 237
- Rate generation (continuous pulse output), described on page 239
- One-shot, described on page 241
- Repetitive one-shot, described on page 242

The following subsections describe these counter/timer operations in more detail.

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 **CounterTimerSubsystem.SupportsCount** property. If this property returns a value of True, event counting mode is supported.

Once you have a CounterTimerSubsystem object, as described on page 146, set up the CounterTimerSubsystem object for a event counting operation as follows:

1. Set the **CounterTimerSubsystem.DataFlow** property to Continuous.
2. Set the **CounterTimerSubsystem.CounterMode** property to a value of Count.
3. 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 244 for more information on specifying a clock source.
4. (Optional) Set the cascade mode of the counter/timer subsystem to either Cascade for cascaded counter/timers or Single for non-cascaded counter/timers using the **CounterTimerSubsystem.CascadeMode**. Refer to page 245 for more information.
5. Specify the gate type that enables the operation; refer to page 245 for more information on specifying the gate type.
6. Configure the subsystem using the **CounterTimerSubsystem.Config** method.

Start an event counting operation using the **CounterTimerSubsystem.Start** method. To read the current number of events counted, use the **CounterTimerSubsystem.ReadCount** method.

To stop an event counting operation, call the **CounterTimerSubsystem.Stop** or **CounterTimerSubsystem.Abort** method. For this subsystem type, **Stop** and **Abort** behave identically.

Figure 6 shows an example of an event counting operation. In this example the gate type is low level.

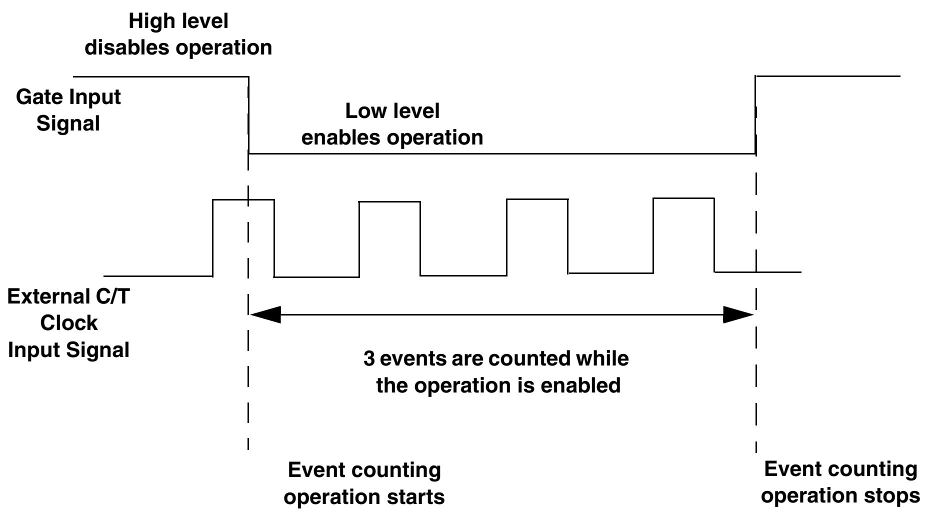


Figure 6: Example of Event Counting

Refer to the example program EventCounting to see how to perform an event counting operation.

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 **CounterTimerSubsystem.SupportsUpDown** property. If this property returns a value of True, up/down counting mode is supported.

Once you have a CounterTimerSubsystem object, as described on page 146, set up the CounterTimerSubsystem object for an up/down counting operation as follows:

1. Set the **CounterTimerSubsystem.DataFlow** property to Continuous.
2. Set the **CounterTimerSubsystem.CounterMode** property to a value of UpDown.
3. Specify the C/T clock source for the operation as External; see page 244 for more information on C/T clock sources.

Note: For up/down counting operations, you do not specify the gate type in software.

4. Start the up/down counting operation using the **CounterTimerSubsystem.Start** method.
5. Read the number of rising edges counted using the **CounterTimerSubsystem.ReadCount** method.

To stop an up/down counting operation, call the **CounterTimerSubsystem.Stop** or **CounterTimerSubsystem.Abort** method. For this subsystem type, **Stop** and **Abort** behave identically.

Figure 7 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.

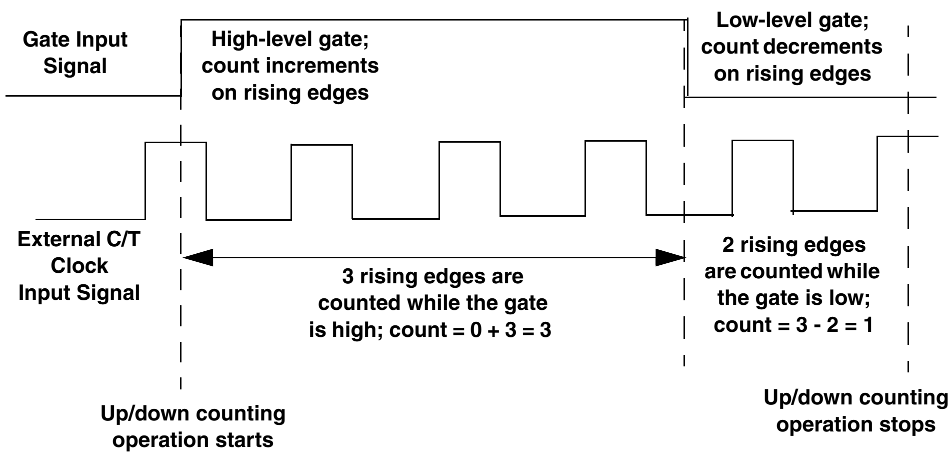

Gate Input Signal
High-level gate; count increments on rising edges
Low-level gate; count decrements on rising edges
External C/T Clock Input Signal
3 rising edges are counted while the gate is high; count = 0 + 3 = 3
2 rising edges are counted while the gate is low; count = 3 - 2 = 1
Up/down counting operation starts
Up/down counting operation stops

Figure 7: Example of Up/Down Counting

Refer to the example program EventCounting to see how to perform an up/down counting operation.

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.

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/Internal CT Clock Freq
- 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/Internal CT Clock Freq
- Frequency of a signal pulse (the number of periods per second). You can calculate the frequency as follows:
  - Frequency = Internal CT Clock Freq/Number of Counts

To determine if the subsystem supports edge-to-edge measurements, use the **CounterTimerSubsystem.SupportsMeasure** property. If this property returns a value of True, edge-to-edge measurement mode is supported.

To determine which edges can be selected in an edge-to-edge measurement operation, use the following properties:

- **CounterTimerSubsystem.SupportsGateRising** – Returns a value of True if the rising edge of the gate signal can be used in an edge-to-edge measurement operation.
- **CounterTimerSubsystem.SupportsGateFalling** – Returns a value of True if the falling edge of the gate signal can be used in an edge-to-edge measurement operation.
- **CounterTimerSubsystem.SupportsClockRising** – Returns a value of True if the rising edge of the clock signal can be used in an edge-to-edge measurement operation.
- **CounterTimerSubsystem.SupportsClockFalling** – Returns a value of True if the falling edge of the clock signal can be used in an edge-to-edge measurement operation.

You can also use the **CounterTimerSubsystem.SupportedEdgeTypes** property. This property returns an array of supported edge types.

Once you have a CounterTimerSubsystem object, as described on page 146, set up the CounterTimerSubsystem object for an edge-to-edge measurement operation as follows:

1. Set the **CounterTimerSubsystem.DataFlow** property to Continuous.
2. Set the **CounterTimerSubsystem.CounterMode** property to a value of Measure.
3. Specify the C/T clock source for the operation as internal; refer to page 244 for more information on this clock source.
4. Specify the start edge with the **CounterTimerSubsystem.StartEdge** property
5. Specify the stop edge with the **CounterTimerSubsystem.StopEdge** property.
6. Configure the counter/timer subsystem using the **CounterTimerSubsystem.Config** method.
7. Start the edge-to-edge measurement operation using the **CounterTimerSubsystem.Start** method.
8. Use the **MeasureDoneHandler** delegate to receive the MeasureDoneEventArgs argument and to handle the event MeasureDoneEvent. The MeasureDoneEventArgs class contains the subsystem that raised the event, the time stamp of when the event occurred, and the value of the counter.

To stop an edge-to-edge measurement operation, call the **CounterTimerSubsystem.Stop** or **CounterTimerSubsystem.Abort** method. For this subsystem type, **Stop** and **Abort** behave identically.

Figure 8 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.

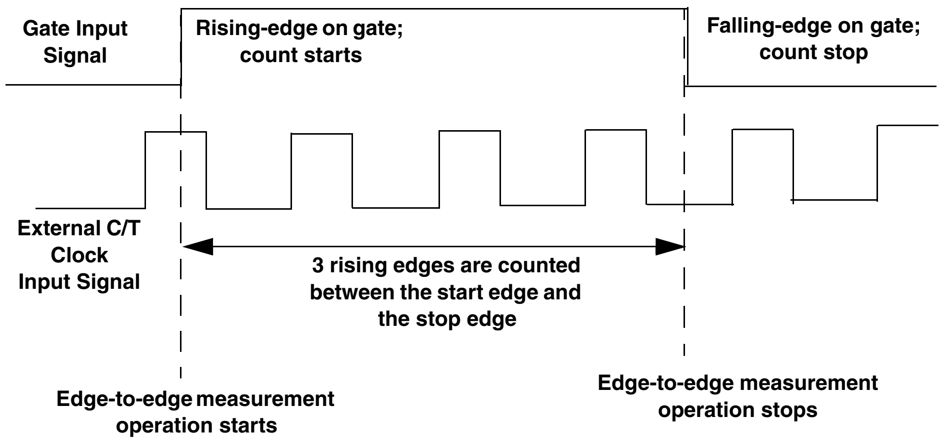

Gate Input Signal
Rising-edge on gate; count starts
Falling-edge on gate; count stop
External C/T Clock Input Signal
3 rising edges are counted between the start edge and the stop edge
Edge-to-edge measurement operation starts
Edge-to-edge measurement operation stops

Figure 8: Example of Edge-to-Edge Measurement

Refer to the example program MeasureEdgeToEdge to see how to perform an edge-to-edge measurement operation.

Continuous Edge-to-Edge Measurement

In continuous edge-to-edge measurement mode, the counter automatically performs an edge-to-edge measurement operation, where the counter starts incrementing when it detects the specified start edge and stops incrementing when it detects the specified stop edge. When the operation completes, the counter remains idle until it is next read. On the next read, the current value of the counter (from the previous edge-to-edge measurement operation) is returned and the next edge-to-edge measurement operation is started automatically.

Note: If you read the counter before the measurement is complete, 0 is returned.

You can use a continuous 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/Internal C/T Clock Freq
- 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/Internal C/T Clock Freq
- Frequency of a signal pulse (the number of periods per second). You can calculate the frequency as follows:
  - Frequency = Internal C/T Clock Freq/Number of Counts

To determine if the subsystem supports continuous edge-to-edge measurements, use the **CounterTimerSubsystem.SupportsContinuousMeasure** property. If this property returns a value of True, continuous edge-to-edge measurement mode is supported.

To determine which edges can be selected in a continuous edge-to-edge measurement operation, use the following properties:

- **CounterTimerSubsystem.SupportsGateRising** – Returns a value of True if the rising edge of the gate signal can be used in a continuous edge-to-edge measurement operation.
- **CounterTimerSubsystem.SupportsGateFalling** – Returns a value of True if the falling edge of the gate signal can be used in a continuous edge-to-edge measurement operation.
- **CounterTimerSubsystem.SupportsClockRising** – Returns a value of True if the rising edge of the clock signal can be used in a continuous edge-to-edge measurement operation.
- **CounterTimerSubsystem.SupportsClockFalling** – Returns a value of True if the falling edge of the clock signal can be used in a continuous edge-to-edge measurement operation.

You can also use the **CounterTimerSubsystem.SupportedEdgeTypes** property. This property returns an array of supported edge types.

Once you have a CounterTimerSubsystem object, as described on page 146, set up the CounterTimerSubsystem object for a continuous edge-to-edge measurement operation as follows:

1. Set the **CounterTimerSubsystem.DataFlow** property to Continuous.
2. Set the **CounterTimerSubsystem.CounterMode** property to a value of ContinuousMeasure.
3. Specify the C/T clock source for the operation as Internal; refer to page 244 for more information.
4. Specify the start edge with the **CounterTimerSubsystem.StartEdge** property.
5. Specify the stop edge with the **CounterTimerSubsystem.StopEdge** property.
6. Configure the counter/timer subsystem using the **CounterTimerSubsystem.Config** method.
   The continuous edge-to-edge measurement operation starts immediately.
7. Read the current value of the counter using the **CounterTimerSubsystem.ReadCount** method.

If your device allows you to stream counter/timer data through the analog input subsystem, you can also add the counter/timer channel to the channel list for an analog input operation. If you read the value of the counter/timer as part of the analog input data stream, you might see results similar to the following:

Table 71: An Example of Performing a Continuous Edge-to-Edge Measurement Operation as Part of the Analog Input ChannelList

Time	A/D Value	Counter/ Timer Value	Status of Continuous Edge-to-Edge Measurement Mode
10	5002	0	Operation started when the C/T subsystem was configured, but is not complete
20	5004	0	Operation not complete
30	5003	0	Operation not complete
40	5002	12373	Operation complete
50	5000	0	Next operation started, but is not complete
60	5002	0	Operation not complete
70	5004	0	Operation not complete
80	5003	12403	Operation complete
90	5002	0	Next operation started, but is not complete

To stop an edge-to-edge measurement operation, call the **CounterTimerSubsystem.Stop** or **CounterTimerSubsystem.Abort** method. For this subsystem type, **Stop** and **Abort** behave identically.

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 mode, use the **CounterTimerSubsystem.SupportsRateGenerate** property. If this property returns a value of True, rate generation mode is supported.

Once you have a CounterTimerSubsystem object, as described on page 146, set up the CounterTimerSubsystem object for a rate generation operation as follows:

1. Set the **CounterTimerSubsystem.DataFlow** property to Continuous.
2. Set the **CounterTimerSubsystem.CounterMode** property to a value of RateGenerator.
3. 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 244 for information on specifying the C/T clock source.
4. Specify the frequency of the C/T clock output signal. For an internal C/T source, setting the **Clock.Frequency** property determines the frequency of the output pulse.

   For an external C/T clock source, setting the external clock divider using the **Clock.ExtClockDivider** property determines the frequency of the output pulse. The frequency of the clock input source divided by the clock divider determines the frequency of the output pulse. Refer to page 244 for information on specifying the C/T clocks.
5. Specify the gate type that enables the operation; refer to page 245 for more information on specifying the gate type.
6. 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 247 for more information.
7. Configure the counter/timer subsystem using the **CounterTimerSubsystem.Config** method.
8. Start rate generation mode using the **CounterTimerSubsystem.Start** method. The counter outputs a pulse of the specified type and frequency continuously as long as the gate enables the operation. As soon as the gate signal disables the operation, the pulse output operation stops.

To stop rate generation while it is in progress, call the **CounterTimerSubsystem.Stop** or **CounterTimerSubsystem.Abort** method. For this subsystem type, **Stop** and **Abort** behave identically.

Figure 9 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.

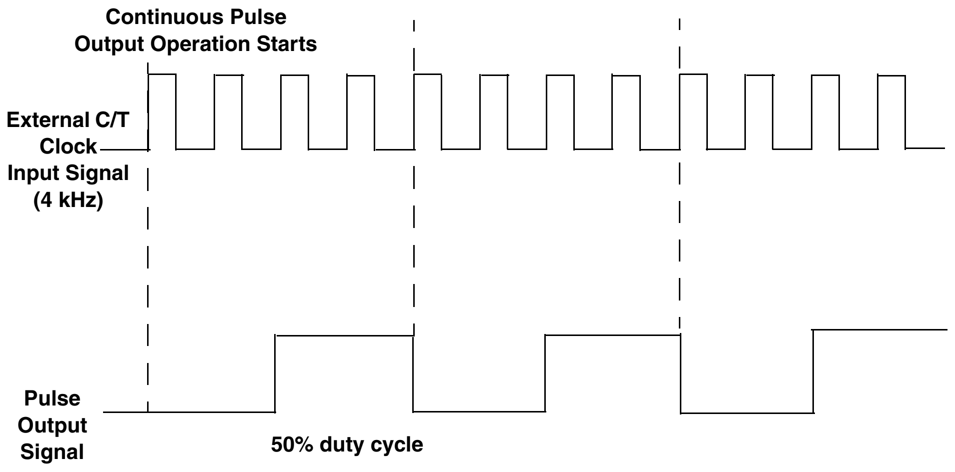


Figure 9: Example of Rate Generation Mode with a 50% Duty Cycle

Figure 10 shows the same example using a duty cycle of 75%.

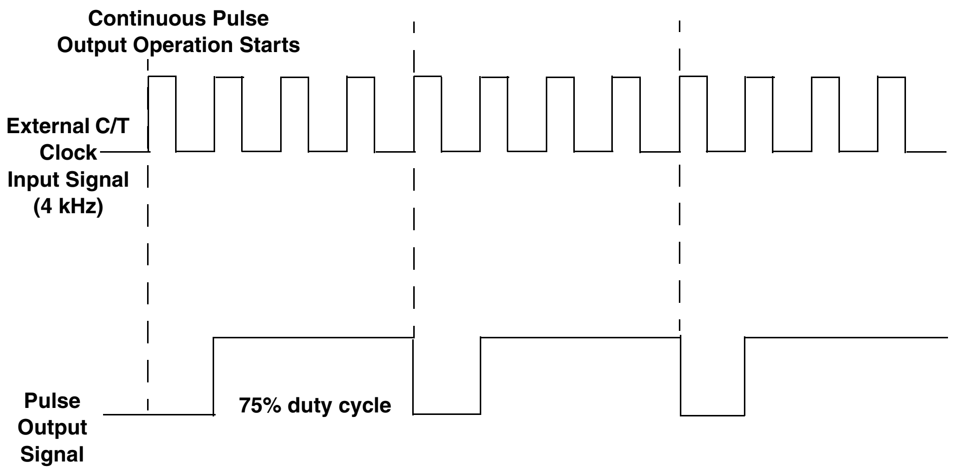


Figure 10: Example of Rate Generation Mode with a 75% Duty Cycle

Figure 11 shows the same example using a duty cycle of 25%.

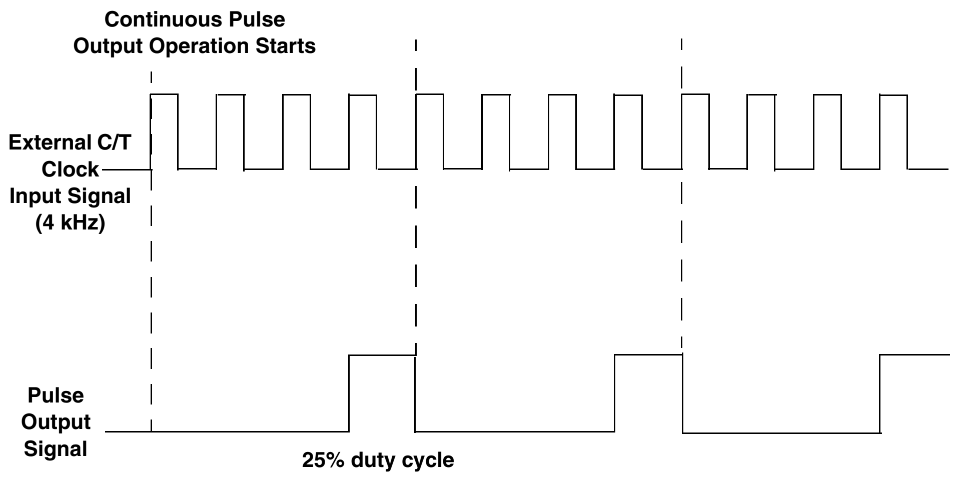

Continuous Pulse Output Operation Starts
External C/T Clock Input Signal (4 kHz)
Pulse Output Signal
25% duty cycle

Figure 11: Example of Rate Generation Mode with a 25% Duty Cycle

Refer to the example program PulseOut_RateGeneration to see how to perform a rate generation operation.

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 **CounterTimerSubsystem.SupportsOneShot** property. If this property returns a value of True, one-shot mode is supported.

Once you have a CounterTimerSubsystem object, as described on page 146, set up the CounterTimerSubsystem object for a one-shot operation as follows:

1. Set the **CounterTimerSubsystem.DataFlow** property to Continuous.
2. Set the **CounterTimerSubsystem.CounterMode** property to a value of OneShot.
3. Specify the C/T clock source for the operation. Refer to page 244 for more information on specifying the C/T clock source.
4. Specify the gate type that triggers the operation; refer to page 245 for more information.
5. Specify the polarity of the output pulse (high-to-low transition or low-to-high transition); refer to page 247 for more information.

Note: In the case of a one-shot operation, the pulse width is automatically set to 100%. The value of the **PulseWidth** property is ignored.

6. Configure the counter/timer subsystem using the **CounterTimerSubsystem.Config** method.
7. Start the one-shot pulse output operation using the **CounterTimerSubsystem.Start** method. 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.

Figure 12 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), and a low-to-high pulse type.

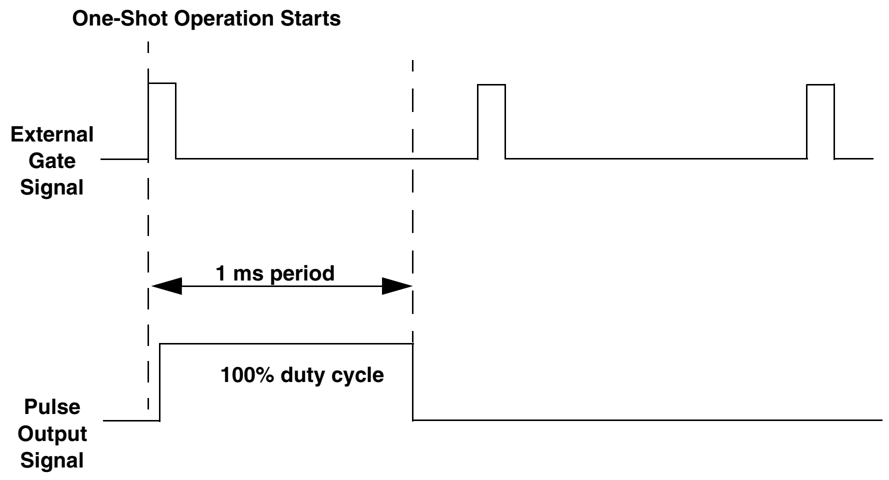


Figure 12: Example of One-Shot Mode

Refer to the example program PulseOut_RateGeneration to see how to perform a one-shot operation.

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 **CounterTimerSubsystem.SupportsOneShotRepeat** property. If this property returns a value of True, repetitive one-shot mode is supported.

Once you have a CounterTimerSubsystem object, as described on page 146, set up the CounterTimerSubsystem object for a repetitive one-shot operation as follows:

1. Set the **CounterTimerSubsystem.DataFlow** property to Continuous.
2. Set the **CounterTimerSubsystem.CounterMode** property to a value of OneShotRepeat.

3. Specify the C/T clock source for the operation. In repetitive one-shot mode, the internal C/T clock source is more useful than the external C/T clock source. Refer to page 244 for more information on specifying the C/T clock source.
4. Specify the polarity of the output pulses (high-to-low transitions or low-to-high transitions). Refer to page 247 for more information.

Note: In the case of a repetitive one-shot operation, the pulse width is automatically set to 100%. The value of the **PulseWidth** property is ignored.

5. Specify the gate type that triggers the operation. Refer to page 245 for more information.
6. Configure the counter/timer subsystem using the **CounterTimerSubsystem.Config** method.
7. Start a repetitive one-shot pulse output operation using the **CounterTimerSubsystem.Start** method. 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.

To stop a repetitive one-shot operation, call the **CounterTimerSubsystem.Stop** or **CounterTimerSubsystem.Abort** method. For this subsystem type, **Stop** and **Abort** behave identically.

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

Figure 13 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), and a low-to-high pulse type.

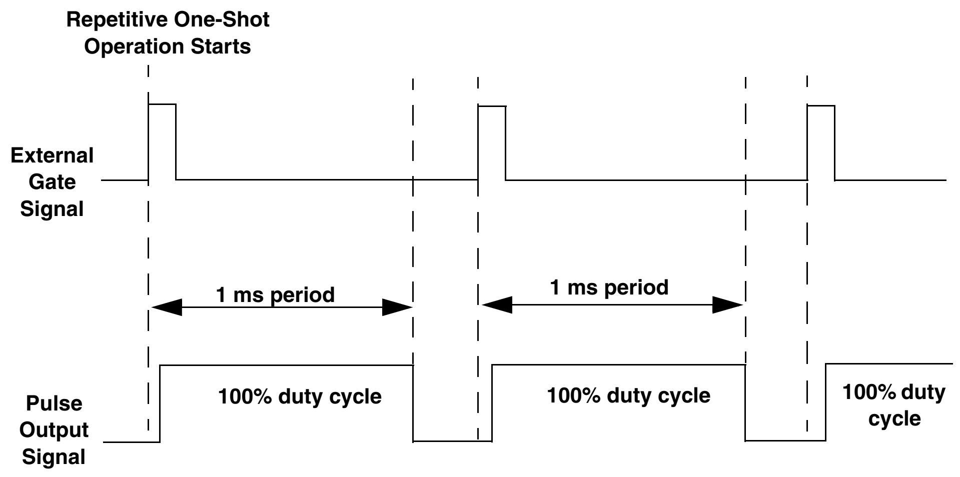


Figure 13: Example of Repetitive One-Shot Mode Using a 99.99% Duty Cycle

Refer to the example program PulseOut_RateGeneration to see how to perform a repetitive one-shot operation.

Setting the C/T Clock

The DT-Open Layers for .NET Class Library defines the following clock sources for counter/timers:

- Internal C/T clock
- External C/T clock
- Internally cascaded clock

The following subsections describe these clock sources.

Using an 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 **Clock.SupportsInternalClock** property. If this property returns a value of True, an internal C/T clock is supported.

To specify the clock source, use the **Clock.Source** property.

Using the **Clock.Frequency** property, specify the frequency of the clock output signal.

To determine the maximum frequency that the subsystem supports, use the **Clock.MaxFrequency** property. To determine the minimum frequency that the subsystem supports, use the **Clock.MinFrequency** property.

Using and External C/T Clock

The external C/T clock is a clock source attached to the counter/timer subsystem that paces counter/timer operations. 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 clock, use the **Clock.SupportsExternalClock** property. If this property returns a value of True, an external clock is supported.

Specify the clock source as internal using the **Clock.Source** property. Then, use the **Clock.ExtClockDivider** property to set or get 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 external clock divider that the subsystem supports, use the **Clock.MaxExtClockDivider** property. To determine the minimum external clock divider that the subsystem supports, use the **Clock.MinExtClockDivider** property.

Using an 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, for example.

To determine if the subsystem supports internal cascading, use the **CounterTimerSubsystem.SupportsCascading** property. If this function returns a value of True, internal cascading is supported.

Set the cascade mode of the subsystem to Cascade or Single (not cascaded) using the **CounterTimerSubsystem.CascadeMode** property.

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.

Setting the Gate Type

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

- None (software)
- HighLevel
- LowLevel
- HighEdge
- LowEdge
- Level

To specify the gate type, use the **CounterTimerSubsystem.GateType** property. The following subsections describe these gate types.

Using a None (Software) Gate Type

A gate type of None (software) enables the counter/timer operation immediately when the **CounterTimerSubsystem.GateType** property is set.

To determine if the subsystem supports a software gate, use the **CounterTimerSubsystem.SupportsGateNone** property. If this property returns a value of True, a gate type of None is supported.

Using a HighLevel Gate Type

A HighLevel 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 the following operations: event counting (see page 232) and rate generation (see page 239).

To determine if the subsystem supports a HighLevel external gate input, use the **CounterTimerSubsystem.SupportsGateHighLevel** property. If this property returns a value of true, a HighLevel gate type is supported.

Using a LowLevel Gate Type

A LowLevel 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 the following operations: event counting (see page 232) and rate generation (see page 239).

To determine if the subsystem supports a LowLevel external gate input, use the **CounterTimerSubsystem.SupportsGateLowLevel** property. If this property returns a value of true, a LowLevel gate type is supported.

Using LowEdge Gate Type

A LowEdge 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 242 for more information on these modes.

To determine if the subsystem supports a LowEdge external gate input, use the **CounterTimerSubsystem.SupportsGateLowEdge** property. If this property returns a value of true, a LowEdge gate type is supported.

Using a HighEdge Gate Type

A HighEdge 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 (see page 241) and repetitive one-shot (see page 242) operations.

To determine if the subsystem supports a HighEdge external gate input, use the **CounterTimerSubsystem.SupportsGateHighEdge** property. If this property returns a value of true, a HighEdge gate type is supported.

Using a Level Gate Type

A Level gate type enables a counter/timer operation on the transition from any level on the gate input signal. Note that this gate type is used only for the following operations: event counting (see page 232) and rate generation (see page 239).

To determine if the subsystem supports a Level external gate input, use the **CounterTimerSubsystem.SupportsGateLevel** property. If this property returns a value of true, a Level gate type is supported.

Setting the Pulse Output Type and Pulse Width

The CounterTimerSubsystem class 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 **CounterTimerSubsystem.SupportsHighToLowPulse** property. If this property returns a value of True, high-to-low transitions are 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 **CounterTimerSubsystem.SupportsLowToHighPulse** property. If this property returns a value of True, low-to-high transitions are supported.

Specify the pulse output type using the **CounterTimerSubsystem.PulseType** property.

The pulse width (or duty cycle) 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.

You can determine whether the pulse width is programmable by using the **CounterTimerSubsystem.SupportsVariablePulseWidth** property. If this property returns a value of True, the pulse width is programmable.

Specify the pulse width, in percentage, using the **CounterTimerSubsystem.PulseWidth** property. The default value is 50%.

Note: In the case of a one-shot or repetitive one-shot operation, the pulse width is automatically set to 100%. The value of the **PulseWidth** property is ignored.

Figure 14 illustrates a low-to-high pulse with a duty cycle of approximately 30%.

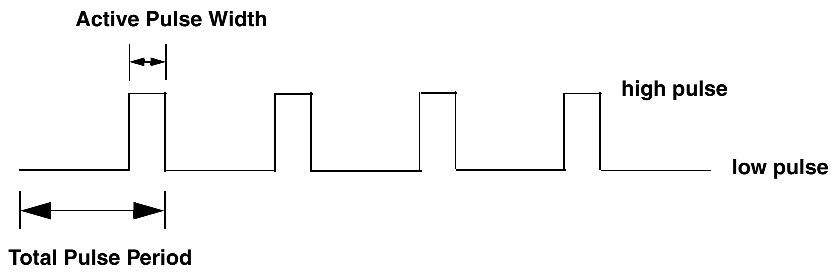


Figure 14: Example of a Low-to-High Pulse Output Type

Performing Measure Counter Operations

The counter/timer subsystem supports general-purpose user counter/timers and measure counters. This section describes the operation of measure counters. Refer to page 232 for information on the operation of general-purpose counter/timers.

If your device supports measure counters, set up the CounterTimerSubsystem object for a measure operation as follows:

1. Select the signal/edge that is used to start the measure operation using the **CounterTimerSubsystem.StartEdge** property.
2. Select the signal/edge that is used to stop the measure operation using the **CounterTimerSubsystem.StopEdge** property.

   The internal counter starts incrementing when it detects the selected start edge of the specified signal and stops incrementing when it detects the selected stop edge of the specified signal.

   To determine which signals and edges are supported for the measure counter, use the **CounterTimerSubsystem.SupportedEdgeTypes** property. This property returns an array of the supported signals/edges.
3. Configure the subsystem using the **CounterTimerSubsystem.Config** method.
4. Read the value of the measure counter in the analog input data stream by specifying the measure counter in the analog input channel list and reading the values corresponding to that channel from the analog input buffer. Refer to the documentation for your device to determine which channel to specify for the measure counter in the analog input channel list.

When you read the value of the measure counter as part of the analog input data stream, you might see results similar to the following:

Table 72: An Example of Reading the Measure Counter as Part of the Analog Input Data Stream

Time	A/D Value	Measure Counter Value	Status of Operation
10	5002	0	Operation started, but is not complete
20	5004	0	Operation not complete
30	5003	0	Operation not complete
40	5002	12373	Operation complete
50	5000	12373	Next operation started, but is not complete
60	5002	12373	Operation not complete
70	5004	12373	Operation not complete
80	5003	14503	Operation complete
90	5002	14503	Next operation started, but is not complete

Using the count that is returned from the measure counter, you can determine the following:

- Frequency of a signal pulse (the number of periods per second). You can calculate the frequency as follows:
  - Frequency = Frequency of the internal counter/(Number of counts – 1)

    For example, if the frequency of the internal counter on the device is 48 MHz and the count is 201, the measured frequency is 240 kHz (48 MHz/200).
- Period of a signal pulse. You can calculate the period as follows:
  - Period = 1/Frequency
  - Period = (Number of counts – 1)/Frequency of the internal counter

Performing Tachometer Operations

Some devices allow you to connect a tachometer signal to the device to measure the frequency or period of the tachometer input signal.

Once you have a TachSubsystem object, as described on page 146, set up the TachSubsystem object for a tachometer operation as follows:

1. Set the edge of the tachometer signal that is used for the measurement using the **TachSubsystem.EdgeType** property.

   In a tachometer operation, the internal counter starts incrementing when it detects the first specified edge of the tachometer input and stops incrementing when it detects the next specified edge of the tachometer input.

   To determine if falling edges of the tachometer signal are supported, use the **TachSubsystem.SupportsFallingEdge** property. To determine if rising edges of the tachometer signal are supported, use the **TachSubsystem.SupportsRisingEdge** property.

2. Specify the value of the stale data flag using the **TachSubsystem.StaleDataFlagEnabled** property. If this flag is True, the most significant bit (MSB) of the value is set to 0 to indicate new data; reading the value before the measurement is complete returns an MSB of 1. If this flag is False, the MSB is always set to 0.

   To determine if the stale data flag is supported, use the **TachSubsystem.SupportsStaleDataFlag** property.

3. Configure the subsystem using the **TachSubsystem.Config** method.

4. Read the tachometer measurement from the analog input data stream using the **TachSubsystem.Count** property or by specifying the tachometer channel in the analog input channel list and reading the values corresponding to that channel from the analog input buffer. Refer to the documentation for your device to determine which channel to specify for the tachometer in the analog input channel list.

5. If desired, convert the returned count to an RPM value by using the **OlBuffer.GetDataAsRpm** method.

When you read the value of the tachometer input as part of the analog input data stream, you might see results similar to the following:

Table 73: An Example of Reading the Tachometer Input as Part of the Analog Input Data Stream

Time	A/D Value	Tachometer Input Value	Status of Operation
10	5002	0	Operation started, but is not complete
20	5004	0	Operation not complete
30	5003	0	Operation not complete
40	5002	12373	Operation complete
50	5000	12373	Next operation started, but is not complete
60	5002	12373	Operation not complete

Table 73: An Example of Reading the Tachometer Input as Part of the Analog Input Data Stream (cont.)

Time	A/D Value	Tachometer Input Value	Status of Operation
70	5004	12373	Operation not complete
80	5003	14503	Operation complete
90	5002	14503	Next operation started, but is not complete

Using the count that is returned from the tachometer input, you can determine the following:

- Frequency of a signal pulse (the number of periods per second). You can calculate the frequency as follows:
  - Frequency = Frequency of the internal counter/(Number of counts – 1)

    For example, if the frequency of the internal counter on the device is 12 MHz and the count is 21, the measured frequency is 600 kHz (12 MHz/20).
- Period of a signal pulse. You can calculate the period as follows:
  - Period = 1/Frequency
  - Period = (Number of counts – 1)/Frequency of the internal counter

Performing Quadrature Decoder Operations

Some devices support quadrature decoder operations. A quadrature decoder accepts signals (A, B, and Index) from a quadrature encoder as inputs and converts these signals into a clock output signal whose pulses are counted by the decoder.

To determine whether your subsystem supports quadrature decoder functionality, use the **QuadratureDecoder.SupportsQuadratureDecoder** property. If this property returns a value of True, your subsystem supports quadrature decoder functionality.

Once you have a QuadratureDecoderSubsystem object, as described on page 146, set up the QuadratureDecoderSubsystem object for a quadrature decoder operation as follows:

1. Set the **QuadratureDecoderSubsystem.DataFlow** property to Continuous.
2. Set the clock source to External with **QuadratureDecoder.Clock.Source**. Refer to page 254 for more information.
3. Set the pre-scale value used to divide the base clock frequency using the **QuadratureDecoderSubsystem.ClockPreScale** property. Refer to page page 254 or more information.
4. Set the value of the **QuadratureDecoderSubsystem.X4Scaling** property to True if you want to use X4 mode, or False if you want to use X1 mode. Refer to page 254 for more information.
5. Set the value of the **QuadratureDecoderSubsystem.IndexMode** property to Disabled if you do not want to use the Index input signal, Low if the Index input signal is low, or High if the Index input signal is high. Refer to page 254 for more information.
6. Configure the subsystem using the **QuadratureDecoderSubsystem.Config** method.
7. Start the operation using the **QuadratureDecoderSubsystem.Start** method.
8. Read the current value of the quadrature decoder subsystem using the **QuadratureDecoderSubsystem.ReadCount** method.

The value of the quadrature decoder determines the relative or absolute position and/or rotational speed. For example, in an X/Y positioning application, you could use one quadrature decoder to determine the position on the X-plane, and another quadrature decoder to determine the position on the Y-plane.

To determine the rotation of a quadrature encoder, use the following formula:

$$\textit{Rotation degrees} = \frac{\text{Count}}{4 * \text{N}} \text{ x 360 degrees}$$

where *N* is the number of pulses generated by the quadrature encoder per rotation. For example, if every rotation of the quadrature encoder generated 10 pulses, and the value read from the quadrature decoder is 20, the rotation of the quadrature encoder is 180 degrees (20/40 x 360 degrees).

Refer to the example program ReadCounts to see how to perform a quadrature decoder operation.

Setting up the Clock

For quadrature decoder operations, the onboard base clock of the hardware device is used to sample the A and B inputs. From a software perspective, the software considers the A and B inputs as external clock sources, therefore, you must specify the clock source as External using the **Clock.Source** property.

You can filter the sampling frequency of the onboard base clock by using the **QuadratureDecoderSubsystem.ClockPreScale** property to divide down the base frequency. Values for this property range from 0 and 255, where 0 corresponds to an actual pre-scale value of 1, and 255 corresponds to an actual prescale value of 256. For example, if you are using a device with a base clock of 36 MHz and specify a pre-scale value of 0, the resulting sampling frequency is 36 MHz (36 MHz/1). Similarly, if you specify a pre-scale value of 255 when using a device with a 48 MHz base clock, the resulting sampling frequency is 18.75 kHz (48 MHz/256).

The filter samples the incoming A and B signals twice, and when it samples a change in the state of any of these signals and the change is present for two samples, the change on the inputs is valid. A minimum of 4 clock pulses is needed to sample each edge of the A and B inputs in one cycle (for a total of 16 clock pulses). In addition, the minimum time between one edge and the next edge is 112 ns. So, if you are using a 36 MHz sample frequency, the A and B inputs can have a maximum frequency of 2.23 MHz, or a period of 448 ns.

In general, if the **QuadratureDecoder.ClockPreScale** value is set too low, the system is more susceptible to noise on the inputs. If the **QuadratureDecoder.ClockPreScale** value is set too high, counts may be missed.

Setting the X4Scaling Mode

You can control whether the quadrature decoder operates in X1 or X4 mode (if it is supported by the encoder) by using the **QuadratureDecoderSubsystem.X4Scaling** property.

Setting the **QuadratureDecoderSubsystem.X4Scaling** property to False selects X1 mode; in X1 mode, the decoder generates one clock pulse for every complete cycle of the A and B inputs. Setting the **QuadratureDecoderSubsystem.X4Scaling** property to True selects X4 mode; in X4 mode, the decoder generates one clock pulse for each edge of the A and B signals – four edges for each cycle of the A and B inputs.

Setting the Index

Using the **QuadratureDecoderSubsystem.IndexMode** property, you can specify how the Index input signal affects the operation of the counter. If you set this property to Disabled, then the Index input signal has no effect. If you set this property to Low, then the quadrature decoder resets its value to 0 whenever it detects a low edge (falling edge) on the Index input signal. If you set this property to High, then the quadrature decoder resets its value to 0 whenever it detects the high edge (rising edge) on the Index input signal.

Note: You must set the **IndexMode** property to Disabled if you set the **X4Scaling** property to True.

Starting Subsystems Simultaneously

If supported, you can set up subsystems to start simultaneously. Note that you cannot perform simultaneous startup on subsystems configured for single-value operations unless you are using a simultaneous sampling module.

To determine if a subsystem supports simultaneous start, use the **SupportsSimultaneousStart** property inherited from the SubsystemBase class. If this property returns a value of True, the subsystem can be simultaneously started.

You can synchronize the triggers of subsystems by specifying the same trigger source for each of the subsystems that you want to start simultaneously; ensure that the triggers are wired appropriately to the device.

Use the **SimultaneousStart.AddSubsystem** method to add the subsystems that you want to start simultaneously to the start list. If, later, you want to remove a subsystem from the start list, use the **SimultaneousStart.RemoveSubsystem** method.

To return an array of subsystems that were added to the simultaneous start list, use the **SimultaneousStart.GetSubsystemList** method.

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

Start the subsystems using the **SimultaneousStart.Start** method. When started, both subsystems are triggered simultaneously.

When you are finished with the operations, call the **SimultaneousStart.Clear** method to remove the subsystems from the simultaneous start list.

Auto-Calibrating a Subsystem

Some devices provide a self-calibrating feature, where a specified subsystem performs an auto-zero function. To determine if the specified subsystem supports this capability, use the **AnalogInputSubsystem.SupportsAutoCalibrate** property. If this property returns a value of True, the subsystem can be calibrated through software.

To calibrate the subsystem in software, call the **AutoCalibrate** method. Ensure that the subsystem is not running when you call this method, or an error is returned.

Handling Events

DT-Open Layers devices notify your application of buffer movement and other system activities by raising events.

Delegates, which behave like function pointers, are provided to handle these events. Each delegate has a specific signature and holds a reference to a method that matches its signature. When an event occurs, the appropriate method (with the matching signature) is called.

The following example shows the declaration for the **BufferDoneHandler** delegate provided by the DT-Open Layers for .NET Class Library:

```
[C#]
// BufferDoneHandler is the delegate for the BufferDoneEvent event.
// BufferDoneEventArgs is the class that holds event data for
// BufferDoneEvent.
// It derives from the base class for event data, GeneralEventArgs.

public delegate void BufferDoneHandler(object sender,
   BufferDoneEventArgs eventArgs);

[Visual Basic]
' BufferDoneHandler is the delegate for the BufferDoneEvent event.
' BufferDoneEventArgs is the class that holds event data for
' BufferDoneEvent.
' It derives from the base class for event data, GeneralEventArgs.
Public Delegate Sub BufferDoneHandler(sender As Object,
   eventArgs As BufferDoneEventArgs)
```

As you can see, the syntax of the delegate is similar to that of a method declaration; however, the delegate keyword informs the compiler that **BufferDoneHandler** is a delegate type. By convention, event delegates in the .NET Framework have two parameters, the source that raised the event and the data for the event.

To handle events, you must define a method that matches the delegate; this is the event handling method that is called when the appropriate event is raised. In the following example, the event handling method called MyBufferDone matches the signature of the **BufferDoneHandler** delegate and is called when the event BufferDoneEvent is raised:

Visual C#

```
// MyBufferDone has the same signature as BufferDoneHandler.
public void MyBufferDone (object sender,
   BufferDoneEventArgs eventArgs);
{
//Add you own code here.
}
```

Visual Basic

```
' MyBufferDone has the same signature as BufferDoneHandler.
   Public Sub MyBufferDone(sender As Object,
      eventArgs As BufferDoneEventArgs)
' Add you own code here
   End Sub
```

Note: To ensure that events are handled in the main application, use the InvokeRequired method. Refer to your .NET documentation for more information on this method.

Lastly, you must associate the event and event handling method with the appropriate subsystem. The following example shows how to associate the event BufferDoneEvent and the MyBufferDoneHandler event handler to the analog input subsystem called *ainSS*:

Visual C#

```
// Associate the event BufferDoneEvent and the event handling method
// MyBufferDone with the analog input subsystem ainSS.
ainSS.BufferDoneEvent += new BufferDoneHandler (MyBufferDoneHandler);
```

Note: In C#, when you want to disable receiving events, use the - = operator instead of the += operator. See your .NET documentation for more information about events and delegates.

Visual Basic

```
' Associate the event BufferDoneEvent and the event handling method
' MyBufferDone with the analog input subsystem ainSS.
AddHandler ainSS.BufferDoneEvent, Address of MyBufferDoneHandler
```

Note: In Visual Basic, when you want to disable receiving events, use the RemoveHandler statement instead of the AddHandler statement. See your .NET documentation for more information about events and delegates.

The following subsections describe the events and delegates that are provided in the DT-Open Layers for .NET Class Library. Refer to the examples provided with this software package to see how to incorporate event handling code into your program.

BufferDoneEvent

For input operations, the event BufferDoneEvent is raised when the internal data buffer of the OlBuffer object has been filled with post-trigger data. For output operations, this event is raised when all the data in the internal data buffer of the OlBuffer object has been output.

If you stop an analog I/O operation, the event BufferDoneEvent is generated for the current OlBuffer object and for up to eight inprocess OlBuffer objects before a QueueStoppedEvent event occurs.

Use the **BufferDoneHandler** delegate with BufferDoneEvent. When BufferDoneEvent is raised, the subsystem that raised the event, the time stamp of when the event occurred, and the completed OlBuffer object are passed in the BufferDoneEventArgs argument of the user-defined method that matches the signature of the **BufferDoneHandler** delegate.

You can add your own code to the event handling method to manage the data in the buffer or perform other operations as required by your application. Refer to page 221 for more information on handling input buffers; refer to page 224 for more information on handling output buffers.

Note: If your program is running under a heavy CPU load, and if the **AnalogInputSubsystem.SynchronousBufferDone** property is set to False (the default condition), .NET may call your BufferDoneEvent delegates out of order under some circumstances. To avoid this problem, it is recommended that you set the **AnalogInputSubsystem.SynchronousBufferDone** property to True, so that all BufferDoneEvent events are executed synchronously in a single worker thread instead of asynchronously using a separate thread for each event.

The following is an example of an event handling routine called HandleBufferDone that handles the event BufferDoneEvent. This event handler converts the data from the internal buffer of the OlBuffer object into sensor values and copies the data into a user-dimensioned array called *buf.* The first 10 samples are printed to the form, and the OlBuffer object is put back on the queue for the subsystem:

Visual C#

```
public void HandleBufferDone (object sender,
   BufferDoneEventArgs bufferDoneData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke( new BufferDoneHandler (HandleBufferDone),
              new object[] {sender, bufferDoneData });
         }
         else
         {
            // Get the data as sensor values
            double[] buf = olBuffer.GetDataAsSensor();
              //requeue the completed buffer
               ainSS.BufferQueue.QueueBuffer (olBuffer);
            // Output the first 10 samples to the user form
            for (int i=0; i<10; ++i)
            {
               OlBufferDataTable.Rows[i][0] = buf[i];
            }
         }
      }
```

Visual Basic

```
Public Sub HandleBufferDone(ByVal sender As Object,
  ByVal bufferDoneData As BufferDoneEventArgs)
         If Me.InvokeRequired Then
            Me.Invoke(New BufferDoneHandler(
            AddressOf HandleBufferDone), New Object()
              {sender, bufferDoneData})
         Else
            ' Get the data as sensor values
            Dim buf As Double() = olBuffer.GetDataAsSensor()
            ' requeue the completed buffer
            ainSS.BufferQueue.QueueBuffer(olBuffer)
            End If
            ' Output the first 10 samples to the user form
            Dim i As Integer
            While i < 10
               OlBufferDataTable.Rows(i)(0) = buf(i)
               i += 1
            End While
         End If
End Sub 'HandleBufferDone
```

PreTriggerBufferDoneEvent

The event PreTriggerBufferDone is raised when the internal buffer of the OlBuffer object is filled with pre-trigger data (for an input operation only). Refer to page 218 for more information about buffers.

Use the **PreTriggerBufferDoneHandler** delegate with PreTriggerBufferDoneEvent. When PreTriggerBufferDoneEvent is raised, the subsystem that raised the event, the time stamp of when the event occurred, and the completed OlBuffer object are passed in the BufferDoneEventArgs argument of the user-defined method that matches the signature of the **PreTriggerBufferDoneHandler** delegate.

You can add your own code to the event handling method to manage the data in the buffer or perform other operations as required by your application. Refer to page 221 for more information on handling input buffers.

The following is an example of an event handling routine called HandlePreTriggerBufferDone that handles the event PreTriggerBufferDoneEvent. This event handler converts the data from the internal buffer of the OlBuffer object into voltage values and copies the data into a user-dimensioned array called *buf.* The first 10 samples are printed to the form, and the OlBuffer object is put back on the queue for the subsystem:

Visual C#
```
public void HandlePreTriggerBufferDone (object
   sender, BufferDoneEventArgs bufferDoneData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke( new PreTriggerBufferDoneHandler (
               HandlePreTriggerBufferDone), new object[] { sender,
                 bufferDoneData});
         }
         else
         {
            // Get the data as voltages
            double[] buf = olBuffer.GetDataAsVolts();

              //requeue the completed buffer
               ainSS.BufferQueue.QueueBuffer (olBuffer);

            // Output the first 10 samples to the user form
            for (int i=0; i<10; ++i)
            {
               OlBufferDataTable.Rows[i][0] = buf[i];
            }
         }
      }
```

Visual Basic
```
Public Sub HandlePreTriggerBufferDone(ByVal sender As Object,
  ByVal bufferDoneData As BufferDoneEventArgs)
         If Me.InvokeRequired Then
            Me.Invoke(New PreTriggerBufferDoneHandler(
             AddressOf HandlePreTriggerBufferDone),
             New Object() {sender, bufferDoneData})
         Else
            ' Get the data as voltages
            Dim buf As Double() = olBuffer.GetDataAsVolts()
            ' requeue the completed buffer
            ainSS.BufferQueue.QueueBuffer(olBuffer)
            End If
            ' Output the first 10 samples to the user form
            Dim i As Integer
            While i < 10
               OlBufferDataTable.Rows(i)(0) = buf(i)
               i += 1
            End While
         End If
End Sub 'HandleBufferDone
```

QueueStoppedEvent

A QueueStoppedEvent is raised when **Stop** or **Abort** is called for a continuous analog input or analog output operation.

Note: The event BufferDoneEvent is generated for the current OlBuffer object and for up to eight inprocess OlBuffer objects before a QueueStoppedEvent event occurs.

Use the **QueueStoppedHandler** delegate with QueueStoppedEvent. When QueueStoppedEvent is raised, the subsystem that raised the event and the time stamp of when the event occurred are passed in the GeneralEventArgs argument of the user-defined method that matches the signature of the **QueueStoppedHandler** delegate.

The following is an example of an event handling routine called HandleQueueStopped that handles the event QueueStoppedEvent. This event handler displays a message on the form that indicates which subsystem raised the QueueStoppedEvent and at what time the event occurred:

Visual C#
```
public void HandleQueueStopped (object sender,
   GeneralEventArgs eventData)
   {
      if (this.InvokeRequired)
         {
            this.Invoke(new QueueStoppedHandler(HandleQueueStopped)
               ,new object[] { sender, eventData });
         }
         else
         {
            string msg = String.Format ("Queue Stopped received on
               subsystem {0} element {1} at time {2}",
                eventData.Subsystem, eventData.Subsystem.Element,
                eventData.DateTime.ToString("T"));

            statusBarPanel.Text = msg;
         }
   }
```

Visual Basic
```
Public Sub HandleQueueStopped(ByVal sender As Object,
  ByVal eventData As GeneralEventArgs)
         If Me.InvokeRequired Then
            Me.Invoke(New QueueStoppedHandler(
               AddressOf HandleQueueStopped),
                New Object() {sender, eventData})
         Else
            Dim msg As String = String.Format(
              "Queue Stopped received on subsystem {0} element {1}
               at time {2}", eventData.Subsystem,
               eventData.Subsystem.Element,
               eventData.DateTime.ToString("T"))
             statusBarPanel.Text = msg
         End If
End Sub 'HandleQueueStopped
```

IOCompleteEvent

For analog input operations that use a reference trigger whose trigger type is something other than software (none), the event IOCompleteEvent is raised when the last post-trigger sample is copied into the user buffer. This event includes the total number of samples per channel that were acquired from the time acquisition was started (with the start trigger) to the last post-trigger sample. For example, a value of 100 indicates that a total of 100 samples (samples 0 to 99) were acquired. In some cases, this message is generated well before the events BufferDoneEvent are generated. You can determine when the reference trigger occurred and the number of pre-trigger samples that were acquired by subtracting the post trigger scan count, described on page 217, from the total number of samples that were acquired. Devices that do not support a reference trigger will never receive this event for analog input operations.

For analog output operations, the event IOCompleteEvent is raised when the last data point has been output from an analog output channel. In some cases, this event is raised well after the data is transferred from the buffer (and, therefore, well after BufferDoneEvent and QueueDoneEvents are raised). Refer to page 218 for more information on buffers.

Use the **IOCompleteHandler** delegate with IOCompleteEvent. When IOCompleteEvent is raised, the subsystem that raised the event and the time stamp of when the event occurred are passed in the IOCompleteEventsArgs argument of the user-defined method that matches the signature of the **IOCompleteHandler** delegate.

You can add your own code to the event handling method to deal with this event as needed.

The following is an example of an event handling routine called HandleIOComplete that handles the event IOCompleteEvent. This event handler displays a message on the form that indicates which subsystem raised the IOCompleteEvent and at what time the event occurred:

Visual C#

```
public void HandleIOComplete (object sender,
  IOCompleteEventArgs eventData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke( new IOCompleteHandler (HandleIOComplete),
              new object[] {sender, eventData });
         }


         else
         {
            string msg = String.Format ("IOComplete received on
             subsystem {0} at time {1}", eventData.Subsystem,
               eventData.DateTime.ToString("T"));
            statusBarPanel.Text = msg;
         }
      }
```

Visual Basic

```
Public Sub HandleIOComplete(ByVal sender As Object,
  ByVal eventData As IOCompleteEventArgs)
         If Me.InvokeRequired Then
           Me.Invoke(New IOCompleteHandler(
            AddressOf HandleIOComplete),
             New Object() {sender, eventData})
         Else
            Dim msg As String = String.Format(
              "IOComplete received on subsystem {0} at time {1}",
               eventData.Subsystem, eventData.DateTime.ToString("T"))
                statusBarPanel.Text = msg
         End If
End Sub 'HandleIOComplete
```

QueueDoneEvent

The event QueueDoneEvent is raised when no OlBuffer objects are available on the queue and the operation stops. Refer to page 218 for more information.

Use the **QueueDoneHandler** delegate with QueueDoneEvent. When QueueDoneEvent is raised, the subsystem that generated the event and the time stamp of when the event occurred are passed in the GeneralEventArgs argument of the user-defined method that matches the signature of the **QueueDoneHandler** delegate.

The following is an example of an event handling routine called HandleQueueDone that handles the event QueueDoneEvent. This event handler displays a message on the form that indicates which subsystem raised the QueueDoneEvent and at what time the event occurred:

Visual C#

```
public void HandleQueueDone (object sender,
  GeneralEventArgs eventData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke(new QueueDoneHandler(HandleQueueDone),
              new object[] { sender, eventData });
         }
         else
         {
            string msg = String.Format ("Queue Done received on {0}
                element {1} at time {2}", eventData.Subsystem,
               eventData.Subsystem.Element,
               eventData.DateTime.ToString("T"));
            statusBarPanel.Text = msg;
         }
      }
```

Visual Basic

```
Public Sub HandleQueueDone(ByVal sender As Object,
   ByVal eventData As GeneralEventArgs)
         If Me.InvokeRequired Then
           Me.Invoke(New QueueDoneHandler(AddressOf HandleQueueDone),
             New Object()
            {sender, eventData})
         Else
           Dim msg As String = String.Format(
           "Queue Done received on {0} element {1} at time {2}",
            eventData.Subsystem, eventData.Subsystem.Element,
             eventData.DateTime.ToString("T"))
             statusBarPanel.Text = msg
         End If
End Sub 'HandleQueueDone
```

DriverRunTimeErrorEvent

The DriverRunTimeErrorEvent occurs when the device driver detects one of the following error conditions:

- FifoOverflow – The driver could not read data from the device FIFO (or Windows USB FIFO) fast enough, resulting in a FIFO overflow condition. To deal with this error, increase the size of the buffers, slow down the sampling rate, or stop other CPU-intensive running programs.

  Note: By setting the **AnalogInputSubsystem.StopOnError** property, you can determine how the subsystem operates if an overrun occurs. If **StopOnError** is True, the subsystem will automatically stop when an overrun is detected. If **StopOnError** is False, the subsystem will continue running if an overrun is detected.

- FifoUnderflow – The driver could not write data to the device FIFO (or Windows USB FIFO) fast enough, resulting in FIFO underflow condition. To deal with this error, increase the size of buffers, slow down the sampling rate, or stop other CPU-intensive running programs.

  Note: By setting the **AnalogOutputSubsystem.StopOnError** property, you can determine how the subsystem operates if an underrun occurs. If **StopOnError** is True, the subsystem will automatically stop when an underrun is detected. If **StopOnError** is False, the subsystem will continue running if an underrun is detected.

- DeviceOverClocked – The A/D clock (usually external clock) is running too fast on the device. To deal with this error, slow down the A/D clock.
- TriggerError – The driver detected a trigger on the device but did not act on it.
- DeviceError – Generated by the driver due to a USB bus or hardware problem. To deal with this error, stop connecting/disconnecting USB devices while streaming data to them.

Use the **DriverRunTimeErrorEventHandler** delegate with DriverRunTimeErrorEvent. When DriverRunTimeErrorEvent is raised, the subsystem that generated the event, the time stamp of when the event occurred, the error code, and the error code descriptor are passed in the DriverRunTimeErrorEventArgs argument of the user-defined method that matches the signature of the **DriverRunTimeErrorEventHandler** delegate.

The following is an example of an event handling routine called HandleDriverRunTimeErrorEvent that handles the event DriverRunTimeErrorEvent. This event handler displays a message on the form that indicates what error occurred, which subsystem raised the DriverRunTimeErrorEvent, and at what time the event occurred:

Visual C#

```
public void HandleDriverRunTimeErrorEvent (object sender,
  DriverRunTimeErrorEventArgs eventData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke(new
               DriverRunTimeErrorEventHandler(
                HandleDriverRunTimeErrorEvent),
               new object[] { sender, eventData });
         }
         else
         {
            string msg = String.Format ("Error: {0}
               Occurred on subsystem {1} element {2} at time {3}",
                eventData.Message, eventData.Subsystem,
                eventData.Subsystem.Element,
                 eventData.DateTime.ToString("T"));
            MessageBox.Show (msg, "Error");
         }
      }
```

Visual Basic

```
Public Sub HandleDriverRunTimeErrorEvent(ByVal sender As Object,
   ByVal eventData As DriverRunTimeErrorEventArgs)
         If Me.InvokeRequired Then
            Me.Invoke(New DriverRunTimeErrorEventHandler(
               AddressOf HandleDriverRunTimeErrorEvent),
                New Object() {sender, eventData})
         Else
            Dim msg As String = String.Format(
             "Error: {0} Occurred on subsystem {1}
              element {2} at time {3}", eventData.Message,
              eventData.Subsystem, eventData.Subsystem.Element,
              eventData.DateTime.ToString("T"))
            MessageBox.Show(msg, "Error")
         End If
End Sub 'HandleDriverRunTimeErrorEvent
```

InterruptOnChangeEvent

The event InterruptOnChangeEvent is raised by some devices when a digital input line changes state.

Use the **InterruptOnChangeHandler** delegate with InterruptOnChangeEvent. When InterruptOnChangeEvent is raised, the subsystem that raised the event, the time stamp of when the event occurred, the digital input lines that changed state, and the current state of the digital input port are passed in the InterruptOnChangeEventArgs argument of the user-defined method that matches the signature of the **InterruptOnChangeHandler** delegate.

The following is an example of an event handling routine called InterruptHandler that handles the event InterruptOnChangeEvent. This event handler displays a message on the form that indicates the new value of the digital input port and what digital lines changed state:

Visual C#

```
void InterruptHandler (object sender,
  InterruptOnChangeEventArgs eventData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke( new InterruptOnChangeHandler(
               InterruptHandler), new object[] {
                 sender, eventData });
         }
         else
         {
            string sNewVal = String.Format
             ("0x{0:X}",eventData.NewValue);
            newValueTextBox.Text = sNewVal;

            sNewVal = String.Format
             ("0x{0:X}",eventData.ChangedBits);
            txtChange.Text = sNewVal;
         }
      }
```

Visual Basic
```
Sub InterruptHandler(ByVal sender As Object,
  ByVal eventData As InterruptOnChangeEventArgs)

         If Me.InvokeRequired Then
           Me.Invoke(New InterruptOnChangeHandler(
            AddressOf InterruptHandler),
             New Object() {sender, eventData})
         Else
            Dim sNewVal As String = String.Format("0x{0:X}",
             eventData.NewValue)
            newValueTextBox.Text = sNewVal

            sNewVal = String.Format("0x{0:X}",eventData.ChangedBits)
            txtChange.Text = sNewVal
         End If
End Sub 'EventDoneHandler
```

EventDoneEvent

The event EventDoneEvent is raised by some devices, such as the DT340, when a digital input line changes state or when an interval timer operation is complete.

Use the **EventDoneHandler** delegate with EventDoneEvent. When EventDoneEvent is raised, the subsystem that raised the event, the time stamp of when the event occurred, and the data associated with that event are passed in the EventDoneEventArgs argument of the user-defined method that matches the signature of the **EventDoneHandler** delegate.

The following is an example of an event handling routine called HandleEventDone that handles the event EventDoneEvent. This event handler displays a message on the form that indicates the count:

Visual C#
```
void HandleEventDone (object sender,
   EventDoneEventArgs eventData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke( new EventDoneHandler (HandleEventDone),
              new object[] {sender, eventData });
         }
         else
         {
            txtEventCount.Text = eventData.Data.ToString();
         }
      }
```

<u>*Visual Basic*</u>

```
Sub HandleEventDone(ByVal sender As Object, ByVal
  eventData As EventDoneEventArgs)
         If Me.InvokeRequired Then
           Me.Invoke(New EventDoneHandler(
            AddressOf HandleEventDone), New Object()
            {sender, eventData})
         Else
            txtEventCount.Text = eventData.Data.ToString()
         End If
End Sub 'HandleEventDone
```

MeasureDoneEvent

The event MeasureDoneEvent is raised when an edge-to-edge measurement (Measure) operation is complete. Refer to page 235 for more information on edge-to-edge measurement operations.

Use the **MeasureDoneHandler** delegate with MeasureDoneEvent. When MeasureDoneEvent is raised, the subsystem that raised the event, the time stamp of when the event occurred, and the count are passed in the MeasureDoneEventArgs argument of the user-defined method that matches the signature of the **MeasureDoneHandler** delegate.

The following is an example of an event handling routine called HandleMeasureDone that handles the event MeasureDoneEvent. This event handler displays a message on the form that indicates the count:

<u>*Visual C#*</u>

```
void HandleMeasureDone (object sender,
  MeasureDoneEventArgs eventData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke( new MeasureDoneHandler(
             HandleMeasureDone), new object[] {
              sender, eventData });
         }
         else
         {
            txtEventCount.Text = eventData.Count.ToString();
         }
      }
```

<u>*Visual Basic*</u>

```
Sub HandleMeasureDone (ByVal sender As Object,
  ByVal eventData As MeasureDoneEventArgs)
         If Me.InvokeRequired Then
           Me.Invoke(New MeasureDoneHandler(
            AddressOf HandleMeasureDone),
             New Object() {sender, eventData})
         Else
            txtEventCount.Text = eventData.Count.ToString()
         End If
End Sub 'HandleMeasureDone
```

GeneralFailureEvent

The event GeneralFailureEvent is raised when a general library failure occurs.

Use the **GeneralFailureHandler** delegate with GeneralFailureEvent. When GeneralFailureEvent is raised, the subsystem that raised the event and the time stamp of when the event occurred are passed in the GeneralEventArgs argument of the user-defined method that matches the signature of the **GeneralFailureHandler** delegate.

You can add your own code to the handler to deal with this event as needed.

DeviceRemovedEvent

The event DeviceRemovedEvent is raised when a device is removed from your system while your application is running.

Use the **DeviceRemovedHandler** delegate with DeviceRemovedEvent. When DeviceRemovedEvent is raised, the subsystem that raised the event and the time stamp of when the event occurred are passed in the GeneralEventArgs argument of the user-defined method that matches the signature of the **DeviceRemovedHandler** delegate.

You can add your own code to the event handling method to deal with this event as needed.

Handling Errors

Errors are generated by the DT-Open Layers .NET Class Library as OlException objects. Each OlException object contains an OlError object, which contains the error code and its description. Your program should handle exceptions as they occur, performing the appropriate actions to deal with any errors that arise.

Refer to Appendix A for a list of error codes and messages. These values are defined as enumerations that are accessible using the **OlException.ErrorCode** and **OlException.Message** properties. If you want to determine which subsystem generated the error, use the **OlException.Subsystem** property.

The following example shows how to catch exceptions in your program; this example the error message is printed to text field on the form:

Visual C#

```
catch (OlException ex)
   {
      string err = ex.Message;
      statusBarPanel.Text = err;
      return;
   }
```

Visual Basic

```
Catch ex As OlException
  Dim err As String = ex.Message
  statusBarPanel.Text = err
  Return
```

Cleaning Up Operations

When you are finished performing data acquisition operations, clean up the memory and resources that were used by the operation by doing the following:

1. Release the simultaneous start list, if used, using the **SimultaneousStart.Clear** method.
2. Deallocate any buffers, if used. Refer to page 227 for more information.
3. Release the subsystem connection to the device using the **Dispose** method within the appropriate subsystem class.
4. Release the Device object using the **Device.Dispose** method.

4

Using the OpenLayers.DeviceCollection Namespace

Overview

To perform a data acquisition operation, you need to do the following:

1. Import the namespace for the library.
2. Get a DeviceMgr object to manage DT-Open Layers devices.
3. Get a Device object for each DT-Open Layers device that you want to use.
4. Get a subsystem of each type that you want to use.
5. Determine what channels are supported on each subsystem, and set up channel parameters.
6. Set up and configure the subsystem.
7. Perform the I/O operations.
8. Start subsystems simultaneously, if supported.
9. Auto-calibrate the subsystem, if supported.
10. Handle events.
11. Handle errors.
12. When finished, clean up the memory and resources used by the operations.

The remaining sections in this chapter describe these steps in detail.

Importing the Namespace for the Library

To use any of the classes in the OpenLayers.DeviceCollection namespace, you first need to import the namespace into your program, as follows:

Visual C#
```
using OpenLayers.DeviceCollection
```

Visual Basic
```
Imports OpenLayers.DeviceCollection
```

Getting a DeviceMgr Object

Before performing any operation using the OpenLayers.DeviceCollection namespace, you must first use the **DeviceMgr.Get** method to return a DeviceMgr object. The DeviceMgr object is responsible for managing all DT-Open Layers device collections in your system.

The following examples shows how to get a DeviceMgr object:

Visual C#
```
DeviceMgr deviceMgr = DeviceMgr.Get();
```

Visual Basic
```
deviceMgr As DeviceMgr = DeviceMgr.Get()
```

Getting a Device Object

Once you have a DeviceMgr object, use the **DeviceMgr.GetDevice** method to return a Device object for each device collection that you want to use.

Note: If you wish, you can also create a Device object using the **Device** constructor instead of using the **GetDevice** method.

The following examples shows how to get a Device object for the device collection named *CollectionName*:

Visual C#
```
Device device = deviceMgr.GetDevice (CollectionName);
```

Visual Basic
```
device As Device = deviceMgr.GetDevice(CollectionName)
```

You can determine if a DT-Open Layers-compatible device collection is plugged into your system by using the **DeviceMgr.HardwareAvailable** method. If this method returns True, at least one DT-Open Layers-compatible device collection is plugged into your system.

To determine the names of all DT-Open Layers-compatible device collections plugged into your system, use the **DeviceMgr.GetDeviceNames** method.

You can also use the use the following properties and/or methods to return information about the specified Device object:

- **Device.CollectionDevices** property – Returns an array of Device objects for each device in the collection. The array is ordered by the collection device numbers (0 to *n*) with devices 0 and *n* being at each end of the Sync Bus chain.
- **Device.DeviceName** property – Returns the user-defined name for the device collection. You can modify this name using the DT Device Collection Manager application.
- **Device.MasterIndex** property – Returns the index of the master Device object in the CollectionDevices array.
- **Device.GetHardwareinfo** method – Returns the collection ID, number of devices in the collection, and the vendor ID for the specified device collection. See page 138 for more information on these fields.

Getting a Subsystem

The following subsystem types are defined in the OpenLayers.DeviceCollection namespace:

- AnalogInputSubsystem – This subsystem type represents the analog input channels of your device collection. Use this subsystem type if you want to acquire data from the analog input channels.

  If your device collection supports streaming digital input, counter/timer, and or quadrature decoder data through the analog input subsystem, use AnalogInputSubsystem to read this data.

- AnalogOutputSubsystem – This subsystem type represents the analog output channels of your device collection, if supported. Use this subsystem type if you want to update the values of the analog output channels.

  Note: The AnalogOutputSubsystem type is supported for the device collection only if the analog output subsystem of your device supports expansion through the Sync Bus. Refer to the hardware documentation for your device for more information.

  If your device supports streaming digital output data through the analog output subsystem, use AnalogOutputSubsystem to update the data on the digital output ports.

Your device may support all or a subset of these functions or subsystem types. In addition, your device may support multiple instances, called elements, of the same subsystem type. 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.

Once you have a Device object, you need to get a subsystem of the appropriate type for each subsystem element that you want to use. While you can do this using the constructor provided in each subsystem class, it is recommended that you use one of the following methods of the Device class:

- **Device.AnalogInputSubsystem** method – Returns an analog input subsystem for a specified element and Device object. Most DT-Open Layers device collections group all the analog input channels into one analog input subsystem element (0).

  The following example shows how to get an AnalogInputSubsystem object for element 0:

  Visual C#
  ```
  AnalogInputSubsystem ainSS = device.AnalogInputSubsystem (0);
  ```
  Visual Basic
  ```
  ainSS As AnalogInputSubsystem = device.AnalogInputSubsystem(0)
  ```

- **Device.AnalogOutputSubsystem** method – Returns an analog output subsystem for a specified element and Device object. Most DT-Open Layers device collections group all the analog output channels into one analog output subsystem element (0). The following example shows how to get an AnalogOutputSubsystem object for element 0:

  Visual C#
  ```
  AnalogOutputSubsystem aoutSS = device.AnalogOutputSubsystem (0);
  ```
  Visual Basic
  ```
  aoutSS As AnalogOutputSubsystem = device.AnalogOutputSubsystem(0)
  ```

You can determine the type of a specified subsystem by using the **SubsystemTyp**e property within the appropriate subsystem class.

To return the number of elements supported by a specified subsystem type on a specified device, use the **Device.GetNumSubsystemElements** method.

You can determine the state of a subsystem using the **State** property within the appropriate subsystem class. The following states have been defined:

- Initialized – The subsystem has been initialized, but not configured.
- ConfiguredForSingleValue – The subsystem has been configured for a single-value operation.
- ConfiguredForContinuous – The subsystem has been configured for a continuous operation.
- Running – The subsystem is running.

Note: You can also use the **IsRunning** property within the appropriate subsystem class to determine if the subsystem is running.

- Stopping – The operation on the subsystem is in the process of stopping.
- Aborting – The operation on the subsystem is in the process of being aborted.
- Prestarted – The subsystem has been prestarted for a continuous simultaneous operation.
- IOComplete – For analog input subsystems, the final post-trigger samples has been copied to the user buffer. For analog output subsystems, the final analog output sample has been written from the FIFO on the device; this is a transient state, which may not be seen, but does occur.

Determining the Available Channels and Setting up Channel Parameters

When you get a subsystem of a specified type, the software automatically determines the number of available channels for the subsystem and creates a SupportedChannelInfo object for each channel. The SupportedChannelInfo object contains the following information:

- physical channel number
- logical channel number
- logical channel word
- channel name
- I/O type
- Information that pertains to voltage input channels:
  - sensor gain
  - sensor offset
- Information that pertains to accelerometer (IEPE) channels:
  - coupling
  - excitation current source
  - value for the internal excitation current source

To get a collection of SupportedChannelInfo objects, use the SupportedChannels class.

You can get the SupportedChannelInfo object for a specific channel using the **SupportedChannels.GetChannelInfo** method and any one of the following arguments:

- The physical channel number.
- The user-defined name of the channel.
- The subsystem type and logical channel number.
- The subsystem type, logical channel number, and logical channel word.

You can also use the **SupportedChannels.Item** ([]) property to return the SupportedChannelsInfo object at a specific index.

The following subsections describe the elements of the SupportedChannelsInfo class in more detail.

Physical and Logical Channels

The logical channel number, which is zero-based, maps the physical channel to the channel's subsystem type. For example, Table 74 lists the SupportedChannels object for the analog input subsystem of a VIBbox-64 system. The VIBbox-64 collection consists of four DT9857E modules (0 to 3) and contains 64 analog input channels, four tachometers, 12 counter/timers (4 general-purpose counter/timers and 8 measure counters), and four digital input ports.

As you can see, physical channels 0 to 63 map to logical channels 0 to 15 of the analog input subsystem for each DT9857E module in the device collection. Physical channels 64 to 68 map to the extended channels (tachometer, counter/timers, and digital input port for device 0 in the device collection, physical channels 69 to 73 map to the extended channels for device 1 in the device collection, and physical channels 74 to 78 map to the extended channels for device 2 in the device collection, and physical channels 79 to 83 map to the extended channels for device 3 in the device collection.

Table 74: Example of Logical and Physical Channels in a SupportedChannels Object for an Analog Input Subsystem

Subsystem Type	Device in Collection	Logical Channel Number	Physical Channel Number
Analog Input	0	0	0
		1	1
		2	2
		3	3
		4	4
		5	5
		6	6
		7	7
		8	8
		9	9
		10	10
		11	11
		12	12
		13	13
		14	14
		15	15

Table 74: Example of Logical and Physical Channels in a SupportedChannels Object for an Analog Input Subsystem (cont.)

Subsystem Type	Device in Collection	Logical Channel Number	Physical Channel Number
Analog Input	1	0	16
		1	17
		2	18
		3	19
		4	20
		5	21
		6	22
		7	23
		8	24
		9	25
		10	26
		11	27
		12	28
		13	29
		14	30
		15	31
	2	0	32
		1	33
		2	34
		3	35
		4	36
		5	37
		6	38
		7	39
		8	40
		9	41
		10	42
		11	43
		12	44
		13	45
		14	46

Table 74: Example of Logical and Physical Channels in a SupportedChannels Object for an Analog Input Subsystem (cont.)

Subsystem Type	Device in Collection	Logical Channel Number	Physical Channel Number
Analog Input	2	15	47
	3	0	48
		1	49
		2	50
		3	51
		4	52
		5	53
		6	54
		7	55
		8	56
		9	57
		10	58
		11	59
		12	60
		13	61
		14	62
		15	63
Tachometer	0	0	64
Counter/Timer	0	0	65
		1	66
		2	67
Digital Input	0	0	68
Tachometer	1	0	69
Counter/Timer	1	0	70
		1	71
		2	72
Digital Input	1	0	73
Tachometer	2	0	74
Counter/Timer	2	0	75
		1	76
		2	77

Table 74: Example of Logical and Physical Channels in a SupportedChannels Object for an Analog Input Subsystem (cont.)

Subsystem Type	Device in Collection	Logical Channel Number	Physical Channel Number
Digital Input	2	0	78
Tachometer	3	0	79
Counter/Timer	3	0	80
		1	81
		2	82
Digital Input	3	0	83

You can determine the number of a physical channel for a given subsystem using the **SupportedChannelInfo.PhysicalChannelNumber** property.

You can determine the number of a logical channels for a given subsystem using the **SupportedChannelInfo.LogicalChannelNumber** property.

To reference a channel by number, specify either the physical channel number or the subsystem type and logical channel number.

Logical Channel Word

Some channels, such as 32-bit counter/timers on some devices, return multi-word data. The logical channel word, which is zero-based, maps the physical channel to the data word that it returns. For example, if a 32-bit counter/timer corresponds use two physical channels, the first physical channel corresponds to logical channel word 0 (the first 16-bits of data), and the second physical channel corresponds to logical channel word 1 (the second 16-bits of data).

For channels that do not return multi-word data, the value of the logical channel word is -1.

Table 75 shows an example of the logical channel words.

Table 75: Example of Logical Channel Words in a SupportedChannels Object for an Analog Input Subsystem

Subsystem Type	Logical Channel Number	Physical Channel Number	Logical Channel Word
Analog Input	0	0	–1
	1	1	–1
	2	2	–1
	3	3	–1
	4	4	–1
	5	5	–1
	6	6	–1
	7	7	–1
Counter/Timer	0	8	0
	0	9	1
	1	10	0
	1	11	1

You can determine the value of the logical channel word for a given channel using the **SupportedChannelInfo.LogicalChannelWord** property.

To reference a channel by logical channel word, specify the subsystem type, logical channel number, and logical channel word.

Channel Name

By default, each channel that is listed in the SupportedChannelInfo class has a name that describes the subsystem type and includes the logical channel number and logical channel word, if applicable. Examples of default names include Ain0 for analog input channel 0, Aout1 for analog output channel 1, Din0 for digital input channel 0, Dout2 for digital output channel 2, CT0 Word 1 for counter/timer channel 0 (word 1), and Quad1 Word 0 for quadrature decoder channel 1 (word 0).

You can specify your own name for a channel using the **SupportedChannelInfo.Name** property.

To reference a channel by name, specify the name of the channel.

IOType

You can determine what kind of I/O operation is supported for a particular channel of a given subsystem using the **SupportedChannelInfo.IOType** property.

This property returns one of the following I/O types:

- VoltageIn
- VoltageOut
- DigitalInput
- DigitalOutput
- QuadratureDecoder
- CounterTimer
- Tachometer
- Current
- Thermocouple
- Rtd
- StrainGage
- Accelerometer
- Bridge
- Thermistor
- Resistance
- MultiSensor

Note: Currently, device collections support only VoltageIn and Accelerometer I/O types.

Setting Up Voltage Input Channels

To determine whether a specific channel of a device collection supports voltage inputs, use the **SupportedChannelInfo.IOType** property.

You can read a single voltage value from one channel using the **AnalogInputSubsystem. GetSingleValueAsVolts** method. Refer to page 292 for more information.

If you are acquiring data to a buffer, you can read the voltage value from the specified channels using the **OlBuffer.GetDataAsVolts** method. Refer to page 325 for more information.

Sensor Gain and Offset

If you want to read a value from a channel in engineering units, like pressure, and your channel supports voltage measurements only, you can specify the gain and offset for the sensor using the **SupportedChannelInfo.SensorGain** and **SupportedChannelInfo.SensorOffset** properties.

The sensor gain and offset are used to scale a sample from raw counts to a sensor format. The scaling occurs in two steps. First, the raw count value is converted to prescaled voltage using the gain applied to the input signal. Then, the prescaled voltage is scaled using the following equation:

```
y = mx + b
```

where *y* is the scaled sensor value, *m* is the sensor gain, *x* is the prescaled value in voltage, and *b* is the sensor offset.

The following example shows how to set the sensor gain and offset of channel 0 of the analog input subsystem using the SupportedChannels object:

Visual C#

```
SupportedChannelInfo Ch0Info =
  ainSS.SupportedChannels.GetChannelInfo(
    SubsystemType.AnalogInput,0);
.
.
// Set the sensor gain and offset
Ch0Info.SensorGain = 2;
Ch0Info.SensorOffset = 10;
```

Visual Basic

```
Dim Ch0Info As SupportedChannelInfo =
  ainSS.SupportedChannels.GetChannelInfo(
    SubsystemType.AnalogInput, 0)
.
.
' Set the sensor gain and offset
Ch0Info.SensorGain = 2
Ch0Info.SensorOffset = 10
```

Setting Up Accelerometer (IEPE) Input Channels

To determine if the analog input subsystem supports IEPE inputs, use the **AnalogInputSubsystem.SupportsIepe** property.

For channels that support accelerometers (IEPE inputs), you can set the following properties:

- Coupling
- Excitation current source

Coupling

To determine if the analog input subsystem supports DC coupling (where DC offset is included), use the **AnalogInputSubsystem.SupportsDCCoupling** property. To determine if the analog input subsystem supports AC coupling (where the DC offset is removed), use the **AnalogInputSubsystem.SupportsACCoupling** property.

You can specify one of the coupling type using the **SupportedChannelInfo.Coupling** property. By default, DC coupling is used.

Excitation Current Source Values

To determine if the analog input subsystem supports an internal excitation current source, use the **AnalogInputSubsystem.SupportsInternalExcitationCurrentSrc** property. To determine if the analog input subsystem supports an external excitation current source, use the **AnalogInputSubsystem.SupportsExternalExcitationCurrentSrc** property.

You can specify the excitation current source (Internal, External, or Disabled) using the **SupportedChannelInfo.ExcitationCurrentSource** property. By default, the excitation current source is disabled.

If you set the excitation current source to Internal, you can also set the value of the excitation current source using the **SupportedChannelInfo.ExcitationCurrentValue** property. To determine what current source values are supported by the subsystem, use the **AnalogInputSubsystem.SupportedExcitationCurrentValues** property. By default, the first value in the list of supported values is used.

Setting Up and Configuring a Subsystem

Once you have gotten a subsystem and know about its supported channels, you can set up the subsystem for the I/O operation that you want to perform, and then configure it.

The way you set up the subsystem depends on the operation that you want to perform. Refer to the following sections for specific information on setting up I/O operations:

- For analog I/O operations, refer to page 292.
- For simultaneous operations, refer to page 330.

Call the **Config** method within the appropriate subsystem class to configure the subsystem before performing the I/O operation.

Performing Analog I/O Operations

Using the OpenLayers.DeviceCollection namespace in the DT-Open Layers for .NET Class Library, you can perform the following types of analog I/O operations.

- Single value analog input, described below
- Single value analog output, described on page 293
- Continuous pre- and post-trigger analog input using a start and reference trigger, described on page 298
- Continuous post-trigger analog input, described on page 300
- Continuously paced analog output, described on page 303
- Continuous waveform generation analog output, described on page 305

Note: On some devices, an AnalogOutputSubsystem element is used to set an analog threshold trigger; these elements support single-value analog output operations only.

Single-Value Analog Input Operations

Single-value operations are the simplest to use but offer the least flexibility and efficiency. In a single-value analog input operation, a single data value is read from a single channel. The operation occurs immediately.

To determine if the subsystem supports single-value operations, use the **AnalogInputSubsystem.SupportsSingleValue** property. If this property returns a value of True, the subsystem supports single-value operations.

Once you have an AnalogInputSubsystem object, as described on page 280, and set up the channels as described on page 288, set up the AnalogInputSubsystem object for a single value operation as follows:

1. Set the **AnalogInputSubsystem.DataFlow** property to SingleValue.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogInputSubsystem.ChannelType** property. See page 308 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogInputSubsystem.Encoding** property. See page 308 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogInputSubsystem.VoltageRange** property. See page 309 for more information on voltage ranges.

5. (Optional) For measurements that require an excitation source (such as IEPE inputs), set the excitation voltage source for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property, and if using an internal excitation source, set the value of the internal excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltageValue** property. See page 309 for more information on excitation voltage sources.
6. Configure the subsystem using the **AnalogInputSubsystem.Config** method.
7. Acquire a single value using one of the following methods:
   - **AnalogInputSubsystem.GetSingleValueAsRaw –** Acquires a single value from a specified analog input channel using a specified gain, and returns the value as a raw count.
   - **AnalogInputSubsystem.GetSingleValueAsVolts –** Acquires a single value from a specified analog input channel using a specified gain, and returns the data as a voltage.
   - **AnalogInputSubsystem.GetSingleValueAsSensor –** Acquires a single value from a specified analog input channel at a specified gain, and returns the data as a sensor value.

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

Refer to the example programs ReadSingleValueAsRaw, ReadSingleValueAsVolts, and ReadSingleValueAsSensor to see how to perform a single-value analog input operation.

Note: After the acquisition is complete, you can convert a raw count value to voltage using the **AnalogInputSubsystem.RawValueToVolts** method or to a sensor value using the **AnalogInputSubsystem.RawToSensorValues** method. If you want to convert voltage to raw counts, you can use the **AnalogInputSubsystem.VoltsToRawValue** method.

Single-Value Analog Output Operations

In a single-value analog output operation, a single data value is output from a single analog output channel. The operation occurs immediately.

To determine if the subsystem supports single-value operations, use the **AnalogOutputSubsystem.SupportsSingleValue** property. If this property returns a value of True, the subsystem supports single-value operations.

Once you have an AnalogOutputSubsystem object, as described on page 280, set up the AnalogOutputSubsystem object for a single value operation as follows:

1. Set the **AnalogOutputSubsystem.DataFlow** property to SingleValue.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogOutputSubsystem.ChannelType** property. See page 308 for more information on channel types.

3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogOutputSubsystem.Encoding** property. See page 308 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogOutputSubsystem.VoltageRange** property. See page 309 for more information on voltage ranges.
5. Configure the subsystem using the **AnalogOutputSubsystem.Config** method.
6. Output a single value using one of the following methods:

   For Devices with Multiplexed D/A architectures:

   – **AnalogOutputSubsystem.SetSingleValueAsRaw** – Outputs a single raw count on the specified analog output channel.

   – **AnalogOutputSubsystem.SetSingleValueAsVolts** – Outputs a single voltage value on a specified analog output channel.

   For Devices with Simultaneous D/A architectures (SupportsSetSingleValues is True):

   – **AnalogOutputSubsystem.SetSingleValuesAsRaw** – Outputs a single raw count on each specified analog output channel. If an analog output channel is not specified, the value of the output channel will not change; the output channel maintains the last value that was written to it.

   – **AnalogOutputSubsystem.SetSingleValuesAsVolts** – Outputs a single voltage value on each specified analog output channel. If an analog output channel is not specified, the value of the output channel will not change; the output channel maintains the last value that was written to it.

Note: You can convert a raw count value to voltage using the **AnalogOutputSubsystem.RawValueToVolts** method. If you want to convert voltage to raw counts, you can use the **AnalogOutputSubsystem.VoltsToRawValue** method.

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

Refer to the example programs WriteSingleValueAsRaw, WriteSingleValueAsVolts, and WriteSingleValueAsRawProgRanges to see how to perform a single-value analog output operation.

Continuous, Pre- and Post-Trigger Analog Input Operations Using a Start and Reference Trigger

Note: This mode requires use of an **AnalogInputSubsystem.Trigger** object and **AnalogInputSubsystem.ReferenceTrigger** object. Some devices may not support this mode.

Use this mode when you want to acquire pre-trigger data from multiple analog input channels continuously when a specified trigger occurs and, when a reference trigger occurs, acquire a specified number of post-trigger samples.

Once you have an AnalogInputSubsystem object, as described on page 280, and set up the channels as described on page 288, set up the AnalogInputSubsystem object for a continuous operation as follows:

1. Set the **AnalogInputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogInputSubsystem.ChannelType** property. See page 308 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogInputSubsystem.Encoding** property. See page 308 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogInputSubsystem.VoltageRange** property. See page 309 for more information on voltage ranges.
5. (Optional) For measurements that require an excitation source (such as IEPE inputs), set the excitation voltage source for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property, and if using an internal excitation source, set the value of the internal excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltageValue** property. See page 309 for more information on excitation voltage sources.
6. Set up the channel list (including setting the gain and inhibit value for each entry), as described on page 310. The channel list must include at least one channel from the master device in the device collection.

Note: If you want to continuously acquire data from the digital input, counter/timer, tachometer, and/or quadrature decoder channels as part of the analog input stream, you must set up the channel list to include these channels. For counter/timer and quadrature decoder channels, you must also configure and start these subsystems before starting the analog input operation. For digital input ports, you must configure the digital input subsystem for a single-value operation before starting the analog input operation. To configure these subsystems, use the OpenLayers.Base namespace. Refer to page 229 for information on continuous digital input operations, page 232 for information on continuous counter/timer operations, page 251 for information on tachometer operations, and page 253 for information on quadrature decoder operations.

7. Set up the clock, as described on page 317.
8. Use the **AnalogInputSubsystem.Trigger.TriggerType** property to specify the trigger type that starts pre-trigger acquisition. Refer to page 318 for more information on supported trigger sources.
9. Use the **AnalogInputSubsystem.ReferenceTrigger.TriggerType** property to specify the trigger type that stops pre-trigger acquisition and starts post-trigger acquisition. Refer to page 318 for more information on supported trigger sources.
10. If the start or reference trigger type is a threshold trigger, do the following:
    a. Specify the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger.ThresholdTriggerChannel** or **AnalogInputSubsystem.ReferenceTrigger.ThresholdTriggerChannel** property. Refer to page 320 for more information.
    b. Specify a voltage value for the threshold level using the **AnalogInputSubsystem.Trigger.Level** or **AnalogInputSubsystem.ReferenceTrigger.Level** property. Refer to page 320 for more information.
11. Specify the number of samples to acquire after the reference trigger occurs using the **AnalogInputSubsystem.ReferenceTrigger.PostTriggerScanCount** property. Refer to page 322 for more information on the post-trigger scan count.
12. Set up the input buffers, as described on page 322.
13. If your program is running under a heavy CPU load, it is recommended that you set the **AnalogInputSubsystem.SynchronousBufferDone** property to True for synchronous execution of each BufferDoneEvent event in a single worker thread.
14. Configure the subsystem using the **AnalogInputSubsystem.Config** method.
15. Call the **AnalogInputSubsystem.Start** method to start the operation.

Pre-trigger acquisition begins when the start trigger is detected. When the reference trigger occurs, pre-trigger acquisition stops and post-trigger acquisition begins at the next sample. The sampled data is placed in the allocated buffer(s). The operation continues until the number of scans that you specify for **PostTriggerScanCount** has been acquired. At the point, you will get the last buffer that has valid samples; the remainder of the buffers are cancelled.

Figure 15 illustrates continuous pre- and post-trigger mode (using a start and reference trigger) with a channel list of three entries: channel 0 through channel 2. In this example, pre-trigger analog input data is acquired when the start trigger is detected. When the reference trigger occurs, the specified number of post-trigger samples (three, in this example) are acquired.

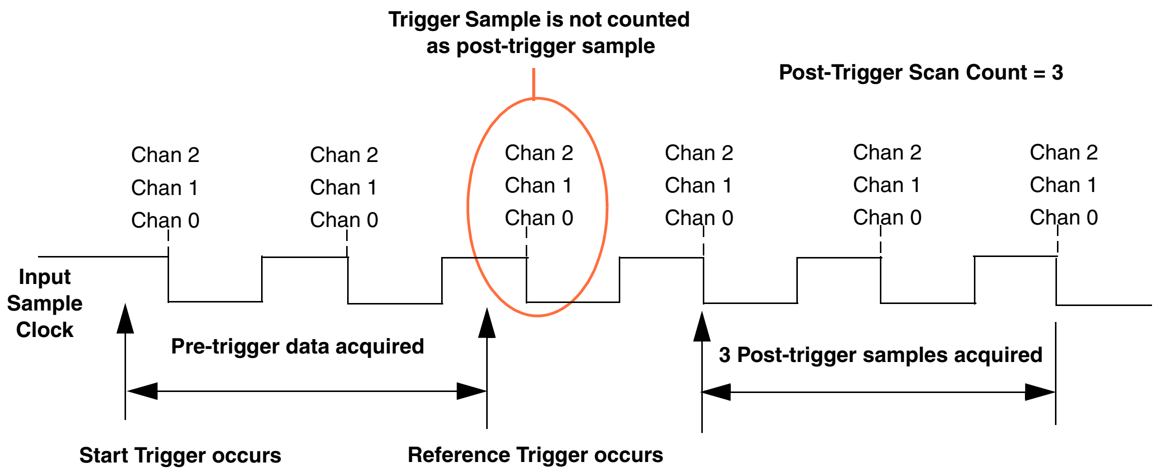

Trigger Sample is not counted as post-trigger sample
Post-Trigger Scan Count = 3
Chan 2
Chan 1
Chan 0
Input Sample Clock
Pre-trigger data acquired
3 Post-trigger samples acquired
Start Trigger occurs
Reference Trigger occurs

Figure 15: Continuous Pre- and Post-Trigger Operations Using a Start and Reference Trigger

If desired, you can also stop a continuous pre- and post-trigger operation using one of the following methods:

- **AnalogInputSubsystem.Stop** – Stops the operation after the current buffer has been filled. The driver raises a BufferDoneEvent event for the completed buffer and sets the **OlBuffer.ValidSamples** property to the number of samples in the completed buffer. It then raises a BufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, before raising a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored. Refer to page 322 for more information on buffers, and to page 332 for information on dealing with events.
- **AnalogInputSubsystem.Abort** – Stops the operation immediately without waiting for the current buffer to be filled and sets the **OlBuffer.ValidSamples** property to the number of samples in the buffer. The driver raises a BufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, and then raises a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored.
- **AnalogInputSubsystem.Reset** – Stops the operation immediately without waiting for the current buffer to be filled, and reinitializes the subsystem to the default configuration.

Note: If you set the **AnalogInputSubsystem.AsynchronousStop** property to True, control returns to your program after **Stop** is called. If you set the **AsynchronousStop** property to False (the default setting) control does not return to your program after **Stop** is called until the buffer completes or 20 seconds elapses (if the buffer takes longer than 20 seconds to fill). If you try to perform another operation while the stop is in progress, an exception is raised with the error code "SubsystemStopping" and the exception message "The subsystem is in the process of stopping or aborting".

Continuous Post-Trigger Analog Input Operations Using One Channel and One Buffer

Use this mode when you want to acquire one buffer of post-trigger data from one analog input channel.

Once you have an AnalogInputSubsystem object, as described on page 280, and set up the channels as described on page 288, perform the following steps:

1. Set the **AnalogInputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogInputSubsystem.ChannelType** property. See page 308 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogInputSubsystem.Encoding** property. See page 308 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogInputSubsystem.VoltageRange** property. See page 309 for more information on voltage ranges.
5. (Optional) For measurements that require an excitation source (such as IEPE inputs), set the excitation voltage source for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property, and if using an internal excitation source, set the value of the internal excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltageValue** property. See page 309 for more information on excitation voltage sources.
6. Set up the channel list (including setting the gain and inhibit value for the channel, and adding the channel to the channel list), as described on page 310.
7. Set up the clock, as described on page 317.
8. Use the **AnalogInputSubsystem.Trigger.TriggerType** property to specify the post-trigger source that starts the operation. Refer to page 318 for more information on supported trigger sources.

9. If the trigger type is a threshold trigger, do the following:
   - **a.** Specify the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger.ThresholdTriggerChannel** property. Refer to page 320 for more information.
   - **b.** Specify a voltage value for the threshold level using the **AnalogInputSubsystem.Trigger.Level** property. Refer to page 320 for more information.
10. Call the **AnalogInputSubsystem.GetOneBuffer** method to acquire one buffer of post-trigger data from the specified channel in the channel list. You specify the number of samples to acquire in the call.

    This method is synchronous and returns only after the requested data has been acquired or the specified timeout value, in milliseconds, has been exceeded. If the buffer is not filled before the specified timeout value is exceeded, **AnalogInputSubsystem.Abort** is called and a TimeoutException is raised. If a GeneralFailureEvent or DriverRuntimeErrorEvent occurs during acquisition, an OlException with the appropriate error code is raised.
11. Handle the input buffer, as described on page 325.

When the trigger occurs, post-trigger acquisition begins. When the number of samples have been acquired or the specified timeout value is exceeded, the OlBuffer object is returned.

Refer to the example program GetOneBuffer to see how to perform a continuous (post-trigger) analog input operation using one buffer.

Continuous, Post-Trigger Analog Input Operations Using Multiple Buffers

Note: This mode does not support use of the **AnalogInputSubsystem.ReferenceTrigger** object. To use a ReferenceTrigger object, refer to page 298.

Use continuous post-trigger mode when you want to acquire data from multiple analog input channel continuously when a specified start trigger occurs.

To determine if the subsystem supports continuous, post-trigger analog input operations, use the **AnalogInputSubsystem.SupportsContinuous** property. If this property returns a value of True, the subsystem supports continuous post-trigger analog input operations.

Once you have an AnalogInputSubsystem object, as described on page 280, and set up the channels as described on page 288, set up the AnalogInputSubsystem object for a continuous operation as follows:

1. Set the **AnalogInputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogInputSubsystem.ChannelType** property. See page 308 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogInputSubsystem.Encoding** property. See page 308 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogInputSubsystem.VoltageRange** property. See page 309 for more information on voltage ranges.
5. (Optional) For measurements that require an excitation source (such as IEPE inputs), set the excitation voltage source for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property, and if using an internal excitation source, set the value of the internal excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltageValue** property. See page 309 for more information on excitation voltage sources.
6. Set up the channel list (including setting the gain and inhibit value for each entry), as described on page 310. The channel list must include at least one channel from the master device in the device collection.

Note: If you want to continuously acquire data from the digital input, counter/timer, tachometer, and/or quadrature decoder channels as part of the analog input stream, you must set up the channel list to include these channels. For counter/timer and quadrature decoder channels, you must also configure and start these subsystems before starting the analog input operation. For digital input ports, you must configure the digital input subsystem for a single-value operation before starting the analog input operation. To configure these subsystems, use the OpenLayers.Base namespace. Refer to page 229 for information on continuous digital input operations, page 232 for information on continuous counter/timer operations, page 251 for information on tachometer operations, and page 253 for information on quadrature decoder operations.

7. Set up the clock, as described on page 317.
8. Use the **AnalogInputSubsystem.Trigger.TriggerType** property to specify the post-trigger source that starts the operation. Refer to page 318 for more information on supported trigger sources.
9. If the trigger type is a threshold trigger, do the following:
   a. Specify the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger.ThresholdTriggerChannel** property. Refer to page 320 for more information.
   b. Specify a voltage value for the threshold level using the **AnalogInputSubsystem.Trigger.Level** property. Refer to page 320 for more information.
10. Set up the input buffers, as described on page 322.
11. If your program is running under a heavy CPU load, it is recommended that you set the **AnalogInputSubsystem.SynchronousBufferDone** property to True for synchronous execution of each BufferDoneEvent event in a single worker thread.
12. Configure the subsystem using the **AnalogInputSubsystem.Config** method.
13. Call the **AnalogInputSubsystem.Start** method to start the continuous post-trigger operation.

When the post-trigger is detected, the device cycles through the channel list, acquiring the value for each ChannelListEntry object 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. The event BufferDoneEvent is generated as each buffer is filled with analog input data; refer to page 332 for information on dealing with events and reading the data in the buffer.

Figure 16 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. The device wraps to the beginning of the channel list and repeats continuously.

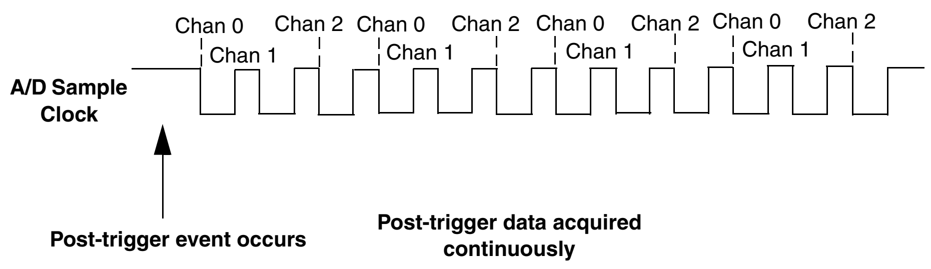


Figure 16: Continuous Post-Trigger Mode

To stop a continuous post-trigger operation, use one of the following methods:

- **AnalogInputSubsystem.Stop** – Stops the operation after the current buffer has been filled. The driver raises a BufferDoneEvent event for the completed buffer and sets the **OlBuffer.ValidSamples** property to the number of samples in the completed buffer. It then raises a BufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, before raising a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored. Refer to page 322 for more information on buffers, and to page 332 for information on dealing with events.
- **AnalogInputSubsystem.Abort** – Stops the operation immediately without waiting for the current buffer to be filled and sets the **OlBuffer.ValidSamples** property to the number of samples in the buffer. The driver raises a BufferDoneEvent event for up to eight inprocess buffers, setting the **OlBuffer.ValidSamples** property to 0, and then raises a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored.
- **AnalogInputSubsystem.Reset** – Stops the operation immediately without waiting for the current buffer to be filled, and reinitializes the subsystem to the default configuration.

Notes: If you set the **AnalogInputSubsystem.AsynchronousStop** property to True, control returns to your program after **Stop** is called. If you set the **AsynchronousStop** property to False (the default setting) control does not return to your program after **Stop** is called until the buffer completes or 20 seconds elapses (if the buffer takes longer than 20 seconds to fill). If you try to perform another operation while the stop is in progress, an exception is raised with the error code "SubsystemStopping" and the exception message "The subsystem is in the process of stopping or aborting".

Refer to the example programs ReadBufferedDataAsRaw, ReadBufferedDataAsRawDigTrigger, ReadBufferedDataAsVolts, ReadBufferedDataAsSensor, and ReadBufferedDataIntoOscilloscope to see how to perform a continuous (post-trigger) analog input operation.

Continuously Paced Analog Output Operations

Use continuously paced output mode if you want to accurately control the period between conversions of individual analog output channels in the channel list.

To determine if the subsystem supports continuous analog output operations, use the **AnalogOutputSubsystem.SupportsContinuous** property. If this property returns a value of True, the subsystem supports continuously paced analog output operations.

Once you have an AnalogOutputSubsystem object, as described on page 280, set up the AnalogOutputSubsystem object for a continuous operation as follows:

1. Set the **AnalogOutputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogOutputSubsystem.ChannelType** property. See page 308 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogOutputSubsystem.Encoding** property. See page 308 for more information on data encoding.
4. (Optional) Set the voltage range of the subsystem using the **AnalogOutputSubsystem.VoltageRange** property. See page 309 for more information on voltage ranges.
5. Set up the channel list, as described on page 310. The channel list must include at least one channel from the master device in the device collection.

Note: If you want to continuously update the digital output channels as part of the analog output stream, you must set up the channel list to include the digital output port. In addition, you must configure the digital output subsystem for a single-value operation using the OpenLayers.Base namespace before starting the analog output operation. Refer to page 229 for information on configuring the digital output subsystem.

6. (Optional) Set up the clock, as described on page 317.
7. (Optional) Use the **AnalogOutputSubsystem.Trigger.TriggerType** property to specify the trigger source that starts the operation. Refer to page 318 for more information on supported trigger sources.
8. If the trigger type is a threshold trigger, do the following:
   - **a.** Specify the channel to use for the threshold trigger using the **AnalogOutputSubsystem.Trigger.ThresholdTriggerChannel** property. Refer to page 320 for more information.
   - **b.** Specify a voltage value for the threshold level using the **AnalogOutputSubsystem.Trigger.Level** property. Refer to page 320 for more information.
9. Set the **AnalogOutputSubsystem.WrapSingleBuffer** property to False (the default value) to specify a buffer wrap mode of none. In this mode, the operation continues indefinitely as long as you process the buffers ad put them back on the queue in a timely manner.

10. Use software to fill the output buffer with the values that you want to write to the analog output channels and to the digital output port, if applicable. Refer to page 322 for more information on output buffers.
11. Configure the subsystem using the **AnalogOutputSubsystem.Config** method.
12. Call the **AnalogOutputSubsystem.Start** method to start the continuous analog output operation.

When it detects the appropriate trigger, the device starts writing output values to the channels, as determined by the channel list. The operation repeats continuously until either all the data is output from the buffers or you stop the operation. The event BufferDoneEvent occurs as each OlBuffer object is completed. If no buffers are available on the queue, the operation stops, and the event QueueDoneEvent is raised. Refer to page 322 for more information about buffers.

Make sure that the host computer transfers data to the output channel list fast enough so that the list always has data to output; otherwise, the event DriverRunTimeErrorEvent is raised. Refer to page 341 for more information on this event.

If your device supports it, you can mute the output, which attenuates the output voltage to 0 V by calling **AnalogOutputSubsystem.Mute**. This does not stop the analog output operation; instead, the analog output voltage is reduced to 0 V over a hardware-dependent number of samples. You can unmute the output voltage to its current level by calling **AnalogOutputSubsystem.UnMute**. To determine if muting and unmuting are supported by your device, read the value of the **AnalogOutputSubsystem.SupportsMute** property. If this value is True, muting and unmuting are supported.

To stop a continuous analog output operation, do not send new data to the device or use one of the following methods:

- **AnalogOutputSubsystem.Stop** – Stops the operation after all the data in the current buffer has been output. The driver raises a BufferDoneEvent event for the completed buffer and up to eight inprocess buffers, before raising a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored. Refer to page 322 for more information on buffers.
- **AnalogOutputSubsystem.Abort** – Stops the operation immediately without waiting for the data in the current buffer to be output. The driver raises a BufferDoneEvent event for the partially completed buffer and up to eight inprocess buffers, before raising a QueueStoppedEvent event. All subsequent triggers are ignored.
- **AnalogOutputSubsystem.Reset** – Stops the operation immediately without waiting for the data in the current buffer to be output, and reinitializes the subsystem to the default configuration.

Notes: If you set the **AnalogOutputSubsystem.AsynchronousStop** property to True, control returns to your program after **Stop** is called. If you set the **AsynchronousStop** property to False (the default setting) control does not return to your program after **Stop** is called until the buffer completes or 20 seconds elapses (if the buffer takes longer than 20 seconds to be output).

If you try to perform another operation while the stop is in progress, an exception is raised with the error code "SubsystemStopping" and the exception message "The subsystem is in the process of stopping or aborting".

Refer to the example program WriteBufferedDataAsVolts to see how to perform a continuously paced analog output operation.

Continuous Waveform Generation Operations

Use waveform generation mode if you want to output a waveform repetitively to analog output channels and, if supported, digital output ports, as specified in the ChannelList object.

To determine if the subsystem supports waveform generation operations, use the following properties:

- **AnalogOutputSubsystem.SupportsContinuous** property – If this property returns a value of True, continuous output operations are supported. This is a requirement for waveform generation operations.
- **AnalogOutputSubsystem.SupportsWrapSingle** property – If this property returns a value of True, the device driver will output data continuously from the first buffer queued to the analog output subsystem. This is a requirement for waveform generation operations. Refer to page 322 for more information on buffers.
- **AnalogOutputSubsystem.SupportsWaveformModeOnly** property – If this property returns a value of True, the device driver will output a waveform continuously from the onboard FIFO only. Set the **AnalogOutputSubsystem.WrapSingleBuffer** property to True. In addition, set the buffer size to be less than or equal to the FIFO size specified by the **AnalogOutputSubsystem.FifoSize** property. Refer to page 322 for more information on buffers.

Once you have an AnalogOutputSubsystem object, as described on page 280, set up the AnalogOutputSubsystem object for a continuous operation as follows:

1. Set the **AnalogOutputSubsystem.DataFlow** property to Continuous.
2. (Optional) Set the channel type of the subsystem to SingleEnded or Differential using the **AnalogOutputSubsystem.ChannelType** property. See page 308 for more information on channel types.
3. (Optional) Set the data encoding of the subsystem to Binary or TwosComplement using the **AnalogOutputSubsystem.Encoding** property. See page 308 for more information on data encoding.

4. (Optional) Set the voltage range of the subsystem using the **AnalogOutputSubsystem.VoltageRange** property. See page 309 for more information on voltage ranges.
5. Set up the channel list, as described on page 310. The channel list must include at least one channel from the master device in the device collection.

Note: If you want to continuously update the digital output channels as part of the analog output stream, you must set up the channel list to include the digital output port. In addition, you must configure the digital output subsystem for a single-value operation using the OpenLayers.Base namespace before starting the analog output operation. Refer to page 229 for information on configuring the digital output subsystem.

---

6. (Optional) Set up the clock, as described on page 317.
7. (Optional) Use the **AnalogOutputSubsystem.Trigger.TriggerType** property to specify the trigger source that starts the operation. Refer to page 318 for more information on supported trigger sources.
8. If the trigger type is a threshold trigger, do the following:
   a. Specify the channel to use for the threshold trigger using the **AnalogOutputSubsystem.Trigger.ThresholdTriggerChannel** property. Refer to page 320 for more information.
   b. Specify a voltage value for the threshold level using the **AnalogOutputSubsystem.Trigger.Level** property. Refer to page 320 for more information.
9. Set the **AnalogOutputSubsystem.WrapSingleBuffer** property to True, so that a single buffer is reused.
10. Use software to fill the output buffer with the values that you want to write to the analog output channels and to the digital output port, if applicable. Refer to your device documentation for details on the waveform pattern that you can specify and to page 322 for more information on output buffers.

Note: For devices that have a FIFO onboard for waveform generation operations, the device driver downloads the buffer into the FIFO on the device if the size of the buffer is less than or equal to the FIFO size. The driver (or device) outputs the data starting from the first location in the FIFO. When it reaches the end of the FIFO, the driver (or device) continues outputting data from the first location of the FIFO and the process continues indefinitely until you stop it.

You can determine the size of the FIFO on the device using the **AnalogOutputSubsystem.FifoSize** property. This property returns the actual FIFO size in kilobytes.

---

11. Configure the subsystem using the **AnalogOutputSubsystem.Config** method.
12. Call the **AnalogOutputSubsystem.Start** method to start the continuous analog output operation.

When it detects a trigger, the host computer writes the pattern in the buffer to specified output channels, as determined by the channel list.

If your device supports it, you can mute the output, which attenuates the output voltage to 0 V by calling **AnalogOutputSubsystem.Mute**. This does not stop the analog output operation; instead, the analog output voltage is reduced to 0 V over a hardware-dependent number of samples. You can unmute the output voltage to its current level by calling **AnalogOutputSubsystem.UnMute**. To determine if muting and unmuting are supported by your device, read the value of the **AnalogOutputSubsystem.SupportsMute** property. If this value is True, muting and unmuting are supported.

To stop a continuous analog output operation, do not send new data to the device or use one of the following methods:

- **AnalogOutputSubsystem.Stop** – Stops the operation after all the data in the current buffer has been output. The driver raises a BufferDoneEvent event for the completed buffer and up to eight inprocess buffers, before raising a QueueStoppedEvent event. All subsequent triggers or retriggers are ignored. Refer to page 322 for more information on buffers.
- **AnalogOutputSubsystem.Abort** – Stops the operation immediately without waiting for the data in the current buffer to be output. The driver raises a BufferDoneEvent event for the partially completed buffer and up to eight inprocess buffers, before raising a QueueStoppedEvent event. All subsequent triggers are ignored.
- **AnalogOutputSubsystem.Reset** – Stops the operation immediately without waiting for the data in the current buffer to be output, and reinitializes the subsystem to the default configuration.

Notes: If you set the **AnalogOutputSubsystem.AsynchronousStop** property to True, control returns to your program after **Stop** is called. If you set the **AsynchronousStop** property to False (the default setting) control does not return to your program after **Stop** is called until the buffer completes or 20 seconds elapses (if the buffer takes longer than 20 seconds to be output).

If you try to perform another operation while the stop is in progress, an exception is raised with the error code "SubsystemStopping" and the exception message "The subsystem is in the process of stopping or aborting".

Setting the Channel Type

The DT-Open Layers for .NET Class Library supports the following channel types for a specified analog I/O subsystem:

- **SingleEnded –** 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 **SupportsSingleEnded** property of the appropriate subsystem. If this property returns a value of True, the subsystem supports single-ended inputs.

  To determine how many single-ended channels are supported by the subsystem, use the **MaxSingleEndedChannels** property of the appropriate subsystem.

- **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 **SupportsDifferential** property of the appropriate subsystem. If this property returns a value of True, the subsystem supports differential inputs.

  To determine how many differential channels are supported by the subsystem, use the **MaxDifferentialChannels** property of the appropriate subsystem.

Set and/or return the channel type using the **ChannelType** property of the appropriate subsystem.

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.

Setting the Data Encoding

Two data encoding types are available: binary and twos complement.

To determine if your subsystem supports binary data encoding, use the **SupportsBinaryEncoding** property of the appropriate subsystem. If this property returns a value of True, the subsystem supports binary data encoding.

To determine if your subsystem supports twos complement data encoding, use the **SupportsTwosCompEncoding** property of the appropriate subsystem. If this property returns a value of True, the subsystem supports twos complement data encoding.

Use the **Encoding** property of the appropriate subsystem to specify the data encoding type.

Setting the Voltage Range

To determine how many ranges the subsystem supports, use the **NumberOfRanges** property of the appropriate subsystem.

To determine all the available voltage ranges for your subsystem, use the **SupportedVoltageRanges** property of the appropriate subsystem.

Some analog output subsystems support both voltage and current output channels. To determine if the subsystem supports current outputs, use the **AnalogOutputSubsystem.SupportsCurrentOutput** property.

Use the **VoltageRange** property of the appropriate subsystem to set or return the voltage range for the subsystem.

Note: If you are using a current output channel, determine how the voltage range maps to your current output range and write the appropriate voltage to the output channel.

The following example shows how to set the voltage range for an analog input subsystem to the first range in the list of supported voltage ranges:

Visual C#
```
ainSS.VoltageRange = ainSS.SupportedVoltageRanges[0];
```

Visual Basic
```
ainSS.VoltageRange = ainSS.SupportedVoltageRanges(0)
```

Setting the Excitation Voltage Source and Value

To determine if the analog input subsystem supports an internal excitation voltage source, use the **AnalogInputSubsystem.SupportsInternalExcitationVoltageSrc** property. To determine if the analog input subsystem supports an external excitation voltage source, use the **AnalogInputSubsystem.SupportsExternalExcitationVoltageSrc** property.

You specify the excitation voltage source to use (Internal, External, or Disabled) for the subsystem using the **AnalogInputSubsystem.ExcitationVoltageSource** property. By default, the excitation voltage source is disabled.

If you set the excitation voltage source to Internal, you can also set the value of the excitation voltage source using the **SupportedChannelInfo.ExcitationVoltageValue** property.

You can determine the minimum allowable value for the internal excitation voltage source using the **AnalogInputSubsystem.MinExcitationVoltageValue** property. Similarly, you can determine the maximum allowable value for the internal excitation voltage source using the **AnalogInputSubsystem.MaxExcitationVoltageValue** property.

Setting up the Channel List

Note: Single-value operations do not use a channel list.

If you want to acquire data from or update multiple channels, you need to use a continuous operation mode and specify the channels that you want to sample (and the order in which to sample them) in a ChannelList object.

Channels are sampled or updated in order from the first entry to the last entry in the ChannelList object. Channel numbering is zero-based; that is, the first entry in the ChannelList is at index 0, the second entry is at index 1, and so on.

The **ChannelList** property is accessible using any subsystem class whose **SupportsContinuous** property is True. Typically, a ChannelList is used with the AnalogInputSubsystem and AnalogOutputSubsystem classes.

For an analog input subsystem, you can specify analog input channels, as well as digital inputs, counter/timers, and/or quadrature decoders in the ChannelList object, if your device supports it. Similarly, for an analog output subsystem, you can specify analog output channels as well as digital outputs in the ChannelList object, if your device supports it. Refer to page 282 for more information on available channels.

Note: For a device collection, the channel list must include at least one channel from the master device in the device collection.

You can add sequential channels (such as channels 0, 1, 2, 3) or random channels (such as channels 2, 9, 7) to the ChannelList object, and can specify a channel more than once in the list (such as channels 1, 2, 1), if your device supports it.

Other devices may limit the order in which you can enter a channel in the channel list. See the user's manual for your device to determine any channel ordering limitations.

The following example shows a ChannelList that contains four channels. Channel 1 is sampled first, followed by channel 2, channel 1 again, and then channel 0:

Table 76: Example of a ChannelList Object

Channel-List Index	Channel	Description
0	1	Sample channel 1.
1	2	Sample channel 2.
2	1	Sample channel 1 again.
3	0	Sample channel 0.

Adding Channels to a Channel List

The **ChannelList.Add** method adds a channel to the end of the ChannelList object, and returns the index of the added channel. You can specify the channel to add in one of the following ways:

- By physical channel number
- By channel name
- By ChannelListEntry object

The following sections describe these methods.

Adding Channels By Physical Channel Number

This method is the simplest way to add channels into the ChannelList object, particularly if you are adding channels that are native to the subsystem type (such as analog input channels on an analog input subsystem).

For native channels, the physical channel number always equals the logical channel number. While non-native channels, such as digital inputs that are streamed through the analog input subsystem, can also be added this way, the physical channel number is not the same as the logical channel number, so you may find it easier to add the channel by name or by ChannelListEntry object instead.

A new ChannelListEntry object is returned for each physical channel that is added this way. Refer to page 314 for more information on ChannelListEntry objects.

The following example shows how to use the **Add** method to add physical channel 0 to the end of a ChannelList for an analog input subsystem:

Visual C#
```
ch = AinSS.ChannelList.Add(0);
```

Visual Basic
```
ch = AinSS.ChannelList.Add(0)
```

Adding Channels By Channel Name

The channel name is the name that you assigned to the channel using the SupportedChannelInfo class, described on page 282. A new ChannelListEntry object is returned for each channel that is added this way. Refer to page 314 for more information on ChannelListEntry objects.

The following example shows how to use the **Add** method to add a channel named Sensor to the end of a ChannelList for an analog input subsystem:

Visual C#
```
//Specify the name Sensor for the first
//analog input channel.
ainSS.SupportedChannels[0].Name = "Sensor";
//Add the channel named Sensor to the ChannelList
ch = ainSS.ChannelList.Add("Sensor");
```

Visual Basic
```
'Specify the name Sensor for the first
'analog input channel.
ainSS.SupportedChannels(0).Name = "Sensor"
ch = AinSS.ChannelList.Add("Sensor")
```

Adding Channels By ChannelListEntry Object

This method is useful if you want a more generic approach to adding channels. This approach frees you from keeping track of physical channel numbers and their names.

To get a ChannelListEntry object, use the **ChannelListEntry** constructor within the ChannelListEntry class, specifying the SupportedChannelInfo object for the channel that you want to sample or update. See page 282 for more information on SupportedChannelInfo objects.

This example creates a ChannelListEntry called Ch0 for physical channel 0 of the analog input subsystem, using all the information contained in SupportedChannelInfo for that channel.

Visual C#
```
ChannelListEntry Ch0 = new ChannelListEntry (
  ainSS.SupportedChannels.GetChannelInfo
    (SubsystemType.AnalogInput, 0 ));
```

Visual Basic
```
Dim Ch0 As New ChannelListEntry (
  ainSS.SupportedChannels.GetChannelInfo
    (SubsystemType.AnalogInput, 0 ))
```

Note: It is recommended that you set the gain (see page 315) and inhibition value (page 316) for each ChannelListEntry object after you create it. However, it is possible to set or change these values after the ChannelListEntry object is added to the ChannelList.

The following example shows how to use the **Add** method to add ChannelListEntry object Ch0 to the end of a ChannelList:

Visual C#
```
AinSS.ChannelList.Add(Ch0);
```

Visual Basic
```
AinSS.ChannelList.Add(Ch0)
```

Inserting Channels in the Channel List

The **ChannelList.Insert** method inserts a channel at the specified index of a ChannelList object, incrementing all higher index entries by 1, and returns the index of the added channel. You can specify the channel to insert in one of the following ways:

- By physical channel number
- By channel name
- By ChannelListEntry object

The following sections describe these methods.

Inserting a Channel By Physical Channel Number

This method is the simplest way to insert channels into the ChannelList object, particularly if you are inserting channels that are native to the subsystem type (such as analog input channels on an analog input subsystem).

For native channels, the physical channel number always equals the logical channel number. While non-native channels, such as digital inputs that are streamed through the analog input subsystem, can also be inserted this way, the physical channel number is not the same as the logical channel number, so you may find it easier to insert the channel by name or by ChannelListEntry object instead.

A new ChannelListEntry object is returned for each physical channel that is inserted this way. Refer to page 314 for more information on ChannelListEntry objects.

The following example shows how to use the **Insert** method to insert physical channel 3 at index 0 of the ChannelList for an analog input subsystem. The channel that was formally at index 0 is now at index 1.

Visual C#
```
ch = AinSS.ChannelList.Insert(0, 3);
```

Visual Basic
```
ch = AinSS.ChannelList.Insert(0, 3)
```

Inserting a Channel By Channel Name

The channel name is the name that you assigned to the channel using the SupportedChannelInfo class, described on page 282. A new ChannelListEntry object is returned for each channel that is inserted this way. Refer to page 314 for more information on ChannelListEntry objects.

The following example shows how to use the **Insert** method to insert a channel named Ain3 at index 0 of the ChannelList for an analog input subsystem. The channel that was formally at index 0 is now at index 1.

Visual C#
```
ch = AinSS.ChannelList.Insert(0, "Ain3");
```

Visual Basic
```
ch = AinSS.ChannelList.Insert(0, "Ain3")
```

Inserting a Channel By ChannelListEntry Object

This method is useful if you want a more generic approach to inserting channels. This approach frees you from keeping track of physical channel numbers and their names.

To get a ChannelListEntry object, use the **ChannelListEntry** constructor within the ChannelListEntry class, specifying the SupportedChannelInfo object for each channel that you want to sample or update. See page 282 for more information on SupportedChannelInfo objects.

This example creates a ChannelListEntry called Ch3 for physical channel 3 of the analog input subsystem, using all the information contained in SupportedChannelInfo for that channel.

Visual C#
```
ChannelListEntry Ch3 = new ChannelListEntry (
  ainSS.SupportedChannels.GetChannelInfo
    (SubsystemType.AnalogInput, 3 ));
```

Visual Basic
```
Dim Ch3 As New ChannelListEntry (
  ainSS.SupportedChannels.GetChannelInfo
    (SubsystemType.AnalogInput, 3 ))
```

Note: It is recommended that you set the gain (see page 315) and inhibition value (page 316) for each ChannelListEntry object after you create it. However, it is possible to set or change these values after the ChannelListEntry object is added to the ChannelList.

The following example shows how to use the **Insert** method to insert ChannelListEntry object Ch3 at index 0 of the ChannelList. The channel that was formally at index 0 is now at index 1.

Visual C#
```
AinSS.ChannelList.Insert(0, Ch3);
```

Visual Basic
```
AinSS.ChannelList.Insert(0, Ch3)
```

Replacing Channels in the ChannelList

The ChannelList.Item ([]) property replaces a ChannelListEntry object at the specified index of the ChannelList. An exception is raised if an entry does not exist at the specified index.

The following example shows how to use the Item ([]) property to replace the ChannelListEntry object at index 1 of the ChannelList with ChannelListEntry object Ch3:

Visual C#
```
AinSS.ChannelList[1] = Ch3;
```

Visual Basic

```
AinSS.ChannelList(1) = Ch3
```

Removing Channels from the Channel List

To remove a ChannelListEntry from the ChannelList object, use the **ChannelList.Remove** method. This method removes the first instance of the specified ChannelListEntry object from the ChannelList object, decrementing all higher index entries by 1.

The following example shows how to remove the first instance of ChannelListEntry object Ch0 from the ChannelList object using the **Remove** method:

Visual C#

```
AinSS.ChannelList.Remove(Ch0);
```

Visual Basic

```
AinSS.ChannelList.Remove(Ch0)
```

Setting the Gain of a ChannelListEntry

The voltage range divided by the gain determines the effective range for a channel. For example, if your device provides a voltage 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.

To determine if the subsystem supports programmable gain, use the **SupportsProgrammableGain** property of the appropriate subsystem. If this property returns a value of True, programmable gain is supported.

To determine the number of gains the subsystem supports, use the **NumberofSupportedGains** property of the appropriate subsystem. To list all of the gain values supported by the subsystem, use the **SupportedGains** property.

The simplest way to specify the gain for a channel is by using a single-value operation. (In this case, a ChannelListEntry object is not used.) Refer to page 292 for more information on single-value analog input operations; refer to page 293 for more information on single-value analog output operations.

If you are using a ChannelListEntry object, specify or return the gain for each ChannelListEntry object using the **ChannelListEntry.Gain** property.

This example shows how to apply a gain of 2 to a ChannelListEntry called Ch0.

Visual C#

```
Ch0.Gain = 2;
```

Visual Basic

```
Ch0.Gain = 2
```

You can also apply gain to a ChannelListEntry in the ChannelList, as shown below; this example applies a gain of 2 to the ChannelListEntry at index 0 of the ChannelList:

Visual C#
```
AinSS.ChannelList[0].Gain = 2;
```

Visual Basic
```
AinSS.ChannelList(0).Gain = 2
```

Note: The driver sets the actual gain as closely as possible to the number specified. You can read back the exact gain after configuring the subsystem using the **Gain** property. If your subsystem does not support programmable gain, enter a value of 1 (the default value) for the gain.

Inhibiting Channels in a Channel List

If supported by your subsystem, you can inhibit data from being returned by the ChannelListEntry object. This feature is useful if you want to discard values that are acquired by specific channels.

To determine if a subsystem supports inhibition, use the **SupportsChannelListInhibit** property inherited from the SubsystemBase class. If this property returns a value of True, the subsystem supports channel inhibition.

Using the **Inhibit** property of the ChannelListEntry class, you can enable or disable inhibition for each ChannelListEntry object. If you set this property to True, the acquired value is discarded after the channel entry is sampled. If you set this property to False (the default value), the acquired value is stored after the channel entry is sampled.

This example shows how to set the channel inhibit value of the ChannelListEntry called Ch0 to True:

Visual C#
```
Ch0.Inhibit = 1;
```

Visual Basic
```
Ch0.Inhibit = 1
```

You can also set the inhibit value of a ChannelListEntry in the ChannelList, as shown below; this example sets the inhibit value to True for the ChannelListEntry at index 3 of the ChannelList:

Visual C#
```
AinSS.ChannelList[3].Inhibit = 1;
```

Visual Basic
```
AinSS.ChannelList(3).Inhibit = 1
```

Getting Information about Channels in the ChannelList Object

You can get information about the contents of a ChannelList object using the following methods:

- **ChannelList.Contains** method – Determines whether a specified ChannelListEntry object is contained in the ChannelList.
- **ChannelList.IndexOf** method – Searches for a specified channel (specified by physical channel or ChannelListEntry object) in the ChannelList and returns the zero-based index of the first occurrence within the ChannelList.
- **ChannelList.CGLDepth** property – Returns the maximum number of channels or ChannelListEntry objects that the ChannelList can contain.

Setting up a Clock Source

The OpenLayers.DeviceCollection namespace defines internal and external clock sources, described in the following subsections. Note that single-value operations do not use clocks.

Note: Some subsystems allow you to read or update multiple channels on a single clock pulse. You can determine whether multiple channels are read or updated on a single clock pulse by using the **Clock.SupportsSimultaneousClocking** property.

Internal Clock Source

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

To determine if the subsystem supports an internal clock, use the **Clock.SupportsInternalClock** property. If this property returns a value of True, an internal clock is supported.

To determine the maximum frequency supported by the internal clock, use the **Clock.MaxFrequency** property. To determine the minimum frequency supported by the internal clock, use the **Clock.MinFrequency** property.

Specify the clock source as internal using the **Clock.Source** property. Then, use the **Clock.Frequency** property to specify the frequency at which to pace the operation.

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.

The driver sets the frequency of the internal clock as close as possible to the value that you specified in the **Frequency** property. You can determine the actual frequency that was set on the hardware by reading the value of the **Frequency** property after the subsystem has been configured (using the **Config** method).

External Clock Source

The external clock is a clock source attached to the device that paces data acquisition or output for each channel in the channel 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 **Clock.SupportsExternalClock** property. If this property returns a value of True, an external clock is supported.

To determine the maximum external clock divider that the subsystem supports, use the **Clock.MaxExtClockDivider** property. To determine the minimum external clock divider that the subsystem supports, use the **Clock.MinExtClockDivider** property.

Specify the clock source as external using the **Clock.Source** property. Then, use the **Clock.ExtClockDivider** property to set or get the clock divider that is used to determine the frequency of the external clock source. The frequency of the external clock input divided by the external clock divider determines the frequency at which to pace the operation.

Setting Up a Trigger Type

Note: Single-value operations do not use triggers.

The OpenLayers.DeviceCollection namespace provides the Trigger class that can be used to set up a start trigger, and the ReferenceTrigger class that can be used to set up a reference trigger, if supported by your device. The following trigger types are available for the start and reference triggers:

- Software
- TTLPos
- TTLNeg
- ThresholdPos

- ThresholdNeg
- DigitalEvent

For devices that support a start trigger and reference trigger for performing continuous pre-and post-trigger analog input operations, specify the start trigger type using the **AnalogInputSubsystem.Trigger.TriggerType** property and specify the reference trigger type using the **AnalogInputSubsystem.ReferenceTrigger.TriggerType** property; refer to page 298 for more information on pre- and post-trigger operations using a start and reference trigger.

For devices that support continuous post-trigger operations without using a reference trigger, specify the post-trigger source using the **AnalogInputSubsystem.Trigger.TriggerType** property; refer to page 300 for more information on post-trigger operations.

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

Software 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 for the start trigger, use the **Trigger.SupportsSoftwareTrigger** property. If this property returns a value of True, a software trigger is supported.

To determine if the subsystem supports a software trigger for the reference trigger, use the **ReferenceTrigger.SupportsSoftwareTrigger** property. If this property returns a value of True, a software trigger is supported.

TTLPos Trigger Source

The TTLPos trigger source is an external digital (TTL) signal attached to the device. The trigger occurs when the device detects a transition on the rising edge of the digital TTL signal.

To determine if the subsystem supports a TTLPos trigger for a start trigger, use the **Trigger.SupportsPosExternalTTLTrigger** property. If this property returns a value of True, a TTLPos trigger is supported.

To determine if the subsystem supports a TTLPos trigger for a reference trigger, use the **ReferenceTrigger.SupportsPosExternalTTLTrigger** property. If this property returns a value of True, a TTLPos trigger is supported.

To determine if the subsystem supports a TTLPos trigger for a single-value operation, use the **Trigger.SupportsSvPosExternalTTLTrigger** property. If this property returns a value of True, a TTLPos trigger is supported.

TTLNeg Trigger Source

The TTLNeg trigger source is an external digital (TTL) signal attached to the device. The trigger occurs when the device detects a transition on the falling edge of the digital TTL signal.

To determine if the subsystem supports a TTLNeg trigger for a start trigger, use the **Trigger.SupportsNegExternalTTLTrigger** property. If this property returns a value of True, a TTLNeg trigger is supported.

To determine if the subsystem supports a TTLNeg trigger for a reference trigger, use the **ReferenceTrigger.SupportsNegExternalTTLTrigger** property. If this property returns a value of True, a TTLNeg trigger is supported.

To determine if the subsystem supports a TTLNeg trigger for a single-value operation, use the **Trigger.SupportsSvNegExternalTTLTrigger** property. If this property returns a value of True, a TTLNeg trigger is supported.

ThresholdPos Trigger Source

A threshold trigger is generally either an analog signal from an analog input channel or an external analog signal attached to the device. A positive analog threshold (ThresholdPos) trigger occurs when the device detects a positive-going signal that crosses a threshold value.

To determine if the subsystem supports a ThresholdPos trigger for the start trigger, use the **Trigger.SupportsPosThresholdTrigger** property. If this property returns a value of True, a ThresholdPos trigger is supported.

To determine if the subsystem supports a ThresholdPos trigger for the reference trigger, use the **ReferenceTrigger.SupportsPosThresholdTrigger** property. If this property returns a value of True, a ThresholdPos trigger is supported.

To determine which channels support a threshold trigger for the start trigger, use the **Trigger.SupportedThresholdTriggerChannels** property. To set the channel that you want to use for the threshold start trigger, use the **Trigger.ThresholdTriggerChannel** property.

To determine which channels support a threshold trigger for the reference trigger, use the **ReferenceTrigger.SupportedThresholdTriggerChannels** property. To set the channel that you want to use for the threshold reference trigger, use the **ReferenceTrigger.ThresholdTriggerChannel** property.

On some devices, the threshold level is set using an analog output subsystem on the device. On other devices, you set the threshold level using the **Trigger.Level** property (for the start trigger) or **ReferenceTrigger.Level** property (for the reference trigger). By default, the trigger threshold value is in voltage unless specified otherwise for the device; see the user's manual for your device for valid threshold value settings.

Note: The threshold level set by the **Trigger.Level** or **ReferenceTrigger.Level** property depends on the voltage range and gain of the subsystem. For example, if the voltage range of the analog input subsystem is ±10 V, and the specified gain is 1, specify a threshold voltage level within ±10 V. Likewise, if the voltage range of the analog input subsystem is ±10 V, and the specified gain is 10, specify a threshold voltage level within ±1 V. Refer to your device documentation for details on how to specify the threshold value for your device.

ThresholdNeg Trigger Source

A threshold trigger is generally either an analog signal from an analog input channel or an external analog signal attached to the device. A negative analog threshold trigger (ThresholdNeg) occurs when the device detects a negative-going signal that crosses a threshold value.

To determine if the subsystem supports a ThresholdNeg trigger for the start trigger, use the **Trigger.SupportsNegThresholdTrigger** property. If this property returns a value of True, a ThresholdNeg trigger is supported.

To determine if the subsystem supports a ThresholdNeg trigger for the reference trigger, use the **ReferenceTrigger.SupportsNegThresholdTrigger** property. If this property returns a value of True, a ThresholdNeg trigger is supported.

To determine which channels support a threshold trigger for the start trigger, use the **Trigger.SupportedThresholdTriggerChannels** property. To set the channel that you want to use for the threshold start trigger, use the **Trigger.ThresholdTriggerChannel** property.

To determine which channels support a threshold trigger for the reference trigger, use the **ReferenceTrigger.SupportedThresholdTriggerChannels** property. To set the channel that you want to use for the threshold reference trigger, use the **ReferenceTrigger.ThresholdTriggerChannel** property.

On some devices, the threshold level is set using an analog output subsystem on the device. On other devices, you set the threshold level using the **Trigger.Level** property (for the start trigger) or the **ReferenceTrigger.Level** property (for the reference trigger). By default, the trigger threshold value is in voltage unless specified otherwise for the device; see the user's manual for your device for valid threshold value settings.

Note: The threshold level set by the **Trigger.Level** or **ReferenceTrigger.Level** property depends on the voltage range and gain of the subsystem. For example, if the voltage range of the analog input subsystem is ±10 V, and the specified gain is 1, specify a threshold voltage level within ±10 V. Likewise, if the voltage range of the analog input subsystem is ±10 V, and the specified gain is 10, specify a threshold voltage level within ±1 V. Refer to your device documentation for details on how to specify the threshold value for your device.

DigitalEvent Trigger Source

For a DigitalEvent trigger source, a trigger is generated when an external digital event occurs.

To determine if the subsystem supports a DigitalEvent trigger for the start trigger, use the **Trigger.SupportsDigitalEventTrigger** property. If this property returns a value of True, a DigitalEvent trigger is supported.

To determine if the subsystem supports a DigitalEvent trigger for the reference trigger, use the **ReferenceTrigger.SupportsDigitalEventTrigger** property. If this property returns a value of True, a DigitalEvent trigger is supported.

Setting up a Post-Trigger Scan Count

On devices that support a reference trigger for performing continuous pre- and post-trigger analog input operations, you can specify how many samples to acquire after the reference trigger occurs using the **AnalogInputSubsystem.ReferenceTrigger.PostTriggerScanCount** property.

To determine if your device supports the ability to specify the number of post-trigger samples to acquire, use the **AnalogInputSubsystem.ReferenceTrigger. SupportsPostTriggerScanCount** property.

Setting up Buffers

Note: Single-value operations do not use buffers.

Continuous analog input and analog output operations require buffers in which to store data. For input operations, a queue exists to hold the buffers that are empty and ready for input. For output operations, the queue holds buffers that you have filled with data and are ready for output.

To determine if the subsystem supports buffering, use the **SupportsBuffering** property within the appropriate subsystem class. If this property returns a value of True, buffering is supported.

If you want to acquire one buffer of data from one channel using a continuous analog input operation, use the **AnalogInputSubsystem.GetOneBuffer** method; this method allocates an OlBuffer object of the size you specify and acquires one buffer of data for you.

For all other operations, use the **OlBuffer** constructor within the OlBuffer class to create an OlBuffer object for use with an analog input or analog output subsystem. The library automatically allocates an internal data buffer, which is encapsulated by the OlBuffer object. You specify the subsystem with which to associate the OlBuffer object as well the size (in samples) of the internal buffer to allocate.

If desired, you can use the **OlBuffer.Tag** property, if desired, to name the buffer with the contents that are contained in the buffer.

Note: If you set the size of the internal buffer that is encapsulated by an OlBuffer object and later you want to change the size, call the **OlBuffer.Reallocate** method. This method reallocates the internal buffer to the specified number of samples; the initial internal buffer is deallocated and any data that it contained is lost.

The **AnalogInputSubsystem.GetOneBuffer** method uses one buffer. Other continuous analog input operations require a minimum of two OlBuffer objects. Continuous analog output operations require a minimum of two OlBuffer objects if **WrapSingleBuffer** is False; if **WrapSingleBuffer** is True, one OlBuffer object is required.

Once you have created the OlBuffer objects for multiple buffer operations (and, for output operations, filled the corresponding internal buffers with data), put the OlBuffer objects on the queue using the **BufferQueue.QueueBuffer** method of the appropriate subsystem.

The following example shows how to create multiple OlBuffer objects for a continuous analog input operation and put them on the queue for the analog input subsystem. In this example, an internal buffer of 1024 samples is allocated when the OlBuffer object is created:

Visual C#

```
// Create the buffers
for (int i=0; i<4; ++i)
{
  AinBuffer[i] = new OlBuffer (1024, ainSS);
  // Put the buffers on the queue
  ainSS.BufferQueue.QueueBuffer (AinBuffer[i]);
}
```

Visual Basic

```
While i < 4
  ' Create the buffers
  AinBuffers(i) = New OlBuffer(1024, ainSS)
  ' Put the buffers on the queue
  ainSS.BufferQueue.QueueBuffer(AinBuffers(i))
  i += 1
End While
```

When you start a continuous operation, the device takes up to eight OlBuffer objects from the subsystem queue and begins filling them (for input operations) or outputting data from them (for output operations) at the specified clock rate. The state of these objects changes from queued to inprocess.

About QueuedCount and InProcessCount

You can determine the number of OlBuffer objects that are on the subsystem queue by using the **BufferQueue.QueuedCount** property. You can determine the number of OlBuffer objects that are inprocess by using the **BufferQueue.InProcessCount** property.

Every time an OlBuffer object transitions from the queued state to the inprocess state, the value of the **QueuedCount** property decreases by 1 and the value of the **InProcessCount** property increases by 1. For example, assume that you call **QueueBuffer** for 10 OlBuffer objects; the **QueuedCount** is 10 and the **InProcessCount** is 0. Once you call **Start** for the subsystem, up to 8 OlBuffer objects are moved from the queued state to the inprocess state. **QueuedCount** is now 2 and **InProcessCount** is 8.

If you do not put the OlBuffer objects back on the queue as they are completed, the **QueuedCount** decreases while the **InProcessCount** remains the same (as a new inprocess buffer replaces a completed buffer) until the **QueuedCount** gets to 0, then the **InProcessCount** starts decreasing until all the OlBuffer objects are completed, as shown below:

Table 77: InProcessCount Example

Completed Buffers	QueueCount	InProcessCount
0	10	0
0	2	8
1	1	8
2	0	8
3	0	7
4	0	6
5	0	5
6	0	4
7	0	3
8	0	2
9	0	1
10	0	0

Buffer Completion Events

Note: Buffer completion events are not generated if you use the **AnalogInputSubsystem.GetOneBuffer** method. This is a synchronous method that does not return until the buffer has been acquired or the timeout value has expired.

One or more of the following events is generated when a buffer is completed:

- **BufferDoneEvent** – For input operations, this event is generated when the internal buffer of the OlBuffer object has been filled with post-trigger data. For output operations, this event is generated when all the data in the internal buffer of the OlBuffer object has been output. Refer to page 334 for more information on this event.
- **PreTriggerBufferDoneEvent** – For input operations only, this event is generated when the internal buffer of the OlBuffer object has been filled with pre-trigger data. Refer to page 336 for more information on this event.
- **QueueStoppedEvent** – This event occurs when you stop a continuous analog I/O operation with **Stop** or **Abort**. Refer to page 337 for more information on this event.
- **IOCompleteEvent** – For analog input operations that use a reference trigger whose trigger type is something other than software (none), this event occurs when the last post-trigger sample is copied into the user buffer; devices that do not support a reference trigger will never receive this event for analog input operations.

  For analog output operations, this event is generated when the last data point has been output from the analog output channel. Refer to page 338 for more information on this event.
- **QueueDoneEvent** – This event is generated when no OlBuffer objects are available on the queue and the operation stops. Refer to page 340 for more information on this event.

Handling Input Buffers

Each time a BufferDoneEvent or PreTriggerBufferDoneEvent event is raised, your application program must handle the event or you will lose the data in the internal buffer of the OlBuffer object. Refer to page 332 for more information about handling events and buffers.

You can post-process OlBuffer objects, if you wish. One technique for doing this is to allocate an array that will hold the OlBuffer objects as they are completed. When the BufferDoneEvent or PreTriggerBufferDoneEvent event occurs, move the OlBuffer object into a array. When the operation is complete, process the OlBuffer objects in your array.

For continuous analog input operations, use one of the following methods to copy the data from the internal buffer of an OlBuffer object into a user-declared array/variable (the data type of this array/variable is dictated by the method/property you choose):

Note: For ease of use, all of these methods allocate the returned array to the correct size. Simply declare an array of the appropriate type for use with one these methods.

- **OlBuffer.GetDataAsRawByte –** Copies the data, as raw counts, from the internal buffer of the OlBuffer object into a user-declared array of bytes. You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of bytes.

  Note: This method is useful when writing binary data to a file. Since each sample takes more than one array entry, other uses may be limited.

- **OlBuffer.GetDataAsRawInt16 –** Used when the resolution of the subsystem is 16 bits or less and when the data encoding is twos complement, copies the data, as raw counts, from the internal buffer of the OlBuffer object into a user-declared array of signed, 16-bit integers (short). You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of signed, 16-bit integers (short).
- **OlBuffer.GetDataAsRawUInt16 –** Used when the resolution of the subsystem is 16 bits or less and when the data encoding is binary, copies the data, as raw counts, from the internal buffer of the OlBuffer object into a user-declared array of unsigned, 16-bit integers (ushort). You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of unsigned, 16-bit integers (ushort).
- **OlBuffer.GetDataAsRawUInt32 –** Used when the resolution of the subsystem is greater than 16 bits, copies the data, as raw counts, from the internal buffer of the OlBuffer object into a user-declared array of unsigned 32-bit integers (uint). You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of unsigned, 32-bit integers (uint).
- **OlBuffer.GetDataAsSensor –** Converts the data from the internal buffer of the OlBuffer object into sensor values using the specified sensor gain and offset (described on page 289), and copies this data into a user-declared array of 64-bit floating-point values (double). You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of 64-bit floating-point (double) values.
- **OlBuffer.GetDataAsVolts –** Converts the data from the internal buffer of the OlBuffer object into voltages, and copies this data into a user-declared array of 64-bit floating-point (double) values. You can copy all the data from the buffer or only the data for a specific ChannelListEntry in the buffer. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of 64-bit floating-point (double) values.

- **OlBuffer.GetDataAsVoltsByte –** For a specified ChannelListEntry, converts the data from the internal buffer of an OlBuffer object into voltage values, and then copies these voltage values into a user-declared array of bytes. Each temperature value is stored as an Int32, and takes 4 bytes.

  To use this method, first declare an array of bytes.
- **OlBuffer.GetDataAsRpm** – For a specified ChannelListEntry, converts the tachometer data from the internal buffer of an OlBuffer object into RPM (rotations per minute) values, and then copies these values into a user-declared array of 64-bit floating-point (double) values. Note that if the ChannelListEntry occurs more than once in the buffer, the data for each occurrence of the ChannelListEntry is copied.

  To use this method, first declare an array of 64-bit floating-point (double) values.
- **OlBuffer.Item** property ([]) – Copies the raw count value at the specified index of the buffer specified by the OlBuffer object into a user-declared signed, 32-bit integer variable (int).

When you have finished copying the data from the internal buffer of the OlBuffer object, you can put the OlBuffer object back on the queue for the analog input subsystem using the **AnalogInputSubsystem.BufferQueue.QueueBuffer** method.

See the example for the event BufferDoneEvent starting on page 334 for an example of using the **GetDataAsSensor** method to handle input buffers.

Handling Output Buffers

For continuous analog output operations, you need to create an array and fill it with data, then copy this data from the array to the internal buffer of the OlBuffer object using one of the following methods:

- **OlBuffer.PutDataAsRaw –** Copies raw counts from a user-specified array into the internal buffer of the OlBuffer object. This is an overloaded method that allows you to copy all the data from the array into the buffer or only the data for a specific ChannelListEntry in the array into the buffer. Note that if the ChannelListEntry occurs more than once in the array, the data for each occurrence of the ChannelListEntry is copied.

  If your subsystem supports a resolution of 16-bits or less, declare an array of unsigned, 16-bit integers (ushort) for use with this method.

  If your subsystem supports a resolution greater than 16 bits, declare an array of unsigned, 32-bit integers (uint) for use with this method.
- **OlBuffer.PutDataAsVolts –** Copies voltages from a user-specified array into the internal buffer of the OlBuffer object. This is an overloaded method that allows you to copy all the data from the array into the buffer or only the data for a specific ChannelListEntry in the array into the buffer. Note that if the ChannelListEntry occurs more than once in the array, the data for each occurrence of the ChannelListEntry is copied.

  Declare an array of 64-bit floating-point values (double) for use with this method.

When you have finished copying the data into the internal buffer of the OlBuffer object, put the OlBuffer object back on the queue for the analog output subsystem using the **AnalogOutputSubsystem.BufferQueue.QueueBuffer** method.

The following example shows how to create an OlBuffer object, fill the internal buffer of this OlBuffer object with 100 samples, and put the OlBuffer object on the analog output subsystem queue:

Visual C#

```
// Allocate a buffer of 100 samples
DacBuffer = new OlBuffer (100, aoutSS);
//Create an array of data
for (int i = 0; i < 100; i++)
   {
      data[i] = i;
   }
// Copy the raw data to the buffer
DacBuffer.PutDataAsRaw (data);
// Queue the buffer for output
aoutSS.BufferQueue.QueueBuffer (DacBuffer);
```

Visual Basic

```
' Allocate a buffer of 100 samples
DacBuffer = New OlBuffer(100, aoutSS)
' Create an array of data
Dim i As Integer
   For i = 0 To 99
      data(i) = i
   Next i
' Copy the raw data to the buffer
DacBuffer.PutDataAsRaw(data)
' Queue the buffer for output
aoutSS.BufferQueue.QueueBuffer(DacBuffer)
```

Getting Information about a Buffer

The DT-Open Layers for .NET Class Library provides the following additional properties for getting information about buffers:

- **OlBuffer.BufferSizeInBytes** – Returns the size, in bytes, of the internal data buffer that is encapsulated by the OlBuffer object.
- **OlBuffer.BufferSizeInSamples** – Returns the size, in samples, of the internal data buffer that is encapsulated by the OlBuffer object.
- **OlBuffer.ChannelListOffset** – Returns the index into the ChannelList that corresponds to the first sample in the internal buffer of the OlBuffer object.
- **OlBuffer.Encoding** – Returns the data encoding for the raw data (binary or twos complement) in the internal buffer of the OlBuffer object.
- **OlBuffer.RawDataFormat** – Returns the format of the raw data (Int16, Uint16, Int32, Float (32-bit float), or Double (64-bit float)) in the internal buffer of the OlBuffer object.
- **OlBuffer.Resolution** – Returns the resolution of the subsystem that is associated with the OlBuffer object.
- **OlBuffer.SampleSizeInBytes** – Returns the size of a sample, in bytes. Typically, each sample requires 2 bytes.

- **OlBuffer.State** property – Returns the state of the OlBuffer object. Valid states are as follows:
  - Idle – The OlBuffer object has been created, but has not been queued to a subsystem.
  - Queued – The OlBuffer object has been queued to a subsystem with **OlBuffer.QueueBuffer**.
  - InProcess – The OlBuffer object has been sent to the device driver for processing. A maximum of eight OlBuffer objects can be inprocess at one time.
  - Completed – For an input operation, the internal buffer of the OlBuffer object has been filled, and the OlBuffer object has not been put back on queue for the subsystem. For an output operation, all the data in the internal buffer of the OlBuffer object has been output, and the OlBuffer object has not been put back on the queue for the subsystem.
  - Released – The internal data buffer of the OlBuffer object has been deallocated by calling **OlBuffer.Dispose**.
- **OlBuffer.ValidSamples** – Returns the number of valid samples in the internal buffer of the OlBuffer object.

  For analog input operations, the **ValidSamples** property is set to the number of samples in the completed buffer under normal circumstances. However, in some cases, like if **Abort** is called in the middle of an operation, **ValidSamples** reflects the number of samples in the buffer when **Abort** was called. In addition, if **Abort** or **Stop** is called, any OlBuffer object whose state is Inprocess will have a **ValidSamples** of 0.

  For analog output operations, **ValidSamples** is always equal to the maximum number of samples that the buffer was allocated to hold.
- **OlBuffer.VoltageRange** – Returns the upper limit and lower limit of the voltage range for the associated subsystem.

Cleaning up Buffers

When you are finished performing continuous analog I/O operations, use can use one of the following methods to clean up the OlBuffer objects:

- **BufferQueue.DequeueBuffer** – Removes and returns the OlBuffer object at the front of the queue.
- **BufferQueue.FreeAllQueuedBuffers** – Removes all OlBuffer objects from the queue and deallocates the internal data buffers that are encapsulated by them.

Starting Subsystems Simultaneously

If supported, you can set up subsystems to start simultaneously. Note that you cannot perform simultaneous startup on subsystems configured for single-value operations unless you are using a simultaneous sampling module.

To determine if a subsystem supports simultaneous start, use the **SupportsSimultaneousStart** property inherited from the SubsystemBase class. If this property returns a value of True, the subsystem can be simultaneously started.

You can synchronize the triggers of subsystems by specifying the same trigger source for each of the subsystems that you want to start simultaneously; ensure that the triggers are wired appropriately to the device.

Use the **SimultaneousStart.AddSubsystem** method to add the subsystems that you want to start simultaneously to the start list. If, later, you want to remove a subsystem from the start list, use the **SimultaneousStart.RemoveSubsystem** method.

To return an array of subsystems that were added to the simultaneous start list, use the **SimultaneousStart.GetSubsystemList** method.

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

Start the subsystems using the **SimultaneousStart.Start** method. When started, both subsystems are triggered simultaneously.

When you are finished with the operations, call the **SimultaneousStart.Clear** method to remove the subsystems from the simultaneous start list.

Auto-Calibrating a Subsystem

Some devices provide a self-calibrating feature, where a specified subsystem performs an auto-zero function. To determine if the specified subsystem supports this capability, use the **AnalogInputSubsystem.SupportsAutoCalibrate** property. If this property returns a value of True, the subsystem can be calibrated through software.

To calibrate the subsystem in software, call the **AutoCalibrate** method. Ensure that the subsystem is not running when you call this method, or an error is returned.

Handling Events

DT-Open Layers devices notify your application of buffer movement and other system activities by raising events.

Delegates, which behave like function pointers, are provided to handle these events. Each delegate has a specific signature and holds a reference to a method that matches its signature. When an event occurs, the appropriate method (with the matching signature) is called.

The following example shows the declaration for the **BufferDoneHandler** delegate provided by the DT-Open Layers for .NET Class Library:

```
[C#]
// BufferDoneHandler is the delegate for the BufferDoneEvent event.
// BufferDoneEventArgs is the class that holds event data for
// BufferDoneEvent.
// It derives from the base class for event data, GeneralEventArgs.

public delegate void BufferDoneHandler(object sender,
   BufferDoneEventArgs eventArgs);

[Visual Basic]
' BufferDoneHandler is the delegate for the BufferDoneEvent event.
' BufferDoneEventArgs is the class that holds event data for
' BufferDoneEvent.
' It derives from the base class for event data, GeneralEventArgs.
Public Delegate Sub BufferDoneHandler(sender As Object,
   eventArgs As BufferDoneEventArgs)
```

As you can see, the syntax of the delegate is similar to that of a method declaration; however, the delegate keyword informs the compiler that **BufferDoneHandler** is a delegate type. By convention, event delegates in the .NET Framework have two parameters, the source that raised the event and the data for the event.

To handle events, you must define a method that matches the delegate; this is the event handling method that is called when the appropriate event is raised. In the following example, the event handling method called MyBufferDone matches the signature of the **BufferDoneHandler** delegate and is called when the event BufferDoneEvent is raised:

Visual C#

```
// MyBufferDone has the same signature as BufferDoneHandler.
public void MyBufferDone (object sender,
   BufferDoneEventArgs eventArgs);
{
//Add you own code here.
}
```

Visual Basic
```
' MyBufferDone has the same signature as BufferDoneHandler.
   Public Sub MyBufferDone(sender As Object,
      eventArgs As BufferDoneEventArgs)
' Add you own code here
   End Sub
```

Note: To ensure that events are handled in the main application, use the InvokeRequired method. Refer to your .NET documentation for more information on this method.

Lastly, you must associate the event and event handling method with the appropriate subsystem. The following example shows how to associate the event BufferDoneEvent and the MyBufferDoneHandler event handler to the analog input subsystem called *ainSS*:

Visual C#
```
// Associate the event BufferDoneEvent and the event handling method
// MyBufferDone with the analog input subsystem ainSS.
ainSS.BufferDoneEvent += new BufferDoneHandler (MyBufferDoneHandler);
```

Note: In C#, when you want to disable receiving events, use the - = operator instead of the += operator. See your .NET documentation for more information about events and delegates.

Visual Basic
```
' Associate the event BufferDoneEvent and the event handling method
' MyBufferDone with the analog input subsystem ainSS.
AddHandler ainSS.BufferDoneEvent, Address of MyBufferDoneHandler
```

Note: In Visual Basic, when you want to disable receiving events, use the RemoveHandler statement instead of the AddHandler statement. See your .NET documentation for more information about events and delegates.

The following subsections describe the events and delegates that are provided in the DT-Open Layers for .NET Class Library. Refer to the examples provided with this software package to see how to incorporate event handling code into your program.

BufferDoneEvent

For input operations, the event BufferDoneEvent is raised when the internal data buffer of the OlBuffer object has been filled with post-trigger data. For output operations, this event is raised when all the data in the internal data buffer of the OlBuffer object has been output.

If you stop an analog I/O operation, the event BufferDoneEvent is generated for the current OlBuffer object and for up to eight inprocess OlBuffer objects before a QueueStoppedEvent event occurs.

Use the **BufferDoneHandler** delegate with BufferDoneEvent. When BufferDoneEvent is raised, the subsystem that raised the event, the time stamp of when the event occurred, and the completed OlBuffer object are passed in the BufferDoneEventArgs argument of the user-defined method that matches the signature of the **BufferDoneHandler** delegate.

You can add your own code to the event handling method to manage the data in the buffer or perform other operations as required by your application. Refer to page 325 for more information on handling input buffers; refer to page 327 for more information on handling output buffers.

Note: If your program is running under a heavy CPU load, and if the **AnalogInputSubsystem.SynchronousBufferDone** property is set to False (the default condition), .NET may call your BufferDoneEvent delegates out of order under some circumstances. To avoid this problem, it is recommended that you set the **AnalogInputSubsystem.SynchronousBufferDone** property to True, so that all BufferDoneEvent events are executed synchronously in a single worker thread instead of asynchronously using a separate thread for each event.

The following is an example of an event handling routine called HandleBufferDone that handles the event BufferDoneEvent. This event handler converts the data from the internal buffer of the OlBuffer object into sensor values and copies the data into a user-dimensioned array called *buf.* The first 10 samples are printed to the form, and the OlBuffer object is put back on the queue for the subsystem:

Visual C#

```
public void HandleBufferDone (object sender,
   BufferDoneEventArgs bufferDoneData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke( new BufferDoneHandler (HandleBufferDone),
              new object[] {sender, bufferDoneData });
         }
         else
         {
            // Get the data as sensor values
            double[] buf = olBuffer.GetDataAsSensor();
              //requeue the completed buffer
               ainSS.BufferQueue.QueueBuffer (olBuffer);
            // Output the first 10 samples to the user form
            for (int i=0; i<10; ++i)
            {
               OlBufferDataTable.Rows[i][0] = buf[i];
            }
         }
      }
```

<u>*Visual Basic*</u>

```
Public Sub HandleBufferDone(ByVal sender As Object,
  ByVal bufferDoneData As BufferDoneEventArgs)
         If Me.InvokeRequired Then
            Me.Invoke(New BufferDoneHandler(
            AddressOf HandleBufferDone), New Object()
              {sender, bufferDoneData})
         Else
            ' Get the data as sensor values
            Dim buf As Double() = olBuffer.GetDataAsSensor()
            ' requeue the completed buffer
            ainSS.BufferQueue.QueueBuffer(olBuffer)
            End If
            ' Output the first 10 samples to the user form
            Dim i As Integer
            While i < 10
               OlBufferDataTable.Rows(i)(0) = buf(i)
               i += 1
            End While
         End If
End Sub 'HandleBufferDone
```

PreTriggerBufferDoneEvent

The event PreTriggerBufferDone is raised when the internal buffer of the OlBuffer object is filled with pre-trigger data (for an input operation only). Refer to page 322 for more information about buffers.

Use the **PreTriggerBufferDoneHandler** delegate with PreTriggerBufferDoneEvent. When PreTriggerBufferDoneEvent is raised, the subsystem that raised the event, the time stamp of when the event occurred, and the completed OlBuffer object are passed in the BufferDoneEventArgs argument of the user-defined method that matches the signature of the **PreTriggerBufferDoneHandler** delegate.

You can add your own code to the event handling method to manage the data in the buffer or perform other operations as required by your application. Refer to page 325 for more information on handling input buffers.

The following is an example of an event handling routine called HandlePreTriggerBufferDone that handles the event PreTriggerBufferDoneEvent. This event handler converts the data from the internal buffer of the OlBuffer object into voltage values and copies the data into a user-dimensioned array called *buf.* The first 10 samples are printed to the form, and the OlBuffer object is put back on the queue for the subsystem:

Visual C#

```
public void HandlePreTriggerBufferDone (object
   sender, BufferDoneEventArgs bufferDoneData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke( new PreTriggerBufferDoneHandler (
               HandlePreTriggerBufferDone), new object[] { sender,
                 bufferDoneData});
         }
         else
         {
            // Get the data as voltages
            double[] buf = olBuffer.GetDataAsVolts();

              //requeue the completed buffer
               ainSS.BufferQueue.QueueBuffer (olBuffer);

            // Output the first 10 samples to the user form
            for (int i=0; i<10; ++i)
            {
               OlBufferDataTable.Rows[i][0] = buf[i];
            }
         }
      }
```

Visual Basic

```
Public Sub HandlePreTriggerBufferDone(ByVal sender As Object,
  ByVal bufferDoneData As BufferDoneEventArgs)
         If Me.InvokeRequired Then
            Me.Invoke(New PreTriggerBufferDoneHandler(
             AddressOf HandlePreTriggerBufferDone),
             New Object() {sender, bufferDoneData})
         Else
            ' Get the data as voltages
            Dim buf As Double() = olBuffer.GetDataAsVolts()
            ' requeue the completed buffer
            ainSS.BufferQueue.QueueBuffer(olBuffer)
            End If
            ' Output the first 10 samples to the user form
            Dim i As Integer
            While i < 10
               OlBufferDataTable.Rows(i)(0) = buf(i)
               i += 1
            End While
         End If
End Sub 'HandleBufferDone
```

QueueStoppedEvent

A QueueStoppedEvent is raised when **Stop** or **Abort** is called for a continuous analog input or analog output operation.

Note: The event BufferDoneEvent is generated for the current OlBuffer object and for up to eight inprocess OlBuffer objects before a QueueStoppedEvent event occurs.

Use the **QueueStoppedHandler** delegate with QueueStoppedEvent. When QueueStoppedEvent is raised, the subsystem that raised the event and the time stamp of when the event occurred are passed in the GeneralEventArgs argument of the user-defined method that matches the signature of the **QueueStoppedHandler** delegate.

The following is an example of an event handling routine called HandleQueueStopped that handles the event QueueStoppedEvent. This event handler displays a message on the form that indicates which subsystem raised the QueueStoppedEvent and at what time the event occurred:

Visual C#

```
public void HandleQueueStopped (object sender,
   GeneralEventArgs eventData)
   {
      if (this.InvokeRequired)
         {
            this.Invoke(new QueueStoppedHandler(HandleQueueStopped)
               ,new object[] { sender, eventData });
         }
         else
         {
            string msg = String.Format ("Queue Stopped received on
               subsystem {0} element {1} at time {2}",
                eventData.Subsystem, eventData.Subsystem.Element,
                eventData.DateTime.ToString("T"));

            statusBarPanel.Text = msg;
         }
   }
```

Visual Basic

```
Public Sub HandleQueueStopped(ByVal sender As Object,
  ByVal eventData As GeneralEventArgs)
         If Me.InvokeRequired Then
            Me.Invoke(New QueueStoppedHandler(
               AddressOf HandleQueueStopped),
                New Object() {sender, eventData})
         Else
            Dim msg As String = String.Format(
              "Queue Stopped received on subsystem {0} element {1}
               at time {2}", eventData.Subsystem,
               eventData.Subsystem.Element,
               eventData.DateTime.ToString("T"))
             statusBarPanel.Text = msg
         End If
End Sub 'HandleQueueStopped
```

IOCompleteEvent

For analog input operations that use a reference trigger whose trigger type is something other than software (none), the event IOCompleteEvent is raised when the last post-trigger sample is copied into the user buffer. This event includes the total number of samples per channel that were acquired from the time acquisition was started (with the start trigger) to the last post-trigger sample. For example, a value of 100 indicates that a total of 100 samples (samples 0 to 99) were acquired. In some cases, this message is generated well before the events BufferDoneEvent are generated. You can determine when the reference trigger occurred and the number of pre-trigger samples that were acquired by subtracting the post trigger scan count, described on page 322, from the total number of samples that were acquired. Devices that do not support a reference trigger will never receive this event for analog input operations.

For analog output operations, the event IOCompleteEvent is raised when the last data point has been output from an analog output channel. In some cases, this event is raised well after the data is transferred from the buffer (and, therefore, well after BufferDoneEvent and QueueDoneEvents are raised). Refer to page 322 for more information on buffers.

Use the **IOCompleteHandler** delegate with IOCompleteEvent. When IOCompleteEvent is raised, the subsystem that raised the event and the time stamp of when the event occurred are passed in the IOCompleteEventsArgs argument of the user-defined method that matches the signature of the **IOCompleteHandler** delegate.

You can add your own code to the event handling method to deal with this event as needed.

The following is an example of an event handling routine called HandleIOComplete that handles the event IOCompleteEvent. This event handler displays a message on the form that indicates which subsystem raised the IOCompleteEvent and at what time the event occurred:

Visual C#
```
public void HandleIOComplete (object sender,
  IOCompleteEventArgs eventData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke( new IOCompleteHandler (HandleIOComplete),
              new object[] {sender, eventData });
         }


         else
         {
            string msg = String.Format ("IOComplete received on
             subsystem {0} at time {1}", eventData.Subsystem,
               eventData.DateTime.ToString("T"));
            statusBarPanel.Text = msg;
         }
      }
```

Visual Basic
```
Public Sub HandleIOComplete(ByVal sender As Object,
  ByVal eventData As IOCompleteEventArgs)
         If Me.InvokeRequired Then
           Me.Invoke(New IOCompleteHandler(
            AddressOf HandleIOComplete),
             New Object() {sender, eventData})
         Else
            Dim msg As String = String.Format(
              "IOComplete received on subsystem {0} at time {1}",
               eventData.Subsystem, eventData.DateTime.ToString("T"))
                statusBarPanel.Text = msg
         End If
End Sub 'HandleIOComplete
```

QueueDoneEvent

The event QueueDoneEvent is raised when no OlBuffer objects are available on the queue and the operation stops. Refer to page 322 for more information.

Use the **QueueDoneHandler** delegate with QueueDoneEvent. When QueueDoneEvent is raised, the subsystem that generated the event and the time stamp of when the event occurred are passed in the GeneralEventArgs argument of the user-defined method that matches the signature of the **QueueDoneHandler** delegate.

The following is an example of an event handling routine called HandleQueueDone that handles the event QueueDoneEvent. This event handler displays a message on the form that indicates which subsystem raised the QueueDoneEvent and at what time the event occurred:

Visual C#

```
public void HandleQueueDone (object sender,
  GeneralEventArgs eventData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke(new QueueDoneHandler(HandleQueueDone),
              new object[] { sender, eventData });
         }
         else
         {
            string msg = String.Format ("Queue Done received on {0}
                element {1} at time {2}", eventData.Subsystem,
               eventData.Subsystem.Element,
               eventData.DateTime.ToString("T"));
            statusBarPanel.Text = msg;
         }
      }
```

Visual Basic

```
Public Sub HandleQueueDone(ByVal sender As Object,
   ByVal eventData As GeneralEventArgs)
         If Me.InvokeRequired Then
           Me.Invoke(New QueueDoneHandler(AddressOf HandleQueueDone),
             New Object()
            {sender, eventData})
         Else
           Dim msg As String = String.Format(
           "Queue Done received on {0} element {1} at time {2}",
            eventData.Subsystem, eventData.Subsystem.Element,
             eventData.DateTime.ToString("T"))
             statusBarPanel.Text = msg
         End If
End Sub 'HandleQueueDone
```

DriverRunTimeErrorEvent

The DriverRunTimeErrorEvent occurs when the device driver detects one of the following error conditions:

- FifoOverflow – The driver could not read data from the device FIFO (or Windows USB FIFO) fast enough, resulting in a FIFO overflow condition. To deal with this error, increase the size of the buffers, slow down the sampling rate, or stop other CPU-intensive running programs.

Note: By setting the **AnalogInputSubsystem.StopOnError** property, you can determine how the subsystem operates if an overrun occurs. If **StopOnError** is True, the subsystem will automatically stop when an overrun is detected. If **StopOnError** is False, the subsystem will continue running if an overrun is detected.

- FifoUnderflow – The driver could not write data to the device FIFO (or Windows USB FIFO) fast enough, resulting in FIFO underflow condition. To deal with this error, increase the size of buffers, slow down the sampling rate, or stop other CPU-intensive running programs.

Note: By setting the **AnalogOutputSubsystem.StopOnError** property, you can determine how the subsystem operates if an underrun occurs. If **StopOnError** is True, the subsystem will automatically stop when an underrun is detected. If **StopOnError** is False, the subsystem will continue running if an underrun is detected.

- DeviceOverClocked – The A/D clock (usually external clock) is running too fast on the device. To deal with this error, slow down the A/D clock.
- TriggerError – The driver detected a trigger on the device but did not act on it.
- DeviceError – Generated by the driver due to a USB bus or hardware problem. To deal with this error, stop connecting/disconnecting USB devices while streaming data to them.

Use the **DriverRunTimeErrorEventHandler** delegate with DriverRunTimeErrorEvent. When DriverRunTimeErrorEvent is raised, the subsystem that generated the event, the time stamp of when the event occurred, the error code, and the error code descriptor are passed in the DriverRunTimeErrorEventArgs argument of the user-defined method that matches the signature of the **DriverRunTimeErrorEventHandler** delegate.

The following is an example of an event handling routine called HandleDriverRunTimeErrorEvent that handles the event DriverRunTimeErrorEvent. This event handler displays a message on the form that indicates what error occurred, which subsystem raised the DriverRunTimeErrorEvent, and at what time the event occurred:

Visual C#

```
public void HandleDriverRunTimeErrorEvent (object sender,
  DriverRunTimeErrorEventArgs eventData)
      {
         if (this.InvokeRequired)
         {
            this.Invoke(new
               DriverRunTimeErrorEventHandler(
                HandleDriverRunTimeErrorEvent),
               new object[] { sender, eventData });
         }
         else
         {
            string msg = String.Format ("Error: {0}
               Occurred on subsystem {1} element {2} at time {3}",
                eventData.Message, eventData.Subsystem,
                eventData.Subsystem.Element,
                 eventData.DateTime.ToString("T"));
            MessageBox.Show (msg, "Error");
         }
      }
```

Visual Basic

```
Public Sub HandleDriverRunTimeErrorEvent(ByVal sender As Object,
   ByVal eventData As DriverRunTimeErrorEventArgs)
         If Me.InvokeRequired Then
            Me.Invoke(New DriverRunTimeErrorEventHandler(
               AddressOf HandleDriverRunTimeErrorEvent),
                New Object() {sender, eventData})
         Else
            Dim msg As String = String.Format(
             "Error: {0} Occurred on subsystem {1}
              element {2} at time {3}", eventData.Message,
              eventData.Subsystem, eventData.Subsystem.Element,
              eventData.DateTime.ToString("T"))
            MessageBox.Show(msg, "Error")
         End If
End Sub 'HandleDriverRunTimeErrorEvent
```

GeneralFailureEvent

The event GeneralFailureEvent is raised when a general library failure occurs.

Use the **GeneralFailureHandler** delegate with GeneralFailureEvent. When GeneralFailureEvent is raised, the subsystem that raised the event and the time stamp of when the event occurred are passed in the GeneralEventArgs argument of the user-defined method that matches the signature of the **GeneralFailureHandler** delegate.

You can add your own code to the handler to deal with this event as needed.

DeviceRemovedEvent

The event DeviceRemovedEvent is raised when a device is removed from your system while your application is running.

Use the **DeviceRemovedHandler** delegate with DeviceRemovedEvent. When DeviceRemovedEvent is raised, the subsystem that raised the event and the time stamp of when the event occurred are passed in the GeneralEventArgs argument of the user-defined method that matches the signature of the **DeviceRemovedHandler** delegate.

You can add your own code to the event handling method to deal with this event as needed.

Handling Errors

Errors are generated by the DT-Open Layers .NET Class Library as OlException objects. Each OlException object contains an OlError object, which contains the error code and its description. Your program should handle exceptions as they occur, performing the appropriate actions to deal with any errors that arise.

Refer to Appendix A for a list of error codes and messages. These values are defined as enumerations that are accessible using the **OlException.ErrorCode** and **OlException.Message** properties. If you want to determine which subsystem generated the error, use the **OlException.Subsystem** property.

The following example shows how to catch exceptions in your program; this example the error message is printed to text field on the form:

Visual C#

```
catch (OlException ex)
   {
      string err = ex.Message;
      statusBarPanel.Text = err;
      return;
   }
```

Visual Basic

```
Catch ex As OlException
  Dim err As String = ex.Message
  statusBarPanel.Text = err
  Return
```

Cleaning Up Operations

When you are finished performing data acquisition operations, clean up the memory and resources that were used by the operation by doing the following:

1. Release the simultaneous start list, if used, using the **SimultaneousStart.Clear** method.
2. Deallocate any buffers, if used. Refer to page 329 for more information.
3. Release the subsystem connection to the device using the **Dispose** method within the appropriate subsystem class.
4. Release the Device object using the **Device.Dispose** method.

5

Programming Flowcharts for the OpenLayers.Base Namespace

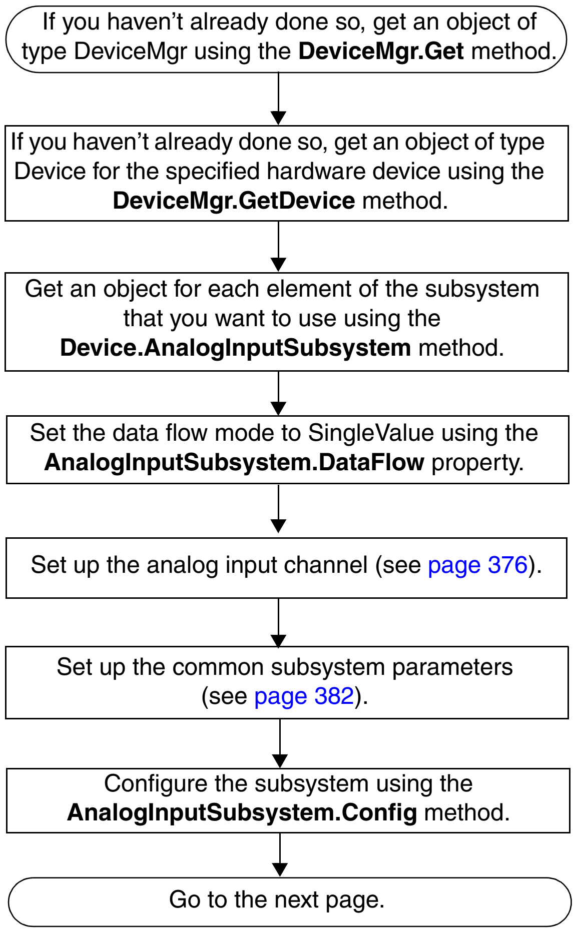
The flowcharts presented in the remainder of this chapter show how to perform typical input/output operations.

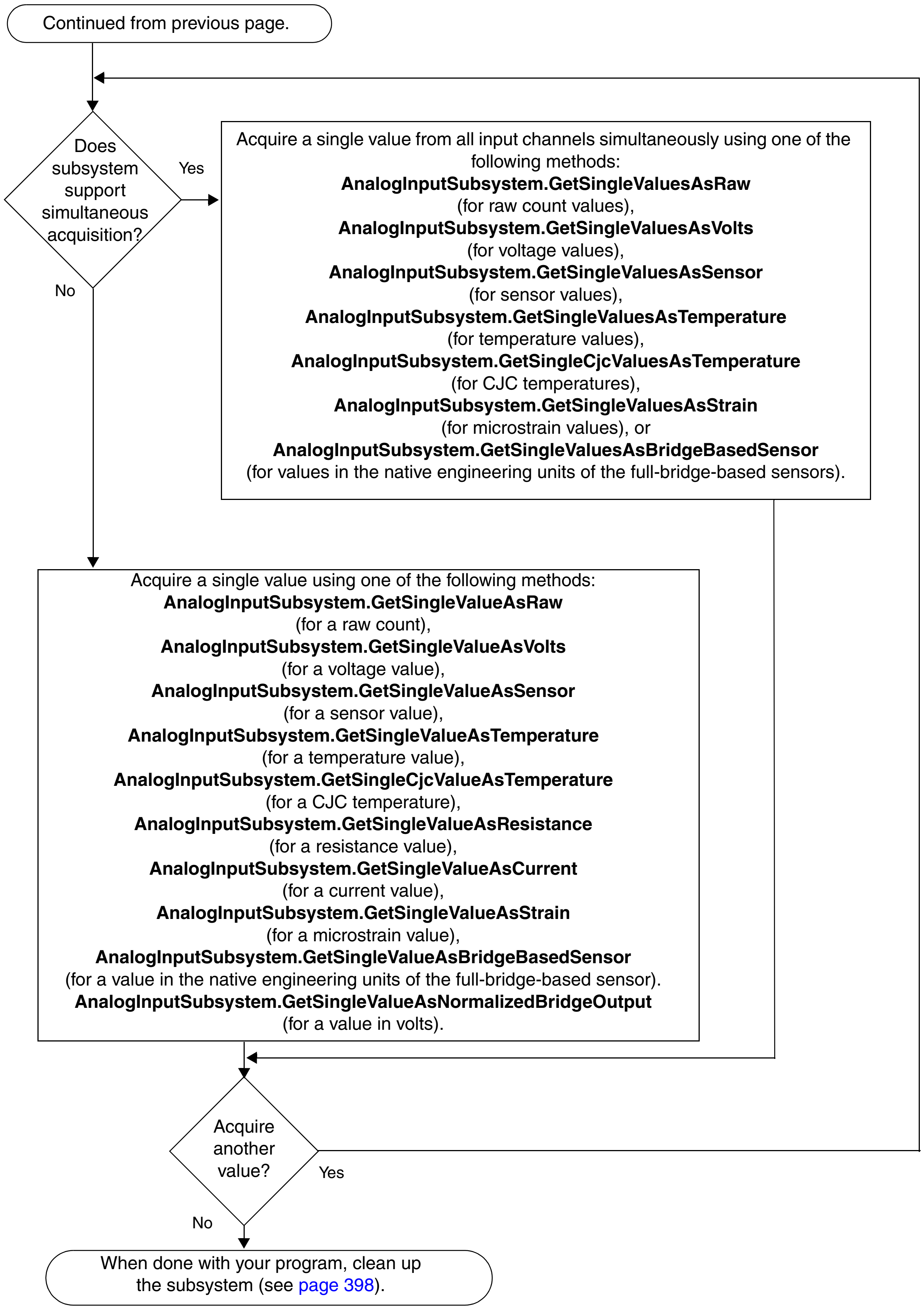
Note: Depending on your device, some of the settings may not be programmable. Refer to your device documentation for details.

Although the flowcharts do not show error checking, it is recommended that you add exception handling to your program.

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.

Single-Value Analog Input Operations


If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.
Get an object for each element of the subsystem that you want to use using the **Device.AnalogInputSubsystem** method.
Set the data flow mode to SingleValue using the **AnalogInputSubsystem.DataFlow** property.
Set up the analog input channel (see page 376).
Set up the common subsystem parameters (see page 382).
Configure the subsystem using the **AnalogInputSubsystem.Config** method.
Go to the next page.

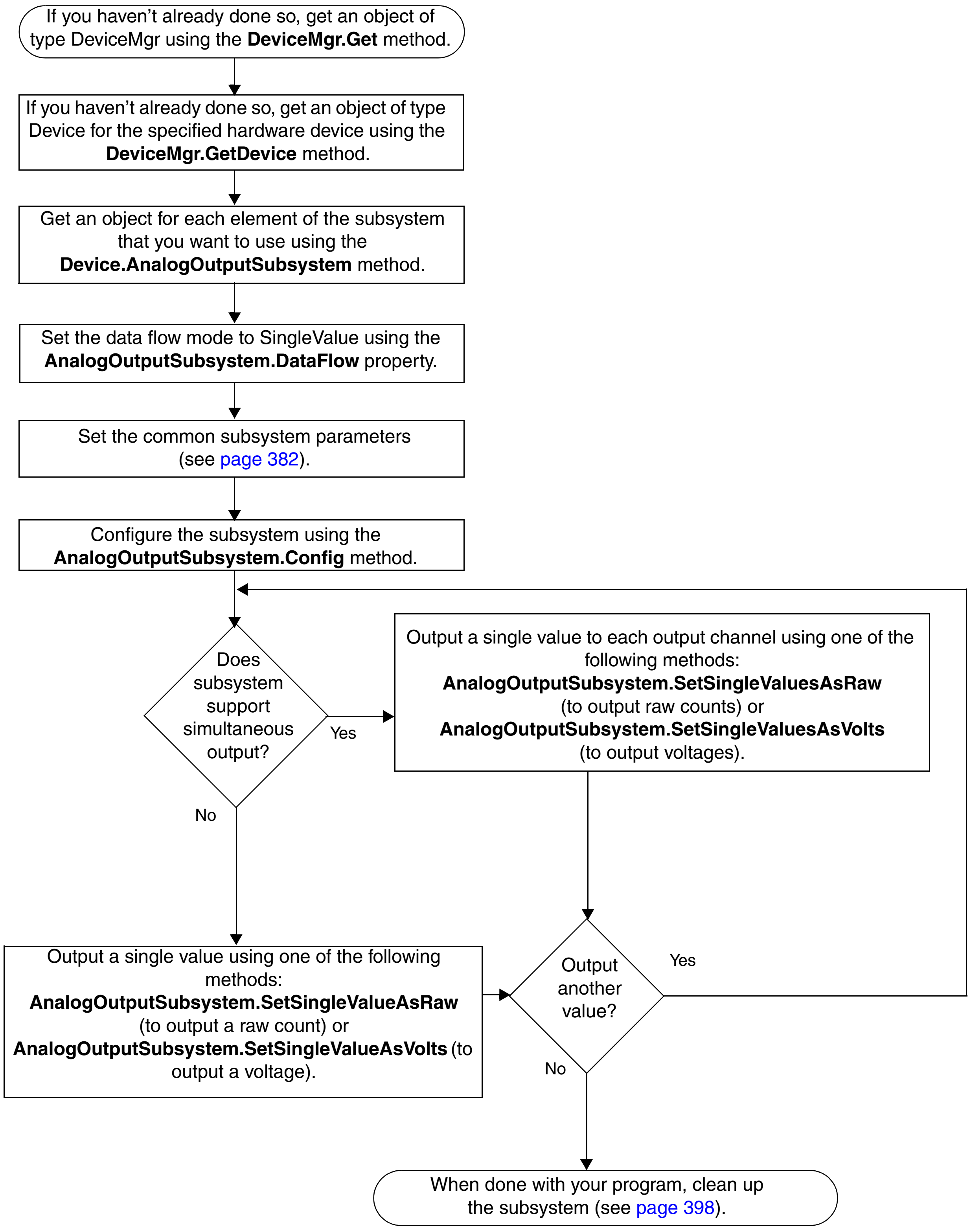

Continued from previous page.
Does subsystem support simultaneous acquisition?
Yes
No
Acquire another value?
Yes
No

Acquire a single value from all input channels simultaneously using one of the following methods:
AnalogInputSubsystem.GetSingleValuesAsRaw
(for raw count values),
AnalogInputSubsystem.GetSingleValuesAsVolts
(for voltage values),
AnalogInputSubsystem.GetSingleValuesAsSensor
(for sensor values),
AnalogInputSubsystem.GetSingleValuesAsTemperature
(for temperature values),
AnalogInputSubsystem.GetSingleCjcValuesAsTemperature
(for CJC temperatures),
AnalogInputSubsystem.GetSingleValuesAsStrain
(for microstrain values), or
AnalogInputSubsystem.GetSingleValuesAsBridgeBasedSensor
(for values in the native engineering units of the full-bridge-based sensors).

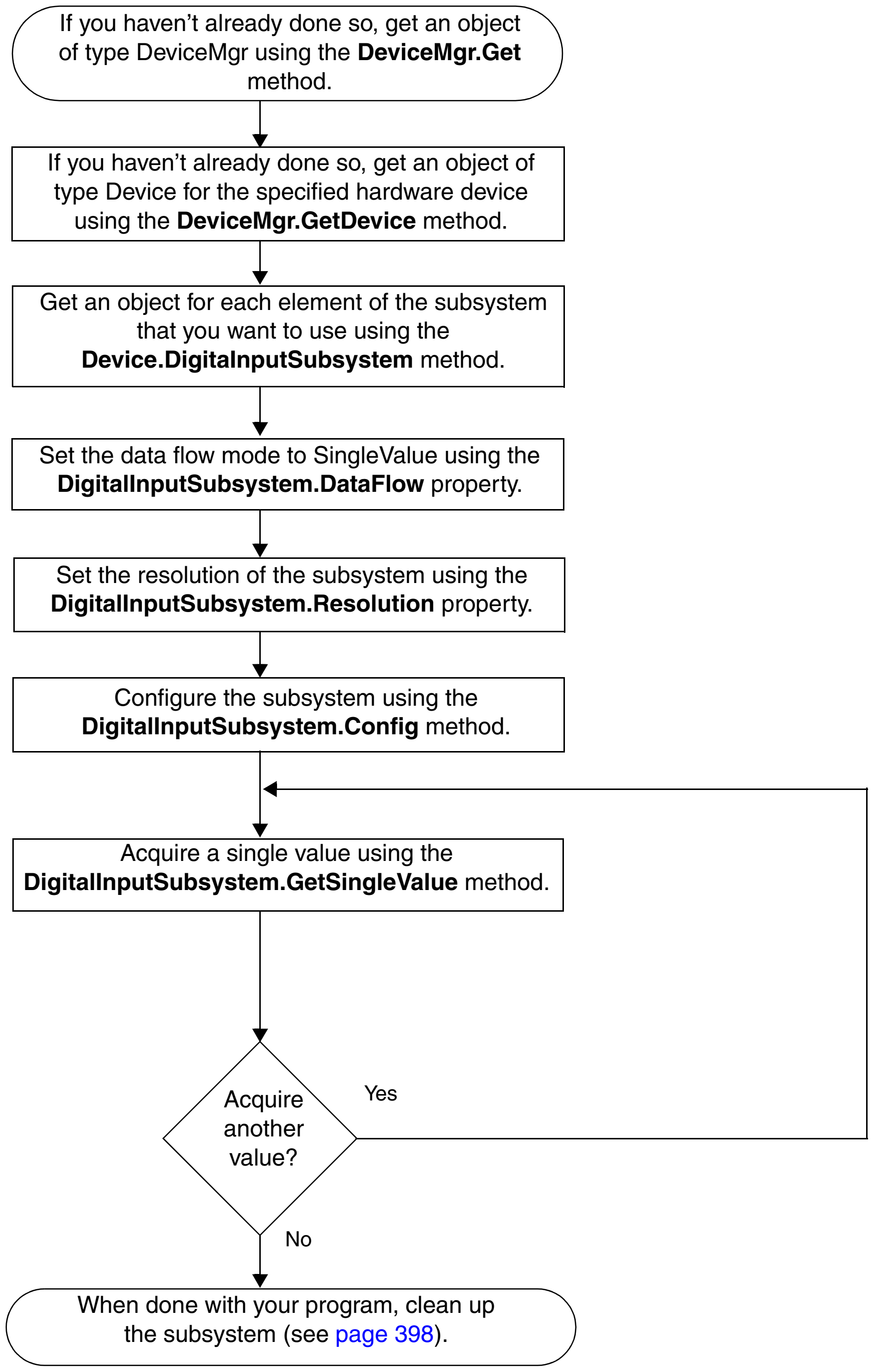
Acquire a single value using one of the following methods:
AnalogInputSubsystem.GetSingleValueAsRaw
(for a raw count),
AnalogInputSubsystem.GetSingleValueAsVolts
(for a voltage value),
AnalogInputSubsystem.GetSingleValueAsSensor
(for a sensor value),
AnalogInputSubsystem.GetSingleValueAsTemperature
(for a temperature value),
AnalogInputSubsystem.GetSingleCjcValueAsTemperature
(for a CJC temperature),
AnalogInputSubsystem.GetSingleValueAsResistance
(for a resistance value),
AnalogInputSubsystem.GetSingleValueAsCurrent
(for a current value),
AnalogInputSubsystem.GetSingleValueAsStrain
(for a microstrain value),
AnalogInputSubsystem.GetSingleValueAsBridgeBasedSensor
(for a value in the native engineering units of the full-bridge-based sensor).
AnalogInputSubsystem.GetSingleValueAsNormalizedBridgeOutput
(for a value in volts).

When done with your program, clean up the subsystem (see page 398).

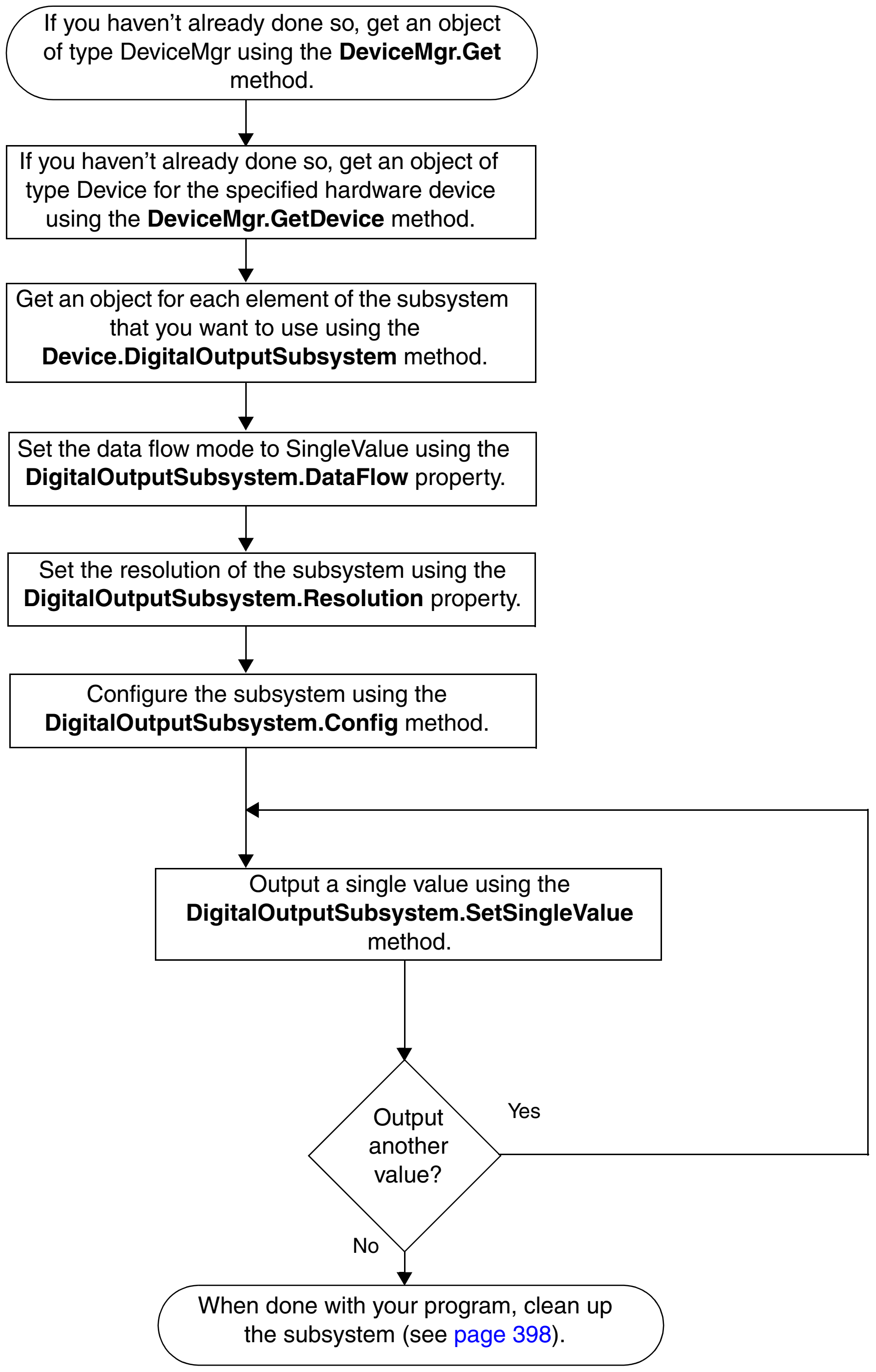
Single-Value Analog Output Operations


If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.
Get an object for each element of the subsystem that you want to use using the **Device.AnalogOutputSubsystem** method.
Set the data flow mode to SingleValue using the **AnalogOutputSubsystem.DataFlow** property.
Set the common subsystem parameters (see page 382).
Configure the subsystem using the **AnalogOutputSubsystem.Config** method.
Does subsystem support simultaneous output?
Yes
Output a single value to each output channel using one of the following methods: **AnalogOutputSubsystem.SetSingleValuesAsRaw** (to output raw counts) or **AnalogOutputSubsystem.SetSingleValuesAsVolts** (to output voltages).
No
Output a single value using one of the following methods: **AnalogOutputSubsystem.SetSingleValueAsRaw** (to output a raw count) or **AnalogOutputSubsystem.SetSingleValueAsVolts** (to output a voltage).
Output another value?
Yes
No
When done with your program, clean up the subsystem (see page 398).

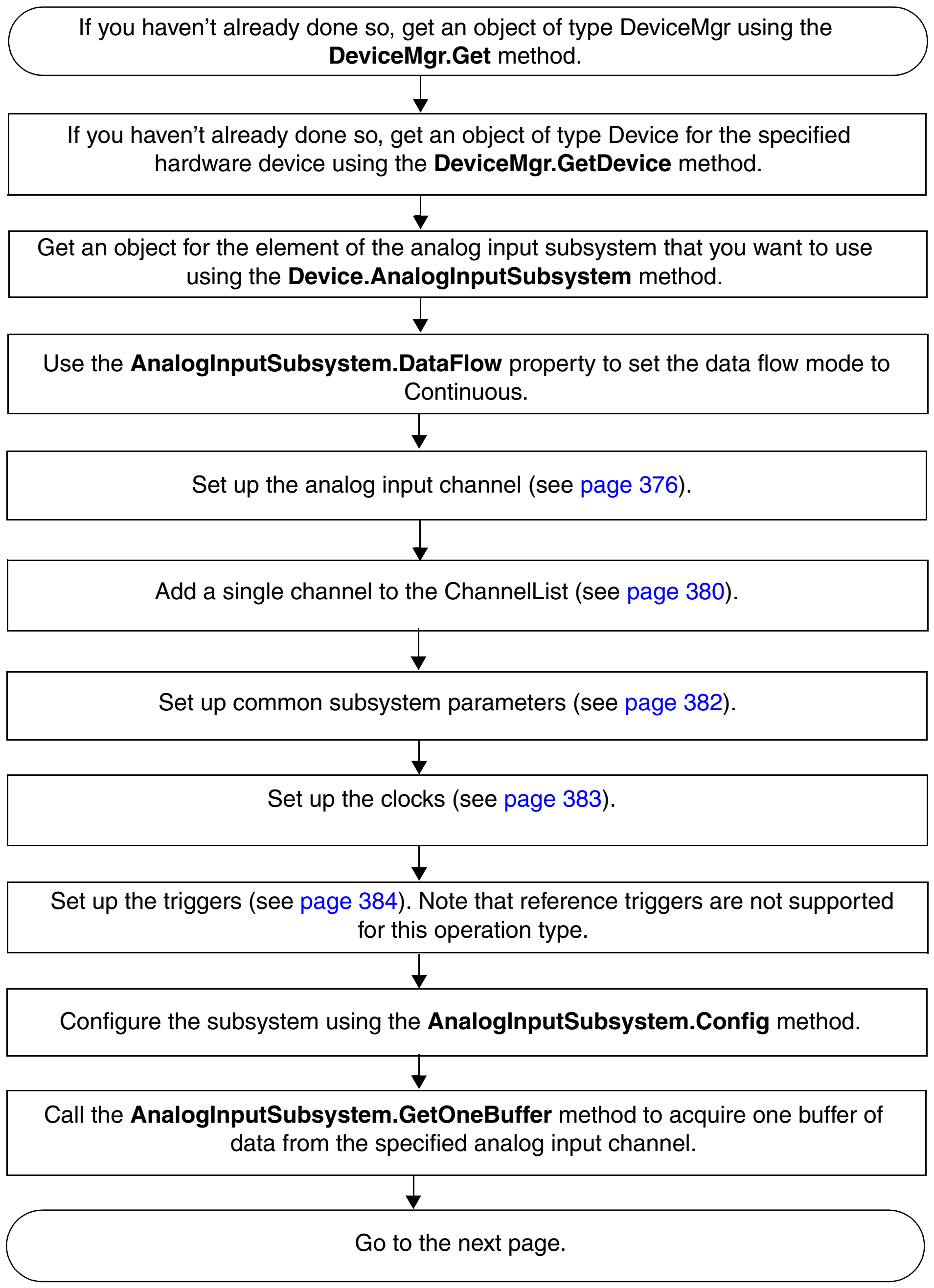
Single-Value Digital Input Operations


If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.
Get an object for each element of the subsystem that you want to use using the **Device.DigitalInputSubsystem** method.
Set the data flow mode to SingleValue using the **DigitalInputSubsystem.DataFlow** property.
Set the resolution of the subsystem using the **DigitalInputSubsystem.Resolution** property.
Configure the subsystem using the **DigitalInputSubsystem.Config** method.
Acquire a single value using the **DigitalInputSubsystem.GetSingleValue** method.
Acquire another value?
Yes
No
When done with your program, clean up the subsystem (see page 398).

Single-Value Digital Output Operations


If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.
Get an object for each element of the subsystem that you want to use using the **Device.DigitalOutputSubsystem** method.
Set the data flow mode to SingleValue using the **DigitalOutputSubsystem.DataFlow** property.
Set the resolution of the subsystem using the **DigitalOutputSubsystem.Resolution** property.
Configure the subsystem using the **DigitalOutputSubsystem.Config** method.
Output a single value using the **DigitalOutputSubsystem.SetSingleValue** method.
Output another value?
Yes
No
When done with your program, clean up the subsystem (see page 398).

Continuous Analog Input Operations - One Buffer



If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.

If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.

Get an object for the element of the analog input subsystem that you want to use using the **Device.AnalogInputSubsystem** method.

Use the **AnalogInputSubsystem.DataFlow** property to set the data flow mode to Continuous.

Set up the analog input channel (see page 376).

Add a single channel to the ChannelList (see page 380).

Set up common subsystem parameters (see page 382).

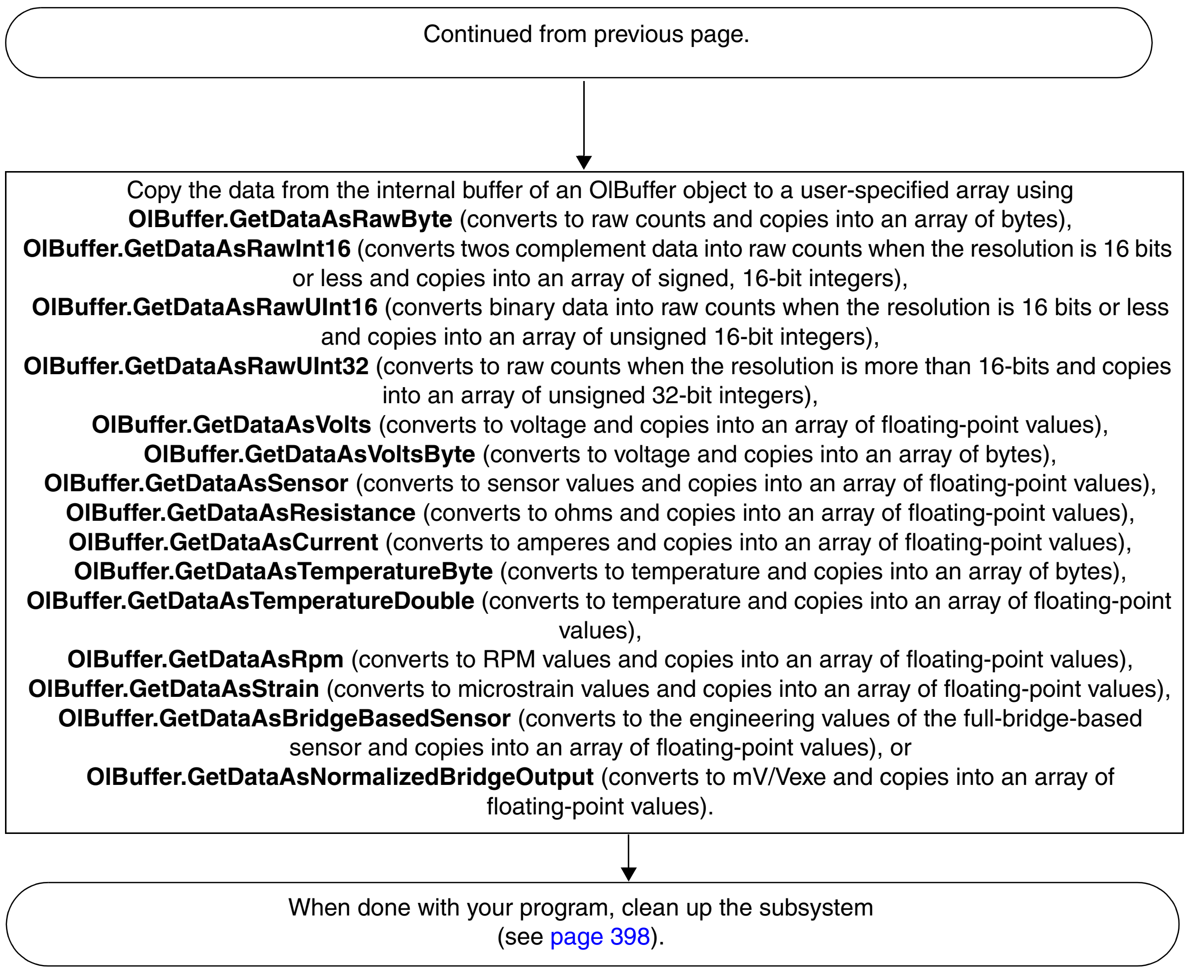
Set up the clocks (see page 383).

Set up the triggers (see page 384). Note that reference triggers are not supported for this operation type.

Configure the subsystem using the **AnalogInputSubsystem.Config** method.

Call the **AnalogInputSubsystem.GetOneBuffer** method to acquire one buffer of data from the specified analog input channel.

Go to the next page.

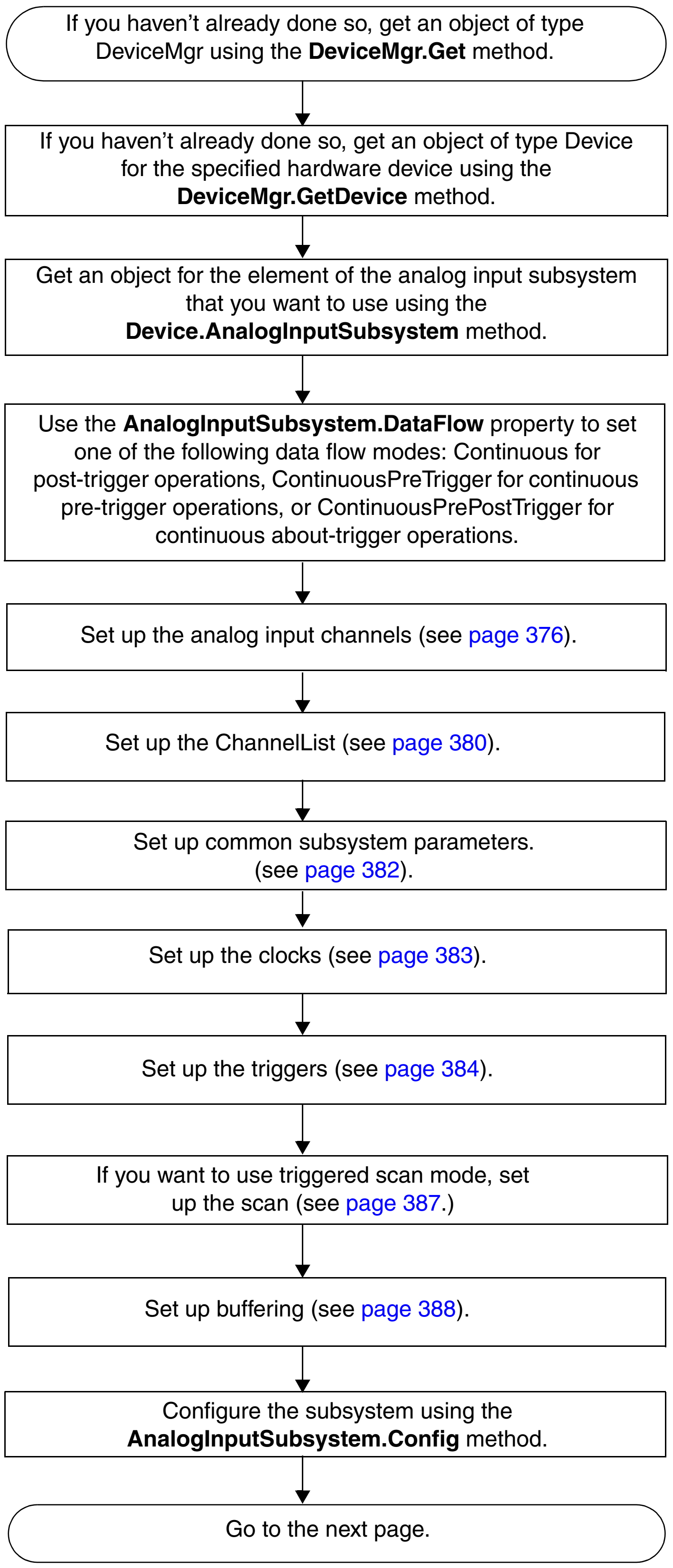


Continued from previous page.

Copy the data from the internal buffer of an OlBuffer object to a user-specified array using **OlBuffer.GetDataAsRawByte** (converts to raw counts and copies into an array of bytes), **OlBuffer.GetDataAsRawInt16** (converts twos complement data into raw counts when the resolution is 16 bits or less and copies into an array of signed, 16-bit integers), **OlBuffer.GetDataAsRawUInt16** (converts binary data into raw counts when the resolution is 16 bits or less and copies into an array of unsigned 16-bit integers), **OlBuffer.GetDataAsRawUInt32** (converts to raw counts when the resolution is more than 16-bits and copies into an array of unsigned 32-bit integers), **OlBuffer.GetDataAsVolts** (converts to voltage and copies into an array of floating-point values), **OlBuffer.GetDataAsVoltsByte** (converts to voltage and copies into an array of bytes), **OlBuffer.GetDataAsSensor** (converts to sensor values and copies into an array of floating-point values), **OlBuffer.GetDataAsResistance** (converts to ohms and copies into an array of floating-point values), **OlBuffer.GetDataAsCurrent** (converts to amperes and copies into an array of floating-point values), **OlBuffer.GetDataAsTemperatureByte** (converts to temperature and copies into an array of bytes), **OlBuffer.GetDataAsTemperatureDouble** (converts to temperature and copies into an array of floating-point values), **OlBuffer.GetDataAsRpm** (converts to RPM values and copies into an array of floating-point values), **OlBuffer.GetDataAsStrain** (converts to microstrain values and copies into an array of floating-point values), **OlBuffer.GetDataAsBridgeBasedSensor** (converts to the engineering values of the full-bridge-based sensor and copies into an array of floating-point values), or **OlBuffer.GetDataAsNormalizedBridgeOutput** (converts to mV/Vexe and copies into an array of floating-point values).

When done with your program, clean up the subsystem (see page 398).

Continuous Analog Input Operations - Multiple Buffers



If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.

If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.

Get an object for the element of the analog input subsystem that you want to use using the **Device.AnalogInputSubsystem** method.

Use the **AnalogInputSubsystem.DataFlow** property to set one of the following data flow modes: Continuous for post-trigger operations, ContinuousPreTrigger for continuous pre-trigger operations, or ContinuousPrePostTrigger for continuous about-trigger operations.

Set up the analog input channels (see page 376).

Set up the ChannelList (see page 380).

Set up common subsystem parameters. (see page 382).

Set up the clocks (see page 383).

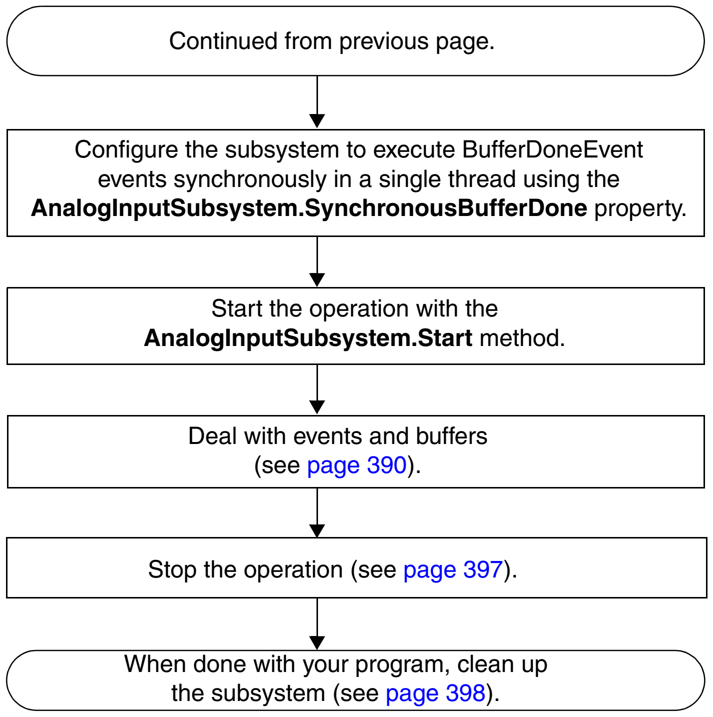
Set up the triggers (see page 384).

If you want to use triggered scan mode, set up the scan (see page 387.)

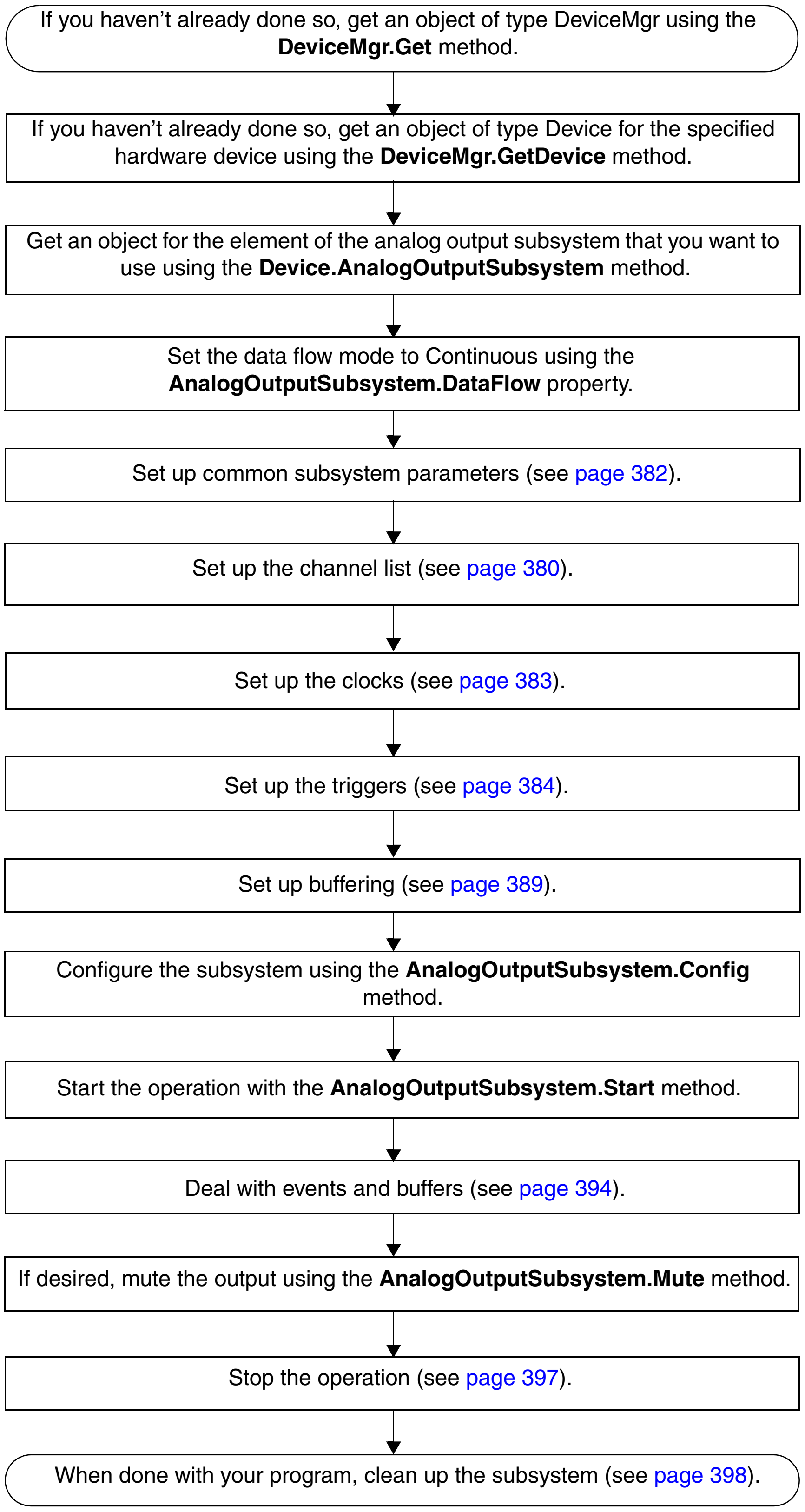
Set up buffering (see page 388).

Configure the subsystem using the **AnalogInputSubsystem.Config** method.

Go to the next page.


Continued from previous page.
Configure the subsystem to execute BufferDoneEvent events synchronously in a single thread using the **AnalogInputSubsystem.SynchronousBufferDone** property.
Start the operation with the **AnalogInputSubsystem.Start** method.
Deal with events and buffers (see page 390).
Stop the operation (see page 397).
When done with your program, clean up the subsystem (see page 398).

Continuous Analog Output Operations



If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.

If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.

Get an object for the element of the analog output subsystem that you want to use using the **Device.AnalogOutputSubsystem** method.

Set the data flow mode to Continuous using the **AnalogOutputSubsystem.DataFlow** property.

Set up common subsystem parameters (see page 382).

Set up the channel list (see page 380).

Set up the clocks (see page 383).

Set up the triggers (see page 384).

Set up buffering (see page 389).

Configure the subsystem using the **AnalogOutputSubsystem.Config** method.

Start the operation with the **AnalogOutputSubsystem.Start** method.

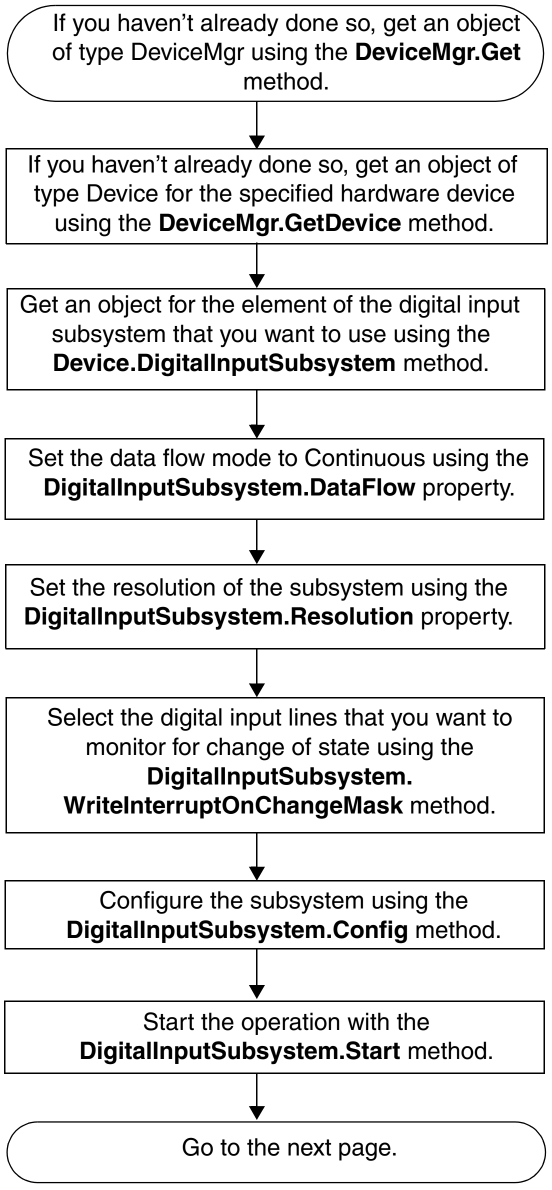
Deal with events and buffers (see page 394).

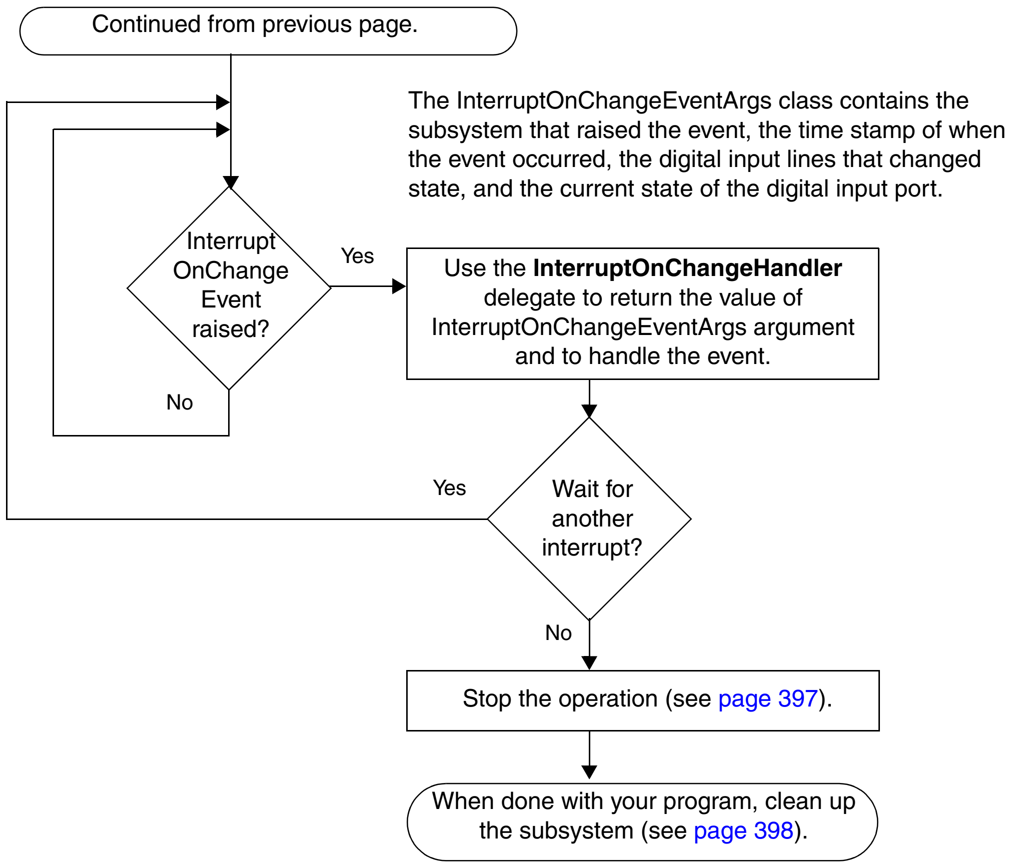
If desired, mute the output using the **AnalogOutputSubsystem.Mute** method.

Stop the operation (see page 397).

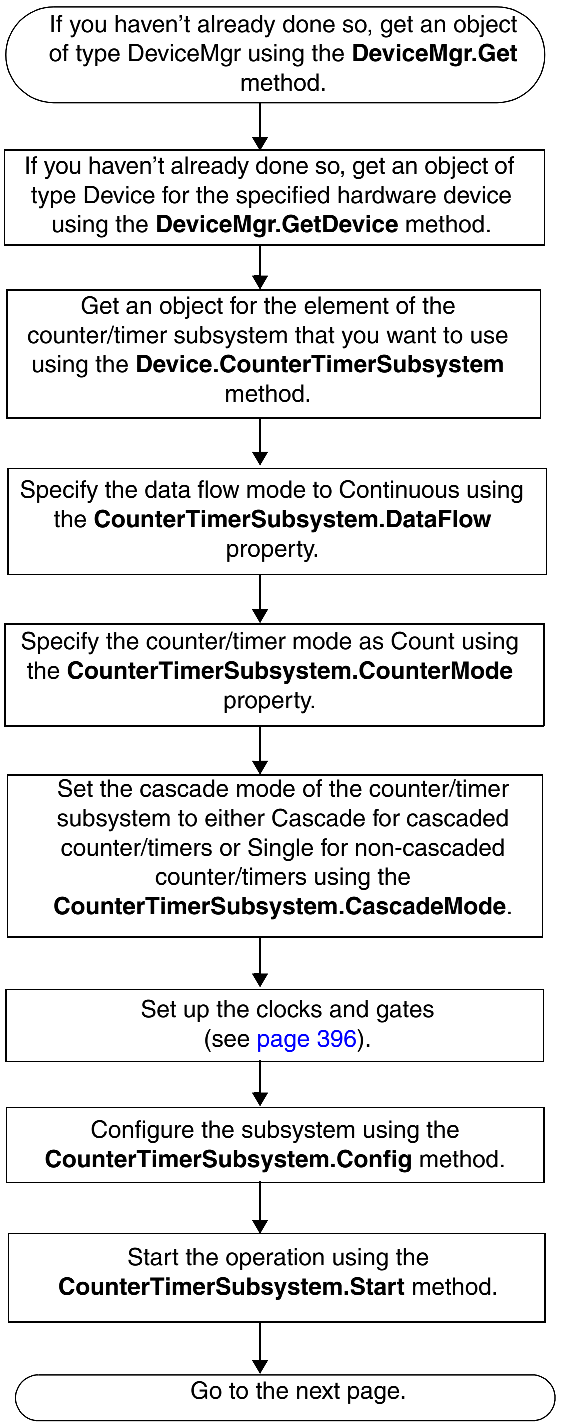
When done with your program, clean up the subsystem (see page 398).

Continuous, Interrupt-On-Change Digital Input Operations


If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.
Get an object for the element of the digital input subsystem that you want to use using the **Device.DigitalInputSubsystem** method.
Set the data flow mode to Continuous using the **DigitalInputSubsystem.DataFlow** property.
Set the resolution of the subsystem using the **DigitalInputSubsystem.Resolution** property.
Select the digital input lines that you want to monitor for change of state using the **DigitalInputSubsystem. WriteInterruptOnChangeMask** method.
Configure the subsystem using the **DigitalInputSubsystem.Config** method.
Start the operation with the **DigitalInputSubsystem.Start** method.
Go to the next page.


Continued from previous page.
The InterruptOnChangeEventArgs class contains the subsystem that raised the event, the time stamp of when the event occurred, the digital input lines that changed state, and the current state of the digital input port.
Interrupt OnChange Event raised?
Yes
No
Use the **InterruptOnChangeHandler** delegate to return the value of InterruptOnChangeEventArgs argument and to handle the event.
Wait for another interrupt?
Yes
No
Stop the operation (see page 397).
When done with your program, clean up the subsystem (see page 398).

Event Counting Operations



If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.

If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.

Get an object for the element of the counter/timer subsystem that you want to use using the **Device.CounterTimerSubsystem** method.

Specify the data flow mode to Continuous using the **CounterTimerSubsystem.DataFlow** property.

Specify the counter/timer mode as Count using the **CounterTimerSubsystem.CounterMode** property.

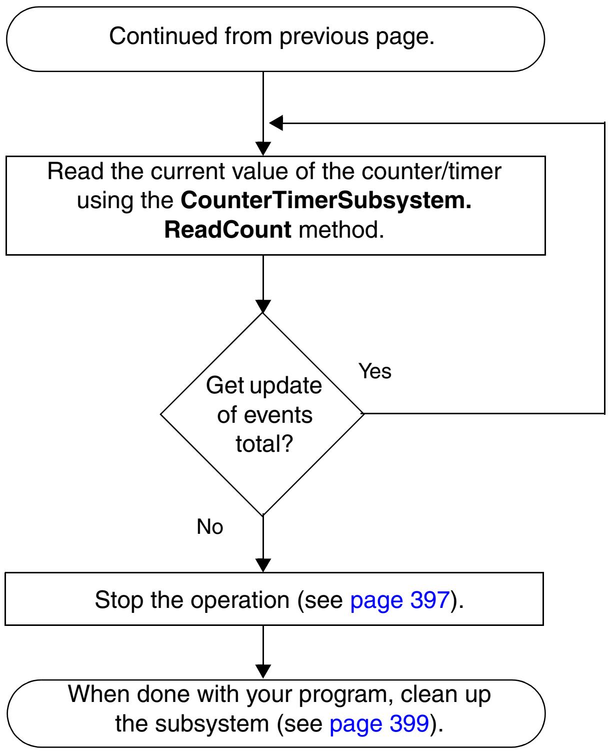
Set the cascade mode of the counter/timer subsystem to either Cascade for cascaded counter/timers or Single for non-cascaded counter/timers using the **CounterTimerSubsystem.CascadeMode**.

Set up the clocks and gates (see page 396).

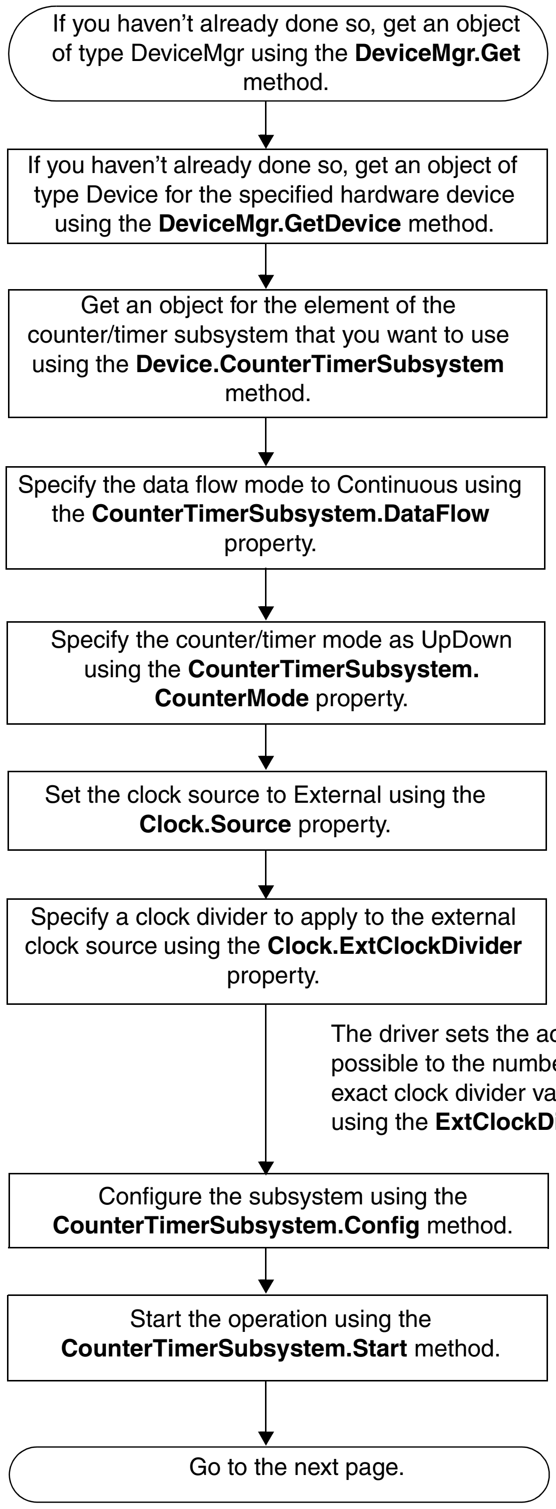
Configure the subsystem using the **CounterTimerSubsystem.Config** method.

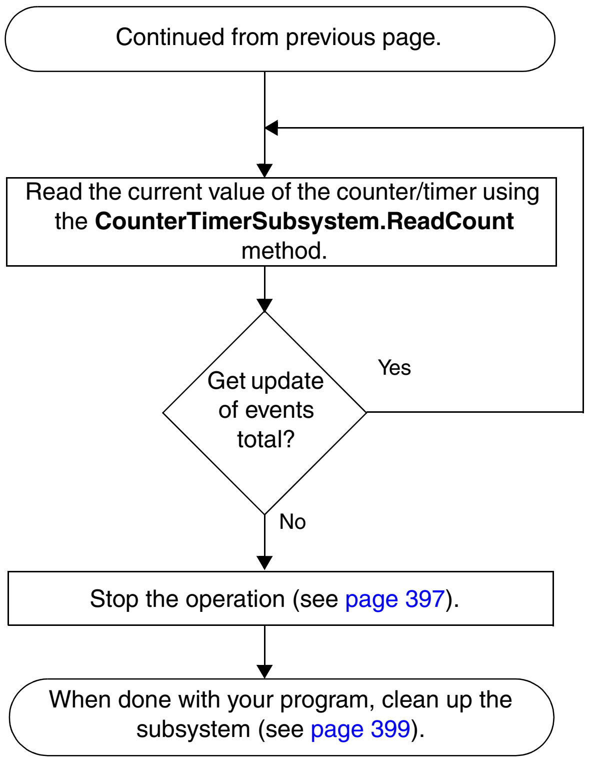
Start the operation using the **CounterTimerSubsystem.Start** method.

Go to the next page.

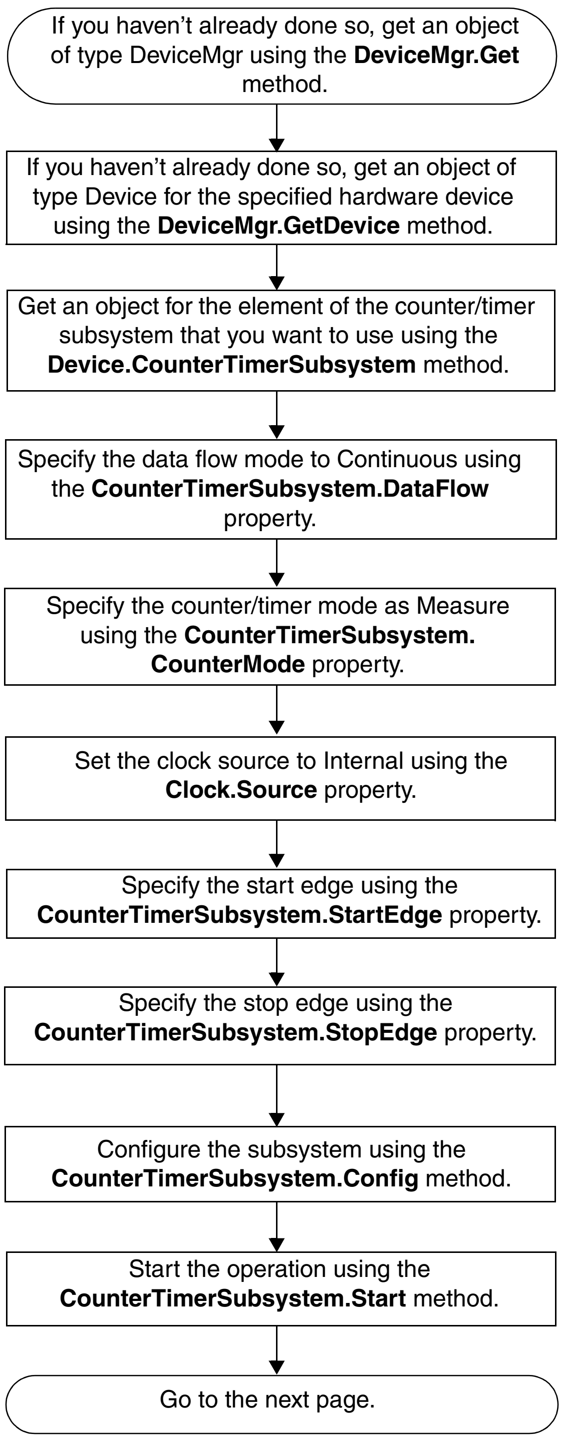

Continued from previous page.
Read the current value of the counter/timer using the **CounterTimerSubsystem. ReadCount** method.
Get update of events total?
Yes
No
Stop the operation (see page 397).
When done with your program, clean up the subsystem (see page 399).

Up/Down Counting Operations


If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.
Get an object for the element of the counter/timer subsystem that you want to use using the **Device.CounterTimerSubsystem** method.
Specify the data flow mode to Continuous using the **CounterTimerSubsystem.DataFlow** property.
Specify the counter/timer mode as UpDown using the **CounterTimerSubsystem. CounterMode** property.
Set the clock source to External using the **Clock.Source** property.
Specify a clock divider to apply to the external clock source using the **Clock.ExtClockDivider** property.
The driver sets the actual clock divider as closely as possible to the number specified. You can read back the exact clock divider value after configuring the subsystem using the **ExtClockDivider** property.
Configure the subsystem using the **CounterTimerSubsystem.Config** method.
Start the operation using the **CounterTimerSubsystem.Start** method.
Go to the next page.


Continued from previous page.
Read the current value of the counter/timer using the **CounterTimerSubsystem.ReadCount** method.
Get update of events total?
Yes
No
Stop the operation (see page 397).
When done with your program, clean up the subsystem (see page 399).

Edge-to-Edge Measurement Operations



If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.

If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.

Get an object for the element of the counter/timer subsystem that you want to use using the **Device.CounterTimerSubsystem** method.

Specify the data flow mode to Continuous using the **CounterTimerSubsystem.DataFlow** property.

Specify the counter/timer mode as Measure using the **CounterTimerSubsystem. CounterMode** property.

Set the clock source to Internal using the **Clock.Source** property.

Specify the start edge using the **CounterTimerSubsystem.StartEdge** property.

Specify the stop edge using the **CounterTimerSubsystem.StopEdge** property.

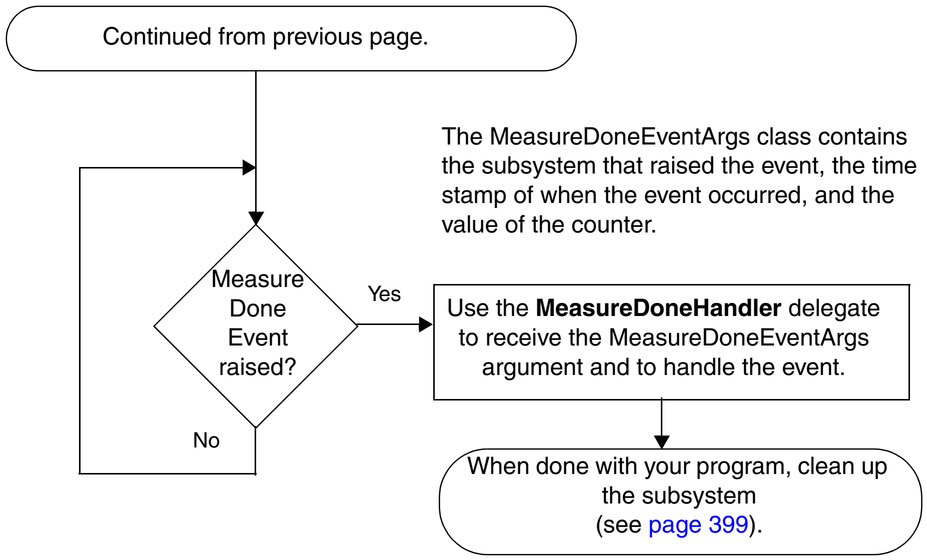
To determine which edges are supported for the **StartEdge** and **StopEdge** properties, read the **CounterTimerSubsystem.SupportedEdgeTypes** property. This property returns an array of the supported signals/edges.

Configure the subsystem using the **CounterTimerSubsystem.Config** method.

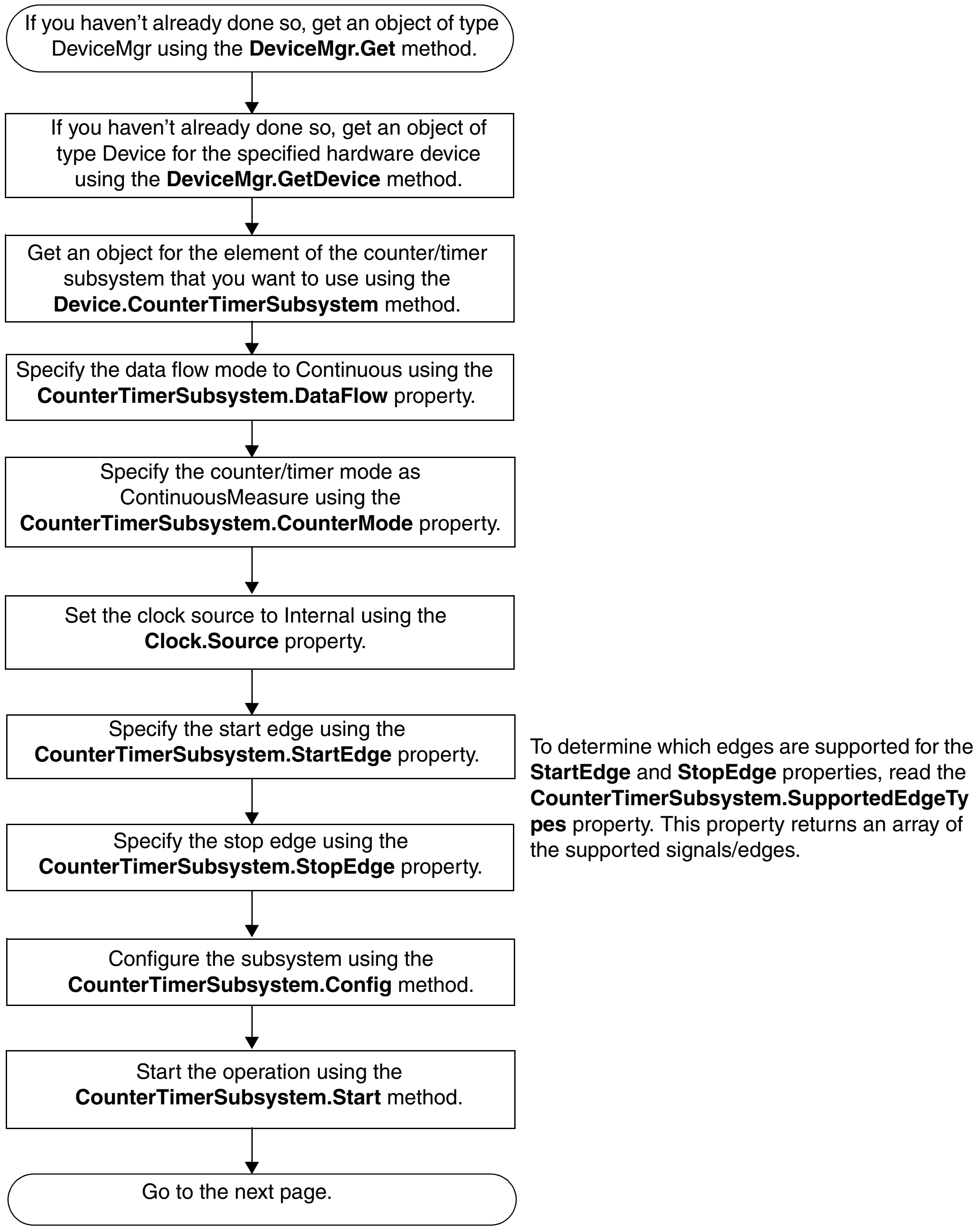
Start the operation using the **CounterTimerSubsystem.Start** method.

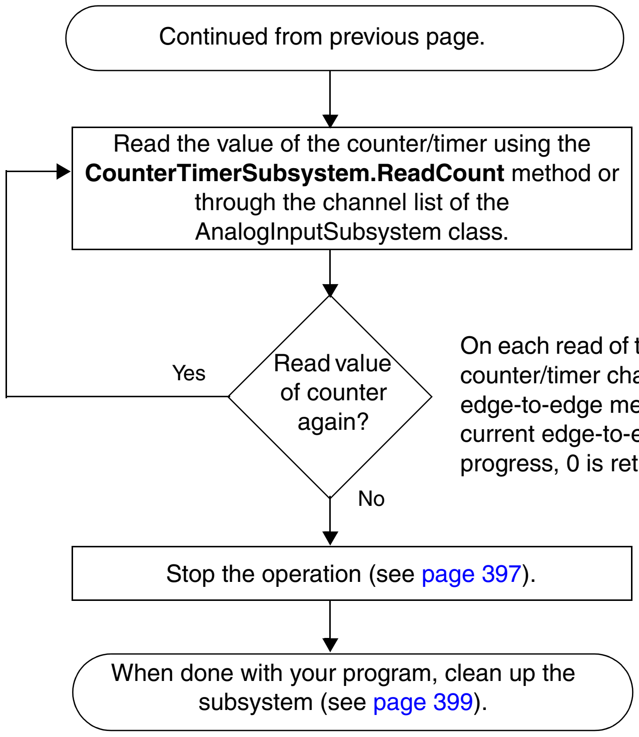
Note that if you want to perform another edge-to-edge measurement, you can call **Start** again.

Go to the next page.


Continued from previous page.
The MeasureDoneEventArgs class contains the subsystem that raised the event, the time stamp of when the event occurred, and the value of the counter.
Measure Done Event raised?
Yes
No
Use the **MeasureDoneHandler** delegate to receive the MeasureDoneEventArgs argument and to handle the event.
When done with your program, clean up the subsystem (see page 399).

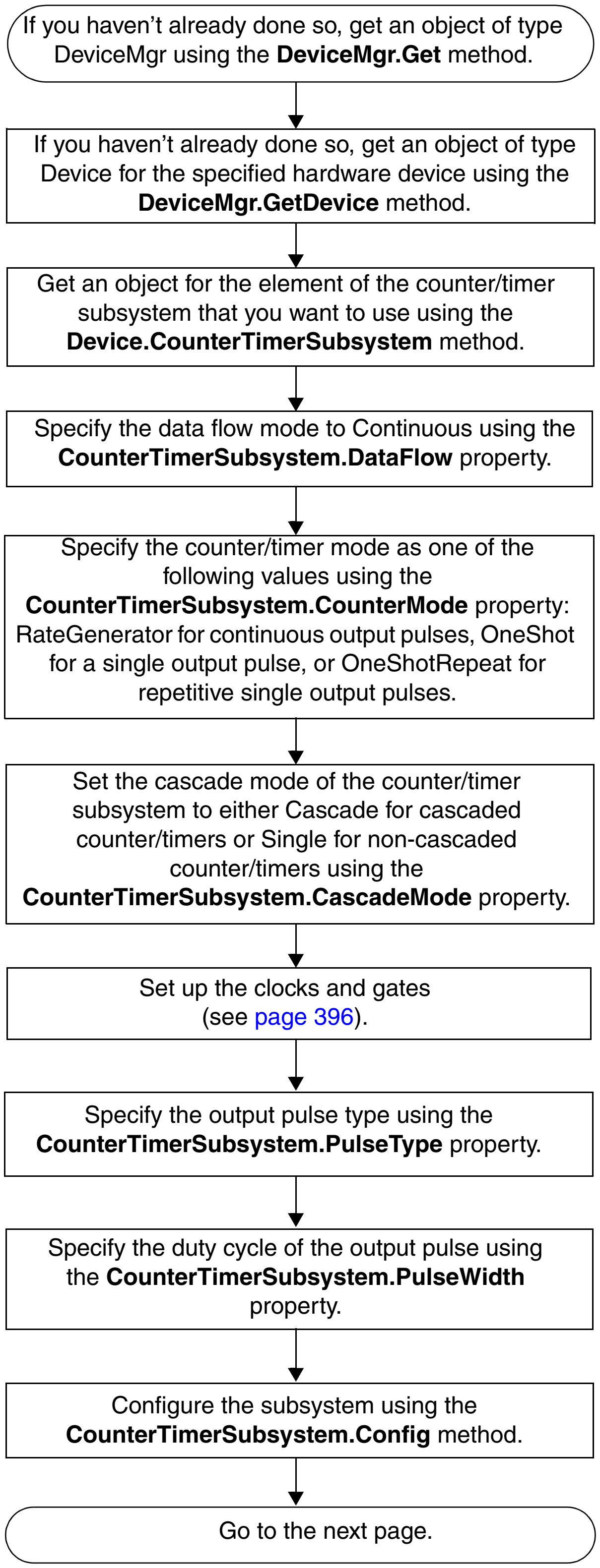
Continuous Edge-to-Edge Measurement Operations


If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.
Get an object for the element of the counter/timer subsystem that you want to use using the **Device.CounterTimerSubsystem** method.
Specify the data flow mode to Continuous using the **CounterTimerSubsystem.DataFlow** property.
Specify the counter/timer mode as ContinuousMeasure using the **CounterTimerSubsystem.CounterMode** property.
Set the clock source to Internal using the **Clock.Source** property.
Specify the start edge using the **CounterTimerSubsystem.StartEdge** property.
To determine which edges are supported for the **StartEdge** and **StopEdge** properties, read the **CounterTimerSubsystem.SupportedEdgeTypes** property. This property returns an array of the supported signals/edges.
Specify the stop edge using the **CounterTimerSubsystem.StopEdge** property.
Configure the subsystem using the **CounterTimerSubsystem.Config** method.
Start the operation using the **CounterTimerSubsystem.Start** method.
Go to the next page.


Continued from previous page.
Read the value of the counter/timer using the **CounterTimerSubsystem.ReadCount** method or through the channel list of the AnalogInputSubsystem class.
Read value of counter again?
Yes
No
Stop the operation (see page 397).
When done with your program, clean up the subsystem (see page 399).

On each read of the counter/timer, the current value of the counter/timer channel is returned and the next edge-to-edge measurement mode operation starts. If the current edge-to-edge measurement operation is still in progress, 0 is returned.

Pulse Output Operations



If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.

If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.

Get an object for the element of the counter/timer subsystem that you want to use using the **Device.CounterTimerSubsystem** method.

Specify the data flow mode to Continuous using the **CounterTimerSubsystem.DataFlow** property.

Specify the counter/timer mode as one of the following values using the **CounterTimerSubsystem.CounterMode** property: RateGenerator for continuous output pulses, OneShot for a single output pulse, or OneShotRepeat for repetitive single output pulses.

Set the cascade mode of the counter/timer subsystem to either Cascade for cascaded counter/timers or Single for non-cascaded counter/timers using the **CounterTimerSubsystem.CascadeMode** property.

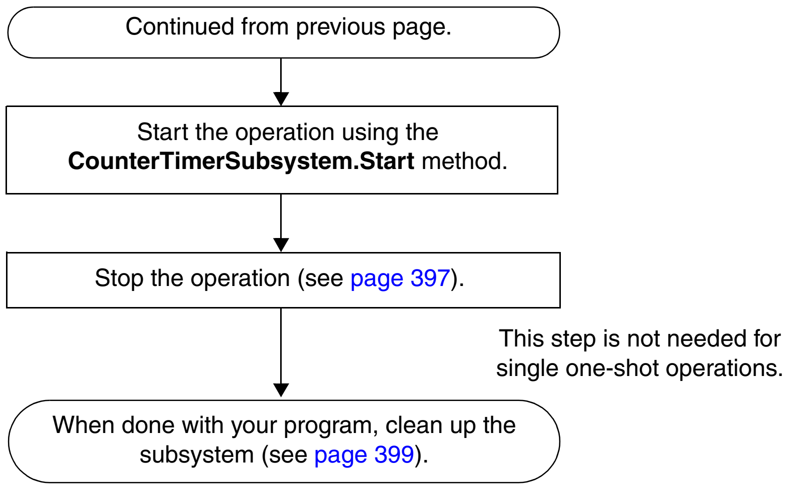
Set up the clocks and gates (see page 396).

Specify the output pulse type using the **CounterTimerSubsystem.PulseType** property.

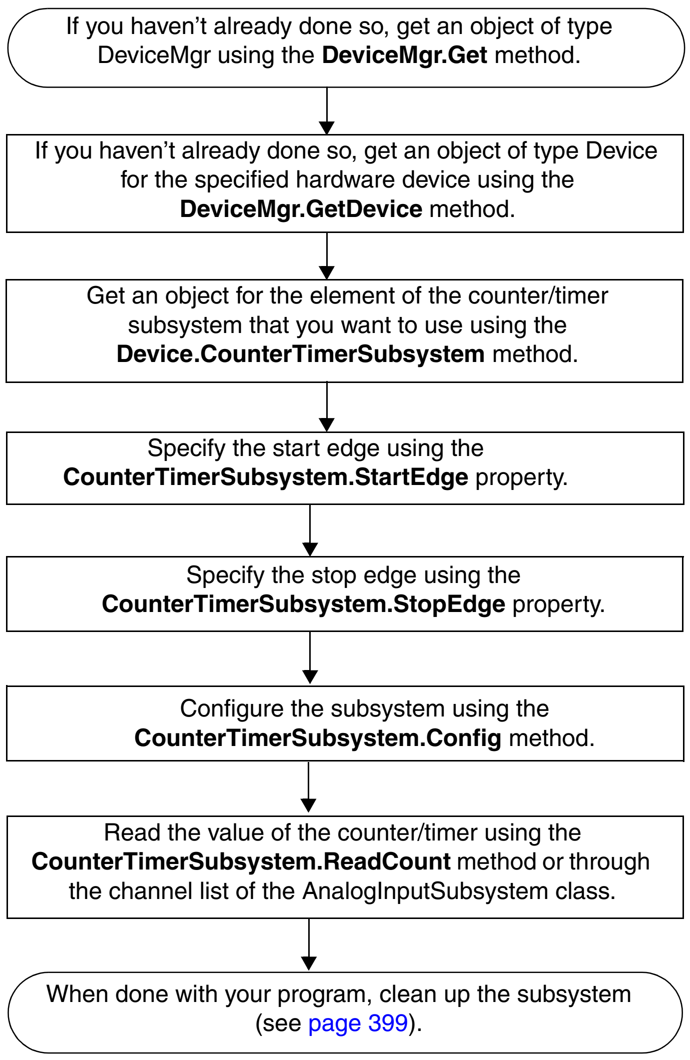
Specify the duty cycle of the output pulse using the **CounterTimerSubsystem.PulseWidth** property.

Configure the subsystem using the **CounterTimerSubsystem.Config** method.

Go to the next page.


Continued from previous page.
Start the operation using the **CounterTimerSubsystem.Start** method.
Stop the operation (see page 397).
This step is not needed for single one-shot operations.
When done with your program, clean up the subsystem (see page 399).

Measure Counter Operations



If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.

If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.

Get an object for the element of the counter/timer subsystem that you want to use using the **Device.CounterTimerSubsystem** method.

Specify the start edge using the **CounterTimerSubsystem.StartEdge** property.

Specify the stop edge using the **CounterTimerSubsystem.StopEdge** property.

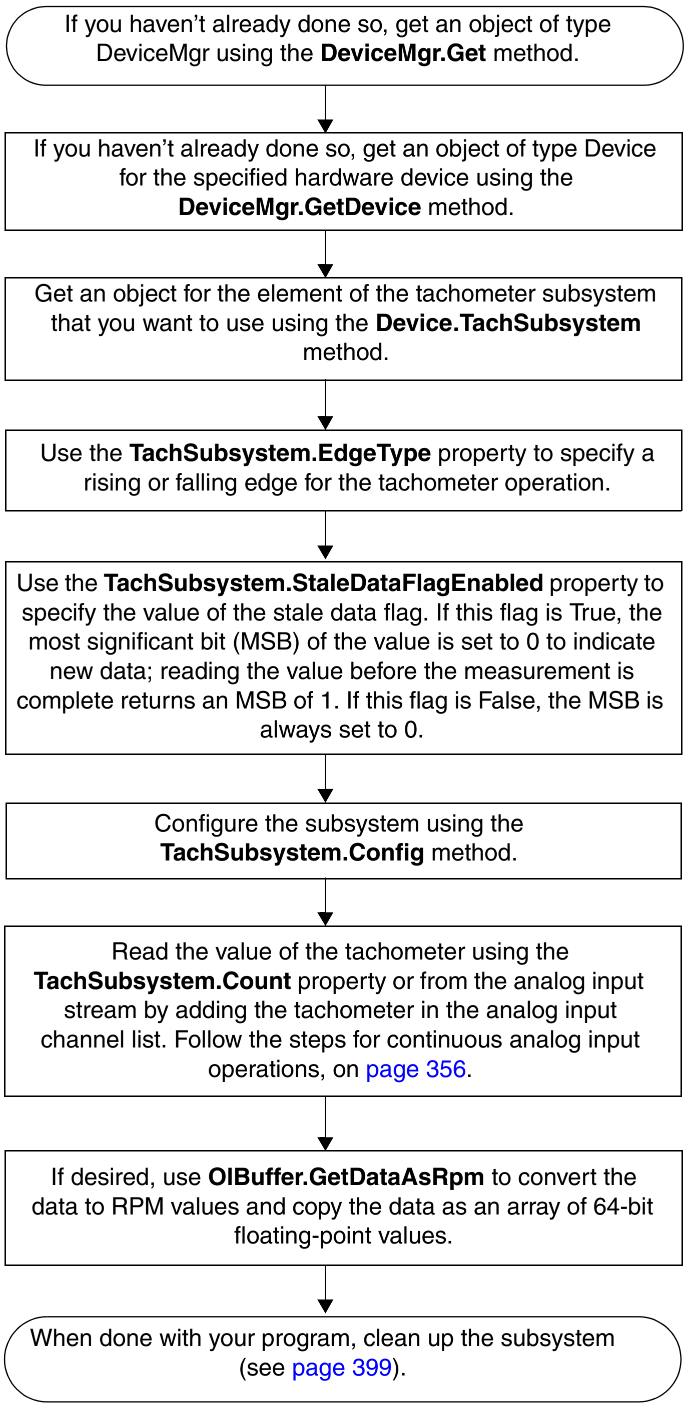
Configure the subsystem using the **CounterTimerSubsystem.Config** method.

Read the value of the counter/timer using the **CounterTimerSubsystem.ReadCount** method or through the channel list of the AnalogInputSubsystem class.

When done with your program, clean up the subsystem (see page 399).

To determine which edges are supported for the **StartEdge** and **StopEdge** properties, read the **CounterTimerSubsystem.SupportedEdgeTyp es** property. This property returns an array of the supported signals/edges.

Tachometer Operations



If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.

If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.

Get an object for the element of the tachometer subsystem that you want to use using the **Device.TachSubsystem** method.

Use the **TachSubsystem.EdgeType** property to specify a rising or falling edge for the tachometer operation.

Use the **TachSubsystem.StaleDataFlagEnabled** property to specify the value of the stale data flag. If this flag is True, the most significant bit (MSB) of the value is set to 0 to indicate new data; reading the value before the measurement is complete returns an MSB of 1. If this flag is False, the MSB is always set to 0.

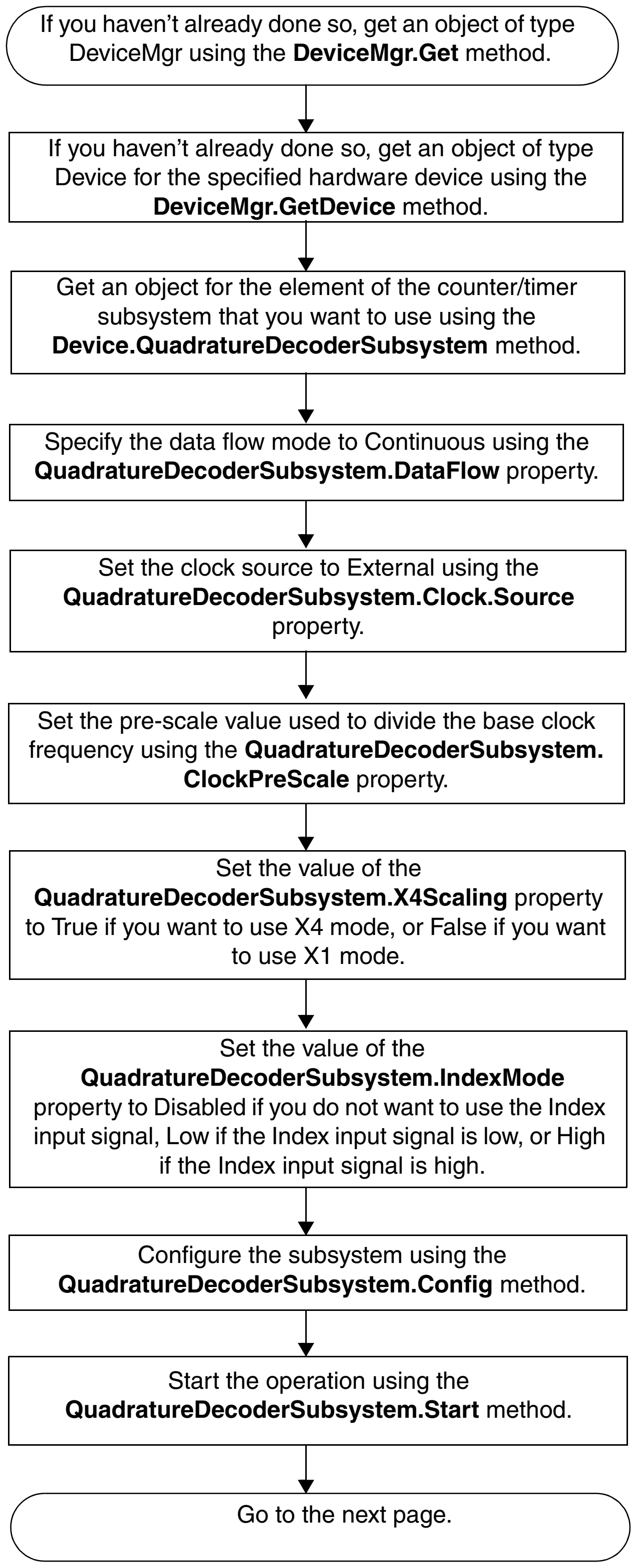
Configure the subsystem using the **TachSubsystem.Config** method.

Read the value of the tachometer using the **TachSubsystem.Count** property or from the analog input stream by adding the tachometer in the analog input channel list. Follow the steps for continuous analog input operations, on page 356.

If desired, use **OlBuffer.GetDataAsRpm** to convert the data to RPM values and copy the data as an array of 64-bit floating-point values.

When done with your program, clean up the subsystem (see page 399).

Quadrature Decoder Operations



If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.

If you haven't already done so, get an object of type Device for the specified hardware device using the **DeviceMgr.GetDevice** method.

Get an object for the element of the counter/timer subsystem that you want to use using the **Device.QuadratureDecoderSubsystem** method.

Specify the data flow mode to Continuous using the **QuadratureDecoderSubsystem.DataFlow** property.

Set the clock source to External using the **QuadratureDecoderSubsystem.Clock.Source** property.

Set the pre-scale value used to divide the base clock frequency using the **QuadratureDecoderSubsystem. ClockPreScale** property.

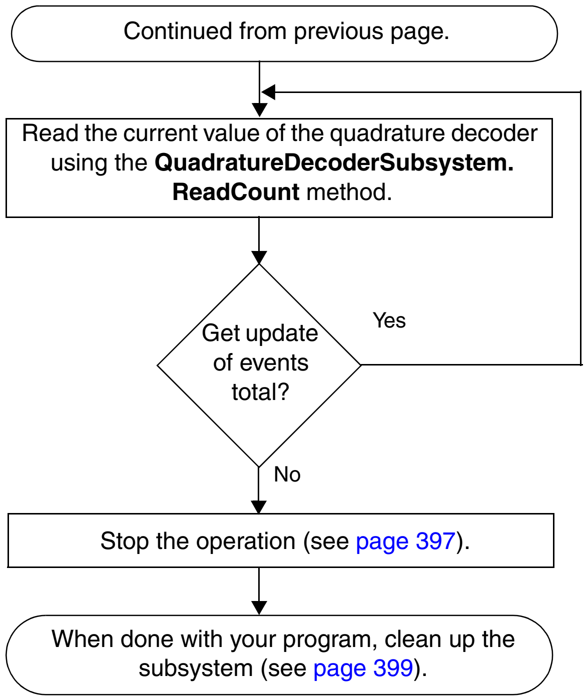
Set the value of the **QuadratureDecoderSubsystem.X4Scaling** property to True if you want to use X4 mode, or False if you want to use X1 mode.

Set the value of the **QuadratureDecoderSubsystem.IndexMode** property to Disabled if you do not want to use the Index input signal, Low if the Index input signal is low, or High if the Index input signal is high.

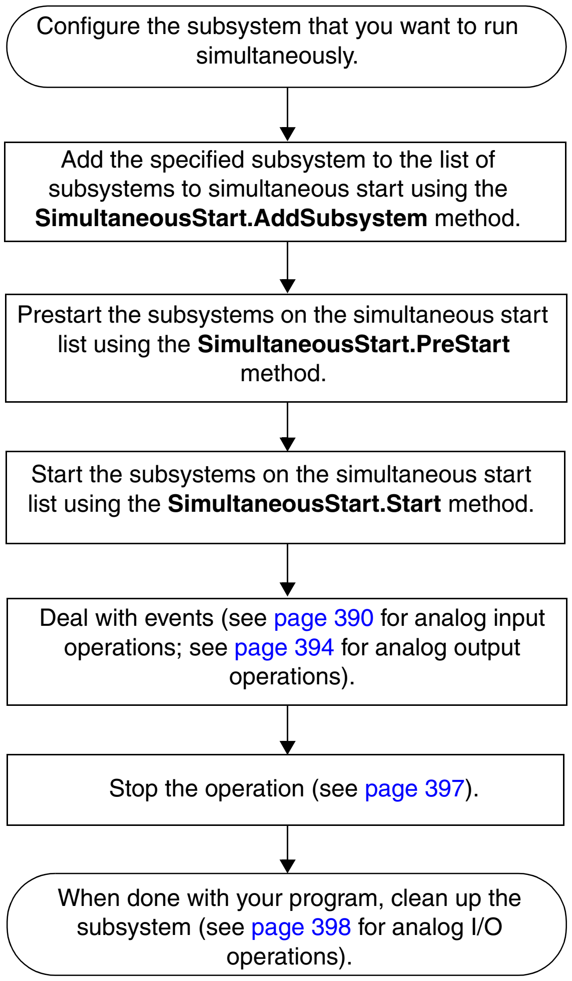
Configure the subsystem using the **QuadratureDecoderSubsystem.Config** method.

Start the operation using the **QuadratureDecoderSubsystem.Start** method.

Go to the next page.

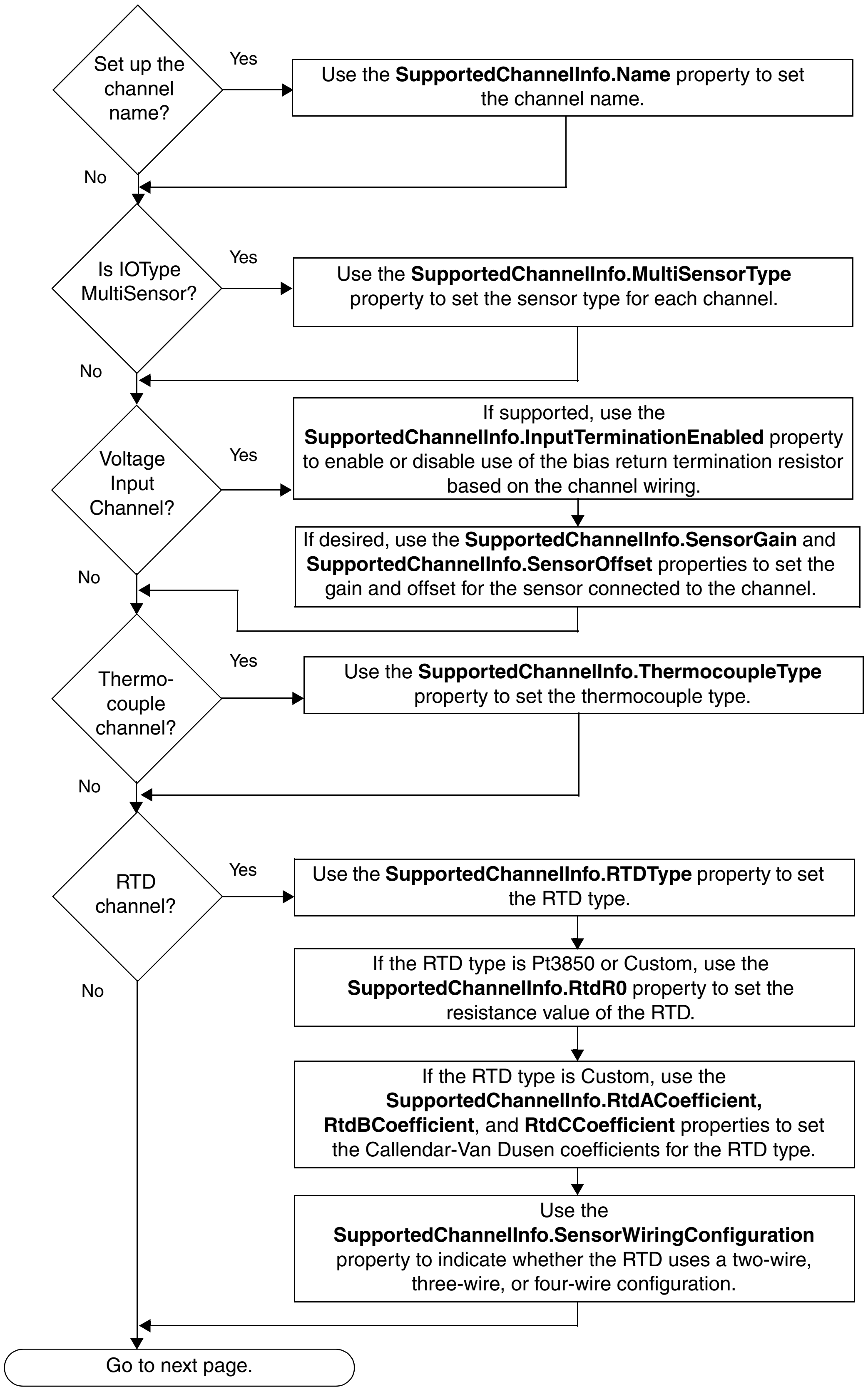

Continued from previous page.
Read the current value of the quadrature decoder using the **QuadratureDecoderSubsystem. ReadCount** method.
Get update of events total?
Yes
No
Stop the operation (see page 397).
When done with your program, clean up the subsystem (see page 399).

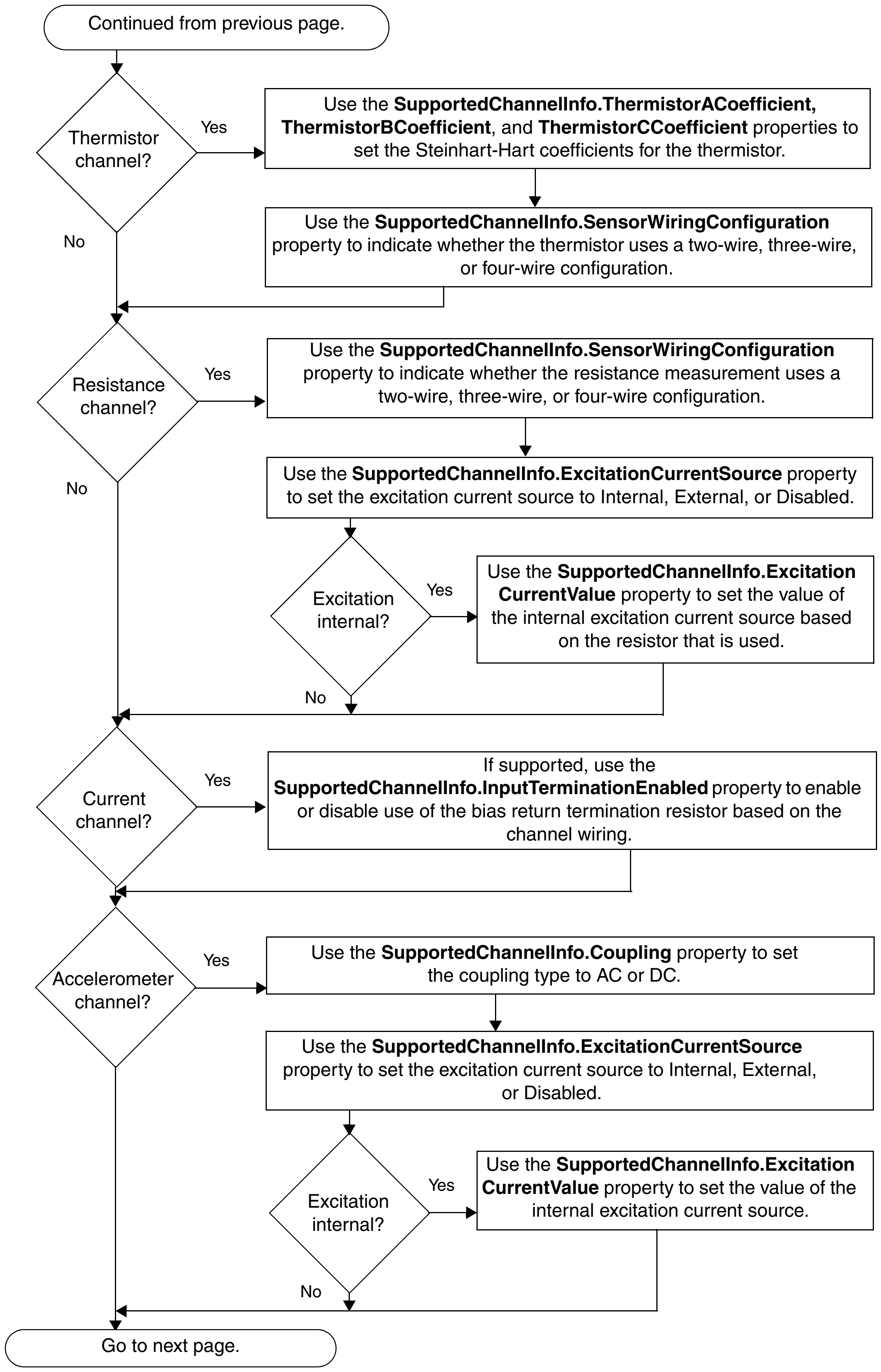
Simultaneously Starting Subsystems

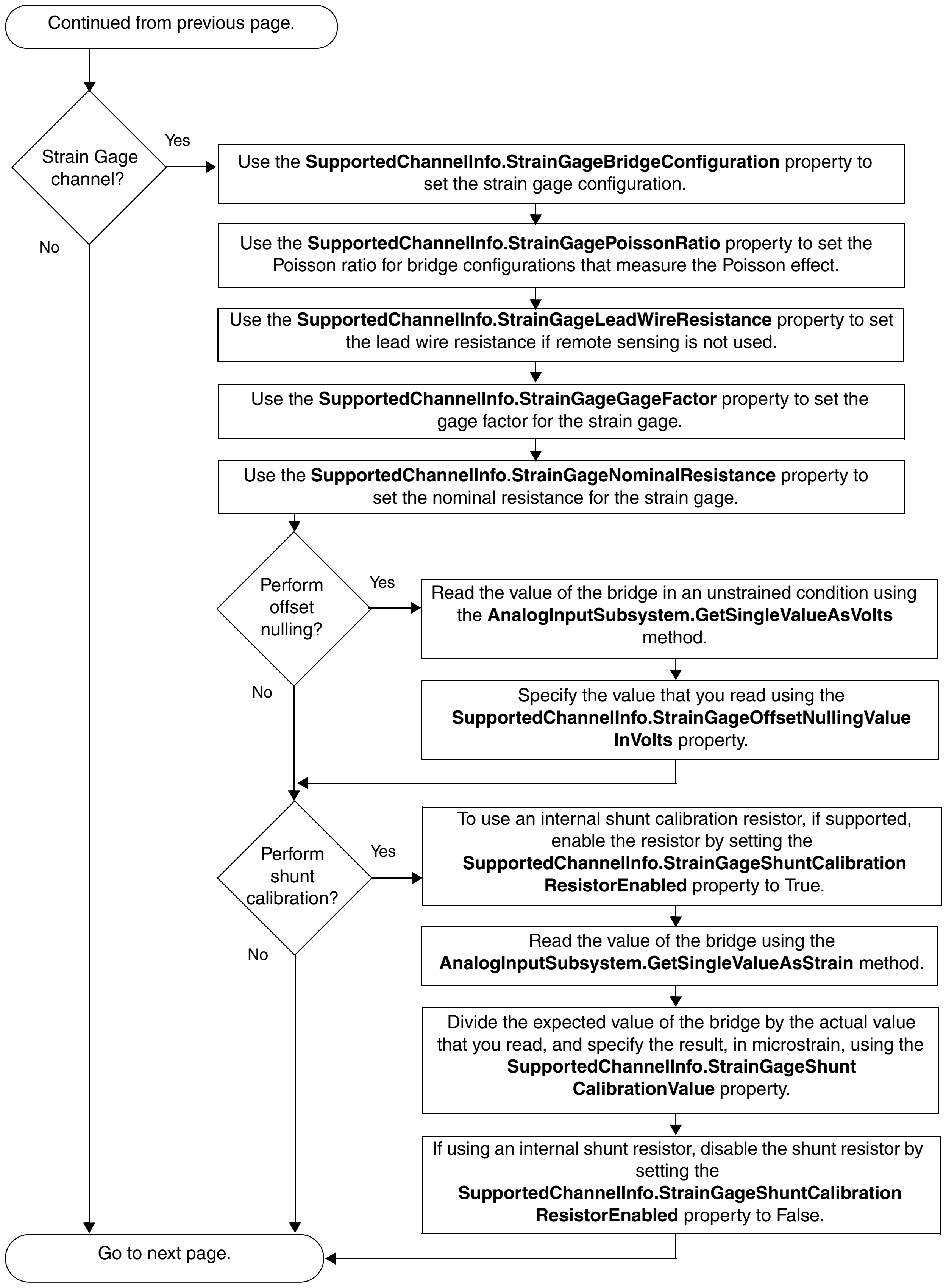

Configure the subsystem that you want to run simultaneously.
Add the specified subsystem to the list of subsystems to simultaneous start using the **SimultaneousStart.AddSubsystem** method.
Prestart the subsystems on the simultaneous start list using the **SimultaneousStart.PreStart** method.
Start the subsystems on the simultaneous start list using the **SimultaneousStart.Start** method.
Deal with events (see page 390 for analog input operations; see page 394 for analog output operations).
Stop the operation (see page 397).
When done with your program, clean up the subsystem (see page 398 for analog I/O operations).

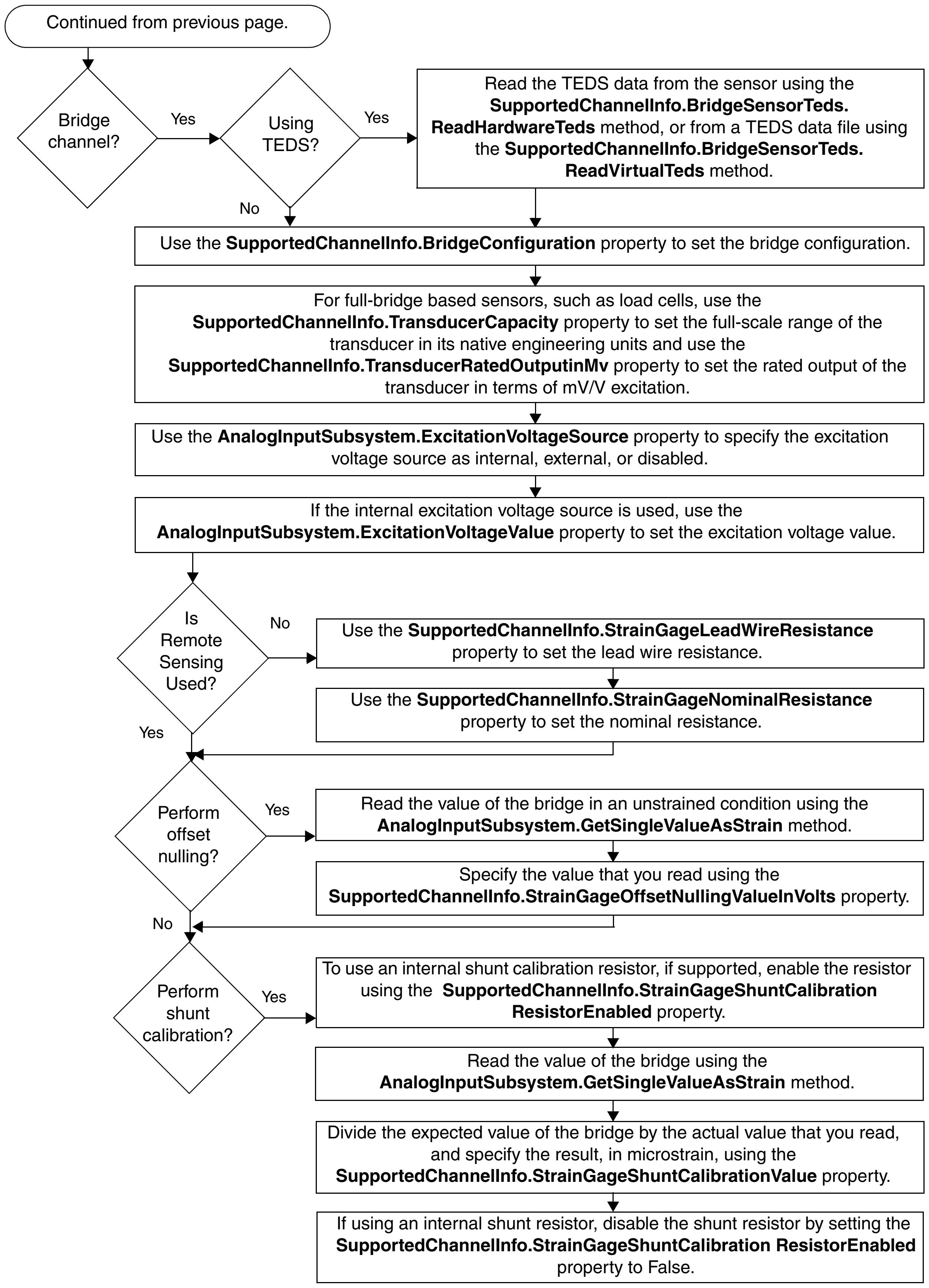
See the previous flow diagrams in this chapter; you cannot perform single-value operations simultaneously on multiplexed A/D modules.

Set Up Analog Input Channels


Set up the channel name?
Yes
No
Use the **SupportedChannelInfo.Name** property to set the channel name.
Is IOType MultiSensor?
Yes
No
Use the **SupportedChannelInfo.MultiSensorType** property to set the sensor type for each channel.
Voltage Input Channel?
Yes
No
If supported, use the **SupportedChannelInfo.InputTerminationEnabled** property to enable or disable use of the bias return termination resistor based on the channel wiring.
If desired, use the **SupportedChannelInfo.SensorGain** and **SupportedChannelInfo.SensorOffset** properties to set the gain and offset for the sensor connected to the channel.
Thermo-couple channel?
Yes
No
Use the **SupportedChannelInfo.ThermocoupleType** property to set the thermocouple type.
RTD channel?
Yes
No
Use the **SupportedChannelInfo.RTDType** property to set the RTD type.
If the RTD type is Pt3850 or Custom, use the **SupportedChannelInfo.RtdR0** property to set the resistance value of the RTD.
If the RTD type is Custom, use the **SupportedChannelInfo.RtdACoefficient, RtdBCoefficient**, and **RtdCCoefficient** properties to set the Callendar-Van Dusen coefficients for the RTD type.
Use the **SupportedChannelInfo.SensorWiringConfiguration** property to indicate whether the RTD uses a two-wire, three-wire, or four-wire configuration.
Go to next page.


Continued from previous page.
Thermistor channel?
Yes
No
Use the **SupportedChannelInfo.ThermistorACoefficient, ThermistorBCoefficient**, and **ThermistorCCoefficient** properties to set the Steinhart-Hart coefficients for the thermistor.
Use the **SupportedChannelInfo.SensorWiringConfiguration** property to indicate whether the thermistor uses a two-wire, three-wire, or four-wire configuration.
Resistance channel?
Yes
No
Use the **SupportedChannelInfo.SensorWiringConfiguration** property to indicate whether the resistance measurement uses a two-wire, three-wire, or four-wire configuration.
Use the **SupportedChannelInfo.ExcitationCurrentSource** property to set the excitation current source to Internal, External, or Disabled.
Excitation internal?
Yes
No
Use the **SupportedChannelInfo.ExcitationCurrentValue** property to set the value of the internal excitation current source based on the resistor that is used.
Current channel?
Yes
If supported, use the **SupportedChannelInfo.InputTerminationEnabled** property to enable or disable use of the bias return termination resistor based on the channel wiring.
Accelerometer channel?
Yes
Use the **SupportedChannelInfo.Coupling** property to set the coupling type to AC or DC.
Use the **SupportedChannelInfo.ExcitationCurrentSource** property to set the excitation current source to Internal, External, or Disabled.
Excitation internal?
Yes
No
Use the **SupportedChannelInfo.ExcitationCurrentValue** property to set the value of the internal excitation current source.
Go to next page.


Continued from previous page.
Strain Gage channel?
Yes
No
Use the **SupportedChannelInfo.StrainGageBridgeConfiguration** property to set the strain gage configuration.
Use the **SupportedChannelInfo.StrainGagePoissonRatio** property to set the Poisson ratio for bridge configurations that measure the Poisson effect.
Use the **SupportedChannelInfo.StrainGageLeadWireResistance** property to set the lead wire resistance if remote sensing is not used.
Use the **SupportedChannelInfo.StrainGageGageFactor** property to set the gage factor for the strain gage.
Use the **SupportedChannelInfo.StrainGageNominalResistance** property to set the nominal resistance for the strain gage.
Perform offset nulling?
Yes
No
Read the value of the bridge in an unstrained condition using the **AnalogInputSubsystem.GetSingleValueAsVolts** method.
Specify the value that you read using the **SupportedChannelInfo.StrainGageOffsetNullingValueInVolts** property.
Perform shunt calibration?
Yes
No
To use an internal shunt calibration resistor, if supported, enable the resistor by setting the **SupportedChannelInfo.StrainGageShuntCalibrationResistorEnabled** property to True.
Read the value of the bridge using the **AnalogInputSubsystem.GetSingleValueAsStrain** method.
Divide the expected value of the bridge by the actual value that you read, and specify the result, in microstrain, using the **SupportedChannelInfo.StrainGageShuntCalibrationValue** property.
If using an internal shunt resistor, disable the shunt resistor by setting the **SupportedChannelInfo.StrainGageShuntCalibrationResistorEnabled** property to False.
Go to next page.



Read the TEDS data from the sensor using the **SupportedChannelInfo.BridgeSensorTeds. ReadHardwareTeds** method, or from a TEDS data file using the **SupportedChannelInfo.BridgeSensorTeds. ReadVirtualTeds** method.

Use the **SupportedChannelInfo.BridgeConfiguration** property to set the bridge configuration.

For full-bridge based sensors, such as load cells, use the **SupportedChannelInfo.TransducerCapacity** property to set the full-scale range of the transducer in its native engineering units and use the **SupportedChannelInfo.TransducerRatedOutputinMv** property to set the rated output of the transducer in terms of mV/V excitation.

Use the **AnalogInputSubsystem.ExcitationVoltageSource** property to specify the excitation voltage source as internal, external, or disabled.

If the internal excitation voltage source is used, use the **AnalogInputSubsystem.ExcitationVoltageValue** property to set the excitation voltage value.

Use the **SupportedChannelInfo.StrainGageLeadWireResistance** property to set the lead wire resistance.

Use the **SupportedChannelInfo.StrainGageNominalResistance** property to set the nominal resistance.

Read the value of the bridge in an unstrained condition using the **AnalogInputSubsystem.GetSingleValueAsStrain** method.

Specify the value that you read using the **SupportedChannelInfo.StrainGageOffsetNullingValueInVolts** property.

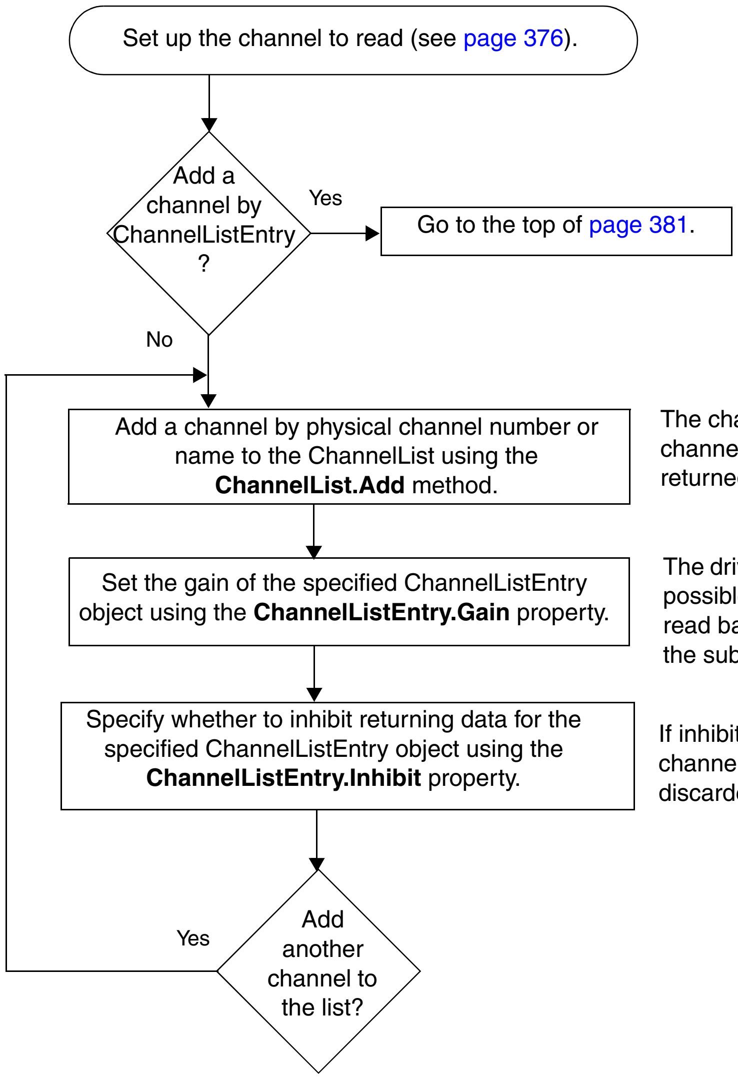
To use an internal shunt calibration resistor, if supported, enable the resistor using the **SupportedChannelInfo.StrainGageShuntCalibration ResistorEnabled** property.

Read the value of the bridge using the **AnalogInputSubsystem.GetSingleValueAsStrain** method.

Divide the expected value of the bridge by the actual value that you read, and specify the result, in microstrain, using the **SupportedChannelInfo.StrainGageShuntCalibrationValue** property.

If using an internal shunt resistor, disable the shunt resistor by setting the **SupportedChannelInfo.StrainGageShuntCalibration ResistorEnabled** property to False.

Set Up the ChannelList

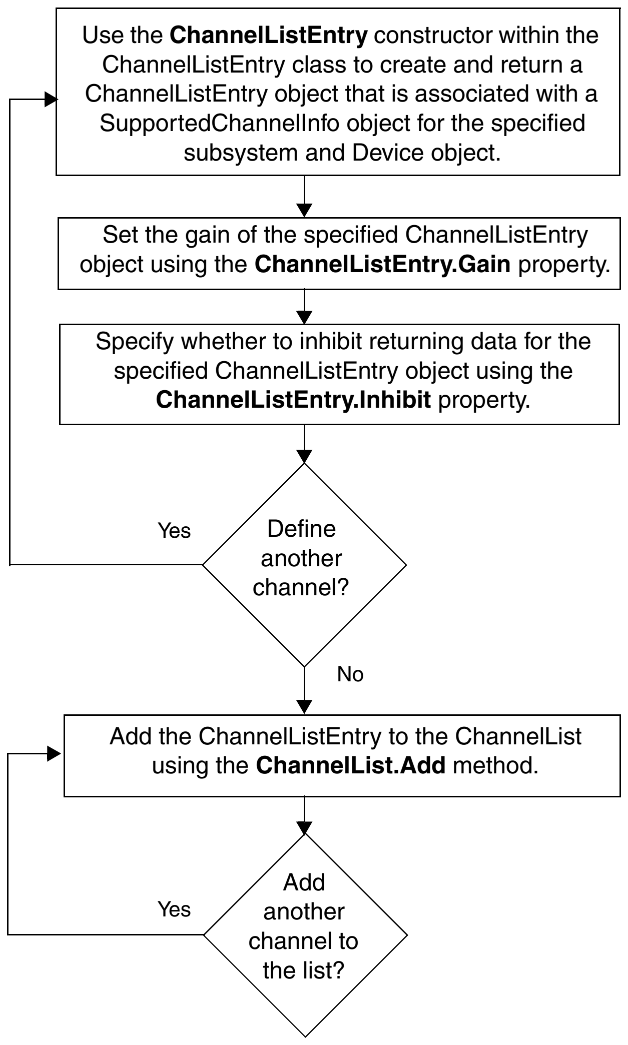

Set up the channel to read (see page 376).
Add a channel by ChannelListEntry ?
Yes
Go to the top of page 381.
No
Add a channel by physical channel number or name to the ChannelList using the **ChannelList.Add** method.
Set the gain of the specified ChannelListEntry object using the **ChannelListEntry.Gain** property.
Specify whether to inhibit returning data for the specified ChannelListEntry object using the **ChannelListEntry.Inhibit** property.
Yes
Add another channel to the list?

The channel is appended to the end of the channel list. A ChannelListEntry object is returned.

The driver sets the actual gain as closely as possible to the number specified. You can read back the exact gain after configuring the subsystem using the **Gain** property.

If inhibited, the values for the specified channel object are acquired, and then discarded.

Add a Channel by ChannelListyEntry

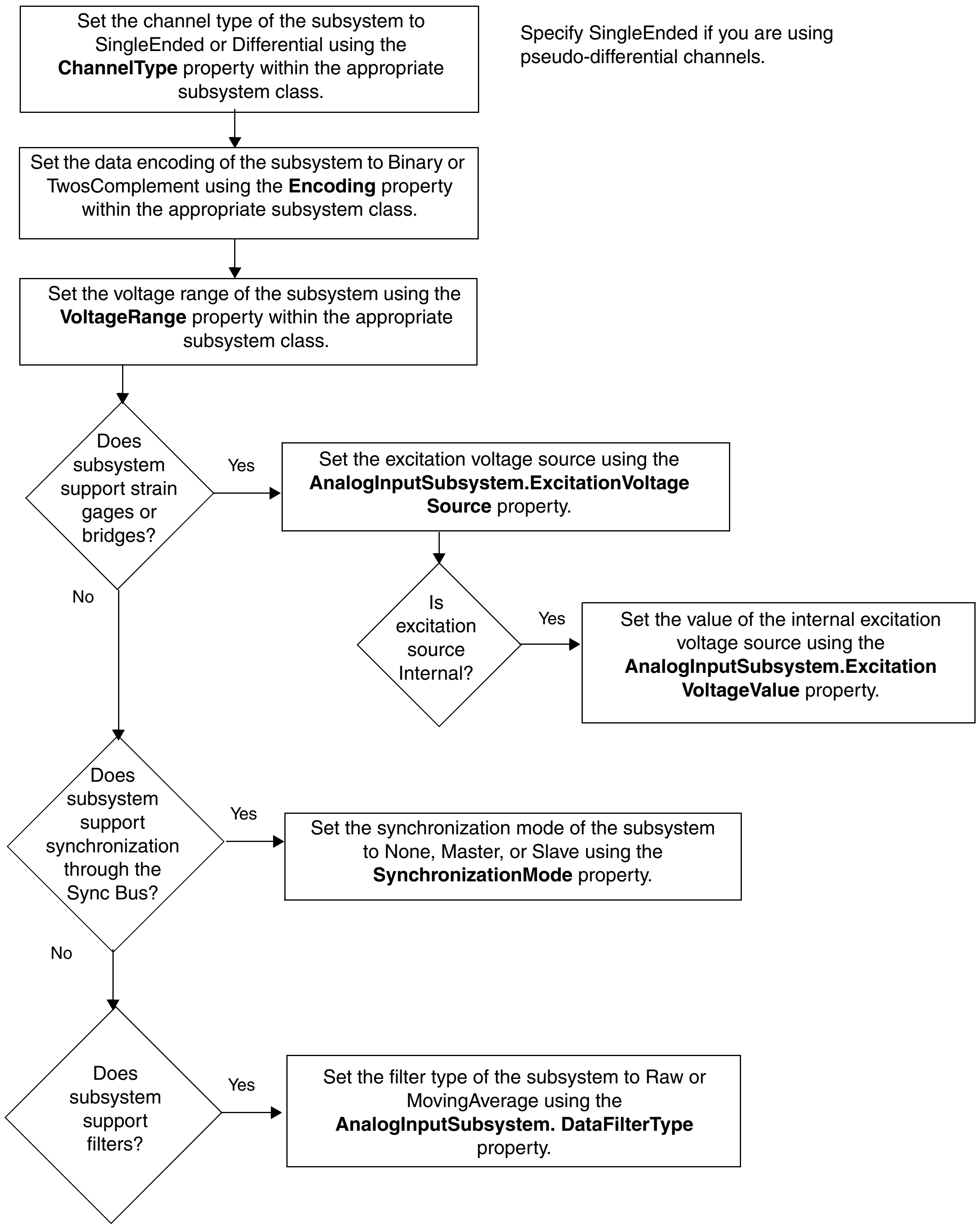

Use the **ChannelListEntry** constructor within the ChannelListEntry class to create and return a ChannelListEntry object that is associated with a SupportedChannelInfo object for the specified subsystem and Device object.
Set the gain of the specified ChannelListEntry object using the **ChannelListEntry.Gain** property.
Specify whether to inhibit returning data for the specified ChannelListEntry object using the **ChannelListEntry.Inhibit** property.
Yes
Define another channel?
No
Add the ChannelListEntry to the ChannelList using the **ChannelList.Add** method.
Yes
Add another channel to the list?

The driver sets the actual gain as closely as possible to the number specified. You can read back the exact gain after configuring the subsystem using the **Gain** property.

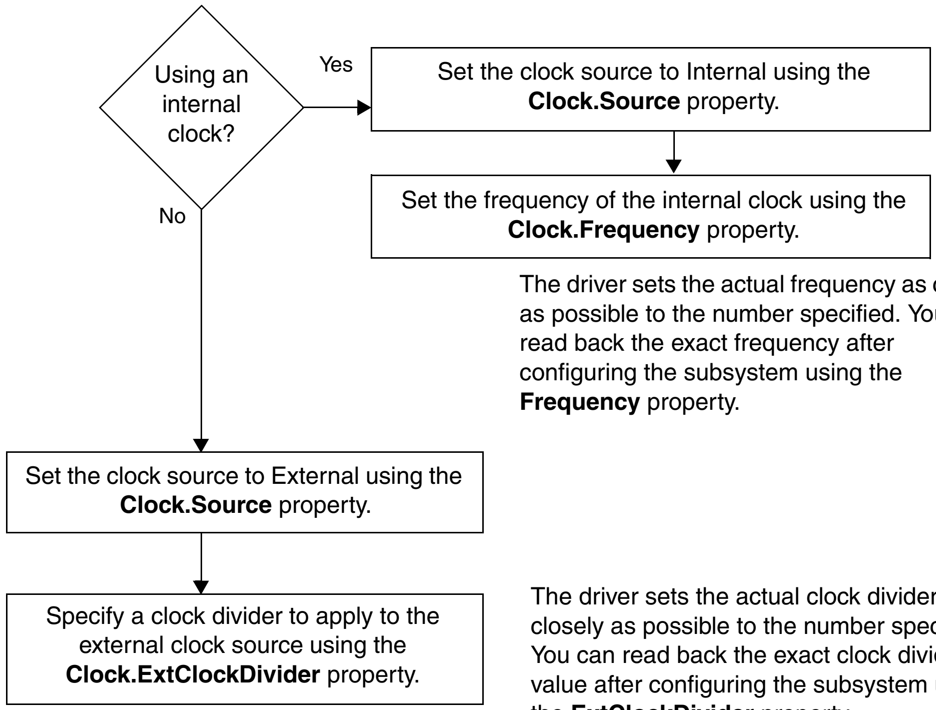
If inhibited, the values for the specified channel object are acquired, and then discarded.

The channel is appended to the end of the channel list.

Set up Common Subsystem Parameters


Set the channel type of the subsystem to SingleEnded or Differential using the **ChannelType** property within the appropriate subsystem class.
Specify SingleEnded if you are using pseudo-differential channels.
Set the data encoding of the subsystem to Binary or TwosComplement using the **Encoding** property within the appropriate subsystem class.
Set the voltage range of the subsystem using the **VoltageRange** property within the appropriate subsystem class.
Does subsystem support strain gages or bridges?
Yes
Set the excitation voltage source using the **AnalogInputSubsystem.ExcitationVoltage Source** property.
Is excitation source Internal?
Yes
Set the value of the internal excitation voltage source using the **AnalogInputSubsystem.Excitation VoltageValue** property.
No
Does subsystem support synchronization through the Sync Bus?
Yes
Set the synchronization mode of the subsystem to None, Master, or Slave using the **SynchronizationMode** property.
No
Does subsystem support filters?
Yes
Set the filter type of the subsystem to Raw or MovingAverage using the **AnalogInputSubsystem. DataFilterType** property.

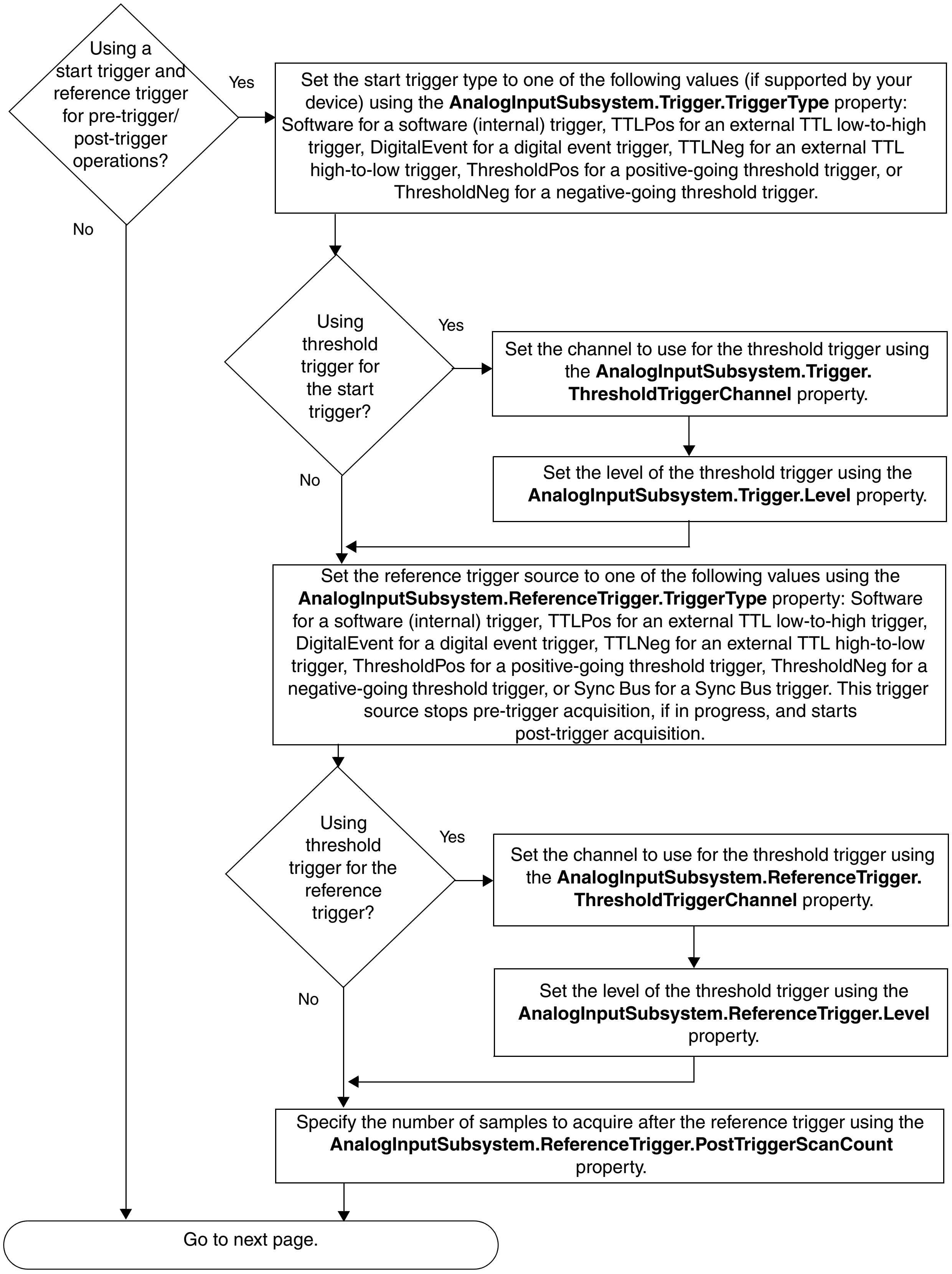
Set Up Clocks

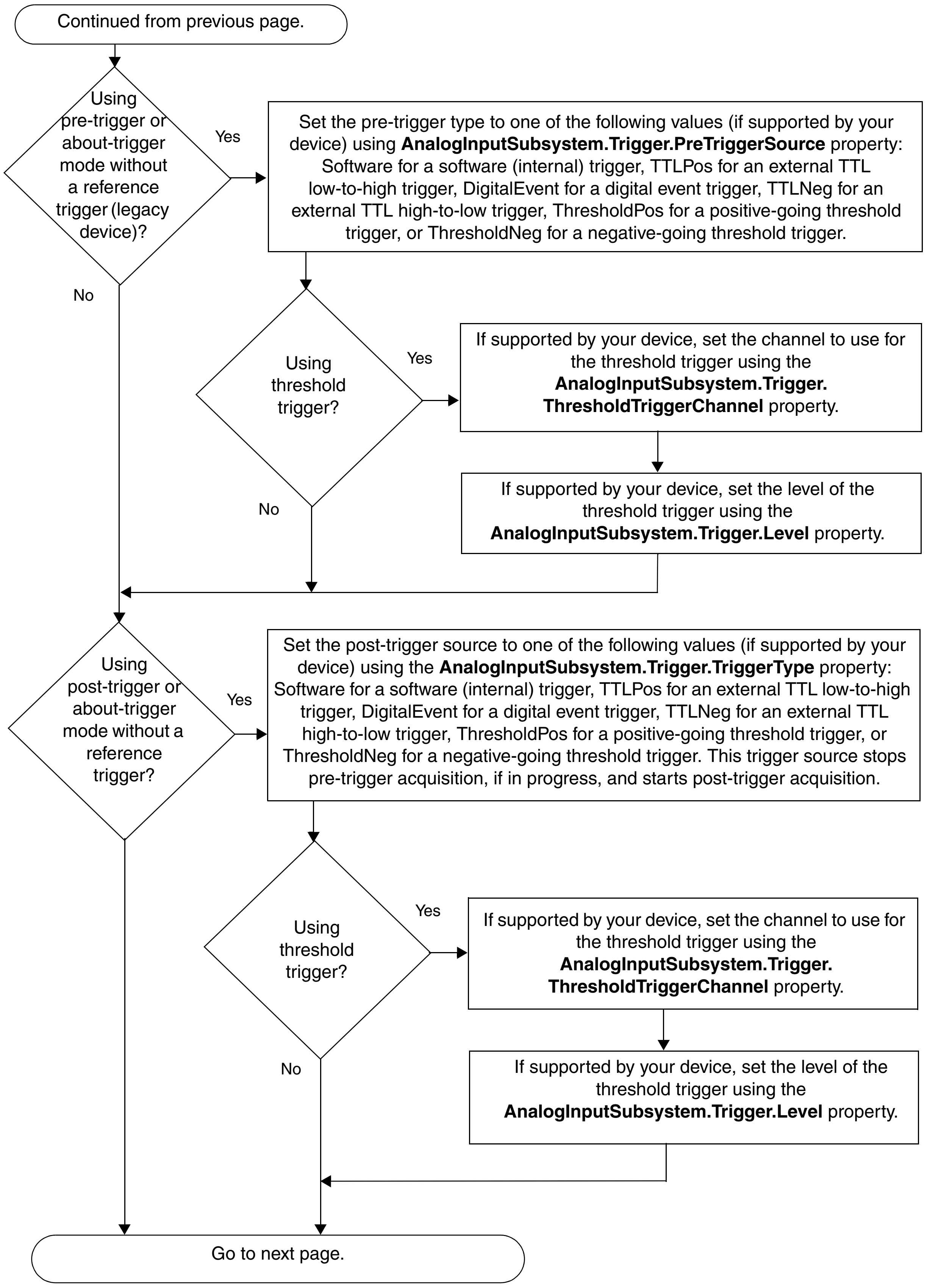

Using an internal clock?
Yes
Set the clock source to Internal using the **Clock.Source** property.
Set the frequency of the internal clock using the **Clock.Frequency** property.
No
Set the clock source to External using the **Clock.Source** property.
Specify a clock divider to apply to the external clock source using the **Clock.ExtClockDivider** property.

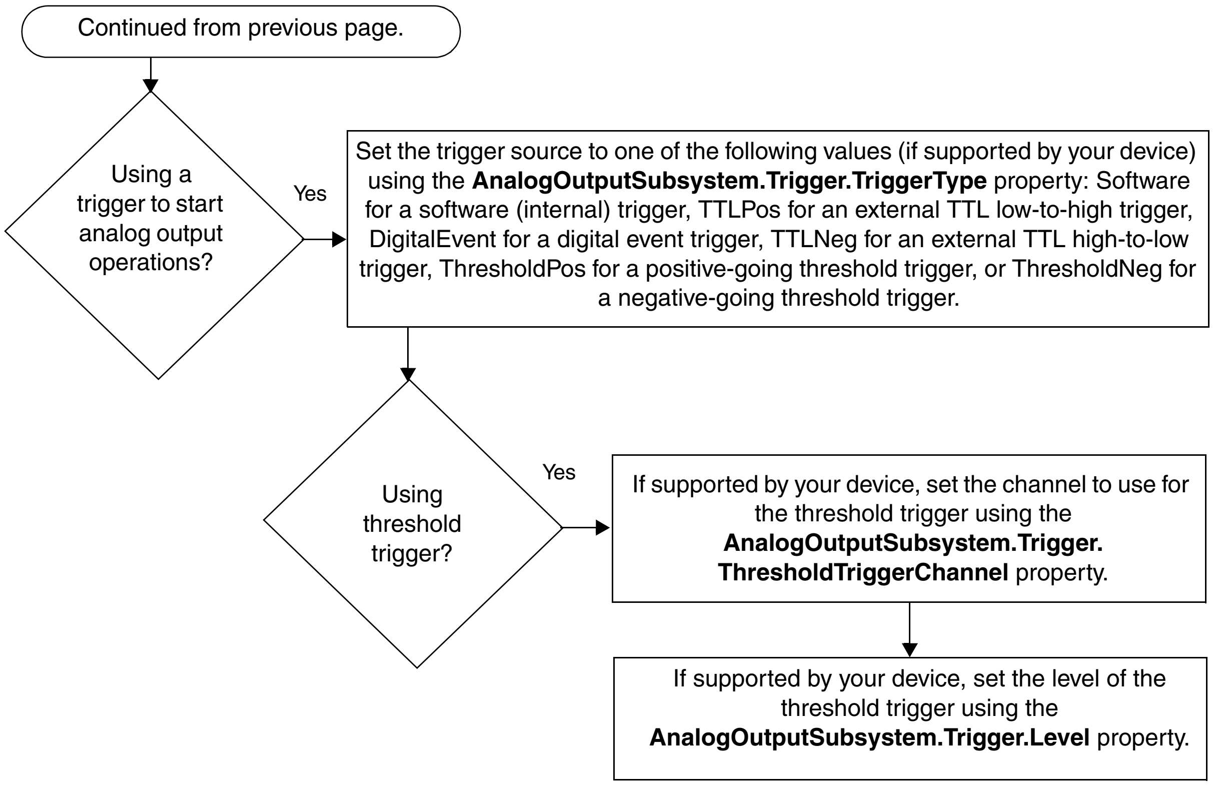
The driver sets the actual frequency as closely as possible to the number specified. You can read back the exact frequency after configuring the subsystem using the **Frequency** property.

The driver sets the actual clock divider as closely as possible to the number specified. You can read back the exact clock divider value after configuring the subsystem using the **ExtClockDivider** property.

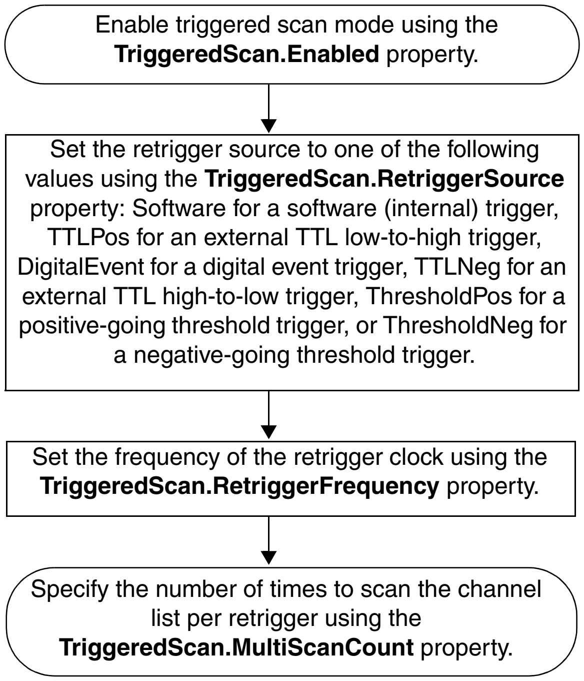
Set Up Triggers


Using a start trigger and reference trigger for pre-trigger/ post-trigger operations?
Yes
No
Set the start trigger type to one of the following values (if supported by your device) using the **AnalogInputSubsystem.Trigger.TriggerType** property: Software for a software (internal) trigger, TTLPos for an external TTL low-to-high trigger, DigitalEvent for a digital event trigger, TTLNeg for an external TTL high-to-low trigger, ThresholdPos for a positive-going threshold trigger, or ThresholdNeg for a negative-going threshold trigger.
Using threshold trigger for the start trigger?
Yes
No
Set the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger. ThresholdTriggerChannel** property.
Set the level of the threshold trigger using the **AnalogInputSubsystem.Trigger.Level** property.
Set the reference trigger source to one of the following values using the **AnalogInputSubsystem.ReferenceTrigger.TriggerType** property: Software for a software (internal) trigger, TTLPos for an external TTL low-to-high trigger, DigitalEvent for a digital event trigger, TTLNeg for an external TTL high-to-low trigger, ThresholdPos for a positive-going threshold trigger, ThresholdNeg for a negative-going threshold trigger, or Sync Bus for a Sync Bus trigger. This trigger source stops pre-trigger acquisition, if in progress, and starts post-trigger acquisition.
Using threshold trigger for the reference trigger?
Yes
No
Set the channel to use for the threshold trigger using the **AnalogInputSubsystem.ReferenceTrigger. ThresholdTriggerChannel** property.
Set the level of the threshold trigger using the **AnalogInputSubsystem.ReferenceTrigger.Level** property.
Specify the number of samples to acquire after the reference trigger using the **AnalogInputSubsystem.ReferenceTrigger.PostTriggerScanCount** property.
Go to next page.


Continued from previous page.
Using pre-trigger or about-trigger mode without a reference trigger (legacy device)?
Yes
No
Set the pre-trigger type to one of the following values (if supported by your device) using **AnalogInputSubsystem.Trigger.PreTriggerSource** property: Software for a software (internal) trigger, TTLPos for an external TTL low-to-high trigger, DigitalEvent for a digital event trigger, TTLNeg for an external TTL high-to-low trigger, ThresholdPos for a positive-going threshold trigger, or ThresholdNeg for a negative-going threshold trigger.
Using threshold trigger?
Yes
No
If supported by your device, set the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger. ThresholdTriggerChannel** property.
If supported by your device, set the level of the threshold trigger using the **AnalogInputSubsystem.Trigger.Level** property.
Using post-trigger or about-trigger mode without a reference trigger?
Yes
Set the post-trigger source to one of the following values (if supported by your device) using the **AnalogInputSubsystem.Trigger.TriggerType** property: Software for a software (internal) trigger, TTLPos for an external TTL low-to-high trigger, DigitalEvent for a digital event trigger, TTLNeg for an external TTL high-to-low trigger, ThresholdPos for a positive-going threshold trigger, or ThresholdNeg for a negative-going threshold trigger. This trigger source stops pre-trigger acquisition, if in progress, and starts post-trigger acquisition.
Using threshold trigger?
Yes
No
If supported by your device, set the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger. ThresholdTriggerChannel** property.
If supported by your device, set the level of the threshold trigger using the **AnalogInputSubsystem.Trigger.Level** property.
Go to next page.


Continued from previous page.
Using a trigger to start analog output operations?
Yes
Set the trigger source to one of the following values (if supported by your device) using the **AnalogOutputSubsystem.Trigger.TriggerType** property: Software for a software (internal) trigger, TTLPos for an external TTL low-to-high trigger, DigitalEvent for a digital event trigger, TTLNeg for an external TTL high-to-low trigger, ThresholdPos for a positive-going threshold trigger, or ThresholdNeg for a negative-going threshold trigger.
Using threshold trigger?
Yes
If supported by your device, set the channel to use for the threshold trigger using the **AnalogOutputSubsystem.Trigger.ThresholdTriggerChannel** property.
If supported by your device, set the level of the threshold trigger using the **AnalogOutputSubsystem.Trigger.Level** property.

Set Up Triggered Scan



Enable triggered scan mode using the **TriggeredScan.Enabled** property.

Set the retrigger source to one of the following values using the **TriggeredScan.RetriggerSource** property: Software for a software (internal) trigger, TTLPos for an external TTL low-to-high trigger, DigitalEvent for a digital event trigger, TTLNeg for an external TTL high-to-low trigger, ThresholdPos for a positive-going threshold trigger, or ThresholdNeg for a negative-going threshold trigger.

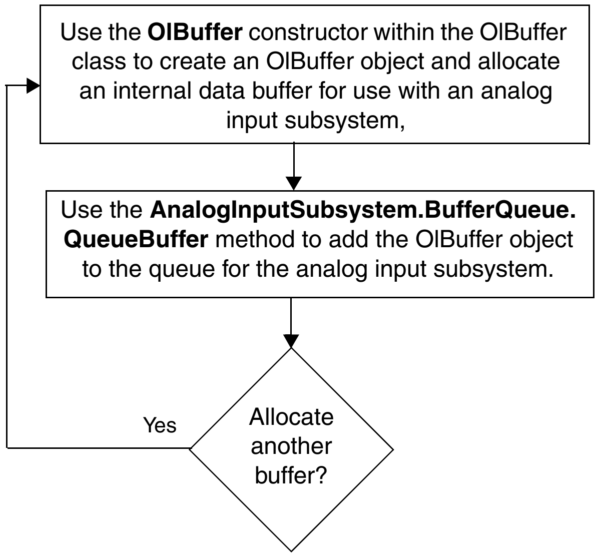
Set the frequency of the retrigger clock using the **TriggeredScan.RetriggerFrequency** property.

The driver sets the actual frequency as closely as possible to the number specified. You can read back the exact frequency after configuring the subsystem using the **RetriggerFrequency** property.

Specify the number of times to scan the channel list per retrigger using the **TriggeredScan.MultiScanCount** property.

The channel is retriggered at the frequency of the retrigger clock.

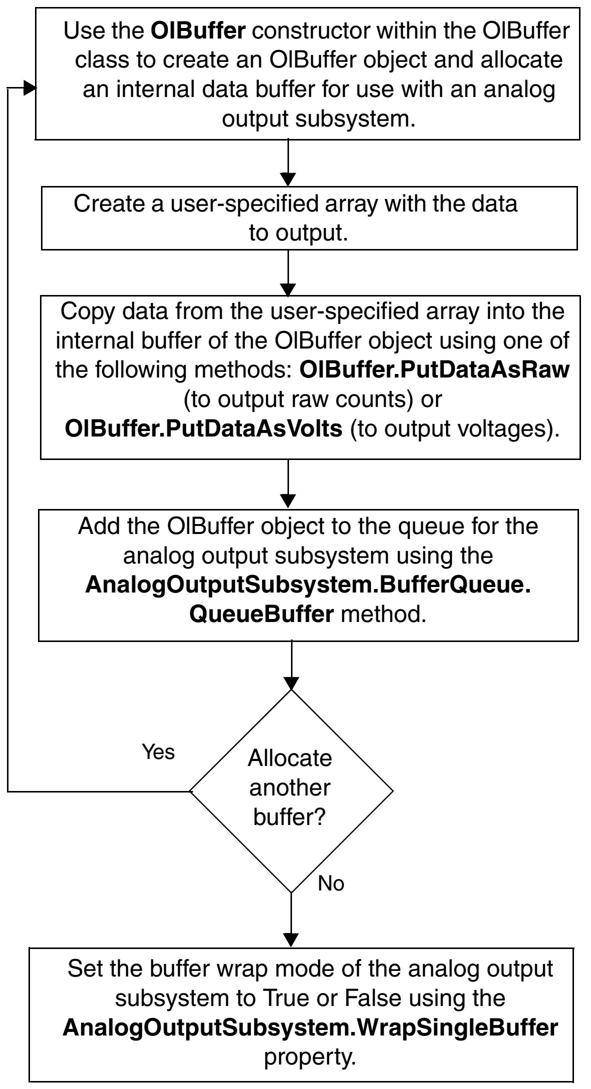
Set Up Input Buffering


Use the **OlBuffer** constructor within the OlBuffer class to create an OlBuffer object and allocate an internal data buffer for use with an analog input subsystem,
Use the **AnalogInputSubsystem.BufferQueue. QueueBuffer** method to add the OlBuffer object to the queue for the analog input subsystem.
Allocate another buffer?
Yes

During this step, you also determine the size of the internal data buffer by specifying the number of samples in the buffer (each sample typically requires 2 bytes).

Continuous input operations require a minimum of two OlBuffer objects.

Set Up Output Buffering



Use the **OlBuffer** constructor within the OlBuffer class to create an OlBuffer object and allocate an internal data buffer for use with an analog output subsystem.

Create a user-specified array with the data to output.

Copy data from the user-specified array into the internal buffer of the OlBuffer object using one of the following methods: **OlBuffer.PutDataAsRaw** (to output raw counts) or **OlBuffer.PutDataAsVolts** (to output voltages).

Add the OlBuffer object to the queue for the analog output subsystem using the **AnalogOutputSubsystem.BufferQueue. QueueBuffer** method.

Set the buffer wrap mode of the analog output subsystem to True or False using the **AnalogOutputSubsystem.WrapSingleBuffer** property.

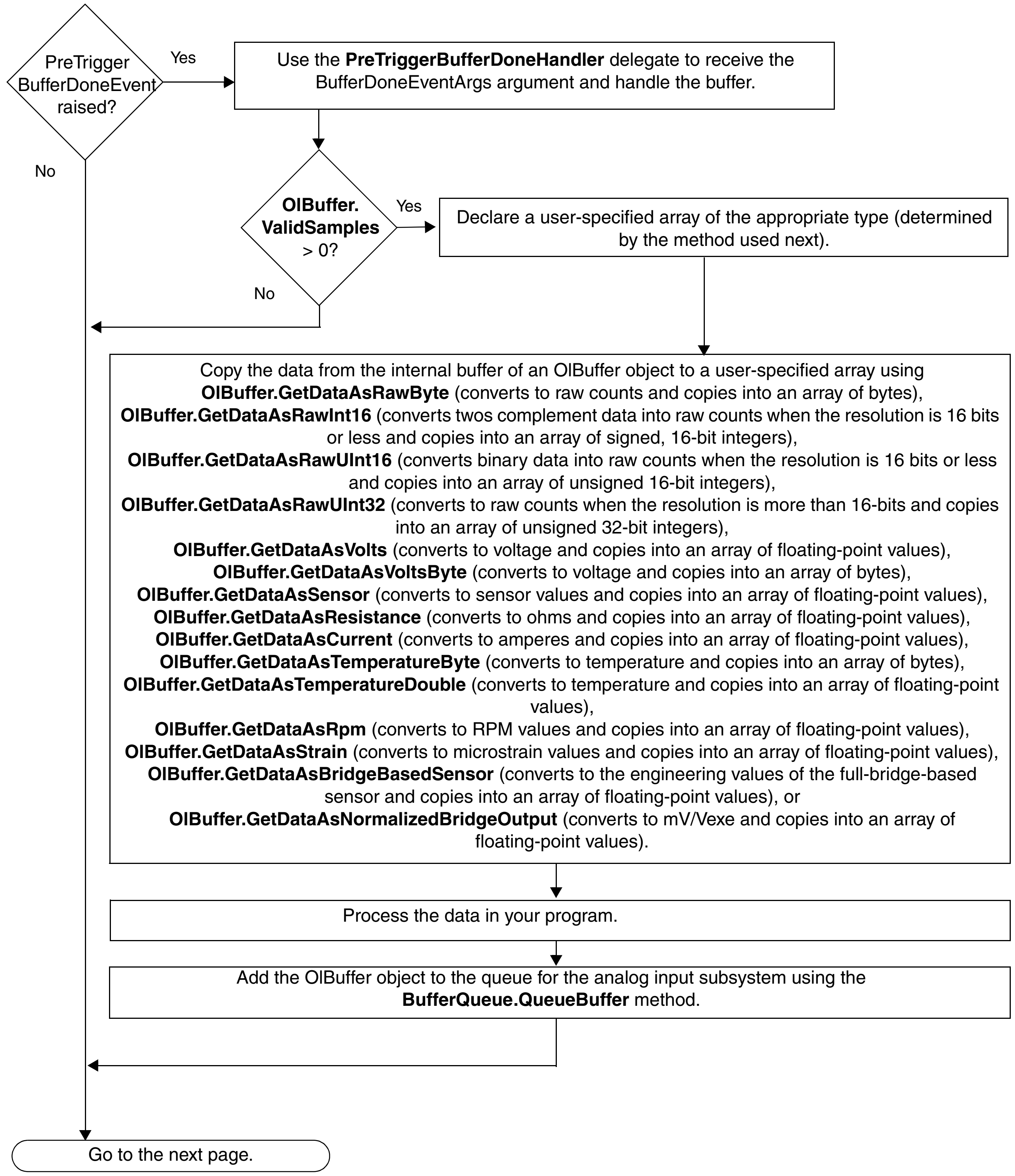
During this step, you also determine the size of the internal buffer by specifying the number of samples in the buffer (each sample typically requires 2 bytes).

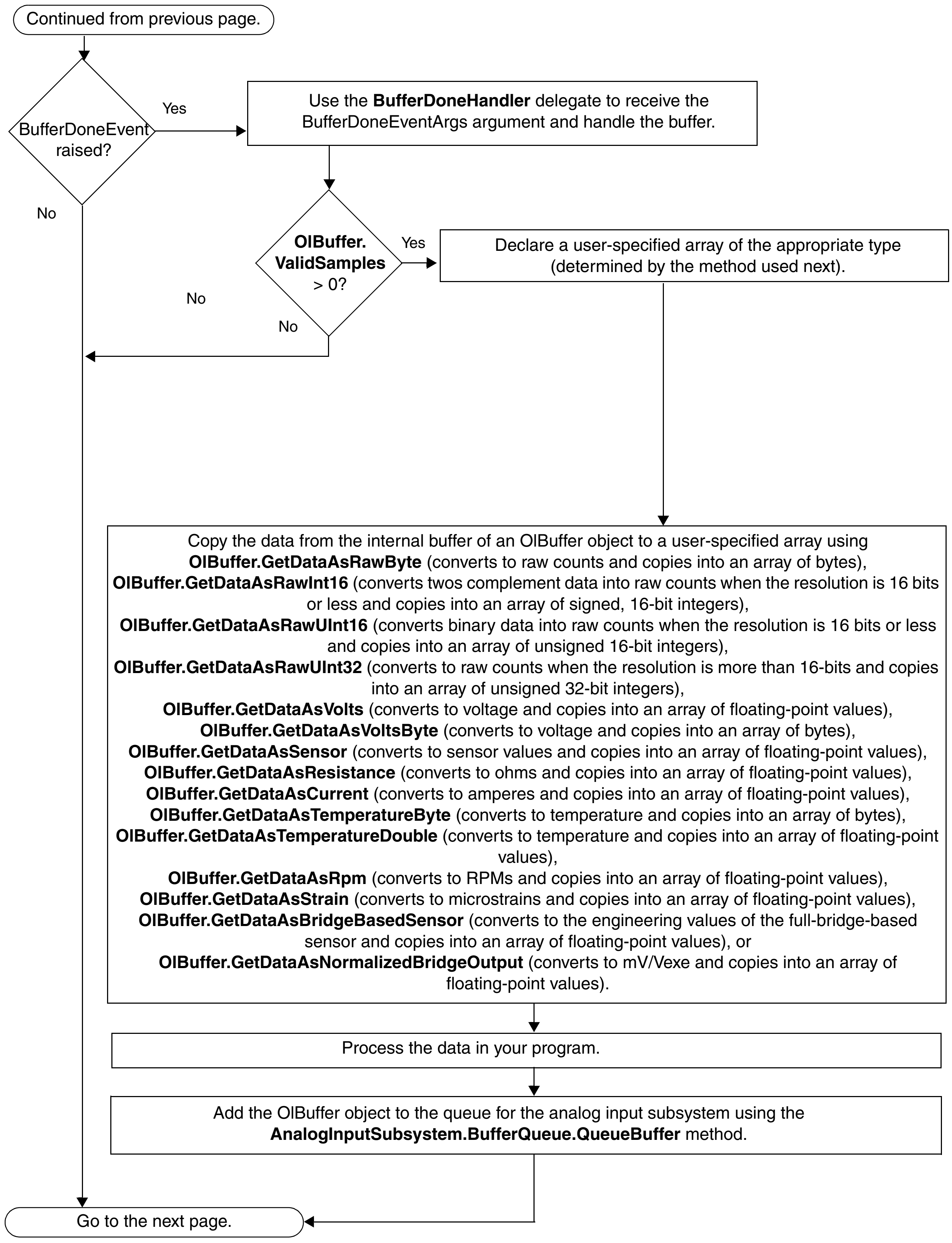
Continuous output operations require two OlBuffer objects if **WrapSingleBuffer** is False (one if **WrapSingleBuffer** is True).

By default, **WrapSingleBuffer** is False. In this state, data is written from the allocated buffers continuously. As each buffer is emptied, a BufferDone event occurs. If no more buffers are available and queued to the subsystem, the operation stops.

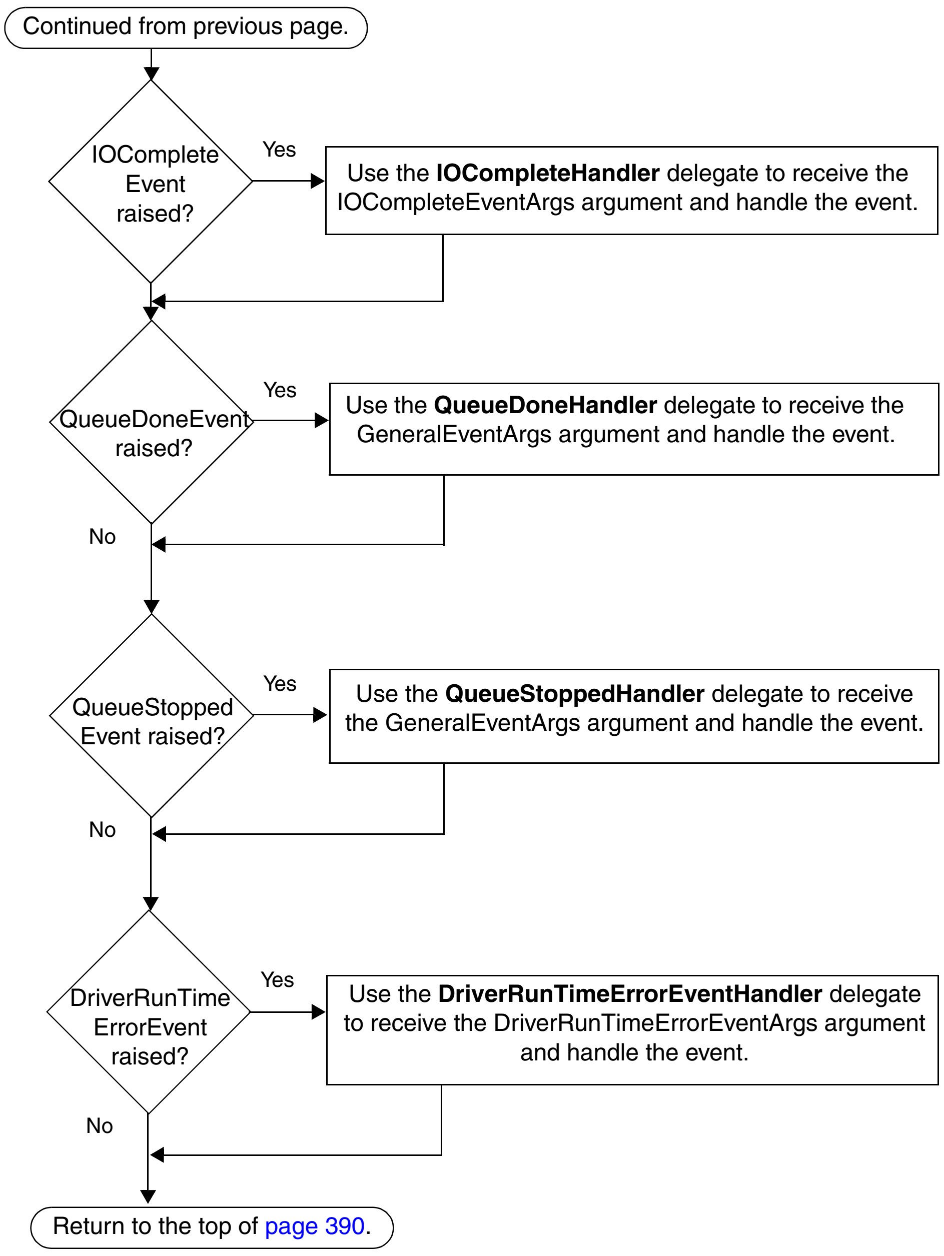
If you set **WrapSingleBuffer** to True, the device driver continuously reuses the first buffer queued to the analog output subsystem. Data from a single output buffer is downloaded to the FIFO of the device (if supported by the device) and is written out starting from the first location of the buffer; when the end of the buffer is reached, the device starts outputting data from the first location of the buffer, and the process repeats.

Deal with Events and Buffers for Input Operations

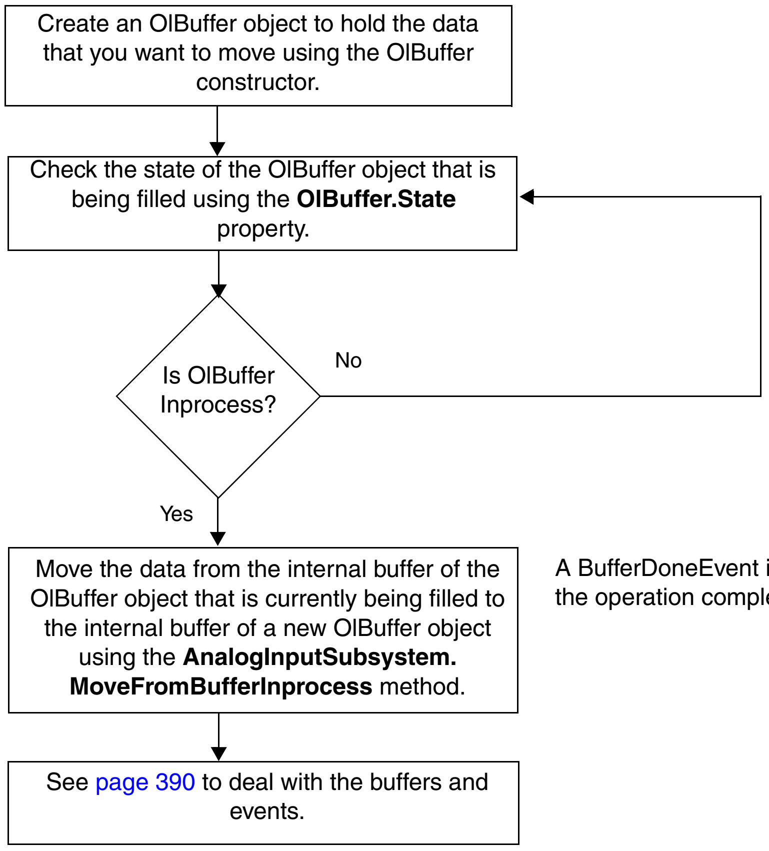

PreTrigger BufferDoneEvent raised?
Yes
No
Use the **PreTriggerBufferDoneHandler** delegate to receive the BufferDoneEventArgs argument and handle the buffer.
OlBuffer. ValidSamples > 0?
Yes
No
Declare a user-specified array of the appropriate type (determined by the method used next).
Copy the data from the internal buffer of an OlBuffer object to a user-specified array using **OlBuffer.GetDataAsRawByte** (converts to raw counts and copies into an array of bytes), **OlBuffer.GetDataAsRawInt16** (converts twos complement data into raw counts when the resolution is 16 bits or less and copies into an array of signed, 16-bit integers), **OlBuffer.GetDataAsRawUInt16** (converts binary data into raw counts when the resolution is 16 bits or less and copies into an array of unsigned 16-bit integers), **OlBuffer.GetDataAsRawUInt32** (converts to raw counts when the resolution is more than 16-bits and copies into an array of unsigned 32-bit integers), **OlBuffer.GetDataAsVolts** (converts to voltage and copies into an array of floating-point values), **OlBuffer.GetDataAsVoltsByte** (converts to voltage and copies into an array of bytes), **OlBuffer.GetDataAsSensor** (converts to sensor values and copies into an array of floating-point values), **OlBuffer.GetDataAsResistance** (converts to ohms and copies into an array of floating-point values), **OlBuffer.GetDataAsCurrent** (converts to amperes and copies into an array of floating-point values), **OlBuffer.GetDataAsTemperatureByte** (converts to temperature and copies into an array of bytes), **OlBuffer.GetDataAsTemperatureDouble** (converts to temperature and copies into an array of floating-point values), **OlBuffer.GetDataAsRpm** (converts to RPM values and copies into an array of floating-point values), **OlBuffer.GetDataAsStrain** (converts to microstrain values and copies into an array of floating-point values), **OlBuffer.GetDataAsBridgeBasedSensor** (converts to the engineering values of the full-bridge-based sensor and copies into an array of floating-point values), or **OlBuffer.GetDataAsNormalizedBridgeOutput** (converts to mV/Vexe and copies into an array of floating-point values).
Process the data in your program.
Add the OlBuffer object to the queue for the analog input subsystem using the **BufferQueue.QueueBuffer** method.
Go to the next page.



Continued from previous page.
BufferDoneEvent raised?
Yes
No
Use the **BufferDoneHandler** delegate to receive the BufferDoneEventArgs argument and handle the buffer.
OlBuffer. ValidSamples > 0?
Yes
No
No
Declare a user-specified array of the appropriate type (determined by the method used next).
Copy the data from the internal buffer of an OlBuffer object to a user-specified array using **OlBuffer.GetDataAsRawByte** (converts to raw counts and copies into an array of bytes), **OlBuffer.GetDataAsRawInt16** (converts twos complement data into raw counts when the resolution is 16 bits or less and copies into an array of signed, 16-bit integers), **OlBuffer.GetDataAsRawUInt16** (converts binary data into raw counts when the resolution is 16 bits or less and copies into an array of unsigned 16-bit integers), **OlBuffer.GetDataAsRawUInt32** (converts to raw counts when the resolution is more than 16-bits and copies into an array of unsigned 32-bit integers), **OlBuffer.GetDataAsVolts** (converts to voltage and copies into an array of floating-point values), **OlBuffer.GetDataAsVoltsByte** (converts to voltage and copies into an array of bytes), **OlBuffer.GetDataAsSensor** (converts to sensor values and copies into an array of floating-point values), **OlBuffer.GetDataAsResistance** (converts to ohms and copies into an array of floating-point values), **OlBuffer.GetDataAsCurrent** (converts to amperes and copies into an array of floating-point values), **OlBuffer.GetDataAsTemperatureByte** (converts to temperature and copies into an array of bytes), **OlBuffer.GetDataAsTemperatureDouble** (converts to temperature and copies into an array of floating-point values), **OlBuffer.GetDataAsRpm** (converts to RPMs and copies into an array of floating-point values), **OlBuffer.GetDataAsStrain** (converts to microstrains and copies into an array of floating-point values), **OlBuffer.GetDataAsBridgeBasedSensor** (converts to the engineering values of the full-bridge-based sensor and copies into an array of floating-point values), or **OlBuffer.GetDataAsNormalizedBridgeOutput** (converts to mV/Vexe and copies into an array of floating-point values).
Process the data in your program.
Add the OlBuffer object to the queue for the analog input subsystem using the **AnalogInputSubsystem.BufferQueue.QueueBuffer** method.
Go to the next page.

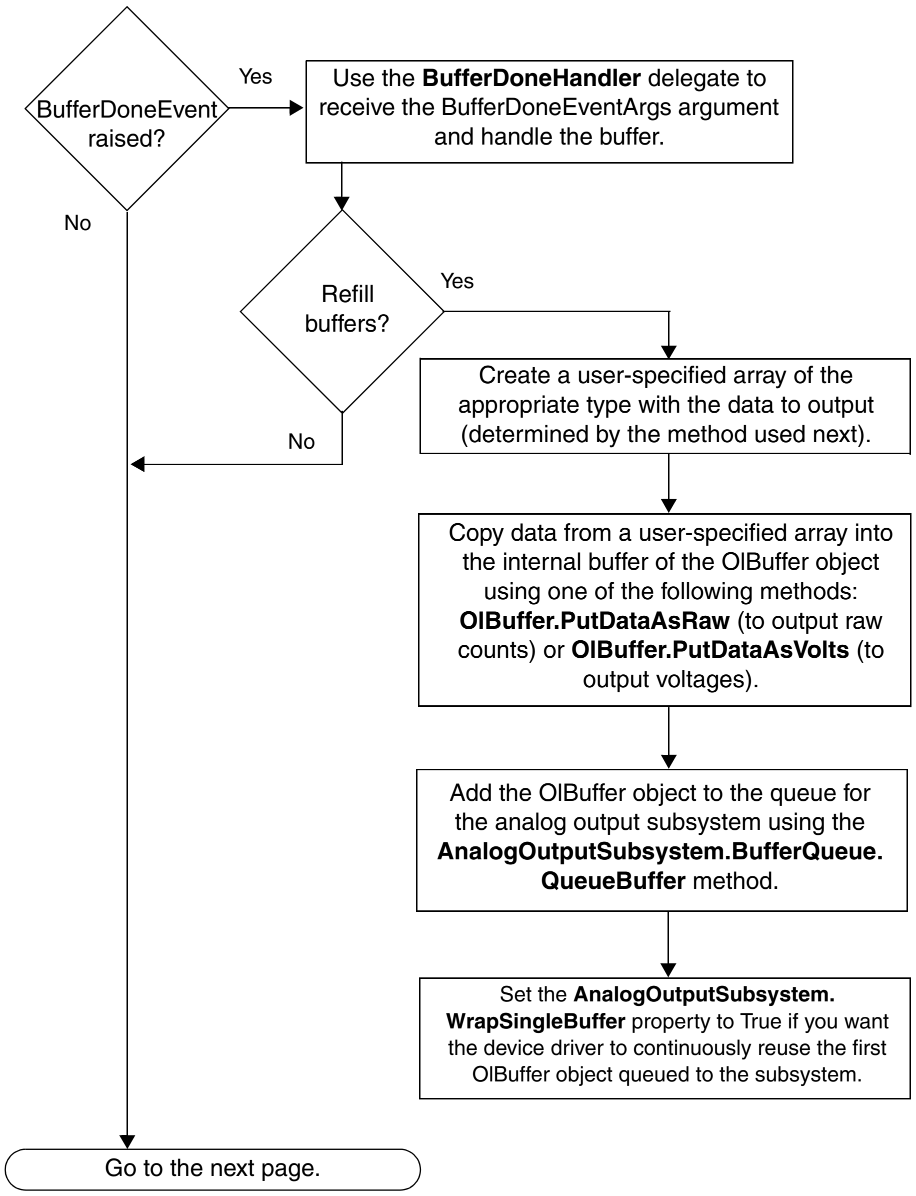

Continued from previous page.
IOComplete Event raised?
Yes
Use the **IOCompleteHandler** delegate to receive the IOCompleteEventArgs argument and handle the event.
QueueDoneEvent raised?
Yes
Use the **QueueDoneHandler** delegate to receive the GeneralEventArgs argument and handle the event.
No
QueueStopped Event raised?
Yes
Use the **QueueStoppedHandler** delegate to receive the GeneralEventArgs argument and handle the event.
No
DriverRunTime ErrorEvent raised?
Yes
Use the **DriverRunTimeErrorEventHandler** delegate to receive the DriverRunTimeErrorEventArgs argument and handle the event.
No
Return to the top of page 390.

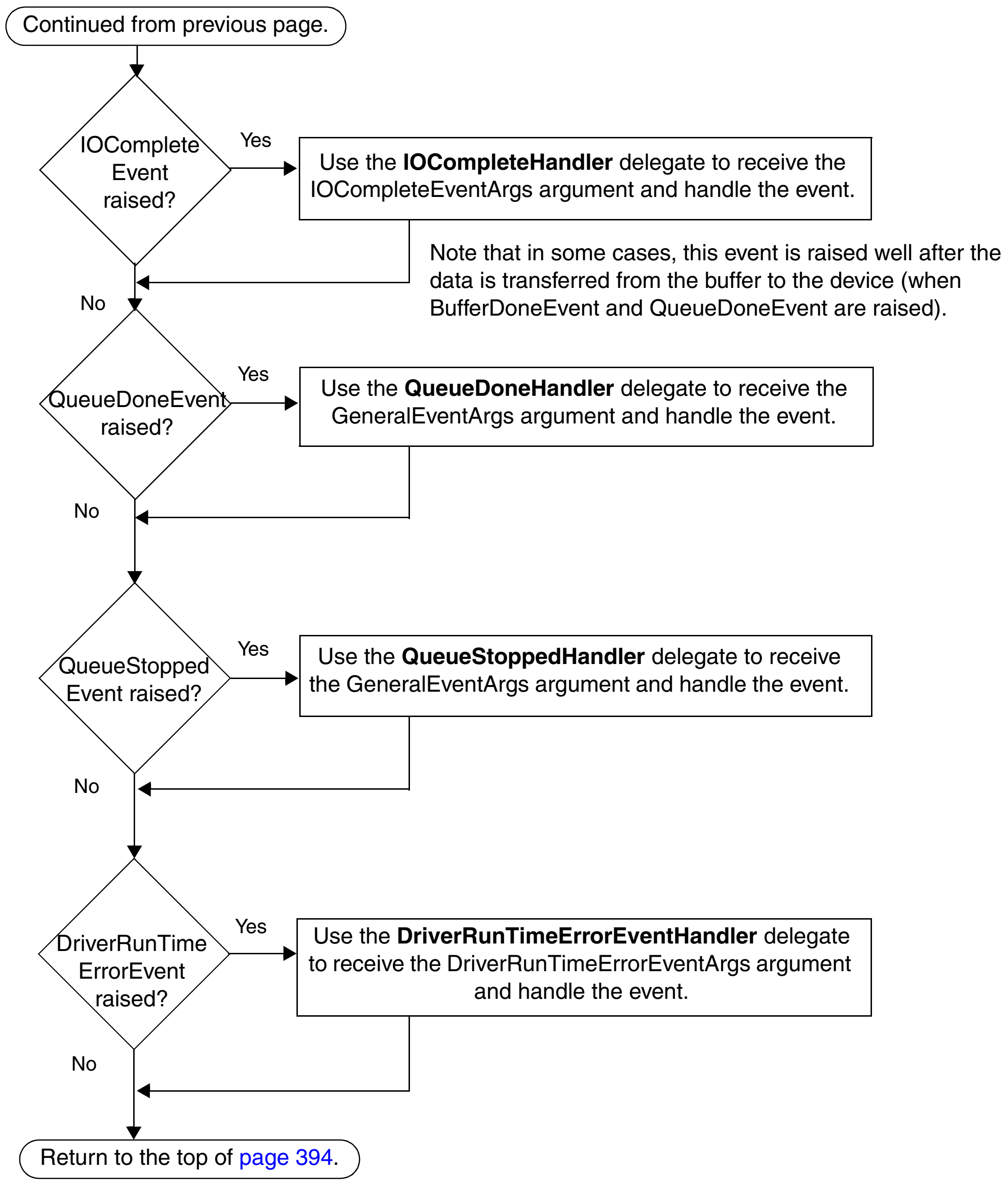
Transfer Data from an Inprocess Buffer


Create an OlBuffer object to hold the data that you want to move using the OlBuffer constructor.
Check the state of the OlBuffer object that is being filled using the **OlBuffer.State** property.
Is OlBuffer Inprocess?
No
Yes
Move the data from the internal buffer of the OlBuffer object that is currently being filled to the internal buffer of a new OlBuffer object using the **AnalogInputSubsystem. MoveFromBufferInprocess** method.
See page 390 to deal with the buffers and events.

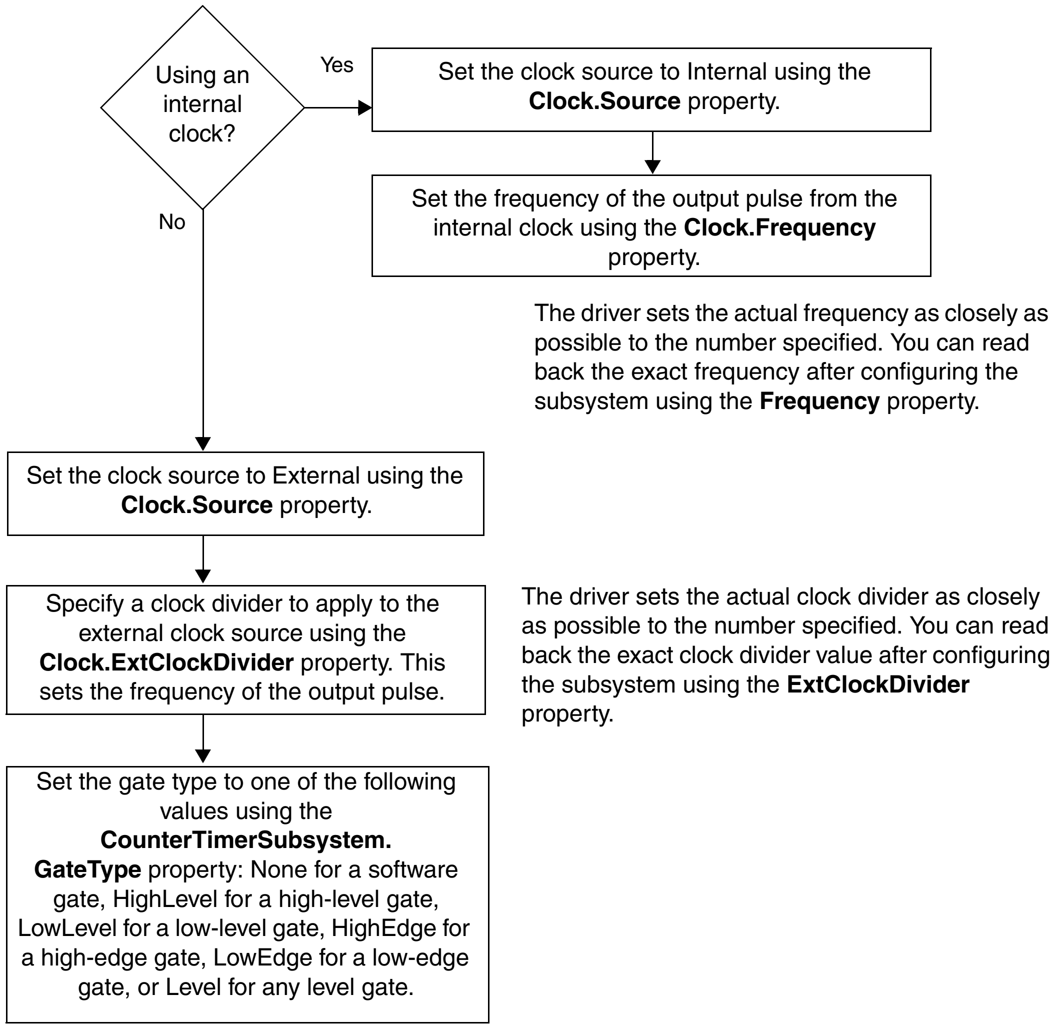
A BufferDoneEvent is generated when the operation completes.

Deal with Events and Buffers for Output Operations

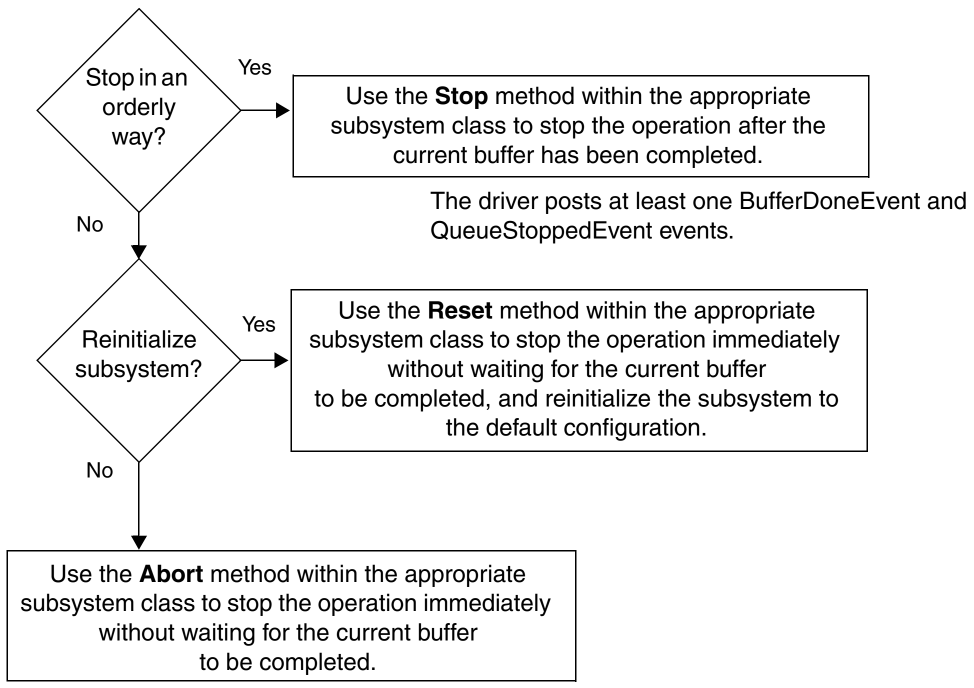

BufferDoneEvent raised?
Yes
Use the **BufferDoneHandler** delegate to receive the BufferDoneEventArgs argument and handle the buffer.
No
Refill buffers?
Yes
Create a user-specified array of the appropriate type with the data to output (determined by the method used next).
No
Copy data from a user-specified array into the internal buffer of the OlBuffer object using one of the following methods: **OlBuffer.PutDataAsRaw** (to output raw counts) or **OlBuffer.PutDataAsVolts** (to output voltages).
Add the OlBuffer object to the queue for the analog output subsystem using the **AnalogOutputSubsystem.BufferQueue. QueueBuffer** method.
Set the **AnalogOutputSubsystem. WrapSingleBuffer** property to True if you want the device driver to continuously reuse the first OlBuffer object queued to the subsystem.
Go to the next page.


Continued from previous page.
IOComplete Event raised?
Yes
No
Use the **IOCompleteHandler** delegate to receive the IOCompleteEventArgs argument and handle the event.
Note that in some cases, this event is raised well after the data is transferred from the buffer to the device (when BufferDoneEvent and QueueDoneEvent are raised).
QueueDoneEvent raised?
Yes
No
Use the **QueueDoneHandler** delegate to receive the GeneralEventArgs argument and handle the event.
QueueStopped Event raised?
Yes
No
Use the **QueueStoppedHandler** delegate to receive the GeneralEventArgs argument and handle the event.
DriverRunTime ErrorEvent raised?
Yes
No
Use the **DriverRunTimeErrorEventHandler** delegate to receive the DriverRunTimeErrorEventArgs argument and handle the event.
Return to the top of page 394.

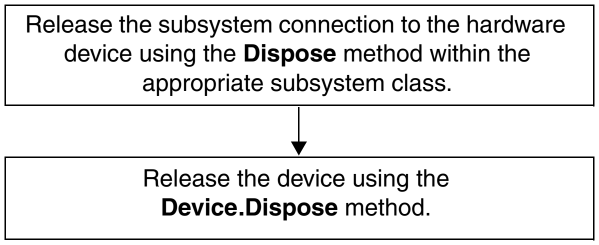
Set Clocks and Gates for Counter/Timer Operations


Using an internal clock?
Yes
Set the clock source to Internal using the **Clock.Source** property.
Set the frequency of the output pulse from the internal clock using the **Clock.Frequency** property.
The driver sets the actual frequency as closely as possible to the number specified. You can read back the exact frequency after configuring the subsystem using the **Frequency** property.
No
Set the clock source to External using the **Clock.Source** property.
Specify a clock divider to apply to the external clock source using the **Clock.ExtClockDivider** property. This sets the frequency of the output pulse.
The driver sets the actual clock divider as closely as possible to the number specified. You can read back the exact clock divider value after configuring the subsystem using the **ExtClockDivider** property.
Set the gate type to one of the following values using the **CounterTimerSubsystem. GateType** property: None for a software gate, HighLevel for a high-level gate, LowLevel for a low-level gate, HighEdge for a high-edge gate, LowEdge for a low-edge gate, or Level for any level gate.

Stop the Operation


Stop in an orderly way?
Yes
Use the **Stop** method within the appropriate subsystem class to stop the operation after the current buffer has been completed.
The driver posts at least one BufferDoneEvent and QueueStoppedEvent events.
No
Reinitialize subsystem?
Yes
Use the **Reset** method within the appropriate subsystem class to stop the operation immediately without waiting for the current buffer to be completed, and reinitialize the subsystem to the default configuration.
No
Use the **Abort** method within the appropriate subsystem class to stop the operation immediately without waiting for the current buffer to be completed.

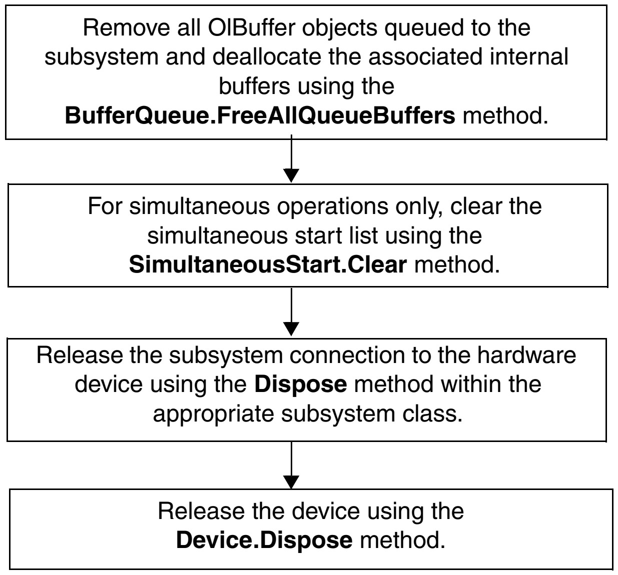
Clean Up Single-Value I/O Operations



Release the subsystem connection to the hardware device using the **Dispose** method within the appropriate subsystem class.

Release the device using the **Device.Dispose** method.

Clean Up Buffered I/O Operations



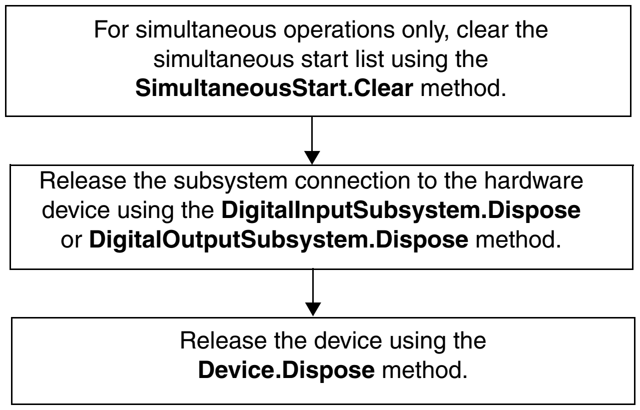
Remove all OlBuffer objects queued to the subsystem and deallocate the associated internal buffers using the **BufferQueue.FreeAllQueueBuffers** method.

For simultaneous operations only, clear the simultaneous start list using the **SimultaneousStart.Clear** method.

Release the subsystem connection to the hardware device using the **Dispose** method within the appropriate subsystem class.

Release the device using the **Device.Dispose** method.

Clean Up Digital I/O Operations

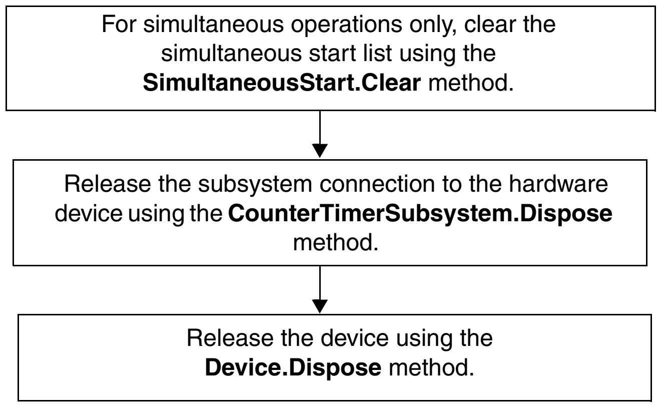


For simultaneous operations only, clear the simultaneous start list using the **SimultaneousStart.Clear** method.

Release the subsystem connection to the hardware device using the **DigitalInputSubsystem.Dispose** or **DigitalOutputSubsystem.Dispose** method.

Release the device using the **Device.Dispose** method.

Clean Up Counter/Timer Operations

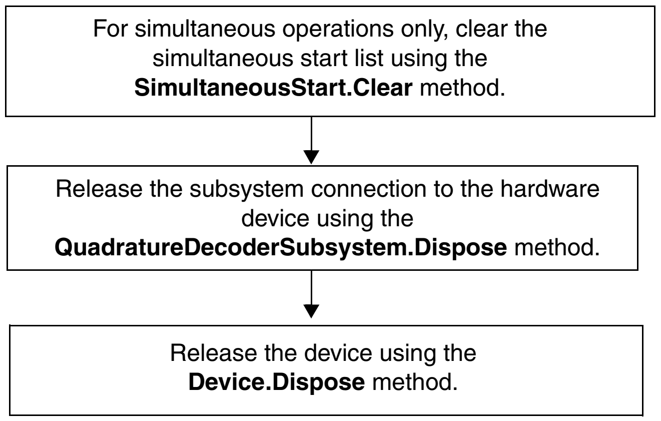


For simultaneous operations only, clear the simultaneous start list using the **SimultaneousStart.Clear** method.

Release the subsystem connection to the hardware device using the **CounterTimerSubsystem.Dispose** method.

Release the device using the **Device.Dispose** method.

Clean Up Quadrature Decoder Operations

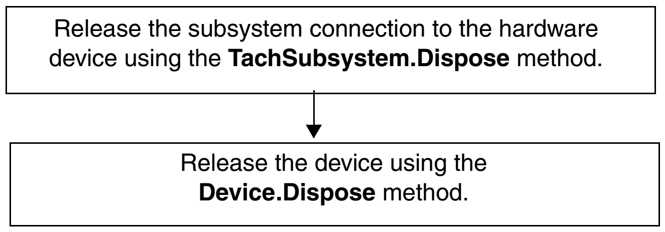


For simultaneous operations only, clear the simultaneous start list using the **SimultaneousStart.Clear** method.

Release the subsystem connection to the hardware device using the **QuadratureDecoderSubsystem.Dispose** method.

Release the device using the **Device.Dispose** method.

Clean Up Tachometer Operations



Release the subsystem connection to the hardware device using the **TachSubsystem.Dispose** method.

Release the device using the **Device.Dispose** method.

6

Programming Flowcharts for the OpenLayers.DeviceCollection Namespace

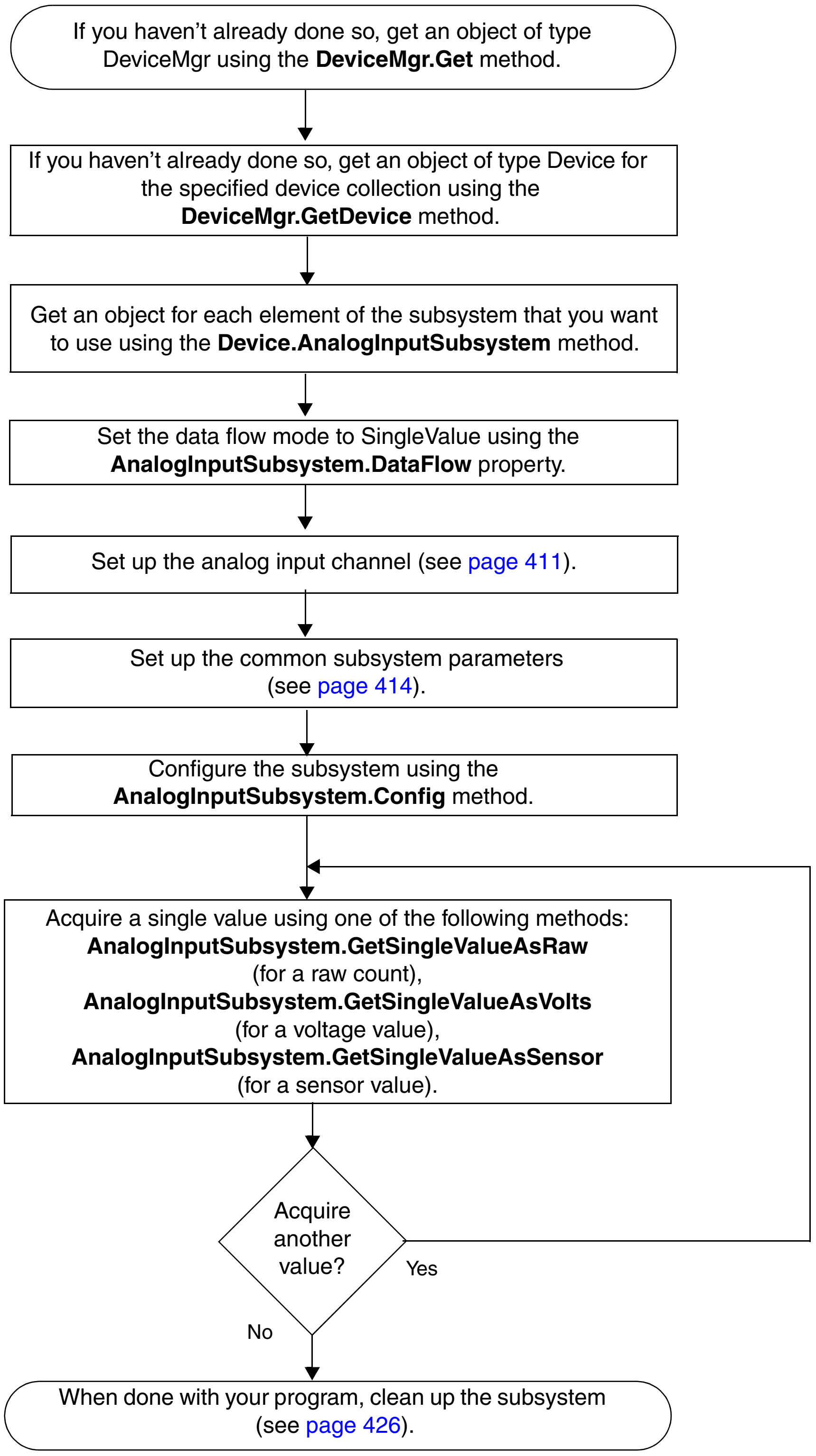
The flowcharts presented in the remainder of this chapter show how to perform typical input/output operations using the OpenLayers.DeviceCollection namespace.

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

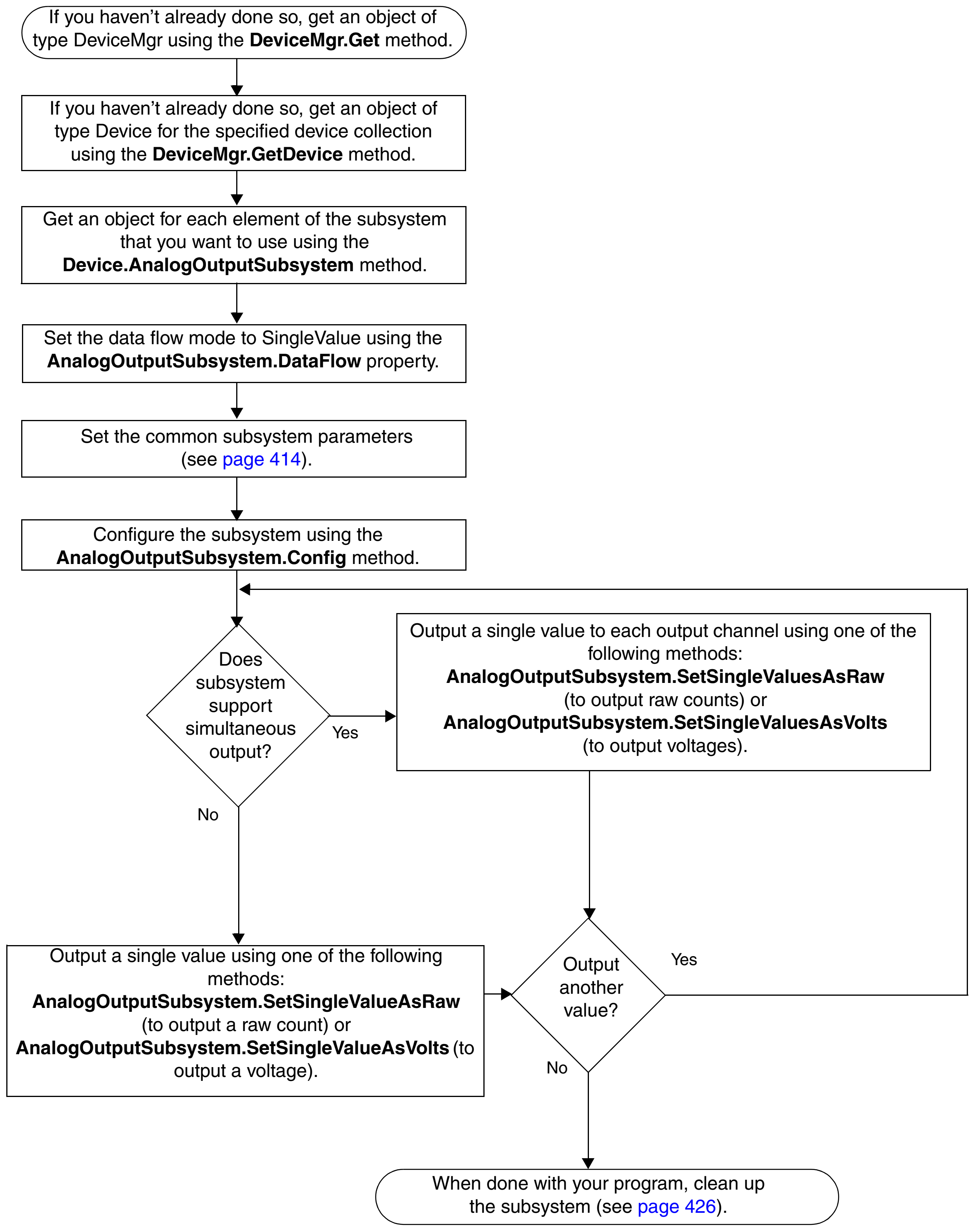
Although the flowcharts do not show error checking, it is recommended that you add exception handling to your program.

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.

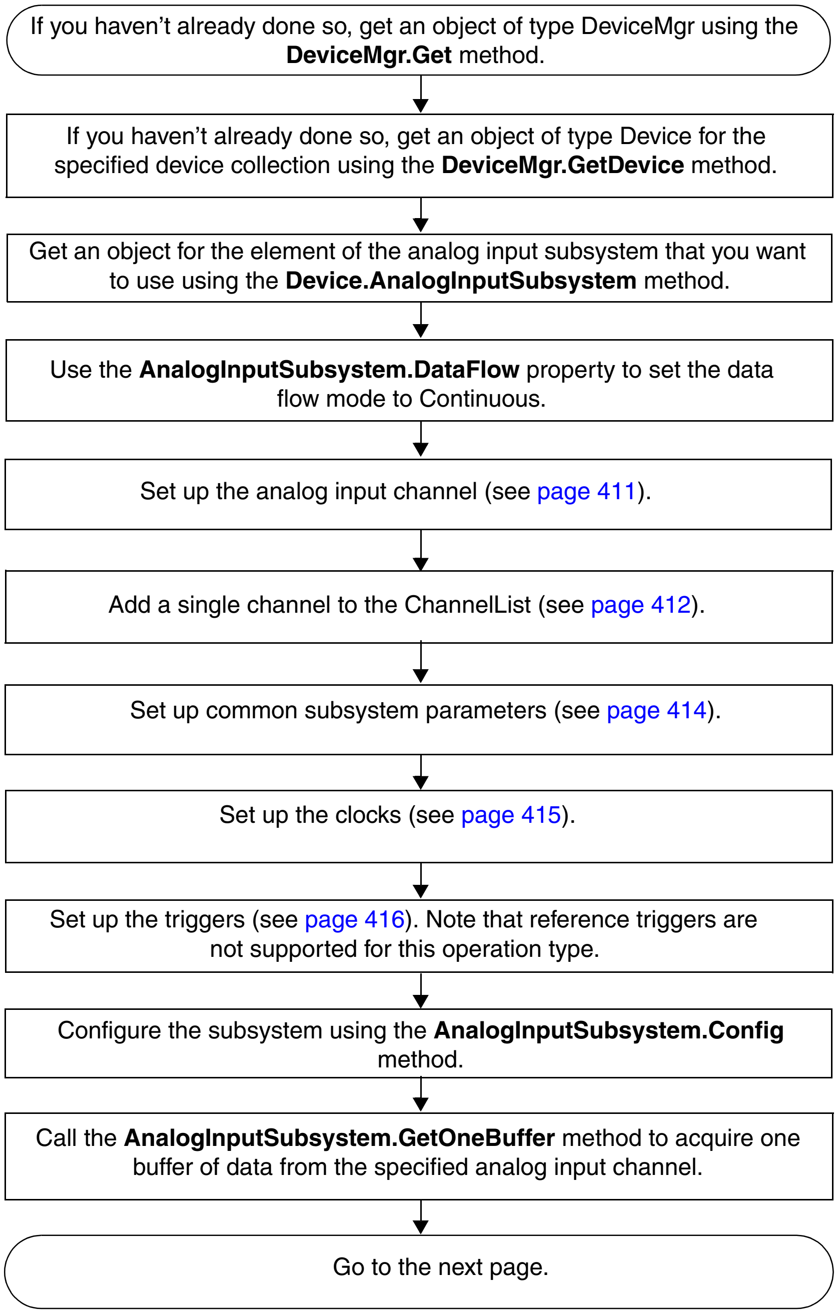
Single-Value Analog Input Operations


If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified device collection using the **DeviceMgr.GetDevice** method.
Get an object for each element of the subsystem that you want to use using the **Device.AnalogInputSubsystem** method.
Set the data flow mode to SingleValue using the **AnalogInputSubsystem.DataFlow** property.
Set up the analog input channel (see page 411).
Set up the common subsystem parameters (see page 414).
Configure the subsystem using the **AnalogInputSubsystem.Config** method.
Acquire a single value using one of the following methods: **AnalogInputSubsystem.GetSingleValueAsRaw** (for a raw count), **AnalogInputSubsystem.GetSingleValueAsVolts** (for a voltage value), **AnalogInputSubsystem.GetSingleValueAsSensor** (for a sensor value).
Acquire another value?
Yes
No
When done with your program, clean up the subsystem (see page 426).

Single-Value Analog Output Operations


If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified device collection using the **DeviceMgr.GetDevice** method.
Get an object for each element of the subsystem that you want to use using the **Device.AnalogOutputSubsystem** method.
Set the data flow mode to SingleValue using the **AnalogOutputSubsystem.DataFlow** property.
Set the common subsystem parameters (see page 414).
Configure the subsystem using the **AnalogOutputSubsystem.Config** method.
Does subsystem support simultaneous output?
Yes
Output a single value to each output channel using one of the following methods: **AnalogOutputSubsystem.SetSingleValuesAsRaw** (to output raw counts) or **AnalogOutputSubsystem.SetSingleValuesAsVolts** (to output voltages).
No
Output a single value using one of the following methods: **AnalogOutputSubsystem.SetSingleValueAsRaw** (to output a raw count) or **AnalogOutputSubsystem.SetSingleValueAsVolts** (to output a voltage).
Output another value?
Yes
No
When done with your program, clean up the subsystem (see page 426).

Continuous Analog Input Operations - One Buffer



If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.

If you haven't already done so, get an object of type Device for the specified device collection using the **DeviceMgr.GetDevice** method.

Get an object for the element of the analog input subsystem that you want to use using the **Device.AnalogInputSubsystem** method.

Use the **AnalogInputSubsystem.DataFlow** property to set the data flow mode to Continuous.

Set up the analog input channel (see page 411).

Add a single channel to the ChannelList (see page 412).

Set up common subsystem parameters (see page 414).

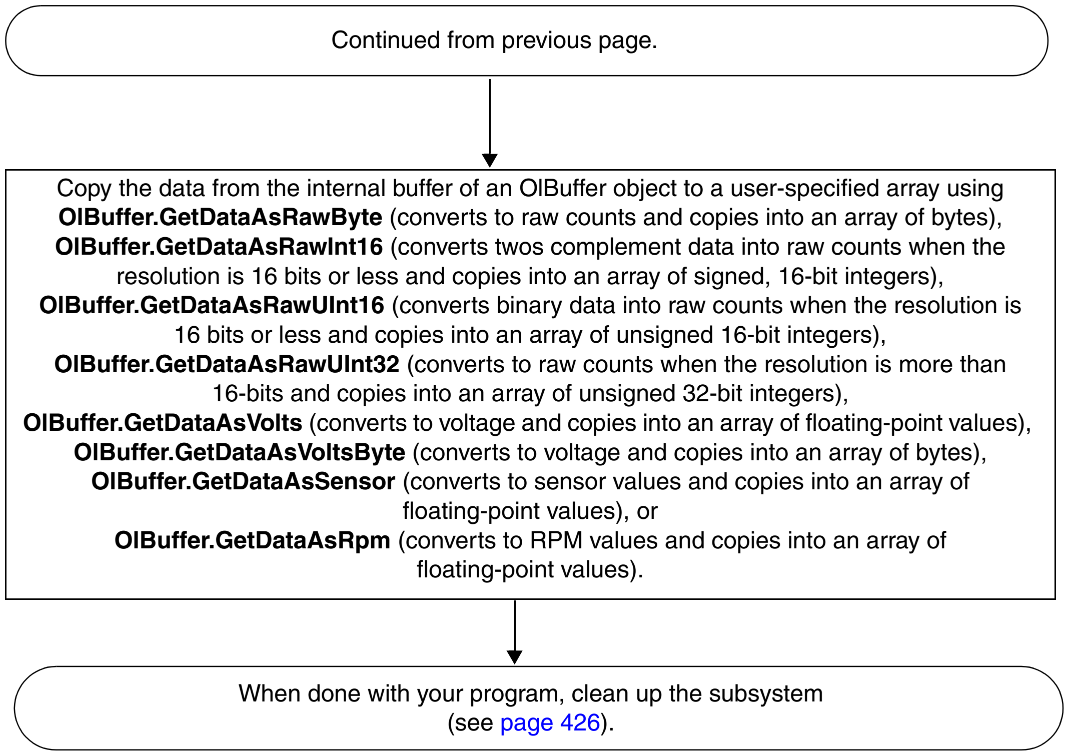
Set up the clocks (see page 415).

Set up the triggers (see page 416). Note that reference triggers are not supported for this operation type.

Configure the subsystem using the **AnalogInputSubsystem.Config** method.

Call the **AnalogInputSubsystem.GetOneBuffer** method to acquire one buffer of data from the specified analog input channel.

Go to the next page.

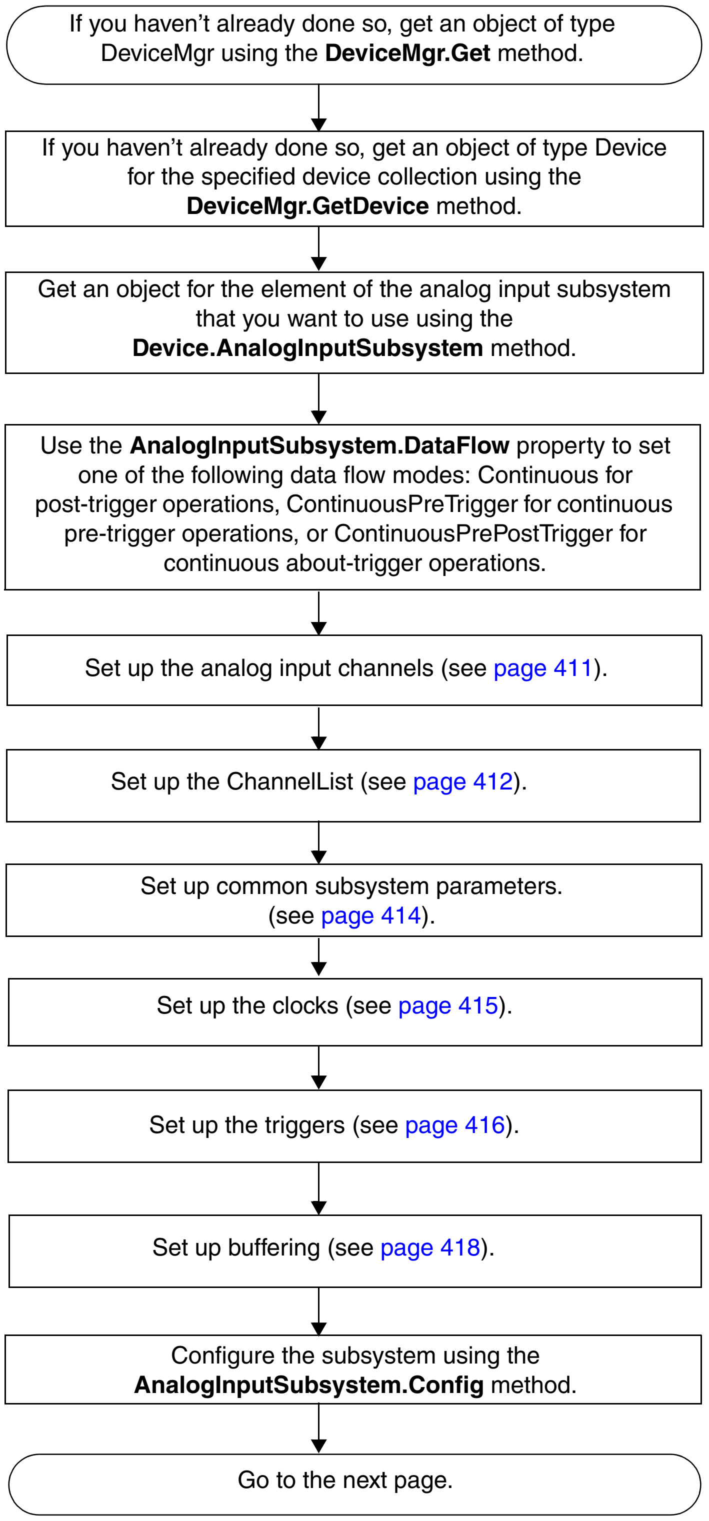


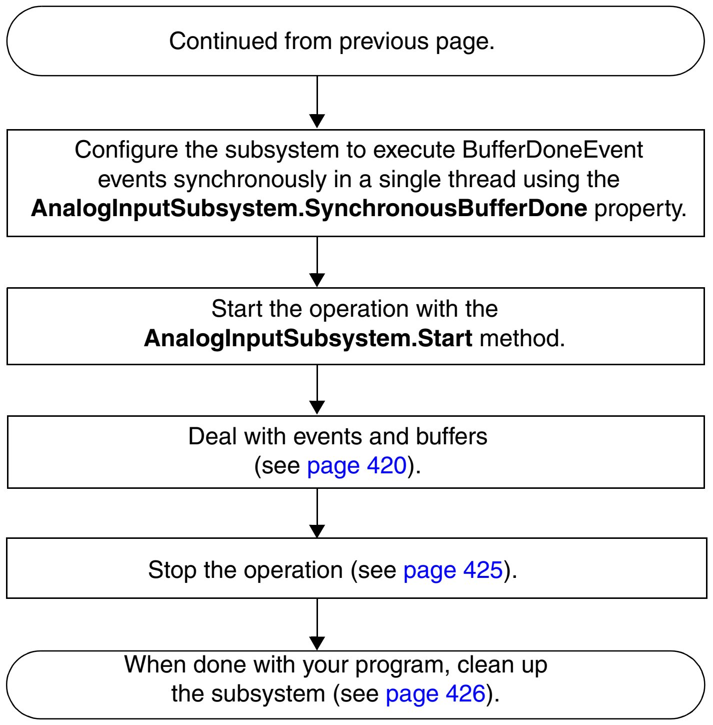
Continued from previous page.

Copy the data from the internal buffer of an OlBuffer object to a user-specified array using **OlBuffer.GetDataAsRawByte** (converts to raw counts and copies into an array of bytes), **OlBuffer.GetDataAsRawInt16** (converts twos complement data into raw counts when the resolution is 16 bits or less and copies into an array of signed, 16-bit integers), **OlBuffer.GetDataAsRawUInt16** (converts binary data into raw counts when the resolution is 16 bits or less and copies into an array of unsigned 16-bit integers), **OlBuffer.GetDataAsRawUInt32** (converts to raw counts when the resolution is more than 16-bits and copies into an array of unsigned 32-bit integers), **OlBuffer.GetDataAsVolts** (converts to voltage and copies into an array of floating-point values), **OlBuffer.GetDataAsVoltsByte** (converts to voltage and copies into an array of bytes), **OlBuffer.GetDataAsSensor** (converts to sensor values and copies into an array of floating-point values), or **OlBuffer.GetDataAsRpm** (converts to RPM values and copies into an array of floating-point values).

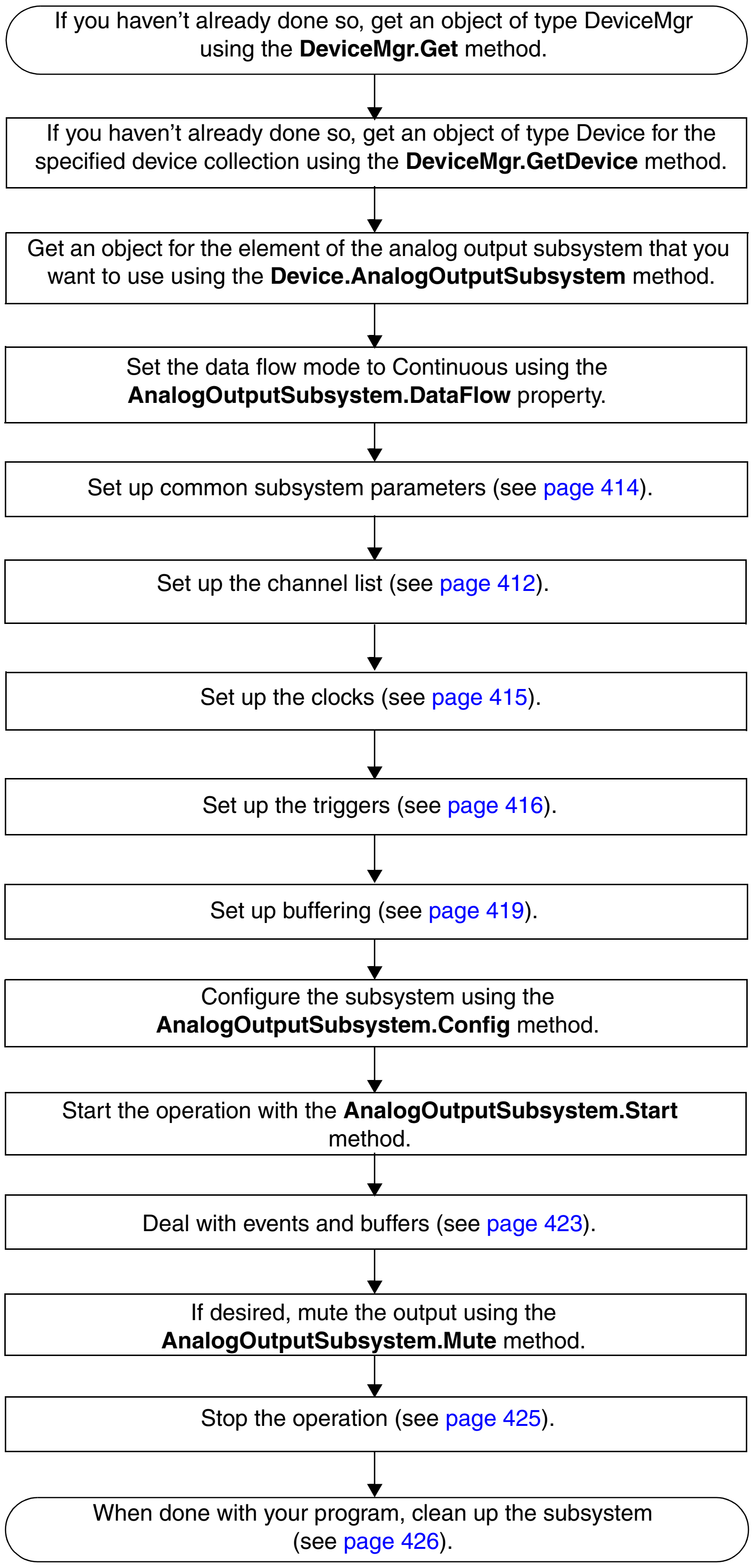
When done with your program, clean up the subsystem (see page 426).

Continuous Analog Input Operations - Multiple Buffers

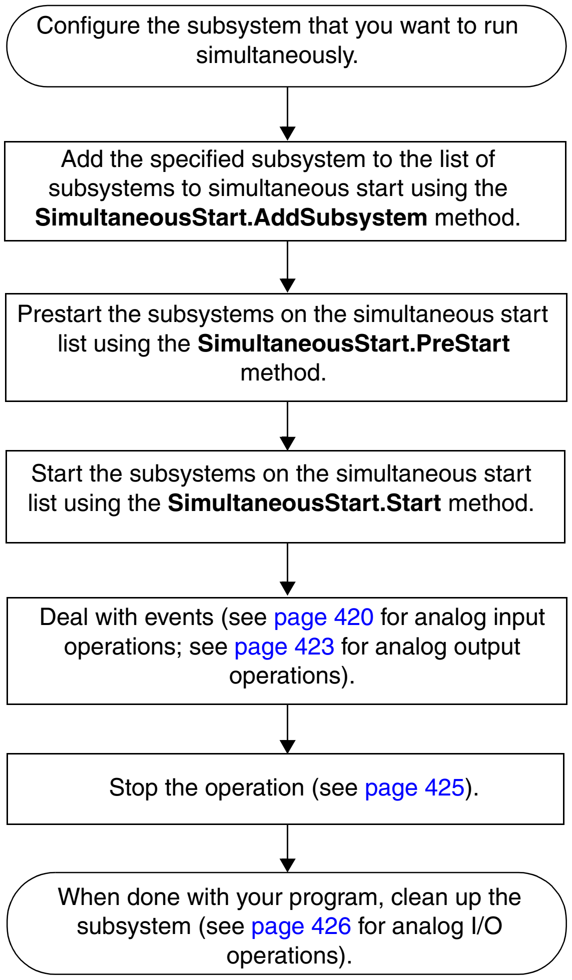

If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified device collection using the **DeviceMgr.GetDevice** method.
Get an object for the element of the analog input subsystem that you want to use using the **Device.AnalogInputSubsystem** method.
Use the **AnalogInputSubsystem.DataFlow** property to set one of the following data flow modes: Continuous for post-trigger operations, ContinuousPreTrigger for continuous pre-trigger operations, or ContinuousPrePostTrigger for continuous about-trigger operations.
Set up the analog input channels (see page 411).
Set up the ChannelList (see page 412).
Set up common subsystem parameters. (see page 414).
Set up the clocks (see page 415).
Set up the triggers (see page 416).
Set up buffering (see page 418).
Configure the subsystem using the **AnalogInputSubsystem.Config** method.
Go to the next page.


Continued from previous page.
Configure the subsystem to execute BufferDoneEvent events synchronously in a single thread using the **AnalogInputSubsystem.SynchronousBufferDone** property.
Start the operation with the **AnalogInputSubsystem.Start** method.
Deal with events and buffers (see page 420).
Stop the operation (see page 425).
When done with your program, clean up the subsystem (see page 426).

Continuous Analog Output Operations

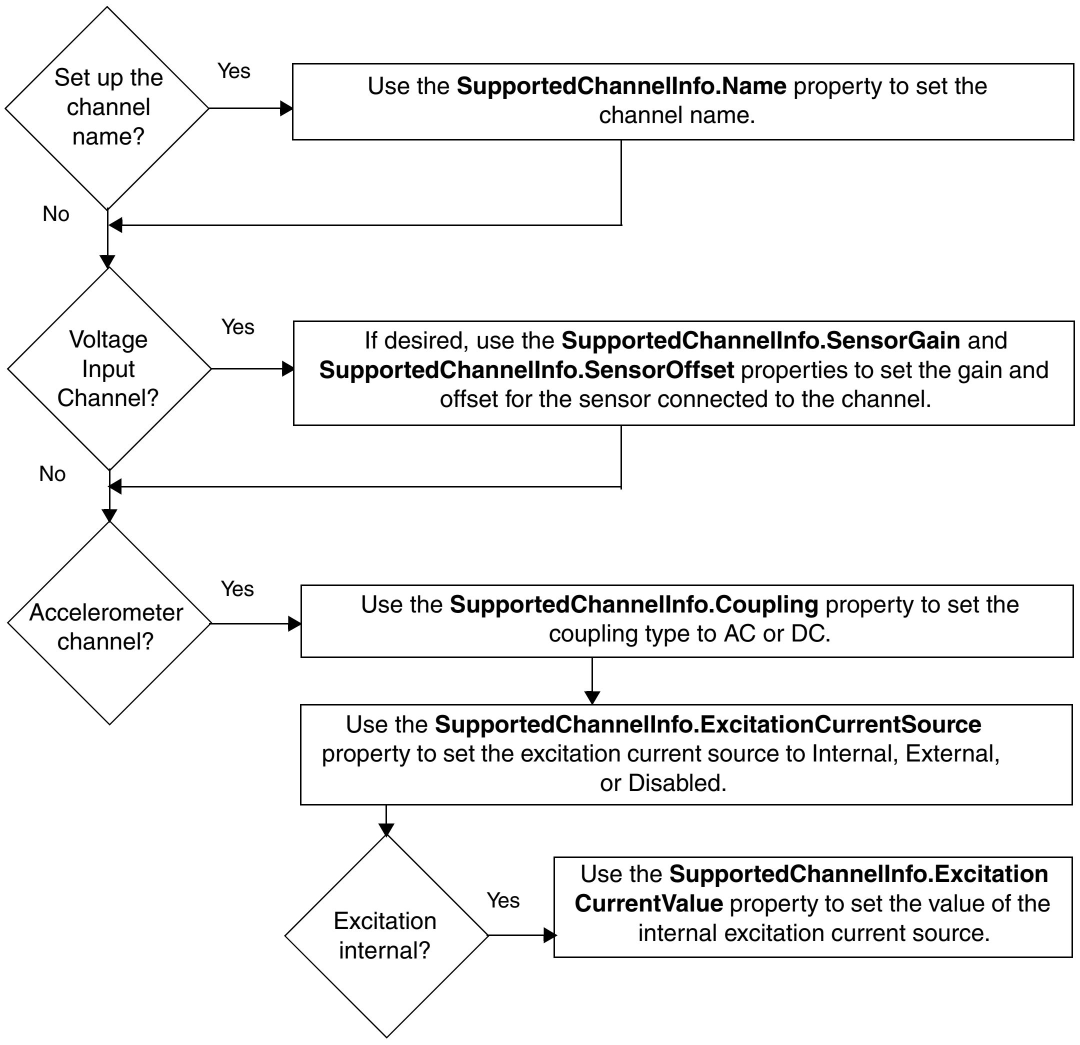

If you haven't already done so, get an object of type DeviceMgr using the **DeviceMgr.Get** method.
If you haven't already done so, get an object of type Device for the specified device collection using the **DeviceMgr.GetDevice** method.
Get an object for the element of the analog output subsystem that you want to use using the **Device.AnalogOutputSubsystem** method.
Set the data flow mode to Continuous using the **AnalogOutputSubsystem.DataFlow** property.
Set up common subsystem parameters (see page 414).
Set up the channel list (see page 412).
Set up the clocks (see page 415).
Set up the triggers (see page 416).
Set up buffering (see page 419).
Configure the subsystem using the **AnalogOutputSubsystem.Config** method.
Start the operation with the **AnalogOutputSubsystem.Start** method.
Deal with events and buffers (see page 423).
If desired, mute the output using the **AnalogOutputSubsystem.Mute** method.
Stop the operation (see page 425).
When done with your program, clean up the subsystem (see page 426).

Simultaneously Starting Subsystems

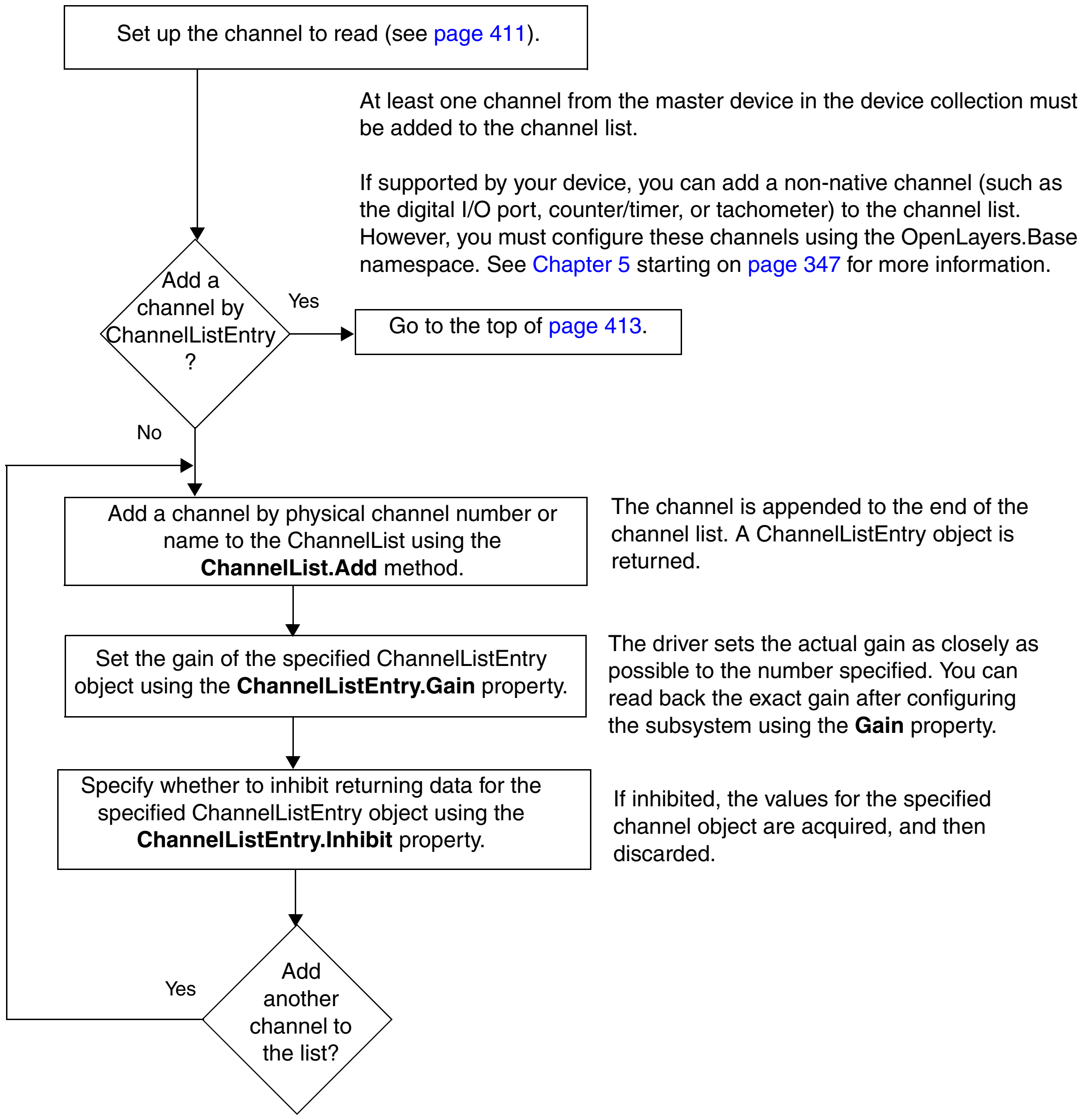

Configure the subsystem that you want to run simultaneously.
Add the specified subsystem to the list of subsystems to simultaneous start using the **SimultaneousStart.AddSubsystem** method.
Prestart the subsystems on the simultaneous start list using the **SimultaneousStart.PreStart** method.
Start the subsystems on the simultaneous start list using the **SimultaneousStart.Start** method.
Deal with events (see page 420 for analog input operations; see page 423 for analog output operations).
Stop the operation (see page 425).
When done with your program, clean up the subsystem (see page 426 for analog I/O operations).

See the previous flow diagrams in this chapter; you cannot perform single-value operations simultaneously on multiplexed A/D modules.

Set Up Analog Input Channels


Set up the channel name?
Yes
Use the **SupportedChannelInfo.Name** property to set the channel name.
No
Voltage Input Channel?
Yes
If desired, use the **SupportedChannelInfo.SensorGain** and **SupportedChannelInfo.SensorOffset** properties to set the gain and offset for the sensor connected to the channel.
No
Accelerometer channel?
Yes
Use the **SupportedChannelInfo.Coupling** property to set the coupling type to AC or DC.
Use the **SupportedChannelInfo.ExcitationCurrentSource** property to set the excitation current source to Internal, External, or Disabled.
Excitation internal?
Yes
Use the **SupportedChannelInfo.ExcitationCurrentValue** property to set the value of the internal excitation current source.

Set Up the ChannelList


Set up the channel to read (see page 411).
Add a channel by ChannelListEntry ?
Yes
Go to the top of page 413.
No
Add a channel by physical channel number or name to the ChannelList using the **ChannelList.Add** method.
Set the gain of the specified ChannelListEntry object using the **ChannelListEntry.Gain** property.
Specify whether to inhibit returning data for the specified ChannelListEntry object using the **ChannelListEntry.Inhibit** property.
Add another channel to the list?
Yes

At least one channel from the master device in the device collection must be added to the channel list.

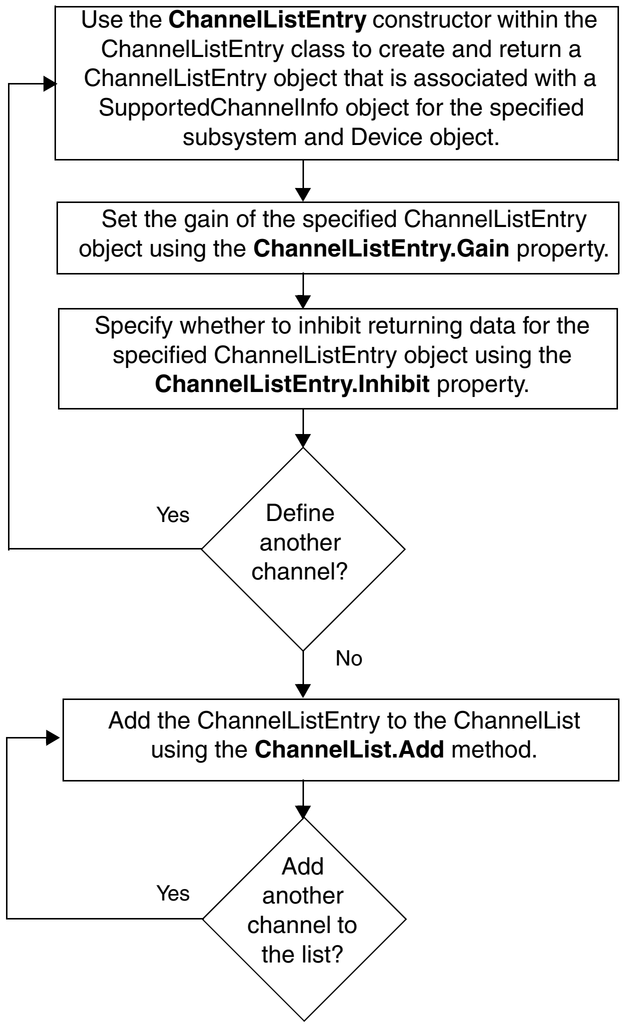
If supported by your device, you can add a non-native channel (such as the digital I/O port, counter/timer, or tachometer) to the channel list. However, you must configure these channels using the OpenLayers.Base namespace. See Chapter 5 starting on page 347 for more information.

The channel is appended to the end of the channel list. A ChannelListEntry object is returned.

The driver sets the actual gain as closely as possible to the number specified. You can read back the exact gain after configuring the subsystem using the **Gain** property.

If inhibited, the values for the specified channel object are acquired, and then discarded.

Add a Channel by ChannelListyEntry

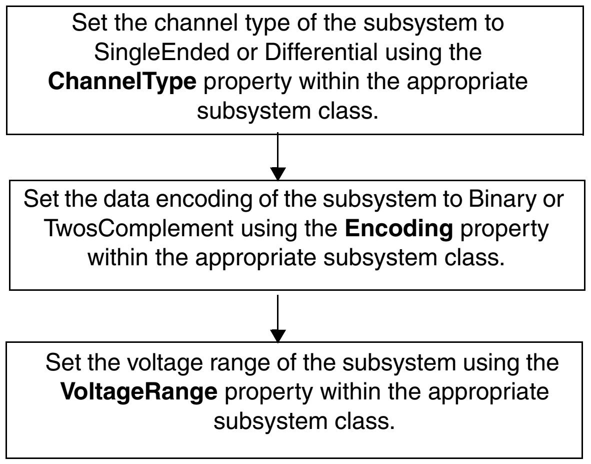

Use the **ChannelListEntry** constructor within the ChannelListEntry class to create and return a ChannelListEntry object that is associated with a SupportedChannelInfo object for the specified subsystem and Device object.
Set the gain of the specified ChannelListEntry object using the **ChannelListEntry.Gain** property.
Specify whether to inhibit returning data for the specified ChannelListEntry object using the **ChannelListEntry.Inhibit** property.
Define another channel?
Yes
No
Add the ChannelListEntry to the ChannelList using the **ChannelList.Add** method.
Add another channel to the list?
Yes

The driver sets the actual gain as closely as possible to the number specified. You can read back the exact gain after configuring the subsystem using the **Gain** property.

If inhibited, the values for the specified channel object are acquired, and then discarded.

The channel is appended to the end of the channel list.

Set up Common Subsystem Parameters



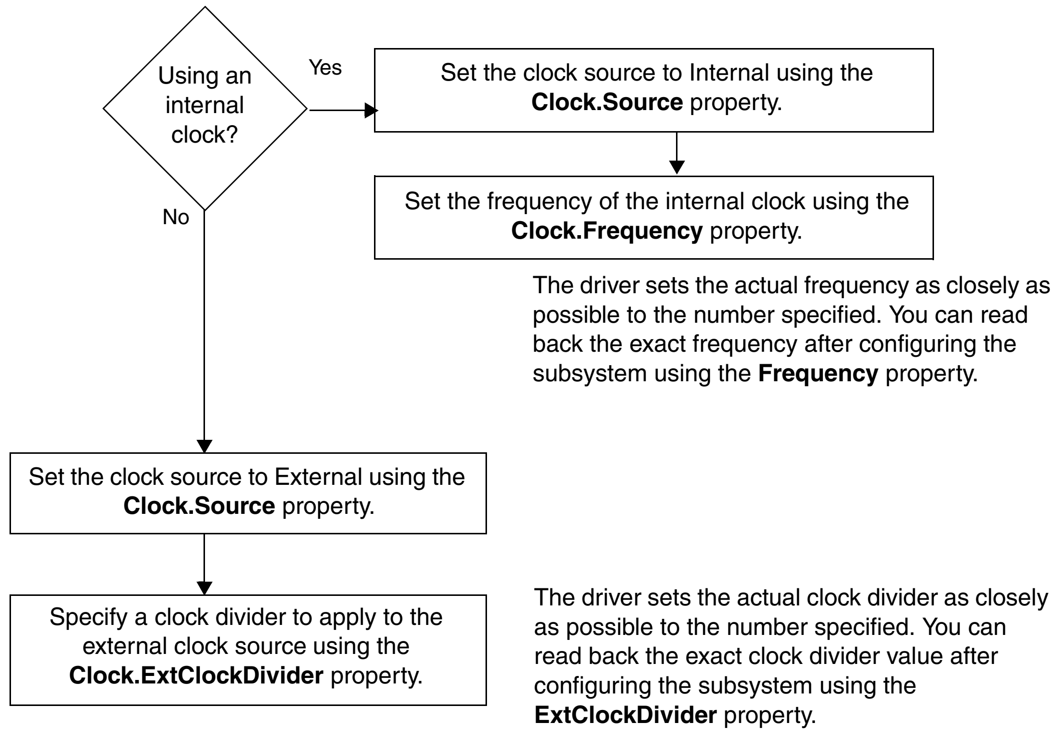
Set the channel type of the subsystem to SingleEnded or Differential using the **ChannelType** property within the appropriate subsystem class.

Set the data encoding of the subsystem to Binary or TwosComplement using the **Encoding** property within the appropriate subsystem class.

Set the voltage range of the subsystem using the **VoltageRange** property within the appropriate subsystem class.

Specify SingleEnded if you are using pseudo-differential channels.

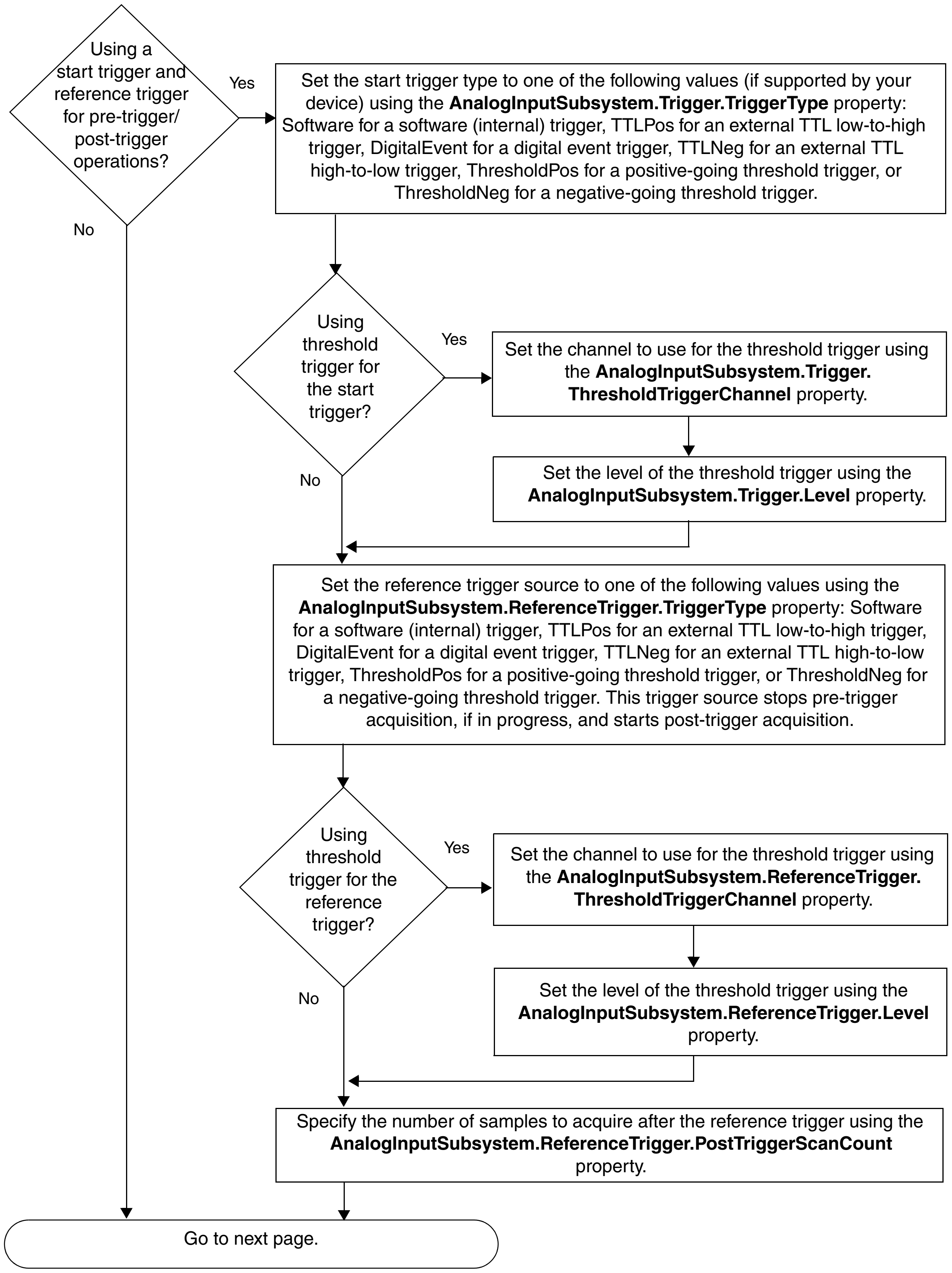
Set Up Clocks

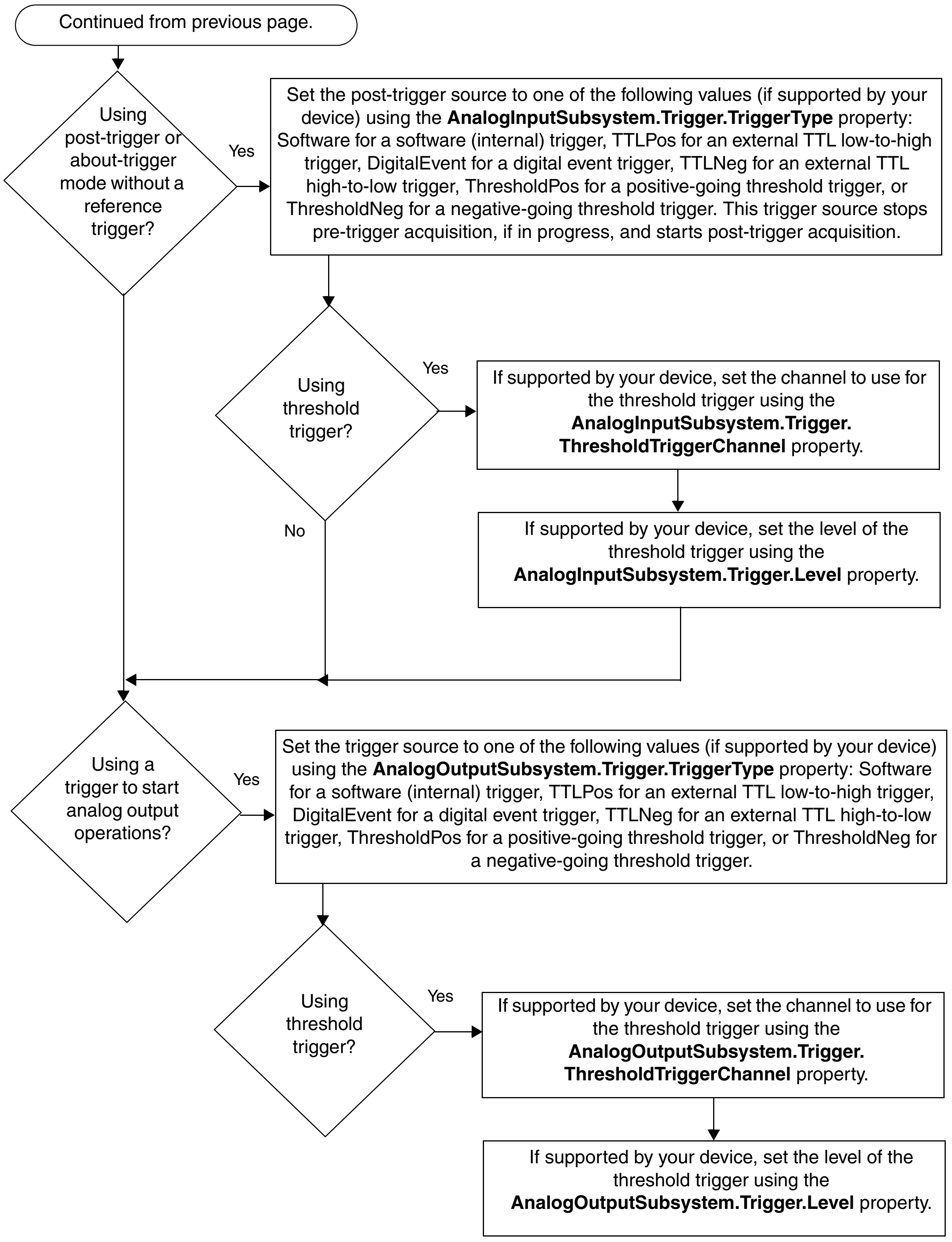

Using an internal clock?
Yes
No
Set the clock source to Internal using the **Clock.Source** property.
Set the frequency of the internal clock using the **Clock.Frequency** property.
Set the clock source to External using the **Clock.Source** property.
Specify a clock divider to apply to the external clock source using the **Clock.ExtClockDivider** property.

The driver sets the actual frequency as closely as possible to the number specified. You can read back the exact frequency after configuring the subsystem using the **Frequency** property.

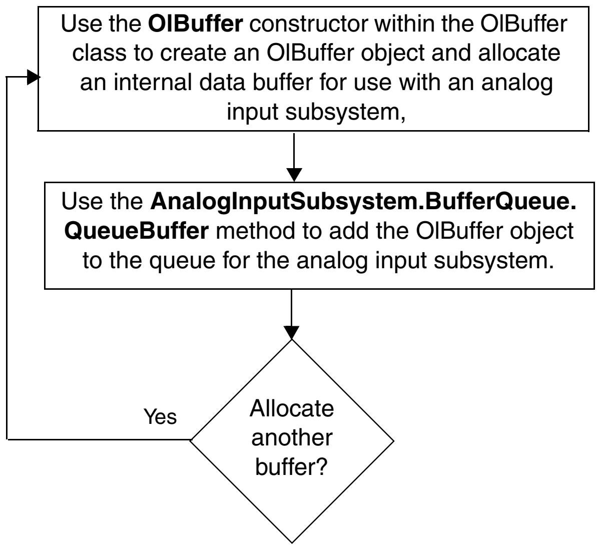
The driver sets the actual clock divider as closely as possible to the number specified. You can read back the exact clock divider value after configuring the subsystem using the **ExtClockDivider** property.

Set Up Triggers


Using a start trigger and reference trigger for pre-trigger/ post-trigger operations?
Yes
No
Set the start trigger type to one of the following values (if supported by your device) using the **AnalogInputSubsystem.Trigger.TriggerType** property: Software for a software (internal) trigger, TTLPos for an external TTL low-to-high trigger, DigitalEvent for a digital event trigger, TTLNeg for an external TTL high-to-low trigger, ThresholdPos for a positive-going threshold trigger, or ThresholdNeg for a negative-going threshold trigger.
Using threshold trigger for the start trigger?
Yes
No
Set the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger. ThresholdTriggerChannel** property.
Set the level of the threshold trigger using the **AnalogInputSubsystem.Trigger.Level** property.
Set the reference trigger source to one of the following values using the **AnalogInputSubsystem.ReferenceTrigger.TriggerType** property: Software for a software (internal) trigger, TTLPos for an external TTL low-to-high trigger, DigitalEvent for a digital event trigger, TTLNeg for an external TTL high-to-low trigger, ThresholdPos for a positive-going threshold trigger, or ThresholdNeg for a negative-going threshold trigger. This trigger source stops pre-trigger acquisition, if in progress, and starts post-trigger acquisition.
Using threshold trigger for the reference trigger?
Yes
No
Set the channel to use for the threshold trigger using the **AnalogInputSubsystem.ReferenceTrigger. ThresholdTriggerChannel** property.
Set the level of the threshold trigger using the **AnalogInputSubsystem.ReferenceTrigger.Level** property.
Specify the number of samples to acquire after the reference trigger using the **AnalogInputSubsystem.ReferenceTrigger.PostTriggerScanCount** property.
Go to next page.


Continued from previous page.
Using post-trigger or about-trigger mode without a reference trigger?
Yes
Set the post-trigger source to one of the following values (if supported by your device) using the **AnalogInputSubsystem.Trigger.TriggerType** property: Software for a software (internal) trigger, TTLPos for an external TTL low-to-high trigger, DigitalEvent for a digital event trigger, TTLNeg for an external TTL high-to-low trigger, ThresholdPos for a positive-going threshold trigger, or ThresholdNeg for a negative-going threshold trigger. This trigger source stops pre-trigger acquisition, if in progress, and starts post-trigger acquisition.
Using threshold trigger?
Yes
If supported by your device, set the channel to use for the threshold trigger using the **AnalogInputSubsystem.Trigger. ThresholdTriggerChannel** property.
No
If supported by your device, set the level of the threshold trigger using the **AnalogInputSubsystem.Trigger.Level** property.
Using a trigger to start analog output operations?
Yes
Set the trigger source to one of the following values (if supported by your device) using the **AnalogOutputSubsystem.Trigger.TriggerType** property: Software for a software (internal) trigger, TTLPos for an external TTL low-to-high trigger, DigitalEvent for a digital event trigger, TTLNeg for an external TTL high-to-low trigger, ThresholdPos for a positive-going threshold trigger, or ThresholdNeg for a negative-going threshold trigger.
Using threshold trigger?
Yes
If supported by your device, set the channel to use for the threshold trigger using the **AnalogOutputSubsystem.Trigger. ThresholdTriggerChannel** property.
If supported by your device, set the level of the threshold trigger using the **AnalogOutputSubsystem.Trigger.Level** property.

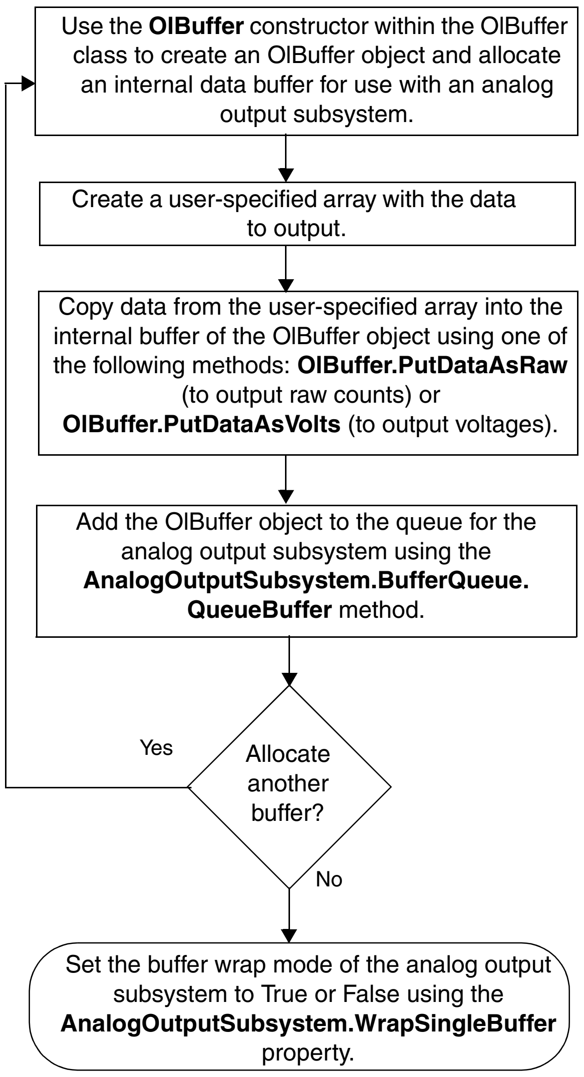
Set Up Input Buffering


Use the **OlBuffer** constructor within the OlBuffer class to create an OlBuffer object and allocate an internal data buffer for use with an analog input subsystem,
Use the **AnalogInputSubsystem.BufferQueue. QueueBuffer** method to add the OlBuffer object to the queue for the analog input subsystem.
Allocate another buffer?
Yes

During this step, you also determine the size of the internal data buffer by specifying the number of samples in the buffer (each sample typically requires 2 bytes).

Continuous input operations require a minimum of two OlBuffer objects.

Set Up Output Buffering


Use the **OlBuffer** constructor within the OlBuffer class to create an OlBuffer object and allocate an internal data buffer for use with an analog output subsystem.
Create a user-specified array with the data to output.
Copy data from the user-specified array into the internal buffer of the OlBuffer object using one of the following methods: **OlBuffer.PutDataAsRaw** (to output raw counts) or **OlBuffer.PutDataAsVolts** (to output voltages).
Add the OlBuffer object to the queue for the analog output subsystem using the **AnalogOutputSubsystem.BufferQueue. QueueBuffer** method.
Allocate another buffer?
Yes
No
Set the buffer wrap mode of the analog output subsystem to True or False using the **AnalogOutputSubsystem.WrapSingleBuffer** property.

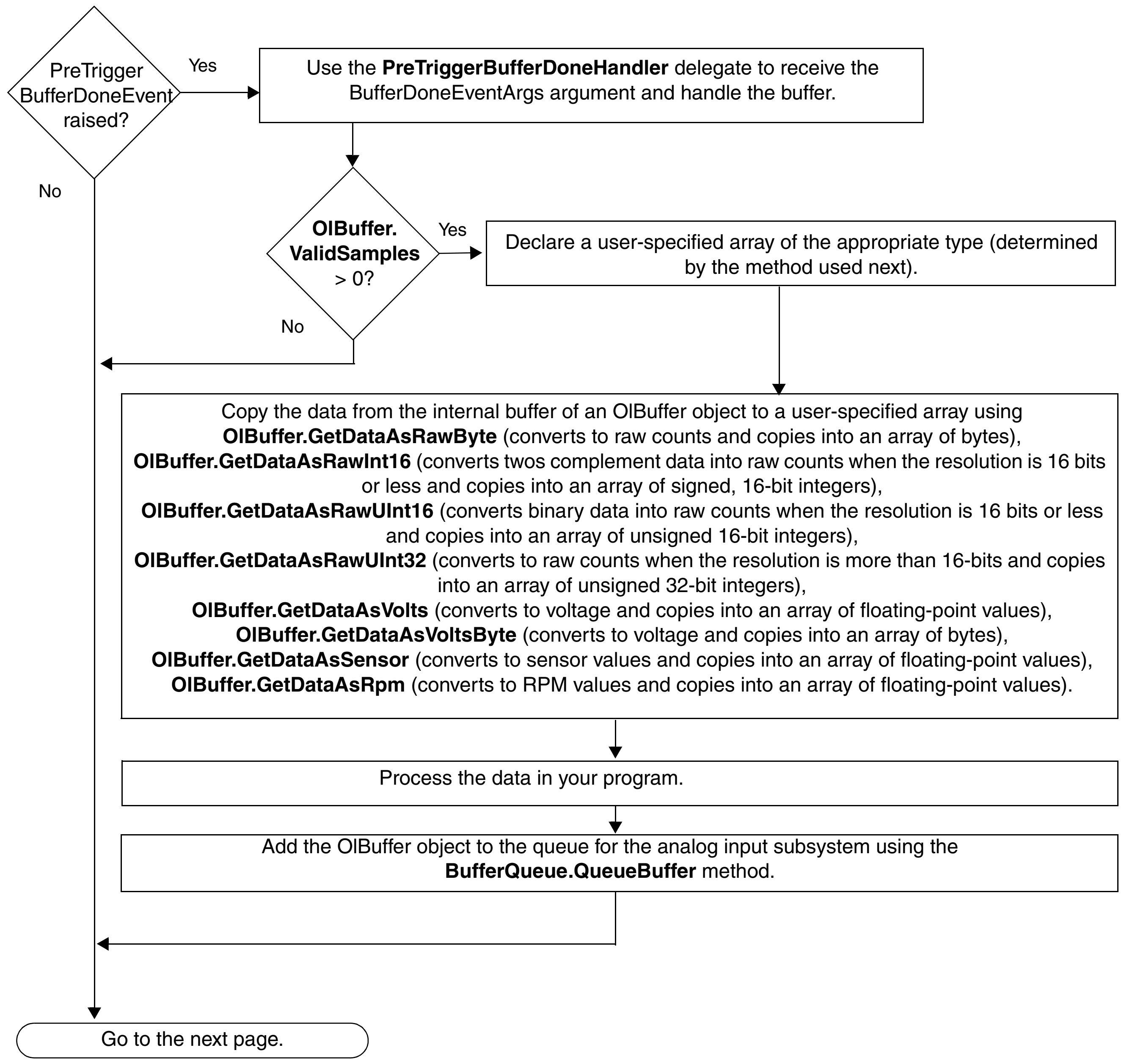
During this step, you also determine the size of the internal buffer by specifying the number of samples in the buffer (each sample typically requires 2 bytes).

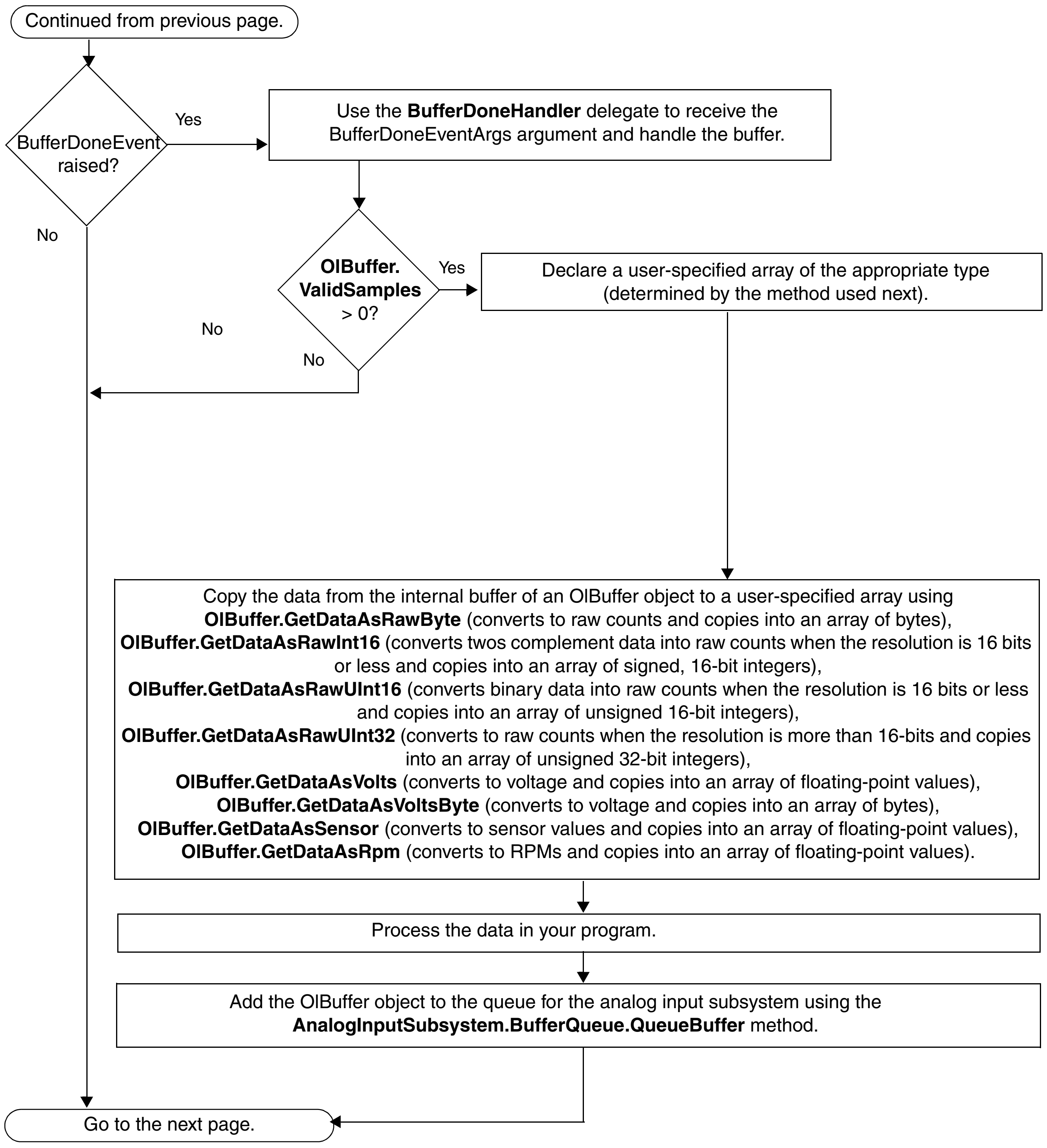
Continuous output operations require two OlBuffer objects if **WrapSingleBuffer** is False (one if **WrapSingleBuffer** is True).

By default, **WrapSingleBuffer** is False. In this state, data is written from the allocated buffers continuously. As each buffer is emptied, a BufferDone event occurs. If no more buffers are available and queued to the subsystem, the operation stops.

If you set **WrapSingleBuffer** to True, the device driver continuously reuses the first buffer queued to the analog output subsystem. Data from a single output buffer is downloaded to the FIFO of the device (if supported by the device) and is written out starting from the first location of the buffer; when the end of the buffer is reached, the device starts outputting data from the first location of the buffer, and the process repeats.

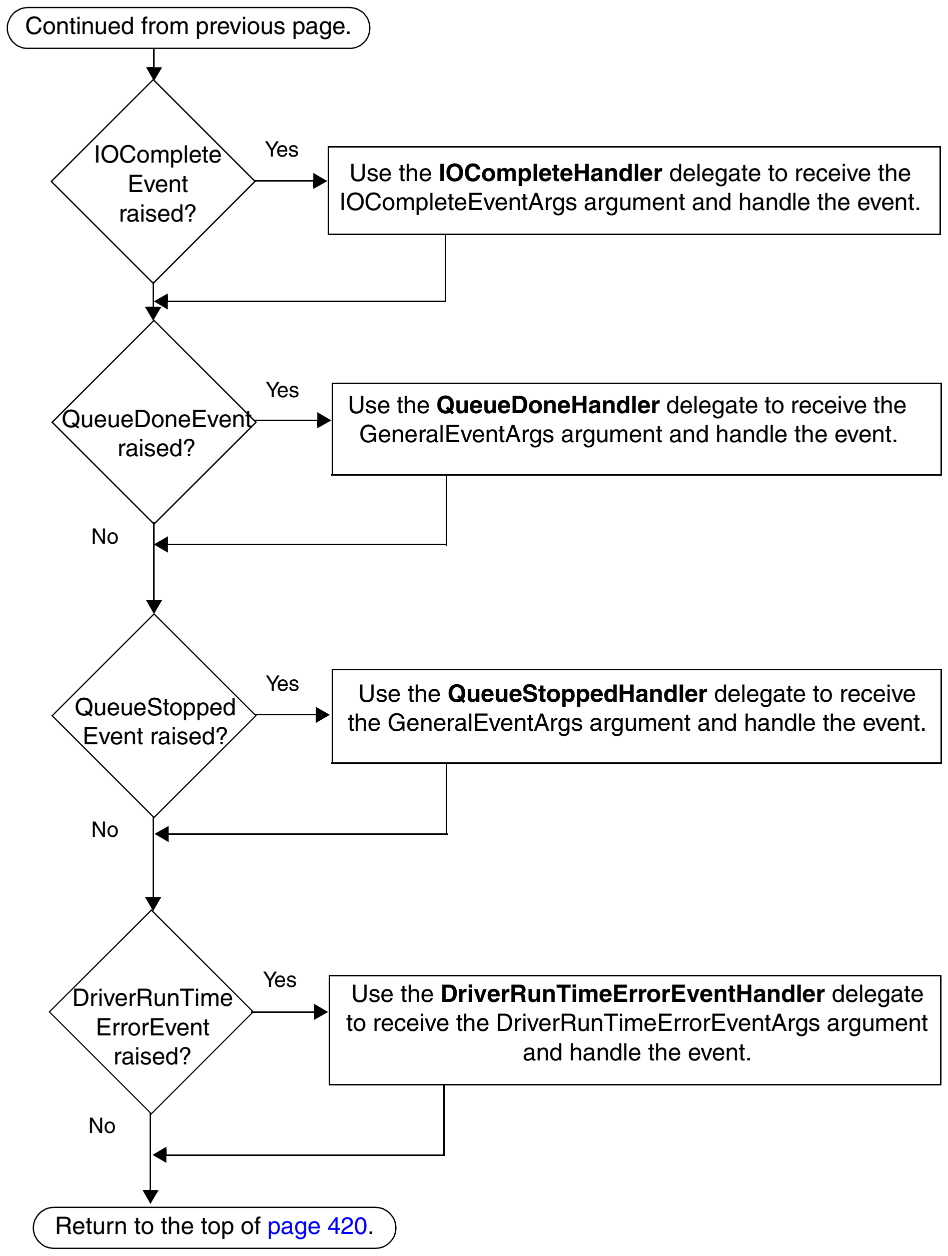
Deal with Events and Buffers for Input Operations


PreTrigger BufferDoneEvent raised?
Yes
No
Use the **PreTriggerBufferDoneHandler** delegate to receive the BufferDoneEventArgs argument and handle the buffer.
OlBuffer. ValidSamples > 0?
Yes
No
Declare a user-specified array of the appropriate type (determined by the method used next).
Copy the data from the internal buffer of an OlBuffer object to a user-specified array using **OlBuffer.GetDataAsRawByte** (converts to raw counts and copies into an array of bytes), **OlBuffer.GetDataAsRawInt16** (converts twos complement data into raw counts when the resolution is 16 bits or less and copies into an array of signed, 16-bit integers), **OlBuffer.GetDataAsRawUInt16** (converts binary data into raw counts when the resolution is 16 bits or less and copies into an array of unsigned 16-bit integers), **OlBuffer.GetDataAsRawUInt32** (converts to raw counts when the resolution is more than 16-bits and copies into an array of unsigned 32-bit integers), **OlBuffer.GetDataAsVolts** (converts to voltage and copies into an array of floating-point values), **OlBuffer.GetDataAsVoltsByte** (converts to voltage and copies into an array of bytes), **OlBuffer.GetDataAsSensor** (converts to sensor values and copies into an array of floating-point values), **OlBuffer.GetDataAsRpm** (converts to RPM values and copies into an array of floating-point values).
Process the data in your program.
Add the OlBuffer object to the queue for the analog input subsystem using the **BufferQueue.QueueBuffer** method.
Go to the next page.

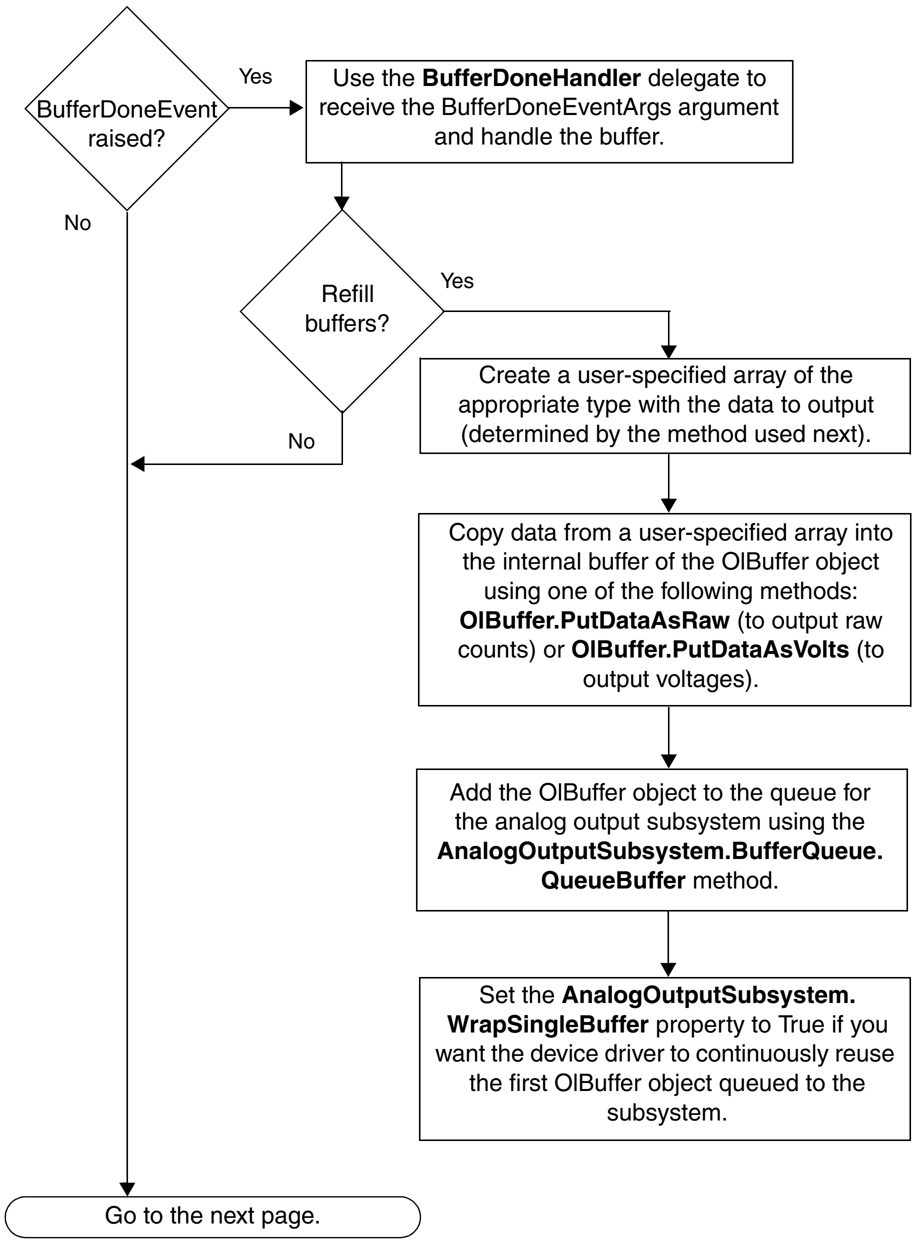

Continued from previous page.
BufferDoneEvent raised?
Yes
No
Use the **BufferDoneHandler** delegate to receive the BufferDoneEventArgs argument and handle the buffer.
OlBuffer. ValidSamples > 0?
Yes
No
No
Declare a user-specified array of the appropriate type (determined by the method used next).
Process the data in your program.
Go to the next page.

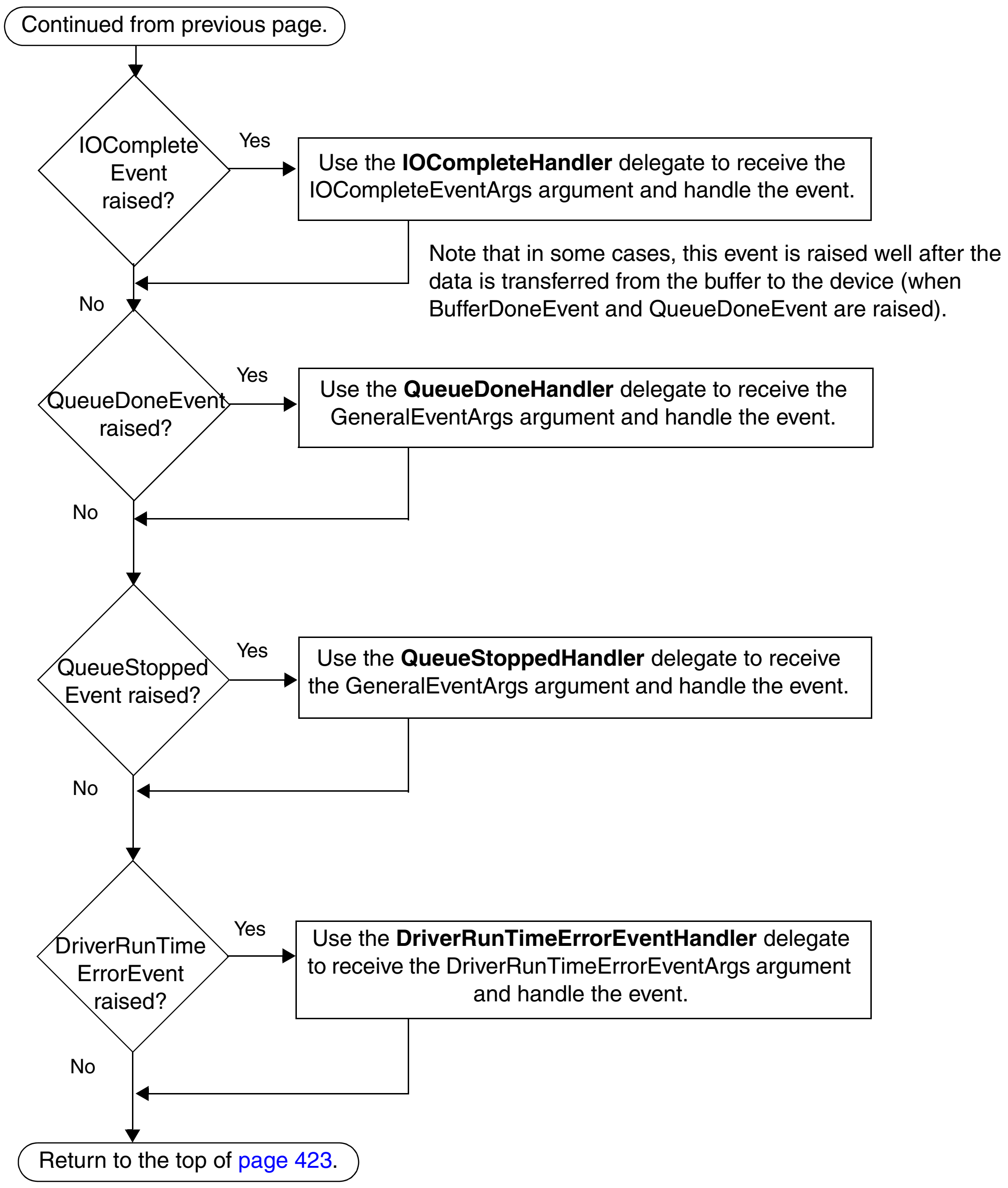
Copy the data from the internal buffer of an OlBuffer object to a user-specified array using **OlBuffer.GetDataAsRawByte** (converts to raw counts and copies into an array of bytes), **OlBuffer.GetDataAsRawInt16** (converts twos complement data into raw counts when the resolution is 16 bits or less and copies into an array of signed, 16-bit integers), **OlBuffer.GetDataAsRawUInt16** (converts binary data into raw counts when the resolution is 16 bits or less and copies into an array of unsigned 16-bit integers), **OlBuffer.GetDataAsRawUInt32** (converts to raw counts when the resolution is more than 16-bits and copies into an array of unsigned 32-bit integers), **OlBuffer.GetDataAsVolts** (converts to voltage and copies into an array of floating-point values), **OlBuffer.GetDataAsVoltsByte** (converts to voltage and copies into an array of bytes), **OlBuffer.GetDataAsSensor** (converts to sensor values and copies into an array of floating-point values), **OlBuffer.GetDataAsRpm** (converts to RPMs and copies into an array of floating-point values).

Add the OlBuffer object to the queue for the analog input subsystem using the **AnalogInputSubsystem.BufferQueue.QueueBuffer** method.

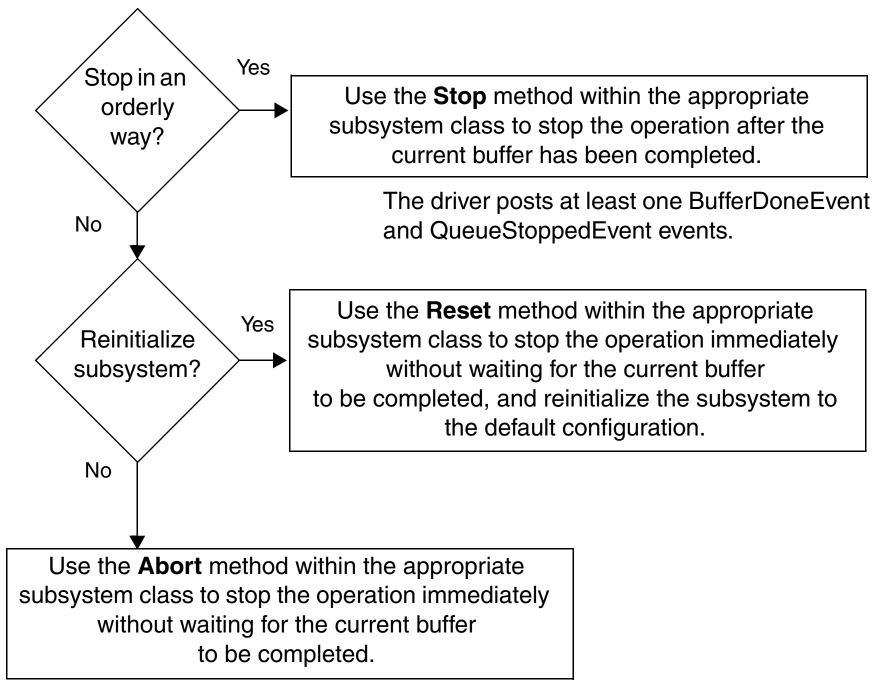

Continued from previous page.
IOComplete Event raised?
Yes
Use the **IOCompleteHandler** delegate to receive the IOCompleteEventArgs argument and handle the event.
QueueDoneEvent raised?
Yes
Use the **QueueDoneHandler** delegate to receive the GeneralEventArgs argument and handle the event.
No
QueueStopped Event raised?
Yes
Use the **QueueStoppedHandler** delegate to receive the GeneralEventArgs argument and handle the event.
No
DriverRunTime ErrorEvent raised?
Yes
Use the **DriverRunTimeErrorEventHandler** delegate to receive the DriverRunTimeErrorEventArgs argument and handle the event.
No
Return to the top of page 420.

Deal with Events and Buffers for Output Operations

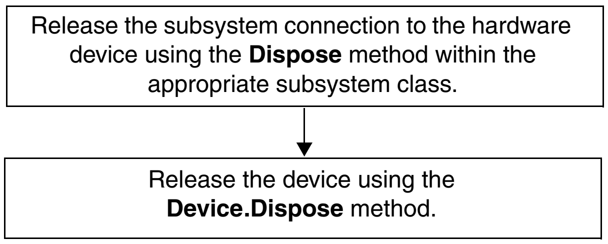

BufferDoneEvent raised?
Yes
No
Use the **BufferDoneHandler** delegate to receive the BufferDoneEventArgs argument and handle the buffer.
Refill buffers?
Yes
No
Create a user-specified array of the appropriate type with the data to output (determined by the method used next).
Copy data from a user-specified array into the internal buffer of the OlBuffer object using one of the following methods: **OlBuffer.PutDataAsRaw** (to output raw counts) or **OlBuffer.PutDataAsVolts** (to output voltages).
Add the OlBuffer object to the queue for the analog output subsystem using the **AnalogOutputSubsystem.BufferQueue. QueueBuffer** method.
Set the **AnalogOutputSubsystem. WrapSingleBuffer** property to True if you want the device driver to continuously reuse the first OlBuffer object queued to the subsystem.
Go to the next page.


Continued from previous page.
IOComplete Event raised?
Yes
Use the **IOCompleteHandler** delegate to receive the IOCompleteEventArgs argument and handle the event.
Note that in some cases, this event is raised well after the data is transferred from the buffer to the device (when BufferDoneEvent and QueueDoneEvent are raised).
No
QueueDoneEvent raised?
Yes
Use the **QueueDoneHandler** delegate to receive the GeneralEventArgs argument and handle the event.
No
QueueStopped Event raised?
Yes
Use the **QueueStoppedHandler** delegate to receive the GeneralEventArgs argument and handle the event.
No
DriverRunTime ErrorEvent raised?
Yes
Use the **DriverRunTimeErrorEventHandler** delegate to receive the DriverRunTimeErrorEventArgs argument and handle the event.
No
Return to the top of page 423.

Stop the Operation


Stop in an orderly way?
Yes
Use the **Stop** method within the appropriate subsystem class to stop the operation after the current buffer has been completed.
The driver posts at least one BufferDoneEvent and QueueStoppedEvent events.
No
Reinitialize subsystem?
Yes
Use the **Reset** method within the appropriate subsystem class to stop the operation immediately without waiting for the current buffer to be completed, and reinitialize the subsystem to the default configuration.
No
Use the **Abort** method within the appropriate subsystem class to stop the operation immediately without waiting for the current buffer to be completed.

Clean Up Single-Value I/O Operations

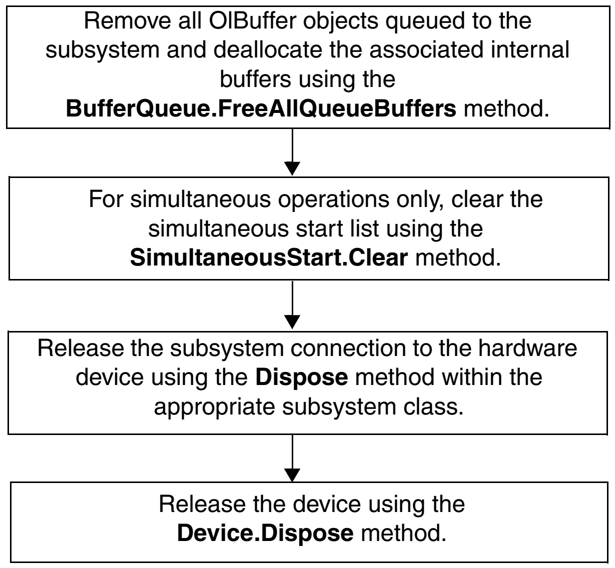


Release the subsystem connection to the hardware device using the **Dispose** method within the appropriate subsystem class.

↓

Release the device using the **Device.Dispose** method.

Clean Up Buffered I/O Operations



Remove all OlBuffer objects queued to the subsystem and deallocate the associated internal buffers using the **BufferQueue.FreeAllQueueBuffers** method.

↓

For simultaneous operations only, clear the simultaneous start list using the **SimultaneousStart.Clear** method.

↓

Release the subsystem connection to the hardware device using the **Dispose** method within the appropriate subsystem class.

↓

Release the device using the **Device.Dispose** method.

7

Product Support

Should you experience problems using the DT-Open Layers for .NET Class Library, follow these steps:

1. Read all the appropriate sections of this manual, including any "Read This First" information.
2. Check for a README file on the Data Acquisition OMNI 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. For information, refer to page 21.

If you are still having problems, Data Translation's Technical Support Department is available to provide technical assistance. To request technical support, go to our web site at www.mccdaq.com and click on the Support link.

When requesting technical support, be prepared to provide the following information:

- Your product serial number
- The hardware/software product you need help on
- The version of the CD you are using
- Your contract number, if applicable

If you are located outside the USA, contact your local distributor; see our web site (www.mccdaq.com) for the name and telephone number of your nearest distributor.

A

Error Codes and Messages

Table 78 lists the errors that can be returned by the DT-Open Layers for .NET Class Library.

Table 78: Error Codes and Messages Returned by the DT-Open Layers for .NET Class Library

Error Code	Message Description
NoError	No error occurred.
Success	The method completed successfully.
InvalidElement	Invalid subsystem element specified.
InvalidListSize	An attempt was made to set the ChannelList to an invalid size.
InvalidListEntry	An invalid ChannelListEntry object was specified.
InvalidChannel	An invalid channel was specified.
InvalidChannelType	An invalid ChannelType was specified.
InvalidTrigger	An invalid TriggerType was specified.
InvalidResolution	An invalid Resolution was specified.
InvalidClockSource	An invalid ClockSource was specified.
InvalidFrequency	An invalid Clock.Frequency was specified.
InvalidPulseType	An invalid PulseType was specified.
InvalidPulseWidth	An invalid PulseWidth was specified.
InvalidCounterMode	An invalid CounterMode was specified.
InvalidDataFlow	An invalid DataFlow was specified.
SubsystemInUse	An attempt was made to access a subsystem that is already in use.
SubsystemNotInUse	An operation was attempted on a subsystem that is not in use.
AlreadyRunning	An operation was attempted on a running subsystem.
NotConfigured	An operation was attempted on a subsystem that was not configured (Config).
DataFlowMismatch	An invalid DataFlow mode was set for the current operation.
NotRunning	The specified subsystem is not running.
InvalidRange	An invalid voltage range was specified.
NotSupported	The operation that you are attempting to perform is not supported.
InvalidDivider	An invalid ExtClockDivider was specified.
InvalidGate	An invalid GateType was set for the current operation.
InvalidChannelList	An invalid ChannelList was specified.
ADOverrun	An A/D overrun error occurred. To deal with this error, increase the size of the buffers, slow down the sampling rate, or stop other CPU-intensive running programs.
NoQueuedBuffers	No OlBuffers are queued to the subsystem (see QueueBuffer).
CannotOpenDriver	The device driver cannot be initialized.

Table 78: Error Codes and Messages Returned by the DT-Open Layers for .NET Class Library

Error Code	Message Description
CantCascade	The specified subsystem cannot be cascaded.
WrongCounterMode	An invalid CounterMode was set for the current operation.
InvalidGain	An invalid Gain value was set for the ChannelListEntry.
InvalidRetriggerFrequency	An invalid RetriggerFrequency was requested for the current ChannelList size.
CommandTimeout	A command has timed out in the device driver.
EventCountOverflow	The counter overflowed during an event counting operation.
NoSubsystemsOnSimultaneousStartList	No subsystems have been added to the SImultaneousStart list.
NoChannelInhibitList	The subsystem does not support ChannelListEntry.Inhibit.
NotPrestarted	The subsystem has not been prestarted (see SimultaneousStart).
InvalidInhibitState	All ChannelListEntry objects are inhibited.
RequiredSubsystemInUse	The additional required subsystem is in use.
WrapModeMismatch	WrapSingleBuffer cannot be true for the requested operation.
BadRetriggerSource	An invalid RetriggerSource was specified.
BadMultiScanCount	The MultiScanCount value exceeds the maximum number of scans of the ChannelList.
InvalidRetrigger	Triggered scan operations are not supported in combination with DataFlow.ContinuousPrePostTrigger or DataFlow.ContinuousPreTrigger.
InvalidPreTrigger	The Trigger.PreTriggerSource must be Software when used with DataFlow.ContinuousPrePostTrigger or DataFlow.ContinuousPreTrigger.
GeneralFailure	A no-specific failure has occurred in the device driver.
BadEdge	An invalid StartEdge or StopEdge was specified for the current counter/timer mode.
HalfCounterEntry	Only half of a 32-bit counter was added to the ChannelList. Both 16-bit words must be added.
InvalidX4IndexCombination	An invalid combination of Index and X4Scaling was specified.
InvalidCouplingType	An invalid CouplingType was specified for the device.
InvalidCurrentSource	An invalid ExcitationCurrentSource was specified for the device.
InvalidValue	An invalid ExcitationCurrentValue was specified for the internal excitation current source.
InvalidWhenADRunning	Operation is prohibited while the A/D is running. Refer to your device documentation for details.
InvalidSynchronizationMode	Invalid synchronization mode.
InvalidWhenDARunning	Operation is prohibited while the D/A is running. Refer to your device documentation for details.
CannotAllocateBuffer	Cannot allocate the requested data buffer.

Table 78: Error Codes and Messages Returned by the DT-Open Layers for .NET Class Library

Error Code	Message Description
BufferAlreadyQueued	The OlBuffer object has already been queued to a subsystem.
BufferInProcess	The OlBuffer object has already been queued to the device driver.
InvalidBufferSize	An invalid size was specified for an OlBuffer object.
OddSizeBuffer	The number samples in the OlBuffer object must be a multiple of 2 for the current operation.
BufferNotAllocated	The internal data buffer, which is encapsulated by the OlBuffer object, has been deallocated.
DataWidthMismatch	An OlBuffer was called using a data type that is not compatible with the subsystem's Resolution.
NoValidSamples	The OlBuffer object has 0 ValidSamples.
ChannelNotInChannelList	The specified channel is not in the ChannelList.
SourceBufferTooSmall	The specified array is too small for the requested operation.
DuplicateChannelName	Each SupportedChannelInfo object in SupportedChannels must have a unique name.
BufferInUse	The OlBuffer object is in use.
SubsystemIncompatible	The subsystem is incompatible with the OlBuffer format.
EmptyChannelList	The ChannelList must have at least one entry for continuous operations.
InvalidChannelListIndex	An invalid ChannelList index was specified.
SubsystemStopping	The subsystem is in the process of stopping or aborting.
FifoOverflow	The driver could not read data from the device FIFO (or Windows USB FIFO) fast enough. To deal with this error, increase the size of the buffers, slow down the sampling rate, or stop other CPU-intensive running programs.
FifoUnderflow	The driver could not write data to the device FIFO (or Windows USB FIFO) fast enough. To deal with this error, increase the size of buffers, slow down the sampling rate, or stop other CPU-intensive running programs.
DeviceOverClocked	The A/D clock (usually the external A/D clock) is running too fast on the device.
TriggerError	This error is generated by the device driver when a trigger is detected but not acted on by the hardware.
DeviceError	This error is generated by the device driver if a USB bus or hardware problem occurs.
InvalidError	An unknown error string was passed to **GetErrorCode**.
NoThermocoupleSupport	The subsystem does not support thermocouples.
NoCjcChannel	No CJC channel was specified in the ChannelList.
NoThermocoupleTypeSpecified	A ThermocoupleType was not specified for the requested channel.
ThermocoupleTypeSpecified	A ThermocoupleType was specified for the requested channel.

Table 78: Error Codes and Messages Returned by the DT-Open Layers for .NET Class Library

Error Code	Message Description
SetSingleValuesNotSupported	The **SetSingleValuesAsRaw** and **SetSingleValuesAsVolts** methods are not supported by this subsystem.
DuplicateChannelSpecified	A duplicate channel was specified for this operation.
AutoCalibrateNotSupported	Auto-calibrate is not supported by this device.
NoRTDSupport	The subsystem does not support RTDs.
NoRTDTypeSpecified	An RTDType was not specified for the requested channel.
RTDTypeSpecified	An RTDType was specified for the requested channel.
ReturnsOhmsNotSupported	The subsystem does not support returning values as Ohms.
AccessDenied	Access was denied to the subsystem or device.
TedsError	TEDs input streaming error.
NoThermistorSupport	The subsystem does not support Thermistor measurement.
CollectionNameMismatch	Device collection name mismatch. There is more than one collection present with the same name.
CollectionDeviceCountMismatch	Device collection device count mismatch. Not all devices in the collection were found.
ChannelNotOnMaster	The specified channel is invalid because it is not on the master device.
BufferSizeNotMultiple	The number samples in all queued OlBuffer objects must be a multiple of the number of channels in the ChannelList for a DeviceCollection.
CannotQueueBuffer	The DeviceCollection cannot queue a buffer since not all devices in the collection have the same state.
LastError	Last error code in the enumeration.

Index