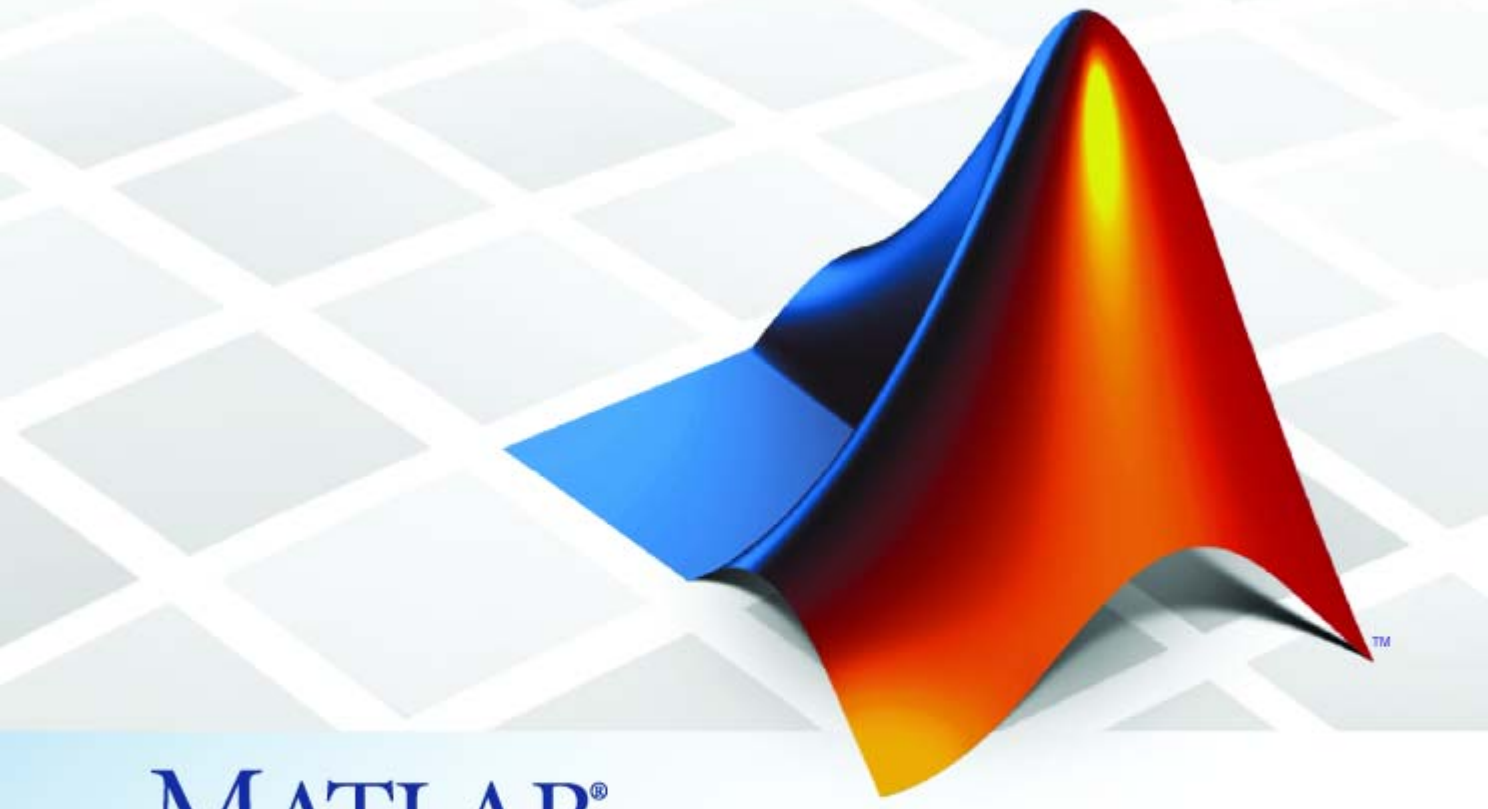
Data Acquisition Toolbox™ Adaptor Kit 2

User's Guide



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Data Aquisition Toolbox™ Adaptor Kit User's Guide

Revision History

November 2000	Online only	New for Version 1 (Release 12)
July 2002	Online only	Revised for Version 2 (Release 13)
June 2004	Online only	Minor revision for Version 2.5 (Release 14)
September 2005	Online only	Minor revision for Version 2.7 (Release 14SP3)
March 2006	Online only	Minor revision for Version 2.9 (Release 2006a)
March 2008	Online only	Minor revision for Version 2.12 (Release 2008a)

Contents

1 Introduction

2 Tutorial

3 Step-by-Step Instructions for Adaptor Creation

4 Working with Properties

5 Buffering Techniques

6 Callbacks and Threading

A Adaptor Kit Interface Reference

B Engine Interface Reference

C Engine Structures

D Sample Property and daqhwinfo Tables

1

Introduction

Overview

Who Should Read This Document?

You should read this document if you want to

- Develop an adaptor to support hardware that is not currently supported by the Data Acquisition Toolbox
- Add new features to an existing adaptor

The Data Acquisition Toolbox Adaptor Kit addresses the needs of individuals who want to interface the toolbox to a single board, and manufacturers wanting to interface the toolbox to a range of hardware. Although this document is aimed primarily at supporting a single board, hardware manufacturers should use this document as the basis for developing a multiple-board adaptor, generalizing the single-board support issues appropriately.

What Knowledge Is Required?

To build an adaptor, you should have a working knowledge of

- C++, Microsoft's Component Object Model (COM), and the Active Template Library (ATL)
- The functionality of your hardware device, and its associated Application Programming Interface (API)
- Data Acquisition Toolbox concepts, functionality, and terminology as described in the *Data Acquisition Toolbox User's Guide*

What Effort Is Required?

The effort required to produce an adaptor depends on the capabilities of the hardware device and your acquisition requirements.

The simplest type of adaptor supports only single-sample acquisition or burst acquisition, and uses software clocking. You can create this type of adaptor by modifying the demo adaptor.

Note Some hardware does not support single-sample acquisition and, as a result, it does not support software clocking. In this case, you cannot build this simple type of adaptor.

The next level of complexity is an adaptor that implements hardware clocking and buffering, but works only for a limited number of similar hardware devices. In this case, you can decrease development time by hard-coding some configuration information or by limiting the hardware features that you use. For example, you might decide to ignore some triggering functionality.

The greatest level of complexity is an adaptor that provides complete support to a line of data acquisition devices. To develop an adaptor of this type typically requires a minimum of four months.

Tools

The example code for the Adaptor Kit was created using Microsoft Visual C++ Version 6, Service Pack 4.

Writing an Adaptor Versus Writing a MEX File

To communicate with your hardware, you can develop either an adaptor DLL, which extends the existing Data Acquisition Toolbox, or you can create a MEX file.

A MEX file is a shared library (DLL in Windows), which you call from MATLAB® as if it is an internal MATLAB command or an M-file. It can contain multiple functions, which are called from MATLAB as parameters added to the MEX file name. MEX files can be implemented on any platform supported by MATLAB.

You might want to create a MEX file if the supported data acquisition functionality is simple, for example, single-sample or burst mode acquisition. You must create a MEX file in these circumstances:

- You want to use a platform not supported by the Data Acquisition Toolbox.
- You want to support features not included in the Data Acquisition Toolbox.

For advanced data acquisition tasks, you should develop an adaptor. This approach gives you an advantage of having multiple prepackaged features, such as high-speed storage to disk, multiple triggering modes, including analog and pretriggering, and a standardized interface to the data acquisition device, including units conversion.

The table below summarizes the capabilities of adaptor DLLs and MEX files.

Table 1-1: Adaptor DLLs Versus MEX Files

Feature	Adaptor DLL	MEX File
Supports all MATLAB platforms	No	Yes
Counter/timer	No	Can be implemented
Software triggering implementation	Implemented automatically	Very difficult to implement
Software clocking implementation	Implemented automatically	Very difficult to implement
Logging to disk	Implemented automatically	Very difficult to implement

Table 1-1: Adaptor DLLs Versus MEX Files (Continued)

Feature	Adaptor DLL	MEX File
Integrated into MATLAB with MATLAB objects	Yes	No
Callbacks	Provided in the toolbox	Difficult to implement
Background (asynchronous) and continuous acquisition	Provided in the toolbox	Difficult to implement

What Is the Adaptor Kit?

The Data Acquisition Toolbox Adaptor Kit consists of three major parts:

- This document
- The demo adaptor source code, which is located in the `matlabroot\toolbox\daq\daqadaptor` directory. This directory contains two subdirectories: `AdaptorKit` and `Demo`.

  `AdaptorKit` contains files that are common to all adaptors. Normally you would place these files in the `include` subfolder. `Demo` contains files that are specific to a particular adaptor — in this case the demo adaptor. The list of files in both directories is given in the following table.

Table 1-2: Demo Adaptor Source Code

Subfolder	File	Description
`AdaptorKit`	`AdaptorKit.h`	Contains definitions for non-device-specific classes and templates that are used for creating all adaptors. The defined classes provide support for software clocking, buffering, and triggering.
	`AdaptorKit.cpp`	Defines functions for the classes contained in `AdaptorKit.h`. Contains GUIDs for the engine. Defines high- and low-resolution timers using Windows Multimedia methods.
	`daqmex.idl`	Interface definition file used to define the COM interfaces of the data acquisition engine (`daqmex`).
	`daqmex.h`	Built from `daqmex.idl` by the Microsoft IDL compiler MIDL.
	`DaqmexStructs.h`	Defines most of the structures used by adaptor DLLs and the data acquisition engine.
	`SArrayAccess.h`	Defines classes and templates used for creating and managing safe arrays and vectors.

Table 1-2: Demo Adaptor Source Code (Continued)

Subfolder	File	Description
Demo	demo.dsp	Project file for building the demo adaptor.
	demo.def	Definition file for building demo.dll.
	demo.cpp	Defines the entry point into demo.dll.
	demo.rc	Resource script file generated by the Microsoft Developer Studio.
	demo.idl	Interface definition file for the demo adaptor. All demo adaptor-specific interfaces are defined here.
	resource.h	File is generated by the Microsoft Developer Studio. Contains definitions for constants used by the demo adaptor program.
	demoin.h	Defines the class Cdemoin, which implements the analog input interface ImwInput. This interface provides for software clocking.
	demoin.cpp	Defines functions for the Cdemoin class, which is defined in demoin.h.
	demoadapt.h	Defines the class Cdemoadapt, which implements the interface ImwDemoadapt. This interface declares methods that are common to the entire adaptor.
	demoadapt.cpp	Defines functions for the Cdemoadapt class, which is defined in demoadapt.h.
	StdAfx.h	Defines some directions for the compiler, and internally includes standard system header files.
	StdAfx.cpp	Internally includes standard system headers. Both StdAfx.cpp and StdAfx.h provide better organization of the header sections of the files in the project.

- The full source code for the adaptor DLLs included with the Data Acquisition Toolbox. All source code files are located in the folder

MATLABROOT/toolbox/daq/daq/src, which contains the subfolders listed below.

Table 1-3: Data Acquisition Toolbox Adaptor Source Files

Folder Name	Description
computerboards	Contains full source code for building the adaptor DLL for ComputerBoards (Measurement Computing Corp.) devices. The adaptor name is cbi and the adaptor DLL name is mwcbi.dll.
hpe1432	Contains full source code for building the adaptor DLL for the Agilent Technologies E1432/33/34 devices. The adaptor name is hpe1432 and the adaptor DLL name is mwhpe1432.dll.
mwnidaq	Contains full source code for building the adaptor DLL for National Instruments devices supported by the NI-DAQ driver. The adaptor name is nidaq and the adaptor DLL name is mwnidaq.dll.
winsound	Contains full source code for building the adaptor DLL for the generic sound card, which uses the Windows Waveform Audio driver. The adaptor name is winsound and the adaptor DLL name is mwwinsound.dll.
keithley	Contains full source code for building the adaptor DLL for Keithley Instruments devices. The adaptor name is keithley and the adaptor DLL name is mwkeithley.dll.
include	Contains common files for building all adaptor DLLs. This folder is practically identical to the folder AdaptorKit, included in the demo adaptor source. However, it includes these three additional files: • daqtbxver.h — Version control file • thread.h — Contains definitions of the thread class and classes necessary to spawn and maintain safe threads (such as mutex, semaphore) • cirbuf.h — Defines a class that implements a circular buffer

Toolbox Architecture

The Data Acquisition Toolbox consists of these components:

- M-files

  M-files contain MATLAB commands that allow you to connect to and communicate with your hardware. For example, you use the `analoginput` M-file to create a MATLAB object associated with your analog input subsystem. The M-files are located in the `MATLABROOT/toolbox/daq/daq` folder.

- The data acquisition engine

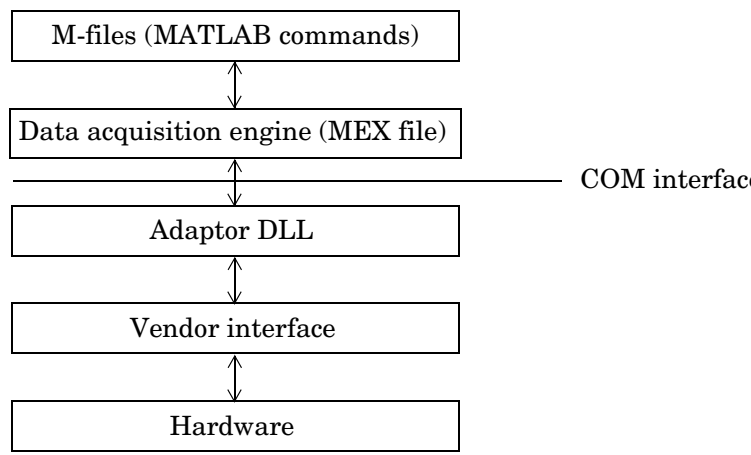
  The data acquisition engine contains functions that handle data acquisition objects and manage their properties. The engine also provides support for buffering and for managing acquired and output data.

- Adaptors

  An adaptor is a DLL that interacts directly with the vendor-supplied hardware device driver. The adaptor communicates with the device driver via the vendor's API. Normally the API functions are contained in a DLL that supplements the device driver.

The flow of information between toolbox components is shown below. The COM interface exists between the data acquisition engine and the adaptor DLL.

Figure 1-1: Flow of Information Between Toolbox Components



The relationship between the data acquisition engine and an adaptor DLL is implemented as a Component Object Model (COM) interface. The communication is always initiated by the engine when the data acquisition object is first created.

Thus, you can apply a client-server architecture model to this interface with the engine as a client and the adaptor as a server. However, when the data acquisition object is initialized, the engine sends a pointer to its main interface to the adaptor. This allows the adaptor to probe for all engine COM interfaces and methods via the `QueryInterface` function. The adaptor itself obtains the pointer to the engine class, based on the main interface. This enables it to call the necessary methods from the engine and use them in the acquisition process. This approach allows for version maintenance on both the engine and the adaptor sides. Additionally, it enables you to create adaptors as EXE files rather than DLL files, and provides for remote communication between the engine and adaptors.

The COM interface between the engine and the adaptor is described in detail in this document. To facilitate your understanding of these interfaces, the adaptor source code is provided as part of the Adaptor Kit.

Since these interfaces are based on COM, the data types you use while writing adaptors must conform to COM standards. Many of the data types found in C are supported, such as *long* and *double*. Other data types, such as BSTR and VARIANT, are also commonly used in COM-based applications. These data types are documented in many texts and in Microsoft's online documentation. Wrapper classes such as variant_t and bstr_t, and the ATL counterparts CComVariant and CComBSTR make using these data types much easier. These classes are documented by Microsoft as well.

Using This Manual

The Adaptor Kit User's Guide provides instructions and information required to implement an adaptor in C++. As such, it is not a conventional MATLAB Toolbox User's Guide, and you should not expect to find a layout similar to a MATLAB Toolbox User's Guide.

The layout of this document is intended to provide sufficient information for

- First-time adaptor implementors, who need to read all chapters in the guide carefully, and might need to refer to the Appendices for additional information on engine and adaptor kit interfaces and data structures.
- Experienced adaptor implementors, who need a checklist of things to do when implementing an adaptor. These implementors would use the Adaptor Kit as a reference guide rather than as a recipe of implementation steps.

In either case, you need to understand how the Adaptor Kit User's Guide is laid out, in order to make most effective use of the information in this Guide.

Chapter 1, "Introduction," provides an overview of the Adaptor Kit, the Toolbox architecture, and the Adaptor Kit files. You should read this chapter to gain an insight into how the Adaptor fits into the Data Acquisition Toolbox architecture.

Chapter 2 provides a tutorial that explains the relationship between a MATLAB user's interaction with the Data Acquisition Toolbox and the adaptor. First-time users should read this document in order to understand how and when the adaptor is called.

The main reference for all adaptor implementors should be Chapter 3, "Step-by-Step Instructions for Adaptor Creation." Both experienced and novice adaptor implementors should use the step-by-step guide when implementing new adaptors or modifying existing adaptors. The chapter is written to allow for easy implementation guidelines, and does not contain all the information required to implement a successful adaptor. Where relevant, information on implementation details has been left for a later chapter, and referenced in Chapter 3.

Chapter 4, "Working with Properties," explains how to implement code that allows you to query and modify adaptor properties. This chapter should be used as an implementation reference for the steps listed in Chapter 3.

Chapter 5, “Buffering Techniques,” explains how the engine manages buffering of data for continuous acquisition tasks. You should only need the information in this chapter if you plan on implementing hardware-clocked acquisition in your adaptor.

Chapter 6, “Callbacks and Threading,” provides some implementation techniques for handling callbacks from hardware device drivers in your adaptor. This chapter, together with Chapter 5, forms the basis for implementing hardware-clocked acquisition in your adaptor. For software-clocked adaptors, the information is not required.

Finally, experienced adaptor implementors wanting to understand the basic COM Interfaces defined by the Data Acquisition Toolbox and the Adaptor Kit should refer to the Appendices, which contain references for the interfaces and for structures defined by the engine.

2

Tutorial

Overview

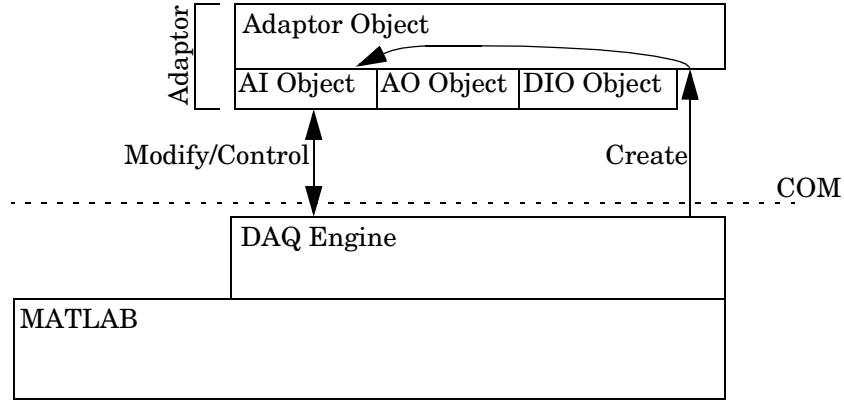
This chapter explains, by way of an example data acquisition session, how a typical user interacts with the Data Acquisition Toolbox, and how those user commands are handled by the engine and the adaptor. The examples include

- An analog input session
- An analog output session
- A digital I/O session

This chapter provides an understanding of how user commands are interpreted by the adaptor. However, no actual C code is presented in this chapter; the implementation details are deferred to Chapter 3, “Step-by-Step Instructions for Adaptor Creation.”

A Basic View of Toolbox-Engine-Adaptor Relationships

As discussed in “Toolbox Architecture” in Chapter 1, the Data Acquisition Toolbox consists of M-files, the data acquisition engine, and adaptors. Each of these components is used in a typical data acquisition session; although the user only interfaces to the hardware through MATLAB code, the MATLAB code uses the engine to create and manage the required data acquisition object, and the engine uses the adaptor to control hardware and those properties’ changes that are deemed to be important to the adaptor. These relationships are shown graphically below.



Example: an Analog Input Session

A typical toolbox session using an analog input object is shown.

```
ai = analoginput('winsound');
set(ai,'SampleRate',11025)
set(ai,'Tag','WinsoundObject')
addchannel(ai,1);
set(ai.Channel,'InputRange',[-.5 .5])
start(ai)
waittilstop(ai, 5);
data = getdata(ai);
delete(ai.Channel(1))
delete(ai)
```

Each command is described below.

Creating an Analog Input Object

The following command creates an analog input object associated with a sound card.

```
ai = analoginput('winsound');
```

The `analoginput` M-file calls the data acquisition engine to construct the `analoginput` object. When the constructor is first called, the engine must determine what COM object to create. It does this by enumerating all class IDs of objects that implement CATID {6FE55F7B-AF1A-11D3-A536-00902757EA8D} (MATLAB Data Acquisition Adaptor), and then asks for the short name of that GUID. In this case, the engine matches the short name to the `winsound` adaptor. The engine then constructs an `mwAdaptor` object and calls the object's `OpenDevice` method for creating the analog input object.

The adaptor's `OpenDevice` method is responsible for creating a new device and initializing it. Typically, this is done by creating a new COM object that implements the appropriate interfaces. After creating the new object, the engine interface can then be used to identify the characteristics of the current driver or device to the MATLAB user. You can also create device-specific properties at this time. The adaptor can also register an interest in some properties by setting the User value of the property. This value serves two purposes: Any value other than 0 causes the engine to call the `SetProperty`

method when the property is changed, and the value can be used in the SetProperty method to identify the property being modified.

The Open method creates any device-specific properties and defines any device-specific values for existing properties. For example, the winsound adaptor has two device-specific properties: BitsPerSample and StandardSampleRates. Both these properties are created with the CreateProperty method of the IPropContainer interface. When the property is created, a pointer to the IProp interface for the property just created is returned that allows you to call IProp methods. The IProp methods allow you to configure your property. For example, the IProp interface contains methods that allow you to display the possible settings of the property, the default value of the property, and the current value of the property.

Configuring the Sampling Rate

The following command configures the sound card to a sampling rate of 11.025 kHz.

```
set(ai,'SampleRate',11025)
```

The set M-file calls the data acquisition engine. In the Open method the adaptor requested a notify on change for the SampleRate property, and so the engine notifies the adaptor when you set the property to a new value. The data acquisition engine calls the adaptor's SetProperty method with two input arguments. The first input argument is a pointer to the IProp interface for the property being set. The second input argument is the value that the property is being set to. Therefore, in this example, the first input argument is a pointer to the SampleRate IProp interface, and the second input argument contains a pointer to 11025.

From within the adaptor's SetProperty method, you can determine which property is being set by examining the user value passed into the function. This value can be compared to the values for each property that you have registered with the engine.

Configuring the Object Tag

The following command configures the analog input object's Tag property to the string WinsoundObject.

```
set(ai,'Tag','WinsoundObject');
```

The set M-file calls the data acquisition engine. The Tag property was not registered by the adaptor. Therefore, when you configure the property, the engine modifies the value and does not notify the adaptor of the change.

Adding Channels to the Analog Input Object

The following command adds one channel to the analog input object ai.

```
addchannel(ai,1);
```

The addchannel M-file calls the data acquisition engine. The engine then calls the adaptor's ChildChange method. This gives the adaptor the opportunity to initialize the hardware and do any error checking for the channel that is added.

Configuring the Channel's Input Range

The following command configures the channel's InputRange property to accept voltages between -5 and 5 volts.

```
set(ai.Channel,'InputRange',[-.5 .5]);
```

The set M-file calls the data acquisition engine. The engine then calls the adaptor, because the InputRange property was registered with the engine (within the adaptor's Open method). The data acquisition engine calls the adaptor's SetChannelProperty method. SetChannelProperty takes four input arguments. The first input argument is a pointer to the IProp interface for the channel property being modified. The second input argument is a pointer to the IPropContainer interface for the channel being modified. The third input argument contains a pointer to the NESTABLEPROP structure, which is described in Appendix C, "Engine Structures." The last input argument contains the new property value.

Note The InputRange property is typically a combination of the hardware device's input range and the gain for a channel. For example, a hardware input range of +/- 5 V with four gain settings of 1, 2, 5, and 10 results in possible InputRange values of [-5 5], [-2.5 2.5], [-1 1], and [-0.5 0.5].

Starting the Analog Input Object

The following command starts the analog input object.

```
start(ai);
waittilstop(ai, 5);
```

The `start` M-file calls the data acquisition engine. The engine then calls the adaptor's `Start` method.

The `Start` method is responsible for initializing any routines necessary for acquiring data from the hardware. Because triggering is by default immediate, the engine then calls the adaptor's `Trigger` method, which starts the acquisition. The adaptor must then run in the background using callbacks or a separate thread. The buffers of data are transferred between the adaptor and the data acquisition engine with the `GetBuffer` and `PutBuffer` methods of the `IDaqEngine` interface. The adaptor uses the `GetBuffer` method to obtain an empty buffer from the data acquisition engine. When the buffer is filled with data acquired from the hardware, the adaptor returns the buffer to the data acquisition engine with the `PutBuffer` method.

When the number of samples requested has been returned from the adaptor to the data acquisition engine, the engine calls the adaptor's `Stop` method.

The `waittilstop` M-file waits until the specified object has stopped, or a particular time has passed (in this case, 5 seconds). The engine knows that the adaptor has stopped when it receives a Stop Event notification from the adaptor.

Extracting Data from the Engine

The following command extracts all the data from the engine and stores it in the MATLAB variable `data`.

```
data = getdata(ai);
```

The `getdata` M-file calls the data acquisition engine, which returns the data buffered in the engine to the specified MATLAB variable. If the number of samples requested by `getdata` is not available, the engine blocks until the adaptor returns the number of samples requested, or errors if the time specified by `TimeOut` elapses.

Deleting a Channel

The following command deletes the channel from the analog input object.

```
delete(ai.Channel(1))
```

The `delete` M-file calls the data acquisition engine, which in turn calls the adaptor's `ChildChange` method.

Deleting an Analog Input Object

The following command deletes the channel from the analog input object.

```
delete(ai)
```

The `delete` M-file calls the data acquisition engine, which calls the adaptor's destructor method. This should stop the device (call the `Stop` method), if the device was running, and close the hardware.

Example: an Analog Output Session

A typical toolbox session using an analog output object is shown.

```
ao = analogoutput('winsound');
set(ao,'SampleRate',11025)
set(ao,'Tag','WinsoundObject')
addchannel(ao,1);
set(ao.Channel,'OutputRange',[-.5 .5])
data = sin(linspace(0,2*pi,8000));
putdata(ao,data')
start(ao)
waittilstop(ao, 2);
delete(ao.Channel(1))
delete(ao)
```

The `analgoutput`, `set`, and `addchannel` commands are not described here because they are functionally identical to the analog input commands described in "Example: an Analog Input Session" on page 2-3. The `sin(linspace())` command is not described because it is handled entirely within MATLAB. All other commands are described below.

Queuing Data in the Engine

The following command queues data in the engine.

```
putdata(ao,data')
```

The `putdata` M-file calls the data acquisition engine, and the data is converted to the native data type and stored within the engine for output to the hardware.

Starting the Analog Output Object

The following command starts the analog output object.

```
start(ao)
```

The `start` M-file calls the data acquisition engine, which in turn calls the adaptor's `Start` method.

The `Start` method is responsible for initializing any routines necessary for outputting data that has been queued in the data acquisition. It often primes the output with data before the trigger function is called. The engine then calls the `Trigger` function, at which point the hardware should be started. The

buffers of data are transferred between the adaptor and the data acquisition engine with the `GetBuffer` and `PutBuffer` methods of the `IDaqEngine` interface. The adaptor requests a buffer of data to be output from the data acquisition engine with the `GetBuffer` method. When the data buffer has been output to the hardware, the adaptor returns the empty buffer to the data acquisition engine with the `PutBuffer` method.

For analog output objects, the adaptor must determine when the last buffer of data is available for being output, call its own `Stop` method, and post a `Stop` event to the object's `EventLog` property. The last buffer can be detected with the `Flags` field of the `BUFFER_ST` structure. The last buffer can also be detected if the buffer obtained by the `GetBuffer` method of the `IDaqEngine` interface is null. An event can be posted with the `IDaqEngine's DaqEvent` method.

Deleting a Channel

The following command deletes the channel from the analog output object.

```
delete(ao.Channel(1))
```

The `delete` M-file function calls the data acquisition engine. The engine then calls the adaptor's `ChildChange` method. The adaptor configures the hardware and performs any necessary error checking for the channel that is being deleted.

Deleting an Analog Output Object

The following command deletes the analog output object.

```
delete(ao)
```

The `delete` M-file calls the data acquisition engine. The engine then calls the adaptor's destructor method. This should stop the device (call the `Stop` method) and close the hardware.

Example: a Digital I/O Session

A typical toolbox session using a digital I/O object is shown.

```
dio = digitalio('nidaq',1);
lin = addline(dio,0:3,'in');
lout = addline(dio,4:7,'out');
p = addline(dio,0:7,1,'in');
data = getvalue(lin);
putvalue(lout,5)
data2 = getvalue(dio);
delete(dio)
```

Each command is described below.

Creating a Digital I/O Object

The following command creates the DIO object `dio` associated with a National Instruments board.

```
dio = digitalio('nidaq',1);
```

A digital I/O (DIO) device need not implement all the interfaces that are required for an analog input or analog output device. When the device is opened, it must fill in the `portdirections`, `portids`, `portlineconfig`, and `portlinemask` properties with the correct values. Given these values, the engine maintains the line information and generates the correct calls to `SetPortDirection`, `ReadValues`, and `WriteValues`. The standard property and child (line) property methods are supported. However, the adaptors implemented so far have not needed to use them.

The object should initialize its properties to the correct values before returning. For a DIO object, the `daqhwinfo` property structure must initialize values for `portdirections`, `portids`, `portlineconfig`, and `portlinemasks`.

Adding Lines to the Digital I/O Object

The following command adds four input lines from the default port (port 0) to the DIO object `dio`.

```
lin = addline(dio,0:3,'in');
```

The `addline` command works the same as the `addchannel` command for AI and AO objects in that the adaptor's `ChildChange` method is called. However, most

adaptors need not implement `ChildChange`, because typically no adaptor actions are necessary when adding or removing lines.

The following command adds four output lines to the DIO object `dio`.

```
lout = addline(dio,4:7,'out');
```

After the lines are added, a call to `SetPortDirection(0,0xf0)` is made to set the port direction to output.

The following command demonstrates that you can also add lines in reverse order.

```
p = addline(dio,7:-1:0,1,'in');
```

Reading Line Values

The following command reads the values from lines 0 to 3 of port 0 and stores the values in the MATLAB variable `data`.

```
data = getvalue(lin);
```

The engine issues the command `ReadValues(1,PortList,Data)` to the device, which must then return the values from the specified ports. The adaptor does not keep track of exactly what lines have been added, and returns all line values in `Data`.

Writing Line Values

The following command writes the value 5 to lines 4 through 7 (the four most significant bits) of port 0.

```
putvalue(lout,5);
```

The write is performed by calling `WriteValues(1,PortList,Data,Mask)` where `Portlist`, `Data`, and `Mask` are pointers to an array. `Portlist` points to 0, `Data` points to 0x50, and `Mask` points to 0xf0.

The following three commands illustrate alternative ways to write the value 5 to port 0:

```
putvalue(lout,[1,0,1,0])

putvalue(dio.lines(8:-1:5),10);

putvalue(lout(4:-1:1),[0,1,0,1])
```

Reading Line Values

The following command reads the values from all currently configured lines:

```
data2 = getvalue(dio);
```

The read is performed by calling `ReadValues(2,PortList,Data)`.

Deleting a Digital I/O Object

The following command deletes the channel from the digital I/O object:

```
delete(dio);
```

It is up to the implementation to decide what state any output lines are left in. The engine releases its reference to the `mwDevice` object and then releases its reference to the `mwAdaptor` object.

Note The engine implements a pseudo line system and caches the values written to output lines. It also takes care of reordering the lines (and data) for the user.

3

Step-by-Step Instructions for Adaptor Creation

Overview: Building the Adaptor

This chapter provides step-by-step instructions for building an adaptor for your hardware. Starting with the demo adaptor provided with the Data Acquisition Toolbox, you can develop a complete adaptor, implementing all the functionality available in the Data Acquisition Toolbox for your hardware, using these instructions.

In this chapter, you learn how to build the adaptor by following these stages:

1. Choose the features of the Data Acquisition Toolbox the adaptor will implement.
2. Create the Adaptor project and Adaptor class, based on the demo adaptor supplied by The MathWorks.
3. Implement the Analog Input object code (if required).
4. Implement the Analog Output object code (if required).
5. Implement the Digital I/O object code (if required).

For each of the stages, the specific actions required to complete that stage are discussed in this chapter. The stages have been designed so that testing can take place often, and changes are typically restricted to a few files and methods within one class.

Note Although this chapter discusses the steps required to implement each stage, details of how to interact with properties, deal with buffers, and handle event messaging are documented in later chapters. Refer to those chapters as necessary.

The stages of development rely heavily on the demo adaptor source code provided with the Adaptor Kit. In many instances, this document also refers to existing adaptors supplied with the Data Acquisition Toolbox. Refer to the code for these adaptors where necessary.

Toolbox Adaptors

The technologies used in the adaptors shipped with the Data Acquisition Toolbox have been presented in this document as approaches for implementing your own adaptor. Each adaptor provides a unique combination of the implementation approaches presented in this manual. The following sections explain how each adaptor has been implemented. The code for each adaptor is in a subdirectory of the Data Acquisition Toolbox. You can find the source code in the `$MATLABROOT\toolbox\daq\daq\src` directory.

The winsound Adaptor

The `winsound` adaptor is used to communicate with Windows-compatible sound cards. The adaptor uses the Windows multimedia drivers and buffers acquired data using multibuffering with direct callback threads.

This adaptor is the most basic of all the adaptors. However, because of the power of the Windows multimedia device interface, it uses an efficient acquisition method: This adaptor uses a linked list of buffers to acquire or output data. The multimedia device is capable of filling (or emptying) these buffers in the order that they are passed to the device. A thread is created to feed buffers to the device from the engine, and to take the filled buffers from the device and return them to the engine. The thread is paced with an event generated by the device driver each time a buffer is filled. This driver also supports variable data types.

Note This adaptor was written prior to complete implementation of the current Adaptor Kit. Although the concepts used in the adaptor are similar to those presented here, you will find that some of the actual implementation of code is more low-level than the ideas presented in this document.

The cbi Adaptor

The `cbi` adaptor is used to communicate with ComputerBoards devices (ComputerBoards are now called Measurement Computing, but the adaptor is referred to in this document as the ComputerBoards adaptor). The adaptor uses the Universal Library drivers, and buffers acquired data using circular buffers with timer callbacks. It also implements software clocking.

The ComputerBoards adaptor has two unique features: First, because the Universal Library does not support callbacks, this adaptor uses a timer to poll the current acquisition. It does this by using the Windows multimedia timer callback. The current transfer location is obtained from the Universal Library, and the appropriate amount of data is then copied into or out of the circular buffer. One disadvantage of this method is that there is no hardware guarantee or protection for an overrun or an underrun condition. The adaptor tries to pick a buffer size and a timer callback rate such that an overrun is unlikely, but there is still the possibility that data can be lost.

The second unique feature of this adaptor is the support for software clocking. Because some ComputerBoards devices do not have an onboard clock, this adaptor implements a software clock based on the Windows multimedia timer.

Note The ComputerBoards adaptor makes extensive use of many of the Adaptor Kit macros used in this document. However, the ComputerBoards adaptor does not implement the adaptor object separately from the analog input class, so you should not use the entire adaptor as a template for creating your own.

The nidaq Adaptor

The `nidaq` adaptor is used to communicate with National Instruments devices. The adaptor uses the NI-DAQ driver, and buffers acquired data using circular buffers with direct callbacks.

The NI-DAQ adaptor is one of the more extensive adaptors because of its implementation of advanced triggering modes and the number of hardware devices supported. It works by acquiring data to or from a circular buffer using NI-DAQ's callback and copy functions. A circular buffer is used because it is the buffering mode supported by the NI-DAQ software. Many advanced triggering modes are also supported by this adaptor. When the number of samples is known and is sufficiently small, a burst acquisition is performed instead of using continuous acquisition.

Note This adaptor was written prior to complete implementation of the current Adaptor Kit. Although the concepts used in the adaptor are similar to those presented here, you will find that some of the actual implementation of code is more low-level than the ideas presented in this document.

The hpe1432 Adaptor

You use the `hpe1432` adaptor to communicate with Agilent Technologies E1432/33/34 devices. The adaptor uses the VXIplug&play driver, and buffers acquired data using ping-pong buffers with callbacks.

For input data, the adaptor uses an unknown buffering method that is internal to the VXIplug&play driver, and a callback to notify the adaptor when data is available. In the callback function, a buffer of data is retrieved from the driver and returned to the engine. For output data, the adaptor uses two buffers and a vendor-supplied callback to send the data. The buffer size is defined by the driver to have a maximum of 4096 values. Therefore, to simplify the copy process, the adaptor limits the engine to this maximum buffer size. It also supports more than 16 bit data output.

Note This adaptor was written prior to complete implementation of the current Adaptor Kit. Although the concepts used in the adaptor are similar to those presented here, you will find that some of the actual implementation of code is more low-level than the ideas presented in this document.

The keithley Adaptor

You use the `keithley` adaptor to communicate with Keithley Instruments devices. The adaptor uses the DriverLINX set of drivers, and implements direct engine-driver buffer transfers using window messaging.

The Keithley adaptor supports a wide range of Keithley Instruments boards. Because each board series uses a different device driver, the adaptor opens all available DriverLINX drivers at initialization, and closes them when the adaptor is destroyed. The adaptor can switch between software-clocked and hardware-clocked acquisition as required.

A unique feature of the Keithley adaptor is the implementation of Window message handling to monitor task progress. The adaptor uses a single message window for all DriverLINX messaging, passing the message to the appropriate subsystem as required. However, due to the implementation of the DriverLINX drivers, the message window thread has to open all DriverLINX drivers in order to receive any messages from those DLLs. Hence, two instances of DriverLINX are open at any one time per driver installed on the machine.

The direct engine-adaptor buffering places some limitations on the minimum size of the engine buffers. This is not enforced, but is rather communicated to the user through warning messages when appropriate.

Note The Keithley adaptor has been implemented based solely on the current Adaptor Kit ideas. You should refer to the Keithley adaptor for code examples where possible.

About the Demo Adaptor Software

The demo adaptor does not communicate with any actual hardware. Instead it simulates data acquisition in order to demonstrate the basic functionality common to most adaptors.

Features

The demo adaptor supports these features:

- Buffered acquisition
- Manual, software, and auto triggering
- Single-sample acquisition
- Saving (retrieving) collected data to (from) a MATLAB internal array

Limitations

The demo adaptor has these limitations:

- It works only with simulated data. The data is stored in the buffer immediately after you open the device. This same buffer is reused whenever data is required.
- It only supports software clocking. Note that the maximum software-clocked sampling rate is 500 samples per second. Therefore, any adaptor you build that uses software clocking includes this limitation. To achieve higher sampling rates, you must use your hardware's onboard timer. However, supporting an onboard timer requires an entirely different software design, which is described later in this guide.

Modifying the Demo Adaptor

Modification of the demo adaptor takes place in each of the stages defined in this chapter. Although you could simply modify the code from the demo adaptor and create an adaptor named "demo" by building that modified adaptor code, this chapter leads you through the process of creating your own project and importing components of the demo adaptor into that project one at a time. In this way, you can test modifications and restrict problems to a single file in the project, which makes implementation quicker and easier to deal with.

The demo adaptor code contains many

```
// TODO
...
// END TODO
```

comment segments. These segments of comment code should be used in conjunction with the steps outlined in this chapter, in order to produce a successful adaptor with minimal trouble.

Stage 1 Select Supported Features

The first stage of writing your adaptor is deciding which features of the Data Acquisition Toolbox to support. Your decisions should be based on

- Hardware capabilities: Can the hardware provide the specified feature?
- Driver knowledge: Does the software driver support the required programming requirements?
- Available time: Some aspects of implementation can be completed relatively quickly, while others require more programming and testing time.

In general, these decisions are driven by the first two points, and only rarely by the last.

The following questions provide an implementation roadmap for you to follow. The questions provide a hierarchy of implementation possibilities based on implementation complexity; as the list continues (for each subsystem) the implementation becomes more complex. Each successive point is also inclusive: to implement that point requires implementation of each previous point.

1. Will the adaptor support analog input?
   a. Single-value transfers only?
   b. Buffered transfers (logging to memory and/or disk) using software clocking?
   c. Hardware-clocked buffered transfers?
   d. Hardware triggering and/or gated acquisition?
2. Will the adaptor support analog output?
   a. Single-value transfers only?
   b. Buffered transfers (logging to memory and/or disk) using software clocking?
   c. Hardware-clocked buffered transfers?
   d. Hardware triggering and/or gated acquisition?
3. Will the adaptor support digital I/O?
   a. Will any digital ports be configurable for write/read?

b Will any digital lines be configurable for write/read?

Based on the questions posed above, a roadmap to implementation can be identified. This roadmap is presented in the following table as methods that must be implemented for each of the steps defined above.

Table 3-1: Classes and Methods to Be Implemented in the Adaptor

Question	Class/Methods to Implement
1) Analog Input	`AnalogInput` class (derived from `ImwDevice` and `ImwInput`) `Open`, `SetDaqHwInfo` methods.
1a) Single-value A/D	`GetSingleValue` and/or `GetSingleValues` methods.
1b) Software-clocked acquisition	No additional methods are required, but the adaptor must call `EnableSwClocking` to set up correct sample rates.
1c) Hardware-clocked acquisition	`Start`, `Trigger`, `Stop` methods, and probably `SetProperty`, `ChildChange`, and `SetChannelProperty` methods, as well as a message handler.
1d) Hardware triggering or gated acquisition	Modify `Start`, `Trigger`, and `Stop` methods, as well as property change methods.
2) Analog Output	`AnalogOutput` class (derived from `ImwDevice` and `ImwOutput`) `Open`, `SetDaqHwInfo` methods.
2a) Single-value D/A	`PutSingleValue` and/or `PutSingleValues` methods.
2b) Software-clocked transfer	No additional methods are required, but the adaptor must call `EnableSwClocking` to set up correct sample rates.
2c) Hardware-clocked transfer	`Start`, `Trigger`, `Stop` methods, and probably `SetProperty`, `ChildChange`, and `SetChannelProperty` methods; message handler.
2d) Hardware triggering or gated transfers	Modify `Start`, `Trigger`, and `Stop` methods, as well as property change methods.

Table 3-1: Classes and Methods to Be Implemented in the Adaptor (Continued)

Question	Class/Methods to Implement
3) Digital I/O	`DigitalIO` class (derived from `ImwDevice` and `ImwDIO`) `Open`, `SetDaqHwInfo`, `WriteValues`, `ReadValues` methods.
3a) Port-configurable I/O	`SetPortDirection` method, trapping `DirectionValue` of `0` (Input) or `0xff` (Output).
3b) Line-configurable I/O	`SetPortDirection` method, with variable `DirectionValue` settings.

Stages 2 through 4 discuss how to implement each of these methods.

By the end of Stage 1, you will have a roadmap defining how you will implement your adaptor. Refer also to Appendix A, "Adaptor Kit Interface Reference," and Appendix B, "Engine Interface Reference," for information on the methods that all adaptors should implement (and which methods are implemented in the Adaptor Kit).

Limitations of Software-Clocked Adaptors

One of the most important implementation issues is whether to support hardware clocking in your adaptor. As long as you use the Adaptor Kit code, software clocking is already implemented for your adaptor, and requires minimal effort, as outlined in the section above. However, software clocking has some limitations which might be too severe for your application:

- The maximum sample rate for any acquisition task is 500 Hz, regardless of the board's published sampling rate and your computer's processor speed.
- You cannot use any hardware triggering without rewriting substantial portions of the adaptor code.

If you are able to achieve your desired objective within these limitations, then you should not use anything other than software clocking. A complete adaptor, however, should use the full features of the hardware for which the adaptor has been written, and should implement hardware clocking.

Stage 2 Create the Adaptor Project and Adaptor Class

Once you have selected the required implementation details, you can create the adaptor project. Use of a suitable compiler and IDE is required for this task. This chapter assumes the use of Microsoft Visual Studio 6, Service Pack 4.

Although you are starting a new Microsoft Visual Studio project, you will make extensive use of the demo adaptor code shipped with the Adaptor Kit by importing that code into your project and modifying it. The benefit of starting a new project is that modifications to the existing demo adaptor code are more manageable. Adding the demo adaptor code initializes all Data Acquisition Toolbox Engine interfaces, and creates shell classes and methods for implementation of the custom-written adaptor.

The following sections describe how to implement this stage by completing the following steps:

1. Choose a suitable name for your adaptor.
2. Create the Microsoft Visual Studio project.
3. Add the demo adaptor code to that project (including renaming the demo files).
4. Test the adaptor with MATLAB.

In Stages 3 to 5, you will implement each of the subsystems of the adaptor.

Step 2.1 Adaptor and Project Naming

Before creating the adaptor, you should select a suitable name. In many cases, the name would be the name of the data acquisition board manufacturer or model. For example, if you are creating a board manufactured by “XYZ Instruments” a suitable name is “xyz”. The name should not begin with numbers, and should be sufficiently unique and representative of the capabilities of the adaptor (if you are writing for a particular board, say the “ad123”, you should name your adaptor after that board, i.e., “xyzad123”).

You must use all lowercase for the adaptor name, to conform to conventions used on existing adaptors.

This adaptor kit provides examples based on a chosen adaptor name of “xyz”.

Note Do not use the word “adaptor” in your project name, as the project name is used extensively in the COM object naming! Otherwise a user would have to create an adaptor by referring to it as the “xyzadaptor” instead of just by the board name, “xyz”.

Once you have named the adaptor, you can create the adaptor project by starting a new project using the “ATL Com AppWizard”. You should not need to use MFC, as all interaction with the adaptor takes place through MATLAB and not other windows (the only exception is when you are creating adaptors that use MFC, for example, the winsound adaptor).

All adaptors created to date are in-process servers. For the AppWizard, this means selecting “Dynamic Link Library” as the server type. Use of out of process servers has not been tested, and is likely to be more difficult, although not impossible.

The created project forms the shell of the adaptor.

Step 2.2 Add Include, Link, and MIDL Directories to Your Project

To successfully compile the adaptor code, you must add the `$MATLAB\Toolbox\daq\daqadaptor\AdaptorKit` directory to the following include paths (*`$MATLAB`* refers to the directory in which MATLAB is installed):

- To the “Additional include directories” option of the C/C++ Preprocessor definitions panel.
- To the “Additional resource include directories” option of the Resources panel.
- To the “Additional include directories” option of the MIDL panel.

Next you must add the `Adaptorkit.cpp` file to your project. You also need to modify the `StdAFX.h` file to include the adaptor kit header file `adaptorkit.h`.

You need to enable exception handling in your project. Select Project Settings and select “All Configurations”. In the C/C++ tab, select “C/C++ Language” and select the **Enable exception handling** box.

Step 2.3 Define Adaptor Classes in the IDL File

The IDL file contains a definition of all COM interfaces defined in the project. In this stage, you add the adaptor interfaces to the project, and the IDL file.

You should complete the following tasks in this stage (consult the demo adaptor IDL file `demo.idl` for more information):

- Add a line to import the `daqmex.idl` file.
- Copy the implementation of the `demoadapt` class into your IDL file. The specific lines to copy are given below:

```
[
    uuid(CE932327-3BD9-11D4-A584-00902757EA8D),
    helpstring("demoadapt Class")
]
coclass demoadapt
{
    [default] interface ImwAdaptor;
};
```

Note You must change the UUID by running GUIDgen to create a new UUID, and you must change the references to "`demo`" to the name of your adaptor.

Note that this definition must appear within the definition of the type library.

Step 2.4 Add the Demo Adaptor Class Code

Copy the files `demoadapt.h` and `demoadapt.cpp` into your project directory, and rename them by replacing "`demo`" with your adaptor name. The best way to do this is to perform a global search and replace on both files, replacing "`demo`" with the name of your adaptor.

You must remove or comment out some portions of the adaptor code in order to test the adaptor in the next step.

- Remove the `#include` lines that import the Analog Input and Analog Output header files into your adaptor code. You will add those in later stages.
- Comment out the blocks of text surrounded by

```
//TO_DO
...
//END TO_DO
```

code segments in the adaptor's `OpenDevice` method. You will require that code in later stages, and you do not actually create a device until that stage.

Make the following changes to the main project file (`xyz.cpp` in this example):

- Add a `#include` for the adaptor header in the main project.
- You also need to add the adaptor definition to the OBJECT MAP in the main project file. As an example, the following entry would be made for the adaptor named "`xyz`":

```
BEGIN_OBJECT_MAP(ObjectMap)
   OBJECT_ENTRY(CLSID_xyzadapt, Cxyzadapt)
END_OBJECT_MAP()
```

The class ID of your adaptor is generated automatically by the MIDL compiler.

Building and Testing the Demo Adaptor

You should now be able to build and test your demo adaptor, using the Debug settings. Once the adaptor has successfully built, you should be able to launch MATLAB and register and query the adaptor using the following MATLAB code:

```
daqregister <PathToProject>\Debug\<adaptorname>.dll
daqhwinfo <adaptorname>
```

where *<PathToProject>* is the full path to your adaptor project and *<adaptorname>* is the name of your adaptor. You should get back fictitious results similar to the following:

```
» daqhwinfo xyz
ans =
           AdaptorDllName: 'c:\xyzadaptor\xyz.dll'
        AdaptorDllVersion: '1, 0, 0, 1'
              AdaptorName: 'xyz'
               BoardNames: {'xyz Board 0'  'xyz Board 1'}
        InstalledBoardIds: {'0'  '1'}
    ObjectConstructorName: {2x3 cell}
```

If this is successful, you are ready to implement the next step of the adaptor. If the results are not similar, or if you get an error message, you need to check that all steps outlined previously have been carried out.

Step 2.5 Modify the Adaptor Class AdaptorInfo() Method

In Step 2.3, the adaptor reported fictitious board information. The previous step simply confirmed that the adaptor was compiling correctly, and registering as a valid data acquisition adaptor object. In the final step of Stage 2 you should implement code that checks the installed hardware on the system by modifying the `AdaptorInfo` method of the adaptor class to provide information about the installed hardware systems supported by the adaptor. Typically, this would take the form of querying a device driver for any board information that it can find, from the registry or other means.

For an example of how `AdaptorInfo` is implemented, consult the Keithley or ComputerBoards adaptors.

Note The `AdaptorInfo` method makes extensive use of SafeArrays to pass information back to MATLAB. See "Passing Arrays to MATLAB Using Safe Arrays" in Chapter 4 for some ideas on how to do this in your adaptor.

The objective of this step is to make the information reported by a call to

```
daqhwinfo <adaptorname>
```

return the correct information for installed boards. For example, a call to the Keithley adaptor on a machine with four hardware devices produces the following results:

```
>> d=daqhwinfo('keithley')
d =
           AdaptorDllName: [1x58 char]
        AdaptorDllVersion: 'Version 1.1   (R12.1+) 23-Aug-2001'
              AdaptorName: 'keithley'
             BoardNames: {'KPCI-1801HC'  'KPCI-3110'  'KPCI-3108'
'KPCI-PIO96'}
        InstalledBoardIds: {'0'  '5'  '1'  '2'}
    ObjectConstructorName: {4x3 cell}
```

Typically device drivers provide some mechanism for differentiating between multiple hardware devices. The Data Acquisition Toolbox assumes that boards are referenced by a unique integer board identifier, listed as the `InstalledBoardIds` field from the result of the `daqhwinfo` call. If the device driver for your adaptor does not implement this system, you should enforce a board identifier on each unique board found by the device drivers, using an appropriate convention. In the example above, four Keithley Instruments boards have been installed on the machine, and their board IDs are 0, 5, 1, and 2, respectively.

The `ObjectConstructorName` field returned by the `daqhwinfo` call lists the object constructor code that can be used to create each of the Analog Input, Analog Output, and Digital I/O subsystems (the columns of the cell array) for each board (the rows of the cell array). If your adaptor is not implementing any of the subsystems, you should remove the constructor string for that particular subsystem. Similarly, if some boards implement a system while others do not, you should leave the unsupported subsystem columns empty for that row of the cell array.

Stage 3 Implement the Analog Input Subsystem

Although all subsystems of an adaptor should be considered equally important, the most commonly implemented subsystem is analog input (since most users of the Data Acquisition Toolbox require measurement of real-world signals). The Adaptor Kit therefore implements the analog input subsystem as the first subsystem of the adaptor. This stage presents a significant discussion of the techniques used in developing the adaptor's subsystems; the steps required to implement the Analog Output and Digital I/O subsystems are similar. Stages 4 and 5 therefore draw heavily on material discussed in this stage.

Implementation of the Analog Input subsystem takes place in the following steps:

1. Select the default values, ranges, and other characteristics of the analog input subsystem properties.
2. Create the Analog Input COM interface and class definitions in the `IDL` file, and incorporate the demo adaptor analog input implementation in your project.
3. Modify the `OpenDevice` method of the adaptor class to create the required subsystem when requested.
4. Modify the `Open` and `SetDaqHwInfo` methods of the Analog Input class to handle device initialization, create custom properties, and set defaults and ranges for all properties.
5. Implement the `SetProperty` and `SetChannelProperty` methods of the Analog Input class to handle property changes.
6. If necessary, overload the `ChildChange` method of the Analog Input class to handle channel addition and removal.
7. Implement the `GetSingleValue` method if software clocking is to be used.
8. Implement the `GetSingleValues` method if the device driver supports easy single acquisition from multiple channels.

9 Implement the `Start`, `Trigger`, and `Stop` methods for buffered acquisition. Typically, this step involves writing buffering routines and message handlers, and might require multithreading of the adaptor.

Each of these steps is discussed in detail in the following sections.

Note You should use the answers to the questions posed in Stage 1 to decide which of the preceding steps you will implement in your adaptor.

Step 3.1 Select Property Values, Ranges, and Defaults for Analog Input

In order to control the behavior of a task (such as duration and volume of acquisition, type of triggering, clocking, and event callbacks) the MATLAB user modifies the properties of the Data Acquisition Toolbox `analoginput` object representing the data acquisition hardware he/she is using. The adaptor must use the property values during acquisition tasks to control driver settings, return messages, and start and stop acquisition. The adaptor must also provide the data acquisition engine with appropriate properties, ranges, and default values for the specific hardware referenced by the adaptor. Successfully creating an adaptor therefore requires careful thought about the existing common Analog Input subsystem properties, and the addition of adaptor-specific properties where appropriate.

For both common and adaptor-specific properties, the adaptor might need to control default values of a property in response to user changes in any associated properties (for example, when the user changes the `ChannelSkewMode` property, the range for the `ChannelSkew` property needs to change to reflect the new mode). The first step in building any subsystem is to plan these default values and ranges and decide on any additional properties that are required in order to describe the hardware completely. For example, the Keithley adaptor implements stop triggers using various additional properties such as `StopTriggerType`, `StopTriggerChannel`, etc.

Typically, this step consists of compiling a *propinfo table* of all common properties for the particular subsystem, and filling in the following information:

- ***Type***: The MATLAB data type (one of `double` or `string`)

- ***Constraint***: The constraint on the property. Typically, this is `Bounded`, if the property lies within a defined range, `Enum` if the property is one of an enumerated list of available values, or `None` if the property can take on any value.
- ***Constraint Value***: The limits for a bounded constraint, or the list of possible values for an enumerated list.
- ***Default Value***: The value of the property when the object is first created. If this might change, indicate all possible values with a note.
- ***Read Only***: A flag indicating whether the property can be changed by the user.
- ***Read Only Running***: A flag indicating that the property cannot change while an acquisition task is running.
- ***Device Specific***: If the property is specific to the particular adaptor or is defined by the engine.
- ***Attach***: Whether the property is to be attached to (for information on attaching to properties, see "Attaching to a Property" in Chapter 4).
- ***Notes***: Any particular note about the property, including the source of the default values (for example, the driver, or an INI file), how the property might change based on other properties, and any additional implementation information.

The `propinfo` table should reflect the state of the desired output from a call to the `propinfo` method of the analoginput object. Thus, when you run

```
propinfo(analoginput(<adaptor>))
```

the result should be the data presented in the `propinfo` table. The output of the MATLAB code given above provides a test to confirm that these properties have been created and initialized successfully.

The table should include only those properties that are directly related to the device hardware, and should not include properties that are used for logging to disk, function callbacks, event information, or internal housekeeping.

For a complete list of properties supported by the adaptor, consult the *Data Acquisition Toolbox User's Guide*

A direct consequence of producing this table is discovering which properties your adaptor will have to monitor closely. Monitoring of a property involves registering that property with the data acquisition engine, effectively notifying

the engine that the adaptor has a particular interest in being notified whenever the user changes that property. This process, called *attaching* to the property, allows methods within the adaptor to be called whenever that property changes. Consequently, the adaptor would be able to change other property enumerated lists, or perform additional checks on a selection of properties to ensure that the subsystem does not perform illegal operations when an acquisition is started. For more information on attaching to properties, see "Attaching to a Property" in Chapter 4.

A sample `propinfo` table is presented in Table 3-2, showing only two properties from an adaptor.

Table 3-2: Sample of Propinfo Table for Analog Input Object

Property/Field	Value
`ChannelSkew`	
`Type`	Double
`Constraint`	fixed
`Constraint Value`	`5e-6` for `Minimum` `1/SampleRate` for `Equisample`
`Default`	`5e-6` if `Minimum` supported, else `1/1000`.
`ReadOnly`	1
`ReadOnly Running`	1
`Device Specific`	0
`Attach`	0
`Note`	Check driver for burst mode support.
`ChannelSkewMode`	
`Type`	String
`Constraint`	Enum

Table 3-2: Sample of Propinfo Table for Analog Input Object (Continued)

Property/Field	Value
Constraint Value	Minimum Equisample
Default	Minimum
ReadOnly	0
ReadOnly Running	1
Device Specific	0
Attach	1
Note	Mimimum only if ADBURST is supported. Change sets ChannelSkew.

The outcome of Step 3.1 is a document that forms the blueprint for the implementation of properties in later steps of this stage.

Step 3.2 Add the Demo Analog Input Code to Your Project

When the IDL file was created, only the adaptor component was included. The IDL file must expose the Analog Input interface to the engine, so that the engine can access the methods in the analog input implementation.

Both the interface and the class need to be defined in the IDL file. Specify the interface prior to the specification of the Adaptor Type Library, so that the library can use the new interface in the Analog Input class. The following tasks should be completed in this stage:

- Add the interface definition prior to the type library definition. The Analog Input interface should inherit from the IDispatch interface (although the IDispatch interface is not currently implemented, the engine assumes that the Analog Input interface inherits from IDispatch). Sample code for the XYZ adaptor follows:

```
[
    object,
    uuid(E721C893-C230-4eae-9F78-B33E30F74B4E),
```

```
    dual,
    helpstring("IxyzAin Interface"),
    pointer_default(unique)
]
interface IxyzAin : IDispatch
{};
```

- Be sure to change the UUID, using GUIDgen, so that your adaptor has a unique ID.
- Add the Analog Input class to the type library (if using the following code, be sure to change the UUID of your class using GUIDGen):

```
// Define the xyzAin class:
[
    uuid(83D8B96C-FE9A-46c7-AAB9-6B3C67FDE863),
    helpstring("xyzAin Class")
]
coclass xyzAin
{
    [default] interface IxyzAin;
    interface ImwDevice;
    interface ImwInput;
};
```

  The Analog Input class should inherit the `ImwDevice` and `ImwInput` interfaces, and your Analog Input interface defined in the adaptor.
- Copy the Analog Input files from the demo adaptor, renaming the files appropriately. In order to facilitate the search and replace operation described below, you should simply replace the word “demo” in the filenames with your adaptor name. For example, the file `demoain.cpp` would become `xyzain.cpp`.
- Search for all instances of the word “demo” and replace that with your adaptor name.

Note Remember to do this with both the `.cpp` and the `.h` files. Unfortunately, the current Microsoft Visual C IDE does not support search and replace across multiple files.

In the next step, you test the changes made in this step by providing sufficient code to enable the creation of the your analog input object as a duplicate of the demo adaptor with a new name.

Step 3.3 Modify the OpenDevice Method of the Adaptor Class

In this step of the implementation of the analog input object, you ensure that the renamed demo adaptor still works in MATLAB.

- Uncomment the analog input construction statements in the `OpenDevice` method of the Adaptor class. Be sure to include the Analog Input class header file in the adaptor class implementation file.
- Compile the project, and you should be able to launch MATLAB and create an analog input object for your adaptor:

```
ai = analoginput('xyz');
```

If the compilation fails, you should ensure that all header files have been created, and that your analog input class name is consistent throughout the project. Also ensure that all the steps from the previous stage have been implemented as well.

Your adaptor should now contain a complete analog input object, which implements software clocking on a fake acquisition channel (the acquisition returns a sine wave from an internal function, and not from any hardware device). In future stages of the analog input implementation, you will progressively implement hardware acquisition tasks. The first of these stages is to ensure that the adaptor's properties are initialized correctly, possibly from hardware device detection.

Step 3.4 Modify the Analog Input Open and SetDaqHwInfo Methods

The `Open` method is called by the Adaptor class `OpenDevice` method to create an analog input subsystem for a particular device. The `Open` method is responsible for checking that the required device has the requested subsystem, initializing the subsystem hardware (if necessary), and configuring the subsystem properties correctly.

You create analog input subsystems by specifying two parameters: the adaptor name and the device ID. Your adaptor code needs to use the device ID to

provide the device driver with a specific reference to the desired hardware device. The device ID is passed as the second parameter of the `Open` method. The first parameter is a pointer to the Engine interface, and this should also be stored by the adaptor to gain access to engine methods.

The `Open` method typically also calls the `SetDaqHwInfo` method, to define hardware and driver information for the subsystem.

The demo adaptor provides a sample `Open` method that illustrates the basic ideas of opening an analog input subsystem:

- The base engine's `Open` method is called. This creates a link to the engine and defines a variable `_engine` containing a pointer to the engine's interface methods.
- The `ClockSource` and `InputType` properties are initialized as remote properties (for a discussion of properties, see Chapter 4, "Working with Properties").
- The `ClockSource` and `InputType` properties are modified to reflect the demo adaptor state more closely.
- The `DaqHwInfo` properties are set by the call to `InitHwInfo`.

All adaptors need to follow the preceding pattern (the `InitHwInfo` method can be replaced by a call to `SetDaqHwInfo`), and include the following additional code:

- Code to initialize the device driver and hardware, if necessary
- Code to check the device ID to confirm that the requested hardware exists
- Code to attach to the required properties listed in Step 3.1 (see "Attaching to a Property" in Chapter 4)
- Code to create adaptor-specific properties

A full discussion of interaction with properties is given in Chapter 4, "Working with Properties."

Because all `Open` methods typically include a call to `SetDaqHwInfo`, you should implement `SetDaqHwInfo` prior to testing the hardware-specific adaptor.

The Data Acquisition Toolbox allows the user to query a specific subsystem to obtain hardware information for that specific subsystem. A MATLAB user would type the following code to obtain information about the XYZ adaptor analog input device 5:

```
>> daqhwinfo(analoginput('xyz', 5))
```

Note You could split this MATLAB statement into two by assigning the `analoginput` object to a variable. Either way, the Data Acquisition Toolbox first creates the subsystem and then provides hardware information about that subsystem.

The information for a call to a subsystem's `daqhwinfo` method is obtained by calling the `SetDaqHwInfo` method of the requested subsystem. The `SetDaqHwInfo` method should also be called by the `Open` method to initialize these values on startup.

The `SetDaqHwInfo` method should use the `_daqhwinfo IPropContainer` member variable defined in the `ImwDevice` interface class. Every subsystem inherits from `ImwDevice`, and so already defines the `_daqhwinfo` member variable. You put information into the `_daqhwinfo IPropContainer` by using the `IPropContainer put_memberValue` method. Table 3-3 lists the values that you should place in `_daqhwinfo`, and describes the type of data that should be used for each member value. For sample code, see the `SetDaqHwInfo` method of any of the adaptors included with the Data Acquisition Toolbox.

Table 3-3: SetDaqHwInfo Member Variables and Descriptions

Member Value	Description	Data Type (C++)
`adaptorname`	Name of the adaptor	String
`bits`	Resolution of analog input subsystem	Double
`coupling`	Device coupling (can be set to "Unknown")	String
`devicename`	Name of device. Typically, the name is given as "`<adaptorname>AI-<id>`".	String
`differentialids`	Channel IDs for differential channels (or empty if differential mode is not supported)	SafeArray (1 x nD)
`gains`	Allowable gains for channels	SafeArray (1 x nG)

Table 3-3: SetDaqHwInfo Member Variables and Descriptions (Continued)

Member Value	Description	Data Type (C++)
`id`	Device ID	Double
`inputranges`	Allowable input ranges	SafeArray (n x 2)
`maxsamplerate`	Maximum permissible sample rate	Double
`minsamplerate`	Minimum permissible sample rate	Double
`nativedatatype`	The data type of raw data that MATLAB can send to the hardware device. The engine uses this value to convert voltages to units that the hardware understands.	VariantType
`polarity`	Vector of allowable polarities	SafeArray
`sampletype`	Either Scanning or Simultaneous Sample and Hold (SSH)	Enum
`singleendedids`	Channel IDs for single-ended channels (or empty if single-ended mode is not supported)	SafeArray (1 x nS)
`subsystemtype`	Always set to `"AnalogInput"`	String
`totalchannels`	Total number of available channels	Double
`vendordriverdescription`	Description of the vendor driver used by the adaptor for this device	String
`vendordriverversion`	Version information for the vendor driver used by the adaptor for this device	String

Compiling and Testing Analog Input Property Creation

After creating the `DaqHwInfo` method and implementing the `Open` method correctly, you should be able to compile your project and test your adaptor's analog input subsystem. Tests should take place in MATLAB, by creating an analog input object and calling the `daqhwinfo` and `propinfo` methods. For example, to test the XYZ adaptor, the code is as follows:

```
ai = analoginput('xyz');
```

```
props = propinfo(ai)
dhinfo = daqhwinfo(ai)
```

You can now check the structures returned in `props` and `dhinfo` to ensure that the correct default values and ranges are returned from a newly created object. If you find that some values in `props` are incorrect, you can correct them by modifying the `Open` method. Similarly, if the `dhinfo` structure returns incorrect information, check the `SetDaqHwInfo` method.

The next step in implementing analog input is ensuring that your adaptor responds appropriately to changes in properties.

About Native Data Types and Bits Properties

The `NativeDataType` property is important for the correct operation of the adaptor. This property is defined by the adaptor in the `SetDaqHwInfo` method, and refers to the native data type that the adaptor uses to return values to the engine or receive data from the engine. Typically, the native data type is an unsigned 16 bit integer (CComVariant type `VT_I2`), but can be other data types. The Winsound adaptor, for example, provides support for 8 bit, 16 bit, and 32 bit native data types.

Closely linked with the `NativeDataType` property is the `Bits` property, which defines the resolution of the analog input subsystem. The `Bits` property is used to indicate to the user the actual resolution of the device; the `NativeDataType` property is used by the engine to convert engineering values to native data, and the reverse. Typically, `Bits` should be less than or equal to `NativeDataType`.

Note The Data Acquisition Toolbox does not currently support floating-point native data types (floats or doubles). If your adaptor does not support reading or writing of raw data values, you must use a “fake” native data type and convert the data to floating point when you read data from a MATLAB buffer or write data to a MATLAB buffer.

Step 3.5 Implement the SetProperty and SetChannelProperty Methods

In Step 3.4 you created default values and valid ranges for the properties you defined in Step 3.1. In this step you implement code that monitors changes to properties and/or channels in the analog input subsystem.

All Data Acquisition Toolbox adaptor properties are managed by the engine. This includes any properties that are specific to your adaptor (as defined in Step 3.1). The engine understands the following types of properties:

- ***Arbitrary strings***. For example, the `Name` property can be set to any string by the user. Arbitrary strings have no specific range.
- ***Enumerated string values***. For example, the `TriggerType` property, which the engine sets to one of `Manual`, `Immediate`, or `Software`. Also known as *enum* values, these properties can take on only one of a specific number of values. Although the user sets these values using a string (or a portion of a string) the values are stored internally as an enumerated data type. An example of an enum type is the `TriggerType` property, which can be set to one of `Manual`, `Immediate`, or `Software`.
- ***Scalars***. For example, the `SampleRate` property is a scalar value. Although scalar properties can be any valid numeric data type in C++, MATLAB converts the value to a double when presenting the information to the user. Scalars typically have a range of valid values. The engine does not allow a user to set the property to values outside that range.
- ***Vectors***. For example, the `BufferingConfig` property, which is a two-element vector. The engine provides these properties to the user, but adaptors cannot create vector properties.

For most properties, your adaptor need only specify the default values, ranges, additional (or reduced) enumerated values for enum property types, and read-only status. However, some properties need to be closely monitored by the adaptor, because they can change the behavior of other properties, or need to be checked with the values of other properties. The adaptor therefore would in these circumstances need to be notified each time a user makes changes that affect such properties.

The Data Acquisition Toolbox provides this functionality by allowing an adaptor to attach to a property. The process involves attaching a *user value* on that property with the engine. The user value can be any nonzero integer. The

Adaptor Kit provides functionality that makes registering and checking a user value intuitive. For more information on attaching to properties, see "Attaching to a Property" in Chapter 4.

For all analog input properties that are attached to by an adaptor, the engine calls the `SetProperty` method of the analog input interface, passing it the user value and the new value that the user has specified. For all analog input *channel* properties that are registered by an adaptor, the engine calls the `SetChannelProperty` method of the analog input interface, passing similar information. For instance, modifying the `InputRange` of a channel calls the `SetChannelProperty`.

The `SetProperty` method can check the user value (using the `USER_VAL` macro defined by the Adaptor Kit) to determine which property has been modified. You should perform one of the following functions in the `SetProperty` or `SetChannelProperty` methods:

- Error, if the new value is invalid. The following example, from the Keithley adaptor, checks whether the hardware supports the requested `TransferMode` property, and errors out appropriately:

```
if (User == USER_VAL(pTransferMode))
{
    if((long)(*val) == TRANSFER_DMA)
    {
        SelectDriverLINX(m_driverHandle);
        if(!(DoesDeviceSupportDMA(m_pSR)))
        {
            return Error(_T("Keithley: This device does not
                support DMA Transfers."));
        }
        m_usingDMA = true;
    }
    else
    {
        m_usingDMA = false;
    }
}
```

- Change the new value if the property needs to be quantized or set to values that the hardware can implement. The following example from the

ComputerBoards adaptor calls `UpdateRateandSkew` to quantize the sample rate and change the channel skew when the user changes the sample rate:

```
else if (User==USER_VAL(pSampleRate))
{
    RETURN_HRESULT(UpdateRateAndSkew(pChannelSkewMode,*val));
    _RequestedRate=*val;
    *val=pSampleRate;
}
```

- Change the values, defaults, and/or ranges of other properties based on the new value. The following example from the ComputerBoards adaptor modifies the `TriggerCondition` property based on changes to the `TriggerType` property:

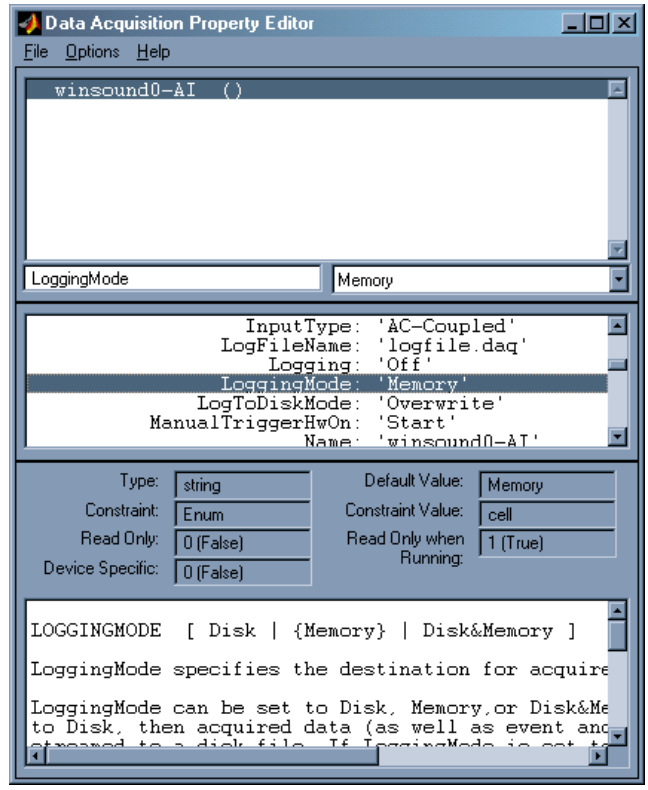
```
else if (User==USER_VAL(pTriggerType))
{
   if (static_cast<long>(*val)==HW_DIGITAL_TRIG)
   {
       pTriggerCondition->ClearEnumValues();
       pTriggerCondition->AddMappedEnumValue(GATE_HIGH,
           L"GateHigh");
        pTriggerCondition->AddMappedEnumValue(GATE_LOW,
           L"GateLow");
        pTriggerCondition->AddMappedEnumValue(TRIG_HIGH,
           L"TrigHigh");
        pTriggerCondition->AddMappedEnumValue(TRIG_LOW,
           L"TrigLow");
        pTriggerCondition=TRIG_NEG_EDGE;
        pTriggerCondition.SetDefaultValue(TRIG_NEG_EDGE);
    }
    else if (static_cast<long>(*val)==HW_ANALOG_TRIG)
    {
        pTriggerCondition->ClearEnumValues();
        pTriggerCondition->AddMappedEnumValue(GATE_NEG_HYS,
           L"GateNegHys");
        pTriggerCondition->AddMappedEnumValue(GATE_POS_HYS,
           L"GatePosHys");
        pTriggerCondition->AddMappedEnumValue(GATE_ABOVE,
           L"GateAbove");
        pTriggerCondition->AddMappedEnumValue(GATE_BELOW,
```

```
            L"GateBelow");
        pTriggerCondition->AddMappedEnumValue(TRIGABOVE,
            L"TrigAbove");
        pTriggerCondition->AddMappedEnumValue(TRIGBELOW,
            L"TrigBelow");
        pTriggerCondition=TRIGABOVE;
        pTriggerCondition.SetDefaultValue(TRIGABOVE);
        RANGE_INFO *rInfo=*_validRanges.begin();
        pTriggerConditionValue.SetRange(rInfo->minVal,
            rInfo->maxVal);
    }
}
```

Once you have completed the `SetProperty` and `SetChannelProperty` methods, you can test the changes you make to your properties, by compiling your adaptor and modifying properties of the adaptor in MATLAB. The easiest technique for modifying properties is to use the `daqpropedit` function in MATLAB, by calling

```
ai = analoginput('xyz');
daqpropedit(ai)
```

A view of the Data Acquisition Toolbox Property Editor is shown below.



For more information on `daqpropedit`, consult the *Data Acquisition Toolbox User's Guide*.

Step 3.6 Implement the ChildChange Method

Data acquisition cannot take place without a list of channels from which to sample data. The Data Acquisition Toolbox stores channels as children of the analog input and analog output objects. Typically, an adaptor should know when the user creates, removes, or reorders the channel list. If the adaptor needs to track these changes to children, you must overload the `ChildChange` method. Adaptors typically take one or more of the following actions in response to a call to `ChildChange`:

- Ensure that the hardware can handle the additional or removed channel.
- Check that the number of channels is within the limits of the hardware's channel list.
- Check whether the added channel is repeated elsewhere in the channel list; some hardware does not allow repeated channels in a channel list.
- Some hardware can only support a channel range; the user would have to add channels in strict order, and the adaptor would have to check this order.
- Update maximum sample rates based on added or removed channels.

Most adaptors should also keep track of each channel's required input range, either as a channel gain or a channel gain code. This list is updated whenever a channel is added or removed through `ChildChange`, and when the input range is changed through `SetChannelProperty`. For this reason, a separate private method is usually implemented to deal with the channel list. For an example of such a private method, see the `UpdateChans` method in the Keithley adaptor.

Understanding the ChildChange Type of Change Parameter

The `ChildChange` method is called whenever the channel list changes, regardless of the reason for that change. Often it is useful to know why the change occurred. The `ChildChange` method is passed as the first argument to the parameter `typeofchange` as a `DWORD`. Using `typeofchange`, you can determine the reason for the change as follows:

- If `(typeofchange & START_CHANGE)` is true, then the `ChildChange` method is being called before the change has taken place. You should check to ensure that the change is valid, by querying the reason for the change (see below).
- If `(typeofchange & END_CHANGE)` is true, then the `ChildChange` method is being called after the change has happened. You should requery channel properties and modify any other properties based on the modified channel list.
- If you mask `(typeofchange & CHILDCHANGE_REASON_MASK)`, you are left with the reason for the change, which can be one of
  - `ADD_CHILD`: A channel has been added.
  - `REINDEX_CHILD`: The channel list has been reordered.
  - `DELETE_CHILD`: A channel has been removed.

For examples of the use of the `typeofchange` parameter, see the Keithley adaptor's `ChildChange` method.

Testing the ChildChange Method

Once you have implemented the `ChildChange` method, you can test the addition, removal, and modification of channels in your adaptor. A typical test might look like the following:

```
>> ai = analoginput('xyz');
>> ch(1) = addchannel(ai, 0);
>> ch(2:3) = addchannel(ai, [1 2])
>> ch(4) = addchannel(ai,1)
>> set(ch(3), 'InputRange', [0 10])
>> delete(ch(2))
```

Your particular tests should include testing that the adaptor produces an error when too many channels are added, and that channel deletion is handled well.

Step 3.7 Implement the GetSingleValue Method

Up to now, the adaptor you have written does not actually read data from the hardware device. The code you have written so far provides good housekeeping, and a consistent interface to the user to ensure flexibility and access to complete hardware capabilities while preventing invalid states in the adaptor. In this step you write the first (and sometimes only) code that actually reads values from the data acquisition hardware.

The `GetSingleValue` method is called when

- The user requests data using the MATLAB `getsample` function and `GetSingleValues` is not implemented (see Step 3.8).
- The user has selected software clocking and starts an acquisition task.

Note Even if you have implemented `GetSingleValues`, you must implement `GetSingleValue` if you support software clocking for your device. The engine does not use `GetSingleValues` for software-clocked acquisitions.

The `GetSingleValue` method should sample a single value from a single channel. The `GetSingleValue` method is defined as follows:

```
HRESULT GetSingleValue(int chan, RawDataType* value)
```

The channel number is defined by the variable `chan`, and the sampled raw data value must be returned in `value`. For example, the Keithley implementation of this code is as follows (note how the adaptor returns the value directly from the device driver to `value`):

```
HRESULT Ckeithleyain::GetSingleValue(int chan, RawDataType*
value)
{
    WORD ResultCode;
    if (!m_isInitialized)
    {
        RETURN_CODE(InitializeDriverLINXSubsystem(m_pSR,
                &ResultCode));
        if (ResultCode != 0)
        {
            char tempMsg[255];
            ReturnMessageString(NULL, ResultCode,tempMsg, 255 );
            return CComCoClass<ImwDevice>::Error(tempMsg);
        }
    }
    SetupDriverLINXSingleValueIO(m_pSR, chan,
        m_chanGain[chan], SYNC);
    DriverLINX(m_pSR);
    if (m_pSR->result != 0)
    {
        return CComCoClass<ImwDevice>::Error(
            TranslateResultCode(m_pSR->result));
    }
    GetDriverLINXAIData(m_pSR, (unsigned short*)value ,0,1,1);
    return S_OK;
}
```

Testing GetSingleValue

When you have implemented this method, you can test acquisition of data by calling `getsample` with your analog input object, as follows:

```
ai = analoginput('xyz');
addchannel(ai, 0); % Set up channel 0
s = getsample(ai); % Return a single value from channel 0
addchannel(ai, 1); % Add another channel
```

```
s2 = getsample(ai); % Return data from channels 0 and 1
```

For adaptors that implement software clocking only, this is the last method you need to implement. Software-clocked adaptors should not require any additional methods to acquire data, as the software clocking methods are created in the Adaptor Kit code. For limitations on software-clocked adaptors, see "Limitations of Software-Clocked Adaptors" on page 3-11.

For adaptors that implement internal clocking, you need to follow the remaining steps of Stage 3.

Step 3.8 Implement the GetSingleValues Method

The `GetSingleValues` method should only be implemented if the device driver supports the acquisition of a single immediate sample from multiple channels in one driver call. For drivers that only allow single immediate samples from one channel, the `GetSingleValue` method is appropriate, as the engine then takes care of looping through all channels in the channel list.

The `GetSingleValues` method should simply implement code that acquires a single immediate sample from all the channels defined in the channel list. The result should be returned in a SafeArray vector, and should contain raw data in the data type defined by the adaptor for that hardware.

A typical example of a `GetSingleValues` implementation is as follows:

```
HRESULT CAin::GetSingleValues(VARIANT * Values)
{
    UpdateChans(true); // Update channel lists.

    TSafeArrayVector <unsigned short > binarray;
    binarray.Allocate(_nChannels);
    // Code to call hardware here.
    // Return result in binarray
    return binarray.Detach(Values);
}
```

When you have written the `GetSingleValues` method, you should retest the adaptor using the `getsamples` function from MATLAB. See Step 3.7 for sample MATLAB code.

Step 3.9 Implement the Start, Trigger, and Stop Methods

The final step in implementing the analog input subsystem is to provide functionality for hardware-clocked or hardware-triggered acquisition tasks. This final step is by far the most complex step in the implementation of the analog input subsystem, and should not be implemented unless software clocking is insufficient for your required task. If you are intending to implement this step, it is assumed that you can configure the device driver to continuously acquire data from a list of channels, clocking the device using the hardware's onboard clock. If this is not possible, you will have to resort to software clocking and live with the limitations of that implementation.

Hardware-clocked adaptors require a number of technologies to be implemented correctly in order to handle the data transfer between the engine and the hardware device. To successfully implement hardware clocking, your adaptor must

- Set up the acquisition task correctly so that continuous data acquisition can take place at the desired rate, using the desired channels and triggering
- Handle buffering of data between the engine and the hardware device, so that the hardware can provide the data to the engine without missing samples or overrunning the hardware buffer on the device
- Handle messaging between the engine and the hardware device, so that the hardware device stops when the engine has sufficient data or when the user stops the acquisition, and so that the engine can keep track of the number of samples acquired since starting the task
- Implement and/or manage multithreading in the adaptor to allow the device to transfer data to the engine without interfering with MATLAB processing, and to prevent MATLAB from blocking messages from the device driver

Support for hardware clocking requires, at a minimum, implementation of the `Start`, `Trigger`, and `Stop` methods. These three methods are used by the engine for managing a data acquisition task. Adaptors also typically write methods to handle event notification or polling of the running task, and data transfer methods to synchronize exchange of data between the engine and the device driver.

Implementing the Start Method

The `Start` method is called by the engine in response to a user's issuing a `start` command in MATLAB. The `Start` method should configure the acquisition task and initialize any buffers being used for acquisition. Typical actions in the `Start` method include

- Initializing the hardware device subsystem (if this can be done without interfering with other subsystems, such as the analog output subsystem)
- Choosing buffer sizes based on the `BufferingConfig` and `SampleRate` properties of the analog input object, and hardware and device driver buffering characteristics
- Performing a final check on all properties to ensure that the acquisition can be run as requested
- Configuring the device driver with the correct channel list, gains, and triggers

The `Start` method should not start the analog input acquisition task. The `Trigger` method is responsible for starting the task, setting the running flag, and initializing any device driver event notification mechanisms.

Note For more information on buffering, particularly on how the engine handles buffers, see Chapter 5, "Buffering Techniques."

The following example code shows how the Keithley adaptor implements the `Start` method:

```
HRESULT Ckeithleyain::Start()
{
    if (pClockSource == CLCK_SOFTWARE)
        return InputType::Start();
    WORD ResultCode;
    CComBSTR _error;
    m_samplesThisRun=0;
    m_triggersProcessed=0;
    m_triggerPosted = false;

    // First Check if the device is in use.
    if(GetParent()->IsOpen((analoginputMask + m_deviceID)))
```

```
            return CComCoClass<ImwDevice>::Error(
                    CComBSTR("Keithley: The device is in use."));

        // Check if there is an active Service request waiting to stop?
        if (m_daqStatus!=STATUS_STOPPED)
        {
            return CComCoClass<ImwDevice>::Error(
                    CComBSTR("Keithley: Attempt to start a device
                    before previous task stopped."));
        }
        // Initialize the subsystem.
        SELECTMYDRIVERLINX(m_driverHandle);
        RETURN_CODE(InitializeDriverLINXSubsystem(m_pSR,
                &ResultCode));
        if (ResultCode != 0)
        {
            char tempMsg[255];
            ReturnMessageString(NULL, ResultCode,tempMsg, 255 );
            return CComCoClass<ImwDevice>::Error(tempMsg);
        }
        RETURN_HRESULT(SetupSRForStart( true ));

        // Audit the Service Request to check for errors.
        Ops ops = m_pSR->operation;
        m_pSR->operation = AUDITONLY;
        SELECTMYDRIVERLINX(m_driverHandle);
        DriverLINX(m_pSR);
        if (m_pSR->result != 0)
        {
            // Now remove the device from the message window:
            GetParent()->DeleteDev((analoginputMask + m_deviceID));
            char tempMsg[255];
            ReturnMessageString(NULL, m_pSR->result,tempMsg, 255 );
            return CComCoClass<ImwDevice>::Error(tempMsg);
        }
        m_pSR->operation = ops;
        return S_OK;
    }
```

Implementing the Trigger Method

The `Trigger` method is responsible for actually initiating the acquisition task. The `Trigger` method is called immediately after the Start method unless `TriggerType` is set to `Manual` and `ManualHwTriggerOn` is set to `Trigger`. See "How Start and Trigger Work Together" on page 3-42 for more information on these properties.

The `Trigger` method should simply start the acquisition task and set a running flag. Additional operations to perform in the `Trigger` method include

- Ensuring that the message handler has a link to the analog input object that initiates the task (see the Keithley adaptor implementation for an example)
- Trapping a failed task and calling the `Stop` method to inform MATLAB that the device has stopped, and removing this device from the running list of the adaptor (see the Keithley implementation of the `Stop` method for an example)

The `Trigger` method should not necessarily handle hardware triggers. Hardware triggers should be configured during the `Start` method and processed by the event handler for your adaptor.

A typical implementation of the `Trigger` method is from the Keithley adaptor:

```
HRESULT Ckeithleyain::Trigger()
{
    if(pClockSource == CLCK_SOFTWARE)
        return InputType::Trigger();
     AUTO_LOCK;

    // Register the device ID with the message window
    if (GetParent()->AddDev((analoginputMask + m_deviceID),this))
    {
        return Error(_T("Keithley: The device is in use."));
    }

    //Start DriverLINX
    SELECTMYDRIVERLINX(m_driverHandle);
    DriverLINX(m_pSR);
    if (m_pSR->result != 0)// If the Service request fails
    {
        // Remove the device from the message window:
```

```
            GetParent()->DeleteDev((analoginputMask + m_deviceID));
            return CComCoClass<ImwDevice>::Error(
                TranslateResultCode(m_pSR->result));
        }
        m_daqStatus = STATUS_RUNNING;
         return S_OK;
    }
```

How Start and Trigger Work Together

The `Start` and `Trigger` methods enable the user complete flexibility in how the acquisition task is launched. A variety of data acquisition analog input object properties influence how these two methods are called:

- When the user issues a `start` command in MATLAB, the engine calls the `Start` method in the adaptor.
- If the `TriggerType` is set to `Immediate` or `Software`, the `Trigger` method is called immediately after the `Start` method.
- If the `TriggerType` is set to `Manual`, the engine checks the `ManualHwTriggerOn` property. If this property is set to `Start`, the `Trigger` method is called immediately after the `Start` method. If `ManualHwTriggerOn` is set to `Trigger`, the `Trigger` method is only called when the user issues the `trigger` command in MATLAB.

  When the `Trigger` method is called, MATLAB starts acquiring data into internal buffers. If `TriggerType` is `Immediate`, this data is logged immediately. If `TriggerType` is `Software` or `Manual`, the data is logged once a software or manual trigger is issued. The engine uses the acquired data to check for software triggers and to provide data to the user, using `peekdata`.

You do not need to handle `ManualHwTriggerOn` in any special way in order for these operations to take place: The Data Acquisition Toolbox engine provides all this functionality transparently to the user or the adaptor developer. However, you do need to understand how the methods are called for debugging purposes.

Implementing the Stop Method

The `Stop` method is responsible for

- Stopping the acquisition task

- Removing the device from the list of running analog input subsystems
- Sending a stop event notification to the engine, which resets the object's `Running` property to off and allows the user to configure properties that are read only when running

The `Stop` method is called in response to the user's stopping a running task by calling `stop`, and should also be called by the adaptor code:

- When the engine returns a buffer with the `Buffer_Is_Last` flag set (see "Understanding Engine Buffers" in Chapter 5). This check is usually performed in the device driver callback routines.
- When the adaptor's event handlers detect that the device has stopped acquiring data (calling `Stop` and sending a stop event notification to the engine allows the engine to configure the analog input object appropriately).
- From the `Start` method, if the device cannot be configured properly for the requested task.

An example of the `Stop` method implementation is from the Keithley adaptor:

```
HRESULT Ckeithleyain::Stop()
{
    if (pClockSource == CLCK_SOFTWARE)
        return InputType::Stop();
    AUTO_LOCK;

    if (m_daqStatus == STATUS_RUNNING)
    {
        if (pStopTriggerType==STP_TRIGT_NONE)
        {
            WORD ResultCode = StopDriverLINXSR(m_pSR);
            if(ResultCode !=0)
            {
                _engine->WarningMessage(
                    TranslateResultCode(ResultCode));
            }
        }

        // Now remove the device from the message window:
        GetParent()->DeleteDev((analoginputMask + m_deviceID));
        m_daqStatus = STATUS_STOPPED;
```

```
            _engine->DaqEvent(EVENT_STOP, -1, m_samplesThisRun,
                NULL);
        }
        return S_OK;
    }
```

Note in the example how a failed hardware stop is provided to the user as a warning message, and not an error. This ensures that the `Stop` method terminates successfully and allows the user to restart an acquisition task.

Keeping Track of Samples Acquired

During an acquisition task, your adaptor should keep track of the number of samples that have been acquired by the hardware device. This information is used by the engine to report on events (events can be sample-based or time-based; sample-based timing is common in events) and to provide status information to the user during an acquisition task. The engine also uses the value to run various callback functions.

The engine keeps track of the number of samples processed by checking the `StartPoint` field of the buffers returned by your adaptor. You should also use the number of samples processed when posting other events, such as Data Missed or Stop events.

Managing and Posting Events

Your adaptor is responsible for posting the following events:

- The Stop event (`EVENT_STOP`), posted when your adaptor receives a `Stop` message (i.e., in your `Stop` method). This event is used by the engine to reset the running status of an analog input object.
- Data Missed events (`EVENT_DATAMISSED`), which should be posted if you detect any missed data events from your hardware device. If the device driver does not support these events, you can check for them in your adaptor.
- Error events (`EVENT_ERR`), which should be posted if the device driver issues an error during a task.
- Overrange events (`EVENT_OVERRANGE`), which should be posted if your hardware is capable of registering when signals are over the input range defined for a channel.
- Trigger events (`EVENT_TRIGGER`), which should only be posted by your adaptor when hardware triggers are being used.

To post an event, use the engine's `DaqEvent` method, defined as follows:

```
HRESULT DaqEvent(DWORD event, double time, __int64  sample, BSTR
Message)
```

You should always pass as much accurate information as you can with your type of event (`event`) but should at least post the event time (using `time`) or the sample at which the event occurred (in `sample`). If you pass a time value of -1, the engine calculates the time of the event based on the sample number. For error events, you can pass the error string in the last parameter (`Message`).

For a complete discussion of the `DaqEvent` method, see "DaqEvent" in Appendix B.

Implementing Callbacks and Threading

If you are using hardware clocking, you will need to implement callbacks from your device driver to inform the engine of the progress of the acquisition, and to pass data back to the engine as necessary. The level of support that your device driver provides for callbacks defines how you implement this feature in your adaptor. For a complete discussion of callbacks and threading, see Chapter 6, "Callbacks and Threading."

Returning Errors from Your Adaptor

There are two ways to return a meaningful error from an adaptor.

The first method is to return an error code in an `HRESULT` from a function call. The error code should be a valid `HRESULT` with a facility of `FACILITY_ITF` and a code greater than `PRIVATE_BASE` defined in `daqmexstruct.h`. One way to make such an error code is to use the macro `MAKE_PRIVATE_ERROR(x)` defined in the same file. If you add a descriptive string for the error to the adaptor's resource file and use the implementation of `ImwAdaptor::TranslateError` from the demo adaptor, the string is returned as an error message in MATLAB.

A second method to return a descriptive error message is to have your adaptor support `ISupportErrorInfo`, and return an `IErrorInfo` object using standard COM methods. See the ATL function `CComCoClass::Error` and its use by the `cbi` adaptor for an example of how to return this type of error.

Stage 4 Implement the Analog Output Subsystem

Implementation of the analog output subsystem follows the same pattern as implementation of the analog input subsystem. Much of the discussion of how to perform the implementation is covered in “Implement the Analog Input Subsystem” on page 3-18 and is not covered in this section. If you are not implementing analog input, you will have to refer to that chapter where appropriate.

Implementation of the analog output subsystem takes place in the following steps (compare these to the analog input subsystem steps for a comparison of the implementation):

1. Select the default values, ranges, and other characteristics of the analog output subsystem properties.
2. Create the analog output COM interface and class definitions in the `IDL` file, and incorporate the demo adaptor analog output implementation in your project.
3. Modify the `OpenDevice` method of the adaptor class to create the required subsystem when requested.
4. Modify the `Open` and `SetDaqHwInfo` methods of the Analog Output class to handle device initialization, create custom properties, and set defaults and ranges for all properties.
5. Implement the `SetProperty` and `SetChannelProperty` methods of the Analog Output class, to handle property changes.
6. If necessary, overload the `ChildChange` method of the Analog Output class to handle channel addition and removal.
7. Implement the `PutSingleValue` method if software clocking is to be used.
8. Implement the `PutSingleValues` method if the device driver supports easy single acquisition from multiple channels.
9. Implement the `Start`, `Trigger`, and `Stop` methods for buffered acquisition. Typically, this step involves writing buffering routines and message

handlers, and can often use the multithreading built during implementation of the analog input subsystem.

As the steps outlined above are no different from those implementing the analog input subsystem, only the differences in implementation are discussed directly in this stage.

Note You should use the answers to the questions posed in Stage 1 to decide which of the preceding steps you will implement in your adaptor.

Step 4.1 Select Property Values, Ranges, and Defaults for Analog Output

In order to control the behavior of a task (such as duration and volume of acquisition, type of triggering, clocking, and event callbacks) the MATLAB user modifies the properties of the Data Acquisition Toolbox `analogoutput` object representing the data acquisition hardware he/she is using. The adaptor must use the property values during acquisition tasks to control driver settings, return messages, and start and stop acquisition. The adaptor must also provide the data acquisition engine with appropriate properties, ranges, and default values for the specific hardware referenced by the adaptor. Successfully creating an adaptor therefore requires careful thought about the existing common analog output subsystem properties, and the addition of adaptor-specific properties where appropriate.

For the analog output subsystem, you should construct the same `propinfo` table you compiled for the analog input subsystem. Properties that should be considered in the table are given in Appendix D, “Sample Property and daqhwinfo Tables.”

The `propinfo` table should reflect the state of the desired output from a call to the `propinfo` method of the `analoginput` object. Thus, when you run

```
propinfo(analogoutput(<adaptor>))
```

the result should be the data presented in the `propinfo` table. The output of the MATLAB code given above provides a test to confirm that these properties have been created and initialized successfully.

The outcome of Step 4.1 is a document that forms the blueprint for the implementation of properties in later steps of this stage.

Step 4.2 Add the Demo Analog Output Code to Your Project

This step is exactly the same as for the analog input subsystem, except that you import the analog output code (`demoaout.cpp` and `demoaout.h`) and change the input to output as appropriate.

Step 4.3 Modify the OpenDevice Method of the Adaptor Class

In this step of the implementation of the analog output object, you ensure that the renamed demo adaptor analog output object works in MATLAB. This stage is the same as for the analog input subsystem.

- Uncomment the analog output construction statements in the `OpenDevice` method of the Adaptor class. Be sure to include the Analog Output class header file in the adaptor class implementation file.
- Compile the project, and you should be able to launch MATLAB and create an analog output object for your adaptor:

```
ai = analogoutput('xyz');
```

If the compilation fails, you should ensure that all header files have been created, and that your analog output class name is consistent throughout the project. Also ensure that all the steps from the previous stage have been implemented as well.

Your adaptor should now contain a complete analog output object, which does not output any data to any device. In future stages of the analog output implementation, you will progressively implement hardware output tasks. The first of these stages is to ensure that the subsystem's properties are initialized correctly, possibly from hardware device detection.

Step 4.4 Modify the Analog Output Open and SetDaqHwInfo Methods

This step is identical for analog input and for analog output. The only differences are in the properties that must be implemented in the analog output subsystem and `SetDaqHwInfo` method. For a discussion of the important

properties for both of these methods, consult Appendix D, "Sample Property and daqhwinfo Tables."

Step 4.5 Implement the SetProperty and SetChannelProperty Methods

This step is identical to Step 3.5 for analog input `SetProperty` and `SetChannelProperty` methods.

Step 4.6 Implement the ChildChange Method

This step is identical to Step 3.6 for analog input `ChildChange` method.

Step 4.7 Implement the PutSingleValue Method

The `PutSingleValue` method is almost identical to the `GetSingleValue` method and should be implemented in a similar fashion. The following points should be noted:

- MATLAB passes native data to the adaptor (as defined by the `NativeDataType` property and the `NativeScaling` and `NativeOffset` engine properties for your adaptor; see "About Native Data Types and Bits Properties" on page 3-28). Hence your `PutSingleValue` method must be able to pass native data to the hardware device (native data is sometimes referred to by driver vendors as *raw* data).
- `PutSingleValue` must transfer a single value to a single channel. The `PutSingleValue` method is defined as follows:

  ```
  HRESULT PutSingleValue(int chan, RawDataType value)
  ```

The channel number is defined by the variable `chan`, and the raw data value to be output is passed in `value`.

Note Even if you have implemented `PutSingleValues`, you must implement `PutSingleValue` if you support software clocking for your device. The engine does not use `PutSingleValues` for software-clocked acquisitions.

For example, the Keithley implementation of this code is as follows:

```
HRESULT Ckeithleyaout::PutSingleValue(int chan,
```

```
        RawDataType value)
{
    SELECTMYDRIVERLINX(m_driverHandle);
    SetupDriverLINXSingleValueIO(m_pSR,chan,
        m_chanGain[0], SYNC);
    PutDriverLINXAOData(m_pSR, (unsigned short*)&value , 0, 1, 1);
    SELECTMYDRIVERLINX(m_driverHandle);
    DriverLINX(m_pSR);
    if (m_pSR->result != 0)
    {
        return CComCoClass<ImwDevice>::Error(
            TranslateResultCode(m_pSR->result));
    }
    return S_OK;
}
```

Testing PutSingleValue

When you have implemented this method, you can test data output by calling `putsample` with your analog input object, as follows:

```
ao = analogoutput('xyz');
addchannel(ao, 0); % Set up channel 0
putsample(ao, 2.5); % Send 2.5V to channel 0
addchannel(ao, 1); % Add another channel (if possible)
putsample(ao, [1 2.5]); % Send data to channels 0 and 1
```

For adaptors that implement software clocking only, this is the last method you need to implement. Software-clocked adaptors should not require any additional methods to acquire data, as the software clocking methods have been created in the Adaptor Kit code. For limitations on software-clocked adaptors, see "Limitations of Software-Clocked Adaptors" on page 3-11.

For adaptors that implement internal clocking, you need to follow the remaining steps of Stage 4.

Step 4.8 Implement the PutSingleValues Method

The `PutSingleValues` method should only be implemented if the device driver supports the output of a single immediate sample to multiple channels in one driver call. For drivers that only allow single immediate output to one channel,

the PutSingleValue method is appropriate, as the engine then takes care of looping through all channels in the channel list.

The PutSingleValues method is passed a SafeArray of raw data values that must be output to the channels defined in the analog output channel list. If you implemented the ChildChange method correctly, you already have the channel numbers and gains set up in local storage (or you can retrieve them from the engine).

For a typical example of a PutSingleValues implementation, consult the Keithley adaptor's analog output subsystem.

When you have written the PutSingleValues method, you should retest the adaptor using the putsample function from MATLAB. See Step 4.7 for sample MATLAB code.

Step 4.9 Implement the Start, Trigger, and Stop Methods

The final step in implementing the analog output subsystem is to provide functionality for hardware-clocked or hardware-triggered acquisition tasks. This step follows almost identically the implementation for the analog input subsystem, and so is not discussed in this stage in as much detail. However, the following sections note some differences in approach and implementation for the analog output subsystem that are particular to that system.

Basic Approach of Hardware-Clocked Analog Output

The basic approach of the analog output subsystem is as follows:

- In the Start method, initialize the device and configure the output task channels, gains, and clock frequency, plus any hardware triggers you might be supporting. If you are using adaptor buffering (see Chapter 5, "Buffering Techniques," for more information), allocate the adaptor buffers. Following initialization, you should prime the internal (or driver) buffers with as much data as possible from MATLAB. This process should occur in the Start method.
- In the Trigger method, set up the event handlers and start the acquisition task. Also, set the property _running to true (see below).
- In the event handler routines, refill the adaptor or driver buffers until MATLAB no longer provides any data (when the buffer pointer is null, or the

BUFFER_IS_LAST flag has been set in a MATLAB buffer), then continue queuing any correct data to the driver and/or wait until the hardware has output all relevant data before calling the Stop method. The simplest way to continue queuing data is to write OutOfDataMode values to the buffers until you know that the last valid output data has been sent.

- During the task, you should update the _samplesOutput property (see below) to reflect how many values have been sent to the hardware DAC.

About the _samplesOutput and _running Properties

All adaptors that inherit the TDADevice class (which your adaptor must inherit) include the two properties _samplesOutput and _running. The _samplesOutput property (a 64 bit integer) should reflect as closely as practically possible the number of samples sent to the hardware DAC at any time. The _running property, a Boolean, should be set to true when the analog output task is running, and false when the task is stopped. Typically, the Trigger method sets _running to true and the Stop method sets _running to false.

These two properties are used by the TDADevice class's implementation of the GetStatus method, which is used by the engine to query the status of a running task.

Understanding OutOfDataMode

After the output hardware has sent all the data requested by the user, most hardware systems hold the last value output on each until a new value is set for that channel. However, some applications require that the channels be placed in some default mode of operation (e.g., output of zero) to protect hardware and/or systems connected to that channel. MATLAB supports the use of *default channel values* for each channel of an analog output subsystem. The user can control whether the last output value for each channel is held or whether the channel should be set to the default value by changing the OutOfDataMode property. The valid values are DefaultValue and Hold, with Hold being the default.

Your adaptor should handle the OutOfDataMode settings appropriately. For example, the Keithley adaptor handles the OutOfDataMode settings by overwriting the internal default channel values with the last output values when OutOfDataMode is set to Hold. When the engine no longer passes buffers to the adaptor, the data in the internal default channel values is repeatedly

output to the driver, effectively filling the hardware FIFO with those values (see below). The acquisition is then stopped after at least one of those samples is sent to the output channels.

Dealing with the Output Hardware FIFO Buffer

Many hardware device drivers only report when data has been sent to the hardware FIFO, and not necessarily how much data the device has sent to the DAC. You may need to monitor and test the output sequences carefully to ensure that all the required data is output to the DAC and not just to the FIFO. An example of such a driver is the Keithley Instruments DriverLINX driver, which provides feedback on a task only when a driver buffer has been emptied (and not when the hardware FIFO buffer has sent the data). To handle this, the Keithley adaptor queues at least a FIFO buffer of default values after the last buffer has been received from MATLAB. The task then waits for the FIFO buffer to be filled with these values before stopping the task. In this way, the FIFO is emptied of real data, and the last value output is always the default value for each channel.

Stage 5 Implement the Digital I/O Subsystem

The digital I/O (DIO) subsystem implemented in MATLAB is not as full-featured as the analog input and analog output subsystems and hence requires far less implementation. The most notable difference is that the DIO subsystem does not support continuous or timed reading or writing of digital data through DIO lines. Instead, the adaptor provides techniques for reading or writing only a single value at a time to the DIO subsystem. Hence, most adaptors implement only five methods for DIO subsystems. This stage discusses these five methods.

Implementation of the digital I/O subsystem takes place in the following steps:

1. Select the default values, ranges, and other characteristics of the DIO subsystem properties.
2. Create the DIO COM interface and class definitions in the `IDL` file, and incorporate an adaptor's DIO implementation in your project.
3. Modify the `OpenDevice` method of the adaptor class to create the DIO subsystem when requested.
4. Modify the `Open` and `SetDaqHwInfo` methods of the DIO class to handle device initialization, create custom properties, and set defaults and ranges for all properties.
5. Implement the `SetPortDirection` method of the DIO class to handle port and/or line direction changes.
6. Implement the `ReadValues` method to read data from digital lines.
7. Implement the `WriteValues` method to write bits to digital lines.

Each of these steps is discussed in detail in the following sections.

Note You should use the answers to the questions posed in Stage 1 to decide which of the preceding steps you will implement in your adaptor.

Step 5.1 Select Property Values, Ranges, and Defaults for Digital I/O

Because the Digital I/O (DIO) subsystem does not use continuous acquisition tasks, the DIO object in the Data Acquisition Toolbox contains far fewer properties than the analog input and analog output objects do. However, planning the behavior of the existing properties for the DIO subsystem is just as important as for the other two objects.

For the DIO subsystem, you should construct a `propinfo` table similar to the one you compiled for the analog input subsystem. Properties that should be considered in the DIO table are given in Appendix D, "Sample Property and daqhwinfo Tables."

The `propinfo` table should reflect the state of the desired output from a call to the `propinfo` method of the `digitalio` object. Thus, when you run

```
propinfo(digitalio(<adaptor>))
```

the result should be the data presented in the `propinfo` table. The output of the MATLAB code given above provides a test to confirm that these properties have been created and initialized successfully.

The outcome of Step 5.1 is a document that forms the blueprint for the implementation of properties in later steps of this stage.

Step 5.2 Add the Digital I/O Code from an Adaptor to Your Project

Because the DIO subsystem is so small relative to the other objects, digital I/O is not included in the demo adaptor. The easiest way to implement the DIO subsystem is to use an existing class from another adaptor. We recommend the Keithley adaptor as an example of implementation of digital I/O.

For the Keithley adaptor, the following methods are specific to the Keithley implementation, and should be removed from your adaptor code.

Method	Description (Reason for Removing)
`GetParent`	Used to control the message window for Keithley acquisition tasks. You should not implement this method in your digital I/O subsystem.
`IsChanClockOrTrig`	Excludes all ports that are reported by the Keithley DriverLINX driver as being digital I/O ports when they are in fact external clock or trigger lines. Not required for your implementation.
`SetParent`	Used to control the message window for Keithley acquisition tasks. You should not implement this method in your digital I/O subsystem.

To include the Keithley DIO files in your project, perform the following:

- Copy the `keithleydio.cpp` and `keithleydio.h` files from the Keithley adaptor to your adaptor project directory, and rename them for your adaptor (for example, `xyzdio.cpp` and `xyzdio.h`).
- Search for all instances of "`keithley`" in these files and replace them with your adaptor name. Be sure to do this for both the `.cpp` and `.h` files.
- Add those files to your project.

Step 5.3 Modify the OpenDevice Method of the Adaptor Class

In this step of the implementation of the digital I/O object, you ensure that the adaptor can create a digital I/O object. This stage is the same as for the analog input subsystem.

Uncomment the digital I/O construction statements in the `OpenDevice` method of the Adaptor class. Be sure to include the digital I/O class header file in the adaptor class implementation file.

Note You are unable to test the adaptor at this stage, as you need to modify the methods written for the Keithley adaptor.

Step 5.4 Modify the DigitalIO Open and SetDaqHwInfo Methods

This step is identical to the previous subsystem implementations. The `Open` method typically just checks the device ID and initializes the subsystem, then calls the `SetDaqHwInfo` method. For a discussion of the important properties for `SetDaqHwInfo`, consult Appendix D, "Sample Property and daqhwinfo Tables."

Step 5.5 Modify the SetPortDirection Method

The `SetPortDirection` method is used to set up the read or write status of each line in each port. The calling syntax of the method is as follows:

```
HRESULT ::SetPortDirection(LONG Port, ULONG DirectionValues)
```

The first argument is the port number, and the second argument is a masked list of directions for that particular port. A 0 in any bit position means that the particular line should be an input (reading), while a 1 in any bit position means that the line must be configured for output (writing).

Note For port-configurable devices, the `DirectionValues` variable is always set to 0 for input, or 255 for output, regardless of the size of the port.

A typical example of the `SetPortDirection` method is given below for the ComputerBoards adaptor:

```
HRESULT CDio::SetPortDirection(LONG Port, ULONG DirectionValues)
{
   if (PortNum[Port]==AUXPORT)
       return S_OK;
   if (DirectionValues==0)
   {
       CBI_CHECK(cbDConfigPort(_BoardNum,
           PortNum[Port],DIGITALIN));
   }
   else
   {
       CBI_CHECK(cbDConfigPort(_BoardNum,
           PortNum[Port],DIGITALOUT));
```

```
    }
    return S_OK;
}
```

Step 5.6 Implement the ReadValues Method

The `ReadValues` method is used to read data from a number of digital lines. The `ReadValues` method is called whenever the user requests digital input using the `getvalue` function in MATLAB (see "Reading Line Values" in Chapter 2 for more information on the `getvalue` function).

A typical implementation of this method is shown for the ComputerBoards adaptor:

```
HRESULT CDio::ReadValues(LONG NumberOfPorts, LONG * PortList,
       ULONG * Data)
{
    if (Data == NULL)
        return E_POINTER;
    USHORT val;
    for (int i=0;i<NumberOfPorts;i++)
    {
        CBI_CHECK(cbDIn(_BoardNum,PortNum[PortList[i]],&val));
        Data[i]=val;
    }
    return S_OK;
}
```

The data is always returned to MATLAB as unsigned long data.

Note Some boards allow lines on a given port to be configured separately. In this case, `ReadValues` still assumes that the whole port will be read. However, the values obtained from the lines configured for output are meaningless, because they reflect values latched from a previous write operation. The engine returns only the requested line information to the user.

Testing the ReadValues Method

Once you have written the `ReadValues` method, you can test your device. The code given in "Reading Line Values" in Chapter 2 provides example code to test your adaptor.

Step 5.7 Implement the WriteValues Method

The `WriteValues` method is used to write data to a number of digital lines. The `WriteValues` method is called whenever the user sends digital input data using the `putvalue` function in MATLAB (see "Writing Line Values" in Chapter 2 for more information on the `putvalue` function).

A typical implementation of this method is shown for the ComputerBoards adaptor:

```
HRESULT CDio::WriteValues(LONG NumberOfPorts,
        LONG * PortList, ULONG * Data, ULONG * Mask)
{
    for (int i=0;i<NumberOfPorts;i++)
    {
      CBI_CHECK(cbDOut(_BoardNum,PortNum[PortList[i]],Data[i]));
    }
    return S_OK;
}
```

Testing the WriteValues Method

Once you have written the `WriteValues` method, you can test your device. The code given in "Writing Line Values" in Chapter 2 provides example code to test your adaptor.

4

Working with Properties

Overview

This chapter contains information on dealing with Data Acquisition Toolbox properties in your adaptor. This information provides you with techniques for using the Adaptor Kit templates for

- Accessing properties from your adaptor
- Attaching to properties to monitor property changes (or for frequent interaction with the property)
- Creating adaptor-specific properties

Once you have created a reference to the property (using one of the preceding techniques), this chapter also provides you with information on

- Setting values, defaults, and ranges
- Adding and removing items from enumerated lists

The Data Acquisition Toolbox uses properties of data acquisition objects to control how the object behaves in response to user requests. Any action taken by a data acquisition object should be due to specific property values, or combinations of values of many properties.

MATLAB users interact with properties by using the `get` and `set` methods on the data acquisition object. However, users cannot set properties to any arbitrary value. Properties have defined types, and can have defined ranges or enumerated lists from which the user can select a value.

Adaptors can access properties through their `IPropRoot` interfaces. Property methods are generally called from within

- The adaptor's `Open` method, when creating an adaptor-specific property
- The adaptor's `Open` method, when changing the characteristics of an existing property
- The adaptor's `SetProperty` method, when the user modifies a property that affects the characteristics of other properties or requires special attention because of hardware limitations (for example, quantization of sampling rates)
- The adaptor's `SetChannelProperty` method, when modifying one channel property affects the characteristics of another property

- The adaptor's `Start` method, when accessing values of a property that are needed to set up the acquisition task

---

Note This chapter should be used as a reference for techniques, and not as a reference for the Adaptor Kit ATL code. The Adaptor Kit ATL code does not need to be understood in order for you to use the ATL templates.

Accessing Properties from Your Adaptor

The data acquisition engine provides property management functions, exposing a number of methods to your adaptor through the `IPropRoot`, `IPropValue`, and `IPropContainer` interfaces. Although you can use these interface methods directly, the Adaptor Kit includes some helper functions to allow you to interact with properties more easily.

The first requirement for interacting with a property is to be able to access that specific property. Some properties require frequent interaction from the adaptor, and those should be accessed using the `ATTACH_PROP` macro. Properties that require less frequent interaction should be accessed through the `GetProperty` method, which your adaptor inherits from `CmwDevice`.

Note Even if you do not need to monitor changes in a property, if you frequently need the value of that property in your adaptor, you should attach to that property.

For information on attaching to properties, see “Attaching to a Property” on page 4-5.

Accessing a Property Using GetProperty

You can obtain a pointer to a property by using the `GetProperty` method provided in the `CmwDevice` class. Because your adaptor inherits from this class, you can use this method anywhere in your adaptor.

The `GetProperty` method takes two arguments: the property name (a wide character array) and a pointer to an `IProp` interface. The following code returns the `ClockSource` property in variable `propCS`:

```
CComPtr<IProp> propCS;
GetProperty(L"ClockSource", &propCS);
```

You can now access the properties through `propCS`. When you finish using the property, you should call the `Release` method:

```
propCS.Release();
```

Attaching to a Property

You should attach to properties when

- Your adaptor needs to monitor changes in that property.
- Your adaptor frequently needs to query the property (usually to obtain the current value).

The `ATTACH_PROP` macro sets up a local pointer to a particular property, which can be accessed throughout the adaptor. `ATTACH_PROP` is the preferred method of attaching to properties. The ComputerBoards and Keithley adaptors use the `ATTACH_PROP` macro.

Note The Winsound and Nidaq adaptors do not implement the `ATTACH_PROP` macros. Instead, they use lower-level methods from the `IPropRoot` and `IProp` interfaces. For code examples of the `ATTACH_PROP` macro, consult the Keithley or ComputerBoards adaptor code.

The `ATTACH_PROP` macro works together with the property templates from the Adaptor Kit. To use the `ATTACH_PROP` macro, you need to define a local (typically private) variable using the property templates from the Adaptor Kit. The variable name must begin with a “p”, and should then have a descriptive name after the “p”. For example, the local variable for the `ClockSource` property would be named `pClockSource`.

Table 4-1 lists, with examples of their use, the property templates defined in the Adaptor Kit. Of these templates, the most commonly used are the `CLocalProp` (for obtaining properties that are seldom used) and `TRemoteProp` class (for obtaining properties that change default values and/or ranges, or attaching to properties).

Table 4-1: Adaptor Kit Defined Property Classes and Templates

Type	Description and Typical Usage	Example
`CLocalProp`	Virtual base class. All other property classes derive from this class. Use this class when obtaining an `IPropRoot` interface for seldom-accessed properties	All `SetProperty` methods
`CRemoteProp`	Class with added `SetRange` and `SetDefaultValue` methods. Used by template classes.	None. Use Template classes.
`TProp`	Base template for properties that should hide the `IPropRoot` interface.	See Table 4-2.
`CEnumProp`	Enumerated property class. Use this class to check an enumerated property value. Use `CachedEnumProp` for modifying enum values.	`pOutOfDataMode` in Keithley analog output
`TRemoteProp`	Template class for defining remote properties. Use this class if you need to change default values and set new ranges for a property.	`pSamplesPerTrigger` in Keithley analog input
`TCachedProp`	Template for cached properties. Use instead of `TRemoteProp` if a property value should be tested locally prior to updating in engine. Then use `SetLocal` to set local values and `SetRemote` to set remote values (= operator sets both local and remote values).	`_chanSkew` property in Nidaq analog input
`TArrayProp`	Template for array properties. Although adaptors cannot create array properties, the engine defines some array properties that might be required in the adaptor. See derived types in Table 4-2.	See Table 4-2.

The Adaptor Kit defines some data types that are derived from these classes. These data types should be used whenever possible, as they ensure clarity and consistency in adaptor data types. The derived data types are given in Table 4-2, including examples of their use in existing adaptors.

Table 4-2: Derived Data Types for Properties

Type	Description	Example
`IntProp`	Integer property. Cannot set default value or ranges.	`pInputType` in ComputerBoards adaptor
`ShortProp`	Short property. Cannot set default value or ranges.	None
`DoubleProp`	Double property. Cannot set default value or ranges.	`_triggerDelay` in Nidaq analog input
`BoolProp`	Boolean property. Cannot set default value or ranges.	`_driveAIS` in Nidaq analog input
`Int64Prop`	64 bit Integer property. Cannot set default value or ranges.	`pTriggerRepeat` in ComputerBoards adaptor
`CachedEnumProp`	Cached property that includes the ability to manipulate enumerated values. See "Working with Enumerated Properties" on page 4-12.	`pChannelSkewMode` in Keithley analog input
`DoubleArrayProp`	For working with array of doubles. Cannot be created by adaptor.	`pTriggerConditionValue` in Keithley analog input
`IntArrayProp`	For working with array of integers. Cannot be created by adaptor.	None

Most properties are attached to in the `Open` method of the subsystem.

Note You do not need to release properties you have attached to; this is done automatically when you delete your object. However, you do need to release properties obtained using the lower level `GetMemberInterface`.

Creating Adaptor-Specific Properties

Apart from attaching to specific common properties, adaptors might require one or more properties that are specific to the hardware being supported by that adaptor. For instance, the Keithley adaptor defines the user-modifiable properties for setting Stop Triggers (`StopTriggerChannel`, for instance, is the channel to set for an analog stop trigger). If your adaptor needs to define new properties, you should create them using the `CREATE_PROP` macro.

The `CREATE_PROP` macro is used in the same manner as the `ATTACH_PROP` macro described in "Attaching to a Property" on page 4-5. First, a public member variable having one of the Adaptor Kit defined property types (see Table 4-1 and Table 4-2) must be defined, beginning with a "p" followed by the descriptive name for the property. Then, in the `Open` method of the required subsystem, the `CREATE_PROP` macro is called with the name of the property and the named member variable for that property.

The following code segment from `keithleyain.h`, the header file of the Keithley analog input subsystem, demonstrates definition of the adaptor-specific property member variables:

```
// Engine Properties - Device Secific
CachedEnumProppStopTriggerType;
TRemoteProp<double>pStopTriggerChannel;
CachedEnumProppStopTriggerCondition;
TRemoteProp<double>pStopTriggerConditionValue;
TRemoteProp<double>pStopTriggerDelay;
CachedEnumProppStopTriggerDelayUnits;
CachedEnumProppTransferMode;
```

These properties are used in the `Open` method of the same subsystem (`keithleyain.cpp`):

```
// This is the Stop Trigger Condition Value Property
CREATE_PROP(StopTriggerConditionValue);
SetDefaultStopTriggerConditionValues();

// This is the Stop Trigger Delay Property
CComVariant _minstoptriggerdelay(0);
CComVariant _maxstoptriggerdelay(INFINITY);
CComVariant _defaultstoptriggerdelay(0);
```

```
_maxstoptriggerdelay.ChangeType(VT_I8);

CREATE_PROP(StopTriggerDelay);
pStopTriggerDelay->SetRange(&_minstoptriggerdelay,
   &_maxstoptriggerdelay);
pStopTriggerDelay->put_DefaultValue(_defaultstoptriggerdelay);
pStopTriggerDelay->put_Value(CComVariant(0L));

// This is the Stop Trigger Delay Units Property
CREATE_PROP(StopTriggerDelayUnits);
pStopTriggerDelayUnits->AddMappedEnumValue(SECONDS, L"Seconds");
pStopTriggerDelayUnits->AddMappedEnumValue(SAMPLES, L"Samples");

pStopTriggerDelayUnits.SetDefaultValue(SECONDS);
```

The next section explains how to set default values, current values, and property ranges. For a discussion of enumerated properties, see "Working with Enumerated Properties" on page 4-12.

Note Most adaptor-specific properties are either enumerated or double types. The current Adaptor Kit does not permit adaptor-specific properties to be arrays (a future version of the Adaptor Kit will support array properties). Hence, most adaptor-specific properties are defined as one of `CachedEnumProp`, `TLocalProp`, or `TRemoteProp` data types.

Supporting daqpropedit for Adaptor-Specific Properties

In order for your adaptor to work well with the property editor GUI (run by calling `daqpropedit` in MATLAB) you need to provide help for your custom properties. You need to modify the `privatePropDesc.m` file in the `$MATLABROOT\toolbox\daq\daq\private` directory. See the help on `privatePropDesc` for more information.

Modifying Property Values, Defaults, and Ranges

Once you have created a reference to a property, either locally or global to the subsystem or adaptor, you can modify property values, set defaults, and set valid ranges for that property.

Note In order to set the range and default values for a property, that property must be defined as type `RemoteProp` or `LocalProp`, either using the templates (`TremoteProp` and `TCachedProp`) or the classes (`CRemoteProp` and `ClocalProp`) defined by the Adaptor Kit.

To set the default value for a property, use the `put_DefaultValue` method of the `IPropRoot` interface (inherited by the classes and templates defined above), passing the default value typecast to a `CComVariant`.

To set the range for a property, use the `SetRange` method on the property, passing the minimum and maximum allowable values for that property to the method as `CComVariant` data types.

To set the current value of a property, you can use either

- The `put_Value` method of the property, passing a `CComVariant` as the value (for `CRemoteProp` data types this is the only way to change property values)
- Assignment, if the property is of type `TRemoteProp`, `TCachedProp`, or `CachedEnumProp`

The following example, from the Keithley analog input subsystem's `Open` method, sets the range, default value, and current value for the `ChannelSkew` property:

```
// This is the Channel Skew Property
ATTACH_PROP(ChannelSkew);
CComVariant _minchannelskew(m_minManChanSkew);
CComVariant _maxchannelskew(m_maxManChanSkew);
pChannelSkew.SetRange(_minchannelskew, _maxchannelskew);
double defaultskew = m_minManChanSkew;
pChannelSkew.SetDefaultValue(defaultskew);
pChannelSkew->put_Value(CComVariant(defaultskew));
```

Note The method calling syntax differs between `put_Value` and the methods `SetRange` and `SetDefaultValue`, because the property class template `CRemoteProp`, from which `TRemoteProp` and `TCachedProp` derive, defines the `SetRange` and `SetDefaultValue` methods, and so these methods need not be dereferenced to the `IPropRoot` interface. However, `put_Value` is not overloaded in any methods and must therefore be dereferenced to the `IPropRoot` method.

Setting a Range to Infinity

To set a property range to infinity, use the `math.h` header file and include the following lines in your code:

```
#include <limits>
#define INFINITY std::numeric_limits<double>::infinity()
```

Then to set a value to infinity, use a `CComVariant` of value `INFINITY`. The following example from the Keithley adaptor demonstrates this:

```
CComVariant _minstoptriggerdelay(0);
CComVariant _maxstoptriggerdelay(INFINITY);
CComVariant _defaultstoptriggerdelay(0);

_maxstoptriggerdelay.ChangeType(VT_R8);

CREATE_PROP(StopTriggerDelay);
pStopTriggerDelay->SetRange(&_minstoptriggerdelay,
   &_maxstoptriggerdelay);
pStopTriggerDelay->put_DefaultValue(_defaultstoptriggerdelay);
pStopTriggerDelay->put_Value(CComVariant(0L));
```

Setting a Null Default Value

Not yet documented.

Working with Enumerated Properties

Apart from double properties, the most common property type is an enumerated property. Enumerated properties appear to the user to be properties that take on one of a possible range of string values. Internally, the enumerated property is a long integer property. The Adaptor Kit provides the `CachedEnumProp` data type to handle these property types.

You work with enumerated properties in the same manner as other properties, with the following additional functionality:

- Removing individual enumerated values with `RemoveEnumValue(StringValue);`
- Removing all enumerated values with `ClearEnumValues();`
- Adding enumerated values with `AddMappedEnumValue(value, StringValue)`

In all cases, values must be passed as `CComVariants`. Typically, you should define all permissible enumerated values as C `enum` data types, to provide consistency in your code.

All strings passed to enumerated data types are case sensitive, and appear as you typed them in the call to `AddMappedEnumValue`. By convention, enumerated values have no spaces or underscores, and the first letter of each word in the type is capitalized. Hence, the `StopTriggerType` for the Keithley adaptor is one of `None`, `HwDigital`, or `HwAnalog`, and not `HW_Analog`, and so on.

The following example from the ComputerBoards adaptor demonstrates all the above actions on the `ClockSource` property of the 7t subsystem:

```
ATTACH_PROP(ClockSource);
pClockSource->AddMappedEnumValue(CLOCKSOURCE_SOFTWARE,
   L"Software");
if (_UseSoftwareClock)
{
   pClockSource->RemoveEnumValue(CComVariant(L"Internal"));
   pClockSource.SetDefaultValue(CLOCKSOURCE_SOFTWARE);
   pClockSource=CLOCKSOURCE_SOFTWARE;
}
else
{
```

```
    pClockSource->AddMappedEnumValue(
        MAKE_ENUM_VALUE(0,EXTCLOCK),
        L"External");
    pTransferMode->AddMappedEnumValue(DMAIO   , L"DMA");
    pTransferMode->AddMappedEnumValue(BLOCKIO ,
        L"InterruptPerBlock");
}
```

In the preceding example, the ComputerBoards adaptor allows software clocking for all devices. If a device only supports software clocking, `Internal` is removed from the enumerated types, and `Software` is set as the default value. If a device supports internal clocking, the adaptor provides three additional types: `External`, `DMA`, and `InterruptPerBlock`.

Passing Arrays to MATLAB Using Safe Arrays

Some adaptor properties need to return an array of data to MATLAB. One example is the `ObjectConstructorNames` property, defined by the adaptor's `AdaptorInfo` method, which must return an Mx3 cell array of object constructor strings. Another example is the `InputRange` property, also returned in `SetDaqHwInfo` as an Mx2 matrix of valid input ranges. This data must be passed to MATLAB as a `CComVariant` containing a SafeArray. This section explains how to pass array information to MATLAB.

This section provides only some sample code on how Safe Arrays may be used in the adaptor, as an example for you to follow. For a complete discussion on SafeArrays, consult your Visual C++ documentation.

The following sample code provides a template for creating and using SafeArrays with your adaptor. In this particular code sample, the `ObjectConstructorNames` property is being created for a parallel port adaptor.

Note Some code has been removed from the final adaptor implementation for conciseness. In practice, the `ObjectConstructorName` is typically populated at the same time as the `BoardNames` and `BoardIDs` properties, which are all SafeArrays. See the Keithley or ComputerBoards adaptors for examples.

```
// Create constructor string Array (NumOfPorts x 3 DAQ Subsystems)
// Build up subsystems arrays -- up to 3 subsystems per board
VARIANT varSubSystem;
CComBSTR *subsystems;
SAFEARRAY *pSubSys;
SAFEARRAYBOUND arrayBounds[2];
// Define array bounds
arrayBounds[0].lLbound = 0; //Rows
arrayBounds[0].cElements = numports;
arrayBounds[1].lLbound = 0; // Columns
arrayBounds[1].cElements = 3;

// Construct a SafeArray
pSubSys = SafeArrayCreate(VT_BSTR, 2, arrayBounds);
if (pSubSys!=NULL)
{
```

```
    // Associate SafeArray with Variant
    varSubSystem.parray = pSubSys;
    varSubSystem.vt = VT_ARRAY | VT_BSTR;
    hRes = SafeArrayAccessData(pSubSys, (void **)&subsystems);
    if (SUCCEEDED(hRes))
    {
        // Define Constructor strings and IDs
        wchar_t str[40];

        // Loop through each found port
        for (int i=0; i<numports; i++)
        {
            //This adaptor only supports digitalIO so set
            //the first (AI, AO) to null:
            subsystems[i].Append("");
            subsystems[i+numports]=(BSTR)NULL;
            // And set the Digital I/O string
            swprintf(str, L"digitalio('%s','LPT%c')",
                    (wchar_t*)ConstructorName,
                    BoardIDs[i]);
            subsystems[i+2*numports]=str;

        }//end for

        // Send Constructor names to DAQENGINE
        hRes = Container->put_MemberValue(
                L"objectconstructorname",
                varSubSystem);
    }
    // Destroy SubSystem SafeArray
    SafeArrayUnaccessData (pSubSys);
    SafeArrayDestroy (pSubSys);
}
return hRes;
```

In the preceding example, the following implementation points should be noted:

- The Safe Array `pSubSys` is contained within a `CComVariant varSubSystem` that is passed back to the engine; the Safe Array is never directly sent back to the engine.
- The Safe Array is created with the `SafeArrayCreate` method, as an array of type `VT_BSTR`, with the dimensions specified in `arrayBounds`.
- To place data into the Safe Array `subsystems`, a pointer to a `CComBStr` is set to the data memory location by the `SafeArrayAccessData` method.
- The strings are passed through the `subsystems` variable to the Safe Array, in column order (column one first, then column two, etc.).
- The only subsystem that this adaptor supports is digital I/O, for each device denoted by the array `BoardIDs` (the allocation of `BoardIDs` is not shown in this code).
- Once the subsystem has been passed to the engine (through the `varSubSystem CComVariant`) the Safe Array is freed and destroyed.

This procedure should be followed for all safe arrays used in the adaptor, with modifications as necessary. For example, you might need to use a different data type for both the Safe Array definition (`SafeArrayCreate`) and the data accessor, or define the dimensions appropriately for your adaptor.

5

Buffering Techniques

Overview

The Data Acquisition Toolbox is designed to provide a flexible range of implementation options to the adaptor developer, and to the user of the toolbox. You can write adaptors that implement only software-clocked acquisition, which limits the sampling frequency for acquisition tasks to around 500 Hz. To provide more functionality, you must be able to configure and start an acquisition task that runs continuously until interrupted by the user. Continuous acquisition tasks require high-speed access to data buffers.

This chapter provides an overview of how to implement buffering for hardware-clocked continuous tasks in the Data Acquisition Toolbox.

Note You do not need to read this chapter if you are only implementing software clocking, as the engine and adaptor kit code already implements software-clocked continuous acquisition.

This chapter starts by introducing buffering in the Data Acquisition Toolbox engine. The simplest kind of buffering (direct buffers) is then discussed, followed by ideas on implementing intermediate buffering in the adaptor.

Three types of buffering are discussed in this chapter:

- Direct buffering between the Data Acquisition Toolbox engine and the hardware device driver. This buffering technique is implemented in the Winsound and Keithley adaptors.
- Circular buffering in the adaptor. The engine and device driver are separated by a circular buffer located within the adaptor, and the adaptor is responsible for queuing data from the engine buffers, through the circular buffer to the hardware device, for analog output tasks, and from the hardware device through the circular buffer to the engine buffers, in the case of analog input tasks.
- Ping-pong buffering in the adaptor. Similar to circular buffering, but uses two separate buffers alternately.

Understanding Engine Buffers

For analog input and analog output tasks, the Data Acquisition Toolbox engine makes use of buffers of data to manage the data transfer, event notification, and memory management tasks. Engine buffers are responsible for storing data from an analog input task, prior to a user's requesting that data with a `getdata` function call, and for queuing data for an analog output task with `putdata`. Thus, data interaction between the adaptor and the engine takes place through engine buffers.

Engine buffers for any task are a predetermined size, presented to the user through the `BufferingConfig` property. Although the engine attempts to size buffers appropriately for the desired sample rate, the user can set the size of engine buffers by modifying the `BufferingConfig` property directly. Although it is possible to restrict the size of the engine buffers (see, for instance, the `hpe1432` adaptor, which limits the maximum size of engine buffers) the adaptor should not expect strict control over engine buffer sizes, as this would limit functionality for the user.

Buffer interaction between the engine and the adaptor takes place through `BUFFER_ST` structures, defined as follows (a full description of this structure is given in Appendix C, "Engine Structures"):

```
typedef  struct tagBUFFER {
long Size; // In bytes
long ValidPoints; // In raw points
                  //(MATLAB samples is ValidPoints/channels)
unsigned char *ptr;
DWORD dwAdaptorData;  // Reserved by the engine for use by adaptor
unsigned long Flags; // Flag values are defined in
unsigned long Reserved; // Reserved for future use by the engine
hyper StartPoint; // Count of points since start
double StartTime; // Time of the start of the buffer from GetTime
double EndTime; // Time of the end of the buffer from GetTime
} BUFFER_ST;
```

The buffer size is set by the `BufferingConfig` property. However, because acquisition tasks might not necessarily stop at a buffer boundary, the `ValidPoints` field contains the number of valid points (samples multiplied by the number of channels) contained (for analog output) or expected (for analog input) in the buffer of data passed by the engine. The data is stored as

interleaved samples in the memory space pointed to by ptr; the first channel of sample N is followed by the second channel of sample N, until all channels are filled, and channel 1 of sample N+1 follows. The Flags field contains, among others, a flag to indicate whether the buffer is the last buffer in the current task (Flags & BUFFER_IS_LAST).

Analog Input Tasks

For an analog input task, the engine provides the adaptor with empty buffers whenever the adaptor calls GetBuffer. The last buffer the engine expects to be requested is flagged with the BUFFER_IS_LAST flag.

Before sending a buffer back to the engine, the ValidPoints and StartPoint fields should be filled in the BUFFER_ST structure, and the acquired data must be copied into the memory space pointed to by ptr. If the adaptor has better time information than the engine, it should fill in the start and end times for the buffer and set the flags BUFFER_START_TIME_VALID and BUFFER_END_TIME_VALID in the flags member.

The adaptor should send the buffer back to the engine using the PutBuffer engine method.

If the BUFFER_IS_LAST flag is set in the buffer, stop the acquisition after filling that buffer with ValidPoints.

Analog Output Tasks

For an analog output task, the engine provides the adaptor with a buffer of data whenever the adaptor calls GetBuffer. The ValidPoints field informs the adaptor how many points in the buffer should be sent to the hardware device. Once the adaptor finishes with the buffer, the buffer must be sent back to the engine using PutBuffer.

If the BUFFER_IS_LAST flag is set in the buffer, stop the output task after sending that buffer's ValidPoints to the hardware.

Buffering with Trigger Repeat

When trigger repeats are used, the engine does not set the BUFFER_IS_LAST flag of the buffer structure until the acquisition has repeated the required number of times. For example, a 1000 sample analog input task with TriggerRepeat set to 1 continues until 2000 samples have been retrieved.

Dealing with Analog Output Out Of Data Mode

The `OutOfDataMode` property of analog output subsystems defines what should happen to the analog output signal when the task ends: Either the value should be held (`OutOfDataMode` set to `Hold`) or the default value for the channel must be sent to the device (`OutOfDataMode` set to `Default`). The output task should send the correct value to the hardware prior to stopping acquisition. Because the device might queue data in a hardware buffer, you should ensure that the `OutOfDataMode` values get to the physical device before stopping the acquisition. Typically, this involves repeating the `OutOfDataMode` value many times. The Keithley adaptor demonstrates this behavior (see the `LoadData` method of the analog output subsystem).

Implementing Buffering in Your Adaptor

Before implementing buffering in your adaptor, you should consider the type of buffering and event notification supported by your device driver. The following sections discuss direct buffering and circular buffering techniques. Although other forms of intermediate buffering exist, this adaptor kit recommends using circular buffers for intermediate buffering.

Direct Buffering

Direct engine-driver buffering is by far the simplest form of buffering to implement. In this form of buffering, data is passed directly between the engine buffers and the device driver buffers. The Keithley and Winsound adaptors implement direct buffering.

Direct buffering requires the following support from the hardware device driver:

- You can define the buffer size for your device driver (otherwise, you cannot match the engine buffer size with the device driver buffer size).
- Your driver implements multiple buffers, or allows you to provide pointers to multiple blocks of memory — the engine can provide noncontiguous memory between each buffer, or
- You are prepared to implement a timer-based checking mechanism, and you can query the number of samples acquired (or output) by the device and can request a variable number of samples from the device.

If any of these conditions is not met, you might have trouble implementing direct buffering, and should consider using circular buffering instead.

For an example of direct buffering, see the Keithley or Winsound adaptors.

The basic points to consider when implementing direct buffering are as follows. Note that in this discussion, analog input acquisition is discussed; for output, simply replace acquisition with output:

- You can use the engine's `GetBufferingConfig` method to obtain the current engine buffer size (in samples).
- You should ensure that your device driver can acquire data for at least two engine buffers (more buffers are recommended, particularly for high sample

rates). This allows you to transfer data out of one buffer while the driver queues data to another location.

- Your adaptor should be able to detect when a buffer has been filled by the driver, either through driver callbacks (on buffer filled, or on every N samples) or through a timer routine that operates at least twice as fast as a buffer of data is filled.
- When your adaptor detects that a buffer has been acquired, you should call `GetBuffer` and transfer the raw data values directly from the device driver into the engine buffer.
- For analog input, you should always check that a valid buffer has been provided by the engine. If the buffer structure is returned as `NULL`, you should stop acquisition immediately.
- For analog output, you should be careful of the device's hardware FIFO buffer. Many device drivers indicate driver status, and not necessarily the sample that has been output by the hardware. You should therefore queue FIFO more samples than the engine indicates before stopping the acquisition task.

The following code is an extract from the Keithley adaptor showing the handling of the buffer filled message (from the `ReceivedMessage` method):

```
if (m_daqStatus == STATUS_RUNNING)
{
    BUFFER_ST * pBuffer;
    _engine->GetBuffer(0, &pBuffer);
    if (pBuffer==NULL)
    {
        double triggerrep = pTriggerRepeat;
        if(m_samplesThisRun < (pSamplesPerTrigger *
                (1 + triggerrep)))
        {
            _engine->DaqEvent(EVENT_DATAMISSED, -1,
                    m_samplesThisRun,NULL);
            return;
        }
        mustStop = true;
    }
    else
    {
```

```
samplesToFetch = pBuffer->ValidPoints/_nChannels;
// Get the Data from DriverLINX.
GetDriverLINXAIData(m_pSR,
        (unsigned short*) pBuffer->ptr,
        bufIndex, _nChannels, samplesToFetch);
ResultCode = GetDriverLINXStatus(m_pSR, Status, Length);
if (ResultCode != 0)
{
    _engine->WarningMessage(
            TranslateResultCode(ResultCode));
    mustStop = true;
}
else
{
    // The conversion worked, now send the data to MATLAB
    long pointsPerBuffer = m_engineBufferSamples *
            _nChannels;
    pBuffer->StartPoint = m_samplesThisRun * _nChannels;
    m_samplesThisRun+=samplesToFetch;

    // Set the number of valid points in this buffer
    pBuffer->ValidPoints = samplesToFetch * _nChannels;
    if ((pBuffer->Flags & BUFFER_IS_LAST) ||
            (pBuffer->ValidPoints < pointsPerBuffer))
        mustStop = true;
    _engine->PutBuffer(pBuffer);
    if (m_triggering && (pTriggerType==TRIGGER_HWDIGITAL)
            && !m_triggerPosted)
    {
        m_triggering=false;
        m_triggerPosted = true;
        _engine->GetTime(&time);
        triggerTime = time -
                m_engineBufferSamples/pSampleRate;
        _engine->DaqEvent(EVENT_TRIGGER, triggerTime,
                pSamplesPerTrigger*m_triggersProcessed,
                NULL);
        m_triggersProcessed++;
    }
}
```

```
    }
    if (mustStop)
        Stop();
}
```

Intermediate Buffering

Intermediate buffering consists of the adaptor's temporarily storing data internally when transferring between the engine and the device driver. Intermediate buffering has the following advantages:

- By providing an intermediate storage layer between the engine and the device driver, you do not need to ensure that the driver provides data in engine-buffer-sized chunks.
- Intermediate storage allows you to convert data between proprietary formats and the native data type defined by your adaptor.
- If you decide to implement circular intermediate buffering, you can transfer irregular sizes of data between the hardware device driver and your intermediate buffer. This makes implementation of the adaptor code robust to changes in operating system loading.

Intermediate buffering is common, only because implementation of intermediate buffering is more flexible than direct buffering discussed previously. Hence, developers of a variety of adaptors might prefer intermediate buffering techniques.

You should consider the following points when implementing intermediate buffering:

- Ensure that the total buffer space is large enough to handle continuous acquisition to or from the engine buffers. Typically this means making the intermediate buffers at least three to four times larger than the engine buffers.
- When using polled acquisition, ensure that the data transfer is polled often enough, or that timer routines execute callbacks often enough, so that data transfer can take place between the circular buffer and both the engine and the device driver in a timely way. Typically, polled acquisition should check the acquisition status at least twice per engine buffer transfer.
- Try to pass data to and from the engine as soon as it is available in your buffer, and not when two or three engine buffers need to be transferred. This

ensures that the engine can provide information to the user as soon as possible, and you do not need to store data in your adaptor for too long.

Circular Buffering: Using cirbuf.h

Circular buffering involves acquiring data into a single contiguous memory space, with older data being overwritten when the buffer is full and newer data becomes available. Circular buffers require two pointers: a write pointer, which writes data into the circular buffer, and a read pointer, which reads data from the buffer. The pointers must be specially implemented, as they have to know to wrap back to the beginning of the buffer when they reach the end. As long as the read pointer does not overlap the write pointer, no data is lost.

Circular buffers are the most common form of intermediate buffering in the Data Acquisition Toolbox, owing to the flexibility of circular buffering, particularly in supporting multiple types of device driver implementation. The Adaptor Kit therefore provides an implementation of circular buffers that should be used in an adaptor that implements circular buffers.

Circular buffers are defined in the `cirbuf.h` header file provided in the `include` directory of the source code for the adaptors shipped with the Data Acquisition Toolbox. You can find the `include` directory in `$MATLAB\toolbox\daq\daq\src`.

The following implementation details describe how to use the Circular Buffer template class provided in `cirbuf.h` (search for `_CircBuff` in the ComputerBoards adaptor source for implementation examples):

- Define the buffer using the `TCircBuffer` template, using the native data type defined by the adaptor.
- In the `Start` method, initialize the buffer using the `Initialize` method, passing the size of the buffer in points.
- Use the `GetPtr` and `GetBufferSize` methods to obtain the pointer location and size of the buffer. These values can be passed directly to device drivers that assume circular buffering in their implementations of continuous acquisition.
- The `ValidData` method returns the number of valid points in the buffer.
- Use the `CopyIn` method to transfer data to the buffer. Pass a pointer to the data to be copied and the number of points to transfer.

- Use the `CopyOut` method to copy data out of the buffer. Pass a pointer to the location to copy data to, and the number of points to copy out of the buffer.
- You can set the write location to a particular value by passing a pointer or a position to `SetWriteLocation`.
- You can query the space available in the buffer by calling the `FreeSpace` method.
- To check for overruns, use the `IsWriteOverrun` method, passing the number of points you want to write to the buffer.

For a full implementation of circular buffers, see the ComputerBoards and Nidaq adaptors.

Implementing Other Intermediate Buffering

Other types of buffers can provide performance advantages over circular buffers. For example, ping-pong buffers (where data is transferred between alternating buffers) might provide a simpler implementation than circular buffering, because buffers are a defined size and data does not wrap around the buffer. Other types of buffering include multiple buffering (the extension of ping-pong buffers to multiple buffers) and threaded buffering. Note, however, that circular buffering can be configured to look like almost any other type of buffering.

6

Callbacks and Threading

Overview

This chapter discusses how to handle data transfer between continuous acquisition tasks and the engine.

Many device drivers provide some mechanism for notification of events in a continuous task. Other device drivers only provide an indication of the current sample being acquired or output from their internal buffers. The Data Acquisition Toolbox expects to receive or send data in relatively small buffers. These buffers are small to enable additional information such as triggers and progress notification to be handled rapidly by the Data Acquisition Toolbox. For example, a MATLAB user might want to know when every 1024 samples of an acquisition have taken place, so that the data can be extracted and an FFT calculated on that new data.

As an adaptor developer, you must be able to monitor the progress of an acquisition task, so that you can provide acquired data to the engine as fast as possible, send more data to the hardware device in the case of analog output tasks, and provide event notification to the engine. This progress monitoring must not be blocked by any other process, including MATLAB. Hence, monitoring of continuous acquisition tasks typically requires the implementation of a separate thread to handle that monitoring task.

This chapter discusses types of progress monitoring, and leads on to the implementation of threads in your adaptor. The problem of monitoring the progress can be broken into three distinct areas:

1. The manner in which your device driver implements progress monitoring dictates one of three general progress monitoring schemes that your adaptor implements.

2. The format in which the device driver returns information dictates how you implement the progress monitoring.

3. The outcomes of the first two areas provide an idea of how to thread the progress monitoring task to ensure that MATLAB does not block acquisition tasks.

Monitoring Progress of Acquisition Tasks

Hardware device drivers that provide continuous acquisition capabilities must provide some mechanism for the program initiating those tasks to obtain information on the progress of the task. Typical progress messages include notification about the current position of the acquisition task (how many samples the analog input task has acquired, or how many samples the analog output task has sent to the hardware device) and other information such as data overrange errors, device driver buffer overruns, triggers, and hardware device errors.

Typically, two types of progress monitoring are implemented: synchronous notification (i.e., after every N samples) through callbacks or messaging, and asynchronous, or polled, acquisition status notification.

Event Messaging from Device Drivers

Device drivers that implement event messaging are often easier to implement than polled drivers. One major reason is that your adaptor can be notified whenever an engine buffer has been processed. Hence, the overhead in monitoring the status of an acquisition task is minimal. However, not all drivers provide this mechanism.

Callback-type drivers provide a mechanism for executing a particular callback each time the driver has finished processing a defined number of samples. For example, the Nidaq adaptor uses the Nidaq event message handler to run a callback function whenever the driver has completed sending or sampling an engine buffer's worth of samples. The callback function can then queue more data, or, in the case of a special event from the device driver, stop acquisition or register nonfatal events (such as overrange errors) with the Data Acquisition Toolbox.

Drivers can also use window message handling techniques to notify the calling application of the progress of an acquisition task. For instance, the Keithley driver posts window messages to a window handle that is registered when your application first calls the DriverLINX driver. The driver then posts Buffer Filled messages when it has finished using an internal buffer, and the application is then free to reuse the buffer (either to fetch data from the buffer, or to put more data into that buffer).

If your adaptor supports synchronous event messaging, either through callbacks or through window messaging, you should configure the messages to

be posted each time the driver handles an engine buffer's worth of data. This minimizes the buffering logic within your adaptor.

A typical implementation of synchronous message handling can be found in the Keithley adaptor's `ReceivedMessage` method, and in the Nidaq adaptor's `Callback` method. These methods are implemented for both analog input and analog output subsystems.

Polling the Driver for Acquisition Status

If your adaptor does not provide synchronous callbacks, you can almost certainly poll the driver for the status of an acquisition task. Typically, information that might be returned from such a polling request would include information on the number of samples processed, as well as any errors that might have occurred with the task, such as buffer overruns, data overrange warnings, and other error messages. The Data Acquisition Toolbox should be notified of these events as soon as possible after they happen.

Note If your driver supports neither polled nor synchronous messaging, then you cannot use continuous sampling, and must resort to using software clocking for your acquisition task.

Polling typically requires a little more work from your adaptor than synchronous event messaging. Typically, you need to set up your own polling mechanism for querying the status of an acquisition task frequently. Unfortunately the Windows operating system is not a hard real-time system, so you cannot be guaranteed that your polling task will execute exactly on time every time. However, if you design the polling system adequately, and implement sufficient buffering within your adaptor, you can handle latencies that polling requests and other tasks running on the system impose on your adaptor.

Polling should be implemented as follows:

- Use a Windows Timer routine to poll the driver for task status. This routine should occur more frequently than the transfer of a single engine buffer to the driver (typical implementations execute the timer routine twice per buffer transfer to ensure minimal latency).

- Each time the driver has transferred an engine buffer's worth of data, transfer another buffer from the engine to your internal buffer.
- If there is sufficient space left in your driver's buffer, queue more data (or fetch more data) from the driver buffer to your adaptor buffer.
- In the event of an error or warning, send an event notification to the engine, using the most recent sample count from the driver for timing information.

From the preceding discussion it is clear that polled acquisition requires the use of an intermediate adaptor buffer. Without the intermediate buffer the buffering routine would become too complex to manage and implement adequately.

Although polling routines can be more difficult to implement, there is one positive benefit of using polling: The Windows Timer routines are automatically multithreaded, which makes implementation of a separate thread for task monitoring easier to implement.

For an example of polled acquisition, see the `TimerRoutine` method in the Winsound adaptor or the `GetScanData` method in the Analog Input subsystem of the ComputerBoards adaptor.

Threading Your Adaptor's Task Monitoring Methods

Your adaptor runs in the same process space as MATLAB's normal operations. During an acquisition task, MATLAB might be asked by the user to perform some other processing, such as visualizing previously acquired data, or performing some analysis on other data.

Some device drivers implement callbacks in a separate thread. In this instance you do not have to explicitly perform any threading yourself, although your adaptor code should be thread safe in this instance.

You might decide in the development of your adaptor that you will not implement threading. If this is the case, you should be aware that the following problems will arise:

- Because MATLAB should not interrupt the adaptor's task, you lose the support of all event callbacks (such as `TriggerFcn`, `SamplesAcquiredFcn`, etc.). This means that your adaptor can never perform analysis while an acquisition task is running.
- Your adaptor cannot block MATLAB's processing, as the engine performs nonblocking calls to the adaptor's methods.
- If MATLAB code is run while a task is executing, the task might lose data.

As long as you are prepared to accept the limitations listed above, you do not need to worry about threading your adaptor. In the event that you need the preceding functionality, you might need to implement threading of the adaptor's callback functions.

Threading can be implemented after the fact. Typically, getting the adaptor working in a single thread is easier than debugging threading and adaptor implementation simultaneously. However, if you know that threading is required, you should implement the threading code when implementing the callback functionality.

The following sections discuss how to thread your adaptor based on the task monitoring your device driver supports.

Implementing Callbacks in a Separate Thread

Many device drivers run callbacks in their own threads, effectively providing you with multithreaded capabilities without requiring any code in your

adaptor. The `nidaq` and `hpe1432` adaptors provide an example of direct threaded callback support.

If your driver does not support callbacks in a separate thread, you must implement threading in the adaptor. Depending on your device driver implementation, this might require you to start the task execution in the separate thread. Consult your device driver API documentation for details on how your device driver supports multithreaded applications.

Implementing Event Messaging in a Separate Thread

Event messaging takes place through standard Windows event handlers. In order to implement threading for such device drivers, you need to create the window receiving all the event messages in a separate thread, and implement a message processing function that handles all driver messages as well as other standard window messages, such as close messages.

The thread you create should be as lightweight as possible, to avoid consuming valuable processor time while an acquisition task is running. A suitable thread processing algorithm is given (modified from the Keithley adaptor to aid in readability):

```
unsigned WINAPI MessageWindow::ThreadProc(void* pArg)
{
    MSG msg;
    try
    {
       CoInitialize(NULL);
       MessageWindow* thisptr=(MessageWindow*) pArg;
       thisptr->CreateMyWindow();
       // Code to open device drivers in this thread
       thisptr->OpenDriverLINXDriver( boardName, &hinst);
       thisptr->_windowEvent.Set();
       // Main message loop:
       while (!thisptr->_isDying)
       {
           GetMessage(&msg, NULL, 0, 0);
           TranslateMessage(&msg);
           DispatchMessage(&msg);
       }
```

```
            CloseDriverLINX(boardName);
         CoUninitialize();
      }
      catch (...)
      {
         _RPTO(_CRT_ERROR,"***** Exception in Keithley
             Thread Terminating Thread\n");
      }
      return 0;
   }
```

The algorithm can be summarized as follows:

- Create the window to handle all messaging.
- If required, open your device driver (the Keithley adaptor requires this step).
- Loop until the thread is asked to quit by the adaptor's closing, calling `GetMessage` and `DispatchMessage` methods in the loop. (Note: `GetMessage` is a lightweight blocking call, and puts the thread into idle priority until a message is sent to the thread or the thread's windows.)
- When the loop exits, close the device driver if necessary.

For an implementation of event message threading, consult the Keithley adaptor class. Typically, messaging should be installed in the adaptor class, as it is required by the analog input and analog output classes, and the code is almost completely repeated. The only exception is that the specific message handling code should be able to call the appropriate analog input or analog output message-handling routines (see `ReceivedMessage` in the analog input and analog output classes of the Keithley adaptor).

Implementing Polling in a Separate Thread

Polling in a separate thread typically involves using a timer to initiate periodic calls to the device driver. For the adaptor that has used this mechanism to date (the ComputerBoards adaptor), the Windows Multimedia Timer functions have provided this functionality. The Windows Multimedia Timer functions automatically implement the callback in a separate thread, and so you should not need to explicitly thread your callback. You can access these functions using the `TTimerCallback` template included with the Adaptor Kit.

The `TTimerCallback` template defines an object with the following methods.

Method	Description
`CallPeriod`	Defines the time in seconds between callback execution
`Stop`	Stops the timer
`TimerRoutine`	Method in object passed to `TTimerCallback` that implements the callback

On construction of your analog input or analog output object, you should construct the `TTimerCallback` object, passing the current object as the parameter.

For an example of using the Windows Multimedia Timer for polling, see the `GetScanData` method of the ComputerBoards adaptor.

A

Adaptor Kit Interface Reference

Overview

Every adaptor DLL that communicates with the data acquisition engine must implement a subset of the following COM interfaces:

- `ImwDevice`
- `ImwAdaptor`
- `ImwInput`
- `ImwOutput`
- `ImwDIO`

Of these interfaces, only `ImwDevice` and `ImwAdaptor` are required. Implementation of the remaining interfaces depends on the functionality provided by your adaptor.

This chapter provides detailed descriptions for all the methods declared by these adaptor kit interfaces. The methods are shown in their appropriate formats and with appropriate return types. The method descriptions use a quasi COM notation that uses the attributes `[in]`, `[out]`, and `[in,out]` to denote input, output, and input/output parameters, respectively. An input parameter is passed by a caller and is not changed by the method being called. An output parameter is assigned by the method and returned to the calling procedure. An input/output parameter combines both properties.

ImwDevice

The ImwDevice interface serves as a base for classes that implement generic device functionality common to all data acquisition devices. ImwDevice declares the methods given below.

Table A-1: ImwDevice Methods

Method	Purpose
AllocBufferData	Allocate requested memory for a data buffer.
FreeBufferData	Free the data field of a buffer.
SetChannelProperty	Configure the specified channel property.
SetProperty	Configure the specified device property.
Start	Initialize a data acquisition process.
Stop	Stop data acquisition process.
GetStatus	Determine the number of samples acquired, or the number of samples output.
ChildChange	Add, delete, or reindex a channel or line.

AllocBufferData

Syntax

```
HRESULT AllocBufferData( [in, out] BUFFER_ST *Buffer )
```

Description

The AllocBufferData method allocates requested memory for a data buffer. The data buffers are used for transferring data between the engine and the adaptor. For analog input, a buffer filled with data is transferred from the adaptor to the engine using the IDaqEngine method PutBuffer. An empty buffer is transferred to the adaptor from the engine using the IDaqEngine method GetBuffer.

For analog output, a buffer containing data is transferred to the adaptor with GetBuffer, and the emptied buffer is returned to the engine with PutBuffer.

A single argument of the type pointer to the BUFFER_ST structure is used as both the input and output parameter. On the call to the function, the size of the requested buffer is passed to the function as the value of the size field of the Buffer parameter. On return, the pointer to the newly allocated data array is returned as the *ptr field of the Buffer parameter.

The structure BUFFER_ST is defined in the file daqmex.idl. It is described in detail in the PeekData method.

The function AllocBufferData has a default implementation in the class CmwDevice, defined in the files AdaptorKit.h and AdpatorKit.cpp. Normally, it does not need to be modified by the adaptor programmer.

FreeBufferData

Syntax

```
HRESULT FreeBufferData( [in, out] BUFFER_ST *Buffer )
```

Description

The FreeBufferData method frees the data field of a buffer. The function is called by the engine to deallocate memory that was previously allocated for the data array of the buffer. It frees the memory belonging to the *ptr field and sets the size field of the buffer to 0.

This function is implemented in the class CmwDevice, and normally should not be redefined for specific adaptors.

SetChannelProperty

Syntax

```
HRESULT SetChannelProperty( [in] long user, [in] NESTABLEPROP
*pChan, [in,out] VARIANT *NewValue )
```

Description

The SetChannelProperty method configures the hardware for a new value of the specified channel property. It is called by the engine after you call the toolbox set function. For example, to configure the SensorRange property for the first channel added to the analog input object ai using the set function

```
set(obj.Channel(1),'SensorRange',[-1 2])
```

After you issue the `set` function, the engine must determine whether to pass the property value to the adaptor. If you attempt to assign to a property the same value it already has, the `SetChannelProperty` function is not called by the engine.

The adaptor does not have to be notified about all property changes. Some of these changes have no bearing on the hardware and should not be communicated to the adaptor. In such cases, the `set` function is processed entirely by the engine. An example of such a property is `ChannelName`. When the adaptor needs the engine to call the `SetChannelProperty` function for a given property, it must register this property. This is accomplished by calling the function `put_User`, which is declared by the engine interface `IPropRoot`.

Parameters

- `user` — Address of the property, which was communicated to the engine when it was registered with the `put_User` function.
- `*pChan` — A pointer to the structure containing the information about the channel whose property is being modified.
- `*NewValue` — A pointer to a new value to be assigned to the property. Because its type is VARIANT, it can accommodate any data type, such as strings, arrays, or simple types. The requested new value is passed to this parameter, and the actual value (which might not coincide with the requested value because of hardware limitations) is returned to the engine via this parameter. Therefore it is qualified as `[in,out]`.

SetProperty

Syntax

```
HRESULT SetProperty( [in] long user, [in,out] VARIANT *NewValue )
```

Description

The `SetProperty` method configures the hardware for a new value of the specified device property. It is called by the engine after you call the toolbox `set` function. For example, to configure the `SampleRate` property for the analog input object `ai` using the `set` function

```
set(ai,'SampleRate',1000)
```

After you issue the `set` function, the engine must determine whether to pass the property value to the adaptor. If you attempt to assign to a property the same value it already has, the `SetProperty` function is not called by the engine.

The adaptor does not have to be notified about all property changes. Some of these changes have no bearing on the hardware and should not be communicated to the adaptor. In such cases, the `set` function is processed entirely by the engine. An example of such a property is `Name`. When the adaptor needs the engine to call the `SetProperty` function for a given property, it must register this property. This is accomplished by calling the `put_User` function, which is declared by the engine interface `IPropRoot`.

Parameters

- `user` — Address of the property, which was communicated to the engine when it was registered with the `put_User` function.
- `*NewValue` — A pointer to a new value to be assigned to the property. Because its type is VARIANT, it can accommodate any data type, such as strings, arrays, or simple types. The requested new value is passed to this parameter, and the actual value (which might not coincide with the requested value because of hardware limitations) is returned to the engine via this parameter. Therefore it is qualified as an `[in,out]` parameter.

Start

Syntax

```
HRESULT Start()
```

Description

The `Start` method initializes the data acquisition process and sets the `Running` property to `On`. Depending on the object on which it is called, the process can be associated with analog input, analog output, or digital input/output. The function is called by the engine as a response to the toolbox `start` function.

The function takes no arguments. For all adaptors that use software clocking, `Start` has adequate default implementation as defined in the file `AdaptorKit.cpp`, and should not be redefined. However, for adaptors that use onboard hardware clocks, `Start` must be overridden within the derived adaptor

classes. For example, for the demo adaptor (presuming it used hardware clocking), it could be redefined in the file `demoin.cpp` inside the class `Cdemoin`.

For analog input and analog output, it is the `Trigger` function that actually starts the acquisition process. In most cases, the engine calls `Trigger` immediately after calling `Start`.

Stop

Syntax

```
HRESULT Stop
```

Description

The `Stop` method stops a data acquisition process and sets the `Running` property to `Off`. It can be called by the engine after you issue the toolbox `stop` function. For analog input objects, `Stop` is called internally by the adaptor when the last available buffer has been filled. The adaptor must then post a stop event using the `DaqEvent` method of the `IDaqEngine` interface.

The function takes no arguments. By default, it is defined in the `AdaptorKit.cpp` file and is adequate as implemented for all adaptors that use software clocking. For adaptors that use onboard hardware timers, `Stop` must be overridden inside classes derived from `CmwDevice`.

GetStatus

Syntax

```
HRESULT GetStatus( [out] hyper *samplesProcessed, [out] BOOL
*running)
```

Description

The `GetStatus` method is called by the engine to determine the number of samples acquired (analog input) or the number of samples output (analog output). This method also informs the engine whether the hardware is currently running.

The current implementation of the engine only calls `GetStatus` for analog output.

Parameters

- `*samplesProcessed` — A pointer to the number of samples that have been output by the time of the query.
- `*running` — A pointer to the Boolean value, indicating whether the device is running. True if running, false otherwise.

ChildChange

Syntax

```
HRESULT ChildChange( [in] DWORD typeofchange, [in,out] NESTABLEPROP
*pChan)
```

Description

The engine calls the `ChildChange` method when a channel or a line is added, deleted, or reindexed. These processes are initiated when you use the toolbox functions `addchannel`, `delete`, or `set`, respectively.

Parameters

- `typeofchange` — Indicates why the function is called. This parameter takes five valid values of an enumerated type, which are defined in the file `DaqmexStructs.h`. The values are given below.

Action	Value
`ADD_CHILD`	1
`REINDEX_CHILD`	2
`DELETE_CHILD`	3
`START_CHANGE`	256
`END_CHANGE`	512

The last two values are used as a mask, which can be ORed with any of the first three values. Thus, the engine can call `ChildChange` with requests to execute different stages of the channel change process. With the

START_CHANGE mask, the engine calls the part of the code that should be executed before the channel is added, deleted, or reindexed. With the END_CHANGE mask, the engine calls the part of the code that must be executed after the channel (or line) change.

- *pChan — A pointer to the structure of the type NESTABLEPROP, containing the information about the channel that is being deleted, added, or reindexed.

  The NESTABLEPROP structure is defined in the file DaqmexStructs.h. This parameter is used by the engine as an [in] parameter to send the channel information to the adaptor. The adaptor then returns this information to the engine after modification, using the parameter as an [out] parameter.

ImwAdaptor

The `ImwAdaptor` interface is responsible for opening a specified function of the physical device and establishing the communication between the DLL and the engine on behalf of this function. Additionally, it supplies the engine information about the particular device via the MATLAB function `daqhwinfo`. `ImwAdaptor` declares the methods given below.

Table A-2: ImwAdaptor Methods

Method	Purpose
`AdaptorInfo`	Return information associated with the specified adaptor.
`OpenDevice`	Construct an instance of an adaptor and initialize the hardware device.
`TranslateError`	Translate error codes into readable error messages.

AdaptorInfo

Syntax

```
HRESULT AdaptorInfo( [in] IPropContainer * Container )
```

Description

The `AdaptorInfo` method is called when you call the toolbox `daqhwinfo` function with the adaptor name as an input. For example, to return adaptor information for the demo adaptor, you issue the command `daqhwinfo('demo')`.

`daqhwinfo` returns certain property values from the hardware driver and uses this information to modify the special property structure. The associated properties are shown below.

Table A-3: Adaptor Properties Returned by daqhwinfo

Property Name	Data Type	Description
`AdaptorName`	BSTR	Name of the adaptor
`AdaptorDllName`	BSTR	Full path name of the adaptor DLL

Table A-3: Adaptor Properties Returned by daqhwinfo (Continued)

Property Name	Data Type	Description
AdaptorDllVersion	BSTR	Revision of the adaptor DLL
BoardIds	BSTR array	IDs of the hardware devices installed in the computer
BoardNames	BSTR array	Names of the hardware devices of the specified type installed in the computer
ObjectConstructorNames	BSTR array	The array of all possible MATLAB commands that can construct all installed hardware devices of the specified type

The ObjectConstructorNames property is an array of strings, where every string is one possible command to open the device. There is one string for each subsystem (analog input, analog output, or digital I/O) supported by the board. If a board does not support a particular subsystem, the corresponding strings are empty. For example, the ObjectConstructorNames property values for the winsound adaptor are shown.

```
info = daqhwinfo('winsound');
info.ObjectConstructorNames
info.ObjectConstructorName(:)
ans =
    'analoginput('winsound',0)'
    'analogoutput('winsound',0)'
```

You now know what commands you can issue to construct all possible objects for the winsound device. You can also use MATLAB's eval command to construct objects programmatically. For example:

```
eval(['hAI = ', info.ObjectConstructorName{1}])
```

AdaptorInfo uses the pointer to the IPropContainer interface (passed by the engine as a single parameter) to call the put_MemberValue method to modify the structure. For example:

```
hRes = Container->put_MemberValue( L"adaptordllname",
CComVariant(name) );
```

The approach you should use to implement the `AdaptorInfo` function depends on the complexity of the adaptor DLL you are building. The implementation given in the demo adaptor example (as well as any adaptor based upon it) automatically correctly loads the first two fields of the DAQHWINFO structure: `AdaptorName` and `AdaptorDllName`. The third field, `AdaptorDllVersion`, is loaded by the engine based on the following line in the resource file `demo.rc`:

```
VALUE "FileVersion", "<******>\0".
```

This line is located in the block `StringFileInfo`. You should replace `"<******>"` with a string that reflects the version number.

The information for the other three fields is not as readily available. There are two possibilities for their realization, depending on your goal:

- For a simple adaptor that is associated with only one type of hardware device, this information can be hard coded. This approach is employed in the demo adaptor.
- If the adaptor DLL is intended to communicate with a variety of devices sharing the same hardware driver, you must use the API calls provided by the driver. Once this information is obtained, it is communicated to the engine with a call to `put_MemberValue`. For an example of this implementation, refer to the `AdaptorInfo` function in the `winsound` adaptor.

The sole purpose of the `AdaptorInfo` function is to present information to the user. It is not used by the engine internally and is called only as a response to your requests. This is the way you obtain information about the installed hardware as well as the adaptors. Prior to using this method, you must register the adaptors of interest, and the hardware drivers for the data acquisition boards must be installed.

OpenDevice

Syntax

```
HRESULT OpenDevice( [in] REFID DevIID, [in] long nParams, [in]
VARIANT *Param, [in] REFID EngineIID, [in] IUnknown *pEngine, [out]
void **ppIDevice )
```

Description

The `OpenDevice` method constructs an instance of an adaptor and initializes the hardware device. Additionally, it makes the engine and the adaptor exchange pointers, which enables subsequent calls to each other's methods.

`OpenDevice` is called by the engine when you request the construction of a data acquisition object. For example, it is called when you issue the `analoginput('demo',1)` command. `OpenDevice` then calls the `Open` function, which communicates with the hardware (via the driver API), performs the actual initialization of the hardware, and sets up adaptor properties if necessary.

Parameters

- `DevIID` — Identifier of the interface, an instance of which is being constructed by this call. For example, interface `ImwInput`, instantiated by the `Cdemoin` class.
- `nParams` — Number of input parameters associated with object construction, not including the adaptor name. For example, for the command `analoginput('demo',0)`, the value of `nParams` is 1 because there is one parameter (the `0`) specified after the adaptor name. For the command `analogoutput('winsound')`, `nParams` is 0.
- `*Param` — An array of input parameters associated with object construction, not including the adaptor name. Using the first example given above for `nParams`, the `*Param` array contains one value of `0`. For the second example given above, the array is empty. The array type is VARIANT.
- `EngineIID` — The reference to the `IID` of the engine interface `IDaqEngine`.
- `*pEngine` — The pointer to the engine interface. The engine passes this pointer to the adaptor to enable it to call the engine functions. This parameter needs to be stored within the adaptor as a data member of one of its classes. In current adaptor implementations, it is saved as a data member of the `CmwDevice` class, which implements the `ImwDevice` interface of the adaptor.
- `*ppIDevice` — The pointer to a pointer to a newly constructed device. It is the only `[out]` parameter of this method. It is returned to the engine to enable it to call the adaptor component methods and functions. It is stored internally to the engine for the duration of the adaptor object's life.

TranslateError

Syntax

```
HRESULT  TranslateError( [in] HRESULT eCode, [out] BSTR *retVal )
```

Description

The `TranslateError` method is called by the engine to translate error codes into readable error messages. Therefore, any nonzero error code from the hardware driver API is used by the engine to display a meaningful text error message.

Parameters

- `eCode` — The numeric code of an error message.
- `*retVal` — The pointer to the error message in a text format (type BSTR).

ImwInput

The `ImwInput` interface serves as a base for the class that implements adaptor functionality specific to analog input. It publishes the following three methods to be implemented by the derived class.

Table A-4: ImwInput Methods

Method	Purpose
`GetSingle Values`	Return an array of data samples from all added channels.
`PeekData`	Called by the engine when the `peekdata` function is issued.
`Trigger`	Called by the engine for triggering a data input device.

GetSingleValues

Syntax

```
HRESULT  GetSingleValues( [out] VARIANT *Values )
```

Description

The `GetSingleValues` function is called by the engine to collect an array of data samples from all channels added to the adaptor device. The engine calls it when you issue the toolbox function `getsample`. For example, `Sample = getsample(ai)` returns a vector that contains one sample from all channels added to the analog input object `ai`. The size of the vector equals the number of added channels. The array of data points (one data sample) is returned to the engine by the single parameter of the function `GetSingleValues` function. The `*Values` parameter is a pointer to type VARIANT.

If the adaptor device is not capable of single-sample acquisition, `GetSingleValues` must return `E_NOTIMPL`.

PeekData

Syntax

```
HRESULT  PeekData( [in,out]  BUFFER_ST *pBuffer)
```

Description

The PeekData function is called by the engine when you issue the toolbox function peekdata. For example, the command data = peekdata(ai,1000) requests the most recent 1000 samples of data. If the adaptor has not collected the requested amount of data, the function returns all available data and issues an appropriate warning. The data is returned to the engine via the parameter *pBuffer, which is the pointer to the structure variable of type BUFFER_ST supplied by the engine.

The BUFFER_ST structure is defined in the file daqmex.idl and documented in Appendix C, "Engine Structures." A copy is shown here for reference.

```
typedef  struct tagBUFFER {
long Size; // in bytes
long ValidPoints; //in raw points
                  //(MATLAB samples is ValidPoints/channels)
[ref,size_is(Size)] unsigned char *ptr;
DWORD dwAdaptorData; //Reserved by engine for use by the adaptor
unsigned long Flags; //Flag values are defined in
unsigned long Reserved; //Reserved for future use by the engine
hyper StartPoint; //Count of points since start
double StartTime; //Start time of the buffer from daqenginetime
double EndTime; //End time of the buffer from daqenginetime
} BUFFER_ST;
```

Prior to calling the function, the engine assigns the requested number of data points to Size, which is a member of the BUFFER_ST structure. On return, the function fills the array *ptr, which is also a member of the BUFFER_ST structure. This way, the requested data is returned to the engine.

The PeekData member function might not necessarily be fully implemented by the adaptor. Instead it can just return E_NOTIMPL. After receiving this code as a return, the engine calls its own version of the PeekData function. To determine whether the adaptor has implemented PeekData, the engine calls PeekData NULL pBuffer after device construction. An adaptor that implements PeekData should return S_OK for this call.

Trigger

Syntax

```
HRESULT  Trigger()
```

Description

The `Trigger` function is called by the engine for triggering a data input device. Depending on the `TriggerType` property configuration, the call to the `Trigger` function can be initiated in these ways:

- Explicitly by the toolbox function `trigger`
- Internally by the engine as a response to certain conditions or signals on the inputs of the device
- Directly after the hardware is started (the toolbox `start` function is issued)

The function takes no parameters.

ImwOutput

The ImwOutput interface serves as a base for the class that implements adaptor functionality specific to analog output. It publishes the following two methods to be implemented by the derived class.

Table A-5: ImwOutput Methods

Method	Purpose
PutSingleValues	Output an array of data samples to all added channels.
Trigger	Called by the engine for triggering a data output device.

PutSingleValues

Syntax

```
HRESULT  PutSingleValues( [out] VARIANT *Values )
```

Description

The PutSingleValues function is called by the engine in response to the toolbox putsample function. For example, putsample(ao,[1 1]) outputs a single value to each of the two channels added to the analog output object ao.

PutSingleValues passes the output values through its *Values parameter, which is of type pointer to VARIANT. If the hardware is not capable of single-value output, the function returns E_NOTIMPL.

Trigger

Syntax

```
HRESULT  Trigger()
```

Description

The Trigger function is called by the engine for triggering a data output device. It works similarly to the Trigger function associated with the ImwInput interface.

ImwDIO

The ImwDIO interface serves as a base for the class that implements adaptor functionality specific to digital I/O. It publishes the following three methods to be implemented by the derived class.

Table A-6: ImwDIO Methods

Method	Purpose
ReadValues	Reads values from the specified digital ports.
WriteValues	Write values to the specified digital ports.
SetPort Directions	Called by the engine for triggering a data output device.

ReadValues

Syntax

```
HRESULT  ReadValues( [in] long NumberOfPorts, [in] long *PortList,
[out] unsigned long *Data )
```

Description

The ReadValues function reads the values from the specified digital ports of the data acquisition device. Note that the ports configured for output are not actually read. Instead, the read request returns the result of the previous write to the port, which is cached (latched in software) by the engine. This allows you to test the state of the previous write operation. Some boards allow lines on a given port to be configured separately. In this case, ReadValues still reads the whole port. However, the values obtained from the lines configured for output are meaningless, because they reflect values latched from a previous write operation.

The ReadValues function is called by the engine in response to the toolbox function getvalue.

Parameters

- NumberOfPorts — The number of ports configured for reading.

- `*PortList` — A list of ports, which are requested to be read by `ReadValues`.
- `*Data` — An array of data values returned from the digital I/O ports specified in the `*PortList` parameter.

WriteValues

Syntax

```
HRESULT  WriteValues( [in] long NumberOfPorts, [in] long *PortList,
[in] unsigned long *Data, [in] unsigned long *Mask  )
```

Description

The `WriteValues` function outputs values to the specified digital ports of the data acquisition board. It is called by the engine when you issue the toolbox function `putsample`.

Parameters

- `NumberOfPorts` — The number of ports configured for output.
- `*PortList` — A list of ports to which the function writes.
- `*Data` — An array of data values for output to the ports. The order of the values must correspond to the order of the ports in `*PortList`.
- `*Mask` — An array of bit masks for the lines of the specified ports to be written to. This parameter is used only if the ports are line configurable. For port-configurable devices `*Mask` is ignored.

SetPortDirections

Syntax

```
HRESULT  SetPortDirections( [in] long Port, [in] unsigned long
DirectionValues )
```

Description

This `SetPortDirections` function is used by the engine to set the directions of signal lines of digital I/O ports. It is called by the engine when the `direction` property is changed via the `set` function.

Parameters

- `Port` — The ID of the port to be modified by the function.
- `DirectionValues` — A bit mask of direction values for the lines of the port. A 1 signifies that the line is set for output. If the port is port configurable, the only admissible values are 0xFF for output and 0 for input.

B

Engine Interface Reference

IPropRoot

The methods of the IPropRoot interface allow you to configure your device properties and channel properties. For example, there are methods for setting a property's default value, for setting a property's current value, and for setting the range of values that a property can be set to. Property methods are generally called from within

- The adaptor's Open method, when creating an adaptor-specific property
- The adaptor's Open method, when changing the characteristics of an existing property
- The adaptor's SetProperty method, when modifying one device property affects the property characteristics of another property
- The adaptor's SetChannelProperty method, when modifying one channel property affects the property characteristics of another property

Each property has its own set of interfaces. To change the characteristics of a property, you must have a pointer to the IPropRoot interface for that property. Any IPropRoot derived interface pointer can be obtained by

- Creating a property with the CreateProperty method of the IPropContainer interface
- Getting an existing property with the GetMemberInterface method of the IPropContainer interface

The CreateProperty and GetMemberInterface methods both return a pointer to the requested interface for the property you are creating or getting. When a property value is modified, the data acquisition engine calls the adaptor's SetProperty or SetChannelProperty methods.

The IPropRoot interface methods are described below.

GetRange

Syntax

```
HRESULT GetRange(VARIANT *min, VARIANT *max)
```

Output Parameters

`min`	`Variant*`
max	`Variant*`

Description

The `GetRange` method returns the range (minimum and maximum) values that a property can be set to. For example, the following code fragment gets the range of the `SampleRate` property.

```
CComVariant min;
CComVariant max;
iprop->GetRange(&min, &max);
```

The `GetMemberInterface` method of the `IPropContainer` interface can be used to return a pointer to the `IPropRoot` interface for the `SampleRate` property.

See Also

`IPropRoot` interface: `SetRange`

SetRange

Syntax

```
HRESULT SetRange(VARIANT *min, VARIANT *max)
```

Input Parameters

`min`	`Variant*`
max	`Variant*`

Description

The `SetRange` method allows you to set the current minimum and maximum values for a given property. For example, the following code fragment sets the `SampleRate` property range from `8000` to `44100`. This range is defined within the `Open` method of the adaptor.

```
CComVariant min(8000.0);
CComVariant max(44100.0);
Prop -> SetRange(&min, &max);
```

See Also

`IPropRoot` interface: `GetRange`

GetType

Syntax

```
HRESULT GetType(long Type)
```

Output Parameters

`Type`	`long`

Description

The `GetType` method returns the VARTYPE type for the given property. VARTYPEs are the enumerated data types supported by VARIANT.

The lower 16 bits of `Type` contains the actual VARENUM. The upper 16 bits are used for special case data types. In general, any type not contained in the VARENUM structure is a special data type used by the engine and should not be accessed externally. See VARENUM in the Windows header file `wtypes.h` for a listing of valid types.

get_DefaultValue

Syntax

```
HRESULT get_DefaultValue(VARIANT *Value)
```

Output Parameters

```
Value       Variant*
```

Description

The `get_DefaultValue` method returns the default value for the property. For example, to determine the default value for the `TriggerType` property.

```
VARIANT Value;
prop->get_DefaultValue(&Value);
```

See Also

`IDaqEngine` interface: `GetProperty`

put_DefaultValue

Syntax

```
HRESULT put_DefaultValue(VARIANT *Value)
```

Input Parameters

```
Value       Variant*
```

Description

The `put_DefaultValue` method allows you to set the default value for a property. For example, the following code fragment sets the default value for the `ChannelSkewMode` property to `None`.

```
prop->ClearEnumValues();
prop->AddMappedEnumValue(0, L"None");
prop->put_DefaultValue(CComVariant(0));
```

get_IsHidden

Syntax

```
HRESULT get_IsHidden(BOOL *IsHidden)
```

Output Parameters

```
IsHidden    BOOL*
```

Description

The `get_IsHidden` method determines whether a property is hidden from you. Hidden properties are not displayed with the `get` and `set` displays. A hidden property's value can be obtained or modified only if you pass the name of the property to the `get` or `set` toolbox function. A value of `false` indicates that the property is not hidden and a value of `true` indicates that the property is hidden.

For example, the following code fragment determines whether the `SampleRate` property is hidden.

```
BOOL IsHidden;
prop->get_IsHidden(&IsHidden);
```

put_IsHidden

Syntax

```
HRESULT put_IsHidden(BOOL IsHidden)
```

Input Parameters

```
IsHidden    BOOL
```

Description

The `put_IsHidden` method allows you to hide a property from users. A hidden property is not displayed with the `get` and `set` displays. A hidden property's value can be obtained or modified only if you pass the name of the property to the `get` or `set` toolbox function. A value of `false` indicates that the property is not hidden and a value of `true` indicates that the property is hidden.

For example, to create a hidden property called `MyHiddenProperty`:

```
pRoot->CreateProperty(L"MyHiddenProperty", NULL, &prop);
prop->put_IsHidden(true);
```

See Also

`IPropContainer` interface: `CreateProperty`

get_IsReadonlyRunning

Syntax

```
HRESULT get_IsReadonlyRunning( BOOL *pVal)
```

Output Parameters

```
pVal        BOOL*
```

Description

The `get_IsReadonlyRunning` method determines whether a property's value can be modified while the object is running. For example, the `SampleRate` property cannot be modified while the object is running. However, the `Tag` property can be modified whether the object is running or not. A value of `false` indicates that the property can be modified while the object is running, while a value of `true` indicates that the property cannot be modified while the object is running.

For example, the following code fragment determines whether the `SampleRate` property can be modified while the object is running.

```
bool pVal;
engine->GetProperty(L"SampleRate", &prop);
prop->get_IsReadonlyRunning(&pVal);
```

See Also

`IDaqEngine` interface: `GetProperty`

put_IsReadonlyRunning

Syntax

```
HRESULT put_IsReadonlyRunning( BOOL pVal)
```

Input Parameters

```
pVal        BOOL
```

Description

The `put_IsReadonlyRunning` method allows you to configure a property so that it cannot be set while the object is running. A value of `false` indicates that the property can be modified while the object is running, while a value of `true` indicates that the property cannot be modified while the object is running.

For example, the `StandardSampleRates` property cannot be modified while the object is running.

```
pRoot->CreateProperty(L"StandardSampleRates", NULL, &prop);
prop->put_IsReadonlyRunning(true);
```

If you try to set the property while the object is running, you receive the following error:

```
set(ai, 'StandardSampleRates', 'off')
??? Error using ==> daqdevice/set
The property: 'StandardSampleRates' is read-only while running.
```

The `ReadonlyRunning` value should not be modified for standard engine properties. Changing this attribute can result in unpredictable behavior.

See Also

`IPropContainer` interface: `CreateProperty`

get_IsReadonly

Syntax

```
HRESULT get_IsReadonly( BOOL *pVal)
```

Output Parameters

```
pVal        BOOL*
```

Description

The `get_IsReadonly` method determines whether a property's value can be modified by you. For example, the `Type` property cannot be modified. However, the `Tag` property can be modified. A value of `false` indicates that the property can be modified, while a value of `true` indicates that the property cannot be modified and is read only.

For example, the following code fragment determines whether the `ChannelSkewMode` property is read only.

```
bool pVal;
prop->get_IsReadonly(&pVal);
```

See Also

`IDaqEngine` interface: `GetProperty`

put_IsReadonly

Syntax

```
HRESULT put_IsReadonly( BOOL *pVal)
```

Input Parameters

```
pVal        BOOL
```

Description

The `put_IsReadonly` method allows you to configure a property so that it cannot be modified by you. A value of `false` indicates that the property can be

modified, while a value of `true` indicates that the property cannot be modified and is read only.

For example, the following code fragment creates a property called `ReadOnlyProperty` that cannot be modified.

```
prop->put_IsReadonly(true);
```

If you try to set the property, you receive the following error:

```
set(ai, 'ReadOnlyProperty', 4)
??? Error using ==> daqdevice/set
The property: 'ReadOnlyProperty' is read-only.
```

This attribute should not be set to `true` for standard engine properties.

See Also

`IDaqEngine` interface: `IpropContainer->CreateProperty`

get_User

Syntax

```
HRESULT get_User(long *User)
```

Output Parameters

```
User        long*
```

Description

The `get_User` method gets the User for a property. When certain properties are modified by you, the adaptor needs to configure the hardware appropriately. The adaptor is notified of property value changes for any property that the adaptor has registered with the data acquisition engine. The adaptor registers a property by passing to the engine the address of a local data member for the property. The address of a local data member can be stored in the User value.

When a property that has been registered by the adaptor is set by you, the data acquisition engine calls the adaptor's `SetProperty` or `SetChannelProperty` method. The User value of the property is then passed to the method.

See Also

`IPropRoot` interface: `put_User`

put_User

Syntax

```
HRESULT put_User(long newVal)
```

Input Parameters

`newVal`	`long`

Description

The `put_User` method sets the User for a property. When certain properties are modified by you, the adaptor needs to configure the hardware appropriately. The adaptor is notified of property value changes for any property that the adaptor has registered with the data acquisition engine. The adaptor registers a property by passing to the engine the address of a local data member for the property. The address of a local data member is called a User.

For example, the following code registers the `SampleRate` property:

```
engine->GetProperty(L"SampleRate", &prop);
prop->put_User((long)&_sampleRate);
```

where `_sampleRate` is a variable that contains the current value of the `SampleRate` property. To set the `SampleRate` property to a new value

```
set(obj, 'SampleRate', 11025);
```

When a property that has been registered by the adaptor is set by you, the data acquisition engine calls the adaptor's `SetProperty` or `SetChannelProperty` method. The User value of the property is then passed to the method.

See Also

`IDaqEngine` interface: `GetProperty`
`IPropRoot` interface: `put_User`

get_Name

Syntax

```
HRESULT get_Name (BSTR *pVal)
```

Output Parameters

```
pVal        BSTR*
```

Description

The `get_Name` method allows you to determine the name of the property. This can be useful when you are creating an error or warning message.

put_Name

Syntax

```
HRESULT put_Name (BSTR pVal)
```

Input Parameters

```
pVal        BSTR
```

Description

The `put_Name` method sets the name of a property.

IsValidValue

Syntax

```
HRESULT IsValidValue([in] VARIANTREF value)
```

Input Parameters

`Value`	Value to check.

Description

The `IsValidValue` function returns an error if the value is not valid, and returns `S_OK` if the value is valid.

IDaqEngine

The IDaqEngine interface methods provide a variety of different engine services. For example, there are methods for creating channel properties, posting events, and buffer management. IDaqEngine methods are generally called from within

- The adaptor's Open method for creating an adaptor-specific property
- The adaptor's Open, SetProperty, or SetChannelProperty methods for obtaining a pointer to an IPropContainer or IPropValue interface for the property being modified
- The adaptor's ChildChange method for obtaining an IPropContainer interface for an existing channel
- Any adaptor method for posting warnings
- The adaptor routines for obtaining buffers of data to output or to fill with acquired data

To execute an IDaqEngine interface method, you must have a pointer to the IDaqEngine interface. The first step in a MATLAB data acquisition session is to create the analog input, analog output, or digital I/O object with the analoginput, analogoutput, or digitalio constructors. These MATLAB functions call the data acquisition engine, which then calls the adaptor's Open method. The IDaqEngine interface pointer is passed from the data acquisition engine to the adaptor's Open method. The IDaqEngine interface pointer should be stored as a local variable so that IDaqEngine interface methods can be executed from other methods within the adaptor. The methods are described below.

DaqEvent

Syntax

```
HRESULT DaqEvent(DWORD event, double time, __int64  sample, BSTR
Message)
```

Input Parameters

`Event`	One of the enumerated values for event type.
`Time`	The engine time of the event, or –1 to tell the engine to check
`Sample`	The sample at which the event occurred, or –1 to state unknown
`Message`	A description of the event. Only used for errors and user events.

Description

The `DaqEvent` method notifies the data acquisition engine that an event of interest occurred. The `Event` input argument is defined as the enum in `daqmexstructs.h`, and can be set to one of the following values:

```
typedef enum {EVENT_START,EVENT_STOP,EVENT_TRIGGER,
EVENT_ERR,EVENT_OVERRANGE,EVENT_DATAMISSED,
EVENT_SAMPLESACQUIRED,EVENT_SAMPLESOUTPUT,EVENT_USER}
EventTypes;
```

The time of the event, the last sample acquired at the time of the event, and an additional message can also be sent to the data acquisition engine. This information is then posted in the object's `EventLog` property. If a time value of -1 is sent to the data acquisition engine, the engine calculates the time of the event.

For example, once the analog output object has finished outputting data, the adaptor is responsible for posting the `Stop` event.

```
engine->DaqEvent(STOP, -1, 8000, NULL);
```

From the preceding `DaqEvent` method call, the object's `EventLog` property would contain the following structure:

```
h = ao.EventLog;
h(3).Type
ans =
```

```
Stop

h(3).Data
ans =
AbsTime: [1999 4 10 13 24 10.3400]
RelSample: 8000
```

GetBuffer

Syntax

```
HRESULT GetBuffer (long Timeout, BUFFER_ST **Buffer)
```

Input Parameters

`TimeOut` Time in milliseconds to wait for a buffer.

Output Parameters

`Buffer` A pointer to the returned buffer.

```
typedef  struct tagBUFFER {
long Size; // In bytes
long ValidPoints; // In raw points
                  // (MATLAB samples is ValidPoints/channels)
unsigned char *ptr;
DWORD dwAdaptorData; // Reserved by the engine for use by adaptor
unsigned long Flags; // Flag values are defined in
unsigned long Reserved; // Reserved for future use by the engine
hyper StartPoint; // Count of points since start
double StartTime; // Time of the start of the buffer from GetTime
double EndTime; // Time of the end of the buffer from GetTime
} BUFFER_ST;
```

Description

The `GetBuffer` method is used to pass data between the adaptor and the engine. For analog input, `GetBuffer` passes an empty buffer from the data acquisition engine to the adaptor. As data is acquired from the hardware, the adaptor fills the empty buffer. When the buffer is full, the adaptor can send the

acquired data to the data acquisition engine using the IDaqEngine interface PutBuffer method. You can then obtain the acquired data with the getdata toolbox function.

For analog output, GetBuffer passes a buffer full of data from the data acquisition engine to the adaptor. You must first queue the data in the data acquisition engine with the putdata toolbox function. The adaptor sends the buffer of data to the hardware. When the adaptor has finished outputting the buffer of data, the empty buffer is returned to the data acquisition engine with the IDaqEngine interface PutBuffer method.

BUFFER_ST is defined in daqmexstructs.h.

See Also

IDaqEngine interface: PutBuffer

GetBufferingConfig

Syntax

```
HRESULT GetBufferingConfig(long *BufferSizeSamples, long *Num-
Buffers)
```

Output Parameters

BufferSize Samples	Current buffer size in samples. It does not change after start.
NumBuffers	Current number of buffers. If the BufferingConfigMode property is set to auto, this value increases as needed. If the property is set to manual, the value is fixed from start.

Description

The GetBufferingConfig method is used to determine the data acquisition engine's setting for the object's BufferingConfig property. The first element of the vector (BufferSizeSamples) specifies the block size, while the second element of the vector (NumBuffers) specifies the number of blocks.

Memory can be allocated either automatically, by the engine, or manually, depending on the value of the BufferingMode property. If BufferingMode is Auto, the BufferingConfig property values are automatically set by the data

acquisition engine. If BufferingMode is Manual, you must manually set the BufferingConfig values. If you change the BufferingConfig values, BufferingMode is automatically set to Manual.

When memory is automatically allocated by the engine, the block-size value depends on the sampling rate and is typically a binary number. The number of blocks is initially set to a value of 30 but can dynamically increase to accommodate the needs of the engine. In most cases, the number of blocks used results in a per-channel memory that is somewhat greater than the SamplesPerTrigger value. When you manually allocate memory, the number of blocks is not dynamic, and you must take care to ensure that there is sufficient memory to store the acquired data.

GetTime

Syntax

```
HRESULT GetTime(double *Time)
```

Output Parameters

Time	The current engine time.

Description

The engine time is defined as the number of seconds since the engine was loaded. It has a better resolution than the standard system clock.

The GetTime method is useful when calling the DaqEvent method (which takes current time as an input argument) for posting an event to the object's EventLog property.

The following code fragment posts an error event when an underrun condition occurs to the object's EventLog property:

```
double time;
engine->GetTime(&time);
engine->DaqEvent(ERR, time, _samplesOutput, L"Underrun");
```

From the preceding DaqEvent method call, the object's EventLog property would contain the following structure:

```
h = ao.EventLog;
h(2).Type
ans =
Error

h(2).Data
ans =
AbsTime: [1999 4 10 13 24 10.3400]
RelSample: 8000
Message: Underrun.
```

See Also

IDaqEngine interface: DaqEvent

PutBuffer

Syntax

```
HRESULT PutBuffer(BUFFER_ST *Buffer)
```

Input Parameters

```
Buffer      BUFFER_ST*

  typedef  struct tagBUFFER {
  long Size; // In bytes
  long ValidPoints; // In raw points
                    // (MATLAB samples is ValidPoints/channels)
  unsigned char *ptr;
  DWORD dwAdaptorData; // Reserved by the engine for use by adaptor
  unsigned long Flags; // Flag values are defined in
  unsigned long Reserved; // Reserved for future use by the engine
  hyper StartPoint; // Count of points since start
  double StartTime; // Time of the start of the buffer from GetTime
  double EndTime; // Time of the end of the buffer from GetTime
  } BUFFER_ST;
```

Description

The PutBuffer method is used to pass data between the adaptor and the engine. For analog input, PutBuffer passes a buffer of acquired data from the adaptor to the data acquisition engine. The original buffer that the adaptor filled with acquired data from the hardware was obtained from the data acquisition engine with the IDaqEngine interface GetBuffer method. You can obtain the acquired data from the data acquisition engine with the getdata toolbox function.

For analog output, PutBuffer passes an empty buffer of data from the adaptor to the data acquisition engine. The buffer originally contained data that was queued in the data acquisition engine with the toolbox putdata function. Once the adaptor finishes outputting the data to the hardware, the empty buffer is returned to the engine with the PutBuffer method.

BUFFER_ST is defined in daqmexstructs.h.

See Also

IDaqEngine interface: GetBuffer

WarningMessage

Syntax

```
HRESULT WarningMessage(BSTR Message)
```

Input Parameters

Message	The warning message to show to the user.

Description

The WarningMessage method is used by the adaptor to post a warning to the MATLAB Command Window. For example:

```
engine->WarningMessage(CComBSTR(L"A warning just occurred."));
```

The use of a simple wide string in this call in some places in the supplied adaptors should be considered a bug. If a string that is not a BSTR is passed to this function from a separate apartment, a segfault occurs.

PutInputData

Syntax

```
HRESULT PutInputData([in] long Timeout,[in] BUFFER_ST *Buffer);
```

Input Parameters

`Timeout`	Time in milliseconds to wait for a new buffer. When the `BufferingConfigMode` property is set to `auto`, this routine never needs to wait, as long as memory is available.
`Buffer`	A pointer to a buffer header for the data.

Description

The `PutInputData` method is a simplified way for giving information to the engine, and is an alternative to using `GetBuffer` and `PutBuffer` for an input device. In most cases when creating a DLL-based adaptor, using the `GetBuffer`–`PutBuffer` sequence is preferred because it does not require copying the data. This function returns success codes containing the buffer flags that were set in the engine buffer that was filled. The size of the buffer put must be less than or equal to the current engine buffer size.

The data members of `Buffer` must be filled in the same way as they are for `PutBuffer`.

Engine pseudocode implementation of PutInputData is shown below.

```
PutInputData(long Timeout, BUFFER_ST *Buffer)
{
   GetBuffer(Timeout,&engineBuffer);
    if (succeded)
    {
   copy Buffer to engineBuffer
   Call PutBuffer on engineBuffer
   Return status if not success or flags otherwise;
    }
else return error
}
```

GetOutputData

Syntax

```
HRESULT GetOutputData([in] long Timeout,[out] BUFFER_ST *Buffer);
```

Input Parameters

`Timeout` Time in milliseconds to wait for a new buffer. When the `BufferingConfigMode` property is set to `auto`, this routine never needs to wait, as long as memory is available.

Output Parameters

`Buffer` The address of a buffer header for the data.

Description

The `GetOutputData` method is used by remote adaptors to retrieve data for output. It is an alternative to using `GetBuffer` and `PutBuffer` for an output device. In most cases when creating a DLL based adaptor, using the `GetBuffer`–`PutBuffer` sequence is preferred because it does not require copying the data. The size of the buffer put must be greater than or equal to the current engine buffer size.

IDaqEnum

The `IDaqEnum` interface is not currently implemented. However, the `ClearEnumValues` and `RemoveEnumValue` methods are implemented, and they are useful for the `IDaqMappedEnum` class.

In the future, `IDaqEnum` will be used to implement an enumerated list of settable property values for integer and floating-point value types. The `IDaqEnum` methods are described below.

AddEnumValues

Not yet implemented.

Syntax

```
HRESULT AddEnumValues(VARIANT* values)
```

Input Parameters

`Value` VARIANT*

Description

The `AddEnumValues` method is used to add enumerated values to a property.

To add more than one value, place the desired values in a SAFEARRAY and pass the SAFEARRAY in the VARIANT.

ClearEnumValues

Syntax

```
HRESULT ClearEnumValues()
```

Parameters

None

Description

The `ClearEnumValues` method clears the property's current enumerated list. This allows you to create a completely new enumerated list for the specified property.

For example, initially the ChannelSkewMode property can be set to None, Equisample, Manual, or Minimum. However, the sound card's ChannelSkewMode property can only be set to None. Therefore, from within the adaptor's Open method, the ChannelSkewMode property is modified to have only one possible setting of None.

```
hRes=GetProperty(L"ChannelSkewMode", &prop);
if (!(SUCCEEDED(hRes))) return hRes;
prop->ClearEnumValues();
prop->AddMappedEnumValue(CHAN_SKEW_NONE,L"None");
prop->put_DefaultValue(CComVariant(CHAN_SKEW_NONE));
prop->put_Value(CComVariant(CHAN_SKEW_NONE));
prop.Release();
```

Note that the same end result can be obtained with the RemoveEnumValue method of the IDaqEnum interface.

See Also

RemoveEnumValue, IDaqMappedEnum::AddMappedEnumValue

RemoveEnumValue

Syntax

```
HRESULT RemoveEnumValue(BSTR StringValue)
```

Input Parameters

StringValue	BSTR

Description

The RemoveEnumValue method allows you to remove specific enumerated values from a property.

For example, initially the ChannelSkewMode property can be set to None, Equisample, Manual, or Minimum. However, the sound card's ChannelSkewMode property can only be set to None. Therefore, from within the adaptor's Open method, the ChannelSkewMode property is modified to have only one possible setting of None. The GetProperty method of the IDaqEngine interface returns a pointer, prop, to the IProp interface for the ChannelSkewMode property.

```
GetProperty(L"ChannelSkewMode", &prop);
prop->RemoveEnumValue(L"Equisample");
prop->RemoveEnumValue(L"Manual");
prop->RemoveEnumValue(L"Minimum");
```

Note that the same end result can be obtained by using the `ClearEnumValues` and `AddMappedEnumValue` methods of the `IProp` interface.

See Also

`IDaqEngine` interface: `GetProperty`
`IProp` interface: `AddMappedEnumValue`, `ClearEnumValues`

EnumValues

Not yet implemented.

Output Parameters

`EnumVARIANT  IEnumVARIANT**`

Purpose

This function returns a standard `IEnumVARIANT` interface to allow the device to enumerate through all possible values.

IDaqMappedEnum

A MappedEnum value represents a mapping between string and integer values. The three methods are AddMappedEnumValue, FindString, and FindValue. When a MappedEnum value is used, the engine takes care of the translation from string to numeric value and passes the numeric value to the adaptor when necessary. The IDaqMappedEnum methods are described below.

AddMappedEnumValue

Syntax

```
HRESULT AddMappedEnumValue(long Value, BSTR StringValue)
```

Input Parameters

Value	long
StringValue	LPCOLESTR

Description

The AddMappedEnumValue method is used to add enumerated property settings to a property. Enumerated values are represented as either a string or an integer value.

The following code fragment modifies an existing property, ChannelSkewMode, to have adaptor-specific enumerated settings:

```
engine->GetProperty(L"ChannelSkewMode", &prop);
prop->ClearEnumValues();
prop->AddMappedEnumValue(0,CComBSTR(L"None"));
```

The GetProperty method of the IDaqEngine interface returns a pointer, prop, to the IProp interface for the ChannelSkewMode property.

See Also

IDaqEngine interface: GetProperty
IDaqEnum interface: ClearEnumValues, RemoveEnumValue

FindString

Syntax

```
HRESULT FindString ([in] long Value,[out] BSTR *StringValue);
```

Input Parameters

`Value`	Numeric value to be looked up.

Output Parameters

`StringValue`	The resulting value of the translation. The caller must free the BSTR `StringValue` when done with it.

Description

The `FindString` method translates the integer value to the string. If the value is not a member of the enumerated set, this function returns the string “Enum Not Found”.

FindValue

Syntax

```
HRESULT FindValue ([in,string] wchar_t *StringValue,[out] long
*value);
```

Input Parameters

`String`	String to be looked up.

Output Parameters

`IntegerValue`	The resulting value of the translation.

Description

The `FindValue` method translates the string into the integer value. In the event that the string cannot be found in the enumerated list or the string is not unique, this function returns a dispatch error with extended information in an `IErrorInfo` object. To display the message, return the error to the engine.

IPropValue

The IPropValue interface is the preferred way to retrieve the value of a given property. These methods are duplicated in IProp. Note that the IProp interface is going to be obsolete in the future. The IPropValue methods are described below.

get_Value

Syntax

```
HRESULT get_Value(VARIANT *pVal)
```

Output Parameters

pVal	VARIANT*

Description

The get_Value method returns the current value of the property. get_Value is generally used within the SetProperty or SetChannelProperty method when a certain setting of a property affects the value of another property.

For example, the following code fragment modifies the SampleRate property based on the StandardSampleRates property value.

```
CRemoteProp IStdSrProp;
IStdSrProp.Attach(GetPropRoot(),L"StandardSampleRates");
variant_t standardSR;
IStdSrProp ->get_Value(&standardSR);

switch ((bool)standardSR){
case false:
// StandardSampleRate is off. Do something to the SampleRate prop.
case true:
// StandardSampleRate is on. Do something to the SampleRate prop.
}
```

See Also

IDaqEngine interface: GetProperty

put_Value

Syntax

```
HRESULT put_Value(VARIANTREF newVal)
```

Input Parameters

`newVal`	The new value of the property.

Description

The `put_Value` method allows you to set the current value of a property. This method is generally called within the adaptor's `Open` method when configuring properties at initialization, and within the adaptor's `SetProperty` and `SetChannelProperty` methods when the value of one property affects the value of another property.

The following code fragment sets the current value of the `SampleRate` property to `8000`.

```
CComPtr<IPropValue>
GetProperty(L"SampleRate", &prop);
prop->put_Value(CcomVariant(8000L));
```

See Also

`IDaqEngine` interface: `GetProperty`

IPropContainer

The IPropContainer interface is the main interface for property containers. The property containers within the Data Acquisition Toolbox are structures of properties that relate to

- A channel group
- A channel
- The structure returned by the daqhwinfo toolbox function

The methods of the IPropContainer interface allow you to add device-specific properties to property containers and modify property values. IPropContainer methods are generally called from the

- AdaptorInfo method for configuring the properties returned by daqhwinfo
- Open method for creating device-specific properties
- SetProperty method, when modifying a device property affects the property value of another property
- SetChannelProperty method, when modifying a channel property affects the current value of another property
- ChildChange method, when adding or deleting a channel

Each property structure defined above has its own IPropContainer interface. In order to change the characteristics of the property structure, you must have a pointer to the IPropContainer interface for that property structure. The IPropContainer interface pointer can be obtained from

- The engine interface using QueryInterface.
- Another container using GetMemberInterface. For example, daqhwinfo is retrieved this way.
- The GetChannelContainer method of the IChannelList interface.
- Within the AdaptorInfo method.
- Within the ChildChange method.

The GetMember and GetChannelContainer methods both return a pointer to the IPropContainer interface for the interface that you are getting. The following commands return a pointer to the IPropContainer interface for a channel group.

```
CComQIPtr<IPropContainer, &__uuidof(IPropContainer)>  pCont;
engine->GetProperty(NULL,&pCont);
```

The following commands return a pointer to the `IPropContainer` interface for the structure returned by the `daqhwinfo` toolbox function.

```
CComQIPtr<IPropContainer, &__uuidof(IPropContainer)>  pCont
engine->GetProperty(L"daqhwinfo",&pCont);
```

The following commands return a pointer to the `IPropContainer` interface for the first channel.

```
CComQIPtr<IPropContainer, &__uuidof(IPropContainer)>  pCont;
hRes = engine->GetChannelContainer(0, &pCont);
```

The `IPropContainer` methods are described below.

CreateProperty

Syntax

```
HRESULT CreateProperty ([in,string] LPCOLESTR Name,[in] VARIANT
*InitialValue,[in] REFIID RequestedIID, [out,iid_is(ReuestedIID0]
void **NewProp)
```

Input Parameters

`Name`	The name of the property being created.
`InitialValue`	The property's initial value. The data type of the property is taken from the data type of this value.
`RequestedIID`	The IID of the property interface to be created and returned in `NewProp`.

Output Parameters

`NewProp`	The requested interface to the property that was created.

Description

The `CreateProperty` method adds a property to a property structure. Note that the `CmwDevice` class implements helper functions `CreateProperty` and `CreateChannelProperty` that can be called instead of this function.

```
template <class T>
HRESULT CreateProperty(LPCWSTR name,  VARIANT *value, T** prop)
{
RETURN_HRESULT(_EnginePropRoot->CreateProperty(name,value,__uuid
of(T),(void**)prop)); return S_OK;
}
```

`CreateProperty` is typically called from within the `Open` method. The first input argument passed to `CreateProperty` contains the name of the property that is being created. The second input argument contains the initial value of the property and is also the default value. `CreateProperty` returns a pointer to the requested interface. This allows access to the methods that are used for configuring the property.

For example, the following code fragment creates a property called `MyProperty` that has a default value of `0` and can range from `0` to `100`.

```
CreateProperty(L"MyProperty", &CComVariant(0L), &NewProp);
NewProp->setRange(&CComVariant(0L), &CComVariant(100L));
```

The following code fragment creates a property called `MyProperty2` that has a default value of `On` and can be set to either `On` or `Off`. `MyProperty2` cannot be modified while the object is running.

```
CreateProperty(L"MyProperty2", &CComVariant(true), &NewProp);
NewProp->put_IsReadonlyRunning(true);
```

Note that by passing an initial value of true, a Boolean property is created. A Boolean property can be set to either `On` or `Off`, where a value of `true` maps to a property value of `On` and an integer value of `1`, and a value of `false` maps to a property value of `Off` and an integer value of `0`.

The following example creates a property called `MyProperty3` that can be set to `Celsius`, `Fahrenheit`, or `Kelvin`. The property has a default value of `Fahrenheit` and a current value of `Celsius`.

```
CreateProperty(L"MyProperty3", NULL, &NewProp);
NewProp->AddMappedEnumValue(0,CComBSTR(L"Celsius"));
NewProp->AddMappedEnumValue(1,CComBSTR(L"Fahrenheit"));
NewProp->AddMappedEnumValue(2,CComBSTR(L"Kelvin"));
NewProp->put_DefaultValue(CComVariant(1));
NewProp->put_Value(CComVariant(0));
```

Note that by passing an initial value of NULL, you create an enumerated property. The interface method `AddMappedEnumValue` is used to add enumerated values to the property.

See Also

`IDaqMappedEnum` interface: `AddMappedEnumValue`, `put_IsReadonlyRunning`, `put_DefaultValue`, `put_Value`, `setRange`

GetMemberInterface

Syntax

```
HRESULT GetMemberInterface(LPCOLESTR MemberName, REFIID
RequestedInterface, void **Interface)
```

Input Parameters

`MemberName`	The property name.
`Requested Interface`	The interface that is used.

Output Parameters

`Interface`	The requested interface if found, or NULL otherwise.

Description

The `GetMemberInterface` method returns an interface pointer to an existing property. Note that the `CmwDevice` methods `GetProperty` and `GetChannelProperty` supply a simplified interface to this function. The code for `GetProperty` is given here as an example of the use of this function.

```
template <class T>
HRESULT GetProperty(LPCWSTR name, T** prop)
{
RETURN_HRESULT(_EnginePropRoot->GetMemberInterface(
name,__uuidof(T),(void**)prop)); return S_OK;
}
```

put_MemberValue

Syntax

```
HRESULT put_MemberValue(LPCOLESTR MemberName, VARIANTREF newVal)
```

Input Parameters

`MemberName`	The property name.
`newVal`	A reference to the new value.

Description

The `put_MemberValue` method sets the value of a property. `put_MemberValue` is typically used when setting the value of a property contained by the `daqhwinfo` property structure. The first input argument contains the name of the property that is being modified. The second input argument contains the new value for the property being modified. This function is a shortcut for retrieving an `IPropValue` interface for the given property and accessing its value.

For example, the following command sets the `AdaptorName` field of the `daqhwinfo` structure to `winsound`.

```
GetHwInfo()->put_MemberValue(L"adaptorname",CComVariant(L"winsou
nd"));
```

The following command sets the `TotalChannels` field of the `daqhwinfo` structure to 2.

```
GetHwInfo()->put_MemberValue(L"totalchannels",CComVariant(2L));
```

get_MemberValue

Syntax

```
HRESULT get_MemberValue(LPCOLESTR MemberName, VARIANT *pVal)
```

Input Parameters

`MemberName`	The requested property name.

Output Parameters

`pVal`	The property's current value.

Description

The `get_MemberValue` method returns the current value of a property. This function is a shortcut for retrieving an `IPropValue` interface for the given property and accessing its value.

For the sound card, up to two channels can be added. If one channel is added, sound is recorded and played in mono. If two channels are added, sound is recorded and played in stereo. By default, if one channel is added to the `winsound` object, the channel is assigned a `ChannelName` of `Mono`. If a second channel is added, and you have not renamed the first channel, the two channels have the names `Left` and `Right`.

The following code fragment determines the first channel's `ChannelName` property value:

```
CComVariant pVal;
CComQIPtr<IPropContainer>  pCont;
GetChannelContainer(0, &pCont);
pCont->get_MemberValue(L"channelname", &pVal);
```

Note: engine is a pointer to the `IDaqEngine` interface that was obtained from the adaptor's `Open` method. The `IPropContainer` interface for the first channel is returned to `pCont`.

See Also

`IChannelList` interface: `GetChannelContainer`

IChannel

`IChannel` is the interface to an individual channel. An `IChannel` interface is acquired by calling `IChannelList::GetChannelContainer`. An `IChannel` interface should not be stored for a device that is not running. Deleting a channel invalidates the `IChannel` interface for that channel. An `IChannel` object implements all methods of `IPropContainer` plus the methods described below.

get_PropValue

Syntax

```
HRESULT get_PropValue(REFIID MemberIID, IPropRoot* Member, [out,
retval] VARIANT *pVal)
```

Input Parameters

`MemberIID`	IID of the interface passed in as `Member`. It must be derived from `IPropRoot`.
`Member`	An interface to the channel property whose value is desired.

Output Parameters

`pVal`	The value of the member property for the given channel.

Description

The `get_PropValue` function retrieves the value of a property for a given channel. This function is more efficient than using `get_MemberValue` because it does not require looking up the string name of the property.

put_PropValue

Syntax

```
HRESULT put_PropValue(REFIID riid, IPropRoot* Member, VARIANTREF
NewVal)
```

Input Parameters

`MemberIID`	IID of the interface passed in as `Member`. It must be derived from `IPropRoot`.
`Member`	An interface to the channel property whose value is set.

Output Parameters

`NewVal`	The new value for the property.

Description

The `put_PropValue` function sets the value of a property on a given channel. This function is more efficient than using `set_MemberValue` because it does not require looking up the string name of the property.

UnitsToBinary

Syntax

```
HRESULT UnitsToBinary([in] double UnitsVal, [out] VARIANT *pVal)
```

Input Parameters

`UnitsVal`	The value to convert in the user's units.

Output Parameters

`pVal`	The converted value in native data type.

Description

This function performs the conversion based on the current settings of the given channel.

BinaryToUnits

Syntax

```
HRESULT BinaryToUnits([in] VARIANTREF BinaryVal,[out] double
*UnitsVal)
```

Input Parameters

`BinaryValue`	The value to convert stored in the native data type of the device.

Output Parameters

`UnitsVal`	The converted value in units.

Description

This function performs the conversion based on the current settings of the given channel.

IChannelList

The MemberValue property inherited from IPropContainer is not implemented for IChannelList because each member does not have one unique value. The IChannelList methods are described below.

GetChannelContainer

Syntax

```
HRESULT GetChannelContainer (long index, REFIID requestedInterface,
void **Container)
```

Input Parameters

index	The channel requested.
requested Interface	Usually &__uuidof(IChannel).

Output Parameters

Container	The container for the requested channel.

Description

The GetChannelContainer method is used to obtain an IPropContainer interface pointer to a specific channel. The index argument specifies the index of the channel (zero-based). An index of 0 specifies the first channel, an index of 1 specifies the second channel, and so on.

The following command fragment returns an IPropContainer interface pointer, Container, for the second channel in the channel array.

```
CComQIPtr<IPropContainer>  Container;
GetChannelList() ->GetChannelContainer(1,
&__uuidof(IPropContainer), &Container);*
```

The IPropContainer interface pointer allows you to obtain or modify the characteristics of the second channel. For example, the following code fragment determines the current property value of the second channel's ChannelName property.

```
CComVariant val;
Container->get_MemberValue(L"channelname", &val);
```

The second channel's `ChannelName` property can be modified with the following command.

```
CComVariant val = L"NewChannelName";
Container->put_MemberValue(L"channelname", val);
```

The `CmwDevice` class supplies a helper template for this function, simplifying its use. Using this template in the preceding example, the starred line can be replaced with

```
GetChannelContainer(1, &Container);
```

See Also

`IPropContainer` interface: `get_MemberValue`, `put_MemberValue`

GetChannelStruct

`GetChannelStructLocal`

Syntax

```
HRESULT GetChannelStruct (long index, NESTABLEPROP **Channel);
```

Input Parameters

`index`	The zero-based index for the channel to be retrieved.

Output Parameters

`Channel`	The data structure for the requested channel.

Description

The `GetChannelStruct` method returns the channel structure for the specified channel to `Channel`. The index argument specifies the index of the channel (zero-based). An index of 0 specifies the first channel, an index of 1 specifies the second channel, and so on.

GetChannelStructLocal returns the actual pointer to the structure. It must never be deleted. GetChannelStruct returns a standard output ptr and the data must be deleted with CoTaskMemFree.

NESTABLEPROP is defined in mwstructs.h and is given below.

```
typedef struct tagNESTABLEPROP
{
long StructSize;
long Index;
NESTABLEPROPTYPES Type;
long HwChan;
BSTR Name;
} NESTABLEPROP;
```

The NESTABLEPROPTYPES are used to categorize the channel structure as an analog input channel, an analog output channel, or a digital I/O line. NESTABLEPROPTYPES is defined in mwstructs.h and is given below.

```
typedef enum tagNESTABLEPROPTYPES
{
NPAICHANNEL,
NPAOCHANNEL,
NPDIGITALLINE
} NESTABLEPROPTYPES;
```

GetNumberOfChannels

Syntax

```
HRESULT GetNumberOfChannels (long *numChans)
```

Output Parameters

numChans	The current number of channels.

Description

The GetNumberOfChannels method returns the number of channels contained in a channel group.

CreateChannel (proposed)

Syntax

```
HRESULT CreateChannel(long HwChannel,[out]IPropContainer** Cont);
```

Input Parameters

`HwChannel`	long

Output Parameters

`Cont`	An interface to the channel created.

Description

The purpose of `CreateChannel` is to get the channel structure for an existing channel. This function is currently not implemented and its interface might change.

DeleteChannel

Syntax

```
HRESULT DeleteChannel(long index);
```

Input Parameters

`index`	Zero-based index of the channel to delete.

Description

Deletes the specified channel or line. This function causes callbacks to the `ChildChange` function.

DeleteAllChannels

Syntax

```
HRESULT DeleteAllChannels();
```

Description

The `DeleteAllChannels` function deletes all channels or lines. This function causes callbacks to the `ChildChange` function. You should use `DeleteAllChannels` when you change a property that invalidates the current configuration. It is currently used by the `nidaq` adaptor when the `InputType` property is changed from single-ended to differential.

C

Engine Structures

This section describes engine-defined structures and enumerated data types. The base structures are defined in `daqmex.h`, built from the IDL file `daqmex.idl` by the MIDL compiler.

Other definitions and structures needed by an adaptor can be found in `daqmexstructs.h`. Over time, many of the definitions found in `daqmexstructs.h` can be moved to `daqmex.idl` to improve the information available in the type library.

The BUFFER_ST Structure

Used by

ImwDevice interface: AllocBufferData, FreeBufferData, PeekData

IDaqEngine interface: GetBuffer, PutBuffer, GetOutputData, PutInputData

Definition

```
typedef  struct tagBUFFER {
long Size; // In bytes
long ValidPoints; // In raw points
                  //(MATLAB samples is ValidPoints/channels)
unsigned char *ptr;
DWORD dwAdaptorData;  // Reserved by the engine for use by adaptor
unsigned long Flags; // Flag values are defined in
unsigned long Reserved; // Reserved for future use by the engine
hyper StartPoint; // Count of points since start
double StartTime; // Time of the start of the buffer from GetTime
double EndTime; // Time of the end of the buffer from GetTime
} BUFFER_ST;
```

Description

The fields contained by the BUFFER_ST structure are given below.

Table C-1: BUFFER_ST Fields

Field Name	Description
Size	Size of the buffer pointed to by ptr in bytes.
ValidPoints	Total number of data points in the buffer or requested to be put into the buffer. ValidPoints/number of channels is usually the buffer size returned by the BufferingConfig property.
ptr	Pointer to the data contained in the buffer.
dwAdaptorData	Reserved in the engine for use by the adaptor as needed.
Flags	Indicates information about the buffer.

Table C-1: BUFFER_ST Fields (Continued)

Field Name	Description
`StartPoint`	Number of points since start.
`StartTime`	Time of the start of the buffer from `IDaqEngine->GetTime`.
`EndTime`	Time of the end of the buffer from `IDaqEngine->GetTime`.

Bit values for the `Flags` field are given below. Note that all values can be ORed together.

Table C-2: Bit Values for the Flags Field

Defined String	Value	Description
`BUFFER_GAP_BEFORE`	`0x1`	Set this flag when data in the buffer is noncontiguous with data in the previous buffer. Noncontiguous data can come from a different trigger, or it can arise when data is lost between buffers.
`BUFFER_START_TIME_VALID`	`0x2`	Set this flag if the adaptor has put a valid time into the `StartTime`.
`BUFFER_END_TIME_VALID`	`0x4`	Set this flag if the adaptor has put a valid time into the `EndTime`.
`BUFFER_IS_LAST`	`0x8`	Set by the engine to notify the adaptor that input/output can stop at the end of this buffer. Determine how many samples to transfer from `ValidPoints`. If the adaptor did not completely fill previous buffers or if new data is available for output, this flag might not be accurate.
`BUFFER_TRANSMIT_DATA`	`0x10`	The buffer contains valid data and should be transmitted over the network.

The NESTABLEPROP Structure

Definition

```
typedef struct tagNESTABLEPROP
{
long StructSize;
long Index;
NESTABLEPROPTYPES Type;
long HwChan;
BSTR Name;
} NESTABLEPROP;
```

Description

The fields contained by the NESTABLEPROP structure are given below.

Table C-3: NESTABLEPROP Fields

Field Name	Description
StructSize	Size of the structure in bytes.
Index	Channel or line index.
Type	Type of channel or line.
HwChan	Hardware channel or hardware line.
Name	Channel name or line name.

Enum NESTABLEPROPTYPES Definition

```
typedef enum tagNESTABLEPROPTYPES
{
NPAICHANNEL,
NPAOCHANNEL,
NPDIGITALLINE
} NESTABLEPROPTYPES;
```

The structure NESTABLEPROP is defined for the purpose of function declarations and COM. A pointer to a NESTABLEPROP can be cast to a pointer to one of the following classes, depending on the value of the Type member.

```
typedef struct tagAICHANNEL {
NESTABLEPROP Nestable;
BSTR Units;
double VoltRange[2];
double UnitRange[2];
double SensorRange[2];
double ConversionExtraScaling; // Can be modified by the adaptor
double ConversionOffset;  // Extra offset for scaling data
double NativeOffset; // The adaptor should consider this read only
double NativeScaling; // Same here
BYTE extra[]; // do not access this NPextrasize is
Nestable.StructSize-sizeof(AICHANNEL)
} AICHANNEL;
```

```
typedef struct tagAOCHANNEL {
NESTABLEPROP Nestable;
BSTR Units;
double VoltRange[2];
double UnitRange[2];
double ConversionExtraScaling; // Can be modified by the adaptor
double ConversionOffset;
double NativeOffset; // The adaptor should consider this read only
double NativeScaling;
double DefaultValue;
BYTE extra[];
} AOCHANNEL;
```

```
typedef struct tagDIGITALLINE {
NESTABLEPROP Nestable;
long Direction;
long Port;
BYTE extra[];
} DIGITALLINE;
```

D

Sample Property and daqhwinfo Tables

Chapter 3, “Step-by-Step Instructions for Adaptor Creation,” discusses the use of `daqhwinfo` and `propinfo` tables in preparing to implement your adaptor. This appendix provides sample `daqhwinfo` tables and lists the properties that you should consider for analog input, analog output, and digital I/O subsystems.

For a complete discussion of the use of these tables when developing an adaptor, refer to “Step 3.1, Select Property Values, Ranges, and Defaults for Analog Input” on page 3-19.

Table of daqhwinfo Properties

The following tables list the `daqhwinfo` properties for all Keithley objects. These tables provide the blueprint for expected behavior when the user issues a `daqhwinfo` request on the adaptor, or an object created by the adaptor.

Adaptor daqhwinfo Table

The following table lists the values returned by a call to

```
daqhwinfo('keithley')
```

Table D-1: Table of daqhwinfo Properties

Property	Value
`AdaptorDLLName`	`mwkeithley.dll`
`AdaptorDLLVersion`	1.0
`AdaptorName`	keithley
`BoardNames`	CellArray of all available board names, constructed from the DriverLINX LDD `DevCap.VendorCode` and `DevCap.ModelCode` fields, plus the logical device number. Available boards are queried from the registry.
`InstalledBoardIDs`	CellArray of all available boards' device numbers, specified by the DriverLINX LDD `DeviceNumber` field. BoardIDs can be nonconsecutive.
`ObjectConstructorName`	Constructed from the above `BoardIDs` plus `'analoginput'`, etc.

Analog Input daqhwinfo Table

The following table lists the values returned by a call to

```
daqhwinfo(analoginput('keithley'))
```

Table D-2: Analog Input daqhwinfo Table

Property	Value
AdaptorName	'keithley' [hardcoded]
Bits	16
Coupling	'DC Coupled' [hardcoded]
DeviceName	Constructed from the DriverLINX LDD DevCap.ModelCode field, plus the logical device number of the device. For example, 'KPCI-3108 (Device 0)'
DifferentialIDs	Dependent on the device. Calculated from the LDD and the INI file.
Gains	Dependent on the device. Calculated from the LDD and the INI file.
ID	Given by the DeviceNumber property of the DriverLINX Configuration Panel.
InputRanges	Unipolar then bipolar ranges as specified in KPCI manual (Note: *not* hardcoded, derived from LDD)
MaxSampleRate	Depends on currently selected clock.
MinSampleRate	Depends on currently selected clock.
NativeDataType	int16
Polarity	{'Unipolar', 'Bipolar'}
SampleType	0 (sampled) [hardcoded. No boards are SSH]
SingleEndedIDs	Dependent on the device. Calculated from the LDD and the INI file.
SubsystemType	'AnalogInput' [engine]
TotalChannels	Dependent on the device. Calculated from the LDD.

Table D-2: Analog Input daqhwinfo Table (Continued)

Property	Value
VendorDriverDescription	Determined from LDD's DevCap.VendorCode field.
VendorDriverVersion	Determined from GetDriverLINXVersion call.

Analog Output daqhwinfo Table

The following table lists the values returned by a call to

```
daqhwinfo(analogoutput('keithley'))
```

Table D-3: Analog Output daqhwinfo Table

Property	Value
AdaptorName	'keithley' [hardcoded]
Bits	16
Coupling	'DC Coupled' [hardcoded]
DeviceName	Constructed from the DriverLINX LDD DevCap.ModelCode field, plus the logical device number of the device. For example, 'KPCI-3108 (Device 0)'
ID	Given by the DeviceNumber property of the DriverLINX configuration panel.
MaxSampleRate	Depends on currently selected Clock.
MinSampleRate	Depends on currently selected Clock.
NativeDataType	int16
OutputRanges	Dependent on the device. Calculated from the LDD.
Polarity	{'Unipolar', 'Bipolar'}
SampleType	0 (sampled)
SubsystemType	'AnalogOutput' [engine]

Table D-3: Analog Output daqhwinfo Table (Continued)

Property	Value
`TotalChannels`	Dependent on the device. Calculated from the LDD.
`VendorDriverDescription`	Determined from LDD's `DevCap.VendorCode` field.
`VendorDriverVersion`	Determined from `GetDriverLINXVersion` call.

Digital I/O daqhwinfo Table

The following table lists the values returned by a call to

```
daqhwinfo(digitalio('keithley'))
```

Table D-4: Digital I/O daqhwinfo Table

Property	Value
`AdaptorName`	`'keithley'` [hardcoded]
`DeviceName`	Constructed from the DriverLINX LDD `DevCap.ModelCode` field, plus the logical device number of the device. For example, `'KPCI-3108 (Device 0)'`
`ID`	Given by the `DeviceNumber` property of the DriverLINX Configuration Panel.
`PortDirections`	Dependent on the device. Calculated from the LDD.
`PortIDs`	Dependent on the device. Calculated from the LDD.
`PortLineConfig`	Dependent on the device. Calculated from the LDD.
`PortLineIDs`	Dependent on the device. Calculated from the LDD.
`SubsystemType`	`'DigitalIO'` [engine]
`TotalLines`	Dependent on the device. Calculated from the LDD.
`VendorDriverDescription`	Determined from LDD's `DevCap.VendorCode` field.
`VendorDriverVersion`	Determined from `GetDriverLINXVersion` call.

Property Info Tables

The following tables list the properties that should be considered when deciding on the `propinfo` table for your adaptor. Use of these properties to build up a `propinfo` table is discussed in Chapter 3, "Step-by-Step Instructions for Adaptor Creation." For a full description of these properties, consult the *Data Acquisition Toolbox User's Guide*.

Analog Input Subsystem Properties

The table below lists the properties that should be considered for analog input subsystems, as discussed in "Step 3.1, Select Property Values, Ranges, and Defaults for Analog Input" on page 3-19.

Table D-5: Analog Input Properties for propinfo Table

Property	Typical Adaptor Interaction
`BufferinConfig`	Cannot control this property directly; query through engine's `GetBufferingConfig` method. Typically, buffer sizes or event notification periods must be multiples of the engine buffer size to facilitate buffer transfers.
`ChannelSkew`	Closely coupled to `SampleRate` and `ChannelSkewMode` properties. Usually `ReadOnly` unless `ChannelSkewMode` is `Manual`. Not required to attach unless `Manual` mode is supported.
`ChannelSkewMode`	If `ChannelSkew` is not configurable, remove `Manual`. Attach if `Manual` is supported.
`ClockSource`	Extend to include `External` and/or `Software` if required. Attach to implement possible range changes for `Internal` to/from `Software` transition.
`InputType`	Defines coupling for the analog input channels. Use if your hardware supports software-configurable input types. Typical enumerated values are `SingleEnded` and `Differential`.
`SampleRate`	Almost always attached for an adaptor, to quantize values and/or check channel skew.
`TriggerChannel`	Might need to attach if hardware triggering is used.

Table D-5: Analog Input Properties for propinfo Table (Continued)

Property	Typical Adaptor Interaction
`TriggerCondition`	Can change depending on `TriggerType`. Typically, if hardware triggering is supported, `TriggerCondition` changes depending on the types of triggers implemented.
`TriggerConditionValue`	See `TriggerCondition`.
`TriggerDelay`	See `TriggerCondition`. For hardware triggering, might need to set to read only and zero if trigger delays are not supported by hardware.
`TriggerRepeat`	If you are performing hardware triggering, you need to monitor this property to stop, trigger, and start the acquisition until all triggers are received. Alternatively, do not support trigger repeat for hardware triggers.
`TriggerType`	Can extend to include hardware triggers. Consult the *User's Guide* for examples of trigger types.
Channel Properties	
`InputRange`	Typically one of a list of valid ranges. Most hardware devices define a gain that sets the input range. When a user selects a range, the adaptor should set the channel's range to the closest possible range that encompasses the required range.
`NativeOffset`	Conversion offset from native (raw) data to input range.
`NativeScaling`	Conversion scaling from native (raw) data to input range.

Analog Output Subsystem Properties

The table following lists the properties that should be considered for analog output subsystems, as discussed in "Step 4.1, Select Property Values, Ranges, and Defaults for Analog Output" on page 3-47.

Table D-6: Analog Output Properties for propinfo Table

Property	Typical Adaptor Interaction
`BufferinConfig`	Cannot control this property directly; query through engine's `GetBufferingConfig` method. Typically, buffer sizes or event notification periods must be multiples of the engine buffer size to facilitate buffer transfers.
`ClockSource`	Extend to include `External` and/or `Software` if required. Attach to implement possible range changes for `Internal` to/from `Software` transition.
`SampleRate`	Almost always attached for an adaptor, to quantize values.
`TriggerRepeat`	If you are performing hardware triggering, you need to monitor this property to stop, trigger, and start the acquisition until all triggers are received. Alternatively, do not support trigger repeat for hardware triggers.
`TriggerType`	Can extend to include hardware triggers. Consult the *User's Guide* for examples of trigger types.
Channel Properties	
`NativeOffset`	Conversion offset from native (raw) data to input range.
`NativeScaling`	Conversion scaling from native (raw) data to input range.
`OutputRange`	Typically one of a list of valid ranges. Most hardware devices define a gain that sets the output range. When a user selects a range, the adaptor should set the channel's range to the closest possible range that encompasses the required range.

Digital I/O Subsystem Properties

Because the digital I/O subsystem is implemented without any continuous acquisition or output, the properties should not need modification or monitoring. See the “Digital I/O daqhwinfo Table” on page D-6 for information on the digital I/O implementation table.