

BridgeVIEW



BridgeVIEW™ Device Server Toolkit Reference Manual

March 1997 Edition
Part Number 321298A-01



Internet Support

support@natinst.com

E-mail: info@natinst.com

FTP Site: ftp.natinst.com

Web Address: <http://www.natinst.com>



Bulletin Board Support

BBS United States: (512) 794-5422

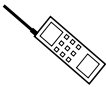
BBS United Kingdom: 01635 551422

BBS France: 01 48 65 15 59



Fax-on-Demand Support

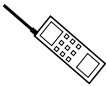
(512) 418-1111



Telephone Support (U.S.)

Tel: (512) 795-8248

Fax: (512) 794-5678



International Offices

Australia 02 9874 4100, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00,
Finland 09 527 2321, France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186,
Israel 03 5734815, Italy 06 5729961, Japan 03 5472 2970, Korea 02 596 7456,
Mexico 5 520 2635, Netherlands 31 348 43 34 66, Norway 32 84 84 00, Singapore 2265886,
Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200,
U.K. 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway Austin, TX 78730-5039 Tel: (512) 794-0100

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THEREFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

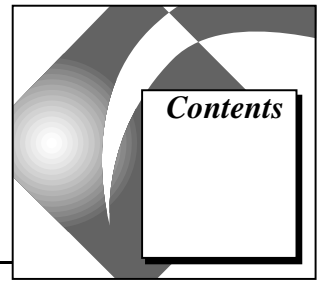
Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

BridgeVIEW™, National Instruments™, and natinst.com™ are trademarks of National Instruments Corporation. Product and company names listed are trademarks or trade names of their respective companies.

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.



About This Manual

Organization of This Manual	xv
Device Server Toolkit Concepts.....	xv
Appendices, Glossary, and Index	xvi
Conventions Used in This Manual.....	xvii
Customer Communication	xviii

Chapter 1

Getting Started

Using the BridgeVIEW Device Server Toolkit	1-1
Steps before Using the Toolkit	1-2
Device Server Toolkit Support	1-3
Viewing and Printing This File.....	1-4

Chapter 2

IAIO System and Device Server Overview

Components of the Device Server	2-1
Understanding Server Operations.....	2-2
Using Server Objects	2-4
Asynchronous Read Operations.....	2-4

Chapter 3

Behaviors of Device Server Components

Devices, I/O Points, and Tasks	3-1
Understanding Tasks.....	3-2
Communications Resources.....	3-3
Completion Mechanisms in Asynchronous I/O Operations	3-5

Chapter 4

IAIO API Function Reference

InitializeIAIOServer	4-1
TerminateIAIOServer	4-1
CreateIOPoint	4-1
CreateTagIOPoint	4-2
DestroyIOPoint	4-3
GetAttribute	4-3
SetAttribute	4-4
Read	4-4
ReadA	4-5
Write	4-6
WriteA	4-7
Advise	4-8
Stop	4-9
Clear	4-10
Wait	4-10
WaitForGroup	4-11
CreateAsynchCBCCompletion	4-12
CreateSynchCBCCompletion	4-13
CreateMessageCompletion	4-14
ReleaseCompletion	4-15
UserCallback	4-15
CheckDeviceStatus	4-16
DeviceStatusMsg	4-17
ErrorMsg	4-17

Chapter 5

IAIO Base Class Reference

Class CIAObject	5-1
Class CRequestQueue	5-1
Alert()	5-2
Remove()	5-2
Select()	5-2
Class CCommRsrcRequestQueue	5-2
Class CCommMgr	5-2
CCommMgr()	5-2
InitResource()	5-3
Alert()	5-3
SubmitRequest()	5-3
RemoveRequest()	5-3

	Read()	5-3
	Write()	5-3
Class CCommRS232Mgr		5-3
Class CDevice		5-4
	AddPendingRequest()	5-4
	RemovePendingRequest()	5-4
	ConstructReadPacket()	5-4
	ConstructWritePacket()	5-4
	Read()	5-4
	Write()	5-4
	Advise()	5-5
	SelectBatchRequests()	5-5
	SelectBatchRequestsLateArrival()	5-5
Class CIOPoint		5-5
	Read()	5-5
	Write()	5-5
	Advise()	5-6
	ExecuteRead()	5-6
	ExecuteWrite()	5-6
	ExecuteResponse()	5-6
	ClientToDeviceTypeConversionNecessary()	5-6
	ApplyClientToDeviceTypeConversion()	5-6
	DeviceToClientTypeConversionNecessary()	5-6
	ApplyDeviceToClientTypeConversion()	5-6
Class CTask		5-7
	IsCompleted()	5-8
	IsTerminated()	5-8
	ServiceRequest()	5-8
	CompleteRequest()	5-8
	AbortRequest()	5-8
	Cleanup()	5-8
	Stop()	5-8
	Clear()	5-8
	GetBuffer() and GetBufferSize()	5-9
	GetCompletion() and GetCompletionParam()	5-9
	NewState()	5-9
	FreePreviousState()	5-9
	FreeCompletedState()	5-9
	GetCurrentState()	5-9
	GetPreviousState()	5-9
	GetCompletedState()	5-9
	GetStateBuffer()	5-9
	GetStatus() and SetStatus()	5-10

GetTerminationStatus() and SetTerminationStatus()	5-10
ReadBuffer()	5-10
WriteBuffer()	5-10
CopyToClientBuffer()	5-10
Class CTaskState	5-10
Class CReadTask	5-11
Class CWriteTask	5-11
Class CAdviseTask	5-11
Class CBatchReadTask	5-12
Class CBatchWriteTask	5-13
Class CCompletion	5-13
Complete()	5-13
CompleteWithError()	5-13
Class CAsynchCBCompletion	5-14
Class CSynchCBCompletion	5-14
Class CMessageCompletion	5-15
Class CClient	5-16

Chapter 6

Server Customization

Global Functions That Can Be Overridden	6-1
ConstructIOPoint	6-2
ConstructDevice	6-2
ConstructCommMgrResource	6-2
Deriving from CDevice	6-2
Deriving From CIOPoint	6-3
Deriving from CCommMgr	6-4
Deriving from CRequestQueue	6-5

Chapter 7

IAIO Configuration Reference

The Framework of the IAIO Configuration API	7-1
GetSupportedProcedures	7-1
DuplicateCommResource	7-3
DuplicateDevice	7-3
DuplicateItem	7-3
EditCommResource	7-3
EditDevice	7-4
EditItem	7-4
EditServer	7-4
GetCommResourceList	7-5
GetDeviceList	7-5

GetItemList	7-5
RegisterCommResource.....	7-5
RegisterDevice	7-6
RegisterItem	7-6
RegisterServer	7-6
UnRegisterCommResource.....	7-7
UnRegisterDevice	7-7
UnRegisterItem	7-7
UnRegisterServer	7-7

Chapter 8

IAIO Configuration Customization

The Configuration Dialog Box Classes	8-1
CCommDialog	8-1
CCommDialog Member Variables.....	8-1
CCommDialog Methods.....	8-2
CCommDialog().....	8-2
CCommDialog (CString CommResourceName).....	8-2
OnInitDialog().....	8-2
OnOk()	8-3
OnCancel()	8-3
CDeviceDialog.....	8-3
CDeviceDialog Member Variables	8-3
CDeviceDialog Methods	8-4
CDeviceDialog()	8-4
CDeviceDialog (CString DeviceName).....	8-4
OnInitDialog().....	8-4
OnOk()	8-4
OnCancel()	8-5
CItemDialog.....	8-5
CItemDialog Member Variables	8-5
CItemDialog Methods	8-5
CItemDialog(CString DeviceName).....	8-5
CItemDialog (CString DeviceName, CString ItemName)	8-6
OnInitDialog().....	8-6
OnOk()	8-6
OnCancel()	8-6
IAIO Configuration API Behavior	8-7
DuplicateCommResourceList	8-7
EditCommResource	8-7
GetCommResourceList.....	8-7
RegisterCommResource.....	8-7

DuplicateDevice	8-7
EditDevice	8-8
RegisterDevice.....	8-8
DuplicateItem	8-8
EditItem	8-8
GetItemList.....	8-8
RegisterItem.....	8-9

Chapter 9 Common Configuration Database Reference

Using the Configuration Database.....	9-1
Active CCDB.....	9-2
CCDB Features.....	9-2
CCDB Tables.....	9-4
The Proxies Table	9-4
The Servers Table	9-5
The Devices Table	9-6
The Items Table	9-7
The Serial Table	9-10
The Generic Resources Table	9-11

Chapter 10 IAIO Servers Automation Reference

The Servers Automation Interface.....	10-1
The IManager Interface	10-1
ConnectDB	10-1
DisconnectDB.....	10-2
CompactDB	10-2
RepairDB	10-3
DeleteRow	10-3
BeginTransactions	10-3
CommitTransactions.....	10-3
RollbackTransactions	10-4
GetCurrentVersion.....	10-4
Query	10-4
RetrieveTable.....	10-4
The IIAIOProxy Interface	10-5
Proxy.....	10-5
Delete.....	10-5
Name.....	10-6
ProxyType	10-6

	ConfigPath	10-6
	LaunchPath.....	10-6
The IIAIOServers Interface		10-7
Server		10-7
Delete		10-7
Name		10-8
CanAddDevices		10-8
ServerType.....		10-8
LaunchPath.....		10-8
ConfigPath		10-9
SpecificInfo.....		10-9
The IIAIODEVICES Interface		10-9
Device		10-9
Delete		10-10
DeviceName.....		10-10
ServerName.....		10-10
DeviceType		10-10
Address.....		10-11
CanAddItems		10-11
DConfig.....		10-11
DefaultRate		10-11
DeviceResource		10-12
SpecificInfo.....		10-12
The IIAIOItems Interface		10-12
Item		10-12
Delete		10-13
ItemName.....		10-13
DeviceName.....		10-13
ServerName.....		10-14
NativeDataType		10-14
Address.....		10-14
Configurable.....		10-14
OnDataChange		10-15
DefaultRate		10-15
ItemCount.....		10-15
AccessRights.....		10-15
MaxRange		10-16
MinRange.....		10-16
MaxLength.....		10-16
Unit.....		10-16
SpecificInfo.....		10-17

The ISerial Interface	10-17
Resource	10-17
Delete	10-18
ResourceName	10-18
Port	10-18
ReadTimeoutInterval	10-18
StopBits	10-19
DataBits	10-19
BaudRate	10-19
Parity	10-19
The IGenericResource Interface	10-20
Resource	10-20
Delete	10-20
ResourceName	10-21
ResourceType	10-21
SpecificInfo	10-21

Appendix A Data Types and Attributes

Appendix B Diagnostic Error Messages

Appendix C Customer Communication

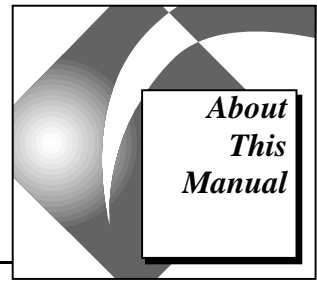
Glossary

Figures

Figure 2-1. Server Components	2-1
Figure 2-2. Server Operations	2-3
Figure 3-1. I/O Point, Task, and Device Relationships	3-2
Figure 3-2. Control Thread Execution	3-4
Figure 3-3. Class Hierarchy of the CCompletion Base Class	3-5
Figure 9-1. The Active CCDB in the Windows System Registry	9-2

Tables

Table 1-1.	IAIO C++ Framework Files	1-2
Table 7-1.	Supported Procedures and Corresponding Decimal Values	7-2
Table 9-1.	Tables and Primary Keys	9-3
Table 9-2.	Tables and Foreign Keys	9-3
Table 9-3.	The Proxies Table Definition	9-4
Table 9-4.	The Servers Table Definition	9-5
Table 9-5.	The Devices Table Definition	9-6
Table 9-6.	The Items Table Definition	9-7
Table 9-7.	The Serial Table Definition	9-10
Table 9-8.	The Generic Resource Table Definition	9-11



The *BridgeVIEW Device Server Toolkit Reference Manual* contains the information you need to get started developing BridgeVIEW Device Servers.

This manual explains the device server component behavior, IAIO API functions, IAIO base classes, IAIO API customization and configuration, the Common Configuration Database (CCDB), and device server automation. This manual also reviews data types and attributes and diagnostic error messages.

This manual presumes that you know how to operate your computer, that you are familiar with its operating system, and the concepts of C++.

Organization of This Manual

This manual is organized to help you write a device server. It is divided into an overview and a discussion of both device servers and the device server configuration utility. To assist you in writing a device server, this manual explains the behavior, function reference, and customization issues for the device server and configuration utilities.

Device Server Toolkit Concepts

- [Chapter 1, *Getting Started*](#), introduces the BridgeVIEW Device Server Toolkit. It outlines the steps you should take before using this toolkit and how to get Extended Development (EXD) Support. You also can find the tools you need to begin using your toolkit documentation in this chapter.
- [Chapter 2, *IAIO System and Device Server Overview*](#), describes the components of a device server and their relationships to each other. In this chapter, you also can learn more about server operations and how to use server objects in a device server.
- [Chapter 3, *Behaviors of Device Server Components*](#), describes the behaviors of I/O points, devices, tasks, and communications

resources. This chapter also reviews the behaviors of the task subclasses of the `CTask` class and completion mechanisms.



- [Chapter 4, IAIO API Function Reference](#), lists each IAIO API function, its purpose, and parameters.
- [Chapter 5, IAIO Base Class Reference](#), describes the base classes that form the architectural abstraction of an industrial automation device network. These classes comprise the foundation for customization of the IAIO server shell so it works with a particular device network.
- [Chapter 6, Server Customization](#), describes the classes, methods, and global functions you can modify to customize your device server.
- [Chapter 7, IAIO Configuration Reference](#), explains the framework of the IAIO Configuration API, including descriptions of its functions.
- [Chapter 8, IAIO Configuration Customization](#), explains how to customize configuration dialog boxes supplied with the IAIO Configuration API framework.
- [Chapter 9, Common Configuration Database Reference](#), describes the configuration database tables and how to use the server automation and interfaces to access the data stored in those tables.
- [Chapter 10, IAIO Servers Automation Reference](#), describes the OLE automation interfaces used to access and configure for the common configuration databases.

Appendices, Glossary, and Index

- [Appendix A, Data Types and Attributes](#), lists and describes data types and attributes.
- [Appendix B, Diagnostic Error Messages](#), breaks diagnostic error messages into three categories, and lists and describes each message.
- [Appendix C, Customer Communication](#), contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation.
- The [Glossary](#) contains an alphabetical list of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.

Conventions Used in This Manual

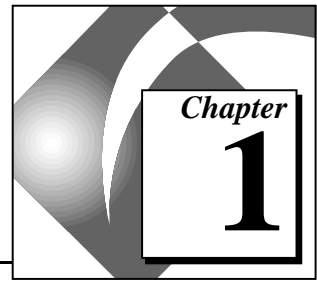
The following conventions are used in this manual:

bold	Bold text denotes a parameter, menu name, palette name, menu item, return value, function panel item, or dialog box button or option.
<i>italic</i>	Italic text denotes mathematical variables, emphasis, a cross reference, or an introduction to a key concept.
<i>bold italic</i>	Bold italic text denotes an activity objective, note, caution, or warning.
monospace	Text in this font denotes text or characters that you should literally enter from the keyboard. Sections of code, programming examples, and syntax examples also appear in this font. This font also is used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, variables, filenames, and extensions, and for statements and comments taken from program code.
<>	Angle brackets enclose the name of a key on the keyboard—for example, <PageDown>.
-	A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Control-Alt-Delete>.
<Control>	Key names are capitalized.
»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence File»Page Setup»Options»Substitute Fonts directs you to pull down the File menu, select the Page Setup item, select Options , and finally select the Substitute Fonts option from the last dialog box.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, and files, as in C:\dir1name\dir2name\filename.
	This icon to the left of bold italicized text denotes a note, which alerts you to important information.
	This icon to the left of bold italicized text denotes a caution, which alerts you to the possibility of data loss or a system crash.
	Abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms are listed in the <i>Glossary</i> .

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix C, *Customer Communication*, at the end of this manual.

Getting Started



This chapter introduces the BridgeVIEW Device Server Toolkit. It outlines the steps you should take before using this toolkit and how to get Extended Development (EXD) Support. You also can find the tools you need to begin using your toolkit documentation in this chapter.

Using the BridgeVIEW Device Server Toolkit

The BridgeVIEW Device Server Toolkit establishes the framework for the rapid development of customized device servers. Servers are software components that perform basic I/O operations such as read and write. Each server accepts the necessary protocols to extract data from a particular device and manages the resources required for communications.

You can develop a device server for any device that collects and reports data on a network. Usually these devices are Programmable Logic Controllers (PLCs) or remote I/O products that communicate to a host computer using serial, Ethernet, or other proprietary connections. The servers you develop with this toolkit conform to the National Instruments IAIO Application Programming Interface (IAIO API) specification, and you can use them directly with BridgeVIEW 1.0, LabVIEW 4.0, and LabWindows[®]/CVI 4.0.

The BridgeVIEW Device Server Toolkit requires that you develop in C++ and understand its many features. This toolkit assumes you thoroughly understand virtual functions and class inheritance. Before starting development, please review the VI Server Development Toolkit to see if it satisfies your development needs.

Steps before Using the Toolkit

You need to complete the following steps before using the material in this document:

1. Install a C/C++ compiler that uses the Microsoft Foundation Class libraries version 4.1 or later. It is recommended that you install Microsoft Visual C++ version 4.1 or later.
2. Install the BridgeVIEW Device Server Toolkit from the CD. This will install a series of C++, header, resource, and make files to use in the development of your server.
3. If you use a C++ development environment, load the `iaiosrvr.mak` file as a new project workspace. Create new `.cpp` and `.h` files in this project for your server-specific code. Save the project. The following table shows the files that are installed.

Table 1-1. IAIO C++ Framework Files

File	Purpose
<code>IAIO.h</code>	IAIO API header file
<code>IAIOmgr.cpp</code>	IAIO manager class implementation file
<code>IAIOmgr.h</code>	IAIO manager class header file
<code>IAIOSrvr.cpp</code>	device server entry points
<code>IAIOSrvr.h</code>	device server entry points header file
<code>IAIOSrvr.mak</code>	device server make file
<code>IAIOSrvr.rc</code>	device server resource file
<code>IAIOutil.cpp</code>	IAIO utility class implementation file
<code>IAIOutil.h</code>	IAIO utility class header file
<code>Resource.h</code>	IAIO resource header file
<code>Servers.cpp</code>	IAIO configuration database class file
<code>Servers.h</code>	IAIO configuration database class header file

Table 1-1. IAIO C++ Framework Files

File	Purpose
StdAfx.cpp	IAIO precompiled file
StdAfx.h	IAIO precompiled header file

4. Install and configure the industrial automation device you plan to use to develop a device server. Verify that the device functions properly before trying to communicate using the IAIO API.
5. Familiarize yourself with the protocol you want to use to develop a device server.
6. If the device uses a proprietary communications network, familiarize yourself with its architecture and how to establish a software connection to it.

Device Server Toolkit Support

National Instruments provides Extended Development (EXD) support for development of device servers. This level of technical support goes beyond basic support to provide you with in-depth technical assistance in using this toolkit.

You can reach EXD support experts by phone, fax, and e-mail during normal business hours. See [Appendix C, Customer Communication](#), for more information about contacting National Instruments.

You can purchase EXD support on a per incident basis by credit card or by annual contract. An annual contract provides a set number of incidents for multiple users during a one year period. The following EXD support packages are available.

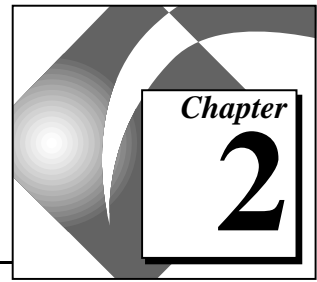
EXD Support Packages	Part Number	Cost
Single EXD Support Incident	960019-01	\$195
EXD Support Annual Contract (up to 6 incidents)	690019-02	\$995
EXD Support Annual Contract (up to 12 incidents)	690019-03	\$1,895

If you anticipate more than 12 incidents, you can purchase multiple annual support contracts.

Viewing and Printing This File

You can view this file using any version of Adobe Acrobat Reader. For best results when printing this file, use Adobe Acrobat Reader 3.0. Adobe Acrobat Reader 3.0 is available from the Adobe web site at <http://www.adobe.com/acrobat>.

IAIO System and Device Server Overview



This chapter describes the components of a device server and their relationships to each other. In this chapter, you also can learn more about server operations and how to use server objects in a device server.

Components of the Device Server

A device server consists of the server software, configuration utility, and configuration database, as shown in the following illustration. In the configuration utility, you specify configuration parameters for the device and the protocol.

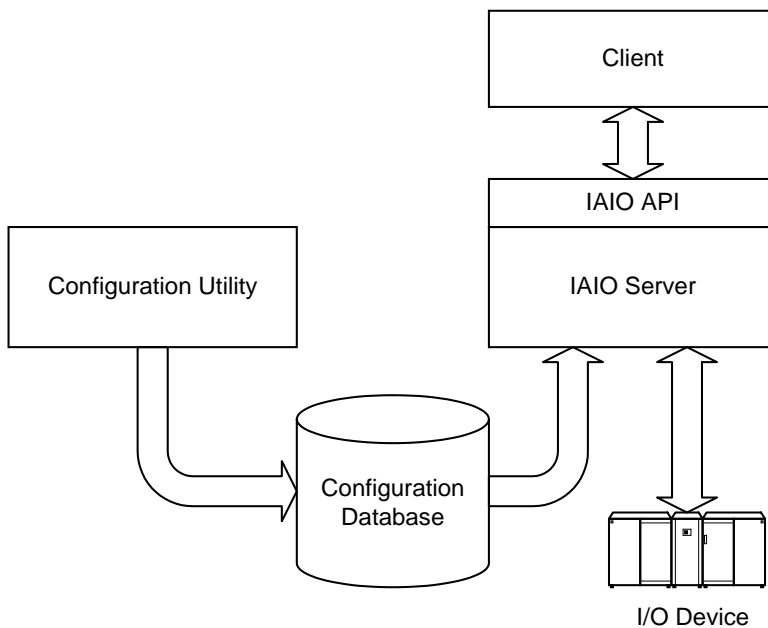


Figure 2-1. Server Components

On many devices, complex addresses reference I/O points in a device network. With this utility, you can specify and assign aliases to these addresses, making it easier to reference data points from a client (e.g. BridgeVIEW) using the device server.

The configuration database stores all configuration information about the device network, the devices, and the aliases. One configuration database is available to all device servers.

A client application using a device server makes requests to read or write to a device through the IAIO API. The server retrieves information that relates to the configuration of that device from the configuration database and uses it to configure and communicate with the proper I/O point on the device.

In the following sections, you can find out more about how the server operates and how you can use objects for the device server.

Understanding Server Operations

In addition to reading and writing to an I/O point on a device, the IAIO API helps you to execute an `Advise` operation. An `Advise` operation monitors an I/O point and returns its status and value to the client at a given interval. Every `Read` and `Advise` operation returns the value from the physical sensor, a status of the operation, and the time when the reading occurred. A `Write` operation returns the status of the operation and the time at which the server sent the value to the device.

The following illustration shows the relationship of the server to the client and the I/O device.

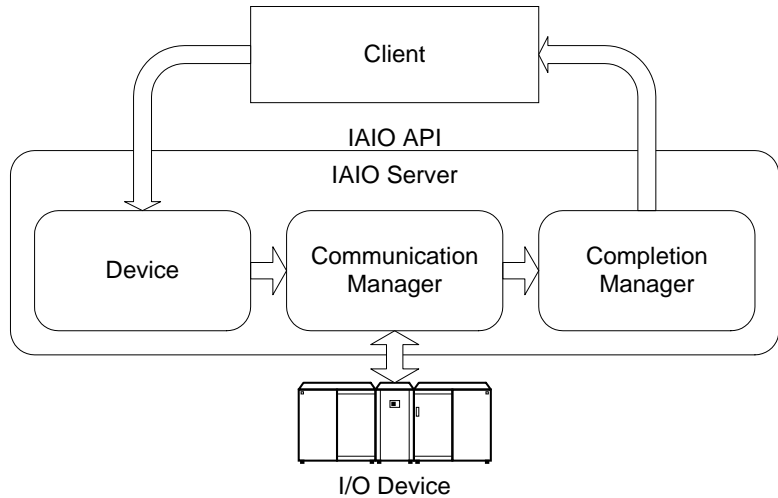


Figure 2-2. Server Operations

All client interaction with the device server occurs through the IAIO API. When the client initiates an operation, the server either creates a new object or identifies which existing object performs the operation. Objects either take the form of an I/O point, a task, or a completion mechanism. An I/O point contains all the information necessary to describe the properties and location of a sensor on the physical device network.

The client can choose a *completion mechanism*, which is the method that notifies a client when an operation is complete. The I/O operations provided by the IAIO API can specify either synchronous or asynchronous execution. Furthermore, asynchronous I/O operations must specify one of three completion mechanisms: asynchronous, synchronous, or window message.

A task represents a particular I/O operation for an I/O point. An example of this is a `Read` or `Write` operation. The completion mechanism describes how the data and status from the device are returned to the client. A callback function notification is an example of a completion mechanism.

Using Server Objects

The server groups all objects by both events and logical associations. For example, when the client initiates a `Read` operation, the object manager creates a task for that I/O point and schedules it for execution. Grouping all tasks makes it convenient for the communication manager to identify which operations to execute and when.

The server also groups each object by a logical association to a physical device and the communications resource. Typically, a device contains many I/O points. A communications resource is the physical gateway from the computer to the device. The most common communications resource is a serial port. Each object connects logical references to I/O points that reside on a device. In addition, each object uses a communications resource and has a task scheduled for execution. It also uses a given completion mechanism.

You can find the configuration details of the I/O point, device, and communications resource in [Chapter 7, IAIO Configuration Reference](#), [Chapter 8, IAIO Configuration Customization](#), and [Chapter 9, Common Configuration Database Reference](#).

Asynchronous Read Operations

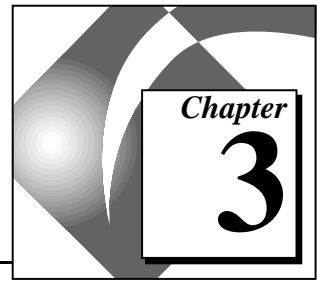
For an asynchronous read, the object manager creates an I/O point based on the information that the client supplies. This includes the device and communications resource information that has been gathered from the configuration database. The client then identifies the completion mechanism for the asynchronous read by specifying and registering the location of the callback function or window message.

After this setup, the client invokes the asynchronous read call. At this time, the servers find the previously configured I/O point and create a task. The servers submit this task to the communications manager for execution. When the task is complete, the execution manager looks up the completion mechanism and passes the data and status to the client through the callback function or window message facility.

Although this is not a complete description of the operation model of a device server, you can gain a basic understanding of the main components and how they interact. If you want to find more detailed discussion of the device server, the configuration database, and other components of the BridgeVIEW Device Server Toolkit, you can refer

to [Chapter 3, Behaviors of Device Server Components](#), [Chapter 4, IAIO API Function Reference](#), [Chapter 7, IAIO Configuration Reference](#), [Chapter 9, Common Configuration Database Reference](#), and [Chapter 10, IAIO Servers Automation Reference](#).

Behaviors of Device Server Components



This chapter describes the behaviors of I/O points, devices, tasks, and communications resources. This chapter also reviews the behaviors of the task subclasses of the `CTask` class and completion mechanisms.

Devices, I/O Points, and Tasks

Upon initialization, the server creates a device object and an I/O point object for each configured device and I/O point in the configuration database. The device object contains a reference to all I/O points configured on it and physically connected to it.

When the client initiates an advise, read, or write operation, the device responds by creating a task. A task is a specific instance of an I/O operation on a physical network. The task contains data indicating on what I/O point it operates, on what device the I/O point belongs, and the completion method to use when finished. Both the device and the referred I/O point keep a reference to the task the device created. The device object submits the task to the communications resource for execution.

When the communications resource is ready, it requests that the task construct a frame to send to the physical device. The task invokes a method on the I/O point object to construct the frame. The communications manager schedules the frame to be sent to the physical device.

After the communications manager reads the response, it again requests that the task parse the frame returned from the physical device. Once more, the task invokes a method on the I/O point object to parse the

frame. The task then submits the data to the proper completion mechanism for return to the client.

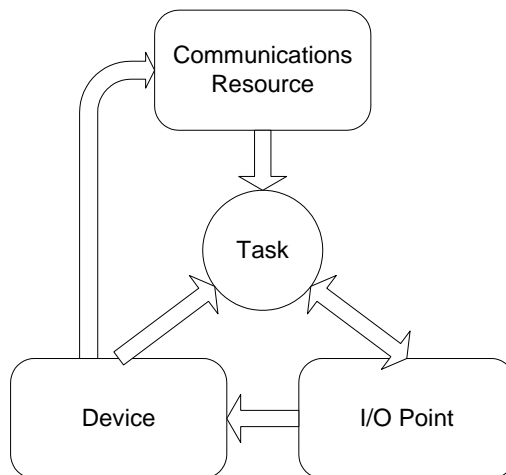


Figure 3-1. I/O Point, Task, and Device Relationships

Understanding Tasks

A task is defined as an I/O operation performed on an I/O point. A completion notification mechanism reports the results of this task. There are three types of tasks: read, write, or advise. A read operation causes the device server to obtain information from an I/O point. A write operation causes the device server to change the value of an I/O point. An advise operation, when it is initiated, automatically reads the value of an I/O point at regular intervals and reports the results to the client.

All tasks operate on I/O points, which belong to devices. You access each device in an industrial automation network through a particular communications resource.

Communications Resources

A communications resource is a communication channel abstraction to a device network. You can access it through specific combinations of communication protocols and actual network hardware. The communications resource manager base class `CCommMgr` represents this abstraction in the device server.

A communications resource manager has the basic I/O operations necessary to access devices on an industrial automation network. It also manages the allocation and access to the communications resource for pending tasks. Communications resources typically represent some communication channel or protocol for the device server to access devices on the network. Examples of communications resources are serial communication ports (for example, RS-232, RS-485), Ethernet network access (TCP/IP), or proprietary hardware network access such as a plug-in network card.

Communications resource management is either serial or concurrent. Only one task at a time can allocate and use a serial communications resource. A common example of this is an RS-232 communications resource (`CRS232CCommMgr`), in which a task must have exclusive access to an RS-232 port for the duration of a transaction. Two different tasks cannot use a single RS-232 port simultaneously. Therefore, a serial `CCommMgr` is responsible for selecting and allocating its resource to a single task at a time.

In contrast, concurrent communications resources can be allocated and used by more than one task simultaneously. However, the number of tasks that a single communications resource can service concurrently typically has a finite limit. A `CCommMgr` class that is concurrent is responsible for selecting and allocating its resources to many tasks at a time. When the number of tasks has saturated its resources (for example, all TCP/IP sockets are currently in use), it must queue pending tasks that need access to that communications resource.

A device server can have any number and combination of communications resources. It is entirely dependent on the topography of the device network being served. For example, a device server might be accessing two factory automation networks on two RS-232 serial ports (represented by two `CRS232CCommMgr` objects).

A class derived from the `CCommMgr` base class can represent any communications resource manager. Each `CCommMgr` object has its own

control thread that manages resource access and task service execution. This control thread is a base class `CCommMgr` method that you can override.

When a communications resource manager has selected a task, the control thread executes steps necessary to service the specific operation represented by the task. Figure 3-2 below illustrates these general steps.

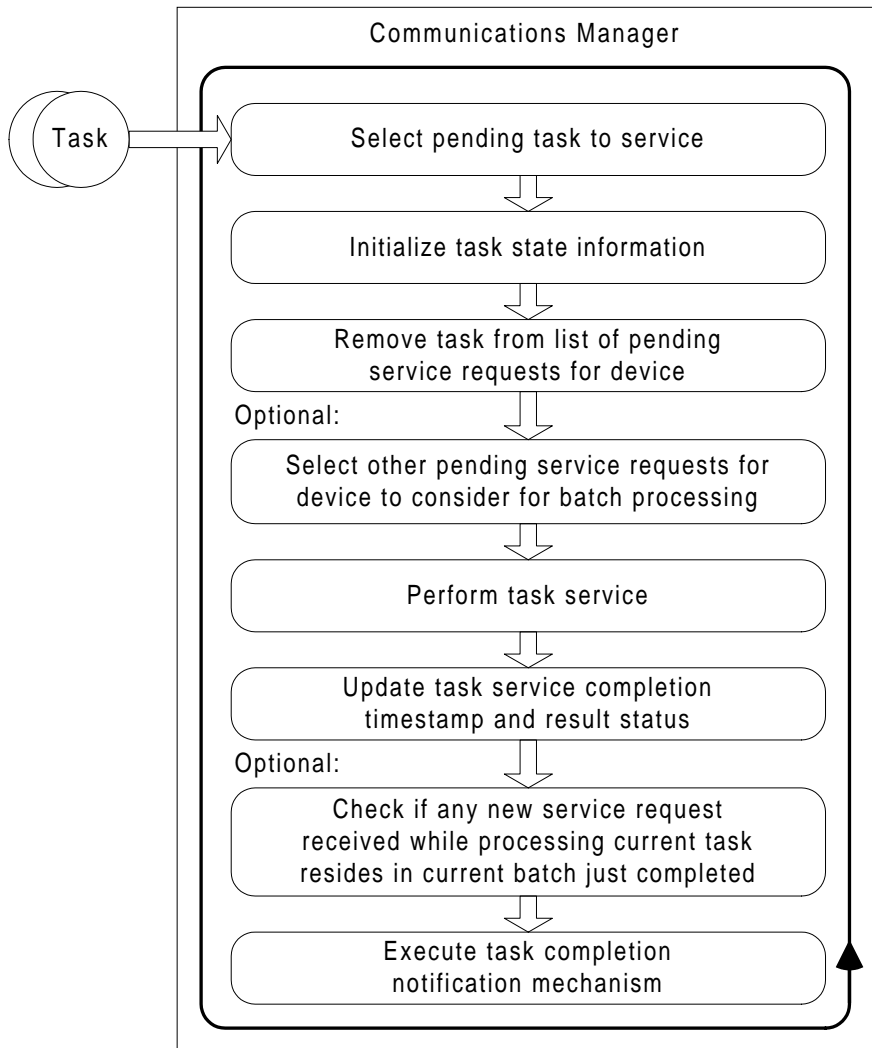


Figure 3-2. Control Thread Execution

Completion Mechanisms in Asynchronous I/O Operations

A completion mechanism is the method the server uses to notify a client of the results and status of an asynchronous I/O operation. There are three different types of completion mechanisms: asynchronous, synchronous, and message. [Chapter 4, IAIO API Function Reference](#), explains their differences in detail.

The `CCompletion` base class represents a completion mechanism, because it is internal to a device server. There are three derived subclasses which represent each of the different types of completion mechanisms. They are `CAsynchCCompletion`, `CSynchCCompletion`, and `CMessageCompletion`. The figure below shows the class hierarchy.

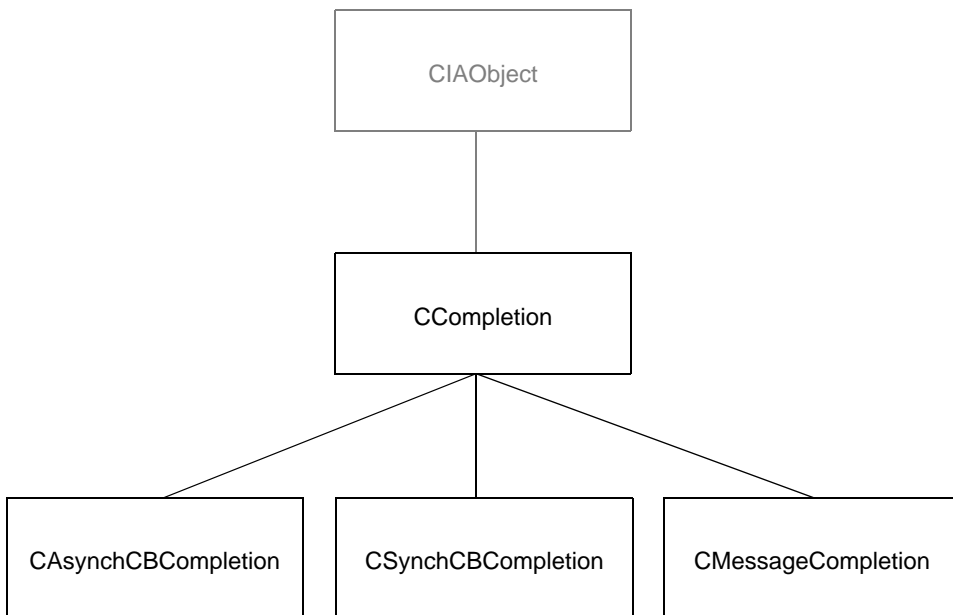


Figure 3-3. Class Hierarchy of the `CCompletion` Base Class

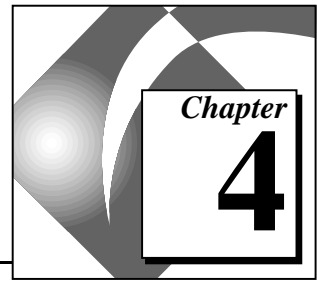
Although each of the three subclasses have common methods inherited from the base class `CCompletion`, they operate very differently from one another.

The control thread of the communications resource manager invokes the `CompleteRequest()` method for the task. This action sends a pointer to the `CCompletion` object for the completion mechanism specified to the I/O operation by the client. It then invokes the `CCompletion::Complete()` virtual method that implements the specific type of completion behavior expected.

At some point during `CCompletion::Complete()`, all of the completion mechanism types carry out the following sequence of steps:

1. Copy results of operation into the client buffer.
2. Free the just completed task state buffer for reuse.
3. Invoke the `CTask::Cleanup()` virtual method.

IAIO API Function Reference



This chapter lists each IAIO API function, its purpose, and parameters.

InitializeIAIOServer

```
IAStatus InitializeIAIOServer(void);
```

Purpose

Initializes the device server. Call this function once before implementing any other IAIO API function calls to initialize the server properly.

TerminateIAIOServer

```
IAStatus TerminateIAIOServer(void);
```

Purpose

Terminates device server execution. Any pending tasks on the server terminate. Call this function once after all other IAIO API function calls have completed.

CreateIOPoint

```
IAStatus CreateIOPoint (  
    IAString device,  
    IAString firstItem,  
    Int32 nItems,  
    IAType type,  
    IAHandle *hIOPoint);
```

Purpose

Creates an I/O point object that references any data item on a device. A single I/O point can reference one item or a block of items on the device. If `nItems` indicates more than one item, the items must be contiguous and the same type. This function returns a handle to the created I/O point in `hIOPoint`.

Parameters

Name	Direction	Description
device	In	Address or name of a device on a device communication network.
firstItem	In	Beginning address of a data item on a device.
nItems	In	Number of items in the I/O point beginning with firstItem.
type	In	Data type associated with all items in the I/O point.
hIOPoint	Out	Pointer to I/O point handle.

CreateTagIOPoint

```
IAStatus CreateTagIOPoint (
    IAString tagName,
    IAHandle *hIOPoint);
```

Purpose

Creates an I/O point object that references any data item on a device using the configuration information of the named tag. You must enter the tag information into the server tag configuration for this call to succeed. hIOPoint returns a handle to the created I/O point.

Parameters

Name	Direction	Description
tagName	In	Name of I/O point in tag configurations.
hIOPoint	Out	Pointer to I/O point handle.

DestroyIOPoint

```
IAStatus DestroyIOPoint (IAHandle hIOPoint);
```

Purpose

Destroys an I/O point as a resource. After `DestroyIOPoint` executes, `hIOPoint` is no longer a valid handle.

Parameters

Name	Direction	Description
<code>hIOPoint</code>	In	Handle to I/O point.

GetAttribute

```
IAStatus GetAttribute (
    IAHandle hIOPoint,
    IAAttr attrib,
    IAType value);
```

Purpose

Reads the value of the indicated attribute for `hIOPoint`.

Parameters

Name	Direction	Description
<code>hIOPoint</code>	In	Handle to I/O point.
<code>attrib</code>	In	Attribute to read.
<code>value</code>	Out	Pointer to value of attribute.

SetAttribute

```
IAStatus SetAttribute (
    IAHandle hIOPoint,
    IAAttr attrib,
    IAType value);
```

Purpose

Modifies the value of the indicated attribute for `hIOPoint`.

Parameters

Name	Direction	Description
<code>hIOPoint</code>	In	Handle to I/O point.
<code>attrib</code>	In	Attribute to set.
<code>value</code>	Out	Value of attribute.

Read

```
IAStatus Read (
    IAHandle hIOPoint,
    IABYTE buffer[],
    UInt32 bufferSize,
    IATimeStamp *timeStamp);
```

Purpose

Reads time-stamped data from an I/O point into the indicated `buffer`. The operation is synchronous and therefore blocks client execution until the read operation completes or the timeout attribute expires.

Parameters

Name	Direction	Description
hIOPoint	In	Handle to I/O point.
buffer[]	Out	Buffer for data transfer.
bufferSize	In	Size of buffer.
timeStamp	Out	Pointer to time when data read.

ReadA

```
IAStatus ReadA (
    IAHandle hIOPoint,
    IABYTE buffer[],
    UInt32 bufferSize,
    IAHandle hCompletion,
    DWORD cParam,
    IATaskID *taskID);
```

Purpose

Reads time-stamped data from the indicated I/O point (`hIOPoint`) into a buffer. `ReadA` is an asynchronous I/O operation. When `ReadA` initiates the operation, it immediately returns and the client thread continues execution. At some point later, `ReadA` completes and notifies the client with the results of the operation. The `hCompletion` parameter indicates the completion and notification mechanism used by `ReadA`. See `CreateAsynchCBCCompletion`, `CreateSynchCBCCompletion`, and `CreateMessageCompletion` for an explanation of the available asynchronous I/O completion mechanisms available

A unique `taskID` is assigned to each `ReadA` operation initiated.

Parameters

Name	Direction	Description
hIOPoint	In	Handle to I/O point.
buffer[]	Out	Buffer for data transfer.
bufferSize	In	Size of buffer.
hCompletion	In	Handle to completion mechanism.
cParam	In	User-defined completion callback or message parameter.
taskID	Out	Pointer to task ID for this operation.

Write

```
IAStatus Write (
    IAHandle hIOPoint,
    IABYTE buffer[],
    UInt32 bufferSize);
```

Purpose

Writes data from the indicated buffer to an I/O point. The operation is synchronous; therefore, it blocks client execution until the write operation completes or the timeout expires.

Parameters

Name	Direction	Description
hIOPoint	In	Handle to I/O point.
buffer []	In	Buffer for data transfer.
bufferSize	In	Size of buffer.

WriteA

```
IAStatus WriteA (I
    AHandle hIOPoint,
    IABYTE buffer[],
    UInt32 bufferSize,
    IAHandle hCompletion,
    DWORD cParam,
    IATaskID *taskID);
```

Purpose

Writes data to the indicated I/O point (`hIOPoint`) from a buffer. `WriteA` is an asynchronous I/O operation. When `WriteA` initiates the operation, it immediately returns and the client thread continues execution. At some point later, `WriteA` completes and notifies the client with the results of the operation. The `hCompletion` parameter indicates the completion and notification mechanism used by `WriteA`. See `CreateAsynchCBCCompletion`, `CreateSynchCBCCompletion`, and `CreateMessageCompletion` for discussion of the asynchronous I/O completion mechanisms available.

A unique `taskID` is assigned to each `WriteA` operation initiated.

Parameters

Name	Direction	Description
<code>hIOPoint</code>	In	Handle to I/O point.
<code>buffer[]</code>	In	Buffer for data transfer.
<code>bufferSize</code>	In	Size of buffer.
<code>hCompletion</code>	In	Handle to completion mechanism.
<code>cParam</code>	In	User-defined completion callback or message parameter.
<code>taskID</code>	Out	Pointer to task ID for this operation.

Advise

```

IAStatus Advise (
    IAHandle hIOPoint,
    Int32 rate,
    IABoolean fNotifyOnChange,
    IAByte buffer[],
    UInt32 bufferSize,
    IAHandle hCompletion,
    DWORD cParam,
    IATaskID *taskID);

```

Purpose

Continuously reads time-stamped data at an indicated rate from the I/O point (`hIOPoint`) into a buffer. `Advise` is an asynchronous I/O operation. When `Advise` initiates the operation, it immediately returns and the client thread continues execution. `Advise` periodically notifies the client with the results of the most recent data read. If `fNotifyOnChange` is `TRUE`, `Advise` only notifies the client if the I/O point value has changed since the last read. If `fNotifyOnChange` is `FALSE`, `Advise` always notifies the client with the most recent I/O point value read.

The `hCompletion` indicates the completion and notification mechanism that `Advise` uses to update the client. See `CreateAsynchCBCCompletion`, `CreateSynchCBCCompletion`, and `CreateMessageCompletion` for discussion of the asynchronous I/O completion mechanisms available.

An `Advise` operation continues to monitor the I/O point at the indicated rate until explicitly terminated by a call to `Stop`.

The server assigns a unique `taskID` to each `Advise` operation initiated.

Parameters

Name	Direction	Description
<code>hIOPoint</code>	In	Handle to I/O point.
<code>rate</code>	In	Rate to monitor for changes.
<code>fNotifyOnChange</code>	In	Flag specifying notification condition.
<code>buffer[]</code>	In	Buffer for data transfer.

Name	Direction	Description
bufferSize	Out	Size of buffer.
hCompletion	In	Handle to completion mechanism.
cParam	In	User-defined completion callback to message parameter.
taskID	Out	Pointer to task ID for this operation.

Stop

```
IAStatus Stop (IATaskID taskID);
```

Purpose

Terminates the asynchronous I/O operation identified by `taskID`. Any pending operation on the I/O point is canceled.

If the terminated asynchronous I/O operation uses an asynchronous callback completion mechanism, the server invokes the callback immediately with a termination status. Conversely, if the terminated asynchronous I/O operation indicates a synchronous callback completion mechanism, the server invokes the callback with a termination status concurrent with the next call to `Wait` or `WaitForGroup`. If the terminated asynchronous I/O operation indicates a message completion mechanism, the server does not post a termination message.

Parameters

Name	Direction	Description
taskID	In	Task ID for the operation to stop.

Clear

```
IAStatus Clear (IATaskID taskID);
```

Purpose

Marks `taskID` as clear and free for reuse by the device server. You cannot clear a `taskID` for an operation that has not completed or terminated. Each `taskID` needs to be cleared upon operation completion or termination.

Parameters

Name	Direction	Description
<code>taskID</code>	In	Task ID for the operation to stop.

Wait

```
IAStatus Wait (
    IATaskID taskID,
    Int32 timeout);
```

Purpose

Blocks execution until either the asynchronous I/O operation indicated by `taskID` has completed or the timeout period has expired. `wait` returns immediately if the asynchronous I/O operation indicated by `taskID` already has completed.

Parameters

Name	Direction	Description
<code>taskID</code>	In	Task ID to wait for completion.
<code>timeout</code>	In	Maximum interval to wait on task completion before returning.

WaitForGroup

```
IAStatus WaitForGroup (
    IATaskID aTaskID[],
    Int32 *nTaskID,
    Int32 timeout,
    IABoolean fWaitForAll);
```

Purpose

Blocks execution until either the asynchronous I/O operations indicated in `aTaskID` have completed or the timeout period expires. If `fWaitForAll` is `TRUE`, `WaitForGroup` blocks until all of the indicated tasks have completed or the timeout expires. If `fWaitForAll` is `FALSE`, `WaitForGroup` blocks until any of the indicated tasks have completed or the timeout expires. Any asynchronous I/O operations using synchronous callback completion that are pending completion are completed and their associated callbacks invoked regardless of whether they are one of the operations indicated in `aTaskID`.

Upon return, `nTaskID` contains the number of the originally indicated operations that have completed, and `aTaskID` contains the task IDs of those completed operations.

Parameters

Name	Direction	Description
<code>aTaskID[]</code>	In and Out	Array of task IDs awaiting completion or having been completed.
<code>nTaskID</code>	In and Out	Number of task IDs in <code>TaskID</code> .
<code>timeout</code>	In	Maximum interval to wait on task completion before returning.
<code>fWaitForAll</code>	In	Flag specifying wait condition.

CreateAsynchCBCompletion

```
IAStatus CreateAsynchCBCompletion (
    IAUserCallback callback,
    IAHandle *hCompletion);
```

Purpose

Creates an asynchronous callback completion mechanism for use with asynchronous I/O operations. A handle uniquely identifying the completion mechanism is returned in `hCompletion`.

Operation completion consists of copying any data into the client buffer, updating status information, and notifying the client through the user-defined callback. Notification of timeout expiration or operation termination (by calling `Stop`) also occurs through the callback interface. Callback notification does not occur if the server does not initiate the operation successfully.

Passing an asynchronous callback completion handle to an asynchronous I/O operation causes the server to invoke the user-defined callback immediately upon completion of the asynchronous I/O operation. No client synchronization using `Wait` or `WaitForGroup` is required for completion of an asynchronous I/O operation using this type of completion mechanism.

The callback function might be invoked from another thread running asynchronously from the client thread that originated the asynchronous I/O operation. Therefore, it is important to use synchronization mechanisms to protect any shared data between the callback function and the rest of the client application. Although the callback function may be invoked synchronously with a `Wait` or `WaitForGroup`, do not assume this will happen.

Parameters

Name	Direction	Description
<code>callback</code>	In	Pointer to the callback function.
<code>hCompletion</code>	Out	Pointer to the completion mechanism handle.

CreateSynchCBCompletion

```
IAStatus CreateSynchCBCompletion (
    IAUserCallback callback,
    IAHandle *hCompletion);
```

Purpose

Creates a synchronous callback completion mechanism for use with asynchronous I/O operations. A handle uniquely identifying the completion mechanism is returned in `hCompletion`.

Operation completion consists of copying any data into the client buffer, updating status information, and notifying the client through the user-defined callback. Notification of timeout expiration or operation termination (by calling `STOP`) also occurs through the callback interface. Callback notification does not occur if the server does not initiate the operation successfully.

Passing a synchronous callback completion handle to an asynchronous I/O operation causes the server to invoke the user-defined callback upon completion of the asynchronous I/O operation. You need to have client synchronization using `wait` or `waitForGroup` for completion of an asynchronous I/O operation using this type of completion mechanism. An asynchronous I/O operation that uses synchronous callback completion and that is ready to complete does not complete and invoke its callback until the client thread that initiated the operation synchronizes its execution with a call to `wait` or `waitForGroup`.

The server invokes the user-defined callback in the context of the client thread that initiated the asynchronous I/O operation.

Parameters

Name	Direction	Description
<code>callback</code>	In	Pointer to the callback function.
<code>hCompletion</code>	Out	Pointer to the completion mechanism handle.

CreateMessageCompletion

```
IAStatus IAIO::CreateMessageCompletion (
    HWND hWnd,
    DWORD message,
    IAHandle *hCompletion);
```

Purpose

Creates a message completion mechanism for use with asynchronous I/O operations. `hCompletion` returns a handle uniquely identifying the completion mechanism.

Operation completion consists of copying any data into the client buffer, updating status information, and notifying the client by posting a message to a user-indicated window. Notification of timeout expiration or operation termination (by calling `Stop`) also occurs through the message interface. Message notification does not occur if the server does not initiate the operation successfully.

Passing a message completion handle to an asynchronous I/O operation causes a message to be posted to the indicated window (`hWindow`) immediately upon completion of the asynchronous I/O operation. You do not need client synchronization using `Wait` or `WaitForGroup` for completion of an asynchronous I/O operation using this type of completion mechanism.

Parameters

Name	Direction	Description
<code>hWindow</code>	In	Handle window to send message to.
<code>message</code>	In	Message to send.
<code>hCompletion</code>	Out	Pointer to the registered callback handle.

ReleaseCompletion

```
IAStatus ReleaseCompletion (IAHandle hCompletion);
```

Purpose

Releases a completion mechanism. After `ReleaseCompletion` returns, `hCompletion` is no longer a valid handle.

Parameters

Name	Direction	Description
<code>hCompletion</code>	In	Handle to completion mechanism.

UserCallback

```
void UserCallback (
    IAHandle hIOPoint,
    IAStatus status,
    IABYTE buffer[],
    UInt32 bufferSize,
    IATimeStamp timeStamp,
    DWORD cParam);
```

Purpose

Serves as the prototype for the user-defined callback function that the asynchronous and synchronous callback completion mechanisms use. The arguments to the function include the I/O point used by the operation, the completion status of the operation, the buffer containing the data that the operation reads/writes, a timestamp when the operations reads/writes the data, and a user-defined parameter passed through when the operation is initiated.

Parameters

Name	Direction	Description
hIOPoint	In	Handle to I/O point.
status	In	Completion status of the operation.
buffer[]	In	Buffer used for data transfer.
bufferSize	In	Size of buffer.
timeStamp	In	Time data was read/written.
cParam	In	User-defined parameter passed through when the operation was initiated. The caller can use this to pass data that is meaningful or useful when processing the reply.

CheckDeviceStatus

```
IAStatus CheckDeviceStatus (
    IAString device,
    IADeviceStatus *status);
```

Purpose

Queries and reports the operational status of the indicated device. `status` returns the device status.

Parameters

Name	Direction	Description
device	In	A single addressable entity on a given device network.
status	Out	Pointer to device operational status code.

DeviceStatusMsg

```
IAStatus DeviceStatusMsg (
    IADeviceStatus status,
    char buffer[]);
```

Purpose

Copies a NULL-terminated ASCII message string corresponding to the device operational status code into `buffer`.

Parameters

Name	Direction	Description
<code>status</code>	In	Device operational status code.
<code>buffer</code>	Out	Client-supplied 256-byte buffer for error message text.

ErrorMsg

```
IAStatus ErrorMsg (
    IADeviceStatus status,
    char buffer[]);
```

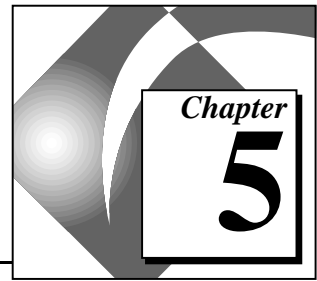
Purpose

Copies a NULL-terminated ASCII message string corresponding to the error code into `buffer`.

Parameters

Name	Direction	Description
<code>errorCode</code>	In	Device server error code.
<code>buffer</code>	Out	Client-supplied 256-byte buffer for error message text.

IAIO Base Class Reference



This chapter describes the base classes that form the architectural abstraction of an industrial automation device network. These classes comprise the foundation for customization of the IAIO server shell so it works with a particular device network.

Class CIAObject

Class `CIAObject` is a base class that contains services and properties often needed by other IAIO object classes derived from it. Those services include the following:

- Unique identification by way of a handle (`IAHandle`).
- Protected access services to ensure synchronized access and thread-safe behavior (`Lock()` and `Unlock()` methods).
- Reference count maintenance to ensure safe object deletion.

`CIAObject` works in conjunction with `CIAObjectMgr`, which manages the global `CIAObject`. The `CIAObject` table entries are references to `IAHandles`. The `CIAObjectMgr` ensures thread-safe deletion of `CIAObjects` from the object table.

Class CRequestQueue

Class `CRequestQueue` implements a simple protected access FIFO (first-in-first-out) queue of pointers to `CTask` objects. You can use it as a base class for many of the other queue classes found in the IAIO server shell.

Class `CRequestQueue` has the following important public methods, all of which derived classes can override.

Alert()

This method sets a signal (`m_evRequestAvailable`) to alert the class that one of its `CTask` elements has changed state and might require immediate attention.

Remove()

This method searches the queue for a specified pointer to a `CTask` object and removes it from the queue.

Select()

This method removes the head `CTask` object from the queue and returns a pointer to it. It resets `m_evRequestAvailable` if there are no more `CTask` objects in the queue.

Class CCommRsrcRequestQueue

This class derives from `CRequestQueue` and implements the default request queue of pending requests for `CCommMgr` classes. It overrides the `Select` method of `CRequestQueue` with one that takes into consideration `CTask` object scheduling requirements. To understand the default task scheduling algorithm completely, see the comments in the `CCommRsrcRequestQueue::Select()` method source code.

Class CCommMgr

Class `CCommMgr` is the default communications resource manager class. You can use it for serial access and allocation of a single communications resource in selected tasks. Each `CCommMgr` object is managed by its own high priority control thread.

Class `CCommMgr` has the following public methods of note, all of which derived classes can override.

CCommMgr()

`CCommMgr` is the base class constructor for `CCommMgr` objects. You can specify a different request queue other than the default `CCommRsrcRequestQueue` for the communications resource management object.

InitResource()

This method creates and initializes the control signals and thread of the `CCommMgr` object.

Alert()

This is a pass-through method to expose publicly the `Alert()` method of `CCommRsrcRequestQueue` for the object. You can override this method to set up any customized `CCommMgr` task submission behavior.

SubmitRequest()

This is primarily a pass-through method to expose publicly the `Submit()` method of `CCommRsrcRequestQueue` for the `CCommMgr` object. This method also increments the reference count for the `CTask` object and does task scheduling bookkeeping.

RemoveRequest()

`RemoveRequest` is a pass-through method to expose publicly the `Remove()` method of `CCommRsrcRequestQueue` for the `CCommMgr` object. You can override this method to set up any customized `CCommMgr` task removal behavior.

Read()

The `Read` method implements the communications resource low-level read operation.

Write()

The `Write` method implements the communications resource low-level write operation.

Class CCommRS232Mgr

This is a derived class of `CCommMgr` that works with RS-232 serial communications resource management. An instance of this class is created for each RS-232 port used by a device server.

Implement this class by overriding the `Read()` and `Write()` methods of the `CCommMgr` base class.

Class CDevice

Class `CDevice` implements the device (PLCs, sensors, actuators, and so on) abstraction of the device server architecture. The class maintains a reference (`m_comm`) to the communications resource manager used to access the device on the network. The class also maintains a private list of pending `CTask` objects that are awaiting service for that device, which you can use to implement customized batching algorithms.

You need to derive your own class abstractions from `CDevice` that model the capabilities and protocols that work with the devices found on their particular networks.

Class `CDevice` includes the following public methods of note, all of which derived classes can override.

AddPendingRequest()

This method adds a `CTask` object to the list (`m_requests`) of tasks that are awaiting service by the device.

RemovePendingRequest()

This method removes a specified `CTask` object pointer from the list (`m_requests`) of tasks that are awaiting service by the device.

ConstructReadPacket()

This method constructs a read packet that is appropriate for the device and protocol requirements of the target device network.

ConstructWritePacket()

This method constructs a write packet that is appropriate for the device and protocol requirements of the target device network.

Read()

This method initiates a `Read` operation on the device.

Write()

This method initiates a `Write` operation on the device.

Advise()

This method initiates an advise operation on the device.

SelectBatchRequests()

You can use this method as a hook for implementing customized batch optimization of pending operations (CTask objects) for a device. The server calls this method immediately before a task for this device is to be serviced. No action results from default implementation.

SelectBatchRequestsLateArrival()

You can use this method as a hook for implementing customized batch optimization of pending operations (CTask objects) for a device. This method verifies that a new task falls within the data read in the recently completed batch. Immediately after a task for this device has finished being serviced, the server calls `SelectBatchRequestsLateArrival`. Default implementation does nothing.

Class CIOPoint

Class `CIOPoint` implements the configured or available user-defined data items that are accessible on a device network. Each `CIOPoint` object defines a particular register, memory address, or otherwise addressable location on a particular device. A `CIOPoint` object represents a data item on a particular `CDevice` object.

You need to derive specific class abstractions from `CIOPoint` that model the capabilities and protocols supported by data items belonging to the devices found on their particular device networks.

The following is a description of the public methods of note, all of which derived classes can override.

Read()

This method initiates a `Read` operation on the I/O point.

Write()

This method initiates a `Write` operation on the I/O point.

Advise()

This method initiates an `Advise` operation on the I/O point.

ExecuteRead()

This method executes a `Read` operation on the I/O point.

ExecuteWrite()

This method executes a `Write` operation on the I/O point.

ExecuteResponse()

This method executes a response or acknowledgment of a protocol packet transmission.

ClientToDeviceTypeConversionNecessary()

This method returns `TRUE` if a client data type to device data type conversion of a source data buffer is necessary.

ApplyClientToDeviceTypeConversion()

This method performs a client data type to device data type conversion of a source data buffer into a target data buffer.

DeviceToClientTypeConversionNecessary()

This method returns `TRUE` if a device data type to client data type conversion of a source data buffer is necessary.

ApplyDeviceToClientTypeConversion()

This method performs a device data type to client data type conversion of a source data buffer into a target data buffer.

Class CTask

Class CTask is the base class used by a device server for management of I/O operations. An instance of a CTask object encapsulates the state, timing, and results of an operation as well as the methods necessary to perform that specific operation.

A CTask object is a self-contained entity in that when its virtual methods are invoked by the device server, the CTask object performs its specific operation. More than any other, this class exploits the features of C++ polymorphism to generalize the device server architecture.

Additionally, the control algorithm performs the operation polymorphically, even though the manner in which an I/O operation is executed differs among Read, Write, Times, and Advise. The C++ polymorphism causes the appropriate method to be invoked, so it can perform the expected behavior for the particular operation executed.

The classes CReadTask, CWriteTask, CAdviseTask, CBatchReadTask, and CBatchWriteTask are derived from CTask. Each class overrides the base CTask methods necessary to implement their specific operation behavior.

A task must maintain basic information about its data, status, and timestamp. This information is known as the *task state* and must be buffered for certain I/O operations. The advise operation must maintain information about the previous advise poll to compare with the results of the most recent advise poll based on the `fNotifyOnChange` flag. Because of the multi-threaded nature of the server, a data buffer must remain allocated until the server notifies the client of the most recent advise poll results. Otherwise, the communications resource management thread might overwrite those results with the next advise poll.

Methods generally fall into the categories of construction, I/O servicing, operation completion, status and buffering state maintenance, and aborting operations.

The following is a description of the CTask methods of note, all of which derived classes can override.

IsCompleted()

`IsCompleted()` is a predicate method returning `TRUE` if the task terminates and executes its completion notification mechanism.

IsTerminated()

`IsTerminated()` is a predicate method returning `TRUE` if the task terminates but not necessarily has executed its completion notification mechanism.

ServiceRequest()

This method carries out the I/O operation using the communications resource passed to it.

CompleteRequest()

This method executes the completion notification mechanism specified for the operation.

AbortRequest()

This method aborts the operation without allocating a new state for the task. It immediately executes the completion notification mechanism with the specified status code. `AbortRequest` typically is invoked when a new state cannot be allocated—that is, when `NewState()` fails.

Cleanup()

This method performs task operation bookkeeping cleanup management by decrementing the reference counts for the `CTask` object as well as the `CIOPoint` and `CCompletion` objects used by the `CTask` object.

Stop()

This method marks the `CTask` object for termination and sets the cause of termination (termination status).

Clear()

This method executes the `Clear()` operation on the task.

GetBuffer() and GetBufferSize()

These methods return a pointer to the buffer and the buffer size specified to the operation by the client.

GetCompletion() and GetCompletionParam()

These methods return a pointer to the completion notification object and the client specified completion parameter specified for the IAIO operation.

NewState()

This method allocates a new task state for the `CTask` object. It returns `FALSE` if allocation fails.

FreePreviousState()

This method frees the task state previous to the current state for reuse. Both `FreePreviousState` and `FreeCompletedState` must release a task state before it is available for reuse.

FreeCompletedState()

This method frees the oldest task state for reuse. Both `FreePreviousState` and `FreeCompletedState` must release a task state before it is available for reuse.

GetCurrentState()

This method retrieves the index to the current task state (`CTaskState`) in use by the `CTask` object.

GetPreviousState()

This method retrieves the previous task state used by the `CTask` object.

GetCompletedState()

This method retrieves the completed task state of the `CTask` object.

GetStateBuffer()

This method retrieves a pointer to the data buffer for a particular task state of the `CTask` object.

GetStatus() and SetStatus()

These methods get and set the operational status of a `CTask` object. By default the `GetStatus()` and `SetStatus()` operate on the current state buffer of a `CTask` object although you can optionally specify to which state buffer of the `CTask` you want the method to apply.

GetTerminationStatus() and SetTerminationStatus()

These methods access and modify the status code that cause a task to terminate.

ReadBuffer()

This method reads block data from the data buffer of a task state.

WriteBuffer()

This method writes block data from the data buffer of a task state.

CopyToClientBuffer()

This method copies the contents of the data buffer of a task state to the client buffer address passed as an argument to the IAIO API function. If necessary, this method invokes the `CIOPoint` method `ApplyDeviceToClientTypeConversion()` to convert from the device data type contents of the task state buffer into the client data type that the client buffer expects.

Class CTaskState

Class `CTask` uses Class `CTaskState`. You can use `CTaskState` to encapsulate one polled execution of an IAIO operation (for example, the `Advise()` poll). This class maintains data buffering, time stamp, task scheduling, and task status information.

This class is an *auxiliary class* to `CTask`, which means `CTaskState` helps `CTask` perform its functions. Do not modify it or use it in a derived class.

Class CReadTask

This class derives from `CTask` and implements the IAIO read operation. `CReadTask` implements the read operation by overriding the base class member functions `ServiceRequest()` and `CompleteRequest()`. `CReadTask::ServiceRequest()` constructs the protocol packet that sends a read request to a device on the network and acts upon any response packet from the device. The server invokes `CIOPoint::ExecuteRead()` and `CIOPoint::ExecuteResponse()` methods to implement the `read` operation on the specified I/O point.

Class CWriteTask

This class derives from `CTask` and implements the IAIO write operation. `CWriteTask` implements the write operation by overriding the base class member functions `ServiceRequest()` and `CompleteRequest()`. `CWriteTask::ServiceRequest()` constructs the protocol packet that sends a write request to a device on the network and acts upon any response packet from the device. The server invokes `CIOPoint::ExecuteWrite()` and `CIOPoint::ExecuteResponse()` methods to implement the `write` operation on the specified I/O point.

Class CAdviseTask

This class derives from `CTask` and implements the IAIO advise operation. `CAdviseTask` implements the advise operation by overriding the base class member functions `ServiceRequest()`, `CompleteRequest()`, `AbortRequest()`, `Cleanup()`, `Stop()`, `Clear()`, `NewState()`, `FreePreviousState()`, `FreeCompletedState()`, and `GetPreviousState()`. `CAdviseTask::ServiceRequest()` constructs the protocol packet that sends a write request to a device on the network and acts upon any response packet from the device.

`CAdviseTask::CompleteRequest()` checks whether the value of the I/O Point has changed from the previous value and then based upon the value of the `fNotifyOnChange` flag specified either notifies the client by executing the completion mechanism specified to the `Advise` operation or does nothing.

`CompleteRequest()` then resubmits the `CAdviseTask` to the device for the next read poll service. `CAdviseTask::Stop()` sets the

termination flag of the task and also notifies the communications resource request queue that the advise operation has been terminated. `Cleanup()`, `Clear()`, `NewState()`, `FreePreviousState()`, `FreeCompletedState()`, and `GetPreviousState()` are similar to the `CTask` base class methods they override. However, they treat the manipulation of the `CTaskState` buffer reference counts differently because a `CAdviseTask` can reference the same `CTaskState` from different threads.

`CAdviseTask` overrides the `CompleteRequest()` method to perform the following sequence of actions:

1. Compute the number of clock ticks to be used as a countdown until the next `advise read poll`.
2. If the `fNotifyOnChange` flag is `TRUE` and the I/O point value is different from its previous value, execute the specified completion mechanism to report the result and status of this `advise poll`.
3. If the `fNotifyOnChange` flag is `FALSE`, always execute the specified completion mechanism to report the result and status of this `advise poll`.
4. Free the previous task state buffer that was used to compare with the current I/O point value.
5. Unless the task has been terminated, resubmit the task (using the `CDevice::Advise()` method) for scheduling of the next `advise read poll`.

Class CBatchReadTask

This class derives from `CTask`. It implements a block read operation for several `CReadTask` and/or `CAdviseTask` objects.

`CBatchReadTask::ServiceRequest()` constructs the protocol packet that sends a read request to a device on the network and acts upon any response packet from the device.

`CBatchReadTask::CompleteRequest()` is implemented so that it copies the relevant portion from its block data buffer contents to the data buffers of the individual `CReadTask` and `CAdviseTask` objects that it is acting on behalf of. It then updates the status information for each and invokes the `CompleteRequest` method on each of these `CTask` objects.

Class CBatchWriteTask

This class derives from `CTask`. It implements a block write operation for several `CWriteTask` objects.

`CBatchWriteTask::ServiceRequest()` constructs the protocol packet that sends a write request to a device on the network and acts upon any response packet from the device.

`CBatchWriteTask::CompleteRequest()` updates the status information of the individual `CWriteTask` objects that it is acting on behalf of and then invokes the `CompleteRequest` method on each of these `CTask` objects.

Class CCompletion

Class `CCompletion` is the base class for the various task completion notification mechanisms. The derived classes `CAsynchCBCCompletion`, `CSynchCBCCompletion`, and `CMessageCompletion` implement the specific completion notification mechanism behavior.

National Instruments strongly recommends that you do not modify or derive from these classes.

The following is a description of the `CCompletion` methods of note, all of which derived classes can override.

Complete()

This method carries out the completion notification (either asynchronous callback, synchronizing callback, or message completion) of the `CCompletion` object.

CompleteWithError()

This method carries out the completion notification (either asynchronous callback, synchronizing callback, or message completion) of the `CCompletion` object along with specific status and time stamp for task object.

Class CAsynchCBCompletion

This class is a derived class of `CCompletion` and implements the asynchronous callback completion notification mechanism explained in the IAIO API document

Although each asynchronous completion mechanism created by a call to the IAIO API function `CreateAsynchCompletion()` results in the creation of a `CAsynchCompletion` object, all `CAsynchCompletion` objects share a common worker thread. This worker thread provides the execution context for the client-specified callback.

In addition, this worker thread (`AsynchCBWorkThread`) has an associated queue of pending `CAsynchCompletion` requests. When invoked, the `CAsynchCBCompletion::Complete()` method submits the asynchronous completion mechanism object to the queue of pending requests for `AsynchCBWorkThread()`. The only operation executed in the context of the `CCommMgr` control thread is the submission of the `CAsynchCompletion` object to the pending request queue for `AsynchCBWorkThread()`.

Executing client callbacks in the `AsynchCBWorkThread()` protects the `CCommMgr` control thread from executing client callback code. This code might perform lengthy operations or operations that block or otherwise tie up communications resources managed by the `CCommMgr` thread.

Do not modify this class because it fully implements the behavior specified in the IAIO API for an asynchronous callback completion mechanism.

Class CSynchCBCompletion

This class is a derived class of `CCompletion` and implements the synchronizing callback completion notification mechanism.

The synchronizing completion mechanism (`CSynchCompletion`) requires the client to synchronize its execution by way of the IAIO API `Wait()` or `WaitForGroup()` function with the execution of the callback notification. A `CClient` object is created for each client thread attached to the device server. The `CClient` object maintains a queue of pending synchronous completion mechanisms for the client thread that it represents.

When a client thread calls `Wait()` or `WaitForGroup()`, the queue for the `CClient` object is checked for any pending synchronous completion callbacks for that client thread. Any synchronous completion callbacks present then execute in the context of the client thread before returning from `Wait()` or `WaitForGroup()`.

The same scenario exists for the synchronizing completion mechanism as it does for the asynchronous completion mechanism; the `CCommMgr` control thread needs to be protected from execution of client callback routines. The `CSynchCBCompletion::Complete()` merely submits a `CSynchCompletion` object to the appropriate `CClient` queue of pending completion requests. The client thread that originally initiated the I/O operation is the context in which the synchronous completion callback is executed.

Do not modify this class because it fully implements the behavior specified in the IAIO API for a synchronizing callback completion mechanism.

Class CMessageCompletion

This class is a derived class of `CCompletion` and implements the message callback completion notification mechanism explained in the IAIO API document.

The most simple completion mechanism is `CMessageCompletion`. The `CMessageCompletion::Complete()` method executes the message completion mechanism.

The only unique action it takes is to post the client-specified notification message to the client thread or window that originally initiated the I/O operation. Posting a message does not require the operation to block waiting for a client response. In addition, it does not depend on the execution of any client callback routine that potentially blocks execution. Therefore, the message completion mechanism can take place entirely in `CCommMgr` control thread.

Do not modify this class because it fully implements the behavior specified in the IAIO API for a synchronizing callback completion mechanism.

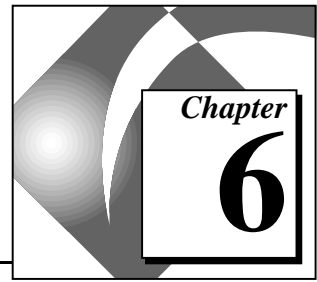
Class CClient

Class `CClient` represents each application thread that is a client of a device server. The primary use of a `CClient` object is in `Wait` synchronization and managing the execution of synchronizing callback completion notification mechanisms.

Class `CClient` maintains a protected access FIFO queue of pending synchronizing callback completion. The queue executes when the client thread synchronizes by calling either `Wait()` or `WaitForGroup()`.

National Instruments strongly recommends that you do not modify or derive from this class.

Server Customization



This chapter describes the classes, methods, and global functions you can modify to customize your device server.

Five main base classes comprise the device server kernel functionality. They are `CIOPoint`, `CDevice`, `CTask`, `CCompletion`, and `CCommMgr`. As a general principle, never customize your server by changing any of the base classes. Accomplish customization primarily through class derivation and virtual method override. Control specific server behavior in subclass methods that override the base class methods.

Typically, when customizing a device server, you need to derive from the base classes `CIOPoint` and `CDevice`. These classes represent the devices and items of interest on the network and construct the protocol packets necessary to communicate over the network.

If you are using serial communication to access the devices on the network then the derived `CRS232CommMgr` class will probably suffice. However, if you are accessing devices through a plug-in card in your host computer, then you need to write a new derived subclass (from `CCommMgr`) that supports communications through your plug-in card.

Under normal conditions, you do not need to derive or modify the `CTask` or `Completion` classes or derived subclasses. Their behavior implements the semantics of the IAIO API.

Global Functions That Can Be Overridden

You need to set up implementations of the following global constructor functions for any derived classes of `CDevice`, `CIOPoint`, or `CCommMgr`. The IAIO server shell invokes these global constructor functions during initialization. Device server initialization uses these functions to build the device network abstraction that the device server configuration contains.

ConstructIOPoint

You must implement `ConstructIOPoint` for any derived `CIOPoint` class. If there is more than one derived `CIOPoint` class, the `ConstructIOPoint` implementation determines which subclass constructor is the appropriate one to invoke, based on the information present in the Common Configuration Database (CCDB) record set pointer. The default implementation invokes the `CIOPoint` constructor passing a pointer to the current CCDB record set for an item in the server configuration CCDB.

ConstructDevice

You must implement `ConstructDevice` for any derived `CDevice` class. If there is more than one derived `CDevice` class, the `ConstructDevice` implementation determines which subclass constructor is the appropriate one to invoke, based on the information present in the CCDB record set pointer. The default implementation invokes the `CDevice` constructor passing a pointer to the current CCDB record set for a device in the server configuration CCDB.

ConstructCommMgrResource

You must implement `ConstructCommMgrResource` for any derived `CCommMgr` class. If there is more than one derived `CCommMgr` class, the `ConstructCommMgrResource` implementation determines which subclass constructor is the appropriate one to invoke based on the information present in the CCDB record set pointer. The default implementation invokes the `CCommMgr` constructor passing a pointer to the current CCDB record set for a device in the server configuration CCDB.

Deriving from CDevice

You can derive your own `CDevice` subclass to implement device-specific or protocol-specific behavior. For example, you might have several different types of devices existing on a network. Although they all communicate using the same protocol, the devices have disparate capabilities. One device might have a different item address range or work with different data types than another. In addition, the device might work with a different operation set. Because of these factors and others, the way a server performs an I/O operation (`read` or `write`) might differ from one device type to another.

By using the virtual methods of the `CDevice` base class, you can override their default behavior with an implementation that matches the capabilities of the device with which they are trying to work.

You might need to construct a different protocol packet for each device type. In that situation, you can override the `ConstructReadPacket()` and `ConstructWritePacket()` in a `CDevice` subclass.

Optional hooks exist in the `CDevice` base class for performing optimization by batching requests for a device into a block operation. The `CDevice` base class maintains a list of pending I/O operations (from `CTask`) for the device in the member variable `m_requests`. `CDevice` has two methods—`SelectBatchRequests()` and `SelectBatchRequestsLateArrival()`. They do nothing in the default case.

However, you can override them in any subclass to the extent that they select `CTasks` suitable for batching with the currently executing task. This selection comes from the pending request list for the device (`m_requests`).

This behavior belongs in a `CDevice` subclass because it is usually device-specific in that requests are within a certain address range of one another. Because the device manages the I/O points, it has access to all the addressing information of these I/O points. Therefore it can best determine if and how I/O points should be batched.

Deriving From `CIOPoint`

As with `CDevice`, you can decide to derive from `CIOPoint` for many of the same reasons.

Although `CIOPoint` uses client/device type data conversion methods (`ClientToDeviceTypeConversionNecessary()`, `ApplyClientToDeviceTypeConversion()`, `DeviceToClientTypeConversionNecessary()`, `ApplyDeviceToClientTypeConversion()`), you can support data types other than the default set. Also, you might discover that the device data type is in some vendor proprietary data type format that must be converted to a standard type.

Deriving from CCommMgr

A `CCommMgr` object control thread executes the scheduling, resource allocation, and operation sequencing for a task. In this way, the set of `CCommMgr` object control threads comprise the core control algorithms for a device server.

The name of the control thread method for a `CCommMgr` object is `ControlThread()`. The default base class implementation of `ControlThread()` performs serial scheduling and allocation for an abstract communication device. Serial access means that a single `CCommMgr` object representing one particular instance of an abstract communications resource can be used by a single task at a time. A task uses its I/O methods to read and write the communication packets necessary to perform operations on the device accessed through this communications resource. When the task finishes with its communications resource, it releases the communications resource to become available for the next selected task.

Several `CCommMgr` objects, each with its own controlling thread, can service communications resources concurrently. In the base class implementation, each `CCommMgr` object can be used by only one task at a time.

Because the `CCommMgr` base class is an abstraction of a serial communications resource, it is useful only as a model. A more useful implementation and good example of customized class derivation is found in the class `CCommRS232Mgr`. `CCommRS232Mgr` implements communications resource management for a single RS-232/485 port. Because `CCommRS232Mgr` manages a serially-accessed device and is derived from `CCommMgr`, you need to override only a few of the base class methods to implement the RS-232/485 behavior you want.

`CCommRS232Mgr` has its own constructor to implement proper opening and initialization of a RS-232/485 serial port. `Read()` and `Write()` overrides also implement the serial read and write operations to the communication port. Finally, the `CCommRS232Mgr` destructor performs the proper closing and shutdown of the RS-232/485 port.

You can have a device server that contains a variety of communications resources. Then, each has its own control behavior encapsulated in a derived `CCommMgr` class. If you choose to derive your own subclass from `CCommMgr`, the method that you most likely need to override is the `ControlThread()`.

You must override `ControlThread()` for a nonserial communications resource. However, if your communications resource permits serial access, you should override only the basic I/O methods `Read()` and `Write()`.

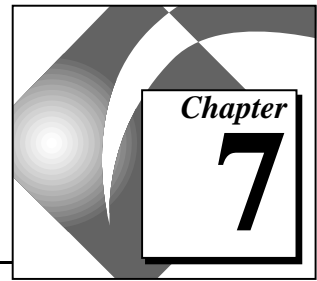
Deriving from `CRequestQueue`

The `m_requests` member of class `CCommMgr` is a pointer to the base class `CRequestQueue`. Member `m_requests` is the queue of pending tasks awaiting service by the `CCommMgr` object. Although `m_requests` points to the base class `CRequestQueue`, it actually is instantiated as a `CCommRsrcRequestQueue` object, which is a derived class of `CRequestQueue`. `CCommMgr` polymorphically invokes the methods of `m_requests` to get the appropriate queue behavior.

The `CCommMgr` constructor takes as an optional argument a pointer to a `CRequestQueue` object. The `CRequestQueue` object is assigned as the `m_request` member for the `CCommMgr` object being constructed.

You can override the request queue behavior of a `CCommMgr` device. Supply a `CRequestQueue` subclass that implements the request queue behavior you want, and pass it to the `CCommMgr` constructor. As a result, the `CRequestQueue::Select()` method implements the task scheduling and selection algorithm for the communications resource. If you want different scheduling and selection or different task submission behavior, you can derive a class in which the default methods are overridden.

IAIO Configuration Reference



This chapter explains the framework of the IAIO Configuration API, including descriptions of its functions.

The Framework of the IAIO Configuration API

The IAIO Configuration API contains a framework in which you can manage and configure a device server and its device network. Because many communications protocols, hardware devices, and data collection methods exist, each device server and its device network usually have configuration data and terminology specific to it.

To facilitate these diverse configuration needs, each device server has a device server configuration DLL that conforms to the IAIO Configuration API.

The primary functions for configuration are `RegisterOBJECT` and `EditOBJECT` where `CommResource`, `Device`, `Item`, or `Server` can be substituted for `OBJECT`. `UnRegisterOBJECT` deletes configured objects and the `GetOBJECTList` procedure returns a list of the currently configured objects for the device server.

GetSupportedProcedures

```
IAStatus GetSupportedProcedures(unsigned long* SupportedProcedures);
```

Purpose

`GetSupportedProcedures` returns the procedures that work with the device server configuration. Each bit within the `SupportedProcedures` parameter represents one IAIO Configuration procedure.

The following table shows the procedures and their corresponding decimal values.

Table 7-1. Supported Procedures and Corresponding Decimal Values

Procedure	Decimal Value
DUPLICATE_COMM_RESOURCE	1
DUPLICATE_DEVICE	2
DUPLICATE_ITEM	4
EDIT_COMM_RESOURCE	8
EDIT_DEVICE	16
EDIT_ITEM	32
EDIT_SERVER	64
GET_COMM_RESOURCE_LIST	128
GET_DEVICE_LIST	256
GET_ITEM_LIST	512
REGISTER_COMM_RESOURCE	1024
REGISTER_DEVICE	2048
REGISTER_ITEM	4096
REGISTER_SERVER	8192
UNREGISTER_COMM_RESOURCE	16384
UNREGISTER_DEVICE	32768
UNREGISTER_ITEM	65536
UNREGISTER_SERVER	131072

DuplicateCommResource

```
IAStatus DuplicateCommResource (
    LPCTSTR CommResourceName,
    long nCommResources);
```

Purpose

The `DuplicateCommResource` procedure duplicates a previously configured IAIO communications resource in the active CCDB.

DuplicateDevice

```
IAStatus DuplicateDevice (
    LPCTSTR DeviceName,
    long nDevices);
```

Purpose

The `DuplicateDevice` procedure duplicates a previously configured device in the active CCDB. In addition, it duplicates all of the items associated with the selected device.

DuplicateItem

```
IAStatus DuplicateItem (
    LPCTSTR DeviceName,
    LPCTSTR ItemName,
    long nItems);
```

Purpose

The `DuplicateItem` procedure duplicates a previously configured item in the active CCDB.

EditCommResource

```
IAStatus EditCommResource (LPCTSTR CommResourceName);
```

Purpose

The `EditCommResource` procedure edits a previously configured communications resource. If you change the name of the communications resource, the returned contents of `CommResourceName` reflect that change. The default behavior of the procedure is to

create a server-specific comm resource dialog box pre-configured with the attributes of the specified communications resource.

EditDevice

```
IAStatus EditDevice (LPTSTR DeviceName);
```

Purpose

The `EditDevice` procedure edits a previously configured device. If you change the name of the device, the returned contents of `DeviceName` reflect that change. The default behavior of the procedure is to create a server-specific device dialog box pre-configured with the attributes of the specified device.

EditItem

```
IAStatus EditItem (
    LPTSTR DeviceName,
    LPTSTR ItemName,
    ColeVariant *ItemAttributes);
```

Purpose

The `EditItem` procedure edits a previously configured item. If you change the device associated with item, the returned `DeviceName` contains the name of the new device. If you change the name of the item, the returned contents of `ItemName` reflect the change. The default behavior of the procedure is to create a server-specific item dialog box pre-configured with the attributes of the specified item. The `ItemAttributes` parameter is reserved for use by National Instruments.

EditServer

```
IAStatus EditServer (LPTSTR ServerName);
```

Purpose

The `EditServer` procedure edits the server. If you change the name of the server, the returned contents of `ServerName` reflect the change.



Note: *Do not alter `ServerName` within the `ServerName` procedure. This procedure is NOT called by the Server Explorer.*

GetCommResourceList

```
IAStatus GetCommResourceList (ColeVariant *CommResourceList);
```

Purpose

The `GetCommResourceList` returns a `SAFEARRAY` of `VARIANTS` that contain the names of the communications resources used by the server. If no communications resources are currently defined the `.vt` flag for the `CommResourceList` parameter is set to `VT_EMPTY`.

GetDeviceList

```
IAStatus GetDeviceList (ColeVariant *DeviceList);
```

Purpose

The `GetDeviceList` returns a `SAFEARRAY` of `VARIANTS` that contain the names of the devices used by the server. If no devices currently are defined the `.vt` flag for the `DeviceList` is set to `VT_EMPTY`.

GetItemList

```
IAStatus GetItemList (
    LPCTSTR DeviceName,
    ColeVariant *ItemList);
```

Purpose

The `GetItemList` returns a `SAFEARRAY` of `VARIANTS` that contains the name of the items defined on the device specified by `DeviceName`. If no items are currently defined for the device, the `.vt` flag for the `ItemList` is set to `VT_EMPTY`.

RegisterCommResource

```
IAStatus RegisterCommResource (LPTSTR CommResourceName);
```

Purpose

The `RegisterCommResource` procedure creates a new communications resource in the active CCDB. The default behavior of the procedure is to create a server-specific comm resource dialog box for configuring the attributes of the new communications resource. The returned `CommResourceName` parameter then contains the name of the newly created communications resource.

RegisterDevice

```
IAStatus RegisterDevice (LPTSTR DeviceName);
```

Purpose

The `RegisterDevice` procedure creates a new device in the active CCDB. The default behavior of the procedure is to create a server-specific device dialog box for configuring the attributes of the new device. The returned `DeviceName` parameter then contains the name of the newly created device.

RegisterItem

```
IAStatus RegisterItem (
    LPTSTR DeviceName,
    LPTSTR ItemName,
    ColeVariant *ItemAttributes);
```

Purpose

The `RegisterItem` procedure creates a new item in the active CCDB. If you change the `DeviceName` for the item, the returned contents of `DeviceName` reflect the change and the returned contents of `ItemName` contain the name of the newly created item. The default behavior of the procedure is to create a server-specific item dialog box for configuring the attributes of the new item. Upon completion, the `DeviceName` parameter then contains the name of the device with which the item is associated. The `ItemName` parameter contains the name of the item that was created. The `ItemAttributes` parameter is reserved for use by National Instruments.

RegisterServer

```
IAStatus RegisterServer (LPCTSTR ActiveDatabase);
```

Purpose

With the `RegisterServer` procedure, a server can register its configuration parameters into the database specified by the `ActiveDatabase` parameter. Clients must precede any calls to the IAIO Configuration API with a call to `RegisterServer`. If the `ActiveDatabase` parameter is the empty string and the `IManager` automation interface already has opened a database session, the server registers itself in the currently active database.

Keep in mind that if the `IManager` interface has opened a session with a database using the `ActiveDatabase` parameter other than the one specified, `RegisterServer` returns a status indicating failure.

UnRegisterCommResource

```
IAStatus UnRegisterCommResource (LPCTSTR CommResourceName);
```

Purpose

The `UnRegisterCommResource` procedure removes the specified communications resource from the active CCDB.

UnRegisterDevice

```
IAStatus UnRegisterDevice (LPCTSTR DeviceName);
```

Purpose

The `UnRegisterDevice` procedure removes the specified device from the active CCDB.

UnRegisterItem

```
IAStatus UnRegisterItem (  
    LPCTSTR DeviceName,  
    LPCTSTR ItemName);
```

Purpose

The `UnRegisterItem` procedure removes the specified item from the active CCDB.

UnRegisterServer

```
IAStatus UnRegisterServer (void);
```

Purpose

The `UnRegisterServer` procedure removes the server from the active CCDB.

IAIO Configuration Customization



This chapter explains how to customize configuration dialog boxes supplied with the IAIO Configuration API framework.

The Configuration Dialog Box Classes

The framework contains three default dialog box classes, `CCommDialog`, `CDeviceDialog`, and `CItemDialog`. If your server conforms to this typical device network topology, only alter the aforementioned classes. Each dialog box contains a default behavior and a good framework in which to add the specific configuration needs of your IAIO Server.

CCommDialog

The `CCommDialog` class contains the configuration logic needed for a user to create and edit communications resources and their attributes graphically.

CCommDialog Member Variables

- `m_create`—Indicates that the dialog box is creating a new communications resource if `TRUE`. A `FALSE` value indicates that the dialog box is editing a previously existing communications resource.
- `m_name`—Contains the name of the communications resource the user enters or is specified by the client application that calls it.

- `m_status`—Specifies the status of the dialog box to the IAIO configuration framework.
 - `IA_SUCCESS`—The user successfully created or edited a communications resource and all dialog box member variables contain valid data.
 - `IA_ERROR_CFG_OBJECT_INVALID`—The specified object was invalid. This status value can occur when the object name passed in by the client application is invalid.
 - `IA_ERROR_CFG_NOOP`—The user clicks on the **Cancel** button on the communications resource dialog box.

CCommDialog Methods

Following are descriptions of the CCommDialog methods that you most likely need to understand and modify.

CCommDialog()

This constructor initializes the member variables of the dialog boxes to their default values and sets the `m_create` flag to `TRUE`. The `RegisterCommResource` procedure calls this constructor.

CCommDialog (CString CommResourceName)

This constructor initializes the member variables of the dialog boxes to their default values, with the exception of `m_name`, which is set to `CommResourceName` and sets the `m_create` flag to `FALSE`. The `EditCommResource` procedure calls this constructor.

OnInitDialog()

This procedure is called when the `DoModal` method is invoked by the IAIO configuration framework in either the `RegisterCommResource` or `EditCommResource` procedures. The procedure behavior is dependent on the value of `m_create`. If you create the dialog box to register a new communications resource, the procedure updates the dialog box controls with their default values and continues execution.

If the dialog box was created to edit an existing communications resource, this method extracts the configuration information from the active CCDB through the appropriate automation interface. The default implementation uses the `ISerial` interface.

OnOk()

The behavior of the `OnOk` procedure depends on the value of `m_create`. If `m_create` is `TRUE`, that is the dialog box is creating a new item, the automation call to create a new resource in the active CCDB is made: `SetResource (TRUE, m_name)`. Otherwise, the `SetResourceName (m_name)` call is made. Execution then continues normally with the attributes of the communications resource stored to the active CCDB.

OnCancel()

The `OnCancel` procedure simply sets the value of `m_status` to `IA_ERROR_CFG_NOOP`.

CDeviceDialog

The `CDeviceDialog` class contains the configuration logic needed for a user to create and edit devices and their attributes graphically.

CDeviceDialog Member Variables

- `m_resourceComboBox`—The combo box dialog control that contains the list of communications resources.
- `m_create`—Indicates that the dialog box is creating a new device if `TRUE`. A `FALSE` value indicates that the dialog box is editing a previously existing device.
- `m_name`—Contains the name of the device you enter or that the calling client application specifies.
- `m_status`—Specifies the status of the dialog box to the IAIO configuration framework.
 - `IA_SUCCESS`—The user successfully created or edited a device and all dialog box member variables contain valid data.
 - `IA_ERROR_CFG_OBJECT_INVALID`—The specified object was invalid. This status value occurs when the object name passed in by the client application is invalid.
 - `IA_ERROR_CFG_NOOP`—This status is specified when the user clicks on the **Cancel** button on the device dialog box.
- `m_resource`—The name of the communications resource associated with the device.

CDeviceDialog Methods

Following are descriptions of the CDeviceDialog methods that you most likely might need to understand and modify.

CDeviceDialog()

This constructor initializes the dialog box member variables to their default values and sets the `m_create` flag to `TRUE`. The `RegisterDevice` procedure calls this constructor.

CDeviceDialog (CString DeviceName)

This constructor initializes the member variables of the dialog boxes to their default values, with the exception of `m_name` which is set to `DeviceName` and sets the `m_create` flag to `FALSE`. The `EditDevice` procedure calls this constructor.

OnInitDialog()

This procedure is called when the IAIO configuration framework invokes the `DoModal` method in either the `RegisterDevice` or `EditDevice` procedures. The behavior is dependent on the value of `m_create`. If you create the dialog box to register a new device, the procedure updates the dialog box controls with their default values and continues execution.

If, however, you create the dialog box to edit an existing device, this method extracts the configuration information from the active CCDB through the `IIAIODevices` interface. In addition, this procedure makes use of the `GetCommResourceList` procedure to build the list of registered communications resources with which to populate the `m_resourceComboBox` dialog box control.

OnOk()

The behavior of the `OnOk` procedure depends on the value of `m_create`. If the `m_create` is `TRUE`, the automation call to create a new device in the active CCDB is made: `SetDevice (TRUE, m_name)`, otherwise the `SetDeviceName (m_name)` call is made. After this, the active CCDB stores the rest of the attributes of the device.

OnCancel()

The `OnCancel` procedure simply sets the value of `m_status` to `IA_ERROR_CFG_NOOP`.

CItemDialog

The `CItemDialog` class contains the configuration logic needed for a user to create and edit items and their attributes graphically.

CItemDialog Member Variables

- `m_deviceComboBox`—The combo box dialog control that contains the list of devices registered for the IAIO Server.
- `m_create`—Indicates that the dialog box is creating a new item if `TRUE`. A `FALSE` value indicates that the dialog box is editing a previously existing item.
- `m_name`—Contains the name of the item as entered by the user or specified by the calling client application.
- `m_status`—Specifies the status of the dialog box to the IAIO configuration framework.
 - `IA_SUCCESS`—The user successfully created/edited a comm resource and all dialog box member variables contain valid data.
 - `IA_ERROR_CFG_OBJECT_INVALID`—The specified object was invalid. This status value occurs when the object name passed in by the client application is invalid.
 - `IA_ERROR_CFG_NOOP`—This status would be specified when the user clicks on the **Cancel** button on the device dialog box.
- `m_device`—The name of the device upon which the item resides.

CItemDialog Methods

Following are descriptions of the `CItemDialog` methods that you will most likely need to understand and modify.

CItemDialog(CString DeviceName)

This constructor initializes the member variables of the dialog boxes to their default values, with the exception of `m_device` which is set to `DeviceName` and sets the `m_create` flag to `TRUE`. The `RegisterItem` procedure calls this constructor.

CItemDialog (CString DeviceName, CString ItemName)

This constructor initializes the member variables of the dialog boxes to their default values, with the exceptions of `m_name` which you set to `ItemName` and `m_device` which you set `DeviceName`. In addition, you set the `m_create` flag `FALSE`. The `EditItem` procedure calls this constructor.

OnInitDialog()

This procedure is called when the IAIO configuration framework invokes the `DoModal` method in either the `RegisterItem` or `EditItem` procedures. The procedure behavior is dependent on the value of `m_create`. If the dialog box was created to register a new item, the procedure updates the dialog box controls with their default values and continues execution.

If you create the dialog box to edit an existing item, this method extracts the configuration information from the active CCDB through the `IIAIOItems` interface. In addition, this procedure makes use of the `GetDeviceList` procedure to build the list of registered devices with which to populate the `m_deviceComboBox` dialog box control.

OnOk()

The `OnOk` procedure behavior depends on the value of `m_create`. If the `m_create` is `TRUE`, the automation call to create a new item in the active CCDB is made (`SetItem(TRUE, m_device, m_name)`). Otherwise the `SetItemName(m_name)` call is made. After this, the active CCDB stores the attributes of the rest of the items.

OnCancel()

The `OnCancel` procedure simply sets the value of `m_status` to `IA_ERROR_CFG_NOOP`.

IAIO Configuration API Behavior

The behavior of each procedure in the IAIO configuration API is related to configuration dialog classes as the following sections describe.

DuplicateCommResourceList

`DuplicateCommResourceList` currently returns a status of `IA_ERROR_CFG_UNSUPPORTED`.

EditCommResource

The default behavior of the `EditCommResource` procedure instantiates an instance of the `CCommDialog` calling the `CCommDialog(CString CommResourceName)` constructor. The framework then calls `DoModal` on the new `CCommDialog` object. When you exit the `CCommDialog` by clicking on **OK**, the framework copies the user-supplied name of the communications resource, `m_name`, into the `CommResourceName` parameter. The contents of the `CCommDialog` member variable `m_status` is the return value for this procedure.

GetCommResourceList

The `GetCommResourceList` procedure performs a SQL query on the active CCDB using the `IManager::Query` method. The default implementation uses `SELECT [resource name] FROM [%s]` where `%s` is the name of the CCDB table that your server uses.

RegisterCommResource

The default behavior of the `RegisterCommResource` procedure instantiates the `CCommDialog` class calling the `CCommDialog()` constructor. The framework then calls `DoModal` on the new `CCommDialog` object. When you exit the `CCommDialog` by clicking on **OK**, the framework copies the user-supplied name of the communications resource, `m_name`, into the `CommResourceName` parameter. The contents of the `CCommDialog` member variable `m_status` is the return value for this procedure.

DuplicateDevice

`DuplicateDevice` currently returns a status of `IA_ERROR_CFG_UNSUPPORTED`.

EditDevice

The default behavior of the `EditDevice` procedure instantiates the `CDeviceDialog` class calling the `EditDevice(CString DeviceName)` constructor. The framework then calls `DoModal` on the new `CDeviceDialog` object. When you exit the `CDeviceDialog` by clicking on **OK**, the framework copies the user-supplied name of the device, `m_name`, into the `DeviceName` parameter. The contents of the `CDeviceDialog` member variable `m_status` is the return value for this procedure.

RegisterDevice

The default behavior of the `RegisterDevice` procedure instantiates the `CDeviceDialog` class calling the `CDeviceDialog()` constructor. The framework then calls `DoModal` on the new `CDeviceDialog` object. When you exit the `CDeviceDialog` by clicking on **OK**, the framework copies the user-supplied name of the device, `m_name`, into the `DeviceName` parameter. The contents of the `CDeviceDialog` member variable `m_status` is the return value for this procedure.

DuplicateItem

The `DuplicateItem` procedure currently creates `n` copies of the client-specified item using iterative brute force.

EditItem

The default behavior of the `EditItem` procedure instantiates the `EditItem` class calling the `EditItem(CString DeviceName, CString ItemName)` constructor. The framework then calls `DoModal` on the new `CItemDialog` object. When you exit the `CItemDialog` by clicking on **OK**, the framework copies the user-supplied name of the device, `m_name`, into the `ItemName` parameter and the `m_device` parameter into the `DeviceName` parameter. The contents of the `CItemDialog` member variable `m_status` is the return value for this procedure.

GetItemList

The `GetItemList` procedure performs a SQL query on the active CCDB using the `IManager::Query` method. The default implementation uses `SELECT [item name] FROM [items] WHERE [server] = '%s' AND [device] = '%s'` where the first `%s` is the

name of your server and the second %s is the name of the device whose item list the client wishes to retrieve.

RegisterItem

The default behavior of the `RegisterItem` procedure instantiates the `CItemDialog` class calling the `CItemDialog(CString DeviceName)` constructor. The framework then calls `DoModal` on the new `CItemDialog` object. When you exit the `CItemDialog` by clicking on **OK**, the framework copies the user-supplied name of the device, `m_name`, into the `ItemName` parameter and the `m_device` parameter into the `DeviceName` parameter. The contents of the `CItemDialog` member variable `m_status` is the return value for this procedure.

Common Configuration Database Reference



This chapter describes the configuration database tables and how to use the server automation and interfaces to access the data stored in those tables.

Using the Configuration Database

The Common Configuration Database is a fully relational database that conforms to the Microsoft Database (MDB) format. CCDB files serve as the centralized repository for configuration information for IAIO servers and, in a limited capacity, the BridgeVIEW Supervisory Control and Data Acquisition (SCADA) software product.

Each CCDB file contains six database tables: proxies, servers, devices, items, serial, and generic resources. These tables allow for a complete definition of the attributes of a device server and its associated device network. Seven automation interfaces manage, store, and retrieve CCDB files and server configuration information. The automation interfaces IIAIOProxy, IIAIOServers, IIAIODevices, IIAIOItems, ISerial, and IGenericResources map to the six tables defined in the CCDB with the exception of the IManager interface which handles the opening and closing of CCDB files in addition to general maintenance operations. Keep in mind that you can create and maintain multiple CCDB files, but you only can designate one file as the active CCDB during execution.

Active CCDB

The active CCDB is the CCDB file that all the device servers and the BridgeVIEW SCADA product use for their configuration data. The active CCDB is specified by a value stored in the system registry: `HKEY_LOCAL_MACHINE\SOFTWARE\National Instruments\NI-Servers\Active CCDB`. The following illustration shows the active CCDB as viewed from the system utility `Regedit.exe` for Windows 95 and `regedt32.exe` for Windows NT.

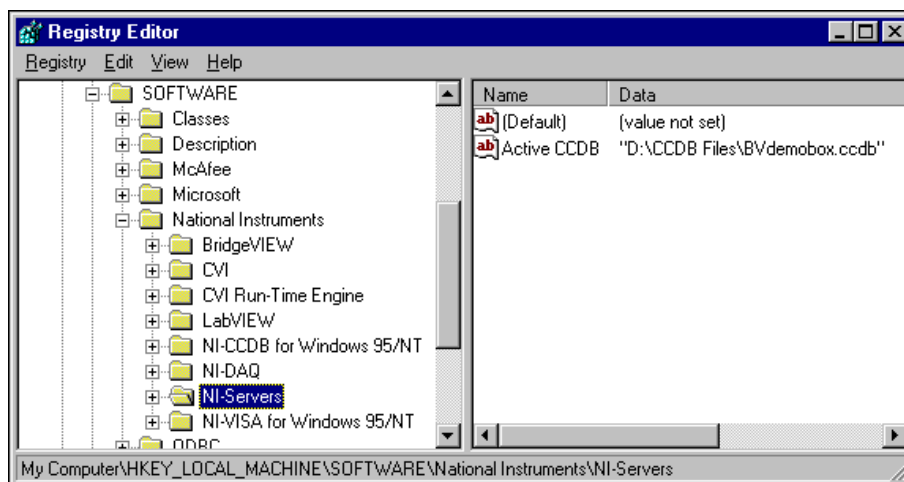


Figure 9-1. The Active CCDB in the Windows System Registry

CCDB Features

Each table contains information specific to its title. In other words, information in the `servers` table is about servers. As in all databases, there must be some sort of unique row identifier called a *primary key*. The following table lists the tables in the CCDB and their corresponding primary keys.

Table 9-1. Tables and Primary Keys

Table Name	Primary Keys
proxies	proxy name
servers	server name
devices	device name, server
items	item name, server, device
serial	resource name
generic resources	resource name, type

CCDB is designed so that all servers have devices and all devices have items. *Enforced referential integrity* maintains this relationship among the tables. Enforced referential integrity is the enforcement of a relationship between two tables based on primary and *foreign keys*. A foreign key is a field in a table which is based on the primary key of another table. For example, the primary key in the `servers` table, `server name`, is specified as the foreign key `server` in the `devices` table. As a result, the `devices` table does not accept input rows whose `server` field does not have a matching entry in the `servers` table.

Table 9-2. Tables and Foreign Keys

Table Name	Foreign Key
servers	None
devices	server name from servers table
items	server name and device name from devices table

Keep in mind that the CCDB performs *cascading updates* and *deletes*. When a row is deleted or updated, all rows in other tables, which contain the primary key as a foreign key also are deleted or updated.

CCDB Tables

Following are descriptions of each of the six database tables common to all CCDB files: proxies, servers, devices, items, serial, and generic resources. With these tables, the attributes of a device server and its associated device network are described completely.

The Proxies Table

The `proxies` table contains information only relevant to BridgeVIEW. For BridgeVIEW to communicate with certain kinds of servers, it uses a proxy layer. The proxy layer handles the necessary tasks of integrating out of environment servers into BridgeVIEW.

Table 9-3. The Proxies Table Definition

Attributes	Data Type	Size	Required	Allow Zero Length
proxy name	Text	255	True	False
type	Text	255	True	False
launch path	Text	255	False	True
configuration path	Text	255	False	True

`proxy name`—The unique name identifying the proxy.

`type`—The type of the proxy. For example, IAIO or DDE.

`launch path`—The path to the VI to load and execute when a server of type `type` is required by BridgeVIEW.

`configuration path`—The path to the VI, EXE, or DLL to load to configure specific attributes of the proxy.

Any server that requires the use of a proxy in BridgeVIEW must set the `type` field of the server to correspond with that of a registered proxy. If you do not take this step, the server cannot be launched from BridgeVIEW.

- **Primary Keys**—The `proxy name` is the primary key for the `proxies` table.
- **Foreign Keys**—None.

The Servers Table

The `servers` table describes the attributes of a server for BridgeVIEW.

Table 9-4. The Servers Table Definition

Attributes	Data Type	Size	Required	Allow Zero Length
server name	Text	255	True	False
can add devices	Bool	2 bytes	True	False
type	Text	255	False	True
launch path	Text	255	False	True
configuration path	Text	255	False	True
specific info	Bin	1.2 Gig	False	False

`server name`—The unique name identifying the server.

`can add devices`—A Boolean indicating whether devices can be added dynamically to the `devices` table for this server. TRUE represents yes and FALSE indicates no.

`type`—The type of the server: DDE, TCP/IP, DAQ, VI, or IAIO.

`launch path`—The path, including the executable name, to the server executable.

`configuration path`—The path, including the executable name, to the server configuration utility.

`specific info`—A binary field for developer-specific information.

- **Primary Keys**—The `server name` is the primary key for the `servers` table.
- **Foreign Keys**—None.

The Devices Table

The `devices` table describes the attributes of devices within the IAIO servers device network hierarchy.

Table 9-5. The Devices Table Definition

Attributes	Data Type	Size	Required	Allow Zero Length
<code>device name</code>	Text	255	True	False
<code>server</code>	Text	255	True	False
<code>type</code>	Text	255	False	True
<code>address</code>	Text	255	False	True
<code>can add items</code>	Bool	2 bytes	True	False
<code>reconfigurable</code>	Bool	2 bytes	True	False
<code>rate</code>	Double	8 bytes	True	False
<code>comm resource</code>	Text	255	True	False

`device name`—The unique name identifying the device.

`server`—The server associated with this device.

`type`—The type of device: PLC, Intelligent Instrument, DAQ board, and so on.

`address`—The protocol specific communications path to the device.

`can add items`—A Boolean indicating whether you can add items dynamically to the items table. TRUE indicates yes and FALSE indicates no.

`reconfigurable`—TRUE if the field values for the device can be configured during run time.

`default rate`—The default sampling rate for advises on the device in milliseconds.

`comm resource`—The communications resource associated with this device.

`specific info`—A binary field for developer-specific information.

- **Primary Keys**—The primary key consists of the `device name` and `server` fields.
- **Foreign Keys**—The foreign key is the `server` field, related to the `server name` of the `servers` table.



Note: *There is enforced referential integrity between the `devices` table and the `servers` table, as well as cascading updates and deletes that affect the `items` table.*

The Items Table

The `items` table describes the attributes of an item for both the IAIO servers and BridgeVIEW.

Table 9-6. The Items Table Definition

Attributes	Data Type	Size	Required	Allow Zero Length
item name	Text	255	True	False
device	Text	255	True	False
server	Text	255	True	False
native data type	Integer	2 bytes	True	False
client data type	Text	255	False	True
address	Text	255	False	True

Table 9-6. The Items Table Definition (Continued)

Attributes	Data Type	Size	Required	Allow Zero Length
reconfigurable	Bool	2 bytes	True	False
on data change	Bool	2 bytes	True	False
rate	Double	8 bytes	False	False
count	Long	4 bytes	False	False
access rights	Integer	2 bytes	True	False
max range	Double	8 bytes	False	False
min range	Double	8 bytes	False	False
max length	Long	4 bytes	False	False
unit	Text	255	False	True
specific info	Bin	1.2 Gig	False	False

`item name`—The unique name identifying the item.

`device`—The device associated with the item.

`server`—The server associated with the item.

`native data type`—The data type of the item, as found on the device. (Refer to the list of IAIO data types for valid field values.)

`client data type`—A BridgeVIEW field representing the final representation of the requested item. (Refer to the list of IAIO and BV data types for valid field values.)

`address`—The complete path to the item protocol-specific string that identifies the physical location of the item, for example, 4:0.

`reconfigurable`—TRUE if the field values for the item can be configured during run time.

on data change—A Boolean indicating whether advise data is posted after every advise or only when the data has changed from the previously acquired item value. TRUE if posted on data change only.

rate—The sampling rate in milliseconds for advises on the item.

count—The number of items the server retrieves from the item address. This field is for block read, write, and advises.

access rights—A numeric value representing the allowable access for the item. Input only (0), output only (1), or input and output (2).

max range—The maximum value in engineering units allowable for this item.

min range—The minimum value in engineering units allowable for this item.

max length—The max length field represents the number of bytes required for a given item. When the `NativeDataType` item is a bit array or bit value the max length field represents the number of bits in the item.

unit—The engineering unit for this item.

specific info—A binary field for developer-specific information.

- Primary Keys—The primary key consists of the `item name`, `device` and `server` fields.
- Foreign Keys—The foreign keys are the `server` field, related to the `server` field of the `devices` table and the `device` field related to the `device name` of the `devices` table.



Note: *There is enforced referential integrity between the `items` table and the `devices` table.*

The Serial Table

The `serial` table describes the common attributes of a serial port.

Table 9-7. The Serial Table Definition

Attributes	Data Type	Size	Required	Allow Zero Length
resource name	Text	255	True	False
port	Text	255	True	False
baud rate	Text	255	True	False
parity	Long	4 bytes	True	False
data bits	Long	4 bytes	True	False
stop bits	Long	4 bytes	True	False
read interval	Long	4 bytes	True	False

`resource name`—The unique name identifying the serial resource.

`port`—The string identifying the physical serial resource, for example, COM1.

`baud rate`—The baud rate at which the port operates, such as 200, 600, 38400, and so on.

`parity`—The line parity for the communications resource.
No parity (0), odd parity (1), even parity (2), mark parity (3), space parity (4).

`data bits`—The number of data bits.

`stop bits`—The number of stop bits for the communications session.
One stop bit (0), 1.5 stop bits (1), two stop bits (2).

`read interval`—Specifies the maximum time, in milliseconds, allowed to elapse between the arrival of two characters on the communications line. A value of 0 indicates that no time out is used.

- **Primary Keys**—The primary key consists of the `resource name` field.
- **Foreign Keys**—None.

The Generic Resources Table

The `generic resource` table serves as the catch-all for communications resource information.

Table 9-8. The Generic Resource Table Definition

Attributes	Data Type	Size	Required	Allow Zero Length
<code>resource name</code>	Text	255	False	False
<code>type</code>	Text	255	False	False
<code>specific info</code>	Blob	1.2 Gig	False	False

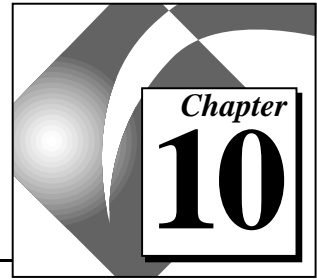
`resource name`—The unique name identifying the serial resource.

`type`—The type of the generic resource. More often than not, the `type` field is the name of a server.

`specific info`—A binary field for developer-specific information.

- **Primary Keys**—The primary key consists of the `resource name` and the `type` fields.
- **Foreign Keys**—None.

IAIO Servers Automation Reference



This chapter describes the OLE automation interfaces used to access and configure for the common configuration databases.

The Servers Automation Interface

The CCDB uses an OLE automation interface to provide programmatic access to CCDB files. The rest of this chapter describes the exposed automation interfaces of the CCDB server.



Note: *All methods and get and/or set property procedures use structured exception handling, so any calls made to the interface should be guarded.*

The IManager Interface

The IManager interface manages client sessions with the CCDB. It also has several utility methods for database maintenance.

ConnectDB

```
Declare Sub ConnectDB (  
    ByVal DBPath As String,  
    ByRef Status As Variant)
```

Purpose

This method attempts to connect to the database whose name and path is specified by the DBPath parameter.

- If the DBPath database does not exist, it is created and the value of status is set to TRUE.
- If the DBPath database exists and is opened successfully, the Status parameter is set to FALSE. DBPath might be an empty or NULL string.
- If DBPath is an empty string, the IManager automation interface attempts to connect to a currently open CCDB file.

- If there is no open database when a call to `ConnectDB` is made with an empty `DBPath`, the procedure throws an exception.
- If multiple calls to `ConnectDB` are made, the reference count is incremented. `ConnectDB` maintains a reference count on the currently open database.

When you call `DisconnectDB`, it decrements the reference count. When the reference count reaches zero, `DisconnectDB` closes the database.



Note: *You cannot connect to more than one database at a time. If the reference count of a database is greater than zero, all calls to `ConnectDB` with a `DBPath` other than the currently open database cause an exception.*

DisconnectDB

```
Declare Function DisconnectDB () As Boolean
```

Purpose

This method decrements the database reference count. If the count is zero, it closes the `CCDB` and calls the method `CompactDB` on the `CCDB` file that is closed. If the procedure fails, the return value is `FALSE`. A successful call returns `TRUE`.

CompactDB

```
Declare Sub CompactDB (ByVal DBPath As String)
```

Purpose

When you delete a row or table from databases in the Microsoft Database (MDB) format, the memory page associated with that row or table is marked as invalid. The page does not disappear and, when the data is written to disk, the invalid pages are written as well. A call to `CompactDB` opens an MDB format database and writes it back out to disk without invalid pages.



Note: *You should use this method sparingly because of the long time required for this call.*

RepairDB

```
Declare Sub RepairDB (ByVal DBPath As String)
```

Purpose

In the event of a sudden system shutdown or some catastrophic failure, a MDB format database may become corrupt. To repair such a database, invoke the `RepairDB` method.

DeleteRow

```
Declare Sub DeleteRow (
    ByVal tableName As String,
    ByVal Param1 As Variant,
    Optional ByVal Param2 As Variant,
    Optional ByVal Param3 As Variant)
```

Purpose

This method removes a row from the table specified in `tableName`. This method is incapable of deleting rows from a table with a complex index containing more than three fields.

BeginTransactions

```
Declare Sub BeginTransactions ()
```

Purpose

This method denotes the beginning of a guarded two-phase database transaction.



Caution: *Do not place* `BeginTransactions()` *around the following methods:*

```
IIAIODevices::Device, IIAIOItems::Item,
IIAIOSerial::Resource, IIAIOGenericResources::Resource,
IIAIOServers::Server, or IIAIOProxy::Proxy
```

Doing so causes the corruption of internal DAO data members.

CommitTransactions

```
Declare Sub CommitTransactions ()
```

Purpose

This method commits guarded transactions that occurred since the last call to `BeginTransactions()`.

RollbackTransactions

```
Declare Sub RollbackTransactions ()
```

Purpose

This method throws out database transactions that occurred since the last call to `BeginTransactions()`.

GetCurrentVersion

```
Declare Function GetCurrentVersion () As String
```

Purpose

This method returns the current version of the Servers DLL.

Query

```
Declare Function Query (
    ByVal SQLQuery As String,
    ByVal TableType As Integer) As PTR
```

Purpose

Perform an SQL (structured query language) query on the currently open CCDB. The `Query` function returns a DAO interface of the table type specified by the `TableType` parameter.

Possible TableType values

`dbOpenSnapshot`

`dbOpenRecordset`

`dbOpenTable`

RetrieveTable

```
Declare Function RetrieveTable (
    ByVal tableName As String,
    ByVal TableType As Integer) As PTR
```

Purpose

This method retrieves a DAO interface to the table specified by `TableName` with the table type denoted by `TableType`.

Possible TableType values

dbOpenSnapshot

dbOpenRecordset

dbOpenTable

The IIAIOProxy Interface

The IIAIOProxy interface provides a simple row marker/attribute based model for creating, editing, and deleting rows in the proxies table of the CCDB.

Proxy

```
Declare Sub Proxy (
    ByVal Mode As Boolean,
    ByVal ProxyName String)
```

Purpose

The `proxy` method of the IIAIOProxy interface performs one of two operations depending on the value of the `Mode` parameter. If the `Mode` parameter is `TRUE`, the `proxy` method attempts to create a new entry in the proxies table with a `proxy` name of `proxyName`. When this occurs, the IIAIOProxy object sets an internal row reference to the new row so that all following calls to get and set proxy row attributes refer to the newly created row entry.

If the parameter `Mode` is `FALSE`, the method attempts to locate the row whose `proxy` name is `ProxyName`. If the method cannot find such a row, the `proxy` method throws an exception. When the method is successful the internal row reference is set to the specified row so that all following calls to get and set proxy row attributes refer to the specified row entry.

Delete

```
Declare Sub Delete (ByVal ProxyName As String)
```

Purpose

This method searches the proxies table for a row whose `proxy` name field value corresponds to the `ProxyName` parameter. If the method locates such a row, it removes the row from the proxies table.

Name

```
Declare Function Name () As String
```

```
Declare Sub Name (ByVal String)
```

Purpose

These two methods get and set the contents of the `proxy name` field for the current row reference for the IIAIOPProxy object.

ProxyType

```
Declare Function ProxyType () As String
```

```
Declare Sub ProxyType (ByVal String)
```

Purpose

These two methods get and set the contents of the `type` field for the current row reference for the IIAIOPProxy object.

ConfigPath

```
Declare Function ConfigPath () As String
```

```
Declare Sub ConfigPath (ByVal String)
```

Purpose

These two methods get and set the contents of the `configuration path` field for the current row reference for the IIAIOPProxy object.

LaunchPath

```
Declare Function LaunchPath () As String
```

```
Declare Sub LaunchPath (ByVal String)
```

Purpose

These two methods get and set the contents of the `launch path` field for the current row reference for the IIAIOPProxy object.

The IAIOServers Interface

The IAIOServers interface implements a simple row marker/attribute-based model for creating, editing, and deleting rows in the `servers` table of the CCDB.

Server

```
Declare Sub Server (
    ByVal Mode As Boolean,
    ByVal String)
```

Purpose

The `Server` method of the IAIOServers interface performs one of two operations depending on the value of the `Mode` parameter. If the `Mode` parameter is `TRUE`, the `Server` method attempts to create a new entry in the `server` table with a `server name` of `ServerName`. When this occurs, the IAIOServers object sets an internal row reference to the new row so that all following calls to `get` and `set` server row attributes refer to the newly created row entry.

If the parameter `Mode` is `FALSE`, the method attempts to locate the row whose `server name` is `ServerName`. If the `Server` method does not find such a row, it throws an exception. When the method is successful, the internal row reference is set to the specified row. Then, all following calls to `get` and `set` server row attributes refer to the specified row entry.

Delete

```
Declare Sub Delete (ByVal ServerName As String)
```

Purpose

This method searches the `servers` table for a row whose `server name` field value corresponds to the `ServerName` parameter. If the method locates such a row, the method removes the row from the `servers` table. The devices associated with the removed server are deleted from the `devices` table, and all of the items associated with those devices are deleted as well.

Name

```
Declare Function Name () As String
```

```
Declare Sub Name (ByVal String)
```

Purpose

These two methods get and set the contents of the `server name` field for the current row reference for the `IIAIOServers` object.

CanAddDevices

```
Declare Function CanAddDevices () As Boolean
```

```
Declare Sub CanAddDevices (ByVal Boolean)
```

Purpose

These two methods get and set the contents of the `can add devices` field for the current row reference for the `IIAIOServers` object.

ServerType

```
Declare Function ServerType () As String
```

```
Declare Sub ServerType (ByVal String)
```

Purpose

These two methods get and set the contents of the `type` field for the current row reference for the `IIAIOServers` object.

LaunchPath

```
Declare Function LaunchPath () As String
```

```
Declare Sub LaunchPath (ByVal String)
```

Purpose

These two methods get and set the contents of the `launch path` field for the current row reference for the `IIAIOServers` object.

ConfigPath

```
Declare Function ConfigPath () As String
```

```
Declare Sub ConfigPath (ByVal String)
```

Purpose

These two methods get and set the contents of the `configuration path` field for the current row reference for the `IIAIOServers` object.

SpecificInfo

```
Declare Function SpecificInfo () As Variant
```

```
Declare Sub SpecificInfo (ByVal Variant)
```

Purpose

These two methods get and set the contents of the `specific info` field for the current row reference for the `IIAIOServers` object.

The IIAIODevices Interface

The `IIAIODevices` interface has a simple row marker/attribute-based model for creating, editing, and deleting rows in the `devices` table of the `CCDB`.

Device

```
Declare Sub Device (
    ByVal Mode As Boolean,
    ByVal ServerName As String,
    ByVal DeviceName String)
```

Purpose

The `Device` method of the `IIAIODevices` interface performs one of two operations, depending on the value of the `Mode` parameter. If the `Mode` parameter is `TRUE`, the `Device` method attempts to create a new entry in the `devices` table with a device name of `DeviceName` and the server field set to `ServerName`. When this occurs, the `IIAIODevices` object sets an internal row reference to the new row so that all following calls to get and set device row attributes refer to the newly created row entry.

If the parameter `Mode` is `FALSE`, the method attempts to locate the row whose device name is `DeviceName` and whose server entry corresponds to `ServerName`. If the method

finds no such row, the method throws an exception. When the method is successful, the internal record reference is set to the specified row so that all following calls to get and set device row attributes refer to the specified row entry.

Delete

```
Declare Sub Delete (
    ByVal ServerName As String,
    ByVal DeviceName As String)
```

Purpose

This method searches the `devices` table for a row whose `server` and `device name` field values correspond to the `ServerName` and `DeviceName` parameters. If the method finds such a row, the method removes it from the `devices` table and deletes all of the items associated with the removed device from the `items` table.

DeviceName

```
Declare Function DeviceName () As String
Declare Sub DeviceName (ByVal String)
```

Purpose

These two methods get and set the contents of the `device name` field for the current row reference for the `IIAIODevices` object.

ServerName

```
Declare Function ServerName () As String
Declare Sub ServerName (ByVal String)
```

Purpose

These two methods get and set the contents of the `server` field for the current row reference for the `IIAIODevices` object.

DeviceType

```
Declare Function DeviceType () As String
Declare Sub DeviceType (ByVal String)
```

Purpose

These two methods get and set the contents of the `type` field for the current row reference for the `IIAIODevices` object.

Address

```
Declare Function Address () As String
Declare Sub Address (ByVal String)
```

Purpose

These two methods get and set the contents of the `address` field for the current row reference for the `IIAIODevices` object.

CanAddItems

```
Declare Function CanAddItems () As Boolean
Declare Sub CanAddItems (ByVal Boolean)
```

Purpose

These two methods get and set the contents of the `can add items` field for the current row reference for the `IIAIODevices` object.

DConfig

```
Declare Function DConfig () As Boolean
Declare Sub DConfig (ByVal Boolean)
```

Purpose

These two methods get and set the contents of the `configurable` field for the current row reference for the `IIAIODevices` object.

DefaultRate

```
Declare Function DefaultRate () As Double
Declare Sub DefaultRate (ByVal Double)
```

Purpose

These two methods get and set the contents of the `rate` field for the current row reference for the `IIAIODevices` object.

DeviceResource

```
Declare Function DeviceResource () As String
```

```
Declare Sub DeviceResource (ByVal String)
```

Purpose

These two methods get and set the contents of the `comm resource` field for the current row reference for the `IIAIODevices` object.

SpecificInfo

```
Declare Function SpecificInfo () As Variant
```

```
Declare Sub SpecificInfo (ByVal Variant)
```

Purpose

These two methods get and set the contents of the `specific info` field for the current row reference for the `IIAIODevices` object.

The IIAIOItems Interface

The `IIAIOItems` interface contains a simple row marker/attribute-based model for creating, editing, and deleting rows in the `items` table of the `CCDB`.

Item

```
Declare Sub Item (
    ByVal Mode As Boolean,
    ByVal ServerName As String,
    ByVal DeviceName As String,
    ByVal ItemName String)
```

Purpose

The `Item` method of the `IIAIOItems` interface performs one of two operations depending on the value of the `Mode` parameter. If the `Mode` parameter is `TRUE`, the `Item` method attempts to create a new entry in the `items` table with an `item name` of `ItemName` for the device, `DeviceName` and the server, `ServerName`. When this occurs, the `IIAIOItems` object sets an internal row reference to the new row so that all following calls to get and set item row attributes refer to the newly created row entry.

If the parameter `Mode` is `FALSE`, the method attempts to locate the row whose `item name` is `ItemName` and whose `server` and `device` entries correspond to `ServerName` and `DeviceName`. If it does not find such a row, the method throws an exception. When the method finds such a row, the function sets the internal record reference to the specified row so that all following calls to `get` and `set` item row attributes refer to the specified row entry.

Delete

```
Declare Sub Delete (
    ByVal ServerName As String,
    ByVal DeviceName As String,
    ByVal ItemName As String)
```

Purpose

This method searches the `items` table for a row whose `server`, `device`, and `item name` field values correspond to the `ServerName`, `DeviceName`, and `ItemName` parameters. If the method locates such a row, the method removes the row from the `items` table. If the method does not find the row, it throws an exception.

ItemName

```
Declare Function ItemName () As String
Declare Sub ItemName (ByVal String)
```

Purpose

These two methods get and set the contents of the `item name` field for the current row reference for the `IIAIOItems` object.

DeviceName

```
Declare Function DeviceName () As String
Declare Sub DeviceName (ByVal String)
```

Purpose

These two methods get and set the contents of the `device` field for the current row reference for the `IIAIOItems` object.

ServerName

```
Declare Function ServerName () As String
```

```
Declare Sub ServerName (ByVal String)
```

Purpose

These two methods get and set the contents of the `server` field for the current row reference for the `IIAIOItems` object.

NativeDataType

```
Declare Function NativeDataType () As String
```

```
Declare Sub NativeDataType (ByVal String)
```

Purpose

These two methods get and set the contents of the `native data type` field for the current row reference for the `IIAIOItems` object.

Address

```
Declare Function Address () As String
```

```
Declare Sub Address (ByVal String)
```

Purpose

These two methods get and set the contents of the `address` field for the current row reference for the `IIAIOItems` object.

Configurable

```
Declare Function Configurable () As Boolean
```

```
Declare Sub Configurable (ByVal Boolean)
```

Purpose

These two methods get and set the contents of the `configurable` field for the current row reference for the `IIAIOItems` object.

OnDataChange

```
Declare Function OnDataChange () As Boolean
```

```
Declare Sub OnDataChange (ByVal Boolean)
```

Purpose

These two methods get and set the contents of the `on data change` field for the current row reference for the `IIAIOItems` object.

DefaultRate

```
Declare Function DefaultRate () As Variant
```

```
Declare Sub DefaultRate (ByVal Variant)
```

Purpose

These two methods get and set the contents of the `rate` field for the current row reference for the `IIAIOItems` object.

ItemCount

```
Declare Function ItemCount () As Variant
```

```
Declare Sub ItemCount (ByVal Variant)
```

Purpose

These two methods get and set the contents of the `count` field for the current row reference for the `IIAIOItems` object.

AccessRights

```
Declare Function AccessRights () As Integer
```

```
Declare Sub AccessRights (ByVal Integer)
```

Purpose

These two methods get and set the contents of the `access rights` field for the current row reference for the `IIAIOItems` object.

MaxRange

```
Declare Function MaxRange () As Variant
```

```
Declare Sub MaxRange (ByVal Variant)
```

Purpose

These two methods get and set the contents of the `max_range` field for the current row reference for the `IIAIOItems` object.

MinRange

```
Declare Function MinRange () As Variant
```

```
Declare Sub MinRange (ByVal Variant)
```

Purpose

These two methods get and set the contents of the `min_range` field for the current row reference for the `IIAIOItems` object.

MaxLength

```
Declare Function MaxLength () As Variant
```

```
Declare Sub MaxLength (ByVal Variant)
```

Purpose

These two methods get and set the contents of the `max_length` field for the current row reference for the `IIAIOItems` object.

Unit

```
Declare Function Unit () As Variant
```

```
Declare Sub Unit (ByVal Variant)
```

Purpose

These two methods get and set the contents of the `unit` field for the current row reference for the `IIAIOItems` object.

SpecificInfo

```
Declare Function SpecificInfo () As Variant
```

```
Declare Sub SpecificInfo (ByVal Variant)
```

Purpose

These two methods get and set the contents of the `specific info` field for the current row reference for the `IIAIOItems` object.

The ISerial Interface

The `ISerial` interface provides a simple row marker/attribute-based model for creating, editing and deleting rows in the `serial` table of the `CCDB`.

Resource

```
Declare Sub Resource (
    ByVal Mode As Boolean,
    ByVal ResourceName String)
```

Purpose

The `Resource` method of the `ISerial` interface performs one of two operations depending on the value of the `Mode` parameter. If the `Mode` parameter is `TRUE`, the `Resource` method attempts to create a new entry in the `serial` table with a `resource` name of `ResourceName`. When this occurs, the `ISerial` object sets an internal row reference to the new row so that all following calls to get and set serial row attributes refer to the newly created row entry.

If the parameter `Mode` is `FALSE`, the method attempts to locate the row whose `resource` name is `ResourceName`. If no such row is found, the method throws an exception. When the method is successful, the internal row reference is set to the specified row so that all following calls to get and set serial row attributes refer to the specified row entry.

Delete

```
Declare Sub Delete (ByVal ResourceName As String)
```

Purpose

This method searches the `serial` table for a row whose `resource name` field values correspond to the `ResourceName` parameter. If the method finds such a row, the method removes the row from the `serial` table. If the method does not find the row, it throws an exception.

ResourceName

```
Declare Function ResourceName () As String
```

```
Declare Sub ResourceName (ByVal String)
```

Purpose

These two methods get and set the contents of the `resource name` field for the current row reference for the `ISerial` object.

Port

```
Declare Function Port () As String
```

```
Declare Sub Port (ByVal String)
```

Purpose

These two methods get and set the contents of the `port` field for the current row reference for the `ISerial` object.

ReadTimeoutInterval

```
Declare Function ReadTimeoutInterval () As Long
```

```
Declare Sub ReadTimeoutInterval (ByVal Long)
```

Purpose

These two methods get and set the contents of the `read interval` field for the current row reference for the `ISerial` object.

StopBits

```
Declare Function StopBits () As Long
```

```
Declare Sub StopBits (ByVal Long)
```

Purpose

These two methods get and set the contents of the `stop bits` field for the current row reference for the `ISerial` object.

DataBits

```
Declare Function DataBits () As Long
```

```
Declare Sub DataBits (ByVal Long)
```

Purpose

These two methods get and set the contents of the `data bits` field for the current row reference for the `ISerial` object.

BaudRate

```
Declare Function BaudRate () As Long
```

```
Declare Sub BaudRate (ByVal Long)
```

Purpose

These two methods get and set the contents of the `baud rate` field for the current row reference for the `ISerial` object.

Parity

```
Declare Function Parity () As Long
```

```
Declare Sub Parity (ByVal Long)
```

Purpose

These two methods get and set the contents of the `parity` field for the current row reference for the `ISerial` object.

The IGenericResource Interface

The IGenericResource interface contains a simple row marker/attribute based model for creating, editing, and deleting rows in the generic resources table of the CCDB.

Resource

```
Declare Sub Resource (
    ByVal Mode As Boolean,
    ByVal ResourceName As String,
    ByVal type As String)
```

Purpose

The Resource method of the IGenericResource interface performs one of two operations depending on the value of the Mode parameter. If the Mode parameter is TRUE, the Resource method attempts to create a new entry in the generic resources table with a resource name of ResourceName. When this occurs, the IGenericResource object sets an internal row reference to the new row so that all following calls to get and set generic resource row attributes refer to the newly created row entry.

If the parameter Mode is FALSE, the method attempts to locate the row whose resource name is ResourceName. If the method finds no such row, the method throws an exception. When the method is successful, the internal record reference is set to the specified row so that all following calls to get and set generic resource row attributes refer to the specified row entry.

Delete

```
Declare Sub Delete (ByVal ResourceName As String)
```

Purpose

This method searches the generic resources table for a row whose resource name field values correspond to the ResourceName, parameter. If the method locates such a row, the method removes it from the generic resources table.

ResourceName

```
Declare Function ResourceName () As String
```

```
Declare Sub ResourceName (ByVal String)
```

Purpose

These two methods get and set the contents of the `resource name` field for the current row reference for the `IGenericResource` object.

ResourceType

```
Declare Function ResourceType () As String
```

```
Declare Sub ResourceType (ByVal String)
```

Purpose

These two methods get and set the contents of the `type` field for the current row reference for the `IGenericResource` object.

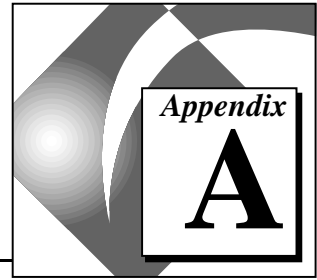
SpecificInfo

```
Declare Function SpecificInfo () As Variant
```

```
Declare Sub SpecificInfo (ByVal Variant)
```

Purpose

These two methods get and set the contents of the `specific info` field for the current row reference for the `IGenericResource` object.



Data Types and Attributes

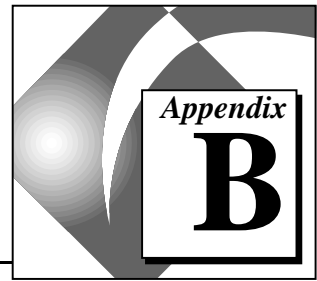
Data Types

Data Type	Description
IAGHandle	Handle to Industrial Automation I/O Server resource.
IAGStatus	Status code data type.
IAGUserCallback	Pointer to user callback function.
IAGType	Data type specification—a subset of the available OLE VARIANT types including VT_I1, VT_I2, VT_I4, VT_I8, VT_UI1, VT_UI2, VT_UI4, VT_UI8, VT_R4, VT_R8, and VT_BOOL.
IAGBoolean	A two byte value where 0 is FALSE and non-0 is TRUE.
IAGByte	An unsigned char.
IAGString	Pointer to a char.
IAGTaskID	A handle to read, write, or advise task.
IAGTimeStamp	A double containing the number of days since midnight December 30, 1899.
IAGAttr	An integer representing an attribute as indicated in the following table.

Attributes

Attribute Name	Access	Description
IA_ATTR_TIMEOUT	R/W	Timeout value for a device or I/O point. Default value is initialized from device configuration.
IA_ATTR_BUFFER_FACTOR	R/W	The buffering factor for an I/O point. If 0, no buffering is in effect. If 1, single buffering is in effect. Any I/O point with an IA_ATTR_BUFFER_FACTOR of less than two automatically is increased to double buffering for any Advise operation.

Diagnostic Error Messages



You can debug errors in the IAIO API by checking for error messages. Diagnostic error messages can be broken down into the following three categories.

- Operational Status—describe any errors in regard to how the IAIO API is running.
- Communications and Device Status—describe errors in network communication and configured devices.
- I/O Point Value Status—indicate if the I/O point is invalid or if it is read-protected or write-protected.

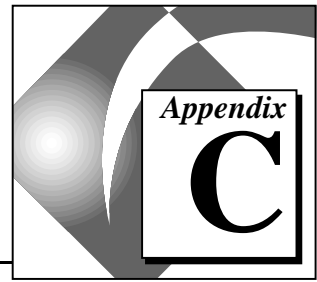
Error Messages

Error Name	Description
IA_SUCCESS	IAIO operation was successful.
IA_TERMINATED	Task is terminated.
IA_PENDING	Task is pending.
IA_ERROR_INVALID_IOPOINT	An invalid I/O point was indicated.
IA_ERROR_INVALID_COMPLETION	An invalid completion was indicated.
IA_ERROR_INVALID_TASK_ID	An invalid task ID was indicated.
IA_ERROR_INVALID_IAHANDLE	An invalid IAHandle was indicated.
IA_ERROR_INVALID_STATUS_CODE	An invalid status code was indicated.
IA_ERROR_UNKNOWN_IATYPE	An unknown IAType was indicated.
IA_ERROR_BUFFER_OVERFLOW	A task buffer overflow occurred.
IA_ERROR_SERVER_NOT_INITIALIZED	The device server has not been initialized.

Error Name	Description
IA_ERROR_SERVER_ALREADY_INITIALIZED	The device server already has been initialized.
IA_ERROR_SERVER_INITIALIZATION_FAILED	The device server initialization has failed.
IA_ERROR_SERVER_SHUTDOWN	The IAIO has shut down.
IA_ERROR_UNKNOWN_TAG	The indicated tag name was not found.
IA_ERROR_IOPOINT_IN_USE	The I/O point is in use.
IA_ERROR_COMPLETION_IN_USE	The completion is in use.
IA_ERROR_TASK_NOT_COMPLETED	Task still is pending.
IA_ERROR_UNKNOWN_ATTRIBUTE	The specific attribute is unknown.
IA_ERROR_INVALID_ATTRIBUTE_FOR_OBJECT	The attribute is not a property of the indicated object.
IA_ERROR_INVALID_IOPOINT_DEFINITION	The I/O point definition is not valid.
IA_ERROR_READONLY_IOPOINT	The I/O point is read-only.
IA_ERROR_WRITEONLY_IOPOINT	The I/O point is write-only.
IA_ERROR_IOPOINT_UNSUPP_TYPE_CNV	The type conversion is not supported for the indicated I/O point.
IA_ERROR_IO_OPERATION_FAILURE	The I/O operation has failed.
IA_ERROR_IOPOINT_TOO_LARGE	The I/O point indicated is too large.
IA_ERROR_TYPE_CNV_FAILURE	The type conversion failed.
IA_ERROR_CCDB_NOT_FOUND	The CCDB server is not configured or was not found.
IA_ERROR_CCDB_INIT_FAILED	The CCDB server initialization failed.
IA_ERROR_CCDB_SERVER_CFG_NOT_FOUND	The CCDB server is not configured or was not found.

Error Name	Description
IA_ERROR_CCDB_DEVICES_NOT_FOUND	There are no configured devices for the server.
IA_ERROR_NOT_IMPLEMENTED	The IAIO operation is not implemented.
IA_ERROR_INTERNAL_ERROR	An unrecoverable device server internal error has occurred.
IA_ERROR_UNKNOWN	An unknown error condition has occurred.
IA_ERROR_TIMEOUT	The timeout period has expired.
IA_ERROR_MSG_PACKET	A message packet error has occurred.
IA_ERROR_NETWORK_FAILURE	The network communication has failed.
IA_ERROR_CONN_BUSY	The network connection is busy.
IA_ERROR_CONN_LOST	The network connection has been lost.
IA_ERROR_CONN_FAILURE	The network connection has failed.
IA_ERROR_DEVICE_INVALID_TIME	The device clock or timer is invalid.
IA_ERROR_DEVICE_FAULT	The device has faulted.
IA_ERROR_DEVICE_OFFLINE	The device is offline.
IA_ERROR_MEMORY_ALLOCATION_FAILURE	Memory allocation has failed.
IA_ERROR_SYSTEM_ERROR	An unrecoverable operating system error has occurred.

Customer Communication



For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services



Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

United States: (512) 794-5422

Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422

Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59

Up to 9,600 baud, 8 data bits, 1 stop bit, no parity



FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as anonymous and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.



Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at (512) 418-1111.



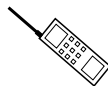
E-Mail Support (currently U.S. only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

support@natinst.com

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.



Telephone



Fax

Australia	02 9874 4100	02 9874 4455
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 527 2321	09 502 2930
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 5734815	03 5734816
Italy	06 5729961	06 57284309
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	31 348 43 34 66	31 348 43 06 73
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

Fax (____) _____ Phone (____) _____

Computer brand _____ Model _____ Processor _____

Operating system (include version number) _____

Clock speed _____ MHz RAM _____ MB Display adapter _____

Mouse ___yes ___no Other adapters installed _____

Hard disk capacity _____ MB Brand _____

Instruments used _____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is: _____

List any error messages: _____

The following steps reproduce the problem: _____

BridgeVIEW Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

DAQ hardware _____

Interrupt level of hardware _____

DMA channels of hardware _____

Base I/O address of hardware _____

Programming choice _____

BridgeVIEW version _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Other Products

Computer make and model _____

Microprocessor _____

Clock frequency or speed _____

Type of video board installed _____

Operating system version _____

Operating system mode _____

Programming language _____

Programming language version _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *BridgeVIEW™ Device Server Toolkit Reference Manual*

Edition Date: March 1997

Part Number: 321298A-01

Please comment on the completeness, clarity, and organization of the manual.

If you find errors in the manual, please record the page numbers and describe the errors.

Thank you for your help.

Name _____

Title _____

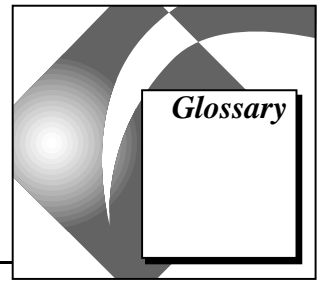
Company _____

Address _____

Phone (____) _____ Fax (____) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678



A

advise	A read of an I/O point at a given rate.
asynchronous	Not synchronized; not controlled by periodic time signals, and therefore unpredictable with regard to the timing of execution of commands.
auxiliary class	A class of objects that exists exclusively to help another class perform a particular function.

C

client	The application that sends or calls messages from the server application in a dynamic data exchange.
communications resource	An instance of a physical pathway used by devices and I/O points.
completion mechanism	A method of returning data to the client.
constructor	The code block that is executed when an instance of the class is created.

D

derived class	In a system that generates or translates code, the class of objects that is derived in the generation or translation.
destructor	The code block that executes when you destroy a class instance.
device	A collection of I/O points.

E

enforced referential integrity The enforcement of a relationship between two tables based on primary and foreign keys.

F

foreign key A field in a table which is based on the primary key of another table.

H

handle An arbitrary value generated by the device server that allows a client to access IAIO resources, such as I/O points and completion mechanisms in a controlled and protected manner.

hook A mechanism in which a function can be created to implement behavior specifically for the device server.

I

I/O point Represents an instance of data.

item A channel or variable in a real-world device that a device server monitors or controls.

P

pointer A data structure that contains an address or other indication of storage location.

primary key A unique row identifier that contains specific information for a specific item, which is found in all databases.

S

server An application that manages communication with a device and returns data to a client through a standard interface.

T

- task An instance of a timed I/O operation (read, write, or advise).
- thread A process that takes place within a larger process or program.